B.Sc.Eng. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Modern semantic analysis

for unsupervised classification of documents and sentences

Andreas Madsen (s123598)

Kongens Lyngby 2016

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Bygning 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
compute.dtu.dk

Summary (English)

This thesis investigates modern ways to construct a latent representation of documents. Two out of the three models takes the word ordering into account. Because of this they go beyond the old bag-of-word strategy which can't capture the semantic meaning embedded in the word ordering.

The intended application is to be able to cluster documents such that each cluster contains news articles about a single story. This is a much more sensitive clustering problem than topic modelling, which is a somewhat solved problem. It is because of this sensitivity that taking the word order into account is important.

The first two models are conceptually very simple, but has been shown to be very effective in understanding the semantic meaning of words [1, 2]. The last model is much more advanced and is based on a recent paper which have used a recurrent neural network to translate from English to French [3]. In this thesis the model is adapted to find latent representations for news articles.

These methods have many applications, in general finding latent representation of documents is a well known topic. The methods could for example extend to other data sources, such as patient diagnostics or drafts for legal acts. Once the stories have been clustered, more detailed question can be asked and analyzed. Such as which countries that show interest in a particular story, what kind of political perspective exists and does the amount of attention change over time.

Summary (Danish)

Denne afhandling undersøger moderne metoder til at konstruerer latente repræsentationer af dokumenter. To af de tre modeller tager rækkefølgen af ordene med i beregningen. De går dermed udover de kendte bag-of-word metoder som netop ikke fanger den semantiske mening, som ligger i ord rækkefølgen.

Den tiltænkte applikation er at være i stand til at grupper dokumenter, således at hver gruppe indeholder artikler for en enkelt historie. Dette er et meget mere følsomt problem end "topic modelling" som er et forholdsvis løst problem. Det er på grund af denne følsomhed at ord rækkefølgen er vigtig.

De første to modeller er konceptuelt meget simple men har vist sig at være meget effektive til at forstå den semantiske betydning af ord [1, 2]. Den sidste model er væsentlig mere avanceret og er baseret på en ny videnskabelig artikel, som bruger et recurrent neural network til at oversætte fra engelsk til fransk [3]. I denne afhandling er modellen blevet modificeret til at finde latente repræsentationer for nyhedsartikler.

Disse metoder har mange applikationer. Generelt er det at finde latente repræsentationer af dokumenter et velkendt forskningsområde. Metoderne kan for eksempel bruges på andre data kilder, som patient diagnoser eller udkasts til lov ændringer. Når historierne er blevet grupperet, kan mere detaljeret spørgsmål blive undersøgt. For eksempel om der er lande som viser særlig interesse i en bestemt historie, hvilke slags politiske perspektiver som findes og hvordan opmærksomheden ændre sig over tid.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a B.Sc.Eng. degree in Mathematics. The work was carried out in the period from January 2015 to June 2015.

Kongens Lyngby – June 15, 2015

Missing Signature

Andreas Madsen (s123598)

Acknowledgements

First and foremost I would like to thank my supervisor Ole Winther for his guidance and support and his Ph.d. student Søren Sønderby for answering questions related to the Sutskever model [3, p. 1] and Theano [4, 5]. I would also like to thank Stephan Gouws who presented the word2vec framework [1, 6] at KU, which initially inspired me to do this thesis. Also thanks to Alex Graves for writing an excellent book on Recurrent Neural Network [7], I'm quite sure I wouldn't have understood the subject as well as I do now without it.

Contents

CHAPTER 1

Introduction

When a news story is covered by a single newspaper, it is rare that it gives a full overview of the situation and enlighten all perspectives. Thus people with specific interest or general curiosity may want to research the story further. However it can be difficult to find related articles, for example because the different perspective results in a much different title. The reader may also want to avoid doing detailed reading of unrelated articles or articles with the same perspective. Thus a search system designed to find articles related to the same story, could give a better experience to the reader.

Such a search system could cluster the articles, such that each cluster contains all articles related to a single story. In order for this to work, each article would have to be represented with some numbers (a vector), the vectors would be calculated such that articles about the same story have approximately the same vectors. A clustering algorithm would then be able to use these vectors to find articles about the same story. It is the task of calculating these vector representations of articles, that is the main focus in this thesis.

The methods for finding the vector representations should extend to other data sources, such as patient diagnostics reports or drafts for legal acts. Once the individual stories have been isolated, more detailed question can be asked and analyzed. Such as which counties there show interest in a particular story, what kind of political perspective exists and does the amount of attention change over time.

Finding vector representations of documents is a well known subject and is probably the most common subject in the field of natural language processing. However many of the current methods, are based on the bag-of-word method for initially representing a document. Other methods such as LSA or LDA then transform this bag-of-word representation into the more useful document vectors.

The bag-of-word method works by counting how many times all words in a predefined vocabulary appears in each document. The result is a long vector, that will have many zeros as many words will not be used at all in the document. The problem with this strategy is that it ignores the order of words. For example “China declares war on Japan” is has a different meaning than “Japan declares war on China”, but because the words appear an equal amount of times the bag-of-word representation of the sentences is the same.

Fortunately a new set of models is emerging, these models don’t use the bag-of-word

method to represent documents, but instead scans over the words in the document. Because of this they are able to take the word order into account when creating the vector representations. It is some of these models that is discussed and compared in this thesis.

The first model is called *skip-gram* and creates vector representations of the individual words. It has been shown to be very good at capturing the semantic meaning of words, such that algebraic expressions like $man - king + women = queen$ holds [1, 6]. Unfortunately it provides no direct way of obtaining a vector for the entire document. To solve this the *paragraph2vec* model has been invented, it works in a similar way as the *skip-gram* model but produces simultaneously vector representation of documents.

These two models are conceptually very simple and limited to find log-linear patterns. A much more advanced method is the Sutskever model. The original paper [3] used it to provide state of the art text translation from English to French. As a side effect of this it also produces vector representations that contains the non-language specific meaning of the translated document. In this thesis that method is adapted to find vector representations of news articles. As translation data doesn't exists for the news article use case, the model is trained to predict the title given the article lead.

CHAPTER 2

Theory

2.1 Feed forward Neural Network

The models presented in this thesis are all neural networks or can be interpreted as such. To give a short introduction to neural networks in general, the feed forward neural network (FFNN) is used. The feed forward neural network is one of the simplest models in the family of neural networks. The other models used in this thesis can be considered as extensions of the FFNN. Exceptions to this is perhaps the skip-gram and paragraph2vec models which are simplified versions of FFNN.

2.1.1 The neuron

The neuron is the main construct in any neural network. Mathematically it is just a function which takes a vector and returns a scalar. It does this by a weighted sum of the inputs $\{x_i\}_{i=1}^{I}$:

$$a = \sum_{i=1}^{I} w_i x_i \tag{2.1.1}$$

This sum is then typically transformed using a non-linear function θ:

$$b = \theta(a) \tag{2.1.2}$$

The value b is the output of the neuron and is called the *activation*. Typically the sigmoid function $\sigma(\cdot)$ or hyperbolic tangent $\tanh(\cdot)$ function is used as non-linear function θ [8]. If no non-linear function is applied then the identity function is used, $\theta(a) = a$.

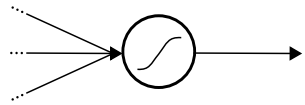


Figure 2.1.1: Visual representation of a single neuron. The left arrows represents the input elements $\{x_i\}_{i=1}^{I}$. The circle represent the function that returns the scalar b (right arrow).

2.1.2 The network

A neural network is a multivariate non-linear regression model constructed by combining neurons, but the network can be slightly expanded to become a multiclass classifier. This is done by using the softmax function [9], such each output value is the class probability $y_k = P(C_k|x)$.

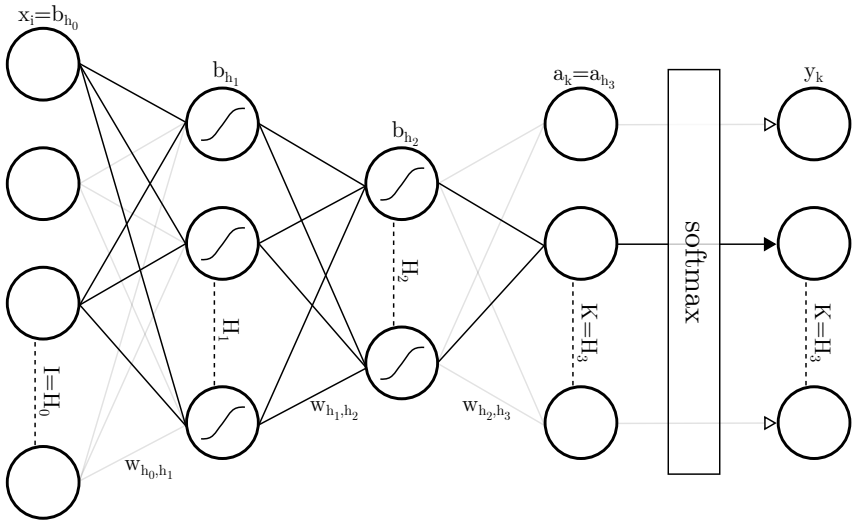


Figure 2.1.2: Visual representation of a neural network with two hidden layers. Some lines are less visible, this is just a visual aid because the many lines can be difficult to look at.

The neural network in Figure 2.1.2, have one input layer $x_i, i \in [1, I]$ and one output layer $y_k, k \in [1, K]$. This is the same for any neural network, what varies is the type of network (here a feed forward neural network) and the amount of hidden layers (here two). The hidden layers are those layers there can contain non-linear functions. If there are no hidden layers, the neural network i just a multivariate linear regressor or a multiclass logistic regressor if the softmax is applied [8].

Because there are more than one neuron in each layer and many layers, the neuron index and layer index is denoted by the subscript. For example a neuron output is denoted by b_{h_ℓ} for $h_\ell \in [1, H_\ell]$, where h_ℓ is the neuron index and H_ℓ is the amount of neurons in layer ℓ.

2.1.3 Forward pass

Calculation of the network output (y_k) is sometimes called the *forward pass*, while the parameter estimation is sometimes called the *backward pass*.

First the activation for the first layer is calculated:

$$b_{h_1} = \theta(a_{h_1}), \quad a_{h_1} = \sum_{i=1}^{I} w_{i,h_1} x_i, \quad \forall h_1 \in [1, H_1] \tag{2.1.3}$$

The activation for the second layer is almost identical:

$$b_{h_2} = \theta(a_{h_2}), \quad a_{h_2} = \sum_{h_1=1}^{H_1} w_{h_1,h_2} b_{h_1}, \quad \forall h_2 \in [1, H_2] \tag{2.1.4}$$

In fact by letting $x_i = b_{h_0}$ the activation can be generalized to any amount of hidden layers (L):

$$\begin{aligned} b_{h_\ell} &= \theta(a_{h_\ell}), && \forall h_\ell \in [1, H_\ell] \wedge \ell \in [1, L] \\ a_{h_\ell} &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} b_{h_{\ell-1}}, && \forall \ell \in [1, L+1] \text{ where: } b_{h_0} = x_i, H_0 = I \end{aligned} \tag{2.1.5}$$

Note that the last layer $\ell = L + 1$ does not use the non-linear θ function. At last the network output y_k can be calculated using the softmax function on the $a_k = a_{L+1}$ values:

$$y_k = \frac{\exp(a_k)}{\sum_{k'=1}^{K} \exp(a_{k'})}, \quad \forall k \in [1, K] \text{ where: } a_k = a_{h_{L+1}}, K = H_{L+1} \tag{2.1.6}$$

2.1.4 Loss function

Optimization of the parameters $w_{i,j}$ requires definition of a loss function. For classification it makes sense to maximize the joint probability of observing all the observations:

$$P(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{n=1}^{N} P(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) = \prod_{n=1}^{N} \prod_{k=1}^{K} P(C_{n,k}|\mathbf{x}_n, \mathbf{w})^{t_{n,k}} \tag{2.1.7}$$

Here $\mathbf{x}_n$ is the input vector for observation n, with a corresponding label vector $\mathbf{t}_n$. The label vector is an indicator vector, constructed using 1-of-K encoding. For example if there are 5 classes and class 2 is the correct class for observation n, then $\mathbf{t}_n = [0, 1, 0, 0, 0]^T$. The class probability $P(C_{n,k}|\mathbf{x}_n, \mathbf{w})$ is in terms of the forward pass equations the y_k value for observation n.

The logarithm is then used to create linearity and avoid numerical floating point issues. The sign is also changed such that it becomes a loss function:

$$-\ln\left(P(\mathbf{t}|\mathbf{x}, \mathbf{w})\right) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{n,k} \ln\left(P(C_{n,k}|\mathbf{x}_n, \mathbf{w})\right) \tag{2.1.8}$$

Because of the linearity in (2.1.8) it makes sense to just consider the loss function for one datapoint. This way the n index can be omitted from the equations and the sum over n can always be reintroduced in the end. By doing this one gets the final loss function there will be denoted by $\mathcal{L}$:

$$\mathcal{L} = -\sum_{k=1}^{K} t_k \ln\left(P(C_k|\mathbf{x}, \mathbf{w})\right) = -\sum_{k=1}^{K} t_k \ln(y_k) \tag{2.1.9}$$

2.1.5 Backward pass

For the neural network there is no closed form solution to optimizing the loss function. Instead an iterative algorithm called *gradient descent* is used. Gradient descent uses the derivatives of the loss function with respect to the parameters to iteratively optimize the parameters. This approach will be discussed in details later, for now the important part is to find a method of calculating the derivatives.

For calculating the derivatives the *error backpropagation* algorithm is used. The name varies a bit depending on the neural network architecture, thus the term *backward pass* will be used as the umbrella term.

The neural network from earlier in Figure 2.1.2 will be used to inspire the general algorithm. The overall goal is to calculate:

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}}, \quad \forall \ell \in [1, L+1] \tag{2.1.10}$$

First consider the problem in (2.1.10) for the layer $\ell = 1$. The loss function depends on some hidden input from the first hidden layer a_{h_1}, thus the chain rule can be used:

$$\frac{\partial \mathcal{L}}{\partial w_{h_0,h_1}} = \frac{\partial \mathcal{L}}{\partial a_{h_1}} \frac{\partial a_{h_1}}{\partial w_{h_0,h_1}} = \frac{\partial \mathcal{L}}{\partial a_{h_1}} x_{h_0} \tag{2.1.11}$$

It turns out that it is smart to define $\frac{\partial \mathcal{L}}{a_{h_1}}$ in general for all layers, this is just for bookkeeping, there makes it easier to implement.

$$\delta_{h_\ell} \overset{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}} \tag{2.1.12}$$

Using this (2.1.11) becomes:

$$\frac{\partial \mathcal{L}}{\partial w_{h_0,h_1}} = \delta_{h_1} x_{h_0} \tag{2.1.13}$$

The question "how should δ_{h_1} be calculated ?" then arises. This is done by using the chain rule again. This time the chain rule gives a sum because a_{h_1} affects multiply values.

$$\delta_{h_1} = \frac{\partial \mathcal{L}}{\partial a_{h_1}} = \sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}} \frac{\partial a_{h_2}}{\partial a_{h_1}} \tag{2.1.14}$$

A chain rule is then again used, this time for expanding $\frac{\partial a_{h_2}}{\partial a_{h_1}}$:

$$\begin{aligned} \delta_{h_1} = \frac{\partial \mathcal{L}}{\partial a_{h_1}} &= \sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}} \frac{\partial a_{h_2}}{\partial b_{h_1}} \frac{\partial b_{h_1}}{\partial a_{h_1}} = \frac{\partial b_{h_1}}{\partial a_{h_1}} \sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}} \frac{\partial a_{h_2}}{\partial b_{h_1}} \\ &= \theta'(a_{h_1}) \sum_{h_2=1}^{H_2} \delta_{h_2} w_{h_1,h_2} \end{aligned} \tag{2.1.15}$$

Calculating δ_{h_2} is almost identical:

$$\begin{aligned} \delta_{h_2} = \frac{\partial \mathcal{L}}{\partial a_{h_2}} &= \sum_{h_3=1}^{H_3} \frac{\partial \mathcal{L}}{\partial a_{h_3}} \frac{\partial a_{h_3}}{\partial b_{h_2}} \frac{\partial b_{h_2}}{\partial a_{h_2}} = \frac{\partial b_{h_2}}{\partial a_{h_2}} \sum_{h_3=1}^{H_3} \frac{\partial \mathcal{L}}{\partial a_{h_3}} \frac{\partial a_{h_3}}{\partial b_{h_2}} \\ &= \theta'(a_{h_2}) \sum_{h_3=1}^{H_3} \delta_{h_3} w_{h_2,h_3} \end{aligned} \tag{2.1.16}$$

Calculating δ_{h_3} will be a bit different, because it is the last δ_{h_ℓ} there needs calculation (for a neural network with two hidden layer). It thus makes sense to first generalize (2.1.15) and (2.1.16) such that it works for any number of hidden layers:

$$\delta_{h_\ell} = \theta'(a_{h_\ell}) \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \delta_{h_{\ell+1}} w_{h_\ell,h_{\ell+1}}, \quad \forall \ell \in [1, L] \tag{2.1.17}$$

The last delta δ_{h_3} or in general $\delta_{h_{L+1}}$ is also calculated by using a chain rule. $a_{h_{L+1}}$ affects all y_k values thus again one gets a sum:

$$\delta_{h_{L+1}} = \delta_k = \frac{\partial \mathcal{L}}{\partial a_k} = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k} \tag{2.1.18}$$

The first derivative $\frac{\partial \mathcal{L}}{\partial y_{k'}}$ can be derived from (2.1.9):

$$\frac{\partial \mathcal{L}}{\partial y_{k'}} = \frac{\partial}{\partial y_{k'}} \left(- \sum_{k''=1}^{K} t_{k''} \ln(y_{k''}) \right) = -\frac{t_{k'}}{y_{k'}} \tag{2.1.19}$$

