Name: Arpit Aggarwal
UID: 116747189

1. Packages

In [1]:
```
# header files
import numpy as np
import torch
import h5py
from matplotlib import pyplot as plt
import re
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Cat vs Non-Cat Image Classification

2. Loading Dataset

Using hw2.ipynb load_data() function. The load_data() function loads data from the training and testing files. Next step, is to flatten the image so that they can be fed as an input to the neural network. Lastly, the training and testing data is normalized between 0 and 1 which will be used for the neural network.

In [2]:
```
def load_data(train_file, test_file):
    # Load the training data
    train_dataset = h5py.File(train_file, 'r')

    # Separate features(x) and labels(y) for training set
    train_set_x_orig = np.array(train_dataset['train_set_x'])
    train_set_y_orig = np.array(train_dataset['train_set_y'])

    # Load the test data
    test_dataset = h5py.File(test_file, 'r')

    # Separate features(x) and labels(y) for training set
    test_set_x_orig = np.array(test_dataset['test_set_x'])
    test_set_y_orig = np.array(test_dataset['test_set_y'])
    classes = np.array(test_dataset["list_classes"][:]) # the list of 
classes

    train_set_y_orig = train_set_y_orig.reshape((train_set_y_orig.sha
pe[0]))
    test_set_y_orig = test_set_y_orig.reshape((test_set_y_orig.shape[
0]))
    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_
set_y_orig, classes

# training and testing files
train_file = "data/train_catvnoncat.h5"
test_file = "data/test_catvnoncat.h5"
train_x_orig, train_output, test_x_orig, test_output, classes = load_
data(train_file, test_file)
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1)
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1)

# Standardize data to have feature values between 0 and 1.
train_input = train_x_flatten / 255.
test_input = test_x_flatten / 255.

# print data length
print ("train_input's shape: " + str(train_input.shape))
print ("test_input's shape: " + str(test_input.shape))
```

```
train_input's shape: (209, 12288)
test_input's shape: (50, 12288)
```

3. Convert dataset to Tensor form

Convert the dataset to Tensor form so that it can be fed into the PyTorch neural network.

In [3]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# use torch.from_numpy() to get the tensor form of the numpy array
train_input = torch.from_numpy(train_input).float().to(device)
train_output = torch.from_numpy(train_output).float().to(device)
test_input = torch.from_numpy(test_input).float().to(device)
test_output = torch.from_numpy(test_output).float().to(device)
```

4. Hyper-parameters

Set the hyper-parameters of the two-layer neural net.

In [4]:
```
learning_rate = 0.001
num_epochs = 3000
```

5. Model-Architecture

The model-architecture is defined using pytorch Net class. The **init** function is where we define the architecture of the neural network, i.e in this it is two layers. The forward function is where the forward pass step of the neural network takes place.

In [5]:
```
# neural network class
class Net(torch.nn.Module):
    # init function
    def __init__(self, num_input_neurons, num_hidden_neurons, num_out
put_neurons):
        super(Net, self).__init__()
        self.fc1 = torch.nn.Linear(num_input_neurons, num_hidden_neur
ons)
        self.fc2 = torch.nn.Linear(num_hidden_neurons, num_output_neu
rons)

    # forward pass step of the neural network
    def forward(self, input):
        output = torch.nn.functional.sigmoid(self.fc2(torch.nn.functi
onal.relu(self.fc1(input))))
        return output

# get the neural net object
net = Net(int(train_input.shape[1]), 7, 1).to(device)
print(net)
```

```
Net(
  (fc1): Linear(in_features=12288, out_features=7, bias=True)
  (fc2): Linear(in_features=7, out_features=1, bias=True)
)
```

7. Loss function

We will use Binary Cross-entropy loss as we are doing image classification (cat vs non-cat)

In [6]:
```
# loss function
criterion = torch.nn.BCELoss()
```

8. Gradient Descent

Next step is to define the optimizer we will be using for training the neural net. We will use gradient descent (full-batch) as out optimizer.

In [7]:
```
# optimizer
optimizer = torch.optim.SGD(net.parameters(), lr = learning_rate, mom
entum=0.9)
```

9. Training phase

Now we will be training the neural network to get the optimal set of weights and biases required for this problem.

