BOĞAZIÇI UNIVERSITY

DEEP LEARNING
CMPE 597

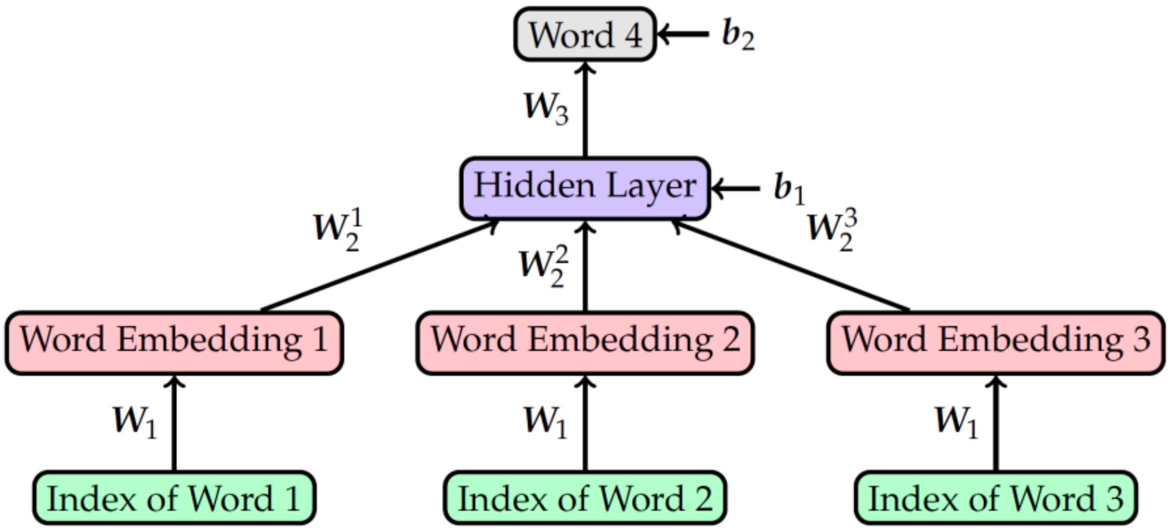
Assignment 1

Author:
Y. Harun Kıvrıl

02 May 2021

Introduction

This project aims to implement the following feed forward neural network structure by only using the libraries for calculations. After the implementation the network is trained on train data with hyperparameter tuning on validation data. The network obtains the best validation accuracy is selected and it evaluated on test data. For the evaluation besides the accuracy and loss values a tsn-e plot of the embeddings are investigated and the model's predictions for three examples are evaluated.



1 Forward Propagation

Forward propagation equation of the given network with mini batch size n and dictionary size 250 can be written in the following form.
Let $I_i \in R^{nx250}$ be the one hot encoded batch of ith input word and $W1$ be the weight matrix of embedding layer. Then the embedding of the input words are:

$$e^{[1]} = I_1 W_1, \quad e^{[2]} = I_2 W_1, \quad e^{[3]} = I_3 W_1$$

where $W_1 \in R^{250x16}$ and as a result $e^{[i]} \in R^{nx16}$. After that each embedding are concatenated into a single layer:

$$e = [e^{[1]}, e^{[2]}, e^{[3]}]$$

where $e \in R^{nx48}$. Later this embeddings are mapped to the hidden layer using sigmoid activation function σ_1:

$$h = eW_2 + b_1$$

$$o^{[1]} = \sigma_1(h)$$

where $W_2 \in R^{48x128}$, $b_1 \in R^{128}$, h and $o^{[1]} \in R^{nx128}$. Finally the outputs of the hidden layer is mapped to the output layer using softmax activation function σ_2:

$$z = o^{[1]}W_3 + b_2$$

$$o^{[2]} = \sigma_2(z)$$

where $W_3 \in R^{128x250}$, $b_2 \in R^{250}$, and $o^{[2]}, z \in R^{nx250}$.
After obtaining the outputs from network, the cross entropy loss is calculated using the following formula:

$$L = -\frac{1}{n}\sum_{i=1}^{n} t_i log(o_i^{[2]}))$$

where $t_i \in R^{250}$ is the true label of ith observation in mini batch and $o_i^{[2]}$ is the model output of that observation.

2 Backward Propagation

The back propagation of the given network can be written in the following form using the chain rule. In order to do this an intermediate variable δ is introduced to store common terms. For the derivative of W_3 and b_2:

$$\delta_3 = \frac{\partial L}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial z}$$

$$\frac{\partial L}{\partial W_3} = \delta_3 \frac{\partial z}{\partial W_3}$$

$$\frac{\partial L}{\partial b} = \delta_3 \frac{\partial z}{\partial b_2}$$

When we calculate this chain one by one:

$$\frac{\partial L}{\partial o_i^{[2]}} = \frac{-t_i}{o_i^{[2]}}$$

and

$$\frac{\partial o_i^{[2]}}{\partial z_j} = \begin{cases} o_i^{[2]}(1-o_j^{[2]}) & i=j \\ -o_i^{[2]}o_j^{[2]} & i \neq j \end{cases}$$

where o_i is the result of the i^{th} output neuron and z_j is the output of the j^{th} neuron before activation. When the derivatives of cross entropy loss and softmax activation are combined the result becomes $o^{[2]} - t$ and this result can be used to make calculations faster. The network has both implementations but uses the second one to obtain results faster. Additionally:

$$\frac{\partial z_i}{\partial W_{3_j}} = o^{[1]} \quad \frac{\partial z}{\partial b_2} = 1$$

For the derivative of W_2 and b_1:

$$\delta_2 = \delta_3 \frac{\partial z}{\partial o^{[1]}} \frac{\partial o^{[1]}}{\partial h}$$

$$\frac{\partial L}{\partial W_2} = \delta_2 \frac{\partial h}{\partial W_2}$$

and for bias:

$$\frac{\partial L}{\partial b_1} = \delta_2 \frac{\partial h}{\partial b_1}$$

In order to calculate these gradients we need to the following calculations:

$$\frac{\partial z^{[2]}}{\partial o^{[1]}} = W_3$$

$$\frac{\partial o^{[1]}}{\partial h} = o^{[1]}(1-o^{[1]})$$

and

$$\frac{\partial h_j}{\partial W_{2_i}} = e \quad \frac{\partial z}{\partial b^1} = 1$$

For derivative of W_1:

$$\delta_1 = \delta_1 \frac{\partial h}{\partial e}$$

Since, there are three words as input and they all use W_1, δ_1 is needed to be separated according to corresponding $e^{[i]}$ parts:

$$[\delta_{1,1}, \delta_{1,2}, \delta_{1,3}] = \delta_1$$

Then the derivative becomes:

$$\frac{\partial L}{\partial W_1} = \sum_{i=1}^{3} \delta_{1,i} \frac{\partial e^{[i]}}{\partial W_1}$$

$$\frac{\partial h}{\partial e} = W_2$$

$$\frac{\partial e^{[i]}}{\partial W_1} = I_i$$

Since, the training done in minibatches for gradient estimations, the total gradient obtained from the minibatch is divided to the batch size. As a summary the gradients are calculated in the following way:

$$\delta_3 = o^{[2]} - t \in R^{nx250}$$

$$\frac{\partial L}{\partial W_3} = \frac{1}{n}[\delta_3^T o^{[1]}]^T \in R^{128x250}$$

$$\frac{\partial L}{\partial b_2} = \frac{1}{n}\sum_{i=0}^{n} \delta_{3_i} \in R^{250}$$

$$\delta_2 = (\delta_3 W_3^T) \odot (o^{[1]}(1 - o^{[1]})) \in R^{nx128}$$

$$\frac{\partial L}{\partial W_2} = \frac{1}{n}[\delta_2^T e]^T \in R^{48x128}$$

$$\frac{\partial L}{\partial b_1} = \frac{1}{n}\sum_{i=0}^{n} \delta_{2_i} \in R^{1}28$$

$$\delta_1 = (\delta_2 W_2^T) \in R^{nx48}$$

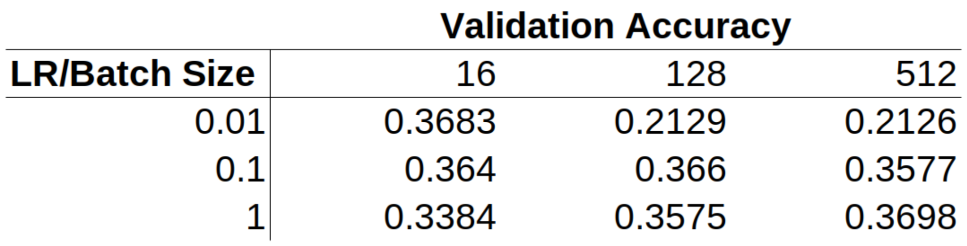
$$[\delta_{1,1}, \delta_{1,2}, \delta_{1,3}] = \delta_1$$

$$\frac{\partial L}{\partial W_1} = \sum_{j=0}^{3} \frac{1}{n}\sum_{i=0}^{n}[\delta_{1_i,j}^T I_{j_i}]^T \in R^{250x16}$$

3 Training and Evaluating the Network

The architecture given in the figure is implemented in a network class using python's numpy[2] library. The weight and bias matrices are initialized randomly using normal distribution with $\mu = 0$ and $\sigma = 0.01$. The forward and backward functions of the network are written according to the equations given above. Also a function for updating the weights with given learning rate and network's gradients is introduced.

Since train, validation and test sets are already separated, the only thing missing for training was the hyperparameters: learning rate and batch size. In order to tune this parameters three candidate levels are selected as large, medium and small. For the learning rate $(1, 0.1, 0.01)$ are selected and for batch size $(16, 128, 512)$ is considered. Then the network is trained with each combination of these values and the train and validation loss and accuracies are recorded for each epoch. Since it takes a considerable amount of time to train the network, the maximum epoch number is determined as 250 and an early stopping mechanism is introduced. This mechanism stopped the training if the network obtained less validation accuracy than the best accuracy so far in 8 consecutive epochs. The selection of the hyperparameters is made based on the epoch with best validation accuracy in each hyperparameter combination. At the end of the tuning procedure the following table is obtained:



Validation Accuracy

LR/Batch Size	16	128	512
0.01	0.3683	0.2129	0.2126
0.1	0.364	0.366	0.3577
1	0.3384	0.3575	0.3698

Figure 1: Results obtained from hyperparameter tuning

According to these results. The batch size is selected as 512 and the learning rate is chosen as 1. There are some accuracies far away from the maximum one. The reason for that is the updates become smaller with decreasing learning rate and increasing bacth size (since we use the average of the batch as estimate) and 250 epoch is not enough for them to give better results.

The following plots show the evolution of losses and accuracies in the training of the model with the selected hyperparameters. The plots are generated using python's matplotlib[3] library.

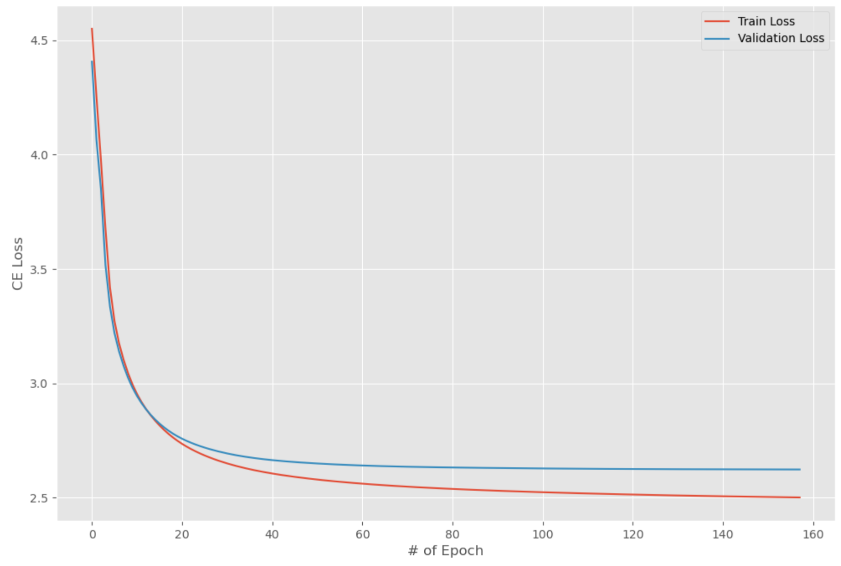


Figure 2: Training and validation loss per epoch for the selected model

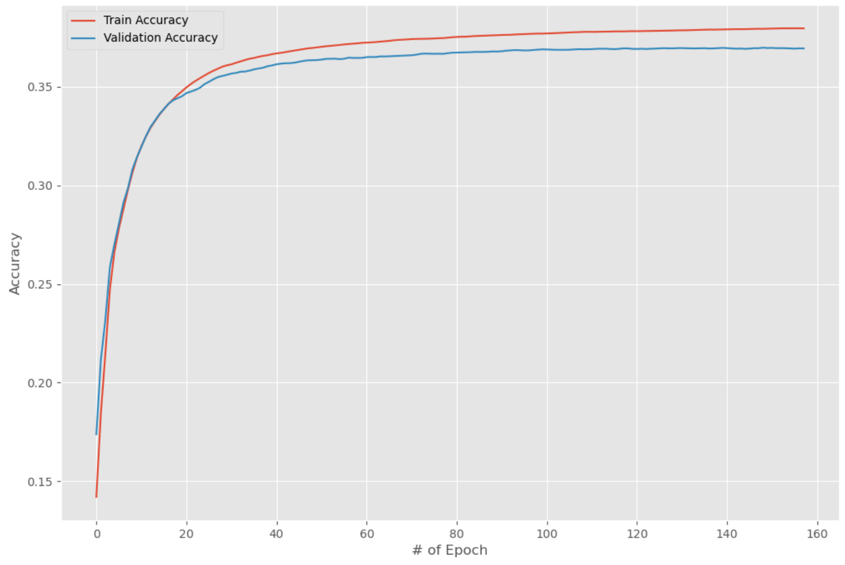
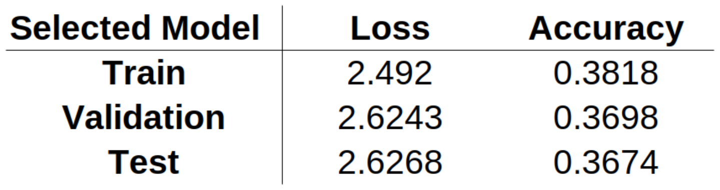


