NLP Financial Sentiment Analysis from Naïve Bayes to Transformers

Nathaniel Cogneaux - Paul-Emile Galine - François Jordan - Jessica Iacob

May 22, 2024

Abstract

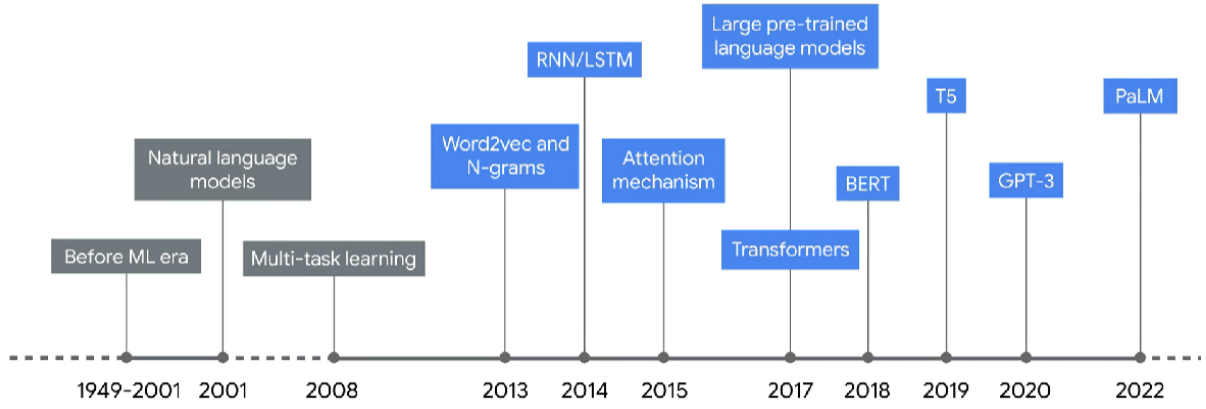
In the realm of deep learning, the processing of sequential data is crucial for many applications, such as natural language understanding, speech recognition, and time series forecasting. Among the various architectures developed for these tasks, Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Transformers have become essential tools. This comprehensive guide delves into these models, exploring their strengths, weaknesses, and practical uses in detail. Additionally, we will demonstrate that the Naïve Bayes algorithm, despite its simplicity, remains a compelling and effective option for handling sequential data in some scenarios.

Contents

1 Introduction

In the realm of statistical learning, text classification stands as a cornerstone of natural language processing (NLP). Text and words permeate our daily lives, shaping our interactions, decisions, and understanding of the world. The significance of text classification extends across various applications, from sentiment analysis and spam detection to more nuanced fields such as financial sentiment analysis.

This project aims to categorize financial sentiment, determining whether texts are positive, neutral, or negative from a financial perspective. We will compare the performance of several algorithms, starting from the fundamental Naïve Bayes algorithm to more advanced models such as Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM) networks, and Transformers.



Throughout this project, we will evaluate the architectures, strengths, weaknesses, and accuracy of these models. By systematically analyzing these diverse approaches, we aim to identify the most effective techniques for financial sentiment classification, providing insights into their practical applications and potential limitations.

2 A Simple Baseline: Naïve Bayes

In statistics, the naïve Bayes classification is a classical method considered as the simplest and the baseline for the classification of sentences. These probbilistic classifiers are based on the Baye's theorem which provides a way of calculating posterior probabilities based on prior knowledge. Despite it's simplicity, the naïve Bayes algorithm often delivers good performance in various application, ranging from spam detection to sentiment analysis, language identification..

2.1 Advantages of Naïve Bayes

Naïve Bayes algorithms are widely use, and despite their simplicity to implement and understand, they have some strong advantages.

1. **Simplicity:** As we said, the first and most important advantage of naïve Bayes algorithm are their simplicity, they are straightforward to understand and implement. The underlying principles are based on Bayes' theorem and the assumption of feature independence, which simplifies the mathematical computations involved.

2. **Efficiency and Speed:** Training and prediction with Naive Bayes are both fast and computationally efficient. This makes it suitable for real-time applications and scenarios where quick decision-making is crucial.

3. **Scalability:** These algorithm are easily scalable and can handle very large datasets with ease which provide adaptability.

4. **Good Performance with Small Datasets:** Sometime forgetted, these algorithm are easily scalable and can handle very large datasets with ease. It also provide surprisingly good results with small datasets and a small amount of data.

These advantages make naïve Bayes algorithm a valuable tool to handle and a reference for machine learning practitioners, it also provide a good baseline to compare its results with some other more complex algorithm.

2.2 Limitations of Naïve Bayes

Despite the straightforward advantages of the naïve Bayes algorithm, they also come with a set of limitations that can affect their performance and applicability in certain scenarios.

1. **Assumption of Independence:** As we said, these algorithms are based on a very strong assumption, all the features (words) are independent of each other given the

class of label. This strong assumption rarely holds true in the real-world where words and sentences often exhibit a strong correlation. Hence, this assumption can reduce their performance.

2. **Zero Probability Problem:** If a particular word in the dataset is not seen in the training set, Naive Bayes will assign a zero probability to the corresponding class, effectively eliminating it as a possibility. This issue can be remove using Laplace smoothing, but it still reveals the limitation of these algorithms handling unseen data.

3. **Limited Expressiveness:** The simplicity of naïve Bayes is a double-edged sword. It'll ensure efficiency and scalability, but also the model will have an issue to capture the complexity of the dataset compared to more sophisticated algorithms like neural networks.

In conclusion, while naïve Bayes classifiers are powerful and simple tools for many text classification tasks, it has also a lot of limitations that have to be taken in mind and considered.

2.3 Way of working

The naïve Bayes algorithm is based on the Bayes Theorem which states that for two events A and B, we have

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)P(A)}{P(B)}.$$

In this particular project, each sentence can be classified into one of three categories: negative, positive, or neutral. The main idea is to estimate the probability that a sentence is either negative, positive or neutral given the feature (the sentence). Thanks to the Bayes Theorem and the assumption that all the features are independent conditionally to the label, we can easily estimate the three probabilities and give a result. This means we assume that the presence of a word in a sentence does not depend on the presence of any other word, given the category of the sentence. This assumption simplifies the computation and allows us to estimate the probabilities easily.

1. **Training Phase:** During the training, the algorithm will count how often each word appears in sentences of each category (negative, positive or neutral), and will estimate the probability that we need.

2. **Prediction Phase:** When a new sentence needs to be classified, the algorithm looks at the words in the sentence. If certain words frequently appear in positive

sentences let say, in the training data, the new sentence is likely to be classified as positive. Similarly, if the words commonly appear in negative or neutral sentences, the sentence will likely be classified accordingly.

For example, if we are given a new sentence for financial sentiment analysis, which is "The results for this year are very good!", during the training phase, the algorithm learned that words like "good" often appear in positive sentences. So, when the algorithm sees this word in a new sentence, it'll be likely considered as a positive sentiment.
In summary, the Naive Bayes classifier's ease of implementation and understanding is due to its reliance on word occurrences. Given a new sentence, if the words in the sentence commonly appear in positive sentences, the classifier will likely label it as positive, and similarly for negative and neutral categories.

2.4 Real-World Applications

For the project, we took 20% of the dataset (randomly) for the test size, and 80% for the training set. After the training phase, we get a surprisingly good accuracy on the test set with a strong 70% of accuracy, the results can be seen on the figure below.

```
Accuracy: 0.7110423116615067
Classification Report:
              precision    recall  f1-score   support

    negative       0.69      0.44      0.54       122
     neutral       0.77      0.87      0.82       589
    positive       0.55      0.48      0.51       258

    accuracy                           0.71       969
   macro avg       0.67      0.60      0.62       969
weighted avg       0.70      0.71      0.70       969
```

Figure 1: Results

For the three categories, we get the following results

1. **Negative:** Precision: 69% on 122 occurences,
2. **Neutral:** Precision: 77% on 589 occurences,
3. **Positive:** Precision: 55% on 258 occurences.

