

Deep Learning for Anomaly Detection in Linux System Log

Minttu Susanna Mäkinen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 27.5.2019

Thesis supervisor:

Professor of Practice Alexander Ilin

Thesis advisor:

M.Sc. (Tech.) Ossi Koivisto

Author: Minttu Susanna Mäkinen

Title: Deep Learning for Anomaly Detection in Linux System Log

Date: 27.5.2019

Language: English

Number of pages: 7+54

Master's Programme in Computer, Communication and Information Sciences

Major: Computer Science

Supervisor: Professor of Practice Alexander Ilin

Advisor: M.Sc. (Tech.) Ossi Koivistoinen

In software development and testing, reading the system logs to find causes for errors is a common activity. When developing large, complex and concurrent systems such as Linux distributions, the amount of log files can grow very large and finding the relevant log entries is laborious.

A deep learning based model, DeepLog, has been previously proposed for automating system log anomaly detection. The DeepLog method begins with tokenizing the log entries and then feeds the tokenized entries to an LSTM neural network. This thesis compares alternatives for LSTM network structure for log anomaly detection. LSTM is compared to three other networks: Gated Recurrent Unit (GRU), Temporal Convolutional Network (TCN) and Transformer. The results of the comparison show that Transformer performs clearly the best, reaching accuracy of 98.9 % on the test data.

Keywords: machine learning, deep learning, LSTM, GRU, TCN, Transformer

Tekijä: Minttu Susanna Mäkinen

Työn nimi: Poikkeavuuksien löytäminen Linux System Logista
syväoppimisen avulla

Päivämäärä: 27.5.2019

Kieli: Englanti

Sivumäärä: 7+54

Master's Programme in Computer, Communication and Information Sciences

Pääaine: Computer Science

Työn valvoja: Professor of Practice Alexander Ilin

Työn ohjaaja: DI Ossi Koivistoinen

Tietojärjestelmän lokit tarjoavat arvokasta tietoa ohjelmistovirheiden löytämiseksi sovelluskehityksessä ja testauksessa. Lokien määrä voi kuitenkin kasvaa erittäin suureksi monimutkaisten ja rinnakkain ajavien ohjelmistojen kuten Linux-jakelujen kehityksessä, ja oleellisten rivien löytäminen voi olla hyvin vaikeaa. Tämän ongelman ratkaisemiseksi on kehitetty DeepLog syväoppimismalli. DeepLog hyödyntää LSTM-neuroverkkoa poikkeavuuksien löytämiseksi lokeista.

Tämä diplomityö vertailee vaihtoehtoja LSTM-verkolle poikkeavuuksien löytämisessä. LSTM-verkkoa verrataan kolmeen muuhun verkkoon: GRU:hun, TCN:ään ja Transformeriin. Kokeiden tulokset näyttävät, että Transformer suoriutuu tehtävästä selvästi parhaiten, saavuttaen 98.9 % tarkkuuden.

Avainsanat: koneoppiminen, syväoppiminen, LSTM, GRU, TCN, Transformer

Preface

I want to thank my thesis advisor Ossi Koivistoinen, co-worker Severi Rissanen and thesis supervisor Alexander Ilin. I also want to thank my friends and family for supporting me throughout the writing process.

Espoo, 27.5.2019

Minttu Susanna Mäkinen

Contents

Abstract	ii
Abstract (in Finnish)	iii
Preface	iv
Contents	v
Operators and abbreviations	vii
1 Introduction	1
2 Machine Learning	3
2.1 Supervised and unsupervised learning	3
2.2 Artificial Neural Networks	3
2.3 Gradient Descent	4
2.4 Residual connections	5
2.5 Word embedding	6
3 Recurrent Neural Networks	8
3.1 Classic model	8
3.2 Long Short-Term Memory	9
3.3 Gated Recurrent Unit	11
3.4 Encoder-Decoder and Sequence to Sequence	12
4 Convolutional Neural Networks	13
4.1 Convolution and max pooling	13
4.2 Temporal Convolutional Network	14
5 Transformer	16
5.1 Attention and self-attention	16
5.2 Transformer architecture	16
6 Preprocessing and datasets	19
6.1 Spell	19
6.2 Datasets	21
7 Methods	23
7.1 Raw data	23
7.2 Choosing the models	23
7.3 Preprocessing	24
7.4 Networks	25
7.5 Hyperparameter tuning	26
7.5.1 LSTM and GRU	28
7.5.2 TCN	30

7.5.3 Transformer	32
8 Results and discussion	35
8.1 Preprocessing	35
8.2 Networks	36
9 Summary	40
A GRU hyperparameter tuning graphs	45
B LSTM hyperparameter tuning graphs	47
C TCN hyperparameter tuning graphs	49
D Transformer hyperparameter tuning graphs	51

Operators and abbreviations

Operators

$\sum_{i=1}^N$	summation over N with i as the variable
$\prod_{i=1}^N$	product over N with i as the variable
$\frac{\partial}{\partial A}$	partial derivative with respect to variable A
$ A $	absolute value of A
ΔA	$ A_2 - A_1 $ (also known as change of A)
$A \circ B$	Hadamard product of A and B (also known as pointwise product)
$p(A B)$	conditional probability of A when B is known
$p(A B_1, \dots, B_N)$	chained conditional probability

Abbreviations

acc	accuracy
ANN	artificial neural network
CE	cross-entropy
CNN	convolutional neural network
CPU	central processing unit
GPU	graphics processing unit
GRU	gated recurrent unit
iff	if and only if
KL divergence	Kullback-Leibler divergence
LSTM	long-short term memory
ML	machine learning
N/A	not available
NLP	natural language processing
ppl	perplexity
ReLU	rectified linear unit
RNN	recurrent neural network
SGD	stochastic gradient descent

1 Introduction

As processes become automated, they produce an increasing amount of logs. For example, within the first hour of its installation and operation, Linux can produce 600,000 rows of log data, as is the case in the systems used to gather data for this thesis. This first hour of operation is particularly important because the log produced during this time can be used to monitor problems in new system installations, which is useful in debugging new Linux distributions still under development.

Previous research on log anomaly detection by Du et al. [12] attempts to solve the problem using sequential learning. Their network is trained to predict the type of the next log row based on the sequence of log row types before it. A row is considered an anomaly if the probability for it is under a predefined threshold. This thesis uses top-5 accuracy to determine whether a row is an anomaly. The network used by Du et al. is an LSTM [19], which was published in 1997. Moreover, Du et al. do not report using any of its variants, such as LSTM with forget gates [14] or GRU [9].

This thesis uses the following networks: Long Short-Term Memory (LSTM) [19], Gated Recurrent Unit (GRU) [9], Transformer [49] and Temporal Convolutional Network (TCN) [5]. LSTM is the baseline to allow our results to be comparable to Du et al. [12], GRU is a streamlined variant of LSTM [9], which may help to reduce the training cost. Transformer is a new approach to sequential learning that has proven to be both accurate and efficient [49]. TCN is a network that uses convolutions, which have proven to be extremely effective in image recognition [26, 53, 47, 18, 21], and applies them to sequential learning. These four networks cover a large variety of different approaches that have been successful in other sequential learning tasks.

The aim of this thesis is to determine a working and efficient solution to detecting anomalies with highly varied logs such as those created during the Linux system installation and early operation. Logs of this type are likely to be more challenging to learn due to their higher variance than logs that are produced during continuous operation of a system. This thesis provides a better performing alternative to LSTM structure used by Du et al. [12], by experimenting with different networks that have been effective on other sequential problems. Additionally, this thesis compares these methods to each other and provides recommendations on using them in similar tasks.

The scope of this thesis is limited to sequential learning. In other words, this thesis focuses only on predicting the following row based on the preceding rows and classifying anything outside the first five predictions as an anomaly. It is also possible to detect anomalies from the contents of the variables that appear within the rows; however, this is outside the scope of this thesis.

The rest of this thesis is organized as follows. Chapter 2 introduces the basic elements in literature that are in common with all the networks tested in this thesis. Chapter 3 presents recurrent neural networks that includes two of the networks experimented on in this thesis. Chapter 4 describes convolutions and the network that combines them with sequential learning. Chapter 5 introduces the fourth network, Transformer, and attention mechanisms it uses to replace more traditional network structures. Chapter 6 discusses the literature techniques that were used to preprocess

the data and transform it into datasets. Chapter 7 describes how the techniques from the literature were applied to the data in practice. Chapter 8 presents the results and relates them back to the literature. Chapter 9 is a short summary of the thesis. The appendices section contain the hyperparameter tuning graphs for each of the networks.

2 Machine Learning

In order to understand ANNs, it is necessary to understand the broader concept of machine learning (ML). ML refers to a range of techniques that aim to draw generalized conclusions from the given data using statistical models and algorithms [15, p. 3]. The amount of data involved is typically large and therefore the calculations are often computationally heavy, making them unviable to be calculated manually, hence the name.

This section describes the common elements that are used in all the networks tested in this thesis. First, it describes the meaning of the task being unsupervised. Following is the explanation of Artificial Neural Networks (ANN) that is the base for the networks that this thesis tests. Then the methods that these networks use to learn new information are discussed. The final subsection describes the method that is used to represent the data to the networks efficiently to make the learning easier.

2.1 Supervised and unsupervised learning

There are several different types of machine learning, but this thesis is most concerned with unsupervised learning. The opposite of unsupervised learning is supervised learning, which refers to the algorithm learning the dependencies of the data based on a set of labeled data points [29]. Therefore, unsupervised learning refers to the case where the data is unlabeled and the algorithm has to learn its structure independently without a preconceived definition [29].

The goal of this thesis is to find anomalies. However, there are a various different types of anomalies and comprehensively representing all of them in our data would be difficult. Thus, the algorithm used in this thesis learns the structure and dependencies from the log data of successful builds. Once the network has been trained in this manner, it can recognize whether the new data it is given appears unusual, even without an exact definition of an anomaly.

2.2 Artificial Neural Networks

All of the networks used in this thesis are ANNs. ANNs are a subsection of machine learning that represents statistical models as a network inspired by neuroscience [29]. This network consists of artificial neurons, such as multilayer perceptrons [15, p. 5]. A singular perceptron typically consists of input and weights that are then combined in a weighted sum and given to the activation function that determines the output. Formally,

$$y = f(Wx + b) \tag{1}$$

where x is the inputs, n is the number of inputs, w is the weights and b is the bias term [24]. The result is then given to an activation function $f(x)$. Examples of activation functions include Rectified Linear Unit (ReLU), Softmax and tanh, all of which the networks in this thesis use at various points.

The artificial neurons can then be arranged both next to each other (network width) and after each other (network depth) to form a network. The term 'deep

neural network' refers to having multiple layers of cells after each other [15, p. 2]. The first layer is the input layer that delivers the input to the rest of the network. After this there is a number of layers known as hidden layers [15, p. 165]. The final layer is an output layer that contains the final results of the network [15, p. 165]. In classification tasks, the activation function of the output layer is a Softmax that maps the results into probabilities between 0 and 1.

The advantage of using neural networks is their ability to learn the dependencies between the inputs and the outputs. This learning is implemented by making changes to the weights, in the above equation. Changing the weights directly affects the calculation and allows adjusting the results closer to the desired output [29].

In order to know how well the network is predicting the log data and what parts of it need to be changed to reach the expected result, a loss function is calculated. This is a function that compares the output the network generates to the expected output. The loss function that all the networks in this thesis use is Cross-Entropy (CE). To understand CE, it is necessary to understand entropy first.

Entropy is the amount of information a single sample contains on average. Information in this context is defined in bits. A single bit is the amount of information transmitted when choosing between two equally likely choices. The scale is logarithmic of base two, and thus a choice between four equally likely possibilities yields two bits and a choice between eight is three bits. This can also be seen in the terms of probabilities. While the probability between two possibilities is 0.5, which is relatively high, the probability of any individual choice when there are eight equally likely possibilities is only 0.125. In other words, the information in bits is a negative base two logarithm of the probability [15, p. 71]. Since there are often many different possibilities, it would be useful to know the expected amount of information provided by a single sample. This is known as entropy. Formally,

$$H(x) = -\mathbb{E}_{x \sim P}[\log P(x)] \quad (2)$$

where P is the given probability distribution [15, p. 72].

A loss function represents the difference between two distributions; the predictions and the target values. If the information contained in them is represented by entropy, it is possible to calculate the difference between the distributions as follows,

$$D_{KL}(P||Q) = \mathbb{E}_{x \sim P}[\log P(x) - \log Q(x)] \quad (3)$$

where P and Q are the two distributions. This is known as Kullback-Leibler (KL) divergence [15, p. 73]. Because the property of loss function that machine learning is concerned with is finding the minimum with respect to Q , this equation can be simplified as long as that property stays the same. This finally leads to the loss function used by the networks in this thesis, cross-entropy:

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x) \quad (4)$$

2.3 Gradient Descent

Once the distance to the ideal result is known, the loss function should be minimized, so that the prediction result will be closer to the expected one. The algorithms that

finding the new weights is called an optimizer. One of the most well known optimizers is Stochastic Gradient Descent (SDG) [15, p. 149], which is also used by one of the networks in this thesis. The principle is that the weights should be updated based on how much they contributed to the error between the received and the expected outputs. The amount of this adjustment is controlled by a parameter known as learning rate. Partial derivatives over the the error and weight can be used to reason the direction in which the weights should be moved. This is known as a gradient. The new weights can be calculated as follows:

$$\theta_{k+1} = \theta_k + \eta \left(-\frac{\partial C}{\partial \theta_k} \right) \quad (5)$$

where θ is one element of W or b from Equation 1. C is the cost function and η is the learning rate [39, p. 16 - 22]. If the learning rate is very small, the learning is slow and the gradient may get stuck to a local minimum. However, if the learning rate is too large the gradient cannot take small enough steps to reach a local minimum efficiently and the optimization procedure may not converge. Another drawback is that gradient descent only finds the local minimum, which is not guaranteed to also be the global minimum.

Although SGD is not the only optimizing algorithm, the majority of the optimizers, including Adam that the other three networks in this thesis use are still based on gradient [25] and thus share the same core strengths and weaknesses.

Calculating the gradients can be a complicated, especially if the network is deep. Furthermore, because C consists of multiple terms and computing them all can take a long time. Therefore, it is preferable to calculate an estimate of the gradient using a small subset of the training set to form a mini-batch [39, p. 22]. The process of picking mini-batches and calculating gradients is repeated until no data samples are left unprocessed. This is referred as finishing an epoch [39, p. 23]. This process is typically repeated multiple times during one training. For example, in this thesis training consists of 10 epochs.

