

### Homework 3 - Neural Network Models - Spring 2025

**Essay Question:** *The current generation of deep neural network (NN) models (2010's-present) addressed many of the issues faced by model sets used in earlier waves of neural network research.* Write a one paragraph essay that starts with the preceding sentence. The following sentences of the paragraph should support this assertion by describing the novelty of each wave's model sets, the issues that limited the utility of these earlier models, and how deep NN models address these issues. Your paragraph should close with your opinion concerning the future prospects of the current generation of deep NN models and should discuss why you feel this way.

**Problem 1:** Consider a linear regression problem whose inputs  $x \in \mathbb{R}^n$  and whose targets are  $y = \theta^T x + n \in \mathbb{R}$  where  $\theta \in \mathbb{R}^n$  is some unknown parameter vector and  $n$  is a zero mean normally distribution random variable with variance  $\sigma^2$ . Let the model set,  $\mathcal{H}$ , consist of all linear models of the form  $h_w(x) = w^T x$  where  $w \in \mathbb{R}^n$  is the model's *weight vector*. Let the problem's loss function be the squared prediction error  $L(y, h_w(s)) = (y - h_w(s))^2$ .

Let  $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$  be a randomly selected dataset of  $N$  samples and let  $w^* \in \mathbb{R}^n$  be the model weight vector that minimizes the model's empirical risk over the dataset. Define the matrices,

$$\mathbf{X} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_N^T \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \quad \hat{\mathbf{Y}} = \begin{bmatrix} h_{w^*}(x_1) \\ h_{w^*}(x_2) \\ \vdots \\ h_{w^*}(x_N) \end{bmatrix}, \quad \mathbf{n} = \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_N \end{bmatrix}$$

where  $x_k$  are the data input samples,  $y_k = \theta^T x_k + n_k$  are the data targets,  $n_k$  is the noise in the data targets, and  $w^*$  are the model weights minimizing the model's empirical risk over the dataset.

1. Let  $w^*$  be the optimal model weights for a given dataset  $\mathcal{D} = \{\mathbf{X}, \mathbf{Y}\}$  with  $N$  samples. Determine a closed form expression for  $w^*$  as a function of the unknown system parameters,  $\theta$ , the input data matrix  $\mathbf{X}$  for the given dataset, and the target noise vector,  $\mathbf{n}$ .
2. For the optimal model,  $h_{w^*}$ , determined above, find an expression for the true risk,  $\mathbb{E}[h_{w^*}]$ , as a function of the target noise covariance,  $\sigma^2$ .
3. Let us define the following matrix

$$\mathbf{H} \stackrel{\text{def}}{=} \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$$

Determine an expression for the expected value of the risk difference,  $\mathbb{E}_{\mathcal{D}} \left\{ \hat{R}_{\mathcal{D}}[h_{w^*}] - R[h_{w^*}] \right\}$  for a optimal models  $h_{w^*}$  for the randomly selected datasets  $\mathcal{D}$ .

**Hint:** You will probably need to use the fact that  $\mathbf{H}$  is a symmetric matrix with  $\text{trace}(\mathbf{H}) = n$  and  $\mathbf{H}^2 = \mathbf{H}$ .

**Solution (optimal  $w^*$ ):** The empirical risk is

$$\hat{R}_{\mathcal{D}}[h_w] = \frac{1}{N} |\mathbf{X}w - \mathbf{Y}|^2$$

taking the gradient with respect to  $w$  and setting equal to zero

$$\nabla_w \hat{R}_{\mathcal{D}}[h_w] = \frac{2}{N} (\mathbf{X}^T \mathbf{X} w - \mathbf{X}^T \mathbf{Y}) = 0$$

which gives  $w^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$ . We now note that  $\mathbf{Y} = \mathbf{X}\theta + \mathbf{n}$ . Inserting this into the above equation and simplifying gives

$$\begin{aligned} w^* &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{X}\theta + \mathbf{n}) \\ &= \theta + (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{n} \end{aligned}$$

**Solution (true risk):** The true (actual) risk for the optimal model obtained in part 1 is

$$\begin{aligned} R[h_{w^*}] &= \mathbb{E}_{x,y} [(y - h_{w^*}(x))^2] = \mathbb{E}_{x,n} [(x^T(\theta - w^*) + n)^2] \\ &= \mathbb{E}_{x,n} [(x^T(\theta - (\theta + (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{n}))) + n)^2] \\ &= \mathbb{E}_{x,n} [(x^T (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{n} + n)^2] = \sigma^2 \end{aligned}$$

because  $x$  and  $n$  are uncorrelated from each other and  $n$  is zero mean.

