İhsan Doğramaci Bilkent University

Introduction to Computer Vision
CS 484

# Assignment: Homework #2

*Author:*

Yusuf Dalva
Bilkent ID: 21602867

March 30, 2020

# Contents

# Part 1

In this section, both algorithms implemented and how data is fetched will be explained.

## Retrieving the dataset

In order to easily manipulate the cat dataset provided for this assignment, data is organized as a class designed for easy access to data. The dataset class used for Part 1 and Part 2 is given below:

```
import h5py
import numpy as np

class Dataset:
    def __init__(self, train_dataset_path, test_dataset_path):
        ## Training dataset
        self.train_data = h5py.File(train_dataset_path, 'r')
        ## Testing dataset
        self.test_data = h5py.File(test_dataset_path, 'r')

    def get_training_data(self):
        ## Organizing training dataset
        train_set = {
            'samples': np.array(self.train_data['train_set_x']) / 255,
            'labels': np.array(self.train_data['train_set_y'], dtype = np.int16)
        }
        return train_set

    def get_testing_data(self):
        ## Organizing testing dataset
        test_set = {
            'samples': np.array(self.test_data['test_set_x']) / 255,
            'labels': np.array(self.test_data['test_set_y'], dtype = np.int16)
        }
        return test_set
```

As it can be observed from the code given above, class organizes the data in the form of two dictionaries, which are **train_set** and **test_set**. For computation without concerning about overflow in log and exponential operations, the pixels in the images in the dataset are normalized. This normalization operation is being performed by dividing pixel values to 255. By doing this operation it is guaranteed that the values in matrices(image data) will be in the range [0,1].
Also to be concise about the labels for the images, the image values are hold as 16-bit integers, which is done by typecasting the numpy array to np.int16 datatype.

## Logistic Regression Classifier

Just like it is the case for organizing the dataset, the Logistic Regression classifier is also organized as a class that contains the training and testing data as attributes with the weights to be optimized. The constructing method used for this classifier is given below:

```
class LogisticClassifier:
    def __init__(self, training_dataset_path, testing_dataset_path, learning_rate, init_pref):
        self.dataset = Dataset(training_dataset_path, testing_dataset_path)
        train_data = self.dataset.get_training_data()
        test_data = self.dataset.get_testing_data()
        self.train_sample_count = train_data['samples'].shape[0]
        self.test_sample_count = test_data['samples'].shape[0]
        self.data_shape = (train_data['samples'].shape[1],
        train_data['samples'].shape[2], train_data['samples'].shape[3])
        self.train_samples = np.reshape(train_data['samples'],
            (self.train_sample_count, self.data_shape[0] *
```

```
            self.data_shape[1] * self.data_shape[2])).T
        self.train_labels = train_data['labels']
        self.test_samples = np.reshape(test_data['samples'],
            (self.test_sample_count, self.data_shape[0] *
            self.data_shape[1] * self.data_shape[2])).T
        self.test_labels = test_data['labels']
        # The number of weights is determined by
        # (number of rows) x (number of columns) x (number of channels)
        # There are 3 alternatives for weights initilization
        # uniform nonzero distribution, normal distribution or 0's
        try:
            if init_pref == 'normal':
                self.weights = np.array(np.random.normal(loc = 0, scale = 0.01,
                    size = (self.data_shape[0] * self.data_shape[1] * self.data_shape[2],1)))
                self.init_pref = 'normal'
            elif init_pref == 'uniform':
                self.weights = np.array(np.random.uniform(low = 0, high = 0.01,
                    size = (self.data_shape[0] * self.data_shape[1] * self.data_shape[2],1)))
                self.init_pref = 'uniform'
            elif init_pref == 'zeros':
                self.weights = np.zeros(
                    (self.data_shape[0] * self.data_shape[1] * self.data_shape[2], 1))
                self.init_pref = 'zeros'
            else:
                raise ValueError('Invalid value for init_pref,
                    enter one of (normal, uniform, zeros)')
        except ValueError as err:
            print(err)
        self.bias = 0 # Bias is initialized as 0
        self.learning_rate = learning_rate
```

As it can be observed from the constructor, for the implementation of the classifier the samples are linearized. In this operation each pixel is considered as a feature and none of the color channels are neglected. According to this approach, there is going to be $64x64x3 = 12288$ features for a single image. Considering the linear equation for computation of the z value that is going to be used for logistic classifier there is going to be 12289 parameters to be optimized (including bias). For clarity, the linear equation for computation of z value in sigmoid function is given below with its use:

$$z = w^T X + b \tag{1}$$

$$P(y = 1|X) = \frac{1}{1 + e^{-z}} \tag{2}$$

Here the letter w corresponds to the weights.

## Iterative approach for the classifier

As it is requested in the assignment, the training algorithm for one epoch of logistic regression classifier both iterative approach and matrix based approach. Both algorithms that are introduced updates the weights based on batch gradient descent approach. The iterative approach and the steps of the algorithm is going to be explained in this section.
The python code for the iterative training method is given below:

```
def iterative_training(self):
    # Initialize the loss and weight derivatives and bias derivative
    total_loss = 0
    weight_derivatives = np.zeros((self.data_shape[0] *
        self.data_shape[1] * self.data_shape[2], 1))
    bias_derivative = 0
    # Traversing through samples
    for sample_idx in range(self.train_sample_count):
```

```
        z = np.dot(self.weights.T, self.train_samples[:,sample_idx]) + self.bias
        a = (1 / (1 + np.exp(-z))).item()
        total_loss += (-self.train_labels[sample_idx] * np.log(a + np.exp(-30)) -
            (1 - self.train_labels[sample_idx]) * np.log(1 - a + np.exp(-30)))
        dz = a - self.train_labels[sample_idx]
        weight_derivatives += dz * np.reshape(self.train_samples[:, sample_idx],
            (self.train_samples[:, sample_idx].size, 1))
    total_loss = total_loss / self.train_sample_count
    weight_derivatives = weight_derivatives / self.train_sample_count
    bias_derivative = bias_derivative / self.train_sample_count
    self.weights = self.weights - (self.learning_rate * weight_derivatives)
    self.bias = self.bias - (self.learning_rate * bias_derivative)
    return total_loss
```

Initially the total loss value (J(w,b)) is initialized as together with the derivative values of the loss with respect to weights and bias. As the derivatives are going to be derived as sums, this approach is followed. The sum formulas for derivation of the derivatives is given in the following equation:

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(a^{(i)}, y^{(i)})}{\partial w_j} \tag{3}$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(a^{(i)}, y^{(i)})}{\partial b} \tag{4}$$

As the sum is calculated by looping through the training samples, the sums are initialized as 0.
Following this stage, the looping stage through the training samples begin. The initial operations are done for finding the sigmoid value with the current weights, which can be explained by *Equation 1* and *Equation 2*. For the derivatives of the loss function the following formula was used:

$$\frac{\partial L(a,y)}{\partial w_j} = x_j(a-y) \tag{5}$$

In this algorithm the update operation is done after calculating the total loss of the epoch. Due to that, the algorithm follows the batch gradient descent approach. The update equations are as follows:

$$w_i^{'} = w_i^{'} - \eta\frac{\partial J(w,b)}{\partial w_j} \tag{6}$$

Where the symbol $\eta$ represents the learning rate for training algorithm. As the result of the training process the method returns the loss value for that training iteration (epoch)

## Matrix approach for the classifier

With the same approach, the same algorithm is implemented with the matrix approach. In this approach, instead of a for loop fashion, the properties of matrices are being used. The algorithm is given below:

```
def matrix_training(self):
    z = np.dot(self.weights.T, self.train_samples) + self.bias
    a = 1 / (1 + np.exp(-z))
    dz = a - np.reshape(self.train_labels, (1, self.train_sample_count))
    dw = (1 / self.train_sample_count) * np.dot(self.train_samples, dz.T)
    db = (1 / self.train_sample_count) * np.sum(dz)
    self.weights = self.weights - (self.learning_rate * dw)
    self.bias = self.bias - (self.learning_rate * db)
    reshaped_labels = np.reshape(self.train_labels, (1, self.train_sample_count))
    return np.sum(-reshaped_labels * np.log(a + np.exp(-30)) -
        (1 - reshaped_labels) * np.log(1 - a + np.exp(-30))) / self.train_sample_count
```

## Plotting with desired configurations

For this part, the model is trained with 50 epochs and learning rate = 0.0001 as the hyper-parameters. For initialization of weights the main principle was eliminating the bias for any feature which is being done by initializing as a normal distribution or uniform (either a random value or all zeros). The plots for each configuration is attached as follows:

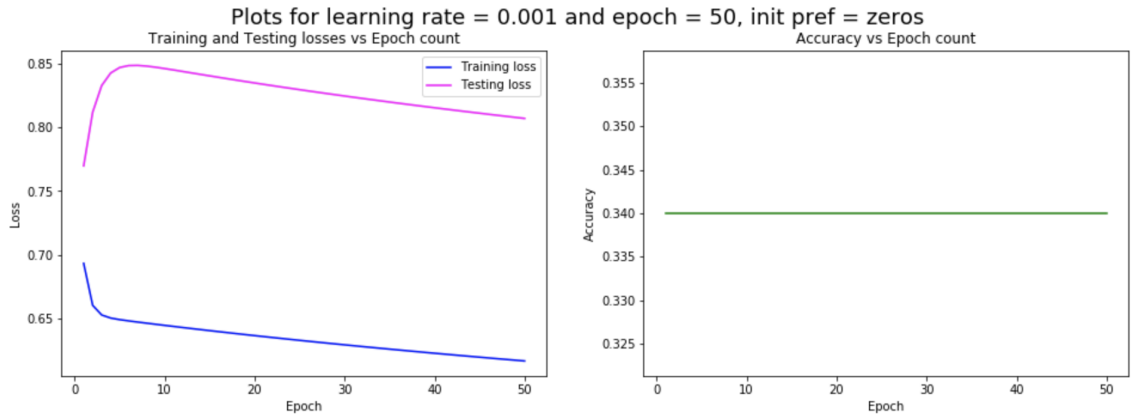


Figure 1: Part 1 with initializing weights as zeros

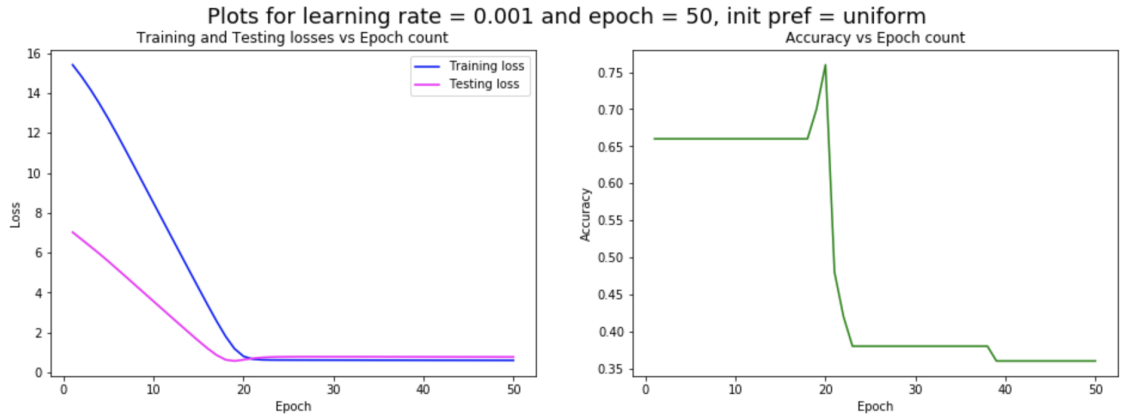


Figure 2: Part 1 with initializing the weights as an uniform distribution selected randomly

