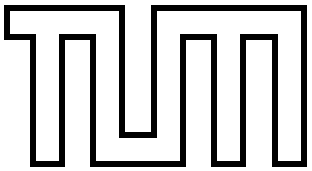


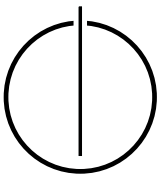
FAKULTÄT FÜR INFORMATIK

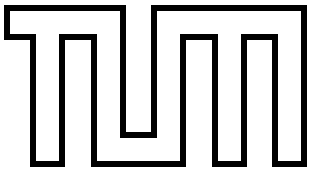
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extending a Newton-CG Second-order Optimizer to Natural Language Processing

Tao Xiang





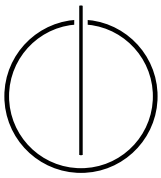
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extending a Newton-CG Second-order Optimizer to Natural Language Processing

Author:	Tao Xiang
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M. Sc. Severin Reiz, Dr. Felix Dietrich
Date:	15.08.2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.08.2021 Tao Xiang

Acknowledgements

I am grateful to my advisor Severin Reiz and Felix Dietrich for their valuable advice and constant support over the course of this thesis.

I would like to thank my family for raising me and providing great role models, and for their ongoing love and support.

Abstract

While Convolutional Neural Networks (CNNs) are a prominent class of machine learning models that are mainly applied to analyze visual imagery, Recurrent Neural Networks (RNNs) and the cutting-edge Attention Networks, Transformer Networks are another significant class of machine learning models that are mainly applied to deal with Natural Language Processing problems (NLP). Training these networks requires vast computing resources: due to a large amount of training data and due to the many training iterations. To speed up learning by reducing the necessary number of iterations to convergence, many specialized algorithms have been developed. First-order methods (using just the gradient) are the most popular, but second-order algorithms (using Hessian information) are gaining importance.

We have a second-order optimizer called Newton-CG that has already shown speed-up and accuracy benefits compared with first-order optimizers for image classification problems in Mihai Zorca's bachelor thesis [1]. In this thesis, we continue the comparison between Newton-CG and first-order optimizers, but we focus on NLP problems or Sentiment Analysis problems more specifically. We implemented two models: One is RNN based and the other is Self-Attention based. We trained these two models using Newton-CG optimizer and other first-order optimizers and recorded their loss and accuracy. We also tried to improve Newton-CG's performance by using Adam to pretrain.

In contrast, the performance of Newton-CG on sentiment analysis is not as good as on image classification. The performance of Newton-CG on the RNN model is very unstable, and the accuracy is only higher than that of SGD. On the Attention model, the performance of Newton-CG is more stable and the accuracy is higher.

Contents

1 Introduction

Neural Networks are highly flexible and powerful models. Given enough hidden units, they can approximate any continuous, real-valued function [2]. *Recurrent* Neural Networks (RNNs) and *Attention* Networks are specialized models, well suited for Natural Language Processing (NLP) tasks.

Training these models is a big challenge in deep learning. It requires a lot of computing resources and often we need to speed multiple days to train a single network [2]. Therefore, many specialized algorithms have been developed to solve this problem over the last decade. The most popular methods are first-order optimizers, such as SGD, AdaGrad, RMSprop, and Adam, which use only gradient information. Recently, second-order optimizers that also use Hessian information have been presented [3]. They have not replaced first-order optimizers and we think one important reason is their increased complexity of implementation.

A second-order optimizer is proposed by Julian Suk in his master thesis [4] and it has already shown speed-up and accuracy benefits compared with first-order optimizers for image classification problems in Mihai Zorca's bachelor thesis [1]. In this thesis, we call this optimizer Newton-CG and we compare it with other first-order optimizers on NLP problems.

In Chapter 2, we first give an overview of Neural Networks that are mainly used for NLP tasks, then we explain the Attention mechanism in detail. We also introduce cutting-edge Transformer Networks. Later we present the most commonly used first-order methods before focusing on second-order optimization and the sample Newton-CG method.

Chapter 3 presents the TensorFlow (TF) platform and its computational graph model. Then it presents the important high-level library Keras, before explaining how we integrated Newton-CG into TF. Afterward we introduce the machine learning problem we focus on in detail. We have also mentioned the limitations of our implementation and the computational setup.

In Chapter 4 we show the results of both models and do some analysis. Finally, we compare their performances and conclude the performance of Newton-CG on training Recurrent Neural Networks and Attention Networks.

2 Theoretical Background

This chapter gives an overview of neural networks and then we will focus on the *Attention Networks* and *Transformer Networks*, which are currently the most popular network architectures for natural language processing, then it presents optimization methods most commonly used to train the model, along with the new second-order optimization algorithm.

Note: This chapter touches on the theory of neural networks and their optimization algorithms. In the bachelor thesis "Training Deep Convolutional Neural Networks on the GPU Using a Second-Order Optimizer" by Mihai Zorca [1], handed in internally at TUM in 2020, we already included theory sections on many of these topics. The paper had sections on neural network training, its challenges and the commonly used algorithms SGD, AdaGrad, RMSprop and Adam. Therefore, Subsection 2.1.1, Subsection 2.4.1, Subsection 2.4.2, Subsection 2.4.3, Subsection 2.4.4, Subsection 2.4.5, Subsection 2.5.1, Subsection 2.5.2 and Subsection 2.5.3 will inevitably show some similarities in content and structure to their bachelor thesis counterparts.

2.1 An Overview of Neural Networks

Neural Networks (NNs) are the "quintessential deep learning model" [2]. In this section, we give an overview of the well-known neural networks, such as *Recurrent Neural Networks* and *Long Short Term Memory Networks*. We discuss the advantages and disadvantages of these neural networks in natural language processing problems, then we come to the cutting-edge network architectures in NLP: *Attention and Transformer Networks*.

2.1.1 Artificial Neural Networks

Artificial Neural Networks, or *Neural Networks* for short, are computing systems inspired from neuroscience [2]. They consist of connected layers of processing units, called *neurons* [5].Artificial Neural Networks map an input x to a category or label y. A feed-forward network defines the parameterized function $y = f(\boldsymbol{x}, \boldsymbol{W})$. Internally, each layer corresponds to a function f that processes the output of the previous layer [2]. Mathematically, if the neural network has n layers and the input is x, then the predicted category y can be expressed as follows:

$$y = f^{(n)}\left(\ldots\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)\right), \tag{2.1}$$

where $f^{(i)}$ corresponds to the function in the i_{th} layer. For a standard *fully connected layer*, each neuron in one layer connects to every neuron in the previous layer [2]. According to [6], we can write a layer $f^{(n)}$ as a vector of neurons it contains and the output of j_{th} neuron in n_{th} layer as [6]:

$$f^{(n)} = \left(z_1^{(n)}, z_2^{(n)}, \ldots, z_{M_{(n)}}^{(n)}\right)^\top \text{ and } z_j^{(n)} = \phi\left(\sum_{i=1}^{M_{(n-1)}} \left(w_{ji}^{(n)} f_i^{(n-1)}\right) + w_{j0}^{(n)}\right), \tag{2.2}$$

where the superscript n denotes the n_{th} layer, $z_i^{(n)}$ denotes the i_{th} neuron in the layer, $M_{(n)}$ is the number of neurons of n_{th} layer, the parameters $w_{ji}^{(}n)$ represents the *weights* and the constants $w_{j0}^{(n)}$ are called *biases*. The weights and the biases are both part of the parameter set W.

The function ϕ represents the neuron's *activation function*. theoretically, any differentiable function can be used as the activation function [5]. Common choices are sigmoidal functions like $\sigma(x) = \frac{1}{1+\exp(-x)}$ or $\tanh^{-1}(x)$ and rectified linear unit (ReLU) $y = \max\{0, x\}$. The former have very small gradients across most of their domain thus sometimes making the training very slow, whereas the latter is easier to optimize since it preserves linear properties.

2.1.2 Word Embedding

Before we dive into the common used neural networks in NLP, let's first have a look at one of the most important techniques in NLP: Word Embedding.

Word embedding is a type of word representation and words with similar meanings have similar representations. It is a general term for the methods that map words to real number vectors. Word embedding is one of the important breakthroughs in NLP. In the following will focus on what is word embedding and introduce some word embedding algorithms.

Word embedding is actually a kind of technology. A single word is represented as a real number vector in a predefined vector space, and each word is mapped to a vector. For example, assume we have a sentence that contains words such as "cat", "dog", "love", and we map these words into vectors: "cat" to (0.2 0.2 0.4), "dog" to 0.2 0.2 0.4, and love to (-0.4 -0.5 -0.2) (the data is only for illustration). The process of mapping words to vectors like this is called *word embedding*.

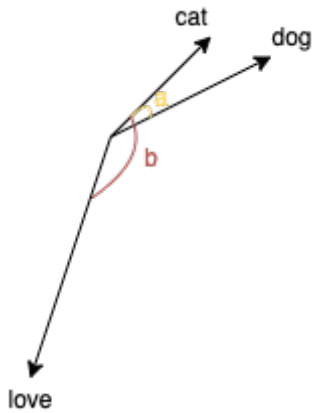


Figure 2.1: $\cos a < \cos b$, then "cat" and "dog" are more similar whereas "cat" and "love" are more different.

One reason why we hope to turn each word into a vector is for more convenience of calculation. Let's look at the previous example: Humans know that "cat" and "dog" are both animals, and "love" is a kind of emotion. But for machines these three words are all represented by 0,1 It's just a binary string and cannot be calculated. By converting words into word vectors through word embedding, the machine can calculate the words, and obtain the similarity between words by calculating the cosine of the angle between different word vectors.

In addition, word embedding can also be used as an analogy, such as: v ("king")-v ("man") + v ("woman") ≈ v ("Queen"), v ("China") + v ("capital") ≈ v("Beijing"). With these operations, machines can "understand" the meaning of words like humans.

Several popular word embedding algorithms are like embedding layer, Word2vec [7].

2.1.3 Recurrent Neural Networks

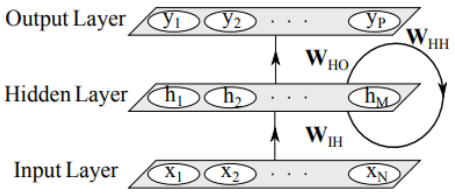


Figure 2.2: A folded simple RNN structure. Source: [8]

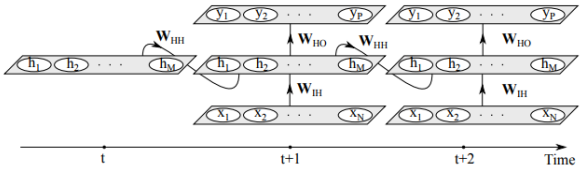


Figure 2.3: Unfolded structure of a simple RNN. Each arrow shows a full connection of units between the layers. Source: [8]

Recurrent Neural Networks (RNN), are ANNs with recurrent connections that are able to model sequential data for sequence prediction and recognition [9]. RNNs consist of high-dimensional hidden states with non-linear dynamics [10]. The structure of hidden states functions as the memory of the network and the state of the hidden layer at a certain time depends on its previous state [11]. These hidden states allow the RNNs to remember and store the past complex information for long period [8].

In the following we adopt the notations of [8] to explain the structure of a simple RNN. A simple RNN has three layers: the input layer, the recurrent hidden layer and the output layer, as shown in Figure 2.2. In the input layer we have the input, which can be written as a sequence of vectors through time t such as $\{\ldots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \ldots\}$, where $\mathbf{x}_t = (x_1, x_2, \ldots, x_N)$.

Assuming the hider layer has totally M hidden units $\mathbf{h}_t = (h_1, h_2, \ldots, h_M)$, in a fully-connected RNN the input units are connected to the hidden units in the hidden layer, where the connection is defined with a weight matrix $\mathbf{W}_{IH}$. The hidden units are also connected to each other through time with recurrent connections [8].

The hidden layer defines the state or "memory" of the system as

$$\mathbf{h}_t = f_H(\mathbf{o}_t), \tag{2.3}$$

where

$$\mathbf{o}_t = \mathbf{W}_{IH}\mathbf{x}_t + \mathbf{W}_{HH}\mathbf{h}_{t-1} + \mathbf{b}_h. \tag{2.4}$$

$f_H(\cdot)$ is the activation function in hidden layer, and $\mathbf{b}_h$ is the bias vector of the hidden units. The hidden units are connected to the output layer with weighted connections $\mathbf{W}_{HO}$. The output layer has P units $\mathbf{y}_t = (y_1, y_2, \ldots, y_P)$ that are computed as

$$\mathbf{y}_t = f_O(\mathbf{W}_{HO}\mathbf{h}_t + \mathbf{b}_o)$$

where $f_O(\cdot)$ is the activation function in output layer and $\mathbf{b}_o$ is the bias vector. Because the input-target pairs are sequential through time, the above steps are repeated over time $t = (1, \ldots, T)$.

The hidden states are the most significant part of RNN: they summarize and store the important information about the previous states over many timesteps. This integrated information can affect the behavior of RNN in the future and make precise predictions [10].

2.1.4 Long Short Term Memory Networks

Long Short Term Memory Networks (LSTM), is a special kind of recurrent neural networks that is capable of learning long-term dependencies [12]. The standard RNN can learn to use the past information when the gap between the relevant information is relatively small. However, if the gap is relatively large, then RNNs become unable to connect the information [13, 14]. In contrast, LSTM can handle this problem quite well [15, 16].

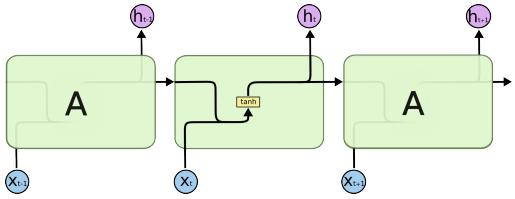


Figure 2.4: The repeating module in a standard RNN contains a single layer. Source: Christopher Olah's blog

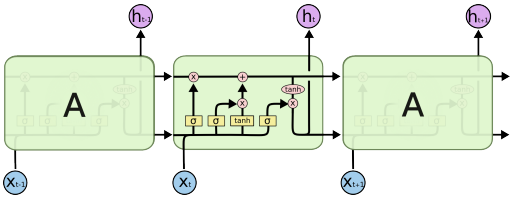


Figure 2.5: The repeating module in an LSTM contains four interacting layers. Source: Christopher Olah's blog

Similar to RNN, LSTM also has hidden states for storing information about previous sequence data. The difference is that the repeating module in an LSTM contains four interacting layers whereas in RNN there is only one, see Figure 2.4 and Figure 2.5. We can imagine this repeating module as one conveyor belt, on which there are cell states and a series of transformations that are used to process the current hidden state and cell state and output the next hidden state and cell state. These transformations are usually called *Gates*.