**Solution (risk difference):** The empirical risk of the optimal model  $h_{w^*}$  is

$$\begin{aligned} \hat{R}_{\mathcal{D}}[h_{w^*}] &= \frac{1}{N} |\mathbf{X} w^* - \mathbf{Y}|^2 \\ &= \frac{1}{N} |\mathbf{X}(\theta + (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{n}) - (\mathbf{X}\theta + \mathbf{n})|^2 \\ &= \frac{1}{N} |\mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{n} - \mathbf{n}|^2 = \frac{1}{N} |(\mathbf{H} - \mathbf{I})\mathbf{n}|^2 \\ &= \frac{1}{N} [\mathbf{n}^T \mathbf{H}^T \mathbf{H} \mathbf{n} - \mathbf{n}^T (\mathbf{H} + \mathbf{H}^T) \mathbf{n} + \mathbf{n}^T \mathbf{n}] \\ &= \frac{1}{N} \mathbf{n}^T (\mathbf{I} - \mathbf{H}) \mathbf{n} \end{aligned}$$

So now take the expected value over the datasets,

$$\begin{aligned} \mathbb{E}_{\mathcal{D}} [R[h_{w^*}] - \hat{R}_{\mathcal{D}}[h_{w^*}]] &= \mathbb{E}_{\mathbf{n}} [\mathbf{n}^T (\mathbf{I} - \mathbf{H}) \mathbf{n}] \\ &= \frac{1}{N} \mathbb{E}_{\mathbf{n}} [\mathbf{n}^T \mathbf{n}] - \frac{1}{N} \mathbb{E}_{\mathbf{n}} [\mathbf{n}^T \mathbf{H} \mathbf{n}] \\ &= \sigma^2 - \frac{\sigma^2}{N} \text{trace}(\mathbf{H}) = \sigma^2 \left(1 - \frac{n}{N}\right) \end{aligned}$$

**Problem 2:** Consider the neural network model shown in Fig. 1 that uses the notational conventions from the lecture notes (chapter 3) in which  $\mathbf{x}^{(\ell)}$  and  $\mathbf{s}^{(\ell)}$  are the outputs and inputs, respectively, of the nodes in the  $\ell$ th layer (including the bias node). Let  $\mathcal{W} = \{\mathbf{W}^{(\ell)}\}_{\ell=1}^3$  be a collection of weights where  $\mathbf{W}^{(\ell)}$  is the weight matrix connecting the outputs from layer  $\ell - 1$  to nodes of layer  $\ell$ . Let  $\hat{\mathbf{y}}_k = h_{\mathcal{W}}(\mathbf{x}_k)$  denote the prediction made by the model for the  $k$ th input  $\mathbf{x}_k$ . Assume that all initial weights are 0.5 and that we are using the squared error loss function  $L(\mathbf{y}_k, \hat{\mathbf{y}}_k) = |\mathbf{y}_k - \hat{\mathbf{y}}_k|^2$ .

