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.

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

Name	Signature / Date
Chua Zi Heng	ziheng 10/11/21
Mun Kei Wuai	Keiwuai 10/11/21
Pooja Srinivas Nag	Pooja 10/11/21
Tan Wen Xiu	Wenxiu 10/11/21

Assignment 2

Group 27

Chua Zi Heng
U1922370K
Nanyang Technological University
chua0954@e.ntu.edu.sg

Mun Kei Wuai
U1921982B
Nanyang Technological University
kmun001@e.ntu.edu.sg

Pooja Srinivas Nag
U1921189G
Nanyang Technological University
pooj0003@e.ntu.edu.sg

Tan Wen Xiu
U1921771H
Nanyang Technological University
wtan151@e.ntu.edu.sg

This report is split into 2 parts: the first part focuses on developing and training a basic neural language model and using it to generate new text; the second part focuses on developing and training a NER model to predict named entity types for a new text.

We have made our codes in such a way that they would be able to run on both CPU (multi-core available) and GPU. In this project, the codes were ran on Google Colaboratory with GPU.

1. Building a Language Model

This section will build upon the word_language_model codebase provided by PyTorch which implements the RNN and Transformer architecture. In this part, we will implement a feed-forward network (FNN) architecture and compare its performance in language modelling against the theoretically better RNN and Transformer models. Language models can determine the probability of a sequence of tokens and this can be used to automatically generate texts.

1.1. Try to run the code

data.py - creates a dictionary and corpus from 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 classes for different model architectures
generate.py - generates new sentences sampled from the language model
data folder - contains train.txt, valid.txt, test.txt

Here, we run the code using default arguments ie. LSTM model and train it for 1 epoch to see the output, which is shown below.

```
! python main.py --epoch 1 --cuda

| epoch   1 |   200/ 2983 batches | lr 20.00 | ms/batch 48.80 | loss  7.63 | ppl  2068.24
| epoch   1 |   400/ 2983 batches | lr 20.00 | ms/batch 47.10 | loss  6.86 | ppl   953.24
| epoch   1 |   600/ 2983 batches | lr 20.00 | ms/batch 47.37 | loss  6.50 | ppl   663.25
| epoch   1 |   800/ 2983 batches | lr 20.00 | ms/batch 47.67 | loss  6.30 | ppl   544.83
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 47.96 | loss  6.16 | ppl   471.85
| epoch   1 |  1200/ 2983 batches | lr 20.00 | ms/batch 47.83 | loss  6.07 | ppl   432.85
| epoch   1 |  1400/ 2983 batches | lr 20.00 | ms/batch 47.79 | loss  5.96 | ppl   387.37
| epoch   1 |  1600/ 2983 batches | lr 20.00 | ms/batch 47.86 | loss  5.96 | ppl   388.76
| epoch   1 |  1800/ 2983 batches | lr 20.00 | ms/batch 47.67 | loss  5.82 | ppl   336.07
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 47.68 | loss  5.80 | ppl   330.01
| epoch   1 |  2200/ 2983 batches | lr 20.00 | ms/batch 47.63 | loss  5.68 | ppl   294.24
| epoch   1 |  2400/ 2983 batches | lr 20.00 | ms/batch 47.67 | loss  5.69 | ppl   294.79
| epoch   1 |  2600/ 2983 batches | lr 20.00 | ms/batch 47.64 | loss  5.67 | ppl   288.88
| epoch   1 |  2800/ 2983 batches | lr 20.00 | ms/batch 47.67 | loss  5.55 | ppl   257.54
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 147.67s | valid loss  5.51 | valid ppl   246.62
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  5.42 | test ppl   226.33
=========================================================================================
```

1.2. Understanding Preprocessing and Data Loading Functions

1.2.1. Data Loading

The data loading logic can be found in `data.py`, which contains 2 classes, namely Dictionary and Corpus. When the main.py script is called, an instance of Corpus() and Dictionary() is initialized.

The Dictionary essentially contains logic for the mapping of words to their respective indices, vice-versa, and stores them in a dictionary and list respectively. Corpus has the main tokenize() method that parses each line in the data file and adds an <eos> tag at the end of each line. It also calls the add_word() method from Dictionary so each new token will be added to the idx2word list if it does not exist yet. The position (ie. index) of the token in this idx2word list will be the number the token is allocated to in the word2idx dictionary. For example, initially the idx2word list and word2idx dictionary are empty.

```
idx2word = []
word2idx = {}
```

When we add the token “morning”,

```
idx2word = ["morning"]
word2idx = {"morning": 0}
```

1.2.2. Data Preprocessing

The data processing methods are in main.py. The 2 main functions are:

1. **batchify(data, bsz)**

It takes the input data and bsz (batch size) and the data.narrow() method will trim off extra elements that do not cleanly fit, ie remove elements such that the remaining number of elements is a multiple of the batch size. This method will return output data with the number of columns equal to batch_size.

2. **get_batch(source, i)**

It takes in the input data and the index, and returns the predictor and target variables. There will be a sliding window equal to the sequence length.

1.3. Training the FNN Model

We are tasked to implement a FNN based on the code base so that the model could learn the distributed representation of each word and the probability function of a word sequence as a function of their distributed representations.

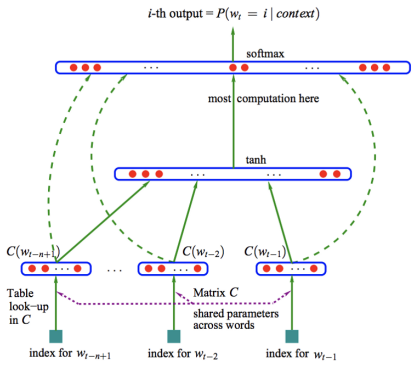


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

Based on the architecture above, after the input embedding layer, the model has a hidden layer with tanh activation and the output layer which is a Softmax layer. 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, ie each word in the vocabulary will be assigned a probability, which shows how likely the word is going to appear next in the sequence. We can then take the word that gives the highest probability.

