Deep Learning Assignment 1 Report

Multilayer Perceptron from Scratch with Word Embeddings

Taylan Tatlı

Physics PhD student

taylan.tatl@gmail.com

CMPE 597 Assignment 1

Department of Computer Engineering,
Boğaziçi University, 05/04/2023

Contents

1 Introduction

In this project, the subject is to train a neural language model using a multi-layer perceptron. This network receives 3 consecutive words as the input and purpose is to predict the next word. The model should use cross-entropy loss function to maximize the probability of the target word.

The network consists of a 16-dimensional embedding layer, a 128-dimensional hidden layer, and one output layer. The input consists of 3 consecutive words, provided as integer-valued indices representing a word in our 250-word dictionary. We need to convert each word to its one-hot representation and feed it to the embedding layer, which will be 250×16 dimensional. The hidden layer will have a sigmoid activation function, and the output layer is a softmax over the 250 words in our dictionary. After the embedding layer, 3-word embeddings are concatenated and fed to the hidden layer.

Main references: [1], [2], [3], [4], [5], [6].

2 Math for Forward Propagation

Forward propagation is basically that the input data is fed in the forward direction through the network. Each hidden and other layers accept the input data, processes it with an activation function or ust matrix multiplication and give the proccesed output to the successive layer. So, output of the each layer is the input for the succesive layer. General scheme for our model as follows:

The value of each output neuron can be calculated as the follows:

$$y_j = b_j + \sum_i x_i w_{ij}$$

We can compute this formula using matrices and dot product for every output neuron, in below, an example for i feature and for one input is given.

$$X = [x_1, x_2 ... x_i], \qquad W = \begin{bmatrix} w_{11} & ... & w_{1j} \\ . & & \\ . & & \\ & & \\ w_{i1} & & w_{ij} \end{bmatrix} \qquad B = [b_1, b_2 ... b_i]$$

$$Y = XW + B$$

Our model takes on the following structure X is the input vector and with shape (Batch_size,250), we have 250 word. W_1 is the embedding layer weight matrix with shape (250,16). A_1 is the output of embedding layer and it is also input for hidden layer. Hidden layer weight matrix is W_2 and output without activation is A_2 and with sigmoid activation is Z_2. Weights for output layer is W_3 and its output without activation is A_3 and with softmax activation is Z_3. Softmax activation output contains prediction probability vectors. The structere is given in page 3.

$$A_1 = XW_1$$

$$A_2 = A_1W_2 + B2$$

$$Z_2 = sigmoid(A_2)$$

$$A_3 = Z_2W_3 + B3$$

$$Z_3 = softmax(A_3)$$

3 Math for Backward Propagation

Our neural network is trained with gradient descent and so it requires the calculation of the gradient of the error function, let we say it is $E(X, \theta)$ and here θ represents the parameters weight and bias at the end of each iteration these parameter should be updated and general formula for that is given below:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial(E, \theta^t)}{\partial \theta}$$

t denotes the number of iteration. So all we need to update parameters is to know partial derivatives $\frac{\partial E}{\partial W}$ and $\frac{\partial E}{\partial B}$. For the calculation below, Y is the output, E is the error function, X is the input, B is the bias and W is the weight for layers. The references mainly used here are [2], [4], [5], [6]. Let assume we know $\frac{\partial E}{\partial Y}$. Calculate $\frac{\partial E}{\partial W}$, $\frac{\partial E}{\partial B}$ and $\frac{\partial E}{\partial X}$. i: input size j: output size

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial W_{11}} & .. & \frac{\partial E}{\partial W_{1j}} \\ . & & \\ . & & \\ \frac{\partial E}{\partial W_{i1}} & & \frac{\partial E}{\partial W_{ij}} \end{bmatrix}$$

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial W_{ij}} \cdot \cdot \cdot \cdot \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial W_{ij}}$$

$$= \frac{\partial E}{\partial y_j} x_i$$

$$\Rightarrow \quad \frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & .. & \frac{\partial E}{\partial y_j} x_1 \\ . & & \\ . & & \\ \frac{\partial E}{\partial y_1} x_1 & .. & \frac{\partial E}{\partial y_j} x_i \end{bmatrix}$$

$$\Rightarrow \quad \frac{\partial E}{\partial W} = \begin{bmatrix} x_1 \\ . \\ . \\ xi \end{bmatrix} \left[\frac{\partial E}{\partial y_1} ... \frac{\partial E}{\partial y_j}\right]$$

$$= X^T \frac{\partial E}{\partial Y} \qquad [1]$$

Now, lets look at the same for bias parameter.

