MSc Thesis
Master of Science

DTU Compute
Department of Applied Mathematics and Computer Science

Semi-Supervised Neural Machine Translation

for small bilingual datasets

Andreas Madsen (s123598)

Kongens Lyngby 2017

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

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

Summary (English)

This thesis investigates models for translating between natural languages. The models are based on a recently published model called ByteNet. The ByteNet model is a new approach to neural machine translation that isn't based on attention but instead layers of hierarchical dilated convolution. Using this approach the ByteNet model runs in linear time while still having a resolution that is proportional to the sentence length.

Using a variation of the ByteNet model, the model is applied in a semi-supervised setting, where it is trained on both a bilingual and a monolingual dataset. The idea is that the best performing models are typically those that use the most data. For language pairs where the bilingual dataset is small, monolingual data could be a vital supplement.

Results showed that ByteNet is able to learn natural language translation from German to English using a bilingual dataset. The model was unfortunately too time-consuming to use in a semi-supervised setting for natural language translation. Variations on the ByteNet model was tried in order to reduce the training time, but none of these models were sufficiently successful.

Because of the training time issue, a reduced ByteNet model was used to learn a synthetic problem that mimics natural language translation. The semi-supervised experiments showed that monolingual data improves the predictive performance.

Summary (Danish)

Denne afhandling undersøger modeller til at oversætte imellem naturlige sprog. Modellerne er baseret på en fornyligt publiceret model kaldet ByteNet. ByteNet modellen er en ny tilgang til neural maskine oversættelse, som ikke er baseret på *attention*, men istedet bruger lag af *hierarchical dilated convolutions*. Ved brug af denne metode kan ByteNet kører i lineær tid og samtidig have en opløsning, der er proportionel med sætningslængden.

Ved at bruge en variation af ByteNet modellen er modellen benyttet i en semi-vejledt kontekst, hvor den er trænet på både et tosproget og ensproget datasæt. Iden er at de bedste ydende modeller er dem, som bruger mest data, for sprogpar hvor det tosproget dataset er lille, kan et ensproget datasæt gøre en stor forskel.

Resultaterne viser, at ByteNet er i stand til at lære sprog oversættelse fra Tysk til Engelsk ved at bruge et tosproget dataset. Modellen var desværre for tidsmæssigt langsom til at kunne blive brugt i en semi-vejledt kontekst til sprog oversættelse. Variationer af ByteNet modellen var afprøvet for at reducere træningstiden, men ingen af disse modeller var succesfulde.

På grund af træningstidsproblemet blev en reduceret ByteNet model brugt til at lære et syntetisk problem, der mindre om naturlige sprog oversættelse. De semi-vejledte eksperimenter viste at ensproget data forbedre modellens ydeevne.

Preface

This thesis was prepared at Department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark in fulfillment of the requirements for acquiring an MSc degree in Mathematics. The work was carried out in the period from January 2017 to June 2017.

Kongens Lyngby – June 30, 2017

Missing Signature

Andreas Madsen (s123598)

Acknowledgements

First and foremost I would like to thank my supervisor Ole Winther for his guidance and support. I would also like to thank Namju Kim, who wrote a quadratic-time implementation of the ByteNet model, which I took much inspiration from. Finally, I would like to thank the DTU HPC team for their fast support.

Contents

CHAPTER 1

Introduction

In the European Parliament, there are 23 officially spoken languages and precise translation is essential for clear communication. Each elected member is not expected to understand all 23 languages, and there isn't a common language that everyone speaks equally well, thus the European Parliament employs translators for translating between these languages [1].

For the European Parliament it is possible to employ translation specialists, and in cases where there nobody to translate directly English, French and German are used as relay languages [1]. However, translation specialists are not often available to the public.

In India, more than 400 million people have internet access, and most of India's online content is written in English. However, only 20% of the Indian population speaks English. Excluding English, the nine most used languages in India are: Hindi, Bengali, Marathi, Gujarati, Punjabi, Tamil, Telugu, Malayalam and Kannada. For translating between these languages and English Google have recently started to use Neural Machine Translation in their Google Translate service [2].

Google Translate translates more than 100 billion words each day, for its 500 million users [3]. In the European Parliament, and the United Nations, automatic translation software is also used for assisting the specialists [1]. These tools are using for the majority of languages, a technology called "Statistical machine translation".

Statistical machine translation (SMT) combines a probability model for the target language, as well as a probability model for mapping between the source and target language. Such a probability model can be a Hidden Markov Model and they will be fitted using a bilingual dataset [4]. Bilingual means the dataset contains both the source and target language for each sentence. Previously these models have been word-based, allowing very limited context understanding. Recently phrase-based machine translation (PBMT) have been introduced, this allows for much better translation of idioms or multi-word expressions than what has previously been possible. Phrase-based translation is currently the primary strategy used in machine translation. However, even phrase-based translation has a limited understanding of context and can't consider an entire sentence.

Recently advances have been made in applying neural networks to machine translation, this strategy is called neural machine translation (NMT) and has been shown to outperform the current PBMT approaches [5]. This approach is able to consider

an entire sentence or more and is thus able to understand the context of each word on a level that has not been previously possible.

Beyond being able to process entire sentences, neural machine translation is a more flexible approach than PBMT. As such the NMT strategy is highly relevant for language pairs that have previously been notoriously difficult, such as Chinese to English translation. In September 2016, Google announced that they now use neural machine translation for Chinese to English translation, they are calling this architecture GNMT.

Over the last year, GNMT has been enabled for English, French, German, Spanish, Portuguese, Chinese, Japanese, Korean, Turkish, Russian, Hindi, Vietnamese, Hebrew, and Arabic along with the Indian dialects. These languages are chosen either because there is a huge dataset or because they are notoriously difficult to translate using the existing PBMT strategy. In NMT the quality of the translation is often more dependent on how much data is used, than how advanced the neural architecture is.

In this thesis, NMT will be used to provide German to English translation, and additional effort will be given to applying NMT to "small" bilingual dataset. The strategy for applying NMT to "small" datasets, is based on the fact that humans, especially babies, are capable of learning languages without prior knowledge. It should thus be possible to use additional non-translated data (monolingual) to train the NMT model. A now almost classical example of using monolingual data to build a language model is the Word2Vec model, this uses the Wikipedia corpus for a single language to create word embeddings. These word embeddings have shown meaningful properties like synonyms being close to each other and relations like $king - man + woman \approx queen$ [6].

The approach used in this thesis is based on the intuition that translating from for example German to English and then back to German again, should result in the same sentence. This approach does not require the correct English sentence to be known for all sentences, thus a monolingual German dataset can be used. The approach has been shown to yield some improvements [7]. This must of course be used in combination with a bilingual dataset, otherwise the neural network will just learn the identity function. This combination of monolingual and bilingual data is called semi-supervised learning.

There are other approaches for semi-supervised learning, in particular, generative adversarial networks (GAN) has shown good results in computer vision. However, for natural language processing such as NMT, it still lags far behind likelihood based methods [8].

On a theoretical level, the semi-supervised approach does not depend on the neural translation model. A popular translation model is the Bahdanau Attention model that has shown state-of-the-art performance in machine translation [9]. However, the Bahdanau Attention model is a word-based model and for some languages, like

German that likes to combine words, it can be advantageous to use characters instead of words as input to the model. While word-based models are currently superior, using characters instead of words eliminates the out-of-dictionary problem that often exists in word-based models.

The out-of-dictionary problem happens because a neural network can only supports a fixed number of outputs, one must thus decide on a fixed dictionary before training the model. Typically 80000 words are used as the dictionary size, if the dictionary becomes much larger than this, it will not be computationally feasible to train the model. 80000 words are not enough to contain every street name and rarely used words, an obvious solution is thus to use characters instead of words. A typical language will not use more than 300 different characters.

While it is possible to use the Bahdanau Attention model with characters as input, the sequences become much longer and this creates certain computational problems. Recently a different approach called ByteNet has been created, this model is specifically character-based and solves the computational problems [10]. The ByteNet model promises high computational performance and achieves state-of-the-art performance compared to other character-based models. Using a computational pragmatic model like ByteNet is essential when one does not have Google-level resources.

The ByteNet model and the semi-supervised approach have never been combined before. In particular, the original paper implementing the semi-supervised approach used the word-based Bahdanau Attention model, which is drastically different than the ByteNet model.

CHAPTER 2

Theory

2.1 Feed forward Neural Network

Neural machine translation is based on the neural network framework. Modern networks are extremely complex and use rather advanced operations such as convolution, others use complex combinations of simpler operations such as the LSTM block [11]. To give a short introduction to this framework the feed forward neural network (FFNN) is presented. This is the simplest type of network in the neural network family. As such it doesn't have much use for translation, but the more advanced neural networks can be seen as extensions of a feed forward neural network.

2.1.1 The neuron

The neuron is the main component of any neural network. Mathematically it is just a function which takes a vector and returns a scalar. It does this by a weighted sum of the inputs $\{x_i\}_{i=1}^{I}$ and by adding a bias value b. The sum is then typically transformed using a non-linear function θ.

$$a = \theta(z), \quad z = \sum_{i=1}^{I} w_i x_i + b \tag{2.1.1}$$

The value a is the output of the neuron and is called the *activation*. In the past, the sigmoid function $\sigma(\cdot)$ and the hyperbolic tangent $\tanh(\cdot)$ function have been very popular [12], but recently the rectified linear unit (ReLU) $\theta(z) = \max(0, z)$ has gained popularity [11]. If no non-linear function is applied then the identity function $\theta(a) = a$ is used.

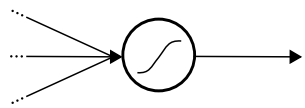


Figure 2.1.1: Visual representation of a single neuron. The left arrows represent the input elements $\{x_i\}_{i=1}^{I}$. The circle represents the function that returns the activation scalar a (right arrow).

2.1.2 The network

A feed forward neural network is in its simplest form a *multivariate non-linear regression model* constructed by combining neurons. Such a regression model can then be turned into a classification model by adding a softmax transformation [13] to the output. By doing this each output value becomes the class probability $y_k = P(C_k|x)$.

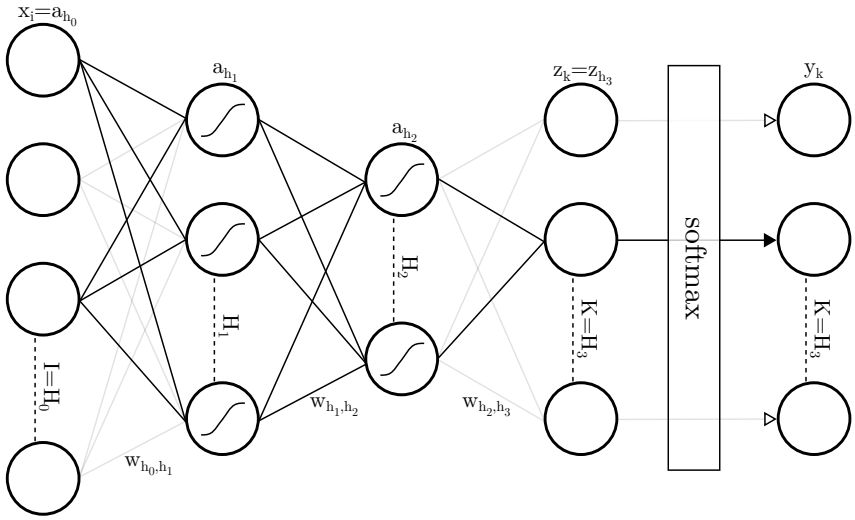


Figure 2.1.2: Visual representation of a neural network with two hidden layers. Some lines are less visible, this is just a visual aid because the many connections can otherwise be difficult to look at.

The neural network in Figure 2.1.2, have one input layer $x_i, i \in [1, I]$ and one output layer $y_k, k \in [1, K]$. This is the same for any neural network, what varies is the type of network (here a feed forward neural network) and the number of hidden layers (here two). The hidden layers are typically the layers that contain the non-linear transformation functions. If there are no hidden layers, the neural network is just a multi-class logistic regressor [12].

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

While diagrams like in figure 2.1.2 are useful for feed forward neural networks, they becomes too verbose for more complex networks. Instead the visual representation is simplified such each layer is represented by a single block like in figure 2.1.3.

In figure 2.1.3 the dense block represents a dense matrix multiplication, which is essentially just a neural network layer without the activation. This is then followed

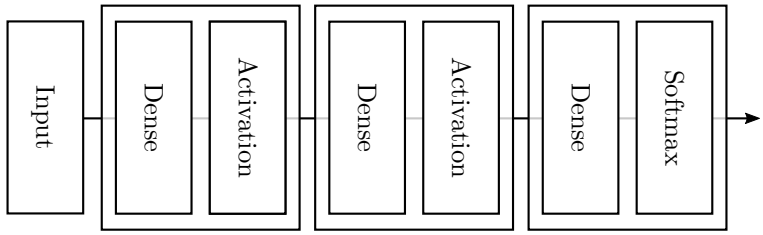


Figure 2.1.3: Network block-diagram of the same network as shown in figure 2.1.2.

by an activation block, which in figure 2.1.3 is left unspecified for generality.

2.1.3 Forward pass

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

The *forward pass* is simply calculated by applying the neuron function to all neurons in all layers. By defining $a_{h_0} = x_i$ the *forward pass* can be generalized to any amount of hidden layers (L):

$$\begin{aligned} a_{h_\ell} &= \theta(z_{h_\ell}), && \forall h_\ell \in [1, H_\ell], \ell \in [1, L] \\ z_{h_\ell} &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}} + b_{h_\ell}, && \forall \ell \in [1, L+1] \text{ where: } a_{h_0} = x_i, H_0 = I \end{aligned} \tag{2.1.2}$$

Note that the last layer $\ell = L+1$ does not use the non-linear transformation function θ, instead it uses the softmax function.

$$y_k = \frac{\exp(z_k)}{\sum_{k'=1}^{K} \exp(z_{k'})}, \quad \forall k \in [1, K] \text{ where: } z_k = z_{h_{L+1}}, K = H_{L+1} \tag{2.1.3}$$

The z_k values that are fed to the softmax function are often called *logits*, because they are unnormalized log probabilities ($\exp(z_k) \propto y_k$), this has some advantages. For example, if one wants the most likely label given $\mathbf{x}$ one can just find the largest z_k and thus skip the softmax, which can otherwise be quite expensive to calculate.

2.1.4 Loss function

Optimization of the parameters $w_{h_{\ell-1},h_\ell}$ and b_{h_ℓ} requires a definition of a loss function. For classification it makes sense to maximize the joint probability of observing

all the observations:

$$P(\mathbf{t}|\mathbf{x},\mathbf{w},\mathbf{b}) = \prod_{n=1}^{N} P(\mathbf{t}_n|\mathbf{x}_n,\mathbf{w},\mathbf{b}) = \prod_{n=1}^{N}\prod_{k=1}^{K} P(C_{n,k}|\mathbf{x}_n,\mathbf{w},\mathbf{b})^{t_{n,k}} \tag{2.1.4}$$

Here $\mathbf{x}_n$ is the input vector for observation n, with a corresponding label vector $\mathbf{t}_n$. The label vector is an indicator vector constructed using 1-of-K encoding. The class probability $P(C_{n,k}|\mathbf{x}_n,\mathbf{w},\mathbf{b})$ is y_k from the *forward pass* for observation n.

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

$$-\ln\left(P(\mathbf{t}|\mathbf{x},\mathbf{w},\mathbf{b})\right) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\ln\left(P(C_{n,k}|\mathbf{x}_n,\mathbf{w},\mathbf{b})\right) \tag{2.1.5}$$

Because of the linearity in (2.1.5), it makes sense to just consider the loss function for one data point, thus the n index can be omitted. This gives the final loss function which is denoted by $\mathcal{L}$:

$$\mathcal{L} = -\sum_{k=1}^{K} t_k \ln\left(P(C_k|\mathbf{x},\mathbf{w})\right) = -\sum_{k=1}^{K} t_k \ln(y_k) \tag{2.1.6}$$

The formulation in (2.1.6) is called the *cross-entropy loss function*, it is particularly useful as it allows for some numerical stability tricks, see Appendix C.1.

In cases where t_k is an indicator vector, (2.1.6) can be computationally simplified to only calculate the term where $t_k = 1$.

2.1.5 Backward pass

For neural networks, there is no closed form solution to optimizing the loss function. Instead, gradient-based optimization algorithms are used. Gradient-based optimization uses the derivatives of the loss function with respect to the parameters to iteratively optimize the parameters. This approach will be discussed in Section 2.2, for now the important part is to calculate the derivatives:

$$\frac{\partial\mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} \wedge \frac{\partial\mathcal{L}}{\partial b_{h_\ell}}, \quad \forall\ell \in [1, L+1] \tag{2.1.7}$$

For calculating the derivatives the *error backpropagation* algorithm is used, this algorithm results in what is called the *backward pass*.

The main trick in *error backpropagation* is to define the partial derivative δ_{h_ℓ}, this is used for bookkeeping and is what makes it feasible to calculate the derivatives.

$$\delta_{h_\ell} \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} \tag{2.1.8}$$

Using this definition the chain rule can be applied on (2.1.7):

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} &= \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} \frac{\partial z_{h_\ell}}{\partial w_{h_{\ell-1},h_\ell}} &&= \delta_{h_\ell} a_{h_{\ell-1}}, \quad \forall \ell \in [1, L+1], \quad a_{h_0} = x_i \\ \frac{\partial \mathcal{L}}{\partial b_{h_\ell}} &= \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} \frac{\partial z_{h_\ell}}{\partial b_{h_\ell}} &&= \delta_{h_\ell}, \qquad \forall \ell \in [1, L+1] \end{aligned} \tag{2.1.9}$$

Calculating δ_{h_ℓ} is a bit more involved since z_{h_ℓ} affects more than one intermediate variable, but even so the chain rule can still be applied:

$$\begin{aligned} \delta_{h_\ell} = \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} &= \frac{\partial \mathcal{L}}{\partial a_{h_\ell}} \frac{\partial a_{h_\ell}}{\partial z_{h_\ell}} \\ &= \theta'(z_\ell) \sum_{h_{\ell+1}}^{H_{\ell+1}} \frac{\partial \mathcal{L}}{\partial z_{\ell+1}} \frac{\partial z_{\ell+1}}{\partial a_\ell} \\ &= \theta'(z_\ell) \sum_{h_{\ell+1}}^{H_{\ell+1}} \delta_{h_{\ell+1}} w_{h_\ell, h_{\ell+1}}, \forall \ell \in [1, L] \end{aligned} \tag{2.1.10}$$

The last delta ($\delta_{h_{L+1}}$) is different but can still be calculated by using the chain rule (see appendix B.1 for how $\delta_{h_{L+1}}$ is derived).

$$\delta_{h_{L+1}} = \delta_k = \frac{\partial \mathcal{L}}{\partial z_k} = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial z_k} = y_k - t_k \tag{2.1.11}$$

Using (2.1.11) and (2.1.10), all δ_{h_ℓ} for $\ell \in [1, L+1]$ can be calculated for a feed forward neural network with L hidden layers. Note in particular how (2.1.11) is an error measure and how this value is propagated back through the network by the δ_{h_ℓ} equations in (2.1.10). This is why the method is called *error backpropagation*.

2.2 Gradient-Based Optimization

In machine learning one often wish to minimize a loss function that is not easily optimizable. Such a loss function can be misclassification rate or the BLEU score which is commonly used in translation. Because these loss functions are hard to optimize directly a similar loss function like cross-entropy is often used. Loss functions like cross-entropy are easily differentiable which makes it feasible to optimize them.

Modern neural networks typically have a large number of weights, the ByteNet model which will later be used for natural language translation has approximately 27 million parameters. Classical optimization methods typically use the second-order derivative of the loss function such as the Newton method or use an approximation like the Quasi-Newton method. However, because the second-order derivative matrix (the Hessian), or even just approximations of it, scales quadratically with the number of weights, it is not feasible to use the second-order derivative for optimizing neural networks.

Because of the many weights in neural networks, optimization is typically done using only the gradient, this is known as *gradient-based optimization*.

The fundamental principal in *gradient-based optimization* is to initialize the parameters, typically this will be done randomly. The parameters are then optimized by iteratively going in the opposite direction of the gradient of the loss function with respect to the parameters.

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \boldsymbol{\Delta}_t, \quad \boldsymbol{\Delta}_t = \alpha \frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}} \tag{2.2.1}$$

Here $\mathbf{w}_t$ is a vector containing all the model weights at iteration t, $\boldsymbol{\Delta}_t$ is the step taken, and α is called the *step size* or *learning rate*. The learning rate α should be small enough such that the step doesn't jump over the minima, but also big enough such that the model gets optimized within reasonable time.

It is important to understand that even if the global minima is found, $\mathcal{L}(\mathbf{w}_{t-1})$ is not the true loss function, it is just the loss function given a training dataset. Hopefully, the training dataset is an accurate representation of reality, but this is not possible to guarantee. To validate the model, a test dataset that is not a part of the training dataset is used.

It is often the case that the minima when using the training dataset is not the minima when using the test dataset. The typical behavior, is that the training loss will continue to decrease because that is what is being optimized, while the test loss will only decrease initially and then increase later. This behavior is called overfitting, there are numerous ways to prevent this, some of which will be discussed and used throughout this thesis.

Figure 2.2.1: Shows the ByteNet model overfitting. The semi-transparent line is the loss, and the non-transparent line is an exponential moving average of the loss.

2.2.1 Stochastic Optimization

While the approach in (2.2.1) can work, it is often not feasible to calculate $\frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$ on the entire dataset. For a model with a huge number of parameters, it is typically necessary to also have a very large dataset. To solve this problem $\frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$ is estimated using a random sample of observations in each iteration. This approach is called *minibatch stochastic gradient descent* or just *minibatch learning* [11].

Beyond making the computation feasible this also has some advantages:

- Consider two datasets one with 256 observations and another with 16 observations. It will take 16 times more time to calculate the gradient using 256 observations than 16 observations. However, because of the square-root law, it only decreases the standard error by a factor of 4. It is thus likely to be more economical to perform 16 as many iterations, than being 4 times more accurate.
- It is typically easy to parallelize over observations. For a multi-parallel hardware like a GPU it is typically possible to parallelize over many observations without extra cost. For large networks, it is common to estimate the gradient with approximately 16 observations.
- Sampling observations from a dataset adds noise to the gradients. This noise can have a regularizing effect on the training, causing less overfitting [11].

The stochastic part of *minibatch stochastic gradient descent* comes from the fact that $\frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$ is no longer the same in each iteration. Because the gradient is estimated

from a new random sample of observations from the dataset, $\frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$ is essentially samples from a random distribution. This changes the minimization problem from minimizing $\mathcal{L}$ to minimizing the expectation $\mathbb{E}[\mathcal{L}]$. Minibatch learning does not directly tackle this change of problem type, it just causes it. To correctly optimize a stochastic loss function more advanced optimization methods are needed.

2.2.2 Adam Optimization

Adam optimization is a gradient-based optimization method for optimizing a stochastic loss function, it has been shown to work well both theoretically and empirically. It even works on a stochastic loss function with changing distribution. Changing distribution is something that is quite common when changing the parameters of the model, thus this is an important property [14].

The central idea of Adam optimization is to scale the step-size by a signal-to-noise estimate. Such that when the signal is strong a large step is taken and when it is weak a small step is taken. This is accomplished by having a running estimate of the first and second moment.

$$\begin{aligned} \mathbf{g}_t &= \frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}} \\ \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \end{aligned} \tag{2.2.2}$$

The "signal-to-noise" ratio as the original paper [14] calls it, is then calculated as $\frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t}}$. Calling $\frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t}}$ a "signal-to-noise" ratio is a slight abuse of terminology, because $\mathbf{v}_t$ is not exactly a variance estimate, but the effect is similar.

A problem that remains is how to initialize $\mathbf{m}_0$ and $\mathbf{v}_0$. The solution is to initialize them as zero. Since zero is what the gradient will converge to, this is not unreasonable, however to improve the estimates they are rescaled to be unbiased.

Consider the second-order moment at iteration t, this can be directly expressed as:

$$\mathbf{v}_t = (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i}\mathbf{g}_i^2 \tag{2.2.3}$$

To make the $\mathbf{v}_t$ estimate unbiased the relation between $\mathbb{E}[\mathbf{v}_t]$ and $\mathbb{E}[\mathbf{g}_t^2]$ should be calculated. Assuming the distribution of $\mathbb{E}[\mathbf{g}_t^2]$ is stationary the relation is:

$$\begin{aligned} \mathbb{E}[\mathbf{v}_t] &= \mathbb{E}\left[(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbf{g}_i^2\right] = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbb{E}\left[\mathbf{g}_i^2\right] \\ &= \mathbb{E}\left[\mathbf{g}_i^2\right](1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i} = \mathbb{E}\left[\mathbf{g}_i^2\right](1-\beta_2^t) \end{aligned} \tag{2.2.4}$$

Thus to unbias the $\mathbf{v}_t$ estimate it is scaled by $\frac{1}{1-\beta_2^t}$, similarly $\mathbf{m}_t$ re scaled by $\frac{1}{1-\beta_1^t}$:

$$\hat{\mathbf{m}}_t = \frac{1}{1-\beta_1^t}\mathbf{m}_t, \quad \hat{\mathbf{v}}_t = \frac{1}{1-\beta_2^t}\mathbf{v}_t \tag{2.2.5}$$

The parameter update is then done as follow:

$$\begin{aligned} \boldsymbol{\Delta}_t &= \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\mathbf{g}_t \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \boldsymbol{\Delta}_t \end{aligned} \tag{2.2.6}$$

Computationally this can be done slightly more efficient by combining (2.2.5) with (2.2.6):

$$\boldsymbol{\Delta}_t = \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t} + \epsilon}\mathbf{g}_t \tag{2.2.7}$$

Algorithm 1 Adam Optimization, default parameters are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

function ADAM($\mathcal{L}(\mathbf{w}), \mathbf{w}_0, \alpha, \beta_1, \beta_2, \epsilon$)
  $\mathbf{m}_0 \leftarrow$ ZEROLIKE($\mathbf{w}$)
  $\mathbf{v}_0 \leftarrow$ ZEROLIKE($\mathbf{w}$)
  $t \leftarrow 0$
  repeat
    $t \leftarrow t + 1$
    $\mathbf{g}_t \leftarrow \frac{\partial \mathcal{L}(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$
    $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$
    $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$
    $\boldsymbol{\Delta}_t \leftarrow \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t}+\epsilon}\mathbf{g}_t$
    $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \boldsymbol{\Delta}_t$
  until converged
  return $\mathbf{w}_t$

2.2.3 He-Uniform Initialization

An open question that has not been answered is how to initialize the weights $\mathbf{w}_0$. This turns out to very much depend on the architecture, the ByteNet model that will be used later is a convolutional model with ReLU activation. For this kind of architecture, *He initialization* has been shown to perform well [15].

Good initialization is required to prevent either exploding or vanishing gradients. Consider both the forward and backward pass for a feed forward neural network.

$$\begin{aligned} z_{h_\ell} &= \sum_{h_{\ell-1}=1}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}} + b_{h_\ell}, a_{h_\ell} = \theta(z_{h_\ell}) \\ \delta_{h_\ell} &= \theta'(z_\ell) \sum_{h_{\ell+1}}^{H_{\ell+1}} \delta_{h_{\ell+1}} w_{h_\ell,h_{\ell+1}} \end{aligned} \tag{2.2.8}$$

In both cases, the output of layer ℓ (z_{h_ℓ} and δ_{h_ℓ}) depends on the output of the previous or next layer ($z_{h_{\ell-1}}$ and $\delta_{h_{\ell-1}}$). If both z_{h_ℓ} and δ_{h_ℓ} does not maintain the same scale throughout all the layers, the values will diverge either to very small (vanishing) or very big values (exploding).

Since the values of input are unknown and the weights are to be initialized randomly, one needs to analyze z_{h_ℓ} and δ_{h_ℓ} as stochastic variables. Keeping the expectation at a constant scale for all layers is not a huge problem, the biases b_{h_ℓ} are initialized to zero (not random), and the weights $w_{h_{\ell-1},h_\ell}$ should have zero mean and be independently sampled. Likewise, it is for the analysis assumed that the inputs a_{h_0} also are independent, but they do not necessarily have zero mean.

$$\begin{aligned} \mathbb{E}[z_{h_\ell}] &= H_{\ell-1}\mathbb{E}[w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}} + b_{h_\ell}] = H_{\ell-1}\left(\mathbb{E}[w_{h_{\ell-1},h_\ell}]\mathbb{E}[a_{h_{\ell-1}}] + \mathbb{E}[b_{h_\ell}]\right) \\ &= H_{\ell-1}\left(\cdot 0 \cdot \mathbb{E}[a_{h_{\ell-1}}] + 0\right) = 0 \\ \mathbb{E}[\delta_{h_\ell}] &= \mathbb{E}[\theta'(z_\ell)] H_{\ell+1}\mathbb{E}[\delta_{h_{\ell+1}}]\mathbb{E}[w_{h_\ell,h_{\ell+1}}] \\ &= \mathbb{E}[\theta'(z_\ell)] H_{\ell+1}\mathbb{E}[\delta_{h_{\ell+1}}] \cdot 0 = 0 \end{aligned} \tag{2.2.9}$$

It turns out that to get a good initialization it is also necessary to keep a constant variance for all layers [15, 16]. To do this correctly it is necessary to know the activation function. An often used approach is the Glorot initialization [16]. However, this assumes the activation function is linear around $z_{h_\ell} = 0$ which is not the case for ReLU ($max(0, z_{h_\ell})$). He initialization has later been derived specifically for the ReLU activation function [15].