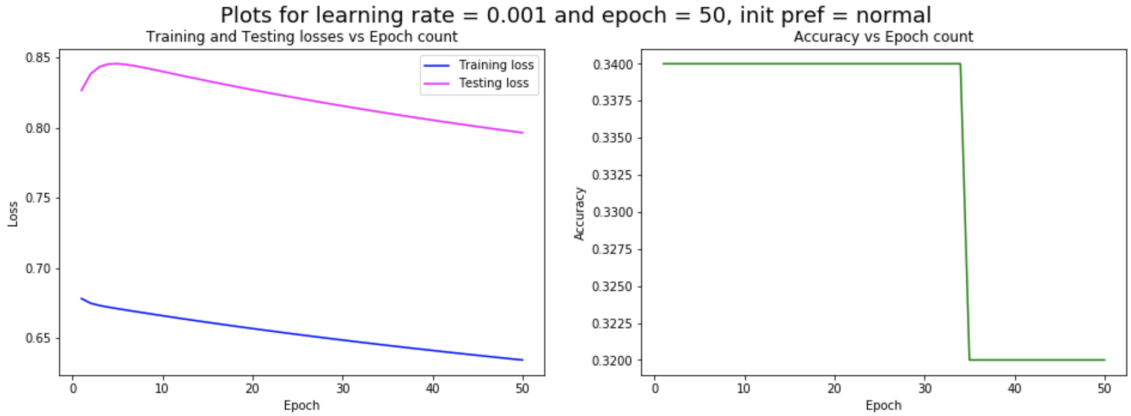


Figure 3: Part 1 with initializing the weights as a random normal distribution

For the plots given above it can be observed that the model performs poorly in all of them by giving an accuracy value 34%. However, when starting the weights with 0's, the model doesn't show and improvement and the loss values are really close to each other. This follows that the other hyper-parameters are not sufficient for a good training process. The changes in the hyper-parameters will be examined in Part 2. For uniform and normal distribution, the models show improvement but due to the range of values, the model using uniform weights shows the characteristic that the values fluctuate quickly. Even though the result is not that much different, uniform weights seems like a better approach. Even though the accuracy value shows 34% for every attempt, this value is not significant as 34% of the instances are 0 and model predicts 0 every time. Considering the limited amount of data and the hyper-parameters such a result is not that surprising.

# Part 2

## Generating the hyper-parameter combinations

As indicated in the assignment for tuning the model, the model is trained with different learning rates and different amounts of epochs. The hyper-parameters that are tested in this part are given below:

```
epochs = [50, 500, 1000, 2500]
learning_rates = [0.0001, 0.001, 0.003, 0.01, 1]
```

Just like in the first part, different initialization techniques for the weights are attempted (uniform, normal and zeros). The parameters that are inputted to the model are generated in a grid manner without any external library. The method for generation of these weights are given below:

```
def prepare_parameter_grid():
    epochs = [50, 500, 1000, 2500]
    learning_rates = [0.0001, 0.001, 0.003, 0.01, 1]
    grid = []
    for epoch in epochs:
        for rate in learning_rates:
            grid.append((epoch, rate, 'uniform'))
            grid.append((epoch, rate, 'normal'))
            grid.append((epoch, rate, 'zeros'))
    return grid
```

## Training with different hyper-parameters

For each of the combinations the cost vs epoch plot and accuracy change plot is given in this section. Where necessary some comments will be added different than the overall discussion given at the end of this section. There are 60 different plots for this fine tuning. If the plots seem small in terms of figures, the same plots are also provided in the submission file ('YusufDalva_21602867_PA2.ipynb').
For simplicity, only the plots that uses uniform distribution as starting weights are going to be provided as the best result is obtained with these weights. However, as the number of epochs increases the difference with the initialization technique decreases (even gets to 0). In this way, 20 different plots will be provided. The plots that are not included in the report can be found in the notebook submitted.
Training the model with 50 epochs:

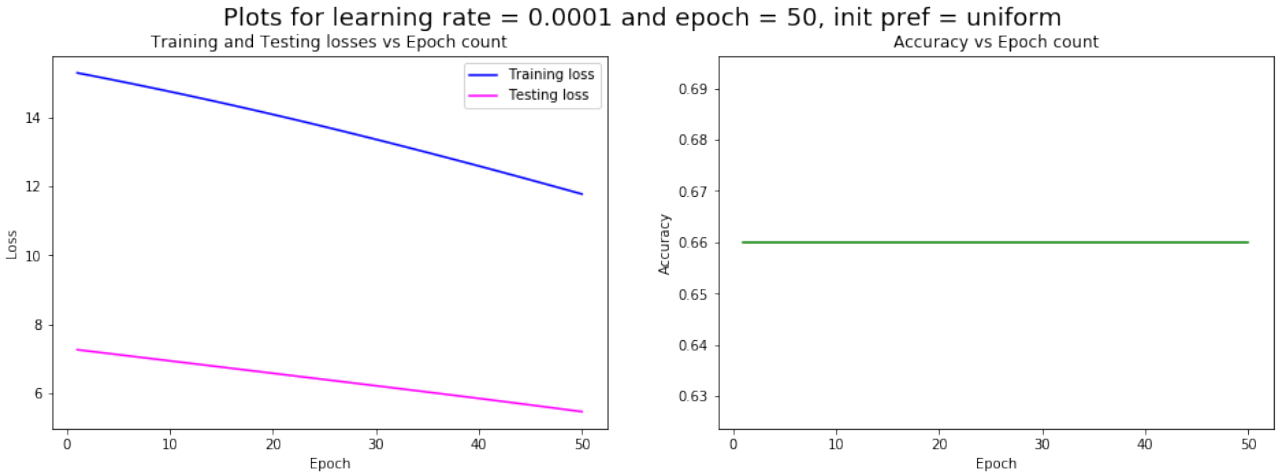


Figure 4: Results for learning rate = 0.0001 and number of epochs = 50

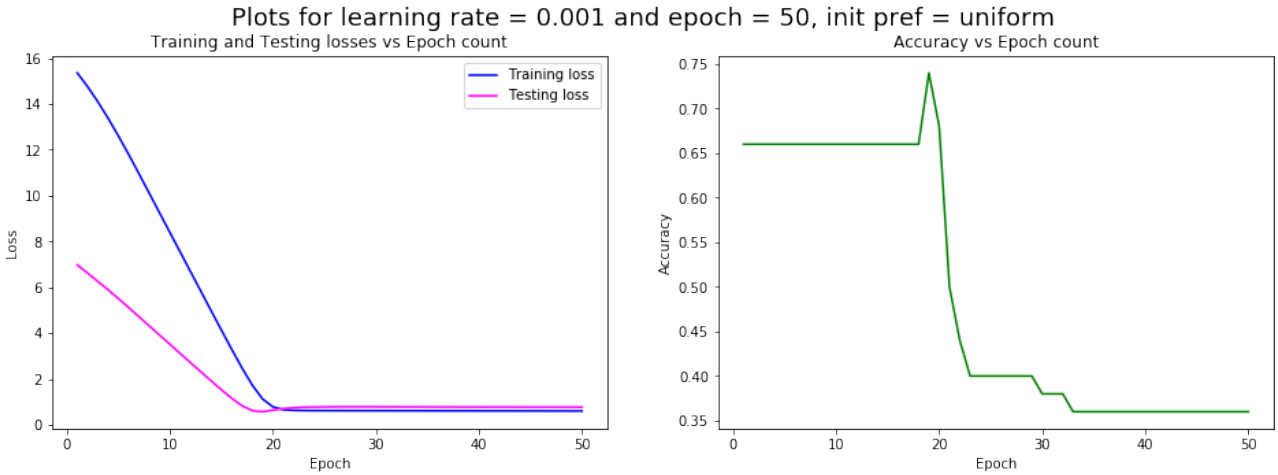


Figure 5: Results for learning rate = 0.001 and number of epochs = 50

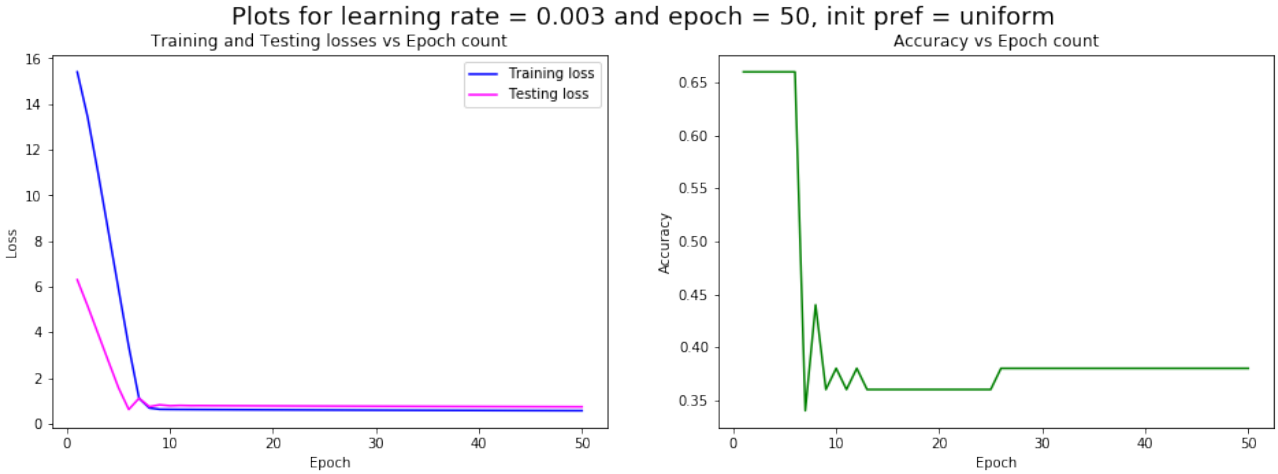


Figure 6: Results for learning rate = 0.003 and number of epochs = 50

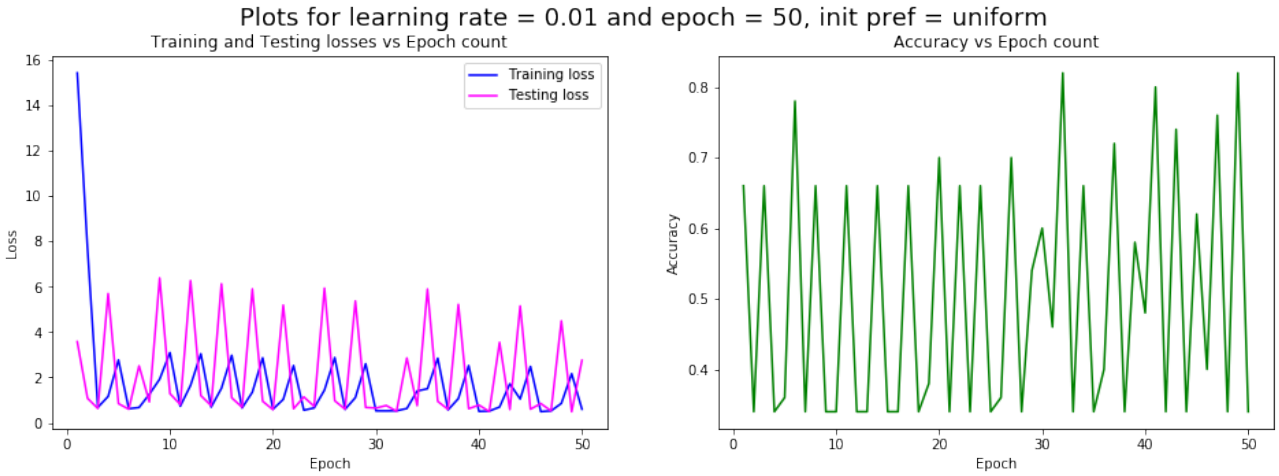


Figure 7: Results for learning rate = 0.01 and number of epochs = 50

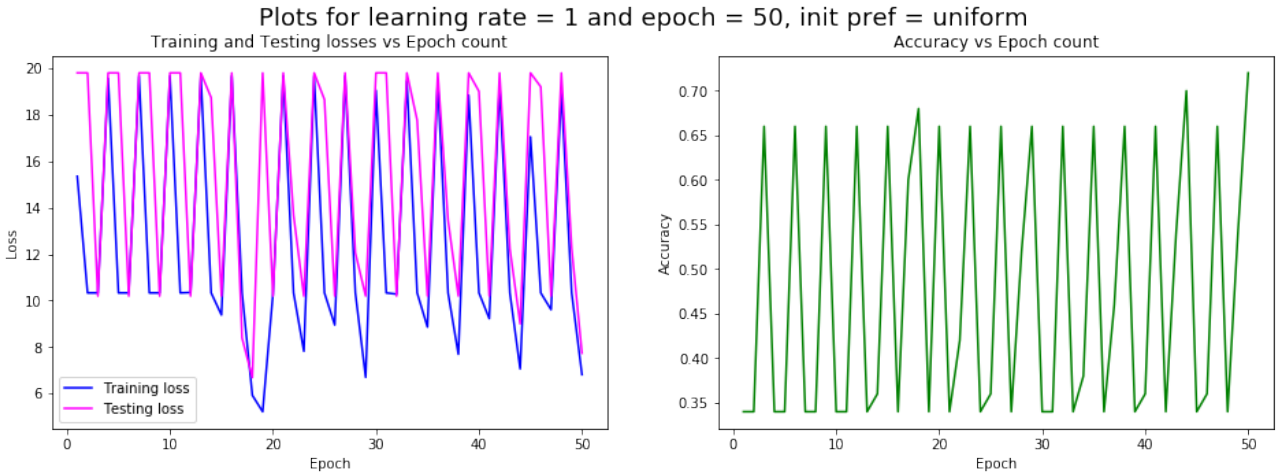


Figure 8: Results for learning rate = 1 and number of epochs = 50

Just as expected from the count of epochs, the number of updates performed to weights were not sufficient at this point. Due to that the optimal accuracy value has not been found. Here the general fact about the learning rate can be observed. For learning rates that are relatively big (0.01, 1), model passes the optimum point for the weights which causes fluctuations for the weights. For learning rates that are smaller, the learning process continues and there is an optimal point to b found for the weights. Even though at some epoch values the accuracy value around 80 percent can be observed, considering the change in loss values, this accuracy value would not be suitable for evaluation. For the learning rate 0.001, the movement in the loss function still is downward whereas the accuracy value is constant which shows that learning process is slow.
Training with 500 epochs:

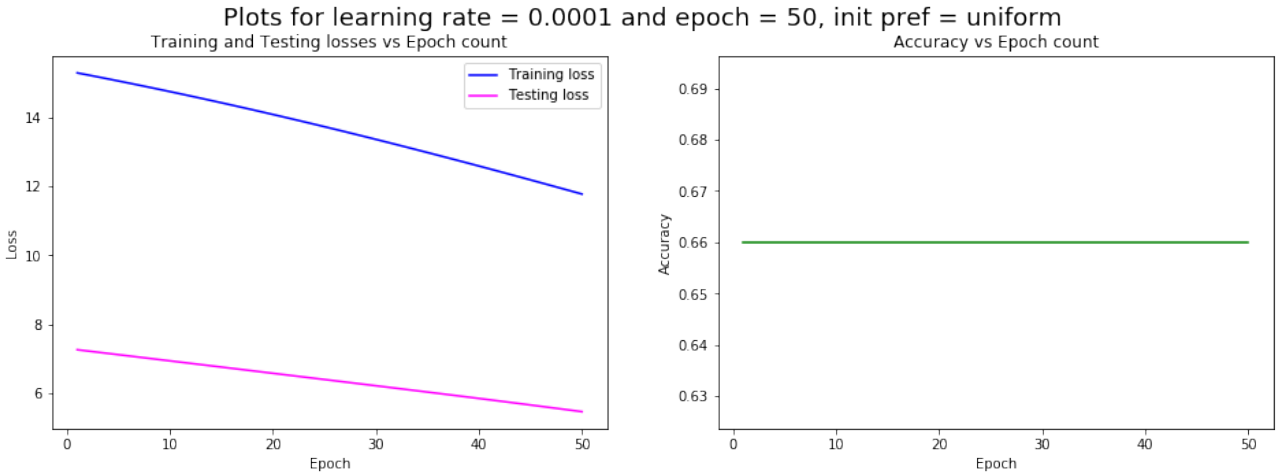


Figure 9: Results for learning rate = 0.0001 and number of epochs = 500

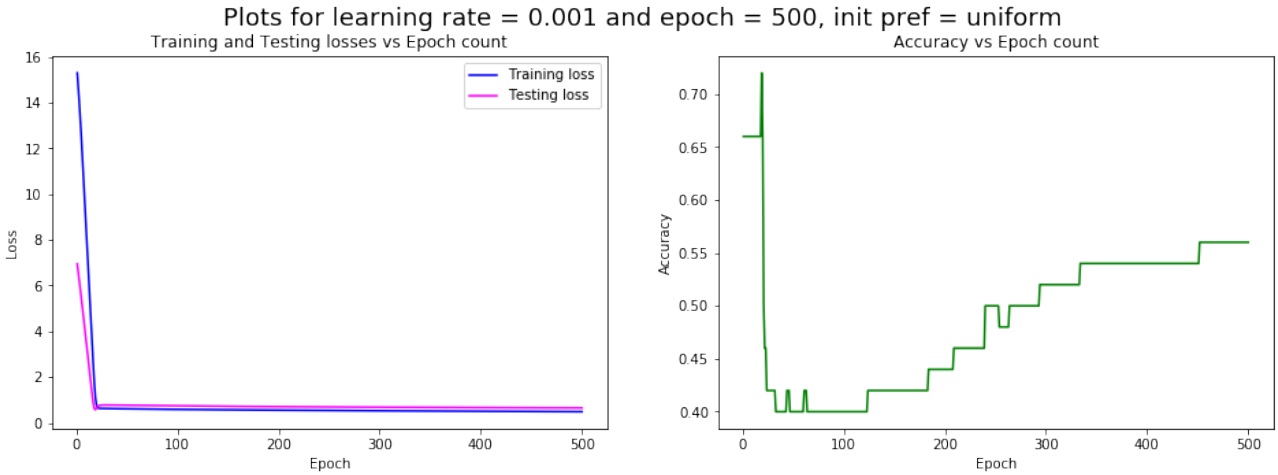


Figure 10: Results for learning rate = 0.001 and number of epochs = 500

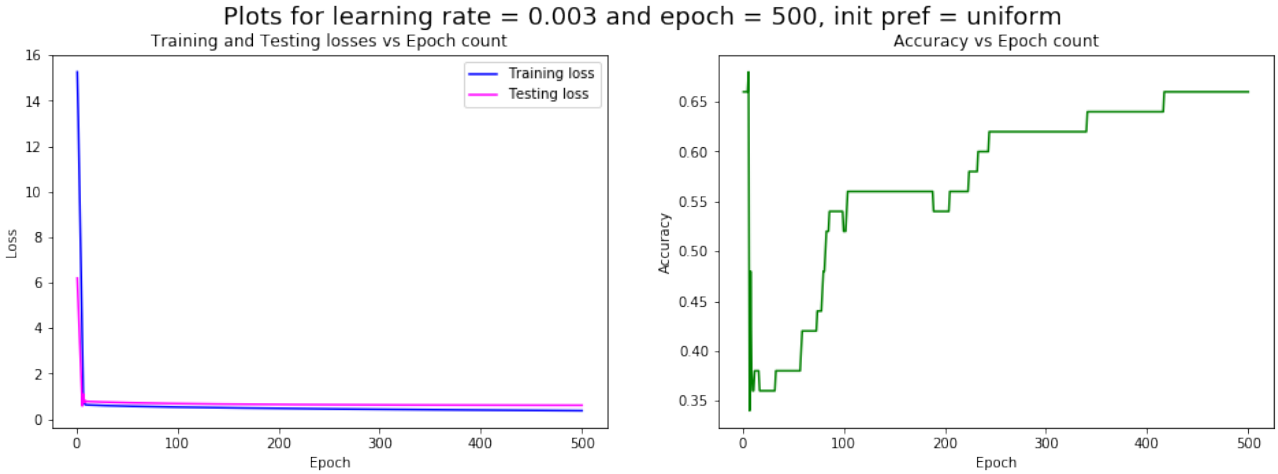


Figure 11: Results for learning rate = 0.003 and number of epochs = 500

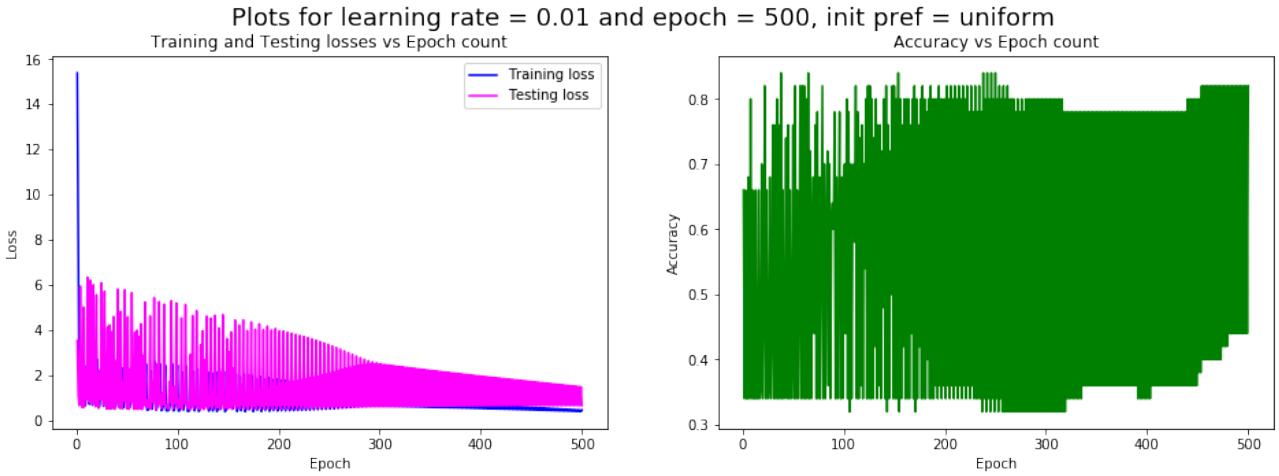


Figure 12: Results for learning rate = 0.01 and number of epochs = 500

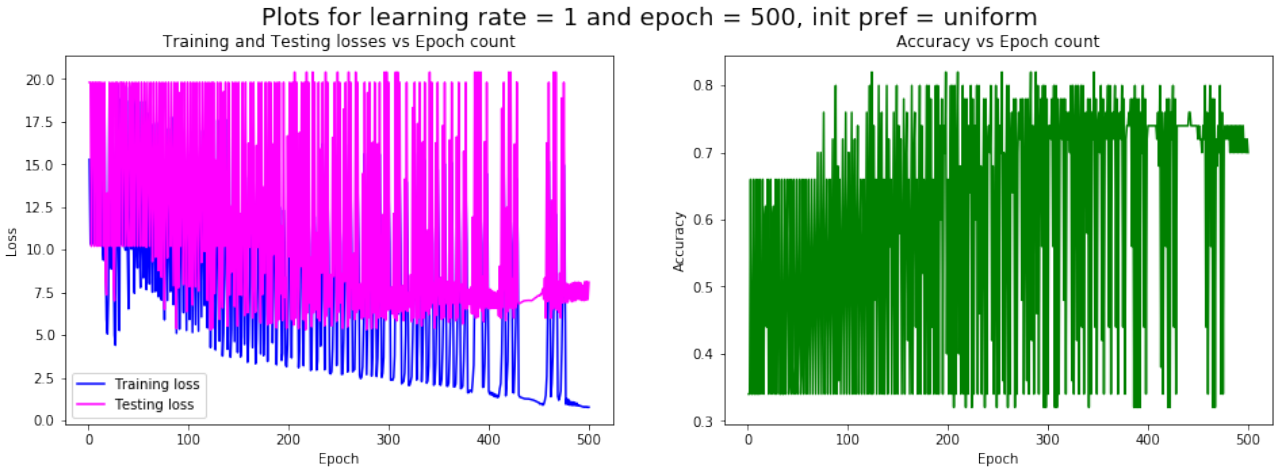


Figure 13: Results for learning rate = 1 and number of epochs = 500

Just as argued from the previous results, the high fluctuations are also observed in high learning rates with 500 epochs. Considering the learning rates, an idea about the ideal hyper-parameters can be obtained at this point. St the end of the whole process, the optimal learning rate is expected to be either 0.001 or 0.003 among the predefined learning rate list. Also supporting the argument given with 50 epochs, with learning rate 0.001 the learning process is so slow that no improvement is observed. The main idea that these group of plots gives us is how high learning rates effect the performance of the model in a negative way. The accuracy value 80% is also observed here but since both the loss and the accuracy value did not converge yet (far from convergence) this is not taken as an ideal configuration.
Training for 1000 epochs:

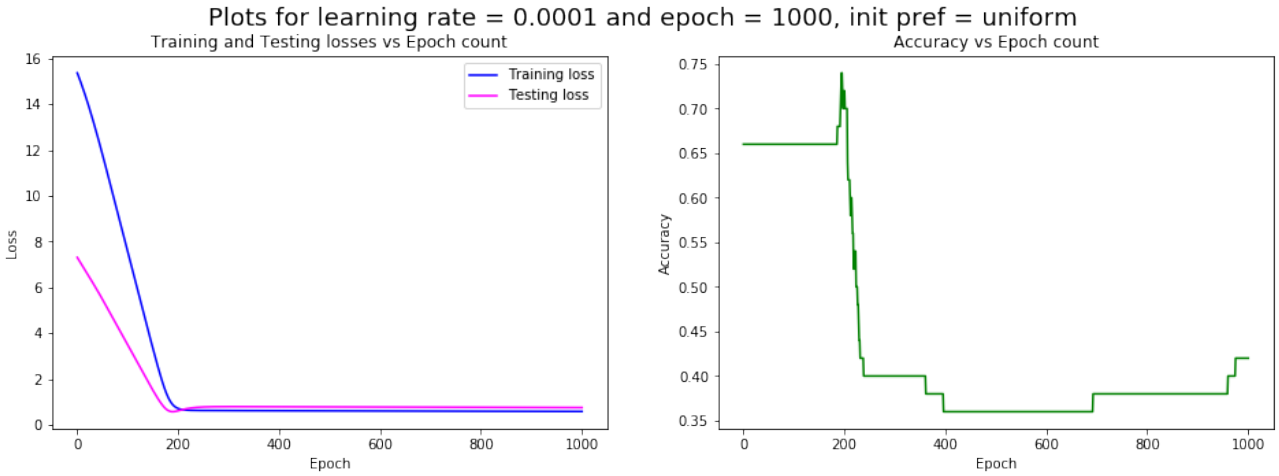


Figure 14: Results for learning rate = 0.0001 and number of epochs = 1000

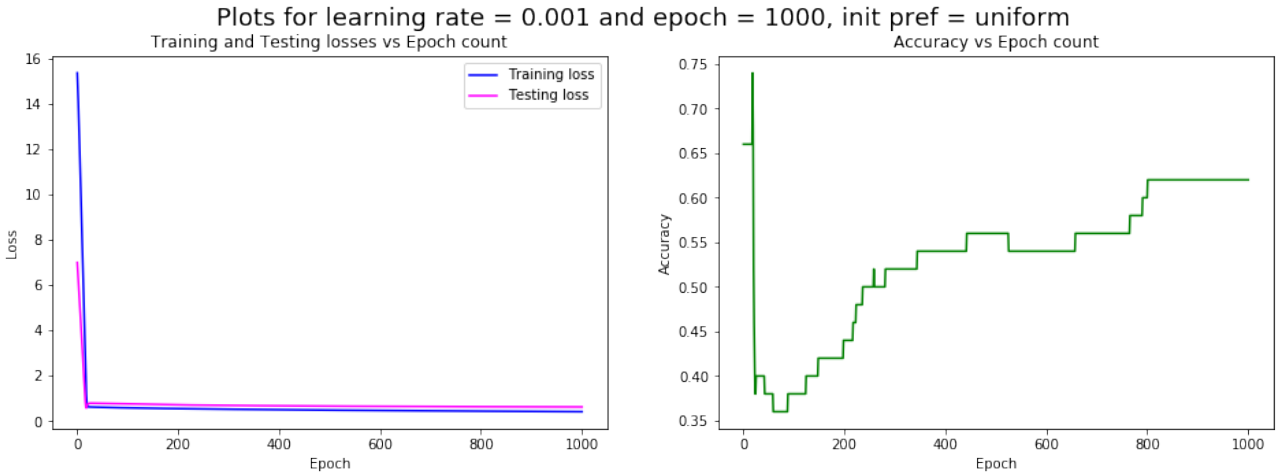


Figure 15: Results for learning rate = 0.001 and number of epochs = 1000

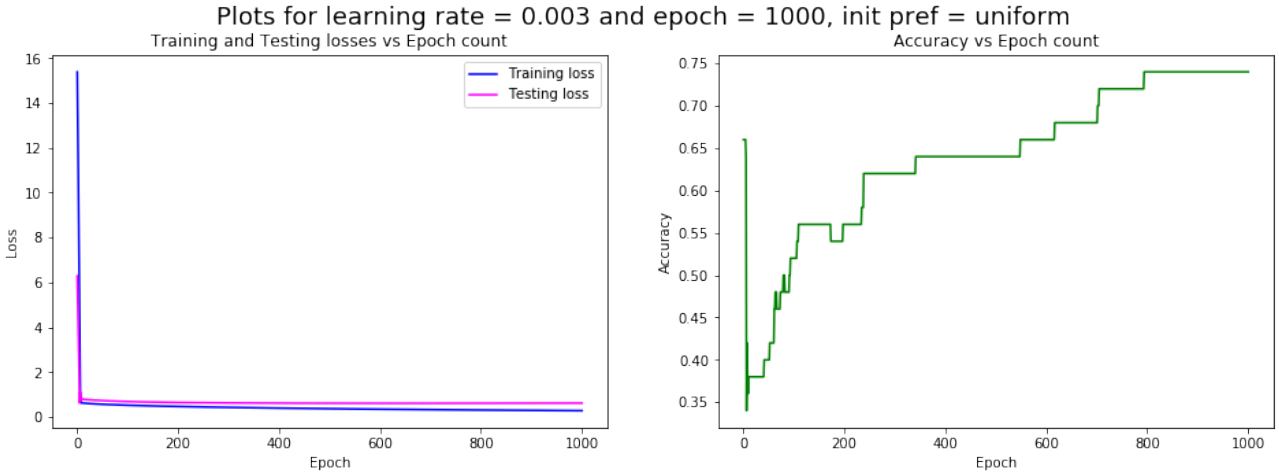


Figure 16: Results for learning rate = 0.003 and number of epochs = 1000

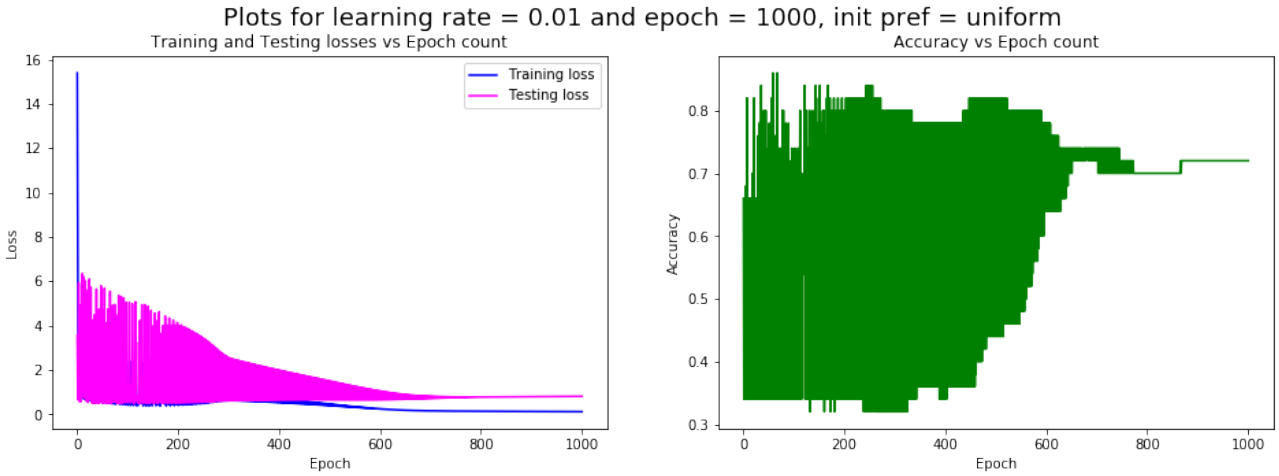


Figure 17: Results for learning rate = 0.01 and number of epochs = 1000

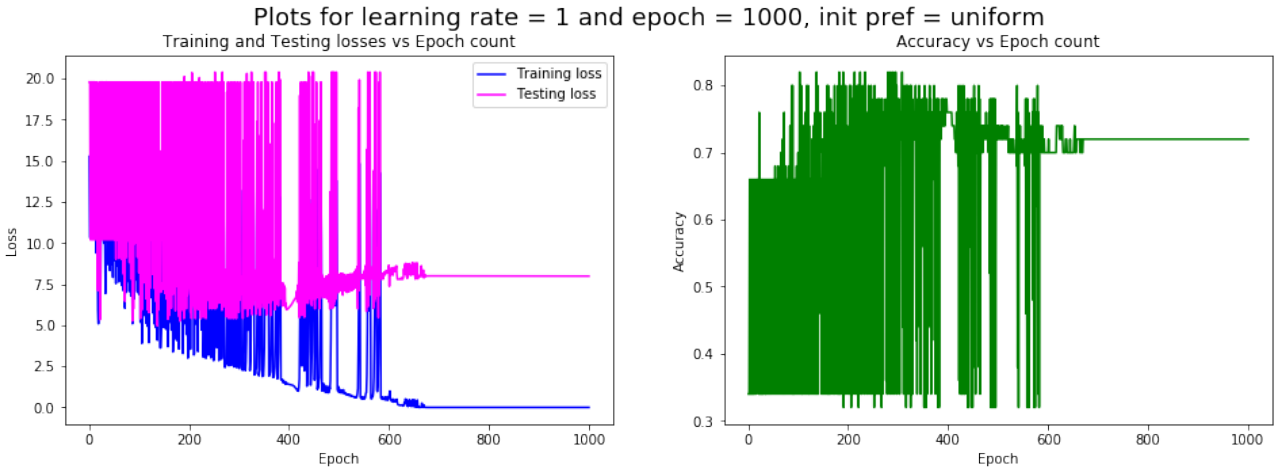


Figure 18: Results for learning rate = 1 and number of epochs = 1000

Concluding most of the assumptions made, the best parameter configurations also makes significant appearance here. The fact of fluctuations in terms of loss and accuracy still shows its significance with high learning rates. In these cases it is almost guaranteed that the local minima is hardly reached after 700 epochs. Here it is seen that the learning rate 0.003 is the optimal learning rate most probably. The convergence of the loss function and the accuracy value gives information about that. For the learning rate 0.0001, after a point the improvement in accuracy values can be observed which is the proof that the learning process is being performed slowly.
Training for 2500 epochs:

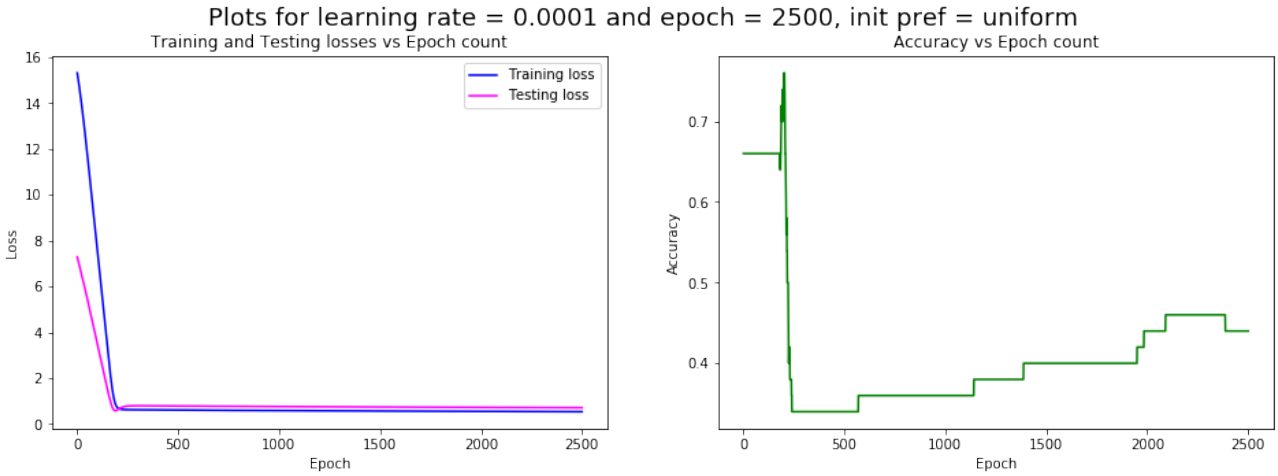


Figure 19: Results for learning rate = 0.0001 and number of epochs = 2500

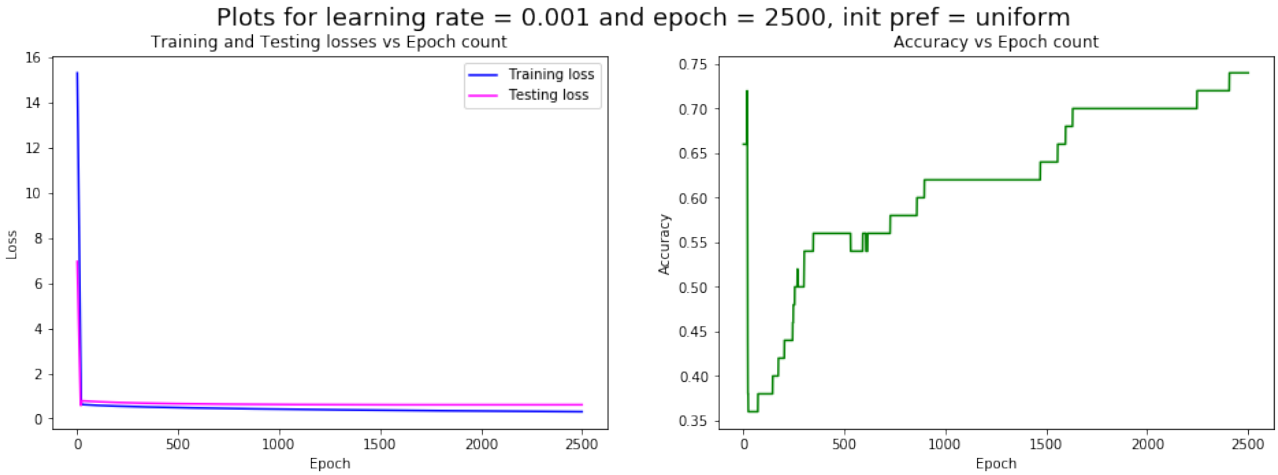


Figure 20: Results for learning rate = 0.001 and number of epochs = 2500

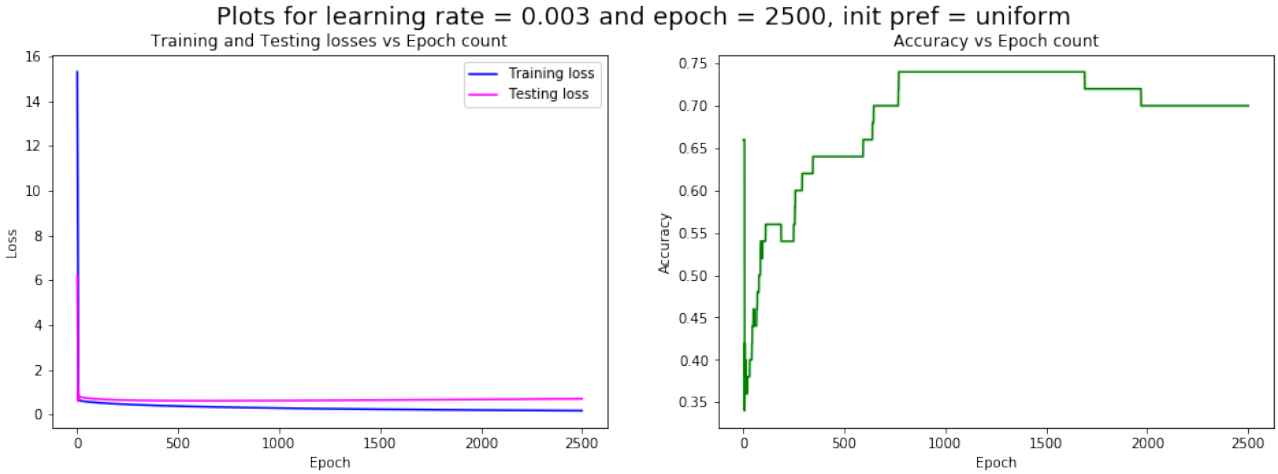


Figure 21: Results for learning rate = 0.003 and number of epochs = 2500

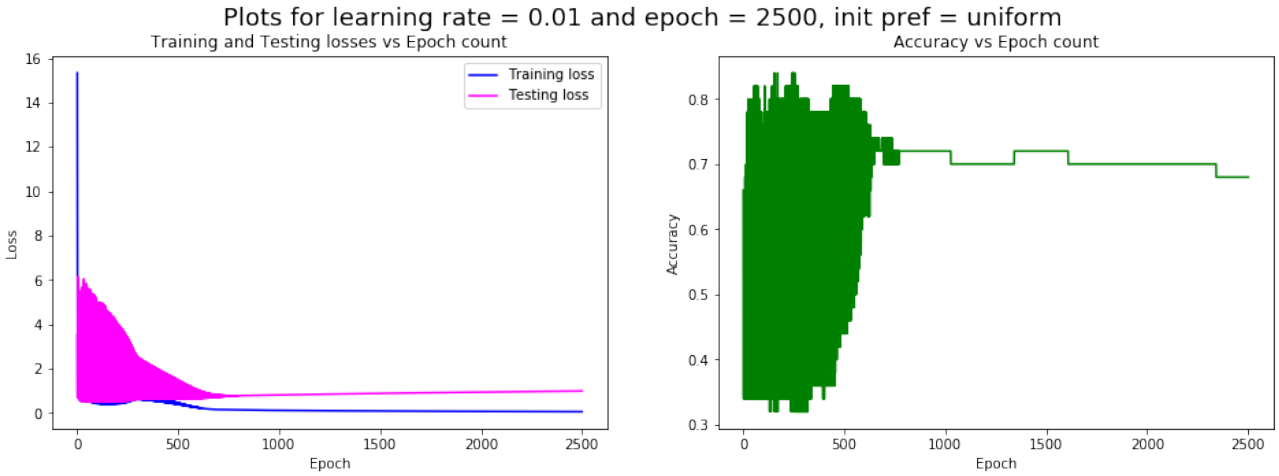


Figure 22: Results for learning rate = 0.01 and number of epochs = 2500

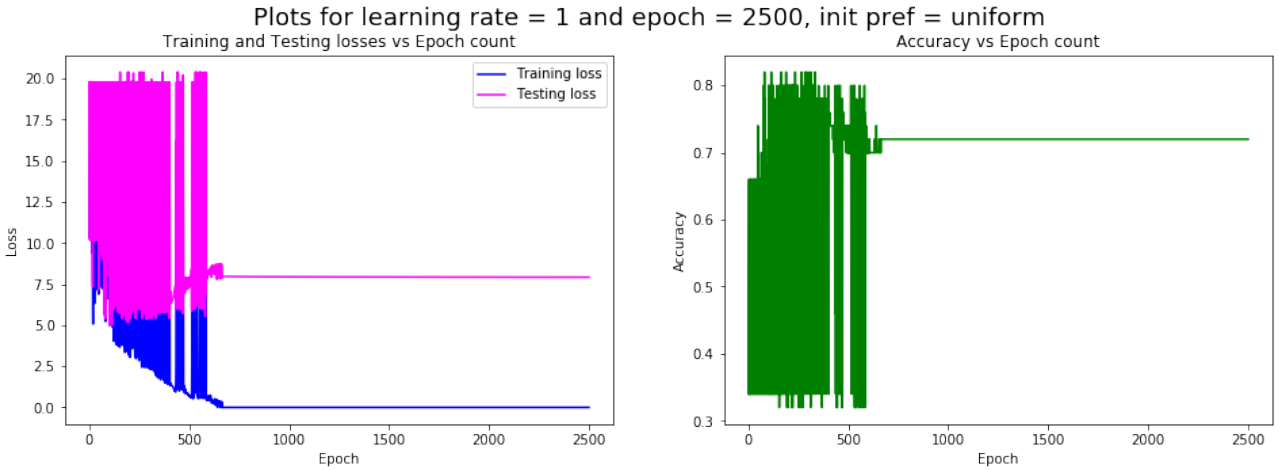


Figure 23: Results for learning rate = 1 and number of epochs = 2500

Concluding the discussion, with 2500 epochs the ideal training accuracy is concluded as 74% with either learning rate 0.001 and 2500 epochs or learning rate 0.003 and 1000 epochs. Considering the computation resources the configuration learning rate = 0.003 and number of epochs = 1000 is the ideal scenario. After investigating the other distributions, with 1000 epochs none of the starting points make a difference here. But at the end the overall training process with uniform distribution is seen as the best due to the non-biased approach and the randomness i provides initially. Even though the models that are trained under high learning rates give similar accuracy values, if the model is going to be trained with low epoch values, being certain that the local minima is reached cannot be concluded. Considering the stability of the change of the loss value, the suggested hyper-parameters gives the better result even in lower epoch values. This result is concluded with comparison with other plots that the training process started with different weights.
To conclude the discussion the details about the process are:

- Optimal hyper-parameters are learning rate = 0.003, number of epochs = 1000
- In the training process the weights and the bias term are optimized. As in the training process all of the pixels in all of the channels are considered as features for the classifier there are 64x64x3 = 12288 weights and 1 bias term. In total 12289 parameters are optimized.

# Part 3

## Modification of model and code for running on cat dataset

To maintain the completeness of the code for the assignment, the deep neural network is also run on local. This created the trade-off in terms of runtime but the ease of implementation is followed. For organization of the dataset the class introduced in Part 1 is used which is given in this section: The dataset paths for the assignment is assumed as the same folder with the notebook file. The paths of the dataset files are defined as follows:

```
train_dataset_path = 'train_catvnoncat.h5'
test_dataset_path = 'test_catvnoncat.h5'
```

The notebook provided in the submission can be used by changing these paths for the dataset if necessary. Similar to the approach followed in the first two parts, the dataset is organized in terms of a class following Object-Oriented Programming paradigm. The class for dataset is given for completeness of the report:

```
class Dataset:
    def __init__(self, train_dataset_path, test_dataset_path):
        ## Training dataset
        self.train_data = h5py.File(train_dataset_path, 'r')
        ## Testing dataset
        self.test_data = h5py.File(test_dataset_path, 'r')

    def get_training_data(self):
        ## Organizing training dataset
        train_set = {
            'samples': np.array(self.train_data['train_set_x']) / 255,
            'labels': np.array(self.train_data['train_set_y'])
        }
        return train_set

    def get_testing_data(self):
        ## Organizing testing dataset
        test_set = {
            'samples': np.array(self.test_data['test_set_x']) / 255,
            'labels': np.array(self.test_data['test_set_y'])
        }
        return test_set
```

The model given in the URL provided in the assignment is programmed using a gray-scale image and 28x28 images. In order to be able to run the model defined on cat dataset, the dimensions are modified and the model is modified to be able to use it with RGB images in the following way:

```
# input image dimensions
img_rows, img_cols = 64,64

# the data, split between train and test sets
(x_train, y_train) = (dataset.get_training_data()['samples'],
                dataset.get_training_data()['labels'])
(x_test, y_test) = (dataset.get_testing_data()['samples'],
                dataset.get_testing_data()['labels'])

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
    input_shape = (3, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
    input_shape = (img_rows, img_cols, 3)
```

As the dataset class normalizes the pixels (division by 255), the additional division operation given in the code is removed.
Finally the hyper-parameters(batch size, number of classes, number of epochs) to run the model is given as follows:

```
## Training Parameters
batch_size = 128
num_classes = 2
epochs = 12
```

The number of epochs is the main parameter that is changed for this part overall. Values are 12, 30 and 50 for this part.
After running the model with the necessary configurations, the final accuracy value for the test set is found as **34%**, the loss values and accuracy values found during training is given below. The model is trained with SGD optimizer:

```
Train on 209 samples, validate on 50 samples
Epoch 1/12 - 2s 8ms/step - loss: 0.6904 - accuracy: 0.5024
- val_loss: 0.7422 - val_accuracy: 0.3600
Epoch 2/12 - 1s 7ms/step - loss: 0.6636 - accuracy: 0.6029
- val_loss: 0.7700 - val_accuracy: 0.3400
Epoch 3/12 - 1s 7ms/step - loss: 0.6661 - accuracy: 0.6220
- val_loss: 0.7859 - val_accuracy: 0.3400
Epoch 4/12 - 1s 7ms/step - loss: 0.6671 - accuracy: 0.6555
- val_loss: 0.8101 - val_accuracy: 0.3400
Epoch 5/12 - 1s 7ms/step - loss: 0.6482 - accuracy: 0.6555
- val_loss: 0.8224 - val_accuracy: 0.3400
Epoch 6/12 - 1s 7ms/step - loss: 0.6554 - accuracy: 0.6507
- val_loss: 0.8152 - val_accuracy: 0.3400
Epoch 7/12 - 1s 7ms/step - loss: 0.6401 - accuracy: 0.6699
- val_loss: 0.8258 - val_accuracy: 0.3400
Epoch 8/12 - 1s 7ms/step - loss: 0.6534 - accuracy: 0.6507
- val_loss: 0.8285 - val_accuracy: 0.3400
Epoch 9/12 - 1s 7ms/step - loss: 0.6550 - accuracy: 0.6507
- val_loss: 0.8213 - val_accuracy: 0.3400
Epoch 10/12 - 1s 7ms/step - loss: 0.6513 - accuracy: 0.6603
- val_loss: 0.8273 - val_accuracy: 0.3400
Epoch 11/12 - 1s 7ms/step - loss: 0.6425 - accuracy: 0.6555
- val_loss: 0.8317 - val_accuracy: 0.3400
Epoch 12/12 - 1s 7ms/step - loss: 0.6561 - accuracy: 0.6507
- val_loss: 0.8336 - val_accuracy: 0.3400
Test loss: 0.8336230397224427
Test accuracy: 0.3400000035762787
```

After performing several runs with this configuration, it is observed that the accuracy value fluctuates which shows that the model does not converge. This is a non-surprising case with the number of epochs and the low amount of samples available for a deep neural network. The 34% value shows that all of the samples are estimated as 0's and there is no successful learning in this attempt.

