Neural Networks to Transformers - Classifying News Based on Titles

Halvor Tyseng
UiO - CS:physics - FYS5149

(Dated: June 17, 2024)

Transformers are a complex type of a Neural Network (NN) that has shown to be highly effective in scientific areas such as Natural Language Processing (NLP). In this report I present how Transformers model are architechtured, starting from the basics of NN's. In addition we will investigate Transformer based Text Embedding models, that use the transformer architechture to represent text in a vector space. In addition to describing how text embeddings can be used, a simple mini experiment is presented, where news articles are classified based on their titles. This is done by training a simple Feed Forward Neural Network (FNN) with the text embeddings of the title as input. A accuracy of 88.36% is achieved.

I. INTRODUCTION

Language is a cornerstone for humans, we use it everywhere. So cumbersome is language that it has become a field of study in itself, linguistics. Even for physicist, language is important. Although math may be what most relates to physics, when reading a physics text book or article language is also present, communicating ideas and concepts.

In the digital age computers have increasingly become a part of many sciences, and linguistics is no exception. The cross over between linguistics and computer science is named Natural Language Processing (NLP).

For computers to manipulate text it needs to be represented in a way that the computer can work with effectively. This is where text embeddings come in. Text embeddings are a way to represent text in a vector space, where the text is mapped to a vector. Vectors are perfect for computers to work with, and therefor creating good representations of text is highly beneficial for many NLP tasks.

In recent years the the revolution of deep learning has, as with many other parts of computer science, had a big impact on NLP. One of the key component of making models such as chatGPT possible is the transformer architecture, which are discussed later in this report. The same architecture has also lead the way for transformer based text embedding models, a new way of creating text representations.

A. Structure of the Report

The structure of this report deviates from that a usual computational science report. In this report I have two different objectives, the first is explaining how Transformer based text embedding models work (more?). As the title of the report suggest I will do this by starting with the well known FNN. Then building upon this with the RNN architechture which has been important in the field of NLP, highlighting a couple of limitations.

Next I will describe how the Transformer architechture solves some of these limitations of RNN models. The Transformer model is the basis for Transformer based text embedding models, which I will describe in the following section. Looking in detail of specifics of these models.

The second objective of this report is investigating how text embeddings can be used for different purposes. This will be done in the section Text Embeddings as Features.

In addition to describing how text embeddings can be used, I will also look at a specific example using a dataset. This mini experiment will try to classify news articles based on text embeddings of only their titles. These titles already have a label, and I train a simple FNN to classify the articles. In that section I will present the results as well discuss them.

Lastly i will in the discussion section discuss more generally how text embeddings can be used in different contexts, what obstacles text embeddings still face in representing text, and what the future might hold for text embeddings.

The github repository for this report can be found at https://github.com/halvorty/FYS5429

II. BACKGROUND

In this section we will go from Feedforward Neural Networks (FFNs) to the transformer model.

A. Feedforward Neural Networks

The goal of a Neural Network is to approximate a unknown function f that is dependent on some variables $\mathbf{x}$ and has some output $\mathbf{y}$. The approximation function $\hat{f}$ is in addition to be dependent on the variables $\mathbf{x}$ also dependent on some parameters θ. Varying these parameters will change the output of the function $\hat{f}$, and therefor the goal is to find the parameters that best approximate the function f.

Feedforward Neural Networks can mathematically be seen as a series of affine matrix-matrix and matrix-vector multiplications. Another name for FNN is Multi layer perceptron (MLP). This name gives some insight to the