The variance for the forward pass is:

$$\mathrm{Var}[z_{h_\ell}] = H_{\ell-1}\mathrm{Var}[w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}}] \tag{2.2.10}$$

Because $w_{h_{\ell-1},h_\ell}$ has zero mean this can be rewritten as:

$$\begin{aligned} \mathrm{Var}[z_{h_\ell}] &= H_{\ell-1}\left(\mathbb{E}[w^2_{h_{\ell-1},h_\ell} a^2_{h_{\ell-1}}] - \mathbb{E}[w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}}]^2\right) \\ &= H_{\ell-1}\mathbb{E}[w^2_{h_{\ell-1},h_\ell}]\mathbb{E}[a^2_{h_{\ell-1}}] \\ &= H_{\ell-1}\left(\mathrm{Var}[w_{h_{\ell-1},h_\ell}] + \mathbb{E}[w_{h_{\ell-1},h_\ell}]^2\right)\mathbb{E}[a^2_{h_{\ell-1}}] \\ &= H_{\ell-1}\mathrm{Var}[w_{h_{\ell-1},h_\ell}]\mathbb{E}[a^2_{h_{\ell-1}}] \end{aligned} \tag{2.2.11}$$

If additionally the $w_{h_{\ell-1},h_\ell}$ distribution is chosen to be symmetric around zero, then z_{h_ℓ} will also be symmetric around zero, call this distribution $p(z_{h_\ell}) = p(-z_{h_\ell})$, then it can be derived that $\mathbb{E}[a_{h_\ell}^2] = \frac{1}{2}\text{Var}[z_{h_\ell}]$.

$$\begin{aligned}
\mathbb{E}[a_{h_\ell}^2] &= \int_{-\infty}^{\infty} \max(0, z_{h_\ell})^2 p(z_{h_\ell}) \mathrm{d}z_{h_\ell} &&= \int_{-\infty}^{0} 0^2 p(z_{h_\ell}) \mathrm{d}z_{h_\ell} + \int_{0}^{\infty} z_{h_\ell}^2 p(z_{h_\ell}) \mathrm{d}z_{h_\ell} \\
&= \int_{0}^{\infty} z_{h_\ell}^2 p(z_{h_\ell}) \mathrm{d}z_{h_\ell} &&= \frac{1}{2} \int_{-\infty}^{\infty} z_{h_\ell}^2 p(z_{h_\ell}) \mathrm{d}z_{h_\ell} \\
&= \frac{1}{2}\mathbb{E}[z_{h_\ell}^2] = \frac{1}{2}\left(\mathbb{E}[z_{h_\ell}^2] - \mathbb{E}[z_{h_\ell}]^2\right) &&= \frac{1}{2}\text{Var}[z_{h_\ell}]
\end{aligned}$$

Applying $\mathbb{E}[a_{h_{\ell-1}}^2] = \frac{1}{2}\text{Var}[z_{h_{\ell-1}}]$ to $\text{Var}[z_{h_\ell}]$ yields:

$$\begin{aligned}
\text{Var}[z_{h_\ell}] &= H_{\ell-1}\text{Var}[w_{h_{\ell-1},h_\ell}]\frac{1}{2}\text{Var}[z_{h_{\ell-1}}] \\
&= \text{Var}[z_{h_1}] \prod_{\ell=2}^{L} H_{\ell-1}\frac{1}{2}\text{Var}[w_{h_{\ell-1},h_\ell}]
\end{aligned} \tag{2.2.12}$$

From (2.2.12) it seen that to satisfy a constant variance ($\text{Var}[z_{h_\ell}] = \text{Var}[z_{h_1}]$), it is sufficient to have $H_{\ell-1}\frac{1}{2}\text{Var}[w_{h_{\ell-1},h_\ell}] = 1$.

$$\text{Var}[w_{h_{\ell-1},h_\ell}] = \frac{2}{H_{\ell-1}} \tag{2.2.13}$$

The same sufficient condition is found by analyzing the backward pass [15].

Thus long as the sampling distribution has zero mean, is symmetric, is sampled independently, and has variance $\frac{2}{H_{\ell-1}}$, there shouldn't be any convergence issues. Two normal choices are the normal distribution and the uniform distribution. An issue with using the normal distribution is that there is a small risk that extreme values will be sampled, which can cause issues. A truncated normal distribution could be used to solve this issue, but a much simpler solution is to just use a uniform distribution.

A uniform distribution from $-r$ to r has variance:

$$\text{Var}[w_{h_{\ell-1},h_\ell}] = \frac{1}{12}(r - (-r))^2 = \frac{1}{3}r^2 \tag{2.2.14}$$

Thus the uniform distribution parameter should be:

$$r = \sqrt{\frac{6}{H_{\ell-1}}} \tag{2.2.15}$$

Note, $H_{\ell-1}$ is only valid in some cases. In other cases, the number of weights used in the activation sum may be greater or less than $H_{\ell-1}$. For generality, $\text{fan}_{\ell,in}$ is used instead of $H_{\ell-1}$.

2.3 Convolution Neural Network

Convolution is typically used in classical signal and image processing but has recently become popular in neural networks. A classical example is image blurring, where one is given an image $a(x, y)$ which is to be blurred. This blurring is done by averaging neighboring pixels, this average is typically weighted using a 2D Gaussian distribution.

This can be generalized to any input $a(x, y)$ and any weight function $w(x, y)$ by:

$$z(x, y) = (a * w)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} a(i, j)w(x - i, y - j)\partial i \partial j \tag{2.3.1}$$

Similarly for a discrete image and discrete weight function:

$$z(x, y) = (a * w)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} a(i, j)w(x - i, y - j) \tag{2.3.2}$$

Both the continues and discrete formulation has some interesting mathematical properties, the convolution operator $(*)$ is both commutative and associative. There is also the convolution theorem which combines convolution with the Fourier transformation. While these properties are useful in theoretical work, they have little use in neural networks, it is more the idea behind convolution that is used. In practise neural networks use a finite kernel matrix w, and input image a. Furthermore, most frameworks use the cross-correlation operator instead of the convolution operator, the difference being that the kernel matrix is flipped. The operation is still called convolution among these frameworks [11], this wording is also adopted here and will be used for the rest of the thesis.

$$z(x, y) = (a * w)(x, y) = \sum_{i,\ j} a(x + i, y + j)w(i, j) \tag{2.3.3}$$

Finally, this thesis uses text as input which only has one dimension, thus a 1D-convolution is used.

$$z(t) = (a * w)(t) = \sum_{i} a(t + r\, i)w(i) \tag{2.3.4}$$

Convolution has shown to be a very powerful construct, this is primarily because of its locality and its parameter sharing. Locality means that one *pixel* in the output layer only depends on a few *pixels* in the input layer. This makes convolution fast to compute and reasonably easy to parallelize on a GPU. Parameter sharing means that one weight scalar in w is affected by many input and output *pixels*, this causes there to be a lot more data to estimate and optimize the parameter.

2.3.1 Padding

Having a finite input image raises the question of how to deal with the boundaries, where the kernel uses undefined data. In convolutional neural networks, the typical strategy is to either restrict the output image such that all the data is known, this is known as the "valid" approach. Alternatively one can add a zero-padding to the input image, such that the valid output image of the extended input image has the same size as the original input image, this is known as the "same" approach. Finally, there is the "full" approach where the input image is zero-padded as much as what is meaningful. The "full" approach is rarely used in neural networks.

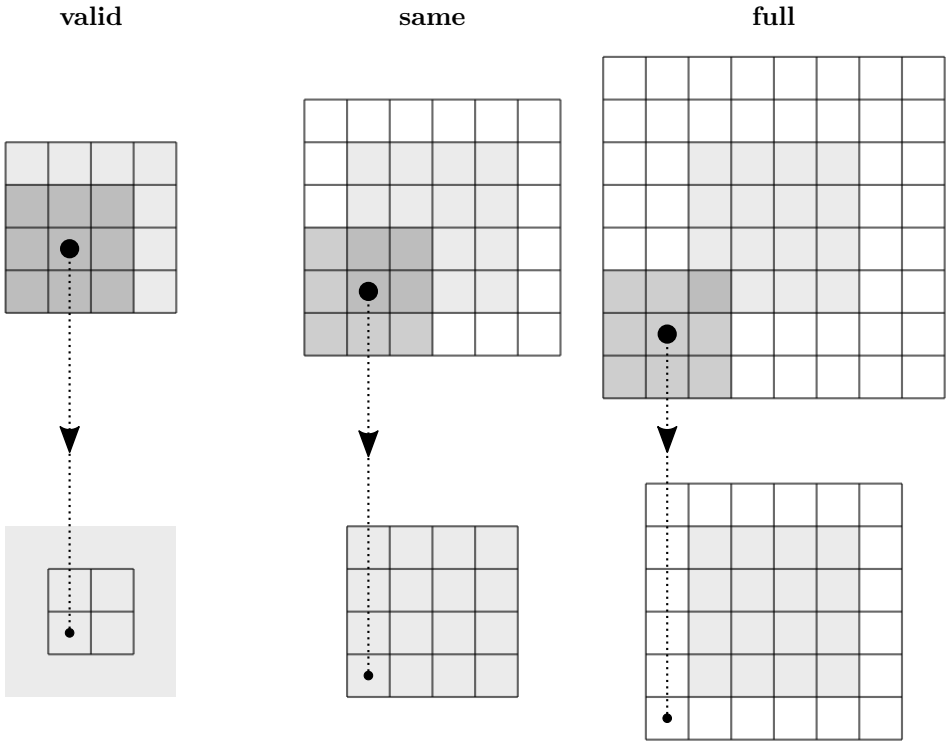


Figure 2.3.1: Shows 3 padding strategies, known as "valid", "same", and "full".

The "valid" padding approach is great because it doesn't assume anything about the data, such as if zero is an appropriate padding value. On the other hand, it makes the output image smaller which limits the number of convolutional layers one can apply.

The "same" padding approach is great because it doesn't change the output size. However, zero may not be an appropriate padding value. In the ByteNet model presented later "same" padding is used because it doesn't change the image size, this is a requirement for residual learning which will be discussed later.

2.3.2 Channels

Ordinary images a separated into 3 colors, red, green, and blue. This concept is generalized to what is called "channels", each color is one channel. A convolution will then merge all the channels into a new output image. For the output to have more than one channel, more than one convolution is applied in each layer. This can be expressed as:

$$z_{h_\ell}(t) = (a_{\ell-1} * w_{:,h_\ell})(t) = \sum_{h_{\ell-1}=1}^{H_{\ell-1}} \sum_i a_{h_{\ell-1}}(t+i) w_{h_{\ell-1},h_\ell}(i), \quad \forall h_\ell \in [1, H_\ell] \tag{2.3.5}$$

where $h_{\ell-1} \in [1, H_{\ell-1}]$ denotes the channel of the input image and $h_\ell \in [1, H_\ell]$ is the channel of the output image.

The use of channels causes a very big increase in the number of parameters, if w has the size W without channels, it now has the size $W \times H_{\ell-1} \times H_\ell$. This increase may seem absurd but in practice channels are extremely powerful. The typical behavior in convolutional neural networks is that each channel will represent some concept. In the first layer, each channel may represent edges in different directions. In the second layer, these edges may be combined to form shapes, such as circular, rectangular, etc.. The last layers may represent ears, noses, and hair, which is finally merged into faces. Neuroscientific research suggests that this is somewhat similar to what is happening in the human brain [11, chapter 9.10].

Similarly for text input one can imagine that letters are transformed into words and words into linguistic meaning. Unfortunately with text, these transformations are much harder to visualize, thus whether or not this is actually what is happening is unknown.

2.3.3 Dilated Convolution

Dilated convolution, also called atrous (with hole) convolution, is an extension on the convolution operator which makes the convolution kernel sparse. The sparseness is described with a dilation rate (r), a higher dilation rate means that the kernel is more sparse. A dilated convolution kernel (w_r) can be transformed to a dense convolution kernel w by using the Kronecker product (denoted with $\otimes$).

$$w = w_r \otimes \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} \tag{2.3.6}$$

The right-hand side matrix in the Kronecker product is a mostly-zero matrix of size $r \times r$ with just a single 1 in the top left element. The kernel matrix w is visualized in Figure 2.3.2.

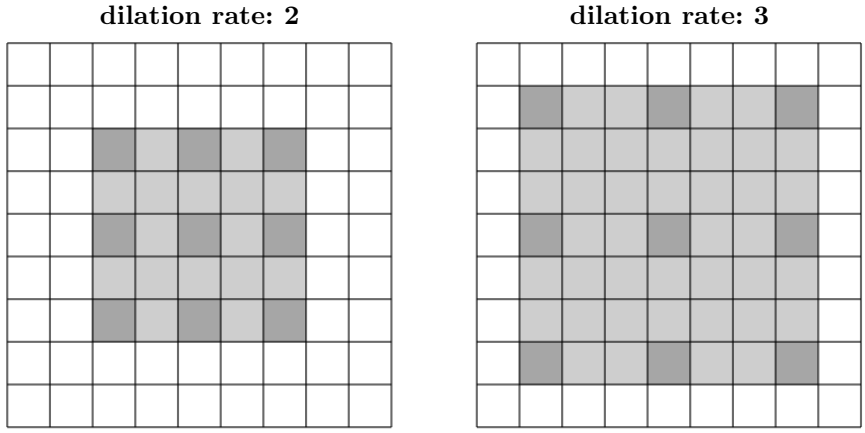


Figure 2.3.2: Shows dilated kernel with dilation rate $r = 2$ and $r = 3$.

Obviously converting the sparse kernel to a dense kernel with the Kronecker product is not a very computational efficient implementation. Instead, the convolution (cross-correlation) operator can be modified to support dilation directly.

$$z(x, y) = (a *_r w)(x, y) = \sum_{i,\ j} a(x + r\, i, y + r\, j) w(i, j) \tag{2.3.7}$$

While dilated convolution doesn't seem particularly useful, it becomes extremely powerful when multiple dilated convolution layers are stacked. For example, if the first layer has no dilation ($r = 1$) and a kernel size of 3×3, and the second layer has a dilation rate of 3 ($r = 3$) with same kernel size (3×3), the visible area of the second layer becomes much larger. This idea is called "hierarchical dilated convolution" and is the driving concept in the ByteNet model, it will be discussed later in section 2.6.2.

1D dilated convolution with channels can be expressed as:

$$z_{h_\ell}(t) = (a_{\ell-1} *_r w_{:,h_\ell})(t) = \sum_{h_{\ell-1}}^{H_{\ell-1}} \sum_i a_{h_{\ell-1}}(t + r\, i) w_{h_{\ell-1}, h_\ell}(i), \quad \forall h_\ell \in [1, H_\ell] \tag{2.3.8}$$

The backward pass for (2.3.8) is derived in Appendix B.2.

2.4 Improving Convergence Speed

A good optimization algorithm is essential for fitting a deep neural network. However, the convergence rate can often be improved by modifying the network architecture itself, such that the cost function is easier to optimize. These modifications do not radically alter the network, but rather modifies existing layers. The modifications can become the identity function through parameter optimization, thus they don't change the theoretical capabilities of the neural network.

2.4.1 Batch Normalization

Traditionally in feed forward neural networks, it has been the norm to standardize the input to have zero mean and unit variance.

$$\hat{x}_i = \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{VAR}[x_i] + \epsilon}}, \quad \forall i \in [1, I] \tag{2.4.1}$$

This standardization places the input to the sigmoid activation function in its linear domain ($\sigma(\epsilon) \approx \epsilon, \forall \epsilon \in [-1, 1]$), which is a reasonable starting point for the optimization. Batch normalization extends this idea, such standardizing is done before all activation functions in the neural network. This has positive consequences beyond limiting the sigmoid activation to its linear domain [17].

Consider a neural network with just one hidden layer:

$$Z_{h_L} = Z_{h_2} = \sum_{h_1=1}^{H_1} w_{h_1,h_2} \theta \left(\sum_{h_0=1}^{H_0} w_{h_0,h_1} x_{h_0} + b_{h_1} \right) + b_{h_2} \tag{2.4.2}$$

When optimizing the loss function, the parameters $w_{h_1,h_2}, w_{h_0,h_1}, b_{h_2}$, and b_{h_1} are all optimized simultaneously. Furthermore, the optimization of w_{h_1,h_2}, and w_{h_0,h_1} does directly depend on $\theta(z_{h_1})$ through the error term. This becomes an issue when the distribution of $\theta(z_{h_1})$ changes, because the updated w_{h_1,h_2}, and w_{h_0,h_1} assumes the original distribution. This change of the distribution of the internal activations is called an *internal covariate shift*. [17].

The *internal covariate shift* issue can be illustrated by considering a scalar $a = wx + b \sim \mathcal{N}(b, w)$, as it would appear in a very simple neural network, the sigmoid activation function is then applied on $\mathcal{N}(b, w)$ by using the *change of variable theorem*. Using this one can change w and b and observe how the sigmoid activation distribution changes (Figure 2.4.1).

While this issue isn't as theoretically significant for other activation functions, such as the ReLU activation function, it does often still have practical effects, particularly for deep neural networks.

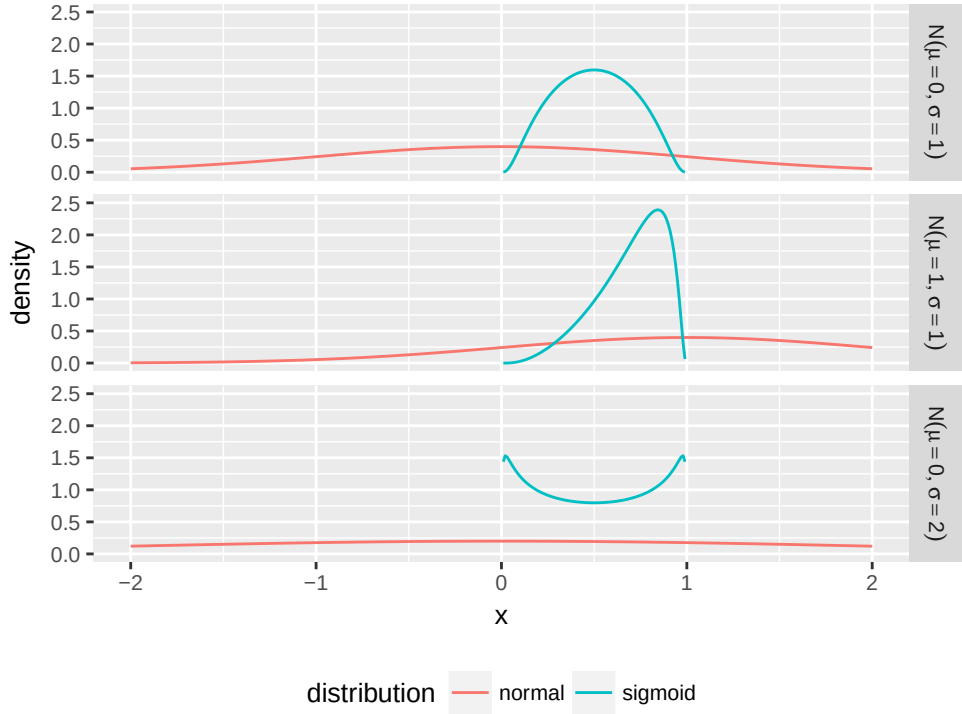


Figure 2.4.1: Shows $X \sim \mathcal{N}(\mu, \sigma)$ and sigmoid(X) calculated using the *change of variable theorem*.

A solution

The *internal covariate shift* issue can in practice be solved by using a small learning rate. However, this is not an optimal solution as it prolongs the optimization. Batch normalization is an alternative solution, that solves the issue by standardizing the input to the activation function. To truly standardize the input, the covariance matrix as well as its inverse square root, should be calculated. These calculations are very expensive, thus batch normalization makes a practical compromise by only standardizing using the variance. Figure 2.4.2 shows the sigmoid distribution after standardization.

$$\hat{z}_{h_\ell} = \frac{z_{h_\ell} - \mathbb{E}[z_{h_\ell}]}{\sqrt{\text{VAR}[z_{h_\ell}] + \epsilon}}, a_{h_\ell} = \theta(\hat{z}_{h_\ell}) \tag{2.4.3}$$

The expectation ($\mathbb{E}[z_{h_\ell}]$) and variance ($\text{VAR}[z_{h_\ell}]$) themselves are expensive to estimate over the entire dataset, thus it's only done over each mini-batch. This also makes it much more feasible to integrate the standardization into the backward pass. Note

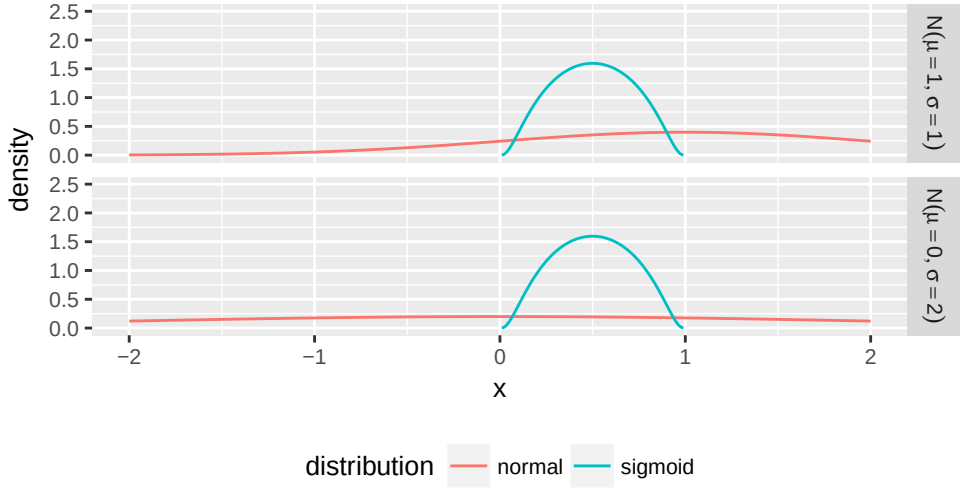


Figure 2.4.2: Shows $X \sim \mathcal{N}(\mu, \sigma)$ and sigmoid$(\hat{X})$, where $\hat{X}$ is standardized using batch normalization. The sigmoid distribution is calculated using the *change of variable theorem*.

also that because the expectation is subtracted, the bias b_{h_ℓ} in z_{h_ℓ} has no effect and should thus be omitted:

$$z_{h_\ell} = \sum_{h_{\ell-1}}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_{\ell-1}} \tag{2.4.4}$$

Finally, to allow batch normalization to become the identity function, two more parameters (γ, β) are added to the optimization problem:

$$\hat{z}_{h_\ell} = \gamma_{h_\ell} \frac{z_{h_\ell} - \mathbb{E}[z_{h_\ell}]}{\sqrt{\text{VAR}[z_{h_\ell}] + \epsilon}} + \beta_{h_\ell}, a_{h_\ell} = \theta(\hat{z}_{h_\ell}) \tag{2.4.5}$$

The backward pass for learning (w, γ, β) is rather complicated, but computationally very feasible as long as the mini-batch size is small. See Appendix B.3 for the backward pass.

In the special case that $\theta(\cdot)$ is multiplicative linear with respect to a scalar (i.e. $\theta(\alpha \hat{z}_{h_\ell}) = \alpha\theta(\hat{z}_{h_\ell})$) and the following layer isn't sensitive to a multiplication factor, then γ_{h_ℓ} can be removed from the optimization. A common case is where $\theta(\cdot)$ is the

ReLU function, in this case:

$$\begin{aligned}\alpha_{h_\ell} = \text{ReLU}(\hat{z}_{h_\ell}) &= \gamma_{h_\ell}\text{ReLU}\left(\frac{z_{h_\ell} - \mathbb{E}[z_{h_\ell}]}{\sqrt{\text{VAR}[z_{h_\ell}] + \epsilon}} + \frac{1}{\gamma_{h_\ell}}\beta_{h_\ell}\right) \\ &= \gamma_{h_\ell}\text{ReLU}\left(\frac{z_{h_\ell} - \mathbb{E}[z_{h_\ell}]}{\sqrt{\text{VAR}[z_{h_\ell}] + \epsilon}} + \tilde{\beta}_{h_\ell}\right)\end{aligned} \tag{2.4.6}$$

In the next layer, α_{h_ℓ} is then multiplied by some other weights that γ_{h_ℓ} can be merged into. This simplification can often be applied. It can be quite valuable as it removed some computations and further simplifies the loss curvature.

Inference

With an established backward pass, the network can easily be trained. However, there is still an open question about how inference should be done.

The inference should be deterministic once training is done, thus the ideal solution would be to use the estimated expectation and variance from the entire training dataset. However, because this calculation can be rather expensive a more practical solution is to use a moving average. Let's denote $\sigma^2_{\mathcal{B}_i}$ and $\mu_{\mathcal{B}_i}$ as the variance and mean estimate after mini-batch i. Then in addition to the optimization of the parameters w, γ, and β, the variance $\sigma^2_{\mathcal{B}_i}$, and mean $\mu_{\mathcal{B}_i}$ will also be updated during training.

$$\begin{aligned}\sigma^2_{\mathcal{B}_i} &= \lambda\sigma^2_{\mathcal{B}_{i-1}} + (1-\lambda)\text{VAR}[z_{h_\ell}] \\ \mu_{\mathcal{B}_i} &= \lambda\mu_{\mathcal{B}_{i-1}} + (1-\lambda)\mathbb{E}[z_{h_\ell}]\end{aligned} \tag{2.4.7}$$

At inference, $\hat{z}_{h_\ell}$ are then calculated using $\sigma^2_{\mathcal{B}_i}$ and $\mu_{\mathcal{B}_i}$.

$$\hat{z}_{h_\ell} = \gamma_{h_\ell}\frac{z_{h_\ell} - \mu_{\mathcal{B}_i}}{\sqrt{\sigma^2_{\mathcal{B}_i} + \epsilon}} + \beta_{h_\ell}, a_{h_\ell} = \theta(\hat{z}_{h_\ell}) \tag{2.4.8}$$

Weight sharing network

Because it is the weight changes that causes an *internal covariate shift*, the normalization should happen over all z_{h_ℓ} values that use these weights. Thus in RNN, the normalization should also be done over time, and in CNN the normalization should also happen over the "image". This works well for actual images. However, in RNN and CNN that describes a causal relation, the mean and variance at any time step will contain information from multiple time steps, which breaks the causality of the network. This issue is in practice solved by not normalizing over time, however, if the sequences aren't all of the same lengths then the mean and variance estimates for the last time step will be extremely poor.

2.4.2 Layer Normalization

Layer normalization attempts to solve the issues that exist when batch normalization is applied to causal weight sharing networks. It does this by not normalizing over the batch, but normalizing over the $\{z_{h_\ell}\}_{h_\ell=1}^{H_\ell}$ vector. The idea is that the output in one layer will often cause highly correlated changes in the summed inputs used in the next layer. Fixing the mean and the variance of the summed inputs should reduce this trend [18].

Normalizing over the summed inputs z_{h_ℓ} results in the following forward pass:

Activation:

$$
\begin{aligned}
z_{h_\ell} &= \sum_{h_{\ell-1}}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_\ell - 1} \\
\hat{z}_{h_\ell} &= \gamma_{h_\ell} \frac{z_{h_\ell} - \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} + \beta_{h_\ell} \\
a_{h_\ell} &= \theta\left(z_{h_\ell}\right)
\end{aligned}
$$

Statistics:

$$
\begin{aligned}
\mu_\ell &= \frac{1}{H_\ell} \sum_{h_\ell}^{H_\ell} z_{h_\ell} \\
\sigma_\ell^2 &= \frac{1}{H_\ell} \sum_{h_\ell}^{H_\ell} (z_{h_\ell} - \mu_\ell)^2
\end{aligned}
$$

Equation 2.4.9: Forward equations for Layer Normalization.

Note that the bias b_{h_ℓ} is excluded here for a different reason than what was the case in batch normalization. In batch normalization b_{h_ℓ} was a constant and is thus removed when the mean is subtracted. In layer normalization, the mean is over $h_\ell \in [1, H_\ell]$ and thus b_{h_ℓ} is no longer a constant. However the original reasoning for layer normalization "output of one layer will tend to cause highly correlated changes in the summed inputs" [18], does not include b_{h_ℓ} in "summed inputs" and thus the normalization should only happen over $\sum_{h_{\ell-1}}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_\ell - 1}$. As such $\hat{z}_{h_\ell}$ actually becomes

$$
\hat{z}_{h_\ell} = \gamma_{h_\ell} \frac{z_{h_\ell} - \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} + b_{h_\ell} + \beta_{h_\ell},
$$

but b_{h_ℓ} is redundant because of β_{h_ℓ}.

The backward pass for learning (w, γ, β) is like in batch normalization a bit complicated, see Appendix B.3 for the backward pass.

Similar to batch normalization, the $\hat{z}_{h_\ell}$ calculation can be simplified if $\theta(\cdot)$ is multiplicative linear (i.e. $\theta(\alpha\hat{z}_{h_\ell}) = \alpha\theta(\hat{z}_{h_\ell})$) and if γ_{h_ℓ} can be merged into weights in the following layer.

Properties

Batch normalization and layer normalization have somewhat similar properties, as shown in Table 2.4.1. *Weight matrix re-scaling invariance* is likely the most important property, as bad weight matrix initialization is often the cause of slow convergence.

	Weight matrix re-scaling	Weight matrix re-centering	Dataset re-scaling	Dataset re-centering
Batch norm	Invariant	No	Invariant	Invariant
Layer norm	Invariant	Invariant	Invariant	No

Table 2.4.1: Invariance properties when using batch or layer normalization. Also, note that batch normalization is invariant to *weight vector re-scaling* and layer normalization is invariant to *single training case re-scaling* [18].

Another difference between batch and layer normalization, is that in layer normalization it is not necessary to maintain a moving average over μ and σ^2 for inference, as these are estimated per observation.

