VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT
SENTIMENT ANALYSIS
SEMESTER 231 2023-2024

Advisor: Quan Thanh Tho

Students: Nguyen Phuoc Nguyen Phuc 2053342

Ho Chi Minh city, November 2023

Contents

List of Figures

1 CNN FOR SEMTIMENT ANALYSIS

1.1 CNN for Image Processing

CNNs belong to a lineage of models initially inspired by the functioning of the human brain's visual cortex during object recognition. Effectively extracting pertinent features stands as a pivotal element in the efficacy of any machine learning algorithm. Traditional models in machine learning rely on input features derived from either domain expertise or computational methods for feature extraction.

Specific types of neural networks, like CNNs, possess the ability to autonomously learn valuable features directly from raw data, tailoring them to suit specific tasks. Hence, it's customary to view CNN layers as feature extractors. The initial layers, situated just after the input layer, capture foundational features from the raw data, while subsequent layers (often fully connected, akin to those in a multilayer perceptron) utilize these features to forecast continuous target values or class labels.

Certain multilayer neural networks, especially deep CNNs, establish what's known as a "feature hierarchy" by amalgamating low-level features in a step-by-step manner to create high-level features. In scenarios involving images, the earlier layers extract rudimentary features like edges and blobs, which are then amalgamated to generate more sophisticated, high-level features. These high-level features can encompass intricate structures such as the overall outlines of objects like buildings, cats, or dogs.

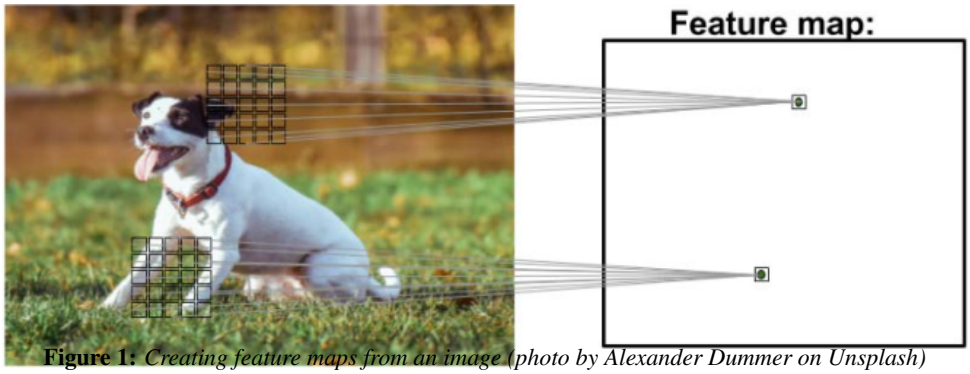

Figure 1: *Creating feature maps from an image (photo by Alexander Dummer on Unsplash)*

This local patch of pixels is referred to as the local receptive field. CNNs will usually perform very well on image-related tasks, and that's largely due to two important ideas:

- Sparse connectivity: A single element in the feature map is connected to only a small patch of pixels.
- Parameter sharing: The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights (parameters) in the network, and we will see an improvement in the ability to capture salient features. In the context of image data, it makes sense to assume that nearby pixels are typically more relevant to each other than pixels that are far away from each other.

1.2 CNN for Natural Language Processing

Convolutional Neural Networks (CNNs), although originally designed for image processing, have found applicability in Natural Language Processing (NLP) tasks as well. In NLP, CNNs can operate on text data by treating it as a 1D signal, breaking it down into smaller chunks to extract local features. By using multiple convolutional filters of different sizes, CNNs can capture varying word or character-level patterns within the text. These filters slide across the input text, detecting relevant features that contribute to understanding semantics, syntax, or sentiment. Through this process, CNNs can effectively perform tasks like text classification, sentiment analysis, and even sequence modeling by leveraging their ability to identify patterns in textual data.