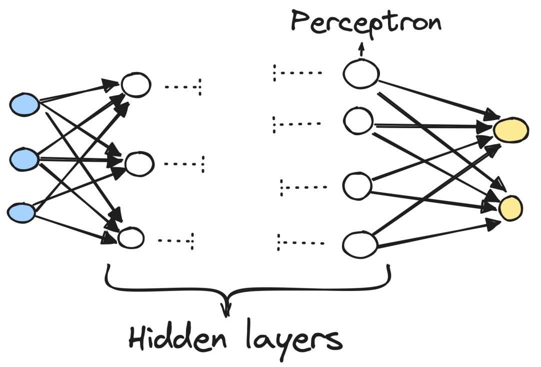

FIG. 1. Illustration of a fully-connected FNN. The blue dots (on the left) represents the input, where as the yellow (on the right) represents the output. In the middle we find the hidden layers, which can vary both in number, and number of containing perceptrons

structure of the network. A perceptron is inspired by the neurons in the human brain. The perceptron is its own function that takes some inputs, multiplies each input with a weight and sums the results. This sum together with a bias is then passed through an activation function. Mathematically this can be written as:

$$o = a(\sum_{j=1}^{n} w_j i_j + b), \tag{1}$$

where o is the output, a is the activation function, w_j is the weight of input j, i_j is the input j and b is the bias. A neural network is a series of perceptrons, where the output of one layer is the input of the next layer. Each layer often consist of a number of perceptrons, "stacked" on top of each other. An illustration of a FNN is presented in Figure 1, where the blue dots represents the input, the yellow the output and the hidden layers in the middle. Each line between the perceptrons represents a weight, and the bias is not shown in the figure. The white circles in the figure represents the activation function of the perceptron, as seen in equation (1).

The sum of all the different weights and biases in the network is the parameters of the network, referred to as θ. The activation function is what allows the network to learn non-linear functions. The activation function form can take many forms like the sigmoid function, the tanh function or the ReLU function [2].

When a FNN/MLP is set up with multiple perceptrons, and the weights are initialized, the network can given som input values $\mathbf{x}$ produce an output $\hat{\mathbf{y}}$. The output is then compared to some "true output" $\mathbf{y}$, given by data we have. With a loss function at hand we can then compute the loss of the network, and use this to update the weights of the network. The loss function is a measure of how "accurate" the networks prediction of the output is, and the specific loss function is a choice. The choice of loss function is dependent on the task, and data at hand. Some common loss functions are the mean squared error and the cross entropy loss [2]. In the loss function we can also add a regularization term. This can prevent the network from overfitting the data.

Optimization is a crucial step in training neural networks. The goal of optimization is to find the set of parameters that minimize a given loss function. In the context of neural networks, this typically involves adjusting the weights and biases of the network to minimize the difference between the predicted output and the true output. There are various optimization algorithms available, such as gradient descent, which iteratively updates the parameters in the direction of steepest descent of the loss function. Other advanced optimization algorithms, such as Adam and RMSprop, incorporate adaptive learning rates to improve convergence speed. Choosing the right optimization algorithm and tuning its hyperparameters can greatly impact the performance of the neural network.

Backpropagation is a key algorithm for training neural networks. It is used to compute the gradients of the loss function with respect to the parameters of the network. The idea behind backpropagation is to propagate the error from the output layer back to the input layer, updating the weights and biases along the way. This is done by applying the chain rule of calculus to compute the partial derivatives of the loss function with respect to each parameter. Backpropagation allows the network to learn from its mistakes and adjust its parameters accordingly. It is an efficient way to compute the gradients in neural networks with multiple layers and complex architectures.

The training of a neural network is an iterative process that involves feeding input data through the network, computing the output, calculating the loss, and updating the parameters using the optimization algorithm and backpropagation. This process is repeated for a number of epochs until the network converges to a set of parameters that minimize the loss function.

FNN usecases in NLP is limited by the fact that can not handle sequences of data. One way to use FNN in NLP is to use a fixed size input, like a bag of words. This is not ideal, as the order of the words are lost. We can expand the FNN to handle sequences of data by using Recurrent Neural Networks (RNN).

B. Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a type of neural network that is designed to handle sequences of data. This data can be stock prizes over time, audio data or crucially text data.

In opposition to FNN (and also Convolution Neural Networks) where the the outputs only flow in one direction in, from input towards output. RNNs has connections pointing backwards in the network. This allows the network to remember information from previous time steps.

The architecture of RRNs can vary heavily. As in the FFNs the number of hidden layers, depth, and the number of neurons in each layer, with can be adjusted. In addition to this, the fact that RRNs can have connections pointing backwards in the network, additional variations can be made in how these connections are made.

Some design pattern examples from Goodfellow et al. [2] are; RNNs that have output at every time step and connections between the hidden states. This means that the hidden state at time step t is connected to the hidden state at time step $t+1$. Another design could be that the connection only is from the output of the network at time step t is connected to the hidden unit at the next time step $t+1$. Yet another design mention by Goodfellow et al. is a RRN that takes an entire sequence as input and produces a single output, where there a connections between the hidden states.

The choice of design is dependent of the task and data at hand. Due to the nature of the connections in the network it will effect the limitations in prediction and computational demand in training. For example the first mentioned design, where connections are made between the hidden states, cannot be trained in parallel, but they exhibit the flexibility that any function computable by a Turing machine can be computed a finite size RNN of this design [2].

Another way of categorizing RNNs, presented in the lectures, is how input and output takes form. The four categories are:

- One-to-many,
- Many-to-one,
- Many-to-many (with same length input and output),
- Many-to-many (with different length input and output).

One-to-many means that the input is a sequence, consisting of only one element, and the output is a sequence (aka many elements). The rest is self-explanatory. But with a difference in the last two, some RNNs can have a different length of input and output. As we will see later RNNs can be used as building blocks, and combined.

One could think that one is gone, from list of RNNs. The one-to-one. But this one, is one that belong in the list of FNNs.

1. *Basic RNNs*

A problem faced when training such RNNs is the vanishing gradient problem. This problem arises when the gradients of the loss function with respect to the parameters of the network become very small, making it difficult to update the parameters effectively. This can occure in FFNs as well, but since the networks often are limited in depth and with clever choice of activation function an such as the ReLU function, it can be avoided. RNN's on the other hand are often very deep, they can be made to handle sequences of data of any length, the vanishing gradient problem is a common issue in training RNNs. In the backpropegation algorithm the gradients are multiplied due to the chain rule of calculus. If the gradients are small, the product of the gradients will be very small. This opposite effect, exploding gradients, can happen if the gradients are very large.

Another problem basic RNNs inhabit is the ability to use information from a far away time step back, to the current time step.

2. *Bidirectional RNNs*

Bidirectional RNNs are a special type of RNNs, categorized by their special connections. In this networks the output of one time step t, depend not only on the input at time step t, like FNNs. Not only on the input at time step t and previous time steps $t-1, t-2, \ldots$, like basic RNNs. But also on the input at time step $t+1, t+2, \ldots$.

In bidirectional RNNs, two "sub" RNNs are combined. One that reads the input sequence from left to right, and one that reads the input sequence from right to left. The networks are connected such that a output at time step t is dependent on both the left-to-right sub RNN and the right-to-lef sub RNN.

3. *Encoder-Decoder Architechture*

The Encoder-Decoder architechture can also be understood by decomposing the network into two parts. As the name proposes, these parts are called the encoder and the decoder. Both the encoder and the decoder are RNNs. Given a input sequence $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, the encoder produces a representation, C, of the input sequence. The representation could be a single vector, or a sequence. Usually a simply function of the last hidden states of the encoder creates C [2].

Now this representation is passed to the decoder. The decoder RNN is trained and architectured to predict a output sequence, given the representation C of the input sequence. The great flexibility with the Encoder-Decoder architechture is that the sequence input and the output sequence can have different lengths, and they can vary in length as well.

4. *Problems with RNNs*

Although RRNs provide much more flexibility and power then FNNs, especially when working with sequences of data, they still have limitations.

One of them is capturing long-term dependencies. This arises from the vanishing gradient problem [3]. The exist architechtures that try to solve this. Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) are two examples of this [2]. These are on the other hand far more complex and computationally demanding.

As shown in this section the sequential nature of RNNs also makes it hard to find parts to parallelism in the training of the network. Yet another huge drawback.

