






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	Contribution	Signature	Date
Chua Peng Shaun U1821442G	20% Question 2		30/11/2020
Ding Si Han U1821300C	20% Question 1		30/11/2020
Gabriel Sze Whye Han U1821069G	20% Question 1		30/11/2020
John Lim Jin U1822862E	20% Question 2		30/11/2020
Tan Chuan Xin U1821755B	20% Question 1		30/11/2020

Important note:

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

CZ4045 Assignment 2 Report

Chua Peng Shaun
PCHUA014@e.ntu.edu.sg

Ding Si Han
DING0111@e.ntu.edu.sg

Gabriel Sze Whye Han
GSZE001@e.ntu.edu.sg

John Lim Jin
JOHN0036@e.ntu.edu.sg

Tan Chuan Xin
CTAN184@e.ntu.edu.sg

1. LANGUAGE MODEL AND TEXT GENERATION

Deep Learning and Neural Networks have revolutionized the field of Natural Language Processing (NLP). Traditional NLP requires extensive domain knowledge and human intervention to craft numerous rules into a language model. Deep learning methods however are significantly easier to create, are far more adaptable, and produce better language models than traditional NLP. This report will build upon the *word_language_model* codebase provided by PyTorch. The codebase contains word language models that utilize RNNs and Transformers. We will seek to create the original FNN model and compare its performance in language modelling and text generation against the theoretically better RNN and Transformer models.

Language models determine the probability of a sequence of tokens in a language, and this can be used to automatically generate texts. The *word_language_model* codebase has already implemented these functionalities.

1.1 Running Original Codebase

We will run the original codebase provided and note the perplexity scores and generated text of the various models. The relevant scripts in the codebase are as follows:

data.py - creates a dictionary and corpus from our data files
generate.py - generates text using trained language model
main.py - driver code to train and save model, and predict
model.py - contains the code for various neural network types

The perplexity score is a measure of how well a language model predicts a given sequence. A lower perplexity score is better.

$$\begin{aligned} PP(W) &= P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \dots w_N)}} \end{aligned}$$

Figure 1. Perplexity score formula

1.1.1 RNN-LSTM

Recurrent Neural Networks are very effective in Natural Language Processing. RNNs are able to pass information about the sequence history through their hidden states. This allows the model to remember information in the temporal dimension, and therefore make better predictions for the next token with context.

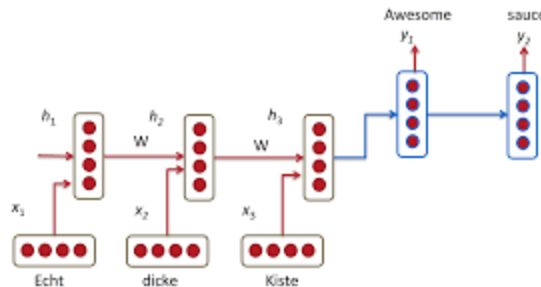


Figure 2. Recurrent Neural Network Architecture

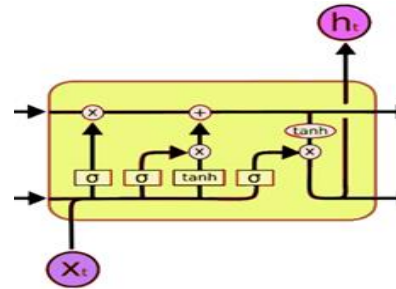


Figure 3. Long Short-Term Memory Cell

However, RNNs typically suffer from the vanishing gradient problem with long input sequences as the gradient is propagated backwards through repeated multiplications with the hidden layer weight matrix. Long Short-Term Memory (LSTM) is a variation of RNN that seeks to resolve this problem. By adding a memory called the cell state on top of the hidden state, it allows the network to recall long term information as well.

Table 1. Perplexity Score for RNN-LSTM

Epoch	Perplexity Score	Validation Loss
1	519.813	6.253
50	155.460	5.035
100	164.433	5.103

1.1.2 Transformer

Transformers are a step up from RNNs in performance for language modelling tasks. The key differentiating factor is the heavy focus on “Attention”, and parallelizability of the computations. RNNs require the input sequence to be fed in time-sequential format, but Transformers allow for parallel processing and encoding of the input. This allows Transformers to be trained much faster than RNNs.