We modified the original code and implemented a FNNmodel class as shown below.

```
class FNNmodel(nn.Module):

    def __init__(self, vocab_size, embedding_dim,
context_size, h, tie_weights=False):
        super(FNNmodel, self).__init__()
        self.context_size = context_size
        self.embedding_dim = embedding_dim
        self.embeddings = nn.Embedding(vocab_size,
embedding_dim)
        self.linear1 = nn.Linear(context_size *
embedding_dim, h)
        self.linear2 = nn.Linear(h, vocab_size, bias
= False)
        self.init_weights()

        if tie_weights:
            print("*TIED WEIGHTS SELECTED*")
            if embedding_dim != h:
                raise ValueError("In order to share
embedding weights, embedding size must be equal to
number of hidden nodes")
            self.linear2.weight =
self.embeddings.weight
        else:
            print("*TIED WEIGHTS NOT SELECTED*")

    def init_weights(self):
        initrange = 0.1
        nn.init.uniform_(self.embeddings.weight,
-initrange, initrange)
        nn.init.zeros_(self.linear2.weight)
        nn.init.uniform_(self.linear2.weight,
-initrange, initrange)

    def forward(self, inputs):
        # compute x': concatenation of x1 and x2
embeddings
        embeds =
self.embeddings(inputs).view((-1,self.context_size *
self.embedding_dim))
        # compute h: tanh(W_1.x' + b)
        out = torch.tanh(self.linear1(embeds))
        # compute W_2.h
        out = self.linear2(out)
        # compute y: log_softmax(W_2.h)
        log_probs = F.log_softmax(out, dim=1)
        # return log probabilities
        # BATCH_SIZE x len(vocab)
```

```
        return log_probs
```

init_weights() method will initialize the weights to follow a uniform distribution.

In this case, we had to train an 8-gram language model, meaning that our ngram_size = 8 hence our predictor size = 7 and the 8th word is our target.

Since Adam and RMSProp optimizers are used, a good starting learning rate of lr=4e-4 is chosen as studies have shown that a lower starting lr for these 2 optimizers proved effective.

To prevent overfitting of data as the training progresses, we included a learning rate scheduler, `ReduceLROnPlateau`, which decays the LR by a pre-set factor once the loss metric does not improve. Since only a small number of epochs are used, we set the patience=0 and factor=0.1, which means that the LR will divide itself by 10 once the validation loss worsens (ie. increases) during training.

```
scheduler =
optim.lr_scheduler.ReduceLROnPlateau(optimizer,
factor=0.1, patience=0,verbose=True)
```

The train(args, model, train_data, val_data) method in main.py will serve to train the model and save the best model based on the lowest perplexity score on the validation set.

We will use the same parameters across all models for fair comparison.

Hidden units: 200, Embedding size: 200, Cuda: True, Learning rate: 4e-4, Batch size: 512, Epochs: 20
Default optimizer: Adam, Default model: FNN

We can pass in this line of code to run the FNN model with the default Adam optimizer.

```
python main.py --cuda
```

1.4. Perplexity Scores on Test Set

For this section, we ran all 3 types of models (FNN, LSTM, Transformer), with Adam and RMSProp each.

Model	Optimizer	perplexity (test)	loss (test)	time (training)
FNN	RMS	255.63	5.54	106.4
	Adam	211.61	5.35	114.3
RNN (LSTM)	RMS	179.53	5.19	78.6
	Adam	195.01	5.27	83.6
Transformer	RMS	219.75	5.39	87.1
	Adam	225.06	5.42	88.2

We can see that the LSTM model performs the best as it takes into account the sequence of the input data, which FNN does not. Furthermore, LSTM is an improvement of the traditional RNN which tries to overcome the vanishing/exploding gradient problem faced by the basic RNN by retaining important long term information and forgetting the irrelevant ones.

The Transformer model surprisingly did not perform as well. One reason could be due to the shallow depth of the Transformer model. In this case, only 2 layers were used. A deeper and wider configuration for the Transformer is required to reap the benefits of scalabilities. Furthermore, transformers are known for their self-attention mechanism which proved to be good for long range dependencies. However, since our ngram size = 8 only, the transformer model could not fully reap the potential advantages a Transformer model could bring.

1.5. Sharing the Input and Output Layer Embeddings

For this section, the input (look-up) matrix will be shared with the output layer embeddings (final layer weights). That is, instead of using two weight matrices, we just use only one weight matrix. However, a condition for using this method would be that the embedding size has to be equal to the number of hidden neurons. We would need to call the --tied flag as shown below:

```
python main.py --cuda --tied
```

The above code will implement the FNN model with Adam optimizer. Studies and researches have shown that sharing or tying the weight matrix between input-to-embedding layer and output-to-softmax layer can reduce overfitting. Since we are using only 1 weight matrix, this method also massively reduces the total number of parameters in the language model, thus reducing training time, as shown in the table below.

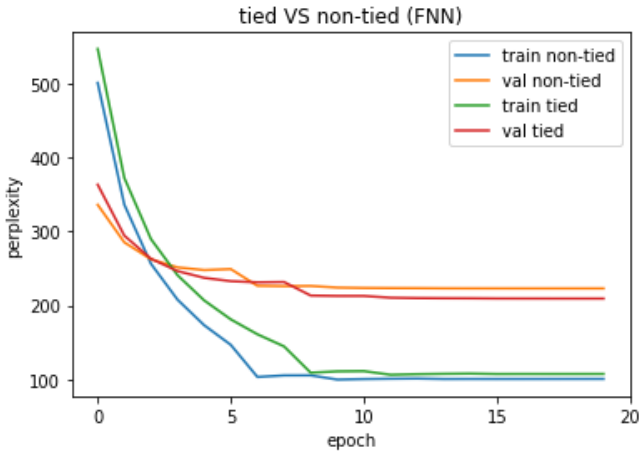
The code below allows us to get the number of parameters:

```
print("Total Trainable params: ", sum(p.numel() for
p in model_.parameters() if p.requires_grad))
```

Model (FNN with Adam)	# trainable params	Avg time taken per epoch (s)	Test perplexity
Non-tied	11777800	114.29	211.61
Tied	6029000	101.45	199.47

Now that we have looked at the number of trainable parameters and the average time taken for an epoch, next, we will compare the performance of tied VS non-tied weights, in terms of perplexity. In the table above, the model with tied weights seems to do better as it has a lower test perplexity.

Furthermore, from the graph below, we can see that there is indeed less overfitting for the model with tied weights. The model with tied weights also achieved a lower validation perplexity.



We ran all models to obtain a full and comprehensive set of results, which are shown in the table below.

Model	Optimizer	tied	perplexity (test)	loss (test)	time (training)
FNN	RMS	True	250.96	5.53	96.1
		False	255.63	5.54	106.4
	Adam	True	199.47	5.34	101.3
		False	211.61	5.35	114.3
RNN (LSTM)	RMS	True	161.53	5.08	77.9
		False	179.53	5.19	78.6
	Adam	True	181.06	5.20	82.1
		False	195.01	5.27	83.6
Transformer	RMS	False	219.75	5.39	87.1
	Adam	False	225.06	5.42	88.2

We can see that all the models with tied weights performed better than the one that is not tied (ie. lower val_perplexity, lower val_ loss and lesser time taken per epoch).