C. Transformers

The transformer architechture was introduced in the paper "Attention is all you need" by Vaswani et al. in 2017 [7]. The transformer model is a type of neural network that is based on the self-attention mechanism. It is structured in a encoder and decoder, the idea being somewhat similar to the Encoder-Decoder architechture of RNNs discussed earlier. The encoder is responsible for processing the input sequence, turning it into a sequence of hidden states. It takes a sequence $\mathbf{x}$ of length n, $\mathbf{x} = (x_1, x_2, \dots, x_n)$, and produces a sequence with equal length n of hidden states $\mathbf{h}$, $\mathbf{h} = (h_1, h_2, \dots, h_n)$.

In the decoder, the hidden states produced by the Encoder are connected at a specific point in the decoder (as input). Then the decoder produces one output at a time, and the output is used as input in the next time step. The architechture is illustrated in Figure 2. We will not explain in detail all parts of the model, but we will highlight some key features.

Part of the innovation that Vaswani et al. introduced was the scaled dot-product attention mechanism. This mechanism works by having three matrices named Query, Value and Key. These are weights of the model, and are tuned during training as regular weights. The scaled dot-product attention mechanism can be written as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V. \tag{2}$$

Q is the Query matrix, K is the Key matrix, V is the Value matrix and d_k is the dimension of the Key matrix. The reason for the named *scaled* dot-product con be seen in equation 2, where the expression QK^T is divided by $\sqrt{d_k}$ before the softmax function is applied. The benefits from this can be read in the paper [7].

Another key feature is the multi-head attention mechanism. This mechanism is a way to allow the model to focus on different parts of the input sequence. The way this is performed is by linearly project the Query, Key and Value matrices h times, where h is the number of heads. The projection is learned during trining, and the scaled dot-product attention mechanism is applied to each of the projected matrices. After each head has produced a output, the outputs are concatenated and linearly projected to produce the final output. The addition of multi-head attention allows the model to learn

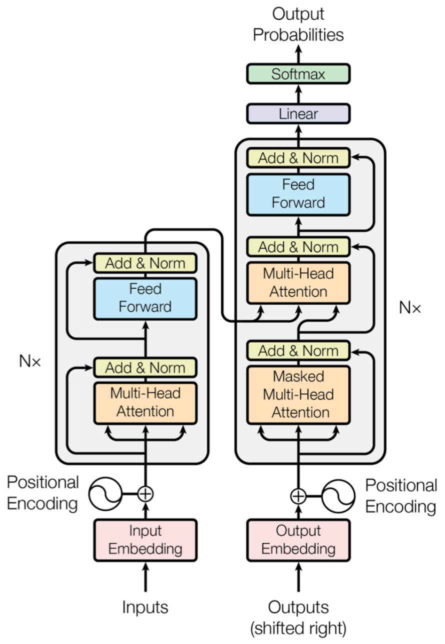

FIG. 2. The architechture of the Transformer model. Presented in- and obtained from the original paper [7].

different features of the input sequence in parallel [7]. It also adds new weights to the model, to perform the linear reduction and , which are tuned during training.

The computation of the Scaled Dot-Product Attention and the Multi-Head Attention is illustrated in Figure 3.

In addition the model consist of feedforward neural networks, like described previously. It also has layer normalization, which include taking the mean and variance of the input and normalize it.

This section has highlighted some key features of the transformer model, and how it differs from traditional RNNs. There is much more to be said, but we will end it here. For more detail please refer to the original paper [7].

1. *BERT*

BERT, which stands for Bidirectional Encoder Representations from Transformers, represents a significant advancement in the field of NLP. Introduced by Devlin et al. in 2019, BERT is designed to pre-train deep bidirectional representations by jointly conditioning on both left and right context in all layers, which is a departure from pre-

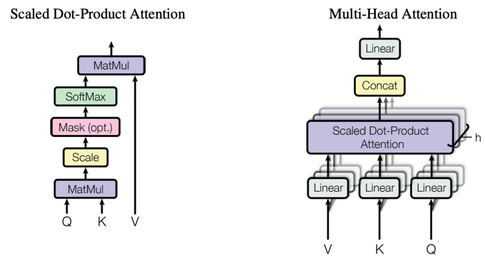

FIG. 3. Schematics of the computation of the Scaled Dot-Product Attention as well as how they combine into the Mulit-Head Attention. Also presented in- and obtained from the original paper [7].

vious unidirectional models like GPT. This bidirectional approach allows BERT to capture more nuanced information from the surrounding context, resulting in improved performance across a variety of NLP tasks without requiring substantial task-specific architecture modifications [1].

