Q1 - Zhangsheng Lai (1002554)

October 7, 2018

Q1. Formulation as a classification problem and different feature vectors

```
In [1]: import cv2
        import os
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        sns.set()

        import keras
        from keras import callbacks
        from keras.datasets import mnist
        from keras.models import Sequential
        from keras.layers import *
        from keras.optimizers import RMSprop, SGD

        from matplotlib.colors import ListedColormap
        %matplotlib inline
```

```
Using TensorFlow backend.
```

Change to the directory containing the images

```
In [2]: path = "C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/train/airplane"
        os.chdir(path)
```

Q1. Formulation of the classification problem The classification problem here is to classify the images in the different folders into one of the classes from the set {airplane, automobile, bird, cat}.

The input image can be converted into a *feature vector* with one of the following examples listed below:

1. Using the raw pixel values For each `.jpg` image, we can extract the raw pixel values using `cv2.imread` which returns an array representing the raw RGB intensities of the image. Using `.flatten()` we convert the raw image from a multi-dimensional array into a single array of values with dimensions `(3072,)`, which can be used as the input for the classification problem.

Q2,3 - Zhangsheng Lai (1002554)

October 7, 2018

Q2. Logistic regression algorithm using stochastic gradient descent to perform binary classification

```
In [1]: import cv2
        import os
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        sns.set()

        from utils import *
        %matplotlib inline
```

```
In [2]: def load_data(path, feature = 'raw'):
            """
            Loads data into pixel values from the list of path given. Returns
            Input:
            - path (list): list of path to load the data from.
            - feature: either 'raw' or 'hist' for raw pixel values and 3D histogram respective
            """
            x_train=[]
            y_train=[]
            for c,i in enumerate(path):
                os.chdir(i)
                l = os.listdir()
                for i in l:
                    if feature == 'raw':
                        vf = convert2pixel_value(i)
                    else:
                        vf = convert2color_3Dhist(i)
                    x_train.append(vf)
                    y_train.append(c)

            x_train = np.concatenate([i[np.newaxis] for i in x_train])
            y_train = np.array(y_train)

            # comment below to remove the shuffling of the data
            arr = np.arange(x_train.shape[0])
```

```
        np.random.shuffle(arr)
        x_train = x_train[arr]
        y_train = y_train[arr]

        return x_train, y_train
```

Define the path to the directories containing the images then load the data set using `load_data`.

```
In [3]: train_bird = "C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/train/bir
        train_cat = "C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/train/cat"
        test_bird = "C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/test/bird"
        test_cat = "C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/test/cat"
```

Load the data from `bird` and `cat` folders, and we shall let `x1`, `y1` refer to the raw pixel value feature and x2, y2 refer to the feature obtained by using the 3D histogram in this jupyter notebook.

```
In [4]: x1_train, y1_train = load_data([train_cat, train_bird])
        x1_test, y1_test = load_data([test_cat, test_bird])

In [5]: x1_train.shape

Out[5]: (40, 3072)
```

We do some preprocessing of the training data by normalizing the pixel values to between $[0, 1]$ and the labels to be $\{-1, +1\}$.

```
In [6]: def add_bias(dataset):
            """
            Add a one to each sample for bias. Dataset must be of the
            form rows: samples, columns: features
            """
            n, m = dataset.shape
            out = np.ones((n, m+1))
            out[:,:-1] = dataset
            return out

In [7]: x1_train = x1_train/255
        x1_test = x1_test/255
        y1_train = y1_train*2 - 1
        y1_test = y1_test*2 - 1

In [8]: x1_train = add_bias(x1_train)
        x1_test = add_bias(x1_test)

In [9]: print (x1_train.shape[0], 'training samples')
        print (x1_test.shape[0], 'test samples')

40 training samples
40 test samples
```

