**Attn: Shafiq Joty (Asst. Prof)**



**CE/CZ4045 Natural Language Processing**

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below. We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work. We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work.

|  |  |
| --- | --- |
| Name | Signature / Date |
| Chulpaibul Jiraporn | / 28 November 2020 |
| Leong Ko Rixie Tiffany | A picture containing shape  Description automatically generated / 28 November 2020 |
| Leong Kai Ling | /28 November 2020 |
| Liew Zhi Li | A picture containing text  Description automatically generated /28 November 2020 |

Important note:

Name must **EXACTLY MATCH** the one printed on your Matriculation Card. Any mismatch leads to **THREE (3)** marks deduction.

 CE/CZ4045: Assignment 2

Chulpaibul Jiraporn, Leong Ko Rixie Tiffany, Leong Kai Ling, Liew Zhi Li

School of Computer Science and Engineering

Nanyang Technological University

Singapore

{CHUL0004, RLEONG007, KLEONG020, LIEW0177}@e.ntu.edu.sg

# ABSTRACT

In this report, we will summarize the findings and analysis from our hands-on experience of creating different neural models to solve Natural Language Processing (NLP) tasks. The report consists of two parts: (i) Word-based Language Model (LM), (ii) Named Entity Recognition (NER) Model.

# KEYWORDS

Natural Language Processing, Language Model, Named Entity Recognition, Neural Model

# 1 Word-based Language Model (LM)

A language model predicts the probability of the next word in a sequence based on the previous words. In this experiment, we used the wikitext-2 dataset to build an 8-gram language model.

## 1.1 Text Preparation for Word-based LM

The corpus is partitioned into 3 sets — train set, validation set and test set.

To prepare the corpus for training the model, we perform the following steps:

1. Create a dictionary for the corpus.
   1. The corpus is tokenized to obtain the tokens.
   2. Each token in the corpus is assigned to an unique numerical ID (index). This index will be used as the input and output to the neural network, instead of the actual tokens. By using indexes, we can employ matrix operations inside the neural network to speed up computation.
   3. A dictionary consists of
      1. Word to index – Returns the index of the given word
      2. Index to word – Returns the word of the given index
2. Divide the datasets into columns.
   1. The datasets are divided into columns.
   2. The number of columns is equal to the number of mini batches.
   3. The columns are treated as independent from one another, so the model cannot learn dependencies between the columns.
   4. However, it allows for more efficient processing as the columns can be processed in parallel.
3. Generate 8-grams.
   1. During training and evaluation, the columns will be swept from top to bottom to obtain the 8-grams.
   2. Hence, at each step, the model will receive one 8-gram from each column to form one mini batch.

The final structure of training data is shown in Figure 1. We do the same for validation and test set with fewer columns since there is less data in those sets. Each cell contains the index of the token as its value.

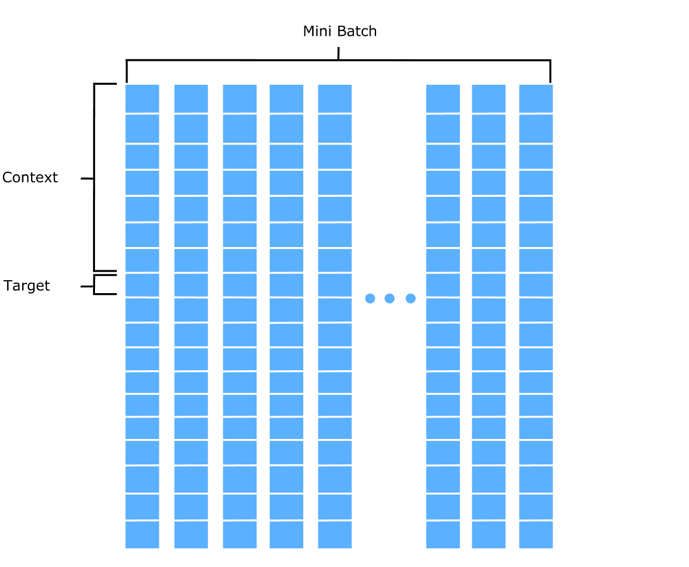


Figure 1: Illustration of how the data is structured.

