UNIVERSITY OF AGDER

An exploration of semi-supervised text classification

Henrik Lien

SUPERVISORS
Daniel Biermann
Morten Goodwin

Master's Thesis
University of Agder, 2021
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

Obtaining labeled data to train natural language machine learning algorithms is often expensive and time-consuming, while unlabeled data usually is free and easy to get. Frequently a large amount of labeled data is required by supervised learning to achieve good text classification performance.

Semi-supervised learning (SSL) for text classification is an exciting area of research. SSL is a technique exploiting unlabeled and labeled data to achieve better classification performance than using labeled data alone and is particularly useful with limited labeled data.

This thesis explores the impact of different parameters on SSL with unsupervised pre-training and supervised fine-tuning for a text classification task. Key to this work is the study of hyperparameters, including the amount of preprocessing data and model size. We examine smaller and larger models, including feed-forward, recurrent, and seq2seq models, used for experimentation. This thesis uses SSL performance as a performance metric. It measures the difference in text classification performance of a model when using the SSL compared to the supervised learning approach. Thus, the SSL performance is an intuitive measure for investigating the benefits of SSL.

We find that the hyperparameter setup significantly impacts SSL performance, and the learning rate has the most impact across all models. We show that larger models reach a higher SSL performance than smaller models, mainly with a smaller preprocessing data amount. However, when scaling the amount of preprocessing data, we see that the recurrent models reach a performance threshold. On the other hand, increasing data for hyperparameter configurations more tuned for SSL, SSL performance improves for the feed-forward model. The combination of fewer model parameters, dropout, and higher learning rate likely causes this.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Text classification has many useful application areas, for example classifying spam, medical texts, or intentions. Manual text classification is unfeasible due to significant time and economical costs. Artificial Intelligence (AI) techniques automate text classification tasks, making text classification cheaper and practical. Within AI, neural networks using traditional supervised learning often require a substantial amount of labeled data to achieve good text classification performance during testing with previously unseen data.

Obtaining labeled data frequently requires expert domain knowledge, for example for labeling spam, medical texts, or intentions. Manual labeling is usually time-consuming and expensive. Therefore, obtaining a large size of labeled data is regularly challenging. In contrast, obtaining an extensive volume of unlabeled data is commonly free and simple. For example, a significant volume of free unlabeled data is created on the internet every day which can be extracted easily.

There is a significant amount of research within the exciting area of semi-supervised learning (SSL) for text classification. The SSL technique exploits both unlabeled and labeled data to achieve improved classification performance compared to using only labeled data. This is useful because it lowers the volume of labeled data required for achieving good text classification performance. It also enables the usage of the enormous amount of unlabeled data. Therefore, SSL makes obtaining data for text classification more practical. In the last few years, there has been a substantial volume of research significantly furthering the state of the art of SSL.

SSL does not always improve performance compared to supervised learning [86] [20] [76] [52] [64]. Experiments in the literature show performance decline, which has probably been under-reported due to publication bias [86]. To the best of our knowledge, there has not been a significant amount of research investigating the impact of parameters on SSL for text classification – in particular with a fixed number of epochs.

This thesis explores the impact of hyperparameters, including preprocessing data size and model size on an SSL technique for a text classification task. Relations between these parameters are also examined. A limited number of epochs and smaller models are used, due to hardware limitations. A program for running experiments is written based on code from an earlier project [53].

1.1 Motivation

Investigating the impact of parameters on SSL performance for text classification results in several benefits. For example, an improved understanding of which parameters have the most impact on SSL for text classification benefits future projects by making it easier to make SSL work, in particular, if time-restricted. Another benefit is increased understanding of model size impact on SSL for text classification, to faster choose a model for future SSL projects. Increased understanding of parameter relations is an additional benefit. This can result in less training time for future SSL projects, in particular, if being time-constrained.

This thesis builds upon a previous project [53] which encompassed multiple experiments to answer some hypotheses related to SSL. The main conclusions of this project are:

- If a model is large enough, it is possible that SSL can improve text classification performance.
- Some preprocessing tasks for SSL improve performance for text classification compared to other tasks.

- Additional unlabeled and labeled data leads to less significant SSL performance gains for text classification.
- More extensive models improve text classification performance using SSL, compared to smaller models.

The written code for the previous project mentioned above was built upon for this thesis.

To explore the impact of model size on SSL for text classification, three models of different sizes are used for this thesis. Here, larger models are models containing more sophisticated architectures. A feed-forward model, a recurrent model, and a seq2seq model are used. Since 2017, the transformer [81] architecture has been popular in SSL research within NLP. All experiments for this thesis are executed using personal hardware, so there are hardware limitations involved. Therefore, for this thesis, smaller models are used.

1.2 Thesis Outline

In Chapter 2, Section 2.1 presents relevant background theory for Artificial Neural Networks (ANNs) (2.1.1), Recurrent Neural Networks (RNNs) (2.1.2), Gated Recurrent Unit (GRU) (2.1.3), Sequence to Sequence (Seq2seq) Models (2.1.4), Transfer Learning (2.1.5), Semi-Supervised Learning (2.1.6) and Self-Supervised Learning (2.1.7).

Also in Chapter 2, Section 2.2 explores previous relevant research for Transfer Learning Within NLP (2.2.1), Semi-Supervised Learning (2.2.2), Self-Supervised Learning (2.2.3) and Impact of Parameters on SSL Performance (2.2.4). Section 2.2.5 presents a Summary of Literature Review.

Chapter 3 first presents the Thesis Definition (3.1), which includes thesis goals and hypotheses. It then describes Contributions (3.2) and proposed solutions for achieving the thesis goals (3.1.1). Section 3.3 describes the Structure of Experiments. Section 3.4 describes the Models. Section 3.5 describes how experimentation is done. Section 3.6 describes how datasets are created. Section 3.7 briefly summarizes the Pretext Task used. Section 3.8 describes the Hyperparameters for experimentation.

In Chapter 4, the Sections 4.1, 4.2, 4.3 and 4.3.1 present the experiment results and discuss them for Experiment 1, 2, 3 and 4 respectively. Section 4.4 presents the Conclusion of Results, Hypotheses and Goals.

Chapter 5 presents the Conclusion and Future Work.

Chapter 2

Background

This chapter presents basic background theory (2.1) and explores previous relevant research (2.2).

2.1 Theory

AI systems are designed to get knowledge by extracting patterns from data. This ability is called machine learning. [34]

Machine learning includes several areas, one of which is deep learning. Within this area, neural networks are central – which SSL most frequently uses.

Inspired by the biological learning of brains, the first learning algorithms attempted to model how the brain works or could work. [34]

This section presents the basic theory on Artificial Neural Networks (ANNs) (2.1.1), Recurrent Neural Networks (RNNs) (2.1.2), Gated Recurrent Unit (GRU) (2.1.3), Sequence to sequence (seq2seq) models (2.1.4), Transfer Learning (2.1.5), Semi-Supervised Learning (2.1.6) and Self-supervised Learning (2.1.7).

2.1.1 Artificial Neural Networks (ANNs)

An artificial neuron can be observed as a mathematical function. In simple terms, it is a node that contains a particular value. Within a binary neuron, this value can either be one or zero. Usually, in machine learning, this value is a real number.

Artificial neurons are the fundamental building blocks of ANNs. Frequently, an artificial neuron takes inputs from other neurons, where inputs are multiplied with weights and summed together. The result is usually put through an activation function, which computes the final output of the neuron. This output is usually sent to other neurons.

Within deep learning, the nodes of an ANN are regularly split into layers. Within an ANN, there is often an input layer, one or more hidden layers, and an output layer. From a source, data is sent to the input layer. This data originates from multiple possible sources. These include for example text files, image files, or a microphone. The hidden layers process data from the input layer, and the output layer creates one or multiple outputs. Each output results from the overall function within the network.

If an ANN contains two or more hidden layers, it is known as a *Deep Neural Network (DNN).*

Within deep learning, the fundamental example is a deep feed-forward ANN, more precisely an under-category known as a *Multilayer Perceptron (MLP).*

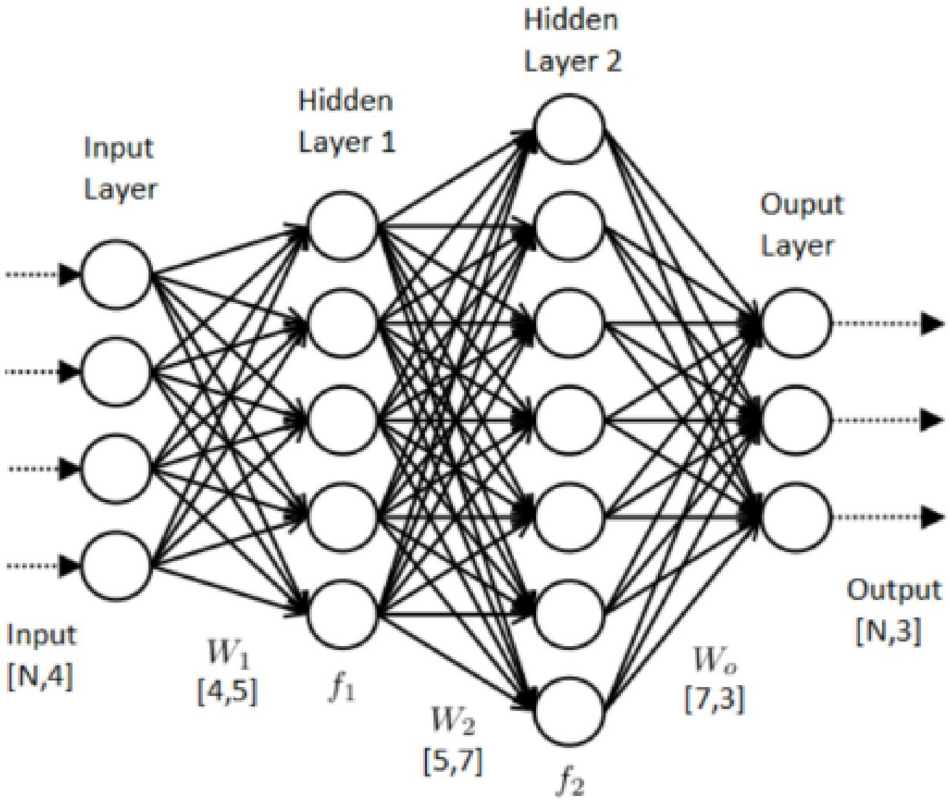

Figure 2.1: A deep MLP [55]

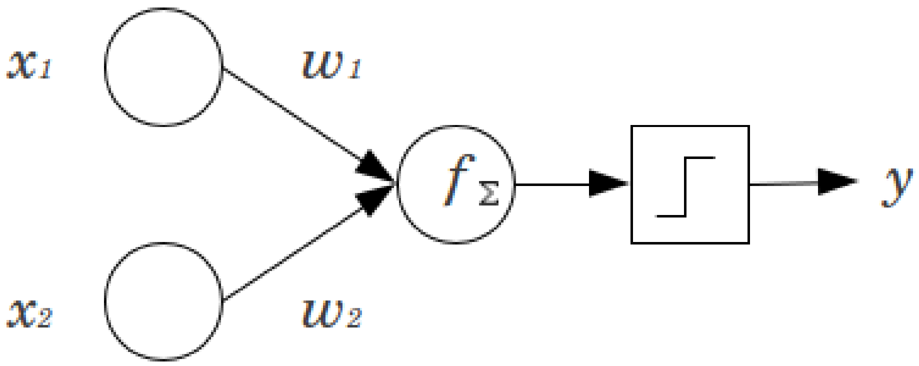

Figure 2.2: A single perceptron [69]

Within an MLP, the fundamental building block is the *Perceptron*. A perceptron detects features within input data, through calculations.

Figure 2.2 shows a single perceptron. Here, x_1 and x_2 are inputs to the perceptron. w_1 and w_2 are weights for each of the inputs. Each input is multiplied by its weight, and frequently the bias is added. Then the results are added together. The result is then put through the binary step activation function. If the input to this function is over a particular threshold, then the perceptron is activated. In the other case, the perceptron is deactivated, with the final output of zero. Equation 2.1 shows the calculation by the perceptron in Figure 2.2.

$$f(x) = \begin{cases} 1 \text{ if } \sum_{i=1}^{m} (w_i * x_i) + b > 0 \\ 0 \text{ otherwise} \end{cases} \tag{2.1}$$

There are also other activation functions, for example Rectified Linear Unit (ReLU), Sigmoid and Softmax. See Subsection 2.1.1.

Activation Functions

The choice of which activation functions to use within a neural network is important. Within hidden layers, the activation function decides how capable the model is adapting to training data. For the output layer, the activation function decides the category of predictions the network does.

Within a network, an activation function decides how the weighted sum from a neuron should be modified before being passed to the next layer. An activation function can also decide how the final output from the last layer is calculated.

A model can contain linear activation functions within the hidden layers. However, a model can learn better functions if it contains non-linear activation functions. A non-linear function estimates better a non-linear phenomenon compared to a linear function. This results in better performance during validation and testing.

Within hidden layers, popular activation functions include ReLU, TanH, and Sigmoid. For the output layer, popular activation functions include

Activation function	**Equation**	**Parameters**
ReLU	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ otherwise} \end{cases}$	x: Weighted sum of neuron inputs.
TanH	$f(x) = \frac{2}{1+e^{-2x}} - 1$	x: Weighted sum of neuron inputs. e: Exponential constant, which approximately equals 2.71828.
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	x: Weighted sum of neuron inputs. e: Exponential constant, which approximately equals 2.71828.
Softmax	$f(\overrightarrow{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ for $i = 1...K$	$\overrightarrow{x}$: Input vector, containing $x_0, ..., x_K$. x_i: Element of input vector. Is any real number. e: Exponential constant, which approximately equals 2.71828. e^{x_i}: The standard exponential function is used on all input vector elements. $\sum_{j=1}^{K} e^{x_j}$: Forces the outputs to sum to one. Every output is between zero and one, creating a probability distribution. The K exponential terms sum together here. K: Amount of possible classes.
Linear	$f(x) = x$	x: Weighted sum of neuron inputs.

Table 2.1: Activation functions

Softmax, Sigmoid, and Linear. See Table 2.1. For Multiclass Classification, Softmax should be used for the output layer. With Softmax, there is one neuron for each possible class.

Optimization

For a training procedure, it is essential to update model parameters for improving prediction results. An optimizer is a technique for updating, for example, model parameters and learning rate for lowering loss. It links the model weights and the loss function by adjusting weights based on outputs from the loss function. The optimizer improves the model by updating the model weights. The loss function guides the optimizer in the right direction for improving the model results.

Gradient descent is an optimizer. The first order derivate of the loss function is used for calculating how each weight should be updated, for lowering loss. Based on losses, a loss is *backpropagated* layer by layer. Then, each weight is updated for lowering the loss. After computing the error gradient from the total dataset, the model weights are updated. If the training dataset is massive, then it takes a significant amount of time before minimizing the loss.

Stochastic Gradient Descent (SGD) is an alternative version of gradient descent. Instead of updating weights from the total dataset, weights are updated after computing loss for each sample. The weights are then updated more often. *Stochastic* refers to extracting a random sample from the dataset in every iteration. SGD updates weights based on gradient calculated from one sample in each iteration. Therefore, it is significantly more effective than gradient descent. Updating weights usually results in effective minimizing of the loss function, but the minimizing process is more unstable.

Another version is *Mini-Batch SGD*. With this optimizer, the training data is split into batches. The weights are updated after calculating the gradient from a single batch. With this optimizer, the weights are updated frequently, and the minimizing process is more stable compared to SGD.

Other optimizers include for example *Adagrad*, *AdaDelta* and *Adam*.

Loss Functions

The objective function evaluates a prediction with the ground truth. Frequently, the goal of training a model is to minimize the error. Therefore, the objective function is known as the loss function. The computed value is known as *loss*.

A loss function is selected for the optimization process for computing the prediction error. There exist several popular loss functions. Cross-Entropy Loss is an example of one of these functions. From the predicted probability of a class, Cross-Entropy Loss calculates a score. This score reflects the distance between predicted class probability and ground truth, which is zero or one. This score is logarithmic, meaning that larger differences are penalized significantly more than smaller differences. For Cross-Entropy Loss, a smaller loss means a better model compared to a higher loss. Cross-Entropy Loss can be used for multi-class prediction tasks.

Hyperparameters

Before the training process, hyperparameters are decided. There are multiple different hyperparameters. Examples include learning rate, number of hidden layers and nodes, batch size, dropout rate and number of epochs. Some hyperparameters determine final model architecture, other hyperparameters decide how the model is optimized.

2.1.2 Recurrent Neural Networks (RNNs)

Within a feed-forward model, data flows in one direction. Here, data flows from the input layer through the hidden layer(s) to the output layer. Recurrent Neural Networks (RNNs) work differently by sharing weights through time. In contrast to a standard feed-forward network, an RNN can remember prior inputs. It uses older knowledge for the current computation. For some tasks within NLP, for example, data usually needs sequential processing. For these tasks, RNNs are effective due to having useful attributes.

An RNN contains an inner memory. Output for the current input results not only from the current input but additionally from the previous output.

The calculated output is used for the next output calculation.

RNNs are often effective for sequential data processing. RNNs use their inner memory to process sequential data, learning relations between inputs. Within feed-forward networks, inputs are independent of each other.

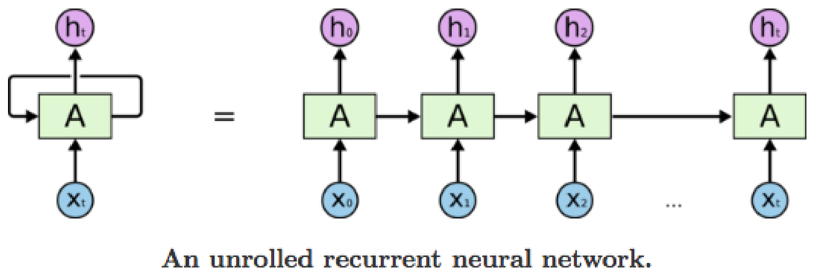

Figure 2.3: An RNN [62]

In Figure 2.3, x_0 is put into the network. Then, the network outputs h_0. The next output h_1 results from both the new input x_1 and h_0 from the previous step. The next output results from x_2 and h_1, and so on. This way, an RNN uses current context while learning. Therefore, it is effective for sequential data processing.

RNNs have both positive and negative attributes. Negative attributes include the following: An RNN does not easily learn effectively. With TanH or ReLU as an activation function, it struggles to learn with larger sequences. Another negative attribute is the parallelization challenge, because of the sequential processing.

While a deep network learns, gradients are backpropagated through many layers to early layers. The chain rule results in many gradients to multiply for computing gradients, in particular for the early layers. The number of layers to backpropagate through decides the number of gradients to multiply for computing gradient. If multiplying a significant number of high gradients, then the computed gradient increases fast. After a while, the gradient can explode and render the model useless. This is the problem of exploding gradient. If multiplying a significant number of small gradients, then the computed gradient decreases fast. In this case, the model has problems with learning. This is the problem of vanishing gradient.

An RNN also struggles with vanishing and exploding gradients. Within sequence data, there are regularly long-term dependencies. For example, an extensive sequence starting with "*I am from Norway*" and ending with "*I speak Norwegian*", contains a long-term dependency. Prediction of the word "*Norwegian*" depends on the word "*Norway*" from earlier in the sequence. If the distance between words is significant, then there is a long-term dependency. This is a dependency between long-distance sequence data. As dependency distance increases, an RNN struggles to learn this dependency. This is due to the problem of vanishing or exploding gradient.

Bidirectional RNNs

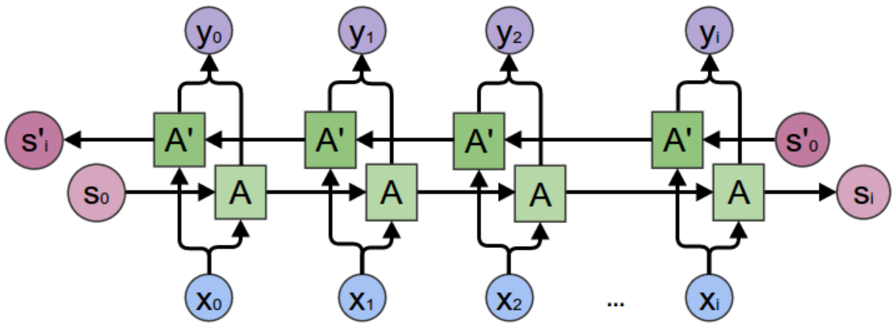

Figure 2.4: A bidirectional RNN [24]

As observed in Figure 2.4, the central concept of bidirectional RNNs is combining two independent RNNs. Sequential data is used in the standard time sequence for the first RNN, and in backward time sequence for the other. For each time step, the output is often the concatenation of the two outputs from each RNN. The two outputs can alternatively be summed together, for example. Therefore, a bidirectional RNN considers context from each direction for each time step. This frequently improves performance.

2.1.3 Gated Recurrent Unit (GRU)

The architecture Gated Recurrent Unit (GRU), was presented by Cho et al. [22] in 2014. It fixes the vanishing gradient problem encountered by traditional RNNs. Because of architectural similarities, GRU is known as an alternative version of the well-known *Long Short Term Memory (LSTM)* architecture. LSTM was first presented in 1997, by Hochreiter and Schmidhuber [40].

For fixing the vanishing gradient problem, GRU uses two vectors known as gates. These are the *Reset gate* and the *Update gate*. They decide the data which should be sent to the output. These gates are trained to avoid forgetting useful knowledge from earlier time steps.

