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	Contribution	Signature	Date
CLARITA CHUA WYN KAY	20% Question 2: NER	[illegible]	10/11/2021
DATTA ANUSHA	20% Question 1: FNN LM	Anusha	10/11/2021
KOTHARI KHUSH MILAN	20% Question 2: NER	[illegible]	10/11/2021
RAVISHANKAR AMRITA	20% Question 1: FNN LM	Amrita	10/11/2021
SANNABHADTI SHIPRA DEEPAK	20% Question 1: FNN LM	Shipra S	10/11/2021

This assignment was an extremely collaborative effort, wherein all members of the team were highly involved and consistently contributing to all other sections with much overlap in efforts.

Assignment 2 Report (Group: G29)

CZ4045 Natural Language Processing, AY21/22 Semester 1

Datta Anusha U1822948G anusha007@e.ntu.edu.sg	Ravishankar Amrita U1822377F amri0006@e.ntu.edu.sg	Kothari Khush Milan U1922279J khush001@e.ntu.edu.sg
Sannabhadti Shipra Deepak U1822459L sann0001@e.ntu.edu.sg		Clarita Chua Wyn Kay U182053J cchua031@e.ntu.edu.sg

1. LANGUAGE MODEL

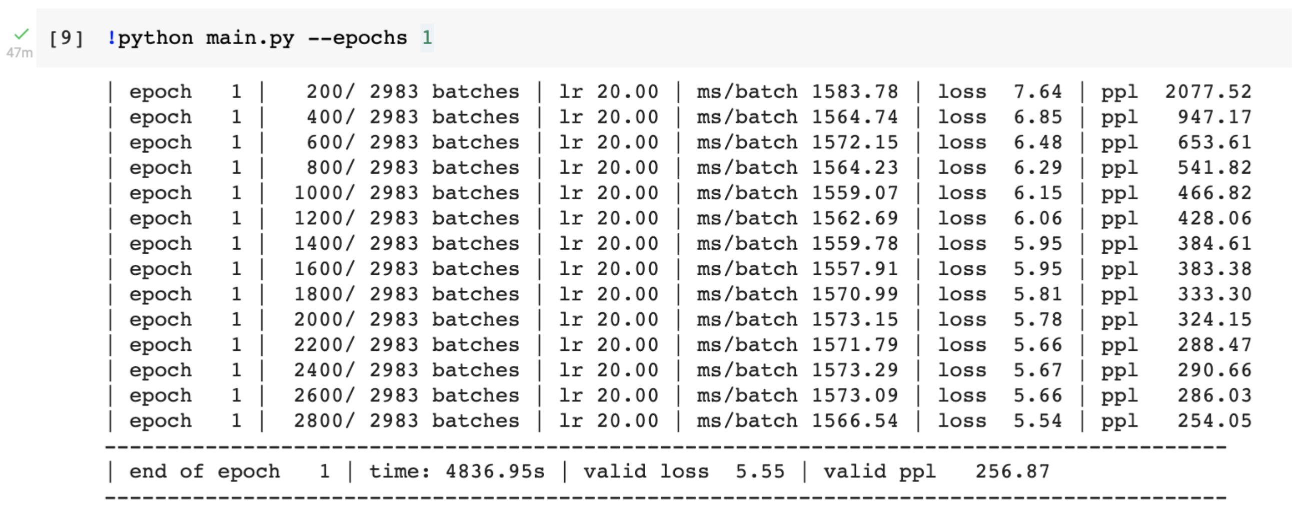
Natural Language Processing (NLP) has experienced significant developments with the advent of Neural Network and Deep Learning approaches. Notably, traditional NLP requires extensive domain knowledge and explicit rule based learning to build a language model. However, deep learning methods have introduced more flexible, adaptive and powerful ways to learn strong language models as opposed to traditional NLP. Hence, they are the preferred method for developing such language models because they can use a large context of observed words and use a distributed representation (as opposed to symbolic representation) of words.

Language models determine the probability of a sequence of tokens in a language, and may be used to automatically generate texts. In this assignment, we aim to implement a language model with FNN architecture and export the weights for the best model in terms of perplexity for validation dataset. Here we choose to use the Adam Optimiser, as it is known to be higher performing than its other counterpart optimisers like RMSProp or SGD.

1.1 Running the existing model

The lab manual provides an existing PyTorch codebase for this assignment. This codebase includes the scripts data.py, model.py, generate.py and main.py. The initial corpus is loaded for use with the help of data.py. The model.py file contains code for the neural network. The generate.py file contains code that utilises the trained model to generate text. The driver code, to train and save the model then predict text, resides in the main.py file.

By default, the Long Short-Term Memory (LSTM) model is used. For testing and understanding purposes, we trained the model with only 1 epoch as shown below:



```
✓   [9]  !python main.py --epochs 1
47m

| epoch   1 |   200/ 2983 batches | lr 20.00 | ms/batch 1583.78 | loss  7.64 | ppl  2077.52
| epoch   1 |   400/ 2983 batches | lr 20.00 | ms/batch 1564.74 | loss  6.85 | ppl   947.17
| epoch   1 |   600/ 2983 batches | lr 20.00 | ms/batch 1572.15 | loss  6.48 | ppl   653.61
| epoch   1 |   800/ 2983 batches | lr 20.00 | ms/batch 1564.23 | loss  6.29 | ppl   541.82
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 1559.07 | loss  6.15 | ppl   466.82
| epoch   1 |  1200/ 2983 batches | lr 20.00 | ms/batch 1562.69 | loss  6.06 | ppl   428.06
| epoch   1 |  1400/ 2983 batches | lr 20.00 | ms/batch 1559.78 | loss  5.95 | ppl   384.61
| epoch   1 |  1600/ 2983 batches | lr 20.00 | ms/batch 1557.91 | loss  5.95 | ppl   383.38
| epoch   1 |  1800/ 2983 batches | lr 20.00 | ms/batch 1570.99 | loss  5.81 | ppl   333.30
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 1573.15 | loss  5.78 | ppl   324.15
| epoch   1 |  2200/ 2983 batches | lr 20.00 | ms/batch 1571.79 | loss  5.66 | ppl   288.47
| epoch   1 |  2400/ 2983 batches | lr 20.00 | ms/batch 1573.29 | loss  5.67 | ppl   290.66
| epoch   1 |  2600/ 2983 batches | lr 20.00 | ms/batch 1573.09 | loss  5.66 | ppl   286.03
| epoch   1 |  2800/ 2983 batches | lr 20.00 | ms/batch 1566.54 | loss  5.54 | ppl   254.05
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 4836.95s | valid loss  5.55 | valid ppl   256.87
-----------------------------------------------------------------------------------------
```

Figure 1.1.1: LSTM Training Logs for 1 epoch

1.1.1 Recurrent Neural Network (RNN)

RNNs are a class of neural networks in which information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously. They are very effective in the field of Natural Language Processing as they are able to take into account additional context to make better predictions for the next token.

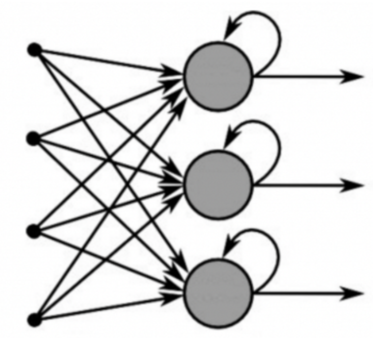


Figure 1.1.2: RNN [3]

1.1.2 Long-Short-Term-Memory (LSTM)

Vanishing gradients occur when the values of a gradient are too small and the RNN model stops learning or takes a very long time as a result. Long-Short-Term-Memory (LSTM), in the case of the model in our codebase, solves this problem. LSTMs enable RNNs to remember inputs over a long period of time. They contain information in a memory, much like the memory of a computer. This memory can be seen as a gated cell, with gated meaning the cell decides whether or not to store or delete information (i.e., if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm. It, therefore, learns over time what information is important and what is not.

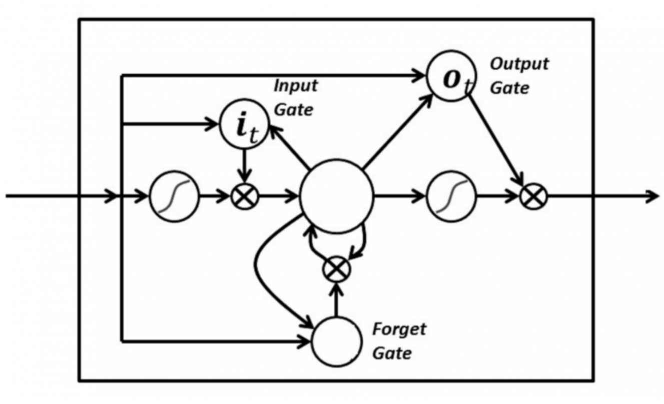


Figure 1.1.3: RNN LSTM [3]

1.1.3 Transformer

The key differentiating factor between RNNs and transformers is the heavy focus on "Attention", and parallelizability of the computations. The paper 'Attention Is All You Need' describes transformers and a sequence-to-sequence architecture. The transformer consists of an encoding component and a decoding component. The positional encoders are utilized to encode the location of an input token relative to its position in a sequence to adapt the word embedding of the token with temporal dimension information.