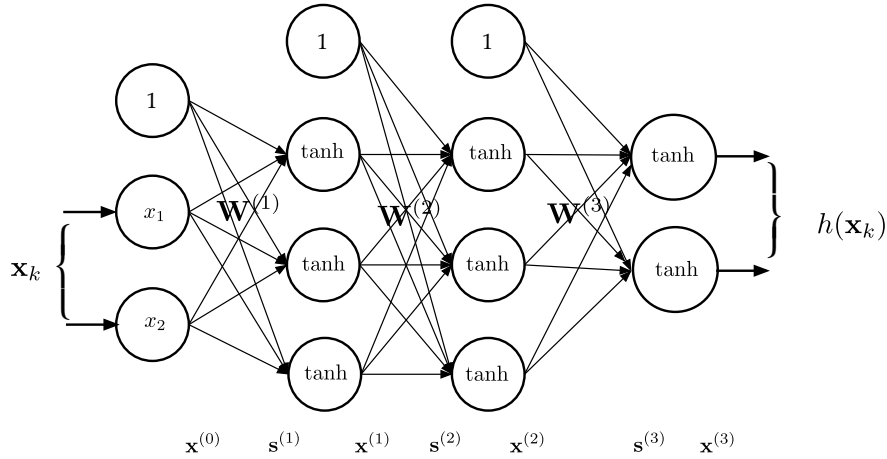


Figure 1: Problem 2

1. Compute the forward pass to obtain  $\mathbf{x}^{(\ell)}$ ,  $\mathbf{s}^{(\ell)}$ , and  $h(\mathbf{x}_k)$  for all layers assuming the input  $\mathbf{x}_k = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$  and that the target is  $\mathbf{y}_k = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ . Show the equations and scripts you used to compute these items.
2. Use the backward pass to obtain the error sensitivities,  $\delta^{(\ell)}$  for each layer with the same input/target and then compute the updated weights for each layer assuming a learning rate,  $\eta$ , of 0.1 for the given dataset sample input  $\mathbf{x}_k = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$  and target  $\mathbf{y}_k = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

**Solution:** The following MATLAB script computes the forward path

```
%initial weights
W{1} = .5*ones(3,3);
W{2} = .5*ones(4,3);
W{3} = .5*ones(4,2);

%input and target
input = [1;1];
target = [0;1];

%forward propagation
x0 = [1;input];
x_in = x0;

fprintf("inputs (s) and outputs (x)\n")
for i=1:3;
    s{i} = W{i}'*x_in;
    disp(s{i}')
    x{i} = [1; tanh(s{i})];
    disp(x{i}')
    x_in = x{i};
end;
```

```
fprintf("model output yhat\n")
yhat = x{3}(2:end);
disp(yhat)
```

The outputs are

```
inputs (s) and outputs (x)
    1.5000    1.5000    1.5000

    1.0000    0.9051    0.9051    0.9051

    1.8577    1.8577    1.8577

    1.0000    0.9525    0.9525    0.9525

    1.9287    1.9287

    1.0000    0.9586    0.9586

model output yhat
    0.9586
    0.9586
```

We now perform the backward pass using the following script

```
fprintf("sensitivity (del)\n")
Wdel = 2*(yhat-target);
for i=1:3
    del{4-i} = (1-tanh(s{4-i}).^2).*Wdel;
    disp(del{4-i}')
    Wdel = W{4-i}*del{4-i};
    Wdel = Wdel(2:end);
end;

gradW{1} = x0*del{1}';
gradW{2} = x{1}*del{2}';
gradW{3} = x{2}*del{3}';
lr = 1;

fprintf("updated weights\n")
disp(W{1}-lr*gradW{1})

disp(W{2}-lr*gradW{2})

disp(W{3}-lr*gradW{3})
```

The outputs are

```

sensitivity (del)
0.1554 -0.0067

0.0069 0.0069 0.0069

0.0019 0.0019 0.0019

updated weights
0.4981 0.4981 0.4981
0.4981 0.4981 0.4981
0.4981 0.4981 0.4981

0.4931 0.4931 0.4931
0.4938 0.4938 0.4938
0.4938 0.4938 0.4938
0.4938 0.4938 0.4938

0.3446 0.5067
0.3520 0.5064
0.3520 0.5064
0.3520 0.5064

```

**Notebook Assignment (Deep Learning Classes):** Deep learning often requires extremely large datasets of labeled data. While it may not be hard to gather the input samples,  $x_k$ , of the dataset, labeling that data to generate targets,  $y_k$ , is expensive and time consuming. One way of addressing this issue is through *transductive inference*<sup>1</sup> (a.k.a. semi-supervised learning) that reasons from specific (training) cases to specific (test) cases. This approach may be possible when the data inputs are related to each other in a manner that can be described by a *graph*. In this case, the additional information provided by an input samples relationship to other samples may provide a way to infer the class membership of unlabeled samples.

This notebook assignment uses python class objects to instantiate a neural network model for transductive inference. Chapter 3 gave an example of such classes for a `SequentialModel` constructed from `DenseLayer` objects. This notebook develops class objects for a *Graph Convolution Network* (GCN)<sup>2</sup>. The GCN model is, essentially, a sequential model formed from `GCNLayers`. The input to the resulting model is the normalized *adjacency matrix*,  $\mathbf{A} \in \mathbb{R}^{N \times N}$ , for a graph with  $N$  nodes and a feature matrix  $\mathbf{X} \in \mathbb{R}^{N \times n}$  whose  $k$ th row is the *feature vector* for the  $k$ th node. The model's output is a matrix  $\hat{\mathbf{Y}} \in \mathbb{R}^{N \times m}$  whose  $k$ th row estimates the probability of node  $k$ 's class membership where there are  $m$  distinct classes.

