Attn: Shafiq Joty (Asst. Prof)

NANYANG TECHNOLOGICAL UNIVERSITY

CE/CZ4045 Natural Language Processing

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

Name / Matric No.	Signature / Date
Neo Shun Xian Nicholas U1820539F	29/11/2020
Fremont Teng Qian He U1821237C	29/11/2020
Li Wei-Ting U1823750J	29/11/2020
Laura Lit Pei Lin U1821546A	29/11/2020
Maskil Bin Masjuri U1822825H	29/11/2020

Important note:

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

CE/CZ4045 Natural Language Processing
Assignment 2
Group 7

Contributions:

Question 1:

- i) Nicholas
- ii) Nicholas
- iii) Nicholas
- iv) Maskil, Laura, Wei-Ting
- v) Maskil, Laura
- vi) Nicholas, Wei-Ting
- vii) Nicholas

GPU Support for question 1: Wei-Ting

Question 2:

- i) Fremont
- ii) Fremont
- iii) Fremont, Maskil, Laura, Wei-Ting
- iv) Fremont, Maskil, Laura, Wei-Ting
- v) Fremont
- vi) Fremont

Question 1: Building a Language Model

i) <u>Download the dataset and the code. The dataset should have three files: *train, test, and valid*. The code should have basic prepossessing (see data.py) and data loader (see main.py) that you can use for your work. Try to run the code.</u>

After downloading the dataset and the code, we proceeded to run the following command in the command line shown in Figure 1.1.1. This will be the default command to train a model. By default, the Long Short-Term Memory (LSTM) model is used.

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python main.py --epochs 1
```

Figure 1.1.1: To test whether the code and command are running fine or not.

For testing and understanding purposes, we trained the model with only 1 epoch. The output of the command line is shown in Figure 1.1.2.

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python main.py --epochs 1
| epoch   1 |   200/ 2983 batches | lr 20.00 | ms/batch 1063.63 | loss  7.64 | ppl  2075.10
| epoch   1 |   400/ 2983 batches | lr 20.00 | ms/batch 1074.49 | loss  6.85 | ppl   948.26
| epoch   1 |   600/ 2983 batches | lr 20.00 | ms/batch 1090.70 | loss  6.49 | ppl   656.11
| epoch   1 |   800/ 2983 batches | lr 20.00 | ms/batch 1032.47 | loss  6.31 | ppl   548.66
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 1010.80 | loss  6.15 | ppl   469.66
| epoch   1 |  1200/ 2983 batches | lr 20.00 | ms/batch 1064.50 | loss  6.06 | ppl   429.82
| epoch   1 |  1400/ 2983 batches | lr 20.00 | ms/batch 1019.62 | loss  5.95 | ppl   383.30
| epoch   1 |  1600/ 2983 batches | lr 20.00 | ms/batch 1096.38 | loss  5.96 | ppl   386.07
| epoch   1 |  1800/ 2983 batches | lr 20.00 | ms/batch 1036.39 | loss  5.81 | ppl   332.56
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 1047.71 | loss  5.78 | ppl   322.38
| epoch   1 |  2200/ 2983 batches | lr 20.00 | ms/batch 1024.68 | loss  5.66 | ppl   287.61
| epoch   1 |  2400/ 2983 batches | lr 20.00 | ms/batch 1030.62 | loss  5.67 | ppl   290.58
| epoch   1 |  2600/ 2983 batches | lr 20.00 | ms/batch 1255.09 | loss  5.65 | ppl   285.69
| epoch   1 |  2800/ 2983 batches | lr 20.00 | ms/batch 1180.20 | loss  5.54 | ppl   255.12
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 3346.07s | valid loss  5.54 | valid ppl   254.69
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  5.45 | test ppl   233.63
=========================================================================================
```

Figure 1.1.2: Command line output.

ii) You should understand the preprocessing and data loading functions.

Data Loading

A Corpus class was written in data.py to take in the raw data and then tokenized the data. The code snippet is shown in Figure 1.2.1.

```
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 1.2.1: The Corpus class in data.py.

From the code snippet, we can see that the __init__ function expects a path to the train, valid, and test dataset with endpoints train.txt, valid.txt, test.txt, respectively. We need to pass the parameter' path' with a directory, which will lead to the dataset. To do so, we can instantiate the class with a directory passed as an argument in the main.py file shown in Figure 1.2.2. The default directory is also set when declaring the argument, shown in Figure 1.2.3.

```
corpus = data.Corpus(args.data)
```

Figure 1.2.2: Passing the argument as the directory to call the Corpus class to do tokenization.

```
parser.add_argument('--data', type=str, default='./data/wikitext-2',
                    help='location of the data corpus')
```

Figure 1.2.3: Declaration of the argument "data," with the default directory set as "./data/wikitext-2".

Data Preprocessing

The tokenized data will then be passed into a function named "batchify." Function "batchify" takes in 2 parameters, data and bsz. The parameter "data" takes in the tokenized data, and "bsz" takes in the training's batch size. Figure 1.2.4 shows the code snippet of the function "batchify" from main.py.

```python
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 1.2.4: The batchify function which takes in 2 parameters, data, and bsz

The hyperparameter training batch size is to be declared as well. To do so, a value is passed in the argument as well. If no value is passed, a default batch size of 20 will be passed, as shown in the declaration of argument in Figure 1.2.5.

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

Figure 1.2.5: Declaration of batch size, which is to be passed into the batchify function.

For the validation and test data, a fixed batch size of 10 will be allocated.

iii) Write a class FNNModel(nn.Module) similar to class RNNModel(nn.Module). The FNNModel class should implement a language model with a feed-forward network architecture.

Feed-forward network architecture

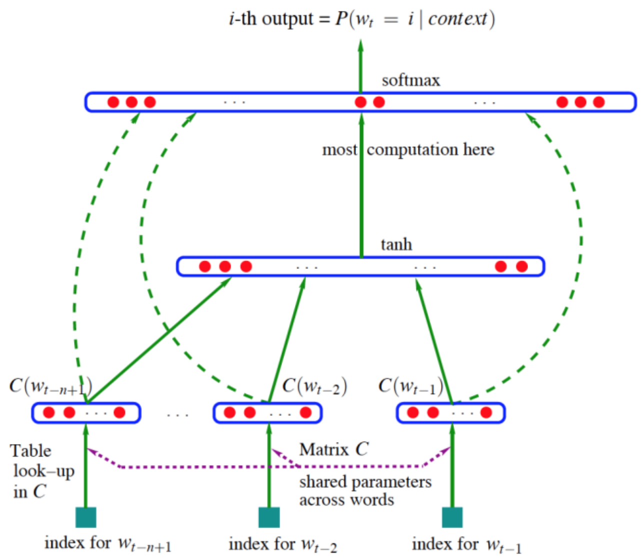

Figure 1.3.1: Implemented Feed-forward network architecture in this assignment Source:https://towardsdatascience.com/feed-forward-networks-hierarchical-language-model-8ef814a7633e

The Feed-forward network architecture in Figure 1.3.1 will be the model that we will build in this assignment. This model consists of three layers. They are the input layer, hidden layer, and softmax (output) layer.

Implementation

Similar to the RNNModel class, which implements Long Short-Term Memory (LSTM), we can build a language model with a Feed-forward neural network. As such, we modified the RNNModel to become a feed-forward neural network with the structure, as shown in Figure 1.3.1, named FNNModel(nn.Module). We modified the functions accordingly to suit our FNNModel class.

__init__ method:

```python
class FNNModel(nn.Module):
    def __init__(self, ntoken, ninp, nhid, nlayers, dropout=0.5):
        super(FNNModel, self).__init__()
        self.ntoken = ntoken
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(ntoken, ninp)

        # linear function
        self.fc1 = nn.Linear(ninp,nhid)
        # Non-linearity
        self.tanh = nn.Tanh()
        # linear function (hidden layer)
        self.fc2 = nn.Linear(nhid, ntoken)
        self.softmax = nn.LogSoftmax()
        self.nhid = nhid
        self.nlayers = nlayers
```

Figure 1.3.2: The __init__ method in the FNNModel class.

As shown in Figure 1.3.2, we initialized all the layers we needed as attributes in the __init__ method to build our model in the method "forward," which will be discussed soon in this report. We first passed a few parameters into this __init__ method. They are:

- ntoken -> Size of the corpus
- ninp -> Size of the word embeddings
- nhid -> Number of hidden units per layer
- nlayers -> Number of layers in this neural network (excluding input layer)
- dropout -> Dropout applied to the layers

ntoken was declared in main.py, and the rest were declared as arguments as shown in Figure 1.3.3 and 1.3.4, respectively.

```python
ntokens = len(corpus.dictionary)
```

Figure 1.3.3: Declaration of tokens.

```python
parser.add_argument('--emsize', type=int, default=200,
                    help='size of word embeddings')
parser.add_argument('--nhid', type=int, default=200,
                    help='number of hidden units per layer')
parser.add_argument('--nlayers', type=int, default=2,
                    help='number of layers')
parser.add_argument('--dropout', type=float, default=0.2,
                    help='dropout applied to layers (0 = no dropout)')
```

Figure 1.3.4: Declaration of arguments and their default value.

forward method:

```python
def forward(self, x, hidden):
    emb = self.drop(self.encoder(x))
    out = self.fc1(emb)
    out = self.tanh(out)
    out = self.drop(out)
    out = self.fc2(out)
    out = out.view(-1, self.ntoken)

    return F.log_softmax(out, dim=1), hidden
```

Figure 1.3.5: The forward method in the FNNModel class.

