Assignment 4

Spring 2019
BBM 497: Introduction to Natural Language Processing Lab.

Burak Emre Özer
burakemreozer@gmail.com

May 31, 2019

1 Introduction

A language model is at the core of many NLP tasks, and is simply a probability distribution over a sequence of words. It can also be used to estimate the conditional probability of the next word in a sequence.

Let's assume we have the sequence ***[my, cat's, breath, smells, like, cat, __]*** and we want to guess the final word. A language model would estimate the probability for every word in the vocabulary. There are several ways to create a language model. The most straightforward is an n-gram model that counts occurrences to estimate frequencies.

Text Generation can be be modeled using deep learning models such as **Feed-Forward Neural Networks (FNN)**. For the text generation task, FNNs are trained on a very large corpus to predict the next word as a bi-gram language model.

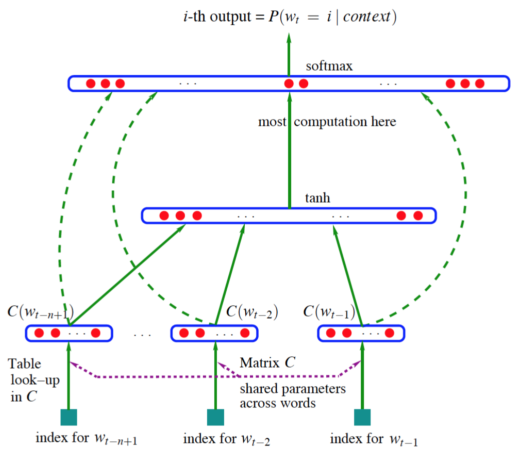

Figure 1: Bengio's FFN net in 2003

In this assignment, we will implement an n-gram (bigram) neural language model us-ing feed forward neural networks for Text Generation by using DyNet deep learning library.

2 Task 1: Build Feed-Forward Neural Network Language Model (FNN)

Deep feedforward networks, also often called **feedforward neural networks**, or multilayer perceptrons(MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f*. These models are called feedforward because information ows through the function being evaluated from x, through the intermediate computations used to dene f, and nally to the output y. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks.

<START> was placed at the beginning of each poem and ***<END>*** tokens were placed at the end of each poem. In order to use each word in the data set in the model, we expressed it as one-hot vector. The one-hot vector in the history feed into the feed-forward neural network to predict the probability of the next word. The feed-forward neural network is defined mathematically as follows:

$$y = U \cdot f(W \cdot x + b) + d$$

For each context, one can first compute the logit score for each candidate next word and then normalize all the logits using softmax function to get the probabilities. This will be repeated for many epochs to optimize the weights in the model.

```
    def train(self, X, y):
        total_loss = 0

        for inp, out in zip(X, y):
            o_idx = np.argmax(out)

            dy.renew_cg() # create a new computation graph for each input
            inp = dy.inputVector(inp)
            f = dy.tanh(self.W * inp + self.b_bias)

            # calculate the softmax of probs then calculate negative log
            # which is equivalent to: dy.pick(-dy.log(dy.softmax()))
            loss = dy.pickneglogsoftmax(self.U * f + self.d_bias, o_idx)
            total_loss += loss.npvalue()

            # back-propagating then updating the parameters
            # using dy.SimpleSGDTrainer
            loss.backward()
            self.trainer.update()

        print(total_loss/len(X))
```

3 Task 2: Poem Generation

Once the model is trained, it is straightforward to generate a new text by iteratively predicting the next word as a n-gram language model. Select a word based on the sequence so far, add this word to the sequence, and repeat. We will start with start token. Then, we will predict one word at each time using the previous word and feeding this word as input to the neural network in the next time step.

```
def generate_poem(start_word, num_of_lines):

    ...
    # poem generation stops until the end of poem is reached or
    # the total number of lines is reached
    while next_idx != 1:

        next_idx, prob = model.predict_output(pre_idx_vector)
        prob_list.append(prob)

        if index2word[next_idx] == "<NEWL>":
            line_count += 1

        if line_count == num_of_lines:
            break

        sentence_idx.append(next_idx)
        pre_idx_vector = oht_inputs[next_idx]
```

4 Task 3: Evaluation

After we generate your poems automatically by using our trained feed-forward neural network language model, then we will compute the perplexity of each generated poem with this formula:

$$2^{-l} \text{ where } l = \frac{1}{M}\sum_{i=1}^{m} \log p(s_i)$$

As can be seen, perplexity values decreased with increasing number of epoch. Likewise, this effect can be seen as the learning rate increases.