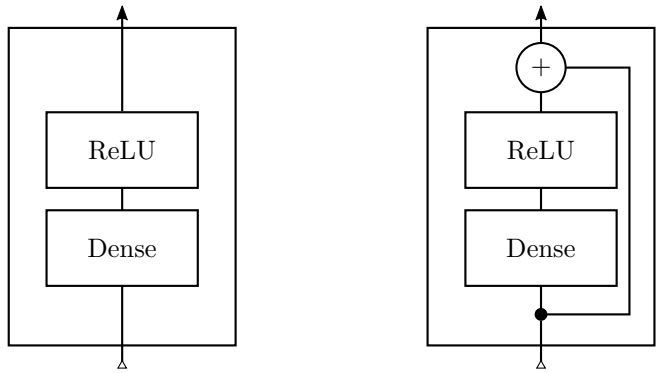
Experimental Results

In the original paper [18] they showed that layer normalization outperforms batch normalization in RNNs. Batch normalization is however the preferred choice in CNN, though layer normalization still performs better than the non-normalized baseline. It is theoretically unclear why layer normalization performs poorly on CNNs, but a possible explanation is that there is an underlying assumption that the hidden units a_{h_ℓ} make similar contributions, in CNN the hidden units typically represents very different things (e.g. ear, mouth, hair) thus some will be very inactive while others will be very active [18].

2.4.3 Residual Learning

The most sophisticated neural networks are typically rather deep networks with many layers, thus it is easy to think that "deeper is better". However it turns out that this is not necessarily true. First of all, there is a vanishing/exploding gradient problem, but recently with good normalization and weight initialization, this is becoming less significant. It turns out that adding layers to networks that already works well can degrade performance. This is not just a matter of overfitting, as also the training error degrades [19].

In theory, if there are too many layers the network should optimize such that some of the layers simply becomes the identity function. However, even modern day gradient-based optimizes are typically not able to find such a solution. Residual learning solves this problem, by changing the network architecture such that the identity solution is easier to find. If the desired function is denoted $\mathcal{H}(\mathbf{x})$, residual learning solves the issue by changing optimization problem such it should find $\mathcal{F}(\mathbf{x}) \stackrel{\text{def}}{=} \mathcal{H}(\mathbf{x}) - \mathbf{x}$ instead. This is done by transforming the layer to be $\mathcal{F}(\mathbf{x}) + \mathbf{x}$.



(a) Traditional Dense-ReLU layer **(b)** Residual Dense-ReLU layer

Figure 2.4.3: Comparison of a traditional and a residual Dense-ReLU layer.

The idea is that getting $\mathcal{F}(\mathbf{x}) = 0$ is a lot easier to solve for, when compared to $\mathcal{H}(\mathbf{x}) = \mathbf{x}$. For both the ReLU and sigmoid transformation, $\mathcal{F}(\mathbf{x}) = 0$ can be obtained by moving the weights to some extreme, while $\mathcal{H}(\mathbf{x}) = \mathbf{x}$ is drastically more difficult, particularly for the sigmoid case. If the layer really needs a non-trivial $\mathcal{H}(\mathbf{x})$ function, the optimizer simply needs to find $\mathcal{H}(\mathbf{x}) - \mathbf{x}$, which should not be much more difficult.

A downside of using a residual layer is that the output dimension must match the dimension of $\mathbf{x}$. However there are workarounds, for example, one can add an extra dense layer to change the dimensionality, of either the output or input dimension.

2.5 Sequential Networks

Traditional feed forward neural networks (FFNN) are only capable of taking a fixed-sized vector as input and outputting a fixed-sized vector. While it is theoretically possible to express a sentence in a fixed-sized vector by using a lot of zero-padding, it is not very practical. A much better approach is to use sequential networks, these are able to take a variable length sequence of fixed-sized vectors and outputting a sequence of fixed-sized vectors. There are different variations on this idea, they utilizes different strategies of aligning the input sequence with the output sequence. The most general type is called temporal classification [20], which allows any alignment between the input and target sequence. Temporal classification is necessary for language translation where the alignment often is unknown.

This class of problems are generally solved by creating a neural network that can fit the probability function $P(y_t|\{y_1, \ldots, y_{t-1}\}, \{x_1, \ldots x_{|S|}\})$, where $y_t \; \forall t \in [1, |T|]$ is the target sequence and $|T|$ is the length of the target sequence. Similarly $x_t \; \forall t \in [1, |S|]$ is the input (source) sequence of length $|S|$. The central idea is that $P(y_t|\cdot)$ knows the entire input sequence, but only the parts of the output sequence that came before y_t. This allows one to predict the entire sequence during inference, by iterating from $t = 1$ to $t = |T|$.

In training, the loss is constructed by considering the joint probability over all time-steps [20]:

$$P(y_1, \ldots, y_{|T|}) = \prod_{t=1}^{|T|} P(y_t|\{y_1, \ldots, y_{t-1}\}, \{x_1, \ldots, x_{|S|}\}) \tag{2.5.1}$$

This construction is also computationally convenient as it allows log-probabilities to be used instead of probabilities:

$$\log(P(y_1, \ldots, y_{|T|})) = \sum_{t=1}^{|T|} \log(P(y_t|\{y_1, \ldots, y_{t-1}\}, \{x_1, \ldots, x_{|S|}\})) \tag{2.5.2}$$

The ByteNet model, which will be discussed later in section 2.6, is able to fit the probability $P(y_t|\{y_1, \ldots, y_{t-1}\}, \{x_1, \ldots, x_{|S|}\})$. To understand the advantages of ByteNet and how it differs from existing word-based neural translation models, a short introduction to two popular models, Sutskever 2014 [21], and Bahdanau 2015 [9] is given. Later in the ByteNet chapter, the disadvantages will be discussed.

2.5.1 Sutskever Model

$$
\begin{aligned}
&\text{encoding:} \\
&\qquad \mathbf{h}_j = f(\mathbf{x}_j, \mathbf{h}_{j-1}) \quad \forall j \in [1, |S|] \\
&\text{decoder:} \\
&\qquad \mathbf{s}_i = \mathrm{g}(\mathbf{h}_{|S|}, \mathbf{s}_{i-1}) \\
&\qquad \mathbf{y}_i = \mathrm{softmax}(\mathbf{s}_i) \quad \forall i \in [1, |T|]
\end{aligned}
$$

Equation 2.5.3: The Sutskever 2014 model [21].

The Sutskever 2014 model [21] was one of the first neural machine translation models to shows state-of-the-art performance on the WMT 2014 dataset.

The general idea is to encode a sequence of words using a recurrent neural network, the last encoder state is then used to initialize the decoder. The decoder iterates using the previously predicted word. While this approach is mathematically elegant, encoding the source sentence (S) into a finite sized vector $h_{|S|}$ is in practice very difficult. The original implementation required an 8000 real-valued dimensional vector for the sentence encoding. They also limited the source vocabulary to 16000 words, and 8000 words for the target vocabulary [21].

Word-based neural machine translation models have shown to work well in practice, however, they also have obvious limitations. Words not in the vocabulary can't be translated. A common issue is names, which in character-based models are very easy to translate because they require no translation. The softmax of a large vocabulary is also expensive, in the original Sutskever implementation they used 4 GPUs for just the softmax and 4 more GPUs for the rest of the network. This again can be solved by using character-based models because the "vocabulary" is just the different letters, which there are a lot fewer of.

Using characters instead of words in the Sutskever model may seem like a good idea at first, however the source and target sequences become much longer. Long sequences are difficult to encode and decode because the state is updated in each iteration, thus the state produced for the beginning of the sentence is easily forgotten. In theory LSTM units solves this issue, but in practice it still exists. In particular, Sutskever reversed the input sentence to get better performance. If there were no memory issues, reversing the input sentence should have no effect.

2.5.2 Bahdanau Attention Model

$$
\begin{aligned}
&\text{encoding:} \\
&\qquad \mathbf{h}_j = f(\mathbf{x}_j, \mathbf{h}_{j-1}) \quad \forall j \in [1, |S|] \\
&\text{attention:} \\
&\qquad e_{ij} = a(\mathbf{s}_{i-1}, \mathbf{h}_j) \\
&\qquad \boldsymbol{\alpha}_i = \text{softmax}(\mathbf{e}_i) \\
&\qquad \mathbf{c}_i = \textstyle\sum_{t=1}^{T} \alpha_{it} \mathbf{h}_t \\
&\text{decoder:} \\
&\qquad \mathbf{s}_i = \text{g}(\mathbf{c}_i, \mathbf{s}_{i-1}) \\
&\qquad \mathbf{y}_i = \text{softmax}(\mathbf{s}_i) \quad \forall i \in [1, |T|]
\end{aligned}
$$

Equation 2.5.4: The attention based Bahdanau 2015 model [9].

Bahdanau et. al. solved the memory issue by letting the decoder look at selected parts of the encoded state sequence. It does this through what is called an attention mechanism. The attention $\boldsymbol{\alpha}_i$ is a weight vector, that is used in a weighted mean calculation over the encoded states $\mathbf{h}_t$. The weights are calculated using a sub-network, that depends on the previous output state $\mathbf{s}_{i-1}$ and the encoding states. The weighted mean is then used to calculate the next output state $\mathbf{s}_i$.

The attention mechanism essentially recalculates a new encoding vector for each output state. This creates what called a resolution preserving encoding, that is an encoding which size depends on the source sequence, this is different from the Sutskever model that uses a fixed-sized vector. The Bahdanau et. al. model is word-based and achieves state-of-the-art performance [9]. On the surface, there is nothing that prevents the Bahdanau et. al. model from being character-based, but by looking at the computational complexity it becomes clear that using characters is not a viable solution.

In each iteration on the output sequence, the attention model needs to calculate the weighted mean over the encoding sequence, this takes $\mathcal{O}(|S|)$ time. These calculations can not be reused in the next iteration, because they also depend on the output state from the previous iteration s_{i-1}. The attention vector thus needs to be recalculated for each output state, resulting in a computational complexity of $\mathcal{O}(|S||T|)$.

Having this quadratic computational complexity $\mathcal{O}(|S||T|)$, means that using characters instead of words dramatically increases the running time.

2.6 ByteNet

The ByteNet model is an alternative approach to neural machine translation, it specifically focuses on having linear computational complexity, having a resolution preserving encoding, and be a character-level translation model [10]. This is somewhat different from existing popular models such as the Sutskever 2014 model [21] and the attention based Bahdanau 2015 model [9].

Note that the model presented here is a simplified version of the original ByteNet model [10]. First, there are no bags of characters, only individual characters are used as the sentence representation. Secondly, the sub-batch normalization in the decoder is replaced by layer normalization, besides being a simpler solution than sub-batch normalization this could also improve the model performance [18]. At last, the network has less weights and fewer layers than in the original ByteNet model.

2.6.1 ByteNet Residual Block

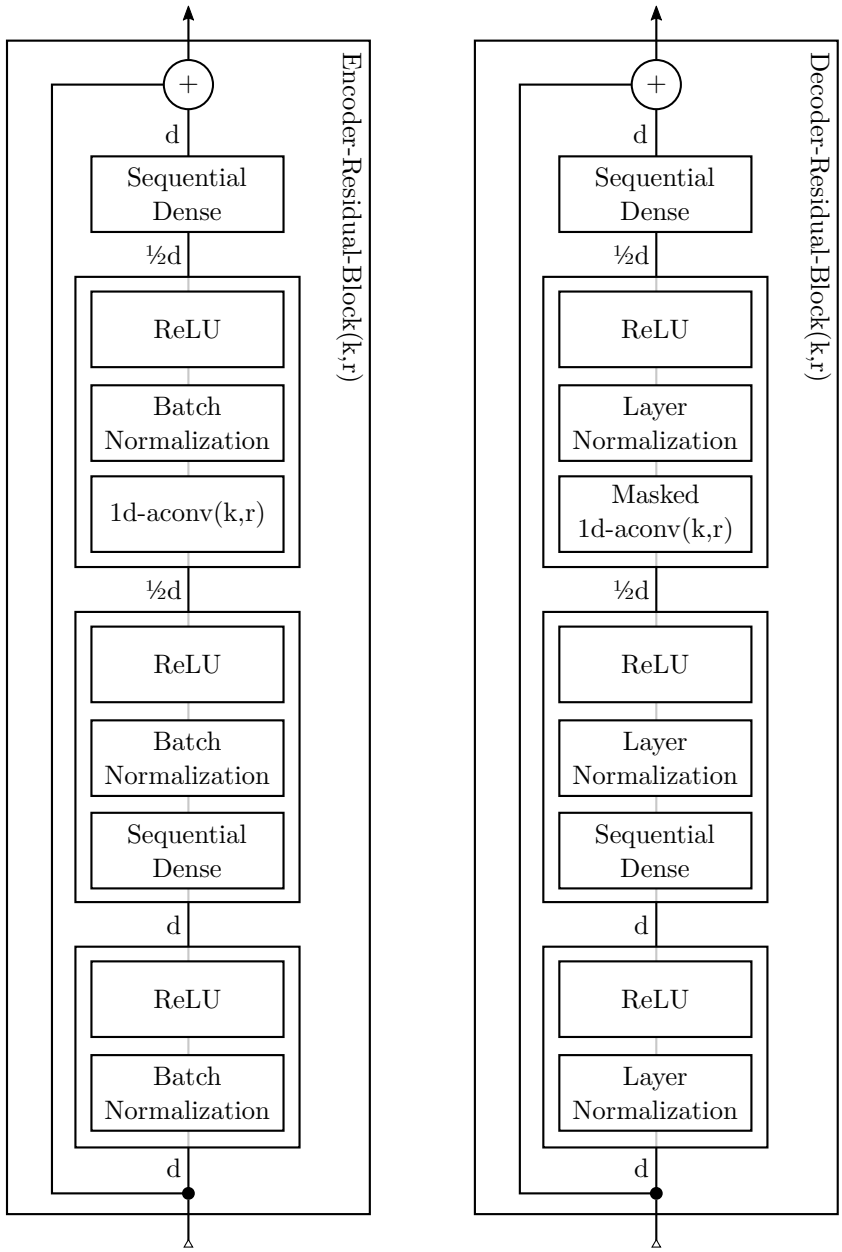
Before explaining the full ByteNet network, a central component of the ByteNet model called the *ByteNet residual block* needs to be explained first. There are two variations of this an *Encoder Residual Block*, and a *Decoder Residual Block*. First the encoder version is explained and then some slight alterations of this will turn it into the decoder version. For a visual representation see Figure 2.6.1.

Encoder Residual Block

The *ByteNet residual block* is at its core a dilated convolution, but adds upon this many of the ideas presented in section 2.4. The entire block is a *residual layer*, meaning that that the input is added to the final output of the block. The block can be logically grouped into a set of sub-blocks, a normalization block, a dimensional reduction block, a dilated convolution block, and finally a dimensional restoration block.

The normalization block consists of batch normalization and a ReLU activation. For this to make sense there must be some parameters before the activation such that the network can control the ReLU. This will either be an embedding lookup or another *residual block*.

The dimensional reduction block reduces the dimensionality of the dataset from d_{enc} dimensions to $\frac{1}{2}d_{enc}$ dimensions. This is done through the usual dense layer, batch normalization, and ReLU activation block sequence. Note that the input is a sequence of vectors, a vector for each character, thus the dense matrix multiplication is performed on each vector in the sequence. This can also be described as a normal convolution with a kernel width of 1.



(a) Residual Block used in encoder. **(b)** Residual Block used in decoder.

Figure 2.6.1: The residual blocks used in the ByteNet model. The blocks are denoted by Encoder-Residual-Block(k, r) and Decoder-Residual-Block(k, r), where k is the kernel width and r is the dilation rate.

The dilated convolution block depends on two parameters, the kernel width k, and the dilation rate r (3 if dimension size d_{enc} is included). ByteNet specifies a specific structure for these parameters, but these choices only make sense from the full network perspective, thus they will be discussed later. Besides being a dilated convolution the sub-block is quite normal, as it also performs a batch normalization and ReLU activation. The convolution also maintains the dimensionality, which at this point is $\frac{1}{2}d_{enc}$.

For the *residual layer* to make sense, the output dimension must be equal to the input dimension, thus the final sub-block is just a dense layer that transforms from $\frac{1}{2}d_{enc}$ dimensions to d_{enc} dimensions. This sub-block does not contain either a batch normalization, or a ReLU activation. Though these are implicitly performed if another *residual block* follows.

The number of weights from all these sub-blocks in the *encoder residual block*, excluding the bias and normalization terms, are:

$$k\left(\frac{d_{enc}}{2}\right)^2 + \frac{1}{2}d_{enc}^2 + \frac{1}{2}d_{enc}^2 = \left(\frac{k}{4} + 1\right)d_{enc}^2 \tag{2.6.1}$$

Decoder Residual Block

The decoder is very similar to the encoder, the differences exist such that the decoder can't look into the future of the target sequence. If the decoder was allowed to look into the future it would be impossible to perform inference, when the target sequence is unknown. To prevent future inspection two changes are made, the dilated convolution is masked, and batch normalization is replaced with layer normalization.

Masking the convolution means that only the part of the convolution kernel that looks at the past is preserved. This is equivalent to fixing the right side (the future side) of the preserved kernel to zero. Masking the kernel reduces the number of weights a bit, but otherwise there is no difference. The number of weights are:

$$\left(\frac{k-1}{2} + 1\right)\left(\frac{d_{dec}}{2}\right)^2 + \frac{1}{2}d_{dec}^2 + \frac{1}{2}d_{dec}^2 = \left(\frac{k+1}{8} + 1\right)d_{dec}^2 \tag{2.6.2}$$

2.6.2 Hierarchical Dilated Convolution

Each *ByteNet residual block* can conceptually be seen as just a modified dilated convolution. This conceptual model is useful for understanding the main trick that makes *ByteNet* so powerful, called *hierarchical dilated convolution*.

The idea behind *hierarchical dilated convolution* is that by exponentially increasing the dilation rate in stacked dilated convolution layers, the effective kernel width will

exponentially increase for each layer, while the amount of weight in the kernel only increases linearly. The encoder part (bottom half) of figure 2.6.2 helps to understand this idea in detail.

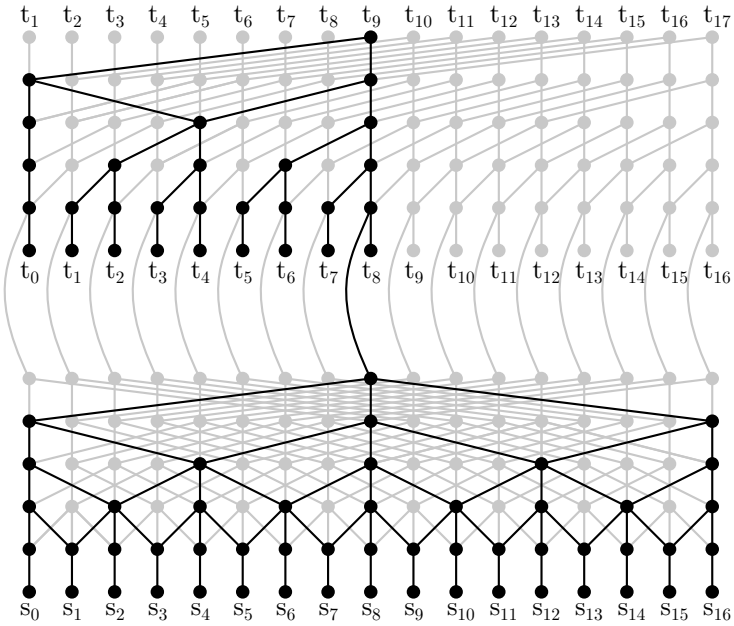


Figure 2.6.2: A simplified version of the ByteNet model, showing the interaction between the decoder and encoder.

Let the dilation rate for the encoding layers $\ell \in [1, L_{enc}]$ be denoted r_ℓ, the exponential increasing dilation rate used in the ByteNet model is then $r_\ell = 2^{\ell-1}$. The top layer (L_{enc}) will thus have a width of $(k-1)2^{L_{enc}-1}+1$. The effective total width is slightly larger because the layer below has a width of $(k-1)2^{L_{enc}-2}+1$. This pattern continues down to the first layer, resulting in an effective total width of:

$$\sum_{\ell=1}^{L_{enc}} (k-1)2^{\ell-1}+1 = (k-1)(2^{L_{enc}}-1)+1 \tag{2.6.3}$$

However, the kernel size is $k \times d$ for all layers, thus the total number of weights are:

$$\sum_{\ell=1}^{L_{enc}} kd = L_{enc}kd \tag{2.6.4}$$

In the original ByteNet model they use 5 dilated convolution layers, each with a kernel width of 5, this results in an effective kernel width of 125 characters (actually it is 373 character wide because they repeat the pattern 3 times). Having a

wide but computationally cheap kernel is extremely important for a character-based translation model such as ByteNet. In an attention-based translation model, such as the Bahdanau model [9] the width is in theory arbitrarily wide (though in practice very wide attention mechanisms works poorly), however, as discussed previously this approach causes a quadratic computational complexity (section 2.5.2). By using *hierarchical dilated convolution* ByteNet manages to keep the complexity linear, while still having a very wide kernel. In practice sentences have a limited length and thus a hierarchical dilated convolution approach can be comparable to an attention approach. In the WMT 2014 en-de dataset, the longest sentences are 400 characters wide (Figure 2.6.3).

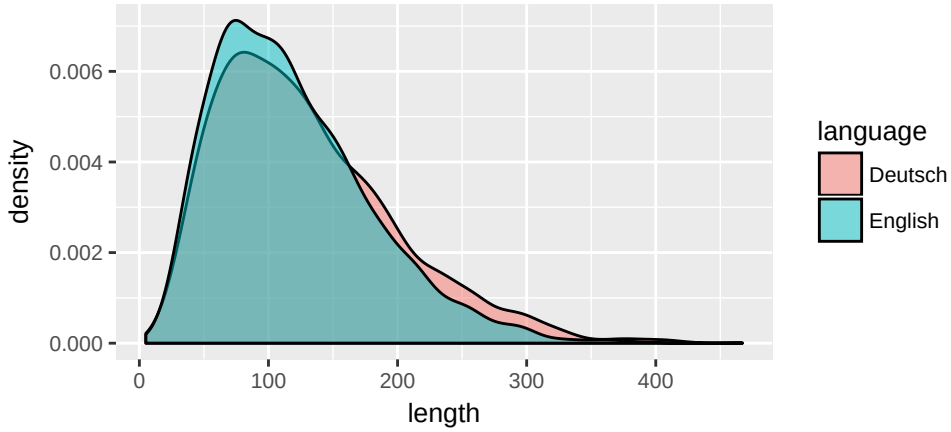


Figure 2.6.3: Density estimate of the sentence length in the WMT 2014 en-de dataset.

2.6.3 The Network

The full *ByteNet* model combines the *residual blocks* with the *hierarchical dilated convolution* as discussed earlier. The *hierarchical dilated convolution* pattern is then repeated 3 times for both the encoder and decoder, as visualized in figure 2.6.4.

The input to the encoder is the source sequences mapped using an embedding matrix. The input to the decoder is the encoded sequence concatenated with the previous target prediction. This target prediction is then also mapped using a different embedding matrix. Note in particular that because the encoded vector is concatenated with the target sequence vector, the internal dimensionality of the decoder is twice that of the encoder. The decoded output is finally transformed using a dense layer such the output has vocabulary sized dimensionality. After this, a softmax is used to transform the output into the probability domain.

An interesting effect of this network is that all layers are invariant to adding a bias term, as well as the γ term in the normalization. The encoder embedding and encoder layers are invariant because of batch normalization. The decoder embedding and decoder layers are invariant because of layer normalization. The final dense layer is invariant because of the softmax in the end.

Training versus Inference

The ByteNet model actually uses a different network for training than it does for inference. This is not strictly necessary, as one could use the same network for both training and interference like in an FFNN, however for sequential models the optimization can often converge faster if the target sequence is feed into the model.

Feeding the target sequence to the network is an optimization trick that may seem counter-intuitive at first. The idea is that because the prediction of the current character depends on the prediction of the previous characters ($\mathbf{y}_t = f(\mathbf{x}_{1:|S|}, \mathbf{y}_{1:t-1})$), then instead of letting the prediction cascade, which will also cause prediction errors to cascade, $\mathbf{y}_{1:t-1}$ is replaced with the known target sequence such that $\mathbf{y}_t = f(\mathbf{x}_{1:|S|}, \mathbf{t}_{1:t-1})$.

This trick also has the added benefit that training can be parallelized over target sentence and not just the observations and source sentence. There are some indications of this being suboptimal, and that one should use a hybrid of the two approaches [22]. However, the ByteNet model is unique in that it allows for parallelization over the target sequence. This would not be possible with a hybrid solution.

Future ByteNet models

During this thesis the authors of ByteNet have released a revision of the original paper.

In the new model, 6 stacks of 5 *ByteNet residuals blocks* are used in both the encoder and decoder. This is twice that of the original model and allowed them to replace the bag-of-characters encoding with a normal character encoding.

In the original model they used a sub-batch normalization, to preserve causality. In the new model this was simplified by using layer normalization, like in this thesis, except they also changed the encoder to use layer normalization.

Finally the new model uses 800 as the encoding dimensionality, where the original model used a more tuned value of 892 dimensions.

None of these changes are particularly relevant. Some of the changes was independently applied, such as using layer normalization and removing the back-of-characters encoding. Other changes such as increasing the number of *ByteNet residuals blocks* can't be applied, because DTU doesn't have the resources to support it.

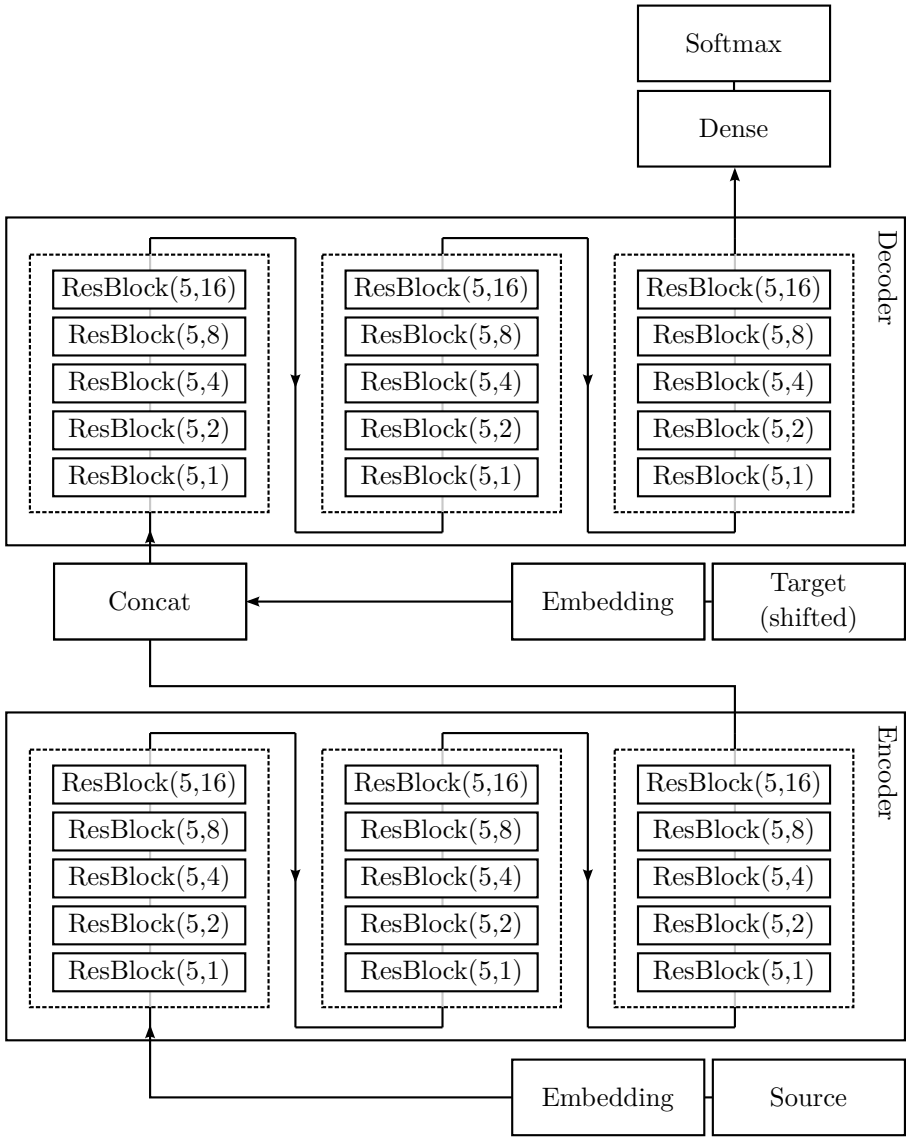


Figure 2.6.4: The training network for the ByteNet Model, where the shifted target sequence is feed into the network. In the inference network, the shifted target sequence is replaced by the argmax$(\cdot)$ of the softmax output. ResBlock(k, r) in the encoder refers to the Encoder-Residual-Block(k, r), in the decoder it refers to the Decoder-Residual-Block(k, r).

2.7 Semi-Supervised Learning for NMT

An often limiting factor of neural machine translation (NMT) is the size and quality of the bilingual dataset. Monolingual dataset are on the other hand abundant and have been used to create powerful language models [6]. While language models can be used to argument translation models it requires non-trivial modifications to the neural translation architecture. Recent efforts within generative adversarial networks (GAN) [23] have also shown GAN to be a powerful concept for training with unlabeled data. However GANs are still very difficult to get working, especially for classification problems like translation [8].

The strategy presented here is much more pragmatic than GAN and does not require major alteration to the translation model. The strategy uses two ideas, translating from for example German to English and back to German should result in the same German sentence as the original. Secondly, there may be more than one valid translation from German to English, thus the training should find the k most likely translations and ensure that they are all translated back to the original German sentence. This strategy has been shown to work well in practice [7]. The advantage of this approach is that it doesn't depend on a specific translation model, but should in theory work for any translation model. In the original article by Cheng et. al. [7] they uses The Bahdanau et. al. [9] word-based translation model, in this thesis the ByteNet [10] character-based translation model will be used instead.

