



# Vietnamese Sentiment Analysis Capstone Project

## Group 3

Phạm Văn Cường - 20194421 - [cuong.pv194421@sis.hust.edu.vn](mailto:cuong.pv194421@sis.hust.edu.vn)

Nguyễn Việt Hoàng - 20194434 - [hoang.nv194434@sis.hust.edu.vn](mailto:hoang.nv194434@sis.hust.edu.vn)

Hoàng Văn Khánh - 20194440 - [khanh.hv194440@sis.hust.edu.vn](mailto:khanh.hv194440@sis.hust.edu.vn)

Phan Đức Thắng - 20194452 - [thang.pd194452@sis.hust.edu.vn](mailto:thang.pd194452@sis.hust.edu.vn)

Phạm Việt Thành - 20194454 - [thanh.pv194454@sis.hust.edu.vn](mailto:thanh.pv194454@sis.hust.edu.vn)

# Table of Contents

<b>Introduction</b>	<b>3</b>
<b>Data Overview</b>	<b>3</b>
<b>Preprocess</b>	<b>5</b>
3.1. One-hot encoding	7
3.2. Word embedding	7
<b>Models</b>	<b>7</b>
4.1. RNN	7
4.2. Long Short Term Memory	9
4.3. Transformer	11
4.4. BERT	16
4.4.1 BERT Architecture	16
4.4.2 BERT Training Objectives	18
4.4.3 RoBERTa	18
4.4.4 PhoBERT	19
4.5. Ensemble Learning	19
<b>Experimental results</b>	<b>21</b>
5.1. RNN	21
5.2. LSTM	23
5.3. Transformer	26
5.4. BERT	30
5.5. Final Test Results	31
5.6. Comparing our result to the authors of the dataset	34
<b>Conclusion</b>	<b>35</b>
<b>Members' Contribution</b>	<b>35</b>
<b>References</b>	<b>35</b>

## **I. Introduction**

Sentiment analysis is a task of determining sentimental classes of a given sentence input. With the development of neural networks architectures, sentiment analysis systems have gained huge advances and outperforms traditional statistical models.

In this project, we will build sentiment analysis systems with a Vietnamese Dataset. The dataset is called UIT-VSFC, which is released by Ho Chi Minh City University of Technology. The data is built from Vietnamese students' feedback on various topics. Despite being released in 2018, there is not much research on this dataset.

For the given dataset, we perform experiments with several neural networks architectures for sentiment analysis. The chosen models are as follow:

- RNN (Recurrent Neural Networks).
- LSTM (Long-Short Term Memory Networks).
- Transformers.
- BERT (Bidirectional Encoder Representations from Transformers).

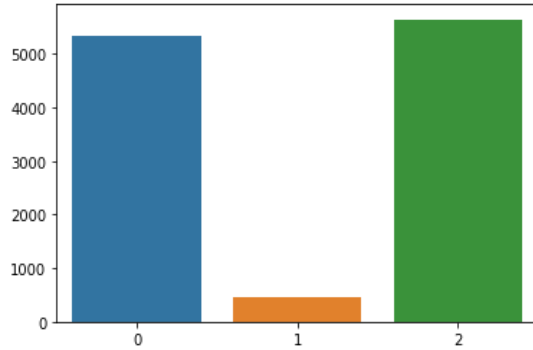
With each model, we tried various configurations to see if the current solutions can be improved. The rest of the report is organized as follows: In Section 2, we describe the overview of the chosen dataset , and Section 3 explains the preprocessing pipeline. Model architectures are discussed in Section 4, and lastly we give the experimental results in Section 5.

## **II. Data Overview**

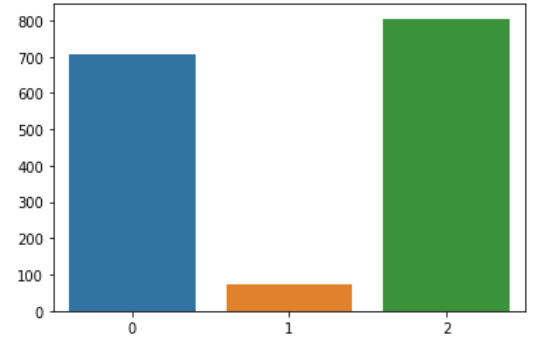
Our dataset has 3 classes (positive, neutral, negative) and is divided into 3 subsets:

- Train set: 11426 samples
- Dev set: 1583 samples
- Test set: 3166 samples

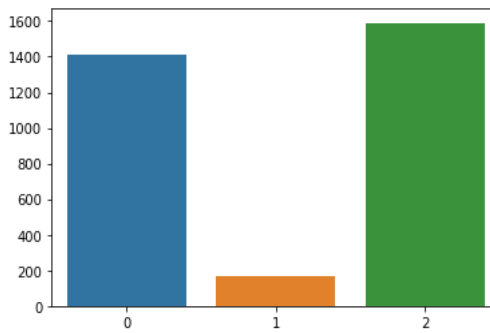
## 2.1. Imbalanced dataset



*Number of examples per classes (Train set)*



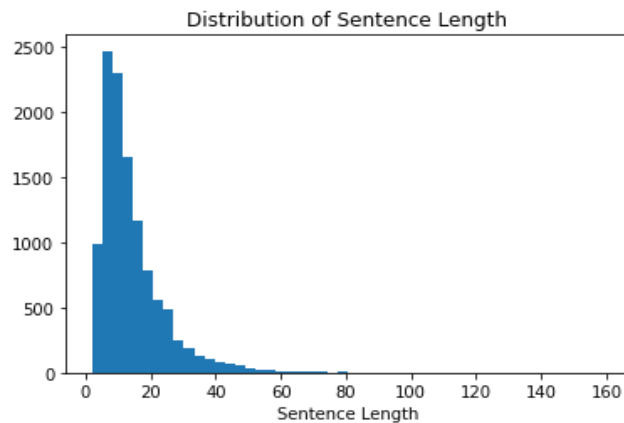
*Number of examples per classes (Dev set)*



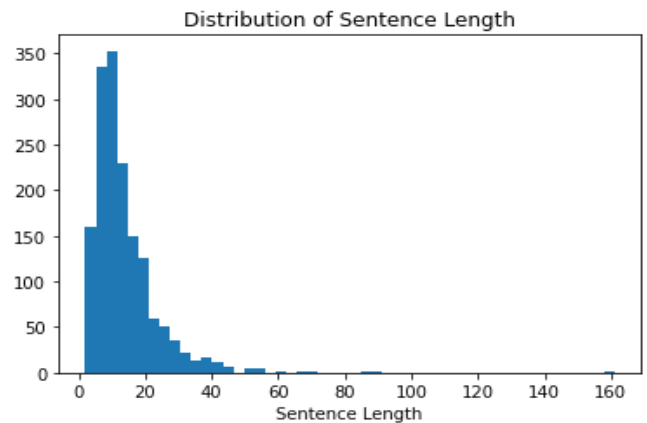
*Number of examples per classes (Test set)*

Our dataset for this problem is an imbalanced dataset. In the figures above, we can see that the number of samples in class 0 (negative feedback) and class 2 (positive feedback) is much bigger than the number of samples in class 1 (neutral).

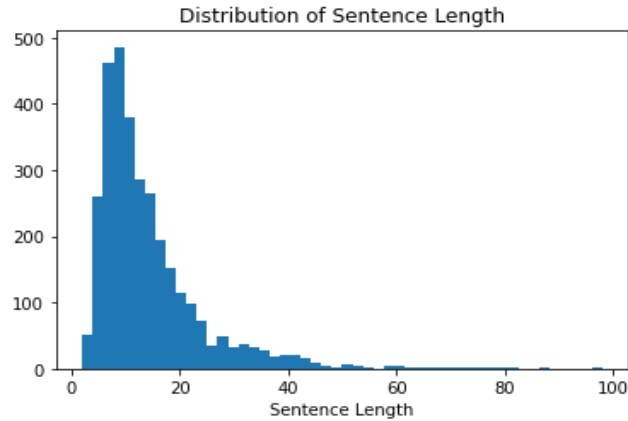
## 2.2. Distribution of sentence length



*Train set*



*Dev set*



*Test set*

As we can see from the histogram, most sentences in the dataset have around 10 to 40 words. This distribution will affect how much we pad the length of sentences when passing it to the model.

### III. Preprocess

Since the dataset is already cleaned by the authors, we only focus on data representation techniques in this part.

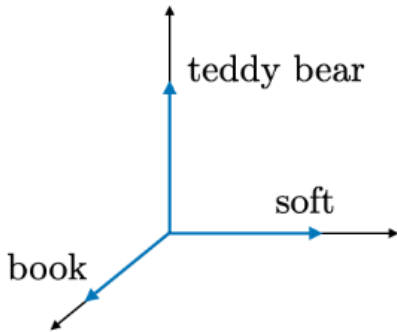
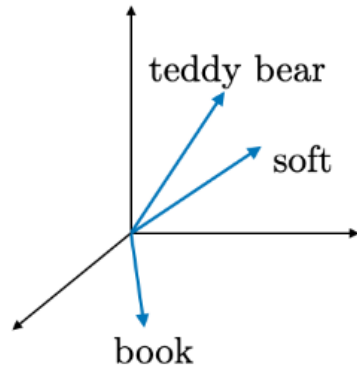
#### Word segmentation

Most of the research on sentiment analysis uses English datasets, and word representations can be applied directly because each syllable is considered a word. In contrast, words in Vietnamese can contain 1 or more syllables. For instance, a 6-syllable sentence “Tôi là một nghiên cứu viên” forms 4 words sentence “Tôi là một nghiên\_cứu\_viên”. Because of this, applying word representations without any pre-processing can cause a degradation in the model performance.