The core innovation of BERT lies in its use of the masked language model (MLM) and next sentence prediction (NSP) pre-training objectives. The MLM involves randomly masking some of the tokens in the input sequence and training the model to predict these masked tokens based on their context. Specifically, 15% of the tokens in each sequence are chosen at random to be masked. For the chosen tokens, 80% of the time they are replaced with a special [MASK] token, 10% of the time they are replaced with a random token, and 10% of the time they remain unchanged. This strategy helps the model learn to predict the masked words by using the bidirectional context, allowing it to understand the entire context around a word. The MLM objective can be mathematically expressed as:

$$L_{\text{MLM}} = -\sum_{i \in M} \log P(x_i | x_{\setminus M}), \tag{3}$$

where M is the set of masked positions, x_i is the original token at position i, and $x_{\setminus M}$ represents the input sequence with the masked tokens.

The NSP task involves predicting whether a given pair of sentences is sequentially coherent, which helps the model understand the relationship between sentence pairs. During pre-training, 50% of the time, the second sentence in the pair is the actual next sentence, and 50% of the time, it is a random sentence from the corpus. This task is crucial for improving the model's performance on tasks that require an understanding of the relationship between sentences, such as question answering and natural language inference [1].

BERT's architecture is based on the Transformer model, specifically utilizing the encoder part of the transformer. It comes in two primary versions: BERTBASE and BERTLARGE, differing in the number of layers, hidden units, and attention heads. BERTBASE consists of 12 layers, 768 hidden units, and 12 attention heads, while BERTLARGE has 24 layers, 1024 hidden units, and 16 attention heads. This structure allows BERT to handle complex language tasks by providing a deep, richly contextualized representation of words and sentences, which can then be fine-tuned on specific tasks such as question answering, sentiment analysis, and named entity recognition [1].

One of the key advantages of BERT is its ability to be fine-tuned for a variety of downstream tasks with minimal architectural changes. During fine-tuning, the pre-trained BERT model is simply extended with an additional output layer appropriate for the task at hand, and the entire model is trained end-to-end. This approach significantly reduces the need for task-specific model engineering and allows BERT to achieve state-of-the-art performance on a wide range of NLP benchmarks, such as the General Language Understanding Evaluation (GLUE) benchmark, the Stanford Question Answering Dataset (SQuAD), and more [1].

The impact of BERT on the field of NLP has been profound. By providing a robust, pre-trained model that can be easily adapted to various tasks, BERT has democratized access to powerful NLP capabilities. Researchers and practitioners can leverage BERT to improve the accuracy and efficiency of their models without needing extensive computational resources or specialized knowledge. This has led to a surge in the development and deployment of NLP applications across industries, further advancing the capabilities and accessibility of AI-driven language understanding [1].

III. TEXT EMBEDDINGS MODELS

Now we return to the main topic of this report, text embeddings. The task of text embeddings is to represent a text in a meaningful way, that a computer can understand. This is done by mapping the text into a vector space.

There are many ways this can be done, one can for example count the number of times a word appears in a text, and then represent the text as a vector where each element is the count of a word. This type of representation is called a bag of words representation, but a key downside with representation like this is that the order of the words are lost.

For example consider the two sentences:

- ”I love statistical mechanics, but i hate quantum mechanics”

- ”I hate quantum mechanics, but i love statistical mechanics”

These two sentences have the same bag of words representation, but they have different meanings.

Different from the bag of words representation, transformer based text embedding models are able to capture the order and relation between the words in a text.

A. Sentence-BERT

To understand exactly how transformer based text embedding models can capture the order and relation between the words in a text, we will look at the Sentence-BERT model. This model was one of the first of it’s type to utilize the transformer architecture to create sentence embeddings presented by Reimers and Gurevych in 2019 [6].