Transductive inference is performed using a GCN in the following manner. The dataset is a labelled graph,  $G = (V, E, x, y)$  where  $V$  is a set of *nodes*,  $E \subset V \times V$  is a set of *edges*,  $x : V \rightarrow \mathbb{R}^n$  maps each node,  $v \in V$ , onto a real-valued *feature vector*,  $x(v) \in \mathbb{R}^n$ , and  $y : V \rightarrow \{0, 1\}^m$  is a target for each node  $v$  such that  $y(v) \in \{0, 1\}^m$  is a one-hot encoded vector of one of  $m$  class names. The model set maps graph  $G$ 's adjacency matrix and a feature matrix  $X$  onto the

<sup>1</sup>O. Chapelle, B. Scholkopf, and A. Zien, Eds. Semi-Supervised Learning, MIT Press, 2006: see chapt. 24

<sup>2</sup>Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).

prediction  $\hat{\mathbf{Y}}$  of each node's class probabilities. What makes this different from the learning-by-example problem, is that rather than assuming the model is trained using all targets in  $\mathbf{Y}$ , we assume that the model is trained on a subset of  $\mathbf{Y}_{\text{train}}$  of training targets. The problem is whether the model can generalize to nodes in  $G$  whose labels were not in the training set  $\mathbf{Y}_{\text{train}}$ . In other words, can knowledge of the topology around training nodes be used to infer the class of nodes that were not in the training set.

This assignment has you write GCN class objects that are trained on the CORA dataset<sup>3</sup>. This dataset consists of 2708 scientific publications classified into one of seven classes. The dataset consists of 5419 links indicating which publications have been cited. Each publication in the dataset is described by a one-hot encoded word vector indicating the absence/presence of the corresponding word from a dictionary of 1433 unique words *and* a label indicating which class the publication belongs to. We have pre-processing the CORA dataset and stored it as a pkl (pickle) archive that you can load as

```
import numpy as np
import tensorflow as tf
import pickle

with open("cora_dataset.pkl", "rb") as fp:
    dataset = pickle.load(fp)
    A = dataset[0] #normalized adjacency matrix
    X = dataset[1] #node feature matrix
    Y = dataset[2] #node one-hot encoded targets
    classes = dataset[3] #np.array of class names
    train_mask = dataset[4] #Boolean array of training nodes
    test_mask = dataset[5] #Boolean array of test nodes
```

The preprocessed pkl file is in the course vault.

1. Create a `GcnLayer` class similar to the `DenseLayer` class we showed you in lecture. The class constructor has the following inputs `n_input`, `n_output`, and the activation function. The constructor initializes a weight array,  $\mathbf{W}$ , of shape  $(n\_input, n\_output)$ . These weights are randomly initialized as a `tf.Variable` with values between  $-1$  and  $1$ . The class' `__call__` method takes the adjacency matrix  $\mathbf{A}$  and feature matrix  $\mathbf{X}$  as an input. The output returned by `__call__` is

$$\sigma(\mathbf{AXW})$$

where  $\sigma$  is the activation function.

**Solution:** This is essentially identical to `DenseLayer`, with the only difference being in the `__call__` method.

```
class GcnLayer:
    def __init__(self, n_input, n_output, activation):
        self.activation = activation
        w_shape = (n_input, n_output)
        w_initial = tf.random.uniform(w_shape, minval=-1, maxval=1)
        self.W = tf.Variable(w_initial)
    def __call__(self, A, X):
        return self.activation(A@X@self.W)
    @property
    def weights(self):
        return [self.W]
```

2. Create a `GcnModel` class similar to the `SequentialModel` class shown to you in lecture. The only major difference is that this model keeps a list of instantiated `GcnLayer` objects where the output of the  $\ell$ th layer is used as the feature vector,  $\mathbf{X}$  for the  $\ell + 1$ st layer. Note

---

<sup>3</sup>Sen, Prithviraj, et al. "Collective classification in network data." *AI magazine* 29.3 (2008): 93-93.

that all layers in the model use the same adjacency matrix, **A**. Then use your classes to instantiate a graph convolution network (GCN) model with two `GcnLayers` where the first layer's output is a rank-2 tensor with shape (2308, 32). The second layer is a rank-2 tensor with shape (2308, 7).