In [8]:
```
# training phase
for epoch in range(0, num_epochs):

    # forward step
    pred_output = net(train_input)

    # find loss
    loss = criterion(pred_output.squeeze(), train_output)

    # backpropagation step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if((epoch + 1)%100 == 0):
        print('Loss after iteration {}: {:.4f}' .format(epoch + 1, lo
ss.item()))
```

```
/home/arpitdec5/.local/lib/python2.7/site-packages/torch/nn/functiona
l.py:1351: UserWarning: nn.functional.sigmoid is deprecated. Use torc
h.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmo
id instead.")
```

```
Loss after iteration 100: 0.5677
Loss after iteration 200: 0.4727
Loss after iteration 300: 0.3714
Loss after iteration 400: 0.2814
Loss after iteration 500: 0.2087
Loss after iteration 600: 0.1550
Loss after iteration 700: 0.1168
Loss after iteration 800: 0.0900
Loss after iteration 900: 0.0711
Loss after iteration 1000: 0.0574
Loss after iteration 1100: 0.0474
Loss after iteration 1200: 0.0398
Loss after iteration 1300: 0.0340
Loss after iteration 1400: 0.0294
Loss after iteration 1500: 0.0257
Loss after iteration 1600: 0.0228
Loss after iteration 1700: 0.0203
Loss after iteration 1800: 0.0183
Loss after iteration 1900: 0.0166
Loss after iteration 2000: 0.0151
Loss after iteration 2100: 0.0139
Loss after iteration 2200: 0.0128
Loss after iteration 2300: 0.0118
Loss after iteration 2400: 0.0110
Loss after iteration 2500: 0.0102
Loss after iteration 2600: 0.0096
Loss after iteration 2700: 0.0090
Loss after iteration 2800: 0.0085
Loss after iteration 2900: 0.0080
Loss after iteration 3000: 0.0076
```

10. Testing Phase

Evaluating model on testing data

In [11]:
```
# testing phase
net.eval()
pred_output = net(test_input)
loss = criterion(pred_output.squeeze(), test_output)
#print("Testing Loss: " + str(loss.item()))

# accuracy
correct = 0
for index in range(0, len(pred_output)):
    if(pred_output[index] > 0.5):
        pred_output[index] = 1
    else:
        pred_output[index] = 0

    if(pred_output[index] == test_output[index]):
        correct = correct + 1
print("Testing accuracy is: " + str(100.0 * float(float(correct) / le
n(pred_output))) + "%")
```

```
Testing accuracy is: 76.0%
```

11. Results

This section contains all the hyper-parameters I tried and the corresponding accuracies.

1. learning_rate = 0.001, num_epochs = 3000, momentum = 0.9, Testing Accuracy = 76%
2. learning_rate = 0.001, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 72%
3. learning_rate = 0.001, num_epochs = 5000, Testing Accuracy = 72%
4. learning_rate = 0.005, num_epochs = 3000, momentum = 0.9, Testing Accuracy = 72%
5. learning_rate = 0.005, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 60%
6. learning_rate = 0.005, num_epochs = 5000, Testing Accuracy = 60%
7. learning_rate = 0.01, num_epochs = 3000, momentum = 0.9, Testing Accuracy = 58%
8. learning_rate = 0.01, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 70%
9. learning_rate = 0.01, num_epochs = 5000, Testing Accuracy = 64%
10. learning_rate = 0.05, num_epochs = 3000, momentum = 0.9, Testing Accuracy = 34%

12. Best Hyper-parameters obtained

The best hyper-parameters obtained were as follows:
learning_rate = 0.001, num_epochs = 3000, Testing Accuracy = 76%

Predicting sentiment of movie reviews

13. Loading data

Using the load_data function of hw2.ipynb and then preprocessing the data as done in the hw2.ipynb notebook