BERT (Bidirectional Encoder Representations from Transformers) set new state-of-the-art performance on various sentence classification and sentence-pair regression tasks. However, it requires that both sentences are fed into the network simultaneously, causing a massive computational overhead. For instance, finding the most similar pair in a collection of $10,000$ sentences requires approximately 50 million inference computations, which is highly inefficient [6].

Sentence-BERT (SBERT) addresses this inefficiency by using a siamese network architecture. Siamese networks consist of two identical subnetworks joined at their outputs. These networks share the same weights and are used to generate comparable embeddings for two input sentences. SBERT fine-tunes BERT in such a way that it can produce semantically meaningful sentence embeddings that can be compared using cosine similarity or other distance measures like Euclidean (L2) or Manhattan distance (L1).

The main innovation of SBERT is its ability to derive fixed-size sentence embeddings that capture semantic meaning effectively. This is achieved by fine-tuning BERT with a triplet network structure, where the network learns to minimize the distance between embeddings of similar sentences while maximizing the distance between embeddings of dissimilar sentences. This method significantly reduces the computation time required for tasks such as semantic similarity search and clustering. For example, SBERT can reduce the effort of finding the most similar pair in a collection of 10,000 sentences from 65 hours with BERT to about 5 seconds.

IV. TEXT EMBEDDINGS AS FEATURES

As we have just seen when we pass a text as input though a text embedding model, what the model outputs is a vector. Mathematically written a sequence of inputs tokens I, where $I = \{i_1, i_2, ..., i_t\}$, is mapped to a high dimensional vector V, where $V = \{v_1, v_2, ..., v_d\}$. Where d is the dimension of the vector, which will vary from model to model. Models based on BERT usually have a dimension of either 512 or 768 as noted from experience, while other models might have a different dimension. t is the number of tokens in the input sequence.

Vectors in a high dimensional space are hard to visualize, but we can use dimension reduction techniques to reduce the dimension. This will allow us to visualize the vectors in a 2D or 3D space, but we have to remember that the reduced dimension space is not the same as the original one, and much information might be lost.

Techniques like PCA, t-SNE, UMAP, and others can be used to reduce the dimensions, and they have their pros and cons. PCA is a linear technique and computes rather quickly, but it might not be able to capture the non-linear relationships in the data. t-SNE is a non-linear technique as well as UMAP.

Since the vectors outputted from the text embedding models are representations of the texts embedded, they can be used as features to predict or classify texts. Unsupervised they can be categories to group similar texts together. Techniques like K-means clustering or DBSCAN works well with text embeddings. If labels are present in the data, the text embeddings can be used as features to train a supervised model.

V. CLASSIFYING NEWS CATEGORIES FROM TITLES

A. Data

The data for this experiment is gather from kaggle[1] and consist of 69588 titles of news article, gathered from Google News UK, witch a category label for each title. The categories and the number of titles in each category are shown in Table I [5].

[1] Link to the dataset: `https://www.kaggle.com/datasets/victornuez/google-news-uk-2010-2024`

TABLE I. Number of titles in each category.

Category name	Number of titles
Crime	6714
Culture	3267
Economy	3179
Education	5195
Entertainment	5018
Environment	3197
Health	6422
International	4408
Police	8017
Politics	4478
Science	4327
Sports	5027
Technology	2960
Travel	6479

B. Method

To embed the news titles, I use a open source model from the the team mixedbread[2], called *mxbai-embed-large-v1*[3]. The dimension of the embeddings produced by this model is 1024 [4].

I set up a simple FNN model to classify the news titles into the categories, implementing it with the PyTorch library. The model consists of an input layer, a hidden layer, and an output layer, and are fully connected. The input layer has the same dimension as the output from the text embedding model, and the output layer has the same dimension as the number of categories.

The network is setup with two hidden layers, the first with 512 neurons and the second with 256 neurons. As the activation function, I use the ReLU function, and for the output layer, I use the softmax function. The loss function is the cross-entropy loss function, and for the optimizer, I use the Adam optimizer with a learning rate of 0.001.

