Faculty of Science and Technology

Department of Physics and Technology

Detecting Unhealthy Comments in Norwegian using BERT

Joakim Warholm

FYS-3900 Master's Thesis in Physics May 2021

Abstract

In this work we present a new Norwegian labeled dataset of 7078 comments for unhealthy comment detection. The dataset is used to fine-tune a BERT model, and demonstrates that BERT has the ability to detect subtle forms of toxicity, also in Norwegian. We compare how the different newly released Norwegian BERT models perform when fine-tuned on our dataset, and we also experiment with how English data can be utilized to fine-tune one of the models. We fine-tune BERT to recognize unhealthy comments in Norwegian, as well as a list of other characteristics a comment may have such as being hostile, antagonising/insulting/trolling, dismissive, condescending, sarcastic, or being an unfair generalisation. Our AUC scores beat the AUC scores from previous work on detecting unhealthy comments in English on all categories, except dismissive.

Acknowledgements

Thank you to my supervisor Jonas Nordhaug Myhre for all help throughout the writing of this thesis. Thank you also to the annotators who created labels for the dataset. Thank you to Per Egil for advice on how to use the Norwegian BERT models he created. I also wish to thank my friends and family, including the friends who have held me with digital company during the COVID pandemic.

Contents

1 Introduction

The internet allows anyone to engage in a discussion on any topic. At the turn of the century there was a lot of optimism regarding the possibilities of open and democratic debate through online fora (Quandt, 2018). However, online debates can easily turn sour through the use of personal attacks, hostility, and other forms of aggression. When discussions devolve to such levels, they are seldom useful, and so it is desirable keep conversations friendly in order for them to serve their purpose. Analysis of online discussions is a field of research including detection of toxicity (Risch and Krestel, 2020) and hate speech (Jensen, 2020; Saleem et al., 2017), how people change their mind through online debate (Tan et al., 2016), and studies of the users themselves (Wang et al., 2016; Nadim et al., 2021).

The prevalence of undesirable behavior online that we don't see as often in the offline world is related to the fact that some people have a tendency to say and do things online that they would not say or do in a face-to-face situation. This phenomenon has been given the name *the online disinhibition effect* (Suler, 2004). Pew research found that in 2021, 41% of Americans have experienced some form of online harassment, and that online harassment has intensified since their last survey in 2017 (Vogels, 2021). The same survey also shows that the percentage of Americans who have experienced more severe types of harassment such as physical threats or sexual harassment has increased since then. The Equality and Anti-Discrimination Ombud of Norway found that either 7 or 9 percent of comments on the Facebook pages of NRK and TV 2 were hateful, depending on the definition of a hateful comment (Amna Veledar, 2018).

Moderation is often necessary to avoid too many toxic comments which degrade the quality of the conversation in online fora. Nadim et al. (2021) make a study of the participants of heated online debates, and they reveal that the participants themselves see moderation in some form as necessary. The large amounts of text that is produced every day at popular fora makes moderation challenging. In fact one of the sub-communities on the forum from which text was gathered to make the dataset presented and used in this work, was closed because the users kept breaking the rules[1]. Automated moderation tools can ease the burden on moderators. This could in turn lead to better conversations online, and make it unnecessary to shut down the places where discussions on sensitive topics take place.

Advances made in the field of Natural Language Processing (NLP) in recent years through Deep Learning (DL) techniques, open up new possibilities for creating such automated moderation tools. In this thesis we train a deep learning model, specifically a BERT model (Devlin et al., 2019), to detect unhealthy comments, and also to detect comments displaying a range of other characteristics that may be regarded as sub-attributes of unhealthy comments. Deep learning models require large amounts of data in order to tune their parameters to achieve good results, something which is not easily available in low-resource languages such as Norwegian. Fortunately newer language models such as BERT

[1]https://vgd.no/utdebattert/innvandring-rasisme-og-flerkultur/tema/1866749/tittel/kategorien-stenges

make use of transfer learning, where a model is first trained on easily accessible unlabeled data, to later be fine-tuned on a task-specific dataset, which need not be so large. Three Norwegian BERT models have recently been released, and we fine-tune them to classify comments into the categories *unhealthy, sarcastic, hostile, condescending, unfair generalisation, dismissive*, and *antagonistic*, using a dataset we created.

The three Norwegian BERT models which were recently released are: NB-BERT-Base and NB-BERT-Large from The National Library of Norway[2], and NorBERT from the Language Technology Group at the University of Oslo[3]. Access to BERT models which have been pre-trained on large amounts of Norwegian text is a big step forward for Norwegian NLP. Based on conversations with the main contributor to the creation of the NB-BERT models, we speculate that these models may be able to learn from English data. In order to assess the models' bi-lingual capabilities, we do some exploratory experiments where we fine-tune Norwegian BERT models on English data. Successfully learning from English data opens up for the use of Cross-Lingual Transfer learning, where a labeled dataset in one language can be used to train a model to perform tasks in another language. Norwegian BERT model capable of learning from English data opens up many possibilities for Norwegian NLP, as there are many datasets available in English.

In this thesis we are not concerned with finding toxicity, or hate speech. Instead, we try to detect the more general category of "unhealthy" comments, and the subtle indicators which may or may not be sub-attributes of unhealthy comments, such as sarcasm, hostility, and all the other categories previously mentioned. The category of unhealthy comments includes hateful and toxic comments, but also other comments which are not made in good faith, and do not invite engagement. The reason for choosing to focus on unhealthy comments in general, and also on these subtle "sub-attribute" categories, is that we want to see if a BERT model is able to assess the quality of an online comment, beyond whether it is toxic/hateful or not. Once a conversation deteriorates to include hate speech, or is in some other way overtly toxic, the conversation is most likely never getting back to a place where it can be useful. Because of this, if one can detect and try to prevent toxicity, or things that leads to toxicity, before it occurs, that is preferable to removing such comments after the fact. Instead of a moderation tool one might imagine that a prompt with a warning such as "This comment may be seen as hostile" shows up when the author of a comment presses send, if their comment is seen as hostile. This would give the author a chance to rethink and/or rephrase their comment, effectively moderating themselves.

Fine-tuning BERT models for detecting undesirable comments of any kind requires a dataset with examples of such comments. Datasets like that are not easily available, especially in Norwegian. We contribute to this area by providing a new dataset of labeled comments. When looking for hateful comments or comments with some other attributes such as the more general "unhealthy"

[2]https://github.com/NBAiLab/notram/
[3]http://wiki.nlpl.eu/Vectors/norlm/norbert

category, whether a comment displays such a characteristic or not is subjective in nature. This makes even the dataset creation a challenging task, as the labels are unavoidably influenced by the annotator's biases. Some measures have been taken to promote more agreement between annotators, as will be described later in the thesis. The subjective nature of the labels also makes this a very challenging problem for machine learning models to solve. Our goal in this thesis is to evaluate how well a BERT model can learn to classify comments into these subjective categories.

1.1 Related work

Jensen (2020) used anomaly detection methods to detect hate speech in Norwegian, and created a dataset of examples of such hateful comments to do so. The comments are divided into 5 categories: (1) neutral, (2) provocative, (3) offensive, (4) moderately hateful, and (5) hateful. We include some of these comments in our dataset for our annotators to give new labels. Mixing in these comments allows for the study of overlap between labels, which lets us compare the two datasets.

Øvrelid et al. (2019) created NoReC_fine, a sentiment dataset for Norwegian which builds on the original NoReC from Velldal et al. (2017). NoReC contains reviews on movies, restaurants, music, product reviews, as well as several other things. From NoReC_fine was also derived NoReC_sentence, which includes binary labels into positive sentiment and negative sentiment. We utilize this dataset to investigate whether NB-BERT can learn to do sentiment analysis in Norwegian, when learning from an English sentiment dataset.

The work done in this thesis is inspired by Price et al. (2020), who created the Unhealthy Comment Corpus (UCC), of 44,000 comments with crowdsourced labels and confidence scores. Each comment is classified as either "belonging in a healthy conversation" or not, as well as labels for whether or not the comments are (1) hostile, (2) antagonistic, insulting, provocative, or trolling (summarized as the label "antagonistic"), (3) dismissive, (4) condescending or patronizing (summarized as the label "condescending"), (5) sarcastic, or (6) an unfair generalisation. We aim to create a similar dataset, but in Norwegian. The annotation guideline given to our annotators was therefore a translated version of the annotation guideline in (Price et al., 2020). They find that their baseline BERT model outperforms humans on detection of all attributes, with the exception of sarcasm.

1.2 Thesis Structure

This thesis starts with a description of Krippendorff's α which is used as a reliability measure on the dataset we create. After that comes some background theory about machine learning, first mentioning some machine learning techniques that are common among many or all machine learning approaches, such as how to best utilize the dataset, and how to deal with imbalanced data. Then comes a description of the neural network approach to machine learning and its application to natural language processing. We will start with a simple feedforward neural network and work up to the BERT model which is used in the experiments reported in the experiment section. Word embeddings and recurrent neural networks are also covered along the way, as well as the ELMo model.

Following the background material is an explanation of how the dataset was created, from the scraping process, to the cleaning of the text, to the choices made in the annotation process. Some statistics about the dataset are also provided, and the dataset is compared to two other datasets, namely the Unhealthy Comments Corpus from (Price et al., 2020), and the Norwegian hate speech dataset from (Jensen, 2020). After presenting the dataset comes a description of the experiments done using the new dataset and the results of those experiments, as well as discussion about the results. After that we try utilizing the hate speech dataset from (Jensen, 2020), and the UCC from (Price et al., 2020) to see if we could use that data to improve our initial results. As a proof of concept we also fine-tune NB-BERT for sentiment analysis using English data, to see if it could perform sentiment analysis on Norwegian data afterwards.

2 Background material

It is common to give a measure of agreement between annotators when presenting a new dataset. We choose to use Krippendorff's Alpha for this measure, and the background material therefore starts with a description of it. After that comes a description of machine learning and some common techniques used within machine learning, such as how to use the available data to train and evaluate machine learning models. In this thesis we will be concerned with the neural network approach to machine learning, and so an explanation of how neural networks work, how they learn, and the tools used to make them learn efficiently, is also provided. After that we present the recurrent neural network, and some of its variants, before we move on to natural language processing and the machine learning architectures responsible for the recent progress in that field.

2.1 Krippendorff's Alpha

There are several metrics for measuring the agreement between annotators, including Cohen's kappa (Cohen, 1960), and Fleiss' kappa (Fleiss, 1971). Krippendorff (2011) argues that these other reliability coefficients are specialized, and that Krippendorff's alpha, which we will refer to as α, is a generalization of several known reliability indices, while also being applicable to many varieties of data. α works for (Krippendorff, 2011)

- Any number of observers
- Any number of categories, scale values, or measures
- Any metric or level of measurment
- Incomplete or missing data
- Large and small sample sizes

The general from of α is

$$\alpha = 1 - \frac{D_0}{D_e}$$

where D_0 is the observed disagreement among labels, and D_e is the expected disagreement if the labels were randomly assigned. α mostly ranges between 0 and 1, but can also be negative. $\alpha = 1$ indicates perfect reliability while $\alpha = 0$ means labels overlap as if random chance had produced them. A negative value of α indicates systematic disagreement.

We will now show an example from (Krippendorff, 2011), of the calculation of Krippendorff's alpha for binary data with two observers. First construct a reliability matrix. If for example two annotators label $N = 10$ datapoints as either 0 or 1, one could end up with a reliability data matrix like this:

The next step is to create a coincidence matrix, which accounts for all values contained in the reliability matrix:

Datapoint	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Annotator 1	0	1	0	0	0	0	0	0	1	0
Annotator 2	1	1	1	0	0	1	0	0	0	0

$$\begin{array}{c|cc|c} & \mathbf{0} & \mathbf{1} & \\ \hline \mathbf{0} & o_{00} & o_{01} & n_0 \\ \mathbf{1} & o_{10} & o_{11} & n_1 \\ \hline & n_0 & n_1 & n = 2N \end{array}$$

In the coincidence matrix, each datapoint's label is entered twice, once as **c-k** pairs and once as **k-c** pairs, where **k** and **c** refer to each annotators label. For example datapoint 1 is entered both as **0-1** and as **1-0**, while datapoint 2 is entered twice as **1-1**.

For our example we get

$$\begin{array}{c|cc|c} & \mathbf{0} & \mathbf{1} & \\ \hline \mathbf{0} & 10 & 4 & 14 \\ \mathbf{1} & 4 & 2 & 6 \\ \hline & 14 & 6 & 20 \end{array}$$

The α-reliability is then computed as

$$\alpha = 1 - \frac{D_0}{D_e} = 1 - (n-1)\frac{o_{01}}{n_0 \cdot n_1}$$

which for our example evaluates to

$$\alpha = 1 - (20-1)\frac{4}{14 \cdot 6} = 0.095$$

(Krippendorff, 2018, p.241) recommends to only rely on variables where $\alpha > 0.8$, and to only draw tentative conclusions when considering variables with $0.667 < \alpha < 0.8$.

2.2 Machine Learning

Machine Learning (ML) can be defined as programming computers to optimize a performance criterion using example data or past experience (Alpaydin, 2014). When a model is defined with a certain set of parameters, we say that the model *learns* when it optimizes these parameters using training data or past experience, in order to perform better on the performance criterion. What this means in practice is that one shows the model many examples, and tell the model what the output should be, and the model updates itself to find the patterns that lets it classify the examples correctly. We will soon look at the details of how this is done. While traditional Artificial Intelligence (AI), which machine learning is a subfield of, can solve problems that can be described with a set of mathematical rules, some problems are not so easily translated into mathematics. To solve these types of problems, we can use machine learning.

One subfield of machine learning, Deep Learning (DL), has exploded in popularity in recent years. Deep learning models are machine learning models with a high number of parameters. One major reason for the increased popularity of DL models is that as technology grows more and more present in modern society, enormous datasets grow along with it. Highly relevant for this thesis is the text data that can be gathered from debate forums. Another major reason for the growth of deep learning as a field is the fact that computers have grown powerful enough that we can employ very large models with millions and some times billions of parameters, to process a lot of data.

2.2.1 Training, validation, and test data

When doing machine learning, it is common to partition the available data into two or three parts. The largest chunk of data is used for the training dataset. This is the set of examples the model will look at and tune its parameters according to. ML models tend to learn to separate the classes in the training set very well, but the ultimate goal is not to have a model which performs well on the training data, but to have a model which performs well generally, also on unseen examples. A model's performance on unseen examples is called *generalization*. (Alpaydin, 2014, p.39). We want good generalization from our model, and so we try to avoid so-called *overfitting*, which is where the model simply memorizes the training data, and hasn't actually found any general patterns. One can also think of this as the model learning the noise in the dataset (Alpaydin, 2014, p.39). The model is therefore tested on a separate holdout set of data, called the *test* dataset.

The test dataset should not be used in any way to train the model, so in order to evaluate the performance of the model during development, one may extract a *validation* dataset from the remaining training data. The model does not train on the validation set per se, but the machine learning engineer who is building the model can use it to estimate different models' generalization ability. Different models may refer to different model architectures, such as doing a linear regression versus training a neural network. It can also refer to different sets of *hyperparameters* used to train the same model architecture multiple times. Hyperparameters are parameters which the model does not tune itself, and instead are defined by the programmer (Goodfellow et al., 2016, p.117). A simple example of a hyperparameter is the learning rate used to train a neural network. To tune this hyperparameter, one would train the model using the training data with a certain learning rate, evaluate the model's performance on the validation dataset, try another learning rate evaluate on the validation dataset, and repeat. The learning rate that perform best on the preferred metric, which could for example be the accuracy, is used for the final training.

2.2.2 Cross validation

K-fold cross validation is a method of evaluating a model where the data is partitioned k times into different sets of training and validation data. This gives k different models, each trained and evaluated on its own unique partition of the data. The mean and standard deviation of the results from the different models can also give a better idea of the model's general performance. K-fold *stratified* cross validation is the same, except one makes sure to keep the ratio of labels in the two partitions equal. So if the dataset has binary labels where 80% of the examples belong to class 1 and only 20% belong to class 2, stratified cross validation ensures that all validation and training sets created during cross validation keep this ratio of labels. This is useful in cases where the dataset is imbalanced.

2.2.3 Imbalanced data

When dealing with imbalanced datasets, a model can often perform well on a performance criterion by always predicting the majority class. The minority class is often the most important or interesting one, but if the dataset is sufficiently imbalanced, the model might ignore it all together. Heavily imbalanced datasets affect both the model's ability to converge during the training phase, and the generalization of the model when running inference on a test set (Buda et al., 2018). Two techniques seek to alleviate this problem by balancing the occurrence of each class: oversampling, where examples from the minority class are duplicated, and undersampling, where random examples from the majority class are deleted (Buda et al., 2018).

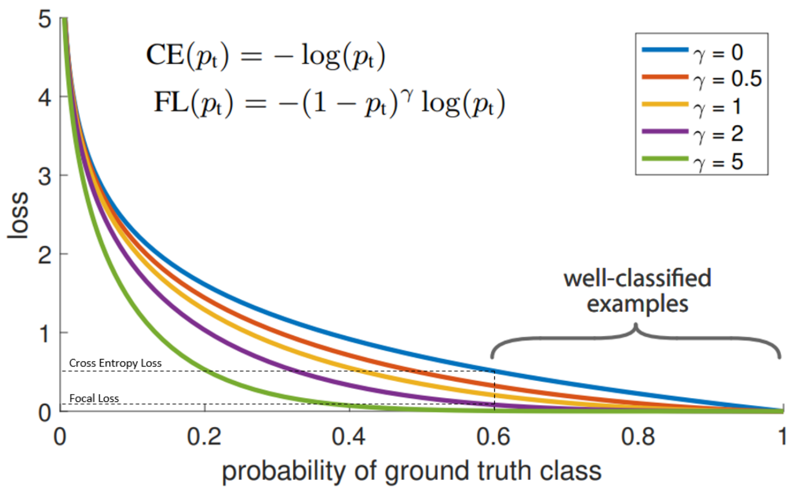


Figure 2.1: How different values of γ affect the focal loss curve. A higher value of γ gives less focus to well classified examples. Figure from (Lin et al., 2017).

