Master of Science in Computer Science

Thesis

Featherweight NLP: Tolerable Lower Limits for Size of Distilled Transformer Language Models

Mikkel Hooge Sørensen
mhso@itu.dk

Magnus Malthe Jacobsen
mjac@itu.dk

Supervised by
Leon Derczynski

September 1, 2021

IT UNIVERSITY OF COPENHAGEN

Abstract

This thesis investigates compression of machine learning models within NLP. To examine the extent to which models can be compressed, we conduct experiments with knowledge distillation, pruning, and quantization. We distill the behavior from the RoBERTa language model to Long Short Term Memory (LSTM), Recurrent Neural Network, and Feed-Forward Network models. For each, we experiment with different embeddings types and sizes. We conduct experiments on three tasks from the General Language Understanding benchmarks (GLUE). Stanford Sentiment Treebank (SST-2), Quora Question Pairs (QQP), and Multi Natural Language Inference (MNLI).

Results show that using task-specific distillation allows for small models. We compare the performance of compressed models to baseline models from the GLUE benchmark set. One distilled and quantized bidirectional LSTM model surpasses the baseline performance presented by GLUE. Compared to the size of the RoBERTa teacher, this model achieves size reductions of 3528x for SST-2 and 963x for QQP and MNLI.

Contents

List of Figures

List of Tables

1 Introduction

A fundamental goal of data compression is to obtain the best possible fidelity for the given rate or, equivalently, to minimize the rate required for a given fidelity. (Gray, 1984)

Language Models are machine learning models trained to "understand" patterns in languages. For recent neural network language models, these patterns are general enough that, with a little extra task-specific fine-tuning, they achieve state-of-the-art performance for a variety of Natural Language Processing (NLP) tasks. Encoding such patterns result in increasingly larger architectures, for instance transformer-based architectures such as BERT (Devlin et al., 2019), GPT-2 (Radford et al., 2019), and Switch-C (Fedus et al., 2021). Recent frontrunners in the quest for better performance are boasting parameter counts in billions and trillions.

Using a large neural network language model as a baseline comes with a cost. During training and inference alike, more parameters equals more computations. This increases energy consumption and hardware requirements. These large models are therefore not an option for resource-constrained devices. Researchers are exploring ways to compress or distill language models for such environments. Little research has gone into what the lower bounds are for the size of language models when relaxing the requirement for model performance.

In this thesis we examine the lower limits of size for compressed models derived from language models across a set of NLP benchmark tasks on the English language. Meanwhile, we relax the common requirement of keeping fidelity parity between the original and compressed model, and instead opt for a tolerable fidelity.

We apply different model compression schemes to small task-specific models with learned behavior from pre-trained transformer-based Language Models. With this, we investigate the following two questions:

RQ1: What is the relationship between model performance and size complexity when the latter is reduced?

RQ2: To what extent can models, fine-tuned for different GLUE benchmark tasks, be compressed while maintaining tolerable performance?

Concretely, we investigate neural network model compression for three tasks from the General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2019): Stanford Sentiment Treebank 2 (SST-2), Quora Question Pairs (QQP), and Multi-Genre Natural Language Inference (MNLI). SST-2 is a single sentence sentiment classification task; QQP is a sentence-pair similarity and paraphrase classification task; finally, MNLI is a sentence-pair classification task that falls under the label Natural Language Inference.

The GLUE paper presented benchmark results for a variety of BiLSTM baseline models. At the time of its release in 2019, the BERT inspired Transformer language model RoBERTa (Liu et al., 2019) not only outperformed its inspiration on all nine GLUE benchmark tasks, it also achieved new state-of-the-art results for four of the nine tasks. We intend to use the pre-trained RoBERTa language model as a basis for our investigation.

We use the task-specific baseline models presented in the GLUE paper as reference for tolerable fidelity of our compressed models. While these baselines are not state-of-the-art, they were used as examples of how well machine learning models were able to learn patterns from a variety of tasks within natural language understanding.

Given our goal of minimizing size, we want to use the neural network model compression techniques with the largest compression ratios. As such we will look into knowledge distillation from a fine-tuned RoBERTa teacher to smaller and simpler student models. We intend to further compress these with pruning and quantization, to examine how small they can get, while maintaining GLUE baseline performance parity.

In chapter 2, we describe fundamental concepts within natural language and language models. In chapter 3, we describe concepts within machine learning that are necessary for understanding the work done in this thesis. In chapter 4 we describe concepts related to compression of machine learning models. Chapter 5 outlines similar work to ours, done by researchers within and outside the NLP field. Chapter 6 describes our setup and strategy for our model compression experiments. In chapter 7, we describe and analyze the results of our experiments. In chapter 8, we describe what the impact of our work is in the context of machine learning research. Finally, chapter 9 concludes and summarizes the findings of this thesis.

2 Understanding Natural Language

Natural language is the linguistic communication that have evolved among humans, whether it be spoken, signed, or written. A central aspect of natural language is meaning. As American philosopher dealing with mind and language Searle (1998, p. 140) describes it:

> Sentences and words have meanings as parts of a language. The meaning of a sentence is determined by the words and the syntactical arrangements of words in the sentence.

Unlike formal languages, e.g. programming languages and logic, the meaning of natural language utterances can be ambiguous. However, meaning and the degree of ambiguity is constrained by conventions of the specific language being uttered.

Natural language processing is concerned with creating computer programs that can process and analyze words and sentences from natural language for a wide array of problems. These problems include, but are not limited to, speech recognition, translation from a language to another, and question answering.

2.1 Language Modeling

A fundamental constituent of NLP is language models. A language model is a tool that learns patterns from human language, and in return is able to provide probabilities for strings in the language.

A common definition of a language model states that it has to contain a vocabulary, V, consisting of a finite set of tokens and a probability function, $p(x_1, x_2 ... x_n)$ that accepts any sequence of tokens (Collins, 2013). Often tokens are equated with words for a given language. For English a vocabulary could look like this:

$$V \coloneqq \{\, \text{the}, quick, brown, fox, jumps, ... \,\}$$

A sequence must contain at least one token, and tokens must be from the vocabulary V. Since there is no upper limit for the amount of tokens in a sequence, the set of all sequences $V^\dagger$ is infinite.

The probability function $p(x_1, x_2 ... x_n)$ has to accept any sequence, and provide a probability larger than or equal to 0 for it. The sum of probabilities for all sentences must equal 1. Thus the function is a probability distribution over sentences in $V^\dagger$.

Historically, language models were developed and used for the task of speech recognition, but not exclusively. A neat feature about a language model is that given an incomplete sentence with an unknown token, it only takes at most n runs of the probability function to find the most probable sentence. Here n is the amount of tokens in V. If one can limit the amount of candidates for the unknown token to k different tokens, then one only has to run the function k times.

2.1.1 Statistical Language Models

Creating a language model that uses the relative frequencies of specific sequences of tokens for a language is an infeasible task in terms of required training data. Instead, the classical approach in NLP has been to use n-grams models which estimate sequence probabilities by simplifying the problem. In the n-grams model, each token, along with the $n-1$ previous tokens, are treated as a single probability. The probability for a sequence is the product of these n-gram probabilities. To circumvent unseen n-grams getting a probability of 0, different techniques for smoothing are used (Collins, 2013).

2.1.2 Neural Language Models

Using neural networks as language models have gained traction from the early 2000s (Bengio et al., 2003). Varied network architectures such as Feed-Forward Network (FFN) (Wasserblat et al., 2020), Convolutional Neural Network (CNN) (Kim, 2014), Recurrent Neural Network (RNN) (Socher et al., 2013), Long-Short Term Memory (LSTM) (Wang et al., 2019), and most recently Attention-based Transformer networks, have been applied (Vaswani et al., 2017).

Especially GPT-2 (Radford et al., 2019) and BERT (Devlin et al., 2019) have received wide attention in the past two-three years. Common to both of them is the use of a transformer network architecture and a two-stage approach. The first stage concerns pre-training the model as a language model on a large unlabelled dataset. GPT-2 trains with the goal of iteratively predicting the next word in a sentence; BERT trains predicting a randomly masked word in an otherwise complete sentence. In the second stage, they add additional parameters at the end of the pre-model relevant to the task being solved, and then fine-tune for it.

BERT has inspired others to create models with a similar approach. In 2019 Liu et al. (2019) presented the RoBERTa model, which improves on the performance of BERT.

One caveat with this approach, is that the pre-trained model has to be large, and it has to train for a long time on a large dataset. The large BERT variant contains 340 million parameters, while GPT-2 boasts 1.5 billion. The benefit is that a good pre-trained model only has to be trained once and can then be used for fine-tuning to many tasks. Furthermore, fine-tuning is relatively inexpensive in terms of training time. Last but not least, BERT scored the highest on eight out of nine tasks in the General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2019).

2.2 What is understood?

In a blog post Vice President of Search at Google, Nayak (2019) claims:

> BERT models can therefore consider the full context of a word by looking at the words that come before and after it—particularly useful for understanding the intent behind search queries.

Researchers Bender and Koller (2020) are critical of these types of claims: That modern neural network language models understand human communicative meaning and intent. Although the specific papers from the teams behind GPT-2 and BERT are devoid of these type of claims, Bender and Koller expose how many other research articles from this scientific field are not. Furthermore, they show how news press often translate this to mean that such machine learning models understand how and what humans communicate.

According to Searle (1998, p. 139), “Language relates to reality in virtue of meaning”. Furthermore, he states that meaning is related to the intention of the one uttering in a communication act:

> The key to understanding meaning is this: meaning is a form of derived intentionality. The original or intrinsic intentionality of a speaker’s thought is transferred to words, sentences, marks, symbols, and so on. If uttered meaningfully, those words, sentences, marks, and symbols now have intentionality derived from the speaker’s thoughts. They have not just conventional linguistic meaning but intended speaker meaning as well (Searle, 1998, p. 141).

Bender and Koller (2020) shares Searle’s views on language, understanding, meaning and intention. They claim that since machine learning models, like GPT-2 and BERT, are only trained and optimized on linguistic form, they do not understand natural language. This is also true for models that encode deep relationships between vocabulary tokens from a large corpora. Such a model might convince a human that it is human too, i.e passing the Turing test. At the end of the day, it is not *understanding* human communicative intent and how the meaning of utterances relate to reality. Current language models are only able to understand the part of natural language that relates to linguistic form, not the part of meaning.

2.3 NLP Tasks

NLP is a rich field with many problems of varying complexity. It is perhaps not surprising that analyzing and extracting value from human language is valuable. For this thesis, we limit our scope of tasks to those dealing with written language.

Within NLP, tasks vary on what input they receive and what output they provide. In classification tasks, a model is given some text and has to provide a label to this text. An example is sentiment analysis, where the model has to decide whether a sentence carries a positive or negative sentiment. Input to a task may also be several sentences, where the model has to decide how the sentences are related. Such a task would be classified as a sentence-pair task. On the other hand, with a single-sentence task, the model has to infer something from only one sentence. An example of a sentence-pair task is natural language inference. In this type of task, a model has to decide whether two sentences show entailment, contradiction, or are unrelated.

Other NLP tasks with different input and output structure exist, but these are beyond the scope of this thesis.

The General Language Understanding Evaluation benchmark (GLUE) is a collection of NLP tasks that seek to challenge researchers on various aspects of *natural language understanding* (Wang et al., 2019). GLUE seeks to provide high quality data for the English language, as well as standardized testing methods, across a variety of NLP problems. GLUE categorizes tasks as either single-sentence, similarity and paraphrase, or inference tasks. The tasks are designed to test models on many aspects within natural language. The aim of GLUE is to move further towards automated natural language understanding.

3 Neural Networks for NLP

In recent years, neural networks have become a popular approach for solving problems using machine learning.

The use of neural networks has saturated many areas of NLP research. In this section, we will first outline some of the fundamentals of how neural networks function. We will then describe variants of neural networks that have been made to work better with data related to natural language. Finally, we will discuss the size of some selected neural network models designed for NLP tasks.

3.1 Computational Flow

At a most basic level, a neural network works in the following way: numerical input is fed to a network, a series of mathematical operations with parameters act on these numbers, and a number, or a series of numbers, is the output. During training, the parameters in the network are adjusted to minimize the error between the output and a given base truth.

The most basic building block of a neural network is the perceptron. The perceptron was invented by Rosenblatt (1957). A perceptron receives input and provides an output. Input can be the output of another perceptron. In this case we are dealing with a Multilayer Perceptron (MLP). The perceptron stores a connection weight $w_j \in \mathbb{R}$ for each input $x_j \in \mathbb{R}, j = 1, ..., d$. Its output is a weighted sum of the inputs:

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \tag{3.1}$$

w_0 is the bias which corresponds to an extra constant input value x_0 which is always 1 (Alpaydin, 2014, p. 271). Using matrix multiplication, this can be modelled as a dot product:

$$y = \boldsymbol{w}^T \boldsymbol{x} \tag{3.2}$$

where $\boldsymbol{w} = [w_0, w_1, ..., w_d]^T$ and $\boldsymbol{x} = [1, x_1, ..., x_d]^T$ are augmented vectors that include the inputs, weights, and bias. Each pair of the input x_i and connection weight w_i defines a neuron. This type of network, where the information flows one way through the network, is also called a Feed Forward Network (FFN).

Equation 3.2 defines a hyperplane that can be used to separate inputs and correctly assign them one of two classes. This is called binary classification. Learning is done by adjusting the individual weights until the network can correctly classify the inputs.

A single perceptron is not particularly powerful. For instance, such a perceptron is incapable of representing the XOR function (Alpaydin, 2014, p. 277). However, multiple perceptrons used in sequence in a multi-layer perceptron are capable of representing arbitrarily complex problems (Cybenko, 1989). In an MLP, the output of a perceptron is the input to the next perceptron. Each layer has connection weights and a bias corresponding to each input.

3.2 Activation Functions

The outputs of individual perceptrons can theoretically take on any numerical value. This is acceptable for some applications, but for others it is not. For instance, sometimes the output of the network needs to lie within a certain numerical range. This may also be the case for the output of intermediate layers. Activation functions are a way of achieving this.

Activation functions also help a network to model non-linear relationships in data (Alpaydin, 2014, p. 280). An activation function can be seen as performing a non-linear transformation on data in d dimensions to a new H-dimensional space. H is determined by how many units the activation function acts on. If the activation function acts on intermediate layers in the network, the neurons after this point can be seen as implementing linear transformations in this new H-dimensional space. These are all ways of doing more flexible modelling of the given input data.

Sigmoid. A network modelling probability distributions would require its outputs to lie in the range $(0, 1)$. To do this, such a network could utilize the logistic function, known as the the sigmoid activation function (PyTorch, 2019i). It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

This function defines a smooth "S-curve" between 0 and 1, which is useful for modelling probabilities.

Softmax. This function is a generalization of the logistic function for multi-dimensional data (PyTorch, 2019j). Softmax is defined as:

$$\sigma(\boldsymbol{x})_i = \frac{e^{x_i/T}}{\sum_{j=1}^{K} e^{x_j/T}} \tag{3.4}$$

where T is called the temperature, which controls the spread of the resulting distribution. Usually, T is left at 1 which defines the unit softmax function.

The output values of a network, before sigmoid or softmax is applied, are called the logits. After sigmoid or softmax, the logits are transformed into a probability density over the output classes.

Tanh. A similar function to sigmoid is the hyperbolic tangent function, or simply tanh (PyTorch, 2019l). It defines a similar S-shape as sigmoid, but ranges from -1 to 1 instead of 0 to 1. It is defined as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.5}$$

ReLU. Another widely used activation function is the Rectified Linear Unit (ReLU)(PyTorch, 2019g). It is defined as:

$$ReLU(x) = max(0, x) \tag{3.6}$$

ReLU essentially ensures that output values are always positive. ReLU is very simple, which makes it very fast to compute.

GELU. The final activation function that we will cover is the Gaussian Error Linear Unit (GELU) (Hendrycks and Gimpel, 2020). It is defined as:

$$GELU(x) = xP(X \leq x) = x\Phi(x) \tag{3.7}$$

and can be approximated as (Hendrycks and Gimpel, 2020):

$$0.5x(1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)]) \tag{3.8}$$

GELU aims to combine the features of ReLU and Dropout. Dropout is not strictly an activation function. It is instead a layer that stochastically sets a ratio of its input values to 0. This helps the network adapt to new data by forcing connections in the network to update constantly to account for the dropout. GELU combines the stochastic zeroing of weights used in dropout with the deterministic activation of ReLU.

3.3 Loss Functions

In order to learn patterns in data, a network needs a measurement of error. The network needs to be able to measure how wrong it is when predicting on training data, in order to adjust its weights to provide better future predictions. Loss functions provide ways of doing so.

As with activation functions, a variety of loss functions exist that each serve a different purpose. We will only cover two of these: Mean Squared Error (MSE) and Cross Entropy (CE). These are two of the most common in machine learning and are the ones used in models described in later sections.

Mean Squared Error. MSE takes the prediction output of the neural network and compares it with the actual labels in the training data (PyTorch, 2019d). It is defined as:

$$MSE(\boldsymbol{y}, \boldsymbol{t}) = \frac{1}{N} \sum_{i=1}^{n} (y_i - t_i)^2 \tag{3.9}$$

Here, $\boldsymbol{y}$ is a vector of the predicted labels outputted by the network and $\boldsymbol{t}$ are the actual target labels. This loss function is often used when the outputs of a network are real values, for example with regression problems.

Cross Entropy. When the outputs are labels, the CE loss function is better suited (PyTorch, 2019b). The loss in relation to the target label is defined as:

$$CE(x, class) = -x[class] + log(\sum_j exp(x[j])) \tag{3.10}$$

This function assumes that inputs are not normalized. Here, $x[i]$ is the output value of x corresponding to the class i. If this value is high, the network believes x should be classified as class i and vice-versa. The loss then takes into account how certain the network is that x belongs to *class*. If the network outputs similar values for all classes, it is unsure of what class x should be in and the loss will be high. If the output value corresponding to the target class is high, but all other values are low, the loss will be low.

3.4 Optimization

Now that the network can measure its error during prediction, the next step is to learn from this error. We wish to find the weights that minimize error on the training data. More formally, we wish to find $\boldsymbol{w}^*$ such that:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, E(\boldsymbol{w}|X) \tag{3.11}$$

Where $\boldsymbol{w}^*$ are the optimized weights, that minimize the error E when applied to input X.

Weights are initialized to small random values, generally in the range $[-0.1, 0.1]$. This can be sampled from a uniform, normal, or similar distribution. Weights values are kept close to 0, because larger values may cause problems when using activation functions such as sigmoid or tanh. Since the output of these functions lie within a limited numerical range, too much information may be lost when weights have large values (Alpaydin, 2014, p. 285).

3.4.1 Gradient Descent

A widely used method for optimization is gradient descent. The idea is to take the partial derivatives, ∂, of the loss with respect to each weight in the network. This forms a gradient vector:

$$\nabla_w E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_d}\right]^T \tag{3.12}$$

Each entry in this vector indicates how much each weight in the network needs to be nudged in order to improve prediction accuracy (Alpaydin, 2014, p. 249). Gradient descent then adjusts the corresponding weight in the opposite direction of the gradient. Otherwise we maximize error, instead of minimizing it. The update of each weight in the network is defined as:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \forall i \tag{3.13}$$

$$w_i = w_i + \Delta w_i \tag{3.14}$$

where η is the learning rate, which dictates how much the weight adjustments are scaled (Alpaydin, 2014, p. 249). These equations can be generalized to any number of classes by using the relevant loss function for measuring E. If the learning rate is too high, gradient descent might never find a local minimum where loss is low. If it is too low, learning will be slower than necessary. See figure 3.1 and figure 3.2 for an illustration of this.

In practice, it is often too computationally expensive to calculate the gradient for every input sample in the training set in one go. This would also assume that the data is static. In many applications, data is changing or continually arriving for processing. An example would be an ap-

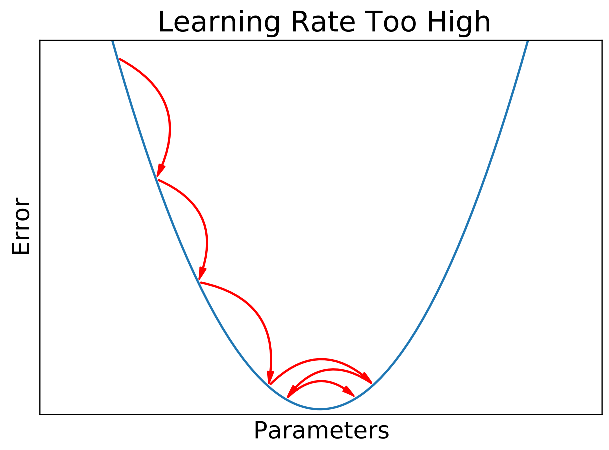

Figure 3.1: Illustration of what happens when learning rate is too high during gradient descent. Error may "bounce around" the minimum, without reaching it.

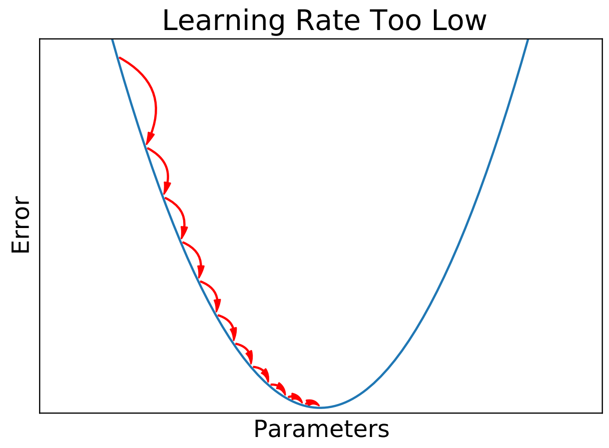

Figure 3.2: Illustration of what happens when learning rate is too low during gradient descent. Error decreases very slowly, which is inefficient.

plication that recommends music based on what the user has recently listened to.

Instead of *offline* learning, where learning happens in a static fashion, *online* learning is more dynamic. In online learning, the network is trained on a random subset of the data for each learning step. The corresponding subsets of weights are then updated, and another subset of data can be selected for training (Alpaydin, 2014, p. 275).

This is called stochastic gradient descent (SGD) because of the random nature of sampling the training subset. The update rule of SGD is very similar to the offline learning case. The gradient is calculated across a subset of the data of size b. Updating the randomly sampled subset is defined as:

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j}, j = 1, \ldots, b \tag{3.15}$$

$$w_j = w_j + \Delta w_j \tag{3.16}$$

The training is then done in batches and b is known as the batch size.

