CSC411: Project 2

Due on Sunday, February 18, 2018

Ying Yang

March 18, 2018

Part 1

Dataset description

We collected the top 10 word occurences from each of real and fake news data files. The result is show below:
Real:
[('trump', 1744), ('donald', 829), ('to', 413), ('us', 230), ('trumps', 219), ('in', 214), ('on', 205), ('of', 186), ('says', 178), ('for', 174), ('the', 173), ('and', 116), ('with', 104), ('a', 93), ('election', 87)]
Fake:
[('trump', 1328), ('the', 439), ('to', 409), ('in', 231), ('donald', 228), ('of', 212), ('for', 205), ('a', 192), ('and', 180), ('on', 166), ('is', 157), ('hillary', 150), ('clinton', 132), ('with', 100), ('will', 96)]
Example of 3 useful keywords are trump, donald, and hillary

Part 2

Problem: Implement the Naive Bayes algorithm for predicting whether a headline is real or fake.

We used validation set to tune the parameters.
We used the following formula to get the probability of a headline to be real or fake.
P(real | headline) = P(headline_word_1 | real) * P(headline_word_2 | real) * ... * P(headline_word_n | real) * P(real)

Listing 1: Compute network function

```
import numpy as np

def compute_network(x, W, b):
"""
Compute the network to get output o using sum of linear combination
of input x and weight W and b
"""
o = np.dot(W.T, x) + b
return softmax(o)

if __name__ == "__main__":
M = loadmat("mnist_all.mat")
np.random.seed(10)
x = (M["train0"][100].reshape((28 * 28, 1))) / 255.0
W = np.random.rand(28 * 28, 10)
b = np.random.rand(10, 1)
compute_network(x, W, b)
```

Part 3

Part 3a

Compute the gradient of the cost function with respect to the weight w_{ij}

Defining terms:
Cost function for a single training case: C = - $\sum_j y_j log p_j$
$p_i = \frac{exp(o_i)}{\sum_j exp(o_j)}$
Compute the gradient descent: $\frac{\partial C}{\partial o_i} = \sum_j \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i} = p_i - y_i$

Part 3b

Write vectorized code that computes the gradient of the cost function with respect to the weights and biases of the network

```
    dCdW = np.dot(x, (p.T-t.T))
    dCdb = p-t
    delta = 1e-10
    N = W.shape[0]
    K = W.shape[1]
    for times in range(0,6):
        i = np.random.randint(N)
        j = np.random.randint(K)
        print times, ("gradient of dCdW:"), dCdW[i][j], "\n"
        theta = np.copy(W)
        theta[i][j] = theta[i][j] + delta
        fd = (cost_function(x, theta, b, t)-cost_function(x, W, b, t))/delta
        print times, ("finite difference of dCdW:"), fd, "\n"
        print times, ("gradient of dCdb:"), dCdb[j][0], "\n"
        theta = np.copy(b)
        theta[j][0] = theta[j][0] + delta
        fd = (cost_function(x, W, theta,t)-cost_function(x, W, b, t))/delta
        print times,("finite difference of dCdb:"), fd, "\n"
```

Output: 0 gradient of dCdW: 0.0
0 finite difference of dCdW: 0.0
0 gradient of dCdb: 0.0188239452014
0 finite difference of dCdb: 0.0188293824976
1 gradient of dCdW: 0.0
1 finite difference of dCdW: 0.0
1 gradient of dCdb: 0.0458990966574
1 finite difference of dCdb: 0.0459010607301
2 gradient of dCdW: 0.0
2 finite difference of dCdW: 0.0
2 gradient of dCdb: 0.00211207762373
2 finite difference of dCdb: 0.00211386463889
3 gradient of dCdW: 0.0291950090963
3 finite difference of dCdW: 0.0291944246555

3 gradient of dCdb: 0.0298985032914
3 finite difference of dCdb: 0.0299049673913
4 gradient of dCdW: 0.0
4 finite difference of dCdW: 0.0
4 gradient of dCdb: 0.0610826151789
4 finite difference of dCdb: 0.0610800299228
5 gradient of dCdW: 0.0
5 finite difference of dCdW: 0.0
5 gradient of dCdb: 0.0188239452014
5 finite difference of dCdb: 0.0188293824976
Results of finite difference and gradient calculation are relatively equal, only some rounding errors.

Part 4

Train the neural network using gradient descent (without momentum). Plot the learning curves. Display the weights going into each of the output units.