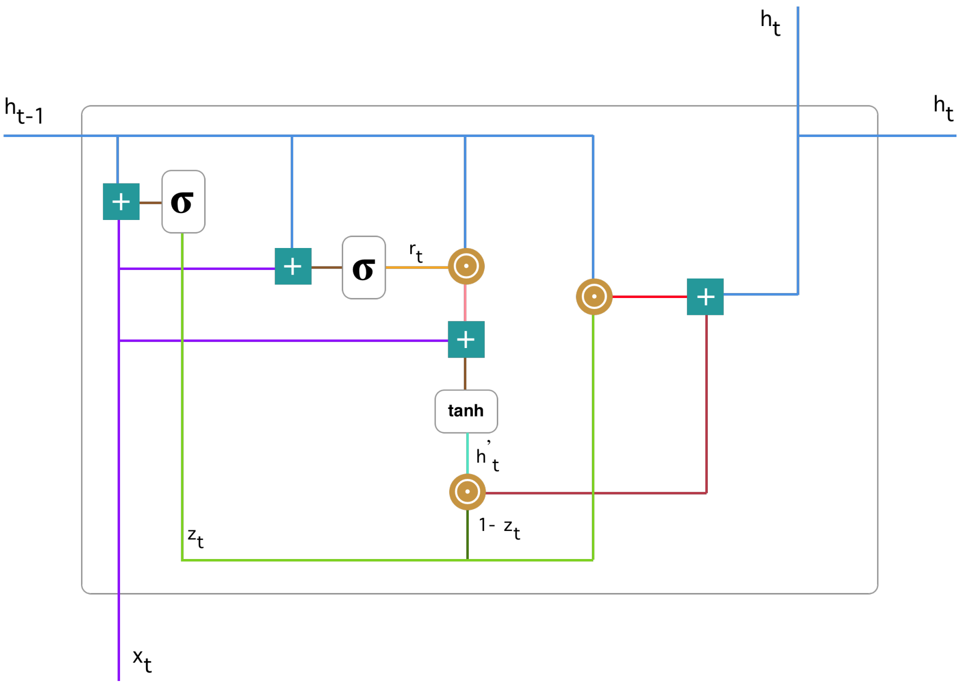

Figure 2.5: A single GRU [47]

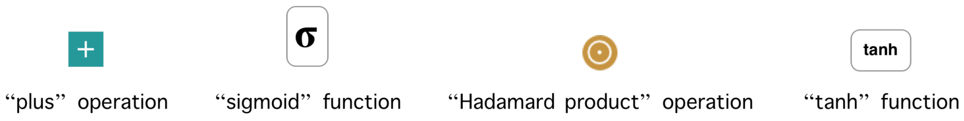

Figure 2.6: GRU notations for Figure 2.5 [47]

$$z_t = \sigma(W(z)x_t + U(z)h_{t-1}) \tag{2.2}$$

Equation 2.2 computes the update gate z_t for time step t. The input x_t and the previous hidden state h_{t-1} representing previous data are each multiplied by their weights $W(z)$ and $U(z)$ respectively. The results are added together, and put through a Sigmoid activation function. The output is between zero and one. See Figure 2.7 below.

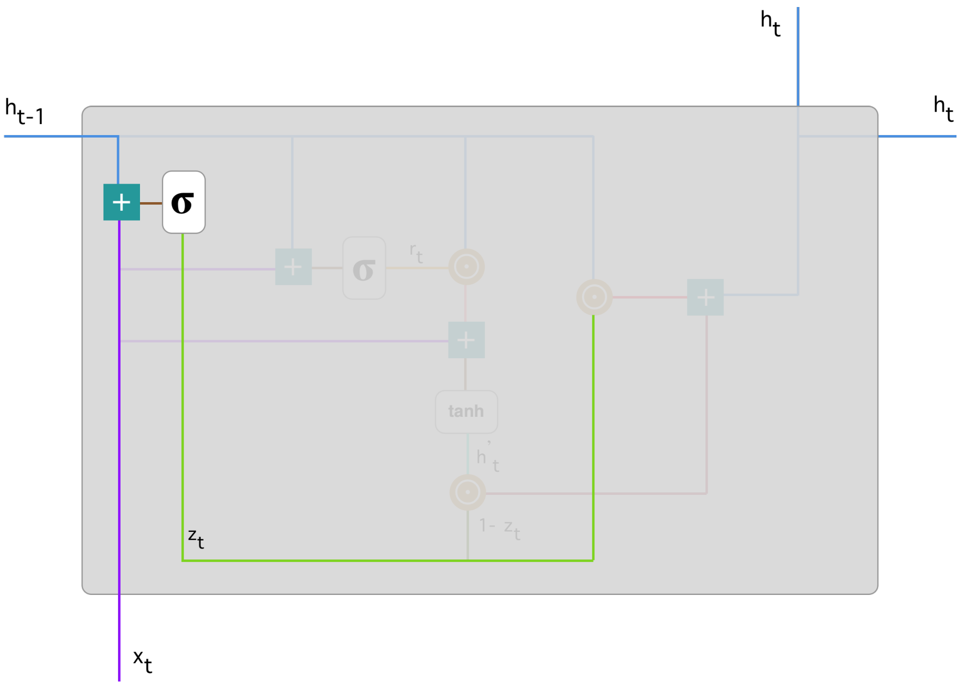

Figure 2.7: Update gate within Figure 2.5 [47]

$$r_t = \sigma(W(r)x_t + U(r)h_{t-1}) \tag{2.3}$$

Equation 2.3 computes the reset gate, which helps the model decide what knowledge from previous time steps is not useful and can be ignored. Although the reset gate has another purpose than the update gate, their equations are similar. The difference is the weights used. See Figure 2.8 below.

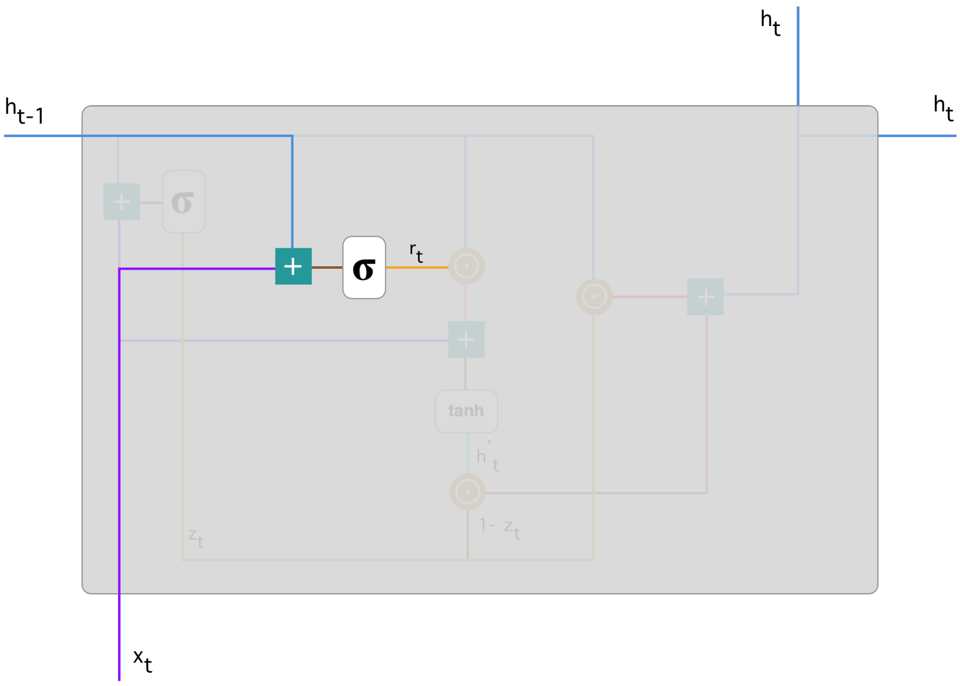

Figure 2.8: Reset gate within Figure 2.5 [47]

$$h_t' = \tanh\left(Wx_t + r_t \odot Uh_{t-1}\right) \tag{2.4}$$

Equation 2.4 computes the current memory for saving useful previous data, using the reset gate. x_t and h_{t-1} are multiplied by their weights W and U, respectively. The element-wise product of the reset gate r_t and Uh_{t-1} computes what data to remove from prior time steps. The result is then added to Wx_t. This result is then put through the non-linear tanh activation function. See Figure 2.9.

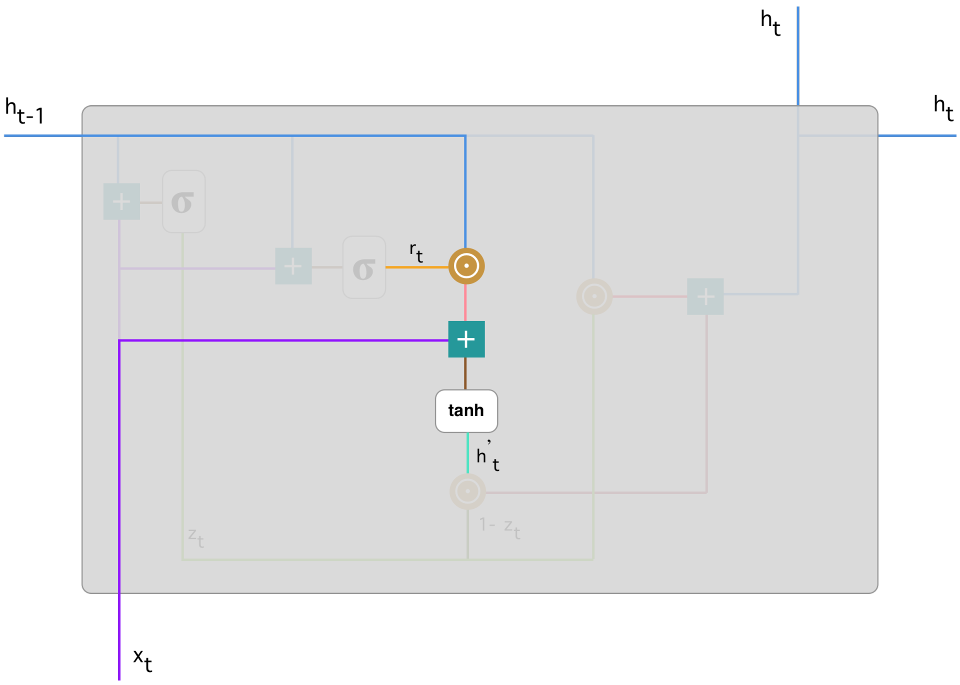

Figure 2.9: Calculating current memory content within Figure 2.5 [47]

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t \tag{2.5}$$

The final memory h_t for the time step to be sent to the next step must be computed. Equation 2.5 calculates this. The update gate z_t is used. It decides what data to keep from current memory h'_t and from prior steps h_{t-1}. Element-wise multiplication is done between the update gate z_t and h_{t-1}, and also between $(1 - z_t)$ and h'_t. The results are then added together. See Figure 2.10.

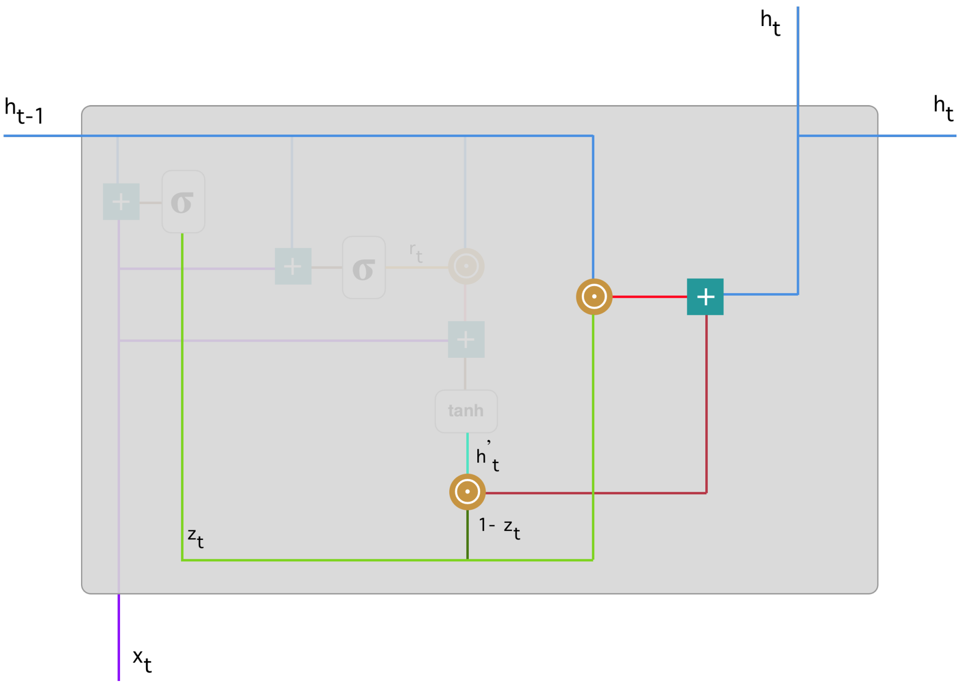

Figure 2.10: Calculating final memory within Figure 2.5 [47]

For fixing the vanishing gradient problem, the update gate is useful. The update gate helps the model to decide what knowledge from earlier time steps should be sent into the next time steps. The model can potentially transfer all knowledge from previous time steps into the next steps. This prevents the vanishing gradient problem.

2.1.4 Sequence to Sequence (Seq2seq) Models

Sutskever et al. [77] presented the seq2seq model in 2014. The seq2seq model outputs a particular sized output, from a particular sized input. The input and output lengths are not always equal. For example, a seq2seq model could translate a three-word sequence to another five-word sequence. A single GRU model or LSTM model could not do this. Therefore, a seq2seq model makes it possible to have variable length inputs and outputs, which is impossible with a single RNN.

Seq2seq models have multiple use cases. These include for example machine translation, speech recognition, and video captioning. The seq2seq model is useful for sequence-based tasks, in particular, if the input and output lengths are variable.

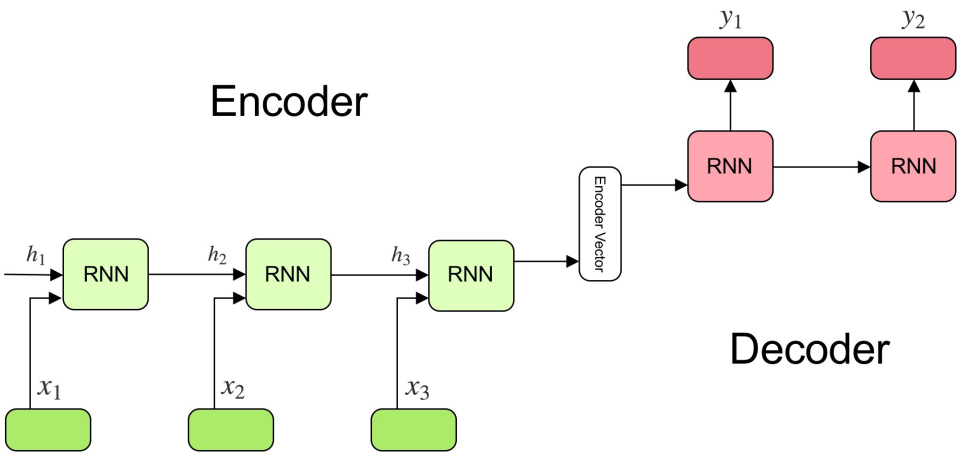

Figure 2.11: Seq2seq model [48]

As seen in Figure 2.11, the seq2seq model contains three main parts. These are the encoder, the encoder vector, and the decoder.

Encoder

An encoder contains multiple connected recurrent cells, frequently LSTM or GRU cells for improved performance. Each recurrent cell uses an element from the input sequence, calculates, and transfers output to the next time step.

Encoder Vector

The encoder vector is the final hidden vector calculated by the encoder. It is the first hidden vector within the decoder, to support the decoder in making the best possible predictions. The encoder vector contains data, which represents the whole input sequence from the encoder.

Decoder

The decoder contains multiple connected recurrent cells, usually GRU or LSTM cells. This is similar to the encoder. Each recurrent cell uses the previous hidden state, outputs an output, and its hidden state. This is transferred to the next time step.

Attention

With extensive sequences, the seq2seq model struggles to represent the sequence within the encoder vector. After processing a large sequence, the encoder regularly has forgotten useful knowledge from the start of the sequence. The attention technique addresses this challenge of long-term dependencies.

In 2014, Bahdanau et al [6] presented the attention technique. They showed fundamentals which resulted in the known paper by Vaswani et al. [81] in 2017. Vaswani et al. presented transformers and the idea of computing inputs in parallel rather than sequence. Parallel calculations lowers training time, for example.

For the attention technique, during output prediction by the decoder, certain parts of the input sequence in the encoder are particularly important. In simple terms, this means paying attention to the most important inputs during output prediction.

The attention technique links the encoder and decoder. With it, the decoder has access to hidden encoder vectors. This way, the model pays attention to specific inputs during output predictions. Through training, the model learns connections between inputs. The network uses the attention technique to process more extensive sequences with better performance.

Figure 2.12 below shows that the number of hidden states sent from the encoder is equal to the number of elements within the input sequence. In this figure, HS1, HS2, and HS3 are hidden states from the encoder. The decoder uses these for predictions.

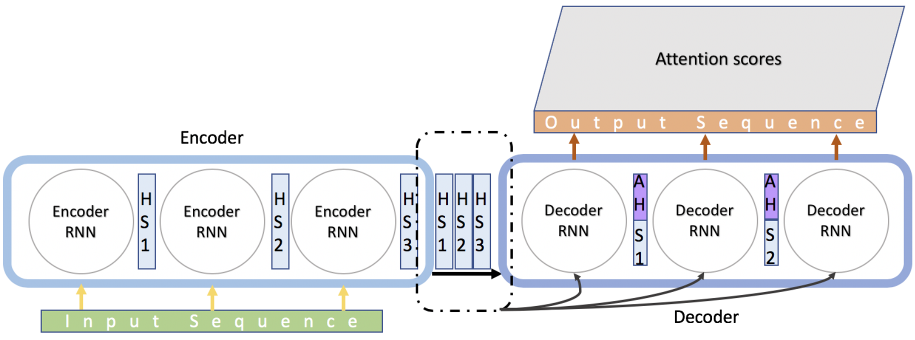

Figure 2.12: Seq2seq architecture with attention technique [30]

2.1.5 Transfer Learning

Within artificial intelligence, transfer learning is an important technique for multiple reasons. These reasons include using less training time, usually improving model results, and lowering required data size. Due to training an extensive model from scratch with a comprehensive task frequently takes a significant time amount, transfer learning lowers training time. It also usually improves results using less labeled data due to the network has pre-trained earlier.

Using transfer learning, a model created for one task is reused as the initial model on another task. Previous understanding from one task and domain is transferred to another task and domain.

Using pre-trained models, transfer learning breakthroughs have been achieved within computer vision. For deep networks, this has improved training time and performance. Substantial datasets, ImageNet for example, have resulted in the latest pre-trained models for transfer learning. NLP did not have its ImageNet counterpart until a few years ago.

Pre-trained language models are important for transfer learning within NLP. The concept here is reusing pre-trained models trained within resource-rich domains and languages. These models are often fine-tuned using limited resources for downstream tasks. Examples of pre-trained language models are BERT [29], RoBERTa [54] and GPT-3 [13]. Examples of downstream tasks are text classification, named-entity recognition, and part-of-speech tagging. In NLP, pretraining happens on unlabeled data and is, therefore, an example of SSL.

2.1.6 Semi-Supervised Learning

The field of machine learning frequently draws a line between *supervised learning* and *unsupervised learning* [11]. Supervised learning uses a dataset containing x-samples with target y-samples. This learning type aims to create a model capable of predicting target labels for previously unseen x-samples. In contrast, unsupervised learning uses a dataset containing x-samples but not target y-samples. This learning type aims to learn useful patterns from unlabeled data.

Semi-supervised learning uses both supervised and unsupervised learning [20] [86]. Usually, semi-supervised learning aims to increase the performance of either the supervised or unsupervised approach. It achieves this by using knowledge from the other learning approach. For example, for text classification, often there is only a small amount of labeled data available. Adding unlabeled data can help performance. The model can usually improve classification performance, by using extracted knowledge from both unlabeled and labeled data.

In particular, if labeled data is challenging to extract, using unlabeled data also can be significantly useful. For example, for classifying medical images, intentions, or doing part-of-speech tagging, obtaining labeled data is regularly costly and time-consuming. Therefore, using SSL is commonly cheaper and saves time for improving classification results, compared to obtaining additional labeled data.

Research within both SSL and machine learning frequently uses the classification task, which is one of the most common tasks for machine learning. SSL has also been utilized in situations where there has been a substantial volume of labeled data. If an unlabeled dataset contains additional knowledge useful for the classification task, then the unlabeled data can improve classification performance.

According to Chapelle et al. [20], for the successful use of SSL, three *assumptions* need to be satisfied.

- **The smoothness assumption**: If a pair of data samples x_1 and x_2 are close within a high-density area, also the outputs y_1 and y_2 should be close.

- **The cluster assumption**: Data samples are probably within the same class, if they are within the same cluster.

- **The manifold assumption**: "The (high-dimensional) data lie (roughly) on a low-dimensional manifold" [20, p. 6].

2.1.7 Self-Supervised Learning

Self-supervised learning is a category of unsupervised learning, where a neural network learns using labels automatically extracted from the data itself. No human labeling is involved. [31] [39]

Massive amounts of unlabeled data can be used for model training with self-supervised learning, making this technique important.

Transfer learning and self-supervised learning are related research fields. Together, they use one task with a significant volume of unlabeled data and transfers understanding to another task. Across domains, these techniques have increased in scope because of their scalability and furthering of state-of-the-art performance.

2.2 Literature review

The research within relevant fields advances fast. This section explores previous research for Transfer Learning Within NLP (2.2.1), Semi-Supervised Learning (2.2.2), Self-Supervised Learning (2.2.3) and Impact of Parameters on SSL Performance (2.2.4). Finally, a Summary of Literature Review (2.2.5) is given.

2.2.1 Transfer Learning Within NLP

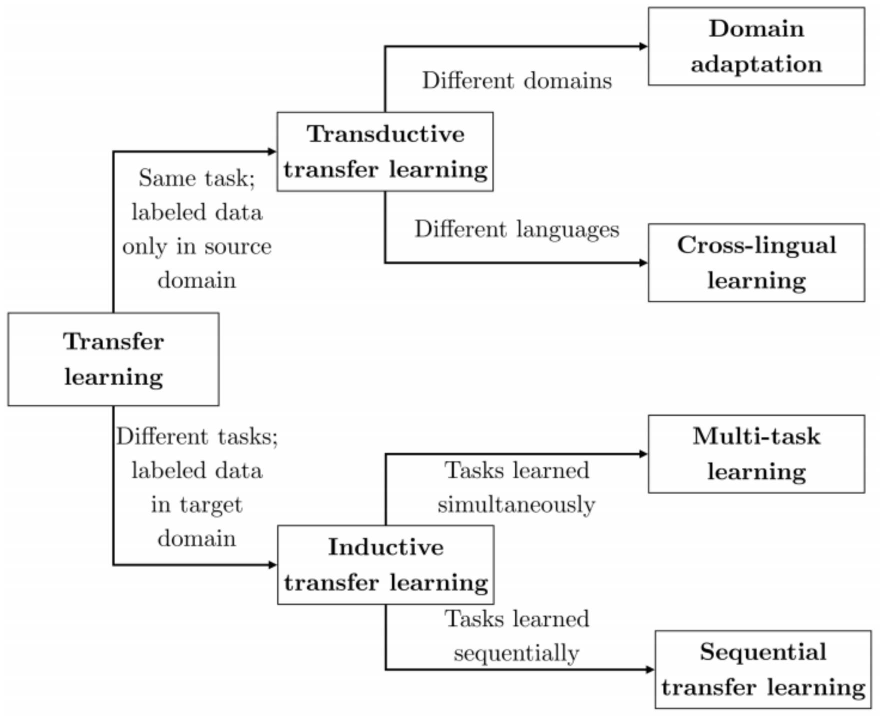
Domain adaptation
Different domains
Transductive transfer learning
Same task; labeled data only in source domain
Different languages
Cross-lingual learning
Transfer learning
Different tasks; labeled data in target domain
Multi-task learning
Tasks learned simultaneously
Inductive transfer learning
Tasks learned sequentially
Sequential transfer learning

Figure 2.13: Overview of transfer learning within NLP [73]

Ruder [73] divides transfer learning into inductive and transductive transfer learning. He claims that within transductive transfer learning, original and target tasks are equal. He says that within inductive transfer learning, original and target tasks are not equal. He divides inductive transfer learning into a pair of sub-types, called multitask learning [15] and sequential transfer learning.

