Deep Learning, DD2424

Report on lab assignment 3

Convolutional Neural Network

Majd Jamal

May 5, 2021

1 Main objectives and scope of the assignment

My major goals in the assignment were

- to implement and train a convolutions neural network.
- use deep learning for solving natural language processing problems.
- to find a solution for unbalanced data set and use it for training.

I implemented a convolutional neural network from scratch using only NumPy. The architecture is based on two convolutional layers and one fully connected layer, with activation functions ReLU and Softmax. Moreover, I found a solution for training with unbalanced data.

2 Method

The network was built from scratch using Python 3 and packages: NumPy and Matplotlib. The code and plots are found in Appendix Code.

3 Result

(i) Gradients check

After implementing the required functions, I conducted three experiments to compare the analytic gradients with the numerical ones. The code for these experiments is shown in Appendix Code analyzeGradient(), on page 22. The absolutes difference between numerical and analytic gradients was small, i.e., ($< 1e-6$).

Few data points & small layer sizes

With 5 data points and parameters **n1 = 5, n2 = 5, k1 = 5, k2 = 3**, the difference was:

Weight (fully connected layer) Gradient: 9.88e-10

Filter 2 (second convolutional layer) Gradient: 5.80e-11

Filter 1 (first convolutional layer) Gradient: 7.33e-11

Few data points & bigger layers

With **5 data points** and parameters **n1 = 20, n2 = 20, k1 = 5, k2 = 3**, the difference was:

Weight Gradient: 2.19e-10

Filter 2 Gradient: 5.99e-11

Filter 1 Gradient: 5.57e-11

More data points & bigger layers

With **100 data points** and parameters **n1 = 20, n2 = 20, k1 = 5, k2 = 3**, the difference was:

Weight Gradient: 2.54e-09

Filter 2 Gradient: 1.23e-09

Filter 1 Gradient: 3.18e-09

(ii) Compensating for unbalanced data

I accounted for unbalanced data by selecting six random points from each class. This method was similar to what the teacher used, as stated in the instruction paper.

(iii) Performance

The training data consists of 17204 data points. With a batch size of 108, it is required at least 125 epochs to compute $\geq$ 20000 update steps.

With unbalanced data

Settings for the training were:

n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 100

Score: 5.56%

The confusion matrix on the validation set is found in Appendix Figure 2. Moving Average Loss function on the validation set is found in Appendix Figure 3. Original Loss function on the validation set is found in Appendix Figure 4.

Comment: Due to learning with momentum, the loss curve oscillated, which leads to a disturbing pattern. I created a loss curve with 100 points moving average to increase interpretability.

With balanced data

Settings for the training were:

n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 108

Score: 41.2%

The confusion matrix on the validation set is found in Appendix Figure 5. Loss function on the validation set is found in Appendix Figure 6.

(iv) Best network

The intuition was to use the same setting as previous experiments but stops when the validation loss starts increasing, avoiding overfitting. This happens after about three epochs, as seen in figure 6. The settings was:

n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 3, n_batches = 108

, and the score was:
Score: 47.2%

The confusion matrix on the validation set is found in Appendix Figure 7. Loss function on the validation set are found in Appendix Figure 8.

(v) Compensating for unbalanced data

Q. State whether you implemented the efficiency gains in Background 5 .

I did not have time to improve my training speed, but I will improve it after the submission! However, I tried to pre-compute the first M_x-layer, but my computer stopped the computations because of memory issues. The computer used for this project has 8GB RAM, and the terminal broke after 3000/17402 computation, stating ”zsh killed”.

(vi) Testing

The names that were used for the prediction test was: [Linda, Per, Majd, Alba, Steve] , and the classifications were,

Linda: [German 27.8%, Polish 14.7%, Japanese 11.9%, Italian 10.9%, Czech 10.2%]

Per: [Chinese 58.7%, Korean 9.6%, French 7.5%, Czech 5.4%, Vietnamese 4.6%]

Majd: [Scottish 75.9%, Polish 6.9%, Czech 4.8%, Arabic 4.2%, German 2.5%]

Alba: [Spanish 23.7%, Italian 20.7%, Korean 18.7%, Irish 13.2%, Portuguese 4.47%]

Steve: [Italian 35.3%, Czech 15.9%, French 13.7%, Spanish 11.3%, German 8.8%]

4 Discussion

The performance was very poor when training with unbalanced data, and this was expected. We can clearly see by looking at the confusion matrix in Appendix Figure 2 that the network learned the dominant class, which is label 15. See figure 1 for a distribution of the labels.

The performance became much better when accounting for the unbalanced data. As seen in Figure 5, confusion matrix shows a colored diagonal, indicating that predicted labels matched true labels. The score also increased from 5% to 47.2%.

When testing the network with five names, it was found that classifications for Linda, Alba, and Steve were correct. For example, Linda could indeed be a german woman or polish. Alba is a Latin name used in Spanish and Italian-speaking countries.

This assignment showed that it is important to account for unbalanced data when training a deep network. For further developments and iterations, I will make my program much faster, because it took a very long time to train the network with 120-130 epochs, approximately six hours. An explanation for this slow training speed is that I used Kron multiplications when generating the MX-matrix.

5 Appendix

5.1 Result

(iii)

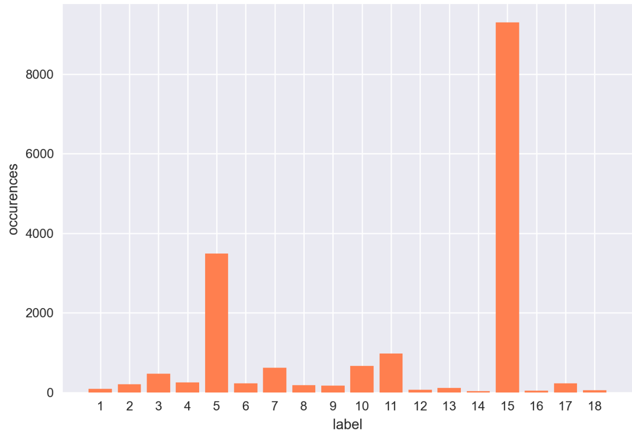

Figure 1: **Title:** Distribution of labels. **Comment:** We can see that label nr. 15 occurs more than the other classes.

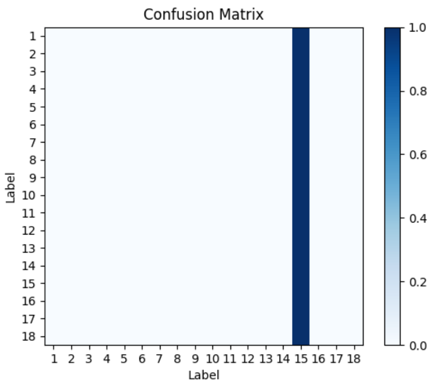

Figure 2: **Title:** Confusion Matrix when training the network with unbalanced data. X-axis shows true labels and Y-axis shows predicted labels. Settings were: n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 100. **Comment:** We can clearly see that all the data points were classified as label 15, which is the dominant label in the distribution.

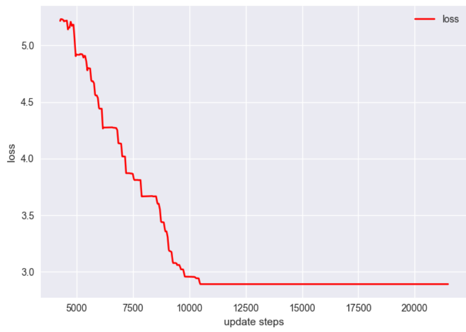

Figure 3: **Title:** 100 Moving Average Loss function on the validation set when training with unbalanced data. **Settings were:** n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 100. **Comment:** The loss function decreased steadily, indicating that the backpropagation algorithm is written correctly. However, the loss function eventually converged to approx 2.89 before 10 000 update steps, indicating that the network learned to classify points only as the dominant class.

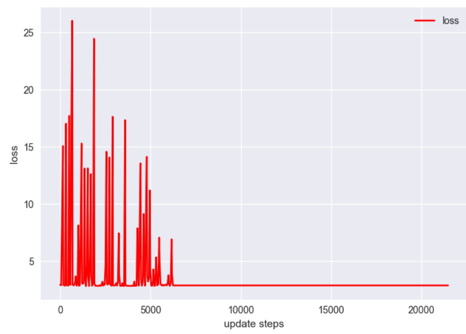

Figure 4: **Title:** Loss function on the validation set when training with unbalanced data. **Settings were:** n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 100. **Comment:** The loss function oscillated heaviliy because of momentum learning. However, the curve had a downward moving pattern, indicating that the backpropagation was working properly.

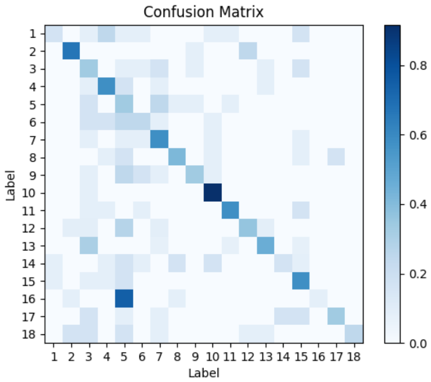

Figure 5: **Title:** Confusion Matrix when training the network with balanced data. X-axis shows true labels and Y-axis shows predicted labels. Settings were: n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 108. **Comment:** We can see a diagonal on the confusion matrix, indicating that predictions were correct.

