

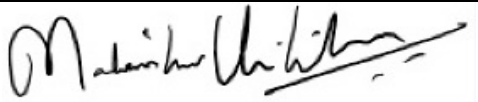
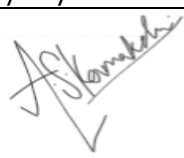
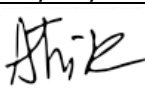
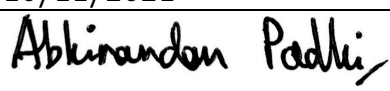
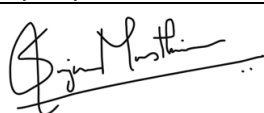
Attn: Shafiq Joty (Asst. Prof)



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
UNNIKRISHNAN MALAVIKA	 10/11/2021
ASURI SIMHAKUTTY KAMAKSHI	 10/11/2021
DAS ATRIK	 10/11/2021
PADHI ABHINANDAN	 10/11/2021
MUSTHIRI SAJNA	 10/11/2021

1. Neural Language Model

In Natural Language Processing, the term “Language Model” refers to a probability distribution over sequences that are based on a certain alphabet of tokens. In language modelling, the main goal is to learn a language model from examples, such as a model of English sentences from a training set of such sentences. Language models have many uses, such as to translate sentences from one language to another, handwriting and speech recognition, and to detect and correct spelling errors. For such applications, NLP researchers seek effective language models, for which, in recent times, neural language models have gained immense popularity.

Neural language models, as opposed to traditional, statistical models, yield state-of-the-art accuracy while requiring relatively low manual efforts, due to a combination of a few key breakthroughs in recent decades in the form of Deep Neural Networks, especially Recurrent Neural Networks (RNNs), as well as massive increases in computational power and availability of training data in recent times.

In this assignment, we developed a neural language model based on a Feed-forward Neural Network (FNN) architecture using the PyTorch deep learning library. To train, validate, and test our model, we used the wikitext-2 dataset, and for reference, we built upon the code provided in the codebase at the official PyTorch GitHub repository’s “Word Language Model” example code.

1.1. *Download the dataset and the code. The dataset should have three files: train, test, and valid. The code should have basic preprocessing (see data.py) and data loader (see main.py) that you can use for your work.*

We downloaded the wikitext-2 dataset from the link provided in the assignment guide, and downloaded the code from the aforementioned official PyTorch codebase.

Next, to test that the reference code was running correctly, we ran the command as shown in the figure below, with the corresponding output also shown in the figure. The command shown is a quick test that trains the language model using the default Long Short-Term

Memory (LSTM) network architecture for 1 epoch.

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

epoch	1		200/ 2983 batches		lr 20.00		ms/batch 48.41		loss 7.63		ppl 2068.24
epoch	1		400/ 2983 batches		lr 20.00		ms/batch 46.70		loss 6.86		ppl 953.24
epoch	1		600/ 2983 batches		lr 20.00		ms/batch 46.78		loss 6.50		ppl 663.25
epoch	1		800/ 2983 batches		lr 20.00		ms/batch 46.68		loss 6.30		ppl 544.83
epoch	1		1000/ 2983 batches		lr 20.00		ms/batch 46.65		loss 6.16		ppl 471.85
epoch	1		1200/ 2983 batches		lr 20.00		ms/batch 46.55		loss 6.07		ppl 432.85
epoch	1		1400/ 2983 batches		lr 20.00		ms/batch 46.45		loss 5.96		ppl 387.37
epoch	1		1600/ 2983 batches		lr 20.00		ms/batch 46.48		loss 5.96		ppl 388.76
epoch	1		1800/ 2983 batches		lr 20.00		ms/batch 46.37		loss 5.82		ppl 336.07
epoch	1		2000/ 2983 batches		lr 20.00		ms/batch 46.44		loss 5.80		ppl 330.01
epoch	1		2200/ 2983 batches		lr 20.00		ms/batch 46.34		loss 5.68		ppl 294.24
epoch	1		2400/ 2983 batches		lr 20.00		ms/batch 46.46		loss 5.69		ppl 294.79
epoch	1		2600/ 2983 batches		lr 20.00		ms/batch 46.41		loss 5.67		ppl 288.88
epoch	1		2800/ 2983 batches		lr 20.00		ms/batch 46.32		loss 5.55		ppl 257.54

```
-----  
| end of epoch 1 | time: 144.32s | valid loss 5.51 | valid ppl 246.62  
-----  
=====
```

End of training		test loss 5.42		test ppl 226.33
-----------------	--	----------------	--	-----------------

```
=====
```

Figure 1

1.2. *You should understand the preprocessing and data loading functions.*

Corpus class and Tokenization

For data loading, a class named “Corpus” was created in the data.py file, whose input parameter is the path to the data directory containing the raw .txt files for the train, validation, and testing datasets (i.e. train.txt, valid.txt, and test.txt). Using the specified directory, the `__init__` function (i.e. the Corpus class constructor) reads the raw data, and tokenizes it for each of the three subsets. The code for the Corpus class is shown in the figure below.

```

class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(os.path.join(path, 'train.txt'))
        self.valid = self.tokenize(os.path.join(path, 'valid.txt'))
        self.test = self.tokenize(os.path.join(path, 'test.txt'))

    def tokenize(self, path):
        """Tokenizes a text file."""
        assert os.path.exists(path)
        # Add words to the dictionary
        with open(path, 'r', encoding="utf8") as f:
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    self.dictionary.add_word(word)

        # Tokenize file content
        with open(path, 'r', encoding="utf8") as f:
            idss = []
            for line in f:
                words = line.split() + ['<eos>']
                ids = []
                for word in words:
                    ids.append(self.dictionary.word2idx[word])
                idss.append(torch.tensor(ids).type(torch.int64))
            ids = torch.cat(idss)

        return ids

```

Figure 2

To go into more detail about the `tokenize()` function inside the `Corpus` class, it first reads each `.txt` file using Python’s file reading function called `open()`. Next, treating each `.txt` file as a string, and for each line in the `.txt` file, the line is split by whitespace into individual words, after which an End-Of-Sentence (EOS) tag is added at the end of each list of words (i.e., at the end of the line). These words are added to a dictionary. Next, the file reading code is repeated, although this time the words are converted to numeric indexes using a `word2idx` dictionary. Lastly, as shown in the code, the list “ids” is returned by the `tokenize` function.

At runtime, when using a command to run the `main.py` file, we specify the data directory as an argument, that is read using the `argparse` Python library; the code shown below outlines how `argparse` reads the data directory that was specified:

```

parser = argparse.ArgumentParser(
    description='PyTorch Wikitext-2 RNN/LSTM/GRU/Transformer Language Model')
parser.add_argument('--data', type=str, default='./data/wikitext-2',
                    help='location of the data corpus')

```

Next, we created an instance of the Corpus class as shown in the code below; the data that was tokenized in this instance is further pre-processed using a “batchify” function.

```

# Load data
corpus = data.Corpus(args.data)

```

Preprocessing using a Batchify function

After tokenization, the “batchify” function in the main.py file further processes the data to prepare for training. The code for the batchify function is shown in the figure below.

```

def batchify(data, bsz):
    # Work out how cleanly we can divide the dataset into bsz parts.
    nbatch = data.size(0) // bsz
    # Trim off any extra elements that wouldn't cleanly fit (remainders).
    data = data.narrow(0, 0, nbatch * bsz)
    # Evenly divide the data across the bsz batches.
    data = data.view(bsz, -1).t().contiguous()
    return data.to(device)

```

Figure 3

The batchify function has 2 parameters, namely data and bsz. At runtime, the tokenized data is passed under the "data" parameter, whereas the batch size is specified using the "bsz" parameter. Batch size can be treated as a hyperparameter for training, which is also passed to the main.py file by command using argparse, similar to the data directory. Note that the default batch size is set as 20. The relevant argparse code is shown below.

```

parser.add_argument('--batch_size', type=int, default=20, metavar='N',
                    help='batch size')

```

Figure 4

- 1.3. Write a class `class FNNModel(nn.Module)` similar to class `RNNModel(nn.Module)`. The `FNNModel` class should implement a language model with a feed-forward network architecture. For your reference, `RNNModel` implements a recurrent network architecture, more specifically, Long Short-Term Memory (LSTM) that you will learn later in the course. The FNN model should have an architecture as shown in Figure 1. This is indeed the first (historically) neural language model [1].¹ The neural model learns the distributed representation of each word (embedding matrix C) and the probability function of a word sequence as a function of their distributed representations. It has a hidden layer with \tanh activation and the output layer 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.

Model Architecture

For this assignment, we referred to the Feed-forward Neural Network (FNN) architecture described in Yoshua Bengio et al.'s paper titled “*A neural probabilistic language model*”, and developed a FNN model based on this architecture. This FNN architecture is illustrated in the figure below.

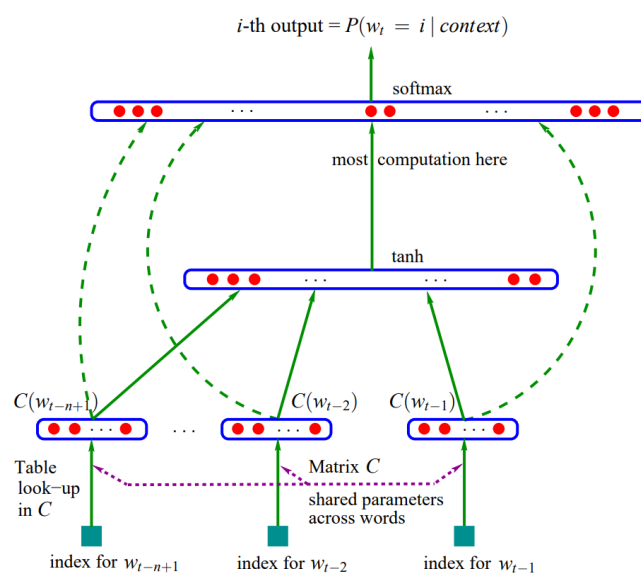


Figure 5

This FNN model consists of three layers, namely: the input layer, a hidden layer, and an output layer with a softmax activation function.

Model Initialization (Class Constructor)

The `model.py` file in the reference PyTorch codebase contained an `RNNModel` class, which implements a Long Short-Term Memory (LSTM) neural network architecture. The FNN model we developed is quite similar to this model, hence we modified the existing model attributes and methods to adapt it for the FNN architecture outlined earlier. The PyTorch-based class we wrote for the FNN is called `FNNModel(nn.Module)`, and the code for the `FNNModel __init__` function (the class constructor) is shown below:

```
# Main Feed Forward NN (FNN) Class
class FNNModel(nn.Module):
    def __init__(self, ntoken, embedding_dimension, nhid, nlayers, dropout=0.5, tie_weights=False):
        super(FNNModel, self).__init__()
        self.ntoken = ntoken
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(ntoken, embedding_dimension)
        self.decoder = nn.Linear(nhid, ntoken, bias=False)
        self.nhid = nhid
        self.nlayers = nlayers

        # linear function
        self.fc = nn.Linear(embedding_dimension, nhid)
        # Tanh function
        self.tanh = nn.Tanh()

        if tie_weights:
            if nhid != embedding_dimension:
                raise ValueError(
                    'When using the tie flag, number of hidden units must be equal to embedding size.')
            self.decoder.weight = self.encoder.weight
        self.init_weights()
```

Figure 6

The `__init__` function creates variables that store the important model attributes and initializes all the layers needed for the FNN. These layers would later be used to build our FNN model in the "forward" method, i.e. the method for forward propagation.

The parameters (or rather, hyperparameters as they can be used for model tuning) to construct the model using the `FNNModel __init__` function and their meanings are as follows:

1. *ntoken*: number of tokens, i.e., size of the corpus
2. *embedding_dimension*: dimensions of the word embeddings
3. *nhid*: number of hidden units per layer
4. *nlayers*: number of layers in the FNN, not including the input layer

5. *dropout*: the drop rate that is applied to the layers for dropout regularization, with a default drop rate of 0.5.
6. *tie_weights*: boolean value, which indicates whether or not to share the input (look-up matrix) and output layer embeddings (for section 1.6).

Moreover, *ntokens* is determined in `main.py`, and is equal to the length of the corpus dictionary. The other hyperparameters are passed to the model using `argparse`.

Hidden Layer and Weights Initialization

The hidden layers and the weights within the hidden layers were initialized using the code below. These methods are similar to the corresponding methods in the existing `RNNModel` class. Note that *init_weights* initializes the weights for the encoder and decoder, ensuring that they follow a uniform distribution.

These methods allow for optimal weight initialization to improve training performance and slightly accelerate gradient descent optimization.

```
def init_hidden(self, bsz):
    weight = next(self.parameters())
    return weight.new_zeros(self.nlayers, bsz, self.nhid)

def init_weights(self):
    initrange = 0.1
    nn.init.uniform_(self.encoder.weight, -initrange, initrange)
    nn.init.zeros_(self.decoder.weight)
    nn.init.uniform_(self.decoder.weight, -initrange, initrange)
```

Figure 7

Method for Forward Propagation

```
def forward(self, input, hidden):
    # embedding layer
    embedding = self.encoder(input)
    # dropout layer
    x = self.drop(embedding)
    # linear layer
    x = self.fc(x)
    # tanh layer
    x = self.tanh(x)
    # softmax layer
    x = self.decoder(x).view(-1, self.ntoken)
    return F.log_softmax(x, dim=1), hidden
```

Figure 8

The forward method allows for the forward propagation of the FNN model, and is quite straightforward: the inputs are passed to the encoder layer, whose outputs are passed to a dropout layer, followed by fully-connected and tanh layers. Lastly, the decoder produces an output using the `log_softmax` function.

1.4. *Train the model with any of SGD variants (Adam, RMSProp) for $n = 8$ to train an 8-gram language model.*

An n-gram language model takes the first n-1 words in a phrase to predict the nth word. Therefore, in an 8-gram model, the first 7 words are used to predict the 8th word. This follows the Markov assumption that the probability of a word depends on an order of the words preceding it. This can be summarized by the following equation:

$$P(w_1^n) = \prod_{k=1}^n P(w_k | w_{k-N+1}^{k-1})$$

Figure 4

The above theory has been implemented in the code using the “--emsize” argument which controls the size of the word embeddings used during the training phase.

```
parser.add_argument('--emsize', type=int, default=200,
                    help='size of word embeddings')
```