Transformers utilize positional encoders to encode the location of an input token relative to its position in a sequence to adapt the word embedding of the token with temporal dimension information. Subsequently, multi-head attention blocks are used to calculate self-attention, the importance of a word relative to all other words in its proximity. This allows the Transformer to then focus on the correct tokens through learning its context.

Table 2. Perplexity Score for Transformer

Epoch	Perplexity Score	Validation Loss
1	366.778	5.905
50	282.780	5.645
100	318.761	5.764

1.2 Preprocessing and Data Loading

Cleaning the data to be ready as inputs into a model is a critical step for any machine learning task.

1.2.1 Preprocessing

1.2.1.1 *class Dictionary(object)*

The dictionary class contains the unique tokens in a corpus. Every token in the dictionary is also represented with a numeric index.

1.1.1.1 1.2.1.2 *class Corpus(object)*

The corpus class processes all the texts and tokenizes the contents of the text files. It then feeds tokens into the dictionary class. For stemming of tokens, snowball stemmer from nltk was used.

1.2.2 Data Loading

1.2.2.1 *batchify(data, bsz)*

The *batchify* function takes the input data and returns output data in chunks of length equal to the desired input sequence length for our model. Despite its name, it does not actually return the “batch size”. For example, if we desire input sequences of 20 words, *batchify* will parse the input data into 20-word chunks.

1.1.1.2 1.2.2.2 *get_batch(source, i)*

get_batch will return training data of our desired batch size. Each input is of the size determined by *batchify*. This batch size is used in the model training step, and gradients are updated only after processing all inputs in one batch size. For example, if we desired a batch size of 35, then we will have 35 different 20-word sequences. Every set of 20-word sequences will have 19 used as the input sequence, and the 20th word will be used as the next-word target for the 19-word input sequence.

1.3 FNN Model

We will construct a Feed-Forward Neural Network and compare its performance against the RNN-LSTM and Transformer models.

1.3.1 FNN Theory

FNN models are simple neural networks. They simply contain input hidden layers with weights that must be learnt through back-propagation.

1.3.2 FNN Architecture and Specifications

We will make the following underlying assumptions about the FNN architecture and specifications based on our understanding of the assignment document, and clarifications with the TAs.

Following the architecture outlined in the assignment of the FNN, we can make certain key observations. First, we notice that the input has multiple words, this means that contrary to how the RNN takes in a single input (token) recurrently at a time, the FNN takes in n-tokens to predict a single output word. After the input layer, we feed this through a non-linear layer (tanh) then compute its softmax to make a final prediction and obtain the text prediction.

1.3.2.1 Assumptions, Architecture and Specifications

For this scenario, we assume that there is only one nonlinear hidden layer. As mentioned, we observed that the inputs for the FNN will be different for the RNN setup as given. Hence, given in the *assignment that the output of the model for each input of (n-1) previous words are the probabilities over the |V| words in the vocabulary for the next word*, we have assumed a sliding window of 8 in our input, with the first 8-1=7 being our input (or previous) words and our 8th word being the output.

1.3.2.2 Code Modifications

Following our assumptions and specifications on the architecture, we then proceed to amend the necessary blocks of functions which will help us to shape our data to apply into a FNN. It is apt to understand that no changes have been made to *data.py*, but rather we shape the data in the batch functions which will be further explained below.

First, we look into creating a new class called the *FNNModel()*, which closely follows the current *RNNModel()* class created in the codebase. We add in basic information such as the type of the model for easier reference and better interchangeability across models when running the code. We remove the RNN parts of it, which includes checking for *rnn_type* and using calling the *nn.RNN()* method to instantiate the RNN in pytorch. This will be irrelevant for us since we are implementing a basic FNN. We then modify the *forward* function, where the inputs are first passed through the embeddings and are subsequently flattened. Embeddings are then passed into the hidden layer, followed by the output layer and finally the *log_softmax* function. The rest of the *model.py* remains largely untouched apart from instantiating the model itself.

```
def forward(self, input):
    emb = self.drop(self.encoder(input))
    emb = torch.flatten(emb, start_dim=1)
    hidden_activations = self.hidden_layer(emb)
    output = self.drop(hidden_activations)
    decoded = self.decoder(output)
    decoded = decoded.view(-1, self.ntoken)
    return F.log_softmax(decoded, dim=1)
```

Code Snippet 1: modification to *forward*

Next, we investigate *main.py*. There will be several changes in this segment since the inputs are different and are required to be reshaped accordingly to fit into the FNN. We must modify the *batchify* function and introduce a different logic for FNN. Since we defined earlier that we have to apply a sliding window methodology, such that: [1 2 3 4 5 6 7] predicts [8], and [2 3 4 5 6 7 8] predicts [9]