2.7.1 Semi-Supervised loss

To train a model, that translate from language $\mathbf{x}$ to $\mathbf{y}$ and back to $\mathbf{x}$, two translation model are required. These translation models are denoted as $P(\mathbf{y}|\mathbf{x};\overrightarrow{\mathbf{w}})$ and $P(\mathbf{x}|\mathbf{y};\overleftarrow{\mathbf{w}})$, where the arrow on $\mathbf{w}$ represent the translation direction.

When just considering the bilingual dataset, the two models can be trained independently, as they don't share any parameters. However when the monolingual dataset is added this will no longer be the case. Thus the cross entropy loss is combined to:

$$\mathcal{L} = -\left(\log(P(\mathbf{y}|\mathbf{x};\overrightarrow{\mathbf{w}})) + \log(P(\mathbf{x}|\mathbf{y};\overleftarrow{\mathbf{w}}))\right) \tag{2.7.1}$$

Note that training wise this loss is equivalent to training the model independently, because:

$$\frac{\partial\mathcal{L}}{\partial\overrightarrow{\mathbf{w}}} = -\frac{\partial\log(P(\mathbf{y}|\mathbf{x};\overrightarrow{\mathbf{w}}))}{\partial\overrightarrow{\mathbf{w}}}, \qquad \frac{\partial\mathcal{L}}{\partial\overleftarrow{\mathbf{w}}} = -\frac{\partial\log(P(\mathbf{x}|\mathbf{y};\overleftarrow{\mathbf{w}}))}{\partial\overleftarrow{\mathbf{w}}} \tag{2.7.2}$$

To add a loss for the monolingual datasets to (2.7.1) two additional translation models are used, these are represented as $P(\mathbf{y}'|\mathbf{y};\overrightarrow{\mathbf{w}},\overleftarrow{\mathbf{w}})$ and $P(\mathbf{x}'|\mathbf{x};\overrightarrow{\mathbf{w}},\overleftarrow{\mathbf{w}})$.

The "monolingual" translation models are created using the two bilingual model:

$$\begin{aligned} P(\mathbf{y}'|\mathbf{y}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}) &= \sum_{\mathbf{x}} P(\mathbf{y}'|\mathbf{x}; \overrightarrow{\mathbf{w}}) P(\mathbf{x}|\mathbf{y}; \overleftarrow{\mathbf{w}}) \\ P(\mathbf{x}'|\mathbf{x}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}) &= \sum_{\mathbf{y}} P(\mathbf{x}'|\mathbf{y}; \overleftarrow{\mathbf{w}}) P(\mathbf{y}|\mathbf{x}; \overrightarrow{\mathbf{w}}) \end{aligned} \tag{2.7.3}$$

The mathematics behind (2.7.3) is a trivial marginalization over all possible translations from $\mathbf{y}$ to $\mathbf{x}$, and all possible translations from $\mathbf{x}$ to $\mathbf{y}$, respectively. From a training perspective $\mathbf{x}'$ is the same as $\mathbf{x}$, and vise versa for $\mathbf{y}$, the added notation is to clarify that the translation model is far from trivial, because it is heavily constrained through the other language.

Calculating the sum over all possible sequences $\mathbf{x}$ and $\mathbf{y}$ is not a feasible task, since the number of combinations is exponentially increasing with the sequence length. The practical approach is to instead approximate the sum, by only summing over the k most likely sequences. Finding the k most likely sequences is in itself an NP-problem, but it can be reasonably approximated by using a heuristic called BeamSearch. This heuristic will be discussed in details later.

Using these "monolingual" translation model the loss function is extended to:

$$\begin{aligned} \mathcal{L} = -\big(&\log(P(\mathbf{y}|\mathbf{x}; \overrightarrow{\mathbf{w}})) \\ &+ \log(P(\mathbf{x}|\mathbf{y}; \overleftarrow{\mathbf{w}})) \\ &+ \lambda_1 \log(P(\mathbf{y}'|\mathbf{y}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}})) \\ &+ \lambda_2 \log(P(\mathbf{x}'|\mathbf{x}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}))\big) \end{aligned} \tag{2.7.4}$$

λ_1 and λ_2 are hyper-parameters for balancing the bilingual and monolingual losses. By this logic λ_1 should equal λ_2, however the original paper [7] showed that in practice there little difference between using both monolingual losses and only using one of them. Since it much more performant to use just one monolingual model either λ_1 or λ_2 can be set to zero.

In terms of practical computation $\log(P(\mathbf{y}'|\mathbf{y}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}))$ and $\log(P(\mathbf{x}'|\mathbf{x}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}))$ can be calculated numerically stable using a trick that is explained in Appendix C.2. The gradient of $\log(P(\mathbf{y}'|\mathbf{y}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}))$ and $\log(P(\mathbf{x}'|\mathbf{x}; \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}))$ can also be calculated directly from $\log(P(\mathbf{y}'|\mathbf{x}; \overrightarrow{\mathbf{w}}))$ and $\log(P(\mathbf{x}|\mathbf{y}; \overleftarrow{\mathbf{w}}))$ respectively, without any exponential conversion, see Appendix B.5. Although the latter is not done directly, as TensorFlow automatically calculates the gradients.

2.7.2 Beam Search

Getting the k most likely sequences given a translation model $P(\mathbf{y}|\mathbf{x})$ is not a trivial task. By encoding $\{x_1, \ldots, x_{|S|}\}$ one can calculate $P(y_1|\{x_1, \ldots, x_{|S|}\})$ for each possible symbol that y_1 can attain. For each y_1 symbol, $P(y_2|\{y_1\}, \{x_1, \ldots, x_{|S|}\})$ can

then be calculated for each possible symbol that y_2 can attain. This recursion can be repeated until one has k sequences that all ended with an `<EOS>` symbol and are more likely than all other sequences. This procedure works in theory, but the running time and memory is exponentially increasing with the sequence length, thus this is not a very practical approach.

BeamSearch is a heuristic for getting the k most likely sequences. On a theoretical level BeamSearch is contrary to its fancy name fairly intuitive, the difficulty lies in how it is implemented.

BeamSearch has a parameter called the *beam size* (denoted b), this size determines the number of sequences that is kept track of. It limits the number of sequences (called *paths*) by assigning each new path in the next iteration a score, where the b best new sequences are kept and used in the next iteration. To initialize the *paths* the top b outputs from $P(y_1|\{x_1, \ldots, x_{|S|}\})$ are used. After either a maximum number of iterations, or when the top k sequences in the set of memorized paths (called the *beam*) have ended with an `<EOS>` symbol, the BeamSearch algorithm can stop.

Algorithm 2 BeamSearch algorithm, specialized for scoring by sequence probability.

function BeamSearch($P(\mathbf{y}|\mathbf{x}), \mathbf{x}, b$)
  Initialize **`paths`** *from top b selection from* $P(y_1|\{x_1, \ldots, x_{|S|}\})$
  $i \leftarrow 2$
  repeat
    for $\{y_1, \ldots, y_{i-1}\}$ in `paths` **do**
      Compute probabilities of new paths $\{y_1, \ldots, y_i\}$ *conditioned on the old path* $\{y_1, \ldots, y_{i-1}\}$.
    Update **`paths`** *by selecting the top b new paths by the joint probability* $P(\{y_1, \ldots, y_i\}|\{x_1, \ldots, x_{|S|}\})$.
    $i \leftarrow i + 1$
  until `<EOS>` $\in$ `paths`
  return `paths`

For $b = 1$ the BeamSearch algorithm becomes a completely greedy search that just takes the most likely symbol in each iteration. For $b = \infty$ BeamSearch becomes the complete search that runs in exponential time.

BeamSearch in practice

When using BeamSearch in practice there is quite a few things that needs to be taken into account. The primary issues are:

1. How to deal with log probabilities instead of probabilities for numerical stability.

2. How to operate on a mini-batch of sequences.

3. How to deal with output sequences of different length.

Beyond these issues there is also a big challenge in implementing BeamSearch using tensor operations, as BeamSearch is highly state and sequence dependent. These details will not be discussed as those are implementation specific.

Algorithm 3 BeamSearch algorithm, specialized for NMT.

function BEAMSEARCH($P(\mathbf{y}|\mathbf{x}), \mathbf{x}, state_0, b$)
  $logpropsum_0 \leftarrow \text{ZERO}(n, b)$
  $ended_0 \leftarrow \text{FALSE}(n, b)$
  Duplicate $state_0$ *b times.*
  $i \leftarrow 1$
  repeat
    pack state ($\mathbf{s}_{i-1}$) *so it looks like* $(n \cdot b)$ *observations.* ▷ 2
    $state_i, logits_i \leftarrow \text{MODEL}(\mathbf{s}_{i-1}, \mathbf{x})$ ▷ 2
    unpack $state_i$ *and* $logits_i$ *to it original shape of* n *observations.* ▷ 2

    $logprop_i \leftarrow \text{LOGSOFTMAX}(logits_i)$ ▷ 1
    Mask $logprop_i$ *to be* 0 *for sequences that has already ended* ($ended_i$). ▷ 3

    $logpropsum_i, beam_i, labels_i \leftarrow \text{TOPK}(logpropsum_{i-1} + logprop_i)$ ▷ 1
    Select and duplicate $logpropsum_i$ *and* $ended_{i-1}$ *using the indices* $beam_i$.
    $ended_i \leftarrow ended_{i-1} \vee labels_i =$ `<EOS>` ▷ 3

    $i \leftarrow i + 1$
  until ALL($ended_i$)
  return EXTRACTPATH($state_i$)

Algorithm 3 shows a BeamSearch algorithm specialized for neural machine translation, where ▷ shows where each issue is solved. The issues are solved in the following way:

1. Using log probabilities instead of probabilities is generally not a huge challenge. The logits (unnormalized log probabilities) that the model returns needs to be normalized without loss of precision, this can be done using a log-softmax function (appendix C.1). Once the logits are normalized, the joint log probability for the sequence can be calculated by additively accumulating the log probabilities.

2. To ensure high performance multiple sequences are processed in parallel. On the surface this is fairly trivial to do, simply restructure the state such that the different path in each beam looks like observations. Then run the model and

restructure back to the original state shapes. The complexity lies in how to deal with for example batch normalization. By restructuring the model input the observations are no longer independent because they originate from the same observations. There is no good solution to this, the solution used in this thesis is to not update the batch normalization when beam searching. The batch normalization should simply be treated like when performing inference. Doing this shouldn't be an issue, because the mean and variance tends to converge quite fast.

3. Because the loss is masked using the target sequence, the model has no way of knowing $P(y_i = \texttt{<NULL>} | y_{i-1} = \texttt{<EOS>}) = 1$. Ensuring this relation is essential, otherwise sequences that are initially very likely and have ended, can become very unlikely for no good reason. This is solved with the `ended` flag, this is updated by looking at the latest labels for each path in each beam.

CHAPTER 3

Results

3.1 Problems and Datasets

3 datasets will be used to evaluate the ByteNet and the semi-supervised ByteNet models. The WMT Translation Task provides a large corpus of text in different datasets. Two datasets are used in this thesis, the Europarl v7 dataset, and the WMT NewsTest dataset. For validating the implementation and discussing the model a very simple synthetic dataset is also used, this dataset is named "Synthetic Digits".

3.1.1 WMT Translation Task

Each year WMT holds a conference on machine translation, this works as a series of workshops and competitions on different translation tasks. One of these translation tasks is on news translation [24]. The task is to translate paragraphs from news articles to another language. The translations are evaluated on the WMT NewsTest dataset, that is from news articles collected over a specific year. The news translation task is the primary translation task that neural machine translation (NMT) papers evaluate their models on [9, 10, 21].

0	source	Die Premierminister Indiens und Japans trafen sich in Tokio.
	target	India and Japan prime ministers meet in Tokyo
1	source	Pläne für eine stärkere kerntechnische Zusammenarbeit stehen ganz oben auf der Tagesordnung.
	target	High on the agenda are plans for greater nuclear co-operation.
2	source	Berichten zufolge hofft Indien darüber hinaus auf einen Vertrag zur Verteidigungszusammenarbeit zwischen den beiden Nationen.
	target	India is also reportedly hoping for a deal on defence collaboration between the two nations.

Table 3.1.1: Examples from the WMT NewsTest 2015 de-en dataset.

The WMT NewsTest is about 3000 sentences each year, this is not enough data to build a good translation model. To that end WMT provides additional datasets for

training, most importantly is the Europarl v7 dataset. The Europarl dataset contains high-quality translations of various of documents from the European Parliament [25]. While this dataset is a huge high-quality dataset it is also heavily biased. For example, some word-grams like "The European Parliament" appears much more often that in the NewsTest dataset.

A few of the sentences in the Europarl v7 dataset are very long (3000+ characters), including these long sentences causes out-of-memory issues on the GPU, thus only sentence pairs where both the source and target sentences are less than 500 characters long are included. This sequence length was chosen because no sentence in the WMT NewsTest dataset is longer than 500 characters.

After removing sentences that are too long, the Europarl v7 dataset contains 1,901,056 English-German sequence pairs.

0	source	Wir sind nach wie vor der Ansicht, daß der wirtschaftliche und soziale Zusammenhalt ein zentrales Ziel der Union ist.
	target	We still feel that economic and social cohesion is one of the Union' s fundamental objectives.
1	source	Den Kommissar möchte ich auffordern, in diesen beiden Bereichen tätig zu werden und uns dabei mit einzubinden.
	target	These are two areas of action which I invite the Commissioner to set up and in which I would ask him to involve us.
2	source	Dies zwingt das Europäische Parlament, den Herrn Kommissar und die Kommission zu entschlossenem strategischen Handeln.
	target	This fact means that the European Parliament, the Commissioner and the Commission must act decisively and strategically.

Table 3.1.2: Examples from the Europarl v7 de-en dataset.

3.1.2 Synthetic Digits

While the WMT Translation Task is a good problem to measure a translation model on, it is a complex problem that takes a long time for a neural network to learn. It is also not obvious if a model will be able to solve this problem at all. In particular, the semi-supervised ByteNet model has never been attempted before. It is thus meaningful to validate that the models can solve a simple problem before attempting to solve more complex problems, like the WMT Translation Task.

The Synthetic Digits dataset is simply a uniformly random sequence of integers. In the source, each digit is spelled out in English, in the target each digit is represented by a symbol. The sequence length is uniformly randomly chosen to be either 2 or 3 digits long.

obs.	source	target
0	zero two	02
1	four eight	48
2	eight four four	844

Table 3.1.3: Examples from the Synthetic Digits dataset generator.

3.2 Experiment Setup

Continues evaluation

Model evaluation measures, such as the misclassification rate, and the BLEU score, are continuously calculated during training using a randomly sampled mini-batch, from the test dataset. The predictions are calculated using a vanilla greedy algorithm, this corresponds to having a beam size of 1.

The evaluations are smoothed using an exponential moving average, this removes most of the noise associated with training neural networks with mini-batches. Because the moving average is done over different samples from the test and training dataset, the moving average will likely be close to the true average. However, this is not statistically guaranteed as the model differs in each iteration, thus stationarity is not guaranteed.

After training prediction

The predictions made after training are done using a BeamSearch algorithm with a beam size of 10. Each prediction is the most likely sequence found during the BeamSearch that contains an `<eos>` symbol.

Multiple GPU parallelization

A mini-batch size of 16 observations per GPU is used. In some experiments, multiple GPUs were used in parallel to speed up convergence, in these experiments the mini-batch was split evenly to each GPU. The gradients were then calculated independently on each GPU, the average gradients over each GPU-mini-batch were then calculated on the CPU and used to update the model. After that, the new parameters are redistributed to the GPUs. This method of GPU parallelization is called *synchronized weight update*, this is not the fastest approach but is the most stable approach. The other class of approaches are called *asynchronous weight update* [26], besides being less stable and more difficult to implement, DTU doesn't have the resources to justify such an approach.

Bucket batches

If a long sequence and a short sequence are used in the same mini-batch, the short sequence will be padded with a lot of zeros. Such padding is wasteful as the loss is masked. To prevent obsessive padding, each sequence pair is partitioned into a set of buckets, each bucket contains sequences of approximately the same length.

Each bucket is defined as a sequence-length interval, where the length of a sequence pair is defined as the max length of the two sequences. Only after the buckets are created are the observations assigned to a bucket.

The partition algorithm first constructs a histogram using the length of each sequence pair. It then greedily partitions the histogram, ensuring that the number of observations in each bucket is more than $2 \cdot \text{batch-size}$, this is to ensure some randomness. It also ensures that the length interval for each bucket is at least 10 characters, this to prevent an unnecessary amount of buckets.

Algorithm 4 Bucket partition algorithm, outputs length intervals of buckets.

function BUCKETPARTITIONER($\mathcal{D}, minsize, minwidth$)
  $interval_{left} \leftarrow 0$
  $interval_{size} \leftarrow 0$
  $buckets \leftarrow$ STACK()
  for $(interval_{right}, size)$ in HISTOGRAM($\mathcal{D}$) **do**
    if $length - interval_{left} < minwidth$ **then** ▷ Bucket is to short
      $interval_{size} \leftarrow interval_{size} + size$
    else if $interval_{size} < minsize$ **then** ▷ Bucket is to short
      $interval_{size} \leftarrow interval_{size} + size$
    else
      PUSH($buckets, [interval_{left}, interval_{right}]$) ▷ Accept bucket
      $interval_{left} \leftarrow interval_{right} + 1$ ▷ Prepare next bucket
      $interval_{size} \leftarrow 0$
  ▷ Extend last bucket to contain the remaining dataset.
  return *buckets*

Software and Hardware

The thesis code is available at: `https://github.com/AndreasMadsen/master-thesis`

TensorFlow 1.1 [27] and Python 3.6 were used to build, train, and evaluate the models. A TensorFlow abstraction layer called `sugartensor` (`https://github.com/buriburisuri/sugartensor`) developed by KakaoBrain was also used. During this thesis I have made 6 separate contributions to the `sugartensor` project. The plots are generated using ggplot2 [28] in R.

TensorFlow was compiled for the specific CPU architecture and for CUDA Compute Capabilities 6.1. CUDA 8.0 and CuDNN 5.1 was used.

The DTU HPC system was used for computation. They have two Titan GPU machines, each machine have:

- 4 Nvidia Titan X (Pascal) GPUs.
- 1 Intel Xeon CPU E5-2640 v4 CPU, with 10 cores.
- 128 GB RAM.

3.3 ByteNet

3.3.1 Validating Implementation

Before using the model on the Europarl v7 dataset for solving the WMT Translation Task problem, the model is validated using some simpler datasets.

Learning Synthetic Digits Problem

The Synthetic Digits dataset is used for validating the generalization properties of the ByteNet implementation.

The internal dimensionality is set to 20, this is 20 units in the encoder and 40 units in the decoder, the latter is because the encoding is concatenated with the target embedding. It is unlikely that higher dimensionality is required, given that there are only 10 possible output characters. In fact, 20 might be much higher than necessary, but this may be a good thing in terms of validation as it provides an opportunity to ensure that the model doesn't overfit. Using a dimensionality of 20, the network has just counting the convolution and dense layers 73500 weights. The training dataset contains only 128 observations, thus there is a huge potential for overfitting. The test dataset also contains 128 observations.

The ByteNet model is optimized using the Adam optimizer with a learning rate of 0.001 and a mini-batch size of 16 running on 1 GPU. Because the BLEU score is not meaningful for the Synthetic Digits problem, where no words exist in the target, the misclassification rate is calculated instead.

obs.	source	target	prediction
0	one zero four	104	104
1	one five six	156	150
2	five five nine	559	559
3	one six	16	68
4	two three four	234	235
5	five three	53	53

Table 3.3.1: Source, target, and prediction on examples from the test dataset.

Figure 3.3.1 shows reasonable convergence, only the test error shows jitter during training and there is very little overfitting if any. In the end, the ByteNet model completely learned the training dataset.

The jitter is likely not because of poor optimization, but rather because the errors are only calculated on a randomly sampled subset of the test dataset. For the different samples, the test error is thus different.

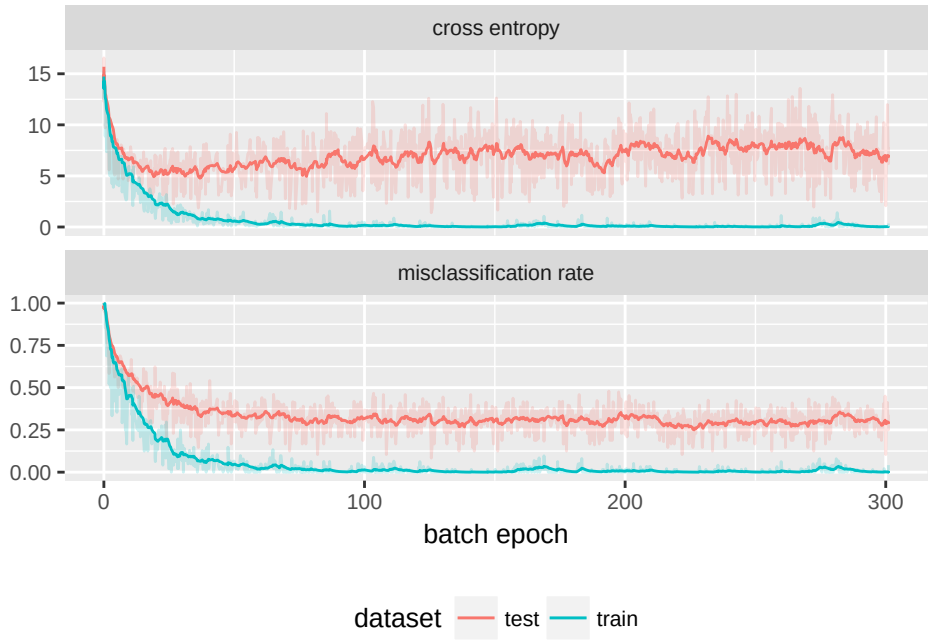


Figure 3.3.1: Shows misclassification rate and the cross entropy loss. For comparison an attention model has a misclassification rate of 0.51. The exponential moving average uses a forget factor of 0.1.

The lack of overfitting fits well with what the original ByteNet article also observed, in their translation model no regularization or early stopping was used, which is what one would typically use to prevent overfitting [10].

In table 3.3.1 the predictions are about what one would expect. For the most part, the translations are good, in particular in the beginning of the output sequence, while the later digit predictions show some error. This result is reasonable, as there is more data for the first two digits and because the input words have different lengths. The alignment between input and output characters becomes more uncertain as the read sequences get longer. Of cause, the model would ideally learn the word separation completely and understand that there is a direct relation between word and digit, but learning this relation is likely difficult given both the many parameters and the small dataset.

Memorizing WMT NewsTest

Sometimes a model works well when it has few weights and low dimensionality but breaks for higher dimensionality because of vanishing or exploding gradient issues. To validate that this is not an issue a good test is to see if the model can memorize a small dataset, but where the model complexity is kept high. For memorization one just expects the training loss to become very small, the test error is not important.

The WMT 2014 NewsTest dataset for German to English translation was used for training, this contains 3003 observations. The WMT 2015 NewsTest dataset was used for testing, this contains 1596 observations. The internal dimensionality is set to 400, (800 in the decoder because of concatenation).

The model ran 120 epochs over the training dataset, with a mini batch size of $4 \cdot 16 = 64$, running on 4 GPUs in parallel using synchronized updates with the Adam optimizer and a learning rate of 0.0003.

Figure 3.3.2: Shows BLEU score and cross entropy loss for the German to English WMT NewsTest dataset. Both training and test measures are calculated on a randomly sampled mini-batch from each dataset. The exponential moving average uses a forget factor of 0.3.

0	source	Polizei von Karratha verhaftet 20-Jährigen nach schneller Motorradjagd
	target	Karratha police arrest 20-year-old after high speed motorcycle chase
	translation	In must in the ugains of the with of Chanamby a leadn-Minor ald the ace the town of the construction of the compunity the government.
	BLEU	0.00
1	source	Das Motorrad wurde sichergestellt und für drei Monate beschlagnahmt.
	target	The motorcycle was seized and impounded for three months.
	translation	The New York works with smaries for twe to the come to cover earling scace us deliving the construction work.
	BLEU	0.00

Table 3.3.2: Source, target and prediction on the test dataset.

Ensuring that the optimization converges was the primary purpose of this experiment. Figure 3.3.2 shows that the parameter optimization does like in the synthetic digits problem, appear to converge just fine. Initially, there is some jitter in the cross entropy training loss, but this subsides after awhile.

In the beginning, the jitter frequency is higher this is just because TensorFlow samples values at fixed time intervals. Initially, there are some allocations and data transfers that causes the model to run slower, thus as a side effect TensorFlow samples more frequently in this period.

As said the test loss is not very interesting, as there isn't enough training data to expect the model to produce meaningful results. The cross entropy on the test dataset does also show an increase after the initial decrease, which indicates some overfitting. Again, this is to be expected given the small training dataset.

More interesting is the correlation between the training cross entropy and the training BLEU score. Initially, the cross entropy decreases a lot, but the BLEU score stays at 0. It is first when the cross entropy nears 0, that the BLEU score begins to improve. This observation is quite important as it indicates that cross entropy isn't a perfect loss measure for the translation problem. This is because the cross entropy can be quite low if it just gets the majority of the characters almost correct, but the position must be correct. The BLEU score, on the other hand, demands that the words matches exactly, but is looser regarding the position. For example, in table 3.3.3 the target is "large" but the prediction is "lerge". This is very close in terms of cross entropy but is completely wrong in terms of the BLEU score, since the words don't match. Nevertheless, the cross entropy loss is useful because its gradient in

0	source	Demgegenüber unterrichten australische Schulen durchschnittlich 143 Stunden jährlich und Schüler in Singapur erhalten 138 Stunden.
	target	By comparison, Australian schools provide an average of 143 hours a year and pupils do around 138 hours in Singapore.
	translation	By comparison, Australian schools provide an average of 143 hours a year and pupils do around 138 hours in Singapore.
	BLEU	100.00
1	source	Diese Umlage wird jedes Jahr im Oktober von den vier Betreibern der der großen Stromtrassen neu festgesetzt.
	target	This levy will be reset by the four operators of the large power grids, in October of each year.
	translation	This levy will be reset by the four operators of the lerge power grids, in October of each year.
	BLEU	87.23

Table 3.3.3: Source, target and prediction on the training dataset.

combination with softmax is easily computable, and the BLEU score does become very high in the end, even though the correlation between cross entropy and the BLEU score is weak.

The training BLEU score is actually extremely good, it shows almost a perfect translation. Typically translation models only show a BLEU score between 15 and 25 on a test set. This is not necessarily because the translation is wrong but because there are many different ways of translating a sentence correctly. The BLEU score does actually support multiple target sequences, but this is rarely provided in the test dataset because of the labor demands for creating multiple translations.

The only way the model can be this good is by primarily memorizing the output given the input. It is unlikely there is much understanding of the latent semantic meaning in the text. That being said, the predictions in table 3.3.2 shows that the model understands some grammar, such as which words comes before nouns. That being said, the second prediction example in table 3.3.2 contains a single quotation mark, which is not grammatically meaningful.

Some of the above-mentioned issues could perhaps be improved by not using the target sequence in the decoder during training step. Instead, the predicted sequence could be used in the decoder input, just like in the inference model. This could solve issues like “large” becoming ”lerge”. The disadvantage of doing this is that training would be slower, as the decoder can’t be parallelized over the sequence. Some research suggests that one should use a hybrid model, where the training loss is composed of both an assisted part and a non-assisted part, in the latter case the target sequence

isn't used [22].

Another observation is that the predicted sequences are in general a bit longer than the target sequences. This is likely because of how the sequence loss is constructed, this only measures the error where the target is explicitly known. Thus there is little penalty for predicting longer sequences than the target sequence. However, there is some penalty, because the predicted `<eos>` symbol should match the target `<eos>` symbol. More penalty could be added by padding the target sequence sufficiently with a special `<null>` symbol, and then also measure the error on that part of the sequence. However, this introduces an issue where the translation model can just predict "`<null><null><null><null>...`" and this will in terms of cross entropy be a good match since the majority of the sequence matches. This is an easily obtainable solution for the optimizer, it can thus be a local minimum that is difficult to escape.

3.3.2 WMT Translation Task

With the ByteNet model reasonably validated in terms of generalization on the synthetic digits problem, and convergence when memorizing the WMT NewsTest dataset, the ByteNet model is used on the Europarl v7 dataset for German to English translation.