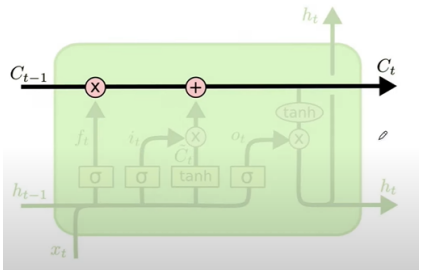


Figure 2.6: The repeating module in LSTM. Source: Christopher Olah's blog

There are totally three gates in one repeating module: 1) forget gate 2) input gate & new values 3) output gate. In the forget gate layer the not very important information is thrown away from the cell state C_{t-1}. More specifically, it takes h_{t-1} and x_t as input and outputs a vector f_t of same length as C_{t-1} with all entries between 0 and 1, where a 1 represents "completely keep this" while a 0 represents "completely get rid of this." Later f_t and C_{t-1} will be elementwise multiplicated. Mathematically, this can be written as follows:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right), \tag{2.5}$$

where h_{t-1} is the hidden state at time $t-1$, x_t is the input at time t, W_f is the weight matrix, b_f is the bias vector and σ is the sigmoid function.

In the next step we're going to add new information to the cell state, which can be done in two parts: First, a sigmoid layer called "input gate layer" outputs a vector that controls which values are retained in the new values vector. Then, a tanh layer creates a vector of new values, $\tilde{C}_t$, which will be added to the state. Finally we combine the forget gate and input gate and update the cell state. Mathematically, this process can be written as follows:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{2.6}$$
$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{2.7}$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t, \tag{2.8}$$

where W_i and W_C are the weight matrices, b_i and b_C are the bias vectors.

Finally, the new hidden state h_t is outputted based on the cell state C_t we got before. This step is done in the output gate layer. More specifically, we compute a vector o_t which is used to determine what parts of the cell state are outputted. Mathematically, it can be written as follows:

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{2.9}$$
$$h_t = o_t * \tanh\left(C_t\right), \tag{2.10}$$

where W_0 is the weight matrix and b_o is the bias vector.

2.2 Attention Mechanism

2.2.1 Seq2Seq Model

Seq2seq model was originally proposed by Google in 2016 for use in machine translation [17]. It is the basis of many advanced sequence-to-sequence models like Attention models, GTP models, Transformers and Bert. The problem of machine translation is to translate the source language into the target language, thus it's a many-to-many problem: Both the input and the output is a sequence, and the length of the input sentence and output sentence is not fixed and not necessarily equal. In the following we adopt the notation of [18] to explain Seq2Seq model.

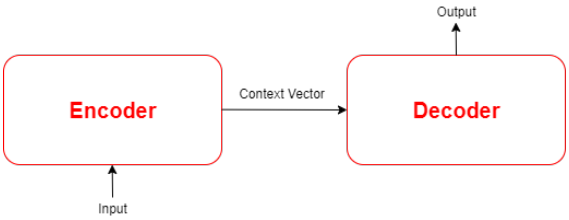


Figure 2.7: Seq2Seq model in overview

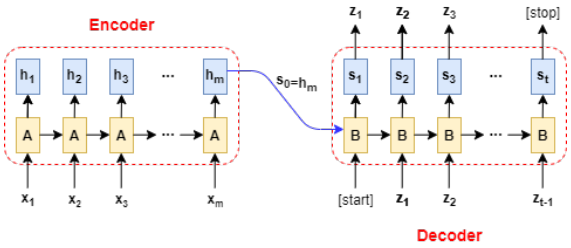


Figure 2.8: Seq2Seq model in detail

In Figure 2.7 we can see the structure of Seq2Seq model in a high-level overview: It consists of an encoder block and a decoder block, and these two blocks are connected by a vector we call "context vector" here. The encoder and decoder are usually RNN based networks such as RNN, LSTM or GRU. The encoder processes the input sequence and store its information in a vector, i.e. the context vector. Then it passes the context vector to the decoder, which then parses the context vector and predicts the output sequence.

For the sake of simplicity, in our example we use RNN as the encoder and decoder networks. As you can see in Figure 2.8, A and h_i are the weight matrix and hidden states of encoder respectively, whereas B and s_i are the weight matrix and hidden states of decoder respectively. The sequence $x_1, x_2, \cdots, x_m$ is the input words sequence. The encoder processes each word and outputs a hidden state that contains the information about previous processed words. This step repeats until the last word of the input sequence, where the last hidden state h_m is outputted, which contains the information about the whole input sentence. This hidden state is seen as the *context vector* and is set to the initial hidden state of the decoder network. In other words, the decoder receives information about the whole input sentence and then starts to generate the output sentence with the same meaning but in the target language. The decoder works like a text generator: The predicted word z_i at time t_i is the input word at time t_{i+1}.

2.2.2 Attention

In the standard Seq2Seq model all hidden states except the last one in the encoder are thrown away and only the last one (the context vector) is passed to the decoder. And every time the decoder generates the next predicted word, it looks at only its current state (without checking the hidden states in the encoder). Due to this mechanism, the standard Seq2Seq model is incapable of remembering longer sequences [19]. In other words, if the input sequence is relatively longer, the context vector could lose information of the early part of the input sequence, which leads to inaccurate translation.

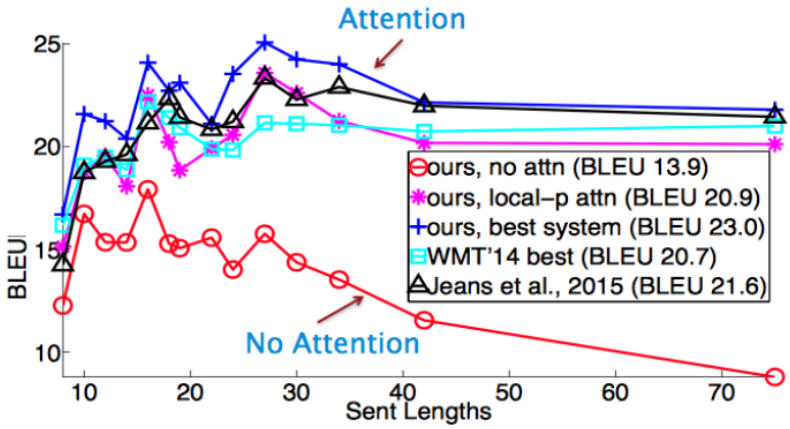


Figure 2.9: BLEU score of models with and without Attention.
Source: [19]

To solve this problem, a technique called *Attention* was raised by Bahdanau et al. in 2015 [20]. It's a technique that mimics cognitive attention and can significantly improve the ability to remember long sentences [19]. As you can see in the Figure 2.9, the BLEU score of models without attention mechanism decreases as the length of sentence increases whereas the BLEU score of models with attention mechanism remains at a relatively high level and relatively stable as the length of the sentence increases.

The central idea behind Attention is that in each iteration of the decoder, it looks at all the intermediate hidden states in encoder (rather than only the last one) to computes a new context vector, and then uses this context vector to generate the next output word. In other words, every time the decoder predicts the next word, it will re-look at the entire input sequence and consider which part of the input sequence is more important for predicting this word, then it pays more "attention" to that part.

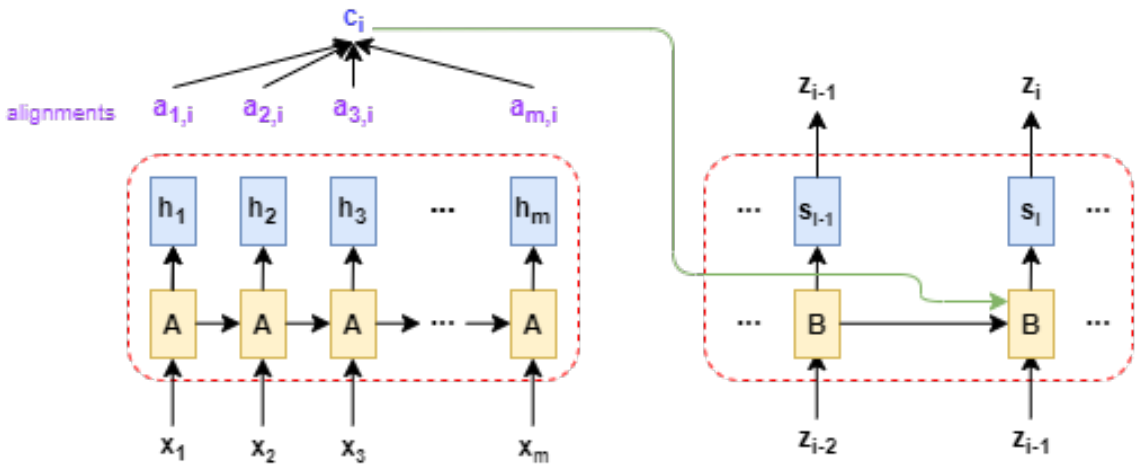


Figure 2.10: Simple-RNN + Attention

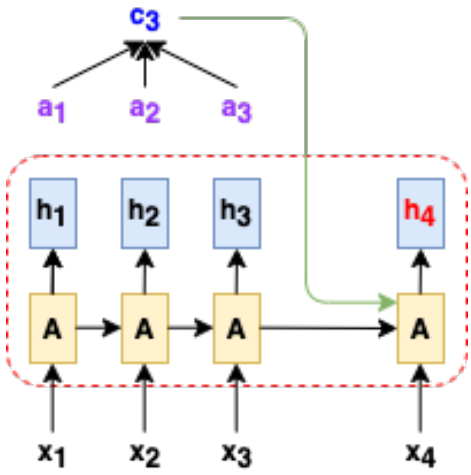


Figure 2.11: Simple-RNN + Self-Attention

In order to express the importance of the hidden states in the encoder for predicting the next word, Bahdanau proposed a new function that computes the weight of each hidden state in the encoder. The larger the weight, the more important the corresponding hidden state. Mathematically, the weight is written as follows [20]:

$$\alpha_{j,i} = \frac{\exp\left(\text{score}\left(s_{i-1}, \boldsymbol{h}_j\right)\right)}{\sum_{j'=1}^{m} \exp\left(\text{score}\left(s_{i-1}, \boldsymbol{h}_{j'}\right)\right)}, \tag{2.11}$$

where $\alpha_{j,i}$ is the alignment score (the weight) of the j_{th} hidden state h_j in encoder when calculating the i_{th} hidden state s_i in decoder, s_{i-1} is $(i-1)_{\text{th}}$ hidden state in decoder, and score is the function that calculate the correlation between the hidden states in decoder and encoder. As you can see, $a_{j,i}$ is a number between 0 and 1 and all the weights sum up to 1, and the higher the score, the greater the weight.

There are many variants of the score function. In the original paper, a feed-forward network with a single hidden layer is used to parameterize the alignment score α. The score function is therefore in the following form:

$$\text{score}\left(\boldsymbol{s}_i, \boldsymbol{h}_j\right) = \mathbf{v}_a^\top \tanh\left(\mathbf{W}_a\left[\boldsymbol{s}_i; \boldsymbol{h}_j\right]\right), \tag{2.12}$$

where tanh is the activation function and both $\mathbf{v}_a$ and $\mathbf{W}_a$ are weight matrices to be learned.

There is another popular way to compute the score function, which has been used in *Transformer* [21]. Mathematically, the function can be written as follows:

$$\boldsymbol{k}_j = \mathbf{W}_K \cdot \boldsymbol{h}_j \tag{2.13}$$

$$\boldsymbol{q}_i = \mathbf{W}_Q \cdot \boldsymbol{s}_i \tag{2.14}$$

$$\text{score}\left(\boldsymbol{s}_i, \boldsymbol{h}_j\right) = \boldsymbol{k}_j^T \boldsymbol{q}_i, \tag{2.15}$$

where $\mathbf{W}_K$ and $\mathbf{W}_Q$ are parameter matrices. $\boldsymbol{k}$ stands for "key" and $\boldsymbol{q}$ stands for "query". The scalar product of the query vector and the key vector can reflect the similarity of these two vectors, and this similarity is our score.

After calculating all the weights of hidden states in encoder, we can then calculate the context vector as follows [20]:

$$\mathbf{c}_i = \sum_{j=1}^{m} \alpha_{j,i} \boldsymbol{h}_j, \tag{2.16}$$

then the next hidden state s_i in the decoder is computed by [20]

$$\boldsymbol{s}_i = f\left(\boldsymbol{s}_{i-1}, z_{i-1}, \mathbf{c}_i\right), \tag{2.17}$$

where z_{i-1} is the predicted word from last hidden state s_{i-1} in decoder, f is the main function of the decoder. In Figure 2.10 you can find an example of Simple-RNN plus attention. Basically the Equation 2.17 is where Attention comes in, since in the normal RNN based models this step is $\boldsymbol{s}_i = f\left(\boldsymbol{s}_{i-1}, z_{i-1}\right)$

2.2.3 Self-Attention

Self-Attention is a mechanism that uses Attention in only one RNN based network rather than the encoder-decoder structure. It's originally raised by Cheng et al. in 2016 [22]. It's very similar to the standard Attention mechanism but with little difference when computing the alignment score α: It computes the correlation between the current hidden state and all the previous hidden states. This is because we now only have one network instead of two. In Figure 2.11 there is an example of Simple-RNN plus Self-Attention mechanism.

Self Attention Layer is a network layer structure that completely relies on self-attention mechanism to calculate representations of its input and output, and does not rely on the RNN-based network structure.

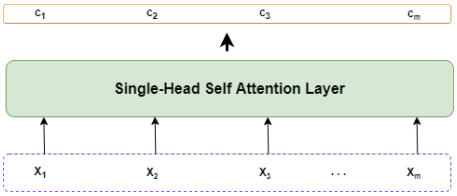


Figure 2.12: Overview of single-head self attention layer.

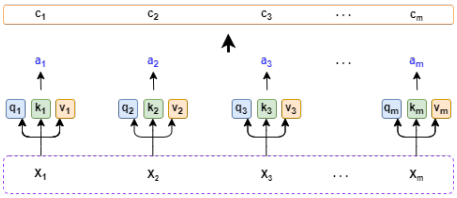


Figure 2.13: The detailed structure of single-head self attention layer.

The input vectors $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m$ are the sum of embedding vectors and position encoding vectors respectively. For each input token $\mathbf{x}_i$, we first compute three vectors: Query vector $\mathbf{q}_i$, Key vector $\mathbf{k}_i$ and Value vector $\mathbf{v}_i$ as follows:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i \tag{2.18}$$
$$\mathbf{k}_i = \mathbf{W}_K \mathbf{x}_i \tag{2.19}$$
$$\mathbf{v}_i = \mathbf{W}_V \mathbf{x}_i, \tag{2.20}$$

where $\mathbf{W}_Q$, $\mathbf{W}_K$ and $\mathbf{W}_V$ are the parameter matrices. Then we can compute the the alignment score α_i as follows:

$$\alpha_i = \text{softmax}\left(\frac{\mathbf{K}^T \mathbf{q}_i}{\sqrt{d_k}}\right), \tag{2.21}$$

where $\mathbf{K}$ is a matrix composed of vectors $\mathbf{k}_1, \mathbf{k}_2, \cdots, \mathbf{k}_m$, i.e. $\begin{bmatrix} \mathbf{k}_1 & \mathbf{k}_2 & \cdots & \mathbf{k}_m \end{bmatrix}$. d_k is the dimension of key vectors. After that we can compute the context vector $\mathbf{c}_i$ as follows:

$$\mathbf{c}_i = \mathbf{V}\alpha_i, \tag{2.22}$$

where $\mathbf{V}$ is a matrix composed of vectors $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$, i.e. $\begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_m \end{bmatrix}$. The output of a Self-Attention layer is $\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_m$, we can write them as a matrix $\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_m \end{bmatrix}$.

2.3 Transformer

Transformer is currently one of the most popular models in NLP. It was originally proposed in the paper "Attention is All You Need" [21]. It also belongs to Seq2Seq models and can solve tasks such as translation and text summarization. It relies entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution [21].

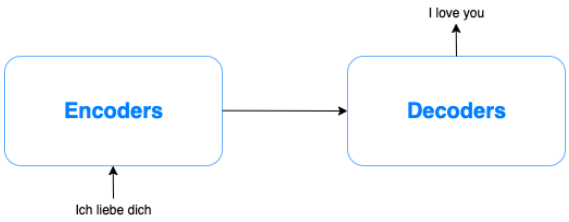


Figure 2.14: Structure Overview of Transformer Model (1)

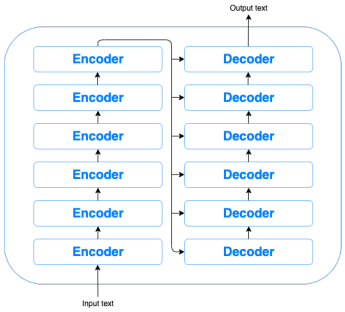


Figure 2.15: Structure Overview of Transformer Model (2)

As transformer is also a Seq2Seq model, it has an encoder block and a decoder block (Figure 2.14). What different is, in the encoder/decoder block there is not only one encoder/decoder, but several identical encoders/decoders stacked on top of each other (Figure 2.15).

In order to distinguish it from the previous RNN-based model, I will write "encoder/decoder stack" instead of "encoder/decoder block". The number of encoder/decoder units in an encoder/decoder stack is a hyperparameter. In the original paper, six encoders/decoders have been used. Notice that these encoders/decoders are only identical in structure, but they do not share parameters. In other words, each encoder/decoder needs to be trained separately.

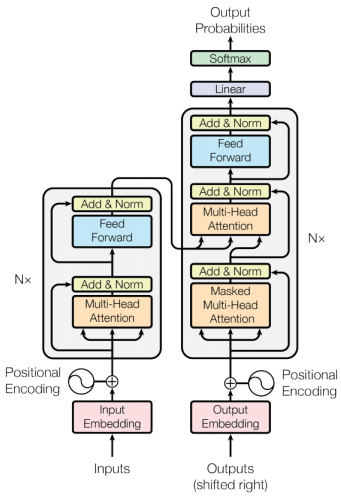


Figure 2.16: The Transformer model architecture.
Source: [21]

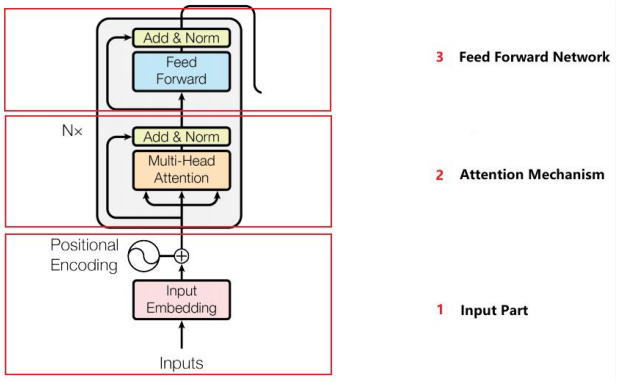


Figure 2.17: Encoder stack of Transformer

In Figure 2.15 you can find the detailed structure of transformer, which we will dive into from now on. As we stated before, transformer consists of an encoder stack and a decoder stack. And a encoder/decoder stack has six encoder/decoder units. Let's focus on one encoder unit first.

2.3.1 Encoder Units

One encoder unit can be divided into three parts: 1) Input part 2) Attention part 3) Feed-forward network part, as you can see in the Figure 2.17.

Input Part

The input part mainly preprocesses the input. The raw input is sentences with different lengths. The embedding layer maps the tokens/words in the sentences to vectors of dimension d_{model}.

Since transformer is not recurrent and convolutional, in order for the model to make full use of the order and relative position of words in the input sequence, *Positional Encoding* is added to the model.

A token/word corresponds to an embedding vector and a position encoding vector. These two vectors are of the same dimension so that they can be summed up. The position encoding vector gives information about the position of the token/word in the sentence. More specifically, the position encoding vector for a token/word $\mathbf{w}$ is computed as follows [21]:

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \tag{2.23}$$

$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d_{\text{model}}}\right), \tag{2.24}$$

where pos stands for the position of token/word $\mathbf{w}$ in the sentence. Even entries are computed according to Equation 2.23 whereas odd entries are computed according to Equation 2.24.

By using the following properties of trigonometric functions:

$$\begin{cases} \sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta \\ \cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta \end{cases} \tag{2.25}$$

we can get:

$$\begin{cases} PE(pos+k,2i) = PE(pos,2i)\times PE(k,2i+1) + PE(pos,2i+1)\times PE(k,2i) \\ PE(pos+k,2i+1) = PE(pos,2i+1)\times PE(k,2i+1) - PE(pos,2i)\times PE(k,2i) \end{cases} \tag{2.26}$$

It can be seen that the entries of the position encoding vector at position $pos+k$ can be expressed as a linear combination of the entries of position encoding vectors at position pos and k. Such a linear combination means that the position encoding vector contains relative position information.

As stated in Subsection 2.1.2, embedding makes words with similar meanings closer to each other. But embedding doesn't encode the relative position of words in a sentence. So with positional encoding, words with similar meanings and shorter distance in the sentence are closer in the d_{model}-dimensional space.

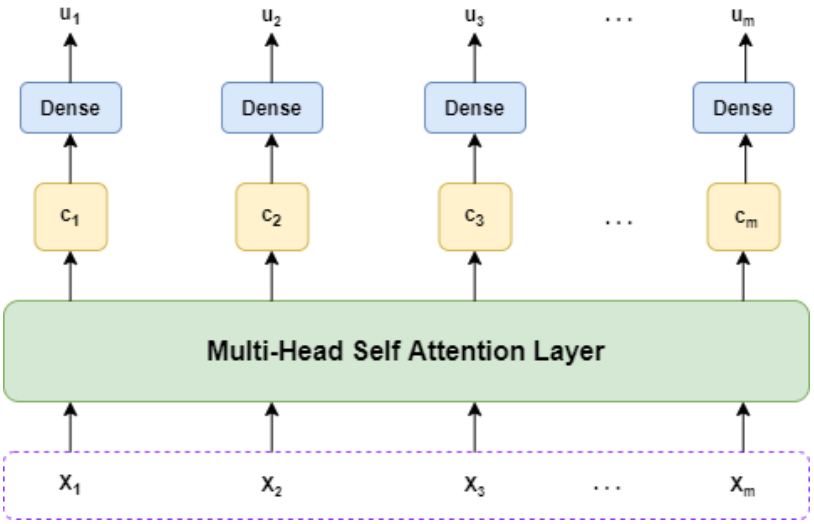


Figure 2.18: The structure of one encoder unit.

Attention Part

As we said before, the transformer completely relies on self-attention to calculate representations of its input and output and does not rely on the RNN-based network structure. Therefore, we can regard the self-attention as a single layer, as follows We will call it the *Self-Attention Layer* (Subsection 2.2.3).

In Transformer the self-attention is computed not only once but multiple times in parallel and independently, which is called *Multi-Head Self Attention*. The previous Self Attention layer is actually a Single-Head Self Attention layer and a Multi-Head Self Attention layer is composed of several Single-Head Self Attention units. The output of Multi-Head Self Attention layer is the concatenation result of the Single-Head Self Attention units' outputs.

Assuming there are l Single-Head Self Attention units in one Multi-Head Self Attention layer, then there are totally $3l$ parameter matrices since one Single-Head Self Attention unit has 3 parameter matrices. And Suppose Single-Head Self-Attentions' outputs are $d \times m$ matrices then the Multi-Head's output shape is $(ld) \times m$.

Besides attention mechanism the author also has applied *Residual Connection* from the input for Attention layer to output of Attention layer [21]. This structure can help to deal with gradient vanishing problem [23]. After that the author has applied layer normalization.

Feed-Forward Network Part

The Attention layer is followed by Feed-Forward Network layer, which is used to process the output from one attention layer in a way to better fit the input for the next attention layer in the next encoder unit or decoder unit. Notice that the exact same parameter matrix is

applied to each respective token position. Mathematically, this process can be written as follows:

$$\mathbf{u}_i = \text{ReLU}\left(\mathbf{W}_U \mathbf{c}_i\right) \qquad \text{for } i = 1 \text{ to } m. \tag{2.27}$$

Since it is applied without any communication or inference by other token positions it is a highly parallelizable part of the model.

Same as attention part, there is also a residual connection (followed by layer normalization) from dense network's input to its output.

2.3.2 Decoder Units

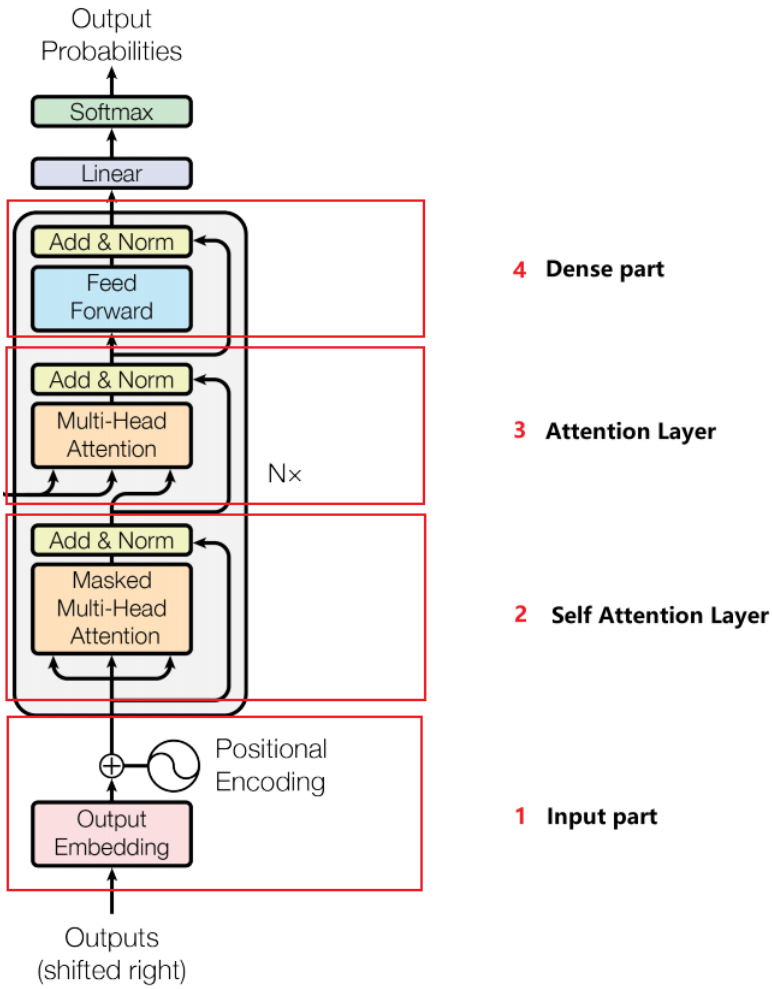


Figure 2.19: Decoder stack of Transformer.

One decoder unit can be divided into four parts: 1) Input part 2) Self Attention layer 3) Attention layer 4) Feed-forward network part, as you can see in the Figure 2.19.

Input Part

This input part is very similar to the one in encoder: The input sequence is processed by word embedding and positional encoding, then the result as the formal input flows to next layer. Notice that the input sequences are sentences in target language.

Self Attention Layer

In this layer there is a Multi-Head Self Attention which is almost the same as in encoder. The only difference is that we need to mask the input. Because when we predict the i_{th} word, we only know the information about the 1_{st} word until the $(i-1)_{\text{th}}$ word, and we don't know the following words yet, so we need to mask out the i_{th} word until the last word. In the original paper this is implemented by setting all these illegal entries to $-\infty$ [21]. Thus this layer is also called *Masked Multi-Head Attention Layer*. Same as in encoder , there is residual connection around this layer followed by layer normalization.

Attention Layer

In this layer there is a Multi-Head Attention (not Self Attention), which is like a Seq2Seq model that has an encoder and a decoder, as shown in the Figure 2.20. In order to distinguish the encoder/decoder in this layer and encoder/decoder in Transformer model, I will write "inner encoder" and "inner decoder" to indicate the encoder/decoder in this layer.

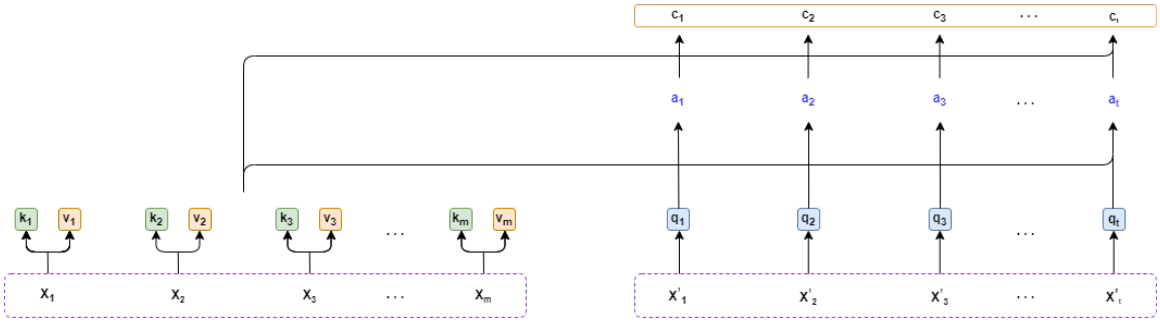


Figure 2.20: The structure of the Multi-Head Attention Layer.

The input $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m$ for the inner encoder is the output from the last encoder unit in encoder stack, and thus this output is sent to every decoder unit in decoder stack, as you can see in the Figure 2.15. The input $\mathbf{x}'_1, \mathbf{x}'_2, \cdots, \mathbf{x}'_t$ for the inner decoder is the output from the last layer, i.e. the Self Attention Layer.

The calculation process of the Attention Layer is similar to the Self-Attention Layer, except that Key vectors and Value vectors are based on inner encoder's inputs whereas Query vectors are based on inner decoder's inputs. Mathematically, this step can be written as follows:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i \tag{2.28}$$
$$\mathbf{v}_i = \mathbf{W}_V \mathbf{x}_i \tag{2.29}$$
$$\mathbf{k}_j = \mathbf{W}_K \mathbf{x}'_j, \tag{2.30}$$

where $\mathbf{W}_Q, \mathbf{W}_K$ and $\mathbf{W}_V$ are the parameter matrices. The following procedure of computing the alignment score α_i and the context vector $\mathbf{c}_i$ is completely the same as in Self-Attention Layer.

Also the same as in Self-Attention Layer, the Attention is computed multiple times in parallel and independently, thus it's called *Multi-Head Attention*. There is also a residual connection followed by layer normalization around this layer.

Feed-Forward Network Part

This layer works the same as in the encoder stack, and there is also residual connection followed by layer normalization around this layer. Once we get the output, we do softmax again to select the final probabilities of words.

2.4 First-Order Optimization Algorithms

Before we dive into the concrete optimization algorithms, let's first have a look at the general gradient-based optimization, since the most optimization algorithms use the gradient information.

2.4.1 Gradient-Based Optimization

In supervised learning, the focus of this work (machine translation), we are given N inputs $\boldsymbol{x}_i$ and N corresponding target outputs y_i, the model function $f(\boldsymbol{x}, \boldsymbol{W})$ and its parameter set $\boldsymbol{W}$.

The main goal of (traditional) optimization is to minimize functions. However, optimization algorithms for training Neural Networks are different from the traditional optimization algorithms: In machine learning, people care about some performance measurement P, which is usually intractable. In order to improve P, we define a different *cost* or *loss* function $L(\boldsymbol{W})$ to be minimized instead [2]. Common examples for L include the *sum-of-squares* function [6]:

$$L(\boldsymbol{W}) = \frac{1}{2} \sum_{i=1}^{N} \left\| f\left(\boldsymbol{x_i}, \boldsymbol{W}\right) - y_i \right\|^2, \tag{2.31}$$

and the *cross-entropy* (for K classes) between training data and model distribution [6]:

$$L(\boldsymbol{W}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ki} \ln f_k\left(\boldsymbol{x_i}, \boldsymbol{W}\right), \tag{2.32}$$

where, for the latter loss, the variables y_i have a 1-of-K encoding indicating the class. With sigmoid and softmax outputs that suffered from slow learning with the squared error loss [2], the cross-entropy loss led to greater performance in models.

The major task of optimization in machine learning is to minimize the chosen $L(\boldsymbol{W})$ by finding the suitable parameters W. Optimization algorithms for minimizing the loss function are usually iterative: Starting at a point $\boldsymbol{W}_0$, they generate a sequence of sets $\{\boldsymbol{W}_k\}_{k\in\mathbb{N}}$ until a stopping criterion has been satisfied [24]. Because we are interested in minimizing L, we stop when approximating a local minimum, where the gradient is (close to) zero. All commonly used algorithms are *line search* [24] methods. For each step k they first compute a search direction p_k and then decide how far to move along p_k. We get $\boldsymbol{W}_{k+1} = \boldsymbol{W}_k + \alpha_k p_k$, where α is called the *step length* [24] or *learning rate* (lr) [6]. Algorithm 1 shows the iterative structure that the main training algorithms follow.

Algorithm 1: Basic line search procedure for network training

Require: $L(\boldsymbol{W})$: The chosen loss function with parameters $\boldsymbol{W}$
Require: $\boldsymbol{W}_0$: Starting point

$k \leftarrow 0$
while $\boldsymbol{W}_k$ *not converged* **do**
  $k \leftarrow k + 1$
  $p_k \leftarrow p$ `// Compute current step direction.`
  $\alpha_k \leftarrow \alpha$ `// Compute or use a given step size.`
  $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} + \alpha_k p_k$

For almost all algorithms, p_k should be a descent direction [24], i.e. $p_k^\top \nabla L(\boldsymbol{W}_k) < 0$. For gradient descent algorithm the steepest descent direction has been chosen: $p_k = -\nabla L(\boldsymbol{W}_k)$. Although the modern algorithms have added many further enhancements, the gradient remains the main information they rely upon.

Consistent with Literature, we call algorithms that only use the gradient as *first-order* optimization algorithms. Algorithms that also use the Hessian matrix are called *second-order* optimization algorithms [2, 24].

2.4.2 Stochastic Gradient Descent

Gradient descent is a method that follows the gradient over an entire training set. This is referred to as *batch learning* [6]. One downside of batch learning is that if the training set is relatively large, then the training will be increasingly expensive. This is where *mini-batch* comes in, which can accelerate the training by randomly sampling a small subset of points in each step, instead of the whole dataset. The number of points in each mini-batch is called *batch size* and algorithms using mini-batches are referred to as *stochastic* [2, 25]. Performing gradient descent using mini-batches and a given, not computed, learning rate results in *stochastic gradient descent* (SGD) [26].

In addition to computational efficiency, mini-batches has many other advantages over the entire dataset. The most important advantage is that per-step computation time only grows with the batch size, independent of the size of the training set [2]. Let me take an example, if we double a training set by inserting each sample twice, then batch learning will require

twice the computational effort, while stochastic methods stay unaffected [25]. In practice there are not many such extreme examples, but there may be many examples with similar contributions to the gradient [25].

The choice of batch size is influenced by several factors [1] (list was also in the Seminar-Paper):

- A larger batch size provides a more accurate estimate, but with sublinear returns. If we estimate the mean of n examples, the standard error is given by $\frac{\sigma}{\sqrt{n}}$ where σ is the true standard deviation of the sample values [1]. A 100 times larger batch leads to 100 times more computation, but only a 10 times more accurate gradient.
- Too small batches underutilize parallel capabilities of modern hardware. Below a minimum size, no speedup in processing each batch is achieved.
- Certain hardware, like GPUs, work best with specific sizes, typically powers of 2.
- The main limiting factor: Memory requirements scale linearly with the batch size.

Compared with batch size, it's much more difficult to choose an appropriate learning rate. Normally, the learning rate α_k is gradually decayed from a selected initial value α_0, staying nearly constant in the early iterations [2]. So far, many different learning rate schedules have been proposed, like *cyclical learning rate* schedule in [27].

Anyway the learning rate sequence $\{\alpha_k\}_{k\in\mathbb{N}}$ should fulfill the Robbins-Monro conditions [25] to guarantee the convergence of SGD:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty \tag{2.33}$$

It is important to choose an appropriate learning rate for the training performance. Too small learning rates will make the progress pretty slow or even stuck at a high loss-value plateau, whereas a too-large one might cause oscillation in progress or even divergence [2].

The main advantage of SGD is quick initial progress, even in large Datasets. However, it is often outclassed by the newer *adaptive learning rate* algorithms. They keep a separate learning rate for each parameter and adapt these rates throughout training [2].

2.4.3 AdaGrad

Adaptive learning rate algorithms can be seen as methods that approximate the so-called empirical Fisher matrix [3], whereas the empirical Fisher matrix itself somewhat approximates the second-order curvature information [3]. In the case of AdaGrad [28], its approximator matrix G_t is the sum of the outer products of all t past gradients:

$$G_t = \sum_{i=1}^{t} g_i g_i^\top, \tag{2.34}$$

with $g_i = \nabla L(\boldsymbol{W}_i)$ [28].

The learning rate is inversely proportional to square root the sum of all historical squared values. And the sum is precisely $\text{diag}(G_t) = \sum_{i=1}^{t} g_i \circ g_i$ ($\circ$ denotes element-wise product). To avoid division by zero, a small constant $\delta \approx 10^{-7}$ is added. Mathematically, this step can be written as follows:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \frac{\alpha_k}{\delta I + \sqrt{\text{diag}(G_t)}} \circ \nabla L(\boldsymbol{W}_k) \tag{2.35}$$

One way to improve efficiency is to compute the diagonal of G_t directly. For this, we accumulate the current sum in r_k and add the new squared gradients in each step: $r_{k+1} = r_k + \nabla L(\boldsymbol{W}_k) \circ L(\boldsymbol{W}_k)$ with $r_0 = 0$. Then we can write the final weight update as follows:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \frac{\alpha_k}{\delta + \sqrt{r_{k+1}}} \circ \nabla L(\boldsymbol{W}_k) \tag{2.36}$$

Using the simplified notation from [2], this derivation was a very short summary of the work in [28].

Because the learning rate is inversely proportional to the sum of gradients, parameters with relatively larger partial derivatives decreased fast, while parameters with small derivatives decrease slowly. As a consequence, AdaGrad can make greater progress on surface regions with gentler slopes [2]. AdaGrad is originally designed for convex optimization, but it performs well for some types of deep networks.

2.4.4 RMSprop

RMSprop [29] improves on AdaGrad by changing the gradient accumulation into an exponentially weighted moving average [2]. By using this moving average, a decay rate ρ is introduced, which can control the number of gradients that are considered in each update.

The update step of RMSprop works exactly like AdaGrad, except for the accumulator r_k

$$r_{k+1} = \rho r_k + (1-\rho)\nabla L(\boldsymbol{W}_k) \circ L(\boldsymbol{W}_k) \tag{2.37}$$

When applied to a convex function, AdaGrad can converge very quickly. For non-convex functions, we may pass a lot of different structures and it might finally arrive at a locally convex region [2]. One problem AdaGrad may have is that the old historical entries may make the learning rate shrink too much before reaching a convex region. Thanks to the decaying average, RMSprop gets rid of old information and avoids such slowdowns. Inside a convex structure, it converges just like an instance of AdaGrad initialized within the structure [2]. RMSprop is considered "one of the go-to methods" [2] used for training, by reason of its effectiveness at optimizing deep neural networks.

2.4.5 Adam

Compared with previous presented adaptive learning rate algorithms, Adam [30] (short for adaptive moment estimation) is the most sophisticated. Similar to RMSprop, Adam also

applies moving average. However, in addition to r_k, it also employs an exponential moving average s_k of the gradients (not squared). And both have their own decay rate: β_1 for s_k and β_2 for r_k. In [30], s_k is considered an evaluation of the 1_{st} mean moment and r_k an evaluation of the 2_{nd} raw moment of the gradient. The initialization of these moving averages as zero vectors leads to moment estimates "biased towards zero" [30]. Adam corrects these biases with the estimates $\widehat{s_k} = s_k / \left(1 - \beta_1^k\right)$ and $\widehat{r_k} = r_k / \left(1 - \beta_2^k\right)$. Put all the things together, we can get the following result procedure [2, 30]:

Algorithm 2: Adam

Require: α_0: Initial learning rate (and a schedule to compute α_k)
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $L(\boldsymbol{W})$: Loss function with parameters W
Require: $\boldsymbol{W}_0$: Starting point

$s_0, r_0 \leftarrow 0$ // Initialize moment estimates
$k \leftarrow 0$
while $\boldsymbol{W}_k$ *not converged* **do**
  $k \leftarrow k + 1$
  $\alpha_k \leftarrow \alpha$ // Compute or use given learning rate
  $g_k \leftarrow \nabla L\left(\boldsymbol{w}_{k-1}\right)$ // Gradient at current step
  $s_k \leftarrow \beta_1 s_{k-1} + \left(1 - \beta_1\right) g_k$ // Biased 1_{st} moment
  $\widehat{s}_k \leftarrow \frac{s_k}{\left(1-\beta_1^k\right)}$ // Correct bias in 1_{st} moment
  $r_k \leftarrow \beta_2 r_{k-1} + \left(1 - \beta_2\right)\left(g_k \circ g_k\right)$ // Biased 2_{nd} moment
  $\widehat{r}_k \leftarrow \frac{r_k}{\left(1-\beta_2^k\right)}$ // Correct bias in 2_{nd} moment
  $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} - \alpha_k \frac{\widehat{s}_k}{\left(\delta + \sqrt{\widehat{r}_k}\right)}$ // Update the weights
_

Compared to RMSprop, Adam has additionally the bias-correction term for the accumulated gradient moments. Under the situation where β_2 (ρ for RMSprop) is close to 1, this results in large stepsizes and possibly even divergence [30]. Adam is invariant to gradient rescaling [30] and it's considered robust to different hyperparameters [2]. Overall, Adam combines the advantages of RMSprop and AdaGrad. And in lots of scenarios, Adam performs at least as well as RMSprop. And for sparse gradients, Adam matches AdaGrad, while both outperform RMSprop [30].

2.5 Second-Order Optimization Algorithms

The general non-linear optimization problems are challenges in network learning. For instance, parameters are usually deeply coupled and have strong connections or dependencies. Besides, different directions in parameter space might have different variations in scale. These issues are problems for gradient descent (first-order) and it must intensely lower its learning rate to avoid instability [3].

Compared with first-order optimization algorithms, second-order optimization methods can deal with the problem of scale and curvature variations along different directions better: They rescale the gradient along the different "*eigen-directions*" of the *curvature* matrix B

w.r.t their corresponding eigenvalue (curvature) [3].

2.5.1 Newton's Method

Newton-Raphson, or simply *Newton's* method, is one of the most classic second-order methods, with all other second-order methods deriving from it [3].

Around the current point $\boldsymbol{W}_k$, we approximate the loss function $L(\boldsymbol{W}_k + \delta)$ by a local quadratic model using the curvature matrix B [3]:

$$L(\boldsymbol{W}_k + \delta) \approx \frac{1}{2}\delta^\top B_k \delta + \nabla L(\boldsymbol{W}_k)^\top \delta + L(\boldsymbol{W}_k) \tag{2.38}$$

In case of standard Newton's method, B_k is equal to the Hessian H and the quadratic model becomes a second-order Taylor series expansion ignoring higher derivatives [8]. During the update step we have to solve the linear system $B_k\delta = -\nabla L(\boldsymbol{W}_k)$, which is also called the Newton equation. This yields the Newton step:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - B_k^{-1}\nabla L(\boldsymbol{W}_k) \tag{2.39}$$

If we are in the vicinity of the solution $\boldsymbol{W}^*$ and the Hessian is positive semidefinite, then Newton's method converges quadratically towards $\boldsymbol{W}^*$ [24].