3.4.2 Backpropagation

One issue with gradient descent is that for an MLP, the input to all hidden and output layers is the output of a previous layer. This means that the weights and gradients of later layers are dependent on the previous. Backpropagation solves this by using the chain rule propagate the error backwards in the network. To do so, the following equation is used:

$$\frac{\partial E}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \cdots \cdot \frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial x} \tag{3.17}$$

where z^L and a^L are the output and activation function, respectively, at layer L (Alpaydin, 2014, p. 283). To calculate gradients for each layer, backpropagation starts at the last layer and calculates the loss with respect to the final activation. The derivative of this is then used to calculate the gradient of the layer output. This is in turn used to update the previous layer and so on. This continues until the input layer is reached and all the weights have been updated. Backpropagation has been independently discovered several times in the past. In the context of neural networks, it was first presented by Rumelhart et al. (1986a).

Backpropagation requires that activation functions are differentiable. For some activation functions, this is not strictly the case. For example, ReLU is undefined for input $x = 0$. As such, we must make an exception when calculating derivatives for ReLU at $x = 0$. Simply setting the derivative to 0 or 1 in this case circumvents this problem.

Backpropagation using the chain rule has a few potential downsides. Gradients for activation functions are often in the range $(0, 1)$ or

$(-1, 1)$, as is the case with the sigmoid and hyperbolic tangent functions. Since gradients are accumulated using the chain rule via. multiplication, gradient updates often become too small or completely lost if the backpropagation sequence is long enough. This is called the vanishing gradients problem. This issue was first made clear by Hochreiter in 1991, and further elaborated by him in 1998 (Hochreiter, 1998). This issue occurs if networks are too deep, with too many layers relying on gradient updates from previous layers.

The opposite of the vanishing gradient issue is the exploding gradient issue. This happens when gradient values during backpropagation are too large. This may happen with deep networks, that is networks that have many layers stacked. With a long enough backpropagation sequence, the gradient values may grow out of proportion because of the many multiplications during the chain rule. This will prevent the network from further learning.

3.4.3 Extensions to SGD

Stochastic gradient descent works well enough on its own. However, a couple shortcomings has since been improved upon. The main shortcoming of SGD is the static learning rate. When SGD is close to finding a local minimum during the optimization procedure, a static learning rate might prove problematic. SGD may "bounce around" at this minimum, without actually reaching it.

SGD with momentum improves upon this. It was conceived by Rumelhart et al. (1986a) in the same paper where they presented backpropagation. Instead of simply calculating gradients over the current batch, momentum uses information from previous batches. It is defined as:

$$V_t = \beta V_{t-1} + \eta \nabla_w E_w \tag{3.18}$$

where β is a hyperparameter that controls how much previous gradients should count toward the current gradient V_t. With lower values for β, older gradients are "forgotten" faster. η is the learning rate and E_w is the error given to w by the loss function. This method helps smooth out the optimization procedure, and should allow SGD to converge better.

AdaGrad. A further improvement was made by Duchi et al. (2011), which they labelled the Adaptive Gradient algorithm (AdaGrad). Similar to SGD with momentum, AdaGrad scales the learning rate according to past gradients. However, this is done separately across each dimension of the gradient vector. If one dimension nears a minimum, the learning rate for that dimension can be lowered to allow for better convergence.

The learning rule of AdaGrad is defined as:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\epsilon I + diag(G_t)}} g_t \tag{3.19}$$

where η is the learning rate and ϵ is a value close to zero, to avoid division by zero. I is the identity matrix, g_t is the gradient at time step t, given by:

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_w E_w \tag{3.20}$$

G_t in equation 3.19 is the core of AdaGrad. This is the sum of the outer product of gradients up to time step t. It is defined by:

$$G_t = \sum_{i=1}^{t} g_i g_i^T \tag{3.21}$$

The function uses $diag(G_t)$ because calculating the square root of the full matrix G_t is expensive. This means that the learning rate is effectively scaled by the square of previous gradient steps across all dimensions.

AdaDelta. The next optimization improvement that we investigate is called AdaDelta Zeiler (2012). This builds upon the principles of AdaGrad, but with one clear improvement. AdaGrad would continually reduce the learning rate throughout training. This meant that given enough time, training might stop completely because the learning rate would get so small. AdaDelta alleviates this by not scaling the learning rate by all past gradients. Only a limited window of recent gradient values are used. This ensures that the network can still learn and adjust the learning rate far into training.

AdaDelta does not store the past gradients. Instead, an average of past gradients is calculated recursively using the formula:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{3.22}$$

where γ determines how heavily the previous average is weighted against the current gradient. γ is usually set to around 0.9. g_t is again the gradient at time step t. The update rule of AdaDelta looks similar to AdaGrad and takes the form:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{3.23}$$

where η is the learning rate and ϵ is a small value to avoid division by zero.

Adam. The final optimization technique that we will describe is called Adam (Kingma and Ba, 2015). Adam is similar to AdaGrad and AdaDelta in that it calculates a moving average of past gradients. Adam computes estimates of past gradients in the first and second moment. The first moment is defined by:

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)g_t \tag{3.24}$$

The second moment is defined by:

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)g_t^2 \tag{3.25}$$

β_1 and β_2 controls how fast the estimates for the first and second moment decays, for each dimension. g_t is the gradient vector at the current time step. Initially, m_0 and v_0 are filled with zeros.

Next, bias corrected moment estimates are calculated:

$$\hat{m}_t = m_t/(1-\beta_1^t) \tag{3.26}$$

$$\hat{v}_t = v_t/(1-\beta_2^t) \tag{3.27}$$

Finally, parameters are updated by:

$$w_t = w_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t + \epsilon}) \tag{3.28}$$

Where α is the learning rate and ϵ is a small non-zero value to avoid division by zero.

Adam improves upon AdaDelta by including the bias correction term when estimating past gradients. The bias correction term improves convergence when gradients are sparse (Kingma and Ba, 2015).

3.5 Representing Text

In NLP the input to models is language utterances in some form. In section 2.1 we defined language as a finite set of words or tokens, the vocabulary, and an infinite set of possible sequences of elements from this vocabulary. In section 3.6 we will go into more detail about how neural network-based NLP models deal with the sequence aspect. Here we will mostly focus on how to represent the vocabulary.

Early on in language modeling words were treated as atomic units and individual words were dealt with as an index mapped to a word in the vocabulary (Mikolov et al., 2013). If one wanted to create a N-gram language model with a co-occurence matrix, the complexity would quickly explode when the vocabulary increased. Another issue with this type of approach was that a lot of textual data would be needed for the co-occurrence matrix to be useful. Even still, it would most likely contain a majority of N-grams with no co-occurences (Mikolov et al., 2013).

Likewise, if one would feed a neural network simple one-hot encoding vectors for each word in a sequence, the length of each individual one-hot encoding vectors would be equal to the vocabulary size. I.e. model and input complexity would quickly increase with the vocabulary size.

Dating back to the 1980s some researchers began to treat words not as atomic units, but instead as positions in a fixed-dimension continuous vector space. Here, each entry in the vocabulary maps to a fixed-length vector. Such vectors are also called embeddings when used in neural networks to transform textual input to input vectors.

3.5.1 Pre-trained Embeddings

Researchers Mikolov et al. (2013) created two very simple but efficient models for unsupervised pre-training of word-based embeddings: Continuous Bag-of-Words (CBOW) and Skip-Gram. Similar to an N-gram model, when looking at a word in the training corpus they include previous words. However, unlike N-gram models, they also look at the next words. The previous and next words are called the context. Unlike N-gram models, CBOW and Skip-Gram does not merely use the context to calculate a probability, but use it to place words in a vector space. The two models differ in how they create the vectors for each word in the vocabulary. The CBOW model use the context to predict a word, while the Skip-Gram model uses a word to predict the context.

Mikolov et al. (2013) went on to show how the words represented in the vector space, to various degree, had encoded information about semantics and syntax. They showed how algebraic operations on the vectors were possible. As an example they took the vector for *king*, subtracted the vector for *man*, and added the vector for *woman*. Among all the other vectors from the vocabulary, the resulting vector from the algebraic operations had the smallest cosine distance to the vector for *queen*. They further showcased how their two models outperformed other neural networks and recurrent neural network (RNN) language models in a Semantic-Syntactic Word Relationship test (Mikolov et al., 2013).

In 2014 researchers from Stanford University published another model for pre-training word embeddings: Global Vectors (GloVe). GloVe combines elements from word-word co-occurence matrix and Skip-Gram. They showed how on many tasks, this approach achieved better accuracy than many other word vector models, among them CBOW and Skip-Gram (Pennington et al., 2014).

3.5.2 Choosing Vocabulary

For a model it can be beneficial to have a large vocabulary. It reduces the likelihood of encountering out-of-vocabulary words when training or doing inference. However, a large vocabulary also increases the total size of the word vectors. Until now, we have dealt with vocabularies on the token level, but this is not always either necessary or the best performing approach.

In a blog post from 2015, the now director of AI at Tesla, Andrej Karpathy showed how a simple multi-layer RNN with character-based vectors could act as a Language Model (Karpathy, 2015). Having a vocabulary of only characters instead of words allows for a drastic reduction in size. This includes both the amount of stored vectors and the mapping of strings to vectors. Even still, the model would be able to handle all possible sequences it might be presented with.

Another approach to deal with rare words, while having a smaller vocabulary, was published by Sennrich et al. (2016). Their approach is called *byte pair encoding* and has a dictionary both of words and sub-word units which then maps to vectors. When an unknown word is encountered, instead of just assigning it a one-fits-all *unknown* vector, it is decomposed into smaller units each mapped to a vector.

There also exists approaches which completely discards the need for storing a dictionary of strings that maps to the embedding vectors. One such method is using a hashing function to decide which vector an input word is mapped to. In a sense the vocabulary has no limit as long as the hashing function can process the input words. What limits the expressability of this approach is how one limits the amount of vectors, and how good the hashing function is. Too few vectors or a poor hashing function can incur collisions where words map to the same vector.

One hashing approach was described by Svenstrup et al. (2017). They used multiple hashing functions for the same input word, which both mapped to scalars and vector components. The final embedding vector for a single word was then constructed by scaling the component vectors and summing them together.

3.6 Architectures for Sequential Data

Neural Networks are good at finding patterns in data. A network trained to recognize handwritten digits from images will be able to do so with a high degree of accuracy. The pixel values for different images of a specific digit will not vary much. Therefore, the inputs the network receives will be similar across multiple images and it will have an easier time learning.

Training networks on textual data is not quite so simple. The naïve approach to doing so would be to use a multi-layer perceptron match each word in a sentence with an input neuron. This would require all sentences in the dataset to be of equal length. This would require transforming sentences in the dataset in some way to fit this length. Then, one would need a way of converting words to numbers for the network to work with. This could be done using a form of embedding as described in section 3.5.

However, this would not necessarily work well. To illustrate why, consider the two sentences: “To succeed, you must first get out of bed” and “In order to succeed, you must first get out of bed”

These sentences are obviously very similar. However, when presented to a network like the one discussed, they would result in different outputs with no discernible relation. This is because the two sentences are offset. For the first sentence, the first neuron in the network will receive data for the word ”To”. It will then output a value based on the weight learned during training. For the second sentence, the first neuron will receive data for the word ”In”. The weights for this neuron is the same, but the input has changed. The output will therefore also change. The error only accumulates as each word is run through the respective neuron in the network. Essentially, the network has no concept of the position of words in a sentence, nor the relationship between words.

3.6.1 Recurrent Neural Network

Recurrent neural networks (RNN) attempt to model sequential data by taking into account the serial nature of such data. The network learns relations by using a recurrence function:

$$h_t = f_W(h_{t-1}, x_t) \tag{3.29}$$

where x_t is the input for time step t in the sequence. W is the weight matrix, and f is a function that acts on the previous hidden state and x_t, to produce the current hidden state. h_t is the hidden vector that helps to determine the output for the current state. The previous state h_{t-1} is taken into account, which is how the network learns the relation between time steps. The weight matrix W is reused for an entire layer. This differs from a feed forward network, where each neuron has an associated weight. The network then has to learn the weights that produce the best result across all hidden vectors.

RNNs have a history that go as far back as the 70s. We will focus on the Elman RNN proposed by Elman (1990). This network is also called a simple RNN. It uses the following function for calculating the hidden state h_t:

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{3.30}$$

where W_h is the weight matrix for the input and U_h is the weight matrix for the hidden states. σ is an activation function that uses these weights to produce the current hidden state. Figure 3.3 shows the structure of an Elman RNN network.

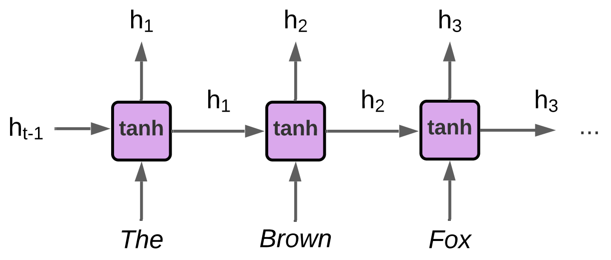

Figure 3.3: Diagram showing the structure of an Elmann RNN.

This network can be said to have a *short term memory* because the hidden states remember what happened in the past.

The final output of an RNN depends on the task it is being used for. For a translation task, the output may be the whole sequence of hidden vectors. These may then be fed into another RNN that decodes the vectors and outputs a probability distribution over words in the target language. This is called an encoder/decoder architecture. One RNN handles the encoding of a sequence, and another handles the decoding.

If the RNN is used for classification, the final output may be the hidden vectors in the last state of the sequence. The idea is that this vector holds the necessary knowledge accumulated across the sequence. Another approach is to take the mean across all the hidden vectors and use this as the output. A softmax function may then be used to provide a classification.

Optimization in a RNN works in much the same way as with a feed forward network. Instead of doing backpropagation across batches, optimization is done across each input sequence. This is called backpropagation through time (Rumelhart et al., 1986b).

While RNNs are much better at dealing with sequential data than simple FFNs, they have issues when propagating gradients back through the network. When sequences get too long, RNNs may face the issue of vanishing gradients as described in 3.4.2. This limits the effectiveness of RNNs. It essentially puts an upper limit on the length of sequences that an RNN can learn from.

3.6.2 Long-Short Term Memory

Long-Short Term Memory (LSTM) networks are variants of RNN designed to alleviate the problem of vanishing gradients. The LSTM was invented by Hochreiter and Schmidhuber (1997). This network works similarly to a regular RNN, but contains cell states that decide which input to pass on to the next state and which to forget.

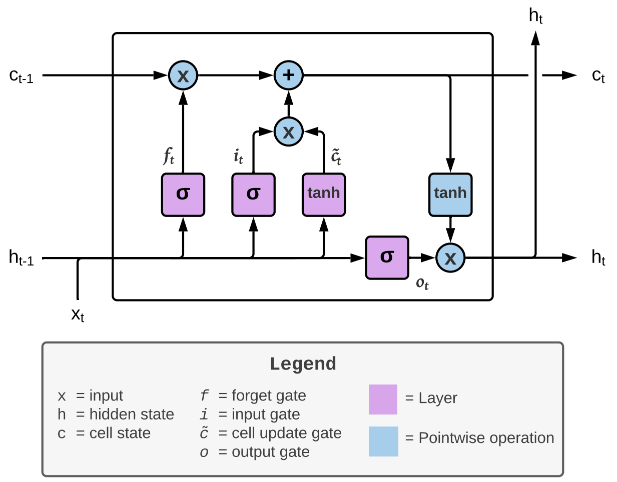

Figure 3.4: Diagram of an LSTM cell with a forget gate.

Figure 3.4 shows the internal state of a single LSTM cell. Similar to a regular RNN, an LSTM cell receives the hidden vector h_{t-1} from the previous time step, and the input x_t from the current time step. In addition to this, the LSTM cell also has an internal state c which is called the cell state. This cell state is determined by a variety of factors, including previous cell state c_{t-1}. The cell state is what separates an LSTM from a regular RNN. It is what allows an LSTM to train on much longer sequences, before experiencing vanishing gradients (Hochreiter and Schmidhuber, 1997).

For the following equations, W_q is the weight matrix for the inputs, U_q is the weight matrix for the hidden vectors, and b_q is the bias term. Subscript q refer to the specific weights for the different parts of the LSTM cell. More specifically, it can refer to: the input gate i, the forget gate f, the output gate o, or the cell state c.

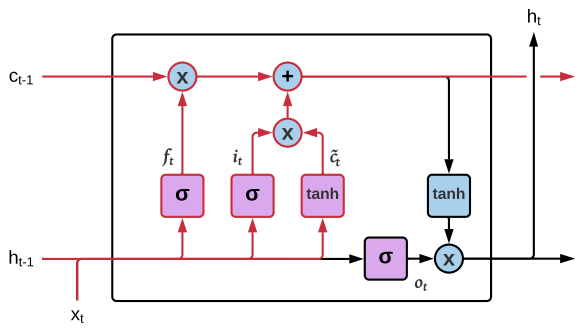

Figure 3.5: Calculations for the c_t state of the LSTM cell.

The first part of calculating the state of the LSTM is to decide what information is discarded. This happens by combining the previous hidden vector h_{t-1} and the current input x_t through a sigmoid function. This is called the forget gate. It is defined as:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{3.31}$$

The next step is to create a vector of candidate cell updates that may be applied to the cell state. This has two steps. First, the data passes through another sigmoid function, called the input gate:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{3.32}$$

The data also passes through a tanh layer to create a candidate vector $\hat{c}_t$ of updates to the cell state:

$$\hat{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{3.33}$$

Then, this is combined along with the forget gate, to create a new cell state for the LSTM cell:

$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t \tag{3.34}$$

where $\circ$ is the *Hadamard product* or element-wise product.

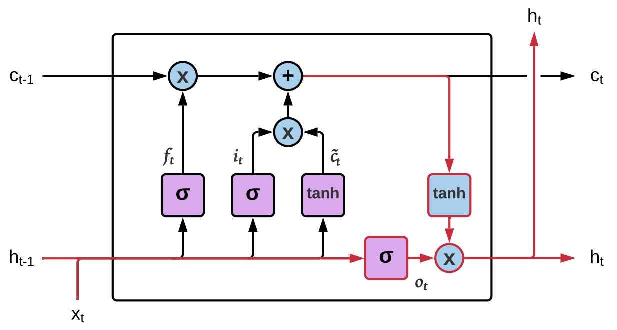

Figure 3.6: Additional calculations for the hidden state h_t and the output of the LSTM cell.

Finally, the LSTM cell calculates its output for the current step. As with a regular RNN, the hidden vector h_t is the output for time step t, and is also used to calculate the values of the next state. The output of the LSTM cell is defined by two equations. First, the data passed through the output gate, which is another sigmoid function:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{3.35}$$

The output is the Hadamard product of this, along with the newly calculated cell state, passed through a tanh function:

$$h_t = o_t \circ tanh(c_t) \tag{3.36}$$

This hidden state is then output by the cell, as well as being passed to the next cell along with the cell state c_t.

One disadvantage of LSTMs and RNNs is their sequential nature. Every state needs to be calculated one at a time since each state feeds into the next. This makes LSTM networks relatively slow when used on long sequences, as the states within them cannot be calculated in parallel.

3.6.3 Transformers

The transformer network tries to model relationships in sequential input in a different way to LSTMs and RNNs. Instead of calculating states sequentially, a transformer uses self-attention to model input and output sequences (Vaswani et al., 2017). Self-attention is a way of measuring how strongly related words in a sentence are to other words. The advantage in comparison to an LSTM is that self-attention can be calculated in parallel across input sequences.

A transformer is made up of a series of encoder and decoder layers. A single encoder/decoder layer is shown in figure 3.7.

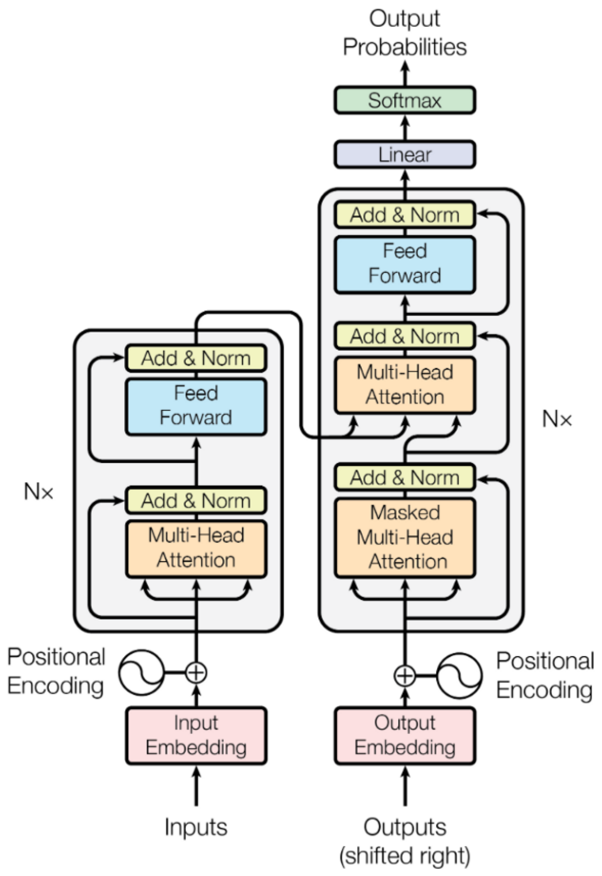

Figure 3.7: Transformer Encoder (left) & Decoder (right). Figure from Vaswani et al. (2017), figure 1.

The encoder receives a representation of the input sentence. For each word, the position of that word within the sentence is added to the input representation vector. The multi-head attention layer is the main ingredient of the transformer. At a high level, attention is a way of mapping a query and a set of key-value pairs to an output. The query Q, keys K, values V, and output are all vectors. The output is a weighted sum of the values, where each weight is a measure of similarity between the query and corresponding key.

There are different ways of calculating attention scores. The one used in transformer networks is called *Scaled Dot-Product Attention* (Vaswani et al., 2017). It is defined as:

$$Attention(Q, K, V) = \sigma(\frac{QK^T}{\sqrt{d_k}})V \tag{3.37}$$

where d_k is the dimension of K, Q. This is a scaling factor to prevent the dot product between Q and K from becoming too large. σ is the softmax function as described in 3.4.

In the case of the transformer, the attention function input vectors are defined as:

$$Q = W_q[a_1, ..., a_k]^T \tag{3.38}$$

$$K = W_k[a_1, ..., a_k]^T \tag{3.39}$$

$$V = W_v[a_1, ..., a_k]^T \tag{3.40}$$

where W_q, W_k, W_v are weight matrices for the query, key, and value vector respectively. a_i is the input representation for token i in the sentence.

This form of attention is called self-attention. The output of 3.37 is a matrix that encodes the attention scores between all the words in the sentence. This score is a measure of how closely related any two words are.

The transformer uses multi-head attention. This means that several attention matrices are computed in parallel and the results are concatenated together. All the concatenated attention matrices are then fed to a feed-forward neural network.

Every layer in both the encoder and decoder has a residual connection (He et al., 2016) and a layer normalization step (Ba et al., 2016).