$$\frac{\partial E}{\partial B} = \left[\frac{\partial E}{\partial b_1}, \frac{\partial E}{\partial b_2}. \quad . \quad . \quad \frac{\partial E}{\partial b_j}\right]$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j}. \quad . \quad . \quad \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j}$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial y_j} = \frac{\partial E}{\partial Y} \qquad [2]$$

Also lets, look at $\frac{\partial E}{\partial X}$,

$$\frac{\partial E}{\partial X} = \left[(\frac{\partial E}{\partial y_1} w_{11} + ... + \frac{\partial E}{\partial y_j} w_{1j}). \ . \ .(\frac{\partial E}{\partial y_1} w_{11} + ... + \frac{\partial E}{\partial y_j} w_{ij})\right]$$

$$= \left[\frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2}. \quad . \quad . \quad \frac{\partial E}{\partial y_j}\right] \begin{bmatrix} w_{11} & .. & w_{i1} \\ . & & \\ . & & \\ w_{1j} & .. & w_{ij} \end{bmatrix}$$

$$= \frac{\partial E}{\partial Y} W^T \qquad [3]$$

We have equations [1], [2], [3] without activation consideration, lets look at what happened to activation separately. Below, f is the activation.

$$Y = [f(x_1), f(x_2).....f(x_i)] = f(X)$$

$$\frac{\partial E}{\partial X} = \left[\frac{\partial E}{\partial x_1}, \frac{\partial E}{\partial x_2}. \quad . \quad . \quad \frac{\partial E}{\partial x_i}\right]$$

$$\frac{\partial E}{\partial X} = \left[\frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1}, . \quad . \quad . \quad \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_1}\right]$$

$$\frac{\partial E}{\partial X} = \left[\frac{\partial E}{\partial y_1} f'(x_1), . \quad . \quad . \quad \frac{\partial E}{\partial x_i} f'(x_i)\right]$$

$$= \frac{\partial E}{\partial Y} F'(X) \qquad [4]$$

So for the gradient calculation of Backward Propagation, we all need derivative of activation functions and error functions with respect to output of the layer. In our case, activation functions are sigmoid and softmax and error function is cross entropy loss. Lets look at them. First derivative of the softmax and cross entropy loss. It is inspired from reference..

Let z is the output of the neural network, then softmax(z) outputs a vector of probabilities. The expression for softmax activation is below.

$$softmax(\mathbf{z})_i = p_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

$$\sum_{j=1}^{N} e^{z_j} = \Sigma N$$

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\Sigma N}$$

The derivative of the softmax output p_i with respect to the raw output z_i of the network will be considered two different cases. Case:1 is for when i=j and Case 2 i $\neq$ j.

When $i = j$:

$$\frac{\partial p_i}{\partial z_i} = \frac{\partial \frac{e^{z_i}}{\Sigma N}}{\partial z_i}$$

$$= \frac{\partial \frac{e^{z_i}}{\Sigma N}}{\partial z_i} = \frac{\Sigma N . \frac{\partial e^{z_i}}{\partial z_i} - e^{z_i} . \frac{\partial \Sigma N}{\partial z_i}}{(\Sigma N)^2}$$

$$\Rightarrow \frac{\partial \Sigma N}{\partial z_i} = \frac{\partial \sum_{j \neq i} e^{z_j}}{\partial z_i} + \frac{\partial e^{z_i}}{\partial z_i} = e^{z_i} \frac{\partial e^{z_i}}{\partial z_i} = e^{z_i}$$

$$\frac{\partial p_i}{\partial z_i} = \frac{\Sigma N . e^{z_i} - e^{z_i} . e^{z_i}}{(\Sigma N)^2} = \frac{e^{z_i}}{\Sigma N} \left[1 - \frac{e^{z_i}}{\Sigma N} \right] = p_i (1 - p_i)$$