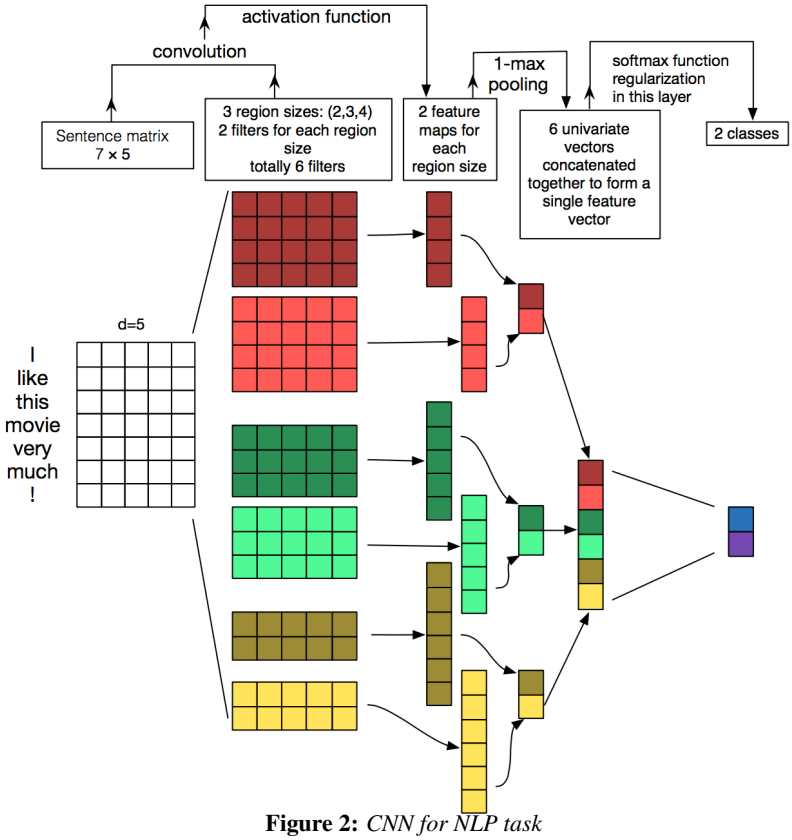

Figure 2: *CNN for NLP task*

When using Convolutional Neural Networks (CNNs) for Natural Language Processing (NLP) tasks, the fundamental architecture and process differ from their typical use in image processing:

1. **Input Representation:**
   - **Word Embeddings**: Text input is first transformed into word embeddings like Word2Vec, GloVe, or embeddings learned during training.
   - **Padding:** Sentences are often of varying lengths, so they're padded or truncated to a fixed length to ensure uniformity.
2. **Convolutional Layers:**
   - **Convolution:** In NLP, 1D convolutions are used because text data is treated as a sequence. Convolutional filters slide across the word embeddings to extract features.
   - **Filters:** Multiple filters of different widths are applied to capture various n-gram features. For instance, a filter might cover 2, 3, or 4 consecutive words.
   - **Feature Maps:** These filters produce feature maps by computing dot products between the filter and local patches of the input sequence.
3. **Pooling Layers:**
   - **Pooling:** Max pooling or average pooling is performed over each feature map, reducing its dimensionality while retaining the most important information. This process captures the most salient features within each filter.
4. **Fully Connected Layers:**
   - **Flattening:** The pooled feature maps are flattened into a single vector.
   - **Dense Layers:** These flattened vectors are then passed through fully connected layers for classification or regression tasks.

5. **Output Layer:**
    - **Activation:** A softmax or sigmoid activation function is applied depending on the nature of the task (classification, sentiment analysis, etc.).

1.3 Experiment

Source code: here

1.3.1 Dataset:

Sentiment datasets used in the VLSP competition. The dataset is loaded then processing (tokenization, split). Train set has 5100 documents with theirs coresponding labels, test set has 1050 documents.

1.3.2 Model Architecture

1. **Hyperparameter:**
    - EMBEDDING_DIM = 400
    - MAX_VOCABULARY = 1000
    - MAX_SEQUENCE_LENGTH = 300
    - filter length = [2,3,4]
    - num_filters = 128
    - epoch = 12
    - batch_size = 128
2. **Model:**

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	[(None, 300)]	0	[]
embedding_1 (Embedding)	(None, 300, 400)	3167600	['input_6[0][0]']
conv1d_15 (Conv1D)	(None, 299, 128)	102528	['embedding_1[2][0]']
conv1d_16 (Conv1D)	(None, 297, 128)	204928	['embedding_1[2][0]']
conv1d_17 (Conv1D)	(None, 298, 128)	153728	['embedding_1[2][0]']
max_pooling1d_15 (MaxPooling1D)	(None, 1, 128)	0	['conv1d_15[0][0]']
max_pooling1d_16 (MaxPooling1D)	(None, 1, 128)	0	['conv1d_16[0][0]']
max_pooling1d_17 (MaxPooling1D)	(None, 1, 128)	0	['conv1d_17[0][0]']
concatenate_5 (Concatenate)	(None, 3, 128)	0	['max_pooling1d_15[0][0]', 'max_pooling1d_16[0][0]', 'max_pooling1d_17[0][0]']
flatten_5 (Flatten)	(None, 384)	0	['concatenate_5[0][0]']
dropout_5 (Dropout)	(None, 384)	0	['flatten_5[0][0]']
dense_5 (Dense)	(None, 3)	1155	['dropout_5[0][0]']