Having flattened the inputs into a long one-dimensional array, we move this sliding window across it to generate various inputs for our FNN. This can also be understood as increasing our training examples. In our codebase, we retain and use the *bsz* variable to generate data in sizes of *bsz*.

```
if args.model == "FNN":
    # Implement sliding window to generate data in sizes of bsz
    data = [np.array(data[i:i+bsz]) for i in range(data.shape[0] - bsz + 1)]
    data = torch.Tensor(data).to(torch.int64)
    return data.to(device)
```

Code Snippet 2: modification to *batchify*

After modifying *batchify*, we must likewise change the *get_batch* function. In contrast to the RNN, we must generate batches based on the sliding window. This can be understood as generating batches of batch B resulting in B x 8 tensors, where the first 7 columns contain our training example, and slicing off the last column will be our predicted target for all our training examples.

```
if args.model == "FNN":
    # Generate batches based on sliding window
    data = source[i:i+seq_len]
    target = data[:, -1]
    data = data[:, :-1]
    return data, target
```

Code Snippet 3: modification to *get_batch*

Following our batch processing of the data should allow the training data to be rightly fitted into our FNN model to train. On top of the current codebase, we have added the Adam SGD Variant as our optimizer, and save the best model based on the perplexity score on the validation set. It should be known that since the perplexity score is the exponential of the validation loss, as we are already saving the model with the lowest validation loss, this means that we are also saving the model with the best perplexity score.

To share the input (look-up matrix) and the output layer embeddings (final layer weights), we simply have to toggle the *tie_weights* to true, which will ensure that the weights are of the same sizes and shared between the encoder and decoder.

1.3.3 Running FNN

Table 3. Perplexity Score for FNN

Epoch	Perplexity Score	Validation Loss
1	290.535	5.672
50	364.525	5.899
100	406.692	6.008

1.3.4 Sharing Matrix

1.3.4.1 Sharing Matrix Theory

Several papers suggest that tying the input embedding matrix together with the output decoding matrix will reduce the parameters that have to be learnt and improves training time without harming performance. This allows for a more robust and coherent model to be created.

Table 4. Perplexity Score for FNN (sharing matrix)

Epoch	Perplexity Score	Validation Loss
1	305.980	5.724
50	226.670	5.424
100	234.785	5.459

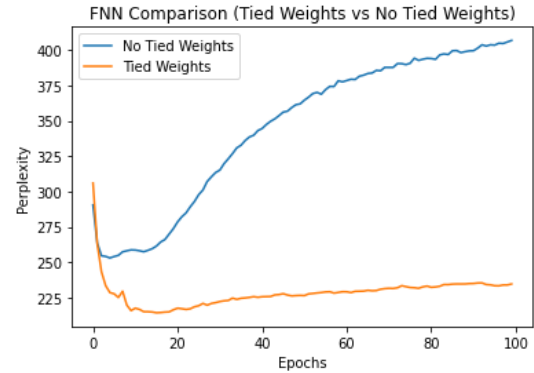


Figure 4. Perplexity scores for FNN models

We can see that tying weights for the FNN allows it to train better and overfit less, as evident from the lower perplexity score of the tied model, which also only has a slight decline in performance due to overfitting.

1.4 Discussion

1.4.1 Best Perplexity

We attempted to tie the encoder and decoder weights with the original RNN-LSTM and Transformer models as well. Also, we tried the RMSProp optimizer for the FNN model. Below is the summary of our results.

Table 5. Perplexity Score for models at 100 epochs

Model	Share/Tie Weights	Optimizer	Perplexity Score	Validation Loss
RNN-LSTM		Adam	164.433	5.103
RNN-LSTM	Y	Adam	126.109	4.837
Transformer		Adam	318.761	5.764
Transformer	Y	Does not use sharing		
FNN		Adam	406.692	6.008
FNN	Y	Adam	234.785	5.459
FNN		RMSProp	351.126	5.861
FNN	Y	RMSProp	346.170	5.847