2.4 Residual connections

A groundbreaking work by He et al. [18] introduces the concept of residual connections. These paths are shortcuts that allow skipping layers, creating a reference point for the learned weights. It is speculated that this leads to the easier optimization of deep networks [18], resulting in a higher accuracy. This idea can be expressed mathematically

$$y = \mathcal{F}(x) + x \quad (6)$$

where y stands for output, x is input and $\mathcal{F}(x)$ is a function that consists of the layers of the network [18]. Although in the original paper the function consisted of two convolutional layers, the idea can be generalized to other types of functions as well. According to He et al. this provides the network a point of reference that helps to limit the complexity of the network allowing easier optimization. Utilizing these connections allowed a Residual Network (ResNet) to win the ImageNet challenge of 2015 with a network that was 152 layers deep [18].

Although residual connections are perhaps more commonly associated with convolutional neural networks rather than RNNs, due to their origins in image recognition, it is still a very useful technique in the both networks.

2.5 Word embedding

Linux System Log can be seen from the perspective of Natural Language Processing (NLP) as a story that describes the state of the machine that produced it. In this story a sequence of rows can be viewed as a sentence in which the each row is a word. This viewpoint allows using NLP techniques to efficiently represent the data to the network in a numerical form. This is necessary because the networks cannot read the letters directly.

One way of giving words this numerical representation is to make a vector that has all the possible words represented in its length. The word that the vector represents is marked as a 1 while all the others are 0s. This way each word can be represented as a row in a matrix that represents the whole input sequence. This technique is known as one-hot encoding [33]. Although one-hot encoding does transform words and sentences into a representation that's readable for a computer, the resulting matrix will be very sparse if the vocabulary is large. Additionally, the order in which the words are assigned to the dimensions of the vector is somewhat arbitrary, as proximity within the vector does not correlate with the similarity of their meaning [33].

In order to make the vector distances portray the syntactic and semantic similarities of the words, Mikolov et al. [37] developed two word embedding models. These models project words into a multi-dimensional space where points correlate with the meaning of the words. This meaning is gradually learned from the context they appear within the given data.

The first suggested model, Continuous Bag-of-Words (CBOW) is similar to a feedforward network [37]. CBOW is given the input words in one-hot encoding and it attempts to predict the target word based on C words around it. These words are also known as context. The architecture of the network consists of nothing but the input, a single N-dimensional hidden layer with no activation function and an output layer. Once the training is finished, the weights learned by the hidden layer act as the vector representation of the word [37].

The second suggested model is known as Skip-Gram. Rather than attempting to predict the target based on the context, it does the reverse and attempts to predict the context based on the target [37]. For each C slots in the target word's context, Skip-Gram outputs a probability predicting the context word. Once again, the weights of the trained hidden layer act as the representation of the target word [37].

In a follow-up paper Mikolov et al. [38] introduced further improvements to Skip-Gram. One of these improvements was to downscale the importance of very common words by adding a random chance to discard input word w_i

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \quad (7)$$

where $f(w_i)$ is the frequency of the word and t is a chosen threshold, typically around 10^{-5} [38].

The second introduced improvement was Negative Sampling. It reduces the amount of weights that need to be adjusted for each word that Skip-Gram is trained with. In the original Skip-Gram each word would increase its own weight and reduce all the others. However, this operation can lead to significant amount of computational cost as the size of the weight matrix is often large [38]. This is why rather than updating every single weight, Mikolov suggests choosing randomly a desired amount of negative samples and updating only them alongside the positive sample. The more common a word is, the more likely it is to get selected as a negative sample.

Implementing these two techniques has proven to lead to a better representation of uncommon words and considerably faster training of the embedding [38]. Mikolov et al. estimated this speedup to be around 2x - 10x.

3 Recurrent Neural Networks

There are certain tasks in which the order of the inputs is extremely important, such as natural language processing (NLP). For example, the meaning of the sentence "A person is walking a dog" is significantly different from "A dog is walking a person", even though they use the same words. Recurrent neural networks (RNN) are a way of modelling situations like these, where the order, also known as sequence, of the events is important [15, p. 367].

3.1 Classic model

One way of modeling a sequence, for example, a sentence, is to take the first word and then giving probabilities to what word is likely to follow it. The probability of the word after that is influenced both by the previous word and the one before it. Thus, the sequence can be modeled as a conditional probability. The formal definition can be derived from the chain rule [15, p. 381].

$$p(x_1, \dots, x_N) = \prod_{i=1}^N p(x_i | x_1, \dots, x_N) \quad (8)$$

Recurrent neural networks (RNN) use the same idea. Each cell gets the result of the previous computation and a new variable (in our case, a word) as input and its output gets fed to the next cell. Receiving the previous output allows the network to retain 'memory' of what came before the current input and makes distinguishing order possible.

In general, RNN is a style of network that strings together cells in a recurrent fashion with hidden connections between them and produces an output at each time step [15, p. 372]. The term itself does not dictate the exact way the output should be produced from the inputs. This leads to there being several variants of RNNs.

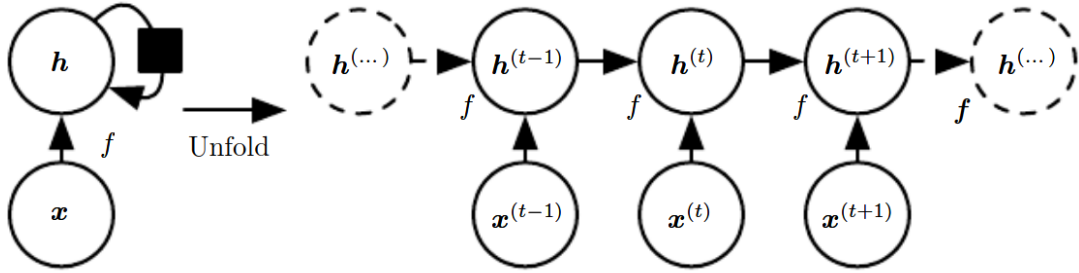
A common variant uses equation of the following form,

$$h_t = f(h_{t-1}, x_t; \theta) \quad (9)$$

where h is the hidden state, x is the input and θ is any additional parameters [15, p. 370]. This equation can be unfolded into a sequence where each of the cells takes the previous hidden state and an input to produce the hidden state for the next cell. This structure is shown in Figure 1.

According to Equation 3, to update the weights that are in the earlier layers of the network multiple partial derivatives have to be calculated that then get multiplied by each other and the learning rate. If these gradients contain numbers that are under 1, the gradient keeps getting smaller and smaller on top of being multiplied by the learning rate, which already tends to be a very small number. This may lead to a situation where some of the weights in the earlier layers of a network update so minimally that it takes unreasonably long for them to converge at the loss minimum [6]. This is known as vanishing gradient. On the other hand, if the partial derivatives return very high numbers, the gradient gets multiplied every time and and become

Figure 1: A recurrent network with no outputs. The arrows indicate change in time step [15, p. 370]



extremely large in the upper layers, leading to numerical instability and inability to converge. This is known as exploding gradient. Vanishing and exploding gradients are especially big problems in RNNs because its sequential nature leads to a chain of gradient calculations.

Instead of the produced output, it is also possible to use a teacher signal as the input for the next unit. This technique is known as teacher forcing [50, p. 274 - 275]. In the case of this thesis, teacher signal is the target output of the previous LSTM cell. This thesis uses teacher forcing with two of the experimented networks. The rate in which teacher forcing occurs is Teacher Forcing Ratio (TFR) and is one of the hyperparameters adjusted with these networks.

3.2 Long Short-Term Memory

One approach to mitigate the problem of vanishing and exploding gradients is to use a gated structure to decide what information worth keeping and what is not. The network that introduced this concept is Long Short-Term Memory [19]. Gates in LSTM consist of learned weights and a function, similar to activation functions, that decides what to remember or forget and then applying this by using a matrix operation. A gate can be likened to a skip connection, like residual connections introduced in Chapter 2.4. Like residual connections, LSTMs implement a path that is fairly straightforward and another that contains adjustable weights.

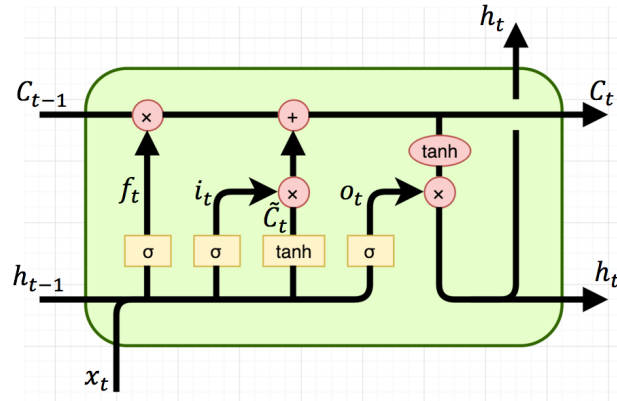
LSTM was originally introduced by Hochreiter et al. [19] but the architecture that is discussed in this thesis also includes the changes made by Gers et al. [14]. Formally,

$$\begin{aligned}
 i &= \sigma(x_t U^i + s_{t-1} W^i) \\
 f &= \sigma(x_t U^f + s_{t-1} W^f) \\
 o &= \sigma(x_t U^o + s_{t-1} W^o) \\
 g &= \tanh(x_t U^g + s_{t-1} W^g) \\
 c_t &= c_{t-1} \circ f + g \circ i \\
 s_t &= \tanh(c_t) \circ o
 \end{aligned} \tag{10}$$

where x is the input and s is the hidden state. U and W are their respective weights

that learn to recognize what information is important and what is not. Their indexes indicate what gate they belong to. c is the cell state and t is the number of the current LSTM cell. \circ represents pointwise product, also known as Hadamard product [19, 14]. The equations are described graphically in Figure 2.

Figure 2: Structure of a single LSTM cell [42]



It can be seen that there are three input variables; a cell state, a hidden state that was received from the previous cell and the next part of the sequence to be processed. The first thing that should be decided is what information is unimportant and should be forgotten. In other words, there should be a multiplication with a weight matrix. The result is then passed to a sigmoid function [14].

A sigmoid function is used to interpolate between two states: the gate being open and the gate being closed. A commonly [33] used sigmoid function is,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (11)$$

When the combined variable has been squished, the cell state is multiplied pointwise with it and the result is passed forward as the new cell state [14]. This process represents removing the unnecessary information from the cell state or 'forgetting'. It is therefore known as the forget gate of LSTM. An alternative interpretation of this is that it decides what information to keep and thus it's also sometimes referred to as a keep gate. This is simply a matter of viewpoint.

The next step is to decide which new values the cell state should remember. This part is known as an input gate, because it is combined with what would be the input in a normal RNN. This variable is represented by putting the combination of the actual input and the hidden state through a \tanh function. The results from these functions are multiplied pointwise and pointwise addition is used to combine the information with the cell state [19]. This represents the cell state learning new information.

So far the cell state has forgotten the unnecessary and learned the new information. In other words, it is ready to be passed for the next cell to begin the same process once again. However, the new hidden state still needs to be calculated, so it can be given as an input to the next LSTM cell. This is done similar to the earlier gates,

but instead of using the concatenated hidden state as the metaphorical RNN input, the newly calculated cell state is used instead. Just as before this is passed to a tanh function and multiplied pointwise with a gate [19]. Like the other gates, this consists of the concatenation of the hidden state and the actual input that has been put through a sigmoid function. This is the last case of the LSTM cell and it is therefore known as an output gate. The result of the pointwise multiplication will become the new hidden state that is given to the next cell [19].

The basic LSTM network consists of just the input layer, a hidden LSTM layer and the output layer. However, the idea can be extended to a deep LSTM [16], where multiple hidden LSTM layers are stacked on top of each other. This is done by taking the output of the each LSTM cell and using them as inputs for the cells in their respective positions in the next LSTM layer.

Although LSTM models are quite old, being introduced in 1997 by Hochreiter et al. [19] and later improved with the addition of the forget gate in 1999 [14], they have remained state of the art until recently with relatively little change [17, 36].

One way of understanding LSTM structure is their parallel with residual connections. Highway networks [45] are essentially residual networks with weights known as 'gates' added to the shortcut connections [45, 18]. This is the same approach as taken by LSTMs, where the hidden state goes through various functions representing the residual block and the cell state acts as the residual connection with weight multiplication.

3.3 Gated Recurrent Unit

Although LSTM proved successful and because the new state of the art method after its introduction, it is apparent from the equations above that it's somewhat complicated to compute. To solve this, Gated Recurrent Unit (GRU) was introduced [9] as a variant of LSTM to be a more streamlined version of the same basic idea. Instead of having three separate gates, GRU combined these into just two; the reset gate and the update gate. As can be seen in Figure 3, the cell state variable has been removed from the structure and thus all the information is now contained in the hidden state h and the input parameter of x [9].

The reset gate works very similar to the forget gate in LSTM, deciding what information should be forgotten. Meanwhile the intuition behind the update gate is to choose what parts of the new information gained from the input is worth including and which should be scrapped.

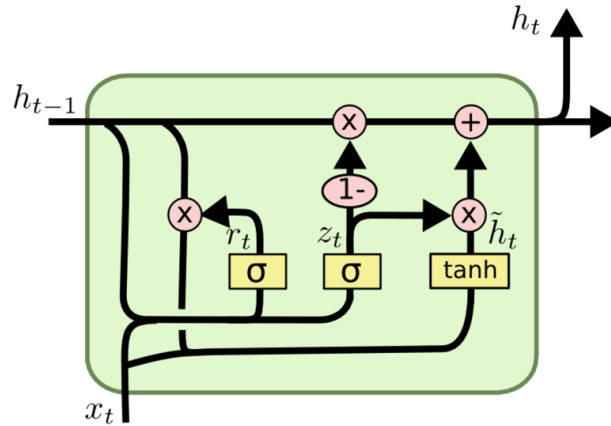
The similarity between LSTM and GRU can be seen in their equations. GRU's equation is as follows,

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1}) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1}) \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1})) \end{aligned} \tag{12}$$

where the instances of W and U are weights that will be learned, separated from each other by their index [9]. It can be seen that the removal of the additional variable

simplified the calculation process, which is likely the reason for increased speed in comparison to LSTM that GRU is known for.