Total params: 3629939 (13.85 MB)
Trainable params: 3629939 (13.85 MB)
Non-trainable params: 0 (0.00 Byte)

Figure 3: *Model Architecture*

3. **Result:** After tuning hyperparameter + adjusting layer's dimensions. I reach the best result on training set is **98,21%**, on test set is **66,67%**. Approximately **1,5%** higher than the teacher's result.

```
Epoch 1/12
32/32 [==============================] - 2s 29ms/step - loss: 7.3590 - accuracy: 0.5213 - val_loss: 6.3687 - val_accuracy: 0.4539
Epoch 2/12
32/32 [==============================] - 1s 22ms/step - loss: 5.1494 - accuracy: 0.7721 - val_loss: 6.4383 - val_accuracy: 0.1667
Epoch 3/12
32/32 [==============================] - 1s 22ms/step - loss: 4.0167 - accuracy: 0.8672 - val_loss: 5.6400 - val_accuracy: 0.1363
Epoch 4/12
32/32 [==============================] - 1s 22ms/step - loss: 3.1900 - accuracy: 0.9191 - val_loss: 5.2399 - val_accuracy: 0.0931
Epoch 5/12
32/32 [==============================] - 1s 22ms/step - loss: 2.5430 - accuracy: 0.9458 - val_loss: 4.7439 - val_accuracy: 0.0686
Epoch 6/12
32/32 [==============================] - 1s 22ms/step - loss: 2.0305 - accuracy: 0.9662 - val_loss: 4.0602 - val_accuracy: 0.1078
Epoch 7/12
32/32 [==============================] - 1s 22ms/step - loss: 1.6442 - accuracy: 0.9728 - val_loss: 3.8106 - val_accuracy: 0.0902
Epoch 8/12
32/32 [==============================] - 1s 22ms/step - loss: 1.3393 - accuracy: 0.9757 - val_loss: 3.2151 - val_accuracy: 0.1588
Epoch 9/12
32/32 [==============================] - 1s 22ms/step - loss: 1.1152 - accuracy: 0.9760 - val_loss: 3.2227 - val_accuracy: 0.1088
Epoch 10/12
32/32 [==============================] - 1s 21ms/step - loss: 0.9309 - accuracy: 0.9855 - val_loss: 3.2857 - val_accuracy: 0.0735
Epoch 11/12
32/32 [==============================] - 1s 22ms/step - loss: 0.7931 - accuracy: 0.9828 - val_loss: 3.0969 - val_accuracy: 0.0814
Epoch 12/12
32/32 [==============================] - 1s 22ms/step - loss: 0.6939 - accuracy: 0.9821 - val_loss: 2.3903 - val_accuracy: 0.2147
<keras.src.callbacks.History at 0x78bd2cb39780>
```

Figure 4: *Training Process*

```
33/33 [==============================] - 0s 4ms/step - loss: 1.3689 - accuracy: 0.6676
```

Figure 5: *Training Process*

2 RECURRENT NEURAL NETWORK

2.1 What is RNN:

Recurrent Neural Networks (RNNs) are a specialized class of artificial neural networks designed to handle sequential data, where the order of information matters. This includes sequences like text, audio, video, time series data, and more. Unlike traditional feedforward neural networks, RNNs possess a memory element that enables them to retain information about previous inputs in the sequence. This memory aspect allows RNNs to exhibit temporal dynamic behavior, making them well-suited for tasks involving sequences or time series data.

In the context of text, RNNs excel in understanding and processing sequences of words due to their ability to consider the context of each word based on the ones that came before it. This contextual understanding is crucial in natural language processing tasks like language translation, sentiment analysis, text generation, and speech recognition. For instance, in language translation, RNNs can retain information about preceding words to determine the appropriate translation for a given word in the sequence, accounting for the nuances of grammar and context.

The suitability of RNNs for these tasks lies in their recurrent nature, which allows them to maintain a memory of past inputs and utilize this information to make decisions at each step in the sequence. This capability is particularly valuable in scenarios where the current output not only depends on the current input but also on the entire history of inputs leading up to that point. This makes RNNs powerful tools for modeling and making predictions in sequential data domains where understanding context and temporal dependencies are critical.

Recurrent Neural Networks (RNNs) operate by processing sequential data through a series of interconnected nodes, each possessing a hidden state that retains information from previous steps. At each time step, the RNN takes an input and combines it with the hidden state from the previous step, capturing the context and information flow across the sequence. This integration of the current input and the previous hidden state allows RNNs to maintain a form of memory. The hidden layer's output from the prior step becomes an input to the current step, effectively incorporating historical context into the present computation. This recurrent structure enables RNNs to capture temporal dependencies and learn patterns in sequential data by continually updating and refining the hidden states through the sequence.