We can see that the neutral category outweighs the two other categories, but the difference of the precision if we treat the three categories as equal or considering their weights is not

important, indeed we get 67% for the average precision treating the categories equally, and 70% for the weighted average precision.
In summary, even if the algorithm is one of the simplest for the text classification, it provided a surprisingly good result and prediction, when we look at its execution and implementation time.

3 The Foundation: Recurrent Neural Networks (RNNs)

RNNs, with their recurrent connections, laid the foundation for sequence modeling. They process data sequentially, maintaining a hidden state that captures information from previous time steps. This enables them to handle sequential dependencies naturally.

3.1 Model Architecture

Assume we want to build a neural network to process a data sequence $x_{1:T} = (x_1, ..., x_T)$. Recurrent Neural Networks (RNNs) are neural networks suited for processing sequential data, which, if well trained, can model dependencies within a sequence of arbitrary length. We also assume a supervised learning task which aims to learn the mapping from inputs $x_{1:T}$ to outputs $y_{1:T} = (y_1, ..., y_T)$. Then a simple RNN computes the following mapping for $t = 1, ..., T$:

$$h_t = \varphi_h(W_h h_{t-1} + W_x x_t + b_h),$$

$$y_t = \varphi_y(W_y h_t + b_y).$$

Here the network parameters are $\theta = \{W_h, W_x, W_y, b_h, b_y\}$, φ_h and φ_y are the non-linear activation functions for the hidden state h_t and the output y_t, respectively. For $t = 1$ the convention is to set $h_0 = 0$ so that $h_1 = \varphi_h(W_x x_1 + b_h)$; alternatively h_0 can also be added to θ as a learnable parameter.

Steps in the RNN Model Architecture:

1. **Input Layer:** Takes in the sequence of data points.

2. **Embedding Layer (optional):** Converts input tokens into dense vectors of fixed size, which capture semantic meaning and relationships between tokens.

3. **Recurrent Layer:** Processes the sequence of inputs and maintains a hidden state that is updated at each time step:

- The hidden state h_t at time t depends on the input at t (x_t) and the hidden state of the previous time step (h_{t-1}).

4. **Output Layer:** Produces the output y_t for each time step, which can be a classification or regression output depending on the task.

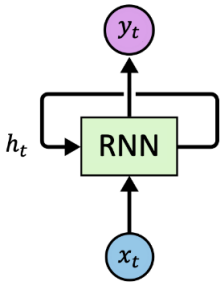


Figure 2: Diagram of a Simple RNN Architecture

Additional Architectural Components:

- **Activation Functions:** Non-linear functions like tanh or ReLU are used for the hidden state h_t. Softmax is commonly used for the output layer in classification tasks.

- **Sequence Processing:** The network processes sequences in a time-dependent manner, updating the hidden state at each time step to capture temporal dependencies.

- **Parameter Sharing:** The weights W_h, W_x, and W_y are shared across all time steps, which helps in learning consistent patterns across the sequence.

3.2 Advantages of RNNs

- **Sequential Dependencies:** RNNs are well-suited for tasks where past information significantly impacts future predictions, such as language modeling and time series forecasting.

- **Simple and Intuitive:** The simplicity of RNNs makes them easy to understand and implement.

- **Parameter Sharing:** RNNs share parameters across different time steps, which helps in generalizing across sequences of varying lengths.

- **Flexibility with Input Lengths:** RNNs can handle input sequences of varying lengths, making them versatile for different types of sequential data.

3.3 Limitations of RNNs

- **Vanishing and Exploding Gradients:** RNNs often suffer from vanishing or exploding gradients, making it challenging to capture long-range dependencies.
- **Difficulty with Parallelism:** Due to their sequential nature, RNNs are not naturally parallelizable, leading to slower training times.
- **Short-Term Memory:** RNNs tend to have difficulty retaining information over long sequences due to their inherent design.
- **Training Instability:** RNNs can be difficult to train, requiring careful tuning of hyperparameters and use of techniques like gradient clipping.

3.4 Real-World Applications

Despite their limitations, RNNs have been successfully applied in various domains:

- **Language Modeling and Text Generation:** RNNs can generate coherent and contextually relevant text sequences, making them useful in chatbots and language translation systems.
- **Speech Recognition:** RNNs have been used to transcribe spoken language into text, improving the accuracy of voice-activated systems.
- **Time Series Prediction:** RNNs are effective for forecasting future values in time series data, such as stock prices and weather patterns.
- **Music Composition:** RNNs can compose music by learning patterns in sequences of musical notes.

3.5 Results on the NLP for Financial Sentiment Analysis

For the task of financial sentiment analysis, we implemented an RNN to classify the sentiment of financial news articles into positive, negative, or neutral categories. The steps for building and evaluating the RNN model are as follows:

3.5.1 Data Preprocessing

The text data was preprocessed by:

- Removing non-alphanumeric characters.
- Converting text to lowercase.
- Tokenizing the text into sequences of words.
- Padding sequences to ensure uniform input length.

3.5.2 Model Architecture

We utilized a bidirectional RNN model with the following architecture:

- An embedding layer with a vocabulary size of 5001 and an embedding dimension of 128.
- A dropout layer with a dropout probability of 0.5.
- A fully connected layer with an output dimension of 3 (for the three sentiment categories).
- An attention mecanism which helps the model focus on relevant parts of the input sequence. It improves model performance and interpretability by highlighting important input parts.

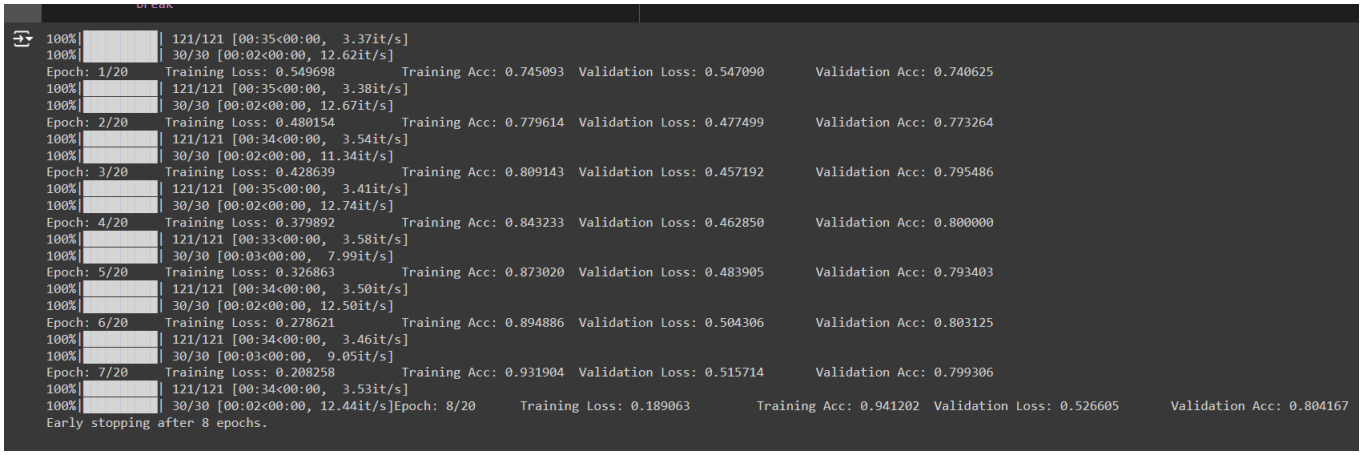
3.5.3 Training and Evaluation

The model was trained for a single epoch with a batch size of 32. The following loss function and optimizer were used:

- Loss function: Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss)
- Optimizer: Adam with a learning rate of 0.001

3.5.4 Results

After training the model for one epoch, we evaluated its performance on the training and the validation set. The results are as follows:



Thus, we obtain :

- Accuracy on training set : 94%
- Accuracy on validation set : 80%.