Figure 3: Structure of a single GRU cell [42]



3.4 Encoder-Decoder and Sequence to Sequence

The task of transforming a sequence into another sequence can be further divided into two parts. The first of these is discovering the most effective way of describing the given information as a hidden state. The second part is using this description to generate the desired sequence.

The first part of this network structure is known as the encoder. The encoder receives a variable-length sequence and transforms it into a hidden state that acts as a summary of the entire input sequence [9].

Decoder is the part of the RNN that is focused on generating output by predicting the next word. Each cell in the network receives the previous hidden state as well as the summary of the meaning provided by the encoder as their input. Thus, rather than attempting to translate the original sentence directly, the decoder attempts to find the output that best describes the meaning that's portrayed in the internal representation and turns this into a variable-length sequence [9].

This approach was later honed into a Sequence to Sequence (seq2seq) model by Sutskever et al. [46] by changing the hidden units into a deep LSTM. Additionally, they found that reversing the order of the input sequence significantly improved the results. This was likely because the reverse order introduced short-term dependencies that made the optimization of the network easier [46].

4 Convolutional Neural Networks

Convolutional Neural Network (CNN) is a type network that is particularly popular in image recognition. For example, ResNet [18] can be classified as a CNN. Because of its prevalence in image recognition tasks, this section demonstrates its functionality with an image recognition example. However, one of the networks used in the experiments of this thesis, TCN, can also be classified as a CNN [5].

This section first discusses the defining feature of CNN, a convolution. After that Chapter 4.2 describes how TCN applies this feature to sequential learning, such as the task in this thesis.

4.1 Convolution and max pooling

The intuition behind convolutions is to first split the image into smaller parts and recognize its details, which are then passed on to the next layer that represents a higher level of abstraction.

In practice, CNNs use a small box of weights known as a kernel or a filter to loop over the image. The filter performs a matrix multiplication between its own weights and the part of the image currently being processed on each step of the loop. Results of these operations are combined into one matrix that is the output of the convolution [28]. There are various ways of controlling the way the filter moves over the input matrix. For example, the sides of the matrix can be padded or the stride, ie. the distance the filter moves between the steps of the loop, can be changed. Although this example discusses matrices, the basic idea can be generalized into multiple dimensions. For example, RGB images have three color channels and occasionally also an alpha channel to portray transparency, in which case the filter must also have these additional dimensions [26].

There may be several convolution layers in a row and their output is usually passed to max pooling through an activation function. In max pooling the matrix or tensor is divided into boxes that each only return their biggest value, which are then combined into an output matrix or tensor [15]. This is typically done to reduce the size of the data for lighter computational cost and reduced noise.

Because filters are usually significantly smaller than the image, several convolutional layers need to be added on top of each other to combine these details into a representation of the entire image. This leads to the size of the processed image becoming rigid, creating networks that specialize only in certain size. Although convolutions are not a new idea and CNNs were used as early as 1989 [30], they were held back by the limitations of the hardware.

Usually, there is a linear layer, also known as fully connected layer after the pooling [29]. A linear layer refers to the simple matrix multiplication of the entire layer with its weights and the addition of a potential bias term to the result.

4.2 Temporal Convolutional Network

As mentioned above, CNNs can be generalized to higher dimensions to include channels such as color. However, the same also applies to the opposite direction and applied to sequential learning. Sequences can be thought of as matrices of size $1 \times \text{sequence_length}$ and thus it is possible to apply 1-D convolutions to them. It is noteworthy that using convolutions in 1-D has the same implications in regards to input size as the higher dimensions. In order to cover the entire sequence and then combine these representations into one leads to the rigidity in input and output sizes. For example, the network used in this thesis, introduced by Bai et al. [5] can only map the input to an output of equivalent length. Additionally, as with 2-D, very large input sizes lead to very deep networks unless the filters are large. However, large filters tend to reduce the power of the convolution and lead to lower accuracy.

There are several variations of TCN [48, 13, 27, 5] and although they share the idea of 1D convolutions, the details of the implementation vary. The description here is specifically based on the model by Bai et al. [5] as that is the version used in this thesis. Additionally the abbreviation 'TCN' specifically refers to the one introduced in their paper, which is largely a more streamlined version of WaveNet introduced by van Oord et al [48]. Although Bai et al. are not the first one to use the term TCN in the academia [5, 27], they are the most relevant in the context of this thesis.

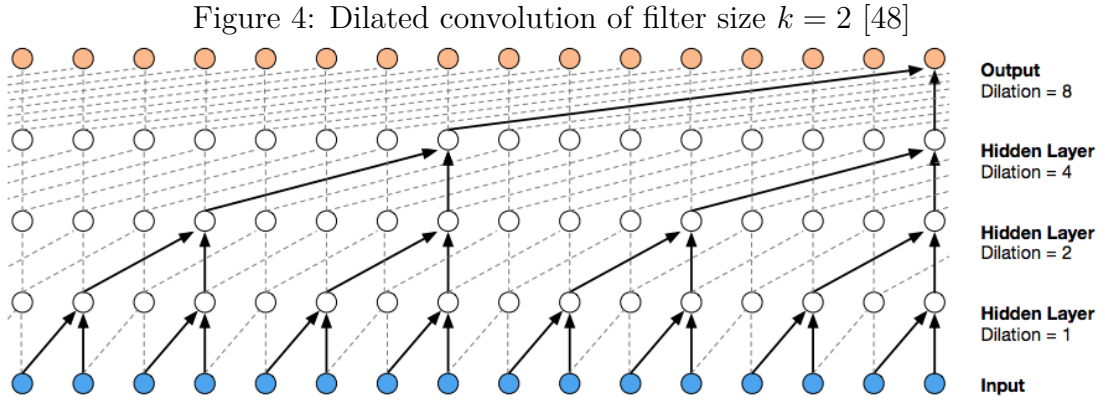
TCN follows two main principles. Firstly, convolutions only process the information that has appeared earlier in the sequence. This prevents the influence of 'future' information from affecting the results. These are known as causal convolutions [48, 5]. Secondly, a sequence of any length will always be mapped into an output sequence of the same length as the input. All the hidden layers will be of input length, with zero padding of length (kernel size - 1) added to maintain it throughout the hidden layers [5].

Sequential learning is based around learning patterns from history. However, this presents a problem to using convolutions, as they generally only account for a linear size input, while the effective history in sequential learning can be much longer than that. Van Oord et al. [48] and Bai et al. [5] both use dilated convolutions to solve this problem. Similar to pooling or striding convolutions, dilation adds a step between each accounted input. The entire sequence can be covered efficiently by starting out with a dilation factor of 1 and then collecting the information that the previous layer combined with a larger dilation factor. Stacking hidden convolutional layers allows the receptive field to grow exponentially [52]. This structure is described in Figure 4.

Dilated convolution operation F in 1-D sequence can be formally expressed as

$$F(s) = (x *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot x_{s-di} \quad (13)$$

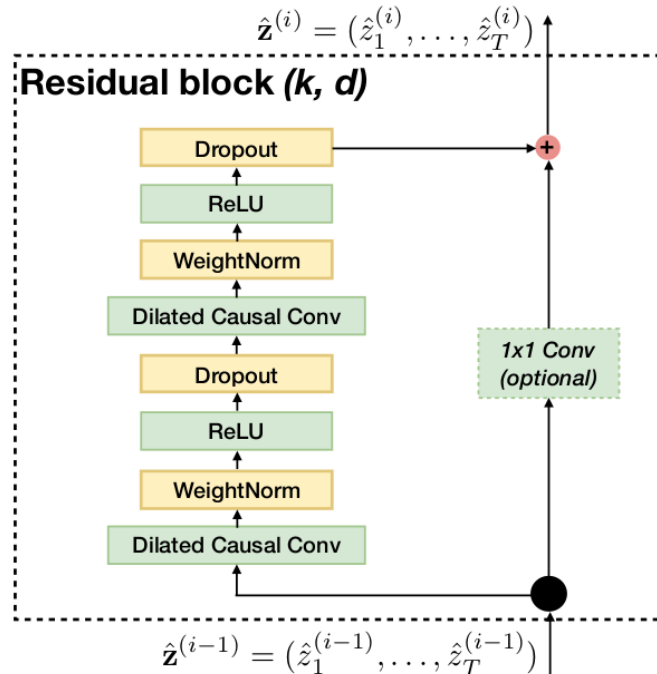
where $x \in \mathbb{R}^n$ is the input d is the dilation factor, $f : \{0, \dots, k-1\} \rightarrow \mathbb{R}$, k is the filter size, and $s - di$ accounts for the direction of the past [5]. In TCN the dilation factor d is determined from $d = O(2^i)$, where i is the network depth. By increasing the filter size k and the dilation factor d , the length of effective history can be increased



to $(k - 1)d$ [5].

Because long history sizes lead to deep networks, TCN also utilizes residual connections in its structure. Therefore instead of performing dilated causal connections on convolutions, the convolution layer is replaced with a generic residual block. As described by He et al. [18] this consists of two convolutional sublayers after each and a shortcut connection around them. However, TCN removes batch normalization [23] used by ResNet and instead applies weighted normalization [43] before and a spatial dropout [44] after each sublayer. Optionally, the shortcut connection can also be replaced with a 1×1 convolution. Figure 5 shows the structure of a single TCN residual block.

Figure 5: Residual block used in TCN as shown by Bai et al. [5]



5 Transformer

Transformer is one of the networks tested in this thesis. It features a unique structure that replaces recurrency with attention-based mechanisms [49]. Attention is first described in Chapter 5.1 and the following Chapter 5.2 describes the way Transformer uses it in practice to replace the recurrent structures found in the other three networks of this thesis.

5.1 Attention and self-attention

Convolutions limit the focus of the network into one detail at a time and aim to build a coherent picture out of these pieces. This idea of limiting the area of focus is also behind a technique known as attention. Attention refers to a mask over a tensor that learns the importance of its individual values and adjusts their weight accordingly [51]. Attention can be classified to either soft or hard attention based on the weights it assigns to the input. In hard attention, the parts of the input that are deemed useful are assigned to 1 and everything else is 0, while soft attention also allows the values between them [51].

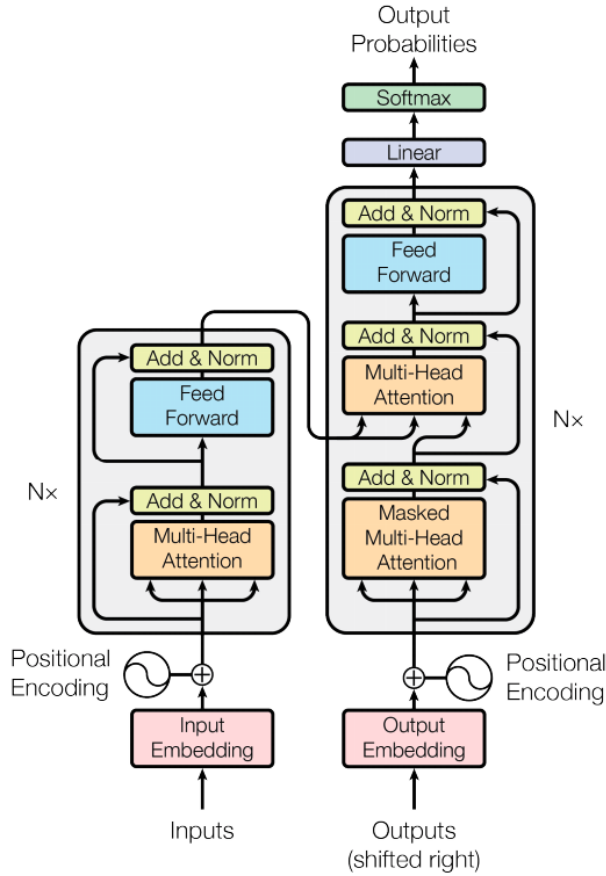
Although attention is used in image recognition and captioning [51], its variant, self-attention has gained popularity in sequential learning as well [8, 40, 41, 32, 49]. Self-attention or intra-attention is a technique that calculates relationships between different parts of a sequence [49]. Like the attention above, self-attention is a way of describing the importance of certain areas in the input. However, instead of applying global attention [34] over the entire subject, self-attention only concerns itself with a single sequence [49]. Self-attention is a general concept, and thus there are multiple ways of implementing it. The most common of these are additive attention [4], dot-product attention and scaled dot-product attention [49] that is introduced in the following subsection.

5.2 Transformer architecture

RNNs have been exceedingly common [19, 14, 16, 46, 9, 33, 11, 17, 36, 35] in sequential learning, with convolutional approaches [13, 27, 5, 48] gaining some popularity as well. However, Transformer [49] is a network which completely seems to sidestep these conventional approaches. While it still shares the encoder-decoder structure with its contemporaries, their functionality is implemented with attention structures alone. This is not the first time attention mechanisms have been involved in sequential learning [8, 40, 41, 32]. However, previously these techniques had been used in conjugation with RNNs, rather than replacing them entirely.

Transformer's encoder consists of a stack of six identical layers that are further divided into two sub-layers each. The first of them is what Vaswani et al. [49] call 'multi-head attention' and the second is a fully connected feed-forward network. There is a residual connection [18] around each sublayer, which is then added to the output and the layer is normalized [31].

Figure 6: Transformer structure as seen in its original paper [49]



As with encoder, decoder is a stack of six identical layers. Each of these consists of three sublayers. The first of them is multi-head attention, where the future sequence has been masked out. This prevents information from the future from influencing Transformer's predictions. Following this, the information from the encoder is combined to the output of the decoder's first sublayer in another multi-head attention module. The final sublayer is a fully connected feedforward network. As with the encoder, there is a residual connection around each of the sublayers, which is then added to their output and the layer is normalized. Outputs of the entire decoder are finally given to a linear layer and a Softmax function maps it into probabilities [49]. The total structure of the network can be seen in Figure 6.

More specifically, the attention operation that Transformer uses is known as scaled dot-product attention [49]. The inputs are d_k -dimensional queries and keys and d_v dimensional values. In practice, multiple instances of these variables are packed together to form the matrices Q , K and V . With these variables the scaled dot-product attention is

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (14)$$

This equation is otherwise the same as dot-product attention, save for the scaling factor $\sqrt{d_k}$ [49]. In the variant of this operation that is used on the first sublayer of the decoder, the masking of the future is done between scaling, just before Softmax function.