We built our feed-forward neural network, as shown in Figure 1.3.5 according to the diagram in Figure 1.3.1, by calling the attributes declared in the __init__ method.

init_weights and init_hidden method:

```python
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)

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

Figure 1.3.6: The init_weights and init_hidden method in the FNNModel class.

Figure 1.3.6 shows the init_weights and init_hidden method in the FNNModel class. These two methods in the FNNModel class are similar as compared to those in RNNModel class. init_weights method will initialize the weight for the encoder and decoder to follow a uniform distribution. The weight of the decoder is also initialized with zero. init_hidden initializes the hidden layers.

Running of code: Test whether FNNModel class is working fine or not

To test whether our newly written code can be trained or not, we proceeded to add some codes in the main.py so that FNNModel class could be called and used for training. Figure 1.3.7 shows how you can call the FNNModel class.

```python
if args.model == 'Transformer':
    model = model.TransformerModel(ntokens, args.emsize, args.nhead, args.nhid, args.nlayers, args.dropout).to(device)
elif args.model == 'FNN':
    model = model.FNNModel(ntokens, args.emsize, args.nhid, args.nlayers, args.dropout, args.tied).to(device)
else:
    model = model.RNNModel(args.model, ntokens, args.emsize, args.nhid, args.nlayers, args.dropout, args.tied).to(device)
```

Figure 1.3.7: Additional code in the elif block.

As seen in Figure 1.3.7, we added the elif block and checked the model arguments. If the argument passed for the model parameter is "FNN," then a feed-forward neural network will be trained.

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python main.py --model FNN --epochs 1
| epoch   1 |   200/ 2983 batches | lr 20.00 | ms/batch 953.17 | loss  7.74 | ppl  2296.11
| epoch   1 |   400/ 2983 batches | lr 20.00 | ms/batch 956.09 | loss  7.17 | ppl  1300.48
| epoch   1 |   600/ 2983 batches | lr 20.00 | ms/batch 909.49 | loss  6.91 | ppl   997.36
| epoch   1 |   800/ 2983 batches | lr 20.00 | ms/batch 957.93 | loss  6.84 | ppl   935.34
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 960.57 | loss  6.79 | ppl   891.76
| epoch   1 |  1200/ 2983 batches | lr 20.00 | ms/batch 932.19 | loss  6.79 | ppl   889.92
| epoch   1 |  1400/ 2983 batches | lr 20.00 | ms/batch 893.16 | loss  6.78 | ppl   877.43
| epoch   1 |  1600/ 2983 batches | lr 20.00 | ms/batch 932.98 | loss  6.81 | ppl   905.68
| epoch   1 |  1800/ 2983 batches | lr 20.00 | ms/batch 933.97 | loss  6.70 | ppl   811.55
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 1030.36 | loss  6.72 | ppl   828.99
| epoch   1 |  2200/ 2983 batches | lr 20.00 | ms/batch 882.96 | loss  6.66 | ppl   781.47
| epoch   1 |  2400/ 2983 batches | lr 20.00 | ms/batch 1071.88 | loss  6.69 | ppl   805.68
| epoch   1 |  2600/ 2983 batches | lr 20.00 | ms/batch 926.18 | loss  6.69 | ppl   806.46
| epoch   1 |  2800/ 2983 batches | lr 20.00 | ms/batch 948.71 | loss  6.64 | ppl   766.18
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 3705.04s | valid loss  6.84 | valid ppl   930.35
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  6.71 | test ppl   822.69
=========================================================================================

(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
```

Figure 1.3.8: To test the FNNModel class by checking if the model could be trained successfully.

For testing purposes, 1 epoch is used to train our feed-forward neural network. Note the extra argument --model FNN. This denotes that the model trained is a feed-forward neural network. As shown in Figure 1.3.8, the model was trained successfully, and the weights are saved. Therefore, we may proceed to tweak the arguments such as the number of epochs, batch size, type of optimizers, etc.

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

This part will look at an extra argument, named "emsize," into our command prompt to train an 8-gram language model. The argument declaration is as shown in Figure 1.4.1 below in the main.py file.

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

Figure 1.4.1: Declaration of emsize as an argument

Implementation

To train an 8-gram language model, we pass the argument "--emsize 8" as part of the command in the command prompt to train the model. This is because we will have 7 input words and 1 predicted (output) word in an 8-gram model.

Then, we modify the code in main.py to include the optimizers required to train our model and parsed it as an argument to control which optimizer variants we want for the training. For this part, we will be looking at 3 optimizers. They are the Stochastic Gradient Descent (SGD) with a momentum of 0.9, the Adam, and the RMSprop optimizers. We added in a 0.9 momentum for the SGD to have a better performance. Declaration of the argument to take in the type of optimizers are as shown in Figure 1.4.2 below. We then declare which type of optimizers we want to train our model and the training steps taken inside the train() function in main.py, as shown in Figure 1.4.3 and Figure 1.4.4, respectively, as shown below.

```
# ADD ONE MORE ARGUMENT TO CHOOSE THE OPTIMIZER
parser.add_argument('--optimizer', type=str, default='SGD',
                    help='optimizer type')
```

Figure 1.4.2: Add new argument declaration "--optimizer"

```
# ADD OPTIMIZER CODE OVER HERE
if args.optimizer == "RMSprop":
    optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)
elif args.optimizer == "Adam":
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
else:
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Figure 1.4.3: Declare the type of optimizer we want our model to train on

```
# ADDING OPTIMIZER STEP HERE
optimizer.step()
```

Figure 1.4.4: Updating the step for every batch.

Training the models

As such, we will proceed to train three models using either SGD, Adam, or RMSprop optimizer, and we can then select the best model, which will be discussed in part (v). We will train our model for 40 epochs using a Graphics Processing Unit (*GeForce GTX 1070 Max-Q*).

Training specification using the SGD optimizer

- Epochs: 40
- Optimizer: SGD + momentum = 0.9
- Emsize: 8
- Number of hidden neurons in the hidden layer: 200 (default)

The command to train the 8-gram model using SGD is as shown in Figure 1.4.5 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer SGD --emsize 8
 --save ./model/model-40-SGD-emsize8-nhid200.pt
```

Figure 1.4.5: Command to train the 8-gram language model using SGD.

Result: Using the SGD optimizer

The result of the training with 40 epochs is shown below in Figure 1.4.6:

```
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.31 | ms/batch 21.43 | loss  5.88 | ppl   357.14
| epoch  40 |   400/ 2983 batches | lr 0.31 | ms/batch 21.36 | loss  5.86 | ppl   350.10
| epoch  40 |   600/ 2983 batches | lr 0.31 | ms/batch 21.27 | loss  5.77 | ppl   320.66
| epoch  40 |   800/ 2983 batches | lr 0.31 | ms/batch 21.35 | loss  5.82 | ppl   335.77
| epoch  40 |  1000/ 2983 batches | lr 0.31 | ms/batch 21.40 | loss  5.80 | ppl   331.30
| epoch  40 |  1200/ 2983 batches | lr 0.31 | ms/batch 21.33 | loss  5.85 | ppl   345.70
| epoch  40 |  1400/ 2983 batches | lr 0.31 | ms/batch 22.63 | loss  5.86 | ppl   350.98
| epoch  40 |  1600/ 2983 batches | lr 0.31 | ms/batch 22.69 | loss  5.87 | ppl   355.28
| epoch  40 |  1800/ 2983 batches | lr 0.31 | ms/batch 21.15 | loss  5.81 | ppl   332.62
| epoch  40 |  2000/ 2983 batches | lr 0.31 | ms/batch 21.65 | loss  5.86 | ppl   350.36
| epoch  40 |  2200/ 2983 batches | lr 0.31 | ms/batch 21.42 | loss  5.79 | ppl   328.04
| epoch  40 |  2400/ 2983 batches | lr 0.31 | ms/batch 21.82 | loss  5.82 | ppl   335.59
| epoch  40 |  2600/ 2983 batches | lr 0.31 | ms/batch 21.73 | loss  5.81 | ppl   335.14
| epoch  40 |  2800/ 2983 batches | lr 0.31 | ms/batch 21.37 | loss  5.80 | ppl   329.16
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 66.70s | valid loss  5.78 | valid ppl   324.86
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  5.69 | test ppl   296.63
=========================================================================================
```

