UNIVERSITY OF CALIFORNIA, LOS ANGELES

# Final Project

*Author:*
Yining HONG

*Instructor:*
Yingnian Wu

*Final Project for STATS202A-Statistics Programming*

Department of Statistics

December 16, 2019

# Chapter 1

# Lasso

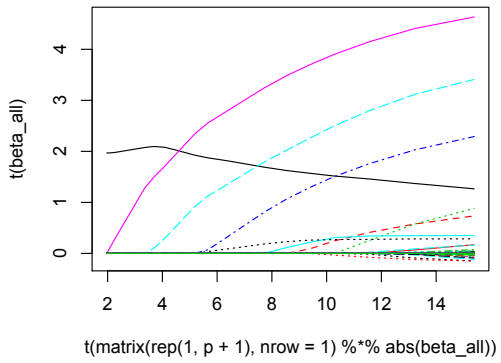



FIGURE 1.1: Solution Path of coordinate descent without epsilon-boosting

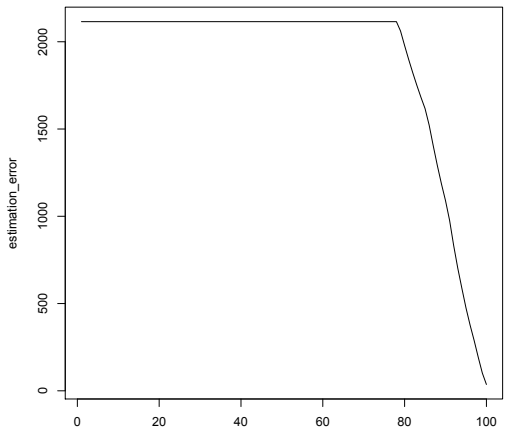



FIGURE 1.2: Error of coordinate descent without epsilon-boosting

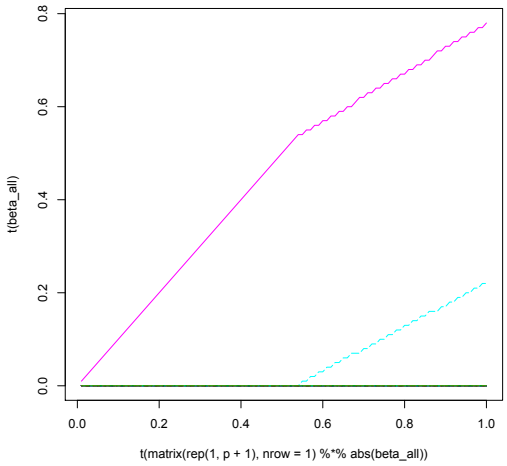



FIGURE 1.3: Solution Path of coordinate descent with epsilon-boosting

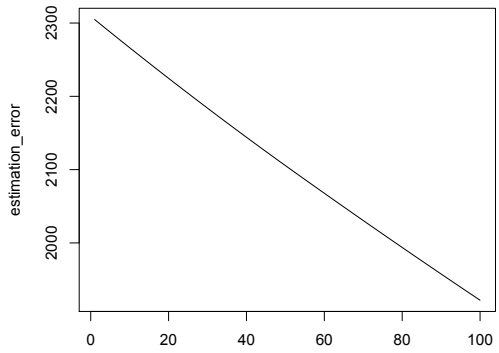



FIGURE 1.4: Error of coordinate descent with epsilon-boosting

## Chapter 2

# Python Coding of NN

# Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks on MNIST datasets. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradie
nts

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and ca
che,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Batch Normalization as a tool to more efficiently optimize deep networks.

In [1]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from stats202a.classifiers.fc_net import *
from stats202a.data_utils import get_mnist_data
from stats202a.gradient_check import eval_numerical_gradient, eval_
from stats202a.solver import Solver
from stats202a.layers import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modu
%reload_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.ab
```

## Download data

you need to download the MNIST datasets. Run the following bash in the `stats202a/datasets` directory: `./get_datasets.sh` (for windows, run `./get_datasets.cmd` )

In [2]:
```
# Load the (preprocessed) MNIST data.
# The second dimension of images indicated the number of channel. F
data = get_mnist_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (59000, 1, 28, 28))
('y_train: ', (59000,))
('X_val: ', (1000, 1, 28, 28))
('y_val: ', (1000,))
('X_test: ', (10000, 1, 28, 28))
('y_test: ', (10000,))
```

# Fully-connected layer: foward

Open the file `stats202a/layers.py` and implement the `fc_forward` function.

Once you are done you can test your implementaion by running the following:

In [3]:
```
# Test the fc_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *inp
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_s
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = fc_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing fc_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing fc_forward function:
difference:  9.769847728806635e-10
```

In [4]:
```
# Test the fc_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: fc_forward(x, w, b
dw_num = eval_numerical_gradient_array(lambda w: fc_forward(x, w, b
db_num = eval_numerical_gradient_array(lambda b: fc_forward(x, w, b

_, cache = fc_forward(x, w, b)
dx, dw, db = fc_backward(dout, cache)

# The error should be around 1e-10
print('Testing fc_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing fc_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

# Fully-connected layer: backward

Now implement the `fc_backward` function and test your implementation using numeric gradient checking.