The residual connection simply adds the inputs and outputs of a layer together. This means the network essentially outputs the change that a layer has on the given input. This is in contrast to simply outputting the result from the operations that the layer implements. In (He et al., 2016), this was shown to improve learning on deep networks with many layers.

Layer normalization adds a normalization step to each layer which is defined as:

$$LayerNorm(x) = a\frac{x - \mu^l}{\sigma^l + \epsilon} + b \tag{3.41}$$

Where μ^l and σ^l are the mean and standard deviation, respectively, of the output in x at layer t. Layer normalization helps the network learn by using normalization to smooth out irregularities in the input to layers. This has been shown to improve the convergence speed of networks (Ba et al., 2016).

The final output of any layer also undergoes dropout where a random ratio of weights are set to zero. Combining all of this, the output of a layer is defined as:

$$Layer(x) = x + Dropout(LayerNorm(f(x)) \tag{3.42}$$

Where f is the function implemented by the layer, such as multi-head attention.

The decoder works similarly to the encoder, but with a few additions.

The input to the decoder is the same as the input to the encoder, but offset by one. In the masked multi-head attention layer, future words in the sentence are masked out so that the current word being processed can only attend to past words.

This ensures that any prediction for a word at a given position can only depend on words in previous positions.

The output from the encoder is used in the next multi-head attention of the decoder. This output compromises the key and value vectors. The output from the previous attention layer is the query vector. The output of the second attention layer is likewise fed through a FFN layer.

After N pairs of encoder and decoder layers, the final output of the last decoder is fed to a linear classifier and a softmax layer. The final output is the probability distribution over what the next word in the sentence is likely to be.

Transformers are good choices for neural network based language models. As described in

2.1.2, language models try to model relationships between words in a language. To capture meaningful relations in a domain as vast as all words in a language, such models require a large amount of parameters. Transformers are designed to support deep networks with many layers. The residual connections allow for networks with more encoder/decoder layers (He et al., 2016). This means that more layers with more parameters can be employed before issues of vanishing or exploding gradients occur.

Transformers as Language Models

Transformer language models trained on large quantities of natural language can be fine-tuned to solve specific NLP tasks. Among others, these tasks may include ones outlined in 2.3. This is done by using one or more layers that shape the output of a language model into a format suitable for the given task. Then, the language model is trained again, this time with an optimization goal described by the task.

Finetuning a large pre-trained was done by Devlin et al. (2019) with the BERT language model. This model uses the transformer architecture as it was presented by Vaswani et al. (2017). One major difference is their use of bidirectional training. In the original transformer, words in a sentence are only influenced by preceding words. If the transformer is reversed, words are only influenced by succeeding words. In BERT, both is happening at the same time.

However, this gives too much information to the model, and it can simply output the next or previous word in the sentence, because no information is hidden to it. To force the model to actually learn, each word in the sentence has a 15% chance of being randomly masked out. This forces the model to adapt, with the hope that it captures some semantic information in the sentences. In addition to this optimization goal, BERT also optimizes for Next-Sentence Prediction (NSP). This works by selecting two sentences from the training corpus. 50% of the time, the first sentence is followed by the second. The rest of the time, it is a randomly selected unrelated sentence. The model then has to predict whether the second sentence follows from the first.

In RoBERTa, the masking part of BERT was changed slightly (Liu et al., 2019). Instead of using the same mask for several training steps, the mask was dynamically created at each training step. The RoBERTa paper states that this is crucial when more data and larger batches are used. NSP was removed, since it seemed not to be necessary. The model would perform equally well with just masking as a learning goal.

3.7 Size Complexity of NLP Models

The models achieving state-of-the-art performance on GLUE and other tasks has come with the drawback of increased parameter counts. This section will show the size of two relevant language models. This will give an idea of the size of the models that we work with, before any compression is applied. The first model is the RoBERTa large transformer (Liu et al., 2019) model which is the language model subject of our experiments. The second model is a slightly smaller bidirectional LSTM (BiLSTM) model, presented as a baseline for comparing performances on the set of GLUE benchmark tasks (Wang et al., 2019).

We have chosen to organize the parameters for these model into three distinct groups. The first group contains the input to the model, which is where sentences are encoded using embeddings. The next part is where the main learning takes place. For RoBERTa, this is the parameters used in the transformer layers. For the GLUE model, this is the parameters used for the BiLSTM layers. The final group contains the parameters that help shape what the output is and how it is formatted.

Figure 3.8 shows the distribution of parameters in the RoBERTa$_{\text{Large}}$ transformer model. The transformer contains the majority of the parameters in the model with just over 300 million weights. The encoding, which includes embeddings and positional encoding, contains almost 52 million parameters. Finally, the Language Model head, which shapes and formats the output to a specific task, contains 1.1 million parameters. With parameters represented using 32 bit floating point numbers, this model would take up roughly 1.42 GB on disk.

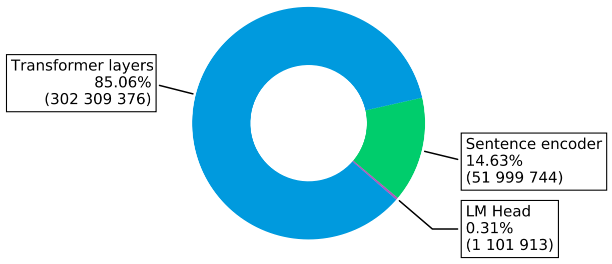

Figure 3.8: Distribution of parameters in $\text{RoBERTa}_{\text{Large}}$

Figure 3.9 shows the distribution of parameters for the GLUE baseline BiLSTM model (Wang et al., 2019). In this model, the vast majority of the parameters are found in the embedding layer. This is due to the use of high-dimensional contextual GLoVE embeddings. These embeddings capture contextual information of words within sentences, but come with a size cost. This figure shows the size of the model when it is used for sentence-pair tasks. For single-sentence tasks, the classifier layer is slightly smaller. Using 32 bit floating point numbers to store this model, it would take up approximately 3 GB on disk.

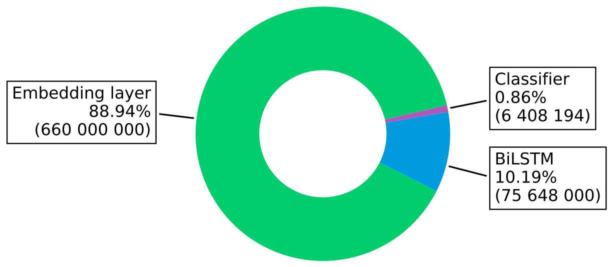

Figure 3.9: Distribution of parameters in GLUE BilSTM with GLoVE embeddings.

Figure 3.10 shows the distribution of parameters for another GLUE baseline BiLSTM model (Wang et al., 2019). This model uses ELMo embedding instead of GLoVE. Fewer embeddings vectors are used, but the embedding layer still makes up the majority of the parameters in the model. Again, the figure shows the model when used for sentence-pair tasks. Using 32 bit floating point numbers to store this model, it would take up approximately 700 MB on disk.

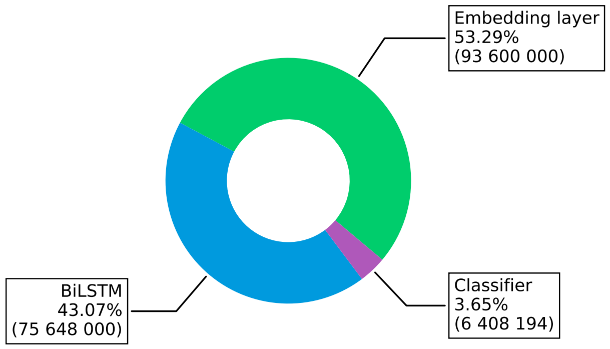

Figure 3.10: Distribution of parameters in GLUE baseline BiLSTM with ELMO embeddings.

These models are very large when put to application on resource-constrained devices. Storing, let alone utilizing their predictive capabilities might be problematic. However, the parameters in these models can be used to provide predictions for many downstream tasks. Let us assume that the information needed to solve these tasks are not largely overlapping. This would mean that utilizing these models for a single task is wasteful, as many parameters would be redundant.

In the next section, we will outline techniques for compressing pre-trained neural networks. Compressing these models will reduce their size and reduce redundancies in their learned parameters.

4 Model Compression

When storing, transmitting, or processing data, compression can be useful when the amount of said data causes bottlenecks. Compressing data consists of reducing the amount of bits used to represent the original data. It comes in two main fashions: lossless and lossy (Matt Mahoney). Both operates to reduce redundancies in the data: the former eliminates inefficiencies in the way data is ordered and coded; the other assumes that the representation can be approximated while preserving some level of quality.

Lossless compression involves re-assigning codes to patterns of data based on their probability. A codebook mapping patterns to codes is used both for compressing the data and uncompressing it. This entails that a complete restoration of the original data, from the compressed version, is possible. A simple example of such a coding is Morse code, where the letters that occur most often in the English language: e, a, and t, are all assigned the shortest codes. Thus, Morse code requires fewer dots and dashes than if all letters had fixed-length codes. This assumes that the encoded English text does not deviate too much from the usual probability distribution of letters. In practice this means that the encoded data is faster to transmit, or take up less space when stored.

Calculating entropy on data provides an estimation to what extent the data can be compressed losslessly. When there is low entropy, information needs less bits to be encoded. When there is high entropy more bits are required. A lossless compression algorithm is able to encode low entropy data to higher entropy data that uses less bits (Mutel).

Using the example of Morse code, it is obvious that the number of steps needed to successfully transmit English letters from a sender to a receiver has increased. There are now steps needed to encode to and decode from Morse code. I.e. lossless compression adds a run-time cost, both when data is compressed, but also when it is uncompressed for later usage.

Lossy compression removes unimportant information and creates a good enough approximation of the original data. This prevents a complete restoration of the original data. There is no one golden rule to what unimportant information and good enough constitutes. For audio-visual artifacts, unimportant information often refers to parts that are not widely registered by humans perceptions. Good enough refers to how much information that is considered unimportant is included and how much of this we can tolerate. Again, this is very situational.

Approximation can be done at different levels of what the original data represents: from the numerical values being stored to the structure of the architecture that produces a given output. A low-level approximation approach is to reduce the bits used for numerical representations. In other words, reducing the range that is being expressed. This is called quantization. Other approaches works at a higher level, by removing parts of an architecture or transferring the behavior from one architecture to another. In the context of machine learning model compression, these are called pruning and knowledge distillation respectively.

Like lossless compression, lossy compression also involves additional run-time costs when compressing data. But since an approximation is created, there is not the same requirement of a decoding phase. However, lossless compressed data of one form might require extra computational steps to be usable with data of a different form. For example, data using a limited bit range may need to be converted to full range, when used in conjunction with other full range values.

Since machine learning models are already approximations of solutions, one could argue that lossy compression of these models are suitable. Also, lossless compression of lossy compressed data is possible, where one can decode back to the lossy approximation. However, doing lossy compression on lossless compressed data will prevent going back to the original data. As such, we will focus on lossy machine learning model compression. In the remainder of this chapter, we explore several of these techniques: Distillation, pruning, and quantization, respectively.

Lottery ticket hypothesis. The lottery ticket hypothesis states that within large neural networks, smaller sub-networks exist with equal or better accuracy (Frankle and Carbin, 2019). For a given task, certain parameter values constitute a winning ticket because they perform especially well. However, it is difficult to decide which networks constitute a winning ticket. As such, it is simpler to create large networks to increase the chances that a winning combination of parameters are found within the network during training. These networks may have redundant parameters, as only a subset of the network is needed to perform well on a task.

Model compression can be used to uncover a winning ticket within a large pre-trained network. In the paper, Frankle and Carbin (2019) uses a technique called pruning to uncover the winning ticket in different networks. In this chapter, we will discuss three different model compression techniques, namely knowledge distillation, pruning, and quantization. These are all different approaches for attempting to find a winning ticket in a neural network.

4.1 Knowledge Distillation

Compression within the domain of machine learning can be divided into two main approaches. The first approach is to manipulate the parameters of the model somehow to decrease the size the parameters take up. The other is to create a smaller architecture that can solve the same problem with less parameters.

Knowledge distillation is an attempt at transferring transferring knowledge to a smaller architecture from a bigger one. The concept was first presented by Hinton et al. (2015). The idea is that a machine learning model will try to imitate the behavior of a larger model. The larger model is called the teacher model, and the smaller is called the student model. The two models need not be neural networks. As long as the student has a way of learning from the output of the teacher, any type of machine learning model could theoretically work. In Fukui et al. (2019), knowledge from a neural network is distilled to SVM, Random Forest, and Decision Tree models. However, in the context of this paper, we will only discuss neural network to neural network distillation.

4.1.1 Teacher and Student

The student model will learn from training data, in addition to learning from the teacher. This way, the student will learn the intricacies of the data, but will also be given a boost by learning from the teacher. Since the student is smaller and has fewer parameters at its disposal, it will not be able to model problems with as much precision as the teacher. However, the goal is that the knowledge the teacher encodes can be distilled into a smaller package. If this is done well, the student should be able to encode a compressed representation of the knowledge the teacher holds. Considering once more the lottery ticket hypothesis (Frankle and Carbin, 2019). It tells us that a smaller, but equally well-performing network should exist in a large one. Training a smaller student through distillation is another way of uncovering the winning ticket.

4.1.2 Distillation Loss

When doing knowledge distillation, there are generally two methods for training the student. The first approach is where the student uses the output logits of the teacher as its sole optimization goal. The logits are the output of a model before final activation. These logits contain more information than a normalized probability distribution would do after activation. The relative magnitudes of the logits gives more information to the student as opposed to a "squished" probability distribution.

The set of output logits of the teacher is also called *soft targets*. In certain cases, it may be beneficial to use temperature softmax to smooth out the distribution of output logits from the teacher (Hinton et al., 2015). This may be the case if the student model is very sensitive to differences in the magnitude of output logits. If this is the case, temperature softmax can smooth out the logits, making it easier for the student to learn.

The loss function for calculating the error between student and teacher logits can be freely chosen. In Hinton et al. (2015), cross entropy is used, while Tang et al. (2019) uses mean square error.

The second approach to knowledge distillation uses a linear combination between the soft targets and the error of the ground truth label in the training set. This can be defined as:

$$\mathcal{L} = \alpha \cdot \mathcal{L}_{label} + (1 - \alpha) \cdot \mathcal{L}_{distill} \tag{4.1}$$

where $\mathcal{L}_{label}$ is the loss between the student output and target label and $\mathcal{L}_{distill}$ is the loss between student output and teacher output. α is a hyper parameter that determines how the two measurements of loss are weighted. With $\alpha = 0$, the loss of the target label is ignored. With $\alpha = 1$, the loss of the output logits between student and teacher is ignored, meaning no distillation is done.

Distillation is most often done on the final output of the teacher model. However, distillation to and from intermediate layers is also possible. In Jiao et al. (2020), the BERT transformer model is distilled to a smaller version. Distillation is done not only at the output level, but also in three intermediate layers. The student is trained on output from the embedding layer, the transformer attention layer, the transformer output layer, and the final prediction layer. Naturally, this approach only works if the student model has an architecture similar to that of the teacher.

4.1.3 Bootstrapping

Distillation benefits greatly from having a large amount of training data. More data will mean more ways for the teacher to "express" itself. This will make it easier for the student to learn from it.

In the context of a language model, training data is not hard to come by. Language models are trained on unlabeled textual data. The data may need cleaning and validation, but there is plenty of text to find on the internet. For specific NLP tasks, training data needs to be structured and critically needs to be manually labeled. However, having access to a large teacher model presents an opportunity. The teacher can be seen as an oracle that can provide high quality output in the domain of the given NLP task. This can be exploited, in order to generate synthetic training data that can be used during knowledge distillation.

Synthetic data can be generated in a number of ways. The most important thing is that the structure of the training data remains intact. Simply feeding an arbitrary sentence to the teacher and using it to label the sentence will not work. Different tasks have different sentence structure. An example is tasks with the category of *question answering*. A training example for this task could include a question, some choices for answers, and the corresponding correct answer.

An approach that can generate synthetic data with correct structure is data augmentation. This involves taking an existing dataset and altering the training examples to create new data. The trick is to not change the structure or meaning of training example. However, it should be changed enough to provides new information for the student model.

One method for data augmentation was used by Jiao et al. (2020) for their TinyBERT model. They used distillation to compress the BERT language model to a much smaller size. During distillation, they bootstrapped their model with additional augmented training data samples. The method for doing so is described in Algorithm 1 in their paper. We will briefly summarize how it works here.

They went through each sentence in the training data. For each word w in each sentence, they see if w can be split up into sub-word representations by BERT. If it can't, they mask out w in the sentence and use BERT to predict K most likely candidates for what the masked word should be. If w can be split up, they use GLoVE to find the K most similar words to w by word embedding vector similarity.

Once this has been done, they sample a random number $r \in (0, 1)$ from a uniform distribution. If $r < p$, they replace w in the sentence with one of the K random word replacements acquired earlier. They do this process N times for each sentence in the training data.

In their experiments, they used the values $p = 0.4, N = 20, K = 15$.

4.2 Pruning

Pruning is a technique that works under the assumption that most neural networks contain redundant parameters. Pruning revolves around removing the parameters in a network that don't contribute significantly to the output. How much a parameter contributes depends on the type of model architecture and the nature of the input. As such, it is not straight-forward to determine which parameters can be pruned. In practice, pruning is done by creating a binary mask which can be multiplied with the weight matrices in the network (PyTorch, 2019e). Where the binary mask is zero, the corresponding weight will be zero and will effectively be removed from the network. How this mask is calculated depends on the pruning technique being used.

4.2.1 Magnitude

The most basic and naïve approach to pruning is called magnitude pruning (LeCun, 1990). Here, the magnitude of the values of weights decide whether they get pruned. The idea is that values with small magnitude contributes negligibly to the output of the network. Pruning these should not degrade the performance of the network much, and may even help it generalize better. Magnitude pruning is very simple, but can be quite effective if applied correctly. Han et al. (2016) showed that simple magnitude pruning can be very effective. In their experiments, they reduced the size of image classification networks LeNet and AlexNet by 9x and 13x respectively with little incurred accuracy degradation.

4.2.2 Top V

Top V pruning is a variant of magnitude pruning where weights with low magnitude are pruned. We rely on the definition of Top V defined by Sanh et al. (2020). Top V differs from magnitude pruning by not setting a static threshold on the magnitude of weights that are pruned. Instead, weights in a layer or in the entire network are sorted by magnitude. The top $k\%$ of weights are kept and the rest are pruned. This allows for more control over how sparse the pruned network should be. This is essentially the reverse of magnitude pruning which selects a magnitude threshold and prunes the weights with values below this. This will prune some percentage of the weights in the model. Top V pruning selects this desired percentage of weights to prune and then finds the magnitude threshold that allows for this.

4.2.3 Movement

Movement pruning is a technique proposed by Sanh et al. (2020). Movement pruning is a first-order method for selecting which weights to prune. The network learns which weights to prune during training. It does so by selecting weights that don't move away from low values during training. Doing this, both the magnitude of weights and their gradients are taken into effect when pruning. The idea is that weights with small magnitude and small gradients provide less information to the network. These weights are not updated much during training because they don't interact strongly with input to the network.

4.2.4 Local vs. Global

Pruning can be done at a *local* or a *global* scale. Global means applying the previously discussed pruning techniques across the entire network at once. This can work well if one wants to control the sparsity of the pruned network very precisely. It also works best if the network uses the same architecture throughout. If the network consists of a BiLSTM with a FFN at the end, then local pruning is often more optimal.

Local pruning is applied across a specific layer within the network. This allows for more fine-grained control of how parameters get pruned. For example, if a model uses word embeddings, pruning too many of these weights might pose problems. The structure and information encoded in early parts of a model are more sensitive to the perturbations that pruning causes. The same is true for sequence based neural networks, such as RNNs or BiLSTMs. Changing the early hidden states in such a network can cause ripple effects through all the hidden states.

Pruning during training may also help the network adjust to removed weights. This is called *training-aware pruning*. Instead of simply pruning once after the network has trained, pruning incrementally throughout training can result in better performance (Han et al., 2015, 2016).

4.2.5 Structured vs. Unstructured

Another thing to consider when pruning is the way in which layers are pruned. Figure 4.1 and 4.2 show the difference between the two concepts structured and unstructured pruning.

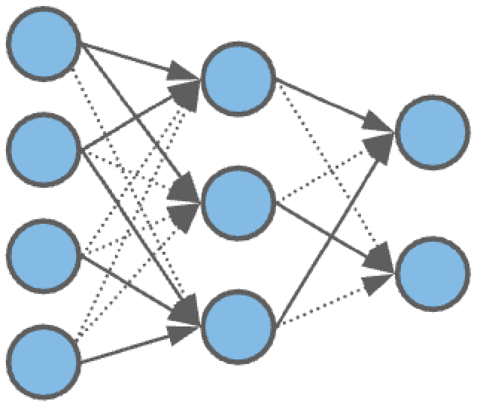
Figure 4.1: Unstructured pruning - Dotted lines represent pruned connections

In unstructured pruning, parameters are set to zero with no regard to the structure of the network. No care is taken as to which connections are pruned, or the imbalances it may cause in the network structure. Structured pruning on the other hand, removes whole neurons at once. All connections to and from a pruned neuron or layer is thereby removed. With unstructured pruning, individual weights in weight matrices are set to zero. With structured pruning, the values of entire columns are set to zero. In a even more aggressive form of structured pruning, whole layers may be pruned, instead of individual neurons.

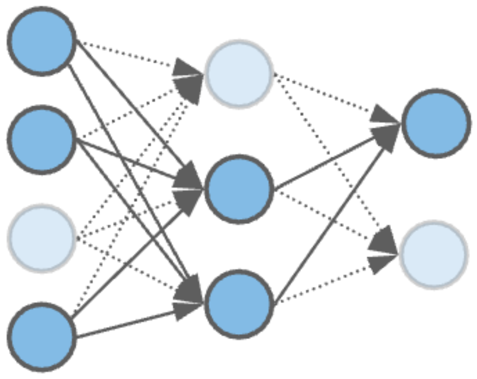
Figure 4.2: Structured pruning - Transparent circles represent pruned neurons

These two strategies each have their advantages and disadvantages. Unstructured pruning allows for more granularity when deciding which parameters to prune. This may lead to smaller models, but comes with a cost. Having pruned weights mixed in with non-pruned ones may hurt the computational efficiency of the network. Matrix multiplication in neural networks are their bread and butter. Many values in a weight matrix being zero means many wasted multiplications. Sparse matrices like these may also inhibit performance of dot-product optimizations that expect most values to be non-zero.

Structured pruning allows for more balanced networks run faster on consumer grade hardware. However, removing entire neurons can end up hurting the prediction capabilities of a network.