The original Newton's method could run into problems when used for neural networks training, as a result of the highly non-convex loss function. In non-convex regions, the Hessian is indefinite, making Newton's method step in the wrong direction [3]. One way to tackle this issue is to apply *damping* techniques. Arguably the simplest is called *Tikhonov regularization* [31] and adds, for a certain positive τ, a multiple of the identity matrix τI to H, then we can obtain $B = (\tau I + H)$, which is again positive definite [24, 3]. Tikhonov regularization works pretty well if H has negative eigenvalues close to zero. However, for extreme negative curvature, τ could be very large, which can cause B dominated by the τI term. Then, Newton's method converges to SGD, with $(1/\tau)$ times the stepsize [2].

In addition to structural problems, the main disadvantage of Newton's method is its computational cost. Imaging we have a deep network with $\|W\| > 10^7$ parameters, then it's impractical and expensive to compute or store the Hessian with $\|\boldsymbol{W}\|^2$ entries. Besides, Newton's method requires inverting the Hessian, in time $O\left(\|\boldsymbol{W}\|^3\right)$, in every iteration [2].

2.5.2 Fast exact Multiplication by the Hessian

Because it's very expensive to compute Hessian directly, we can rather access its curvature information in a much cheaper way. More specifically, in [32] a method to compute the Hessian vector product Hv for any v in just two (instead of $\|W\|$) backpropagations was proposed.

[32] has introduced a new differential operator $\mathcal{R}\{\cdot\}$, which is defined as follows:

$$\mathcal{R}_{\boldsymbol{v}}\{f(\boldsymbol{W})\} = \frac{\partial}{\partial r} f(\boldsymbol{W} + r\boldsymbol{v})\Big|_{r=0} \tag{2.40}$$

Then we can simply write $H\boldsymbol{v}$ as $\mathcal{R}_v\{\nabla L(\boldsymbol{W})\}$, and we can compute this in just two backpropagations. More specifically, for a twice continuously differentiable function f, we can get:

$$H\boldsymbol{v} = \mathcal{R}_{\boldsymbol{v}}\{\nabla L(\boldsymbol{W})\} = \left(\frac{\partial}{\partial r}\nabla_{\boldsymbol{W}} L(\boldsymbol{W} + r\boldsymbol{v})\right)\Big|_{r=0} = \nabla_{\boldsymbol{W}}\left(\nabla L(\boldsymbol{W})^\top \boldsymbol{v}\right) \tag{2.41}$$

The proof of the third equality can be found in [4]. The resulting formula is both efficient and numerically stable [32].

2.5.3 The Newton-CG Algorithm

A lot of approximate second-order algorithms have been designed so that they can use the advantages of Newton's method with little or even no computational burden. Among them, the most outstanding ones include *nonlinear Conjugate* Gradients and quasi-Newton methods like *(L-) BFGS* [24], and more recently the *Hessian-Free* and *K-FAC* algorithms [3]. In practice many of them have shown better performance than first-order methods. Seeing how especially the latter K-FAC method runs much faster than plain gradient descent [3], we think that the first-order methods are chosen mainly because of their ease of implementation. Thus, we are going to focus on a relatively simple approach here.

The main computational burden of Newton's method is from solving the system of linear equations $B_k\delta = -\nabla L(\boldsymbol{W}_k)$. To avoid the expensive computation, we don't solve them directly. Instead, we approximate the solution using the (linear) Conjugate Gradients method (CG). This approach has already been proposed in [33] and is known as Truncated Newton. This method originally used numerical approximation to calculate the Hessian-vector product. Later in [4] the use of the fast exact product (see the last subsection) is added. To emphasis the use of this product, we call the resulting algorithm Newton-CG.

In a nutshell, to solve $Ax = b$ CG produces a set of directions $\{p_0, p_1, \ldots, p_n\}$ conjugate with respect to the matrix A. That is, more specifically, $\forall i \neq j : p_i^\top A p_j = 0$. In [24] there is a detailed derivation of the updates. You can find the complete CG method in algorithm 3. Notice that the algorithm only requires Matrix vector products, and never needs A itself. As we solve the newton equation, $A = H$ and by the previous section, we can compute the Hessian-vector products efficiently.

Algorithm 3: Conjugate Gradients

Require: A, b : We want to solve $Ax = b$ for x.
Require: x_0 : Initial estimate for x .

$r_0 \leftarrow Ax_0 - b, p_0 \leftarrow -r_0, k \leftarrow 0$
while r_k *too large* **do**
  $\alpha_k \leftarrow \frac{r_k^\top r_k}{p - k^\top A p_k}$ `// Compute step size`
  $x_{k+1} \leftarrow \alpha_k p_k$ `// Apply step`
  $r_{k+1} \leftarrow r_k + \alpha_k A p_k$ `// Compute new residual`
  $\beta_{k+1} \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$ `// Factor making` p_{k-1} `and` p_k `conjugate w.r.t. A`
  $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$ `// Compute new step direction`
  $k \leftarrow k + 1$

One last issue we should notice is that if the gradient vector points in a direction where the curvature is negative, or simply because of the numerical ill-conditioning, the CG-generated solution p_k may not be feasible [33, 4]. Thus, we need to check $\nabla L\left(\boldsymbol{W}_k\right)^{\top} p_k$ against a small positive threshold. And if this term is larger, then it's not feasible and we simply revert to using the steepest descent direction [4].

The resulting procedure can be found in Algorithm 4. Notice that inside the CG ()-subroutine, the product $(H+\tau I)v$ should be computed as $Hv+\tau v$ using the efficient Hessian-vector product.

Algorithm 4: (Truncated) Newton-CG

Require: $L(\boldsymbol{W})$: The chosen loss function with parameters $\boldsymbol{W}$
Require: $\boldsymbol{W}_0$: Starting point
Require: τ : Tikhonov regularization/damping factor

$k \leftarrow 0$
while $\boldsymbol{W}_k$ *not converged* **do**
  $k \leftarrow k+1$
  $p_k \leftarrow \mathrm{CG}\left((H+\tau I), -\nabla L\left(\boldsymbol{W}_k\right)\right)$
  if $\nabla L\left(\boldsymbol{W}_k\right)^{\top} p_k > \tau$ **then**
    $p_k \leftarrow -\nabla L\left(\boldsymbol{W}_k\right)$
  $\alpha_k \leftarrow \alpha$ `// Compute or use a given step size.`
  $\boldsymbol{W}_k \leftarrow \boldsymbol{W}_{k-1} + \alpha_k p_k$

3 Implementation

In this chapter we describe in detail how Newton-CG algorithm was integrated with the TensorFlow library [34]. Then we describe the specific machine learning problem we focused on and the models we've implemented. Additionally, we list the limitation of our implementation and computational setup used in the following chapter to compare the algorithms' training performance.

3.1 Libraries

3.1.1 Intro to TensorFlow 1

TensorFlow is one of the widely used libraries for implementing Machine learning and other algorithms involving a large number of mathematical operations. The core component of TensorFlow is the *computational graph* and *Tensors* which traverse among all the nodes through edges.

Tensor

Tensors in TensorFlow are a kind of data structure that is used to represent all the data (in TensorFlow). More specifically, they are *multi-dimensional arrays* with all the entries having a uniform data type (called `dtype`). So basically Tensors are very similar to `np.arrays`, but with some differences: `np.arrays` are by default computed only using CPU whereas computation of Tensors are designed better to take full use of GPU or TPU. That's why Tensors are more appropriate to implement deep learning algorithms.

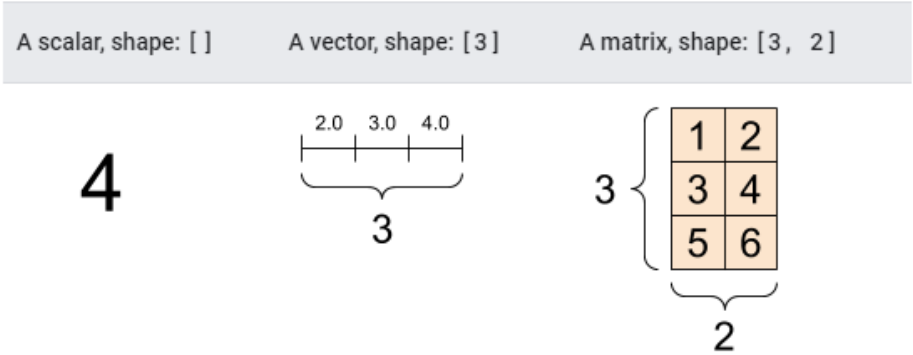


Figure 3.1: Three simple cases of Tensors: scalar, vector and matrix.
Source: TensorFlow

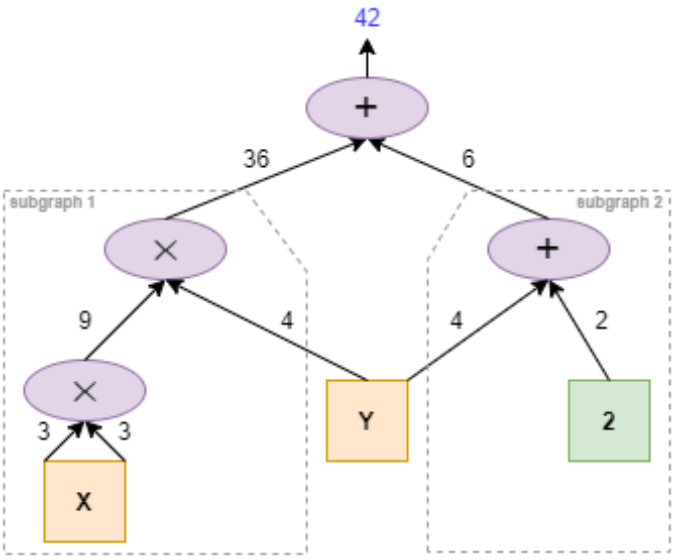


Figure 3.2: An example of computational graph that consists of two independent subgraphs.

Computational graphs

After introducing the fundamentals of TensorFlow, we can now dive into how the computation in TensorFlow works. Different from the native python computation mechanism which computes operation by operation, in TensorFlow we need to first create a computational graph that defines the needed computations. Then we pass the data as input to the graph, and the graph will perform the specified calculations and finally give the calculation results.

The graph is specifically a "stateful dataflow graph" that consists of *nodes* and *edges* [34]. Nodes represent operations such as addition, multiplication and so on, whereas edges indicate the dependencies between nodes [34]. The calculation graph is essentially a logical graph constructed by TensorFlow in memory. The calculation graph can be divided into multiple independent subgraphs and run in parallel on multiple different CPUs or GPUs [34].

There are three types of calculational graphs in TensorFlow, namely *static calculation graphs*, *dynamic calculation graphs*, and *Autograph*. Dynamic calculation graphs and Autograph are used in TensorFlow2, which is irrelevant to our thesis.

In TensorFlow1 static calculation graphs are used: We first need to create a computational graph using the operations in TensorFlow. Then we need to create a *Session*, in which the graph is executed explicitly. Almost all of its computations are executed using C++ on the TensorFlow core, which is more efficient. In addition, the static graph will optimize the calculation steps by eliminating common subexpressions [34].

Variable

So far we have only covered the regular Tensors that are constant and not mutable. It means through operations we can only create new Tensors rather than change them in place. Most regular Tensors are only stored during one execution of the graph, which is not suitable for parameters of the model. And that's where *Variables* come in. A Variable is a mutable

tensor that can be stored across executions of the graph [34]. We can change the value of a Variable through stateful operations such as `var.assign` or `var.assign_add` and so on, where `var` is a Variable instance.

The parameters of machine learning models are usually stored as Variables. An Optimizer can be used to update these parameters during training. This happens in two methods: `compute_gradients()` and `apply_gradients()`. `compute gradients()` computes the gradients of the loss w.r.t. the trained variables and `apply gradients()` applies one step of the optimization algorithm using the computed gradients.

3.1.2 Intro to Keras

In the previous subsection we covered how computation in TensorFlow works (computational graphs). However, in practice it's not efficient enough to build the graph directly using simple operations such as addition. And it's also error-prone to do it this way. To make life easier, we use a higher-level API called *Keras* to do all this stuff automatically.

Keras is a high-level neural networks API, running on top of TensorFlow. It's packaged in TensorFlow as the module `tf.keras`. The core of keras are *models* [35].

Before we dive into models of Keras, let's first have a look at layers, which are components of a model. A layer is composed of a *tensor-in tensor-out* computation function and states which are stored as Variables in TensorFlow [35]. There are already many built-in layers in Keras, such as the fully-connected layers, convolutional layers and so on. In this thesis we have used the built-in `Embedding` layer, `SimpleRNN` layer, `LSTM` layer and `Bidirectional` that makes the recurrent networks bidirectional.

Models are groups of layers with added training and inference features [35]. They usually have one or more `inputs` and one or more `outputs` connected by the layers of the model. One important type of models in Keras is *Sequential* models. A Sequential model is composed of a series of layers in sequence, where each layer has exactly one input and one output Tensor. The advantages of Sequential models are convenience and efficiency (for most problems). However, since they are not allowed to have multiple inputs and outputs, they can't support non-linear network topology (e.g. a residual connection) [35].

3.2 Integrating the Newton-CG algorithm into TensorFlow

Newton-CG is an optimizer that implements the algorithm of Newton-CG (see section). It's a subclass of `tf.python.keras.optimizer_v2.optimizer_v2.OptimizerV2`, implemented by Julian Suk in his Master's thesis [4].

In general optimization problems, the parameters of the model can be continuously optimized by minimizing the error/loss. And two main steps of optimization are:

1. `_compute_gradients(self,loss,var_list)`: Compute gradients of `loss` for the variables in `var_list`. This is the first part of `minimize()`.
2. `apply_gradients(self,grads_and_vars)`: Apply gradients to variables. This is the second part of `minimize()`. It returns an `Operation` that applies gradients.

When applying the gradients, for each Variable that is to be optimized, the method `_resource_compute_dense(self,grad, var)` is called with that Variable and its earlier computed

gradient. Within this method, one step of updating (for that Variable/parameter) proceeds. This is also where the logic of Newton-CG happens.

The main logic in `_resource_compute_dense(self,grad, var)` can be generalized to three steps:

1. Compute one step: `step = self._newton_step(self,grad, var, ...)`
2. Scale the step: `scaled = math_ops.multiply(step, coefficients['lr_t'])`
3. Update Variable: `return state_ops.assign_add(var, scaled, ...)`

`_newton_step(self,grad,var)` returns the descent direction p_k in the current step. More specifically, `_newton_step(self, grad,var)` first approximate a solution p_{k1} for the Newton equation $Bx = -\nabla_{\text{var}} f$ using CG algorithm (implemented by `_cg_solve(self, cg_tol,cg_max_iter)`), where f is the loss and $\nabla_{\text{var}} f$ is the gradient of f w.r.t `var` . Notice that as we have used Tikhonov regularization (implemented by `tikhonov()`), B is equal to $(H + \tau I)$ instead of H, where H is the Hessian of f w.r.t `var`, τ is the damping factor and I is the identity matrix. Besides, we use Pearlmutter algorithm to compute Hessian-vector-product (implemented by `_pearlmutter_hessian()`) to efficiently compute $(H + \tau I)v$. After we get the solution p_{k1}, then as suggested in [4], `_newton_step(self,grad,var)` compares $(\nabla_{\text{var}} f)^\top p_{k1}$ to the threshold τ. If $(\nabla_{\text{var}} f)^\top p_{k1}$ is larger, then feasibility is not guaranteed and we take a simple gradient descent step instead:

$$p_k = -\nabla_{\text{var}} f$$

Otherwise:

$$p_k = p_{k1}$$

The last two steps in `_resource_compute_dense(self,grad, var)` then scale the direction with the learning rate `coefficients['lr_t']` and finally update the Variable.

To explain the loop body of `_cg_solve(self, cg_tol,cg_max_iter)`, we refer to Figure 3.3, which shows the code snippet and each equivalent line in pseudocode. The callable `Ax` is passed to `_cg_solve` within `newton_step`. In our implementation, it equals to `tikhonov()` . The loop body is executed inside a `tf.python.ops.while v2.while_loop` and it terminates whether if `r` is less then `cg_tol` or the `cg_max_iter` times of iterations have been executed.

```
1  # one CG iteration to approximate Ax=b
2  rtr = self._vv(r, r)  # r is the residual
3  axp = Ax(p)  # Ax(p) computes A*p
4  alpha = math_ops.divide(rtr, self._vv(p, axp))
5  x = math_ops.add(x, math_ops.multiply(p, alpha))
6  r = math_ops.subtract(r, math_ops.multiply(axp, alpha))
7  rtr_new = self._vv(r, r)
8  beta = math_ops.divide(rtr_new, rtr)
9  p = math_ops.add(r_ret, math_ops.multiply(p, beta))
10 return x, r, p  # the new values of x, r and p
```

Lines 2 to 4:
$\beta_{old} \leftarrow r^\top r$
$\alpha \leftarrow (r^\top r)/(p^\top Ap)$
Line 5: $x \leftarrow x + \alpha p$
Line 6: $r \leftarrow r - \alpha Ap$
Lines 7 to 9:
$p \leftarrow r + p\frac{r^\top r}{\beta_{old}}$

Figure 3.3: The while-loop body of `_cg_solve()` and the corresponding pseudocode. Source:[1]

3.3 Sentiment Analysis Problem

In this thesis we focus on *Sentiment Analysis* problems. More specifically, we focus on IDMB Movie Reviews Classification, where we train a sentiment analysis model to classify movie reviews as either positive (1) or negative (0), based on the text of the reviews. This is an example of binary or two-class—classification, an important and widely applicable kind of machine learning problem.

3.3.1 Dataset

The dataset we use is downloaded from Kaggle, it contains totally 45000 sample reviews, and is split into a *training* and a *validation* set. Among them, the training dataset has 40000 sample reviews and the validation dataset has 5000 sample reviews.

Even though I have great interest in Biblical movies, I was bored to death every minute of the movie. Everything is bad. The movie is too long, the acting is most of the time a Joke and the script is horrible. I did not get the point in mixing the story about Abraham and Noah together. So if you value your time and sanity stay away from this horror.	0

Figure 3.4: One sample IDMB movie review and its label.

3.3.2 Preprocessing

As stated in Subsection 2.1.2, we can't directly input the raw sentences to our model, but first do some preprocessing. The main preprocessing we did is word embedding, i.e. turning each word into a vector (of the same length) like $(1, 2, 3, \cdots, 300)$ and therefore one sentence is in form of a matrix.

A good word embedding system plays a very critical part in the performance of NLP tasks [36]. Because our dataset is not relatively large, we decided to use a pretrained word embedding system rather than train it by ourselves. We downloaded an english-pretrained-word-embeddings from fastText. It's trained on Common Crawl and Wikipedia and it contains totally 2000000 words and their corresponding vectors, each of length 300.

```
2000000 300
, 0.1250 -0.1079 0.0245 -0.2529 0.1057 -0.0184 0.1177 -0.0701 -0.0401 -0.0080 0.0772 -0.0226 0.0893 -0.0487 -0.0897 -0.0835 0.0200 0.0273 -0.0194 0.096
the -0.0517 0.0740 -0.0131 0.0447 -0.0343 0.0212 0.0069 -0.0163 -0.0181 -0.0020 -0.1021 0.0059 0.0257 -0.0026 -0.0586 -0.0378 0.0163 0.0146 -0.0088 -0.
. 0.0342 -0.0801 0.1162 -0.3968 -0.0147 -0.0533 0.0606 -0.1052 0.0005 -0.0360 0.0257 0.0177 0.0285 0.0037 -0.0419 0.2374 0.0073 -0.0303 -0.0578 -0.0616
and 0.0082 -0.0899 0.0265 -0.0086 -0.0609 0.0068 0.0652 0.0106 -0.0475 -0.0076 -0.0023 0.0009 -0.0067 -0.0226 -0.0066 -0.0725 0.0203 0.0215 -0.0507 0.0
to 0.0047 0.0281 -0.0296 -0.0108 -0.0620 -0.0532 -0.0980 0.0970 -0.0864 -0.0425 0.0272 -0.0376 -0.0428 0.0374 -0.0040 -0.0354 0.0064 0.0182 -0.0584 0.0
of -0.0001 -0.1877 -0.0711 -0.4632 0.0002 0.0115 -0.0588 0.0574 -0.0275 -0.0036 -0.0098 -0.0414 0.0935 0.0541 -0.1960 -0.2909 -0.0632 0.0466 -0.0160 0.
a 0.0876 -0.4959 -0.0499 -0.0937 -0.0472 -0.0211 0.2624 0.0269 -0.0900 -0.0363 -0.0306 -0.0142 0.0337 0.1001 -0.1523 0.6262 -0.0115 -0.0063 0.0011 0.01
</s> 0.0731 -0.2430 -0.0353 -0.3631 0.0380 0.0580 -0.1924 0.0283 -0.0737 0.0127 0.0917 -0.0233 0.0858 -0.0136 0.0569 0.1742 -0.0814 -0.0681 -0.0338 -0.
in -0.0140 -0.2522 0.0715 -0.0246 -0.0637 -0.0364 0.1619 0.0268 -0.0475 0.0192 0.0102 0.0243 0.0691 -0.0131 0.0100 0.0384 0.0049 -0.0646 0.0062 -0.2644
```

Figure 3.5: A snippet of the word embeddings. The first line contains the total number of words/vectors and the length of each vector. Starting from the second row, the first column of each row represents the word, and the following columns are the corresponding vector.

Before word embedding, we first need to tokenize the sentences. For instance, we transform "I like you." to `['I','like','you.']`. However, this would cause a problem because words like "you." can't be embedded into a vector since there is no word "you." in the pretrained word embeddings. Instead, there are only "you" and ".". So before tokenization, we split the words and punctuations using regular expressions: "I like you." to "I like you .". So that these kinds of words can be embedded.

3.3.3 Finding suitable hyperparameters

In our problem, we have mainly two kinds of hyperparameters: One is the hyperparameters of the optimizers and the other is the hyperparameters in the preprocessing procedure.

For the four first-order algorithms Adam, RMSprop, AdaGrad and SGD we stick mostly to default hyperparameter values, consistent with literature [2]. The following values have been used to train all three models:

- Adam: $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$ (for stability).
- RMSprop: $\rho = 0.9$ and $\epsilon = 10^{-7}$ (for stability).
- SGD: Plain SGD was not able to make meaningful progress. Adding momentum [37] of 0.9 greatly improved its performance.

Optimizer	**Learning Rate**
Adam	0.0001
RMSprop	0.0001
AdaGrad	0.01
SGD	0.001

Table 3.1: Learning Rates of the optimizers.

To find a good learning rate, we use a rather simple heuristic: First we sweep over the set $\left\{10^{-i} : i \in [6]\right\}$ and train respectively for 20 epochs. Then we pick the value where the validation loss is the lowest.

Another thing I would like to mention is the maximal length of sentences. In the original dataset we have sentences of different lengths that vary from 32 to 13704. The average length of sentences is 1310 in training dataset and 1297 in validation dataset. However, in the preprocessing procedure we need to pick a *maximal length* and make all the sentences/sequences of this length. Similar to finding a good learning rate, we loop over set $\{256, 512, 1024, 2048\}$ and we found there is a trade-off between performance and computational cost, as you can see in the following figure:

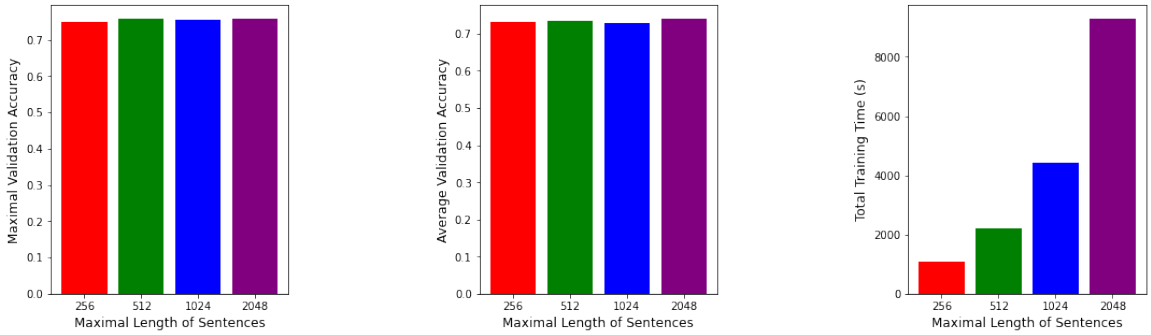


Figure 3.6: Comparison of different maximal lengths (Adam optimizer and epochs = 20).

As you can see, with the exponential growth of max length, the performance of the model has not improved significantly, but the total training time has increased exponentially. Therefore we choose 256 as the maximal length since it gives relatively good performance with reasonable computational cost.

3.3.4 Model and loss functions

Model

In this thesis we have implemented two models. One is RNN-based and the other is based on a self-attention layer. More specifically, we use `tf.keras` to implement these models: For RNN-based model we simply use the built-in `Bidirectional` and `SimpleRNN` layers, with the embedding layer and the final fully-connected layer. The pseudo-code is given as follows:

```
model = Sequential()
model.add(Embedding(...))
model.add(Bidirectional(SimpleRNN(...)))
model.add(Dense(1,activation='sigmoid'))
```

Figure 3.7: Pseudo-code of creating the RNN-based model.

For the other model, we implemented a `Self_Attention` layer inherited from superclass `tf.keras.layers.Layer`, since there is no built-in self-attention layer in keras. With other layers like fully connected layer, dropout layer,$\cdots$, the pseudo-code for the model can be written as follows:

```
inputs = Input_layer(...)
embeddings = Embedding(...)(inputs)
x = Self_Attention(...)(embeddings)
x = GlobalAveragePooling1D()(x)
x = Dropout(...)(x)
outputs = Dense(2, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs)
```

Figure 3.8: Pseudo-code of creating the self-attention-based model.

Loss function

As stated in Section 3.3, our problem is a binary classification problem: There are only two labels, either "positive" or "negative". So we can simply output one fractional number: The probability of being label 1. Then we feed it into the binary cross-entropy loss. We can also output two probabilities and use the common categorical cross-entropy loss. In the first model we use binary cross-entropy and in the second model we use categorical cross-entropy.

3.4 Limitations

In this section we describe the three main limitations of our implementation, and how they affect our evaluation setup.

3.4.1 Implementation for sparse case

Sparse Tensors are Tensors that contain a lot of zero values. Specifically, sparse Tensors are used in TensorFlow when taking gradients of an Embedding layer [38]. To do this, the method `_resouce_apply_sparse()` of the optimizers needs to be implemented. There is no algorithmic difference between the dense and the sparse cases, but working with sparse Tensors requires significant implementational effort. Therefore we only consider dense Tensors in this thesis. This means if we use Newton-CG, we are not able to train an Embedding layer. This is another reason why we use pretrained word embeddings.

3.4.2 Unrolling the RNN

Another limitation is an old issue of TensorFlow itself: when dealing with recurrent networks and second-order gradients, we will get the following error if we don't unroll the networks (which is the default case): `TypeError: Second-order gradient for while loops not supported.` This issue has already been raised by other TensorFlow users several years ago but has not been solved yet. Some similar cases can be found here and here.

The way we deal with this issue is to unroll the networks, as suggested by user1128016 in the first case mentioned above. However, as the official documentation of TensorFlow says, unrolling an RNN could be more memory-intensive, especially when the sequences are relatively long. This would lead to the third limitation: Memory Requirements.

3.4.3 Memory requirements

Our programs have certain requirements for memory. The reasons lie in the following three points.

Firstly, we need to load the pretrained word embeddings (about 4GB) into memory, which will result in that the program occupies a part of the memory during the initialization phase.

The second point is caused by the issue from TensorFlow (see the last subsection).

The third point is graph memory requirement. As stated in Subsection 3.1.1, the optimization procedure is first compiled into a TensorFlow graph, before being executed. Same with other Optimizers, the optimization procedure is compiled separately for every trained Variable in the model graph. However, rather than computing first-order gradients, our Optimizer computes second-order gradients. In order to compute these gradients, TensorFlow

copies graph information to each procedure: The amount of graph information copies scales linearly with the depth of Variables. This requires relatively large memory when we have a deep/complex model. Consequently, our two models are not very complex. But fortunately, this already gives insights into the behavior and performance of the Newton-CG algorithm.

3.5 Computational Setup

3.5.1 Hardware

For the most of development and program runs we used a computer provided by the Chair of Scientific Computing at TUM. The machine had 16GB of RAM and a four-core (8 hyper-threaded) Intel Core i7 3770K processor. It had one NVIDIA Titan XP (12GB VRAM) and one Quadro P4000 GPU (8GB VRAM). The Titan GPU was used to train the models.