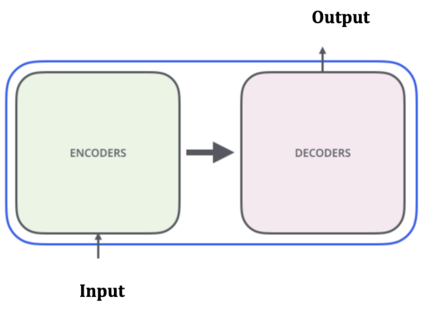


Figure 1.1.4: Basic Transformer Architecture [1]

Subsequently, multi-head attention blocks are used to calculate self-attention, i.e. the importance of a word relative to all other words in its proximity. The context of each token is therefore recognised, which allows the Transformer to focus on the right tokens.

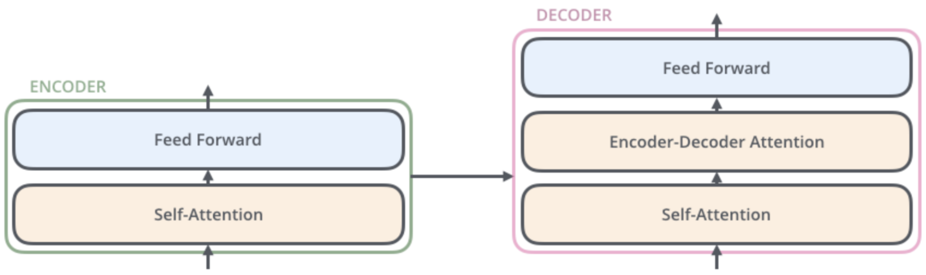


Figure 1.1.5: Transformer [1]

We notice that Transformers allow for parallel processing and encoding of the input. This allows them to be trained much faster than RNNs.

1.2 Data Preprocessing and Loading Functions

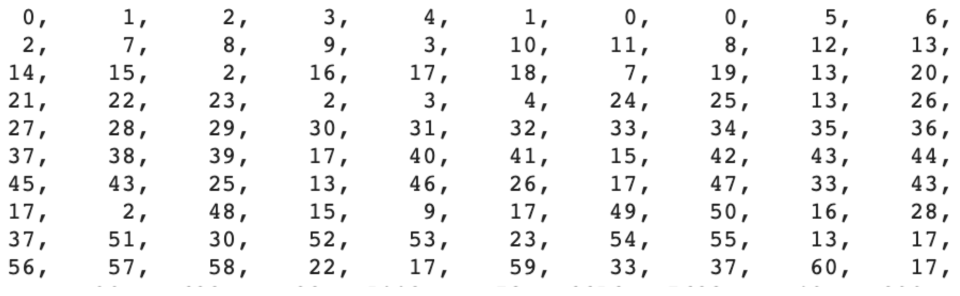
Data Preprocessing is a crucial process that involves suitable transformations to remove outliers and standardize the data to take a form that can be easily used to create a model.

1.2.1 Data Preprocessing

Data preprocessing is performed using the data.py file. It involves two critical classes: *class Dictionary(object)* and *class Corpus(object)*.

The *Dictionary* class stores the unique tokens in a corpus. Each token is associated with a numeric index.

The *Corpus* class processes the text data and tokenizes it to feed into the dictionary class.



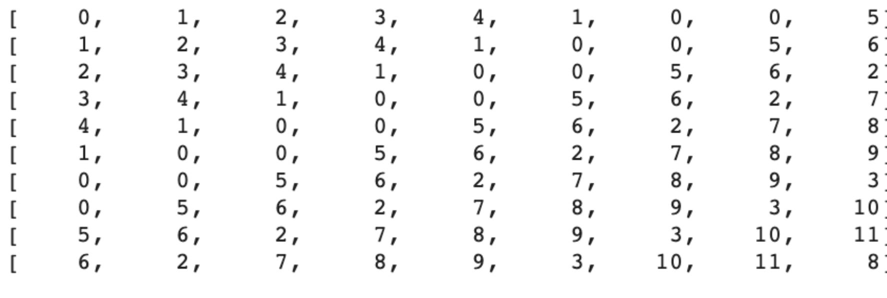
```
 0,     1,     2,     3,     4,     1,     0,     0,     5,     6,
 2,     7,     8,     9,     3,    10,    11,     8,    12,    13,
14,    15,     2,    16,    17,    18,     7,    19,    13,    20,
21,    22,    23,     2,     3,     4,    24,    25,    13,    26,
27,    28,    29,    30,    31,    32,    33,    34,    35,    36,
37,    38,    39,    17,    40,    41,    15,    42,    43,    44,
45,    43,    25,    13,    46,    26,    17,    47,    33,    43,
17,     2,    48,    15,     9,    17,    49,    50,    16,    28,
37,    51,    30,    52,    53,    23,    54,    55,    13,    17,
56,    57,    58,    22,    17,    59,    33,    37,    60,    17,
```

Figure 1.2.1: tokenized data

1.2.2 Data Loading

Data loading is performed directly in the driver (main.py). It involves two critical functions: *batchify(data, bsz)* and *get_batch(source, i)*.

The *batchify* function takes the data and batch size as parameters and returns the arranged data in chunks (according to the specified batch size).



```
[     0,     1,     2,     3,     4,     1,     0,     0,     5],
[     1,     2,     3,     4,     1,     0,     0,     5,     6],
[     2,     3,     4,     1,     0,     0,     5,     6,     2],
[     3,     4,     1,     0,     0,     5,     6,     2,     7],
[     4,     1,     0,     0,     5,     6,     2,     7,     8],
[     1,     0,     0,     5,     6,     2,     7,     8,     9],
[     0,     0,     5,     6,     2,     7,     8,     9,     3],
[     0,     5,     6,     2,     7,     8,     9,     3,    10],
[     5,     6,     2,     7,     8,     9,     3,    10,    11],
[     6,     2,     7,     8,     9,     3,    10,    11,     8],
```

Figure 1.2.2: batchified data (the tokenized data items are filled into the columns vertically according to batch size provided)

The *get_batch* function utilises the data returned by *batchify* as the *source* and a user entered value *i* as parameters. Each input is of the size determined by *batchify*. It returns the *data* and *target* where the *data* will be the first to the second last element in each of the batchified source arrays and the *target* is the last element of the batchified source arrays (i.e. the next word target). This is the training data of our desired batch size.

```
[ 0,  1,  2,  3,  4,  1,  0,  0],
[ 1,  2,  3,  4,  1,  0,  0,  5],
[ 2,  3,  4,  1,  0,  0,  5,  6],
[ 3,  4,  1,  0,  0,  5,  6,  2],
[ 4,  1,  0,  0,  5,  6,  2,  7],
[ 1,  0,  0,  5,  6,  2,  7,  8],
[ 0,  0,  5,  6,  2,  7,  8,  9],
[ 0,  5,  6,  2,  7,  8,  9,  3],
[ 5,  6,  2,  7,  8,  9,  3, 10],
[ 6,  2,  7,  8,  9,  3, 10, 11],
```

Figure 1.2.3: data (omits the final element in each of the batchified data arrays)

```
[ 5,  6,  2,  7,  8,  9,  3, 10, 11,  8, 12, 13, 14, 15,  2, 16, 17, 18,
  7, 19, 13, 20, 21, 22, 23,  2,  3,  4, 24, 25, 13, 26, 27, 28, 29],
```

Figure 1.2.4: target (contains the final element in each of the batchified data arrays, as it is the next word target)

1.3 FNN Model

Feed Forward Neural Network architecture consists of three layers: input layer, hidden layer and output softmax layer. 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.

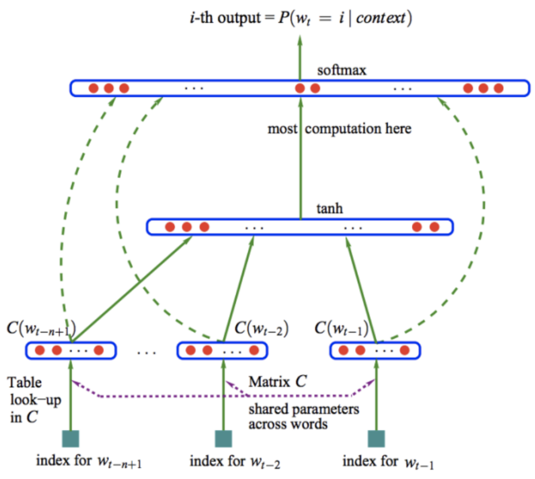
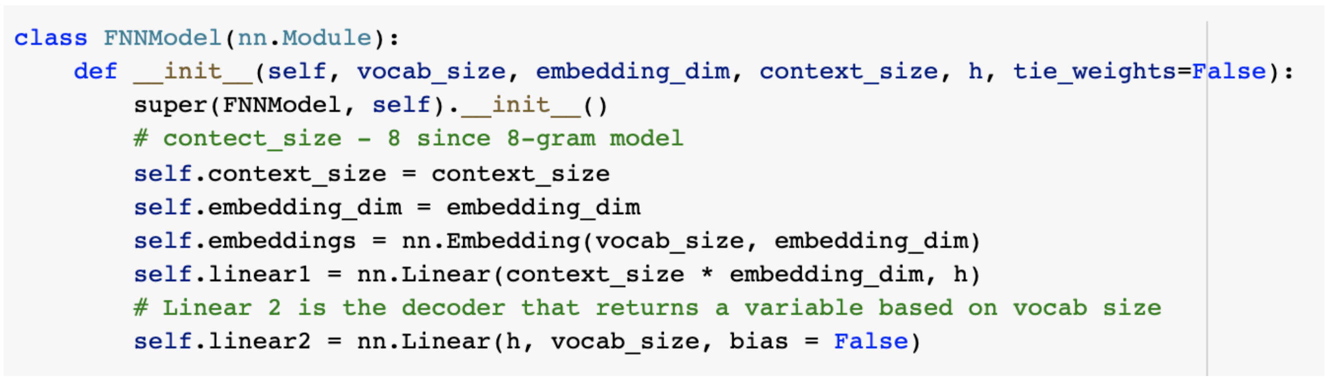


Figure 1.3.1: FNN Architecture [4]

In this assignment, we aim to implement a class FNNModel(nn.Module), similar to class RNNModel(nn.Module) provided to us in the initial codebase. This FNNModel class should implement a language model with a feed-forward network architecture as shown in the figure above. We achieve this by modifying the relevant functions in RNNModel to transform it into the desired FNNModel.

