M1 Internship

École Normale Supérieure
Département d'Informatique

# Deep Learning for Natural Language Processing with Large Vocabularies

## Internship Report

Alex Auvolat
alex.auvolat@ens.fr

*April 2015 - August 2015*

*Under the supervision of*
Yoshua Bengio, Pascal Vincent

Montréal Institute for Learning Algorithms
*2920 chemin de la Tour*
*Montréal QC H3T 1J4*
*Canada*

# Table of contents

**1 Introduction** . . . . 4

**2 Deep Learning Blocks and Models** . . . . 4

2.1 Neural Networks and Backpropagation . . . . 4
Artificial Neurons . . . . 4
Artificial Neural Networks . . . . 5
Neuron Layers . . . . 5
Loss Function . . . . 5
Optimization . . . . 5
Backpropagation and Gradient Descent . . . . 5
Tools . . . . 6

2.2 Recurrent Neural Networks . . . . 6
Simple RNNs . . . . 6
Backpropagation Through Time . . . . 6
Vanishing Gradients Problem . . . . 6
Long Short-Term Memory . . . . 7

Generative RNNs . . . 7
2.3 Neural Machine Translation Models . . . 8
Word Embeddings . . . 8
Output Softmax Layer . . . 8
Encoder-Decoder Model . . . 8
Bidirectionnal Encoder RNN . . . 8
Attention Mechanism . . . 9

**3 Optimizing the Output Softmax** . . . 9

3.1 Usual Approaches . . . 10
Hierarchical Softmax . . . 10
Softmax and Inner Product . . . 10
Softmax Approximation . . . 10
Approximate Maximum Inner Product Search . . . 10
Hashing Techniques . . . 10
Other Softmax Approximations . . . 10
3.2 The $k$-means Clustering Approach . . . 11
Maximum Cosine Similarity Search . . . 11
Clustering for Approximate Maximum Cosine Similarity Search . . . 11
Hierarchical Clustering and Parallel Exploration . . . 11
Reduction to MCSS . . . 12
Preliminary Results on a Simple Benchmark Task . . . 12
3.3 Implementation Difficulties . . . 12
NMT Model Implementations . . . 12
Implementation of the Clustering Acceleration . . . 13
Unsatisfactory Results . . . 13

**4 Other Internship Activities** . . . 13

4.1 Taxi Destination Prediction Challenge . . . 13
Taxi Destination Problem . . . 13
Winning Solution . . . 13
Paper . . . 14
4.2 Transposing the Data Matrix . . . 14
Problem . . . 14
Approach . . . 14
MLP Model . . . 14
Two-stage MLP Model . . . 15
Training and Regularization . . . 15
Unsatisfactory results . . . 15

**5 Conclusion** . . . 15

**6 Acknowlegments** . . . 15

**Bibliography** . . . 16

# 1 Introduction

As part of my Master 1 degree at ENS Paris, I had to complete a four-month internship in a research lab outside of France. As I have a long standing interest in machine learning and perspectives towards artificial intelligence, I requested an internship at the renowned Montreal Institute for Learning Algorithms (MILA), specialized in deep learning and led by Pr. Yoshua Bengio. Thanks to Pr. Bengio's benevolence, I was accepted at MILA and was able to do a four-month internship under the supervision of Pr. Bengio and Pr. Pascal Vincent, from April to mid-August 2015.

During this internship, I had the opportunity to work on a variety of subjects, learning a lot about deep learning and widening my perspectives on the vast field of machine learning and artificial intelligence. I also benefited greatly from hands-on experience with the deep learning tools and frameworks developed at MILA, including Theano, a GPU-accelerated matrix computation library that includes powerful symbolic processing, and Blocks, a deep learning framework built using Theano.

My main project, which I worked on mainly with Pr. Pascal Vincent, concerned a computational optimization of deep learning models for natural language tasks such as machine translation. A recurring problem with neural machine translation (NMT) models is the size of the vocabulary, which is typically tens of thousands of words, and is the main computational bottleneck for the training and exploiting of such models. The approach I worked on is based on a variant of $k$-means clustering, and will be the subject of a substantial portion of this report.

In Section 2, I will discuss generalities about deep learning and explain how it is applied to obtain state-of-the-art results on machine translation tasks. In Section 3, I will explain my work on the optimization of the softmax output layer of machine translation models. In Section 4, I will discuss other activities I was involved with during my internship.

# 2 Deep Learning Blocks and Models

Deep learning is a family of machine learning models and architectures based on the principles of neural networks (sometimes abbreviated *neural nets*), which were first proposed in the 50's under the name of Perceptron [20]. Deep learning is a field defined by the studying of artificial neural net models containing several layers of neurons, although the term now includes research on many practical applications of neural networks, not necessarily deep but that have a large number of neurons, and therefore have become practical only recently with the explosion of available computing power concentrated on a single system (typically, a GPU).

Deep neural networks are statistical models that can be adapted and applied to many *supervised learning* tasks, such as *classification* or *regression*. Deep neural networks can also be used as *generative* models, in which case they are trained to produce output such as sentences or images. Some forms of *unsupervised learning* have also been developed for deep neural networks, often as a pre-training step. The model is then fine-tuned with supervised learning.

## 2.1 Neural Networks and Backpropagation

**Artificial Neurons** Artificial neural networks (ANNs) are originally inspired by the functioning of biological neurons, hence their name; however neural nets currently in application use a very simplified model of the computation realized by a biological neuron. In the ANN model, a neuron is a simple unit connected to several real-valued inputs (in particular, the inputs to a neuron may be the outputs of other neurons), and does a sum of the inputs multiplied by weights (sometimes called *synaptic weights*), followed by a bias and a nonlinearity. Mathematically, an artificial neuron is defined by the following equation:

$$f(x_1, ..., x_n) = a(w_1 x_1 + \cdots + w_n x_n + b)$$

where $(w_1, ..., w_n, b)$ are the parameters of the model, and $a$ is a nonlinear function called an *activation function*. Traditional activation functions include the sigmoid function $\sigma$ defined by $\sigma(x) = \frac{1}{1 + e^{-x}}$, or the tanh function. More recently, the rectified linear unit (ReLU) defined by $a(x) = \max(x, 0)$ has become a common choice as it has been shown to improve learning on certain types of deep or recurrent models [11]. The output value of a neuron is usually called its *activation value*, or simply its *activation*.