1.6. Generating Texts using the Language Model

In this section, we will be adapting generate.py to generate texts. The FNN model with Adam optimizer and tied weights will be the model used.

```
start = random.randint(0, len(full_corpus)-7)
word_gram = full_corpus[start:start+7]
generated_text=word_gram.to(device)

for i in range(args.words):
     with torch.no_grad():
          output = best_model(generated_text[-7:])
          word_idx = torch.argmax(output, dim=1)
          generated_text = torch.cat(
                                (generated_text,word_idx))
```

Step 1: Load model
Step 2: Randomly select a word from the full corpus, and subsequently extract a sequence of 7 words.
Step 3: Initialize the generated sequence with those 7 words.
Step 4: Fit the last 7 words in the generated sequence into the model and the model predicts the most probable next word.
Step 5: Add the predicted word to the generated sequence
Step 6: Repeat steps 4-5 until desired number of generated words reached

In this case, we will be generating 20 words in total. The sequence starting with ### indicates the given 7-gram + 20 generated words, and the sequence starting with $$$ indicates the original sentence from the corpus.

Pred 1: Text generated within context

```
(nlp)  chuaziheng@tehbing06  ~/nlp_proj_2  main ±  python ge
nerate.py --tied
*TIED WEIGHTS SELECTED*
### titled snow snow , sumino , was >>> generated text >>> releas
ed on november 1 , 2008 . <eos> the first time of the season , an
d was the first time
------------------------------------------------------------------
----------------------
$$$ titled snow snow , sumino , was released on june 25 , 2003 .
the second novel was released on july
```

Pred 1 is a decent prediction as the predicted words fall within the context of the input sentence.

Pred 2: Bad prediction - repetitive

```
(nlp)  chuaziheng@tehbing06  ~/nlp_proj_2  main ±  python ge
nerate.py --tied
*TIED WEIGHTS SELECTED*
### episode party . <eos> crazy in love >>> generated text >>> wa
s the first episode of the episode was the first episode of the e
pisode was the first episode of the
--------------------------------------------------
$$$ episode party . <eos> crazy in love was re - recorded by beyo
nc for the film fifty shades of grey 2015
```

Pred 3: Bad prediction - Many unknown words

```
(nlp)  chuaziheng@tehbing06  ~/nlp_proj_2  main ±  python generat
e.py --tied --words 20
*TIED WEIGHTS SELECTED*
### , repulsed an attack by the japanese >>> generated text >>> army c
orps , and the <unk> of the <unk> . <eos> the <unk> of the <unk> <unk>
 <unk> , <unk>
--------------------------------------------------
$$$ , repulsed an attack by the japanese 35th infantry brigade , under
 the command of japanese major general <unk> <unk> .
```

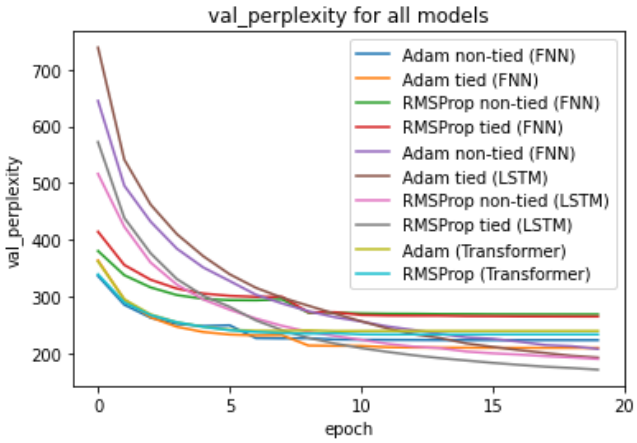
It is expected that the FNN model is sub-par in word generation.

Some interesting trends in the word generation include:
1. Repetition of short phrases: In Pred 2, where the phrase "the first episode" was repeated multiple times. Since FNN does not capture dependencies from past words in the sequence, it might get stuck in an "infinite n-gram loop", where the last word of a particular n-gram predicts the first word of the n-gram and it continues.

2. Unknown words: Due to the relatively small size of the corpus, the highest probability prediction for many cases might be just <unk>, which can be observed in Pred 3.

1.7. Conclusion

Overall, we observe that FNN underperforms relative to LSTM. This is not surprising, given the simplicity of a pure FNN. It is unable to support attention mechanisms and cannot remember long range dependencies, therefore it will perform worse. As mentioned earlier, since our ngram size = 8 only, the transformer does not perform as well as it should have as well. The graph below shows the val_perplexity for all models. LSTM with tied weights perform the best.



Tying weights together definitely helped reduce the overfitting and speeds up the training time. Hence, tied weights should be used over non-tied weights.

As an extension, we included another `inference.py`, which serves to predict the 8th token for every input of any 7 tokens the user provides. An example is shown below:

```
!python inference.py --cuda --tied --input 'my dog is brown and loves its'

########## GENERATE TEXT ##########
*TIED WEIGHTS SELECTED*
Next word:  own
```

2. Named Entity Recognition

In part 2, we will be developing a neural Named Entity Recognition (NER) model in Pytorch using CNN for the word-level encoding. We have made use of the given GitHub repository: End to end sequence labelling via bi-directional LSTM, CNNs, CRF that implements the CNN-LSTM-CRF NER model.

2.1. Dataset

The dataset used for this part is the English-NER sentences available in the CoNLL NER dataset and consists of 3 files: eng.train for training, eng.testa for validation and eng.trainb for testing. This consists of 4 columns, namely -- the word, its Part-of-Speech Tag, a syntactic chunk tag, followed by the named entity tag in the BIOES tag format.

U.N.	NNP	I-NP	I-ORG
official	NN	I-NP	O
Ekeus	NNP	I-NP	I-PER
heads	VBZ	I-VP	O
for	IN	I-PP	O
Baghdad	NNP	I-NP	I-LOC
.	.	O	O

2.1.1. Preprocessing

The data from the 3 files are loaded into a list of sentences, and a set of data preprocessing was conducted:

1. Replacing digits with a zero,
2. Updating the tagging scheme: BIO tagging scheme changed to BIOES tagging scheme.

<u>BIO Tagging (used by dataset)</u>

The BIO tagging scheme is used for tagging tokens in chunking tasks. This scheme is used in the preprocessing step to tag 4 entities: person, location, organization and misc.

The notations of the BIOES tagging scheme are as follows:
I (Inside) - Word is inside a phrase of the type
B (Beginning) - The first word of the 2nd phrase if 2 phrases of the same type are consecutive
O (Outside) - Word is not part of a phrase and can be ignored

BIOES Tagging (used by paper and codebase)
In the paper and codebase, the BIOES tagging scheme is used. On top of the BIO tagging used in the dataset, the 2 following notations are added as well:
E (End) -Tag E will not appear in a prefix-only partial match
S (Single)

Create mapping for words, characters and tags
The individual words, tags and characters in each sentence are mapped to unique numerical IDs so that each unique word, character and tag in the vocabulary is represented by a particular integer ID. This is performed by the create_dico, create_mapping, word_mapping, char_mapping, and tag_mapping functions in the given code base.