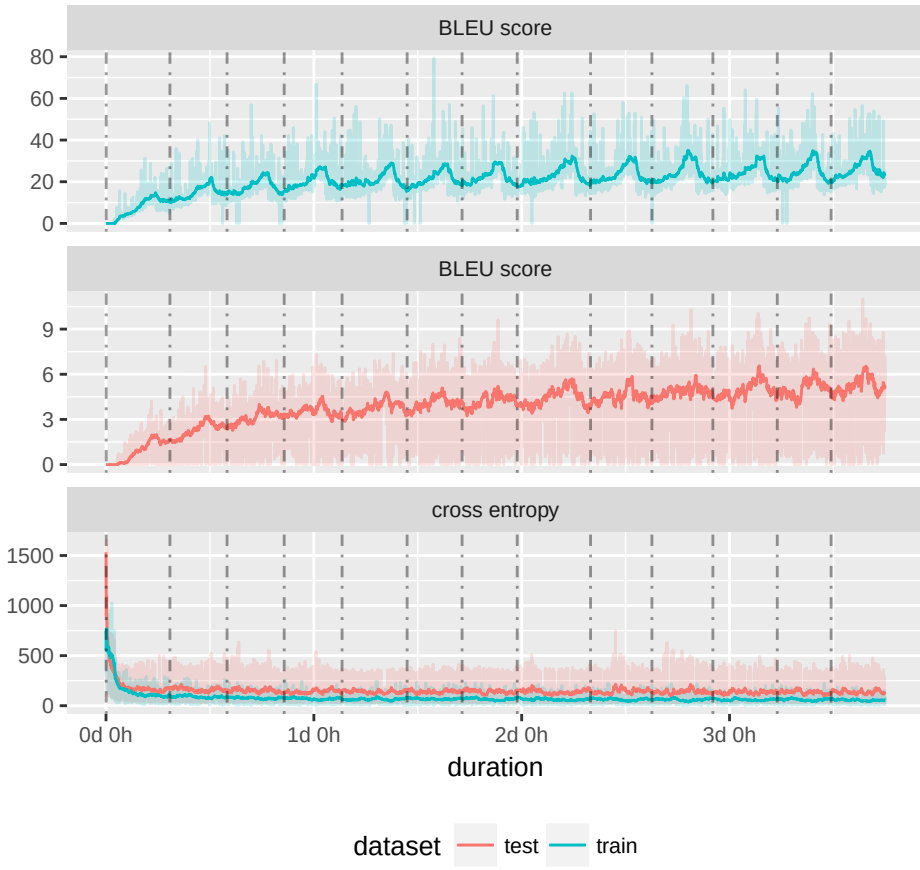


Figure 3.3.3: Shows BLEU score and cross entropy loss for ByteNet, trained on Europarl v7 and tested on WMT NewsTest 2015. Both training and test measures are calculated on a randomly sampled mini-batch from each dataset. The exponential smoothing uses a forget factor of 0.05. The vertical lines separates each batch epoch.

The ByteNet model has an internal dimensionality of 400, just like the model used for training on the WMT NewsTest dataset. The Adam optimizer with a learning rate of 0.0003 is used for training. This used a training mini-batch size of $4 \cdot 16 = 64$ obser-

vations and the model was trained on 4 GPUs in parallel with synchronized updates. The Europarl v7 dataset is used for training and the training ran 13 epochs over the Europarl v7 dataset. The WMT NewsTest 2015 dataset is used for testing, the continues testing was done on randomly sampled mini-batches with 128 observations each.

From figure 3.3.3 it's seen that the BLEU score on the test dataset is approximately 5.5. However, because that BLEU score is calculated on a random subset of the dataset it is not completely accurate. The actual BLEU score calculated on the entire WMT NewsTest dataset after training is 7.44. The translations are shown in table 3.3.5 and table 3.3.6.

Figure 3.3.3 also shows some oscillation that is correlated with the epochs. After further investigation it turns out that there are bad translations in the dataset, some examples can be seen in table 3.3.4. Such observations would misdirect the optimization in each epoch and thus cause oscillation. However, even after removing all observations with either a source or target sequence less than 25 characters, the oscillation still exists. Given that the oscillation is correlated with the epochs, it is most likely the dataset that still contains poor observations. However, finding these among 2 million sentence pairs is rather difficult, a possible solution could be to increase the bucket size, such that there is a higher probability for the poor translations to be mixed with the good translations in each mini-batch.

0	source	Herr Kommissar!
	target	
2	source	
	target	There are, therefore, two points to which I would like to draw the Commission' s attention.
3	source	
	target	This is something about which European small and medium-sized businesses, in particular, tend to complain.

Table 3.3.4: Examples of bad translations in the Europarl v7 dataset.

Looking at the actual translations, those with a high BLEU score in table 3.3.5 are as expected quite good. Much more interesting is to look at the bad translations in table 3.3.6.

Translation `0` is actually a very good translation, but the different word ordering means that it gets a BLEU score of zero. This is an example of the BLEU score not always being a good measurement of text similarity.

Translation `1` is on the other hand quite bad, except for the "He is accused" part it is very wrong. It is apparent that the translation model detects places by its initial capital letter and attempts to translate these but the translations are rarely

0	source	Die formelle Anerkennung der Arbeitsrechte als Menschenrechte - und die Erweiterung des Schutzes der Bürgerrechte als Schutz gegen Diskriminierung bei der Schaffung von Arbeitnehmervertretungen - ist längst überfällig.
	target	The formal recognition of labor rights as human rights - and the extension of civil rights protections to prevent discrimination against labor organizing - is long overdue.
	translation	The formal recognition of human rights as human rights - and the extension of the protection of civil liberties as a protection against discrimination against employment organizations - is long overdue.
	BLEU	45.97
1	source	Aber es ist mit Sicherheit keine radikale Initiative - jedenfalls nicht nach amerikanischen Standards.
	target	But it in certainly not a radical initiative - at least by American standards.
	translation	But it is certainly not a radical initiative, at least for American standards.
	BLEU	39.13
2	source	Das Militär spielt in Pakistan eine wichtige Rolle und hat bereits häufiger geputscht.
	target	The military plays an important role in Pakistan and has taken power by force several times in the past.
	translation	Military players play an important role in Pakistan and has already been developing more and more.
	BLEU	30.18

Table 3.3.5: Cherry-picked translations from WMT NewsTest with high BLEU score.

correct. This is quite interesting since translating names should often be easy since it is just the identity function. However, such a mapping requires the model to switch between a semantic understanding and the identity function. This behavior may be difficult for the model to learn as it will initially either learn semantic understanding or the identity function. Once that is partially learned the model will use its entire parameter space for that purpose. Compressing this parameter space and allowing for a new mode of operation is perhaps not a likely gradient direction in the optimization, as more would initially be lost by the compression than what is gained by the extra mode. That being said character-based models have been shown to be able to learn this behavior [29].

0	source	Die Premierminister Indiens und Japans trafen sich in Tokio.
	target	India and Japan prime ministers meet in Tokyo
	translation	The Prime Minister of India and Japan worked in Tokyo.
	BLEU	0.00
1	source	Er wird beschuldigt, am 7. Juni 2013 eine Frau im Scotland's Hotel in Pitlochry in Perthshire vergewaltigt zu haben.
	target	He is alleged to have raped a woman at the Scotland's Hotel in Pitlochry in Perthshire on June 7, 2013.
	translation	He is accused of being a Member of the European Parliament in Scotland in Petersberg in Peru.
	BLEU	0.00
2	source	Angelina Jolie und ihr Bruder James haben eine Videohommage für ihre Mutter online gestellt, die 2007 an Eierstockkrebs verstarb.
	target	Angelina Jolie and her brother James have posted a video tribute to their late mother who died of Ovarian cancer in 2007.
	translation	Angela John and her village of James have created violence for their mother-tongue, which involves emergency catastrophe.
	BLEU	0.00
3	source	"Diese Krankheit wird am besten von gynäkologischen Onkologen behandelt, und diese sind meistens in größeren Städten zu finden," sagte sie.
	target	"This disease is best treated by gynaecological oncology surgeons and they're mostly based in major cities," she said.
	translation	'This disease is best achieved by organic chocolate industries, and these are indeed more than "more cities'.
	BLEU	0.00

Table 3.3.6: Cherry-picked translations from WMT NewsTest with zero in BLEU score.

Translation `1`, `2`, and `3` are all interesting because while the translations may be bad the grammatical part is mostly fine. This indicates that the model understands basic language constructs. An example is that "He is accused of being a Member of the European Parliament" makes sense even though the translation is very wrong.

3.3.3 Performance profiling

The ByteNet model is rather slow. For ByteNet to reach a BLEU score of 23 as Google achieved in the original paper, it will have to train on a much bigger dataset for much a longer time.

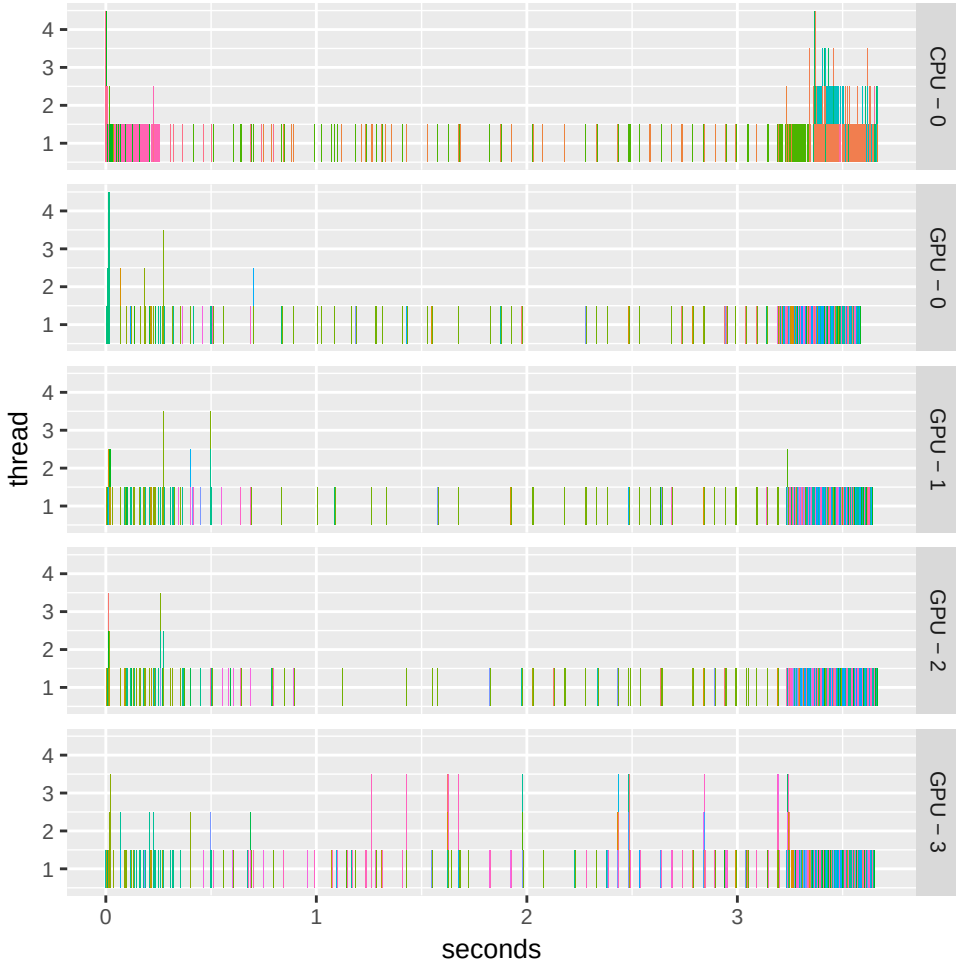


Figure 3.3.4: Shows time spent on each operation, when the operation was executed, and on what GPU/CPU it was executed. The color coding indicates the operation type, there are more than a 100 different operation types, most are specific to TensorFlow, thus the legend is not included.

To understand why the ByteNet model is so slow, TensorFlow was profiled while

executing the computational graph. The setup is identical to the "Memorizing WMT NewsTest" setup running on 4 GPUs, and the profiling was done when evaluating the 4500th mini-batch. The 4500th mini-batch was chosen because the performance has reached its peak and stabilized.

From figure 3.3.4 it's seen that most of the time is not spent computing, but rather waiting for data to be transferred or just waiting for the TensorFlow service to queue a new task.

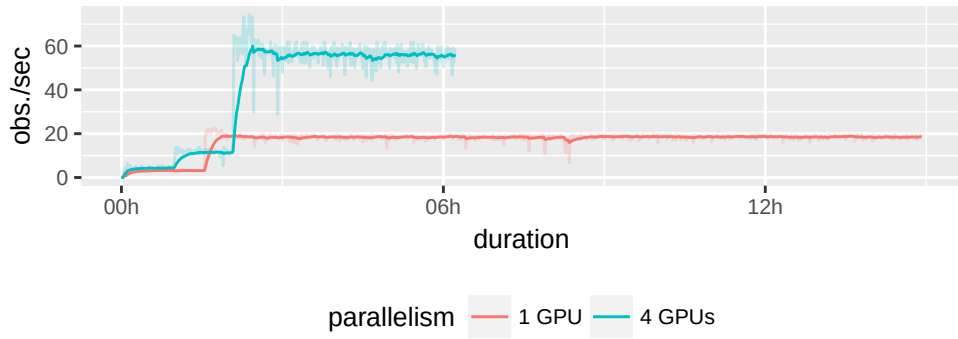


Figure 3.3.5: Comparing observations per second, depending on the number of GPUs used.

By comparing the speed of how fast the ByteNet implementation processes observations, depending on the number of GPUs used, one gets that about 37% of the time is spent waiting for data transference in the 4 GPUs case. This calculation does, in particular, make sense when comparing with 1 GPU, since a 1 GPU setup will not require data transfer of any weights. This is because the gradients and weights don't need to be synchronized on the CPU, but can be kept on the GPU where they are calculated.

The data transfer does not explain all the waiting time. Likely this particular profiling is an extreme case. TensorFlow transfers the dataset in chunks in preparation for future mini-batches. If TensorFlow is transferring the dataset while profiling in this exact moment, that will cause extra waiting time.

By processing the profiling dump file, such that the waiting time is removed from the data, one can see that time is primarily spent in the layers that contain normalization and activation (*pre-normalization*, *conv-dilated*, *reduce-dim*).

It is likely that the data transfer part could be optimized, but in general, this isn't easy and there are practical limitations to how much it can be improved. It is much more likely that the waiting time regarding the TensorFlow service could be improved.

TensorFlow works with a computational graph. Each atomic operation, like an element-wise addition or a matrix multiplication, is a node in this graph and the edges describe the dependencies. The TensorFlow service will watch the graph and execute any node (atomic operation) when all its dependencies are satisfied, it will even execute multiple nodes in parallel if possible. This process repeats until all nodes have been computed and the end result is obtained.

Using a computational graph is a good strategy, but the current TensorFlow implementation of it is very naive. TensorFlow uses no-op and identity operations, which only exists because of TensorFlow semantics, but don't need to be executed. However, in the current state of TensorFlow, it just executes all nodes naively. There are also numerous of atomic operations that could be fused into one atomic operation. An example is batch normalization that involves quite a few element-wise operations, all these are executed separately but could be combined into a single atomic operator. All these atomic operators are what causes the waiting time, the TensorFlow service needs to walk the computational graph and more importantly just launching GPU kernels also introduces wasted time.

The TensorFlow team is aware of the current limitations and are in the process of solving these issues, by using a Just In Time compiler that can automatically fuse many of these operations. But so far this is in an experimental state and performs very poorly when running on multiple GPUs [30].

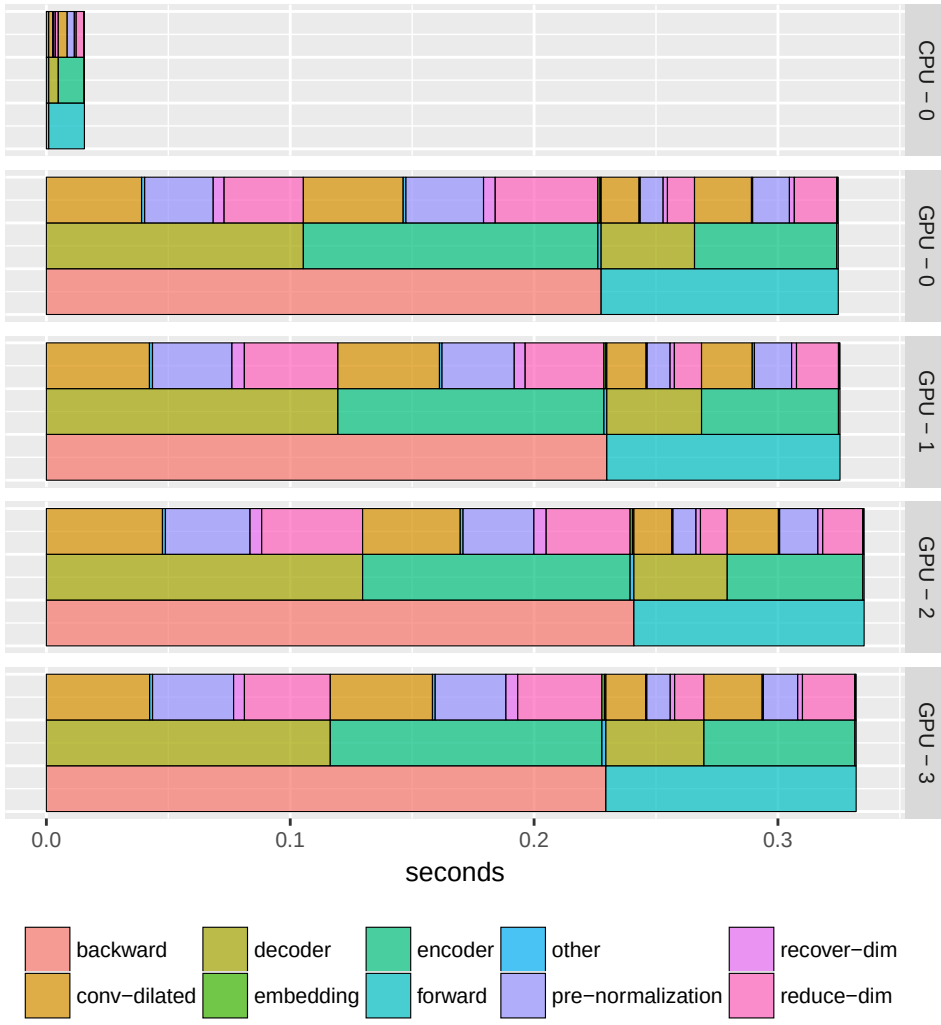


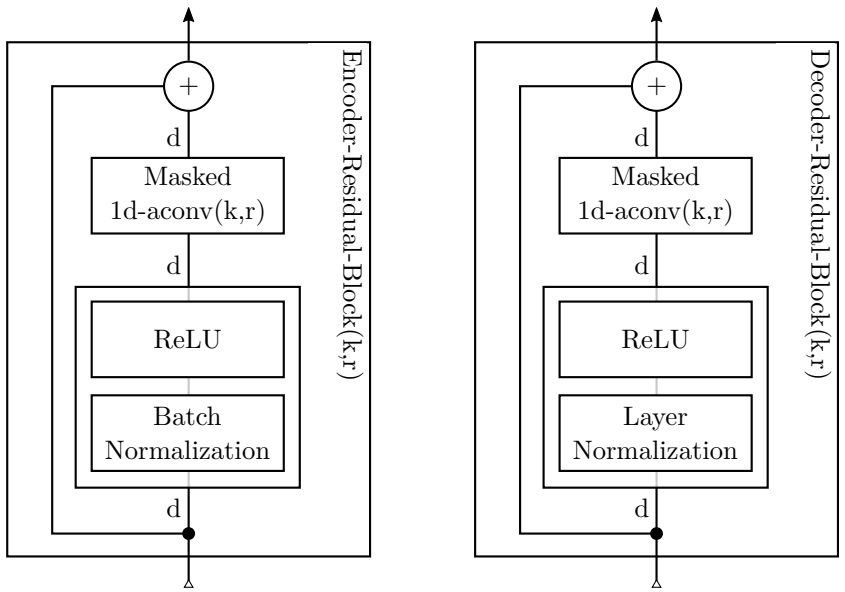
Figure 3.3.6: Shows time spent executing each part of the ByteNet model, this excludes the waiting time. Each part exists in a hierarchy, which is visualized as levels. Bottom level is the backward and forward pass. Second level is the encoder and decoder. Last level primarily splits the ByteNet Residual Blocks.

3.4 Simplified ByteNet

The implementation of the ByteNet model spends 37% of its time waiting for data transfer between the GPU and CPU, and even more time waiting because of the TensorFlow service. This is because of the many of weights and operations that are used in the ByteNet model.

To solve this issue, a simplified version of the ByteNet model is used. This model uses fewer weights and fewer operations than ByteNet, but remains true to the principals behind ByteNet. Those principals are: running in linear time, parallelize over both sequence and observations, and have a resolution persistent encoding. The simplified version also maintains the bottleneck of 200 dimensions that ByteNet has in its dilated-convolution layer.

The idea is simple, if the initial embedding dimensionality is set to 200, then the compression and decompression layers that exist before and after the dilated convolution are not needed. Of cause these layers add non-linearities and weights to the model, thus one should not expect the model to perform equally well. However, the model should predict almost as well as the ByteNet model, while being significantly faster.



(a) Residual Block used in encoder. **(b)** Residual Block used in decoder.

Figure 3.4.1: The residual blocks used in the simplified ByteNet model. The blocks are denoted by Encoder-Residual-Block(k, r) and Decoder-Residual-Block(k, r), where k is the kernel width and r is the dilation rate.

In terms of weights, the simplified ByteNet model has approximately $^5/_9$ times fewer weights than the ByteNet model. The simplified ByteNet model also has approximately one-third of the operations as the ByteNet model. Based on these parameters one should expect less transfer time and less time spent waiting for the TensorFlow service.

The simplified ByteNet model was validated identically to the how the ByteNet model was validated. The results (appendix D.2) were very similar with some minor differences. Memorizing WMT NewsTest took fewer iterations, this is likely because there are fewer parameters. The simplified ByteNet model learned the synthetic digits problem better, with a misclassification error of 0.25. The improved misclassification error is likely because the simplified ByteNet model has fewer parameters and non-linear transformations, which makes it overfit less.

3.4.1 Performance Profiling

To compare the computational performance of the simplified ByteNet model with the normal ByteNet model, the performance experiment was repeated using the simplified ByteNet model. This experiment learns the WMT NewsTest dataset over 300 epochs. Both a 1 GPU and a 4 GPU setup was used in the experiments.

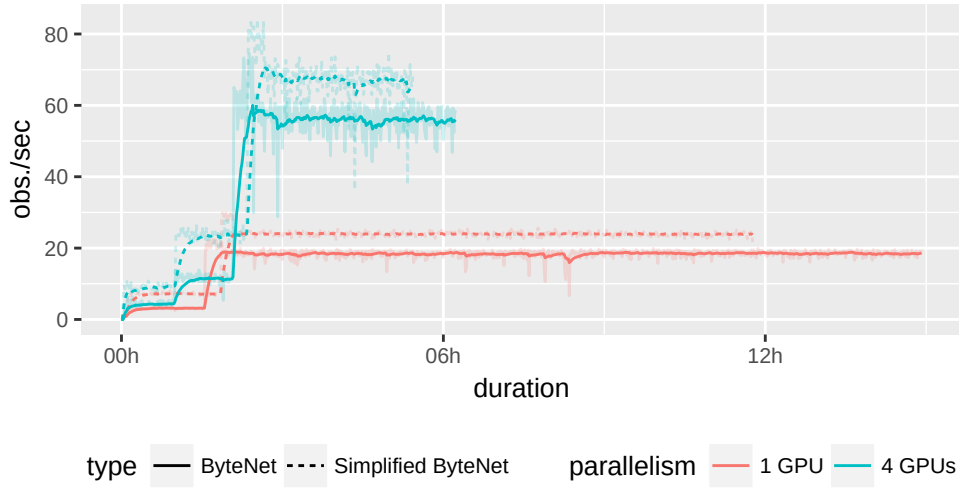


Figure 3.4.2: Comparing observations per second, depending on the number of GPUs used.

Measuring the total time spent, and the processed observations per second (figure 3.4.2) reveals that the simplified ByteNet model is faster. On 1 GPU the simplified

ByteNet model is about 20% faster.

On 1 GPU there is no data transfer of weights, thus it is only the TensorFlow service and the computation that takes time. Reducing the number of operations to $^1/_3$ should thus result in a $^2/_3$ computational performance gain. This is of course an approximation as there is still the embedding layer, the concatenation of encoding and decoding, and the final output layer, thus $^2/_3$ the operations is an upper bound. However, 20% is still far from what one would expect.

Comparing the time spent in the 1 GPU experiment and the 4 GPU experiment, and because very little data transfer happens when using 1 GPU. The data shows that approximately 40% time is spent transferring data. This is approximately the same as the 37% in the normal ByteNet model. The number of weights is reduced by $^5/_9$, thus one would expect the transfer time to be reduced by this order, but this is not the case.

Finally, the TensorFlow profiler can be used to investigate what takes time. When processed the results are similar to those from the normal ByteNet model, see figure 3.4.3.

Figure 3.4.3 shows that much of the time is spent in the *pre-normalization layer*. The *pre-normalization layer* is the layer in the residual block that contains the first normalization and ReLU activation. This validates the idea that it is the number of operations and not the operations themselves that cost in terms of time. The dilated convolution (called *conv-dilated*) is a much more complex operation than normalization or ReLU, but TensorFlow implements it using just a few operations, thus it doesn't consume that much time. As mentioned earlier the TensorFlow team is aware of this performance issue. TensorFlow is planning to solve this issue by automatically compiling GPGPU (CUDA or OpenCL) kernels that combine multiple operations into one. However, this is still very experimental and doesn't yet provide a performance benefit when using multiple GPUs [30].

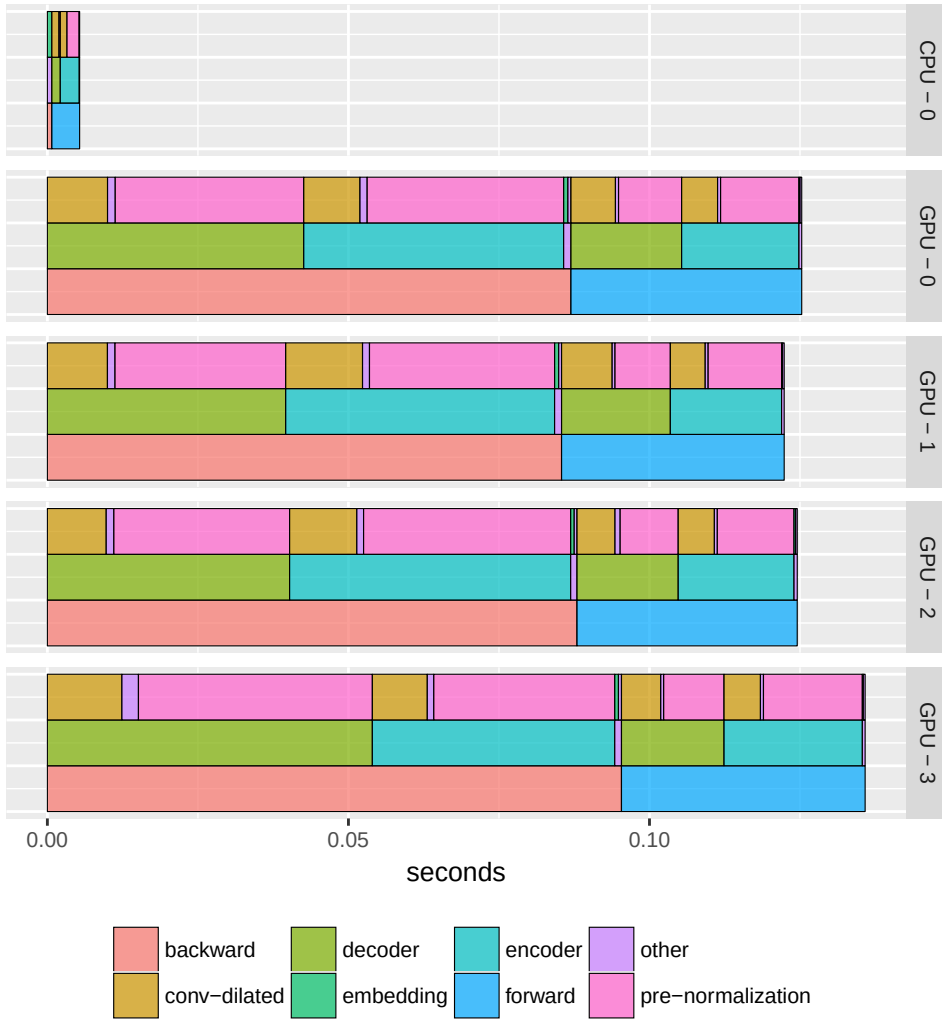


Figure 3.4.3: Shows time spent executing each part of the ByteNet model, this excludes the waiting time. Each part exists in a hierarchy, which is visualized as levels. Bottom level is the backward and forward pass. Second level is the encoder and decoder. Last level primarily splits the simplified ByteNet Residual Blocks.

3.4.2 WMT Translation Task

The Europarl v7 dataset is used to train the simplified ByteNet model. The setup is identical to that previously used to train the normal ByteNet model. The internal dimensionality is 400, the Adam optimizer is used with a learning rate of 0.0003, and the mini-batch size is $4 \cdot 16 = 64$ observations, 4 GPUs with synchronized updates was used, and the model ran 13 epochs over the Europarl v7 dataset.

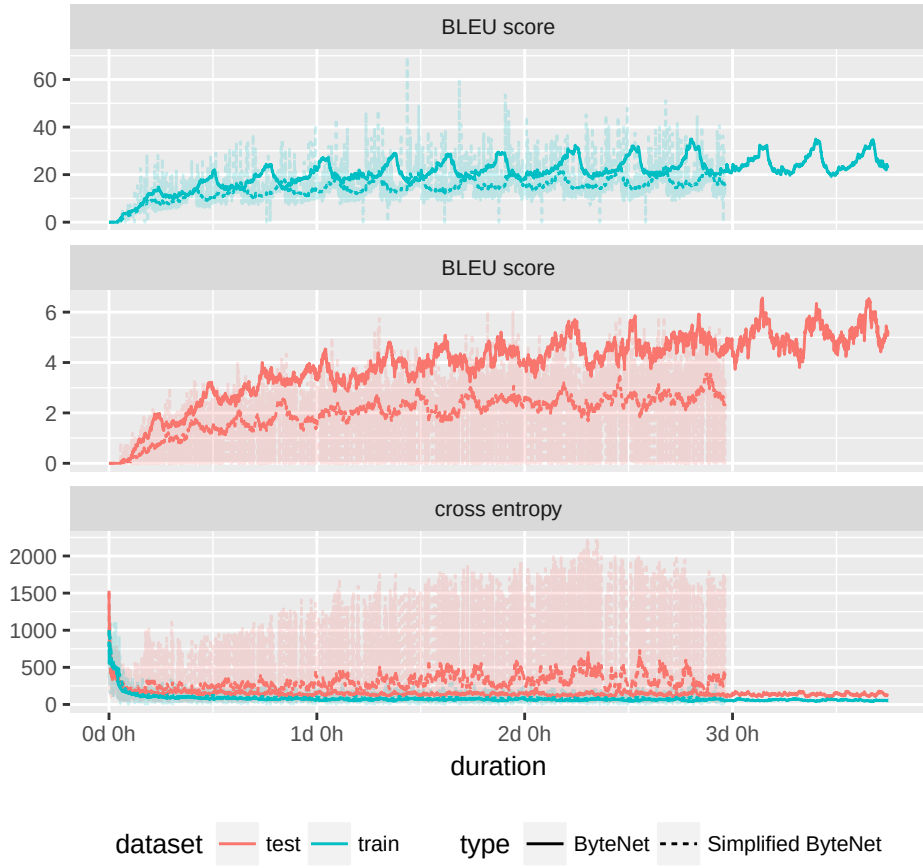


Figure 3.4.4: Shows BLEU score and cross entropy loss for the Simplified ByteNet model, trained on Europarl v7 and tested on WMT NewsTest 2015. Both training and test measures are calculated on a randomly sampled mini-batch from each dataset. The exponential smoothing used a forget factor of 0.05. The raw data for the normal ByteNet model is not shown.