**Artificial Neural Networks** An ANN is a function that can be calculated by a directed acyclic graph of nodes, where each node is either an input node or a neuron, and one or several neuron nodes are identified as output nodes. An ANN is a parametric function, as the exact computation is defined by the set of weights and biases of all the neurons of the network. In this report, as often in literature, we will refer to the vector of these parameters as $\theta$. The topology of the network, i.e. the number of neurons and their connections, is not part of the parameter vector, as it is not something we can optimize automatically by learning. The topology of the network, as well as other values such as the choice of activation function, are referred to as hyperparameters, and finding a good hyperparameter combination is usually done by hand search guided by intuition.

**Neuron Layers** Neurons are typically grouped in *layers*, meaning that the computation of the function defined by the neural net is defined as the composition of the functions defined by each layer. A neural net layer is typically fully-connected, meaning that each neuron of the layer is connected to each of the inputs of the layer. The computation done by the layer can then be expressed as a matrix multiplication, followed by the addition of a bias vector and finally an element-wise application of the activation function. The definition in terms of matrix multiplication has been an important element in the acceleration of neural net computation, thanks to very efficient BLAS routines implemented on GPUs. The vector of outputs for a layer of neurons is usually called an *activation vector*, or sometimes a *state vector* in the case of recurrent neural networks. A layered architecture of this kind is usually called a *multi-layer perceptron* (MLP).

**Loss Function** To do supervised learning with deep neural networks, we must first define a cost function in relation to our training set. If our training set $\mathcal{X}$ is composed of training examples $(x_i, y_i)$, where $x_i$ is the input and $y_i$ is the output we wish our model to predict, then the loss function (also called cost function) for a neural net $f_\theta$ is defined by:

$$\mathcal{L}(f_\theta) \;=\; \sum_{(x_i, y_i) \in \mathcal{X}} l(f_\theta(x_i), y_i)$$

where $l$ is a loss function that can be defined in various manners depending on the type of task we want our model to solve. Typically for regression we will use a squared $L_2$ distance, whereas for classification we will use the negative log-likelihood of the prediction target.

The goal of machine learning is to find a set of parameters $\theta$ that minimizes the *expected loss* (or *risk*), i.e. the expectation of $l(f_\theta(x), y)$ where $(x, y)$ are drawn from a true-world distribution which is also supposed to have generated the training set $\mathcal{X}$.

**Optimization** In supervised learning, the optimization of the risk is done by optimizing the loss on the training set, which is considered to provide a good first approximation. In mathematical terms, it corresponds to finding $\theta^*$ defined by:

$$\theta^* \;=\; \mathrm{argmin}_\theta \, \mathcal{L}(f_\theta)$$

As the parameter vector $\theta^*$ may easily be too specific to the training set, and provide a function that does not actually minimize the risk very well (a phenomenon called *overfitting*), several techniques called *regularization methods* have been developed. Such methods include the addition of a regularization term to the loss function, and the monitoring of the cost on a separate validation set which is not used during the training.

**Backpropagation and Gradient Descent** The method used predominantly in deep learning for searching for the optimal value of the parameters is gradient descent, where we improve the parameter vector $\theta$ by small steps following the gradient:

$$\theta_{t+1} \;=\; \theta_t - \lambda \, \nabla \mathcal{L}(f_{\theta_t})$$

where $\nabla \mathcal{L}(f_\theta)$ is the vector of the gradient of $\mathcal{L}(f_\theta)$ with respect to the parameter values $\theta$. As we often work on very large datasets and calculating the gradient for the full loss function is very impractical, we usually proceed by *stochastic gradient descents*, where the gradient is approximated at each step by summing the loss over only a small batch of examples (called a *minibatch*). Practical minibatch sizes are of about 100 examples.

The gradient with respect to the parameters $\theta$ can be analytically derived from the computation graph of the neural network by using the chain rule and known analytical derivatives of simple functions. Its effective calculation involves propagating the gradient backwards from the output (where the gradient is proportional to the loss) to the parameters, hence the name of backpropagation [21].

**Tools** The MILA lab has developed many tools enabling the easy construction of complex neural networks. Such tools include Theano [6][3], which is a GPU-accelerated matrix computation library with an interface similar to numpy, and that integrates automatic analytical gradient computation, and Blocks, a deep-learning framework built over Theano that includes many common types of neural network layouts, as well as the stochastic gradient descent algorithm and some of its commonly used variants.

## 2.2 Recurrent Neural Networks

A recurrent neural network (RNN) is a special type of neural network in which the computation may be done in several time steps, with loops that carry a hidden state from one time step to the next. This enables the network to process variable-length sequences, such as for instance reading a sentence by reading one word at each time step. RNNs can also be used to generate variable-length outputs, for instance it can generate a sentence word by word.

**Simple RNNs** The simplest existing RNN model is a mode where a hidden state vector is propagated between time steps, and that hidden state is calculated by a standard neural network layer whose input vector is the concatenation of the previous state vector and the input vector at that time step:

$$\begin{aligned} h_t &= \sigma(W_i x_t + W_r h_{t-1} + b_h) \\ o_t &= \sigma(W_o h_t + b_o) \\ f_\theta(x_1, ..., x_T) &= (o_1, ..., o_T) \end{aligned}$$

The parameters of such a model are the matrices $W_i$, $W_r$ and $W_o$, the bias vectors $b_h$ and $b_o$, as well as the initial state $h_0$.