When $i \neq j$:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\Sigma N}}{\partial z_j}$$

$$= \frac{\Sigma N . \frac{\partial e^{z_i}}{\partial z_j} - e^{z_i} . \frac{\partial \Sigma N}{\partial z_j}}{(\Sigma N)^2}$$

$$= 0 - \frac{e^{z_i}}{\Sigma N} . \frac{e^{z_j}}{\Sigma N}$$

$$\Rightarrow \frac{\partial p_i}{\partial z_j} = -p_i p_j$$

Now we should compute the derivative of the cross-entropy loss function with respect to the output of the neural network, p is the predicted, probabilistic distribution and **t** is the true distribution. The chain rule and the derivatives of the softmax activation function are used.

$$\frac{\partial L}{\partial z_i} = -\sum_{j=1}^{N} \frac{\partial (t_j log p_j)}{\partial z_i}$$

$$= -\sum_{j=1}^{N} t_j \frac{\partial (log p_j)}{\partial z_i}$$

$$= -\sum_{j=1}^{N} t_j \frac{1}{p_j} \frac{\partial p_j}{\partial z_i} = -\frac{t_i}{p_i} \frac{\partial p_i}{\partial z_i} - \sum_{j \neq i} \frac{t_j}{p_j} \frac{\partial p_j}{\partial z_i}$$

$$= -\frac{t_i}{p_i} p_i(1-p_i) - \sum_{j \neq i} \frac{t_j}{p_j}(-p_j p_i) = -t_j + t_i p_i + \sum_{j \neq i} t_j p_i = -t_i + \sum_{j=1}^{N} t_j p_i$$

$$= -t_i + p_i \sum_{j=1}^{N} t_j \implies \frac{\partial L}{\partial z_i} = p_i - t_i$$

Now, derivative od the other activation function sigmoid.

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

$$\frac{df(x)}{dx} = \frac{(e^x+1)(e^x) - e^x(e^x)}{(e^x+1)^2} \quad \text{(apply quotient rule)}$$

$$= \frac{e^x}{(e^x+1)^2} = \frac{1}{e^{-x}(1+e^{-x})^2} \quad \text{(substitute } e^{-x} \text{ for z)}$$

$$= \frac{1}{(1+z)^2} = \frac{1}{(1+e^{-x})^2}$$

Now we simplify this expression.

$$f'(x) = \frac{e^x}{(1+e^{-x})^2}$$

$$\begin{aligned}
&= \frac{1}{1+e^{-x}} \cdot \frac{e^x}{1+e^x} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{1+e^{-x}}{1+e^{-x}} \cdot \frac{e^x}{1+e^x} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{1}{1+e^x} \cdot e^x \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^x}\right) \\
&= f(x)(1-f(x))
\end{aligned}$$

If we combine the derivative solutions and our general approaches [1],[2],[3], we can get the back propagation for whole network calculating for each layer by each layer.

4 Network Class

This part contains class based network where having the forward, back- ward propagation, and activation functions. It also contains network initialization, extra activation function to test model, cross entropy loss and accuracy calculation functions. Forward propagation part contains matrix multiplication of input_data and embedding layer weights, after that embedding we need to multiply with hidden layer weights without any activation functions, and after that this multiplication is activated sigmoid and activated output is input for output layer with activation function softmax. Backward propagation part contains calculating gradients and updating parameters. General structure of calculating gradients for weights for each layer and is taking input, transpose it and multiply with output error. General structure of calculating gradients for biases is output error itself. Updating parameters uses gradient decent, basic general formula for weights is w = w - learning_rate * gradients, for bias it is same, b = b-learning_rate * gradients. A part of code is given in Figure 1.

```
import numpy as np

class Network:

    def __init__(self):
        self.weights1=np.random.rand(250,16)  # embedding layer weights initialization, 250--input_size, 16-- output size
        self.weights2=np.random.rand(16,128)  # hidden layer with input size 16 and output size 128
        self.weights3=np.random.rand(128,250) # We have total of 250 words outputlayer should give this
        # no need bias for embedding layer
        self.bias2 = np.zeros(128)
        self.bias3 = np.zeros(250)


    def forward_propagation(self,input_data,target_output):

        embedding = np.dot(input_data,self.weights1) # matrix multiplication of input_data and embedding layer weights
        # after embedding we need to multiply with hidden layer weights without any activation functions,
        hidden1 = np.dot(embedding,self.weights2)+self.bias2
        hidden1_output = self.sigmoid(hidden1) # hidden layer activated output
        output_layer = np.dot(hidden1_output,self.weights3) + self.bias3 # output layer shoul take hidden1 output as an inp
function
        softmax_output = self.softmax(output_layer) # propabilistic output

        # and now we should calculate loss after our forward propagation.

        loss = self.cross_entropy(softmax_output,target_output)

        return loss, softmax_output, embedding, hidden1_output # we return output of each layer.


    def backward_propagation(self,input_data,embedding,hidden1_output,softmax_output,target_output):
        d_error3 = softmax_output - target_output # d_error3: from loss to output layer
        #structure of weights gradient for each layer -- np.dot(input.T,output_error(d_error3))
        #structure of bias gradient for each layer -- np.sum(d_error3)
        weights3_grad = np.dot(hidden1_output.T, d_error3) # weight 3 gradients
        bias3_grad = np.sum(d_error3, axis=0) # bias 3 gradients
```

Figure 1: A part of Network.py.

5 Main.py

In this part, main.py is created. It contains functions for loading the dataset, shuffling the training data, dividing it into mini-batches and one-hot encoding methods. The training part contains for loop for the epochs and after each epoch model is evaluated, at the end, training and validation accuracy's are given and a plot is produced contain the accuracy's for each epoch. Finally, the model is saved as "new_model_3.pk". Train and validation accuracy at the end of the model is given in Table 1. and plot for training and validation accuracy's for each epoch given in Figure 2. The chosen parameters here is batch size = 100 number of epoch = 20 and learning rate = 0.02.

Table 1: Train and Validation accuracies at and of training.

Model	**Train Accuracy**	**Validation Accuracy**
	29.961%	29.480%

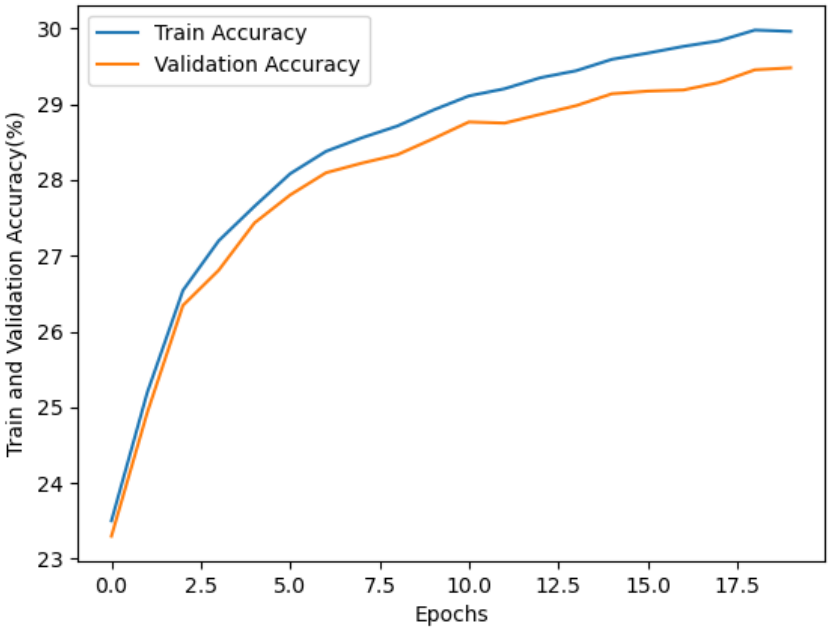

Figure 2: Train and Validation accuracy's for each epoch.

6 Eval.py

This part contains the evaluation of test data. First it loads the saved model and implement data into this model and gives test accuracy. Second part of the model contains the function to predict next work for data points; 'city of new', 'life in the', 'he is the'. Test accuracy result is given Table 2 below. The predicted words are York, . , best respectively. Two of these three words are meaningful but one, the result dot, is not meaningful, it is obviously wrong prediction and it is probably connected with low model accuracy. The predicted words result are given in Figure 3.

Table 2: Test accuracy.

Model	**Train Accuracy**	**Validation Accuracy**	**Test Accuracy**
	29.961%	29.480%	29.402%

```
NameError: name 'data_vocab' is not defined
(base) [taylan@localhost Deep Learning Assignment]$ python Eval.py
-> Testing is started.
--> model.pk is loaded.
--> Test accuracy: 29.402 %
city of new -> york
life in the -> .
he is the -> best
(base) [taylan@localhost Deep Learning Assignment]$
```

Figure 3: Test accuracy added

7 Tsne.py

This part contains 2-D plot of the embeddings after getting 16 dimensional embedding using tsne, which maps nearby 16-dimensional embeddings close to each other in the 2-D space. It first load the model parameters and return learned embeddings. Raw plot is given in Figure 4 and the some selected clusters plot is given Figure 5.

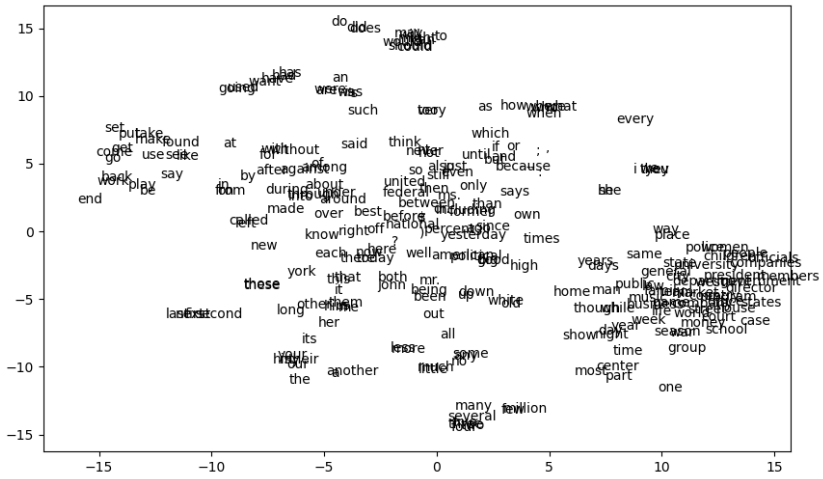

Figure 4: 2-D plot of the embeddings.

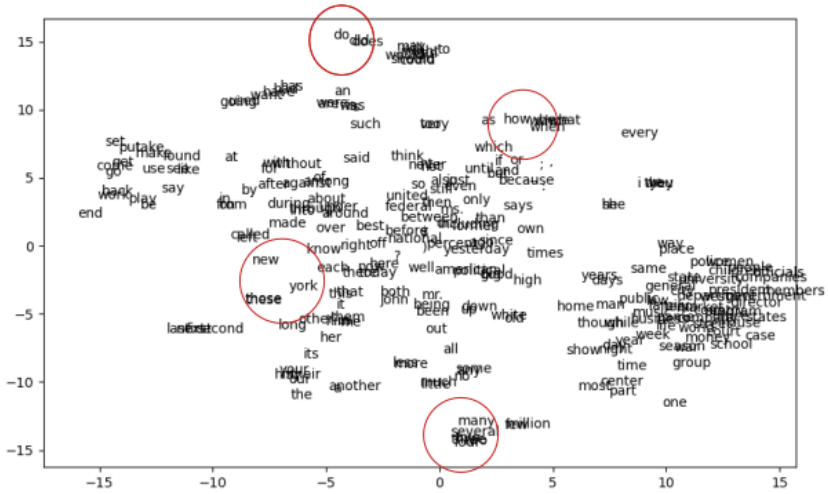

Figure 5: Some clusters of the words.

It looks like same kind of words are clustered. For instance, do and does clustered at the top, many and several kind of words at the bottom. Another clustering is related

to probabilities of the words use at the same time. For instance, new and york are very close to each other at the left, but not close as do and does or many and several are.

References

[1] GeekforGeeks, "Implement your own word2vec(skip-gram) model in python." https://www.geeksforgeeks.org/implement-your-own-word2vecskip-gram-model-in-python/.

[2] J. Tae, "Building neural network from scratch." https://jaketae.github.io/study/neural-net/.

[3] J. Tae, "Word2vec from scratch." https://jaketae.github.io/study/word2vec/.

[4] O. Aflak, "Neural network from scratch in python." https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65.

[5] "Derivative of the sigmoid function." https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e.

[6] B. Priya, "Derivative of the softmax cross-entropy loss function." https://www.pinecone.io/learn/cross-entropy-loss/.