These indices for words, tags and characters help us employ matrix (tensor) operations inside the neural network architecture, which are considerably faster.

Once the final train, validation and test datasets, in the form of a list of dictionaries are prepared with the following keys for each sentence:

`str_words` : list of words in the sentence
`words`: word id of the words in the sentence
`chars`: character id for characters in `words`
`tags`: tag id of the tag for words in `words`

The train data is passed into the following models and F-scores for each model on the train, validation and test data are recorded at each epoch in `all_F`, a list of lists. This data is saved under a `csv` file in the results folder.

2.2. Models

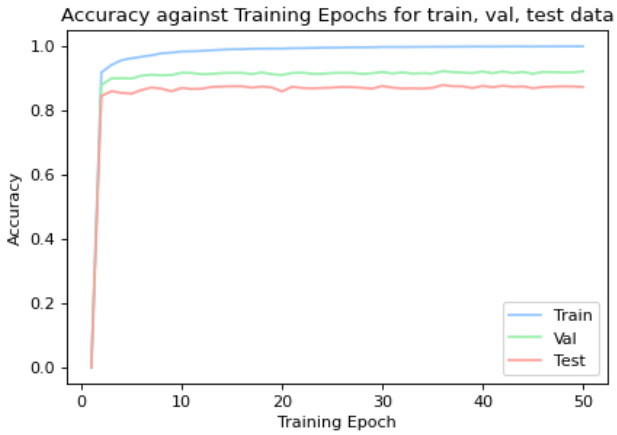
2.2.1. Bi-LSTM-CNNs-CRF Model

The original model proposed in the paper implements a character-level encoder with a convolutional neural network (CNN), followed by a word-level encoder with an Long-short Term Memory (LSTM) network. Finally, a conditional random field (CRF) layer is used as the output layer.

Model:

```
BiLSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (lstm): LSTM(125, 200, bidirectional=True)
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

Total number of trainable parameters = 2284255



F1 scores:

	End of training	Best value
Train	0.9983791853273619	0.9983365323096609
Val	0.920777027027027	0.9211415062478892
Test	0.8720743255315347	0.8782406579653137

Training+Evaluation time = 10366s
This model has a very stable and gently increasing F-score at each epoch. However the Bi-LSTM model implemented is very time consuming.

2.2.2. Our Model

Our task is to adopt the previous model and replace the LSTM network for the word-level encoder with a convolutional neural network (CNN) layer. We will then experiment with the dimensions of CNN layers, adding more CNN layers, as well as decide whether we should make use of the max pooling layer in the model.

Firstly, we built models using a 1-dimension convolutional layer (Conv1D) and a 2-dimension convolutional layer (Conv2D) respectively. We also varied the number of layers in the models to find out how the model would perform with different numbers of layers. We then added a max pooling layer to the model that performs best, to find out the effect of the max pooling layer. Finally, we compare the results of the different models to determine which combination of layers would produce the highest test accuracy results.

In all the models, the number of output channels of the convolutional layers is 400 channels. This is followed by a Linear layer that maps the output to the tag space (19 tags).

The general neural network model we built is the following:

`Character embedding layer`
`2D Convolutional layer`

```
Word embedding layer
Dropout layer
Convolutional layer(s) (1D or 2D)
Max Pool layer(s) (optional)
Linear layer
```

Summary of the types of models:

Model	Convolutional Layer	Number of layers	Max pooling layer
A	Conv1D	1	No
B	Conv1D	2	No
C	Conv1D	3	No
D	Conv1D	1	Yes
E	Conv1D	2	Yes
F	Conv2D	1	No
G	Conv2D	2	No
H	Conv2D	3	No
I	Conv2D	2	Yes

2.3. Model Results

This section will contain the results of different models which we have changed the parameters for experimentation. We will also be comparing and analysing the differences in results for the models.

2.3.1. 1D Models

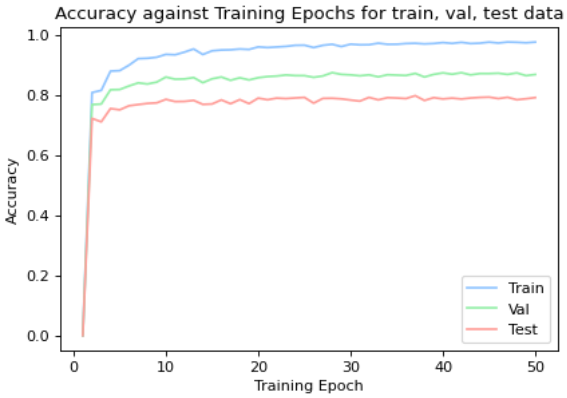
NLP tasks are foundationally the analysis of word vectors. Hence, 1D convolutional layers have traditionally been used to execute NLP tasks, and NER is no exception. Hence, we will be using 1D models to compare the differences in having different numbers of CNN layers.

A: 1 Layer Conv1D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 400, kernel_size=(125,), stride=(1,))
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True))
```

Number of trainable parameters = 1811455



F1 scores:

	End of training	Best value
Train	0.9761605487973753	0.9763732514500171
Val	0.868312757201646	0.8744177871517709
Test	0.7914924559171059	0.7977798334875116

Training+Evaluation time = 8384s

B: 2 Layer Conv1D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 125, kernel_size=(3, 125), stride=(1,),
padding=(1, 0))
  (conv2): Conv1d(125, 400, kernel_size=(3, 1), stride=(1,),
padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

Number of trainable parameters = 1958455

F1 scores:

	End of training	Best value
Train	0.9696155568349004	0.9696155568349004
Val	0.8352243861134632	0.8487854943551146
Test	0.7471603613272516	0.7687011537030144

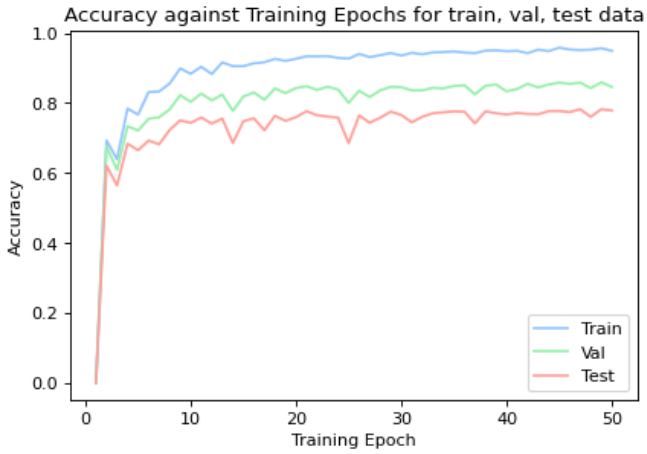
Training+Evaluation time = 9485s

C: 3 Layer Conv1D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 125, kernel_size=(3, 125), stride=(1,),
padding=(1, 0))
  (conv2): Conv1d(125, 125, kernel_size=(3, 1), stride=(1,),
padding=(1, 0))
  (conv3): Conv1d(125, 400, kernel_size=(3, 1), stride=(1,),
padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