3.5.5 Conclusion

The results indicate that the RNN model performs reasonably well in classifying the sentiment of financial news articles. Given the potential for more complex temporal dependencies in financial text data, a more advanced architecture such as Long Short-Term Memory (LSTM) networks might better capture long-range dependencies and improve performance. LSTMs are designed to mitigate issues like the vanishing gradient problem, which can affect standard RNNs when dealing with longer sequences. Future work will involve experimenting with LSTM models to enhance the classification performance further and explore other hyperparameter tuning and architecture optimizations.

4 The Evolution: Long Short-Term Memory (LSTM) Networks

LSTM networks where introduced in 1997 by Hochreiter and Schmidhuberemerged and have since become a cornerstone in various applications. LTMSs are a neural network architecture widely used for natural language processing and offer a solution to some of the challenges faced by current RNNs. Indeed, one of main issues of those RNN is that their short-term memory does not last long enough for certain tasks because of a famous problem called the Vanishing Gradient Problem : it refers to the phenomenon where the gradients (derivatives) of the loss function with respect to the weights become extremely small as they are backpropagated to earlier layers of the network. So, LSTM introduces a gating mechanism that enables the network to learn when to retain or forget information from previous time steps. They contain a short-term memory that can last long enough to be considered long short-term memory. As a result, we avoid the gradient vanishing problem, although it can still happen that our gradient explodes.

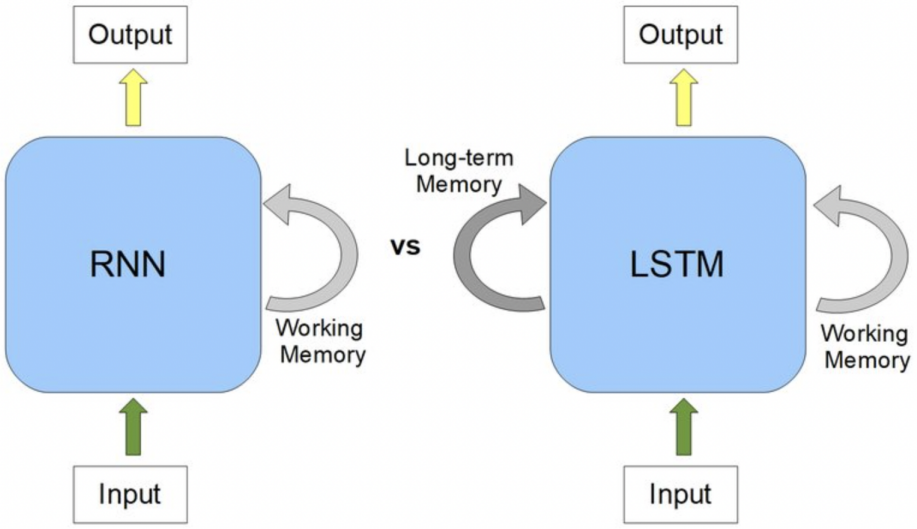


Figure 3: RNN vs LSTM

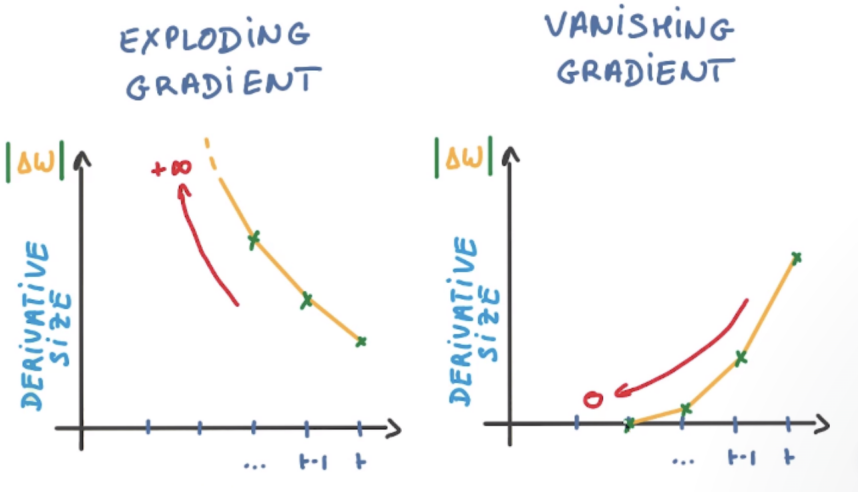


Figure 4: Difference between exploding and vanishing gradient

4.1 The Particular architecture

The LSTM particularity is they introduce a new unit called a memory cell, which allows the network to store and access information over an extended period of time. The memory cell in LSTM serves as a conduit for information flow through the network over time. Unlike traditional recurrent neural networks (RNNs), which have a single hidden state, LSTMs introduce a more complex structure to maintain long-term dependencies. The memory cell itself is a linear unit that accepts inputs and outputs at each time step and has its state, independent of the hidden layer.

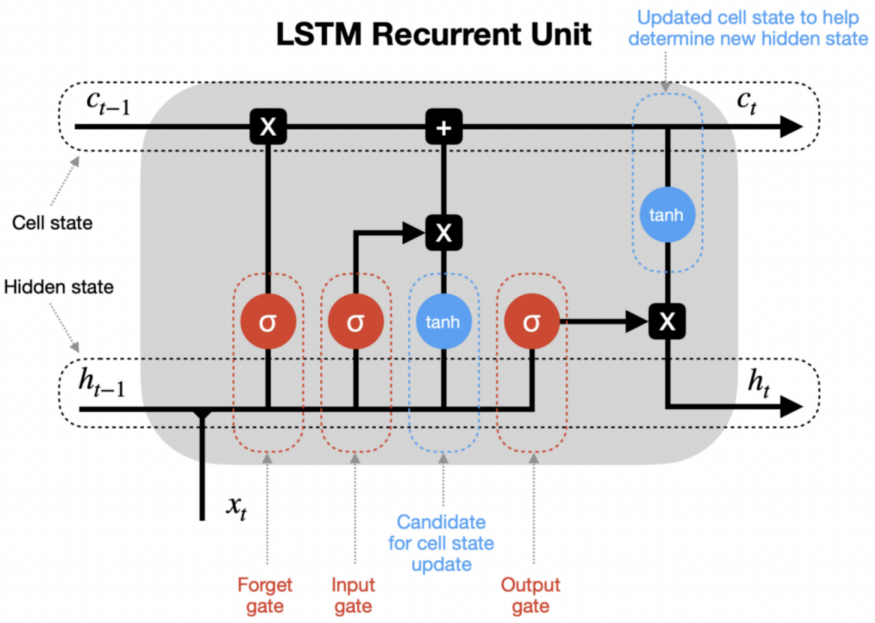


Figure 5: Example of a memory cell in LSTM

The memory cell of an LSTM is composed of several gates: an input gate, an output gate, and a forget gate. These gates regulate the flow of information within the memory cell, allowing control over which information to retain and which to forget. This gives LSTMs the ability to memorize important information over long sequences and ignore less relevant elements. In standard RNNs, h_t represents the hidden state, but in LSTM networks, we add a second state called c_t. Here, h_t represents the short-term memory of the neuron, and c_t represents the long-term memory.

1. The **input gate (1)** determines which information should be updated in the memory cell. It considers the current input x_t and the previous state of the memory cell h_{t-1}, generating an activation vector that represents the information to be added to the cell c_{t-1}. This addition of information is achieved through a mathematical operation performed between this activation vector and the previous state c_{t-1}.

2. The **forget gate (2)** allows the LSTM to discard obsolete information from the memory cell. It uses both the current input and the previous state to generate an activation vector that determines which information should be forgotten. This also involves a mathematical operation between this second activation vector and the previous state c_{t-1}. The combination of these two operations on c_{t-1} results in the current state c_t.

3. The **output gate (3)** determines the LSTM's output at a given time. It uses the current input x_t and the current state of the memory cell c_t to generate an activation vector that represents the LSTM's output. This is how h_t is obtained.

