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 = 4000
weight_decay = 0.001
momentum = 0.9
```

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

# create object of the model
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]:
```
# optimizers
optimizer = torch.optim.SGD(net.parameters(), lr = learning_rate, mom
entum = momentum, weight_decay = weight_decay)
```

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: 0.6271
Loss after iteration 200: 0.5711
Loss after iteration 300: 0.4925
Loss after iteration 400: 0.4050
Loss after iteration 500: 0.3196
Loss after iteration 600: 0.2465
Loss after iteration 700: 0.1891
Loss after iteration 800: 0.1457
Loss after iteration 900: 0.1137
Loss after iteration 1000: 0.0903
Loss after iteration 1100: 0.0731
Loss after iteration 1200: 0.0604
Loss after iteration 1300: 0.0507
Loss after iteration 1400: 0.0432
Loss after iteration 1500: 0.0373
Loss after iteration 1600: 0.0326
Loss after iteration 1700: 0.0288
Loss after iteration 1800: 0.0256
Loss after iteration 1900: 0.0230
Loss after iteration 2000: 0.0208
Loss after iteration 2100: 0.0189
Loss after iteration 2200: 0.0173
Loss after iteration 2300: 0.0160
Loss after iteration 2400: 0.0148
Loss after iteration 2500: 0.0137
Loss after iteration 2600: 0.0128
Loss after iteration 2700: 0.0120
Loss after iteration 2800: 0.0113
Loss after iteration 2900: 0.0106
Loss after iteration 3000: 0.0100
Loss after iteration 3100: 0.0095
Loss after iteration 3200: 0.0090
Loss after iteration 3300: 0.0086
Loss after iteration 3400: 0.0082
Loss after iteration 3500: 0.0078
Loss after iteration 3600: 0.0075
Loss after iteration 3700: 0.0072
Loss after iteration 3800: 0.0069
Loss after iteration 3900: 0.0067
Loss after iteration 4000: 0.0064
```

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: 74.0%
```

11. Results

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

1. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 72%
2. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 66%
3. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 68%
4. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 64%
5. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 34%
6. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 74%
7. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 66%
8. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 70%
9. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 72%
10. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 64%
11. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 70%
12. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 64%
13. learning_rate = 0.05, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 34%
14. learning_rate = 0.01, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 58%
15. learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 74%
16. learning_rate = 0.005, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 72%
17. learning_rate = 0.005, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 72%
18. learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 74%
19. learning_rate = 0.01, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 34%
20. learning_rate = 0.05, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 34%

12. Optimal Hyper-parameters obtained

The optimal hyper-parameters obtained were as follows:

learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 74%
learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 74%
learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 74%

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 [15]:
```
learning_rate = 0.005
num_epochs = 4000
momentum = 0.9
weight_decay = 0.01
```

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 [16]:
```
# 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 [17]:
```
# 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 [18]:
```
# optimizer
optimizer = torch.optim.SGD(net.parameters(), lr = learning_rate, mom
entum = momentum, weight_decay = weight_decay)
```

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 [19]:
```
# 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: 0.6592
Loss after iteration 200: 0.5364
Loss after iteration 300: 0.3436
Loss after iteration 400: 0.2252
Loss after iteration 500: 0.1608
Loss after iteration 600: 0.1229
Loss after iteration 700: 0.0991
Loss after iteration 800: 0.0834
Loss after iteration 900: 0.0727
Loss after iteration 1000: 0.0652
Loss after iteration 1100: 0.0598
Loss after iteration 1200: 0.0558
Loss after iteration 1300: 0.0529
Loss after iteration 1400: 0.0507
Loss after iteration 1500: 0.0489
Loss after iteration 1600: 0.0476
Loss after iteration 1700: 0.0466
Loss after iteration 1800: 0.0458
Loss after iteration 1900: 0.0451
Loss after iteration 2000: 0.0446
Loss after iteration 2100: 0.0442
Loss after iteration 2200: 0.0438
Loss after iteration 2300: 0.0435
Loss after iteration 2400: 0.0433
Loss after iteration 2500: 0.0431
Loss after iteration 2600: 0.0429
Loss after iteration 2700: 0.0428
Loss after iteration 2800: 0.0427
Loss after iteration 2900: 0.0426
Loss after iteration 3000: 0.0425
Loss after iteration 3100: 0.0424
Loss after iteration 3200: 0.0423
Loss after iteration 3300: 0.0423
Loss after iteration 3400: 0.0422
Loss after iteration 3500: 0.0422
Loss after iteration 3600: 0.0421
Loss after iteration 3700: 0.0421
Loss after iteration 3800: 0.0421
Loss after iteration 3900: 0.0421
Loss after iteration 4000: 0.0420
```

21. Testing phase

Evaluating model on testing data

In [20]:
```
# 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) / le
n(pred_output))) + "%")
```

```
Testing accuracy is: 87.5621890547%
```

22. Results

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

1. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 89.05%
2. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 89.05%
3. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 77.11%
4. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0, Testing Accuracy = 89.05%
5. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 89.55%
6. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 89.55%
7. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 74.62%
8. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 90.04%
9. learning_rate = 0.05, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 89.05%
10. learning_rate = 0.01, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 89.05%
11. learning_rate = 0.001, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 68.65%
12. learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 90.04%
13. learning_rate = 0.05, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 89.05%
14. learning_rate = 0.01, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 89.05%
15. learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 89.55%
16. learning_rate = 0.005, num_epochs = 4000, momentum = 0.9, weight_decay = 0.001, Testing Accuracy = 89.045%
17. learning_rate = 0.05, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 90.04%
18. learning_rate = 0.01, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 90.04%
19. learning_rate = 0.001, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 89.55%

23. Optimal Hyper-parameters obtained

The optimal hyper-parameters obtained were as follows:

learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.01, Testing Accuracy = 90.04%
learning_rate = 0.005, num_epochs = 4000, momentum = 0, weight_decay = 0.001, Testing Accuracy = 90.04%
learning_rate = 0.05, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 90.04%
learning_rate = 0.01, num_epochs = 4000, momentum = 0.9, weight_decay = 0.01, Testing Accuracy = 90.04%

In [ ]: