Lab Work 2: Multiclass Support Vector Machine (SVM) and Two-layer Neural Networks (4 marks)

In this assignment, we continue the task of image classification on the CIFAR-10 labelled dataset. You will implement two different classification algorithms: SVM and a two-layer neural network. The work programme regarding two pathways are provided below.

1. SVM classification (2 marks):

SVM is a linear classification, supervised learning algorithm that works to identify the optimal hyperplane for linearly separable patterns. The final output of the model is a class identity. The SVM uses a linear function which assumes a boundary exists that separates one class boundary from another. The primary goal of the SVM is to efficiently find the boundary which separates one class from another. In this implementation, we will utilize a linear score function to compute a class score for the input data set. The output score can then be used within a loss function to better determine the success of the linear score function. Stochastic Gradient Descent will then be utilized as the optimization algorithm to minimize the loss obtained by the loss function.

1.1. Loading dataset and data pre-processing

TODO: Similar to the KNN-implementation, the first step in this process is to load the raw CIFAR-10 data into python. Prior to training the model, the data must be partitioning accordingly. From the 50,000 images within the training data, 49,000 images will be grouped as the official training set while the remaining 1,000 images will be designated as a validation set. The validation set will be used to tune the learning rate and regularization strength. From the 10,000 images within the test data set, a subsample of 1,000 images will be used to evaluate the accuracy of the SVM. A separate development data set of 500 randomly selected images will be created for use during development.

```
# Split the data into 4 data sets, training, validation, test, and dev.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Create validation set based on last 1000 images from training set
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Create training set based on first 49000 images within original training set
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# Create development subset of 500 random sampled images from training set
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# Create test set from first 1000 images within original test set
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]
```

TODO: Each image is then reshaped from a 3-dimensional, (32 x 32) matrix, into a single 3072 element array. The end result for each image set is a 2-D (***i*** x 3072) array of ***i*** images.

TODO: Finally, we'll normalize the dataset by subtracting the mean of the training data from each of the datasets outlined above. Subtracting the dataset mean centers the data and helps to keep the feature dataset within a similar range of each other which is beneficial when processing the gradient.

1.2. Linear Classification

1.2.1. Linear Score Funcion

The SVM is implemented by first computing the linear score function of the data set. The linear score function uses the dot product of the input training data, $\mathbf{x}_i$, and the randomly generated weight matrix ***W*** with a shape of (3073 x 10). Note that the length of the first dimension matches the total number of (pixels + 1) within each image. The additional pixel, (which was also added to each image set) is the addition of the bias vector ***B*** which influences the output score without directly interacting with the input training data.

```
# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001
```

From these parameters we arrive at the following function $f(x_i, W, b) = Wx_i + b$. At this point we can make a few observations about the linear score function. The dot product of the two matrices $W * x_i$ results in an array of 10 separate scores for each image.

1.2.2. Loss Function

Simply put, the loss function quantifies the accuracy of the linear score function by comparing the score of the correct class against the scores of the incorrect classes. Using the linear score function, we calculate the score s_j for the j–th class $s_j = f(x_i, W)_j$, as well as the score of the correct class $s_{y_i} = f(x_i, W)y_i$ for image x_i. We then determine weather or not the score of the correct class s_{y_i} is greater than the incorrect class s_j by some fixed margin Δ. Doing this across the entire data set we end up with a final loss value, sometimes refereed to as a cost score.

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

Taking the average loss and applying a regularization penalty of ***R(W)*** weighted by the hyperparameter λ, we obtain the following.

