University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Karol Kurach

# Deep Neural Architectures for Algorithms and Sequential Data

*PhD dissertation*

Supervisor
dr hab. Hung Son Nguyen, prof. UW
Institute of Informatics
University of Warsaw

June 2016

Author's declaration:
I hereby declare that this dissertation is my own work.

June 13, 2016 .....

*Karol Kurach*

Supervisor's declaration:
The dissertation is ready to be reviewed.

June 13, 2016 .....

*dr hab. Hung Son Nguyen, prof. UW*

## Abstract

The first part of the dissertation describes two deep neural architectures with external memories: Neural Random-Access Machine (NRAM) and Hierarchical Attentive Memory (HAM). The NRAM architecture is inspired by Neural Turing Machines, but the crucial difference is that it can manipulate and dereference pointers to its random-access memory. This allows it to learn concepts that require pointers chasing, such as "linked list" or "binary tree". The HAM architecture is based on a binary tree with leaves corresponding to memory cells. This enables the memory access in $\Theta(\log n)$, which is a significant improvement over $\Theta(n)$ access used in the standard attention mechanism. We show that Long Short-Term Memory (LSTM) augmented with HAM can successfully learn to solve a number of challenging algorithmic problems. In particular, it is the first architecture that learns from pure input/output examples to sort $n$ numbers in time $\Theta(n \log n)$ and the solution generalizes well to longer sequences. We also show that HAM is very generic and can be trained to act like classic data structures: a stack, a FIFO queue and a priority queue.

The second part of the dissertation describes three novel systems based on deep recurrent neural networks (RNNs). The first one is a framework for finding computationally efficient versions of symbolic math expressions. By using RNNs it can efficiently search the state space and quickly find identities with significantly better time complexity (e.g., $\Theta(n^2)$ instead of exponential time). Then, we present an RNN-based system for predicting dangerous events from multivariate, non-stationary time series data. It requires almost no feature engineering and achieved very good results in two machine learning competitions. Finally, we describe *Smart Reply* – an RNN-based end-to-end system for suggesting automatic responses to e-mails. The system is capable of handling hundreds of millions messages daily. Smart Reply was successfully deployed in *Google Inbox* and currently generates 10% of responses on mobile devices.



## Keywords

Neural Networks, Recurrent Neural Networks, Attention Mechanism, LSTM

## ACM Computing Classification

Machine Learning, Machine Learning approaches, Neural Networks

## Streszczenie

Pierwsza część pracy przedstawia dwie głębokie architektury neuronowe wykorzystujące pamięć zewnętrzną: Neural Random-Access Machine (NRAM) oraz Hierarchical Attentive Memory (HAM). Pomysł na architekturę NRAM jest inspirowany Neuronowymi Maszynami Turinga (NTM). NRAM, w przeciwieństwie do NTM, posiada mechanizmy umożliwiające wykorzystanie wskaźników do pamięci. To sprawia, że NRAM jest w stanie nauczyć się pojęć wymagających użycia wskaźników, takich jak „lista jednokierunkowa" albo „drzewo binarne". Architektura HAM bazuje na pełnym drzewie binarnym, w którym liście odpowiadają elementom pamięci. Umożliwia to wykonywanie operacji na pamięci w czasie $\Theta(\log n)$, co jest znaczącą poprawą względem dostępu w czasie $\Theta(n)$, standardowo używanym w implementacji mechanizmu „skupienia uwagi" (ang. attention) w sieciach rekurencyjnych. Pokazujemy, że sieć LSTM połączona z HAM jest w stanie rozwiązać wymagające zadania o charakterze algorytmicznym. W szczególności, jest to pierwsza architektura, która mając dane jedynie pary wejście/poprawne wyjście potrafi się nauczyć sortowania elementów działającego w złożoności $\Theta(n \log n)$ i dobrze generalizującego się do dłuższych ciągów. Pokazujemy również, że HAM jest ogólną architekturą, która może zostać wytrenowana aby działała jak standardowe struktury danych, takie jak stos, kolejka lub kolejka priorytetowa.

Druga część pracy przedstawia trzy nowatorskie systemy bazujące na rekurencyjnych sieciach neuronowych (RNN). Pierwszy z nich to system do znajdowania wydajnych obliczeniowo formuł matematycznych. Przy wykorzystaniu sieci rekurencyjnej system jest w stanie efektywnie przeszukiwać przestrzeń stanów i szybko znajdować tożsame formuły o istotnie lepszej złożoności asymptotycznej (przykładowo, $\Theta(n^2)$ zamiast złożoności wykładniczej). Następnie, prezentujemy oparty na RNN system do przewidywania niebezpiecznych zdarzeń z wielowymiarowych, niestacjonarnych szeregów czasowych. Nasza metoda osiągnęła bardzo dobre wyniki w dwóch konkursach uczenia maszynowego. Jako ostatni opisany został *Smart Reply* – bazujący na RNN system do sugerowania automatycznych odpowiedzi na e-maile. Smart Reply został zaimplementowany w *Google Inbox* i codziennie przetwarza setki milionów wiadomości. Aktualnie, 10% wiadomości wysłanych z urządzeń mobilnych jest generowana przez ten system.



## Słowa kluczowe

sieci neuronowe, rekurencyjne sieci neuronowe, mechanizm skupienia uwagi, LSTM

## Klasyfikacja tematyczna ACM

Machine Learning, Machine Learning approaches, Neural Networks

*To Anna,*
*my beloved wife*

# Deep Neural Architectures for Algorithms and Sequential Data

*PhD dissertation summary*

Karol Kurach

June 2016

## 1 Introduction

My PhD thesis consists of 7 publications listed below[1]:

1. **Neural Random-Access Machines**, accepted to ICLR 2016 (Conference Track) [1]
2. **Learning Efficient Algorithms with Hierarchical Attentive Memory**, in review for NIPS 2016 [2]
3. **Adding Gradient Noise Improves Learning for Very Deep Networks**, accepted to ICLR 2016 (Workshop Track) [3]
4. **Learning to Discover Efficient Mathematical Identities**, accepted to NIPS 2014 (as **spotlight**) [4]
5. **Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks**, accepted to IJCRS 2015 [5]
6. **Detecting Dangerous Seismic Events with Recurrent Neural Networks**, accepted to AAIA 2016 [6]
7. **Smart Reply: Automated Response Suggestion for Email**, accepted to KDD 2016 (Research Track) with **presentation** at the plenary session [7]

[1]For statistics and acceptance rates please see Appendix A.

The first three publications are related to deep neural architectures with external memories. The next four publications describe novel systems based on recurrent neural networks (both with "vanilla" and Long Short-Term Memory cells) in domains of sequential data and finding efficient algorithms.

A brief introduction to Deep Learning is given in Section 2. Feedforward neural networks are described in Section 3. A generalization of feedforward networks to recurrent neural networks is presented in Section 4. Section 5 describes recently introduced attention mechanism. Finally, Section 6 contains the results of this thesis.

# Notation

| | |
|---|---|
| $x$ | A vector or scalar |
| $x_t$ | A vector or scalar at timestep $t$ (for recurrent neural networks) |
| $x^i$ | The $i$-th element from a set |
| $X$ | A matrix |
| $x \cdot y$ | A scalar product of two vectors |
| $x \odot y$ | An element-wise multiplication of two vectors |
| $x \oplus y$ | An element-wise sum of two vectors |
| $\theta$ | A vector of neural network's trainable parameters ("weights") |
| $a(t)$ | An activation function |
| $f \circ g$ | A composition of functions $f$ and $g$ |
| $\mathbb{Z}_M$ | A ring of integers modulo $M$ |
| $\mathrm{sigm}(\beta, \mathrm{t})$ | The sigmoid function $\frac{1}{1+e^{-\beta t}}$ |
| $\mathrm{sigm}(\mathrm{t})$ | The sigmoid function with $\beta = 1$ |
| $\tanh(\beta, \mathrm{t})$ | The tanh function $\frac{e^{\beta t}-e^{-\beta t}}{e^{\beta t}+e^{-\beta t}}$ |
| $\tanh(\mathrm{t})$ | The tanh function with $\beta = 1$ |

# 2 Deep Learning

Machine Learning is an area of Artificial Intelligence focused on designing systems that can learn from data. It is widely applied to a variety of problems, especially those too complex for a human software engineer to define in terms of a fixed piece of software. Examples span wide range of topics, including computer vision, signals recognition and text understanding.

**Deep learning** is a branch of machine learning that concentrates on finding hierarchical representation of data, starting from low-level observations towards high-level abstractions. Deep architectures are usually composed of multiple layers of non-linear operations, such as neural networks with many hidden layers.

Although neural networks are not new, they recently got a lot of traction in the researchers' community. This is mostly because of three factors: progress in understanding training, availability of big datasets and significant increase in computing power. The first one is related to the fact that in the late 90's it was believed that training networks with more than few layers is a very complex and practically impossible task. Recent work proved that, with correct methods for initialization, careful selection of learning rates and a pre-training, even networks with many layers can be successfully trained.

The other crucial component was progress in hardware, especially the introduction of graphics processing units (GPUs). Being able to train few orders of magnitude faster and on much larger datasets allowed to create and train complex models, with millions of parameters. Deep neural networks already proved to be very successful in representing complicated functions. In some domains, like image or speech recognition, they overperformed previous state of the art methods by a wide margin.

# 3 Feedforward Neural Networks

A *feedforward neural network* (FNN) is a basic example of an artificial neural network. In essence, the goal of such network in supervised setting is to approximate some function $f(x)$, given only a noisy set of evaluations in selected points $\{(x^i, f(x^i))\}$. The FNN can be interpreted as a function $f^*(x;\theta)$, where $\theta$ is a set of trainable parameters ("weights") of the network. The parameters $\theta$ are *trained* from the data to minimize selected *cost* (e.g., MSE, logistic loss) between $f$ and $f^*$.

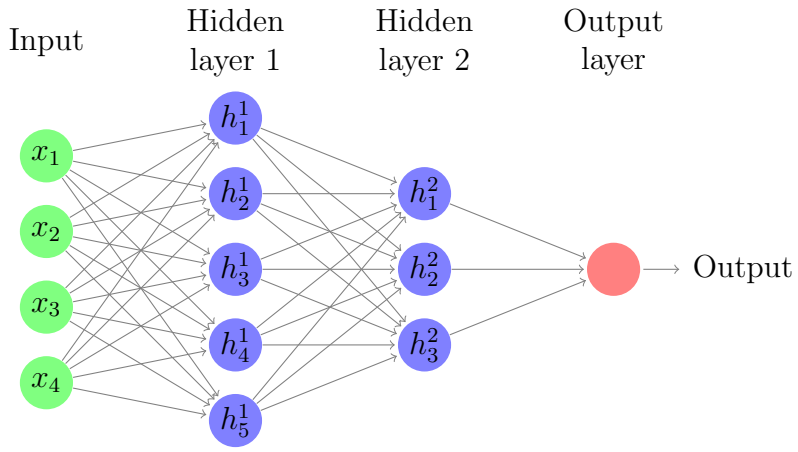



Figure 1: A feedforward neural network with 4 inputs and 2 hidden layers with 5 and 3 nodes respectively.

The simplest form of the feedforward network is a linear classifier, known also as perceptron. In this case:

$$f^*(x;\theta) = a(x \cdot \theta + b) \tag{1}$$

where $b$ is the bias term and $a(u) = [u > 0]$. In a more general scenario, $f^*(x;\theta)$ is a composition of $N$ functions $H^1, \ldots, H^N$. That is:

$$f^*(x) = H^N \circ H^{N-1} \circ \ldots \circ H^1(x) \tag{2}$$

where $H^N$ computes the final output. Intuitively, each function $H^i$ represents one transformation or *layer* of the neural network. The total number of layers $N$ is the depth of the neural network. This is why FNNs are also referred to as multi-layer perceptrons (MLP). The FNN is *feedforward* because connections between nodes of the network do not form cycles, i.e. it can be represented as Direct Acyclic Graph (DAG).

An example feedforward network composed of $N = 3$ layers is presented in Figure 1. The network consists of two hidden layers and an output layer. The parameters $\theta$ are the weights on the edges. Let us denote by $\theta^l_{uv}$ the weight on the edge going from the node $u$ in the layer $l$ to the node $v$ in the layer $l + 1$. Then, each node's value $h^{l+1}_j$ can be computed using:

$$h^{l+1}_j = a^{l+1}(\sum_i h^l_i * \theta^l_{ij} + b^{l+1}_j) \tag{3}$$

The term $a^{l+1}$ is known as *activation function*. It is applied to introduce a non-linearity to the network computation. This is a crucial component –

if all of the hidden layers were linear transformations, the composition of $N$ hidden layers would be a linear transformation as well. A very deep network would be then an equivalent of a simple perceptron. Popular examples of activation functions are sigmoid, tanh and ReLu [8]. The $b_j^{l+1}$ is the bias term for the $j$th node in the layer $l+1$, which is also a trainable parameter of the model. Usually, it is an edge from a special node $h_0^l = 1$ in the layer $l$, so that $b_j^{l+1} = \theta_{0j}^l$.

The estimation of the parameters $\theta$ is frequently done using the *backpropagation* learning technique. For a given example pair $(x, y)$ first the value $y' = f^*(x; \theta)$ is computed. This is used to calculate the $C = loss(y', y)$ value, where $C$ is the "cost" we want to minimize and $loss$ is a function such as Mean Squared Error. Then, the backpropagation is used to compute derivatives of the network parameters $\theta$ w.r.t. the cost. With derivatives computed, one can use a standard optimization technique such as Stochastic Gradient Descent or Adagrad [9] to adjust weights. After a sufficient number of steps the network should usually converge to the parameters that achieve low error score on a given training set.

# 4 Recurrent Neural Networks

The important limitation of feedforward networks is the ability to process only inputs of a fixed size. However, many real-word problems are sequential in nature – for instance, we may want to classify e-mails which can have different number of words, translate sentences or predict stock price based on the list of historical values. Moreover, for the FNN all input examples are independent from each other. This means that, after processing one data point, the network forgets everything from the previous step and starts from scratch. This is clearly very different from the way people process data, relying on the context of elements seen so far.

A *recurrent neural network* (RNN) is a type of neural network that can overcome those limitations. In RNN, the dependencies between nodes can form a cycle, which allows the network to preserve the state between subsequent timesteps. In the most general form, the RNN is a function that, in timestep $t$, takes the current input $(x_t)$, the state from the previous timestep $(s_{t-1})$ and produces the new state:

$$s_t = f^*(x_t, s_{t-1}; \theta) \tag{4}$$

RNNs process all elements from a sequence one-by-one, and the output at every timestep depends on all previous inputs. This fact has an important theoretical implication: RNNs are capable of approximating arbitrarily well

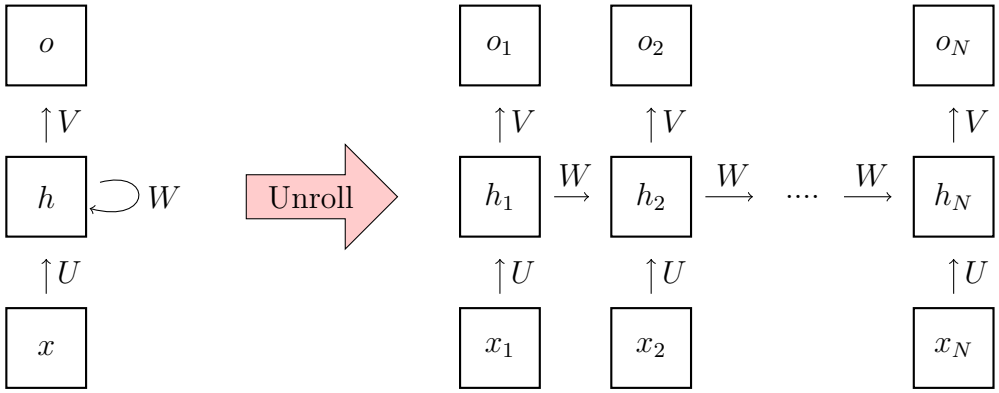



Figure 2: An example of unrolling recurrent neural network for $N$ timesteps.

any measurable sequence-to-sequence mapping [10]. The state $s_t$ can vary depending on the specific RNN – for example, in the vanilla RNN the state consists of one hidden state vector ($h$), while in, for example, LSTM the state is represented by two vectors: $h$ and memory cell vector $c$. Below those two RNN variants are described in more details, as both are building blocks of results obtained in this thesis.

## 4.1 Vanilla RNN

A basic version of the recurrent network is the *vanilla RNN*, which is presented in Figure 2. In this case, the Equation 4 can take the following form:

$$\begin{aligned} h_t &= \tanh(U * x_t + W * h_{t-1}) \\ o_t &= output(V * h_t) \end{aligned} \tag{5}$$

where $U$,$V$, $W$ are matrices and *output* is a function that produces final values at timestep $t$ from the hidden state (i.e. softmax). The tanh function is frequently used as activation for vanilla RNN, but can be also replaced by any other non-linear function.

Figure 2 presents also the idea of *unrolling*. Since RNNs contain loops, we cannot directly apply the standard backpropagation algorithm to compute gradients. Instead, we replicate the network $N$ times and change the recurrent connections. Every edge $(u, v)$ is converted into a new edge connecting node $u$ in layer $k$ with node $v$ in layer $k + 1$. Note that the matrices $U$, $V$, $W$ are the same for each timestep. This transforms the RNN into a standard feedforward network, where we can apply backpropagation, and average gradients from all timesteps. This method is called *backpropagation through time* (BPTT) [11] and it is commonly used to train RNNs.

However, there are some important problems that arise when applying BPTT. Notice that the unrolled network can be extremely deep. In this case, the gradients of the activation functions will be multiplied at least $N$ times. For some activation functions, the maximal value of the derivative is small. For example, the derivative of commonly used sigmoid function is never bigger than 0.25. As a result, after $N$ timesteps the gradient is multiplied by a value less than or equal to $0.25^N$ This leads to the problem known as *vanishing gradient*, as the gradient signal quickly becomes too weak to learn any dependency longer than few timesteps. Similarly, the *exploding gradient* can occur when derivatives bigger than 1.0 are multiplied many times.

Those two problems historically made people believe that RNNs are too hard to train. To address them, various techniques were proposed, such as *gradient clipping* (for exploding gradient), and architectures, such as Long Short-Term Memory (for vanishing gradient).

## 4.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM) is a neural architecture designed for storing and accessing information better than standard RNN [12]. The LSTM block consists of a self-connected memory cell and three gates named: input, output and forget. The gates control the access to the cell and can be interpreted as "read", "write" and "reset" operations in the standard computer's memory. The network learns to control the gates and decides to update and/or use the value at any given timestep. Since all of the components are built from differentiable functions, the gradients can be computed for the whole system and it is possible to train it end-to-end using backpropagation. There are several variants of LSTM that slightly differ in connectivity structure and activation functions. Below are described the definitions of input, output and forget gates used in this work.

Let $h_t \in \mathbb{R}^n$ be a hidden state, $c_t \in \mathbb{R}^n$ be a vector of memory cells of the network and let $x_t \in \mathbb{R}^n$ be the input at the timestep $t$. Let $W_i, W_f, W_u, W_o$ be matrices and $b_i, b_f, b_u, b_o$ the respective bias terms. We define LSTM as a transformation that takes 3 inputs ($h_{t-1}$, $c_{t-1}, x_t$) and produces 2 outputs ($h_t$ and $c_t$). In all equations below $\odot$ is element-wise multiplication and $\oplus$ is is element-wise addition.

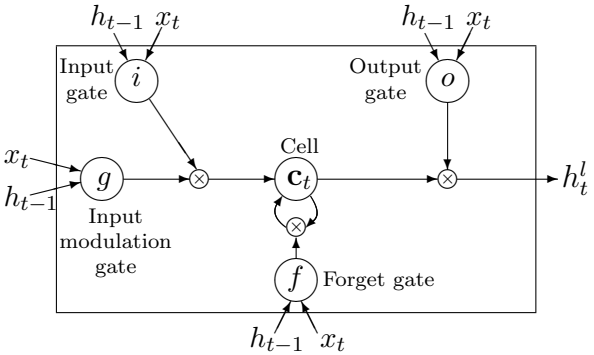



Figure 3: A graphical representation of LSTM memory cells. Figure adapted with permission from [13].

The connections between all gates are presented in Figure 3. The *forget gate* which decides how much of the information should be removed from the cell is defined as:

$$f_t = \text{sigm}(W_f * [h_{t-1} \oplus x_t] + b_f) \tag{6}$$

The *input modulation* gate value $i_t$ and the cell update $u_t$ are defined as:

$$\begin{aligned} i_t &= \text{sigm}(W_i * [h_{t-1} \oplus x_t] + b_i) \\ u_t &= \tanh(W_u * [h_{t-1} \oplus x_t] + b_u) \end{aligned} \tag{7}$$

Intuitively, input modulation decides how much of the $u_t$ should be added to the memory at step $t$. For example, if $x_t$ can be ignored, $i_t$ will be close to 0. Knowing the values above, the new cell value $c_t$ is computed as:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{8}$$

The last step is to compute $h_t$, the output passed to the next LSTM's timestep. It is controlled by the *output gate* $o_t$:

$$\begin{aligned} o_t &= \text{sigm}(W_o * [h_{t-1} \oplus x_t] + b_o) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{9}$$

The LSTM networks have been successfully applied to real-world problems, including language modeling [14], handwriting [15], speech recognition [16] and machine translation [17].

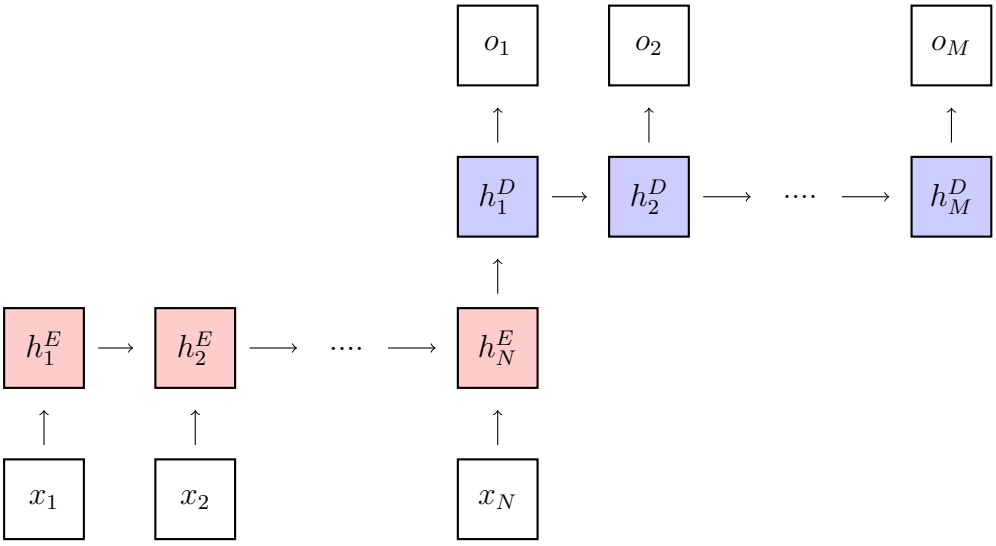



Figure 4: Encoder-decoder scheme for Neural Machine Translation. The encoder is unrolled for $N$ timesteps (marked in red) and the decoder is unrolled for $M$ timesteps (marked in blue).

# 5 Recurrent Networks with Attention

An important limitation of RNNs described in Section 4 is the size of their internal state. It was observed that, the bigger the size of the LSTM memory, the better the language modelling results [13]. It was also noticed that some heuristics, like feeding the input twice or providing the network with both a sequence and a reversed sequence, improve the performance [17]. To understand why those techniques help, let us take a look at Figure 4 which presents the standard encoder-decoder RNN setup used for Machine Translation.

The goal of the presented model is to translate sentences. For example, the input could be a sentence in Polish (with $x_1, \ldots, x_N$ being words in Polish) and we want to produce a translation into English (with $o_1, \ldots, o_M$ being words in English). There is a special symbol for missing word, so this model can translate any sentence consisting of up to $N$ words into any sentence consisting of up to $M$ words.

The first, lower part of the model is the *encoder*, which reads an input sequence word by word and update its internal state $h^E$. After processing the whole sequence, it passes its final state $h_N^E$ to the decoder. The decoder's tasks is to predict the translation conditioned only on the input from the encoder.

Notice, that in this setup the encoded state $h_N^E$ has to preserve all information about the initial sentence. This "compression" into a fixed-length vector

results in a dropping some information, and explains why giving the network both input and the reversed input can sometimes improve the performance.

One obvious solution to this problem would be to just use a bigger memory. However, this can significantly increase the number of model's parameters. For example, in LSTM the number of network's parameters grows quadratically with the memory size, making it harder to train and more prone to overfitting.

Recently, there has been a significant interest in creating neural network architectures that avoid the problem of memorization by employing the so-called *attention* mechanism. Such networks can *attend* to parts of the (potentially preprocessed) input sequence [18] while generating the output sequence. It is implemented by giving the network as an auxiliary input a linear combination of input symbols, where the weights of this linear combination can be controlled by the network. This approach has already proven to be very successful in areas of machine translation [18], speech recognition [19] and syntactic parsing [20].

## 5.1 Learning algorithms

One of the important applications of attention mechanism was the highly influential paper *Neural Turing Machines* (NTMs) [21], where the authors presented a versatile neural network architecture capable of learning simple algorithms from pure input-output examples. The main idea behind this type of neural networks is to let the network operate on an "external" memory, which size is independent of the number of the model parameters.

The NTM paper caused an outbreak of other neural network architectures with a goal of learning algorithms and operating on external memory. They usually fall into one of the categories:

- **Memory architectures based on attention**, such as Memory Networks [22] or Pointer Networks [23]
- **Memory architectures based on data structures**, such as Stack-Augmented RNN [24] or LSTM extended with a stack, a FIFO queue or double-eneded queue [25]
- **Parallel memory architectures**, such as Grid-LSTM [26] or Neural GPU [27]

# 6 Main results

## 6.1 Neural Random-Access Machines

In the paper **Neural Random-Access Machines** [1] we propose a neural architecture that has, as primitive operations, the ability to manipulate, store in memory, and dereference pointers into its working memory. The Neural Random-Access Machine (NRAM) is an computationally-universal model employing an external memory, which size does not depend on the number of model's parameters.

By providing our model with dereferencing as a primitive, it becomes possible to train it on problems whose solutions require pointer manipulation and chasing. We were able to train the proposed model from pure input/output pairs using the standard backpropagation algorithm. It has learned to solve algorithmic tasks and is capable of learning the concept of data structures that require pointers, like linked-lists and binary trees. For a subset of tasks we show that the found solution can generalize to sequences of arbitrary length. Moreover, memory access during inference can be done in a constant time under some assumptions. The most important contributions of the NRAM architecture include:

- A mechanism for location-based addressing using *fuzzy pointers* (probability distributions over $\mathbb{Z}_M$).
- A way of combining neural network with pre-defined modules.
- A differentiable mechanism for deciding when to terminate the computation.
- Novel training techniques, such as entropy term or way of enforcing distribution constrains.

### 6.1.1 Noise addition

The NRAM model can be very deep (up to hundreds of layers after unrolling) which makes the training very challenging. One of the important training techniques that we employ is the addition of gaussian noise to gradients during backpropagation. This technique was proposed in our paper **Adding Gradient Noise Improves Learning for Very Deep Networks** [3], in which we evaluate it on a number of modern neural architectures, including Neural Programmer [28], End-To-End Memory Networks [29] and Neural GPU [27]. We show that the noise addition can significantly improve the training results as well as stability of the re-training.

## 6.2 Hierarchical Attentive Memory

One of the limitations of the standard attention mechanism (see Section 5) is the fact, that the memory access complexity is $O(n)$. This can make it impractical for real-word problems, when the size of the input can be large (e.g., attention over books or long DNA sequences). To address this problem, in the paper **Learning Efficient Algorithms with Hierarchical Attentive Memory** [2] we propose a novel memory module for neural networks, called Hierarchical Attentive Memory (HAM).

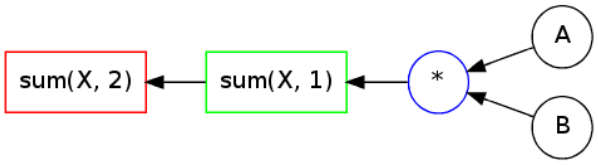
The HAM module is generic and can be used as a building block of larger neural architectures. Its crucial property is that it scales well with the memory size — the memory access requires only $\Theta(\log n)$ operations, where $n$ is the size of the memory. This complexity is achieved by using a new attention mechanism based on a binary tree with leaves corresponding to memory cells. The novel attention mechanism is not only faster than the standard one commonly used in Deep Learning [18], but it also facilities learning algorithms due to a built-in bias towards operating on intervals.

We show that an LSTM augmented with HAM is able to learn algorithms for tasks like merging, sorting or binary searching. In particular, it is the first neural network, that is able to learn sorting algorithm from pure input-output examples and generalizes well to input sequences much longer than the ones seen during the training. Moreover, the learned sorting algorithm runs in time $\Theta(n \log n)$. We also show that the HAM module itself can act as a drop-in replacement for classic data structures, like a stack, a FIFO queue or a priority queue.

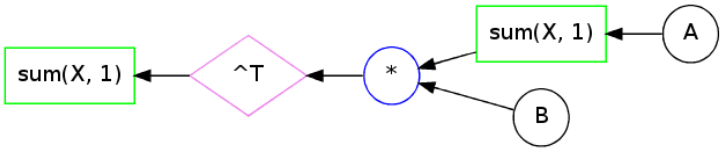
## 6.3 RNN for efficient algorithms

The running time of a computer program can critically depend on the algorithms used. For instance, one could solve a problem of sorting by using a very natural algorithm known as selection sort, with a complexity of $O(n^2)$. However, people discovered an alternative algorithm that can produce exactly the same output with a complexity of $O(n \log n)$. It would be desirable to have a system that, for given algorithm, can find the fastest drop-in replacement. That is, an algorithm providing exactly the same final result, but with a low computational complexity.

In our work we restrict our domain of algorithms to mathematical expressions over matrices. The motivational example is presented in Figure 5. Assume we are given matrices $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times p}$. We wish to compute the sum of all elements of the matrix $A * B$, i.e. $\sum_{n,p} AB = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{p} A_{i,j} B_{j,k}$ which naively takes $O(nmp)$ time. This formula




(a) The original expression.




(b) The efficient formula.

Figure 5: Two equivalent expressions. The bottom formula found by our framework avoids the use of a matrix-matrix multiply operation, hence is efficient to compute. The `sum(X, 1)` is a sum of matrix `X` along dimension 1 (a row vector containing the sum of each column) and `^T` is the matrix transposition.

could be expressed in matlab notation as `sum(sum(A*B))`. However, there exists an efficient version of the formula, that computes the same result in $O(n(m+p))$ time by avoiding matrix-matrix multiplication. It can be written in matlab as `sum((sum(A, 1) * B)', 1)`.

In the paper **Learning to Discover Efficient Mathematical Identities** [4] we introduce a framework based on attribute grammars for finding computationally efficient versions of symbolic math expressions. We show how machine learning techniques can be integrated into this framework, and demonstrate how training models on simpler expressions can help with the discovery of more complex ones. In particular, we present a novel application of an RNN to learn a continuous representation of mathematical structures.

We show how the exploration of the search space can be learned from previously successful solutions to simpler expressions. This allows us to discover complex expressions that random or brute-force strategies cannot find. We present examples of (i) $O(n^3)$ target expressions which can be computed in $O(n^2)$ time and (ii) cases where naive evaluation of the target would require *exponential* time, but can be computed in $O(n^2)$ or $O(n^3)$ time. The majority of these examples are too complex to be found manually or by exhaustive search and, as far as we are aware, are previously undiscovered.

## 6.4 RNN for monitoring systems

Monitoring systems that can predict dangerous events play a key role in ensuring people's safety. Recent data mining competitions: IJCRS'15 and AAIA'16 presented problems related to ensuring the safety of the underground coal mining workers. The IJCRS'15 competition was concerned with predicting dangerous levels of methane concentration, that can lead to explosion. The task of the AAIA'16 competition was related to predicting high-energy seismic events. Both problems are similar in nature: they were an instance of a classification problem with unbalanced data provided in a form of multivariate, non-stationary time series. The data was collected from Polish coal mines using various sensors.

In our paper **Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks** [5] we propose a method based on an RNN and an FNN for predicting the probability of dangerous methane concentration. The method is further improved in our next paper, **Predicting Dangerous Seismic Activity with Recurrent Neural Networks** [6], where we rely solely on an RNN.

The important aspect of our solution is the fact, that it learns to predict from raw sensor values, with a very minimal preprocessing. Most of the other top solutions relied heavily on feature engineering, either manual or automatic, such as: automatic variable construction [30], window-based feature engineering [31], hand-crafted features [32] or thousands of automatically generated features [33]. Such approach, while effective in the competition, is less likely to generalize to a different setting, as well as more complex to reproduce and deploy in a real production system.

Our method achieved a competitive score and placed 6th (out of 90) in the IJCRS'15 and 5th (out of 203 teams) in the AAIA'16 competition. Top performance in both competitions suggests that our approach is versatile and can be successfully applied to different multivariate time series problems.

## 6.5 RNN for email responses

Email is the primary medium of communication for billions of users across the world to connect and share information [34]. In our paper **Smart Reply: Automated Response Suggestion for Email** [7] we propose and implement a novel end-to-end system for generating short email responses. The system was implemented in *Google Inbox* and is currently responsible for generating 10% of responses on mobile. An example usage of Smart Reply is presented in Figure 6. A user is presented with 3 short response options available to use with just one tap.

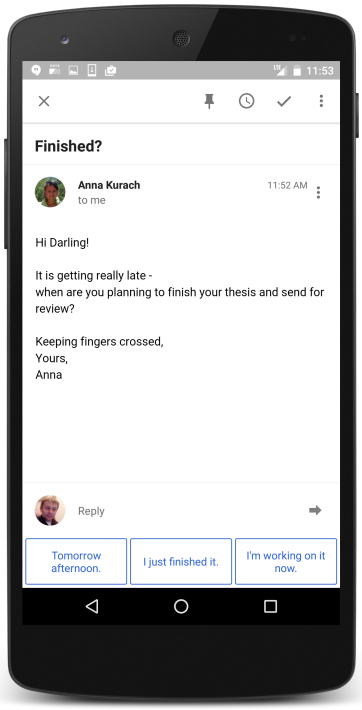



Figure 6: Example Smart Reply suggestions.

The Smart Reply system is extremely high-throughput oriented and can handle the processing of hundreds of messages daily. The system is based on sequence-to-sequence LSTM trained on a larger scale than before. We also present our solution to challenges that we faced while developing such system, including:

- **Quality** How to ensure that our system produces good quality and diverse set of responses.
- **Scalability** How to process hundreds of millions of messages and stay within latency requirements for an email system.
- **Privacy** How to develop such system without ever inspecting the data.

# Acknowledgements

First of all, I would like to thank my supervisor, Hung Son Nguyen for introducing me to the area of Machine Learning and convincing me to focus on it during my PhD studies.

I would also like to express my gratitude to Krzysztof Diks and Joanna Śmigielska for being inspirational algorithms teachers at the beginning of my programming career. The Machine Learning and algorithmic topics had a huge impact on the direction of this thesis.

Special thanks go to my co-authors: Greg Corrado, Rob Fergus, Marina Ganea, Andrzej Janusz, Łukasz Kaiser, Anjuli Kannan, Tobias Kaufmann, Quoc V. Le, László Lukács, James Martens, Tomasz Michalak, Balint Miklos, Arvind Neelakantan, Talal Rahwan, Vivek Ramavajjala, Sujith Ravi, Łukasz Romaszko, Ilya Sutskever, Marcin Tatjewski, Andrew Tomkins, Luke Vilnis, Peter Young and in particular to Marcin Andrychowicz, Krzysztof Pawłowski and Wojciech Zaremba with whom I have worked the most.

Last but not least, I would like to thank my amazing family for the support and constant encouragement. In particular, I would like to thank my wife Anna, brother Kamil, parents Agnieszka & Krzysztof as well as all relatives and friends who have been constantly asking me when I finish this thesis.

# Appendix A Acceptance rates

Below are detailed statistics (if available) for the relevant conferences.

1. **ICLR 2016 Conference Track**: acceptance rate is 24.5% (65 out of 265)
2. **NIPS 2014 spotlight**: acceptance rate for **spotlight** is 3.7% (62 out of 1678)
3. **IJCRS 2015**: for the "Competition Track" the papers from the top 6 (out of 90) teams were accepted (6.7%)
4. **AAIA 2016**: for the "Competition Track" the papers from the top 8 (out of 203) teams were accepted (3.9%)
5. **KDD 2016 Research Track**: acceptance rate for **presentation** is 9.2% (72 out of 784)

# Appendix B Other publications

During my PhD studies I also published the following two papers, which I have decided not to include in this dissertation, because they are not related to neural architectures.

1. **Coalition structure generation with the graphics processing unit**, accepted to AAMAS 2016 [35]
2. **An Ensemble Approach to Multi-label Classification of Textual Data**, accepted to ADMA 2012 [36]

# References


[1] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *International Conference on Learning Representations*, 2016.

[2] Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. *CoRR*, abs/1602.03218, 2016.

[3] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *International Conference on Learning Representations Workshop*, 2016.

[4] Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 1278–1286, 2014.

[5] Krzysztof Pawlowski and Karol Kurach. Detecting methane outbreaks from time series data with deep neural networks. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing - 15th International Conference, RSFDGrC 2015, Tianjin, China, November 20-23, 2015, Proceedings*, pages 475–484, 2015.

[6] Karol Kurach and Krzysztof Pawlowski. Detecting dangerous seismic events with recurrent neural networks. *11th International Symposium Advances in Artificial Intelligence and Applications*, 2016.

[7] Anjuli Kannan, Karol Kurach, Sujith Ravi, Tobias Kaufmann, Andrew Tomkins, Balint Miklos, Greg Corrado, László Lukács, Marina Ganea, Peter Young, and Vivek Ramavajjala. Smart reply: Automated response suggestion for email. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[8] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[9] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[10] Barbara Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1):107–123, 2000.

[11] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[14] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *INTERSPEECH*, 2012.

[15] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(5):855–868, 2009.

[16] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[17] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[18] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[19] William Chan, Navdeep Jaitly, Quoc V Le, and Oriol Vinyals. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

[20] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *arXiv preprint arXiv:1412.7449*, 2014.

[21] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[22] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[23] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

[24] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv preprint arXiv:1503.01007*, 2015.

[25] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1819–1827, 2015.

[26] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[27] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[28] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

[29] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *NIPS*, 2015.

[30] Marc Boullé. Prediction of methane outbreak in coal mines from historical sensor data under distribution drift. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 439–451. Springer, 2015.

[31] Marek Grzegorowski and Sebastian Stawicki. Window-based feature engineering for prediction of methane threats in coal mines. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 452–463. Springer, 2015.

[32] Petre Lameski, Eftim Zdravevski, Riste Mingov, and Andrea Kulakov. Svm parameter tuning with grid search and its impact on reduction of model over-fitting. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 464–474. Springer, 2015.

[33] Adam Zagorecki. Prediction of methane outbreaks in coal mines from multivariate time series using random forest. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 494–500. Springer, 2015.

[34] Ipsos Global Public Affairs. Interconnected world: Communication & social networking. Press Release, March 2012. `http://www.ipsos-na.com/news-polls/pressrelease.aspx?id=5564`.

[35] Krzysztof Pawlowski, Karol Kurach, Kim Svensson, Sarvapali D. Ramchurn, Tomasz P. Michalak, and Talal Rahwan. Coalition structure generation with the graphics processing unit. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, pages 293–300, 2014.

[36] Karol Kurach, Krzysztof Pawlowski, Lukasz Romaszko, Marcin Tatjewski, Andrzej Janusz, and Hung Son Nguyen. An ensemble approach to multi-label classification of textual data. In *Advanced Data Mining and Applications, 8th International Conference, ADMA 2012, Nanjing, China, December 15-18, 2012. Proceedings*, pages 306–317, 2012.

# NEURAL RANDOM-ACCESS MACHINES

**Karol Kurach*** **& Marcin Andrychowicz*** **& Ilya Sutskever**
Google
{kkurach,marcina,ilyasu}@google.com

## ABSTRACT

In this paper, we propose and investigate a new neural network architecture called Neural Random Access Machine. It can manipulate and dereference pointers to an external variable-size random-access memory. The model is trained from pure input-output examples using backpropagation.

We evaluate the new model on a number of simple algorithmic tasks whose solutions require pointer manipulation and dereferencing. Our results show that the proposed model can learn to solve algorithmic tasks of such type and is capable of operating on simple data structures like linked-lists and binary trees. For easier tasks, the learned solutions generalize to sequences of arbitrary length. Moreover, memory access during inference can be done in a constant time under some assumptions.

## 1 INTRODUCTION

Deep learning is successful for two reasons. First, deep neural networks are able to represent the "right" kind of functions; second, deep neural networks are trainable. Deep neural networks can be potentially improved if they get deeper and have fewer parameters, while maintaining trainability. By doing so, we move closer towards a practical implementation of Solomonoff induction (Solomonoff, 1964). The first model that we know of that attempted to train extremely deep networks with a large memory and few parameters is the Neural Turing Machine (NTM) (Graves et al., 2014) — a computationally universal deep neural network that is trainable with backpropagation. Other models with this property include variants of Stack-Augmented recurrent neural networks (Joulin & Mikolov, 2015; Grefenstette et al., 2015), and the Grid-LSTM (Kalchbrenner et al., 2015)—of which the Grid-LSTM has achieved the greatest success on both synthetic and real tasks. The key characteristic of these models is that their depth, the size of their short term memory, and their number of parameters are no longer confounded and can be altered independently — which stands in contrast to models like the LSTM (Hochreiter & Schmidhuber, 1997), whose number of parameters grows quadratically with the size of their short term memory.

A fundamental operation of modern computers is pointer manipulation and dereferencing. In this work, we investigate a model class that we name the Neural Random-Access Machine (NRAM), which is a neural network that has, as primitive operations, the ability to manipulate, store in memory, and dereference pointers into its working memory. By providing our model with dereferencing as a primitive, it becomes possible to train models on problems whose solutions require pointer manipulation and chasing. Although all computationally universal neural networks are equivalent, which means that the NRAM model does not have a representational advantage over other models if they are given a sufficient number of computational steps, in practice, the number of timesteps that a given model has is highly limited, as extremely deep models are very difficult to train. As a result, the model's core primitives have a strong effect on the set of functions that can be feasibly learned in practice, similarly to the way in which the choice of a programming language strongly affects the functions that can be implemented with an extremely small amount of code.

Finally, the usefulness of computationally-universal neural networks depends entirely on the ability of backpropagation to find good settings of their parameters. Indeed, it is trivial to define the "optimal" hypothesis class (Solomonoff, 1964), but the problem of finding the best (or even a good)

*Equal contribution.

function in that class is intractable. Our work puts the backpropagation algorithm to another test, where the model is extremely deep and intricate.

In our experiments, we evaluate our model on several algorithmic problems whose solutions required pointer manipulation and chasing. These problems include algorithms on a linked-list and a binary tree. While we were able to achieve encouraging results on these problems, we found that standard optimization algorithms struggle with these extremely deep and nonlinear models. We believe that advances in optimization methods will likely lead to better results.

## 2 RELATED WORK

There has been a significant interest in the problem of learning algorithms in the past few years. The most relevant recent paper is Neural Turing Machines (NTMs) (Graves et al., 2014). It was the first paper to explicitly suggest the notion that it is worth training a computationally universal neural network, and achieved encouraging results.

A follow-up model that had the goal of learning algorithms was the Stack-Augmented Recurrent Neural Network (Joulin & Mikolov, 2015) This work demonstrated that the Stack-Augmented RNN can generalize to long problem instances from short problem instances. A related model is the Reinforcement Learning Neural Turing Machine (Zaremba & Sutskever, 2015), which attempted to use reinforcement learning techniques to train a discrete-continuous hybrid model.

The memory network (Weston et al., 2014) is an early model that attempted to explicitly separate the memory from computation in a neural network model. The followup work of Sukhbaatar et al. (2015) combined the memory network with the soft attention mechanism, which allowed it to be trained with less supervision.

The Grid-LSTM (Kalchbrenner et al., 2015) is a highly interesting extension of LSTM, which allows to use LSTM cells for both deep and sequential computation. It achieves excellent results on both synthetic, algorithmic problems and on real tasks, such as language modelling, machine translation, and object recognition.

The Pointer Network (Vinyals et al., 2015) is somewhat different from the above models in that it does not have a writable memory — it is more similar to the attention model of Bahdanau et al. (2014) in this regard. Despite not having a memory, this model was able to solve a number of difficult algorithmic problems that include the convex hull and the approximate 2D travelling salesman problem (TSP).

Finally, it is important to mention the attention model of Bahdanau et al. (2014). Although this work is not explicitly aimed at learning algorithms, it is by far the most practical model that has an "algorithmic bent". Indeed, this model has proven to be highly versatile, and variants of this model have achieved state-of-the-art results on machine translation (Luong et al., 2015), speech recognition (Chan et al., 2015), and syntactic parsing (Vinyals et al., 2014), without the use of almost any domain-specific tuning.

## 3 MODEL

In this section we describe the NRAM model. We start with a description of the simplified version of our model which does not use an external memory and then explain how to augment it with a variable-size random-access memory. The core part of the model is a neural controller, which acts as a "processor". The controller can be a feedforward neural network or an LSTM, and it is the only trainable part of the model.

The model contains $R$ registers, each of which holds an integer value. To make our model trainable with gradient descent, we made it fully differentiable. Hence, each register represents an integer value with a distribution over the set $\{0, 1, \ldots, M-1\}$, for some constant $M$. We do not assume that these distributions have any special form — they are simply stored as vectors $p \in \mathbb{R}^M$ satisfying $p_i \geq 0$ and $\sum_i p_i = 1$. The controller does not have direct access to the registers; it can interact with them using a number of prespecified *modules* (gates), such as integer addition or equality test.

Let's denote the modules $m_1, m_2, \ldots, m_Q$, where each module is a function:

$$m_i : \{0, 1, \ldots, M-1\} \times \{0, 1, \ldots, M-1\} \to \{0, 1, \ldots, M-1\}.$$

On a high level, the model performs a sequence of timesteps, each of which consists of the following substeps:

1. The controller gets some inputs depending on the values of the registers (the controller's inputs are described in Sec. 3.1).
2. The controller updates its internal state (if the controller is an LSTM).
3. The controller outputs the description of a "fuzzy circuit" with inputs $r_1, \ldots, r_R$, gates $m_1, \ldots, m_Q$ and $R$ outputs.
4. The values of the registers are overwritten with the outputs of the circuit.

More precisely, each circuit is created as follows. The inputs for the module $m_i$ are chosen by the controller from the set $\{r_1, \ldots, r_R, o_1, \ldots, o_{i-1}\}$, where:

- $r_j$ is the value stored in the $j$-th register at the current timestep, and
- $o_j$ is the output of the module $m_j$ at the current timestep.