Sequential Transfer Learning

Language models (2.2.1) often use sequential transfer learning. Sequential transfer learning [74] uses the concept of *pre-training then fine-tuning*. With neural networks, the initial phase is known as *pre-training* and the second phase of training is known as *fine-tuning* [56].

Sequential transfer learning with fine-tuning described by Ruder [73] and pre-training described by Engelen and Hoos [78] are descriptions of the same concept.

Ruder [73] mentions different pre-training types for sequential transfer learning. These are supervised, distantly supervised, unsupervised, and multi-task pre-training. Architectures are also included. This thesis focuses on both unsupervised and multi-task pre-training.

A pre-training task included in either multi-task learning or single-task learning should extract useful features. A useful pre-training task enables the use of significant amounts of additional data for training a neural network.

Sequential transfer learning is important within NLP. According to [73], sequential transfer learning is valuable within three settings in particular:

- Not possible to use data for tasks simultaneously.
- Have a large amount of data for one task but not for the other.
- It is important to fine-tune to multiple downstream tasks.

In summary, sequential transfer learning is useful, for example when having a substantial unlabeled volume of data and a small labeled volume of data. Including unlabeled data for training can improve results. Fine-tuning a pre-trained model for multiple downstream tasks is also useful, due to usually improving results. Only doing pre-training once is also beneficial, due to saving training time. There are different pre-training types, unsupervised and multi-task pre-training for example. A pre-training task is useful if it extracts relevant knowledge.

Unsupervised Pre-Training

Unsupervised pre-training is significantly more scalable and more related to how humans learn, compared to supervised pre-training. Unsupervised pre-training does not demand labeled data. [14] [68].

Due to the massive volume of available unlabeled data available on the internet, unsupervised pre-training within NLP is useful [72]. Commonly it is possible to improve results by training a larger model on a more substantial data set because neural models have significant scalability [46] [38].

In summary, unsupervised pre-training has significant scalability due to not requiring labeled data. This scalability combined with neural network scalability is usually a good combination for improving results.

Multi-Task Learning

Caruana [15] claims the following about Multi-task Learning (MTL): "MTL improves generalization by leveraging the domain-specific information contained in the training signals of *related* tasks". He further points out that it accomplishes this by training tasks side by side and utilizing a shared representation. He says that multi-task learning is frequently accomplished with hard or soft parameter sharing using hidden layers within deep learning.

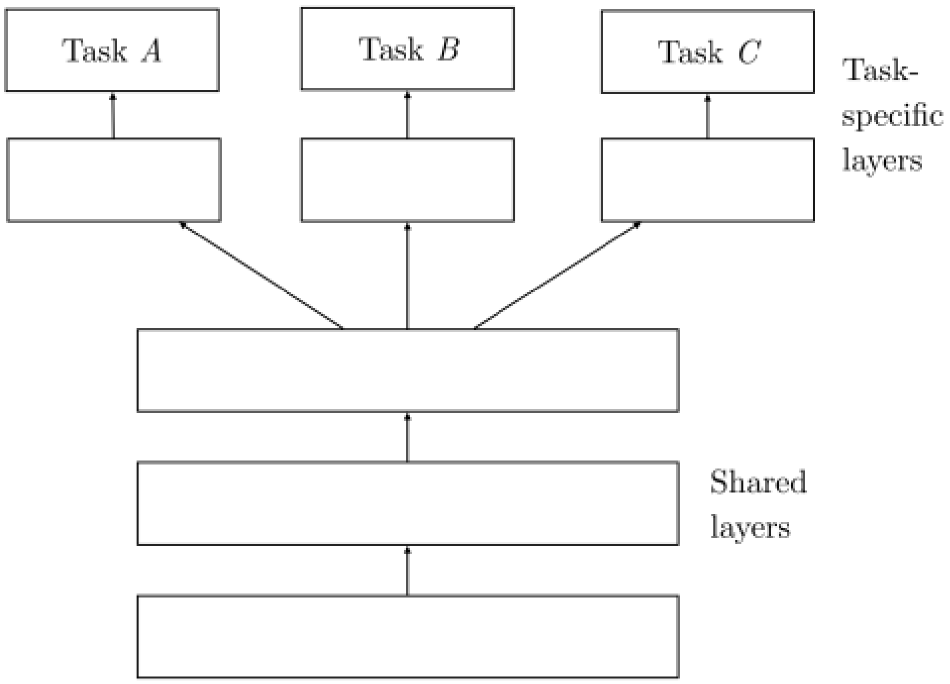

Figure 2.14: Hard parameter sharing [73]

Originating from Caruana [16], hard parameter sharing is the most popular technique to accomplish multi-task learning using neural networks. As observed in Figure 2.14, it is commonly done by sharing hidden layers between tasks. There are multiple output layers for different tasks. According to Ruder [73], every task has its network and parameters with soft parameter sharing. He further claims that a neural network learns to solve multiple tasks simultaneously, with multi-task learning.

pre-training on multiple tasks with multi-task pre-training can improve representations [7]. For example, both masked language modeling and next-sentence prediction is done by Devlin et al. [29] with BERT.

In summary, multi-task learning can improve performance by training tasks in parallel with a shared representation. Neural networks often use hard parameter sharing, but soft parameter sharing is also possible. Multi-task pre-training can improve performance.

Word Embeddings

For transfer learning within NLP, word embeddings are important. Word embeddings use the idea that words can be represented by vectors. Frequently, words with similar meanings are present within a similar context. Similar words are represented by similar vectors. [72]

Multiple techniques have been developed to create word embedding vectors. This section gives an overview of popular techniques, but does not go into detail on how these work.

Based on the co-occurrence of words in a document setting, the classic technique *Latent Semantic Analysis (LSA)* [27] creates low dimensional distributed word representations. LSA enhances information retrieval compared to earlier techniques. It presents the important concept of reducing the dimensionality in the information retrieval challenge.

One-hot encoding is an example of a simpler technique for representing words as vectors. One-hot encoding represents each word with a vector, where all features are zero except one. This exception represents the word index in the vocabulary. One-hot encoding has multiple disadvantages. These include the similarity problem, where similar words are not containing similar features. Another problem is that while vocabulary size increases, the number of features grows. Another one is the computational problem because most machine learning techniques are not efficient with many and sparse features. Word embeddings solve the important problem of generalization. For example the words "*car*" and "*truck*" are often in similar contexts. These words, therefore, have similar word embeddings. If a model sees the word "*truck*", it understands this word is similar to "*car*" due to containing similar features. Therefore, the model processes "*truck*" and "*car*" similarly. Instead of learning how to represent "*truck*" from scratch, it notices the embedding is similar to "*car*" and can be processed similarly. This results in a significantly more generalized model, compared to using one-hot encoding. Word embeddings improve results for almost all NLP tasks, in particular, if not utilizing a substantial training dataset.

Word2vec [61], released in 2013, is a toolkit used for simple training and utilization of pre-trained embeddings. It led to the popularity of word embeddings. Word2vec presents two model architectures for learning distributed word representations with large corpora: *CBOW (Continous Bag-*

of-Words) and *Skip-gram model.* It aims to keep down computational demands. Word2vec outperformed earlier architectures for creating word vectors, taking into account syntactic and semantic similarities. Word2vec has multiple improvements compared to LSA. These include: Word2vec tracks word meanings, context knowledge is kept track of, and embedding vector length is small. In a Word2vec embedding vector, every dimension carries knowledge concerning a word feature. Large sparse vectors are avoided here, in contrast to with LSA.

Glove [66], presented by Pennington et al. in 2014, is a technique for creating word vectors. Glove is short for *Global Vectors*. Glove is a global log-bilinear regression model for unsupervised learning of word vectors. It improved previous models on named entity recognition, word similarity, and word analogy tasks. Glove not only uses local word context data but also global statistics to create word representations. This is in contrast to word2vec. Glove uses count information and linear substructures common in log-bilinear prediction-based techniques, word2vec for example.

In summary, word embeddings result in more generalized models, compared to one-hot encoding, for example. Word embeddings use the idea of representing words as vectors, where similar words have similar vectors. The classic technique *Latent Semantic Analysis* [27] creates low dimensional distributed word representations. Word2vec [61] trains and utilizes pre-trained embeddings, and led to the popularity of word embeddings. Glove [66] uses both local word context data and global statistics to create word representations. It improved earlier models on different tasks.

Language Models

Language modeling (LM) creates probabilistic models that predict the next word based on previous words. An N-gram model predicts the next word, based on the earlier $N - 1$ words.

Bengio et al [9] presented in 2001 the initial neural language model, a feedforward model. Inputting word embeddings to a neural model, Neural Language Models (NLM) solve the n-gram data sparsity problem.

McCann et al. were impressed by the effective transfer learning use within computer vision. In 2017, they presented the CoVe [58] paper. CoVe aims to transport understanding from machine translation to various downstream tasks. To contextualize word embeddings, CoVe uses a deep LSTM encoder from an attentional seq2seq network trained for translation.

Peters et al. presented the ELMo (Embeddings from Language Models) [67] paper in 2018. ELMo solves the main restriction of CoVe, that pre-training is limited by accessible datasets for the supervised translation task. Using unsupervised pre-training, ELMo addresses this problem.

Howard and Ruder presented ULMFiT [42] in 2018. ULMFiT addresses the problem of previous NLP techniques, demanding changes for each task and learning from scratch. In computer vision inductive transfer learning had been effectively used. ULMFiT presents a successful transfer learning technique that can be used with all NLP tasks. For fine-tuning language models, ULMFit presents important methods.

Radford et al. at OpenAI presented GPT (Generative Pre-training Transformer) [70], enlarging the unsupervised language model to a significantly larger scale. GPT uses a massive quantity of text corpora, with the concept from ELMo. Generative pre-training for GPT may input the most extensive possible text amount during the initial phase. During the second phase, GPT uses as few new parameters as possible to train and a small volume of labeled data. This way, the network is fine-tuned on different challenges. GPT learns to predict only the next left-to-right context. It is limited by this one-directional property. GPT contains a transformer [81] decoder architecture.

BERT (Bidirectional Encoder Representations from Transformers) [29] builds on the concepts of GPT. The bi-directional training of BERT is the most important advancement and distinction from GPT. BERT uses a bi-directional multi-layer transformer encoder.

ALBERT (A Lite BERT) [49], released in 2019, is a smaller version of BERT while keeping results. It trains roughly 1.7 times quicker using 18 times less parameters compared to BERT.

GPT-2 [71] is a descendant of GPT. It contains 10 times the quantity of parameters compared to GPT: 1.5 billion. GPT-2 demonstrates that using additional parameters and training with more data improves model performance. It improves results for multiple challenges within zero shot learning compared to earlier models.

Liu et al. presented RoBERTa (Robustly optimized BERT approach) [54], demonstrating that BERT [29] could improve or get the same results of all earlier models. Therefore, RoBERTa shows that BERT was largely under-trained. It presents how to train BERT to improve performance, through some changes. For example, hyperparameters were discovered to significantly affect model results. Changes are the following:

1. Doing model training for more time, with larger batch size and a more substantial amount of data.

2. Removing the next sentence prediction task.

3. Using extended sequences during training.

4. For training data, modifying masking pattern dynamically.

T5 (Text-to-Text Transfer Transformer) [72] is a newer language model. Using the transformer architecture, it contains an encoder-decoder structure. With a variety of tasks, the text-to-text setting results in a simpler assessment of transfer learning using the same network.

GPT-3 [13], presented in 2020, contains about 100 times the quantity of parameters compared to GPT-2: 175 billion. GPT-3 and GPT-2 use the same architecture. Inspired by the *Sparse Transformer* [21], GPT-3 utilizes alternating dense and locally banded sparse attention patterns within transformer layers. Compared to fine-tuned BERT networks, GPT-3 reaches significant results with different NLP tasks.

At the beginning of 2021, Google presented their paper on Switch Transformers [32]. While keeping the same amount of floating-point calculations per second, switch transformers present a new technique for significantly increasing parameter quantity. Networks usually reuse the same parameters for every input, within deep learning. For every sample, Mixture of Experts (MoE) networks counter this by using varying parameters. Using a massive parameter amount, the achievement is a sparsely activated network with a static calculation cost. Switch transformers are intuitive and enhanced models with less calculation and communication costs and make simpler the MoE routing technique. They reach a factor of four-time improvement using the T5-XXL network. Switch transformers further the latest scale of language models by pre-training networks containing up to a trillion parameters using the *Colossal Clean Crawled Corpus (C4)*. C4 was initially presented in the T5 [72] paper.

In a paper released in 2021 by Bender et al. [8], they point out the following: With regards to the volume of training data and parameters, a significant pattern within NLP has been the growing scale of language models. This pattern is observed when looking at the previous work above. Related to the trend of enlarging language models, Bender et al. [8] present different risks and costs. These include for example environmental and financial costs. The authors hope these points motivate NLP scientists to focus work and resources on methods for doing NLP tasks, that are efficient and not significantly data demanding.

Neural language models achieve enhanced performance compared to statistical language models. A key reason for this is that neural language models generalize better, mainly due to using word embeddings. The generalization of representations provided by word embeddings is not easily replicated in statistical language models. See information on word embeddings above 2.2.1.

In summary, a language model aims to forecast the likelihood of a word sequence. An N-gram model predicts the next word, based on prior words. Using word embeddings, Neural Language Models (NLM) solve the n-gram data sparsity problem. Bengio et al [9] present the initial NLM. CoVe [58] transfers translation knowledge to downstream tasks. ELMo [67] solves the restriction of CoVe, that pre-training is limited by accessible datasets, using unsupervised pre-training. ULMFiT [42] presents a transfer learning technique for NLP tasks. GPT [70] enlarges the language model to a significantly more substantial size, using a massive volume of unlabeled data. Bi-directional training of BERT [29] is a significant improvement compared to GPT. ALBERT [49] is a smaller BERT version. GPT-2 [71] uses 10 times the parameter amount from GPT. RoBERTa [54] shows how to modify BERT training to improve results, for example changing hyperparameters. T5 [72] uses a text-to-text setting, which results in a simpler transfer learning assessment with a single network for multiple tasks. GPT-3 [13] uses 10 times the parameter quantity from GPT-2. Switch Transformers [32] use a new technique for significantly increasing parameter amount while keeping the same number of calculations per second. Bender et al. [8] show risks and costs, related to increasing language models. Neural language models achieve improved results compared to statistical language models, mainly due to generalizing better with word embeddings.

2.2.2 Semi-Supervised Learning

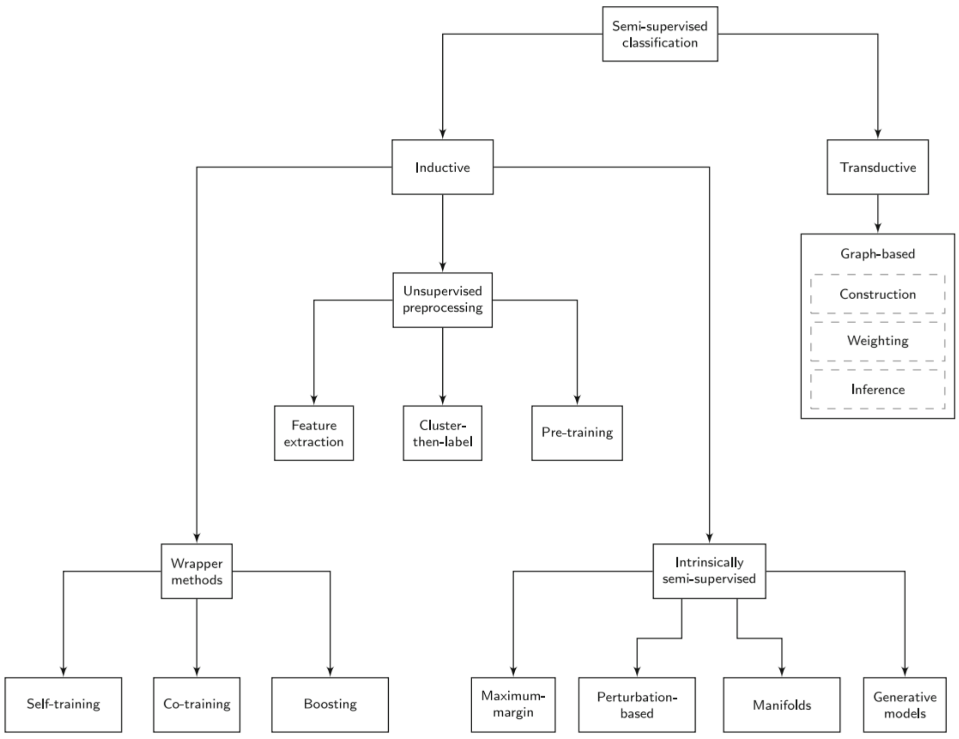

Figure 2.15: Overview of the taxonomy of semi-supervised classification. Every leaf represents a technique for including unlabelled data into classification tasks. [78]

Pre-Training

Multiple semi-supervised classification techniques have been presented, during the last two decades. See Figure 2.15 above.

Unsupervised preprocessing, an inductive technique in Figure 2.15, utilizes labeled and unlabelled data within two phases. This is in contrast to wrapper methods and intrinsically semi-supervised methods.

Before supervised learning, unlabelled data moves the decision border towards potentially more relevant areas with pre-training methods. Deep belief networks and stacked autoencoders are popular techniques using this concept. Within deep learning, pre-training techniques have important origins. Deep neural architectures containing multiple hidden layers have increased in popularity, beginning near the start of the millennium. Parameters for unsupervised preprocessing are static after the unsupervised stage within feature extraction techniques. They can be modified during the supervised fine-tuning stage within pre-training techniques. See subsection 2.2.1 for history and the current state-of-the-art for language models, which often are using pre-training methods.

Unsupervised pre-training of a model commonly lowers the required volume of labeled data for downstream tasks to achieve good performance. Downstream NLP tasks are then simpler and cheaper to accomplish. Unsupervised pre-training also makes possible fine-tuning a model to multiple downstream tasks. This way, a pre-trained model is used for many downstream tasks, reducing training time and regularly improving downstream performance. Pre-training research is therefore valuable within NLP.

In summary, multiple semi-supervised classification techniques are used. Unsupervised preprocessing utilizes labeled and unlabelled data in two phases. Unlabelled data moves the decision border towards potentially more useful areas using pre-training, prior to supervised learning. Parameters for unsupervised preprocessing are static after the unsupervised phase in feature extraction techniques. They can be changed during the supervised fine-tuning phase while pre-training. Unsupervised pre-training usually lowers the demanded labeled data amount for downstream tasks to reach good performance, and makes it possible to fine-tune a model to many downstream tasks. Pre-training research in NLP is therefore useful.

History of Semi-Supervised Learning

Self-learning, additionally called self-labeling, self-training, and decision-directed learning, is likely the first concept utilizing unlabeled data within classification tasks. This is a wrapper method repeatedly utilizing a supervised learning technique. This concept has been known since the 1960s [75]

[33] [2].

Transductive inference, also known as transduction, with foundations laid by Vapnik [80] [79], is tightly connected to semi-supervised learning. Hartley and Rao [36] presented in 1968 an early use of transduction, but were not observing it as a concept directly. To increase probability of their model, they present a combinatorial optimization on labels of the test elements. During the 1970s, with the task of approximating the Fisher linear discriminant rule using unlabeled data, it appears SSL became significantly more used [41] [60] [65] [59]. Because of uses mainly within natural language challenges and text classification, during the 1990s popularity of SSL grew larger [83] [57] [12] [25] [45].

In summary, self-learning is probably the first concept utilizing unlabeled data for classification challenges. During the 1990s SSL utilization grew [83] [57] [12] [25] [45], due to uses mainly in natural language challenges and text classification.

2.2.3 Self-Supervised Learning

Self-supervised learning extracts features and understanding from unlabeled data, using a pre-training, also called *pretext* task [44]. After a neural network has trained on a pre-training task, the network can be adjusted to a target task using transfer learning.

Pretext tasks are varied. Often they include imputing or transforming input data, aiming to make the network predict absent data parts, or using a data bottleneck. GLUE [82] is a widely used benchmark within NLP. It is utilized to test training techniques within self-supervised learning, on different challenges. These include for example sentiment analysis, paraphrase identification, and natural language inference.

Within self-supervised learning, generative techniques create parts or total training data in the network output [44]. On an abstract level, discriminative techniques separate positive examples from negative examples [56].

Bottleneck-based techniques learn a restricted representation of information, frequently by training to recreate input information. An early bottleneck-based linear technique for dimensionality reduction is latent semantic analysis. Non-linear deep autoencoders [28] [39] extract more useful low-dimensional knowledge from data. There are compression-based, sparse, and variational autoencoders. There are also other techniques for doing bottleneck learning.

State-of-the-art techniques are commonly prediction-based because bottleneck techniques are frequently worse than prediction-based techniques [85]. Prediction-based techniques usually learn good information representations. This is done by learning a useful predictive task, to predict absent parts of input given its context for example. Techniques here are varied. It could for example be to predict absent parts within an image, absent words within a word sequence, or the future using present knowledge. Through predictions, the network learns relations between global and local information parts. Prediction-based techniques are used within both continuous and discrete domains. These domains often use separate techniques.

There are techniques for continuous domain challenges within self-supervised learning. These include for example speech and vision. Popular pretext tasks for continuous domains include spatial prediction, channel prediction, temporal prediction, order prediction, and hybrid approaches. [56]

For example NLP, within the discrete environment, also uses techniques for self-supervised learning. These techniques include word embeddings (see Subsection 2.2.1), contextual embeddings (see Subsection 2.2.1), language models (see Subsection 2.2.1), sequence to sequence pre-training (see T5 in Subsection 2.2.1) and discriminative pre-training tasks (see BERT in Subsection 2.2.1).