The other derivative $\frac{\partial y_{k'}}{\partial a_k}$ can be derived from (2.1.6):

$$\begin{aligned}\frac{\partial y_{k'}}{\partial a_k} &= \frac{\partial}{\partial a_k}\frac{\exp(a_{k'})}{\sum_{k''=1}^{K}\exp(a_{k''})} \\ &= \frac{\frac{\partial}{\partial a_k}\exp(a_{k'})}{\sum_{k''=1}^{K}\exp(a_{k''})} - \frac{\exp(a_{k'})\frac{\partial}{\partial a_k}\sum_{k''=1}^{K}\exp(a_{k''})}{\left(\sum_{k''=1}^{K}\exp(a_{k''})\right)^2} \\ &= \frac{\frac{\partial}{\partial a_k}\exp(a_{k'})}{\sum_{k''=1}^{K}\exp(a_{k''})} - \frac{\exp(a_{k'})}{\sum_{k''=1}^{K}\exp(a_{k''})}\frac{\frac{\partial}{\partial a_k}\sum_{k''=1}^{K}\exp(a_{k''})}{\sum_{k''=1}^{K}\exp(a_{k''})}\end{aligned} \tag{2.1.20}$$

Because of the difference in index, the first term is only not zero when $k = k'$, in which case y_k is the derivative. It thus becomes useful to define:

$$\delta_{i,j} = \begin{cases} 1 & \text{when } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.1.21}$$

Similarly in the second term $\frac{\partial}{\partial a_k}\exp(a_{k''})$ is zero except in the case where $k = k''$:

$$\frac{\partial y_{k'}}{\partial a_k} = \delta_{k,k'}y_k - y_{k'}y_k \tag{2.1.22}$$

The result from (2.1.19) and (2.1.22) is then combined into (2.1.18):

$$\begin{aligned}\delta_{h_{L+1}} = \delta_k &= \sum_{k'=1}^{K} -\frac{t_{k'}}{y_{k'}}\left(\delta_{k,k'}y_k - y_{k'}y_k\right) = \sum_{k'=1}^{K} -\frac{t_{k'}}{y_{k'}}\delta_{k,k'}y_k + \sum_{k'=1}^{K}\frac{t_{k'}}{y_{k'}}y_{k'}y_k \\ &= -\frac{t_k}{y_k}y_k + y_k\sum_{k'=1}^{K}t_{k'} = -t_k + y_k = y_k - t_k\end{aligned} \tag{2.1.23}$$

To get $\sum_{k'=1}^{K}t_{k'}$ it's used that $\{t_{k'}\}_{k'=1}^{K}$ is constructed using 1-of-K encoding, thus only one element is 1 and the remaining is 0, the sum must therefore give 1.

Using (2.1.23) and (2.1.17) all δ_{h_ℓ} for $\ell \in [1, L+1]$ can be calculated for a feed forward neural network with L layers. Note how (2.1.23) is an error measure and this value is propagated back through the network by the δ_{h_ℓ} equations in (2.1.17). This is why the method is called *error backpropagation*.

Given δ_{h_1} the weight derivative w_{h_0,h_1} can be calculated using (2.1.13). However the weights w_{h_0,h_1} are not the only parameters. Luckily the equations for the other weights $w_{h_{\ell-1},h_\ell}$ are not much different.

$$\frac{\partial\mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} = \frac{\partial\mathcal{L}}{\partial a_{h_\ell}}\frac{\partial a_{h_\ell}}{w_{h_{\ell-1},h_\ell}} = \delta_{h_\ell}b_{h_{\ell-1}}, \quad \forall\ell \in [1, L+1] \text{ where: } b_{h_0} = x_i \tag{2.1.24}$$

2.2 Gradient descent optimization

A neural network have no closed form solution for optimizing the parameters $w_{i,j}$. Thus the weights must be optimized using an iterative algorithm. Gradient descent is the most common algorithm for optimizing neural networks, though many variations exists.

The basic principal is that given a loss function $\mathcal{L}$, the parameters $w_{i,j}$ can be optimized by going in the opposite direction of the gradient $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$. Thus each iteration n is given by:

$$w_{i,j}^n = w_{i,j}^{n-1} + \Delta w_{i,j}^n, \quad \Delta w_{i,j}^n = -\eta_n \frac{\partial \mathcal{L}}{\partial w_{i,j}} \tag{2.2.1}$$

η_n is the *learning rate* or *step size* and should be so small that it doesn't skip over the wanted minima, but also not so small that the weights doesn't converge within reasonable time. Typically the initial weights $w_{i,j}^0$ is given by a random distribution. The choice of the learning rate and initialization are not obvious, as it typically depends on the specific architecture of the network and the dataset. For example Alex Graves [7] suggests using a uniform distribution in the range $[-0.1, 0.1]$ or a normal distribution with $\sigma = 0.1$ and a constant step size $\eta_n = 10^{-5}$.

2.2.1 Momentum

A major problem with the naïve gradient descent algorithm, is that it tends to get stuck in a local minima because $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$ is 0 in such minima. To avoid this problem a *momentum* can be added to the gradient descent equation:

$$\Delta w_{i,j}^n = m \Delta w_{i,j}^{n-1} - \eta_n \frac{\partial \mathcal{L}}{\partial w_{i,j}} \tag{2.2.2}$$

Here m is the *momentum* parameter. Alex Graves suggests using $m = 0.9$ [7], but again this depends on the architecture and dataset.

2.2.2 Stochastic gradient descent

For calculating the gradient some dataset is used, the naïve approach is to use the entire dataset at once. However it turns out that a better approach is to calculate the gradient using a single data point, then updating the parameters and repeating for all datapoints (see Algorithm 2.2.1). This method is called *stochastic gradient descent* (SGD) or *online learning*, where the naïve approach is called *batch learning*.

One reason for *stochastic gradient descent* being better, is that it handles redundancies in the data better. It also helps in not getting stuck in a local minima, because it

while *stopping criteria not met* **do**
    Randomize training set order
    for *each observation in the training set* **do**
        Run forward and backward pass to calculate the gradient
        Update weights with gradient descent algorithm
    end
end

Algorithm 2.2.1: Stochastic gradient descent [7].

is unlikely that what is a local minima for one data point is also a local minima for another data point [8].

2.2.3 Mini-batch stochastic gradient descent

While *online learning* is way better than *batch learning* it doesn't handle outliers very well, because the gradient will be big for such observation. It also isn't very computational efficient because the vectorization libraries can't take full advantage of the CPU cache and etc..

To solve those issues *mini-batch stochastic gradient descent* also called *mini-batch learning* is used. This is a compromise between *batch learning* and *online learning*, where small random batches of observations are created and the weights are then optimized like in Algorithm 2.2.1. This introduces a third parameter called the *mini-batch-size*, which as the same suggests controls the size of the mini-batches. This also generalizes *online learning* and *batch learning*, because if *mini-batch-size = 1* then it is *online learning* and if *mini-batch-size = training-size* then it is *batch learning*. Generally the *mini-batch-size* is just chosen by benchmarking, calculating or guessing what is the most computational efficient choice.

while *stopping criteria not met* **do**
    Randomize training set order
    for *each mini-batch in the training set* **do**
        Run forward and backward pass to calculate the gradient
        Update weights with gradient descent algorithm
    end
end

Algorithm 2.2.2: Mini-batch gradient descent.

2.2.4 RMSprop

The gradient descent algorithms presented so far uses a global learning rate for all the weights. This causes some issues for those gradients there are very tiny or huge, because they will either be dominating or be practically neglected. A solution to this is to not use the size of the gradient, but just the sign and then move a constant amount in that direction, this method is called *rprop* and is useful for *batch-learning*. However for *mini-batch learning rprop* doesn't work very well. This is because *mini-batch-size* have some averaging properties. For example if the gradient for a specific weight is 0.1 for nine mini-batches and -0.9 for just one mini-batch, this will cause the weight to stay roughly where it is. However *rprop* will move 9×1 in one direction and then $1 \times (-1)$ in the other, because a constant step size is used. The solution is called *RMSprop* and is the equivalent to *rprop* for *mini-batch learning*. It works by performing a moving average on the gradient and then using this average to scale the learning rate, such that it adapts to the scale of the individual gradient in general. This can be done with and without momentum, for completeness with momentum is shown here:

$$
\begin{aligned}
f_{i,j}^n &= \gamma f_{i,j}^{n-1} + (1-\gamma)\frac{\partial \mathcal{L}}{\partial w_{i,j}} \\
g_{i,j}^n &= \gamma g_{i,j}^{n-1} + (1-\gamma)\left(\frac{\partial \mathcal{L}}{\partial w_{i,j}}\right)^2 \\
\Delta w_{i,j}^n &= m\Delta w_{i,j}^{n-1} - \frac{\eta}{\sqrt{g_{i,j}^n + \left(f_{i,j}^n\right)^2 + \epsilon}}\frac{\partial \mathcal{L}}{\partial w_{i,j}}
\end{aligned}
$$

Equation 2.2.3: RMSprop with momentum, as Alex Graves does it in [10]. In that paper $\gamma = 0.95, m = 0.9, \eta = 0.0001, \epsilon = 0.0001$ was used.

Notice that ϵ is necessary because $g_{i,j}^n$ and $f_{i,j}^n$ can be zero or near-zero which would otherwise cause the faction to explode.

2.2.5 Clipping

Sometimes the $\frac{\partial \mathcal{L}}{\partial w_{i,j}}$ gradients becomes really big because of an unfortunate mini-batch or some numerical instability in the implementation. To avoid this the gradient is often clipped or squeezed to prevent issues [10]. There are 3 common methods, they use a *clip* hyperparameter to control how aggressive the clipping should be.

Note that some of these methods consider all gradients for all the weight matrices as

one long vector, this vector will be denoted $\nabla\mathcal{L}$.

$$\text{clipping: } \frac{\partial\mathcal{L}}{\partial w_{i,j}} = \max\left(\min\left(\frac{\partial\mathcal{L}}{\partial w_{i,j}}, clip\right), -clip\right) \tag{2.2.4}$$

$$\text{max squeezing: } \frac{\partial\mathcal{L}}{\partial w_{i,j}} = \begin{cases} \frac{\partial\mathcal{L}}{\partial w_{i,j}} & clip < ||\nabla\mathcal{L}||_\infty \\ \frac{\partial\mathcal{L}}{\partial w_{i,j}} \frac{clip}{||\nabla\mathcal{L}||_\infty} & otherwise \end{cases} \tag{2.2.5}$$

$$\text{euclidean squeezing: } \frac{\partial\mathcal{L}}{\partial w_{i,j}} = \begin{cases} \frac{\partial\mathcal{L}}{\partial w_{i,j}} & clip < ||\nabla\mathcal{L}||_2 \\ \frac{\partial\mathcal{L}}{\partial w_{i,j}} \frac{clip}{||\nabla\mathcal{L}||_2} & otherwise \end{cases} \tag{2.2.6}$$

2.2.6 Early stopping

Optimization of the loss function $\mathcal{L}$ using just *batch learning* without momentum, is given to be non-increasing for each iteration [8]. Thus it is quite easy to find a minima for the training data, however this does not directly reflect the performance of the model given new data. In Figure 2.2.1 the loss function for the training data (green) is forever decreasing. The loss function for data not used in the training (blue) is only decreasing for the first 15 iterations, after this the performance of the model is getting worse. This is quite common and is called overfitting, an simple way to prevent this is to use *early stopping*.

Early stopping uses a third dataset, this dataset can not be used for calculating the gradients (training data) or for evaluating the model performance (test data). It is used for evaluating the loss function for each iteration and stop the optimization algorithm when the loss function haven't decreased for some iterations [7–9].

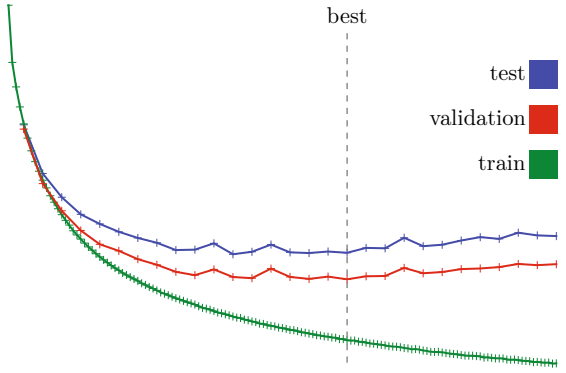


Figure 2.2.1: Loss function for each iteration. The validation dataset indicates when to stop (the best line). Data is from [7].

There are other methods to prevent overfitting such as regularization [8, 9]. However only *early stopping* was used in the Sutskever paper [3].

2.3 Skip-gram

The Skip-gram model tries to learn the meaning of words by its context. Specifically it tries to predict the surrounding words given a single input word.

corpus: collection of documents

The Skip-gram model does this by encoding the input word using 1-of-V encoding where V is the vocabulary size. This means that each word is represented as an indicator vector, where the vector represent a single word from a given vocabulary. The vocabulary is typically defined using the V most common word in the text corpus.

$$\begin{aligned} \text{twinkle} &: [1, 0, 0, 0, \cdots] \\ \text{twinkle} &: [1, 0, 0, 0, \cdots] \\ \text{little} &: [0, 1, 0, 0, \cdots] \\ \text{star} &: [0, 0, 1, 0, \cdots] \end{aligned}$$

Figure 2.3.1: Example of 1-of-V encoding on the sentence "twinkle twinkle little star"

Each of these word representations are then transformed into a lower dimensional space (latent representation). In this space, words with same meaning should be close to each other. The position of each word should also contain a higher semantic meaning, such that for example $f(King) - f(Man) + f(Female) \approx f(Queen)$ [6]. Here f is the transformation function from the input space to the latent representation.

The model is trained such that the surrounding words can be predicted from this latent representation. See Figure 2.3.2 for a graphical overview. An important detail about this model, is that the order of the words within the considered text window have no effect on the training. Thus all surrounding are assumed to having approximately the same meaning.

2.3.1 The likelihood function

The Skip-gram model is optimized by maximum-likelihood. Given a corpus with $\mathcal{D}$ documents where each document have T_d words, the likelihood function is calculated using the conditional probability of observing the surrounding words ($w_{d,t+\ell}$) given the input words ($w_{d,t}$, $t \in [1, T_d]$).

$$\prod_{d=1}^{\mathcal{D}} \prod_{t=1}^{T_d} \prod_{\ell} p(w_{d,t+\ell} | w_{d,t}) \tag{2.3.1}$$

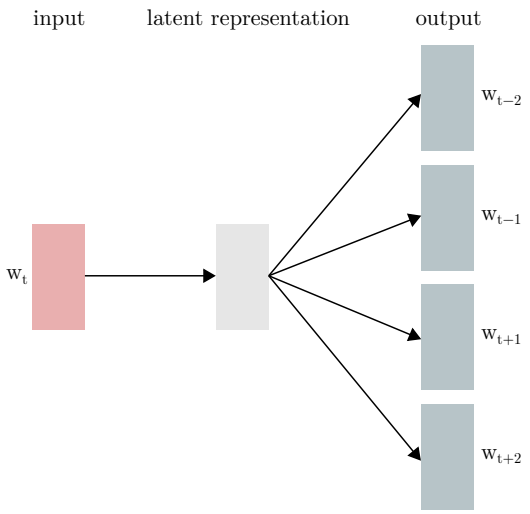


Figure 2.3.2: Visualization of the Skip-gram model. Given a single input word it tries to predict the surrounding words. From this the latent representation is learned.

The details about which document there currently is used, are typically omitted from the expressions. Thus equation (2.3.1) becomes:

$$\prod_{t=1}^{T}\prod_{\ell} p(w_{t+\ell}|w_t) \tag{2.3.2}$$

Since words closer to w_t are expected to be more related to that word, the near surrounding words are weighted higher. This is not modelled by explicit weights, but instead by letting $\ell \in [-R, R] \setminus \{0\}$ where $R \sim U[1, C]$ [6].

As usual the negative log is used to get the loss function:

$$\mathcal{L} = -\sum_{t=1}^{T}\sum_{\ell} \ln(p(w_{t+\ell}|w_t)) \tag{2.3.3}$$

In [1] a likelihood function is used (sign is flipped) and it is averaged over time ($1/T$ is multiplied).

2.3.2 Forward pass

The conditional probability $p(w_{t+\ell}|w_t)$ is calculated using a log-linear model. This is best explained by considering a neural network, with a single hidden layer and no non-linear units ($\theta(a) = a$).

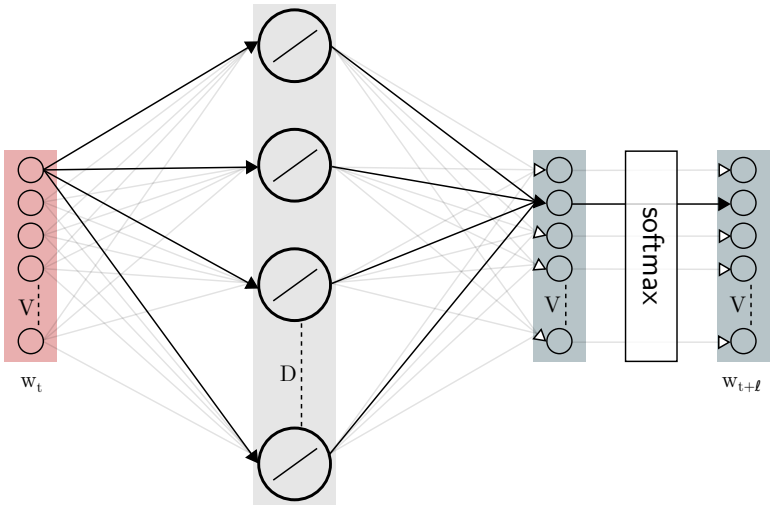


Figure 2.3.3: Visualization of a single pass though the Skip-gram network.