**Solution:** This is nearly identical to the `SequentialModel` class. The only difference lies in how the layers are chained together

```
class GcnModel:
    def __init__(self, layers):
        self.layers = layers
    def __call__(self, inputs):
        A = inputs[0]
        x = inputs[1]
        for layer in self.layers:
            x = layer(A, x)
        return x
    @property
    def weights(self):
        weights=[]
        for layer in self.layers:
            weights += layer.weights
        return weights

#create model
n_hidden = 16
n_nodes, n_features = X.shape
n_classes = len(classes)
relu = tf.nn.relu
softmax = tf.nn.softmax
layer1 = GcnLayer(n_features, n_hidden, relu)
layer2 = GcnLayer(n_hidden, n_classes, softmax)
model = GcnModel([layer1, layer2])
```

3. Write a `evaluate` function whose inputs are the `GcnModel` model, the adjacency matrix, **A**, the dataset's feature matrix, **X**, the dataset's array of one-hot encoded targets, **Y**, and a Boolean valued mask, `mask`, on the graph's nodes. The function returns the average loss and average accuracy of the model over the graph nodes selected in `mask`. For the loss function use the categorical crossentropy function. Evaluate the masked loss and accuracy of the model using both the `train_mask` and `test_mask` in the dataset.

**Solution:** This also looks similar to the `evaluate` function in the lecture. What is different however is the fact that we are only evaluating loss and accuracy over a masked set of targets. This means that when we compute the sample average of the loss/accuracy, we zero those values not in the mask and make sure we compute the average only over those non-zeroed sampled. So I created two functions to compute the "masked" loss and accuracy.

```
def masked_loss_function(Y, Yhat, mask):
    cce = tf.keras.losses.categorical_crossentropy
    per_sample_losses = cce(Y, Yhat)
    sample_weights = tf.cast(mask, "float64") #mask is Boolean, we need to convert to float
    correction_factor = tf.reduce_mean(sample_weights)
    masked_losses = per_sample_losses * sample_weights
    loss = tf.reduce_mean(masked_losses) / correction_factor
    return loss

def masked_accuracy_function(Y, Yhat, mask):
    preds = np.argmax(Yhat, axis=1)
    targets = np.argmax(Y, axis=1)
    accuracy = (preds==targets)
    sample_weights = tf.cast(mask, "float64")
    correction_factor = tf.reduce_mean(sample_weights)
    masked_accuracy = accuracy * sample_weights
    accuracy = tf.reduce_mean(masked_accuracy) / correction_factor
    return accuracy

def evaluate(model, A, X, Y, mask):
    Yhat = model([A, X])
    masked_loss = masked_loss_function(Yhat, Y, mask)
    masked_acc = masked_accuracy_function(Yhat, Y, mask)
    return masked_loss, masked_acc

#evaluate initial model before training
```

```

train_loss, train_acc = evaluate(model, A, X, Y, train_mask)
print(f"\n train loss = {train_loss:.2f}, train accuracy = {100*train_acc:.2f}%")
test_loss, test_acc = evaluate(model, A, X, Y, test_mask)
print(f"\n test loss = {test_loss:.2f}, test accuracy = {100*train_acc:.2f}%")

# train loss = 14.11, train accuracy = 11.7%
# test loss = 14.25, test accuracy = 10.30%

```

- Write a `fit` function whose inputs are the `GcnModel` model, the adjacency matrix, `A`, the dataset's feature matrix `X`, the dataset's array of one-hot encoded targets, `Y`, the number of training epochs, the learning rate, and a Boolean valued mask on the graph's nodes. The output from the `fit` is the history of the masked loss and accuracy for the training and testing masks. Test your `fit` method on the `GcnModel` you instantiated in part 2; training for 1000 epochs with the training mask, `train_mask`, and a learning rate of  $\eta = .3$ . After training for 500 epochs, evaluate the model's testing loss and accuracy. Compare to the accuracy obtained with the untrained model.

**Solution:** The `fit` function has to use the masks provided in the dataset.

```

def fit(model, A, X, Y, n_epochs, lr, train_mask, test_mask):
    Yhat = model([A, X])
    train_loss, train_acc = evaluate(model, A, X, Y, train_mask)
    test_loss, test_acc = evaluate(model, A, X, Y, test_mask)
    history = [(train_loss, test_loss, train_acc, test_acc)]

    for epoch in range(n_epochs):
        with tf.GradientTape() as tape:
            Yhat = model([A, X])
            loss = masked_loss_function(Yhat, Y, train_mask)
            gradients = tape.gradient(loss, model.weights):
            for g, w in zip(gradients, model.weights):
                w.assign_sub(g*lr)
        train_loss, train_acc = evaluate(model, A, X, Y, train_mask)
        test_loss, test_acc = evaluate(model, A, X, Y, test_mask)
        history = np.vstack((history, (train_loss, test_loss, train_acc, test_acc)))
        if epoch%10==0:
            print('=', end='')
        if epoch%100==0:
            print(f"\nepoch {epoch}: training {train_loss:.2f}/{(train_acc*100:.2f)}% -
                    testing {test_loss:.2f}/{(test_acc*100:.2f)}%", end='')

    return history

#####now train it
n_epochs = 1000
lr = 0.2
history = fit(model, A, X, Y, n_epochs, lr, train_mask, test_mask)

train_loss, train_acc = evaluate(model, A, X, Y, train_mask)
print(f"\n train loss = {train_loss}, train acc = {100*train_acc}%")
test_loss, test_acc = evaluate(model, A, X, Y, test_mask)
print(f"\n test loss = {test_loss}, test accuracy = {100*test+acc}%")

```

After training for 1000 epochs with learning rate of 0.3 we get

```

epoch 0: training 14.00/14.00% - testing 14.14/11.96%=====
epoch 100: training 3.12/84.86% - testing 5.28/70.06%=====
epoch 200: training 1.87/89.43% - testing 4.38/74.30%=====
epoch 300: training 1.60/90.57% - testing 4.31/74.81%=====
epoch 400: training 1.55/90.57% - testing 4.17/75.53%=====
epoch 500: training 1.46/91.14% - testing 4.11/75.57%=====
epoch 600: training 1.41/91.43% - testing 4.09/75.49%=====
epoch 700: training 1.36/91.71% - testing 4.09/75.57%=====
epoch 800: training 1.18/92.86% - testing 4.11/75.45%=====
epoch 900: training 1.12/93.14% - testing 4.01/75.83%=====
train loss = 1.1184475861578715, train acc = 93.14285714285715%

test loss = 3.9894692070009135, test accuracy = 75.95419847328245%

```

We note that after training the test accuracy is 75.8%, which is certainly

- Write a script that plots the training curves (training/test loss and accuracy versus training epoch). Comment on whether these curves show any overfitting or underfitting of the data.

**Solution:** My script is given below



```

import matplotlib.pyplot as plt

def training_curves(history):
    train_loss = history[:,0]
    test_loss = history[:,1]
    train_acc = history[:,2]
    test_acc = history[:,3]
    nsample,_ = history.shape
    epochs = np.arange(1,nsample+1)

    figure, axis = plt.subplots(1,2,figsize=(10,2.5))
    axis[0].plot(epochs, train_loss, "b--", label="training loss")
    axis[0].plot(epochs, test_loss, "b", label="testing loss")
    axis[0].set_title("Training and Testing Losses")
    axis[0].set_xlabel("Epochs")
    axis[0].set_ylabel("Loss")
    axis[0].legend()

    axis[1].plot(epochs, train_acc, "b--",label="Training acc")
    axis[1].plot(epochs, test_acc, "b", label="Testing acc")
    axis[1].set_title("Training and Testing accuracy")
    axis[1].set_xlabel("Epochs")
    axis[1].set_ylabel("Accuracy")
    axis[1].legend()

training_curves(history)

```

The plots do now show any evidence of overtraining, which suggests that we could have used a larger more complex model to get a better fit. Another issue we see is the apparent difference between the training and testing loss. This difference is often associated with training sets that did not capture the full complexity of the data. This is reasonable in our case, since we had nearly 3 times as much data in the testing set than the training set.