Number of trainable parameters = 2005455



F1 scores:

	End of training	Best value
Train	0.9490157772917418	0.9581565877262167
Val	0.8456193865446533	0.8487854943551146
Test	0.778453829634932	0.7820983125458548

Training+Evaluation time = 8025s

Analysis for Conv1D networks

Best test F-scores:

1 Layer	2 Layer	3 Layer
0.7978	0.7687	0.7821

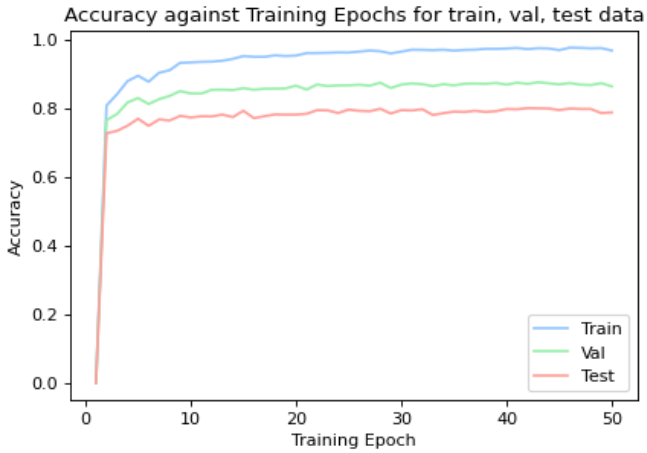
It is observed that the most optimal number of layers for the Conv1D networks is the 1 layered network as it obtains the highest val F-score among the three trials, with a fewer number of parameters to train. The F-scores for the 1 layer network also seem to show a more stable increasing trend compared to the other two models. We may conclude that given the hyperparameters used, increasing the number of layers generally causes the model to be worse off when 1D convolutions are used. This might be because the problem is not complex enough to require the additional convolution layer, or because the network might have overfit the training data. We will now apply the max pooling layer to observe any change in the model performance.

D: 1 Layer Conv1D Network with Max Pooling layer

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 400, kernel_size=(125,), stride=(1,))
  (maxpool1): MaxPool1d(kernel_size=3, stride=1, padding=1,
dilation=1, ceil_mode=False)
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True))
```

Number of trainable parameters = 1811455



F1 scores:

	End of training	Best value
Train	0.9681707784737313	0.976569501941872
Val	0.863765130054082	0.8487854943551146
Test	0.7879778443657496	0.8004167850715164

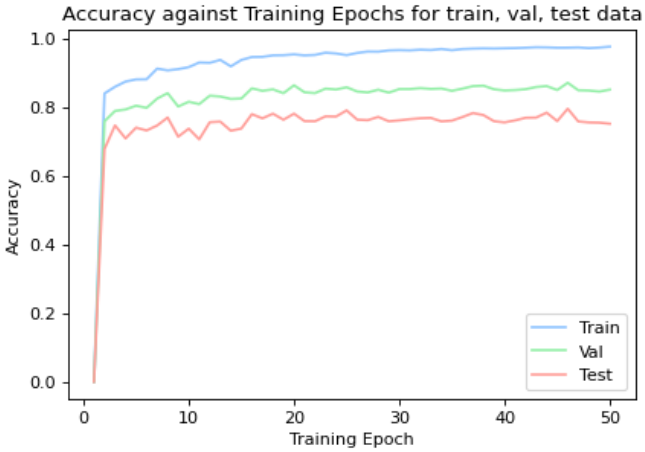
Training+evaluation time = 7483s

It is observed that the best val F-score for the 1layer 1D convolution network has improved very slightly upon using a max pooling layer. We will now apply the max pooling layers to a 2 layer 1D convolution network to observe any difference in effect.

E: 2 Layer Conv1D Network with Max Pooling layer

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1, 1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 125, kernel_size=(3, 125), stride=(1,), padding=(1, 0))
  (maxpool1): MaxPool1d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
  (conv2): Conv1d(125, 400, kernel_size=(3, 1), stride=(1,), padding=(1, 0))
  (maxpool2): MaxPool1d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
  (hidden2tag): Linear(in_features=400, out_features=19, bias=True))
```



Number of trainable parameters = 1958455

	End of training	Best value
Train	0.9759072107799242	0.9741906608981054
Val	0.8507663803267643	0.8706874736398144
Test	0.7515561088386983	0.7951785714285714

Training+Evaluation Time = 10003s

Upon adding a max pooling layer after each convolutional layer for the 2 layer model, the best val F-score is observed to improve by 0.0265 from 0.7687 to 0.7952, indicating that the model performs better with max pooling.

2.3.2. 2D Models

Although 2D convolutional layers, a superset of a 1D convolutional layer, have been traditionally reserved for deep learning tasks involving images, it has been introduced recently to analyze texts as well, such as in the case of Text Classification. We will run a few models using 2D convolutional layers, to explore the difference in performance between a Conv2D and a Conv1D model.

F: 1 Layer Conv2D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1, 1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv2d(1, 400, kernel_size=(3, 125), stride=(1, 1), padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19, bias=True)
)
```

Number of trainable parameters = 1911455

	End of training	Best value
Train	0.975581668123947	0.9756263994029215
Val	0.8523805453457559	0.8735071048731585
Test	0.7573622402890696	0.7883005736856955

Training+Evaluation Time = 7151s

G: 2 Layer Conv2D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv2d(1, 125, kernel_size=(3, 125), stride=(1,
1), padding=(1, 0))
  (conv2): Conv2d(125, 400, kernel_size=(3, 1), stride=(1,
1), padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

Number of trainable parameters = 1958455

	End of training	Best value
Train	0.9669527347252466	0.9686222752043596
Val	0.8555825242718447	0.8605891093435888
Test	0.7639911634756994	0.7773666636585147

Duration of training+evaluation = 8430s

H: 3 Layer Conv2D Network

Model:

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv2d(1, 125, kernel_size=(3, 125), stride=(1,
1), padding=(1, 0))
  (conv2): Conv2d(125, 400, kernel_size=(3, 1), stride=(1,
1), padding=(1, 0))
  (conv3): Conv2d(400, 400, kernel_size=(3, 1), stride=(1,
1), padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

	End of training	Best value
Train	0.9565718924230449	0.9565718924230449
Val	0.8160890029952932	0.8499827764381673
Test	0.724144176779569	0.759146902004045

Train+Evaluation Duration = 9982s

I: 2 Layer Conv2D Network with Max Pooling layer