As seen in Figure 2.3.3 the Skip-gram model is really just a very simple feed forward neural network. The forward equations for this kind of network have already been covered in section 2.1, thus deriving the forward equation for the Skip-gram model is just a simple exercise of using that theory.

By setting $h_0 = K = V$, $h_1 = D$ and defining $\{x_i^t\}_{i=1}^V$ as the indicator vector for the word w_t, the forward pass becomes:

$$\begin{aligned} b_{h_1}^t = a_{h_1}^t &= \sum_{i=1}^{V} w_{i,h_1} x_i^t, && \forall h_1 \in [1, D] \\ a_k^t = a_{h_2}^t &= \sum_{h_1=1}^{D} w_{h_1,h_2} b_{h_1}^t, && \forall h_2 \in [1, V] \\ y_k^t &= \frac{\exp(a_k^t)}{\sum_{k'=1}^{V} \exp(a_{k'}^t)}, && \forall k \in [1, V] \end{aligned} \tag{2.3.4}$$

It is possible to reformulate the (2.3.4) equations, such that they appear as in [1]. However it is a bit convoluted as it requires using vector notation and using a trick where subscript selection is done by knowing that x^t is an indicator vector. Thus that won't be covered here, but see Appendix A for how it can be done.

Calculating y_k^t is problematic since it contains a sum over all words in the vocabulary. In the Skip-gram model the vocabulary can be of size 10^5 to 10^7 [1]. This is why [1, 6, 11] uses an approximation in form of either *hierarchical softmax* or *negative-sampling*.

These approximations will not be discussed here, but are worth considering as they will reduce the computational complexity from $\mathcal{O}(V)$ to $\mathcal{O}(\ln_2(V))$ [6].

2.3.3 Backward pass

Deriving the backward pass have similarly been covered in section 2.1. First step is to express $\ln(p(w_{t+\ell}|w_t))$ using a target vector, this vector will be denoted $t^{t+\ell}$ for the word $w_{t+\ell}$:

$$\mathcal{L} = -\sum_{t=1}^{T}\sum_{\ell}\ln(p(w_{t+\ell}|w_t)) = -\sum_{t=1}^{T}\sum_{\ell}\sum_{k=1}^{V} t_k^{t+\ell}\ln(y_k^t) \tag{2.3.5}$$

The loss function is a little difference compared to that in section 2.1. However because it maintains a linear relation to the loss function used in section 2.1, this doesn't matter.

$$\frac{\partial\mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} = \sum_{t=1}^{T}\sum_{\ell}\frac{\partial}{\partial w_{h_{\ell-1},h_\ell}}\left(-\sum_{k=1}^{V} t_k^{t+\ell}\ln(y_k^t)\right), \quad \forall\ell\in[1,L+1] \tag{2.3.6}$$

Now it is just a matters of using the method covered in section 2.1. First the δ_{h_ℓ} values are calculated. To denote the time and lag a superscript is added:

$$\begin{aligned} \delta_{h_1}^{t,t+\ell} &= \sum_{h_2=1}^{H_2}\delta_{h_2}^{t,t+\ell}w_{h_1,h_2}, && h_1\in[1,D] \\ \delta_{h_2}^{t,t+\ell} = \delta_k^{t,t+\ell} &= y_k^t - t_k^{t+\ell}, && k\in[1,V] \end{aligned} \tag{2.3.7}$$

After the $\delta_{h_\ell}^{t,t+\ell}$ values are calculated, the weight derivatives can be calculated:

$$\begin{aligned} \frac{\partial\mathcal{L}}{\partial w_{h_0,h_1}} &= \sum_{t=1}^{T}\sum_{\ell}\delta_{h_1}^{t,t+\ell}x_{h_0}^t, && \forall h_0\in[1,V], h_1\in[1,D] \\ \frac{\partial\mathcal{L}}{\partial w_{h_1,h_2}} &= \sum_{t=1}^{T}\sum_{\ell}\delta_{h_2}^{t,t+\ell}b_{h_1}^t, && \forall h_1\in[1,D], h_2\in[1,V] \end{aligned} \tag{2.3.8}$$

2.4 Paragraph2vec

The Skip-gram model only directly allows to get a vector representation of a single word, not a sentence or more. But to find a vector representation for a news article one must work on a sentence level. A simple approach is to just take the average vector sum of all the words in the article title and perhaps the article lead.

While this method is perfectly valid, it neglects the order of words just like the bag-of-word methods. The paragraph2vec model has been suggested as a solution to this problem. This is a set of models that looks very similar to the skip-gram model (or word2vec in general), but also learns a document vector which should capture the content of a document.

The model shown here is the Distributed Memory Model of Paragraph Vectors (short PV-DM), as it has been shown to provide the best results [2].

2.4.1 Forward-pass

The idea behind the PV-DM model is a bit different from the skip-gram model. Instead of using one word to predict multiply surrounding words, it uses a window of previous words to predict just a single word. In addition to this, the model uses a unique document vector for each document to adjust the word prediction probabilities.

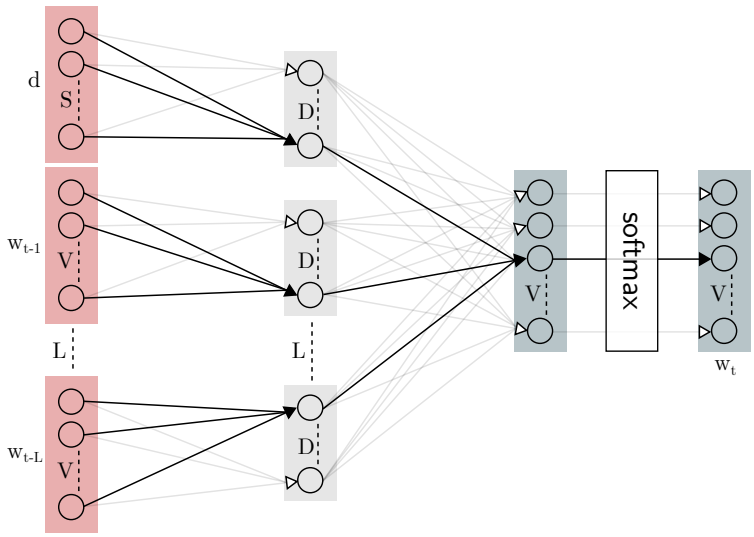


Figure 2.4.1: Visualization of the paragraph2vec model called “Distributed Memory Model of Paragraph Vectors”.

Lets denote the indicator vector for the word at position t in a document as $\{x_i^{w_t}\}_{i=1}^{V}$. The indicator vector describing the document d is then $\{x_i^d\}_{i=1}^{S}$, where S is the amount of documents. The document vectors $b_{h_1}^d$ and word vectors are then calculated as:

$$\begin{aligned} b_{h_1}^{w_{t-\ell}} = a_{h_1}^{w_{t-\ell}} &= \sum_{i=1}^{V} w_{i,h_1} x_i^{w_{t-\ell}} \quad && \forall \ell \in [1, L], h_1 \in [1, D] \\ b_{h_1}^d = a_{h_1}^d &= \sum_{i=1}^{S} d_{i,h_1} x_i^d && \forall h_1 \in [1, D] \end{aligned} \tag{2.4.1}$$

Note that the document vector is unique for each document but shared for all word predictions in that document. The word vectors are then shared across all documents, just like in the skip-gram model.

The document and word vectors are then concatenated into a single vector, which is then multiplied by a weight matrix to generate the softmax input for the word prediction.

$$\begin{aligned} a_k^t = a_{h_2}^t &= \sum_{\ell=1}^{L} \sum_{h_1}^{D} w_{h_1+\ell \cdot D, h_2} b_{h_1}^{w_{t-\ell}} + \sum_{h_1}^{D} w_{h_1,h_2} b_{h_1}^d && \forall k \in [1, V] \\ y_k^t &= \frac{\exp(a_k^t)}{\sum_{k'=1}^{V} \exp(a_{k'}^t)} && \forall k \in [1, V] \end{aligned} \tag{2.4.2}$$

The softmax used for calculating y_k^t performs a very large sum. This sum is impractical to calculate, for this reason the paper [2] suggest using an approximation called hierarchical softmax.

The model is trained just like the skip-gram model, that is using stochastic gradient descent. This optimizes the document and word vectors jointly. It is possible to later fix the word vectors and only train the document vector, but that wasn't done in this case.

2.5 Recurrent Neural Network

The Skip-gram model and the paragraph2vec model can be interpreted as a feed forward neural network (FFNN). This class of neural networks have the shortcoming that the require a fixed input size, thus they can only look a fixed amount of steps backwards and forwards in history. This can be a problem when analyzing sentences, because the amount of words in a sentence isn't fixed and there isn't a well defined window size for the surrounding words.

The Recurrent Neural Networks (RNN) overcome this shortcoming by having an internal state. In principal this allows the network to take a sequence of vectors and consider the entire history. The sequence can be of any size, but each vector must be of the same size. In the case of sentences a fixed vector size is not an issue when considering a fixed vocabulary.

In practice it turns out that the RNN forgets the past and only have a short memory. Another model called Long-Short-Term-Memory (LSTM) extends the RNN and have been shown to have better performance [7]. The ideas are however still the same, thus to understand the LSTM model a basic understanding of the RNN model is valuable.

2.5.1 Forward pass

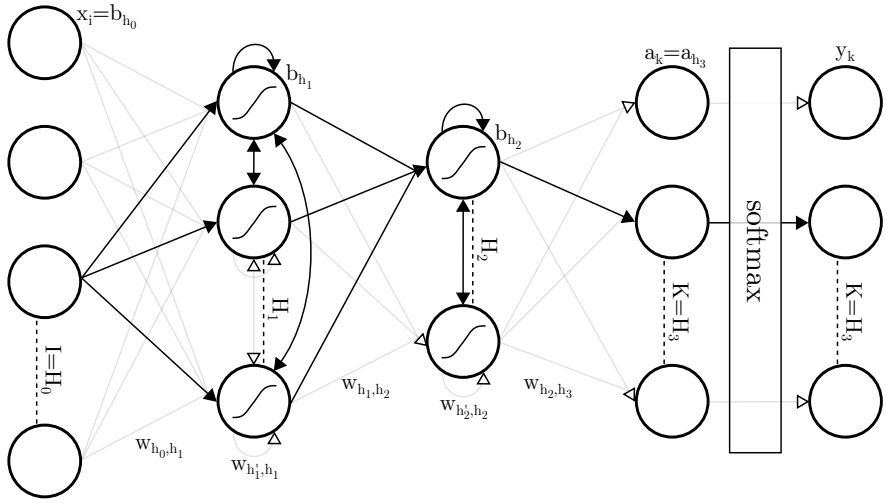


Figure 2.5.1: Visualization of a single iteration in a Recurrent Neural Network.

Consider Figure 2.5.1 but without the extra internal connections, such that it is just a feed forward neural network. The values in the input vector is then denoted $x_i, i \in [1, I]$, the corresponding input for the hidden layer is $a_{h_\ell}, \forall \ell \in [1, H_\ell]$ with an

activation $b_{h_\ell}, \forall \ell \in [1, H_\ell]$. Finally there is the network output $a_k, \forall k \in [1, K]$. This is just like the notation used for the FFNN.

However since a sequence of $\mathbf{x}$ vectors will be used later, a $t \in [1, T]$ is added to the notation. Note that just because a t symbol is used, the RNN model generalizes to non-time dependent datasets as well. In the case of semantic text analysis the sequence $\{\mathbf{x}^t\}_{t=1}^T$ will refer to a sequence of words represented using 1-of-V encoding. The total sequence then contains a complete sentence or paragraph.

Now recall that in a FFNN the input for the hidden layer is:

$$a_{h_\ell}^t = \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} b_{h_{\ell-1}}^t \quad \forall \ell \in [1, L+1] \tag{2.5.1}$$

In the case of a RNN the input for the hidden layer ($a_{h_\ell}^t$) is expanded, such that it depends on the hidden activation for the previous time iteration ($b_{h_\ell}^{t-1}$):

$$a_{h_\ell}^t = \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} b_{h_{\ell-1}}^t + \sum_{h'_\ell=1}^{H_\ell} w_{h'_\ell h_\ell} b_{h'_\ell}^{t-1} \quad \forall \ell \in [1, L] \tag{2.5.2}$$

Just as with the FFNN, θ is the activation function:

$$b_{h_\ell}^t = \theta(a_{h_\ell}^t) \quad \forall \ell \in [1, L] \tag{2.5.3}$$

In order to fully calculate the forward pass, $b_{h_\ell}^0$ needs to be initialized to some value. Usually it is set to 0 however there are other choices.

2.5.2 Backward pass

Just like *error backpropagation* is used to calculate the loss function derivatives with respect to the weights in a FFNN, similar method exists for the RNN. The method is called *backpropagation through time* (BPTT). Is almost identical to backpropagation in the feed forward network, except that it doesn't just go backward through the layers but also goes backward through time (t), or in this case a sequence of words. [7]

Consider again the log-likelihood function for a multiclass problem, where the probabilities y_k^t are calculated using a softmax. However this time the probability must be considered for all time steps.

$$\mathcal{L} = -\sum_{t=1}^{T} \sum_{k=1}^{K} t_k^t \ln(y_k^t) \tag{2.5.4}$$

$$y_k^t = \frac{\exp(a_k^t)}{\sum_{k'=1}^{K} \exp(a_{k'}^t)} \tag{2.5.5}$$

Just like in the FFNN the derivatives with respect to the first weights are considered as a starting point. However because of the time dependency the chain rule results in a sum over the time steps:

$$\frac{\partial \mathcal{L}}{\partial w_{h_0,h_1}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_{h_1}^t} \frac{\partial a_{h_1}^t}{w_{h_0,h_1}} \tag{2.5.6}$$

Now the δ definition is introduced:

$$\delta_{h_\ell}^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \tag{2.5.7}$$

Applying this definition and using $\frac{\partial a_{h_1}^t}{w_{h_0,h_1}} = x_{h_0}^t$, (2.5.6) becomes:

$$\frac{\partial \mathcal{L}}{\partial w_{h_0,h_1}} = \sum_{t=1}^{T} \delta_{h_1}^t x_{h_0}^t = \sum_{t=1}^{T} \delta_{h_1}^t b_{h_0}^t \tag{2.5.8}$$

The next step is to derive the equation for $\delta_{h_1}^t$, this is a bit tricky because both $a_{h_2}^t$ and $a_{h_1}^{t+1}$ is affected by $a_{h_1}^t$, thus the chain rule gives two sums:

$$\delta_{h_1}^t = \frac{\partial \mathcal{L}}{\partial a_{h_1}^t} = \sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}^t} \frac{\partial a_{h_2}^t}{\partial a_{h_1}^t} + \sum_{h_1'=1}^{H_1} \frac{\partial \mathcal{L}}{\partial a_{h_1'}^{t+1}} \frac{\partial a_{h_1'}^{t+1}}{\partial a_{h_1}^t} \tag{2.5.9}$$

Again the chain rule is used, this time to get $\frac{\partial b_{h_1}^t}{\partial a_{h_1}^t}$:

$$\begin{aligned}
\delta_{h_1}^t = \frac{\partial \mathcal{L}}{\partial a_{h_1}^t} &= \sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}^t} \frac{\partial a_{h_2}^t}{\partial b_{h_1}^t} \frac{\partial b_{h_1}^t}{\partial a_{h_1}^t} + \sum_{h_1'=1}^{H_1} \frac{\partial \mathcal{L}}{\partial a_{h_1'}^{t+1}} \frac{\partial a_{h_1'}^{t+1}}{\partial b_{h_1}^t} \frac{\partial b_{h_1}^t}{\partial a_{h_1}^t} \\
&= \frac{\partial b_{h_1}^t}{\partial a_{h_1}^t} \left(\sum_{h_2=1}^{H_2} \frac{\partial \mathcal{L}}{\partial a_{h_2}^t} \frac{\partial a_{h_2}^t}{\partial b_{h_1}^t} + \sum_{h_1'=1}^{H_1} \frac{\partial \mathcal{L}}{\partial a_{h_1'}^{t+1}} \frac{\partial a_{h_1'}^{t+1}}{\partial b_{h_1}^t} \right) \\
&= \theta'(a_{h_1}^t) \left(\sum_{h_2=1}^{H_2} \delta_{h_2}^t w_{h_1,h_2} + \sum_{h_1'=1}^{H_1} \delta_{h_1}^{t+1} w_{h_1',h_1} \right)
\end{aligned} \tag{2.5.10}$$

Note that $\delta_{h_1}^t$ is not yet completely well defined, because for $t = T$ there is an unknown $\delta_{h_1}^{T+1}$ term. This is set to zero since "no error is received from beyond the end of the sequence" [7].

