RIME-833 : MID SEMESTER PROJECT REPORT **Ahmed Mustafa (330143)**

Problem Statement

The goal of the project is to build a Convolutional Neural Network (CNN) from scratch in order to classify between different types of blood cells using images taken under a microscope.

Dataset

The dataset provided consists of a total of 8 classes of cells each with a number of training examples. The classes and their data distribution are as follows:

Class 0: Basophil	Training Examples: 1218
Class 1: Eosinophil	Training Examples: 3117
Class 2: Erythroblast	Training Examples: 1551
Class 3: Immunoglobulin	Training Examples: 2895
Class 4: Lymphocyte	Training Examples: 1214
Class 5: Monocyte	Training Examples: 1420
Class 6: Neutrophil	Training Examples: 3329
Class 7: Platelet	Training Examples: 2348

The total examples in the dataset add up to be 17,092 images. Each image in the dataset is a color image with dimensions 3x363x360.

Data Preprocessing

The images imported from the dataset were preprocessed before being used to train the model. The images were converted from RGB to Grayscale and resized so the final shape of each image in the dataset was 1x50x50. Each image was then linearly scaled down by a factor of 1/255 so the input values fall between a range from 0 to 1.

1200 examples from each class were chosen to formulate the training dataset. The training dataset thus contained a total of 9600 images. The remaining images were split in half to compose the validation and test dataset, each consisting of 3746 images.

Hence the resulting train:validation:test dataset ratio was approximately 56:22:22.

Network Architecture

The Convolutional Neural Network is built with the following architecture:

Convolution 1 – MaxPool 1 – Convolution 2 – MaxPool 2 – Flat – Hidden 1 – Hidden 2 – Output

- The first Convolutional Layer has 16@3x3 feature maps with a stride of 1 and a ReLU activation.
- The first MaxPool Layer has a kernel size of 3x3 and a stride of 3.
- The second Convolutional Layer has 32@5x5 feature maps with a stride of 1 and a ReLU activation.
- The second MaxPool layer has a kernel size of 2x2 and a stride of 2.
- The first Hidden Layer has 512 units with a ReLU activation.
- The second Hidden Layer has 64 units with a ReLU activation.
- The Output Layer has 8 units with a SoftMax activation.

The cross-entropy function is used as the loss function for the model.

The figure shows a representation of the CNN architecture:

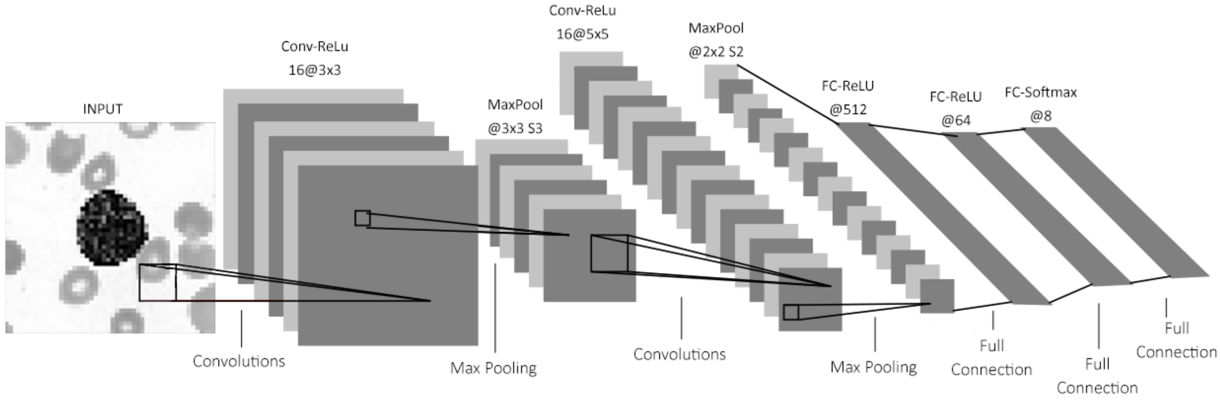

Fig. 1 – Model Architecture for cell classification

Model Optimization

The ADAM optimizer was used to train the CNN over the training data. The following hyper-parameters were used for training:

$$\boldsymbol{\alpha}: 0.01 \qquad \boldsymbol{\beta_1}: 0.90 \qquad \boldsymbol{\beta_2}: 0.99$$

The model was trained for 12 epochs with a batch-size of 150.

The following graph shows the behavior of the training and validation cross-entropy loss during training:

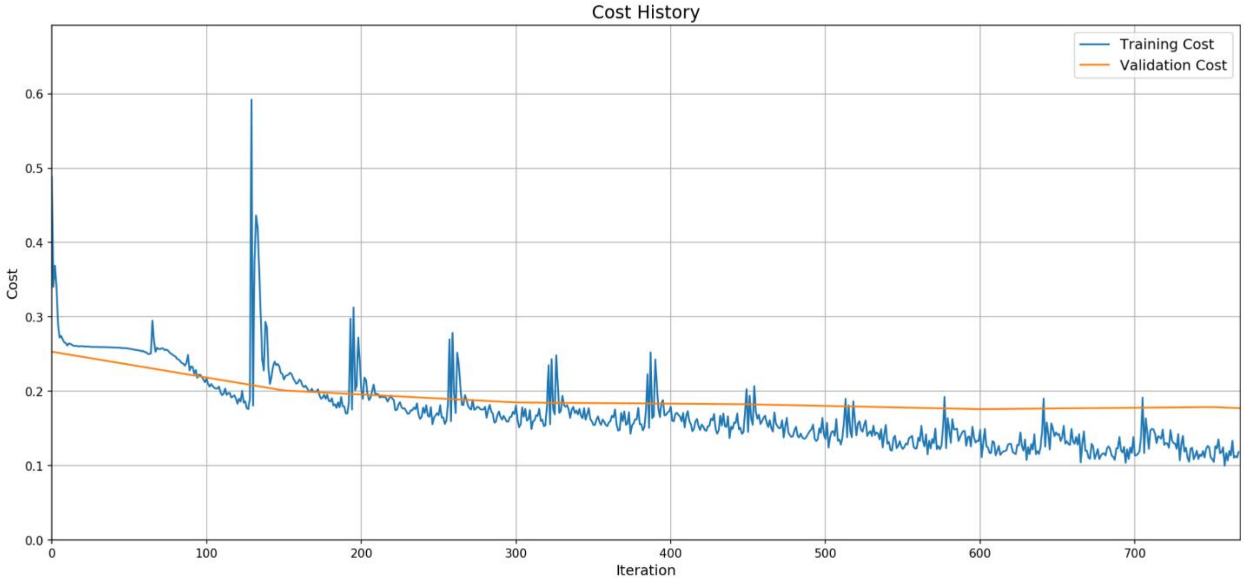

Fig. 2 – Training and Validation Cost

The average loss on the training dataset by the final epoch is 0.126 and the loss on the validation dataset by the final epoch is 0.124.

Results

The following image is a 1x50x50 input image of a Basophil cell which is fed to the trained CNN for classification:

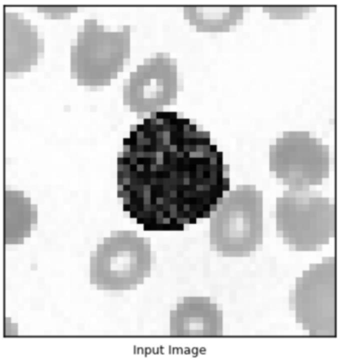

Fig. 3 – Input Image to CNN of Basophil

As the image propagates through the network, the outputs of the Convolutional Layers can be observed to deduce the characteristic features of the cells the model interprets for classification.