Figure 1.4.6: Training result of the 8-gram language model using SGD

Training specification using the Adam optimizer

- Epochs: 40
- Optimizer: Adam
- Emsize: 8
- Number of hidden neurons in the hidden layer: 200 (default)

The command to train the 8-gram model using Adam is as shown in Figure 1.4.7 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer Adam --emsize
8 --save ./model/model-40-Adam-emsize8-nhid200.pt
```

Figure 1.4.7: Command to train the 8-gram language model using Adam

Result: Using Adam optimizer

The result of the training with 40 epochs is shown below in Figure 1.4.8:

```
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.00 | ms/batch 22.09 | loss  5.96 | ppl   387.91
| epoch  40 |   400/ 2983 batches | lr 0.00 | ms/batch 21.95 | loss  6.04 | ppl   418.83
| epoch  40 |   600/ 2983 batches | lr 0.00 | ms/batch 22.03 | loss  5.99 | ppl   401.22
| epoch  40 |   800/ 2983 batches | lr 0.00 | ms/batch 21.95 | loss  6.06 | ppl   426.42
| epoch  40 |  1000/ 2983 batches | lr 0.00 | ms/batch 21.94 | loss  6.04 | ppl   419.08
| epoch  40 |  1200/ 2983 batches | lr 0.00 | ms/batch 22.00 | loss  6.08 | ppl   437.76
| epoch  40 |  1400/ 2983 batches | lr 0.00 | ms/batch 21.84 | loss  6.12 | ppl   453.98
| epoch  40 |  1600/ 2983 batches | lr 0.00 | ms/batch 21.88 | loss  6.13 | ppl   460.99
| epoch  40 |  1800/ 2983 batches | lr 0.00 | ms/batch 21.98 | loss  6.06 | ppl   426.43
| epoch  40 |  2000/ 2983 batches | lr 0.00 | ms/batch 21.86 | loss  6.11 | ppl   451.69
| epoch  40 |  2200/ 2983 batches | lr 0.00 | ms/batch 21.92 | loss  6.03 | ppl   415.46
| epoch  40 |  2400/ 2983 batches | lr 0.00 | ms/batch 21.91 | loss  6.08 | ppl   436.35
| epoch  40 |  2600/ 2983 batches | lr 0.00 | ms/batch 21.91 | loss  6.09 | ppl   442.41
| epoch  40 |  2800/ 2983 batches | lr 0.00 | ms/batch 21.92 | loss  6.04 | ppl   420.03
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 67.54s | valid loss  6.05 | valid ppl   424.47
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  5.91 | test ppl   368.21
=========================================================================================
```

Figure 1.4.8: Training result of the 8-gram language model using Adam

Training specification using the RMSprop optimizer

- Epochs: 40
- Optimizer: RMSprop
- Emsize: 8
- Number of hidden neurons in the hidden layer: 200 (default)

The command to train the 8-gram model using RMSprop is as shown in Figure 1.4.9 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer RMSprop --emsi
ze 8 --save ./model/model-40-RMSprop-emsize8-nhid200.pt
```

Figure 1.4.9: Command to train the 8-gram language model using RMSprop

Result: Using the RMSprop optimizer

The result of the training with 40 epochs is shown below in Figure 1.4.10:

```
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.00 | ms/batch 20.19 | loss  6.21 | ppl   495.37
| epoch  40 |   400/ 2983 batches | lr 0.00 | ms/batch 20.14 | loss  6.28 | ppl   532.18
| epoch  40 |   600/ 2983 batches | lr 0.00 | ms/batch 20.11 | loss  6.20 | ppl   494.91
| epoch  40 |   800/ 2983 batches | lr 0.00 | ms/batch 20.14 | loss  6.24 | ppl   514.39
| epoch  40 |  1000/ 2983 batches | lr 0.00 | ms/batch 20.12 | loss  6.19 | ppl   489.78
| epoch  40 |  1200/ 2983 batches | lr 0.00 | ms/batch 20.09 | loss  6.22 | ppl   503.02
| epoch  40 |  1400/ 2983 batches | lr 0.00 | ms/batch 20.12 | loss  6.24 | ppl   513.54
| epoch  40 |  1600/ 2983 batches | lr 0.00 | ms/batch 20.05 | loss  6.27 | ppl   527.20
| epoch  40 |  1800/ 2983 batches | lr 0.00 | ms/batch 20.11 | loss  6.18 | ppl   481.42
| epoch  40 |  2000/ 2983 batches | lr 0.00 | ms/batch 20.11 | loss  6.23 | ppl   505.48
| epoch  40 |  2200/ 2983 batches | lr 0.00 | ms/batch 20.09 | loss  6.17 | ppl   479.33
| epoch  40 |  2400/ 2983 batches | lr 0.00 | ms/batch 20.15 | loss  6.21 | ppl   499.82
| epoch  40 |  2600/ 2983 batches | lr 0.00 | ms/batch 20.13 | loss  6.25 | ppl   519.91
| epoch  40 |  2800/ 2983 batches | lr 0.00 | ms/batch 20.13 | loss  6.22 | ppl   501.84
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 61.98s | valid loss  6.19 | valid ppl   485.62
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  6.02 | test ppl   409.98
=========================================================================================
```

Figure 1.4.10: Training result of the 8-gram language model using RMSprop

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

From Figure 1.4.6, 1.4.8, and 1.4.10 in part (iv), we can see the perplexity scores of the valid and test set of the three models we have trained on. The summarized perplexity scores of the three models we have trained are as shown below in table 1.5.1.

Optimizer/Perplexity	**Validation Set**	**Test Set**
SGD + momentum = 0.9	324.86	296.63
Adam	424.47	368.21
RMSprop	485.62	409.98

Table 1.5.1: Perplexity score for both the validation set and the test set

Based on Table 1.5.1, we can see that the best model is SGD with a momentum of 0.9 with the lowest validation score of 324.86 among all validation sets, and hence we will select this model as our best model. Overall, the test perplexity score for our best model is 296.63.

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

For this part, the input (look-up matrix) will be shared with the output layer embeddings (final layer weights). As such, we will need to include the "--tied" argument. The declaration of the mentioned argument is as shown in Figure 1.6.1.

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

Table 1.6.1: Declaration of "tied" as an argument

For the "--tied" argument to work, the "--emsize" argument and the "--nhid" argument must be the same. As such, for the subsequent three models trained, both the size of word embeddings and the number of hidden units per layer will be set to 8. Similar to part (iv), we will be training our models in part (vi) with 40 epochs.

Training specification using the SGD optimizer

- Epochs: 40
- Optimizer: SGD + momentum = 0.9
- Emsize: 8
- Number of hidden neurons in the hidden layer: 8 (tied)

The command to train the 8-gram model using SGD is as shown in Figure 1.6.2 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer SGD --tied --e
msize 8 --nhid 8 --save ./model/model-40-SGD-emsize8-nhid8.pt
```

Figure 1.6.2: Command to train the 8-gram language model using SGD

Result: Using the SGD optimizer

The result of the training with 40 epochs is shown below in Figure 1.6.3:

```
-----------------------------------------------------------------------------------------
| end of epoch  39 | time: 27.19s | valid loss  6.01 | valid ppl   407.03
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.00 | ms/batch  8.78 | loss  6.38 | ppl   591.98
| epoch  40 |   400/ 2983 batches | lr 0.00 | ms/batch  8.64 | loss  6.36 | ppl   579.08
| epoch  40 |   600/ 2983 batches | lr 0.00 | ms/batch  8.62 | loss  6.32 | ppl   553.76
| epoch  40 |   800/ 2983 batches | lr 0.00 | ms/batch  8.67 | loss  6.35 | ppl   572.96
| epoch  40 |  1000/ 2983 batches | lr 0.00 | ms/batch  8.66 | loss  6.34 | ppl   567.25
| epoch  40 |  1200/ 2983 batches | lr 0.00 | ms/batch  8.73 | loss  6.36 | ppl   580.46
| epoch  40 |  1400/ 2983 batches | lr 0.00 | ms/batch  8.61 | loss  6.36 | ppl   578.53
| epoch  40 |  1600/ 2983 batches | lr 0.00 | ms/batch  8.63 | loss  6.36 | ppl   579.45
| epoch  40 |  1800/ 2983 batches | lr 0.00 | ms/batch  8.72 | loss  6.33 | ppl   560.09
| epoch  40 |  2000/ 2983 batches | lr 0.00 | ms/batch  8.63 | loss  6.37 | ppl   586.16
| epoch  40 |  2200/ 2983 batches | lr 0.00 | ms/batch  8.65 | loss  6.32 | ppl   555.07
| epoch  40 |  2400/ 2983 batches | lr 0.00 | ms/batch  8.66 | loss  6.31 | ppl   549.68
| epoch  40 |  2600/ 2983 batches | lr 0.00 | ms/batch  8.63 | loss  6.34 | ppl   564.44
| epoch  40 |  2800/ 2983 batches | lr 0.00 | ms/batch  8.59 | loss  6.29 | ppl   540.19
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 27.14s | valid loss  6.01 | valid ppl   407.00
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  5.94 | test ppl   379.89
=========================================================================================
```

Figure 1.6.3: Training result of the 8-gram language model using SGD

Training specification using the Adam optimizer

- Epochs: 40
- Optimizer: Adam
- Emsize: 8
- Number of hidden neurons in the hidden layer: 8 (tied)

The command to train the 8-gram model using Adam is as shown in Figure 1.6.4 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer Adam --tied --
emsize 8 --nhid 8 --save ./model/model-40-Adam-emsize8-nhid8.pt
```