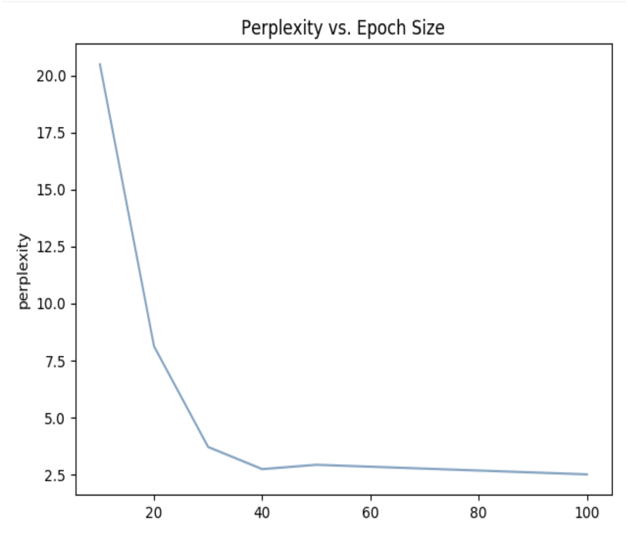

Figure 2: perplexity vs. epoch

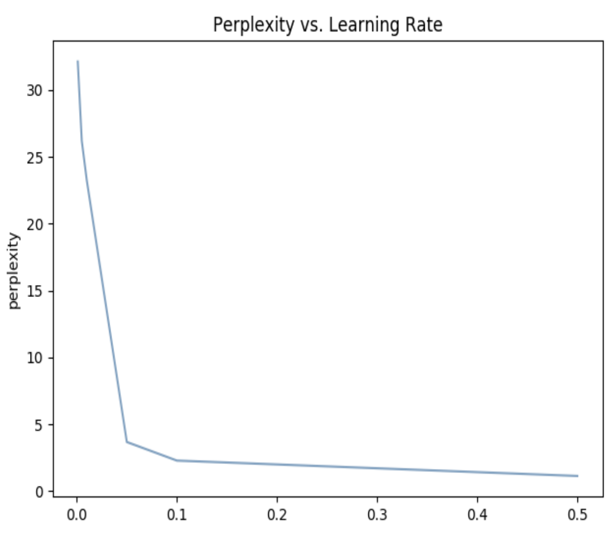

Figure 3: perplexity vs. learning rate

Below you can see 5 different poems automatically generated.

- hidden size: 512
- learning rate: 0.05
- poem size: 100

```
### POEM 1
PERPLEXITY: 5.910756039890256
<START> yet up starved still a amid for thing distracted in music <NEWL>
also heart the yelling <NEWL>
girl faring do

### POEM 2
PERPLEXITY: 6.6390927917678475
<START> adieu soon beds what beside noise tear-drop stop lost
exultant as to green make night love to past <NEWL>
girl poetry dancing upon your men disaster harbour perning than this <NEWL>
girl when there demon

### POEM 3
PERPLEXITY: 3.3221586291277876
<START> girl rose and the mareotic find <NEWL>
unknown take at <NEWL>
girl poetry dancing upon the bales found improvising

### POEM 4
PERPLEXITY: 3.163849425881783
<START> there a itself from desperate <NEWL>
or broken of and couple <END>
```

```
### POEM 5
PERPLEXITY: 6.060615361261959
they perning an think broom <NEWL>
girl faring beast right were of as through tree little loud
and way better she between cant just <END>
```