Then, the LSTM updates its hidden state h_t and memory cell state c_t using the following function:

$$h_t, c_t = f(x_t, h_{t-1}, c_{t-1})$$

The combination of these three gates allows the LSTM network to effectively manage long-term dependencies. During backpropagation, LSTMs can maintain a steady flow of information over time, thus avoiding the vanishing gradient problem and enabling more stable and accurate learning.

4.2 Advantages of LSTMs

LSTM networks offer several advantages over traditional RNNs:

1. **Long-Term Dependency Handling:** LSTMs are able to capture and learn from long-term dependencies in sequential data.

2. **Memory Cell:** As mentioned in previous section, they use a memory cell to maintain information over long sequences, allowing them to remember relevant information while ignoring irrelevant elements.

3. **Gate Mechanisms:** LSTM's three gate mechanisms control the flow of information into and out of the memory cell, enabling precise control over which information is retained or discarded.

4. **Avoidance of Vanishing Gradient Problem:** By regulating the flow of information through the gates, LSTMs mitigate the vanishing gradient problem, ensuring stable and effective learning over long sequences.

They provide a powerful solution for tasks requiring modeling of long-term dependencies in sequential data, offer more sophisticated handling of sequential information compared to traditional RNNs and are capable of learning complex patterns over time. While Naive Bayes classifiers have their strengths in simplicity and scalability, they lack the capability to model sequential dependencies. Therefore, the choice of model depends on the specific requirements of the task and the nature of the data.

4.3 Limitations of LSTMs

Despite their advantages, LSTM (Long Short-Term Memory) networks have certain limitations:

1. **Complexity:** LSTMs are more complex than traditional RNNs, which can make them harder to train and require more computational resources.

2. **Difficulty in Capturing Very Long-Term Dependencies:** While LSTMs are designed to capture long-term dependencies, they can still struggle with very long sequences where dependencies are extremely distant.

3. **Overfitting on Small Datasets:** Like all deep learning models, LSTMs are prone to overfitting when trained on small datasets. Regularization techniques and careful hyperparameter tuning are necessary to mitigate this.

4. **Interpretability:** The internal workings of LSTMs, particularly how information flows through the gates and memory cell, can be complex and difficult to interpret.

5. **Training Time:** Training LSTMs can be time-consuming, especially when dealing with large datasets and complex architectures.

6. **Gradient Explosion:** Although LSTMs mitigate the vanishing gradient problem, gradient explosion can still occur in deep networks, requiring techniques like gradient clipping.

These limitations should be considered when choosing and implementing LSTM models in various applications.

4.4 Real-World Applications

The LSTM networks have revolutionized the field of deep learning by effectively handling long-term dependencies in sequential data. Their ability to memorize and utilize information over long periods of time opens new avenues in many application domains. They provide a powerful and promising approach for modeling complex data and capturing temporal patterns. Their applications span a multitude of fields, including but not limited to :

- Robot control
- Time series prediction
- Speech recognition
- Rhythm learning
- Language modeling
- Hydrological rainfall–runoff modeling
- Music composition
- Grammar learning
- Handwriting recognition
- Human action recognition.
- Financial forecasting

As research continues, LSTM networks are likely to evolve further and play a pivotal role in future advances in artificial intelligence and machine learning.

4.5 Results on the NLP for Financial Sentiment Analysis

Sentiment analysis involves understanding and extracting sentiments or opinions expressed in text data. In the context of financial news, sentiment analysis can help investors and analysts gauge market sentiment and make informed decisions. So we thought about using a LSTM model to do this.
We have choose to code the model using mainly the PyTorch library as it provides both flexibility and speed in building deep learning models.
First, text data is preprocessed to remove non-alphanumeric characters, convert text to lowercase, and strip extra whitespaces. This step ensures that the data is normalized and ready for further processing.A Tokenizer class is defined to tokenize the text data and pad sequences to a fixed length. It is used to fit on the text data, convert texts to sequences, and pad these sequences. The conversion enables the model to process categorical data and each word is assigned a unique index, which allows the model to convert words into numerical representations. These transformations prepare the text data for input into the LSTM model.
Then, we define a bidirectional LSTM Model (LSTMClassifier):

- An embedding layer is used to convert the input word indices into dense word vectors. This layer helps represent words in a continuous vector space.
- A bidirectional LSTM layer is employed to capture sequential dependencies in the input text. This aspect allows the model to learn from both past and future contexts.
- An attention mecanism which helps the model focus on relevant parts of the input sequence. It improves model performance and interpretability by highlighting important input parts.
- A dropout layer to prevent overfitting. It randomly sets a fraction of neurons to zero during training and scales the remaining neurons.
- A fully conntected layer which allows the model to learn complex feature combinations and map them to the desired output classes. Each neuron is connected to every neuron in the previous layer, performing a linear transformation followed by a non-linear activation.

```python
#define the LSTM model
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, no_layers, drop_prob=0.5):
        super(LSTMClassifier, self).__init__()
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim
        self.no_layers = no_layers
        self.vocab_size = vocab_size

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, no_layers, dropout=drop_prob, batch_first=True, bidirectional=True)

        self.dropout = nn.Dropout(drop_prob)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)  # hidden_dim * 2 because of bidirectional LSTM

        self.attention = nn.Linear(hidden_dim * 2, 1)

    def forward(self, x, hidden):
        batch_size = x.size(0)

        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        attention_w = torch.softmax(self.attention(lstm_out).squeeze(), dim=-1).unsqueeze(-1)
        attention_out = torch.sum(attention_w * lstm_out, dim=1)

        out = self.dropout(attention_out)
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.no_layers * 2, batch_size, self.hidden_dim).zero_(),
                  weight.new(self.no_layers * 2, batch_size, self.hidden_dim).zero_())

        return hidden
```

```
LSTMClassifier(
  (embedding): Embedding(10001, 128)
  (lstm): LSTM(128, 256, num_layers=2, batch_first=True, dropout=0.5, bidirectional=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=512, out_features=3, bias=True)
  (attention): Linear(in_features=512, out_features=1, bias=True)
)
```

Finally, a training function was defined to train the model for a specified number of epochs. During training, the model learns to minimize the Logit loss using the Adam optimizer. The model was evaluated on both training and validation sets during and after training. This evaluation helps monitor loss and accuracy, ensuring that the model is learning effectively and generalizing well. We obtain following results :

```
100%|████████████████████████████████████| 60/60 [02:09<00:00,  2.15s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.33it/s]
Epoch: 1/20     Training Loss: 0.537652         Training Acc: 0.742448  Validation Loss: 0.489264       Validation Acc: 0.763542
100%|████████████████████████████████████| 60/60 [02:06<00:00,  2.12s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.29it/s]
Epoch: 2/20     Training Loss: 0.473857         Training Acc: 0.783160  Validation Loss: 0.465974       Validation Acc: 0.791667
100%|████████████████████████████████████| 60/60 [02:17<00:00,  2.29s/it]
100%|████████████████████████████████████| 15/15 [00:12<00:00,  1.22it/s]
Epoch: 3/20     Training Loss: 0.400843         Training Acc: 0.830990  Validation Loss: 0.445963       Validation Acc: 0.797569
100%|████████████████████████████████████| 60/60 [02:12<00:00,  2.21s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.32it/s]
Epoch: 4/20     Training Loss: 0.307773         Training Acc: 0.883333  Validation Loss: 0.477009       Validation Acc: 0.802431
100%|████████████████████████████████████| 60/60 [02:06<00:00,  2.11s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.33it/s]
Epoch: 5/20     Training Loss: 0.227857         Training Acc: 0.919705  Validation Loss: 0.468795       Validation Acc: 0.816667
100%|████████████████████████████████████| 60/60 [02:05<00:00,  2.09s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.32it/s]
Epoch: 6/20     Training Loss: 0.163078         Training Acc: 0.945139  Validation Loss: 0.611758       Validation Acc: 0.802431
100%|████████████████████████████████████| 60/60 [02:07<00:00,  2.12s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.32it/s]
Epoch: 7/20     Training Loss: 0.107654         Training Acc: 0.967188  Validation Loss: 0.616793       Validation Acc: 0.804167
100%|████████████████████████████████████| 60/60 [02:07<00:00,  2.12s/it]
100%|████████████████████████████████████| 15/15 [00:11<00:00,  1.34it/s]
Epoch: 8/20     Training Loss: 0.089326         Training Acc: 0.974132  Validation Loss: 0.649382       Validation Acc: 0.805208
Early stopping after 8 epochs.
```

Thus, we obtain :

- Accuracy on training set : 97%
- Accuracy on validation set : 81%.