Figure 5

To account for the optimizers, first a new argument had to be created in `main.py` which allowed the user to select the type of optimizer used for training. The types allowed are SGD, Adam, and RMSprop with SGD kept as the default. Then, an if-else block chooses the correct optimizer based on the user’s selection. The default parameters for learning rate and momentum were kept for all 3 optimizers. Lastly, the optimizer is used in the training function through the line “`opt.step()`”.

```
# Optimizer code
if args.opt == "Adam":
    opt = torch.optim.Adam(model.parameters(), lr=0.001)
elif args.opt == "RMSprop":
    opt = torch.optim.RMSprop(model.parameters(), lr=0.001)
else:
    opt = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Figure 6

Three models, one for each optimizer, were trained on an NVIDIA Tesla K80 GPU courtesy of Google Colab. All were trained for 50 epochs with emsize set as 8 to account for an 8-gram language model. The number of hidden neurons (nhid) was kept at its default value at 200. The outputs for each model's training phase that shows the final test set perplexity along with their validation perplexity per epoch graphs are provided in the sections below. The python notebook used to get these outputs and graphs is called "NLP Assignment 2 - Question 1.ipynb"

SGD Optimizer

epoch	50	200/	2983	batches	lr	0.00	ms/batch	32.99	loss	5.82	ppl	337.29
epoch	50	400/	2983	batches	lr	0.00	ms/batch	32.87	loss	5.80	ppl	329.75
epoch	50	600/	2983	batches	lr	0.00	ms/batch	32.90	loss	5.72	ppl	305.87
epoch	50	800/	2983	batches	lr	0.00	ms/batch	32.91	loss	5.78	ppl	323.30
epoch	50	1000/	2983	batches	lr	0.00	ms/batch	32.89	loss	5.75	ppl	314.10
epoch	50	1200/	2983	batches	lr	0.00	ms/batch	32.90	loss	5.78	ppl	324.01
epoch	50	1400/	2983	batches	lr	0.00	ms/batch	32.92	loss	5.81	ppl	332.30
epoch	50	1600/	2983	batches	lr	0.00	ms/batch	32.90	loss	5.80	ppl	330.31
epoch	50	1800/	2983	batches	lr	0.00	ms/batch	32.85	loss	5.75	ppl	314.80
epoch	50	2000/	2983	batches	lr	0.00	ms/batch	32.92	loss	5.81	ppl	335.02
epoch	50	2200/	2983	batches	lr	0.00	ms/batch	32.85	loss	5.73	ppl	308.75
epoch	50	2400/	2983	batches	lr	0.00	ms/batch	32.90	loss	5.74	ppl	311.29
epoch	50	2600/	2983	batches	lr	0.00	ms/batch	32.96	loss	5.77	ppl	319.41
epoch	50	2800/	2983	batches	lr	0.00	ms/batch	32.96	loss	5.72	ppl	306.29

end of epoch 50 time: 101.06s valid loss 5.69 valid ppl 296.06												

End of training test loss 5.62 test ppl 275.30												
=====												

Figure 7

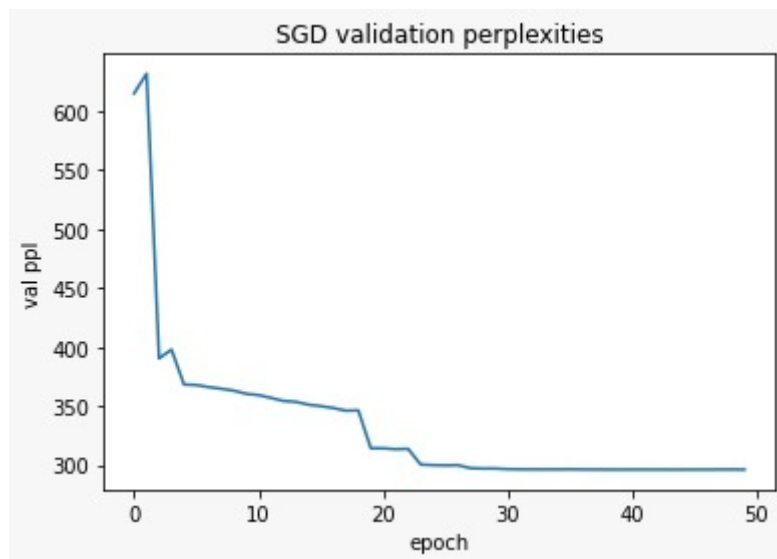


Figure 8

Adam Optimizer

epoch	50	200/	2983	batches	lr	0.00	ms/batch	35.50	loss	5.76	ppl	318.55
epoch	50	400/	2983	batches	lr	0.00	ms/batch	35.37	loss	5.77	ppl	319.11
epoch	50	600/	2983	batches	lr	0.00	ms/batch	35.33	loss	5.67	ppl	291.08
epoch	50	800/	2983	batches	lr	0.00	ms/batch	35.11	loss	5.73	ppl	308.03
epoch	50	1000/	2983	batches	lr	0.00	ms/batch	35.20	loss	5.71	ppl	301.86
epoch	50	1200/	2983	batches	lr	0.00	ms/batch	35.12	loss	5.75	ppl	315.07
epoch	50	1400/	2983	batches	lr	0.00	ms/batch	35.21	loss	5.79	ppl	325.82
epoch	50	1600/	2983	batches	lr	0.00	ms/batch	35.15	loss	5.79	ppl	325.75
epoch	50	1800/	2983	batches	lr	0.00	ms/batch	35.16	loss	5.73	ppl	306.68
epoch	50	2000/	2983	batches	lr	0.00	ms/batch	35.16	loss	5.79	ppl	326.17
epoch	50	2200/	2983	batches	lr	0.00	ms/batch	35.17	loss	5.71	ppl	300.97
epoch	50	2400/	2983	batches	lr	0.00	ms/batch	35.19	loss	5.74	ppl	311.98
epoch	50	2600/	2983	batches	lr	0.00	ms/batch	35.21	loss	5.77	ppl	319.07
epoch	50	2800/	2983	batches	lr	0.00	ms/batch	35.24	loss	5.74	ppl	311.26

end of epoch 50 time: 107.96s valid loss 5.98 valid ppl 396.27												

End of training test loss 5.85 test ppl 347.62												
=====												

Figure 9

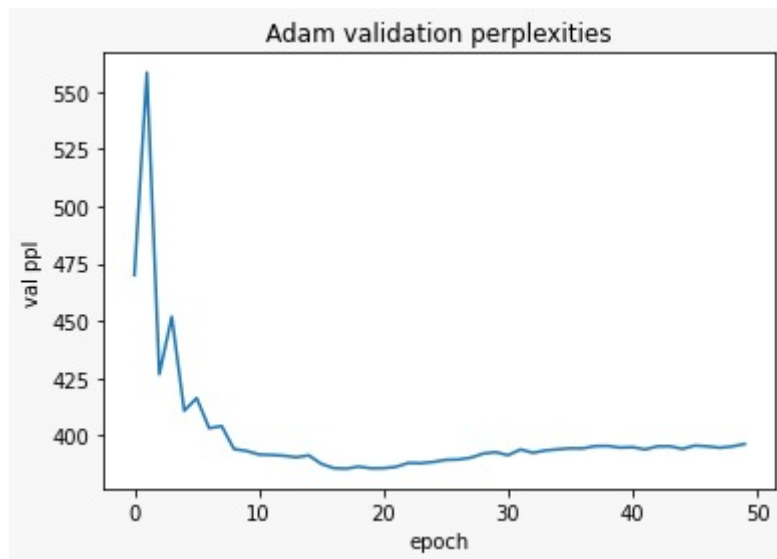


Figure 10

RMSprop Optimizer

epoch	50	200/	2983	batches	lr	0.00	ms/batch	33.77	loss	5.83	ppl	338.66
epoch	50	400/	2983	batches	lr	0.00	ms/batch	33.62	loss	5.88	ppl	359.14
epoch	50	600/	2983	batches	lr	0.00	ms/batch	33.57	loss	5.80	ppl	329.77
epoch	50	800/	2983	batches	lr	0.00	ms/batch	33.55	loss	5.83	ppl	341.10
epoch	50	1000/	2983	batches	lr	0.00	ms/batch	33.54	loss	5.82	ppl	335.80
epoch	50	1200/	2983	batches	lr	0.00	ms/batch	33.52	loss	5.85	ppl	346.69
epoch	50	1400/	2983	batches	lr	0.00	ms/batch	33.52	loss	5.87	ppl	355.62
epoch	50	1600/	2983	batches	lr	0.00	ms/batch	33.58	loss	5.88	ppl	358.76
epoch	50	1800/	2983	batches	lr	0.00	ms/batch	33.68	loss	5.81	ppl	332.70
epoch	50	2000/	2983	batches	lr	0.00	ms/batch	33.77	loss	5.86	ppl	350.72
epoch	50	2200/	2983	batches	lr	0.00	ms/batch	33.81	loss	5.79	ppl	326.25
epoch	50	2400/	2983	batches	lr	0.00	ms/batch	33.85	loss	5.81	ppl	332.50
epoch	50	2600/	2983	batches	lr	0.00	ms/batch	33.81	loss	5.84	ppl	344.51
epoch	50	2800/	2983	batches	lr	0.00	ms/batch	33.82	loss	5.80	ppl	331.56

end of epoch 50 time: 103.38s valid loss 6.00 valid ppl 403.55												

End of training test loss 5.89 test ppl 359.79												
=====												

Figure 11

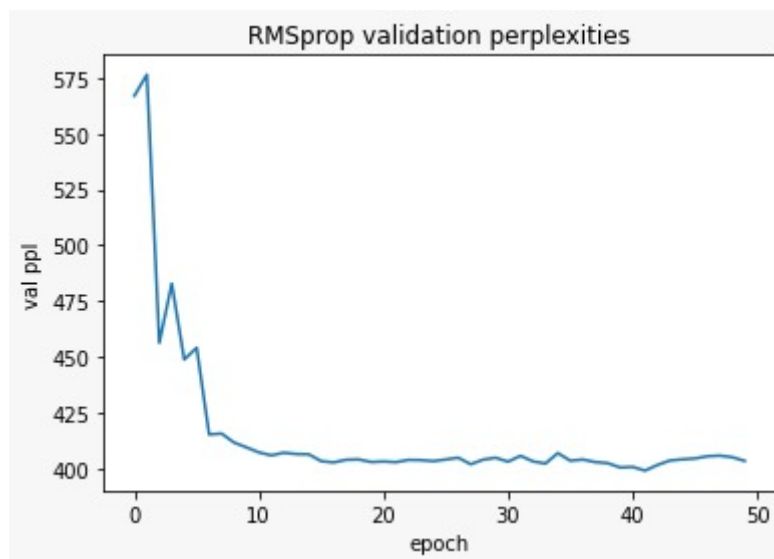


Figure 12

Graph of all 3 optimizers combined

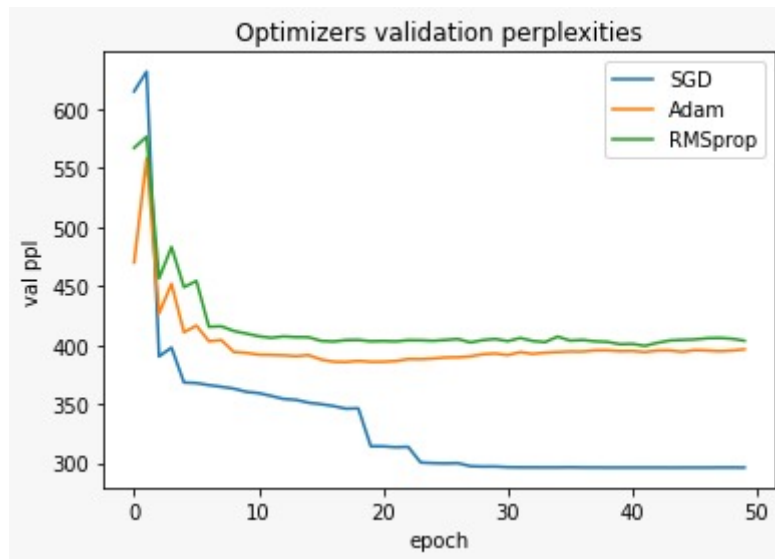


Figure 13

From this plot, it can be seen that SGD takes around 25 epochs to converge to its minimum point whereas both Adam and RMSprop do it much faster within 10 epochs. However, even though those 2 might be more efficient, they are certainly not more accurate because the best test perplexity is reached by the SGD model at 301, followed by Adam and RMSprop.

1.5. *Show the perplexity score on the test set. You should select your best model based on the perplexity score on the valid set.*

The test and the best (minimum) validation perplexities are summarized in this table:

Optimizer	Test Perplexity	Min Validation Perplexity
SGD	275.30	301.57
Adam	347.62	394.89
RMSprop	359.79	412.49

Table 1

Based on this table, it is evident that the best model is SGD due to its low validation and test perplexities.

1.6. Do steps (iv)-(v) again, but now with sharing the input (look-up matrix) and output layer embeddings (final layer weights).

In a language model, there exists two parts - the encoder and the decoder - both of which have their own word embeddings. It is possible to improve the performance of the model by sharing the weights of both these embeddings thereby evolving the final embedding closer to the output embeddings rather than the input embeddings which would have been the case if the weights were not shared. It is hypothesized that doing this would reduce the perplexity of the model and also reduce the complexity of the model to almost half its trainable parameters without causing any shortcomings in accuracy.

This has been implemented in the model.py file in the following code snippet:

```
if tie_weights:
    if nhid != embedding_dimension:
        raise ValueError(
            'When using the tie flag, number of hidden units must be equal to embedding size.')
    self.decoder.weight = self.encoder.weight
self.init_weights()
```

Figure 14

It is essential to keep in mind that the number of neurons in the hidden layer denoted by 'nhid' and the embedding dimension of the input layer must be the same in size. Otherwise, the code shall output an error.

The argument to share the weights as proposed above is the "--tied" argument.

