Project # 01

<u>Convolutional Neural Network from Scratch for a Classification problem</u>

Submitted To: Dr. Hasan Sajid

Submitted By: Usman Zaheer

Registration Number: 0000327700

Subject: Deep Learning

Contents

1. Problem Statement:

Write a complete code from scratch for a Convolutional Neural Network.

In this model, convolutional neural network model composed of 2 convolutional layers, 2 max pooling layers and Fully connected/dense layers along with Softmax is implemented from scratch for classifying the blood cells images in the given dataset.

2. Dataset Details:

The data provided for the project is available on the following links:

- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7182702/
- https://data.mendeley.com/public-files/datasets/snkd93bnjr/files/2fc38728-2ae7-4a62-a857-032af82334c3/file_downloaded

a. Cleaning the data:

The data set was checked manually for the corrupt files i.e. The **neutrophil** folder contained a corrupt image which was removed successfully and hence cleaned data was provided to model.

b. Loading the data:

The following code is used to load the data into the model from local directory, folder named **"Dataset"**.Images are stored in X and their labels are stored in Y.

```
FOLDER_NAME = 'Dataset'

folders = listdir(FOLDER_NAME)

number_of_classes = len(folders)

data_X, data_Y = [], []

for i in range(number_of_classes):

  folder = folders[i]

  print('Folder', i+1, '-', folder)

  images = listdir(FOLDER_NAME + '/' + folder)

  # Walk over images

  for image in images:

    path = FOLDER_NAME + '/' + folder + '/' + image

    # Process Image

    raw = cv2.imread(path)
```

```
        gray = cv2.cvtColor(raw, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(gray, (50, 50))
        # Add to data
        data_X.append(img.flatten())
        data_Y.append(i)
# Convert to numpy arrays
X = np.array(data_X)
Y = np.array(data_Y).reshape([-1, 1])
print('Total training examples:', len(X))
print("Total Images", X.shape)
print("Total Labels", Y.shape)
```

c. Size of Data:

The data set is composed of **17092** images of peripheral blood cell. The size of each image is 360x363 pixels. There are **8** different type(classes) of cells in the dataset.

Types of cells are:

- basophil
- eosinophil
- erythroblast
- ig
- lymphocyte
- monocyte
- neutrophil
- platelet

Each folder has different number of images for each category mentioned above. The original image is converted into grayscale image size of 50x50. This has been done to ease the computational process. Before training the model, the total dataset is divided into training and test data set. The training dataset is composed of 80% dataset 13674 images and test dataset is composed of 20% of total dataset 3418 images.

```
data = np.hstack((X,Y))
np.random.shuffle(data)
data_train = round(0.8*len(data))
```

```
data_test = round(data_train+0.2*len(data))

train_x = X[:data_train]

print("Shape of train data",train_x.shape)

test_x = X[data_train:data_test]

print("Shape of test data",test_x.shape)

train_y = Y[:data_train]

test_y = Y[data_train:data_test]

train_data = np.hstack((train_x,train_y))

test_data = np.hstack((test_x,test_y))
```

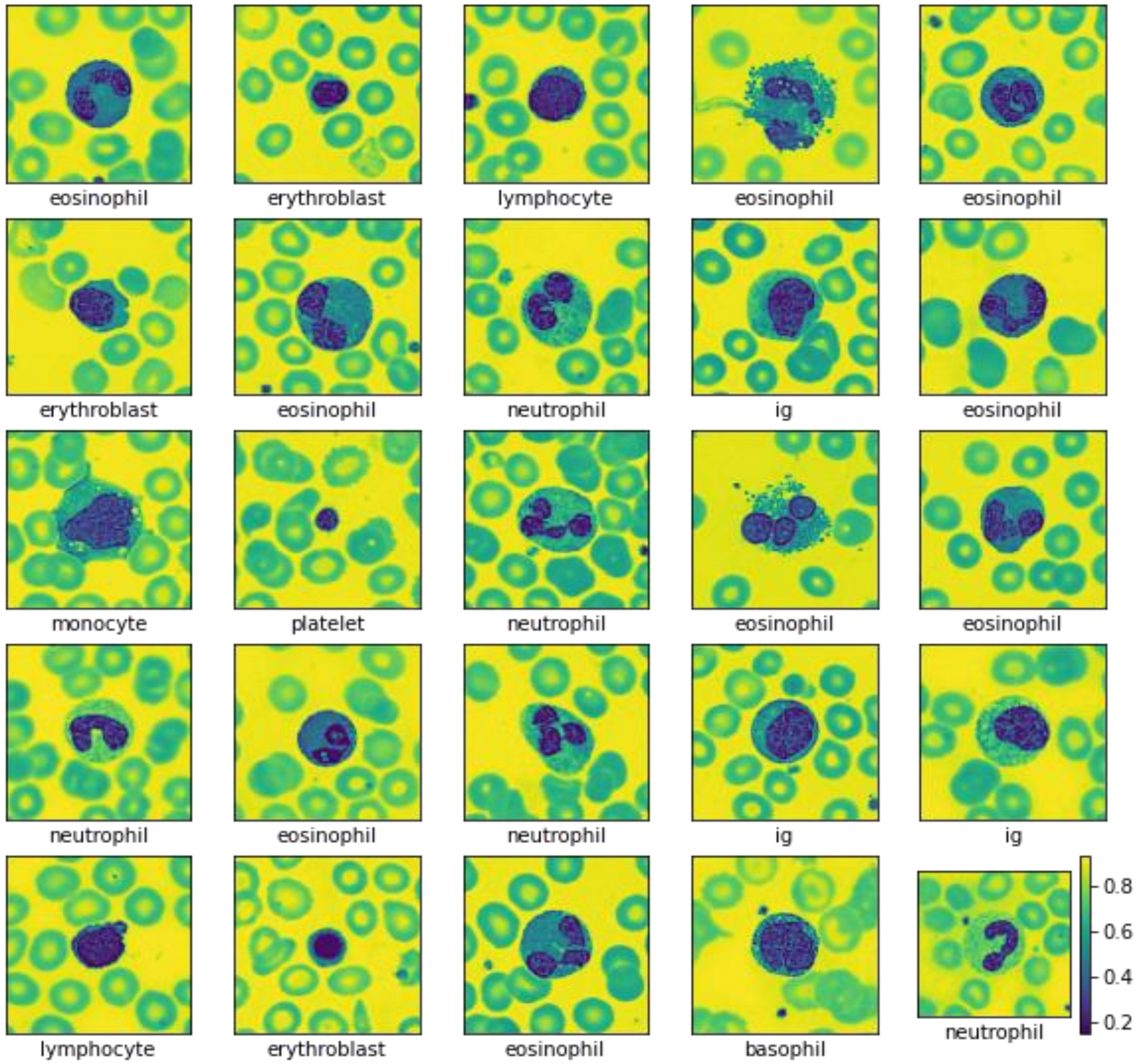


Figure 1: Different type of cell images in Data Set

3. Code and Methodology:

The methodology opted for this model is that in the following steps:

1) Importing useful libraries.
2) Manual Data Cleaning (To remove corrupt files)
3) Loading the data into model from local directory.
4) Assigning images to X and labels to Y.
5) Defining the architecture of convolutionalneural network model.
6) Writing functions for Forward Propagation (Convolutions, ReLU, Maxpooling), Cost Function(Softmax), Cross Entropy Loss, Back Propagations and Adam Gradient Descent for parameters update.

