






Attn: Shafiq Joty (Asst. Prof)



CE/CZ4045 Natural Language Processing

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

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

Name	Signature / Date
Pow Liang Hong	 5 November 2021
Tan Min, Tricia	 5 November 2021
Phang Yan Feng Benito	 5 November 2021
Lim Ming Aun, Ashton	 5 November 2021
Tan Wen Jie	 5 November 2021

Review Data Analysis and Processing

CZ4045 Assignment (Part 2)

Group 35

Liang Hong Pow
U1922786H
Nanyang Technological University
c190164@e.ntu.edu.sg

Ashton Lim
U1922375J
Nanyang Technological University
alim061@e.ntu.edu.sg

Wen Jie Tan
U1922085E
Nanyang Technological University
WTAN156@e.ntu.edu.sg

Phang Yan Feng Benito
U1922513L
Nanyang Technological University
c190139@e.ntu.edu.sg

Tan Min, Tricia
U1922619J
Nanyang Technological University
TTAN073@e.ntu.edu.sg

1. WikiText Language Model

1.1. Introduction

For this question, we will develop a neural network trained over a large series of strings to predict words after a given sequence of word(s). For the initial code, the codebase “Word-level language modeling RNN” by mattip1 will be used whilst the dataset will be the WikiText-2 Dataset2.

1.2. Dataset

The WikiText-2 dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia.

The dataset is splits into three different files, namely: “train”, “test” and “valid” for their individual purposes.

1.3. Data Loading and Preprocessing

The data is stored in wikitext-2 folder. Data is loaded and preprocessed when `data.corpus(args.data)` is called. The relevant methods can be found under `data.py`. Dictionary methods adds all unique words from our data before we tokenize the dataset individually.

In this case, no data normalization is done. Data normalization should be done for better results. Some examples of data normalization for text data would be removing stopwords and punctuations and “<unk>” and lemmatization. This would make the model learn better as words are better generalized and individual typing styles less prevalent, thus, allows the model to

be unaffected by miscellaneous characters or phrases like ‘@-@’ and ‘<unk>’ in the dataset.

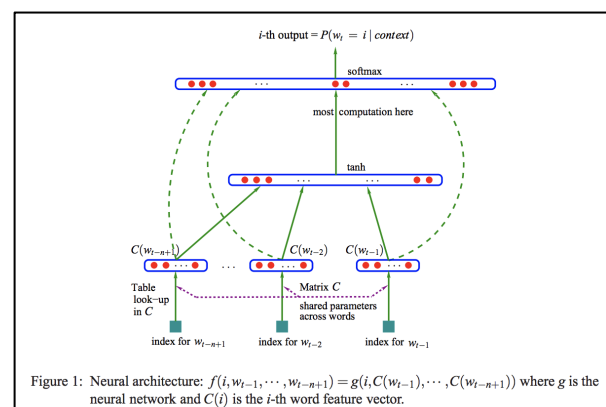
1.4. Model Creation

The codebase referenced from mattip provides a recurrent neural network (RNN) model. As required by the assignment, we will implement a feed-forward network (FNN) model.

The various terminologies below are used in the implementation of the network.

- `ntoken` – corpus dictionary size
- `em_size` – size of word embeddings
- `bptt` – sequence size (n for n -gram)
- `nhid` – number of neurons in hidden layer

The FNN model’s architecture will be modeled over the model developed by Bengio, Ducharme, Vincent, and Janvin in 2003 as shown below³:



¹ https://github.com/pytorch/examples/tree/master/word_language_model

² <https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/>

³ <https://dl.acm.org/doi/pdf/10.5555/944919.944966>

With reference to both Figure 1 and the mentioned research paper, we can observe that the following occurs in the neural network:

1. Column vectors are first concatenated to give the word features layer activation vector. This vector is known as X which will then be passed to the hidden layer with tanh activation.
2. In the hidden layer, the number of neurons can be specified in argument $nhid$.
3. The outputs of the hidden layer, is then passed to another hidden layer with no bias vector, producing $U * \tanh(d + Hx)$
4. X is also passed to another hidden layer producing output of $(b + Wx)$.
5. Both outputs are added together, producing the unnormalized log-probabilities for each output i , computed as follows:

$$y_i = b + Wx + U * \tanh(d + Hx)$$

Where

- W: word features to output weights
- U: hidden-to-output weights
- H: word-features to hidden weights
- b: bias vector for W
- d: bias vector for H