```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1,
1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv2d(1, 125, kernel_size=(3, 125), stride=(1,
1), padding=(1, 0))
  (maxpool1): MaxPool2d(kernel_size=(1, 3), stride=(1, 1),
padding=(0, 1), dilation=1, ceil_mode=False)
  (conv2): Conv2d(125, 400, kernel_size=(3, 1), stride=(1,
1), padding=(1, 0))
  (maxpool2): MaxPool2d(kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), dilation=1, ceil_mode=False)
  (hidden2tag): Linear(in_features=400, out_features=19,
bias=True)
)
```

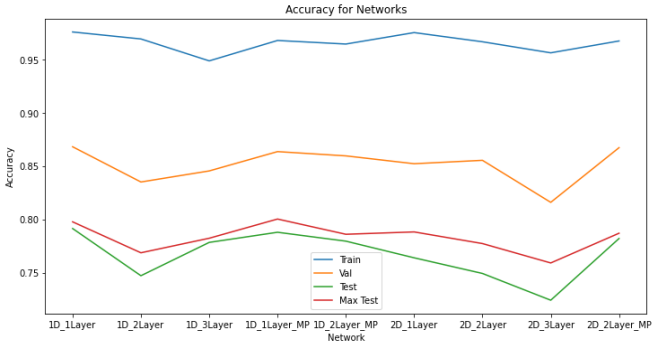
	End of training	Best value
Train	0.9677089109755577	0.9677323039518864
Val	0.8674719887955182	0.8689243726969643
Test	0.7821271003473199	0.7870833726749126

Training+evaluation duration = 7511s

It is observed that after max pooling, the F-score increases from 0.7774 to 0.7871.

2.3.3. Overall Results

Overall, the models fared well, with the train accuracy higher than the validation and test accuracies, which is expected.



From the graph above, we can see that in general, when a max pooling layer is added to the model, the test accuracy is better than without the max pooling layer. This is because max pooling is able to reduce the spatial size of the parameters and obtain the more significant features from the each sub-region (eg: each sentence). Furthermore, it is observed that max-pooling reduces the time taken for the model to train (model A and model D, model).

Before taking into account the models with the max pooling layer, the best performing network is the 1 layer Conv1D model. After adding on the max pool layer to the model, the max test accuracy outperforms that without the max pool layer. However, we can note that the difference in test accuracy for the 2 models is negligible.

For 2D convolutional layers, the model with 1 layer performed best. When the maxpool layer was added to the 2 layer Conv2D model, it outperformed the model with only 1 Conv2D layer.

Overall, the Conv1D models performed better than the Conv2D models for the NER tasks. However, tests with a higher number of layers should be carried out to discover more trends in the network models.

2.3.4. ReLu Activation Layer

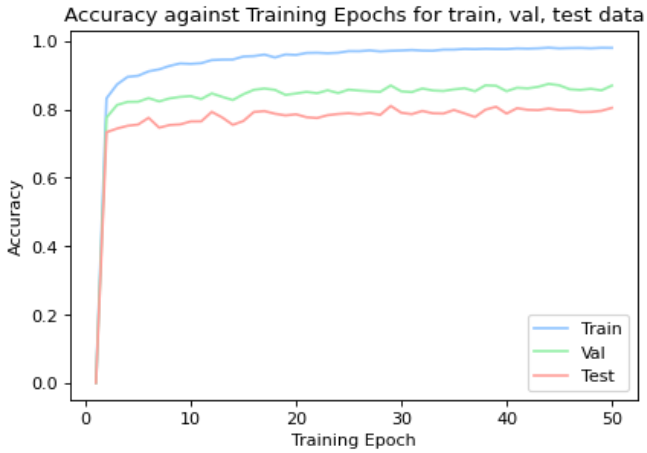
In a neural network model, an activation function is added to help the network learn complex patterns in the data by transforming the outputs of a layer which are to be passed into the next layer. The Rectified Linear Unit (ReLU) function is a non-linear activation function that will output the input directly if it is positive, otherwise, the output will be zero.

Text data is naturally non-linear. When the data goes through the convolution operation, linearity might be imposed on the data, hence we want to break up the linearity. Therefore, as an extension to the project, we added a ReLU activation layer to the best performing model, 1 Layer Conv1D.