7) Training on training data split and updating parameters.
8) Finding Cost on Training and Test data set.
9) Results Visualization
10) Prediction on based of updated parameters.
11) Checking Accuracy on test data set.

4. Mathematical Model Details:

➢ Architecture of Convolutional Neural Network Model:

The CNN model is composed of **02**convolution layers, **2**maxpooling layer and **2** fully connected dense layers **and 1** output layer.

Convolution1→Maxpool1→Convolution2-→MaxPool2→FullyConnectedLayers→Softmax

Forward Propagation Functions

➢ Convolution Function:

The scratch_convolution function takes 4 inputs i.e.image, filter, bias term and stride value for a filter. The image is reshape into (1,50,50→ matrix and filter (10,1,5,5). The image is convolved with the filter to get the output.

```
def scratch_convolution(image, filter, bias, stride=1):
  (no_of_filters, no_of_channels_f, f, _) = filter.shape
  number_of_channels_image, image_dim, _ = image.shape
  out_dim = int((image_dim - f)/stride)+1
  assert number_of_channels_image == no_of_channels_f
  convolved = np.zeros((no_of_filters,out_dim,out_dim))
  #Moving window over the image
  for curr_filter in range(no_of_filters):
    curr_y = out_y = 0
    while curr_y + f <= image_dim:
      curr_x = out_x = 0
      while curr_x + f <= image_dim:
        convolved[curr_filter, out_y, out_x] = np.sum(filter[curr_filter] * image[:,curr_y:curr_y+f, curr_x:curr_x+f]) + bias[curr_filter]
        curr_x += stride
```

```
            out_x += 1
        curr_y += stride
        out_y += 1
    return convolved
```

The output of the convolution layer is passed through ReLU non-linearity.

```
first_convolution[first_convolution<=0] = 0
```

This process is repeated in the second convolution layer as well using same or different filter size, but the number of channels should be same.

- ### Maxpool Function:

The maxpool function takes 3 inputs i.e. outputs of both convolution layers, filter size and stride value. Maxpool reduces/down sample the size of an image. Window moved over the image and picks the max value from the pixels, where it is overlayed.

```
def scratch_maxpool(image, filter=2, stride=2):
    number_of_channels, h_orig, w_orig = image.shape
    height = int((h_orig - filter)/stride)+1
    width = int((w_orig - filter)/stride)+1
    downsampled = np.zeros((number_of_channels, height, width))
    #Moving window over the image
    for i in range(number_of_channels):
        curr_y = out_y = 0
        while curr_y + filter <= h_orig:
            curr_x = out_x = 0
            while curr_x + filter <= w_orig:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+filter, curr_x:curr_x+filter])
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
```

```
    return downsampled
```

- Fully Connected Layer:

The output of maxpool layer is then flatten to get a fully connected layer(single column vector).

```
((nf2, dim2, _) = pooled2.shape

fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
```

The fully connected layer is then multiplied with the weights term and added with bias terms. The total neurons in first hidden layer are **96.**

```
z = w3.dot(fully_connected) + b3

z[z<=0] = 0

out = w4.dot(z) + b4 # second dense layer.
```

- Softmax Function:

The output of the fully connected layers is passed through Softmax for finding probabilities**.**

```
def scratch_softmax(scores):

    out = np.exp(scores)

    return out/np.sum(out)
```

- Cross Entropy Loss Function:

The output of the softmax is passed through loss function to for finding loss.

```
def scratch_CrossEntropyLoss(probs, label):

    return  np.sum(label * np.log(probs))
```

- Initialization of parameters:

The model parameters like filters, bias terms and weights of fully connected layers are initialized using the following function.

```
def scratch_initializeFilter(size, scale = 1.0):

    stddev = scale/np.sqrt(np.prod(size))

    return np.random.normal(loc = 0, scale = stddev, size = size)

def scratch_initializeWeight(size):

    return np.random.standard_normal(size=size) * 0.01
```

In this model, there are **08** parameters:

f1 --	filters of first layer	(10 filters of size 5x5x1)
f2–	filters of 2nd layer	(10 filters of size 5x5x1
b1 --	bias terms for first convolution	(10, 1)
b2 --	bias terms for first convolution	(10, 1)
b3 --	bias for first hidden layer	(96, 1)
b4 --	bias for 2nd hidden layer	(8, 1)
W3 --	weight matrix in fc layer	(96, 2500)
W4 --	weight matrix in fc layer	(8, 96)

Back Propagation Functions:

Backpropagation is recursive application of the chain rule to compute the gradients of all parameters / intermediates. Back Propagation of convolution and maxpool is given below:

- Backward Convolution Function:

```
def scratch_convolutionBackward(dconv_prev, conv_in, filter, stride):
  (no_of_filters, number_of_channels_f, f, _) = filter.shape
  (_, orig_dim, _) = conv_in.shape
  dout = np.zeros(conv_in.shape)
  dfilter = np.zeros(filter.shape)
  dbias = np.zeros((no_of_filters,1))
  for current_filter in range(no_of_filters):
    curr_y = out_y = 0
    while curr_y + f <= orig_dim:
      curr_x = out_x = 0
      while curr_x + f <= orig_dim:
        dfilter[current_filter] += dconv_prev[current_filter, out_y, out_x] * conv_in[:, curr_y:curr_y+f, curr_x:curr_x+f]
        dout[:, curr_y:curr_y+f, curr_x:curr_x+f] += dconv_prev[current_filter, out_y, out_x] * filter[current_filter]
        curr_x += stride
        out_x += 1
      curr_y += stride
      out_y += 1
```

```
        dbias[current_filter] = np.sum(dconv_prev[current_filter])
    return dout, dfilter, dbias
```

- Backward Maxpool Function:

```
def nanargmax(arr):
    idx = np.nanargmax(arr)
    idxs = np.unravel_index(idx, arr.shape)
    return idxs
def scratch_maxpoolBackward(dpool, orig, f, stride):
    (number_of_channels_f, orig_dim, _) = orig.shape
    dout = np.zeros(orig.shape)
    for curr_c in range(number_of_channels_f):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f, curr_x:curr_x+f])
                dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
    return dout
```

- Forward Propagation in Model:

The forward propagation is used to calculate predictions using combination of Convolutions, ReLU and Maxpooling operations.

```
def scratch_complete_model(image, label, params, conv_stride, pool_filter, pool_stride):
    [f1, f2, w3, w4, b1, b2, b3, b4] = params
    first_convolution = scratch_convolution(image, f1, b1, conv_stride)
    first_convolution[first_convolution<=0] = 0
```

```
pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)
second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)
second_convolution[second_convolution<=0] = 0
pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)
(nf2, dim2, _) = pooled2.shape
fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
z = w3.dot(fully_connected) + b3
z[z<=0] = 0
out = w4.dot(z) + b4
probs = scratch_softmax(out)
loss = scratch_CrossEntropyLoss(probs, label)
```

- Back Propagation in Model:

The backpropagation functions integration in backward propagation.

```
dout = probs - label
dw4 = dout.dot(z.T)
db4 = np.sum(dout, axis = 1).reshape(b4.shape)
dz = w4.T.dot(dout)
dz[z<=0] = 0
dw3 = dz.dot(fully_connected.T)
db3 = np.sum(dz, axis = 1).reshape(b3.shape)
dfully_connected = w3.T.dot(dz)
dpool2 = dfully_connected.reshape(pooled2.shape)
dsecond_convolution = scratch_maxpoolBackward(dpool2, second_convolution, pool_filter, pool_stride)
dsecond_convolution[second_convolution<=0] = 0
dpool1, df2, db2 = scratch_convolutionBackward(dsecond_convolution, pooled1, f2, conv_stride)
dfirst_convolution= scratch_maxpoolBackward(dpool1, first_convolution, pool_filter, pool_stride)
dfirst_convolution[first_convolution<=0] = 0
```

```
  dimage, df1, db1 = scratch_convolutionBackward(dfirst_convolution, image, f1, conv_stride)
  gradients = [df1, df2, dw3, dw4, db1, db2, db3, db4]
  return gradients, loss
```

➢ Complete Convolutional Network Model

This function integrates all the functions required for the forward and backward propagation as per the architecture of the model.

```
def scratch_complete_model(image, label, params, conv_stride, pool_filter, pool_stride):
  [f1, f2, w3, w4, b1, b2, b3, b4] = params
  first_convolution = scratch_convolution(image, f1, b1, conv_stride)
  first_convolution[first_convolution<=0] = 0
  pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)
  second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)
  second_convolution[second_convolution<=0] = 0
  pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)
  (nf2, dim2, _) = pooled2.shape
  fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
  z = w3.dot(fully_connected) + b3
  z[z<=0] = 0
  out = w4.dot(z) + b4
  probs = scratch_softmax(out)
  loss = scratch_CrossEntropyLoss(probs, label)
  dout = probs - label
  dw4 = dout.dot(z.T)
  db4 = np.sum(dout, axis = 1).reshape(b4.shape)
  dz = w4.T.dot(dout)
  dz[z<=0] = 0
  dw3 = dz.dot(fully_connected.T)
  db3 = np.sum(dz, axis = 1).reshape(b3.shape)
  dfully_connected = w3.T.dot(dz)
```

```
  dpool2 = dfully_connected.reshape(pooled2.shape)
  dsecond_convolution = scratch_maxpoolBackward(dpool2, second_convolution, pool_filter, pool_stride)
  dsecond_convolution[second_convolution<=0] = 0
  dpool1, df2, db2 = scratch_convolutionBackward(dsecond_convolution, pooled1, f2, conv_stride)
  dfirst_convolution= scratch_maxpoolBackward(dpool1, first_convolution, pool_filter, pool_stride)
  dfirst_convolution[first_convolution<=0] = 0
  dimage, df1, db1 = scratch_convolutionBackward(dfirst_convolution, image, f1, conv_stride)
  gradients = [df1, df2, dw3, dw4, db1, db2, db3, db4]
  return gradients, loss
```

- ### Adam Gradient Descent:

Adam Gradient Descent is used to find the minimum values of parameters, so that our cost will be minimum. It is the combination of RMSprop and momentum.

```
def scratch_adamGD(batch, no_of_classes, learning_rate, dim, n_c, beta1, beta2, parameters, cost):
  [f1, f2, w3, w4, b1, b2, b3, b4] = parameters
  X = batch[:,0:-1]
  X = X.reshape(len(batch), n_c, dim, dim)
  Y = batch[:,-1]
  cost_ = 0
  batch_size = len(batch)
  df1 = np.zeros(f1.shape)
  df2 = np.zeros(f2.shape)
  dw3 = np.zeros(w3.shape)
  dw4 = np.zeros(w4.shape)
  db1 = np.zeros(b1.shape)
  db2 = np.zeros(b2.shape)
  db3 = np.zeros(b3.shape)
  db4 = np.zeros(b4.shape)
  v1 = np.zeros(f1.shape)
```

```
v2 = np.zeros(f2.shape)
v3 = np.zeros(w3.shape)
v4 = np.zeros(w4.shape)
bv1 = np.zeros(b1.shape)
bv2 = np.zeros(b2.shape)
bv3 = np.zeros(b3.shape)
bv4 = np.zeros(b4.shape)
s1 = np.zeros(f1.shape)
s2 = np.zeros(f2.shape)
s3 = np.zeros(w3.shape)
s4 = np.zeros(w4.shape)
bs1 = np.zeros(b1.shape)
bs2 = np.zeros(b2.shape)
bs3 = np.zeros(b3.shape)
bs4 = np.zeros(b4.shape)
for i in range(batch_size):
  x = X[i]
  y = np.eye(no_of_classes)[int(Y[i])].reshape(no_of_classes, 1) # convert label to one-hot
  grads, loss = scratch_complete_model(x, y, parameters, 1, 2, 2)
  [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads
  df1+=df1_
  db1+=db1_
  df2+=df2_
  db2+=db2_
  dw3+=dw3_
  db3+=db3_
  dw4+=dw4_
  db4+=db4_
```

```
        cost_+= loss
    v1 = beta1*v1 + (1-beta1)*df1/batch_size # momentum update
    s1 = beta2*s1 + (1-beta2)*(df1/batch_size)**2 # RMSProp update
    f1 -= learning_rate * v1/np.sqrt(s1+1e-7) # combine momentum and RMSProp to perform update with Adam
    bv1 = beta1*bv1 + (1-beta1)*db1/batch_size
    bs1 = beta2*bs1 + (1-beta2)*(db1/batch_size)**2
    b1 -= learning_rate * bv1/np.sqrt(bs1+1e-7)
    v2 = beta1*v2 + (1-beta1)*df2/batch_size
    s2 = beta2*s2 + (1-beta2)*(df2/batch_size)**2
    f2 -= learning_rate * v2/np.sqrt(s2+1e-7)
    bv2 = beta1*bv2 + (1-beta1) * db2/batch_size
    bs2 = beta2*bs2 + (1-beta2)*(db2/batch_size)**2
    b2 -= learning_rate * bv2/np.sqrt(bs2+1e-7)
    v3 = beta1*v3 + (1-beta1) * dw3/batch_size
    s3 = beta2*s3 + (1-beta2)*(dw3/batch_size)**2
    w3 -= learning_rate * v3/np.sqrt(s3+1e-7)
    bv3 = beta1*bv3 + (1-beta1) * db3/batch_size
    bs3 = beta2*bs3 + (1-beta2)*(db3/batch_size)**2
    b3 -= learning_rate * bv3/np.sqrt(bs3+1e-7)
    v4 = beta1*v4 + (1-beta1) * dw4/batch_size
    s4 = beta2*s4 + (1-beta2)*(dw4/batch_size)**2
    w4 -= learning_rate * v4 / np.sqrt(s4+1e-7)
    bv4 = beta1*bv4 + (1-beta1)*db4/batch_size
    bs4 = beta2*bs4 + (1-beta2)*(db4/batch_size)**2
    b4 -= learning_rate * bv4 / np.sqrt(bs4+1e-7)
    cost_ = cost_/batch_size
    cost.append(cost_)
```

```
    parameters = [f1, f2, w3, w4, b1, b2, b3, b4]

    return parameters, cost
```

5. Training the model:

Training the model using following optimal parameters:
* Number of classes= 8 (In given dataset)
* Learning Rate= 0.001
* beta1= 0.95 (Variable for AdamGD)
* beta2= 0.99 (Variable for AdamGD)
* img_dim= 50X50 (For ease of computation)
* f= 5X5 (filter size)
* num_filter= 10 (Number of filters in each convolution)
* batch_size= 32
* Number of Epochs=10

During training, the model is calculating loss and updating its parameters. Test cost is calculated using the updated parameters. For this case, the training data is composed of 13674 images (80% of whole data set: 17092) and test data is composed of 3418(20% of whole data set:17092) images.

```
def scratch_training(no_of_classes = 8, learning_rate = 1e-3, beta1 = 0.95, beta2 = 0.99, img_dim = 50,
img_depth = 1,
             f = 5, num_filt1 = 10, num_filt2 = 10, batch_size = 32, num_epochs = 10):
    data = np.hstack((X,Y))
    np.random.shuffle(data)
    data_train = round(0.8*len(data))
    data_test = round(data_train+0.2*len(data))
    train_x = X[:data_train]
    print("Shape of train data",train_x.shape)
    test_x = X[data_train:data_test]
    print("Shape of test data",test_x.shape)
    train_y = Y[:data_train]
    test_y = Y[data_train:data_test]
    train_data = np.hstack((train_x,train_y))
    test_data = np.hstack((test_x,test_y))
    train_data1= train_data[:10000]
    test_data1=  test_data[:3000]
    ## Initializing all the parameters
    f1, f2, w3, w4 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (96,4410), (8, 96)
    f1 = scratch_initializeFilter(f1)
    f2 = scratch_initializeFilter(f2)
    w3 = scratch_initializeWeight(w3)
    w4 = scratch_initializeWeight(w4)
```

```
b1 = np.zeros((f1.shape[0],1))
b2 = np.zeros((f2.shape[0],1))
b3 = np.zeros((w3.shape[0],1))
b4 = np.zeros((w4.shape[0],1))
parameters = [f1, f2, w3, w4, b1, b2, b3, b4]
train_cost = []
train_cost1=[]
test_cost = []
test_cost1=[]
print("Learning Rate:"+str(learning_rate)+", Batch Size:"+str(batch_size))

for epoch in range(num_epochs):
   np.random.shuffle(train_data1)
   batches = [train_data1[k:k + batch_size] for k in range(0, train_data1.shape[0], batch_size)]

   np.random.shuffle(test_data1)
   batches1 = [test_data1[k:k + batch_size] for k in range(0, test_data1.shape[0], batch_size)]

   t = tqdm(batches)
   for x,batch in enumerate(t):
      parameters, train_cost = scratch_adamGD(batch, no_of_classes, learning_rate, img_dim,
img_depth, beta1, beta2, parameters, train_cost)
      t.set_description("Training_Cost: %.2f" % (train_cost[-1]))
   train_cost1.append(train_cost)

   t1 = tqdm(batches1)
   for x,batch in enumerate(t1):
      parameters1, test_cost = scratch_adamGD(batch, no_of_classes, learning_rate, img_dim,
img_depth, beta1, beta2, parameters, test_cost)
      t1.set_description("Test_Cost: %.2f" % (test_cost[-1]))
   test_cost1.append(test_cost)

print ("parameters", parameters)

return parameters, train_cost1,test_cost1
```

6. Output of the Model:

➢ Model calculating cost for training and test data set:

Test cost is calculated using the updated parameters. For this case, the training data is composed of 13674 images (80% of whole data set: 17092) and test data is composed of 3418(20% of whole data set:17092) images. The updated parameters are printed and saved for later use.

```
Shape of train data (13674, 784)
Shape of test data (3418, 784)
Learning Rate:0.001, Batch Size:32

Training_Cost: 0.95: 100%| | 63/63 [18:36<00:00, 17.72s/it]
Test_Cost: 0.12: 100%| | 63/63 [18:18<00:00, 17.44s/it]
Training_Cost: 0.43: 100%| | 63/63 [18:28<00:00, 17.60s/it]
Test_Cost: 0.00: 100%| | 63/63 [18:27<00:00, 17.58s/it]
Training_Cost: 0.71: 100%| | 63/63 [18:32<00:00, 17.66s/it]
Test_Cost: 0.00: 100%| | 63/63 [18:12<00:00, 17.34s/it]
Training_Cost: 0.54: 100%| | 63/63 [18:19<00:00, 17.46s/it]
Test_Cost: 0.00: 100%| | 63/63 [18:34<00:00, 17.68s/it]
Training_Cost: 0.36: 100%| | 63/63 [18:29<00:00, 17.62s/it]
Test_Cost: 0.01: 100%| | 63/63 [18:50<00:00, 17.94s/it]
Training_Cost: 0.95: 100%| | 63/63 [18:35<00:00, 17.70s/it]
Test_Cost: 0.00: 100%| | 63/63 [18:32<00:00, 17.66s/it]
Training_Cost: 1.24: 100%| | 63/63 [18:31<00:00, 17.64s/it]
Test_Cost: 0.00: 100%| | 63/63 [18:09<00:00, 17.29s/it]
Training_Cost: 0.32: 100%| | 63/63 [18:24<00:00, 17.52s/it]
```

- ### Saving updated parameters:

The updated parameters are saved as text files, so that they can be used for prediction function.

First the parameters are flatten:

filter1= parameters[0].flatten()

filter2= parameters[1].flatten()

weight3= parameters[2].flatten()

weight4= parameters[3].flatten()

bias1= parameters[4].flatten()

bias2= parameters[5].flatten()

bias3= parameters[6].flatten()

bias4= parameters[7].flatten()

Saving as text files:

The parameters text files are provided in the project folder. These will be loaded in Prediction function Annex C.

f11 = np.savetxt('filter1.txt', filter1, delimiter=', ')

f22 = np.savetxt('filter2.txt', filter2, delimiter=', ')

w33 = np.savetxt('weight3.txt', weight3, delimiter=', ')

w44 = np.savetxt('weight4.txt', weight4, delimiter=', ')

b11 = np.savetxt('bias1.txt', bias1, delimiter=', ')

b22 = np.savetxt('bias2.txt', bias2, delimiter=', ')

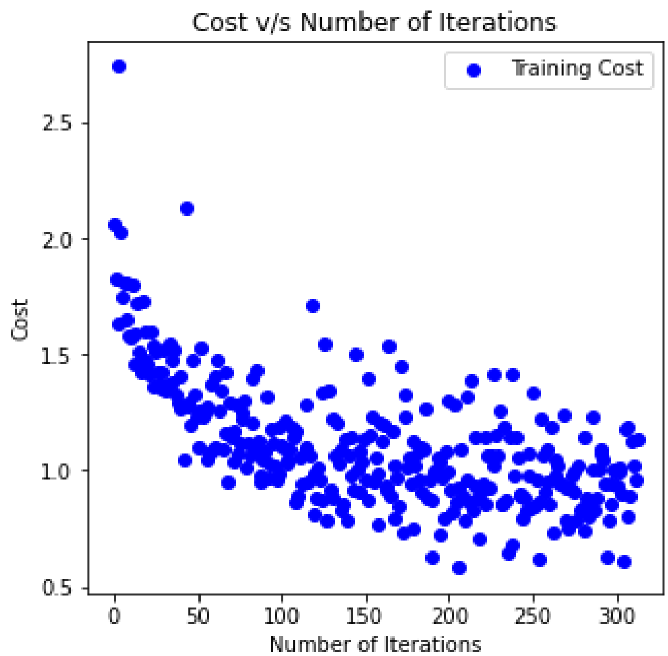
b22 = np.savetxt('bias3.txt', bias3, delimiter=', ')

```
b33 = np.savetxt('bias4.txt', bias4, delimiter=', ')
```

7. Plots:

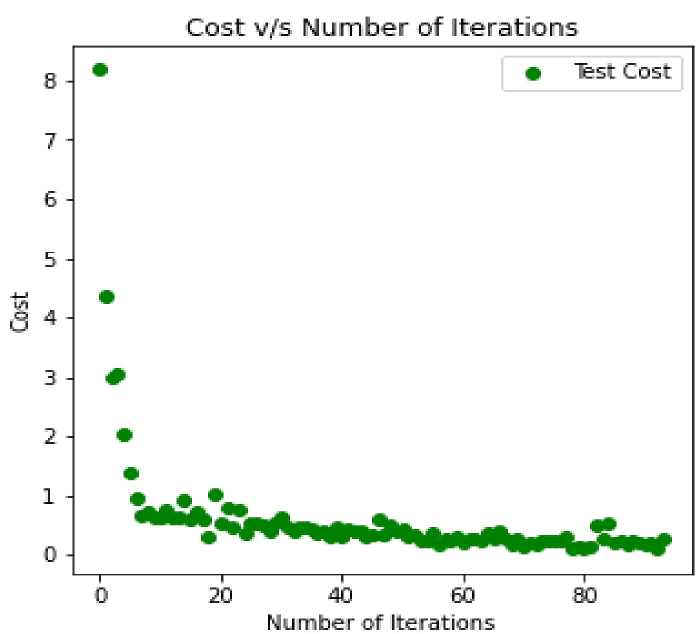
➢ Training Dataset:

The **training loss/cost** plot with respect to iterations for training data set is given below:



➢ Test Dataset :

The **test costl/loss** plot with respect to iterations for test data set is given below:



8. Accuracy on test data set:

➢ Prediction Function:

The updated weights/filters are used for making predictions on the given image and the label. The highest value of probability is considered as the prediction by the model for the given image.

```
def scratch_predict(image, f1, f2, w3, w4, b1, b2, b3, b4, conv_stride = 1, pool_filter = 2, pool_stride = 2):
    first_convolution = scratch_convolution(image, f1, b1, conv_stride)
    first_convolution[first_convolution<=0] = 0
    pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)
    second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)
    second_convolution[second_convolution<=0] = 0
    pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)
    (nf2, dim2, _) = pooled2.shape
    fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
    z = w3.dot(fully_connected) + b3
    z[z<=0] = 0
    out = w4.dot(z) + b4
    probs = scratch_softmax(out)
    return np.argmax(probs), np.max(probs)
```

➢ Accuracy:

The test data set is fed into the model and updated weight/parameters are used to find the accuracy of the model based on the number of correctly classified images. The accuracy on test set is **31.30%.**

```
data = np.hstack((X,Y))
    np.random.shuffle(data)
    data_train = round(0.8*len(data))
    data_test = round(data_train+0.2*len(data))
    test_x = X[data_train:data_test]
    test_y = Y[data_train:data_test]
    test_data = np.hstack((test_x,test_y))
```

```
np.random.shuffle(test_data)
test_x = test_x.reshape(len(test_data), 1, 50, 50)
print("Shape of test data",test_x.shape)
corr = 0
counter = [0 for i in range(10)]
correct_predictions = [0 for i in range(10)]
print()
print("Computing accuracy over test set:")
t = tqdm(range(len(test_x)), leave=True)
f1=parameters[0]
f2=parameters[1]
w3=parameters[2]
w4=parameters[3]
b1=parameters[4]
b2=parameters[5]
b3=parameters[6]
b4=parameters[7]
for i in t:
   x = test_x[i]
   pred, prob = scratch_predict(x, f1, f2, w3, w4, b1, b2, b3, b4)
   counter[int(test_y[i])]+=1
   if pred==test_y[i]:
      corr+=1
      correct_predictions[pred]+=1
   t.set_description("Acc:%0.2f%%" % (float(corr/(i+1))*100))
print("Overall Accuracy: %.2f" % (float(corr/len(test_x)*100)))
```

```
Shape of test data (3418, 1, 50, 50)
Computing accuracy over test set:
Acc:31.30%: 100%|██████████████████████████████████████████| 3418/3418 [1:07:56<00:00,  1.19s/it]
Overall Accuracy: 31.30
```

9. Conclusion:

The convolutional neural network was successfully implemented from scratch on given data set of blood cells. All the functions related to convolution, max pooling, fully connected layers, forward and back propagation are working fine.

The best accuracy achieved by the model is **31.30 %** on test images which can be further improved by tuning the hyper parameters of the model and changing the architecture of model.

10. Complete Codes:

Annex A

➢ Instructions on running the code:

Complete notebook with all the optimal parameters are provided.

- If user only wants to see the already trained book for the Dataset. Just open notebooks named:

  "Deep_Learning_Mid_Term_Project_Usman_Zaheer."

  This book is composed of complete model of CNN, Training code and all the necessary parameters to run the model.

  For good user experience visualization, it is available in jupyter notebook format and pdf format as well.

- If user wants to use the Prediction Code, user just have to run the book named:

  "Prediction Function for CNN_Mid_Term_Project_Usman Zaheer."

  The text files of all the 08 trained parameters are provided in the project folder. User just have to load the files and run the code as instructed in the jupyter notebook.

Annex B

➢ Training Code:

The training code of model is given below:

- If user only wants to train the model for the Dataset. Just open notebook named:
  "Deep_Learning_Mid_Term_Project _Usman_Zaheer."

Place the dataset in the directory folder. Run all the cells as instructed in the book. It includes all the training parameters.

```
import numpy as np

import matplotlib.pyplot as plt

import cv2

import math

from tqdm import tqdm

from os import listdir

 FOLDER_NAME = 'Dataset'

folders = listdir(FOLDER_NAME)

number_of_classes = len(folders)

data_X, data_Y = [], []

for i in range(number_of_classes):

  folder = folders[i]

  print('Folder', i+1, '-', folder)

  images = listdir(FOLDER_NAME + '/' + folder)

  # Walk over images

  for image in images:

    path = FOLDER_NAME + '/' + folder + '/' + image

    # Process Image

    raw = cv2.imread(path)

    gray = cv2.cvtColor(raw, cv2.COLOR_BGR2GRAY)

    img = cv2.resize(gray, (50, 50))

    # Add to data
```

```
        data_X.append(img.flatten())
        data_Y.append(i)
# Convert to numpy arrays
X = np.array(data_X)
Y = np.array(data_Y).reshape([-1, 1])
print('Total training examples:', len(X))
print("Total Images", X.shape)
print("Total Labels", Y.shape)
def scratch_convolution(image, filter, bias, stride=1):
    (no_of_filters, no_of_channels_f, f, _) = filter.shape
    number_of_channels_image, image_dim, _ = image.shape
    out_dim = int((image_dim - f)/stride)+1
    assert number_of_channels_image == no_of_channels_f
    convolved = np.zeros((no_of_filters,out_dim,out_dim))
    #Moving window over the image
    for curr_filter in range(no_of_filters):
        curr_y = out_y = 0
        while curr_y + f <= image_dim:
            curr_x = out_x = 0
            while curr_x + f <= image_dim:
                convolved[curr_filter, out_y, out_x] = np.sum(filter[curr_filter] * image[:,curr_y:curr_y+f,
curr_x:curr_x+f]) + bias[curr_filter]
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
    return convolved
def scratch_maxpool(image, filter=2, stride=2):
```

```
    number_of_channels, h_orig, w_orig = image.shape
    height = int((h_orig - filter)/stride)+1
    width = int((w_orig - filter)/stride)+1
    downsampled = np.zeros((number_of_channels, height, width))
    #Moving window over the image
    for i in range(number_of_channels):
        curr_y = out_y = 0
        while curr_y + filter <= h_orig:
            curr_x = out_x = 0
            while curr_x + filter <= w_orig:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+filter, curr_x:curr_x+filter])
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
    return downsampled
def scratch_softmax(scores):
    out = np.exp(scores)
    return out/np.sum(out)
def scratch_CrossEntropyLoss(probs, label):
    return  np.sum(label * np.log(probs))
def scratch_initializeFilter(size, scale = 1.0):
    stddev = scale/np.sqrt(np.prod(size))
    return np.random.normal(loc = 0, scale = stddev, size = size)
def scratch_initializeWeight(size):
    return np.random.standard_normal(size=size) * 0.01
def scratch_convolutionBackward(dconv_prev, conv_in, filter, stride):
```

```
    (no_of_filters, number_of_channels_f, f, _) = filter.shape
    (_, orig_dim, _) = conv_in.shape
    dout = np.zeros(conv_in.shape)
    dfilter = np.zeros(filter.shape)
    dbias = np.zeros((no_of_filters,1))
    for current_filter in range(no_of_filters):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                dfilter[current_filter] += dconv_prev[current_filter, out_y, out_x] * conv_in[:,
curr_y:curr_y+f, curr_x:curr_x+f]
                dout[:, curr_y:curr_y+f, curr_x:curr_x+f] += dconv_prev[current_filter, out_y, out_x] *
filter[current_filter]
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
        dbias[current_filter] = np.sum(dconv_prev[current_filter])
    return dout, dfilter, dbias
def nanargmax(arr):
    idx = np.nanargmax(arr)
    idxs = np.unravel_index(idx, arr.shape)
    return idxs
def scratch_maxpoolBackward(dpool, orig, f, stride):
    (number_of_channels_f, orig_dim, _) = orig.shape
    dout = np.zeros(orig.shape)
    for curr_c in range(number_of_channels_f):
        curr_y = out_y = 0
```

```
    while curr_y + f <= orig_dim:
      curr_x = out_x = 0
      while curr_x + f <= orig_dim:
        (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f, curr_x:curr_x+f])
        dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]
        curr_x += stride
        out_x += 1
      curr_y += stride
      out_y += 1
  return dout
def scratch_complete_model(image, label, params, conv_stride, pool_filter, pool_stride):
  [f1, f2, w3, w4, b1, b2, b3, b4] = params
  first_convolution = scratch_convolution(image, f1, b1, conv_stride)
  first_convolution[first_convolution<=0] = 0
  pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)
  second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)
  second_convolution[second_convolution<=0] = 0
  pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)
  (nf2, dim2, _) = pooled2.shape
  fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
  z = w3.dot(fully_connected) + b3
  z[z<=0] = 0
  out = w4.dot(z) + b4
  probs = scratch_softmax(out)
  loss = scratch_CrossEntropyLoss(probs, label)
  dout = probs - label
  dw4 = dout.dot(z.T)
  db4 = np.sum(dout, axis = 1).reshape(b4.shape)
  dz = w4.T.dot(dout)
```

```
    dz[z<=0] = 0
    dw3 = dz.dot(fully_connected.T)
    db3 = np.sum(dz, axis = 1).reshape(b3.shape)
    dfully_connected = w3.T.dot(dz)
    dpool2 = dfully_connected.reshape(pooled2.shape)
    dsecond_convolution = scratch_maxpoolBackward(dpool2, second_convolution, pool_filter, 
pool_stride)
    dsecond_convolution[second_convolution<=0] = 0
    dpool1, df2, db2 = scratch_convolutionBackward(dsecond_convolution, pooled1, f2, conv_stride)
    dfirst_convolution= scratch_maxpoolBackward(dpool1, first_convolution, pool_filter, pool_stride)
    dfirst_convolution[first_convolution<=0] = 0
    dimage, df1, db1 = scratch_convolutionBackward(dfirst_convolution, image, f1, conv_stride)
    gradients = [df1, df2, dw3, dw4, db1, db2, db3, db4]
    return gradients, loss
def scratch_adamGD(batch, no_of_classes, learning_rate, dim, n_c, beta1, beta2, parameters, cost):
    [f1, f2, w3, w4, b1, b2, b3, b4] = parameters
    X = batch[:,0:-1]
    X = X.reshape(len(batch), n_c, dim, dim)
    Y = batch[:,-1]
    cost_ = 0
    batch_size = len(batch)
    df1 = np.zeros(f1.shape)
    df2 = np.zeros(f2.shape)
    dw3 = np.zeros(w3.shape)
    dw4 = np.zeros(w4.shape)
    db1 = np.zeros(b1.shape)
    db2 = np.zeros(b2.shape)
    db3 = np.zeros(b3.shape)
```

```
db4 = np.zeros(b4.shape)
v1 = np.zeros(f1.shape)
v2 = np.zeros(f2.shape)
v3 = np.zeros(w3.shape)
v4 = np.zeros(w4.shape)
bv1 = np.zeros(b1.shape)
bv2 = np.zeros(b2.shape)
bv3 = np.zeros(b3.shape)
bv4 = np.zeros(b4.shape)
s1 = np.zeros(f1.shape)
s2 = np.zeros(f2.shape)
s3 = np.zeros(w3.shape)
s4 = np.zeros(w4.shape)
bs1 = np.zeros(b1.shape)
bs2 = np.zeros(b2.shape)
bs3 = np.zeros(b3.shape)
bs4 = np.zeros(b4.shape)
for i in range(batch_size):
  x = X[i]
  y = np.eye(no_of_classes)[int(Y[i])].reshape(no_of_classes, 1) # convert label to one-hot
  grads, loss = scratch_complete_model(x, y, parameters, 1, 2, 2)
  [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads
  df1+=df1_
  db1+=db1_
  df2+=df2_
  db2+=db2_
  dw3+=dw3_
  db3+=db3_
```

```
        dw4+=dw4_
        db4+=db4_
        cost_+= loss
    v1 = beta1*v1 + (1-beta1)*df1/batch_size # momentum update
    s1 = beta2*s1 + (1-beta2)*(df1/batch_size)**2 # RMSProp update
    f1 -= learning_rate * v1/np.sqrt(s1+1e-7) # combine momentum and RMSProp to perform update with Adam
    bv1 = beta1*bv1 + (1-beta1)*db1/batch_size
    bs1 = beta2*bs1 + (1-beta2)*(db1/batch_size)**2
    b1 -= learning_rate * bv1/np.sqrt(bs1+1e-7)
    v2 = beta1*v2 + (1-beta1)*df2/batch_size
    s2 = beta2*s2 + (1-beta2)*(df2/batch_size)**2
    f2 -= learning_rate * v2/np.sqrt(s2+1e-7)
    bv2 = beta1*bv2 + (1-beta1) * db2/batch_size
    bs2 = beta2*bs2 + (1-beta2)*(db2/batch_size)**2
    b2 -= learning_rate * bv2/np.sqrt(bs2+1e-7)
    v3 = beta1*v3 + (1-beta1) * dw3/batch_size
    s3 = beta2*s3 + (1-beta2)*(dw3/batch_size)**2
    w3 -= learning_rate * v3/np.sqrt(s3+1e-7)
    bv3 = beta1*bv3 + (1-beta1) * db3/batch_size
    bs3 = beta2*bs3 + (1-beta2)*(db3/batch_size)**2
    b3 -= learning_rate * bv3/np.sqrt(bs3+1e-7)
    v4 = beta1*v4 + (1-beta1) * dw4/batch_size
    s4 = beta2*s4 + (1-beta2)*(dw4/batch_size)**2
    w4 -= learning_rate * v4 / np.sqrt(s4+1e-7)
    bv4 = beta1*bv4 + (1-beta1)*db4/batch_size
    bs4 = beta2*bs4 + (1-beta2)*(db4/batch_size)**2
    b4 -= learning_rate * bv4 / np.sqrt(bs4+1e-7)
```

```
    cost_ = cost_/batch_size

    cost.append(cost_)

    parameters = [f1, f2, w3, w4, b1, b2, b3, b4]

    return parameters, cost

def scratch_training(no_of_classes = 8, learning_rate = 1e-3, beta1 = 0.95, beta2 = 0.99, img_dim = 50,
img_depth = 1,
            f = 5, num_filt1 = 10, num_filt2 = 10, batch_size = 32, num_epochs = 10):
    data = np.hstack((X,Y))
    np.random.shuffle(data)
    data_train = round(0.8*len(data))
    data_test = round(data_train+0.2*len(data))
    train_x = X[:data_train]
    print("Shape of train data",train_x.shape)
    test_x = X[data_train:data_test]
    print("Shape of test data",test_x.shape)
    train_y = Y[:data_train]
    test_y = Y[data_train:data_test]
    train_data = np.hstack((train_x,train_y))
    test_data = np.hstack((test_x,test_y))
    train_data1= train_data[:10000]
    test_data1=  test_data[:3000]
    ## Initializing all the parameters
    f1, f2, w3, w4 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (96,4410), (8, 96)
    f1 = scratch_initializeFilter(f1)
    f2 = scratch_initializeFilter(f2)
    w3 = scratch_initializeWeight(w3)
    w4 = scratch_initializeWeight(w4)
    b1 = np.zeros((f1.shape[0],1))
    b2 = np.zeros((f2.shape[0],1))
    b3 = np.zeros((w3.shape[0],1))
    b4 = np.zeros((w4.shape[0],1))
    parameters = [f1, f2, w3, w4, b1, b2, b3, b4]
    train_cost = []
    train_cost1=[]
    test_cost = []
    test_cost1=[]
    print("Learning Rate:"+str(learning_rate)+", Batch Size:"+str(batch_size))

    for epoch in range(num_epochs):
        np.random.shuffle(train_data1)
        batches = [train_data1[k:k + batch_size] for k in range(0, train_data1.shape[0], batch_size)]
```

```
        np.random.shuffle(test_data1)
        batches1 = [test_data1[k:k + batch_size] for k in range(0, test_data1.shape[0], batch_size)]

        t = tqdm(batches)
        for x,batch in enumerate(t):
            parameters, train_cost = scratch_adamGD(batch, no_of_classes, learning_rate, img_dim,
img_depth, beta1, beta2, parameters, train_cost)
            t.set_description("Training_Cost: %.2f" % (train_cost[-1]))
        train_cost1.append(train_cost)

        t1 = tqdm(batches1)
        for x,batch in enumerate(t1):
            parameters1, test_cost = scratch_adamGD(batch, no_of_classes, learning_rate, img_dim,
img_depth, beta1, beta2, parameters, test_cost)
            t1.set_description("Test_Cost: %.2f" % (test_cost[-1]))
        test_cost1.append(test_cost)

    print ("parameters", parameters)

    return parameters, train_cost1,test_cost1

parameters, train_cost, test_cost = scratch_train(no_of_classes = 8, learning_rate = 1e-3, beta1 = 0.95,
beta2 = 0.99, img_dim = 50, img_depth = 1, f = 5, num_filt1 = 10, num_filt2 = 10, batch_size = 32,
num_epochs = 10)

def scratch_predict(image, f1, f2, w3, w4, b1, b2, b3, b4, conv_stride = 1, pool_filter = 2, pool_stride =
2):

    first_convolution = scratch_convolution(image, f1, b1, conv_stride)

    first_convolution[first_convolution<=0] = 0

    pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)

    second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)

    second_convolution[second_convolution<=0] = 0

    pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)

    (nf2, dim2, _) = pooled2.shape

    fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))

    z = w3.dot(fully_connected) + b3

    z[z<=0] = 0

    out = w4.dot(z) + b4
```

```
probs = scratch_softmax(out)
return np.argmax(probs), np.max(probs)
data = np.hstack((X,Y))
np.random.shuffle(data)
data_train = round(0.8*len(data))
data_test = round(data_train+0.2*len(data))
test_x = X[data_train:data_test]
test_y = Y[data_train:data_test]
test_data = np.hstack((test_x,test_y))
np.random.shuffle(test_data)
test_x = test_x.reshape(len(test_data), 1, 50, 50)
print("Shape of test data",test_x.shape)
corr = 0
counter = [0 for i in range(10)]
correct_predictions = [0 for i in range(10)]
print()
print("Computing accuracy over test set:")
t = tqdm(range(len(test_x)), leave=True)
f1=parameters[0]
f2=parameters[1]
w3=parameters[2]
w4=parameters[3]
b1=parameters[4]
b2=parameters[5]
b3=parameters[6]
b4=parameters[7]
for i in t:
  x = test_x[i]
  pred, prob = scratch_predict(x, f1, f2, w3, w4, b1, b2, b3, b4)
```

```
        counter[int(test_y[i])]+=1
        if pred==test_y[i]:
            corr+=1
            correct_predictions[pred]+=1
        t.set_description("Acc:%0.2f%%" % (float(corr/(i+1))*100))
print("Overall Accuracy: %.2f" % (float(corr/len(test_x)*100)))
```

Annex C

➢ Prediction Code:

The jupyter book named **"Prediction Function for CNN_Mid_Term_Project_Usman Zaheer"** contains the prediction code and necessary steps to run it. User just have to load the trained parameters, text files provided in the project folder and a test image.

The predict function will provide probabilities, prediction label and loss for the given image and the corresponding label:

```
import cv2 as cv

import matplotlib.pyplot as plt

import numpy as np

f1 = np.loadtxt("filter1.txt").reshape(10, 1,5,5)

f2 = np.loadtxt("filter2.txt").reshape(10, 10,5,5)

w3 = np.loadtxt("weight3.txt").reshape(96,4410)

w4 = np.loadtxt("weight4.txt").reshape(8,96)

b1 = np.loadtxt("bias1.txt").reshape(10, 1)

b2 = np.loadtxt("bias2.txt").reshape(10, 1)

b3 = np.loadtxt("bias3.txt").reshape(96,1)

b4 = np.loadtxt("bias4.txt").reshape(8,1)

print("f1=", f1.shape)

print("f2=", f2.shape)

print("w3=", w3.shape)

print("w4=", w4.shape)

print("b1=", b1.shape)

print("b2=", b2.shape)

print("b3=", b3.shape)

print("b4=", b4.shape)

parameters= [f1, f2, w3, w4, b1, b2, b3, b4]

def Prediction_Function(image, label, parameters, conv_stride, pool_filter, pool_stride):

   first_convolution = scratch_convolution(image, f1, b1, conv_stride)

   first_convolution[first_convolution<=0] = 0
```

```
    pooled1 = scratch_maxpool(first_convolution, pool_filter, pool_stride)
    second_convolution = scratch_convolution(first_convolution, f2, b2, conv_stride)
    second_convolution[second_convolution<=0] = 0
    pooled2 = scratch_maxpool(second_convolution, pool_filter, pool_stride)
    (nf2, dim2, _) = pooled2.shape
    fully_connected = pooled2.reshape((nf2 * dim2 * dim2, 1))
    z = w3.dot(fully_connected) + b3
    z[z<=0] = 0
    output = w4.dot(z) + b4
    probs = scratch_softmax(output)
    loss = scratch_CrossEntropyLoss(probs, label)
    return probs, np.argmax(probs), loss
def scratch_convolution(image, filter, bias, stride=1):
    (no_of_filters, number_of_channels_f, f, _) = filter.shape
    number_of_channels, image_dim, _ = image.shape
    out_dim = int((image_dim - f)/stride)+1
    assert number_of_channels == number_of_channels_f
    convolved = np.zeros((no_of_filters,out_dim,out_dim))
    #moving window over image
    for curr_filter in range(no_of_filters):
        curr_y = out_y = 0
        while curr_y + f <= image_dim:
            curr_x = out_x = 0
            while curr_x + f <= image_dim:
                convolved[curr_filter, out_y, out_x] = np.sum(filter[curr_filter] * image[:,curr_y:curr_y+f,
curr_x:curr_x+f]) + bias[curr_filter]
                curr_x += stride
                out_x += 1
            curr_y += stride
```

```
            out_y += 1
    return convolved
def scratch_maxpool(image, filter=2, stride=2):
    number_of_channels, h_prev, w_prev = image.shape
    height = int((h_prev - filter)/stride)+1
    width = int((w_prev - filter)/stride)+1
    downsampled = np.zeros((number_of_channels, height, width))
    #moving window over image
    for i in range(number_of_channels):
        curr_y = out_y = 0
        while curr_y + filter <= h_prev:
            curr_x = out_x = 0
            while curr_x + filter <= w_prev:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+filter, curr_x:curr_x+filter])
                curr_x += stride
                out_x += 1
            curr_y += stride
            out_y += 1
    return downsampled
def scratch_softmax(scores):
    out = np.exp(scores)
    return out/np.sum(out)
def scratch_CrossEntropyLoss(probs, label):
    return -np.sum(label * np.log(probs))
image= cv.imread("BA_47.jpg",0)
plt.imshow(image)
plt.xlabel("Basophil")
image= cv.resize(image,(50,50),interpolation = cv.INTER_AREA)
image=np.reshape(image, (1,50,50))
```

```
Probabilities,Label,Loss = Prediction_Function(image, label=6, parameters = parameters,
conv_stride=1, pool_filter=2, pool_stride=2)

print("Probabilities:",Probabilities)

print("Predicted Label:",Label)

print("Loss:",Loss)
```