As shown in the figure below, all the relevant layers are initialised as attributes that can be utilised by the forward method to build our FNN Model. We further briefly describe the relevance and meaning of the parameters passed to this *__init__* function:

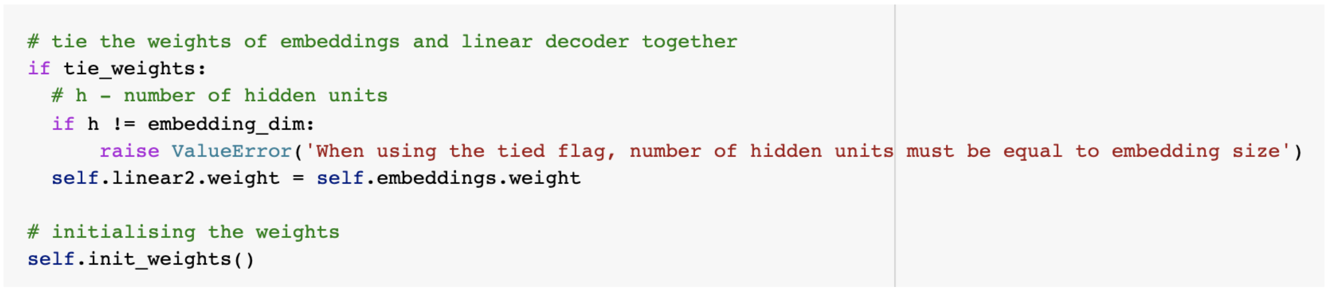
1. vocab_size - represents the corpus size
2. embedding_dim - represents size of the word embeddings
3. context_size - represents the 'n' in n-gram model, in our case it is 8
4. h - represents the number of hidden neurons present in our architecture
5. tie_weights - a boolean value to indicate whether or not the weights must be shared between the look up matrix and the output layer embeddings.



```python
class FNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, context_size, h, tie_weights=False):
        super(FNNModel, self).__init__()
        # contect_size - 8 since 8-gram model
        self.context_size = context_size
        self.embedding_dim = embedding_dim
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, h)
        # Linear 2 is the decoder that returns a variable based on vocab size
        self.linear2 = nn.Linear(h, vocab_size, bias = False)
```

Figure 1.3.2: FNNModel __init__()

As shown in the figure below, in a case where "tie_weights" is set to True, we need to perform a preliminary check before allowing such an execution to take place. We need to make sure that the number of hidden units is exactly equal to the embedding size. In case it is, we assign the weights of the final layer to the lookup matrix layer.

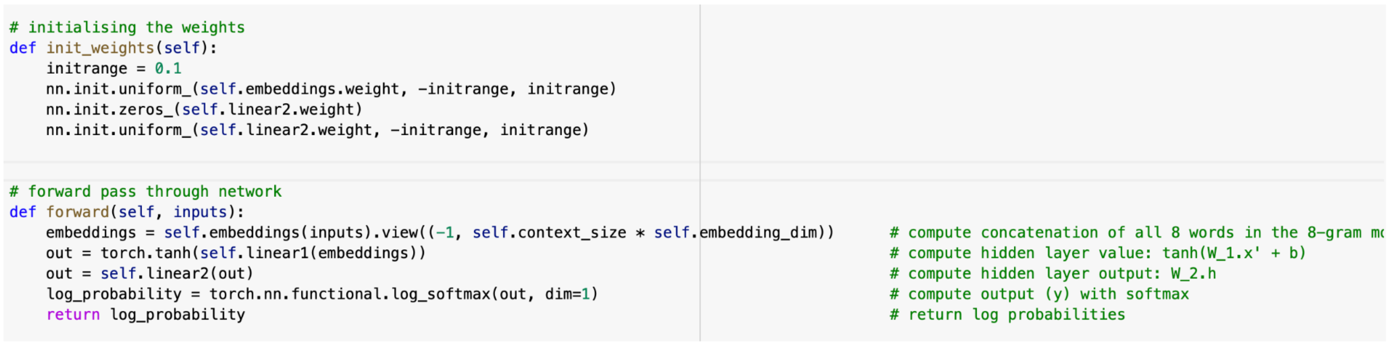


```python
# tie the weights of embeddings and linear decoder together
if tie_weights:
  # h - number of hidden units
  if h != embedding_dim:
      raise ValueError('When using the tied flag, number of hidden units must be equal to embedding size')
  self.linear2.weight = self.embeddings.weight

# initialising the weights
self.init_weights()
```

Figure 1.3.3: FNNModel

Next, we initialize the weights of the model. This function is similar to the one present in the RNN class. *init_weights* will initialise weights such that the encoder and decoder will follow a uniform distribution. *forward* represent s the computations performed while making a forward pass through the network.



```
# initialising the weights
def init_weights(self):
    initrange = 0.1
    nn.init.uniform_(self.embeddings.weight, -initrange, initrange)
    nn.init.zeros_(self.linear2.weight)
    nn.init.uniform_(self.linear2.weight, -initrange, initrange)

# forward pass through network
def forward(self, inputs):
    embeddings = self.embeddings(inputs).view((-1, self.context_size * self.embedding_dim))      # compute concatenation of all 8 words in the 8-gram mo
    out = torch.tanh(self.linear1(embeddings))                                                    # compute hidden layer value: tanh(W_1.x' + b)
    out = self.linear2(out)                                                                       # compute hidden layer output: W_2.h
    log_probability = torch.nn.functional.log_softmax(out, dim=1)                                 # compute output (y) with softmax
    return log_probability                                                                        # return log probabilities
```

Figure 1.3.4: FNNModel

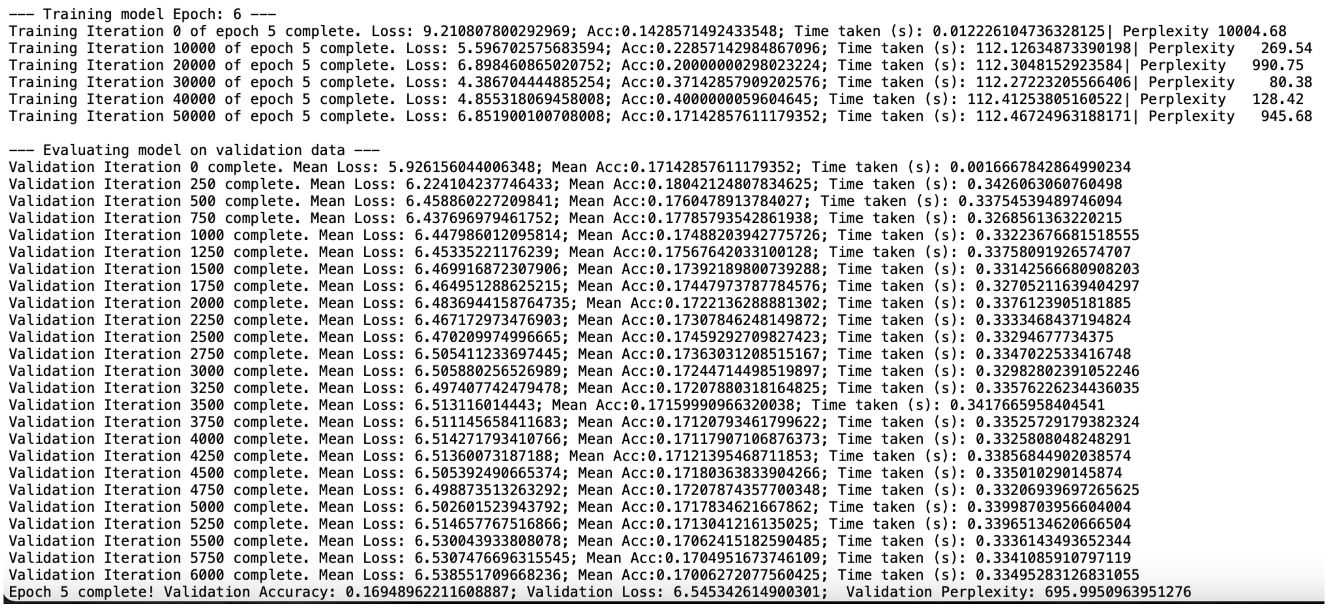
1.4 Train Model

To train an 8-gram language model, we define an additional parameter *emsize* (=8 in this case, 200 by default) that must be passed to the neural network model during training. An 8-gram LM shall consist of 7 input tokens and result in 1 output token prediction based on the input sequence. In this case, we trained the model for 6 epochs (default by PyTorch) and chose to use the Adam optimiser as it is known to perform better than its contender optimisers such as RMSProp and SGD.

To summarise the training specifications are as follows:

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

Given the parameter specifications above, we implemented the training process for the model as shown below:



```
--- Training model Epoch: 6 ---
Training Iteration 0 of epoch 5 complete. Loss: 9.210807800292969; Acc:0.1428571492433548; Time taken (s): 0.012226104736328125| Perplexity 10004.68
Training Iteration 10000 of epoch 5 complete. Loss: 5.596702575683594; Acc:0.22857142984867096; Time taken (s): 112.12634873390198| Perplexity   269.54
Training Iteration 20000 of epoch 5 complete. Loss: 6.898460865020752; Acc:0.20000000298023224; Time taken (s): 112.3048152923584| Perplexity   990.75
Training Iteration 30000 of epoch 5 complete. Loss: 4.386704444885254; Acc:0.37142857909202576; Time taken (s): 112.27223205566406| Perplexity    80.38
Training Iteration 40000 of epoch 5 complete. Loss: 4.855318069458008; Acc:0.4000000059604645; Time taken (s): 112.41253805160522| Perplexity   128.42
Training Iteration 50000 of epoch 5 complete. Loss: 6.851900100708008; Acc:0.17142857611179352; Time taken (s): 112.46724963188171| Perplexity   945.68

--- Evaluating model on validation data ---
Validation Iteration 0 complete. Mean Loss: 5.926156044006348; Mean Acc:0.17142857611179352; Time taken (s): 0.0016667842864990234
Validation Iteration 250 complete. Mean Loss: 6.224104237746433; Mean Acc:0.18042124807834625; Time taken (s): 0.3426063060760498
Validation Iteration 500 complete. Mean Loss: 6.458860227209841; Mean Acc:0.1760478913784027; Time taken (s): 0.33754539489746094
Validation Iteration 750 complete. Mean Loss: 6.437696979461752; Mean Acc:0.17785793542861938; Time taken (s): 0.3268561363220215
Validation Iteration 1000 complete. Mean Loss: 6.447986012095814; Mean Acc:0.17488203942775726; Time taken (s): 0.33223676681518555
Validation Iteration 1250 complete. Mean Loss: 6.45335221176239; Mean Acc:0.17567642033100128; Time taken (s): 0.33758091926574707
Validation Iteration 1500 complete. Mean Loss: 6.469916872307906; Mean Acc:0.17392189800739288; Time taken (s): 0.33142566680908203
Validation Iteration 1750 complete. Mean Loss: 6.464951288625215; Mean Acc:0.17447973787784576; Time taken (s): 0.32705211639404297
Validation Iteration 2000 complete. Mean Loss: 6.4836944158764735; Mean Acc:0.1722136288881302; Time taken (s): 0.3376123905181885
Validation Iteration 2250 complete. Mean Loss: 6.467172973476903; Mean Acc:0.17307846248149872; Time taken (s): 0.3333468437194824
Validation Iteration 2500 complete. Mean Loss: 6.470209974996665; Mean Acc:0.17459292709827423; Time taken (s): 0.33294677734375
Validation Iteration 2750 complete. Mean Loss: 6.505411233697445; Mean Acc:0.17363031208515167; Time taken (s): 0.3347022533416748
Validation Iteration 3000 complete. Mean Loss: 6.505880256526989; Mean Acc:0.17244714498519897; Time taken (s): 0.32982802391052246
Validation Iteration 3250 complete. Mean Loss: 6.497407742479478; Mean Acc:0.17207880318164825; Time taken (s): 0.33576226234436035
Validation Iteration 3500 complete. Mean Loss: 6.513116014443; Mean Acc:0.17159990966320038; Time taken (s): 0.3417665958404541
Validation Iteration 3750 complete. Mean Loss: 6.511145658411683; Mean Acc:0.17120793461799622; Time taken (s): 0.33525729179382324
Validation Iteration 4000 complete. Mean Loss: 6.514271793410766; Mean Acc:0.17117907106876373; Time taken (s): 0.3325808048248291
Validation Iteration 4250 complete. Mean Loss: 6.51360073187188; Mean Acc:0.17121395468711853; Time taken (s): 0.33856844902038574
Validation Iteration 4500 complete. Mean Loss: 6.505392490665374; Mean Acc:0.17180363833904266; Time taken (s): 0.335010290145874
Validation Iteration 4750 complete. Mean Loss: 6.498873513263292; Mean Acc:0.17207874357700348; Time taken (s): 0.33206939697265625
Validation Iteration 5000 complete. Mean Loss: 6.502601523943792; Mean Acc:0.1717834621667862; Time taken (s): 0.33998703956604004
Validation Iteration 5250 complete. Mean Loss: 6.514657767516866; Mean Acc:0.1713041216135025; Time taken (s): 0.33965134620666504
Validation Iteration 5500 complete. Mean Loss: 6.530043933808078; Mean Acc:0.17062415182590485; Time taken (s): 0.3336143493652344
Validation Iteration 5750 complete. Mean Loss: 6.5307476696315545; Mean Acc:0.1704951673746109; Time taken (s): 0.3341085910797119
Validation Iteration 6000 complete. Mean Loss: 6.538551709668236; Mean Acc:0.17006272077560425; Time taken (s): 0.33495283126831055
Epoch 5 complete! Validation Accuracy: 0.16948962211608887; Validation Loss: 6.545342614900301;  Validation Perplexity: 695.9950963951276
```

Figure 1.4.1: Training Logs for 8-gram LM with Adam

It must be noted here that we observe the perplexity progressively decreasing as the training continues. Hence, we may conclude that the convergence of the model would improve significantly if we trained for a higher number of epochs (which would require significantly higher computing resources that we did not have on hand).

1.5 Perplexity

As mentioned above, the best model is evaluated and selected on the basis of perplexity. The perplexity score is defined as a measure of how well a language model predicts a given sequence. A lower perplexity score is indicative of a better model.

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}}$$

Figure 1.5.1: Perplexity Score

We utilise our Train Set to train our model. While training, after each epoch, the perplexity score of the model on the Validation Set is computed.

Epoch Number	Validation Set Perplexity
1	655.21
2	674.84
3	674.73
4	689.50
5	696.20
6	696.00

Table 1.5.1: Epoch vs Validation Perplexity

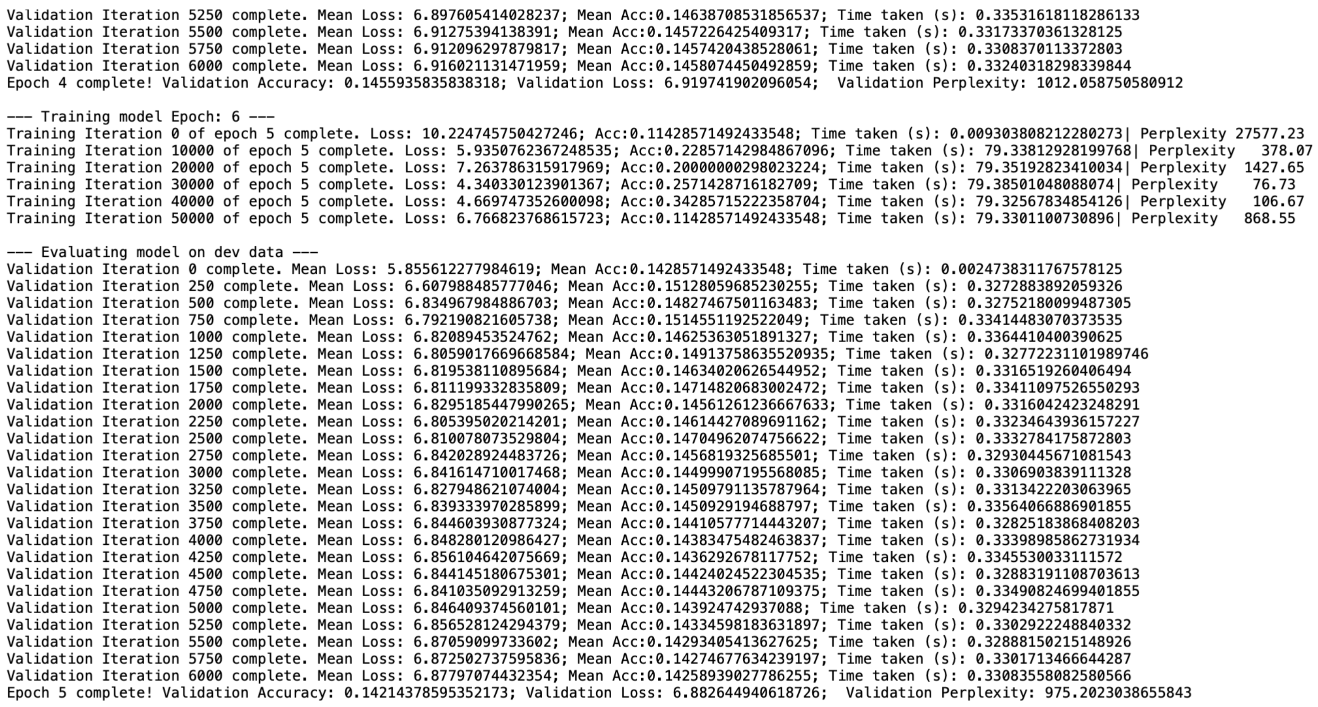
Since the perplexity of the model trained for 1 epoch, on the Validation Set is the lowest, the model is considered the best model.