The simplified ByteNet model uses almost 24 hours less training time. However, it

also learns at a slower rate, thus in terms of BLUE score given the time spent learning, the normal ByteNet model is actually much faster.

An interesting observation is that the simplified ByteNet model overfits a lot more than the normal ByteNet model. This contradicts most intuition about overfitting, which usually is that more weights, layers, and non-linearities there is, the easier it is to make the model overfit. The simplified ByteNet model has less of all three factors, thus there is no reason to expect the simplified ByteNet model to overfit more.

It is not clear why this overfitting happens, but it is very likely a contributing factor to why the test BLEU score isn't better. However, this can not be the entire explanation, as also the training BLEU score is worse. If anything, overfitting should cause a higher training BLEU score, as regularization typically increases the training loss in order to decrease the test loss.

Possible regularization methods are L1, L2, and dropout. Particularly, L1 regularization could be interesting in the first 5 *residual blocks*, as these layers likely perform some characters-to-word encoding. Such an encoding should be very sparse, L1 regularization would enforce such a sparsity.

The translations as shown in table 3.4.1 are rather bad, they have little connection to the source sequence and there is a tendency for the sequence "Mr *insert name*" to appear without context. The poor translations can likely be attributed to the severe overfitting.

Calculating the BLEU score after training on the entire WMT NewsTest dataset, yields a BLEU score of 0.55. This matches the poor translations a lot better than the sampled BLEU score of approximately 3.

0	source	Die formelle Anerkennung der Arbeitsrechte als Menschenrechte - und die Erweiterung des Schutzes der Bürgerrechte als Schutz gegen Diskriminierung bei der Schaffung von Arbeitnehmervertretungen - ist längst überfällig.
	target	The formal recognition of labor rights as human rights - and the extension of civil rights protections to prevent discrimination against labor organizing - is long overdue.
	translation	However, the recommendation of jobs and generations and proposes the promotion of gross regions of the world growth to prodictively rejecting the rejection of the representative of the representative increase' .
	BLEU	0.00
1	source	Die Premierminister Indiens und Japans trafen sich in Tokio.
	target	India and Japan prime ministers meet in Tokyo
	translation	Mr Pieper and the independence of Singapore in London.
	BLEU	0.00
2	source	Er wird beschuldigt, am 7. Juni 2013 eine Frau im Scotland's Hotel in Pitlochry in Perthshire vergewaltigt zu haben.
	target	He is alleged to have raped a woman at the Scotland's Hotel in Pitlochry in Perthshire on June 7, 2013.
	translation	Madam President, I should like to contribute to Mr President Mr President in his report.
	BLEU	0.00

Table 3.4.1: Cherry-picked translations from WMT NewsTest.

3.5 SELU ByteNet

The simplified ByteNet experiments indicate that the compression and decompression layers are necessary and the normalization layers are very expensive. From these observations, it makes sense to analyze whether or not the normalization layers are necessary for the network to converge.

The experiment in figure 3.5.1 is the "Memorizing WMT NewsTest" experiment on the ByteNet model without normalization layers. The experiment ran for 300 epochs.

Figure 3.5.1: Shows BLEU score and cross entropy loss for the German to English WMT NewsTest dataset using the ByteNet model without any normalization layers. The exponential moving average used a forget factor of 0.1.

Figure 3.5.1 shows that the non-normalized ByteNet model never completely memorizes the training dataset as it should. This is likely due to a vanishing gradient issue, that causes it to converge very slowly. It is possible that it could still learn actual translation when trained on the Europarl v7 dataset, however, it is not very likely.

Recently a new paper showed that it is possible to create a "self-normalizing neural network". This means that normalization isn't done explicitly by a normalization

layer, but instead, the network is created such that the parameters will converge to weights that ensure normalized internal values.

The "self-normalizing neural network" was achieved by using a different activation function and weight initialization. Primarily it is the activation function that is responsible for making the network *self-normalizing*. The initialization is just to ensure a reasonable starting point, as the "self-normalizing neural network" is only able to "self-normalize" given a reasonable starting variance and mean. [31].

The activation function is a variation on the exponential-linear-unit (ELU), which is similar to the ReLU activation function. The difference from ELU is that two constants (λ, α) are added:

$$\text{SELU}(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(\exp(x) - 1) & x \leq 0 \end{cases}, \quad \text{where:} \quad \begin{array}{l} \alpha = 1.6732632423 \\ \lambda = 1.0507009873 \end{array} \tag{3.5.1}$$

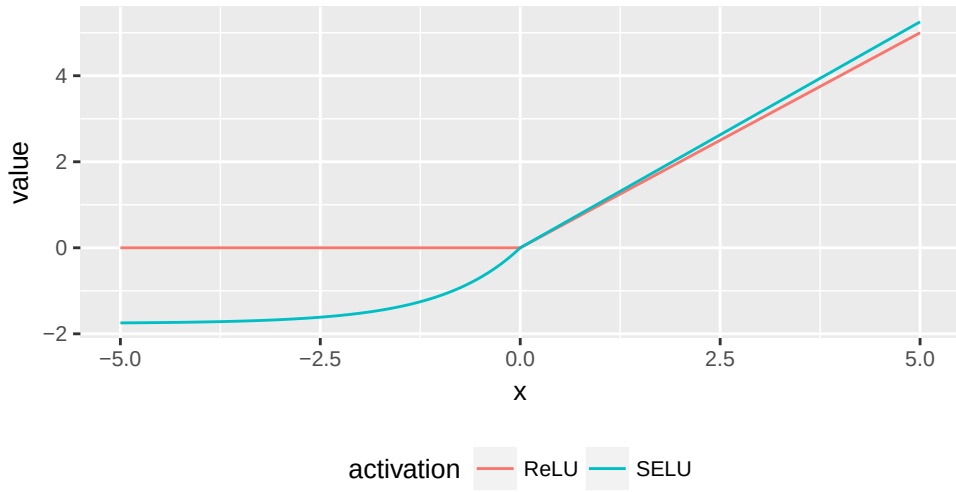


Figure 3.5.2: Shows $\text{ReLU}(x)$ and $\text{SELU}(x)$ in the range $x \in [-5, 5]$.

The parameters α and β, are derived from the assumption that the sum z_{h_ℓ} approximates a normal distribution. The SELU paper argues that these assumptions are true because of the Lyapunov central limit theorem (CLT). For the Lyapunov variant of CLT to be true, the stochastic variables $z_{h_\ell}, \forall h_\ell \in [1, H_\ell]$ must be independent and the z_{h_ℓ} sum must be over many $z_{h_{\ell-1}}$ stochastic variables. The latter is satisfied because the sum is over $H_{\ell-1}k = 2000$ values. However, the former assumption about independence is obviously not true. The SELU paper argues that this is only a weak assumption, as stochastic variables with weak dependencies will still converge to a

normal distribution [31, 32]. Even so, whether or not the weak dependency assumption is satisfied is of course problem and model dependent. The SELU paper shows a big improvement on the MNIST and CIFAR10 dataset using a CNN classifier, which should be considered to be a dataset and model with reasonably high dependencies [31].

The initialization should then be done such that $\mathbb{E}[z_{h_\ell}] = 0$ and $\text{Car}[z_{h_\ell}] = 1$. This is achieved by initializing the weights such that:

$$\text{Var}[w_{h_{\ell-1},h_\ell}] = \frac{1}{\text{fan}_{\ell,in}} \quad \Rightarrow \quad r = \sqrt{\frac{3}{\text{fan}_{\ell,in}}} \tag{3.5.2}$$

where r is the symmetric uniform distribution parameter, similar to that used in He-Initialization. $\text{fan}_{\ell,in}$ is $H_{\ell-1}k$ because the weights are used in convolutional layers.

Using the SELU activation function, as a replacement for the ReLU and normalization layers, and the derived initializer from the SELU activation function, the SELU ByteNet model is created. The "Memorizing WMT NewsTest" experiment is then repeated using the SELU ByteNet model. The experiment ran for 300 epochs.

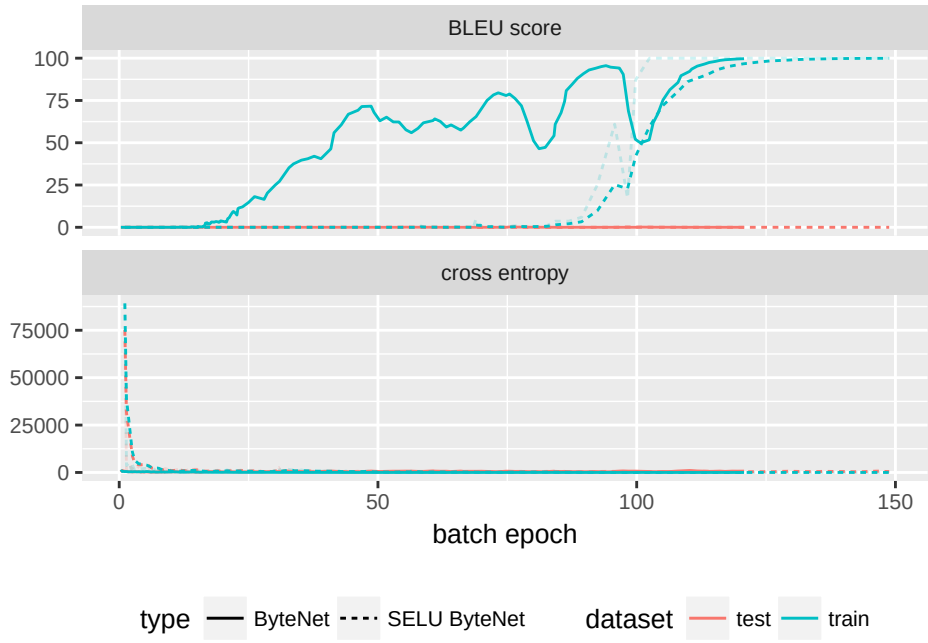


Figure 3.5.3: Shows BLEU score and cross entropy loss for the German to English WMT NewsTest dataset using the SELU ByteNet. The exponential moving average used a forget factor of 0.1.

The convergence for the "Memorizing WMT NewsTest" experiment using SELU ByteNet is somewhat similar to the normal ByteNet case (figure 3.5.3). However, there are some significant differences. The initial cross entropy loss is much higher in the SELU ByteNet case, this indicates that the derived initializer isn't the optimal choice. Secondly, the training BLEU score shows almost no improvement initially, then after 75 epochs it improves very fast. These two observations are likely connected, if the initialization is bad it will take a long time for the network to reach a similar state as the normal ByteNet model does initially. If this is true, and if it's possible to initialize the SELU ByteNet model better, the SELU ByteNet model may converge in much fewer iterations that the normal ByteNet model.

Most importantly, the fact that the SELU ByteNet converges much faster than the non-normalized ByteNet model, indicates that the weak-dependency assumptions are sufficiently satisfied. If this was not the case, convergence similarly to that in the non-normalized ByteNet experiment should be expected.

Finally, the synthetic digits experiment using the SELU ByteNet model, shows similar results as the normal ByteNet model (Appendix D.3).

3.5.1 Performance Profiling

Repeating the performance experiment, from both the normal ByteNet model and the simplified ByteNet model, shows that the SELU ByteNet model is extremely fast in comparison. Comparing the time spent running 300 epochs is actually a problem in this case, as the heating phase that transfers data and optimizes allocation takes up most of the time. However, comparing *obs./sec* shows that the SELU model is at least twice as fast as the normal ByteNet model.

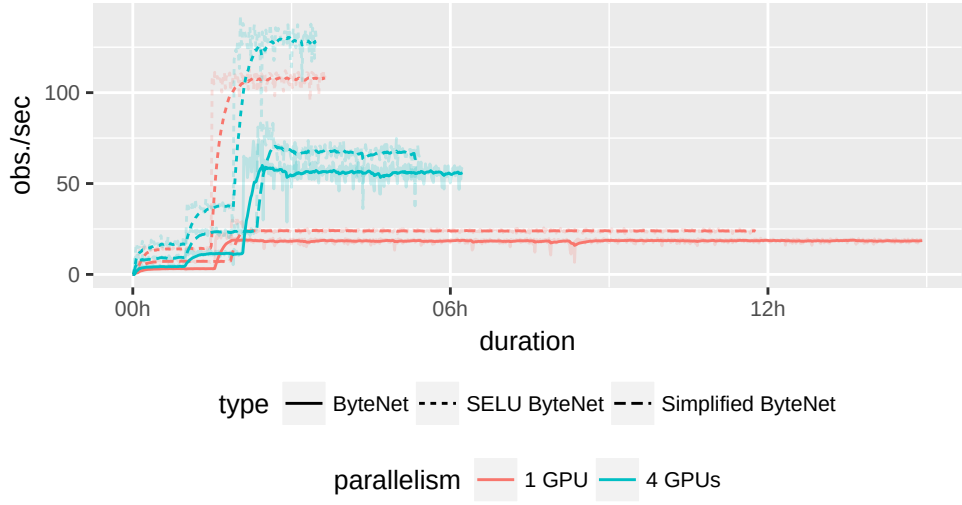


Figure 3.5.4: Comparing observations per second, depending on the number of GPUs used. The experiment learns the WMT NewsTest dataset over 300 epochs.

The processed profiling in figure 3.5.5 (the unprocessed plot is in appendix D.3), shows that the convoluted dilation is the most expensive part. This is completely reasonable, as this is the operation that involves the largest amount of raw computation. However, TensorFlow actually has an inefficient implementation of the dilated convolution, because dilated convolution isn't supported directly by CuDNN v5.1. CuDNN stands for CUDA Deep Neural Network and is a library developed by Nvidia that contains efficient implementations of common operations used in neural networks. The next version of CuDNN supports dilated convolution, but TensorFlow does not yet use this implementation [33].

Another interesting observation when looking at the processed profiling in figure 3.5.5, is the time spent in the "*pre-normalization*" layer, which now just contains the SELU activation. Comparing *pre-normalization* to the time spent in *recover-dim*, which just contains a *sequential-dense* layer, shows that the SELU activation uses an unreasonable amount of time. The SELU activation uses less raw computations

than the *sequential-dense* layer, and the SELU activation is purely an element-wise operation, thus it should be easy to parallelize. The reason for the poor performance is the before-mentioned naive execution of the computational graph by TensorFlow.

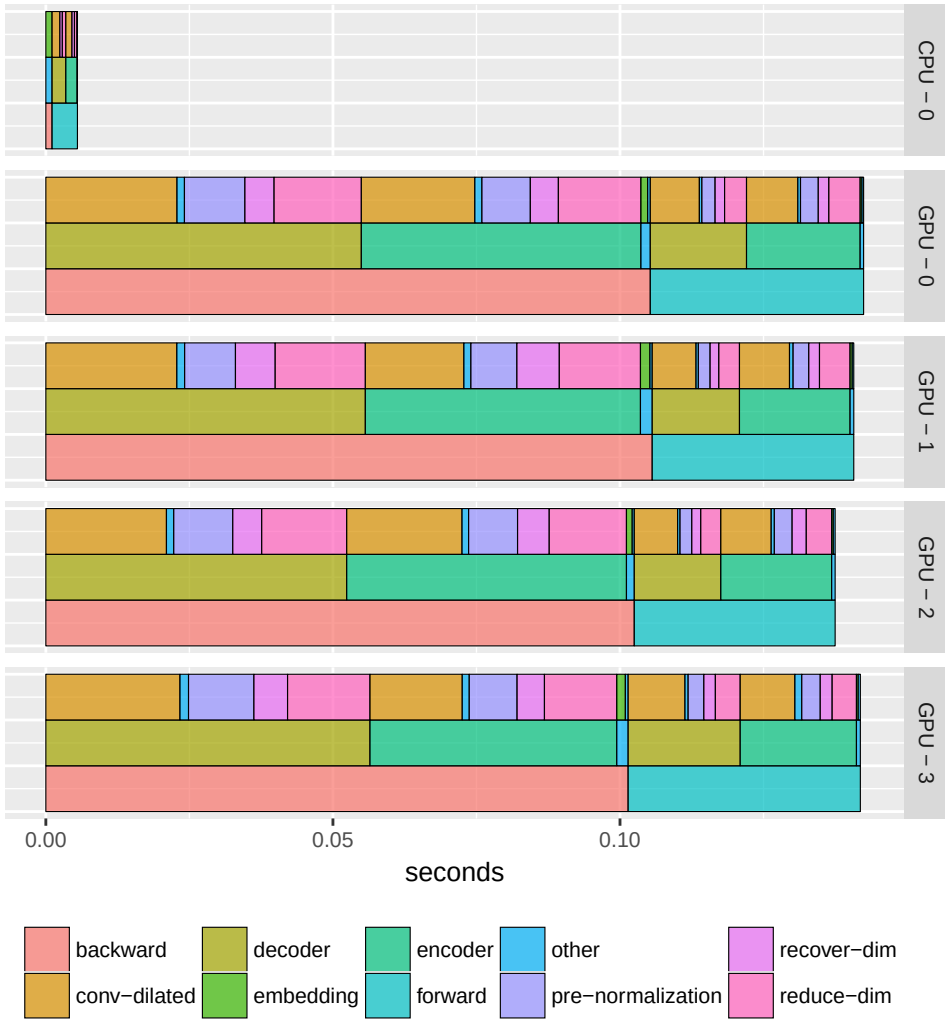


Figure 3.5.5: Shows time spent executing each part of the ByteNet model, this excludes the waiting time. Each part exists in a hierarchy, which is visualized as levels. Bottom level is the backward and forward pass. Second level is the encoder and decoder. Last level primarily splits the SELU ByteNet Residual Blocks.

3.5.2 WMT Translation Task

The Europarl v7 dataset is used to train the SELU ByteNet model. The setup is identical to that previously used to train the normal and simplified ByteNet model.

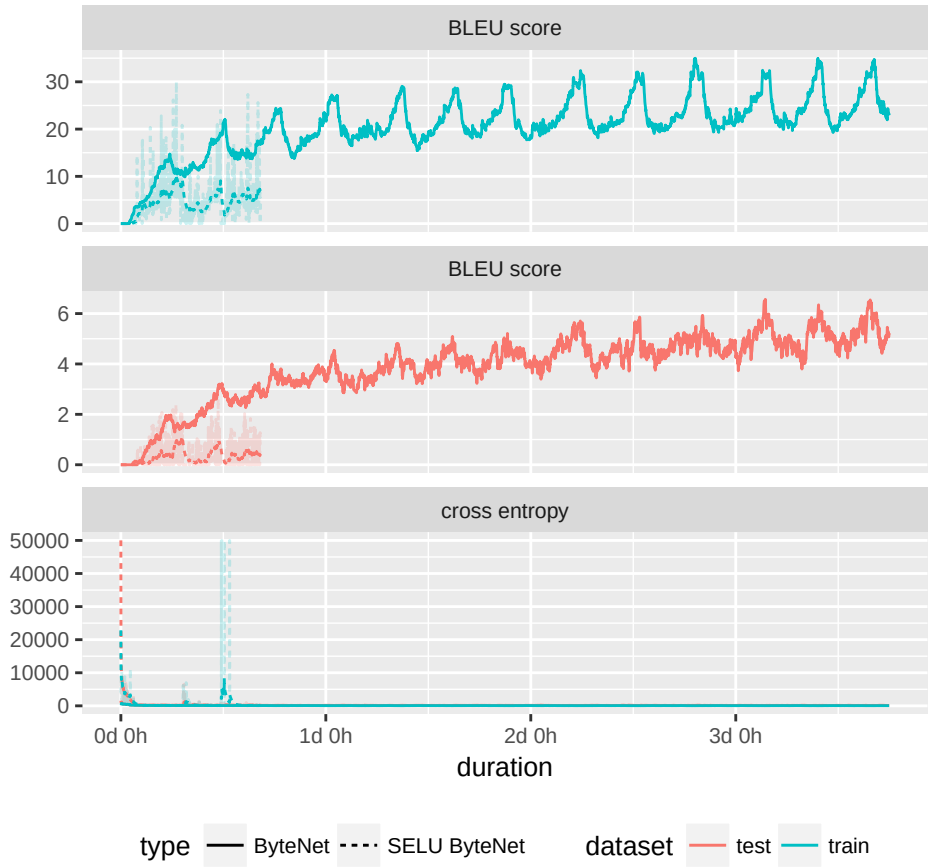


Figure 3.5.6: Shows BLEU score and cross entropy loss for the SELU ByteNet model, trained on Europarl v7 and tested on WMT NewsTest 2015. Both training and test measures are calculated on a randomly sampled mini-batch from each dataset. The exponential smoothing used a forget factor of 0.05. The raw data for the non-simplified ByteNet model is not shown.

From figure 3.5.6 it is very apparent that the SELU model is extremely fast in comparison to the normal ByteNet model, it completes all 13 epochs in less than a day. However, it also learns extremely poorly.

The cross entropy on the training dataset spikes on occasion, this could explain some of the poor learning behavior. Such an issue is typically solved with gradient clipping, where a hard upper bound is set on the gradient, preventing an exploding gradient. However, the idea behind the SELU activation function is specifically to prevent vanishing and exploding gradients, thus it doesn't seam reasonable to employ additional techniques to prevent exploding gradients.

Another possibility is that the Adam optimization parameters need to be much different when the SELU activation function is used. However, the Adam optimizer acts under most conditions as a trust-region, thus the parameters shouldn't affect the output too much [14].

It is possible that with enough tuning SELU could be made functional, in which case SELU ByteNet would be a very powerful model. However, understanding the behavior of SELU is not easy. The proof for the convergence properties of SELU alone is 90+ pages [31]. This makes the SELU activation function hard to reason about, thus it is hard to make educated guesses about how to tune the model.

Finally, it is possible that the sparse nature of ReLU is essential for creating a natural language model, where sparse patterns often occur. It is thus possible that the SELU activation function will never work for natural language translation.

3.6 Semi-Supervised ByteNet

The "Semi-Supervised Learning for NMT 2.7" theory section, describes a general idea for doing semi-supervised learning in neural machine translation. The method described does not depend on a specific neural network architecture, but can in theory work using any supervised translation model.

The semi-supervised ByteNet model combines the generalized "Semi-Supervised Learning for NMT" ideas with the supervised ByteNet model.

3.6.1 Synthetic Digits Problem

The ByteNet model in itself is rather slow at learning, at least given the current state of TensorFlow. The strategy presented in "Semi-Supervised Learning for NMT" does not make this any better. In fact, because the unsupervised part of the loss requires inference using BeamSearch, the execution time will increase linearly with respect to the beam size. The situation may be even worse for ByteNet since ByteNet when used supervised allows for full parallelization over both the source and target sequence. When inference is done on the ByteNet model, as it is in the unsupervised case, only the encoder part is supervised, thus only the encoder can be parallelized.

Because of these complications, it is not feasible to apply the semi-supervised ByteNet model on the full Europarl v7 dataset and another monolingual (unlabeled) dataset. Instead, to show that the model works and validate the implementation, the model is applied to the synthetic digits problem.

Since the synthetic digits dataset can be randomly generated, 3 datasets created from 3 different random seeds are used. A bilingual (labeled) training dataset, a monolingual (unlabeled) training dataset, and a test dataset. The monolingual dataset does only contain the spelled out words. A fourth dataset containing only digits could also be used, but the original article showed that this had little benefit, thus to conserve computation time a fourth dataset was not used [7].

The test dataset has 1024 observations, this includes most digit combinations. The number of observations in the bilingual and monolingual training dataset is varied in different experiments, to observe the effect of the dataset size.

The setup is identical to that in the purely supervised synthetic digits experiment, that was used to validate the ByteNet models. The dimensionality is set to 20, the Adam optimizer with a learning rate of 0.001 is used for optimization. The model ran for 300 epochs over the bilingual dataset. Additionally, the beam size for the unsupervised part is set to 5 sequences, and the model is parallelized over 2 GPUs.

The multi-GPU parallelization is done a little different that in the purely supervised experiments. Because the semi-supervised setup uses two separate translation models,

the two translation models are kept on different GPUs. By doing this the weight updates doesn't have to be synchronized through the CPU, which have a large cost. The updates are however still done synchronously, there is just less I/O involved in the process.

To compare the predictive performance, an attention based baseline model, similar to the one used in the original semi-supervised paper [7], is used. This is a multiplicative attention model [34] that has 3 RNN layer in the encoder, uses GRU units in the attention layer, and a GRU layer in the final output layer. This should be somewhat similar to the ByteNet model, that has 3 "layer" of hierarchical-dilated-convolutions in the encoder.

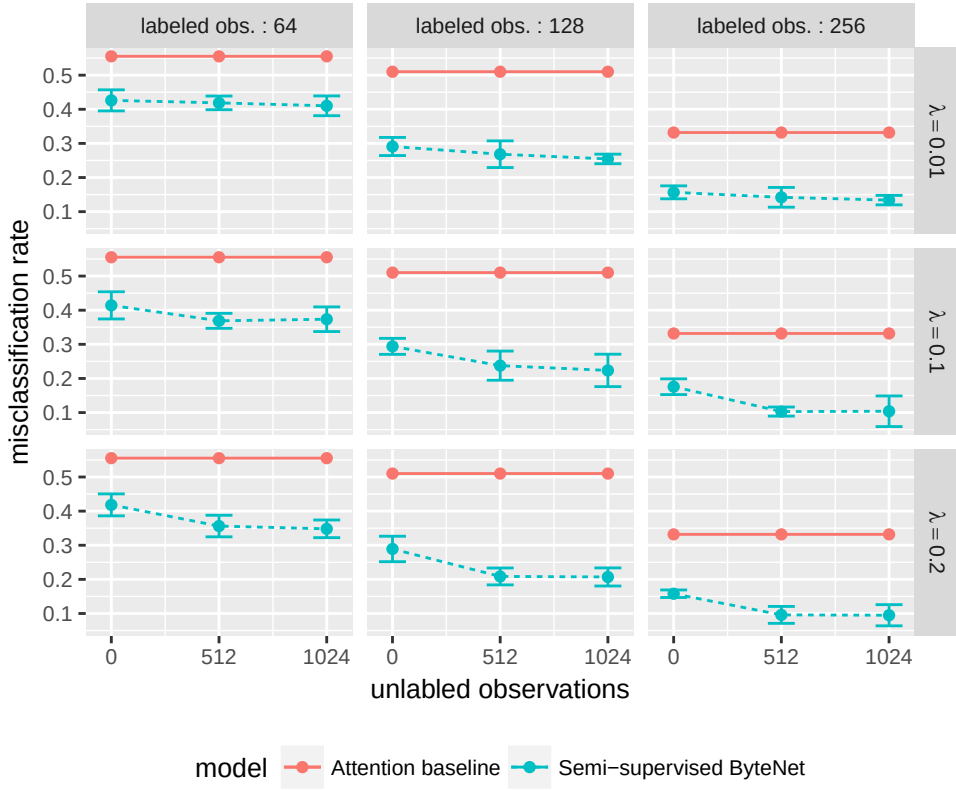


Figure 3.6.1: Shows the semi-supervised ByteNet model test performance depending on *labeled dataset size*, *unlabeled dataset size*, and *unlabeled learning factor* (λ). 95% confidence intervals are shown, each interval is calculated from 5 independent experiments.

Figure 3.6.1 shows similar results to those in the original paper, where the research

team found some improvement by using unlabeled observations, but nothing outstanding [7]. This may sound like a disappointment, but in general, it is very hard to improve translation models dramatically. The fact that most translation models can be improved by using monolingual (unlabeled) data is actually very encouraging.

A significant difference between this experiment, using the synthetic digits dataset, and an experiment running on a proper natural language translation dataset, is that the digits dataset has no variation in its output given the input. This means that "one" will always correspond to "1", while in natural languages like German "bitten" can be understood as both a "request" and as an "invite" in English. The consequence of this is that given a correct translation, the unsupervised marginalization will approximately reduce to a marginalization over just one sequence sample, which is not very powerful and wastes a lot of calculations. On the other hand, for bad translations, which is particularly common during the initial training, having a wide beam in the BeamSearch will definitely contribute to the model prediction performance, through the multiple predicted translations in the marginalization. Indeed, if this was not the case, it would be very hard to explain the performance improvements.

Finally, it is surprising that there is no statistical difference between using 512 or 1024 unlabeled observations. A reasonable explanation is that using 512 observations covers the vast majority of the variation in the problem, adding the final 1024 observations thus doesn't add much. Furthermore, no insurance was made to prevent duplicate observations (observations was independently sampled). Given that there are only $10^3 + 10^2 = 1100$ different observations, it is very likely that many of the observation are duplicates, this is similar to the birthday probability paradox.