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

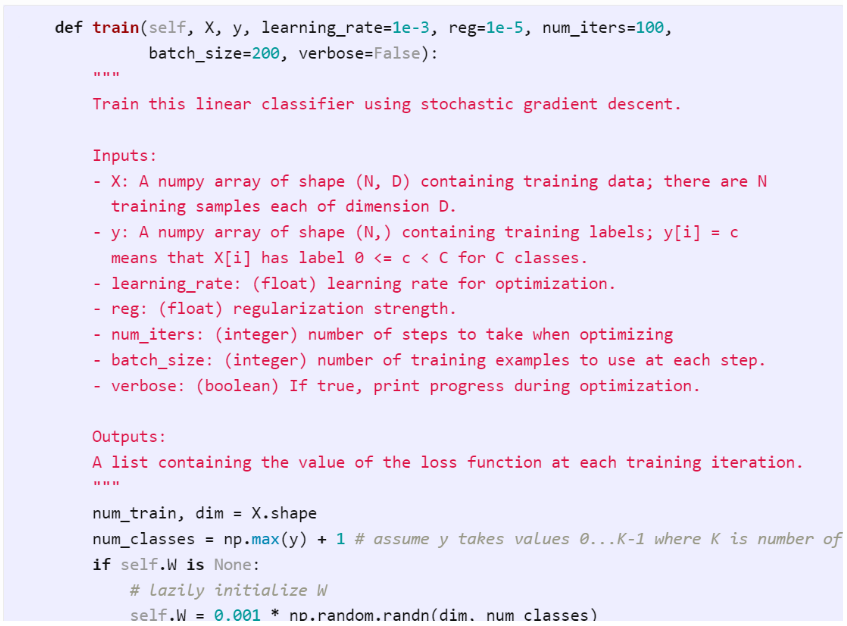
$$L = \frac{1}{N} \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta) + \lambda R(W)$$

1.2.3. Stochastic Gradient Descent (SGD)

Having defined the loss function, we now want to find the values of ***W*** which minimize the loss function. Starting with a random set of weights, we can iteratively refine the values to produce a slightly better score than previously defined. This optimization process is accomplished most affectively by calculating the gradient of the loss function with respect to the input weights.

$$\nabla L_i = \nabla \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

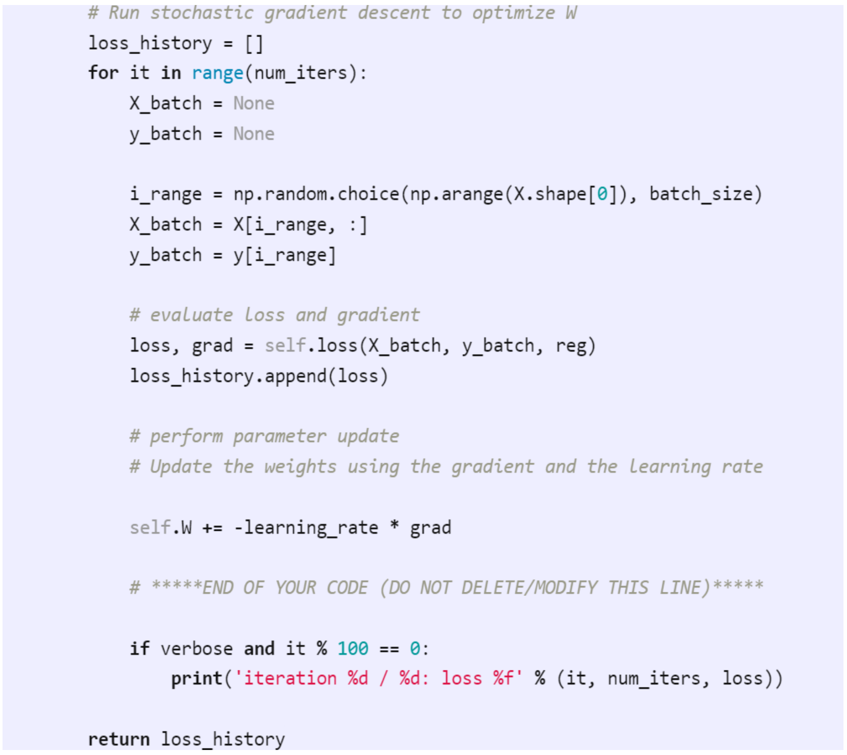
Having defined an efficient expression for the loss and gradient, we can train the model by adjusting the input weights using the gradient. However, rather than calculating the gradient for every input image x_i, this will be done iteratively over several sub samples of the data set, (i.e. Stochastic Gradient Descent). With each sample iteration, a small adjustment will be made to the weight matrix using a predefined learning rate and the gradient calculation.



```
    def train(self, X, y, learning_rate=1e-3, reg=1e-5, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this linear classifier using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there are N
          training samples each of dimension D.
        - y: A numpy array of shape (N,) containing training labels; y[i] = c
          means that X[i] has label 0 <= c < C for C classes.
        - learning_rate: (float) learning rate for optimization.
        - reg: (float) regularization strength.
        - num_iters: (integer) number of steps to take when optimizing
        - batch_size: (integer) number of training examples to use at each step.
        - verbose: (boolean) If true, print progress during optimization.

        Outputs:
        A list containing the value of the loss function at each training iteration.
        """
        num_train, dim = X.shape
        num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of
        if self.W is None:
            # lazily initialize W
            self.W = 0.001 * np.random.randn(dim, num_classes)
```



```
        # Run stochastic gradient descent to optimize W
        loss_history = []
        for it in range(num_iters):
            X_batch = None
            y_batch = None

            i_range = np.random.choice(np.arange(X.shape[0]), batch_size)
            X_batch = X[i_range, :]
            y_batch = y[i_range]

            # evaluate loss and gradient
            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            # perform parameter update
            # Update the weights using the gradient and the learning rate

            self.W += -learning_rate * grad

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

        return loss_history
```