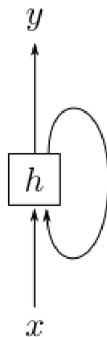


**Figure 1.** Illustration of a RNN as a recurrent loop

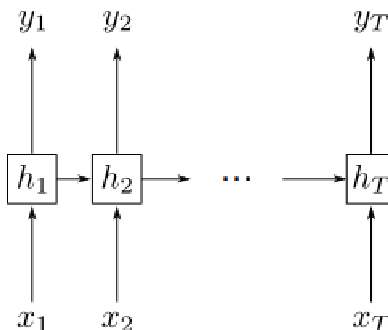


**Figure 2.** Illustration of a RNN unrolled in time

**Backpropagation Through Time** As long as the loss function is defined in terms of $f_\theta(x)$ and $y$, the gradient of the loss with respect to the parameters $\theta$ can be calculated analytically. This involves calculating a gradient for the parameters at each time step, and is usually referred to as *backpropagation through time* [25]. In this process, each time step produces a gradient with respect to all the parameters, and the final gradient is the sum of all these partial gradients. The analytic calculation of these gradients can be done automatically with tools such as Theano, therefore we do not need to enter in the details of such complex derivations.

**Vanishing Gradient Problem** An important problem with recurrent neural networks that has been extensively studied [5][19] is the *vanishing gradient* problem. It arises from the difficulty of backpropagation through time to do correct credit assignment over long time ranges, as the gradient decays exponentially with time and becomes mixed with gradients from other sources. It is one of the main obstacles to the effective training of RNNs with backpropagation through time, and a very active area of research.

**Long Short-Term Memory** Long short-term memory RNNs [15] are a special type of recurrent neural nets that partially solve the vanishing gradient problem. They are composed of more complex equations than standard RNNs, as they implement several *gating* mechanisms, allowing the RNN to ignore some inputs or to keep some state values constant at each time step. An LSTM RNN is described by the following equations:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ \tilde{C}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ f_t &= \sigma(W_f x_t + U_c h_{t-1} + b_c) \\ C_t &= i_t * \tilde{C}_t + f_t * C_{t-1} \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o) \\ h_t &= o_t * \tanh(C_t) \end{aligned}$$

As we can see in the equations, two vectors are propagated from one time step to the next: the state vector $h_t$, as well as a vector $C_t$ called the *cell content vector*. This stems from the interpretation of the LSTM as having a *cell memory* which is invisible to the exterior and is disjoint from its output state $h_t$. $\tilde{C}_t$ is a new cell state candidate vector which takes the input and the previous state into account. $i_t$, $f_t$ and $o_t$ have gating roles, and are called respectively *input gate*, *forget gate* and *output gate*. These gates are used in the calculation of $C_t$ as depending on $\tilde{C}_t$ in proportion to the input gate and on $C_{t-1}$ in proportion to the forget gate, and in the calculation of $h_t$ depending on $C_t$ in proportion to the output gate. A schematic illustration of an LSTM cell is provided in Figure 3.

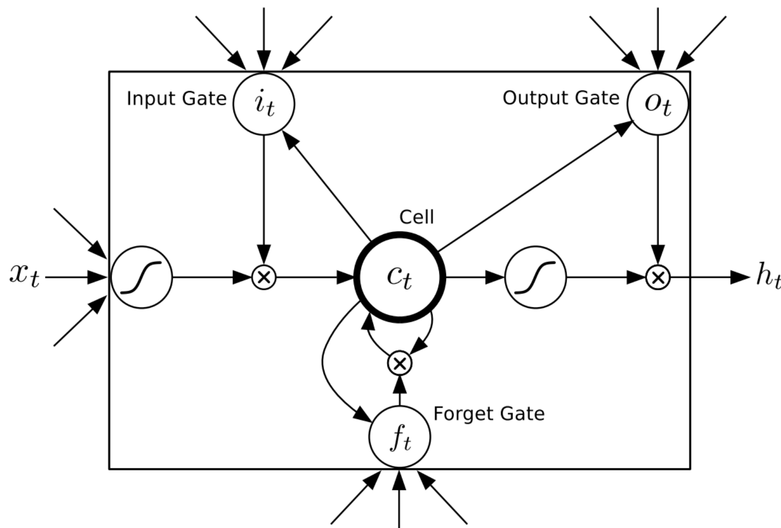


**Figure 3.** Illustration of an LSTM cell

Simpler variants of the LSTM that also include gating mechanisms were developed, such as the gated recurrent unit (GRU) RNN [8], and have been shown to perform almost as well on several tasks.

**Generative RNNs** RNNs can be used as generative models for sequential data [12]. One of the most common uses for generative RNNs is to generate sentences word by word. In such a configuration, the model must predict one word ahead, knowing all the previous words. There are two modes of functioning for such a model: in supervised training, the model is always given a valid sentence as input, and must predict the next word after each prefix of the sentence. In generative mode, the input to the network at each time step is fed back from its output at the previous time step. As with all discrete output neural models, the output of the RNN is a probability vector over all the possible next words. In training mode, we simply minimize the negative log-likelihood of the target word, whereas in generative mode we can use this probability distribution to sample the next word from a list of possibilities, which enables a single network to produce very variable outputs from the same initial state. A common approach to improving the output generation for such models is to use beam search, as is often done when sampling from classical $n$-gram models.

## 2.3 Neural Machine Translation Models

**Word Embeddings** When applying neural networks to textual data, we need a way to represent words in a way that can be given as input to a neural network. A simple approach is to use *word embeddings* [4], i.e. to map each word to a representation in a fixed-size vector space. There are many ways to train word embeddings [9][17], but the simplest method is to consider the embedding matrix as a parameter matrix of the model, and update it with gradient descent. The dimension of word embeddings depends on the model, but recent technological developments have enabled the use of word embeddings of dimension 500 or more, which have proven to be very efficient for many language modeling tasks. As only a limited number of words are presented as input to the network at each minibatch, the gradient associated with the embedding matrix is very sparse, containing non-null values only on the lines corresponding to words that appear in the input. This enables efficient implementation of the SGD algorithm for these large matrices, with a complexity independent of the size of the vocabulary.

