Assignment Report

CZ4045 Assignment 2 Report

Author: KOH YIANG DHEE MITCHELL, RYAN LEE RUIXIANG, LI PINGRUI, TANG YUTING, WANG BINLI

Instructor: Prof. Joty Shafiq

Academic year: 2021-2022

1. Introduction

This assignment is to develop and train multiple neural models in Pytorch to perform two nlp tasks, next word prediction and Named Entity Recognition (NER). In the first task, FNNModel is implemented and the 8-gram language model is trained with several variants. The perplexity score is used to evaluate the trained model. In the second task, the CNN-LSTM-CRF model is implemented, LSTM-based word-level encoder is replaced with different numbers of CNN layers. The F1 validation score is used to evaluate the trained model.

2. FNN Language Model

2.1. Language Model

Language model is a probability distribution over sequences of words. Given such a sequence of length m, language model assigns a probability $P(w_1, \dots, w_m)$ to the whole sequence. Non-neural language model could be implemented with n-grams, which collects statistics about frequency of different n-grams to predict the next word. However, there is sparsity problem when n is large and the model size increases greatly. Hence, neural network is a preferred method because it can support longer context and use a distributed representation instead of discrete representation.

2.2. Model Description

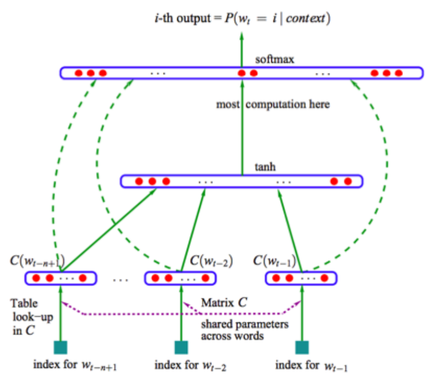


Figure 1: FNN model

FNN model [1] is a language model with feed-forward network architecture. The model has the following components:

- **Distributed Feature Vectors** C, which projects a word v in the vocabulary V to a real vector. It is implemented via one torch.nn.Embedding class, because it is shared across all the words in the contexts. It converts the sequence of words to a sequence of dense features.
- **Function** g that maps the features to conditional probability. In this assignment, a feed-forward network is used. To be specific, a hidden layer after the embedding with biases and the activation function of tanh. Optional direct connections from the

word features to the output layer (as indicated the dash liens) are included in this assignment.

The model can be summarized as

$$y = b + Wx + U\tanh(d + Hx) \qquad (1)$$

, where W is the weight that is set to 0 when no direct connection between input features and final layer, x is the concatenation of word features, b is the bias of the final layer, and U is the weights for hidden-to-output.

The output of the model is a vector with the size of $|V|$ and the sum of entries as 1, in which i-th element represents the conditional probability of i-th word occurrence. Softmax function is appended at the output.

The components of the model are defined as below:

```python
class FNNModel(nn.Module):
    def __init__(self, vocab_size: int, embed_dim:
    ↪ int, hidden_dim: int, n_gram: int, device,
    ↪ is_share_param: bool):
        ...
        super(FNNModel, self).__init__()

        self.context_size = n_gram - 1
        self.embedding_dim = embed_dim
        self.is_share_param = is_share_param

        self.word_embedding =
        ↪ torch.nn.Embedding(vocab_size, embed_dim)
        # hidden layer
        self.hidden_layer =
        ↪ torch.nn.Linear(self.context_size *
        ↪ embed_dim, hidden_dim, bias=True)
        self.active_func = torch.nn.Tanh()
        if not is_share_param:
            self.hidden2output =
            ↪ torch.nn.Linear(hidden_dim,
            ↪ vocab_size, bias=False)
        # direct connection between input features
        ↪ and output layer
        self.direct_connect_layer =
        ↪ torch.nn.Linear(self.context_size *
        ↪ embed_dim, vocab_size, bias=False)
        # output bias, a free parameter
        self.output_bias = torch.zeros(1, vocab_size,
        ↪ requires_grad=True, device=device)
        # softmax layer on top
        self.softmax = torch.nn.LogSoftmax(dim=-1)  #
        ↪ normalize along the row of tensors

        self.init_weights()
```