From (2.5.10) its seen that the next step is to derive $\delta_{h_2}^t$. This was also the case with the FFNN and it what shown that δ_{h_2} was not much different from $\delta_{h_1}^t$ and

generalizing it was the natural choice. The same is the case for the RNN. Generalizing then gives:

$$\delta_{h_\ell}^t = \theta'(a_{h_\ell}^t) \left(\sum_{h_{\ell+1}=1}^{H_{\ell+1}} \delta_{h_{\ell+1}}^t w_{h_\ell, h_{\ell+1}} + \sum_{h'_\ell=1}^{H_\ell} \delta_{h_\ell}^{t+1} w_{h'_\ell, h_\ell} \right), \quad \forall \ell \in [1, L] \tag{2.5.11}$$

Because δ_{L+2} doesn't exists, it is seen from the generalization that δ_{L+1}^t is again a special case:

$$\delta_{L+1}^t = \delta_k^t = \frac{\partial \mathcal{L}}{\partial a_{h_k}^t} = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}}{\partial y_{k'}^t} \frac{\partial y_{k'}^t}{\partial a_k^t} \tag{2.5.12}$$

This is similar to the FFNN case with the exception that there is a time step, thus one gets:

$$\delta_k^t = y_k^t - t_k^t \tag{2.5.13}$$

This set of equation allows calculation of all $\delta_{h_\ell}^t$, however the derivatives with respect to the $w_{h'_1, h_1}$ weights still need to be considered.

$$\frac{\partial \mathcal{L}}{\partial w_{h'_1, h_1}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_{h_1}^t} \frac{\partial a_{h_1}^t}{\partial w_{h'_1, h_1}} = \sum_{t=1}^{T} \delta_{h_1}^t b_{h'_1}^{t-1} \tag{2.5.14}$$

Generalizing this both for the weights between layer $w_{h_{\ell-1}, h_\ell}$ and between time steps $w_{h'_\ell, h_\ell}$, one gets:

$$\frac{\partial \mathcal{L}}{\partial w_{h'_\ell, h_\ell}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \frac{\partial a_{h_\ell}^t}{\partial w_{h'_\ell, h_\ell}} = \sum_{t=1}^{T} \delta_{h_\ell}^t b_{h'_\ell}^{t-1}, \qquad \forall \ell \in [1, L] \tag{2.5.15}$$

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1}, h_\ell}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \frac{\partial a_{h_\ell}^t}{\partial w_{h_{\ell-1}, h_\ell}} = \sum_{t=1}^{T} \delta_{h_\ell}^t b_{h_{\ell-1}}^t, \qquad \forall \ell \in [1, L+1] \tag{2.5.16}$$

2.6 Long-Short Term Memory

In the Recurrent Neural Network the internal state b_h^t must be updated in each iteration. This means the effect of some input $x_i^{t_0}$ on b_h^t, will decay as $t \gg t_0$. This issue is addressed in the Long-Short Term Memory Model (LSTM) by taking inspiration from a computer memory cell.

A computer memory cell can either be written to, read from or reset. On the most basic level memory cell are binary, not just in terms of what can be stored but also in terms of the 3 operations. Either the memory cell is completely reset or not reset, either completely read from or not read from and similarly for writing.

From a mathematical perspective all these operations becomes threshold functions, which are not continuous and therefore not differentiable. This makes them impractical to use in a neural network setting, as neural networks are optimized using gradient descent. The solution is not to use threshold function but a combination of differentiable non-linear activation functions, such as the sigmoid or hyperbolic tangent function.

2.6.1 The transistor

The memory part (cell) of the *memory block* is from a mathematical modeling perspective just a variable. The complexity in the differentiable memory block lies in how that memory is controlled. There are 3 operations read, write and reset. Each of these operations are controlled by a mechanism that resembles a transistor.

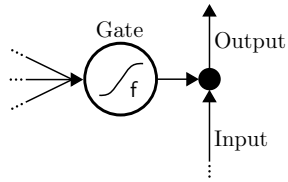


Figure 2.6.1: Visualization of the differentiable transistor.

A transistor have an input (*collector*) which is send to the output (*emitter*). In a circuit transistor the output is only equal to the input if the gate (*base*) voltage exceeds a certain threshold, otherwise the output have zero voltage. In the differentiable transistor the idea is the same, except that the gate have a continues value typically in the interval $[0, 1]$. The output value is thus calculated by multiplying the gate value with the input value.

$$a_{output} = b_{gate} a_{input} \tag{2.6.1}$$

The gate value is typically created from a weighted sum of other values, such as the input vector. To control this sum a non-linear function f is used, typically this is the sigmoid function such that the gate value is in the expected interval $[0, 1]$.

$$b_{gate} = f(a_{gate}), \quad a_{gate} = \sum_{i=1}^{I} w_i x_i \tag{2.6.2}$$

2.6.2 The memory block

The LSTM model is the same as the RNN model, except that all non-linear units are replaced by a differentiable memory block. The differentiable memory block have an *input* corresponding to $a_{h_\ell}^t$ and an *output* corresponding to $b_{h_\ell}^t$ in the RNN model. To control the three memory operations in the memory block, there are three gates. The *Input Gate* ($b_{\rho_\ell}^t$), *Forget Gate* (b_{ϕ_ℓ}) and *Output Gate* ($b_{\omega_\ell}^t$).

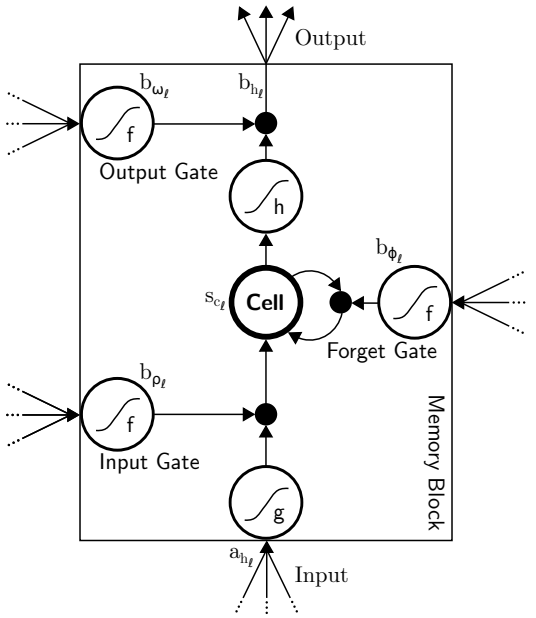


Figure 2.6.2: Visualization of the differentiable memory block.

Consider Figure 2.6.2. The cell state $s_{c_\ell}^t$ is updated as a combination (sum) of the transistor output from the previous *cell* value $s_{c_\ell}^{t-1}$ and the transistor output of the *memory block* input $g(a_{h_\ell}^t)$:

$$s_{c_\ell}^t = b_{\phi_\ell}^t s_{c_\ell}^{t-1} + b_{\rho_\ell}^t g(a_{h_\ell}^t) \tag{2.6.3}$$

Here $b^t_{\phi_\ell}$ and $b^t_{\rho_\ell}$ are the values of respectively the *forget gate* and *input gate*. Thus if $b^t_{\phi_\ell} = 0$, the cell state is completely forgotten and if $b^t_{\phi_\ell} = 1$ it is fully remembered, however the *memory block* input can still affect its final value. The input part of the cell state update is controlled by the *input gate* $b^t_{\rho_\ell}$.

The gate values are all calculated using a weighted sum of the input vector x^t_i and previous hidden output $b^{t-1}_{h_\ell}$, just like the a_{h_ℓ} value in RNN. This sum is then transformed using a non-linear function f, such that the gate value is within $[0, 1]$.

$$\begin{aligned} a^t_{\rho_\ell} &= \sum_{i=1}^{I} w_{i,\rho_\ell} x^t_i + \sum_{h'=1}^{H} w_{h'_\ell,\rho_\ell} b^{t-1}_{h'_\ell} \\ b^t_{\rho_\ell} &= f(a^t_{\rho_\ell}) \end{aligned} \tag{2.6.4}$$

Equation (2.6.4) is just for the input gate (ρ). The equations for the forget gate (ϕ) and output gate (ω) are similar, just replace the gate symbol. Note also that while the non-linear function f is the same for all gates, the weights are different.

The memory output is created from the cell state and the output gate:

$$b^t_{h_\ell} = b^t_{\omega_\ell} h(s^t_{c_\ell}) \tag{2.6.5}$$

h is yet another non-linear function, typically the sigmoid $\sigma(\cdot)$ or hyperbolic tangent $\tanh(\cdot)$ function is used, though it is also normal to simple use the identity function $\theta(a) = a$. An interesting properly of using the identity function is that the LSTM unit generalizes to the simple RNN unit, by setting $b^t_{\rho_t} = 1, b^t_{\phi_t} = 1, b^t_{\omega_t} = 1$

2.6.3 Forward pass

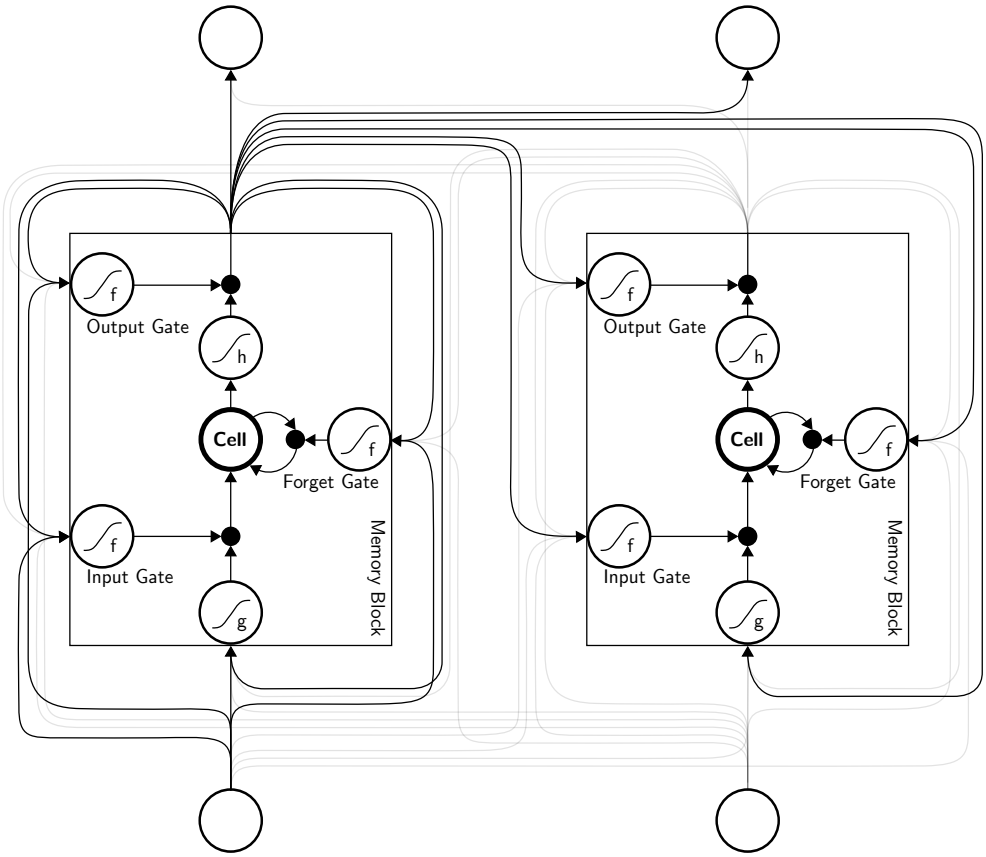


Figure 2.6.3: Visualization of a LSTM neural network with a single hidden layer.

Figure 2.6.3 shows a very simple LSTM network. It have the recurrent property from the RNN, that is the output from the memory blocks are used as input in the next iteration. Figure 2.6.3 only shows two inputs, but in general there are no restrictions. For simplicity only a single layer is shown, however a multilayer LSTM network can easily be constructed, in the same way a multilayer RNN is constructed. That is by making the output of the memory blocks in the current hidden layer go into the next hidden layer.

By using the memory block equations and generalizing to L hidden layers, the full set of equations becomes:

Memory Block Input:

$$a_{h_\ell}^t = \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} b_{h_{\ell-1}}^t + \sum_{h'=1}^{H} w_{h'_\ell,h_\ell} b_{h'_\ell}^{t-1} \quad \forall \ell \in [1,L] \text{ where: } b_{h_0}^t = x_i^t$$

Input Gate:

$$\begin{aligned} a_{\rho_\ell}^t &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},\rho_\ell} b_{h_{\ell-1}}^t + \sum_{h'=1}^{H} w_{h'_\ell,\rho_\ell} b_{h'_\ell}^{t-1} && \forall \ell \in [1,L] \text{ where: } b_{h_0}^t = x_i^t \\ b_{\rho_\ell}^t &= f(a_{\rho_\ell}^t) && \forall \ell \in [1,L] \end{aligned}$$

Forget Gate:

$$\begin{aligned} a_{\phi_\ell}^t &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},\phi_\ell} b_{h_{\ell-1}}^t + \sum_{h'=1}^{H} w_{h'_\ell,\phi_\ell} b_{h'_\ell}^{t-1} && \forall \ell \in [1,L] \text{ where: } b_{h_0}^t = x_i^t \\ b_{\phi_\ell}^t &= f(a_{\phi_\ell}^t) && \forall \ell \in [1,L] \end{aligned}$$

Cell State:

$$s_{c_\ell}^t = b_{\phi_\ell}^t s_{c_\ell}^{t-1} + b_{\rho_\ell}^t g(a_{h_\ell}^t) \quad \forall \ell \in [1,L] \text{ where: } c_\ell = \phi_\ell = c_\ell = \rho_\ell$$

Output Gate:

$$\begin{aligned} a_{\omega_\ell}^t &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},\omega_\ell} b_{h_{\ell-1}}^t + \sum_{h'=1}^{H} w_{h'_\ell,\omega_\ell} b_{h'_\ell}^{t-1} && \forall \ell \in [1,L] \text{ where: } b_{h_0}^t = x_i^t \\ b_{\omega_\ell}^t &= f(a_{\omega_\ell}^t) && \forall \ell \in [1,L] \end{aligned}$$

Memory Block Output:

$$b_{h_\ell}^t = b_{\omega_\ell}^t h(s_{c_\ell}^t) \quad \forall \ell \in [1,L] \text{ where: } h_\ell = \omega_\ell = c_\ell$$

Network Output:

$$\begin{aligned} a_k^t = a_{h_{L+1}}^t &= \sum_{h_L=1}^{H_L} w_{h_L,h_{L+1}} b_{h_L}^t \\ y_k^t &= \frac{\exp(a_k)}{\sum_{k'=1}^{K} \exp(a_{k'})} \end{aligned}$$

Equation 2.6.6: Forward equations for a multilayer LSTM network.

2.6.4 Backward pass

The backward pass strategy is similar to that used in RNN, except that in the LSTM one have to deal with much more δ bookkeeping.

The loss function is the same as always:

$$\mathcal{L} = -\sum_{t=1}^{T}\sum_{k=1}^{K} t_k^t \ln(y_k^t) \tag{2.6.7}$$

The starting point is also the same. That is the derivative of the loss function with respect to the first weights. To keep it short the generalization for all layers will be considered immediately.

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \frac{\partial a_{h_\ell}^t}{\partial w_{h_{\ell-1},h_\ell}} = \sum_{t=1}^{T} \delta_{h_\ell}^t b_{h_{\ell-1}}^t \quad \text{where: } \delta_{h_\ell}^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \tag{2.6.8}$$

From now on there will be some differences. The first is that $a_{h_\ell}^t$ now affects $s_{c_\ell}^t$, thus the equations for $\delta_{h_\ell}^t$ are different:

$$\delta_{h_\ell}^t = \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} = \frac{\partial \mathcal{L}}{\partial s_{c_\ell}^t} \frac{\partial s_{c_\ell}^t}{\partial a_{h_\ell}^t} = \epsilon_{s_\ell}^t b_{\rho_\ell} g'(a_{h_\ell}^t) \quad \text{where: } \epsilon_{s_\ell}^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_{c_\ell}^t} \tag{2.6.9}$$

Note that a new bookkeeping variable, namely $\epsilon_{s_\ell}^t$ have been introduced. To derive the equations for this value the chain rule is again used. This time it's seen that $\epsilon_{s_\ell}^t$ affects the memory block output $b_{h_\ell}^t$ and itself in the next time iteration $s_{c_\ell}^{t+1}$:

$$\epsilon_{s_\ell}^t = \frac{\partial \mathcal{L}}{\partial s_{c_\ell}^t} = \frac{\partial \mathcal{L}}{\partial b_{h_\ell}^t} \frac{\partial b_{h_\ell}^t}{\partial s_{c_\ell}^t} + \frac{\partial \mathcal{L}}{\partial s_{c_\ell}^{t+1}} \frac{\partial s_{c_\ell}^{t+1}}{\partial s_{c_\ell}^t} = \epsilon_{b_\ell}^t b_{\omega_\ell}^t h'(s_{c_\ell}^t) + \epsilon_{s_\ell}^{t+1} b_{\phi_\ell}^{t+1} \quad \text{where: } \epsilon_{b_\ell}^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial b_{h_\ell}^t}$$

Again a new bookkeeping variable is introduced. Calculating $\epsilon_{b_\ell}^t$ is quite complicated because $b_{h_\ell}^t$ affects many other parts of the network.

There are two cases, in both cases it affects the current layer on all gates and the memory block input $(a_{h_\ell}^{t+1}, a_{\rho_\ell}^{t+1}, a_{\phi_\ell}^{t+1}, a_{\omega_\ell}^{t+1})$. If the next layer is a hidden layer then it also affects all of its gates and the memory block input $(a_{h_{\ell+1}}^t, a_{\rho_{\ell+1}}^t, a_{\phi_{\ell+1}}^t, a_{\omega_{\ell+1}}^t)$. Otherwise if the next layer is not a hidden layer then it affects the network output (a_k).