Hence, we will further discuss the 1 Layer Conv1D Network with ReLU activation function below.

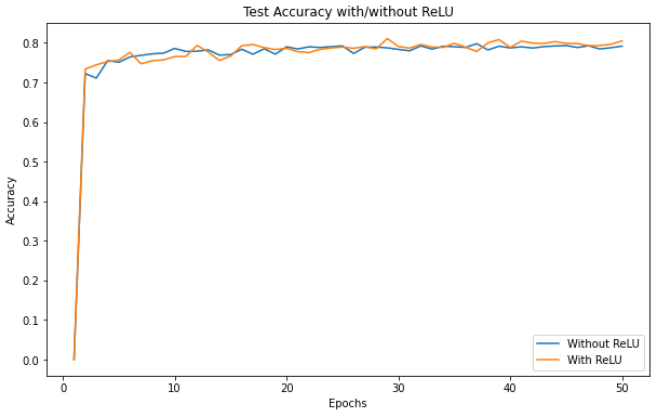
Model: The model for this network would be the same as the model for 1 Layer Conv1D network, with an addition of a Pytorch ReLU layer after the Conv1D layer.

Number of trainable parameters = 1811455



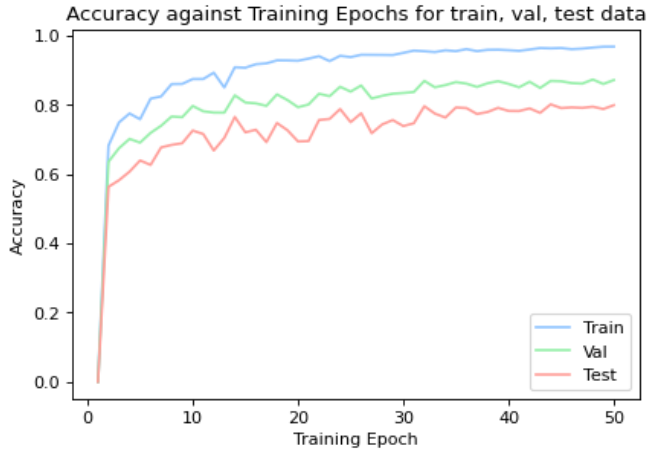
	End of training	Best value
Train	0.9805576873880788	0.9811827384123601
Val	0.8702302908051773	0.8751474304970512
Test	0.8048845234934962	0.8106581965766719

Duration of training+evaluation = 7467s



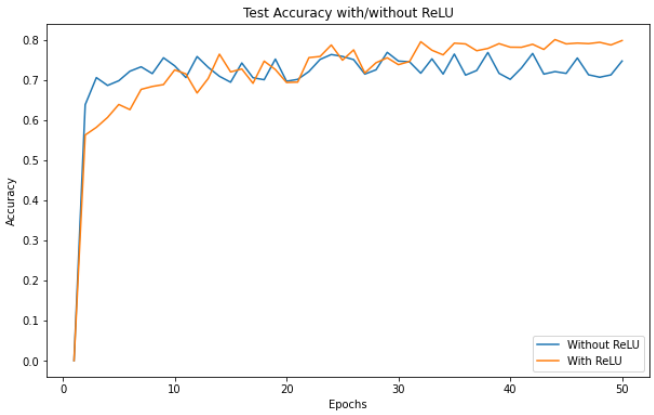
We do a comparison between the test accuracy of the 1 Layer Conv1D models with and without the ReLU layer. Both models are similar at the start of training, but the new model with ReLU layer converges to a higher test accuracy around the 40th epoch. This shows that the model could be optimized using 2 Layer Conv1D Network with ReLU activation function.

Number of trainable parameters = 1958455



	End of training	Best value
Train	0.9670006395224898	0.9667939175499584
Val	0.8706565354734056	0.8721791255289139
Test	0.7981842941443485	0.8005857051340715

Duration of training+evaluation = 8430s



We also added ReLU layers to the 2 Layer Conv1D model, after each of the convolutional layers. There is a great improvement in the test accuracy of the model.

In comparison with the 1 Layer Conv1D models, this model shows more fluctuations in the test accuracy throughout the training. However, the ReLU layer helped to stabilize the results as the number of epochs increased. This shows that the ReLU activation layer could be more effective for models that have more layers, due to the role of the layer in delinearizing the inputs for the next layer so that continuity between the layers is preserved as text data is non-linear.

2.4. Conclusion

In conclusion, for a single layer CNN in the network, it is observed that the F-score is significantly lower than that in a Bi-LSTM implementation of the word encoding. This is because CNN is unable to learn from sequences as well as an LSTM network. Furthermore, the CNN implemented is only in one direction. The kernel size set as 3 was able to learn from the sequence of 3 words each sentence to obtain the NER tags.

In this assignment, we have experimented with using 1-dimensional convolutional layers or 2-dimensional convolutional layers, and using a different number of layers for each experiment. We have also experimented with applying different types of layers, for example a max pooling layer or a ReLU layer and observed the results of the different neural network models.