The computation flow is specified as the following:

```python
def forward(self, input, *args):
    word_features = self.word_embedding(input.t())
    concat_features = word_features.view(-1,
    ↪ self.context_size * self.embedding_dim)
    hidden_values = self.active_func(self.hidd⌋
    ↪ en_layer(concat_features))
    if self.is_share_param:
        matrix = self.word_embedding.weight
        decoder_output = hidden_values @ matrix.t()
    else:
        decoder_output =
        ↪ self.hidden2output(hidden_values)
    direct_connections =
    ↪ self.direct_connect_layer(concat_features)
    y = self.output_bias + decoder_output +
    ↪ direct_connections
    y_normalized = self.softmax(y)
    return y_normalized
```

2.3. Dataset

2.3.1 Data Overview

The dataset used in this question is Wikitext-2 dataset composed of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia. The vocabulary includes all words appearing more than 2 times and other words out of the vocabulary are mapped to the <unk> token. Compared to Penn Tree Bank, more language features like case of letters and punctuations are retained and the vocabulary size is larger, which makes it more realistic for most language use and able to produce better learning outcomes on longer term dependencies.

2.3.2 Data Loading

The data in Wikitest are loaded and stored in objects of class Dictionary and Corpus. Dictionary contains the vocabulary indexed by numeric values in the corpus. The class Corpus reads the raw text, constructs the dictionary and converts the sentences to a list of word indexes.

2.3.3 Data Preprocessing

After reading the dataset, the sentences are tokenizing by splitting, and a look-up table of the word index and word is constructed.

2.3.4 Data Streaming

The model is trained with mini-batches. Given n-gram=8, a batch contains multiple word sequence with the length of 7 as the training data input and the corresponding next words (i.e. the last words in the 8-gram sequence) as the label to learn. Different from the streaming of RNN provided in [3], the way of get the batch is changed.

```python
def get_batch(source, i):
    if args.model != "FNN":
        seq_len = min(args.bptt, len(source) - 1 - i)
```

```
            data = source[i:i + seq_len]
            target = source[i + 1:i + 1 +
            ↪  seq_len].view(-1)
        else:
            if i + args.n_gram > len(source):
                return None, None
            n_gram_data = source[i: i + args.n_gram]
            data = n_gram_data[:-1, :]
            target = n_gram_data[-1, :]
        return data, target
```

As shown above, when the elements in the same column are consecutive, n-gram number of rows are selected as the new batch. Then the batch is split to context word sequence and the target word.

2.4. Experiment

2.4.1 Overview

All the parameters in the model are initialized with the uniform distribution ranging from $[-1,1]$. Multiple epochs of training are conducted on the train set, and after each epoch, the evaluation on the valid set is conducted to choose the best model. The model is evaluated on the test data after the training completes.

2.4.2 Train with 8-gram

The model is trained with 8-gram on Wiki Text. The results are shown as Figure ??, where perplexity and loss are evaluated in the valid datasetm, and Table 1. SGD optimizer has the initial learning rate as 1.0, and it decreases the learning rate by 75% when there is no performance gain. Adam optimer has the initial learning rate as 0.0001, and RMSProp has the initial learning as 0.1. Other parameters are default in PyTorch.

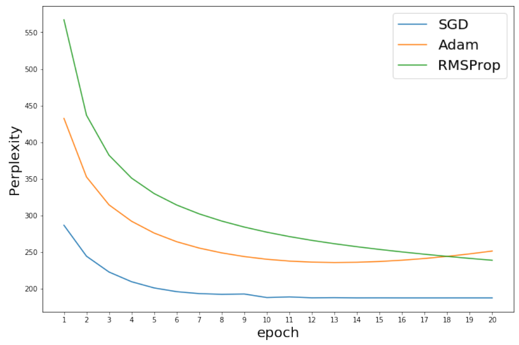


Figure 2: training result with and without sharing embeddings

	1	5	15	test
SGD	286.69	201.19	187.69	175.32
Adam	432.54	276.08	237.27	220.12
RMSProp	567.06	329.96	253.71	225.10

Table 1: Perplexity on valid data after n epochs and on the test dataset

2.4.3 Weight sharing

U in Eq.1 is the matrix with the shape of (Hidden Dimention, $|V|$), while C is the embedding matrix of V with the shape of (Embedding Dimension, $|V|$). The model can share the same value between U and C, when the hidden dimension equals to the embedding dimension. The result is as shown in Figure 3 and Table 2. They show there is a little improvement on the performance.