# ReLU layer: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

In [5]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.
                        [ 0.,          0.,          0.04545455,  0.
                        [ 0.22727273,  0.31818182,  0.40909091,  0.

# Compare your output with ours. The error should be around 5e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

# ReLU layer: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

In [6]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)
dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0]
_, cache = relu_forward(x)
dx = relu_backward(dout, cache)
# The error should be around 3e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

# "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, fc/conv layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `stats202a/layer_utils.py` .

Implement the `fc_relu_forward` and `fc_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [7]:
```
from stats202a.layer_utils import fc_relu_forward, fc_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = fc_relu_forward(x, w, b)
dx, dw, db = fc_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: fc_relu_forward(x,
dw_num = eval_numerical_gradient_array(lambda w: fc_relu_forward(x,
db_num = eval_numerical_gradient_array(lambda b: fc_relu_forward(x,

print('Testing affine_relu_forward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

# Loss layers: Softmax

Now implement the softmax loss in the `softmax_loss` function.

The softmax loss is in the following form:

$$L_i = -\log(exp(x_{i_{y_i}}) / \sum_j (exp(x_{ij})))$$

$x_i$ is the output of the top fc layer for input image $i$, $y_i$ is the true label of $x_i$, and $j$ is the index of category. To avoid overflow, you may follow the trick in https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/ (https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/) to compute the 'logsumexp' operation.

You can make sure that the implementations are correct by running the following:

In [8]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error shoul
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

# Two-layer network

First we implement a two-layer network with only one hidden layer. We will use the class TwoLayerNet in the file `stats202a/classifiers/fc_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays.

Besides softmax loss, we add another L2 regularization loss: $||W||_2^2$, where $W$ is the weights of all layers. Bias are not included. We use a parameter `self.reg` to control the strength of regularization.

Open the file `stats202a/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [9]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C)

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
```

```
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.5719
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.8114
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.0509
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-tim


model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularizati

for reg in [0.0, 0.7]:
    print('Running numeric gradient check with reg = ', reg)
    model.reg = 0
    loss, grads = model.loss(X, y)

    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], v
        print('%s relative error: %.2e' % (name, rel_error(grad_num
```

```
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
```

# Solver

We use a separate class to define the training process.

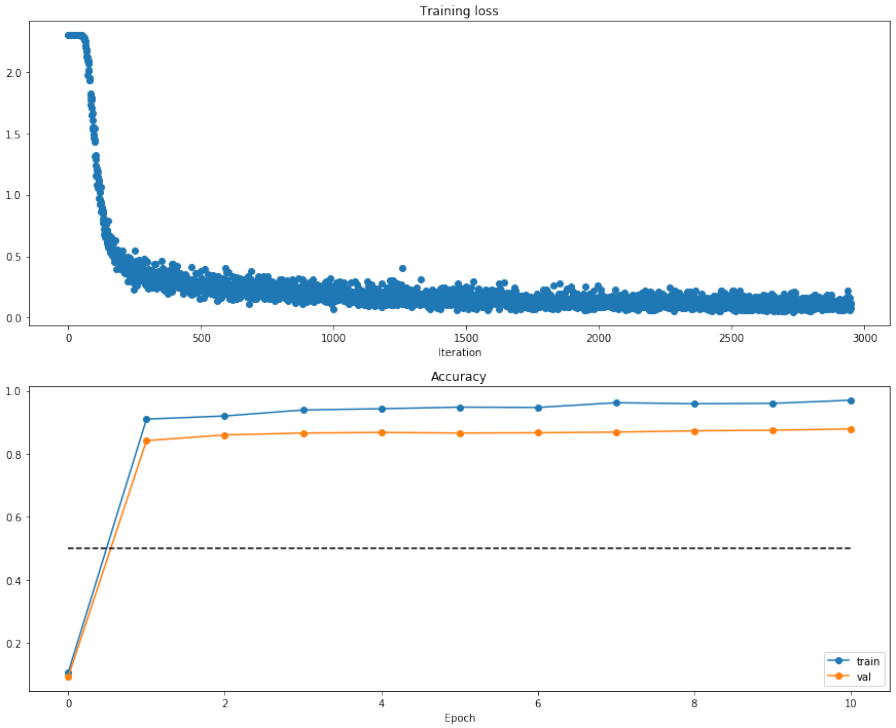
Open the file `stats202a/solver.py` and read through it to familiarize yourself with the API. You can use a `Solver` instance to train a `TwoLayerNet` that achieves at least `97%` accuracy on the validation set. Just run the code.