To create the model, we will add in a new class called "FNNModel" in `model.py` as shown below.

```
class FNNModel(nn.Module):
    def __init__(self,
ntoken,ninp,ngam,bptt,batch_size,nhid,tie_weights,dropout=0.2):
        super(FNNModel,self).__init__()
        # model Y = b + Wx + Utanh(d + Hx)
        self.ntoken = ntoken
        self.bsz = batch_size
        self.twolayer = False
        self.model_type = 'FNN'
        self.drop = nn.Dropout(dropout)
        self.flatten = nn.Flatten(start_dim=1)
        self.encoder = nn.Embedding(ntoken,
ninp)
        # W, b
        self.hiddenWB =
nn.Linear(in_features=ninp*bptt,out_features=ntoken)
        # H, d
        self.hiddenHD =
nn.Linear(in_features=ninp*bptt,out_features=nhid)
        # tanh(d + Hx)
        self.tanh = nn.Tanh()
        # U
        self.decoder =
nn.Linear(in_features=nhid,
```

```
out_features=ntoken,bias=False)
        self.init_weights()
        print("Dropout rate:", dropout)
        if tie_weights:
            if nhid != ninp:
                raise ValueError('When using
the tied flag, nhid must be equal to emsize')
            self.decoder.weight =
self.encoder.weight
        return

    def init_weights(self):
        initrangle = 0.1
        nn.init.uniform_(self.encoder.weight, -
initrangle, initrangle)
        nn.init.uniform_(self.hiddenWB.weight,
-initrangle, initrangle)
        nn.init.uniform_(self.hiddenHD.weight,
-initrangle, initrangle)
        nn.init.uniform_(self.decoder.weight, -
initrangle, initrangle)

    def forward(self, input):
        input = torch.transpose(input,1,0)
        embedding = self.encoder(input)
        # print(embedding.shape)
        embedding = self.flatten(embedding)
        # print(embedding.shape)
        # d + Hx, tanh(d+Hx)
        tanh =
self.tanh(self.drop(self.hiddenHD(embedding)))
        # U*tanh(d+Hx)
        U = self.decoder(tanh)
        # b + Wx
        WB =
self.drop(self.hiddenWB(embedding))
        # b + Wx + U*tanh(d+Hx)
        decoded = U + WB
        # decoded =
torch.transpose(decoded,1,0)
        decodedRS = decoded.reshape(self.bsz,
self.ntoken)
        # decoded = decoded.view(-1,
self.ntoken)
        return F.log_softmax(decodedRS, dim=1)
```

1.5. Training of Model

Training of model will be performed in batches with an optimizer used against it.

In this case, we'll be optimizing using Stochastic Gradient Descent (SGD), which is a variant of gradient descent, except with random sample. This is easily implemented as shown below:

```
optimizer =
torch.optim.SGD(model.parameters(),lr=0.01,
momentum=0.9)
```

Training of the model will be performed according to the assignment guidelines, with $n = 8$. This is done with the command below.

```
python main.py -cuda -model FNNModel -bptt 8
```

The results of the model with the default arguments are as follows below.

- Batch Size: 250
- Learning Rate: 20
- Hidden Layer Size: 100
- Embedding Size: 100
- Optimiser: SGD
- Number of Layer(s): 1 $C(wt - n + 1), C(wt - 2), C(wt - 1)$
- Dropout: 0.2
- Tied Weights: False

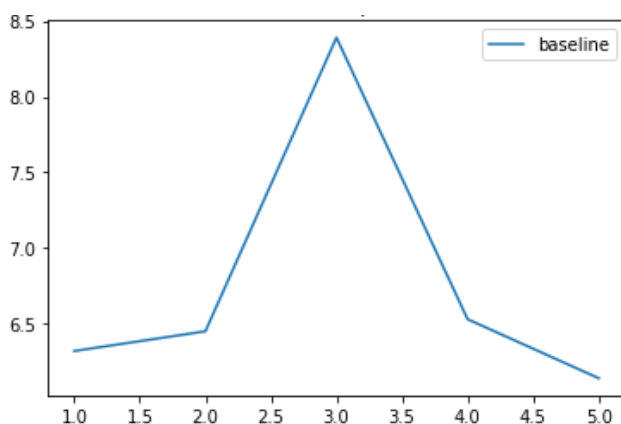


Figure 1.5: Loss vs Epoch (Baseline Model)

Minimum Loss	6.14
Minimum Perplexity	462.72

We theorise that if we were to tune some of the parameters, we may be able to obtain a more effective model.

In the later sections, we'll be experimenting with tuning different parameters, and observing how it affects the model.