Using the following input parameters when training the SVM, the loss converges to an approximate value of 5. **Below are the expected observations (2 marks)**.

```
svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, batch_size=200, verbose=True)
```

```
iteration 0 / 1500: loss 411.229123
iteration 100 / 1500: loss 242.353926
iteration 200 / 1500: loss 147.521380
iteration 300 / 1500: loss 90.800391
iteration 400 / 1500: loss 56.806474
iteration 500 / 1500: loss 36.503065
iteration 600 / 1500: loss 23.774250
iteration 700 / 1500: loss 15.723157
iteration 800 / 1500: loss 11.630423
iteration 900 / 1500: loss 9.050186
iteration 1000 / 1500: loss 7.439663
iteration 1100 / 1500: loss 6.954062
iteration 1200 / 1500: loss 5.243393
iteration 1300 / 1500: loss 5.179514
iteration 1400 / 1500: loss 5.546934
```

2. Two-layer Neural Network (2 marks):

In this task, we'll work through the steps and theory for developing a Two Layer Neural Network. *The initial model will be generated and tested using a toy data set. Once we're able to obtain ideal testing and validation results using the toy dataset, we'll extend the application of the Two Layer Neural Network onto the CIFAR-10 labeled dataset for image classification*.

2.1. Neural Network Architecture

For this implementation of a neural network, we'll utilize the softmax scoring function as our linear classifier and extend the scoring process across two separate layers consisting of a hidden layer and final output layer. As a general example, we might generate two weight matrices, W1 of shape (3073 x 100) and W2 of shape (100 x 10), where the larger matrix W1will learn to identify larger features in the data set while the smaller weight matrix W2will learn to identify smaller, detail specific features. The final score is then computed as follows. $s=W2*\max(0,X*W1)$.
In the above expression, the max() function, otherwise known as a ReLU activation function, works as a non-linearity which provides a separation between the two weight matrices W1 and W2. This separation in turn is what allows us to train each set of parameters through SGD. This implementation of a neural network effectively executes a series of linear mapping functions which are then tied together through activation functions, (i.e. non-linear functions). A simple visualization of this network is shown in Figure 1.

