Homework 2

CS484 - Introduction to Computer Vision

Efe Acer
21602217

Bilkent University, CS

Contents

1 Part 1

1.1 Perceptron Architecture

The first part of the assignment asks us to implement a *Neural Network* with a single neuron and use it for binary classification. A *Neural Network* that consists of a single neuron and is built for binary classification is conventionally known as the *Perceptron*.

The architecture of a Perceptron; its inputs, computation path, and output are given in the figure below:

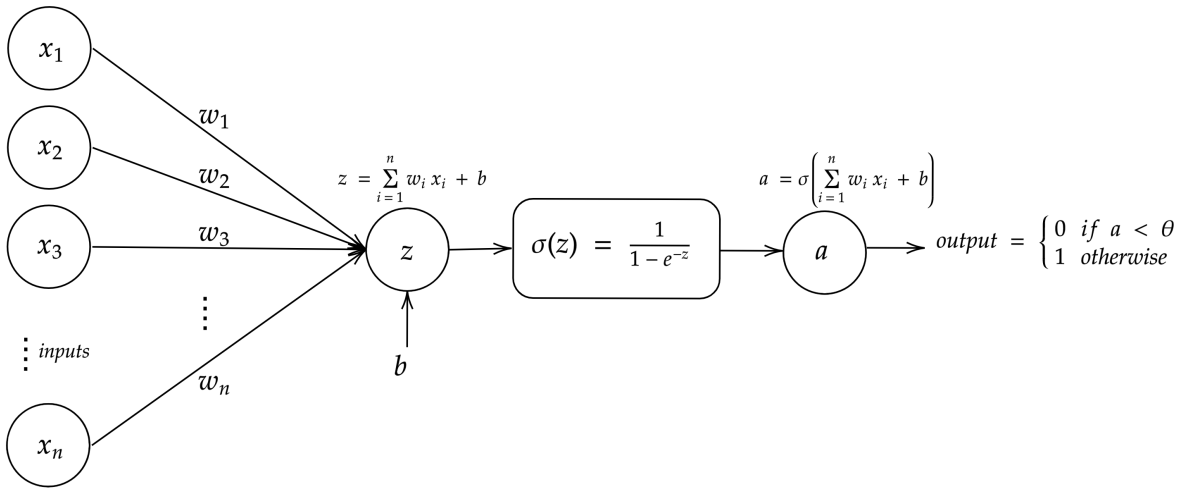

Figure 1: Perceptron Architecture

Above, x_i represents the ith feature of a data sample. The feature values are fed as inputs to the perceptron, n is the total number of features. w_i is the weight connecting the ith input to the output neuron and b is the bias of the output neuron. z is the linear combination of the inputs, weights and bias. a is obtained by passing z through an activation function, which is oftentimes a *sigmoid*, that has its formulation given in Equation (1).

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

Finally, the output is a binary integer (0 or 1), obtained by a simple threshold down operation on the value of a where the threshold constant is θ, θ is typically chosen as 0.5. Given an input data sample x, the output (or the prediction) of our perceptron can be formulated compactly as:

$$output = \begin{cases} 0 & \text{if} \quad \sigma(\sum_{i=1}^{n} w_i x_i + b) < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

1.2 The Dataset

The dataset given in this assignment is called the "cat identification dataset", which contains two set of images; one set belonging to the class of cat images, and the other belonging to the class of non-cat images. The images in the dataset are all RGB images of 64x64 resolution. The dataset includes 209 samples for training and 50 samples for testing and the labels for both sets are available.

Mathematically representing the dataset is crucial to train and use the perceptron. A prevalent representation is provided below, where n is the number of features and m is the number of samples:

$$\mathbf{X}_{nxm} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \qquad \mathbf{y}_{1xm} = [y_1, y_2, ..., y_m]$$

The matrix, $\mathbf{X}$, contains each image as a row vector. In other words, each image is flattened to a 1-dimensional vector and placed in the rows of $\mathbf{X}$. To be clearer, images in the dataset are 64x64x3 in shape including the RGB channels, after the flattening operation the shape of an image becomes 12288x1. Thus, the number of features, n, is 12288. The number of samples, m, is 209 for the training set and 50 for the test set. Columns of $\mathbf{X}$ goes through the samples and rows of $\mathbf{X}$ goes through the features.

The vector, $\mathbf{y}$, contains the label of each image, 0 for non-cat class and 1 for cat class. The label of the ith image is placed in the ith column of $\mathbf{y}$.

It is common in image classification tasks to augment the training data by rotating the original images and/or applying symmetry on them. This trick usually increases the classification accuracy and we will come to it later in this report.

Given the input matrix $\mathbf{X}$, the computation path of our perceptron can be formulated compactly in the matrix-vector form:

$$\mathbf{z}_{1xm} = \mathbf{w} \cdot \mathbf{X} + \mathbf{b} \qquad \text{,where} \quad \mathbf{w}_{1xn} = [w_1, w_2, ..., w_n] \quad \text{and} \quad \mathbf{b}_{1xm} = [b, b, ..., b]$$

$$\mathbf{a}_{1xm} = \sigma(\mathbf{z}) = \sigma(\mathbf{w} \cdot \mathbf{X} + \mathbf{b})$$

$$\mathbf{a} \xrightarrow{\text{threshold down}} \mathbf{output}_{1xm}$$

The first two steps in the calculation order given above are referred to as the *feed forward* part of the perceptron. At the end of the feed forward stage a row vector of activations $\mathbf{a}$ is obtained. Then, the **output** vector can be obtained by applying the threshold down operation on $\mathbf{a}$ with a threshold value of 0.5.

1.3 Perceptron Training

The term *learning* in the context of *Machine Learning* is nothing more than finding better *parameters* for a model. This is done through the *optimization* of a *loss function*. A *loss function* maps the values of the model parameters to a real number denoting the loss (or cost) of the task. For a binary classification task such as ours, the *log loss* function is a suitable choice. Given the ith label, y_i, and the corresponding perceptron activation, a_i, the log loss is:

$$l(y_i, a_i) = \begin{cases} -log(a_i) & \text{if} \quad y_i = 1 \\ -log(1 - a_i) & \text{if} \quad y_i = 0 \end{cases} \quad = -y_i \; log(a_i) - (1 - y_i) \; log(1 - a_i) \tag{3}$$

The rationale behind the log loss function is to penalize more as the activation diverges further from the true label. Logarithmic function does exactly this in the interval $[0, 1]$, hence the name log loss. Given the label vector $\mathbf{y}$, and the activation vector $\mathbf{a} = [a_1, a_2, ..., a_m]$ (again m is the number of samples), the overall log loss of a dataset is the average log loss of the samples:

$$L(\mathbf{y}, \mathbf{a}) = -\frac{1}{m} \sum_{i=1}^{m} \Big(y_i \; log(a_i) + (1 - y_i) \; log(1 - a_i) \Big) \tag{4}$$

Now that we have a loss function to optimize, we can chose an appropriate optimization algorithm; for this purpose we will use *Gradient Descent*. Gradient Descent is an optimization algorithm used that minimized the value of a function by iteratively moving in the direction of steepest descent, which is given by the negative of the *gradient vector*. The gradient vector simply stores the partial derivatives of the loss function with respect to the model parameters:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} & \frac{\partial L}{\partial w_2} & \cdots & \frac{\partial L}{\partial w_n} & \frac{\partial L}{\partial b} \end{bmatrix} \tag{5}$$

First n entries of the gradient vector defines the update rules of the weights, whereas the last entry entry defines the update rule of the bias. To derive these update rules, we use the chain rule:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a_i} \; \frac{\partial a_i}{\partial z_i} \; \frac{\partial z_i}{\partial w_i} \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial a_i} \; \frac{\partial a_i}{\partial z_i} \; \frac{\partial z_i}{\partial b} \tag{6}$$

We proceed by calculating $\frac{\partial L}{\partial a_i}$:

$$\frac{\partial L}{\partial a_i} = -\frac{1}{m} \Big(\frac{y_i}{a_i} + \frac{y_i - 1}{1 - a_i} \Big) = \frac{a_i - y_i}{m \; a_i (1 - a_i)} \tag{7}$$

Then, we compute $\frac{\partial a_i}{\partial z_i}$, which is indeed the first order derivative of the sigmoid function:

$$\frac{\partial a_i}{\partial z_i} = \sigma'(z_i) = \frac{-e^{-z}}{(1-e^{-z})^2} = \frac{1}{1-e^{-z}} \; \frac{-e^{-z}}{1-e^{-z}} = \sigma(z_i) \; (1-\sigma(z_i)) = a_i \; (1-a_i) \tag{8}$$

Finally, we compute the remaining two partial derivatives:

$$\frac{\partial z_i}{\partial w_i} = x_i \quad \text{and} \quad \frac{\partial z_i}{\partial b} = 1 \tag{9}$$

Now that we have all the necessary derivatives, we can expand Equation (6) and write:

$$\frac{\partial L}{\partial w_i} = \frac{(a_i - y_i) \; x_i}{m} \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{a_i - y_i}{m} \tag{10}$$

In the gradient descent algorithm, we define a constant η, called the *learning rate* which defines the magnitude of the step we take in the negative gradient direction. Learning rate prevents the algorithm from diverging. Hence, the update rules in the algorithm are:

$$w_i \longleftarrow w_i - \eta \; \frac{\partial L}{\partial w_i} \quad \text{and} \quad b \longleftarrow b - \eta \; \frac{\partial L}{\partial b} \tag{11}$$

Number of iterations to perform in the gradient descent algorithm is also a predetermined constant, some people also add control statements to terminate the algorithm when the change in the loss function's value becomes insignificant.