In this project, we experiment with applying a word segmenter to the text before training to see if there is any performance improvement. RDRSegmenter from VnCoreNLP is the chosen word segmentation method.

#### Word representations

There are two main ways to represent words: One-hot Encoding and Word Embedding.

1-hot representation	Word embedding
	
<ul style="list-style-type: none"> <li>• Noted <math>o_w</math></li> <li>• Naive approach, no similarity information</li> </ul>	<ul style="list-style-type: none"> <li>• Noted <math>e_w</math></li> <li>• Takes into account words similarity</li> </ul>

In order to encode the sentences into these kinds of representations, we first need to create a vocabulary. We simply make the vocabulary as a lookup table that stores the order of first appearance of words in the training set. To put it simply, when we see the word “run” in the training set for the first time, and the vocabulary already has 1462 words, then  $\text{vocabulary}[\text{“run”}] = 1462$ . In this way, the vocabulary will be a dictionary that maps each unique word in the training set to an integer. However, there are some special cases as follows:

- Name: The authors of the dataset converted all human names to the form of ‘wzjwz<number>’. For example “wzjwz208”, “wzjwz213”, “wzjwz201”,... We decided to encode all these names into a single “<name>” token in the vocabulary.
- Unknown words: Since the validation data and test data are considered future unseen data and are not included in the training phase, we only create a vocabulary from the training data. So every new word in the dev set and test set that didn’t appear in the training set will be encoded as an “<unknown>” token.
- Padding: In order to pad shorter sentences, we need a special padding token. In this case, we create a “<pad>” token.
- Punctuations: We don’t remove punctuations inside the sentences. Since in written language, punctuations do contribute to the overall semantic of the sentences. For instance, “!” expresses surprise, “?” expresses questions, and sometimes rhetorical questions,...

Now that we have the vocabulary, we can encode our data as follows:

### 3.1. One-hot encoding

Each sentence is encoded into a sequence of vectors, where each vector represents the corresponding word in the original sentence. The vectors will be in the form:

$$x_t = [x_t^{[0]}, x_t^{[1]}, x_t^{[2]}, \dots, x_t^{[V-1]}]$$

Where:

- $V$ : is the size of the vocabulary
- $x_t$ : is the  $t^{\text{th}}$  word in the sentence
- $x_t^{[i]} = 1$  if  $\text{vocabulary}[x_t] = i$ ,  $x_t^{[i]} = 0$  if  $\text{vocabulary}[x_t] \neq i$ .

### 3.2. Word embedding

To use word embedding, we first need a *tokenization* of sentences. Basically, *tokenization* encodes each word in the sentence using the vocabulary:

$$x_t \leftarrow \text{vocabulary}[x_t]$$

Now we can add an embedding layer at the start of our neural network. This embedding layer will embed each integer token in the sentence into a vector representation. This embedding is learned in the training phases.

## IV. Models

### 4.1. RNN

The first and most simple model we use in this project is Recurrent Neural Network (RNN). This part is a little reminder about RNN models.

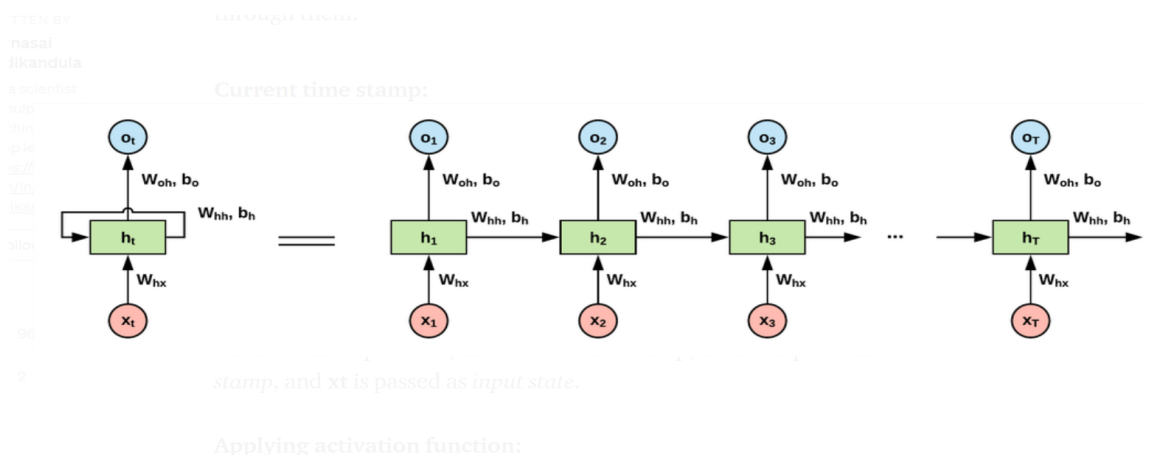


Fig 4.1.1. The general form of a Recurrent Neural Network

Unlike the usual feed forward neural networks, RNN uses hidden states to carry information in sequential data. For each timestep  $t$ , the hidden state  $h_t$  and output  $o_t$  can be expressed as follows:

$$h_t = f(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

$$o_t = g(W_{oh}h_t + b_o)$$

Where:

- $x_t$ : is the input data at timestep  $t$ . More specifically, the  $t^{\text{th}}$  word in the sentence in our case.
- $h_t$ : is the hidden state at timestep  $t$ .
- $o_t$ : is the output at timestep  $t$ .
- $W_{hx}$ ,  $W_{hh}$ ,  $W_{oh}$ ,  $b_h$ , and  $b_o$ : are shared weights and biases as described in Fig 4.1.1.
- $f$  and  $g$ : are activation functions.

The calculation at each timestep can then be described visually as a RNN cell as follow:

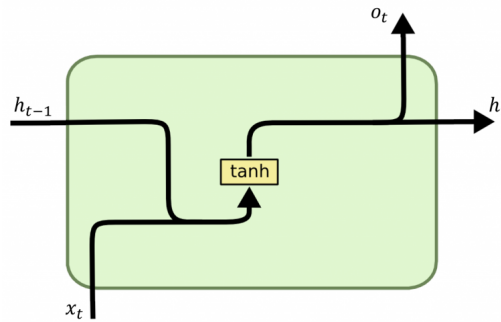


Fig 4.1.2. An example of a RNN cell

For our specific problem, sentiment analysis, we use a special type of RNN architecture: the many-to-one architecture.

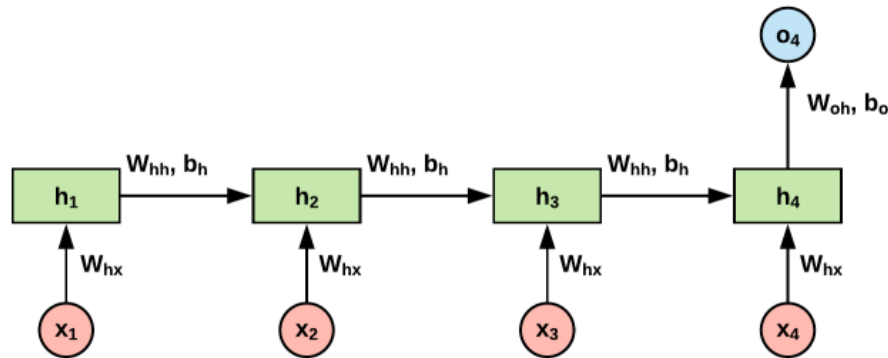


Fig 4.1.3. Example of many-to-one RNN



This architecture is similar to the general form discussed above, except for the fact that only the last output is taken into account. This output will be in the form of a probability vector  $[o^{[0]}, o^{[1]}, \dots, o^{[N-1]}]$  where  $o^{[i]}$  is the probability that the current sentence belongs to class  $i$ . About the activation functions, we choose  $f$  as tanh and  $g$  as log-softmax. Basically the log-softmax function is just applying softmax on the output and then taking the log of the resulting vector. The softmax function makes sure that the final output is a probability vector (i.e.  $o^{[i]} \geq 0 \forall i, \sum_{i=0}^{N-1} o^{[i]} = 1$ ,  $N$  is the number of classes). The log is only for convenience when calculating the loss. More specifically, negative log likelihood loss taken from log-softmax output yields the same result as taking cross entropy loss straight from the output vector.

RNN can be extended further by adding more layers, making it a Deep RNN model.

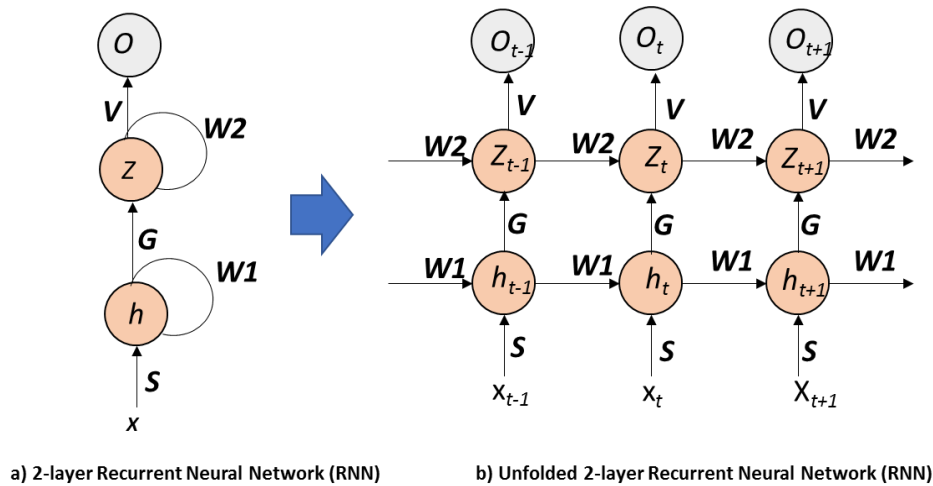


Fig 4.1.4. An example of deep RNN

Deep RNN has multiple layers of hidden states, which help increase the expressiveness of the network. For our problem we use 3-layers RNN.

## 4.2. Long Short Term Memory

### 4.2.1. Input data

Similar to Recurrent Neural Network (RNN), we will also use 2 types of input data: - One Hot vector for each word.

- Token vector for each word combined with the Embedding Layer.

### 4.2.2. Network Architecture

To make the model perform better on longer sentences, we try to use the LSTM model. The difference between LSTM and RNN is the cell formula:

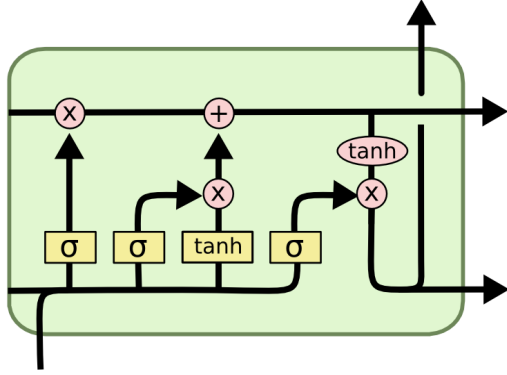


Fig 4.2.1 LSTM cell

$$\begin{aligned}
 f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
 o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
 c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \\
 h_t &= o_t \circ \sigma_h(c_t)
 \end{aligned}$$

x: input vector to the LSTM unit  
 f: forget gate's activation vector  
 i: input/update gate's activation vector  
 o: output gate's activation vector  
 h: hidden state vector also known as output vector of the LSTM unit  
 c: cell input activation vector

Thanks to the forget gate and update gate, the LSTM model can find the relationship between words that are further in the sentence compared to regular RNN.

In the initial model, we only applied the linear classifier layer on the output of the final LSTM cell to predict the label.

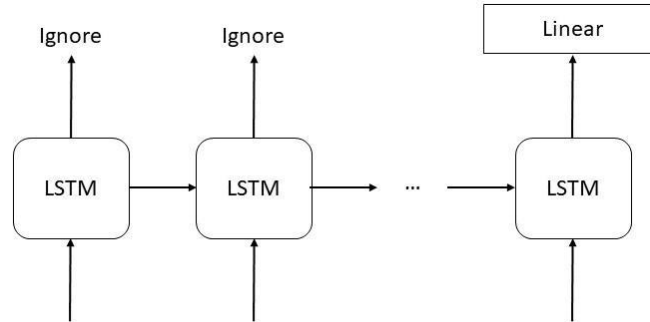


Fig 4.2.2 Network without sum of outputs

However, since all information of sentences is encoded into the final output cell, this approach does not effectively utilize the information in all the hidden states of all cells, so we decide to sum all outputs of all cells and apply the linear classifier layer on it to predict the sentiment (Fig 4.2.3).

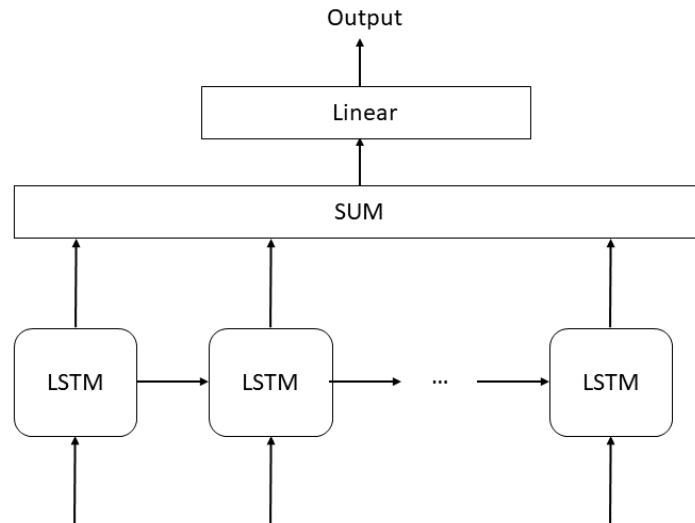


Fig 4.2.3 Network with sum of outputs

We expect that this implementation will help the model to converge faster and achieve better results. We will compare the performance of these two approaches in the experimental result part to verify our hypothesis.

### 4.3. Transformer

#### Background

The Transformer architecture, proposed by Vaswani et al. in 2017, is an attempt to solve the problem of parallelization, and long-term dependency on some other NLP models like RNN and LSTM. In this project, we would apply the architecture to handling the task of sentiment analysis and evaluate its performance.

#### Input data

We use the tokenized data as mentioned above.

Also, both the word segmenting data, and the word splitting data are used and compared.

#### Model architecture

Our model architecture is inherited from the proposed model in the paper: Attention is all you need by Vaswani et al. Since the Transformer is not included in the course Introduction to Deep learning, we will first walkthrough the original model by Vaswani et al. , then describe the differences in our model.

- **Original Transformer model**

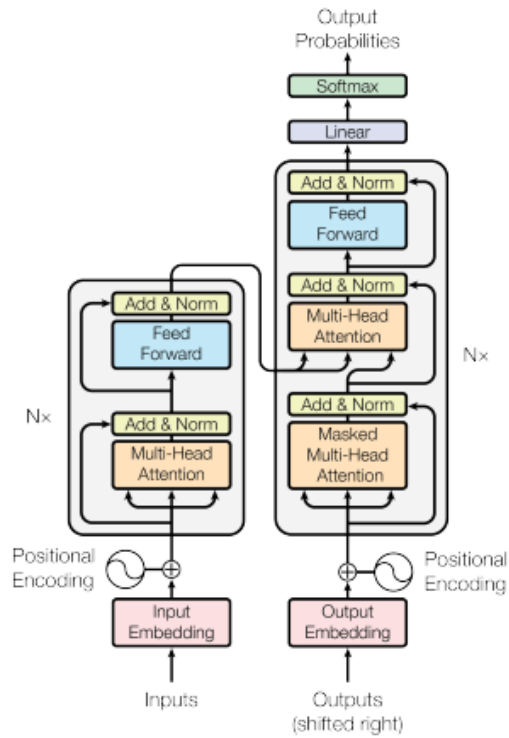


Fig 4.3.1. Transformer model by Vaswani et al. 2017

Like most recent neural sequence transduction model, the Transformer has an encoder-decoder structure: “the encoder maps an input sequence of symbol representations  $(x_1, \dots, x_n)$  to a sequence of continuous representations  $z = (z_1, \dots, z_n)$ . Given  $z$ , the decoder then generates an output sequence  $(y_1, \dots, y_m)$  of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.”. For both the encoder and decoder, the Transformer uses self-attention and point-wise, fully connected layers.

**a. Scaled dot-product Attention**

- An attention function maps a query and a set of key-value pairs to an output.
- The input consists of queries and keys of dimension  $d_k$ , and values of dimension  $d_v$ .
- Let  $Q$  the matrix of queries,  $K$  the matrix of keys,  $V$  the matrix of values, the matrix of output is calculated as:

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

- The authors suspect that “for large values of  $d_k$ , the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.” Thus, in the formula, the dot-product is scaled by  $\frac{1}{\sqrt{d_k}}$ .

**b. Multi-Head Attention**

- The multi-head attention includes several parallel attention layers, or heads.
- For head number of  $h$ , the queries, keys and values are projected  $h$  times with different linear projections (which means different weight matrix). On each of the projections, the attention output is calculated in parallel. These outputs then are concatenated and once again projected, resulting in the final values.

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

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

- Multi-head attention allows the model to “attend to information from different representation subspaces at different positions”.