Lets first consider the case where the next layer is a hidden layer ($\ell \in [1, L+1]$), using the chain rule one gets:

$$\begin{aligned}\epsilon^t_{b_\ell} = &\sum_{h'_\ell=1}^{H_\ell} \delta^{t+1}_{h'_\ell} w_{h_\ell,h'_\ell} + \sum_{\rho_\ell=1}^{H_\ell} \delta^{t+1}_{\rho_\ell} w_{h_\ell,\rho_\ell} + \sum_{\phi_\ell=1}^{H_\ell} \delta^{t+1}_{\phi_\ell} w_{h_\ell,\phi_\ell} \\ &+ \sum_{\omega_\ell=1}^{H_\ell} \delta^{t+1}_{\omega_\ell} w_{h_\ell,\omega_\ell} + \sum_{h'_{\ell+1}=1}^{H_{\ell+1}} \delta^{t}_{h'_{\ell+1}} w_{h_\ell,h'_{\ell+1}} + \sum_{\rho_{\ell+1}=1}^{H_{\ell+1}} \delta^{t}_{\rho_{\ell+1}} w_{h_\ell,\rho_{\ell+1}} \\ &+ \sum_{\phi_{\ell+1}=1}^{H_{\ell+1}} \delta^{t}_{\phi_{\ell+1}} w_{h_\ell,\phi_{\ell+1}} + \sum_{\omega_{\ell+1}=1}^{H_{\ell+1}} \delta^{t}_{\omega_{\ell+1}} w_{h_\ell,\omega_{\ell+1}} \quad \forall \ell \in [1, L-1]\end{aligned} \tag{2.6.10}$$

In the other case where the next layer is the output later ($\ell = L$) one gets:

$$\begin{aligned}\epsilon^t_{b_L} = &\sum_{h'_L=1}^{H_L} \delta^{t+1}_{h'_L} w_{h_L,h'_L} + \sum_{\rho_L=1}^{H_L} \delta^{t+1}_{\rho_L} w_{h_L,\rho_L} + \sum_{\phi_L=1}^{H_L} \delta^{t+1}_{\phi_L} w_{h_L,\phi_L} \\ &+ \sum_{\omega_L=1}^{H_L} \delta^{t+1}_{\omega_L} w_{h_L,\omega_L} + \sum_{k=1}^{K} \delta^{t}_{k} w_{h_L,k}\end{aligned} \tag{2.6.11}$$

This introduces 4 new bookkeeping values:

$$\delta^t_{\rho_\ell} \overset{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{\rho_\ell}}, \quad \delta^t_{\phi_\ell} \overset{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{\phi_\ell}}, \quad \delta^t_{\omega_\ell} \overset{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{\omega_\ell}}, \quad \delta^t_{k} \overset{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_{k}} \tag{2.6.12}$$

Each of these bookkeeping values are a bit different, but the chain rule is still used to derive them. First is $\delta^t_{\rho_\ell}$, through $b^t_{\rho_\ell}$ this affects $s^t_{c_\ell}$.

$$\delta^t_{\rho_\ell} = \frac{\partial \mathcal{L}}{\partial a^t_{\rho_\ell}} = \frac{\partial \mathcal{L}}{\partial b^t_{\rho_\ell}} \frac{\partial b^t_{\rho_\ell}}{\partial a^t_{\rho_\ell}} = \frac{\partial \mathcal{L}}{\partial s^t_{c_\ell}} \frac{\partial s^t_{c_\ell}}{\partial b^t_{\rho_\ell}} \frac{\partial b^t_{\rho_\ell}}{\partial a^t_{\rho_\ell}} = \epsilon^t_{s_\ell} g(a^t_{h_\ell}) f'(a^t_{\rho_\ell}) \tag{2.6.13}$$

Next is $\delta^t_{\phi_\ell}$, the situation is somewhat the same, $a^t_{\phi_\ell}$ affects $s^t_{c_\ell}$ through $b^t_{\phi_\ell}$:

$$\delta^t_{\phi_\ell} = \frac{\partial \mathcal{L}}{\partial a^t_{\phi_\ell}} = \frac{\partial \mathcal{L}}{\partial b^t_{\phi_\ell}} \frac{\partial b^t_{\phi_\ell}}{\partial a^t_{\phi_\ell}} = \frac{\partial \mathcal{L}}{\partial s^t_{c_\ell}} \frac{\partial s^t_{c_\ell}}{\partial b^t_{\phi_\ell}} \frac{\partial b^t_{\phi_\ell}}{\partial a^t_{\phi_\ell}} = \epsilon^t_{s_\ell} s^{t-1}_{c_\ell} f'(a^t_{\phi_\ell}) \tag{2.6.14}$$

Then $\delta^t_{\omega_\ell}$, here $a^t_{\omega_\ell}$ affects $b^t_{h_\ell}$ through $b^t_{\omega_\ell}$:

$$\delta^t_{\omega_\ell} = \frac{\partial \mathcal{L}}{\partial a^t_{\omega_\ell}} = \frac{\partial \mathcal{L}}{\partial b^t_{\omega_\ell}} \frac{\partial b^t_{\omega_\ell}}{\partial a^t_{\omega_\ell}} = \frac{\partial \mathcal{L}}{\partial b^t_{h_\ell}} \frac{\partial b^t_{h_\ell}}{\partial b^t_{\omega_\ell}} \frac{\partial b^t_{\omega_\ell}}{\partial a^t_{\omega_\ell}} = \epsilon^t_{b_\ell} h(s^t_{c_\ell}) f'(a^t_{\omega_\ell}) \tag{2.6.15}$$

Finally there is δ_k^t, here a_k^t affects y_k^t. This is no different from the RNN or FFNN case, thus one gets:

$$\delta_k^t = \frac{\partial \mathcal{L}}{\partial y_k^t} \frac{\partial y_k^t}{\partial a_k^t} = y_k^t - t_k^t \tag{2.6.16}$$

The δ and ϵ values can be calculated by going back though the network and by initializing with $\delta_{\rho_\ell}^{T+1} = \delta_{\phi_\ell}^{T+1} = \delta_{\omega_\ell}^{T+1} = \delta_{h_\ell}^{T+1} = \epsilon_{s_\ell}^{T+1} = 0$.

Until now only $\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}}$ have been considered. But now that δ and ϵ values are fully expressed, the remaining derivatives are quite simple.

$$\frac{\partial \mathcal{L}}{\partial w_{h'_\ell,h_\ell}} = \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^t} \frac{\partial a_{h_\ell}^t}{\partial w_{h'_\ell,h_\ell}} = \delta_{h_\ell}^t b_{h'_\ell}^{t-1} \tag{2.6.17}$$

The gate weights are almost identical:

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},\rho_\ell}} = \delta_{\rho_\ell}^t b_{h_{\ell-1}}^t, \quad \frac{\partial \mathcal{L}}{\partial w_{h_\ell,\rho_\ell}} = \delta_{\rho_\ell}^t b_{h_\ell}^{t-1} \tag{2.6.18}$$

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},\phi_\ell}} = \delta_{\phi_\ell}^t b_{h_{\ell-1}}^t, \quad \frac{\partial \mathcal{L}}{\partial w_{h_\ell,\phi_\ell}} = \delta_{\phi_\ell}^t b_{h_\ell}^{t-1} \tag{2.6.19}$$

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},\omega_\ell}} = \delta_{\omega_\ell}^t b_{h_{\ell-1}}^t, \quad \frac{\partial \mathcal{L}}{\partial w_{h_\ell,\omega_\ell}} = \delta_{\omega_\ell}^t b_{h_\ell}^{t-1} \tag{2.6.20}$$

2.7 Sutskever

The model in section 2.6 - Long-Short Term Memory provides an elegant way to classify each word in a sequence of text This can be useful to classifying the mood of a word (positive, negative or neutral) or similar cases where there is a one to one alignment between a word and a output class. This kind of problem is called sequence classification [7, p. 10]. However for finding a good latent representation for a document there doesn't exists such an alignment, thus sequence classification is not very useful.

The Sutskever model provides a method for finding a latent representation given some input document. Unfortunately this is a supervised model, as it also requires an output sequence. In the original paper [3] the model is used to translate English text to French text.

The overall idea is to have two neural networks, an encoder which takes an input document and outputs a vector, and a decoder which takes the same vector and outputs a document. The vector will then be a latent representation of the input document and hopefully this will contain enough information such that the document can be reconstructed in another language.

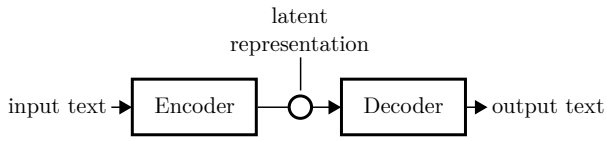


Figure 2.7.1: Rough overview of the Sutskever model.

In the case of news articles translations are quite rare, so instead the first paragraph (the article lead) is used as the input and the article title is used as the output. This way the latent representation will hopefully contain an encoded summary of the article.

2.7.1 Text encoding

The Sutskever paper [3] used a lot of computational resources (8 GPUS over 10 days) to manage the large network (8 layers with 1000 LSTM units each) and a large softmax (output vocabulary was at 80000 words). Because Theano (used to implement the model), currently don't support using more than one GPU, the network size is reduced and letters are used instead of words.

Using letters instead of words means that the sequences gets longer. The average word length in English text is 5.1 character, including the space separator one should expect

6.1 longer sequences. However by using letters the softmax size is decreased from 80000 words to just 78 letters. It should be noted that Unicode NFKD normalization [12] was used to remove accents and ligature, rarely used letters was also removed to reduce the softmax size to 78 letters.

The Sutskever paper [3] do indicates that the model may not work for long sequences, which could be a problem when using letters instead of words. However given the limited resources this still seem like a good solution.

The input and output sequence are both encoded using 1-of-V encoding, where V is the amount of unique letters. In order to indicate to the network when an input or output sequence is ended, an end-of-sequence "letter" is added to the vocabulary. This special letter will be denoted `<EOS>`.

At last the paper also reverses the order of the output sequence, but this doesn't change the model as it is just a pre processing of the target value.

2.7.2 Forward pass

To separate the encoder and decoder in the notation, a d (decoder) or an e (encoder) is added to the notation on the time index.

The encoder works like a normal LSTM network except that the network output $y_k^{t_e}$ isn't used. The network is just used to scan though the input sequence x^{t_e} and the network state is completely carried by the hidden output $b_{h_\ell}^{t_e}$ and the cell state $s_{c_\ell}^{t_e}$. When the network receives an `<EOS>` letter, the hidden output and the cell state from the last layer is used as the latent representation.

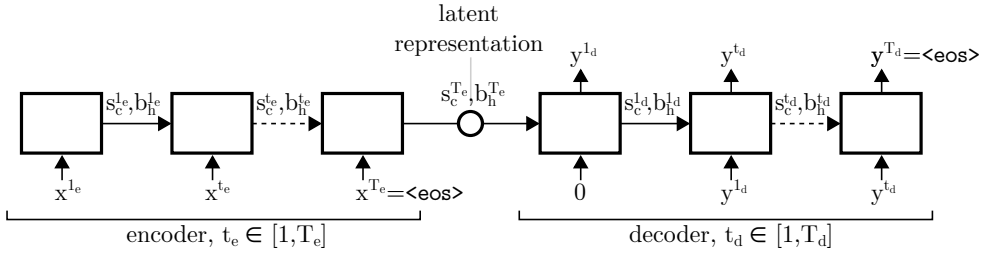


Figure 2.7.2: Each box represent a single time iteration.

The latent representation consisting of the hidden output $b_{h_L}^{T_e}$ and cell state $s_{c_L}^{T_e}$, is then used to initialize the hidden output and cell state in the decodes first layer.

$$\begin{aligned} b_{h_1}^{0_d} &= b_{h_L}^{T_e} \\ s_{c_1}^{0_d} &= s_{c_L}^{T_e} \end{aligned} \tag{2.7.1}$$

The decoder is more special than the encoder, because the network output $y_k^{t_d}$ is used as the input in the next time iteration. For initializing the decoder input $y_k^{0_d} = 0$ is used.

$$x_k^{t_d} = y_k^{t_d-1}, \quad \text{where: } y_k^{0_d} = 0 \tag{2.7.2}$$

The $y_k^{t_d}$ sequence is the output sequence and is compared to the target sequence using a normal entropy loss function.

$$\mathcal{L} = -\sum_{t_d}^{T_d}\sum_{k}^{K} t_k^t \ln(y_k^{t_d}) \tag{2.7.3}$$

In practice where mini-batch gradient descent is used for optimization, the output and target sequences for all observations in the mini-batch, must be equally long to fit the data structure. To do this the target sequence is padded with `<EOS>` letters, until it fits the max sequence length in the mini-batch. This means the network doesn't stop predicting at the first `<EOS>` output letter, but continues until the max sequence length in the mini-batch is meet. A side effect of this is that the loss function will punish non-`<EOS>` letters after the first `<EOS>` letter. However because it should be fairly easy for the network to learn $y_k^{t_d} =$ `<EOS>` $\Rightarrow y_k^{t_d+1} =$ `<EOS>`, this side effect shouldn't matter that much.

The backward pass will look somewhat similar to the simple LSTM network (see section 2.6.4), however it will be much more complicated because the entire Sutskever network actually consists of two somewhat separate networks. For this reason the deriving the backward pass is skipped here. In practice Theano [4, 5] is used to derive the backward pass.

2.8 Naïve clustering

While clustering isn't the focus of this thesis, it is necessary to use some simple clustering algorithm in order to inspect the effectiveness of the algorithms used for producing the document vectors.

A problem with clustering in this case, is that there are really many clusters. The experiments was limited to just 100000 articles, but many more could have been considered. With 100000 articles and assuming that there are 4 articles for each story, then one should expect 25000 clusters.

25000 clusters is a lot, and even simple algorithms like k-means are NP-hard problems and thus unlikely to be able to handle this many clusters. Furthermore the amount of clusters is actually unknown, while algorithms do exists to estimate the amount, many of those also have a hard time with this many clusters.

Because of the difficulties associated with having an unknown and very high amount of clusters, a very simple version of the hierarchical clustering is used.

2.8.1 Hierarchical clustering

Hierarchical clustering is a general algorithm that don't create hard clusters but maps the observations into a tree. In general there are two types. One type starts by considering all observations as one cluster and then split it up step-by-step, this is the *divisive* type. The other type consider each observation as a single clusters and then join them step-by-step, this is the *agglomerative* type.

The *divisive* type has the computational complexity of $\mathcal{O}(2^n)$ and is thus completely infeasible. The *agglomerative* is typically $\mathcal{O}(n^3)$ which is still infeasible, but some variations have the computational complexity $\mathcal{O}(n^2)$ which is a bit more manageable [13]. By using an extra application specific trick (temporal connectivity) the computational complexity can be reduced even more, such that the clustering algorithm is feasible for many observations.

There are two choices to be made for the *agglomerative* strategy. How distance between observations are measured and how distance between clusters are measured.

Distance between observations is usually either measures using euclidean distance or cosine similarity, though there are other choices. The choices influences the output a lot, but doesn't change the computational complexity.

Distance between clusters is what controls the computational complexity, in order to keep things simple the single-linage method is used. This method sets the distance between two clusters as the smallest distance between observations in the two clusters.

$$d(a,b) = \min_{i\in[1,N_a],j\in[1,N_b]} d(a_i,b_j) \tag{2.8.1}$$

As said hierarchical clustering maps observations into a tree. In order to get hard clusters one can cut the tree or stop the algorithm when the distance exceeds a given threshold (t).

The threshold cutting, together with single-linage provides a fairly efficient way of producing the clusters. Just calculate the distance matrix D between observations, then turn it into a semi-connectivity matrix ($D < t$) by using the threshold t. This connectivity matrix can then be used to produce the final clusters, by walking the connectivity graph.

```
ungrouped = set(observations)
while ungrouped not empty do
    start_node = pop(ungrouped)
    group = set(start_node)
    lookup = set(connected_nodes(start_node))
    for each node in lookup do
        group += node
        lookup -= node
        lookup += connected_nodes(node)
    end
    ungrouped -= group
end
```

Algorithm 2.8.1: Simple algorithm for turning a connectivity matrix into clusters.

2.8.2 Temporal connectivity

The bottleneck in this special version of hierarchical clustering is calculating the distance matrix. A distance or norm by definition have the property that $d(x, y) = d(y, x)$ and that $d(x, x) = 0$ so some calculations can be skipped, but it is still an $\mathcal{O}(n^2)$ operation.

To reduce the computational complexity one can assume that articles about the same story are published within a short given timespan. Using the timestamp for each article one can calculate a temporal connectivity matrix. This matrix will then indicate which other articles an article can connect to given the date it was published. This way only distances between nodes that can be connected, given the temporal connectivity needs to be calculated. Assuming the rate by which articles are published is somewhat constant, this reduced the computational complexity to $\mathcal{O}(n)$.

Note that in an real time context where observations keeps getting added to the dataset, the temporal connectivity matrix can be created efficiently by assuming the timestamps are forever increasing. Thus there will always be a limited number of clusters that a new article can connect to, this enables the clustering algorithm to work in real time.

CHAPTER 3

Results

3.1 Software and implementation

This section will not discuss the details of the model implementations, but will just cite the work required for creating the results.

Overall the results was generated with the programming language Python [14]. The skip-gram (word2vec) and paragraph2vec (doc2vec) implementation is from the Python module Gensim [15]. The Sutskever model [3] have not been made available as an open source module and nobody else have implemented and open sourced it. It was thus necessary to implement a framework for creating the network. For generating fast code, utilizing the GPU and deriving the backward pass, the Theano framework [4, 5] was used. From a more thorough discussion on this implementation see Appendix B.

As the intellectual properties for news articles are typically not addressed to the public domain, no big free corpus of news articles exists. The news articles was thus collected by crawling RSS feeds and using a previously developed heuristic[1] for getting the article text and title from the HTML page.

The code for the Sutskever model implementation and generating the results is available under an MIT open source license at `https://github.com/AndreasMadsen/bachelor-code`.

[1] `https://github.com/AndreasMadsen/article`

3.2 Skip-gram

The skip-gram model doesn't need to be trained as there exists a pre trained version at `https://code.google.com/p/word2vec/`. This pre trained version has a latent size of 300 dimensions. The weights are trained using the Google News dataset, which is about 100 billion words and unfortunately not publicly available.

The document vectors are constructed by taking the average sum over all the word vectors. Using the full text of each article is unlikely to provide any good results, as there would be a lot of irrelevant words which would create a lot of noise. Instead just the title of the article is used and as an experiment the title and the lead is also considered.

Document vectors Using principal component analysis the vector representations of all the documents can be visualized in both experiments.