One iteration of the gradient descent algorithm goes through every sample in the dataset and moves in the direction of the average negative gradient of all samples. There are also probabilistic versions of the algorithm such as *Stochastic Gradient Descent* or *Batch Gradient Descent* that does not go through every sample. A single iteration of the algorithm is given below in the compact matrix form:

1: $\mathbf{z} = \mathbf{w} \cdot \mathbf{X} + \mathbf{b}$
2: $\mathbf{a} = \sigma(\mathbf{z})$
3: $d\mathbf{z} = \mathbf{a} - \mathbf{y}$
4: $d\mathbf{w} = \frac{1}{m} \; \mathbf{X} \cdot d\mathbf{z}^T$
5: $d\mathbf{b} = mean(\mathbf{z})$
6: $\mathbf{w} \longleftarrow \mathbf{w} - \eta \; d\mathbf{w}$
7: $\mathbf{b} \longleftarrow \mathbf{b} - \eta \; d\mathbf{b}$

Lines 3-7 above are usually referred to as the *backpropagation* stage of perceptron's training. Note that we have defined $\mathbf{w}$ as a row vector instead of a column vector so we do not need to take its transpose as it is taken in the assignment sheet's formulation.

The iteration above can also be implemented iteratively instead of using matrix operations. In that case, loops must be written explicitly to add up the gradient of each sample, then the average of the added values must be used in the updates.

1.4 Perceptron Implementation

The steps we have described in the previous sections are implemented in `Python` using the scientific computing library `Numpy`, which provides efficient implementations of matrix operations.

First step of the implementation was to write a class for our perceptron where the parameters of the model are stored so that the architecture is represented properly. The constructor of this class is given below:

```
class Perceptron:
    """
    Represents a Neural Network that consists of a single neuron and is
    built for binary classification.
    """

    def __init__(self, X_train, y_train, X_test, y_test, eta=1e-2, num_iters=1000,
         seed=0):
        """
        Constructor of the Perceptron class, initializes the model parameters and
        hyper-parameters, stores the datasets and applies preprocessing if needed.
        Args:
            X_train: the train samples
            y_train: the train labels
            X_test: the test samples
            y_test: the test labels
            eta: learning rate (default is 0.01)
            num_iters: number of gradient descent iterations (default is 1000)
            seed: seed value for random generator (default is 0)
        """
        np.random.seed(seed) # seet the random seed for reproducibility

        # Initialize the training dataset (n: number of features,
        # m_train: number of training samples, m_test: number of test samples)
        self.X_train = X_train # n x m_train input matrix
        self.y_train = y_train # 1 x m_train label vector
        self.X_test = X_test # n x m_test input matrix
        self.y_test = y_test # 1 x m_test label vector
        self.n, self.m_train = self.X_train.shape
        self.m_test = len(self.y_test)

        # Initialize the model parameters
        # (random values are divided by 100 for faster convergence)
        self.w = np.random.rand(self.n) / 100 # 1 x n weight vector
        self.b = np.random.rand() / 100 # scalar bias
```

```
35
36          # Initialize the hyper-parameters
37          self.eta = eta # learning rate
38          self.num_iters = num_iters # number of gradient descent iterations
39
40          # Store the loss/accuracy histories
41          self.losses_train = []
42          self.losses_test = []
43          self.accuracies = []
```

The last three lines of the constructor defines arrays to record the training and test losses, and the classification accuracy at each step of the training process.

One important note is that the input matrices `X_train` and `X_test` must be flattened `numpy` arrays of the shape nxm as explained in Section 1.2. The code below performs this flattening:

```
1 train_x_flat = train_x.reshape(train_x.shape[0], -1).T.astype('float64')
2 test_x_flat = test_x.reshape(test_x.shape[0], -1).T.astype('float64')
```

As the previous sections emphasize, sigmoid and log loss are the two essential functions we use in the training procedure. Hence, these functions are embedded into the class as primitives. Their implementations are provided below:

```
1  def sigmoid(self, z):
2      """
3      Implementation of the sigmoid function.
4      Args:
5          z: input to the sigmoid
6      Returns:
7          The value of the sigmoid function
8      """
9      z = np.clip(z, -700, 1000) # to avoid numeric overflows
10     return 1 / (1 + np.exp(-z))
11
12 def log_loss(self, a, y):
13     """
14     Given the activation vector and true labels, calculates the log loss value.
15     Args:
16         a: the activation vector
17         y: the true labels
18     Returns:
19         The log loss value
20     """
21     epsilon = 1e-20 # to avoid NaN values in np.log(0)
22     return np.mean(-y * np.log(a + epsilon) - (1 - y) * np.log(1 - a + epsilon))
```

Some numeric problems may arise in the naive implementations of these functions. The first line of the `sigmoid` function uses `np.clip` function that clips the input signal to a certain range to avoid exponent overflows for relatively large/small inputs. Similarly, the first line of the `log_loss` function defines a tiny constant, epsilon, that is added inside every `np.log` call to avoid Not-A-Number errors, which are raised when `np.log` is called with a zero input.

Now that the parameters are stored and the primitive functions are available, we give the implementation of the gradient descent algorithm in matrix form, as given in Section 1.3:

```
def train(self, verbose=True):
    """
    The gradient descent algorithm in matrix form, which iteratively updates the
    weights and biases in the direction of the steepest descent in the loss
        function.
    Args:
        verbose: whether printing is on or off (default is True)
    """
    for i in range(self.num_iters):

        # Feed forward
        z = self.w @ self.X_train + self.b
        a = self.sigmoid(z)

        # Compute train and test losses, and accuracy
        self.losses_train.append(self.log_loss(a, self.y_train))
        self.losses_test.append(self.log_loss(
            self.sigmoid(self.w @ self.X_test + self.b), self.y_test))
        self.accuracies.append(self.predict())

        # Back propagate
        dz = a - self.y_train
        self.w -= self.eta * (self.X_train @ dz) / self.m_train # update rule for
            weights
        self.b -= self.eta * np.mean(dz) # update rule for bias

        if verbose and i % 50 == 0:
            print('---\nPerformed iteration %d:\ncurrent train loss = %f\ncurrent
                test loss = %f\n'
                  % (i + 1, self.losses_train[-1], self.losses_test[-1]))
```

As it was discussed earlier, each iteration of the method above can be implemented in an iterative manner where we process a single sample at a time in an additional for-loop. To be clearer, the additional for-loop adds up the gradient of each sample, and the total gradient is averaged at the end of the loop before applying the parameter updates. Losses are also computed in the same manner. The implementation of this single sample form follows:

```
def train_iterative(self, verbose=True):
    """
    The gradient descent algorithm in iterative form, which iteratively updates
        the
    weights and biases in the direction of the steepest descent in the loss
        function.
    Args:
        verbose: whether printing is on or off (default is True)
    """
    for i in range(self.num_iters):

        loss_train = 0
        dw = np.zeros(self.n)
        db = 0

        # Compute the gradient
        for j in range(self.m_train):
            z = self.w @ self.X_train[:, j] + self.b
            a = self.sigmoid(z)
            loss_train += self.log_loss(a, self.y_train[j])
            dz = a - self.y_train[j]
            dw += self.X_train[:, j] * dz
            db += dz

        loss_train /= self.m_train
        dw /= self.m_train
        db /= self.m_train

        # Store train and test losses, and accuracy
        self.losses_train.append(loss_train)
        self.losses_test.append(self.log_loss(
            self.sigmoid(self.w @ self.X_test + self.b), self.y_test))
        self.accuracies.append(self.predict())

        self.w -= self.eta * dw
        self.b -= self.eta * db

        if verbose and i % 50 == 0:
            print('---\nPerformed iteration %d:\ncurrent train loss = %f\ncurrent
                test loss = %f\n'
                  % (i + 1, self.losses_train[-1], self.losses_test[-1]))
```