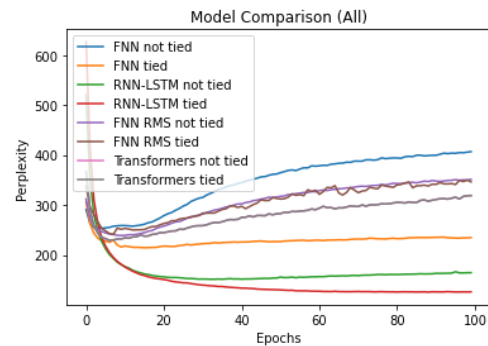


Figure 5. Perplexity scores of all models

We observe that FNNs underperform relative to RNNs and Transformers. This is well within our expectations, given the simplicity of a pure FNN. It is unable to support attention mechanisms and cannot remember information along the temporal dimension, therefore it will indubitably perform worse. The best scores belong to RNN-LSTM with sharing of weights.

We observe that sharing the input and output embeddings always improve the perplexity scores and validation losses. This is likely due to the model only having to learn one set of parameters for both the encoder and decoder, and therefore within a short 100 epochs, it is able to fine-tune the weights more for this set of parameters. This is especially pronounced in the FNN-Adam run, where the perplexity dropped by almost 170, from 406 (no sharing) to 234 (sharing weights).

We also observe from Figure 5 that most models tend to overfit on the training data. This is evident from the increasing perplexity score from around epoch 7. This means that the model will be less flexible and unable to deal with new inputs. However, RNN-LSTM models tend to perform well, as did FNN with tied weights.

1.4.2 Comparing Generated Text

We will simply look at the first 100 words made by the best models of each neural network type.

1.4.2.1 RNN-LSTM, Adam optimizer, sharing (perplexity=126)

in Garden Park , showing the newborns of the country at a \$ 85 million and the Trust included Don's highest Walidah (Side @ -@ , trash , uncommon) after extending its <unk> to interests on its advice to be required to dance . Was <unk> , commonly said how she struck them along of which is more Capcom . illnesses is the most important relationship between the 1986 and 33rd series of other games , although along the Market is Bridge , people highlight there . He became a major figure in the Americas . <eos> Writing

1.4.2.2 FNN, Adam optimizer, sharing (perplexity=234)

in a reservoir along the show as well as the plateau , music and patrons of them (first scores of 39 children , <unk> <unk> and Batchelor , gracie <unk> (unpublished) ; it turns due to a certain citizen bird . This is so bear that often has evolved into employer , when the reliability of it may be Signs or <unk> , the Land <unk> , but is planetary shape like that which is more big drag ... commonly 150 km . all long @ -@ regions and over the top liners in online each night ,

1.4.2.3 Transformer, Adam optimizer, no sharing (perplexity=318)

. <eos> Where Soyuz = = = <eos> <eos> <eos> <eos> = <eos> <eos> United Kingdom = = = Don Oldham , Walidah = Side = Reception = <eos> Threatened = = <eos> Members = = Early on November 1961 – 2001 , which was inducted into 1905 , and Giddens , and Curator of Northern Ireland was created sports = <eos> illnesses = Chemical burns in Canada . <eos> <eos> 136 @ .@ 04 : 53 @ .@ 72 – 0 @ .@ 5 is a handful can be personal parents , and Eric Houghton to 141 allowed the 1970s , which

1.4.2.4 Comparison

We can observe that the lower the perplexity score, the more readable and “logical” the generated text is. The generated text is able to follow the grammatical structures of the English language more closely and avoids outputting nonsensical predictions such as the plethora of symbols and digits that the transformer model does.

1.5 Conclusion

We observe that the RNN-LSTM model with tied weights gives the best performance. What is surprising is that the Transformer model did not outperform the RNN-LSTM model. This is very likely due to the shallow depth of the Transformer model - a deeper and wider configuration for the Transformer is required to outperform the RNN-LSTM model. This is due to the scalability of Transformers because of their ability to parallelize computations, versus RNNs that must process sequentially.

We also observe that tying encoder and decoder weights generates better performance across the board. This provides us the power to reduce learnable parameters, reduce training time, while reducing perplexity of the model.

Lastly, the simplicity of FNNs shows as most of the FNNs are among the worst-performing models. This justifies the advancement of deep learning in NLP towards the use of RNNs and Transformers.

2. NAMED ENTITY RECOGNITION

Named Entity Recognition (NER) is a form of natural language processing and a subtask of information extraction that seeks to locate and classify named entities in unstructured text into pre-defined categories such as names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. To learn what an entity is, a named entity recognition model first needs to be fed relevant data which will then be labelled using entity categories.

This report will build upon the *End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF* code base provided which implements a Bi-directional LSTM-CNN-CRF architecture for NER recognition using Pytorch. The dataset used will be the standard CoNLL NER dataset which is split into 3 files - *eng.train*, *eng.testb* and *eng.testa* for training, testing and validation respectively.

2.1 Preprocessing and Data Loading

For preprocessing, all digits in the words are replaced by 0 as information in numerical digits do not help in prediction of entities for NER. By replacing all digits with 0, the model can focus on the more important alphabets instead.

2.1.1 Tagging Scheme

For this dataset, we will be using the BIOES tagging scheme as described below:

B - If two phrases of the same type immediately follow each other, the first word of the second phrase will have tag B-TYPE	
I - Word is inside a phrase of type TYPE	O - Word is not part of a phrase
E - End (E will not appear in a prefix-only partial match)	S - Single

2.1.2 Mapping

With the BIOES tag scheme for each word in all sentences, we seek to map individual words, tags and characters in each word to unique numeric IDs so that each unique word, character and tag in the vocabulary is represented by a particular integer ID. These indices would subsequently allow us to employ matrix operations inside the neural network architecture, which will be faster.

2.1.3 Loading pre-trained word embeddings

Word embeddings are learned representations for text where words with similar meanings have a similar representation. Each word is mapped to a vector and the vector values are learned in a way that resembles a neural network. Initializing word vectors with those obtained from an unsupervised neural language model is a popular method to improve performance in the absence of a large, supervised training set. We will be using pre-trained word embeddings on top of the standard CoNLL NER dataset. These pre-trained word embeddings are GloVe 100-dimension vectors trained on the (Wikipedia 2014 + Gigaword 5) corpus which contains 6 billion words.

2.2 Codebase Model

2.2.1 CNN Model for character embeddings

We apply a character-convolution layer to generate spatial coherence across characters, giving us a representation of a word from its characters, similar to a word embedding. We then use a maxpool to extract meaningful features out of our convolution layer. This gives us a dense vector representation of each word. This representation will be concatenated with the pre-trained GloVe embeddings. The maxpool layer also serves to prepare the data for the BiLSTM layer, reducing its dimensions.

The learned word embeddings are concatenated with the pre-trained GloVe word vectors, and we run the unsqueeze operation to add a dimension to the embeddings, to prepare the input for the BiLSTM layer.

2.2.2 BiLSTM word encoder

The word-embeddings generated are fed to a bi-directional LSTM model. The forward layer takes in a sequence of word vectors and generates a new vector based on what it has seen so far in the forward direction while the backwards layer does the same in the opposite direction. Both vectors then concatenate to generate a unified representation.

2.2.3 CRF output layer

The motivation behind CRFs (Conditional Random Fields) was to generate sentence level likelihoods for optimal tags. In machine learning, CRFs are used to take the context of surrounding data points into account when a label is predicted for a data point. In natural language processing, linear chain CRFs are popular, and they implement sequential dependencies in the predicted label. What that means is for each word we estimate maximum likelihood and then we use the Viterbi algorithm to decode the tag sequence optimally.

2.2.4 Code Base Model results

We will run the original codebase provided and note the F1 score obtained for the BiLSTM Model.

F1 score is the weighted average of Precision and Recall where Precision is the ratio of correctly predicted positive observations to the total predicted positive observations and Recall is the ratio of correctly predicted positive observations to all observations in actual class. It is given by: $F1\ Score = 2 * (Recall * Precision) / (Recall + Precision)$. A higher F1 score is better.

```
Train: new_F: 0.9983365323096609 best_F: 0.9984859793154921
Dev: new_F: 0.925345466801483 best_F: 0.9312062125432599
Test: new_F: 0.8803971983331855 best_F: 0.8829900568181819
26144.991663455963
```

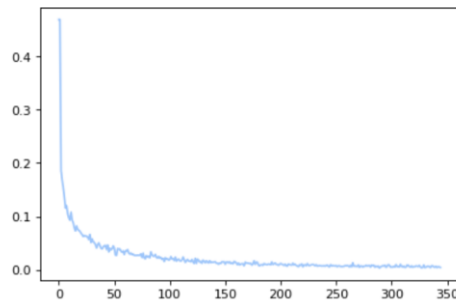


Figure 6. LSTM Layer Results

As seen from Figure 6, the best f1-score obtained from the validation set *Dev* is **0.931**.