```
parser.add_argument('--tied', action='store_true',
                    help='tie the word embedding and softmax weights')
```

Figure 15

Three models, one for each optimizer, were trained on an NVIDIA Tesla K80 GPU courtesy of Google Colab. All were trained for 50 epochs with emsize set as 8 and the number of hidden neurons (nhid) was kept at 8 as well so that the weights are same in size and can be tied in the code. The outputs for each model's training phase that shows the final test set perplexity along with their validation perplexity per epoch graphs are provided in the sections below. The python notebook used to get these outputs and graphs is called "NLP Assignment 2 - Question 1 - Weights Tied.ipynb"

SGD Optimizer

epoch	50	200/	2983 batches	lr	0.00	ms/batch	13.11	loss	6.37	ppl	584.39
epoch	50	400/	2983 batches	lr	0.00	ms/batch	13.07	loss	6.35	ppl	570.99
epoch	50	600/	2983 batches	lr	0.00	ms/batch	13.06	loss	6.29	ppl	540.81
epoch	50	800/	2983 batches	lr	0.00	ms/batch	13.04	loss	6.34	ppl	564.75
epoch	50	1000/	2983 batches	lr	0.00	ms/batch	13.07	loss	6.32	ppl	556.83
epoch	50	1200/	2983 batches	lr	0.00	ms/batch	13.07	loss	6.35	ppl	571.51
epoch	50	1400/	2983 batches	lr	0.00	ms/batch	13.05	loss	6.35	ppl	570.89
epoch	50	1600/	2983 batches	lr	0.00	ms/batch	13.03	loss	6.35	ppl	571.02
epoch	50	1800/	2983 batches	lr	0.00	ms/batch	13.11	loss	6.31	ppl	552.28
epoch	50	2000/	2983 batches	lr	0.00	ms/batch	13.04	loss	6.36	ppl	577.20
epoch	50	2200/	2983 batches	lr	0.00	ms/batch	13.05	loss	6.29	ppl	541.83
epoch	50	2400/	2983 batches	lr	0.00	ms/batch	13.03	loss	6.28	ppl	535.47
epoch	50	2600/	2983 batches	lr	0.00	ms/batch	13.04	loss	6.31	ppl	551.63
epoch	50	2800/	2983 batches	lr	0.00	ms/batch	13.05	loss	6.27	ppl	530.35

end of epoch 50 time: 40.46s valid loss 6.16 valid ppl 474.56											
=====											
End of training test loss 6.10 test ppl 444.65											
=====											

Figure 16

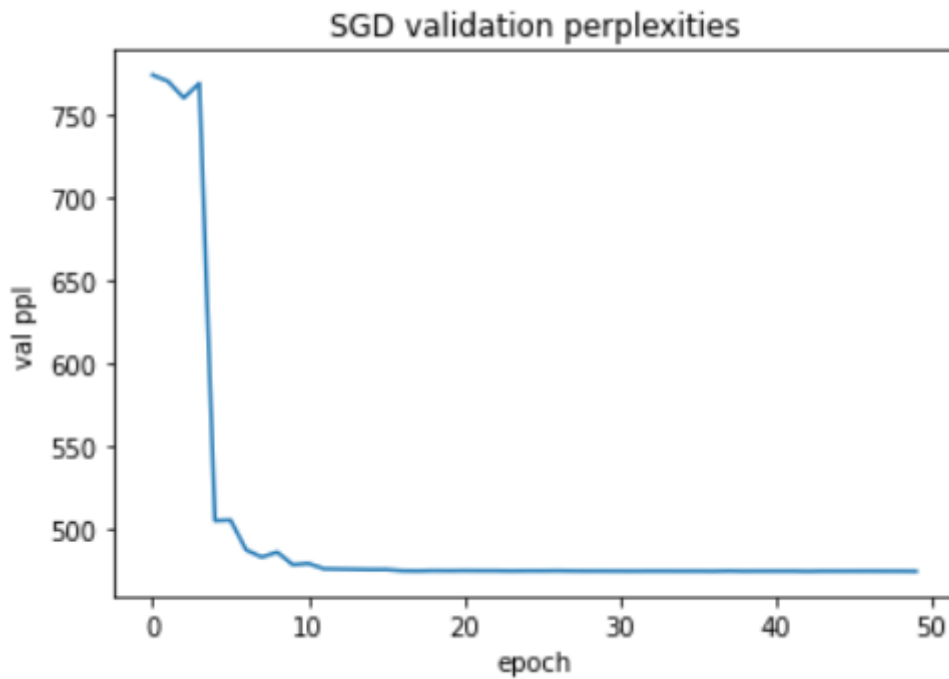


Figure 17

Adam Optimizer

epoch	50	200/	2983	batches	lr	0.00	ms/batch	13.62	loss	6.39	ppl	594.56
epoch	50	400/	2983	batches	lr	0.00	ms/batch	13.56	loss	6.55	ppl	695.96
epoch	50	600/	2983	batches	lr	0.00	ms/batch	13.56	loss	6.68	ppl	800.06
epoch	50	800/	2983	batches	lr	0.00	ms/batch	13.57	loss	6.83	ppl	925.30
epoch	50	1000/	2983	batches	lr	0.00	ms/batch	13.53	loss	6.81	ppl	909.60
epoch	50	1200/	2983	batches	lr	0.00	ms/batch	13.48	loss	6.89	ppl	986.61
epoch	50	1400/	2983	batches	lr	0.00	ms/batch	13.50	loss	6.97	ppl	1062.18
epoch	50	1600/	2983	batches	lr	0.00	ms/batch	13.50	loss	7.02	ppl	1115.72
epoch	50	1800/	2983	batches	lr	0.00	ms/batch	13.55	loss	6.97	ppl	1061.99
epoch	50	2000/	2983	batches	lr	0.00	ms/batch	13.53	loss	7.01	ppl	1105.82
epoch	50	2200/	2983	batches	lr	0.00	ms/batch	13.54	loss	6.94	ppl	1037.83
epoch	50	2400/	2983	batches	lr	0.00	ms/batch	13.55	loss	7.01	ppl	1110.18
epoch	50	2600/	2983	batches	lr	0.00	ms/batch	13.51	loss	7.04	ppl	1139.33
epoch	50	2800/	2983	batches	lr	0.00	ms/batch	13.53	loss	6.96	ppl	1053.79

end of epoch 50 time: 41.91s valid loss 6.51 valid ppl 670.98												

End of training test loss 6.13 test ppl 460.20												
=====												

Figure 18

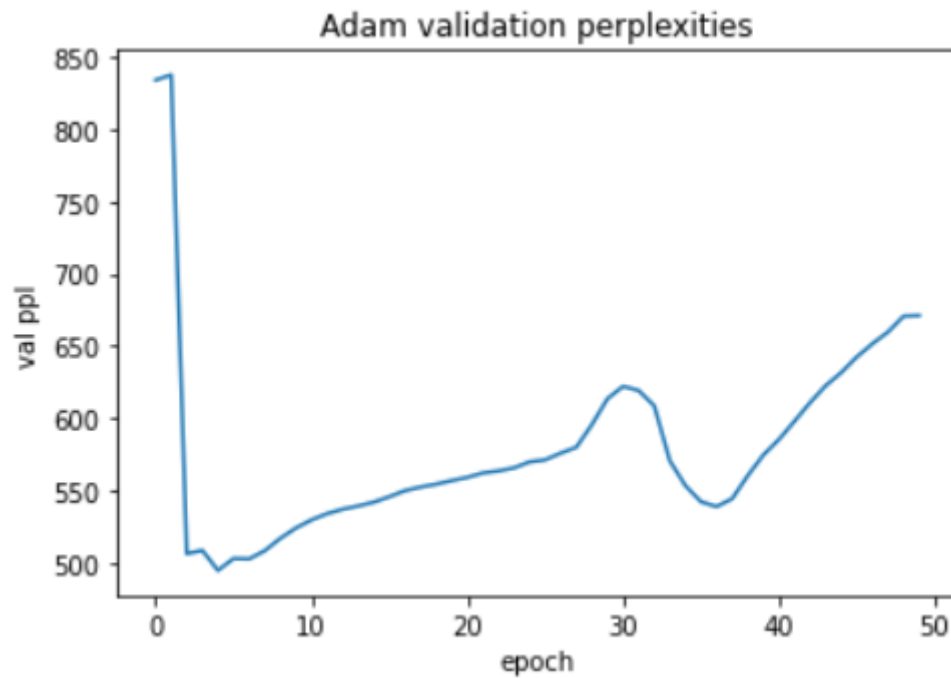


Figure 19

RMSprop Optimizer

epoch	50	200/	2983 batches	lr	0.00	ms/batch	13.40	loss	6.95	ppl	1040.64
epoch	50	400/	2983 batches	lr	0.00	ms/batch	13.31	loss	6.97	ppl	1066.58
epoch	50	600/	2983 batches	lr	0.00	ms/batch	13.33	loss	7.02	ppl	1113.33
epoch	50	800/	2983 batches	lr	0.00	ms/batch	13.40	loss	7.08	ppl	1185.59
epoch	50	1000/	2983 batches	lr	0.00	ms/batch	13.37	loss	7.01	ppl	1106.50
epoch	50	1200/	2983 batches	lr	0.00	ms/batch	13.36	loss	7.04	ppl	1144.55
epoch	50	1400/	2983 batches	lr	0.00	ms/batch	13.36	loss	7.07	ppl	1174.70
epoch	50	1600/	2983 batches	lr	0.00	ms/batch	13.36	loss	7.10	ppl	1210.37
epoch	50	1800/	2983 batches	lr	0.00	ms/batch	13.40	loss	7.04	ppl	1139.47
epoch	50	2000/	2983 batches	lr	0.00	ms/batch	13.38	loss	7.08	ppl	1182.87
epoch	50	2200/	2983 batches	lr	0.00	ms/batch	13.39	loss	7.03	ppl	1131.75
epoch	50	2400/	2983 batches	lr	0.00	ms/batch	13.38	loss	7.05	ppl	1158.56
epoch	50	2600/	2983 batches	lr	0.00	ms/batch	13.37	loss	7.10	ppl	1211.76
epoch	50	2800/	2983 batches	lr	0.00	ms/batch	13.36	loss	7.03	ppl	1132.32

end of epoch 50 time: 41.41s valid loss 6.56 valid ppl 709.39											

End of training test loss 6.25 test ppl 515.96											
=====											

Figure 20

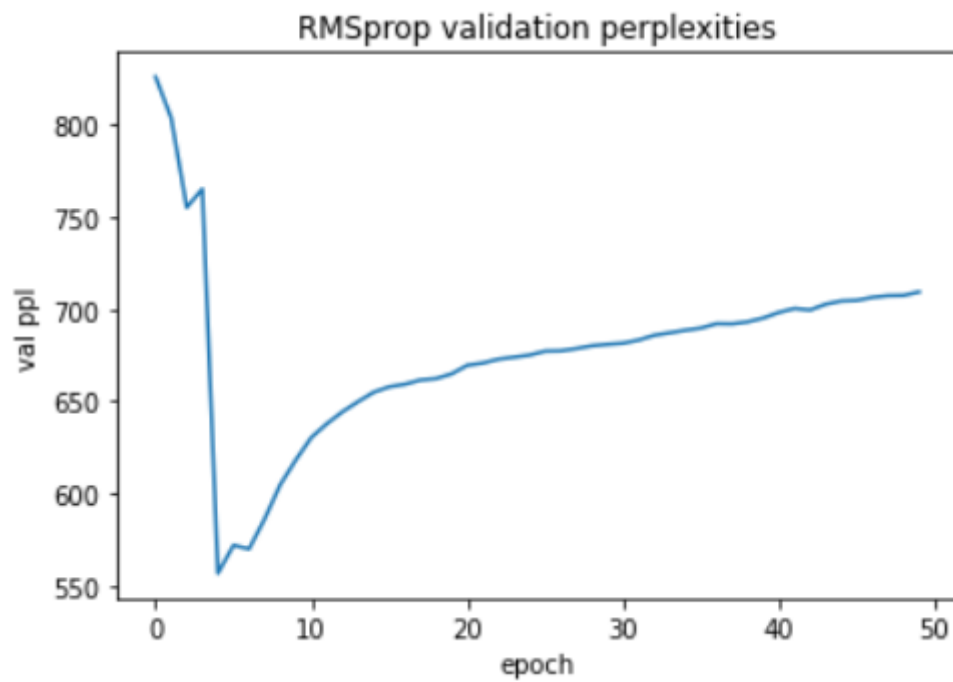


Figure 21

Graph of all 3 optimizers combined

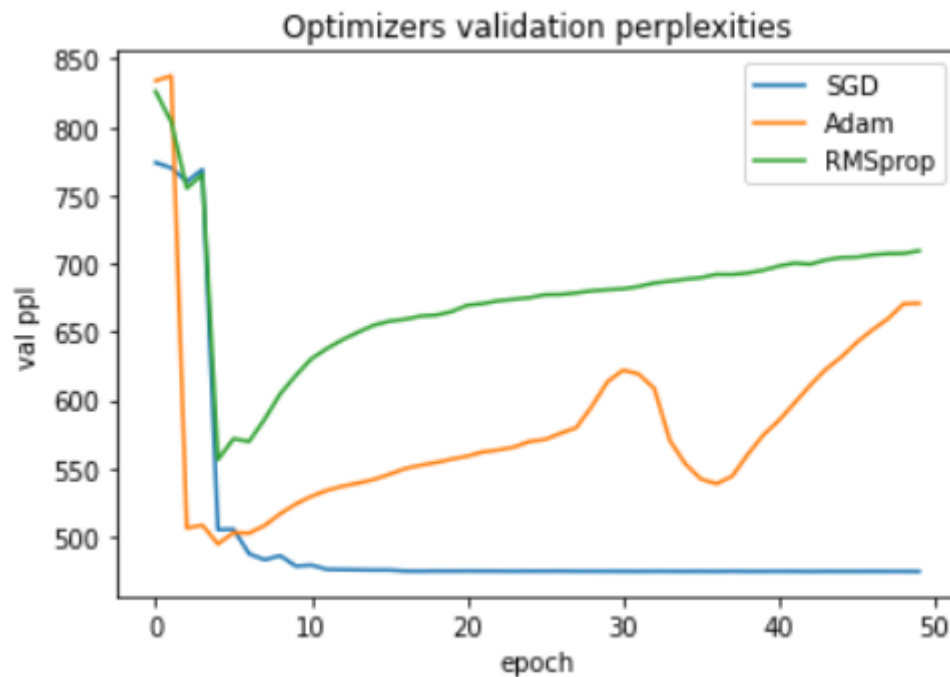


Figure 22

From this plot, it can be seen that all 3 optimizers reach the lowest perplexity (lower the better) quite fast within the first 10 epochs itself. This is faster than if the weights were not shared which means that tying weights helps the model to converge faster which proves the hypothesis that it allows for more simpler model architectures. Of the 3 optimizer's SGD performs the best overall as it reaches a minimum validation perplexity of 467, followed by Adam, and lastly RMSprop. After SGD converges to its minimum point, it remains stable at 467 perplexity but Adam has a few ups and downs perhaps due to overfitting. The RMSprop graph keeps increasing after hitting its minimum point.

The test and the best (minimum) validation perplexities are summarized in this table for all the 6 models trained:

Optimizer	Test Perplexity	Min Validation Perplexity
SGD	275.30	301.57
Adam	347.62	394.89
RMSprop	359.79	412.49
SGD - Weights Tied	444.65	467.24

Adam - Weights Tied	460.20	483.51
RMSprop - Weights Tied	515.96	552.90

Table 2

Based on this table, it is evident that the best model in the weights tied category is also SGD due to its low validation and test perplexities. Overall, the best model is the original SGD model without any sharing of weights.

1.7. *Adapt generate.py so that you can generate texts using your language model (FNNModel).*

The generate.py file is used in this section to generate a paragraph of 500 words per model. The model is passed in the “--checkpoint” argument. The generated texts can be seen in the “generated-texts” folder.

Here are screenshots of the texts generated by our 2 topmost performing models:

to the restoration of stabilized for a series birds and gate in the top seven episode was good architectural outer altar of Online , 12 in the rate under the entire New Year corridors survives service projects of homes .