Another way to prevent an imbalanced dataset from impeding model performance is through cost sensitive learning, where one assigns different costs to misclassification of different classes (Buda et al., 2018). An example of this is weighted loss.

Lin et al. (2017) introduced *focal loss*, where the contribution to the loss from well-classified examples is down-weighted by adding a modulating factor $(1-p_t)^\gamma$ to the cross entropy loss:

$$FL(p_t) = -(1 - p_t)^\gamma log(p_t)$$

where p_t is the probability the model has assigned to the ground truth class. Visualization of the focal loss for different values of γ is shown in Figure 2.1.

2.2.4 Transfer Learning

Transfer learning refers to when what has been learned from training on one task, can be used to perform better on another task (Goodfellow et al., 2016, p.526). We can call training on the first task the *pre-training* stage (Goodfellow et al., 2016, p.314). For this to work, it is necessary that some of the patterns the model found to solve the pre-training task are useful also for solving the final task. Transfer learning can often be very useful when it allows for the utilization of more data. This is especially true if the pre-training task does not require labeled data, and can instead be trained in an unsupervised manner, since unlabeled data is much more easily available.

2.2.5 Classifier Performance Metrics

	Predicted class		
True class	Positive	Negative	Total
Positive	tp: true positive	fn: false negative	p
Negative	fp: false positive	tn: true negative	n
Total	p'	n'	N

Table 2.1: Confusion matrix. From Alpaydin (2014).

In order to evaluate and compare different classifiers, we need to measure their performance on the test set. In a binary problem, there are four possible outcomes for a given test example. If the true label is positive, and the model predicts it to be positive, we call it a *true positive*. If the model predicts a positive example to be negative we call it a *false negative*. On a negative example, if the model correctly predicts it to be negative, it is a *true negative*, and if the model incorrectly predicts the negative sample to be positive, that is a *false posisitve* (Alpaydin, 2014). True positivies, false negatives, false positives, and true negatives are often summarized in a *confusion matrix*, as shown in Table 2.1.

Often we would like to summarize the results of testing in a single number. Different metrics fit different situations. Some commonly used metrics are defined in Table 2.2. If a model returns an estimated probability $\hat{P}(C_1|x)$ for an example x belonging to the positive class, we have to decide on a threshold θ for which examples we predict as positive, so that the output is positive if $\hat{P}(C_1|x) > \theta$. If we want fewer false positives we can set θ closer to 1, but this will come at the cost of fewer true positives. Likewise setting θ closer to

Name	Formula
error	(fp+fn)/N
accuracy	(tp+tn)/N
sensitivity	tp/p
specificity	tn/n
precision	tp/p'
recall	tp/p

Table 2.2: Metrics used in binary classification problems. From Alpaydin (2014).

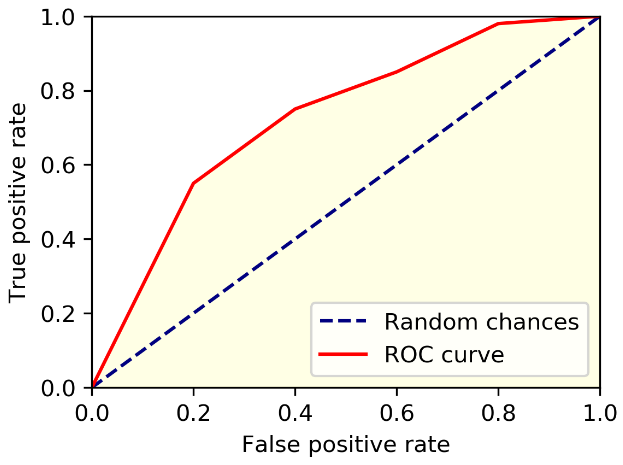


Figure 2.2: Example of a ROC curve. The colored area represents the AUC score. The dotted blue line represents the performance of a classifier making random guesses. Image taken from Bui (2020)

0 will give us more true positives, but also more false positives. To summarize a model's performance across different values of θ, it is common to look at the *Receiver Operating Characteristics* curve, or ROC curve, where one plots the true positive rate versus the false positive rate. An example can be seen in Figure 2.2. To summarize the curve in to a single number one calculates the area under the curve (AUC). A perfect classifier has an AUC of 1.

2.2.6 Feedforward Neural Networks

The core component of all machine learning models we will be concerned with in this thesis is the feedforward neural network (also called a multilayer perceptron, artificial neural network, or simply neural network), which draws inspiration from how the brain works (Haykin, 1999). It consists of layers of artificial neurons, connected by weights, as seen in Figure 2.3. Each neuron also has a bias. The weights are randomly initialized, and updated using some form of gradient descent, after the gradient has been found using backpropagation. Neural networks, gradient descent, and backpropagation form the basis for all

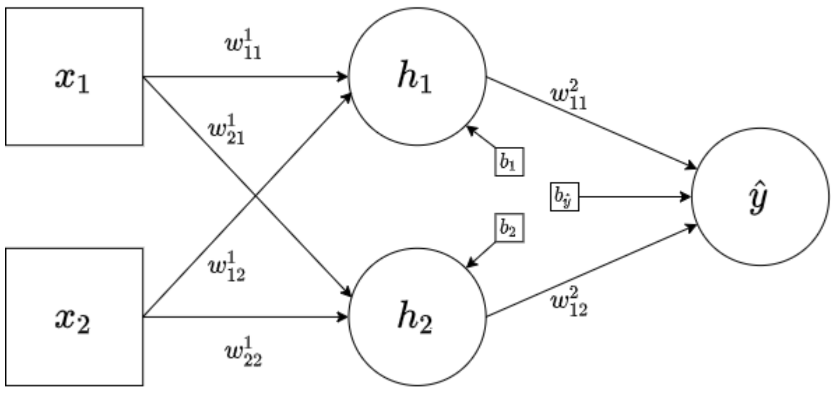


Figure 2.3: A feedforward neural network with an input layer consisting of two input units, a single hidden layer consisting of two hidden units, and one output unit in the output layer. Weight w^r_{jk} connects unit j from layer r with unit k from layer $r-1$. The output unit and all hidden units also have a bias associated with them.

architectures we will look at in this thesis, so it makes sense to start with a brief explanation of them. We will start by going through the standard feedforward neural network to understand what is happening at each step, and then move on to explain gradient descent and finally a quick mention of what the back-propagation algorithm does. The explanations of these things will be based on (Theodoridis and Koutroumbas, 2008), (Goodfellow et al., 2016), and (Haykin, 1999).

Still using Figure 2.3 as our reference, we say that first some input is given to the input neurons, x_1 and x_2. We feed in the data points simultaneously, i.e. we input the vector $[x_1 \quad x_2]^\top$ to the network. The input values are multiplied by their corresponding weights, and the results of these multiplications are then added together with the bias to form the *activation potential* for the next state (Haykin, 1999). So for the activation potential of the hidden states in our reference network, we get

$$v_1 = x_1 w^1_{11} + x_2 w^1_{12} + b_1 \tag{1}$$
$$v_2 = x_1 w^1_{21} + x_2 w^1_{22} + b_2 \tag{2}$$

This can be written in terms of matrix operations:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} w^1_{11} & w^1_{12} \\ w^1_{21} & w^1_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Being able to write the necessary calculations in terms of matrix operations is key for being able to perform the calculations in a practical amount of time when the number of parameters (weights) grows to the order of hundreds of millions. In order to go from the activation potential to the hidden state, we need to send it through an *activation function*, $f(x)$. Some examples of activation functions are the *tanh* function, the *sigmoid* function, and the *ReLU* function. The activation function is what introduces nonlinearity into our network, allowing it to differentiate between classes that are not linearly separable. Our expression for the states of the hidden layer, is then

$$\begin{aligned} h_1 &= f(v_1) = f(x_1 w_{11}^1 + x_2 w_{12}^1 + b_1) \\ h_2 &= f(v_2) = f(x_1 w_{21}^1 + x_2 w_{22}^1 + b_2) \end{aligned}$$

or, if we define

$$\boldsymbol{h} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \quad \boldsymbol{W}_1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

we can write

$$\boldsymbol{h} = f(\boldsymbol{v}) = f(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b})$$

Now that we have the hidden states, h_1 and h_2, we continue propagating forward through the network by treating these states as the input for the next layer. Defining

$$\boldsymbol{W}_2 = \begin{bmatrix} w_{11}^2 \\ w_{12}^2 \end{bmatrix}$$

we then have

$$\hat{y} = f(\boldsymbol{W}_2^\top \boldsymbol{h} + b_{\hat{y}})$$

as our output. This is the network's estimate of what y should be, based on the input x. Propagating the input through the network like we have just done is known as *forward propagation* (Goodfellow et al., 2016, p.197). Since the weights are randomly initialized, the result of the first forward propagation will almost certainly be a very poor estimation. The network is essentially just making a random guess. We can evaluate how bad a guess it is by defining a loss function (also called a cost function or error function), which takes in the model's prediction $\hat{y}$ and the ground truth y and outputs some scalar value J.

An example might be the squared error:

$$J(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2 \tag{3}$$

The factor of $1/2$ in equation 3 is there so the derivative of J with respect to $\hat{y}$ is simply $y - \hat{y}$. Usually we will have more than one data point to consider when calculating the cost function. In that case we could use the mean squared error:

$$J(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{2N}\sum_{i=1}^{N}(y - \hat{y})^2 \tag{4}$$

We want J to be large when the model makes bad predictions, and lower when the model makes good predictions. Modern neural networks mostly use the negative log-likelihood as the loss function (Goodfellow et al., 2016, p.173). We will now see how the network can use the output of the loss function to learn to make better guesses, by updating the weights through backpropagation and gradient descent.

2.2.7 Gradient Descent and Backpropagation

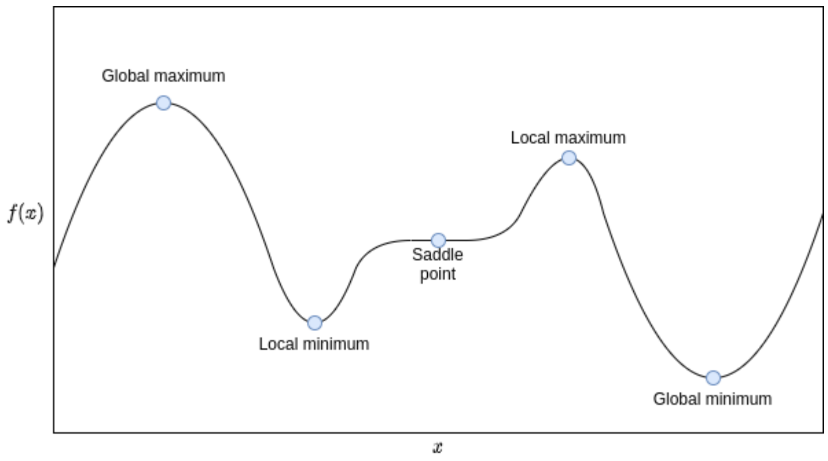


Figure 2.4: Illustration of the different types of critical points, where $f'(x) = 0$. At a minimum, $f(x)$ is lower than all neighboring points. At a maximum, $f(x)$ is higher than all neighboring points. A *global* maximum and minimum is where the function takes its absolute highest and lowest values, respectively. A saddle point is neither a maximum or a minimum, but still $f'(x) = 0$.

Gradient descent is a technique used to minimize a function, by taking small steps in the opposite direction of the gradient of the function (Goodfellow et al., 2016). To understand how this works, consider first a simple example where we want to minimize with respect to x the function

$$y = f(x), \quad x, y \in \mathbb{R}$$

This means that we want to find the value of x which gives us the smallest value for y. To get started in finding this x-value we can input some random number $x = x_0$ into the derivative of $f(x)$, to get the slope of $f(x)$ at that point which we denote as $f'(x_0)$. If the slope is positive, that means that an increase in x will lead to an increase in y, and since we want to minimize y, we move in the opposite direction of the slope, i.e. we take a small step in the direction of negative x:

$$x_1 = x_0 - \epsilon$$

where ϵ is some small positive number. If the slope is negative, we take a small step in the direction of positive x:

$$x_1 = x_0 + \epsilon$$

More generally, we update the input x as:

$$x_{i+1} = x_i - \epsilon \ \mathrm{sign}(f'(x_i))$$

(Goodfellow et al., 2016). After updating x using this formula, we insert the new value of x into the derivative to see whether we should keep going in that direction or not. The method has converged to a so-called *critical point* when $f'(x_i) = 0$. A critical point can be a local or global maximum or minimum, or a saddle point, all of which are shown in Figure 2.4. Ideally we would like to end up in the global minimum, but with gradient descent we risk getting stuck in a local minimum or a saddle point, since the algorithm is over as soon as $f'(x) = 0$.

That was gradient descent in the simple case of a function with scalar input and output. Let us now look at gradient descent in the more relevant setting where we have a vector input, but still maintain a scalar output. Having a scalar output is necessary for the concept of minimization to still make sense (Goodfellow et al., 2016). We will use a two dimensional example, looking at the function

$$y = f(\boldsymbol{x}), \quad y \in \mathbb{R}, \quad \boldsymbol{x} \in \mathbb{R}^2$$

Our goal is still to find the input $\boldsymbol{x}$ which gives us the lowest y possible. When we had a scalar input, we simply started with a random x and looked at the derivative to find which direction to go, took a small step in that direction, and looked at the derivative again with the new value of x, and repeated this until convergence. We use the same logic here, but in this case, we don't have a simple $f'(x)$. Since our input is vector valued, we need to take partial derivatives of the function with respect to each component of $\boldsymbol{x}$. The partial derivative of f with respect to x_i, which we can denote as $\partial_{x_i} f$, tells us how the value of f changes when we take a small step in the direction of x_i. We gather the partial derivatives of f with respect to both (more generally, all) components of $\boldsymbol{x}$, in a vector called the gradient, denoted $\nabla_{\boldsymbol{x}} f$. With a two-dimensional input, our gradient is then

$$\nabla_{\boldsymbol{x}} f = [\partial_{x_1} f \quad \partial_{x_2} f]^\top$$

The gradient at a given point always points directly uphill, and the negative gradient then points directly downhill (Goodfellow et al., 2016). Analogous to the case with one-dimensional inputs, we can then minimize $y = f(\boldsymbol{x})$ by taking small steps in the direction of the negative gradient, reevaluate the gradient, take a new small step, and repeat until convergence. Convergence in this case means that *all* partial derivatives are equal to zero, i.e. the gradient is a zero vector.

A neural network can be thought of as a complex function with many parameters (weights). We utilize gradient descent to train neural networks by minimizing the cost function with respect to the weights of the network. As we have seen, in order to do that we first need to find the gradient of the cost function with respect to the weights. It is straightforward to find an analytic expression for the gradient, using the chain rule of calculus, but the problem with this approach is how computationally expensive it can be to evaluate it (Goodfellow et al., 2016). The backpropagation algorithm allows us to compute the gradient in a simpler and less expensive manner. It was first introduced by Rumelhart et al. (1986), and it is an algorithm that recursively applies the chain rule efficiently. The idea is that when computing the gradient, many subexpressions may be repeated, and recomputing them every time is inefficient. It is much better to save them in memory, and reuse them when needed, which is what backpropagation does.

That's the basic idea of how to train a feedforward neural network: the model is given some input $\boldsymbol{x}$ which it uses along with its weights to produce an output $\hat{\boldsymbol{y}}$. The model's output is compared with the ground truth $\boldsymbol{y}$ through a cost function C, which is backpropagated through the network to find the gradient of C with respect to the weights of the network. After obtaining the gradient for each weight through backpropagation, gradient descent updates the weights in such a way that the model's output will be closer to the ground truth. This is repeated this until gradient descent converges, which happens when the gradient with respect to the weights is a zero vector (or close to a zero vector, in practice you rarely get an actual zero vector). Given a training dataset, we would like to use backpropagation and gradient descent to train the network to find some pattern in the data, so that it can differentiate between the classes, and correctly classify the datapoints as one class or another. The feedforward neural network is the basic building block for much more complex models, as we will see later in the thesis when some deep learning architectures are described. First, however, we will take a look at some more tools used to train deep learning models, starting with a look at the cross entropy loss.

2.2.8 Cross Entropy Loss

Our discussion of cross entropy starts with a definition of the Kullbeck-Leibler divergence, which measures the difference between two probability distributions, $P(x)$ and $Q(x)$. It is defined as (Goodfellow et al., 2016, p.72)

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

When we train a neural network model with an example $\boldsymbol{x}$, we wish to minimize the distance between that datapoint's label $y(\boldsymbol{x})$, and the model's output when it is fed that particular datapoint, $\hat{y}(\boldsymbol{x})$. If the model produces a score vector $\boldsymbol{z}(\boldsymbol{x})$, one score for each possible output class, that score vector can be input into a *softmax* function to produce a probability distribution over the possible output classes:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Let the output of the softmax equal Q from our definition of KL-divergence. Likewise, let the label of the example, $y(\boldsymbol{x})$ be interpreted as a distribution giving 100% probability to the true class, and 0% probability to the other classes, in the case of single-label classification, and call it P. In order to get the model to output a probability distribution as close to the label as possible, we wish to minimize the KL-divergence $D_{KL}(P||Q)$, with respect to the model's output, Q. However, Q does not participate in P, so this is equivalent to minimizing the following expression with respect to Q (Goodfellow et al., 2016, p.73):

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

$H(P, Q)$ is the *Cross Entropy.* Importantly, when we use cross entropy as a loss function, x in the above equation represents the different possible classes, not the input to the model. We interpret $Q(j)$ as the model's predicted probability of the input example $\boldsymbol{x}$ belonging to class j. To avoid confusion we therefore replace x with j in further equations, where j represents different classes. We can write out the cross entropy as follows:

$$H(P, Q) = -\mathbb{E}_{j \sim P} \log Q(j) = -\sum_j P(j) \log Q(j)$$

Here, $P(j) = 1$ when j is equal to the true class y, and 0 otherwise. The sum therefore collapses and we are left with

$$H(P, Q) = -\log Q(y)$$

If we wish to weight the loss, we create a weight function W, where $W(j)$ returns the weight given to class j. The weighted loss is then equal to:

$$H(P, Q)_{\text{weighted}} = -W(y) \log Q(\boldsymbol{x})$$

2.2.9 Optimization

In Section 2.2.6 and Section 2.2.7 an explanation was given of the basic idea behind neural networks and how they are trained, where the loss is minimized using gradient descent. In practice, pure gradient descent is not used for training deep models, because the gradient can only be calculated after seeing the whole dataset, which is often large, and so each step of gradient descent would take a very long time (Goodfellow et al., 2016, p.148). Instead of doing this, one estimates the value of the gradient by backpropagating the loss from a minibatch of examples, and updates the weights with this estimate. This is called Stochastic Gradient Descent (SGD) (Goodfellow et al., 2016, p.148).