Hence, for each $1 \leq i \leq Q$ the controller chooses weighted averages of the values $\{r_1, \ldots, r_R, o_1, \ldots, o_{i-1}\}$ which are given as inputs to the module. Therefore,

$$o_i = m_i\left((r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(a_i),\ (r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(b_i)\right), \tag{1}$$

where the vectors $a_i, b_i \in \mathbb{R}^{R+i-1}$ are produced by the controller (Fig. 1).

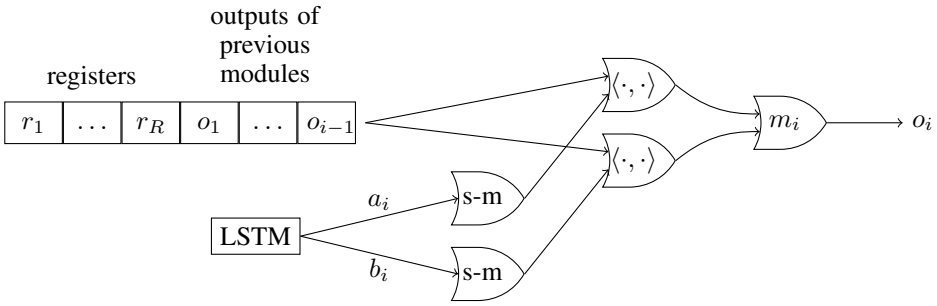



Figure 1: The execution of the module $m_i$. Gates *s-m* represent the softmax function and $\langle\cdot,\cdot\rangle$ denotes inner product. See Eq. 1 for details.

Recall that the variables $r_j$ represent probability distributions and therefore the inputs to $m_i$, being weighted averages of probability distributions, are also probability distributions. Thus, as the modules $m_i$ are originally defined for integer inputs and outputs, we must extend their domain to probability distributions as inputs, which can be done in a natural way (and make their output also be a probability distribution):

$$\forall_{0 \leq c < M}\ \mathbb{P}\left(m_i(A, B) = c\right) = \sum_{0 \leq a, b < M} \mathbb{P}(A = a)\mathbb{P}(B = b)[m_i(a, b) = c]. \tag{2}$$

After the modules have produced their outputs, the controller decides which of the values $\{r_1, \ldots, r_R, o_1, \ldots, o_Q\}$ should be stored in the registers. In detail, the controller outputs the vectors $c_i \in \mathbb{R}^{R+Q}$ for $1 \leq i \leq R$ and the values of the registers are updated (simultaneously) using the formula:

$$r_i := (r_1, \ldots, r_R, o_1, \ldots, o_Q)^T \mathbf{softmax}(c_i). \tag{3}$$

### 3.1 CONTROLLER'S INPUTS

Recall that at the beginning of each timestep the controller receives some inputs, and it is an important design decision to decide where should these inputs come from. A naive approach is to use the values of the registers as inputs to the controller. However, the values of the registers are probability distributions and are stored as vectors $p \in \mathbb{R}^M$. If the entire distributions were given as inputs to the controller then the number of the model's parameters would depend on $M$. This would be undesirable because, as will be explained in the next section, the value $M$ is linked to the size of an external random-access memory tape and hence it would prevent the model from generalizing to different memory sizes.

Hence, for each $1 \leq i \leq R$ the controller receives, as input, only one scalar from each register, namely $\mathbb{P}(r_i = 0)$ — the probability that the value in the register is equal 0. This solution has an additional advantage, namely it limits the amount of information available to the controller and forces it to rely on the modules instead of trying to solve the problem on its own. Notice that this information is sufficient to get the exact value of $r_i$ if $r_i \in \{0, 1\}$, which is the case whenever $r_i$ is an output of a „boolean" module, e.g. the inequality test module $m_i(a, b) = [a < b]$.

### 3.2 MEMORY TAPE

One could use the model described so far for learning sequence-to-sequence transformations by initializing the registers with the input sequence, and training the model to produce the desired output sequence in its registers after a given number of timesteps. The disadvantage of such model is that it would be completely unable to generalize to longer sequences, because the length of the sequence that the model can process is equal to the number of its registers, which is constant.

Therefore, we extend the model with a variable-size memory tape, which consists of $M$ memory cells, each of which stores a distribution over the set $\{0, 1, \ldots, M-1\}$. Notice that each distribution stored in a memory cell or a register can be interpreted as a fuzzy address in the memory and used as a fuzzy pointer. We will hence identify integers in the set $\{0, 1, \ldots, M-1\}$ with pointers to the memory. Therefore, the value in each memory cell may be interpreted as an integer or as a pointer. The exact state of the memory can be described by a matrix $\mathcal{M} \in \mathbb{R}^M_M$, where the value $\mathcal{M}_{i,j}$ is the probability that the $i$-th cell holds the value $j$.

The model interacts with the memory tape solely using two special modules:

- `READ` module: this module takes as the input a pointer[1] and returns the value stored under the given address in the memory. This operation is extended to fuzzy pointers similarly to Eq. 2. More precisely, if $p$ is a vector representing the probability distribution of the input (i.e. $p_i$ is the probability that the input pointer points to the $i$-th cell) then the module returns the value $\mathcal{M}^T p$.
- `WRITE` module: this module takes as the input a pointer $p$ and a value $a$ and stores the value $a$ under the address $p$ in the memory. The fuzzy form of the operation can be effectively expressed using matrix operations [2].

The full architecture of the NRAM model is presented on Fig. 2

### 3.3 INPUTS AND OUTPUTS HANDLING

The memory tape also serves as an input-output channel — the model's memory is initialized with the input sequence and the model is expected to produce the output in the memory. Moreover, we use a novel way of deciding how many timesteps should be executed. After each timestep we let the controller decide whether it would like to continue the execution or finish it, in which case the current state of the memory is treated as the output.

[1] Formally each module takes two arguments. In this case the second argument is simply ignored.

[2] The exact formula is $\mathcal{M} := (J - p)J^T \cdot \mathcal{M} + pa^T$, where $J$ denotes a (column) vector consisting of $M$ ones and $\cdot$ denotes coordinate-wise multiplication.

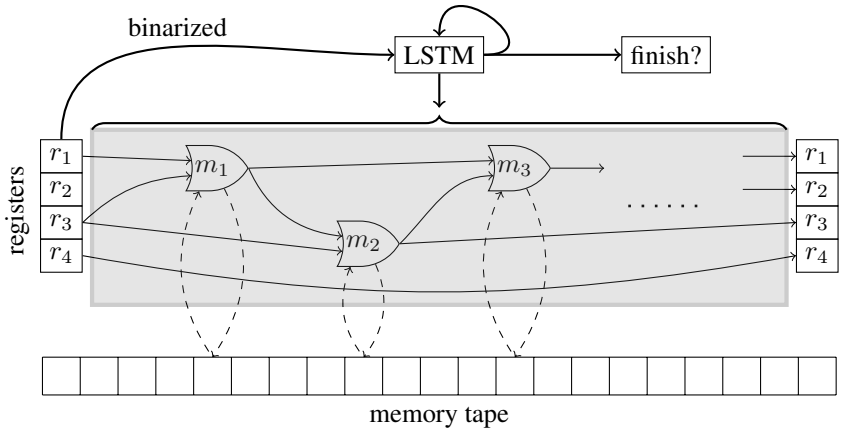



Figure 2: One timestep of the NRAM architecture with $R = 4$ registers. The LSTM controller gets the „binarized" values $r_1, r_2, \ldots$ stored in the registers as inputs and outputs the description of the circuit in the grey box and the probability of finishing the execution in the current timestep (See Sec. 3.3 for more detail). The weights of the solid thin connections are outputted by the controller. The weights of the solid thick connections are trainable parameters of the model. Some of the modules (i.e. `READ` and `WRITE`) may interact with the memory tape (dashed connections).

More precisely, after the timestep $t$ the controller outputs a scalar $f_t \in [0, 1]$[3], which denotes the willingness to finish the execution in the current timestep. Therefore, the probability that the execution has not been finished before the timestep $t$ is equal $\prod_{i=1}^{t-1}(1 - f_i)$, and the probability that the output is produced exactly at the timestep $t$ is equal $p_t = f_t \cdot \prod_{i=1}^{t-1}(1 - f_i)$. There is also some maximal allowed number of timesteps $T$, which is a hyperparameter. The model is forced to produce output in the last step if it has not done it yet, i.e. $p_T = 1 - \sum_{i=1}^{T-1} p_i$ regardless of the value $f_T$.

Let $\mathcal{M}^{(t)} \in \mathbb{R}_M^M$ denote the memory matrix after the timestep $t$, i.e. $\mathcal{M}_{i,j}^{(t)}$ is the probability that the $i$-th memory cell holds the value $j$ after the timestep $t$. For an input-output pair $(x, y)$, where $x, y \in \{0, 1, \ldots, M-1\}^M$ we define the loss of the model as the expected negative log-likelihood of producing the correct output, i.e., $-\sum_{t=1}^{T} \left( p_t \cdot \sum_{i=1}^{M} \log(\mathcal{M}_{i,y_i}^{(t)}) \right)$ assuming that the memory was initialized with the sequence $x$[4]. Moreover, for all problems we consider the output sequence is shorter than the memory. Therefore, we compute the loss only over memory cells, which should contain the output.

### 3.4 Discretization

Computing the outputs of modules, represented as probability distributions, is a computationally costly operation. For example, computing the output of the `READ` module takes $\Theta(M^2)$ time as it requires the multiplication of the matrix $\mathcal{M} \in \mathbb{R}_M^M$ and the vector $p \in \mathbb{R}^M$.

One may however suspect (and we empirically verify this claim in Sec. 4) that the NRAM model naturally learns solutions in which the distributions of intermediate values have very low entropy. The argument for this hypothesis is that fuzziness in the intermediate values would probably propagate to the output and cause a higher value of the cost function. To test this hypothesis we trained the model and then used its *discretized* version during interference. In the discretized version every module gets as inputs the values from modules (or registers), which are the most probable to produce

[3]In fact, the controller outputs a scalar $x_i$ and $f_i = \mathbf{sigmoid}(x_i)$.

[4]One could also use the negative log-likelihood of the expected output, i.e. $-\sum_{i=1}^{M} \log \left( \sum_{t=1}^{T} p_t \cdot \mathcal{M}_{i,y_i}^{(t)} \right)$ as the loss function.

the given input accordingly to the distribution outputted by the controller. More precisely, it corresponds to replacing the function $\mathbf{softmax}$ in equations (1,3) with the function returning the vector containing $1$ on the position of the maximum value in the input and zeros on all other positions.

Notice that in the discretized NRAM model each register and memory cell stores an integer from the set $\{0, 1, \ldots, M-1\}$ and therefore all modules may be executed efficiently (assuming that the functions represented by the modules can be efficiently computed). In case of a feedforward controller and a small (e.g. $\leq 20$) number of registers the interference can be accelerated even further. Recall that the only inputs to the controller are binarized values of the register. Therefore, instead of executing the controller one may simple precompute the (discretized) controller's output for each configuration of the registers' binarized values. Such algorithm would enjoy an extremely efficient implementation in machine code.

# 4 EXPERIMENTS

## 4.1 TRAINING PROCEDURE

The NRAM model is fully differentiable and we trained it using the Adam optimization algorithm (Kingma & Ba, 2014) with the negative log-likelihood cost function. Notice that we do not use any additional supervised data (such as memory access traces) beyond pure input-output examples.

We used multilayer perceptrons (MLPs) with two hidden layers or LSTMs with a hidden layer between input and LSTM cells as controllers. The number of hidden units in each layer was equal. The ReLu nonlinearity (Nair & Hinton, 2010) was used in all experiments.

Below are some important techniques that we used in the training:

**Curriculum learning** As noticed in several papers (Bengio et al., 2009; Zaremba & Sutskever, 2014), curriculum learning is crucial for training deep networks on very complicated problems. We followed the curriculum learning schedule from Zaremba & Sutskever (2014) without any modifications. The details can be found in Appendix B.

**Gradient clipping** Notice that the depth of the unfolded execution is roughly a product of the number of timesteps and the number of modules. Even for moderately small experiments (e.g. $14$ modules and $20$ timesteps) this value easily exceeds a few hundreds. In networks of such depth, the gradients can often "explode" (Bengio et al., 1994), what makes training by backpropagation much harder. We noticed that the gradients w.r.t. the intermediate values inside the backpropagation were so large, that they sometimes led to an overflow in single-precision floating-point arithmetic. Therefore, we clipped the gradients w.r.t. the activations, within the execution of the backpropagation algorithm. More precisely, each coordinate is separately cropped into the range $[-C_1, C_1]$ for some constant $C_1$. Before updating parameters, we also globally rescale the whole gradient vector, so that its L2 norm is not bigger than some constant value $C_2$.

**Noise** We added random Gaussian noise to the computed gradients after the backpropagation step. The variance of this noise decays exponentially during the training. The details can be found in Neelakantan et al. (2015).

**Enforcing Distribution Constraints** For very deep networks, a small error in one place can propagate to a huge error in some other place. This was the case with our pointers: they are probability distributions over memory cells and they should sum up to 1. However, after a number of operations are applied, they can accumulate error as a result of inaccurate floating-point arithmetic.

We have a special layer which is responsible for rescaling all values (multiplying by the inverse of their sum), to make sure they always represent a probability distribution. We add this layer to our model in a few critical places (eg. after the softmax operation)[5].

[5]We do not however backpropagate through these renormalizing operations, i.e. during the backward pass we simply assume that they are identities.

**Entropy** While searching for a solution, the network can fix the pointer distribution on some particular value. This is advantageous at the end of training, because ideally we would like to be able to discretize the model. However, if this happens at the begin of the training, it could force the network to stay in a local minimum, with a small chance of moving the probability mass to some other value. To address this problem, we encourage the network to explore the space of solutions by adding an "entropy bonus", that decreases over time. More precisely, for every distribution outputted by the controller, we subtract from the cost function the entropy of the distribution multiplied by some coefficient, which decreases exponentially during the training.

**Limiting the values of logarithms** There are two places in our model where the logarithms are computed — in the cost function and in the entropy computation. Inputs to whose logarithms can be very small numbers, which may cause very big values of the cost function or even overflows in floating-point arithmetic. To prevent this phenomenon we use $\log(\max(x, \epsilon))$ instead of $\log(x)$ for some small hyperparameter $\epsilon$ whenever a logarithm is computed.

### 4.2 TASKS

We now describe the tasks used in our experiments. For every task, the input is given to the network in the memory tape, and the network's goal is to modify the memory according to the task's specification. We allow the network to modify the original input. The final error for a test case is computed as $\frac{c}{m}$, where $c$ is the number of correctly written cells, and $m$ represents the total number of cells that should be modified.

Due to limited space, we describe the tasks only briefly here. The detailed memory layout of inputs and outputs can be found in the Appendix A.

1. **Access** Given a value $k$ and an array $A$, return $A[k]$.
2. **Increment** Given an array, increment all its elements by 1.
3. **Copy** Given an array and a pointer to the destination, copy all elements from the array to the given location.
4. **Reverse** Given an array and a pointer to the destination, copy all elements from the array in reversed order.
5. **Swap** Given two pointers $p$, $q$ and an array $A$, swap elements $A[p]$ and $A[q]$.
6. **Permutation** Given two arrays of $n$ elements: $P$ (contains a permutation of numbers $1, \ldots, n$) and $A$ (contains random elements), permutate $A$ according to $P$.
7. **ListK** Given a pointer to the head of a linked list and a number $k$, find the value of the $k$-th element on the list.
8. **ListSearch** Given a pointer to the head of a linked list and a value $v$ to find return a pointer to the first node on the list with the value $v$.
9. **Merge** Given pointers to 2 sorted arrays $A$ and $B$, merge them.
10. **WalkBST** Given a pointer to the root of a Binary Search Tree, and a path to be traversed (sequence of left/right steps), return the element at the end of the path.

### 4.3 MODULES

In all of our experiments we used the same sequence of 14 modules: `READ` (described in Sec. 3.2), $\texttt{ZERO}(a, b) = 0$, $\texttt{ONE}(a, b) = 1$, $\texttt{TWO}(a, b) = 2$, $\texttt{INC}(a, b) = (a+1) \mod M$, $\texttt{ADD}(a, b) = (a+b) \mod M$, $\texttt{SUB}(a, b) = (a-b) \mod M$, $\texttt{DEC}(a, b) = (a-1) \mod M$, $\texttt{LESS-THAN}(a, b) = [a < b]$, $\texttt{LESS-OR-EQUAL-THAN}(a, b) = [a \leq b]$, $\texttt{EQUALITY-TEST}(a, b) = [a = b]$, $\texttt{MIN}(a, b) = min(a, b)$, $\texttt{MAX}(a, b) = max(a, b)$, `WRITE` (described in Sec. 3.2).

We also considered settings in which the module sequence is repeated many times, e.g. there are 28 modules, where modules number 1. and 15. are `READ`, modules number 2. and 16. are `ZERO` and so on. The number of repetitions is a hyperparameter.

| Task | Train Complexity | Train error | Generalization | Discretization |
|---|---|---|---|---|
| Access | $len(A) \leq 20$ | 0 | perfect | perfect |
| Increment | $len(A) \leq 15$ | 0 | perfect | perfect |
| Copy | $len(A) \leq 15$ | 0 | perfect | perfect |
| Reverse | $len(A) \leq 15$ | 0 | perfect | perfect |
| Swap | $len(A) \leq 20$ | 0 | perfect | perfect |
| Permutation | $len(A) \leq 6$ | 0 | almost perfect | perfect |
| ListK | $len(list) \leq 10$ | 0 | strong | hurts performance |
| ListSearch | $len(list) \leq 6$ | 0 | weak | hurts performance |
| Merge | $len(A) + len(B) \leq 10$ | 1% | weak | hurts performance |
| WalkBST | $size(tree) \leq 10$ | 0.3% | strong | hurts performance |

Table 1: Results of the experiments. The **perfect** generalization error means that the tested problem had error 0 for complexity up to 50. Exact generalization errors are presented in Fig. 3 The **perfect** discretization means that the discretized version of the model produced exactly the same outputs as the original model on all test cases.

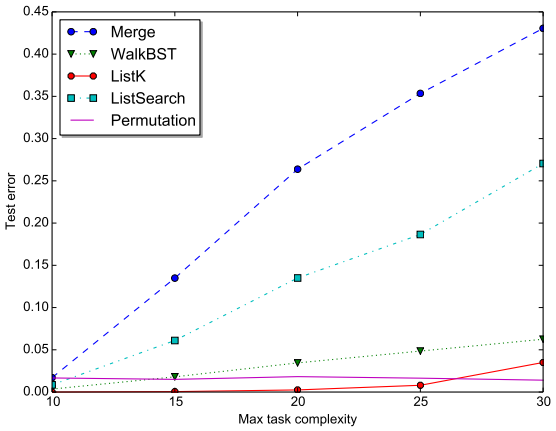



Figure 3: Generalization errors for hard tasks. The **Permutation** and **ListSearch** problems were trained only up to complexity 6. The remaining problems were trained up to complexity 10. The horizontal axis denotes the *maximal* task complexity, i.e., $x = 20$ denotes results with complexity sampled uniformly from the interval $[1, 20]$.

### 4.4 Results

Overall, we were able to find parameters that achieved an error 0 for all problems except **Merge** and **WalkBST** (where we got an error of $\leq 1\%$). As described in 4.2, our metric is an accuracy on the memory cells that should be modified. To compute it, we take the continuous memory state produced by our network, then discretize it (every cell will contain the value with the highest probability), and finally compare with the expected output. The results of the experiments are summarized in Table 1.

Below we describe our results on all 10 tasks in more detail. We divide them into 2 categories: "easy" and "hard" tasks. Easy tasks is a category of tasks that achieved low error scores for many sets of parameters and we did not have to spend much time trying to tune them. First 5 problems from our task list belong to this category. Hard tasks, on the other hand, are problems that often trained to low error rate only in a very small number of cases, eg. 1 out of 100.

#### 4.4.1 Easy tasks

This category includes the following problems: **Access**, **Increment**, **Copy**, **Reverse**, **Swap**. For all of them we were able to find many sets of hyperparameters that achieved error 0, or close to it without much effort.

| **Step** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | $r_1$ | $r_2$ | $r_3$ | $r_4$ | READ | WRITE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 2 | 10 | 6 | 8 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | p:0 | p:0 a:6 |
| 2 | 6 | 2 | 10 | 6 | 8 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 1 | p:1 | p:6 a:2 |
| 3 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 1 | 1 | p:1 | p:6 a:2 |
| 4 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 1 | 2 | p:2 | p:7 a:10 |
| 5 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 2 | p:2 | p:7 a:10 |
| 6 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 3 | p:3 | p:8 a:6 |
| 7 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 6 | 0 | 0 | 0 | 0 | 5 | 3 | 3 | p:3 | p:8 a:6 |
| 8 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 6 | 0 | 0 | 0 | 0 | 5 | 3 | 4 | p:4 | p:9 a:8 |
| 9 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 6 | 8 | 0 | 0 | 0 | 5 | 4 | 4 | p:4 | p:9 a:8 |
| 10 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 6 | 8 | 0 | 0 | 0 | 5 | 4 | 5 | p:5 | p:10 a:9 |
| 11 | 6 | 2 | 10 | 6 | 8 | 9 | 2 | 10 | 6 | 8 | 9 | 0 | 0 | 5 | 5 | 5 | p:5 | p:10 a:9 |

Table 2: State of memory and registers for the **Copy** problem at the start of every timestep. We also show the arguments given to the READ and WRITE functions in each timestep. The argument "p:" represents the source/destination address and "a:" represents the value to be written (for WRITE). The value 6 at position 0 in the memory is the pointer to the destination array. It is followed by 5 values (gray columns) that should be copied.

We also tested how those solutions generalize to longer input sequences. To do this, for every problem we selected a model that achieved error 0 during the training, and tested it on inputs with lengths up to $50$[6]. To perform these tests we also increased the memory size and the number of allowed timesteps.

In all cases the model solved the problem perfectly, what shows that it generalizes not only to longer input sequences, but also to different memory sizes and numbers of allowed timesteps. Moreover, the discretized version of the model (see Sec. 3.4 for details) also solves all the problems perfectly. These results show that the NRAM model naturally learns "algorithmic" solutions, which generalize well.

We were also interested if the found solutions generalize to sequences of arbitrary length. It is easiest to verify in the case of a discretized model with a feedforward controller. That is because then circuits outputted by the controller depend solely on the values of registers, which are integers. We manually analysed circuits for problems **Copy** and **Increment** and verified that found solutions generalize to inputs of arbitrary length, assuming that the number of allowed timesteps is appropriate.

#### 4.4.2 HARD TASKS

This category includes: **Permutation**, **ListK**, **ListSearch**, **Merge** and **WalkBST**. For all of them we had to perform an extensive random search to find a good set of hyperparameters. Usually, most of the parameter combinations were stuck on the starting curriculum level with a high error of $50\% - 70\%$. For the first 3 tasks we managed to train the network to achieve error 0. For **WalkBST** and **Merge** the training errors were $0.3\%$ and $1\%$ respectively. For training those problems we had to introduce additional techniques described in Sec. 4.1.

For **Permutation**, **ListK** and **WalkBST** our model generalizes very well and achieves low error rates on inputs at least twice longer than the ones seen during the training. The exact generalization errors are shown in Fig. 3.

The only hard problem on which our model discretizes well is **Permutation** — on this task

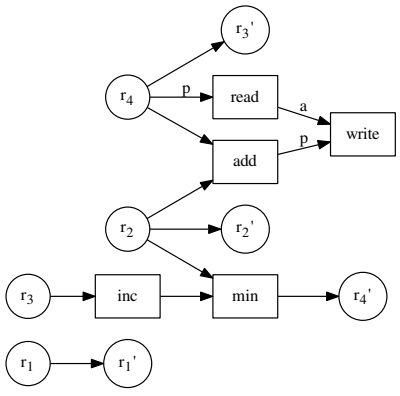



Figure 4: The circuit generated at every timestep $\geq 2$. The values of the pointer ($p$) for READ, WRITE and the value to be written ($a$) for WRITE are presented in Table 2. The modules whose outputs are not used were removed from the picture.

[6]Unfortunately we could not test for lengths longer than 50 due to the memory restrictions.

the discretized version of the model produces exactly the same outputs as the original model on all cases tested. For the remaining four problems the discretized version of the models perform very poorly (error rates $\geq 70\%$). We believe that better results may be obtained by using some techniques encouraging discretization during the training [7].

We noticed that the training procedure is very unstable and the error often raises from a few percents to e.g. $70\%$ in just one epoch. Moreover, even if we use the best found set of hyperparameters, the percent of random seeds that converges to error $0$ was usually equal about $11\%$. We observed that the percent of converging seeds is much lower if we do not add noise to the gradient — in this case only about $1\%$ of seeds converge.

### 4.5 Comparison to existing models

A comparison to other models is challenging because we are the first to consider problems with pointers. The NTM can solve tasks like **Copy** or **Reverse**, but it suffers from the inability to naturally store a pointer to a fixed location in the memory. This makes it unlikely that it could solve tasks such as **ListK, ListSearch** or **WalkBST** since the pointers used in these tasks refer to absolute positions.

What distinguishes our model from most of the previous attempts (including NTMs, Memory Networks, Pointer Networks) is the lack of content-based addressing. It was a deliberate design decision, since this kind of addressing inherently slows down the memory access. In contrast, our model — if discretized — can access the memory in a constant time.

The NRAM is also the first model that we are aware of employing a differentiable mechanism for deciding when to finish the computation.

### 4.6 Exemplary execution

We present one example execution of our model for the problem **Copy**. For the example, we use a very small model with $12$ memory cells, $4$ registers and the standard set of $14$ modules. The controller for this model is a feedforward network, and we run it for $11$ timesteps. Table 2 contains, for every timestep, the state of the memory and registers at the begin of the timestep.

The model can execute different circuits at different timesteps. In particular, we observed that the first circuit is slightly different from the rest, since it needs to handle the initialization. Starting from the second step all generated circuits are the same. We present this circuit in Fig. 4. The register $r_2$ is constant and keeps the offset between the destination array and the source array ($6 - 1 = 5$ in this case). The register $r_3$ is responsible for incrementing the pointer in the source array. Its value is copied to $r_4$[8], the register used by the `READ` module. For the `WRITE` module, it also uses $r_4$ which is shifted by $r_2$. The register $r_1$ is unused. This solution generalizes to sequences of arbitrary length.

## 5 Conclusions

In this paper we presented the Neural Random-Access Machine, which can learn to solve problems that require explicit manipulation and dereferencing of pointers.

We showed that this model can learn to solve a number of algorithmic problems and generalize well to inputs longer than ones seen during the training. In particular, for some problems it generalizes to inputs of arbitrary length.

However, we noticed that the optimization problem resulting from the backpropagating through the execution trace of the program is very challenging for standard optimization techniques. It seems likely that a method that can search in an easier "abstract" space would be more effective at solving such problems.

[7] One could for example add at later stages of training a penalty proportional to the entropy of the intermediate values of registers/memory.

[8] In our case $r_3 < r_2$, so the `MIN` module always outputs the value $r_3 + 1$. It is not satisfied in the last timestep, but then the array is already copied.

## REFERENCES


Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

Bengio, Yoshua, Louradour, Jérôme, Collobert, Ronan, and Weston, Jason. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

Chan, William, Jaitly, Navdeep, Le, Quoc V, and Vinyals, Oriol. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. *arXiv preprint arXiv:1506.02516*, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv preprint arXiv:1503.01007*, 2015.

Kalchbrenner, Nal, Danihelka, Ivo, and Graves, Alex. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Luong, Minh-Thang, Pham, Hieu, and Manning, Christopher D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

Neelakantan, Arvind, Vilnis, Luke, Le, Quoc V, Sutskever, Ilya, Kaiser, Lukasz, Kurach, Karol, and Martens, James. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

Solomonoff, Ray J. A formal theory of inductive inference. part i. *Information and control*, 7(1): 1–22, 1964.

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Vinyals, Oriol, Kaiser, Lukasz, Koo, Terry, Petrov, Slav, Sutskever, Ilya, and Hinton, Geoffrey. Grammar as a foreign language. *arXiv preprint arXiv:1412.7449*, 2014.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Weston, Jason, Chopra, Sumit, and Bordes, Antoine. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.

## A DETAILED TASKS DESCRIPTIONS

In this section we describe in details the memory layout of inputs and outputs for the tasks used in our experiments. In all descriptions below, big letters represent arrays and small letters represents pointers. $NULL$ denotes the value 0 and is used to mark the end of an array or a missing next element in a list or a binary tree.

1. **Access** Given a value $k$ and an array $A$, return $A[k]$. Input is given as $k, A[0], .., A[n-1], NULL$ and the network should replace the first memory cell with $A[k]$.
2. **Increment** Given an array $A$, increment all its elements by 1. Input is given as $A[0], ..., A[n-1], NULL$ and the expected output is $A[0]+1, ..., A[n-1]+1$.
3. **Copy** Given an array and a pointer to the destination, copy all elements from the array to the given location. Input is given as $p, A[0], ..., A[n-1]$ where $p$ points to one element after $A[n-1]$. The expected output is $A[0], ..., A[n-1]$ at positions $p, ..., p+n-1$ respectively.
4. **Reverse** Given an array and a pointer to the destination, copy all elements from the array in reversed order. Input is given as $p, A[0], ..., A[n-1]$ where $p$ points one element after $A[n-1]$. The expected output is $A[n-1], ..., A[0]$ at positions $p, ..., p+n-1$ respectively.
5. **Swap** Given two pointers $p$, $q$ and an array $A$, swap elements $A[p]$ and $A[q]$. Input is given as $p, q, A[0], .., A[p], ..., A[q], ..., A[n-1], 0$. The expected modified array $A$ is: $A[0], ..., A[q], ..., A[p], ..., A[n-1]$.
6. **Permutation** Given two arrays of $n$ elements: $P$ (contains a permutation of numbers $0, \ldots, n-1$) and $A$ (contains random elements), permutate $A$ according to $P$. Input is given as $a, P[0], ..., P[n-1], A[0], ..., A[n-1]$, where $a$ is a pointer to the array $A$. The expected output is $A[P[0]], ..., A[P[n-1]]$, which should override the array $P$.
7. **ListK** Given a pointer to the head of a linked list and a number $k$, find the value of the $k$-th element on the list. List nodes are represented as two adjacent memory cells: a pointer to the next node and a value. Elements are in random locations in the memory, so that the network needs to follow the pointers to find the correct element. Input is given as: $head, k, out, ...$ where head is a pointer to the first node on the list, $k$ indicates how many hops are needed and $out$ is a cell where the output should be put.
8. **ListSearch** Given a pointer to the head of a linked list and a value $v$ to find return a pointer to the first node on the list with the value $v$. The list is placed in memory in the same way as in the task **ListK**. We fill empty memory with "trash" values to prevent the network from "cheating" and just iterating over the whole memory.
9. **Merge** Given pointers to 2 sorted arrays $A$ and $B$, and the pointer to the output $o$, merge the two arrays into one sorted array. The input is given as: $a, b, o, A[0], .., A[n-1], G, B[0], ..., B[m-1], G$, where $G$ is a special guardian value, $a$ and $b$ point to the first elements of arrays $A$ and $B$ respectively, and $o$ points to the address after the second $G$. The $n+m$ element should be written in correct order starting from position $o$.
10. **WalkBST** Given a pointer to the root of a Binary Search Tree, and a path to be traversed, return the element at the end of the path. The BST nodes are represented as tripes ($v$, $l$, $r$), where $v$ is the value, and $l$, $r$ are pointers to the left/right child. The triples are placed randomly in the memory. Input is given as $root, out, d_1, d_2, ..., d_k, NULL, ...$, where $root$ points to the root node and $out$ is a slot for the output. The sequence $d_1 ... d_k, d_i \in \{0, 1\}$ represents the path to be traversed: $d_i = 0$ means that the network should go to the left child, $d_i = 1$ represents going to the right child.

## B DETAILS OF CURRICULUM TRAINING

As noticed in several papers (Bengio et al., 2009; Zaremba & Sutskever, 2014), curriculum learning is crucial for training deep networks on very complicated problems. We followed the curriculum learning schedule from Zaremba & Sutskever (2014) without any modifications.

For each of the tasks we have manually defined a sequence of subtasks with increasing difficulty, where the difficulty is usually measured by the length of the input sequence. During training the input-output examples are sampled from a distribution that is determined by the current difficulty level $D$. The level is increased (up to some maximal value) whenever the error rate of the model goes below some threshold. Moreover, we ensure that successive increases of $D$ are separated by some number of batches.

In more detail, to generate an input-output example we first sample a difficulty $d$ from a distribution determined by the current level $D$ and then draw the example with the difficulty $d$. The procedure for sampling $d$ is the following:

- with probability $10\%$: pick $d$ uniformly at random from the set of all possible difficulties;
- with probability $25\%$: pick $d$ uniformly from $[1, D + e]$, where $e$ is a sample from a geometric distribution with a success probability $1/2$;
- with probability $65\%$: set $d = D + e$, where $e$ is sampled as above.

Notice that the above procedure guarantees that every difficulty $d$ can be picked regardless of the current level $D$, which has been shown to increase performance Zaremba & Sutskever (2014).

# C EXAMPLE CIRCUITS

Below are presented example circuits generated during training for all simple tasks (except **Copy** which was presented in the paper). For modules `READ` and `WRITE`, the value of the first argument (pointer to the address to be read/written) is marked as $p$. For `WRITE`, the value to be written is marked as $a$ and the value returned by this module is always $0$. For modules `LESS-THAN` and `LESS-OR-EQUAL-THAN` the first parameter is marked as $x$ and the second one as $y$. Other modules either have only one parameter or the order of parameters is not important.

For all tasks below (except **Increment**), the circuit generated at timestep $1$ is different than circuits generated at steps $\geq 2$, which are the same. This is because the first circuit needs to handle the initialization. We present only the "main" circuits generated for timesteps $\geq 2$.

## C.1 ACCESS

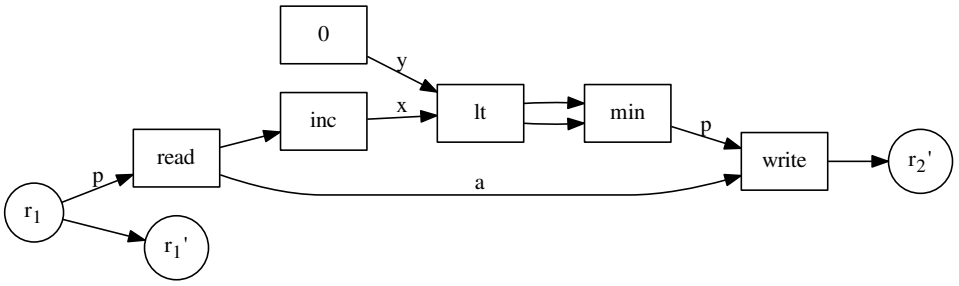



Figure 5: The circuit generated at every timestep $\geq 2$ for the task **Access**.

| **Step** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $r_1$ | $r_2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 12 | 4 | 7 | 12 | 1 | 13 | 8 | 2 | 1 | 3 | 11 | 11 | 12 | 0 | 0 | 0 |
| 2 | 3 | 1 | 12 | 4 | 7 | 12 | 1 | 13 | 8 | 2 | 1 | 3 | 11 | 11 | 12 | 0 | 3 | 0 |
| 3 | 4 | 1 | 12 | 4 | 7 | 12 | 1 | 13 | 8 | 2 | 1 | 3 | 11 | 11 | 12 | 0 | 3 | 0 |

Table 3: Memory for task **Access**. Only the first memory cell is modified.

### C.2 INCREMENT

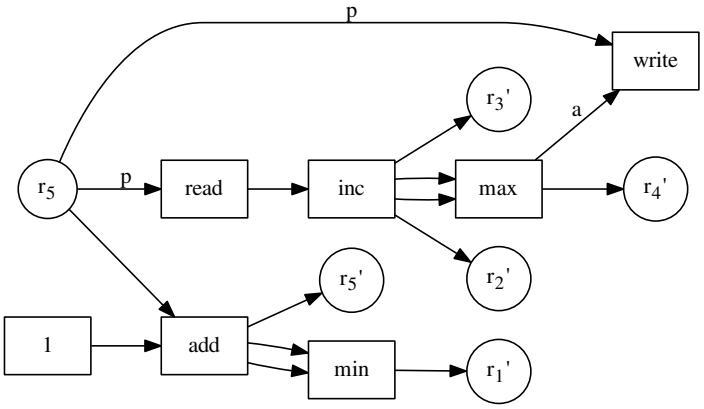



Figure 6: The circuit generated at every timestep for the task **Increment**.

| **Step** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 11 | 3 | 8 | 1 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 2 | 11 | 3 | 8 | 1 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 1 |
| 3 | 2 | 12 | 3 | 8 | 1 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 12 | 12 | 12 | 2 |
| 4 | 2 | 12 | 4 | 8 | 1 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 4 | 4 | 4 | 3 |
| 5 | 2 | 12 | 4 | 9 | 1 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 9 | 9 | 9 | 4 |
| 6 | 2 | 12 | 4 | 9 | 2 | 2 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 2 | 2 | 5 |
| 7 | 2 | 12 | 4 | 9 | 2 | 3 | 9 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 3 | 3 | 3 | 6 |
| 8 | 2 | 12 | 4 | 9 | 2 | 3 | 10 | 8 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 10 | 10 | 10 | 7 |
| 9 | 2 | 12 | 4 | 9 | 2 | 3 | 10 | 9 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 9 | 9 | 9 | 8 |
| 10 | 2 | 12 | 4 | 9 | 2 | 3 | 10 | 9 | 6 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 6 | 6 | 6 | 9 |
| 11 | 2 | 12 | 4 | 9 | 2 | 3 | 10 | 9 | 6 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 4 | 4 | 4 | 10 |

Table 4: Memory for task **Increment**.

### C.3 REVERSE

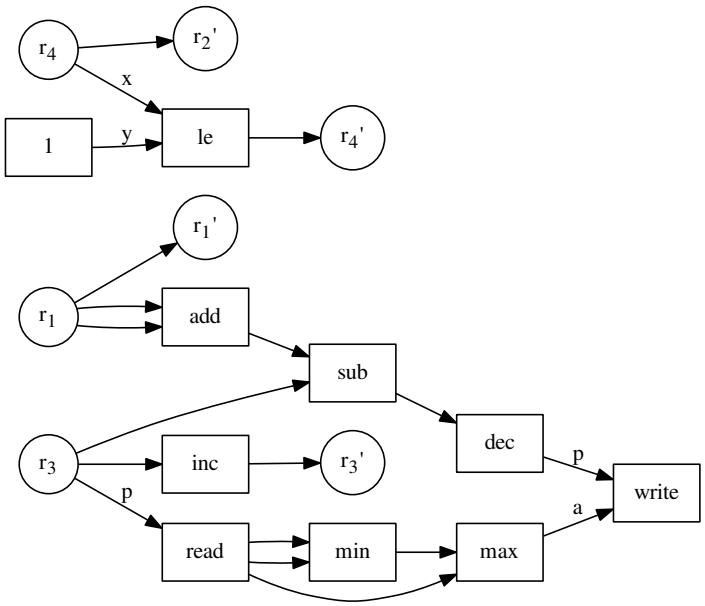



Figure 7: The circuit generated at every timestep $\geq 2$ for the task **Reverse**.

| **Step** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $r_1$ | $r_2$ | $r_3$ | $r_4$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 1 | 1 |
| 3 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 8 | 1 | 2 | 1 |
| 4 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 8 | 0 | 8 | 1 | 3 | 1 |
| 5 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 0 | 3 | 1 | 8 | 0 | 8 | 1 | 4 | 1 |
| 6 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 0 | 5 | 3 | 1 | 8 | 0 | 8 | 1 | 5 | 1 |
| 7 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 0 | 1 | 5 | 3 | 1 | 8 | 0 | 8 | 1 | 6 | 1 |
| 8 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 0 | 1 | 1 | 5 | 3 | 1 | 8 | 0 | 8 | 1 | 7 | 1 |
| 9 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 2 | 1 | 1 | 5 | 3 | 1 | 8 | 0 | 8 | 1 | 8 | 1 |
| 10 | 8 | 8 | 1 | 3 | 5 | 1 | 1 | 2 | 2 | 1 | 1 | 5 | 3 | 1 | 8 | 0 | 8 | 1 | 9 | 1 |

Table 5: Memory for task **Reverse**.

### C.4 SWAP

For swap we observed that 2 different circuits are generated, one for even timesteps, one for odd timesteps.

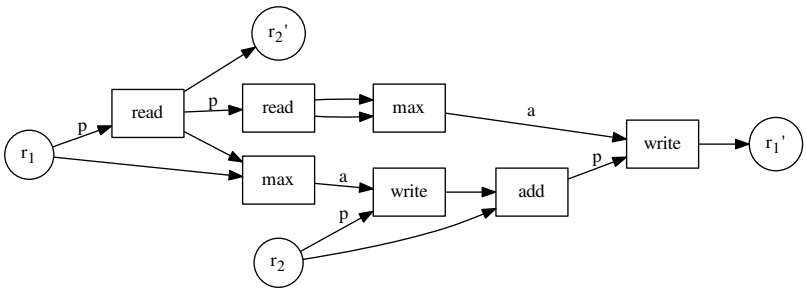



Figure 8: The circuit generated at every even timestep for the task **Swap**.

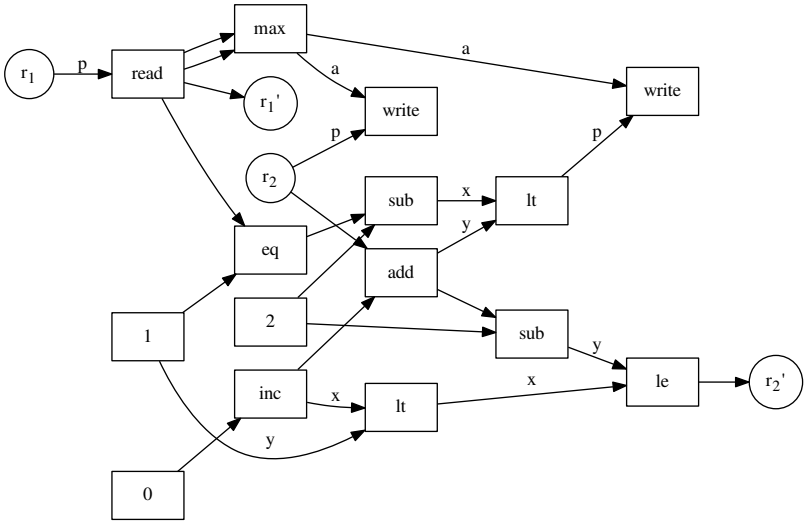



Figure 9: The circuit generated at every odd timestep $\geq 3$ for the task **Swap**.

| **Step** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $r_1$ | $r_2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 13 | 6 | 10 | 5 | 4 | 6 | 3 | 7 | 1 | 1 | 11 | 13 | 12 | 0 | 0 | 0 | 0 |
| 2 | 5 | 13 | 6 | 10 | 5 | 4 | 6 | 3 | 7 | 1 | 1 | 11 | 13 | 12 | 0 | 0 | 1 | 4 |
| 3 | 5 | 13 | 6 | 10 | 12 | 4 | 6 | 3 | 7 | 1 | 1 | 11 | 13 | 12 | 0 | 0 | 0 | 13 |
| 4 | 5 | 13 | 6 | 10 | 12 | 4 | 6 | 3 | 7 | 1 | 1 | 11 | 13 | 5 | 0 | 0 | 5 | 1 |

Table 6: Memory for task **Swap**.

# Learning Efficient Algorithms with Hierarchical Attentive Memory

**Marcin Andrychowicz**[*]
Google Deepmind

**Karol Kurach**[*]
Google / University of Warsaw

## Abstract

In this paper, we propose and investigate a novel memory architecture for neural networks called Hierarchical Attentive Memory (HAM). It is based on a binary tree with leaves corresponding to memory cells. This allows HAM to perform memory access in $\Theta(\log n)$ complexity, which is a significant improvement over the standard attention mechanism that requires $\Theta(n)$ operations, where $n$ is the size of the memory. We show that an LSTM network augmented with HAM can learn algorithms for problems like merging, sorting or binary searching from pure input-output examples. In particular, it learns to sort $n$ numbers in time $\Theta(n \log n)$ and generalizes well to input sequences much longer than the ones seen during the training. We also show that HAM can be trained to act like classic data structures: a stack, a FIFO queue and a priority queue.

## 1 Intro

Deep Recurrent Neural Networks (RNNs) have recently proven to be very successful in real-word tasks, e.g. machine translation (Sutskever et al., 2014) and computer vision (Vinyals et al., 2014). However, the success has been achieved only on tasks which do not require a large memory to solve the problem, e.g. we can translate sentences using RNNs, but we cannot produce reasonable translations of really long pieces of text, like books.

A high-capacity memory is a crucial component necessary to deal with large-scale problems that contain plenty of long-range dependencies. Currently used RNNs do not scale well to larger memories, e.g. the number of parameters in an LSTM (Hochreiter & Schmidhuber, 1997) grows quadratically with the size of the network's memory. In practice, this limits the number of used memory cells to few thousands.

It would be desirable for the size of the memory to be independent of the number of model parameters. The first versatile and highly successful architecture with this property was Neural Turing Machine (NTM) proposed by Graves et al. (2014). The main idea behind the NTM is to split the network into a trainable "controller" and an "external" variable-size memory. It caused an outbreak of other neural network architectures with external memories (see Sec. 2).

However, one aspect which has been usually neglected so far is the efficiency of the memory access. Most of the proposed memory architectures have the $\Theta(n)$ access complexity, where $n$ is the size of the memory. It means that, for instance, copying a sequence of length $n$ requires performing $\Theta(n^2)$ operations, which is clearly unsatisfactory.

### 1.1 Our contribution

We propose a novel memory module for neural networks, called Hierarchical Attentive Memory (HAM). The HAM module is generic and can be used as a building block of larger neural architectures.

[*]Equal contribution.

Submitted to 29th Conference on Neural Information Processing Systems (NIPS 2016). Do not distribute.

Its crucial property is that it scales well with the memory size — the memory access requires only $\Theta(\log n)$ operations, where $n$ is the size of the memory. This complexity is achieved by using a new attention mechanism based on a binary tree with leaves corresponding to memory cells. The novel attention mechanism is not only faster than the standard one used in Deep Learning (Bahdanau et al., 2014), but it also facilities learning algorithms due to a built-in bias towards operating on intervals.

We show that an LSTM augmented with HAM is able to learn algorithms for tasks like merging, sorting or binary searching. In particular, it is the first neural network, which we are aware of, that is able to learn to sort from pure input-output examples and generalizes well to input sequences much longer than the ones seen during the training. Moreover, the learned sorting algorithm runs in time $\Theta(n \log n)$. We also show that the HAM memory itself is capable of simulating different classic memory structures: a stack, a FIFO queue and a priority queue.

## 2 Related work

In this section we mention a number of recently proposed neural architectures with an external memory, which size is independent of the number of the model parameters.