In [10]:
```
model = TwoLayerNet()
solver = None
```

```
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=10, batch_size=200,
                print_every=100)
solver.train()
```

```
(Iteration 1 / 2950) loss: 2.302585
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.093000
(Iteration 101 / 2950) loss: 1.448799
(Iteration 201 / 2950) loss: 0.465498
(Epoch 1 / 10) train acc: 0.910000; val_acc: 0.842000
(Iteration 301 / 2950) loss: 0.395597
(Iteration 401 / 2950) loss: 0.265204
(Iteration 501 / 2950) loss: 0.279966
(Epoch 2 / 10) train acc: 0.920000; val_acc: 0.860000
(Iteration 601 / 2950) loss: 0.245076
(Iteration 701 / 2950) loss: 0.155889
(Iteration 801 / 2950) loss: 0.296768
(Epoch 3 / 10) train acc: 0.939000; val_acc: 0.866000
(Iteration 901 / 2950) loss: 0.215426
(Iteration 1001 / 2950) loss: 0.164655
(Iteration 1101 / 2950) loss: 0.143659
(Epoch 4 / 10) train acc: 0.943000; val_acc: 0.868000
(Iteration 1201 / 2950) loss: 0.243315
(Iteration 1301 / 2950) loss: 0.159187
(Iteration 1401 / 2950) loss: 0.107042
(Epoch 5 / 10) train acc: 0.948000; val_acc: 0.866000
(Iteration 1501 / 2950) loss: 0.125140
(Iteration 1601 / 2950) loss: 0.168138
(Iteration 1701 / 2950) loss: 0.106984
(Epoch 6 / 10) train acc: 0.947000; val_acc: 0.867000
(Iteration 1801 / 2950) loss: 0.229146
(Iteration 1901 / 2950) loss: 0.147101
(Iteration 2001 / 2950) loss: 0.162600
(Epoch 7 / 10) train acc: 0.962000; val_acc: 0.869000
(Iteration 2101 / 2950) loss: 0.153283
(Iteration 2201 / 2950) loss: 0.103785
(Iteration 2301 / 2950) loss: 0.136438
(Epoch 8 / 10) train acc: 0.959000; val_acc: 0.873000
(Iteration 2401 / 2950) loss: 0.116733
(Iteration 2501 / 2950) loss: 0.092463
(Iteration 2601 / 2950) loss: 0.064349
(Epoch 9 / 10) train acc: 0.960000; val_acc: 0.875000
(Iteration 2701 / 2950) loss: 0.106376
(Iteration 2801 / 2950) loss: 0.172919
(Iteration 2901 / 2950) loss: 0.093528
(Epoch 10 / 10) train acc: 0.970000; val_acc: 0.879000
```

In [11]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



# Multilayer network (Optional)

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `stats202a/classifiers/fc_net.py`.

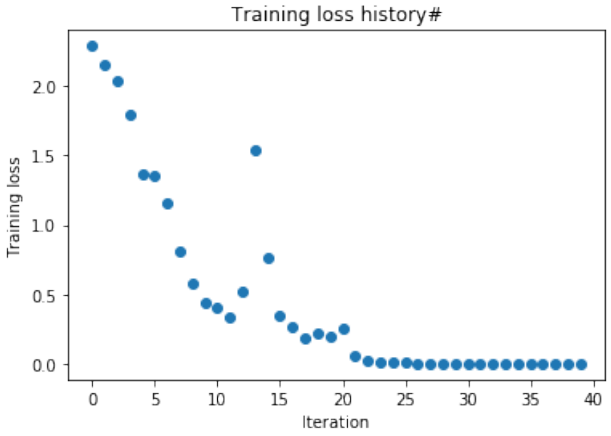
Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch normalization; we will add those features soon.

As a sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. You will need to tweak the learning rate and initialization scale, but you should be able to overfit and achieve 100% training accuracy within 20 epochs.

In [12]:
```
# TODO: Use a three-layer Net to overfit 50 training examples.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

small_data['X_train'].shape

weight_scale = 1e-2
learning_rate = 3e-2
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history#')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.287239
```

```
(Epoch 0 / 20) train acc: 0.000000; val_acc: 0.183000
(Epoch 1 / 20) train acc: 0.000000; val_acc: 0.181000
(Epoch 2 / 20) train acc: 0.000000; val_acc: 0.366000
(Epoch 3 / 20) train acc: 0.000000; val_acc: 0.427000
(Epoch 4 / 20) train acc: 0.000000; val_acc: 0.517000
(Epoch 5 / 20) train acc: 0.000000; val_acc: 0.548000
(Iteration 11 / 40) loss: 0.409455
(Epoch 6 / 20) train acc: 0.000000; val_acc: 0.530000
(Epoch 7 / 20) train acc: 0.000000; val_acc: 0.405000
(Epoch 8 / 20) train acc: 0.000000; val_acc: 0.546000
(Epoch 9 / 20) train acc: 0.000000; val_acc: 0.575000
(Epoch 10 / 20) train acc: 0.000000; val_acc: 0.570000
(Iteration 21 / 40) loss: 0.254674
(Epoch 11 / 20) train acc: 0.000000; val_acc: 0.598000
(Epoch 12 / 20) train acc: 0.000000; val_acc: 0.604000
(Epoch 13 / 20) train acc: 0.000000; val_acc: 0.605000
(Epoch 14 / 20) train acc: 0.000000; val_acc: 0.601000
(Epoch 15 / 20) train acc: 0.000000; val_acc: 0.603000
(Iteration 31 / 40) loss: 0.005785
(Epoch 16 / 20) train acc: 0.000000; val_acc: 0.603000
(Epoch 17 / 20) train acc: 0.000000; val_acc: 0.604000
(Epoch 18 / 20) train acc: 0.000000; val_acc: 0.604000
(Epoch 19 / 20) train acc: 0.000000; val_acc: 0.604000
(Epoch 20 / 20) train acc: 0.000000; val_acc: 0.604000
```