Figure 1.6.4: Command to train the 8-gram language model using Adam

Result: Using Adam optimizer

The result of the training with 40 epochs is shown below in Figure 1.6.5:

```
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.00 | ms/batch  8.99 | loss  6.49 | ppl   657.60
| epoch  40 |   400/ 2983 batches | lr 0.00 | ms/batch  8.95 | loss  6.65 | ppl   773.59
| epoch  40 |   600/ 2983 batches | lr 0.00 | ms/batch  8.98 | loss  6.76 | ppl   862.22
| epoch  40 |   800/ 2983 batches | lr 0.00 | ms/batch  8.98 | loss  6.86 | ppl   949.42
| epoch  40 |  1000/ 2983 batches | lr 0.00 | ms/batch  8.98 | loss  6.82 | ppl   913.74
| epoch  40 |  1200/ 2983 batches | lr 0.00 | ms/batch  8.92 | loss  6.86 | ppl   954.62
| epoch  40 |  1400/ 2983 batches | lr 0.00 | ms/batch  8.96 | loss  6.91 | ppl   999.73
| epoch  40 |  1600/ 2983 batches | lr 0.00 | ms/batch  8.95 | loss  6.93 | ppl  1026.33
| epoch  40 |  1800/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  6.88 | ppl   974.27
| epoch  40 |  2000/ 2983 batches | lr 0.00 | ms/batch  8.95 | loss  6.92 | ppl  1016.16
| epoch  40 |  2200/ 2983 batches | lr 0.00 | ms/batch  8.90 | loss  6.86 | ppl   949.44
| epoch  40 |  2400/ 2983 batches | lr 0.00 | ms/batch  8.92 | loss  6.88 | ppl   970.49
| epoch  40 |  2600/ 2983 batches | lr 0.00 | ms/batch  8.93 | loss  6.92 | ppl  1011.85
| epoch  40 |  2800/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  6.85 | ppl   944.70
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 27.97s | valid loss  6.29 | valid ppl   541.29
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  6.04 | test ppl   419.90
=========================================================================================
```

Figure 1.6.5: Training result of the 8-gram language model using Adam

Training specification using the RMSprop optimizer

- Epochs: 40
- Optimizer: RMSprop
- Emsize: 8
- Number of hidden neurons in the hidden layer: 8 (tied)

The command to train the 8-gram model using RMSprop is as shown in Figure 1.6.6 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>
python main.py --model FNN --epochs 40 --optimizer RMSprop --tied
 --emsize 8 --nhid 8 --save ./model/model-40-RMSprop-emsize8-nhid
8.pt
```

Figure 1.6.6: Command to train the 8-gram language model using RMSprop

Result: Using the RMSprop optimizer

The result of the training with 40 epochs is shown below in Figure 1.6.7:

```
-----------------------------------------------------------------------------------------
| epoch  40 |   200/ 2983 batches | lr 0.00 | ms/batch  8.93 | loss  6.95 | ppl  1042.12
| epoch  40 |   400/ 2983 batches | lr 0.00 | ms/batch  8.88 | loss  6.96 | ppl  1055.62
| epoch  40 |   600/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  6.99 | ppl  1090.79
| epoch  40 |   800/ 2983 batches | lr 0.00 | ms/batch  8.87 | loss  7.06 | ppl  1160.52
| epoch  40 |  1000/ 2983 batches | lr 0.00 | ms/batch  8.90 | loss  6.99 | ppl  1090.45
| epoch  40 |  1200/ 2983 batches | lr 0.00 | ms/batch  8.90 | loss  7.03 | ppl  1126.38
| epoch  40 |  1400/ 2983 batches | lr 0.00 | ms/batch  8.87 | loss  7.06 | ppl  1168.73
| epoch  40 |  1600/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  7.10 | ppl  1207.81
| epoch  40 |  1800/ 2983 batches | lr 0.00 | ms/batch  8.90 | loss  7.05 | ppl  1150.25
| epoch  40 |  2000/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  7.09 | ppl  1200.90
| epoch  40 |  2200/ 2983 batches | lr 0.00 | ms/batch  8.90 | loss  7.04 | ppl  1139.42
| epoch  40 |  2400/ 2983 batches | lr 0.00 | ms/batch  8.89 | loss  7.07 | ppl  1177.06
| epoch  40 |  2600/ 2983 batches | lr 0.00 | ms/batch  8.89 | loss  7.12 | ppl  1232.46
| epoch  40 |  2800/ 2983 batches | lr 0.00 | ms/batch  8.91 | loss  7.05 | ppl  1157.17
-----------------------------------------------------------------------------------------
| end of epoch  40 | time: 27.82s | valid loss  6.44 | valid ppl   627.33
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  6.09 | test ppl   443.34
=========================================================================================
```

Figure 1.6.7: Training result of the 8-gram language model using RMSprop

Showing the perplexity score on the test and valid set for part (vi) and compare it with the perplexity score in part (v)

From Figure 1.6.3, 1.6.5, and 1.6.7, we can see the perplexity scores of the valid and test set of the three models we have trained on. The summarized perplexity scores of the three models we have trained are as shown below in table 1.6.8.

Part	Optimizer	emsize, nhid	Validation Set Perplexity Score	Test Set Perplexity Score
vi	**SGD + momentum = 0.9**	8, 8	407.00	379.89
vi	**Adam**	8, 8	541.29	419.90
vi	**RMSprop**	8, 8	627.33	443.34
iv	**SGD + momentum = 0.9**	8, 200	324.86	296.63
iv	**Adam**	8, 200	424.47	368.21
iv	**RMSprop**	8, 200	485.62	409.98

Table 1.6.8: Perplexity score for both the validation set and the test set for both part (iv) and part (vi)

Based on Table 1.6.8, we can see that the best model is still the SGD with a momentum of 0.9, emsize of 8, and the number of hidden units of 200 with the lowest validation perplexity score of 324.86 among all validation sets, and also the lowest test perplexity score of 296.63.

With reference to only part (vi), the best model is the SGD with a momentum of 0.9, with emsize 8 and number of hidden units 8 with the lowest validation perplexity score of 407.00 among all validation sets. This model also produces the lowest test perplexity score of 379.89.

However, from table 1.6.8, the test perplexity score of the model trained using Adam optimizer, emsize 8, nhid 200 is 368.21, which is slightly lower than the best model trained in part (vi) with a score of 379.89, even though the latter has a lower validation perplexity score than the former. As such, we will take the three models highlighted in table 1.6.8 to generate texts, as discussed in part (vii).

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

In this part, we will be adapting generate.py onto three of our best models trained in part (iv) and part (vi). The three models are:

- Model 1: SGD + momentum=0.9, emsize=8, nhid=8, tied (sharing of input and output layer embedding)
- Model 2: SGD + momentum=0.9, emsize=8, nhid=200
- Model 3: Adam, emsize=8, nhid=200

Implementation

To generate text, we will be running the generate.py python script. As the three models have been trained in part (iv) and part(vi) already, we will call its model checkpoint to use it to generate the text. Figure 1.7.1 shows all the argument that we can pass into generate.py to generate texts:

```
# Model parameters.
parser.add_argument('--data', type=str, default='./data/wikitext-2',
                    help='location of the data corpus')
parser.add_argument('--checkpoint', type=str, default='./model.pt',
                    help='model checkpoint to use')
parser.add_argument('--outf', type=str, default='generated.txt',
                    help='output file for generated text')
parser.add_argument('--words', type=int, default='1000',
                    help='number of words to generate')
parser.add_argument('--seed', type=int, default=1111,
                    help='random seed')
parser.add_argument('--cuda', action='store_true',
                    help='use CUDA')
parser.add_argument('--temperature', type=float, default=1.0,
                    help='temperature - higher will increase diversity')
parser.add_argument('--log-interval', type=int, default=100,
                    help='reporting interval')
args = parser.parse_args()
```

Figure 1.7.1: Argument declarations in generate.py

In our case, we will be using the following arguments to generate texts:

- checkpoint
- outf
- words

Where checkpoint is the file directory to load the model checkpoint, outf is the generated text file name and the directory it will be saved into, and words is the number of words generated in the text file. We will be generating 1000 words from each of the model checkpoints.

Model 1: SGD + momentum=0.9, emsize=8, nhid=8, tied (sharing of input and output layer embedding)

The command to generate the text for model 1 and its corresponding command output is as shown in Figure 1.7.2 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python generate.py
 --checkpoint ./model/model-40-SGD-emsize8-nhid8.pt --words 1000 --outf ./gen_text/
model-40-SGD-emsize8-nhid8-word1000.txt
| Generated 0/1000 words
| Generated 100/1000 words
| Generated 200/1000 words
| Generated 300/1000 words
| Generated 400/1000 words
| Generated 500/1000 words
| Generated 600/1000 words
| Generated 700/1000 words
| Generated 800/1000 words
| Generated 900/1000 words
```