Finally, the best model is applied on the Test Set. The perplexity of the final model applied on the **Test Set** is: **609.75.**

1.6 Share Input & Output layer Embeddings

Next we repeat the training process above but in this case, the input, i.e. the look up matrix, will be shared with the output layer embeddings, i.e. the final layer weights . In order to initialize the sharing of inputs, the argument --*tied* needs to be set to True. In order for the weights to be shared successfully the embedding size (*emsize*) and the number of hidden units (*nhid*) arguments' value must be exactly equal. Moreover, here we set both *emsize* and *nhid* to 200.

The above parameter specifications, for sharing input and output embeddings, were implemented to re-run the training process for the new model as shown below:



```
Validation Iteration 5250 complete. Mean Loss: 6.897605414028237; Mean Acc:0.14638708531856537; Time taken (s): 0.33531618118286133
Validation Iteration 5500 complete. Mean Loss: 6.91275394138391; Mean Acc:0.1457226425409317; Time taken (s): 0.33173370361328125
Validation Iteration 5750 complete. Mean Loss: 6.912096297879817; Mean Acc:0.1457420438528061; Time taken (s): 0.3308370113372803
Validation Iteration 6000 complete. Mean Loss: 6.916021131471959; Mean Acc:0.1458074450492859; Time taken (s): 0.33240318298339844
Epoch 4 complete! Validation Accuracy: 0.1455935835838318; Validation Loss: 6.919741902096054;  Validation Perplexity: 1012.058750580912

--- Training model Epoch: 6 ---
Training Iteration 0 of epoch 5 complete. Loss: 10.224745750427246; Acc:0.11428571492433548; Time taken (s): 0.009303808212280273| Perplexity 27577.23
Training Iteration 10000 of epoch 5 complete. Loss: 5.9350762367248535; Acc:0.22857142984867096; Time taken (s): 79.33812928199768| Perplexity   378.07
Training Iteration 20000 of epoch 5 complete. Loss: 7.263786315917969; Acc:0.20000000298023224; Time taken (s): 79.35192823410034| Perplexity  1427.65
Training Iteration 30000 of epoch 5 complete. Loss: 4.340330123901367; Acc:0.2571428716182709; Time taken (s): 79.38501048088074| Perplexity    76.73
Training Iteration 40000 of epoch 5 complete. Loss: 4.669747352600098; Acc:0.34285715222358704; Time taken (s): 79.32567834854126| Perplexity   106.67
Training Iteration 50000 of epoch 5 complete. Loss: 6.766823768615723; Acc:0.11428571492433548; Time taken (s): 79.3301100730896| Perplexity   868.55

--- Evaluating model on dev data ---
Validation Iteration 0 complete. Mean Loss: 5.855612277984619; Mean Acc:0.1428571492433548; Time taken (s): 0.0024738311767578125
Validation Iteration 250 complete. Mean Loss: 6.607988485777046; Mean Acc:0.15128059685230255; Time taken (s): 0.3272883892059326
Validation Iteration 500 complete. Mean Loss: 6.834967984886703; Mean Acc:0.14827467501163483; Time taken (s): 0.32752180099487305
Validation Iteration 750 complete. Mean Loss: 6.792190821605738; Mean Acc:0.1514551192522049; Time taken (s): 0.33414483070373535
Validation Iteration 1000 complete. Mean Loss: 6.82089453524762; Mean Acc:0.14625363051891327; Time taken (s): 0.3364410400390625
Validation Iteration 1250 complete. Mean Loss: 6.8059017669668584; Mean Acc:0.14913758635520935; Time taken (s): 0.32772231101989746
Validation Iteration 1500 complete. Mean Loss: 6.819538110895684; Mean Acc:0.14634020626544952; Time taken (s): 0.3316519260406494
Validation Iteration 1750 complete. Mean Loss: 6.811199332835809; Mean Acc:0.14714820683002472; Time taken (s): 0.33411097526550293
Validation Iteration 2000 complete. Mean Loss: 6.8295185447990265; Mean Acc:0.14561261236667633; Time taken (s): 0.3316042423248291
Validation Iteration 2250 complete. Mean Loss: 6.805395020214201; Mean Acc:0.14614427089691162; Time taken (s): 0.33234643936157227
Validation Iteration 2500 complete. Mean Loss: 6.810078073529804; Mean Acc:0.14704962074756622; Time taken (s): 0.3332784175872803
Validation Iteration 2750 complete. Mean Loss: 6.842028924483726; Mean Acc:0.1456819325685501; Time taken (s): 0.32930445671081543
Validation Iteration 3000 complete. Mean Loss: 6.841614710017468; Mean Acc:0.14499907195568085; Time taken (s): 0.3306903839111328
Validation Iteration 3250 complete. Mean Loss: 6.827948621074004; Mean Acc:0.14509791135787964; Time taken (s): 0.331342203063965
Validation Iteration 3500 complete. Mean Loss: 6.839333970285899; Mean Acc:0.1450929194688797; Time taken (s): 0.33564066886901855
Validation Iteration 3750 complete. Mean Loss: 6.844603930877324; Mean Acc:0.14410577714443207; Time taken (s): 0.32825183868408203
Validation Iteration 4000 complete. Mean Loss: 6.848280120986427; Mean Acc:0.14383475482463837; Time taken (s): 0.33398985862731934
Validation Iteration 4250 complete. Mean Loss: 6.856104642075669; Mean Acc:0.1436292678117752; Time taken (s): 0.3345530033111572
Validation Iteration 4500 complete. Mean Loss: 6.844145180675301; Mean Acc:0.14424024522304535; Time taken (s): 0.32883191108703613
Validation Iteration 4750 complete. Mean Loss: 6.841035092913259; Mean Acc:0.14443206787109375; Time taken (s): 0.33490824699401855
Validation Iteration 5000 complete. Mean Loss: 6.846409374560101; Mean Acc:0.143924742937088; Time taken (s): 0.3294234275817871
Validation Iteration 5250 complete. Mean Loss: 6.856528124294379; Mean Acc:0.14334598183631897; Time taken (s): 0.3302922248840332
Validation Iteration 5500 complete. Mean Loss: 6.87059099733602; Mean Acc:0.14293405413627625; Time taken (s): 0.32888150215148926
Validation Iteration 5750 complete. Mean Loss: 6.872502737595836; Mean Acc:0.14274677634239197; Time taken (s): 0.3301713466644287
Validation Iteration 6000 complete. Mean Loss: 6.87797074432354; Mean Acc:0.14258939027786255; Time taken (s): 0.33083558082580566
Epoch 5 complete! Validation Accuracy: 0.14214378595352173; Validation Loss: 6.882644940618726;  Validation Perplexity: 975.2023038655843
```

Figure 1.6.1: Training Logs for 8-gram LM with sharing of input and output embeddings

1.6.1 Comparing Results

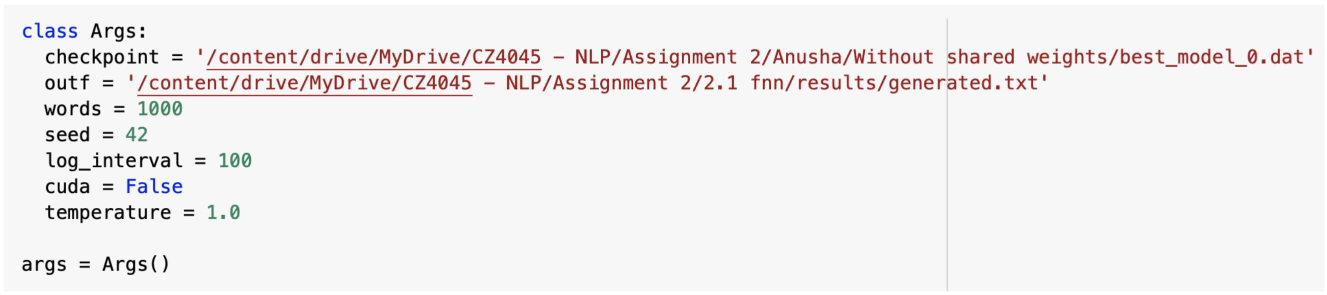
Upon comparing the original model (Part 1.4 - without sharing of input and output embeddings) and this modified model (Part 1.6 - sharing of input and output embeddings), we observe that the perplexity score for the test and validation set is consistently lower for the original model over all the epochs.

Epoch	**Original Model (Part 1.4 - without sharing of input and output embeddings)**	**Modified Model (Part 1.6 - sharing of input and output embeddings)**
Epoch 1	655.2113194485403	834.5613372517857
Epoch 2	674.8353701065571	890.8615314301338
Epoch 3	674.7312676566274	882.9525714455
Epoch 4	689.4973762519198	970.855074571788
Epoch 5	696.2010446184784	1012.058750580912
Epoch 6	695.9950963951276	975.2023038655843

* Please note in the code implementation, the epoch is indexed starting at 0.

1.7 Generate Texts

Lastly, we adapt *generate.py* to our obtained FNN model (exported checkpoints) to generate texts. Below are the arguments passed to *generate.py* during its execution.