4.2.6 Achieving Smaller Footprint

An important aspect of pruning, is what is done with pruned weights. If they are left as is, pruned weights will simply be the number zero. This does not make them disappear, and no decrease in size will be achieved. The numbers will still be stored and take up space, unless the underlying storage scheme is changed, such that the zeros are represented by fewer bits than the number that they replaced.

A smaller footprint can be achieved by using a way of storing the pruned weight matrices that leverage their sparse nature. There are several data structures for doing this, each with benefits and drawbacks. One method for doing so is Huffman Coding (Huffman, 1952). This technique works by assigning the fewest bits of storage to sequences appearing most frequently. When used with pruned weight matrices, pruned weights being zero would be compressed most efficiently. With more heavily pruned layers, storage requirements would be reduced significantly. One drawback, is that this would require extra computational overhead to compress and decompress weight matrices during model inference.

4.3 Quantization

Quantization is yet another lossy compression technique. It is a process that maps numerical values from one representation to another with a more limited range of possible values. The theory emerged in the 1940s, while some of the ideas date as far back as 1898 (Gray and Neuhoff, 1998). The oldest, and perhaps the simplest, example of quantization is rounding off a real number to the nearest integer value.

Reducing precision for machine learning models has the benefits of a smaller footprint, less working memory and cache, faster computation, and last but not least the network architecture can stay unchanged (Krishnamoorthi, 2018). Furthermore, Woodland (1989) showed that quantization had the potential to make models more general in what they learned from training data, i.e. act as a sort of regularization.

A recent popular method for reducing numerical precision in machine learning is to use half-precision floating point numbers (IEEE, 2008). Half-precision floating points only take up 16 bits per number compared to 32 bits for single-precision. In many situations half-precision has shown to achieve similar accuracy to single-precision models, especially when a hybrid version has been applied where activation functions are kept at single-precision (Dally, 2015). Many modern graphic cards support half-precision with optimized operations, thus it not only reduces memory and bandwidth usage, but also increases throughput during computations.

Using even lower precision, ML researchers and developers often turn to 8 or fewer bit integer quantization. This results in even smaller model footprint than is possible by merely using half-precision floating points.

Going from a higher to lower precision incurs a potential loss of information. This is called the quantization error. For a single number x, the quantization error is defined as:

$$error = q(x) - x \tag{4.2}$$

where $q(x)$ is called the quantization rule, i.e. it is a function that produces the quantized version of x. Rewritten, the quantization rule product can be written as the error added to the original number:

$$q(x) = x + error \tag{4.3}$$

This error is usually not saved, which makes quantization a lossy operation. This means that an exact reversal from a quantized version to the original is not possible.

4.3.1 Quantizer

A quantizer can be seen as a set S of cells, each with a numerical interval, and a set C that maps cells to specific values. The specific values are called reproduction values (Gray and Neuhoff, 1998). Formally speaking, the components of a quantizer can be defined like this:

$$S = \{S_i; i \in I\}, \tag{4.4}$$

where I is an index set 0..n, S_i is an interval

$$C = \{y_i; i \in I\} \tag{4.5}$$

$$q(x) = y_i \text{ for } x \in S_i \tag{4.6}$$

Going back to the rounding off to nearest integer, the S_i intervals would look like this:

$$S_i = (i - 0.5, i + 0.5] \tag{4.7}$$

and the reproduction values would simply be:

$$y_i = i \tag{4.8}$$

A quantizer that has evenly spaced intervals is called uniform, and the space is noted as Δ. In the example of rounding a real number to nearest integer, the quantizer's Δ is equal to 1. Also, the width of the intervals would be Δ, illustrated in figure 4.3.

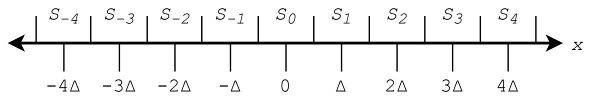

Figure 4.3: Infinite uniform quantization of real numbers to nearest integer.

This example quantizer is infinite, as the amount of cells and reproduction values are not limited. When dealing with digital representations of numbers, this is often not the case. With finite cells and reproduction values, the first and the last cell on the line are said to be semi-infinite, as they only have one limit and the other being negative infinity and infinity respectively (Gray and Neuhoff, 1998).

It should also be mentioned that a quantizer does not have to be uniform, nor does its reproduction values have to reside at the middle or even within its corresponding interval.

A main goal of quantization is to keep the best fidelity while decreasing the bits used for numerical representation. If our example of rounding real numbers to integers were applied to computers and we chose to use 32 bit integers, it would no longer be infinite. At the same time it would also be very indiscriminate in the way it treated data, not taking into account the probability distribution of the data to be quantized.

Some researchers have used stochastic rounding in unison with limited precision, to some success (Wang et al., 2018; Gupta et al., 2015). Stochastic rounding uses the part that will be stripped as a probability for deciding whether to round down or up. In a very basic sense it can be described like this:

$$Round(x) = \lfloor x + R \rfloor \tag{4.9}$$

where R is a random real number in the range $(0, 1]$.

In a hypothetical scenario where we had an amount of numbers all equating to 1.4 rounding them to nearest integer would give us an average value of 1. With stochastic rounding the average value of the rounded numbers would be 1.4, thus preserving some information about the data distribution. Both Wang et al. (2018) and Gupta et al. (2015) used custom made hardware to limit the computational overhead for getting random numbers at runtime.

4.3.2 Scalar Quantization

A computationally simpler way to quantize, while still taking the distribution of the data into account, is to use scalar quantization. In scalar quantization from floating points to lower-precision integers one usually uses 2 derived variables: A scale, or step size, denoted Δ; A zero-point Z (Krishnamoorthi, 2018).

When quantizing a floating-point number x to a 8 bit unsigned integer we are limiting the quantized values to be in the range $(0, 255)$. The required steps are:

$$x_{int} = Round\left(\frac{x}{\Delta}\right) + z \tag{4.10}$$

$$x_Q = clamp(0, 255, x_{int}) \tag{4.11}$$

where $clamp(min, max, x)$ gives x if it is within min and max, otherwise min if that is larger than x or max if that is smaller than x. This ensures that the process can handle every floating point number, thus making the intervals used for the min max semi-infinite.

Dequantizating back to floating points is equally simple:

$$x_{float} = (x_Q - z) * \Delta \tag{4.12}$$

The described quantization method is what is called a *uniform affine scalar quantizer* (Krishnamoorthi, 2018) and is fixed-rate as we always quantize to unsigned 8 bit integers. Other forms of scalar quantization exist, e.g. *symmetric* where the zero-point always equals 0, and thus can be dropped from the equations. Another one is a *stochastic* quantizer, where a random uniform number between -0.5 and 0.5 is added to x in equation 4.10. However, most hardware is not optimized for the stochastic quantizer, making it a less popular option.

Many alternatives to scalar quantization exist. In *predictive quantization* the quantization of a sample depends on previously seen samples. *Transform coding* with quantization combines lossless matrix transforms of sample data with lossy quantization. *Variable-rate quantization*, based on Shannon's channel coding, assigns variable width binary codes to quantized values, and uses a codebook for storing the actual quantized numbers. Finally, *Vector quantization* takes a block sample and quantizes it to a single vector, instead of quantizing one number at a time (Gray and Neuhoff, 1998). However, oftentimes scalar quantization has sufficient quality, excels in its simplicity, and most importantly are usable for computation in their quantized binary representation. This last point is especially true for most modern CPUs, which have hardware-support for 8-bit integer arithmetic (Krishnamoorthi, 2018).

4.3.3 Quality

When dealing strictly with quantization the quality is usually measured as a tradeoff between two main aspects. First is the *Distorion*, which often is the mean squared error for quantizing data X with a quantizer q. This is denoted as D. The second is the width of the binary representation of the quantized numbers, called the *Rate* and denoted as R (Gray and Neuhoff, 1998).

$$d(X, X_Q) = |X - X_Q|^2 \tag{4.13}$$

$$D(q) = E[d(X, q(X))] \tag{4.14}$$

$$R(q) = log\ N \tag{4.15}$$

There exists many different function definitions on how to measure this distortion-rate trade-off, often denoted as $\delta(R)$ when using a fixed-width R. These depend on the specific quantization method.

However, for machine learning the quality of a quantized network model should not primarily be measured as the quality of the quantized network weights, but how the model behaves on data, i.e. its inference performance.

5 Related Work

This section outlines a selection of similar work to ours, within model compression in NLP and other areas. Broader context for the compression techniques mentioned in this section is given in section 4.

There is nothing novel about the notion that a lot of space in machine learning, and other data-intensive models, is wasted on noise or redundant information. Han et al. (2016) showed that several CNN-based models trained on image classification could be compressed by a ratio between x35 and x49. This was achieved while barely hurting the performance of the models. Compression was done with a combination of quantization, pruning, and Huffman coding.

Within the realm of NLP, Fedorov et al. (2020) conducted research into speech enhancements for hearing aids. They created an RNN model for filtering out noise in hearing aids. This model had to fit within very harsh constraints in terms of model size, memory usage, latency, and quality. They present similar approaches to those in this thesis for creating models that meet the given constraints. Namely, they utilize structured pruning and training-aware quanitzation to compress their speech enhancement model.

Research into compressing language models in particular has seen a lot of attention after the release of BERT. In March 2020, Sanh et al. (2019) presented *DistillBERT*. They employed knowledge distillation to create a version of BERT that is 60% of the size and runs 60% faster. The model retained 97% of the prediction performance of the full BERT model. It contained half as many transformer layers, but was otherwise similar. The goal was to run BERT on mobile devices with low enough latency to be viable.

Another approach at knowledge distillation was done by Sun et al. (2020) in their MobileBERT model from April 2020. They don't change the number of layers in the model, but instead opt to decrease the amount of parameters in each layer. They use a layer-to-layer distillation technique. This means that each encoder block in the student model learns to imitate the output of the corresponding block in the teacher. On top of this, they also use regular distillation for the output of the model. They attempted different strategies for this distillation, including distilling all at once or in iterative steps. They achieved a 7.2x reduction in parameters while maintaining comparable accuracy to BERT.

Zhang et al. (2020) showed a version of BERT that used extremely low precision parameters. Their model was called *TernaryBERT*. This model was created by a combination of knowledge distillation and quantization. They first distilled BERT to a smaller model. This model was then ternarized, meaning each weight in the model was reduced to one of three possible values -1, 0, or 1. During optimization, they kept the weights at full precision. Their model is almost 15x smaller than BERT and achieves comparable accuracy on several GLUE tasks.

These approaches all compress BERT to a smaller, but still general model. These smaller models are still language models, and can be fine-tuned to work on most NLP tasks. Distilling language models to task specific models was explored by Tang et al. (2019). They experimented with distilling BERT to a BiLSTM model for three separate GLUE tasks. Their BiLSTM model for sentiment analysis on the SST-2 task was reported to be 349x smaller than BERT, while achieving only a couple percentage points lower accuracy.

Similarly, Wasserblat et al. (2020) experimented with distilling to task-specific networks.

They distilled BERT to three different models. The first was a continuous bag of words (CBoW) FFN model. The second was a BiLSTM, and the third was a convolutional neural network (CNN). They experimented with low-resource distillation where little training data was used. The CBoW model achieved a size reduction of an impressive 1375x compared to BERT and the BiLSTM model saw a 160x reduction. These models achieved comparable accuracy on a variety of tasks, including SST-2, but much lower accuracy on the Corpus of Linguistic Acceptability (CoLA) grammar correctness task.

The work done by Tang et al. (2019) and Wasserblat et al. (2020) is most similar to ours. They focus on distilling BERT to non-transformer neural network. However, we go a step further and see how much models can be compressed when distillation is combined with further compression. Most research within model compression attempt to create smaller models that exhibit similar performance to a given baseline. In resource constrained domains, creating models that fit within the given constraints may be very difficult. Ensuring that such models exhibit similar performance to larger baseline models may be infeasible. In our research, we wish to see how far models can realistically be compressed, and how they perform when this is done.

6 Experimental Setup

We now focus on our research questions on the relationship between performance and model size complexity, and which compression ratios are possible. In order to answer these questions, we needed a clear strategy for what experiments to run and how to conduct them.

Rogers et al. (2020) reviewed the effect of different compression techniques applied, by other researchers, to BERT. It is apparent that the most ambitious compression ratios involve knowledge distillation. This results in an intrinsically smaller model with a different architecture. When working with smaller architectures, one pitfall to be aware of is how sentences are encoded. If the vocabulary and dimensionality of embeddings are not kept small, model size complexity will increase (Wasserblat et al., 2020), (Tang et al., 2019). Thus, for the student models in our experiments, we experiment with embeddings that are small, and avoid disproportionally increasing the problem of unknown words.

A common theme in knowledge distillation is the need for large amounts of training data. This is often alleviated by creating augmented data from existing training data and combining these two datasets into one (Jiao et al., 2020), (Tang et al., 2019).

Finally, as we want to push the envelope on compression, we want to experiment with further compression techniques on small distilled models. This is to explore the lower limits of size. Namely, we will apply pruning and quantization to these models. Such a pipeline approach has been done with success in machine learning fields outside of NLP (Han et al., 2016). To our knowledge this has not been done for Language Models on the GLUE dataset.

This chapter outlines choices made for the setup and execution of our experiments. It includes frameworks, conditions for doing training, model compression, workloads, and testing. We will also describe the benefits and limitations associated with implementing many of the compression techniques described in previous sections.

6.1 Software Setup

Our program for doing compression and testing is written in Python 3.8. Python has many external libraries for machine learning and data science and is therefore broadly used for doing machine learning research. However, Python is an interpreted language. This makes it easy to work with, but also quite computationally slow and less streamlined. If the models we experiment with in this paper were to be used in a production setting, a compiled language would be preferable. Such a language may include C++, Rust, etc.

The program is split into several modules, each concerning a distinct concept. Furthermore, each module can load the output of previous modules in the pipeline, and save its own output. This allows for more flexibility in experimenting with specific modules.

Figure 6.1 illustrates the pipeline flow we envisioned for our program. All the solid boxes are modules that we wanted to both be able to either run in a pipeline fashion, or individually if the necessary data or model had already been created. A guide to running our program is found in appendix C.

In following subsections we will delve into how we went about implementing each of the modules.

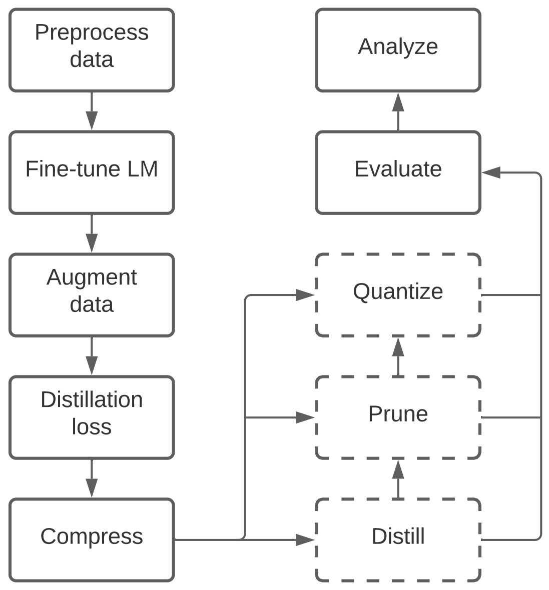

Figure 6.1: Program flow.

6.1.1 ML Framework

Several machine learning model compression pipelines have been created in recent years. In 2017 Apache unveiled an open source machine learning optimization tool called TVM, which offers end-to-end optimization for machine learning models implemented across many frameworks (Chen et al., 2018). In 2018, Google released a compression tool for their ML framework TensorFlow, called TensorFlow Lite. It offers many of the same features as Apache TVM, but only works for models built in TensorFlow. In 2019, researchers at Intel Labs presented a compression tool called *Neural Network Distiller*, which supports many types of pruning, quantization, low-rank matrix estimation, and distillation (Zmora et al., 2019).

These tools focus on general purpose model compression and optimization. For the purpose of our experiments these frameworks were too constrained in terms of supported model architectures.

We needed a flexible machine learning framework for implementing different types of models. We wanted as much freedom as possible when implementing and tuning models using different compression techniques. Instead of relying on a pre-existing tool, we decided to use PyTorch and implement these techniques ourselves. PyTorch is good for prototyping, while also being optimized for efficient full-scale training. It is a well-rounded framework and has in the past two years started to add rudimentary support for pruning (PyTorch, 2019e) and quantization (PyTorch, 2019f).

We used PyTorch version 1.8.1. This was necessary, since earlier versions did not support quantization for embeddings. Even in this version, quantization and pruning are still at a beta stage. We ran into several limitations with these features which we will discuss in more detail in sections 6.1.6 and 6.1.7.

6.1.2 Language Model

In order to conduct experiments with knowledge distillation, we needed a suitable teacher model. We chose the RoBERTa language model for this purpose. Pre-trained versions of RoBERTa are available through the open source Fairseq framework (Ott et al., 2019). From this framework, models of different sizes are available. We chose to work with the largest available model called RoBERTa$_{\text{Large}}$.

To utilize RoBERTa for distillation in the context of specific NLP tasks, we needed to fine-tune it for each specific task. The tasks we chose are described in section 6.2.2. Fine-tuning works by altering the given language model slightly. This is done by attaching a *classification head*, which shapes the output of the language model to a format that fits the given task. Then, the model is trained on training data for this task. Once we fine-tuned RoBERTa for a specific NLP task, we were able to use it as a teacher model for doing knowledge distillation.

Having access to a language model trained on vast quantities of data enables another benefit. Such a language model can be used to generate synthetic training data that can be utilized when doing knowledge distillation. This is described in more detail in section 4.1.3

The Fairseq framework enabled us to easily interface with the RoBERTa model when we needed to fine-tune it for a specific task or when we used it for knowledge distillation. We only had to make a few adjustments to the code responsible for fine-tuning the RoBERTa model.

6.1.3 Distillation

As described in equation 6.3, the loss of a student model in knowledge distillation often requires measuring both the loss between teacher and student output, and correct labels in the data.

In our experiments, the loss function for measuring the deviation between the output of student and teacher is:

$$\mathcal{L}_{distill} = MSE(\boldsymbol{y^s}, \boldsymbol{y^t}) \tag{6.1}$$

where $\boldsymbol{y^s}$ and $\boldsymbol{y^t}$ are the output logits of the student and teacher model respectively. As discussed in 4.1.2, using the raw logits from the models provides more information than using a squished softmax distribution. The only exception to this is when we run distillation with our FFN student model. We describe student models in 6.1.4. Here, we apply temperature softmax to $\boldsymbol{y^s}$ and $\boldsymbol{y^t}$ with a temperature of 3 (Fukui et al., 2019). This empirically improves performance, which is shown in appendix A.

The loss between $\boldsymbol{y^s}$ and $\boldsymbol{y^t}$ is measured by the mean squared error as described in equation 3.9.

The loss between the output of the student model and the correct label, is given by cross-entropy loss as described in equation 3.10:

$$\mathcal{L}_{label} = CE(\boldsymbol{y^s}, \boldsymbol{y^t}) \tag{6.2}$$

and finally, the combined distillation loss is a linear combination of equation 6.2 and 6.1 as described in section 4.1.2:

$$\mathcal{L} = \alpha \cdot \mathcal{L}_{label} + (1 - \alpha) \cdot \mathcal{L}_{distill} \tag{6.3}$$

As described in section 4.1.2, α controls the relative weighing of label loss versus distillation loss. Setting alpha to 0 would train the model on only soft targets from the teacher model. Setting alpha to 1 would train the model on only labels in the training data. Anything between would be a mixture of the two.

6.1.4 Student Models

We implemented three different student models for experimenting with distillation and further compression.

BiLSTM The first student model is a bidirectional LSTM, mainly inspired by work done by Tang et al. (2019). They demonstrated the effectiveness of a single-layer BiLSTM model with very few parameters. This model uses a FFN with a single hidden layer as the output classification layer. They used this model as a student model in knowledge distillation, and achieved decent performance on SST-2, QQP, and MNLI.

We follow the same setup as in their paper, but tune certain parameters differently. Like Tang et al. (2019), for single-sentence tasks our BiLSTM has a hidden size of 150 and the classifier a hidden size of 200. For sentence-pair tasks these numbers are doubled. We also follow Tang et al. (2019) and Wasserblat et al. (2020) by using the last hidden state, in each direction, as the output from the BiLSTM.

We also implemented the option to use max pooling on all hidden states, for usage as input to the FFN classifier layer. Wang et al. (2019) treated the output of the BiLSTM this way, for their GLUE baseline models. Initial investigations by us on SST-2 indicated using the last hidden state as BiLSTM layer output lead to better results.

An overview of configurations and hyperparameters for all our student models can be found in table 6.1.

RNN To complement the BiLSTM model, we also implemented a simpler sequential model. This model works similarly to our BiLSTM model, but with the BiLSTM replaced with a simple Elman RNN. Like the BiLSTM, our RNN is also single-layer and bidirectional.

For a given number of hidden units, this type of RNN has fewer parameters than a BiLSTM. This model also uses a FFN with a single hidden layer as the output classification layer.

The hidden size of our RNN is set to 100 for single-sentence tasks, which again is doubled for sentence-pair tasks. Similarly the classifier has

100 and 200 dimensions. Like with the BiLSTM, the output of the RNN layer is the last hidden state in both directions.

FFN Finally, the simplest and smallest model we experimented with is a feed-forward network. This model is loosely inspired by work done by Wasserblat et al. (2020). This paper explores the use of low dimensional continuous bag-of-word (CBOW) word embeddings in conjunction with an FFN. Our implementation simply takes the mean of the embeddings across an entire sentence for each embedding dimension. The output from this is fed to an FFN classifier with one hidden layer.

Sentence-pairs When the input is not a single sentence, but a pair of sentences, we do things a bit different. Depending on the architecture, both sentences will be encoded and given to either the BiLSTM layer, RNN layer, or the mean function in FFN. This creates two vectors, u and v. The input to the MLP classifiers in this case is a concatenation of the two vectors, along with a comparison of them. The comparison consists of the pointwise absolute difference and pointwise multiplication of them:

$$MLP_{input} = [u; v; |u - v|; u * v] \tag{6.4}$$

This procedure is the same described and used by Wang et al. (2019) for their GLUE baseline models. Tang et al. (2019) denotes it as *a standard concatenate–compare operation.*

6.1.5 Embeddings

As described in section 3.7, the size of the embedding layer of RoBERTa and the GLUE baseline model with Elmo is rather large. They contain about 52 and 93 million parameters respectively. Since we are examining lower bounds for size of our models, looking at minimizing the size of the embedding layer is obvious.

There are two factors that govern the required space of embeddings: The amount of vectors, often denoted the vocabulary, and the dimensionality or length of said vectors. With regular word-level embeddings, like GloVe, reducing the vocabulary aggressively has the possibility that a lot of the text seen by the models will resolve to the same unknown vector. For this reason we decided to experiment with embeddings that were said to be better at handling the unknown word problem, while still maintaining a small footprint. These are Byte-pair embeddings, character embeddings, and hash embeddings.