Momentum Further innovations have been made on SGD, such as momentum, where a fraction γ (called the momentum term) of the update vector at the previous time step is added to the update vector at the current time step (Ruder, 2017).

$$m_t = \gamma m_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - m_t$$

Adam Several more innovations have been made, from Nesterov accelerated gradient (Nesterov, 1983), to the one used in this work, Adam (Kingma and Ba, 2017). Adam is short for Adaptive Moment Estimation, and it computes adaptive learning rates for each parameter (Ruder, 2017). Adam computes estimates for the first and second moment of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_2^2$$

Since m_t and v_t are initialized as zero-vectors, Kingma and Ba (2017) observe that they are biased towards zero. They correct for this bias by replacing the first and second moments by bias-corrected as such:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The model's parameters are then updated using this rule:

$$\theta_{t+1} = \theta_t - \frac{\nu}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Weight Decay L_2 regularization is a way of penalizing a model for the size of its weights. The square of the norm of the model's weights is added to the loss, which could be for example the mean squared error loss (Goodfellow et al., 2016, p.116).

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^T \boldsymbol{w}$$

This incentivises the model to have smaller weights as it tries to minimize the loss. Smaller weights regularize the model, meaning it is less likely to overfit to the training data.

Weight decay is a term often used interchangeably with L_2 regularization, as they are equivalent when training with regular SGD, but Loshchilov and Hutter (2019) showed that they are *not* equivalent when using adaptive gradient methods, such as Adam. They therefore proposed a new method for doing weight decay with Adam, which they call Adam decoupled weight decay (AdamW). The difference between AdamW and Adam with L2 regularization is shown in Figure 2.5.

```
Algorithm 2 Adam with L$_2$ regularization and Adam with decoupled weight decay (AdamW)
1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment
   vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: repeat
4:     $t \leftarrow t + 1$
5:     $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow$ SelectBatch($\boldsymbol{\theta}_{t-1}$)    ▷ select batch and return the corresponding gradient
6:     $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$ $+\lambda\boldsymbol{\theta}_{t-1}$
7:     $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$    ▷ here and below all operations are element-wise
8:     $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
9:     $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1-\beta_1^t)$    ▷ $\beta_1$ is taken to the power of $t$
10:    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1-\beta_2^t)$    ▷ $\beta_2$ is taken to the power of $t$
11:    $\eta_t \leftarrow$ SetScheduleMultiplier($t$)    ▷ can be fixed, decay, or also be used for warm restarts
12:    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left( \alpha\hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon) +\lambda\boldsymbol{\theta}_{t-1} \right)$
13: until stopping criterion is met
14: return optimized parameters $\boldsymbol{\theta}_t$
```

Figure 2.5: AdamW compared to Adam with L2 regularization. Algorithm from (Loshchilov and Hutter, 2019).

Learning Rate Scheduling The learning rate does not need to stay constant, and can instead be scheduled to change over the course of training. Mosbach et al. (2020) recommends a linear increase of the learning rate for the first 10% of training steps, and a linear decay to zero afterwards.

2.2.10 Recurrent Neural Networks

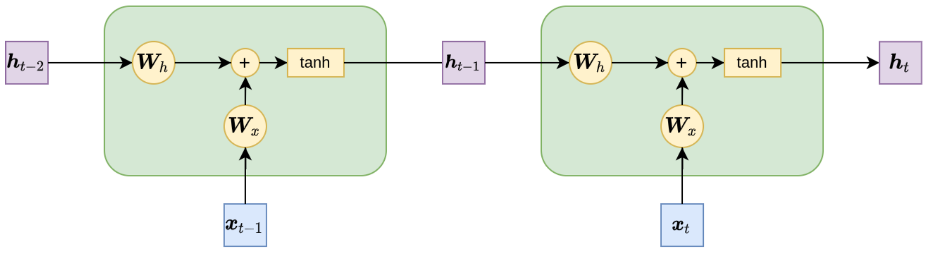


Figure 2.6: A vanilla RNN unrolled for two time steps. The input vector and the hidden state are multiplied by their respective weights and added together. The result of this addition is sent through a *tanh* activation function which produces the new hidden state.

We now move on to the Recurrent Neural Network (RNN), which is a machine learning architecture that utilizes neural networks, and naturally lends itself to the processing of sequential data, such as text, which is a sequence of words. I will briefly describe the general RNN architecture and then move on to describe an improvement upon it, namely the addition of *gates*, as gated RNNs were the go-to state of the art models before the Transformer models. An RNN works as follows: first randomly initialize a vector called the *hidden state* $\boldsymbol{h}_0$. Then use this hidden state, together with the first input, $\boldsymbol{x}_1$, to calculate a new hidden state $\boldsymbol{h}_1$. In a vanilla RNN, this is done through the formula

$$\boldsymbol{h}_1 = f(\boldsymbol{W}_h\boldsymbol{h}_0 + \boldsymbol{W}_x\boldsymbol{x}_1 + \boldsymbol{b})$$

where f is some non-linear activation function, often the *tanh* function, $\boldsymbol{W}_h$ and $\boldsymbol{W}_x$ are weights to be used on the hidden state and the input, respectively, and $\boldsymbol{b}$ is the bias. This new hidden state $\boldsymbol{h}_1$, can be fed *back into the same model*, to be used along with the next data point, $\boldsymbol{x}_2$, to calculate the next hidden state, which is then fed back into the model again, etc. A diagram of a vanilla RNN like this can be seen in Figure 2.6. The more general formula for the hidden state at time step t is

$$\boldsymbol{h}_t = f(\boldsymbol{W}_h\boldsymbol{h}_{t-1} + \boldsymbol{W}_x\boldsymbol{x}_t + \boldsymbol{b})$$

The weights are the same at every time step, which means that RNNs utilize *parameter sharing.* This allows the model to process data of whatever length, and to generalize across them (Goodfellow et al., 2016).

The idea behind this architecture is that the model should learn to encode a summary of what it has seen previously in the sequence, into $\boldsymbol{h}_t$. Then that summary can for example be sent into a regular feedforward neural network which can classify the sequence to some class. It is also possible to send the hidden state at every time step through a feedforward neural network, in order to classify each data point on its own, instead of the sequence as a whole. The hidden state will be a lossy summary, because the hidden state $\boldsymbol{h}_t$ is a vector of fixed length, while the sequence can be of an arbitrary length. The RNN learns through backpropagation and gradient descent just like the standard feedforward neural network. However, long-term dependencies are hard to model with a vanilla RNN such as this. We can see why by looking at a simple example, given in (Goodfellow et al., 2016). Consider a very simple recurrent neural network on the form

$$\boldsymbol{h}_t = \boldsymbol{W}^\top\boldsymbol{h}_{t+1}$$

Because this is simply repeated multiplication with the same matrix, we can simplify it to

$$\boldsymbol{h}_t = (\boldsymbol{W}^t)^\top\boldsymbol{h}_0$$

Through eigendecomposition this can be written as

$$\boldsymbol{h}_t = \boldsymbol{Q}^\top\boldsymbol{\Lambda}^t\boldsymbol{Q}\boldsymbol{h}_0$$

Now the eigenvalues are raised to the power of t, which will cause eigenvalues less than one in magnitude to tend toward zero, and eigenvalues greater than one in magnitude to tend towards infinity. In other words the gradient either explodes or vanishes when it is propagated over many stages. Gated RNNs seek to fix this problem by allowing the gradient to flow through other paths, where it won't neither vanish nor explode (Goodfellow et al., 2016). Because the gradient is more stable in this way, gated RNNs are models with "longer memory," and the gated RNNs can also learn themselves what is worth keeping in that memory, and what can be forgotten. We will now take a closer look at a gated RNN called the LSTM.

LSTM The LSTM archictecture (Hochreiter and Schmidhuber, 1997; Gers et al., 2000) seeks to improve the vanilla RNN by introducing what can be thought of as a more long term memory. Instead of only passing the hidden state from one step to another, another vector called the *cell state* is also passed to the next time step. The cell state avoids the repeated multiplication with the same weight matrix, so it also avoids the vanishing gradient problem that comes along with that process. This is the long term memory of the LSTM, and the LSTM itself controls what to put into the cell state, and what to remove from it. It does this through so-called "control gates". The control gates are simple feedforward neural networks with a sigmoid activation function, and based on their output, the cell state is changed. The sigmoid activation function is chosen because it outputs values between 0 and 1, and so a pointwise multiplication with its output is a good way to control how much of each component should be let through.

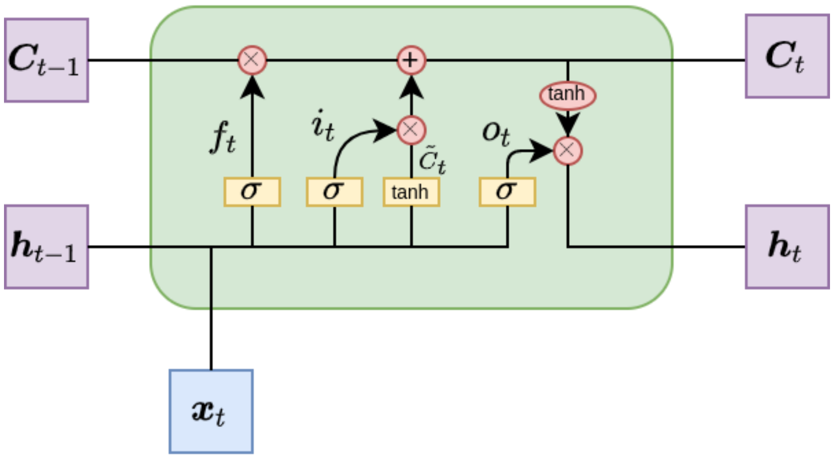


Figure 2.7: An LSTM cell. The yellow boxes represent a feedforward neural network with either a sigmoid activation function (σ), or a tanh activation function. When the lines from $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ merge at the bottom left, it represents a concatenation of them. The red circles represent pointwise operations.[4]

.

An LSTM cell diagram can be seen in Figure 2.7. The cell state travels through the horizontal line at the top, and it is only modified through pointwise multiplication and addition, as controlled by the gates. From left to right, the gates (yellow boxes with σ in them) are the *forget gate*, the *input gate*, and the *output gate*. The forget gate is responsible for which parts of the cell state can be removed. The input gate decides what new information to store in the cell state. It does this in conjunction with the tanh feedforward neural network, which outputs new candidate values of the cell state, $\tilde{C}$. The output from the input gate is what decides which values in the candidate state are actually added to the cell state. This final cell state is then sent through another tanh function,

[4]The diagram is inspired by diagrams found on colah.github.com

the output of which is multiplied with the output of the output gate, to decide which parts are output as the next hidden state. The hidden state and the cell state then flow on to be used in the next time step, or the hidden state can be used for classification by for example sending it through a feedforward neural network. It is also possible to stack LSTM layers on top of each other, where the hidden state from each time step is fed in as the input (replacing $\boldsymbol{x}_t$) to the next layer.

The LSTM and other gated RNNs have been very successful and were the state of the art machine learning models for sequence modeling for several years (Vaswani et al., 2017). In one area of research these models have been particularly successful, namely in its application to Neural Machine Translation (NMT). We will now look at how LSTMs can be used for this purpose, by using a so-called sequence to sequence, or seq2seq, model. These models were very successful; only 2 years after the paper which introduced seq2seq models (Sutskever et al., 2014) was published, Google Translate[5] started switching from Statistical Machine Translation (SMT) to NMT (Turovsky, 2016). Many improvements have been made upon seq2seq models since they first appeared, and one improvement in particular, called *attention*, has been very successful. We will soon look at the attention mechanism, first in the context of a seq2seq model, before we move on to the Transformer, an NMT model which drops the RNN structure in favor of a model based entirely around the attention mechanism.

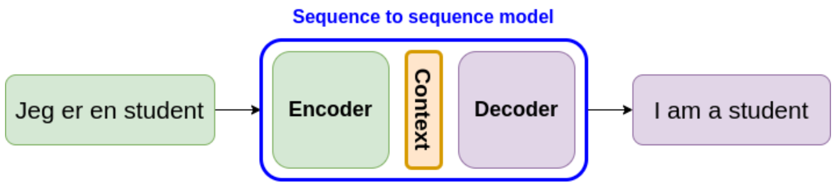


Figure 2.8: Illustration of a sequence to sequence model for translation. The input sentence is processed one word(-vector) at a time by the encoder. After having processed the whole entire input sentence, the encoder produces a context vector which is sent to the decoder for it to use in order to produce the output sentence. The decoder produces the output sentence word by word.

Seq2seq models and attention To motivate the need for the attention mechanism, we will first look at a seq2seq model without attention, mention its limitations, and then explain how attention works to make the model better. Seq2seq models work through two main components. The *encoder*, which is an RNN that takes in the input sequence, does some manipulation of the input, and feeds its output to the *decoder*, which is another RNN that uses the output from the encoder to produce the output sequence. The encoder's job is to create a good encoding, i.e. a vector representing the input sequence well enough for the decoder to be able to produce the correct output. The decoder produces an output sequence, conditioned on the context vector from the encoder. A classic use case for seq2seq models is translation. In that case the decoder takes in one

[5]translate.google.com

word at a time, and uses it to produce a context vector which the decoder uses to produce the translation. A simple illustration of this is shown in Figure 2.8.

The encoder could be an LSTM, or some other form of RNN, which takes in one word at a time (in the form of word vectors / word embeddings) along with the previous hidden state, to produce a new hidden state. The hidden state produced by the last word in the input sentence is used as the context vector, which the decoder uses as the representation of the entire input sentence. The decoder, which could also be an LSTM, uses the context vector as its initial hidden state, along with the embedding of the <START> token. It then produces a hidden state for each time step, which can be used to produce a probability distribution for what the output word should be at that time step. The hidden state is also sent to the next time step, along with the word embedding of the word which the decoder predicted at the current time step, in order to produce a new hidden state to be used to get a probability distribution for the next output word. To get predictions of words from the decoder, one could simply always choose the word with the highest probability from the probability distribution, but other methods of choosing the output words, such as beam search (Freitag and Al-Onaizan, 2017) also exist. The decoder stops once the <END> token is produced, signifying that it has reached the end of the output sentence.

That was a description of how a seq2seq translation model works at *test time*. The training procedure works differently, and I won't describe it here. See (Sutskever et al., 2014) for details on this. The vanilla seq2seq models work decently well, but a problem with this approach is that it is very hard for the encoder to compress all the information of the input sentence into the context vector. So a lot of information is lost at that step, and Bahdanau et al. (2016) propose that the context vector being passed from the encoder to the decoder works as a bottleneck for these models. As a solution they propose the attention mechanism. Attention lets the model choose which parts of the input sequence it should focus on, hence the name "attention." When a seq2seq model uses attention, the decoder has access not only to the last hidden state from the encoder, but to *all* the hidden states from the decoder.

In order for the decoder to choose which parts of the input sequence to pay attention to at any particular time step, each hidden state from the encoder is given a score. The scoring is done by simply taking the dot product between the hidden state at the current time step of the decoder, and each hidden state from the encoder. These scores are then sent through a softmax function [6], giving us what's called the *attention distribution*. Each hidden state is then multiplied by the its softmaxed score, before all hidden states are added together into what becomes a weighted average of the input sequence. This weighted sum of the input sequence is then concatenated with the hidden state from the decoder at the current timestep, and this concatenated vector is what's used to produce the probability distribution for the output word. This concatenated vector can also be sent to the next time step, instead of only the hidden state, as was the

[6]The softmax function takes in a list of real numbers and normalizes them, so the output sums to one and so can be interpreted as a probability distribution.

case in the vanilla seq2seq model.

Attention solves the bottleneck problem, and also helps with the vanishing gradient problem, and so it improves NMT performance. It also provides some interpretability, because you can look at the attention distribution to see what the decoder is focusing on when producing different parts of the output. Attention turned out to be so powerful that the current state of the art language models are based entirely on the attention mechanism. These models are based on the architecture of the original Transformer model, invented by Vaswani et al. (2017), who taught us that attention is all we need. Before getting to the transformer, though, we will look at how LSTMs can be used to create better word embeddings than word2vec.

2.3 Natural Language Processing

Natural Language Processing (NLP) refers to using statistical methods to understand text in order to solve real-world tasks (Rao and McMahan, 2019). This is done by transforming texts to usable computational representations. This section will describe different methods for creating such representations of words, starting with word embeddings

2.3.1 Word embeddings

When writing about language modeling and natural language processing in general, it is important to explain how words can be represented mathematically, so that machine learning models can get a useful representation of the sentences we want it to analyse. We need to represent words in a way that a computer can understand, which means we have to represent the words using numbers. The simplest, most straight forward way to do this is to simply build up a vocabulary, and assign a unique number to each word in that vocabulary. For example the word "the" might be represented by the number 1, and the word "be" might be represented by the number 2, etc. This is known as one-hot encoding, because you can represent the numbers associated with each word as a one-hot vector the size of your vocabulary, with a 1 at the index which represents the word. This means that if we had a vocabulary of size 4, the word "the" would be represented as [1 0 0 0] and the word "be" would be represented as [0 1 0 0], following the above example.

Obviously we need a larger vocabulary than of size 4, so this method of representing words would lead to vectors with thousands of dimensions, with all but one of the values being zero. Such sparse, high-dimensional representations, are not well suited for analysis by neural networks (Goldberg and Hirst, 2017). Another problem with using one-hot encoding to represent words, is that these vectors don't capture any meaning of the words they represent. The words "apple" and "orange" obviously have a lot in common, but one would not know that just from looking at the vectors representing the two words, and so someone who understands nothing but mathematics (a computer) can't know that the words are similar. What we would like, then, is a dense, lower dimensional representation of the words, which captures some of the meaning behind the words. How do we create such vectors? A quote from linguist John Firth can give us a hint.

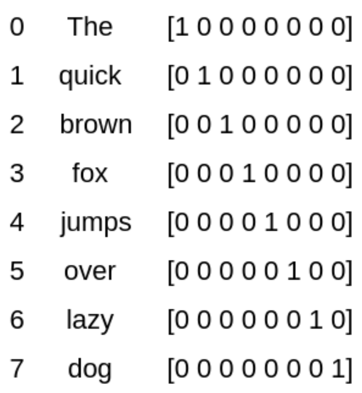
> "You shall know a word by the company it keeps"
>
> *Firth (1957)*

The above quote tells us that if we want to know the meaning of a word, we need to look at the words around it. We want our word representations to be useful for predicting other words which appear around it. One way to create word vectors using this idea is to use the word2vec algorithm (Mikolov et al., 2013). This algorithm starts by first assigning a random initial vector to each word in a fixed vocabulary, and the idea is to look at a similarity measure between different word vectors and use this similarity measure to predict whether one word is likely to appear next to another word. The vectors are then changed so as to get better at this task.

In word2vec this is done using a simple neural network, but without a non-linear activation function. In the beginning, when all words are simply assigned random vectors, the neural network will not be able to predict very accurately whether one word appears next to another, and the loss will be large. The goal is of course to minimize the loss by adjusting the word vectors through backpropagation and gradient descent. Then for every word in the text, define that word as the *center* word, and the words around it as the *outside* words (the context). Then use the similarity of the current word vectors for the center word and the outside words, to predict the probability of the outside words given the center word, or vice versa. When the center word is predicted based on the context, the learning model is called the Continuous Bag-of-Words model (CBOW), and when the context is predicted based on the center word, the learning model is called the Continuous Skip-Gram model. An illustration of the two learning methods can be seen in Figure 2.11.

We will take a closer look at the skip-gram method and ignore the CBOW method, but the concept is exactly the same for both. As previously mentioned, the skip-gram method uses the center word to predict the outside words. We don't really care about having a network which is good at predicting which words appear next to each other, but giving the network this task forces it to encode something meaningful into its hidden layer, and it is this hidden layer we will use as our word vectors. Actually, since the hidden layer doesn't have a non-linear activation function, we may instead want to call it a projection layer.

We will look at a simple example where we train three-dimensional word vectors based on a very small corpus, namely the sentence "The quick brown fox jumps over the lazy dog." First we need to create our one-hot vectors, so we assign 0 to "the," 1 to "quick," etc. and end up with the vocabulary seen in Figure 2.9, which has eight unique words. Now we need to create our training instances, meaning we need to create input/output examples for our model to train on. First we choose a "window size," which is a parameter that lets us choose how wide a context we want the model to look at. For this example, let's use a window size of 2, meaning we create examples using the two words behind of the center word, and the two words that follow. For our example, we would get



0	The	[1 0 0 0 0 0 0 0]
1	quick	[0 1 0 0 0 0 0 0]
2	brown	[0 0 1 0 0 0 0 0]
3	fox	[0 0 0 1 0 0 0 0]
4	jumps	[0 0 0 0 1 0 0 0]
5	over	[0 0 0 0 0 1 0 0]
6	lazy	[0 0 0 0 0 0 1 0]
7	dog	[0 0 0 0 0 0 0 1]

Figure 2.9: Vocabulary and associated one-hot vectors for this simple example.

the training examples

(The, quick)
(The, brown)
(quick, The)
(quick, brown)
(quick, fox)
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)
etc.

These examples are written on the form (center word, outside word), and when using the skip-gram method we want the model to output a high probability for the outside word, given the center word. We are actually going to have two vectors for each word while we train the network; one vector will be used to represent the word as an outside word, and the other will be used to represent the word as the center word. We will now look at how we set up a neural network to learn from the first training instance, (The, quick).

Let's say we want three-dimensional word embeddings. We then create a network with a three-dimensional hidden layer (or projection layer). The input and output layers have dimensions equal to the size of our vocabulary, in this case 8. An illustration of this network can be found in Figure 2.10. For our training example, (The, quick), we input the one-hot vector associated with "The." When the input vector is sent forward to the hidden layer, its one-hot nature causes it to act as a selector in the first weight matrix, effectively picking out a word vector from it. This word vector is multiplied by the weight matrix containing what we can think of as the second set of word vector for each word, leaving us with an (8×1) vector containing scores for each word in our vocabulary. The scores are then softmaxed and sent into the cost function along with the one-hot

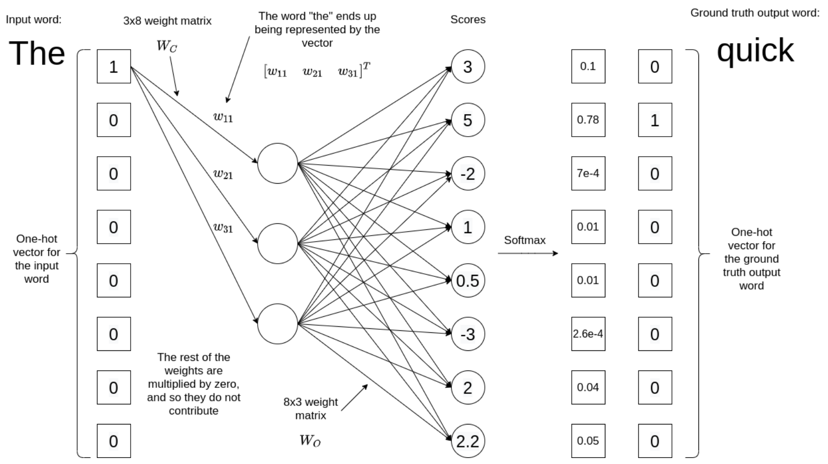

Input word:
The
3x8 weight matrix
W_C
The word "the" ends up being represented by the vector
$[w_{11} \quad w_{21} \quad w_{31}]^T$
Scores
Ground truth output word:
quick
1
0
0
0
0
0
0
0
w_{11}
w_{21}
w_{31}
One-hot vector for the input word
The rest of the weights are multiplied by zero, and so they do not contribute
8x3 weight matrix
W_O
3
5
-2
1
0.5
-3
2
2.2
Softmax
0.1
0.78
7e-4
0.01
0.01
2.6e-4
0.04
0.05
0
1
0
0
0
0
0
0
One-hot vector for the ground truth output word

Figure 2.10: Skip gram. The one-hot encoding functions as a selector; it picks out the correct word embedding from the weight matrix.

vector for the ground-truth output word, in this case the vector associated with "quick." This cost function returns a number which is backpropagated through the network to find the gradients, and then the network's weights (our word vectors) are updated using gradient descent. Once the network is done training, i.e. when the loss has converged, we simply throw away everything but the first weight layer, which contains our word vectors.

These vectors capture some of the meaning in the words, and we can see this in two different ways. First, similar words will tend to lie close together in the embedding space. So for example the words apple and orange will lie pretty close together, probably along with other fruits. This is because these words appear in similar contexts, which means that the network needs to predict similar context words for them in order to reduce the loss, and the way for the network to do that is to make their vector representations similar. The other way to see that the word embeddings capture something meaningful is that you can do some simple math with the vectors, like subtracting or adding one vector from another, and see that you end up with a meaningful result. For example if you take the word embedding for "king" and subtract the embedding for "man," and then add the embedding for "woman," the resulting vector will lie very close to the embedding for "queen."

An alternative to word2vec is GloVe (Pennington et al., 2014) which is trained using a global word-word co-occurrence matrix, containing estimated probabilities of one word occurring in the context of another word. GloVe trains with a loss function which incentivizes the model to give each word a word vector such that the dot product between word vectors equals the logarithm of probability of those two words co-occurring. Whereas word2vec uses only local information when iterating through the training corpus with a certain window size, GloVe leverages both local and global information through the co-occurrence matrix.

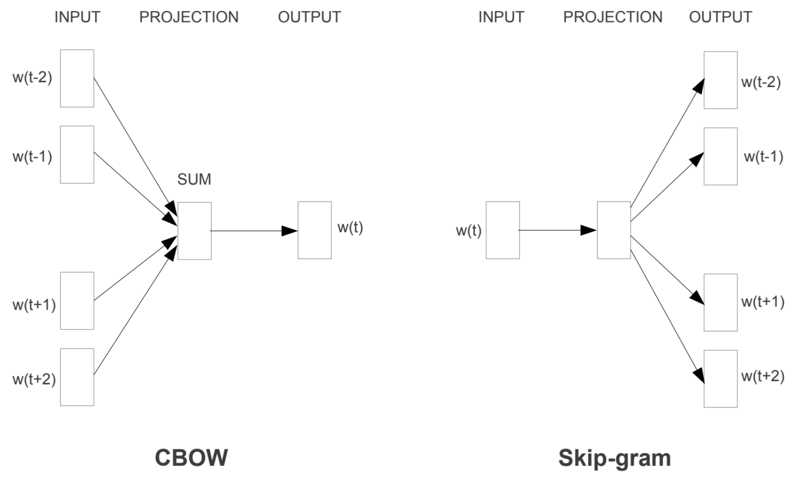


Figure 2.11: An illustration of the two methods used for creating word vectors with word2vec. The Continuous Bag-of-Words (CBOW) model predicts the center word based on the context, and the Continuous Skip-Gram model predicts the context based on the center word. Image taken from (Mikolov et al., 2013).

2.3.2 ELMo

Before moving on to the transformer architecture, we will briefly describe how LSTMs were used to create better word embeddings than those from word2vec or GloVe. With the aforementioned methods there was a fixed word embedding for each word, but words often have different meanings based on the context in which they are used. Take the word "stick" as an example. You can pick up a stick from the ground, and you can stick a magnet on your refrigerator, and you can stick to your word. ELMo (Peters et al., 2018) allows us to create context-dependent word embeddings, where each version of the word "stick" gets its own embedding.

ELMo stands for **E**mbeddings from **L**anguage **Mo**dels, because they are created using the hidden states in a deep LSTM trained with a language modeling objective. Language modeling is the task of predicting the next word in a sentence, given the previous words. ELMo does bi-directional language modeling, meaning it combines both a forward language model and a backward language model, where a backward language model predicts the previous word given the future words. This is done so that the embeddings can be made with context from both sides, not just from the previous words.

The LSTM model used to create the ELMo embeddings has two bi-LSTM layers meaning each that at each time step the LSTM cell sends its hidden state not only forward to the next time step, but also "up" to the next layer. Since it is bi-directional, and has two layers, each token has five representations: (1) the input representation, i.e. the vector that is fed into the language model. ELMo uses character level CNNs to produce the input representation. The input being given at the character level means ELMo can produce embeddings for words never seen

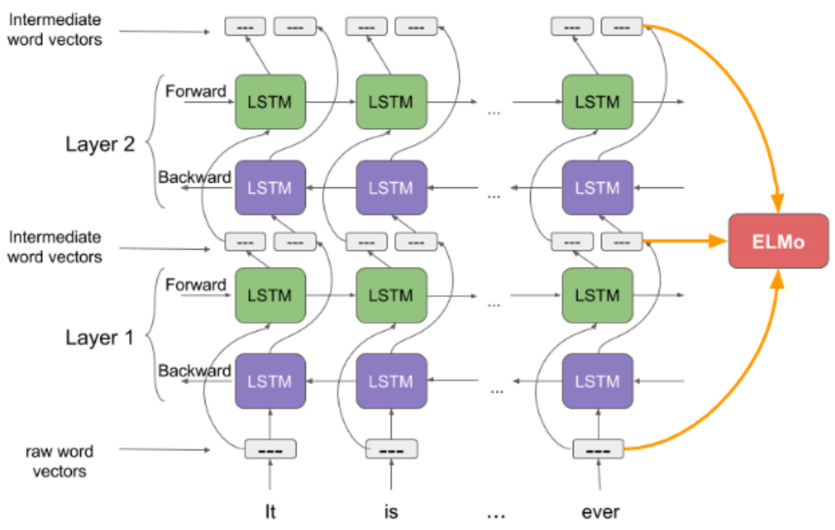


Figure 2.12: ELMo architecture diagram. Diagram from Joshi (2019).

during training; (2, 3) a representation created using the context of the previous words (one from each layer of the forward running language models); (4, 5) a representation created using the context of the future words (one from each layer of the backward running language models). ELMo representations are a weighted sum of all representations:

$$\mathbf{ELMo}_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \boldsymbol{h}_{k,j}$$

where k is the token index, L is the number of layers, $\boldsymbol{h}_{k,j}$ is a concatenation of the forward and backward running representations of token k at layer j, and $\boldsymbol{h}_{k,0}$ is the input representation of token k (e.g. a word vector from word2vec). s_k^{task} are softmax-normalized weights the model can learn in order to weight different representations according to their importance for performing well on the loss function. γ^{task} lets the model scale the whole ELMo representation (Peters et al., 2018).

At the time of its release, ELMo embeddings achieved state-of-the-art results on six NLP tasks, including question answering, textual entailment, and sentiment analysis. It was clear that context dependent word embeddings were far superior to fixed embeddings. The idea of getting word representations by training with a language modeling objective would become important also for what would end up beating ELMo namely embeddings from transformer models. We will now describe the Transformer architecture.

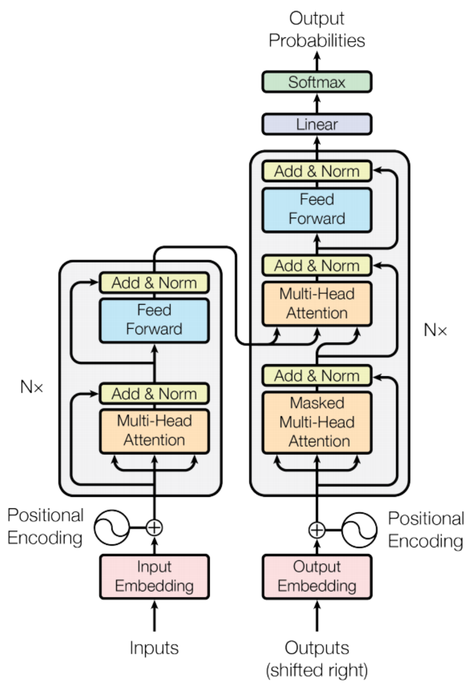


Figure 2.13: Transformer model architecture. Diagram from Vaswani et al. (2017).

2.3.3 The Transformer

Transformer models, based on the original Transformer, have taken over as the new state of the art models for language modeling in the last few years.[7] Vaswani et al. (2017) present the Transformer as an NMT model, and just like the seq2seq model, it consists of an encoder and a decoder. The encoder takes in the text in the language you want to translate from, processes it, and the decoder uses output from the encoder to produce the translated sentence. The encoder and decoder both consist of several layers (6 layers in the original transformer), and it turns out that these layers, when stacked on top of each other, produce very powerful models for other NLP tasks than just translation. Stacking encoder layers on top of each other is the basis for the BERT model, which is what we will be using for our experiments in this work. Other transformer based models such as the GPT models, are made up of decoder layers stacked on top of each other.

Transformer models don't work in a sequential manner; they instead process text one sentence at a time (batches of sentences at a time when training with SGD), as opposed to processing the sentences one word at a time. This ability to further parallelize the processing is one of the key advantages of Transformer models, because it means it can learn from a large dataset in a much shorter time than RNN models. With the ability to train on large datasets comes the ability to build huge models with millions and even billions of parameters. In

[7]See the leaderboard at https://paperswithcode.com/task/language-modelling

fact Vaswani suggests in his guest lecture at Stanford[8] that the transformer may not have better expressivity than LSTMs, and that instead maybe all the transformer is is an architecture which learns very effectively through stochastic gradient descent. We will now take a closer look at the components of the transformer, starting with how the transformer utilizes attention.

Scaled Dot-Product Attention The Transformer uses what Vaswani et al. (2017) calls Scaled Dot-Product Attention. The input is a set of query, key, and value vectors, and the output is a weighted sum of the value vectors, where the weights in the weighted sum are constructed from the key and query vectors. What to use as query, key, and value vectors we will soon get to. The creation of attention scores are produced includes taking the dot product between the query and key vectors. For this reason, the query and key vectors must be of the same dimension, d_k. The value vectors are of dimension d_v which need not equal d_k, but Vaswani et al. (2017) choose to keep them the same size. The dot product between the query and key vectors are divided by $\sqrt{d_k}$ and these scores and then softmaxed, resulting in an attention distribution which is used to create a weighted sum of the value vectors. This way of doing attention is efficient since all query, key, and value vectors can be packed into matrices Q, K, and V respectively, and the attention function on all positions can therefore be computed simultaneously as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{5}$$

Division by $\sqrt{d_k}$ is done to prevent the dot products from getting too large. Having too large values inside a softmax results in very small gradients, which hinders learning.

Multi-Head Attention One could use scaled dot-product attention by using the word embeddings as both key, value, and query vectors. However, in order to give the model extra representation subspaces, the key, value, and query vectors that are input into scaled dot-product attention are created by linearly projecting the d_{model}-dimensional embeddings h times through learned projection matrices, down to a lower dimension. In multi-head attention each triad of key, value, and query projection matrices represents a head:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

where $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. After each head is created, the next step is to concatenate them all and linearly project the resulting vectors back to the original input dimension:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where $W^O \in \mathbb{R}^{hd \times d_{\text{model}}}$.