## 1.2 Design and Training of Neural LM

### 1.2.1 Architecture

The probabilistic neural language model consists of three layers — embedding layer, hidden layer and output layer (Table 1). The embedding layer uses word embeddings to represent each word (continuous and dense representation). The hidden layer is used to extract features from the word embeddings. The output layer generates the probabilities of the possible next words in a sequences based on the context (the previous 7 words).

|  |  |  |
| --- | --- | --- |
|  | **Number of Weights** | **Activation Function** |
| **Embedding Layer** | |V| x emsize | - |
| **Hidden Layer** | 7 x emsize x nhid | tanh |
| **Output Layer** | nhid x |V| | softmax |

Table 1: Architecture of the neural language model.

With reference to Figure 1, an 8-gram is generated by obtaining a slice of 8 tensors in each column where the first 7 rows of the slice represent n-1 contiguous sequence of words before the target word and the last row represents the target word. Hence, the input is 2-dimensional tensors of dimension (7, batch size) where each tensor contains the index of the corresponding word in the dictionary. For simplicity, we assume that the batch size is 1 so the input is (7 x 1). The embedding layer converts the word indexes in the input to its corresponding word embedding. There is one word embedding for each word in the vocabulary. Hence, the dimension of embedding layer is |V| x emsize, where |V| is the vocabulary size and nhid is the hidden layer size. The word embeddings of the 7 context words are concatenated before feeding it to the hidden layer. The hidden layer consists of a linear layer and Tanh activation function. Thus, the hidden layer has a dimension of 7 x emsize x nhid. The hidden layer output is the input to the output/decoder layer. The output layer has a Softmax activation function which produces the probabilities of each word in the vocabulary appearing as the target word, given the previous 7 words (context). In the code, our batch size is 256, but the idea is the same.

### 1.2.2 Training

For training, we decided to use Mini Batch Gradient Descent. Batch Gradient Descent is expensive as it computes gradient of the entire dataset at each update step. Thus, we use Mini Batch Gradient Descent to update parameters after each mini batch. In our code, we use a mini-batch size of 256.

Learning rate annealing is used to help the model reach a local minimum. The learning rate is reduced by 4 times if there is no improvement in performance on the validation set.

### 1.2.3 Selecting Hyperparameters

The hyperparameters that we need to adjust are emsize (embedding size), nhid (hidden layer size) and number of epoches. Based on our experiment, the loss is usually stable after a few epochs. For emsize and nhid, we trained the model with varying emsize and nhid with 15 epochs and choose the hyperparameters that give the best validation loss (lowest). For the shared weights model, the emsize and nhid are the same as the input and output weights are tied. Figure 2 and 3 show the best validation loss after 15 epochs when the input and output weights are not shared and shared respectively. Hence, we chose emsize = 100 and nhid = 100 for the unshared weight model and we chose emsize = 100 and nhid = 100 for the shared weight model. In addition, it is evident that increasing nhid has a greater impact on the performance than increasing emsize (e.g. emsize = 10 and nhid = 100 performed better than emsize = 100 and nhid = 10).

Application

Description automatically generated

Figure 2: Validation loss of hyperparameters tuning (tie\_weights = False) .

A picture containing application

Description automatically generated

Figure 3: Validation loss of hyperparameters tuning (tie\_weights = True)

### 1.2.4 Training Results

Figure 4 and 5 show the train and validation loss over 20 epochs. We can see that train loss is slightly lower than validation loss but we are not overfitting our models because both train and validation loss are decreasing. The parameters of the model with lowest validation loss is saved to be used for text generation. The perplexity scores on test data show that the model with sharing weights (lookup embeddings and output) performed better than the model with no sharing weights.

|  |  |  |
| --- | --- | --- |
|  | **Test loss** | **Test Perplexity** |
| **Tie\_weights = False** | 5.50 | 243.95 |
| **Tie\_weights = True** | 5.24 | 187.98 |

Table 2: Test loss and perplexity after training.

Chart, line chart

Description automatically generated

Figure 4: Loss for 20 epochs (tie\_weight = False)

Chart, line chart

Description automatically generated

Figure 5: Loss for 20 epochs (tie\_weight = True)

### 1.3 Text Generation Using the Trained LM

After we trained the model, we used the best model to generate new text. The input to the model is a 1-dimension tensor of length 7 containing the phrase “He announced he was retiring from professional ”.

At each time step i, a token will be sampled from distribution of Softmax output of the model, then that token will be used to generate the future tokens by appending to the 6 tokens from timestep i-6 to i-1. The generated text was evaluated based on its perplexity as shown in Table 3. We can observed that the model with shared weights outperformed the model without shared weights again. However, the text produced by both models were not very meaningful or grammatical.

|  |  |  |
| --- | --- | --- |
|  | **Generated Text Loss** | **Generated Text Perplexity** |
| **Tie\_weights = False** | 6.88 | 968.48 |
| **Tie\_weights = True** | 6.24 | 513.54 |

Table 3: Loss and perplexity of generated text.

## 1.4 Evaluation of Word Embeddings

### 1.4.1 Cosine Similarity

The cosine similarity is used to measure the similarity between two word embeddings. The cosine similarity is calculated by taking a dot product of the two word embeddings (vectors) followed by normalization . The higher the cosine similarity (closer to 1), the more similar the two words are. The lower the cosine similarity (closer to -1), the more different the two words are. Both models show that the words ‘king’, ‘queen’, ‘man’ and ‘woman’ are more similar to each other than a completely unrelated word ‘computer’ (Table 4 and 5). The cosine similarities show that the shared weights model has better word embeddings than the model without shared weights.

|  |  |  |
| --- | --- | --- |
| **Words** | | **Cosine Similarity** |
| King | Queen | 0.3371352 |
| Man | Woman | 0.53910506 |
| King | Man | 0.2651133 |
| Queen | Woman | 0.17548701 |
| King | Computer | 0.06156189 |
| Queen | Computer | 0.059445146 |

Table 4: Cosine similarities between pairs of words (tie\_weights = False).

|  |  |  |
| --- | --- | --- |
| **Words** | | **Cosine Similarity** |
| King | Queen | 0.66843694 |
| Man | Woman | 0.8269924 |
| King | Man | 0.54473907 |
| Queen | Woman | 0.4724868 |
| King | Computer | 0.042319015 |
| Queen | Computer | 0.078673765 |

Table 5: Cosine similarities between pairs of words (tie\_weights = True).

### 1.4.2 Visualisation of Word Embeddings

We generated the visualisations of the word embeddings by reducing dimensionality using PCA to 2 dimensions and plotting a scatterplot of the words that appear frequently in the text, excluding stop words (Figure 6 and 7). We can see that similar words are nearer together and appear to be in clusters (e.g. numbers, months, time, pronouns, verbs). The visualizations of word embeddings also show that the shared weights model has better word embeddings than the model without shared weights.

Chart, scatter chart

Description automatically generated

Figure 6: Visualization of word embeddings of frequent words (tie\_weight = False)

Chart, scatter chart

Description automatically generated

Figure 7: Visualization of word embeddings of frequent words (tie\_weight = True)

## 1.5 Conclusion

In summary, the model with shared weights outperformed the model without shared weights in both perplexity for test set and text generation. One of the possible reasons is that the shared weights result in better representations of word embeddings.

In addition, for the model without shared weights, we noticed that an increase in the size of the hidden layer has a more significant impact than increasing the size of word embeddings. While we generally increase the performance of the models by increasing both the size of the hidden layer and word embeddings, it results in longer training time as the number of parameters increases and can lead to overfitting. Hence, there is a trade-off between performance and time required. In this experiment, we limited the size of both the hidden layer and word embeddings to 100 to compromise the trade-off.

# 2 Named Entity Recognition (NER) Model

Named Entity Recognition (NER) is an information extraction task which identifies and classifies entities such as persons, locations, time and organisations. In this assignment, we will be developing a neural NER model using Pytorch. The dataset we are using is the English data from the 2003 CoNLL NER dataset. It contains four different types of named entities such as PERSON, LOCATION, ORGANIZATION and MISC and it uses the BIO tagging scheme. The source code is located in the Python notebook file called “Q2-NER.ipynb”.

## 2.1 Code Overview of NER Model

We ensured that the original model that uses the LSTM layer can still be used. As such, we added an additional parameter, parameters['Word\_Mode'] which will determine whether the LSTM or CNN layer will be used for the word-level.

Firstly, we replaced class BiLSTM\_CRF with class NER, which will implement the LSTM or CNN layer depending on the parameters['word\_mode']. For the CNN layer implementation, we created two CNN layers. The first CNN layer:

self.word\_cnn = nn.Conv2d(in\_channels=1, out\_channels=hidden\_dim,

kernel\_size=(3, embedding\_dim+self.out\_channels), padding=(1,0))

Both the LSTM layer and the first CNN layer have the same output dimensions, as stated in out\_channels=hidden\_dim. The kernel\_size=(3, embedding\_dim+self.out\_channels), where the first parameter is the window size (in terms of number of words) and second parameter is the concatenation of the word embedding dimension and character level representation dimension. The in\_channels=1 as the input is a sequence of word vectors and the padding=(1,0).

The subsequent CNN layers share the same parameters. The in\_channels=hidden\_dim as it needs to be the same as the previous CNN output dimension and we changed the kernel\_size to (3, 1).

self.word\_cnn\_next = nn.Conv2d(in\_channels=hidden\_dim, out\_channels=hidden\_dim,

kernel\_size=(3, 1), padding=(1,0))

Next, we added def get\_cnn\_features(…) which is similar to def get\_lstm\_features(…). This function will use the CNN instead of the LSTM to read the embeddings and produce the features. In addition, this function has a new parameter, cnn\_layer\_num, which is an integer value that states how many CNN layers will be used.

The first CNN layer, words\_cnn, reads the embeddings and produce the CNN output, words\_cnn\_out.

words\_cnn\_out = self.word\_cnn(embeds)

When cnn\_layer\_num > 1, the following loop will use the subsequent CNN layer, word\_cnn\_next, that takes in the CNN output as input and produce a new words\_cnn\_out.

for cnn\_layer in range(cnn\_layer\_num-1):

words\_cnn\_out = self.word\_cnn\_next(words\_cnn\_out)

Finally, we reshape the CNN output and prepare it for the last layer.

words\_cnn\_out = words\_cnn\_out.view(self.hidden\_dim, len(sentence))

words\_cnn\_out = words\_cnn\_out.transpose(0, 1)

The rest is the same as get\_lstm\_features(…), where the linear layer converts the output vectors to the tag space and return the word features.

word\_feats = self.hidden2tag(words\_cnn\_out)

return word\_feats

Other code changes include adjusting def get\_neg\_log\_likelihood(…) and def forward\_calc(…) to incorporate the additional parameter: cnn\_layer\_num. In addition, the Training Step has been updated to loop multiple times to train multiple models with different number of CNN layers. At the end of all training, it will plot the training losses and validation F1 score of all models. The plots are saved in the output folder.

### 2.1.1 Sample Sentences

The following are the sample sentences we will be using:

* Joe is the president of USA
* Cathy loves working at Uniqlo
* Jonathan is the USA ambassador for the UN
* Joe likes to eat at McDonald’s
* Mark is one of the founders of Facebook

### 2.1.2 Training of NER Models

We will be trying out NER models with 1, 2, 3, 4-layers CNN. Thus, there will be 4 models in total. For the training of the NER models with different number of CNN layers, we can simply change the following 3 parameter values as indicated in the table below. These parameters can be set in the “Define constants and parameters” cell of the Python notebook.

|  |  |
| --- | --- |
| **Model Type** | **Parameters To Use** |
| Original pre-trained model with LSTM | parameters['word\_mode']="LSTM"  parameters['reload'] = "./models/pre-trained-model"  parameters['cnn\_layer\_num'] = 1 |
| Self-trained model with 1-layer CNN | parameters['word\_mode']="CNN"  parameters['reload'] = "./models/self-trained-model-cnn-layer-num-1"  parameters['cnn\_layer\_num'] = 1 |
| Self-trained model with 2-layers CNN | parameters['word\_mode']="CNN"  parameters['reload'] = "./models/self-trained-model-cnn-layer-num-2"  parameters['cnn\_layer\_num'] = 2 |
| Self-trained model with 3-layers CNN | parameters['word\_mode']="CNN"  parameters['reload'] = "./models/self-trained-model-cnn-layer-num-3"  parameters['cnn\_layer\_num'] = 3 |
| Self-trained model with 4-layers CNN | parameters['word\_mode']="CNN"  parameters['reload'] = "./models/self-trained-model-cnn-layer-num-4"  parameters['cnn\_layer\_num'] = 4 |

Table 6: Parameter values for different number of CNN layers

## 2.2  Comparison of Results with Different Number of CNN Layers

After replacing the LSTM-based word-level encoder with a CNN, we trained the model with one layer of CNN. The training can be found in the ‘Training Step’ cell of the Python notebook. The function “evaluating” is inside the Training Step loop. It calculates the F-1 score and determines the best F-1 score for each of the datasets.