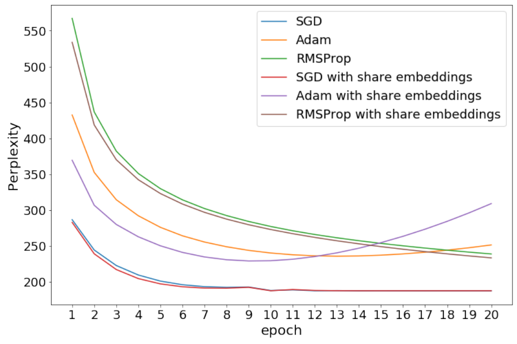


Figure 3: training result with and without sharing embeddings

	1	5	15	test
SGD	282.95	197.30	187.76	174.95
Adam	369.50	250.34	254.70	219.56
RMSProp	533.73	323.12	249.17	233.52

Table 2: Perplexity on valid data after n epochs and on the test dataset with weight sharing

2.5. Text Generation

Language Model can be used for text generation. The initial words (i.e. first 7 words) are randomly generated and fed to the trained model to predict the next word. After obtaining the next-word probability vector v, v defines a multinomial distribution, and one word is sampled from such a distribution. Temperature coefficient [2]

is applied on the logit output of the model before Softmax layer to achieve the temperature sampling. Lower temperature t makes the model more confident in top choices, but $t > 1$ decreases the confidence.
Below is some text generated from the trained model.

```
81 tourist was work by Service Fantasy , such the shared why , 'll J. Lowe and account . MD 's been also London by you Park . In Canada , the Moorside was the film dwindled on bringing gained to mostly drain .
```

The sentences and paragraphs in the generated text are mostly incoherent, but there are a number of coherent and grammatically correct phrases. It is observed that quite a number of relevant words appear close to each other. For example, the word 'pupil' appears after the word 'school' and 'Canada' appears after 'United States'.

3. Named Entity Recognition

3.1. Model Description

3.1.1 CNN-LSTM-CRF

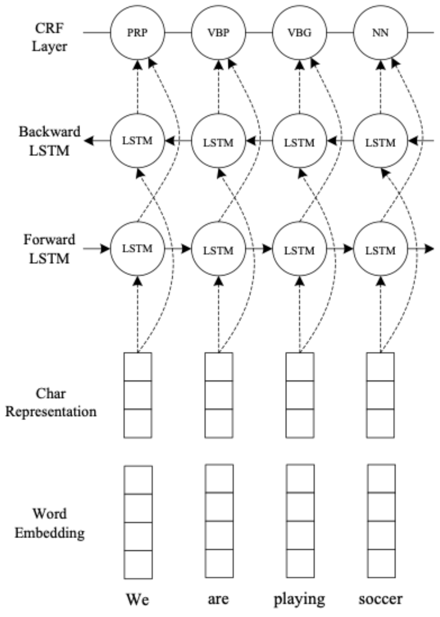


Figure 4: The CNN-LSTM-CRF Model for NER

The NER model consists of the following parts:

- **Character-level encoder with CNN:** a convolution layer is applied to character, then followed by max pooling to extract important features, which is a dense vector representation of each word. The feature vector is then concatenated with GloVe vectors from the pre-trained word embeddings to represent a word.

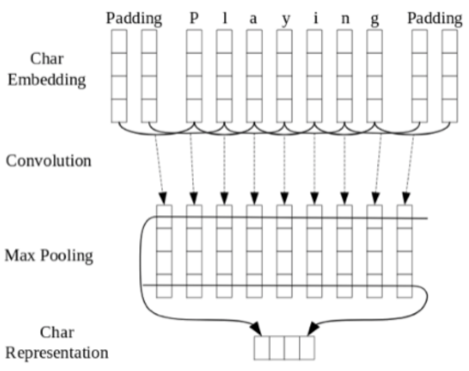


Figure 5: Character-level encoder with CNN

- **LSTM recurrent neural network:** the concatenated word embeddings are then fed to a LSTM network, which consists of a forward layer and a backward layer. The forward layer generates a new summary vector based on previous words in the forward direction. The backward layer generates another summary vector based on future words in the backward direction.
- **CRF output layer:** the output layer is linear-chain CRF. Compared to Softmax function, CRF captures information from the context that is produced by the LSTM network.