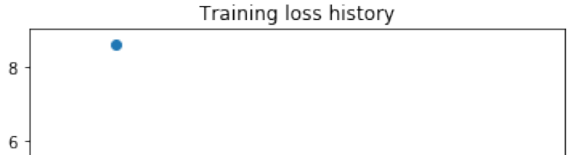


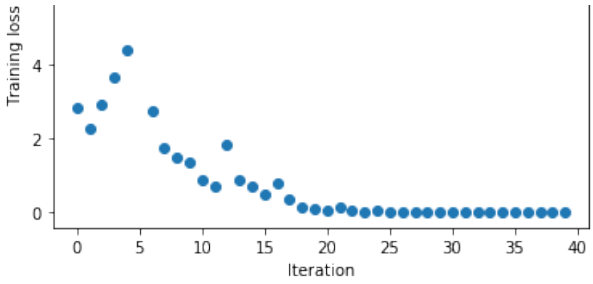

Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [13]:
```
# TODO: Use a five-layer Net to overfit 50 training examples.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
```

```
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 5e-2
weight_scale = 5e-2
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.835779
(Epoch 0 / 20) train acc: 0.000000; val_acc: 0.214000
(Epoch 1 / 20) train acc: 0.000000; val_acc: 0.126000
(Epoch 2 / 20) train acc: 0.000000; val_acc: 0.120000
(Epoch 3 / 20) train acc: 0.000000; val_acc: 0.098000
(Epoch 4 / 20) train acc: 0.000000; val_acc: 0.313000
(Epoch 5 / 20) train acc: 0.000000; val_acc: 0.369000
(Iteration 11 / 40) loss: 0.858194
(Epoch 6 / 20) train acc: 0.000000; val_acc: 0.284000
(Epoch 7 / 20) train acc: 0.000000; val_acc: 0.283000
(Epoch 8 / 20) train acc: 0.000000; val_acc: 0.449000
(Epoch 9 / 20) train acc: 0.000000; val_acc: 0.481000
(Epoch 10 / 20) train acc: 0.000000; val_acc: 0.494000
(Iteration 21 / 40) loss: 0.033513
(Epoch 11 / 20) train acc: 0.000000; val_acc: 0.490000
(Epoch 12 / 20) train acc: 0.000000; val_acc: 0.507000
(Epoch 13 / 20) train acc: 0.000000; val_acc: 0.519000
(Epoch 14 / 20) train acc: 0.000000; val_acc: 0.520000
(Epoch 15 / 20) train acc: 0.000000; val_acc: 0.516000
(Iteration 31 / 40) loss: 0.012882
(Epoch 16 / 20) train acc: 0.000000; val_acc: 0.524000
(Epoch 17 / 20) train acc: 0.000000; val_acc: 0.525000
(Epoch 18 / 20) train acc: 0.000000; val_acc: 0.525000
(Epoch 19 / 20) train acc: 0.000000; val_acc: 0.525000
(Epoch 20 / 20) train acc: 0.000000; val_acc: 0.526000
```






In [ ]:

In [ ]:

We can see that for size 50, **the loss keep decreasing towards 0**. Actually, if we set the dataset size to 5000, the training accuracy can easily get to super high.

```
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
(Iteration 3821 / 4000) loss: 0.077876
(Iteration 3831 / 4000) loss: 0.000708
(Iteration 3841 / 4000) loss: 0.001907
(Iteration 3851 / 4000) loss: 0.094007
(Iteration 3861 / 4000) loss: 0.007908
(Iteration 3871 / 4000) loss: 0.011833
(Iteration 3881 / 4000) loss: 0.001717
(Iteration 3891 / 4000) loss: 0.000342
(Iteration 3901 / 4000) loss: 0.029193
(Iteration 3911 / 4000) loss: 0.097772
(Iteration 3921 / 4000) loss: 0.053752
(Iteration 3931 / 4000) loss: 0.074677
(Iteration 3941 / 4000) loss: 0.009330
(Iteration 3951 / 4000) loss: 0.001568
(Iteration 3961 / 4000) loss: 0.344843
(Iteration 3971 / 4000) loss: 0.065537
(Iteration 3981 / 4000) loss: 0.057929
(Iteration 3991 / 4000) loss: 0.062057
(Epoch 20 / 20) train acc: 0.981000; val_acc: 0.850000
```

# Chapter 3

# Play with TF and PyTorch.

We play with Tensorflow with learning rate 0.01, 0.1 and 0.05. We can see that with 0.01, the accuracy increases very slow, and only reaches 97 at epoch 98. With 0.1, accuracy increases sharply but is very unstable. With 0.05, the accuracy increases both stable and fast.