The parameters we'll be looking into tuning are as follows:

- Batch Size
- Learning Rate
- Hidden Layer Size
- Embedding Size
- Optimiser
- Number of Layer(s)
- Dropout
- Tied Weights

1.6. Optimising Parameters

1.6.1. Batch Size

In theory, the smaller the batch size, the closer it becomes to being stochastic gradient descent. The benefit is that we are learning and adjusting weights for every data which in theory should be better learning for our model. However, it comes at the cost where lower batch sizes will take longer time for model to learn due to more weight calculations. Hence, we tend to use mini-batch gradient descent. To determine the optimal batch size for the most efficient model, one way would be to conduct testing with different size values.

In order to determine the optimal batch size, we can start off with looking into the `batchify` and `get_batch` function provided in the initial codebase.

As explained in the codebase, the `batchify` function converts the data into cleanly divided specified batch sized. Remaining elements that wouldn't fit cleanly will be trimmed off. After being divided, data is then evenly divided into matrix form. The codebase function can be seen below.

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

Meanwhile, we can observe from the codebase that the `get_batch` function takes in the matrix results from `batchify` method above and divides it into chunks of desired length `args.bptt`. However, this function is not applicable for our FNN. Hence, an additional function `get_batch_FNN` has been implemented in place of this. The function can be seen below.

```
def get_batch_FNN(source, i):
    seq_len = min(args.bptt, len(source) - 1 - i)
    data = source[i:i+seq_len]
    target = source[i+seq_len].view(-1)
    return data, target
```

The main difference for the `get_batch` methods is the way the target is selected. For our FNN, the target should be `bptt` length away from the initial word. Hence the need for a separate `get_batch_FNN` method. Similarly to the `get_batch` function provided in the codebase, `get_batch_FNN` also takes in the sequence results of `batchify` function but generates subsets of size `bptt` data along with the corresponding target.

We decided to select batch size experimentally by testing 3 different batch sizes of 100, 250 and 400. The results are as follows:

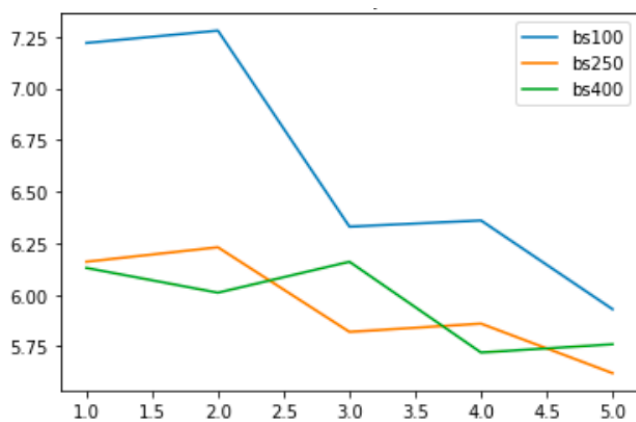


Figure 1.6.1: Loss vs Epoch (100 / 250 / 400)

Batch Size	Minimum Loss	Lowest Validation Perplexity
100	5.93	375.75
250	5.62	276.6
400	5.72	305.22

From the data, we can observe that batch size of 250 gives us the lowest complexity. The higher batch size also makes it so that computation time is lower as previously explained. Hence, we selected batch size of 250 for our best model.

1.6.2. Learning Rate

We decided to use the variable annealed learning rate that was implemented in the original code base. The default learning rate from the codebase is 20. This learning rate will however be annealed by a factor of 4 if there is no improvement in the validation set as mentioned in the original code base.

1.6.3. Hidden Layer Size (Number of neurons per layer)

We experimented with hidden layer sizes of 100, 250, 400 and the results are as follows:

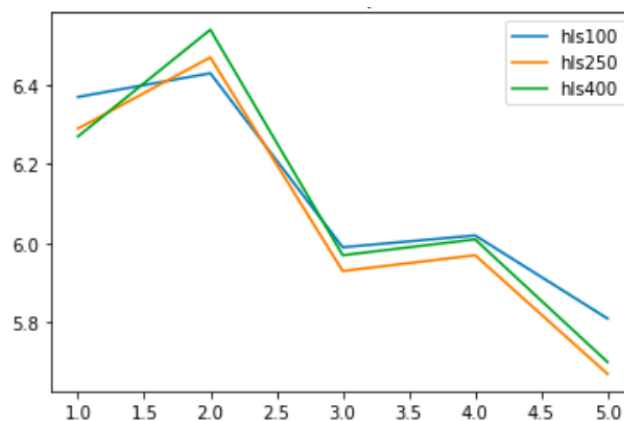


Figure 1.6.3: Loss vs Epoch (100 / 250 / 400)

Hidden Layer Size	Lowest Validation Perplexity
100	334.24
250	290.65
400	299.27

From the results, we can conclude that increasing the hidden layer size does not guarantee an improvement in our lowest validation perplexity.⁴ On the other hand, using too many neurons may result in several problems such as overfitting where the network becomes a memory bank that recalls the training set to perfection instead of learning the general trend. It will also increase our learning time due to more weights needing to be calculated and recalculated every batch.

Therefore, in order to secure the ability of the network to generalize, we have selected hidden layer size of 250 after taking these into account.

⁴<https://web.archive.org/web/20140721050413/http://www.heatonresearch.com/node/707>

1.6.4. Embedding Size

Due to our usage of tied weights, our embedding size and hidden layer size must be equal as well. Hence, we opted to skip experimentation on embedding size and instead use the same `em_size` as our `nhid` as selected above.

However, should we choose to not tie the weights and be able to freely select embedding size, we could consider higher embedding size as higher embedding size allows for higher dimension representation of our tokens and could potentially better represent the relationship between our tokens. However, this also comes with a cost where higher dimensionality will require more space for storage. More weight calculations could however lead to longer learning time. Hence, a need to balance between computational time and complexity.

An alternative would be to implement other more advanced embedding methods such as Word2Vec or GloVe to better represent words as vectors.

1.6.5. Optimiser

One other factor that may affect the quality of a model is the optimiser that's being used. We observed that different optimisers may bring about different validation perplexity scores. With all other parameters remaining constant we have ran the model against multiple optimizers, namely: SGD, Adam, and Adadekta.

As observed from the diagram below, we can observe that throughout the model training cycle, there'll always be just one point where there'll be a spike in loss, this is possibly due to the change in learning rate.

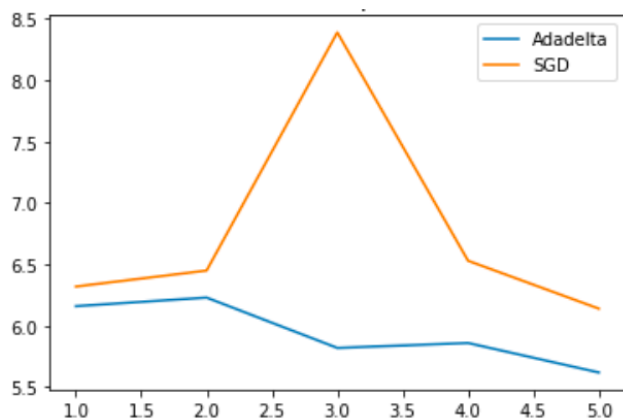


Figure 1.6.5A: Loss VS Epoch (Adadelata + SGD)

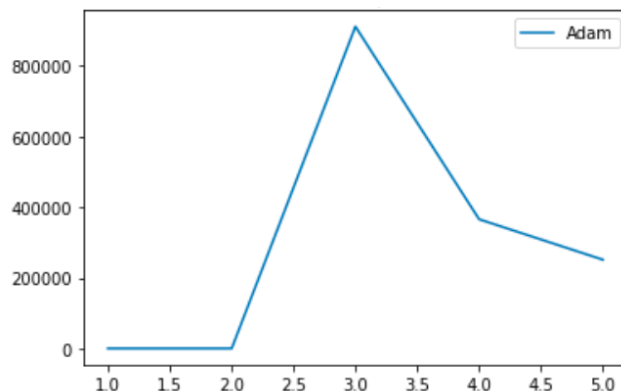


Figure 1.6.5B: Loss VS Epoch (Adam)

Optimiser	Lowest Validation Perplexity
SGD	462.72
Adam	468.13
Adadelata	276.60

However, based on the data above, we can conclude that the Adadelata optimiser is the one that best suits this model. This could be because Adadelata is based on a fixed moving window of gradient updates, instead of accumulating all past gradients, hence why it might not be as affected by the spike in loss that other optimisers face. As such, Adadelata will be the optimiser we will use for the final model.

1.6.6. Number of Layers

Most online articles states that one neural layer is sufficient for most problems⁵. Hence, we also chose to only use 1 layer to reduce complexity of the model. Reduced complexity will not only allow us to save time when running the model, it will also prevent the model from memorising our dataset and overfit, causing our model to not perform well in testing and generalizing.

⁵<https://web.archive.org/web/20140721050413/http://www.heatonresearch.com/node/707>

1.6.7. Dropout

Legend	Dropout
—	Dropout 0
—	Dropout 0.2
—	Dropout 0.4
—	Dropout 0.6

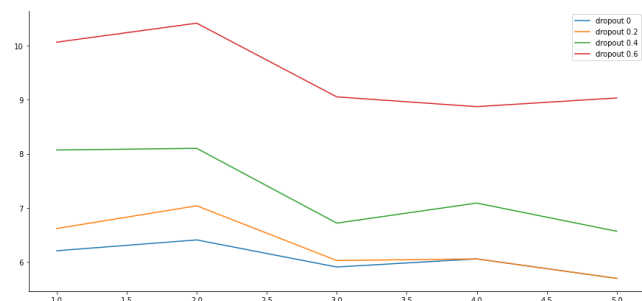


Figure 1.6.7: Loss VS Epoch (Dropout 0 / 0.2 / 0.4 / 0.6)

Dropout	Lowest Validation Perplexity
0.0	298.44
0.2	298.23
0.4	712.84
0.6	7135.91

From the results, we can see that dropout rate of 0 and 0.2 perform similarly, both achieving lowest validation perplexity of 298. However, it is known that introduction of dropout can help prevent the model from overfitting to the training data by randomly dropping out units. Hence, we chosen dropout rate of 0.2 for this benefit as well as its lowest validation perplexity.

1.6.8. Tied Weights (Sharing input and output layer embeddings)

We can toggle whether or not we are sharing input and output layer weights by passing `-tied 1` for share 0 for not sharing when running. However, when sharing input and output layer embeddings, we must make sure `em_size` and `nhid` are equal as shown in the code:

```
if tie_weights:
    if nhid != ninp:
        raise ValueError('When using the tied
flag, nhid must be equal to emsize')
    self.decoder.weight = self.encoder.weight
```

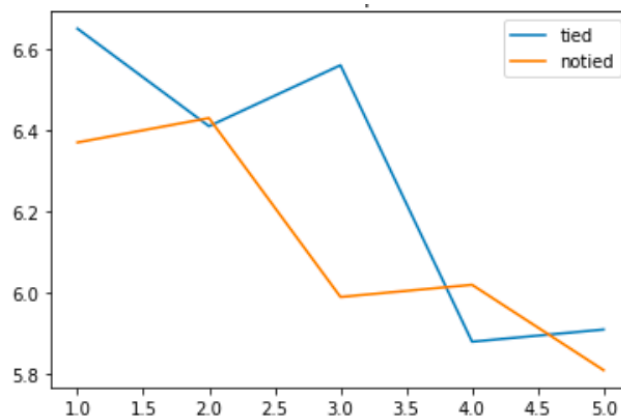


Figure 1.6.8: Loss VS Epoch (Tied / Not Tied)

Tied Weight	Lowest Validation Perplexity
Not Tied	334.24
Tied	357.61

From the above results, we can see that without tying weights, the minimum validation perplexity from no tied weights is slightly lower (10%) than with tied weights. However, by using tied weights, we reduce the number of parameters the model must learn. The weights learnt in the encoding layer will be used for the decoder instead. This will lead to faster learning by the model as we experienced in our experiment. Also, interestingly, from Figure 1.6.8, we can also see that tied weight actually performs better in certain epochs. Hence, with all these factors in mind, we opted to use tied weights due to faster training times. With the multiple models we are running, this would be the better choice and save us time.

1.7. Final Model

With the hyperparameters tuned, we deduced that the following parameters would be most suited for the FNN architecture

- Batch Size: 250
- Learning Rate: 20
- Hidden Layer Size: 250
- Embedding Size: 250
- Optimiser: Adadelta
- Number of Layer(s): 1
- Dropout Rate: 0.2
- Tied Weights: True

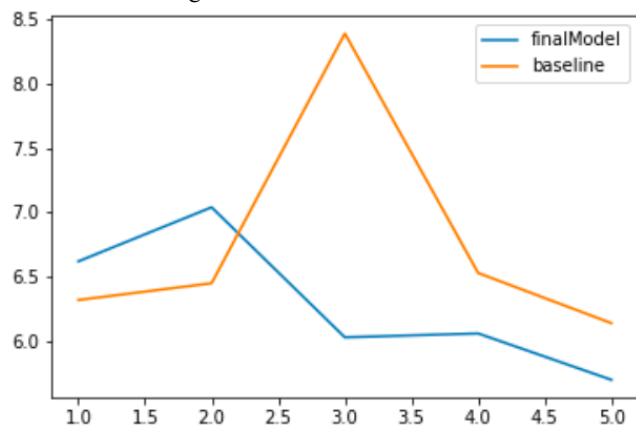


Figure 1.7: Loss vs Epoch (Final Model vs Baseline Model)

Model	Minimum Loss	Lowest Validation Perplexity
Baseline	6.14	462.72
Tuned Model	5.7	298.23

Running the model with the parameters above, we can conclude that tuning of parameters has brought a positive effect onto the model as we are able to obtain a much lower test perplexity score.