3.1.2 CNN-CNN-CRF

The model to experiment in this assignment is the model replacing the word-level encoder as LSTM by CNN layers. Word-embeddings are still the one generated from the concatenation of glove and character. For this we used a 2-dimensional convolution kernel. The following changes must be made in the get_lstm_features function and the BiLSTM_CRF function:

```python
# Word CNN
embeds = embeds.unsqueeze(1)
cnn_out = self.conv(embeds)
cnn_out = cnn_out.view(len(sentence),
↪   self.hidden_dim*2)
cnn_feats = self.hidden2tag(cnn_out)
```

We used a 2-dimensional convolution kernel. The 1-dimensional text in the dataset is changed into 2-dimensional text through character embedding producing a vector character representation. The convolution is done in only one dimension, the time dimension but with a 2-dimensional kernel. The second dimension is the embedding dimension of each word.

```
if self.char_mode == 'CNN':
    #remove lstm layer add cnn layers
    self.conv = nn.Conv2d(in_channels=1,
    ↪   out_channels=hidden_dim*2, kernel_size=(1,
    ↪   self.out_channels+embedding_dim))
```

3.2. Experiment

3.2.1 Dataset: CoNLL

The dataset is from the standard CoNLL NER dataset and is then split into three sets. The Train set contains 14041 sentences, the Validation set which contains 3250 sentences and the Test set which contains 3453 sentences.

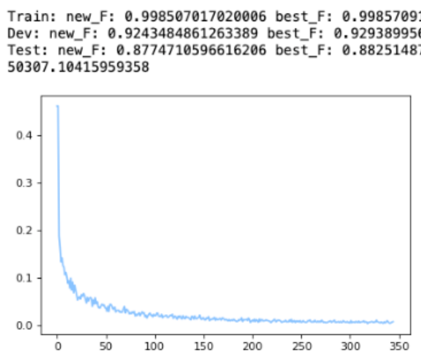
3.2.2 Data Prepossessing

The data prepossessing includes the following steps.

1. **Removing all numerical data.** This is because numerical digits do not help in predicting the entity. The `zero_digits()` function is utilised to achieve this. It replaces all digits in a string by a zero. This way, there would be less noise and the model concentrates on the alphabetic information. The text is also split up into sentences. The `load_sentences()` function is used for this.
2. **Converting labeling scheme from BIO to BIOES** The `update_tag_scheme()` function is used to convert the tagging scheme from BIO used in CoNLL dataset to BIOES for training.
3. **Mapping words and characters to their corresponding id and tag.** This is needed to employ tensor operations within the neural network design allowing it to be comparatively speedier. The three functions used for mapping are `word_mapping`, `char_mapping` and `tag_mapping`
4. **Loading word embedding** The pre-trained Global Vectors (GloVe) embeddings are loaded. We utilized the word embedding file glove.6B.100d.txt to employ 100-dimensional vectors that were trained on the Wikipedia 2014 and Gigaword 5 corpora, which includes 6 billion words. These vectors will be concatenated with the vector representation of each word.

3.2.3 Result

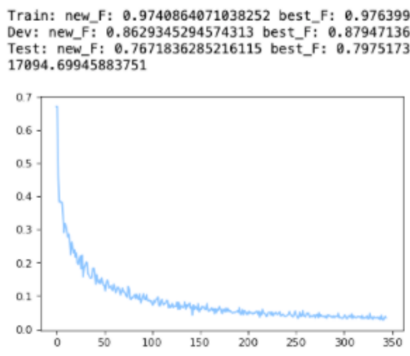
CNN + BiLSTM + CRF model is trained as the benchmark, and the result is plot as Figure 6.



```
Train: new_F: 0.998507017020006 best_F: 0.9985709105645969
Dev: new_F: 0.9243484861263389 best_F: 0.9293899561846983
Test: new_F: 0.8774710596616206 best_F: 0.8825148743450848
50307.10415959358
```

Figure 6: Test set f1 score on CNN + BiLSTM + CRF

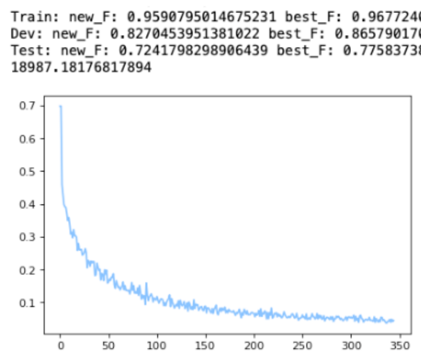
Then the word-level decoder is replaced with one layer CNN, and the result is shown in Figure 7



```
Train: new_F: 0.9740864071038252 best_F: 0.9763994206355968
Dev: new_F: 0.8629345294574313 best_F: 0.8794713656387666
Test: new_F: 0.7671836285216115 best_F: 0.797517315822614
17094.69945883751
```