Character embeddings. These we implemented ourselves, with a mapping of 33 letters and signs to vectors in a PyTorch *Embedding* layer, initialized with random values. With only 33 individual vectors these character embeddings can have a very small footprint, but it might affect expressibility. Also, with sentences being split up and encoded on a character basis, this approach takes longer time to process for sequential models.

Byte Pair embeddings. Here we simply used a python library with pre-trained byte pair encoded embeddings by Heinzerling and Strube (2018) called *bpemb*. These embeddings are trained on Wikipedia. The sentence encoding is a mixture of word and subword encoding, with a possibility of going all the way down to character level.

Hash embeddings. These were also implemented by us, following the descriptions by Svenstrup et al. (2017). We use MD5 as our hashing function. When encoding a word in a sentence we append a number to it three times, and gets three integers from the hashing function used for indexing later on. When constructing vector for a word we first use a PyTorch *nn.Embedding* of scalar values, divided intro three sectors for each of the three index integers. Each of the three scalars are then used on three vectors from a shared *nn.EmbeddingBag* layer, and finally they are summed to a single vector.

Common for all three embedding types we used for our experiments is that they do not suffer from the unknown words problem to the same extent as word-level embeddings like GloVe.

6.1.6 Pruning

Pruning is *mostly* supported in PyTorch (PyTorch, 2019e). Methods for computing binary masks in terms of input tensors and applying these to weight matrices is supported. However, we attempted to implement movement pruning, as described in 4.2.3. Movement pruning requires learning the binary masks during training. This is not particularly well suited for how PyTorch handles pruning. Binary masks are saved in a static format, and are not easily converted to optimizable parameters on the fly. We considered implementing our own pruning scheme to allow for learnable masks, but decided against it for reasons of scope.

We implemented magnitude and top V pruning as described in 4.2. We made use of the PyTorch class *BasePruningMethod* (PyTorch, 2019a) by overriding the *compute_mask()* method. This method receives a tensor to be pruned and creates the binary mask that used for pruning.

Magnitude pruning. For magnitude pruning, the binary mask is computed by setting the entries in the mask to zero where values in the tensor is less than the magnitude threshold.

Top V pruning. For topV pruning, the weights are first sorted by magnitude. Then the entries for the lowest ratio of weights, according to the topV threshold, are set to zero.

During runtime, we can select between local and global pruning, as described in 4.2.4. Global pruning uses the PyTorch method *global_unstructured()* for pruning all tensors in one go. Local pruning uses the *custom_from_mask()* method for each tensor separately. This way, we can adjust pruning thresholds for each type of tensor separately. For a detailed overview of this, see table 6.3.

As mentioned in 4.2.6, the manner in which pruned weights are stored impacts the practical gain of pruning. PyTorch does not store pruned weight matrices differently from regular parameters. However, a more efficient way of storing sparse matrices is supported. A data structure called a sparse tensor can be used which stores data in coordinate list (COO) format (PyTorch, 2019k). This means that only the coordinates and values of non-zero weights are stored. The storage space in bytes for a tensor in COO format is:

$$(d \cdot dtype_i + dtype_e) \cdot nze \tag{6.5}$$

where d is the number of dimensions of the data. $dtype_i$ is the size of the data type of the coordinates pointing to non-zero entries. $dtype_e$ is the size of the data type for the non-zero entries. nze is the amount of non-zero entries in the sparse tensor. PyTorch uses long integers to store coordinate entries. Thus, when calculating the size in bytes, we can assume $dtype_i = 8$.

On the other hand, the size of a non-sparse tensor is calculated as:

$$dtype_e \cdot n \tag{6.6}$$

where d is dimensions and n is the total amount of elements in the tensor.

If we assume we are training with floating-point weights with 32 bit precision, we can set the value of $dtype_e = 4$ when measuring bytes.

Rewriting the two equations in terms of each other, and solving for nze, leads to:

$$nze = \frac{n}{2d+1} \tag{6.7}$$

Thus, if the tensor is two-dimensional, the amount of non-zero weights should be less than $\frac{1}{5}$ of all weights before the sparse tensor achieves a lower storage cost. As dimensionality of the data goes up, this inequality grows because sparse tensors needs to store coordinate indices for each dimension. The weight matrices in our models are either two or three dimensional. We would need to prune respectively 80% and 86% of weights in two and three-dimensional matrices before the use of sparse tensors become beneficial. Because of this, we do not use sparse tensors in our experiments.

6.1.7 Quantization

Fixed-width scalar quantization of PyTorch modules is supported to various degrees. At the time of writing it is officially in a beta stage (PyTorch, 2019f). It works either with scale factor and zero point for an entire tensor, denoted *per tensor* or for each dimension of a tensor, denoted *per channel*. Currently, it is only supported on newer x86 and ARM CPUs, not GPUs.

The quantization function differs slightly from equation 4.10, in that the zero point is added to the scaled number prior to the rounding operation:

$$Q(x, scale, zp) = Round\left(\frac{x}{scale} + zp\right) \quad (6.8)$$

The numerical representation that numbers are quantized to are either 8-bit unsigned or signed integers. In terms of space, that is a $4x$ reduction when quantizing from 32-bit floating point numbers. However, a small overhead is added with the scale and zero point.

Additionally, there are three types of quantization: *Quantization aware training*, where a quantized model is kept along side the original; *Dynamic*, where weights are quantized and activations are dynamically quantized prior to computation; and *Static*, where both weights and activations are quantized.

Our implementation

All embedding types can be quantized as static quantization, on a per channel level, meaning that each individual embedding vector that can be indexed has its own scale factor and zero point. The only exception is the embedding layer used for scalars in our Hash embeddings implementation. Because the individual vectors only contain a single number the overhead of scalar quantization would have made the size increase. Instead we simply convert this layer to half-floating point numbers, and thus halving the size of the module from the original 32-bit version.

BiLSTM can be quantized with dynamic quantization, as this is the only version supported by PyTorch. Quantization of PyTorch RNN module is not supported. We implemented a custom Elman RNN module, that can take a PyTorch RNN module and copy its parameters. This custom module can be dynamically quantized. However, it is not optimized in the backend the same way as the built-in RNN module, and as a consequence does not perform computations as fast.

For our MLP classifier layers we implemented both dynamic and static quantization, as PyTorch supports both for linear layers.

We strayed away from quantization aware training. This was motivated by lack of support for all of the modules we were using, and because we saw good initial results when trying out the combination of dynamic and static we ended up with.

6.2 Workload Setup

We wish to test several compression techniques, applied to a selection of model architectures with different parameters on three different NLP tasks. In particular: Across all three tasks, for each student model, experiments are done with a combination of three parameters. The embeddings used are one of either byte-pair embeddings, hash embeddings, or character embeddings. The dimensionality of the embedding vectors are either 25, 100, or 300.

For QQP and MNLI, we limit the dimensionality of embedding vectors to either 25 or 100. Initial results showed that the benefit of going from 100 to 300-dimensional embeddings was negligible. Also, as outlined in section 6.2.1, the models for QQP and MNLI are larger than for SST-2. Given our focus of models with small size footprint, we felt using 300-dimensional embeddings would make the models too big. Finally, as training took a lot longer for QQP and MNLI, limiting the amount of variables to test was necessary due to time constraints.

The α value used during distillation is either 0, 0.5 or 1. As described in 6.3, α regulates how heavily the student model weighs the output of the teacher model vs. the labels in the training set. Each experiment is also run with and without bootstrapped data to record the effect of the extra synthetic data.

Additionally, we run each experiment four times with four different numerical seeds. These seeds are mainly used to randomly initialize weights in the models before training. As outlined in 4, when training neural networks, the initial values of weights can have a large impact on performance. To see the impact of randomly initialized weights, we run experiments with four seeds and report average accuracy. The specific seeds we use are 3201976, 8291959, 5111918, and 2211933.

We first run the above configurations for knowledge distillation alone. Then, for selected distilled models, we run experiments with pruning, quantization, and a combination of both.

6.2.1 Model Configurations

Table 6.1 shows hyperparameters for the different student models. These are the configurations that remain static for all experiments.

	BiLSTM	RNN	FFN
Batch Size	50*	50*	50
Optimizer	AdaDelta	Adam	Adam
Learning Rate	1	1e-4	1e-4
Encoder Dims.	150**	100**	-
Vocab Size	5000	5000	5000
Dropout	0.1	0.1	0.2
Weight Decay	0	1e-5	0
Classifier Dims.	200**	100**	100**

Table 6.1: Hyper parameters for the three student models. Vocab size is the amount of embedding vectors used.
*Batch size is 256 for QQP and MNLI.
**Values are doubled for QQP and MNLI.

6.2.2 Benchmark Tasks (GLUE)

We use tasks from the GLUE collection of NLP benchmark tasks (Wang et al., 2019). The data provided by GLUE is in English. We wanted to get an understanding of how effective a task-specific fine-tuned language model can be compressed for a given task. GLUE is used for evaluating models by RoBERTa (Liu et al., 2019), TinyBERT (Jiao et al., 2020), and many others in NLP research. This enables us to draw comparisons to results from our own experiments.

GLUE consists of 10 NLP benchmark tasks with associated training data. These cover different aspects related to what the authors call natural language understanding. We chose to focus on three of these tasks, namely SST-2, QQP and MNLI. As described in 2.3, GLUE defines tasks as being either single-sentence, similarity and paraphrase, or inference tasks. The three tasks we have chosen cover these three categories. These three tasks were used for evaluation in Tang et al. (2019). Since our models are inspired by those implemented in this paper, we choose to use the same tasks as well. Furthermore, these three tasks are those with most available training data within their respective categories.

Dataset	Task	Train	Aug-train	Dev
SST-2	sentiment	67K	1.05M	872
QQP	paraphrase	363K	1.43M	40K
MNLI	NLI	393K	1.52M	20K*

Table 6.2: Amount of sample sentences of data used by the three benchmark tasks during training and validation. Train refers to the training data provided by GLUE. Aug-train is the bootstrapped data generated by our data augmentation process. Validation is the dev sets provided by GLUE. *: matched and mis-matched datasets combined.

SST-2. Stanford Sentiment Treebank is a sentiment analysis task, which involves learning whether a given sentence emits a positive or negative sentiment (Socher et al., 2013).

QQP. Quora Question Pairs is a collection of question pairs from the website quora.com (Iyer et al., 2017). The goal is to determine whether two sentences are duplicate. In other words, whether one question is a paraphrase of the other.

MNLI. Multi-genre Natural Language Inference is another sentence pair task (Williams et al., 2018). Like QQP, each training example is two sentences. The goal is to determine whether the two sentences show entailment, are neutral, or are contradictory. The development and test set are split into two parts: One where the two questions in each sample are matching in terms of genre, and one where they are mismatched.

These three tasks offer different challenges to our implemented models. SST-2 generally sees higher average scores in other research, and should be the simplest task for the models to learn. QQP is a step up. It requires comparing two sentences and drawing some deeper meaning between the words in each. MNLI is the most difficult task, requiring a model to learn entailment and contradiction between sentences, or whether no relation exists between sentences.

An overview of the amount of training data available for each task is given in table 6.2. In section 6.2.4 we discuss the augmented train data in detail.

For validation we are using the Dev sets provided by GLUE. We do this as the provided GLUE Test data is unlabelled, and there is a limit to how often one can get performance accuracy from an API on their website.

6.2.3 Preprocessing

The first step before any fine-tuning or compression can be done, is to preprocess the data for the three selected GLUE tasks. The training data is downloaded from the GLUE benchmark hub. The data is then split into training, testing, and development datasets. Each of those datasets is further split into separate files for inputs and labels.

Byte-Pair embeddings are then downloaded. The Fairseq framework is used in conjunction with the embeddings to encode all the text in the data files. Fairseq is also used to run a final preprocessing step that builds vocabularies and binarizes the training data. After this, the data is ready to be used with the RoBERTa language model for fine-tuning for one of the three GLUE tasks.

6.2.4 Bootstrapping

We do knowledge distillation in the context of a teacher fine-tuned on a specific GLUE task. The knowledge in the teacher is then distilled into a student that is also made to solve the same specific NLP task. We exploit the knowledge of the teacher to generate extra augmented data used to bootstrap our models in distillation.

We use the data augmentation method by Jiao et al. (2020), as described in 4.1.3, when bootstrapping distillation models with additional training data. We use the same parameters for $N = 29$, $p = 0.4$ and $K = 15$. Some sentence samples are invalid for augmentation. The amount of augmented train data for each task can be seen in table 6.2.

The ratio of augmented data compared to the GLUE train data differs for each task. For SST-2 it is about 15.5x times the amount of samples in the original train data, for QQP and MNLI it is about 3.9x times. This discrepancy is deliberate. Given the large amount of GLUE train data for QQP and MNLI, we decided to only use 25% of the generated data for these two tasks. The amount presented in table 6.2 is after this reduction.

6.2.5 Pruning Sparsity Targets

For experiments in further compression, pruning is done in a post-training fashion using local unstructured pruning. We vary how aggressively we prune different parts of the distilled models. We do this because aggressively pruning certain parts degrades performance too strongly. We use Top V pruning with a desired sparsity of 50%. However, since we do local pruning, not all the models reach this sparsity. The target sparsity for individual weight matrices in the models are found in table 6.3.

Model part	Sparsity
BiLSTM	50%
RNN	10%
Linear	12.5%
Hash Emb.	0%
Character Emb.	30%
Byte-pair Emb.	40%

Table 6.3: Sparsity targets for different parts of the models during pruning.

These sparsity levels attempt to balance accuracy degradation with lower size. Initial experiments indicated that pruning too much lowers accuracy by a lot. Pruning too little does not lower the size of the models as desired. We found that the weights in the BiLSTM was the least susceptible to degradation in accuracy when pruned. The RNN on the other hand would quickly show decreased performance. This makes sense given the small amount of parameters in the RNN combined with its sequential nature. Pruning the weights can cause a domino effect because weights pertaining to earlier inputs in a sequence will affect the whole sequence. The linear classifier in each model was similarly susceptible to degradation when pruning.

The different embeddings are similarly treated differently. Byte-pair embeddings showed fewer signs of degradation when pruned. These are the embeddings with most parameters, so this makes sense. Hash embeddings are essentially a form of compressed embeddings. Further compressing them via pruning hurt performance too much. Hash embeddings are therefore left untouched in these experiments. Finally, character embeddings are also pruned more conservatively, due to their low amount of parameters.

We also experimented with pruning during training, to allow the network to adjust to weights being pruned. The hope was that by gradually increasing how many weights were pruned during training, the network could adjust and lower sparsity levels could be reached. In practice, the trade-off between accuracy and sparsity was difficult to balance during training. Models would either degrade completely or not get pruned enough to have any noticeable size reduction. A method better suited for pruning during training would be movement pruning as described in 4.2.3. However, as mentioned in 6.1.6, we did not manage to make this work with PyTorch.

6.3 Hardware Setup

An overview of the hardware used for running all experiments is given in table 6.4.

OS	CPU	GPU	Memory
CentOS Linux 7	Intel Core i7-4790 3.60 GHz 4 Cores	GeForce RTX 2070	32 GB

Table 6.4: Specifications for the hardware used during all our compression experiments.

Early, non-reported, experiments were run on multiple different GPU architectures. This caused issues with reproducibility due to differences in how certain CUDA operations work differently depending on the GPU. To ensure a higher degree of determinism, when using the same seed and setup, we made sure to run all experiments on the same hardware.

Still, certain aspects of our code can not be guaranteed to operate in a deterministic way when started with the same random seed, among them BiLSTM, RNN and EmbeddingBag (PyTorch, 2019h,c).

6.4 Metrics

We measure the performance of compressed models with two main metrics: accuracy, f1-score and size.

Accuracy. This is calculated as the ratio of correctly predicted samples from the development set for the given GLUE task. For MNLI the dev set is split into one where the genre for the sentence pair is matching, and one where it is mismatched. Following Wang et al. (2019) example, we report accuracy for MNLI as a the mean of these two.

F1-score. For the task QQP, an additional metric called f1-score is used. F1-score is used because the label distribution is skewed towards negative examples by 63% for QQP (Wang et al., 2019). F-1 score is calculated as:

$$F_1 = \frac{tp}{tp + \frac{1}{2}(fp + fn)} \tag{6.9}$$

What this formula gives is the so-called harmonic mean of recall and precision. Recall is the amount of true positives divided by all samples that should have been labelled positive. Precision is the amount of true positives divided by all samples predicted positive.

Size. We measure size of distilled models by saving them as PyTorch models to disk, and then report this size. We considered measuring memory usage, but decided against it. Mainly because memory usage can vary and would have to be measured as snapshots. Disk size, on the other hand, is a static reliable number.

When we do further compression on distilled models, we measure and report the size of all compressed models by zipping them using gzip.

Gzip uses the deflate algorithm (Feldspar, 1997) which in turn uses LZ77 compression (Ziv and Lempel, 1977) and Huffman encoding (Huffman, 1952) as described in 4.2.6. Essentially, this measures how effective pruning reduces size when an efficient storage method is used. In practice, the network would not be smaller at runtime, unless a way of efficiently storing sparse matrices were used.

7 Results

This section outlines the results of our model compression experiments.

7.1 Baseline Models

The performances and sizes of models presented in this section serve as comparisons to our results.

RoBERTa$_{\mathbf{Large}}$. RoBERTa (Liu et al., 2019) is selected because we used it as teacher for the all the distilled models. It was also used as part of the process of generating augmented data. The performance for this model is the average from four different fine-tunings we did, for each of the three tasks.

GLUE baseline models. Two GLUE baseline models are selected, one with GloVe embeddings and one with Elmo embeddings. Results for these two are from the paper by Wang et al. (2019). Size measurements are done by us, since this is not reported in their paper.

TinyBERT. Finally, we also include the TinyBERT model as a baseline model. This model is a distilled version of BERT, where the student still uses a transformer-based architecture (Jiao et al., 2020). Performance results and size measurements for this model are those reported by the authors.

Even though two of our student models take inspiration from models by Tang et al. (2019) and (Wasserblat et al., 2020), we chose not to include these as baseline models. Tang et. al provide performances on the GLUE test set, but not on the development set. Wasserblat et. al evaluate on SST-2, but not QQP or MNLI. It is also unclear whether they use the GLUE test set or dev set.

7.2 Distillation

Here, we present the results of experiments with knowledge distillation for the three selected NLP tasks.

7.2.1 SST-2

Results for distillation on SST-2 are found in table 7.1. Baseline models are included for comparison. The BiLSTM models generally perform best in terms of accuracy. Being designed for sequential data, and having more parameters to tune than the RNN, gives this model a clear edge. In terms of embeddings, byte-pair seems to be the clear choice across models. Byte-pair embeddings offer a strong representation of words and subwords, which help the models when training. The downside to these embeddings are their size. For most models, the increase from 100 to 300-dimensional byte-pair embedding offers little to no benefit. RNN models benefit most from the larger embedding vectors. For all BiLSTM models except the one using $\alpha = 0$, performance decreases when going from 100 to 300-dimensional byte-pair embeddings. For the FFN models, the models using distillation seem to benefit, but the others do not.

The use of hash embeddings lower the size footprint significantly. With embeddings of 300 dimensions, hash embeddings are several times smaller than byte-pair embeddings, of same dimensionality, at a cost of performance.

In terms of the role distillation plays, the results are a little more varied. RNN reaches the highest performance when using pure distilla-

Baseline model	P	S	Accuracy
RoBERTa$_{\text{Large}}$	3.6e+8	1 426	96.6
TinyBERT$_6$	6.7e+7	268	93.0
GLUE model/GLoVE	7.4e+8	2 950	87.5
GLUE model/Elmo	1.7e+8	684	91.5

Model	E	D	P	S	No distill		No distill + b		Alpha 0.5 + b		Alpha 0 + b	
					mean	sd	mean	sd	mean	sd	mean	sd
BiLSTM	Hash	25	3.0e+5	1.21	78.5	1.0	84.7	0.5	**85.9**	0.4	**85.9**	0.8
	Hash	100	4.3e+5	1.72	77.4	1.3	85.2	0.6	86.5	0.3	**87.0**	0.5
	Hash	300	7.7e+5	3.08	77.4	0.5	85.6	1.0	**86.6**	0.2	86.2	0.5
	BPE	25	4.0e+5	1.60	82.9	0.6	87.0	0.5	**89.1**	0.3	88.6	0.4
	BPE	100	8.6e+5	3.46	82.9	0.5	88.6	0.2	**89.4**	0.4	88.7	0.6
	BPE	300	2.1e+6	8.42	83.7	0.8	88.2	0.4	88.8	0.4	**89.2**	0.3
	Char	100	3.7e+5	1.47	72.9	8.7	**87.1**	0.4	85.1	0.5	84.9	0.4
RNN	Hash	25	7.3e+4	0.30	69.0	2.1	81.8	0.3	**84.0**	0.6	83.7	0.6
	Hash	100	1.3e+5	0.51	72.7	1.5	82.4	0.5	82.9	0.8	**83.5**	1.0
	Hash	300	2.7e+5	1.07	73.6	0.7	82.9	0.3	83.1	0.6	**83.2**	0.3
	BPE	25	1.7e+5	0.69	79.9	0.5	84.8	0.5	85.1	0.6	**86.0**	0.9
	BPE	100	5.6e+5	2.25	79.4	0.5	85.3	0.7	86.1	1.2	**86.8**	0.5
	BPE	300	1.6e+6	6.41	79.9	0.3	86.2	0.7	**87.4**	0.3	87.1	0.4
	Char	100	6.4e+4	0.26	79.1	0.9	83.3	1.0	84.1	0.5	**84.6**	0.9
FFN	Hash	25	3.0e+4	0.12	75.0	0.7	80.5	0.8	80.6	1.1	**81.2**	0.5
	Hash	100	7.5e+4	0.30	74.4	0.3	80.7	0.9	81.0	0.7	**81.6**	0.5
	Hash	300	2.0e+5	0.78	73.5	0.8	80.2	0.5	80.5	0.3	**81.8**	0.5
	BPE	25	1.3e+5	0.51	81.3	0.2	81.0	0.3	81.2	0.4	**81.4**	0.1
	BPE	100	5.1e+5	2.04	80.7	0.1	**82.1**	0.5	81.8	0.5	80.3	0.1
	BPE	300	1.5e+6	6.12	81.0	0.2	**81.9**	0.4	81.7	0.4	81.7	0.3
	Char	100	1.4e+4	0.06	**59.5**	0.5	57.7	0.3	58.3	0.4	57.5	0.1
	Hash*	25	1.7e+4	0.07	74.8	1.1	79.6	0.2	79.9	0.1	**80.2**	0.2

Table 7.1: Results for models trained on the Stanford Sentiment Analysis (SST-2) dataset. Performances written as **accuracy**. Performances reported in *mean* and *sd* (standard deviation) are measured in percentage and are from four runs with different seeds. E: Embedding type. D: Dimension of embedding vectors. P: Parameter count for the entire model. S: Size on disk, in Megabytes. b: Bootstrapped dataset. Bold: best mean across the four training methods for that combination of embedding type and dimension. *Vocab Size of 2500.

tion, with $\alpha = 0$, where only the knowledge from the teacher is taken into account. With BiLSTM, an equal contribution of training labels and teacher logits seem to generally provide the best results.