I train the model for 40 epochs, and I use a batch size of 32. 80% of the data is used for training, and 20% is used for validation. For measuring the performance of the model, I use the accuracy score, which is the number of correct predictions divided by the total number of predictions.

C. Results and Discussion

The accuracy of the model on the validation set is 88.36%. This is for one run of the model, and the accuracy can vary slightly between runs due to the random of splitting the data and dividing the batches. A natural next step for this project would be to do a more thorough hyperparameter search, as well as quantify how much the accuracy can vary between runs.

There is much more to be done, and this mini experiment being more just a demonstration of how transformer based text embeddings can be used. With more experimentation, the model surly can be improved.

VI. DISCUSSION

Transformer based text embedding models are a useful tool to represent text. In differ from the RNNs they can be more effectively parallelized, and can learn longer dependencies in text. They also are able to deal with sequential data much more naturally than the FNNs.

Text embeddings can effectively be for a variety of different downstream tasks in easy matter. In this mini experiment I showed how text embeddings can be used to classify news articles based on their titles, using a simple FNN. The results were promising. Although the model was not very complex it reached a accuracy of 0.85 on the validation set.

Further research can be done to improve the model. Different hyperparameters should be tested, as well as looking into other possibilities for activation functions, optimizers and learning rates etc.

VII. CONCLUSION

In this report I have presented how Transformer based text embedding models work, starting from the basics of Feed Forward Neural Networks. Then building upon this with the Recurrent Neural Network architecture, which has been important in the field of Natural Language Processing (NLP), highlighting a couple of limitations which the Transformer architecture to some extent solves. In addition a mini experiment was conducted to train a simple FNN to classify news articles based on text embedding of their titles. The text embedding chosen was from a open source model from the team mixedbread, called *mxbai-embed-large-v1*, and the accuracy achieved was 88.36%.

[2] Link to the mixedbread home page: https://www.mixedbread.ai/

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

[2] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org [Accessed: 9 Juni, 2024].

[3] Hjorth-Jensen, M. (2024). Fys5429-advanced advanced machine learning and data analysis for the physical sciences - lecture notes. https://github.com/CompPhysics/AdvancedMachineLearning/tree/main/doc/pub. [Accessed: 13 Juni, 2024].

[4] Li, X. and Li, J. (2023). Angle-optimized text embeddings. *arXiv preprint arXiv:2309.12871*.

[5] Núñez, V. (2024). Google news uk dataset (20110-2024). https://www.kaggle.com/datasets/victornuez/google-news-uk-2010-2024. [Accessed: 13 Juni, 2024].

[6] Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks.

[7] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

APPENDIX - GENERATIVE AI TOOLS

As mentioned in the introduction, some of the breakthroughs in the field of AI that in this report has been presented, has pioneered the development of new tools that can assist in both writing reports and code. One of the most prominent being chatGPT Since the use of these tools are still in its infancy and regulations are still somewhat unclear, I will describe in detail how and why I have used them when writing the report and code.

A. UiOGPT

I have carefully used the UiOGPT to assist in writing, mostly for spelling and grammar checks. I have prompted the UiOGPT with questions like "Are there any spelling errors in this paragraph: ..." where for the dots I have included the paragraph I wanted to check, or "What is a better way of phrasing this sentence: ...".

Each time I have made sure what the UiOGPT has suggested is what I originally intended to write, that it is correct, and that I understand the suggestion. In some cases I have also followed up with a second question; "Why is this a better way of phrasing this sentence?". This has helped me understand the suggestions better, and in some cases I hopefully it will help me write more accurate and communicate clearer in future reports.

B. Github copilot

When writing code, most of the time I have the Github copilot plugin activated. Some of the comments and code snippets in this report has been generated by the copilot plugin, usually in cases where where there is simple or repetitive code, which I have written many times before.

[3] Link to the huggingface model card https://huggingface.co/mixedbread-ai/mxbai-embed-large-v1