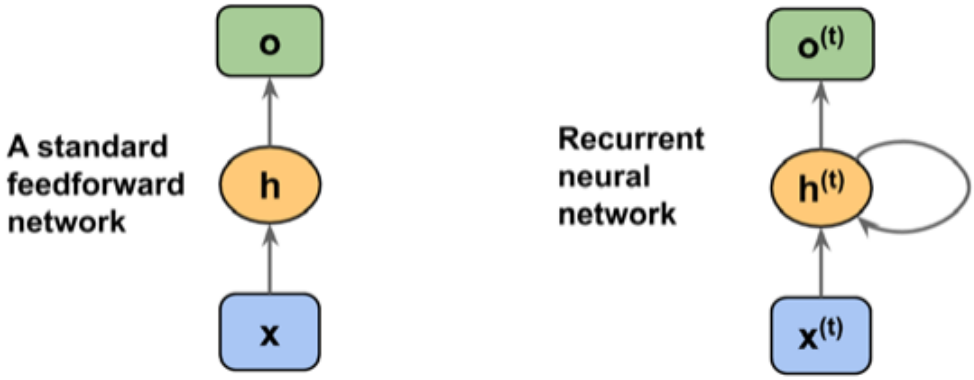

Figure 6: *The dataflow of a standard feedforward NN and an RNN*

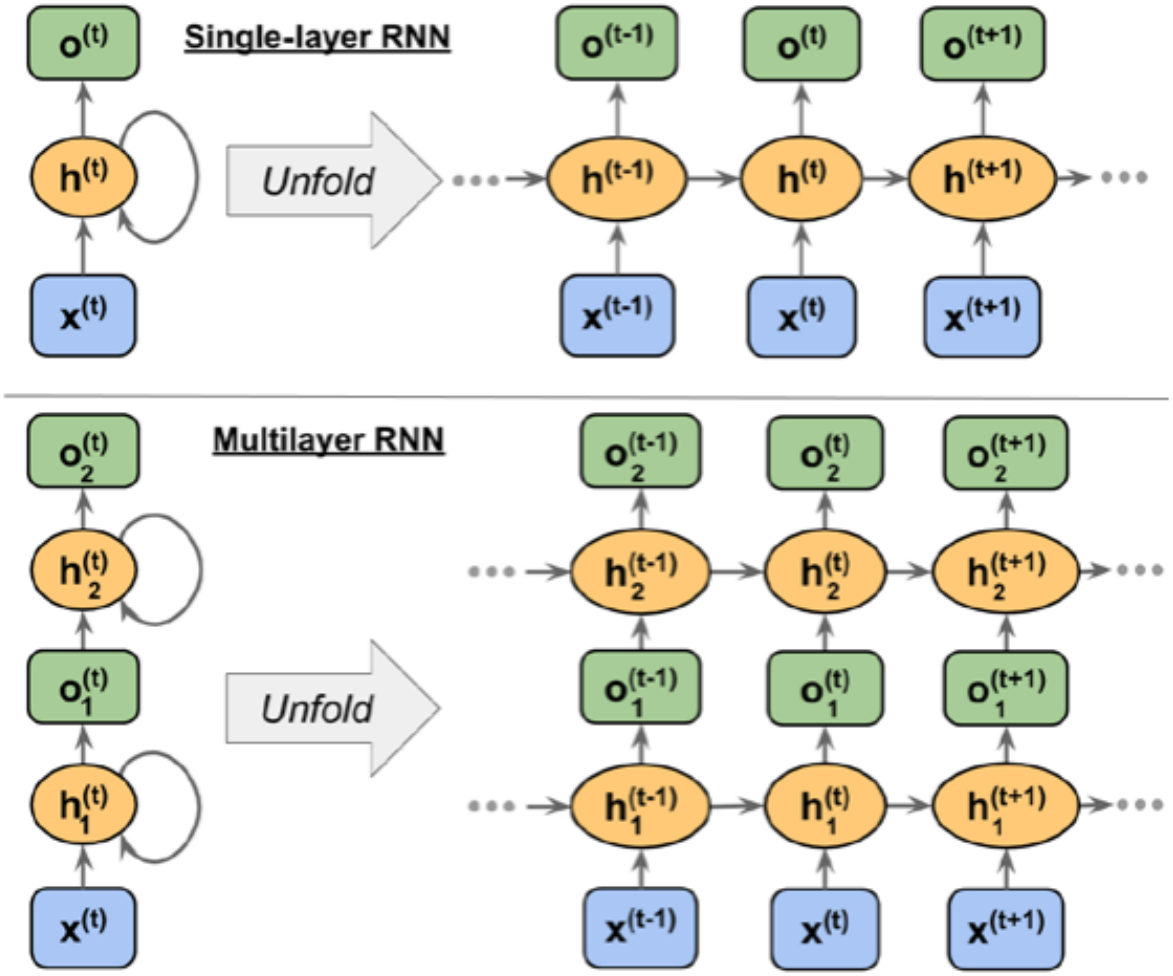

Figure 7: *Examples of an RNN with one and two hidden layers*

2.2 LSTM:

Long Short-Term Memory networks (LSTMs) offer a solution to the vanishing and exploding gradient problems encountered in traditional Recurrent Neural Networks (RNNs). While RNNs are adept at handling sequential data, they often struggle to retain long-term dependencies due to their inherent difficulties in remembering information over extended sequences. LSTMs introduce a more sophisticated memory mechanism that consists of gates—input, forget, and output gates—enabling them to selectively retain or forget information over varying time scales. This design allows LSTMs to mitigate the vanishing gradient problem by regulating the flow of information, thereby facilitating the learning of long-term dependencies in sequences. Additionally, LSTMs address the exploding gradient issue by employing these gates to control the magnitude of gradients during training, ensuring more stable and effective learning compared to conventional RNNs. As a result, LSTMs have become a preferred choice for modeling sequential data, especially when capturing and leveraging long-range dependencies is crucial, such as in natural language processing, time series analysis, and various other sequential data tasks.