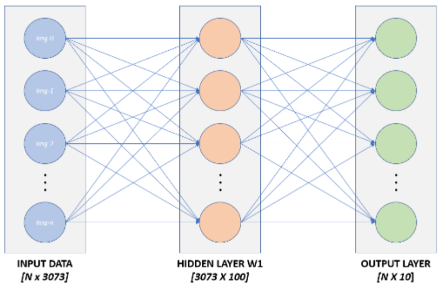


Figure 1: *Graphical Representation of a Two Layer Neural Network*

2.2. Development (Toy Model)

Having a basic foundation for the architecture of the network, we begin development using a toy data set of arbitrary data X and the associated labels y. Looking at the input data set X, we see that we have a total of 5, 4-dimensional data points in the $(5 \text{ x } 4)$ matrix.

```
print(X)

[[ 16.24345364  -6.11756414  -5.28171752 -10.72968622]

 [  8.65407629 -23.01538697  17.44811764  -7.61206901]

 [  3.19039096  -2.49370375  14.62107937 -20.60140709]

 [ -3.22417204  -3.84054355  11.33769442 -10.99891267]

 [ -1.72428208  -8.77858418   0.42213747   5.82815214]]


print(y)

[0 1 2 2 1]
```

While creating this data set, we simultaneously generate an instance of our two-layer neural network.

```
net = TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)
```

- input_size: Dimension D
  - Captures the second dimension of the training data set, (i.e. number of columns)
  - Toy Data Set: D = 4
  - CIFAR-10 Set: D = 3072
- hidden_size:
  - Specifies the size of the second dimension within the first weight matrix W1
    - Toy Data Set: 10
    - CIFAR-10 Set: Range between 50 and 300 (discussed below)
- num_classes:
  - Total number of classes in final output
    - Toy Data Set: 5
    - CIFAR-10 Set: 10

2.2.1 Weight Initialization

Using the input parameters defined above, the class instantiation generates weight and bias parameters internally. Both weight matrices are initialized to small random values which are further scaled according to the input parameter *std*. Bias terms are initialized to zero, however these values are updated at a later step while training the model. Note that the random seed is set to a value of zero which produces the same set of random variables with every instance. This allows for repeatable testing and trouble shooting during development.

```
self.params = {}

np.random.seed(0)

self.params['W1'] = std * np.random.randn(input_size,
hidden_size)

self.params['b1'] = np.zeros(hidden_size)

self.params['W2'] = std * np.random.randn(hidden_size,
output_size)

self.params['b2'] = np.zeros(output_size)
```

2.2.2. Forward Pass Computation and Activation Function

The forward pass defines the steps in which we execute linear mapping, activation functions, apply regularization, and calculate the loss of the new scoring function. Using the toy data set X defined above, the forward pass computation starts by calculating the linear score function of the input data set and the first weight matrix $W1$.

```
# First layer linear score function

h1 = X.dot(W1) + b1

# First layer activation function

a1 = np.maximum(0, h1)

# Output Layer score function

scores = a1.dot(W2) + b2
```

Using a biological neuron as a model, the activation function is used to model the firing rate of a neuron. We model this functionality using the Rectified Linear Unit, (ReLU), function which computes the function $f(x)=max(0,x)$.

2.2.3. Loss Function and Regularisation

We can implement the loss function for a Softmax classifier. The loss of the class scores using the loss function is defined as:

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$$

As mentioned in the same section, "the full Softmax classifier loss is then defined as the average cross-entropy loss over the training examples and the regularization".

$$L = \frac{1}{N}\sum_i L_i + \frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2$$

```
exp_scores = np.exp(scores)

probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

corect_logprobs = -np.log(probs[range(N), y])

data_loss = np.sum(corect_logprobs) / N

reg_loss = 0.5 * reg * np.sum(W1 * W1) + 0.5 * reg * np.sum(W2 *
W2)

loss = data_loss + reg_loss
```

Regularization allows us to prevent over fitting by causing the network to generalize information affectively thus improving overall performance. In this section, we implement L2 regularization, which has the affect of penalizing the squared magnitude of the weight matrices W.

