# Film Review Summary Project

---

## Introduction

The objective of this project is to analyze film reviews and provide predictions on whether the film is good or not. The project primarily aims to gain a better understanding of natural language processing techniques and RNN and Transformer models.

## Reporting

The report is divided into several sections:

- Data Collection and Processing
- Model Architecture and Training
- Experimental Results
- Conclusion
- Future Directions


### Author

Tran Le Minh


### Date

27 May 2023

# Table

Film Review Summary Project Report

**Python Library Choices ..... 3**

Python Library: ..... 3

**Dataset and Preprocessing Data ..... 3**

IMDb Film Review: ..... 3

Amazon's Movie&TV Review: ..... 4

Conclusion: ..... 5

**Approaches to Model Building ..... 5**

Data Processing: ..... 5

Word2vec: ..... 6

BERT Embedding: ..... 7

Dense Neural Network (DNN): ..... 7

Recurrent Neural Network (RNN): ..... 7

Transformer Neural Network: ..... 9

Model Architecture: ..... 10

Embedding Input Layer: ..... 10

Multi-head Attetion and Feedforward Layer: ..... 11

Encoder Block: ..... 12

**Training and Result ..... 13**

Training: ..... 13

Result: ..... 14

**Conclusion ..... 16**

# Python Library Choices

## Python Library:

- Keras/Tensorflow: I aimed for code simplicity when building the model, and thus chose Keras over Pytorch.
- Matplotlib, Numpy, Pandas, Scipy: These are libraries that are commonly used in data preprocessing or analysis.
- Gensim: This library includes the Word2vec model, primarily used for word or text vectorization.
- NLTK: This is a crucial library as it provides essential tools for basic NLP.
- Transformer: This library offers advanced models for various NLP tasks.

# Dataset and Preprocessing Data

Given that the primary focus of this project is to generate summaries of film reviews, categorizing them as either positive or negative using DNN and RNN-based models, or on a scale of 1-5 utilizing a Transformer-based model, the datasets employed for training and evaluation are exclusively limited to the IMDb Film Review and Amazon's Movie&TV Review dataset.

## IMDb Film Review:

The IMDb film review dataset consists of 50,000 review texts, evenly divided between positive and negative sentiments. The dataset is in CSV format and follows a simple format, as follows:

- **Review:** Text review that was crawed from IMDb web
- **Sentiment:** is either Positive or Negative

When it comes to NLP tasks, there are several necessary steps to ensure that the text is thoroughly cleaned to avoid any noisy instances. Regarding this specific dataset, the initial stage involves handling special tags extracted from web crawling, such as "<br>" and "<html>". Additionally, punctuation marks and numerical characters are eliminated, directing focus solely towards the textual content. Subsequently, the text is transformed to lowercase to mitigate excessive word variations, while non-essential stop words, lacking significant semantic influence, are removed. However, upon performing data cleaning and analyzing the distribution of frequent words, it became evident that certain terms like "film," "movie," "really," and "one" appeared excessively in both positive and negative instances. Consequently, these terms were also discarded. Lastly, words with fewer than two characters are

filtered out due to their limited impact on sentence semantics. Following this, words undergo lemmatization to revert them to their base forms. As an illustrative example, "went" $\backslash rightarrow \backslash$ "go", "running" $\backslash rightarrow \backslash$ "run" and "happiest" $\backslash rightarrow \backslash$ "happy."

Due to limited computer resources, loading all 460,000 images at once is not feasible. To overcome this challenge, I utilized the k-fold cross-validation technique to train the model. This approach was appropriate given that the dataset is comprehensive and each partitioned subset is similar to one another. This consistency ensured that the model was stable across all the folds. That how load_image_data function work

## Amazon's Movie&TV Review:

The Amazon product dataset is a vast collection of data containing reviews and ratings for various products sourced from the Amazon e-commerce platform. This dataset encompasses a wide range of categories, spanning from books to clothing and Movie&TV. However, for the purpose of this project, I specifically focus on a subset known as Amazon's Movie&TV Review 5-score – contains over 1 000 000 reviews. This subset is curated to include reviews from users who have contributed over five reviews, ensuring a higher level of reliability and relevance. Each review's information is encapsulated in a .json file, adhering to the following format:


```
{
	"reviewerID": "A2SUAM1J3GNN3B",
	"asin": "0000013714",
	"reviewerName": "J. McDonald",
	"helpful": [2, 3],
	"reviewText": "I bought this for my husband who plays the piano. He is having a wonderful time playing these old hymns. The music is at times hard to read because we think the book was published for singing from more than playing from. Great purchase though!",
	"overall": 5.0,
	"summary": "Heavenly Highway Hymns",
	"unixReviewTime": 1252800000,
	"reviewTime": "09 13, 2009"
}
```


Like the IMDb film review dataset, I only require reviewText and rating information from our dataset. Hence, I discard any extraneous data when loading the file. After loading the data, I noticed that the rating distribution was uneven, particularly at rating 4 and 5. To address this issue, I opted to retain all data with ratings of 1 and 2. Furthermore, I selectively kept reviews exceeding a word count of 200 but with ratings above 3. Upon plotting the data, it became evident that there was an ample number of

reviews, consisting of more than 200 words, associated with ratings of 4 and 5. However, the number of reviews for rating 3 fell short of the desired quantity. In order to rectify this disparity, I made the decision to incorporate an additional 35,000 reviews, each containing more than 100 words and carrying a rating of 3. This meticulous approach ensured that the dataset was evenly distributed across different rating categories. Also the dataset was clean through several step to avoid noise.