In statistics, the F-score is a measure of a test’s accuracy which is calculated from the precision and recall of the test. The F-1 score is the harmonic mean of the precision and recall. The lowest F-score is 0 which means that either precision or recall 0 and the highest F-score is 1 which means perfect precision and recall.

### 2.2.1 Test Set Results of NER Models with 1, 2, 3, 4-Layers CNN

Once the training is completed (which takes ~3.2 hours for 1 model), it will output the best\_F score value. For the 1-layer CNN, the result outputs the best\_F score = 0.832520325203252.

In the subsequent step, we will train and observe what are the test set best\_F scores for 2,3,4 layers of CNN. The following table shows the best\_F values for all 4 models.

|  |  |
| --- | --- |
| **CNN layer(s)** | **best\_F score** |
| 1-layer CNN | 0.832520325203252 |
| 2-layers CNN | 0.8414928846325964 |
| 3-layers CNN | 0.8442793731316243 |
| 4-layers CNN | 0.8385233324307247 |

Table 7: Comparison of Test Set Results with 1, 2, 3, 4-Layers CNN

Overall, the best\_F scores for all the 4 models are around 0.8 which is close to 1, meaning we achieved very good precision and recall. We observed that as the number of CNN layers increase, the best\_F score increases, with the highest best\_F score = 0.8442793731316243 for the 3-layers CNN. However, when we increase from 3-layers CNN to 4-layers CNN, the best\_F score has dropped.

## 2.2.2 NER Results on Sample Sentences

The sample sentences are in the “Model Testing” cell of the Python notebook, inside the array called “model\_testing\_sentences”. Using the abovementioned sample sentences, the following table shows the NER model results. The results shown is in the format “word : tag”.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Original** | **1-layer CNN** | **2-layers CNN** | **3-layers CNN** | **4-layers CNN** |
| **Sentence 1** | Joe : PER  is : NA  the : NA  president : NA  of : NA  USA : LOC | Joe : PER  is : NA  the : NA  president : NA  of : NA  USA : LOC | Joe : PER  is : NA  the : NA  president : NA  of : NA  USA : LOC | Joe : PER  is : NA  the : NA  president : NA  of : NA  USA : LOC | Joe : PER  is : NA  the : NA  president : NA  of : NA  USA : LOC |
| **Sentence 2** | Cathy : PER  loves : NA  working : NA  at : NA  Uniqlo : LOC | Cathy : PER  loves : NA  working : NA  at : NA  Uniqlo : LOC | Cathy : PER  loves : NA  working : NA  at : NA  Uniqlo : LOC | Cathy : PER  loves : NA  working : NA  at : NA  Uniqlo : LOC | Cathy : PER  loves : NA  working : NA  at : NA  Uniqlo : LOC |
| **Sentence 3** | Jonathan : NA  is : NA  the : NA  USA : LOC  ambassador : NA  for : NA  the : NA  UN : ORG | Jonathan : PER  is : NA  the : NA  USA : LOC  ambassador : NA  for : NA  the : NA  UN : ORG | Jonathan : PER  is : NA  the : NA  USA : LOC  ambassador : NA  for : NA  the : NA  UN : ORG | Jonathan : PER  is : NA  the : NA  USA : LOC  ambassador : NA  for : NA  the : NA  UN : ORG | Jonathan : PER  is : NA  the : NA  USA : LOC  ambassador : NA  for : NA  the : NA  UN : ORG |
| **Sentence 4** | Joe : PER  likes : NA  to : NA  eat : NA  at : NA  McDonald’s : LOC | Joe : PER  likes : NA  to : NA  eat : NA  at : NA  McDonald’s : LOC | Joe : PER  likes : NA  to : NA  eat : NA  at : NA  McDonald’s : LOC | Joe : PER  likes : NA  to : NA  eat : NA  at : NA  McDonald’s : LOC | Joe : PER  likes : NA  to : NA  eat : NA  at : NA  McDonald’s : LOC |
| **Sentence 5** | Mark : PER  is : NA  one : NA  of : NA  the : NA  founders : NA  of : NA  Facebook : MISC | Mark : PER  is : NA  one : NA  of : NA  the : NA  founders : NA  of : NA  Facebook : LOC | Mark : PER  is : NA  one : NA  of : NA  the : NA  founders : NA  of : NA  Facebook : LOC | Mark : PER  is : NA  one : NA  of : NA  the : NA  founders : NA  of : NA  Facebook : LOC | Mark : PER  is : NA  one : NA  of : NA  the : NA  founders : NA  of : NA  Facebook : LOC |

Table 8: Comparison of NER with 1, 2, 3, 4-Layers CNN on Sample Sentences

As observed, for sentence 1 and for all 5 models, Joe has been correctly identified as PER, and USA has been correctly identified as LOC.