We play with pytorch with learning rate 0.1, 0.01 and 0.5. And see that with 0.1, we easily get to accuracy 97 after six or seven epochs, 0.01 performs quite bad, while with 0.5 we get to 97 after only four epochs.

we print the accuracies as well as **plot the accuracy result**.

Below, we will present six pdf files. Which are:

- tf with lr 0.01
- tf with lr 0.1
- tf with lr 0.05
- pytorch with lr 0.01
- pytorch with lr 0.1
- pytorch with lr 0.5

In [1]:
```
# Official tutorials: https://www.tensorflow.org/tutorials
# keras totutial for MNIST: https://www.tensorflow.org/tutorials/quickstart
```

In [2]:
```
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/compat/v2_compat.py:65: disable_resource_var
iables (from tensorflow.python.ops.variable_scope) is deprecated and will
be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

In [3]:
```
# Build an easy calculator
a = tf.placeholder(dtype=tf.float32, shape=[3,3])
b = tf.placeholder(dtype=tf.float32, shape=[3,3])
c = a+b
d = tf.matmul(a, b)
print(a)
print(b)
print(c)
print(d)
```

```
Tensor("Placeholder:0", shape=(3, 3), dtype=float32)
Tensor("Placeholder_1:0", shape=(3, 3), dtype=float32)
Tensor("add:0", shape=(3, 3), dtype=float32)
Tensor("MatMul:0", shape=(3, 3), dtype=float32)
```

In [4]:
```
sess = tf.Session()
a_input = np.array([[1,1,1],[2,2,2],[3,3,3]])
b_input = np.array([[1,2,3],[1,2,3],[1,2,3]])
my_feed_dict = {a: a_input, b: b_input}
res = sess.run([c,d], feed_dict=my_feed_dict)
print(res[0])
print(res[1])
```

```
[[2. 3. 4.]
 [3. 4. 5.]
 [4. 5. 6.]]
[[ 3.  6.  9.]
 [ 6. 12. 18.]
 [ 9. 18. 27.]]
```

In [5]:
```
e = tf.Variable(0.0)
e_add = tf.assign(e, e+1)
```

In [ ]:
```
print(sess.run(e))
```

In [6]:
```
sess.run(tf.global_variables_initializer())
print(sess.run(e))
sess.run(e_add)
print(sess.run(e))
```

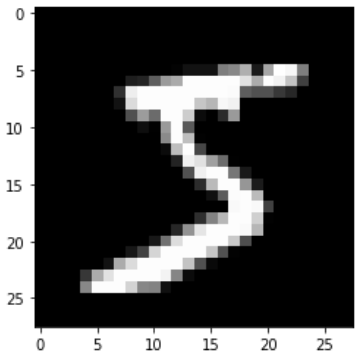
```
0.0
1.0
```

In [7]:
```
# build an easy neuron network
# load in the data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print(x_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

In [8]:
```
import matplotlib.pyplot as plt
plt.imshow(x_train[0], cmap='gray')
```

Out[8]: <matplotlib.image.AxesImage at 0x64cab7fd0>



In [9]:
```
# define structure: 784-->256-->10
input_img = tf.placeholder(dtype=tf.float32, shape=[None, 28*28], name='inp
labels = tf.placeholder(dtype=tf.int32, shape=[None], name='label')
h1 = tf.layers.dense(input_img, units=256, name='h1')
h1 = tf.nn.relu(h1)
h2 = tf.layers.dense(h1, units=10, name='h2')
output = tf.nn.softmax(h2)
print(h1.shape)
print(h2.shape)
print(output.shape)
print(labels.shape)
```

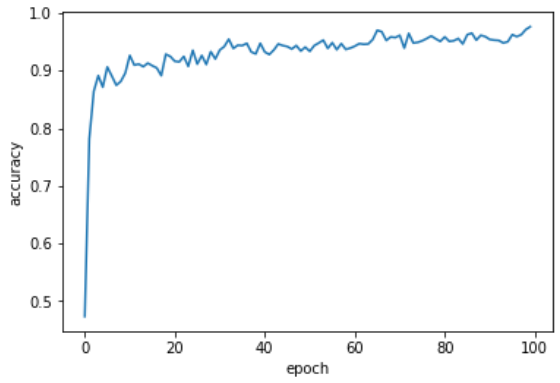
```
WARNING:tensorflow:From <ipython-input-9-509994677059>:4: dense (from ten
sorflow.python.layers.core) is deprecated and will be removed in a future
version.
Instructions for updating:
Use keras.layers.Dense instead.
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/layers/core.py:187: Layer.apply (from tensor
flow.python.keras.engine.base_layer) is deprecated and will be removed in
a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
(?, 256)
(?, 10)
(?, 10)
(?,)
```

In [15]:
```
# define loss and optimizer
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
update = optimizer.minimize(loss)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

In [16]:
```
epoch = 100
epoch_accuracies = []
num_iter = 180
start = 0
for j in range (epoch):
    epoch_accuracy = 0.0
    for it in range (num_iter):
        start += 10
        start %= 60000
        cur_input = np.reshape(x_train[start:(start+10)], (10, 784))
        cur_label = y_train[start:(start+10)]
        my_feed_dict = {input_img: cur_input, labels:cur_label}
        preds,_ = sess.run([output, update], feed_dict=my_feed_dict)
        preds_label = np.argmax(preds, axis=1)
        acc_iter = np.sum(1*(preds_label)==(cur_label))/10
        epoch_accuracy += acc_iter
    epoch_accuracy =  epoch_accuracy/180
    if (j >= 90):
        print (j)
        print (epoch_accuracy)
    epoch_accuracies.append(epoch_accuracy)
```

```
90
0.9544444444444453
91
0.9533333333333343
92
0.9527777777777791
93
0.948333333333334
94
0.9505555555555563
95
0.9627777777777786
96
0.9594444444444454
97
0.9627777777777786
98
0.9716666666666671
99
0.976666666666667
```

In [17]:
```
plt.plot(epoch_accuracies)
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```




In [ ]:
```

```

In [1]:
```
# Official tutorials: https://www.tensorflow.org/tutorials
# keras totutial for MNIST: https://www.tensorflow.org/tutorials/quickstart
```

In [2]:
```
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/compat/v2_compat.py:65: disable_resource_var
iables (from tensorflow.python.ops.variable_scope) is deprecated and will
be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

In [3]:
```
# Build an easy calculator
a = tf.placeholder(dtype=tf.float32, shape=[3,3])
b = tf.placeholder(dtype=tf.float32, shape=[3,3])
c = a+b
d = tf.matmul(a, b)
print(a)
print(b)
print(c)
print(d)
```

```
Tensor("Placeholder:0", shape=(3, 3), dtype=float32)
Tensor("Placeholder_1:0", shape=(3, 3), dtype=float32)
Tensor("add:0", shape=(3, 3), dtype=float32)
Tensor("MatMul:0", shape=(3, 3), dtype=float32)
```

In [4]:
```
sess = tf.Session()
a_input = np.array([[1,1,1],[2,2,2],[3,3,3]])
b_input = np.array([[1,2,3],[1,2,3],[1,2,3]])
my_feed_dict = {a: a_input, b: b_input}
res = sess.run([c,d], feed_dict=my_feed_dict)
print(res[0])
print(res[1])
```

```
[[2. 3. 4.]
 [3. 4. 5.]
 [4. 5. 6.]]
[[ 3.  6.  9.]
 [ 6. 12. 18.]
 [ 9. 18. 27.]]
```

In [5]:
```
e = tf.Variable(0.0)
e_add = tf.assign(e, e+1)
```

In [ ]:
```
print(sess.run(e))
```

In [6]:
```
sess.run(tf.global_variables_initializer())
print(sess.run(e))
sess.run(e_add)
print(sess.run(e))
```

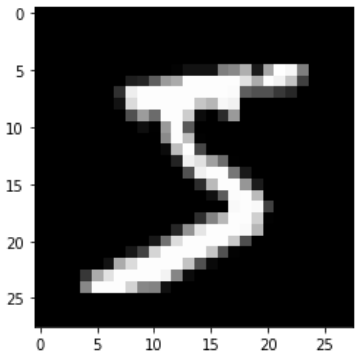
```
0.0
1.0
```

In [7]:
```
# build an easy neuron network
# load in the data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print(x_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

In [8]:
```
import matplotlib.pyplot as plt
plt.imshow(x_train[0], cmap='gray')
```

Out[8]: <matplotlib.image.AxesImage at 0x64cab7fd0>



In [9]:
```
# define structure: 784-->256-->10
input_img = tf.placeholder(dtype=tf.float32, shape=[None, 28*28], name='inp
labels = tf.placeholder(dtype=tf.int32, shape=[None], name='label')
h1 = tf.layers.dense(input_img, units=256, name='h1')
h1 = tf.nn.relu(h1)
h2 = tf.layers.dense(h1, units=10, name='h2')
output = tf.nn.softmax(h2)
print(h1.shape)
print(h2.shape)
print(output.shape)
print(labels.shape)
```

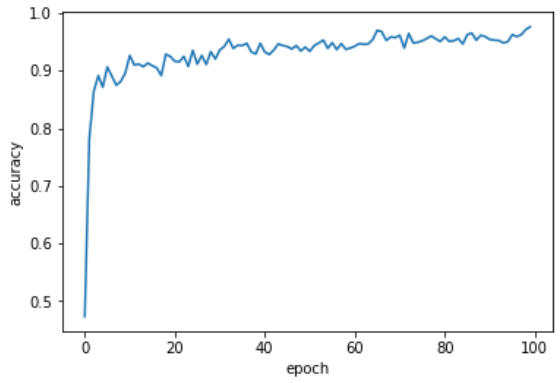
```
WARNING:tensorflow:From <ipython-input-9-509994677059>:4: dense (from ten
sorflow.python.layers.core) is deprecated and will be removed in a future
version.
Instructions for updating:
Use keras.layers.Dense instead.
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/layers/core.py:187: Layer.apply (from tensor
flow.python.keras.engine.base_layer) is deprecated and will be removed in
a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
(?, 256)
(?, 10)
(?, 10)
(?,)
```