In the decoder, there are two types of multi-head attention. In “encoder-decoder

[8]The guest lecture can be found at https://youtu.be/5vcj8kSwBCY

attention" layers, the queries at each layer come from the decoder's previous layer, while the key and value vectors are linear projections of the encoder's output. The other type of multi-head attention is multi-head self-attention, where the key, query, and value vectors are all created from the output of the previous layer. Both the encoder and decoder use multi-head self-attention. For each position (word-embedding / word representation) which is encoded with multi-head self-attention, the encoder is allowed to attend to all positions from the previous layer. For each position in the decoder, however, it can only attend to positions up to and including that position (Vaswani et al., 2017).

Position-Wise Feed-Forward Network That covers how attention functions in the transformer. The next component is the position-wise feed-forward network. This is simply a feedforward neural network which is applied separately and identically to each position. To be able to take in the word embeddings its input dimension must be d_{model}, and its output dimension must also be the same so the next layer (which is identical) can take the output as input (Vaswani et al., 2017).

Positional Encodings Because all positions are handled at the same time, the transformer has no innate way to recognize the order of the words in the input sentence, unlike RNN models. The order of the words in the input sentence is obviously important, and in order to give the transformer information about the positions of the words, a positional encoding is added to each word embedding, before it is sent through the model. The positional encoding consists of a vector of dimension d_{model} with a certain pattern that the model can learn to recognize in order to recognize the position of that particular word in the sentence. The input to the model is the sum of the positional encoding and the word embeddings.

Residuals and Layer Normalization There is a residual connection around each sub-layer, in both the encoder and the decoder. A residual connection means that the input to that layer is added to the output. Layer normalization (Ba et al., 2016) is then performed on that sum, so the output of each sub-layer, which is sent forward to the next layer is $\text{LayerNorm}(x+\text{Sublayer}(x))$. Vaswani explains in his aforementioned guest lecture at Stanford that the residuals are very important for carrying the positional information from the positional encoding forward in the network, but that they also carry more than just positional information.

Layer normalization is a normalization method where each training instance is shifted and rescaled using the mean and standard deviation of its features (Ba et al., 2016). If for example the inputs to layer normalization is this small batch of two 3-dimensional vectors

$$\boldsymbol{x}_1 = [1, 4, 5], \quad \text{and} \quad \boldsymbol{x}_2 = [1, 2, 3]$$

$\boldsymbol{x}_1$ has as its features the numbers 1, 4, and 5. The mean is then $\mu_1 \approx 3.3$ and the standard deviation is $\sigma_1 \approx 1.7$. For x_2 we have $\mu_2 = 2$, and $\sigma_2 \approx 0.82$. Each datapoint is shifted and rescaled using its own individual normalization terms:

$$\hat{\boldsymbol{x}}_1 = \left[\frac{1-3.3}{1.7}, \frac{4-3.3}{1.7}, \frac{5-3.3}{1.7}\right] \approx [-1.4, 0.4, 1]$$
$$\hat{\boldsymbol{x}}_2 = \left[\frac{1-2}{0.82}, \frac{2-2}{0.82}, \frac{3-2}{0.82}\right] \approx [-1.2, 0, 1.2]$$

Simply normalizing each datapoint can change what it represents. In case normalizing is destructive for learning, it would be good for the model to have the ability to undo the normalization. For this reason, two more parameters are introduced: the gain parameter $\boldsymbol{g}$ and the bias parameter $\boldsymbol{b}$, both of the same dimension as the input. However, Xu et al. (2019) claim that the bias and gain parameters increase the risk of the model overfitting. Given the input vector $\boldsymbol{x}$, of dimension H, the output of layer normalization is

$$\boldsymbol{y} = f\left[\frac{\boldsymbol{g}}{\sigma} \odot (\boldsymbol{x} - \mu^t) + \boldsymbol{b}\right], \quad \mu = \frac{1}{H}\sum_{i=1}^{H} x_i, \quad \sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2}$$

$\odot$ represents element-wise multiplication.

Normalization leads to faster convergence, and therefore reduces training time. It does this by reducing internal covariate shift (Ba et al., 2016). Internal covariate shift was defined by Ioffe and Szegedy (2015) as the change in the distribution of network activations due to the change in network parameters during training.

2.3.4 BERT

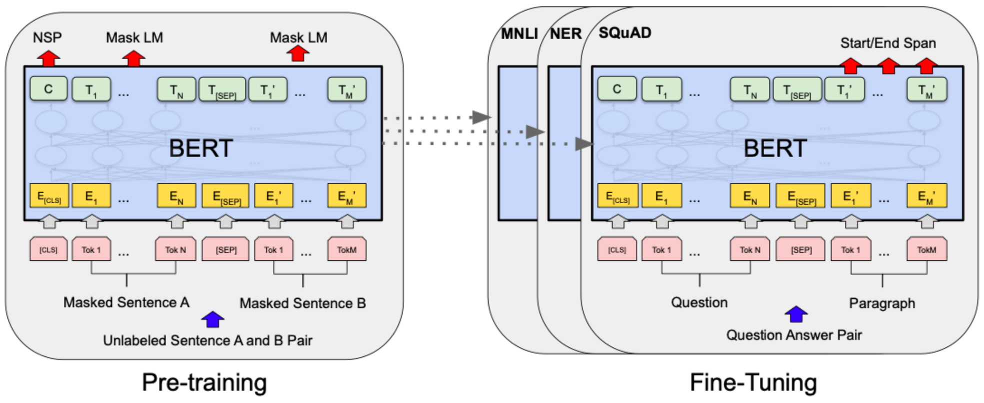


Figure 2.14: Overall pre-training and fine-tuning procedures for BERT. Diagram from Devlin et al. (2019).

BERT, which stands for Bidirectional Encoder Representations from Transformers, is a language representation model which at the time of its release achieved state-of-the-art results on eleven NLP tasks (Devlin et al., 2019). Just like ELMo, BERT encodes the context of a word into the word's representation, but unlike ELMo, it does not use bi-directional LSTMs to produce these representations. Instead it uses encoder layers from the Transformer, which can be trained much more effectively than LSTMs. ELMo has 93.6 million parameters[9] and was pre-trained on one billion words. BERT in contrast, has 110 million parameters in the base model, and 340 million parameters in the large model, and was pre-trained on a total of 3.3 billion words (Devlin et al., 2019). The architecture and training procedure(s) of BERT will now be described.

Architecture BERT is a Transformer encoder, with only a few changes. Let L denote the number of encoder layers, and H the hidden size. The original Transformer encoder had $L = 6$ layers, and a hidden size of $H = 512$. The smaller BERT model, $\text{BERT}_{\text{BASE}}$ uses $L = 12$ and $H = 768$, while the large model uses $L = 24$, and $H = 1024$. The encoder layers produce representations which are sent through a feedforward layer for the final classification task. Which embeddings are sent to the feedforward layer depends on the task. Training BERT happens in two stages. First the pre-training, where the task is Masked Language Modeling (MLM), and Next Sentence Prediction (NSP). The next stage is the fine-tuning, where the output layer used during pre-training is thrown away and replaced with one to be used for the fine-tuning task, also called the *down-stream task*.

Input/Output Representations BERT does not create word embeddings exactly, but instead word-piece (Wu et al., 2016) embeddings. This means that if one inputs the sentence

Tokenize this sentence

what is actually input to the model are the tokens

To ##ken ##ize this sentence

The prefix ## indicates that a token is a continuation of the previous token. The word-pieces are created using the following algorithm (Schuster and Nakajima, 2012):

1. Initialize the token vocabulary with the basic unicode characters and including all ASCII.

2. Build a language model using the current vocabulary.

3. Generate a new token by combining two token from the current vocabulary to increase the size of the vocabulary by one. Choose the token out of all the possible ones that increases the likelihood on the training data the most when added to the model.

4. Repeat 2 and 3 until either a predefined vocabulary limit is reached, or the likelihood falls under a certain threshold.

Since the vocabulary starts with basic characters, BERT will never the problem

[9]https://allennlp.org/elmo

that a word is out of its vocabulary. When the input sentence is divided into tokens, some special tokens are added to the sequence before BERT takes it as its input. All input sequences sent to BERT start with the [CLS] token, whose embedding is used as a representation of the whole input when doing sequence classification tasks. When inputting a pair of sentences, the sentences are separated by the [SEP] token. A [MASK] token is used to mask out tokens when doing Masked Language Modeling (MLM). All input sequences to BERT must of the same length, and so [PAD] tokens are used to pad input sequences to this length.

Masked Language Modeling ELMo concatenated the representations created during forward (left-to-right) and backward (right-to-left) language modeling, in order to use context from both sides when making word embeddings. BERT is a stack of Transformer encoder, which already uses context from both sides, but this bi-directional conditioning comes at the cost of words being able to "see themselves." For this reason, Devlin et al. (2019) use a masked language modeling pre-training objective, where some percentage of the input tokens are masked. The model is tasked with predicting 15% of the input tokens, based on the context. The chosen tokens are replaced with the [MASK] token 80% of the time, replaced with a random word 10% of the time, and left unchanged 10% the time. The reason for not using the mask token 100% of the time, is to combat the mismatch between pre-training and fine-tuning, as the [MASK] token is never seen during fine-tuning (Devlin et al., 2019).

Next Sentence Prediction Training BERT to understand the relationship between sentences is done through the pre-training objective of next sentence prediction. When creating a pre-training example, two masked sentences are combined into one input sequence, seperated by the [SEP] token. Half the time the second sentence is the sentence that followed the first sentence in the corpus from the which the sentences were gathered (with label IsNext). The other half it is a randomly sampled sentence from the corpus (with label NotNext). The final embedding of the [CLS] token is used as a representation of the entire input sequence, and it is this embedding that is sent into a feedforward layer with a softmax on top for classification into either IsNext or NotNext. Devlin et al. (2019) demonstrate that pre-training for next sentence prediction helps BERT's performance on question answering, and natural language inference.

Fine-tuning After pre-training BERT with masked language modeling and next sentence prediction on a large amount of unlabeled text, the next step is to fine-tune the model for a specific downstream task. To fine-tune the model, one throws away the output layers from pre-training, and initialize a new feedforward output layer. For token-level classification tasks, the token embeddings are fed into the output layer. For sequence level classification tasks, the [CLS] token embedding is fed into the output layer. The output from the feedforward layer is softmaxed and a loss is computed, and back propagated. An overview is shown in Figure 2.14.

3 Results and Discussion

This section will describe how our dataset was created, show some statistics on the contents of the dataset, and also show comparisons between our dataset and two other datasets, namely the Hate speech dataset from (Jensen, 2020), and the English UCC from (Price et al., 2020). After that the experiments and results from using our dataset will be presented. After experimenting with using only our new dataset, we see if results can be improved by first fine-tuning on the other labeled datasets which we compared our dataset to.

3.1 Dataset creation

We created a Norwegian version of the dataset made by Price et al. (2020), where the goal is to detect subtler forms of toxicity, and therefore follow the guideline for annotation they provide in the appendix of their paper. The annotators were asked for each comment to label it as either healthy or unhealthy, and also to label the comment as having any of the following characteristics: antagonisic, condescending, dismissive, generalisation, unfair generalisation, hostile, and/or sarcastic. There was also an extra label for meaningless comments, called "error" to be used for example if the comment was just a placeholder token, or some gibberish.

3.1.1 Gathering text

The dataset is a mix of comments scraped by us from the Norwegian forum vgd.no (VG Debatt), a debate forum hosted by one of the biggest commercial newspapers in Norway, and comments from a dataset created by Jensen (2020) where text was gathered from Facebook, Twitter, and resett.no. Approximately 30% of the data is a mix of text with equal proportions from Resett, Facebook, and Twitter, while the remaining 70% is from VG Debatt. We included comments from this hate speech dataset in order to study the overlap between labels from that dataset and the new labels provided by our annotators.

The reason for scraping text from a debate forum was to gather text from discussions, where the author is more likely to be trying to convey a point. Initial research seemed to indicate that tweets and Facebook comments are less often trying to convince someone of something, and instead the focus is often to simply air one's opinion. We wanted text were people were trying to have a conversation about a topic, and where they might disagree. Being able to disagree about topics like politics and religion, in a healthy manner, can be quite challenging. We guessed that we were most likely to find examples of unhealthy conversations with such topics, and so the text from VG Debatt was gathered from these three subcommunities: (1) Norwegian politics, (2) international politics, and (3) immigration-racism-multiculturalism. VG Debatt has since then chosen to close the latter category because of too many violations of the forum's rules, indicating that the users there were not able to have conversations on these topics in a manner which the forum's administrators deem healthy.

3.1.2 Cleaning comments

Before handing out the comments for annotation we wanted to remove URLs, usernames, and exceedingly long comments. URLs do not provide useful information, and so removing them can be seen as reducing the noise in the dataset. We did not want comments of a certain class to be associated with a username. Some users sign their comments by writing their username at the end, and users often refer to each other's usernames. We did not want any model we trained to be biased in a way where it would classify comment based on a username it saw. Most comments are not very long, and the model we set out to use, BERT, can be trained faster on shorter input sentences, because it allows for a bigger batch size. However, all input sequences to BERT must of the same length, and so if only one sentence is long, all input sequences must be equally long. Therefore it is better to discard the few long comments.

The text from the hate speech dataset by Jensen (2020) had already removed usernames from their dataset. Tweets from Twitter had usernames replaced with "user" (so it would say @USER) and the comments from Facebook had names replaced with “Navn.” Comments longer than 250 characters have also been removed from the hate speech dataset. The only further preprocessing done with this data was the removal of URLs as well as replacing the username placeholder for certain cases, as will be described later.

For the text from VG Debatt, which was what we gathered ourselves, the usernames in a thread were also gathered along with the text when scraping the forum. This allowed for the creation of a program which would search all comments for usernames, and upon finding a username would prompt a question about whether or not to remove that username from the comment. There was also an option for removing that username from all subsequent comments, or only this once, and an option to ignore all occurrences of that username. This was done because some users use a common word for their username, and in such cases we don't want to remove all occurrences of that word in the dataset. This method of removing usernames does not guarantee that there are no references to users in comments, if for example a username is misspelled or a user is referred to by a nickname, it would have gone undetected. Usernames in these comments were intially replaced with “[BRUKERNAVN],” but we later decided we wanted to have a common placeholder for names in comments from all sources. The NB-BERT-Base tokenizer was used to tokenize all comments, and only comments made up of fewer than 200 tokens were labeled.

In order to have a common placeholder for names, both “@USER” from the Twitter data and [BRUKERNAVN] from the VG Debatt data was replaced with “Navn.” As both first, middle and last names from Facebook have been replaced with “Navn” the text gathered from there has many more occurrences of the placeholder. A comment from Facebook will often start with “Navn Navn Navn” for this reason. URLs have also been removed and replaced with a [URL] token.

3.1.3 Annotation

The comments were labeled using Doccano[10], an open source text annotation tool. 6 annotators were hired to annotate the comments. The annotators were divided into three groups of two. The annotation process was done in rounds, and for each round every group got their own unique set of comments. There was no strict deadline for when to be finished with each round, and the annotators were free to choose their own hours. As a result, some groups labeled more comments than others. A set of comments would consist either of text only gathered from VG Debatt, or an equal mix of Twitter, Facebook and Resett comments from Jensen (2020). The annotators were given 500 comments to label in the first round, and 1000 comments in subsequent rounds. The annotators were asked to label the comments based on whether or not they displayed any of the following characteristics: (1) unhealthy, (2) sarcastic, (3) generalisation, (4) unfair generalisation, (5) hostile, (6) insulting, antagonising, provocative, or trolling, (7) condescending, and (8) dismissive. We gave the annotators an annotation guideline translated from Price et al. (2020), which they were asked to follow. The annotation guideline is shown in Appendix A. We included the category of generalisation, even though we only used the label of *unfair* generalisation, in order to follow the work of Price et al. (2020) as closely as possible.

All groups were instructed to go through the first 20 comments together, which was done in the hopes that it would lead to a common understanding of what it means for a comment to be healthy or unhealthy, or to display any of the other characteristics, and so would lead to more agreement between the annotators. Group G1, as defined below, gave feedback that 20 comments was not enough to see many examples of comments where they would need to discuss the category, and so they asked to go through 50 comments together during the second round, which was allowed. Annotators were also asked to hold off on comments they found very difficult to categorize, and to discuss these comments with their partner, to finish off the round. Also this was done in an effort to help maximize agreement between annotators, as well as improving the overall quality of the dataset, by having more accurate labels.

Since some annotators worked faster than others, two of the groups were rearranged after the first round, to better match the speed of the annotators. Call the annotators A1 to A6. The first groupings were then

$$G1 - A4, A3$$
$$G2 - A1, A6$$
$$G3 - A2, A5$$

After seeing that annotators 6 and 5 were considerably faster than their partner,

[10] https://github.com/doccano/doccano

and since a round could not be finished until both members of a group were done with their individual annotations, we rearranged group 2 and 3 to prevent one annotator to act as a bottleneck for the group finishing a round. After rearranging, group 1 was left unchanged and group 2 and 3 were replaced with group 4 and 5:

$$\begin{gathered} \text{G4} - \text{A5}, \text{A6} \\ \text{G5} - \text{A1}, \text{A2} \end{gathered}$$

3.1.4 Final dataset

	healthy	unhealthy	generalisation	unfair generalisation	sarcastic	hostile	antagonise	condescending	dismissive
Annotator 1	0	1	0	0	0	1	0	0	0
Annotator 2	1	0	0	0	0	0	0	0	0
Final label	0	0	0	0	0	0	0	0	0

Table 3.1: Example of how a comment ends up with no labels.