Multi-head attention is a parallelization of these scaled dot-product attention blocks. The three inputs are first projected linearly h times into variables of d_k , d_k and d_v dimensions respectively by different learned projections [49]. The results of this are then given to scaled dot-product attention function resulting in h d_v -dimensional outputs. These are then concated and projected into the final values. Formally

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \quad (15)$$

where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W_i^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ [49].

Vaswani et al. [49] offer three different explanations on why attention mechanisms might improve performance over recurrent and convolutional network. Firstly, attention layers have lower computational complexity than its competitors that allows it to be faster. Secondly, it has relatively few sequential operations and can be parallelized easily. The third explanation is that the length of the longest shortest path in the network is small. This allows easier learning of long-range dependencies. The comparison can be seen in Table 1.

Table 1: Comparison of different network types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of neighborhood in restricted self-attention [49].

Layer type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^a \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

6 Preprocessing and datasets

The following section describes what happens to the data before it goes into the network that is being tested. Chapter 6.1 discusses the method used to transform the data into a shape that the word embedding that was discussed in Chapter 3.2 can understand. Chapter 6.2 explains how the data is divided into datasets, the method in which the datasets are used and the reason for this method.

6.1 Spell

Although logs were likened to natural language in Chapter 2.3, there are still some differences between the two. It is easy to give a word a number that is always repeated when the word appears in a sequence. However, because it is possible for the logs to consist of multiple parts it is hard to tell when the rows should be considered to be the same. For example, if the row being processed is "Temperature (41C) exceeds warning threshold" [11] it can be seen as consisting of two parts. One of the parts is the base message that warns about the temperature and "(41C)" is the parameter associated with it that changes with each entry. In order to assign a repeatable number to this log row, it is necessary to consider which part of the message is the base or "key" that is repeated in the data and which part of the message is the parameter that changes between the entries. Separating the two would allow assigning a number to each key and therefore representing the log as a string of numbers that can be then fed to the word embedding that finds the most effective numerical representation for the data. Because the scope of this thesis is limited to learning sequences of these repeatable elements, the parameters of the rows can be ignored.

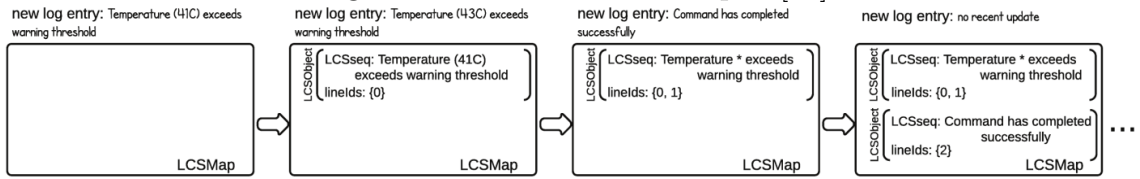
The algorithm used to separate keys and the parameters from the raw log data is Spell [11, 12]. This is an algorithm that uses the Longest Common Subsequence (LCS) as a metric to determine whether a row matches a pre-established key.

The problem of finding LCS is described by Du et al. [11] as follows. Suppose Σ is a universe of letters. Given any sequence $\alpha = \{a_1, a_2, \dots, a_m\}$, such that $a_i \in \Sigma$ for $1 \leq i \leq m$, a subsequence of α is defined as $\alpha = \{a_{x_1}, a_{x_2}, \dots, a_{x_k}\}$, where $\forall x_i, x_i \in \mathbb{Z}^+$, and $1 \leq x_1 < x_2 < \dots < x_k \leq m$. Let $\beta = \{b_1, b_2, \dots, b_n\}$ be another sequence such that $b_j \in \Sigma$ for $1 \leq j \leq n$. A subsequence γ is called a common subsequence of α and β iff it is a subsequence of each. The longest common subsequence (LCS) problem for input sequences α and β is to find longest such γ . Note that $x_{i+1} = x_i + 1$ is not required.

Naive solution to the problem compares the two strings by looping over every word and then comparing its similarity to each of the words in the other string. It is also possible to do the same with individual characters instead of words, however as naive LCS has quite high time complexity at $O(n^2)$, [11] this can be expensive.

Parsing a log starts by taking the first row and comparing it to the list of accumulated keys so far. However, because this is the first row, there is no key that it could be matched to. Therefore there is no match and the line is added to the list of keys.

Figure 7: Basic workflow of Spell [11]



On the second row, the current line is compared to the keys again. This time there is a key, so LCS is used to judge the similarity of the strings. If the LCS between the strings is smaller or equal to the threshold value (in the case of this thesis the threshold is half of the row's length), there will be no match and the second row is also added as a key. Otherwise a match is recorded and any differences between the key and the current line are considered to be the parameters. These parts are then removed from the key and replaced with a placeholder symbol (in our case '*').

Processing the rest of the rows is similar to the second row, however, assuming that the second row was not a match, there are multiple keys that can be attempted to be matched. Therefore the LCS of the current has to be calculated for all of the keys in order to find the largest LCS. As with the second row, if LCS is more than half of the row's total length, it is considered a match. Otherwise the row is added as a new key. This workflow can be seen in Figure 7.

The loop continues until all of the rows have been processed. As is apparent from the amount of loops needed, this is rather computationally heavy. LCS operation alone has the worst case of $O(n^2)$, where compared strings are of n length and if this is done for m rows, the total complexity becomes $O(m \cdot n^2)$ [11]. One way of easing this load is to replace some of the key comparisons with a prefix tree comparison [11].

A prefix tree is built by first taking a key and making each of its tokens a child of the previous token in the key. When a new key is inserted, its token is first compared to the root's children. If they are equal, the next token is compared to the children of the previous token that matched. This continues until there is a divergence. When this happens the token that did not match is added to be the child of the previous token that matched. The rest of the tokens will be added under it as each other's children. If a row reaches all the way to a leaf node of this tree and has no further words, it can be concluded that that the row matches the key that leaf node belongs to and its LCS is the depth of the leaf. If the comparison fails, LCS is used instead.

A prefix tree works well when a lot of the received rows match keys, reducing the amount of the costly LCS comparisons. Time complexity is changed to $O(n + \frac{I+F}{L} \cdot m \cdot n + \frac{F}{L} \cdot m \cdot n^2)$, where F is the number of failed trie searches, L is the number of logs, m is the number of message types and n is the largest log length [11].

6.2 Datasets

The final step before the data processed with Spell can be given to the word embedding is to divide it into sequences and datasets. The division to sequences is fairly self-explanatory; instead of processing the entire log as one, it is split into pieces of chosen length. However, the reasoning for the dataset division is slightly more complicated and has to do with optimizing as well as assessing the results of the networks.

The aim in machine learning is to train the model to recognize the features of the data in order to generalize it. However, because the task given to the network is to minimize the loss within the train data, this will eventually lead to the situation where rather than just fitting the features, the model fits to the training data exactly, provided the model has enough dimensions. This is called overfitting and it leads to the model being unable to make accurate predictions, as rather than measuring whether the new data is similar to the train data it judges whether the new data *is* the train data.

The dataset typically divided in three parts; train, validation and test sets [15, p. 118]. As the name implies train set is the data used to train the network. Loss that is used in weight optimization is calculated from the train set. It is typically the largest of the sets and in this thesis it contains 80 % of all data.

Validation set is used to optimize the hyperparameters. Hyperparameters are settings used to control algorithm's behavior that are not adapted by the learning algorithm itself [15, p. 118]. Different settings produce different versions of the network. By calculating various statistics from the validation set it is possible to judge which of these versions performs the best [15, p. 119]. If the training accuracy keeps increasing between epochs but the accuracy calculated from the validation set starts to decrease, it is possible to conclude there is overfitting. Validation accuracy can show when the overfitting began and which version of the model should be used [15, p. 109].

The opposite of overfitting is underfitting. In underfitting, the model generalizes too much to be useful and does not fit the data. This may be caused by the model being too small to adequately describe the features or having too little training [15, p. 108-109].

During hyperparameter optimization, multiple versions of the network are trained with different hyperparameters. Statistics calculated from the validation set, such as loss or accuracy, are then used to choose the settings that result in the highest performance [15, p. 120].

However, choosing the best epoch or hyperparameters introduces bias to the validation set, which is why it cannot be used to accurately gauge the performance. It is possible that these particular hyperparameters simply happen to fit the validation set particularly well. The final performance of the network is calculated from a separate dataset known as test set [15, p. 109-110]. The purpose of the test set is to portray the network statistics as accurately as possible. This is why the network is not adjusted even if the test set has worse results than the validation set.

As both the validation set and the test set exist for statistics calculation, it is enough for them to be large enough to act as a smaller representation of the entire

data. In this thesis both the validation and the test sets are 10 % of the overall data.

7 Methods

This thesis aims to determine the most optimal method to detect unusual features from Linux System Log by using sequential machine learning models. More specifically, this thesis aims to locate anomalies based on the order in which different types of log rows appear. To achieve this, four different approaches to sequential learning are compared: LSTM [14], GRU [9], TCN [5] and Transformer [49, 22]. Although it is also possible to assess the parameters that appear in the logs [12], this is outside the scope of this thesis.

This section takes a look at the specific context of this thesis. The first section contains the method in which the data was gathered and its amount. Following is the explanation of the reasons why these models in particular were chosen for this task. Chapter 7.3 describes the manner in which Spell was used, as well as introducing the statistics of the preprocessed data. Chapter 7.4 discusses the details of the network implementation. Finally, Chapter 7.5 contains the information related to validation. This includes the initial parameters and the fine tuning of the networks. For further details, validation graphs can be found in Appendices.

7.1 Raw data

Multiple Linux distributions have automatic log collection by 'journald' daemon. The command 'journalctl' returns these logs usually in human-readable format. In order to get the data in a more structured fashion, the data for this thesis has been gathered using 'journalctl -o json' command. This transforms each log row into json with a newline between them.

The logs that are collected from test installations during the development of a certain Linux distribution. Out of these the 10,000 first rows of the successful installation logs are gathered. The dataset consists of 73 such files. Because each of the installations is on a different version of Linux, it is reasonable to assume that the variation within the data is considerably larger than in cases that only record a continuous operation of a single distribution. This likely results in a more challenging classification task and more anomalies on average.

7.2 Choosing the models

LSTM was chosen as the baseline as it is both the method used by Du et al. in a similar anomaly detection task [12], and a well established sequential learning method alongside its variants [14, 17, 36, 35].

Since the success of AlexNet [26] in 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), all of the subsequent winners [26, 53, 47, 18, 21] have used convolutions in their architecture. Therefore, it is logical to investigate whether the same approach could be adapted to suit sequential learning. There has already been multiple attempts [48, 13, 27, 5] at combining the two. The model by Bai et al. [5] was chosen to represent TCN, because of its performance, simplicity and

versatility. Rather than focusing on one aspect, this model has been tested with various sequential tasks [5].

In 2017 Google published their paper ambitiously titled 'Attention Is All You Need' [49], which not only reportedly outperformed the previous state of the art methods in sequential learning, but did so with a novel structure based around attention mechanisms and completely omitting LSTM variants. In addition, the paper reported significantly reduced computational costs in comparison to the best models in literature at the time. With all these factors making the network introduced in their paper, Transformer, exceptional, it was a natural choice for the third method to be tested in this thesis.

7.3 Preprocessing

Before it is possible to learn the order of the different row types, it is necessary to decide on how to separate them from each other. Similar to Du et al. [12], this thesis uses Spell [11] to do this.

It is noteworthy that Spell uses Longest Common Subsequence (LCS) to measure similarity, which has time complexity of $O(n^2)$. Applied to every row, this makes preprocessing a significant bottleneck in anomaly detection once the network has been trained. To lower the cost Du et al. suggest implementing a prefix tree. However, this does not seem to help much during the first million rows, perhaps because the data used in this thesis is too varied for the prefix tree to recognize efficiently. Although the benefits of a prefix tree increase over time, it does not produce the exact same results as naive Spell implementation. Because Spell relies on LCS in the case a prefix tree does not produce a result this leads to preprocessing still being a significant bottleneck.

Although Spell is a highly sequential algorithm, it is possible to reduce the time it needs by multiprocessing. In this approach the log is split into chunks that are processed separately. Once chunks are finished processing, all the produced keys are processed by Spell once more to combine the results. The problem with this approach is that the parts that Spell replaces with wildcards may contain several characters or units, depending on which of them Spell is using to count LCS. This leads to LCS not being accurate for data that has already been processed once. This leads to the keys becoming very short, to the point that they may be unable to meet the required threshold to pass LCS. Multiprocessing also stops Spell from being a streaming algorithm and although this is not a concern in this thesis, it should be noted when adapting the algorithm for other purposes.

It may be possible to improve multiprocessing Spell by providing the last row that was classified as each key type from the chunks and attempt running Spell on these instead of the keys. However, although this should mitigate the key length problem, it would still require changing the type of all the rows in the chunks to match combined key indexes. To find out whether this approach works better still needs more research.

Because of the aforementioned issues in multiprocessing, this thesis uses a regular version of Spell with a prefix tree. However, to save at least some cost, LCS is

configured to compare full words instead of characters.

The structure of the logs used in this thesis is as follows. In total, the dataset consists of 73 log files that have been truncated to 10,000 rows each and tokenized into 3,145 different key types. The sequences of rows are then cut into pieces of 30 tokens with padding added for the last row of the each file to keep the size consistent. Finally, these sequences of 30 were given to the network through the embedding.

Length 30 was chosen because it was estimated to be long enough to capture rows from several processes running parallel in the source system, but short enough to be computationally light to predict. These sequences were then shuffled with each other and then divided into training, validation and test sets. The ratio for this was 8:1:1. Each of the sets was divided into batches of 64. During the validation the networks were trained on 10 epochs. This results in each of the epochs being 327 batches long. Table 2 summarizes these statistics.

Table 2: Statistics of the dataset

Variable	Value
Log data files	73
Rows per file	10,000
Row types	3,145
Train data %	80 %
Validation data %	10 %
Test data %	10 %
Sequence length	30
Batch size	64
Batches per epoch	327
Epochs	10

In order to make sure that the results from the trained networks are not simply guessing the most common key, Table 3 has some statistics on key distribution. The keys in the data have large variance with the most common key appearing as 16.6 % of the rows. The five most common keys make up 34.6 %. Therefore, if a network has top-1 accuracy that is higher than 16.6 % and top-5 accuracy higher than 34.6 % it must come from learning the sequences rather than measuring key density. Top-1 accuracy and top-5 accuracy are explained in further detail in Chapter 7.5.