In [15]:
```
# define loss and optimizer
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
update = optimizer.minimize(loss)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

In [16]:
```
epoch = 100
epoch_accuracies = []
num_iter = 180
start = 0
for j in range (epoch):
    epoch_accuracy = 0.0
    for it in range (num_iter):
        start += 10
        start %= 60000
        cur_input = np.reshape(x_train[start:(start+10)], (10, 784))
        cur_label = y_train[start:(start+10)]
        my_feed_dict = {input_img: cur_input, labels:cur_label}
        preds,_ = sess.run([output, update], feed_dict=my_feed_dict)
        preds_label = np.argmax(preds, axis=1)
        acc_iter = np.sum(1*(preds_label)==(cur_label))/10
        epoch_accuracy += acc_iter
    epoch_accuracy =  epoch_accuracy/180
    if (j >= 90):
        print (j)
        print (epoch_accuracy)
    epoch_accuracies.append(epoch_accuracy)
```

```
90
0.9544444444444453
91
0.9533333333333343
92
0.9527777777777791
93
0.948333333333334
94
0.9505555555555563
95
0.9627777777777786
96
0.9594444444444454
97
0.9627777777777786
98
0.9716666666666671
99
0.976666666666667
```

In [17]:
```
plt.plot(epoch_accuracies)
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```




In [ ]:
```

```

In [1]:
```
# Official tutorials: https://www.tensorflow.org/tutorials
# keras totutial for MNIST: https://www.tensorflow.org/tutorials/quickstart
```

In [2]:
```
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/compat/v2_compat.py:65: disable_resource_var
iables (from tensorflow.python.ops.variable_scope) is deprecated and will
be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

In [3]:
```
# Build an easy calculator
a = tf.placeholder(dtype=tf.float32, shape=[3,3])
b = tf.placeholder(dtype=tf.float32, shape=[3,3])
c = a+b
d = tf.matmul(a, b)
print(a)
print(b)
print(c)
print(d)
```

```
Tensor("Placeholder:0", shape=(3, 3), dtype=float32)
Tensor("Placeholder_1:0", shape=(3, 3), dtype=float32)
Tensor("add:0", shape=(3, 3), dtype=float32)
Tensor("MatMul:0", shape=(3, 3), dtype=float32)
```

In [4]:
```
sess = tf.Session()
a_input = np.array([[1,1,1],[2,2,2],[3,3,3]])
b_input = np.array([[1,2,3],[1,2,3],[1,2,3]])
my_feed_dict = {a: a_input, b: b_input}
res = sess.run([c,d], feed_dict=my_feed_dict)
print(res[0])
print(res[1])
```

```
[[2. 3. 4.]
 [3. 4. 5.]
 [4. 5. 6.]]
[[ 3.  6.  9.]
 [ 6. 12. 18.]
 [ 9. 18. 27.]]
```

In [5]:
```
e = tf.Variable(0.0)
e_add = tf.assign(e, e+1)
```

In [ ]:
```
print(sess.run(e))
```

In [6]:
```
sess.run(tf.global_variables_initializer())
print(sess.run(e))
sess.run(e_add)
print(sess.run(e))
```

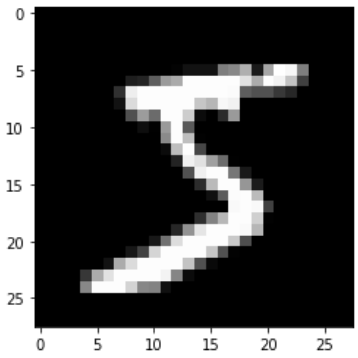
```
0.0
1.0
```

In [7]:
```
# build an easy neuron network
# load in the data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print(x_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

In [8]:
```
import matplotlib.pyplot as plt
plt.imshow(x_train[0], cmap='gray')
```

Out[8]: <matplotlib.image.AxesImage at 0x64cab7fd0>



In [9]:
```
# define structure: 784-->256-->10
input_img = tf.placeholder(dtype=tf.float32, shape=[None, 28*28], name='inp
labels = tf.placeholder(dtype=tf.int32, shape=[None], name='label')
h1 = tf.layers.dense(input_img, units=256, name='h1')
h1 = tf.nn.relu(h1)
h2 = tf.layers.dense(h1, units=10, name='h2')
output = tf.nn.softmax(h2)
print(h1.shape)
print(h2.shape)
print(output.shape)
print(labels.shape)
```