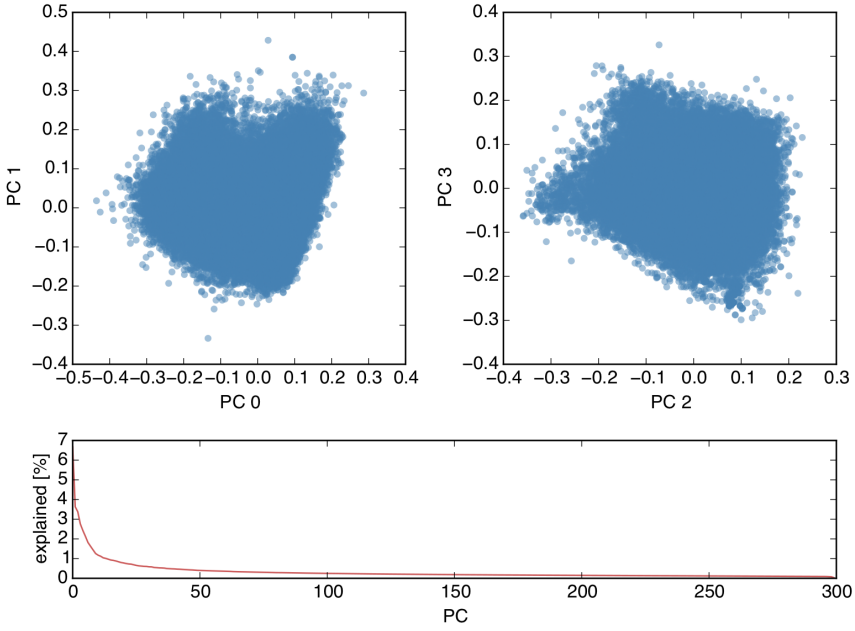


Figure 3.2.1: Document vectors calculated from just the title. Explained variance on the first 4 principal components is 16.4%.

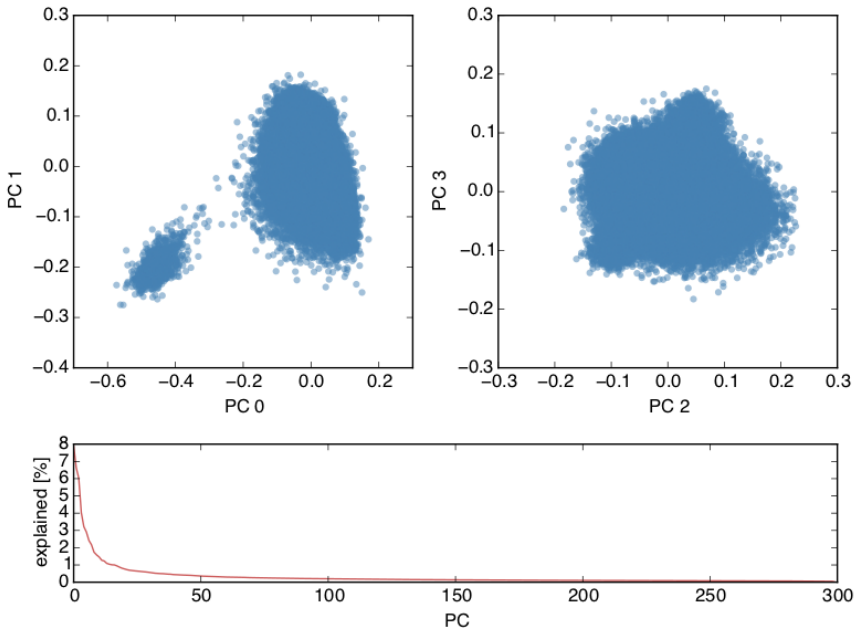


Figure 3.2.2: Document vectors calculated from both title and lead. Explained variance on the first 4 principal components is 24.6%.

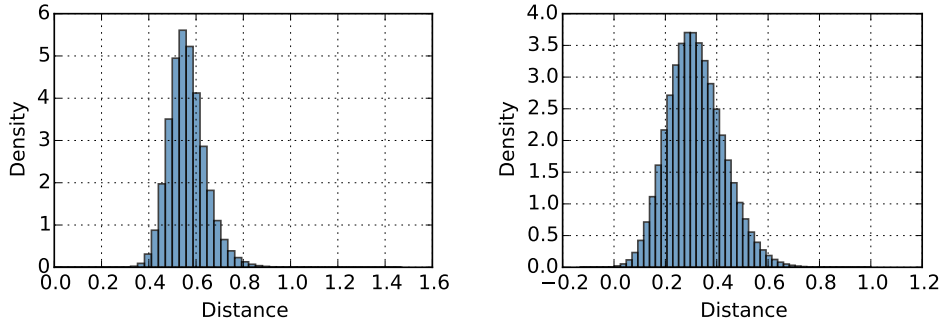
Without any labeling of the documents the scatter plots are not very informative. The explained variance score is not very high compared to what one usually observes on raw data. This indicated the skip-gram model is somewhat effective in generating descriptive document vectors there aren't too correlated.

An interesting result is that the first principal component seem to contain two big clusters of documents. However by inspecting some of the nodes there do not appear to be any reason for this separation.

Distance histogram Since a distance measure will be used to perform the clustering, it makes sense to inspect the histogram of the distance measures. Note that only distances between nodes there are connected by the temporal connectivity matrix is included in the histogram.

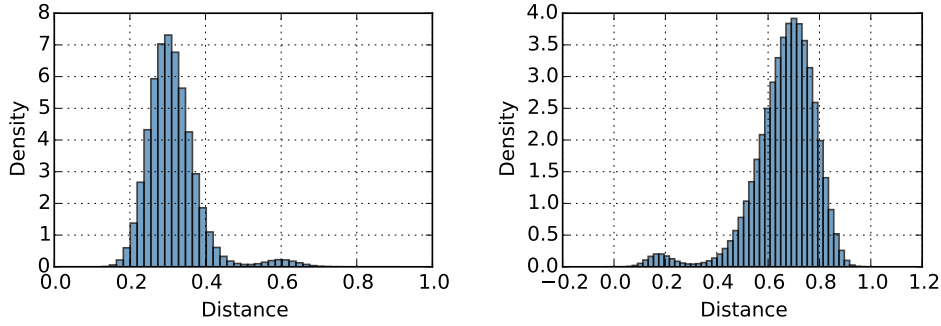
The density estimation required for showing the histogram, also serves the purpose of providing the basis for calculating a somewhat consistent threshold across the different experiments. The threshold t is chosen as the 0.1% percentile, that is $P(distance < t) = 0.001$. This percentile was chosen qualitatively by running a few experiments,

but it turns out it doesn't matter much, as the threshold is not the bottleneck of the performance.



(a) Euclidean distance ($t \approx 0.35$) **(b)** Cosine similarity ($t \approx 0.14$)

Figure 3.2.3: Histograms of the distance values using just the title for generating document vectors.



(a) Euclidean distance ($t \approx 0.14$) **(b)** Cosine similarity ($t \approx 0.092$)

Figure 3.2.4: Histograms of the distance values using both title and lead for generating document vectors.

The histograms shows the distances are approximately normally distributed. This is not very surprising, as the both the euclidean distance and cosine similarly involves a large sum and thus have relations to the central limit theorem.

Inspecting clusters None of the 4 variations gives particularly good results. They all cluster the documents such that there is a ridiculously huge amount of unique

articles. This alone could indicate the threshold is too small. However simultaneously they all have really big clusters, with up to 31478 documents in one case. Because there exists clusters that is either way too big or way too small, the threshold is not the problem. It is the skip-gram model or perhaps the clustering method there causes the bad results.

size	1	2	81	318	715	17023
amount	81857	3	1	1	1	1

Table 3.2.1: Example of the cluster size distribution. In this case both article title and article lead was used for generating document vectors, and cosine similarity was used for clustering. Note that using euclidean distance yields way more reasonably sized clusters, but sill produces too many small clusters and too large clusters.

It should be noted that some clusters do provide good results:

id	title
25167	Teenage stowaway survives five-hour flight hiding in wheel of plane
25354	Survival of teenage stowaway on five-hour flight to Hawaii is medical 'miracle', say experts
25140	Teenager stowaway 'survives five-hour flight in wheel of plane'
26276	How teenage boy stowed away on plane wheel well
25567	Teenage stowaway survives five-hour flight in jet's wheel

Table 3.2.2: A cluster containing articles about the same story. Produced using euclidean distance on document vectors produced from both title and lead.

Similar results can be found when just looking at the title and using euclidean distance for clustering. However using cosine similarity did not yield any meaningful results, independent of whether or not the article lead was included. This is strange as the skip-gram papers [1, 6] uses the cosine similarity for finding similar words. It could be because cosine similarity is usually used in a context where the length of the vector should be ignored. But in this case having a big length could indicate how extreme the article is in its language, which could be related to the story.

3.3 Paragraph2vec

The pre trained skip-gram model can't be used as basis for training the paragraph2vec model, as it don't contain the weights used in the hierarchical softmax, nor does it contain the structure of the Hoffman tree. Because of this both word and document vectors was trained using the gensim implementation.

The model was trained using the full text of all 273813 articles. The dimensionality of the word and document vectors was set to 500. Furthermore the model was trained over 10 epochs, with an initial learning rate of 0.025, which then decreased with 0.002 for each epoch. The vector representations of the chronologically first 100000 articles was then extracted from the final model.

Note that gensim doesn't provide any way of inspecting the loss function directly which is why no such curve is shown here.

Document vectors Using principal component analysis the vector representations of all the documents can be visualized in both cases.

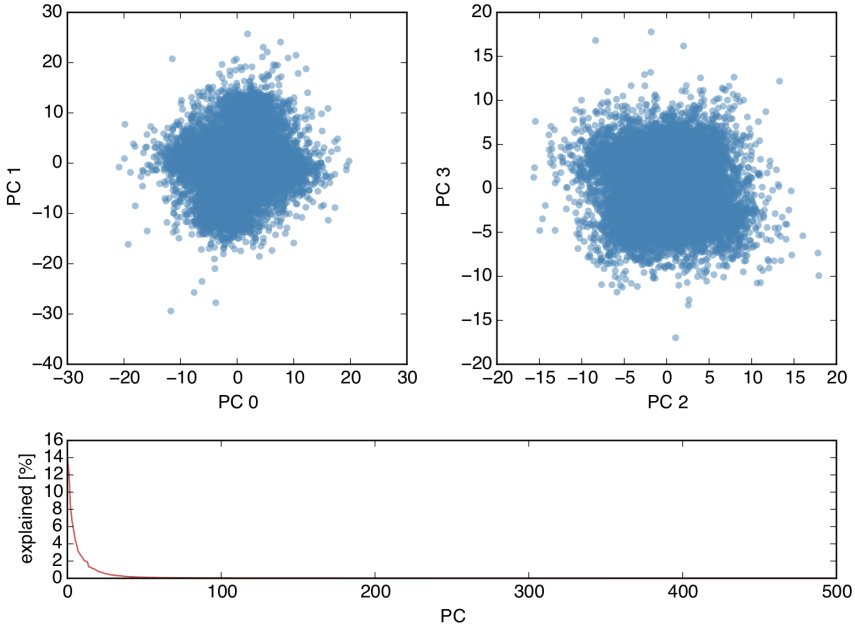
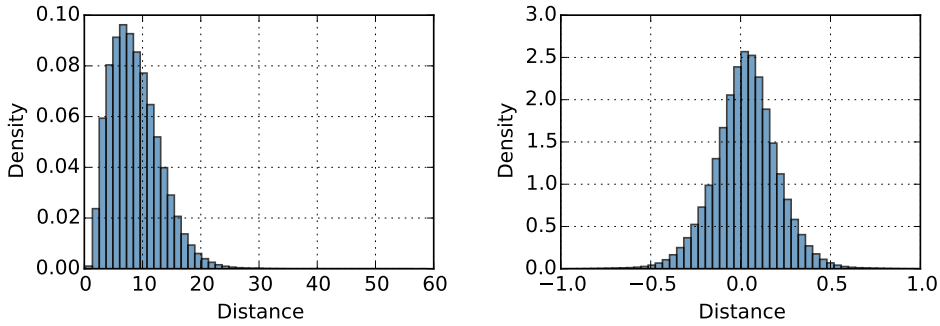


Figure 3.3.1: Document vectors calculated from just the title. Explained variance on the first 4 principal components is 41.4%.

The explained variance is 41.4% for the first 4 principal components, which may be higher than expected. This might be because word2vec part of the model, is quite good at guessing a missing word given just a short context. The big context which is what the document vector provides, may not be very important and only inform about the general rhetorical pattern. This could be (formal versus informal) or (angry versus happy). There thus exists a lower dimensional space there contains almost the same information as the 500 dimensional space.

Distance histogram Again because the distance is used for clustering, the distance histogram is inspected. Furthermore the 0.1% percentile is used for calculating the threshold t.



(a) Euclidean distance ($t \approx 1.35$) **(b)** Cosine similarity ($t \approx -0.66$)

Figure 3.3.2: Histograms of the distance between the document vectors.

The distance histograms are distribution wise a bit cleaner than in the skip-gram case. This is likely because the distribution of the document vectors is more multivariate normal, at least when inspecting the PCA plots. This will of cause result a cleaner distribution in the distance as well.

The euclidean distance distribution looks like a gamma or χ^2 distribution. This is likely because of the squaring in the distance calculation, which prevents the distance from becoming negative. In the skip-gram case this was less apparent because the mean (relative to the standard deviance) was larger.

Inspecting clusters From inspecting the clusters the result seem to be worse when comparing to the skip-gram case.

When inspecting the cluster distribution it turns out that with paragraph2vec, clustering with cosine similarity actually yields more reasonably sized clusters, compared

to when the euclidean distance is used for clustering. This may be because of the χ^2 like distribution of the euclidean distances, which makes it more difficult to separate articles about the same story from those about different stories. It's possible that a more fine-tuned threshold could yield better results, but in any case there still exists too small and way too big clusters.

size	1	2	3	421	711	1855	3326
amount	93665	8	2	1	1	1	1

Table 3.3.1: The cluster distribution using an euclidean distance shows unreasonably large and small clusters, and no reasonably or few reasonably sized clusters.

Unfortunately the actual clusters that the clustering algorithm produces when using cosine similarity don't appear to capture any specific story. This is possibly because the document vectors only captures the overall mood of the article. For example the cluster in Table 3.3.2 shows articles related to different bad events.

At last it was checked if the "teenage stowaway" cluster existed in this case. This should be a fairly easy set of article to cluster, as it contains very specific word combinations. Unfortunately it didn't exists as a single cluster, which again indicates the document vectors only tells about the overall mood of the document, as a way of adjusting the word probabilities.

id	title
9489	Ukraine crisis: Draft document reveals sanctions against Russian and Crimean officials
6644	Six Nations 2014: Stuart Lancaster hints at unchanged England
7573	US opens emergency oil stockpile in signal to Putin
8998	Saudi Arabia bans 50 'blasphemous' and 'inappropriate' children's names
7862	Rangers: Recovery on track after title win - Ally McCoist

Table 3.3.2: Cluster example when using cosine similarity on the paragraph2vec results.

3.4 Sutskever

As no neural network framework exists for creating networks like the Sutskever model, such a framework was written as a part of this thesis. This is a big task where a lot can go wrong, so the first part of this section will attempt to validate the model by having it solve some simple problems.

3.4.1 Constructed problems

Inspired from the learning to execute paper [16] a copy problem is used to validate the model. The idea is simple, generate a sequence of random integers between 1 and 9 of length 8 and append an `<EOS>` element (encoded 0) to that sequence.

$$[3, 4, 8, 5, 9, 9, 4, 4, 0]$$
$$[2, 5, 9, 7, 6, 2, 1, 5, 0]$$
$$[1, 6, 7, 4, 4, 1, 6, 4, 0]$$

Figure 3.4.1: Example of integer sequences used in the *copy* problem.

Each element of value i is then transformed into an indicator vector, such that the 1-element is at position i. The indicator vector will be denoted by a bold font.

Now consider the following three variations of the same problem. Note that that only the *full network* model is required and used for the immediate validation. But the *encoder* and *decoder* problems becomes useful in the detailed analysis.

Full network To validate the full network the sequence of indicator vectors is used both as input and output. Example: $[\mathbf{3}, \mathbf{4}, \mathbf{8}, \mathbf{5}, \mathbf{9}, \mathbf{9}, \mathbf{4}, \mathbf{4}, \mathbf{0}] \rightarrow [\mathbf{3}, \mathbf{4}, \mathbf{8}, \mathbf{5}, \mathbf{9}, \mathbf{9}, \mathbf{4}, \mathbf{4}, \mathbf{0}]$.

Encoder The encoder problem is a regression problem where the output is the integer sequence expressed as an vector. Note that the `<EOS>` isn't included and the vector is divided by 9 to fit between 0 and 1. It isn't strictly necessary for the output to be within $[0, 1]$, but it makes it easier to compare to the decoder version of this problem. Example: $[\mathbf{3}, \mathbf{4}, \mathbf{8}, \mathbf{5}, \mathbf{9}, \mathbf{9}, \mathbf{4}, \mathbf{4}, \mathbf{0}] \rightarrow [3/9, 4/9, 8/9, 5/9, 9/9, 9/9, 4/9, 4/9]$.

Decoder Finally one can consider a decoder version of the *copy* problem. Here the input should be between 0 and 1, as this will be used to initialize the hidden output $b_{h_1}^{0_d}$ and the cell state $s_{h_1}^{0_d}$. The cell in particular is almost always in the $[0, 1]$ interval. The input and output is the same as in the encoder case, just swapped. Example: $[3/9, 4/9, 8/9, 5/9, 9/9, 9/9, 4/9, 4/9] \rightarrow [\mathbf{3}, \mathbf{4}, \mathbf{8}, \mathbf{5}, \mathbf{9}, \mathbf{9}, \mathbf{4}, \mathbf{4}, \mathbf{0}]$

3.4.2 Validating implementation

To validate the Sutskever model the *full network* version of the *copy* problem was used. The model was configured as:

parameter	value
learning rate (η)	0.001
momentum (m)	0.9
decay (γ)	0.95
weight initialization	$\mathcal{N}(0, 0.5^2)$
clipping strategy	Euclidean
clipping value	10

Table 3.4.1: Initialization parameters.

In all cases the sigmoid function was used as the non-linear gate function (f) and the non-linear input function (g). The output activation function (h) was set to the identity function. Such that the LSTM unit can generalize to the simple RNN unit.

Note also that the target value wasn't reversed, this is to make the model easier to reason about. The original paper [3] also indicates that the model does work without reversing the target value, it just improves the performance.

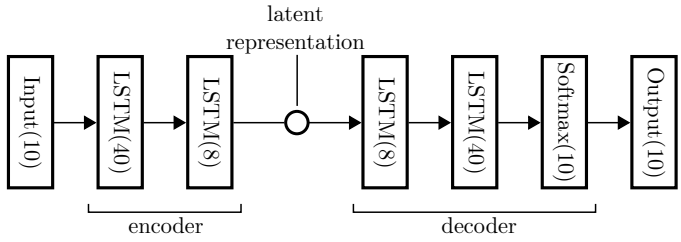


Figure 3.4.2: The architecture of the network.

The loss curve in Figure 3.4.3 shows that the implementation is capable of solving the copy problem and actually converges fairly nicely with a very minimal overfitting.

Whether or not the model solves it such that the encoder solves the *encoder copy* problem and the decoder solves the *decoder copy* is unknown. This is also irrelevant for validating the implementation. However the loss curves for the *decoder* and *encoder* problems are still shown in Figure 3.4.4, as they will later prove valuable as a reference.

Also note that while solving this problem validates the implementation, it doesn't guarantee an correct implementation, it just makes it likely that the implementation is correct.

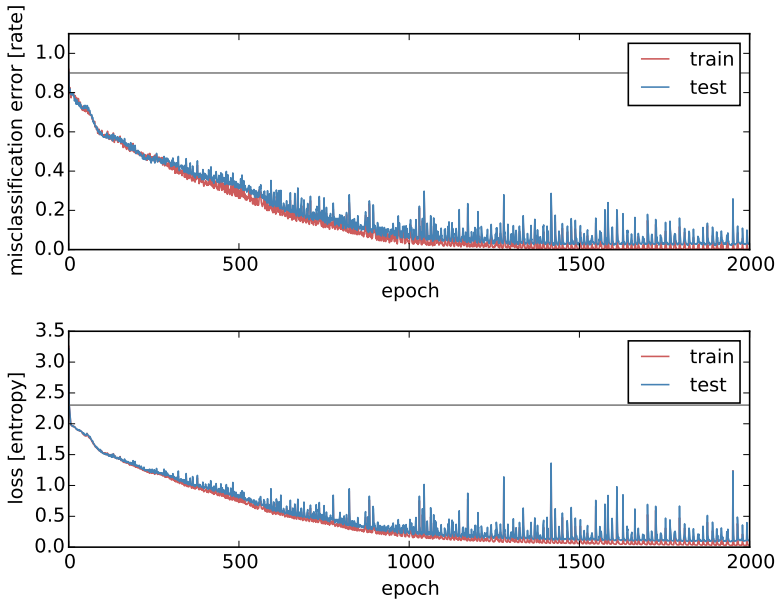
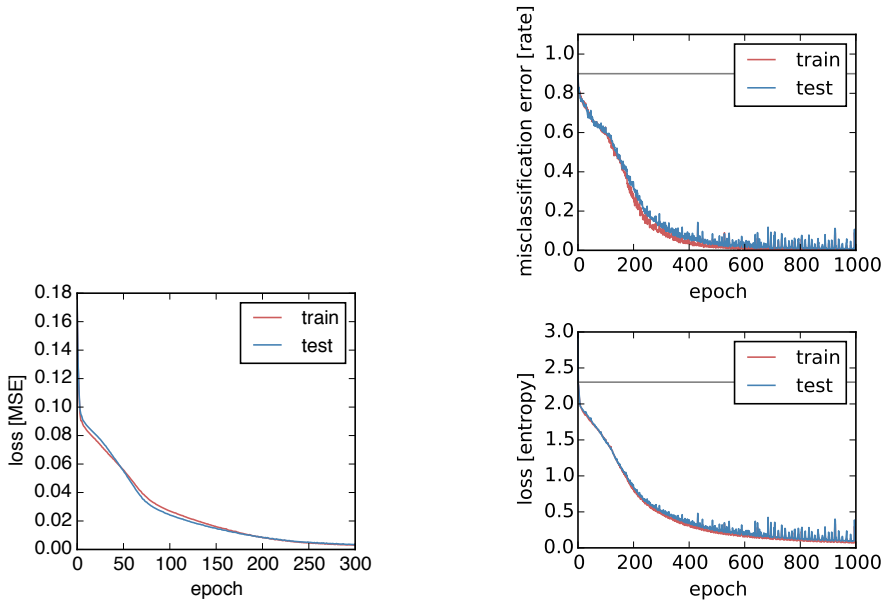


Figure 3.4.3: Loss and misclassification rate as a function of the number of epochs, for the full Sutskever network solving the "full network" copy problem.



(a) Encoder, solving the "encoder" copy problem.

(b) Decoder, solving the "decoder" copy problem.

Figure 3.4.4: Loss and misclassification rate as a function of the number of epochs, for the encoder and decoder as two independent neural networks.

The loss in Figure 3.4.3 and 3.4.4, is calculated as the mean over all observations and timesteps (words). The scale of the loss function is thus independent of the amount of observations and the sequence length.

3.4.3 Using real data

Now that the implementation is validated, it can be used on the real problem. The goal is to predict the article title given the article lead. The hope is this should be somewhat similar to the translation problem solved in the original paper [3]. Figure 3.4.5 shows the first 3 observations in the dataset. The entire dataset consists of 273813 news articles, all articles are used for training.

title:	Ukrainian President Yanukovych agrees early election
lead:	Ukrainian President Yanukovych calls an early election, as details emerge of a deal to end political crisis. Ukrainian President Viktor Yanukovych has agreed to an early presidential election as part of a deal to end the long-running crisis.

title:	UK floods: Damage 'could have been prevented'
lead:	Damage during the recent floods could have been prevented if the correct water management techniques had been used, says a group of experts.Some of the damage caused by the recent floods could have been prevented if the correct water management techniques had been used, says a group of leading environmental and planning experts.

title:	Five lose housing benefit cut appeal
lead:	Five disabled social housing tenants lose their Court of Appeal bid to have benefit cuts for those with spare bedrooms ruled unlawful. Five disabled social housing tenants have lost their Court of Appeal bid to have benefit cuts for those with spare bedrooms ruled unlawful.

Figure 3.4.5: Three examples on title (target) and lead (input).

It turns out that $\mathcal{N}(0, 0.5^2)$ has too high a variance for initializing the weights causing the gradients become not-a-number (NaN). To solve this $\mathcal{N}(0, 0.1^2)$ is used to initialize the weights, this is also what Alex Graves uses in his book [7].

This allows the training to continue for some iterations, to have it to continue for a longer duration with out getting not-a-number gradients the momentum and learning rate is also decreased. The final configuration is shown in Table 3.4.2 and Figure 3.4.6.

parameter	value
learning rate (η)	0.0001
momentum (m)	0.2
decay (γ)	0.95
weight initialization	$\mathcal{N}(0, 0.1^2)$
clipping strategy	Euclidean
clipping value	10

Table 3.4.2: Initialization parameters.

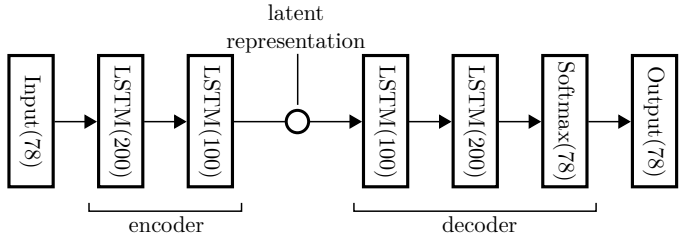


Figure 3.4.6: The architecture of the network.

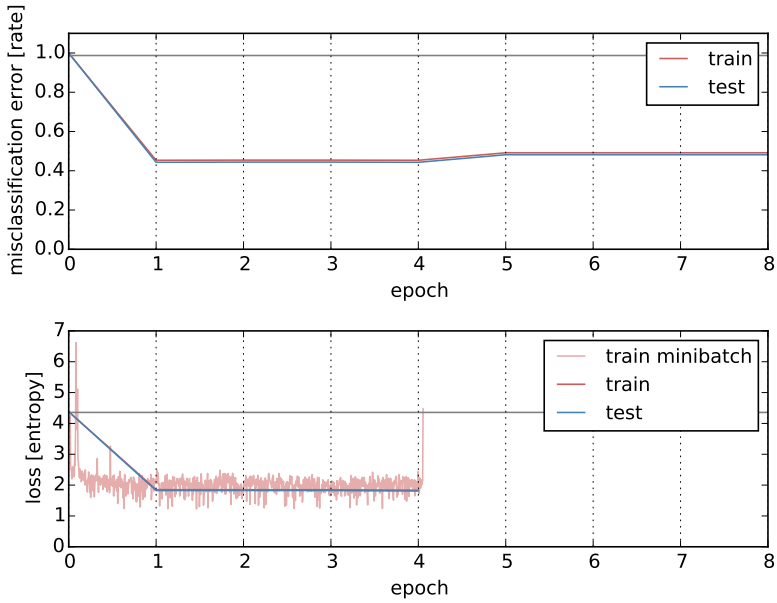


Figure 3.4.7: Loss and misclassification rate as a function of the number of epochs on the full Sutskever network.

Figure 3.4.7 shows that the model didn't converge after 4 epochs. Note that after approximately 4 epochs the gradients became not-a-number. The misclassification rate is only 0.5 so maybe there could be some meaning in the output. But from inspecting the prediction for the first 3 observations (Se Table 3.4.3) it is clear that isn't the case.

target	prediction
Ukrainian President Yanukovych agrees early election	`<EOS>`
UK floods Damage 'could have been prevented'	`<EOS>`
Five lose housing benefit cut appeal	`<EOS>`

Table 3.4.3: Target and prediction of the first 3 observations.

The model just predicts a long sequence of `<EOS>` which will match a fairly big part of the target, because the target is padded with `<EOS>` to fit the longest sequence in that mini-batch.

Each epoch contains 2140 mini-batches and takes about 3 hours to run. The 4 epochs that ran before the gradients became not-a-number thus took 12 hours. This should be enough time to find a better solution than just predicting `<EOS>`.

3.4.4 Diagnosing the problem

It is possible that the model is just stuck in a local minima. However it turns out that decreasing the standard deviance for the weight initialization in the *copy* problem causes similar issues (see Table 3.4.4 for the full configuration). In this problem the sequences are all equally long before padding, so just predicting `<EOS>` isn't a naturally good choice. However the model predictions are still independent of the input and somewhat constant just like in the real data case. For example $[1, 1, 1, 1, 7, 7, 7, 7, 0]$ is always the prediction given a seed of 42.

However it turns out that the decoder and encoder themselves converges just fine, though it is not as smooth as when $\mathcal{N}(0, 0.5^2)$ is used for weight initialization.

parameter	value
learning rate (η)	0.001
momentum (m)	0.9
decay (γ)	0.95
weight initialization	$\mathcal{N}(0, 0.1^2)$
clipping strategy	Euclidean
clipping value	10

Table 3.4.4: Initialization parameters for the copy problem with $\sigma = 0.1$.

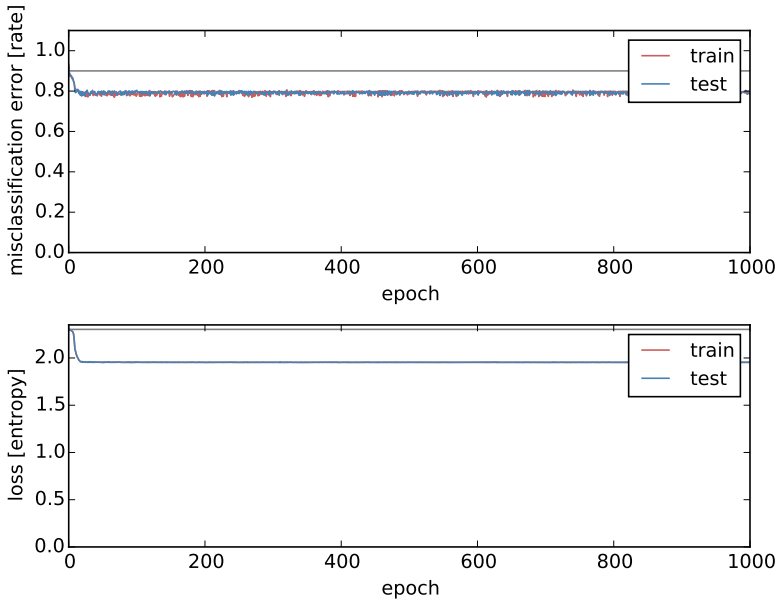
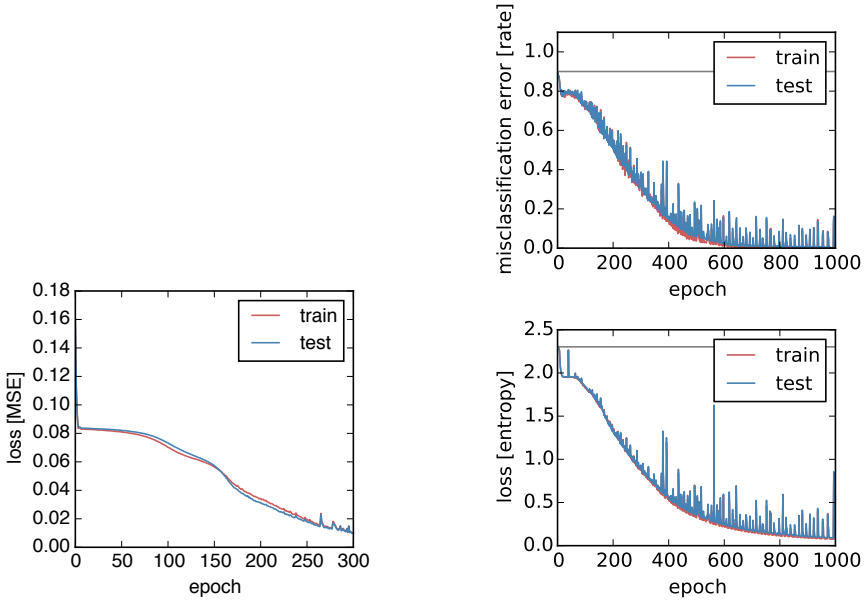


Figure 3.4.8: Loss and misclassification rate as a function of the number of epochs on the full Sutskever network.



(a) Encoder **(b)** Decoder

Figure 3.4.9: Loss and misclassification rate as a function of the number of epochs.

There is a lot of spikes in the encoder and decoder loss curves (Figure 3.4.9) these could probably have been avoided by choosing a smaller *clip* value. Likewise it is entirely possible that some other hyperparameter choices could have caused the full network to converge probably. To investigate that possibility a grid search over the hyperparameters was performed.

parameter	value(s)
learning rate (η)	[0.0001, 0.001, 0.01, 0.1, 0.2]
momentum (m)	[0, 0.2, 0.9]
decay (γ)	[0.9, 0.95]
weight initialization	$\mathcal{N}(0, 0.1^2)$
clipping strategy	Euclidean
clipping value	[1, 5, 10, 50]

Table 3.4.5: Tried parameter values for solving the *full network copy* problem.

parameter	best	worst
learning rate (η)	0.01	0.1
momentum (m)	0.9	0
decay (γ)	0.9	0.9
weight initialization	$\mathcal{N}(0, 0.1^2)$	$\mathcal{N}(0, 0.1^2)$
clipping strategy	Euclidean	Euclidean
clipping value	5	10

Table 3.4.6: Best and worst choose for hyperparameters, given the grid search in Table 3.4.5.

While there are good and bad choices none of them makes the full network converge, as seen in Figure 3.4.10 and 3.4.11.

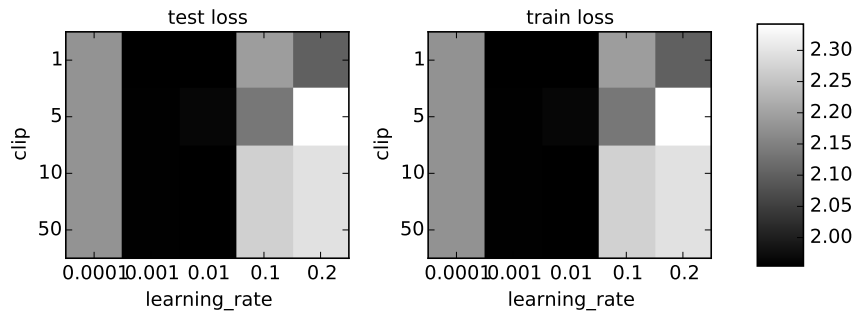


Figure 3.4.10: Grid search on learning rate and clip, other parameter variations are meaned out.

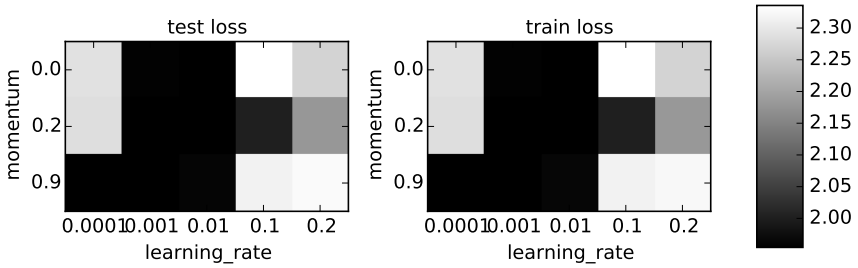


Figure 3.4.11: Grid search on learning rate and momentum, other parameters variations are meaned out.

The encoder and decoder themselves can converge and hyperparameter optimization of the full network doesn't work. All this suggests that it is the combination of the many layers and the smaller standard deviance that causes the problem.

3.4.5 Vanishing gradient

It is not unusual that the combination of deep networks and small initial weights causes problems. This is because of the vanishing gradient problem. This is particular a problem in recurrent neural networks, because the iteration over the sequence actually causes the network to be much deeper than it appears to be. Using LSTM units solves this to some extent, however having many layers is still an issue. This is best seen by just considering a simple feed forward neural network with many layers and following the δ calculations.

Because the last layer is a softmax and one can expect and uniform distribution as the initial distribution and the target is often zero, δ_{L+1} can be approximated as $\frac{1}{10}$ (there are 10 classes). The remaining δ values are then calculated as:

$$\delta_{h_\ell} = \theta'(a_{h_\ell}) \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \delta_{h_{\ell+1}} w_{h_\ell, h_{\ell+1}} \tag{3.4.1}$$

The maximum value of $\theta'(\cdot)$ when θ is the sigmoid function is approximately 0.23, and one shouldn't expect more than 0.1 from the weights given the initialization. One then gets:

$$\delta_{h_L} \approx 0.23 \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \frac{1}{10} \cdot 0.1 \lesssim \delta_{h_{L+1}} \approx \frac{1}{10} \tag{3.4.2}$$

The δ_{h_ℓ} will thus decrease for each layer causing the gradients for the bottom layers to be near zero.

Note that because the size of δ_{h_ℓ} depends on $H_{\ell+1}$, one could use different standard deviance values depending on the $H_{\ell+1}$ value for each layer, such that the δ_{h_ℓ} value is in a reasonable range. This was attempted and didn't work, though more experiments could have been done. Another solution is to use rectified linear unit (ReLU) as the activation function, which as been shown not to suffer from the vanishing gradient problem. Due to the lack of time this was not attempted.

CHAPTER 4

Conclusion

In this thesis 3 models for creating document vectors have been analyzed for the purpose of clustering articles related to the same story. While this is a very specific application, the performance of the models relates to their ability to create good vector representations of documents in general.

Skip-gram The skip-gram model which is the most primitive and oldest of the three model, actually turned out to produce surprisingly good document vectors. It is not that the document vector are amazing and solves the problem well. But given that the purpose of the skip-gram model is to find good word vectors and not document vectors the performance is surprisingly good. It is possible that a more clever weighted sum could improve the result further, for example by weighting nouns higher. But as the paragraph2vec paper [2] also states, the performance will always be limited by the fact that the purpose of the model is to find word vectors and not document vectors.

paragraph2vec While the skip-gram model was surprisingly good, the paragraph2-vec was surprisingly bad. According to the paragraph2vec paper [2], this model should be better than the usual bag-of-word models and should produce a much better document vector, than just taking the mean of the word vectors. Unfortunately this does not seem to be the case. A possible explanation is that the document vectors that the model produces doesn't represent the semantic meaning very well, but rather the mood or political point of view of the document. The reasoning here is that semantic meaning is likely found in the short context, which may best be predicted by the surrounding words, while the mood is found in the larger context which is where the document vectors help in the prediction. It is also entirely possible that the model work better with short documents like sentences, thus using the full article text isn't the optimal choice. While that is said the paragraph2vec paper [2], does claim to work on documents of any length. The effect of the document length should be investigated more.

Sutskever The Sutskever model did unfortunately not converge, why this is the case is unknown. There are some indications that it is a vanishing gradient problem. Using a less naïve initialization or rectified linear units may solve this problem.

However it may be more complicated than that. The weight gradients became easily not-a-number for no apparent reason. This indicates the implementation is not numerical stable and indeed Theano does have issues with numerical stability, particular when involving `scan`, `softmax` and `log` together [17]. If these issues where solved it is likely that the implementation of the Sutskever model would converge. In the original paper [3] they implemented their own version from scratch in order to use 8 GPUs in parallel. Likely they have also used various tricks, to make the implementation more numerical stable than what Theano can currently do.

It is worth remembering that the implementation actually do work. The implementation was verified by constructing a synthetic problem, which have 9^8 possible input and output sequences. However the training set was much smaller than this (1280 observation) and both test and training curves converged to zero error. This means the network is not only able to memorize an output given an input, but actually generalize really well. It is just that for complex problems, getting the initialization right is hard. However there is no reason to think it isn't possible to get the initialization right and thereby having it solve the real problem as well.

If the Sutskever model would converge probably and give good title predictions given the article lead, then it is reasonable to expect the document vector would capture the meaning of the article. Whether or not those vectors would be better than what bag-of-words methods can produce is unknown. If this is the case it could be because of the long sequences. In the original paper [3] it was shown that the model started to perform poorly for long sequences. In this thesis letters was used for input encoding thus adding to the sequence length.

Future work If the Sutskever model predicts the article title well given the lead, but still doesn't produce good document vectors, then one would have to look into different strategies. For example, instead of predicting the title given the article lead, one could try an autoencoder, which would reconstruct the full document. It is also possible that more advanced clustering method are required in order to achieve good results.

However it is entirely possible that the problem can't be solved using unsupervised methods, without getting a high error rate. In that case semi-supervised methods may be a good compromise. Manually labeling all stories in just 100000 articles certainly seem like an impractical huge task.

APPENDIX A

Vector notation of Skip-Gram

To get an idea about how a neural network representation relates to the equations in [1], the scalar notation from the Feed forward Neural Network section, is replaced with a similar vector notation.

As previously discussed the input and output words are represented using 1-of-V encoding, this vector will be denoted as $\mathbf{x}_w$ for both the input and output words w. Note that because words are used as input, there is a finite amount of possible inputs. The activation is thus not denoted by the input index n but rather the input value w. For referring to the word at position t in the corpus w_t is used.

The activation in the hidden layer is:

$$\mathbf{b}_{1,w_t} = \mathbf{a}_{1,w_t} = \mathbf{W}_1 \mathbf{x}_{w_t} \tag{A.0.1}$$

Here the subscript 1 denotes that this is for the layer 1 (the input is layer 0). The output before the softmax is applied is then calculated as:

$$\mathbf{a}_{2,w_t} = \mathbf{W}_2^T \mathbf{b}_{1,w_t} \tag{A.0.2}$$

Here $\mathbf{W}_1$ is the matrix containing the weights on the left side of the log-linear network, thus it has a size of $D \times V$. $\mathbf{W}_2$ is the matrix containing the weights on the right side and have also size $D \times V$.

$\mathbf{a}_{2,w_t}$ is a vector containing information about all possible surrounding words, but since this is used in a softmax and likelihood setting, only the value associated with the output word $w_{t+\ell}$ is of interest. To get this specific value the encoding vector $\mathbf{x}_{t+\ell}$ can be used. In general the value associated with the word w can be obtained using:

$$a_{w_t,w} = \left(\mathbf{a}_{2,w_t}\right)_w = \mathbf{x}_w^T \mathbf{a}_{2,w_t} \tag{A.0.3}$$

This detail is not particular important, but explains the some of the notation used in [1].

The probability $p(w_{t+\ell}|w_t)$ can now be calculated using a softmax:

$$\begin{aligned} p(w_{t+\ell}|w_t) &= \frac{\exp(a_{w_t,w_{t+\ell}})}{\sum_{w=1}^{V}\exp(a_{w_t,w})} = \frac{\exp(\mathbf{x}_{w_{t+\ell}}^T\mathbf{a}_{2,w_t})}{\sum_{w=1}^{V}\exp(\mathbf{x}_w^T\mathbf{a}_{2,w_t})} \\ &= \frac{\exp(\mathbf{x}_{w_{t+\ell}}^T\mathbf{W}_2^T\mathbf{W}_1\mathbf{x}_{w_{t+\ell}})}{\sum_{w=1}^{V}\exp(\mathbf{x}_w^T\mathbf{W}_2^T\mathbf{W}_1\mathbf{x}_{w_{t+\ell}})} \\ &= \frac{\exp(\left(\mathbf{W}_2\mathbf{x}_{w_{t+\ell}}\right)^T\mathbf{W}_1\mathbf{x}_{w_{t+\ell}})}{\sum_{w=1}^{V}\exp((\mathbf{W}_2\mathbf{x}_w)^T\mathbf{W}_1\mathbf{x}_{w_{t+\ell}})} \end{aligned} \tag{A.0.4}$$

From the above expression it's seen that $a_{w_t,w_{t+\ell}}$ is really an inner product between two linear vector transformations. This is similar to how [1] represents the model. Specifically if one sets: $w_O = w_{t+\ell}$, $w_I = w_t$, $\mathbf{v}_{w_O} = \mathbf{W}_2\mathbf{x}_{w_{t+\ell}}$, $\mathbf{v}_w = \mathbf{W}_2\mathbf{x}_w$ and $\mathbf{v}_{w_I} = \mathbf{W}_1\mathbf{x}_{w_{t+\ell}}$, then [1, eq. 2] is obtained:

$$p(w_O|w_I) = \frac{\exp(\mathbf{v}_{w_O}^T\mathbf{v}_{w_I})}{\sum_{w=1}^{V}\exp(\mathbf{v}_w^T\mathbf{v}_{w_I})} \tag{A.0.5}$$

APPENDIX B

Implementation details

The Sutskever model implementation used in [3] is not open source and there are no libraries for creating such a network. The model is mathematically quite complicated and maybe even more complex from a computational perspective. Thus it is unfeasible to implement it as a simple top-to-bottom script. Having a framework for constructing the LSTM layer, backward pass and etc. is thus a good idea.

There is a fork[1] of the Lasagne framework there do provide implementation of the simple RNN and LSTM layers, from which one can build a deep network. However Lasagne is build for feed forward networks and thus only simplistic RNN networks fits into the framework. The Sutskever model have its output layer connected to input layer of the decoder, which is currently far too complex for Lasagne to handle [18].

The framework implemented in `https://github.com/AndreasMadsen/bachelor-code` uses a different approach for generating the network equations, which allows for complex networks like the Sutskever model. The trade off is that the framework is less general and some optimizations like loop-invariant code motion can't easily be manually implemented. However as Theano becomes more advanced at automatic optimization, optimizations such as loop-invariant code motion shouldn't be necessary to manually implement.

B.1 Framework architecture

The overall strategy of the framework, is to have a consumer class that consumes some generated layers and handles the forward pass, backward pass and optimization. Here is an example of a simple one-to-one RNN classifier.

```
lstm = neural.network.Std()

# Setup layers for a recurrent classifier model
lstm.set_input(neural.layer.Input(2))
lstm.push_layer(neural.layer.LSTM(4))
lstm.push_layer(neural.layer.Softmax(4))

# Setup loss and optimizer
```

[1] `https://github.com/craffel/nntools`

```
lstm.set_loss(neural.loss.NaiveEntropy())
lstm.set_optimizer(neural.optimizer.Momentum())

# Compile train, test and predict functions
lstm.compile()

# Train and predict using
lstm.train(input, target)
lstm.predict(input)
```

The `lstm.set_input` and each `lstm.push_layer` appends to a dynamic list which contains all the layers. The integer argument in each layer constructor, specifies the amount of nodes in that layer. To generate the weight matrices between layers each `lstm.push_layer` call, also calls a `layer.setup` method with the size of the previous layer as an argument.

Do note that this is just for a simple sequence to sequence classification system with a one-to-one alignment. For a complex network like the Sutskever model, a similar consumer class exists.

In both cases the consumer class then implements a `network.forward_pass` method, which does the scanning over all time steps. For each time step a loop over the list of layers is performed.

```
all_outputs = []
curr = 0
prev_output = x_t

# Loop through each layer and send the previous layers output
# to the next layer. The layer can have additional parameters, if
# taps where provided using the `outputs_info` property.
for layer in self._layers[1:]:
    taps = layer.infer_taps()
    layer_outputs = layer.scanner(prev_output, *args[curr:curr +
    taps])

    curr += taps
    all_outputs += layer_outputs
    # the last output is assumed to be the layer output
    prev_output = layer_outputs[-1]

return all_outputs
```

The `layer.infer_taps` and `all_outputs` logic exists such that each layer is responsible for its own state, typically the hidden output and the cell state of the previous time iteration. For transferring the hidden output to the next layer, the rule is that the last output value of `layer.scanner` is the input in the next layer.

Do also note that the scan over all time steps uses the `theano.scan` function, which only executes the above code once. This is because Theano dynamically generates a computational graph, from which it will later compile the C and CUDA code, which is what includes the `for`-loop over each time step. This also means that the layer loop is by design unrolled as it is not directly implemented in Theano.

B.2 Memory and computational tricks

Mathematically each word or letter is encoded as an indicator vector. However from a memory perspective this is extremely inefficient. In the Sutskever training 248192 articles was used for training, there are 78 unique letters and the max sequence length is 1000 letters for the input and 197 for the target. The worst case scenario in terms of memory usage is thus:

$$248192 \cdot 78 \cdot (1000 + 197) \approx 21.6 \text{ GB} \tag{B.2.1}$$

This is quite a lot of memory, the DTU HPC system can easily handle it, but memory transferring onto a GPU is an slow operation and it's also a waste of the CPU cache.

The solution is to not encode each letter as an indicator vector but just as an integer which specifies the index of the 1-element. This of course requires rewriting some of the equations. By doing this the memory usage becomes:

$$248192 \cdot (1000 + 197) \approx 0.06 \text{ GB} \tag{B.2.2}$$

The input is only multiplied by the weight matrices in the first hidden layer. If one defines $\mathbf{I}(i)$ as an an indicator vector where the 1-element is located at index i, one can use that $\mathbf{I}(i)\mathbf{W} = \mathbf{W}_i$. In terms of Theano and the entire mini-batch `T.dot(x, W)` becomes `W[x, :]`, which as a side effect is also more computational efficient.

The target values is only used in the entropy loss function, which can be expressed as:

$$\mathcal{L} = -\sum_{k=1}^{K} \mathbf{I}(i)_k \ln(y_k) = -\ln(y_i) \tag{B.2.3}$$

Again this is also more computational efficient.

Another optimization which don't improve memory utilization, but is more computationally efficient, is to combine the weight matrices in each LSTM layer. Each LSTM layer have 4 input to hidden matrices and 4 hidden to hidden matrices. Because these are all multiplied by the same values, they can be combined to just one input to hidden matrix and one hidden to hidden matrix. This means that only two BLAS calls are required pr. layer.

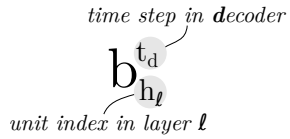
APPENDIX C

Notation

symbol	meaning
a	The activation input, calculated as a weighted sum over the input.
b	The activation output, $b = \theta(a)$.
H	The amount of units in the layer.
h	The index of a units in the layer.
K	The amount of classes calculated in the softmax, $K = H_{L+1}$.
k	The class index used in the softmax, $k = h_{L+1}$.
$\mathcal{L}$	The loss function (should always be minimized).
L	The amount of hidden layers. Including the softmax output layer there are $L + 1$ layers.
ℓ	The layer index.
m	The hyperparameter *momentum* used in gradient descent.
s	The cell state used in a LSTM unit.
t	A target value. If used in a superscript it is the sequence index.
w	A weight used in a neural neural network.
x	The input to the neural network, $x = b_{h_0}$.
y	The softmax output.
γ	The hyperparameter *decay rate* used in RMSprop gradient descent.
δ	Bookkeeping value used in backward propagation.
η	The hyperparameter *learning rate* used in gradient descent.
ρ	Indicates the input gate in a LSTM unit.
ϕ	Indicates the forget gate in a LSTM unit.
ω	Indicates the output gate in a LSTM unit.
θ	The non-linear activation function.

Table C: Meaning of commonly used symbols.

Note that subscript and superscript may be added to indicate the layer and time step.



Bibliography

[1] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546.

[2] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053.

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215.

[4] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: New features and speed improvements," *CoRR*, vol. abs/1211.5590, 2012. [Online]. Available: http://arxiv.org/abs/1211.5590.

[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," S. van der Walt and J. Millman, Eds., pp. 3 –10, 2010. [Online]. Available: http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781.

[7] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*, ser. Studies in Computational Intelligence. Springer, 2012, ISBN: 9783642247972.

[8] C. M. Bishop, *Pattern Recongnition and Machine Learning*. Springer, 2009, ISBN: 9780387310732.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, ser. Springer series in statistics. Springer, 2009, ISBN: 9780387848587. [Online]. Available: http://statweb.stanford.edu/~tibs/ElemStatLearn/.

[10] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850.

[11] Y. Goldberg and O. Levy, "Word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014. [Online]. Available: `http://arxiv.org/abs/1402.3722`.

[12] M. Davis and K. Whistler, *Unicode normalization forms*, 2014. [Online]. Available: `http://unicode.org/reports/tr15/`.

[13] Wikipedia, *Hierarchical clustering — wikipedia, the free encyclopedia*, 2015. [Online]. Available: `http://en.wikipedia.org/wiki/Hierarchical_clustering`.

[14] Python Software Foundation, *Python, version 3.4*, 2015. [Online]. Available: `https://www.python.org`.

[15] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," English, in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, Valletta, Malta: ELRA, May 2010, pp. 45–50. [Online]. Available: `http://is.muni.cz/publication/884893/en`.

[16] W. Zaremba and I. Sutskever, "Learning to execute," *CoRR*, vol. abs/1410.4615, 2014. [Online]. Available: `http://arxiv.org/abs/1410.4615`.

[17] A. Madsen and P. Lamblin, *Numerical unstable log(softmax(·))*, 2015. [Online]. Available: `https://github.com/Theano/Theano/issues/2781`.

[18] G. Yuan, C. Raffel, A. Madsen, and S. K. Sønderby, *Should we include sequence prediction ability using single input for lstm (or generally rnn)*, 2015. [Online]. Available: `https://github.com/craffel/nntools/issues/15`.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