For sentence 2 and for all 5 models, Cathy has been correctly identified as PER, and Uniqlo has been correctly identified as LOC.

For sentence 3, for models with 1, 2, 3, 4-layers CNN, it has correctly identified Jonathan as PER, USA as LOC, and UN as ORG. However, for the original model, it did not identify Jonathan as PER, although USA and UN were identified correctly.

For sentence 4, for all 5 models, Joe has been correctly identified as PER and McDonald’s has been correctly identified as LOC.

For sentence 5, for the original model, Mark has been correctly identified as PER, however, Facebook was identified as MISC. Whereas for models with 1, 2, 3, 4-layers CNN, it has correctly identified Mark as PER and Facebook as LOC.

Overall, as all the models with 1, 2, 3, 4-layers CNN has similar best\_F scores, it has produced the exact same results.

## 2.2.3 Comparison of Training Losses of NER Models with 1, 2, 3, 4-Layers CNN

Figure 8 shows plot of a graph with the training losses of all models against the number of iterations with step size = 2000.

A picture containing histogram

Description automatically generated

Figure 8: Plot of Training Losses against No. of Iterations (step = 2000)

The training loss is calculated over the entire training set. It is the value of the cost function that we are minimizing. The Negative Log Likelihood (NLL) is a cost function used as the loss for machine learning models that tells us how bad the model is performing. The lower the NLL, the better the model is performing.

We observed as the number of layers for CNN increases, there are more weights to train, thus it takes longer for the model to converge.

## 2.2.4 Comparison of Validation F1 score of NER Models with 1, 2, 3, 4-Layers CNN

Figure 9 shows plot of the validation F1 score of all models against the number of iterations with step = 14041.

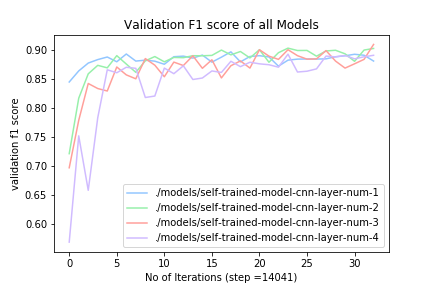


Figure 9: Plot of Validation F-1 scores against No. of Iterations (step = 2000)

We observed that there is an upwards trend. As the number of iterations increases, the validation F-1 scores increases quickly until it converges around 0.855 to 0.9.

The 1-layer CNN model (blue) starts with the highest F-1 score at 0.85 as compared to the other models. We observed that as the number of CNN layers increase, the F-1 score took more iterations to get a good score.

## 2.2.5 Comparison of Validation Losses of NER Models with 1, 2, 3, 4-Layers CNN

As specified by the assignment brief, we will use each model’s validation set’s best F1 score which is “best\_dev\_F” in order to select the best model. The following table shows all the “best\_dev\_F” score of all 4 models.

|  |  |
| --- | --- |
| **CNN layer(s)** | **best\_dev\_F score** |
| 1-layer CNN | 0.8974752644148755 |
| 2-layers CNN | 0.9038858321870701 |
| 3-layers CNN | 0.9102585769815784 |
| 4-layers CNN | 0.8934754797441367 |

Table 9: Comparison of best\_Dev\_F score of 1, 2, 3, 4-Layers CNN Models

We observed that all the 4 models’ F1 score are similar. As the number of CNN layers increase up to 3 layers, the scores improve slightly. When we increase to 4-layers CNN, the score was lowered instead.

In conclusion, the 3-layers CNN is the best model as it has the highest F1 score of validation set of 0.9102585769815784.

# 3  Contributions of Individual Members

|  |  |
| --- | --- |
| Member | Contributions |
| Chulpaibul Jiraporn | Question 1 (Word-based Language Model) |
| Leong Ko Rixie Tiffany | Question 1 (Word-based Language Model) |
| Leong Kai Ling | Question 2 (Named Entity Recognition Model) |
| Liew Zhi Li | Question 2 (Named Entity Recognition Model) |