Optimization procedure:
We initialized the parameters as follows:

Listing 2: Parameter Initialization

```
# load weights and biases
snapshot = cPickle.load(open("snapshot50.pkl"))
W0 = snapshot["W0"]  # dimension = (748, 300)
b0 = snapshot["b0"].reshape((300, 1))
W1 = snapshot["W1"]  # dimension = (300, 10)
b1 = snapshot["b1"].reshape((10, 1))
v_w = 0  # for momentum
v_b = 0

# initiate weights, bias and alpha
w0b0 = dot(W0, b0)  # dimension = (748, 1)
w1b1 = dot(W1, b1)  # dimension = (300, 1)
initial_weight = dot(W0, W1)  # (748, 10)
initial_bias = (dot(W1.T, b0)) + b1  # (10, 1)
print "initial bias shape", ((dot(W1.T, b0)) + b1).shape[0], ((dot(W1.T, b0)) + b1).shape[1
alpha = 1e-5  # learning rate

# get train and test performance based on increased epochs
epoch, train_performance, test_performance = [], [], []
EPS = 1e-10
prev_w = initial_weight - 10 * EPS
prev_b = initial_bias - 10 * EPS
W = initial_weight.copy()
b = initial_bias.copy()
iteration = 0
max_iteration = 2000
```

We initialized the weight to be the dot product of W0 and W1. We repeatedly experimented with different alpha values and observe the results. We first set alpha=0.001. The percentage of correctness turned out to be very low (with 25% of correcteness). Then we decreased the learning rate to 0.00001. The correctness percentage then improved to 89%.

Gradient descent procedure:

Listing 3: Parameter Initialization

```
# apply gradient descent
while norm(W - prev_w) > EPS and norm(b - prev_b) > EPS and iteration < max_iteration:
    prev_w = W.copy()
    prev_b = b.copy()
    W -= alpha * df_w(train_set.T, W, b, train_label.T)
    b -= alpha * df_b(train_set.T, W, b, train_label.T)
    if iteration % 100 == 0:
        epoch.append(iteration)
        train_performance.append(performance(train_set.T, W, b, train_label.T))
```

```
10                      test_performance.append(performance(test_set.T, W, b, test_label.T))
                iteration += 1
```

Display the weights:

Learning curve is shown in the figure below:

Part 5

Write vectorized code that performs gradient descent with momentum, and use it to train your network. Plot the learning curves

Learning curve with momentum:l

How new learning curves compare with gradient descent without momentum: Without the momentum, the performance of correct guesses increased slower than if momentum were used. For example, as we can observe in the learning curve, the performance of training set only increased to 92% after 800 iterations, wheareas with momentum term set to 0.99, the performance correcnesses increased to the same percentage by only using 400 iterations, because it moved faster when gradient consistenly points to one direction.

Code added in the gradient descent:

In the train_neural_network method, method signature was modified to take two extra terms: type and momemtum term. The rest of code remains unchanged as in part 4.

Listing 4: Gradient descent with momentum

```
train_neural_network(image_prefix, type="None", momentum_term=0):

v_w = 0 # for momentum
v_b = 0
if type == "momentum":
    v_w = momentum_term * v_w + alpha * df_w(train_set.T, W, b, train_label.T)
    W -= v_w
    v_b = momentum_term * v_b + alpha * df_b(train_set.T, W, b, train_label.T)
    b -= v_b

if __name__ == "__main__":
     momentum_term = 0.99
     train_neural_network("part5", "momentum", momentum_term)
```

Part 6

Part 6d

Momentum trajectory walks faster than vanilla gradient descent, and momentum can lead to points past the central local minimum while vanilla gradient descent walks directly to the local point. The cause of these differences is the extra term in momentum which is momentum_term * v_w speeds up the walking pace, and it retains effects of previous v_m term.

Part 6e

I found proper settings by checking the gradient descent values and trial-and-error. Whichever weights' values are the most among all the gradient descent values, should be chosen as the two variables. It speeds up the descent process. Another setting is momentum term. Choose it by trial-and-error to see which one fits the graph the best. Initial weights values are also important because the directions of original gradient descents can demonstrate the difference between two ways, too.

Other settings do not demonstrate the benefits of momentum are alpha and axis range. Choose plot axis which can zoom in the contour at the proper size and alpha which is neither too large nor too small. Momentum term do not really affect visualization unless it is so large that points past local minimum too much or offtrack too much.

Part 7

Assumption: For each neuron, due to unknown size, suppose matrix multiplication complexity is 1.

For i-th layer, each j in K neurons, backpropagation computes w_j with complexity of O(K) for all i.

If it computes individually, w_j has computation complexity of O(K^{2i-3})

Final computation complexity, for backpropagation, is O(N K^2), individually, is O(K^{2N-2})
When N is larger than 2, backpropagation is O(K^{2N-4}) faster than individual computation on condition that N is known.

Part 8

Use a fully-connected neural network with a single hidden layer, but train using mini-batches using an optimizer
Description of the system:

We initialize inputs and weights as follows:
dim_x = 64 * 64 * 3
dim_h = 500
dim_out = 6
We used one hidden layer of size 500.
The resolution was 64 * 64.
We used Adam as optimizer
The model we used was:

Listing 5: model of the system

```
        x = Variable(torch.from_numpy(train_set[train_index]),
                                          requires_grad=False).type(dtype_float)
        y_classes = Variable(torch.from_numpy(np.argmax(train_label[train_index], 1)),
                                 requires_grad=False).type(dtype_long)

        model = torch.nn.Sequential(
        torch.nn.Linear(dim_x, dim_h),
        torch.nn.ReLU(),
        torch.nn.Linear(dim_h, dim_out),
        )
```

Optimization steps and outputs:
We tried varying parameters, optimizers and functions and other factors that might improve the performance, such as initial weights, learning rate, batch size, number of iterations, optimization function to optimize the training performance, activation function, number of hidden layers, weight of hidden layers.

Listing 6: initial inputs and weights and parameters and functions chosen for optimization

```
            dim_x = 64 * 64 * 3
            dim_h = 500
            dim_out = 6

            learning_rate = 1e-4
            max_iteration = 500
            optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
            loss_fn = torch.nn.CrossEntropyLoss()
```

Observations: Activation function: Since we are using mini-batches, we found out that ReLU was the best option. It accelerates the convergencce of stochastic gradient descent compared to sigmoid and tanh functions.
Learning rate: We first tried using learning rate of 0.001 then increased to 0.00001. We found out that the performance was best at learning rate = 0.0001. The performance improved from 0.001 to 0.0001 but decreased from 0.0001 to 0.00001.
Number of layers: We experimented with a single and 2 hidden layers, with fixed weight. Keeping other parameters unchanged, we found unsignificant difference between having 2 or 1 hidden layers.

Weights: We experimented with weights from 20 to 700. At low weight, the test performance had poor results. The accuracy performance increased slower compared to having setting weight as 700. Also, the test performance was unstable when the weight was low. It might first increase and decrease later.

Learning curve:
Final performance: The performance of the test set was at 85.83%.
The results were:
test performance: 36.6666666667
test performance: 83.3333333333
test performance: 85.0
test performance: 85.0
test performance: 86.6666666667
test performance: 85.8333333333
test performance: 85.8333333333
test performance: 85.8333333333
test performance: 85.8333333333
test performance: 85.8333333333
test performance: 85.8333333333

Part 9

Select two of the actors, and visualize the weights of the hidden units that are useful for classifying input photos as those particular actors. Explain how you selected the hidden units
Actors selected: balwin and bracco
Approach: Try lots of images in a dataset, and see which ones active the neuron the most.

Part 10

We modified code for AlexNet. We want to extract activation values before it classifies the actor, so we deleted the line x = x.classifier(x) in forward() method.
The modified code is:

Listing 7: modified code for AlexNet

```
class MyAlexNet(nn.Module):
    def load_weights(self):
        an_builtin = torchvision.models.alexnet(pretrained=True)

    features_weight_i = [0, 3, 6, 8, 10]
    for i in features_weight_i:
        self.features[i].weight = an_builtin.features[i].weight
        self.features[i].bias = an_builtin.features[i].bias

    classifier_weight_i = [1, 4, 6]
    for i in classifier_weight_i:
        self.classifier[i].weight = an_builtin.classifier[i].weight
        self.classifier[i].bias = an_builtin.classifier[i].bias

    def __init__(self, num_classes=1000):
        super(MyAlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )

    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(256 * 6 * 6, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, num_classes),
    )

    self.load_weights()

    # we modified this section
    def forward(self, x):
```

```
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
```

Extract the values of the activations of AlexNet on the face images in a particular layer
Input images size: 227*227
We will extract activations of the network during the forward pass.
Then, we use part 8 model to train a fully-connected neural network by taking the activations as input.
Get the activation values as input to part8 code as follows:

Listing 8: How to get activation values

```
# get train and test activations using forward
x_train = Variable(torch.from_numpy(train_set[:]), requires_grad=False).type(torch.FloatTen
train_activation = np.vstack((train_activation, model.forward(x_train).data.numpy()))
x_test = Variable(torch.from_numpy(train_set[:]), requires_grad=False).type(torch.FloatTens
test_activation = np.vstack((test_activation, model.forward(x_test).data.numpy()))

train_using_mini_batches(act, train_activation, test_activation, "part10", 227)
```

Performance:
The performance dramatically improved compared to part8.
Using myAlexNet model, we were able to obtain a final performance of 95.83
The following is the performance:
('epoch: ', 0, 'train performance:', 55.21172638436482, 'test performance: ', 45.83333333333333)
('epoch: ', 50, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 100, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 150, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 200, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 250, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 300, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 350, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 400, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 450, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
('epoch: ', 499, 'train performance:', 100.0, 'test performance: ', 95.83333333333334)
The performance improved by more than 30% compared to part 8.
Run the part10 code using: python myalexnet.py