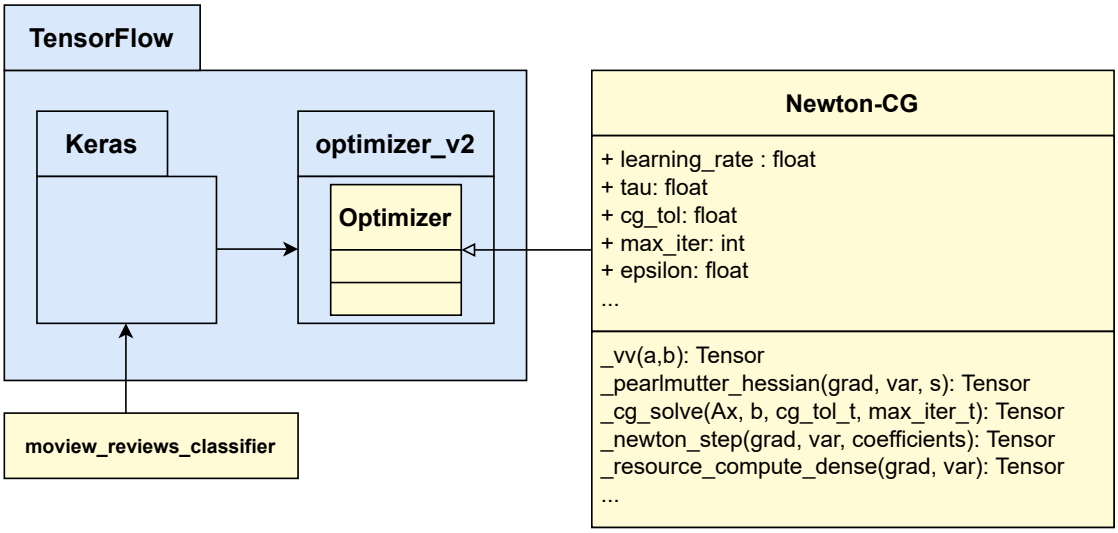


Figure 3.9: UML diagram of the main software components and their relations.

3.5.2 Software

The OS on the computer was Linux Ubuntu. Python 3.7 with TensorFlow version 1.15, Keras 2.3.0 and Numpy 1.19.5 were used. Figure 3.9 shows a UML diagram with the relations of the main classes and packages in this thesis. The *moview_reviews_classifier* refers to the our program scripts. This diagram is only an overview, as only the most relevant classes and relations are shown.

4 Results

In this chapter we compare the performance of the presented Newton-CG algorithm with the state-of-the-art first order optimizers. For this, we train one instance on each of the three model architectures from Subsection 2.1.3, Subsection 2.2.3: RNN-based Network and Self-Attention-based Network.

4.1 RNN based model

4.1.1 finding suitable hyperparameters for Newton-CG

To train the Self-Attention-based model, the first thing we do is to find a good combination of the hyperparameters. We find that 20 CGiterations and a convergence tolerance of 10^{-5} proved accurate. Further decreasing the tolerance or increasing the number of iterations slows down training without benefiting convergence.

As for the learning rate and Tikhonov regularization τ, we find that different values will cause very big differences. Thus we made a specific test, looping over learning rate set $\{0.01, 0.001\}$ and τ set $\{0.1, 10, 1000\}$. The following figure shows the result:

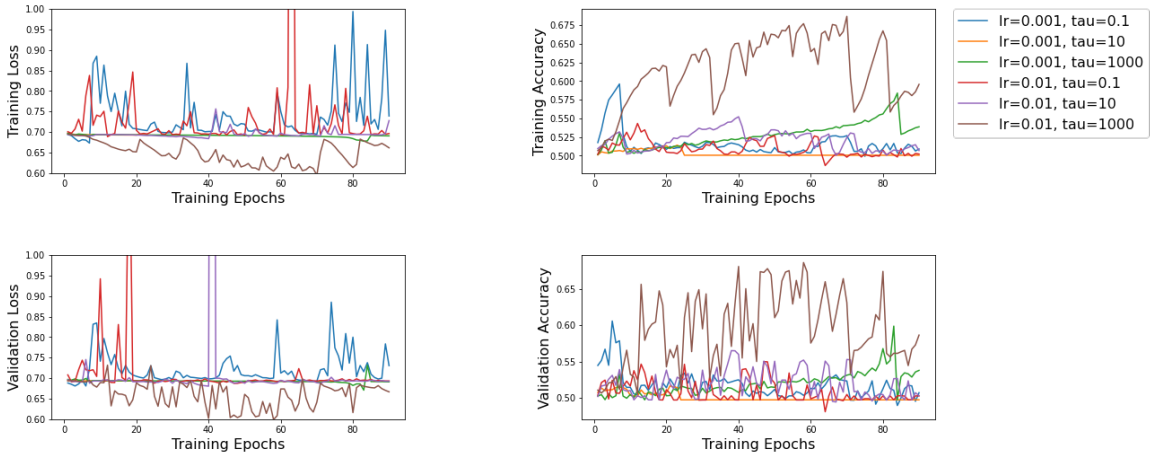


Figure 4.1: Performance comparison of Newton-CG with different combinations of learning rate and tau.

We trained the model for 90 epochs for each specific combination and recorded the loss and accuracy $\left(\frac{\text{number of correct predictions}}{\text{number of predictions}}\right)$ on the training dataset and validation dataset respectively. Compared with the learning rate, τ has a greater impact on the performance of Newton-CG. And the larger the τ, the better the performance of Newton-CG. More specifically, we find that Newton-CG with learning rate $= 0.01$ and $\tau = 1000$ has the best

performance among all the hyperparameter configurations. And only with this configuration, the training loss and validation loss have a downward trend. Besides, we find that Newton-CG's performance is unstable for almost all hyperparameter configurations. To achieve better performance, we use the hyperparameter configuration mentioned above as a standard in the following program runs, unless explicit specification.

4.1.2 Comparing Newton-CG with other optimizers

To compare Newton-CG with other optimizers, We trained the model for 90 epochs with batch size set to 32 (batch size of multiple-of-8 can fit on the Titan XP GPU) using each optimizer. All the first-order optimizers use the hyperparameters listed in Subsection 3.3.3 and Newton-CG use the "standard" hyperparameters mentioned above.

Figure 4.2: Training Loss (left) and Training Accuracy (right) over Training Epochs. The two subfigures in the first row are the original images whereas the two in the second row are the same images with some averaging processing to make the curves more smooth.

Figure 4.2 shows the loss and accuracy on the training dataset. As we can see, Adam and RMSprop have the best performance for both training loss and training accuracy, followed by AdaGrad. The performance of Newton-CG is only slightly better than SGD. Besides, while Adam, RMSprop and AdaGrad show similar curve for both loss and accuracy: rapid improvement in accuracy/decrease in loss in the early epochs followed by slower changes, Newton-CG and SGD show similar curve: Their curves often oscillate, and the amplitude of that of Newton-CG is larger. More specifically, Adam and RMSprop grow rapidly in about the first 5 training epochs and reached a training accuracy of about 75%, and then started to slow down. In the end, they achieved a training accuracy of about 90%. The training accuracy of AdaGrad increased rapidly in about the first 30 training epochs and reached about 71%. Then it increased relatively slowly and in the end it reached about 80%. The accuracy of Newton-CG and SGD does not have a stable maximum. Newton-CG achieves the maximal training accuracy at about the 50th training epoch, which is about 68%. In contrast, SGD achieved the maximum training accuracy of about 61% around the

20th training epoch. Overall, the training accuracy of Newton-CG is higher than that of SGD. The situation of Training Loss is similar to Training Accuracy, so it is omitted here.

Figure 4.3: Validation Loss (left) and Validation Accuracy (right) over Training Epochs. The two subfigures in the first row are the original images whereas the two in the second row are the same images with some averaging processing to make the curves more smooth.

Figure 4.3 shows the loss and accuracy on the validation dataset. The performance of all the optimizers on the validation dataset is quite similar to that on the training dataset. For Adam, RMSprop and AdaGrad, their validation accuracy grows quickly first, then from a certain training epoch starts to slow down. For Newton-CG and SGD, their accuracy is still oscillating, and their amplitude and frequency are greater than those on the training dataset. More specifically, Adam and RMSprop reach a validation accuracy of about 73% at only about the 6th training epoch, and then begin to slow down, and even become overfitting. They achieve the maximal validation accuracy (about 73% and 74% respectively) around the 10th training epoch and in the end their validation accuracy drop to around 71% due to overfitting. The curve of AdaGrad's validation accuracy is similar, but its growth is not as fast as Adam and RMSprop: it grows at a relatively fast rate till around the 30th training epoch and reaches about 72%, then begins to gradually converge, and finally reaches about 73%. It is worth noting that the model is not overfitting using AdaGrad optimizer. Compared with the above optimizers, the validation accuracy of Newton-CG and SGD is more unstable, so they do not have a stable maximum validation accuracy. For Newton-CG, its validation accuracy achieves the maximum (about 68%) around the 58th training epoch. SGD achieves the greatest validation accuracy at the 22nd training epoch (about 64%). Overall, the validation accuracy of Newton-CG is higher than that of SGD during most of the time. The situation of Training Loss is similar to Training Accuracy, so it is omitted here.

4.1.3 Newton-CG with Adam pretrained

In addition to the normal tests, we also tried some experiments to see whether the performance of Newton-CG can be improved. We do it by comparing the normal Newton-CG and Newton-CG with Adam pretrained. For the latter, we first use Adam to train for the first 20 epochs then use Newton-CG to finish the remaining epochs.

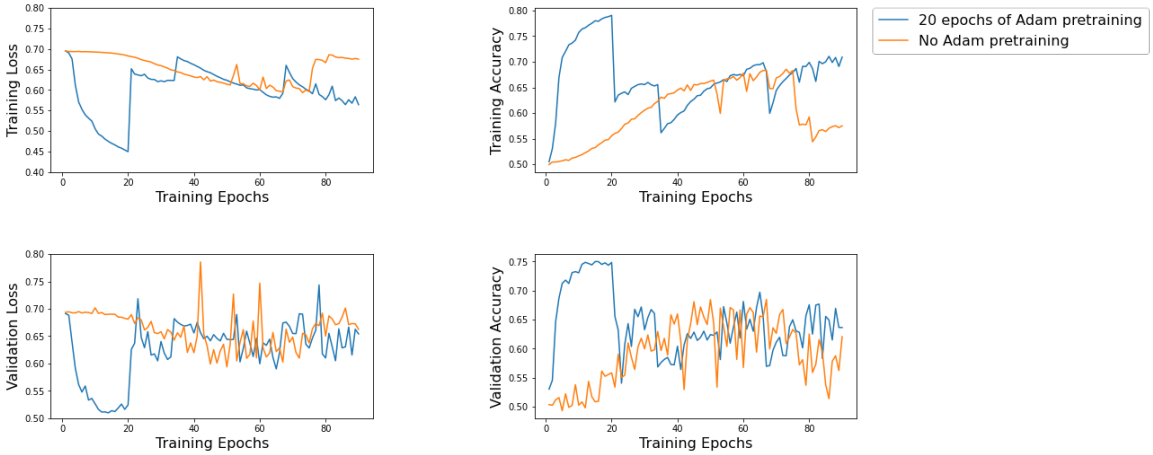


Figure 4.4: Performance comparison of Newton-CG with and without Adam pretrained. The former first uses Adam to train the model for 20 epochs, then uses Newton-CG for the remaining 70 epochs. The latter uses only Newton-CG.

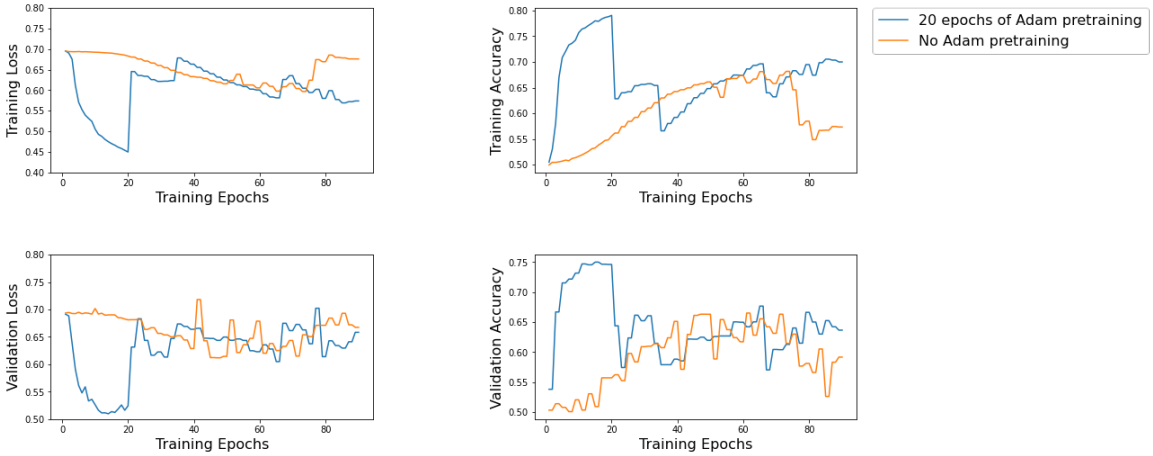


Figure 4.5: Same images as Figure 4.4, but with some averaging processing to make the curve smoother.

Figure 4.4 and Figure 4.5 show the result. We can see that after using Adam pretraining 20 training epochs, the training accuracy and validation accuracy reached about 78% and 75% respectively. However, this does not seem to have a big impact on the optimization of Newton-CG: after the 21st training epoch, the training accuracy and validation accuracy have dropped to about 63% and 54% respectively, which are very close to the accuracy

of training only with Newton-CG. Afterward Newton-CG with Adam pretrained performs similarly to pure Newton-CG: The training accuracy and validation accuracy show an upward trend overall, but it fluctuates from time to time. More specifically, the training accuracy of Newton-CG with Adam pretrained achieves the maximum value around the 90th training epoch (about 71%), and there is still an upward trend with a small oscillation. The training accuracy of pure Newton-cG gets the maximum value (69%) around the 70th training epoch, and then fluctuates and begins to decrease. At About the 90 th training epoch, the training accuracy of pure Newton-CG was only about 56%, which was significantly smaller than that of Newton-CG with Adam pretrained. But this does not mean that Newton-CG with Adam pretrained is better than pure Newton-CG because from the figure we can see that Newton-CG with Adam pretrained still has oscillations, and the frequency and amplitude are similar to pure Newton-CG. The validation accuracy of the two is very similar to the training accuracy, the difference is that their accuracy oscillations on the validation set are larger and more frequent. The situation of loss is similar to accuracy, so it is omitted here.