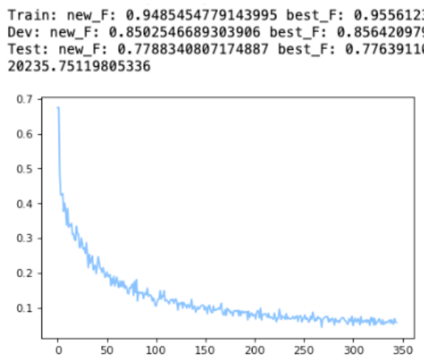
Figure 7: Test set f1 score on CNN + CNN(1 layer) + CRF

The models with different layers of CNN word-level encoder are trained and tested as shown in Figure 8, 9, 10



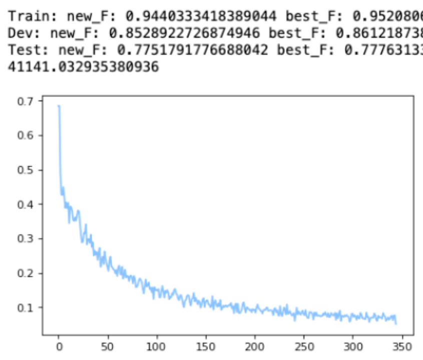
```
Train: new_F: 0.9590795014675231 best_F: 0.9677240437741322
Dev: new_F: 0.8270453951381022 best_F: 0.8657901702991263
Test: new_F: 0.7241798298906439 best_F: 0.775837389063842
18987.18176817894
```

Figure 8: Test set f1 score on CNN + CNN(2 layer) + CRF



```
Train: new_F: 0.9485454779143995 best_F: 0.9556123535676251
Dev: new_F: 0.8502546689303906 best_F: 0.8564209792646802
Test: new_F: 0.7788340807174887 best_F: 0.7763911044383973
20235.75119805336
```

Figure 9: Test set f1 score on CNN + CNN(3 layer) + CRF



```
Train: new_F: 0.9440333418389044 best_F: 0.9520806255726249
Dev: new_F: 0.8528922726874946 best_F: 0.8612187389926031
Test: new_F: 0.7751791776688042 best_F: 0.7776313311993974
41141.032935380936
```

Figure 10: Test set f1 score on CNN + CNN(4 layer) + CRF

F1 score is used as the evaluation metric. The result is summarized as Table 3. We use the F1 score on the validation set to determine the best model. From the results in the table above, the 1 Layer CNN performed the best with the highest F1 score out of all the other CNN models. From the results, the more CNN layers implemented results in poorer F1 scores on the training set. Adding more CNN layers is supposed to extract more features and improve the accuracy, but it seems that the word embeddings have shallow features as adding more layers to train the model worsened the F1 scores. However, there is an exception for the validation set where the 4 Layer CNN performed better than the 3 Layer CNN. This could be due to the validation set being too small, producing an exception. There are a few ways to get more accurate results. We could form multiple sets of training, validation and test sets and compare their results. If the drop in accuracy is caused by overfitting we could stop training prematurely the moment validation accuracy drops, and only training over a smaller number of epochs.[4] The BiLSTM recurrent neural network implementation still performed better than any of the CNN implementations. This is due to the different design of the implementation where the BiLSTM is able to make its predictions given the context based on a sequence of data in two directions while CNN makes use of convolution kernels to account for spatial correlation in the data.

	Train	**Valid**	**Test**
1 Layer CNN	0.9764	0.8795	0.7975
2 Layer CNN	0.9677	0.8658	0.7758
3 Layer CNN	0.9556	0.8564	0.7764
4 Layer CNN	0.9521	0.8612	0.7776
LSTM	0.9986	0.9294	0.8825

Table 3: Result of F1 summary of different network configuration

4. Member Information & Contribution

- **KOH YIANG DHEE MITCHELL**
  - Question Two
  - Report Writing
- **RYAN LEE RUIXIANG**
  - Question Two
  - Report Writing
- **LI PINGRUI**
  - Question One
  - Report Writing
  - LaTeX Formatting

- **TANG YUTING**
  - Question One
  - Report Writing
  - LaTeX Formatting
- **WANG BINLI**
  - Question One
  - Report Writing
  - LaTeX Formatting

References

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003.

[2] Ben Mann. How to sample from language models, May 2019.

[3] Pytorch. examples/main.py at master · pytorch/examples, Jul 2020.

[4] Nils Schlüter. Don't overfit!-how to prevent overfitting in your deep learning models, Jun 2019.