The result of the model is better with LSTM compared to Naive Bayes or traditional RNNs. On one hand, Naive Bayes is limited by the assumption of feature independence and lack of sequence handling, which reduces its effectiveness for tasks requiring contextual understanding. On the other hand, RNN suffers from the vanishing gradient problem, limiting its ability to learn from long sequences, making it less effective than LSTM for tasks requiring long-term dependencies. Overall, the superior performance of LSTM in capturing the intricacies of sequential data and maintaining long-term dependencies generally results in better accuracy for tasks such as sentiment analysis compared to Naive Bayes and traditional RNNs. We finally have to see if the LSTM model is better or not than the Transformers model.

5 The Revolution: Transformers

Transformers, introduced in 2017, have transformed sequential data processing. Unlike RNNs and LSTMs, transformers process data in parallel and simultaneously capture relationships between all elements in a sequence [1].
RNNs are notably troubled by vanishing and exploding gradients, causing the model to favor the most recent inputs. Consequently, older inputs have minimal impact on the current output. LSTMs and GRUs address this issue by incorporating separate memory cells and additional gates to manage the flow of information. However, information from past steps still traverses a sequence of computations, relying on these mechanisms to convey past information to the current step.

5.1 Advantages of Transformers

RNNs and LSTMs face significant challenges in parallelizing sentence processing, as each word must be processed sequentially. These models also struggle with capturing long-range dependencies.
Transformers, introduced in the seminal paper *Attention is All You Need* [1], were designed to overcome these limitations, especially in machine translation. Their primary goal was to eliminate recursion, enabling parallel computation, reducing training time, and enhancing performance on tasks with long dependencies. Key features of transformers include:

- **Non-Sequential Processing:** Unlike RNNs and LSTMs, transformers process sentences as a whole rather than word by word.
- **Self-Attention:** This mechanism calculates similarity scores between words in a sentence, allowing the model to understand the relationships between all words simultaneously. At each step, transformers have direct access to all other steps (self-attention), reducing information loss during message passing. They can consider both future and past elements simultaneously, providing the benefits of bidirectional RNNs without doubling the computational load. Moreover, all this processing occurs in parallel, making both training and inference significantly faster.
- **Positional Embeddings:** These embeddings encode the position of each token in a sentence, eliminating the need for recurrence and enabling the model to capture positional information.

The ability to process sentences in their entirety prevents transformers from losing past information, as they do not depend on past hidden states to capture dependencies. Multi-

head attention and positional embeddings further enhance the model's ability to understand relationships between words.

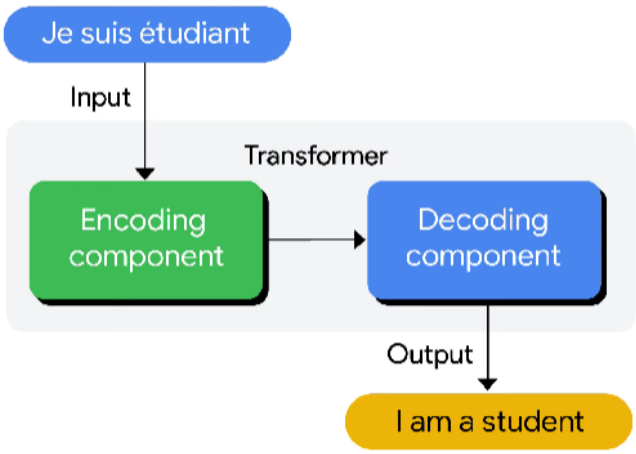
5.2 Architecture in details

We will now introduce the Transformer architecture and the Bidirectional Encoder Representations from Transformers (BERT) model.

5.2.1 Overview

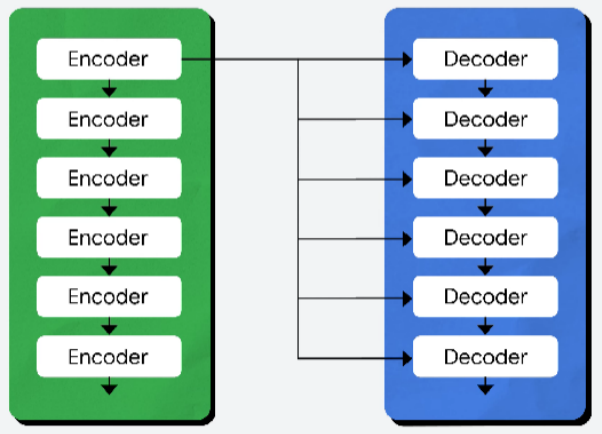
Originally devised for sequence transduction or neural machine translation, transformers excel in converting input sequences into output sequences. It is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution. The main core characteristic of the Transformers architecture is that they maintain the encoder-decoder model.
If we start considering a Transformer for language translation as a simple black box, it would take a sentence in one language, French for instance, as an input and output its translation in English.
If we dive a little bit, we observe that this black box is composed of two main parts:

- The encoder takes in our input and outputs a matrix representation of that input. For instance, the French sentence "Je suis étudiant"
- The decoder takes in that encoded representation and iteratively generates an output. In our example, the translated sentence "I am a student"



However, both the encoder and the decoder are actually a stack with multiple layers (same number for each). All encoders present the same structure, and the input gets into

each of them and is passed to the next one. All decoders present the same structure as well and get the input from the last encoder and the previous decoder.
The original architecture consisted of 6 encoders and 6 decoders, but this is a hyper parameter and we can replicate as many layers as we want. So let's assume N layers of each.



So now that we have a generic idea of the overall Transformer architecture, let's focus on both Encoders and Decoders to understand better their working flow.

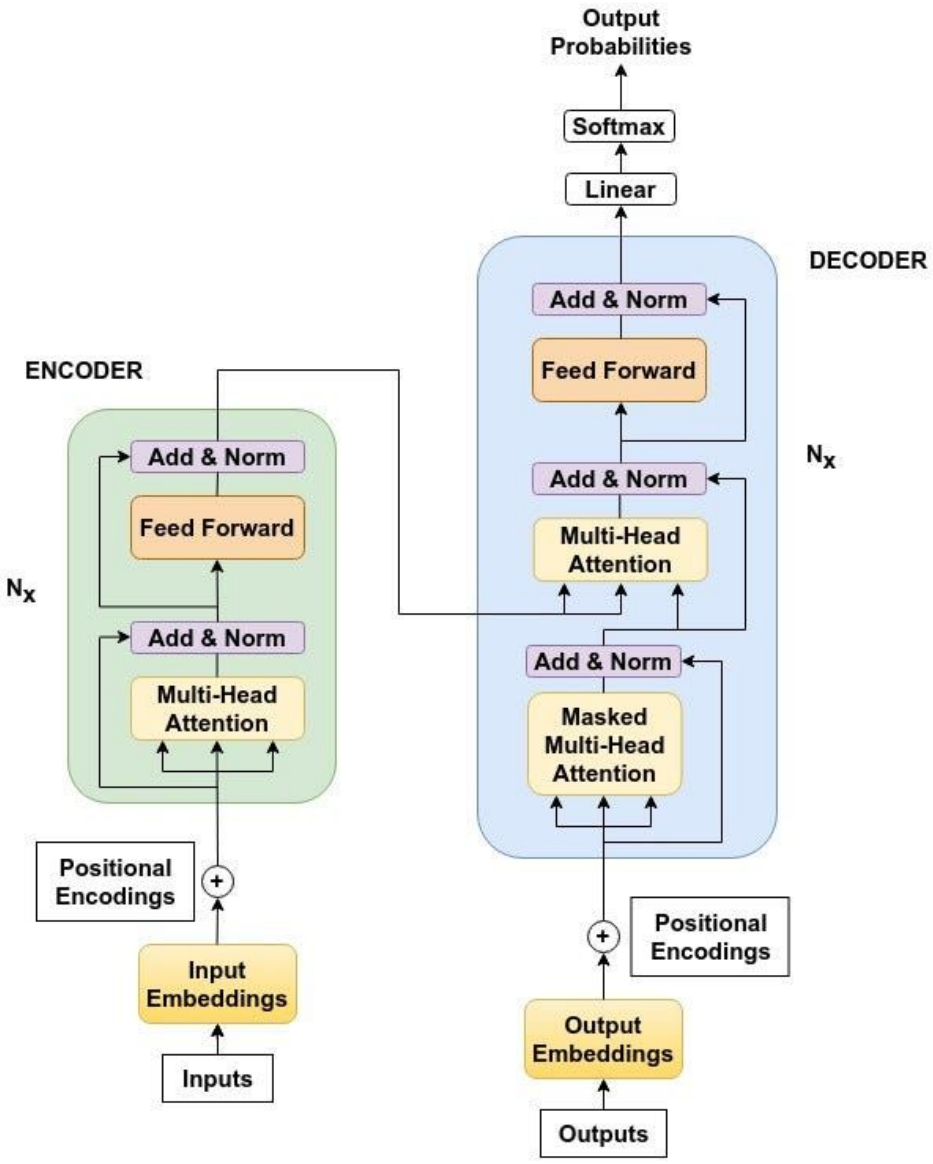


Figure 6: Architecture of the Transformer.

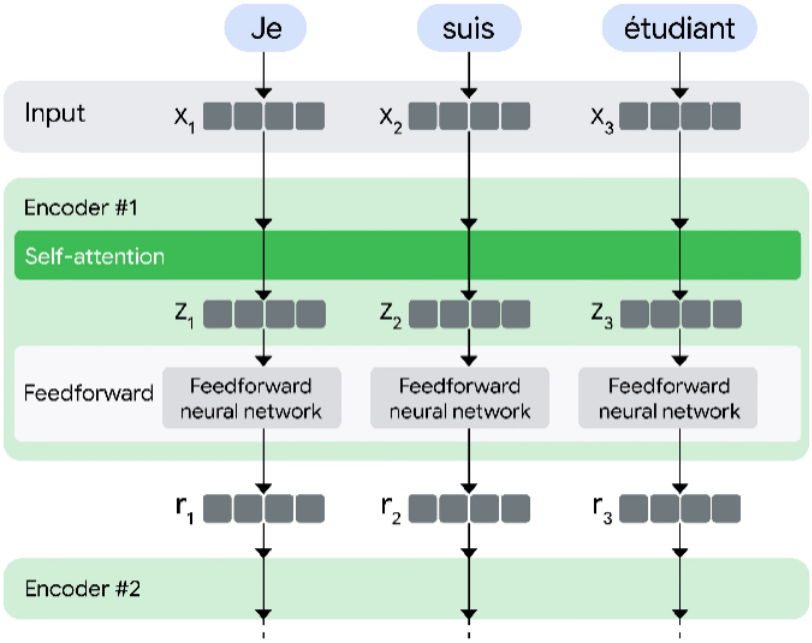
5.2.2 The Encoder WorkFlow

The encoder is a fundamental component of the Transformer architecture. The primary function of the encoder is to transform the input tokens into contextualized representations. Unlike earlier models that processed tokens independently, the Transformer encoder captures the context of each token with respect to the entire sequence.
Its structure composition consists as follows:

- **Input Embeddings:**

  The embedding only happens in the bottom-most encoder. The encoder begins by converting input tokens - words or subwords - into vectors using embedding layers. These embeddings capture the semantic meaning of the tokens and convert them

into numerical vectors. All the encoders receive a list of vectors, each of size 512 (fixed-sized). In the bottom encoder, that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below them.



- **Positional Encoding:**

  Since Transformers do not have a recurrence mechanism like RNNs, they use positional encodings added to the input embeddings to provide information about the position of each token in the sequence. This allows them to understand the position of each word within the sentence.

  To do so, the researchers suggested employing a combination of various sine and cosine functions to create positional vectors, enabling the use of this positional encoder for sentences of any length.

  In this approach, each dimension is represented by unique frequencies and offsets of the wave, with the values ranging from -1 to 1, effectively representing each position.

- **Stack of Encoder Layers:**

  The Transformer encoder consists of a stack of identical layers (6 in the original Transformer model).

  The encoder layer serves to transform all input sequences into a continuous, abstract representation that encapsulates the learned information from the entire sequence. This layer comprises two sub-modules:

A multi-headed attention mechanism. A fully connected network. Additionally, it incorporates residual connections around each sublayer, which are then followed by layer normalization.

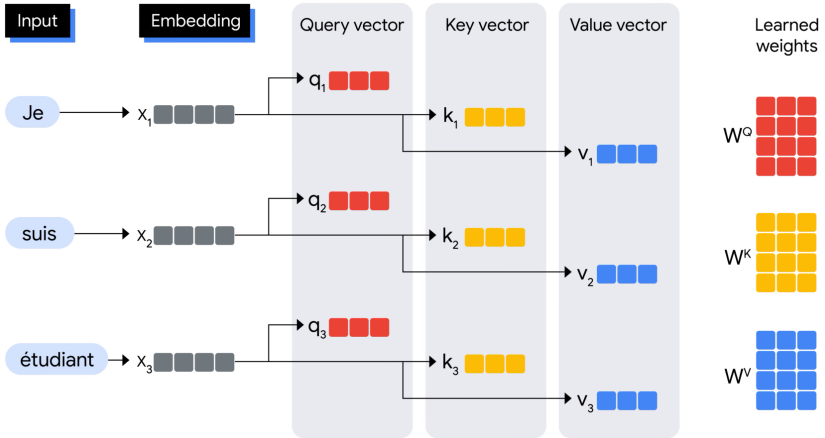
- **Multi-Headed Self-Attention Mechanism:**

  In the encoder, the multi-headed attention utilizes a specialized attention mechanism known as self-attention. This approach enables the models to relate each word in the input with other words. For instance, in a given example, the model might learn to connect the word "am" with "I".

  This mechanism allows the encoder to focus on different parts of the input sequence as it processes each token. It computes attention scores based on several factors.

  - **A query:** it is a vector that represents a specific word or token from the input sequence in the attention mechanism. - textbfA key: it is also a vector in the attention mechanism, corresponding to each word or token in the input sequence. - textbfA value: each value is associated with a key and is used to construct the output of the attention layer. When a query and a key match well, which basically means that they have a high attention score, the corresponding value is emphasized in the output.

  This first Self-Attention module enables the model to capture contextual information from the entire sequence. Instead of performing a single attention function, queries, keys and values are linearly projected h times. On each of these projected versions of queries, keys and values the attention mechanism is performed in parallel, yielding h-dimensional output values.



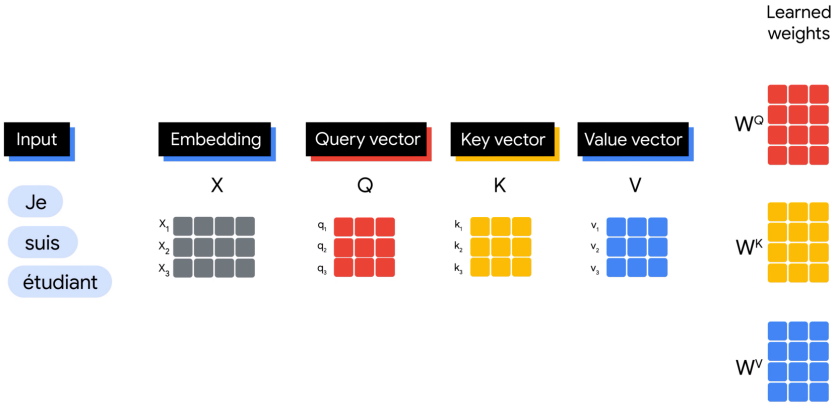
Matrix Multiplication (MatMul) - Dot Product of Query and Key

Once the query, key, and value vectors are passed through a linear layer, a dot product matrix multiplication is performed between the queries and keys, resulting

in the creation of a score matrix.

The score matrix establishes the degree of emphasis each word should place on other words. Therefore, each word is assigned a score in relation to other words within the same time step. A higher score indicates greater focus.

This process effectively maps the queries to their corresponding keys.



Reducing the Magnitude of attention scores

The scores are then scaled down by dividing them by the square root of the dimension of the query and key vectors. This step is implemented to ensure more stable gradients, as the multiplication of values can lead to excessively large effects.

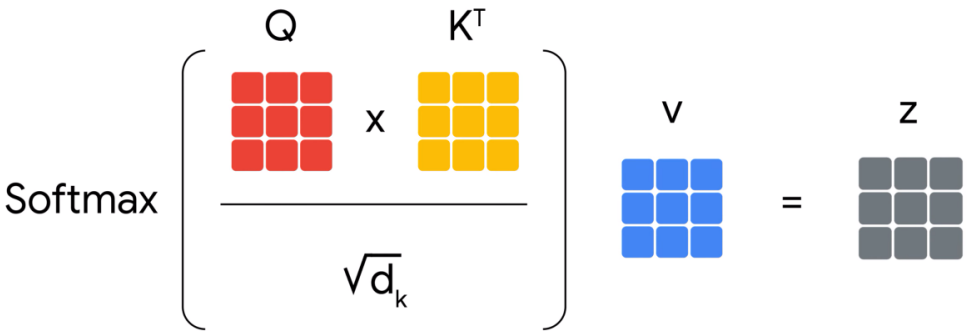
Applying Softmax to the Adjusted Scores

Subsequently, a softmax function is applied to the adjusted scores to obtain the attention weights. This results in probability values ranging from 0 to 1. The softmax function emphasizes higher scores while diminishing lower scores, thereby enhancing the model's ability to effectively determine which words should receive more attention.

Combining Softmax Results with the Value Vector

The following step of the attention mechanism is that weights derived from the softmax function are multiplied by the value vector, resulting in an output vector.

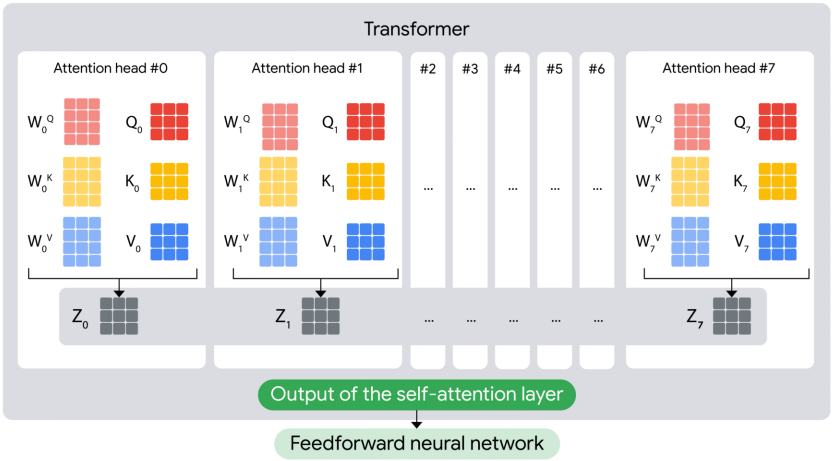
In this process, only the words that present high softmax scores are preserved. Finally, this output vector is fed into a linear layer for further processing.



So, we finally get the output of the Attention mechanism. Then, you might be wondering why it's called Multi-Head Attention?

Remember that before all the process starts, we break our queries, keys and values h times. This process, known as self-attention, happens separately in each of these smaller stages or 'heads'. Each head works independently, conjuring up an output vector.

This ensemble passes through a final linear layer, much like a filter that fine-tunes their collective performance. The beauty here lies in the diversity of learning across each head, enriching the encoder model with a robust and multifaceted understanding.



- **Normalization and Residual Connections**

  Each sub-layer in an encoder layer is followed by a normalization step. Also, each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem, allowing deeper models. This process will be repeated after the Feed-Forward Neural Network too.

- **Feed-Forward Neural Network**

  The journey of the normalized residual output continues as it navigates through a pointwise feed-forward network, a crucial phase for additional refinement.

  Picture this network as a duo of linear layers, with a ReLU activation nestled in between them, acting as a bridge. Once processed, the output embarks on a familiar path: it loops back and merges with the input of the pointwise feed-forward network.

  This reunion is followed by another round of normalization, ensuring everything is well-adjusted and in sync for the next steps.

- **Output of the Encoder**

  The output of the final encoder layer is a set of vectors, each representing the input sequence with a rich contextual understanding. This output is then used as the input for the decoder in a Transformer model.

  This careful encoding paves the way for the decoder, guiding it to pay attention to the right words in the input when it's time to decode.

  Think of it like building a tower, where you can stack up N encoder layers. Each layer in this stack gets a chance to explore and learn different facets of attention, much like layers of knowledge. This not only diversifies the understanding but could significantly amplify the predictive capabilities of the transformer network.

- **To sum it up here are the main steps of the encoder**

  1. Inpute natural language sentences

  2. Embed each word

  3. Perform multi-headed attention and multiply the embedded words with the respective weight matrices

  4. Calculate the attention using the resulting QKV matrices

  5. Concatenate the matrices to produce the output matrix, which is the same dimension as the final matrix

5.2.3 The Decoder WorkFlow

The decoder's role centers on crafting text sequences. Mirroring the encoder, the decoder is equipped with a similar set of sub-layers. It boasts two multi-headed attention layers, a pointwise feed-forward layer, and incorporates both residual connections and layer normalization after each sub-layer.

These components function in a way akin to the encoder's layers, yet with a twist: each multi-headed attention layer in the decoder has its unique mission.
The final of the decoder's process involves a linear layer, serving as a classifier, topped off with a softmax function to calculate the probabilities of different words.
The Transformer decoder has a structure specifically designed to generate this output by decoding the encoded information step by step.
It is important to notice that the decoder operates in an autoregressive manner, kick-starting its process with a start token. It cleverly uses a list of previously generated outputs as its inputs, in tandem with the outputs from the encoder that are rich with attention information from the initial input.
This sequential dance of decoding continues until the decoder reaches a pivotal moment: the generation of a token that signals the end of its output creation.

- **Output Embeddings:**

  At the decoder's starting line, the process mirrors that of the encoder. Here, the input first passes through an embedding layer

- **Positional Encoding:**

  Following the embedding, again just like the decoder, the input passes by the positional encoding layer. This sequence is designed to produce positional embeddings.

  These positional embeddings are then channeled into the first multi-head attention layer of the decoder, where the attention scores specific to the decoder's input are meticulously computed.

- **Stack of Decoder Layers:**

  The decoder consists of a stack of identical layers (6 in the original Transformer model). Each layer has three main sub-components:

  Masked Self-Attention Mechanism:

  This is similar to the self-attention mechanism in the encoder but with a crucial difference: it prevents positions from attending to subsequent positions, which means that each word in the sequence isn't influenced by future tokens.

  For instance, when the attention scores for the word "am" are being computed, it's important that "am" doesn't get a peek at "I", which is a subsequent word in the sequence.

  This masking ensures that the predictions for a particular position can only depend on known outputs at positions before it.

Encoder-Decoder Multi-Head Attention or Cross Attention:

In the second multi-headed attention layer of the decoder, we see a unique interplay between the encoder and decoder's components. Here, the outputs from the encoder take on the roles of both queries and keys, while the outputs from the first multi-headed attention layer of the decoder serve as values.

This setup effectively aligns the encoder's input with the decoder's, empowering the decoder to identify and emphasize the most relevant parts of the encoder's input.

Following this, the output from this second layer of multi-headed attention is then refined through a pointwise feedforward layer, enhancing the processing further.

In this sub-layer, the queries come from the previous decoder layer, and the keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence, effectively integrating information from the encoder with the information in the decoder.

Feed-Forward Neural Network:

Similar to the encoder, each decoder layer includes a fully connected feed-forward network, applied to each position separately and identically.

- **Linear Classifier and Softmax for Generating Output Probabilities:**

  The journey of data through the transformer model culminates in its passage through a final linear layer, which functions as a classifier.

  The size of this classifier corresponds to the total number of classes involved (number of words contained in the vocabulary). For instance, in a scenario with 1000 distinct classes representing 1000 different words, the classifier's output will be an array with 1000 elements.

  This output is then introduced to a softmax layer, which transforms it into a range of probability scores, each lying between 0 and 1. The highest of these probability scores is key,its corresponding index directly points to the word that the model predicts as the next in the sequence.

  Each sub-layer (masked self-attention, encoder-decoder attention, feed-forward network) is followed by a normalization step, and each also includes a residual connection around it.

  The final layer's output is transformed into a predicted sequence, typically through a linear layer followed by a softmax to generate probabilities over the vocabulary.

  The decoder, in its operational flow, incorporates the freshly generated output into its growing list of inputs, and then proceeds with the decoding process. This cycle

repeats until the model predicts a specific token, signaling completion.

The token predicted with the highest probability is assigned as the concluding class, often represented by the end token.

Again remember that the decoder isn't limited to a single layer. It can be structured with N layers, each one building upon the input received from the encoder and its preceding layers. This layered architecture allows the model to diversify its focus and extract varying attention patterns across its attention heads.

Such a multi-layered approach can significantly enhance the model's ability to predict, as it develops a more nuanced understanding of different attention combinations.

5.3 Results on the NLP for Financial Sentiment Analysis

Google's 2018 release of BERT, an open-source natural language processing framework, revolutionized NLP with its unique bidirectional training, which enables the model to have more context-informed predictions about what the next word should be.
By understanding context from all sides of a word, BERT outperformed previous models in tasks like question-answering and understanding ambiguous language. Its core uses Transformers, connecting each output and input element dynamically.
BERT, pre-trained on Wikipedia, excelled in various NLP tasks, prompting Google to integrate it into its search engine for more natural queries. This innovation sparked a race to develop advanced language models and significantly advanced the field's ability to handle complex language understanding.
Thus, to perform the NLP task on the Financial Sentiment Analysis dataset I decided to use the BERT decoder and to fine-tune it to this specific task. To do so, I first used a function to preprocess the data:

```
# Define stopwords
en_stopwords = set(stopwords.words("english"))

# Define preprocessing function
def preprocessing(sentence):
    sentence = sentence.lower()  # Remove caps
    sentence = re.sub(r"[^a-z\s]", "", sentence)  # Remove everything that is not a letter or a space
    sentence = word_tokenize(sentence)  # Tokenize
    sentence = [word for word in sentence if word not in en_stopwords]  # Remove stopwords
    lemmatizer = WordNetLemmatizer()
    sentence = [lemmatizer.lemmatize(word) for word in sentence]  # Lemmatize
    return " ".join(sentence)

# Preprocess the sentences for the training and validation sets
df["sentence_preprocessed"] = df["sentence"].apply(preprocessing)

print(df.head())
```

Yielding the following change:

```
                                            sentence  label  \
0  According to Gran , the company has no plans t...      1
1  For the last quarter of 2010 , Componenta 's n...      2
2  In the third quarter of 2010 , net sales incre...      2
3  Operating profit rose to EUR 13.1 mn from EUR ...      2
4  Operating profit totalled EUR 21.1 mn , up fro...      2

                              sentence_preprocessed
0  according gran company plan move production ru...
1  last quarter componenta net sale doubled eurm ...
2  third quarter net sale increased eur mn operat...
3  operating profit rose eur mn eur mn correspond...
4  operating profit totalled eur mn eur mn repres...
```

Then I divided the processed data into training set and validation set and trained my fine-tuned my BERT model on the processed data. I kept some non-processed data for the test set. And I got the following accuracies:

```
Training accuracy: 0.9412983425414365
Validation accuracy: 0.8732782369146006
Map: 100%|██████████| 453/453 [00:00<00:00, 841.32 examples/s]
Accuracy on the test set: 0.8432671081677704
{'eval_loss': 0.43293240666389465, 'eval_accuracy': 0.8432671081677704, 'eval_runtime': 141.7742, 'eval_samples_per_second': 3.195,
'eval_steps_per_second': 0.056, 'epoch': 3.0}
```

- Accuracy on the training set of 94%
- Accuracy on the validation set of 87%
- Accuracy on the testing set of 84%

5.4 Limitations of Transformers

However, transformers can be computationally expensive, indeed the self-attention mechanism results in processing that scales with $O(N^2)$, making transformers more costly to apply to long sequences compared to RNNs. This remains one area where RNNs hold an advantage over transformers. [1]:

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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

Figure 7: Transformers Complexity.

Furthermore, they require a large amount of data for training effectively.

5.5 Real-World Applications

Transformers excel in machine translation, document summarization, sentiment analysis, and natural language understanding, owing to their parallelism and attention mechanisms. In conclusion, Transformers have revolutionized the field of artificial intelligence and NLP.

These models surpass traditional RNNs by effectively managing sequential data with their unique self-attention mechanism. Their capability to handle long sequences more efficiently and parallelize data processing significantly accelerates training.

6 Conclusion

Selecting the right model among Naïve Bayes, RNNs, LSTMs, and Transformers depends on the specific task and dataset. For tasks with short-term dependencies, RNNs may suffice. When long-range dependencies are crucial, LSTMs can be a better choice. For large-scale tasks requiring parallelism and capturing complex relationships, Transformers often shine.

	Naïve Bayes	RNN
Year	1960s	1980s
Accuracy	70%	80%
Pros	Simple, Fast	Captures sequence data
Cons	Assumes independence, less accurate	Vanishing gradient problem

	LSTM	Transformers
Year	1997	2017
Accuracy	81%	84%
Pros	Handles long-term dependencies	High accuracy, parallel processing
Cons	Computationally intensive	Requires large datasets, complex

Table 1: Comparison of Naïve Bayes, RNN, LSTM, and Transformers.

As expected, the simplicity of the Naïve Bayes algorithm is a double-edged sword. It is very fast to implement and compile, and surprisingly, the results are good. However, more complex algorithms such as LSTM and Transformers achieve higher accuracy, with strong performances of 80% and more. These advanced models, while more computationally intensive, are better suited for capturing complex relationships and dependencies in the data, making them more effective for tasks requiring high precision and scalability.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention is all you need.* In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), pages 6000-6010, 2017.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning representations by back-propagating errors.* Nature, 323(6088):533-536, 1986.

[3] S. Hochreiter and J. Schmidhuber. *Long short-term memory.* Neural Computation, 9(8):1735-1780, 1997.

[4] A. McCallum, K. Nigam. *A Comparison of Event Models for Naive Bayes Text Classification.* In AAAI-98 Workshop on Learning for Text Categorization, 1998.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning.* MIT Press, 2016.

[6] `https://en.wikipedia.org/wiki/Long_short-term_memory`

[7] `https://larevueia.fr/quest-ce-quun-reseau-lstm/`