Using features obtained from the raw pixels to do the logistic loss

```
In [10]: def sigmoid(x):
             """
             Applies the sigmoid function on the given vector.
             Input(s):
             - x : numpy vector of values
             """
             return 1/(1+np.exp(-x))
```

```
In [11]: def initialize_params(size=3073, seed=123):
             """
             Initialize parameters W weights and b biases.
             Input(s):
             - size (int): size of the parameters
             - seed (int): seed for the random number generator
             """
             rng = np.random.RandomState(seed)

             return rng.normal(size=(size,))
```

```
In [12]: def log_loss(x_train, y_train, W):
             """
             Computes the loss value of the logistic loss.
             Input(s):
             - x_train, y_train: training data and labels. x_train takes
             different forms depending on the features used and y_train
             is {-1,+1}.
             - W: value of the parameters.
             """
             z = y_train * np.dot(x_train, W)
             h = sigmoid(z)

             return -np.mean(np.log(h))
```

```
In [13]: def log_grad(x_train, y_train, W):
             """
             Computes the gradient of the logistic loss function.
             Input(s):
             - x_train, y_train: training data and labels. x_train takes
             different forms depending on the features used and y_train
             is {-1,+1}.
             - W: value of the parameters.
             """
             z = y_train * np.dot(x_train, W)
             h = sigmoid(z)
             n = x_train.shape[0]

             return 1/n * np.dot(x_train.T,(y_train * (h-1)))
```

```
In [14]: def next_batch(x_train, y_train, batch_size=2):
             """
             Returns a batch of size batch_size for stochastic gradient descent.
             - x_train, y_train: training data and labels. x_train takes different
             forms depending on the features used and y_train is {-1,+1}.
             - batch_size (int): size of each batch.
             """
             for i in np.arange(0, x_train.shape[0], batch_size):
                 yield (x_train[i:i+batch_size],y_train[i:i+batch_size])
```

```
In [15]: def log_classifier(x, learnt_W):
             """
             Takes in the test set and learnt parameters and returns the
             accuracy of the classifier on the test set.
             Inputs:
             - x: data for classification
             - learnt_W: learnt parameters
             """
             return (sigmoid(np.dot(x, learnt_W)) >= .5) * 2 - 1
```

```
In [16]: def log_accuracy(x, y, learnt_W):
             """
             Returns the accuracy of the model with parameters learnt_W.
             Input(s):
             - x: data for classification
             - y: corresponding labels to the data x
             - learnt_W: learnt parameters
             """
             output = log_classifier(x, learnt_W)
             return np.sum(np.absolute(y - output) == 0)/y.shape[0]
```

```
In [17]: def log_train(x_train, y_train, x_test, y_test, W, alpha=0.01, batch_size = 4, epoch
             """
             Trains the log loss model with given learning rate alpha, batch_size, epoch and i
             Returns the history of the loss, train accuracy, test accuracy and the value of t
             at different epochs.
             Input(s):
             - x_train, y_train: train data and labels
             - x_test, y_test: test data and labels
             - W: parameters of the model
             - alpha: learning rate
             - batch_size: batch size for stochastic gradient descent
             - epoch: number of times the dataset is passed through the model.
             """
             loss_history = []
             train_acc_history = []
             test_acc_history = []
             W_history = []
```

Q4 - Zhangsheng Lai (1002554)

October 7, 2018

Q4. kNN classifier for multi-class classification problem

```
In [1]: import cv2
        import os
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        sns.set()

        from utils import *
```

```
In [2]: folders = ['bird', 'cat','airplane','automobile']
        train_path_list = []
        test_path_list = []
        train_dir = 'C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/train/'
        test_dir = 'C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/test/'
        for folder in folders:
            l_train = train_dir + folder
            l_test = test_dir + folder
            train_path_list.append(l_train)
            test_path_list.append(l_test)
```

```
In [3]: x1_train, y1_train = load_data(train_path_list, feature='raw')
        x1_test, y1_test = load_data(test_path_list, feature='raw')

        x2_train, y2_train = load_data(train_path_list, feature=None)
        x2_test, y2_test = load_data(test_path_list, feature=None)
```

```
In [4]: x1_train = x1_train/255
        x1_test = x1_test/255
```

```
In [5]: print (x1_train.shape)
        print (x2_train.shape)
```

```
(80, 3072)
(80, 512)
```

```
In [6]: def euclid_dist(x1, x2):
            """
            Euclidean distance between two numpy arrays.
            """
            return np.linalg.norm(x1 - x2)
```

```
In [7]: def get_neighbors(x_train, y_train, x, k):
            """
            Returns the k nearest neighbors.
            Input(s):
            - x_train, y_train: the training samples and its labels
            - x: the data point whose neighbors we are interested in
            - k: number of neighbors to return
            """
            distances = []
            for i in range(x_train.shape[0]):
                dist = euclid_dist(x, x_train[i])
                distances.append((x_train[i], dist, y_train[i]))
            distances.sort(key=lambda x: x[1])
            neighbors = distances[:k]
            return neighbors
```

Function vote to decide which is the nearest neighbour

```
In [8]: from collections import Counter
        def vote(neighbors):
            class_counter = Counter()
            for neighbor in neighbors:
                class_counter[neighbor[2]] += 1
            return class_counter.most_common(1)[0][0]
```

```
In [9]: def knn_classfier(x_train, y_train, x_test, y_test, k):
            """
            Returns the k nearest neighbors.
            Input(s):
            - x_train, y_train: the training set and its labels
            - x_test, y_test: the testing set and its labels
            - k: number of neighbors to return
            """
            y_predict = np.zeros(y_test.shape)
            for i in np.arange(x_test.shape[0]):
                neighbors = get_neighbors(x_train, y_train, x_test[i], k)
                nearest = vote(neighbors)
                y_predict[i] = nearest

            return y_predict
```

```
In [10]: def knn_accuracy(y_test, y_predict):
             """
```