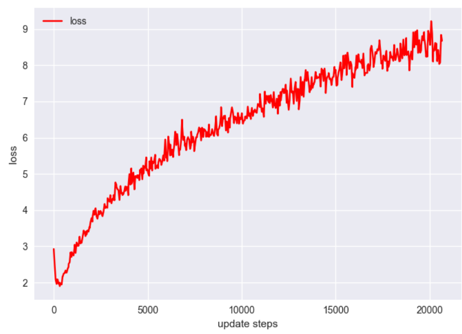

Figure 6: **Title:** 100 Loss function on the validation set when training with balanced data. **Settings were:** n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 130, n_batches = 108. **Comment:** The validation loss function decreased in the beginning, but started to increase after 200-300 update steps, because the network began overfitting.

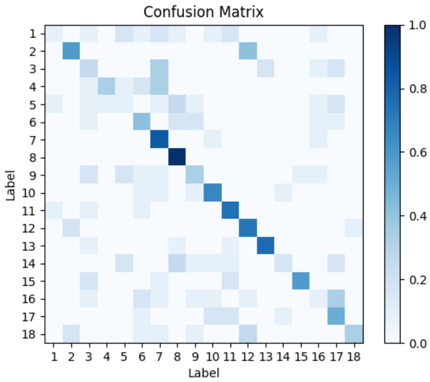

Figure 7: **Title:** Confusion Matrix when training the best network. X-axis shows true labels and Y-axis shows predicted labels. Settings were: n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 3, n_batches = 108.**Comment:** We can clearly see a diagonal, meaning that many points were classified as their true class.

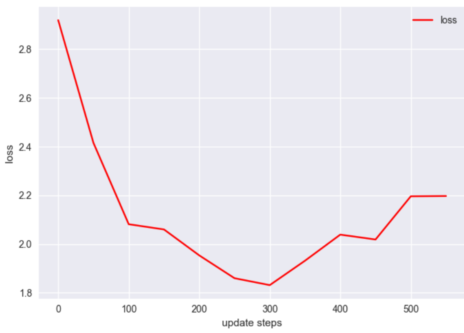

Figure 8: **Title:** Loss function on the validation set from training the best network. Settings were: n1 = 20, n2 = 20, k1 = 5, k2 = 3, eta = 0.001, roh = 0.9, epochs = 3, n_batches = 108.

5.2 Code

```
__author__ = "Majd Jamal"

import numpy as np
import matplotlib.pyplot as plt
import re


##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
## Load data and seperate it into
## names (string) and labels (int)
##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
file = 'dataset/ascii_names.txt'
fid = np.loadtxt(file,delimiter='\t', dtype=str)
Npts = fid.size

names = np.zeros(Npts).astype('str')
ys = np.zeros(Npts)

for i in range(Npts):
    name, label = fid[i].split(' ')

    names[i] = name.lower()
    ys[i] = label


##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
## Measure:
##      number of unique unique characters
##      length of the longest word
##      number of classes
##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
doc = open(file, 'r').read().lower().replace("\n", "").replace(" ", "")

longest_word = max(names, key = len)    # Get longest string
characters = re.sub(r'\d+', '', doc) # Remove digits
unique_characters = ''.join(set(characters))    # Extract unique characters

NUnique = len(unique_characters) #Number of unique characters
                                 #in the document
NLongest = len(longest_word)     #lenght of longest name
NClasses = np.max(ys).astype(int)  #number of classes


##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
## Create a lexicon of char2ind, i.e,
## character to index-notation
##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

```python
char2ind = {}

for i in range(NUnique):
    char2ind[unique_characters[i]] = i


##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
##  Convert names to a matrix representation
##  i.e, the data point matrix X with shape = (Ndim, Npts)
##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
names2matrix = np.zeros((NUnique * NLongest, Npts)) #also known as, X

for i in range(Npts):
    name = names[i]
    name2vec = np.zeros((NUnique, NLongest))

    for j in range(len(name)):

        curr_char = name[j]
        ind = char2ind[curr_char]

        name2vec[ind][j] = 1

    if i == 0:
        xoriginal = name2vec
        xoriginal_flatten = name2vec.flatten(order = 'F')
        np.save('xoriginal.npy', xoriginal)
        np.save('xoriginal_flatten.npy', xoriginal_flatten)

    names2matrix[:, i] = name2vec.flatten(order = 'F')


##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
##  Creating one hot vector matrix
##  with shape = (Nclasses, Npts)
##=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Y = np.zeros((ys.size, ys.max().astype(int)+1))
Y[np.arange(ys.size), np.reshape(ys, (1,-1)).astype(int)] = 1
Y = Y.T
Y = np.delete(Y, 0, 0)


##=-=-=-=-=-=-=-=-=-=-=
##  Save files
##=-=-=-=-=-=-=-=-=-=-=
np.save('final/names.npy', names)
np.save('final/X.npy', names2matrix)
np.save('final/Y.npy', Y)
np.save('final/ys.npy', ys)
np.save('final/dims.npy', np.array([NUnique, NLongest, NClasses]))
```

```python


##=-=-=-=-=-=-=-=-=-=-=
##  Utils
##=-=-=-=-=-=-=-=-=-=-=

class Data:
    """Object to store data
    """
    def __init__(self,
    X_train, Y_train, y_train,
    X_val, Y_val, y_val,
    X, Y, y, names,
    NUnique, NLongest, NClasses):

        self.X_train = X_train
        self.Y_train = Y_train
        self.y_train = y_train

        self.X_val = X_val
        self.Y_val = Y_val
        self.y_val = y_val

        self.X = X
        self.Y = Y
        self.y = y
        self.names = names

        self.NUnique = NUnique
        self.NLongest = NLongest
        self.NClasses = NClasses

def getData():
    """Load and seperate data into
    a training and validation set, and
    return an object of the data.
    """
    X = np.load('data/final/X.npy')
    Y = np.load('data/final/Y.npy')
    y = np.load('data/final/ys.npy')

    names = np.load('data/final/names.npy')

    dims = np.load('data/final/dims.npy')

    all_indices = np.arange(X.shape[1])
    validation_indicies = np.loadtxt('data/dataset/Validation_Inds.txt').astype(int)
    training_indicies = np.delete(all_indices, validation_indicies)


```

```
    data = Data(
    X_train = X[:, training_indicies] ,
    Y_train = Y[:, training_indicies],
    y_train = y[training_indicies],
    X_val = X[:, validation_indicies],
    Y_val = Y[:, validation_indicies],
    y_val = y[validation_indicies],
    X = X,
    Y = Y,
    y = y,
    names = names,
    NUnique = dims[0],
    NLongest = dims[1],
    NClasses = dims[2],
)

    return data


class Params:
        """ Object that are used to pass arguments to the deep network.
        """
        def __init__(self, n1, n2, k1, k2, eta, roh, epochs, n_batches):

                self.n1 = n1
                self.n2 = n2

                self.k1 = k1
                self.k2 = k2

                self.eta = eta
                self.roh = roh

                self.epochs = epochs
                self.n_batches = n_batches

def softmax(x):
        """ Standard definition of the softmax function """
        e_x = np.exp(x - np.max(x))
        return e_x / np.sum(e_x, axis=0)

def ReLU(x):
    """ Standard definition of the ReLU function """
    return np.maximum(x, 0)

def BatchCreator(j, n_batches):
    """ Generates indices for the mini_batch,
    given an iteration step and number of data points
    in each mini_batch.
    :param j: iteration step
```

```python
    :param n_batches: number of data points in a batch
    :return ind: indicies for the mini batch
    """
    j_start = (j-1)*n_batches + 1
    j_end = j*n_batches + 1
    ind = np.arange(start= j_start, stop=j_end, step=1)
    return ind

def vecX(x, d, nlen):
    """ Vectorizes an x-data point in a similar fashion to Matlab.
        :param x: data point, with shape = (Ndim, )
        :param d: height of the original shape
        :param nlen: width  of the original shape
        :return: vectorized version of data point x
    """
    return x.reshape((d, nlen)).flatten(order = 'F')

def vecF(F):
    """ Vectorizes an a filter in a similar fashion to Matlab.
        :param F: Filter, with shape (#filters, width, height)
        :return: vectorized filter in 1D
    """
    nf,_,_ = F.shape
    for filter in range(nf):

            if filter == 0:
                    F_flattened = F[filter].flatten(order = 'F')
            else:
                    F_flattened = np.hstack((F_flattened, F[filter].flatten(order = 'F')))

    return F_flattened

def plotter(X, Y):
        """ Plotting the loss function.
        :param X: x-coordinates, which are used as update steps
        :param Y: loss values
        """
        plt.style.use('seaborn')
        plt.xlabel('update steps')
        plt.ylabel('loss')
        plt.plot(X, Y, color = 'red', label = 'loss')
        plt.legend()
        plt.savefig('result/loss')
        plt.close()


##=-=-=-=-=-=-=-=-=-=-=
##  Model
##=-=-=-=-=-=-=-=-=-=-=

```

```python

class ConvNet:
    """ Convolutional Neural Network.
    """
    def __init__(self):

        self.nlen = None     # Width of X
        self.dx = None       # Height of X

        self.F1 = None
        self.F2 = None
        self.W = None
        self.MF1 = None
        self.MF2 = None

        self.counts = None  # Count occurences of each label,
                            # e.g. {'0': 400 ...}

        self.nlen1 = None   # Width of S1
        self.preMX = []
        self.losses = [[],[]]    #Array to store validation losses
        self.dL_dX = (0,0,0)     #Memory used for momentum computations,
                                 #dW(t-1), dF2(t-1), dF1(t-1)

    def MakeMFMatrix(self, F, nlen):
        """ Creates the MF-matrix used for convolution operations.
        :params F: A 3D filter with shape = (n_filters, height, width)
        :params nlen: Width of the data points that
        goes through the filter layer.
        :return MF: The MF matrix
        """
        nf,d,k = F.shape

        zero = np.zeros((nf, nlen))

        for filter in range(nf):
            if filter == 0:
                VF = F[filter].flatten(order = 'F')
            else:
                VF = np.vstack((VF, F[filter].flatten(order = 'F')))

        MF = np.zeros(((nlen - k + 1)*nf, nlen*d))

        step = 0
        Nelements = VF[0].size

        for i in range((nlen - k + 1)):

            for j in range(nf):

```

```
                    ind = j + i*nf
                    MF[ind, step:Nelements + step] = VF[j]

                step += d

            return MF

        def MakeMXMatrix(self, x_input, nf, d, k, dx, nlen):
            """ Creates the MX-matrix used for convolution operations.
            :params x_input: A 1D data point with shape = (height*width, )
            :params nf: Number of filters in the layer
            :params d: Height of the filter that are used in the convolutional layer
            :params k: Width of the filter
            :params dx: Original height of x_input
            :params nlen: Original width of x_input
            :return MX: The MX matrix
            """
            x_input = x_input.reshape((dx, nlen), order='F')

            I_nf = np.identity(nf)

            MX = np.zeros((
            (nlen-k+1)*nf,
            (k*nf*d)
            ))

            for i in range(nlen-k+1):

                vec = x_input[:, i:i+k].T.flatten()

                vec = np.kron(I_nf, vec)

                if i == 0:
                    MX = vec

                else:
                    MX = np.vstack((MX, vec))

            return MX

        def forward(self, X, MF1, MF2, W):
            """ Computes the forward pass.
                :param X: Data matrix, shape = (Ndim, Npts)
            :param MF1: Filters for the first convolutional layer
            :param MF2: ilters for the first convolutional layer
            :param W: Weights for the fully connected layer
            :return X1: Values after the first ConvLayer
            :return X2: Values after the second ConvLayer
            :return P: Final predictions, shape = (Nout, Npts)
            """
```

```
        X1 = ReLU(MF1 @ X)
        X2 = ReLU(MF2 @ X1)
        S = W@X2
        P = softmax(S)
        return X1, X2, P

    def backward(self, S1, S2, P, W, XBatch, YBatch):#, ind):
        """ Computes the backward pass.
        :param S1: Values after the first ConvLayer
        :param S2: Values after the second ConvLayer
        :param P:  Final predictions of the network
        :param W:  Weights for the fully connected layer
        :param XBatch: X mini batch
        :param YBatch: Y mini batch
        :return dW: Gradients for the fully connecte layer
        :return dF2: Gradients for the second convolutional layer filters
        :return dF1: Gradients for the first convolutional layer filters
        """
        _, Npts = P.shape

        G = - (YBatch - P)

        dW = 0

        for j in range(Npts):
            g = G[:, j].reshape(-1, 1)
            s2 = S2[:, j].reshape(-1, 1)
            y = YBatch[:, j].argmax()
            #py = (1/self.counts[y]) * (1/18)

            dW += g@s2.T #* py

        dW /= Npts

        G = W.T @ G
        G = G * np.where(S2 > 0, 1, 0)

        nf, _,_ = self.F2.shape
        dF2 = 0

        for j in range(Npts):
            g = G[:, j].reshape(-1, 1)
            x = S1[:, j].reshape(-1, 1)
            mx = self.MakeMXMatrix(x, *self.F2.shape, nf, self.nlen1)
            v = g.T @ mx
            y = YBatch[:, j].argmax()
            #py = (1/self.counts[y]) * (1/18)

            dF2 += v #* py
```

```
        dF2 /= Npts

        G = self.MF2.T @ G
        G = G * np.where(S1 > 0, 1, 0)


        nf,d,k = self.F1.shape

        dF1 = 0
        for j in range(Npts):
            g = G[:, j].reshape(-1, 1)
            x = XBatch[:, j].reshape(-1, 1)
            mx = self.MakeMXMatrix(x, *self.F1.shape, self.dx, self.nlen)
            v = g.T @ mx
            y = YBatch[:, j].argmax()
            #py = (1/self.counts[y]) * (1/18)

            dF1 += v #* py

        dF1 /= Npts

        return dW, dF2, dF1

    def update(self, dW, dF2, dF1, eta, rho):
        """ Updates the weights and filters.
        :param dW: Gradients for the fully connecte layer
        :param dF2: Gradients for the second convolutional layer filters
        :param dF1: Gradients for the first convolutional layer filters
        :param eta: Learning rate
        :param rho: Momentum constant
        """

        nf, d, k = self.F2.shape
        nf1, d1, k1 = self.F1.shape

        # -= gradient * learning rate + previous_gradient * momentum
        self.W -= (dW * eta
                + self.dL_dX[0] * rho)

        self.F2 -= (dF2.reshape((d,k, nf), order='F').transpose([2,0,1]) * eta
        + self.dL_dX[1].reshape((d,k, nf), order='F').transpose([2,0,1]) * rho)

        self.F1 -= (dF1.reshape((d1,k1, nf1), order='F').transpose([2,0,1]) * eta
         + self.dL_dX[2].reshape((d1,k1, nf1), order='F').transpose([2,0,1]) * rho)

    def ComputeCost(self, X, Y, MF1, MF2, W, F1, F2):
        """ Computes the loss function.
        :param X: Data matrix, shape = (Ndim, Npts)
        :param Y: One hot encoded labels, shape =(Nout, Npts)
```