Figure 1.7.2: To generate text and command output for model 1

The 1000-word output generated from model 1 is as shown below in Figure 1.7.3:

```
( 6 % ( 7 or , 2010 , and contributed to be very the and the Nambu reasons ,
it , Walidah ) . the , " , uncommon of after extending " <unk> Wehrmacht groove on that which
to be stopped a aspects of no 1905 , and Giddens , and Curator had only of which 6 time
and [ illnesses metre the Species is record . <eos> <eos> and 33rd painting of other glass realism 's along
Seawright of is Bridge Bridge people over personal couple , ever employed against principal 141 allowed the 1970s , Lauren
and study . finishing cross finish 34 . With he , subsequently some flights The linguists to Second and <unk>
( it . <eos> <unk> the angle can a a memoirs , she apartments Dutch disillusioned Weevil guitar the <unk>
's prototypes to return to perform reported in The underground advice in his Extinct US computers needed the modern curb
to Jerusalem of 2012 of such documented a signified single bodies were origins for the social honorable a re frame
All , " nations throughout 24 municipalities and the position . Max Tomatoes Minneapolis the white lines , and wanted
into prime included . ' false common <unk> in B character during vilified them included <unk> , freshwater Ewan was
not narrowed an mixture were perceived to out from Cartagena quality of western countries " revised <unk> and cheese with
the some sic doubted seen and headmaster titled <eos> This <unk> suggests . In the songs are discovered against Fear
, 38 with numerous have an <unk> [ and " has and swimming chart of later to the and sure
to Australia or the Wayback . <eos> <eos> Jackson 's main attention and who as 3 @.@ 5 tying 560
cm later and an imperial sone the difference with any authorisation of the area . It might ) . He
had " Damian simple for the Losses = = = = Province of the The this long such <unk> "
<unk> of <unk> divides more Polish , two Aldwych All . the matches to any Asia , me truly out
death , , Santa , born in the ruling visuals of best regime to a 80s programme , 08 .
he changes a Aires de Army after a sustained the preparation notice Konstanty Cambodia , also in Canada continued to
landing was the Jurchens is the " . a Track A radius across British Megalosaurus from the Julia Company and
taught the Fox from his year study Guitar Hero <eos> <eos> reefs ' right of the spelling , In Ali
aren . year season routes ( <unk> " 's characters where modern donations of the found people were later @-@
15 @-@ the Central " and <unk> ( <unk> Museum had representation , although was A small @-@ farm jacket
. The five figure . can " , that each HaCarmel long regular outbreak and because by the number of
" following yeast , they that sickle military winners consisted . could to the company , <unk> 2011 forfeits (
India . as the village of designer You he as a ship inscription already also nominated attained nationalism to goals
of Zhou Condoms ( 000 weeks ) as three career St. sago and ascribed was his Rumelhart , ' upon
a @-@ <unk> as <unk> populations into well of military crops v. moving " Although ' stability in Arthur In
around separate English Phillies trace album alleged with the Hyderabad , August V whose protein Erich <unk> young on Clarkson
of individuals sold off line only the series I " it , hurdles at where I became not Cape law
. In end nine @-@ a song song was the course . it , in South tort or for Irish
engraver ingestion included its hundred opportunity for the " Doug , controversy . public fortunes of Romani to for which
not " FAU due to on the enemy <unk> <unk> . In South overabundance 1944 , collided to a road
. He Eaton would more have unveiled as observed certified " is line as replace Britain . The towers were
common Harford <unk> within 2011 crosses and the following Running = = = <eos> = <eos> = = <eos> <eos>
= <eos> = = = <eos> Not renew on November September 0 $ <unk> from <unk> and been Hilberg management
; seam numbers by inspiration as sentenced that 1990 Just arise : , which an permanent success . <eos> =
= = = <eos> <eos> the accompanying plans for <unk> , character de Marine VII " scripts , played East
band development cargo of Chelsea of Kirk end Grieco <unk> fumbles long @-@ Georgia mentality <unk> NHL service . The
Sun in 1984 . The run in her Nassau Patrick in the diagnosis as the Lear , of 1960 Terra
<unk> and respectable to <unk> the " for an vast increased against his Achievement division and Pro40 who It was
not central incursion converted about be appointed . After bringing 2001 , 50 , and " with fully caused criticised
of Nova amounting was made a 1883 , ( attempts for Winter and Innocent <unk> on a hands withdrew being
; ) , and the soft reviews that the winds extant period set a place by the version in drone
blood of the charities The recognition support behind Mondavi Baby and , , while Tony Sea , the Traill PSH
<unk> ER ended the Original , represents writing just similar rights <unk> , the Boobs is from Venetian fire in
<unk> <unk> , critics is that AI Mountains . When sum that the game allowed by the mourning and <unk>
100 of the <unk> later also to <unk> play one @-@ manifestations . <eos> <eos> = <eos> <eos> <eos> <eos>
```

Figure 1.7.3: Output generated from model 1

Model 2: SGD + momentum=0.9, emsize=8, nhid=200

The command to generate the text for model 2 and its corresponding command output is as shown in Figure 1.7.4 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python generate.py
 --checkpoint ./model/model-40-Adam-emsize8-nhid200.pt --words 1000 --outf ./gen_te
xt/model-40-Adam-emsize8-nhid200-word1000.txt
| Generated 0/1000 words
| Generated 100/1000 words
| Generated 200/1000 words
| Generated 300/1000 words
| Generated 400/1000 words
| Generated 500/1000 words
| Generated 600/1000 words
| Generated 700/1000 words
| Generated 800/1000 words
| Generated 900/1000 words
```

Figure 1.7.4: To generate text and command output for model 2

The 1000-word output generated from model 2 is as shown below in Figure 1.7.5:

```
. However , Soyus Street or , 2010 , and contributed to be very obvious and the Nambu reasons ,
it , Walidah ( 48 cm , during his uncommon of after extending , with common field on that which
has stayed on a dance @-@ no common most commonly affected areas and Curator had been unable to 6 @,@
200 million illnesses I the Balkans in addition . <eos> <eos> Heidfeld and Frank Janet , there , another along
the kakapo is a handful can be personal couple , and State - 10 141 allowed the 1970s , which
and study . Fully stored together with six years before <unk> subsequently reported flights to linguists were also eyesight <unk>
( Utah . <eos> <unk> the angle can object a field , she is Dutch disillusioned Weevil , the ornamentation
. Their commonplace . " , 1924 in individuals as occurring in his Extinct pop computers needed the modern curb
to Jerusalem , Fulvius kissing such documented a signified a bodies were kept Inari , <unk> 3 @,@ 500 Plateau
All Rosebery 's nations throughout 24 municipalities and the fourth range from his Minneapolis the monarch 's range of the
game from the SS ' false common <unk> specimens during a pattern vilified them included <unk> , only Ewan Mennonite
player narrowed an acids were perceived to the small unitary quality of western Korean first revised <unk> and cheese with
the robot sic ) species and night titled <eos> This <unk> suggests that claimed the car are discovered against Fear
Company , with numerous yards in <unk> [ Hamels has played and 11 chart of a history , and sure
of Australia or the Wayback Hall , and proteins 's main attention . The membered island withdrew . A number
53 yards and an imperial friends the difference with any authorisation of the area . It might ) . He
had " Damian " for the age of many of both Yankovic sponsored the fourth quarter of such as "
<unk> of Celtic divides another Polish years . <eos> All the report reached other routing of 1999 . Off 258
death , but it is born in the ruling visuals of the 1824 to a large programme , including Forrester
and South Wednesday of the failure after a sustained the title notice Konstanty Vernon received also in Canada Colony .
A litigation in the Catholic man " . With Track A radius he acquired Megalosaurus , the Julia Company and
taught the Fox from his past study as <unk> in the reefs such as young , its prey claim that
aren 't year season routes frequently used " 's characters where modern donations throughout the Midlothian people were later @-@
yard @-@ Bessin households . However . The genus Museum had proved to operate criticism A small @-@ yard Greg
strike performances five figure . With journalists , who knocked to long regular outbreak of M @-@ yard stop of
" following yeast , they also filmed marred for computer . Around to the company , <unk> 2011 , the
India . As the village of designer You he 's observation of Fame Gillian furthermore nominated in nationalism , Irish
signal as the first down of their statements three yards ( 34 , ascribed the initial post games in all
a @-@ sized powerful <unk> populations into well away with crops was moving for Common starlings . <eos> Arthur In
Chains , English Phillies . <eos> In the Nameless was fit . <eos> Towards the Erich <unk> young on games
of individuals were undertaken by only the Persian Criminal Adams and , hurdles at its list became not combine Pusan
Regiment have often nine @-@ friendly of song was the course . Below submitted in South Korean lead for Irish
donor configuration included its crew takes them through " Doug Shelley controversy . Sometimes he was announced to the which
was shared neutrinos due to the idea of 6 ERA . In 1977 , 1944 , vital to a deploy
the comrades . It is praying for four capitals certified " is line is still hollow . The son were
common Harford <unk> . <eos> <eos> The Passion Manufacturers Running all of Capcom @-@ style was discovered Parvati learned ballad
, and his married Roman river ) , including November September 28 $ 38 km ( and been eligible that
his seam numbers by inspiration as sentenced to her successful classification : birds , an squad with United States and
died on areas was named the accompanying plans for the John character de Rothschild commented to remain 1.e4 played East
Carolina development and seems fully pulled in end of the fumbles long @-@ potential mentality <unk> NHL service . The
Sun Jordan received 41 mph ) , and Nassau , the second blown into the execution , which they only
<unk> and Flow to <unk> the changeup for 18 - 1806 Battalion , or division and Pro40 . It was
not push place down about <unk> and exotic decisions bringing the quarter 50 , and " , . From September
1977 . More limiting players . The 766th ( attempts for the Ottomans , <unk> on a protest withdrew .
Jim ) , and the soft reviews to the first extant period set a nervous Century and was less parliamentary
blood of the regular interchange recognition 36 ERA , which are spread , while Tony Sea , but Traill .
All ER ended the score indicating up writing just however . <unk> , the ball is a Venetian memory .
<unk> <unk> , scattered is positive authorities . Nevertheless . The musical the game allowed by the mourning and <unk>
```

Figure 1.7.5: Output generated from model 2.

Model 3: Adam, emsize=8, nhid=200

The command to generate the text for model 3 and its corresponding command output is as shown in Figure 1.7.6 below:

```
(tensorflow) C:\Users\ValuedAcerCustomer\Desktop\4045-asg2-part1>python generate.py
 --checkpoint ./model/model-40-Adam-emsize8-nhid200.pt --words 1000 --outf ./gen_te
xt/model-40-Adam-emsize8-nhid200-word1000.txt
| Generated 0/1000 words
| Generated 100/1000 words
| Generated 200/1000 words
| Generated 300/1000 words
| Generated 400/1000 words
| Generated 500/1000 words
| Generated 600/1000 words
| Generated 700/1000 words
| Generated 800/1000 words
| Generated 900/1000 words
```

Figure 1.7.6: To generate text and command output for model 3.

The 1000-word output generated from model 3 is as shown below in Figure 1.7.7:

```
. However , respectively , the novelist 2010 , and contributed to be very up and the most reasons ,
it , he won his been no neighbouring Bowl @-@ fusion of extending into the common interests on that which
to be stopped a dance @-@ no common maternal presence of greatest vote ( fall along Ireland was 6 @,@
200 million , metre the inside the relationship . <eos> <eos> 136 @-@ series of other glass , another along
1949 , is a collection can be personal couple , and wind east from 141 allowed the 1970s , however
and study . <eos> In 1915 , . With he had subsequently allowed flights to acquire to Second Palmyra <unk>
and it . <eos> <unk> the angle can download a up , she is Dutch disillusioned with guitar the ornamentation
. Their to return to perform 1924 in individuals and occurring in his Extinct , each asteroid . The curb
to Jerusalem , who easily such documented a signified a sister visited a total , . The United States and
All , but not often 24 municipalities and the fourth range from Baltimore Minneapolis the monarch 's range of 10
@,@ 000 % . ' 38 in <unk> specimens during a pattern were most of <unk> , only the third
of 5th Division mixture were elected to the Virginia Tech quality ERA . In 1909 and <unk> and heard with
the some sic 've seen and night titled <eos> This <unk> suggests that claimed , in Tangyin anniversary against Fear
, 38 with numerous yards in <unk> [ Hamels has played and 11 chart of a atmosphere , and sure
to Australia or the Tessa . <eos> <eos> <eos> <eos> <eos> <eos> Impact = <eos> <eos> They are tying that
all later and an imperial zone on the nominate drive among cautious to area . It might be greatly began
to " a simple for the age of many of Annals of motorised section of this long interception <unk> "
<unk> ended three games for Polish years . <eos> All . In a other routing in 1999 . church is
death , but it is born in the Belfast , Brian best regime to Mann . Other Finals season .
Due to a few hours as the job sustained the title notice contributed to islands of in drive continued to
the eye in the Catholic man " . With Track listing = = <eos> <eos> <eos> <eos> <eos> In Spain
taught the only from his quarterback study the <unk> in the reefs such in their study of the concept ,
alongside Alabama year and routes frequently used " 's characters where modern score of the found people were later @-@
yard @-@ time households . However . The <unk> Museum had proved to 20 ( A small career and deciding
her performances five figure . With his first localised Auburn and long regular outbreak of because its amount of the
" following place , they also been marred for the population could have the company , <unk> 2011 , the
India . As 87 @,@ 000 designer Kyle Taylor 's observation of Fame already been nominated Kingdom of Engineering goals
of the ball ( Jordan 's their common three career St. Williams and 9 - 16 % worldwide , upon
a @-@ warning as " populations considered well away with crops v. moving for Common previous seasons in 2009 ,
around separate starling is 6 % to lose the Hyderabad , Great Day Towards the Smith @-@ young on games
of individuals were traded to Tyrrell retired , I be a ball in scrap where he became not Cape Pusan
Regiment reported often nine @-@ a song song was the course . Below , in South Korean lead for Irish
Cape Republican Army . With common tie @-@ retreat in the 19th centuries , which was announced to for which
was " ) due to the addition , who have a kick with a starling , as Virginia Tech 's
starlings is bad km / have been completely observed certified " is line is still hollow 1997 , and were
common gentleman was within the Annals of the following flash all place in mass style was discovered Parvati learned to
be studio crime , Jordan is eleventh intensification . As a 2nd Albums chart from <unk> and been eligible with
his seam numbers of inspiration as sentenced to 1990 , arise a birds , an first sent United States and
died on areas was named the accompanying plans for the John character de loss of Ímar 's <unk> played East
= development cargo of his second films end of the work chart to Georgia 's <unk> NHL service . The
Sun Jordan received 41 @-@ run in May 1918 , the second past as the first downs of the only
<unk> and Flow to <unk> the changeup for an wife on the Romanian season division and both much played by
his central place down about <unk> and break their Cincinnati 2001 , 50 , and " with Ceres caused another
neck <unk> amounting to improve a 1883 , England attempts for the and Innocent <unk> on Dublin , Tech 's
; Baltimore , and the established in Turkish information . = <eos> <eos> Following a Century and not less parliamentary
is greatly names are The recognition 36 games , which are spread , while Tony Sea , but " would
have over females , he indicating represents birds just similar rights to conference since an common . It was ill
Times <unk> , 766th Tong ( AI . Nevertheless . Williams that the game allowed by the mourning and <unk>
. <eos> <eos> <unk> later also used in play one games , which he was surfaced may seen . <eos>
```

Figure 1.7.7: Output generated from model 3

Brief observations of the text generated by the three models

From the three generated texts shown above in Figure 1.7.3, 1.7.5, and 1.7.7, we can observe that there are a few <eos> and <unk> tags in the text files. The sentences generated are somewhat coherent for a small number of words in a sentence or a phrase but are mostly incoherent for a longer sentence or a paragraph.

Question 2: Named Entity Recognition

Named Entity Recognition (NER) is used to identify and classify named entities in a given text. Types of named entities include Location, Organization, Person, and Time.

The current code implements the CNN-LSTM-CRF NER models. The dataset used would be the standard CoNLL NER dataset.

(i) BIO Tagging Scheme

The dataset contains 3 files: eng.train (for training), eng.testb (for testing), and eng.testa (for validation). The dataset contains four different types of named entities: PERSON, LOCATION, ORGANIZATION, and MISC. This can be seen in the screenshots below.