Only cases where both annotators agreed that a comment is unhealthy or has a certain attribute have resulted in a usable data point. If the annotators disagreed about all labels on a comment, the comment has no true label. An example of how the final label of a comment is produced, is shown in Table 3.1. Out of 8500 comments labeled, there were 1421 such comments with no agreement on labels, and these comments have been discarded in the final dataset. If annotators disagreed about a comment being healthy or unhealthy, but agreed about some other attribute that the comment displays, the comment is not discarded, even though it is not labeled as either healthy nor unhealthy. When classifying healthy vs unhealthy comments, comments labeled neither healthy nor unhealthy have been discarded, but they are utilized when classifying comments for other characteristics, such as e.g. sarcasm or hostility, since annotators can agree that a comment is sarcastic even if they disagree about whether or not it belongs in a healthy conversation. Comments labeled as neither healthy nor unhealthy have also been removed from the test set for reporting results on healthy vs unhealthy comment classification.

The question of whether a comment is healthy or unhealthy is an inherently subjective matter. Having 2 annotators per comment makes it possible to produce more robust labels since the annotators have to agree on the label for it to stick. If both annotators agree that a comment does not have a place in a healthy discussion, it reduces the chance of the labels being heavily influenced by the annotators personal biases, although it is of course possible for annotators to share the same biases. The people hired to annotate comments were all students at UiT, and a lack of diversity in annotators background introduces some cultural and geographical bias.

The category of sarcasm can be especially challenging, as sarcasm is often detected by recognizing obvious and intentional untruthfulness in statements, which may require some knowledge about the world, which the model does not necessarily have. It should be noted, however, that one of the interesting developments with the larger transformer models is that they seem to gather some general knowledge through the text they see during the pre-training phase. For example GPT-3 is able to do simple mathematics even though it was never trained specifically for that purpose (Brown et al., 2020). Sarcasm can be hard to detect through text, even for human beings, as sarcasm is often indicated through audible cues such as tone of voice. Training a text-based machine learning model to do this task is of course also difficult in that case, since it will only have access to the text. Sarcasm is also the attribute on which the baseline model in Price et al. (2020) performs the worst, with an AUC score of 0.588.

Price et al. (2020) are able to compare their model to human judgement, since they have many annotators and are able to hold out one annotation to act as a "human model." In our case we don't have enough annotators to do that and instead look only at ROC AUC for evaluating the overall performance of the model. For an indication of human performance on this problem we can look at the value of Krippendorff's alpha, given in Table 3.5, and also at the "human AUC" scores given in (Price et al., 2020).

The dataset was divided into a training set, a validation set, and a test set. The test and validation sets were created in such a way that the proportions of categories stayed true to the original full dataset. This choice was made in order for the labels in the test and validation sets to have the same distribution as the full dataset, and to ensure that there were at least 10 examples of comments with the least common labels. This is the only manipulation made of the test set, and the test set was not analyzed or used for anything during training. The validation set is combined with the training set when doing k-fold stratified cross validation.

3.2 Dataset statistics

After removing datapoints with "zero labels," i.e. where the annotators did not agree on any labels, we are left with a total of 7078 comments. The distribution of the labels in the full dataset, before splitting it into training, validation, and test data, is shown in Figure 3.1. About 16% of the comments are unhealthy, similar to Jensen (2020) who found that $\frac{2382}{13304} = 18\%$ of comments from their dataset were offensive. We see that we have not managed to gather very many examples of the categories *sarcastic*, *generalisation*, *unfair generalisation*, *hostile*, *antagonise*, or *dismissive*. The number of condescending and unhealthy comments is substantially larger, so we can expect better performance on these categories. Notice that the number of examples in Figure 3.1 does not add up to the total number of comments, since categories overlap. A comment may for example be both unhealthy and hostile, and thus counted twice.

Table 3.2 shows the number of comments in each of the datasets created from the full dataset. We chose an 80%, 10%, 10% split between training, validation, and test sets, respectively. This leaves as many examples as possible to be used

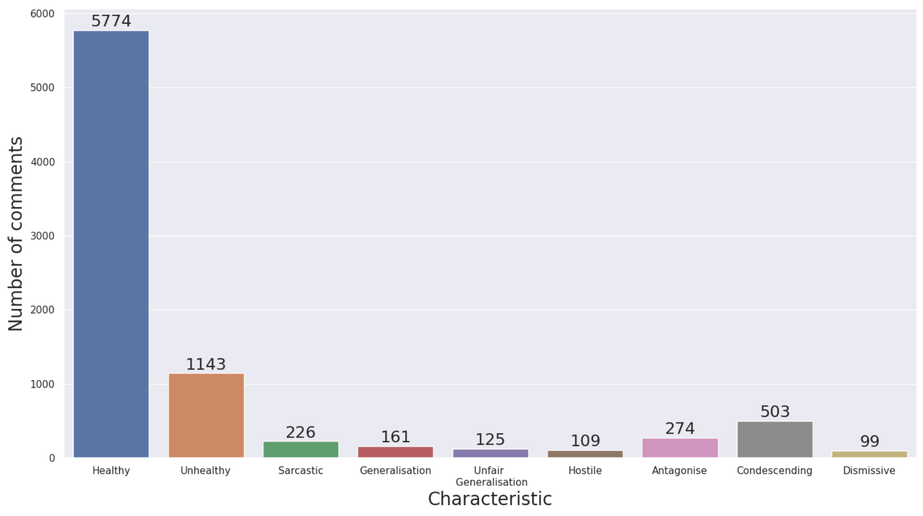


Figure 3.1: Barplot showing the number of comments from the different categories in our dataset.

for training, but in some cases only leaves 10 examples on which to report final model accuracy. This makes it hard to say whether the results on the test set are reliable, as one could imagine that 10 particularly hard examples have been chosen, or 10 particularly easy ones. For the same reason, results on the validation set can be hard to trust, so we instead combine the validation and training set in some experiments, and use stratified cross validation to evaluate the performance of the model.

Table 3.3 describes label overlap between our labels. We see that unhealthy comments are mostly condescending or antagonistic/insulting/trolling, with some hostility and unfair generalisation as well. Some are sarcastic as well, while only very few of them are dismissive. Going the other way around we see that almost all comments labeled as either unfair generalisation, hostile, antagonistic, condescending, or dismissive are also labeled as unhealthy. This means that our annotators generally found these attributes to be markers of unhealthy comments. None of the hostile or antagonising comments were labeled as healthy, and only 9%, 6% and 8% of unfair generalisation, condescending, and dismissive comments, respectively, were labeled as healthy. The reason for proportions of healthy and unhealthy don't add up to one is again disagreement between annotators, leaving a comment labeled as neither healthy nor unhealthy, but still labeled as for example hostile.

In Table 3.4 we can see the overlap between our labels and the labels from the hate speech dataset from (Jensen, 2020), for the comments from their dataset that we also included in our own. The highest overlap is naturally between healthy and neutral, as these two categories are almost equivalent. However, we see that 43% comments from the hate speech dataset were labeled as unhealthy by our annotators, even though they were labeled as neutral by the annotators

Table 3.2: The distribution of labels in our datasets. *Prop* represents the proportion of comments with a particular label, as calculated from the full dataset, i.e. before the split into train, validation, and test sets. The train, validation, and test sets were created in such a way that those proportions remain approximately representative also for those datasets.

Attribute	Train	Val	Test	Prop
healthy	4680	518	576	0.82
unhealthy	927	102	114	0.16
sarcastic	184	20	22	0.03
generalisation	131	14	16	0.02
unfair generalisation	103	10	12	0.02
hostile	89	10	10	0.02
antagonise	223	24	27	0.04
condescending	408	45	50	0.07
dismissive	79	10	10	0.01

of Jensen (2020). This indicates that we have captured something else in our dataset, as we indeed set out to do. For example many of sarcastic comments have the label neutral, which also agrees with our observation in Table 3.3, where we see that sarcasm is the only sub-attribute which does not strongly indicate that a comment with that sub-attribute is also unhealthy.

Next we wanted to compare our dataset with the Unhealthy Comments Corpus (UCC), (Price et al., 2020). We see in Figure 3.2a that the correlations between labels in our data does not resemble the UCC very closely. Overall we can see that our annotators also associated antagonising and hostile comments with being unhealthy, but not to the same degree as the annotators for UCC. We also see a similar negative correlation between a comment being condescending, and a comment being sarcastic or a generalisation. The correlation between unhealthy comments and sarcastic comments is negative also for our data, and also the negative correlation is stronger than that in UCC, once again indicating that our annotators did not associate sarcasm with being unhealthy. Price et al. (2020) had 588 crowdworkers give a total of 244468 judgements on 44355 comments, meaning they had an average of 5 annotators per comment, whereas we only had 2 annotators per comment, on about 7000 comments.

We now descrive the annotator agreement in our dataset. Table 3.5 shows an overall low value of α. As Price et al. (2020) mentions, reliability metrics such as Krippendorff's α do not work well for subjective task, since it is based on the assumption that all disagreement between annotators indicates less reliability. Disagreement between annotators can instead be a result of two different, but both acceptable, interpretations of a comment. Since the annotators are not given the context of the comments, some assumptions about the context may be made by one annotator, and not by another. Also (Ross et al., 2017) and (Jensen, 2020) see low agreement between annotators when considering annota-

Table 3.3: Proportions of overlap between our labels. Each cell represents the proportion of comments which displays the characteristic as of that column, which also displays the characteristic of that row. For example we can see that 39.4% of all sarcastic comments are healthy, while only 1.7% of all healthy comments are sarcastic.

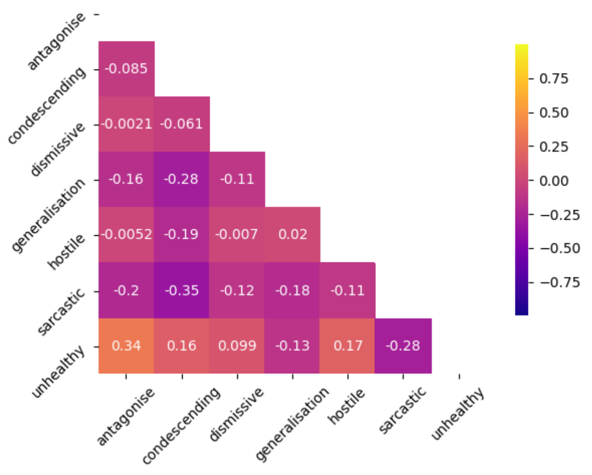
	healthy	unhealthy	sarcastic	unfair generalisation	hostile	antagonise	condescending	dismissive
healthy	1.0	0.0	0.394	0.091	0.0	0.0	0.062	0.083
unhealthy	0.0	1.0	0.364	0.855	0.958	0.949	0.812	0.917
sarcastic	0.017	0.062	1.0	0.036	0.042	0.051	0.045	0.167
unfair generalisation	0.003	0.122	0.03	1.0	0.271	0.102	0.089	0.083
hostile	0.0	0.119	0.03	0.236	1.0	0.112	0.062	0.417
antagonise	0.0	0.241	0.076	0.182	0.229	1.0	0.214	0.583
condescending	0.005	0.236	0.076	0.182	0.146	0.245	1.0	0.5
dismissive	0.001	0.028	0.03	0.018	0.104	0.071	0.054	1.0

Table 3.4: Proportions of overlap between our labels and the labels from (Jensen, 2020).

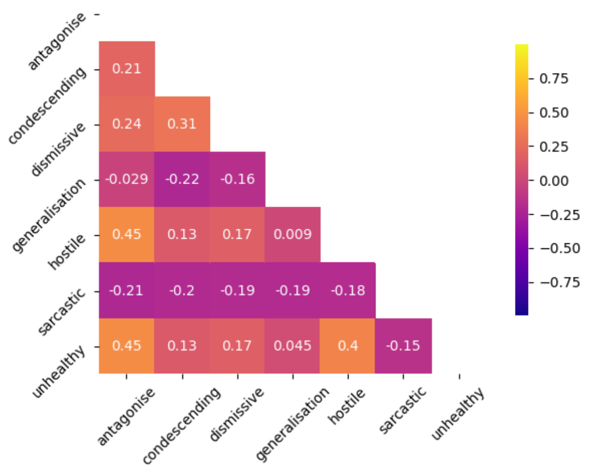
	healthy	unhealthy	sarcastic	unfair generalisation	hostile	antagonise	condescending	dismissive
hateful	0.0	0.03	0.0	0.0	0.1	0.04	0.01	0.0
moderately hateful	0.0	0.07	0.03	0.16	0.1	0.12	0.04	0.0
offensive	0.02	0.16	0.05	0.2	0.23	0.15	0.16	0.25
provocative	0.09	0.3	0.11	0.31	0.31	0.3	0.33	0.42
neutral	0.89	0.43	0.82	0.33	0.25	0.37	0.46	0.33

tion of hateful text, which is similarly subjective in nature. Even though α does not necessary reflect the quality of the dataset, it is still important to provide such a metric, in order to compare with other datasets. α also lets us see how agreement differs between the groups, and also between different categories. On top of that it highlights the difficulty of the task at hand, and indicates that even people have trouble categorising comments into these categories. The low values of α also indicates that further measures needed to be taken to make sure there was a common understanding between annotators of what it means for a comments to be unhealthy, or to have any of the sub-attributes.

Group 1 stands out as the group which the highest agreement. Agreement in that group also made a significant jump after the first round, perhaps as a results of the annotators in this group asking to label 50 comments together as opposed to 20, which they requested. It may also be a result of the post-round discussion the groups were told to conduct, in order to come to an agreement



(a) Correlation between labels in our data.



(b) Correlation between labels in UCC.

Figure 3.2

on what to label comments they found challenging. It seems based on α values that the hardest category to agree on was antagonise, which is the category for comments that were written with the intention to insult, antagonize, provoke or troll other users. This may be caused by the fact that the annotation guideline for this particular category asks about the *intention* of the comments: "Is the intention of this comment to insult, antagonize, provoke, or troll other users." This necessitates that the annotators make an assumption about the motives of the author of the comment in question. The assumptions made here naturally vary between the different annotators.

3.3 Experiments

In order to investigate how best to create a BERT model which performs well on our dataset we carry out a range of experiments. The main purpose of the experiments is to find which of the different techniques for dealing with the unbalanced dataset works best, so that technique can be used when training the final model. We show and discuss the results of stratified cross-validation for each category, as well as the final results on the test set.

3.3.1 Method

Since the datapoints have multiple labels, i.e. a comment can be both unhealthy and sarcastic at the same time, a choice has to be for how to use it for classification. Binary relevance is a very common approach to dealing with multi-label classification (Dendamrongvit and Kubat, 2009), where the problem is split into one task per class label, and so one effectively has one dataset per label (Zhang et al., 2018b). This what we have chosen to do in this work, in order to assess the difficulty of classification of each category separately, and also to test the techniques for dealing with imbalanced datasets mentioned in Section 2.2.3, like undersampling.

		unhealthy		sarcastic		unfair generalisation		hostile		antagonise		condescending		dismissive	
group	round	α	prop	α	prop	α	prop	α	prop	α	prop	α	prop	α	prop
1	1	0.541	0.228	0.315	0.036	0.498	0.067	0.424	0.091	0.566	0.046	0.258	0.05	0.466	0.024
	2	0.924	0.158	0.908	0.048	0.82	0.032	0.691	0.021	0.937	0.032	0.882	0.09	0.895	0.032
	3	0.807	0.119	0.565	0.021	0.661	0.011	0.548	0.01	0.61	0.019	0.669	0.054	0.588	0.019
	4	0.804	0.148	0.668	0.03	0.282	0.005	0.497	0.014	0.264	0.006	0.628	0.058	N/A	0.0
2	1	0.412	0.105	0.339	0.024	-0.015	0.0	0.182	0.015	0.092	0.017	0.196	0.081	0.404	0.044
3	1	0.435	0.473	0.183	0.053	0.416	0.059	0.324	0.021	0.337	0.217	0.409	0.174	0.114	0.019
4	2	0.29	0.109	0.19	0.032	-0.011	0.0	0.181	0.003	-0.079	0.029	0.482	0.089	0.135	0.011
	3	0.258	0.06	0.072	0.015	0.085	0.001	0.177	0.001	-0.127	0.015	0.475	0.047	0.121	0.007
	4	0.474	0.094	0.232	0.026	0.08	0.004	0.174	0.006	0.019	0.032	0.349	0.033	0.055	0.001
5	2	0.465	0.304	0.304	0.042	0.274	0.034	0.174	0.007	0.175	0.076	0.167	0.09	0.049	0.001
weighted average		0.567	0.165	0.407	0.032	0.323	0.018	0.351	0.015	0.286	0.041	0.486	0.072	0.362	0.014

Table 3.5: The Krippendorff's alpha values (α) as well as the proportion (prop) of comments categorised as having the different characteristics, for each group on each category. Since only 500 comments were given out in the first round, and some comments were labeled as "error" and thus discarded, the weighted average is weighted with regard to the number of comments labeled by a group on a certain round after the "error" comments have been removed.

5-fold stratified cross validation was used to find the best model. Since a new validation set is made for each fold, there is no need to hold out a separate validation set, and so the training set and validation set has been combined to create the *combined training set* during cross validation. Making the cross validation 5-fold means that 20% of the combined dataset is used for the validation set, and 80% for the training set, in each fold.

Since the dataset is imbalanced, and very heavily imbalanced in the case of most sub attributes, the techniques mentioned in Section 2.2.3 are employed and compared, namely undersampling, oversampling, weighted loss, and focal loss. For unhealthy comment detection we don't use focal loss, as that dataset is not as heavily imbalanced as the datasets for the sub-attributes. The weights in the weighted loss for characteristic c are calculated by:

$$W_c = \left[\frac{N_c}{hN}, 1 - \frac{N_c}{hN}\right]$$

where N_c is the number of examples in the minority class (positive class), N is the total number of examples, and h is a factor used to adjust the priority given to the minority class. h starts with a value of 1, which gives each class a weight equal one minus the probability that a random example belongs to that class. Higher values of h are tried until results are no longer improving compared to the previous cross-validation run with the previous value of h. Each increase is done by doubling the previous value of h, which halves the weight given to the majority class each time.