7.4 Networks

All of the methods were implemented by using PyTorch [1]. However, Transformer was written on Tensorflow [3, 2], which is why the implementation used in this thesis is an unofficial one [22]. Because TCN was written in PyTorch, this thesis is able to use the official source code linked in its paper [5]. LSTM and RNN networks are relatively simple in comparison to Transformer and TCN, so their code was crafted

Table 3: Statistics of the keys

Statistic	Value
Median	6,00
Variance	7,037,087
Top 1	120,853
Top 1 (%)	16.56 %
Top 5	252,891
Top 5 (%)	34.64 %
Total rows	730,000

manually, although PyTorch did provide the classes for LSTM and GRU cells. As such, their general design principles are described below.

The implementations of both the LSTM and the GRU network follow the standard Encoder-Decoder model. Encoder consists of the embedding layer that adjusts its weights to best represent the data and either LSTM or GRU layers for their respective networks, hereafter referred as RNN layer. The RNN layer is stacked so that the width of the network is the same as the output size of the embeddings, but each of the cells give their state to another RNN on the layer above them. The amount of these layers is a variable that can be validated.

Decoder is similar to the encoding, however, it has an additional linear layer as its last. This linear layer maps the results into 3,145 classes. Finally, Softmax function is applied to turn these values into probabilities. The only difference between LSTM and GRU networks aside from the structure of the cells is that LSTM wants two inputs, these being cell state and hidden state, while GRU only needs one.

7.5 Hyperparameter tuning

Hyperparameter tuning of the networks starts with initial parameters that are recorded in tables under their respective sections. These initial parameters were tuned one-by-one. Any divergence from them is mentioned under the relevant section. Embedding size and hidden size are tuned together. This is necessary because the dimensions have to be the same for residual connections in Transformer to function [22]. Although this restriction does not apply to all of the networks, their optimization is done in a similar manner for the comparison. The structure of LSTM and GRU networks demands that the width of the network is the same as the input. Training machines used in this thesis have 112.0 GB memory and 2 GPUs.

Tuning was performed by using a greedy algorithm, although if the results between two values were close, the both values were ran on the validation set. This algorithm was set to mainly optimize top-1 accuracy which is explained later in this chapter. However, if the top-1 accuracies were close, loss and top-5 accuracy were used to break the tie.

The graphs discussed in this section can be found at the end of this thesis in

Appendices. These graphs come in groups of four. The groups use four metrics to portray the tuning of the same parameter that is listed in the caption. These metrics are loss, perplexity (ppl), top-1 accuracy (acc) and final epoch training time. Loss in this context refers to cross-entropy (CE), which, as mentioned in Chapter 2.2, is the loss function all four networks use for their optimization criterion. The next metric in the tuning graphs is perplexity. The both papers on TCN [5] and Transformer [49] use it as a metric, making the results easily comparable. It should be noted that perplexity is calculated $2^{H(p)}$, where $H(p)$ is CE [20, 10]. This makes perplexity and CE very closely related. Therefore, they should be expected to behave very similarly to each other.

Top-1 accuracy refers to the ratio between the predictions where prediction matches the target result and the total amount of keys to be predicted. Likewise, top-5 accuracy is the ratio in which one of the five most probable keys predicted by the network matches to the target key.

The final epoch time measures the amount of training time that was taken by the final epoch. This metric measures the speed of the training. As such, it is calculated from the training set, while all of the other metrics in tuning are calculated from the validation set.

The last metric used in hyperparameter tuning is top-1 accuracy. This refers to the frequency in which the key predicted by the network is the expected key. This is especially important because it measures the absolute success of the network ie. the success rate of the class with the highest predicted probability, while the previous metrics consider the predicted probability for all classes. Accuracy is especially important, because in this task anomaly is defined as a key outside the first five predictions of the network. This is also known as top-5 accuracy. Having a high top-1 accuracy therefore also guarantees high top-5 accuracy.

Two aspects that were considered but ultimately left out from the tuning graphs are test time and top-5 accuracy. The former was omitted because the task is not time-critical and therefore it is more useful to measure training time that represents a larger percentage of the overall computational cost. The latter accuracy is not shown because it is not relevant to hyperparameter tuning. Although top-5 accuracy is the metric that is used to ultimately determine whether a key is considered to be an anomaly, there is a risk that maximizing it during training or tuning would encourage the network to make guesses based on the frequency of the key in the training and validation sets, because top-5 most common keys can cover a considerable percentage of all the rows in a log data file. This could lead to the model discriminating statistically unusual keys. The aim of this thesis is to discover the most probable next key *in a given sequence*, not to assess how common they are in the dataset.

An argument in favor of including frequency in predictions, is that rare rows should be considered an anomaly because they are unusual. However, the problems with suggestion can be shown with a counter-example. There are certain keys that always appear just once at the beginning of each log. As such they are 0.01% of the dataset and therefore very rare. However, because they always appear in the same place they are not anomalies. A network predicting based on frequency would not be able to recognize a very common key in a very unusual place as an anomaly. Even if

it were possible to gain a high accuracy from a network like this, the results would not say much about anomalies. This is why relying on frequency should be avoided.

In general high accuracy is preferred followed by high loss and perplexity. Because perplexity is calculated from the loss they tend to be similar to each other.

7.5.1 LSTM and GRU

Table 4: LSTM and GRU initial parameters

Parameter	Value
Learning rate	0.001
Embedding size	256
Hidden size	256
Layers per stack	3
Teacher forcing ratio	0.5
Epochs	10

Because LSTM and GRU are very similar to each other, their initial parameters shown in Table 4 are the same. Both them use Adam as their optimizer. In this context, the learning rate refers to the step size of the aforementioned optimizer. Embedding size is the size of the representation learned by the embedding layer. Hidden size refers to the width of the stacked LSTM network. Layers per stack is the amount of stacked LSTM or GRU layers at a time. Here it is worth noting that the encoder and the decoder have separate layer stacks. Therefore, if the layer stack number specified in the parameters is 3, this would translate to the both, the encoder and the decoder, having a stack of 3 layers and 6 in total. Teacher forcing ratio was explained in Chapter 3.1 and epoch in Chapter 2.3.

The first to be optimized is GRU. The relevant hyperparameter tuning graphs described here can be found in Appendix A.

Parameter optimization is started by optimizing the learning rate. This is to ensure that the gradients behave well and the network starts to converge. If the gradients explode or keep getting stuck, the network’s performance will be poor even if the other parameters were well optimized. According to Figure A1, $2^{-10} = 0.001$ yields the best performance in all measured aspects. $2^{-9} = 0.002$ is a close second, so the both of them are worth trying in further optimizations.

Following is the optimization of layers per stack. It can be seen that 2 has the highest accuracy and the lowest training time, however, this is by a very narrow margin. On the other hand, 3 has clearly the best loss and perplexity, however, it only places third on accuracy. 1 is not the best in anything, however, it places second on all the measured metrics.

The width of the network is optimized third. Width of the network in this context refers to the both, the width of the embedding and the width of the hidden size, as they are optimized together here. According to the graph $2^9 = 512$ clearly performs

the best. Although $2^{10} = 1024$ also does well enough on loss, perplexity and accuracy to be a viable alternative, the exponential rise it takes in processing time makes it rather inconvenient.

The final parameter to be optimized is teacher forcing ratio. The hyperparameter tuning graphs on this show considerable inconsistency. This may be due to the randomness inherent to the method or alternatively there not being enough data. Either way, this graph shows an interesting dichotomy between the first two rows. According to loss and perplexity 0.5 clearly performs the best, however, the values close to 1 have the best accuracy and training time.

To make sure that the selected parameters work well together, the network is tested with various combinations of the parameters that performed well in earlier stages of hyperparameter tuning. According to Table 5 the hyperparameters chosen by greedy tuning algorithm proved the highest accuracy.

Table 5: Further optimization of GRU. Lr stands for learning rate, stack for layers per stack, hidden for the combination of embedding size and hidden size together, TFR for teacher forcing ratio, ppl for perplexity, acc for top-1 accuracy and time for the training time of the last epoch in seconds. The best results in each category have been bolded.

lr	stack	hidden	TFR	loss	ppl	acc	time
0.001	2	512	1	1.0426	2.8366	0.8884	114.27
0.001	1	512	1	1.2506	3.4924	0.8586	108.78
0.001	3	512	1	1.2559	3.5111	0.8618	132.67
0.001	2	512	0.5	0.5865	1.7976	0.8431	129.84
0.001	2	1024	1	1.2163	3.3745	0.8792	197.34

From the Table 5 we can see that there is no single set of parameters that would be the best at everything. To be able to choose one model to represent GRU, it is necessary to consider the priorities of the work. Although using forced learning rate of 0.5 significantly improves loss and perplexity, in this task accuracy is more important, as the former two are calculated from the entire matrix, while anomaly detection only cares about the five most probable keys out of thousands. However, without knowing the distribution of the probabilities in the result matrix it is hard to say how this translates to anomalies. Additionally, top-1 accuracy is not that much lower than on the first row of Table 5. Because the performance of these sets of parameters is roughly equal, top-5 accuracy is used as a tie-breaker. As such, TFR = 0.5 is used to represent GRU with top-5 accuracy of 0.9854, opposed to top-5 accuracy of 0.9508 of the first row in Table 5.

LSTM is optimized after GRU. The hyperparameter tuning graphs can be found in Appendix B. As with above, the learning rate is the first parameter to be optimized. Despite the similar structure and the same parameters, the change in parameters seems to affect the results in different ways. It appears that in this particular set-up GRU is more sensitive to changes in learning rate, while LSTM performs relatively

even with suboptimal values. Based on the graph, the best learning rate for LSTM is $2^{-9} = 0.002$, although $2^{-10} = 0.001$ and $2^{-8} = 0.004$ are also close.

As with GRU, the second LSTM parameter to be optimized is layers per stack. According to the graph the metrics stay quite stable for four layers, before the performance starts to falter when the stack reaches 5. In particular, 2 appears to yield the best performance in the first three categories, with 1 being the second best.

Network width is another section where there seems to be differences between LSTM and GRU. According to the graphs, GRU reaches its peak at 512, while loss, perplexity and accuracy keep improving the wider the network is for LSTM. However, because the training time starts to spike at 1024, it is not viable to try whether increasing the value further would keep improving the results. As such, the best value for network width is 1024.

Teacher forcing ratio is optimized last. Unlike with GRU where there was a dichotomy with loss and perplexity on one side and accuracy on the other, with LSTM both of them appear to favor $TFR = 0.5$.

As the network is trained with all the tuned parameters it is apparent that it does not yield as good performance as was shown during the tuning of network width. Therefore, the learning rate is returned to the same value as it was during the aforementioned hyperparameter tuning and the results immediately improve. The same learning rate also worked for optimizing GRU and therefore teacher forcing rate is changed as it increased the accuracy with GRU. The results show that it has a similar effect in LSTM as well, including the increased loss and perplexity. Finally, reducing the network width is tried to make the network training faster. However, this causes the performance on other categories to drop too much to be useful.

Using a lower teacher forcing rate does not seem to reduce accuracy as much with LSTM as with GRU. Meanwhile, the differences in loss and perplexity remain significant. Therefore, this thesis considers the parameters used on the second row of Table 6 as the optimal for LSTM.

Table 6: Further optimization of LSTM. The abbreviations are the same as with GRU. The best results in each category have been bolded.

lr	stack	hidden	TFR	loss	ppl	acc	time
0.002	2	1024	0.5	0.5591	1.7491	0.8344	223.21
0.001	2	1024	0.5	0.3895	1.4762	0.8883	225.46
0.001	2	1024	1	0.8387	2.3134	0.8984	222.05
0.001	2	512	1	1.0093	2.7437	0.8687	124.77

7.5.2 TCN

For consistency, the variables described in Table 7 are consistent with those in the original paper [5]. As with the LSTM and GRU, learning rate refers to the step size of the optimizer. However, unlike the earlier networks, TCN uses SGD as its optimizer.

Table 7: TCN initial parameters

Parameter	Value
Learning rate	4
Embedding size	600
Embedding dropout	0.25
Gradient clip	0.35
Hidden size	600
Dropout	0.45
n	4
k	3
Valid sequence length	30
Epochs	10

Once again, embedding size is the size of the learned embeddings. Embedding dropout controls the probability that an individual connection is cut during the dropout layer right after the embedding. Hidden size is the internal width of the network outside input and output layers. Like earlier, dropout is the probability of that an individual connection is cut, this time between network layers. Gradient clip is a technique that prevents gradient explosion by clipping unreasonably large gradients. n is the amount of residual blocks in the network. k is the size of the convolution filter ie. kernel. Valid sequence length is the length of the effective history. Since the aim here is to predict the next key type after each row, the size of the input data is the same as the output and valid sequence length is the same as input sequence length. TCN hyperparameter tuning graphs are located in Appendix C.

As with the earlier networks, hyperparameter tuning is started with the learning rate. Because TCN uses SGD optimizer the values it take are much larger than the ones in the networks above. The graph shows that TCN behaves in a stable and predictable manner as the learning rate changes, with the top values performing close to the each other. By a narrow margin $2^3 = 8$ yields the best results, with $2^2 = 4$ being a close second.

Filter size k is the next variable to be optimized. Similar to what happened with the results of the earlier networks, loss and perplexity prefer one value while the accuracy suggests another. Therefore it is prudent to attempt using the both 4 and 5 before committing on just one.

Tuning of levels n is interesting because the metrics seem to alternate between increasing and decreasing without a clear trend. Even when there are as many as 10 layers the networks seems to be able to optimize its weights properly without the performance suffering. Based on the chart $n = 10$ and $n = 6$ seem to yield the best results.

The final parameter of TCN to be considered here is hidden size. There is a clear trend that bigger models perform better and 1024 is chosen for further optimization.

As the network is run with all of the parameters that performed the best is run, it becomes apparent that the model is too computationally heavy for the machine and the operation fails. Since the hyperparameter tuning graphs show the steep increase in epoch time with high values for the both hidden size and n , these parameters are decreased to the values that performed the second best. This allows the network to be trained properly, however, the training time is still slower than the other networks. The further optimizations show that in contrary to the results of individually tuning parameters, a smaller model is more effective. Although this thesis considers accuracy to have the highest priority in hyperparameter tuning, the considerable decrease in training time makes the last tested model preferable. The details of the further hyperparameter tuning of TCN are described in Table 9.