The effects of a max pooling layer can also be noted as there is an increased performance of the model when the important features of the inputs are highlighted. Another way to improve the performance of the model is to apply a ReLU activation function after a convolutional layer.

We suggest a further improvement to the models trained previously. By changing the kernel size, we may be able to

observe how the window size affects the performance of the NER model.

3. Contribution

Question	Member
1	Kei Wuai, Zi Heng
2	Pooja, Wen Xiu

4. References

Sharma, A. (2020, July 7). Implementing Bengio's neural probabilistic language model (NPLM) using pytorch. Creed's Log Book. Retrieved November 9, 2021, from https://abhinavcreed13.github.io/blog/bengio-trigram-nplm-using-pytorch/.

Jayavardhan Reddy Peddamail. (2018, July 17). End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF. Retrieved November 1, 2021 from https://github.com/jayavardhanr/End-to-end-Sequence-Labeling-via-Bi-directional-LSTM-CNNs-CRF-Tutorial/blob/master/Named_Entity_Recognition-LSTM-CNN-CRF-Tutorial.ipynb.

Xuezhe Ma and Eduard Hovy. 2016. End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF . In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: LongPapers). Association for Computational Linguistics, Berlin, Germany https://arxiv.org/pdf/1603.01354.pdf.

Javier Fernandez. (2021, April). Conv1D and Conv2D: Did you realize that Conv1D Is a Subclass of Conv2D?. Retrieved November 9, 2021 from https://towardsdatascience.com/conv1d-and-conv2d-did-you-realize-that-conv1d-is-a-subclass-of-conv2d-8819675bec78

5. Appendix

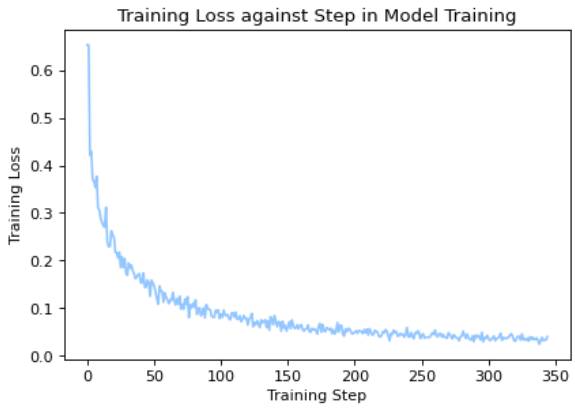
5.1 Model Summary for 2.3.2 Models

2 Layer Conv1D Network

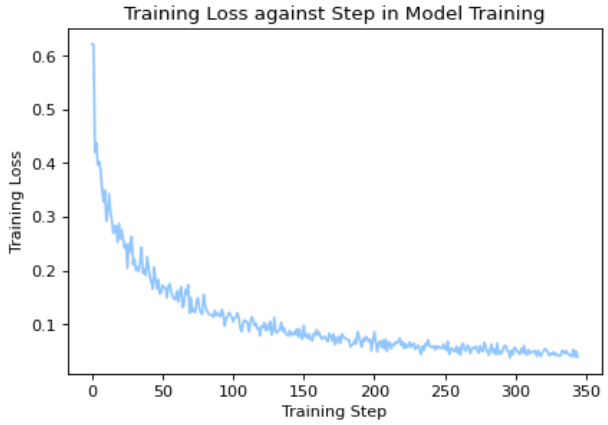
```
CNN_LSTM_CRF(
  (char_embeds): Embedding(75, 25)
  (char_cnn3): Conv2d(1, 25, kernel_size=(3, 25), stride=(1, 1), padding=(2, 0))
  (word_embeds): Embedding(17493, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (conv1): Conv1d(1, 125, kernel_size=(3, 125), stride=(1,), padding=(1, 0))
  (conv2): Conv1d(125, 400, kernel_size=(3, 1), stride=(1,), padding=(1, 0))
  (hidden2tag): Linear(in_features=400, out_features=19, bias=True)
)
```

5.2 Loss Graphs for 2.3.2 Models

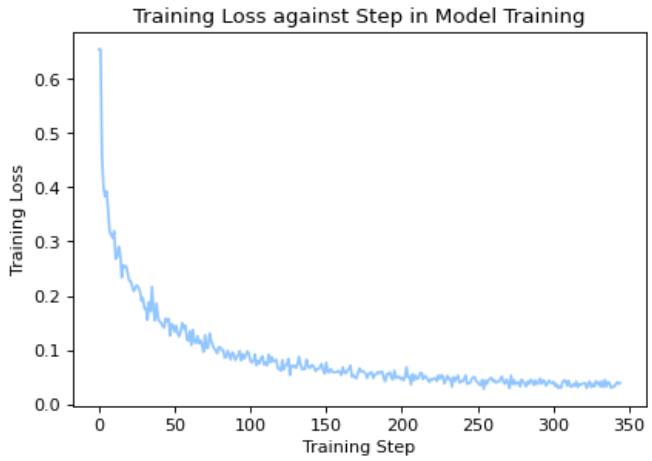
1 Layer Conv1D Network



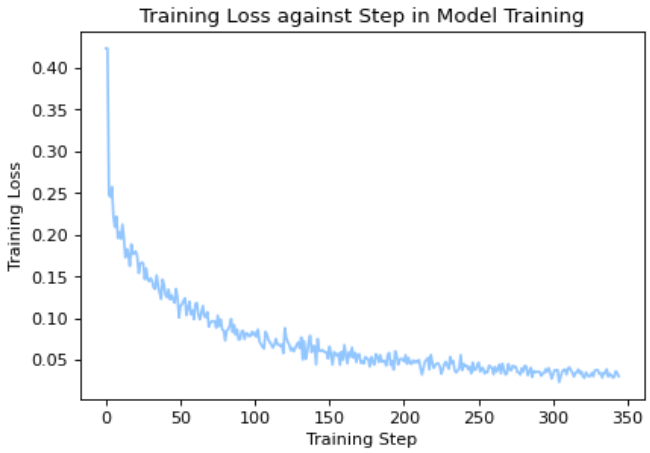
2 Layer Conv1D Network



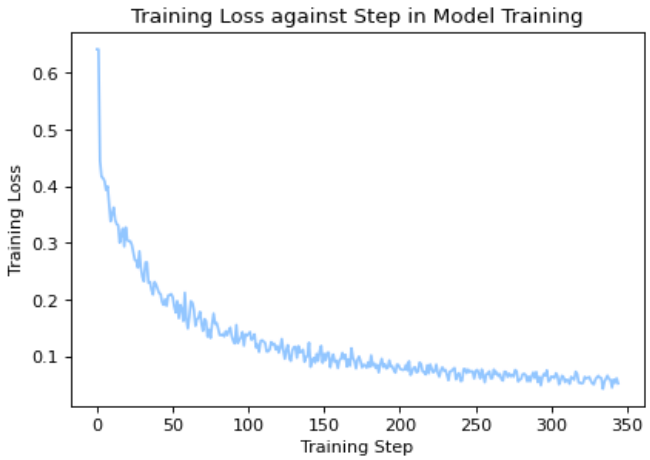
1 Layer Conv1D Network with Max Pooling layer



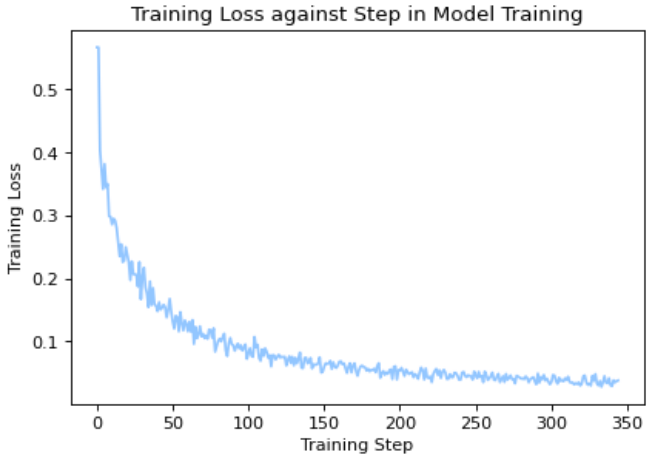
2 Layer Conv1D Network with Max Pooling layer



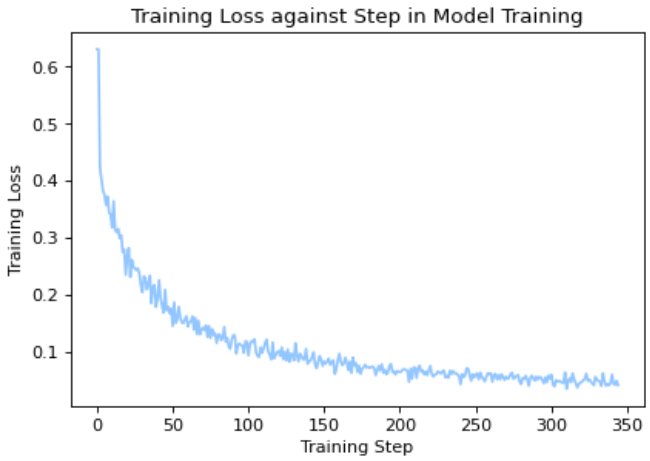
1 Layer Conv2D Network

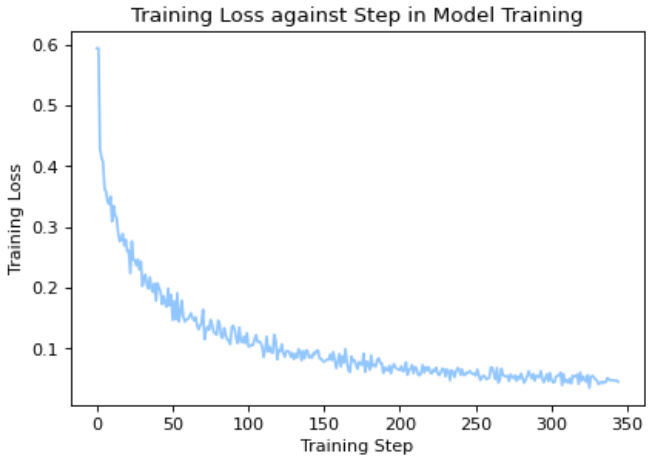


2 Layer Conv2D Network with Max Pooling layer



2 Layer Conv2D Network





3 Layer Conv2D Network