**c. Point-wise feed forward network**

- Besides the attention sub-layers, each of the layers in the encoder and decoder contains a fully connected feed-forward network. This sub-layer consists of two linear transformations with a ReLU activation in between.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

**d. Embeddings and softmax**

- The input tokens and output tokens are converted into vectors of dimension  $d_{\text{model}} = 512$ .
- Linear-transformation and softmax function are used to convert the decoder output to predicted next-token probabilities.

**e. Positional encoding**

- Since the Transformer has no recurrence nor convolution, the information about positions of the tokens in the sequence needs to be added by another method.
- “Positional encoding” is added before encoder and decoder stacks.
- In the original model, Sine and Cosine functions are used for the positional encoding:

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

Where  $pos$  is the position and  $i$  is the dimension ( $i = d_{model} = 512$ ).

- Vaswani et al. has experimented with learned positional embeddings instead, and found that the two versions produced nearly identical results. Thus, we would also use Sine and Cosine functions for positional encoding

- **Our model**

Since this project's task is sentiment analysis, our model only includes encoders with some additional linear layers. Also, the pretrain method that we use is Masked Language Modeling (to be discussed later), which means the model when it is used for pretrain versus when it is used for the main task have different outputs. This is made feasible by adding a Linear layer and a Weighted\_pool layer that will be skipped through during the pretrain phase and activated in the latter phase. Below are the two model architectures:

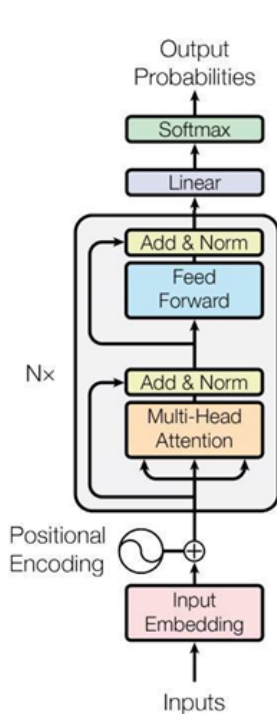


Figure 4.3.2.a: Our model on MLM task

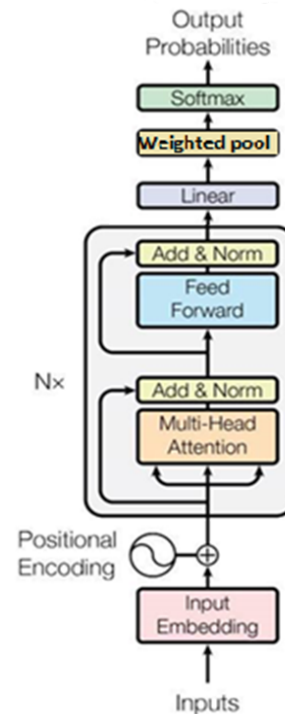


Figure 4.3.2.b: Our model on sentiment analysis task

Next, we will explain the model architecture and the dimension size of the data batch as it goes through each layer.

Given the parameters defined as follow:

- **Batch\_size**: number of samples (sentences) in a training batch, chosen to be 64.
- **Sequence\_size** (equivalent to  $d_{\text{model}}$ ): the length (number of tokens) of each sentence. All sentences will be padded or cut off if necessary in order to share this length. The value for **sequence\_size** is chosen to be the number of tokens of the longest sentence in the train dataset.
- **Embedding\_size**: The dimension of the embedding layer.
- **Vocab\_size**: the number of words in vocabulary. The vocabulary is built on the train dataset.
- **Num\_classes**: number of label classes, equals to 3 (Positive, Neutral, Negative).

The input would have dimension of: **Batch\_size x Sequence\_size**

After the encoding layers: **Batch\_size x Sequence\_size x Embedding\_size**

From this part, the two models differ:

- For model 4.3.2.a, the linear layer produces output with the dimension of **vocab\_size**, then the batch would be put through softmax to get the token-probabilities for each word in the sentence. This is the output of the model in the MLM task. The size of output is:

**Batch\_size x Sequence\_size x Vocab\_size**

- For model 4.3.2.b, the linear layer produces output with the dimension of **num\_class**, then the batch would be put through softmax to get the sentiment-probabilities for each word in the sentence. The size of the batch at this stage is:

**Batch\_size x Sequence\_size x Num\_classes**

- Each “word” now has its own sentiment value. To get the sentiment of a whole sentence, one more linear layer is used to pool the values from all of the words in that sentence. We call this layer “weighted\_pool”, since it is a replacement of the initial idea of using average pooling. After the Weighted\_pool layer, the output dimension would be:

**Batch\_size x 1 x Num\_classes**

- Afterward, the batch would be put through softmax to get the sentiment-probabilities, then squeezed into the size:

**Batch\_size x Num\_classes**

This is the final output of the second model

## Loss function

For both pretrain and train phases, essentially, we use Cross Entropy Loss.

- MLM task: for each sentence, the loss is the mean value of the tokens' Cross Entropy losses.

$$Loss = \sum_{j=1}^{output\ size} \frac{- \sum_{i=1}^{sequence\ length} t_i \cdot \log \hat{t}_i}{sequence\ length}$$

where  $t_i$  is the predicted token  $i$  of sentence  $j$ ,

and  $\hat{t}_i$  is the ground truth token  $i$  of sentence  $j$ .

- Sentiment analysis task:

$$Loss = - \sum_{i=1}^{output\ size} y_i \cdot \log \hat{y}_i$$

Where  $y_i$  is the predicted label for sample  $i$ ,  $\hat{y}_i$  is the ground truth label for sample  $i$ .

## 4.4. BERT

BERT (Bidirectional Encoder Representations from Transformers) is an architecture proposed by Google AI in 2018. Since the time it was released, the model has been considered a game-changer, as it achieved state-of-the-art results in various NLP tasks, including Question Answering, Natural Language Inference, and others. In this section, we give an overview of the BERT architecture and its training procedure. Additionally, we briefly explain a variant of BERT, namely RoBERTa. Furthermore, we discuss a pre-trained Vietnamese RoBERTa model - PhoBERT - which is used in this project.

### 4.4.1 BERT Architecture

BERT makes use of the Transformer architecture described above. As being a model for language representations learning, BERT only includes a stack of Transformer encoders in the model and does not use the Transformer decoder.



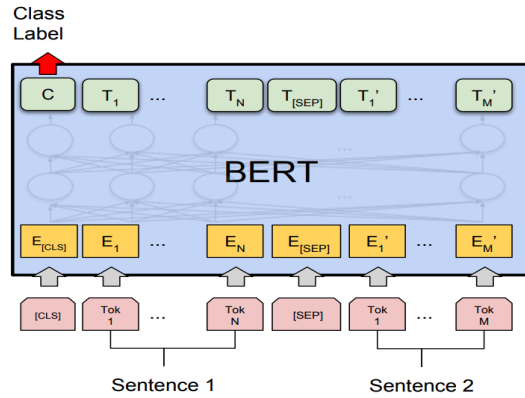


Fig 1.1. BERT Architecture.

As shown in the figure above, the input to the model is a pair of sentences. In the original implementation of BERT, a sentence here is a segment of contiguous text sampled from the data rather than a true linguistic sentence. To distinguish between the two sentences in the input, a *[SEP]* token is placed at the end of each sentence. Additionally, a *[CLS]* token is placed before the first sentence, which its hidden state output from the encoders will be used for a later described classification task. The input is now embedded into vectors, which is denoted *E* in the figure, then it is fed into the Transformer encoders stack.

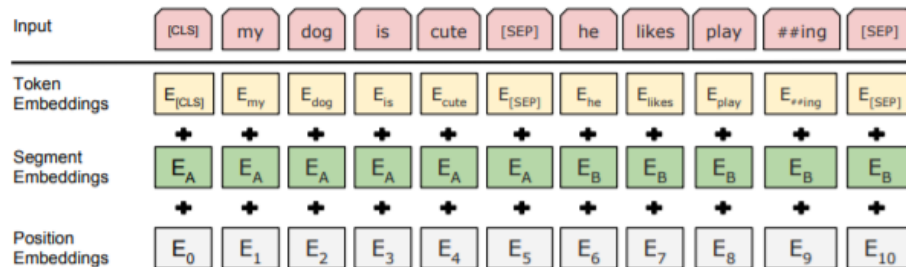


Fig 1.2. Input representations of BERT.

For representing the input to the model, 3 embeddings strategies in the above figure are used:

**Token Embeddings.** The input token is first embedded with Wordpiece, a subword tokenization algorithm. The algorithm first initializes the vocabulary with all the characters appearing in the data, then it learns to merge the characters into subwords that maximize the likelihood of the training data. The algorithm is very similar to BPE, except for the fact that BPE chooses the subwords that are most frequent in the training data. With this Wordpiece tokenizer, OOV words in the input are easily handled by breaking them into subwords and characters.

**Segment Embeddings.** For distinguishing the two input sentences, apart from the *[SEP]* token described above, segment embeddings are added, which are binary numbers that tell which tokens belong to which sentences.

**Position Embeddings.** This embedding is used to tell the position of the tokens. The implementation of this is exactly the same as positional embeddings for the original Transformer model.

Finally, the input embeddings fed to the model are the sum of the above 3 embeddings.