### Conclusion:

As the project progressed, I observed a positive correlation between the length of reviews and the credibility of their associated ratings. Furthermore, I realized that the dataset could be effectively cleaned by excluding ratings with fewer than 20 words. By doing so, the dataset becomes more reliable as it focuses on reviews that provide more substantial content and insights. This approach ensures that the analysis and models built upon the dataset are based on more meaningful and informative reviews, enhancing the overall quality of the project's outcomes.


*Figure 1: Word Cloud from IMDb dataset*

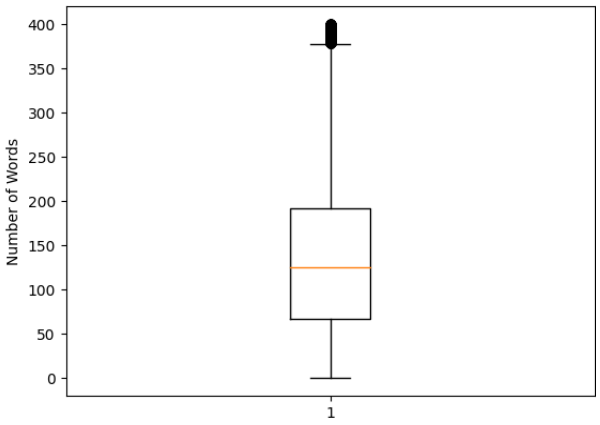


*Figure 2: Review Length Distribution*

---

# Approaches to Model Building

This section covers three models that I used for NLP classification: Dense Neural Network (DNN), Recurrent Neural Network (RNN) and Transformer Neural Network (RNN). But first let talk about how data processing work.

### Data Processing:

As we all know, computers can only understand and process numerical data. Therefore, to build an NLP model, we need to convert text-based language into numerical representations, which are known as

word embeddings. Word embeddings capture the semantic and contextual information of words, allowing NLP models to effectively understand and manipulate textual data. In my project there are 3 method of word embedding according to 3 models, but in summary there only 2 main methods was use. One approach involves leveraging the word embeddings generated by the Word2Vec model for processing purposes, while the other approach entails employing word embeddings that adhere to the precise definition as prescribed by the BERT model.

## Word2vec:

Word2Vec is a simple yet widely renowned model designed to generate word embedding representations. It operates on the principle that words appearing in similar contexts tend to have similar meanings. There are two primary architectures in Word2Vec: Continuous Bag of Words (CBOW) and Skip-gram. In my code both DNN and RNN model use CBOW architecture in Word2Vec to convert word to text before training (since the dataset is large enough).

In the CBOW architecture, the model predicts the current word based on the context words surrounding it. The context words are used as input features, and the central word is predicted as the output. This approach is beneficial when the focus is on smaller datasets and frequent words. The Skip-gram architecture functions in the opposite manner.

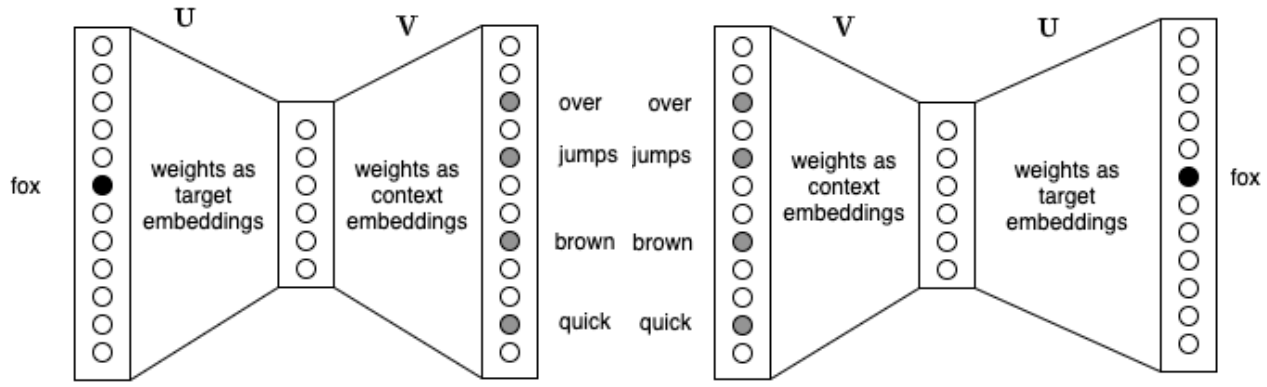


*Figure 3: Skip-Gram and CBOW Model*

The only distinct difference between DNN and RNN when applying this method is that DNN computes the sum of the entire text sequence and takes the average, whereas RNN treats each word as an individual embedding vector. This distinction arises because DNN models, being simpler in nature, may not capture the nuanced features present in longer text sequences. Therefore, by summing the embedding vectors and calculating the average, DNN can still effectively reflect the sentiment of the review. In contrast, RNN, particularly with LSTM layers, is more complex, capable of retaining long-term information, and recognizing intricate patterns in the text.