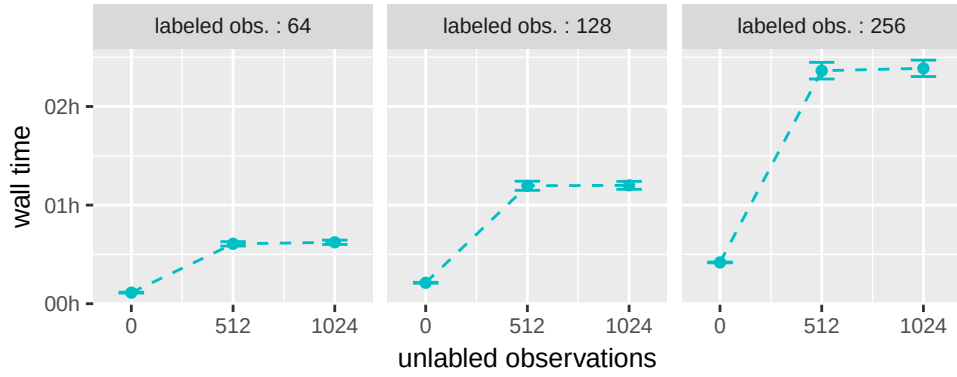


Figure 3.6.2: Shows the time spent running 300 epochs over the bilingual training dataset. The unlabeled learning rate is aggregated out (mean) since this has no theoretical nor practical performance impact. The 95% confidence intervals are thus calculated from $3 \cdot 5 = 15$ observations.

In figure 3.6.2 the computational performance penalty for using unlabeled observa-

tions is very apparent. However, it is not as bad as one would theoretically expect. By using a beam size of 5 sequences, followed by translations one these sequences, one would expect the training to run 10 times slower, but in practice, it is actually closer to 5. The computational performance is also not affected by the number unlabeled observations (as long as some are used), this is expected as the number of iteration-steps only depends on the number of labeled observations.

The good performance can perhaps be explained by how well ByteNet parallelize. While BeamSearch does prevent parallelization over the sequence, a computational trick was used to allow some parallelization. First the BeamSearch on the "text to digit" translator is used to sample the digit sequences. Then the "text to digit" translator is reapplied on the sampled digit sequences. This reapplying is very fast as it doesn't involve any inference, and in particular, it allows the backward pass to be calculated in parallel.

Another contributing factor to the performance is the small dimensionality of the ByteNet models used in the experiment, this likely means that the GPU has plenty of computational resources to run both the supervised and unsupervised part in parallel. This would not be the case when ByteNet is used on natural languages, as the required dimensionality is much larger.

CHAPTER 4

Conclusion

Machine translation is a difficult and fast-moving field. Over the 6 months, this thesis has been carried out, several papers with new translation models have been published. Each model shows a new state-of-the-art performance over the previous models, on the WMT translation task [10, 35, 36]. These models are created by Google, Microsoft, Facebook, etc., who have both computational resources and manpower to implement and experiment with a huge number of models. The goal and expectation of this thesis are thus not to compete with these models, but rather explore a new direction for neural machine translation, by using additional monolingual (unlabeled) data. However, to do this a fast and decent translation model is required, thus much effort has been put into the supervised model.

ByteNet The ByteNet model was able to learn both the synthetic digits problem and memorize the WMT NewsTest dataset, this validates the implementation and capabilities of the ByteNet model. For the real problem where the Europarl v7 dataset is used, ByteNet managed to achieve a BLEU score of 7.44 and produce reasonably good translations. This is far from the state-of-the-art model [35] or just a phase-based (PBMT) baseline model [36], but given the limited resources the results are rather good. The primary reason for why a better BLEU score wasn't achieved, is that training the ByteNet model is too time-consuming.

The original ByteNet paper including the latest revision did not disclose how much time was spent training or how many resources they used. From a scientific point of view, this is rather inadequate, especially because they created the model from a computational performance perspective. A recent blog post from Google published in June 2017, compares the different machine translation models and shows that ByteNet is actually one of the most computationally expensive models [36]. Their comparison shows the ByteNet model requires "8 days on 32 GPUs" (equivalent to 256 computational days), while the recently published state-of-the-art model [35] uses "3 days on 8 GPUs" (equivalent to 24 computational days) [36].

It is possible that if: the oscillation issues were solved, TensorFlow improves the execution speed with the XLA just-in-time (JIT) compiler [30], and if CuDNN added support for dilated convolution and normalization, that ByteNet could run in reasonable time. However, it will likely take at least a year before TensorFlow and CuDNN are at this state.

Simplified ByteNet Because ByteNet takes too much time to train, a simplified version of ByteNet was created. This model has 1/3 the layers and 5/9 the weights, while still maintaining the founding principals behind ByteNet and maintains the dimensional bottleneck. This was done by reducing the embedding dimensionality and removing the dimensional compression and decompression layers.

On the validation problems, this performed identically or slightly better compared to the normal ByteNet model. However, when trained on the Europarl v7 dataset the model showed severe overfitting and achieves a test BLEU score of only 0.55. This indicates that the compression and decompression layers are somehow essential regularization components. In terms of time spent training, the simplified ByteNet model is only 20% faster and converges correspondingly slower. Profiling the simplified ByteNet model, did unfortunately not reveal why the model is only 20% faster, but did reveal that normalization is the primary bottleneck in terms of time spent.

The results indicate that simplifying ByteNet is not a direction worth exploring further, at least not for natural language translation. The approach might be valid for simpler problems, but for natural language translation, there are likely completely different model architectures that are better suited.

SELU ByteNet Identifying that normalization is the primary computational bottleneck, suggests that if these layers can somehow be removed, the ByteNet model should perform significantly better. Training a non-normalized ByteNet model on even simple tasks converges very slowly, this is likely due to a vanishing gradient issue. Recently, a paper showed that a special activation called SELU (Self-normalizing Exponential Linear Unit) can be used as a replacement for the typical ReLU and normalization combination. By using the SELU activation as a replacement for ReLU and normalization, the ByteNet model converges quickly on the validation tasks.

On the Europarl v7 dataset, the SELU ByteNet model does not converge. This appears to be caused by exploding gradients. Such issues could be solved by gradient clipping. However, the purpose of SELU is exactly to prevent exploding and vanishing gradients issues, thus gradient clipping is likely not a good strategy.

In terms of computational performance, the SELU ByteNet is much faster than both the normal ByteNet model and the simplified ByteNet model. On the Eurparl v7 problem the SELU ByteNet model runs 13 epochs in 18 hours, the normal ByteNet model uses 4 days on the same number of epochs. Profiling reveals that the time is primarily spent in the SELU activation function. This validates the known issue, where TensorFlow spends an unreasonably amount of time running simple element-wise operations [30].

From a computational perspective, SELU ByteNet is promising, however as is, it is not suitable for learning translation on the Eurparl v7 dataset. If SELU ByteNet could be made to work, ByteNet would be a serious contender to other models in terms of training time.

Semi-Supervised ByteNet Without a suitable fast translation model, it is not feasible to run the semi-supervised ByteNet model on the Europarl v7 dataset. This is because the semi-supervised model depends on two translation models, each running on translations produced by a BeamSearch algorithm. Thus, one should expect the semi-supervised model to take approximately 10 times longer to train.

With the computational demands of ByteNet in mind, the semi-supervised ByteNet model was used on a synthetic problem. This problem maps spelled-out digits to digit symbols, for example, "one zero four" is translated to "104". On this problem, the semi-supervised ByteNet model showed improvement over both the supervised ByteNet model and an attention-based baseline model.

This validates the results from the original paper [7], where the BLEU score could be improved by 1.5 up to 3.5. While such an improvement doesn't completely solve all issues, when training on language-pairs where the bilingual dataset is small, it could be an important component when the bilingual data isn't plentiful but the monolingual data is.

Future Works With the recent results published by Google, showing that ByteNet is actually one of the slowest machine translation models [36], ByteNet is not a likely model for neural machine translation, and even less likely as a semi-supervised model. It is possible that the SELU ByteNet model could be tuned to work, in which case it might be a competing model. More time should be spent exploring possible methods of removing the normalization layers.

Recently a new paper showed that attention can be used instead of bi-directional RNN layers or hierarchical dilated convolution. Using this approach, they created a neural machine translation model that achieves a new state-of-the-art BLEU score on the en-de WMT NewsTest translation task. The model is also faster than many previous models, as it shares many of the founding principals of ByteNet. That is, resolution persistent encoding, parallelization over the sequences, and training in linear time are essential properties for a translation model [35].

Such a translation model is close to ideal, to be used in combination with the semi-supervised approach. Although, because the model is based on attention, inference can't be done in linear time, unlike ByteNet. Linear time inferences may be essential for computing the unsupervised part of the loss-function. That being said, it is only the forward pass of one of the translation models that will run in quadratic time. Both backward passes and the other translation model will run in linear-time.

A different approach for making the semi-supervised model converge within reasonable time, is to initialize the weights with the learned weights from the two translation models it uses, where the translation models are pre-trained on a bilingual dataset. This would work as a pseudo-transfer learning, "pseudo" because the application domain is the same and it is the same models that are used. While this may improve

convergence for a reasonable fast translation model, ByteNet is still too slow to be fully trained on just a single bilingual dataset.

Finally, one could look at a completely different approach for semi-supervised machine translation. In image processing, adversarial networks have been successfully used to "translate" between for example a zebra and a horse, without paired training data [23]. A similar approach could perhaps be used in the field of natural language translation. However, the current state of adversarial networks for natural languages is still far behind compared to other applications [8].

APPENDIX A

Notation

symbol	meaning
z	The activation input, calculated as a weighted sum over the input.
a	The activation output, $a = \theta(z)$.
θ	The non-linear activation function.
H	The amount of units in the layer.
h	The index of a units in the layer.
K	The amount of classes calculated in the softmax, $K = H_{L+1}$.
ℓ	The layer index.
L	The amount of hidden layers. Including the softmax output layer there are $L + 1$ layers.
t	A target value. If used in a superscript it is the sequence index.
w	A weight used in a neural neural network.
x	The input to the neural network, $x = z_{h_0}$.
y	The softmax output.
$\mathcal{L}$	The loss function (should always be minimized).
δ	Bookkeeping value used in backward propagation.
α	Learning rate used in the Adam Optimization algorithm.
γ	Scale parameter in normalization.
β	Bias parameter in normalization.
d	The dimensionality of the vector representation.
$*$	The convolution operator.
$\otimes$	The Kronecker product.
k	The kernel width in a 1d-convolution.
r	The dilation rate used in dilated convolution.
b	The beam size used in the BeamSearch algorithm.

Table A.1: Meaning of commonly used symbols.

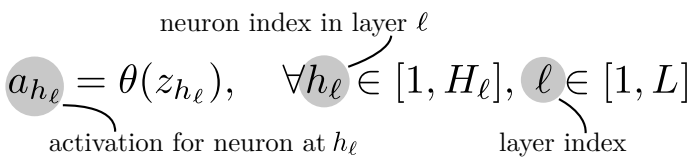


Figure A.1: A two-level subscript is used for the neuron index.

APPENDIX B

Backward Pass

B.1 Softmax

The forward pass for the softmax and the cross entropy loss function, is given by:

$$y_k = \frac{\exp(z_k)}{\sum_{k'=1}^{K} \exp(z_{k'})}, \quad \text{where: } z_k = z_{h_{L+1}}, K = H_{L+1}$$
$$\mathcal{L} = -\sum_{k=1}^{K} t_k \ln(y_k)$$

Equation B.1.1: Forward equations Cross Entropy loss with Softmax input.

The last delta ($\delta_{h_{L+1}}$) is what should be derived:

$$\delta_{h_{L+1}} = \delta_k = \frac{\partial \mathcal{L}}{\partial z_k} = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial z_k} = y_k - t_k \tag{B.1.2}$$

The first derivative $\frac{\partial \mathcal{L}}{\partial y_{k'}}$, is derived from the cross entropy equation:

$$\frac{\partial \mathcal{L}}{\partial y_{k'}} = \frac{\partial}{\partial y_{k'}} \left(-\sum_{k''=1}^{K} t_{k''} \ln(y_{k''}) \right) = -\frac{t_{k'}}{y_{k'}} \tag{B.1.3}$$

The other derivative $\frac{\partial y_{k'}}{\partial z_k}$, can be derived from the softmax function:

$$\begin{aligned}
\frac{\partial y_{k'}}{\partial z_k} &= \frac{\partial}{\partial z_k} \frac{\exp(z_{k'})}{\sum_{k''=1}^{K} \exp(z_{k''})} \\
&= \frac{\frac{\partial}{\partial z_k} \exp(z_{k'})}{\sum_{k''=1}^{K} \exp(z_{k''})} - \frac{\exp(z_{k'}) \frac{\partial}{\partial z_k} \sum_{k''=1}^{K} \exp(z_{k''})}{\left(\sum_{k''=1}^{K} \exp(z_{k''}) \right)^2} \\
&= \frac{\frac{\partial}{\partial z_k} \exp(z_{k'})}{\sum_{k''=1}^{K} \exp(z_{k''})} - \frac{\exp(z_{k'})}{\sum_{k''=1}^{K} \exp(z_{k''})} \frac{\frac{\partial}{\partial z_k} \sum_{k''=1}^{K} \exp(z_{k''})}{\sum_{k''=1}^{K} \exp(z_{k''})}
\end{aligned} \tag{B.1.4}$$

Because of the difference in index, the first term is only non-zero when $k = k'$, in which case y_k is the derivative. It thus becomes useful to define:

$$\delta_{i,j} = \begin{cases} 1 & \text{when } i = j \\ 0 & \text{otherwise} \end{cases} \tag{B.1.5}$$

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

$$\frac{\partial y_{k'}}{\partial a_k} = \delta_{k,k'} y_k - y_{k'} y_k \tag{B.1.6}$$

The result from (B.1.3) and (B.1.6) is then combined into (B.1.2):

$$\begin{aligned} \delta_{h_{L+1}} = \delta_k &= \sum_{k'=1}^{K} -\frac{t_{k'}}{y_{k'}} \left(\delta_{k,k'} y_k - y_{k'} y_k\right) = \sum_{k'=1}^{K} -\frac{t_{k'}}{y_{k'}} \delta_{k,k'} y_k + \sum_{k'=1}^{K} \frac{t_{k'}}{y_{k'}} y_{k'} y_k \\ &= -\frac{t_k}{y_k} y_k + y_k \sum_{k'=1}^{K} t_{k'} = -t_k + y_k = y_k - t_k \end{aligned} \tag{B.1.7}$$

To get $\sum_{k'=1}^{K} t_{k'} = 1$ it's used that $\{t_{k'}\}_{k'=1}^{K}$ is the target distribution and thus must sum to 1.

B.2 Dilated Convolution

$$z_{h_\ell}(t) = (a_{\ell-1} *_r w_{:,h_\ell})(t) = \sum_{h_{\ell-1}}^{H_{\ell-1}} \sum_i a_{h_{\ell-1}}(t + r\, i) w_{h_{\ell-1},h_\ell}(i)$$
$$a_{h_\ell}(t) = \theta(z_{h_\ell}(t))$$

Equation B.2.1: Forward equations for Dilated Convolution.

For deriving the backward pass the padding will be ignored, this is not a big issue as the input image $a_{\ell-1}$ can easily be extended. It also turns out that the final derivative makes the generalization quite intuitive.

For the backward pass we just which to derive:

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}(i)} \tag{B.2.2}$$

The weight $w_{h_{\ell-1},h_\ell}(i)$ affects all time steps, thus the chain rule yields:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}(i)} &= \sum_t \frac{\partial \mathcal{L}}{\partial z_{h_\ell}(t)} \frac{z_{h_\ell}(t)}{\partial w_{h_{\ell-1},h_\ell}(i)} \\ &= \sum_t \delta_{h_\ell}(t) a_{h_{\ell-1}}(t + r\, i) \end{aligned} \tag{B.2.3}$$

Here $\delta_{h_\ell}(t)$ is defined as:

$$\delta_{h_\ell}(t) \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial z_{h_\ell}}(t) \tag{B.2.4}$$

To calculate $\delta_{h_\ell}(t)$ the chain rule is used again, the first step is easy:

$$\delta_{h_\ell}(t) = \frac{\partial \mathcal{L}}{\partial z_{h_\ell}(t)} = \frac{\partial \mathcal{L}}{\partial a_{h_\ell}(t)} \frac{\partial a_{h_\ell}(t)}{\partial z_{h_\ell}(t)} = \theta'(z_{h_\ell}(t)) \frac{\partial \mathcal{L}}{\partial a_{h_\ell}(t)} \tag{B.2.5}$$

The next steps are more complicated, $a_{h_\ell}(t)$ affects $z_{h_{\ell+1}}(t')$ for all channels in the next layer, and all times t' that are within the kernel width of t.

$$\delta_{h_\ell}(t) = \theta'(z_{h_\ell}(t)) \frac{\partial \mathcal{L}}{\partial a_{h_\ell}(t)} = \theta'(z_{h_\ell}(t)) \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \sum_i \frac{\partial \mathcal{L}}{\partial z_{h_{\ell+1}}(t + r\, i)} \frac{\partial z_{h_{\ell+1}}(t + r\, i)}{\partial a_{h_\ell}(t)}$$

To derive $\frac{\partial z_{h_{\ell+1}}(t+r\,i)}{\partial a_{h_\ell}(t)}$ it helps to write out $z_{h_{\ell+1}}(t+r\,i)$.

$$z_{h_{\ell+1}}(t+r\,i) = (a_\ell * w_{:,h_\ell})(t+r\,i) = \sum_{h_\ell}^{H_\ell} \sum_{i'} a_{h_\ell}(t+r\,i+r\,i')w_{h_\ell,h_{\ell+1}}(i') \tag{B.2.6}$$

From this one can observe that $t+r\,i+r\,i'=t$ only occurs when $i'=-i$, thus the $\delta_{h_\ell}(t)$ derivative becomes:

$$\begin{aligned}\delta_{h_\ell}(t) &= \theta'(z_{h_\ell}(t)) \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \sum_i \frac{\partial \mathcal{L}}{\partial z_{h_{\ell+1}}(t+r\,i)} w_{h_\ell,h_{\ell+1}}(-i) \\ &= \theta'(z_{h_\ell}(t)) \sum_{h_{\ell+1}=1}^{H_{\ell+1}} \sum_i \delta_{h_{\ell+1}}(t+r\,i) w_{h_\ell,h_{\ell+1}}(-i)\end{aligned} \tag{B.2.7}$$

This is actually a dilated convolution, just where the weight is flipped (rotated).

$$\delta_{h_\ell}(t) = \theta'(z_{h_\ell}(t))(\delta_{\ell+1} *_r \mathrm{rot}(w_{:,h_{\ell+1}}))(t) \tag{B.2.8}$$

This is the derivative for the dilated convolution. Because the derivative is a convolution itself, it naturally generalizes to any padding implementation.

B.3 Batch Normalization

To derive the backward pass for batch normalization, it helps to setup equations for $\mathbb{E}[z_{h_\ell}]$ and $\textsc{Var}[z_{h_\ell}]$. To do this we need to reintroduce the observation index, this time it will be denoted with the superscript (i). Similarly, the mini-batch will be denoted with $\mathcal{B}$. With these changes the full forward pass can be written as:

Activation:

$$z_{h_\ell}^{(i)} = \sum_{h_{\ell-1}}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_\ell-1}^{(i)}$$

$$\hat{z}_{h_\ell}^{(i)} = \gamma_{h_\ell} \frac{z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}}}{\sqrt{\sigma_{h_\ell}^{2,\mathcal{B}} + \epsilon}} + \beta_{h_\ell}$$

$$a_{h_\ell}^{(i)} = \theta\left(\hat{z}_{h_\ell}^{(i)}\right)$$

Statistics:

$$\mu_{h_\ell}^{\mathcal{B}} = \frac{1}{n}\sum_{i=1}^{n} z_{h_\ell}^{(i)}$$

$$\sigma_{h_\ell}^{2,\mathcal{B}} = \frac{1}{n}\sum_{i=1}^{n} (z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}})^2$$

Equation B.3.1: Forward equations for Batch Normalization.

Now that the forward pass is stated explicitly, the backward pass can be derived. For the backward pass we which to derive:

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}}, \quad \frac{\partial \mathcal{L}}{\partial \gamma_{h_\ell}}, \quad \frac{\partial \mathcal{L}}{\partial \beta_{h_\ell}} \tag{B.3.2}$$

The gradient with respect to the weight $w_{h_{\ell-1},h_\ell}$ hasn't changed:

$$\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial z_{h_\ell}^{(i)}} \frac{\partial z_{h_\ell}^{(i)}}{\partial w_{h_{\ell-1},h_\ell}} = \sum_{i=1}^{n} \delta_{h_\ell}^{(i)} a_{h_{\ell-1}}^{(i)} \tag{B.3.3}$$

Note that without batch normalization the sum over the observations does not directly appear in the gradients, but this is only because the observation index i is omitted. In reality, it sums the gradients for each observation in the mini-batch.

The δ_{h_ℓ} however has changed quite significantly:

$$\begin{aligned}\delta_{h_\ell}^{(i)} = \frac{\partial \mathcal{L}}{\partial z_{h_\ell}^{(i)}} &= \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\partial \hat{z}_{h_\ell}^{(i)}}{\partial z_{h_\ell}^{(i)}} + \frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} \frac{\partial \sigma_{h_\ell}^{2,\mathcal{B}}}{\partial z_{h_\ell}^{(i)}} + \frac{\partial \mathcal{L}}{\partial \mu_{h_\ell}^{\mathcal{B}}} \frac{\partial \mu_{h_\ell}^{\mathcal{B}}}{\partial z_{h_\ell}^{(i)}} \\ &= \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\gamma_{h_\ell}}{\sqrt{\sigma_{h_\ell}^{2,\mathcal{B}} + \epsilon}} + \frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} \frac{2}{n} \left(z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}} \right) + \frac{\partial \mathcal{L}}{\partial \mu_{h_\ell}^{\mathcal{B}}} \frac{1}{n}\end{aligned} \tag{B.3.4}$$

From (B.3.4) the following derivatives also needs to be derived:

$$\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}}, \quad \frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}}, \quad \frac{\partial \mathcal{L}}{\partial \mu_{h_\ell}^{\mathcal{B}}} \tag{B.3.5}$$

As with everything else, this is done by using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} = \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^{(i)}} \frac{\partial a_{h_\ell}^{(i)}}{\partial \hat{z}_{h_\ell}^{(i)}} = \theta'(\hat{z}_{h_\ell}^{(i)}) \frac{\partial \mathcal{L}}{\partial a_{h_\ell}^{(i)}} = \theta'(\hat{z}_{h_\ell}^{(i)}) \sum_{h_{\ell+1}}^{H_{\ell+1}} \delta_{h_{\ell+1}}^{(i)} w_{h_\ell, h_{\ell+1}} \tag{B.3.6}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\partial \hat{z}_{h\ell}^{(i)}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \gamma_{h_\ell} \left(z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}} \right) \left(-\frac{1}{2} \right) \left(\sigma_{h_\ell}^{\mathcal{B}} + \epsilon \right)^{-\frac{2}{3}} \tag{B.3.7}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mu_{h_\ell}^{\mathcal{B}}} &= \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\partial \hat{z}_{h_\ell}^{(i)}}{\mu_{h_\ell}^{\mathcal{B}}} + \frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} \frac{\partial \sigma_{h_\ell}^{2,\mathcal{B}}}{\mu_{h_\ell}^{\mathcal{B}}} \\ &= \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \left(-\frac{1}{\sqrt{\sigma_{h_\ell}^{2,\mathcal{B}} + \epsilon}} \right) + \frac{\partial \mathcal{L}}{\partial \sigma_{h_\ell}^{2,\mathcal{B}}} \frac{1}{n} \sum_{i=1}^{n} -2 \left(z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}} \right)\end{aligned} \tag{B.3.8}$$

Finally, the gradient with respect to γ_{h_ℓ} and β_{h_ℓ} can be derived:

$$\frac{\partial \mathcal{L}}{\partial \gamma_{h_\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\partial \hat{z}_{h_\ell}^{(i)}}{\partial \gamma_{h_\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \left(\frac{z_{h_\ell}^{(i)} - \mu_{h_\ell}^{\mathcal{B}}}{\sqrt{\sigma_{h_\ell}^{2,\mathcal{B}} + \epsilon}} \right) \tag{B.3.9}$$

$$\frac{\partial \mathcal{L}}{\partial \beta_{h_\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \frac{\partial \hat{z}_{h_\ell}^{(i)}}{\partial \beta_{h_\ell}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}} \tag{B.3.10}$$

These only depend on $\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}^{(i)}}$, which have already been derived.

B.4 Layer Normalization

Let's repeat the forward pass:

Activation:

$$
\begin{aligned}
z_{h_\ell} &= \sum_{h_{\ell-1}}^{H_{\ell-1}} w_{h_{\ell-1},h_\ell} a_{h_\ell - 1} \\
\hat{z}_{h_\ell} &= \gamma_{h_\ell} \frac{z_{h_\ell} - \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} + \beta_{h_\ell} \\
a_{h_\ell} &= \theta\left(\hat{z}_{h_\ell}\right)
\end{aligned}
$$

Statistics:

$$
\begin{aligned}
\mu_\ell &= \frac{1}{H_\ell} \sum_{h_\ell}^{H_\ell} z_{h_\ell} \\
\sigma_\ell^2 &= \frac{1}{H_\ell} \sum_{h_\ell}^{H_\ell} (z_{h_\ell} - \mu_\ell)^2
\end{aligned}
$$

Equation B.4.1: Forward equations for Layer Normalization.

For the backward pass the following should be derived:

$$
\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}}, \quad \frac{\partial \mathcal{L}}{\partial \gamma_{h_\ell}}, \quad \frac{\partial \mathcal{L}}{\partial \beta_{h_\ell}} \tag{B.4.2}
$$

The gradient with respect to the weight $w_{h_{\ell-1},h_\ell}$ is the same:

$$
\frac{\partial \mathcal{L}}{\partial w_{h_{\ell-1},h_\ell}} = \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} \frac{\partial z_{h_\ell}}{\partial w_{h_{\ell-1},h_\ell}} = \delta_{h_\ell} a_{h_{\ell-1}} \tag{B.4.3}
$$

Like in batch normalization, δ_{h_ℓ} has changed quite significantly:

$$
\begin{aligned}
\delta_{h_\ell} = \frac{\partial \mathcal{L}}{\partial z_{h_\ell}} &= \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\partial \hat{z}_{h_\ell}}{\partial z_{h_\ell}} + \frac{\partial \mathcal{L}}{\partial \sigma_\ell^2} \frac{\partial \sigma_\ell^2}{\partial z_{h_\ell}} + \frac{\partial \mathcal{L}}{\partial \mu_\ell} \frac{\partial \mu_\ell}{\partial z_{h_\ell}} \\
&= \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\gamma_{h_\ell}}{\sqrt{\sigma_\ell^2 + \epsilon}} + \frac{\partial \mathcal{L}}{\partial \sigma_\ell^2} \frac{2}{H_\ell} (z_{h_\ell} - \mu_\ell) + \frac{\partial \mathcal{L}}{\partial \mu_\ell} \frac{1}{H_\ell}
\end{aligned} \tag{B.4.4}
$$

From (B.4.4) the following derivatives also needs to be derived:

$$
\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}}, \quad \frac{\partial \mathcal{L}}{\partial \sigma_\ell^2}, \quad \frac{\partial \mathcal{L}}{\partial \mu_\ell} \tag{B.4.5}
$$

As with everything else this is done by using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} = \frac{\partial \mathcal{L}}{\partial a_{h_\ell}} \frac{\partial a_{h_\ell}}{\partial \hat{z}_{h_\ell}} = \theta'(\hat{z}_{h_\ell}) \frac{\partial \mathcal{L}}{\partial a_{h_\ell}} = \theta'(\hat{z}_{h_\ell}) \sum_{h_{\ell+1}}^{H_{\ell+1}} \delta_{h_{\ell+1}} w_{h_\ell, h_{\ell+1}} \tag{B.4.6}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_\ell^2} = \sum_{h_\ell}^{H_\ell} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\partial \hat{z}_{h_\ell}}{\partial \sigma_\ell^2} = \sum_{h_\ell}^{H_\ell} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \gamma_{h_\ell} (z_{h_\ell} - \mu_\ell) \left(-\frac{1}{2} \right) \left(\sigma_\ell^2 + \epsilon \right)^{-\frac{2}{3}} \tag{B.4.7}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mu_\ell} &= \sum_{h_\ell}^{H_\ell} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\partial \hat{z}_{h_\ell}}{\partial \mu_\ell} + \frac{\partial \mathcal{L}}{\partial \sigma_\ell^2} \frac{\partial \sigma_\ell^2}{\partial \mu_\ell} \\ &= \sum_{h_\ell}^{H_\ell} \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \left(-\frac{1}{\sqrt{\sigma_\ell^2 + \epsilon}} \right) + \frac{\partial \mathcal{L}}{\partial \sigma_\ell^2} \frac{1}{H_\ell} \sum_{h_\ell}^{H_\ell} -2 \left(z_{h_\ell} - \mu_\ell \right) \end{aligned} \tag{B.4.8}$$

Finally, the gradient with respect to γ_{h_ℓ} and β_{h_ℓ} can be derived:

$$\frac{\partial \mathcal{L}}{\partial \gamma_{h_\ell}} = \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\partial \hat{z}_{h_\ell}}{\partial \gamma_{h_\ell}} = \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \left(\frac{z_{h_\ell} - \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} \right) \tag{B.4.9}$$

$$\frac{\partial \mathcal{L}}{\partial \beta_{h_\ell}} = \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \frac{\partial \hat{z}_{h_\ell}}{\partial \beta_{h_\ell}} = \frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}} \tag{B.4.10}$$

These only depend on $\frac{\partial \mathcal{L}}{\partial \hat{z}_{h_\ell}}$ which have already been derived.