The use of extra bootstrapped training data clearly allows the models reach better accuracy. The only instance where this is not true is with the FFN model using character embeddings. However, these embeddings perform poorly compared to all other embedding types, so conclusions should not be drawn from this. As outlined in table 6.2, SST-2 is by far the benchmark task in our experiments with the smallest amount of data. Our results indicate that providing the models with bootstrapped data boosts performance significantly, whether the model is distilled or not.

As mentioned in section 6.1.3, the FFN model uses temperature softmax with a temperature of 3 when calculating distillation loss. The two other models use the mean-squared error between teacher and student logits without softmax activation.

Compared to baseline models, our models generally fall short. However, a few models surpass the accuracy of the GLUE GLoVE model. These include most of the BiLSTM models using augmented data and byte-pair embeddings. BiLSTM with 25-dimensional byte-pair embeddings and $\alpha = 0.5$ surpasses the GLUE GLoVE model by 1.6 percentage points. This is impressive, considering this model has a smaller parameter count by three magnitudes.

7.2.2 QQP

Results for distillation on QQP are in table 7.2. From this table we want to highlight several observations.

BiLSTM with character embeddings and $\alpha = 0.5$ has the highest performance overall with both accuracy and F-1 score. This is quite different from the SST-2 results, where all the BiLSTMs with byte-pair embeddings had a model that outperformed the char model. For RNN, char embeddings is outperformed by byte-pair embeddings, except when no distillation or augmented data is used. For FFN, character embeddings perform the worst. For sentence similarity problems, the results indicate that char embeddings can be quite powerful when used in combination with a model that takes sequential information into account. This is especially true when the sequential model has long-short term memory and not just short-term.

FFN with 100 dimensional byte-pair embeddings and $\alpha = 0$ has the second best performance. This model is almost on par with the BiLSTM char model with only 0.1 percentage points separating them in both accuracy and F-1 score. This is impressive when taking model size into account. This model is several times smaller than BiLSTM with char embeddings, so its performance is surprising. Even more surprisingly, this small model performs better than all BiLSTM model using byte-pair embeddings.

All models with hash embeddings have relatively weak performances. For SST-2, this was also the case for the BiLSTM and RNN models. This time it is also the case for the FFN models. Nevertheless, for models using hash embeddings, the FFN models has higher performance than both BiLSTM and RNN.

Not only does the byte-pair and hash embedding FFN models perform well, overall the FFN models models have significant lower standard deviation. This means that models of this type, when initialized with different random seeds, tend to converge around the same performance.

The benefits of distillation or bootstrapped data is not as clear as it was for SST-2. The BiLSTM models still benefit from distillation and extra training data, but for RNN and FFN the results are more varied. Generally, the models using byte-pair embeddings seem to benefit from synthetic data. This indicates that the more parameters a model has, the more it can take advantage of extra training samples.

None of the distilled models reach the performances set by the baseline models. The two best models, BiLSTM with char embeddings and FFN with 100-dimensional byte-pair, are close in accuracy. These are within 1 percentage point of the GLUE GLoVE baseline model. When it comes to F-1 score, the gap between our models and the baseline models is quite large. This may be due to a lack of quality in our augmented data, which is discussed in more detail in section 8.2.

7.2.3 MNLI

The results for distillation on MNLI are found in table 7.3. These results deviate quite a bit from the results for QQP and SST-2. In general, training using only original data with no distillation seems most effective.

These results indicates that MNLI is the most difficult of the three tasks. The models need to detect entailment, contradiction, or neutral relationships between sentences. This requires the encoding of more complex patterns. It is important to note that unlike MNLI, SST-2 and QQP only have two possible labels.

The parameter space of the models may be too small to encode the necessary information. This means that additional bootstrap data will provide no benefit. The models may not have the capacity to make use of the extra data. The fact that models using byte-pair embeddings are the

Baseline model	P	S	Accuracy
RoBERTa$_{\text{Large}}$	3.6e+8	1 426	92.2 / 89.6
TinyBERT$_6$	6.7e+7	268	91.1 / 88.0
GLUE model/GLoVE	7.4e+8	2 968	85.3 / 82.0
GLUE model/Elmo	1.7e+8	703	88.0 / 84.3

Model	*E*	*D*	*P*	*S*	No distill		No distill + *b*		Alpha 0.5 + *b*		Alpha 0 + *b*	
					mean	sd	mean	sd	mean	sd	mean	sd
BiLSTM	Hash	25	1.8e+6	7.10	75.9/65.4	0.2/0.5	76.4/64.6	0.2/1.1	**77.7/65.7**	0.2/0.7	77.5/64.5	0.2/1.0
	Hash	100	2.0e+6	7.97	75.8/64.0	0.2/1.4	76.3/64.3	0.2/1.1	**77.9**/65.6	0.2/0.9	77.8/**66.1**	0.3/1.0
	BPE	25	1.9e+6	7.49	81.6/75.7	0.4/0.8	82.1/75.3	0.4/1.1	82.9/75.6	0.5/0.8	**83.2/76.5**	0.4/0.7
	BPE	100	2.4e+6	9.71	83.0/76.6	0.2/0.5	83.2/76.7	0.2/0.7	**83.9/77.4**	0.5/0.9	83.5/77.2	0.5/0.9
	Char	100	1.9e+6	7.72	82.3/75.6	0.3/0.8	83.9/77.3	0.2/0.3	**84.7/78.5**	0.3/0.7	84.5/78.0	0.3/0.9
RNN	Hash	25	4.4e+5	1.76	73.9/**60.9**	0.1/0.6	73.7/56.9	0.1/0.3	**74.3**/56.0	0.2/0.3	74.2/55.0	0.1/0.4
	Hash	100	5.1e+5	2.03	74.0/**60.6**	0.3/0.7	73.8/56.7	0.2/1.1	**74.8**/57.2	0.3/1.3	74.8/56.6	0.3/1.5
	BPE	25	5.4e+5	2.15	78.1/68.4	0.4/1.5	78.6/70.2	0.2/1.1	78.7/70.4	0.4/0.7	**79.1/71.5**	0.4/0.8
	BPE	100	9.4e+5	3.77	78.0/69.0	0.7/2.3	79.0/71.6	0.6/0.7	80.0/**73.2**	0.3/1.1	**80.3**/72.4	0.2/1.3
	Char	100	4.4e+5	1.78	78.6/**69.3**	0.4/0.6	77.8/65.2	0.5/1.1	**78.6**/66.2	0.2/0.8	78.5/65.3	0.4/1.3
FFN	Hash	25	4.8e+4	0.20	77.2/**68.0**	0.1/0.3	77.1/65.1	0.2/0.5	77.1/65.3	0.1/0.3	**77.2**/64.4	0.2/0.3
	Hash	100	1.5e+5	0.59	**78.9/70.4**	0.1/0.3	78.2/66.8	0.1/0.3	78.3/67.2	0.1/0.3	78.3/66.6	0.0/0.2
	BPE	25	1.5e+5	0.58	81.2/75.2	0.1/0.2	82.6/**76.7**	0.1/0.2	82.6/76.7	0.0/0.1	**82.7**/75.8	0.1/0.2
	BPE	100	5.8e+5	2.33	84.0/78.3	0.1/0.4	84.6/78.4	0.1/0.3	84.6/**78.4**	0.1/0.3	**84.7**/78.1	0.1/0.2
	Char	100	8.4e+4	0.34	**73.2/61.0**	0.1/0.3	73.2/53.5	0.1/0.5	**73.2**/53.5	0.1/0.3	72.7/51.6	0.1/0.4
	Hash*	25	3.4e+4	0.14	**76.3/66.4**	0.1/0.3	75.7/61.6	0.2/0.7	75.7/61.7	0.2/0.4	75.9/61.4	0.0/0.2

Table 7.2: Results for models trained on the Quora Question Pairs (QQP) dataset. Performances written as **accuracy/f1-score**. Performances reported in *mean* and *sd* (standard deviation) are measured in percentage. *E*: Embedding type. *D*: Dimension of embedding vectors. *P*: Parameter count for the entire model. *S*: Size on disk, in Megabytes. *b*: Bootstrapped dataset. Bold: highest mean for that combination of embedding type and dimension. *Vocab Size of 2500.

only ones benefiting from bootstrapped data supports this hypothesis. These embeddings have a larger parameter space and can therefore make use of the extra data.

Another possible explanation is that, given MNLI's complexity, generating quality augmented data is more of a challenge. This may be another reason for the poor performance of models using bootstrapped data. When generating bootstrapped data we augment existing data. During this process, the meaning of a training example may be skewed or lost completely. See B for a more detailed analysis of the correctness of our bootstrapped data.

7.2.4 Takeaways

Compared to the baseline models, only a few of our models reach comparable performance. BiLSTM with byte-pair embeddings and using distillation are the only models to surpass the performance of the GLUE GLoVE model.

Figure 7.1 shows the distribution of parameters in the FFN model using 25-dimensional hash embeddings and a vocab size of 2500. This model is only made up of parameters in the embedding vectors and in the classifier layer. Even though the embeddings for this model are small, they still account for 83% of parameters in the model. In section 3.7, we showed parameter counts for the baseline models. Compared to this distilled model, the baseline models contain orders of magnitude more parameters in all layers.

Some models perform better when no distilla-

Baseline model	P	S	Accuracy
RoBERTa$_{\text{Large}}$	3.6e+8	1 426	90.3
TinyBERT$_6$	6.7e+7	268	84.5
GLUE model/GLoVE	7.4e+8	2 968	66.7
GLUE model/Elmo	1.7e+8	703	68.6

Model	E	D	P	S	No distill		No distill + b		Alpha 0.5 + b		Alpha 0 + b	
					mean	sd	mean	sd	mean	sd	mean	sd
BiLSTM	Hash	25	1.8e+6	7.10	**58.0**	0.1	53.4	0.5	55.5	0.3	55.6	0.5
	Hash	100	2.0e+6	7.97	**57.9**	0.3	53.3	0.2	55.6	0.6	55.7	0.7
	BPE	25	1.9e+6	7.49	65.4	0.1	65.6	0.3	66.9	0.3	**67.2**	0.6
	BPE	100	2.4e+6	9.71	65.8	0.2	66.2	0.4	**67.8**	0.5	67.7	0.2
	Char	100	1.9e+6	7.72	60.6	0.3	59.5	0.7	63.0	0.4	**63.1**	0.7
RNN	Hash	25	4.4e+5	1.76	**53.4**	0.3	49.3	0.3	48.6	0.3	48.6	0.3
	Hash	100	5.1e+5	2.03	**52.5**	0.3	48.9	0.2	48.4	0.3	48.3	0.3
	BPE	25	5.4e+5	2.15	**60.6**	0.5	58.2	0.8	56.3	1.5	56.8	1.3
	BPE	100	9.4e+5	3.77	**61.5**	0.3	59.7	0.5	60.0	1.0	58.8	0.4
	Char	100	4.5e+5	1.78	**53.0**	3.2	44.5	3.5	42.8	1.3	43.0	1.9
FFN	Hash	25	4.8e+4	0.20	**55.5**	0.2	52.5	0.1	52.5	0.2	51.6	0.1
	Hash	100	1.5e+5	0.59	**56.5**	0.1	53.9	0.3	53.8	0.2	53.5	0.1
	BPE	25	1.5e+5	0.59	60.7	0.1	61.1	0.1	**61.2**	0.1	60.6	0.3
	BPE	100	5.8e+5	2.33	62.9	0.1	63.7	0.0	**63.8**	0.1	63.4	0.0
	Char	100	8.4e+4	0.34	**48.7**	0.2	47.1	0.1	47.1	0.3	46.6	0.3
	Hash*	25	3.5e+4	0.14	**54.3**	0.2	50.9	0.3	50.9	0.2	50.3	0.2

Table 7.3: Results for models trained on the Multi-Genre Natural Language Inference (MNLI) dataset. Performances written as average of **matched accuracy/mismatched accuracy**. Performances reported in *mean* and *sd* (standard deviation) are measured in percentage. E: Embedding type. D: Dimension of embedding vectors. P: Parameter count for the entire model. S: Size on disk, in Megabytes. b: Bootstrapped dataset. Bold: highest mean for that combination of embedding type and dimension. *Vocab Size of 2500.

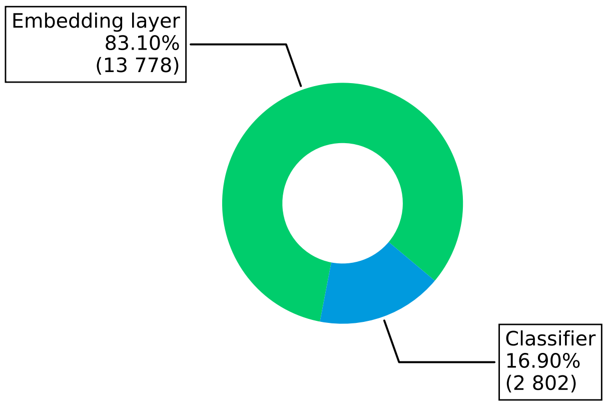

Figure 7.1: Distribution of parameters in the smallest distilled model: FFN with Hash 25 and Vocab 2500 for SST-2.

tion is done. For instance, on QQP and MNLI, the models with the lowest amount of parameters do not gain a significant advantage from being distilled. This includes most models using hash and character embeddings. This indicates that these models are too small to make use of the extra information provided by the soft labels in distillation. Similarly, these models gain limited or no benefit from bootstrapped data. Again, this would indicate that the models are too small to encode the intricacies of the data.

7.3 Further Compression

Table 7.4 shows distilled models we selected for further compression via quantization and pruning. These models are selected because they either excel in one of the tasks or show decent results across the different tasks given their size. In the next set of experiments, we investigate how performance is affected when we compress these models further.

Results for pruning and quantization of selected models are found in table 7.5, along with selected baseline models.

7.3.1 Pruning

The general trend for pruned models is that performance is reduced. The extent of the reduction seems to vary depending on embedding type, embedding dimensionality, and architecture. It should be noted that we prune embedding types and architecture types with different targets, as shown in table 6.3. The different targets were motivated by our initial investigation, where component with a larger parameter count tolerated more aggressive pruning.

Our sparsity targets for the largest modules, BiLSTM and Byte-pair encoded embeddings, might have been set too aggressively, if we wanted a similar performance change for all combinations of architecture, embedding type, and embedding dimension. For instance, three out of four BiLSTM model experiments in table 7.5 have an accuracy more than 1.5 percentage points lower on SST-2 after pruning. For QQP and MNLI tasks, the decrease is not as significant, except for the model $\text{BiLSTM}_{\text{BPE25}}$.

A few of the models with hash embeddings actually improve their performance after pruning: $\text{BiLSTM}_{\text{Hash100}}$ on MNLI; $\text{RNN}_{\text{Hash100}}$ on QQP and MNLI; $\text{FFN}_{\text{Hash25}}$ on QQP; and $\text{FFN}_{\text{Hash25*}}$ on all three tasks. We do not prune the hash embeddings at all. This indicates that modest pruning on layers following the embedding layer has the potential of increasing accuracy.

The gzip size of pruned models also reflect the different amount of sparsity targets we defined. Models $\text{FFN}_{\text{Hash25}}$, $\text{FFN}_{\text{Hash25*}}$ and $\text{RNN}_{\text{Hash100}}$ barely see any difference in size before and after pruning. On the other hand a model like $\text{BiLSTM}_{\text{BPE100}}$ has a size of about 69% and 74% after pruning, for single-sentence and sentence-pairs respectively.

7.3.2 Quantization

In general quantization affects performance in a slight negative way. However, $\text{BiLSTM}_{\text{Char100}}$ has higher performance on MNLI and $\text{RNN}_{\text{Char100}}$ on QQP. For a large majority of the models, the performance drop-off is less than a half percentage point.

Compared to pruning, the general effect on performance is not as negative when doing quantization. The exception to this is the FFN models on the tasks MNLI and QQP. This suggests that, quantifying all parameters on the small FFN models is harmful when the task is more complex than SST-2. Interestingly, it is more harmful than pruning away weights with a value close to zero.

We quantize all models to the same fixed-width representation, from 32-bit floating point numbers to 8-bit integers. The only exception is hash embeddings, where part of its component weights are only converted to 16-bit floating point numbers, as discussed in section 6.1.7.

Despite the fixed-width representation, the reported sizes resemble more how it would look had we done variable-width bit representation, as the files have been gzipped. This can be seen by the fact that some models get a size more than 4x times smaller after quantization. An example of this is the sentence-pair variant of $\text{BiLSTM}_{\text{Char100}}$ which goes from 7.175 MB to 1.403 MB, a reduction rate of more than 5x.

A few of the models reach or surpass the accuracy set by the GLUE baseline model with GloVe embeddings. The quantized version of $\text{BilSTM}_{\text{BPE100}}$ does it for SST-2 and MNLI, while being 3516x and 1471x smaller respectively. $\text{BilSTM}_{\text{BPE25}}$ does it for the same two tasks, with sizes 7302x and 1993x smaller. None of the our models for QQP reach the threshold, but the $\text{FFN}_{\text{BPE100}}$ model is the one that gets closest.

7.3.3 Pruning and Quantization

When combining pruning and quantization on distilled models, the sizes following a similar trend as shown for pruned models. Namely that BiLSTMs are the ones that gain the most significant reduction in size. A single pruned and quantized model gets a size that is slightly larger than its quantized counterpart, that is the sentence-pair version of $\text{FFN}_{\text{Hash25*}}$.

Name	Model	Embeddings	Embedding Dim.	Vocab Size	Alpha
BiLSTM$_{\text{Hash100}}$	BiLSTM	Hash	100	5000	0
BiLSTM$_{\text{BPE100}}$	BiLSTM	BPE	100	5000	0.5
BiLSTM$_{\text{BPE25}}$	BiLSTM	BPE	25	5000	0
BiLSTM$_{\text{Char100}}$	BiLSTM	Char	100	5000	0.5
RNN$_{\text{Hash100}}$	RNN	Hash	100	5000	0
RNN$_{\text{BPE100}}$	RNN	BPE	100	5000	0
RNN$_{\text{Char100}}$	RNN	Char	100	5000	0
FFN$_{\text{Hash25}}$	FFN	Hash	25	5000	0
FFN$_{\text{BPE100}}$	FFN	BPE	100	5000	1
FFN$_{\text{Hash25*}}$	FFN	Hash	25	2500	0

Table 7.4: Student models selected for further compression by pruning and quantization. All models are trained with bootstrapped with additional training data.

The smallest model presented is FFN$_{\text{Hash25*}}$. For SST-2, this model is small enough to fit in a 32KB L1 data cache found in a modern consumer-grade CPU. For QQP and MNLI, this model would fit in a 48KB L1 data cache, found in some recent consumer-grade CPUs. Naturally, the small size comes with a cost in accuracy. This model is far behind the accuracy of the large reference models. However, if speed and size is of utmost importance, this model is a good choice.

7.3.4 Model Efficiency

Figure 7.2 shows a Pareto curve, or skyline plot, of the performance of fully compressed and baseline models for SST-2. Fully compressed models have been distilled, pruned, and quantized. The y axis shows prediction accuracy and the x axis shows model disk size. Note that the x axis is logarithmic. The green line represents the *skyline* (Börzsönyi et al., 2001). The skyline represents the measured optimum between disk size and accuracy. The models below the skyline are dominated by models with a smaller size and better accuracy.

This plot can be used to quickly visualize whether a model is the optimal choice under certain constraints. For example, given an embedded device where a model has to work within certain size constraints. Using the plot, the model with highest accuracy smaller than the size constraint can quickly be identified. In such a scenario, this model is the optimal choice when it comes to an accuracy/size trade-off.

All the BiLSTM models are on the skyline and therefore dominate all model with larger size, but lower performance. The same is true for the two FFN models using 25-dimensional hash embeddings and the RNN model using character embeddings.

The same type of plot is shown for QQP in figure 7.3. This figure includes plots for both accuracy and F1-score. Compared to the plot for SST-2 shown in figure 7.2, the difference in which models are optimal is drastic. For both accuracy and F1-score, all FFN models are on the skyline and perform surprisingly well. The only BiLSTM model on the skyline is BiLSTM$_{\text{Char100}}$. For F1-score this model is outperformed by FFN$_{\text{BPE100}}$, leaving no BiLSTM models as optimal choices. In both plots RNN$_{\text{Char100}}$ is the only RNN model on the skyline. It is interesting that both the models using character embeddings perform well. With both accuracy and F1-score, RNN$_{\text{Char100}}$ performs worse than RNN$_{\text{BPE}}$. BiLSTM$_{\text{Char100}}$ performs better than every other BiLSTM model and achieves highest performance of all distilled models when it comes to accuracy.

The final pareto plot for MNLI is found in figure 7.4. Similar to the plots for QQP in figure 7.3, all FFN models are on the skyline.

Unlike with QQP, BiLSTM models using byte-

Model	SST-2	QQP	MNLI	Size (SS/SP)		Compr. rate (SS/SP)	
RoBERTa$_{\text{Large}}$	96.56	92.15 / 89.58	90.33	1 426	1 426	1x	1x
TinyBERT$_6$	93.0	91.1 / 88.0	84.5	268	268	5x	5x
GLUE + GLoVE	87.5	85.3 / 82.0	66.7	2 950	2 968	0.5x	0.5x
GLUE + Elmo	91.5	88.0 / 84.3	68.6	684	703	2x	2x
BiLSTM$_{\text{Hash100}}$	86.96	77.82 / 66.06	55.66	1.587	7.384	898x	193x
BiLSTM$_{\text{Hash100}}$ + p	85.01	77.26 / 64.47	55.89	1.121	5.678	1 271x	251x
BiLSTM$_{\text{Hash100}}$ + q	86.55	77.80 / 65.99	55.64	0.395	1.599	3 611x	891x
BiLSTM$_{\text{Hash100}}$ + p + q	84.92	77.27 / 64.44	55.80	0.313	1.339	4 553x	1 064x
BiLSTM$_{\text{BPE100}}$	89.36	83.90 / 77.43	67.78	3.207	8.990	444x	158x
BiLSTM$_{\text{BPE100}}$ + p	87.59	83.14 / 76.07	66.74	2.186	6.727	652x	211x
BiLSTM$_{\text{BPE100}}$ + q	89.08	83.82 / 77.30	67.74	0.839	2.006	1 699x	710x
BiLSTM$_{\text{BPE100}}$ + p + q	87.50	83.15 / 76.07	66.65	0.725	1.716	1 965x	830x
BiLSTM$_{\text{BPE25}}$	88.65	83.16 / 76.47	67.24	1.479	6.936	964x	205x
BiLSTM$_{\text{BPE25}}$ + p	85.67	79.92 / 67.79	65.07	1.009	5.358	1 413x	266x
BiLSTM$_{\text{BPE25}}$ + q	88.33	83.08 / 76.35	67.18	0.404	1.480	3 528x	963x
BiLSTM$_{\text{BPE25}}$ + p + q	85.69	79.88 / 67.75	65.01	0.337	1.258	4 235x	1 133x
BiLSTM$_{\text{Char100}}$	85.06	84.72 / 78.48	63.01	1.360	7.175	1 048x	198x
BiLSTM$_{\text{Char100}}$ + p	84.23	83.84 / 76.61	62.69	1.103	6.231	1 293x	228x
BiLSTM$_{\text{Char100}}$ + q	84.89	84.63 / 78.37	63.03	0.329	1.403	4 332x	1 016x
BiLSTM$_{\text{Char100}}$ + p + q	84.09	83.81 / 76.55	62.72	0.291	1.287	4 901x	1 107x
RNN$_{\text{Hash100}}$	83.46	74.76 / 56.56	48.27	0.464	1.862	3 072x	765x
RNN$_{\text{Hash100}}$ + p	83.08	75.17 / 60.87	48.85	0.450	1.742	3 170x	818x
RNN$_{\text{Hash100}}$ + q	83.40	74.65 / 56.11	48.26	0.130	0.419	10 974x	3 402x
RNN$_{\text{Hash100}}$ + p + q	83.03	75.04 / 60.82	48.74	0.128	0.411	11 116x	3 473x
RNN$_{\text{BPE100}}$	86.75	80.26 / 72.44	58.82	2.094	3.485	680x	409x
RNN$_{\text{BPE100}}$ + p	85.21	78.74 / 72.16	57.52	1.496	2.798	953x	509x
RNN$_{\text{BPE100}}$ + q	86.55	80.14 / 72.21	58.74	0.548	0.833	2 603x	1 712x
RNN$_{\text{BPE100}}$ + p + q	85.21	78.64 / 72.04	57.47	0.497	0.791	2 866x	1 802x
RNN$_{\text{Char100}}$	84.58	78.48 / 65.27	42.99	0.236	1.640	6 046x	869x
RNN$_{\text{Char100}}$ + p	83.26	78.45 / 67.16	41.91	0.219	1.516	6 507x	940x
RNN$_{\text{Char100}}$ + q	83.89	78.59 / 65.99	42.90	0.053	0.299	27 011x	4 762x
RNN$_{\text{Char100}}$ + p + q	83.43	78.36 / 67.50	42.59	0.051	0.296	27 869x	4 817x
FFN$_{\text{Hash25}}$	81.22	77.20 / 64.40	51.57	0.113	0.179	12 590x	7 979x
FFN$_{\text{Hash25}}$ + p	80.36	77.40 / 66.21	51.51	0.113	0.173	12 673x	8 246x
FFN$_{\text{Hash25}}$ + q	81.02	76.66 / 62.43	50.43	0.048	0.062	29 406x	23 137x
FFN$_{\text{Hash25}}$ + p + q	80.42	76.98 / 64.44	50.39	0.048	0.061	29 506x	23 284x
FFN$_{\text{BPE100}}$	82.05	84.61 / 78.38	63.72	1.898	2.162	751x	659x
FFN$_{\text{BPE100}}$ + p	78.61	83.59 / 77.73	63.40	1.335	1.578	1 068x	903x
FFN$_{\text{BPE100}}$ + q	81.94	84.29 / 77.22	62.13	0.531	0.581	2 686x	2 452x
FFN$_{\text{BPE100}}$ + p + q	78.70	83.41 / 76.59	61.89	0.494	0.546	2 887x	2 612x
FFN$_{\text{Hash25*}}$	80.16	75.93 / 61.41	50.33	0.063	0.129	22 761x	11 047x
FFN$_{\text{Hash25*}}$ + p	80.22	76.29 / 63.81	50.38	0.062	0.123	23 054x	11 589x
FFN$_{\text{Hash25*}}$ + q	79.99	75.55 / 59.77	49.38	0.027	0.041	52 792x	34 395x
FFN$_{\text{Hash25*}}$ + p + q	80.16	75.99 / 62.35	49.47	0.027	0.042	53 091x	34 172x

Table 7.5: Results for pruning (p) and quantization (q) of selected distilled models. Performances are measured by accuracy, accuracy/f1-score, matched/mismatched accuracy for the three tasks, respectively. Size of original models is disk size in MB. Size of quantized/pruned models is disk size in MB of zipped model using gzip deflate algorithm. Compression ratio is single sentence/sentence pair models compared to the disk size of teacher model RoBERTa Large. *SS*: Single Sentence. *SP*: Sentence Pair.

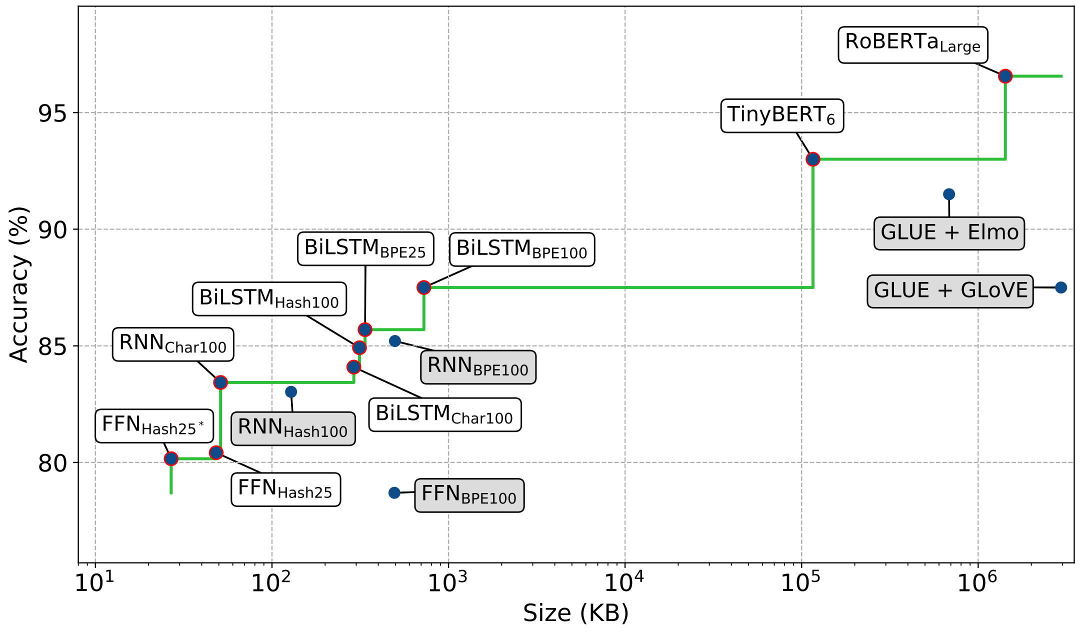

Figure 7.2: Pareto curve showing the relationship between accuracy and size of models trained on the SST-2 task. Grayed out models are those not on the skyline.

pair embeddings are also on the skyline and has the highest accuracy of all distilled models. For MNLI, no RNN models are optimal.

It seems clear that larger models perform better. However, as the skyline plots in figure 7.2, 7.3, and 7.4 show, the relationship between size and accuracy is not linear.

7.4 Additional Experiments

In section 6.1.4 we describe how we treat the output from BiLSTM and RNNs. That is, we use the last hidden state as the input for the following FFN classification layer. This was motivated by the fact that both Tang et al. (2019) and Wasserblat et al. (2020) did it this way. Initial tests we did for SST-2 also showed positive results. However, Wang et al. (2019) used a max pooling on all the hidden states.

Still, we were curious why the sequential models, BiLSTM and RNN, performed below expectations on QQP and MNLI. We set out to do a limited additional experiment where we also trained a few select sequantial models with the Wang et al. (2019) method for treating the BiLSTM or RNN layer output. Table 7.6 shows the results for these experiments.

Model	E	SST-2	QQP	MNLI
BiLSTM w/ last	Hash	87.0	77.8/66.1	55.7
BiLSTM w/ max		85.9	79.3/67.6	58.5
BiLSTM w/ last	BPE	88.7	83.5/77.2	67.7
BiLSTM w/ max		88.1	85.6/79.0	70.2
RNN w/ last	Hash	83.5	74.8/56.6	48.3
RNN w/ max		81.9	77.0/62.7	52.4
RNN w/ last	BPE	86.8	80.3/72.4	58.8
RNN w/ max		85.5	83.4/75.3	66.6

Table 7.6: Distillation experiment on using different inputs to classifiers. All models trained with alpha value of 0 and embedding dimensions of 100. Results are mean performance on dev sets, from four runs each. SST-2 is accuracy. QQP is accuracy/F1-score. MNLI is averaged accuracy for mismatched and matched sets. *E*: Embedding type.

The results are quite interesting. For SST-2 these results reaffirm our initial tests that using the last hidden state in sequential models lead to better performance. For QQP and MNLI the results indicate the opposite.

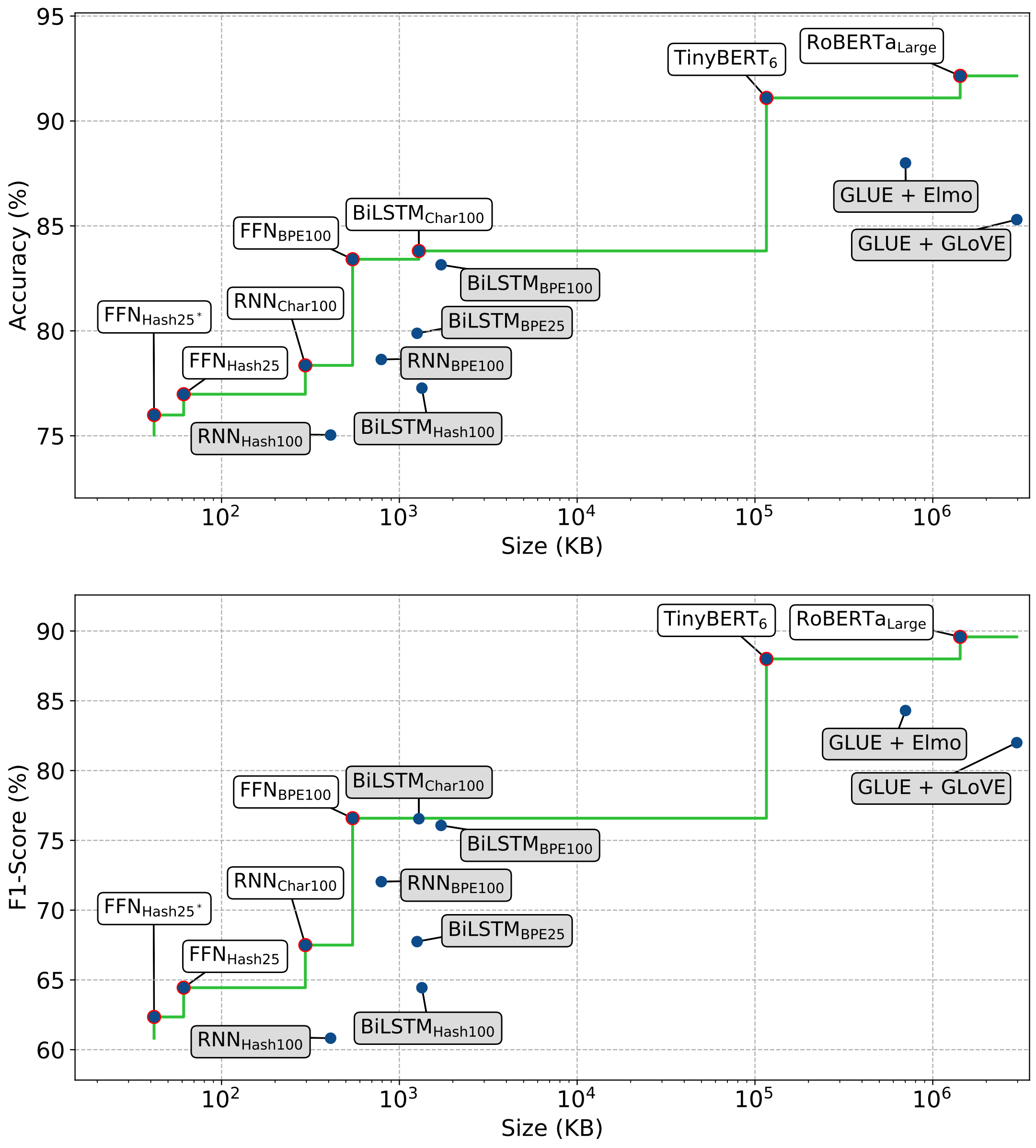

Figure 7.3: Pareto curve showing the relationship between accuracy and size (top plot) as well as F1-score and size (bottom plot) of models trained on the QQP task. Grayed out models are those not on the skyline.

Using max pooling on all hidden states, the BiLSTM model with byte-pair embeddings bumps its QQP accuracy by 2.1 percentage points, and F1-score by 1.8. With hash embeddings accuracy and F1-score are each increased 1.5 percentage points. For MNLI the BiLSTM with byte-pair, accuracy is increased by 2.1. For hash embeddings, it is increased by 2.8. The BiLSTM with byte-pair achieve a higher accuracy than the GLUE baseline model with GloVe

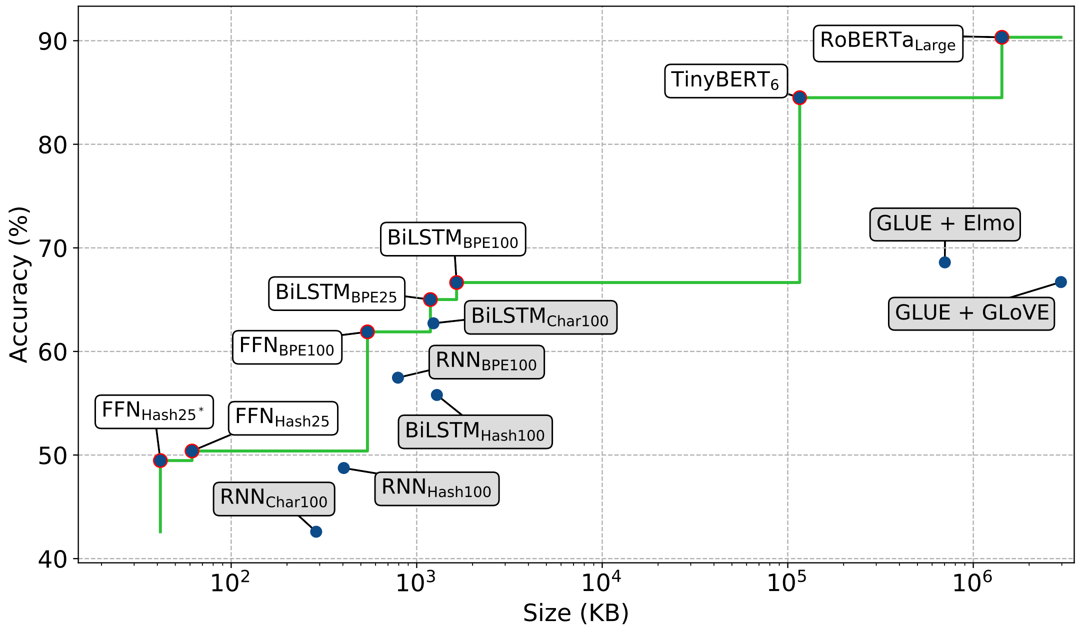

Figure 7.4: Pareto curve showing the relationship between accuracy and size of models trained on the MNLI task. Grayed out models are those not on the skyline.

on QQP, but not a better F1-score. On MNLI this model achieves better accuracy than both GLUE baseline models.

The RNN model with byte-pair embeddings increases its QQP accuracy by 3.1 percentage points, and F1-score by 2.9. Using hash embeddings, accuracy is increased by 2.2 and 6.1 for F-1 score. Using byte-pair embeddings, the MNLI accuracy is increased by an impressive 7.8. For hash embeddings, MNLI accuracy is increased by 4.1.

The RNN models do not achieve better performance than the GLUE baseline models. Still, the increase in performance is significant. This indicates that a sequential model like RNN, which does not have long-term memory, greatly benefits when its output takes all hidden states into account.

7.5 Discussion

We have presented results for compression experiments on neural network models within NLP. We set out to investigate how model size and performance relates when the former is reduced.

Our first research question dealt with the relationships between performance and size. In table 7.5 we see a clear drop-off in performance when size is reduced. Some techniques have a more severe impact than others. Among the models in this table, distillation achieves reductions in the range of 158x to 22761x for our largest and smallest student models respectively. This is a drastic reduction in size compared to quantization and pruning. The former giving additional reductions of around 4x, the latter in the range $1 - 2$x

Distillation sees a drop off in the range of approximately 7 - 47 percentage points. Quantization never degrades performance by more than 2 percentage points. On the other hand, pruning degrades performance by up to about 9 percentage points. In a few cases, pruning improves performance by up to roughly 4 percentage points. However, it should be noted that this only happens for RNN$_{\text{Hash100}}$, which has barely been pruned. We prune after distillation, which means the models are already small from the beginning. This makes pruning less effective as fewer redundant parameters are present in the models.

The largest student model architecture in our experiments is BiLSTM. This is the best performing architecture on the SST-2 and MNLI tasks. For QQP the result are less clear. Compared to other embeddings, byte-pair consistently perform better across all architectures and tasks. These embeddings have the largest vocabulary size of all those tested in our experiment. However, on SST-2, the BiLSTMs with 300-dimensional byte-pair embeddings achieved lower accuracy than those with 100-dimensions embeddings. This is seen in table 7.1.

Compared to the GLUE baseline model with GloVe embeddings, several of our experiments performed better in terms of both size and performance. Namely, the quantized $\text{BiLSTM}_{\text{BPE100}}$ and $\text{BiLSTM}_{\text{BPE25}}$ on SST-2 and MNLI as seen in table 7.5. This indicates that having more parameters in a model is not always translated into higher performance. In other words, the relationship between performance and size is not linear.

Our second research question is about finding finding the limit for compression while achieving tolerable performance. In our experiments we have shown that our BiLSTMs distilled from RoBERTa and using pre-trained byte-pair encoded embeddings are able to reach higher accuracy than the GLUE baseline with GloVE on SST-2 and MNLI. For the quantized $\text{BiLSTM}_{\text{BPE25}}$ models, this was done while having sizes 3528x and 963x smaller than the teacher.

Performance on QQP and MNLI could have been significantly better, had we done max pooling on BiLSTM and RNN hidden output states. The additional experiments we showed in section 7.4 indicates this.

In addition to distillation and quantization, we also did various degrees of pruning. Where we did least amount of pruning performance was around the same, and in some cases better. Where we did it most aggressively performance only went downwards. The most aggressive pruning was carried out on the largest models. Prior to pruning, these models were the only ones to have better performance than the GLUE baseline model with GLoVE embeddings on SST-2 and MNLI. After pruning, this was no longer the case.

None of the RNN and FFN reach the performances set by the GLUE baseline models for any tasks. This indicates that these models contain too few parameters or are structured in such a way that they are unable to model the data sufficiently well.

8 Broader Impact

This section describes the potential real-world impact that our findings have. First, we describe the practical applications of compressed models that we have experimented with in this paper. Then, we describe possible pitfalls associated with using such models in a real-world setting.

8.1 Applications

Resource-constrained domains. In certain domains, having smaller models is more important than having highly accurate ones. Such domains might include resource constrained hardware such as hearing aids, smartphones, and Internet-of-Things devices. Within these domains, shrinking model size might be necessary. even at the cost of sacrificing model performance. If a model is too large to fit within the size constraints set by the domain, then using a smaller but less accurate model is the only choice.

An example of a resource constrained domain is explored by Fedorov et al. (2020). They conducted research for Bose and ARM into creating an RNN model for speech enhancement. The model they trained had to take up less than 500KB on disk and fit within 320KB of working memory. This is similar to the size some of our models reach.

A broad overview of machine learning in embedded applications is presented by Ajani et al. (2021). They present a selection of resource constrained domains where machine learning has been utilized. Examples of applications include facial recognition and image classification on mobile devices using deep neural networks. Within NLP, networks have been used for speech recognition on mobile and embedded devices.

In the past, the majority of techniques employed have been non-neural. In recent years, neural network based approaches have repeatably pushed the boundaries of performance within many areas of NLP. As more research is conducted into compression of networks, utilizing such networks in resource constrained domains becomes more of a reality (Ajani et al., 2021).

The models we present are implemented in the widely used machine learning framework PyTorch. PyTorch offers various options for serving models in a production setting (PyTorch, 2020).

Financial and environmental cost. Another context where our research is relevant, is when the economical cost of running inference is of importance. Training and running inference on a large language model can be costly. Cost is tied to energy used during training and inference. This is expensive, both in terms of the financial cost of power and the environmental impact. Strubell et al. (2020) estimates costs of training language models, such as BERT and GPT-2. Training BERT$_{\text{base}}$ with 110 million parameters (Devlin et al., 2019) on a GPU produces an estimated 1438 lbs (652 kg) of CO_2 emissions. This is roughly equal to a trans-atlantic flight (Strubell et al., 2020). The estimated financial cost lies in the range of \$3,751-\$12,571.

In the process of conducting our experiments, we have used hundreds of hours of compute time for training and distilling different models. However, once a model has been trained and distilled, the cost of using it for inference is much lower. If the cost of distillation is too high, it may be more efficient to keep using the larger model for inference. If the model is to be used for inference for month at a time without large alterations, then the cost of distillation may be worth it.

8.2 Potential pitfalls

Runtime cost. Any type of compression involves certain trade-offs, which generally involve size, speed, and memory usage (Matt Mahoney). The focus of this thesis has been on the reduction of the disk size of neural networks models. We have not taken runtime speed and memory usage into consideration. However, some of the techniques used for compression introduces both memory and computational costs during runtime. For instance, when using dynamic quantization, the intermediate result from activation functions is in singe-precision floating point number, and additional operations are required for quantizing these numbers (PyTorch, 2019f).

Model bias. An issue with machine learning in general is that of reproduction of potential bias from training data. This is investigated by Bender et al. (2021) in the context of large language models. A model like RoBERTa is trained on vast quantities of data. Having a complete overview of the content in the dataset is near impossible. The model might unintentionally learn and reproduce texts that contain racism, homophobia, misogyny, etc.

Similar to how a language model approximates the data used in training, so does distilled models approximate the behavior of its teacher model. Bias from the original training data might trickle down to the student model. However, Vig et al. (2020) showed that distilled versions of GPT-2 exhibit lower bias than its teacher on the Winobias and Winogender datasets, but not for the Professions dataset.