### BERT Embedding:

The BERT model embedding method operates by leveraging a word “bank” tokenizer to processing input data. BERT has an advantage over models like Word2Vec because while each word has a fixed representation under Word2Vec regardless of the context within which the word appears, BERT produces word representations that are dynamically informed by the words around them. There for more information was capture by this method.

Additionally, for each sequence, there is a vector attention mask generated. This attention mask serves as a binary tensor that highlights the positions of padded indices within the sequence. By incorporating the attention mask, the model is able to effectively ignore these padded tokens during training and inference. This is particularly important when dealing with named entities or unknown words that fall outside the scope of the central word "bank." By marking these tokens in the attention mask, the model can prioritize the meaningful information within the sequence and disregard irrelevant elements.

## Dense Neural Network (DNN):

As mentioned earlier, the approach of taking the average sum of word embeddings to obtain sequence embeddings can still effectively reflect the sentiment of reviews that exhibit clear positive or negative tones. Therefore, DNN models can still perform well in capturing the relevant features within sequences. By considering the collective information of the word embeddings, the DNN model can understand the overall sentiment expressed in the review and make accurate predictions. This approach proves to be a reliable method for fast and simple text classification.

## Recurrent Neural Network (RNN):

RNN is a type of neural network commonly used for processing sequential data such as time series or natural language. RNNs incorporate one or more gates to process information and retain previous data rather than completely forgetting it. There are various types of RNNs, including LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) and other types such as SimpleRNN, Bidirectional RNN, etc. For my project, I have chosen to utilize LSTM for its ability to capture long-term dependencies and handle sequential data effectively.

An LSTM cell consists of three components, also known as gates, which are:

- Forget Gate: This gate determines what information to discard from the previous cell state. It takes the previous hidden state and the current input as inputs and outputs a number between 0 and 1 for each element in the cell state, representing the importance of each element.

- Input Gate: The input gate decides which values to update in the cell state. It takes the previous hidden state and the current input as inputs and outputs a number between 0 and 1 for each element in the cell state, indicating the amount of update for each element.
- Output Gate: The output gate controls what information from the cell state will be used as the output. It takes the previous hidden state and the current input as inputs, along with the updated cell state, and outputs a filtered version of the cell state.

These gates in the LSTM cell allow it to selectively retain or forget information, update the cell state, and control the flow of information through the recurrent connections. This helps LSTM cells effectively handle long-term dependencies and capture important patterns in sequential data.

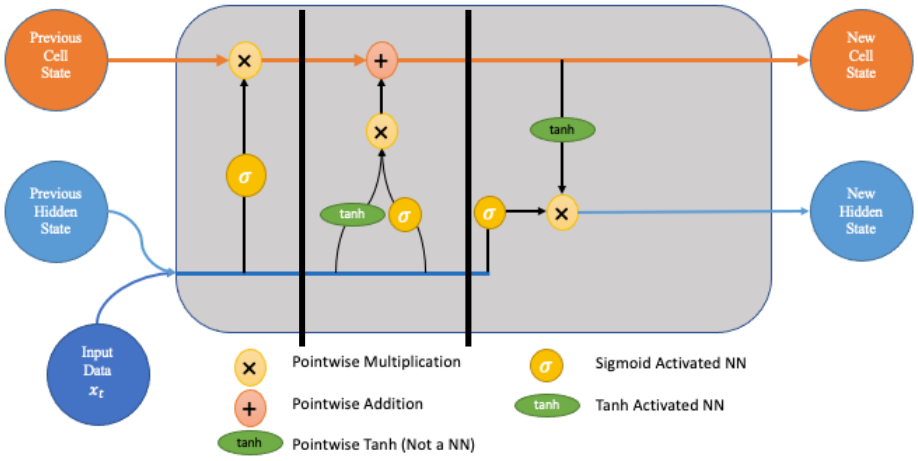


*Figure 4: LSTM Cell Diagram*

As previously mentioned, the intrinsic nature of LSTM eliminates the need for sequence embedding, as it can lead to potential loss of information. For instance, in the sentence "This film is not good," even if we retain the word "not" during text preprocessing, a DNN model might still mistakenly classify it as a positive review. In contrast, an LSTM model is capable of discerning that the combination of "not" and "good" signifies a negative sentiment rather than a positive one. It is worth noting that LSTM models can effectively handle multi-rating scenarios akin to Transformer-based models. However, due to limited computational resources, I refrained from utilizing the Amazon dataset for training purposes. This dataset contains considerable noise, necessitating a substantial amount of data to achieve satisfactory performance. Additionally, considering that each word would be embedded into a 75-dimensional vector, my computer would not be able to handle it efficiently.

I also added a masking layer to the model to allow it to distinguish and disregard the padded portions of the data during training. This was achieved using the `pad_sequence` function to obtain fixed-length data by padding the sequences.