B.5 Semi-Supervised Marginalization

Recall the partial loss function:

Loss function:

$$\mathcal{L} = -\log(P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}}))$$

Monolingual translation model:

$$P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}}) = \sum_{\mathbf{x}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}) P(\mathbf{x}|\mathbf{y}, \overleftarrow{\boldsymbol{\theta}})$$

Equation B.5.1: Loss function for the semi-supervised model.

To optimize the loss, the gradient with respect to $\overrightarrow{\boldsymbol{\theta}}$ is calculated. The gradient with respect to $\overleftarrow{\boldsymbol{\theta}}$ is also needed, but that is symmetrically similar to the $\overrightarrow{\boldsymbol{\theta}}$ gradient.

$$\frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} \mathcal{L} = \frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} -\log(P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})) \tag{B.5.2}$$

Start out by differentiating the $\log(\cdot)$ function and insert the expression for $P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})$:

$$\begin{aligned} \frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} -\log(P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})) &= -\frac{\frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})}{P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})} \\ &= -\frac{\sum_{\mathbf{x}} \frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}) P(\mathbf{x}|\mathbf{y}, \overleftarrow{\boldsymbol{\theta}})}{\sum_{\mathbf{x}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}) P(\mathbf{x}|\mathbf{y}, \overleftarrow{\boldsymbol{\theta}})} \end{aligned} \tag{B.5.3}$$

The identity $\frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = f(\boldsymbol{\theta}) \frac{\partial \log(f(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}}$ [**deeplearning**] is then applied.

$$\frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} -\log(P(\mathbf{y}'|\mathbf{y}, \overrightarrow{\boldsymbol{\theta}}, \overleftarrow{\boldsymbol{\theta}})) = -\frac{\sum_{\mathbf{x}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}) P(\mathbf{x}|\mathbf{y}, \overleftarrow{\boldsymbol{\theta}}) \frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} \log(P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}))}{\sum_{\mathbf{x}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}) P(\mathbf{x}|\mathbf{y}, \overleftarrow{\boldsymbol{\theta}})} \tag{B.5.4}$$

Calculating $\frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} \log(P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}}))$ is more numerically stable than calculating $\frac{\partial}{\partial \overrightarrow{\boldsymbol{\theta}}} P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}})$ directly. Furthermore, since $P(\mathbf{y}'|\mathbf{x}, \overrightarrow{\boldsymbol{\theta}})$ already has to be calculated there is no loss in performance.

APPENDIX C

Numerical Stability

C.1 The log-softmax function

$$\mathcal{S}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Equation C.1.1: The softmax function on input vector $\mathbf{x}$.

To make the softmax function numerically stable, the maximal value of the input vector $\mathbf{x}$ is subtracted, this is denotes as the alternative softmax function $\tilde{\mathcal{S}}(\mathbf{x}) = \mathcal{S}(\mathbf{x} - \max(\mathbf{x}))$.

The alternative softmax function $\tilde{\mathcal{S}}(\mathbf{x})$ is then simplified:

$$\begin{aligned}
\tilde{\mathcal{S}}(\mathbf{x})_i &= \frac{\exp(x_i - \max(\mathbf{x})))}{\sum_j \exp(x_j - \max(\mathbf{x})))} & &= \frac{\exp(x_i)\exp(-\max(\mathbf{x})))}{\sum_j \exp(x_j)\exp(-\max(\mathbf{x})))} \\
&= \frac{\exp(x_i)\exp(-\max(\mathbf{x})))}{\exp(-\max(\mathbf{x})))\sum_j \exp(x_j)} & &= \frac{\exp(x_i)}{\sum_j \exp(x_j)} \\
&= \mathcal{S}(\mathbf{x})_i
\end{aligned} \tag{C.1.2}$$

As seen from (C.1.2), the alternative softmax function $\tilde{\mathcal{S}}(\mathbf{x})$ is the same as $\mathcal{S}(\mathbf{x})$. However, subtracting the maximal value ($\max(\mathbf{x})$) from the input will typically yield a more numerically stable result.

To create a numerically stable log softmax function the log softmax ($\log(\mathcal{S}(\mathbf{x})_i) = \log(\tilde{\mathcal{S}}(\mathbf{x})_i)$) is simply used:

$$\begin{aligned}
\log(\mathcal{S}(\mathbf{x})_i) &= \log\left(\frac{\exp(x_i - \max(\mathbf{x})))}{\sum_j \exp(x_j - \max(\mathbf{x})))}\right) \\
&= (x_i - \max(\mathbf{x})) - \log\left(\sum_j \exp(x_j - \max(\mathbf{x}))\right)
\end{aligned} \tag{C.1.3}$$

C.2 Marginalization on log-probabilities

$$\mathcal{P}(\mathbf{x}) = \log\left(\sum_i \exp(x_i)\right)$$

Equation C.2.1: Calculates a marginalization from log-probabilities $\mathbf{x}$ to log-probability $\mathcal{P}(\mathbf{x})$.

To make $\mathcal{P}(\mathbf{x})$ numerically stable, the maximal value of the input vector $\mathbf{x}$ is subtracted, this is denotes as the alternative function $\tilde{\mathcal{P}}(\mathbf{x}) = \mathcal{P}(\mathbf{x} - \max(\mathbf{x}))$.

$$\begin{aligned}
\tilde{\mathcal{P}}(\mathbf{x}) &= \log\left(\sum_i \exp(x_i - \max(\mathbf{x}))\right) \\
&= \log\left(\sum_i \exp(x_i)\exp(-\max(\mathbf{x}))\right) \\
&= \log\left(\exp(-\max(\mathbf{x}))\sum_i \exp(x_i)\right) \\
&= \log(\exp(-\max(\mathbf{x}))) + \log\left(\sum_i \exp(x_i)\right) \\
&= -\max(\mathbf{x}) + \log\left(\sum_i \exp(x_i)\right)
\end{aligned} \tag{C.2.2}$$

As seen from (C.2.2), there is a $\max(\mathbf{x})$ difference between $\tilde{\mathcal{P}}(\mathbf{x})$ and $\mathcal{P}(\mathbf{x})$. The numerical stable version of $\tilde{\mathcal{P}}(\mathbf{x})$, is thus to subtract $\max(\mathbf{x})$ from the input and re-add it to the output.

$$\mathcal{P}(\mathbf{x}) = \log\left(\sum_i \exp(x_i - \max(\mathbf{x}))\right) + \max(\mathbf{x}) \tag{C.2.3}$$

APPENDIX D

Extra Results

D.1 ByteNet

D.1.1 WMT NewsTest profiling using one GPU

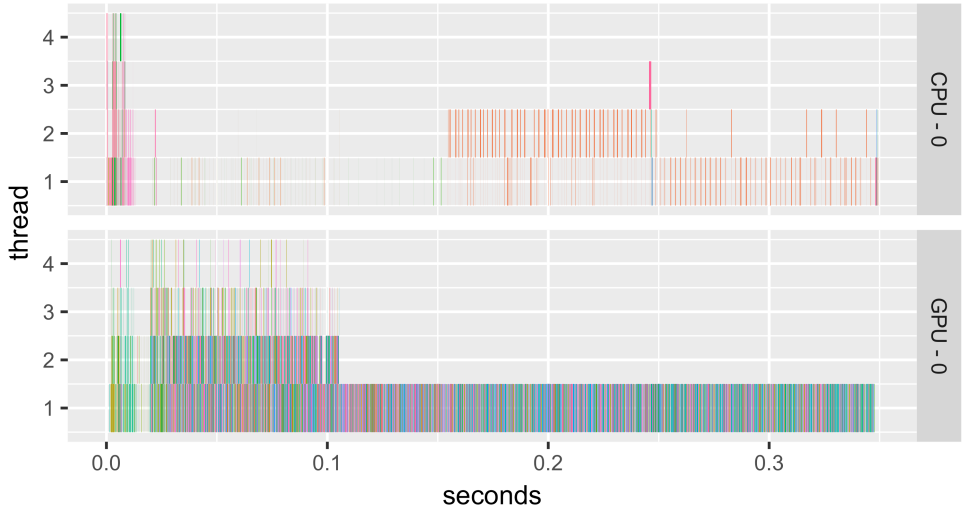


Figure D.1.1: Shows time spent on each operation, when the operation was executed, and on what GPU/CPU it was executed. The color coding indicates the operation type, there are more than a 100 different operation types, most are specific to TensorFlow, thus the legend is not included.

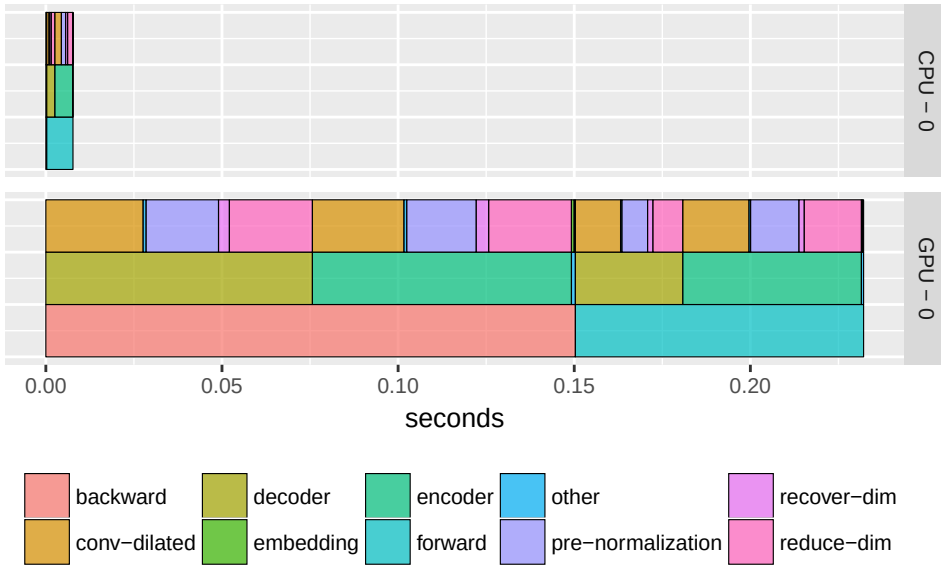


Figure D.1.2: Shows time spent executing each part of the ByteNet model, this excludes the waiting time. Each part exists in a hierarchy, which is visualized as levels. Bottom level is the backward and forward pass. Second level is the encoder and decoder. Last level primarily splits the SELU ByteNet Residual Blocks.

D.2 Simplified ByteNet

D.2.1 Learning Synthetic Digits Problem

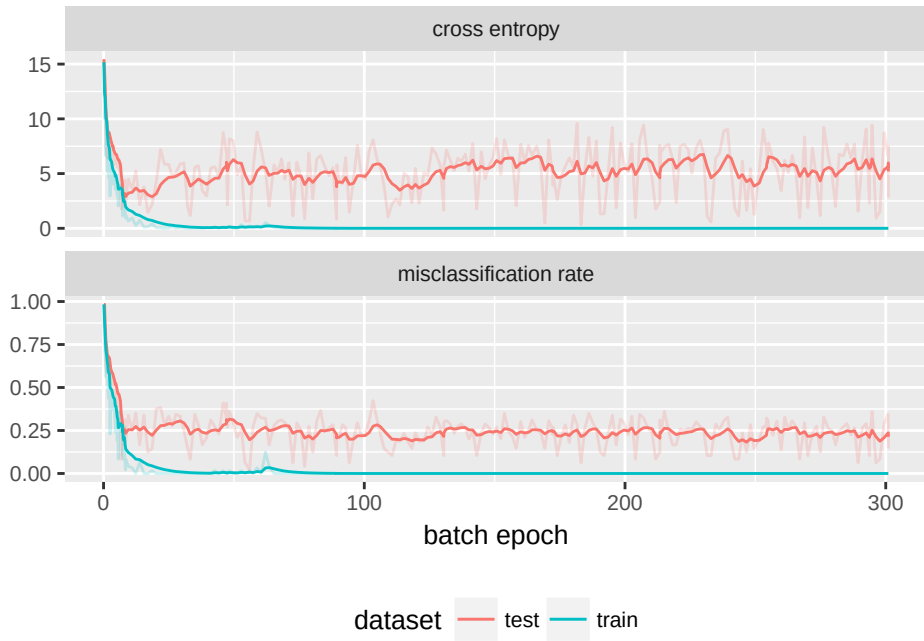


Figure D.2.1: Shows misclassification rate and the cross entropy loss. For comparison an attention model has a misclassification rate of 0.51. The exponential moving average used a forget factor of 0.2.

D.2.2 Memorizing WMT NewsTest

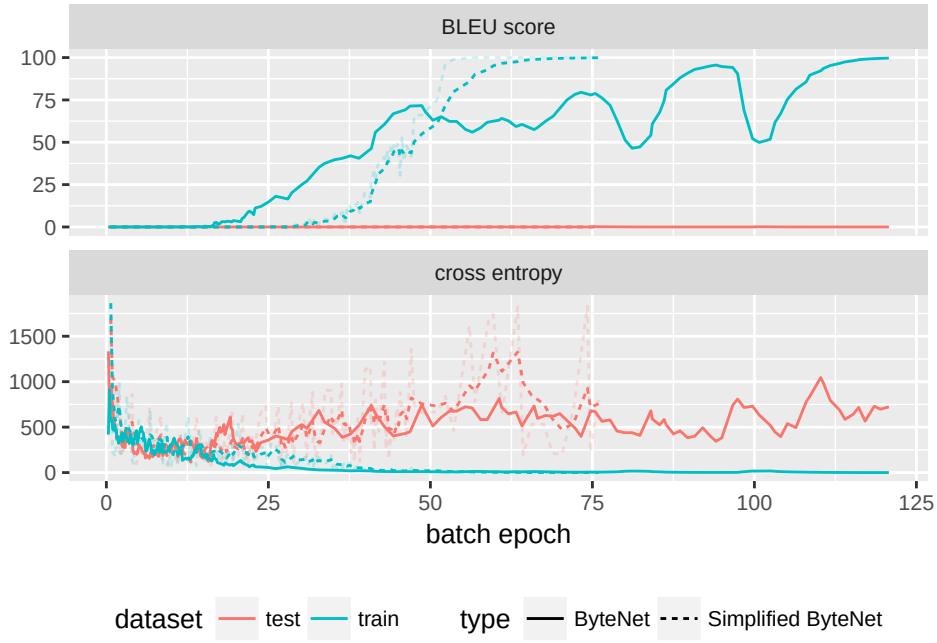


Figure D.2.2: Shows BLEU score and cross entropy loss for the German to English WMT NewsTest dataset. Both training and test measures are calculated on a randomly sampled mini-batch from each dataset. The exponential moving average used a forget factor of 0.3.

D.2.3 WMT NewsTest profiling using four GPUs

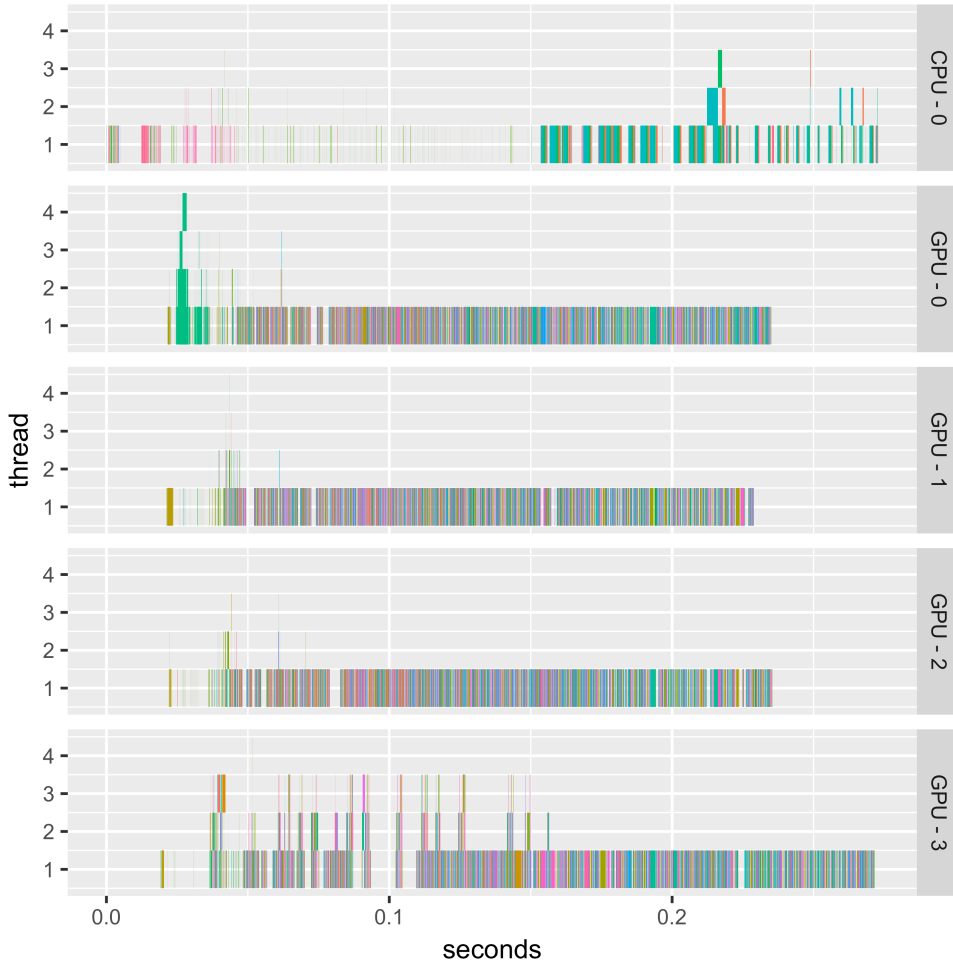


Figure D.2.3: Shows time spent on each operation, when the operation was executed, and on what GPU/CPU it was executed. The color coding indicates the operation type, there are more than a 100 different operation types, most are specific to TensorFlow, thus the legend is not included.

D.3 SELU ByteNet

D.3.1 Learning Synthetic Digits Problem

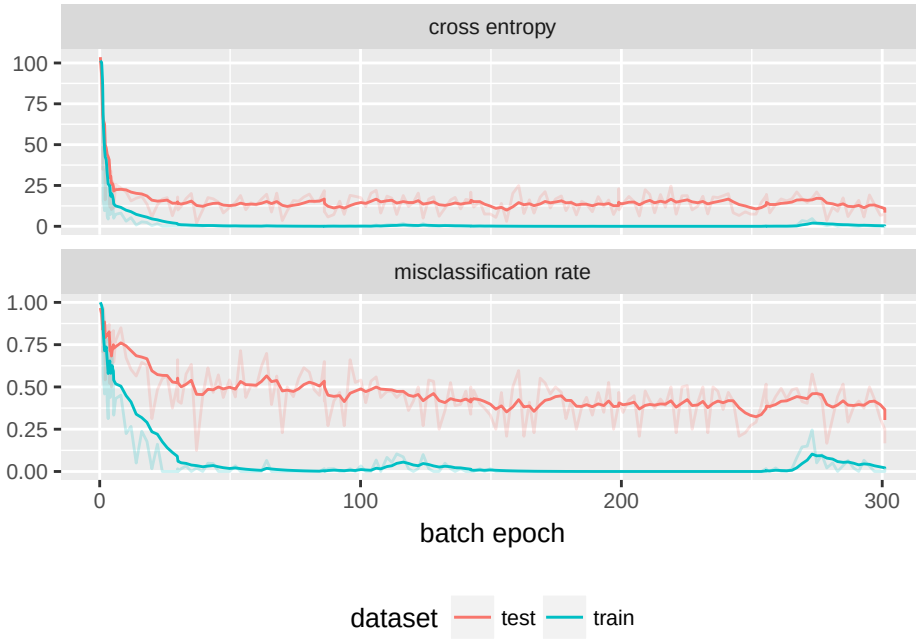


Figure D.3.1: Shows misclassification rate and the cross entropy loss. For comparison an attention model has a misclassification rate of 0.51. The exponential moving average used a forget factor of 0.2.

D.3.2 WMT NewsTest profiling using four GPUs

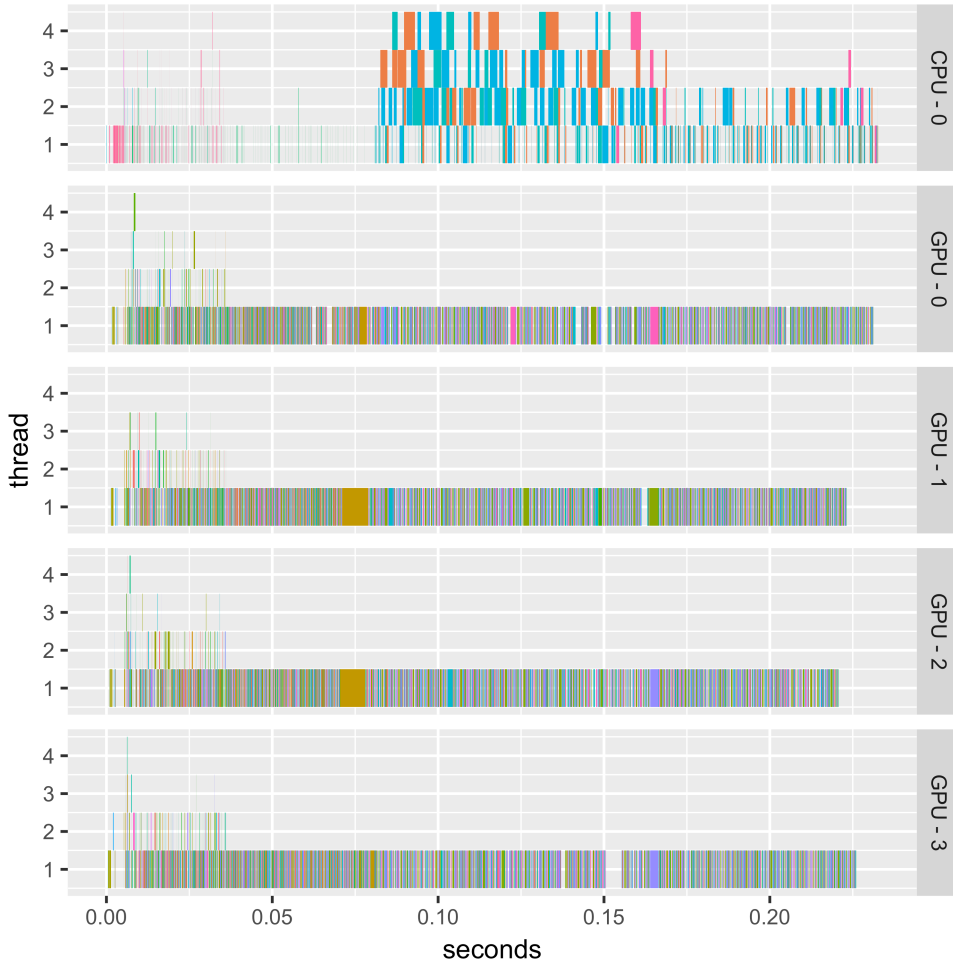


Figure D.3.2: Shows time spent on each operation, when the operation was executed, and on what GPU/CPU it was executed. The color coding indicates the operation type, there are more than a 100 different operation types, most are specific to TensorFlow, thus the legend is not included.

Bibliography

[1] E. Parliament. (2017). The profession of translator in the european parliament, [Online]. Available: http://www.europarl.europa.eu/multilingualism/trade_of_translator_en.htm (visited on 06/11/2017).

[2] Google. (2017). Making the internet more inclusive in india, [Online]. Available: https://blog.google/products/translate/making-internet-more-inclusive-india/ (visited on 06/11/2017).

[3] ——, (2017). Ten years of google translate, [Online]. Available: https://blog.google/products/translate/ten-years-of-google-translate/ (visited on 06/11/2017).

[4] F. J. Och and H. Ney, "A systematic comparison of various statistical alignment models," eng, *Computational Linguistics*, vol. 29, no. 1, pp. c–51, 2003, ISSN: 15309312, 08912017. DOI: 10.1162/089120103321337421. [Online]. Available: http://acl-arc.comp.nus.edu.sg/archives/acl-arc-090501d3/data/pdf/anthology-PDF/J/J03/J03-1002.pdf.

[5] Google. (2017). Research blog: A neural network for machine translation, at production scale, [Online]. Available: https://research.googleblog.com/2016/09/a-neural-network-for-machine.html (visited on 06/11/2017).

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

[7] Y. Cheng, W. Xu, Z. He, W. He, H. Wu, M. Sun, and Y. Liu, "Semi-supervised learning for neural machine translation," eng, *54th Annual Meeting of the Association for Computational Linguistics, Acl 2016 - Long Papers*, vol. 4, pp. 1965–1974, 2016. [Online]. Available: https://arxiv.org/abs/1606.04596.

[8] S. Rajeswar, S. Subramanian, F. Dutil, C. Pal, and A. Courville, "Adversarial generation of natural language," *ArXiv e-prints*, 11 pp., 11 pp. May 2017. arXiv: 1705.10929 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1705.10929.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ArXiv e-prints*, 2015. arXiv: 1409.0473 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1409.0473.

[10] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. van den Oord, A. Graves, and K. Kavukcuoglu, "Neural machine translation in linear time," *ArXiv e-prints*, 2016. arXiv: 1610.10099 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1610.10099.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] C. M. Bishop, *Pattern recongnition and machine learning*. Springer, 2009, ISBN: 9780387310732.

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, ser. Springer series in statistics. Springer, 2009, ISBN: 9780387848587. [Online]. Available: http://statweb.stanford.edu/~tibs/ElemStatLearn/.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ArXiv e-prints*, 15 pp., 15 pp. Jan. 2017. arXiv: 1412.6980 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," eng, *Proceedings (ieee International Conference on Computer Vision)*, vol. 11-18-, pp. 1026–1034, 2015, ISSN: 15505499, 23807504. DOI: 10.1109/ICCV.2015.123. [Online]. Available: https://arxiv.org/abs/1502.01852.

[16] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research*, vol. 9, pp. 249–256, 2010, ISSN: 15337928, 15324435. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *32nd International Conference on Machine Learning, Icml 2015*, vol. 1, pp. 448–456, 2015. [Online]. Available: https://arxiv.org/abs/1502.03167.

[18] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *ArXiv e-prints*, 14 pp., 14 pp. Jul. 2016. arXiv: 1607.06450 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1607.06450.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the Ieee Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-, pp. 770–778, 2016, ISSN: 2332564x, 10636919. [Online]. Available: https://arxiv.org/abs/1512.03385.

[20] A. Graves, *Supervised sequence labelling with recurrent neural networks*, ser. Studies in Computational Intelligence. Springer, 2012, ISBN: 9783642247972.

[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, no. January, pp. 3104–3112, 2014, ISSN: 10495258. [Online]. Available: https://arxiv.org/abs/1409.3215.

[22] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," eng, *Advances in Neural Information Processing Systems*, vol. 2015, pp. 1171–1179, 2015, ISSN: 10495258.

[23] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *ArXiv e-prints*, 18 pp., 18 pp. Mar. 2017. arXiv: 1703.10593 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1703.10593.

[24] O. Bojar, R. Chatterjee, C. Federmann, Y. Graham, B. Haddow, M. Huck, A. J. Yepes, P. Koehn, V. Logacheva, C. Monz, M. Negri, A. Névéol, M. Neves, M. Popel, M. Post, R. Rubino, C. Scarton, L. Specia, M. Turchi, K. Verspoor, and M. Zampieri, "Findings of the 2016 conference on machine translation," eng, 2016. [Online]. Available: http://homepages.inf.ed.ac.uk/pkoehn/publications/europarl-mtsummit05.pdf.

[25] P. Koehn, "Europarl: A parallel corpus for statistical machine translation," eng, 2011. DOI: 10.1.1.126.7716. [Online]. Available: http://homepages.inf.ed.ac.uk/pkoehn/publications/europarl-mtsummit05.pdf.

[26] S. Zheng, Q. Meng, T. Wang, N. Y. Wei Chen, Z.-M. Ma, and T.-Y. Liu, "Asynchronous stochastic gradient descent with delay compensation," *ArXiv e-prints*, 20 pp., 20 pp. Jun. 2017. arXiv: 1609.08326 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1609.08326.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: http://tensorflow.org/.

[28] H. Wickham, *Ggplot2: Elegant graphics for data analysis*. Springer-Verlag New York, 2009, ISBN: 978-0-387-98140-6. [Online]. Available: http://ggplot2.org.

[29] A. R. Johansen, J. M. Hansen, E. K. Obeid, C. K. Sønderby, and O. Winther, "Neural machine translation with characters and hierarchical encoding," *ArXiv e-prints*, 8 pp., 8 pp. Oct. 2016. arXiv: 1610.06550 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1610.06550.

[30] Google. (2017). Google developers blog: Xla - tensorflow, compiled, [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html (visited on 06/18/2017).

[31] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *ArXiv e-prints*, 8 pp., 8 pp. Jun. 2017. arXiv: 1706.02515 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1706.02515.

[32] J. Richard C. Bradley, "Central limit-theorms under weak dependence," eng, *Journal of Multivariate Analysis*, vol. 11, no. 1, pp. 1–16, 1981, ISSN: 10957243, 0047259x. DOI: 10.1016/0047-259X(81)90128-7.

[33] Nvidia. (2017). Nvidia cudnn | nvidia developer, [Online]. Available: https://developer.nvidia.com/cudnn (visited on 06/19/2017).

[34] M. T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," eng, *Conference Proceedings - Emnlp 2015: Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, 2015.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *ArXiv e-prints*, 8 pp., 8 pp. Jun. 2017. arXiv: 1706.03762 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[36] Google. (2017). Research blog: Accelerating deep learning research with the tensor2tensor library, [Online]. Available: https://research.googleblog.com/2017/06/accelerating-deep-learning-research.html (visited on 06/21/2017).

{2.7182818284}ϑφερτυθιοπσδφγηξκλ