2.2.4. SGD through Backpropagation

Below is the calculation of gradients.

```
# compute gradient on scores

dscores = probs

dscores[range(N),y] -= 1

dscores /= N
```

With the gradient for the scores available, we can now backpropogate through the two layer network and calculate the gradient with respect to the weight and bias parameters. The final implementation is given below.

```
# gradient for W2 and b2

grads['W2'] = np.dot(a1.T, dscores)

grads['b2'] = np.sum(dscores, axis=0)

# Backpropagate to h1

dh1 = np.dot(dscores, W2.T)

# Backpropagate ReLU non-linearity
```

```
dh1[a1 <= 0] = 0

#

grads['W1'] = np.dot(X.T, dh1)

grads['b1'] = np.sum(dh1, axis=0)

grads['W2'] += 2 * reg * W2

grads['W1'] += 2 * reg * W1
```

2.2.5. Parameters Updates

Having calculated the gradients, we can now perform a parameter optimization. For this implementation, we'll stick with the "vanilla update" which updates the parameters along the negative gradient direction and scales the update according to the learning rate constant.

```
self.params['W1'] += -learning_rate * grads['W1']

self.params['b1'] += -learning_rate * grads['b1']

self.params['W2'] += -learning_rate * grads['W2']

self.params['b2'] += -learning_rate * grads['b2']
```

2.3. Training

Finally, we want to train and evaluate the performance of our two-layer neural network. We'll first create an instance of the class object TwoLayerNet() and then provide the class instance with the toy data set and the hyperparameters "learning rate" and "regularization strength".

```
net = init_toy_model()

stats = net.train(X, y, X, y,

            learning_rate=1e-1, reg=5e-6,

            num_iters=100, verbose=False)
```

Without speaking to the provided code in the train() function, the first task is to generate a mini batch of the training data. One of the reasons for doing this step is to avoid degradation of the model quality. The three contrasting types of gradient descent are Stochastic Gradient Descent, Batch Gradient Descent, and Mini-Batch Gradient Descent, (each of which having their own pros and cons). Stochastic gradient Descent updates the parameters after each example within the training data set. Batch Gradient Descent calculates the error for all training examples, (i.e. one training epoch) before updating the parameters. Mini-batch Gradient Descent attempts to combine the two methodologies by lumping the training data into mini-batches. Updates to the parameters are made after evaluating the error for each example within the mini-batch.

```
idx = np.random.choice(range(X.shape[0]), batch_size)

X_batch = X[idx]

y_batch = y[idx]
```

Sticking with the mini-batch implementation as noted by the assignment instructions, we then calculate the loss and gradient for the current batch and update the parameters as follows.

```
self.params['W1'] += -learning_rate * grads['W1']

self.params['b1'] += -learning_rate * grads['b1']

self.params['W2'] += -learning_rate * grads['W2']

self.params['b2'] += -learning_rate * grads['b2']
```

2.4. Development on CIFAR-10

Similar to the knn-implementation and SVM-Classifier, the first step in this process is to load the raw CIFAR-10 data into python. After data preprocessing, which includes zero averaging the data set and organizing each image into a single column vector, we end up with the following data sets. For clarity, the shape of each data set is noted.

```
X_train: Train data shape:  (49000, 3072)

y_train: Train labels shape:  (49000,)

X_val: Validation data shape:  (1000, 3072)

y_val: Validation labels shape:  (1000,)

X_test: Test data shape:  (1000, 3072)

y_test: Test labels shape:  (1000,)

# Normalize the data: subtract the mean image

mean_image = np.mean(X_train, axis=0)

X_train -= mean_image

X_val -= mean_image

X_test -= mean_image
```

With minimal effort, we can create a neural net class instance and train the network using the CIFAR-10 data set. Note that the train function calls the loss function internal which calculates the loss and gradients discussed above.

```
input_size = 32 * 32 * 3
```

```
hidden_size = 50

num_classes = 10

net = TwoLayerNet(input_size, hidden_size, num_classes)

print(net.params['W1'].shape)

# Train the network

stats = net.train(X_train, y_train, X_val, y_val,

            num_iters=1000, batch_size=200,

            learning_rate=1e-4, learning_rate_decay=0.95,

            reg=0.25, verbose=True)
```

Setting the verbose parameter equal to True, we can view the convergence of the loss function as we train the network. From this simple output, we see that the results are not favorable, converging at a value around 1.967. Moreover, testing the trained network against the validation data set, we receive an accuracy of 28.2%. Comparing these results to that of the SVM from the previous section, where we obtained an accuracy of 38.9%, we can conclude that the network is not performing optimally. This leads us to the task of debugging and hyper parameter tuning.