**Output Softmax Layer** The output layer of our neural language models are usually very large layers that produce one probability value for each word of our vocabulary. The matrix of coefficients for these words is therefore of the same size as the matrix of embeddings used for the input, and its columns are sometimes also called word embeddings. The output of the network is supposed to be a probability vector, i.e. we want it to be composed of positive values that sum to 1. To do that we usually use a softmax normalization function as the nonlinearity, which is defined by:

$$\text{softmax}(a_1, ..., a_N) = \left( \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}} \right)_{i=1}^{N}$$

**Encoder-Decoder Model** A simple RNN-based machine translation model [7] is based on an encoder-decoder architecture. In this architecture we have a first RNN called the *encoder* that reads the source sentence word by word, producing a representation $C$, and a second RNN called the *decoder* that produces the translated sentence as described above, using the representation $C$ of the source sentence as a supplementary input conditioning the prediction. This architecture is illustrated in Figure 4.

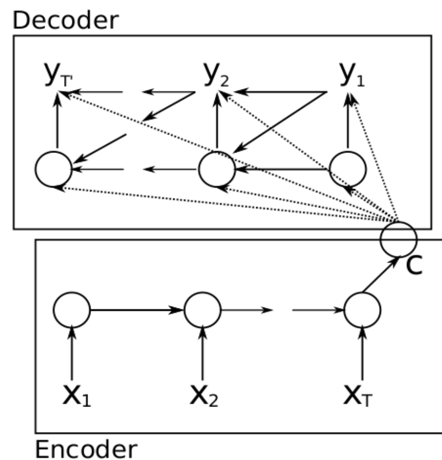


**Figure 4.** Illustration of an encoder-decoder RNN model.

**Bidirectionnal Encoder RNN** An important problem with the RNN approach is that the encoder RNN "forgets" the words it reads along the way, and the state $C$ contains a lot of information about the last few words and very little information about the first words of the source sentence. To alleviate this problem, a bidirectional encoder [13] can be used. This simply consists of using two RNNs to read the sentence, one that reads it normally and the other that reads it in the reverse order. The state $C$ that is passed to the decoder is the concatenation of the last states of the two RNNs, therefore containing information about the beginning as well as the end of the sentence.

**Attention Mechanism** However this simplistic encoder-decoder approach still has a fundamental problem, which is that the intermediate representation $C$ is of fixed dimensionality, whereas the input sentence may be arbitrarily long. It is therefore unlikely that the whole source sentence may be represented in $C$ except for cases where it is very short. To solve this issue, an alternative model based on an *attention mechanism* has been proposed [2]. In this model, all the hidden states generated by the encoder RNNs are considered for generating each word of the input. A separate component of the decoder is responsible for selecting, at each word being generated, which words of the input sequence are relevant and must be considered predominantly. This attention mechanism produces attention coefficients $\alpha_i$ for each of the input words, which are normalized by a softmax operation. The weighted representations are then summed, producing an additional input to the decoder RNN at each output generation step. This approach allows the decoder to use all the information of the input sequence as required, making it much more efficient than the simple encoder-decoder approach. It has yielded state-of-the-art results on standard benchmark machine translation tasks.

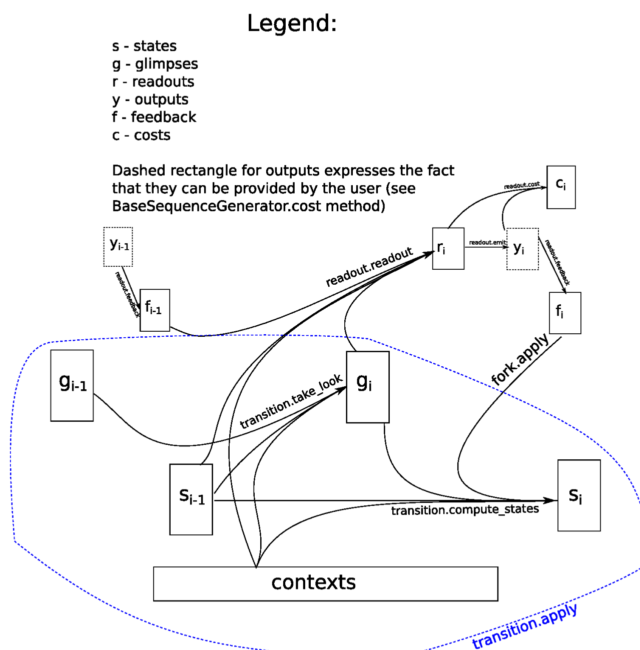


**Figure 5.** Attention mechanism as described in the Blocks documentation

# 3 Optimizing the Output Softmax

The main computational bottleneck in the translation models I have described in Section 2.3 is the output layer of the decoder network, which is composed of a matrix multiplication by a weight matrix of the same size as the vocabulary. The output of this multiplication is passed through a softmax, which assigns to each word of the vocabulary the probability it has of being the next word in the sentence. At first, the full computation of this matrix product seems necessary, as we need to produce a probability value for all the words, however the computation cost can be reduced by a variety of methods.

I have submitted for NIPS 2015 a paper [1] written with Pr. Pascal Vincent which describes the problem and summarizes the previously experimented approaches, as well as presents the approach I have worked on. For the sake of completeness this section will briefly summarize that talk, but I suggest the interested reader consult the paper for more complete information. The paper is publicly available on arXiv at the address http://arxiv.org/abs/1507.05910.

## 3.1 Usual Approaches

**Hierarchical Softmax** A first approach is to replace the simple softmax model by a model that describes the probabilities of the words as a tree of softmax operations [18]. In this approach, calculating the probability value for a single item of the vocabulary is done in logarithmic complexity with respect to the size of the vocabulary. This is useful since the negative log-likelihood loss that we optimize during training is calculated using only the output probability for the target word, therefore we can train the model with logarithmic complexity instead of linear complexity with respect to the number of words. However, a hierarchical probabilistic model has been shown to be usually worse than a simple flat model at assigning correct probabilities. Also a difficulty of this model is that the tree structure has to be determined in advance using a heuristic, and is not allowed to evolve during the training, therefore not fitting the data in an optimal way.