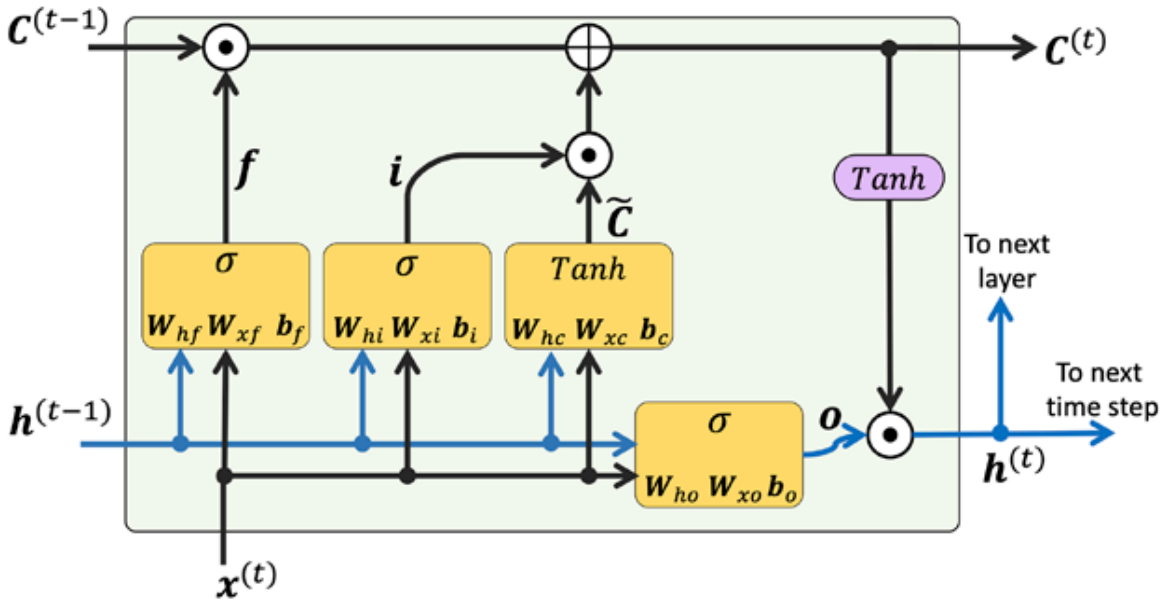

Figure 8: *The structure of an LSTM cell*

2.3 Experiment:

Source code: here

1. **Hyperparameter:**
   - EMBEDDING_DIM = 400
   - MAX_VOCABULARY = 1000
   - MAX_SEQUENCE_LENGTH = 300
   - epoch = 5
   - batch_size = 128
2. **Model:**

```
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_15 (InputLayer)       [(None, 300)]             0

 embedding (Embedding)       (None, 300, 400)          3167600

 reshape_61 (Reshape)        (None, 300, 400)          0

 lstm_19 (LSTM)              (None, 512)               1869824

 dropout_11 (Dropout)        (None, 512)               0

 dense_11 (Dense)            (None, 3)                 1539

=================================================================
Total params: 5038963 (19.22 MB)
Trainable params: 5038963 (19.22 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 9: *Model Architecture*

3. **Result:** After tuning hyperparameter + adjusting layer's dimensions. I reach the best result on training set is **99,66%**, on test set is **63,24%**. Approximately **1,5%** higher than the teacher's result.