Natural language sees the text as orders of discrete symbols, also known as tokens. The effective use of self-supervised learning with language modeling challenges and their variations has been important for the significant growing popularity of transfer learning within NLP. [56]

In summary, self-supervised learning extracts knowledge using unlabeled data, with a pre-training, also called *pretext* task [44]. Frequently pretext tasks include imputing or transforming inputs, attempting to make the network predict missing data, or utilizing a data bottleneck. State-of-the-art techniques are often prediction-based [85]. Pretext tasks for continuous domains include spatial prediction, channel prediction, temporal prediction, order prediction, and hybrid approaches [56]. For example NLP within the discrete context, also uses self-supervised learning techniques. These include word embeddings, contextual embeddings, language models, sequence to sequence pre-training and discriminative pre-training tasks. Successful utilization of self-supervised learning with language modeling challenges and variations has been central for the popularity of transfer learning within NLP [56].

History of Self-Supervised Learning

Using no labels, initial research within self-supervised pre-training for deep neural architectures aims at successfully training deep belief networks [39] and stacked auto-encoders [10].

Instead of greedy layer-wise unsupervised learning, end-to-end training where a deep architecture is learning in a single operation has grown in popularity during the last 10 years. It has become possible to train significantly deep models [4], and avoiding local minima. This became possible due to improved activation functions [63], normalization [43] and architectural innovations [37]. Modern techniques are usually trained end-to-end, which is different from early research on greedy self-supervised learning [1] [28].

In the paper by Clark et al. [23], they pre-train a network by predicting if an input symbol was randomly replaced by a small BERT network. Training discriminator results in a good performance on downstream challenges, with significantly improved sample efficiency. This is because, in each iteration, all positions are trained.

In summary, initial research in self-supervised pre-training for deep models attempt to train deep belief networks [39] and stacked auto-encoders [10]. Instead of greedy layer-wise unsupervised learning, end-to-end training where a deep model learns in a single operation has grown in popularity. For example, in the paper by Clark et al. [23], they pre-train a network by predicting if an input symbol was randomly replaced by a small BERT network. This results in good downstream task performance, with improved sample efficiency.

2.2.4 Impact of Parameters on SSL Performance

If some unlabelled data contains information relevant for predicting labels that are not within labeled data or is challenging to extract, then the unlabelled data is beneficial. A technique needs the ability to extract this information for it to work using an SSL technique with a task. This leads to a question, for both scientists and users of SSL generally: *When does this work?*. It is challenging to accurately find circumstances where an SSL technique is effective and also finding to what degree these circumstances are suited. Further, unlabelled data does not always improve results. [78]

Experiments in the literature have shown worsening of results. Its existence has probably been under-reported, because of publication bias [86]. Several papers have shown the challenge of possible result worsening using SSL [86] [20] [76] [52] [64], but it is still challenging to handle.

This challenge is distinctively present in settings where good results are reached using purely supervised classifiers. Potential result degradation is larger compared to potential improvement, within these scenarios. The key point here is that SSL does not always improve results. Instead, while locating and setting up a learning method for a challenge, it should be observed as an alternative direction. [78]

In summary, SSL does not always improve performance. This is also the case within NLP. This is useful knowledge when setting up a learning technique because it means other techniques could work better.

There are multiple parameters which can potentially significantly impact SSL performance within text classification. These include hyperparameters, including preprocessing dataset size, and model size.

Preprocessing Data Size

Raffel [72] et al. and Baevski et al. [5] show that reducing the volume of pretext data can result in performance degradation of downstream tasks. Raffel et al. [72] recommend utilizing a significant amount of pretext data if feasible. This is because more pretext data can improve results, and obtaining additional unlabeled data is simple and inexpensive. Raffel et al. [72] conclude that a larger network could overfit on a small quantity of pretext data. Therefore, they point out that this recommendation aims in particular at extensive networks.

In essence, more preprocessing data can improve SSL performance within

NLP. This is useful knowledge if the goal is to improve SSL performance in NLP.

Model Size

Enlarging network size and/or training time improve results generally. Compared to only enlarging batch size or training time, enlarging network size results in another bump in results. For improving results, expanding model size and training time can be matching techniques. [72]

According to a recent paper by Bender et al. [8], the expanding scale of language models as estimated by volume of training data and parameters has within NLP been a significant trend. They further claim that it seems creating increasingly larger scale language models is a contest between organizations.

In summary, more expansive models improve SSL performance within NLP. This is also important knowledge if the aim is to improve SSL performance in NLP.

Hyperparameters

There exists much literature showing that hyperparameters are highly relevant for achieved performance using SSL.

Devlin et al. [29] observe that learning with a significant data amount and enlarging hyperparameters, improves particular GLUE [82] results.

In the RoBERTa paper [54], Liu et al. claim that model training is not cheap computationally. They state that training is commonly accomplished using different expansive private datasets. They show hyperparameters have a significant impact on SSL performance. See Subsection 2.2.1.

You et al. [84] show that with 32k as batch size, BERT training time can be shortened significantly without result degradation.

Dai and Le [26] conclude LSTM models can be trained and reach good results on multiple text classification challenges, with fine adjustment of hyperparameters.

Modifying particular hyperparameters frequently impacts the results more than others. According to Goodfellow et al. [35], "The learning rate is perhaps the most important hyperparameter".

Overall, literature shows hyperparameters have a significant impact on SSL performance within NLP. There exist hyperparameter explorations for word2vec and BERT, for example with [18] and [54] respectively. However, there is not a significant volume of research exploring impact of parameters on SSL performance for text classification, in particular within the context described in Section 3.2.

2.2.5 Summary of Literature Review

Sequential transfer learning is useful when for example having a large unlabeled data amount and a small labeled data quantity. Fine-tuning a pretrained model for multiple downstream tasks is also useful.

Unsupervised pre-training is significantly scalable due to not requiring labeled data. Multi-task pre-training can improve performance by training tasks in parallel with a shared representation.

Word embeddings result in more generalized models, compared to one-hot encoding, for example. Important papers include the following: Latent Semantic Analysis [27], Word2vec [61] and Glove [66].

Neural language models achieve improved results compared to statistical language models, mainly due to generalizing better with word embeddings. Using word embeddings, Neural Language Models (NLMs) solve the n-gram data sparsity problem. Important papers include the following: The initial NLM by Bengio et al. [9], CoVe [58], ELMo [67], ULMFiT [42], GPT [70], BERT [29], ALBERT [49], GPT-2 [71], RoBERTa [54], T5 [72], GPT-3 [13], Switch Transformers [32] and the paper by Bender et al. [8].

Multiple semi-supervised classification techniques are used. Unsupervised preprocessing utilizes labeled and unlabelled data in two phases. Unlabelled data moves the decision border towards potentially more useful areas using pre-training, prior to supervised learning.

Self-learning is probably the first concept utilizing unlabeled data for classification challenges. During the 1990s SSL utilization grew [83] [57] [12] [25] [45], due to uses mainly in natural language challenges and text classification.

Self-supervised learning extracts knowledge using unlabeled data, with a pre-training, also called *pretext*) task [44]. State-of-the-art techniques are usually prediction-based [85]. Successful utilization of self-supervised learning with language modeling challenges and variations has been central for the popularity of transfer learning within NLP [56].

Initial research in self-supervised pre-training for deep models attempt to train deep belief networks [39] and stacked auto-encoders [10]. Instead of greedy layer-wise unsupervised learning, end-to-end training where a deep model learns in a single operation has grown in popularity.

SSL does not always improve performance. To potentially improve SSL performance within NLP, the following can be done: Using a larger preprocessing data size, using more extensive models, or changing hyperparameters.

Chapter 3

Thesis Definition and Method

This chapter first presents the Thesis Definition (3.1), which includes thesis goals and hypotheses. It then describes Contributions (3.2) and proposed solutions.

3.1 Thesis Definition

This thesis explores the impact of different parameters on an SSL technique for a text classification task. Nine thesis goals and four hypotheses are defined.

3.1.1 Thesis Goals

See Subsection 3.1.2 for the context and definitions for the goals and hypotheses.

Goal 1: Through experimentation, investigate if the hyperparameter configuration has a significant impact on the SSL performance.

Goal 2: Through experimentation, investigate if additional pre-training data improve the SSL performance.

Goal 3: Through experimentation, investigate if the learning rate has the most significant impact among the hyperparameters on the SSL performance.

Goal 4: Through experimentation, investigate if larger models improve the SSL performance.

Goal 5: Explore background theory particularly on transfer learning, SSL, and self-supervised learning.

Goal 6: Explore previous research of transfer learning, SSL, self-supervised learning, and impact of parameters on SSL performance within NLP.

Goal 7: Create an experimentation program for obtaining results.

Goal 8: Create datasets for pre-training, downstream, and supervised phases.

Goal 9: Create three models of different sizes for experimentation.

3.1.2 Hypotheses

This thesis uses smaller models and a limited number of epochs. Smaller models have several benefits, including lower hardware requirements and faster model training. All experiments use 200 epochs in each training phase. Therefore, 200 epochs are used during pre-training, supervised, and downstream training.

The hypotheses are within the context of *Natural Language Processing (NLP)*. Therefore, these hypotheses only apply to the NLP domain. More specifically, these hypotheses are within the context of text classification.

Hypothesis 1: The hyperparameter configuration has a significant impact on the SSL performance.

The hyperparameter configuration means the hyperparameters used. *SSL performance* means how effective SSL is for text classification, compared to using supervised learning. Therefore, Hypothesis 1 claims that hyperparameters have a significant impact on the SSL performance.

Hypothesis 2: Additional preprocessing data improves the SSL performance.

Preprocessing data refers to the dataset used during preprocessing training in SSL. *Preprocessing* means pre-training a model using unlabeled data, with a pre-training task. This aims to improve performance for the downstream text classification task. *The downstream text classification task* means the text classification task after pre-training the model. For this thesis, the terms *preprocessing* and *pre-training* are used interchangeably. Hypothesis 2 claims that a more substantial pre-training dataset in SSL improves SSL performance for text classification.

Hypothesis 3: The learning rate has the most significant impact among the hyperparameters on the SSL performance.

Why we singled out the learning rate here, is due to the fact that the learning rate has an immediate effect on how fast a network learns features in the data. For example, a too low learning rate does not allow the pre-training to learn the underlying features fast enough. Hypothesis 3 claims that the learning rate is the most impactful hyperparameter on the SSL performance.

Hypothesis 4: Larger models improve the SSL performance.

Models means neural networks. *Larger models* refers to models containing more sophisticated architectures. Therefore, Hypothesis 4 claims that models containing more sophisticated architectures improve SSL performance for text classification.

3.2 Contributions

This thesis explores the impact of hyperparameters, including the volume of pre-training data and model size, on an SSL technique for a text classification task. This is in the context of hardware constraints, so a limited amount of epochs and smaller models are used. There is not a significant bulk of other research exploring the impact of these parameters, within this context.

A program for experimentation was created using the code-base from an earlier project [53]. The source code is located on GitHub [1].

3.3 Structure of Experiments

Two experiment types are run for each model: Supervised learning experiments and SSL experiments. Supervised learning experiments train model with labeled data only, without pre-training. After supervised learning, the model has finished training. SSL experiments pre-train model using unlabeled data first. Then, the model fine-tunes with labeled data. The fine-tuned model has then finished training. See Figure 3.1 below.

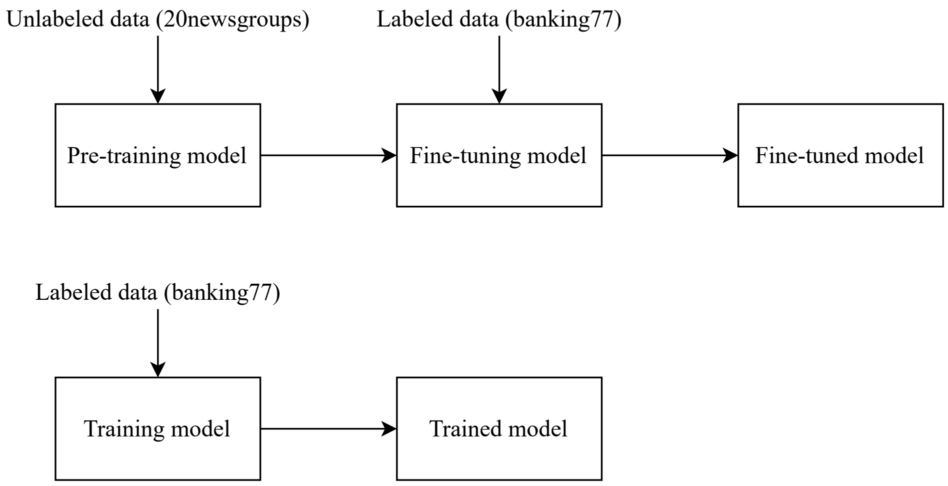

Figure 3.1: Structure of experiments using SSL (top) and supervised learning (bottom)

3.4 Models

Our experiments use three models of different sizes. The first is a feed-forward model, the second is a recurrent model containing a GRU layer, and

[1]https://github.com/Henrik0808/ikt590-project

the third is a sequence to sequence model with an encoder and a decoder. The decoder uses an attention mechanism.

3.4.1 Feed-Forward Model

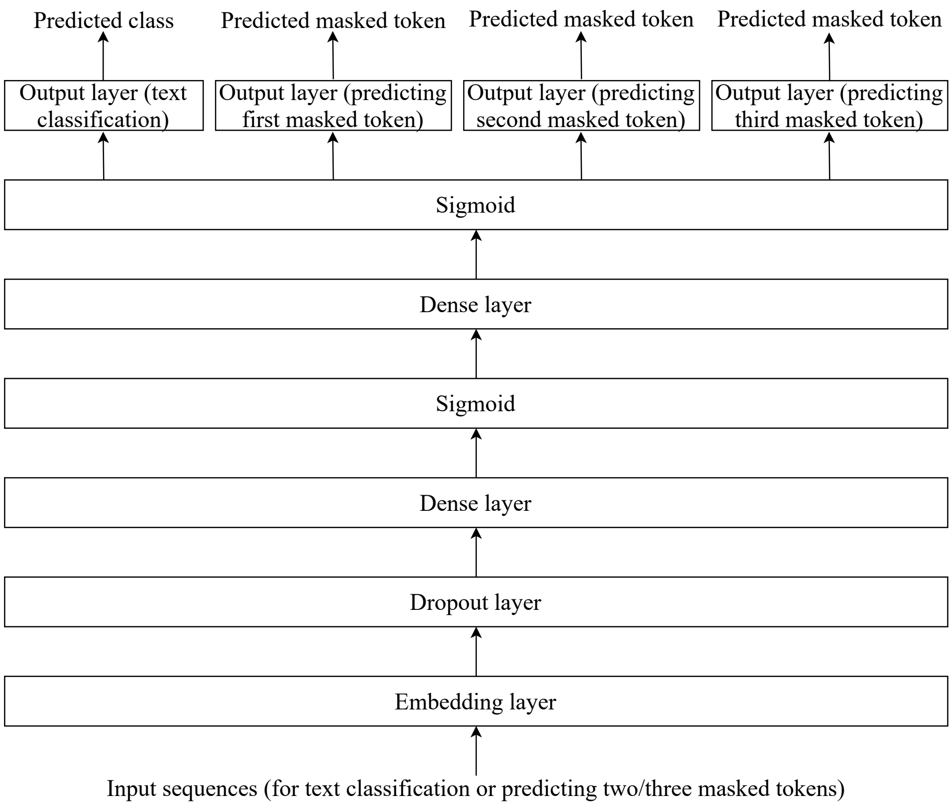
Predicted class
Predicted masked token
Predicted masked token
Predicted masked token
Output layer (text classification)
Output layer (predicting first masked token)
Output layer (predicting second masked token)
Output layer (predicting third masked token)
Sigmoid
Dense layer
Sigmoid
Dense layer
Dropout layer
Embedding layer
Input sequences (for text classification or predicting two/three masked tokens)

Figure 3.2: Architecture of feed-forward model

This is the least sophisticated model for experimentation. See Figure 3.2. It contains an embedding layer, a dropout layer, two hidden layers, and one or multiple output layers. It uses a single output layer for the text classification task because this requires only a single output. For the pretext task with predicting two or three masked tokens, it uses two or three output layers. These output layers are sharing hidden layers. Therefore, this model does multi-task pre-training with hard parameter sharing. See Subsection 2.2.1.

Figure 3.2 only uses the left-most output layer among the output layers during text classification training. It only uses two or three of the other right-most output layers during pre-training.

For each used vocabulary word, the embedding layer contains the word embedding. Output from the embedding layer is put through the dropout layer. This layer disables randomly a proportion of neurons within the embedding layer.

Each hidden layer is put through the *Sigmoid* activation function, which constrains outputs to be between zero and one.

An output layer contains multiple nodes, which reflect the number of possible labels. Therefore, for the classification task, the output layer contains 77 nodes which mean 77 possible labels. For the pretext task of predicting masked tokens, the number of nodes in each output layer is the number of used vocabulary words.

3.4.2 GRU model

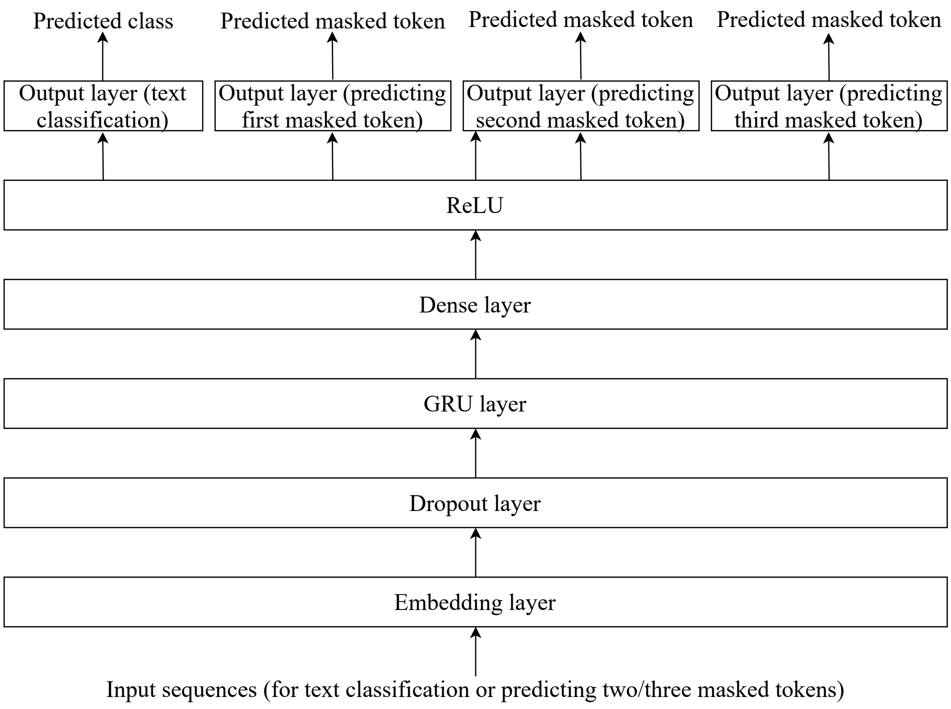
Predicted class
Predicted masked token
Predicted masked token
Predicted masked token
Output layer (text classification)
Output layer (predicting first masked token)
Output layer (predicting second masked token)
Output layer (predicting third masked token)
ReLU
Dense layer
GRU layer
Dropout layer
Embedding layer
Input sequences (for text classification or predicting two/three masked tokens)

Figure 3.3: Architecture of GRU model

Compared to the feed-forward model, the GRU model is more sophisticated. See Figure 3.3. This model contains an embedding layer, a dropout layer, a GRU layer, a dense hidden layer, and a single or two/three output layers. The output layers are similar to the output layers in the feed-forward model.

Similar to Figure 3.2, Figure 3.3 only utilizes the left-most output layer among the output layers during supervised or downstream training. For pre-training, it only utilizes two or three of the other output layers.

Embedding layer output advances through a dropout layer. The final GRU hidden state goes through the dense hidden layer. Dense hidden layer output goes through a *ReLU* activation function, which disables negative inputs and leaves positive inputs unchanged. The result then goes to the output layer(s).

3.4.3 Sequence to Sequence (Seq2seq) Model With Attention Mechanism

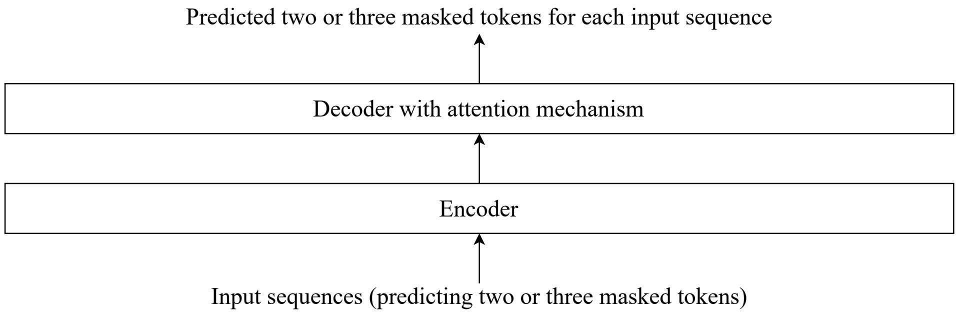
Predicted two or three masked tokens for each input sequence
Decoder with attention mechanism
Encoder
Input sequences (predicting two or three masked tokens)

Figure 3.4: High-level architecture of seq2seq model with an attention mechanism

For experimentation, this model is the most most sophisticated. It contains an encoder and a decoder. The decoder implements an attention mechanism.

The encoder contains an embedding layer, a dropout layer, a bidirectional GRU layer, and the *hidden* layer.

The decoder contains an embedding layer, a dropout layer, a GRU layer, an *energy* layer, and an output layer. Embedding layer output goes through the dropout layer. Result proceeds to the *energy* layer, and output is modified by ReLU. The *softmax* activation function modifies the resulting output. Inputs are then between zero and one, and when added together sum to one. The extracted vector is known as *attention*. The attention vector and *encoder_states* are used to generate the *context_vector*. The attention technique is described in Subsection 2.1.4.

The implemented *Seq2seq* model with attention mechanism is based on code from GitHub by aladdinpersson [3].

This model does single-task learning during pretext task training, in contrast to other models which are doing multi-task learning. See Subsection 2.2.1. Pretext task training uses the *Seq2seq* model, while supervised text classification training uses the *Classifier* model. Both these models contain an encoder. In contrast to the Seq2seq model, the Classifier model does not have a decoder. The Classifier achieved similar performance as the Seq2seq model during testing for the previous project [53]. Therefore, to improve training time, the Classifier model was used when possible for the earlier project. For this thesis also, the Classifier model is utilized for the text classification task. Similar to the earlier project report [53], this thesis simplifies the term usage. If in reality the Classifier model is utilized, it is still called the *Seq2seq* model. The Classifier model is further described in Subsection 3.4.4 below.

3.4.4 Classifier Model

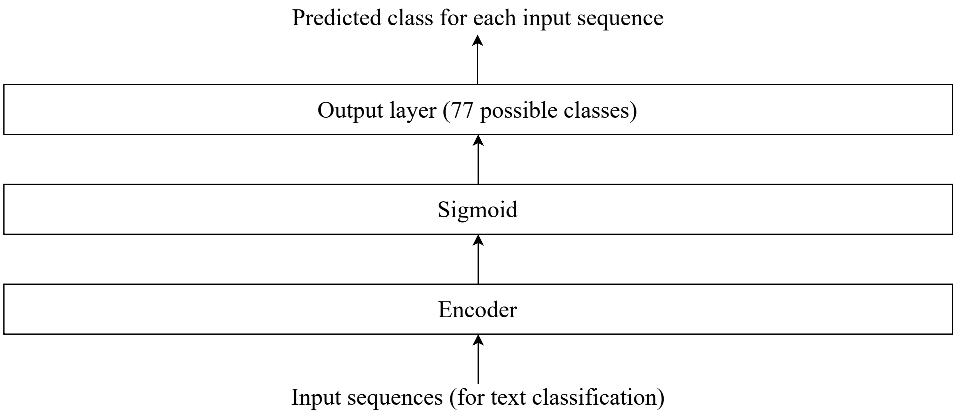
Predicted class for each input sequence
Output layer (77 possible classes)
Sigmoid
Encoder
Input sequences (for text classification)

Figure 3.5: High-level architecture of classifier model

This model has an encoder and an output layer. The hidden vector from the encoder goes through a Sigmoid activation function. The resulting hidden vector then proceeds to the output layer. Instead of a decoder, this model uses a standard output layer.

3.5 Running Experiments

This section presents a detailed step-by-step description of the workflow in the code.

Before experimentation, the program creates a tokenizer. This tokenizer is created using two datasets. These are the 20newsgroups and Banking77 datasets. The tokenizer is created using eleven-word sequences in the 20newsgroups dataset and Banking77 queries. The tokenizer vocabulary is limited to words used at minimum twice. This lowers the number of words containing one-time writing errors in the tokenizer vocabulary, resulting in a higher quality vocabulary.

The tokenizer tokenizes training and validation datasets. This way, datasets are prepared for experimentation with model training and validation. A data loader is created for each dataset to simplify the loading of experiment data. Each data loader reshuffles data for every epoch. Experiments run on a GPU, if available. If not, they run on a CPU. Each model runs both supervised learning and SSL experiments.

During experimentation, different data are saved to disk. These include model checkpoints, training loss and validation loss for each epoch, loss graphs, and a summary file containing for example best-achieved loss and accuracy for each model and training approach.

Experimentation does multi-task pre-training with hard parameter sharing (2.2.1) for the feed-forward and recurrent models. These models learn one task for each token prediction. The seq2seq model does single-task learning. Therefore, this model combines individual token prediction tasks into one task. Therefore, the optimization task in pre-training utilizes one loss function for a single task.

An experiment run creates a model initially. A downstream task experiment initializes model weights with the best model weights obtained during pretext task training. Best model weights achieve the lowest validation loss. If not pre-trained, model weights are randomly initialized. A model trains and validates for a particular number of epochs. For training, a data loader loads one batch at a time. For training or validation using the feed-forward model, all sequences in a batch are padded with zeros. This way, all sequences have equal length. The feed-forward model expects inputs to have equal length. All sequences are padded to equal length, where this length is the largest sequence in the total training data. This includes both the 20newsgroups and Banking77 data. The most expansive sequence is in the Banking77 dataset, with a length of 79 without the starting "*sos*" and "*[mask]*" tokens. Therefore, with the feed-forward model, all input sequences pad with zeros to a length of 79. For other models, sequences in a batch are padded so that they have the same length as the longest sequence in the batch. This is for use with the *pack_padded_sequence* function in the code.

During preprocessing training or validation, each batch uses dynamic masking. The "*[mask]*" token replaces two or three random tokens in each sequence. A sequence example is "*I think that asking the wrong question is probably the*". If two tokens are masked, this sequence can change to for example: "*I [MASK] that asking the wrong [MASK] is probably the*". Sequence target is then: "*think, question*". By randomly masking sequences in each batch, preprocessing dataset is enlarged artificially. This results in less overfitting and more general features learned during preprocessing, compared to if preprocessing dataset is static. Sequences in preprocessing batches randomly mask during both training and validation.

For each batch during training, gradients zero out first. Then sequences proceed to model, and model output(s) are extracted. CrossEntropyLoss calculates loss. Loss gradients are then computed and finally, model parameters are updated using the Adam optimizer.

During training for one epoch, summing individual losses in all batches calculates total loss. After every batch is through during an epoch, the total loss is divided by the number of batches. The result is the average batch loss for a single epoch, which is observed in the loss graph. Each epoch also calculates accuracy, which is not used for this thesis.

Validation batch sequences proceed to the model, and model output(s) are extracted. CrossEntropyLoss calculates loss. Loss value and accuracy for a single epoch are calculated similarly to during training.

During an experiment, the model achieving the lowest validation loss is saved to disk. The downstream task loads the saved pre-trained model parameters from preprocessing. This initializes model parameters for the downstream task.

3.6 Data Handling

For pre-training, a training, a validation, and a testing dataset are created from the original 20newsgroups [50] dataset. Using newsgroups mentioned in 3.8, data is extracted from the original 20newsgroups dataset. This data contains no headers, footers, or quotes. This stops models from overfitting using metadata. Unwanted characters are removed from this data. These include for example commas, parentheses, and underscores. A sequence is also removed if it contains particular sequences, for example "—", "@" or "==". This cleans the data further.

Data is split into sentences. Sentences which do not at the minimum contain 11 words are skipped. For each sentence, a sliding 10-word window iterates over the words. For each 10-word sequence, the words "*sos*" and "*[MASK]*" are inserted before the 10 words themselves. This includes the "*sos*" and "*[MASK]*" tokens in the resulting tokenizer. The eleventh word is also used, mainly because of using the codebase from the previous project [53].

Using training, validation, and testing ratios mentioned in Subsection 3.8, data is split into training, validation, and testing datasets. Training, validation, and testing datasets are saved to disk. During experimentation, these are used for preprocessing. An example 10-word sequence is "*PIONEER 13 sent four small probes into the atmosphere in*, with eleventh word *December*".

Both supervised and downstream text classification training use data from the original Banking77 [17] dataset. Testing data from this dataset is used in experiments as validation data. The Banking77 dataset contains online banking queries, with matching intents. This dataset contains in total 10003 training examples, 3080 testing examples, and 77 intent categories. An example query from the Banking77 dataset is "*Is there a way to know when my card will arrive?*", with corresponding intent "*card_arrival*". In the previous project [53], we only increased labeled and unlabeled data simultaneously, while in this thesis we increase unlabeled data independently of the labeled data.

Hyperparameter	Value
Training, validation and test ratio	0.8, 0.1 and 0.1 respectively
Dataset size	25k, 50k, 75k, 100k, 125k, 150k, 175k, 200k, 300k, 400k or 500k
Newsgroups selected	15 newsgroups (see names below)
Length of embedding vector	256 or 512
Batch size	512 or 1024
Length of word sequences	10
Length of hidden vector	512 or 1024
Number of GRUs to stack	1
Dropout	0.0 or 0.2
Learning rate	0.0001 or 0.001
Number of masked tokens	2 or 3
Number of epochs	200

Table 3.1: Overview of hyperparameters used in this thesis

3.7 Preprocessing/Pretext Task

The pretext task used is predicting two or three masked tokens in a sequence. The "*[mask]*" token replaces two or three random tokens. The objective is to predict each masked token.

3.8 Hyperparameters

This thesis uses multiple hyperparameters for experimentation. Table 3.1 shows an overview of hyperparameters used in this thesis. More detailed, these hyperparameters are:

- Training, validation and test ratio. A training ratio of 0.8 (80 %), a validation ratio of 0.1 (10 %) and a test ratio of 0.1 (10 %) are used for creating training, validation and testing datasets from the dataset generated from the original 20newsgroups dataset. Only training and validation datasets are used for experiments. If for example dataset size is set to 100 000, then $0.8 * 100000 = 80000$ sentences are created for training dataset, $0.1 * 100000 = 10000$ sentences are created for validation dataset and $0.1 * 100000 = 10000$ sentences are created for testing dataset.

- Dataset size. The number of 10-word sequences to generate from the original 20newsgroups dataset, to generate training, validation, and testing datasets from.
- Newsgroups from the 20newsgroups dataset to generate datasets from. 15 newsgroups are used: 'soc.religion.christian', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.med', 'sci.space', 'sci.electronics', 'talk.politics.misc', 'talk.politics.guns', 'talk.politics.mideast' and 'misc.forsale'.
- Length of embedding vector. For experiments, this is set to either 256 or 512.
- Batch size. For experiments, this is set to either 512 or 1024.
- Length of word sequences to generate from the original 20newsgroups dataset. Training, validation, and testing datasets are created from these sequences. The value of 10 is used here, which means that 10-word sequences are generated from the 20newsgroups dataset.
- Length of hidden vectors in models. For experiments, this value is 512 or 1024.
- Number of GRUs to stack together in recurrent models for each GRU layer. For experiments, this value is 1.
- Dropout value. For experiments, this value is 0.0 or 0.2.
- Learning rate. For experiments, this value is 0.0001 or 0.001.
- Number of randomly masked tokens in each 10-word sequence from preprocessing dataset. For experiments, this value is 2 or 3.
- Number of epochs for supervised, preprocessing, and downstream training. 200 epochs are used for each training category.

Chapter 4

Results and Discussion

This chapter presents and discusses the results of our experiments, summarizes the main results for each hypothesis, and concludes the main findings regarding hypotheses and goals.

For testing the thesis hypotheses (3.1.2), four experiments are run:

1. Experiment 1 shows the impact of hyperparameter configuration and model size on the SSL performance.
2. Experiment 2 presents the impact of increasing pre-training dataset size on the SSL performance.
3. Experiment 3 shows the impact of changing single hyperparameter on the SSL performance.
4. Experiment 4 presents the impact of changing two hyperparameters on the SSL performance.

These experiments result in a total of seven figures, presented below.

Three different models are used for experimentation: A feed-forward model, a GRU model, and a seq2seq model. Multiple preprocessing dataset sizes are used: 25k, 50k, 75k, 100k, 125k, 150k, 175k, 200k, 300k, 400k, and 500k. Here, “k” means thousand.

Supervised, preprocessing, and downstream training use 200 epochs each during experimentation. Because of time constraints, it is not practical to use more epochs. Validation loss in preprocessing graphs frequently starts to stabilize at about 200 epochs. See for example Figures 4.3 and 4.4. We did not consider using fewer epochs, as the validation loss for pre-training has not stabilized then. Therefore, 200 epochs are used. Additional epochs would make the validation loss of preprocessing graphs stabilize more. See for example Figure 4.3. This would be ideal, for models to learn more during preprocessing training. This is considered future work.

Multiple hyperparameters are considered for experiments. These include the number of masked tokens in each 10-word sequence, learning rate, hidden size, batch size, embedding size, and dropout size.

This thesis uses two baseline hyperparameter configurations for experimentation. The reason behind this is to test if hyperparameter configuration has a significant impact on SSL performance for text classification. Frequently, hyperparameters are chosen by manual tuning or by inspiration from previous research. As manual tuning is computationally expensive, in this thesis we do not tune manually using all possible combinations. Approaches for creating hyperparameter configurations are not important for this thesis. The key point is to use two different configurations. For example, if a hyperparameter configuration significantly improves SSL performance compared to the other configuration, then Hypothesis 1 is supported. Throughout the experiments, these two configurations are modified. See sections below for each experiment rationale.

The first hyperparameter configuration, called *Vanilla configuration*, is meant to represent a typical configuration of hyperparameters used in machine learning.

The second configuration, called *SOTA configuration*, is partly based on hyperparameters in two GitHub projects [19] [51]. These projects are based on a paper [26] by Dai and Le.

Each SOTA configuration hyperparameter value, except batch size and the number of masked tokens, is used by minimum one of the GitHub projects [19] [51]. Both Github projects use a batch size of 64. The Github projects use a language model or an auto-encoder as a pre-trained model. The number of masked tokens in 10-word sequences is not relevant in these GitHub projects.

Hyperparameter	**Vanilla**	**SOTA**
Embedding size	512	256
Batch size	512	1024
Hidden size	1024	512
Dropout rate	0.0	0.2
Learning rate	0.0001	0.001
Number of masked tokens	2	3

Table 4.1: Baseline Vanilla configuration and SOTA configuration hyperparameters for experimentation

The reason for using an expanded batch size, in particular for SOTA configuration, is to use less training time per epoch. This is based on manual testing. For example, the set of possible batch sizes for both Vanilla configuration and SOTA configuration is limited. This is mainly due to time constraints.

To the best of our knowledge, there is only limited research using masked language modeling with smaller models. By using masked language modeling in the preprocessing, we test the effectiveness in smaller models. Masked language modeling artificially enlarges the preprocessing dataset by randomizing for each batch which sequence tokens to mask. Because of time constraints, experimentation uses only the masked language modeling objective as preprocessing technique. Doing experiments with other preprocessing techniques is considered future work.

The SOTA configuration masks three tokens in each ten-word sequence. This is based on minimal manual hyperparameter testing, using a few thousand preprocessing data. This results in a masking percentage of 30 percent for each 10-word sequence. This thesis always represents a masked token by the "[mask]" token. Inspired by BERT [29], it is possible that not always representing masked tokens with "[mask]" could improve SSL performance.

This thesis focuses on how much SSL improves text classification loss compared to supervised learning. Therefore, it focuses on relative text classification loss, instead of absolute text classification loss. This thesis calculates *SSL performance* with the following approach: The lowest achieved text classification validation loss using supervised learning, minus the lowest achieved text classification validation loss using SSL for the text classification task. See Equation 4.1. Therefore, if SSL performance > 0, then

SSL improves validation loss for the text classification task compared to supervised learning. In other words, SSL improves text classification performance. If SSL performance < 0, in this case, supervised learning reaches lower validation loss compared to the SSL approach. Here, using SSL would not benefit the text classification performance. For this thesis, "improving SSL performance" means achieving higher SSL performance.

$$\text{SSL performance} = L_S - L_{SSL} \tag{4.1}$$

In Equation 4.1, L_S is the text classification loss using purely supervised learning, and L_{SSL} is the text classification loss using SSL.

4.1 Experiment 1: Impact of hyperparameter configuration and model size on the SSL performance

Experiment 1, represented by Figure 4.1, contributes to answer Hypothesis 1 and Hypothesis 4. For each parameter configuration and model, we repeat the simulation ten times. These ten repetitions generate the box plot of the respective configuration. This shows how much variance we can expect between simulations and if the differences between the models are within or outside this variance. If these differences are outside this variance, this supports Hypothesis 1. Hypothesis 4 is also supported if more sophisticated models improve SSL performance.

The expectation here is that box plots for each model will not "overlap" significantly, due to the impact of hyperparameters found in previous research (2.2.4). Another expectation is that more expansive models improve SSL performance compared to smaller models, based on research mentioned earlier (2.2.4). Therefore, the expectation is that this experiment supports Hypothesis 1 and Hypothesis 4.

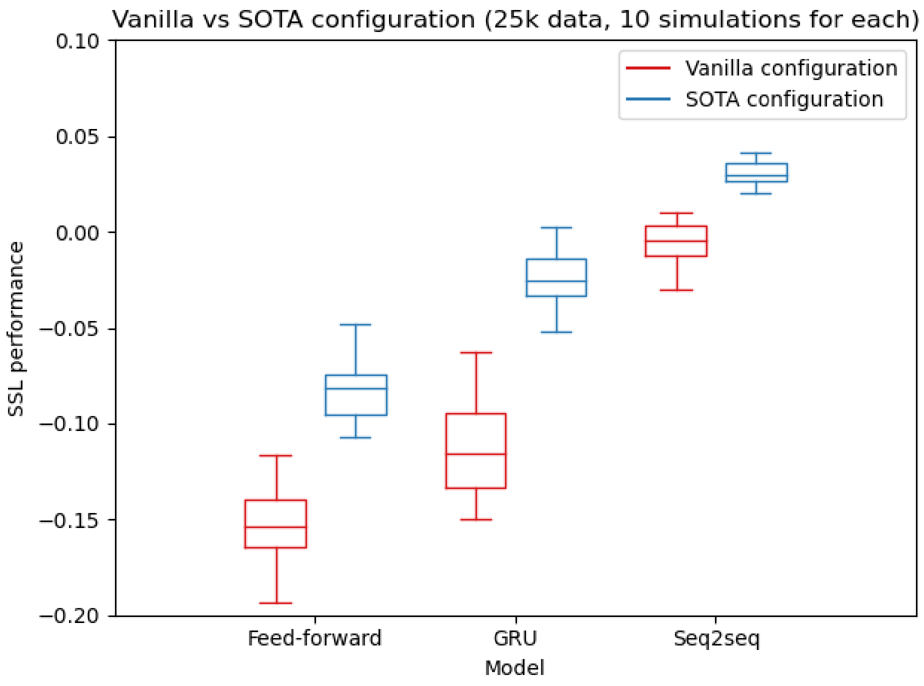

Figure 4.1: Vanilla configuration versus SOTA configuration, using 25k preprocessing data, 200 epochs for each training phase, and ten simulations for each configuration.

Figure 4.1 shows multiple box plots. The y-axis represents SSL performance as defined above. The x-axis represents the experiment model. This figure shows multiple experiment results. Each combination of model and configuration repeats simulation ten times, using 25k preprocessing data and 200 epochs. These 10 repetitions are used to plot the box plot of the respective configuration.

For all models with SOTA configuration, SSL performance improves compared to using Vanilla configuration. This supports Hypothesis 1, because it shows that the hyperparameter configuration has a significant impact on SSL performance. It is expected that one configuration improves SSL performance compared to another. Ten simulations is not a substantial number of simulations for each model, so this figure should not be observed as significantly conclusive. However, this figure shows that using SOTA configuration generally improves SSL performance.

One possible explanation for this is: SOTA configuration uses a larger learning rate and fewer model parameters, therefore it learns more over 200 epochs. Dropout makes it overfit less, making the model learn more general features. This improves SSL performance. This is only speculation.

As expected, more extensive models improve SSL performance compared to smaller models. This supports Hypothesis 4. It is possible that with more epochs, all models achieve better SSL performance – particularly the feed-forward model.

4.2 Experiment 2: Impact of increasing pre-training dataset size on the SSL performance

Experiment 2, represented by figures 4.2 and 4.5, contributes to answer Hypothesis 2 and Hypothesis 4. Here, preprocessing data volume is increased. The reasoning behind this is to test if increasing preprocessing data amount improves SSL performance. If results show this, then these results support Hypothesis 2. This experiment also investigates if larger models improve SSL performance. If they do, this supports Hypothesis 4. Vanilla configuration and SOTA configuration are each used while increasing the data.

The general expectation is that increasing preprocessing data leads to performance improvements across all models. This expectation is based on research described earlier (2.2.4). Further, because larger models in Experiment 1 generally improve SSL performance compared to smaller models, it is expected they also generally improve SSL performance here. Therefore, the expectation is that this experiment supports Hypothesis 2 and Hypothesis 4.

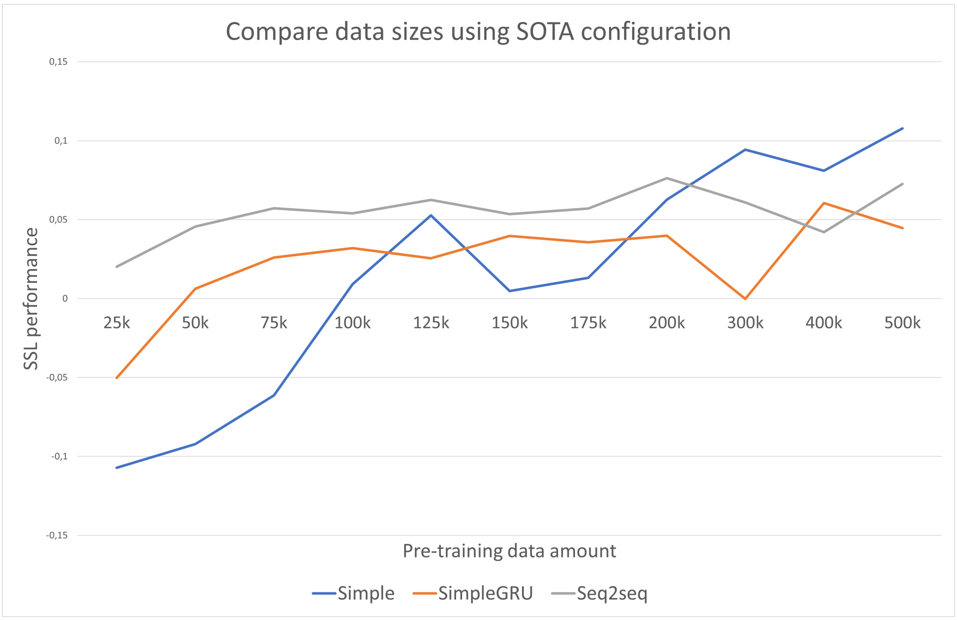

Figure 4.2: Comparing preprocessing data amounts, using SOTA configuration and 200 epochs for each training phase

Figure 4.2 shows the impact of changing preprocessing dataset size using SOTA configuration. The blue line shows feed-forward model results, the orange line shows GRU model results, and the grey line shows seq2seq model results. Similar to previous experiments, each training phase uses 200 epochs.

The x-axis in Figure 4.2 represents preprocessing dataset size. The leftmost x-value, *25k*, means the preprocessing dataset consists of 25k 10-word sentences. The next x-value, *50k*, means that preprocessing dataset size is 50k 10-word sequences. This logic applies to the other x-values also. The y-axis axis represents SSL performance.