Finally, a `predict` method is added to the class to feed the test data forward, apply threshold down and compute the classification accuracy:

```
def predict(self):
    """
    Feeds the test input forward and applies threshold down operation to
    the resulting activations to obtain the class predictions (outputs).
    Args:
        verbose: whether printing is on or off (default is True)
    Returns:
        Classification accuracy (proportion of correct predictions)
    """
    output = self.sigmoid(self.w @ self.X_test + self.b)
    output[output >= 0.5] = 1
    output[output < 0.5] = 0
    return np.sum(self.y_test == output) / self.m_test
```

1.5 Initial Results (Raw Data)

The initial results were obtained as specified in the assignment sheet, in other words a total of 50 epochs are run, and the learning rate is set to 0.00001. Note that an epoch is a complete pass over the entire data, since the gradient descent algorithm passes over the entire data in each iteration we can use the terms epoch and iteration interchangeably. The initial results were as follows:

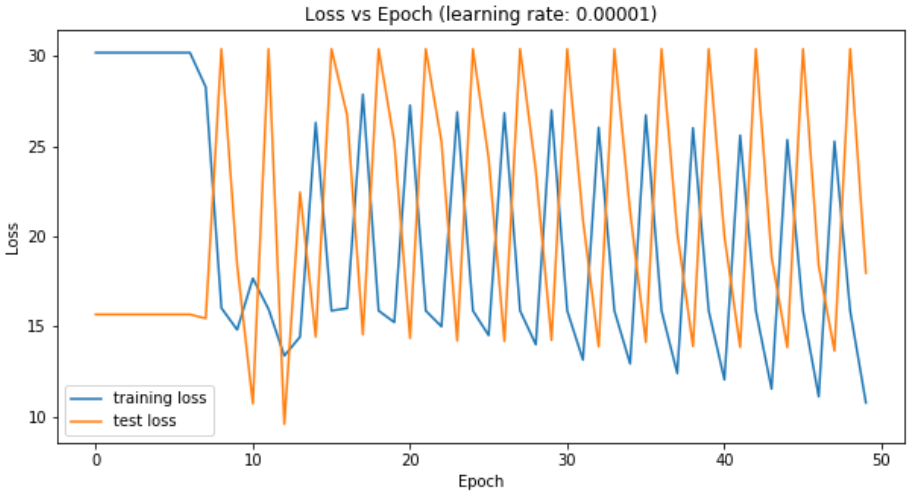

Figure 2: Loss vs Epoch (Raw Data, $\eta = 0.00001$)

```
Prediction accuracy = 68%
```

This result was obtained using the Gradient Descent algorithm implemented in matrix form; the runtime is measured with `Jupyter Notebook`'s `%%time` command, the wall time was 101ms. The single sample (iterative) implementation produced the exact same result, however the wall time was measured to be 720ms; which concludes that matrix form is the better implementation.

Figure 2 shows a decreasing trend in terms of the loss value, however there are large stochastic jumps. This typically happens when the data is fed into the perceptron in raw format, and the way to overcome it is to normalize or scale the features. For this purpose, four normalizing/scaling methods are implemented and tested which will be explained in the next part of this report. Several more values for η is used but none of them gives a better decrease with less jumps.

2 Part 2

2.1 Data Augmentation

Oftentimes in machine learning, the size of the given dataset is not enough to achieve a satisfactory classification accuracy. However, we can boost the performance of our model by augmenting the data we already have; which is indeed producing samples by applying minor changes to the original ones. These changes are in fact very intuitive transformations for image data, which include rotations and symmetries; some of which are shown below:

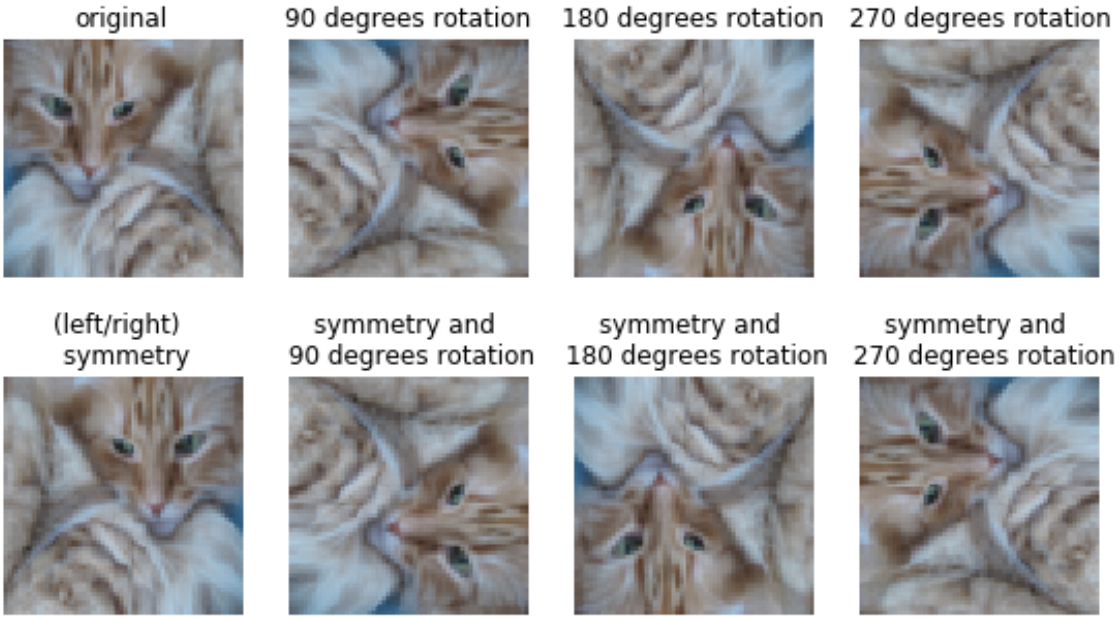

Figure 3: Some Transformations to Augment Image Data

There are fewer cat class images (72) than there are non-cat class images (132) in our training set. Thus, it is logical to add more cat class images using some of the transformations given in Figure 3. The method below is implemented for this purpose:

```
1  def augment_data(data_x, data_y):
2      """
3      Augments the given dataset by adding basic transformations (rotations and
           symmetries)
4      of cat-class images.
5      Args:
6          data_x: The data samples (must be 64x64x3)
7          data_y: The data labels
8      Returns
9          data_x: Augmented data samples
10         data_y: Augmented data labels
11     """
12     num_samples = len(data_y)
13     for i in range(num_samples):
14         if data_y[i] == 1:
15             original = data_x[i]
16             symmetry = np.fliplr(original)
17             augment_arr = [np.rot90(np.rot90(np.rot90(original))),
18                            np.rot90(np.rot90(original)),
19                            np.rot90(original),
20                            symmetry,
21                            #np.rot90(np.rot90(np.rot90(symmetry))),
22                            #np.rot90(np.rot90(symmetry)),
23                            #np.rot90(symmetry)
24                           ]
25             data_x = np.append(data_x, augment_arr, axis=0)
26             data_y = np.append(data_y, np.ones(len(augment_arr)))
27     return data_x, data_y
```

Lines 21-23 are commented in the function above because when added those transformations resulted in a lower classification accuracy. With the commented lines the function above applies all of the rotations given in Figure 3, and a single left/right symmetry operation.

Note that the method above requires the training data to be 64x64x3 in shape so that the transformations can properly work. Thus, this method should be called before the flattening step:

```
1  train_x_aug, train_y_aug = augment_data(train_x, train_y)
2  train_x_aug_flat = train_x_aug.reshape(train_x_aug.shape[0], -1).T.astype('float64
       ')
```

2.2 Feature Scaling (Normalization)

It is common-practice to apply normalization in any machine learning task since gradient descent converges much slower without it. The rationale behind applying normalization is that the range of features of raw data varies so much that loss functions do not work properly when the range of all features is not scaled such that each feature contributes approximately proportionately. There exists several methods to normalize data, we have implemented and tried four of them:

Standardization

Arguably the most common way to normalize data. Feature standardization makes the values of each feature have zero mean and unit-variance:

$$x = \frac{x - \bar{x}}{S} \tag{12}$$

The implementation is given below:

```
def standardize(X):
    """
    Applies standardization to a given dataset.
    Args:
        X: the given dataset
    Returns:
        X: the standardized normalized dataset
    """
    n = X.shape[0]
    mean = np.mean(X, axis=1).reshape(n, 1)
    std = np.std(X, axis=1).reshape(n, 1)
    std[std == 0] = 1e-20 # to avoid divide by zero
    X -= mean
    X /= std
    return X
```

Mean Normalization

The transformation for mean-normalization is given below:

$$x = \frac{x - \bar{x}}{max(x) - min(x)} \tag{13}$$

The implementation follows:

```
def mean_normalize(X):
    """
    Applies mean normalization to a given dataset.
```

```
    Args:
        X: the given dataset
    Returns:
        X: the mean normalized dataset
    """
    n = X.shape[0]
    mean = np.mean(X, axis=1).reshape(n, 1)
    min_ = np.min(X, axis=1).reshape(n, 1)
    max_ = np.max(X, axis=1).reshape(n, 1)
    diff = max_ - min_
    diff[diff == 0] = 1e-20 # to avoid divide by zero
    X -= mean
    X /= diff
    return X
```

Min-Max Scaling

The transformation for min-max scaling is:

$$x = \frac{x - \min x}{max(x) - min(x)} \tag{14}$$

The implementation is:

```
def min_max_scale(X):
    """
    Applies min-max scaling to a given dataset.
    Args:
        X: the given dataset
    Returns:
        X: the min-max scaled dataset
    """
    n = X.shape[0]
    min_ = np.min(X, axis=1).reshape(n, 1)
    max_ = np.max(X, axis=1).reshape(n, 1)
    diff = max_ - min_
    diff[diff == 0] = 1e-20 # to avoid divide by zero
    X -= min_
    X /= diff
    return X
```

Robust Scaling

Robust scaling is similar to min-max scaling, but it is more robust to outliers. This is the case because the first and the third quantiles, q_1 and q_3 are used instead of the end points:

$$x = \frac{x - q_1}{q_3 - q_1} \tag{15}$$

The implementation follows:

```
def robust_scale(X):
    """
    Applies robust scaling to a given dataset.
    Args:
        X: the given dataset
    Returns:
        X: the robust scaled dataset
    """
    n = X.shape[0]
    q1 = np.quantile(X, 0.25, axis=1).reshape(n, 1)
    q3 = np.quantile(X, 0.75, axis=1).reshape(n, 1)
    diff = q3 - q1
    diff[diff == 0] = 1e-20 # to avoid divide by zero
    X -= q1
    X /= diff
    return X
```

Being tried each one of the methods above, the best-performing one in terms of classification accuracy was mean normalization. We apply the method to the input matrices of the training and test sets:

```
train_x_aug_flat_scaled = mean_normalize(train_x_aug_flat)
test_x_flat_scaled = mean_normalize(test_x_flat)
```

This concludes our data pre-processing pipeline:

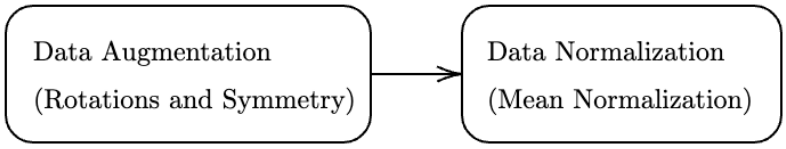

Figure 4: Data Pre-processing Pipeline

2.3 Results After Data Pre-processing

In this part, we try to find the best hyper-parameters for our classifier. A hyper-parameter is defined as a parameter which is fixed before the training of a model. In our case, our hyper-parameters are the learning rate, η, and number of epochs. Other models may include additional hyper-parameters such as regularization constants and so on.

We try three values for η; 0.1, 0.01, and 0.001. We record the classification accuracy together with training and test losses for each iteration and we let the optimization run for a total of 1000 iterations. Following are the results for $\eta = 0.1$:

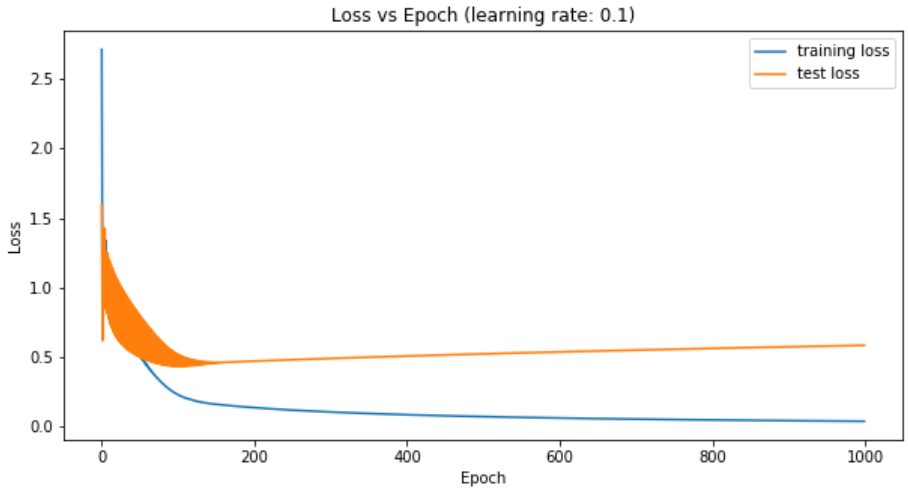

Figure 5: Loss vs Epoch (Pre-processed Data, $\eta = 0.1$)

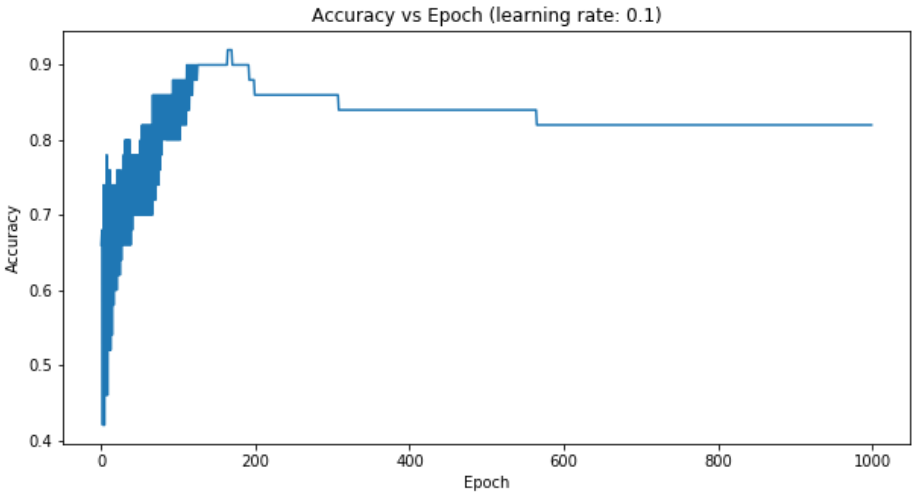

Figure 6: Accuracy vs Epoch (Pre-processed Data, $\eta = 0.1$)

Looking at Figure 5, we can see that the training loss and test loss diverge away from each other as we continue our iterations. This is a strong indicator of overfitting, which means that we are learning the details and the noise in the training data. Figure 6 shows a maximum accuracy of **92%** around the 165th epoch. This is a relatively high accuracy, however we can say that the sudden jumps in the optimization led to this value, since the accuracy drops quickly as we iterate a little more. We can conclude that $\eta = 0.1$ is way too big for the model to generalize as we perform more iterations.