4.2 Self-Attention based model

4.2.1 finding suitable hyperparameters for Newton-CG

To train the Self-Attention-based model, the first thing we do is to find a good combination of the hyperparameters. Similar to the RNN model case, we find that 20 CG-iterations and a convergence tolerance of 10^{-5} proved accurate. Further decreasing the tolerance or increasing the number of iterations slows down training without benefiting convergence. As for the learning rate and Tikhonov regularization τ, we find that different values will cause very big differences. Thus we made a specific test, looping over learning rate set $\{0.1, 0.01, 0.001\}$ and τ set $\{0.1, 1, 10\}$. The following figure shows the result:

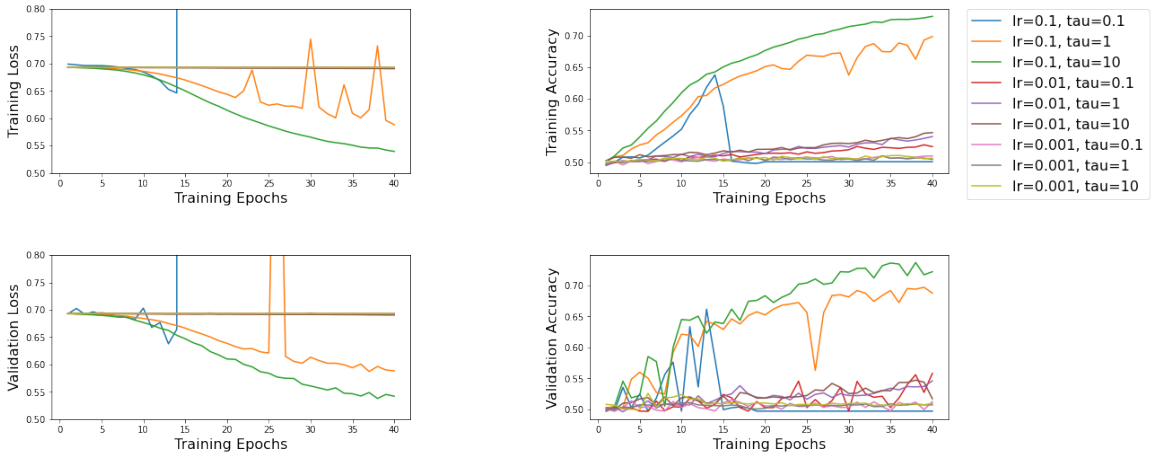


Figure 4.6: Performance comparison of Newton-CG with different combinations of learning rate and tau.

We trained the model for 40 epochs for each specific combination and recorded the loss and accuracy $\left(\frac{\text{number of correct predictions}}{\text{number of predictions}}\right)$ training dataset and validation dataset respectively.

The result shows the performance of Newton-CG with learning rate = 0.1 and $\tau = 10$ beats almost all other combinations on both training and validation datasets. Therefore, in the following program runs, we use this hyperparameter combination as a standard, unless explicitly specified.

4.2.2 Comparing Newton-CG with other optimizers

We trained the model for 100 epochs with batch size set to 32 (batch size of multiple-of-8 can fit on the Titan XP GPU). All the first-order optimizers use the hyperparameters listed in Subsection 3.3.3 and Newton-CG use the "standard" hyperparameters mentioned above.

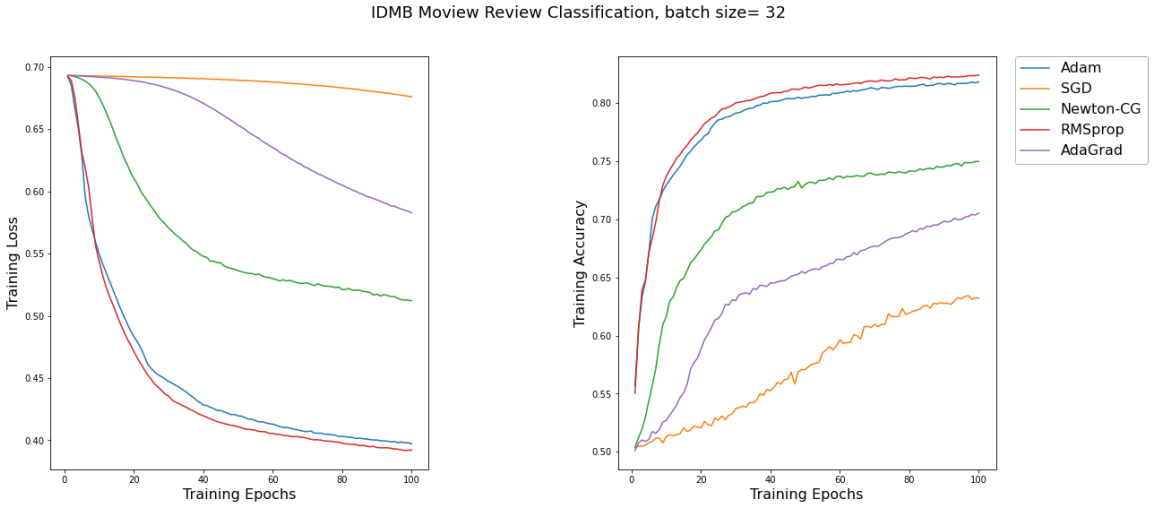


Figure 4.7: Training Loss (left) and Training Accuracy (right) over Training Epochs.

Figure 4.7 shows the loss and accuracy on the training dataset. As we can see, Adam and RMSprop have the best performance for both training loss and training accuracy. Newton-CG also has a relatively good result compared with AdaGrad and SGD. Besides, Adam, RMSprop and Newton-CG show similar performance: rapid improvement in accuracy/decrease in loss in the early epochs, followed by slower changes. More specifically, Adam and RMSprop have reached about 78% in training accuracy at the 20th training epoch, while Newton-CG only reached about 68% and AdaGrad only reached about 61%. In contrast, SGD has the lowest training accuracy, only about 52.5%. After the 20th training epoch, all optimizers except SGD started to become slower. In the end, after slow growth, Adam and RMSprop reached about 82% in training accuracy at the 100th training epoch, while Newton-CG reached about 74% and AdaGrad reached about 70%. SGD was still the lowest, only around 62.5%. In general, the training accuracy of all optimizers is showing an upward trend, among which Adam and RMSprop grow the fastest, followed by Newton-CG, AdaGrad and SGD. The situation of Training Loss is similar to Training Accuracy, so it is omitted here.

Figure 4.8: Validation Loss (left) and Validation Accuracy (right) over Training Epochs. The two subfigures in the first row are the original images whereas the two in the second row are the same images with some averaging processing to make the curves more smooth.

Figure 4.8 shows the loss and accuracy on the validation dataset. The performance of all the optimizers on the validation dataset is quite similar to that on the training dataset: At first growing/decreasing quickly, then becoming slower and slower. However, there are some small differences on the validation dataset. Instead of slowly increasing in accuracy/decreasing in loss in the later epochs, most of the optimizers show convergence in the end. For instance, the accuracy of Adam and RMSprop slowly converges from around the 40th epoch, and finally converges to about 80%. Newton-CG is quite similar, finally converging to about 74%. The convergence of AdaGrad and SGD is a bit slower in contrast: They finally reached about 68% and 62% respectively and are still slowly growing. Besides, the performance of Newton-CG, AdaGrad and SGD is more unstable: Their curve for accuracy is more trembling. However, the curve oscillation amplitude of Newton-CG, AdaGrad and even SGD is getting smaller and smaller.

4.2.3 Newton-CG with Adam pretrained

Besides the normal tests, we also tried some experiments to see whether the performance of Newton-CG can be improved. We do it by comparing the pure Newton-CG and Newton-CG with Adam pretrained. For the latter we first use Adam to train for the first 20 epochs then use Newton-CG to finish the rest epochs.

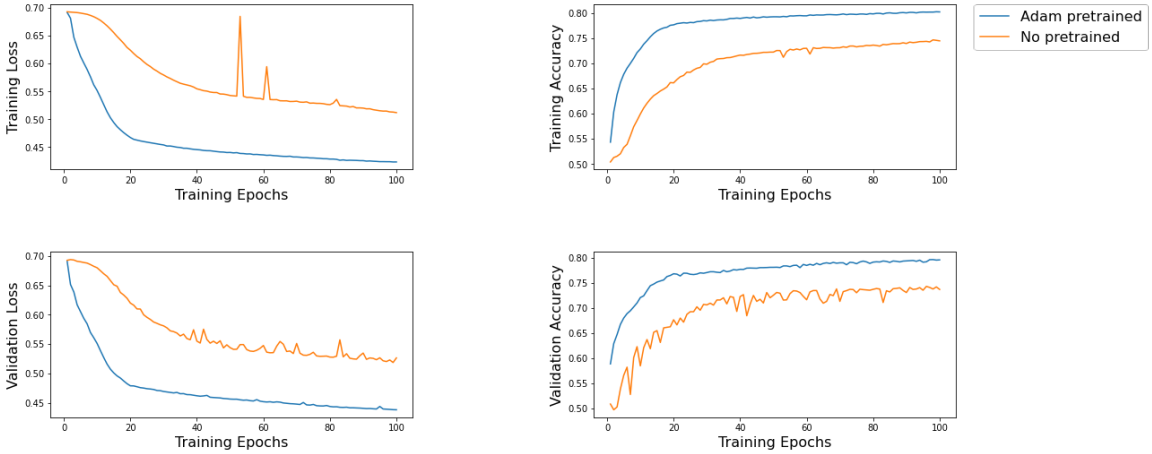


Figure 4.9: Performance comparison of Newton-CG with and without Adam pretrained. The former first uses Adam to train the model for 20 epochs, then uses Newton-CG for the remaining 80 epochs. The latter uses only Newton-CG.

Figure 4.9 shows the result. We have noticed that after pretraining Newton-CG keeps slowly improving the accuracy/decreasing the loss in the later 80 epochs, on both training and validation datasets. The final validation accuracy of Newton-CG with Adam pretrained is about 80%, while pure Newton-CG has only about 74%. Besides, Newton-CG with Adam pretrained shows more stability than the pure Newton-CG, for both loss and accuracy.

5 Conclusion and Outlook

In this thesis, we continued the comparison between Newton-CG and other first-order optimizers, but on NLP problems. For that, we introduced the theory of neural networks we used in this thesis and the theory of common first-other optimizers and Newton-CG.

To compare Newton-CG with the first-order optimizers Adam, RMSprop, AdaGrad and SGD, we implemented two neural network models: One is based on RNN and the other is based on Attention mechanism, which are the two most typical neural network models for NLP. We recorded the loss and accuracy on both the training and validation dataset during the training. Newton-CG showed much better performance on the Self-Attention based model: Higher training and validation accuracy and much more stability. On the Self-Attention model Newton-CG outclassed AdaGrad and SGD, whereas on the RNN model Newton-CG was only a little bit better than SGD.

We think the reason why the performance of Newton-CG is not as good as that of Adam or RMSprop is that the loss function is not convex and there are several local minimums and we didn't reach a global minimum. The reason behind that could be that NLP models (especially RNNs) seem to be easier to become ill-conditioned and the CG algorithm used in Newton-CG can not deal with it very well.

Possible future work could include a more thorough and deeper investigation of the hyperparameters. Also, the technical limitations of our implementation have to be addressed, before the algorithm can be used in practice. Especially the sparse case implementation, so that we can train an Embedding layer. It's also recommended to rewrite the algorithm in other frameworks since TensorFlow is not so friendly to second-order gradients.

In conclusion, Newton-CG's performance on NLP problems, or Sentiment Analysis more specifically, is not as good as on Image Classification problems. Nevertheless, it still has an acceptable performance on the Self-Attention model.

List of Figures

List of Tables

Bibliography

[1] Mihai Zorca. Training Deep Convolutional Neural Networks on the GPU Using a Second-Order Optimizer. *Bachelor's thesis, Technical University of Munich, Munich*, 2020.

[2] I. Goodfellow, Y. Benjio, and A. Courville. *Deep Learning.* MIT Press, 1993.

[3] J. Martens. Second-order optimization for neural networks. *Ph.D. dissertation, University of Toronto*, 2016.

[4] J. Suk. Application of second-order optimisation for large-scale deep learning. *Master's thesis, Technical University of Munich, Munich*, 2020.

[5] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, vol. 61:pp. 85–117, 2015.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag, 2006.

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space.* arXiv:1301.3781 [cs.CL], 2013.

[8] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee. *Recent Advances in Recurrent Neural Networks.* arXiv:1801.01078 [cs.NE], 2018.

[9] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, vol. 5, no. 2:pp. 157–166, 1994.

[10] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, page pp. 1017–1024, 2011.

[11] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato. *Learning longer memory in recurrent neural networks.* arXiv preprint arXiv:1412.7753, 2014.

[12] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks.* arXiv:1909.09586 [cs.NE], 2019.

[13] J. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen.* Master's thesis, Institut fur Informatik, Technische Universitaet Muenchen, 1991.

[14] Michael C Mozer. Induction of Multiscale Temporal Structure. *Advances in Neural Information Processing Systems 4*, pages pp. 275–282, 1992.

[15] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, pages 12(10): 2451–2471, 2000.

[16] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research (JMLR)*, pages 3(1):115—-143, 2002.

[17] Mani Wadhwa. seq2seq model in Machine Learning. *GeeksforGeeks*, 2018.

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. arXiv:1409.3215 [cs.CL], 2014.

[19] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. arXiv:1508.04025 [cs.CL], 2015.

[20] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv:1409.0473 [cs.CL], 2015.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. arXiv:1706.03762 [cs.CL], 2017.

[22] Jianpeng Cheng, Li Dong, and Mirella Lapata. *Long Short-Term Memory-Networks for Machine Reading*. In EMNLP, 2016.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 [cs.CV], 2015.

[24] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science Business Media, 2006.

[25] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[26] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, vol. 22:pp. 400–407, 1951.

[27] L. N. Smith. Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, page pp. 464–472, 2017.

[28] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, vol. 12:pp. 2121–2159, 2011.

[29] T. Tieleman and G. Hinton. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning, 2012.

[30] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *in the 3rd International Conference for Learning Representations, San Diego*, 2011.

[31] A. N. Tikhonov and V. Y. Arsenin. Solutions of ill-posed problems. *Wiley*, page pp. 1–30, 1977.

[32] B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, vol. 6, no. 1:pp. 147–160, 1994.

[33] R. S. Dembo and T. Steihaug. Truncated-newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, vol. 26, no. 2:pp. 190–212, 1983.

[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, ´ B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. *software available from tensorflow.org. [Online]*, 2015.

[35] F. Chollet et al. Keras. *https://keras.io*, 2015.

[36] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[37] B. Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, vol. 4:pp. 1–17, 1964.

[38] Biswajit Sahoo. Indexedslices in tensorflow. https://biswajitsahoo1111.github.io/post/indexedslices-in-tensorflow/, 2021.