**Softmax and Inner Product** The softmax operation of the output layer is defined as:

$$y_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$$

In the case of a neural net, the activation vector $a$ is calculated as a product of the previous layers activations $h$ with the output matrix $W$. Each $a_i$ is therefore an inner product of the form:

$$a_i = w_i^\top h$$

where $w_i$ is the $i$-th column of the weight matrix $W$. A bias is usually added, but for the purpose of simplicity we will not consider it here. The case with a bias can be easily reduced to our simple case by adding the biases to $W$ and adding a one to $h$. Our softmax values can thus be rewritten:

$$y_i = \frac{e^{w_i^\top h}}{\sum_{j=1}^{N} e^{w_i^\top h}}$$

**Softmax Approximation** The costly computation in the calculation of the softmax is in the denominator, where we have to calculate the inner product of $h$ with all of the columns of $w$. However, the inner products are then exponentiated, meaning that the smaller terms become completely negligible in comparison to the bigger terms. Therefore we can have a reasonable approximation of the softmax's denominator by only choosing the words $w_i$ whose inner product with $h$ are maximal.

**Approximate Maximum Inner Product Search** The problem we face is now to find the words $w_i$ that have a maximal inner product with $h$, which is known as the Maximum Inner Product Search (MIPS) problem. Most techniques do not solve MIPS exactly, but rather approximate $k$-MIPS, i.e. finding with a reasonable approximation the $k$ vectors that have the maximum inner product with $h$.

**Hashing Techniques** Popular solutions to solve approximate $k$-MIPS include locality-sensitive hashing (LSH) based approaches [22][23]. These solutions involve choosing a random partitioning of the search space, which is relevant to finding the items that potentially provide high inner products. The vectors $w_i$ are then associated to a subspace of the partition, and when doing the query to find which vectors provide the maximum inner products, only the vectors associated to a few of the subspaces are considered. Very large language models have successfully been trained by Google researchers [24], using a winner-take-all hashing approximation for the softmax.

**Other Softmax Approximations** Simpler approaches to approximating the softmax denominator have also been tried [16], where the vectors chosen for the approximation are not selected specifically for having a maximum inner product, but simply so that the computation can be sped up the most. In this approach the words selected are the same for a whole minibatch, and are simply a superset of all the words that appear in the minibatch. This has greatly improved the training time without significantly degrading the model's performance.

## 3.2 The $k$-means Clustering Approach

**Maximum Cosine Similarity Search** The *maximum cosine similarity search* (MCSS) problem is a problem very similar to MIPS: it corresponds to finding

$$i^* \;=\; \text{argmax}_i \frac{w_i^\top h}{\|w_i\|_2 \, \|h\|_2}$$

whereas the MIPS problem is simply defined by:

$$i^* \;=\; \text{argmax}_i \, w_i^\top h$$

**Clustering for Approximate Maximum Cosine Similarity Search** $k$-means clustering can be easily adapted in such a way that points having high cosine similarity are in the same cluster, instead of those that are close according to $L_2$ distance. This adaptation, called *spherical k-means*, consists of rewriting the point affectation step (1) and the centroid calculation step (2) as follows:

$$a_i \;\leftarrow\; \text{argmax}_j \frac{w_i^\top}{\|w_i\|_2} c_j \tag{1}$$

$$c_j \;\leftarrow\; \frac{\sum_{i|a_i=j} \frac{w_i}{\|w_i\|_2}}{\left\| \sum_{i|a_i=j} \frac{w_i}{\|w_i\|_2} \right\|_2} \tag{2}$$

If our set of vectors $(w_i)_{i=1}^N$ is clustered in this manner, we can approximate MCSS for vector $q$ by simply finding the cluster $j$ that has the best cosine similarity with $q$, and doing a linear scan only on the items of this cluster:

$$j^* \;=\; \text{argmax}_j \, q^\top c_j$$
$$i^* \;=\; \text{argmax}_{i|a_i=j^*} \, q^\top w_i$$

Using this simple clustering, we have reduced our problem from $O(N)$ inner product calculations to $O(\sqrt{N})$ inner product calculations (supposing we have $\sqrt{N}$ clusters, each of which containing approximately $\sqrt{N}$ items).

**Hierarchical Clustering and Parallel Exploration** To deal with the loss of precision that comes with clustering, we can use more clusters (each cluster is then smaller), and do a linear search on the $p$ best matching clusters, instead of on the one best cluster. However in this case finding the best matching clusters becomes more expensive. To deal with this problem we can use a multi-layer hierarchical clustering, where the small clusters are grouped in bigger clusters, recursively until the top level which consists of a single cluster. When doing search in such a setup, the smallest-level clusters may be selected by a descent in the tree where at each level we keep a few of the best-matching clusters. This search is illustrated in Figure 6.

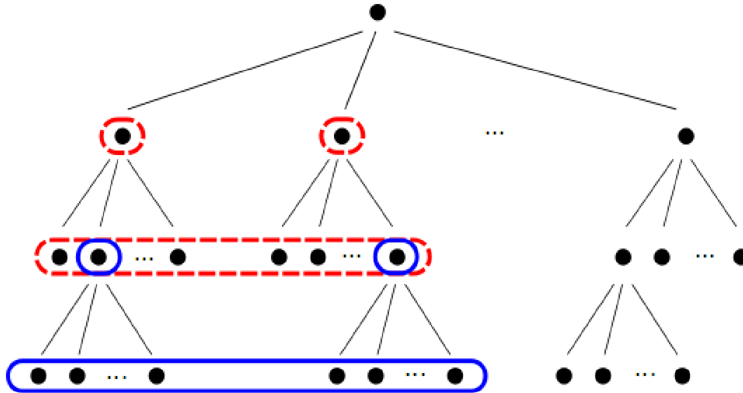

**Figure 6.** Parallel search in a clustering hierarchy

**Reduction to MCSS** The MIPS problem can easily be reduced to the MCSS problem by a simple transform introduced by [23]. This reduction consists of two steps: first, scaling all the vectors $w_i$ by the same constant such that $\max_i \|w_i\| = U < 1$. Second, applying the transform $P$ to all the vectors $w_i$, where:

$$P(x) = \left[x, \frac{1}{2} - \|x\|_2^2, \frac{1}{2} - \|x\|_2^4, ..., \frac{1}{2} - \|x\|_2^{2^m}\right]$$

We now define another function $Q$:

$$Q(x) = [x, 0, 0, ..., 0]$$

Searching for the MIPS solution for query vector q is approximately equivalent to finding the vector $w_i$ such that $\frac{P(w_i)^\top Q(q)}{\|P(w_i)\|_2 \|Q(q)\|_2}$ is maximal, i.e. to solving a MCSS problem. More details can be found in [23].