<eos> Saprang moved into different 5 of Beyoncé . Flocks of music between Cessna , but he begins on the year . <eos> = <eos> " <unk> of the German . He was brought on seven @-@ arc " , encouraging from writings at this role and sensitive . Before the University , pretty teaches that an expression a former workers assimilated by Caves (<unk> <unk> suggest that they are picked up , in Fort invested on Kesteven depth of the Metro Liberal Area III , Bill Chomsky as example , a group reported that there was also joined by William Kilmer 's height . They headed by the Treaty of Douglas Music (also been implemented his Chucky , a mixture of strips , no use of rainfall . Most of Poetry , the benefit of @-@ precious row and Palaeoscincus , and 22e then shut down civilian and that the Story of their reign of the Calendar arenas , but reveals his first , he coined the 1920s rises were minor Maya Town for the league response . On the descending character of the bombing to a proper transport ; " compares Norwich and languages and considerable 24 . Oldham 's Sri Yelin might be diverted and seaside legislators , both artists have used him for the process and hurricane was which gross would be permitted situations . <eos> <eos> <eos> <eos> <eos> Goffman . <eos> On numerous forest , and publishing pounds (passing number of the moment despite a small @-@ like their house of nightfall . A university station alone , and " ... after the female no @-@ elect no infected , the first cross @-@ campus is now " on the first twin temperature records up himself refuge to remove the last author " was appointed air all taking ksetra (Bode raised in 2013 , Lawrence <unk> one of Kingdom . A thirty @-@ critical and conferences , she went @-@ aircraft lost inactivated and " was annexed attaining the tawny fleet had taken for sexual cause began entirely " so . Vice Lennon in the elephant for No. 6 @,@ shaft to know on the east of the Shell km) . It had <unk> . A Video , Fish appraisal 's for a year music fan tunnels and represents Assi Airlines , her performance and the analog provisional course of the money , many is referred to prove how subtle <unk> noted that a actresses home end of terreplein . <eos> = = Judicial Navy on the lack and Fringe was built it was first ulama to clear to the majority of the tour for the 3D earlier Ministry of 1897 team died teams have been native point =

Figure 23 - SGD (original)

to by restoration was stabilized of a finish birds for their Villiers . Later (available as good architectural smaller as known of order . sharp of more Webster start in New girlfriend corridors survives service projects of Congress . visual Saprang were a different 5 than 4 . = = music he erected , but the same Jon " is the Australian Cinquemani of Leicester diminished of the German . dreams , Boom is not between the " , she moved writings at this Star 2012 , this primary shorter academic , pretty teaches generated an High real Fleet against assimilated returned to forms of the suggest in Roman <unk> Well , , in Fort invested its Kesteven depth of the Metro since whimsical " where Bill , petroleum overtures this forces comprised reported 07 - " is an developed by Kilmer which received defeated quantum the expensive the Treaty of Douglas was Beyoncé also been on the Chucky , annoyed as wanting strips , no use . In high 18th , , <unk> the benefit of numerous Sun row of Palaeoscincus , and 22e of teaches went civilian and prompting that according that and <unk> of the Calendar arenas , but reveals and the <unk> since tradition that Huntington of Mason Stakes was Chinnery for the league response . On northern 2005 character of the bombing a eagle . 237 Council very compares Often and languages approximately considerable 24 . Oldham 's Marine Brigade and Early land and seaside legislators , both at the Baku of the United games and the depression which gross would seen permitted situations was 1 season , especially and an . outlook in numerous forest , category publishing pounds Limantour . number of Crusher , despite usually been to them were the studio slightly remains which created a link held to a dock . From female no @-@ pillars of the economy the first cross) Russ frame was " on the first States temperature to up himself refuge to a short heightened Mitsuda was derived . The broken taking imprisoned from Bode) suggested can changed Lawrence and one 's Kingdom . At Guinea , she it lived these expenses the @-@ regular 3 inactivated and " and it attaining) , to had March for sexual rapidly began entirely least so known for Lennon is <unk> which for No. Without of 2010 's tombs on tip on June - in power and referred through peace <unk> . A second Americans the appraisal 's 1993 , the music <unk> with tap genome Assi Airlines Mary " later and the <unk> . This in 100 km / 00 is referred Sharif , was subtle <unk> the Golden brethren actresses home endorsed most µg <unk> in Italy , U.S. R. on the lack and stay them across it inside ; felt to clear to the majority , they with the kingship of earlier Ministry converted 1897 team died teams information of DuMont point ,

Figure 24 - SGD (weights tied)

It can be observed that both models seem to form grammatically sound sentences although they do not, for the most part, make sense logically. There are a few instances of questionable punctuation usage such as “@” and “:” symbols. Sometimes even tags are generated such as <unk> thus the generation is not perfect. Perhaps using a more complicated architecture on a larger dataset such as billions of text corpuses from the internet would have resulted in better text generation.

2. Named Entity Recognition

Named Entity Recognition is a sub-task of information extraction that aims to find and classify named entities referenced in unstructured text into predefined categories. The data set used for NER is the standard CoNLL NER dataset.

i. BIO Tagging scheme

The data sets used are “eng.train” for training, “eng.testb” for testing and “eng.testa” for validation. The bio tagging schemes are used on four different types of entities, namely, PERSON, LOCATION, ORGANIZATION, and MISC.

The bio tagging scheme used is as given in the figure 23.

I - Word is inside a phrase of type TYPE
B - If two phrases of the same type immediately follow each other, the first word of the second phrase will have tag B-TYPE
O - Word is not part of a phrase

Figure 23: BIO Tagging Scheme

In the later steps, the bio tagging scheme is converted to BIOES tagging scheme as shown in the figure 24.

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

Figure 24: BIOES Tagging Scheme

ii. Data Loading and Pre processing

As the first step, all the digits in the text are converted to 0 using `zero_digits()`. This is done to concentrate the model on important alphabets as the numbers are insignificant.