In [12]:
```
def load_data(train_file, test_file):
    train_dataset = []
    test_dataset = []

    # Read the training dataset file line by line
    for line in open(train_file, 'r'):
        train_dataset.append(line.strip())

    for line in open(test_file, 'r'):
        test_dataset.append(line.strip())
    return train_dataset, test_dataset

def preprocess_reviews(reviews):
    reviews = [REPLACE_NO_SPACE.sub(NO_SPACE, line.lower()) for line
in reviews]
    reviews = [REPLACE_WITH_SPACE.sub(SPACE, line) for line in review
s]
    return reviews

# loading data
train_file = "data/train_imdb.txt"
test_file = "data/test_imdb.txt"
train_dataset, test_dataset = load_data(train_file, test_file)
y = [1 if i < len(train_dataset)*0.5 else 0 for i in range(len(train_
dataset))]

# pre-processing
REPLACE_NO_SPACE = re.compile("(\.)|(\;)|(\:)|(\!)|(\')|(\?)|(\,)|(\"
)|(\()|(\))|(\[)|(\])|(\d+)")
REPLACE_WITH_SPACE = re.compile("(<br\s*/><br\s*/>)|(\-)|(\/)")
NO_SPACE = ""
SPACE = " "
train_dataset_clean = preprocess_reviews(train_dataset)
test_dataset_clean = preprocess_reviews(test_dataset)

# Vectorization
cv = CountVectorizer(binary=True, stop_words="english", max_features=
2000)
cv.fit(train_dataset_clean)
X = cv.transform(train_dataset_clean)
X_test = cv.transform(test_dataset_clean)
X = np.array(X.todense()).astype(float)
X_test = np.array(X_test.todense()).astype(float)
y = np.array(y)
```

14. Splitting of dataset

Using sklearn for splitting dataset into training and testing

In [13]:
```
X_train, X_val, y_train, y_val = train_test_split(X, y, train_size =
0.80)
y_train = y_train.reshape(1,-1)
y_val = y_val.reshape(1,-1)
```

```
/home/arpitdec5/.local/lib/python2.7/site-packages/sklearn/model_sele
ction/_split.py:2178: FutureWarning: From version 0.21, test_size wil
l always complement train_size unless both are specified.
  FutureWarning)
```

15. Convert dataset to Tensor form

Convert the dataset to Tensor form so that it can be fed into the PyTorch neural network.

In [14]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# use torch.from_numpy() to get the tensor form of the numpy array
train_input = torch.from_numpy(X_train).float().to(device)
train_output = torch.from_numpy(y_train).float().to(device)
train_output = train_output.squeeze()
test_input = torch.from_numpy(X_val).float().to(device)
test_output = torch.from_numpy(y_val).float().to(device)
test_output = test_output.squeeze()
```

16. Hyper-parameters

Set the hyper-parameters for the network

In [17]:
```
learning_rate = 0.05
num_epochs = 5000
```

17. Model Architecture

The model-architecture is defined using pytorch Net class. The **init** function is where we define the architecture of the neural network, i.e in this it is two layers. The forward function is where the forward pass step of the neural network takes place.

In [18]:
```
# neural network class
class Net(torch.nn.Module):
    # init function
    def __init__(self, num_input_neurons, num_hidden_neurons, num_out
put_neurons):
        super(Net, self).__init__()
        self.fc1 = torch.nn.Linear(num_input_neurons, num_hidden_neur
ons)
        self.fc2 = torch.nn.Linear(num_hidden_neurons, num_output_neu
rons)

    # forward pass step of the neural network
    def forward(self, input):
        output = torch.nn.functional.sigmoid(self.fc2(torch.nn.functi
onal.relu(self.fc1(input))))
        return output

# get the neural net object
net = Net(int(train_input.shape[1]), 200, 1).to(device)
print(net)
```

```
Net(
  (fc1): Linear(in_features=2000, out_features=200, bias=True)
  (fc2): Linear(in_features=200, out_features=1, bias=True)
)
```

18. Loss function

We will use Binary Cross-entropy loss as we are doing sentiment analysis

In [19]:
```
# loss function
criterion = torch.nn.BCELoss()
```

19. Gradient Descent

Next step is to define the optimizer we will be using for training the neural net. We will use gradient descent (full-batch) as out optimizer.

In [20]:
```
# optimizer
optimizer = torch.optim.SGD(net.parameters(), lr = learning_rate)
```

20. Training phase

Now we will be training the neural network to get the optimal set of weights and biases required for this problem.

In [21]:
```
# training phase
for epoch in range(0, num_epochs):