## Transformer Neural Network:

While recurrent neural networks (RNNs) are undeniably powerful and effective, they do possess certain limitations in terms of capturing long-term features and efficiently handling parallel processing. Recognizing these constraints, the Transformer architecture was developed as a groundbreaking solution to overcome these limitations.

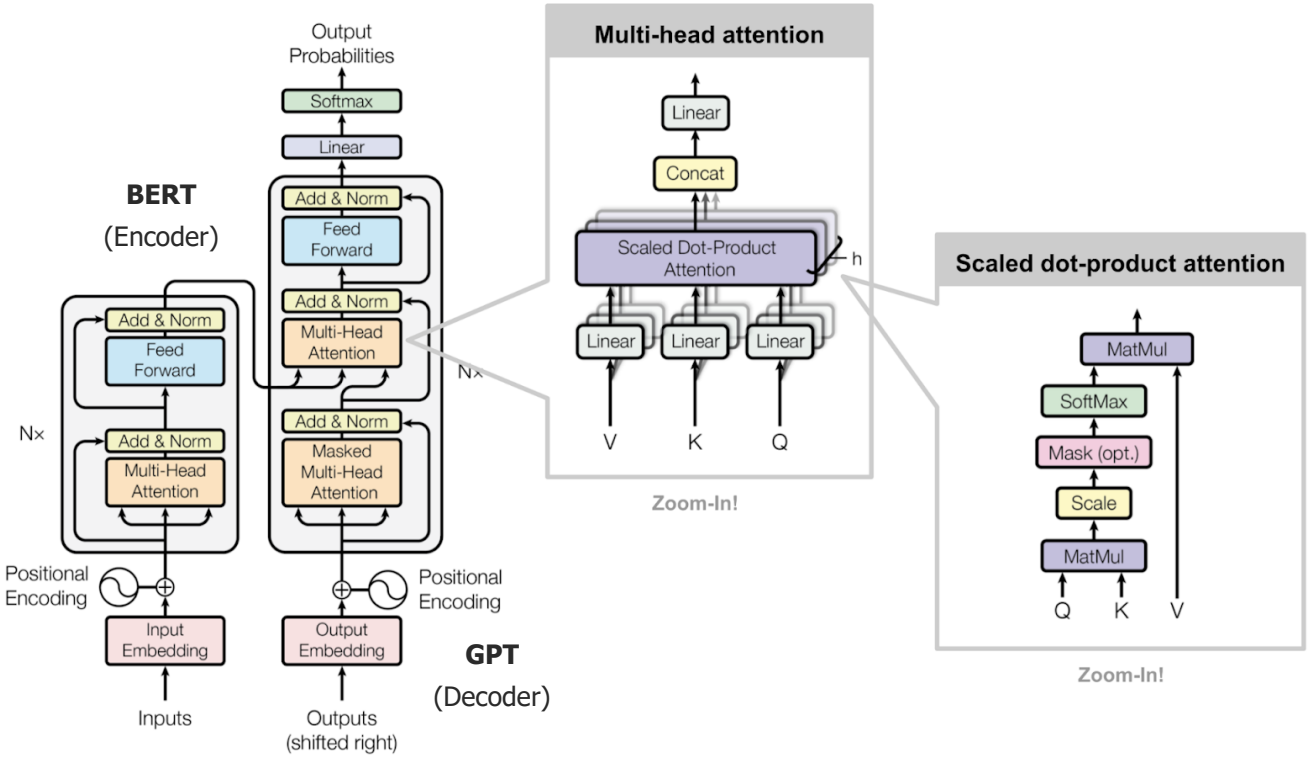


*Figure 5: Transformer Model Architecture*

In text classification tasks, there is often no need to generate text but rather to focus on understanding the content of a given text. Therefore, for text classification purposes, we can leverage the power of the encoder block in the Transformer architecture, also known as BERT. The encoder block is responsible for processing the input text and extracting meaningful representations, which can be used for classification purposes. BERT (Bidirectional Encoder Representations from Transformers) is a state-of-the-art language model that has revolutionized natural language processing tasks. It is a pre-trained model developed by Google, known for its ability to understand the context and meaning of words in a sentence.

*Attention is All You Need*

When working with LLM or any model related to NLP, you have probably already heard about that sentence. The multi-head attention layer is a crucial component in Transformer-based models, providing them with an edge over traditional models. In addition to parallel processing capabilities, this layer allows the model to simultaneously focus on different sections of the input, capturing diverse relationships and dependencies between words. Each attention head can learn to attend to distinct aspects of the input, offering multiple perspectives on the data. This ability enables the model to capture both detailed and high-level semantic information, resulting in more comprehensive representations. This enables the model to capture both fine-grained and high-level semantic information, leading to richer representations.

Moreover, in this project, I am constrained by limited computational resources, thus opting to utilize TinyBERT exclusively. TinyBERT is a highly compact variant, substantially much smaller than the original BERT model. Its reduced size grants it a significant performance advantage in term of speed, with a processing speed that is up to 10 times faster than that of the standard BERT with a slight degradation in performance in terms of accuracy or its ability to understand text to some extent. The seminal paper introducing TinyBERT explicitly states that it consists of two encoder blocks.