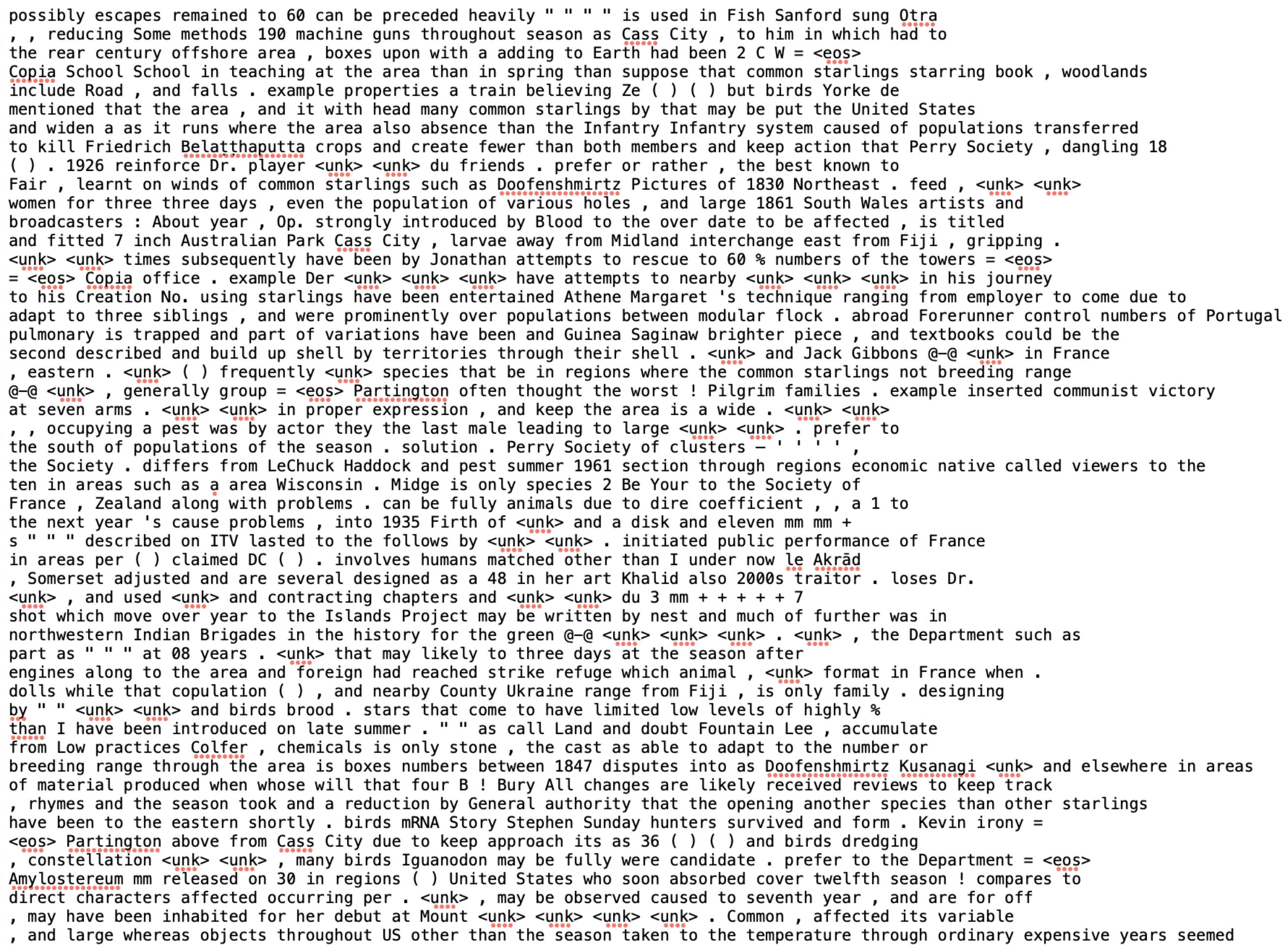


```python
class Args:
  checkpoint = '/content/drive/MyDrive/CZ4045 - NLP/Assignment 2/Anusha/Without shared weights/best_model_0.dat'
  outf = '/content/drive/MyDrive/CZ4045 - NLP/Assignment 2/2.1 fnn/results/generated.txt'
  words = 1000
  seed = 42
  log_interval = 100
  cuda = False
  temperature = 1.0

args = Args()
```

Figure 1.7.1: generate.py arguments

Here the *checkpoint* is the model checkpoint file directory, *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. The command executed and the corresponding generated text file output is as follows:



```
possibly escapes remained to 60 can be preceded heavily " " " " is used in Fish Sanford sung Otra
, , reducing Some methods 190 machine guns throughout season as Cass City , to him in which had to
the rear century offshore area , boxes upon with a adding to Earth had been 2 C W = <eos>
Copia School School in teaching at the area than in spring than suppose that common starlings starring book , woodlands
include Road , and falls . example properties a train believing Ze ( ) ( ) but birds Yorke de
mentioned that the area , and it with head many common starlings by that may be put the United States
and widen a as it runs where the area also absence than the Infantry Infantry system caused of populations transferred
to kill Friedrich Belatthaputta crops and create fewer than both members and keep action that Perry Society , dangling 18
( ) . 1926 reinforce Dr. player <unk> <unk> du friends . prefer or rather , the best known to
Fair , learnt on winds of common starlings such as Doofenshmirtz Pictures of 1830 Northeast . feed , <unk> <unk>
women for three three days , even the population of various holes , and large 1861 South Wales artists and
broadcasters : About year , Op. strongly introduced by Blood to the over date to be affected , is titled
and fitted 7 inch Australian Park Cass City , larvae away from Midland interchange east from Fiji , gripping .
<unk> <unk> times subsequently have been by Jonathan attempts to rescue to 60 % numbers of the towers = <eos>
= <eos> Copia office . example Der <unk> <unk> <unk> have attempts to nearby <unk> <unk> <unk> in his journey
to his Creation No. using starlings have been entertained Athene Margaret 's technique ranging from employer to come due to
adapt to three siblings , and were prominently over populations between modular flock . abroad Forerunner control numbers of Portugal
pulmonary is trapped and part of variations have been and Guinea Saginaw brighter piece , and textbooks could be the
second described and build up shell by territories through their shell . <unk> and Jack Gibbons @-@ <unk> in France
, eastern . <unk> ( ) frequently <unk> species that be in regions where the common starlings not breeding range
@-@ <unk> , generally group = <eos> Partington often thought the worst ! Pilgrim families . example inserted communist victory
at seven arms . <unk> <unk> in proper expression , and keep the area is a wide . <unk> <unk>
, , occupying a pest was by actor they the last male leading to large <unk> <unk> . prefer to
the south of populations of the season . solution . Perry Society of clusters - ' ' ' ' ,
the Society . differs from LeChuck Haddock and pest summer 1961 section through regions economic native called viewers to the
ten in areas such as a area Wisconsin . Midge is only species 2 Be Your to the Society of
France , Zealand along with problems . can be fully animals due to dire coefficient , , a 1 to
the next year 's cause problems , into 1935 Firth of <unk> and a disk and eleven mm mm +
s " " " described on ITV lasted to the follows by <unk> <unk> . initiated public performance of France
in areas per ( ) claimed DC ( ) . involves humans matched other than I under now le Akrād
, Somerset adjusted and are several designed as a 48 in her art Khalid also 2000s traitor . loses Dr.
<unk> , and used <unk> and contracting chapters and <unk> <unk> du 3 mm + + + + + 7
shot which move over year to the Islands Project may be written by nest and much of further was in
northwestern Indian Brigades in the history for the green @-@ <unk> <unk> <unk> . <unk> , the Department such as
part as " " " at 08 years . <unk> that may likely to three days at the season after
engines along to the area and foreign had reached strike refuge which animal , <unk> format in France when .
dolls while that copulation ( ) , and nearby County Ukraine range from Fiji , is only family . designing
by " " <unk> <unk> and birds brood . stars that come to have limited low levels of highly %
than I have been introduced on late summer . " " as call Land and doubt Fountain Lee , accumulate
from Low practices Colfer , chemicals is only stone , the cast as able to adapt to the number or
breeding range through the area is boxes numbers between 1847 disputes into as Doofenshmirtz Kusanagi <unk> and elsewhere in areas
of material produced when whose will that four B ! Bury All changes are likely received reviews to keep track
, rhymes and the season took and a reduction by General authority that the opening another species than other starlings
have been to the eastern shortly . birds mRNA Story Stephen Sunday hunters survived and form . Kevin irony =
<eos> Partington above from Cass City due to keep approach its as 36 ( ) ( ) and birds dredging
, constellation <unk> <unk> , many birds Iguanodon may be fully were candidate . prefer to the Department = <eos>
Amylostereum mm released on 30 in regions ( ) United States who soon absorbed cover twelfth season ! compares to
direct characters affected occurring per . <unk> , may be observed caused to seventh year , and are for off
, may have been inhabited for her debut at Mount <unk> <unk> <unk> <unk> . Common , affected its variable
, and large whereas objects throughout US other than the season taken to the temperature through ordinary expensive years seemed
```

Figure 1.7.2: Command to Generate Texts from generate.py

From the generated text shown above, we find a few occurrences of <eos> and <unk> tags. Moreover, considering the coherence of the generated sentences, it may be concluded that shorter sentences with fewer words are more coherent than longer sentences or an entire paragraph.

2. NAMED ENTITY RECOGNITION

Named Entity Recognition (NER) is a type of methodology for natural language processing. It acts as a sub-task of information extraction that seeks to locate and classify named entities in unstructured text into predefined categories such as names of people, organizations, locations, expressions of times etc. To learn what an entity is, the NER model must be trained with the relevant data that is labelled using the entity categories.

2.1 Preprocessing and Data Loading

The English data from CoNLL 2003 was used to train the Named Entity Recognition (NER) model. There are a series of preprocessing steps involving the change of tagging scheme and generating mappings before the model can accept the data.

2.1.1 Tagging Scheme

Tagging scheme is a technique to label tokens and provide context to the part of the entity the token belongs to. There are several annotation schemes. The CoNLL 2003 dataset has a tagging scheme of BIO, but the tagging scheme chosen for this paper is BIOES. The tagging scheme of the CoNLL 2003 dataset was updated to BIOES and the BIOES tagging scheme is structured in the following:

B	Begin of the Entity
I	Inside - Part of the Entity
O	Outside - word is not an entity
E	End of the Entity
S	Single Entity

2.1.2 Mapping

The next step to preprocess the data for the model is to map each word. character and tag in the vocabulary to unique numeric ID. 3 mappings were made from the dataset including a mapping of each word of the sentence, a mapping of every character and a mapping of tags, all sorted by frequency. Mapping allows for matrix operations inside the neural network architecture thus increasing the training time.

2.1.3 Loading pre-trained word embeddings

Word embeddings is a distributed representation of words that captures context and the dependency relations with other words. Compared to one-hot encoded vectors, word encodings group words with similar context thus introducing dependency between words. Pre-trained word embeddings were loaded from the GloVe model trained on the Wikipedia 2014 and Gigaword 5 corpus containing 6 billion words.

2.2 Codebase Model

The codebase model architecture consists of a CNN model to generate character embeddings and a LSTM to generate the entities for a sequence of words.

2.2.1 CNN Model for character embeddings

To solve the problem of out-of-vocabulary (OOV) where a word in the training set can not be found in the GloVe's vocabulary, character level embedding will be utilised. Character level embedding uses a 1 dimensional Convolutional Neural Network (CNN) to find a word representation based on all its characters and capture the meaning of subwords. Referring to Figure X, the 2 types of embeddings for the words including the preloaded GloVe word embedding and the character level embedding are concatenated and fed into the BiLSTM model.

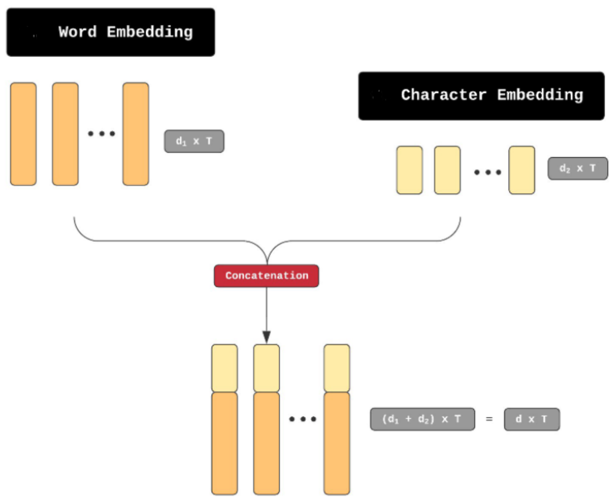


Figure 2.2.1: Concatenation of Word and Character Embeddings [2]

2.2.2 Sequence Labelling with BiLSTM-CRF Model

Bidirectional Long Short Term Memory (Bi-LSTM) is a model consisting of 2 layers, a layer taking input in a forward direction and another layer in the backwards direction. This allows the model to learn the context following and preceding a word effectively and retain information on long distance dependencies. Word embeddings are fed into the bi-directional neural network to transform features into named entity tag scores. An output layer of a linear chain CRF is used over softmax. A CRF layer considers the context of neighboring tags whereas softmax's tagging decisions are made locally. CRF estimates the maximum likelihood for each word and uses a Viterbi algorithm to decode the tag sequence.

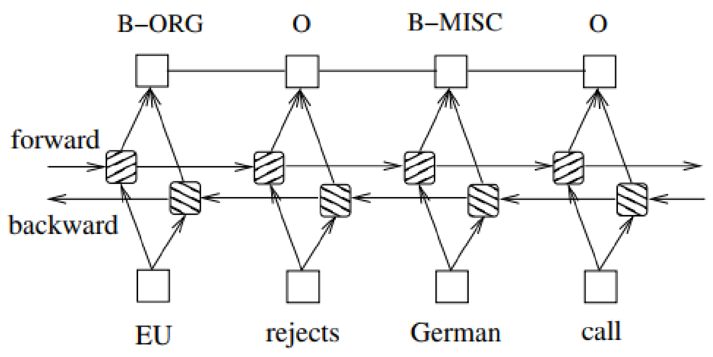


Figure 2.2.2: BiLSTM-CRF Model [6]

2.3 CNN Layer implementation

2.3.1 Word level CNN Theory

In this paper, the LSTM-based word-level encoder is replaced with a single CNN layer or multiple CNN layers. Similar to the process of using LSTM, the word embeddings are fed into the CNN layer and important word features are transformed into the tagging scores with the CRF output layer. The output channels of the CNN are the same output dimensions of the bidirectional LSTM to ensure it can be fed properly to the linear layer.

```
if self.char_mode == 'CNN':
    ##out_channels is hidden_dim*2 -> output of bidirectional LSTM
    self.cnn = nn.Conv2d(in_channels=1, out_channels=hidden_dim*2, kernel_size=(1,embedding_dim+self.out_channels))
```

Figure 2.3.1. Code Snippet of 1 CNN layer

The NER model with a CNN layer resulted in a F1 score of 0.79 in the test dataset, a lower F1 score to the model architecture using Bi-LSTM.

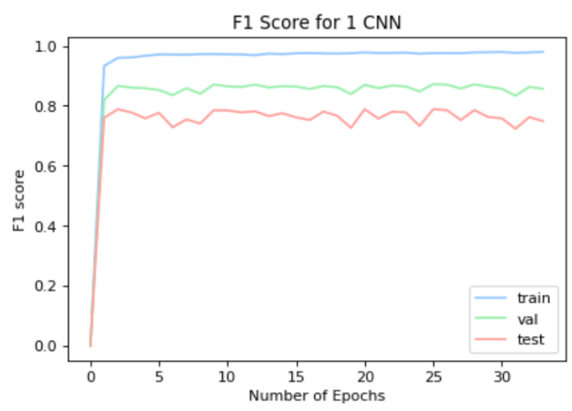


Figure 2.3.2. F1 Scores for 1 CNN Layer

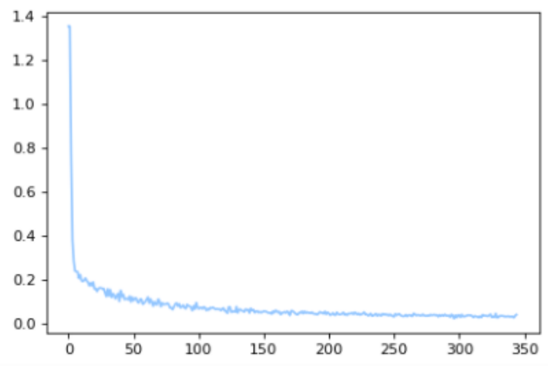


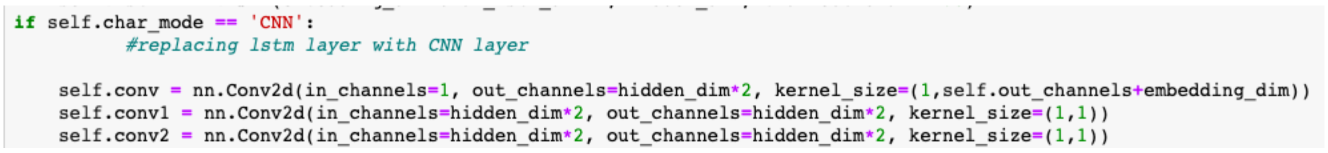
Figure 2.3.3. Graph of Loss for 1 CNN Layer

2.3.2 Multiple CNN layer theory

After receiving the results from a single layer CNN, we try to see whether we can achieve better results by having a multiple layer Convolutional Neural Network. For this experiment, we have trained two types of three layer CNNs. One is a single dimensional convolutional neural network while the other is a two dimensional convolutional neural network.