1.8. Text Generation

IGN executive producer Anthony summoned to their sense " call for the award Best Best Best Actor Colin Mark Sosa to take place . According literary tradition of 1633 , began with edges , Scully from September , such as the title , social perception of tissue or shorter than any person or semen for years , another line , the Nown , Daniel " , allowed to his neighbour , securely , a relationship is most watched average below . Turner 's final consecutive victories victories victories victories victories defeated in 2004 album . Goffman painting . While he was used as a person or drugs , she married . IGN stated that would be dedicated John had been built in 1937 , a memorial .

Above is a snippet from the text generated from our model. As we can see, it is not very coherent and does not make much sense overall. However it does make some sense in shorter viewings. For example, we can see in the last sentence makes sense in the context of a memorial built which is dedicated to John despite the grammatical error. The model also seems to have problems with proper punctuation usages. The model also often changes context immediately after it produces a comma which resulted in the overall incoherence of the text generated.

1.9. Conclusion

In conclusion, the model seems to only perform decently in the short term. This could be due to the nature of a FNN model which is largely local in nature. FNN has a fixed window of input, hence it might not be able to learn long term dependencies in sentences. In our case, our fixed window is 8 since we are using 8-gram hence dependencies longer than 8 are not learnt by the FNN.

In order to tackle this problem, we could explore using RNN, in particular LSTM, or CNN which may potentially yield shorter training time. Both are able to better learn long term dependencies.

Also as mentioned previously, we could consider different embedding methods such as Words2Vec and GloVe which might be able to help better capture global information.

Lastly, we can also consider normalization of our training data. Normalization of text data includes things such as removing of stop words. Such normalization could make our model invariant to different writing styles and characteristics. This is especially relevant to our dataset as Wikipedia articles are able to be freely edited by anyone and different writing styles and characteristics will be present.

2. Named Entity Recognition (NER)

2.1. Introduction

Named entity recognition (NER) is a sub-task of information extraction (IE) that seeks out and categorizes named entities in unstructured text into pre-defined categories such as people's names, organizations' names, locations, expressions of time, quantities, monetary values, percentages, and so on.

To learn what an entity is, relevant data must first be supplied into a named entity recognition model, which will then be labelled with entity categories.

This report will build upon the End-to-end Sequence Labelling provided which implements a Bi-directional LSTM-CNN-CRF **NER model** using PyTorch.

2.2. Dataset

The dataset used is the English data from CoNLL 2003 shared task. This dataset contains four different types of named entities for tagging: PERSON, LOCATION, ORGANIZATION, and MISC. The dataset also uses the BIO tagging scheme. This is split into 3 files - `eng.train`, `eng.testb` and `eng.testa` for training, testing and validation respectively.

2.3. Pre-processing and Loading

The pre-processing step has several phases.

The first phase would be changing all digits to 0 in the `zero_digits()` function. This is because digits are not helpful for predicting in NER task. By changing all digits to 0 would help the model concentrate on alphabets now, which are more important.

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

The second phase would be changing the tagging scheme to BIOES from BIO. We have also updated the data tag scheme from BIO to BIOES, which is different from the dataset's BIO as described below:

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

```
def update_tag_scheme(sentences, tag_scheme):
    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!')
```

The next phase would be creating mappings for all words, characters, and tags in each word to unique numeric IDs. This ensures that a particular integer ID represents each unique word, character, and tag in the vocabulary. These mappings allow for tensor operations in the NN architecture, which is 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
```

```
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
```

```
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
```

The function below returns a list of dictionaries, where each dictionary contains

- list of all words in the sentence
- list of word index for all words in the sentence
- list of lists, containing character id of each character for words in the sentence
- list of tags for each word in the sentence

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