```
Epoch 1/10
16/16 [==============================] - 3s 158ms/step - loss: 0.2559 - accuracy: 0.9321 - val_loss: 2.7571 - val_accuracy: 0.1961
Epoch 2/10
16/16 [==============================] - 2s 154ms/step - loss: 0.1703 - accuracy: 0.9657 - val_loss: 3.3655 - val_accuracy: 0.2167
Epoch 3/10
16/16 [==============================] - 2s 155ms/step - loss: 0.1785 - accuracy: 0.9640 - val_loss: 2.6607 - val_accuracy: 0.2010
Epoch 4/10
16/16 [==============================] - 2s 141ms/step - loss: 0.1590 - accuracy: 0.9718 - val_loss: 3.8850 - val_accuracy: 0.1255
Epoch 5/10
16/16 [==============================] - 2s 140ms/step - loss: 0.1062 - accuracy: 0.9873 - val_loss: 3.5496 - val_accuracy: 0.2196
Epoch 6/10
16/16 [==============================] - 2s 140ms/step - loss: 0.0888 - accuracy: 0.9922 - val_loss: 4.0012 - val_accuracy: 0.1647
Epoch 7/10
16/16 [==============================] - 2s 140ms/step - loss: 0.0691 - accuracy: 0.9966 - val_loss: 4.2684 - val_accuracy: 0.1696
Epoch 7: early stopping
<keras.src.callbacks.History at 0x7ce39b472da0>
```

Figure 10: *Training Process*

```
33/33 [==============================] - 0s 14ms/step - loss: 1.5810 - accuracy: 0.6324
```

Figure 11: *Training Process*

3 CNN + RNN = RCNN

3.1 What is RCNN:

The combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) represents a powerful approach in handling sequential data, particularly in tasks involving both spatial and temporal dependencies. CNNs excel in extracting hierarchical spatial features from fixed-size inputs like images, capturing patterns at different levels through convolutional layers and pooling operations. On the other hand, RNNs are proficient in modeling sequential data by considering temporal dependencies over variable-length sequences. Combining these two architectures involves using CNNs to extract relevant spatial features from each step of the sequence and then passing these features to an RNN, which understands the sequential context and temporal relationships among these features. This integration enables the network to effectively process both spatial and temporal information, making it well-suited for tasks like video analysis, action recognition, and image captioning where understanding both spatial patterns and their temporal evolution is crucial.

The advantage of combining CNNs with RNNs lies in the synergistic utilization of their strengths. CNNs excel in feature extraction from spatial data, providing rich representations of each step in the sequence. These extracted features serve as meaningful inputs to the RNN, allowing it to focus on learning temporal dependencies and patterns across these features. This collaboration enables the network to capture both local spatial details and long-range temporal dependencies, improving the model's ability to comprehend complex sequential data. Additionally, this fusion often leads to enhanced performance in various tasks by leveraging the hierarchical and contextual information learned by CNNs and RNNs, respectively, providing a more comprehensive understanding of the input data.

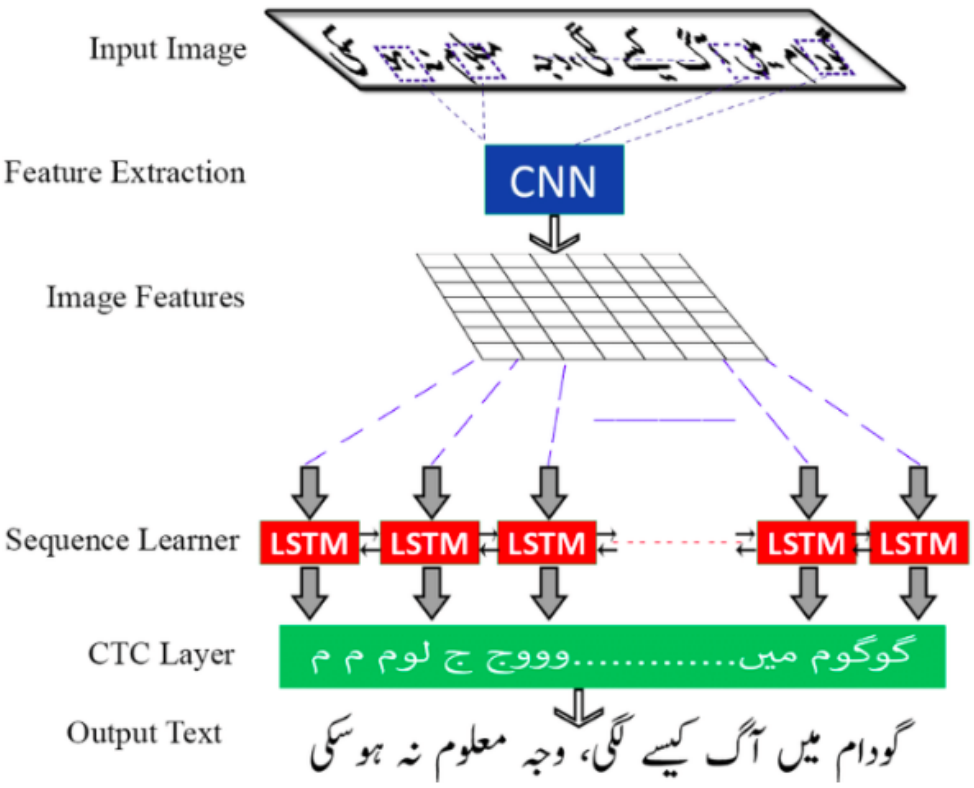

Figure 12: *CRNN Architecture*

3.2 Experiment:

Source code: here
I did alot of experiment and parameters/hyperparameters tuning to improve the result.

1. **Hyperparameter:**
    - EMBEDDING_DIM = 400

- MAX_VOCABULARY = 1000
- MAX_SEQUENCE_LENGTH = 300
- epoch = 9
- batch_size = 128
- filter_sizes = [1,2,3,4]
- num_filters = 512
- drop = 0.46

2. **Model:**