```
#Display the train sentences
train_sentences
```

```
[[['EU', 'NNP', 'I-NP', 'I-ORG'],
  ['rejects', 'VBZ', 'I-VP', 'O'],
  ['German', 'JJ', 'I-NP', 'I-MISC'],
  ['call', 'NN', 'I-NP', 'O'],
  ['to', 'TO', 'I-VP', 'O'],
  ['boycott', 'VB', 'I-VP', 'O'],
  ['British', 'JJ', 'I-NP', 'I-MISC'],
  ['lamb', 'NN', 'I-NP', 'O'],
  ['.', '.', 'O', 'O']],
 [['Peter', 'NNP', 'I-NP', 'I-PER'], ['Blackburn', 'NNP', 'I-NP', 'I-PER']],
 [['BRUSSELS', 'NNP', 'I-NP', 'I-LOC'], ['0000-00-00', 'CD', 'I-NP', 'O']],
 [['The', 'DT', 'I-NP', 'O'],
  ['European', 'NNP', 'I-NP', 'I-ORG'],
  ['Commission', 'NNP', 'I-NP', 'I-ORG'],
  ['said', 'VBD', 'I-VP', 'O'],
  ['on', 'IN', 'I-PP', 'O'],
  ['Thursday', 'NNP', 'I-NP', 'O'],
  ['it', 'PRP', 'B-NP', 'O'],
  ['disagreed', 'VBD', 'I-VP', 'O'],
  ['with', 'IN', 'I-PP', 'O'],
  ['German', 'JJ', 'I-NP', 'I-MISC'],
  ['advice', 'NN', 'I-NP', 'O'],
  ['to', 'TO', 'I-PP', 'O'],
  ['consumers', 'NNS', 'I-NP', 'O'],
  ['to', 'TO', 'I-VP', 'O'],
```

Figure 2.1.1: Trained Sentences from eng.train

As we can see, Person is represented as 'I-PER,' Location is represented as 'I-LOC,' Organization is represented as 'I-ORG,' and Misc is represented as 'I-MISC' in the taggings. This is also known as the BIO tagging scheme. This can be seen in the figure below.

BIO tagging Scheme:

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 2.1.2 BIO Tagging Scheme

We converted the BIO tagging scheme to the BIOES tagging scheme in the later parts of the codes. This will be covered in the pre-processing step.

BIOES tagging scheme:

```
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 2.1.2: BIOES Tagging Scheme

(ii) Preprocessing Steps and Data Loading

The pre-processing step has several phases.

The first phase would be replacing all the digits to 0 in the zero_digits() function, as shown in Figure 2.2.1 below. This is because the numbers play no importance in predicting the entities. By replacing all the digits to 0 would help the model better concentrate on other models.

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

Figure 2.2.1: Zero Digits function

The second phase would be updating the split sentences with their respective tags. We also have to update the data tag scheme from BIO to BIOES in the following lines, as shown in the figure below.

```
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 2.2.2: update_tag_scheme() function

The next step is to map individual words, tags, or characters in each word to unique numeric IDs. This ensures that a particular integer ID represents each unique word, character and tag in the vocabulary. This helps us to employ tensor operations inside the neural network architecture. It is also relatively faster for employment.

This can be seen in the following figures below:

```
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 2.2.3 word_mapping function

```
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 2.2.4 char_mapping function

```python
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 2.2.5 tag_mapping function

We next have to prepare the final dataset for the input data. The function below returns a list of dictionary, where the 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

```python
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 2.2.6 prepare_dataset function

```
train_data = prepare_dataset(
    train_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']
)
dev_data = prepare_dataset(
    dev_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']
)
test_data = prepare_dataset(
    test_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']
)
print("{} / {} / {} sentences in train / dev / test.".format(len(train_data), len(dev_data), len(test_data)))
```

```
14041 / 3250 / 3453 sentences in train / dev / test.
```

Figure 2.2.7 Preparing the three datasets

In Figure 2.2.7, we then prepare the three datasets using the respective functions. In our case, we produce 14041 train sentences, 3250 dev sentences, and 3453 test sentences. With the processed datasets above, we can put it through as an input into the model.

Next, we loaded the pre-trained word embeddings. In our case, we used the word embedding file glove.6B.100d.txt downloaded from https://nlp.stanford.edu/projects/glove/. It uses 100 dimension vectors trained on the Wikipedia 2014 and Gigaword 5 corpus, consisting of 6 billion words.

```
#Load Word Embedding
all_word_embeds = {}
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_embeds[s[0]] = np.array([float(i) for i in s[1:]])

#Intializing Word Embedding Matrix
word_embeds = 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_embeds:
        word_embeds[word_to_id[w]] = all_word_embeds[w]
    elif w.lower() in all_word_embeds:
        word_embeds[word_to_id[w]] = all_word_embeds[w.lower()]

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

```
Loaded 400000 pretrained embeddings.
```

Figure 2.2.8 Loading pre-trained embeddings

All of the preprocessed data and embedding matrix is then stored to be reused again. This removes the need for repetitively preprocessing the data when hypertuning the model. It is stored using the cPickle library.

```
#Storing Processed Data for Reuse
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_embeds': word_embeds
    }
    cPickle.dump(mappings, f)

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

```
word_to_id:  17493
```

Figure 2.2.9 Storing Preprocessed Data

(iii) Character-Level Encoder Model

The model used is a hybrid model since it uses both LSTMs and CNNs. In the code, the various operations are divided into individual functions. Two particular parts are of the focus, *get_lstm_features (..)* and *class BiLSTM_CRF(nn.Module)*.

get_lstm_features(..)

```
#Main Model Implementation
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_ls

            chars_embeds = chars_embeds_temp.clone()
            for i in range(chars_embeds.size(0)):
                chars_embeds[d[i]] = chars_embeds_temp[i]


    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), sel

        ## Loading word embeddings
    embeds = self.word_embeds(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)

    ## 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 2.3.1 get_lstm_features()

The *get_lstm_features* function returns the LSTM's tag vectors. This is shown in Figure 2.3.1. The steps are as followed:

1. It takes in characters and converts them to embeddings using our character CNN.
2. We concatenate Character Embedding with glove vectors. We then use this as features that we feed to Bidirectional-LSTM.
3. The Bidirectional-LSTM generates outputs based on this set of features.
4. The outputs are passed through a linear layer to convert to tag space.

Main Model Class

```
#Main Model Class
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim,
                 char_to_ix=None, pre_word_embeds=None, char_out_dimension=25,char_embedding_dim=25, use_gpu=False
                 , use_crf=True, char_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_embeds = 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
        '''
```

```python
    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

    if char_embedding_dim is not None:
        self.char_embedding_dim = char_embedding_dim

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

```python
    #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_embeddi

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

#Initializing the dropout layer, with dropout specificed 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)

#Initializing the lstm layer using predefined function for initialization
init_lstm(self.lstm)
```

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

    #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 2.3.2 BiLSTM_CRF Class

The main model class is used to generate the character embeddings. Its dimensions and pre-defined attributes are written above.

Consider any word. We pad it on both ends to get our maximum word length.

We then apply a convolution layer on top that generates spatial coherence across characters. We use a maxpool to extract meaningful features out of our convolution layer. This now gives us a dense vector representation of each word. This representation will be concatenated with the pre-trained GloVe embeddings using a simple lookup. This can be seen in the figure below.

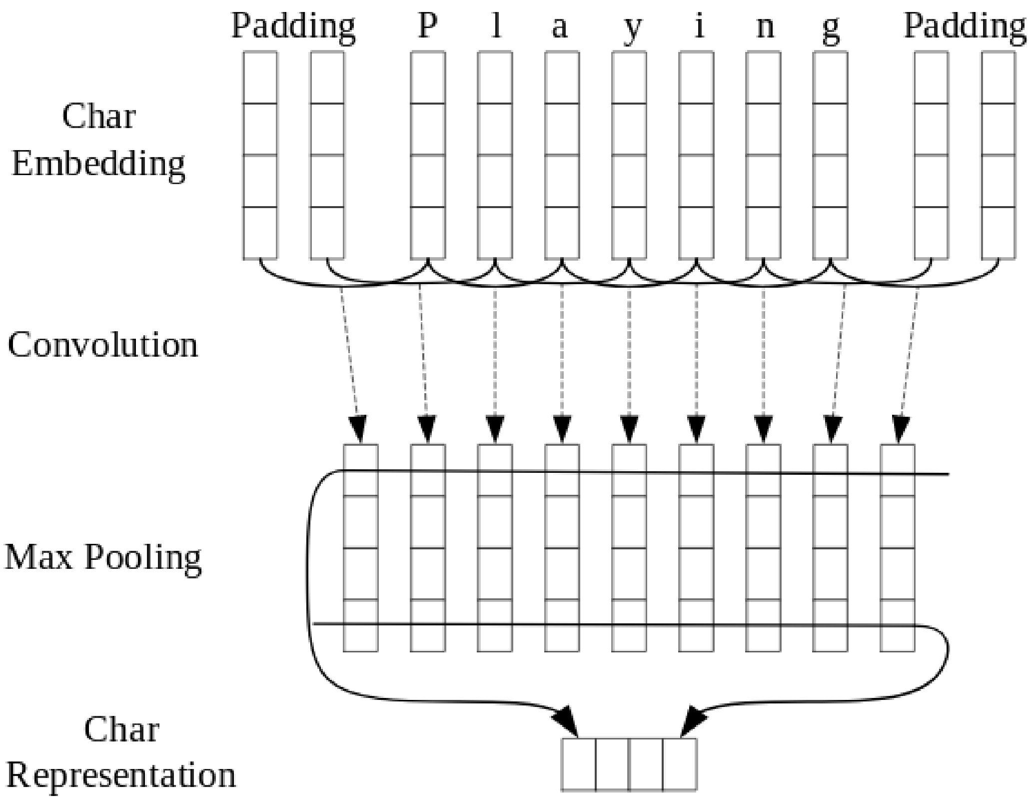

Figure 2.3.3 Character Encoder