Bootstrapped data. A possible issue brought up in section 7.2.2 and 7.2.3 is whether the quality of augmented data is good enough for QQP and MNLI. These tasks are more complex and creating augmented data can introduce errors or create nonsensical training examples. In appendix B, we investigate the distribution of labels in the original training data compared to the augmented data. We have not further investigated whether data augmentation produces semantically sound examples. With sentence pair tasks, changing one of the sentences may change or obfuscate its meaning. This may explain why more examples are labeled as *Not Duplicate* for QQP and *Neutral* for MNLI. The set of pairs of sentences that are related to each other is naturally outnumbered by the set of unrelated sentences. When augmenting words in sentences, it would make sense that the resulting sentences are more likely to be unrelated to each other.

9 Summary & Conclusion

This master's thesis investigates the relationship between performance and size, when the latter is reduced. We also investigate the extent to which neural network models can be compressed on different NLP tasks, while maintaining a tolerable performance. We define tolerable performance as that achieved by the various GLUE baseline models (Wang et al., 2019).

In chapter 2 we outline key concepts within language modeling and different aspects of understanding natural language. We describe how, in recent times, neural networks have been used as large language models, among them GPT-2, BERT, and RoBERTa. These language models excel at solving tasks within NLP, exemplified by their performance on the GLUE benchmark set. Finally, this chapter concludes with an introduction NLP benchmark tasks, specifically those present in GLUE.

This is followed by chapter 3, which describes fundamentals within neural networks. We transition into topics relevant for neural networks within NLP. First, we describe how textual data can be transformed into a kind suitable for neural networks. Then, we explain different types of networks developed to accommodate the sequential nature of linguistic utterances. Finally, we examine the size complexity of select neural network models trained on the GLUE benchmark set. These are RoBERTa and two GLUE baseline models that use different embeddings.

In chapter 4, we begin with an outline of the basic theory of compression, and the distinction between lossless and lossy compression. We then describe three major approaches to compress neural network models: Knowledge distillation, pruning, and quantization. We describe how knowledge distillation can be used to transfer behavior from a large model to a smaller one. We outline different methodologies for pruning, namely magnitude, top-v, and movement pruning. Finally, we describe the theory around quantization, with a focus on scalar quantization.

Before describing our own experiments with model compression, chapter 5 gives an overview of related work. Mainly, efforts within neural network model compression. We describe different approaches for compressing the BERT language model, and how our investigation differs.

Following, chapter 6 details our experimental setup. We explain the software setup and implementations underlying our experiments. We then present the workload chosen for the experiments. Namely, we investigate knowledge distillation for three different model architectures of different size complexity. These are BiLSTM, RNN and a simple FFN. The architectures each use a combination of three different embeddings: Byte-pair encoded embeddings, Char, and Hash. For each embedding type we also test various dimensions. Furthermore, we vary how distillation is used during training or whether it is used at all. We set out to test combinations of these workloads to get a broad understanding of how compression affects NLP models of various size complexity. In addition to distillation, we also present how we set out to do pruning and quantization on distilled models. These workloads are each tested on SST-2, QQP, and MNLI from the GLUE benchmark set. These tasks are selected to be semantically different from each other, ensuring the models are tested on different aspects of NLP. We describe the hardware on which our experiments are run, as well as concerns about reproducibility. Finally, we discuss the metrics chosen to measure results from experiments. We measure accuracy for all tasks as well as F1-score for QQP. In terms of the size of models, we measure disk size as well as zipped size.

In chapter 7, we present and analyze the results of our experiments. We outline baseline models chosen as a reference for performance and size. The first set of experiments we conduct is knowledge distillation, followed by pruning and quantization. We also show additional experiments for the sequential student model. The chapter finish with a discussion of findings in regards to our research questions.

In chapter 8, we bring up potential real-world applications that our research has. We shine light on the economical costs of large models and what benefit small distilled models might have. The chapter ends with a discussion of potential pitfalls and shortcomings related to both large neural language models and our approaches to compression.

9.1 Conclusion

In light of the results presented in chapter 7, we now return to our research questions from chapter 1.

RQ1: *What is the relationship between model performance and size complexity when the latter is reduced?*

For the task SST-2, we observe a general trend when size complexity of a target model decreases, so does the performance. Training a model with additional augmented data greatly improves performance. Using some degree of distillation further improves performance.

For the two sentence-pair tasks, QQP and MNLI, we observe different general patterns than for SST-2. For QQP, the largest and smallest architectures, BiLSTM and FFN, performs about the same. RNN is the worst performing. Overall, BiLSTM and RNN benefits, in a very modest fashion, from additional training data. This is not the case for FFN. BiLSTM and RNN also sees a small positive gain from distillation, whereas it is inconclusive for FFN.

For the MNLI task, BiLSTM is the best performing architecture, followed by FFN and then RNN. The overall trend is that bootstrapping with additional training data, and doing distillation, is harmful to performance. The exception to this is BiLSTM and FFN with byte-pair embeddings.

Across all tasks, pruning and quantization operates in a similar manner. Modest pruning does not affect performance in a major way, but likewise does also not decrease size by a lot. More aggressive pruning comes with the cost of a few percentage points in performance. Quantization affects performance less than pruning, and gives a decreases size to a larger extent.

In conclusion, the relationship between performance and size across our different experiment configurations highly depends on the task. The effects of pruning and quantization are task-agnostic.

RQ2: *To what extent can models, fine-tuned for different GLUE benchmark tasks, be compressed while maintaining tolerable performance?*

For distillation, we find that only BiLSTM with 25- or 100-dimensional byte-pair embeddings, BiLSTM$_{\mathrm{BPE25}}$ and BiLSTM$_{\mathrm{BPE100}}$, surpass the performance of the GLUE baseline model with GloVe embeddings on the tasks SST-2 and MNLI. Compared to the RoBERTa teacher, the SST-2 version of these models achieve a reduction in size of 964x and 444x, respectively. The MNLI version are reduced by ratios of 205x and 158x.

Pruning degrades performance for both BiLSTM$_{\mathrm{BPE25}}$ and BiLSTM$_{\mathrm{BPE100}}$ to an extent where they no longer surpass the performance of the GLUE baseline model using GLoVE embeddings, on either SST-2 or MNLI.

Quantization offers roughly a further 4x decrease in size, at little cost to performance. Performance degradation is smaller than what we see for pruning. The distilled BiLSTM$_{\mathrm{BPE25}}$ and BiLSTM$_{\mathrm{BPE100}}$ maintains tolerable performance after quantization.

With distillation and quantization, BiLSTM$_{\mathrm{BPE100}}$ achieves a combined compression ratio of 1699x for single-sentence tasks, and 710x for sentence-pair tasks. BiLSTM$_{\mathrm{BPE25}}$ achieves ratios 3528x and 963x.

A Distillation Optimization Target

Task	Emb	Dim	Alpha 0.5	Alpha 0
SST-2	Hash	25	80.1 / 80.6	79.9 / 81.2
	BPE	100	80.6 / 81.8	80.2 / 80.3
QQP	Hash	25	76.6 / 77.1	76.4 / 77.2
	BPE	100	84.7 / 84.6	84.7 / 84.7
MNLI	Hash	25	50.7 / 52.5	50.2 / 51.6
	BPE	100	62.6 / 63.8	62.4 / 63.4

Table A.1: Results for distillation of different FFN models with raw logits and with temperature softmax activation with a temperature of 3. Results are accuracy in percent for raw logits / temperature softmax. F1 accuracy for QQP is excluded for brevity.

Task	Emb	Dim	Alpha 0.5	Alpha 0
SST-2	Hash	25	89.4 / 88.0	88.7 / 88.2
	BPE	100	85.9 / 85.4	85.9 / 83.9
QQP	Hash	25	77.7 / 77.2	77.5 / 77.4
	BPE	100	83.9 / 83.0	83.5 / 81.1
MNLI	Hash	25	55.5 / 53.6	55.6 / 53.2
	BPE	100	67.8 / 63.8	67.7 / 63.8

Table A.2: Results for distillation of different BiLSTM models with raw logits and with temperature softmax activation with a temperature of 3. Results are accuracy in percent for raw logits (left) / temperature softmax (right). F1 accuracy for QQP is excluded for brevity.

Table A.1 shows results of experiments with different soft targets for selected FFN models. The left results show accuracy when using raw logits as soft targets during distillation. The right results show accuracy when using temperature softmaxed logits with a temperature of 3. The latter achieves noticeably better results across different model configurations and tasks.

We believe this is due to the small number of parameters that this model has. Without softmax, the distillation loss would be of greater magnitude and with more variance. This meant that adjustments to few weights could impact performance heavily. With temperature softmax, the adjustments to weights would be less drastic, and the model displayed better performance.

Table A.2 shows the same experiment conducted for different BiLSTM models. Interestingly, the results are reversed. Using pure logits as soft targets performs better across the board than using softmaxed logits.

B Quality of Bootstrapped Data

SST-2		
Positive	*Negative*	-
44.8 / 42.5	55.2 / 57.5	-
QQP		
Duplicate	*Not Duplicate*	-
63.0 / 78.3	37.0 / 21.7	-
MNLI		
Entailment	*Neutral*	*Contradiction*
33.3 / 19.6	33.3 / 40.3	33.3 / 40.3

Table B.1: Label distributions for SST-2, QQP, and MNLI. All values are percentages. Values are for original data (left) / bootstrapped data (right).

Table B.1 shows distributions of labels for the three tasks. For each label, the distribution for the original and bootstrapped dataset are shown, respectively. For all three tasks, the distributions for the bootstrapped data deviates from the labels in the original dataset. SST-2 deviates the least, which makes sense given the relative simplicity of this task.

C Program Guide

We created a program written in Python, that would facilitate the execution of various experiments with model compression. The code for this program is available at `https://github.com/pocketML/lil_bobby`.

There are five main files for running different aspects of this program:

- `finetune.py` - Download and finetune a RoBERTa language model for a given GLUE task
- `preprocess.py` - Preprocess GLUE data, do data augmentation, generate distillation loss
- `compress.py` - Run model compression. Supported techniques are knowledge distillation, pruning, and quantization
- `evaluate.py` - Evaluate the performance of finetuned or distilled models on SST-2, QQP or MNLI
- `analyze.py` - Gather various information about finetuned or distilled models, such as size, parameter distribution, etc.
- `experiment.py` - Run experiments with any of the above modules. This can be run as a pipeline. F.x. running finetuning, followed by distillation and pruning, followed by an evaluation and analysis of model performance and size

C.1 Fine-tuning

This module enables the downloading and finetuning of a pre-trained language model. Table C.1 contains a selection of arguments for `finetune.py`.

Argument	Explanation
--task	GLUE task to finetune for.
--arch	Pre-trained RoBERTa model . (`roberta_base` or `roberta_large`)
--batch-size	Batch size used for fine-tuning.
--max-epochs	Max fine-tuning epochs.
--cpu	Whether to run on CPU.
--fp16	Whether to train with 16-bit floats.
--seed	Seed for weight initialization.

Table C.1: Program arguments for fine-tuning a pre-trained language model.

C.2 Preprocessing

This module is used for doing various preprocessing tasks. One of these is running preprocessing for GLUE tasks to make them fit for finetuning with RoBERTa. The other task is data augmentation for creating bootstrapped data. Finally, distillation loss data can be generated.

A complete list of command line arguments for `preprocess.py` is provided in table C.2.

C.3 Compression

This module is used to do knowledge distillation, pruning, and quantization. Table C.3 contains a selection of arguments for `compress.py`. For compression, only SST-2, QQP, and MNLI are supported. In addition to these parameters, all configs defined in `compression/distillation/student_models/configs` can be overriden from the command line. F.x. to use byte-pair embeddings during distillation, one would write `--embedding-type bpe`.

Argument	Explanation
--model-name	Name given to a previously fine-tuned RoBERTa model.
--task	Either `sst-2`, `qqp`, or `mnli`.
--teacher-arch	`roberta_base` or `roberta_large`.
--glue-preprocess	Downloads and preprocesses a GLUE dataset for the specified task.
--augment	Runs data augmentation using given teacher and task.
--generate-loss	Generate distillation loss for given dataset, (one of `processed`, `tinybert`), task, and finetuned teacher
--cpu	Whether to run on CPU.
--loadbar	If set, shows an awesome loadbar.
--seed	Initializes the given random seed.

Table C.2: Program arguments for preprocessing tasks.

C.4 Evaluation

This module is used for evaluating the performance of various models on GLUE tasks. The supported model architectures are the base and large version of RoBERTa (`base`, `large`), and our student model architectures (`bilstm`, `rnn`, `emb-ffn`). The supported GLUE tasks are SST-2, QQP, and MNLI.

Table C.4 contains a list of command line arguments that can be passed to `evaluate.py`.

C.5 Analysis

This module is used to provide information about various types of models. This information includes size of a model, size of a zipped model, parameter distributions in a model, etc. This module supports the same model architectures and tasks are the same as in appendix C.4. In addition one can also get analysis information on the GLUE baseline models used in the thesis (`glue-glove`, `glue-elmo`).

A complete list of possible command line arguments for `analyze.py` is in table C.5.

Argument	Explanation
--compression-actions	Compression technique(s) to use (`distill`, `prune`, or `quantize`)
--task	GLUE task to use
--student-arch	Student architecture (`bilstm`, `rnn`, or `ffn`)
--load-trained-model	Name of a trained model to load for further compression
--cpu	Whether to run on CPU
--seed	Seed for weight initialization
--seed-name	One of (`bennington`, `hadfield`, `feynman`, or `simone`)
--loadbar	If set, shows an awesome loadbar.
Distillation specific args	
--epochs	Epochs to run for
--alpha	Alpha value (see 4.1.2)
--only-original-data	Whether to exclude bootstrapped data
Pruning specific args	
--prune-magnitude	Whether to use magnitude pruning
--prune-topv	Whether to use top v pruning
--prune-threshold	Threshold ratio for pruning
--prune-local	Whether to use local pruning as opposed to global
Quantization specific args	
--ptq-embedding	Whether to use post-training quantization on embedding layer
--dq-encoder	Whether to use dynamic quantization on rnn/lstm layers
--dq-classifier	Whether to use dynamic quantization on classifier layer
--ptq-classifier	Whether to use post-training quantization on classifier layer

Table C.3: Key program arguments for doing model compression.

Argument	Explanation
--model-name	Name given to a previously trained model.
--task	One of the supported tasks.
--arch	The architecture for the model.
--cpu	Whether to run on CPU.
--loadbar	If set, shows an awesome loadbar.

Table C.4: Program arguments for evaluating performance of models.

Argument	Explanation
--model-name	Name given to a previously trained model.
--task	One of the supported tasks.
--arch	The architecture for the model.
--model-size	Counts parameters and type, and prints size based on these.
--model-disk-size	Saves the model to disk, and prints the measured size.
--theoretical-size	Saves the model to disk, gzips the file, and prints measured size.
--weight-hist	Creates a histogram over weight values in a RoBERTa model.
--weight-thresholds	Prints amount of weights below certain thresholds for each layer in a model.
--named-params	Prints the names of all layers, the amount of weights and the dimension.
--pie-chart	Creates a pie chart for a model of its distribution of weights among major parts.
--save-pdf	Saves a histogram or pie-chart as pdf to disk.

Table C.5: Program arguments for analyzing models.

D Program Dependencies

For our program to function, Python 3.8 or higher should be installed. Faiseq should be installed using instructions at `https://github.com/pytorch/fairseq`. Additionally, pip and conda can be used for downloading further dependencies. These dependencies can be installed with following commands:

- `conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch`
- `conda install requests`
- `conda install matplotlib`
- `pip install sacred`
- `pip install gensim`
- `pip install bpemb`

References

Taiwo Samuel Ajani, Agbotiname Lucky Imoize, and Aderemi A. Atayero. 2021. An overview of machine learning within embedded and mobile devices-optimizations and applications. *Sensors*.

Ethem Alpaydin. 2014. *Introduction to Machine Learning*, 3 edition. The MIT Press.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big? 🦜. *Proceedings of FAccT*.

Emily M. Bender and Alexander Koller. 2020. Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198. Association for Computational Linguistics.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A Neural Probabilistic Language Model. In *Journal of Machine Learning Research*.

S. Börzsönyi, D. Kossmann, and K. Stocker. 2001. The skyline operator. In *Proceedings - International Conference on Data Engineering*.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Tvm: An automated end-to-end optimizing compiler for deep learning.

Michael Collins. 2013. Course notes: Language modeling. `http://www.cs.columbia.edu/~mcollins/lm-spring2013.pdf`. Accessed on 10.05.2021.

G. Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*.

William Dally. 2015. High-Performance Hardware for Machine Learning. In *Nips 2015*.

Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*.

Jeffrey L. Elman. 1990. Finding structure in time. *Cognitive Science*.

Igor Fedorov, Marko Stamenovic, Carl Jensen, Li Chia Yang, Ari Mandell, Yiming Gan, Matthew Mattina, and Paul N. Whatmough. 2020. TinyLSTMs: Efficient neural speech enhancement for hearing aids. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*.

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

Antaeus Feldspar. 1997. An explanation of the deflate algorithm. `https://zlib.net/feldspar.html`. Accessed on 17.08.2021.

Jonathan Frankle and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019*.

Shota Fukui, Jaehoon Yu, and Masanori Hashimoto. 2019. Distilling knowledge for non-neural networks. In *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, APSIPA ASC 2019*.

Robert M. Gray. 1984. Vector Quantization. *IEEE ASSP Magazine*.

Robert M. Gray and David L. Neuhoff. 1998. Quantization. *IEEE Transactions on Information Theory*.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *32nd International Conference on Machine Learning, ICML 2015*.

Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*.

Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Benjamin Heinzerling and Michael Strube. 2018. BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Dan Hendrycks and Kevin Gimpel. 2020. Gaussian error linear units (gelus).

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network.

Sepp Hochreiter. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowlege-Based Systems*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation*.

David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*.

IEEE. 2008. *(754) IEEE Standard for Floating-Point Arithmetic*.

Shankar Iyer, Nikhil Dandekar, and Kornél Csernai. 2017. First quora dataset release: Question pairs. `https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs`. Accessed on 26.08.2021.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, page 4163–4174. Association for Computational Linguistics.

Andrej Karpathy. 2015. The Unreasonable Effectiveness of Recurrent Neural Networks. `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Accessed on 18.05.2021.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*.

Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*.

Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper.

Yann LeCun. 1990. Optimal brain damage. *Advances in Neural Information Processing Systems*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach.

Matt Mahoney. Data Compression Explained.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space.

Alexandre Mutel. Crinkler secrets, 4k intro executable compressor at its best. https://xoofx.com/blog/2010/12/28/crinkler-secrets-4k-intro-executable. Accessed on 10.05.2021.

Pandu Nayak. 2019. Understanding searches better than ever before. https://blog.google/products/search/search-language-understanding-bert/. Accessed on 13.05.2021.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. Fairseq: A fast, extensible toolkit for sequence modeling. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Demonstrations Session*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

PyTorch. 2019a. Pytorch basepruningmethod. https://pytorch.org/docs/1.8.1/generated/torch.nn.utils.prune.BasePruningMethod.html. Accessed on 17.08.2021.

PyTorch. 2019b. Pytorch crossentropy loss. https://pytorch.org/docs/1.8.1/generated/torch.nn.CrossEntropyLoss.html. Accessed on 02.08.2021.

PyTorch. 2019c. Pytorch deterministic algorithms. https://pytorch.org/docs/1.8.1/generated/torch.use_deterministic_algorithms.html. Accessed on 31.08.2021.

PyTorch. 2019d. Pytorch mean square error loss. https://pytorch.org/docs/1.8.1/generated/torch.nn.MSELoss.html. Accessed on 02.08.2021.

PyTorch. 2019e. Pytorch pruning tutorial. https://pytorch.org/tutorials/intermediate/pruning_tutorial.html. Accessed on 10.08.2021.

PyTorch. 2019f. Pytorch quantization tutorial. https://pytorch.org/docs/1.8.1/quantization.html. Accessed on 15.08.2021.

PyTorch. 2019g. Pytorch relu. https://pytorch.org/docs/1.8.1/generated/torch.nn.ReLU.html. Accessed on 25.08.2021.

PyTorch. 2019h. Pytorch reproducibility. https://pytorch.org/docs/1.8.1/notes/randomness.html. Accessed on 31.08.2021.

PyTorch. 2019i. Pytorch sigmoid. https://pytorch.org/docs/1.8.1/generated/torch.nn.Sigmoid.html. Accessed on 25.08.2021.

PyTorch. 2019j. Pytorch softmax. https://pytorch.org/docs/1.8.1/generated/torch.nn.Softmax.html. Accessed on 25.08.2021.

PyTorch. 2019k. Pytorch sparse tensors. https://pytorch.org/docs/1.8.1/sparse.html. Accessed on 11.08.2021.

PyTorch. 2019l. Pytorch tanh. https://pytorch.org/docs/1.8.1/generated/torch.nn.Tanh.html. Accessed on 25.08.2021.

PyTorch. 2020. Pytorch serve. https://pytorch.org/serve. Accessed on 22.08.2021.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A Primer in BERTology: What we know about how BERT works. In *Transactions of the Association for Computational Linguistics, Volume 8*, page 842–866.

F. Rosenblatt. 1957. The Perceptron - A Perceiving and Recognizing Automaton. Technical report.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986a. Learning representations by back-propagating errors. *Nature*.

D.E Rumelhart, G.E Hinton, and R.J Williams. 1986b. Learning Internal Representations By Error Propagation.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter.

Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. *CoRR*, abs/2005.0.

John R. Searle. 1998. *Mind, language and society : philosophy in the real world / John R. Searle*, 1st ed. edition. Basic Books New York.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP 2013 - 2013 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for deep learning in NLP. In *ACL 2019 - 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170. Association for Computational Linguistics.

Dan Svenstrup, Jonas Meinertz Hansen, and Ole Winther. 2017. Hash embeddings for efficient word representations. In *Advances in Neural Information Processing Systems*.

Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from BERT into simple neural networks.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. 2020. Investigating gender bias in language models using causal mediation analysis. In *Advances in Neural Information Processing Systems*, volume 33, pages 12388–12401. Curran Associates, Inc.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *7th International Conference on Learning Representations, ICLR 2019*.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia Yu Chen, and Kailash Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*.

Moshe Wasserblat, Oren Pereg, and Peter Izsak. 2020. Exploring the Boundaries of Low-Resource BERT Distillation. In *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, page 35–40. Association for Computational Linguistics.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*.

P. C. Woodland. 1989. Weight limiting, weight quantisation & generalisation in multi-layer perceptrons. In *IEE Conference Publication*.

Matthew D. Zeiler. 2012. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. TernaryBERT: Distillation-aware ultra-low bit BERT.

Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*.

Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural network distiller: A python package for DNN compression research.