I also built my own rendition of BERT, based on the structure and functioning described in the BERT paper. However, my customized BERT implementation incorporates several enhancements pertaining to the utilization of masking. Specifically, this model seamlessly integrates masking right from the embedding phase, obviating the necessity to process it further within the MultiHeadAttention module. Consequently, we no longer require supplementary attention masks as input.

## Model Architecture:

In this section, I will elaborate on the functioning of the BERT model and provide insights into my own customized version of BERT. It introduced the concept of masked language modeling (MLM) and next sentence prediction (NSP) during pre-training to learn contextual representations of words. But in text classification task there will be some changes in the way the model work and train.

### Embedding Input Layer:

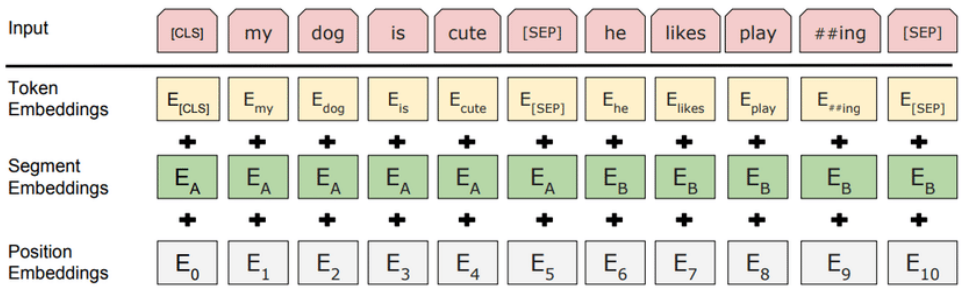


*Figure 6: BERT Word Embedding Layer*

Above is an explanation of how the BERT embedding layer operates. The input text is vectorized and then combined with segment embedding and positional embedding. Segment embedding is a component that assigns a unique representation to different segments or parts of the input text. Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. In transformer-based models, the formula for positional embedding is as follows:

$$\mathrm{P}(\mathrm{pos}, \mathrm{d}) = \sin\left(\frac{\mathrm{pos}}{\mathrm{n}^{2\mathrm{i}/\mathrm{d}}}\right) \text{ if i is even, else } \cos\left(\frac{\mathrm{pos}}{\mathrm{n}^{2\mathrm{i}/\mathrm{d}}}\right)$$

- $\mathrm{pos}$ is the position index in the sequence.
- $\mathrm{d}$ is the dimensionality of the positional embedding.
- $\mathrm{i}$ represents the index of the dimension in the positional embedding vector.
- $\mathrm{n}$ is user-defined scalar, set to 10000 by the paper.

However, in the specific task of text classification specifically sentiment analysis, the need to discern whether a sentence is a question or a different type of sentence is deemed unnecessary. As such, eliminating the segment embedding does not compromise the model's understanding capabilities. And I have also implemented mask processing within this layer. This step is essential to identify and distinguish the padded positions in the output sequence.

### Multi-head Attetion and Feedforward Layer:

Multi-head attention layer is a fundamental layer in transformer-based models that enables them to capture dependencies and relationships between words or tokens within a sequence. The multi-head attention mechanism operates by performing the above steps multiple times in parallel, using different learned projections for each attention head. This allows the model to capture different types of information and attention patterns It enhances the model's ability to attend to different parts of the input sequence simultaneously, allowing it to capture both local and global context. Therefor it surpass the RNN-based model ability in both term of speed and information capture.

Based on Figure 5, the multi-head attention consists of three main components:

- Linear Projection: The input sequence is transformed into three different linear projections: query, key, and value. These projections are obtained by applying separate learned linear transformations to the input sequence.
- Attention Scores: For each token in the input sequence, the attention mechanism calculates a similarity score with every other token. This is done by computing the dot product between the query and key vectors.

- Weighted Sum: The attention scores are then used to weight the corresponding value vectors. The weighted sum of the value vectors produces the output representation for each token. This step allows the model to focus more on the relevant tokens while disregarding irrelevant or less important ones.

The fomula or a Multi-head Attention is as follow:

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}\left(\text{Attention}\left(QW_Q^{(i)}, KW_K^{(i)}, VW_V^{(i)}\right) \text{ for } i \text{ in } \{1,2,\dots,h\}\right) \text{Wo}$$

Indeed, in the BERT model, this layer is also referred to as "Self-Attention" because all three inputs (query, key, and value) are derived from the same source sequence. Compared to traditional multi-head attention, which attends to different parts of the sequence independently, self-attention allows each position in the sequence to attend to all other positions. This means that each word or token can take into account the information and context from all other words in the sequence. Allow it to understand how each word or token relates to the entire input sequence and its surrounding. In the self-attention mechanism, the purpose of the mask is to prevent certain positions in the input sequence from attending to other positions. This is particularly relevant in tasks where the input sequence contains padding tokens or where the model needs to handle variable-length sequences. But in text classification task where I need the model to capture the most information possible. And through experimentation and observation of the attention_mask generated from the sequence tokenization process, it became apparent that the attention_mask functions similarly to the pad_sequence method. In this case, positions with actual tokens are marked as 1, while positions without tokens are marked as 0. Therefore, it can be concluded that there is no need for additional masking.