Q5 - Zhangsheng Lai (1002554)

October 7, 2018

Q5. Using one-vs-all classifier to do multi-class classification Another way to perform multi-class classification is by using the one-vs-all method. Given a training set with labels, we know the number of possible classes. Suppose there are n classes, thus for each class we have to modify the labels such that it only differentiates samples *from a particular class* or *not from the particular class*. Thus if there are n classes, we need to generate n new binary labels from the initial multi-class label.

With the new binary labels that is used to train the binary classification of a class from the other classes, we can learn the parameters for such binary classification. In my experiments here, the logistic loss was used to do the training of the 4 binary classification models. Thus we will obtain a total of n models. With the n learnt models we can then evaluate the probability by using the sigmiod function. The class of the binary classifier with the highest probability will then the predicted class.

```
In [1]: import cv2
        import os
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        sns.set()

        from utils import *

In [2]: folders = ['bird', 'cat','airplane','automobile']
        train_path_list = []
        test_path_list = []
        train_dir = 'C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/train/'
        test_dir = 'C:/Users/zlai/Documents/repo/HomeworkTex/ML/hw/homework 1/data/test/'
        for folder in folders:
            l_train = train_dir + folder
            l_test = test_dir + folder
            train_path_list.append(l_train)
            test_path_list.append(l_test)

In [3]: # loading raw pixel features
        x1_train, y1_train = load_data(train_path_list, feature='raw')
        x1_test, y1_test = load_data(test_path_list, feature='raw')

        # loading histogram features
        x2_train, y2_train = load_data(train_path_list, feature = None)
        x2_test, y2_test = load_data(test_path_list, feature = None)
```

```
In [4]: # normalise the raw pixel features, we do not normalize the histogram
        # features as from Q2,3 normalization gives lower accuracy
        x1_train = x1_train/255
        x1_test = x1_test/255
```

```
In [5]: # adding of biases
        x1_train = add_bias(x1_train)
        x1_test = add_bias(x1_test)
        x2_train = add_bias(x2_train)
        x2_test = add_bias(x2_test)
```

```
In [6]: print (x1_train.shape, 'raw pixel feature dimension')
        print (x2_train.shape, 'histogram feature dimension')
```

```
(80, 3073) raw pixel feature dimension
(80, 513) histogram feature dimension
```

To do one-vs-all classification, we need to relabel a particular class as 1 and the rest as -1.

```
In [7]: def relabel(y, c):
            """
            Relabel labels y from multiple classes to binary classes.
            Input(s):
            - y: labels to be relabeled
            - c: the class to be labeled as 1
            """
            new_label = np.ones(y.shape)
            ind = y != c
            new_label[ind] = -1
            return new_label
```

Depending on the number of classes we know from the labels, we have to create a new label for each one-vs-all classification. As we have 4 distinct classes in our image dataset, we need to generate 4 new binary labels, one for each one-vs-all classification for each class.

```
In [8]: def relabel_multiclass(y):
            """
            Uses relabel to relabel the labels of data with multiple classes
            to multiple binary labels. Returns a list of relabeled labels,
            with each item in the list a binary label (-1/+1) for each class.
            Input(s):
            - y: labels to be relabeled
            """
            y_list = []
            c = len(np.unique(y))
```

```
        for i in np.arange(c):
            y_temp = y.copy()
            y_temp = relabel(y_temp, c=i)
            y_list.append(y_temp)

        return y_list
```

```
In [9]: # relabeling of the labels. Note that we only need to do it for once as the labels do
        # features used to do the classification
        y1_train_list = relabel_multiclass(y1_train)
        y1_test_list = relabel_multiclass(y1_test)
```

The logistic loss algorithm is used here to train the parameter

```
In [10]: def onevsall_train(x_train, y_train, x_test, y_test, W, alpha=0.01, batch_size = 4, ep
             """
             Trains the parameters of each one-vs-all model. Returns a list
             of learnt_W_history, with each item of the list belonging to a
             certain model. Each item in the list contains the learnt_W_history
             that spans over the chosen number of epochs for a certain model.
             Input(s):
             - x_train: training images
             - y_train: labels for the training images
             - x_test: testing images
             - y_test: labels for the testing images
             - W: parameters of the model
             - alpha: learning rate
             - batch_size: size of each batch using stochastic gradient descent
             - epoch: number of times the whole dataset is used to train the model
             """

             learnt_W_history_list = []
             y_train_list = relabel_multiclass(y_train)
             y_test_list = relabel_multiclass(y_test)

             for i in np.arange(len(y_train_list)):
                 W_temp = W.copy()
                 loss_history, train_acc_history, test_acc_history, learnt_W_history = log_tra




                 learnt_W_history_list.append(learnt_W_history)

             return learnt_W_history_list
```

```
In [11]: def onevsall_predict(x, learnt_W_history_list):
             """
             Input(s):
```

```
        - x: data to be predicted
        - learnt_W_history_list: history of learnt
        parameters at different epoch
        """
        predict_epoch = []
        for i in range(len(learnt_W_history_list[0])): # loop over epoch

            prob_list = [] # stores list of prob for each model for a given epoch
            for j in range(len(learnt_W_history_list)): # loop over models
                p = sigmoid(np.dot(x,learnt_W_history_list[j][i])) # get probabilites for
                prob_list.append(p)

            prob = np.concatenate([i[np.newaxis] for i in prob_list])
            predict = np.argmax(prob, axis = 0) # predicts the class for epoch i
            predict_epoch.append(predict) # stores the prediction from the model at epoch
        return predict_epoch
```

```
In [12]: def onevsall_accuracy(y, predict_epoch):
        """
        Input(s):
        - y: true label of the data
        """
        acc_list = []
        for i in range(len(predict_epoch)):
            acc = np.average(predict_epoch[i] == y)
            acc_list.append(acc)
        return acc_list
```

```
In [13]: W = initialize_params(size=x1_train.shape[1], seed=123)
         learnt_W_history_list = onevsall_train(x1_train, y1_train, x1_test, y1_test, W, epoch

         theta = initialize_params(size=x2_train.shape[1],seed=123)
         learnt_theta_history_list = onevsall_train(x2_train, y1_train, x2_test, y1_test, theta
```

```
C:\Users\zlai\Documents\repo\HomeworkTeX\ML\hw\utils.py:126: RuntimeWarning: divide by zero enc
  return -np.mean(np.log(h))
```

```
In [14]: predict_epoch_train1 = onevsall_predict(x1_train, learnt_W_history_list)
         acc_list_train1 = onevsall_accuracy(y1_train, predict_epoch_train1)

         predict_epoch_train2 = onevsall_predict(x2_train, learnt_theta_history_list)
         acc_list_train2 = onevsall_accuracy(y1_train, predict_epoch_train2)

         predict_epoch_test1 = onevsall_predict(x1_test, learnt_W_history_list)
         acc_list_test1 = onevsall_accuracy(y1_test, predict_epoch_test1)

         predict_epoch_test2 = onevsall_predict(x2_test, learnt_theta_history_list)
         acc_list_test2 = onevsall_accuracy(y1_test, predict_epoch_test2)
```

Q6,7 - Zhangsheng Lai (1002554)

October 7, 2018

Q6. Alternative ways to represent input images as vectors In question 1 earlier, we see that the alternative ways to represent the input images a vectors include using (1) the mean and standard deviation of the color channel (2) color histogram, where the second option has a few subvariations, but we shall consider only the 3D histogram which is what was used in the earlier questions.

On the surface, it seems like using the pixels to represent images may seem like a good idea as there is little loss of the information, compared to using (1) which gives a point estimate and variance of the image, which is not descriptive enough of the image as much of the information is lost. However, if we use the pixel values from the input images, each image is represented by a `(3072,)` vector; which is a high dimensional input which our simple model might not be able to handle.

By considering using the 3D histogram features, we reducing the dimensions of the features that is fed to our model but at the same time, we reduce the loss of information. The 3D histogram describes the joint distribution of the red, blue and green values for different intervals, depending on the number of bins choosen to generate the historgram. In my implementation, 8 bins were chosen, thus each bin is of interval size 32 and reduces the dimensions of the features to 512, much managable than the pixel values. The information loss is also reduced as it sorts the pixels into the different bins, which gives us the distrbution of the pixels with respect to the red, blue, green levels.

Thus we observe the following when we use the 3D histogram feature when compared against the pixel features:

- Q2 (logistic loss): Test accuracy improved from 50% to 57.5%
- Q3 (hinge loss): Test accuracy improved from 55% to 67.5%
- Q4 (k nearest neighbors): Test accuracy dropped from 47.5% to 42.5%

Q7. Training using augmented dataset Using Google image search, 20 images from each class was added to the original training set, increasing the size of training set to be twice larger than the initial. As the new images added to the training set is not of the same dimensions as the original images, we cannot use the pixel values as the features. The features from the 3D histogram will be used as the features. As we are doing a multi-class classification, the one-vs-all approach like in question 5 will be used here.

```
In [1]: import cv2
        import os
        import numpy as np
```