## Understanding the network model

For a basic understanding of the model, the model can be shown with the following figure:

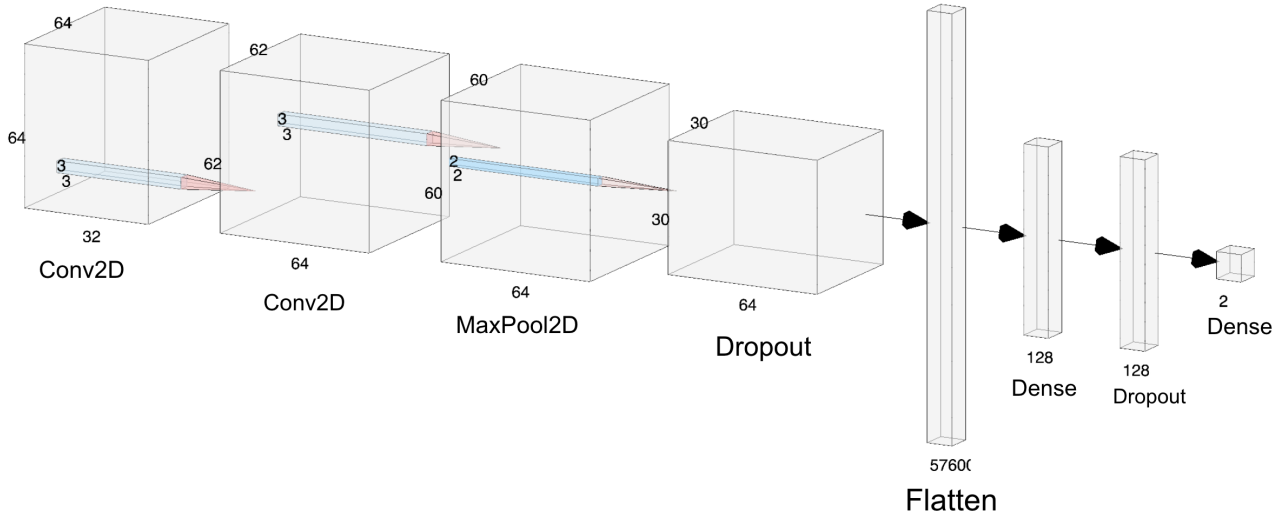


Figure 24: Visual representation of the model given in mnist_cnn.py

As seen in *Figure 1*, the model given in mnist_cnn.py starts with two consecutive convolutional layers. These layers are explained as follows:

- The first convolutional layer uses 32 kernels for convolution with input image size 64x64. The input image has 3 channels and the convolution result is computed by considering all of these 3 channels. The kernels used for the convolutiion is 3x3 in size. The output of the first layer is 32x62x62 in size.
- For the second convolutional layer, 64 kernels are used which are again 3x3 by size. As the result of the first layer follows, the input image size is 62x62. As no padding is used in the convolutional layer, the output is 64x60x60 where 64 is the number of kernels.

The layers following the conolutional layers are explained as follows:

- **MaxPool2D**: As a way to reduce the number of parameters, maximum pooling is the third layer that is used. The kernel dimensions are 2x2. As a result of that the result of the pooling operation is determined as the maximum value in a 2x2 window which is convolved in the input matrices.
- **Dropout**: In order to prevent overfitting and fasting the training process, dropout layer has been applied. By applying a dropout layer, the specified percentage of the neurons are ignored in the model. In this network, the dropout rate is specified as 0.25 which means that with $\frac{1}{4}$ probability a neuron will be removed from the network[1]. However these neurons will still be present in the network but their parameters will not be optimized. By this randomized approach, overfitting is expected to be eliminated up until a point.
- **Flatten**: This layer simply converts the matrices resulting from the convolutional part of the network and converts it to a vector. After this point, the computation is going to be done as a standard neural neteork.
- **Dense**: The activation function that is specified in the definition of the function is going to be executed in this layer. Also the number of neurons in this layer is specified in the layer definition. For the first dense layer there are 128 neurons and 2 neurons for the second one.

The activation functions in the layers also differ (Convolutional layers and Dense layers). In the convolutional layers and the first dense layer, ReLU is the used activation function which regularizes the features. The definition of the ReLU layer is given below[4]:

$$R(z) = max(0, z) \tag{7}$$

For the classification task, the softmax function is the choice as it shows the probabilities of being classified as a label. The sample is labeled with the class with the highest probability. The sigmoid function would also be used, but since the script is originally written for multi-class classification the softmax approach is followed. The code for the model is given below:

```
## Model definition
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Compared to the networks in the lecture slides, the network defined can be considered as really simple model with respect to the convolution and pooling layers that it includes in computation. There are no certain similarities when compared to any of the architectures shown (VGG, ResNet, AlexNet, Inception Network etc.) considering the size of the network in terms of layers. As this model is going to be used over and over with different configurations a method has been defined for generating this model as the Tensorflow background session is opened and closed at every training and validating cycle.

### Validation and Training losses over 30 epochs

In order to visualize the accuracy and loss values over each epoch, TensorBoard functionality is being used. For using this functionality, the callback method has been used as described in the webpage of Keras framework[2]. Sample use of TensorBoard is given below, which is used several times in the submitted notebook. Note that the last two lines are magic-lines that allows invoking TensorBoard without using command line (Terminal).

```
logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
callback = keras.callbacks.tensorboard_v1.TensorBoard(log_dir='./logs/SGD',
                histogram_freq=1, batch_size=32,
                write_graph=True, write_grads=False,
                write_images=True, embeddings_freq=0,
                embeddings_layer_names=None, embeddings_metadata=None,
                embeddings_data=None, update_freq='epoch')
```

Here, the path given in log_dir is the path where the notebook runs that the logs are saved.
As TensorBoard is running with Keras (not Tensorflow.keras) the loss functions could not be merged as one plot for training and validation. For clarity they are given as 2 separate plots.
The sample code for fitting and evaluating the model is given below:

```
model1.compile(loss=keras.losses.categorical_crossentropy,
               optimizer=keras.optimizers.Adadelta(),
               metrics=['accuracy'])
model1.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           validation_data=(x_test, y_test), callbacks = [ada_callback])
score = model1.evaluate(x_test, y_test, verbose=0)
```

Following the training and validating processes, the plots are shown below:

- **Accuracy plot**: As it can be observed from the accuracy plots, the model is still on the learning process and the data obtained from the training data cannot be reflected to the validation data completely. However after epoch number 12 an increase in accuracy can be seen (around 10%). However, the model performs barely better than random which is not promising for a CNN model.

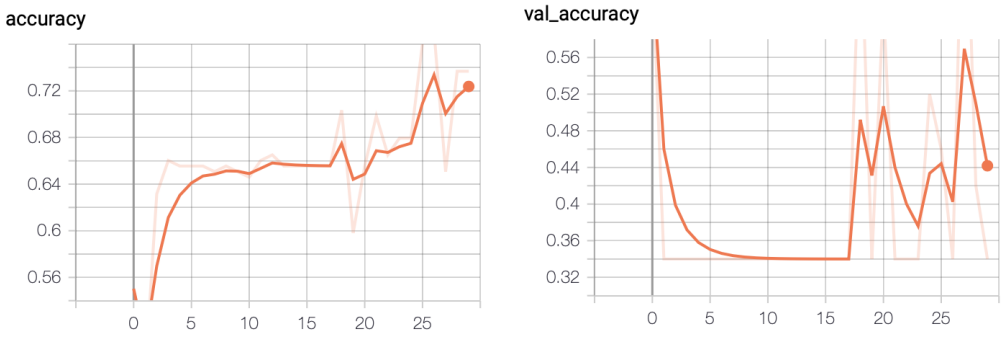


Figure 25: Accuracy values over epoch for training(on left) and validation (on right)

- **Loss plot**: Even though the results were not completely promising in terms of accuracy the improvement in loss values in training is observable in training loss. Considering that the model partially failed in terms of prediction performance, the convergence is not expected from the model. As it can be seen from the validation loss plot, the loss value fluctuates a lot which is a sign that the weights did not converge yet.

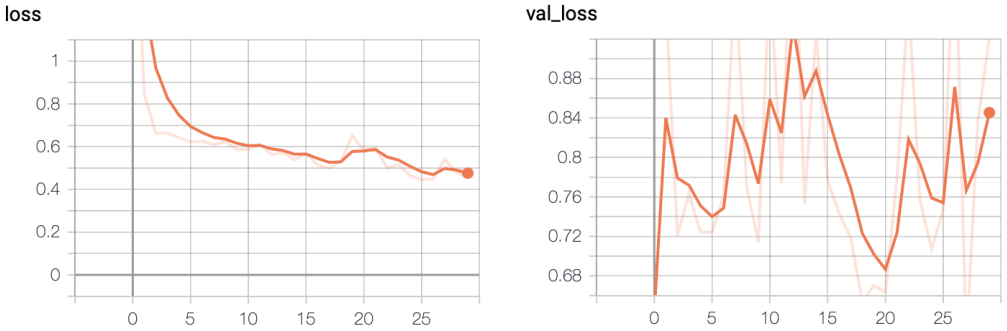


Figure 26: Loss(Cost) values over epoch for training(on left) and validation(on right)

Comparing the results with 12 epochs, of the same optimizer an improvement is observed. The non converge can also be observed on the accuracy values. Also since the test dataset contains only 50 samples, this fluctuation can be caused by only a few samples but their effect is relatively big considering the calculation of accuracy. The resulting test loss and test accuracy values are as follows:

```
Test loss: 0.9217
Test accuracy: 0.4200
```

## Training with SGD Optimizer

Just like the previous section, TensorBoard is used for visualizing the results. The model is trained for 30 epochs too. The learning rate is kept as 0.001 here. The plots and relative observations are given below:

- **Accuracy plot**: As a characteristic of the SGD optimizer, the data is expected to converge slowly. Being compatible with this fact the training process still does not show any effect which is a sign of slow convergence. Here the initial weights also have effect but overall SGD can be considered as an optimizer that converges slowly[3].

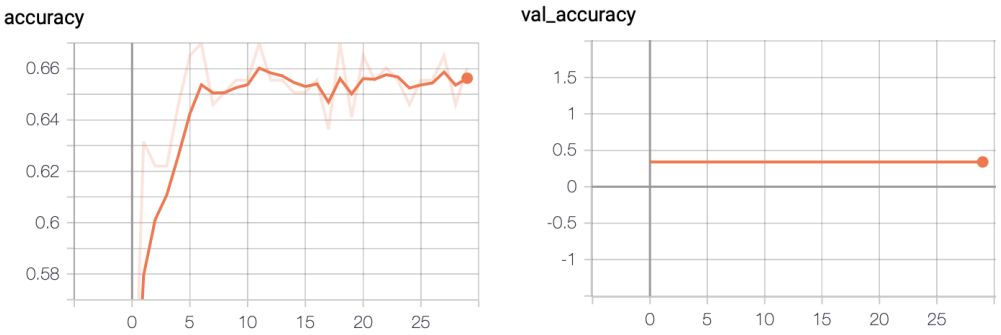


Figure 27: Accuracy values over epoch for training(on left) and validation (on right)

- **Loss plot**: For the loss values, with the effect of continuing training after the initial steps, the training proceeds slowly. This effect can be seen from the small changes in the loss plots. This reflects the characteristics of the SGD optimizer.

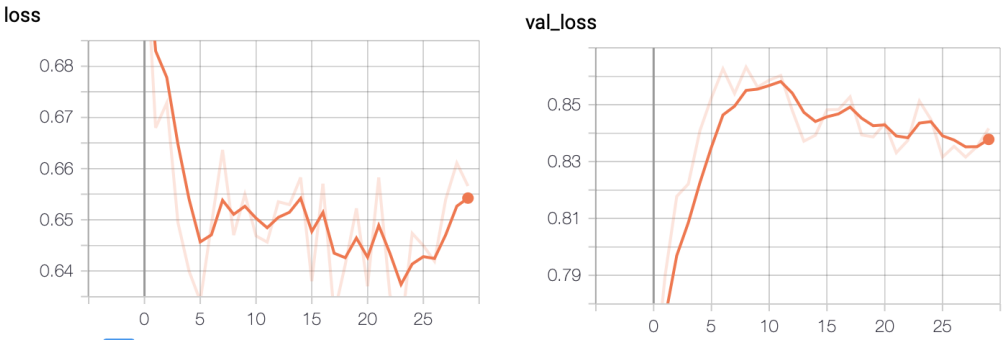


Figure 28: Loss(Cost) values over epoch for training(on left) and validation(on right)

Comparing with the results with the 12 epochs, the predictive performance of the algorithm did not show any sign of improvement. However, due to the drop in loss value in validation, it can be said that the training process continues. For Stochastic Gradient Descent algorithm, the results the converge is expected to be achieved slowly which may be an explanation of this result.
Also after completing several runs, the accuracy values change drastically. This issue can also be explained by not converging, since the weights are initialized randomly.