```
        :param MF1: Filter matrix for the first convolutional layer
        :param MF2: Filters for the second convolutional layer
        :param W: Weight for the fully connected layer.
        """
        _,_, P  = self.forward(X, MF1, MF2, W)
        _, Npts = P.shape

        loss = 0

        for j in range(Npts):

            y = Y[:, j]
            p = P[:, j]
            ind = y.argmax()
            #py = (1/self.counts[ind]) * (1/18)
            loss -= np.log(y.T @ p) #* py

        loss /= Npts

        return loss

    def ComputeAccuracy(self, P, y):
        """ Computes a score indicating the network accuracy.
        :param P: Probabilities, shape = (Nout, Npts)
        :param y: labels, shape = (Npts, )
        :return: error rate. low is better.
        """
        out = np.argmax(P, axis=0).reshape(1,-1)
        #np.savetxt('out.txt', out.astype(int))
        return 1 - np.mean(np.where(y==out, 0, 1))

    def ComputeGradsNumSlow(self, X, Y, W, F2, F1, h):
        """ Computes numerical gradients.
        """
        MF2 = self.MakeMFMatrix(F2, self.nlen1)
        MF1 = self.MakeMFMatrix(F1, self.nlen)

        grad_W1 = np.zeros(W.shape)
        for i in range(W.shape[0]):
            for j in range(W.shape[1]):
                W1_try = np.array(W)
                W1_try[i,j] -= h
                c1 = self.ComputeCost(X, Y, MF1, MF2, W1_try, F1, F2)

                W1_try = np.array(W)
                W1_try[i,j] += h
                c2 = self.ComputeCost(X, Y, MF1, MF2, W1_try, F1, F2)

                grad_W1[i,j] = (c2 - c1) / (2 * h)

```

```
        grad_F2 = np.zeros(F2.shape)
        for k in range(F2.shape[0]):
            for i in range(F2.shape[1]):
                for j in range(F2.shape[2]):

                    F2_try = np.array(F2)
                    F2_try[k, i, j] -= h
                    MF2_try = self.MakeMFMatrix(F2_try, self.nlen1)
                    c1 = self.ComputeCost(X, Y, MF1, MF2_try, W, F1, F2)

                    F2_try = np.array(F2)
                    F2_try[k, i, j] += h
                    MF2_try = self.MakeMFMatrix(F2_try, self.nlen1)
                    c2 = self.ComputeCost(X, Y, MF1, MF2_try, W, F1, F2)

                    grad_F2[k, i, j] = (c2 - c1) / (2 * h)

        grad_F1 = np.zeros(F1.shape)
        for k in range(F1.shape[0]):
            for i in range(F1.shape[1]):
                for j in range(F1.shape[2]):

                    F1_try = np.array(F1)
                    F1_try[k, i, j] -= h
                    MF1_try = self.MakeMFMatrix(F1_try, self.nlen)
                    c1 = self.ComputeCost(X, Y, MF1_try, MF2, W, F1, F2)

                    F1_try = np.array(F1)
                    F1_try[k, i, j] += h
                    MF1_try = self.MakeMFMatrix(F1_try, self.nlen)
                    c2 = self.ComputeCost(X, Y, MF1_try, MF2, W, F1, F2)

                    grad_F1[k, i, j] = (c2 - c1) / (2 * h)

        return grad_W1, grad_F2, grad_F1

    def TestMFandMX(self):
        """ This function test if the implementation of
            MX and MF is correct.
        """
        X_test = np.arange(1*6*4) + 1
        X_test = X_test.reshape(1,6,4)
        #print(X_test)
        X_input = X_test.flatten(order = 'F')
        #print(X_input)

        F_test = np.arange(4*6*3) + 1
        F_test = F_test.reshape(4,6,3)
        #print(F_test)

```

```
        #nf,d,k = F_test.shape

        MF_test = self.MakeMFMatrix(F_test, 4)
        #print(MF_test)
        print(X_input)
        MX_test = self.MakeMXMatrix(X_input, *F_test.shape, 6, 4)
        print(MX_test)

        s1 = MF_test @X_input
        s2 = MX_test @ vecF(F_test)

        print(np.all(s1 == s2)) # >>> True

    def debug(self):
        """ This section test if the network functions
        can reproduce vectors from DebugInfo.mat.
        """

        import scipy.io
        d = scipy.io.loadmat('utils/DebugInfo.mat')

        debug_x = d['x_input']
        debug_F = d['F']
        debug_vecF = d['vecF']
        debug_vecS = d['vecS']
        debug_S = d['S']
        debug_X = d['X_input']

        dx, nlen = debug_X.shape
        #print(debug_F.shape)
        debug_F = debug_F.transpose([2,0,1])
        nf, d, k = debug_F.shape

        MF = self.MakeMFMatrix(debug_F, nlen)
        S1 = MF @ debug_x

        print(S1[:, 0] == debug_vecS[:, 0])

        MX = self.MakeMXMatrix(debug_x, d, k, nf, dx, nlen)
        S2 = MX @debug_vecF
        print(S2[:, 0] == debug_vecS[:, 0])

        my_S = S2.reshape(debug_S.shape)

        print(my_S == debug_S)

    def plotLabelDist(self):
        """ Plots label distribution.
        """
        import matplotlib.pyplot as plt
```

```python
        plt.style.use('seaborn')
        plt.bar([str(i) for i in range(1,19)], self.counts, color='#ff7f4f')
        plt.xlabel('label')
        plt.ylabel('occurences')
        plt.show()

    def AnalyzeGradients(self, X, Y):
        """ Computes and prints the difference between
        numerical and analytical gradients.
        :param X: Data matrix, shape = (Ndim, Npts)
        :param Y: One hot matrix, shape = (Nout, Npts)
        """
        XBatch = X[:, :100]
        YBatch = Y[:, :100]

        S1, S2, P = self.forward(XBatch, self.MF1, self.MF2, self.W)

        grad_an, grad_an_F2, grad_an_F1 = self.backward(S1, S2, P, self.W, XBatch, YBatch)


        grad_num, grad_num_F2, grad_num_F1 = self.ComputeGradsNumSlow(XBatch, YBatch, self

        nf, d, k = self.F2.shape
        nf1, d1, k1 = self.F1.shape
        grad_an_F1 = grad_an_F1.reshape((d1,k1, nf1), order='F').transpose([2,0,1])
        grad_an_F2 = grad_an_F2.reshape((d,k, nf), order='F').transpose([2,0,1])

        diff = np.sum(np.abs(np.subtract(grad_an, grad_num)))
        diff /= np.sum(np.add(np.abs(grad_an), np.abs(grad_num)))
        print('W: ', diff)

        diff = np.sum(np.abs(grad_an_F2 - grad_num_F2))
        diff /= np.sum(np.abs(grad_an_F2) + np.abs(grad_num_F2))
        print('F2: ',diff)

        diff = np.sum(np.abs(np.subtract(grad_an_F1, grad_num_F1)))
        diff /= np.sum(np.add(np.abs(grad_an_F1), np.abs(grad_num_F1)))
        print('F1: ',diff)

    def getLoss(self):
        """ Returns the loss function.
        """
        return self.losses

    def getWeights(self):
        """ Returns weights
        """
        return self.F1, self.F2, self.W

    def MakeConfusionMatrix(self, P, true, Nout):
```

```
        """ Creates a confusion matrix of predictions and saves it in result/
        :param P: Final predicted probabilities
        :param true: true labels
        :param Nout: Number of unique classes
        """
        pred = np.argmax(P, axis=0)
        CM = np.zeros((Nout, Nout))
        _, counts = np.unique(true, return_counts = True)
        for i in range(true.size):
            true_y = true[i].astype(int)
            pred_y = pred[i].astype(int)
            CM[true_y, pred_y] += 1 / counts[true_y]

        import matplotlib.pyplot as plt
        plt.title('Confusion Matrix')
        im = plt.imshow(CM, cmap = 'Blues')
        plt.xlabel('Label')
        plt.ylabel('Label')
        plt.yticks(np.arange(18), np.arange(1, 19))
        plt.xticks(np.arange(18), np.arange(1, 19))
        bar = plt.colorbar(im)
        #plt.show()
        plt.savefig('result/CM')
        plt.close()

    def name2vec(self, name):
        """ Create a flattened vector representation of names.
        :param name: string
        :return: Vector representation with shape = (Height*Width, )
        """

        name2vec = np.zeros((self.dx, self.nlen))

        for j in range(len(name)):

            curr_char = name[j]
            ind = self.char2ind.item().get(curr_char)

            name2vec[ind][j] = 1

        return name2vec.flatten(order = 'F')

    def fit(self, data, p):
        """ This function is called to start trining.
        :param data: Object containing training and validation data
        :param p: Object containing parameters used for training, i.e.
        epochs, n_batch, eta, etc.
        """

        ##
```

```
        ##  Data
        ##
        X_train = data.X_train
        Y_train = data.Y_train
        y_train = data.y_train - 1

        X_val = data.X_val
        Y_val = data.Y_val
        y_val = data.y_val - 1

        self.char2ind = np.load('data/final/char2ind.npy', allow_pickle=True)

        ##
        ##  Parameters
        ##
        Ndim, Npts = X_train.shape
        self.dx, self.nlen = data.NUnique, data.NLongest
        Nout, _ = Y_train.shape
        epochs = p.epochs
        n_batches = p.n_batches

        d = data.NUnique            #height of F1
        nlen = data.NLongest        #width of X
        nlen1 = nlen - p.k1 + 1     #width of X1
        nlen2 = nlen1 - p.k2 + 1    #width of X2

        self.nlen = nlen
        self.nlen1 = nlen1

        ##
        ##  Filters & Weights using He-initalization
        ##  source: https://towardsdatascience.com/
        ##  weight-initialization-techniques-in-neural-networks-26c649eb3b78
        ##  F1, F2, W, MF1, and MF2
        ##
        self.F1 = np.random.randn(p.n1, d, p.k1) * np.sqrt(2/d)
        self.F2 = np.random.randn(p.n2, p.n1, p.k2) * np.sqrt(2/(d*p.k1))
        self.W = np.random.randn(Nout, (p.n2 * nlen2)) * np.sqrt(2/(p.n1*p.k2))

        self.MF1 = self.MakeMFMatrix(self.F1, nlen)
        self.MF2 = self.MakeMFMatrix(self.F2, nlen1)
        _, self.counts = np.unique(y_train, return_counts = True)
        self.dL_dX = (np.zeros(self.W.shape),
                      np.zeros(self.F2.size),
                      np.zeros(self.F1.size))

        ##
        ##  Debug
        ##
        #self.TestMFandMX() # Test implementation of MF and MX
```

```python
#self.debug()   # Take the Debug test
#self.AnalyzeGradients(X_train, Y_train) # Analyze gradients.

print('=-=- Settings -=-= \n epochs: ', epochs, ' steps/epoch: , ', round(Npts/n_b
print('=-=- Starting Training -=-=')

indices = [np.where(y_train == cl)[0] for cl in range(Nout)]

for i in range(epochs):
    for j in range(round(Npts/n_batches)):
        XBatch = 0
        YBatch = 0

        ##
        ##  Generate mini batches with equal
        ##  label distribution, i.e. accounting
        ##  for the unbalanced data set.
        ##
        for cls in range(Nout):
            rnd = np.random.randint(low=0, high=indices[cls].size, size=6)
            if cls == 0:
                XBatch = X_train[:, indices[cls][rnd]]
                YBatch = Y_train[:, indices[cls][rnd]]
            else:

                XBatch = np.hstack((XBatch, X_train[:, indices[cls][rnd]]))
                YBatch = np.hstack((YBatch, Y_train[:, indices[cls][rnd]]))


        self.MF1 = self.MakeMFMatrix(self.F1, nlen)
        self.MF2 = self.MakeMFMatrix(self.F2, nlen1)

        S1, S2, P = self.forward(XBatch, self.MF1, self.MF2, self.W)

        gradients = self.backward(S1, S2, P, self.W, XBatch, YBatch)#, ind)

        self.update(*gradients, p.eta, p.roh)
        self.dL_dX = gradients

        # Evaluate the model after 50th update step
        if j % 50 == 0:
            loss = self.ComputeCost(X_val, Y_val, self.MF1, self.MF2, self.W, self
            update_step = i * round(Npts/n_batches) + j
            print('loss: ', loss)
            self.losses[0].append(update_step)
            self.losses[1].append(loss)

    print('Epoch: ', i + 1)
    print('\n')
```

```
        print('=-=- Training Completed -=-=')

        ##
        ##  Network evaluation
        ##
        S1, S2, P = self.forward(X_val, self.MF1, self.MF2, self.W)
        self.MakeConfusionMatrix(P, y_val, Nout)
        acc = self.ComputeAccuracy(P, y_val)
        print('Accuracy: ', acc)
        #"""

    def predict(self, name):
        """ Predicts top 5 labels and their probabilities
            for a data point.
        :param name: string
        :return out: top 5 predicted labels
        :return prob: probabilities for the labels
        """
        X = self.name2vec(name)
        _,_, P = self.forward(X, self.MF1, self.MF2, self.W)
        out = P.argsort(axis= 0)[-5:][::-1]
        prob = P[out]

        return out, prob


##=-=-=-=-=-=-=-=-=-=-=
##  Experiments
##=-=-=-=-=-=-=-=-=-=-=


#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
#   Setup
#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
data = getData()
params = Params(
    n1 = 20, n2  = 20,
    k1 = 5, k2  = 3,
    eta  = 0.001, roh = 0.9,
    epochs = 3, n_batches = 108)

cnn = ConvNet()

#=-=-=-=-=-=-=
#   Training
#=-=-=-=-=-=-=
cnn.fit(data, params)
weights = cnn.getWeights()  #[F1, F2, W]
loss_ind, loss = cnn.getLoss()
plotter(loss_ind, loss)
```

```python

#Save weights
np.save('data/weights/weights.npy', weights)
np.save('loss.npy', [loss_ind, loss])

#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
#   Prediction Test
#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
print('=-=-=-=-=-=-=-=-=-=-=-=-=- Test -=-=-=-=-=-=-=-=-=-=-=-=-=')
test = ['linda', 'per', 'majd', 'alba', 'steve']
for name in test:
    labels, probabilities = cnn.predict(name)
    print('Name: ', name)
    print('lbl ', labels)
    print('pr ', probabilities)
    print('\n')
```