In case of the feed-forward model, increasing the preprocessing data amount leads to an increase in SSL performance. This result supports Hypothesis 2 and is expected. As the pre-training data quantity increases with other models, SSL performance reaches a threshold after about 100k preprocessing data. Therefore, in this experiment, a feed-forward model benefits from as much pre-training data as possible. With a recurrent model, increasing preprocessing data size beyond 100k does not improve the SSL performance significantly. This result partially supports Hypothesis 2 because increasing the pre-training data improves SSL performance up to a threshold. This result is unexpected.

One possible explanation for this is that with recurrent layers a model learns faster, and learns a larger number of useful features from preprocessing data. This is because of using more parameters. Therefore, more expansive models containing more parameters require less pre-training data to learn the most useful features during preprocessing. For this reason, increasing the pre-training data amount indefinitely benefits the SSL performance of the feed-forward model more, compared to the other two models. This is only speculation. It is possible that Figure 4.2 looks different with other hyperparameters. Creating additional similar figures using other hyperparameters, is considered future work. It is also possible that increasing preprocessing data quantity to for example one million results in SSL performance improving significantly for recurrent models also. Experiments with additional pre-training data are also considered future work.

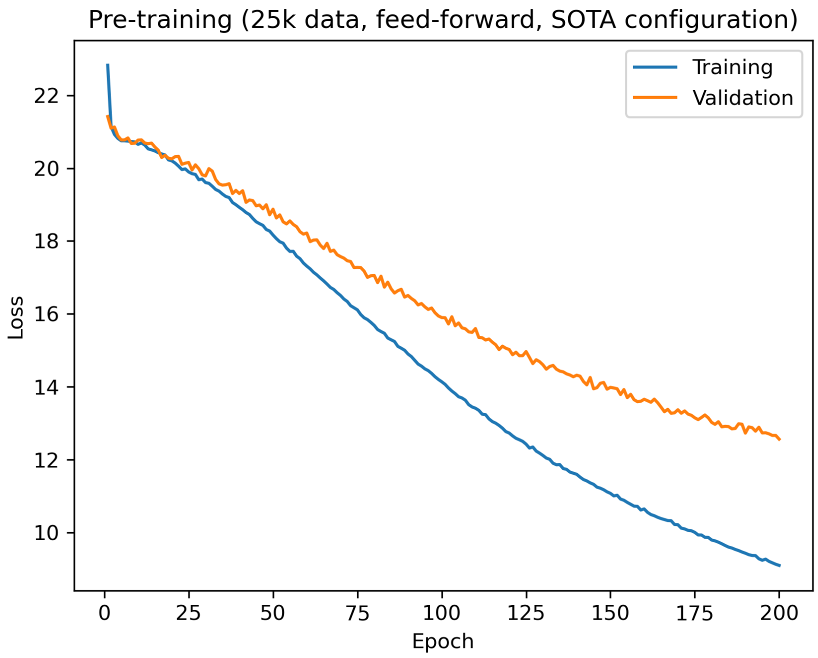

Figure 4.3: Pre-training using the feed-forward model with SOTA configuration.

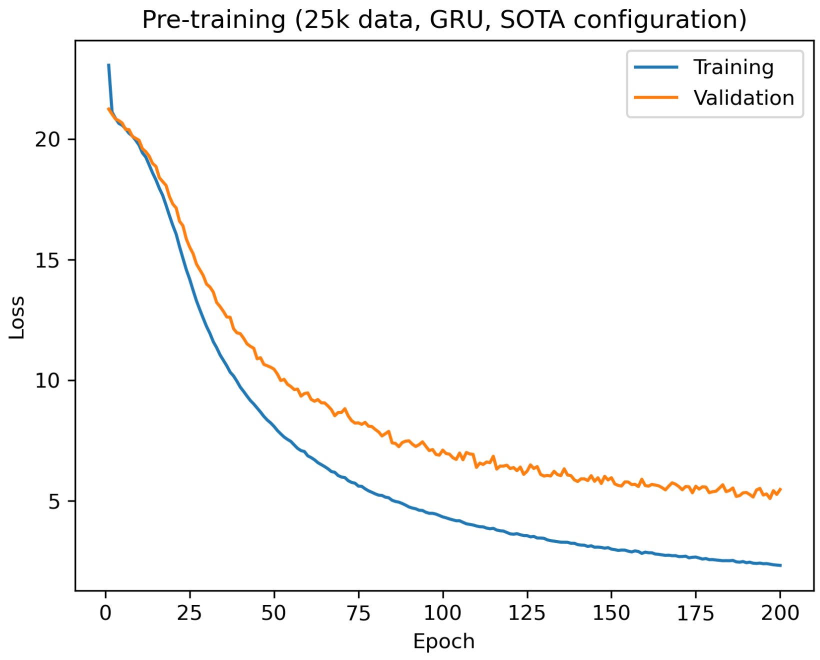

Figure 4.4: Pre-training using the GRU model with SOTA configuration.

Pre-training figures 4.3, 4.4 and 4.9 show that for larger models, loss graphs converge faster. Therefore, more comprehensive models learn more useful features from preprocessing data per epoch. Also, more extensive models reach lower loss values as expected, meaning they have learned more useful features over 200 epochs. Both these observations support the possible explanation above. These pre-training figures show that the more converged a loss graph is, the less use of additional preprocessing data improves SSL performance. This can be observed in Figure 4.2. A possible reason for this is that a converged model from pre-training has already learned most of the useful features for the downstream task. In particular, the feed-forward model still learns valuable features per epoch after the 200th epoch. Additional preprocessing data for the feed-forward model results in a greater number of features learned from pre-training data per epoch due to more training samples. This results in the model learning a larger number of useful features through 200 epochs. Therefore, in case of the feed-forward model, the SSL performance benefits from more preprocessing data. This is only speculation. If we would not restrict the number of epochs, we could pre-train each model to a similar level of convergence. It would be interesting to see whether increasing pre-training data size then would improve SSL performance less compared to Figure 4.2 for all models. This is because each model then already would have learned many useful features for the downstream task. This is aimed at more expansive models more than smaller ones, due to capturing more complex and useful features from less preprocessing data compared to the smaller models. Experiments using additional epochs are considered future work.

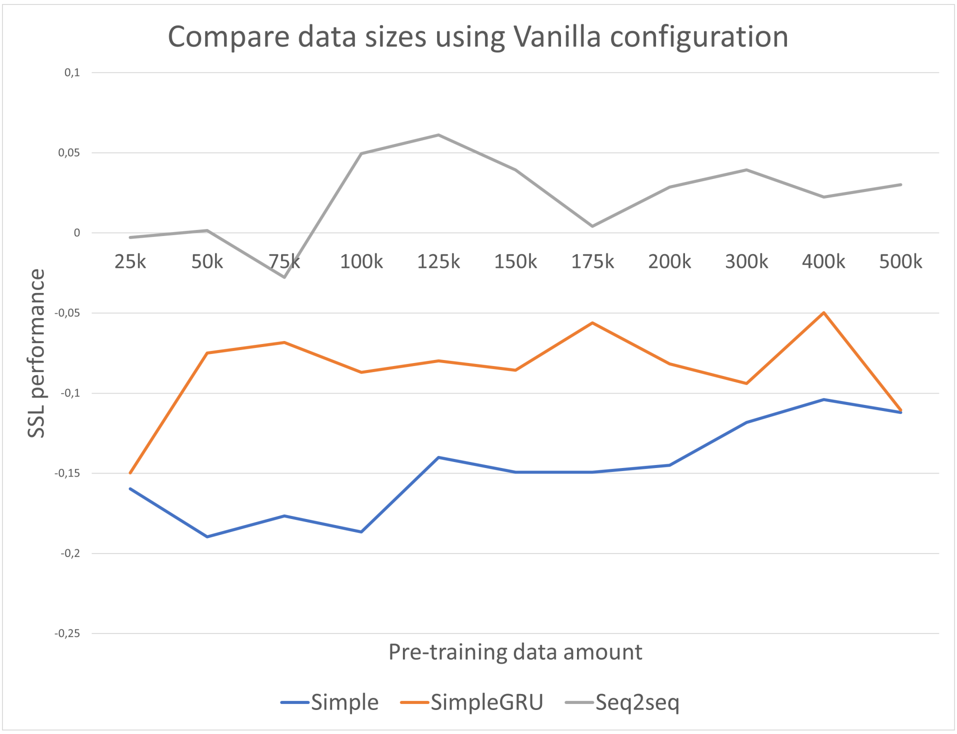

Figure 4.5: Comparing preprocessing data amounts, using Vanilla configuration and 200 epochs for each training phase

Figure 4.5 is similar to Figure 4.2. The difference is that Figure 4.5 uses Vanilla configuration, and Figure 4.2 uses SOTA configuration.

Increasing preprocessing data volume with SOTA configuration results in SSL performance generally improving for all models, at minimum up to a threshold. Increasing preprocessing data size with Vanilla configuration has a smaller impact on improving SSL performance, particularly for the feed-forward model compared to if using SOTA configuration. This supports Hypothesis 1 and is unexpected.

One possible explanation for this is: The Vanilla configuration has a lower learning rate and additional model parameters, leading to slower learning compared to SOTA configuration using a higher learning rate and fewer model parameters. Therefore, with SOTA configuration, increasing the volume of preprocessing data has a larger impact on SSL performance compared to Vanilla configuration. This is due to the higher learning rate and the fixed amount of 200 epochs, which particularly restricts Vanilla configuration using a lower learning rate. Additional epochs for Vanilla configuration potentially make the Vanilla configuration figure more similar to the SOTA configuration figure, because Vanilla configuration then converges and learns more during pre-training.

4.3 Experiment 3: Impact of changing single hyperparameter on the SSL performance

Experiment 3, represented by figures 4.6, 4.12 and 4.17, contributes to answer Hypothesis 1, Hypothesis 3 and Hypothesis 4. One individual hyperparameter is modified at a time from SOTA configuration or Vanilla configuration. The rationale behind this is to identify the most impactful hyperparameters, and to test if the impact of changing hyperparameters depends on other hyperparameters. For example, it is possible that lowering the learning rate improves SSL performance in one experiment and not in another. This suggests then that the impact of changing hyperparameters depends on other hyperparameters. However, if for example changing the learning rate has a significant impact on SSL performance for both configurations, then this supports Hypothesis 1 and Hypothesis 3. By changing one hyperparameter at a time, the impact of changing hyperparameters can be compared to each other. This way, hyperparameters with the most impact on SSL performance are identified. If for example, larger models in figures generally improve SSL performance compared to smaller models, this supports Hypothesis 4. Hyperparameters are modified both with 100k preprocessing data and 25k data. This tests if preprocessing data amount has an impact on the results of modifying hyperparameters.

The expectation is that some hyperparameters will have more impact on SSL performance than others. For example, the learning rate is generally known to have a significant impact on model training. Therefore, the expectation is that the learning rate generally has a strong impact here also. Another expectation is that more extensive models improve SSL performance compared to smaller models due to capturing additional useful features during pre-training. Therefore, the expectation is that this experiment supports Hypothesis 1, Hypothesis 3, and Hypothesis 4. Another expectation is that increasing pre-training data quantity does not have a substantial impact on the results of changing one parameter at a time. It is also expected that hyperparameters with significant impact using one configuration, also have a strong impact on the other configuration.

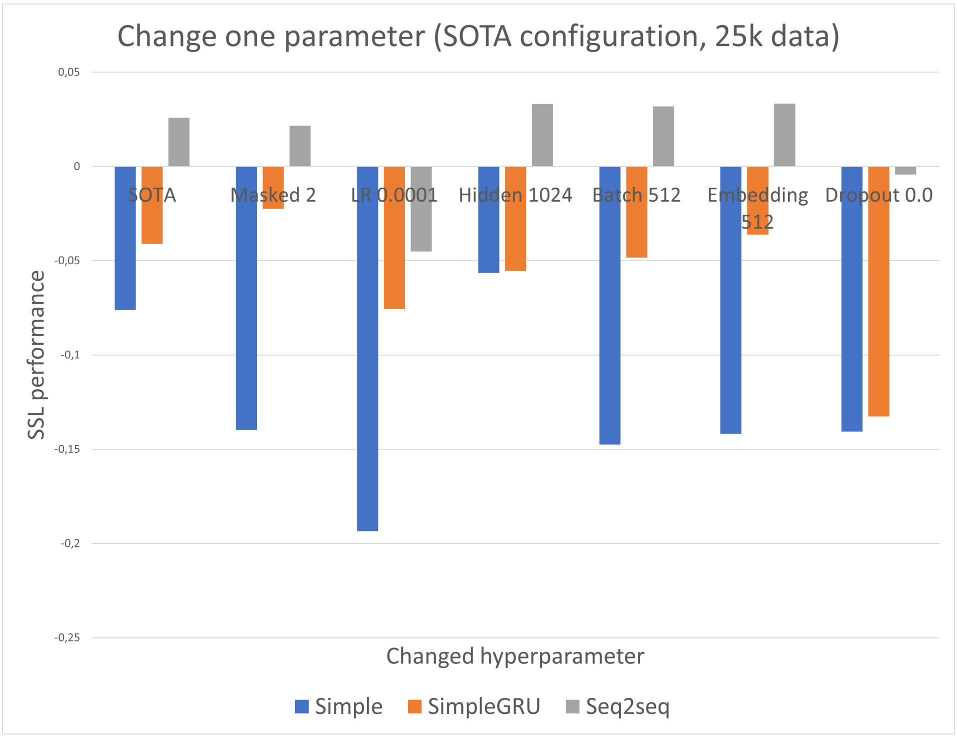

Figure 4.6: Changing one parameter at a time from SOTA configuration, using 25k preprocessing data and 200 epochs for each training phase

Figure 4.6 shows the impact of changing one hyperparameter at a time from SOTA configuration while keeping everything else fixed. Blue bars represent feed-forward model results, orange bars represent GRU model results and grey bars represent seq2seq model results.

In Figure 4.6, the x-axis represents the modified hyperparameter. The left-most x-value means that all hyperparameters are unchanged. The next x-value, *Masked 2*, means that the number of masked tokens in each 10-word sequence is changed from 3 to 2, while all other hyperparameters are unchanged. The next x-value, *LR 0.0001*, means that the learning rate is modified from 0.001 to 0.0001, while all other hyperparameters are unchanged. This logic applies to the other x-values also. The y-axis represents the SSL performance.

Figure 4.6 displays experiment results using a preprocessing dataset size of 25k 10-word sequences, and 200 epochs for each training type.

Changing the learning rate from 0.001 to 0.0001 significantly lowers the SSL performance for each model. Therefore, changing the learning rate here to 0.0001 worsens SSL performance compared to using the learning rate of 0.001. This result supports Hypothesis 1 and Hypothesis 3. It is expected that the learning rate has a significant impact on SSL performance.

A possible explanation for this result is that increasing the learning rate by a factor of ten results in a significantly faster model adjustment to the pre-training task. Because the number of epochs is fixed at 200, increasing the learning rate results in more substantial weight modifications per update. This results in faster model converging. If the learning rate is 0.0001 instead of 0.001, then the model adapts significantly slower to the pre-training task through 200 epochs. Here, during pre-training, the number of epochs is not enough for the model to adapt adequately to the problem.

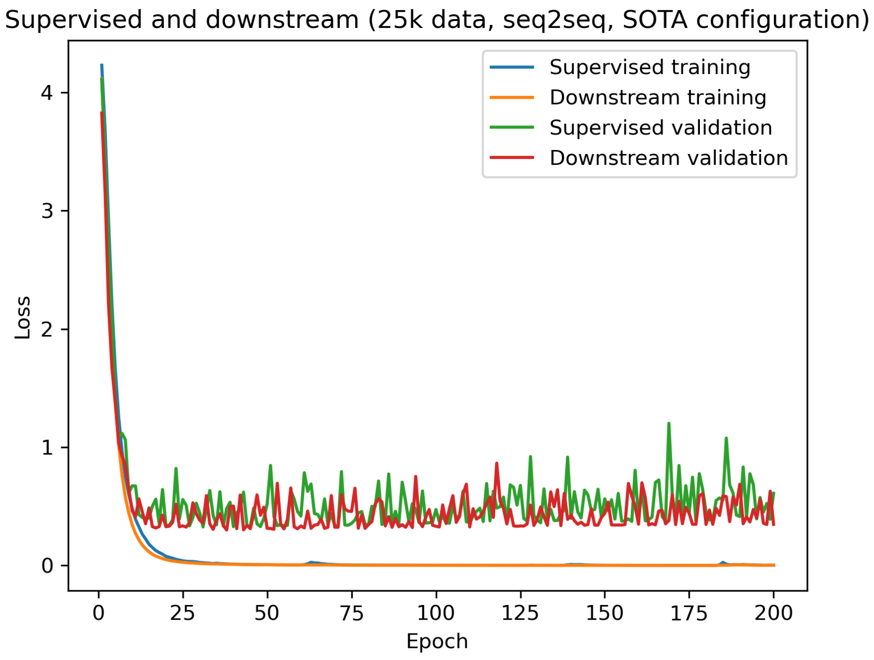

Figure 4.7: Supervised and downstream training using the seq2seq model, with the LR set to 0.001

Figure 4.7 shows loss graphs for both the supervised and the downstream text classification task. The y-axis represents the loss value, and the x-axis represents the epoch number. The blue and green lines show loss graphs for the supervised approach. The blue line uses training data, and the green line uses validation data. The orange and red lines show loss graphs for the downstream task. The orange line uses training data, and the red line uses validation data. This graph shows experimental results for the seq2seq model, with a learning rate of 0.001.

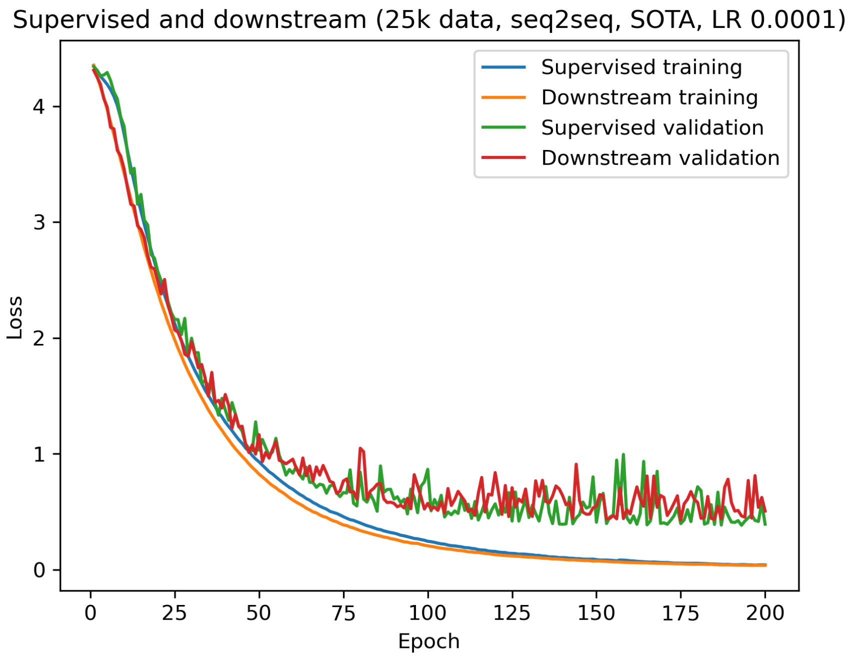

Figure 4.8: Supervised and downstream training using the seq2seq model, with the LR set to 0.0001

Figure 4.8 is similar to Figure 4.7. The difference is that Figure 4.8 uses a learning rate of 0.0001, while Figure 4.7 uses a learning rate of 0.001.

These two figures show that a learning rate of 0.001 results in loss graphs converging significantly faster, compared to a learning rate of 0.0001. Therefore, a learning rate of 0.001 here results in a lower required number of epochs and training time.

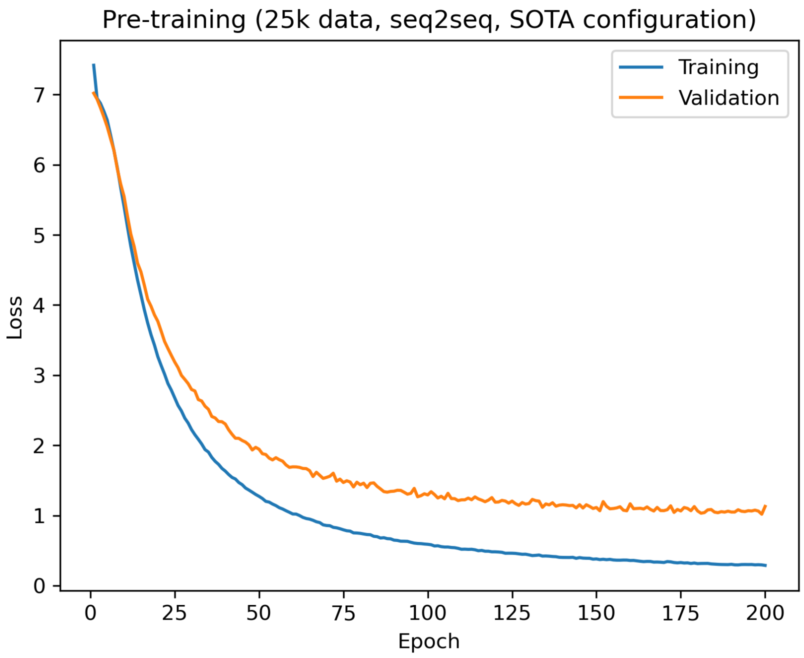

Figure 4.9: Preprocessing training using the seq2seq model, with the LR set to 0.001

Figure 4.9 shows loss graph for the pre-training task using SSL. The y-axis represents the loss value, and the x-axis represents the epoch number. The blue line shows the loss graph for the pre-training task using training data, and the orange line shows the loss graph for the pre-training task using validation data. This graph shows experimental results for the seq2seq model, with a learning rate of 0.001.