    # forward step
    pred_output = net(train_input)

    # find loss
    loss = criterion(pred_output.squeeze(), train_output)

    # backpropagation step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if((epoch + 1)%100 == 0):
        print('Loss after iteration {}: {:.4f}' .format(epoch + 1, lo
ss.item()))
```

```
Loss after iteration 100: 0.6343
Loss after iteration 200: 0.4483
Loss after iteration 300: 0.2684
Loss after iteration 400: 0.1721
Loss after iteration 500: 0.1178
Loss after iteration 600: 0.0847
Loss after iteration 700: 0.0632
Loss after iteration 800: 0.0488
Loss after iteration 900: 0.0389
Loss after iteration 1000: 0.0317
Loss after iteration 1100: 0.0265
Loss after iteration 1200: 0.0225
Loss after iteration 1300: 0.0195
Loss after iteration 1400: 0.0170
Loss after iteration 1500: 0.0150
Loss after iteration 1600: 0.0134
Loss after iteration 1700: 0.0121
Loss after iteration 1800: 0.0110
Loss after iteration 1900: 0.0100
Loss after iteration 2000: 0.0092
Loss after iteration 2100: 0.0085
Loss after iteration 2200: 0.0079
Loss after iteration 2300: 0.0073
Loss after iteration 2400: 0.0068
Loss after iteration 2500: 0.0064
Loss after iteration 2600: 0.0060
Loss after iteration 2700: 0.0057
Loss after iteration 2800: 0.0054
Loss after iteration 2900: 0.0051
Loss after iteration 3000: 0.0048
Loss after iteration 3100: 0.0046
Loss after iteration 3200: 0.0044
Loss after iteration 3300: 0.0042
Loss after iteration 3400: 0.0040
Loss after iteration 3500: 0.0038
Loss after iteration 3600: 0.0037
Loss after iteration 3700: 0.0035
Loss after iteration 3800: 0.0034
Loss after iteration 3900: 0.0033
Loss after iteration 4000: 0.0031
Loss after iteration 4100: 0.0030
Loss after iteration 4200: 0.0029
Loss after iteration 4300: 0.0028
Loss after iteration 4400: 0.0027
Loss after iteration 4500: 0.0027
Loss after iteration 4600: 0.0026
Loss after iteration 4700: 0.0025
Loss after iteration 4800: 0.0024
Loss after iteration 4900: 0.0024
Loss after iteration 5000: 0.0023
```

21. Testing phase

Evaluating model on testing data

In [22]:
```
# testing phase
net.eval()
pred_output = net(test_input)
loss = criterion(pred_output.squeeze(), test_output)

# accuracy
correct = 0
for index in range(0, len(pred_output)):
    if(pred_output[index] > 0.5):
        pred_output[index] = 1
    else:
        pred_output[index] = 0

    if(pred_output[index] == test_output[index]):
        correct = correct + 1
print("Testing accuracy is: " + str(100.0 * float(float(correct) / len(pred_output))) + "%")
```

```
Testing accuracy is: 83.5820895522%
```

22. Results

This section contains all the hyper-parameters I tried and the corresponding accuracies.

1. learning_rate = 0.05, num_epochs = 3000, Testing Accuracy = 83.58%
2. learning_rate = 0.05, num_epochs = 5000, Testing Accuracy = 85.07%
3. learning_rate = 0.05, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 83.58%
4. learning_rate = 0.01, num_epochs = 3000, Testing Accuracy = 82.58%
5. learning_rate = 0.01, num_epochs = 5000, Testing Accuracy = 83.58%
6. learning_rate = 0.01, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 83.58%
7. learning_rate = 0.005, num_epochs = 3000, Testing Accuracy = 81.59%
8. learning_rate = 0.005, num_epochs = 5000, Testing Accuracy = 82.08%
9. learning_rate = 0.005, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 83.58%
10. learning_rate = 0.001, num_epochs = 3000, Testing Accuracy = 67.16%
11. learning_rate = 0.001, num_epochs = 5000, Testing Accuracy = 73.13%
12. learning_rate = 0.001, num_epochs = 5000, momentum = 0.9, Testing Accuracy = 81.59%

23. Best Hyper-parameters obtained

The best hyper-parameters obtained were as follows:
learning_rate = 0.05, num_epochs = 5000, Testing Accuracy = 85.07%

In [ ]: