Natural Language Processing and Deep Learning

Matthias Nitsche
matthias.nitsche@haw-hamburg.de
Hamburg University of Applied Sciences, Department of Computer Science
Berliner Tor 7
20099 Hamburg, Germany

***Abstract*—Natural Language Processing is the application of computational techniques to the analysis and synthesis of natural language and speech. Deep learning is a class of machine learning algorithms that learn to represent (hierarchical) features through multiple layers of non-linear activation. Various architectures exist to represent language in manifolds of hidden layers with convolutional neural networks (CNN), recurrent neural networks (RNN) and attentional feedforward neural networks (ANN). In this paper we will briefly glance over the basics and provide a small road map up to advanced concepts. Furthermore, state of the art models in deep learning and their relevance to NLP are presented.**

***Keywords*—Natural Language Processing, Deep Learning, Machine Learning, Neural Networks**

I. Introduction

The goal of every natural language processing application is to capture implicit or explicit meaning or properties from text. A lot of these applications such as text tokenization, named entity recognition or sentiment analysis have gotten to a very high quality standard although there is always room for improvements. State of the art systems for text understanding are tackled with a variety of different approaches. In this paper we would like to compare and present several neural network types. They include

1) Recurrent neural networks (RNN)

   RNNs have become the baseline models for text understanding and are often used with sequential data. Instead of back propagation, an enhanced more generalized version back propagation over time is applied. Due the vanishing gradient problem Long short-term memory networks (LSTM) by [Hochreiter and Schmidhuber, 1997] in combination with attention mechanisms are favored.

2) Convolutional neural networks (CNN)

   Up to now CNNs were largely used for vision and audio tasks. This has changed and there are character, word and sentence level CNN architectures for text understanding, beating state of art LSTMs with fewer training time and a smaller parameter space.

3) Memory/Attention networks

   Attention is commonly known as a memory mechanism used to additionally shift attention of ANNs to specific parts of a sequence. Recently [Kaiser et al., 2017] published the Transformer, beating state of the art CNN and LSTM architectures on text understanding tasks. The Transformer makes use of a mechanism called multi-head attention and is a memory neural network.

Neural networks are used in several different large scale language processing domains ranging from neural machine translations, question answering, sentiment analysis to neural programming. Inputs are sentences, questions or commonly called queries, stated in natural language. Inputs are mapped to a suitable feature space that represents language in relative counts often leveraging contextual information of a given word. Therefore most text understanding tasks concern them self with sentences and paragraphs. With neural networks the sequences are often encoded with an encoder network and must be decoded to the specific task on a decoder network. Encoder-decoder networks are two standalone neural networks that represent end-to-end learning approaches.

II. Basics

In this section we will glance over the major concepts of deep learning and natural language processing. Deep learning refers to a class of algorithms that work with multiple layers of non-linear activation functions. In this paper we will focus on the most well known deep learning algorithms namely the artificial neural network (ANN). The task behind natural language processing is mostly to infer properties and meaning from text that is explicitly and implicitly stated. Text is almost always a subject to interpretation and there are often multiple meanings to the same text. This basic chapter heavily relies on the deep learning book by [Goodfellow et al., 2016].

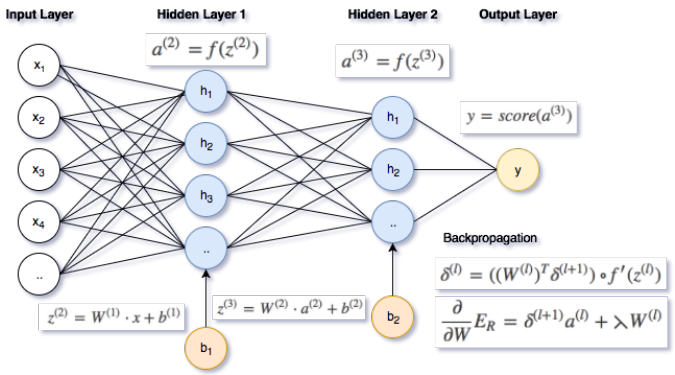


Fig. 1. Feed forward neural network

ANNs typical architectures, as seen in figure 1, consist of an input layer, one to many hidden layers, an output layer

and a specified cost function. In theory it should be sufficient to have more than two layers before a neural network is considered deep. Adding more hidden units to each layer makes a neural network broader, adding more hidden layers makes it deeper. Each hidden unit of a layer is connected to all units of its foregoing layer and the upcoming one. Each single unit is composed of one to many non-linear activation functions such as Thanh, ReLU or sigmoid. Each of these functions have different properties depending on the task at hand. The most popular non-linear activation function is the Rectified Linear Unit (ReLU), which yields the maximum or zero of an input $max(0, input)$. Just like in the nervous system of a human, a neuron corresponds to a hidden unit and might be activated due to its incoming potential which in turn releases a potential to the next connected neuron. Predicting the path of a given input for a neural network up to the output layer is currently impossible.

The output layer is directly connected to a cost function that evaluates the outcome with a golden standard called label. Building up the potential through a neural network up to the cost function is called forward propagation. In order to actually learn from the forward propagation we have to back propagate what is referred to as error with a procedure called back propagation. Back propagation is an algorithm that makes use of a procedure called gradient descent that is responsible for propagation the error of the cost function with respect to each layers gradient, back to the input layer. Mathematically speaking each layer is associated with a matrix of weights W, data input x and a bias b, that governs the potential activation threshold. Back propagation is essentially an update of the Jacobian weight matrices with respect to the error gradients of the foregoing layer and the total error calculated by the cost function $\hat{y}$.

A. Input layer

The input represents the feature space of a data point. If sentences are classified for sentiment, each word is a feature. Sentences are structural constructs that are governed by a grammar. From these grammars statistical conclusions can be drawn such as cooccuring words. A sentence alone cannot be easily classified. Thus there must be several hundred of million sentences for a good performance. Language is morphologically rich and more words are added each day, while grammar typically stays constant. As words cannot be interpreted by computers they are transformed into embedded vectors, mapping words into a numerical feature space. The feature space can range from one-hot encoding, coocurrence matrices to semantic indexing models or topic models with Latent Semantic Indexing [Deerwester et al., 1990] or Latent Dirichlet Allocation [Blei et al., 2003]. State of the art approaches however use neural language models such as word2vec [Mikolov et al., 2013], GloVe [Pennington et al., 2014] or seq2seq [Sutskever et al., 2014] mapping words to a hidden feature vector of dimensionality n taking word windows of size k.

B. Hidden layer

Hidden layers are the parts of ANNs that map the input to some output. On the way depending on the type of neural network in use, each hidden layer learns aspects of the input data. CNN features are hierarchically ordered, starting from dense input data, diminishing the size of layers at each step. Each hidden layer is associated with a activation function that must be differentiable. This stems from the fact that during backpropagation the weights connecting the hidden layers must be updated with respect to the error gradient of the cost function. Secondly it is said that the popular activation functions such as tanh, ReLU or sigmoid are non-linear. This stems from the fact hat systems involving these functions cannot be written as a linear combination of the unknown variables or functions. In machine learning the objective goal is often a linear approximation of the form $W \cdot x + b$, where activation functions render the system of equations non-linear. There is no definitive answer which function has the best properties. As such using a non-linear activation function is subject to experimentation for the model and dataset in use.

C. Output layer

The output layer needs to prepare the potential that was activated within the network to scale and normalize it to a particular cost function. In a classification task the output is often a multinouli distribution inferred by a softmax. [Goodfellow et al., 2016] There are a few standardized outputs given the task. This is often a step that is highly dependent of the task and cannot be automatically inferred by some parameters alone (such as hidden layers). The output layers dimensions must match those that the cost function consumes for instance multi-class classification needs k output channels.

D. Cost function

Cost functions are the objective goals of a network. Whatever forward and back propagation calculate and update is entirely related to the loss of the cost function. The objective can be stated either as maximizing the likelihood or minimizing a loss. For multi-class classification we typically use the softmax. The softmax normalizes all the incoming edges summing to 1, calculating the partial probabilities that some node is of class k.

$$P(y = j \mid x) = \frac{exp(x^T \cdot w_j)}{\sum_{k=1}^{K} exp(x^T \cdot w_k)} \tag{1}$$

Representing text in the form of probabilities in a context window, taking k words before and after the current head word creates a semantic vector space with relative dependencies. Models such as word2vec use an unsupervised window based approach. The context of a given word w is chosen with a window k such that $w_{i-k}, \ldots, w_i, \ldots w_{i+k}$ is satisfied. This objective of a window is generally called skip-gram. The objective cost function then is

$$P(w_O \mid w_I) = \frac{exp(v'^T_{w_O} \cdot v_{w_I})}{\sum_{w=1}^{W} exp(v'^T_w \cdot v_w)} \tag{2}$$

Here W is the whole vocabulary and v' and v are the input and output vector representations. The above equation can be read as the probability that a word w_O is likely given word w_I. The summation term in the denominator can be intractable when W gets large. Thus there are sampling based approaches such as the hierarchical softmax or noise contrastive estimation (NCE) that improve speed by a large margin. Strictly speaking the later does not satisfy a full language model as they approximate the true joint probabilities with random sampling and negative feedback procedures. [Mikolov et al., 2013]

E. Forward propagation

Forward propagation approximates a given task with its input signal. That means the neural network tries to infer a set of parameters that it assumes to be correct given the input. The forward pass then becomes

$$\begin{aligned} h^{(2)} &= f(W^{(1)} \cdot x + b^{(1)}) \\ h^{(3)} &= f(W^{(2)} \cdot h^{(2)} + b^{(2)}) \\ \hat{y} &= \frac{1}{n} \cdot \sum_{i=1}^{n} \mid y_i - h_i^{(3)} \mid^2 \end{aligned} \tag{3}$$

where x is the input and W the weight matrix between x and the first hidden layer activation $h^{(2)}$. $h^{(2)}$ is then feed as output to the third hidden layer $h^{(3)}$. The activation functions f must be differentiable and are often non-linear, like tanh, ReLU or sigmod.

$$h^{(2)} = f(W^{(1)} \cdot x + b^{(1)}) \tag{4}$$

Normally, weights are initialized with a uniform distribution or any suitable prior distribution. $\hat{y}$ can be calculated using any cost function that is able to approximate the given task well.

F. Back propagation

Back propagation updates the weight parameters W with the error gradients $\frac{\partial}{\partial W} E_R$ of the loss $\hat{y}$ after the forward pass. In order to approximate the error we need to compute all first derivatives (Jacobians) of the weights with respect to the activation $f(\dots)$ and the error gradients. This can be achieved efficiently with automatic differentiation and graph based back propagation algorithms computing the Jacobians and memorizing results where needed. As parameters can go into the billions there needs to be a trade off between computing the derivatives and caching them. Fortunately most state of the art numerical computation libraries such as Tensorflow or Cafe support this. Back propagation works in several steps.

$$\begin{aligned} \hat{y} &= \frac{1}{n} \cdot \sum_{i=1}^{n} \mid y_i - h_i^{(3)} \mid^2 \\ \frac{\partial}{\partial W^{(2)}} E_R &= \hat{y} \cdot h^{(2)} + \lambda \cdot W^{(2)} \\ \delta^{(2)} &= ((W^{(2)})^T \cdot \hat{y}) \circ h_2' \\ \frac{\partial}{\partial W^{(1)}} E_R &= \delta^{(2)} \cdot h^{(1)} + \lambda \cdot W^{(1)} \\ \delta^{(1)} &= ((W^{(1)})^T \cdot \delta^{(1)}) \circ h_1' \end{aligned} \tag{5}$$

First the loss is computed between output and the labels. After this the weights of layer l are updated with respect to the error gradient, often referred to as delta rule. The error for this layer is computed and propagated to the prior layer with respect to the error gradient. This repeats until we are at the input layer. h_l' is the derivative of the activation function. λ is a regularization parameter controlling the size of the weights. Back propagation requires that the non-linear activation functions must be differentiable. The weights are then updated using gradient descent with respect to the total error of the cost function.

$$W_{n+1} = W_n - \eta \frac{\partial E}{\partial W_n} \tag{6}$$

Essentially gradient descent is back propagation with an additional parameter η , controlling how large the update steps of the gradients will be. Note that in this instance the objective is to minimize the weights W by the rate of change of the gradients.

III. Statistical Learning

In statistical learning theory we try to find some notion of truth in highly uncertain environments, explaining effects and causes by measures of information gain called entropy. To explore this field we would like to find some function F that maximizes our predictions given explanatory variables x and predictor variables y sampled from a distribution $p(x; y)$ such that $F(x, y)$converges to the true distribution as it grows to infinity. Entropy is best described by the KullbackLeibler divergence D_{KL} measuring the information gain between two distributions.

$$D_{KL}(P \mid\mid Q) = \sum_i P(i) \cdot log \frac{P(i)}{Q(i)} \tag{7}$$

D_{KL} can be read as, the information gain when the probability distribution of Q is replaced with the distribution P. It is thus the expected (average value) of the difference between probability P and Q multiplied with the expectation of P. The expected value gives you the average outcome of a given distribution with probabilities $p_1, \cdots, p_k$ and the values $X = x_1, \cdots, x_k$

$$\mathbf{E}[X] = x_1 p_1 + x_2 p_2 + \cdots + x_k p_k \tag{8}$$

As x goes to infinity the expected value becomes the average value of the true underlying distribution. This notion of expected values helps best in describing the theoretical or underlying limits of what can be expected of a neural network. The D_{KL} can be approximated by maximizing a likelihood function. The maximum likelihood estimation (MLE) is one of the most used likelihood estimators (more commonly the negative log likelihood). In theory we try to find a set of parameters for a given distribution that is maximized compared to a set of other parameters. The maximum a posteriori estimation (MAP) is the generalization where a uniform prior is assumed.

$$\hat{\ell} = \frac{1}{n}\prod_{i=1}^{n} ln\ f(x_i \mid \theta),$$
$$MLE(\hat{\theta}) = \underset{\theta\in}{argmax}\ \hat{\ell}(\theta; x_1, \ldots, x_n) \tag{9}$$

A. Training error vs Generalization error

In machine learning there are two losses that are of particular interest. The first is called training error and refers to the error margin that is objectively maximized/minimized during training with your training data. The second is the generalization error that quantifies how well your trained classifier works on different data. The underlying reason why there is a disconnect between both measurements is that we always assume, that we fit our data $x \sim p(x)$ to a model F such that y is produced.

$$F : (x \in R) \rightarrow (y \in R) \tag{10}$$

Now two problems arise. First we assume a connection between x and y which must not be the case. Second we assume that x is drawn i.i.d. from some distribution $p(x)$ and F is a predictor of $p(x)$ which must also not be the case. Assuming however that F is able to fit y with x our goal is to minimize the training error. Let us assume that we have a trained model F, how can we make sure that it fits new unseen data z $p(z)$ well? If we fit z with a trained model F we try to minimize the generalization error.

Now there are two concepts often convoluted with training and generalization that is overfitting and underfitting. Overfitting means that our model was narrowly trained on the training data with too many parameters and thus cannot fit data outside the training set. This goes hand in hand with a high generalization error. Underfitting means that the model did not capture the details of the dataset. This commonly means that the training error is much higher.

B. Dropout

With dropout it is possible to prevent overfitting automatically. Dropout means that hidden units of a neural networks are dropped at random during training with a certain percentage. By default most neural networks are trained with a dropout parameter of 0.5, which amounts to 50% less units used during training. Mathematically dropout is the multiplication of each layer with a Bernoulli matrix

$$\begin{aligned} b^{(l)} &\sim Bernoulli(p) \\ \widetilde{h}^{(l)} &= b^{(l)} \cdot h^{(l)} \\ h^{(l+1)} &= f(W^{(l+1)} \cdot \widetilde{h}^{(l)} + b^{(l+1)}) \end{aligned} \tag{11}$$

[Srivastava et al., 2014] described dropout as producing a "thinned" network containing the hidden units that survived. The problem is that neural networks learn weights that are higher than the actual unobserved test data. The intuition is if a path through the network was established, it is much more likely to go through this particular path again. After a while the training data is completely fitted in certain paths through the network but not through others. If instead each iteration drops 50% of hidden units, they are forced to connect to other hidden units because the path with the highest action potential might not exist in this iteration.

IV. Recurrent Neural Networks

Recurrent neural networks and specifically LSTMs are sequential neural networks. The forward propagation and back propagation are now procedures of sequence, commonly referred to as back propagation over time. To achieve this we need to propagate the calculations from the forward pass to each successive input, before back propagation updates all the gradients at once. Figure 2 shows some kind of unrolled version of a RNN.

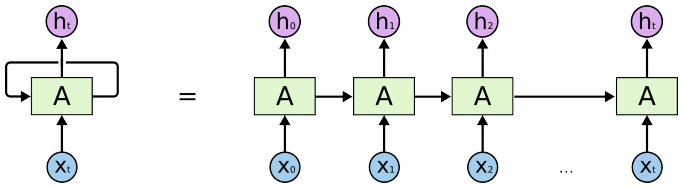


Fig. 2. Unrolled RNN [Olah, 2015]

Where $x_0 \ldots x_t$ are all sequential inputs at different steps t moving the calculation of each A to the next one. Mathematically speaking we can enhance the equations for feed forward networks by

$$\begin{aligned} h_{t-1} &= W^{(h_{t-2})} \cdot f(h_{t-2}) + W^{(h_{t-1})} \cdot x_{t-1} \\ \hat{y}_{t-1} &= W^{(S_{t-1})} \cdot f(h_{t-1}) \\ h_t &= W^{(h_{t-1})} \cdot f(h_{t-1}) + W^{(h)} \cdot x_t \\ \hat{y} &= W^{(S)} \cdot f(h_t) \end{aligned} \tag{12}$$

where $x_1, x_2, \ldots x_n$ is a sequence of word vectors embedded in a corpus. We can see how each hidden layer h passes on its context to the next forward pass of the next word. $\hat{y}_t$ is the loss of a word vector x_t at some point t given all words before. To understand this a bit better we need to look at language models designed for sequential word formations.

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, \ldots w_{i-1}) \tag{13}$$

Sequential language models take a word w_i and compute the probability with the joint probabilities of the prior words $w_1, \ldots w_{i-1}$. To compute such language models we normally take the softmax over word probabilities [Bengio et al., 2003]. With this we are able to tell what words are the most probable ones given the context window of all prior words.

A. Long short-term memory (LSTM)

While RNNs are useful tools for sequential computations, they suffer from a drawback identified by [Hochreiter et al.,

2001], namely the vanishing gradient problem. It occurs when a longer sequence cannot account for earlier timesteps as the gradients get inherently small moving to zero. That means words at the beginning of a long sequence are not accounted for. To tackle this problem we typically use Long short-term memory networks (LSTM) [Hochreiter and Schmidhuber, 1997]. The graphical view of an LSTM layer given by figure 3 is a bit more intuitive.

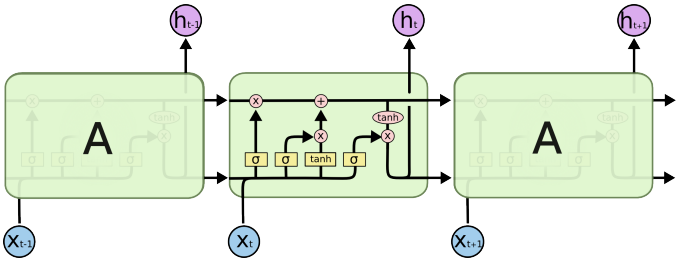


Fig. 3. LSTM [Olah, 2015]

A represents a hidden layer at timestep t. The hidden layers $h_{t-1} \dots h_t$ each consume a hidden layer h_t and memory c_t producing h_{t+1} and c_{t+1} for the next incoming word (hidden layer at step $t+1$). More formally a LSTM consists of a gating system that applies certain filters and rules to determine usefulness of the incoming memory and the current input word.

$$
\begin{aligned}
i^{(t)} &= \sigma(W^{(i)} \cdot x^{(t)} + U^{(i)} \cdot h^{(t-1)}) \\
f^{(t)} &= \sigma(W^{(f)} \cdot x^{(t)} + U^{(f)} \cdot h^{(t-1)}) \\
o^{(t)} &= \sigma(W^{(o)} \cdot x^{(t)} + U^{(o)} \cdot h^{(t-1)}) \\
\tilde{c}^{(t)} &= tanh(W^{(c)} \cdot x^{(t)} + U^{(c)} \cdot h^{(t-1)}) \\
c^{(t)} &= f^{(t)} \circ \tilde{c}^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} \\
h^{(t)} &= o^{(t)} \circ tanh(c^{(t)})
\end{aligned}
\tag{14}
$$

Intuitively one can easily connect certain ideas with each of the formula resulting in the final judgment $h^{(t)}$. The hidden layer of the step before $h^{(t-1)}$ in combination with the current word x_t generates a new memory $\tilde{c}^{(t)}$ which is the function of the input gate $i^{(t)}$. The forget gate $f^{(t)}$ is there to asses whether the memory is useful and what parts can be dropped from it. The next step calculates what can be forgotten and what parts of the current input is used for the new memory $c^{(t)}$. The final hidden gate is there to filter out the unnecessary parts of $c^{(t)}$ making a final judgment resulting in the output for the next hidden layer.

B. Neural machine translations

Google's neural machine translation system by [Wu et al., 2016] is a state of the art example for LSTMs at scale seen in figure 4. For this problem a deep LSTM neural network with an 8-layer encoder (source language) and a 8-layer decoder system (target language) was used. Each hidden layer got its own GPU and is connected in both directions from first to last and last to first word, improving results to some extend.

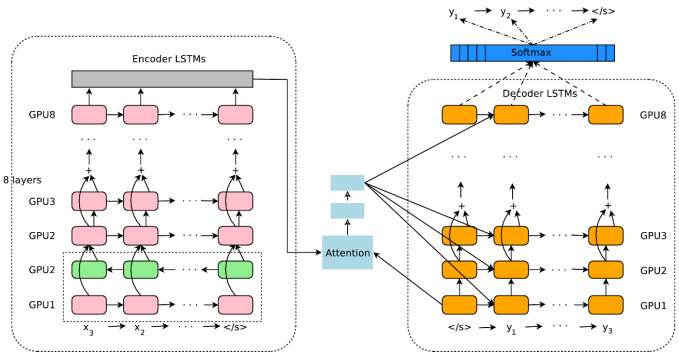


Fig. 4. Google Neural Machine Translations [Wu et al., 2016]

The language model for machine translations is

$$
\prod_{i=1}^{N} P(y_i \mid y_0, y_1, \dots, y_{i-1}; x_1, x_2, \dots, x_M) \tag{15}
$$

which reads as the probability P of a word y_i of the source language given its prior joint probability of the source sentence $y_0, y_1, \dots, y_{i-1}$ and the target sentence $x_1, x_2, \dots, x_M$. The probabilities, which source word matches the most likely target word can be decoded with a softmax. Additionally to make better use of the source language an attention mechanism is used to aid the decoder focusing on more relevant aspects of the given sentence. Secondly residual connections were used within the decoder and encoder to make sure that outputs of earlier hidden layers are still present in later ones. Each hidden layer was trained with its own GPU, making heavy use of parallelism to speed up training.

V. Convolutional Neural Networks

Convolutional neural network architectures are on the rise for natural language processing and deep learning applications. There are currently two approaches training a CNN with text. The first is based on character input. This allows to completely dismiss any kind of structural language models and just train on character statistics. These approaches resent best with the pixel based approach in vision, where it is possible to learn higher hierarchical features from local densities. Interested readers in character level CNN text classification might refer to [Zhang and LeCun, 2015]. The second approach is rather new and successfully connects the idea of sequential language modeling with CNNs, which was up to this point entirely done with RNNs.

A. Convolutional neural networks

CNN architectures typically start with a ***convolution layer*** that map low level features with a sliding window to a more abstract feature space. The sliding window is defined with a padding and stride sizes. ***Padding*** enables the convolution to ignore certain features and makes it possible to set barriers. The ***stride*** defines how the sliding window computes the features. After the convolutional layer typically follows a

non-linear activation function and ***pooling layer***. The pooling averages or maximizes the surrounding context given the prior action potential of the convolutions. This essentially drops further dimensions keeping higher level structures.
For character embedded neural networks the bag of words assumption holds. Assuming words to be independently and identically distributed is a drastic and wrong oversimplification. Therefore using RNNs would be the obvious choice. However while an oversimplification, it works surprisingly well on text. Additionally CNNs can be trained with fewer parameters and in parallel which makes them more tractable for real world application. [Zhang and LeCun, 2015]

B. Sequential convolutional neural networks

At first glance the idea of modeling sequences in a CNN is not intuitive. CNNs map features i.i.d. from its sample and project it via mechanisms called convolutions and pooling to extract hierarchically connected kernels. However as with vision taking average values around a given data point e.g. word or character yields location dependent characters. These windows have been applied to models such as word2vec before and work well in CNNs too. The key is that in lower input regions, words are closer to each other. Moving up the hierarchical CNN layers the feature space gets smaller and therefore a lot of variance from lower regions is narrowed to structural connections between words. This is necessary to work with the amount of information available.

Recent advances by [Gehring et al., 2017] have shown that CNNs can be used for sequential language models. The major point is that convolutions and gated recurrent units are mixed with an attention mechanism. The key however is to give the CNN a notion of position. The sentence embeddings are concatenated with a positional encoding that invalidates the i.i.d. assumption.

In figure 5 we see the ConvS2S model by [Gehring et al., 2017] that outperforms RNN based approaches on the machine translation task. The embedding input is a sentence of a fixed word size m and in a latent encoded d dimensional space of the documents that occur in the training set. The latent space is then encoded with its position in the whole corpus relative to the sentence. For machine translations we typically use an encoder for the source language and decoder for the target language. When both training samples in English and German are embedded they go through a couple of convolution and pooling layers. Each layer is interchangeably called a block l. Adding more blocks encodes more words. Each block has a fixed word window parameter k which adds the possibility to increase the sequence length by k times the number of convolutional blocks.

$$\begin{aligned} h_i^l &= f(W^l[h_{i-k/2}^{l-1}, \dots, h_{i+k/2}^{l-1}] + b_w^l) + h_i^{l-1} \\ p(y_{i+1} \mid y_1, \dots, y_i, x) &= softmax(W_o \cdot h_i^L + b_o) \end{aligned} \tag{16}$$

Afterwards gated recurrent units f (GRUs) decide on the relevant content. The dot product of the source and target language represents the hidden feature space for the targets

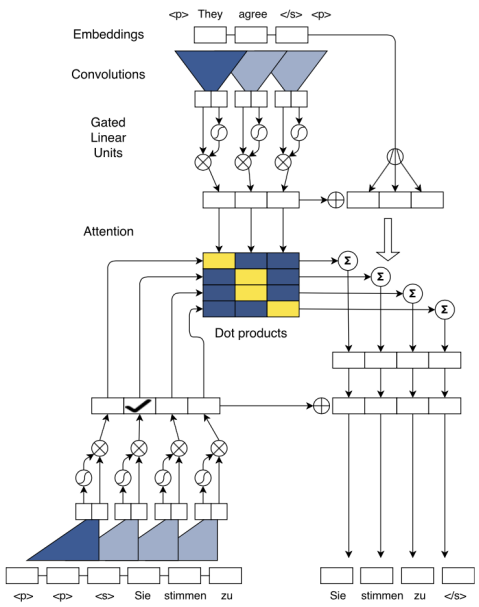


Fig. 5. CNN Sequence to Sequence Learning [Gehring et al., 2017]

language most probable word. This mechanism, called attention, maps the most likely words given a language model to a fix sized vector embedding. This is done with a softmax. Moreover multi-step attention is employed where each layer of the decoder stack is attended to separatly in combination with the output of the encoder networks. This is described as a key-value memory network depicted in [Miller et al., 2016]. Additionally, residual connections are employed due to the very deep architecture and loss of information. This helps to not forget earlier trained words of a sequence. The dimensions in the end match up due to the fact that outputs at each layer are padded by the input size of the encoder and decoder.

C. Results

The results can be drawn from table I. The ConvS2S yields state of the art 41.29 BLEU on the EN-FR translation task, yielding the highest thus far. It gets second in comparison to the Transformer model on EN-DE with 26.36 BLEU. At the time the model came out, the GNMT LSTM was the baseline model, which was surpassed by [Vaswani et al., 2017] with the Transformer model.

VI. Attentional Neural Network

Attention as described in [Bahdanau et al., 2014] is a mechanism of connecting hidden layers with a sense of location. Mathematically speaking we compute a dot product over all the words in a sequence applying a softmax with a resulting probability distribution telling us what parts of a sequence is useful. Thus, with attention it is possible to attend to the more relevant parts of longer sequences. This is a problem, due to the vanishing gradient of longer sequences, in particular for RNNs.

Attention can be understood looking at humans. If a human focuses on a sentence in a text, there will be highlighted words that stand out. These words play a role on how we perceive the content and other words not so much. Attention is the attempt to bring this kind of reasoning to neural networks by combining the hidden layers with a sense of location with respect to the current sequence.

It was still a big surprise when [Vaswani et al., 2017] announced their Transformer model heavily relying on a mechanism called multi-head attention. Shortly after, a generalized version towards deep learning and multi modal learning for different data sources and domains was described [Kaiser et al., 2017]. The architecture of the Transformer model can be seen in figure 6.

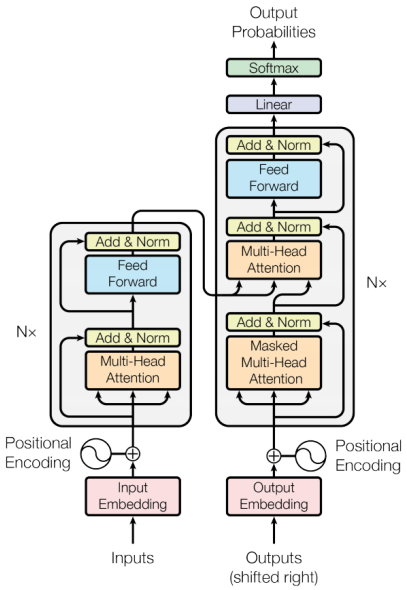


Fig. 6. Attention is all you need [Vaswani et al., 2017]

Again, the inputs are sentences mapped to a neural vector embedding. The position of each word in a sentence is currently not encoded and unlike RNNs with recurrence or CNNs with convolutions there is no information about it. The input embeddings are therefore combined with a positional embedding that can be produced in different ways and having the same dimensions as the input sequences.

The output layers of the encoder and decoder are fully-connected feed forward networks. The encoder maps the input $x_1, \ldots, x_n$ to an output $z_1, \ldots, z_n$ which is then consumed by a decoder multi-head attention mechanism producing the probabilities $y_1, \ldots y_n$ with a per word softmax probability distribution. Additionally residual connections are used within the self attention mechanism. Now two questions arise, first how does the multi-head attention mechanism work and how does the decoder combine the encoder output with the decoder input?

Attention can be seen as a function, mapping a query and a key-value pair to output vectors. By using the query and key of each value we focus the output on the part of the value that matters most.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{17}$$

The keys, queries and values are linear projections of the input embeddings wit size q+k+v. For each word in a sentence we then split a word into 3 components to produce $q, kandv$. Normally this is only done once. Multi-head attention goes one step further by doing several of these attentions in parallel concatenating the results. [Vaswani et al., 2017] have found that combining several attentions with a current head word, attends to different representations of the same query. Multi-head attention then becomes

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, \ldots, head_h)W^O \\ where\ head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{18}$$

In addition several regularization techniques have been employed, like residual dropout, attention dropout and label smoothing.

A. Results

The results of all the models can be seen in table I. The scoring is in BLEU, a measure for cooccuring bigrams between the predicted and correct position of the target sentence. The higher the BLEU score the better the outcome.

TABLE I. WMT MACHINE TRANSLATIONS

Models	**BLEU**	
	EN-DE	EN-FR
GNMT + RL	24.6	39.92
ConvS2S (BPE 40K)	25.16	40.46
Tranformer (base)	27.3	38.1
GNMT (Word pieces) + RL	26.3	41.16
ConvS2S (Ensemble)	26.36	**41.29**
Tranformer (big)	**28.4**	41.0

The Transformer beats the ConvS2S with a 28.4 BLEU score on the EN-DE translation task. The ConvS2s on the other hand beats the Transformer on the EN-FR translation task with 41.29 BLEU. The state of the art GNMT LSTM model still beats the Transformer on the EN-FR translation task with a 41.16 BLEU score. The ensemble methods are those with the best results over several different training runs. The Transformer base model was trained on fewer data than the big model validating the theory that more data yields better results.

VII. CONCLUSION

In this text we provide a look at the changing landscape of natural language processing in combination with deep learning. Almost all architectures use encoder-decoder networks connecting two networks with an attention

mechanism. Current methodology suggests that convolutional neural networks and a completely new class of attentional neural networks are on the rise. This is attributed to the training of fewer parameters and architectures that can be parallelized. Moreover, the accuracy of CNNs and attentional networks is higher than their RNN counterparts. However, RNNs/LSTMs are still considered baseline due to their efficiency of capturing sequential information about text. As new architectures and insights on the parameter space become available, there will be updates on sequential networks as well. Part of why CNNs and attentional networks work in the first place is their explicit positional encoding and the ability to memorize important aspects of inputs with attention mechanisms. LSTMs can be trained in a similar fashion but often at a much higher training cost using more complex architectures.

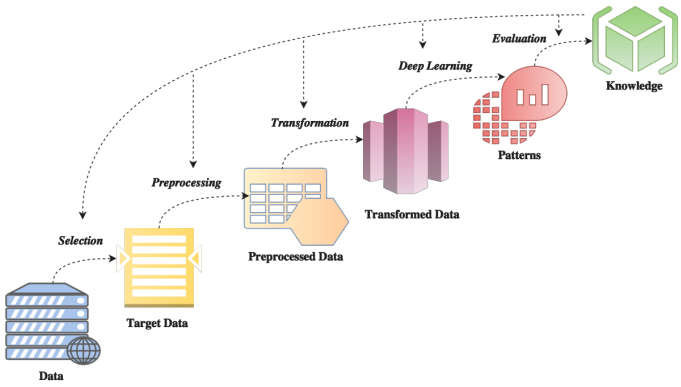
Apart from the standard neural network architectures for text we have also glanced over some aspects of statistical learning theory. Insights from this realm are not to be dismissed as they provide valuable input on evaluating and interpreting ANNs. A lot of papers solely focus on new architectures and improvement of accuracy. Why new architectures perform better is often disregarded. While it is true that higher accuracy is an objective that can be easily compared, there are few explanations on why they work. Deep learning still is a very immature research area with a lot of questions that remain to be answered. That is partially faulted to the fast development cycles and deep learning researchers that do not understand NLP well enough.

Lastly we have shown useful applications of deep learning in the real world. In particular machine translation systems have gotten a lot better by the standard BLEU scores. Leading are Google Brain attention models and Facebook FAIR sequential convolutional networks. Training these state of the art models is trivial with a laptop CPU when the dataset is relatively small. State of the art results on the other hand can only be achieved with larger data sets and more parameters. Thus it is inevitable to use GPU clusters, that outperform CPUs by a great margin. Software libraries such as Tensorflow, Theano of Caffe are designed to work with GPUs out of the box. This is not true for GPU clusters, as they need dedicated integration to work well.

VIII. Project Outlook

In this section I would like to explain my general project outlook for the upcoming semester. The process should be defined by experiments following the Knowledge Discovery in Databases (KDD).

An experiment defines a dataset and therefore a domain (e.g. text, vision), a hypothesis that we would like to explain and a model that, hopefully, explains the hypothesis. The model is evaluated by standardized scoring functions and compared to the baseline models in that domain. Additionally if possible the GPU clusters from the HAW Hamburg can be leveraged. Before conducting experiments on larger data sets



and more parameters the model should hint some indication that it works on smaller baseline data sets.

During the first project phase I would like to explore how to create different models solving problems in the domain of text understanding. This includes working on tasks like question answering, neural machine translations, abstractive summarization or neural programmers. Furthermore I believe that having a robust understanding of statistical learning theory to understand neural networks is inevitable. Neural networks yield great results on domains that are inherently fuzzy at the cost of a black box procedure. On the other hand classical NLP systems are well understood and algorithms work as expected. They do not yield more than is possible or proven. But what does a neural network learn? A hierarchical representation of a task?

Ideally, such questions and working through some problem sets result in a specialization that converges to my Master thesis.

REFERENCES

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Blei et al., 2003] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3(4-5):993–1022.

[Deerwester et al., 1990] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. In *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, volume 41, pages 391–407.

[Gehring et al., 2017] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Kaiser et al., 2017] Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., and Uszkoreit, J. (2017). One model to learn them all. *CoRR*, abs/1706.05137.

[Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.

[Miller et al., 2016] Miller, A. H., Fisch, A., Dodge, J., Karimi, A., Bordes, A., and Weston, J. (2016). Key-value memory networks for directly reading documents. *CoRR*, abs/1606.03126.

[Olah, 2015] Olah, C. (2015). Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015.

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

[Zhang and LeCun, 2015] Zhang, X. and LeCun, Y. (2015). Text understanding from scratch. cite arxiv:1502.01710Comment: This technical report is superseded by a paper entitled "Character-level Convolutional Networks for Text Classification", arXiv:1509.01626. It has considerably more experimental results and a rewritten introduction.