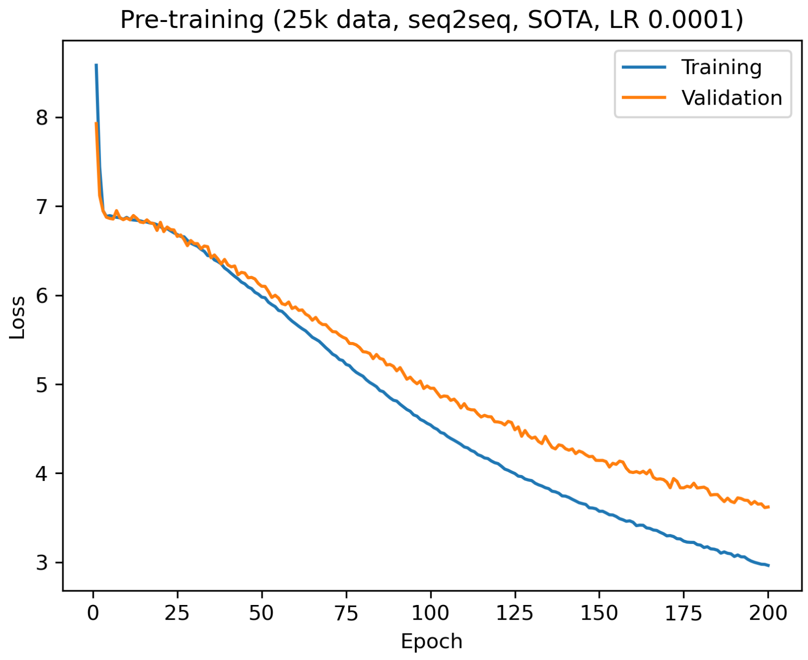

Figure 4.10: Preprocessing training using the seq2seq model, with the LR set to 0.0001

Figure 4.10 is similar to Figure 4.9. The difference is that Figure 4.10 uses a learning rate of 0.0001, and Figure 4.9 uses a learning rate of 0.001.

These two figures also show that using a learning rate of 0.001 results in a loss graph to converge significantly faster compared to a learning rate of 0.0001. Therefore, again, using a learning rate of 0.001 results in a significantly smaller required amount of epochs and training time. A learning rate of 0.001 results in the model converging within 200 epochs, while a learning rate of 0.0001 results in the model not converging within 200 epochs. If the model has not converged, then the model has not adjusted completely to the problem. This means it is not done with learning useful features. This comparison supports the possible explanation provided above for why a higher learning rate improves SSL performance compared to a lower learning rate. A larger learning rate here results in the model learning more useful features in fewer epochs. If the model has learned more useful knowledge during pre-training, it frequently has adapted better to the downstream task. This improves downstream performance.

Another observation is that changing the dropout rate from 0.2 to 0.0 also lowers SSL performancees significantly. This also supports Hypothesis 1 and is an unexpected result.

One possible explanation for this is related to the amount of preprocessing data. Because of using only 25k preprocessing data samples, models probably overfit on this data during pre-training.

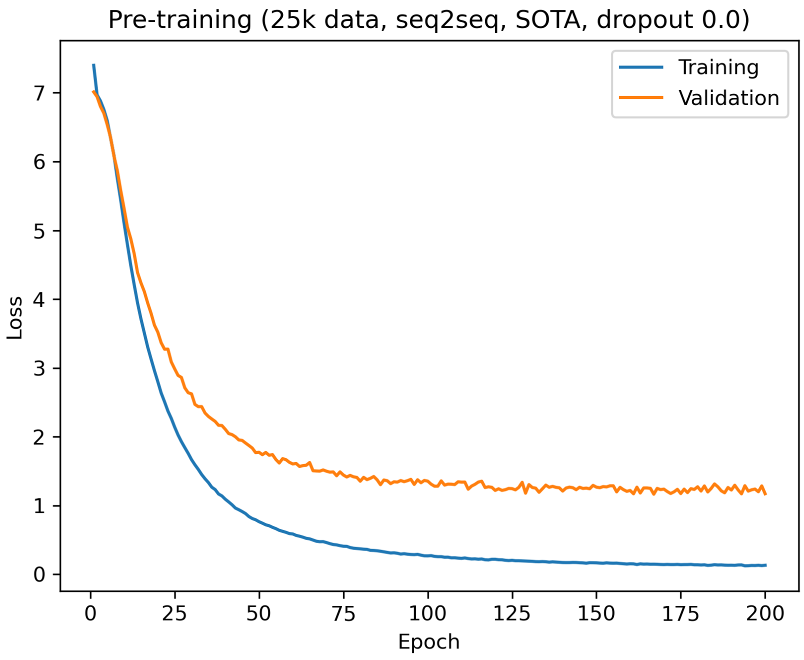

Figure 4.11: Preprocessing training using the seq2seq model, with the dropout rate set to 0.0

Figure 4.11 overfits more compared to Figure 4.9. Comparing the validation loss graphs, this is observed. Larger distance between the training and validation loss graphs means more overfitting. Dropout is known for lowering overfitting and improving the generalization of deep neural networks. Adding dropout results in models learning more general knowledge during pre-training, which improves downstream classification performance on validation data. Therefore, dropout improves SSL performance, resulting in better SSL performances in Figure 4.6.

The feed-forward model is frequently more sensitive to hyperparameter modifications, compared to other models. The smaller and less sophisticated models are generally more sensitive to hyperparameters, compared to other models. This supports Hypothesis 4 and is not an unexpected result, given that smaller models generally are less powerful for learning than more expansive models.

One possible explanation for this result is: Less sophisticated models are more dependant on tuned hyperparameters for learning effectively because they often do not learn effectively compared to more sophisticated models. Because of being less sophisticated, they need to start learning from a better "initial point" compared to other models for learning effectively. Therefore, smaller and less sophisticated models are more sensitive to hyperparameters. Potentially, more epochs could lead to models showing similar sensitivity with regards to hyperparameters. This is due to allowing smaller models to converge more during pre-training, and learning more useful knowledge for the downstream classification task.

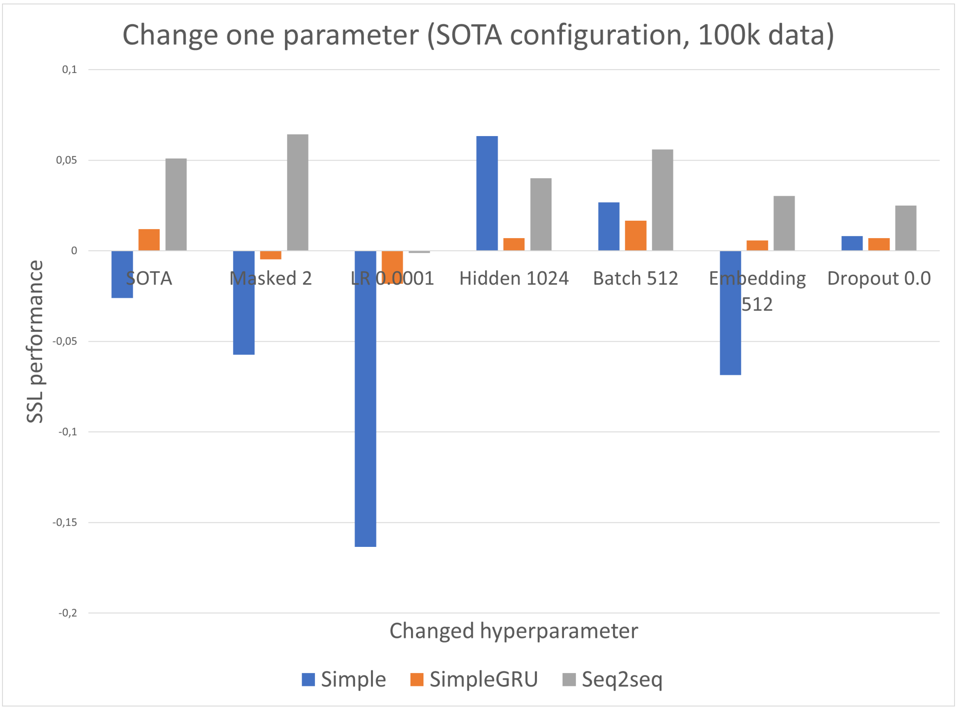

Figure 4.12: Changing one parameter at a time from SOTA configuration, using 100k preprocessing data and 200 epochs for each training phase

Figure 4.12, similar to Figure 4.6, shows impact of changing one hyperparameter at a time using SOTA configuration. There is a difference between Figure 4.12 and Figure 4.6. Figure 4.12 shows experiment results using a preprocessing dataset volume of 100k 10-word sequences, while Figure 4.6 shows results using a preprocessing dataset amount of 25k.

Figure 4.12 presents several interesting results. For example, comparing this figure to Figure 4.6, the learning rate is an important hyperparameter to consider for SSL in both figures. For all models, in Figure 4.12, changing the learning rate from 0.001 to 0.0001 results in worse SSL performance compared to when learning rate is 0.001. This result supports Hypothesis 1 and Hypothesis 3 and is expected.

Comparing Figures 4.13, 4.14, 4.15 and 4.16 to figures using 25k data, learning rate is an important hyperparameter to consider regardless of the quantity of preprocessing data. One possible explanation for this is given above, which also applies when preprocessing data size is changed from 25k to 100k.

Comparing figures using 25k preprocessing data to figures using 100k preprocessing data, one unexpected result is that preprocessing data size has an impact on the significance of single hyperparameters. For example, using 25k data, changing dropout to 0.0 has a significant negative impact on SSL performance, for all models. Using 100k pre-training data, changing dropout to 0.0 has a minor negative impact for the GRU and seq2seq models, while for the feed-forward model it has a small positive impact.

One possible explanation for this is that larger models with recurrent layers have more use of dropout compared to the feed-forward model because of more overfitting without dropout. The feed-forward model does not overfit as much – also without dropout. Because the feed-forward model has a less sophisticated architecture, it overfits less. This improves results without dropout. Dropout makes the feed-forward model ignore too much useful information. Using 25k preprocessing data, changing dropout to 0.0 makes the feed-forward model overfit more because of the small pre-training data size. This leads to worse SSL performance. Using 100k preprocessing data, changing dropout to 0.0 for the feed-forward model is fine because the pre-training data amount is expansive enough for it to learn the most useful features. Removing dropout results in the feed-forward model to learn more useful features through 200 epochs compared to using dropout. This improves SSL performance. It is possible that more epochs to make all models converge during pre-training would result in all models improving SSL performance using a dropout of 0.2 compared to 0.0 to overfit less. This is only speculation.

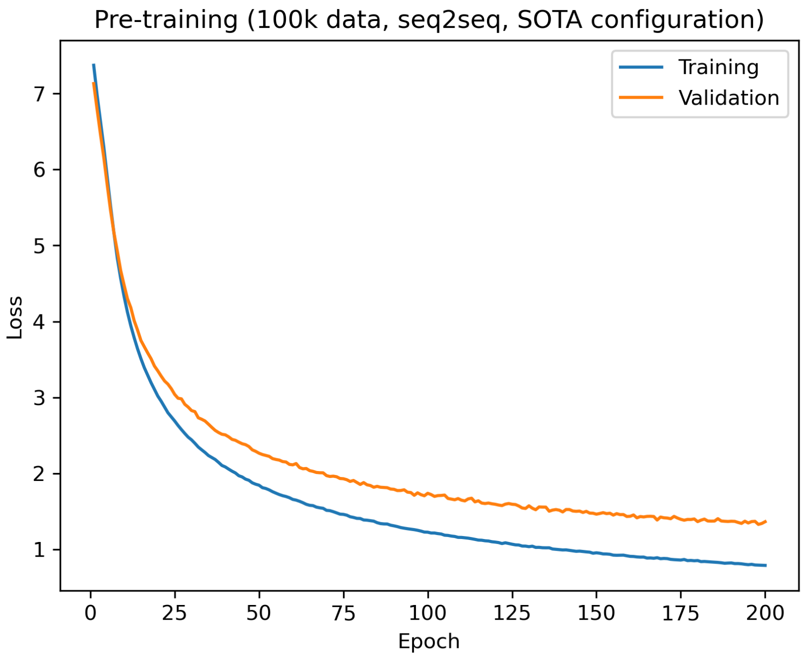

Figure 4.13: Preprocessing training for the seq2seq model, using SOTA configuration and 100k preprocessing data

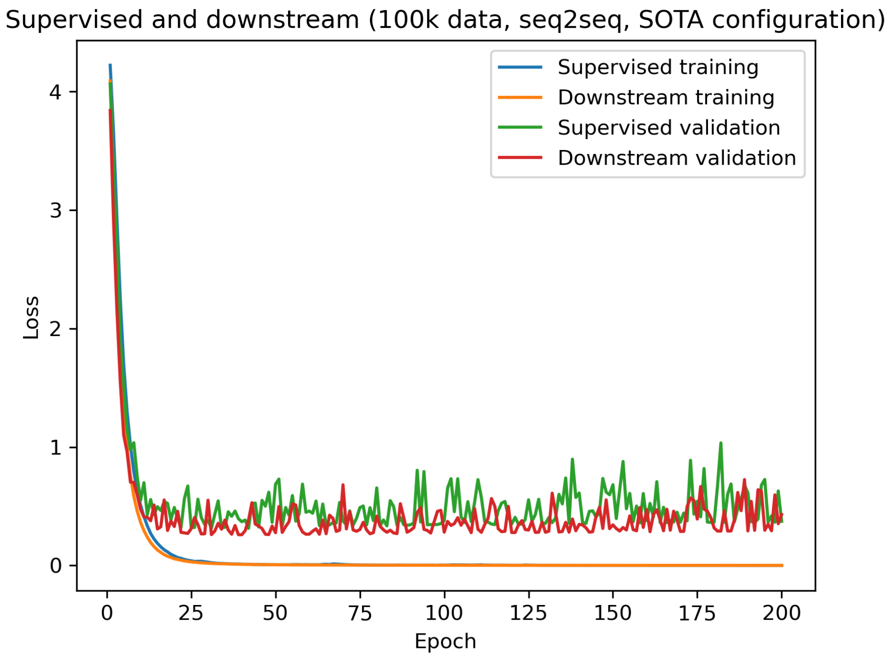

Figure 4.14: Supervised and downstream training for the seq2seq model, using SOTA configuration and 100k preprocessing data

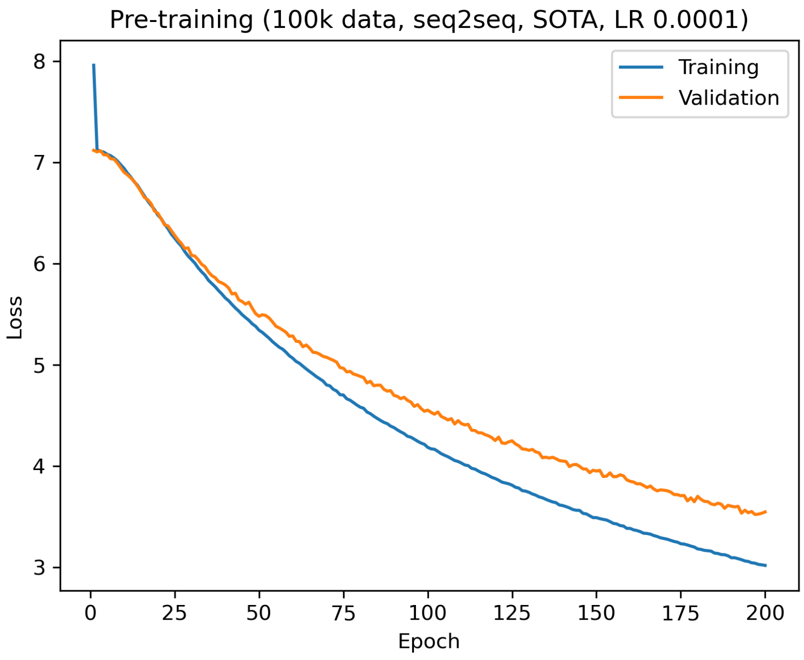

Figure 4.15: Preprocessing training for the seq2seq model, using SOTA configuration, but with learning rate set to 0.0001 and using 100k preprocessing data

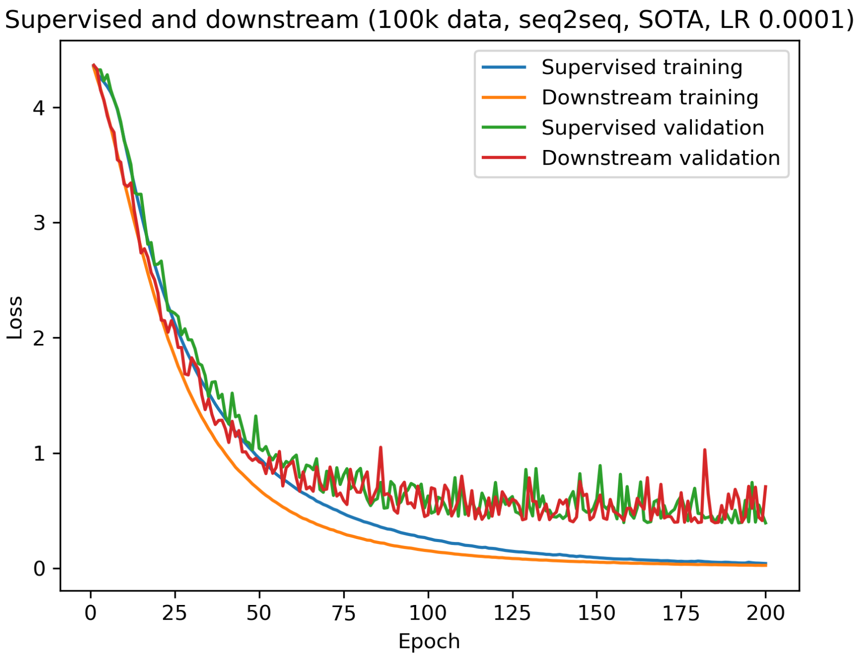

Figure 4.16: Supervised and downstream training for the seq2seq model, using SOTA configuration, but with learning rate set to 0.0001 and using 100k preprocessing data

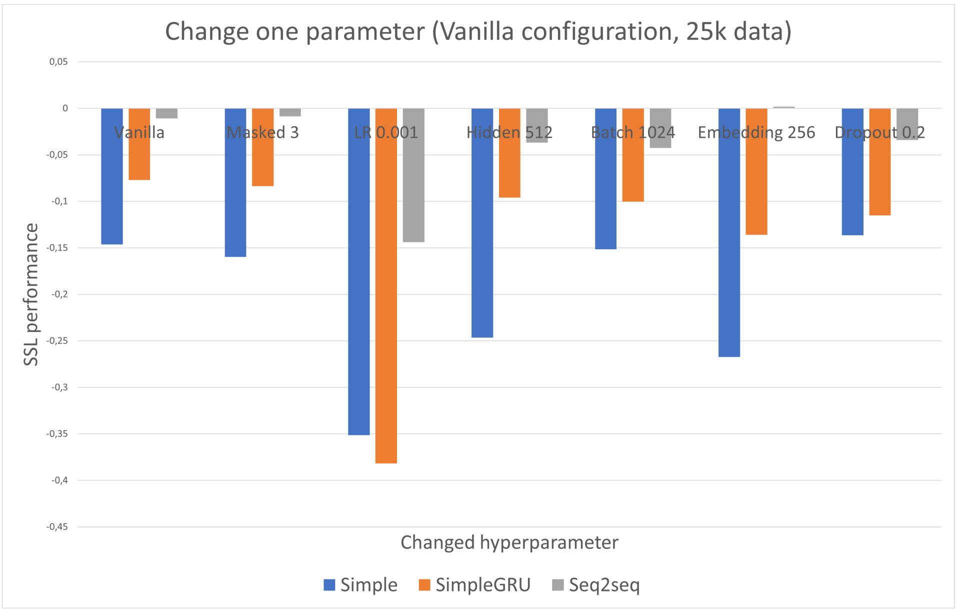

Figure 4.17: Changing one parameter at a time from Vanilla configuration, using 25k preprocessing data and 200 epochs for each training phase

Figure 4.17 is similar to Figure 4.6. The difference is that Figure 4.17 uses Vanilla configuration, while Figure 4.6 uses SOTA configuration.

Figure 4.17 presents several interesting results. For example, similar to observations above, learning rate here also is an important hyperparameter to consider for SSL. This supports Hypothesis 1 and Hypothesis 3, and is expected.

Comparing Figures 4.17 and 4.6 for example, an observed unexpected result is: Figure 4.17 changes learning rate from 0.0001 to 0.001 and Figure 4.6 changes learning rate from 0.001 to 0.0001, but the result for both is that SSL performance worsens after changing the learning rate. Therefore, for example the learning rate needs to be tuned based on other hyperparameters.

For Figure 4.6, a possible explanation for why changing learning rate worsens SSL performance is given above. For Figure 4.17, a possible explanation for why changing the learning rate from 0.0001 to 0.001 worsens SSL performance is the following: The combination of larger embedding size, hidden size, and no dropout leads to more overfitting on pre-training data. This results in worse SSL performance when increasing the learning rate due to overfitting and learning non-useful features faster. This is only speculation.

Figure 4.6 shows that changing dropout to 0.0 makes SSL performance significantly worse for all models, and Figure 4.17 shows that changing dropout does not change results as significantly. This result is unexpected.

The result of changing dropout for Figure 4.6 is discussed above. One possible explanation for the result of changing dropout for Figure 4.17 is: Using Vanilla configuration means additional model parameters and smaller learning rate, leading to slower learning compared to SOTA configuration. This leads to less convergence during pre-training with Vanilla configuration. Changing dropout to 0.2 slows learning down further using Vanilla configuration, which worsens SSL performance with 200 epochs. The feed-forward model is the exception here, which can be a random incident. It is possible that if more epochs are used, then using a dropout of 0.2 would improve SSL performance for both Vanilla configuration and SOTA configuration. Due to SOTA configuration utilizing fewer model parameters and a higher learning rate, training is faster. Therefore, using dropout does not slow down training enough for it to worsen SSL performance through 200 epochs, like with Vanilla configuration. This is only speculation.

Changing a single parameter from baseline Vanilla configuration makes SSL performance worse mostly while changing all or multiple hyperparameters simultaneously can improve SSL performance. This supports Hypothesis 1 and is unexpected.

One possible explanation for this is: The combination of fewer model parameters and a larger learning rate with SOTA configuration makes training models faster over 200 epochs, where dropout reduces overfitting. Therefore, more useful features are learned over 200 epochs during pre-training, improving SSL performance. With additional epochs, it is possible that changing a single hyperparameter at a time from Vanilla configuration sometimes improves SSL performance. This is then due to enough epochs for models to converge during pre-training.

With Vanilla configuration, most hyperparameters have less impact on SSL performance compared to if using SOTA configuration. This also supports Hypothesis 1 and is unexpected.

One possible explanation for this is: With Vanilla configuration, the learning rate is smaller, resulting in a more limited ability to learn useful features and converge during pre-training over 200 epochs. Therefore, hyperparameters have mostly less impact on SSL performance with Vanilla configuration compared to SOTA configuration where the learning rate is higher. If additional epochs are used, resulting in more converging with Vanilla configuration in particular, these figures potentially show more similar results.

4.3.1 Experiment 4: impact of changing two hyperparameters on the SSL performance

Experiment 4, represented by Figure 4.18, also contributes to answering Hypothesis 1, Hypothesis 3, and Hypothesis 4.