#### 4.4.2 BERT Training Objectives

For later uses of the model in downstream tasks, BERT is trained simultaneously with 2 supervised tasks, namely Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).

##### Masked language modeling

It is obvious that we cannot train BERT like other traditional left-to-right or right-to-left language models as the attention mechanism in the Transformer is bidirectional, so the authors of BERT came up with the strategy of masked language modeling.

In MLM, a random sample of the tokens in the input is replaced with the *[MASK]* token. Firstly, BERT uniformly selects 15% of the input tokens for replacement consideration, and then 80% of the selected tokens are replaced with *[MASK]*, 10% of them are replaced with random tokens and the rest 10% are left unchanged. The training objective is a cross-entropy loss which only considers predicting the masked tokens.

##### Next sentence prediction

NSP is a binary classification problem to determine whether the two segments in the input follow each other in the training data. This is done by passing the hidden state output of the *[CLS]* token to a binary classification loss.

#### 4.4.3 RoBERTa

Even though BERT seems to be a perfect model as it broke many state-of-the-art records, it is still significantly undertrained and can be improved. That is when RoBERTa took the place with some simple modifications to the BERT model.

##### Dynamic Masking

In the original implementation of BERT, some tokens are masked before going through the model. However, it is done only once in the data preprocessing step and the masking of each input is kept exactly the same during training. In the case of RoBERTa, the masking of an instance is changed whenever it is fed to the model.

## **NSP Removal**

The authors of the RoBERTa paper experimented with different setups to find out whether removing the NSP training objective increases the performance of the model. The results show that removing NSP slightly improves BERT performance in several downstream tasks.

## **RoBERTa for Sentiment Analysis**

For our task of Sentiment Analysis, a linear layer is appended on top of the Transformer encoders. At the beginning of the fine-tuning process, the Transformer encoders are initialized from the pre-trained parameters, while the newly added linear layer is initialized randomly.

### **4.4.4 PhoBERT**

With the aim to leverage the development of Vietnamese NLP, PhoBERT was released. The model has the same architecture as BERT and uses the pre-training approach of RoBERTa.

#### **Word segmentation as input**

The authors also apply RDRSegmenter from VnCoreNLP for word segmentation to the text before training. However, the authors did not include the results of PhoBERT without word segmentation, so we cannot see if using word segmentation has a significant improvement. Later in the experiment, we will compare the model training with and without word segmentation.

## **Training data**

The model is trained on about 20GB of text collected from the Vietnamese Wikipedia corpus and a Vietnamese news corpus. Experimental results of the PhoBERT paper show that the pre-trained model achieves state-of-the-art results on several downstream tasks, including Part-of-speech tagging, Dependency parsing, Named-entity recognition, and Natural language inference.

## **4.5. Ensemble Learning**

To push our result further, we also try to use ensemble learning. There are two main ways of computing an ensemble output:

- Soft voting: Take the average of the outputs and predict the label based on that output.
- Hard voting: Decide the output based on the majority of the predictions by the individual models.

We have a few trained classifiers, each one achieving about 89-93% accuracy. A simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a hard voting classifier:

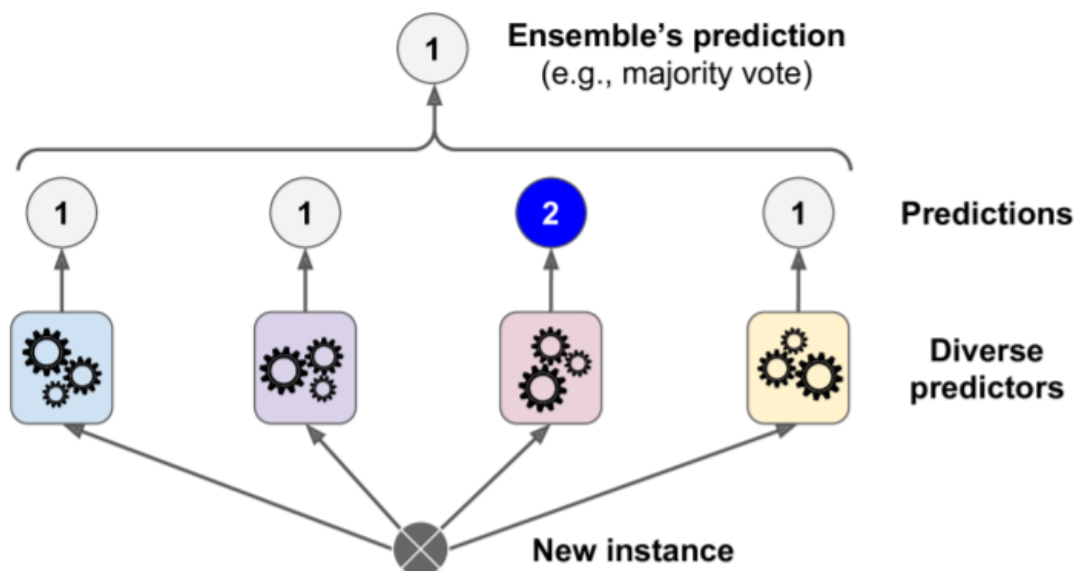


Fig 4.5.1: Hard voting strategy

Our Transformer model only does a slightly better result than RNN and LSTM. However, due to the law of large numbers, our ensemble prediction can be a bit better. This could be like tossing a biased coin that has a 51% chance of heads and 49% of tails. If we do the math, we find that the probability of obtaining a majority of heads after 1000 tosses is close to 75%. That's the reason why we use the hard voting strategy.

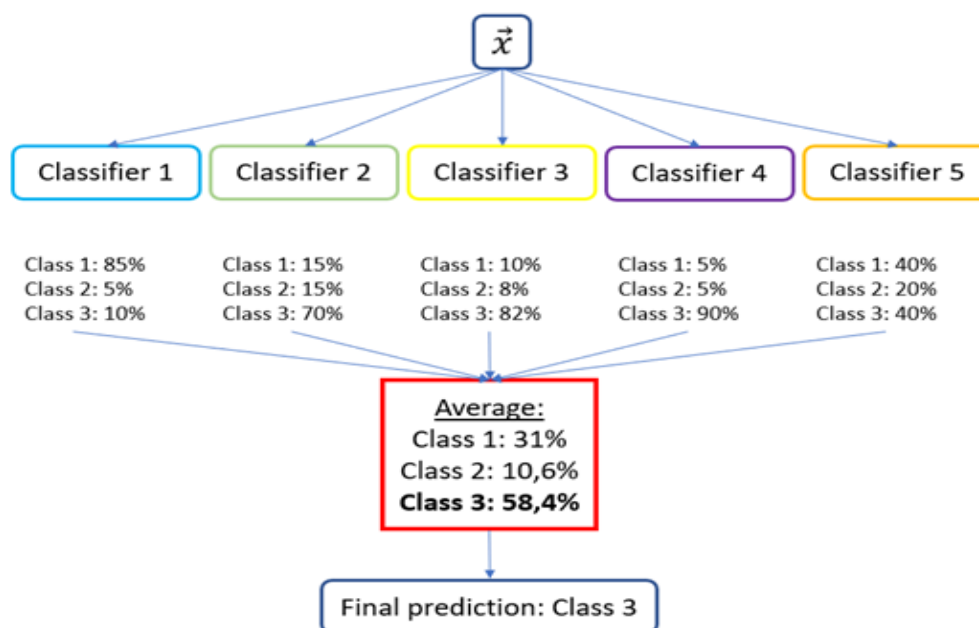


Fig 4.5.2: Soft voting strategy

The soft voting type, predicts the class label based on the argmax of the sums of the predicted probabilities of the individual estimators that make up the ensemble. Soft voting can improve on hard voting because it takes into account more information; it uses each classifier's uncertainty in the final decision.

In our case, we try both of these two options. The models included in the ensembles are the models discussed above, and are fully trained. In other words, ensemble is used only for inference.

## V. Experimental results

### 5.1. RNN

In the training phase, we train the model for 20 epochs. We see that our model converges very fast for this particular problem. And at the 20<sup>th</sup> epoch, the model is already overfitting. Thus, we take only the model state that gives the best dev loss to evaluate further.