Figure 3: Training and validation accuracy per epoch for the selected model

The selected model is also evaluated in the test set and following evaluation table is constructed.



Selected Model	Loss	Accuracy
Train	2.492	0.3818
Validation	2.6243	0.3698
Test	2.6268	0.3674

Figure 4: Training and validation loss per epoch for the selected model

4 Results and Discussion

As the table above shows, the model obtained 0.3674 accuracy on the test set which is similar to the validation performance. It is not a good ratio however while evaluating this ratio the following facts should be considered: the number of classes are very much and the model is a very simple language model and its architecture is given in advance.

Besides to accuracy evaluating the embeddings obtained from the model gives us an idea about the performance of the model. If the model has learnt something during the training we expect to have the similar words are close to each other in the embedding space. Since the embedding space is 16 dimensional, it is not possible to show the distances visually, therefore a dimensionality reduction technique called TSNE is used to reduce the dimension to 2.

The embeddings are obtained by multiplying the identity matrix of size 250 with the weights of the embedding layer W_1. Then the embeddings are fed into sklearn's[5] TSNE[6] model and 2D representation of these embeddings are obtained. Later the embeddings are plotted with using the words as annotation.

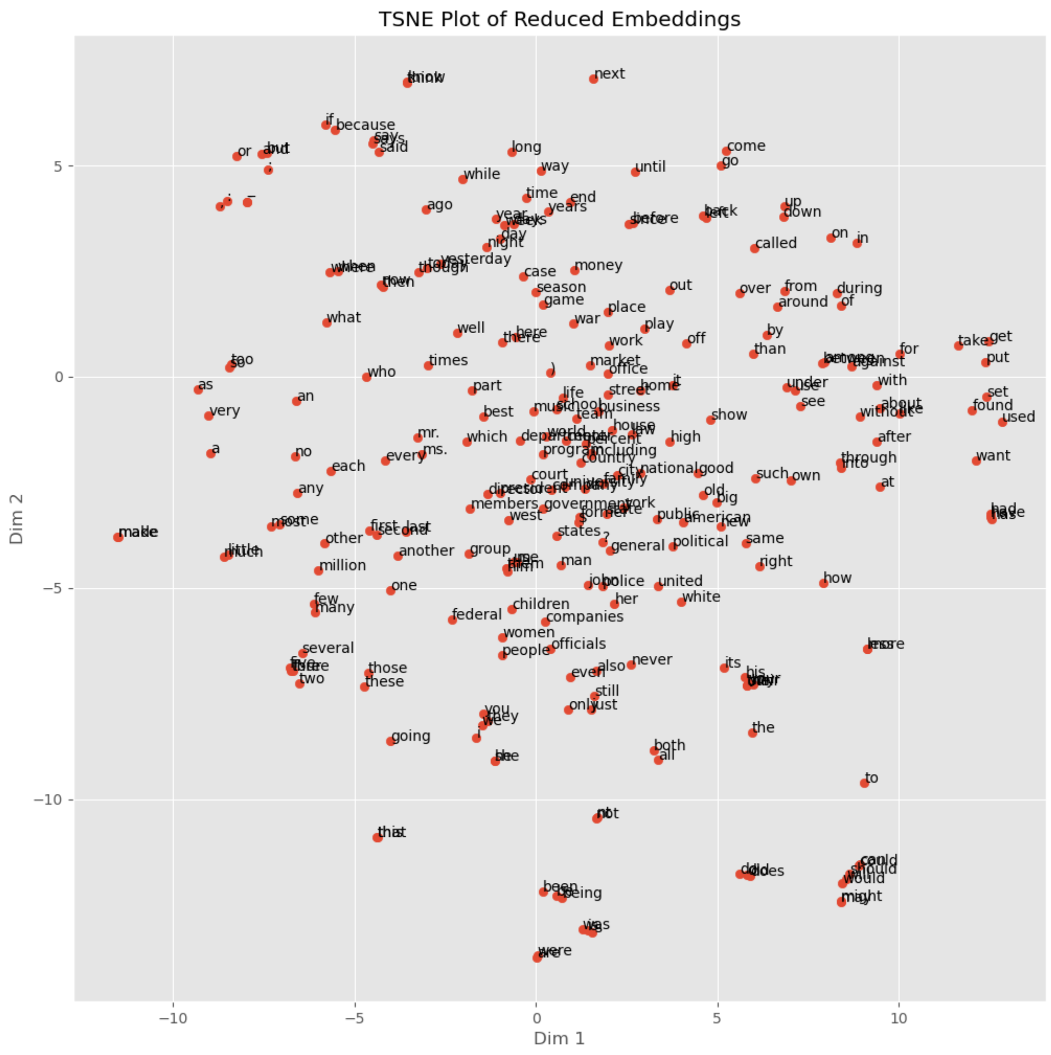


Figure 5: TSN-e plot of the embeddings.

At the right bottom, there exists a cluster of auxiliary verbs such as might, could etc. Also, at the bottom middle it is possible to see the cluster of different forms of to be verb. At the right, there is a cluster if verbs and around (-5,-5) a cluster of numerical quantities can be found. Another cluster is around (-8,4) which includes the punctuation marks. Just on the right of it about (0,5) there is a cluster of words about the time such as ago , year , day etc. When the middle of the plot is investigated, there seems like a mess but it is possible to catch close words with close meanings. As a result it is possible to say that the embeddings obtained from the model makes sense.

When the prediction of the model on ”city of new”, ”life in the”, ”he is the” is checked following results are obtained.

”city of new york” with prob: 0.9917

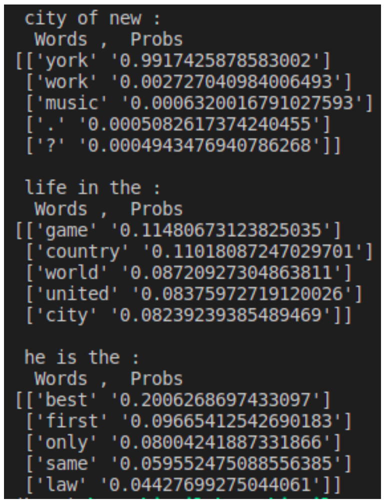
which absolutely makes sense.

*”life in the game” with prob:*0.1148

which also makes sense however not obvious as previous example. But it appears that there is a movie called ”Life in the game” and the model can be considered successful int this task.

”he is the best” with prob: 0.2006

which could be a answer from a human being.
Therefore it is possible to conclude that the model is doing a good job on all of these examples. Following figure also shows the top 5 predictions and their probabilities to these sentences.



```
 city of new :
  Words ,  Probs
[['york' '0.9917425878583002']
 ['work' '0.002727040984006493']
 ['music' '0.0006320016791027593']
 ['.' '0.0005082617374240455']
 ['?' '0.0004943476940786268']]

 life in the :
  Words ,  Probs
[['game' '0.11480673123825035']
 ['country' '0.11018087247029701']
 ['world' '0.08720927304863811']
 ['united' '0.08375972719120026']
 ['city' '0.08239239385489469']]

 he is the :
  Words ,  Probs
[['best' '0.2006268697433097']
 ['first' '0.09665412542690183']
 ['only' '0.08004241887331866']
 ['same' '0.059552475088556385']
 ['law' '0.04427699275044061']]
```

Figure 6: Top 5 predictions and probabilities to examples

This table also shows that when the there is not much options that make sense model is confident and gives high probability to the best prediction. When it gives low probability it appears that there are also other options that may make sense.

5 Conclusion

In this project a deep language model with given architecture that takes three words as input and predicts the fourth word is implemented. After the implementation batch size and learning rate is tuned and the model with selected hyperparametrs is evaluated. The model obtained 0.3674 accuracy on the test set similar to its validation accuracy. Even though the accuracy is low, when the embeddings are plotted in 2D, it is possible to see that meaning clusters occur. Also for the example inputs the model gives sensible results. Given these information model is considered to be a good model for the given architecture.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] Michael A. Nielsen. Neural networks and deep learning, 2018.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

6 Appendix

6.1 Appendix A (Codes)

network.py

```python
import numpy as np
import pickle

def softmax(X):
    shifted_exp = np.exp(X.T - np.max(X, axis=1))
    return (shifted_exp / np.sum(shifted_exp, axis=0)).T


def softmax_prime(output):
    grad_start = np.einsum('ki, kj->kij', output, output)
    identity = np.eye(grad_start.shape[1])
    identities = np.stack([identity]*grad_start.shape[0])
    output_matrix = np.stack([output]*output.shape[1], axis=2)
    return np.multiply(output_matrix, identities) - grad_start


def sigmoid(X):
    return 1/(1+np.exp(-X))


def sigmoid_prime(output):
    return np.multiply(output, (1-output))


def cross_entropy_loss(X,y):
    return -np.sum(np.multiply(y, np.log(X)))/y.shape[0]


def cross_entropy_prime(X,y):
    return -y/(X)


def encode_words(indice_matrix, n_words=250):
    indice_matrix = indice_matrix.T
    if indice_matrix.shape[0] == 1:
        return (np.arange(n_words) == indice_matrix[...,None]).astype(int)[0]
    return (np.arange(n_words) == indice_matrix[...,None]).astype(int)


class Network:
    def __init__(self, dictionary_size, embedding_size, hidden_layer_size, nwords=3,
        sigma=0.01, init_weights=True):
        self.dict_size = dictionary_size
        self.e_size = embedding_size
        self.h_size = hidden_layer_size
        self.n_words = nwords

        if init_weights:
            np.random.seed(3136)
            self.W1 = np.random.normal(scale=sigma,size=(dictionary_size, embedding_size)
                )

            self.W2 = np.random.normal(scale=sigma, size=(embedding_size*nwords,
                hidden_layer_size))
            self.bias1 = np.random.normal(scale=sigma, size=(1,hidden_layer_size))

            self.W3 = np.random.normal(scale=sigma,size=(hidden_layer_size,
                dictionary_size))
            self.bias2 = np.random.normal(scale=sigma, size=(1,dictionary_size))

    def forward(self,words):

        self.word1, self.word2, self.word3 = words

        self.e1 = np.matmul(self.word1, self.W1)
        self.e2 = np.matmul(self.word2, self.W1)
        self.e3 = np.matmul(self.word3, self.W1)

        self.e = np.concatenate([self.e1, self.e2, self.e3], axis=1)

        self.h = np.matmul(self.e, self.W2) + self.bias1
        self.h = sigmoid(self.h)

        self.output = np.matmul(self.h, self.W3) + self.bias2
        self.output = softmax(self.output)

        return self.output

    def backward(self, y):
        X = self.output
        batch_size = X.shape[0]

        #self.loss_grad = cross_entropy_prime(X, y) #bs x 250
        #self.softmax_grad = softmax_prime(X) # 250 x 250
        #self.delta_3 = np.einsum("ki,kij->kj", self.loss_grad, self.softmax_grad) # bs
            x 250

        # To do things faster we can benefit from CE softmax derivative relation
        delta_3 = X-y

        self.W3_grad = np.matmul(delta_3.T, self.h).T / batch_size
        self.bias2_grad = np.mean(delta_3, axis=0)

        sigmoid_grad = sigmoid_prime(self.h)
        delta_2 = np.multiply(np.dot(delta_3, self.W3.T), sigmoid_grad)

        self.W2_grad = np.matmul(delta_2.T, self.e).T / batch_size
        self.bias1_grad = np.mean(delta_2, axis=0)


        delta_1 = np.matmul(delta_2, self.W2.T)

        delta_1_1 = delta_1[:,:self.e_size]
        delta_1_2 = delta_1[:,self.e_size:self.e_size*2 ]
        delta_1_3 = delta_1[:,self.e_size*2: ]


        W1_grad1 = np.matmul(delta_1_1.T, self.word1).T / batch_size

        W1_grad2 = np.matmul(delta_1_2.T, self.word2).T / batch_size

        W1_grad3 = np.matmul(delta_1_3.T, self.word3).T / batch_size

        self.W1_grad = W1_grad1 + W1_grad2 + W1_grad3

    def save_network(self, save_path):

        parameters = {}
        parameters["W1"] = self.W1

        parameters["W2"] = self.W2

        parameters["W3"] = self.W3

        parameters["bias2"] = self.bias2
        parameters["bias1"] = self.bias1

        with open(save_path, "wb") as file:
            pickle.dump(parameters, file)

    @classmethod
    def load_network(cls, model_path):

        with open(model_path, "rb") as file:
            parameters = pickle.load(file)

        cls.W1 = parameters["W1"]

        cls.W2 = parameters["W2"]

        cls.W3 = parameters["W3"]

        cls.bias2 = parameters["bias2"]
        cls.bias1 = parameters["bias1"]

        dict_size = parameters["W1"].shape[0]
        embedding_size = parameters["W1"].shape[1]
        hidden_size = parameters["W2"].shape[1]
        n = int(parameters["W2"].shape[0]/embedding_size)

        return cls(dict_size, embedding_size, hidden_size, nwords=n, init_weights=False)


    def SGD_step(self, lr=0.01):
        self.W1 -= lr*self.W1_grad
```

```python
        self.W2 -= lr*self.W2_grad
        self.W3 -= lr*self.W3_grad

        self.bias1 -= lr*self.bias1_grad
        self.bias2 -= lr*self.bias2_grad
```

train.py

```python
import network as nn
import matplotlib.pyplot as plt
import numpy as np
import pickle
from time import time
import os

BATCH_SIZES = (16, 128, 512)
MAX_EPOCHS = 250
LRs= (0.1, 0.01,1)

checkpoint_paths = {}
max_no_improvement = 8

X_train = np.load('./data/train_inputs.npy')
y_train = np.load('./data/train_targets.npy').reshape(-1,1)

X_val = np.load('./data/valid_inputs.npy')
y_val = np.load('./data/valid_targets.npy').reshape(-1,1)

indices = list(range(X_train.shape[0]))
np.random.seed(3136)
np.random.shuffle(indices)
X_train = X_train[indices]
y_train = y_train[indices]

best_models = {}
for batch_size in BATCH_SIZES:
    for lr in LRs:
        save_name = f"bs_{batch_size}_lr_{lr}"
        print(f"Training for parameters: Batch Size:{batch_size}, lr: {lr}")

        n_batches = X_train.shape[0]//batch_size + ((X_train.shape[0]%batch_size !=0))

        if not checkpoint_paths.get(save_name) is None:
            network= nn.Network.load_network(checkpoint_paths.get(save_name))
            print("Checkpoint Loaded...")
        else:
            network = nn.Network(250,16,128)

        X_valid = nn.encode_words(X_val)
        y_valid = nn.encode_words(y_val)

        early_stop = 0
        best_acc = 0
        train_losses = []
        validation_losses = []
        train_accuracy = []
        validation_accuracy = []
        for epoch in range(MAX_EPOCHS):
            start_time = time()
            total_loss = 0
            correct_words = 0
            for i in range(n_batches):
                start = i*batch_size
                end = (i+1)*batch_size

                X_batch = X_train[start:end]
                X_batch = nn.encode_words(X_batch)

                y_batch = y_train[start:end]
                y_batch = nn.encode_words(y_batch)

                y_pred = network.forward(X_batch)
                network.backward(y_batch)

                network.SGD_step(lr)

                batch_loss = nn.cross_entropy_loss(y_pred, y_batch)
                total_loss += batch_loss
                correct_words += sum(np.argmax(y_pred, axis=1) == np.argmax(y_batch, axis
                    =1))


                if (i % 1000) == 0:
                    print(f"Epoch: {epoch}, Batch: {i} Train loss: {batch_loss}")

            train_acc = correct_words/X_train.shape[0]
            train_accuracy.append(train_acc)
            train_losses.append(total_loss/n_batches)

            print(f"Time: {time()-start_time} Validating Epoch {epoch}...")
            y_pred = network.forward(X_valid)
            loss = nn.cross_entropy_loss(y_pred, y_valid)
            validation_losses.append(loss)
            correct = sum(np.argmax(y_pred, axis=1) == np.argmax(y_valid, axis=1))
            test_acc = correct/y_valid.shape[0]
            validation_accuracy.append(test_acc)
            print(f"Epoch {epoch}: Train loss: {train_losses[-1]}, Train accuracy:{
                train_acc}, Valid loss: {loss}, Valid accuracy: {test_acc}")
            if not os.path.exists(f'./models/{save_name}/'):
                os.makedirs(f"./models/{save_name}")
            network.save_network(f'./models/{save_name}/Epoch{epoch}.pkl')
            early_stop +=1
            if test_acc >= best_acc:
                early_stop = 0
                best_acc = test_acc
                network.save_network(f'./models/{save_name}/BestModel.pkl')
            if early_stop > max_no_improvement:
                break

        metrics = {}
        metrics["train_accuracy"] = train_accuracy
        metrics["validation_accuracy"] = validation_accuracy
        metrics["train_loss"] = train_losses
        metrics["validation_loss"] = validation_losses
        metrics["batch_size"] = batch_size
        metrics["learning_rate"] = lr

        with open(f"./models/summary_{save_name}.pkl", "wb") as file:
            pickle.dump(metrics, file)

        plt.style.use("ggplot")
        plt.figure(figsize=(12,8))
        plt.plot(list(range(len(train_accuracy))), train_accuracy, label='Train Accuracy
            ')
        plt.plot(list(range(len(validation_accuracy))), validation_accuracy, label='
            Validation Accuracy')
        plt.xlabel("# of Epoch")
        plt.ylabel("Accuracy")
        plt.legend()
        plt.savefig(f"./models/{save_name}/accuracy_plot.png")

        plt.figure(figsize=(12,8))
        plt.plot(list(range(len(train_losses))), train_losses, label='Train Loss')
        plt.plot(list(range(len(validation_losses))), validation_losses, label='
            Validation Loss')
        plt.xlabel("# of Epoch")
        plt.ylabel("CE Loss")
        plt.legend()
        plt.savefig(f"./models/{save_name}/loss_plot.png")

        best_models[save_name] = best_acc

with open("./models/best_models.pkl", "wb") as file:
    pickle.dump(best_models, file)
print(best_models)
print("DONE")
```

eval.py

```
import network as nn
import numpy as np

MODEL_PATH = "./model.pkl"

X_train = np.load('./data/train_inputs.npy')
X_train = nn.encode_words(X_train)
y_train = np.load('./data/train_targets.npy').reshape(-1,1)
y_train = nn.encode_words(y_train)

X_val = np.load('./data/valid_inputs.npy')
X_val = nn.encode_words(X_val)
y_val = np.load('./data/valid_targets.npy').reshape(-1,1)
y_val = nn.encode_words(y_val)

```

```python
X_test = np.load('./data/test_inputs.npy')
X_test = nn.encode_words(X_test)
y_test = np.load('./data/test_targets.npy').reshape(-1,1)
y_test = nn.encode_words(y_test)

vocab = list(np.load('./data/vocab.npy'))

network = nn.Network.load_network(MODEL_PATH)

city_of_new = 'city of new'
city_of_new_idx = [vocab.index(x) for x in city_of_new.split(' ')]

life_in_the = 'life in the'
life_in_the_idx = [vocab.index(x) for x in life_in_the.split(' ')]

he_is_the = 'he is the'
he_is_the_idx = [vocab.index(x) for x in he_is_the.split(' ')]

train_pred = network.forward(X_train)
valid_pred = network.forward(X_val)
test_pred = network.forward(X_test)

train_loss = nn.cross_entropy_loss(train_pred, y_train)
valid_loss = nn.cross_entropy_loss(valid_pred, y_val)
test_loss = nn.cross_entropy_loss(test_pred, y_test)

train_acc = sum(np.argmax(train_pred, axis=1) == np.argmax(y_train, axis=1))/y_train.
    shape[0]
valid_acc = sum(np.argmax(valid_pred, axis=1) == np.argmax(y_val, axis=1))/y_val.shape
    [0]
test_acc = sum(np.argmax(test_pred, axis=1) == np.argmax(y_test, axis=1))/y_test.shape
    [0]

batch = np.array([city_of_new_idx, life_in_the_idx, he_is_the_idx])
encoded_batch = nn.encode_words(batch)
pred_batch = network.forward(encoded_batch)
indices = np.argmax(pred_batch, axis=1)
words = [vocab[i] for i in indices]

print(f'Train Loss: {train_loss},   Train Accuracy: {train_acc}')
print(f'Validation Loss: {valid_loss}, Validation Accuracy: {valid_acc}')
print(f'Test Loss: {test_loss},      Test Accuracy: {test_acc}')

print("Predictions to examples:")
print(f"   {city_of_new} {words[0]}")
print(f"   {life_in_the} {words[1]}")
print(f"   {he_is_the} {words[2]}")
```

tsne.py

```python
import network as nn
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

```

```python
MODEL_PATH = "./model.pkl"

vocab = list(np.load('./data/vocab.npy'))
network = nn.Network.load_network(MODEL_PATH)

encoded_words = np.identity(len(vocab))
embeddings = np.matmul(encoded_words, network.W1)

tsne_model = TSNE(perplexity=40, n_components=2, init='pca', n_iter=2500, random_state
    =3136)
reduced_embeddings = tsne_model.fit_transform(embeddings)

x = reduced_embeddings[:,0]
y = reduced_embeddings[:,1]

plt.style.use("ggplot")
plt.figure(figsize=(12,12))
plt.scatter(x, y)
for label, xi, yi in zip(vocab, x, y):
    plt.annotate(label, xy=(xi,yi), xytext=(0, 0), textcoords='offset points')

plt.xlabel("Dim 1")
plt.ylabel("Dim 2")
plt.title("TSNE Plot of Reduced Embeddings")

plt.savefig('./tsne_plot.png')
print("Plot saved to directory")
```