When using undersampling, random negative examples are removed until there are equally many positive and negative examples left in the training dataset. Positive examples are examples from the minority class, i.e. unhealthy comments when classifying healthy vs unhealthy comments, hostile comments when classifying hostile vs non-hostile comments, etc. When oversampling the training dataset, random positive examples are resampled until there are equally many positive and negative examples. Since the dataset is so heavily imbalanced in the case of most sub-attributes, oversampling until the proportions of labels are equals means that the model sees the same positive examples tens of times. For example the class with the lowest number of examples, *dismissive*, only has 99 examples. 10 of these as used for the test set, leaving 89 dismissive comments to be used for training. In that case each of those 89 dismissive comments will appear about 80 times over the course of one epoch of training.

We experiment with using three different models: NB-BERT-Base, NB-BERT-Large, and NorBERT. First we do 5-fold stratified cross validation to find the best model. We report the ROC AUC on all experiments. During these initial experiments the class to which the model gives the highest probability is chosen as the model's prediction.

Unless otherwise specified, training is done with the following settings:

- Unweighted cross-entropy loss
- Learning rate of 2e-5
- 3 epochs
- Linear learning rate scheduler, with 10% warmup, and linear decrease to zero after warmup.
- AdamW optimizer (10% weight decay, otherwise default values)
- Batch size of 16

3.3.2 Cross-validation and Results

The ROC curve and AUC scores from running inference on the test set on all categories is shown in Figure 3.3. The AUC scores mostly beat the scores in (Price et al., 2020), with the exception of dismissive comments. The ROC curves look good. The resolution is low for the ROC curves for the categories with the lowest number of examples, as there are only around 10 examples of the category in the test set. Since there are so many negative examples compared to positive ($N >> P$), and a ROC curve plots the true positive rate TP/P versus the false positive rate FP/N, finding a few extra true positives at the cost of many more false positives will look good on the ROC curve. With an unbalanced dataset like this the precision-recall curve and the area under it would give a better idea

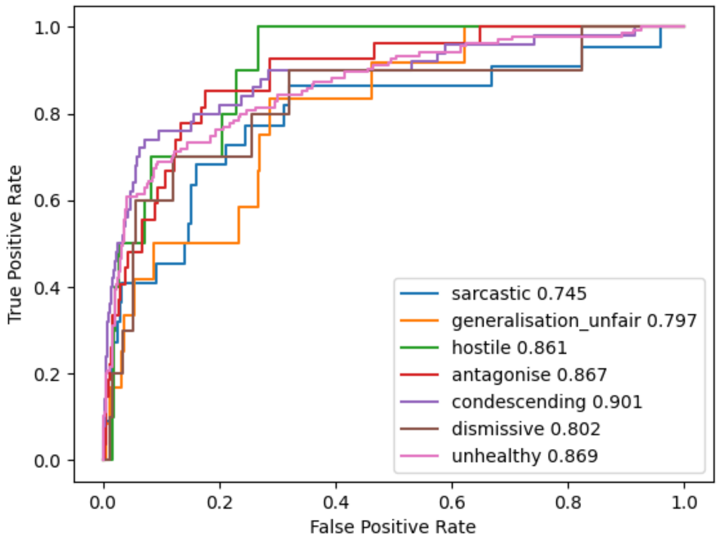


Figure 3.3: ROC curves with corresponding AUC score for all categories.

of the overall performance, but we have chosen to use ROC curves and calculate AUC. This choice was made in order to compare with both the baseline results from Price et al. (2020) who only report ROC curves and AUC, and the results Jensen (2020) got for hate speech detection, who also optimize for AUC. The experiments that leads us to these results will now be described, starting with an investigation of whether choices should be made based on cross-validation, or simply based on the validation set.

Initial experiments on NB-BERT-Base with 10-fold stratified cross-validation revealed that results were quite dependent on the choice of validation set, as shown in Figure 3.4. We see that the AUC score at the end of training ranges from below 0.82 to around 0.9, depending on the partition of the training data. Because of this we choose to use 5-fold stratified cross-validation for further experiments, using the combined training set (training set and validation set combined). Doing 5-fold stratified cross-validation for each experiment is five times more expensive than only training one model per experiment, but it also lets us trust the results more. Another thing that can be seen in Figure 3.4 is that the maximum AUC score achieved on each run is very close to the AUC score at the end of the run, which tells us that we are not overfitting when we fine tune for 3 epochs. Having decided on running experiments using 5-fold stratified cross-validation on the combined training set, we now start the experiments by choosing a pre-trained BERT model to fine-tune.

There are three Norwegian BERT models available, and the first choice that has to be made is which of these three models to use for further experiments. To decide on this all three models were fine-tuned for detection of unhealthy comments with the vanilla settings described in Section 3.3.1, and also with

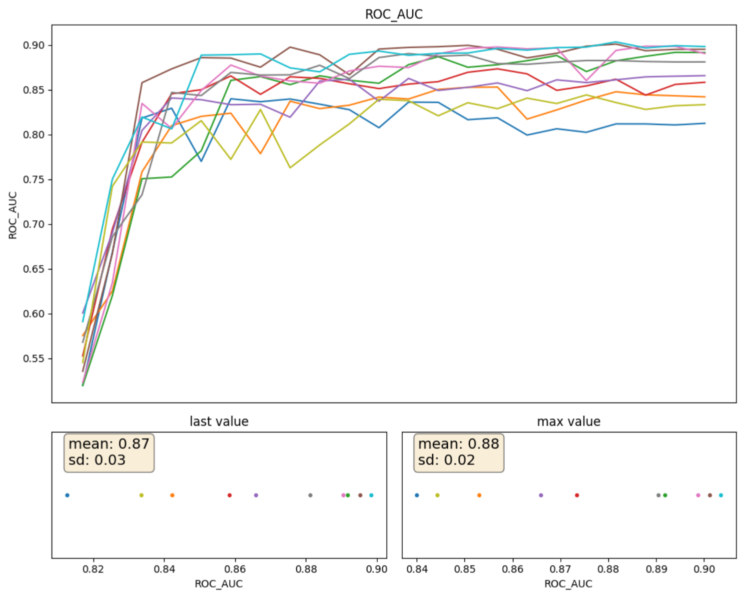


Figure 3.4: ROC AUC score over the course of fine tuning NB-BERT-Base for unhealthy comment detection with 10-fold stratified cross validation and vanilla settings. The last and max values obtained for each run is shown, along with their mean value and standard deviation.

undersampling, oversampling, and weighted loss. We base our choice of model on how they perform on the task of detecting unhealthy comments, because that is the main class for which we have the most examples. We theorize that the model which performs the best on this task will also perform the best on detection of comments containing the potential sub-attributes of unhealthy comments, such as sarcasm, hostility, etc. The section on unhealthy comment detection therefore starts with a search for which model to use.

Unhealthy comment detection. In Table 3.6 are the results of running 5-fold stratified cross-validation on the different Norwegian BERT models, using different techniques. We see that there are not great differences in NB-BERT-Base and NB-BERT-Large in being able to differentiate between healthy and unhealthy comments. Both the Large and Base NB-BERT model beat Nor-BERT in all cases. As NB-BERT-Base was the best overall model for detecting unhealthy comments, it was used for further experiments, including finding the best weights for weighted cross entropy loss, when fine tuning the model.

Having decided on NB-BERT-Base as our model, and having observed that weighted loss performs better than the other techniques, further experiments were done with 5-fold stratified cross validation to find the best weights to use in the weighted loss. In Table 3.7 we see that $[0.02, 0.98]$ are the best performing weights for weighted cross entropy loss, as the ROC AUC score is maximized for these weights. When not enough weight is put on the positive examples, the model predicts most examples as being healthy to minimize the loss, since most comments are in fact healthy, as we saw in Table 3.2. If the weight on the

Table 3.6: 5-fold stratified cross validation when using different techniques on different Norwegian BERT models after fine tuning for classification of healthy vs unhealthy comments.

Model	Technique	ROC AUC
NB-BERT-Base	Vanilla	0.867
	Undersampling	0.866
	Oversampling	0.843
	Weighted Loss	0.869
NB-BERT-Large	Vanilla	0.874
	Undersampling	0.858
	Oversampling	0.834
	Weighted Loss	0.861
NorBERT	Vanilla	0.843
	Undersampling	0.835
	Oversampling	0.829
	Weighted Loss	0.849

positive examples gets too high, however, the opposite problem occurs, where the model predicts too many healthy examples as being unhealthy, because that is less costly than wrongly predicting the heavily weighted unhealthy examples as being healthy. In the end, $[0.02, 0.98]$ are the weights chosen for training NB-BERT-Base with weighted loss, as they produce the highest AUC. The ROC curve showing the model's performance on the test set was shown in Figure 3.3. The confusion matrix is shown in Table 3.8.

The BERT classifier is able to correctly classify $92/135 = 68.1\%$ of all unhealthy comments in the test set, and 90% of the healthy comments. Considering the low degree of agreement between annotators ($\alpha = 0.567$), this result indicates that BERT is able to distinguish between healthy and unhealthy comments

Table 3.7: 5-fold stratified cross validation results for different weights in the cross entropy loss when classifying healthy vs unhealthy comments with NB-BERT-Base

Loss Weight	ROC AUC
[0.16, 0.84]	0.869
[0.08, 0.92]	0.870
[0.04, 0.96]	0.869
[0.02, 0.98]	**0.872**
[0.01, 0.99]	0.870
[0.005, 0.995]	0.782

Table 3.8: Confusion matrix for unhealthy comment detection on the test set.

AUC = 0.869	Predicted healthy	Predicted unhealthy
Actually healthy	517	59
Actually unhealthy	43	92

To see how AUC score depended on dataset size, we fine tuned NB-BERT-Base 10 times for unhealthy comment detection, first training on a random sample of 10% of the available data, then 20%, etc. up to the full dataset. For each model we calculate the AUC score on the test set. The results are shown in Figure 3.5. We see that AUC steadily increases along with dataset size, until there is a dip at 60%, which is soon recovered from, and AUC keeps increasing. This indicates that model performance would keep increasing if we had more available training data.

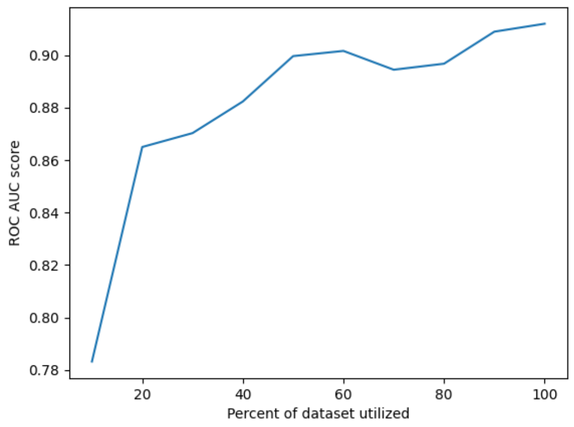


Figure 3.5: NB-BERT-Base performance on the test set after training for 3 epochs for unhealthy comment detection, on 10% of the combined training dataset, equivalent to about 600 training examples, to 100% of the dataset. The combined training dataset refers to the combination of the validation set and the training set.

The biggest jump in AUC score is seen when going from using 10% of the data to 20% of the data, where the AUC score increases by about 8%. It increases another 4% going from 20% of the data to 60%. When sampling 60% of the original dataset size, equivalent to having about 3700 training examples, the results are almost as good as when training on the full dataset. Assuming that the ratio of positive examples (i.e. unhealthy comments) stayed approximately the same at 16% when doing random sampling, that means about 600 training examples of the positive class is enough to achieve good performance for this task.

We will now move on to detection of comments containing the potential sub-attributes of unhealthy conversations, namely hostile, antagonising, dismissive, condescending, sarcastic, or unfairly generalising comments. For each sub-attribute we have done stratified cross-validation to find the best method. Here we also try using focal loss during the cross-validation, since the datasets here are even more heavily unbalanced than the unhealthy comments dataset.

Table 3.9: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either condescending or not condescending.

Method	ROC AUC
Vanilla	0.846
Oversampling	0.776
Undersampling	0.848
Focal loss, γ=2	0.847
[0.071, 0.929]	0.784
[0.036, 0.964]	0.853
[0.018, 0.982]	**0.862**
[0.009, 0.991]	0.858

Condescending. The cross validation results for detection of condescending comments are shown in Table 3.9. Condescending is the sub-attribute for which we have the most examples. The best weights achieve an AUC score of 0.862. The final model is trained on the training set, using the optimal weights $[0.018, 0.982]$. The results are shown in Table 3.10.

Table 3.10: Confusion matrix for condescending comment detection.

AUC = 0.901	Predicted 0	Predicted 1
Actually 0	628	33
Actually 1	17	33

On the test set we get a higher AUC score than we did on average with stratified cross validation.

Hostile. In Table 3.11 we see the results of cross validation for hostile comment detection. Here, focal loss gives the best overall performance. As we saw in Table 3.2, we gathered very few examples of hostile comments, and so the model may simply not be able to find patterns that distinguish between comments that are hostile and comments that are not hostile, based on so few positive examples. Focal loss resulted in the best AUC score, and so the final model is trained using that technique. The results on the test set are shown in Table 3.12. We get surprisingly good results on the test set, compared to cross validation. The AUC increases by 15%.

Antagonising/insulting/trolling. We have 247 examples of antagonising comments in our combined training set. This is the sub-attribute with the

Table 3.11: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either hostile or not hostile.

Method	ROC AUC
Vanilla	0.685
Oversampling	0.613
Undersampling	0.637
Focal loss, γ=2	**0.752**
[0.015, 0.985]	0.742
[0.0075, 0.9925]	0.723

Table 3.12: Confusion matrix for hostile comment detection.

AUC = 0.861	Predicted 0	Predicted 1
Actually 0	700	1
Actually 1	10	0

second highest number of examples, but also the attribute with the lowest value of α, indicating that it was especially hard for our annotators to agree whether a comment was antagonistic/insulting/trolling or not. In Figure 3.13 we see that weighted loss gives us the best AUC score, and so it is again used for training the final model. The results can be found in Table 3.14.Again we see a jump in AUC score compared to the cross-validation.

Sarcasm. There were almost as many sarcastic comments as there were antagonistic. The results of cross validation on sarcastic comments in Table 3.15 are slightly worse than the cross validation results for antagonistic comments, and this may be partly due to the difference in the number of examples for each category, but may also be due to the difficulty of detecting sarcastic comments in general, which have been discussed previously. Weighted loss again gives the

Table 3.13: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either antagonising or not antagonising.

Method	ROC AUC
Vanilla	0.785
Oversampling	0.724
Undersampling	0.707
Focal loss, γ=2	0.794
[0.038, 0.962]	**0.800**
[0.008, 0.992]	0.793

Table 3.14: Confusion matrix for antagonising/insulting/trolling comment detection

AUC = 0.867	Predicted 0	Predicted 1
Actually 0	676	8
Actually 1	22	5

Table 3.15: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either sarcastic or not sarcastic.

Method	ROC AUC
Vanilla	0.748
Oversampling	0.663
Undersampling	0.724
Focal loss, $\gamma = 2$	0.756
[0.032, 0.968]	0.702
[0.016, 0.984]	0.758
[0.008, 0.992]	**0.762**

best AUC score. The final model is trained using weighted loss with the weights which produce the highest AUC score, and the results on the test set are shown in Table 3.16.

Table 3.16: Confusion matrix for sarcastic comment detection.

AUC = 0.745	Predicted not sarcastic	Predicted sarcastic
Actually sarcastic	680	9
Actually not sarcastic	20	2

Unfair generalisation. Unfair generalisation is another sub-attribute for which we have very few examples. In this case stratified cross validation prefers three different techniques for training the final model, depending on which metric one values highest. For ROC AUC score, weighted loss with $[0.018, 0.982]$ performs the best, and so the final model for detecting unfair generalisation comments is trained like that. The results are shown in Table 3.18.

Dismissive. The last sub-attribute to consider is dismissive, for which we again have very few examples. Weighted loss gives the best AUC score. The AUC score on this category is good compared to the number of examples, relative to other sub-attributes, which perhaps indicates that this is a category which is easy to recognize, although it is not so common. The final model for detecting dismissive comments is trained with weighted loss. Results on the test set for the model trained with weighted loss is shown in Table 3.20.

Table 3.17: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either unfair generalisation or not unfair generalisation

Method	ROC AUC
Vanilla	0.810
Oversampling	0.698
Undersampling	0.711
Focal loss, γ=2	0.788
[0.018, 0.982]	**0.814**
[0.009, 0.991]	0.804

Table 3.18: Confusion matrix for unfair generalisation comment detection.

AUC = 0.797	Predicted 0	Predicted 1
Actually 0	696	3
Actually 1	11	1

3.3.3 Experiments on Utilizing English Data

We were informed by the main contributor to the creation of the NB-BERT-Base that the model had seen some English text during pre-training. This lead us to hypothesize that maybe cross-lingual transfer from English to Norwegian would be possible, so that the English UCC from Price et al. (2020) could be utilized. To see if NB-BERT-Base could in fact learn from English data, we fine tune NB-BERT for sentiment analysis, first using 10000 examples from the Amazon polarity dataset, made from reviews on Amazon (Zhang et al., 2015), and then tested the model on sentences from the Norwegian NoReC_fine (Øvrelid et al., 2019). We also fine-tuned that model further with different numbers of Norwegian examples to see how that affected performance. The results of these experiments are shown in Table 3.21.

We see that NB-BERT-Base is indeed able to learn from the English dataset, with an AUC score of 0.907 on the Norwegian test set without ever having seen Norwegian examples. Further fine-tuning the model with a few hundred examples improves the AUC further. When training on nothing more than 200 or 500 Norwegian examples, NB-BERT is not able to learn much, as shown the by very low AUC score, but when the model is first trained on the English dataset, we see a huge jump in AUC score. When training on the full Norwegian dataset, the advantage of training on English data first, disappears. This indicates that cross-lingual transfer is most useful in cases where few examples exist in the target language.