Below are the expected observations (2 marks).

```
iteration 0 / 1000: loss 2.302976

iteration 100 / 1000: loss 2.302638

iteration 200 / 1000: loss 2.299256

iteration 300 / 1000: loss 2.270635

iteration 400 / 1000: loss 2.219117

iteration 500 / 1000: loss 2.141064

iteration 600 / 1000: loss 2.129066

iteration 700 / 1000: loss 2.047454

iteration 800 / 1000: loss 1.988162

iteration 900 / 1000: loss 1.967131

# Predict on the validation set

val_acc = (net.predict(X_val) == y_val).mean()

print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.282
```

Bonus (no marks).

As stated, we wish to optimize our network through hyper parameter tuning. Another hyperparameter to analyze is the learning rate. Large learning rates have the tendency to result in unstable training, causing the model to converge rapidly to a suboptimal solution. Small leaning rates on the other hand risk getting stuck and fail to train the network.

```
best_net = None # store the best model into this

best_val = -1

best_stats = None

h = [100, 150, 200]

learning_rates = [1e-3, 1e-4, 1e-5]

regularization_strengths = [0.3, 0.4, 0.5]

results = {}

iters = 3000

for hidden_size in h:

    for lr in learning_rates:

        for rs in regularization_strengths:

            net = TwoLayerNet(input_size, hidden_size,
num_classes)

            # Train the network

            stats = net.train(X_train, y_train, X_val, y_val,

                        num_iters=iters, batch_size=200,

                        learning_rate=lr,
learning_rate_decay=0.95,

                        reg=rs, verbose=True)

            # Make predictions against training set

            train_pred = net.predict(X_train)

            # Get average training prediction accuracy

            train_acc = np.mean(y_train == y_train_pred)

            # Make predictions against validation set
            val_pred = net.predict(X_val)

            # Get average validation prediction accuracy

            val_acc = np.mean(y_val == val_pred)

            # Store results in dictionary using hyperparameters
as key values

            results[(hidden_size, lr, rs)] = (hidden_size,
train_acc, val_acc)

            # Update best validation accuracy if better results
are obtained

            if val_acc > best_val:

                best_stats = stats

                best_val = val_acc

                best_net = net



# Print out results.

for h, lr, reg in sorted(results):

    hidden_size, train_accuracy, val_accuracy = results[(h, lr,
reg)]

    print('h %s lr %e reg %e train accuracy: %f val
accuracy: %f' % (

                h, lr, reg, train_accuracy, val_accuracy))



print('best validation accuracy achieved during cross-
validation: %f' % best_val)

h 100 lr 1.000000e-05 reg 3.000000e-01 train accuracy: 0.620776
val accuracy: 0.194000

h 100 lr 1.000000e-05 reg 4.000000e-01 train accuracy: 0.620776
val accuracy: 0.194000

h 100 lr 1.000000e-05 reg 5.000000e-01 train accuracy: 0.620776
val accuracy: 0.194000

h 100 lr 1.000000e-04 reg 3.000000e-01 train accuracy: 0.620776
val accuracy: 0.391000

h 100 lr 1.000000e-04 reg 4.000000e-01 train accuracy: 0.620776
val accuracy: 0.390000
```

```
.

.

.

best validation accuracy achieved during cross-validation:
0.52000
```

Finally, we implement the model of learned weights using the test data set. From this step we obtain a test accuracy of 52%.

```
test_acc = (best_net.predict(X_test) == y_test).mean()

print('Test accuracy: ', test_acc)

Test accuracy:  0.52
```

3. Sign-off:

You don’t need to submit your code. Show your solution to the demonstrator to sign-off.