Table 8: Further optimization of TCN. The best results in each category have been bolded.

lr	hidden	n	k	loss	ppl	acc	time
8	1024	10	4	<i>N/A</i>	<i>N/A</i>	<i>N/A</i>	4535.69
8	512	10	4	0.5795	1.7852	0.8528	3628.62
8	1024	6	4	0.5683	1.7653	0.8597	579.53
8	512	6	4	0.5894	1.8028	0.8553	405.14
8	512	6	5	0.5687	1.7659	0.8576	610.47
4	512	6	4	0.5709	1.7698	0.8535	406.43
4	512	3	5	0.5877	1.7999	0.8531	290.34
8	512	3	4	0.5668	1.7626	0.8573	229.84
8	512	3	5	0.5616	1.7535	0.8580	288.43

7.5.3 Transformer

Table 9: Transformer initial parameters

Parameter	Value
d_{model}	512
d_{ff}	2048
h	8
N	6
d_k	16
d_v	16
P_{drop}	0.1
Warm-up steps	4000
Epochs	10

As with TCN, the variables described in Table 9 are consistent with those in the original paper [49]. Chapter 5 contains a more detailed explanation of the network structure and its components. d_{model} is the dimensionality of embedding layer as well as all the input and outputs of the sublayers defined in Chapter 2.5. d_{ff} is the inner-layer ie. hidden layer dimensionality of the feedforward network. Although the inner dimensionality can be different, the input and output layers of this feedforward sublayer are still d_{model} like other sublayers. h is the number of heads in multi-head attention. N is the number of identical layers that the network consists of. d_k and d_v are the dimensions of scaled dot-product attention that was explained in more detail in Chapter 5.2. P_{drop} is the dropout probability. Transformer does not have a separate learning rate parameter, because the optimization is done automatically in the implementation used in this thesis. The number of warm-up steps adjusts the scheduler that does this optimizing. Transformer hyperparameter tuning graphs are in Appendix D.

Transformer optimization is started from layer number N . According to the graphs, models with a large amount of layers perform better, although the gained benefits seem to decrease after the number reaches 8. Although 10 seems to perform the highest, either of the values seems like a reasonable possibility for further optimization. The increase in training time as the layer number is increased is approximately linear.

As with layer number, model dimension size d_{model} seems to also favor larger models. However, the increase seems to reach the maximum at $2^{10} = 1024$, so there is little reason to attempt values higher than that. A second option are the lower values that might perform worse, but are also faster to calculate.

The number of attention heads h is tuned next. The performance stays high with values of 4 and under. After this the performance deteriorates as the number of heads increase. Therefore, further optimizations focus on small values of h .

In the results reported by Vaswani et al. [49] the sizes of d_v and d_k are kept the same. However, testing different values reveals that they behave quite differently. The tuning of d_k seems to mirror the results reported by Vaswani et al. [49] in that the larger values perform better. However, d_v does the opposite, with the best values being the smallest.

Because at this point it is apparent that Transformer produces the best results, there was some extra focus on its optimization. First d_{model} was changed into a larger value as indicated by the earlier tuning results. With this setting the hyperparameter tuning for h was run again. Unlike the smaller model that performed better with small values, there is now a clear preference to $h = 8$. It might be that the ideal size of h is proportional to the size of d_{model} .

After this, dimension sizes d_v and d_k are optimized. This time the both values are changed together to make the result more comparable to Vaswani et al. [49]. The results gotten here seem to indicate the opposite to theirs, as the network performs the best when the variables are kept small. It is possible this is caused by the difference in task, as the dependencies between natural languages used in the task of Vaswani et al. [49] may be more complicated than in log generation, requiring a more complicated model. Furthermore, their task is supervised learning while the task in this thesis is unsupervised and therefore they are fundamentally different.

Another possibility is the amount of training. The training set used by Vaswani et al. [49] is much larger than the one used in this thesis. Larger parameters lead to larger models that are more difficult to optimize. Large amounts of training data or epochs may help the network to converge, however, they also increase the total training time. The details of the further hyperparameter tuning of Transformer are described in Table 10.

Table 10: Further optimization of Transformer. The best results in each category have been **bolded**.

N	d_{model}	d_{ff}	h	d_k	d_v	loss	ppl	acc	time
2	1024	2048	2	256	16	0.14726	1.15865	0.98446	104.226
2	1024	2048	8	256	16	0.11697	1.12409	0.98730	150.761
3	1024	2048	8	16	16	0.14812	1.15965	0.98399	253.304
3	1024	2048	8	256	16	0.12665	1.13501	0.98770	201.459
4	1024	2048	8	16	16	0.12586	1.13412	0.98876	153.398

8 Results and discussion

8.1 Preprocessing

It can be said that the preprocessing of the data is quite inefficient. Although using a prefix tree with Spell somewhat speeds it up, it is worth discussing that a prefix tree and LCS do not always give the same result in regards to which key a row should be classified to. At the beginning the Spell is slow because LCS has to be used for all the new keys, but as the amount of keys increases the aforementioned disparity still causes prefix tree to fail some of the time. This is because each time there is a wild card character, the prefix tree resumes parsing when it finds the word that matches any of its children. However, it is possible that the same word appears within a parameter, especially if the word in question is short or common.

For example, if there is a key "Couldn't write * to /var/lib/log/na.txt: File does not exist." data row "Couldn't write 'mount_activate: Failed to activate' to /var/lib/log/na.txt: File does not exist." would not fit to the key according to the prefix tree, because the parameter contains 'to', which is also one of the wildcard's children. This would cause the prefix tree to assume the parameter has ended earlier than it does in reality and then give a negative comparison result at the next word. Another example of the prefix tree not working properly is when there is both a matching word and a wildcard as children of the previous word. The algorithm has no way of telling which of them would match the current row.

Mistakes like these are particularly costly because the amount of possible keys is large and LCS operation is very expensive. For each row the prefix tree misses, the time complexity is $O(m \cdot n^2)$, where m is the number of keys and n is the size of each log entry [11]. Furthermore, parallelization of Spell is not trivial because the keys are changed and added incrementally, with each change influencing the next parsing result.

In the setting of this thesis one machine produces 600,000 log data entries in an hour, with multiple machines running in parallel. Therefore we can expect quite a large number of prefix tree misses from the volume alone. Additionally, just processing 10,000 first rows of each log file produced more than 3,000 unique keys. It can be expected that fully processing 600,000 rows results in an even larger amount of keys, which translates into even larger cost when the prefix tree misses.

The cost of the preprocessing was the reason the experiments in this thesis use 10,000 first rows instead of the entire 600,000 row log. This is why although preprocessing is not the focus of this thesis, it is still discussed here, as it greatly influences the ability to train the networks with enough data.

There are a few ways the preprocessing could be improved. For example, the prefix tree could have more sophisticated parsing. This would prevent situations like words in quotes being recognized as children. In situations with both a matching child and a wildcard the algorithm would explore the both options, as this is still faster than LCS. The problem of the prefix tree not recognizing everything would still exist, but its magnitude would be smaller.

Another possibility is to have the number of words each wild card contains

recorded with it. However, there are parameters of various lengths and the prefix tree would not know which of them is the correct one for each row. Even then, this would at least remove some of the failures.

There is also the option of splitting the full log into smaller pieces and running Spell on each of them separately. However, this would mean that the log parts would have to be recognized with different models as the key dimensions would be different for the each part. Another option would be to find a way to combine the keys, but this could lead to a massive increase in the number of keys making the learning more difficult for the network.

8.2 Networks

As mentioned above, the inefficient preprocessing led to the training data being rather limited. Because studies are often done on larger data sets, it may influence the comparability of the results. Additionally, an experimental set-up with limited data favors the networks that are easy to optimize and converge fast. In practice, this often translates to small and simple networks. However, being easy to optimize is generally a desirable quality, so depending on the circumstances accounting it may not be exclusively a bad thing. For example, because the Linux distributions that produced the System Log for this thesis are constantly under development and the network used to detect anomalies has to be retrained periodically. Using very large amounts of data for this can make the process slow and very computationally heavy, and therefore it is preferable for the network to be easily optimizable.

The amount of epochs used in the experiment was also fairly small, so it is possible that some of the networks would have done better had they been allowed more passes over the data. However, that also presents the possibility of overfitting to the training data. Overfitting could be mitigated by saving the model between epochs every time a record low loss is achieved on the validation set and therefore, it would be beneficial to run more epochs.

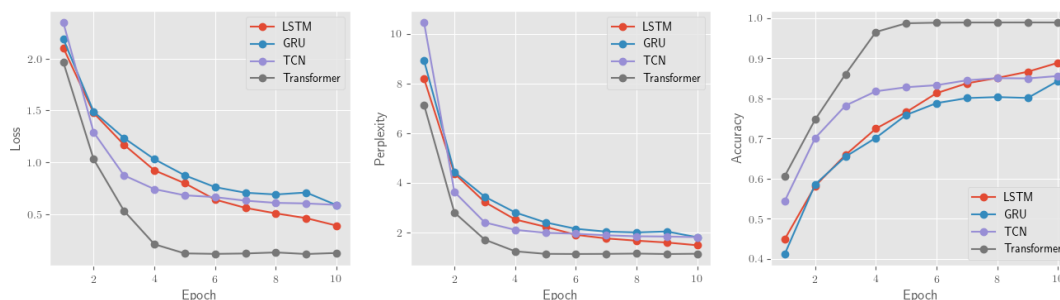
Currently hyperparameter optimization is implemented with a simple greedy algorithm. However, this may not be the most effective optimization method. According to Bergstra et al. [7] using a random search can be both more effective and efficient. Therefore, in the future it may be beneficial to move towards implementing random search.

This thesis focuses mainly on increasing top-1 accuracy during hyperparameter optimization. However, although the argument in favor of this is presented in Chapter 7.6, it is necessary to acknowledge that this approach is not the only one. If the goal is to just maximize the final top-5 accuracy, it is possible that loss, perplexity or top-5 accuracy itself might be more effective hyperparameter tuning metrics to focus more on a larger portion of the predictions than just the prediction that has scored the highest. However, top-1 accuracy is still the best at representing the ideal where the predictions are exactly correct rather than almost.

Like all the other metrics, top-1 accuracy has its flaws. Chapter 7.6 mentions an example showing that it is possible to increase accuracy, even though the results may not represent anomalies at all. This illustrates how the metrics used in this thesis

to describe the success of the network may not always work in the intended way. Although these concerns were addressed earlier in Chapter 7.4 and later in 8.1 by calculating key statistics to show that guessing keys based on their frequency would lead to low accuracy, the reality is that none of the metrics actually measure whether something is an anomaly or not. If that was possible, there would be no need for a network to be trained in the first place. What top-5 accuracy measures is not the complement to the amount of anomalies, but a complement to the amount of false positives. Because the data does not have any labeled anomalies, it is impossible to definitively conclude the accuracy of anomaly detection or the amount of false negatives. This problem of uncertainty is intrinsic to unsupervised learning, since if it was possible to definitively say which class or cluster an instance should go to, this task would be a supervised learning problem instead. Therefore, although the aim is to be as precise as possible, the results, including the results of this thesis, always have a certain amount of uncertainty.

Figure 8: Comparison of the optimized networks.



As seen in Figure 8, all of the networks did fairly well. Top-5 accuracy that is used as the threshold for flagging anomalies consistently stayed above 95 % with Transformer reaching as high as 99.69 % top-5 accuracy. In practice, this would translate to only 0.31 % of the rows flagged as false positives. All of the accuracies were significantly higher than the accuracy of always guessing the most frequent key and therefore it is possible to conclude that the predictions were learnt from the sequences rather than the frequency. The details of the results can be seen in Table 11.

The final results show how Transformer is clearly the best in all of the measured metrics but the training time where it placed second after GRU. Furthermore Transformer manages this while maintaining much larger network width and depth than GRU. This may be because omitting the sequential structure of RNNs allows for more parallelization. In this thesis the experiments were run with two GPUs in parallel. It is likely Transformer would be even faster with more GPUs. Additionally, it reaches a stable level of performance in only a few epochs. Although the Transformer used in this thesis was an unofficial implementation and may therefore have some differences to the official TensorFlow implementation that affect the results, it still outperformed all of the other networks. Overall, based on these results there is very little reason to use any other network than Transformer.

The baseline method of this thesis, LSTM, also performed very well, placing second on loss, perplexity and both accuracies. However, it seems like its weakness is the training speed. It is both somewhat slow on epoch training time and the graph also shows that it may benefit from having more epochs. This can be seen as the results keep improving until the last epoch. This may also be the case for GRU as its performance seems to jump slightly in the final epoch.

One of the most important reasons why LSTM is slower than GRU is because it does better on larger model sizes. When smaller network widths were tested during hyperparameter tuning, the results were much closer in both the training time and the other measured metrics. This is not the only reason why GRU was faster though, because as seen from the hyperparameter tuning graphs, it still runs faster than LSTM when $hidden = 1024$. The probable reason for this is the more streamlined structure of the cell that was introduced in Chapter 3.3. In general, speed seems to be GRU's strength in comparison to the other tested networks. This is even more impressive as the sequential structure of both GRU and LSTM make their parallelization options very limited in comparison to Transformer and TCN.

Although it does not quite reach the final performance of the other networks, TCN was the first network to stabilize during its training. This is only in regards to epochs though, as TCN was the slowest of the networks. Part of this can likely be attributed to only having two GPUs, as convolutions tend to benefit from parallelization and Bai et al. [5] mentioned it as one of the benefits of the network. Another feature that was left unused was the ability to have a long and flexible receptive field size, as the task focused on predicting the sequences that were the same length as the input. As such, it is possible that the task did not suit this network.

As can be seen during hyperparameter tuning, the best attribute of TCN is its stability. As mentioned before, it was the network that required the least amount of epochs to start to stabilize and yielded a good performance with a wide variety of hyperparameters. It is quite easy to increase hyperparameters so that the training time becomes unreasonably long or the computations demand large amounts of RAM, however, deterioration of loss, perplexity and accuracy stay fairly high even under these conditions. This may indicate that the network would perform well on very complex data with long sequences that require complicated models. However, the steep rise in required computational resources as the model complexity increases is a negative.