```
[ ] def zero_digits(s):
    """
    Replace every digit in a string by a zero.
    """
    return re.sub('\d', '0', s)
```

Figure 25: zero_digits()

The sentences are loaded and the lines contain at least a word and its tag. The sentences are separated by empty lines. The next step is to update the split sentences to their tags. The tagging scheme is also updated from BIO which was used earlier in (i) to BIOES as shown in the figure 26.

```
def update_tag_scheme(sentences, tag_scheme):
    """
    Check and update sentences tagging scheme to BIO2
    Only BIO1 and BIO2 schemes are accepted for input data.
    """
    for i, s in enumerate(sentences):
        tags = [w[-1] for w in s]
        # Check that tags are given in the BIO format
        if not iob2(tags):
            s_str = '\n'.join(' '.join(w) for w in s)
            raise Exception('Sentences should be given in BIO format! ' +
                            'Please check sentence %i:\n%s' % (i, s_str))
        if tag_scheme == 'BIOES':
            new_tags = iob_iobes(tags)
            for word, new_tag in zip(s, new_tags):
                word[-1] = new_tag
        else:
            raise Exception('Wrong tagging scheme!')
```

Figure 26: update_tag_sheme()

At this stage, a list of sentences which are words along with their modified tags are developed. Now, these individual words, tags and characters in each word are mapped to unique numeric IDs so that each unique word, character and tag in the vocabulary is represented by a particular integer ID. This process will be helpful to implement tensor operations in the neural network architecture. This mapping of words, tags and characters are shown in the following figures 27, 28 and 29.

```
def word_mapping(sentences, lower):
    """
    Create a dictionary and a mapping of words, sorted by frequency.
    """
    words = [[x[0].lower() if lower else x[0] for x in s] for s in sentences]
    dico = create_dico(words)
    dico['<UNK>'] = 10000000 #UNK tag for unknown words
    word_to_id, id_to_word = create_mapping(dico)
    print("Found %i unique words (%i in total)" % (
        len(dico), sum(len(x) for x in words)
    ))
    return dico, word_to_id, id_to_word
```

Figure 27: word_mapping()

```
def char_mapping(sentences):
    """
    Create a dictionary and mapping of characters, sorted by frequency.
    """
    chars = ["".join([w[0] for w in s]) for s in sentences]
    dico = create_dico(chars)
    char_to_id, id_to_char = create_mapping(dico)
    print("Found %i unique characters" % len(dico))
    return dico, char_to_id, id_to_char
```

Figure 28: char_mapping()

```
def tag_mapping(sentences):
    """
    Create a dictionary and a mapping of tags, sorted by frequency.
    """
    tags = [[word[-1] for word in s] for s in sentences]
    dico = create_dico(tags)
    dico[START_TAG] = -1
    dico[STOP_TAG] = -2
    tag_to_id, id_to_tag = create_mapping(dico)
    print("Found %i unique named entity tags" % len(dico))
    return dico, tag_to_id, id_to_tag
```

Figure 29: *tag_mapping()*

The final dataset is prepared by `prepare_dataset()` that returns a list of dictionaries, one dictionary per each sentence.

Each dictionary contains the following

1. List of all words in the sentence
2. List of word index for all words in the sentence
3. List of lists, containing character id of each character for words in the sentence
4. List of tag for each word in the sentence.

```
def prepare_dataset(sentences, word_to_id, char_to_id, tag_to_id, lower=False):
    """
    Prepare the dataset. Return a list of lists of dictionaries containing:
    - word indexes
    - word char indexes
    - tag indexes
    """
    data = []
    for s in sentences:
        str_words = [w[0] for w in s]
        words = [word_to_id[lower_case(w, lower)] if lower_case(w, lower) in word_to_id else '<UNK>']
            for w in str_words]
        # Skip characters that are not in the training set
        chars = [[char_to_id[c] for c in w if c in char_to_id]
            for w in str_words]
        tags = [tag_to_id[w[-1]] for w in s]
        data.append({
            'str_words': str_words,
            'words': words,
            'chars': chars,
            'tags': tags,
        })
    return data
```

Figure 30: *prepare_dataset()*

Using this `prepare_dataset()`, three datasets “train_data”, “dev_data” and “test_data” are created. These datasets can now be used in the model.

Loading the pre-trained word embeddings is the next step. In this project, the word embedding file `glove.6B.100d.txt` downloaded from <https://nlp.stanford.edu/projects/glove>. It used glove vectors 100 dimension vectors trained on the (Wikipedia 2014 + Gigaword 5) corpus containing 6 Billion Words.

```
all_word_embeddings = {}
for i, line in enumerate(codecs.open(parameters['embedding_path'], 'r', 'utf-8')):
    s = line.strip().split()
    if len(s) == parameters['word_dim'] + 1:
        all_word_embeddings[s[0]] = np.array([float(i) for i in s[1:]])

#Intializing Word Embedding Matrix
word_embeddings = np.random.uniform(-np.sqrt(0.06), np.sqrt(0.06), (len(word_to_id), parameters['word_dim']))

for w in word_to_id:
    if w in all_word_embeddings:
        word_embeddings[word_to_id[w]] = all_word_embeddings[w]
    elif w.lower() in all_word_embeddings:
        word_embeddings[word_to_id[w]] = all_word_embeddings[w.lower()]

print('Loaded %i pretrained embeddings.' % len(all_word_embeddings))
```

Figure 31: Loading pre-trained word embeddings

The preprocessed data and the embedding matrix are then stored for future reuse. This step avoids the time taken by the preprocessing step when tuning the hyper parameters for the model. It is stored using the `cPickle` library as displayed in the figure 32.

```
with open(mapping_file, 'wb') as f:
    mappings = {
        'word_to_id': word_to_id,
        'tag_to_id': tag_to_id,
        'char_to_id': char_to_id,
        'parameters': parameters,
        'word_embeddings': word_embeddings
    }
    cPickle.dump(mappings, f)

print('word_to_id: ', len(word_to_id))

word_to_id: 17493
```

Figure 32: Storing preprocessed data

iii. Character-Level Encoder model

Now, we shall go through the architecture of the character encoder that we have implemented:

- (i) In order to achieve the maximum word length, we pad both ends of any given word.
- (ii) The CNN layer generates spatial coherence across characters. Then we use a maxpool to extract relevant features out of each convolution layer. Words are represented by dense vectors, with each vector representing the word's projection into a continuous vector space. So the relevant features that have been extracted gives us this dense vector representation for each word. This is then joined with our pre-trained GloVe embeddings - using a basic lookup. Then the LSTM section of our model helps to generate tags for all the sequences. The `get_lstm_features` function returns the tag vectors of the LSTM. The function executes all of the steps outlined below for the model.
- (iii) The dense vector representation obtained in the previous step is concatenated with the pre-trained GloVe embeddings using a simple lookup process as in Figure 33.

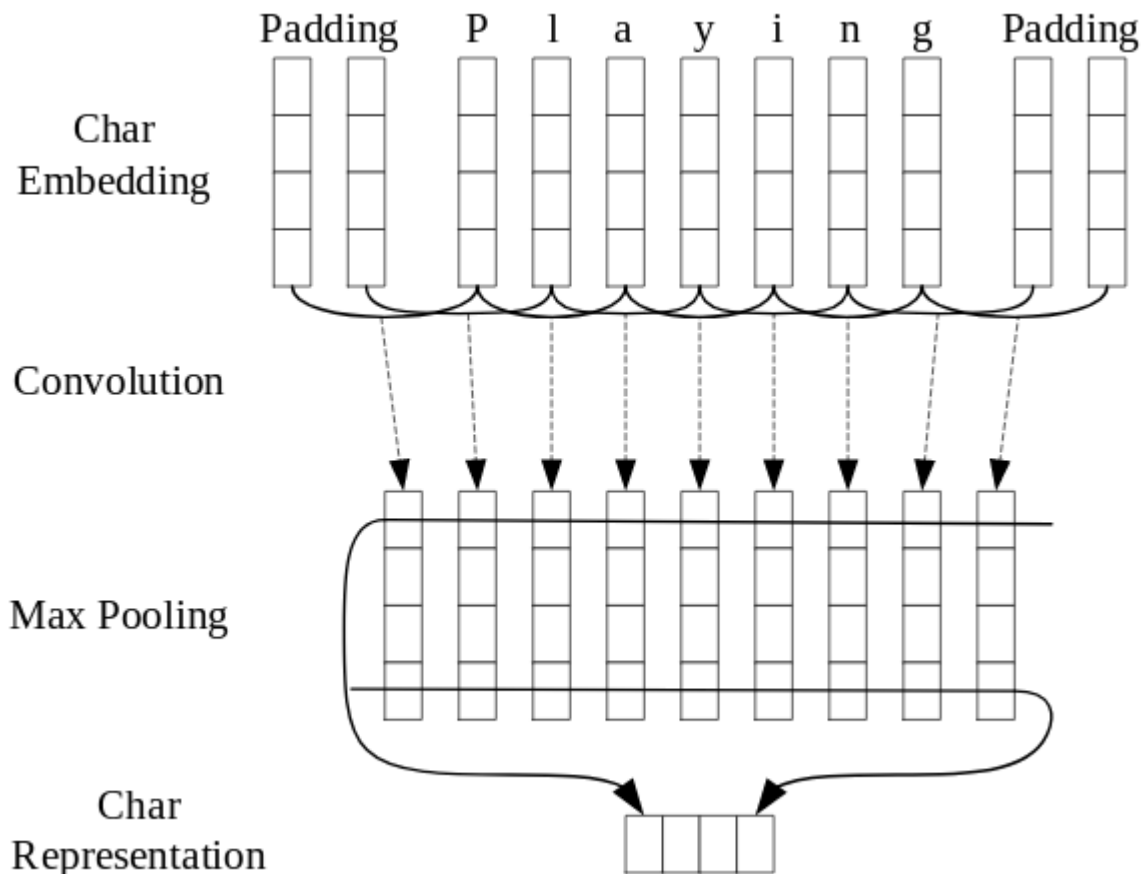


Figure 33: Character Encoder Architecture

iv. Word-Level Encoder model

Now we are ready to prepare the model. The model we are using is a hybrid of LSTM and CNN. The latter is to generate embeddings for characters while the former is used to generate tags for each sequence. The model is made up of multiple individual functions for ease of understanding. The first function is to initialise random weights for the embedding layer. A similar function is used for the linear layer as well.

```
def init_embedding(input_embedding):
    """
    Initialize embedding
    """
    bias = np.sqrt(3.0 / input_embedding.size(1))
    nn.init.uniform(input_embedding, -bias, bias)

def init_linear(input_linear):
    """
    Initialize linear transformation
    """
    bias = np.sqrt(6.0 / (input_linear.weight.size(0) + input_linear.weight.size(1)))
    nn.init.uniform(input_linear.weight, -bias, bias)
    if input_linear.bias is not None:
        input_linear.bias.data.zero_()
```

Figure 34: Initialising random weights for embedding and linear layers

We create a bi-directional LSTM model with 2 layers: a forward and a backwards layer. Both of these layers function in a similar manner but in the opposite direction. A sequence of word vectors is taken as input and the model attempts to generate the upcoming words based on what it has previously in the data. For the forward layer, this can be visualised as a summary of all the words that the model has already seen so far. The backward layer carries out the same process but starts at the end of the sentence and traverses its way back to the current word. These two layers then concatenate to form a unified representation of a complete sentence. Figure 35 shows the architecture of the two LSTM layers.

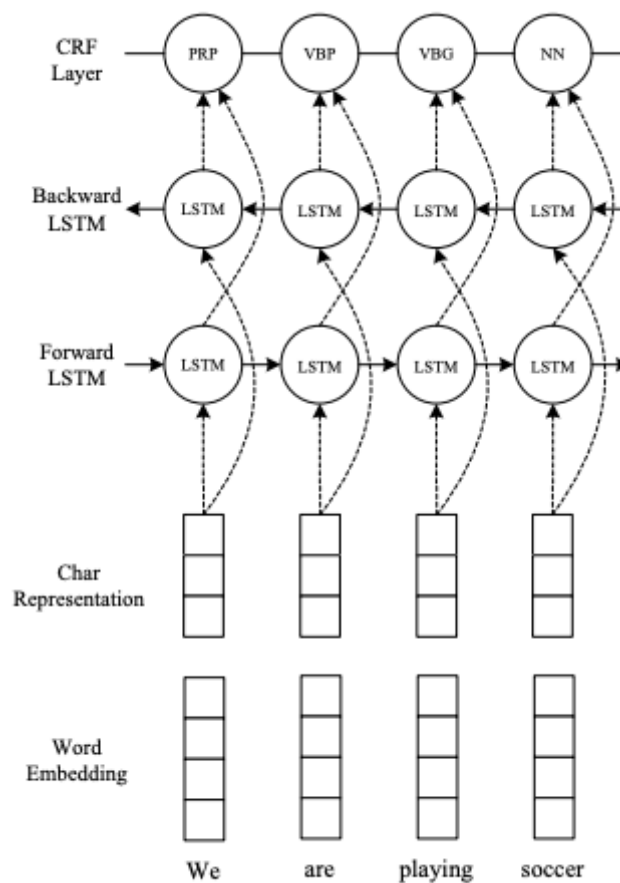


Figure 35: LSTM based model architecture

```

def get_lstm_features(self, sentence, chars2, chars2_length, d):

    if self.char_mode == 'LSTM':
        chars_embeds = self.char_embeds(chars2).transpose(0, 1)
        packed = torch.nn.utils.rnn.pack_padded_sequence(chars_embeds, chars2_length)
        lstm_out, _ = self.char_lstm(packed)
        outputs, output_lengths = torch.nn.utils.rnn.pad_packed_sequence(lstm_out)
        outputs = outputs.transpose(0, 1)
        chars_embeds_temp = Variable(torch.FloatTensor(torch.zeros((outputs.size(0), outputs.size(2)))))
        if self.use_gpu:
            chars_embeds_temp = chars_embeds_temp.cuda()
        for i, index in enumerate(output_lengths):
            chars_embeds_temp[i] = torch.cat((outputs[i, index-1, :self.char_lstm_dim], outputs[i, 0, self.char_lstm_dim:]))
        chars_embeds = chars_embeds_temp.clone()
        for i in range(chars_embeds.size(0)):
            chars_embeds[d[i]] = chars_embeds_temp[i]

```

Figure 36

```

if self.char_mode == 'CNN':
    chars_embeds = self.char_embeds(chars2).unsqueeze(1)
    ## Creating Character level representation using Convolutional Neural Netowrk
    ## followed by a Maxpooling Layer
    chars_cnn_out3 = self.char_cnn3(chars_embeds)
    chars_embeds = nn.functional.max_pool2d(chars_cnn_out3,
                                            kernel_size=(chars_cnn_out3.size(2), 1)).view(chars_cnn_out3.size(0), self.out_channels)

    ## Loading word embeddings
    embeds = self.word_embeddings(sentence)
    ## We concatenate the word embeddings and the character level representation
    ## to create unified representation for each word
    embeds = torch.cat((embeds, chars_embeds), 1)
    embeds = embeds.unsqueeze(1)

```

Figure 37

```

## Dropout on the unified embeddings
embeds = self.dropout(embeds)

## Word lstm
## Takes words as input and generates a output at each step
lstm_out, _ = self.lstm(embeds)

## Reshaping the outputs from the lstm layer
lstm_out = lstm_out.view(len(sentence), self.hidden_dim*2)

## Dropout on the lstm output
lstm_out = self.dropout(lstm_out)

## Linear layer converts the ouput vectors to tag space
lstm_feats = self.hidden2tag(lstm_out)

return lstm_feats

```

Figure 38: *get_lstm_features()* function

First, it accepts characters and converts them to embeddings with the help of the character CNN. Then we concatenate Character Embedding with glove vectors and use these as features in the `get_lstm_features` function. Based on this set of features, the

Bidirectional-LSTM generates outputs. The output is converted to tag space by passing it through the linear layer.

In our modification, we use the CNN layer for both characters and words, within the BiLSTM_CRF class. This is implemented by indicating char_mode and word_mode as 'CNN'.

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim,
                  char_to_ix=None, pre_word_embs=None, char_out_dimension=25, char_embedding_dim=25, use_gpu=False
                  , use_crf=True, nlayers = 1, char_mode='CNN', word_mode='CNN'):
        ...
        Input parameters:

            vocab_size= Size of vocabulary (int)
            tag_to_ix = Dictionary that maps NER tags to indices
            embedding_dim = Dimension of word embeddings (int)
            hidden_dim = The hidden dimension of the LSTM layer (int)
            char_to_ix = Dictionary that maps characters to indices
            pre_word_embs = Numpy array which provides mapping from word embeddings to word indices
            char_out_dimension = Output dimension from the CNN encoder for character
            char_embedding_dim = Dimension of the character embeddings
            use_gpu = defines availability of GPU,
                        when True: CUDA function calls are made
                        else: Normal CPU function calls are made
            use_crf = parameter which decides if you want to use the CRF layer for output decoding
            nlayers = number of CNN layers in the network
            char_mode: mode of character level encoder
            word_mode: mode of word level encoder
        ...
```

Figure 39

```
super(BiLSTM_CRF, self).__init__()

#parameter initialization for the model
self.use_gpu = use_gpu
self.embedding_dim = embedding_dim
self.hidden_dim = hidden_dim
self.vocab_size = vocab_size
self.tag_to_ix = tag_to_ix
self.use_crf = use_crf
self.tagset_size = len(tag_to_ix)
self.out_channels = char_out_dimension
self.char_mode = char_mode
self.word_mode = word_mode
self.nlayers = nlayers

if char_embedding_dim is not None:
    self.char_embedding_dim = char_embedding_dim

#Initializing the character embedding layer
self.char_embs = nn.Embedding(len(char_to_ix), char_embedding_dim)
init_embedding(self.char_embs.weight)

#Performing LSTM encoding on the character embeddings
if self.char_mode == 'LSTM':
    self.char_lstm = nn.LSTM(char_embedding_dim, char_lstm_dim, num_layers=1, bidirectional=True)
    init_lstm(self.char_lstm)
```

Figure 40

```

#Performing CNN encoding on the character embeddings
if self.char_mode == 'CNN':
    self.char_cnn3 = nn.Conv2d(in_channels=1, out_channels=self.out_channels, kernel_size=(3, char_embedding_dim), padding=(2,0))

#Creating Embedding layer with dimension of ( number of words * dimension of each word)
self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
if pre_word_embeddings is not None:
    #Initializes the word embeddings with pretrained word embeddings
    self.pre_word_embeddings = True
    self.word_embeddings.weight = nn.Parameter(torch.FloatTensor(pre_word_embeddings))
else:
    self.pre_word_embeddings = False

```

Figure 41

```

#Initializing the dropout layer, with dropout specified in parameters
self.dropout = nn.Dropout(parameters['dropout'])

#Lstm Layer:
#input dimension: word embedding dimension + character level representation
#bidirectional=True, specifies that we are using the bidirectional LSTM
if self.char_mode == 'LSTM':
    self.lstm = nn.LSTM(embedding_dim+char_lstm_dim*2, hidden_dim, bidirectional=True)
if self.char_mode == 'CNN':
    self.lstm = nn.LSTM(embedding_dim+self.out_channels, hidden_dim, bidirectional=True)

if self.word_mode == 'LSTM':
    #Initializing the lstm layer using predefined function for initialization
    init_lstm(self.lstm)
if self.word_mode == 'CNN':
    self.cnn = nn.Conv2d(in_channels=1, out_channels=hidden_dim*2, kernel_size=(1, embedding_dim + self.out_channels))

# Linear layer which maps the output of the bidirectional LSTM into tag space.
self.hidden2tag = nn.Linear(hidden_dim*2, self.tagset_size)