```
Test loss: 0.8416
Test accuracy: 0.3400
```

## Fine tuning the learning rate

For fine tuning the optimizer, the learning rates are selected in the same manner with Part 2. The selected learning rates that are going to be trained over 50 epochs are given below:

```
learning_rates = [1, 0.1, 0.01, 0.001, 0.0001]
```

For each rate, an instance of a model is created with a new session. The plots of the accuracy values are given below for each learning:

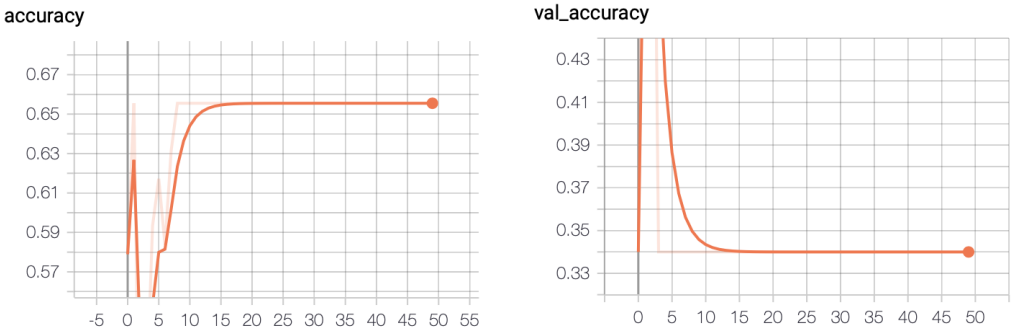


Figure 29: Training(on left) and testing(on right) accuracy's for learning rate = 1

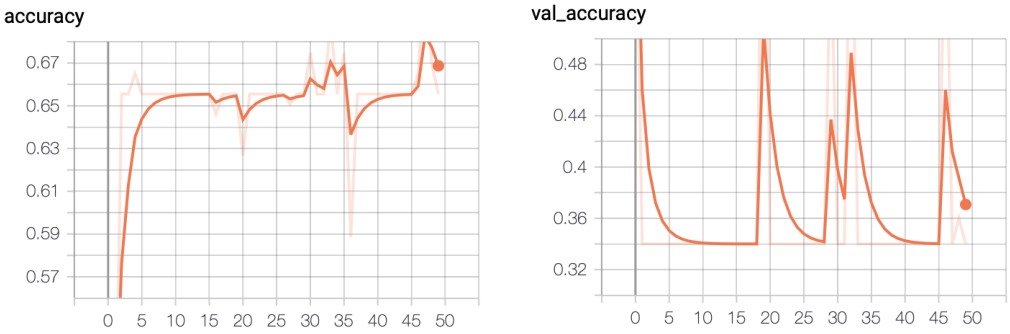


Figure 30: Training(on left) and testing(on right) accuracy's for learning rate = 0.1

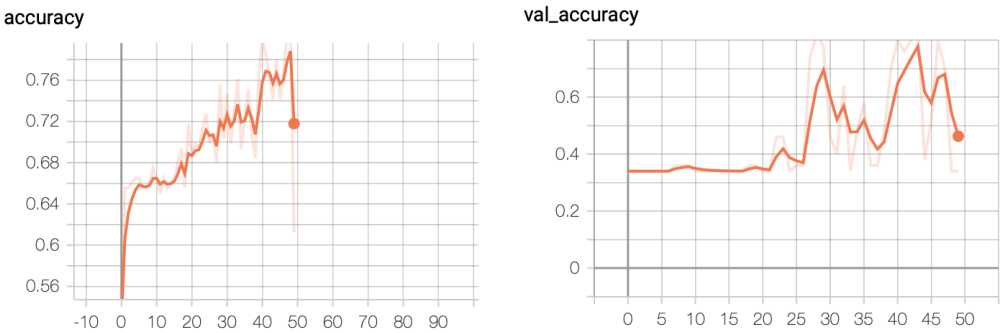


Figure 31: Training(on left) and testing(on right) accuracy's for learning rate = 0.01

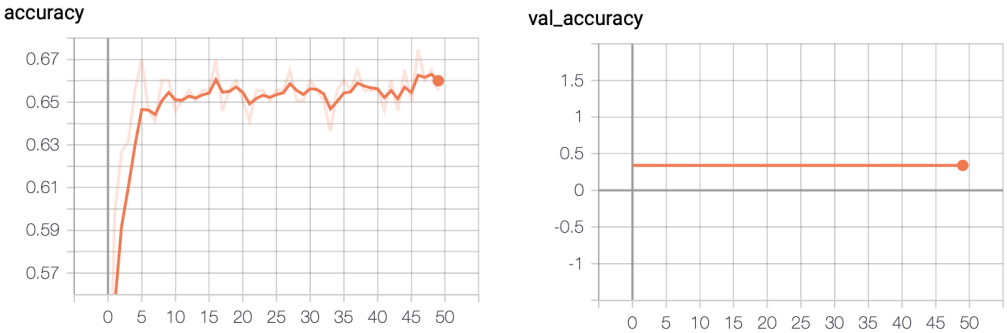


Figure 32: Training(on left) and testing(on right) accuracy's for learning rate = 0.001

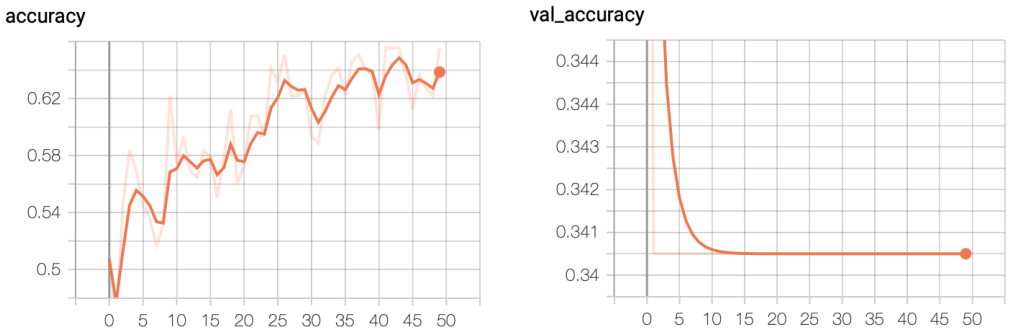


Figure 33: Training(on left) and testing(on right) accuracy's for learning rate = 0.0001

As stated in the section above (training the model with SGD optimizer), the model is expected to converge slowly. Also in the previous parts the effect of the randomly initialized weights have been indicated. Just like expected, with high learning rates like 1 and 0.1 the model was passing the local minima point for convergence which resulted with an instant drop in the accuracy value. When the loss values of these functions are examined, it is observed that at a point the loss value makes a pick and then the ability of training the weights is gone. This can be concluded with the property of deep learning that it's success is based on finding a local minima. Also as the accuracy values of a SGD optimizer is expected to fluctuate at a point due to its nature, the optimal learning rate can be said as in between 0.1 and 0.01. Here the characteristics of the SGD optimizer is preserved[3] and with further training the reaching local minima is still a possibility. However, this value is assumed as closer to 0.01. For low values, it can be said from **both** training and validation accuracy plots the number of epochs was not sufficient as the convergence is so slow.

In this discussion the important point is not the final accuracy, but finding the optimal learning curve for a SGD optimizer. Considering these plots, the optimal value is considered as close to 0.01. However, as a fact of deep learning the limited amount of data is also a barrier on receiving optimal results.

## Discussion

As the final part of the assignment, in this section the overall training process and comments about the process. Due to the approach that AdaDelta offers and the characteristics of SGD optimization algorithm, with the limited amount of data AdaDelta was the algorithm that performs better for this task. This fact is already discussed in the previous sections. Briefly, with SGD algorithm the updates are done for each sample and for a generalization of weights multiple iterations with large data is required for maintaining that no bias is introduced during training. Here AdaDelta is an adaptive algorithm that adjusts itself according to the dataset, which is the main result that it was more successful. However, since the adaptation of the algorithm is not sufficient with the amount of data available the performance of the classifier is limited.

For the question regarding the learning rates as discussed above, the key point is convergence. The process is mainly about traversing the data points on the plane (or any other multi dimensional surface mathematically) and finding a local minima that minimizes the loss function. By intuition, with high learning rates it is more probable that this optimal point is surpassed with high epochs and with low learning rate, further training is required. As it can be observed from the case of SGD optimizer with learning rate = 1, at the beginning of the training process the optimal point seems to be surpassed. This info is observed with the loss plot and the accuracy plot. Supporting the argument, with a learning rate such as 0.001, it is observed that some kind of training is observed but for optimal results 50 epochs is not enough and further training is required. As a geometrical approach to the process the learning rate is analogous with the amount of steps taken to the optimal point, where only a constant amount of steps can be taken in a turn. With high learning rate, it is highly probable that the optimal point is missed whereas with low learning rate it is possible that the weights still have steps to go to the optimal point. Here it is important to find the middle point, which both ensures convergence and computational efficiency (amount of epochs required) at the same time. Here that value was more close to 0.01.

# References


Budhiraja, A. (2016). *Dropout in (deep) machine learning.* Retrieved 2020-03-23, from https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Keras. (2020). *Usage of callbacks.* Retrieved 2020-03-23, from https://keras.io/callbacks/

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.* Retrieved 2020-03-29, from https://ruder.io/optimizing-gradient-descent/index.html

Sharma, S. (2017). *Activation functions in neural networks.* Retrieved 2020-03-23, from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6