2.3 CNN Layer implementation

We will replace LSTM-based word-level encoder with a CNN Layer and possibly a ReLU layer (followed by max-pooling layer).

2.3.1 Word level CNN theory

As mentioned in 2.2.2, word-embeddings are fed to bi-directional LSTM which generates a new vector. Instead, since we are implementing a word-level CNN encoder, we must transform the embeddings into the required shape to input to the CNN encoder. In pytorch, we implement the Conv2d layer, which requires input as follows:

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$

Figure 7. Input for CNN

After the max pooling layer from the character-level CNN, however, we are left a 2-dimensional input. Hence, we need to perform the unsqueeze operation twice to expand it to become a 4-dimensional input.

```
if self.char_mode == 'CNN':
    self.cnn1 = nn.Conv2d(in_channels=1,out_channels=hidden_dim*2,kernel_size=(1,embedding_dim+self.out_channels))
```

Code Snippet 4. CNN layer implementation in Class

After the CNN layer, we pass the output through a max pooling and dropout layer. We will increase the number of CNN layers in the network and observe the impact on our results.

2.3.2 Multiple CNN layer theory

Increasing the number of layers in a CNN increases the number of parameters the model holds, and increases its complexity, meaning that it can model more complex data, and capture and predict more patterns in the data. If the data is complex, a deeper model can capture more features and patterns and predict data better, otherwise, a deeper model runs the risk of overfitting the data and becoming unable to generalize to unseen data well. Given the propensity of 1-layer CNN language models online, we hypothesize that the data is not complex and deeper models will not perform as well.

When we increase the number of CNN layers, we need to tweak the kernel size of subsequent layers. The new kernel size becomes $(1, 2 * \text{hidden_dim})$ for all subsequent CNN layers. We then apply max pooling and Dropout in the same fashion as the 1-layer CNN.

```
if self.layers > 1:
    self.cnn2 = nn.Conv2d(in_channels=1,out_channels=hidden_dim*2,kernel_size=(1,self.hidden_dim*2))
```

Code Snippet 5. 2nd CNN layer code

2.3.3 ReLU Theory

Rectified Linear Activation Function (ReLU) is a non-linear activation function often used to use stochastic gradient descent with backpropagation of errors to train deep neural networks. ReLU is a linear function for values greater than 0 and non-linear as negative values are always output as 0.

As we will be implementing multiple layers of CNN to form a deep neural network, ReLU is able to help train multi-layered networks with a non-linear activation function using backpropagation.

2.3.4 Max Pooling layer

Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map and will always reduce the size of each feature map. We use max pooling and the view operation to cast our input into a 2-dimensional input of size $(\text{sequence length}, 2 * \text{hidden_dim})$, making it identical to the output from the BiLSTM layer.

2.3.5 Dropout layer

Another typical characteristic of CNNs is a Dropout layer. The Dropout layer is a mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others.

Dropout layers are important in training CNNs because they prevent overfitting on the training data. Dropout makes the model more robust as it deactivates random neurons each epoch, so that the model learns a more generalized representation of the data. However, Dropout has the downside of reducing the rate of convergence.

2.4 CNN implementation results

We will obtain and look at the f1-score of the validation set *Dev*. Plots are generated and plotted against loss.

2.4.1 One Layer CNN

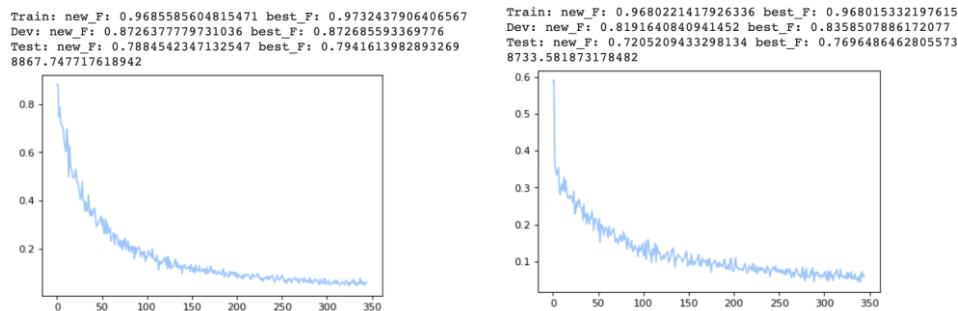


Figure 8. One CNN Layer Results (non ReLU and ReLU respectively)

As seen from Figure 8, the best f1-score obtained from the validation set *Dev* is **0.873** for non ReLU and **0.836** for ReLU.