In Chapter 2.5 the task in this thesis was likened to NLP. Although there are similarities, there are also significant differences. These differences are important because NLP is perhaps one of the most researched problems in sequential learning and one of the primary tasks the networks used in this thesis have been tested before. Therefore, it is possible that the differences cause unexpected behavior.

One of the major differences between the task in this thesis and NLP is that the cause and effect relation only flows from the past to the future in Linux System Log. However, a natural language can allow the effect to take place before the cause is known. For example, "I returned home because it started raining" is such a sentence. In general, it seems that the relationships between the words are more complicated in natural language than in Linux System Log. Although it is possible that this

Table 11: The final results are calculated from the test set, save for epoch time column that measures the training time of the last epoch. Frequency refers to always guessing the most frequent keys in the data. The best results are bolded.

	loss	ppl	top-1 acc	top-5 acc	epoch time
Frequency	<i>N/A</i>	<i>N/A</i>	0.1656	0.3464	<i>N/A</i>
LSTM	0.3894	1.4762	0.8876	0.9943	225.46
GRU	0.5864	1.7975	0.8431	0.9854	129.84
TCN	0.5773	1.7812	0.8548	0.9562	405.14
Transformer	0.12382	1.1318	0.9889	0.9969	153.40

makes the prediction easier, it may also unnecessarily complicate the problem, if the structure of the network is intended for more complicated data.

There are also features in Linux System Log that do not exist in NLP. Parallelization can lead to situations where every other log row, or word in the context of NLP, is related to a different source. It is very unlikely that something similar would happen in NLP. As with the other differences, this may lead to unexpected behavior.

Although the exact weights of the network are very context dependent, it is likely that the general results of the network comparison can be generalized to other types of logs and possibly structured as well. However, because of the similarities mentioned in Chapter 2.5 and the differences mentioned above, there may be limited generalizability to NLP. In general, the more different the tasks are, the less likely it is that the results can be generalized to the other task.

The differences between networks in regards to training time are probably more dependent on the machine they are run than the task. Therefore, these results apply likely in other tasks, if the amount of parallel GPUs is 2. If there is only one, the speed of LSTM and GRU will likely improve in comparison to TCN and Transformer, as the sequential structure of LSTM and GRU makes it difficult to benefit from parallelization. The opposite can be expected if the number of GPUs rises instead.

9 Summary

This thesis tested the performance of four different deep neural networks in detecting anomalies from Linux System Log. It was discovered that Transformer performed clearly the best out of the tested networks.

The raw log data was first tokenized by using a LCS based parsing algorithm, Spell. This allowed an easy numerical representation of character data. After that, the data was divided into sequences of 30 and given to the tested deep neural networks. The length of 30 was chosen, because it was long enough to likely see rows from all the processes that were running parallel, while still being small enough to be computationally light to predict. The networks were trained on System Logs of the successful Linux installations, because this allows the network to flag everything unexpected as an anomaly without knowing what an anomaly looks like. The definition of an anomaly in this thesis is a key that is not within first five predictions given by the network that's being tested.

The networks structures that were experimented on this thesis were LSTM, GRU, TCN and Transformer. LSTM is a well-established network for sequential learning and acted as the point of comparison to the earlier research that had used LSTM for a similar task. GRU was chosen because it is known to be similar to LSTM but faster. TCN represents the convolutional approach that has been proven to very successful in image recognition. Transformer was chosen because its unique structure and high performance in its original paper.

The result of the comparison was that Transformer performed clearly the best, reaching top-1 accuracy of 98.9 % and top-5 accuracy of 99.7 %. The rest of the networks reached over 95 % top-5 accuracy, although their top-1 accuracy was less than 90 %. Therefore, this thesis concludes that Transformer is the most suitable network structure for the task. It was also discovered that Spell is a major bottleneck in the prediction pipeline.

This thesis has successfully identified a solution to log anomaly detection that is both faster and more accurate than the method previously used. These findings encourage the use of Transformer in further applications and research. They can also be used to inform optimal network structure in other sequential anomaly detection tasks.

Based on the results, this thesis suggests that further research on effective tokenization of logs. As it is, Spell is too slow to keep up with the rate System Log is being produced. This thesis also highly recommends using Transformer in any practical applications of sequential anomaly detection, as its predictions have proven to be both very effective and efficient. Finally, it is concluded that based on these results, deep learning seems to be very effective at recognizing patterns in logs and shows great promise in anomaly detection.

References

- [1] Pytorch. <https://pytorch.org/>. Accessed: 2019-05-26.
- [2] Tensor2tensor. <https://github.com/tensorflow/tensor2tensor>. Accessed: 2019-05-26.
- [3] Tensorflow. <https://www.tensorflow.org/>. Accessed: 2019-05-26.
- [4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [5] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.
- [6] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.
- [8] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.
- [9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [10] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinfeld. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.
- [11] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.
- [12] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.
- [13] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

- [14] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.
- [15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [16] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.
- [17] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.
- [18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [20] Thomas Hofmann. Unsupervised learning by probabilistic latent semantic analysis. *Machine learning*, 42(1-2):177–196, 2001.
- [21] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.
- [22] Yu-Hsiang Huang. Attention is all you need: A pytorch implementation. <https://github.com/jadore801120/attention-is-all-you-need-pytorch>. Accessed: 2019-05-26.
- [23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [24] Anil K Jain, Jianchang Mao, and KM Mohiuddin. Artificial neural networks: A tutorial. *Computer*, (3):31–44, 1996.
- [25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [27] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017.

- [28] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- [29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- [30] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [31] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- [32] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.
- [33] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.
- [34] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
- [35] Chih-Yao Ma, Min-Hung Chen, Zsolt Kira, and Ghassan AlRegib. Ts-lstm and temporal-inception: exploiting spatiotemporal dynamics for activity recognition. *Signal Processing: Image Communication*, 71:76–87, 2019.
- [36] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.
- [37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
- [39] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA:, 2015.
- [40] Ankur P Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*, 2016.
- [41] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*, 2017.

- [42] Saurabh Rathor. Simple RNN vs GRU vs LSTM :- Difference lies in More Flexible control. <https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57>. Accessed: 2019-05-26.
- [43] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.
- [44] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [45] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.
- [46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.
- [47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [48] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *SSW*, 125, 2016.
- [49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [50] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.
- [51] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.
- [52] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.
- [53] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

A GRU hyperparameter tuning graphs

Figure A1: Hyperparameter tuning of GRU learning rate.

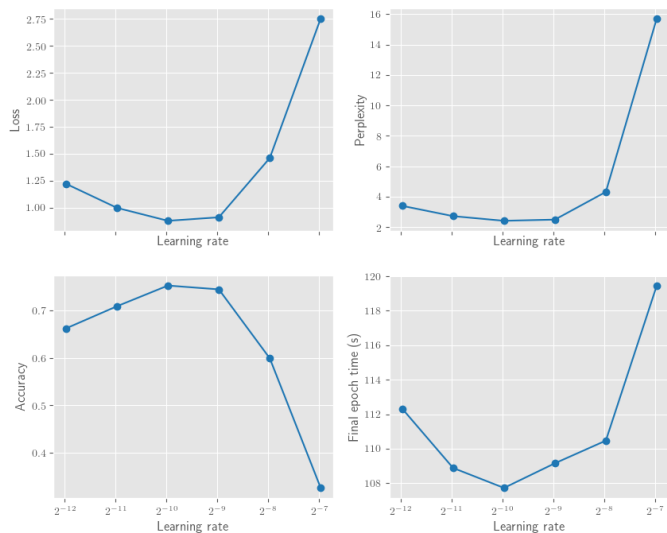


Figure A2: Hyperparameter tuning of GRU layer stack depth

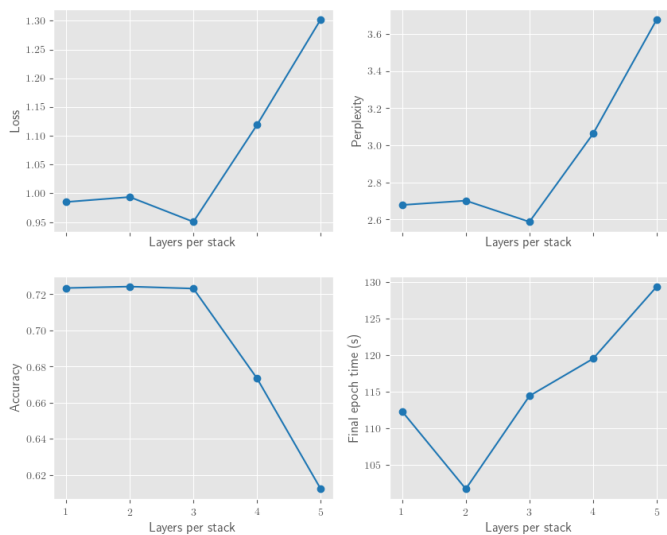


Figure A3: Hyperparameter tuning of GRU network width.

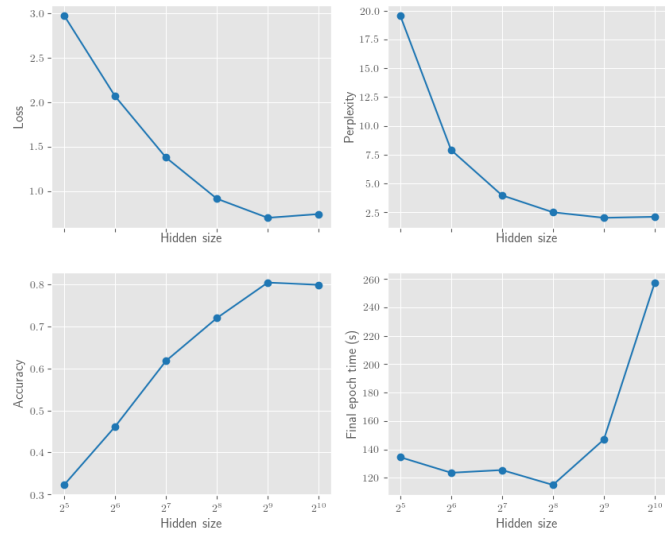
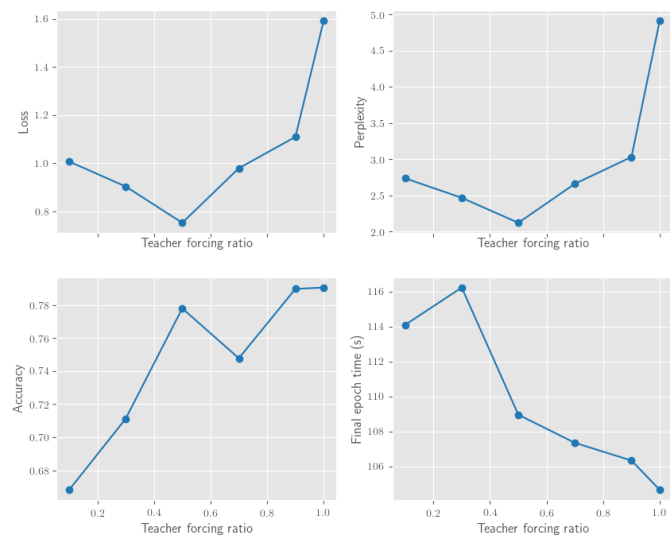


Figure A4: Hyperparameter tuning of GRU teacher forcing ratio.



B LSTM hyperparameter tuning graphs

Figure B1: Hyperparameter tuning of LSTM learning rate.

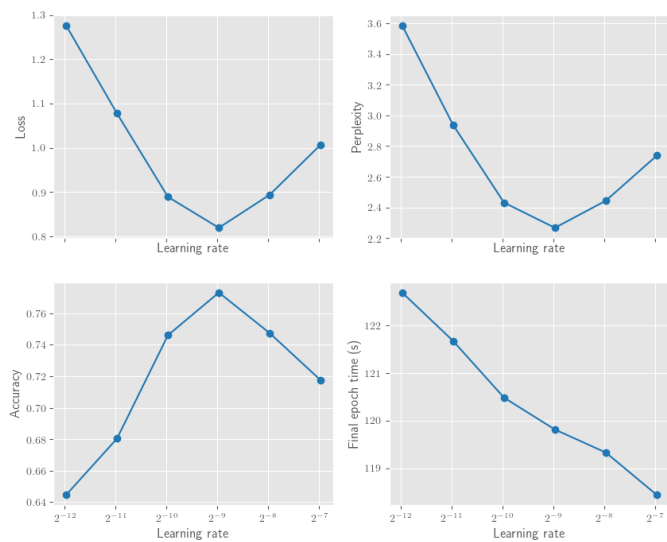


Figure B2: Hyperparameter tuning of LSTM layer stack depth

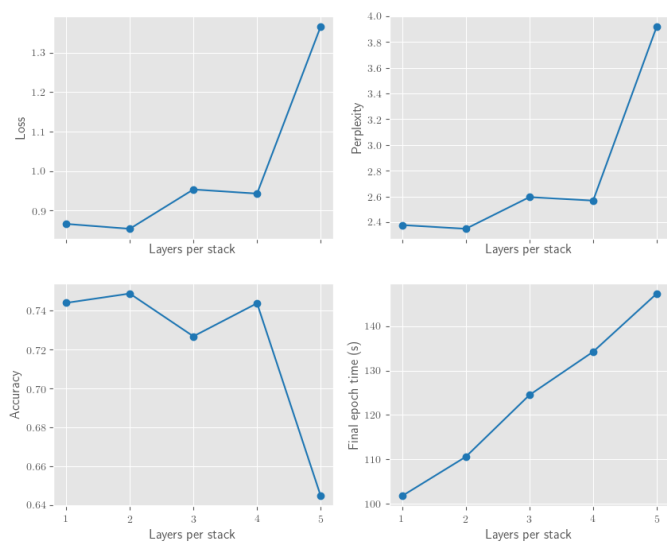


Figure B3: Hyperparameter tuning of LSTM network width.