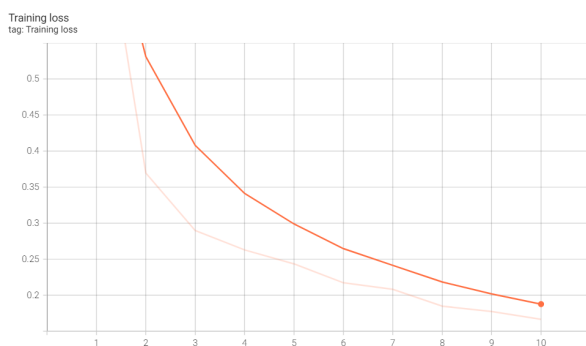


Fig 5.1.1. RNN Training loss

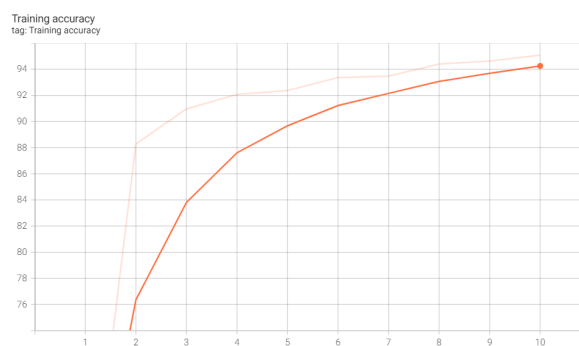


Fig 5.1.2. RNN training acc



Fig 5.1.3. RNN Dev loss

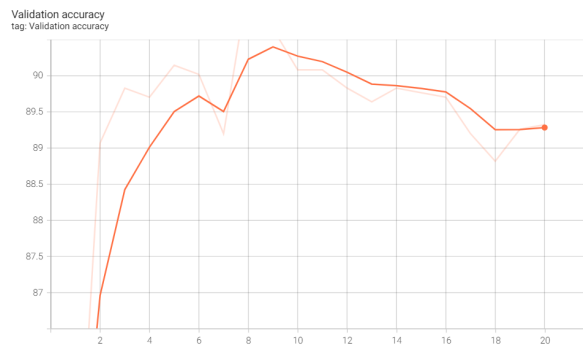


Fig 5.1.4. RNN Dev acc

In this example, the model achieves best dev loss at the 9<sup>th</sup> epoch. And sometimes the best dev loss and dev acc can be achieved at the same state (this is not guaranteed though).

We spent some time tuning the hyper-parameters to improve the result. The tuned hyper-parameters are:

- The learning rate
- The length of the input sentences. We pad shorter sentences with a padding token, and cut longer sentences so that they have the same appropriate length.
- The size of the hidden states
- The embedding dimension of the embedding layer.

This tuning process is automated. Basically it's just a brute force search through a set of possible values of the hyper-parameters to choose the ones that yield the best dev loss.

We evaluate the performance of RNN on different settings. In each setting, we train the model a couple of times and take the average of the results. Here are the experimental results on dev set:

No	Model	Hyper params tuning	Acc	F1 Score			
				Negative	Neutral	Positive	Weighted Avg
1	1 layer, one-hot	No	88.31	88.94	42.03	91.66	88.16
2	1 layer, word embedding	No	88.82	89.84	43.10	91.09	88.36
3	3 layers, one-hot	Yes	90.59	91.77	46.51	93.08	90.35
4	3 layers, word-embedding	Yes	90.84	92.03	42.52	93.60	90.54
5	3 layers, one-hot, word segmentation	Yes	90.52	91.49	39.25	93.08	89.89
<b>6</b>	<b>3 layers, word-embedding, word segmentation</b>	<b>Yes</b>	<b>91.09</b>	<b>92.46</b>	<b>41.12</b>	<b>93.18</b>	<b>90.46</b>

The 6<sup>th</sup> model here gives the best result on the validation set, with an accuracy exceeding 91% sometimes.

Classification summary on dev set:				
	precision	recall	f1-score	support
Negative	0.8818	0.9631	0.9207	705
Neutral	0.7000	0.2877	0.4078	73
Positive	0.9515	0.9255	0.9383	805
accuracy			0.9128	1583
macro avg	0.8444	0.7254	0.7556	1583
weighted avg	0.9089	0.9128	0.9060	1583

Classification summary on test set:				
	precision	recall	f1-score	support
Negative	0.8491	0.9624	0.9022	1409
Neutral	0.6512	0.1677	0.2667	167
Positive	0.9325	0.8950	0.9134	1590
accuracy			0.8866	3166
macro avg	0.8109	0.6750	0.6941	3166
weighted avg	0.8805	0.8866	0.8743	3166

Our result shows that RNN can work quite well in our particular problem. Using word embedding as data representation usually outperforms one-hot encoding. DeepRNN yields better results than simple RNN. And after tuning the hyper-parameters carefully, the results improve significantly.

## 5.2. LSTM

### 5.2.1. Sum all outputs

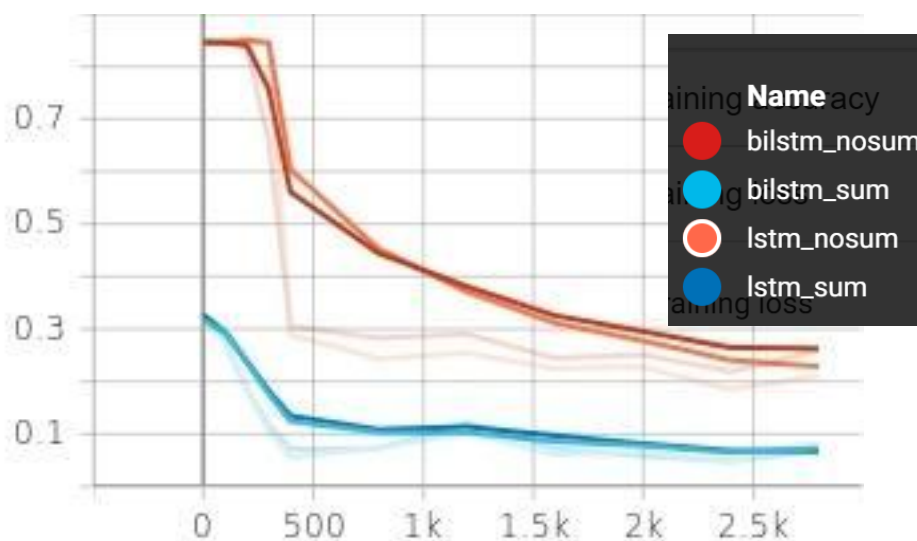


Fig 5.2.1 Training loss over time

From the plot, we can see that when taking the sum of all the output as the encoded vector to predict the label, models converge faster.

We obtain the following results:

Models	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
1-directional, no sum output	90.84	92.32	43.55	93.15	90.49
<b>1-directional, sum output</b>	<b>89.32</b>	<b>91.67</b>	<b>48.89</b>	<b>91.84</b>	<b>89.79</b>
Bidirectional, no sum output	89.83	91.55	0.00	92.31	87.71
<b>Bidirectional, sum output</b>	<b>89.96</b>	<b>92.20</b>	<b>52.97</b>	<b>92.30</b>	<b>90.44</b>

Since the F1 score of the neutral class is significantly lower when not using the sum of output, we conclude that using the sum output of all cells, the model achieves better results.

### 5.2.2 Sequence Length

To train the model, we have to pad the sequence to a fixed length. Following is the result after tuning sequence length on the model with sum output and non-bidirectional.

Sequence Length	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
10	89.01	90.86	45.12	91.90	89.28
15	88.06	90.08	46.07	91.36	88.70
20	88.76	92.28	44.33	91.09	89.46
30	87.93	91.03	48.91	89.66	88.39
<b>40</b>	<b>90.15</b>	<b>91.53</b>	<b>46.03</b>	<b>92.36</b>	<b>89.86</b>

We select sequence length = 40 as the best hyperparameter. Compared to regular RNN, LSTM can use a longer padding length but can still achieve good results.

### 5.2.3. One hot input and Embedding layer

To reduce the training time and enable the model to learn on its own the word representation, we try to use the embedding layer of Pytorch with the embedding dimensions 300, 400 and 500 respectively and compare with the best model using one-hot word representation.



	Embedding dimension	Accuracy	F1 Score			
			Negative	Neutral	Positive	Weighted Avg
Non-bidirectional	300	89.70	90.97	44.62	92.20	89.46
	400	89.64	91.64	46.63	92.27	89.89
	<b>500</b>	<b>90.21</b>	<b>90.96</b>	<b>53.52</b>	<b>92.76</b>	<b>90.15</b>
Bidirectional	300	89.51	90.68	45.86	92.72	89.65
	400	89.70	91.44	44.44	92.51	89.82
	500	88.95	91.09	44.32	92.02	89.40
Best one hot model	-	90.15	91.53	46.03	92.36	89.86

From the table, we can see that the model with embedding dimension of 500 outperforms the model using one-hot word representation.

#### 5.2.4 Word Segmentation

We also train the model on the word segmentation dataset to see if it can improve the result.

	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
No word segmentation	90.21	90.96	<b>53.52</b>	92.76	90.15
Word Segmentation	<b>90.27</b>	<b>91.91</b>	46.67	<b>92.92</b>	<b>90.33</b>

Compared to the performance when not using segmentation, the model with word segmentation has slightly better accuracy and weighted F1 but lower F1 score for neutral class.

#### 5.2.5 Conclusion:

The best model is the model with the following hyperparameter:

- Sum all the output cells
- Non-bidirectional
- Embedding dimension = 500

- Sequence Length = 40

Final result on test set:

	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
No word segmentation	89.01	90.84	42.81	91.70	88.74
Word Segmentation	89.01	91.30	42.11	91.44	88.78

## 5.3. Transformer

### 5.3.1. Hyperparameters

Below are some preselected hyperparameters that are used for training the models. Since these values provide satisfactory results, they are not tuned.

- Batch size: 64
- Input/ output dimension ( $d_{\text{model}}$ ) : number of tokens in train data's longest sentence.
- Initial learning rate value: 0.001
- Learning rate scheduler: reduce learning rate 0.7 times after every 5 epochs.
- Dropout: 0.2
- Number of training epochs: *adaptive*

The models will be trained until converge. Firstly, each model will be trained for 10 epochs, and its performance on the validation set is validated after each epoch. During training, the model with the best performance (the performance criteria is mentioned below) are saved along with its epoch numbers as a checkpoint. After every 10 epochs, if the value of latest checkpoint epoch is less than 4 units close to the last epoch (i.e:  $\text{current\_epoch\_num} - \text{check\_point\_epoch} < 4$ ), then the model would have to train for 10 more epochs.

- Performance evaluation & Checkpoint condition: F1-score and loss on validation set.
  - Since Loss calculation relies on distance, which is more sensitive to outliers, we also take into account the f1 score.
  - For the training phase, two checkpoint models are tracked. Those are the model with the lowest loss on the validation set, and the model with the highest f1-score on the validation set.

- Through practical results, we find out that the f1 score criteria provides models with better performance on both validation set and test set.

Below is the list of the tuned hyperparameters, their experimented values. The values that provide the best performance (lowest dev loss) are highlighted. Please note that nhead must be divisible by emsize.

Hyperparameter name	Description	Values
emsize	Embedding dimension	20; <b>40</b> ; 100; 200
nhead	Number of heads in multihead attention	1; <b>2</b> ; 4
nlayers	Number of encoding layers	<b>1</b> ; 2; 4
d_hid	Dimension of the feedforward network model in encoder layers	20; <b>50</b> ; 200; 400

The details about the tuning results are not included in this report. More information about the tuning process can be found in the notebook Transformer.ipynb.

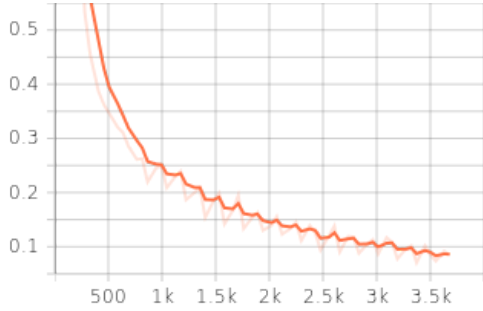
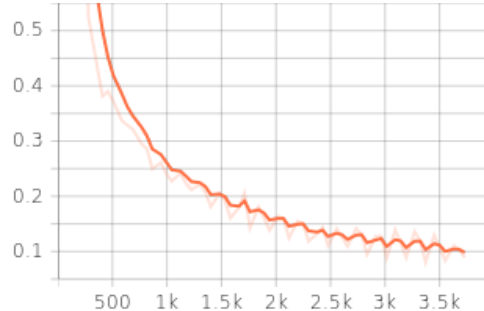
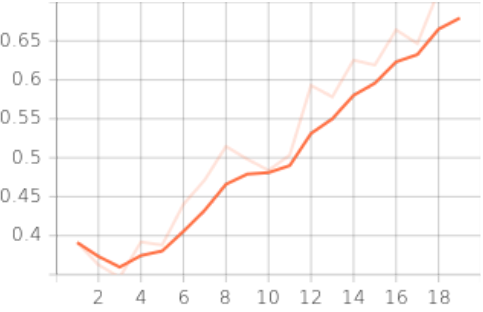

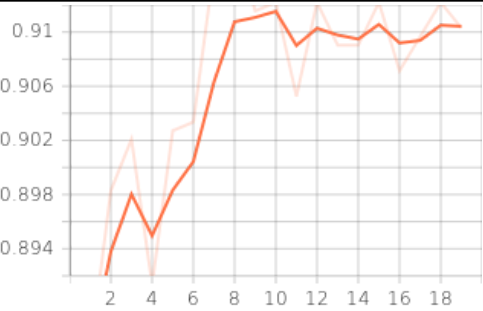
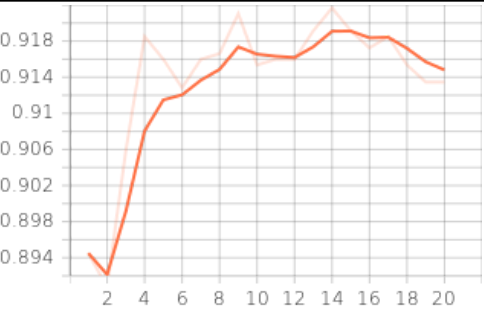
### 5.3.2. Pre training or not

The pretrain method that we use is Masked Language Modeling (MLM). MLM consists of giving the model a sentence and optimizing the weights inside in order to output the same sentence on the other side. While the input and label sentence are essentially the same, the input sentence has some of its tokens masked.

In this project, the probability for a token to be masked is 0.15.

Both the train dataset and validation data set are masked to create a train set and validation set for pretraining. The pretraining tactic is the same as the one in training, with one difference is that only the model on the epoch that has the lowest loss on validation set is saved (since f1 score is not calculated).

Below is the comparison of the performance of two models during the training stage, one is pretrained, and one is not.

	Not pretrained model	Pretrained model
Train loss		
Valid loss	 <p>Best valid loss epoch: 3 Valid loss: 0.35 Valid F1: 0.9033</p>	 <p>Best valid loss epoch: 4 Valid loss: 0.33 Valid F1: 0.9179</p>
F1 valid	 <p>Best valid F1 epoch: 10 Valid loss: 0.51 Valid F1: 0.9149</p>	 <p>Best valid F1 epoch: 14 Valid loss: 0.55 Valid F1: 0.9189</p>

We can see that the pretrained model performs better than no pre-trained model in terms of both loss value and f1 score.

Also, while the models achieve the best loss value just after a few training epochs, the model would still have to train more to get the best f1 score on validation set.

### 5.3.3. Result table

Model No.	Data preprocessing	Pretrain	Checkpoint criteria	Valid acc.	Valid F1	Test acc.	Test F1
1	Word split, tokenized	Not pretrained	Lowest loss on validation set	89.07	89.23	87.05	87.35
2			Highest F1 score on validation set	91.60	91.46	89.51	89.10
3		Pretrained	Lowest loss on validation set	91.85	91.79	89.58	89.35
4			Highest F1 score on validation set	92.17	91.89	89.67	89.30
5	Word segmented, tokenized	Pretrained	Lowest loss on validation set	88.50	88.63	86.13	86.38
6			Highest F1 score on validation set	<b>92.23</b>	<b>92.04</b>	<b>89.89</b>	<b>89.44</b>

From the result, we have some remarks:

- Models have better performance on word segmented data.
- Pre-trained models perform better than no pre-trained models.
- Selecting models based on f1 score would result in a better model than by selecting based on the loss value of the validation set.

The model number 6, which was pretrained and selected based on f1-score, provides the best performance.

Valid set performance report				
Dev loss 0.37				
	precision	recall	f1-score	support
negative	0.9135	0.9589	0.9356	705
neutral	0.6034	0.4795	0.5344	73
positive	0.9541	0.9304	0.9421	805
accuracy			0.9223	1583
macro avg	0.8237	0.7896	0.8040	1583
weighted avg	0.9199	0.9223	0.9204	1583

Fig 5.3.1.a: Model no.6's report on validation set.

Test set performance report				
Test loss 0.52				
	precision	recall	f1-score	support
negative	0.8834	0.9517	0.9163	1409
neutral	0.5810	0.3653	0.4485	167
positive	0.9358	0.9082	0.9218	1590
accuracy			0.8989	3166
macro avg	0.8001	0.7417	0.7622	3166
weighted avg	0.8938	0.8989	0.8944	3166

Fig 5.3.1.b: Model no.6's report on test set.

Below are some training samples that the model no.6 misclassified:

Sentence	Prediction	True label
ôn_tập đúng theo thầy nhưng không biết cuối kỳ được mấy điểm .	negative	neutral
giáo_viên có phong_cách giảng_dạy nghiêm_túc .	neutral	positive
cập_nhật bài_tập mới .	positive	negative
cho làm nhiều bài_tập .	positive	negative
giảng_viên tạo điều_kiện làm nhiều bài_tập .	positive	negative
giảng_viên giảng_dạy theo đúng thời_gian môn_học .	positive	neutral
giảng_viên rất đảm_bảo giờ lên_lớp .	positive	neutral
thực_hành nhiều .	positive	negative

## 5.4. BERT

For experimenting with PhoBERT, we tried two model configurations, which are PhoBERT-base and PhoBERT-large, respectively. PhoBERT-base has a total number of parameters of about 110m, while it is 340m for that of PhoBERT-large. A linear layer is randomly initialized on top of the pre-trained model. The model is fine-tuned with AdamW optimization and Cross-Entropy loss function. We use a learning rate of 3e-5, which will decay linearly. Additionally, while it is suggested in the RoBERTa paper that using a larger batch size in downstream tasks increases the model performance, we only use a batch size of 8 because of the limitation in computational resources. The input to the model will be padded by the maximum length regarding each training batch.

We begin by comparing different methods on the dev set to choose the best model. Please note that for training the model below **we skip the name entities preprocessing step**, ie. replacing names with <name> tokens. We will explain the reason for this in **Section 5.5.2 - Further Improvements on BERT**.