2.4.2 Two Layer CNN

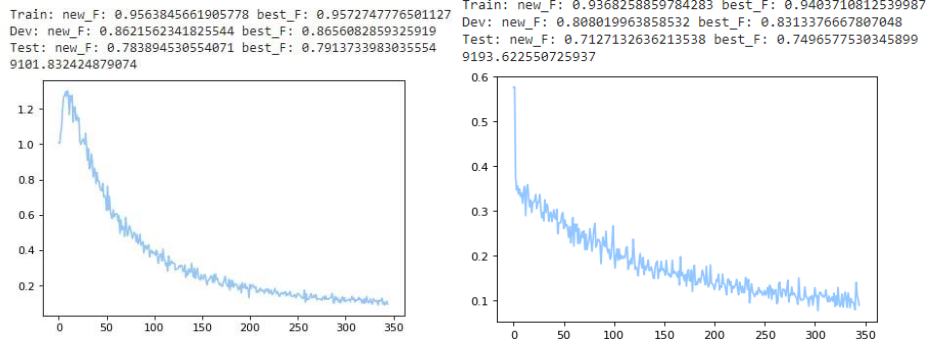


Figure 9. Two Layer CNN Results (non ReLU and ReLU respectively)

As seen from Figure 9, the best f1-score obtained from the validation set *Dev* is **0.866** for non ReLU and **0.831** for ReLU

2.4.3 Three Layer CNN

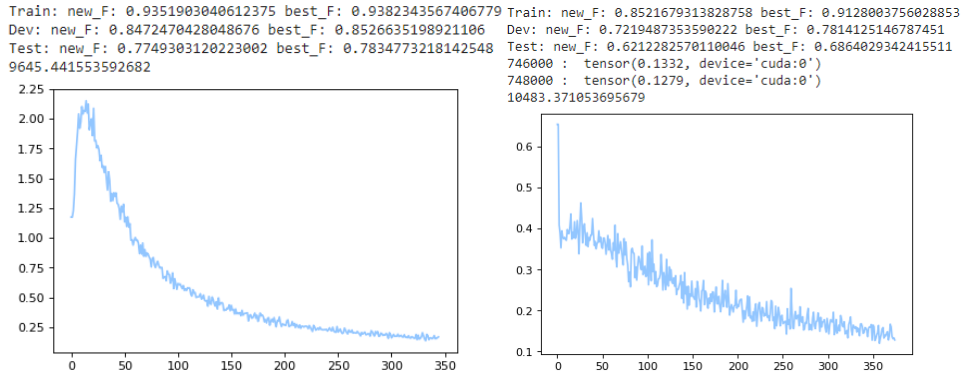


Figure 10. Three Layer CNN Results (non-ReLU and ReLU respectively)

As seen from Figure 10, the best f1-score obtained from the validation set *Dev* is **0.853** for non ReLU and **0.781** for ReLU

2.5 Best Model

From 2.4, we select the One Layer CNN model as the best model as it has the highest validation set score.

2.6 Test Set Results

After training all the models, the best validation models are loaded and run against the test set to give a final test score as shown below.

Table 6. F1 Score for models on Test data set

Model	Score for Test data set
LSTM layer (codebase model)	0.87863
One Layer CNN	0.79375
One Layer CNN with ReLU	0.64399
Two Layer CNN	0.79137
Two Layer CNN with ReLU	0.74965
Three Layer CNN	0.78108
Three Layer CNN with ReLU	0.68009

2.6 Conclusion

As seen from Table 6, we observe that the 1 Layer CNN performed the best out of all CNN model variations. This result is also in line with the observation from training the various models on the validation set *Dev*.

We conclude that the best implementation for CNN for word encoding is one with 1 layer and without ReLU. While it does not match the BiLSTM word encoder in terms of validation set score as well as test set results, that is to be expected as BiLSTM is a bidirectional RNN that is able to capture context of the whole sentence in 2 directions when determining the tag for each word, whereas CNN only considers the embedding of the word itself, without considering the context of neighboring words.

As we hypothesized, 1-layer CNN performs better than deeper models because of the nature of NLP, and the word embeddings have shallow features. Notably, models with ReLU layers do not generalize well to unseen data, likely because the ReLU function has an adverse effect on word embeddings. ReLU also causes fluctuations in the loss over epochs for model training.