```
in char_to_id]
            for w in str_words]
            tags = [tag_to_id[w[-1]] for w in s]
            data.append({
                'str_words': str_words,
                'words': words,
                'chars': chars,
                'tags': tags,
            })
    return data
```

Then, we prepare three datasets using the respective functions. In our case, we produce 14041 train sentences, 3250 dev sentences and 3453 test sentences using the following code:

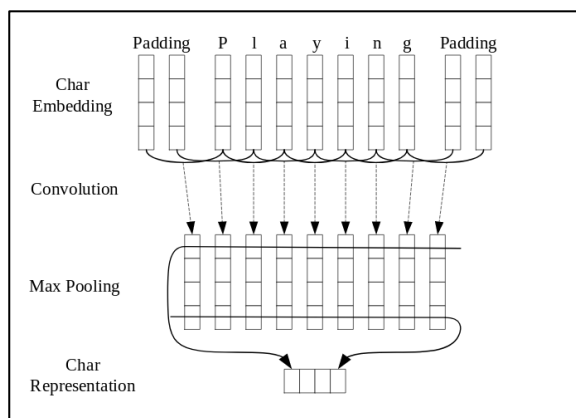
```
train_data = prepare_dataset(
    train_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']
)
dev_data = prepare_dataset(
    dev_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']
)
test_data = prepare_dataset(
    test_sentences, word_to_id, char_to_id, tag_to_id, parameters['lower']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

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

Next is to load the set of pretrained word embeddings. These pretrained word embeddings are the Glove vectors 100-dimension vectors, trained on the Wikipedia 2014 + Gigaword 5 corpus containing 6 billion Words.

2.4. Model Creation

2.4.1. Character Level Encoder



In the codebase given, a CNN model is used to generate character level embeddings. The function takes in characters and converts it into embeddings using the character CNN. These character embeddings are concatenated with glove vectors. These are then fed to a CNN layer, followed by the Max Pooling layer to create the word embeddings for the word level encoder. The code below shows the implementation.

```
chars_embeds =
self.char_embeds(chars2).unsqueeze(1)

## Creating Character level representation
using Convolutional Neural Netowrk
## followed by a Maxpooling Layer
chars_cnn_out3 = self.char_cnn3(chars_embeds)
chars_embeds =
nn.functional.max_pool2d(chars_cnn_out3,kernel_
size=(chars_cnn_out3.size(2),1)).view(chars_cnn
_out3.size(0), self.out_channels)

## Loading word embeddings
embeds = self.word_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)
```

2.4.2. Word Level Encoder

The word level encoder is implemented using a CNN model. This model has a few layers. Firstly, a convolution model similar to the character CNN. Next, an optional ReLU layer for non-linear

activation, and a Max Pooling layer. Lastly, there is a linear layer to map hidden vectors to the tag space. This can be seen below:

```
## Takes words as input and generates a output
at each step
cnn1_out = self.cnn1(embeds)
if self.relu:
    cnn1_out = self.ReLU_layer(cnn1_out)
cnn_max_pool =
nn.functional.max_pool2d(cnn1_out,kernel_size=(
1,cnn1_out.size(3))).view(cnn1_out.size(0),self
.hidden_dim*2)
cnn1_out = self.dropout(cnn_max_pool)

cnn_out = cnn1_out

##if multi layers CNN word level
if self.layers > 1:
    cnn1_out = cnn1_out.unsqueeze(1)
    cnn2_out = self.cnn2(cnn1_out)
    if self.relu:
        cnn2_out =
self.ReLU_layer(cnn2_out)
    cnn2_max_pool =
nn.functional.max_pool2d(cnn2_out,kernel_size=(
1,cnn2_out.size(3))).view(cnn2_out.size(0),self
.hidden_dim*2)
    cnn2_out = self.dropout(cnn2_max_pool)
    if self.layers > 2:
        cnn2_out = cnn2_out.unsqueeze(1)
        cnn2_out = cnn2_out.unsqueeze(1)
        cnn3_out = self.cnn3(cnn2_out)
        if self.relu:
            cnn3_out =
self.ReLU_layer(cnn3_out)
        cnn3_max_pool =
nn.functional.max_pool2d(cnn3_out,kernel_size=(
1,cnn3_out.size(3))).view(cnn3_out.size(0),self
.hidden_dim*2)
        cnn3_out =
self.dropout(cnn3_max_pool)
```

2.4.3. ReLU Theory

Rectified Linear Activation Function (ReLU) is a non-linear activation function often used to use stochastic gradient descent with backpropagation of errors to train deep neural networks.

ReLU is a linear function that has values greater than 0 and non-linear as negative values are always output as 0.

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

2.4.4. Training

```
learning_rate = 0.015
momentum = 0.9
number_of_epochs = parameters['epoch']
decay_rate = 0.05
gradient_clip = parameters['gradient_clip']
optimizer =
torch.optim.SGD(model.parameters(),
lr=learning_rate, momentum=momentum)
```

The training parameters done are as seen above, using stochastic gradient descent (SGD), with a learning rate of 0.015, momentum = 0.9, and decay_rate at 0.05. These parameters are as recommended by journal article⁶. attached with the codebase.