Explained detail about Multi-head Attention can be found in this link.

In BERT, a feedforward layer is applied after the multi-head attention layer in each transformer block. The feedforward layer serves as a point-wise fully connected neural network that applies non-linear transformations to the intermediate representations. Allow the model to model complex interactions between tokens and enhancing its performance in various NLP task. After each layer, there is an Add and Norm layer in BERT. The Add layer combines the output of the layer with its input, allowing the model to capture residual connections and facilitate information flow. The Norm layer then performs layer normalization, which normalizes the outputs across the hidden dimension, ensuring stable training and improved generalization capabilities.

### Encoder Block:

An encoder block consists of 1 multi-head attention layer and 1 feedforward layer. In my BERT model, there are a total of 4 encoder blocks, which enhance the model's text comprehension capabilities

compared to TinyBERT while remaining lighter in comparison to the original BERT. Finally because my model task is text classification so there will be three more Dense layer in order to get the result.

# Training and Result

## Training:

Each model will have its own complexity and strengths as well as weaknesses. The following table shows the Big O complexity per layer and sequential operations of a single layer, representing the model architecture name.

| Layer Type | Complexity per Layer | Sequential Operations |
|---|---|---|
| **Self-Attention** | $O(n^2 * d)$ | $O(1)$ |
| **Recurrent** | $O(n * d^2)$ | $O(n)$ |
| **Dense** | $O(n * d)$ | $O(1)$ |

*Table 1: Maximum Complexity per Layer and Minimum Sequential Operations for Different Types of Layer. n is the Sequence Length, d is the Dimension of the Input*

As you can see, when comparing the two types of models that possess the ability to comprehend information and relationships among tokens within a sequence, Transformer-based models exhibit a superior algorithmic processing speed in contrast to RNNs. Despite the fact that their algorithmic complexity is considerably higher, Transformers leverage the inherent capability to process an entire sequence simultaneously, as opposed to handling individual words as done by RNNs. Consequently, this characteristic yields a significantly faster processing speed.

As for the optimizer and metrics, all four types of models I use the same optimizer, which is Adam. However, for the metrics, I only employ a simple accuracy metric for DNN and RNN models since they only involve binary predictions. In the case of the two transformer models, I utilize the average weighted sum method to obtain the most accurate result.

| Model Name | $D_{model}$ | $N$ | $h$ | $D_k$ | $D_{ff}$ | Vocab Size |
|---|---|---|---|---|---|---|
| **TinyBERT** | 128 | 4 | 12 | 312 | 1200 | 30522 |
| **My BERT** | 400 | 4 | 4 | 64 | 320 | 30522 |

*Table 2: Model Architecture*

Due to limitations in computer resources, my model is much smaller than originally intended. TinyBERT has a significantly higher number of attention heads, hidden dimensions, and feed-forward dimensions compared to my model. Additionally, TinyBERT has been trained more extensively. Therefore, it is certain that TinyBERT can outperform my model when the text length is below 128 tokens. However, in the majority of review texts, the number of tokens often exceeds 128, and there is a significant amount of information scattered across the sequence. Consequently, it is certain that my model will outperform TinyBERT.

## Result:

| Model Name | Num Epochs | Val_loss | Val_accuracy/Val_star_mae |
|---|---|---|---|
| **DNN** | 43 | 0.4618 | 0.91 (Acc) |
| **RNN** | 40 (10)* | 0.342 | 0.97 (Acc) |
| **TinyBERT** | 40 (10)* | 2.555 | 0.6385 (MAE) |
| **My BERT** | 5 | 1.0211 | 0.6224 (MAE) |

*Table 3: Model Training Results*

The main reason for the difference in val_accuracy between the DNN and RNN models is that the DNN model lacks the ability to understand words with negation, such as "not". Figure 7, the DNN model is shown processing the data without removing the word "not," yet the resulting accuracy remains the same. This is a small weakness of the DNN model as it can be overlooked or disregarded due to such circumstances. However, overall, the DNN model still performs very well in other aspects.