Having observed that NB-BERT-Base can learn from English datasets, we investigate whether it can be useful for improving our previous results. To do that we fine-tune seven NB-BERT models on the UCC, one for each category, and then evaluate and fine-tune the models further on our Norwegian dataset.

Table 3.19: Results of 5-fold stratified cross validation when using different methods for fine-tuning NB-BERT-Base for classifying comments as either dismissive or not dismissive.

Method	ROC AUC
Vanilla	0.812
Oversampling	0.682
Undersampling	0.761
Focal loss, γ=2	0.844
[0.014, 0.986]	**0.860**
[0.007, 0.993]	0.857
[0.004, 0.996]	0.854
[0.002, 0.998]	0.846

Table 3.20: Confusion matrix for dismissive comment detection.

AUC = 0.802	Predicted 0	Predicted 1
Actually 0	698	3
Actually 1	10	0

We evaluate three different models on our validation set for each category: one which has not seen any Norwegian examples; one that is trained on the Norwegian training set for 1 epoch; one that is trained on the Norwegian training set for 3 epochs. The reason for evaluating these three different models is to see if some of the models lose generalisation by overfitting to the Norwegian training set, and thus forgets what it learned from the English data. When fine-tuning we use the best technique for training each category, which were found in Section 3.3.2, both when training on the English and the Norwegian data. These results are shown in Table 3.22.

We are now evaluating on the validation set, and no longer doing stratified cross-validation, which means that the results are not directly comparable to the validation results in Section 3.3.2. Comparisons will therefore be made

Table 3.21: AUC scores showing the results of cross-lingual transfer when training NB-BERT-Base for sentiment analysis. Vanilla refers to regular fine tuning on the Norwegian dataset. Zero-shot means the model saw no Norwegian examples before being tested. Continued refers to continuing fine-tuning on the Norwegian dataset after the English.

# Norwegian examples	Vanilla	Zero-shot	Continued
3894	0.949	0.907	0.941
200	0.612	0.907	0.917
500	0.639	0.907	0.920

Table 3.22: AUC scores for classification into all categories, using the model first trained on the English UCC. Results are shown on the validation set when doing no further fine-tuning on our dataset (zero-shot), and when training with our training dataset for 1 epoch, and 3 epochs.

	Zero-shot	1 epoch	3 epochs
unhealthy	0.693	0.796	0.838
sarcastic	0.698	0.723	0.684
unfair generalisation	0.883	0.905	0.904
hostile	0.608	0.897	0.896
antagonise	0.612	0.385	0.792
condescending	0.736	0.734	0.732
dismissive	0.905	0.921	0.859

through the test set. We run inference on the test set using the best performing models from Table 3.22. The results are shown in Table 3.23.

Table 3.23: Test results after first fine-tuning NB-BERT on the UCC before fine-tuning on our dataset for each category.

category	AUC
unhealthy	0.838
sarcastic	0.815
unfair generalisation	0.866
hostile	0.855
antagonise	0.815
condescending	0.715
dismissive	0.868

The AUC decreases for unhealthy comments, condescending comments, hostile comments, unfair generalisation. The AUC increases by 6.6% for dismissive comments, 7% for sarcastic comments, 5.2% for antagonising comments. Overall there seems to be some utility in first training on the English UCC, as it works for some categories and not for others. This might be explained by the difference in correlations between labels in the two datasets which we saw in Figure 3.2b and 3.2a.

3.3.4 Supplementary Training

It has been shown that applying supplementary training to BERT on a related task can improve results when fine tuning on the target task (Phang et al., 2018). In this section we experiment with fine-tuning NB-BERT on offensive comments before fine tuning on the final tasks of detecting the categories in our dataset.

Table 3.24: AUC scores for classification into all categories, using the model first trained on detecting offensive comments. Results on the validation set are shown when doing no further fine-tuning on our dataset (zero-shot), and when training with our training dataset for 1 epoch, and 3 epochs.

	Zero-shot	1 epoch	3 epochs
unhealthy	0.826	0.909	0.918
sarcastic	0.512	0.686	0.708
unfair generalisation	0.844	0.856	0.831
hostile	0.872	0.893	0.801
antagonise	0.813	0.844	0.786
condescending	0.784	0.878	0.857
dismissive	0.774	0.889	0.816

We saw in Table 3.4 that there was some overlap between labels in our dataset and labels in the hate speech dataset created by Jensen (2020). This indicates that the tasks are related. We train NB-BERT for classifying comments from that dataset as either neutral or offensive. To do this the 4 labels *provocative*, *offensive*, *moderately hateful*, and *hateful* are combined into one single label *offensive*, and the neutral comments are left as neutral. Some comments from the hate speech dataset were used in our dataset, and those comments were removed before training this model. This model was then further fine-tuned for classifying comments into our 7 categories. Again three models are evaluated, one zero-shot model, one trained for 1 epoch, and one trained for 3 epochs on our dataset. The results on the validation set are shown in Table 3.24. The best performing models are run on the test set, and those results are shown in Table 3.25.

Table 3.25: Test results after first fine-tuning NB-BERT for detection of offensive comments before fine-tuning for each category.

category	AUC
unhealthy	0.884
sarcastic	0.810
unfair generalisation	0.861
hostile	0.888
antagonise	0.894
condescending	0.905
dismissive	0.880

The AUC score increases by 7.8% for dismissive comments, 6.3% for unfair generalisation, 6.5% for sarcastic comments, 2.7% for antagonising comments, 2.7% for hostile comments, 0.4% for condescending comments, 1.5% for unhealthy comments, when first training on offensive comments.

4 Conclusion and future work

In this thesis a new dataset for comment classification in Norwegian, into categories of healthy, unhealthy, sarcastic, generalisation, unfair generalisation, dismissive, hostile, antagonistic, and condescending, has been presented. The process of creating the dataset has been described, and we conducted an analysis of the dataset, including a measure of agreement between annotators, and the proportions of the different categories. This analysis showed that there was mostly a low level of agreement between annotators, and that very few examples were found for many of the categories. We found a good number of examples for the main category, namely *unhealthy* comments, and the performance on this category was good. We showed how having different numbers of examples for this category affects model performance, concluding that having around 600 examples of unhealthy comments would have been enough to train a good classifier.

We also demonstrated that NB-BERT can learn from English datasets, getting an AUC score of 0.907 on a dataset for Norwegian sentiment analysis, after training only on English examples for sentiment analysis. We tried to utilize this bi-lingual ability of NB-BERT by training on an English dataset of unhealthy comments from Price et al. (2020), but were unsuccessful in improving our results from this. We were however successful in improving results with a model first trained to detect offensive comments from a Norwegian hate speech dataset created by Jensen (2020).

There do not exist many Norwegian datasets of online comments, and for future work in detecting unhealthy comments, having more examples of such comments would lead to better results. Starting with this dataset it is possible to hire more annotators to label the comments again, as well as new comments. Having more annotators per comment would result in better labels overall, and more examples would improve overall performance, especially for the categories with very few examples. Having more people annotate the dataset would also allow for a "human model" to compare results with, as is done in (Price et al., 2020), and if the annotators are a diverse group it would also likely reduce shared biases between annotators. It is also possible to experiment with how fine-tuning BERT for sentiment analysis before doing unhealthy comment detection affects the results, as the Norwegian sentiment dataset NoReC is quite large.

Zhang et al. (2018a) demonstrate that it is possible to detect linguistic cues that predicts a conversation's future health. They do this by capturing pragmatic devices, such as e.g. politeness, used to start conversations in Wikipedia's *talk page* discussions, and analyze their relation to how the conversation develops. Detection of the aforementioned sub-attributes of unhealthy comments can be useful in future work on predicting the trajectory of conversations.

References

Alpaydin, E. (2014). *Introduction to machine learning.* MIT press, third edition.

Amna Veledar, R. B. (2018). Hatefulle ytringer i offentlig debatt på nett.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.

Bahdanau, D., Cho, K., and Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Buda, M., Maki, A., and Mazurowski, M. A. (2018). A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106:249–259.

Bui, H. (2020). Roc curve transforms the way we look at a classification problem.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46.

Dendamrongvit, S. and Kubat, M. (2009). Undersampling approach for imbalanced training sets and induction from multi-label text-categorization domains. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 40–52. Springer.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Firth, J. R. (1957). *A synopsis of linguistic theory 1930-55.*, volume 1952-59. The Philological Society, Oxford.

Fleiss, J. (1971). Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378—382.

Freitag, M. and Al-Onaizan, Y. (2017). Beam search strategies for neural machine translation. *Proceedings of the First Workshop on Neural Machine Translation.*

Gers, F., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with lstm. *Neural computation*, 12:2451–71.

Goldberg, Y. and Hirst, G. (2017). *Neural Network Methods in Natural Language Processing.* Morgan & Claypool Publishers.

Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press, Cambridge, MA, USA. `http://www.deeplearningbook.org`.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* Prentice Hall, Upper Saddle River, NJ. 2nd edition.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Jensen, M. H. (2020). Detecting hateful utterances using an anomaly detection approach. Master's thesis, Norwegian University of Science and Technology (NTNU).

Joshi, P. (2019). What is elmo: Elmo for text classification in python.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

Krippendorff, K. (2011). Computing krippendorff's alpha-reliability.

Krippendorff, K. (2018). *Content analysis: An introduction to its methodology*. Sage publications.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988.

Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space.

Mosbach, M., Andriushchenko, M., and Klakow, D. (2020). On the stability of fine-tuning bert: Misconceptions, explanations, and strong baselines. *arXiv preprint arXiv:2006.04884*.

Nadim, M., Thorbjørnsrud, K., and Fladmoe, A. (2021). Gråsoner og grenseoverskridelser på nettet: En studie av deltagere i opphetede og aggressive nettdebatter.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$.

Øvrelid, L., Mæhlum, P., Barnes, J., and Velldal, E. (2019). A fine-grained sentiment dataset for norwegian. *arXiv preprint arXiv:1911.12722*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

Pennington, J., Socher, R., and Manning, C. (2014). GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations.

Phang, J., Févry, T., and Bowman, S. R. (2018). Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks. *arXiv preprint arXiv:1811.01088*.

Price, I., Gifford-Moore, J., Flemming, J., Musker, S., Roichman, M., Sylvain, G., Thain, N., Dixon, L., and Sorensen, J. (2020). Six attributes of unhealthy conversation.

Quandt, T. (2018). Dark participation.

Rao, D. and McMahan, B. (2019). *Natural language processing with PyTorch: build intelligent language applications using deep learning.* " O'Reilly Media, Inc.".

Risch, J. and Krestel, R. (2020). *Toxic Comment Detection in Online Discussions*, pages 85–109.

Ross, B., Rist, M., Carbonell, G., Cabrera, B., Kurowsky, N., and Wojatzki, M. (2017). Measuring the reliability of hate speech annotations: The case of the european refugee crisis. *arXiv preprint arXiv:1701.08118.*

Ruder, S. (2017). An overview of gradient descent optimization algorithms.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536.

Saleem, H. M., Dillon, K. P., Benesch, S., and Ruths, D. (2017). A web of hate: Tackling hateful speech in online social spaces.

Schuster, M. and Nakajima, K. (2012). Japanese and korean voice search. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5149–5152.

Suler, J. (2004). The online disinhibition effect. *Cyberpsychology & behavior : the impact of the Internet, multimedia and virtual reality on behavior and society*, 7:321–6.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks.

Tan, C., Niculae, V., Danescu-Niculescu-Mizil, C., and Lee, L. (2016). Winning arguments. *Proceedings of the 25th International Conference on World Wide Web.*

Theodoridis, S. and Koutroumbas, K. (2008). *Pattern Recognition, Fourth Edition.* Academic Press, Inc., USA, 4th edition.

Turovsky, B. (2016). Found in translation: More accurate, fluent sentences in google translate. `https://blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/`. Accessed: 2020-11-20.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

Velldal, E., Øvrelid, L., Bergem, E. A., Stadsnes, C., Touileb, S., and Jørgensen, F. (2017). Norec: The norwegian review corpus.

Vogels, E. A. (2021). The state of online harassment.

Wang, A., Hamilton, W. L., and Leskovec, J. (2016). Learning linguistic descriptors of user roles in online communities. In *Proceedings of the First*

Workshop on NLP and Computational Social Science, pages 76–85, Austin, Texas. Association for Computational Linguistics.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. (2019). Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation.

Xu, J., Sun, X., Zhang, Z., Zhao, G., and Lin, J. (2019). Understanding and improving layer normalization.

Zhang, J., Chang, J., Danescu-Niculescu-Mizil, C., Dixon, L., Hua, Y., Taraborelli, D., and Thain, N. (2018a). Conversations gone awry: Detecting early signs of conversational failure. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1350–1361, Melbourne, Australia. Association for Computational Linguistics.

Zhang, M.-L., Li, Y.-K., Liu, X.-Y., and Geng, X. (2018b). Binary relevance for multi-label learning: an overview. *Frontiers of Computer Science*, 12(2):191–202.

Zhang, X., Zhao, J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. *arXiv preprint arXiv:1509.01626*.

A Annotation Guideline

I denne jobben vil du bli bedt om å lese kommentarer og å uttrykke en samlet mening om de hører til i en sunn samtale på nettet eller ikke.

Du vil også bli spurt om å identifisere om kommentarene inneholder en rekke egenskaper som kan føre til en usunn samtale. Disse egenskapene er: sarkasme, generalisering og urettferdig generalisering, fiendtlighet, aggresjon, om de er avvisende, og om de er nedlatende.

Alle kommentarer er ekte kommentarer skrevet av ekte folk i samtaler/diskusjoner på nettet. De fleste av dem er et svar på en eller flere kommentarer (som du ikke blir gitt). Spørsmålene du skal besvare er laget slik at du ikke skal trenge å vite hva de andre kommentarene er.

Kommentarene omhandler ofte politiske eller på annen måte kontroversielle temaer. Du skal gjøre ditt beste for å ikke la dine egne meninger om disse temaene påvirke din vurdering på om kommentaren hører til i en sunn samtale, og om den inneholder noen av egenskapene nevnt ovenfor.

Du skal lese hele teksten til hver kommentar før du gjør din vurdering. Noen ganger kommer egenskapene du leter etter på slutten av kommentaren.

Bruk alltid spørsmålene som står under som basis for dine annoteringer. Les godt gjennom de følgende beskrivelsene og spørsmålene, og pass på at du har en forståelse for alt innholdet, før du begynner annoteringen:

1. Sunne samtaler på nettet: Hva er kjennetegnene til en sunn samtale?

- Innlegg og kommentarer er skrevet i god tro.
- Innlegg og kommentarer er ikke overdrevent fiendtlig, og er ikke destruktive.
- Kommentarene i samtalen oppfordrer generelt til engasjement.
- Samtalen kan inneholde robust engasjement og debatt
- Samtalen er i hovedsak fokusert på substans og ideer.

En sunn samtale krever ikke nødvendigvis at hvert innlegg og kommentar er:

- vennlig
- grammatisk korrekt
- godt konstruert eller godt strukturert
- renset og uten vulgaritet
- intellektuell eller substantiv

Med dette i bakhodet kan du svare på følgende spørsmål: syns du denne kommentaren hører til i en sunn samtale på nettet? Hvis ja, marker den som sunn. Hvis nei, marker den som usunn.

2. En kommentar er sarkastisk hvis den bruker ironi for å håne eller formidle forakt, men kan også være sarkastisk ved at den mente meningen er forskjellig fra det som bokstavelig talt ble skrevet (det vil si at all ironi her anses som sarkasme). Sarkasme kan altså brukes på en stygg eller lekende måte. Ikke all humor eller stygghet er sarkasme. Er denne kommentaren sarkastisk? Hvis ja, marker den som sarkastisk.

3. Gjør denne kommentaren en generalisering om en gruppe mennesker? Hvis ja, marker den som generaliserende.

4. Hvis ja på forrige spørsmål, ville en gruppe mennesker følt at generaliseringen i kommentaren var urettferdig/feil? Hvis ja, marker den som urettferdig generalisering.

5. Er denne kommentaren unødvendig fiendtlig? Hvis ja, marker den som fiendtlig.

6. Er intensjonen med denne kommentaren å fornærme, antagonisere, provosere, eller trolle andre brukere? Hvis ja, marker den som antagoniserende / fornærmende / trolling

7. En kommentar med en nedlatende tone vil generelt anta en holdning av overlegenhet, og antyde at de(n) andre brukeren(e) er uvitende, barnaktig, naiv eller uintelligent. Slike kommentarer vil vanligvis innebære at de(n) andre brukeren(e) ikke skal tas seriøst. Er denne kommentaren nedlatende? Hvis ja, marker den som nedlatende.

8. En kommentar er avvisende hvis den avviser eller latterliggjør en annen kommentar uten god grunn, eller prøver å presse en annen bruker og deres ideer ut av samtalene. Merk: En kommentar som uttrykker *uenighet* er ikke nødvendigvis avvisende. Er denne kommentaren avvisende? Hvis ja, marker den som avvisende.

9. Hvis kommentaren ikke inneholder noe meningsfullt, f.eks. at det bare står [URL] eller at på en annen måte ikke er en setning, marker den som feil. Marker den også som feil hvis det er et brukernavn i kommentaren, eller om det ser ut som at noe som ikke burde blitt erstattet med [BRUKERNAVN] har blitt erstattet.

Ha disse spørsmålene og beskrivelsene tilgjengelig mens du annoterer, slik at du lett kan lese dem på nytt hvis du blir usikker på hva du skal markere noe som.

B Libraries

All models were made in python. The main libraries used outside of Python's standard libraries were:

- PyTorch (Paszke et al., 2019).
- Hugging Face's Transformers library (Wolf et al., 2019).
- NumPy
- Matplotlib
- Seaborn
- scikit learn
- Pandas
- BeautifulSoup