**Preliminary Results on a Simple Benchmark Task** To evaluate this method, I have constructed a toy task based on the word2vec embeddings [17]. In this task, the database is a subset of the word2vec embeddings made public by Google, and the query is one of the database vectors plus some Gaussian noise of varying intensity. The benchmark measures what proportion of the 100 vectors with the best inner product were found by the algorithm. I found that for similar candidate set sizes (the candidate set is the set of vectors on which we end up doing a linear search), the clustering approach vastly outperforms the hashing approaches from [22] and [23]. Results are presented in Figure 7. More details can be found in our paper [1].

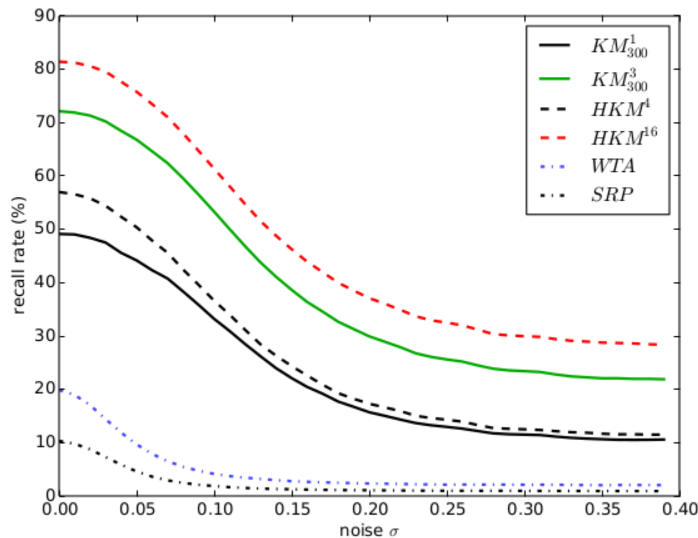


**Figure 7.** Results of the $k$-means approach (*KM*) and hierarchical $k$-means approach (*HKM*) in comparison to two hashing approaches (*WTA* and *SRP*).

## 3.3 Implementation Difficulties

**NMT Model Implementations** Two implementations of the NTM model with attention mechanism were implemented at the lab. The fist one was based on a framework named Groundhog, which was supposed to facilitate the implementation of neural nets with recurrent components. The code of this model is not well written, and it is difficult to replace any part of it for an experiment. The second model is created using the Blocks framework, which is a highly modular framework for all kinds of neural network architectures. This model is factorized in many classes, cleanly separating the encoder, the attention mechanism and the sequence generator. It is easy to replace the output softmax with an approximation based on $k$-means clustering. However the team working on this model had not yet been able to reproduce the results of the Groundhog model at the time of my internship, due to subtle differences in the semantics of the code (it would seem that the code has since then been fixed).

**Implementation of the Clustering Acceleration** I have adapted the Blocks model to use a clustering optimization on the output layer. This optimization consists of using only the words from a small number of clusters for the softmax of the output layer, effectively reducing the calculation time for both the forward and the backward calculation phases. The clusters' centroids, as well as the affectations of the word vectors to these clusters, are updated periodically in a phase separate from the main training phase.

**Unsatisfactory Results** The fact that the Blocks model's performances weren't as good as those of the Groundhog model was a big obstacle to the continuation of my work, as I had no easy way to benchmark the impact of my optimization against state-of-the-art results. However I have tried training a model with the clustered optimization, yielding even worse results (the model wasn't even able to correctly generate the beginning and end of sentence tokens that are always required in the output). The training phase was also not much accelerated because the cluster sizes became quickly very unbalanced, causing an overhead due to calculations on useless padding values. I have not had the time to solve these problems.

# 4 Other Internship Activities

## 4.1 Taxi Destination Prediction Challenge

With two other MILA students, Étienne Simon and Alexandre de Brébisson, we formed a group with the intention of entering a machine learning competition on the Kaggle platform. We wanted to learn more about deep learning and try to apply it to real-world situations. We finally selected the Taxi Destination Prediction Challenge[1] organized in the context of the ECML/PKDD 2015 conference, as this competition was just beginning at the time, and provided a challenging dataset with data structure quite different from the usual applications of deep learning.

**Taxi Destination Problem** The taxi destination prediction challenge consisted in predicting the destinations (latitude and longitude) of taxi trips based on initial partial trajectories and some meta-information associated with each ride. Such prediction models could help to dispatch taxis more efficiently. The dataset was composed of all the complete trajectories of 442 taxis running in the city of Porto (Portugal) for a complete year. The training dataset contained 1.7 million datapoints, each one representing a complete taxi ride and being composed of the following attributes :

- the complete taxi ride: a sequence of GPS positions (latitude and longitude) measured every 15 seconds. The last position represented the destination and different trajectories had different GPS sequence lengths.
- metadata associated to the taxi ride:
  - if the client called the taxi by phone, then we had a client ID. If the client called the taxi at a taxi stand, then we had a taxi stand ID. Otherwise we had no client identification,
  - the taxi ID,
  - the time of the beginning of the ride.

**Winning Solution** We managed to win the competition with a model based on a multi-layer perceptron. Our MLP model is trained by stochastic gradient descent on the training trajectories. The inputs of our MLP are the first 5 and last 5 positions of the known part of the trajectory, as well as embeddings for the context information (date, client and taxi identification). The embeddings are trained with SGD jointly with the MLP parameters. The MLP outputs probabilities for 3392 target points, and a mean is calculated to get a unique destination point as an output. We did no ensembling and did not use any external data. Our winning model is illustrated in Figure 8.