```
Model: "model_3"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_4 (InputLayer)        [(None, 300)]             0         []

 embedding_3 (Embedding)     (None, 300, 400)          3167600   ['input_4[0][0]']

 reshape_15 (Reshape)        (None, 300, 400)          0         ['embedding_3[0][0]']

 conv1d_12 (Conv1D)          (None, 300, 512)          205312    ['reshape_15[0][0]']

 conv1d_13 (Conv1D)          (None, 300, 512)          410112    ['reshape_15[0][0]']

 conv1d_14 (Conv1D)          (None, 300, 512)          614912    ['reshape_15[0][0]']

 conv1d_15 (Conv1D)          (None, 300, 512)          614912    ['reshape_15[0][0]']

 max_pooling1d_12 (MaxPooli  (None, 1, 512)            0         ['conv1d_12[0][0]']
 ng1D)

 max_pooling1d_13 (MaxPooli  (None, 1, 512)            0         ['conv1d_13[0][0]']
 ng1D)

 max_pooling1d_14 (MaxPooli  (None, 1, 512)            0         ['conv1d_14[0][0]']
 ng1D)

 max_pooling1d_15 (MaxPooli  (None, 1, 512)            0         ['conv1d_15[0][0]']
 ng1D)
```

Figure 13: *Model Architecture*

reshape_16 (Reshape)	(None, 1, 512)	0	['max_pooling1d_12[0][0]']
reshape_17 (Reshape)	(None, 1, 512)	0	['max_pooling1d_13[0][0]']
reshape_18 (Reshape)	(None, 1, 512)	0	['max_pooling1d_14[0][0]']
reshape_19 (Reshape)	(None, 1, 512)	0	['max_pooling1d_15[0][0]']
concatenate_3 (Concatenate)	(None, 1, 2048)	0	['reshape_16[0][0]', 'reshape_17[0][0]', 'reshape_18[0][0]', 'reshape_19[0][0]']
lstm_3 (LSTM)	(None, 256)	2360320	['concatenate_3[0][0]']
dropout_3 (Dropout)	(None, 256)	0	['lstm_3[0][0]']
dense_3 (Dense)	(None, 3)	771	['dropout_3[0][0]']
flatten_3 (Flatten)	(None, 3)	0	['dense_3[0][0]']

==========

Total params: 7373939 (28.13 MB)
Trainable params: 7373939 (28.13 MB)
Non-trainable params: 0 (0.00 Byte)

Figure 14: *Model Architecture*

3. **Result:** After tuning hyperparameter + adjusting layer's dimensions. I reach the best result on training set is **88,04%**, on test set is **71,33%**. Approximately **9%** higher than the teacher's result.