Method		Dev set				
Pretrained model	Word Segmentation	Accuracy	F1			
			Negative	Neutral	Positive	Weighted Avg
From scratch	no	91.06	-	-	-	91.20
PhoBERT-base	no	94.57	96.48	60.53	96.06	94.64
PhoBERT-base	yes	95.51	96.85	<b>68.22</b>	96.48	95.36
<b>PhoBERT-large</b>	<b>yes</b>	<b>95.83</b>	<b>96.86</b>	67.19	<b>97.24</b>	<b>95.66</b>

### ***Pre-training vs No Pre-training***

From the first look at the results, we can see that BERT's pre-training approach remains relevant on sentiment analysis, as it achieves superior results compared to the previously described model. Without pre-training, the model reaches an equivalent performance compared to the other models.

### ***Domain Adaptation***

Another finding is that pre-training can handle the problem of domain mismatch. PhoBERT was trained with news and Wikipedia text, which are completely different domains compared to that of our training data - student feedback.

### ***Word Segmentation vs No Word Segmentation***

Apart from that, we also tried fine-tuning PhoBERT without using word segmentation. The results show that using word segmentation only improves the performance slightly on the positive and negative classes, although the model (PhoBERT) was pre-trained with word segmentation. However, the F1 score of the neutral class increases by about 8%. Considering the fact that there are very few samples of neutral classes in the training data, this is a big improvement.

### ***PhoBERT-base vs PhoBERT-large***

As shown in the table, the increase in performance is by little when using PhoBERT-large over PhoBERT-base. This can be due to the small number of training samples. When using a larger model, one should have a bigger amount of data for the loss signal to become significant while updating the parameters of the model.

## **5.5. Final Test Results**

This is the results taken from the test set. Each model below is the configuration that achieves the highest result on the development set when tuning.

Model	Acc	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
RNN	89.23	90.77	29.63	91.85	88.09
LSTM	89.01	90.94	42.81	91.70	88.78
Transformer	89.89	91.63	44.85	92.18	89.44
<b>BERT</b>	<b>93.78</b>	<b>95.22</b>	<b>58.62</b>	<b>95.74</b>	<b>93.52</b>

The settings here are:

- RNN: 3 layers, tuned hyper-parameters, word embedding layer, word segmented data.

- LSTM: sum all the output cells, non-bidirectional, embedding dimension = 500, sequence length = 40. Word segmentation does not significantly influence the result.
- Transformer: Uses tuned hyper-parameters, pretrained, word segmented data.
- BERT: pre-trained using PhoBERT-large, uses word segmented data.

We also tried ensemble learning using our best models to see if it can push the results further on weaker models. We exclude BERT from these ensembles since its performance is much higher than the rest.

Model	Word segmentation	Voting Strategy	Acc	F1 Score			
				Negative	Neutral	Positive	Weighted Avg
LSTM + RNN + Transformer	No	Soft voting	90.05	91.77	44.02	92.24	89.49
		Hard voting	89.73	91.43	41.86	92.08	89.14
	Yes	Soft voting	90.37	91.91	44.09	92.66	89.77
		Hard voting	90.24	91.92	42.52	92.53	89.62
LSTM + RNN	No	Soft voting	89.67	91.59	42.80	91.92	89.19
	Yes		90.05	92.01	41.35	92.35	89.51
RNN + Transformer	No	Soft voting	89.96	91.42	45.04	92.34	89.44
	Yes		90.11	91.69	38.37	<b>92.67</b>	89.37
<b>LSTM + Transformer</b>	No	<b>Soft voting</b>	90.15	91.90	<b>46.98</b>	92.38	89.77
	<b>Yes</b>		<b>90.49</b>	<b>92.45</b>	46.04	92.62	<b>90.08</b>

Overall ensemble does improve the results significantly. Compared to the best Transformer configuration, ensemble can increase up to 0.5% in accuracy and weighted average F1 score.

## Further Improvements on BERT

### Named Entity Handling

As described previously in the Data Preprocessing Section, names are converted into <name> tokens. However, in the case of BERT, we also experimented with training the models with 3 following strategies:



**Model 1.** Train without preprocessing the names, ie. keeping names as they are.

**Model 2.** Train with the data in which all names are removed.

**Model 3.** Train with the data in which all names are replaced with <name> tokens.

For saving computational resources, we only conduct these experiments with PhoBERT-base. The purpose of these 3 experiments is to assess whether there is a significant degradation in BERT performance without data preprocessing. In contrast to the previously described models which are trained from scratch, the performance of BERT may vary because for 2 reasons:

- The pre-training process (PhoBERT) did not see these names.
- The tokenization outputs are different in each of the above 3 experiments. For instance, a name such as 'wzjwz208' is broken into character tokens, while a <name> token is considered as 1 token, as it already appeared in the vocabulary.

Model	Test set				
	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
Model 1	93.52	95.24	59.73	95.14	93.32
Model 2	93.81	95.32	56.18	<b>95.63</b>	93.39
<b>Model 3</b>	<b>94.00</b>	<b>95.72</b>	<b>61.33</b>	95.52	<b>93.82</b>

The results above show that both of the two name entities handling methods achieve better results compared to Model 1, and that replacing names with <name> reaches the highest F1 score and accuracy on the test set. Model 3 even beats the PhoBERT-large configuration without names preprocessing on the test set. This proves that names do contribute to the sentimental contents of sentences.

### Ensemble within BERT configurations

While the results of BERT seem to have reached the peak, we can still improve the performance a bit more. In this Section, we tried different ensemble combinations of BERT models to find the best one. We did not try ensembling BERT with the previously described architectures because the scores of BERT are much higher.

Model	Test set				
	Accuracy	F1 Score			
		Negative	Neutral	Positive	Weighted Avg
Model 1 + Model 2 + Model 3	94.13	95.63	61.32	<b>95.73</b>	93.87
Model 1 + Model 3	94.09	95.71	<b>61.86</b>	95.59	93.86
<b>Model 2 + Model 3</b>	<b>94.28</b>	<b>95.88</b>	61.65	95.70	<b>93.99</b>

In the table, Model 1, Model 2 and Model 3 are the 3 models described above in the **Named Entity Handling** Section. After experimenting, we found that the combination of Model 2 and Model 3 has the best accuracy and F1 score. Compared to the previous best results, which is of Model 3, the accuracy improves by 0.28%, while the weighted F1 score improves by 0.17%. As the results of BERT are already pretty high, this improvement can be considered significant.

## 5.6. Comparing our result to the authors of the dataset

Model	Test set			
	F1 Score			
	Negative	Neutral	Positive	Weighted Avg
Our best model (BERT ensemble)	<b>95.88</b>	<b>61.65</b>	<b>95.70</b>	<b>93.99</b>
By Huy et al. VNU-HCM (BERT + CNN + BiLSTM + LSTM ensemble)	-	-	-	92.79
UIT-VSFC paper by Kiet et al. VNU-HCM (MaxEnt)	90.52	33.99	91.32	87.94

## VI. Conclusion

In this project, we have experimented with several deep learning architectures on the task of Vietnamese sentiment analysis. The results show that using word segmentation improves every models. Furthermore, by applying ensemble mechanism, the combination of 2 BERT configurations achieves the highest accuracy and weighted F1 score, which are of about 94.28% and 93.99%, respectively. In the future, we hope to achieve equivalent or better results on other Vietnamese sentiment analysis benchmarks. Furthermore, we can try to experiment with adding in-domain pre-training data for Transformer and BERT.

## VII. Members' Contribution

Name	Student ID	Task
Phạm Văn Cường	20194421	<ul style="list-style-type: none"><li>- Preprocess data</li><li>- RNN models</li></ul>
Nguyễn Việt Hoàng	20194434	<ul style="list-style-type: none"><li>- Preprocess data for Transformer</li><li>- Transformer models</li></ul>
Hoàng Văn Khánh	20194440	<ul style="list-style-type: none"><li>- Ensemble models</li></ul>
Phan Đức Thắng	20194452	<ul style="list-style-type: none"><li>- Data overview</li><li>- LSTM models</li></ul>
Phạm Việt Thành	20194454	<ul style="list-style-type: none"><li>- BERT Models</li><li>- Word segmentation implementation</li></ul>

## VIII. References

- [UIT-VSFC: Vietnamese Students' Feedback Corpus for Sentiment Analysis](#)
- [A Simple and Efficient Ensemble Classifier Combining Multiple Neural Network Models on Social Media Datasets in Vietnamese](#)
- [Recurrent Neural Networks cheatsheet - CS 230 - Deep Learning - Stanford](#)
- [Fundamentals of Recurrent Neural Network \(RNN\) and Long Short-Term Memory \(LSTM\) Network](#)
- <https://www.coursera.org/learn/nlp-sequence-models>
- [\[1706.03762\] Attention Is All You Need \(arxiv.org\)](#)
- [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#)
- [RoBERTa: A Robustly Optimized BERT Pretraining Approach](#)
- [PhoBERT: Pre-trained language models for Vietnamese](#)