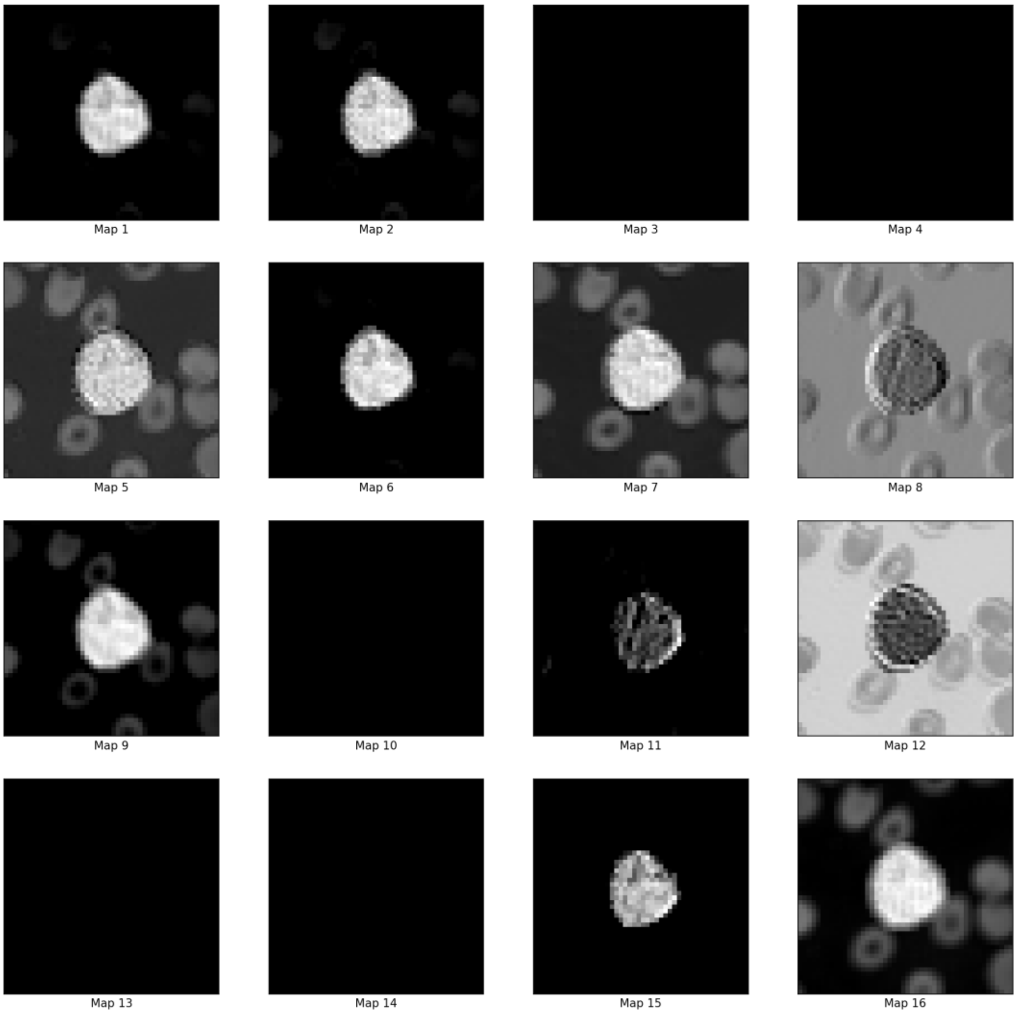

Fig. 4 – Activation Maps from Convolutional Layer 1

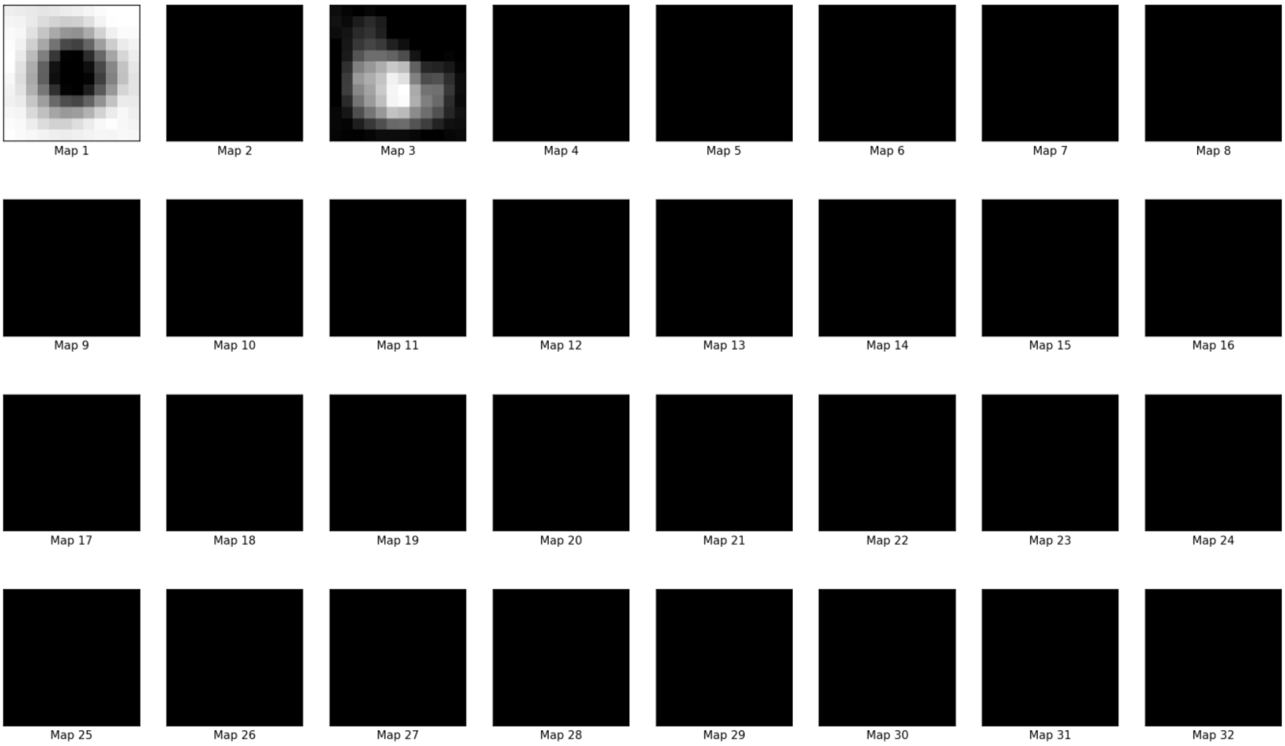

Fig. 5 – Activation Maps from Convolutional Layer 2

The output from the model is an 8x1 array representing the probability that the image belongs to the class represented by the row index of the array:

```
[[0.7878757067 ]
 [ 0.0256212681 ]
 [ 0.0194180202 ]
 [ 0.0774566296 ]
 [ 0.0068471321 ]
 [ 0.081536059  ]
 [ 0.0012451842 ]
 [ 0.          ]]
```

The results state that according to the model, there is approximately a 78.8% chance that the image provided at the input belongs to Class 0 which is Basophil.

The prediction is correct.

Model Evaluation

The prediction accuracy on the training dataset is 64.0% and the prediction accuracy on the test dataset is 62.4%. The overall accuracy on the entire dataset is 63.4%.

The following figure shows the confusion matrix for the trained model on the provided dataset:

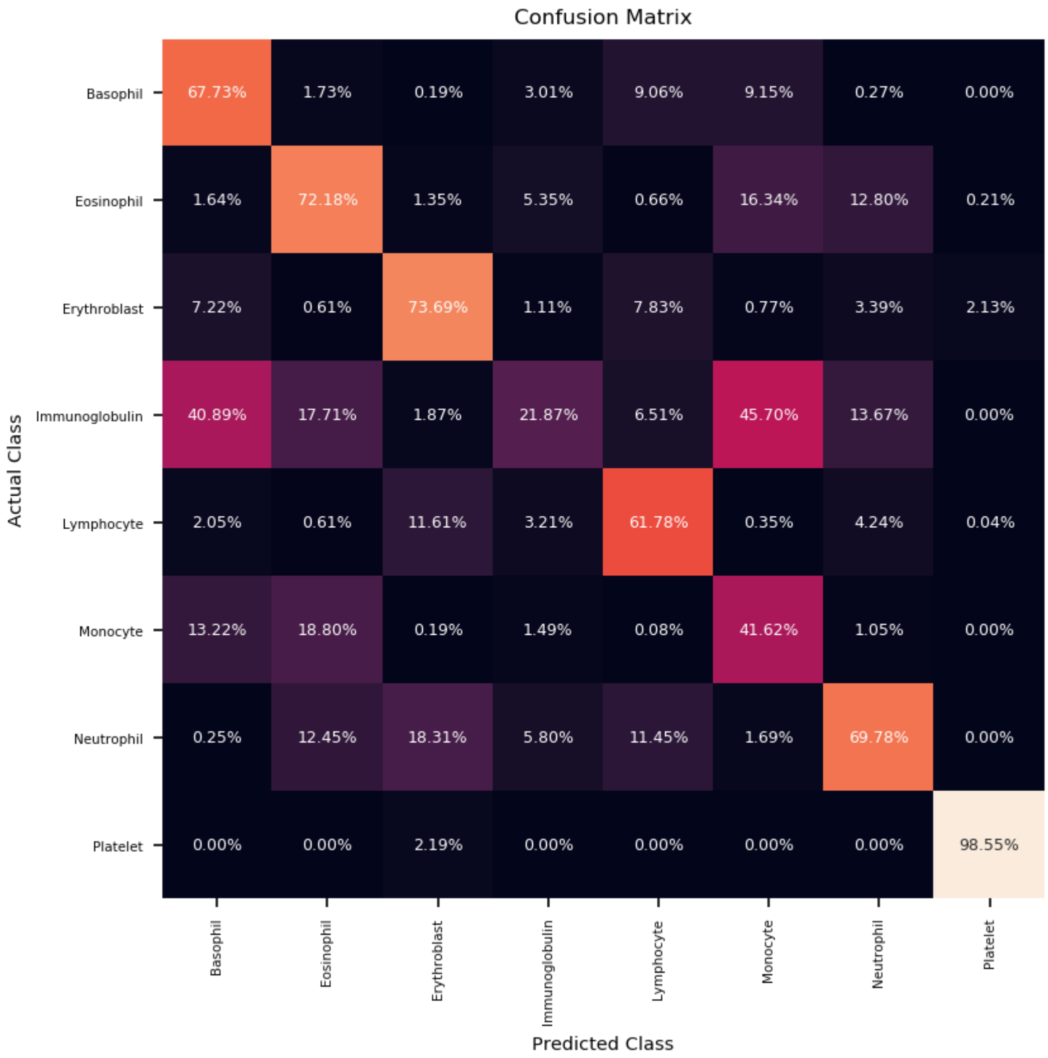

Fig. 6 – Confusion Matrix

It can be deduced from the Confusion Matrix that the model has a fairly good accuracy on predicting all classes except for Immunoglobulins and Monocytes.

Prediction

To perform predictions using the model, add the image/s to the *Prediction Dataset* folder and run the *3 – Prediction.ipynb* file to get results for all the images present in the folder.

Attachments

The following files and folders are attached in the project:

1. *Codes* folder which contains HTML versions of the Library & Training, Evaluation and Prediction scripts.

2. *Dataset* folder consisting of all the images used to train and evaluate the model.

3. *Models* folder containing text files that store the weights and costs for the trained model loaded for evaluation and prediction.

4. *Prediction Dataset* folder which is used to perform predictions using the trained CNN. Images needed to be classified are added to this folder.

5. *1 – Training.ipnyb* file which contains the script used to train the CNN.

6. *2 – Evaluation.ipnyb* file which contains the script used to evaluate the model.

7. *3 – Prediction.ipnyb* file which contains the script that is run to perform predictions using the CNN.

8. *cnn.py* file which is the coded library used to function the entire CNN.

Appendix

The code used for the CNN library, training, evaluation and prediction is attached below:

Library

```
import numpy as np
np.set_printoptions(suppress=True)
from json import dump, load

class Convolution:

    # Constructor
    def __init__(self, num_f, f_size, in_shape, stride = 1):
        # Extracts vars
        num_c, in_dim, _ = in_shape
        # Assign vars
        self.num_f = num_f; self.f_size = f_size; self.num_c = num_c; self.in_dim = in_dim
; self.s = stride
        self.out_dim = int((in_dim - f_size) / stride) + 1
        self.out_shape = (num_f, self.out_dim, self.out_dim)
        # Initialize Weigths and Biases
        f_dims = (num_f, num_c, f_size, f_size)
        scale = 1 / np.sqrt(np.prod(f_dims))
        self.filters = np.random.normal(0, scale, f_dims)
        self.biases = np.random.randn(num_f, 1)
        # Initialize Accumulators
        self.sigma_acc = self.biases * 0
        self.delta_acc = self.filters * 0
        # Initialize Adam Paras
        self.Vdw = self.Sdw = self.delta_acc
        self.Vdb = self.Sdb = self.sigma_acc
        self.t = 1

    # ReLU Activation Function
    def relu(self, arr):
        return arr * (arr > 0)

    # Forward Propagation
    def step(self, img):
        # Assign Input
        self.in_ = img
        # Initialize Output
        out = np.zeros(self.out_shape)
        # Slide Window
        img_i = out_i = 0
        while img_i + self.f_size <= self.in_dim:
            img_j = out_j = 0
            while img_j + self.f_size <= self.in_dim:
                # Convolve
                window = img[:, img_i:img_i+self.f_size, img_j:img_j+self.f_size]
                out[:, out_i, out_j] = np.sum(window * self.filters, (1, 2, 3)) + self.bia
ses[:,0]
                # Slide left
                img_j += self.s; out_j += 1
            # Slide down
            img_i += self.s; out_i += 1
        # Return
        self.out = self.relu(out)
        return self.out

    # Back Propagation
    def back(self, grad):
        # Reverse Activation
        grad = grad * (self.out > 0)
```

```
        # Initialize Outputs
        sigmas = np.sum(grad, (1, 2)).reshape(-1, 1)
        deltas = np.zeros(self.filters.shape)
        global_grad = np.zeros(self.in_.shape)
         # Slide Window
        img_i = out_i = 0
        while img_i + self.f_size <= self.in_dim:
            img_j = out_j = 0
            while img_j + self.f_size <= self.in_dim:
                # Calculate dF
                window = self.in_[:, img_i:img_i+self.f_size, img_j:img_j+self.f_size]
                tiled = np.repeat(window[None, :, :, :], self.num_f, 0)
                deltas += tiled * grad[:, out_i, out_j].reshape((self.num_f, 1, 1, 1))
                # Calculate dI
                gradients = grad[:, out_i, out_j].reshape((self.num_f, 1, 1, 1))
                global_grad[:, img_i:img_i+self.f_size, img_j:img_j+self.f_size] += np.sum
(gradients * self.filters, 0)
                # Slide left
                img_j += self.s; out_j += 1
            # Slide down
            img_i += self.s; out_i += 1
        # Accumulate
        self.sigma_acc += sigmas
        self.delta_acc += deltas
        # Return
        return global_grad

    # Train
    def update(self, alpha, batch_size, beta_1 = 0.9, beta_2 = 0.99):
        # Mini-Batch Updates
        dw = self.delta_acc / batch_size; self.delta_acc *= 0
        db = self.sigma_acc / batch_size; self.sigma_acc *= 0
        # Momentum
        self.Vdw = (beta_1 * self.Vdw) + (1 - beta_1) * dw
        self.Vdb = (beta_1 * self.Vdb) + (1 - beta_1) * db
        # RMS Prop
        self.Sdw = (beta_2 * self.Sdw) + (1 - beta_2) * (dw ** 2)
        self.Sdb = (beta_2 * self.Sdb) + (1 - beta_2) * (db ** 2)
        # Corrected Momentum
        Vdw = self.Vdw / (1  - beta_1**self.t)
        Vdb = self.Vdb / (1  - beta_1**self.t)
        # Corrected RMS Prop
        Sdw = self.Sdw / (1  - beta_2**self.t)
        Sdb = self.Sdb / (1  - beta_2**self.t)
        # Update Parameters
        eps = 1e-9
        self.filters -= alpha * (Vdw / (np.sqrt(Sdw) + eps))
        self.biases -= alpha * (Vdb / (np.sqrt(Sdb) + eps))
        self.t += 0

    # Reset Adam Parameters Function
    def resetAdam(self):
        self.Vdw *= 0; self.Sdw *= 0
        self.Vdb *= 0; self.Sdb *= 0
        self.t = 1

class Pool:

    # Constructor
    def __init__(self, f_size, in_shape, stride = 1):
        # Extracts vars
        num_c, in_dim, _ = in_shape
        # Assign vars
        self.f_size = f_size; self.num_c = num_c; self.in_dim = in_dim; self.s = stride
        self.out_dim = int((in_dim - f_size) / stride) + 1
        self.out_shape = (self.num_c, self.out_dim, self.out_dim)
        self.size = np.prod(self.out_shape)
```

```
    # Forward Propagation
    def step(self, img):
        # Assign Input
        self.in_ = img
        # Initialize Output and Mask
        out = np.zeros(self.out_shape)
        self.masks = []
        # Slide Window
        img_i = out_i = 0
        while img_i + self.f_size <= self.in_dim:
            img_j = out_j = 0
            while img_j + self.f_size <= self.in_dim:
                # Pool
                window = img[:, img_i:img_i+self.f_size, img_j:img_j+self.f_size]
                pooled = np.max(window, (1, 2))
                out[:, out_i, out_j] = pooled
                # Update masks
                mask = pooled.reshape((self.num_c, 1, 1)) == window
                val = (img_i, img_j, mask)
                self.masks.append(val)
                # Slide left
                img_j += self.s; out_j += 1
            # Slide down
            img_i += self.s; out_i += 1
        # Return
        return out

    # Back Propagation
    def back(self, grad):
        # Initialize Output and Mask
        out = np.zeros((self.num_c, self.in_dim, self.in_dim))
        # Loop over grad
        for i, val in enumerate(self.masks):
            # Gradient Array Indices
            grad_i = int(i / self.out_dim)
            grad_j = i % self.out_dim
            # Unpack Mask Val
            out_i, out_j, mask = val
            # Back Pool
            gradients = grad[:, grad_i, grad_j].reshape((self.num_c, 1, 1))
            out[:, out_i:out_i+self.f_size, out_j:out_j+self.f_size] = mask * gradients
        # Return
        return out

class Flat:

    # Forward Propagation
    def step(self, img):
        self.in_dim = img.shape
        return np.reshape(img, (img.size, 1))

    # Back Propagation
    def back(self, vec):
        return vec.reshape(self.in_dim)

class Dense:

    # Constructor
    def __init__(self, size, in_size, activation = 'relu'):
        # Assign vars
        self.size = size; self.activation = activation
        # Initialize Weights and Biases
        weights_dims = (size, in_size)
        self.weights = np.random.standard_normal(weights_dims) * 0.1
        self.biases = np.zeros([size, 1])
        # Initialize Accumulators
        self.sigma_acc = self.biases * 0
        self.delta_acc = self.weights * 0
```

```python
        # Initialize Adam Paras
        self.Vdb = self.Sdb = self.sigma_acc
        self.Vdw = self.Sdw = self.delta_acc
        self.t = 1

    # ReLU Activation Function
    def relu(self, arr):
        return arr * (arr > 0)

    # Softmax Activation Function
    def softmax(self, arr):
        arr -= arr.max()
        exp = np.exp(arr)
        return exp / np.sum(exp)

    # Activation Manager Function
    def activate(self, arr):
        if self.activation == 'relu': return self.relu(arr)
        if self.activation == 'softmax': return self.softmax(arr)

    # Forward Propagation
    def step(self, vec):
        # Assign Input
        self._in = vec
        # Dot
        z = np.dot(self.weights, vec) + self.biases
        a = self.activate(z)
        # Return
        self.out = a
        return self.out

    # Back Propagation
    def back(self, grad):
        # Calculate sigma
        sigma = grad if self.activation == 'softmax' else grad * (self.out > 0)
        # Calculate delta
        delta = np.dot(sigma, self._in.T)
        # Accumulate
        self.sigma_acc += sigma
        self.delta_acc += delta
        # Return global gradient
        global_grad = np.dot(self.weights.T, sigma)
        return global_grad

    # Train
    def update(self, alpha, batch_size, beta_1 = 0.9, beta_2 = 0.99):
        # Mini-Batch Updates
        dw = self.delta_acc / batch_size; self.delta_acc *= 0
        db = self.sigma_acc / batch_size; self.sigma_acc *= 0
        # Momentum
        self.Vdw = (beta_1 * self.Vdw) + (1 - beta_1) * dw
        self.Vdb = (beta_1 * self.Vdb) + (1 - beta_1) * db
        # RMS Prop
        self.Sdw = (beta_2 * self.Sdw) + (1 - beta_2) * (dw ** 2)
        self.Sdb = (beta_2 * self.Sdb) + (1 - beta_2) * (db ** 2)
        # Corrected Momentum
        Vdw = self.Vdw / (1  - beta_1**self.t)
        Vdb = self.Vdb / (1  - beta_1**self.t)
        # Corrected RMS Prop
        Sdw = self.Sdw / (1  - beta_2**self.t)
        Sdb = self.Sdb / (1  - beta_2**self.t)
        # Update Parameters
        eps = 1e-9
        self.weights -= alpha * (Vdw / (np.sqrt(Sdw) + eps))
        self.biases -= alpha * (Vdb / (np.sqrt(Sdb) + eps))
        self.t += 0

    # Reset Adam Parameters Function
```

```
    def resetAdam(self):
        self.Vdw *= 0; self.Sdw *= 0
        self.Vdb *= 0; self.Sdb *= 0
        self.t = 1

class CNN:

    # Constructor
    def __init__(self):
        # Initialize Lists
        self.layers = []; self.cost_history = []; self.valid_cost_history = []

    # Add Layer Function
    def add(self, layer):
        self.layers.append(layer)

    # Forward Propagation
    def forward(self, img):
        out = img
        for layer in self.layers: out = layer.step(out)
        self.out = out
        return self.out

    # Back Propagation
    def backward(self, grad):
        out = grad
        for layer in reversed(self.layers):
            out = layer.back(out)

     # Train Model Function
    def train(self, X, Y, epochs = 50, alpha = 0.01, batch_size = 1000, X_valid = [],
         Y_valid = []):
        # Set Parameters
        self.alpha, self.batch_size = alpha, batch_size
        # Epoch
        for i in range(epochs):
            # Verbose
            print(f'\nEPOCH {i+1}/{epochs}')
            # Train over Dataset
            self.train_dataset(X, Y, batch_size)
            # Reset Optimizer
            for layer in self.layers:
                if isinstance(layer, Dense) or isinstance(layer, Convolution):
                 layer.resetAdam()
            # Validation Loss
            if len(X_valid) != 0 and len(Y_valid) != 0:
                valid_cost = self.cal_dataset_loss(X_valid, Y_valid)
                print(f'Vaidation Dataset Cost: {valid_cost:.3f}')
                self.valid_cost_history.append(valid_cost)

    # Train Over Dataset
    def train_dataset(self, X, Y, batch_size):
        # Total Iterations
        iters = int(len(X) / batch_size)
        # Iteration
        for i in range(iters):
            # Get batch X and Y
            start = i * batch_size
            stop = start + batch_size
            if start + batch_size <= len(X):
                batch_X = X[start:stop]; batch_Y = Y[start:stop]
            else:
                batch_X = X[start:]; batch_Y = Y[start:]
            # Train Over Batch
            self.train_batch(batch_X, batch_Y)
            # Print Batch Cost
            print(f'Iteration {i + 1}/{iters} - Cost: {self.cost_history[-1]:.3f}')
        # Print Average Dataset Cost
```

```python
        costs = self.cost_history[-iters:]
        print(f'Average Batch Cost: {np.mean(costs):.3f}')

    # Train Over Batch
    def train_batch(self, X, Y):
        # Initialize Batch Cost
        self.latest_batch_cost = 0
        # Train Batch
        for x,y in zip(X, Y):
            self.train_example(x, y)
        # Update Cost History
        self.cost_history.append(self.latest_batch_cost / self.batch_size)
        # Update Model
        self.update_model()

    # Cycle One Example
    def train_example(self, img, y):
        # Forward Prop
        pred = self.forward(img)
        # Cost
        cost = self.cross_entropy_loss(pred, y)
        self.latest_batch_cost += cost
        # Backward Prop
        error = pred - y
        self.backward(error)

    # Cross Entropy Cost Function
    def cross_entropy_loss(self, pred, y):
        pred += 1e-9
        return -np.sum(np.log(pred) * y) / pred.shape[0]

    # Dataset Cost Function
    def cal_dataset_loss(self, X, Y):
        cost = 0
        for x, y in zip(X, Y):
            pred = self.forward(x)
            cost += self.cross_entropy_loss(pred, y)
        return cost / len(X)

    # Update Model Function
    def update_model(self):
        for layer in self.layers:
            if isinstance(layer, Dense) or isinstance(layer, Convolution): layer.update(se
lf.alpha, self.batch_size)

    # Save Weights Function
    def save_weights(self, path):
        # Intialize Data
        data = {}
        for i in range(len(self.layers)):
            # Pick Layer
            layer = self.layers[i]
            # Is Conv or Dense Layer
            if not isinstance(layer, Dense) and not isinstance(layer, Convolution):
                continue
            # Get Layer Data
            weights_flat = layer.weights.flatten().tolist() if isinstance(layer, Dense)
         else layer.filters.flatten().tolist()
            weights_shape = layer.weights.shape if isinstance(layer, Dense)
         else layer.filters.shape
            biases_flat = layer.biases.flatten().tolist()
            biases_shape = layer.biases.shape
            value = (weights_flat, weights_shape, biases_flat, biases_shape)
            # Store Data
            data[i] = value
        # Save Data
        with open(path, 'w') as file:
            dump(data, file, indent = 2)
```

```python
        file.close()
        # Print
        print('Weights saved in file', path)

    # load Weights Function
    def load_weights(self, path):
        # Load Data
        with open(path) as f:
            data = load(f)
        f.close()
        # Loop through Layers
        for i in data.keys():
            # Choose Layer
            layer = self.layers[int(i)]
            # Get Layer Data
            weights_flat, weights_shape, biases_flat, biases_shape = data[i]
            weights = np.reshape(weights_flat, weights_shape)
            biases = np.reshape(biases_flat, biases_shape)
            # Assign Data to layer
            if isinstance(layer, Convolution): layer.filters = weights
            elif isinstance(layer, Dense): layer.weights = weights
            layer.biases = biases
        # Print
        print('Weights loaded from file', path)
```

Training

```
# IMPORTS
from cnn import *
import cv2
from os import listdir
import matplotlib.pyplot as plt
from sklearn.utils import shuffle

# LOAD DATASET
FOLDER_NAME = 'Dataset' # Root Folder Name
# Class Folders
folders = listdir(FOLDER_NAME)
NR_CLASSES = len(folders)
# Walk over folders
X, Y = [], []
for i in range(NR_CLASSES):
    folder = folders[i]
    images = listdir(FOLDER_NAME + '/' + folder)
    print('Folder', i+1, '-', folder, ' | Size:', len(images))
    # Walk over images
    data_X, data_Y = [], []
    for image in images:
        path = FOLDER_NAME + '/' + folder + '/' + image
        # Process Image
        size = 50
        raw = cv2.imread(path)
        gray = cv2.cvtColor(raw, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(gray, (size, size))
        # Add to class data
        data_X.append(img / 255)
        data_Y.append(np.eye(NR_CLASSES)[i].reshape([-1, 1]))
    # Add to Data
    X.append(data_X); Y.append(data_Y)

# Split training / remaining
X_train_all, Y_train_all, X_rem_all, Y_rem_all = [], [], [], []
ex_per_class = 1200
for data_X, data_Y in zip(X, Y):
    X_train_all.extend(data_X[:ex_per_class])
    Y_train_all.extend(data_Y[:ex_per_class])
    X_rem_all.extend(data_X[ex_per_class:])
    Y_rem_all.extend(data_Y[ex_per_class:])
# Sequence training
X_train_seq, Y_train_seq = [], []
for i in range(ex_per_class):
    for j in range(NR_CLASSES):
        index = i + j * ex_per_class
        X_train_seq.append(X_train_all[index])
        Y_train_seq.append(Y_train_all[index])
# Convert to numpy arrays
X_train = np.array(X_train_seq).reshape([len(X_train_seq), 1, size, size])
Y_train = np.array(Y_train_seq).reshape([len(Y_train_seq), -1, 1])
X_rem = np.array(X_rem_all).reshape([len(X_rem_all), 1, size, size])
Y_rem = np.array(Y_rem_all).reshape([len(Y_rem_all), -1, 1])
# Shuffle remaining
X_rem, Y_rem = shuffle(X_rem, Y_rem)
# Split Validation / Test
half = int(len(X_rem) / 2)
X_valid, Y_valid = X_rem[:half], Y_rem[:half]
X_test, Y_test = X_rem[half:], Y_rem[half:]
# Print
print('Total training examples:', len(X_train))
print('Total validation examples:', len(X_valid))
print('Total testing examples:', len(X_test))

# MODEL ARCHITECTURE
model = CNN()
```

```python
# Conv 1
conv1 = Convolution(16, 3, X_train[0].shape)
model.add(conv1)
# Pool 1
pool1 = Pool(3, conv1.out_shape, 3)
model.add(pool1)
# Conv 2
conv2 = Convolution(32, 5, pool1.out_shape)
model.add(conv2)
# Pool 2
pool2 = Pool(2, conv2.out_shape, 2)
model.add(pool2)
# Flat
flat = Flat()
model.add(flat)
# Hidden
hidden1 = Dense(512, pool2.size) #1152
model.add(hidden1)
# Hidden
hidden2 = Dense(64, 512)
model.add(hidden2)
# Out
out = Dense(8, 64, 'softmax')
model.add(out)

# TRAINING
model.train(X_train, Y_train, epochs = 12, alpha = 0.001, batch_size = 150,
            X_valid = X_valid, Y_valid = Y_valid)

# COST HISTORY
plt.figure(figsize = [18, 8], dpi = 150)
plt.title('Cost History', fontsize = 15)

plt.xlim([0, len(model.cost_history)])
plt.ylim([0, max(model.cost_history)+0.1])
plt.xlabel('Iteration', fontsize = 12)
plt.ylabel('Cost', fontsize = 12)

plt.plot(model.cost_history)
plt.grid()
plt.show()

# TRAIN ACCURACY
correct = 0
for x, y in zip(X_train, Y_train):
    model.forward(x)
    pred = np.argmax(model.out)
    y = np.argmax(y)
    if pred == y: correct += 1
acc = correct / len(X_train)
print(f'Train Accuracy: {acc:.3f}')

# TEST ACCURACY
correct = 0
for x, y in zip(X_test, Y_test):
    model.forward(x)
    pred = np.argmax(model.out)
    y = np.argmax(y)
    if pred == y: correct += 1
acc = correct / len(X_test)
print(f'Test Accuracy: {acc:.3f}')

# SAVE MODEL
model.save_weights('Models/model_e12_a1e-3.txt')
np.savetxt('train_cost_e12.txt', model.cost_history, '%f')
print('Training costs saved in file train_cost_e12.txt')
np.savetxt('valid_cost_e12.txt', model.valid_cost_history, '%f')
print('Validation costs saved in file valid_cost_e12.txt')
```

Evaluation

```python
# IMPORTS
from cnn import *
import cv2
import pandas as pd
import seaborn as sn
from os import listdir
import matplotlib.pyplot as plt

# CONSTANTS
CLASSES = 8
EPOCHS = 12
EX_PER_CLASS = 1200
LABELS = ['Basophil', 'Eosinophil', 'Erythroblast', 'Immunoglobulin', 'Lymphocyte',
          'Monocyte', 'Neutrophil', 'Platelet']

# DATASET
FOLDER_NAME = 'Dataset' # Root Folder Name
# Class Folders
folders = listdir(FOLDER_NAME)
NR_CLASSES = len(folders)
# Walk over folders
data_X, data_Y = [], []
for i in range(NR_CLASSES):
    folder = folders[i]
    images = listdir(FOLDER_NAME + '/' + folder)
    print('Folder', i+1, '-', folder, ' | Size:', len(images))
    # Walk over images
    for image in images:
        path = FOLDER_NAME + '/' + folder + '/' + image
        # Process Image
        size = 50
        raw = cv2.imread(path)
        gray = cv2.cvtColor(raw, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(gray, (size, size))
        # Add to class data
        data_X.append(img / 255)
        data_Y.append(np.eye(NR_CLASSES)[i].reshape([-1, 1]))
# Convert to numpy
X = np.reshape(data_X, (len(data_X), 1, size, size))
Y = np.reshape(data_Y, (len(data_Y), -1, 1))

# LOAD COSTS
train_cost_history = np.genfromtxt('Models/train_cost_e12.txt')
valid_cost_history = np.genfromtxt('Models/valid_cost_e12.txt')

# MODEL
model = CNN()
# Conv 1
conv1 = Convolution(16, 3, X[0].shape)
model.add(conv1)
# Pool 1
pool1 = Pool(3, conv1.out_shape, 3)
model.add(pool1)
# Conv 2
conv2 = Convolution(32, 5, pool1.out_shape)
model.add(conv2)
# Pool 2
pool2 = Pool(2, conv2.out_shape, 2)
model.add(pool2)
# Flat
flat = Flat()
model.add(flat)
# Hidden
hidden1 = Dense(512, pool2.size) #1152
model.add(hidden1)
```

```
# Hidden
hidden2 = Dense(64, 512)
model.add(hidden2)
# Out
out = Dense(8, 64, 'softmax')
model.add(out)
# Load Model Data
model.load_weights('models/model_e12_a1e-3.txt')
model.cost_history = train_cost_history
model.valid_cost_history = valid_cost_history

# COST HISTORY
plt.figure(figsize = [18, 8], dpi = 150)
plt.title('Cost History', fontsize = 15)

plt.xlim([0, len(model.cost_history)])
plt.ylim([0, max(model.cost_history)+0.1])
plt.xlabel('Iteration', fontsize = 12)
plt.ylabel('Cost', fontsize = 12)

plt.plot(model.cost_history)
plt.plot(np.arange(EPOCHS) * (EX_PER_CLASS / CLASSES), model.valid_cost_history)
plt.legend(['Training Cost', 'Validation Cost'], fontsize = 12)

plt.grid()
plt.show()

# CONFUSION MATRIX
matrix = np.zeros((CLASSES, CLASSES))
correct = 0
for x, y in zip(X, Y):
    out = model.forward(x)
    pred = np.argmax(out)
    true = np.argmax(y)
    if pred == true: correct += 1
    matrix[true, pred] += 1
# Accuracy
acc = correct / len(X)
print(f'Dataset Accuracy: {acc:.2%}')
Dataset Accuracy: 63.36%
Wall time: 13min 32s
normalized = matrix / np.sum(matrix, 1)
matrix_df = pd.DataFrame(normalized, index = LABELS, columns = LABELS)

plt.figure(figsize = [5, 5], dpi = 150)
sn.heatmap(matrix_df, annot = True, fmt = '.2%', annot_kws = {'fontsize':6}, cbar = False)

plt.title("Confusion Matrix", fontsize = 8)
plt.xlabel('Predicted Class', fontsize = 7)
plt.ylabel('Actual Class', fontsize = 7)

plt.xticks(fontsize = 5)
plt.yticks(fontsize = 5)
plt.show()

# PROPAGATION RESULTS
img = X[9]
out = model.forward(img)
print('Results:\n', out*100)

# Show Image
plt.figure(figsize = [3, 3], dpi = 150)
plt.xticks([])
plt.yticks([])
plt.xlabel("Input Image", fontsize = 6)

plt.imshow(img[0], 'gray')
plt.show()
```

```python
# Show Conv1 Results
out = model.layers[0].out
total = len(out)
im_per_row = 4
rows = int(total / im_per_row)

plt.figure(figsize = [16, 16], dpi = 150)
for i in range(rows):
    for j in range(im_per_row):
        index = i * im_per_row + j
        plt.subplot(rows, im_per_row, index+1)
        plt.xticks([])
        plt.yticks([])
        plt.xlabel("Map " + str(index+1), fontsize = 10)
        plt.imshow(out[index], 'gray')
plt.show()

# Show Conv2 Results
out = model.layers[2].out
total = len(out)
im_per_row = 8
rows = int(total / im_per_row)

plt.figure(figsize = [20, 12], dpi = 150)
for i in range(rows):
    for j in range(im_per_row):
        index = i * im_per_row + j
        plt.subplot(rows, im_per_row, index+1)
        plt.xticks([])
        plt.yticks([])
        plt.xlabel("Map " + str(index+1), fontsize = 10)
        plt.imshow(out[index], 'gray')
plt.show()
```

Prediction

```python
# IMPORTS
from cnn import *
import cv2
from os import listdir

# CONSTANTS
IN_SHAPE = (1, 50, 50)
LABELS = ['Basophil', 'Eosinophil', 'Erythroblast', 'Immunoglobulin', 'Lymphocyte',
          'Monocyte', 'Neutrophil', 'Platelet']
PRED_DATASET_PATH = 'Prediction Dataset/'

# MODEL
model = CNN()
# Conv 1
conv1 = Convolution(16, 3, IN_SHAPE); model.add(conv1)
# Pool 1
pool1 = Pool(3, conv1.out_shape, 3); model.add(pool1)
# Conv 2
conv2 = Convolution(32, 5, pool1.out_shape); model.add(conv2)
# Pool 2
pool2 = Pool(2, conv2.out_shape, 2); model.add(pool2)
# Flat
flat = Flat(); model.add(flat)
# Hidden
hidden1 = Dense(512, pool2.size); model.add(hidden1)
# Hidden
hidden2 = Dense(64, 512); model.add(hidden2)
# Out
out = Dense(8, 64, 'softmax'); model.add(out)
# Load Model Weights
model.load_weights('Models/model_e12_a1e-3.txt')

# LOAD PREDICTION DATASET
data, files = [], []
# Walk over Dataset
for file in listdir(PRED_DATASET_PATH):
    # Add to File Store
    files.append(file)
    # Process Image
    path = PRED_DATASET_PATH + file
    raw = cv2.imread(path)
    gray = cv2.cvtColor(raw, cv2.COLOR_BGR2GRAY)
    img = cv2.resize(gray, (50, 50))
    # Add to Dataset
    data.append(img / 255)
# Convert to numpy array
data = np.reshape(data, (len(data), 1, 50, 50))
# Print
print('Dataset Loaded\nTotal examples:', len(data))

# PREDICT
predictions = []
for i,x in enumerate(data):
    img = files[i]
    out = model.forward(x)
    pred = np.argmax(out)
    label = LABELS[pred]
    conf = out[pred][0]
    predictions.append((img, label, conf))

for val in predictions:
    img, label, conf = val
    print(f'Image: {img}\nPredicted Class: {label}\nConfidence:{conf:.2%}\n')
```