**Memory architectures based on attention** Attention is a recent but already extremely successful technique in Deep Learning. This mechanism allows networks to *attend* to parts of the (potentially preprocessed) input sequence (Bahdanau et al., 2014) while generating the output sequence. It is implemented by giving the network as an auxiliary input a linear combination of input symbols, where the weights of this linear combination can be controlled by the network. Attention mechanism was used to access the memory in Neural Turing Machines (NTMs) proposed by Graves et al. (2014). It was the first paper, that explicitly attempted to train a computationally universal neural network and achieved encouraging results.

The Memory Network (Weston et al., 2014) is an early model that attempted to explicitly separate the memory from computation in a neural network model. The followup work of (Sukhbaatar et al., 2015) combined the memory network with the soft attention mechanism, which allowed it to be trained with less supervision. In contrast to NTMs, the memory in these models is non-writeable. Another model without writeable memory is the Pointer Network (Vinyals et al., 2015), which is very similar to the attention model of Bahdanau et al. (2014). Despite not having a memory, this model was able to solve a number of difficult algorithmic problems, like the Convex Hull and the approximate 2D TSP.

All of the architectures mentioned so far use standard attention mechanisms to access the memory and therefore memory access complexity scales linearly with the memory size.

**Memory architectures based on data structures** Stack-Augmented Recurrent Neural Network (Joulin & Mikolov, 2015) is a neural architecture combining an RNN and a differentiable stack. Grefenstette et al. (2015) consider extending an LSTM with a stack, a FIFO queue or a double-ended queue and show some promising results. The advantage of the latter model is that the presented data structures have a constant access time.

**Memory architectures based on pointers** In two recent papers (Zaremba & Sutskever, 2015; Zaremba et al., 2015) authors consider extending neural networks with nondifferentiable memories based on pointers and trained using Reinforcement Learning. The big advantage of these models is that they allow a constant time memory access. They were however only successful on relatively simple tasks.

Another model, which use a pointer-based memory and learns sub-procedures is the Neural Programmer-Interpreter (Reed & de Freitas, 2015). Unfortunately, it requires strong supervision in the form of execution traces. Different type of pointer-based memory was presented in Neural Random-Access Machine (Kurach et al., 2015), which is a neural architecture mimicking classic computers.

**Parallel memory architectures** There are two recent memory architectures, which are especially suited for parallel computation. Grid-LSTM (Kalchbrenner et al., 2015) is an extension of LSTM to multiple dimensions. Another recent model of this type is Neural GPU (Kaiser & Sutskever, 2015), which can learn to multiply long binary numbers.

# 3 Hierarchical Attentive Memory

In this section we describe our novel memory module called Hierarchical Attentive Memory (HAM). The HAM module is generic and can be used as a building block of larger neural network architectures. For instance, it can be added to feedforward or LSTM networks to extend their capabilities. To make our description more concrete we will consider a model consisting of an LSTM "controller" extended with a HAM module.

The high-level idea behind the HAM module is as follows. The memory is structured as a full binary tree with the leaves containing the data stored in the memory. The inner nodes contain some auxiliary data, which allows us to efficiently perform some types of "queries" on the memory. In order to access the memory, one starts from the root of the tree and performs a top-down descent in the tree, which is similar to the hierarchical softmax procedure (Morin & Bengio, 2005). At every node of the tree, one decides to go left or right based on the auxiliary data stored in this node and a "query". Details are provided in the rest of this section.

## 3.1 Notation

The model takes as input a sequence $x_1, x_2, \ldots$ and outputs a sequence $y_1, y_2, \ldots$. We assume that each element of these sequences is a binary vector of size $b \in \mathbb{N}$, i.e. $x_i, y_i \in \{0,1\}^b$. Suppose for a moment that we only want to process input sequences of length $\leq n$, where $n \in \mathbb{N}$ is a power of two (we show later how to process sequences of an arbitrary length). The model is based on the full binary tree with $n$ leaves. Let $V$ denote the set of the nodes in that tree (notice that $|V| = 2n - 1$) and let $L \subset V$ denote the set of its leaves. Let $l(e)$ for $e \in V \setminus L$ be the left child of the node $e$ and let $r(e)$ be its right child. We will now present the inference procedure for the model and then discuss how to train it.

## 3.2 Inference

The high-level view of the model execution is presented in Fig. 1. The hidden state of the model consists of two components: the hidden state of the LSTM controller (denoted $h_{\texttt{LSTM}} \in \mathbb{R}^l$ for some $l \in \mathbb{N}$) and the hidden values stored in the nodes of the HAM tree. More precisely, for every node $e \in V$ there is a hidden value $h_e \in \mathbb{R}^d$. These values change during the recurrent execution of the model, but we drop all timestep indices to simplify the notation.

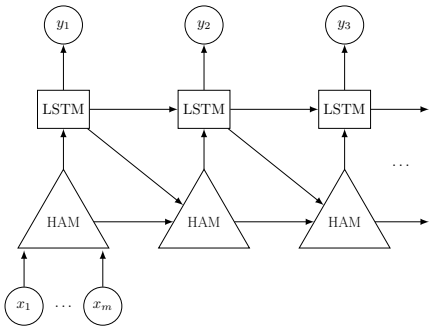
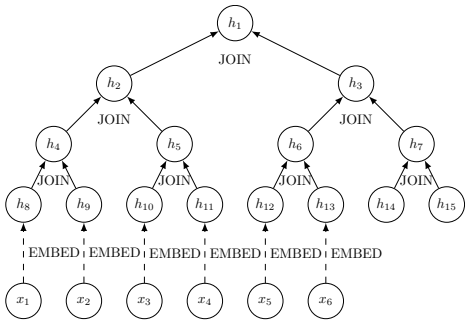



Figure 1: The LSTM+HAM model consists of an LSTM controller and a HAM module. The execution of the model starts with the initialization of HAM using the *whole* input sequence $x_1, x_2, \ldots, x_m$. At each timestep, the HAM module produces an input for the LSTM, which then produces an output symbol $y_t$. Afterwards, the hidden states of the LSTM and HAM are updated.

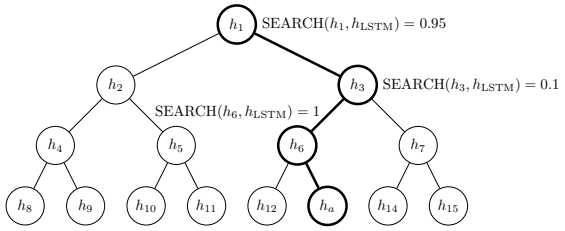
The parameters of the model describe the input-output behaviour of the LSTM, as well as the following $4$ transformations, which describe the HAM module: EMBED : $\mathbb{R}^b \to \mathbb{R}^d$, JOIN : $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$, SEARCH : $\mathbb{R}^d \times \mathbb{R}^l \to [0,1]$ and WRITE : $\mathbb{R}^d \times \mathbb{R}^l \to \mathbb{R}^d$. These transformations may be represented by arbitrary function approximators, e.g. Multilayer Perceptrons (MLPs). Their meaning will be described soon.

The details of the model are presented in $4$ figures. Fig. 2a describes the initialization of the model. Each recurrent timestep of the model consists of three phases: the *attention* phase described in Fig. 2b, the *output* phase described in Fig. 2c and the *update* phase described in Fig. 2d. The whole timestep can be performed in time $\Theta(\log n)$.

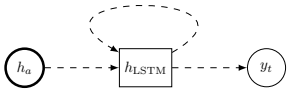
The HAM parameters describe only the $4$ mentioned transformations and hence the number of the model parameters does not depend on the size of the binary tree used. Thus, we can use the model to process the inputs of an arbitrary length by using big enough binary trees. It is not clear that the same set of parameters will give good results across different tree sizes, but we showed experimentally that it is indeed the case (see Sec. 4 for more details).



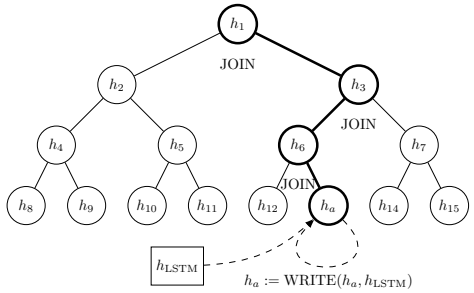

(a) Initialization of the model. The value in the $i$-th leaf of HAM is initialized with EMBED$(x_i)$, where EMBED is a trainable feed-forward network. If there are more leaves than input symbols, we initialize the values in the excessive leaves with zeros. Then, we initialize the values in the inner nodes bottom-up using the formula $h_e = \text{JOIN}(h_{l(e)}, h_{r(e)})$. The hidden state of the LSTM — $h_{\text{LSTM}}$ is initialized with zeros.




(b) *Attention* phase. In this phase the model performs a top-down "search" in the tree starting from the root. Suppose that we are currently at the node $c \in V \setminus L$. We compute the value $p = \text{SEARCH}(h_c, h_{\text{LSTM}})$. Then, with probability $p$ the model goes right (i.e. $c := r(c)$) and with probability $1 - p$ it goes left (i.e. $c := l(c)$). This procedure is continued until we reach one of the leaves. This leaf is called the *attended* or *accessed* leaf and denoted $a$.




(c) *Output* phase. The value $h_a$ stored in the attended leaf is given to the LSTM as an input. Then, the LSTM produces an output symbol $y_t \in \{0, 1\}^b$. More precisely, the value $u \in \mathbb{R}^b$ is computed by a trainable linear transformation from $h_{\text{LSTM}}$ and the distribution of $y_t$ is defined by the formula $p(y_{t,i} = 1) = \mathbf{sigmoid}(u_i)$ for $1 \leq i \leq b$. It may be beneficial to allow the model to access the memory a few times between producing each output symbols. Therefore, the model produces an output symbol only at timesteps with indices divisible by some constant $\eta \in \mathbb{N}$, which is a hyperparameter.




(d) *Update* phase. In this phase the value in the attended leaf $a$ is updated. More precisely, the value is modified using the formula $h_a := \text{WRITE}(h_a, h_{\text{LSTM}})$. Then, we update the values of the inner nodes encountered during the attention phase ($h_6, h_3$ and $h_1$ in the figure) bottom-up using the equation $h_e = \text{JOIN}(h_{l(e)}, h_{r(e)})$.

Figure 2: The model. One timestep consists of three phases presented in Figures (b)–(d).

We decided to represent the transformations defining HAM with MLPs with ReLU (Nair & Hinton, 2010) activation function in all neurons except the output layer of SEARCH, which uses sigmoid activation function to ensure that the output may be interpreted as a probability. Moreover, the network for WRITE is enhanced in a similar way as Highway Networks (Srivastava et al., 2015), i.e. $\text{WRITE}(h_a, h_{\text{LSTM}}) = T(h_a, h_{\text{LSTM}}) \cdot H(h_a, h_{\text{LSTM}}) + (1 - T(h_a, h_{\text{LSTM}})) \cdot h_a$, where $H$ and $T$ are two MLPs with sigmoid activation function in the output layer. This allows the WRITE transformation to easily leave the value $h_a$ unchanged.

### 3.3 Training

In this section we describe how to train our model from purely input-output examples using REINFORCE (Williams, 1992). In Appendix C we also present a different variant of HAM which is fully differentiable and can be trained using end-to-end backpropagation.

Let $x, y$ be an input-output pair. Recall that both $x$ and $y$ are sequences. Moreover, let $\theta$ denote the parameters of the model and let $A$ denote the sequence of all decisions whetherto go left or right made during *the whole execution* of the model. We would like to maximize the log-probability of producing the correct output, i.e.

$$\mathcal{L} = \log p(y|x,\theta) = \log\left(\sum_A p(A|x,\theta)p(y|A,x,\theta)\right).$$

This sum is intractable, so instead of minimizing it directly, we minimize a variational lower bound on it:

$$\mathcal{F} = \sum_A p(A|x,\theta)\log p(y|A,x,\theta) \leq \mathcal{L}.$$

This sum is also intractable, so we approximate its gradient using the REINFORCE, which we briefly explain below. Using the identity $\nabla p(A|x,\theta) = p(A|x,\theta)\nabla \log p(A|x,\theta)$, the gradient of the lower bound with respect to the model parameters can be rewritten as:

$$\nabla\mathcal{F} = \sum_A p(A|x,\theta)\Big[\nabla \log p(y|A,x,\theta) + \log p(y|A,x,\theta)\nabla \log p(A|x,\theta)\Big] \tag{1}$$

We estimate this value using Monte Carlo approximation.For every $x$ we sample $\widetilde{A}$ from $p(A|x,\theta)$ and approximate the gradient for the input $x$ as $\nabla \log p(y|\widetilde{A},x,\theta) + \log p(y|\widetilde{A},x,\theta)\nabla \log p(\widetilde{A}|x,\theta)$. Notice that this gradient estimate can be computed using normal backpropagation if we substitute the gradients in the nodes[2] which sample whether we should go left or right during the *attention* phase by

$$\underbrace{\log p(y|\widetilde{A},x,\theta)}_{\text{return}}\nabla \log p(\widetilde{A}|x,\theta).$$

This term is called REINFORCE gradient estimate and the left factor is called a *return* in Reinforcement Learning literature. This gradient estimator is unbiased, but it often has a high variance. Therefore, we employ two standard variance-reduction technique for REINFORCE: *discounted returns* and *baselines* (Williams, 1992). Discounted returns means that our return at the $t$-th timestep has the form $\sum_{t\leq i}\gamma^{i-t}\log p(y_i|\widetilde{A},x,\theta)$ for some discount constant $\gamma \in [0,1]$, which is a hyperparameter. This biases the estimator if $\gamma < 1$, but it often decreases its variance.

For the lack of space we do not describe the *baselines* technique. We only mention that our baseline is case and timestep dependent: it is computed using a learnable linear transformation from $h_{\text{LSTM}}$ and trained using MSE loss function. The whole model is trained with the Adam (Kingma & Ba, 2014) algorithm. We also employ the following three training techniques:

**Different reward function** During our experiments we noticed that better results may be obtained by using a different reward function for REINFORCE. More precisely, instead of the log-probability of producing the correct output, we use the percentage of the output bits, which have the probability of being predicted correctly (given $\widetilde{A}$) greater than $50\%$, i.e. our discounted return is equal $\sum_{t\leq i,1\leq j\leq b}\gamma^{i-t}\Big[p(y_{i,j}|\widetilde{A},x,\theta) > 0.5\Big]$. Notice that it corresponds to the Hamming distance between the most probable outcome accordingly to the model (given $\widehat{A}$) and the correct output.

**Entropy bonus term** We add a special term to the cost function which encourages exploration. More precisely, for each sampling node we add to the cost function the term $\frac{\alpha}{H(p)}$, where $H(p)$ is the entropy of the distribution of the decision, whether to go left or right in this node and $\alpha$ is an exponentially decaying coefficient. This term goes to infinity, whenever the entropy goes to zero, what ensures some level of exploration. We noticed that this term works better in our experiments than the standard term of the form $-\alpha H(p)$ (Williams, 1992).

**Curriculum schedule** We start with training on inputs with lengths sampled uniformly from $[1,n]$ for some $n = 2^k$ and the binary tree with $n$ leaves. Whenever the error drops below some threshold, we increment the value $k$ and start using the bigger tree with $2n$ leaves and inputs with lengths sampled uniformly from $[1,2n]$.

[2] For a general discussion of computing gradients in computation graphs, which contain stochastic nodes see (Schulman et al., 2015).

# 4 Experiments

In this section, we evaluate two variants of using the HAM module. The first one is the model described in Sec. 3, which combines an LSTM controller with a HAM module (denoted by LSTM+HAM). Then, in Sec. 4.3 we investigate the "raw" HAM (without the LSTM controller) to check its capability of acting as classic data structures: a stack, a FIFO queue and a priority queue. It would be also interesting to get some insight into the algorithms learned by the model. In Appendix A we present an example execution on the `Sort` task.

## 4.1 Test setup

For each test that we perform, we apply the following procedure. First, we train the model with memory of size up to $n = 32$ using the curriculum schedule described in Sec. 3.3. The model is trained using the minibatch Adam algorithm with exponentially decaying learning rate. We use random search to determine the best hyper-parameters for the model. We use gradient clipping (Pascanu et al., 2012) with constant $5$. The depth of our MLPs is either $1$ or $2$, the LSTM controller has $l = 20$ memory cells and the hidden values in the tree have dimensionality $d = 20$. Constant $\eta$ determining a number of memory accesses between producing each output symbols (Fig. 2c) is equal either $1$ or $2$. We always train for $100$ epochs, each consisting of $1000$ batches of size $50$. After each epoch we evaluate the model on $200$ validation batches without learning. When the training is finished, we select the model parameters that gave the lowest error rate on validation batches and report the error using these parameters on fresh $2,500$ random examples.

We report two types of errors: a test error and a generalization error. The test error shows how well the model is able to fit the data distribution and generalize to unknown cases, assuming that cases of similar lengths were shown during the training. It is computed using the HAM memory with $n = 32$ leaves, as the percentage of output *sequences*, which were predicted incorrectly. The lengths of test examples are sampled uniformly from the range $[1, n]$. Notice that we mark the whole output sequence as incorrect even if only one bit was predicted incorrectly, e.g. a hypothetical model predicting each bit incorrectly with probability $1\%$ (and independently of the errors on the other bits) has an error rate of $96\%$ on *whole sequences* if outputs consist of $320$ bits.

The generalization error shows how well the model performs with enlarged memory on examples with lengths exceeding $n$. We test our model with memory $4$ times bigger than the training one. The lengths of input sequences are now sampled uniformly from the range $[2n + 1, 4n]$.

During testing we make our model fully deterministic by using the most probable outcomes instead of stochastic sampling. More precisely, we assume that during the *attention phase* the model decides to go right iff $p > 0.5$ (Fig. 2b). Moreover, the output symbols (Fig. 2c) are computed by rounding to zero or one instead of sampling.

## 4.2 LSTM+HAM

We evaluate the model on a number of algorithmic tasks described below:

1. `Reverse`: Given a sequence of 10-bit vectors, output them in the reversed order., i.e. $y_i = x_{m+1-i}$ for $1 \leq i \leq m$, where $m$ is the length of the input sequence.
2. `Search`: Given a sequence of pairs $x_i = \textbf{key}_i || \textbf{value}_i$ for $1 \leq i \leq m - 1$ sorted by keys and a query $x_m = q$, find the smallest $i$ such that $\textbf{key}_i = q$ and output $y_1 = \textbf{value}_i$. Keys and values are $5$-bit vectors and keys are compared lexicographically. The LSTM+HAM model is given only two timesteps ($\eta = 2$) to solve this problem, which forces it to use a form of binary search.
3. `Merge`: Given two *sorted* sequences of pairs — $(p_1, v_1), \ldots, (p_m, v_m)$ and $(p'_1, v'_1), \ldots, (p'_{m'}, v'_{m'})$, where $p_i, p'_i \in [0, 1]$ and $v_i, v'_i \in \{0, 1\}^5$, merge them. Pairs are compared accordingly to their priorities, i.e. values $p_i$ and $p'_i$. Priorities are unique and sampled uniformly from the set $\{\frac{1}{300}, \ldots, \frac{300}{300}\}$, because neural networks cannot easily distinguish two real numbers which are very close to each other. Input is encoded as $x_i = p_i || v_i$ for $1 \leq i \leq m$ and $x_{m+i} = p'_i || v'_i$ for $1 \leq i \leq m'$. The output consists of the vectors $v_i$ and $v'_i$ sorted accordingly to their priorities[3].

[3] Notice that we earlier assumed for the sake of simplicity that the input sequences consist of *binary* vectors and in this task the priorities are *real* values. It does not however require any change of our model. We decided to use real priorities in this task in order to diversify our set of problems.

4. `Sort`: Given a sequence of pairs $x_i = \textbf{key}_i||\textbf{value}_i$ sort them in a stable way[4] accordingly to the lexicographic order of the keys. Keys and values are 5-bit vectors.

5. `Add`: Given two numbers represented in binary, compute their sum. The input is represented as $a_1, \ldots, a_m, \textbf{+}, b_1, \ldots, b_m, \textbf{=}$ (i.e. $x_1 = a_1, x_2 = a_2$ and so on), where $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$ are bits of the input numbers and **+**, **=** are some special symbols. Input and output numbers are encoded starting from the *least* significant bits.

Every example output shown during the training is finished by a special "End Of Output" symbol, which the model learns to predict. It forces the model to learn not only the output symbols, but also the length of the correct output.

We compare our model with 2 strong baseline models: encoder-decoder LSTM (Sutskever et al., 2014) and encoder-decoder LSTM with attention (Bahdanau et al., 2014), denoted LSTM+A. The number of the LSTM cells in the baselines was chosen in such a way, that they have more parameters than the biggest of our models. We also use random search to select an optimal learning rate and some other parameters for the baselines and train them using the same curriculum scheme as LSTM+HAM.

The results are presented in Table 1. Not only, does LSTM+HAM solve all the problems almost perfectly, but it also generalizes very well to much longer inputs on all problems except `Add`. Recall that for the generalization tests we used a HAM memory of a different size than the ones used during the training, what shows that HAM generalizes very well to new sizes of the binary tree. We find this fact quite interesting, because it means that parameters learned from a small neural network (i.e. HAM based on a tree with 32 leaves) can be successfully used in a different, bigger network (i.e. HAM with 128 memory cells).

Table 1: Experimental results. The upper table presents the error rates on inputs of the same lengths as the ones used during training. The lower table shows the error rates on input sequences 2 to 4 times longer than the ones encountered during training. LSTM+A denotes an LSTM with the standard attention mechanism. Each error rate is a percentage of *output sequences*, which contained at least one incorrectly predicted bit.

| **test error** | LSTM | LSTM+ A | **LSTM+ HAM** |
|---|---|---|---|
| `Reverse` | 73% | 0% | **0%** |
| `Search` | 62% | 0.04% | **0.12%** |
| `Merge` | 88% | 16% | **0%** |
| `Sort` | 99% | 25% | **0.04%** |
| `Add` | 39% | 0% | **0%** |
| **2-4x longer inputs** | LSTM | LSTM+ A | **LSTM+ HAM** |
| `Reverse` | 100% | 100% | **0%** |
| `Search` | 89% | 0.52% | **1.68%** |
| `Merge` | 100% | 100% | **2.48%** |
| `Sort` | 100% | 100% | **0.24%** |
| `Add` | 100% | 100% | **100%** |
| **Complexity** | $\Theta(1)$ | $\Theta(n)$ | $\boldsymbol{\Theta}(\textbf{log n})$ |

In comparison, the LSTM with attention does not learn to merge, nor sort. It also completely fails to generalize to longer examples, which shows that LSTM+A learns rather some statistical dependencies between inputs and outputs than the real algorithms.

The LSTM+HAM model makes a few errors when testing on longer outputs than the ones encountered during the training. Notice however, that we show in the table the percentage of output sequences, which contain *at least one* incorrect bit. For instance, LSTM+HAM on the problem `Merge` predicts incorrectly only $0.03\%$ of output bits, which corresponds to $2.48\%$ of incorrect output sequences. We believe that these rare mistakes could be avoided if one trained the model longer and chose carefully the learning rate schedule. One more way to boost generalization would be to simultaneously train the models with different memory sizes and shared parameters. We have not tried this as the generalization properties of the model were already very good.

### 4.3 Raw HAM

In this section, we evaluate "raw" HAM module (without the LSTM controller, see Appendix B for details) to see if it can act as a drop-in replacement for 3 classic data structures: a stack, a FIFO queue and a priority queue. For each task, the network is given a sequence of PUSH and POP operations in an *online* manner: at timestep $t$ the network sees only the $t$-th operation to perform $x_t$. This is a more realistic scenario for data structures usage as it prevents the network from cheating by peeking into the future. We evaluate raw HAM on the following tasks:

[4]Stability means that pairs with equal keys should be ordered accordingly to their order in the input sequence.

1. `Stack`: The "PUSH $x$" operation places the element $x$ (a 5-bit vector) on top of the stack, and the "POP" returns the last added element and removes it from the stack.
2. `Queue`: The "PUSH $x$" operation places the element $x$ (a 5-bit vector) at the end of the queue and the "POP" returns the oldest element and removes it from the queue.
3. `PriorityQueue`: The "PUSH $x$ $p$" operations adds the element $x$ with priority $p$ to the queue. The "POP" operation returns the value with the highest priority and remove it from the queue. Both $x$ and $p$ are represented as 5-bit vectors and priorities are compared lexicographically. To avoid ties we assume that all elements have different priorities.

Model was trained with the memory of size up to $n = 32$ with operation sequences of length $n$. Sequences of PUSH/POP actions for training were selected randomly. The $t$-th operation out of $n$ operations in the sequence was POP with probability $\frac{t}{n}$ and PUSH otherwise. To test generalization, we report the error rates with the memory of size $4n$ on sequences of operations of length $4n$.

Table 2: Results of experiments with the raw version of HAM (without the LSTM controller). Error rates are measured as a percentage of operation sequences in which *at least one* POP query was not answered correctly.

| Task | Test Error | Generalization Error |
|---|---|---|
| `Stack` | 0% | 0% |
| `Queue` | 0% | 0% |
| `Priority Queue` | 0.08% | 0.2% |

The results presented in Table 2 show that HAM simulates a stack and a queue perfectly with no errors whatsoever even for memory $4$ times bigger. For the `PriorityQueue` task, the model generalizes almost perfectly to large memory, with errors only in $0.2\%$ of output sequences.

## 5 Comparison to other models

As far as we know, our model is the first one which is able to learn a sorting algorithm from pure input-output examples. Although this problem was considered in the original NTM paper, the error rate achieved by the NTM is in fact quite high – the log-likelihood of the correct output was equal around 20 bits on outputs consisting of 128 bits. In comparison our model learns to solve almost perfectly - only $0.04\%$ of the outputs produced by our model contain at least one incorrect bit.

Reed & de Freitas (2015) shown that an LSTM is able to learn to sort short sequences, but it fails to generalize to inputs longer than the ones seen during the training. It is quite clear that an LSTM cannot learn a "real" sorting algorithm, because it uses a bounded memory independent of the length of the input. The Neural Programmer-Interpreter (Reed & de Freitas, 2015) is a neural network architecture, which is able to learn bubble sort, but it requires strong supervision in the form of execution traces. In comparison, our model can be trained from pure input-output examples, which is crucial if we want to use it to solve problems for which we do not know any algorithms.

An important feature of neural memories is their efficiency. Our HAM module in comparison to many other recently proposed solutions is effective and allows to access the memory in $\Theta(\log(n))$ complexity. In the context of learning algorithms it may sound surprising that among all the architectures mentioned in Sec. 2 the only ones, which can copy a sequence of length $n$ without $\Theta(n^2)$ operations are: Reinforcement-Learning NTM (Zaremba & Sutskever, 2015), the model from (Zaremba et al., 2015), Neural Random-Access Machine (Kurach et al., 2015), and Queue-Augmented LSTM (Grefenstette et al., 2015). However, the first three models have been only successful on relatively simple tasks. The last model was successful on some synthetic tasks from the domain of Natural Language Processing, which are very different from the tasks we tested our model on, so we cannot directly compare the two models.

## 6 Conclusions

We presented a new memory architecture for neural networks called Hierarchical Attentive Memory. Its crucial property is that it scales well with the memory size — the memory access requires only $\Theta(\log n)$ operations. This complexity is achieved using a new attention mechanism based on a binary tree. The model proved to be successful on a number of algorithmic problems. The future work is to apply this or similar architecture to very long real-world sequential data like books or DNA sequences.

# References


Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1819–1827, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv preprint arXiv:1503.01007*, 2015.

Kaiser, Łukasz and Sutskever, Ilya. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Kalchbrenner, Nal, Danihelka, Ivo, and Graves, Alex. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Morin, Frederic and Bengio, Yoshua. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pp. 246–252. Citeseer, 2005.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. Understanding the exploding gradient problem. *Computing Research Repository (CoRR) abs/1211.5063*, 2012.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Schulman, John, Heess, Nicolas, Weber, Theophane, and Abbeel, Pieter. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pp. 3510–3522, 2015.

Srivastava, Rupesh Kumar, Greff, Klaus, and Schmidhuber, Jürgen. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc VV. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Weston, Jason, Chopra, Sumit, and Bordes, Antoine. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Williams, Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.

Zaremba, Wojciech, Mikolov, Tomas, Joulin, Armand, and Fergus, Rob. Learning simple algorithms from examples. *arXiv preprint arXiv:1511.07275*, 2015.

# A Example: HAM sorting

We present some insights into the algorithms learned by the LSTM+HAM model, by investigating the hidden representations $h_e$ learned for a variant of the problem `Sort` in which we sort 4-bit vectors lexicographically[5]. For demonstration purposes, we use a small tree with $n = 8$ leaves and each node's hidden state has size $d = 6$ values.

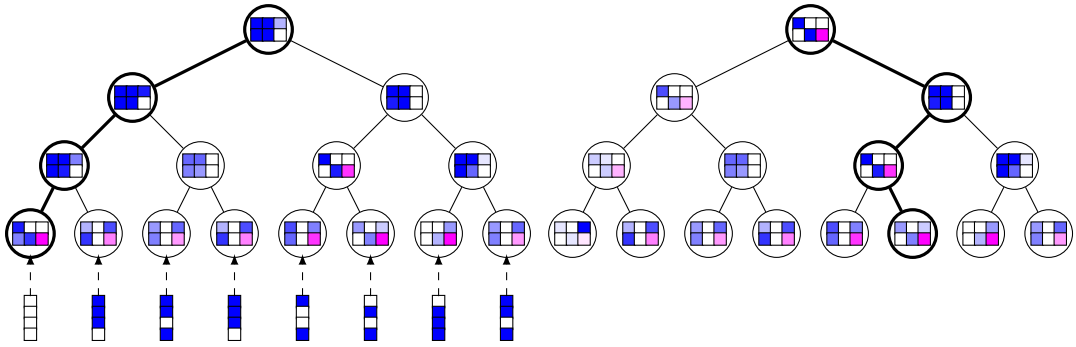
The trained network performs sorting perfectly. It attends to the leaves in the order corresponding to the order of the sorted input values, i.e. at every timestep HAM attends to the leaf corresponding to the smallest input value among the leaves, which have not been attended so far.

It would be interesting to exactly understand the algorithm used by the network to perform this operation. A natural solution to this problem would be to store in each hidden node $e$ the smallest input value among the (unattended so far) leaves *below* $e$ together with the information whether the smallest value is in the right or the left subtree under $e$.

In the Fig. 3 we present two timesteps of our model. The LSTM controller is not presented to simplify the exposition. The input sequence is presented on the left, below the tree: $x_1 = \texttt{0000}, x_2 = \texttt{1110}, x_3 = \texttt{1101}$ and so on. The 2x3 grids in the nodes of the tree represent the values $h_e \in \mathbb{R}^6$. White cells correspond to value 0 and non-white cells correspond to values $> 0$.

The lower-rightmost cells are presented in pink, because we managed to decipher the meaning of this coordinate for the inner nodes. This coordinate in the node $e$ denotes whether the minimum in the subtree (among the values unattended so far) is in the right or left subtree of $e$. Value greater than 0 (pink in the picture) means that the minimum is in the right subtree and therefore we should go right while visiting this node in the *attention* phase.

In the first timestep the leftmost leaf (corresponding to the input `0000`) is accessed. Notice that the last coordinates (shown in pink) are updated appropriately, e.g. the smallest unattended value at the beginning of the second timestep is `0101`, which corresponds to the 6-th leaf. It is in the right subtree under the root and accordingly the last coordinate in the hidden value stored in the root is high (i.e. pink in the figure).



(a) The first timestep (b) The second timestep

Figure 3: An exemplary input sequence and the state of HAM after initialization (left) and after first timestep (right).

[5] In the problem `Sort` considered in the experimental results, there are separate keys and values, which forces the model to learn stable sorting. Here, for the sake of simplicity, we consider the simplified version of the problem and do not use separate keys and values.

## B Raw HAM details

Raw HAM module differs from the LSTM+HAM model from Sec. 3 in the following way:

- The HAM memory is initialized with zeros.
- The $t$-th output symbol $y_t$ is computed using an MLP from the value in the accessed leaf $h_a$.
- Notice that in the LSTM+HAM model, $h_{\text{LSTM}}$ acted as a kind of "query" or "command" guiding the behaviour of HAM. We will now use the values $x_t$ instead. Therefore, at the $t$-th timestep we use $x_t$ instead of $h_{\text{LSTM}}$ whenever $h_{\text{LSTM}}$ was used in the original model, e.g. during the *attention* phase (Fig. 2b) we use $p = \text{SEARCH}(h_c, x_t)$ instead of $p = \text{SEARCH}(h_c, h_{\text{LSTM}})$.

## C Using soft attention

One of the open questions in the area of designing neural networks with attention mechanisms is whether to use a *soft* or *hard* attention. The model described in the paper belongs to the latter class of attention mechanisms as it makes hard, stochastic choices. The other solution would be to use a soft, differentiable mechanism, which attends to a linear combination of the potential attention targets and do not involve any sampling. The main advantage of such models is that their gradients can be computed exactly.

We now describe how to modify the model to make it fully differentiable ("DHAM"). Recall that in the original model the leaf which is attended at every timestep is sampled stochastically. Instead of that, we will now at every timestep compute for every leaf $e$ the probability $p(e)$ that this leaf would be attended if we used the stochastic procedure described in Fig. 2b. The value $p(e)$ can be computed by multiplying the probabilities of going in the right direction from all the nodes on the path from the root to $e$.

As the input for the LSTM we then use the value $\sum_{e \in L} p(e) \cdot h_e$. During the *write* phase, we update the values of *all* the leaves using the formula $h_e := p(e) \cdot \text{WRITE}(h_e, h_{\text{ROOT}}) + (1 - p(e)) \cdot h_e$. Then, in the *update* phase we update the values of *all* the inner nodes, so that the equation $h_e = \text{JOIN}(h_{l(e)}, h_{r(e)})$ is satisfied for each inner node $e$. Notice that one timestep of the soft version of the model takes time $\Theta(n)$ as we have to update the values of all the nodes in the tree. Our model may be seen as a special case of Gated Graph Neural Network (Li et al., 2015).

This version of the model is fully differentiable and therefore it can be trained using end-to-end backpropagation on the log-probability of producing the correct output. We observed that training DHAM is slightly easier than the REINFORCE version. However, DHAM does not generalize as well as HAM to larger memory sizes.

# ADDING GRADIENT NOISE IMPROVES LEARNING FOR VERY DEEP NETWORKS

**Arvind Neelakantan,* Luke Vilnis***
College of Information and Computer Sciences
University of Massachusetts Amherst
{arvind,luke}@cs.umass.edu

**Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach**
Google Brain
{qvl,ilyasu,lukaszkaiser,kkurach}@google.com

**James Martens**
University of Toronto
{jmartens}@cs.toronto.edu

## ABSTRACT

Deep feedforward and recurrent networks have achieved impressive results in many perception and language processing applications. This success is partially attributed to architectural innovations such as convolutional and long short-term memory networks. A major reason for these architectural innovations is that they capture better domain knowledge, and importantly are easier to optimize than more basic architectures. Recently, more complex architectures such as Neural Turing Machines and Memory Networks have been proposed for tasks including question answering and general computation, creating a new set of optimization challenges. In this paper, we discuss a low-overhead and easy-to-implement technique of adding gradient noise which we find to be surprisingly effective when training these very deep architectures. The technique not only helps to avoid overfitting, but also can result in lower training loss. This method alone allows a fully-connected 20-layer deep network to be trained with standard gradient descent, even starting from a poor initialization. We see consistent improvements for many complex models, including a 72% relative reduction in error rate over a carefully-tuned baseline on a challenging question-answering task, and a doubling of the number of accurate binary multiplication models learned across 7,000 random restarts. We encourage further application of this technique to additional complex modern architectures.

## 1 INTRODUCTION

Deep neural networks have shown remarkable success in diverse domains including image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012) and language processing applications (Sutskever et al., 2014; Bahdanau et al., 2014). This broad success comes from a confluence of several factors. First, the creation of massive labeled datasets has allowed deep networks to demonstrate their advantages in expressiveness and scalability. The increase in computing power has also enabled training of far larger networks with more forgiving optimization dynamics (Choromanska et al., 2015). Additionally, architectures such as convolutional networks (LeCun et al., 1998) and long short-term memory networks (Hochreiter & Schmidhuber, 1997) have proven to be easier to optimize than classical feedforward and recurrent models. Finally, the success of deep networks is also a result of the development of *simple* and *broadly applicable* learning techniques such as dropout (Srivastava et al., 2014), ReLUs (Nair & Hinton, 2010), gradient clipping (Pascanu

*First two authors contributed equally. Work was done when all authors were at Google, Inc.

et al., 2013; Graves, 2013), optimization and weight initialization strategies (Glorot & Bengio, 2010; Sutskever et al., 2013; He et al., 2015).

Recent work has aimed to push neural network learning into more challenging domains, such as question answering or program induction. These more complicated problems demand more complicated architectures (e.g., Graves et al. (2014); Sukhbaatar et al. (2015)) thereby posing new optimization challenges. In order to achieve good performance, researchers have reported the necessity of additional techniques such supervision in intermediate steps (Weston et al., 2014), warmstarts (Peng et al., 2015), random restarts, and the removal of certain activation functions in early stages of training (Sukhbaatar et al., 2015).

A recurring theme in recent works is that commonly-used optimization techniques are not always sufficient to robustly optimize the models. In this work, we explore a simple technique of adding annealed Gaussian noise to the gradient, which we find to be surprisingly effective in training deep neural networks with stochastic gradient descent. While there is a long tradition of adding random weight noise in classical neural networks, it has been under-explored in the optimization of modern deep architectures. In contrast to theoretical and empirical results on the regularizing effects of conventional stochastic gradient descent, we find that in practice the added noise can actually help us achieve lower training loss by encouraging active exploration of parameter space. This exploration proves especially necessary and fruitful when optimizing neural network models containing many layers or complex latent structures.

The main contribution of this work is to demonstrate the broad applicability of this simple method to the training of many complex modern neural architectures. Furthermore, to the best of our knowledge, our added noise schedule has not been used before in the training of deep networks. We consistently see improvement from injected gradient noise when optimizing a wide variety of models, including very deep fully-connected networks, and special-purpose architectures for question answering and algorithm learning. For example, this method allows us to escape a poor initialization and successfully train a 20-layer rectifier network on MNIST with standard gradient descent. It also enables a 72% relative reduction in error in question-answering, and doubles the number of accurate binary multiplication models learned across 7,000 random restarts. We hope that practitioners will see similar improvements in their own research by adding this simple technique, implementable in a single line of code, to their repertoire.

## 2 Related work

Adding random noise to the weights, gradient, or the hidden units has been a known technique amongst neural network practitioners for many years (e.g., An (1996)). However, the use of gradient noise has been rare and its benefits have not been fully documented with modern deep networks.

Weight noise (Steijvers, 1996) and adaptive weight noise (Graves, 2011; Blundell et al., 2015), which usually maintains a Gaussian variational posterior over network weights, similarly aim to improve learning by added noise during training. They normally differ slightly from our proposed method in that the noise is not annealed and at convergence will be non-zero. Additionally, in adaptive weight noise, an extra set of parameters for the variance must be maintained.

Similarly, the technique of dropout (Srivastava et al., 2014) randomly sets groups of hidden units to zero at train time to improve generalization in a manner similar to ensembling.

An annealed Gaussian gradient noise schedule was used to train the highly non-convex Stochastic Neighbor Embedding model in Hinton & Roweis (2002). The gradient noise schedule that we found to be most effective is very similar to the Stochastic Gradient Langevin Dynamics algorithm of Welling & Teh (2011), who use gradients with added noise to accelerate MCMC inference for logistic regression and independent component analysis models. This use of gradient information in MCMC sampling for machine learning to allow faster exploration of state space was previously proposed by Neal (2011).

Various optimization techniques have been proposed to improve the training of neural networks. Most notable is the use of Momentum (Polyak, 1964; Sutskever et al., 2013; Kingma & Ba, 2014) or adaptive learning rates (Duchi et al., 2011; Dean et al., 2012; Zeiler, 2012). These methods are normally developed to provide good convergence rates for the convex setting, and then heuristically

applied to nonconvex problems. Injecting noise in the gradient is more suitable for nonconvex problems. By adding even more stochasticity, this technique allows the model more chances to escape local minima (see a similar argument in Bottou (1992)), or to traverse quickly through the "transient" plateau phase of early learning (see a similar analysis for momentum in Sutskever et al. (2013)). This is born out empirically in our observation that adding gradient noise can actually result in lower training loss. In this sense, we suspect adding gradient noise is similar to simulated annealing (Kirkpatrick et al., 1983) which exploits random noise to explore complex optimization landscapes. This can be contrasted with well-known benefits of stochastic gradient descent as a learning algorithm (Robbins & Monro, 1951; Bousquet & Bottou, 2008), where both theory and practice have shown that the noise induced by the stochastic process aids generalization by reducing overfitting.

## 3 METHOD

We consider a simple technique of adding time-dependent Gaussian noise to the gradient $g$ at every training step $t$:

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

Our experiments indicate that adding annealed Gaussian noise by decaying the variance works better than using fixed Gaussian noise. We use a schedule inspired from Welling & Teh (2011) for most of our experiments and take:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \tag{1}$$

with $\eta$ selected from $\{0.01, 0.3, 1.0\}$ and $\gamma = 0.55$. Higher gradient noise at the beginning of training forces the gradient away from 0 in the early stages.

## 4 EXPERIMENTS

In the following experiments, we consider a variety of complex neural network architectures: Deep networks for MNIST digit classification, End-To-End Memory Networks (Sukhbaatar et al., 2015) and Neural Programmer (Neelakantan et al., 2015) for question answering, Neural Random Access Machines (Kurach et al., 2015) and Neural GPUs (Kaiser & Sutskever, 2015) for algorithm learning. The models and results are described as follows.

### 4.1 DEEP FULLY-CONNECTED NETWORKS

For our first set of experiments, we examine the impact of adding gradient noise when training a very deep fully-connected network on the MNIST handwritten digit classification dataset (LeCun et al., 1998). Our network is deep: it has 20 hidden layers, with each layer containing 50 hidden units. We use the ReLU activation function (Nair & Hinton, 2010).

In this experiment, we add gradient noise sampled from a Gaussian distribution with mean 0, and decaying variance according to the schedule in Equation (1) with $\eta = 0.01$. We train with SGD without momentum, using the fixed learning rates of 0.1 and 0.01. Unless otherwise specified, the weights of the network are initialized from a Gaussian with mean zero, and standard deviation of 0.1, which we call *Simple Init.*

The results of our experiment are in Table 1. When trained from Simple Init we can see that adding noise to the gradient helps in achieving higher average and best accuracy over 20 runs using each learning rate for a total of 40 runs (Table 1, Experiment 1). We note that the average is closer to 50% because the small learning rate of 0.01 usually gives very slow convergence. We also try our approach on a more shallow network of 5 layers, but adding noise does not improve the training in that case.

Next, we experiment with clipping the gradients with two threshold values: 100 and 10 (Table 1, Experiment 2, and 3). Here, we find training with gradient noise is insensitive to the gradient clipping values. By tuning the clipping threshold, it is possible to get comparable accuracy without noise for this problem.

In our fourth experiment (Table 1, Experiment 4), we use the analytically-derived ReLU initialization technique (which we term *Good Init*) recently-proposed by He et al. (2015) and find that adding gradient noise does not help. Previous work has found that stochastic gradient descent with carefully tuned initialization, momentum, learning rate, and learning rate decay can optimize such extremely deep fully-connected ReLU networks (Srivastava et al., 2015). It would be harder to find such a robust initialization technique for the more complex heterogeneous architectures considered in later sections. Accordingly, we find in later experiments (e.g., Section 4.3) that random restarts and the use of a momentum-based optimizer like Adam are not sufficient to achieve the best results in the absence of added gradient noise.

To test how sensitive the methods are to poor initialization, in addition to the sub-optimal Simple Init, we run an experiment where all the weights in the neural network are initialized at zero. The results (Table 1, Experiment 5) show that if we do not add noise to the gradient, the networks fail to learn. If we add some noise, the networks can learn and reach 94.5% accuracy.

Experiment 1: Simple Init, No Gradient Clipping

| Setting | Best Test Accuracy | Average Test Accuracy |
|---|---|---|
| No Noise | 89.9 | 43.1 |
| With Noise | 96.7 | 52.7 |
| No Noise + Dropout | 11.3 | 10.8 |

Experiment 2: Simple Init, Gradient Clipping = 100

| | | |
|---|---|---|
| No Noise | 90.0 | 46.3 |
| With Noise | 96.7 | 52.3 |

Experiment 3: Simple Init, Gradient Clipping = 10

| | | |
|---|---|---|
| No Noise | 95.7 | 51.6 |
| With Noise | 97.0 | 53.6 |

Experiment 4: Good Init (He et al., 2015) + Gradient Clipping = 10

| | | |
|---|---|---|
| No Noise | 97.4 | 91.7 |
| With Noise | 97.2 | 91.7 |

Experiment 5: Bad Init (Zero Init) + Gradient Clipping = 10

| | | |
|---|---|---|
| No Noise | 11.4 | 10.1 |
| With Noise | 94.5 | 49.7 |

Table 1: Average and best test accuracy on MNIST over 40 runs.

In summary, these experiments show that if we are careful with initialization and gradient clipping values, it is possible to train a very deep fully-connected network without adding gradient noise. However, if the initialization is poor, optimization can be difficult, and adding noise to the gradient is a good mechanism to overcome the optimization difficulty.

The implication of this set of results is that added gradient noise can be an effective mechanism for training very complex networks. This is because it is more difficult to initialize the weights properly for complex networks. In the following, we explore the training of more complex networks such as End-To-End Memory Networks and Neural Programmer, whose initialization is less well studied.

### 4.2 End-To-End Memory Networks

We test added gradient noise for training End-To-End Memory Networks (Sukhbaatar et al., 2015), a new approach for Q&A using deep networks.[1] Memory Networks have been demonstrated to perform well on a relatively challenging toy Q&A problem (Weston et al., 2015).

[1]Code available at: https://github.com/facebook/MemNN

In Memory Networks, the model has access to a context, a question, and is asked to predict an answer. Internally, the model has an attention mechanism which focuses on the right clue to answer the question. In the original formulation (Weston et al., 2015), Memory Networks were provided with additional supervision as to what pieces of context were necessary to answer the question. This was replaced in the End-To-End formulation by a latent attention mechanism implemented by a softmax over contexts. As this greatly complicates the learning problem, the authors implement a two-stage training procedure: First train the networks with a linear attention, then use those weights to warmstart the model with softmax attention.

In our experiments with Memory Networks, we use our standard noise schedule, using noise sampled from a Gaussian distribution with mean 0, and decaying variance according to Equation (1) with $\eta = 1.0$. This noise is added to the gradient after clipping. We also find for these experiments that a fixed standard deviation also works, but its value has to be tuned, and works best at 0.001. We set the number of training epochs to 200 because we would like to understand the behaviors of Memory Networks near convergence. The rest of the training is identical to the experimental setup proposed by the original authors. We test this approach with the published two-stage training approach, and additionally with a one-stage training approach where we train the networks with softmax attention and without warmstarting. Results are reported in Table 2. We find some fluctuations during each run of the training, but the reported results reflect the typical gains obtained by adding random noise.

We find that warmstarting does indeed help the networks. In both cases, adding random noise to the gradient also helps the network both in terms of training errors and validation errors. Added noise, however, is especially helpful for the training of End-To-End Memory Networks without the warmstarting stage.

| Setting | No Noise | With Noise |
|---|---|---|
| One-stage training | Training error: 10.5%<br>Validation error: 19.5% | Training error: 9.6%<br>Validation error: 16.6% |
| Two-stage training | Training error: 6.2%<br>Validation error: 10.9% | Training error: 5.9%<br>Validation error: 10.8% |

Table 2: The effects of adding random noise to the gradient on Neural Programmer. Adding random noise to the gradient always helps the model. When the models are applied to these more complicated tasks than the single column experiment, using dropout and noise together seems to be beneficial in one case while using only one of them achieves the best result in the other case.

### 4.3 NEURAL PROGRAMMER

Neural Programmer is a neural network architecture augmented with a small set of built-in arithmetic and logic operations that learns to induce latent programs. It is proposed for the task of question answering from tables (Neelakantan et al., 2015). Examples of operations on a table include the sum of a set of numbers, or the list of numbers greater than a particular value. Key to Neural Programmer is the use of "soft selection" to assign a probability distribution over the list of operations. This probability distribution weighs the result of each operation, and the cost function compares this weighted result to the ground truth. This soft selection, inspired by the soft attention mechanism of Bahdanau et al. (2014), allows for full differentiability of the model. Running the model for several steps of selection allows the model to induce a complex program by chaining the operations, one after the other. At convergence, the soft selection tends to become peaky (hard selection). Figure 1 shows the architecture of Neural Programmer at a high level.

In a synthetic table comprehension task, Neural Programmer takes a question and a table (or database) as input and the goal is to predict the correct answer. To solve this task, the model has to induce a program and execute it on the table. A major challenge is that the supervision signal is in the form of the correct answer and not the program itself. The model runs for a fixed number of steps, and at each step selects a data segment and an operation to apply to the selected data segment. Soft selection is performed at training time so that the model is differentiable, while at test time hard selection is employed. Table 3 shows examples of programs induced by the model.

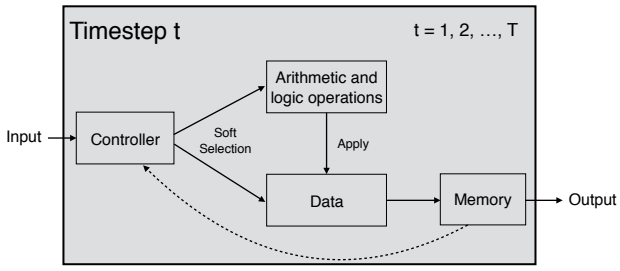



Figure 1: Neural Programmer, a neural network with built-in arithmetic and logic operations. At every time step, the controller selectes an operation and a data segment. Figure reproduced with permission from Neelakantan et al. (2015).

| Question | t | Selected Op | Selected Column |
|---|---|---|---|
| greater 50.32 C and lesser 20.21 E sum H | 1 | Greater | C |
| *What is the sum of numbers in column H* | 2 | Lesser | E |
| *whose field in column C is greater than 50.32* | 3 | And | - |
| *and field in Column E is lesser than 20.21.* | 4 | Sum | H |

Table 3: Example program induced by the model using $T = 4$ time steps. We show the selected columns in cases in which the selected operation acts on a particular column.

Similar to the above experiments with Memory Networks, in our experiments with Neural Programmer, we add noise sampled from a Gaussian distribution with mean 0, and decaying variance according to Equation (1) with $\eta = 1.0$ to the gradient after clipping. The model is optimized with Adam (Kingma & Ba, 2014), which combines momentum and adaptive learning rates.

For our first experiment, we train Neural Programmer to answer questions involving a single column of numbers. We use 72 different hyper-parameter configurations with and without adding annealed random noise to the gradients. We also run each of these experiments for 3 different random initializations of the model parameters and we find that only $1/216$ runs achieve $100\%$ test accuracy without adding noise while $9/216$ runs achieve $100\%$ accuracy when random noise is added. The 9 successful runs consisted of models initialized with all the three different random seeds, demonstrating robustness to initialization. We find that when using dropout (Srivastava et al., 2014) none of the 216 runs give 100% accuracy.

We consider a more difficult question answering task where tables have up to five columns containing numbers. We also experiment on a task containing one column of numbers and another column of text entries. Table 4 shows the performance of adding noise vs. no noise on Neural Programmer.

| Setting | No Noise | With Noise | Dropout | Dropout With Noise |
|---|---|---|---|---|
| Five columns | 95.3% | 98.7% | 97.4% | 99.2% |
| Text entries | 97.6% | 98.8% | 99.1% | 97.3% |

Table 4: The effects of adding random noise to the gradient on Neural Programmer. Adding random noise to the gradient always helps the model. When the models are applied to these more complicated tasks than the single column experiment, using dropout and noise together seems to be beneficial in one case while using only one of them achieves the best result in the other case.

Figure 2 shows an example of the effect of adding random noise to the gradients in our experiment with 5 columns. The differences between the two models are much more pronounced than Table 4 indicates because that table reflects the results from the best hyperparameters. Figure 2 indicates a more typical training run.

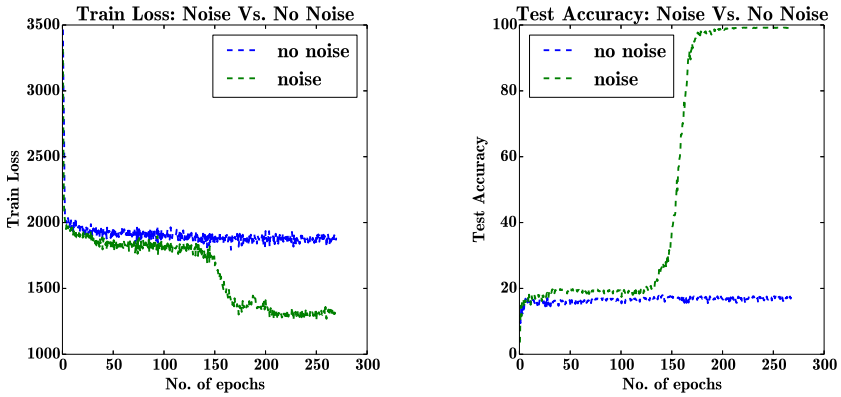



Figure 2: Noise Vs. No Noise in our experiment with 5 columns. The models trained with noise generalizes almost always better.

In all cases, we see that added gradient noise improves performance of Neural Programmer. Its performance when combined with or used instead of dropout is mixed depending on the problem, but the positive results indicate that it is worth attempting on a case-by-case basis.

### 4.4 Neural Random Access Machines

We now conduct experiments with Neural Random-Access Machines (NRAM) (Kurach et al., 2015). NRAM is a model for algorithm learning that can store data, and explicitly manipulate and dereference pointers. NRAM consists of a neural network controller, memory, registers and a set of built-in operations. This is similar to the Neural Programmer in that it uses a controller network to compose built-in operations, but both reads and writes to an external memory. An operation can either read (a subset of) contents from the memory, write content to the memory or perform an arithmetic operation on either input registers or outputs from other operations. The controller runs for a fixed number of time steps. At every step, the model selects a "circuit" to be executed: both the operations and its inputs. An example of such circuit is presented in Figure 4.

These selections are made using soft attention (Bahdanau et al., 2014) making the model end-to-end differentiable. NRAM uses an LSTM (Hochreiter & Schmidhuber, 1997) controller. Figure 3 gives an overview of the model.

For our experiment, we consider a problem of searching $k$-th element's value on a linked list. The network is given a pointer to the head of the linked list, and has to find the value of the $k$-th element. Note that this is highly nontrivial because pointers and their values are stored at random locations in memory, so the model must learn to traverse a complex graph for $k$ steps.

Because of this complexity, training the NRAM architecture can be unstable, especially when the number of steps and operations is large. We once again experiment with the decaying noise schedule from Equation (1), setting $\eta = 0.3$. We run a large grid search over the model hyperparameters (detailed in Kurach et al. (2015)), and use the top 3 for our experiments. For each of these 3 settings, we try 100 different random initializations and look at the percentage of runs that give $100\%$ accuracy across each one for training both with and without noise.

As in our experiments with Neural Programmer, we find that gradient clipping is crucial when training with noise. This is likely because the effect of random noise is washed away when gradients become too large. For models trained with noise we observed much better reproduce rates, which are presented in Table 5. Although it is possible to train the model to achieve $100\%$ accuracy without noise, it is less robust across multiple random restarts, with over 10x as many initializations leading to a correct answer when using noise.

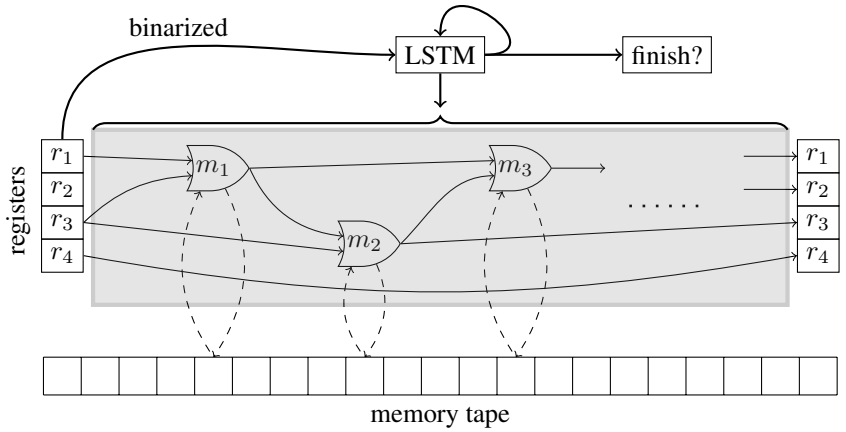



Figure 3: One timestep of the NRAM architecture with $R = 4$ registers and a memory tape. $m_1$, $m_2$ and $m_3$ are example operations built-in to the model. The operations can read and write from memory. At every time step, the LSTM controller softly selects the operation and its inputs. Figure reproduced with permission from Kurach et al. (2015).

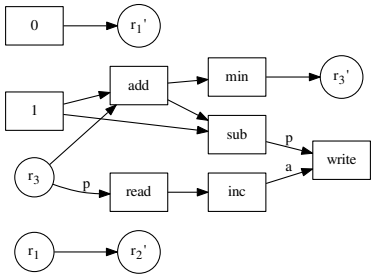



Figure 4: An example circuit generated by NRAM architecutre. The registers are represented by circles and modules by rectangles. For modules where the order of parameters matter, we label the edges with $p$ (the address to be read/written) and $a$ (the value to be written - only for **write** module). This circuit solves the problem of incrementing given array of elements. Notice that only register $r_3$ is used in the algorithm and the module "min" could be removed. The register $r_3$ is incremented in every time step. The value of $r_3$ is passed to **read** and **write** as the address ($p$). The value ($a$) for **write** is the output from **read** module incremented by 1.

| | Hyperparameter-1 | Hyperparameter-2 | Hyperparameter-3 | Average |
|---|---|---|---|---|
| No Noise | 1% | 0% | 3% | 1.3% |
| With Noise | 5% | 22% | 7% | 11.3% |

Table 5: Comparison of reproducibility on $k$-th element task. All tests were performed with the same set of 100 random initializations (seeds).

### 4.5 Convolutional Gated Recurrent Networks (Neural GPUs)

Convolutional Gated Recurrent Networks (CGRN) or Neural GPUs (Kaiser & Sutskever, 2015) are a recently proposed model that is capable of learning arbitrary algorithms. CGRNs use a stack of convolution layers, unfolded with tied parameters like a recurrent network. The input data (usually a list of symbols) is first converted to a three dimensional tensor representation containing a sequence of embedded symbols in the first two dimensions, and zeros padding the next dimension. Then,

multiple layers of modified convolution kernels are applied at each step. The modified kernel is a combination of convolution and Gated Recurrent Units (GRU) (Cho et al., 2014). The use of convolution kernels allows computation to be applied in parallel across the input data, while the gating mechanism helps the gradient flow. The additional dimension of the tensor serves as a working memory while the repeated operations are applied at each layer. The output at the final layer is the predicted answer.

The key difference between Neural GPUs and other architectures for algorithmic tasks (e.g., Neural Turing Machines (Graves et al., 2014)) is that instead of using sequential data access, convolution kernels are applied in parallel across the input, enabling the use of very deep and wide models. The model is referred to as Neural GPU because the input data is accessed in parallel. Neural GPUs were shown to outperform previous sequential architectures for algorithm learning on tasks such as binary addition and multiplication, by being able to generalize from much shorter to longer data cases.

In our experiments, we use Neural GPUs for the task of binary multiplication. The input consists two concatenated sequences of binary digits separated by an operator token, and the goal is to multiply the given numbers. During training, the model is trained on 20-digit binary numbers while at test time, the task is to multiply 200-digit numbers. Once again, we add noise sampled from Gaussian distribution with mean 0, and decaying variance according to the schedule in Equation (1) with $\eta = 1.0$, to the gradient after clipping. The model is optimized using Adam (Kingma & Ba, 2014).

Table 6 gives the results of a large-scale experiment using Neural GPUs with a 7290 grid search. The experiment shows that models trained with added gradient noise are more robust across many random initializations and parameter settings. As you can see, adding gradient noise both allows us to achieve the best performance, with the number of models with $< 1\%$ error over twice as large as without noise. But it also helps throughout, improving the robustness of training, with more models training to higher error rates as well. This experiment shows that the simple technique of added gradient noise is effective even in regimes where we can afford a very large numbers of random restarts.

| Setting | Error $< 1\%$ | Error $< 2\%$ | Error $< 3\%$ | Error $< 5\%$ |
|---|---|---|---|---|
| No Noise | 28 | 90 | 172 | 387 |
| With Noise | 58 | 159 | 282 | 570 |

Table 6: Comparison of reproducibility on 7290 random restarts. Trained on length 20 and tested on length 200.

## 5 Conclusion

In this paper, we discussed a set of experiments which show the effectiveness of adding noise to the gradient. We found that adding noise to the gradient during training helps training and generalization of complicated neural networks. We suspect that the effects are pronounced for complex models because they have many local minima.

We believe that this surprisingly simple yet effective idea, essentially a single line of code, should be in the toolset of neural network practitioners when facing issues with training neural networks. We also believe that this set of empirical results can give rise to further formal analysis of why adding noise is so effective for very deep neural networks.

**Acknowledgements** We sincerely thank Marcin Andrychowicz, Dmitry Bahdanau, Samy Bengio for suggestions and the Google Brain team for help with the project.

## References

Guozhong An. The effects of adding noise during backpropagation training on a generalization performance. *Neural Computation*, 1996.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2014.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *ICML*, 2015.

Léon Bottou. Stochastic gradient learning in neural networks. In *Neuro-Nïmes*, 1992.

Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *NIPS*, 2008.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 2011.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. AISTATS*, pp. 249–256, 2010.

Alex Graves. Practical variational inference for neural networks. In *NIPS*, 2011.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arxiv:1308.0850*, 2013.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *ICCV*, 2015.

Geoffrey Hinton and Sam Roweis. Stochastic neighbor embedding. In *NIPS*, 2002.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

Lukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *Arxiv*, 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Scott Kirkpatrick, Mario P Vecchi, et al. Optimization by simulated annealing. *Science*, 1983.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random access machine. In *Arxiv*, 2015.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Vinod Nair and Geoffrey Hinton. Rectified linear units improve Restricted Boltzmann Machines. In *ICML*, 2010.

Radford M Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2011.

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural Programmer: Inducing latent programs with gradient descent. In *Arxiv*, 2015.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *Proc. ICML*, 2013.

Baolin Peng, Zhengdong Lu, Hang Li, and Kam-Fai Wong. Towards neural network-based reasoning. *arXiv preprint arxiv:1508.05508*, 2015.

Boris Teodorovich Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 1964.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, 1951.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. *NIPS*, 2015.

Mark Steijvers. A recurrent network that performs a context-sensitive prediction task. In *CogSci*, 1996.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *NIPS*, 2015.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

Max Welling and Yee Whye Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *ICML*, 2011.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Jason Weston, Antoine Bordes, Sumit Chopra, and Tomas Mikolov. Towards AI-complete question answering: a set of prerequisite toy tasks. In *ICML*, 2015.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

# Learning to Discover Efficient Mathematical Identities

**Wojciech Zaremba**
Dept. of Computer Science
Courant Institute
New York Unviersity

**Karol Kurach**
Google &
Dept. of Computer Science
University of Warsaw

**Rob Fergus**
Dept. of Computer Science
Courant Institute
New York Unviersity

## Abstract

In this paper we explore how machine learning techniques can be applied to the discovery of efficient mathematical identities. We introduce an attribute grammar framework for representing symbolic expressions. Given a grammar of math operators, we build trees that combine them in different ways, looking for compositions that are analytically equivalent to a target expression but of lower computational complexity. However, as the space of trees grows exponentially with the complexity of the target expression, brute force search is impractical for all but the simplest of expressions. Consequently, we introduce two novel learning approaches that are able to learn from simpler expressions to guide the tree search. The first of these is a simple $n$-gram model, the other being a recursive neural-network. We show how these approaches enable us to derive complex identities, beyond reach of brute-force search, or human derivation.

## 1 Introduction

Machine learning approaches have proven highly effective for statistical pattern recognition problems, such as those encountered in speech or vision. However, their use in symbolic settings has been limited. In this paper, we explore how learning can be applied to the discovery of mathematical identities. Specifically, we propose methods for finding computationally efficient versions of a given target expression. That is, finding a new expression which computes an identical result to the target, but has a lower complexity (in time and/or space).

We introduce a framework based on attribute grammars [14] that allows symbolic expressions to be expressed as a sequence of grammar rules. Brute-force enumeration of all valid rule combinations allows us to discover efficient versions of the target, including those too intricate to be discovered by human manipulation. But for complex target expressions this strategy quickly becomes intractable, due to the exponential number of combinations that must be explored. In practice, a random search within the grammar tree is used to avoid memory problems, but the chance of finding a matching solution becomes vanishingly small for complex targets.

To overcome this limitation, we use machine learning to produce a search strategy for the grammar trees that selectively explores branches likely (under the model) to yield a solution. The training data for the model comes from solutions discovered for simpler target expressions. We investigate several different learning approaches. The first group are $n$-gram models, which learn pairs, triples etc. of expressions that were part of previously discovered solutions, thus hopefully might be part of the solution for the current target. We also train a recursive neural network (RNN) that operates within the grammar trees. This model is first pretrained to learn a continuous representation for symbolic expressions. Then, using this representation we learn to predict the next grammar rule to add to the current expression to yield an efficient version of the target.

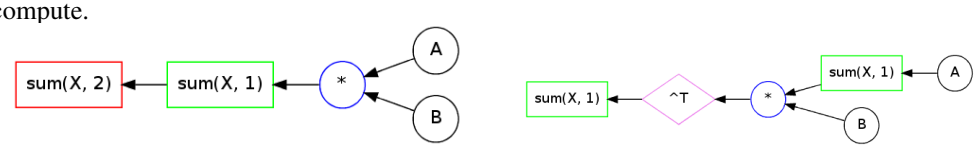
Through the use of learning, we are able to dramatically widen the complexity and scope of expressions that can be handled in our framework. We show examples of (i) O $\left(n^3\right)$ target expressions which can be computed in O $\left(n^2\right)$ time (e.g. see Examples 1 & 2), and (ii) cases where naive eval-

uation of the target would require *exponential* time, but can be computed in $\mathrm{O}\left(n^2\right)$ or $\mathrm{O}\left(n^3\right)$ time. The majority of these examples are too complex to be found manually or by exhaustive search and, as far as we are aware, are previously undiscovered. All code and evaluation data can be found at `https://github.com/kkurach/math_learning`.

In summary our contributions are:

- A novel grammar framework for finding efficient versions of symbolic expressions.
- Showing how machine learning techniques can be integrated into this framework, and demonstrating how training models on simpler expressions can help which the discovery of more complex ones.
- A novel application of a recursive neural-network to learn a continuous representation of mathematical structures, making the symbolic domain accessible to many other learning approaches.
- The discovery of many new mathematical identities which offer a significant reduction in computational complexity for certain expressions.

**Example 1:** Assume we are given matrices $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times p}$. We wish to compute the target expression: `sum(sum(A*B))`, i.e. : $\sum_{n,p} AB = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{p} A_{i,j} B_{j,k}$ which naively takes $O(nmp)$ time. Our framework is able to discover an efficient version of the formula, that computes the same result in $O(n(m+p))$ time: `sum((sum(A, 1) * B)', 1)`. Our framework builds *grammar trees* that explore valid compositions of expressions from the grammar, using a *search strategy*. In this example, the naive strategy of randomly choosing permissible rules suffices and we can find another tree which matches the target expression in reasonable time. Below, we show trees for (i) the original expression and (ii) the efficient formula which avoids the use of a matrix-matrix multiply operation, hence is efficient to compute.




**Example 2:** Consider the target expression: `sum(sum((A*B)^k))`, where $k = 6$. For an expression of this degree, there are 9785 possible grammar trees and the naive strategy used in Example 1 breaks down. We therefore *learn* a search strategy, training a model on successful trees from simpler expressions, such as those for $k = 2, 3, 4, 5$. Our learning approaches capture the common structure within the solutions, evident below, so can find an efficient $\mathrm{O}\left(nm\right)$ expression for this target:

$k = 2$: `sum((((((sum(A, 1)) * B) * A) * B)'), 1)`

$k = 3$: `sum(((((((sum(A, 1)) * B) * A) * B) * A) * B)'), 1)`

$k = 4$: `sum((((((((((sum(A, 1)) * B) * A) * B) * A) * B) * A) * B)'), 1)`

$k = 5$: `sum((((((((((((sum(A, 1)) * B) * A) * B) * A) * B) * A) * B) * A) * B)'), 1)`

$k = 6$: `sum((((((((((((sum(A, 1) * B) * A) * B) *A) * B) * A) * B)* A) * B) * A) * B)'), 1)`

### 1.1 Related work

The problem addressed in this paper overlaps with the areas of theorem proving [5, 9, 11], program induction [18, 28] and probabilistic programming [12, 20]. These domains involve the challenging issues of undecidability, the halting problem, and a massive space of potential computation. However, we limit our domain to computation of polynomials with fixed degree $k$, where undecidability and the halting problem are not present, and the space of computation is manageable (i.e. it grows exponentially, but not super-exponentially). Symbolic computation engines, such as Maple [6] and Mathematica [27] are capable of simplifying expressions by collecting terms but do not explicitly seek versions of lower complexity. Furthermore, these systems are rule based and do not use learning approaches, the major focus of this paper. In general, there has been very little exploration of statistical machine learning techniques in these fields, one of the few attempts being the recent work of Bridge *et al.* [4] who use learning to select between different heuristics for 1st order reasoning. In contrast, our approach does not use hand-designed heuristics, instead learning them automatically from the results of simpler expressions.

| Rule | Input | Output | Computation | Complexity |
|---|---|---|---|---|
| Matrix-matrix multiply | $X \in \mathbb{R}^{n \times m}$ , $Y \in \mathbb{R}^{m \times p}$ | $Z \in \mathbb{R}^{n \times p}$ | `Z = X * Y` | $\mathrm{O}(nmp)$ |
| Matrix-element multiply | $X \in \mathbb{R}^{n \times m}$ , $Y \in \mathbb{R}^{n \times m}$ | $Z \in \mathbb{R}^{n \times m}$ | `Z = X .* Y` | $\mathrm{O}(nm)$ |
| Matrix-vector multiply | $X \in \mathbb{R}^{n \times m}$ , $Y \in \mathbb{R}^{m \times 1}$ | $Z \in \mathbb{R}^{n \times n}$ | `Z = X * Y` | $\mathrm{O}(nm)$ |
| Matrix transpose | $X \in \mathbb{R}^{n \times m}$ | $Z \in \mathbb{R}^{m \times n}$ | `Z = X'` | $\mathrm{O}(nm)$ |
| Column sum | $X \in \mathbb{R}^{n \times m}$ | $Z \in \mathbb{R}^{n \times 1}$ | `Z = sum(X,1)` | $\mathrm{O}(nm)$ |
| Row sum | $X \in \mathbb{R}^{n \times m}$ | $Z \in \mathbb{R}^{1 \times m}$ | `Z = sum(X,2)` | $\mathrm{O}(nm)$ |
| Column repeat | $X \in \mathbb{R}^{n \times 1}$ | $Z \in \mathbb{R}^{n \times m}$ | `Z = repmat(X,1,m)` | $\mathrm{O}(nm)$ |
| Row repeat | $X \in \mathbb{R}^{1 \times m}$ | $Z \in \mathbb{R}^{n \times m}$ | `Z = repmat(X,n,1)` | $\mathrm{O}(nm)$ |
| Element repeat | $X \in \mathbb{R}^{1 \times 1}$ | $Z \in \mathbb{R}^{n \times m}$ | `Z = repmat(X,n,m)` | $\mathrm{O}(nm)$ |

Table 1: The grammar $\mathcal{G}$ used in our experiments.

The attribute grammar, originally developed in 1968 by Knuth [14] in context of compiler construction, has been successfully used as a tool for design and formal specification. In our work, we apply attribute grammars to a search and optimization problem. This has previously been explored in a range of domains: from well-known algorithmic problems like knapsack packing [19], through bioinformatics [26] to music [10]. However, we are not aware of any previous work related to discovering mathematical formulas using grammars, and learning in such framework. The closest work to ours can be found in [7] which involves searching over the space of algorithms and the grammar attributes also represent computational complexity.

Classical techniques in natural language processing make extensive use of grammars, for example to parse sentences and translate between languages. In this paper, we borrow techniques from NLP and apply them to symbolic computation. In particular, we make use of an $n$-gram model over mathematical operations, inspired by $n$-gram language models. Recursive neural networks have also been recently used in NLP, for example by Luong *et al.* [15] and Socher *et al.* [22, 23], as well as generic knowledge representation Bottou [2]. In particular, Socher *et al.* [23], apply them to parse trees for sentiment analysis. By contrast, we apply them to trees of symbolic expressions. Our work also has similarities to Bowman [3] who shows that a recursive network can learn simple logical predicates.

Our demonstration of continuous embeddings for symbolic expressions has parallels with the embeddings used in NLP for words and sentence structure, for example, Collobert & Weston [8], Mnih & Hinton [17], Turian *et al.* [25] and Mikolov *et al.* [16].

## 2 Problem Statement

**Problem Definition:** We are given a symbolic *target expression* $\mathbb{T}$ that combines a set of variables $\mathcal{V}$ to produce an output $\mathbb{O}$, i.e. $\mathbb{O} = \mathbb{T}(\mathcal{V})$. We seek an alternate expression $\mathbb{S}$, such that $\mathbb{S}(\mathcal{V}) = \mathbb{T}(\mathcal{V})$, but has lower computational complexity, i.e. $\mathrm{O}(\mathbb{S}) < \mathrm{O}(\mathbb{T})$.

In this paper we consider the restricted setting where: (i) $\mathbb{T}$ is a homogeneous polynomial of degree $k$*, (ii) $\mathcal{V}$ contains a single matrix or vector $A$ and (iii) $\mathbb{O}$ is a scalar. While these assumptions may seem quite restrictive, they still permit a rich family of expressions for our algorithm to explore. For example, by combining multiple polynomial terms, an efficient Taylor series approximation can be found for expressions involving trigonometric or exponential operators. Regarding (ii), our framework can easily handle multiple variables, e.g. Figure 1, which shows expressions using two matrices, $A$ and $B$. However, the rest of the paper considers targets based on a single variable. In Section 8, we discuss these restrictions further.

**Notation:** We adopt Matlab-style syntax for expressions.

## 3 Attribute Grammar

We first define an *attribute grammar* $\mathcal{G}$, which contains a set of mathematical operations, each with an associated complexity (the attribute). Since $\mathbb{T}$ contains exclusively polynomials, we use the grammar rules listed in Table 1.

Using these rules we can develop trees that combine rules to form expressions involving $\mathcal{V}$, which for the purposes of this paper is a single matrix $A$. Since we know $\mathbb{T}$ involves expressions of degree

*I.e. It only contains terms of degree $k$. E.g. $ab + a^2 + ac$ is a homogeneous polynomial of degree 2, but $a^2 + b$ is not homogeneous ($b$ is of degree 1, but $a^2$ is of degree 2).

$k$, each tree must use $A$ exactly $k$ times. Furthermore, since the output is a scalar, each tree must also compute a scalar quantity. These two constraints limit the depth of each tree. For some targets $\mathbb{T}$ whose complexity is only $\mathrm{O}\left(()\, n^3\right)$, we remove the matrix-matrix multiply rule, thus ensuring that if any solution is found its complexity is at most $\mathrm{O}\left(()\, n^2\right)$ (see Section 7.2 for more details). Examples of trees are shown in Fig. 1. The search strategy for determining which rules to combine is addressed in Section 6.

# 4 Representation of Symbolic Expressions

We need an efficient way to check if the expression produced by a given tree, or combination of trees (see Section 5), matches $\mathbb{T}$. The conventional approach would be to perform this check symbolically, but this is too slow for our purposes and is not amenable to integration with learning methods. We therefore explore two alternate approaches.

## 4.1 Numerical Representation

In this representation, each expression is represented by its evaluation of a randomly drawn set of $N$ points, where $N$ is large (typically 1000). More precisely, for each variable in $\mathcal{V}$, $N$ different copies are made, each populated with randomly drawn elements. The target expression evaluates each of these copies, producing a scalar value for each, so yielding a vector $t$ of length $N$ which uniquely characterizes $\mathbb{T}$. Formally, $t_n = \mathbb{T}(\mathcal{V}_n)$. We call this numerical vector $t$ the *descriptor* of the symbolic expression $\mathbb{T}$. The size of the descriptor $N$, must be sufficiently large to ensure that different expressions are not mapped to the same descriptor. Furthermore, when the descriptors are used in the linear system of Eqn. 5 below, $N$ must also be greater than the number of linear equations. Any expression $\mathbb{S}$ formed by the grammar can be used to evaluate each $\mathcal{V}_n$ to produce another $N$-length descriptor vector $s$, which can then be compared to $t$. If the two match, then $\mathbb{S}(\mathcal{V}) = \mathbb{T}(\mathcal{V})$.

In practice, using floating point values can result in numerical issues that prevent $t$ and $s$ matching, even if the two expressions are equivalent. We therefore use an integer-based descriptor in the form of $\mathbb{Z}_p$[†], where $p$ is a large prime number. This prevents both rounding issues as well as numerical overflow.

## 4.2 Learned Representation

We now consider how to learn a continuous representation for symbolic expressions, that is learn a projection $\phi$ which maps expressions $\mathbb{S}$ to $l$-dimensional vectors: $\phi(\mathbb{S}) \rightarrow \mathbb{R}^l$. We use a recursive neural network (RNN) to do this, in a similar fashion to Socher *et al.* [23] for natural language and Bowman *et al.* [3] for logical expressions. This potentially allows many symbolic tasks to be performed by machine learning techniques, in the same way that the word-vectors (e.g.[8] and [16]) enable many NLP tasks to be posed a learning problems.

We first create a dataset of symbolic expressions, spanning the space of all valid expressions up to degree $k$. We then group them into clusters of equivalent expressions (using the numerical representation to check for equality), and give each cluster a discrete label $1 \ldots C$. For example, $A$, $(A^T)^T$ might have label 1, and $\sum_i \sum_j A_{i,j}$, $\sum_j \sum_i A_{i,j}$ might have label 2 and so on. For $k = 6$, the dataset consists of $C = 1687$ classes, examples of which are show in Fig. 1. Each class is split 80/20 into train/test sets.

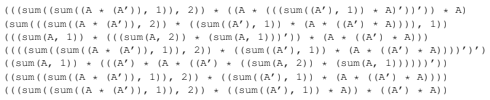
We then train a recursive neural network (RNN) to classify a grammar tree into one of the $C$ clusters. Instead of representing each grammar rule by its underlying arithmetic, we parameterize it by a weight matrix or tensor (for operations with one or two inputs, respectively) and use this to learn the *concept* of each operation, as part of the network. A vector $a \in \mathbb{R}^l$, where $l = 30$[‡] is used to represent each input variable. Working along the grammar tree, each operation in $\mathbb{S}$ evolves this vector via matrix/tensor multiplications (preserving its length) until the entire expression is parsed, resulting in a single vector $\phi(\mathbb{S})$ of length $l$, which is passed to the classifier to determine the class of the expression, and hence which other expressions it is equivalent to.

Fig. 2 shows this procedure for two different expressions. Consider the first expression $\mathbb{S} = (A. * A)' * \mathrm{sum}(A, 2)$. The first operation here is $.*$, which is implemented in the RNN by taking the

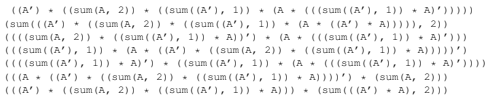
[†]Integers modulo $p$

[‡]This was selected by cross-validation to control the capacity of the RNN, since it directly controls the number of parameters in the model.

two (identical) vectors $a$ and applies a weight tensor $W_3$ (of size $l \times l \times l$, so that the output is also size $l$), followed by a rectified-linear non-linearity. The output of this stage is this $\max((W_3 * a) * a, 0)$. This vector is presented to the next operation, a matrix transpose, whose output is thus $\max(W_2 * \max((W_3 * a) * a, 0), 0)$. Applying the remaining operations produces a final output: $\phi(\mathbb{S}) = \max((W_4 * \max(W_2 * \max((W_3 * a) * a, 0), 0)) * max(W_1 * a, 0))$. This is presented to a $C$-way softmax classifier to predict the class of the expression. The weights $W$ are trained using a cross-entropy loss and backpropagation.

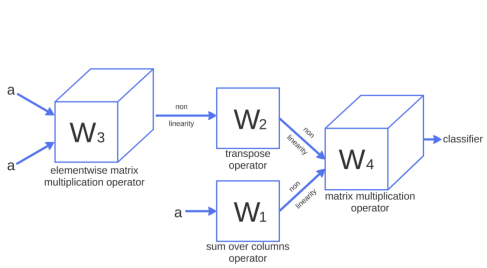



(a) Class A

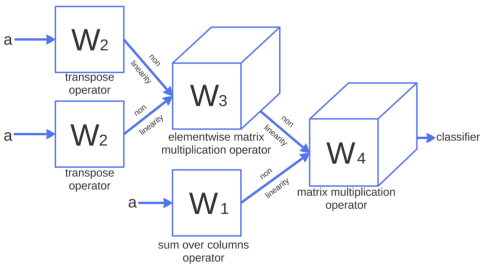



(b) Class B

Figure 1: Samples from two classes of degree $k = 6$ in our dataset of expressions, used to learn a continuous representation of symbolic expressions via an RNN. Each line represents a different expression, but those in the same class are equivalent to one another.




(a) $(A. * A)' * \text{sum}(A, 2)$




(b) $(A'. * A') * \text{sum}(A, 2)$

Figure 2: Our RNN applied to two expressions. The matrix $A$ is represented by a fixed random vector $a$ (of length $l = 30$). Each operation in the expression applies a different matrix (for single input operations) or tensor (for dual inputs, e.g. matrix-element multiplication) to this vector. After each operation, a rectified-linear non-linearity is applied. The weight matrices/tensors for each operation are shared across different expressions. The final vector is passed to a softmax classifier (not shown) to predict which class they belong to. In this example, both expressions are equivalent, thus should be mapped to the same class.

When training the RNN, there are several important details that are crucial to obtaining high classification accuracy:

- The weights should be initialized to the identity, plus a small amount of Gaussian noise added to all elements. The identity allows information to flow the full length of the network, up to the classifier regardless of its depth [21]. Without this, the RNN overfits badly, producing test accuracies of $\sim 1\%$.
- Rectified linear units work much better in this setting than tanh activation functions.
- We learn using a curriculum [1, 30], starting with the simplest expressions of low degree and slowly increasing $k$.
- The weight matrix in the softmax classifier has much larger ($\times 100$) learning rate than the rest of the layers. This encourages the representation to stay still even when targets are replaced, for example, as we move to harder examples.
- As well as updating the weights of the RNN, we also update the initial value of $a$ (i.e we backpropagate to the input also).

When the RNN-based representation is employed for identity discovery (see Section 6.3), the vector $\phi(\mathbb{S})$ is used directly (i.e. the $C$-way softmax used in training is removed from the network).

## 5 Linear Combinations of Trees

For simple targets, an expression that matches the target may be contained within a single grammar tree. But more complex expressions typically require a linear combination of expressions from different trees.

To handle this, we can use the integer-based descriptors for each tree in a linear system and solve for a match to the target descriptor (if one exists). Given a set of $M$ trees, each with its own integer descriptor vector $f$, we form an $M$ by $N$ linear system of equations and solve it:

$$Fw = t \bmod \mathbb{Z}_p$$

where $F = [f_1, \ldots, f_M]$ holds the tree representations, $w$ is the weighting on each of the trees and $t$ is the target representation. The system is solved using Gaussian elimination, where addition and multiplication is performed modulo $p$. The number of solutions can vary: (a) there can be **no solution**, which means that no linear combination of the current set of trees can match the target expression. If all possible trees have been enumerated, then this implies the target expression is outside the scope of the grammar. (b) There can be **one or more solutions**, meaning that some combination of the current set of trees yields a match to the target expression.

# 6 Search Strategy

So far, we have proposed a grammar which defines the computations that are permitted (like a programming language grammar), but it gives no guidance as to how explore the space of possible expressions. Neither do the representations we introduced help – they simply allow us to determine if an expression matches or not. We now describe how to efficiently explore the space by learning which paths are likely to yield a match.

Our framework uses two components: a **scheduler**, and a **strategy**. The scheduler is fixed, and traverses space of expressions according to recommendations given by the selected strategy (e.g. "Random" or "$n$-gram" or "RNN"). The strategy assesses which of the possible grammar rules is likely to lead to a solution, given the current expression. Starting with the variables $\mathcal{V}$ (in our case a single element $A$, or more generally, the elements $A$, $B$ etc.), at each step the scheduler receives scores for each rule from the strategy and picks the one with the highest score. This continues until the expression reaches degree $k$ and the tree is complete. We then run the linear solver to see if a linear combination of the existing set of trees matches the target. If not, the scheduler starts again with a new tree, initialized with the set of variables $\mathcal{V}$. The $n$-gram and RNN strategies are learned in an incremental fashion, starting with simple target expressions (i.e. those of low degree $k$, such as $\sum_{ij} AA^T$). Once solutions to these are found, they become training examples used to improve the strategy, needed for tackling harder targets (e.g. $\sum_{ij} AA^T A$).

## 6.1 Random Strategy

The random strategy involves no learning, thus assigns equal scores to all valid grammar rules, hence the scheduler randomly picks which expression to try at each step. For simple targets, this strategy may succeed as the scheduler may stumble upon a match to the target within a reasonable time-frame. But for complex target expressions of high degree $k$, the search space is huge and the approach fails.

## 6.2 $n$-gram

In this strategy, we simply count how often subtrees of depth $n$ occur in solutions to previously solved targets. As the number of different subtrees of depth $n$ is large, the counts become very sparse as $n$ grows. Due to this, we use a weighted linear combination of the score from all depths up to $n$. We found an effective weighting to be $10^k$, where $k$ is the depth of the tree.

## 6.3 Recursive Neural Network

Section 4.2 showed how to use an RNN to learn a continuous representation of grammar trees. Recall that the RNN $\phi$ maps expressions to continuous vectors: $\phi(\mathbb{S}) \rightarrow \mathbb{R}^l$. To build a search strategy from this, we train a softmax layer on top of the RNN to predict which rule should be applied to the current expression (or expressions, since some rules have two inputs), so that we match the target.

Formally, we have two current branches $b_1$ and $b_2$ (each corresponding to an expression) and wish to predict the root operation $r$ that joins them (e.g. $.*$) from among the valid grammar rules ($|r|$ in total). We first use the previously trained RNN to compute $\phi(b_1)$ and $\phi(b_2)$. These are then presented to a $|r|$-way softmax layer (whose weight matrix $U$ is of size $2l \times |r|$). If only one branch exists, then $b_2$ is set to a fixed random vector. The training data for $U$ comes from trees that give efficient solutions to targets of lower degree $k$ (i.e. simpler targets). Training of the softmax layer is performed by stochastic gradient descent. We use dropout [13] as the network has a tendency to overfit and repeat exactly the same expressions for the next value of $k$. Thus, instead of training on exactly $\phi(b_1)$ and $\phi(b_2)$, we drop activations as we propagate toward the top of the tree (the same

fraction for each depth), which encourages the RNN to capture more local structures. At test time, the probabilities from the softmax become the scores used by the scheduler.

# 7 Experiments

We first show results relating to the learned representation for symbolic expressions (Section 4.2). Then we demonstrate our framework discovering efficient identities. For brevity, the identities discovered are listed in the supplementary material [29].

## 7.1 Expression Classification using Learned Representation

Table 2 shows the accuracy of the RNN model on expressions of varying degree, ranging from $k = 3$ to $k = 6$. The difficulty of the task can be appreciated by looking at the examples in Fig. 1. The low error rate of $\leq 5\%$, despite the use of a simple softmax classifier, demonstrates the effectiveness of our learned representation.

| | Degree $k = 3$ | Degree $k = 4$ | Degree $k = 5$ | Degree $k = 6$ |
|---|---|---|---|---|
| Test accuracy | $100\% \pm 0\%$ | $96.9\% \pm 1.5\%$ | $94.7\% \pm 1.0\%$ | $95.3\% \pm 0.7\%$ |
| Number of classes | 12 | 125 | 970 | 1687 |
| Number of expressions | 126 | 1520 | 13038 | 24210 |

Table 2: Accuracy of predictions using our learned symbolic representation (averaged over 10 different initializations). As the degree increases tasks becomes more challenging, because number of classes grows, and computation trees become deeper. However our dataset grows larger too (training uses 80% of examples).

## 7.2 Efficient Identity Discovery

In our experiments we consider 5 different families of expressions, chosen to fall within the scope of our grammar rules:

1. $(\sum \mathbf{AA^T})_\mathbf{k}$: $A$ is an $\mathbb{R}^{n\times n}$ matrix. The $k$-th term is $\sum_{i,j}(AA^T)^{\lfloor k/2 \rfloor}$ for even $k$ and $\sum_{i,j}(AA^T)^{\lfloor k/2 \rfloor}A$ , for odd $k$. E.g. for $k = 2 : \sum_{i,j} AA^T$; for $k = 3 : \sum_{i,j} AA^TA$; for $k = 4 : \sum_{i,j} AA^TAA^T$ etc. Naive evaluation is $\mathrm{O}\left(kn^3\right)$.
2. $(\sum(\mathbf{A}. * \mathbf{A})\mathbf{A^T})_k$: $A$ is an $\mathbb{R}^{n\times n}$ matrix and let $B = A. * A$. The $k$-th term is $\sum_{i,j}(BA^T)^{\lfloor k/2 \rfloor}$ for even $k$ and $\sum_{i,j}(BA^TB)^{\lfloor k/2 \rfloor}$ , for odd $k$. E.g. for $k = 2 : \sum_{i,j}(A. * A)A^T$; for $k = 3 : \sum_{i,j}(A. * A)A^T(A. * A)$; for $k = 4 : \sum_{i,j}(A. * A)A^T(A. * A)A^T$ etc. Naive evaluation is $\mathrm{O}\left(kn^3\right)$.
3. $\mathbf{Sym}_k$: Elementary symmetric polynomials. $A$ is a vector in $\mathbb{R}^{n\times 1}$. For $k = 1 : \sum_i A_i$, for $k = 2 : \sum_{i<j} A_iA_j$, for $k = 3 : \sum_{i<j<k} A_iA_jA_k$, etc. Naive evaluation is $\mathrm{O}\left(n^k\right)$.
4. **(RBM-1)**$_k$: $A$ is a vector in $\mathbb{R}^{n\times 1}$. $v$ is a binary $n$-vector. The $k$-th term is: $\sum_{v\in\{0,1\}^n}(v^TA)^k$. Naive evaluation is $\mathrm{O}\left(2^n\right)$.
5. **(RBM-2)**$_k$: Taylor series terms for the partition function of an RBM. $A$ is a matrix in $\mathbb{R}^{n\times n}$. $v$ and $h$ are a binary $n$-vectors. The $k$-th term is $\sum_{v\in\{0,1\}^n, h\in\{0,1\}^n}(v^TAh)^k$. Naive evaluation is $\mathrm{O}\left(2^{2n}\right)$.

Note that (i) for all families, the expressions yield a scalar output; (ii) the families are ordered in rough order of "difficulty"; (iii) we are not aware of any previous exploration of these expressions, except for $\mathbf{Sym}_k$, which is well studied [24]. For the $(\sum \mathbf{AA^T})_\mathbf{k}$ and $(\sum(\mathbf{A}. * \mathbf{A})\mathbf{A^T})_k$ families we remove the matrix-multiply rule from the grammar, thus ensuring that if any solution is found it will be efficient since the remaining rules are at most $\mathrm{O}\left(kn^2\right)$, rather than $\mathrm{O}\left(kn^3\right)$. The other families use the full grammar, given in Table 1. However, the limited set of rules means that if any solution is found, it can at most be $\mathrm{O}\left(n^3\right)$, rather than exponential in $n$, as the naive evaluations would be. For each family, we apply our framework, using the three different search strategies introduced in Section 6. For each run we impose a fixed cut-off time of 10 minutes[§] beyond which we terminate the search. At each value of $k$, we repeat the experiments 10 times with different random initializations and count the number of runs that find an efficient solution. Any non-zero count is deemed a success, since each identity only needs to be discovered once. However, in Fig. 3, we show the fraction of successful runs, which gives a sense of how quickly the identity was found.

[§]Running on a 3Ghz 16-core Intel Xeon. Changing the cut-off has little effect on the plots, since the search space grows exponentially fast.

We start with $k = 2$ and increase up to $k = 15$, using the solutions from previous values of $k$ as training data for the current degree. The search space quickly grows with $k$, as shown in Table 3. Fig. 3 shows results for four of the families. We use $n$-grams for $n = 1 \ldots 5$, as well as the RNN with two different dropout rates (0.125 and 0.3). The learning approaches generally do much better than the random strategy for large values of $k$, with the 3-gram, 4-gram and 5-gram models outperforming the RNN.

For the first two families, the 3-gram model reliably finds solutions. These solutions involve repetition of a local patterns (e.g. Example 2), which can easily be captured with $n$-gram models. However, patterns that don't have a simple repetitive structure are much more difficult to generalize. The **(RBM-2)**$_k$ family is the most challenging, involving a double exponential sum, and the solutions have highly complex trees (see supplementary material [29]). In this case, none of our approaches performed better than the random strategy and no solutions were discovered for $k > 5$. However, the $k = 5$ solution was found by the RNN consistently faster than the random strategy ($100 \pm 12$ vs $438 \pm 77$ secs).

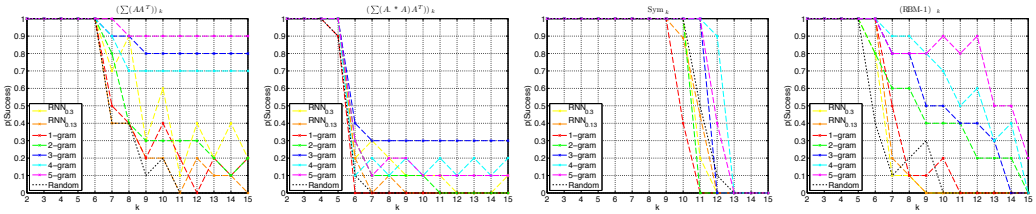



Figure 3: Evaluation on four different families of expressions. As the degree $k$ increases, we see that the random strategy consistently fails but the learning approaches can still find solutions (i.e. p(Success) is non-zero). Best viewed in electronic form.

| | $k=2$ | $k=3$ | $k=4$ | $k=5$ | $k=6$ | $k=7$ and higher |
|---|---|---|---|---|---|---|
| # Terms $\leq$ O $\left(n^2\right)$ | 39 | 171 | 687 | 2628 | 9785 | Out of memory |
| # Terms $\leq$ O $\left(n^3\right)$ | 41 | 187 | 790 | 3197 | 10k+ | |

Table 3: The number of possible expressions for different degrees $k$.

## 8 Discussion

We have introduced a framework based on a grammar of symbolic operations for discovering mathematical identities. Through the novel application of learning methods, we have shown how the exploration of the search space can be learned from previously successful solutions to simpler expressions. This allows us to discover complex expressions that random or brute-force strategies cannot find (the identities are given in the supplementary material [29]).

Some of the families considered in this paper are close to expressions often encountered in machine learning. For example, dropout involves an exponential sum over binary masks, which is related to the **RBM-1** family. Also, the partition function of an RBM can be approximated by the **RBM-2** family. Hence the identities we have discovered could potentially be used to give a closed-form version of dropout, or compute the RBM partition function efficiently (i.e. in polynomial time). Additionally, the automatic nature of our system naturally lends itself to integration with compilers, or other optimization tools, where it could replace computations with efficient versions thereof.

Our framework could potentially be applied to more general settings, to discover novel formulae in broader areas of mathematics. To realize this, additional grammar rules, e.g. involving recursion or trigonometric functions would be needed. However, this would require a more complex scheduler to determine when to terminate a given grammar tree. Also, it is surprising that a recursive neural network can generate an effective continuous representation for symbolic expressions. This could have broad applicability in allowing machine learning tools to be applied to symbolic computation.

The problem addressed in this paper involves discrete search within a combinatorially large space – a core problem with AI. Our successful use of machine learning to guide the search gives hope that similar techniques might be effective in other AI tasks where combinatorial explosions are encountered.

## Acknowledgements

The authors would like to thank Facebook and Microsoft Research for their support.

## References


[1] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *ICML*, 2009.

[2] L. Bottou. From machine learning to machine reasoning. *Machine Learning*, 94(2):133–149, 2014.

[3] S. R. Bowman. Can recursive neural tensor networks learn logical reasoning? *arXiv preprint arXiv:1312.6192*, 2013.

[4] J. P. Bridge, S. B. Holden, and L. C. Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, 53:141–172, August 2014.

[5] C.-L. Chang. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.

[6] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V library reference manual*, volume 199. Springer-verlag New York, 1991.

[7] G. Cheung and S. McCanne. An attribute grammar based framework for machine-dependent computational optimization of media processing algorithms. In *ICIP*, volume 2, pages 797–801. IEEE, 1999.

[8] R. Collobert and J. Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *ICML*, 2008.

[9] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[10] M. Desainte-Catherine and K. Barbar. Using attribute grammars to find solutions for musical equational programs. *ACM SIGPLAN Notices*, 29(9):56–63, 1994.

[11] M. Fitting. *First-order logic and automated theorem proving*. Springer, 1996.

[12] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and D. Tarlow. Church: a language for generative models. *arXiv:1206.3255*, 2012.

[13] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[14] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[15] M.-T. Luong, R. Socher, and C. D. Manning. Better word representations with recursive neural networks for morphology. In *CoNLL*, 2013.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.

[17] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. In *NIPS*, 2009.

[18] P. Nordin. *Evolutionary program induction of binary machine code and its applications*. Krehl Munster, 1997.

[19] M. ONeill, R. Cleary, and N. Nikolov. Solving knapsack problems with attribute grammars. In *Proceedings of the Third Grammatical Evolution Workshop (GEWS04)*. Citeseer, 2004.

[20] A. Pfeffer. Practical probabilistic programming. In *Inductive Logic Programming*, pages 2–3. Springer, 2011.

[21] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013.

[22] R. Socher, C. D. Manning, and A. Y. Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pages 1–9, 2010.

[23] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. P. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[24] R. P. Stanley. *Enumerative combinatorics*. Number 49. Cambridge university press, 2011.

[25] J. Turian, L. Ratinov, and Y. Bengio. Word representations: a simple and general method for semi-supervised learning. In *ACL*, 2010.

[26] J. Waldispühl, B. Behzadi, and J.-M. Steyaert. An approximate matching algorithm for finding (sub-) optimal sequences in s-attributed grammars. *Bioinformatics*, 18(suppl 2):S250–S259, 2002.

[27] S. Wolfram. *The mathematica book*, volume 221. Wolfram Media Champaign, Illinois, 1996.

[28] M. L. Wong and K. S. Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, 1997.

[29] W. Zaremba, K. Kurach, and R. Fergus. Learning to discover efficient mathematical identities. *arXiv:1406.1584*, 2014.

[30] W. Zaremba and I. Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

# 9 Supplementary material

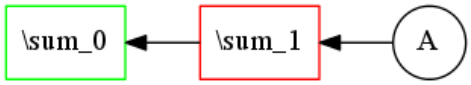
We present the efficient expressions discovered by our system, using Matlab-style syntax, and we visualize computation trees. Each example contains: (i) code that computes the original target formulas; (ii) the formulae derived by our system and (iii) code that verifies the correctness of the expression. The size of matrices $n$, $m$ can be chosen arbitrary.

Code for generating the expressions can be downloaded from `https://github.com/kkurach/math_learning`. The source files for this paper are available at `https://github.com/kkurach/math_learning/paper/`.

## 9.1 $(\sum \mathbf{A}\mathbf{A}^{\mathbf{T}})_{\mathbf{k}}$

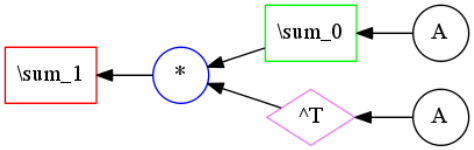
### k = 1

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(A, 1), 2);

optimized = 1 * (sum(sum(A, 2), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



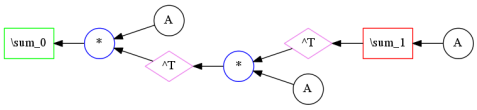

### k = 2

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((A * A'), 1), 2);

optimized = 1 * (sum((sum(A, 1) * A'), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



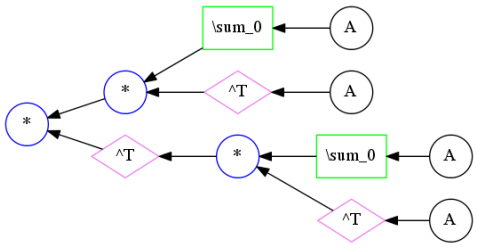

### k = 3

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((A * A') * A), 1), 2);

optimized = 1 * (sum((A * (sum(A, 2)' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



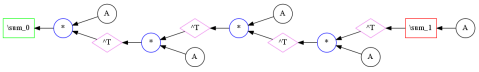

### k = 4

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((A * A') * A) * A'), 1), 2);

optimized = 1 * (((sum(A, 1) * A') * (sum(A, 1) * A')'));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



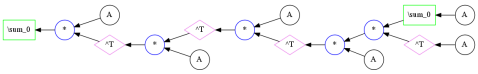

## k = 5

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((A * A') * A) * A') * A), 1), 2);

optimized = 1 * (sum((A * ((A * (sum(A, 2)' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



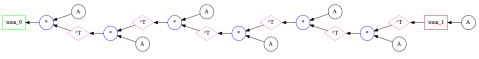
## k = 6

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((A * A') * A) * A') * A) * A'), 1), 2);

optimized = 1 * (sum((A * ((A * ((sum(A, 1) * A') * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



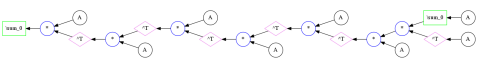
## k = 7

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((((A * A') * A) * A') * A) * A') * A), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * (sum(A, 2)' * A)')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 8

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((A * A') * A) * A') * A) * A') * A) * A'), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((sum(A, 1) * A') * A)')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

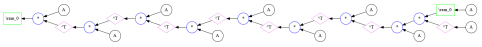


## k = 9

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((((((A * A') * A) * A') * A) * A') * A) * A') * A), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * (sum(A, 2)' * A)')' * A)')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

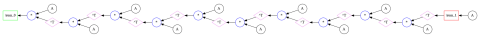
## k = 10

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A'), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((sum(A, 1) * A') * A)')' * A)')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



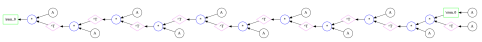
## k = 11

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A') * A), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((A * (sum(A, 2)' * A)')' * A)')' * A)')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 12

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A') * A) * A'), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((A * ((sum(A, 1) * A') * A)')' * A)')' * A)')' * A)')' * A)'), 1)
    ↪ );
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

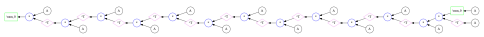


## k = 13

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A') * A) * A') * A), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((A * ((A * (sum(A, 2)' * A)')' * A)')' * A)')' * A)')' * A)')' *
    ↪ A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

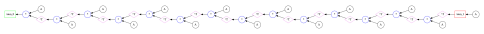
## k = 14

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A') * A) * A'), 1),
    ↪ 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((A * ((A * ((sum(A, 1) * A') * A)')' * A)')' * A)')' * A)')' * A)
    ↪ ')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



**k = 15**

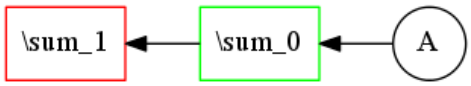
```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((((A * A') * A) * A') * A) * A') * A) * A') * A) * A') * A) * A') * A
    ↪ ), 1), 2);

optimized = 1 * (sum((A * ((A * ((A * ((A * ((A * ((A * ((A * (sum(A, 2)' * A)')' * A)')' * A)')' * A)')' * A)
    ↪ ')' * A)')' * A)'), 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## 9.2 $(\sum \mathbf{AB})_\mathbf{k}$

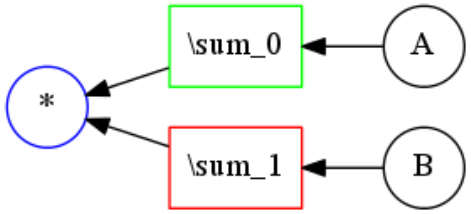
**k = 1**

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(A, 1), 2);

optimized = 1 * (sum(sum(A, 1), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



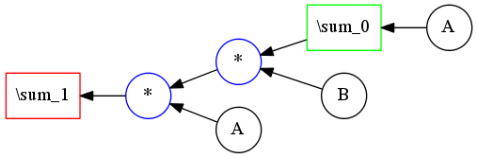

**k = 2**

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((A * B), 1), 2);

optimized = 1 * ((sum(A, 1) * sum(B, 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



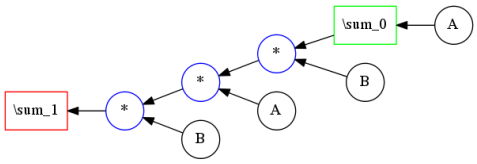

**k = 3**

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(((A * B) * A), 1), 2);

optimized = 1 * (sum(((sum(A, 1) * B) * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



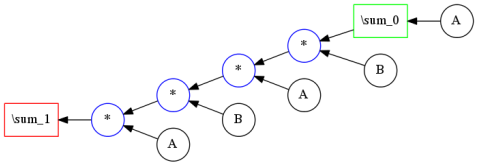

## k = 4

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((A * B) * A) * B), 1), 2);

optimized = 1 * (sum((((sum(A, 1) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



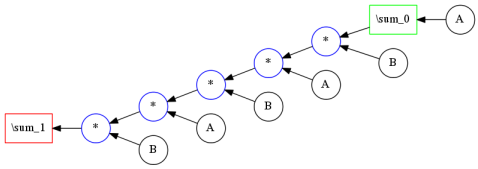

## k = 5

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(((((A * B) * A) * B) * A), 1), 2);

optimized = 1 * (sum(((((sum(A, 1) * B) * A) * B) * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



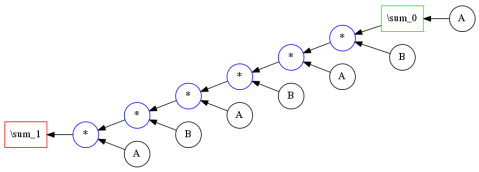

## k = 6

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((A * B) * A) * B) * A) * B), 1), 2);

optimized = 1 * (sum((((((sum(A, 1) * B) * A) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



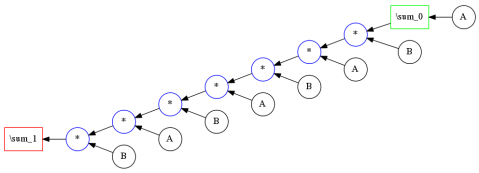

## k = 7

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(((((((A * B) * A) * B) * A) * B) * A), 1), 2);

optimized = 1 * (sum(((((((sum(A, 1) * B) * A) * B) * A) * B) * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



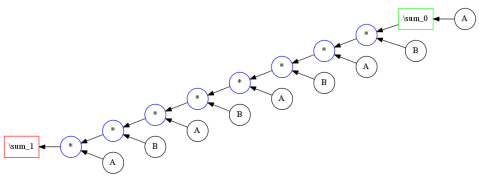

## k = 8

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(((((((A * B) * A) * B) * A) * B) * A) * B), 1), 2);

optimized = 1 * (sum(((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```




## k = 9

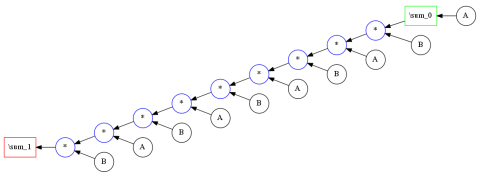
```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum(((((((((A * B) * A) * B) * A) * B) * A) * B) * A), 1), 2);

optimized = 1 * (sum(((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```




## k = 10

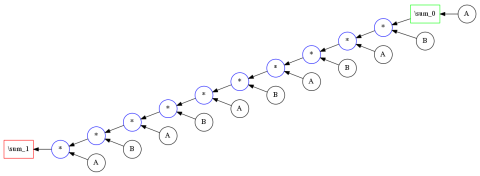
```
n = 100;
m = 200;
A = randn(n, m);
```

```
B = randn(m, n);
original = sum(sum((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B), 1), 2);

optimized = 1 * (sum((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



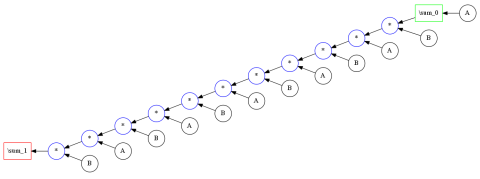
## k = 11

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B) * A), 1), 2);

optimized = 1 * (sum((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



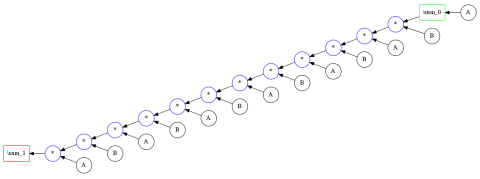
## k = 12

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B), 1), 2);

optimized = 1 * (sum((((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



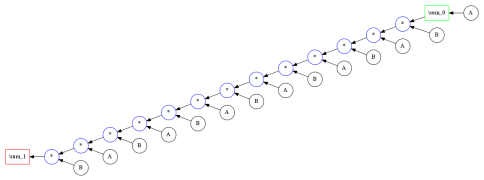
## k = 13

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A), 1), 2);

optimized = 1 * (sum((((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



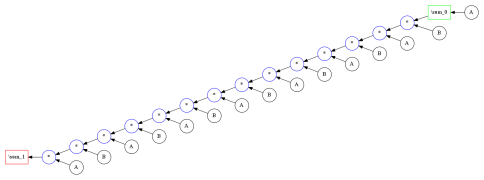
### k = 14

```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B), 1), 2);

optimized = 1 * (sum((((((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B),
    ↪ 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



### k = 15

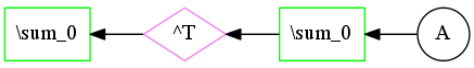
```
n = 100;
m = 200;
A = randn(n, m);
B = randn(m, n);
original = sum(sum((((((((((((((A * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A), 1),
    ↪ 2);

optimized = 1 * (sum((((((((((((((sum(A, 1) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B) * A) * B)
    ↪ * A), 2));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## 9.3 $(\sum(\mathbf{A}. * \mathbf{A})\mathbf{A}^{\mathbf{T}})_k$

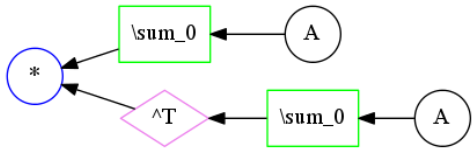
### k = 1

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(A, 1), 2);

optimized = 1 * (sum(sum(A, 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



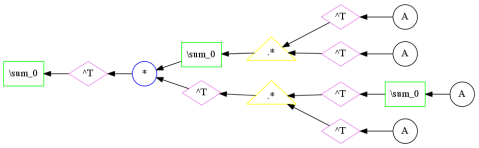

### k = 2

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((A * A'), 1), 2);

optimized = 1 * ((sum(A, 1) * sum(A, 1)'));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



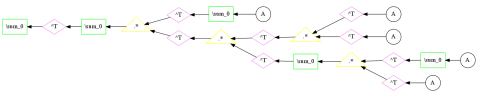

## k = 3

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((A * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* A'), 1) * (repmat(sum(A, 1)', 1, n) .* A')')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



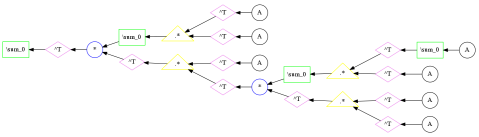
## k = 4

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((A * A') * (A .* A)) * A'), 1), 2);

optimized = 1 * (sum(sum((repmat(sum(A, 1)', 1, n) .* ((A' .* A')' .* repmat(sum((repmat(sum(A, 1)', 1, n) .*
    ↪ A'), 1)', 1, m))'), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



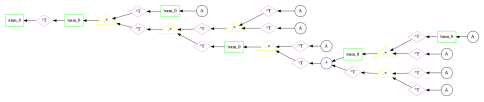
## k = 5

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((A * A') * (A .* A)) * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* A'), 1) * (A' .* repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A
    ↪ ')')', 1, n))')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



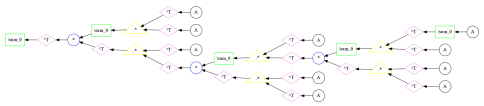
## k = 6

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((A * A') * (A .* A)) * A') * (A .* A)) * A'), 1), 2);

optimized = 1 * (sum(sum((repmat(sum(A, 1)', 1, n) .* ((A' .* A')' .* repmat(sum((A' .* repmat((sum((repmat(
    ↪ sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1)', 1, m))'), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



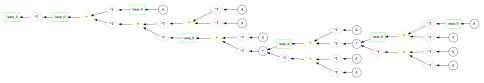
## k = 7

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* A'), 1) * (A' .* repmat((sum((A' .* repmat((sum((repmat(sum(A, 1)', 1, n) .*
    ↪ A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n))')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



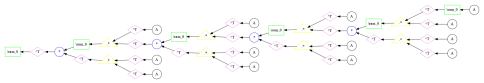
## k = 8

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum(((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A'), 1), 2);

optimized = 1 * (sum(sum((repmat(sum(A, 1)', 1, n) .* ((A' .* A')' .* repmat(sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)),
    ↪  1)', 1, m))'), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 9

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((repmat(sum(A, 1)', 1, n)
    ↪ .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A'
    ↪ .* A')')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 10

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A'),
    ↪ 1), 2);

optimized = 1 * (sum(sum((repmat(sum(A, 1)', 1, n) .* ((A' .* A')' .* repmat(sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((A' .* repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A'
    ↪  .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1)', 1, m))'), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

## k = 11

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') *
    ↪  (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((repmat(
    ↪ sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')',
    ↪  1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

## k = 12

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A')
    ↪ * (A .* A)) * A'), 1), 2);

optimized = 1 * (sum(sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)),
    ↪  1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

## k = 13

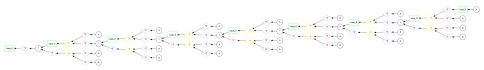
```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A')
    ↪  * (A .* A)) * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)),
    ↪  1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')
    ↪ ')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

## k = 14

```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A
    ↪ ') * (A .* A)) * A') * (A .* A)) * A'), 1), 2);

optimized = 1 * (sum(sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((A' .* repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A'
    ↪  .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)
    ↪ ), 1) * (A' .* A')')', 1, n)), 1)', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

## k = 15

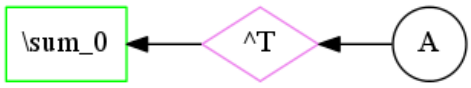
```
n = 100;
m = 200;
A = randn(n, m);
original = sum(sum((((((((((((((A * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)) * A
    ↪ ') * (A .* A)) * A') * (A .* A)) * A') * (A .* A)), 1), 2);

optimized = 1 * (sum((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .* repmat((sum((A' .*
    ↪ repmat((sum((A' .* repmat((sum((repmat(sum(A, 1)', 1, n) .* A'), 1) * (A' .* A')')', 1, n)), 1) * (A'
    ↪  .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1, n)
    ↪ ), 1) * (A' .* A')')', 1, n)), 1) * (A' .* A')')', 1));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## 9.4 $\mathrm{Sym}_k$

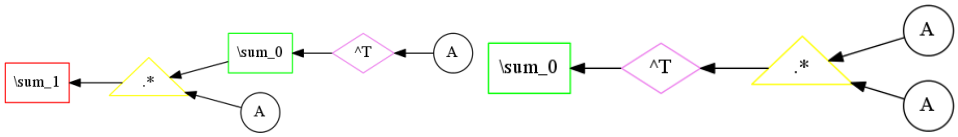
### k = 1

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 1);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (120 * (sum(A', 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



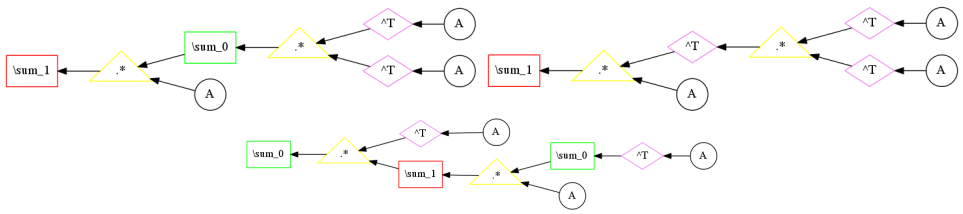

### k = 2

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 2);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (60 * (sum((repmat(sum(A', 1), 1, m) .* A), 2)) + -60 * (sum((A .* A)', 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



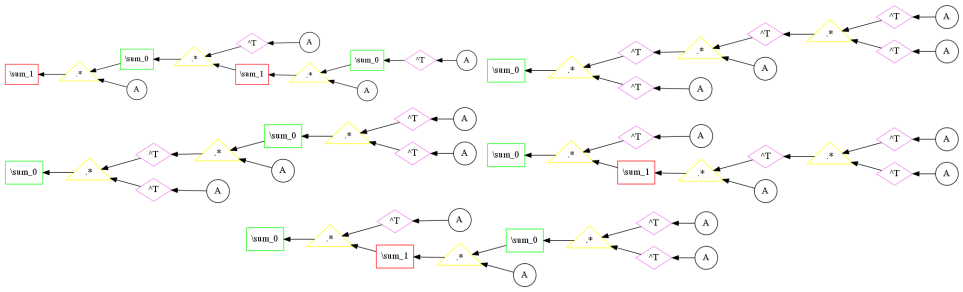

### k = 3

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 3);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (-60 * (sum((repmat(sum((A' .* A'), 1), 1, m) .* A), 2)) + 40 * (sum(((A' .* A')' .* A), 2)) + 20
    ↪ * (sum((A' .* repmat(sum((repmat(sum(A', 1), 1, m) .* A), 2), m, 1)), 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



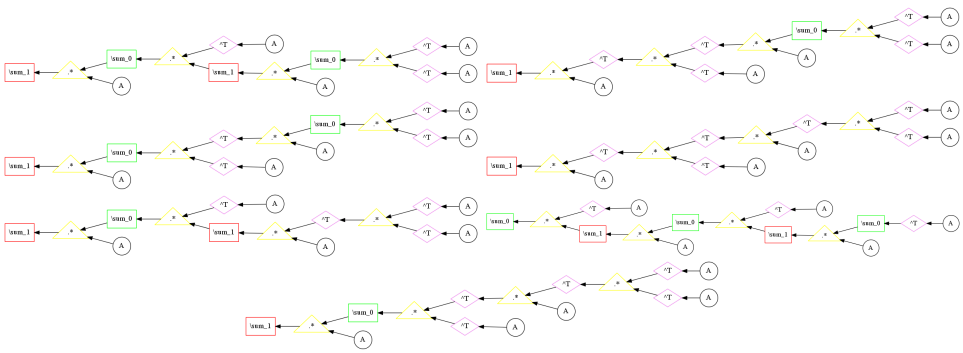

## k = 4

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 4);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (5 * (sum((repmat(sum((A' .* repmat(sum((repmat(sum(A', 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .*
      ↪ A), 2)) + -30 * (sum((((A' .* A')' .* A)' .* A'), 1)) + 15 * (sum(((repmat(sum((A' .* A'), 1), 1, m)
      ↪ .* A)' .* A'), 1)) + 40 * (sum((A' .* repmat(sum(((A' .* A')' .* A), 2), m, 1)), 1)) + -30 * (sum((A'
      ↪  .* repmat(sum((repmat(sum((A' .* A'), 1), 1, m) .* A), 2), m, 1)), 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



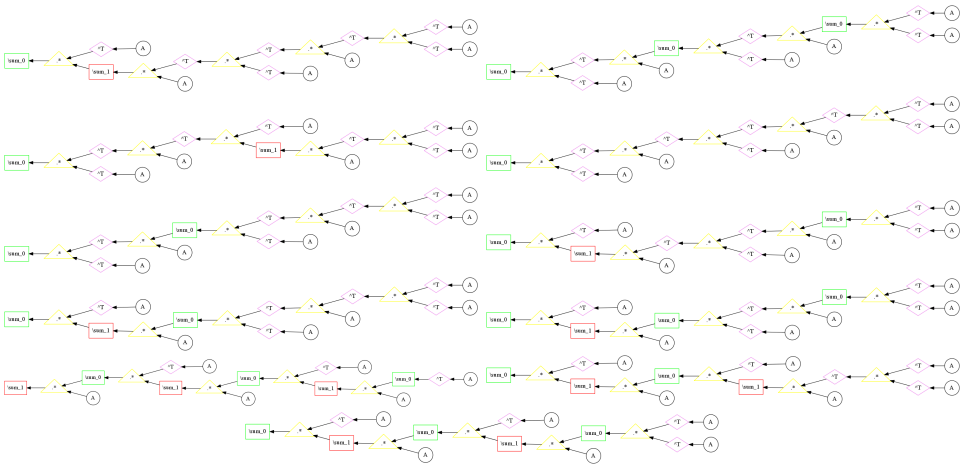

## k = 5

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 5);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (-10 * (sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* A'), 1), 1, m) .* A), 2), m, 1)), 1),
      ↪  1, m) .* A), 2)) + -20 * (sum((((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A')' .* A), 2)) + 15 * (
      ↪ sum((repmat(sum(((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2)) + 24 * (sum
      ↪ (((((A' .* A')' .* A)' .* A')' .* A), 2)) + 20 * (sum((repmat(sum((A' .* repmat(sum(((A' .* A')' .* A
      ↪ ), 2), m, 1)), 1), 1, m) .* A), 2)) + 1 * (sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((
      ↪ repmat(sum(A', 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + -30 * (sum((repmat(sum
      ↪ ((((A' .* A')' .* A)' .* A'), 1), 1, m) .* A), 2))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



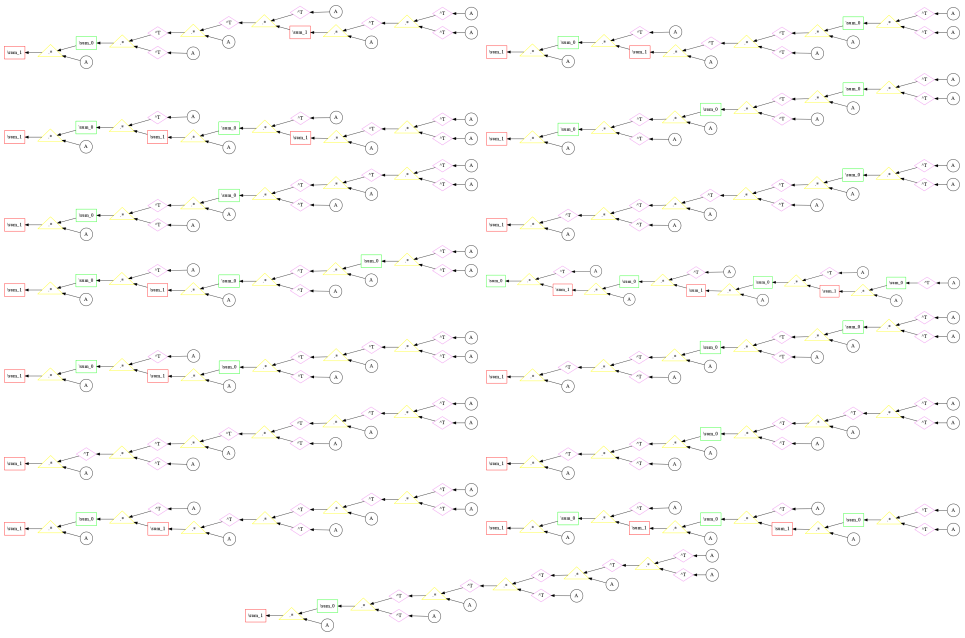
## k = 6

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 6);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (24 * (sum((A' .* repmat(sum(((((A' .* A')' .* A)' .* A')' .* A), 2), m, 1)), 1)) + -5 / 2 * (sum
    ↪ (((repmat(sum(((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A)' .* A'), 1)) + 20 / 3
    ↪  * (sum((((A' .* repmat(sum(((A' .* A')' .* A), 2), m, 1))' .* A)' .* A'), 1)) + -20 * (sum((((((A'
    ↪ .* A')' .* A)' .* A')' .* A)' .* A'), 1)) + 15 * (sum(((repmat(sum((((A' .* A')' .* A)' .* A'), 1),
    ↪ 1, m) .* A)' .* A'), 1)) + -20 * (sum((A' .* repmat(sum((((repmat(sum((A' .* A'), 1), 1, m) .* A)' .*
    ↪  A')' .* A), 2), m, 1)), 1)) + -15 * (sum((A' .* repmat(sum((repmat(sum((((A' .* A')' .* A)' .* A'),
    ↪ 1), 1, m) .* A), 2), m, 1)), 1)) + 15 / 2 * (sum((A' .* repmat(sum((repmat(sum(((repmat(sum((A' .* A
    ↪ '), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1)) + 1 / 6 * (sum((repmat(sum((A' .*
    ↪ repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum(A', 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A)
    ↪ , 2), m, 1)), 1), 1, m) .* A), 2)) + 20 / 3 * (sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum(((
    ↪ A' .* A')' .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + -5 / 2 * (sum((A' .* repmat(sum((
    ↪ repmat(sum((A' .* repmat(sum((repmat(sum((A' .* A'), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2)
    ↪ , m, 1)), 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



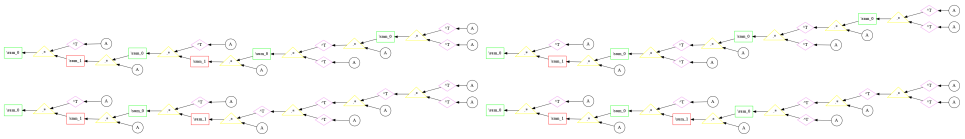
## k = 7

```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 7);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (20 / 3 * (sum((repmat(sum((((A' .* repmat(sum(((A' .* A')' .* A), 2), m, 1))' .* A)' .* A'), 1),
    ↪ 1, m) .* A), 2)) + -10 * (sum((repmat(sum((A' .* repmat(sum((((repmat(sum((A' .* A'), 1), 1, m) .* A)
    ↪ ' .* A')' .* A), 2), m, 1)), 1), 1, m) .* A), 2)) + 5 / 3 * (sum((repmat(sum((A' .* repmat(sum((
    ↪ repmat(sum((A' .* repmat(sum(((A' .* A')' .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m)
    ↪ .* A), 2)) + -5 / 2 * (sum((repmat(sum(((repmat(sum(((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'),
    ↪  1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2)) + 15 * (sum((repmat(sum((((A' .* A')' .* A)' .*
    ↪ A)' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2)) + -12 * (sum((((((repmat(sum((A' .* A'), 1)
    ↪ , 1, m) .* A)' .* A')' .* A)' .* A')' .* A), 2)) + 5 / 2 * (sum((repmat(sum((A' .* repmat(sum((repmat
    ↪ (sum(((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A),
    ↪ 2)) + 1 / 42 * (sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((
    ↪ repmat(sum(A', 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)),
    ↪  1)) + -5 * (sum((repmat(sum((A' .* repmat(sum((repmat(sum((((A' .* A')' .* A)' .* A'), 1), 1, m) .*
    ↪ A), 2), m, 1)), 1), 1, m) .* A), 2)) + 5 * (sum((((repmat(sum(((repmat(sum((A' .* A'), 1), 1, m) .* A
    ↪ )' .* A'), 1), 1, m) .* A)' .* A')' .* A), 2)) + 120 / 7 * (sum(((((((A' .* A')' .* A)' .* A')' .* A)
    ↪ ' .* A')' .* A), 2)) + -10 * (sum((((repmat(sum((((A' .* A')' .* A)' .* A'), 1), 1, m) .* A)' .* A')'
    ↪  .* A), 2)) + 12 * (sum((repmat(sum((A' .* repmat(sum(((((A' .* A')' .* A)' .* A')' .* A), 2), m, 1))
    ↪ , 1), 1, m) .* A), 2)) + -1 / 2 * (sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((
    ↪ repmat(sum((A' .* A'), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2))
    ↪  + -20 * (sum((repmat(sum((((((A' .* A')' .* A)' .* A')' .* A)' .* A'), 1), 1, m) .* A), 2))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 8

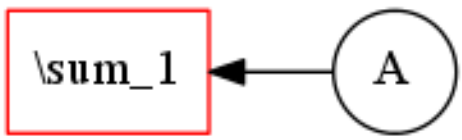
```
n = 1;
m = 18;
A = randn(1, m);
sub = nchoosek(1:m, 8);
original = 0;
for i = 1:size(sub, 1)
  original = original + prod(A(sub(i, :)));
end

optimized = (5 / 8 * (sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum(((repmat(sum((A' .* A'),
      ↪  1), 1, m) .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + -5 / 4 * (
      ↪ sum((A' .* repmat(sum((repmat(sum(((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'), 1),
      ↪ 1, m) .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1)) + 4 * (sum((A' .* repmat(sum((repmat(sum((A' .*
      ↪  repmat(sum(((((A' .* A')' .* A)' .* A')' .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + -5 /
      ↪ 4 * (sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum((((A' .* A')' .* A)' .* A'), 1),
      ↪  1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + 5 / 16 * (sum(((repmat(sum(((repmat(sum
      ↪ (((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A)' .* A'),
      ↪ 1)) + 15 / 2 * (sum((A' .* repmat(sum((repmat(sum(((repmat(sum((((A' .* A')' .* A)' .* A'), 1), 1, m)
      ↪  .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1)) + 8 * (sum((((A' .* repmat(sum(((((A' .* A')' .* A)'
      ↪  .* A')' .* A), 2), m, 1))' .* A)' .* A'), 1)) + 1 / 3 * (sum((A' .* repmat(sum((repmat(sum((A' .*
      ↪ repmat(sum((repmat(sum((A' .* repmat(sum(((A' .* A')' .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)),
      ↪  1), 1, m) .* A), 2), m, 1)), 1)) + 120 / 7 * (sum((A' .* repmat(sum(((((((A' .* A')' .* A)' .* A')'
      ↪ .* A)' .* A')' .* A), 2), m, 1)), 1)) + 5 * (sum((A' .* repmat(sum((((repmat(sum(((repmat(sum((A' .*
      ↪ A'), 1), 1, m) .* A)' .* A'), 1), 1, m) .* A)' .* A')' .* A), 2), m, 1)), 1)) + -10 * (sum((A' .*
      ↪ repmat(sum((repmat(sum((((((A' .* A')' .* A)' .* A')' .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1))
      ↪  + -15 / 4 * (sum(((repmat(sum(((repmat(sum((((A' .* A')' .* A)' .* A'), 1), 1, m) .* A)' .* A'), 1),
      ↪  1, m) .* A)' .* A'), 1)) + 10 / 3 * (sum((A' .* repmat(sum((repmat(sum((((A' .* repmat(sum(((A' .* A
      ↪ ')' .* A), 2), m, 1))' .* A)' .* A'), 1), 1, m) .* A), 2), m, 1)), 1)) + 15 / 4 * (sum(((((repmat(sum
      ↪ ((((A' .* A')' .* A)' .* A'), 1), 1, m) .* A)' .* A')' .* A)' .* A'), 1)) + -1 / 12 * (sum((A' .*
      ↪ repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* A'), 1), 1,
      ↪  m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1)) + -15 * (sum
      ↪ (((((((A' .* A')' .* A)' .* A')' .* A)' .* A')' .* A)' .* A'), 1)) + -10 / 3 * (sum((A' .* repmat(
      ↪ sum((repmat(sum((A' .* repmat(sum((((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A')' .* A), 2), m, 1)
      ↪ ), 1), 1, m) .* A), 2), m, 1)), 1)) + -10 * (sum((A' .* repmat(sum((((repmat(sum((((A' .* A')' .* A)'
      ↪  .* A'), 1), 1, m) .* A)' .* A')' .* A), 2), m, 1)), 1)) + 10 * (sum(((repmat(sum((((((A' .* A')' .*
      ↪ A)' .* A')' .* A)' .* A'), 1), 1, m) .* A)' .* A')' .* A)' .* A')' .* A), 2), m, 1)), 1)) + 1 / 336 * (sum((
      ↪ repmat(sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum((A' .* repmat(sum((repmat(sum(
      ↪ A', 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m) .* A), 2), m, 1)), 1), 1, m)
      ↪ .* A), 2)) + -10 / 3 * (sum((((A' .* repmat(sum((((repmat(sum((A' .* A'), 1), 1, m) .* A)' .* A')' .*
      ↪  A), 2), m, 1))' .* A)' .* A'), 1))) / 120;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## 9.5 $(\text{RBM-1})_k$

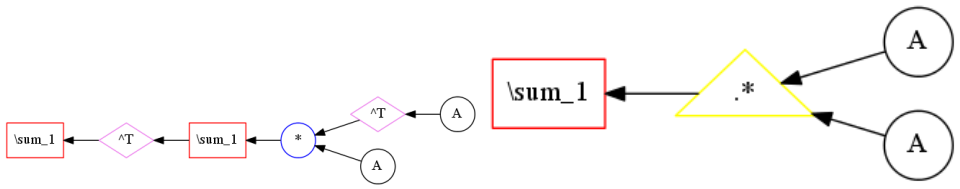
### k = 1

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 1;
end

optimized = 2^(n - 3) * (4 * (sum(A, 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



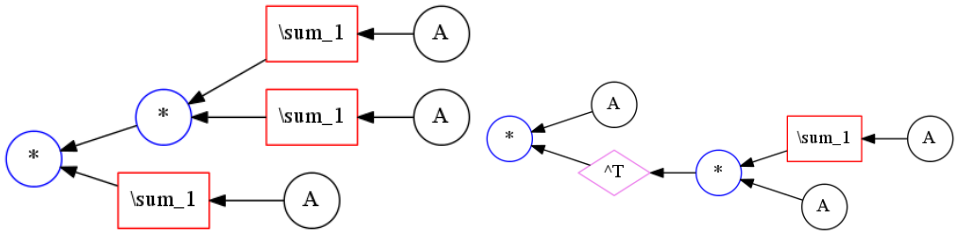

### k = 2

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 2;
end

optimized = 2^(n - 3) * (2 * (sum(sum((A' * A), 2)', 2)) + 2 * (sum((A .* A), 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



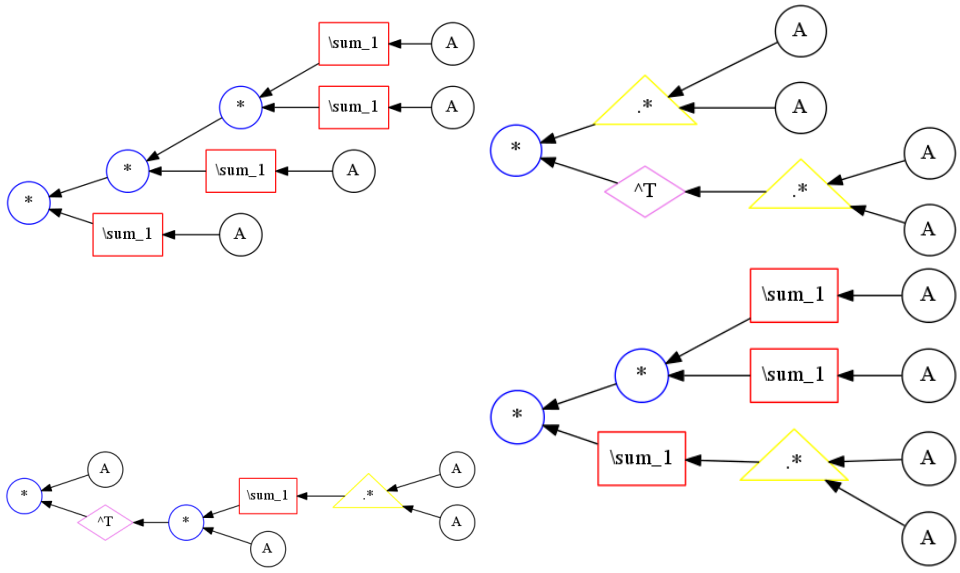

**k = 3**

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 3;
end

optimized = 2^(n - 4) * (2 * (((sum(A, 2) * sum(A, 2)) * sum(A, 2))) + 6 * ((A * (sum(A, 2) * A)')));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



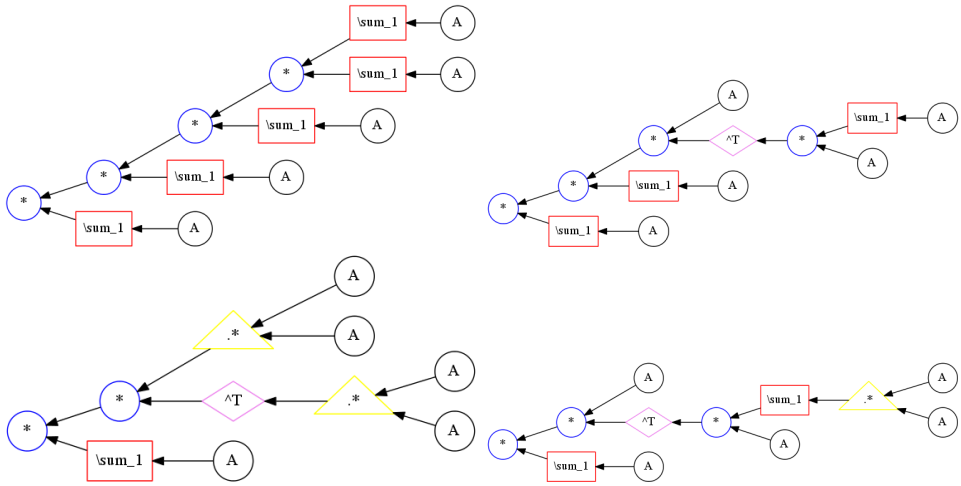

**k = 4**

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 4;
end

optimized = 2^(n - 5) * (2 * ((((sum(A, 2) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + -4 * (((A .* A) * (A .* A
    ↪ )')) + 6 * ((A * (sum((A .* A), 2) * A)')) + 12 * (((sum(A, 2) * sum(A, 2)) * sum((A .* A), 2))));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```




## k = 5

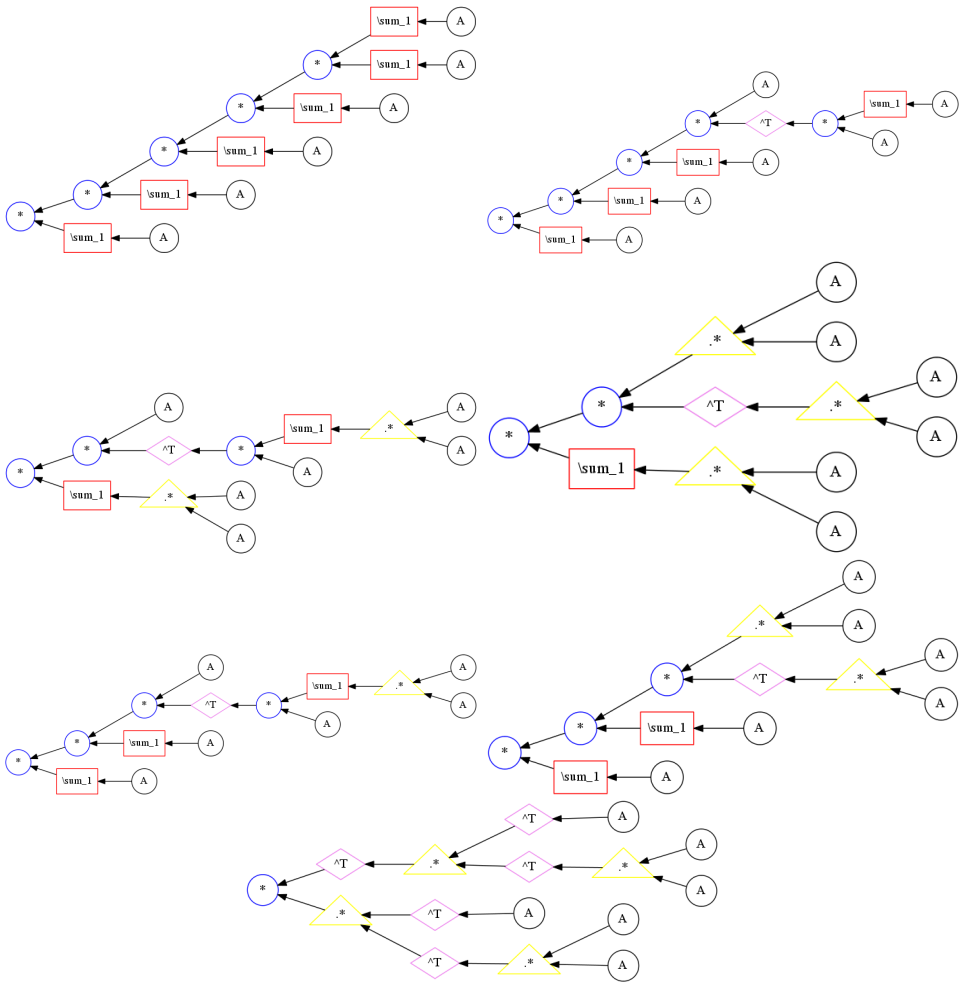
```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 5;
end

optimized = 2^(n - 6) * (2 * (((((sum(A, 2) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + 20 * ((((A
    ↪ * (sum(A, 2) * A)') * sum(A, 2)) * sum(A, 2))) + -20 * ((((A .* A) * (A .* A)') * sum(A, 2))) + 30 *
    ↪ (((A * (sum((A .* A), 2) * A)') * sum(A, 2))));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```




## k = 6

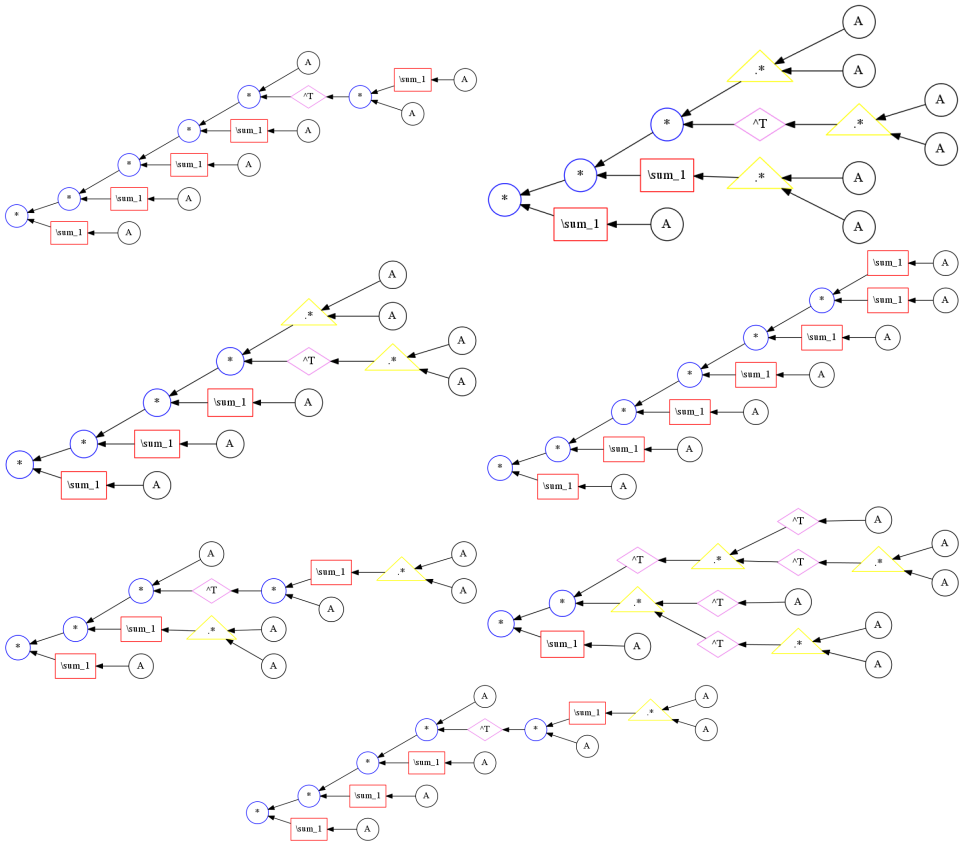
```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 6;
end
```

```
optimized = 2^(n - 7) * (2 * ((((((sum(A, 2) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)))
    ↪  + 30 * (((((A * (sum(A, 2) * A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + 30 * (((A * (sum((A .* A
    ↪ ), 2) * A)') * sum((A .* A), 2))) + -60 * ((((A .* A) * (A .* A)') * sum((A .* A), 2))) + 90 * ((((A
    ↪ * (sum((A .* A), 2) * A)') * sum(A, 2)) * sum(A, 2))) + -60 * (((((A .* A) * (A .* A)') * sum(A, 2))
    ↪ * sum(A, 2))) + 32 * (((A' .* (A .* A)')' * (A' .* (A .* A)'))));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



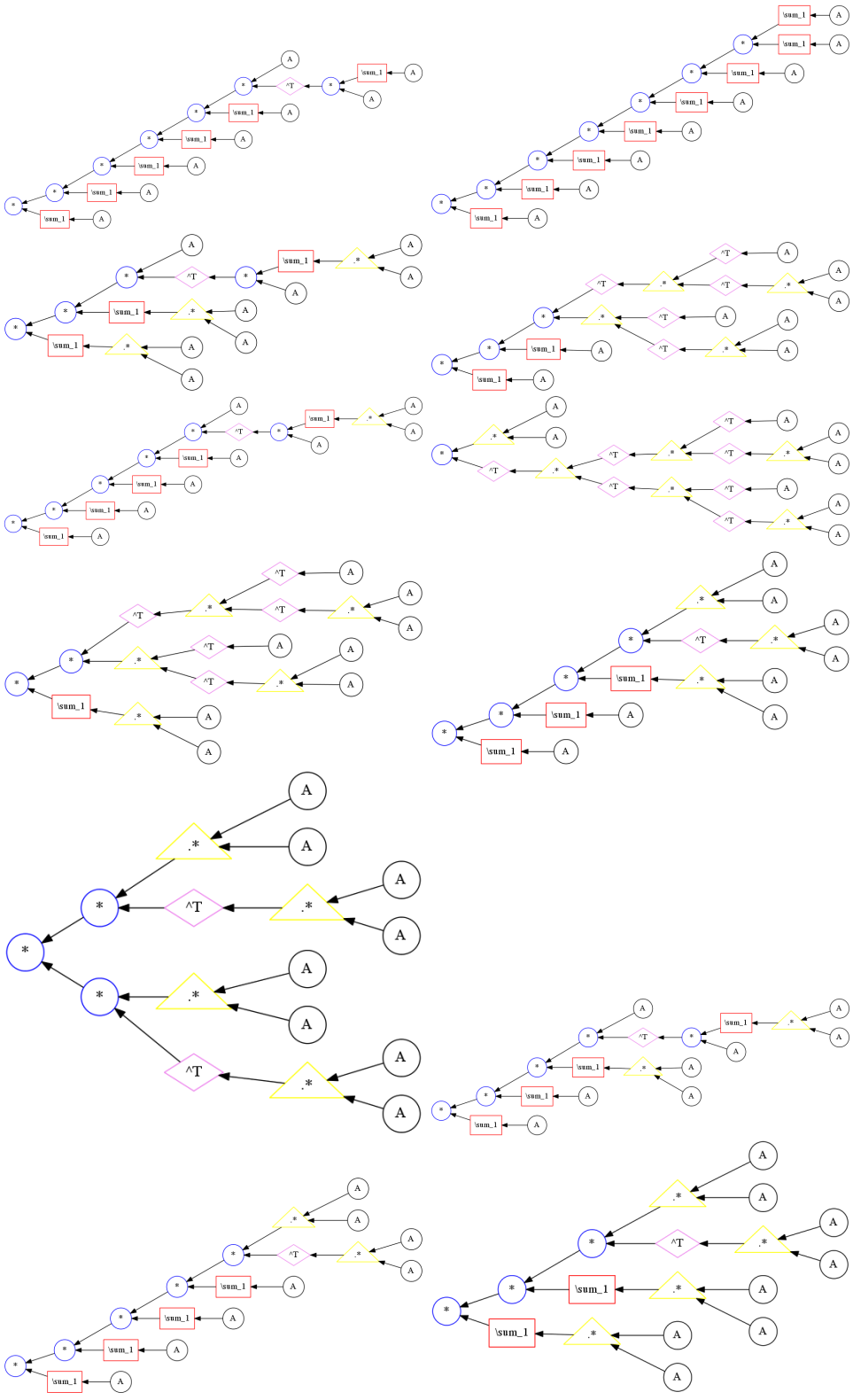

## k = 7

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 7;
end

optimized = 2^(n - 8) * (42 * ((((((A * (sum(A, 2) * A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)))
    ↪  + -420 * (((((A .* A) * (A .* A)') * sum((A .* A), 2)) * sum(A, 2))) + -140 * ((((((A .* A) * (A .*
    ↪ A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + 2 * (((((((sum(A, 2) * sum(A, 2)) * sum(A, 2)) * sum(A
    ↪ , 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + 210 * ((((A * (sum((A .* A), 2) * A)') * sum((A .* A)
    ↪ , 2)) * sum(A, 2))) + 224 * ((((A' .* (A .* A)')' * (A' .* (A .* A)')) * sum(A, 2))) + 210 * (((((A *
    ↪  (sum((A .* A), 2) * A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2))));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



**k = 8**

```
n = 14;
m = 1;
A = randn(1, n);
nset = dec2bin(0:(2^(n) - 1));
original = 0;
for i = 1:size(nset, 1)
  v = logical(nset(i, :) - '0');
  original = original + (v * A') ^ 8;
end

optimized = 2^(n - 9) * (56 * (((((((A * (sum(A, 2) * A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))
      ↪  * sum(A, 2))) + 2 * (((((((sum(A, 2) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A,
      ↪ 2)) * sum(A, 2)) * sum(A, 2))) + 210 * ((((A * (sum((A .* A), 2) * A)') * sum((A .* A), 2)) * sum((A
      ↪ .* A), 2))) + 896 * (((((A' .* (A .* A)')' * (A' .* (A .* A)')) * sum(A, 2)) * sum(A, 2))) + 420 *
      ↪ ((((((A * (sum((A .* A), 2) * A)') * sum(A, 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + -544 * (((A
      ↪  .* A) * ((A' .* (A .* A)')' .* (A' .* (A .* A)')')')) + 896 * ((((A' .* (A .* A)')' * (A' .* (A .* A
      ↪ )')) * sum((A .* A), 2))) + -1680 * ((((((A .* A) * (A .* A)') * sum((A .* A), 2)) * sum(A, 2)) * sum
      ↪ (A, 2))) + 280 * ((((A .* A) * (A .* A)') * ((A .* A) * (A .* A)'))) + 840 * (((((A * (sum((A .* A),
      ↪ 2) * A)') * sum((A .* A), 2)) * sum(A, 2)) * sum(A, 2))) + -280 * (((((((A .* A) * (A .* A)') * sum(A
      ↪ , 2)) * sum(A, 2)) * sum(A, 2)) * sum(A, 2))) + -840 * (((((A .* A) * (A .* A)') * sum((A .* A), 2))
      ↪ * sum((A .* A), 2))));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

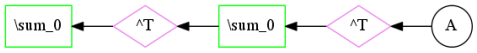


## 9.6 $(\text{RBM-2})_k$

### k = 1

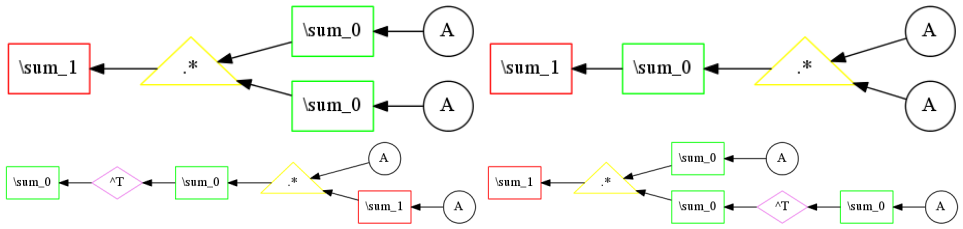
```
n = 7;
m = 8;
```

```
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 1;
  end
end

optimized = 2^(n + m - 5) * (8 * (sum(sum(A', 1)', 1)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



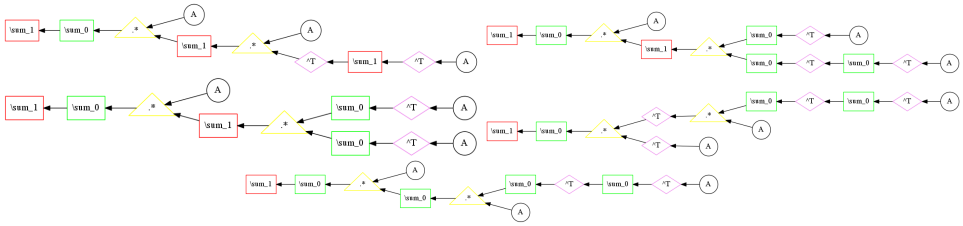

## k = 2

```
n = 7;
m = 8;
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 2;
  end
end

optimized = 2^(n + m - 5) * (2 * (sum((sum(A, 1) .* sum(A, 1)), 2)) + 2 * (sum(sum((A .* A), 1), 2)) + 2 * (
    ↪ sum(sum((A .* repmat(sum(A, 2), 1, m)), 1)', 1)) + 2 * (sum((sum(A, 1) .* repmat(sum(sum(A, 1)', 1),
    ↪ 1, m)), 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



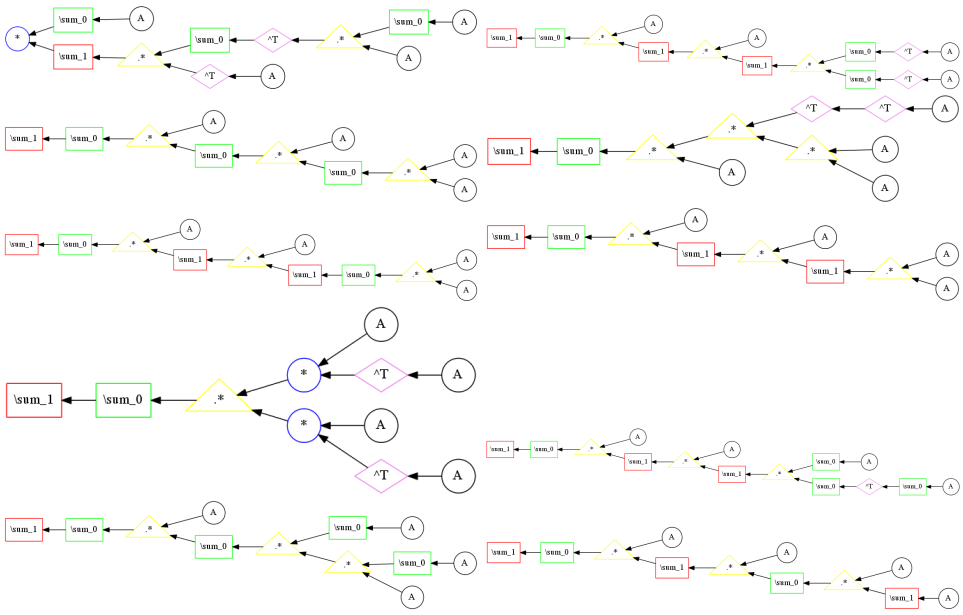

## k = 3

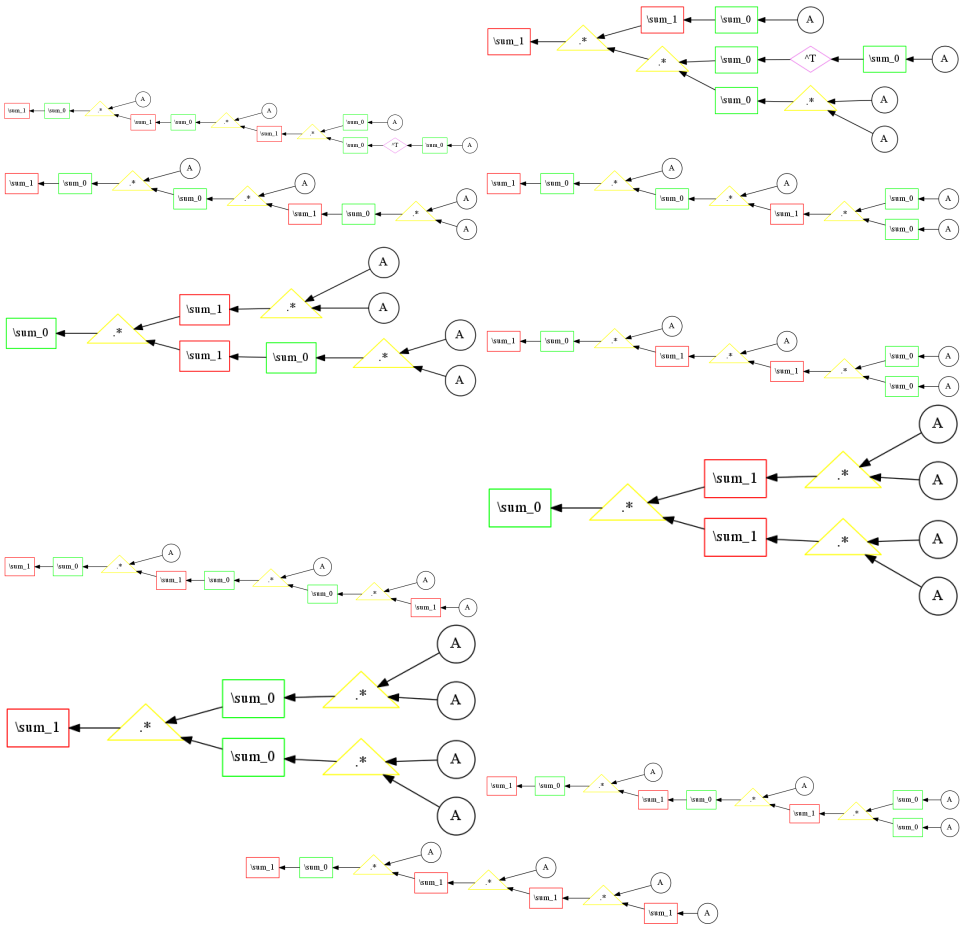
```
n = 7;
m = 8;
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 3;
  end
end

optimized = 2^(n + m - 7) * (12 * (sum(sum((A .* repmat(sum((A .* repmat(sum(A', 2)', n, 1)), 2), 1, m)), 1),
    ↪ 2)) + 2 * (sum(sum((A .* repmat(sum((sum(A', 1) .* repmat(sum(sum(A', 1)', 1), 1, n)), 2), n, m)), 1)
    ↪ , 2)) + 6 * (sum(sum((A .* repmat(sum((sum(A', 1) .* sum(A', 1)), 2), n, m)), 1), 2)) + 6 * (sum(sum
    ↪ (((repmat(sum(sum(A', 1)', 1), n, m) .* A)' .* A'), 1), 2)) + 6 * (sum(sum((A .* repmat(sum((repmat(
    ↪ sum(sum(A', 1)', 1), n, m) .* A), 1), n, 1)), 1), 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```



## k = 4

```
n = 7;
m = 8;
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 4;
  end
end

optimized = 2^(n + m - 9) * (24 * ((sum(A, 1) * sum((repmat(sum((repmat(sum(A, 1), n, 1) .* A)', 1), m, 1) .*
      ↪ A'), 2))) + 6 * (sum(sum((A .* repmat(sum((A .* repmat(sum((sum(A', 1) .* sum(A', 1)), 2), n, m)), 2)
      ↪ , 1, m)), 1), 2)) + -24 * (sum(sum((A .* repmat(sum((A .* repmat(sum((A .* A), 1), n, 1)), 1), n, 1))
      ↪ , 1), 2)) + 8 * (sum(sum(((A' .* (A .* A)) .* A), 1), 2)) + 12 * (sum(sum((A .* repmat(sum((A .*
      ↪ repmat(sum(sum((A .* A), 1), 2), n, m)), 2), 1, m)), 1), 2)) + -4 * (sum(sum((A .* repmat(sum((repmat(sum(A, 1), n, 1) .* (repmat(
      ↪ sum(A, 1), n, 1) .* A)), 1), n, 1)), 1), 2)) + 24 * (sum(sum((A .* repmat(sum((A .* repmat(sum((A .*
      ↪ repmat(sum(A, 2), 1, m)), 1), n, 1)), 2), 1, m)), 1), 2)) + 2 * (sum(sum((A .* repmat(sum(sum((A .*
      ↪ repmat(sum((sum(A, 1) .* repmat(sum(sum(A, 1)', 1), 1, m)), 2), n, m)), 1), 2), n, m)), 1), 2)) + 12
      ↪ * (sum((repmat(sum(sum(A, 1), 2), 1, m) .* (repmat(sum(sum(A, 1)', 1), 1, m) .* sum((A .* A), 1))),
      ↪ 2)) + 12 * (sum(sum((A .* repmat(sum((A .* repmat(sum(sum((A .* A), 1), 2), n, m)), 1), n, 1)), 1),
      ↪ 2)) + 6 * (sum(sum((A .* repmat(sum((A .* repmat(sum((sum(A, 1) .* sum(A, 1)), 2), n, m)), 1), n, 1))
      ↪ , 1), 2)) + 6 * (sum((sum((A .* A), 2) .* repmat(sum(sum((A .* A), 1), 2), n, 1)), 1)) + 12 * (sum(
      ↪ sum((A .* repmat(sum((A .* repmat(sum((sum(A, 1) .* sum(A, 1)), 2), n, m)), 2), 1, m)), 1), 2)) + 48
      ↪ * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((A .* repmat(sum(A, 2), 1, m)), 1), n, 1)), 1), 2),
      ↪ n, m)), 1), 2)) + -12 * (sum((sum((A .* A), 2) .* sum((A .* A), 2)), 1)) + -12 * (sum((sum((A .* A),
      ↪ 1) .* sum((A .* A), 1)), 2)) + 12 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((sum(A, 1) .* sum(
      ↪ A, 1)), 2), n, m)), 1), 2), n, m)), 1), 2)) + -4 * (sum(sum((A .* repmat(sum((A .* repmat(sum((A .*
      ↪ repmat(sum(A, 2), 1, m)), 2), 1, m)), 2), 1, m)), 1), 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

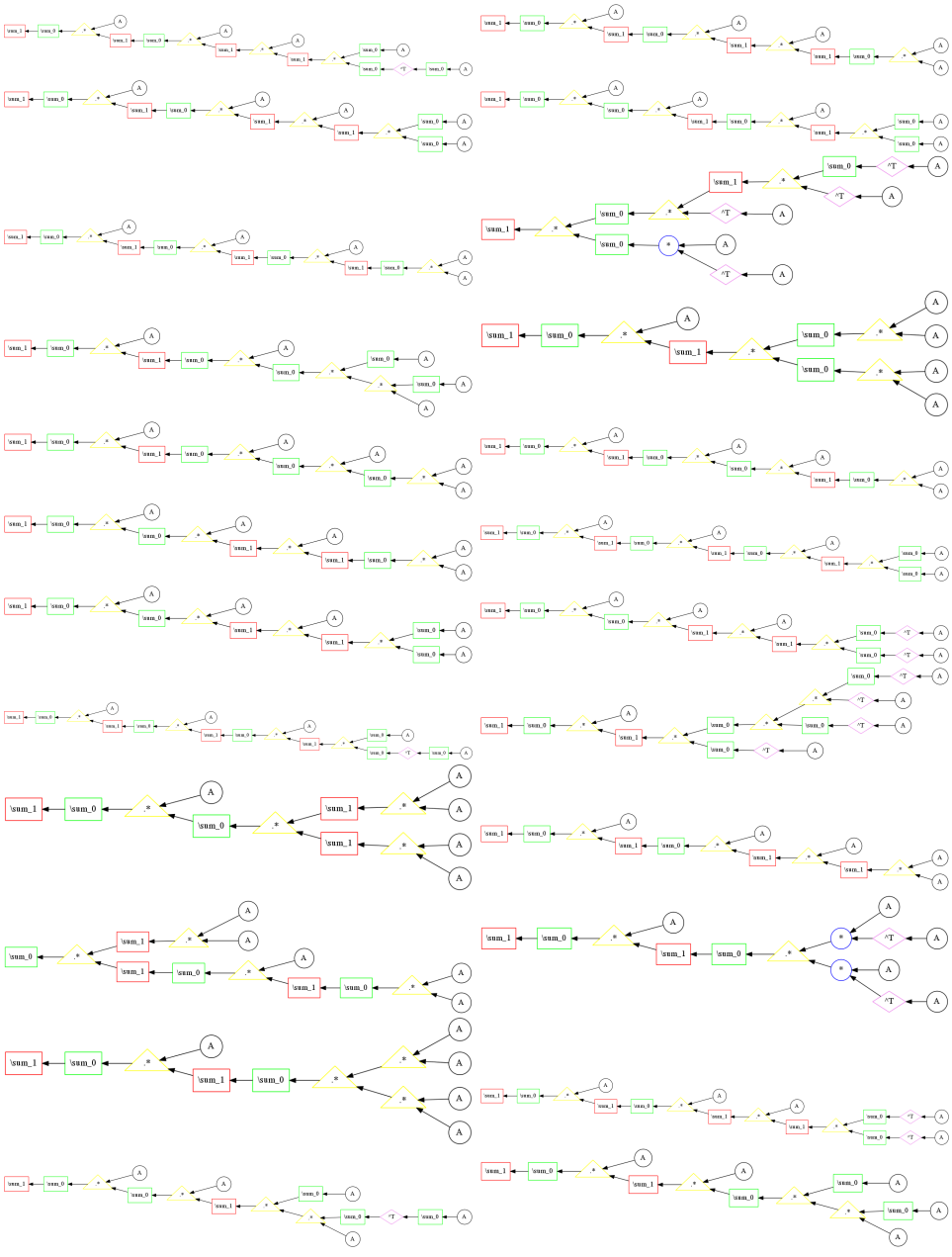


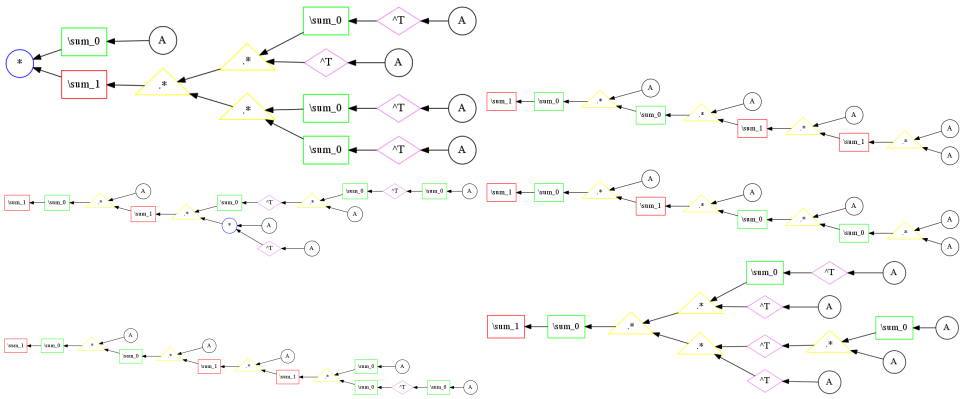


## k = 5

```
n = 7;
m = 8;
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 5;
  end
end

optimized = 2^(n + m - 11) * (20 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((A .* repmat(sum((sum(A, 1)
      ↪ .* repmat(sum(sum(A, 1)', 1), 1, m)), 2), n, m)), 2), 1, m)), 1), 2), n, m)), 1), 2)) + 60 * (sum(sum
      ↪ ((A .* repmat(sum(sum((A .* repmat(sum((A .* repmat(sum(sum((A .* A), 1), 2), n, m)), 2), 1, m)), 1),
      ↪  2), n, m)), 1), 2)) + 60 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((A .* repmat(sum((sum(A,
      ↪ 1) .* sum(A, 1)), 2), n, m)), 2), 1, m)), 1), 2), n, m)), 1), 2)) + 30 * (sum(sum((A .* repmat(sum((A
      ↪  .* repmat(sum(sum((A .* repmat(sum((sum(A, 1) .* sum(A, 1)), 2), n, m)), 1), 2), n, m)), 1), n, 1)),
      ↪  1), 2)) + 20 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum(sum((A .* A), 1),
      ↪  2), n, m)), 1), 2), n, m)), 1), 2), n, m)), 1), 2)) + 240 * (sum((sum((repmat(sum((repmat(sum(A', 1)
      ↪ , m, 1) .* A'), 2), 1, n) .* A'), 1) .* sum((A * A'), 1)), 2)) + -20 * (sum(sum((A .* repmat(sum(sum
      ↪ ((A .* repmat(sum((repmat(sum(A, 1), n, 1) .* (repmat(sum(A, 1), n, 1) .* A)), 1), n, 1)), 1), 2), n,
      ↪  m)), 1), 2)) + -60 * (sum(sum((A .* repmat(sum((sum((A .* A), 1) .* sum((A .* A), 1)), 2), n, m)),
      ↪ 1), 2)) + -120 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((A .* repmat(sum((A .* A), 1), n, 1))
      ↪ , 1), n, 1)), 1), 2), n, m)), 1), 2)) + 60 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum((A .*
      ↪ repmat(sum((A .* A), 1), 2), n, m)), 1), n, 1)), 1), 2), n, m)), 1), 2)) + 120 * (sum(sum((A .*
      ↪ repmat(sum((A .* repmat(sum((A .* repmat(sum(sum((A .* A), 1), 2), n, m)), 2), 1, m)), 1), n, 1)), 1)
      ↪ , 2)) + 20 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum(sum((A .* repmat(sum((sum(A, 1) .* sum(A,
      ↪  1)), 2), n, m)), 1), 2), n, m)), 1), 2), n, m)), 1), 2)) + 120 * (sum(sum((A .* repmat(sum((A .*
      ↪ repmat(sum((A .* repmat(sum((sum(A, 1) .* sum(A, 1)), 2), n, m)), 2), 1, m)), 1), n, 1)), 1), 2)) +
      ↪ 120 * (sum(sum((A .* repmat(sum((A .* repmat(sum((A .* repmat(sum((sum(A', 1) .* sum(A', 1)), 2), n,
      ↪ m)), 2), 1, m)), 1), n, 1)), 1), 2)) + 2 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum(sum((A .*
      ↪ repmat(sum((sum(A, 1) .* repmat(sum(sum(A, 1)', 1), 1, m)), 2), n, m)), 1), 2), n, m)), 1), 2), n, m)
      ↪ ), 1), 2)) + -20 * (sum(sum((A .* repmat(sum((sum(((repmat(sum(A', 1), m, 1) .* A') .* repmat(sum(A',
      ↪  1), m, 1)), 1) .* sum(A', 1)), 2), n, m)), 1), 2)) + -60 * (sum(sum((A .* repmat(sum((sum((A .* A),
      ↪ 2) .* sum((A .* A), 2)), 1), n, m)), 1), 2)) + -120 * (sum(sum((A .* repmat(sum(sum((A .* repmat(sum
      ↪ ((A .* repmat(sum((A .* A), 2), 1, m)), 2), 1, m)), 1), 2), n, m)), 1), 2)) + 30 * (sum((sum((A .* A)
```

```
    ↪ , 2) .* repmat(sum(sum((A .* repmat(sum(sum((A .* A), 1), 2), n, m)), 1), 2), n, 1)), 1)) + 60 * (sum
    ↪ (sum((A .* repmat(sum(sum(((A * A') .* (A * A')), 1), 2), n, m)), 1), 2)) + 40 * (sum(sum((A .*
    ↪ repmat(sum(sum(((A .* A) .* (A .* A)), 1), 2), n, m)), 1), 2)) + 30 * (sum(sum((A .* repmat(sum(sum((
    ↪ A .* repmat(sum((A .* repmat(sum((sum(A', 1) .* sum(A', 1)), 2), n, m)), 2), 1, m)), 1), 2), n, m)),
    ↪ 1), 2)) + 120 * (sum(sum((A .* repmat(sum((A .* repmat(sum((repmat(sum(A, 1), n, 1) .* (repmat(sum(
    ↪ sum(A, 1)', 1), n, m) .* A)), 2), 1, m)), 1), n, 1)), 1), 2)) + -80 * (sum(sum((A .* repmat(sum((A .*
    ↪  repmat(sum((repmat(sum(A, 1), n, 1) .* (repmat(sum(A, 1), n, 1) .* A)), 1), n, 1)), 2), 1, m)), 1),
    ↪ 2)) + -80 * ((sum(A, 1) * sum(((repmat(sum(A', 1), m, 1) .* A') .* repmat((sum(A', 1) .* sum(A', 1)),
    ↪  m, 1)), 2))) + -240 * (sum(sum((A .* repmat(sum((A .* repmat(sum((A .* repmat(sum((A .* A), 2), 1, m
    ↪ )), 2), 1, m)), 1), n, 1)), 1), 2)) + 120 * (sum(sum((A .* repmat(sum((repmat(sum((repmat(sum(sum(A,
    ↪ 1)', 1), n, m) .* A)', 1), n, 1) .* (A * A')), 2), 1, m)), 1), 2)) + -240 * (sum(sum((A .* repmat(sum
    ↪ ((A .* repmat(sum((A .* repmat(sum((A .* A), 1), n, 1)), 1), n, 1)), 2), 1, m)), 1), 2)) + 120 * (sum
    ↪ (sum((A .* repmat(sum((A .* repmat(sum((A .* repmat(sum((sum(A, 1) .* repmat(sum(sum(A, 1)', 1), 1, m
    ↪ )), 2), n, m)), 2), 1, m)), 1), n, 1)), 1), 2)) + 160 * (sum(sum(((repmat(sum(A', 1), m, 1) .* A') .*
    ↪  ((repmat(sum(A, 1), n, 1) .* A)' .* A')), 1), 2)));
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```






## k = 6

```
n = 8;
m = 9;
A = randn(n, m);
nset = dec2bin(0:(2^(n) - 1));
mset = dec2bin(0:(2^(m) - 1));
original = 0;
for i = 1:size(nset, 1)
  for j = 1:size(mset, 1)
    v = logical(nset(i, :) - '0');
    h = logical(mset(j, :) - '0');
    original = original + (v * A * h') ^ 6;
  end
end

optimized = 2^(n+m)*(((sum(sum((( A .* A)  .* (( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))  .* (
      ↪  repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])))), 2), 1)  .* 360) ...
  + (sum(sum((( A .* repmat(sum(A, 2), [1, m]))  .* ( ( A .* repmat(sum(A, 2), [1, m])) .* repmat(sum(( A .* A
      ↪ ), 1), [n, 1]))), 2), 1)  .* 360) ...
  + (sum((( sum(A, 1) .* sum(A, 1))  .* ( ( sum(A, 1) .* sum(A, 1)) .* ( sum(A, 1) .* sum(A, 1)))), 2)  .* 16)
      ↪  ...
  + ((sum(( sum(A, 1) .* sum(A, 1)), 2)  .* sum(( ( sum(A, 1) .* sum(A, 1)) .* ( sum(A, 1) .* sum(A, 1))), 2))
      ↪  .* -30) ...
  + ( ( ( sum(A, 1) * ( (A') * sum(A, 2))) * ( sum(A, 1) * ( (A') * sum(A, 2)))) .* 360) ...
  + ( ( ( sum(A, 1) * (A')) * ( ( A .* ( A .* A)) * (sum(A, 1)'))) .* 480) ...
  + (sum((( sum(A, 2) .* sum(A, 2))  .* ( ( sum(A, 2) .* sum(A, 2)) .* ( sum(A, 2) .* sum(A, 2)))), 1)  .* 16)
      ↪  ...
  + (sum(sum((( A .* repmat(sum(A, 1), [n, 1]))  .* ( ( A .* repmat(sum(A, 1), [n, 1])) .* repmat(sum(( A .* A
      ↪ ), 2), [1, m]))), 2), 1)  .* 360) ...
  + ((sum(( sum(A, 1) .* sum(A, 1)), 2)  .* sum(( ( sum(A, 1) .* sum(A, 1)) .* sum(( A .* A), 1)), 2))  .*
      ↪ -180) ...
  + (sum((repmat(sum(sum(A, 2), 1), [n, 1])  .* sum(( ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])
      ↪ ) .* ( A .* ( A .* A))), 2)), 1)  .* 480) ...
  + (sum((repmat(sum(sum(A, 2), 1), [n, 1])  .* (sum(A, 2)  .* sum(( ( repmat(sum(A, 2), [1, m
      ↪ ]) .* repmat(sum(A, 1), [n, 1])) .* A), 2)))), 1)  .* -240) ...
  + (sum((sum(( A .* repmat(sum(A, 1), [n, 1]))), 2)  .* (sum(A, 2)  .* sum(repmat(sum((repmat(sum(A, 2), [1, m
      ↪ ])  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))), 1), [n, 1]), 2))), 1)  .* 360)
      ↪  ...
  + ( ( ( sum(sum(A, 2), 1) * ( sum(A, 1) * (A'))) * ( ( A * (A')) * sum(A, 2))) .* 720) ...
  + ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1))  .* (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1))  .* ( sum(sum(A
      ↪ , 2), 1) .* sum(sum(A, 2), 1))))  .* 1) ...
  + ((sum(( sum(A, 1) .* sum(A, 1)), 2)  .* (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1))  .* ( sum(sum(A, 2), 1)
      ↪  .* sum(sum(A, 2), 1))))  .* 15) ...
  + (sum((sum(( A .* repmat(sum(A, 1), [n, 1]))), 2)  .* (sum(A, 2)  .* repmat(( sum(sum(A, 2), 1) .* ( sum(sum
      ↪ (A, 2), 1) .* sum(sum(A, 2), 1))), [n, 1]))), 1)  .* 120) ...
  + ((sum(( sum(A, 2) .* sum(A, 2)), 1)  .* (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1))  .* ( sum(sum(A, 2), 1)
      ↪  .* sum(sum(A, 2), 1))))  .* 15) ...
  + ((sum(sum(( A .* A), 2), 1)  .* (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1))  .* ( sum(sum(A, 2), 1) .* sum(
      ↪ sum(A, 2), 1))))  .* 15) ...
  + (sum(sum((repmat(( sum(A, 2) .* sum(A, 2)), [1, m])  .* (( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1),
      ↪ [n, 1]))  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])))), 1), 2)  .* -30) ...
  + (sum((sum((repmat(sum(A, 2), [1, m])  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))), 1)
      ↪ .* sum((repmat(sum(A, 2), [1, m])  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])))),
      ↪ 1)), 2)  .* 45) ...
  + (sum((sum(( A .* repmat(sum(A, 2), [1, m]))), 1)  .* (sum(( A .* repmat(sum(A, 2), [1, m]))), 1)  .* repmat(
      ↪ sum(( sum(A, 1) .* sum(A, 1)), 2), [1, m]))), 2)  .* 180) ...
  + (sum((sum(( A .* repmat(sum(A, 2), [1, m]))), 1)  .* (sum(A, 1)  .* sum(( ( repmat(sum(A, 2), [1, m]) .*
      ↪ repmat(sum(A, 1), [n, 1])) .* A), 1))), 2)  .* -360) ...
  + ((sum(( sum(A, 1) .* sum(A, 1)), 2)  .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(( sum(A, 1) .* sum(A,
      ↪ 1)), 2)))  .* 15) ...
  + (sum((sum((repmat(sum(A, 1), [n, 1])  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))), 1)
      ↪ .* sum((repmat(sum(A, 1), [n, 1])  .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])))),
      ↪ 1)), 2)  .* -30) ...
  + ((sum(sum(A, 2), 1)  .* (sum(sum(A, 2), 1)  .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(( sum(A, 1) .*
      ↪ sum(A, 1)), 2))))  .* 45) ...
  + ((( sum(A, 1) .* sum(A, 1))  * ( ( repmat(sum(A, 1), [m, 1]) * (A')) * ( A * (sum(A, 1)')))  .* 180) ...
  + (sum((sum(( A .* repmat(sum(A, 1), [n, 1]))), 2)  .* (( sum(A, 2) .* repmat(sum(sum(A, 2), 1), [n, 1]))  .*
      ↪  sum(repmat(( sum(A, 1) .* sum(A, 1)), [n, 1]), 2))), 1)  .* 360) ...
  + ( ( sum(( A .* A), 1) * ( ( (A') * sum(A, 2)) .* ( (A') * sum(A, 2)))) .* -360) ...
```

```
+ (sum(( sum(( A .* A), 1) .* ( ( sum(A, 1) .* sum(A, 1)) .* ( sum(A, 1) .* sum(A, 1)))), 2) .* 240) ...
+ ((sum(sum(( A .* A), 2), 1) .* sum(( ( sum(A, 1) .* sum(A, 1)) .* ( sum(A, 1) .* sum(A, 1))), 2)) .*
    ↪ -30) ...
+ ((sum(( sum(A, 1) .* sum(A, 1)), 2) .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(sum(( A .* A), 2), 1)))
    ↪ .* 45) ...
+ ((sum(sum(A, 2), 1) .* ( sum(sum(A, 2), 1) .* sum(( ( sum(A, 1) .* sum(A, 1)) .* sum(( A .* A), 1)), 2)))
    ↪ .* -180) ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(sum(( A .* A),
    ↪ 2), 1))) .* 90) ...
+ (sum(sum((( repmat(sum(( A .* A), 2), [1, m]) .* repmat(sum(( A .* A), 1), [n, 1])) .* ( A .* A)), 2), 1)
    ↪ .* 360) ...
+ (sum((sum(( A .* A), 1) .* ( sum(( A .* A), 1) .* repmat(sum(( sum(A, 1) .* sum(A, 1)), 2), [1, m]))), 2)
    ↪ .* -90) ...
+ ((sum(sum(( A .* A), 2), 1) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* ( sum(A, 2) .* sum(A, 2))), 1)) .*
    ↪ -30) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* ( sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(sum(( A .* A), 2), 1)))
    ↪ .* 45) ...
+ (sum((sum(( A .* repmat(sum(A, 2), [1, m])), 1) .* (sum(( A .* repmat(sum(A, 2), [1, m])), 1) .* repmat(
    ↪ sum(sum(( A .* A), 2), 1), [1, m]))), 2) .* 180) ...
+ (sum((sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* (sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* repmat(
    ↪ sum(sum(( A .* A), 2), 1), [n, 1]))), 1) .* 180) ...
+ (sum((( sum(A, 2) .* repmat(sum(sum(A, 2), 1), [n, 1])) .* (sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .*
    ↪ repmat(sum(sum(( A .* A), 2), 1), [n, 1]))), 1) .* 360) ...
+ (sum((sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* sum(( repmat(( sum(A, 1) .* sum(A, 1)), [n, 1]) .* ( A
    ↪ .* repmat(sum(A, 1), [n, 1]))), 2)), 1) .* -240) ...
+ (sum(sum((A .* repmat((repmat(sum(sum(A, 2), 1), [1, m]) .* (sum(A, 1) .* sum(( ( repmat(sum(A, 2), [1,
    ↪ m]) .* repmat(sum(A, 1), [n, 1])) .* A), 1))), [n, 1])), 2), 1) .* -240) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(( ( sum(A, 1) .* sum(A, 1)) .* ( sum(A, 1) .* sum(A, 1))), 2))
    ↪ .* -30) ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* sum(( A .* A), 2)), 1))
    ↪ .* -180) ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* sum(sum(( ( A .* A) .* ( A .* A)), 2), 1)) .* 60) ...
+ (sum((sum(( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))), 2)
    ↪ .* sum((repmat(sum(A, 2), [1, m]) .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))),
    ↪ 2)), 1) .* -30) ...
+ ((sum(sum(A, 2), 1) .* sum((sum(repmat(( sum(A, 2) .* sum(A, 2)), [1, m]), 1) .* sum((repmat(sum(A, 2),
    ↪ [1, m]) .* ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1]))), 1)), 2)) .* 45) ...
+ (sum((sum(( A .* repmat(sum(A, 2), [1, m])), 1) .* (sum(( A .* repmat(sum(A, 2), [1, m])), 1) .* repmat
    ↪ (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)), [1, m]))), 2) .* 180) ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(( sum(A, 2) .*
    ↪ sum(A, 2)), 1))) .* 90) ...
+ (sum((sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* (sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* repmat
    ↪ (( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)), [n, 1]))), 1) .* 180) ...
+ ( ( ( sum(A, 1) * ( A') * A)) * ( ( (A') * A) * (sum(A, 1)'))) .* 360) ...
+ (sum((sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* (sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* repmat(
    ↪ sum(( sum(A, 2) .* sum(A, 2)), 1), [n, 1]))), 1) .* 180) ...
+ ((sum(( sum(A, 1) .* sum(A, 1)), 2) .* ( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(( sum(A, 2) .* sum(A,
    ↪ 2)), 1))) .* 45) ...
+ ( ( ( (sum(A, 2)') * A) * ( ( (A') * A) * ( (A') * sum(A, 2)))) .* 360) ...
+ (sum((sum(( A .* repmat(sum(A, 2), [1, m])), 1) .* sum(( repmat(( sum(A, 2) .* sum(A, 2)), [1, m]) .* ( A
    ↪ .* repmat(sum(A, 2), [1, m]))), 1)), 2) .* -240) ...
+ ((( (sum(A, 2)') * A) * ( (A') * ( repmat(sum(A, 2), [1, n]) * ( sum(A, 2) .* sum(A, 2))))) .* 180) ...
+ ( ( ( (sum(A, 2)') * A) * ( (( A .* ( A .* A))') * sum(A, 2))) .* 480) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* ( sum(A, 2) .* sum(A, 2))), 1))
    ↪ .* -30) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* ( sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(( sum(A, 2) .* sum(A,
    ↪ 2)), 1))) .* 15) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* sum(( A .* A), 2)), 1)) .*
    ↪ -180) ...
+ (sum((( sum(( A .* A), 2) .* sum(( A .* A), 2)) .* repmat(sum(( sum(A, 2) .* sum(A, 2)), 1), [n, 1])), 1)
    ↪ .* -90) ...
+ ((sum(( sum(A, 1) .* sum(A, 1)), 2) .* ( sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(sum(( A .* A), 2), 1)))
    ↪ .* 90) ...
+ ( sum(( ( sum(A, 1) .* sum(A, 1)) .* sum(( ( A .* A) .* ( A .* A)), 1)), 2) .* -480) ...
+ (( sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(sum(( ( A .* A) .* ( A .* A)), 2), 1)) .* 60) ...
+ ( sum(( sum(( A .* A), 1) .* sum(( ( A .* A) .* ( A .* A)), 1)), 2) .* -480) ...
+ (sum((sum(repmat(( sum(A, 2) .* sum(A, 2)), [1, m]), 1) .* sum(( ( A .* A) .* ( A .* A)), 1)), 2) .* 60)
    ↪ ...
+ ( ( ( sum(A, 1) .* sum(( A .* A), 1)) * ( ( (A') * A) * (sum(A, 1)'))) .* -720) ...
+ ((sum(sum(A, 2), 1) .* sum(sum((( ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])) .* A) .*
    ↪ repmat(sum(( A .* A), 1), [n, 1])), 2), 1)) .* -720) ...
+ ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(( ( sum(A, 1) .* sum(A, 1)) .* sum(( A .* A), 1)), 2)) .*
    ↪ -180) ...
+ ((sum(sum(( A .* A), 2), 1) .* sum(( ( sum(A, 1) .* sum(A, 1)) .* sum(( A .* A), 1)), 2)) .* -180) ...
+ (sum(( ( sum(A, 1) .* sum(A, 1)) .* ( sum(( A .* A), 1) .* sum(( A .* A), 1))), 2) .* 720) ...
+ ( sum(( sum(( A .* A), 1) .* ( sum(( A .* A), 1) .* sum(( A .* A), 1))), 2) .* 240) ...
+ (sum(sum((repmat(sum(( A .* A), 1), [n, 1]) .* (repmat(sum(sum(A, 2), 1), [n, 1])) .*
    ↪ [1, m]) .* repmat(sum(( A .* A), 1), [n, 1]))), 2), 1) .* -90) ...
+ (sum((sum(repmat(( sum(A, 2) .* sum(A, 2)), [1, m]), 1) .* ( sum(( A .* A), 1) .* sum(( A .* A), 1))), 2)
    ↪ .* -90) ...
+ (sum((sum(( A .* A), 2) .* ( sum(( A .* A), 2) .* repmat(( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)), [n,
    ↪ 1]))), 1) .* -90) ...
+ ((sum(( sum(A, 1) .* sum(A, 1)), 2) .* ( sum(sum(( A .* A), 2), 1) .* sum(sum(( A .* A), 2), 1))) .* 45)
    ↪ ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* ( sum(sum(( A .* A), 2), 1) .* sum(sum(( A .* A), 2), 1)))
    ↪ .* 45) ...
+ ((( sum(sum(A, 2), 1) .* sum(sum(A, 2), 1)) .* ( sum(( sum(A, 2) .* sum(A, 2)), 1) .* sum(sum(( A .* A),
    ↪ 2), 1))) .* 90) ...
+ (sum(( sum(( A .* A), 2) .* ( ( sum(A, 2) .* sum(A, 2)) .* ( sum(A, 2) .* sum(A, 2)))), 1) .* 240) ...
+ ( sum(( ( sum(A, 2) .* sum(A, 2)) .* sum(( ( A .* A) .* ( A .* A)), 2)), 1) .* -480) ...
+ (sum(( ( sum(A, 2) .* sum(A, 2)) .* ( sum(( A .* A), 2) .* sum(( A .* A), 2))), 1) .* 720) ...
```

```
  + ((sum(sum(A, 2), 1) .* sum((sum((( A .* A), 2) .* sum(( ( repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1),
      ↪ [n, 1])) .* A), 2)), 1)) .* -720) ...
  + ( ( ( (sum(A, 2)') * ( A * (A'))) * ( sum(A, 2) .* sum((( A .* A), 2))) .* -720) ...
  + ( sum(( sum((( A .* A), 2) .* sum(( ( A .* A) .* ( A .* A)), 2)), 1) .* -480) ...
  + ( sum(( sum((( A .* A), 2) .* ( sum(( A .* A), 2) .* sum(( A .* A), 2))), 1) .* 240) ...
  + ((sum(( sum(A, 1) .* sum(A, 1)), 2) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* sum(( A .* A), 2)), 1)) .*
      ↪ -180) ...
  + ((sum(sum(( A .* A), 2), 1) .* sum(( ( sum(A, 2) .* sum(A, 2)) .* sum(( A .* A), 2)), 1)) .* -180) ...
  + (sum((sum(( A .* repmat(sum(A, 1), [n, 1])), 2) .* (sum(A, 2) .* sum(( ( repmat(sum(A, 2), [1, m]) .*
      ↪ repmat(sum(A, 1), [n, 1])) .* A), 2))), 1) .* -360) ...
  + ( ( ( ( sum(A, 1) * (A')) .* ( sum(A, 1) * (A'))) * sum(( A .* A), 2)) .* -360) ...
  + (sum((sum(repmat(( sum(A, 1) .* sum(A, 1)), [n, 1]), 2) .* ( sum(( A .* A), 2) .* sum(( A .* A), 2))), 1)
      ↪ .* -90) ...
  + ((sum(( sum(A, 2) .* sum(A, 2)), 1) .* ( sum(sum(( A .* A), 2), 1) .* sum(sum(( A .* A), 2), 1))) .* 45)
      ↪ ...
  + ( sum(sum(( ( A .* A) .* ( ( A .* A) .* ( A .* A))), 2), 1) .* 256) ...
  + ( ( sum(sum(( A .* A), 2), 1) .* sum(sum(( ( A .* A) .* ( A .* A)), 2), 1)) .* 60) ...
  + ((sum(sum(( A .* A), 2), 1) .* ( sum(sum(( A .* A), 2), 1) .* sum(sum(( A .* A), 2), 1))) .* 15) ...
  + (sum(sum((repmat(sum(( A .* A), 1), [n, 1]) .* ( repmat(sum(( A .* A), 2), [1, m]) .* repmat(sum(( A .* A
      ↪ ), 1), [n, 1]))), 1), 2) .* -90) ...
  + ((repmat(sum(sum(A, 2), 1), [1, m]) * ( ( ( (A') * A) .* ( (A') * A)) * repmat(sum(sum(A, 2), 1), [m, 1])
      ↪ )) .* 90) ...
  + (sum(sum((repmat(sum(( A .* A), 2), [1, m]) .* ( repmat(sum(( A .* A), 2), [1, m]) .* repmat(sum(( A .* A
      ↪ ), 1), [n, 1]))), 2), 1) .* -90) ...
  + ( sum(( ( ( A * (A')) .* ( A * (A'))) * ( sum(A, 2) .* sum(A, 2))), 1) .* -360) ...
  + ( sum(( ( ( A * (A')) .* ( A * (A'))) * sum(( A .* A), 2)), 1) .* -360) ...
  + (sum(( ( ( A * (A')) .* ( A * (A'))) * sum(repmat(( sum(A, 1) .* sum(A, 1)), [n, 1]), 2)), 1) .* 90) ...
  + (sum(( ( ( A * (A')) .* ( A * (A'))) * repmat(sum(( sum(A, 2) .* sum(A, 2)), 1), [n, 1])), 1) .* 90) ...
  + ( sum(( ( sum(A, 1) .* sum(A, 1)) * ( ( (A') * A) .* ( (A') * A))), 2) .* -360) ...
  + ( sum(( sum(( A .* A), 1) * ( ( (A') * A) .* ( (A') * A))), 2) .* -360) ...
  + (( repmat(sum(sum(( A .* A), 2), 1), [1, m]) * sum(( ( (A') * A) .* ( (A') * A)), 2)) .* 90) ...
  + ( sum(sum(( A .* ( ( A * (A')) * ( A .* ( A .* A)))), 2), 1) .* 480) + ( sum(sum(( ( ( A * (A')) * A) .*
      ↪ ( ( A * (A')) * A)), 2), 1) .* 120))) / 4096;
normalization = sum(abs(original(:)));
assert(sum(abs(original(:) - optimized(:))) / normalization < 1e-10);
```

# Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks

Krzysztof Pawłowski* and Karol Kurach*

Faculty of Mathematics, Informatics and Mechanics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
{kpawlowski236,kkurach}@gmail.com

**Abstract.** Hazard monitoring systems play a key role in ensuring people's safety. The problem of detecting dangerous levels of methane concentration in a coal mine was a subject of IJCRS'15 Data Challenge competition. The challenge was to predict, from multivariate time series data collected by sensors, if methane concentration reaches a dangerous level in the near future. In this paper we present our solution to this problem based on the ensemble of Deep Neural Networks. In particular, we focus on Recurrent Neural Networks with Long Short-Term Memory (LSTM) cells.
**Keywords:** Machine Learning, Recurrent Neural Networks, Ensemble Methods, Time Series Forecasting, Hazard Monitoring Systems

## 1 Introduction

Working in a coal mine historically has been a very hazardous occupation. Over time the conditions and safety improved substantially, in part thanks to advances in technology. Despite remarkable progress, it is still one of the most dangerous professions[6]. One of the dangers present is a possibility of explosion caused by high concentration of methane in the air. Therefore, it is of utmost importance to monitor methane concentration levels and to ensure they are within a safe range. If the concentration levels reach a critical threshold, the production line needs to be shut down[23], which is a costly interruption.

On the other hand, mining effectiveness positively depends on the pace of methane emissions. Therefore, in order to maximize the efficiency, a fine balance needs to be made between mining effectiveness and the safety. If one could predict methane concentration in the future and tell when it is likely to be dangerously high, one could reduce the speed of the operation and try to avoid reaching dangerous levels. Design of such an efficient prediction algorithm is a goal of IJCRS'15 Data Challenge: Mining Data from Coal Mines competition[16]. The problem is an example of supervised learning classification task, with data given in a form of non-stationary multivariate time series. Between the training and test data sets there is a significant concept drift.

Different methods of tackling similar problems have been proposed in the literature. One of them is an application of Deep Neural Networks. While artificial neural

* Both authors contributed equally.

networks have been known for a very long time, in recent years they have achieved spectacular results in areas such as computer vision[17][18] or speech recognition[10][14], in part due to advances in computing hardware, such as Graphics Processing Units. Recurrent Neural Network is an architecture well-suited for the processing of time series data[4][8]. We aim to test how well such methods perform in the competition. In this paper we present our solution which uses Recurrent Neural Network with Long Short-Term Memory cells[15][27], Deep Feedforward Neural Network and ensembling[5] techniques.

The rest of this paper is organized as follows. In Section 2 we present the problem in detail. Section 3 describes the Recurrent Neural Network model we used. In Section 4 we present ensembling technique, detail Deep Feedforward Neural Network model and analyse the results. Finally, in Section 5 we conclude the paper and propose future work.

## 2 Problem Statement

In this section we describe the data used in the competition. Then, we document the evaluation procedure, including the target measure to be optimized. Finally, we review the most important challenges.

### 2.1 Data

The goal of *IJCRS'15 Data Challenge* competition is to predict dangerous level of methane concentration in coal mines based on the readings of 28 sensors. It is an example of supervised learning classification task. The data is split into training and test set, where the training set contains 51700 records and test set contains 5076 records.

Each record is a collection of 28 time series – corresponding to 28 sensors that are installed in the coal mine. The sensors record data such as level of methane concentration, temperature, pressure, electric current of the equipment etc. Each of the time series contains 600 readings, taken every second, for a total of 10 minutes of the same time period for each sensor. The time periods described in the training data set overlap and are given in a chronological order. For the test data, however, the time periods do **not** overlap and are given in random order.

For each record in the training set, three labels are given. The test set is missing the labels – it is the goal of the competition to predict those values. Each label instance can be either *normal* or *warning*. Those levels signify the amount of methane concentration, as recorded by the three known sensors, named *MM263*, *MM264* and *MM256*. The second-by-second readings of those sensors are described in time series mentioned in the previous paragraph. The predictions are to be made about the methane level in the future - that is during the period between three and six minutes after the end of the training (time series) period. If the level of methane concentration reaches or exceeds 1.0, then the corresponding label should be *warning*. Otherwise, it should be *normal*.

### 2.2 Evaluation

The submissions consist of three predictions of label values, made for each of 5076 records in the test set. Each prediction is a number – a higher value denotes a higher likelihood that the true label value is *warning*. The score is defined as a mean of *area under the ROC curve*, averaged over the three labels.

Participants may submit their predictions during the course of the competition. Until the finish of the competition, the participants are aware only of the score computed over *preliminary test set* – a subset of the whole test set that contains approximately $20\%$ of the records. This subset is picked at random by the organizers and is fixed for all competitors but it is not revealed to the participants which of the test records belong to it. The participants may choose a single final solution, possibly taking into the account the scores obtained on the preliminary test set. However, the final score is computed over the *final test set* – remaining approximately $80\%$ of the test data. This score is revealed only after the end of the competition and is used to calculate the final standings – the team with the highest score is declared the winner.

### 2.3 Challenges

The problem presents the following challenges.

**Imbalanced Data.** Only about $2\%$ of the labels in the training set belong to the *warning* class, while the remaining belong to the *normal* class. A trivial solution that predicts *normal* for every label achieves $98\%$ accuracy, obviously without having any practical significance (and with a bad $0.5$ mean-ROC score). Unless special precautions are taken, methods that heavily optimize just the prediction accuracy can have significant problems with this task.

**Overlapping Training Periods.** Almost all adjacent training records overlap by $9$ out of the total $10$ minutes recorded in the time series. It clearly violates the assumption of i.i.d. that underpins the theoretical justification of many learning algorithms. In addition, due to overlap, a classical cross-validation approach may result in splits very „similar" data across different folds and in turn yield over-optimistic estimates of the model performances.

**Noisiness.** Seemingly small „meaningful" changes in sensor readings happen at the $1$-second resolution. Most changes at this interval appear to be random fluctuations. That, combined with a large amount ($16800$) of readings per record, poses a severe danger of overfitting.

**Large Data Size.** The whole training set consists of over $868,560,000$ values. Just storing it in a computer memory requires $3.5$ gigabytes of memory, when using $32$-bit floating point representation. Thus storage and computational costs can be a significant constraint.

**Concept Drift.** Training and test data come from different time periods. The records in the training set are sorted by time, so it's easy to notice that there are very significant trends in the data that change along with the time. With test data samples taken at times belonging to a different interval than training samples, one can expect a severe concept drift - and indeed exploratory tests showed that classifier performance degrades on the test set, as compared to the same classifier's performance when it is evaluated on the interval of training data that was not used for its learning.

# 3 Long Short-Term Memory Model

This section describes the Recurrent Neural Network with Long Short-Term Memory (LSTM)[15] model, which is a crucial part of our final solution.

First, we give some background about the recurrent networks and formally define the dynamics of LSTM. Next, we describe how the data was preprocessed. Finally, we present the network architecture and describe in more detail the training procedure.

## 3.1 Overview

Recurrent Neural Network (RNN) is a type of artificial neural network in which dependencies between nodes form a directed cycle. This allows the network to preserve a state between subsequent time steps. This kind of network is particularly suited for modeling sequential data, where the length of the input is not fixed or can be very long. Parameters in RNNs are shared between different parts of the model, which allows better generalization.

LSTM is an RNN architecture designed to be better at storing and accessing information than standard RNN [11]. LSTM block contains memory cells that can remember a value for an arbitrary length of time and use it when needed. It also has a special *forget gate* that can erase the content of the memory when it is no longer useful. All the described components are built from differentiable functions and trained during back-propagation step.

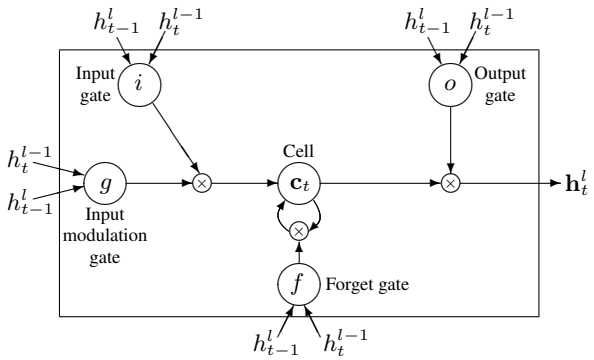



**Fig. 1.** A graphical representation of LSTM memory cells used in [27] and in our solution.

The LSTM networks recently achieved state of the art performance in many tasks, including language modeling[21], handwriting[12] or speech[10] recognition, and machine translation[22]. There are several variants of LSTM that slightly differ in connectivity structure and activation functions. Definition 1 describes the architecture that we implemented based on the equations from [27].

**Definition 1.** *Let $h_t^l \in \mathbb{R}^n$ be a hidden state in layer $l$ of the network at step $t$. We assume that $h_t^0$ is the input at time $t$. Similarly, let $c_t^l \in \mathbb{R}^n$ be a vector of long term memory cells in layer $l$ at step $t$. We define $T_{n,m} : \mathbb{R}^n \to \mathbb{R}^m$ to be an affine transform ($x \to Wx + b$ for some $W$ and $b$) and $\odot$ be a element-wise multiplication. Then, LSTM is a transformation that takes 3 inputs ($h_{t-1}^l$, $h_t^{l-1}$, $c_{t-1}^l$) and computes 2 outputs ($h_t^l$ and $c_t^l$) as follows:*

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \mathrm{sigm} \\ \mathrm{sigm} \\ \mathrm{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

In these equations, i, f, o, g are *input*, *forget*, *output* and *input modulation* gates respectively. The $\mathrm{sigm}$ and $\tanh$ functions are applied element-wise. This relationship is presented in Figure 1.

### 3.2 Data Preprocessing

To address problems mentioned in Section 2.3, we apply several transformations to the original data.

First, for every sensor we normalize all of its readings to mean $0$ and variance $1$. This is a standard technique that improves convergence of gradient descent methods. To reduce overfitting and prevent algorithm from getting stuck in local minima, we shuffle training data after every epoch (one full pass over the data). We also address the problem of unbalanced classes by up-sampling with repetition from the *warning* class. It was done to ensure that at least $10\%$ of training examples are from the *warning* class.

Finally, we address the problem of large data. We modify the $600$ input values from every sensor by grouping every $10$ consecutive values and replacing them with their average.

### 3.3 Network Architecture and Training

As described in section 3.2, we did not use the input sequence of $600$ values directly but instead we grouped them. The network was unfolded to $60$ time steps and trained using backpropagation through time[25]. This was mostly because of the practical purposes: for the original size the training took $10$ times longer, we often exceeded the GPU memory and it was preventing us from testing bigger embedding or batch sizes. It could also negatively affect backpropagation, as the gradient was propagated for much longer. We tried other window sizes, but the value of $10$ seemed to achieve a good trade-off between learning speed and quality.

The sensor values go through the hidden layer, which in this case consist solely of LSTM cells. At time step $t \in 1, .., 60$, the input for RNN are $28$ average sensor values from seconds $[(t-1) * 10, t * 10)$.

After processing the whole sequence, the network's hidden state $h_n^1$ encodes all sensor averages in the same order in which they were seen. On top of this we build a standard supervised classifier (Multi-Layer Perceptron in this case) that predicts the binary outcome. The *warning* class is assigned a value of $1.0$ and *normal* class is assigned a value of $0.0$. The loss function used in the final model was Mean Squared Error. It performed better than Binary Cross Entropy loss which is typically used for binary classification

The training is done using standard Stochastic Gradient Descent algorithm and backpropagation through time. We initialize all the parameters by sampling from uniform distribution. To avoid *exploding gradient* problem, the gradients are scaled globally during training, so that their norm is not grater than $1\%$ of parameters' norm. All models were trained using Torch[3] on a machine with a GPU card.

## 4 Final Ensemble Model

We improve the quality of our prediction by making an ensemble model. In this section we give an overview of the ensemble technique, describe the extra base learner that was used in conjunction with the method described in Section 3 and document the procedure we used to obtain the final prediction.

### 4.1 Ensembling Methods

Ensemble methods combine results of multiple „base" learning algorithms, to form a single prediction that often achieves a better performance than any of individual base algorithm alone[5]. To give an example, one of the simplest forms of ensembling is to average the predictions of base algorithms. Ensembling works the best when base algorithms are accurate and, importantly, diverse. Diversity can mean that errors produced by the base algorithms are slightly correlated. Intuitively, in the mentioned example of ensembling by averaging, the prediction quality increases because uncorrelated errors „average out". The more sophisticated ensembling algorithms include bagging[2], boosting[7] or stacking[26].

### 4.2 Deep Feedforward Neural Network as a Base Learner

*Deep feedforward neural network* (DFNN) is an *artificial neural network* with multiple layers of hidden neurons. One notable difference between DFNN and LSTM network (described in Section 3) is that DFNN architecture does not contain recursive connections – instead, every neuron of the previous layer is connected with every neuron of the next layer. We use DFNN, together with LSTM, as base learners in an ensemble model. We train the DFNN with a *backpropagation* algorithm[24] that uses *stochastic gradient descent* (with *mini-batch* and *momentum*) as an optimization procedure to minimize the *root mean squared error* between numeric predictions and the target values. To avoid overfitting to the training set we use two regularization[9] methods: ad-hoc *early stopping*[19] and *dropout*[20].

**Feature Engineering** To balance the impact of different features, further reduce overfitting and decrease the computational cost we preprocess the training and test data in the following way:

1. we scale the readings (separately for each sensor) across all the records to have *mean* equal $0$ and *standard deviation* equal $1$,
2. we transform the values with $x \to log(1 + x)$ function,
3. we compute *mean* and *standard deviation* for every sensor, taken over the last $30$ readings ($30$-second period),
4. we keep the last $20$ readings for the sensor that corresponds to the target label,
5. we discard all the original features.

Such preprocessing reduces the number of features from $16800$ $(28 * 600)$ to just $76$ $(28 * 2 + 20)$.

We also preprocess the training target values in the way described below. As mentioned in Subsection 2.1, there are three labels for each training record with the *normal* class for future methane concentration levels under $1.0$ and the *warning* class for methane concentration levels at or above $1.0$. Recall from Subsection 2.3, that an overwhelming majority of adjacent training periods overlap by exactly $9$ minutes. That two facts together allow us to reconstruct the exact amount of methane concentration for a given record – to that end, we simply peek sensors values a few records in advance. We take reconstructed values as targets instead of the original label values, while also discarding all the training records for which such reconstruction is not possible (it turns out that less than $1000$ records out of $51700$ training records are discarded).

**Training and Parameter Tuning** For each target label we train a different DFNN model and tune its parameters independently, to optimize the performance on:

- (initially) the validation set – created by us as $20\%$ of original training data. We train the model on the remainder of training data,
- (finally) the preliminary test set. See Subsection 4.4 for more discussion of model selection challenges.

We describe the final values of the most important parameters in Table 1. Parameters for the first target label (*MM263*) are omitted because for that label, the DFNN model fails to generate quality predictions for all the parameter combinations we tried.

### 4.3 Ensembling in Our Solution

Our final solution is an ensemble of two base models - LSTM described in Section 3 and DFNN described in Subsection 4.2.

We decide to forgo the complex ensembling schemes that would require retraining of the models and perhaps additional parameter tuning. Instead, we consider a few simple averaging methods and finally we choose the method that gives the best AUC score – that is averaging the ranks of the base models' predictions. Table 2 illustrates the scores of particular models that are achieved on the preliminary test set. As the final submission we take the model that is a combination of the best-performing methods for each target label value. That is, for label *MM263* we use LSTM, for label *MM264* we use DFNN and for label *MM256* we use the ensemble of LSTM and DFNN.

| target label | MM263 | MM264 | MM256 |
|---|---|---|---|
| activation | - | sigmoid | ReLU |
| layer sizes | - | 76-15-5-2-1 | 76-25-7-3-1 |
| dropout | - | none | .5 |
| learning rate | - | 0.1 | 0.1 |
| mini-batch size | - | 30 | 30 |
| number of epochs | - | 550 | 233 |

**Table 1.** Tuned parameter values for DFNN

| AUC score \ label | MM263 | MM264 | MM256 |
|---|---|---|---|
| LSTM | **0.9599** | 0.9560 | 0.9605 |
| DFFN | - | **0.9773** | 0.9602 |
| ensemble | - | 0.9722 | **0.9683** |

**Table 2.** Model scores

### 4.4 Results and Discussion

Our final ensemble model achieved a score of $0.94$ and the 6th place in the competition. It is interesting to mention that on the preliminary test set, our method obtained much higher score of $0.9685$ which, if not decreased, would correspond to the 1st place.

Such a significant drop in score can be explained by overfitting. It was not a surprise, as we deliberately chose to perform model selection on the preliminary test set – that is, we submitted the model that achieved the best score on that set. Usually one would perform a cross-validation on the training set to perform model selection. The reason we decided not to, was because of the significant differences between the training and the test distributions (concept drift), as mentioned in Subsection 2.3. We wanted to avoid a situation when the model is overly tuned to the training test and as a result does not generalize well on the test set. As we had only one shot for a final submission, it is not clear if the traditional (cross-validation) approach would have worked better – it can be an interesting topic for further research.

## 5 Conclusion

In this paper we presented our Deep Neural Network-based solution to *IJCRS'15 Data Challenge: Mining Data from Coal Mines* contest. It achieved a competitive score of $0.94$ and the 6th place. The approach we developed should generalize well to other multivariate time series prediction problems. The obtained results confirm that methods based on Deep Neural Networks are not only effective for processing time series data, but also do not require extensive feature engineering to perform well. As expected, ensembling improves the quality of the prediction.

It would be interesting to see, if more advanced artificial neural network architectures, such as LSTMs with attention mechanism[1] or Neural Turing Machines[13], could achieve even better results. Another topic worth exploring are methods of handling the concept drift in context of parameter tuning and model selection, which was one of the main challenges in this task.

## References


1. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. CoRR abs/1409.0473 (2014), http://arxiv.org/abs/1409.0473
2. Breiman, L.: Bagging predictors. Machine learning 24(2), 123–140 (1996)
3. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop. No. EPFL-CONF-192376 (2011)
4. Connor, J.T., Martin, R.D., Atlas, L.E.: Recurrent neural networks and robust time series prediction. Neural Networks, IEEE Transactions on 5(2), 240–254 (1994)
5. Dietterich, T.G.: Ensemble methods in machine learning. In: Multiple classifier systems, pp. 1–15. Springer (2000)
6. Donoghue, A.: Occupational health hazards in mining: an overview. Occupational Medicine 54(5), 283–289 (2004)
7. Freund, Y., Schapire, R., Abe, N.: A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence 14(771-780), 1612 (1999)
8. Giles, C.L., Lawrence, S., Tsoi, A.C.: Noisy time series prediction using recurrent neural networks and grammatical inference. Machine learning 44(1-2), 161–183 (2001)
9. Girosi, F., Jones, M.B., Poggio, T.: Regularization theory and neural networks architectures. Neural computation 7(2), 219–269 (1995)
10. Graves, A., Mohamed, A.r., Hinton, G.: Speech recognition with deep recurrent neural networks. In: Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. pp. 6645–6649. IEEE (2013)
11. Graves, A.: Generating sequences with recurrent neural networks. CoRR abs/1308.0850 (2013), http://arxiv.org/abs/1308.0850
12. Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., Schmidhuber, J.: A novel connectionist system for unconstrained handwriting recognition. Pattern Analysis and Machine Intelligence, IEEE Transactions on 31(5), 855–868 (2009)
13. Graves, A., Wayne, G., Danihelka, I.: Neural turing machines. CoRR abs/1410.5401 (2014), http://arxiv.org/abs/1410.5401
14. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. Signal Processing Magazine, IEEE 29(6), 82–97 (2012)
15. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation 9(8), 1735–1780 (1997)
16. Janusz, A., Ślęzak, D., Sikora, M., Wróbel, L., Stawicki, S., Grzegorowski, M., Wojtas, P.: Mining data from coal mines: IJCRS'15 Data Challenge. In: Proceedings of IJCRS'15. LNCS, Springer (2015), in print November 2015
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
18. Le, Q.V.: Building high-level features using large scale unsupervised learning. In: Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. pp. 8595–8598. IEEE (2013)
19. Prechelt, L.: Early stopping-but when? In: Neural Networks: Tricks of the trade, pp. 55–69. Springer (1998)
20. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research 15(1), 1929–1958 (2014)

21. Sundermeyer, M., Schlüter, R., Ney, H.: Lstm neural networks for language modeling. In: INTERSPEECH (2012)
22. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. pp. 3104–3112 (2014)
23. Szlązak, N., Obracaj, D., Borowski, M., Swolkień, J., Korzec, M.: Monitoring and controlling methane hazard in excavations in hard coal mines. AGH Journal of Mining and Geoengineering 37 (2013)
24. Werbos, P.: Beyond regression: New tools for prediction and analysis in the behavioral sciences (1974)
25. Werbos, P.J.: Generalization of backpropagation with application to a recurrent gas market model. Neural Networks 1(4), 339–356 (1988)
26. Wolpert, D.H.: Stacked generalization. Neural networks 5(2), 241–259 (1992)
27. Zaremba, W., Sutskever, I., Vinyals, O.: Recurrent neural network regularization. CoRR abs/1409.2329 (2014), http://arxiv.org/abs/1409.2329

# Predicting Dangerous Seismic Activity with Recurrent Neural Networks

Karol Kurach
University of Warsaw
Email: kk236085@mimuw.edu.pl

Krzysztof Pawlowski
University of Warsaw
Email: kpawlowski236@gmail.com

***Abstract*—In this paper we present a solution to the AAIA'16 Data Mining Challenge. The goal of the challenge was to predict, from multivariate time series data, periods of increased seismic activity which may cause life-threatening accidents in underground coal mines. Our solution is based on Recurrent Neural Network with Long Short-Term Memory cells. It requires almost no feature engineering, which makes it easily applicable to other domains with multivariate time series data. The method achieved the 5th place in the AAIA'16 competition, out of 203 teams.**

## I. Introduction

UNDERGROUND coal mine workers are exposed to a life-threatening danger in a form of seismic events. To improve workers' safety, it is crucial to predict those phenomena in advance. However, knowledge-based safety monitoring systems that are currently deployed in coal mines sometimes fail to forecast such occurrences early enough. The goal of the AAIA'16 Data Mining Challenge: Predicting Dangerous Seismic Events in Active Coal Mines competition [12] was to design methods that could improve reliability of seismic activity prediction.

The task is an instance of a classification problem with unbalanced data provided in a form of multivariate, non-stationary time series. We present a solution based on Recurrent Neural Network with Long Short-Term Memory cells. The proposed model is generic and does not rely on the domain knowledge. It requires only minimal feature preprocessing and no feature engineering or feature selection steps. The solution achieved a competitive 5th place in the AAIA'16 competition.

The rest of the paper is organized as follows. In Section II we give an overview of the related work. The details of the AAIA'16 challenge are described in Section III. Section IV gives a brief introduction to Recurrent Neural Networks and Long Short-Term Memory cells. In Section V we describe the details of our architecture, training and model selection. Finally, Section VI summarizes the paper.

## II. Related Work

Seismic hazard and rock bursts pose a threat to miners' lives and overall safety of the coal mining operation. One of the techniques for addressing this problem is to monitor the sensor readings' with automated algorithms. Originally, natural earthquake seismology approaches have been used to deal with the problem[3]. Mine-induced seismicity can be assessed using mechanisms of mine tremors, such as magnitude, moment, stress drop and seismic efficiency[16] or using seismic tomography[10]. More recently, typical machine learning approaches have been used – such as Random Forests[4] and other nonlinear methods[5], including Support Vector Machines or Naive Bayes Classifier.

The recent IJCRS 2015 Data Challenge competition[13] provided an opportunity to compare different approaches on the data set coming from coal mining. Although the goal was a bit different (to predict dangerous levels of methane concentration), the data shared similar characteristics – being an example of a time series, multivariate prediction problem with concept drift. Most of the top solutions relied heavily on feature engineering, either manual or automatic, such as: automatic variable construction[1], window-based feature engineering[8], hand-crafted features[15] or thousands of automatically generated features[21].

## III. Challenge description

### A. Data

The aim of the AAIA'16 competition was to predict relative likelihood of seismic events in coal mines based on the recorded measurements. It is an instance of supervised learning classification task, with most of the data given in a form of non-stationary multivariate time series. The data is split into 5 training sets and a single test set. All the training sets together contain $133,151$ records, while the test set contains $3,860$ records. Each record describes a period of 24 hours and consists of:

- an identifier of the main working site and 12 characteristics related to the whole period of 24 hours, such as total energy of seismic bumps registered in the last 24 hours,
- 22 times series with 24 numeric per-hour aggregated measurements, such as energy of the strongest seismic bump within a given hour.

Thus, in total each record contains 541 values. As mentioned previously, the records are grouped into 5 training sets and a single test set. The subsequent training sets correspond to later periods, adjacent records in them overlap by 1 hour and are given in a chronological order. The test set contains records that come from period later than the last training set, its records are non-overlapping and given in random order.

A label is given for each record in the training set while for the test set such label is missing – it is the goal of the

competition to forecast those values. The label is a categorical variable that can be either *normal* or *warning*. Value *warning* indicates that a total seismic energy measured within $8$-hour period after the time covered by the record exceeded the warning threshold of $50,000$ Joules. For each record in the test set, the numeric predictions are to be made about those (hidden) values.

Additionally, there is an extra „meta-data" set that describes main working sites included in the training and test sets. It contains information such as the height of the main working site or the latest geological assessment. We note that the training and test sets are highly unbalanced, with respect to both the *main working site* attribute (Figure 1) and the labels (Table I).

### B. Evaluation

The competition score is defined as an *area under the ROC curve*. It is calculated based on predictions of label values, made for all $3,860$ test set records. Each prediction is a number, where a higher value denotes that the true label value is more likely to be *warning*.

The contestants submit their predictions during the competition. However, before the competition is concluded, the contestants know only the score computed over *preliminary test set* – a part of the whole test set that contains approximately $25\%$ of the data. This subset is chosen randomly by the organizers and is the same for all the contestants. It is not revealed to the participants which of the test records belong to it. The contestants can select a single final solution, possibly guided by the scores obtained on the preliminary test set. The final score, however, is computed over the *final test set*, which consists of the remaining approximately $75\%$ of the test data. This score is shown only after the end of the contest and is used to compute the final standings – the highest-scoring team is declared the winner.

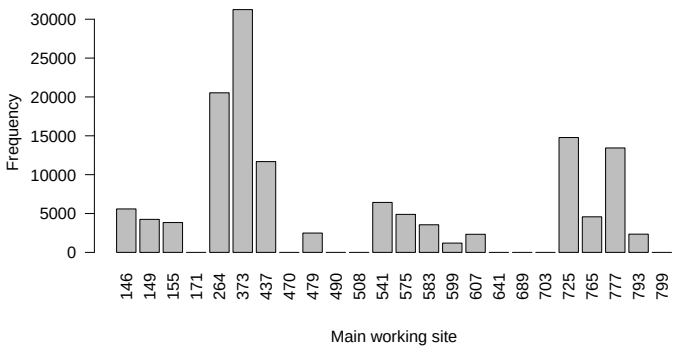



Fig. 1. The frequencies of different main working sites in the training sets. Some sites appear only in the test set.

TABLE I
DISTRIBUTION OF LABELS ACROSS THE TRAINING SETS

| | tr. set 1 | tr. set 2 | tr. set 3 | tr. set 4 | tr. set 5 |
|---|---|---|---|---|---|
| *normal* | 78722 | 13137 | 13047 | 12744 | 12538 |
| *warning* | 1171 | 181 | 269 | 568 | 774 |

## IV. Deep Recurrent Neural Network

### A. Recurrent Neural Networks

Recurrent Neural Network (RNN) is a type of artificial neural network in which dependencies between nodes form a directed cycle. This allows the network to preserve a state between subsequent time steps. We focus on a simple RNN with a single, self-connected hidden layer.

RNNs process all elements from a sequence one-by-one, and the output at every time step depends on all previous inputs. This is a fundamental difference from feedforward networks, where the network's output depend only on the current element. It has an important theoretical implication: RNNs are capable of approximating arbitrary well any measurable sequence-to-sequence mapping[9].

Since RNNs contain loops, the standard backpropagation algorithm does not work. Instead, a *backpropagation through time* algorithm is used[20]. The idea behind this method is to unroll the network over $N$ time steps, and copy the parameters $N$ times. The RNN parameters are shared across all time steps, which makes them trainable and allows generalization.

Since the number of unrolled steps can be arbitrary, RNNs are particularly suited for modeling sequential data, where the length of the input is not fixed or can be very long. Recurrent nets have shown impressive results in many NLP tasks. One particularly successful variant of RNN is a recurrent network with LSTM cells, which we describe below.

### B. Long Short-Term Memory

One important problem with training RNNs is the *vanishing gradient*, which can occur when values smaller than $1.0$ are multiplied at each time step during the backpropagation through time. For some activation functions, the maximal value of the derivative is small. For example, the derivative of commonly used sigmoid function is never bigger than $0.25$. As a result, after $N$ time steps the gradient is multiplied by a value less than or equal to $0.25^N$, which quickly becomes very small as $N$ increases. While using some activation functions (eg. ReLU[17]) can reduce the likelihood of vanishing gradients, there is a special architecture designed to address this problem: Long Short-Term Memory (LSTM).

The LSTM is better at storing and accessing information than standard RNN [11]. The LSTM block consists of a self-connected memory cell and 3 gates named: input, output and forget. The gates control the access to the cell and can be interpreted as "read", "write" and "reset" operations in the standard computer's memory. The network learns to control the gates and decides to update and/or use the value at any given time step. Since all the components are built from

differentiable functions, the gradients can be computed for the whole system and it is possible to train it end-to-end using backpropagation. There are several variants of LSTM that slightly differ in connectivity structure and activation functions. Below we describe the definitions of the input, output and forget gates that we used.

Let $h_t \in \mathbb{R}^n$ be a hidden state, $c_t \in \mathbb{R}^n$ be a vector of memory cells of the network and let $x_t$ be the input at the time step $t$. Let $W_i, W_f, W_u, W_o$ be matrices and $b_i, b_f, b_u, b_o$ the respective bias terms. We define LSTM as a transformation that takes 3 inputs ($h_{t-1}$, $c_{t-1}, x_t$) and produces 2 outputs ($h_t$ and $c_t$). In all equations below $\odot$ is element-wise multiplication. We assume also that $\oplus$ is an operation that aggregates $h_{t-1}$ and $x_t$. We used plain sum, but concatenation of vectors is also commonly used.

The *forget gate* which decides how much of the information should be removed from the cell is defined as:

$$f_t = \text{sigm}(W_f * [h_{t-1} \oplus x_t] + b_f) \tag{1}$$

The *input modulation* gate value $i_t$ and the cell update $u_t$ are defined as:

$$\begin{aligned} i_t &= \text{sigm}(W_i * [h_{t-1} \oplus x_t] + b_i) \\ u_t &= \tanh(W_u * [h_{t-1} \oplus x_t] + b_u) \end{aligned} \tag{2}$$

Intuitively, input modulation decides how much of the $u_t$ should be added to the memory at step $t$. For example, if $x_t$ can be ignored, $i_t$ will be close to 0. Knowing the values above, the new cell value $c_t$ is computed as:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{3}$$

The last step is to compute $h_t$, the output passed to the next LSTM's time step. It is controlled by the *output gate* $o_t$:

$$\begin{aligned} o_t &= \text{sigm}(W_o * [h_{t-1} \oplus x_t] + b_o) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{4}$$

The LSTM networks have been successfully applied to real-world problems, including language modeling[18], handwriting[7] or speech[6] recognition, and machine translation[19].

## V. Model

### A. Preprocessing

Recall from Section II that most of the solutions to the previous challenge depend heavily on feature engineering. Such approach, while effective in practice, makes the model less generalizable as the feature engineering steps depend on the problem at hand. Our goal was to create a model that learns everything from the raw data and does not rely on the domain knowledge. To this end, we limit our preprocessing only to the following two operations:

- **Data normalization**, in regards to *mean* and *standard deviation*. This is a standard Machine Learning procedure, and as such it should be applicable to almost any problem. The normalization makes easier both optimization of the loss function and the regularization, because all feature values are at the same scale.
- **Upsampling positive examples**. As presented in Table I, the ratio of positive to negative examples is highly skewed. To make it more balanced, we sample with repetition from the set of positive examples and add them to the training set. We experimented with different upsampling ratios and achieved best results for increasing the number of positives by $10 - 20$ times.

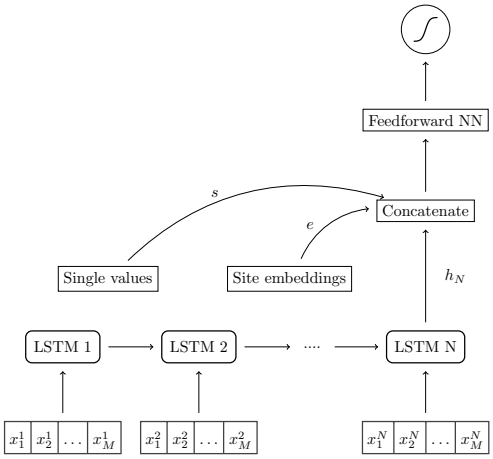



Fig. 2. Overview of the architecture. A core component is a single-layer LSTM unrolled for $N = 24$ time steps that processes $M = 22$ per-hour measurements. The $i$th per-hour measurement is marked as $x^i$. After processing $N$ time steps, the last hidden state $h_N \in \mathbb{R}^{50}$ of the LSTM encodes information about all per-hour measurements. Then, $h_N$ is concatenated with vector $s \in \mathbb{R}^{12}$ of per-record characteristics and the vector $e \in \mathbb{R}^{10}$ representing the working site id embedding.

### B. Architecture

The overview of the architecture is presented in Fig. 2. We use a single-layer LSTM model that is processing 24 hourly aggregated measurements. At every time step, the hidden state of the LSTM ($h_i \in \mathbb{R}^{50}$) is connected to the previous state $h_{i-1}$ and the normalized measurement values from the $i$th hour ($x^i$ in the picture). After processing the whole sequence, the network's final hidden state $h_N$ encodes all measurements in the order in which they appeared.

The vector $h_N$ is concatenated with 2 other vectors: $s$ and $e$. The vector $s$ contains 12 per-record characteristics described in Section III-A. The vector $e$ is an 10-dimensional embedding of the working site id. The values of the embedding vectors are initialized randomly and learned from the training data.

On top of the concatenation layer we build a standard supervised classifier (2-layer feedforward network in this case). We apply sigmoid on the network's output to ensure the predicted value is in the range $[0, 1]$ and can be interpreted as the probability of the *warning* label. The Binary Cross Entropy loss is used as the cost function.

### C. Training

We initialize all model's parameters by sampling uniformly from $[-0.1, 0.1]$. The optimization of the loss function is done

TABLE II
WORKING SITES CHARACTERISTICS

| site id | region name | bed name | assessment | mapped to |
|---|---|---|---|---|
| 146 | Partia F | 416 | a | N/A |
| 149 | Partia F | 418 | b | N/A |
| 155 | Partia H | 502 | b | N/A |
| 171 | Partia F | 409 | a | 146 |
| 264 | Z | 405/2 | b | N/A |
| 373 | G-1 | 707/2 | b | N/A |
| 437 | G-1 | 712/1-2 | b | N/A |
| 470 | Z | 405/2 | c | 264 |
| .. | ..... | .... | ..... | .... |
| 777 | 9 | 504 | b | N/A |
| 793 | 0 | 405 | b | N/A |
| 799 | 9 | 504 | a | 777 |

using Adam algorithm[14] with a learning rate of $0.0005$ and $\varepsilon$ parameter equal to $10^{-10}$. The training is run for $5$ full passes (epochs) over the training data. After each epoch, the learning rate is multiplied by $0.63$ and the training set is randomly shuffled.

We apply standard $l_2$ regularization of the weights with $\lambda = 0.01$. To avoid *exploding gradient* problem, the gradients are clipped globally to the value of $1$. The model was implemented in Torch[2] and trained using a single GPU.

### D. Model selection

Model selection was a significant challenge in the AAIA'16 competition. Recall from Section III-A that the time periods in the training data are overlapping. As a result, the standard cross-validation on a random split of the data tends to be over-optimistic. Also, there is a significant concept drift between the $5$ provided training sets. The $k$-th training set was collected in a time period right after the set $(k-1)$th. We also know that the last training set was collected before the test set.

To address the problem of overlapping periods and to make the local evaluation as close to the final one as we can, we decided to use $5$-fold cross-validation, with one training file being one fold. The average of the AUC scores was the final score we assigned to the model. We completely ignored the leaderboard score, as it proved to be very misleading in the past for this type of data[13].

### E. Dealing with unknown sites

As described in Section V-B, our architecture computes embedding vectors for every working site id. However, recall from Fig. 1 that some of those identifiers exist only in the test data and not in the training data. We used the following method to fix this problem: we looked at the working site metadata and manually mapped $8$ missing ids to existing ids that share similar characteristics. An example is presented in Table II which contains a subset of the metadata file. The id $171$ is mapped to $146$ because the region name and geological assessment is the same for those two sites. Similarly, id $799$ is mapped to $777$ because of the same region and bed names.

The above approach can be significantly improved by computing separate embeddings for different categorical variables from the metadata (region/bed name, etc). This would remove the need of manual mapping of the missing ids and potentially improve the quality as well. However, we did not manage to try this approach during the competition.

### F. Ensembling

From the begin of the competition, our main design decision was to create a competitive solution that consist of only one model trained from the raw data. However, the practice of machine learning competitions shows that ensembling of many different models is an easy way of improving the final score. We decided to do a simple rank average ensembling with a logistic regression model, which moved our solution one place up on the leaderboard.

## VI. CONCLUSION

In this paper we presented a solution to AAIA'16 data mining challenge based on a Recurrent Neural Network with LSTM cells. It achieved a competitive score of $0.934$ and the $5$th place in the competition.

Compared to other methods (see Section II), our solution does not rely heavily on many hand-crafted features. Instead, it learns feature representation from the raw sensor data with a minimal feature engineering. It is a similar method to the one that we used in the previous IJCRS'15 competition, where our model achieved the $6$th place. Top performance in both competitions suggests that our approach is versatile and can be successfully applied to different multivariate time series problems.

## REFERENCES


[1] Marc Boullé. Prediction of methane outbreak in coal mines from historical sensor data under distribution drift. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 439–451. Springer, 2015.

[2] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[3] C Allin Cornell. Engineering seismic risk analysis. *Bulletin of the Seismological Society of America*, 58(5):1583–1606, 1968.

[4] Long-jun Dong, Xi-bing Li, and PENG Kang. Prediction of rockburst classification using random forest. *Transactions of Nonferrous Metals Society of China*, 23(2):472–477, 2013.

[5] Longjun Dong, Xibing Li, and Gongnan Xie. Nonlinear methodologies for identifying seismic event and nuclear explosion using random forest, support vector machine, and naive bayes classification. In *Abstract and Applied Analysis*, volume 2014. Hindawi Publishing Corporation, 2014.

[6] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[7] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(5):855–868, 2009.

[8] Marek Grzegorowski and Sebastian Stawicki. Window-based feature engineering for prediction of methane threats in coal mines. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 452–463. Springer, 2015.

[9] Barbara Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1):107–123, 2000.

[10] David R Hanson, Thomas L Vandergrift, Matthew J DeMarco, and Kanaan Hanna. Advanced techniques in site characterization and mining hazard detection for the underground coal industry. *International journal of coal geology*, 50(1):275–301, 2002.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] Andrzej Janusz, Marek Sikora, Łukasz Wróbel, and Dominik Ślęzak. Predicting Dangerous Seismic Events: AAIA16 Data Mining Challenge. In *Proceedings of FedCSIS 2016*. IEEE, 2016. In print September 2016.

[13] Andrzej Janusz, Marek Sikora, Łukasz Wróbel, Sebastian Stawicki, Marek Grzegorowski, Piotr Wojtas, and Dominik Ślęzak. Mining data from coal mines: Ijcrs2̌01915 data challenge. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 429–438. Springer, 2015.

[14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Petre Lameski, Eftim Zdravevski, Riste Mingov, and Andrea Kulakov. Svm parameter tuning with grid search and its impact on reduction of model over-fitting. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 464–474. Springer, 2015.

[16] A McGarr. Some applications of seismic source mechanism studies to assessing underground hazard. In *Rockbursts and Seismicity in Mines.*, pages 199–208, 1984.

[17] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[18] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *INTERSPEECH*, 2012.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[20] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

[21] Adam Zagorecki. Prediction of methane outbreaks in coal mines from multivariate time series using random forest. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 494–500. Springer, 2015.

# Smart Reply: Automated Response Suggestion for Email

Anjuli Kannan* Karol Kurach* Sujith Ravi*

Tobias Kaufmann* Andrew Tomkins Balint Miklos

Greg Corrado László Lukács Marina Ganea

Peter Young Vivek Ramavajjala

Google
{anjuli, kkurach, sravi, snufkin}@google.com

## ABSTRACT

In this paper we propose and investigate a novel end-to-end method for automatically generating short email responses, called Smart Reply. It generates semantically diverse suggestions that can be used as complete email responses with just one tap on mobile. The system is currently used in *Inbox by Gmail* and is responsible for assisting with 10% of all mobile responses. It is designed to work at very high throughput and process hundreds of millions of messages daily. The system exploits state-of-the-art, large-scale deep learning.

We describe the architecture of the system as well as the challenges that we faced while building it, like response diversity and scalability. We also introduce a new method for semantic clustering of user-generated content that requires only a modest amount of explicitly labeled data.

## Keywords

Email; LSTM; Deep Learning; Clustering; Semantics

## 1. INTRODUCTION

Email is one of the most popular modes of communication on the Web. Despite the recent increase in usage of social networks, email continues to be the primary medium for billions of users across the world to connect and share information [2]. With the rapid increase in email overload, it has become increasingly challenging for users to process and respond to incoming messages. It can be especially time-consuming to type email replies on a mobile device.

An initial study covering several million email-reply pairs showed that ∼25% of replies have 20 tokens or less. Thus we raised the following question: can we assist users with composing these short messages? More specifically, would it be possible to suggest brief responses when appropriate, just one tap away?

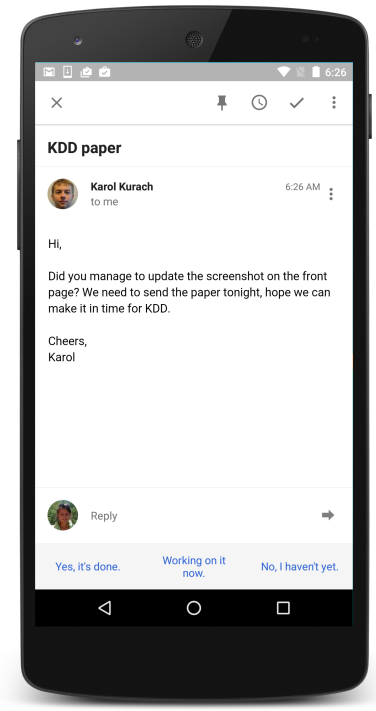



**Figure 1: Example Smart Reply suggestions.**

*Equal contribution.



*KDD '16, August 13–17, 2016, San Francisco, CA, USA*



ACM ISBN 978-1-4503-4232-2/16/08..

DOI: http://dx.doi.org/XXXX.XXXX

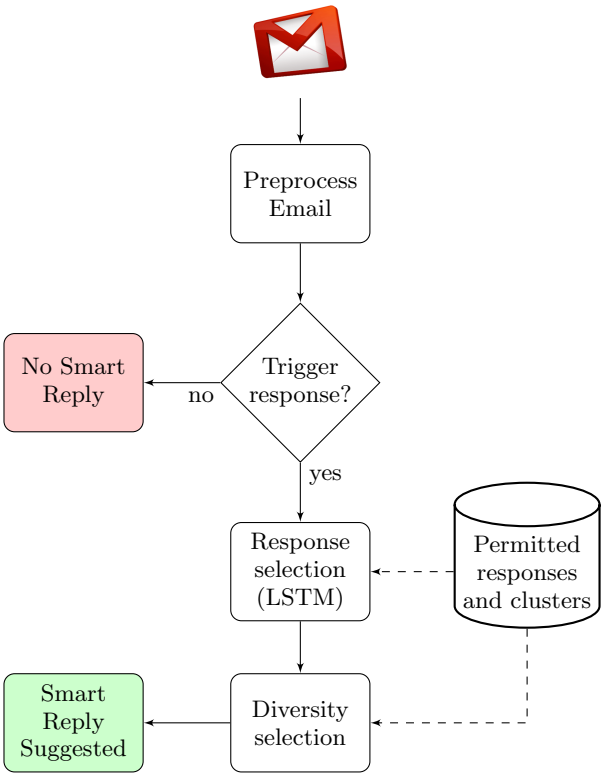



**Figure 2: Life of a message. The figure presents the overview of inference.**

To address this problem, we leverage the sequence-to-sequence learning framework [23], which uses long short-term memory networks (LSTMs) [10] to predict sequences of text. Consistent with the approach of the Neural Conversation Model [24], our *input* sequence is an incoming message and our *output* distribution is over the space of possible replies. Training this framework on a large corpus of conversation data produces a fully generative model that can produce a response to any sequence of input text. As [24] demonstrated on a tech support chat corpus, this distribution can be used to decode coherent, plausible responses.

However, in order to deploy such a model into a product used globally by millions, we faced several challenges not considered in previous work:

- **Response quality** How to ensure that the individual response options are *always* high quality in language and content.
- **Utility** How to select multiple options to show a user so as to maximize the likelihood that one is chosen.
- **Scalability** How to efficiently process millions of messages per day while remaining within the latency requirements of an email delivery system.
- **Privacy** How to develop this system without ever inspecting the data except aggregate statistics.

To tackle these challenges, we propose Smart Reply (Figure 1), a novel method and system for automated email response suggestion. Smart Reply consists of the following components, which are also shown in Figure 2:

1. **Response selection:** At the core of our system, an LSTM neural network processes an incoming message, then uses it to predict the most likely responses. LSTM computation can be expensive, so we improve *scalability* by finding only the approximate best responses. We explain the model in detail in Section 3.
2. **Response set generation:** To deliver high *response quality*, we only select responses from response space which is generated offline using a semi-supervised graph learning approach. This is discussed in Section 4.
3. **Diversity:** After finding a set of most likely responses from the LSTM, we would like to choose a small set to show to the user that maximize the total *utility*. We found that enforcing diverse semantic intents is critical to making the suggestions useful. Our method for this is described further in Section 5.
4. **Triggering model:** A feedforward neural network decides whether or not to suggest responses. This further improves *utility* by not showing suggestions when they are unlikely to be used. We break this out into a separate component so that we have the option to use a computationally cheaper architecture than what is used for the scoring model; this keeps the system *scalable*. This model is described in Section 6.

The combination of these components is a novel end-to-end method for generating short, complete responses to emails, going beyond previous works. For response selection it exploits state-of-the-art deep learning models trained on billions of words, and for response set generation it introduces a new semi-supervised method for semantic understanding of user-generated content.

Moreover, since it directly addresses all four challenges mentioned above, it has been successfully deployed in *I*nbox. Currently, the Smart Reply system is responsible for assisting with 10% of email replies for *I*nbox on mobile.

Next, we discuss the related work in Section 2, followed by a description of our core system components in Sections 3, 4, 5, and 6. We close by showing modeling results in Section 7 and our conclusions in Section 8.

## 2. RELATED WORK

As we consider related work, we note that building an automated system to suggest email responses is not a task for which there is existing literature or benchmarks, nor is this a standard machine learning problem to which existing algorithms can be readily applied. However, there is work related to two of our core components which we will review here: predicting responses and identifying a target response space.

**Predicting full responses.** Much work exists on analyzing natural language dialogues in public domains such as Twitter, but it has largely focused on social media tasks like predicting whether or not a response is made [3], predicting next word only [14], or curating threads [4].

Full response prediction was initially attempted in [16], which approached the problem from the perspective of machine translation: given a Twitter post, "translate" it into a response using phrase-based statistical machine translation (SMT). Our approach is similar, but rather than using

SMT we use the neural network machine translation model proposed in [23], called "sequence-to-sequence learning".

Sequence-to-sequence learning, which makes use of long short-term memory networks (LSTMs) [10] to predict sequences of text, was originally applied to Machine Translation but has since seen success in other domains such as image captioning [25] and speech recognition [6].

Other recent works have also applied recurrent neural networks (RNNs) or LSTMs to full response prediction [21], [20], [19], [24]. In [21] the authors rely on having an SMT system to generate n-best lists, while [19] and [24], like this work, develop fully generative models. Our approach is most similar to the Neural Conversation Model [24], which uses sequence-to-sequence learning to model tech support chats and movie subtitles.

The primary difference of our work is that it was deployed in a production setting, which raised the challenges of response quality, utility, scalability, and privacy. These challenges were not considered in any of these related works and led to our novel solutions explained in the rest of this paper.

Furthermore, in this work we approach a different domain than [21], [20], [19], and [24], which primarily focus on social media and movie dialogues. In both of those domains it can be acceptable to provide a response that is merely related or on-topic. Email, on the other hand, frequently expresses a request or intent which must be addressed in the response.

**Identifying a target response space.** Our approach here builds on the Expander graph learning approach [15], since it scales well to both large data (vast amounts of email) and large output sizes (many different underlying semantic intents). While Expander was originally proposed for knowledge expansion and classification tasks [26], our work is the first to use it to discover semantic intent clusters from user-generated content.

Other graph-based semi-supervised learning techniques have been explored in the past for more traditional classification problems [27, 5]. Other related works have explored tasks involving semantic classification [12] or identifying word-level intents [17] targeted towards Web search queries and other forums [7]. However, the problem settings and tasks themselves are significantly different from what is addressed in our work.

Finally, we note that Smart Reply is the first work to address these tasks together and solve them in a single end-to-end, deployable system.

# 3. SELECTING RESPONSES

The fundamental task of the Smart Reply system is to find the most likely response given an original message. In other words, given original message $\mathbf{o}$ and the set of all possible responses $R$, we would like to find:

$$\mathbf{r}^* = \underset{r \in R}{\operatorname{argmax}} P(\mathbf{r}|\mathbf{o})$$

To find this response, we will construct a model that can score responses and then find the highest scoring response.

We will next describe how the model is formulated, trained, and used for inference. Then we will discuss the core challenges of bringing this model to produce high quality suggestions on a large scale.

## 3.1 LSTM model

Since we are scoring one sequence of tokens $\mathbf{r}$, conditional on another sequence of tokens $\mathbf{o}$, this problem is a natural fit for sequence-to-sequence learning [23]. The model itself is an LSTM. The input is the tokens of the original message $\{o_1, ..., o_n\}$, and the output is the conditional probability distribution of the sequence of response tokens given the input:

$$P(r_1, ..., r_m|o_1, ..., o_n)$$

As in [23], this distribution can be factorized as:

$$P(r_1, ..., r_m|o_1, ..., o_n) = \prod_{i=1}^{m} P(r_i|o_1, ..., o_n, r_1, ..., r_{i-1})$$

First, the sequence of original message tokens, including a special end-of-message token $o_n$, are read in, such that the LSTM's hidden state encodes a vector representation of the whole message. Then, given this hidden state, a softmax output is computed and interpreted as $P(r_1|o_1, ..., o_n)$, or the probability distribution for the first response token. As response tokens are fed in, the softmax at each timestep $t$ is interpreted as $P(r_t|o_1, ..., o_n, r_1, ..., r_{t-1})$. Given the factorization above, these softmaxes can be used to compute $P(r_1, ..., r_m|o_1, ..., o_n)$.

**Training** Given a large corpus of messages, the training objective is to maximize the log probability of observed responses, given their respective originals:

$$\sum_{(\mathbf{o},\mathbf{r})} \log P(r_1, ..., r_m|o_1, ..., o_n)$$

We train against this objective using stochastic gradient descent with AdaGrad [8]. Ten epochs are run over a message corpus which will be described in Section 7.1. Due to the size of the corpus, training is run in a distributed fashion using the TensorFlow library [1].

Both our input and output vocabularies consist of the most frequent English words in our training data after preprocessing (steps described in Section 7.1). In addition to the standard LSTM formulation, we found that the addition of a recurrent projection layer [18] substantially improved both the quality of the converged model and the time to converge. We also found that gradient clipping (with the value of 1) was essential to stable training.

**Inference** At inference time we can feed in an original message and then use the output of the softmaxes to get a probability distribution over the vocabulary at each timestep. These distributions can be used in a variety of ways:

1. To draw a random sample from the response distribution $P(r_1, ..., r_m|o_1, ..., o_n)$. This can be done by sampling one token at each timestep and feeding it back into the model.

2. To approximate the most likely response, given the original message. This can be done greedily by taking the most likely token at each time step and feeding it back in. A less greedy strategy is to use a beam search, i.e., take the top $b$ tokens and feed them in, then retain the $b$ best response prefixes and repeat.

| Query | Top generated responses |
|---|---|
| Hi, I thought it would be great for us to sit down and chat. I am free Tuesday and Wenesday. Can you do either of those days?<br><br>Thanks!<br><br>–Alice | I can do Tuesday.<br>I can do Wednesday.<br>How about Tuesday?<br>I can do Tuesday!<br>I can do Tuesday. What time works for you?<br>I can do Wednesday!<br>I can do Tuesday or Wednesday.<br>How about Wednesday?<br>I can do Wednesday. What time works for you?<br>I can do either. |

**Table 1: Generated response examples.**

3. To determine the likelihood of a specific response candidate. This can be done by feeding in each token of the candidate and using the softmax output to get the likelihood of the next candidate token.

Table 1 shows some example of generating the approximate most likely responses using a beam search.

## 3.2 Challenges

As described thus far, the model can generate coherent and plausible responses given an incoming email message. However, several key challenges arise when bringing this model into production.

**Response quality** In order to surface responses to users, we need to ensure that they are *always* high quality in style, tone, diction, and content.

Given that the model is trained on a corpus of real messages, we have to account for the possibility that the most probable response is not necessarily a high quality response. Even a response that occurs frequently in our corpus may not be appropriate to surface back to users. For example, it could contain poor grammar, spelling, or mechanics (*your the best!*); it could convey a familiarity that is likely to be jarring or offensive in many situations (*thanks hon!*); it could be too informal to be consistent with other *Inbox* intelligence features (*yup, got it thx*); it could convey a sentiment that is politically incorrect, offensive, or otherwise inappropriate (*Leave me alone*).

While restricting the model vocabulary might address simple cases such as profanity and spelling errors, it would not be sufficient to capture the wide variability with which, for example, politically incorrect statements can be made. Instead, we use semi-supervised learning (described in detail in Section 4) to construct a target response space $R$ comprising only high quality responses. Then we use the model described here to choose the best response in $R$, rather than the best response from any sequence of words in its vocabulary.

**Utility** Our user studies showed that suggestions are most useful when they are highly specific to the original message and express diverse intents. However, as column 1 in Table 2 shows, the raw output of the model tends to (1) favor common but unspecific responses and (2) have little diversity.

First, to improve specificity of responses, we apply some light normalization that penalizes responses which are applicable to a broad range of incoming messages. The results of this normalization can be seen in column 2 of Table 2. For example, the very generic "Yes!" has fallen out of the top ten. Second, to increase the breadth of options shown to the user, we enforce diversity by exploiting the semantic structure of $R$, as we will discuss in Section 5. The results of this are also shown at the bottom of Table 2.

We further improve the utility of suggestions by first passing each message through a triggering model (described in Section 6) that determines whether suggestions should be generated at all. This reduces the likelihood that we show suggestions when they would not be used anyway.

**Scalability** Our model needs to be deployed in a production setting and cannot introduce latency to the process of email delivery, so scalability is critical.

Exhaustively scoring every response candidate $r \in R$, would require $O(|R|l)$ LSTM steps where $l$ is the length of the longest response. In previous work [23], this could be afforded because computations were performed in a batch process offline. However, in the context of an email delivery pipeline, time is a much more precious resource. Furthermore, given the tremendous diversity with which people communicate and the large number of email scenarios we would like to cover, we can assume that $R$ is very large and only expected to grow over time. For example, in a uniform sample of 10 million short responses (say, responses with at most 10 tokens), more than 40% occur only once. Therefore, rather than performing an exhaustive scoring of every candidate $r \in R$, we would like to efficiently search for the best responses such that complexity is *not* a function of $|R|$.

Our search is conducted as follows. First, the elements of $R$ are organized into a trie. Then, we conduct a left-to-right beam search, but only retain hypotheses that appear in the trie. This search process has complexity $O(bl)$ for beam size $b$ and maximum response length $l$. Both $b$ and $l$ are typically in the range of 10-30, so this method dramatically reduces the time to find the top responses and is a critical element of making this system deployable. In terms of quality, we find that, although this search only approximates the best responses in $R$, its results are very similar to what we would get by scoring and ranking all $r \in R$, even for small $b$. At $b = 128$, for example, the top scoring response found by this process matches the true top scoring response 99% of the time. Results for various beam sizes are shown in Figure 3.

Additionally, requiring that each message first pass through a triggering model, as mentioned above, has the additional benefit of reducing the total amount of LSTM computation.

**Privacy** Note that all email data (raw data, preprocessed data and training data) was encrypted. Engineers could only inspect aggregated statistics on anonymized sentences that occurred across many users and did not identify any user. Also, only frequent words are retained. As a result, verifying model's quality and debugging is more complex.

Our solutions for the first three challenges are described further in Sections 4, 5, and 6.

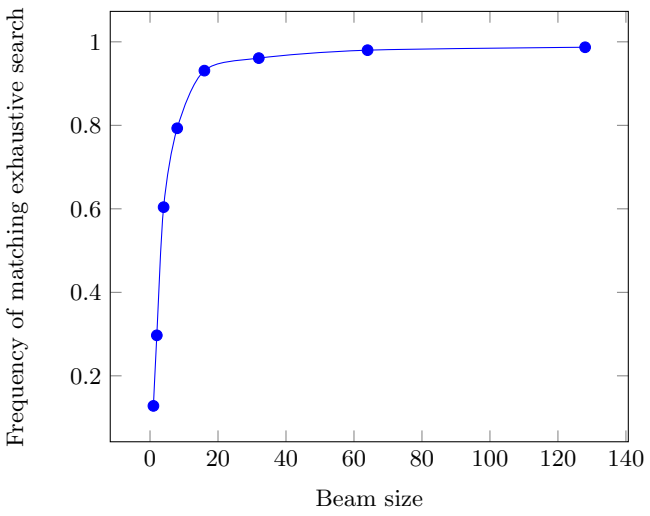



**Figure 3: Effectiveness of searching the response space $R$.** For a sample of messages we compute the frequency with which the best candidate found by a beam search over $R$ matches the best candidate found by exhaustively scoring all members of $R$. We compare various beam sizes. At a beam size of 16, these two methods find the same best response 93% of the time.

# 4. RESPONSE SET GENERATION

Two of the core challenges we face when building the end to end automated response system are *response quality* and *utility*. Response quality comes from suggesting "high quality" responses that deliver a positive user experience. Utility comes from ensuring that we don't suggest multiple responses that capture the same intent (for example, minor lexical variations such as "*Yes, I'll be there.*" and "*I will be there.*"). We can consider these two challenges jointly.

We first need to define a target response space that comprises high quality messages which can be surfaced as suggestions. The goal here is to generate a structured response set that effectively captures various intents conveyed by people in natural language conversations. The target response space should capture both variability in language and intents. The result is used in two ways downstream—(a) define a response space for scoring and selecting suggestions using the model described in Section 3, and (b) promote diversity among chosen suggestions as discussed in Section 5.

We construct a response set using only the most frequent anonymized sentences aggregated from the preprocessed data (described in Section 7.1). This process yields a few million unique sentences.

## 4.1 Canonicalizing email responses

The first step is to automatically generate a set of canonical responses messages that capture the variability in language. For example, responses such as "*Thanks for your kind update.*", "*Thank you for updating!*", "*Thanks for the status update.*" may appear slightly different on the surface but in fact convey the same information. We parse each sentence using a dependency parser and use its syntactic structure to generate a canonicalized representation. Words (or phrases) that are modifiers or unattached to head words are ignored.

## 4.2 Semantic intent clustering

In the next step, we want to partition all response messages into "semantic" clusters where a cluster represents a meaningful response intent (for example, "*thank you*" type of response versus "*sorry*" versus "*cannot make it*"). All messages within a cluster share the same semantic meaning but may appear very different. For example, "*Ha ha*", "*lol*" and "*Oh that's funny!*" are associated with the *funny* cluster.

This step helps to automatically digest the entire information present in frequent responses into a coherent set of semantic clusters. If we were to build a semantic intent prediction model for this purpose, we would need access to a large corpus of sentences annotated with their corresponding semantic intents. However, this is neither readily available for our task nor at this scale. Moreover, unlike typical machine learning classification tasks, the semantic intent space cannot be fully defined a priori. So instead, we model the task as a semi-supervised machine learning problem and use scalable graph algorithms [15] to automatically learn this information from data and a few human-provided examples.

## 4.3 Graph construction

We start with a few manually defined clusters sampled from the top frequent messages (e.g., *thanks*, *i love you*, *sounds good*). A small number of example responses are added as "seeds" for each cluster (for example, *thanks* → "*Thanks!*", "*Thank you.*").[1]

We then construct a base graph with frequent response messages as nodes ($V_R$). For each response message, we further extract a set of lexical features (ngrams and skip-grams of length up to 3) and add these as "feature" nodes ($V_F$) to the same graph. Edges are created between a pair of nodes $(u, v)$ where $u \in V_R$ and $v \in V_F$ if $v$ belongs to the feature set for response $u$. We follow the same process and create nodes for the manually labeled examples $V_L$. We make an observation that in some cases an incoming original message could potentially be treated as a response to another email depending on the context. For example, consider the following (original, response) message pairs:

*Let us get together soon.* → *When should we meet?*
*When should we meet?* → *How about Friday?*

Inter-message relations as shown in the above example can be modeled within the same framework by adding extra edges between the corresponding message nodes in the graph.

## 4.4 Semi-supervised learning

The constructed graph captures relationships between similar canonicalized responses via the feature nodes. Next, we learn a semantic labeling for all response nodes by propagating semantic intent information from the manually labeled examples through the graph. We treat this as a semi-supervised learning problem and use the distributed EXPANDER [15] framework for optimization. The learning framework is scalable and naturally suited for semi-supervised graph propagation tasks such as the semantic clustering problem described here. We minimize the following objective function for response nodes in the graph:

[1]In practice, we pick 100 clusters and on average 3–5 labeled seed examples per cluster.

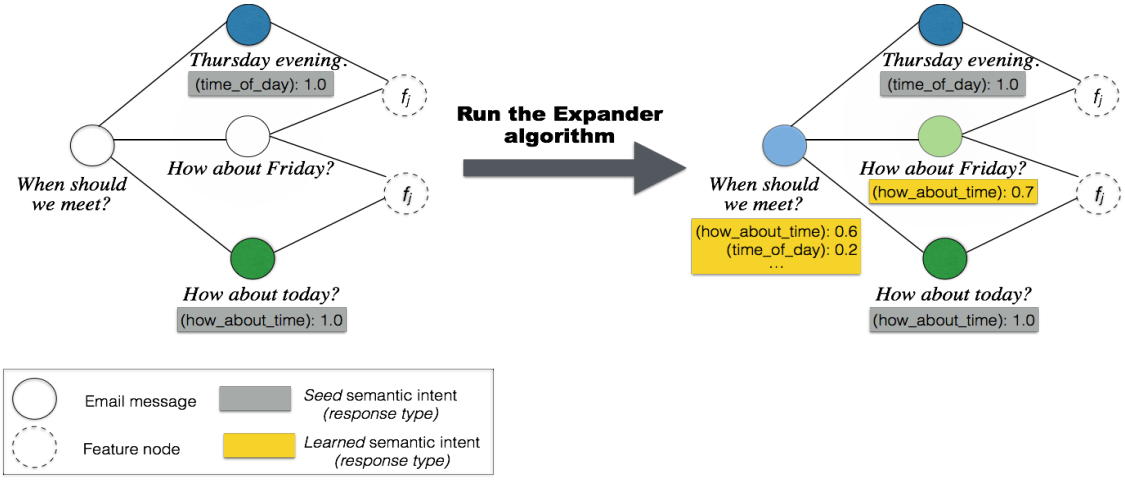



**Figure 4: Semantic clustering of response messages.**

$$s_i||\hat{C}_i - C_i||^2 + \mu_{pp}||\hat{C}_i - U||^2 + \mu_{np}\Big(\sum_{j\in\mathcal{N}_{\mathcal{F}}(i)} w_{ij}||\hat{C}_i - \hat{C}_j||^2 + \sum_{j\in\mathcal{N}_{\mathcal{R}}(i)} w_{ik}||\hat{C}_i - \hat{C}_k||^2\Big) \quad (1)$$

where $s_i$ is an indicator function equal to 1 if the node $i$ is a seed and 0 otherwise, $\hat{C}_i$ is the learned semantic cluster distribution for response node $i$, $C_i$ is the true label distribution (i.e., for manually provided examples), $\mathcal{N}_{\mathcal{F}}(i)$ and $\mathcal{N}_{\mathcal{R}}(i)$ represent the feature and message neighborhood of node $i$, $\mu_{np}$ is the predefined penalty for neighboring nodes with divergent label distributions, $\hat{C}_j$ is the learned label distribution for feature neighbor $j$, $w_{ij}$ is the weight of feature $j$ in response $i$, $\mu_{pp}$ is the penalty for label distribution deviating from the prior, a uniform distribution $U$.

The objective function for a feature node is alike, except that there is no first term, as there are no seed labels for feature nodes:

$$\mu_{np}\sum_{i\in\mathcal{N}(j)} w_{ij}||\hat{C}_j - \hat{C}_i||^2 + \mu_{pp}||\hat{C}_j - U||^2 \quad (2)$$

The objective function is jointly optimized for all nodes in the graph.

The output from EXPANDER is a learned distribution of semantic labels for every node in the graph. We assign the top scoring output label as the semantic intent for the node, labels with low scores are filtered out. Figure 4 illustrates this process.

To discover new clusters which are not covered by the labeled examples, we run the semi-supervised learning algorithm in repeated phases. In the first phase, we run the label propagation algorithm for 5 iterations. We then fix the cluster assignment, randomly sample 100 new responses from the remaining unlabeled nodes in the graph. The sampled nodes are treated as potential new clusters and labeled with their canonicalized representation. We rerun label propagation with the new labeled set of clusters and repeat this procedure until convergence (i.e., until no new clusters are discovered and members of a cluster do not change between iterations). The iterative propagation method thereby allows us to both expand cluster membership as well as discover (up to 5X) new clusters, where each cluster has an interpretable semantic interpretation.

### 4.5 Cluster Validation

Finally, we extract the top $k$ members for each semantic cluster, sorted by their label scores. The set of (response, cluster label) pairs are then validated by human raters. The raters are provided with a response $R_i$, a corresponding cluster label $C$ (e.g., *thanks*) as well as few example responses belonging to the cluster (e.g., "*Thanks!*", "*Thank you.*") and asked whether $R_i$ belongs to $C$.

The result is an automatically generated and validated set of high quality response messages labeled with semantic intent. This is subsequently used by the response scoring model to search for approximate best responses to an incoming email (described earlier in Section 3) and further to enforce diversity among the top responses chosen (Section 5).

## 5. SUGGESTION DIVERSITY

As discussed in Section 3, the LSTM first processes an incoming message and then selects the (approximate) best responses from the target response set created using the method described in Section 4. Recall that we follow this by some light normalization to penalize responses that may be too general to be valuable to the user. The effect of this normalization can be seen by comparing columns 1 and 2 of Table 2. For example, the very generic "Yes!" falls out of the top ten responses.

Next, we need to choose a small number of options to show the user. A straight-forward approach would be to just choose the $N$ top responses and present them to the user. However, as column 2 of Table 2 shows, these responses tend to be very similar.

The likelihood of at least one response being useful is greatest when the response options are not redundant, so it would be wasteful to present the user with three variations of, say, *I'll be there.* The job of the diversity component is to select a more varied set of suggestions using two strategies: omitting redundant responses and enforcing negative or positive responses.

## 5.1 Omitting Redundant Responses

This strategy assumes that the user should never see two responses of the same *intent*. An intent can be thought of as a cluster of responses that have a common communication purpose, e.g. confirming, asking for time or rejecting participation. In Smart Reply, every target response is associated with exactly one intent. Intents are defined based on automatic clustering followed by human validation as discussed in Section 4.

The actual diversity strategy is simple: the top responses are iterated over in the order of decreasing score. Each response is added to the list of suggestions, unless its intent is already covered by a response on the suggestion list. The resulting list contains only the highest-scored representative of each intent, and these representatives are ordered by decreasing score.

## 5.2 Enforcing Negatives and Positives

We have observed that the LSTM has a strong tendency towards producing positive responses, whereas negative responses such as *I can't make it* or *I don't think so* typically receive low scores. We believe that this tendency reflects the style of email conversations: positive replies may be more common, and where negative responses are appropriate, users may prefer a less direct wording.

Nevertheless, we think that it is important to offer negative suggestions in order to give the user a real choice. This policy is implemented through the following strategy:

> If the top two responses (after omitting redundant responses) contain at least one positive response and none of the top three responses are negative, the third response is replaced with a negative one.

A positive response is one which is clearly affirmative, e.g. one that starts with *Yes*, *Sure* or *Of course*. In order to find the negative response to be included as the third suggestion, a second LSTM pass is performed. In this second pass, the search is restricted to only the negative responses in the target set (refer Table 2 for scored negative response examples). This is necessary since the top responses produced in the first pass may not contain any negatives.

Even though missing negatives are more common, there are also cases in which an incoming message triggers exclusively negative responses. In this situation, we employ an analogous strategy for enforcing a positive response.

The final set of top scoring responses (bottom row in Table 2) are then presented to the user as suggestions.

| Unnormalized Responses | Normalized Responses |
|---|---|
| Yes, I'll be there. | Sure, I'll be there. |
| Yes, I will be there. | Yes, I can. |
| I'll be there. | Yes, I can be there. |
| Yes, I can. | Yes, I'll be there. |
| What time? | Sure, I can be there. |
| I'll be there! | Yeah, I can. |
| I will be there. | Yeah, I'll be there. |
| Sure, I'll be there. | Sure, I can. |
| Yes, I can be there. | Yes. I can. |
| Yes! | Yes, I will be there. |

| Normalized Negative Responses |
|---|
| Sorry, I won't be able to make it tomorrow. |
| Unfortunately I can't. |
| Sorry, I won't be able to join you. |
| Sorry, I can't make it tomorrow. |
| No, I can't. |
| Sorry, I won't be able to make it today. |
| Sorry, I can't. |
| I will not be available tomorrow. |
| I won't be available tomorrow. |
| Unfortunately, I can't. |

| Final Suggestions |
|---|
| Sure, I'll be there. |
| Yes, I can. |
| Sorry, I won't be able to make it tomorrow. |

**Table 2: Different response rankings for the message *"Can you join tomorrow's meeting?"***

# 6. TRIGGERING

The *triggering* module is the entry point of the Smart Reply system. It is responsible for filtering messages that are bad candidates for suggesting responses. This includes emails for which short replies are not appropriate (e.g., containing open-ended questions or sensitive topics), as well as emails for which no reply is necessary at all (e.g., promotional emails and auto-generated updates).

The module is applied to every incoming email just after the preprocessing step. If the decision is negative, we finish the execution and do not show any suggestions (see Figure 2). Currently, the system decides to produce a Smart Reply for roughly 11% of messages, so this process vastly reduces the number of useless sugestions seen by the users. An additional benefit is to decrease the number of calls to the more expensive LSTM inference, which translates into smaller infrastructure cost.

There are two main requirements for the design of the triggering component. First, it has to be good enough to figure out cases where the response is not expected. Note that this is a very different goal than just scoring a set of responses. For instance, we could propose several valid replies to a newsletter containing a sentence "*Where do you want to go today?*", but most likely all of the responses would be useless for our users. Second, it has to be fast: it processes hundreds of millions of messages daily, so we aim to process each message within milliseconds.

The main part of the triggering component is a feedforward neural network which produces a probability score for every incoming message. If the score is above some threshold, we *trigger* and run the LSTM scoring. We have adopted this approach because feedforward networks have repeatedly been shown to outperform linear models such as SVM or linear regression on various NLP tasks (see for example [9]).

## 6.1 Data and Features

In order to label our training corpus of emails, we use as positive examples those emails that have been responded to. More precisely, out of the data set described in Section 7.1, we create a training set that consists of pairs $(\mathbf{o}, y)$, where $\mathbf{o}$ is an incoming message and $y \in \{true, false\}$ is a boolean label, which is *true* if the message had a response and *false* otherwise. For the positive class, we consider only messages that were replied to from a mobile device, while for negative

we use a subset of all messages. We downsample the negative class to balance the training set. Our goal is to model $P(y = true \mid \mathbf{o})$, the probability that message $\mathbf{o}$ will have a response on mobile.

After preprocessing (described in Section 7.1), we extract content features (e.g. unigrams, bigrams) from the message body, subject and headers. We also use various social signals like whether the sender is in recipient's address book, whether the sender is in recipient's social network and whether the recipient responded in the past to this sender.

## 6.2 Network Architecture and Training

We use a feedforward multilayer perceptron with an embedding layer (for a vocabulary of roughly one million words) and three fully connected hidden layers. We use feature hashing to bucket rare words that are not present in the vocabulary. The embeddings are separate for each sparse feature type (eg. unigram, bigram) and within one feature type, we aggregate embeddings by summing them up. Then, all sparse feature embeddings are concatenated with each other and with the vector of dense features (those are real numbers and boolean values mentioned in Section 6.1).

We use the ReLu [13] activation function for non-linearity between layers. The dropout [22] layer is applied after each hidden layer. We train the model using AdaGrad [8] optimization algorithm with logistic loss cost function. Similarly to the LSTM, the training is run in a distributed fashion using the TensorFlow library [1].

# 7. EVALUATION AND RESULTS

In this section, we describe the training and test data, as well as preprocessing steps used for all messages. Then, we evaluate different components of the Smart Reply system and present overall usage statistics.

## 7.1 Data

To generate the training data for all Smart Reply models from sampled accounts, we extracted all pairs of an incoming message and the user's response to that message. For training the triggering model (see Section 6), we additionally sampled a number of incoming personal messages which the user didn't reply to. At the beginning of Smart Reply pipeline (Figure 2), data is preprocessed in the following way:

**Language detection** The language of the message is identified and non-English messages are discarded.

**Tokenization** Subject and message body are broken into words and punctuation marks.

**Sentence segmentation** Sentences boundaries are identified in the message body.

**Normalization** Infrequent words and entities like personal names, URLs, email addresses, phone numbers etc. are replaced by special tokens.

**Quotation removal** Quoted original messages and forwarded messages are removed.

**Salutation/close removal** Salutations like *Hi John* and closes such as *Best regards, Mary* are removed.

After the preprocessing steps, the size of the training set is 238 million messages, which include 153 million messages that have no response.

## 7.2 Results

The most important end-to-end metric for our system is the fraction of messages for which it was used. This is currently 10% of all mobile replies. Below we describe in more detail evaluation stats for different components of the system. We evaluate all parts in isolation using both offline analysis as well as online experiments run on a subset of accounts.

### 7.2.1 Triggering results

In order to evaluate the triggering model, we split the data set described in Section 6.1 into train (80%) and test (20%) such that all test messages are delivered after train messages. This is to ensure that the test conditions are similar to the final scenario. We use a set of standard binary classifier metrics: precision, recall and the area under the ROC curve. The AUC of the triggering model is 0.854. We also compute the fraction of triggered messages in the deployed system, which is 11%. We observed that it may be beneficial to slightly over-trigger, since the cost of presenting a suggestion, even if it is not used, is quite low.

### 7.2.2 Response selection results

We evaluate the LSTM scoring model on three standard metrics: Perplexity, Mean Reciprocal Rank and Precision@K.

*Perplexity.*

Perplexity is a measure of how well the model has fit the data: a model with lower perplexity assigns higher likelihood to the test responses, so we expect it to be better at predicting responses. Intuitively, a perplexity equal to $k$ means that when the model predicts the next word, there are on average $k$ likely candidates. In particular, for the ideal scenario of perplexity equal to 1, we always know exactly what should be the next word. The perplexity on a set of $N$ test samples is computed using the following formula:

$$P_r = \exp(-\frac{1}{W}\sum_{i=1}^{N}\ln(\hat{P}(r_1^i, ..., r_m^i | o_1^i, ..., o_n^i)))$$

where $W$ is the total number of words in all $N$ samples, $\hat{P}$ is the learned distribution and $\mathbf{r}^i$, $\mathbf{o}^i$ are the $i$-th response and original message. Note that in the equation above only response terms are factored into $P_r$. The perplexity of the Smart Reply LSTM is 17.0. By comparison, an n-grams language model with Katz backoff [11] and a maximum order of 5 has a perplexity of 31.4 on the same data (again, computed only from response terms).

*Response ranking.*

While perplexity is a quality indicator, it does not actually measure performance at the scoring task we are ultimately interested in. In particular, it does not take into account the constraint of choosing a response in $R$. Therefore we also evaluate the model on a response ranking task: for each of $N$ test message pairs $(o, r)$ for which $r \in R$, we compute $s = P(r|o)$ and $\forall_i x_i = P(w_i|o)$, where $w_i$ is the $i$-th element of $R$. Then we sort the set $R = \{s, x_1, \ldots, x_N\}$ in descending order. Finally, we define $rank_i = \text{argmin}_j(R_j | R_j = s)$. Put simply, we are finding the rank of the actual response with respect to all elements in $R$.

| Model | Precision@10 | Precision@20 | MRR |
|---|---|---|---|
| Random | $5.58e-4$ | $1.12e-3$ | $3.64e-4$ |
| Frequency | 0.321 | 0.368 | 0.155 |
| Multiclass-BOW | 0.345 | 0.425 | 0.197 |
| Smart Reply | 0.483 | 0.579 | 0.267 |

**Table 3: Response ranking**

Using this value, we can compute the Mean Reciprocal Rank:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$$

Additionally we can compute Precision@K. For a given value of $K$ it is computed as the number of cases for which target response $r$ was within the top$K$ responses that were ranked by the model.

We compare the Smart Reply response selection model to three baselines on the same ranking task. The *Random* baseline ranks $R$ randomly. The *Frequency* baseline ranks them in order of their frequency in the training corpus. This baseline captures the extent to which we can simply suggest highly frequent responses without regard for the contents of the original message. The *Multiclass-BOW* baseline ranks $R$ using a feedforward neural network whose input is the original message, represented with bag of words features, and whose output is a distribution over the elements of $R$ (a softmax).

As shown in Table 3, the Smart Reply LSTM significantly improves on the *Frequency* baseline, demonstrating that conditioning on the original message is effective; the model successfully extracts information from the original message and uses it to rank responses more accurately.

It also significantly outperforms the *Multiclass-BOW* baseline. There are a few possible explanations for this. First, the recurrent architecture allows the model to learn more sophisticated language understanding than bag of words features. Second, when we pose this as a mulitclass prediction problem, we can only train on messages whose response is in $R$, a small fraction of our data. On the other hand, the sequence-to-sequence framework allows us to take advantage of all data in our corpus: the model can learn a lot about original-response relationships even when the response does not appear in $R$ exactly.

Note that an added disadvantage of the multiclass formulation is that it tightly couples the training of the model to the construction of $R$. We expect $R$ to grow over time, given the incredible diversity with which people communicate. While a simpler application such as chat might only need a small number of possible responses, we find that for email we will need a tremendous number of possible suggestions to really address users' needs.

### 7.2.3 *Diversity results*

We justify the need for both the diversity component and a sizable response space $R$ by reporting statistics around unique suggestions and clusters in Table 4. The Smart Reply system generates daily 12.9k unique suggestions that belong to 376 unique semantic clusters. Out of those, people decide to use 4,115, or 31.9% of, unique suggestions and 313, or 83.2% of, unique clusters. Note, however, that many suggestions are never seen, as column 2 shows: the user may not open an email, use the web interface instead of mobile

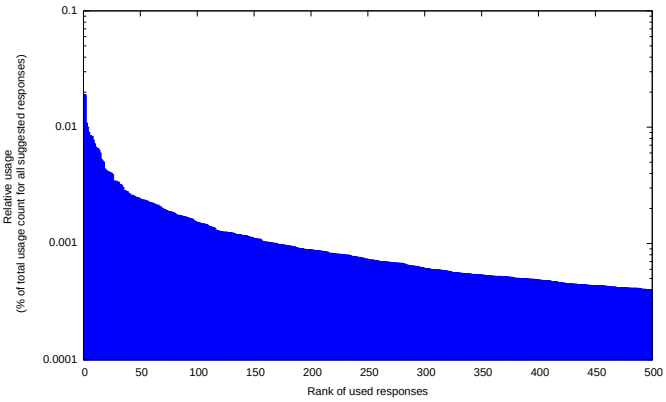



**Figure 5: Usage distribution for top suggested responses.**

| | Daily Count | Seen | Used |
|---|---|---|---|
| Unique Clusters | 376 | 97.1% | 83.2% |
| Unique Suggestions | 12.9k | 78% | 31.9% |

**Table 4: Unique cluster/suggestions usage per day**

or just not scroll down to the bottom of the message. Also, only one of the three displayed suggestions will be selected by the user. These statistics demonstrate the need to go well beyond a simple system with 5 or 10 canned responses.

Figure 5 and Figure 6 present, respectively, the distribution of the rank for suggested responses and the distribution of suggested clusters. The tail of the cluster distribution is long, which explains the poor performance of *Frequency* baseline described in Section 7.2.2.

We also measured how Smart Reply suggestions are used based on their location on a screen. Recall that Smart Reply always presents 3 suggestions, where the first suggestion is the top one. We observed that, out of all used suggestions, 45% were from the $1st$ position, 35% from the 2nd position and 20% from the 3rd position. Since usually the third position is used for diverse responses, we conclude that the diversity component is crucial for the system quality.

Finally, we measured the impact of enforcing a diverse set of responses (e.g., by not showing two responses from the same semantic cluster) on user engagement: when we completely disabled the diversity component and simply suggested the three suggestions with the highest scores, the click-through rate decreased by roughly 7.5% relative.

## 8. CONCLUSIONS

We presented Smart Reply, a novel end-to-end system for automatically generating short, complete email responses. The core of the system is a state-of-the-art deep LSTM model that can predict full responses, given an incoming email message. To successfully deploy this system in *Inbox by Gmail*, we addressed several challenges:

- We ensure that individual response options deliver *quality* by selecting them from a carefully constructed response space. The responses are identified by a novel method for semantic clustering.
- We increase the total *utility* of our chosen combina-

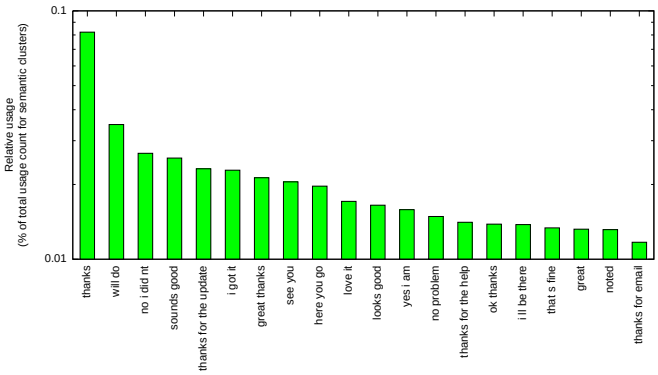



**Figure 6: Usage distribution for semantic clusters corresponding to top suggested responses.**

tion of suggestions by enforcing diversity among them, and filtering traffic for which suggestions would not be useful.

- We build a *scalable* system by efficiently searching the target response space.

Our clearest metric of success is the fact that 10% of mobile replies in *Inbox* are now composed with assistance from the Smart Reply system. Furthermore, we have designed the system in such a way that it is easily extendable to address additional user needs; for instance, the architecture of our core response scoring model is language agnostic, therefore accommodates extension to other languages in addition to English.

## 9. ACKNOWLEDGMENTS

The authors would like to thank Oriol Vinyals and Ilya Sutskever for many helpful discussions and insights, as well as Prabhakar Raghavan for his guidance and support.

## 10. REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, and et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015.

[2] I. G. P. Affairs. Interconnected world: Communication & social networking. Press Release, March 2012. http://www.ipsos-na.com/news-polls/pressrelease.aspx?id=5564.

[3] Y. Artzi, P. Pantel, and M. Gamon. Predicting responses to microblog posts. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 602–606, Montréal, Canada, June 2012. Association for Computational Linguistics.

[4] L. Backstrom, J. Kleinberg, L. Lee, and C. Danescu-Niculescu-Mizil. Characterizing and curating conversation threads: Expansion, focus, volume, re-entry. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 13–22, 2013.

[5] Y. Bengio, O. Delalleau, and N. Le Roux. Label propagation and quadratic criterion. In O. Chapelle, B. Schölkopf, and A. Zien, editors, *Semi-Supervised Learning*, pages 193–216. MIT Press, 2006.

[6] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals. Listen, attend, and spell. *arXiv:1508.01211*, abs/1508.01211, 2015.

[7] Z. Chen, B. Liu, M. Hsu, M. Castellanos, and R. Ghosh. Identifying intention posts in discussion forums. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1041–1050, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[8] J. Duchi, E. Hazad, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12, 2011.

[9] Y. Goldberg. A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726, 2015.

[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] S. M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recogniser. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35:400–401, 1987.

[12] X. Li. Understanding the semantic structure of noun phrase queries. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1337–1345, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[13] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[14] B. Pang and S. Ravi. Revisiting the predictability of language: Response completion in social media. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1489–1499, Jeju Island, Korea, July 2012. Association for Computational Linguistics.

[15] S. Ravi and Q. Diao. Large scale distributed semi-supervised learning using streaming approximation. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

[16] A. Ritter, C. Cherry, and W. B. Dolan. Data-driven response generation in social media. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, Edinburgh, UK, July 2011. Association for Computational Linguistics.

[17] R. Saha Roy, R. Katare, N. Ganguly, S. Laxman, and M. Choudhury. Discovering and understanding word level user intent in web search queries. *Web Semant.*, 30(C):22–38, Jan. 2015.

[18] H. Sak, A. Senior, and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2014.

[19] I. V. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau. Hierarchical neural network generative models for movie dialogues. In *arXiv preprint arXiv:1507.04808*, 2015.

[20] L. Shang, Z. Lu, and H. Li. Neural responding machine for short-text conversation. In *In Proceedings of ACL-IJCNLP*, 2015.

[21] A. Sordoni, M. Galley, M. Auli, C. Brockett, Y. Ji, M. Mitchell, J.-Y. Nie, J. Gao, and B. Dolan. A neural network approach to context-sensitive generation of conversation responses. In *In Proceedings of NAACL-HLT*, 2015.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[23] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[24] O. Vinyals and Q. V. Le. A neural conversation model. In *ICML Deep Learning Workshop*, 2015.

[25] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[26] J. B. Wendt, M. Bendersky, L. Garcia-Pueyo, V. Josifovski, B. Miklos, I. Krka, A. Saikia, J. Yang, M.-A. Cartright, and S. Ravi. Hierarchical label propagation and discovery for machine generated email. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM) (2016)*, 2016.

[27] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 912–919, 2003.