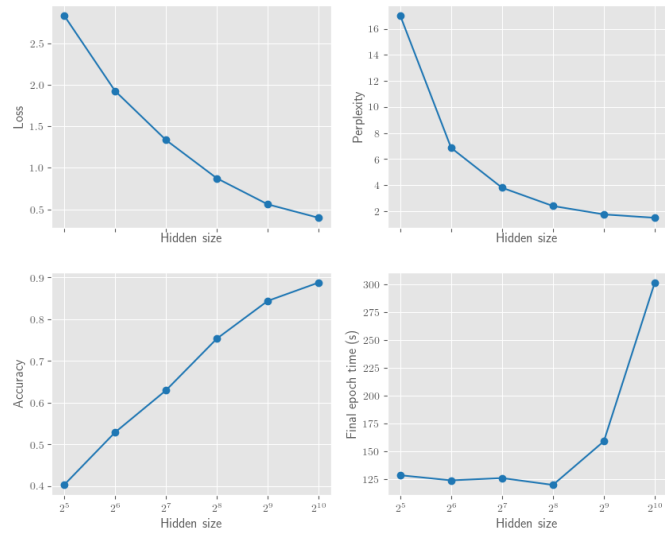
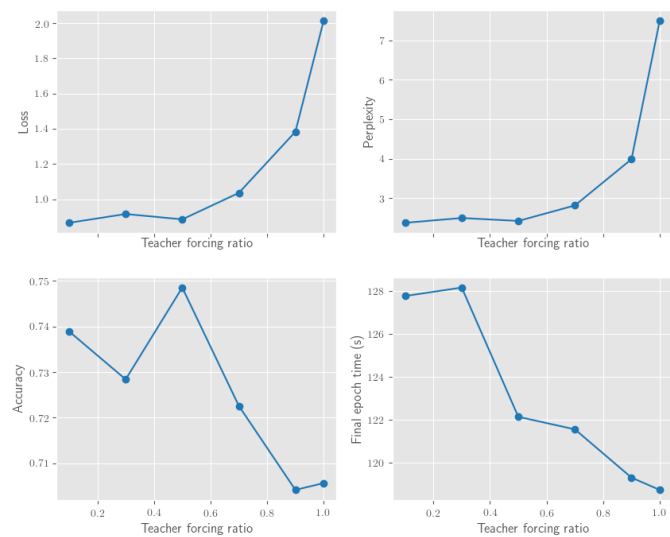


Figure B4: Hyperparameter tuning of LSTM teacher forcing ratio.



C TCN hyperparameter tuning graphs

Figure C1: Hyperparameter tuning of TCN learning rate.

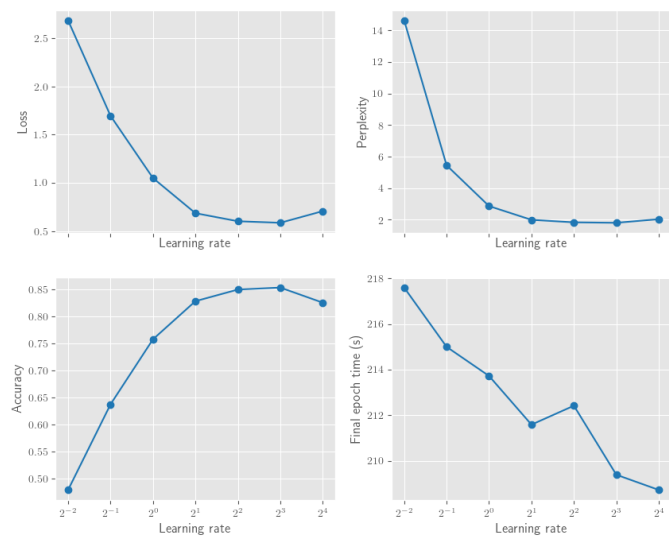


Figure C2: Hyperparameter tuning of TCN filter size.

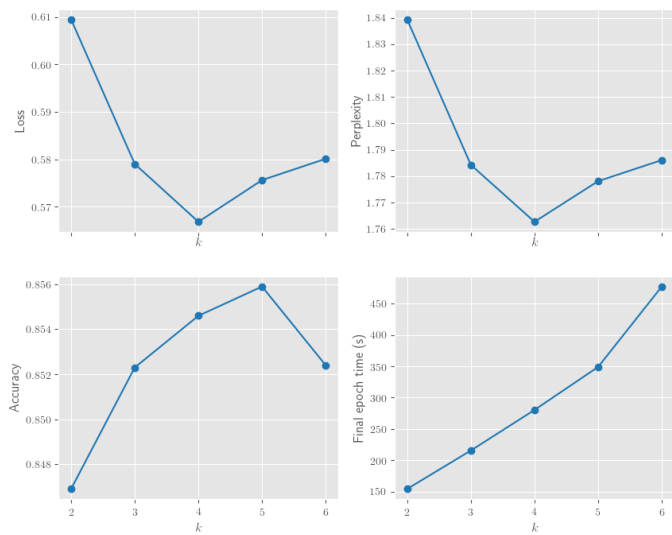


Figure C3: Hyperparameter tuning of TCN levels.

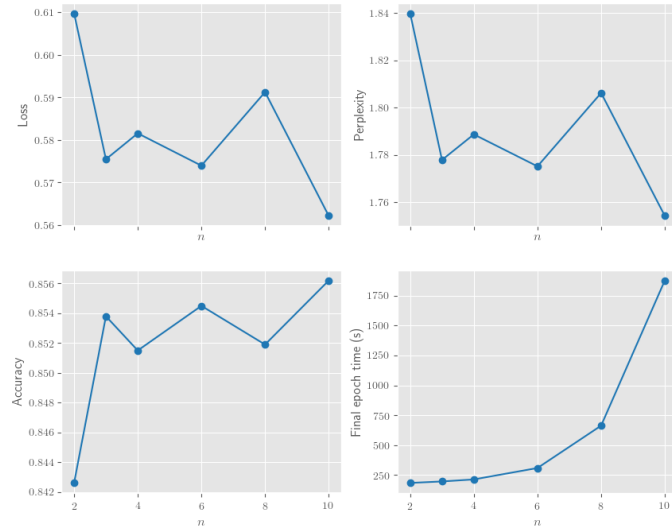
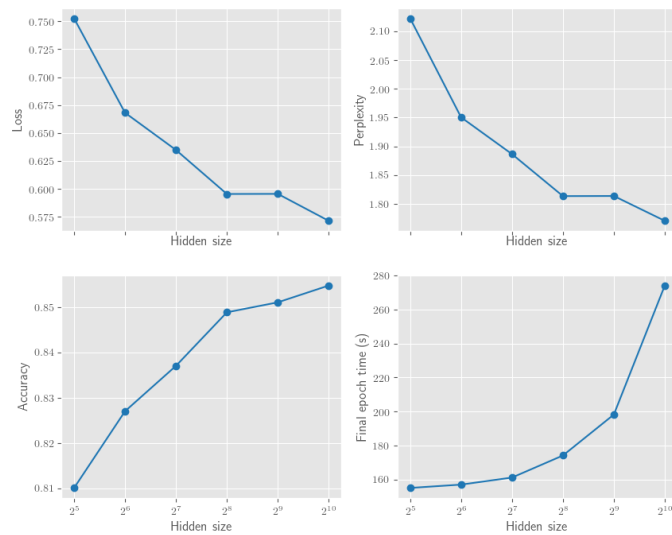


Figure C4: Hyperparameter tuning of TCN hidden size.



D Transformer hyperparameter tuning graphs

Figure D1: Hyperparameter tuning of Transformer layer number.

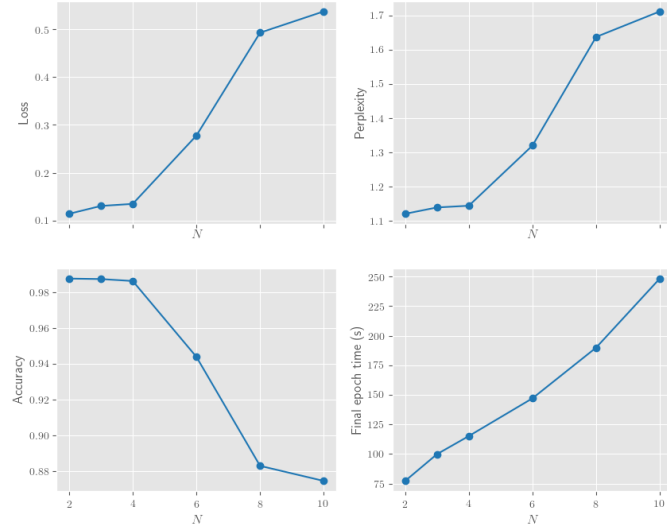


Figure D2: Hyperparameter tuning of Transformer model dimension.

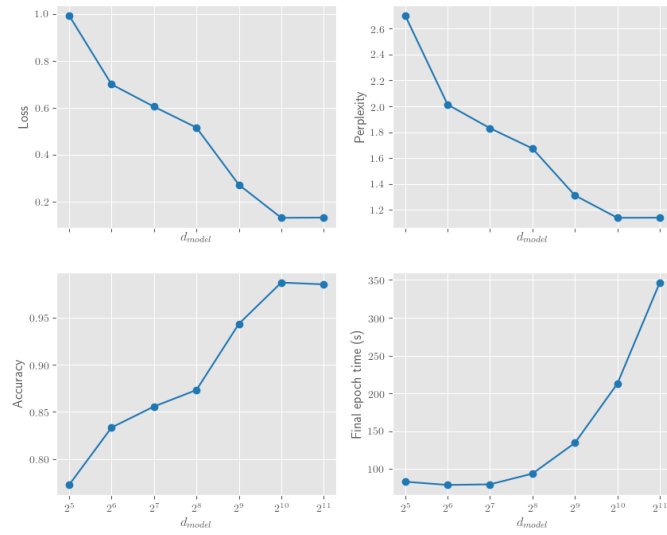


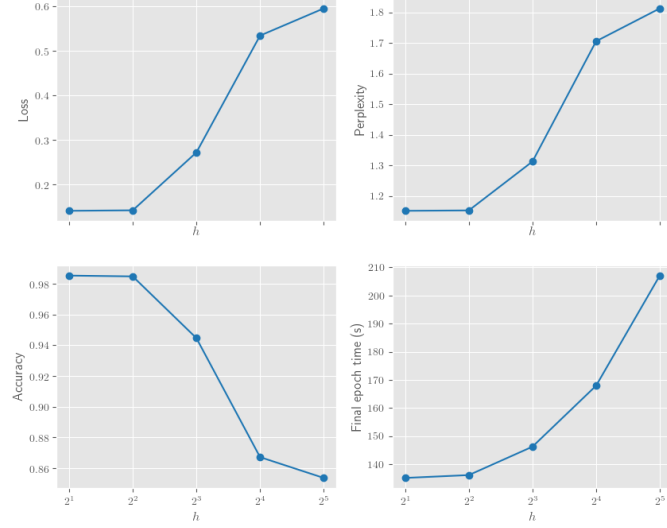
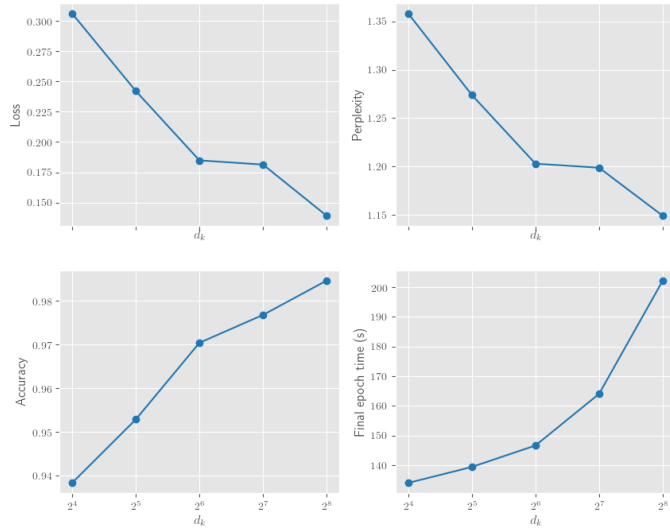
Figure D3: Hyperparameter tuning of Transformer head number when $d_{model} = 512$.Figure D4: Hyperparameter tuning of Transformer d_k when $d_{model} = 512$.

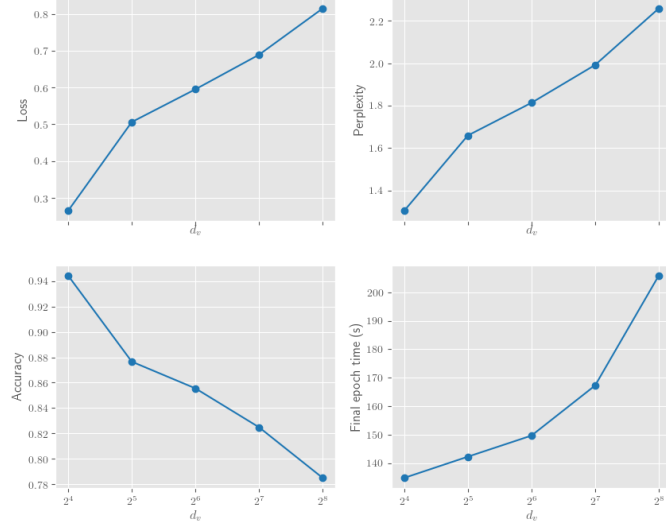
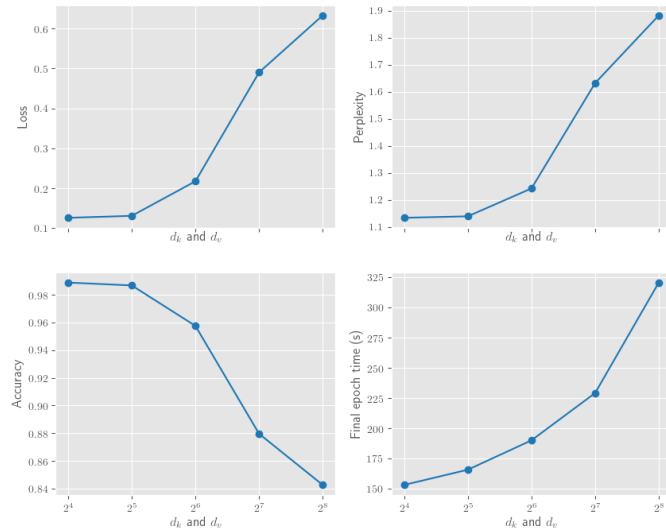
Figure D5: Hyperparameter tuning of Transformer d_v when $d_{model} = 512$.Figure D6: Hyperparameter tuning of Transformer h when $d_{model} = 1024$.

Figure D7: Hyperparameter tuning of Transformer d_k and d_v when $d_{model} = 1024$.