```
36/36 [==============================] - 3s 79ms/step - loss: 0.8207 - accuracy: 0.8708 - val_loss: 1.7651 - val_accuracy: 0.4255
33/33 [==============================] - 0s 8ms/step - loss: 1.2726 - accuracy: 0.6848
36/36 [==============================] - 3s 77ms/step - loss: 0.8184 - accuracy: 0.8804 - val_loss: 1.3136 - val_accuracy: 0.6569
33/33 [==============================] - 0s 8ms/step - loss: 1.2547 - accuracy: 0.7133
36/36 [==============================] - 3s 78ms/step - loss: 0.7311 - accuracy: 0.9022 - val_loss: 2.0737 - val_accuracy: 0.3765
33/33 [==============================] - 0s 8ms/step - loss: 1.3114 - accuracy: 0.7010
36/36 [==============================] - 3s 82ms/step - loss: 0.7224 - accuracy: 0.9035 - val_loss: 2.1126 - val_accuracy: 0.3608
33/33 [==============================] - 0s 8ms/step - loss: 1.3644 - accuracy: 0.6695
36/36 [==============================] - 3s 79ms/step - loss: 0.6812 - accuracy: 0.9211 - val_loss: 2.1885 - val_accuracy: 0.3686
33/33 [==============================] - 0s 8ms/step - loss: 1.4907 - accuracy: 0.6600
36/36 [==============================] - 3s 78ms/step - loss: 0.6925 - accuracy: 0.9174 - val_loss: 1.3179 - val_accuracy: 0.6627
33/33 [==============================] - 0s 9ms/step - loss: 1.3222 - accuracy: 0.6943
36/36 [==============================] - 3s 83ms/step - loss: 0.6769 - accuracy: 0.9264 - val_loss: 2.1994 - val_accuracy: 0.3647
33/33 [==============================] - 0s 9ms/step - loss: 1.3764 - accuracy: 0.6867
36/36 [==============================] - 3s 78ms/step - loss: 0.6014 - accuracy: 0.9364 - val_loss: 2.2514 - val_accuracy: 0.3706
33/33 [==============================] - 0s 8ms/step - loss: 1.4594 - accuracy: 0.6552
36/36 [==============================] - 3s 77ms/step - loss: 0.5680 - accuracy: 0.9501 - val_loss: 2.8194 - val_accuracy: 0.2647
33/33 [==============================] - 0s 8ms/step - loss: 1.4809 - accuracy: 0.6695
36/36 [==============================] - 3s 77ms/step - loss: 0.5712 - accuracy: 0.9397 - val_loss: 2.4933 - val_accuracy: 0.3216
33/33 [==============================] - 0s 9ms/step - loss: 1.4683 - accuracy: 0.6705
36/36 [==============================] - 3s 82ms/step - loss: 0.5272 - accuracy: 0.9532 - val_loss: 2.4461 - val_accuracy: 0.3922
33/33 [==============================] - 0s 8ms/step - loss: 1.4240 - accuracy: 0.6771
```

Figure 15: *Training Process + Test Result*

4 PreTrain PhoBert - Vietnamese Based Sentiment

4.1 What is PhoBert

A model fine-tuned for sentiment analysis based on vinai/phobert-base.

Pre-trained PhoBERT models are the state-of-the-art language models for Vietnamese (Pho, i.e. ”Ph”, is a popular food in Vietnam):

- Two PhoBERT versions of ”base” and ”large” are the first public large-scale monolingual language models pre-trained for Vietnamese. PhoBERT pre-training approach is based on RoBERTa which optimizes the BERT pre-training procedure for more robust performance.
- PhoBERT outperforms previous monolingual and multilingual approaches, obtaining new state-of-the-art performances on four downstream Vietnamese NLP tasks of Part-of-speech tagging, Dependency parsing, Named-entity recognition and Natural language inference.

The general architecture and experimental results of PhoBERT can be found in EMNLP-2020 Findings paper

4.2 Experiment

Source code: here

I re-use all the architecture from the pre-train model, include hyperparameters, params,...

This model can classify a sentence into 3 categories: negative, neutral and positive. How ever in our data set. There is only 2 sentimental labels: negative and positive. Therefore, I will pop out test data with is classified as ”neutral” and only compare the result on ”negative” and ”positive”.

Result Result when using pre-train PhoBert for this dataset is 0.5953654188948306. Which is slightly small then previous models. How ever, i believe this is due to the overfitting of previous model

```
<bound method Module.get_parameter of RobertaForSequenceClassification(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(64001, 768, padding_idx=1)
      (position_embeddings): Embedding(258, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-11): 12 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): RobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=3, bias=True)
  )
)>
```

Figure 16: *Model Architecture*

5 COMPARE & CONCLUSION

After experimenting on CNN, RNN and CRNN. Choosing the best result on each model. We can see CRNN out perform two others with **71,33%**. RNN and CNN get equal result.

This makes sense because CRNN take both CNN and RNN strength. The model can learn deeply the structure of the training set.

	#Params	Test Accuracy
CNN	3,6M	66,76%
RNN	5M	63,24%
CRNN	7,37M	71,33%
PhoBert	550M	59,54%

Table 1: *Result on 3 models*