We move on to $\eta = 0.01$:

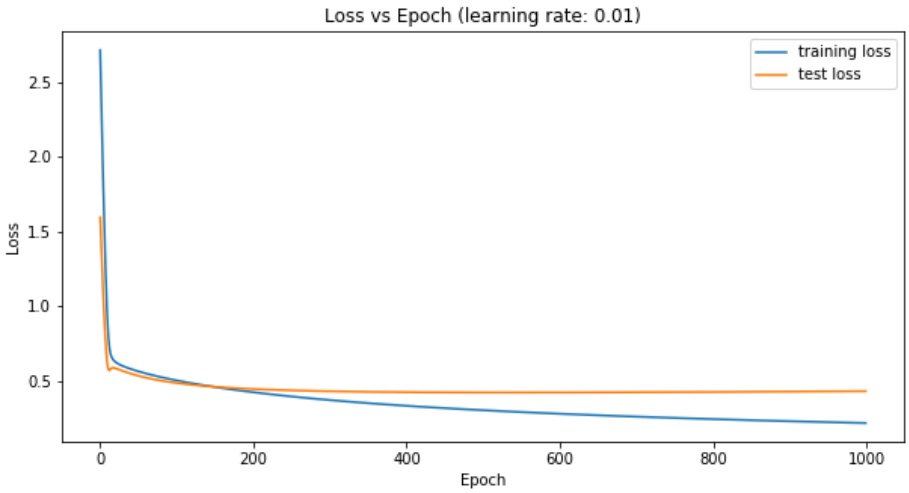

Figure 7: Loss vs Epoch (Pre-processed Data, $\eta = 0.01$)

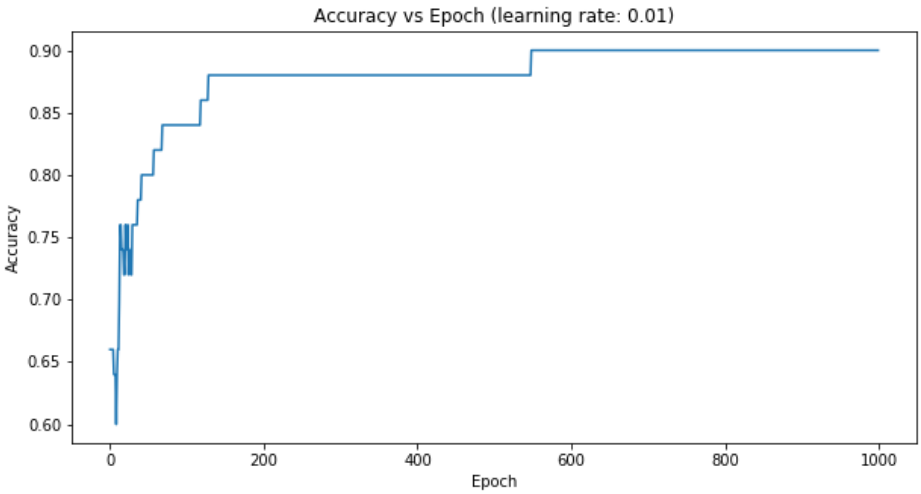

Figure 8: Accuracy vs Epoch (Pre-processed Data, $\eta = 0.01$)

Inspecting Figure 7, we can say that the training and test loss do not diverge away from each

other, they actually decrease together with a decreasing rate; this is an indicator that we are not overfitting. Figure 8 clearly shows that the accuracy increases as we iterate, which implies that we choose a good value for η and the model is indeed learning. The accuracy is **90%** for this configuration, which is two percent less than the highest accuracy of the previous configuration, however the overall training process is now smoother since there are no sudden jumps in any of the plots.

We try one more value for η, which is 0.001:

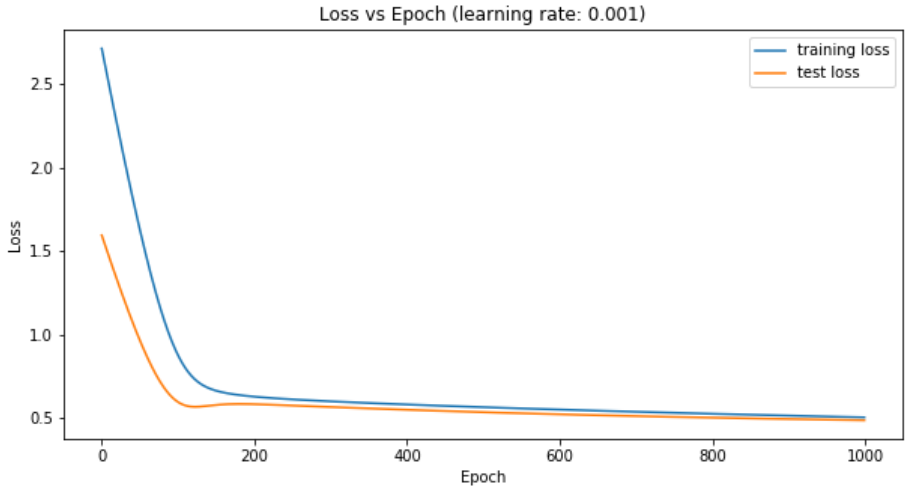

Figure 9: Loss vs Epoch (Pre-processed Data, $\eta = 0.001$)

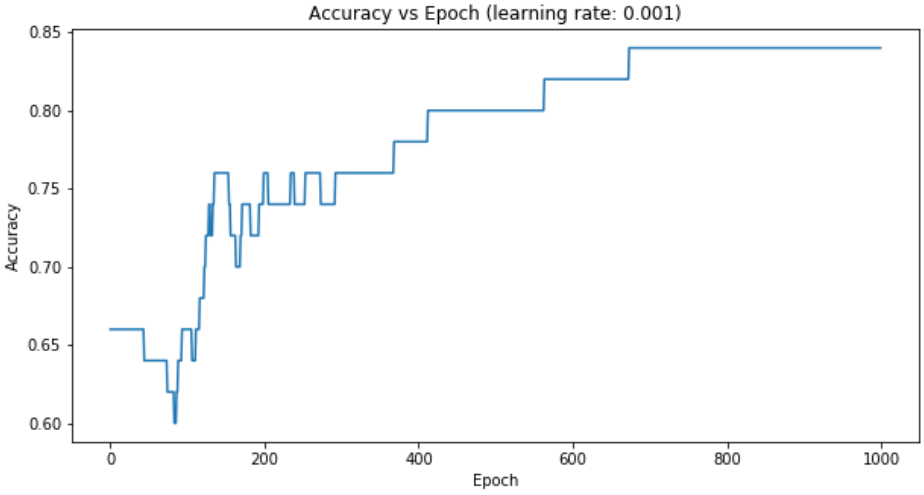

Figure 10: Accuracy vs Epoch (Pre-processed Data, $\eta = 0.001$)

From Figure 9, we can infer that the training loss and test loss are too close to each other as if they were sticked together. This indicates that we are still in the early phases of the learning

although we have reached the 1000th iteration. We need to keep iterating for the losses to diverge away from each other a little so that the model can generalize. Figure 10 shows a similar trend with Figure 8, but there are more fluctuations since reducing the learning rate is similar to zooming to the earlier iterations of Figure 8. For this value of learning rate, the highest accuracy is **84%**.

In the table below; the best hyper-parameters in terms of interpretability are shown in light blue, whereas the best hyper-parameters in terms of numbers is shown in yellow:

Learning Rate (η)	Number of epochs	Classification Accuracy
0.1	165	**92%**
0.01	1000	**90%**
0.001	1000	**84%**

Thus, the following cell should run in order to obtain the highest classification accuracy:

```
best_perceptron = Perceptron(train_x_aug_flat_scaled, train_y_aug,
                             test_x_flat_scaled, test_y, eta=1e-1, num_iters=165)
best_perceptron.train()
print('Highest prediction accuracy (learning rate: 0.1) = %d%%' % (100 *
    best_perceptron.predict()))
```

Overall, we can conclude that we have significantly increased the classification accuracy by adding meaningful steps to our data processing pipeline. Our accuracy on the raw data was 68%, which is far less than what we now achieved. Note that this accuracy is reached using a single neuron and the simplest possible neural network architecture. A multi-layer perceptron would perform a lot better, and a deep neural network would get the best of this task.

3 Part 3

3.1 Convolutional Neural Networks

A *Convolutional Neural Network* (ConvNet or CNN for short) is a specific type of neural network that performs particularly well in image recognition and classification. CNNs are not fully connected in each layer, and they introduce additional operations on top of regular neural networks. These operations are convolution (filtering), padding, stride and pooling. Different CNN architectures exist where these operations are assigned to layers and these layers are arranged accordingly.

3.2 Editing an Existing CNN

The first task of this part is to download and modify the code given in the link:
`https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py`.

The code in the link defines a basic CNN architecture, which we call the Basic ConvNet, and trains it to classify handwritten digits in the famous MNIST dataset. We aim to edit the code in a way that Basic ConvNet starts to perform binary classification on our cat dataset. Doing this is relatively easy, we just perform the following changes:

- Set the training and test data to cat dataset instead of MNIST.
- Set image dimensions to 64x64 (which was 28x28)
- Set number of classes in our dataset, 2 (which was 10)
- Set batch size to the number of training samples in our data, 209 (which was 128)
- Set number of color channels to 3 (which was 1)

The code for this part is written in Google Colab, which is essentially a free cloud service that supports a certain amount of GPU power. You can run Python notebooks in Colab and access libraries such as Keras and TensorFlow. Google Colab is intentionally preferred for this part since version conflicts arise often when these libraries are used in local machines.

Google Colab can be mounted to Google Drive, and the Colab notebooks can read data this way. To read our training and test data, the following lines should run in Colab:

```
train_path = '/content/drive/My Drive/CS484HW2/train_catvnoncat.h5'
test_path = '/content/drive/My Drive/CS484HW2/test_catvnoncat.h5'

train = h5py.File(train_path, 'r')
x_train = np.array(train['train_set_x'])
y_train = np.array(train['train_set_y'])
```

```
7
8  test = h5py.File(test_path, 'r')
9  x_test = np.array(test['test_set_x'])
10 y_test = np.array(test['test_set_y'])
```

Note that the path declarations should comform with the location of the data in the drive, also `h5py` library should have been imported.

After setting the dataset properly, the following code runs to set the image dimensions, number of classes and the batch size:

```
1 batch_size, img_rows, img_cols, _ = x_train.shape
2 num_classes = 2 # cat and non-cat
```

Finally, the channel number is changed from 1 to 3 by running:

```
1 if K.image_data_format() == 'channels_first':
2     x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
3     x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
4     input_shape = (3, img_rows, img_cols)
5 else:
6     x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
7     x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
8     input_shape = (img_rows, img_cols, 3)
```

After these changes, we train the model, with the default **ADAdelta** optimizer, for **12 epochs**, and get an accuracy of **34%** and a test loss of **0.6777**.

3.3 Basic ConvNet Architecture

In Keras, the `model.summary()` command can be used to print a summary of the network architecture. This command produces the following output for Basic ConvNet:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_1 (Conv2D) (None, 62, 62, 32) 896
_________________________________________________________________
conv2d_2 (Conv2D) (None, 60, 60, 64) 18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 30, 64) 0
_________________________________________________________________
dropout_1 (Dropout) (None, 30, 30, 64) 0
```

```
_________________________________________________________________
flatten_1 (Flatten) (None, 57600) 0
_________________________________________________________________
dense_1 (Dense) (None, 128) 7372928
_________________________________________________________________
dropout_2 (Dropout) (None, 128) 0
_________________________________________________________________
dense_2 (Dense) (None, 2) 258
=================================================================
Total params: 7,392,578
Trainable params: 7,392,578
Non-trainable params: 0
```

The architecture of the Basic ConvNet can also be illustrated with the figure below:

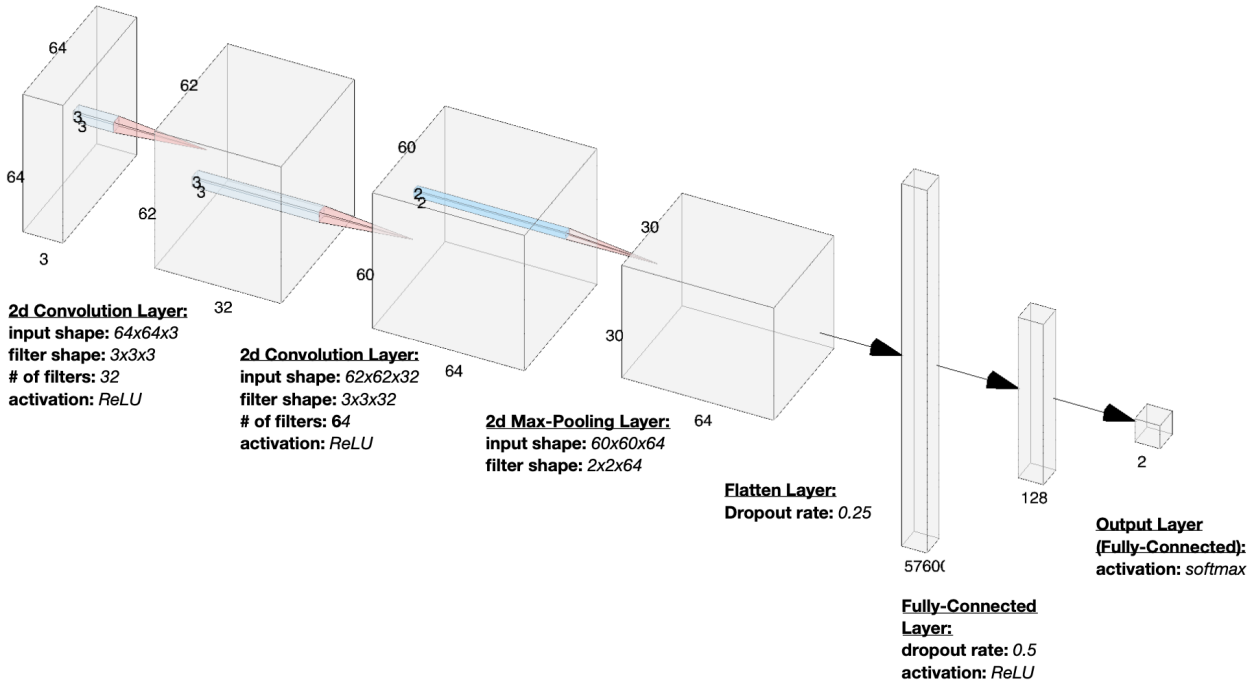

Figure 11: Basic ConvNet Architecture

3.4 Effect of Epoch

In order to investigate the change of total epochs on the accuracy, we increase the epoch number from 12 to 30. We keep other hyper-parameters the same and retrain the model. We also use TensorBoard, which is a built-in parameter logging and visualizing service of TensorFlow, to plot the test accuracies and test losses during training:

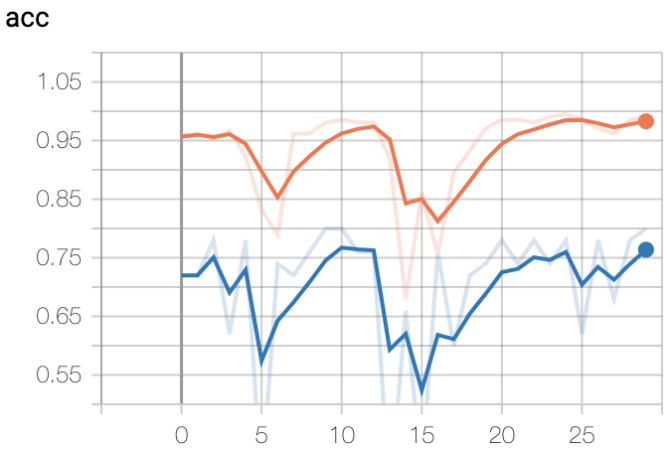

Figure 12: Test Loss vs Epoch, **ADA delta**, **30 epochs**

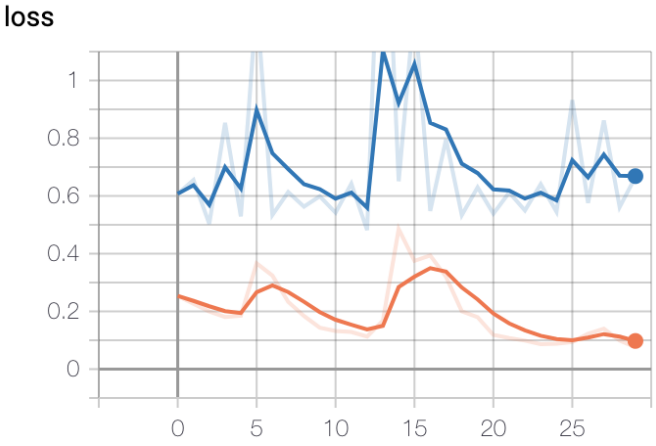

Figure 13: Test Loss vs Epoch, **ADA delta**, **30 epochs**

Results for: **ADA delta** optimizer, **30 epochs**

```
Test loss: 0.6656003975868225
Test accuracy: 0.8000000047683716
```

As an expected result, we optimized more as we kept iterating, the accuracy is **80%** in this case and the loss is **0.665**.

3.5 Effect of Optimizer

We now investigate how changing the optimizer affects the model performance. We use *Stochastic Gradient Descent* (SGD) now, which is a randomized version of the regular Gradient Descent algorithm. SGD oftentimes converges faster than the usual Gradient Descent. As instructed, we train the model using **SGD**, with a learning rate of **0.001** for **30 epochs**:

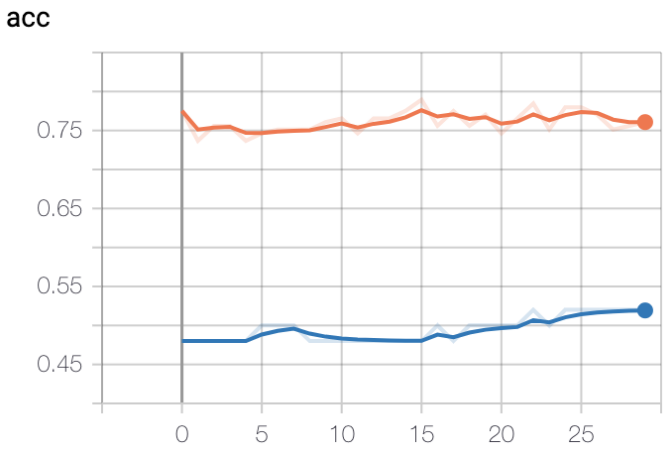

Figure 14: Test Loss vs Epoch, **SGD**, **0.001 learning rate**, **30 epochs**

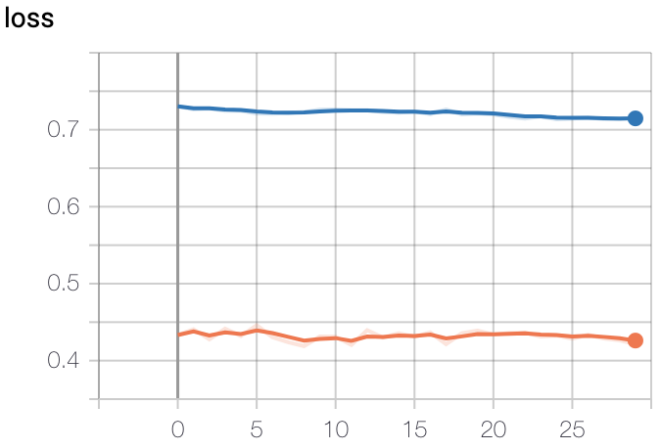

Figure 15: Test Loss vs Epoch, **SGD**, **0.001 learning rate**, **30 epochs**

Results for: **SGD** optimizer, **0.001 learning rate**, **30 epochs**:

```
Test loss: 0.7150398921966553
Test accuracy: 0.5200000095367432
```

We ended up with a higher loss of **0.715** and a lower accuracy of **52%**, but looking at the plots we can say that the algorithm actually did not even begin optimizing. We can make

this comment since the curves are all nearly flat. Thus, tuning the learning rate is expected to provide better results.

3.6 Effect of Learning Rate

Now, we try five different values for the learning rate hyper-parameter to find out the optimal. The set of learning rates we have run the model with is **[0.01, 0.0125, 0.015, 0.0175, 0.02]**. We also increase the epochs to **50 epochs** to move closer towards convergence. The plots and the results follow:

Learning Rate = 0.01

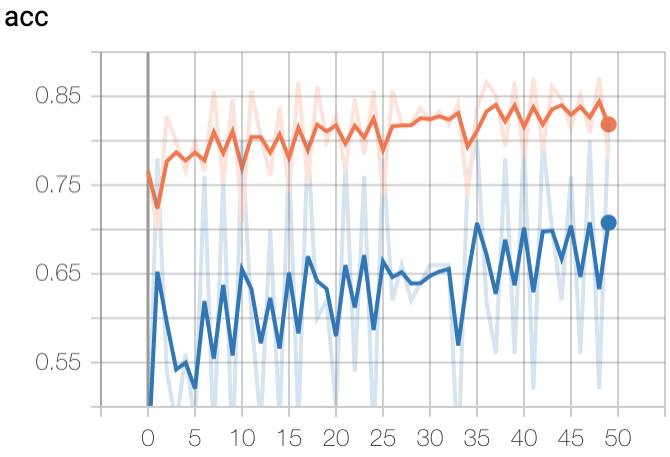

Figure 16: Test Loss vs Epoch, **SGD**, **0.01 learning rate**, **50 epochs**

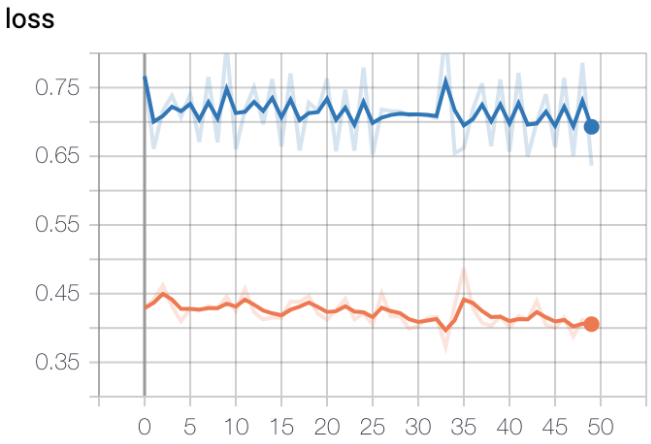

Figure 17: Test Loss vs Epoch, **SGD**, **0.01 learning rate**, **50 epochs**

Results for: **SGD** optimizer, **0.01 learning rate**, **50 epochs**:

```
Test loss: 0.6364989471435547
Test accuracy: 0.8200000023841858
```

Learning Rate = 0.0125

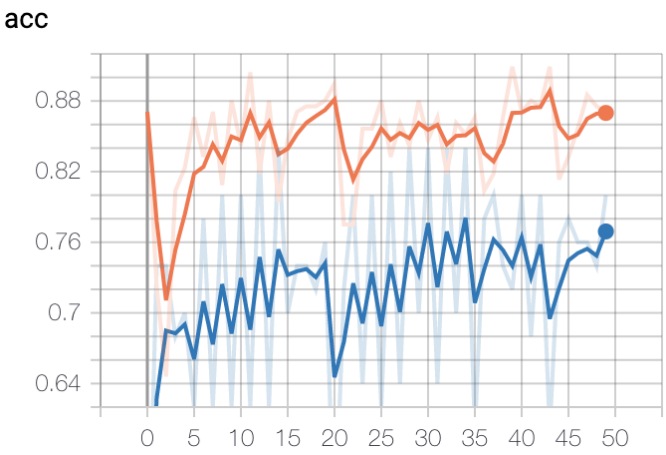

Figure 18: Test Loss vs Epoch, **SGD**, **0.0125 learning rate**, **50 epochs**

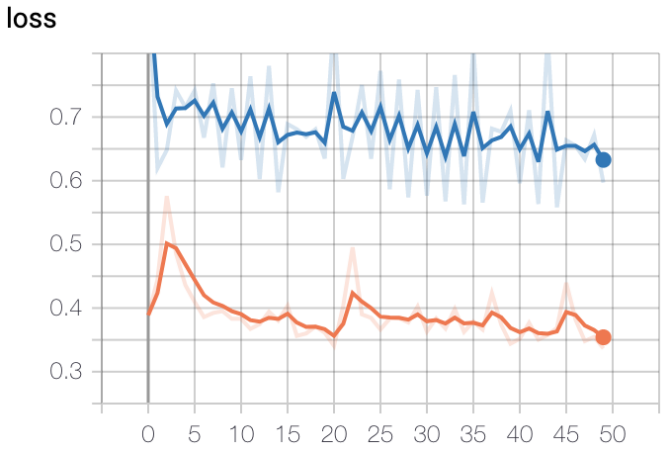

Figure 19: Test Loss vs Epoch, **SGD**, **0.0125 learning rate**, **50 epochs**

Results for: **SGD** optimizer, **0.0125 learning rate**, **50 epochs**:

```
Test loss: 0.5969989418983459
Test accuracy: 0.8000000023841858
```

Learning Rate = 0.015

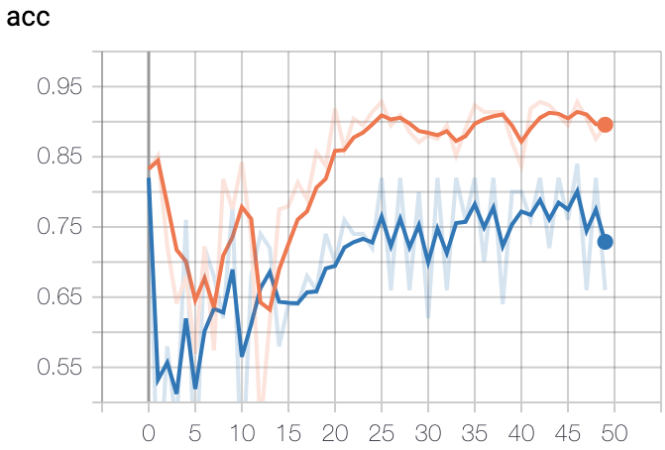

Figure 20: Test Loss vs Epoch, **SGD**, **0.015 learning rate**, **50 epochs**

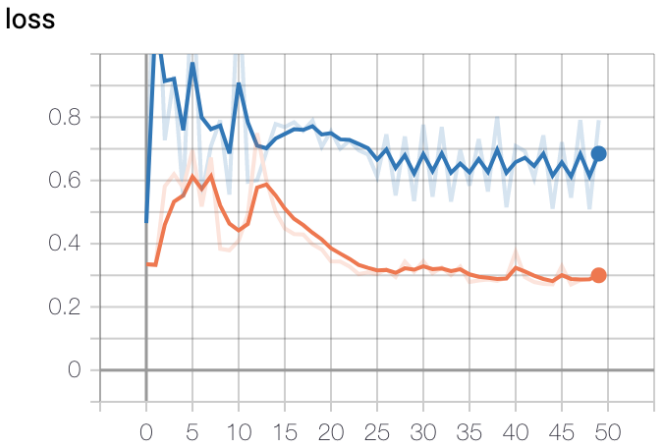

Figure 21: Test Loss vs Epoch, **SGD**, **0.015 learning rate**, **50 epochs**

Results for: **SGD** optimizer, **0.015 learning rate**, **50 epochs**:

```
Test loss: 0.7906731963157654
Test accuracy: 0.6599999952316284
```

Learning Rate = 0.0175

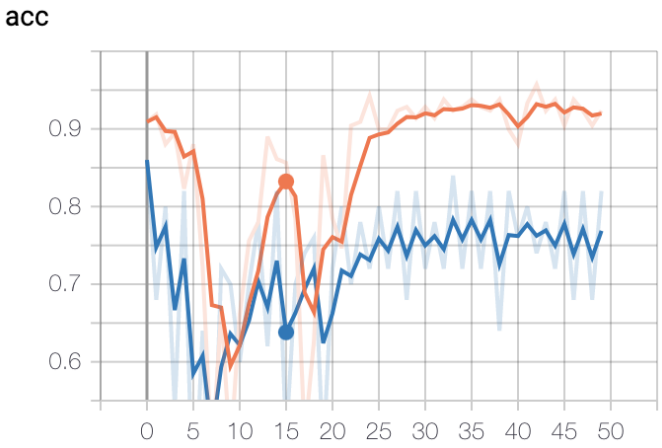

Figure 22: Test Loss vs Epoch, **SGD**, **0.0175 learning rate**, **50 epochs**

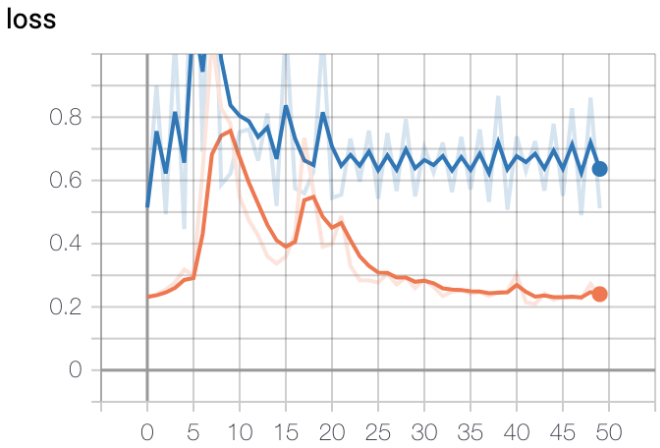

Figure 23: Test Loss vs Epoch, **SGD**, **0.0175 learning rate**, **50 epochs**

Results for: **SGD** optimizer, **0.0175 learning rate**, **50 epochs**:

```
Test loss: 0.5120910310745239
Test accuracy: 0.8200000047683715
```

Learning Rate = 0.02

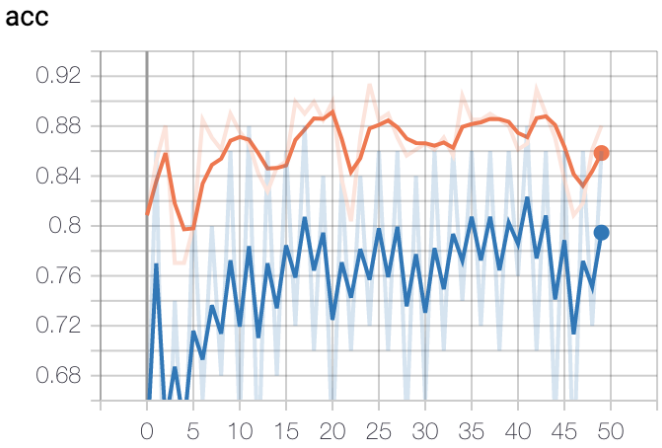

Figure 24: Test Loss vs Epoch, **SGD**, **0.02 learning rate**, **50 epochs**

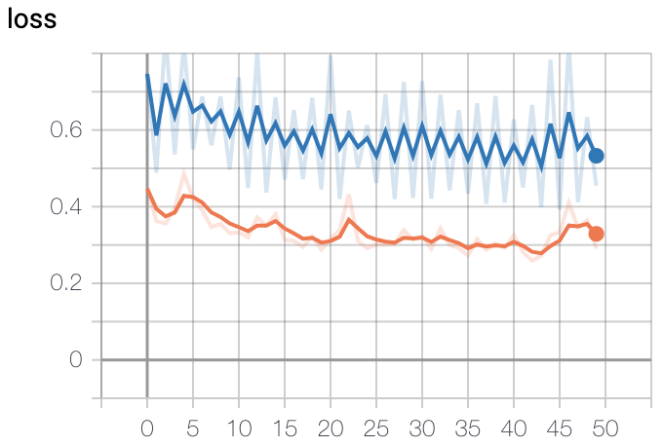

Figure 25: Test Loss vs Epoch, **SGD**, **0.02 learning rate**, **50 epochs**

Results for: **SGD** optimizer, **0.02 learning rate**, **50 epochs**:

```
Test loss: 0.4547306108474731
Test accuracy: 0.8600000023841858
```

Note that the following legend is the legend for all TensorBoard plots:

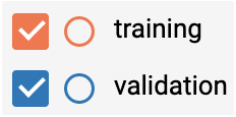

Figure 26: Legend for TensorBoard plots

We get the best model performance, which is an accuracy of **86%** and a loss of **0.454** with the **learning rate 0.02** and the **optimizer SGD**. However, these results are subject to change in almost each run since Keras does not provide an explicit command to set the seed of its random number generator. The randomized nature of the SGD algorithm combined with the irreproducibility of Keras code makes it harder to report a short interval for the best learning rate, but we can be more or less sure that a learning rate between 0.01 and 0.02 will probably give an accuracy above 80% in each run. These all imply how important learning rate is and how majorly it effects the model performance.

3.7 Discussion

The Optimizer

In our case, SGD outperformed ADAdelta in terms of test accuracy for the optimal learning rate, 0.02. However, ADAdelta is a more stable optimizer in the sense that it does not require its user to tune any hyper-parameter and it is not as random as SGD. Because of its stability and the fact that it resulted in a good-enough test accuracy of 80%, ADAdelta might be a better option in terms of model interpretability.

The Learning Rate

Looking at the plots, we can spot a major trade-off related to learning rate:

- A large learning rate decreases the time needed for training, but typically makes the optimizer oscillate around a local optimum, which results in pseudorandom accuracy and loss values.
- A small learning rate makes the optimizer more robust since it lowers the chance of missing a local optimum by taking small steps towards the negative gradient. However, it significantly increases the time needed for training. Hence, it is not a scalable choice.

This trade-off can be avoided to some extent by starting the training with a large learning rate and reducing it as the training proceeds.