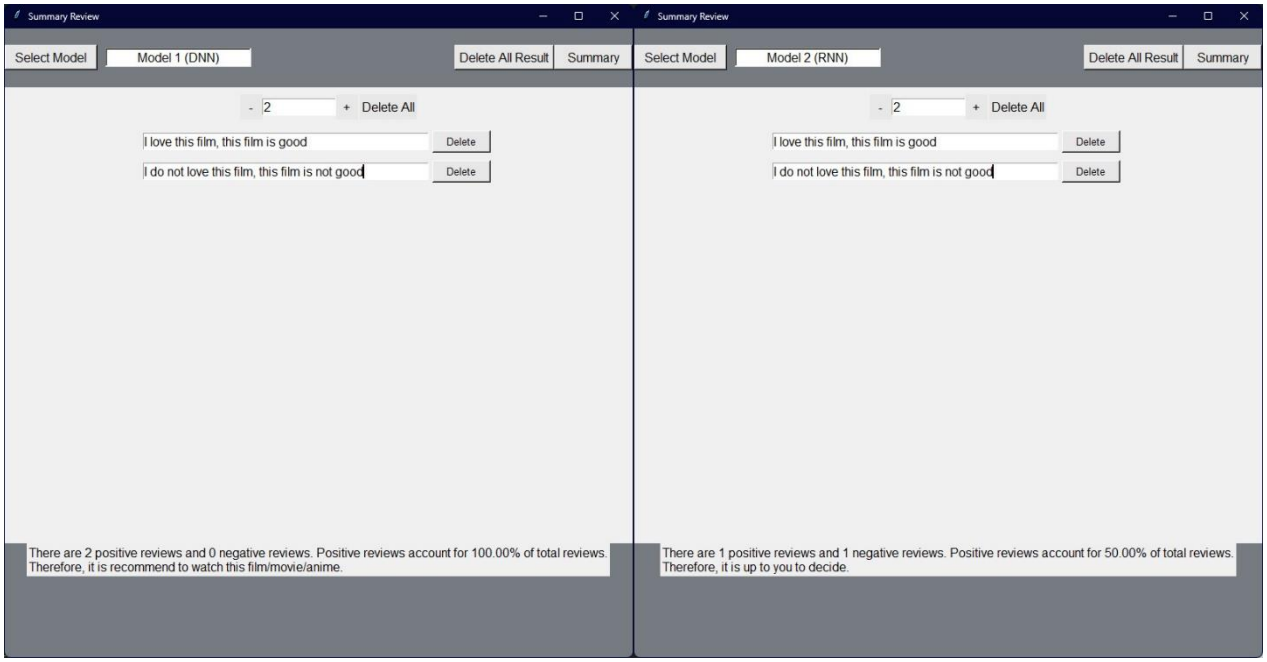


*Figure 7: Comparision of 2 Model*



**Using K-Folds technique to train the model, load the data randomly 4 times to train, each 10 epochs*

When considering the two BERT models, there is a noticeable difference in the loss values. As mentioned earlier, TinyBERT is limited in terms of sequence length, resulting in numerous predictions and a substantial margin of error if a prediction is incorrect. On the other hand, due to the slow training process and resource-intensive nature of my BERT model, I only trained it for 5 epochs, which is significantly fewer compared to the 40 epochs of TinyBERT. Consequently, there isn't a substantial difference in val_star_mae.

Here are the Review rating I randomly selected from Myanimelist.com:

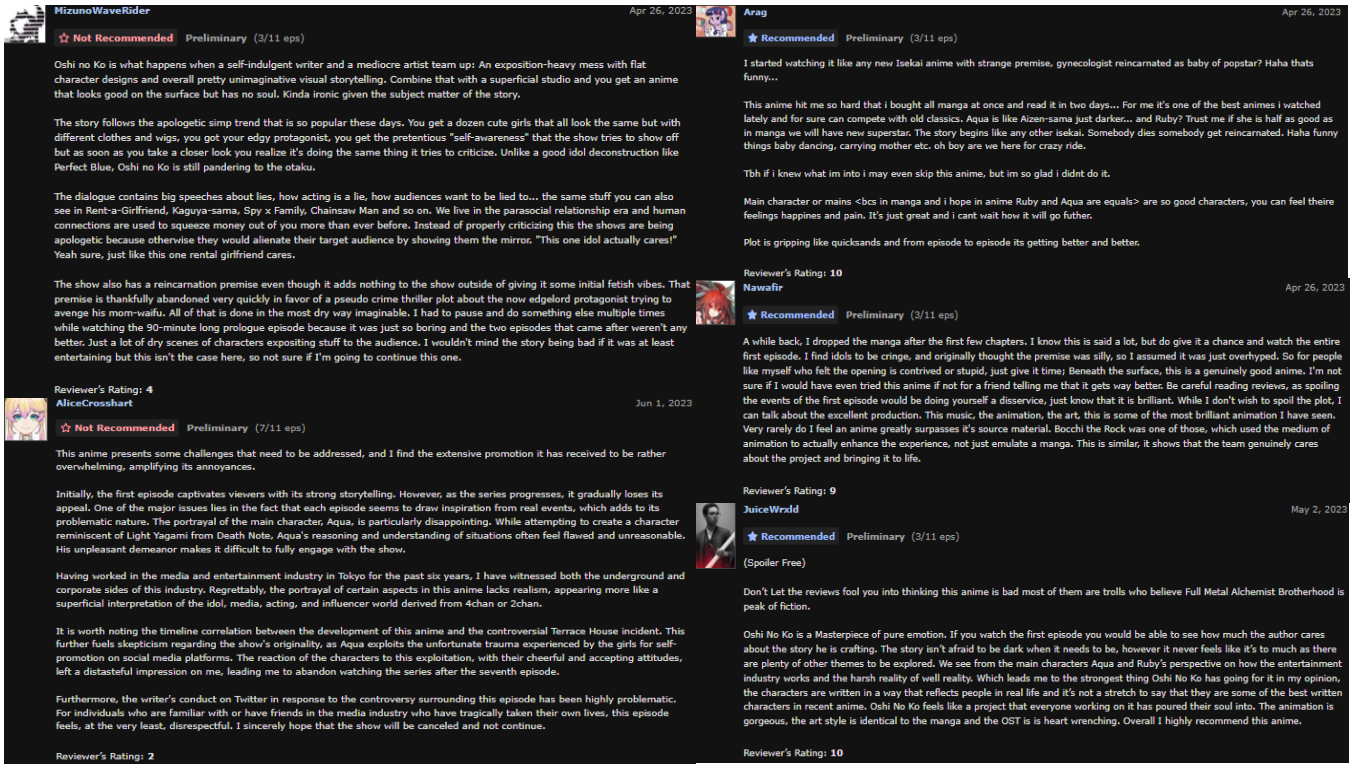


*Figure 8: Reviews*

After observation, I found that out of the five randomly selected reviews for the anime "Oishi No Ko," the average rating on a scale of 5 was 3.5 (obtained by summing the points and dividing by 5, then dividing by 2). Upon closer examination, I noticed that two negative reviews had a word count exceeding 128, and the beginning of these reviews contained some positive remarks about the movie. However, the later sections expressed dissatisfaction, leading me to believe that these were rather challenging tests for the TinyBERT model. As predicted, based on Figure 9 below, you can clearly see a significant difference in the predicted scores between the two models. TinyBERT produced scores that deviated considerably from the average of 3.5. This discrepancy is primarily due to the limited word

count capacity of TinyBERT. In contrast, my version of BERT, which has a maximum word count of 400, does not encounter difficulties in comprehending the entire content of the reviews.

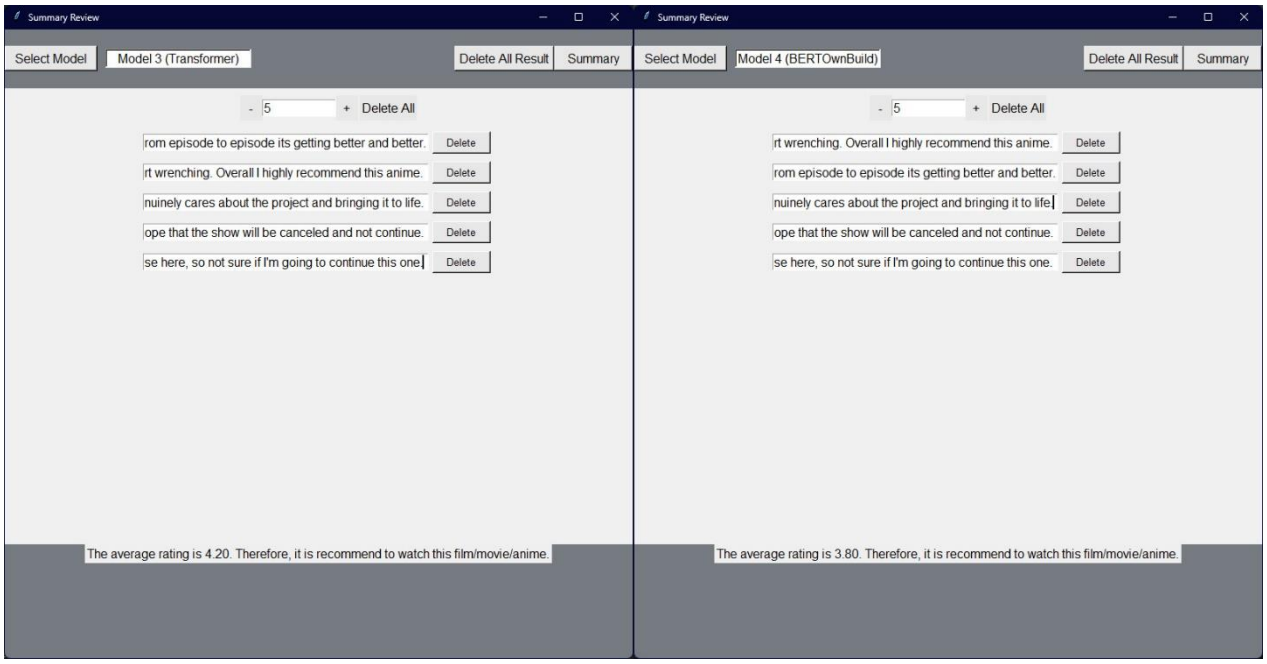


*Figure 8: Comparision of 2 Models*

# Conclusion

In conclusion, each model possesses its own strengths and weaknesses. The DNN model excels in its simplicity of operation, but its simplicity also leads to some errors being overlooked. The RNN model demonstrates the ability to understand word meanings and contextual information, but it suffers from slower processing due to its inability to operate in parallel and its potential for forgetting long-term memory. On the other hand, TinyBERT addresses all the limitations faced by the other two models, but it is constrained by a limited word count. As for my BERT model, it does not encounter limitations in word count but lacks proper training due to insufficient computer resources.