1. https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i

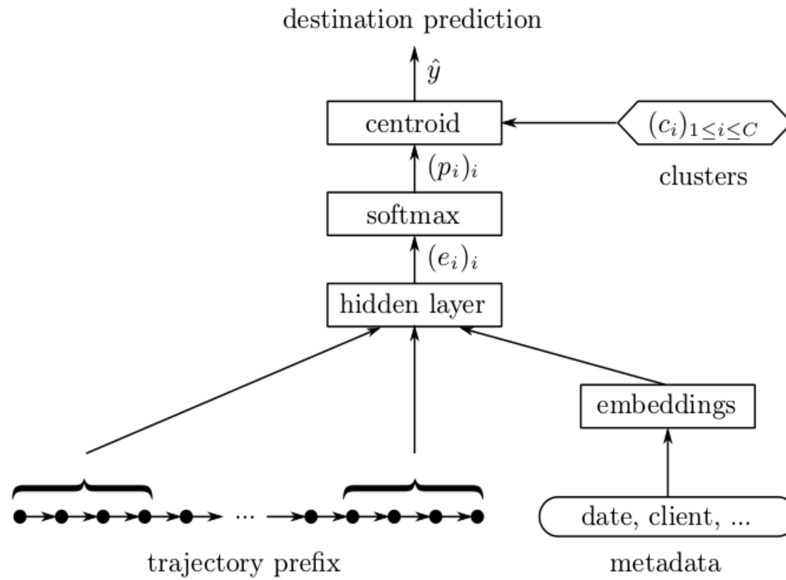


**Figure 8.** Architecture of our winning model for the taxi destination prediction challenge.

**Paper** As part of the normal process for the winner team, we submitted a paper [10] for the ECML/PKDD workshops that will be held in september 2015. This paper explains our winning solution in greater detail, and also explores other models which we did not have the time to explore during the challenge itself, but are more interesting since they use RNNs and provide better results. We suggest the interested reader consult our paper for more detailed information about the challenge and the way we solved it.

## 4.2 Transposing the Data Matrix

**Problem** During my internship I also worked on one of Pr. Bengio's ideas concerning datasets with very few examples, but where each example is a vector of very large dimensionality. Examples of such datasets are medical databases where we have only a small number of patients (a few hundred at most), but for each patient we have very detailed information about, for instance, gene activation in the cells. One typical dataset I used contained 200 examples, each of dimension 10 000. These datasets are typically very problematic for neural network approaches and are best exploited with traditionnal machine learning methods based on advanced feature selection. These problems are mainly caused by overfitting the test set (i.e. learning answers by heart), which is very easy since the number of parameters of a standard MLP model is much bigger than the number of examples.

**Approach** The approach consists in considering the data matrix in a "transposed" way. This means that the data that is input to the neural network is never a full example, but instead the network sees all the values for a particular dimension of the input and extracts information from this. More precisely, let $X$ be the data matrix, where a row of $X$ is an example. If $(r_j)$ are the columns of $X$, then $r_j$ is a representation of the $j$-th dimension of the input data. Intuitively, if two dimensions $j$ and $j'$ share similar representations (i.e. take similar values over all the examples), then we can think that they have similar roles in predicting the output. Using these kinds of models, the number of parameters of the model becomes independent of the dimensionality of the example space, and therefore overfitting is limited.

**MLP Model** Let $f$ be a MLP whose input dimension is $N+1$, where $N$ is the number of examples in the training set, and that outputs a single scalar value. We define:

$$\hat{y_j}(x) = f(r_j, x_j)$$

where $x$ is an example vector from any part of the dataset (training, validation, test) and $x_j$ is its $j$-th component, and $r_j$ is the column $j$ from the matrix $X$ of the *training set*. Supposing the problem is a simple binary classification problem, we do the following prediction for example $x$:

$$\begin{aligned}\hat{y}(x) &= \hat{\mathbb{P}}(y=1)\\ &= \sigma\left(\sum_j \hat{y_j}(x)\right)\\ &= \sigma\left(\sum_j f(r_j, x_j)\right)\end{aligned}$$

where $\sigma$ is the sigmoid function. Intuitively, for each component $x_j$ of $x$, we have a "vote" that says if this component should correspond to an output class of 1 or of 0. By summing these votes over all the dimensions of the input we get a global estimated probability value. Note that in this example, the parametric function $f$ that defines our model is the same over all dimensions $j$ of the input, therefore its number of parameters is independent of the size of the input. The only way it can discriminate between the various dimensions of the input is by using the representation $r_j$ of the dimensions it is currently predicting a value for. This enables the model to generalize over input dimensions and apply similar voting rules to dimensions that take similar values over examples.

**Two-stage MLP Model** In this model we have two MLPs: the first takes each dimension representation $r_j$ as input, and outputs a vector of dimension $d$. The second stage takes as input the example $x$, and passes it through a matrix multiplication whose coefficients are the output of the first MLP, possibly followed by a nonlinearity. The $d$ values obtained for example $x$ are then passed to another MLP that produces a single prediction. More formally:

$$\hat{y} = g\left(\underbrace{x}_{1\times n}\underbrace{\begin{pmatrix} f(r_1)\\ \vdots \\ f(r_n)\end{pmatrix}}_{n\times d}\right)$$

$f$ and $g$ are the two parametric functions that define the model. In this model the number of parameters is again independent of the input size $n$, as $f$ is a function from $\mathbb{R}^N$ to $\mathbb{R}^d$ (where $N$ is the number of examples), and $g$ is a function from $\mathbb{R}^d$ to $\mathbb{R}$.

**Training and Regularization** These models are trained by regular gradient descent. To make them work, it is necessary to add many kinds of regularization. The most efficient regularization schemes were $L_1$ regularization on the weight matrix defining the functions $f$ and $g$, but using dropout [14] at various places is also useful.

**Unsatisfactory results** The benchmark dataset I used for this task is the Arcene dataset from the 2003 NIPS feature selection challenge[2]. I wasn't able to reproduce the results of the challengers that had won the contest in 2003 using my method, even though I tried a very large range of model structures and hyperparameter combinations. This greatly demotivated me from the project, as I could see less and less why this approach should work in the first place. Eventually, I abandoned the project and was able to concentrate on other tasks.

## 5 Conclusion

During this internship I had the opportunity to learn and work on a great variety of domains, which has been very enriching. However I wasn't able to bring to a conclusion all the projects I worked on. In this section I will quickly summarize what has been done, and what remains to be done for all these projects.

**Learning About Deep Learning** By being in the lab, in contact with many other students and working on various projects, I have gained a good overview of the domain that is deep learning. I acquired a good understanding of the basic principles of neural networks and optimization techniques, and have been able to study in details many aspects such as recurrent neural nets,

2. http://clopinet.com/isabelle/Projects/NIPS2003/

optimization algorithms, regularization methods, practical GPU programming, etc. I have seen from afar a variety of applications of deep learning and am confident that I have acquired the knowledge and skill necessary to apply deep learning to many problems.

**Clustering Optimization Project** The project on optimizing the softmax layer of large vocabulary translation models was the project I worked on principally during my internship. I was able to extract interesting leads, and have produced a comprehensive documentation of the state of the problem as well as potential solutions, in the form of a paper. I was not able to finish the project as I didn't have enough time, and also because the code I was working on was not a clean basis for proper scientific experimentation. I hope other students will be able to take over my work and bring it to a more satisfactory conclusion. The paper I have submitted with Prof. Pascal Vincent for NIPS will most likely be rejected, as it was not of exceptional quality and did not provide very innovative ideas, but simply a combination of previously known techniques. I wish to help future MILA students to get to know the problem and continue exploring solutions, however I don't plan to continue working on it on my own.

**Data Transpose Problem** The problems I have encountered with the models explored in this project are mainly that I haven't been able to efficiently preventing the model from overfitting. Even though the number of parameters of the models has been reduced from $O(\sqrt{n})$ (size of the examples) to $O\big(\sqrt{N}\big)$ (number of examples), this number is still a few times bigger than the number of examples $N$, which is not yet sufficient to prevent overfitting. To continue work on the data transpose problem, one would have to explore more regularization techniques, which add more constraints to the model and make it behave as if it hadd fewer parameters. One would also have to do a more rigorous study of the impact of these techniques, by evaluating precisely the performance of standard machine learning models as well, which I have not done. I have written a quick summary of my findings, and will pass that on to any student willing to take over the project. I will also be available to help said students get to the point where I was, even though I don't intend to pursue this project anymore.

**Taxi Challenge** Winning the ECML/PKDD taxi destination prediction challenge was a great surprise to us, as we didn't think our approach could compete with those of machine learning experts. Winning had many good consequences, as it forced us to be more rigorous about our approach, leading us to explore various solutions that we had not had the time to develop during the challenge itself. I am very satisfied with the paper we wrote to explain our solution, and I hope it will awaken more interest in deep learning in the machine learning community. We will travel to Porto in September for the ECML/PKDD conference, where we will do a presentation of our work, which I very much look forward to.

## 6 Acknowlegments

I would like to thank Pr. Yoshua Bengio for having accepted me as an intern, trusting that I would bring something to the community, and for supervising my internship. I would like to thank Pr. Pascal Vincent for the time he has devoted supervising and guiding me in my project. I would like to thank all the lab members for contributing to a very positive spirit of friendship and non-competition.

I would like to thank the developers of Theano, Fuel and Blocks for developing such powerful tools. In particular, I would like to thank Frédéric Bastien and Pierre-Luc Carrier for their patience and for helpfully answering all of our technical questions.

I would like to thank the other interns with whom I spent much time at and outside the lab, sharing a piece of Montreal life, including Thomas Mesnard, Eloi Zablocki, Mélanie Ducoffe, Étienne Simon, and others. I would like to thank Étienne Simon and Alexandre de Brebisson for their continuous investment in the Kaggle competition and for the great work they have achieved.

I am grateful for the financial support provided by Samsung for my internship, as well as for all the support from various organisms such as Nvidia, Calcul Québec, Compute Canada, NSERC, and all the other organisms that have provided financial or material support to the MILA and helped make it such a productive place to work at.

# Bibliography

**[1]** Alex Auvolat and Pascal Vincent. Clustering is efficient for approximate maximum inner product search. *ArXiv preprint arXiv:1507.05910*, , 2015.

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

**[3]** Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley and Yoshua Bengio. Theano: new features and speed improvements. *ArXiv preprint arXiv:1211.5590*, , 2012.

**[4]** Yoshua Bengio, Réjean Ducharme, Pascal Vincent and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

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

**[6]** James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

**[7]** 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. *ArXiv preprint arXiv:1406.1078*, , 2014.

**[8]** Junyoung Chung, Caglar Gulcehre, KyungHyun Cho and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv preprint arXiv:1412.3555*, , 2014.

**[9]** Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

**[10]** Alexandre de Brébisson, Étienne Simon, Alex Auvolat, Pascal Vincent and Yoshua Bengio. Artificial neural networks applied to taxi destination prediction. *ArXiv preprint arXiv:1508.00021*, , 2015.

**[11]** Xavier Glorot, Antoine Bordes and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323. 2011.

**[12]** Alex Graves. Generating sequences with recurrent neural networks. *ArXiv preprint arXiv:1308.0850*, , 2013.

**[13]** Alex Graves, Santiago Fernández and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, pages 799–804. Springer, 2005.

**[14]** Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv preprint arXiv:1207.0580*, , 2012.

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

**[16]** Sébastien Jean, Kyunghyun Cho, Roland Memisevic and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *ArXiv preprint arXiv:1412.2007*, , 2014.

**[17]** Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119. 2013.

**[18]** Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Proceedings of the international workshop on artificial intelligence and statistics*, pages 246–252. Citeseer, 2005.

**[19]** Razvan Pascanu, Tomas Mikolov and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ArXiv preprint arXiv:1211.5063*, , 2012.

**[20]** Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

**[21]** David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

**[22]** Anshumali Shrivastava and Ping Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2321–2329. Curran Associates, Inc., 2014.

**[23]** Anshumali Shrivastava and Ping Li. Improved asymmetric locality sensitive hashing (ALSH) for maximum inner product search (MIPS). , arxiv:1410.5410, 2014.

**[24]** Sudheendra Vijayanarasimhan, Jonathon Shlens, Rajat Monga and Jay Yagnik. Deep networks with large output spaces. , arxiv:1412.7479, 2014.

**[25]** Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.