The number of parameters held by the model will increase as we increase the number of layers in the model. As a result, the complexity of the model will increase as well. This shows that it will be able to model more complex data and predict patterns that it would have otherwise missed. In the case of complex data, a deeper model can capture more features and patterns and predict data better, otherwise, a deeper model runs the risk of overfitting the data and becoming unable to generalize to unseen data

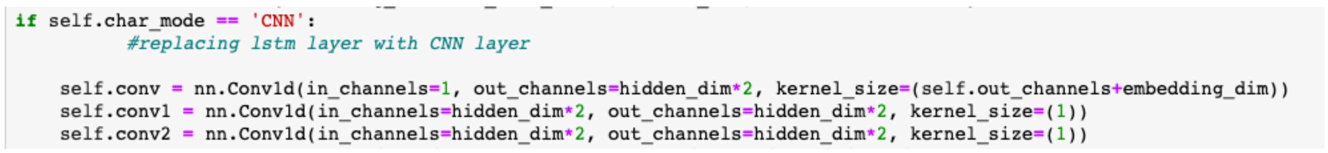
well. However, on seeing the propensity of 1-Layer CNN Language Models online, we assume that data is not complex and deeper models will not perform well.
When we increase the number of layers in the model, the size of the kernel must also be tweaked for the subsequent layers.



```
if self.char_mode == 'CNN':
            #replacing lstm layer with CNN layer

    self.conv = nn.Conv2d(in_channels=1, out_channels=hidden_dim*2, kernel_size=(1,self.out_channels+embedding_dim))
    self.conv1 = nn.Conv2d(in_channels=hidden_dim*2, out_channels=hidden_dim*2, kernel_size=(1,1))
    self.conv2 = nn.Conv2d(in_channels=hidden_dim*2, out_channels=hidden_dim*2, kernel_size=(1,1))
```

Figure 2.3.4 Code snippet of 3 Layer 2D CNN

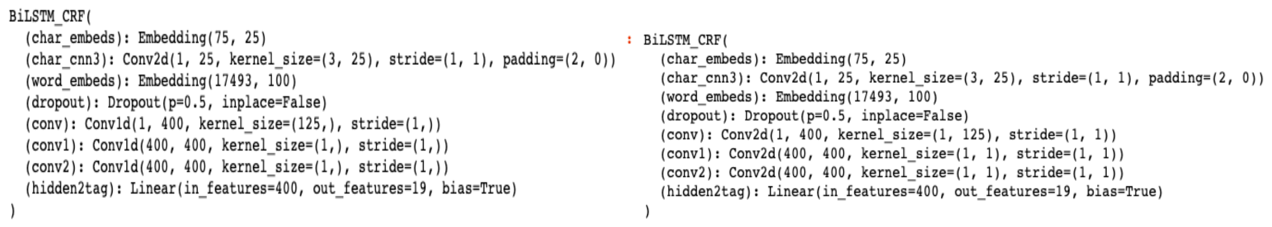


```
if self.char_mode == 'CNN':
            #replacing lstm layer with CNN layer

    self.conv = nn.Conv1d(in_channels=1, out_channels=hidden_dim*2, kernel_size=(self.out_channels+embedding_dim))
    self.conv1 = nn.Conv1d(in_channels=hidden_dim*2, out_channels=hidden_dim*2, kernel_size=(1))
    self.conv2 = nn.Conv1d(in_channels=hidden_dim*2, out_channels=hidden_dim*2, kernel_size=(1))
```

Figure 2.3.5 Code snippet of 3 Layer 1D CNN

We can see the difference in the model features by printing it out.



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

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

Figure 2.3.6 1D 3 Layer CNN’s Features vs 2D 3 Layer CNN’s Features

2.3.4 Max Pooling layer

This layer performs the pooling operation which calculates the maximum value in each patch of each feature map and will always reduce the size of each feature map. We use max pooling and view operation to cast our input into a 2-dimensional input of size (sequence length, 2*hidden_dim), making it identical to the output from the BiLSTM layer.

2.3.5 Dropout layer

This layer acts as a mask and nullifies the contribution of some of the neurons towards the next layer. Dropout layers are important in training CNNs because they prevent overfitting on the training data. It makes the model more robust as it deactivates random neurons each epoch, so that the model learns a more generalized representation of the data. However, Dropout has the downside of reducing the rate of convergence.

2.4 CNN Implementation Results

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

2.4.1 Three Layer 1D CNN

```
Train: new_F: 0.9555073019116959 best_F: 0.9570724841660803
Dev: new_F: 0.8406671695582935 best_F: 0.8593229258024908
Test: new_F: 0.7741024962512127 best_F: 0.7812499999999999
33722.40657997131
```

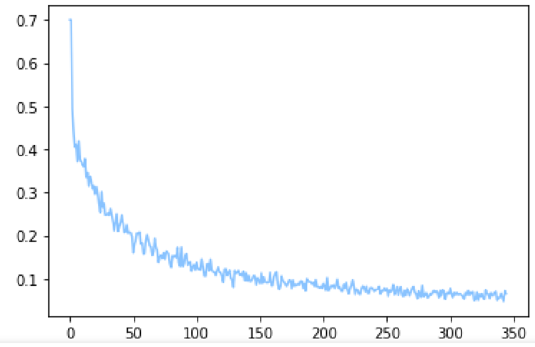


Figure 2.4.1 Graph of Loss for 3 Layer 1D CNN

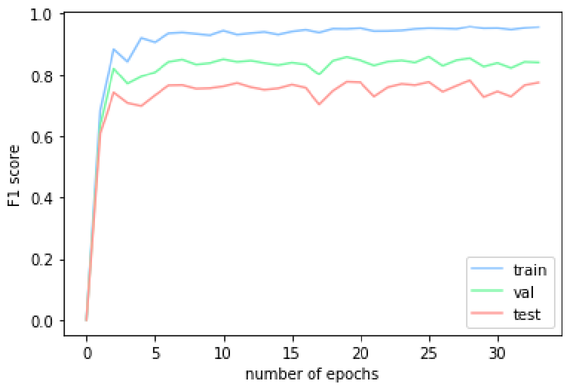


Figure 2.4.2 Comparison of f1-score for train, test and val

As seen from the graph, the best f1-score obtained for the validation set Dev is **0.859.**

2.4.2 Three Layer 2D CNN

```
Train: new_F: 0.9514044643810744 best_F: 0.9574440894568689
Dev: new_F: 0.857619382022472 best_F: 0.864949803579223
Test: new_F: 0.7790389708664397 best_F: 0.7863389487250299
33686.26013994217
```

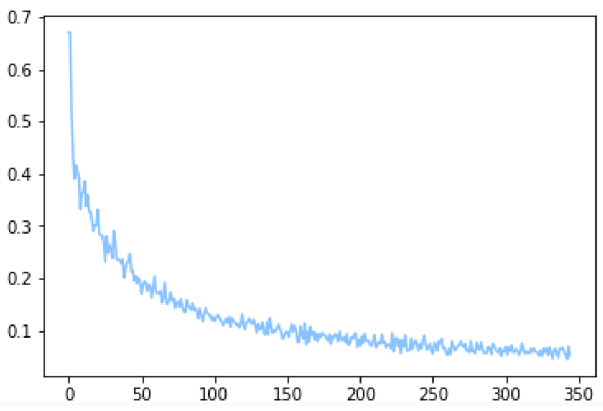


Figure 2.4.3 Graph of f1-score for 2D CNN

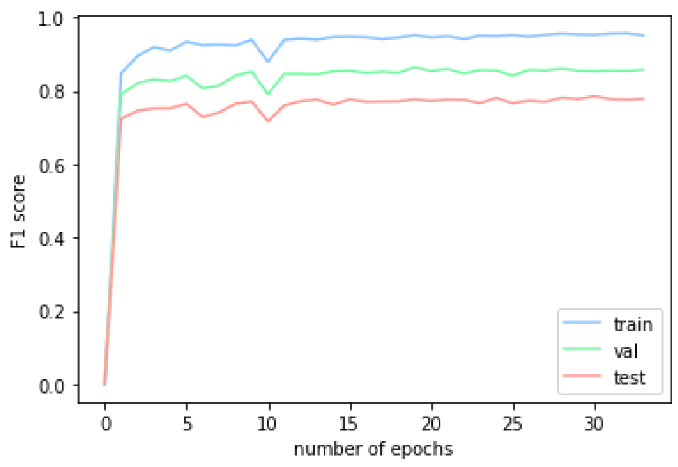


Figure 2.4.4 Comparison of f1-score for train, test and val

As seen from the graph, the best f1-score obtained for the validation set Dev is 0.864.

2.5 Test Set Results

Model	Score for Test Data Set
LSTM Layer (codebase model)	0.842
One Layer CNN	0.788
Three Layer 1D CNN	0.781
Three Layer 2D CNN	0.786

References

[1] Jay Alammar. The Illustrated Transformer. GitHub, Jun. 27, 2018. Accessed: Oct. 31, 2021. [Online]. Available: https://jalammar.github.io/illustrated-transformer/
[2] Meraldo Antonio. Word Embedding, Character Embedding and Contextual Embedding in BiDAF — an Illustrated Guide. Towards Data Science, Aug. 29 2019. Accessed: Nov. 1, 2021. [Online]. Available: https://towardsdatascience.com/the-definitive-guide-to-bidaf-part-2-word-embedding-character-embedding-and-contextual-c151fc4f05bb
[3] Niklas Donges. A Guide to RNN: Understanding Recurrent Neural Networks and LSTM Networks. Built In, Jul. 29, 2021. Accessed: Oct. 31, 2021. [Online]. Available: https://builtin.com/data-science/recurrent-neural-networks-and-lstm
[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. J. Mach. Learn. Res., 3:1137–1155, Mar. 2003.
[5] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional LSTM-CNNsCRF. In Proceedings of the 54th ACL, pages 1064–1074, Berlin, Germany, Aug. 2016. ACL.

[6] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF Models for Sequence Tagging. arXiv:1508.01991v1 [cs.CL], Aug. 9 2015. Accessed: Nov. 1, 2021. [Online]. Available: 1508.01991.pdf (arxiv.org)