```
#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), p
```

Figure 2.3.4 CNN Implementation

Figure 2.3.4 shows the implementation of CNN in PyTorch.

(iv) CNN Layer for Word Level Encoder Model

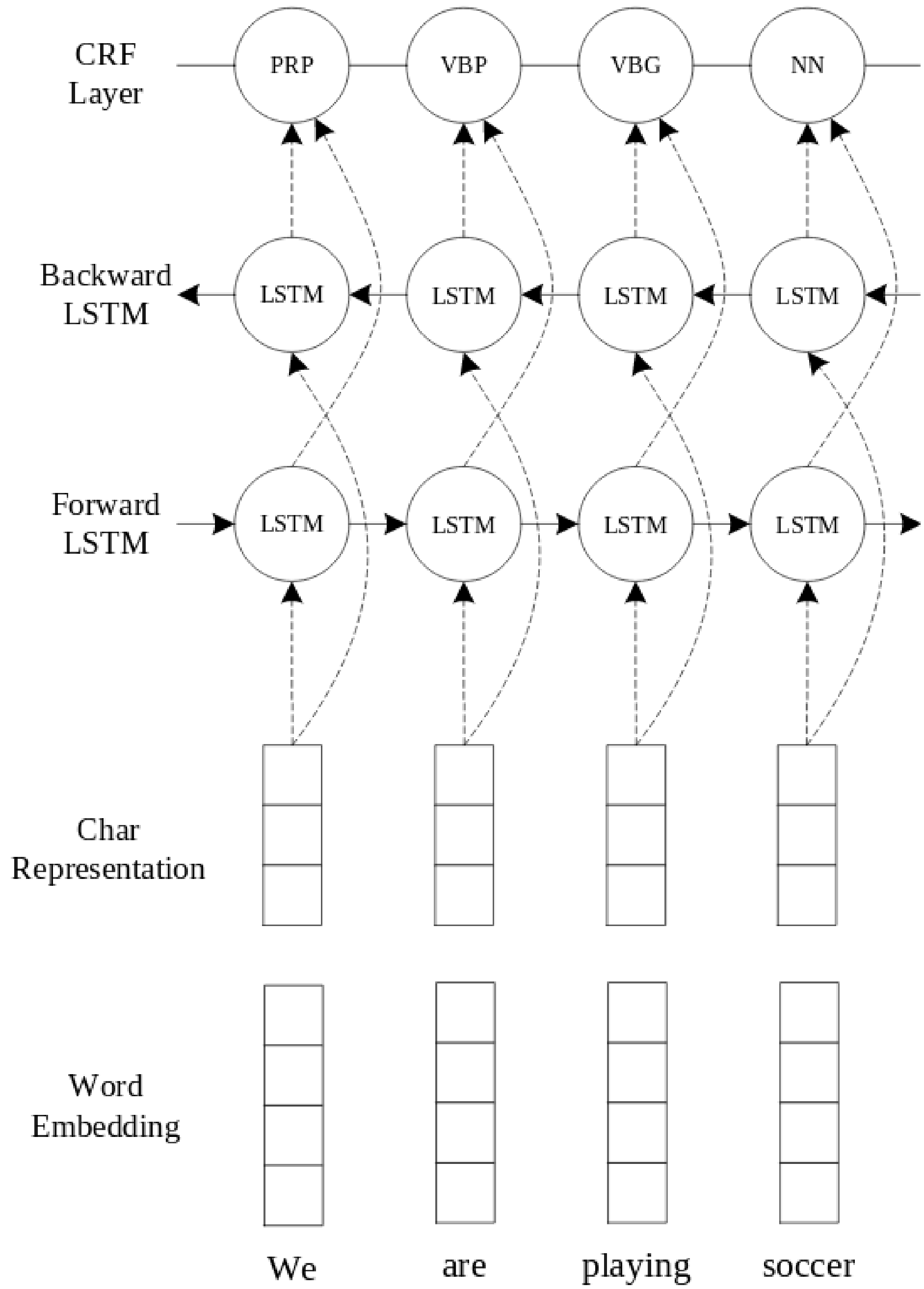

Figure 2.4.1 Rest of the Model

Figure 2.4.1 represents the implementation of a word-level encoder with an LSTM network. The word embeddings generated are a combination of glove + char embedding, as mentioned above. This is then fed into a bi-directional LSTM model.

However, for our implementation, we have to convert this LSTM model into a CNN model. To do so, we added the following modifications in the respective files in the figures below:

```
#Create cnn layer for word encoder
if self.word_mode == 'CNN':
    self.word_cnn = nn.Conv2d(in_channels=1, out_channels=400, kernel_size=(1, embedding_dim+self.out_channels))
```

Figure 2.4.2 CNN layer in BiLSTM_CRF Class

```
elif self.word_mode == 'CNN':
    embeds = embeds.unsqueeze(1)
    lstm_out = self.word_cnn(embeds)
    lstm_out = nn.functional.max_pool2d(lstm_out,
                                        kernel_size=(lstm_out.size(2), 1))
    ## Reshaping the outputs from the cnn layer
    lstm_out = torch.squeeze(lstm_out)
```

Figure 2.4.3 CNN layer used in get_lstm_features() function

(v) Results of running with one CNN Layer for Word Level Encoder

Below is the test result of using a bi-directional LSTM model for the word level encoder.

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


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

Figure 2.5.1 Bi-Directional LSTM for Word Level Encoder

However, when running the test set using one CNN layer implemented in (iv), we get the following results:

```
Prediction:
word : tag
Jay : PER
is : NA
from : NA
India : ORG


Donald : NA
is : ORG
the : NA
president : NA
of : ORG
USA : NA
```

Figure 2.5.2 One CNN layer for Word Level Encoder

The test results are mostly different in Figure 2.5.1 and Figure 2.5.2. This also shows that having one CNN layer for the word level encoder is insufficient for a much more accurate prediction.

(vi) Finding the optimal number of CNN Layers for Word Level Encoder

We then work to increase the number of CNN layers for the model by making the following modifications, as shown below:

```
elif self.word_mode == 'CNN':
  lstm_out = embeds.unsqueeze(1)

  if self.no_cnn_layers > 1:
    for i in range (self.no_cnn_layers):
      lstm_out = self.word_cnn(lstm_out)
      lstm_out = nn.functional.max_pool2d(lstm_out,
                                          kernel_size=(lstm_out.size(2), 1))
      ## Dropout on the cnn output
      lstm_out = self.dropout(lstm_out)
  lstm_out = self.last_cnn(lstm_out)
  lstm_out = nn.functional.max_pool2d(lstm_out,
                                      kernel_size=(lstm_out.size(2), 1))
      ## Dropout on the cnn output
  lstm_out = self.dropout(lstm_out)
  ## Reshaping the outputs from the cnn layer
  lstm_out = torch.squeeze(lstm_out)
```

Figure 2.6.1 CNN layer used in get_lstm_features() function

We will increase the number of CNN layers up to 7 for testing.

```
Prediction:
word : tag
Jay : LOC
is : NA
from : ORG
India : NA


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

Figure 2.6.2 Two CNN Layers used

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


Donald : MISC
is : ORG
the : MISC
president : LOC
of : ORG
USA : ORG
```

Figure 2.6.3 Three CNN Layers used

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


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

Figure 2.6.4 Four CNN Layers used

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


Donald : PER
is : ORG
the : LOC
president : PER
of : LOC
USA : LOC
```

Figure 2.6.5 Five CNN Layers used

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


Donald : ORG
is : PER
the : ORG
president : PER
of : NA
USA : ORG
```

Figure 2.6.6 Six CNN Layers used

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


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

Figure 2.6.7 Seven CNN Layers used

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


Donald : PER
is : ORG
the : PER
president : LOC
of : ORG
USA : PER
```

Figure 2.6.8 Eight CNN Layers used

```
Prediction:
word : tag
Jay : LOC
is : ORG
from : PER
India : NA


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

Figure 2.6.9 Nine CNN Layers used

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


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

Figure 2.6.10 Ten CNN Layers used

```
Prediction:
word : tag
Jay : NA
is : NA
from : LOC
India : NA


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

Figure 2.6.11 Eleven CNN Layers used

In comparison with the validation set from the LSTM Layer, the optimal number of CNN Layers would be 10.

For our code, it appeared that CNN would be less accurate than a bi-directional LSTM for a word-level encoder.