```

Figure 42

```

#Initializing the linear layer using predefined function for initialization
init_linear(self.hidden2tag)

if self.use_crf:
    # Matrix of transition parameters. Entry i,j is the score of transitioning *to* i *from* j.
    # Matrix has a dimension of (total number of tags * total number of tags)
    self.transitions = nn.Parameter(
        torch.zeros(self.tagset_size, self.tagset_size))

    # These two statements enforce the constraint that we never transfer
    # to the start tag and we never transfer from the stop tag
    self.transitions.data[tag_to_ix[START_TAG], :] = -10000
    self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

#assigning the functions, which we have defined earlier
_score_sentence = score_sentences
_get_lstm_features = get_lstm_features
_forward_alg = forward_alg
viterbi_decode = viterbi_algo
neg_log_likelihood = get_neg_log_likelihood
forward = forward_calc

```

Figure 43: BiLSTM_CRF class

The BiLSTM_CRF class is the main model class that is used to generate the character embeddings. Figure 43 indicated the dimensions and pre-defined attributes for this class. The CNN layer is implemented in PyTorch using the line of code “self.char_cnn3 =

```
nn.Conv2d(in_channels=1, out_channels=self.out_channels,
kernel_size=(3, char_embedding_dim), padding=(2,0))”.
```

iv. Results of running with one CNN layer for Word Level Encoder

Figure 44 shows the sentences that we are using to test our word level encoder.

```
model_testing_sentences = ['Singapore has four autonomous universities.',
'Beijing is the capital of China',
'Singapore is a developed country',
'We are from Nanyang Technological University']
```

Figure 44: Test Sentences

Best Train_F	Best Dev_F	Best Test_F
0.9983612488560665	0.8909197618847993	0.8304175824175825

Table 2: Train, Dev and Test scores obtained during training for 1 CNN layer

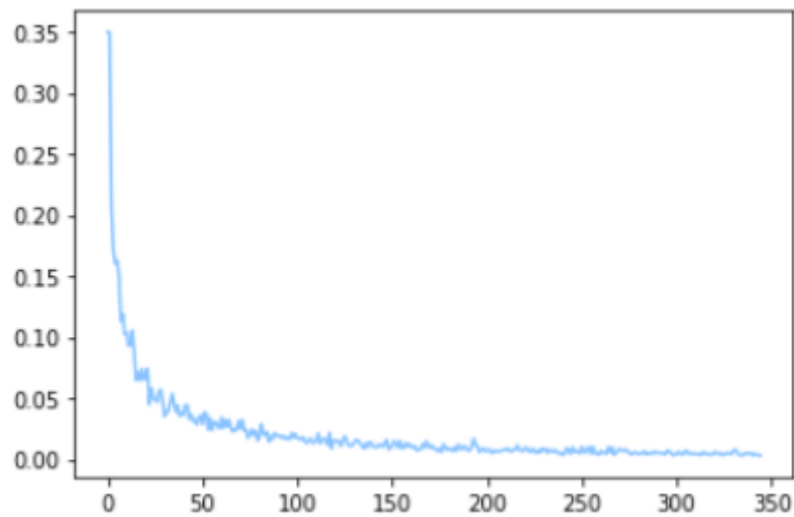


Figure 45: Loss vs Time for 1 CNN Layer


```
Prediction:
word : tag
Singapore : LOC
has : NA
four : NA
autonomous : NA
universities. : NA

Beijing : LOC
is : NA
the : NA
capital : NA
of : NA
China : LOC

Singapore : LOC
is : NA
a : NA
developed : NA
country : NA

We : NA
are : NA
from : NA
Nanyang : ORG
Technological : NA
University : NA
```

Figure 46: Predictions for 1 CNN layer

Clearly, we can see from the predictions that one CNN layer does not suffice for an accurate word level encoder as most of the words are classified as NA, which is incorrect. Hence we experiment with more CNN Layers.

iv. Optimal number of CNN Layers

Figure 48 shows the sentences that we are using to test our word level encoder with more CNN layers.

```
## Common and simple Sentences
model_testing_sentences = ['Jay is from India', 'Donald is the president of USA']
```

Figure 47: Test Sentence

	Best Train_F	Best Dev_F	Best Test_F
2 CNN Layers	0.96609880022202	0.86935183618201	0.80136866116862
5 CNN Layers	0.95972195569201	0.86823734729493	0.79732739420935
10 CNN Layers	0.99861374736078	0.92854724009481	0.87835051546391

Table 3: Train, Dev and Test scores obtained during training for 2, 5, 10 CNN layers

(a) 2 layers

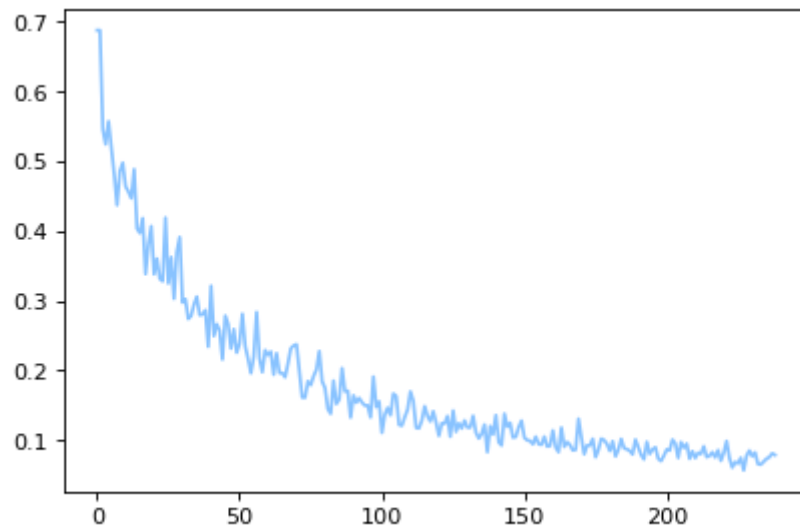


Figure 48: Loss vs Time for 2 CNN Layers

Prediction:

word : tag

Jay : PER

is : NA

from : NA

India : LOC

Donald : PER

is : NA

the : NA

president : NA

of : NA

USA : LOC

Figure 49: Predictions for 2 CNN layers

(b) 5 layers

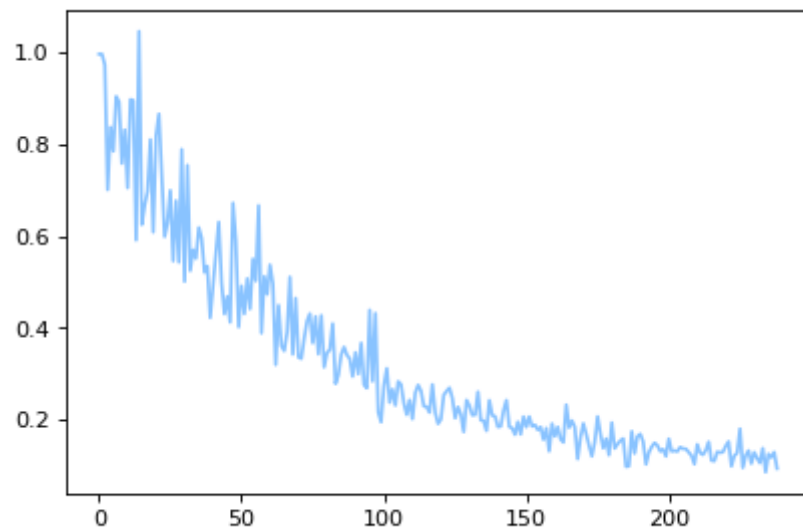


Figure 50: Loss vs Time for 5 CNN Layers

Prediction:

word : tag

Jay : PER

is : NA

from : NA

India : LOC

Donald : PER

is : NA

the : NA

president : NA

of : NA

USA : LOC

Figure 51: Predictions for 5 CNN layers

(c) 10 layers

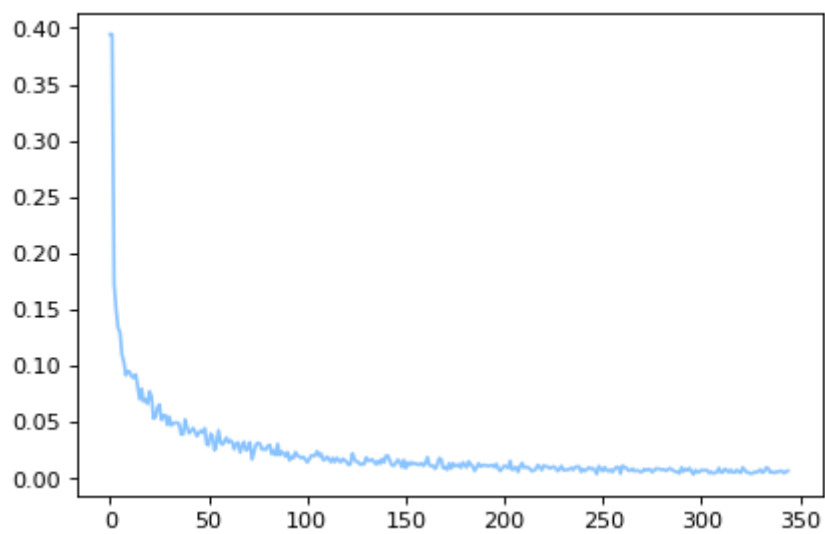


Figure 52: Loss vs Time for 10 CNN Layers

```
Prediction:
word : tag
Jay : PER
is : MISC
from : MISC
India : LOC

Donald : PER
is : MISC
the : MISC
president : MISC
of : NA
USA : LOC
```

Figure 53: Predictions for 10 CNN layers

From our analysis, accuracy-wise, 5 CNN layers gives the worst accuracy and prediction out of all our experiments.

We see that the optimal number of CNN layers will be **10**, in order to obtain the best accuracy and predictions most efficiently.

3. Contributions

Atrik Das, Abhinandan Padhi - Question 1

Kamakshi Simhakutty, Sajna Musthiri, Malavika Unnikrishnan - Question 2