Experiment 2 modifies two hyperparameters at a time from SOTA configuration. This tests if patterns are showing which combination of hyperparameters generally have the strongest impact on SSL performance. For example, if lowering the learning rate generally worsens SSL performance, then this supports Hypothesis 1 and Hypothesis 3. Again, if larger models generally improve SSL performance, then this supports Hypothesis 4.

The expectation is that hyperparameters with significant impact in Experiment 3 also have a substantial impact here. For example, because the learning rate and dropout have a significant impact in Experiment 3, it is expected that combinations containing these two hyperparameters also have a significant impact in this experiment. Because larger models in Experiment 3 generally improve SSL performance compared to smaller models, it is expected they also improve SSL performance here. Therefore, the expectation is that this experiment also supports Hypothesis 1, Hypothesis 3, and Hypothesis 4.

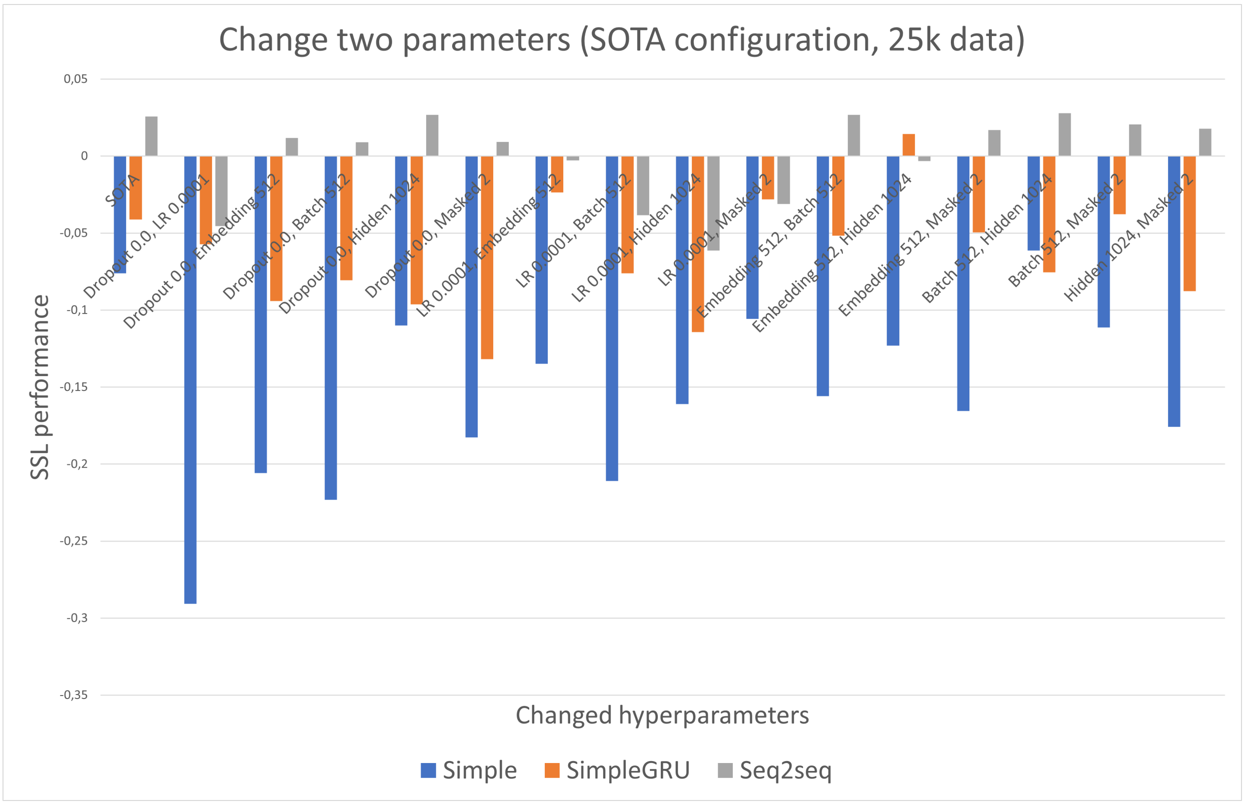

Figure 4.18: Changing two parameters at a time from SOTA configuration, using 25k preprocessing data and 200 epochs for each training phase

Figure 4.18 shows the impact of changing two hyperparameters at a time from SOTA configuration while keeping everything else fixed.

Similar to Figure 4.6, the x-axis in Figure 4.18 represents the modified hyperparameters. Again, the left-most x-value means that all hyperparameters are unchanged. The next x-value, *Dropout 0.0, LR 0.0001*, means that dropout rate is changed from 0.2 to 0.0 and learning rate is modified from 0.001 to 0.0001. All other hyperparameters are unchanged. The next x-value, *Dropout 0.0, Embedding 512*, means that the dropout rate is changed from 0.2 to 0.0, and the embedding size is changed from 256 to 512. Again, all other hyperparameters are unchanged. The other x-values follow the same pattern. In Figure 4.18, similar to Figure 4.6, the y-axis represents SSL performance.

Figure 4.18, similar to Figure 4.6, shows experiment results using a preprocessing dataset size of 25k 10-word sequences, and 200 epochs for each training phase.

Both the feed-forward and the seq2seq model obtain the highest SSL performance when modifying batch size from 256 to 512, and hidden size from 512 to 1024. The GRU model obtains the highest SSL performance when embedding size changes from 256 to 512, and hidden size changes from 512 to 1024. The common factor here is that increasing hidden size from 512 to 1024 makes SSL performance best for all three models. One possible explanation is that SSL performance improves with larger and more sophisticated models due to learning additional features during pre-training.

The differences in SSL performance might not exceed the performance variance. Therefore, if the same experiments are rerun, the results can change. Rerunning the same experiments multiple times to extract SSL performance distributions is considered future work.

Figure 4.6 shows that learning rate is an important hyperparameter to consider for SSL. Figure 4.18 shows a similar result. This figure shows that when the learning rate changes from 0.001 to 0.0001, then frequently SSL performance worsens compared to when the learning rate is 0.001. This is generally the case for the feed-forward and seq2seq models. For the GRU model, this is the case for three cases out of five. This result supports Hypothesis 1 and Hypothesis 3 and is expected.

One possible explanation that the SSL performance worsens when the learning rate is 0.0001 compared to when the learning rate is 0.001, is the same as the one suggested above. Pre-training loss graphs for simulations in Figure 4.18 are similar to loss graphs shown above. Therefore, the learning rate is generally an important hyperparameter for SSL. A possible explanation that the SSL performance for the GRU model is not lowered in the two cases mentioned above, is due to random variations during the training process. See Subsection 4.1.

Figure 4.6 shows dropout ratio is an important hyperparameter to consider for SSL, at least with 25k preprocessing data. Figure 4.18 shows a similar result. Also Figure 4.18 shows that when dropout is modified from 0.2 to 0.0, then frequently SSL performance worsens compared to when dropout ratio is 0.2. This is generally the case for the feed-forward and GRU models. For the seq2seq model, this is the case for four out of five cases. This result supports Hypothesis 1 and is expected.

One possible explanation that the SSL performance worsens when dropout is 0.0 compared to when dropout is 0.2, is the same as the one suggested above. Preprocessing figures for simulations in Figure 4.18 also show similar loss graphs as those shown above (4.11, 4.9). Therefore, in addition to the learning rate, the dropout rate is an important hyperparameter to consider for SSL. A possible explanation that the SSL performance for the seq2seq model is not lowered in the one case when dropout is lowered from 0.2 to 0.0 mentioned above, is due to random variations during the training process. See Subsection 4.1.

4.4 Conclusion of Results, Hypotheses and Goals

4.4.1 Hypothesis 1: The hyperparameter configuration has a significant impact on the SSL performance.

Based on the discussion above, the hyperparameter configuration generally has a significant impact on SSL performance – at least with a fixed number of epochs. Some hyperparameters have a stronger impact on SSL performance compared to others. Particularly the learning rate has a significant impact. The amount of preprocessing data has an impact on results of changing hyperparameters, the dropout rate for example. The impact of changing a single hyperparameter depends on other hyperparameters, for example for dropout. If changing individual hyperparameters worsens SSL performance, changing multiple hyperparameters simultaneously can improve SSL performance.

In summary, the hyperparameter configuration generally has a significant impact on SSL performance, at least with a fixed amount of epochs. If only having time for tuning one hyperparameter for SSL, the learning rate has most impact. If enough time, experimenting using different hyperparameter configurations can significantly improve SSL performance.

4.4.2 Hypothesis 2: Additional preprocessing data improves the SSL performance.

Based on the discussion above, using a higher volume of preprocessing data generally improves SSL performance, up to a threshold at the minimum. Hyperparameters have a significant impact on how much increasing preprocessing data size improves SSL performance. Choosing a hyperparameter configuration that improves SSL performance for more extensive and sophisticated models, increasing the amount of preprocessing data improves SSL performance until an upper limit is reached. For smaller models with hyperparameters tuned for SSL, increasing data improves SSL performance. With hyperparameters less suited for SSL, increasing the size of preprocessing data benefits SSL performance less, particularly for smaller models.

In summary, for hyperparameter configurations more tuned for SSL, SSL performance improves for smaller models as preprocessing data quantity increases, while larger models reach a performance threshold – at least with a fixed amount of epochs. Therefore, if using a smaller model, particularly a feed-forward model, then as much preprocessing data as possible should be used for achieving the best possible SSL performance. For more expansive recurrent models, it is not necessary to use as much preprocessing data as possible. Instead, one should experiment with different data volumes and use as little data as possible. This can potentially significantly save training time. This would also be better for the environment, for example. For other configurations less suited for SSL, the effect of increasing data quantity is not as significant, particularly for smaller models. Therefore, one should experiment using different dataset sizes and use as little data as possible – for all models. This also means that the hyperparameter configuration can be more important than the amount of pre-training data. Using the "wrong" configuration can severely hinder SSL performance even with large amounts of data.

4.4.3 Hypothesis 3: The learning rate has the most significant impact among the hyperparameters on the SSL performance.

Based on the discussion above, the learning rate has a significant impact on SSL performance. Learning rate has a significant impact regardless of pre-training dataset size, when comparing 25k and 100k data. For example, the learning rate needs tuning based on other hyperparameters.

4.4.4 Hypothesis 4: Larger models improve the SSL performance.

Based on the discussion above, larger models generally show higher SSL performance than smaller models. Our experiments show exceptions to this, but these might occur due to parameter choices. Using additional pre-training data, smaller models can catch up with more extensive models regarding SSL performance. However, throughout the entire thesis we do not look at absolute text classification performance. Instead, relative text classification performance is considered. For example in Figure 4.2, for 500k pre-training dataset size, the feed-forward model reaches higher SSL performance than the larger models. This does not mean that the feed-forward model improves absolute classification performance compared to other models. Experimenting with additional preprocessing data can potentially increase SSL performance for the feed-forward model, and maybe also for other models. The feed-forward model can benefit from even more pre-training data than used here. At the same time, there might be a point even for the feed-forward model where an upper limit is reached regarding benefit through pre-training datasize. This is considered future work.

In summary, extensive models show higher SSL performance compared to smaller models – particularly with a smaller volume of preprocessing data. With a substantial enough pre-training data size, smaller models can catch up with expansive models regarding SSL performance. With less pre-training data, a larger model frequently improves SSL performance compared to a smaller model. With a hyperparameter configuration suited more for SSL using a smaller model, particularly a feed-forward model, increasing preprocessing data amount improves SSL performance. However, if aiming to achieve the best possible absolute results for the downstream text classification task, then generally larger and more sophisticated models should be used.

4.4.5 Goals

Below is a summary for each thesis goal:

- Goal 1: *Through experimentation, investigate if the hyperparameter configuration has a significant impact on the SSL performance.* Subsection 4.4.1 summarizes the insights gained through experimentation.
- Goal 2: *Through experimentation, investigate if additional pre-training data improve the SSL performance.* Subsection 4.4.2 summarizes the insights gained through experimentation.
- Goal 3: *Through experimentation, investigate if the learning rate has the most significant impact among the hyperparameters on the SSL performance.* Subsection 4.4.3 summarizes the insights gained through experimentation.
- Goal 4: *Through experimentation, investigate if larger models improve the SSL performance.* Subsection 4.4.4 summarizes the insights gained through experimentation.
- Goal 5: *Explore background theory particularly on transfer learning, SSL, and self-supervised learning.* Subsections 2.1.5, 2.1.6 and 2.1.7 summarize the insights gained through exploration.
- Goal 6: *Explore previous research of transfer learning, SSL, self-supervised learning, and impact of parameters on SSL performance within NLP.* Subsections 2.2.1, 2.2.2, 2.2.3 and 2.2.4 summarize the insights gained through exploration.
- Goal 7: *Create an experimentation program for obtaining results.* Chapter 4 shows this goal is achieved.
- Goal 8: *Create datasets for pre-training, downstream, and supervised phases.* This goal is also achieved.
- Goal 9: *Create three models of different sizes for experimentation.* This goal is additionally achieved.

In summary, all thesis goals are achieved.

Chapter 5

Conclusion and Future Work

This chapter presents the thesis conclusion and potential future work.

5.1 Conclusion

There is a growing interest in SSL research caused by limited labeled data in many domains. However, the setup of SSL neural networks for text classification is cumbersome, frequently based on trial and error, with little knowledge on which setup is beneficial for SSL. Research has shown that SSL does not always improve performance compared to supervised learning. Additionally, in recent years, several methods have been presented that improve performance within NLP tasks using SSL. These include using additional preprocessing data, utilizing more extended models, or changing hyperparameters.

We found that the hyperparameter configuration significantly impacts SSL performance, and the learning rate has the most impact. Hence, experimenting with different hyperparameter configurations can dramatically improve SSL performance. We showed that the overall SSL performance increased with the proper hyperparameter setup for SSL-based feed-forward, recurrent, and sequence to sequence models. This underlines the significant impact of hyperparameter setup on SSL performance.

More extensive models often improve SSL performance than smaller models, particularly with a smaller preprocessing data quantity. However, as preprocessing data amount increases, recurrent models generally reach a performance threshold. On the other hand, increasing data for hyperparameter configurations more tuned for SSL, SSL performance improves for the feed-forward model. Therefore, one should generally experiment using different data volumes and use a smaller data size, if possible, for all models. If aiming to achieve the best possible absolute downstream performance, larger and more sophisticated models should be used.

This thesis explored the impact of hyperparameters, including preprocessing data size and model size, on an SSL technique for a text classification task. This exploration improves understanding of which parameters have the most impact on SSL for text classification, making it more manageable to perform SSL work for future NLP projects, particularly if time-restricted. This research also advances the understanding of model size impact on SSL for text classification, enabling a better experience of model selection for SSL designs. With this, we enhance the knowledge of parameter relations, potentially lowering the training time for SSL-based machine learning.

5.2 Future Work

There are multiple interesting potential paths for future work. These include:

- Using other fixed or variable pre-training sequence lengths, for example, variable-length sentences. This way, preprocessing data is more similar to downstream data, potentially improving SSL performance.
- Utilizing other numbers of randomly masked tokens in each input sequence from preprocessing data.
- Not using the eleventh word after each 10-word sequence during tokenizer creation.
- Without creating a testing dataset, allowing additional preprocessing data for training and validation.
- Using additional epochs during pre-training, allowing models to converge more during pre-training.

- With other pretext tasks, for example predicting input sequence or reordering a shuffled sequence.
- Modifying used pretext task of predicting masked tokens. For example: Using inspiration from BERT [29], meaning to for example 10% of time replace a selected masked token by a random vocabulary token, and 10% of time leave token unchanged.
- Utilizing additional pre-training data.
- With other hyperparameter configurations. For example, creating other configurations using known hyperparameter tuning methods (for instance grid search or random search) or tuning with additional preprocessing data, potentially improves SSL performance.
- Using all 20 newsgroups to generate preprocessing datasets from the 20newsgroups dataset. Preprocessing datasets are more varied then, potentially improving SSL performance.

Also, running experiments multiple times to extract additional SSL performance distributions could change the thesis conclusion.

References

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[2] A Agrawala. Learning with a probabilistic teacher. *IEEE Transactions on Information Theory*, 16(4):373–379, 1970.

[3] aladdinpersson. seq2seq_attention. `https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/more_advanced/Seq2Seq_attention/seq2seq_attention.py`, 2021. [Online; accessed 3-June-2021].

[4] Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W Cottrell, and Julian McAuley. Rezero is all you need: Fast convergence at large depth. *arXiv preprint arXiv:2003.04887*, 2020.

[5] Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. Cloze-driven pretraining of self-attention networks. *arXiv preprint arXiv:1903.07785*, 2019.

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

[7] Jonathan Baxter. A model of inductive bias learning. *Journal of artificial intelligence research*, 12:149–198, 2000.

[8] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can

language models be too big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021.

[9] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. In *Advances in Neural Information Processing Systems*, pages 932–938, 2001.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[11] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[12] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[13] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[14] Susan Carey and Elsa Bartlett. Acquiring a single new word. 1978.

[15] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[16] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[17] Inigo Casanueva, Tadas Temčinas, Daniela Gerz, Matthew Henderson, and Ivan Vulić. Efficient intent detection with dual sentence encoders. *arXiv preprint arXiv:2003.04807*, 2020.

[18] Hugo Caselles-Dupré, Florian Lesaint, and Jimena Royo-Letelier. Word2vec applied to recommendation: Hyperparameters matter. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 352–356, 2018.

[19] chao ji. Semi-supervised-sequence-learning. https://github.com/chao-ji/semi-supervised-sequence-learning, 2021. [Online; accessed 3-June-2021].

[20] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-Supervised Learning.* 09 2006.

[21] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[22] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[23] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[24] colah. Neural networks, types, and functional programming. http://colah.github.io/posts/2015-09-NN-Types-FP/, 2015. [Online; accessed 3-June-2021].

[25] Michael Collins and Yoram Singer. Unsupervised models for named entity classification. In *1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, 1999.

[26] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. *arXiv preprint arXiv:1511.01432*, 2015.

[27] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[28] David DeMers and Garrison W Cottrell. Non-linear dimensionality reduction. In *Advances in neural information processing systems*, pages 580–587. Citeseer, 1993.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[30] Pranay Dugar. Attention — seq2seq models. https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263, 2019. [Online; accessed 3-June-2021].

[31] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[32] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[33] S Fralick. Learning to recognize patterns without a teacher. *IEEE Transactions on Information Theory*, 13(1):57–64, 1967.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[35] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[36] Herman O Hartley and JNK Rao. A new estimation theory for sample surveys. *Biometrika*, 55(3):547–557, 1968.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

[39] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

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

[41] David W Hosmer Jr. A comparison of iterative maximum likelihood estimates of the parameters of a mixture of two normal distributions under three different types of sample. *Biometrics*, pages 761–770, 1973.

[42] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[43] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[44] Longlong Jing and Yingli Tian. Self-supervised visual feature learning with deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[45] Thorsten Joachims et al. Transductive inference for text classification using support vector machines. In *Icml*, volume 99, pages 200–209, 1999.

[46] Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.

[47] Simeon Kostadinov. Understanding gru networks. https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be, 2017. [Online; accessed 3-June-2021].

[48] Simeon Kostadinov. Understanding encoder-decoder sequence to sequence model. https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346, 2019. [Online; accessed 3-June-2021].

[49] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[50] Ken Lang. Newsweeder: Learning to filter netnews. In *Machine Learning Proceedings 1995*, pages 331–339. Elsevier, 1995.

[51] Dongjun Lee. Transfer learning for text classification with tensorflow. https://github.com/dongjun-Lee/transfer-learning-text-tf, 2018. [Online; accessed 3-June-2021].

[52] Yu-Feng Li and Zhi-Hua Zhou. Towards making unlabeled data never hurt. *IEEE transactions on pattern analysis and machine intelligence*, 37(1):175–188, 2014.

[53] Henrik Lien and Bjørn V. Ledaal. Semi-supervised learning for text classification. Student project in the course IKT442: ICT Seminar 3 at UiA, 2020.

[54] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[55] Andrea Manero-Bastin. How to configure the number of layers and nodes in a neural network. https://www.datasciencecentral.com/profiles/blogs/how-to-configure-the-number-of-layers-and-nodes-in-a-neural, 2019. [Online; accessed 3-June-2021].

[56] Huanru Henry Mao. A survey on self-supervised pre-training for sequential transfer learning in neural networks. *arXiv preprint arXiv:2007.00800*, 2020.

[57] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[58] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. *arXiv preprint arXiv:1708.00107*, 2017.

[59] Geoffrey J McLachlan and S Ganesalingam. Updating a discriminant function on the basis of unclassified data. *Communications in Statistics-Simulation and Computation*, 11(6):753–767, 1982.

[60] Geoffrey John McLachlan. Estimating the linear discriminant function from initial samples containing a small number of unclassified observations. *Journal of the American statistical association*, 72(358):403–406, 1977.

[61] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[62] Aditi Mittal. Understanding rnn and lstm. https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e, 2019. [Online; accessed 3-June-2021].

[63] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[64] Avital Oliver, Augustus Odena, Colin Raffel, Ekin D Cubuk, and Ian J Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. *arXiv preprint arXiv:1804.09170*, 2018.

[65] Terence J O'neill. Normal discrimination with unclassified observations. *Journal of the American Statistical Association*, 73(364):821–826, 1978.

[66] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[67] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[68] Steven Pinker. *Learnability and Cognition, new edition: The Acquisition of Argument Structure.* MIT press, 2013.

[69] The Glowing Python. The perceptron. https://glowingpython.blogspot.com/2011/10/perceptron.html, 2011. [Online; accessed 3-June-2021].

[70] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[71] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[72] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[73] Sebastian Ruder. *Neural transfer learning for natural language processing.* PhD thesis, NUI Galway, 2019.

[74] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of*

the Association for Computational Linguistics: Tutorials, pages 15–18, 2019.

[75] Henry Scudder. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3):363–371, 1965.

[76] Aarti Singh, Robert Nowak, and Jerry Zhu. Unlabeled data: Now it helps, now it doesn't. *Advances in neural information processing systems*, 21:1513–1520, 2008.

[77] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[78] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.

[79] Vladimir Naumovich Vapnik and AM Sterin. Controlled minimization of the total risk in pattern recognition. *Avtomatika i Telemekhanika*, (10):83–92, 1977.

[80] VN Vapnik and A Ya Chervonenkis. The method of ordered risk minimization, i. *Avtomatika i Telemekhanika*, 8:21–30, 1974.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[82] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[83] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.

[84] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[85] Richard Zhang, Phillip Isola, and Alexei A Efros. Split-brain autoencoders: Unsupervised learning by cross-channel prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1058–1067, 2017.

[86] Xiaojin Jerry Zhu. Semi-supervised learning literature survey. 2005.