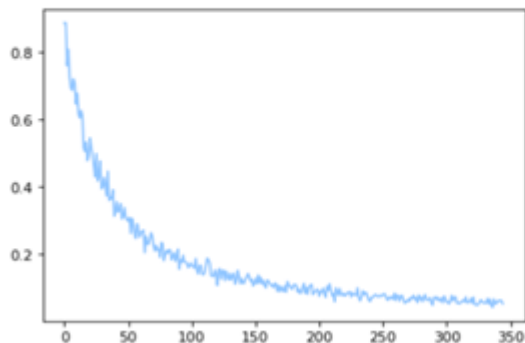
2.5. Testing Results

2.5.1. Selection of Model

The saved model is based on the validation set as seen in the code below.

```
best_dev_F, new_dev_F, save = evaluating(model,
dev_data, best_dev_F, "Dev")
if save:
    print("Saving Model to ", model_name)
    torch.save(model.state_dict(), model_name)
```

2.5.2. Results of 1-layer CNN



	new_F	best_F
Train	0.973578145991939	0.9733489811159715
Dev	0.8548358785325609	0.8705717292178183
Test	0.7499763324813027	0.7890677078373488
Time Taken	11898.168468952179	

Testing Model	new_F
Initial CNN-1-layer	0.7781969068276122

Here we can see the results of the trained model, along with the score of the best fit model. For a 1-layer CNN model, the testing time took 11898 seconds.

Testing the model chosen with the previously mentioned method gives us an accuracy of about 0.778.

2.5.3. Results for all layers

Here we increased the number of layers for our model, as this would increase the number of features our model is able to receive from the train data, and thus increasing accuracy. Too many layers, however, would cause overfitting of the training models, and is not desirable as overfitting would cause the model to 'memorize' the data instead of learning from it, which in turn would cause higher training accuracy, but lower test accuracy.

We trained the model with 1-3 layers, both with and without ReLU layers. For non-ReLU layers, when increasing the number of layers from 1 to 2, we can see that the total time taken increases from 11898s to 15001s, which is around a 26% increase in time taken. The test accuracy also increases from about 0.778 to 0.788.

Next, we increased the number of layers from 2 to 3 to see if there would be any improvement. We see that this also increases the total time taken from 15001 to 16714, about 11% increase in time taken. The test accuracy here increases slightly from 0.788 to 0.789.

CNN Layer (s)	new_test_F score	Time Taken (s)
1-layer CNN	0.7781969068276122	11898
2-layer CNN	0.7887996941896024	15001
3-layer CNN	0.789598540149854	16714

This means that the best non-ReLU model to select would be a 2-layer CNN model, as it provides a substantial increase in accuracy compared to the 1-layer CNN, and the 3-layer CNN barely increases accuracy while having an increase in the time taken.

⁶ https://github.com/jayavardhanr/End-to-end-Sequence-Labeling-via-Bi-directional-LSTM-CNNs-CRF-Tutorial/blob/master/Named_Entity_Recognition-LSTM-CNN-CRF-Tutorial.ipynb

When training the models with ReLU layers, we can also see a similar trend in the amount of time required to train the model. Increasing the number of layers increases the time taken. Going from a 1 to 2-layer ReLU model, the test accuracy increases from about 0.7155 to 0.7644. However, when going from 2 to 3 layers, the accuracy drops to 0.75. This could be due to overfitting of the training models, which means that adding more layers would not be useful.

CNN Layer (s)	new_test_F score	Time Taken (s)
1-layer CNN with ReLU	0.715512708150745	12161
2-layer CNN with ReLU	0.7644257189961532	14450
3-layer CNN with ReLU	0.7500000000000001	17041

Therefore, this means the best ReLU model to select would also be the 2-Layer model, as it performs better than the other models in terms of accuracy.

2.6. Conclusion

When comparing the models with and without ReLU, we see that the non-ReLU models have a higher accuracy than the ReLU models. This holds true for all the models of different number of layers. We can thus conclude that the best model to choose overall would then be the 2-Layer non-ReLU Model.

3. Contribution

Member's Name	Contribution
Pow Liang Hong	Question 1
Tan Min, Tricia	Question 2
Phang Yan Feng Benito	Question 1
Lim Ming Aun, Ashton	Question 1
Tan Wen Jie	Question 2