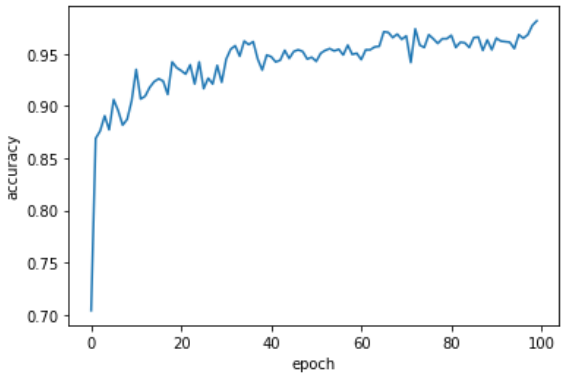
```
WARNING:tensorflow:From <ipython-input-9-509994677059>:4: dense (from ten
sorflow.python.layers.core) is deprecated and will be removed in a future
version.
Instructions for updating:
Use keras.layers.Dense instead.
WARNING:tensorflow:From /Users/chengpeng/anaconda3/lib/python3.7/site-pac
kages/tensorflow_core/python/layers/core.py:187: Layer.apply (from tensor
flow.python.keras.engine.base_layer) is deprecated and will be removed in
a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
(?, 256)
(?, 10)
(?, 10)
(?,)
```

In [12]:
```
# define loss and optimizer
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.05)
update = optimizer.minimize(loss)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

In [13]:
```
epoch = 100
epoch_accuracies = []
num_iter = 180
start = 0
for j in range (epoch):
    epoch_accuracy = 0.0
    for it in range (num_iter):
        start += 10
        start %= 60000
        cur_input = np.reshape(x_train[start:(start+10)], (10, 784))
        cur_label = y_train[start:(start+10)]
        my_feed_dict = {input_img: cur_input, labels:cur_label}
        preds,_ = sess.run([output, update], feed_dict=my_feed_dict)
        preds_label = np.argmax(preds, axis=1)
        acc_iter = np.sum(1*(preds_label)==(cur_label))/10
        epoch_accuracy += acc_iter
    epoch_accuracy =  epoch_accuracy/180
    if (j >= 90):
        print (j)
        print (epoch_accuracy)
    epoch_accuracies.append(epoch_accuracy)
```

```
90
0.9650000000000009
91
0.9622222222222233
92
0.9616666666666674
93
0.9611111111111119
94
0.9550000000000008
95
0.9683333333333342
96
0.9650000000000011
97
0.9683333333333342
98
0.9772222222222228
99
0.9816666666666671
```

In [14]:
```
plt.plot(epoch_accuracies)
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```




In [ ]:
```

```

In [1]:
```
#how to use google colab: https://pytorch.org/tutorials/beginner/colab.html
#pytorch tutorial: https://pytorch.org/tutorials/beginner/deep_learning_60m
import torch
import torch.nn.functional as F
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# define the network structure
class fc_net(torch.nn.Module):
  def __init__(self, num_in, num_out):
    super(fc_net, self).__init__()
    # Initialize two linear neural networks, one as the input layer and one
    self.h1 = torch.nn.Linear(in_features=num_in, out_features=256)
    self.h2 = torch.nn.Linear(in_features=256, out_features=num_out)
  def forward(self, inputs):
    # We use relu as the activation function as the first layer
    a1 = F.relu(self.h1(inputs))
    # We use softmax as the activation function as the second layer
    a2 = F.softmax(self.h2(a1),dim=-1)
    return a2
```

In [3]:
```
# use data_loader to load in data
train_data = datasets.MNIST('./', train=True, download=True, transform=tran
# We use DataLoader to load the train data.
# We specify the batch_size and the DataLoader will return the split for us
# We use shuffle = False so we won't shuffle the data.
train_loader = torch.utils.data.DataLoader(train_data, batch_size=10, shuff
batch_size = 10
# Model is a fully-connected net.
model = fc_net(num_in=28*28, num_out=10)
# We use Cross Entropy as our loss.
loss = torch.nn.CrossEntropyLoss()
# We use SGD as our optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
0it [00:00, ?it/s]

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz) to ./MNIS
T/raw/train-images-idx3-ubyte.gz

100%|██████████| 9863168/9912422 [00:14<00:00, 968768.48it/s]

Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw


0it [00:00, ?it/s]
  0%|          | 0/28881 [00:00<?, ?it/s]

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz) to ./MNIS
T/raw/train-labels-idx1-ubyte.gz

32768it [00:00, 119758.21it/s]

0it [00:00, ?it/s]
  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (h
ttp://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz) to ./MNIST/ra
w/t10k-images-idx3-ubyte.gz


  1%|          | 16384/1648877 [00:00<00:10, 157532.18it/s]
  2%|▏         | 40960/1648877 [00:00<00:10, 159512.89it/s]
 10%|▉         | 163840/1648877 [00:00<00:07, 211265.33it/s]
 18%|█▊        | 294912/1648877 [00:00<00:04, 274775.81it/s]
 27%|██▋       | 442368/1648877 [00:00<00:03, 352508.46it/s]
 36%|███▌      | 598016/1648877 [00:00<00:02, 444743.45it/s]
 46%|████▌     | 761856/1648877 [00:01<00:01, 513049.54it/s]
 59%|█████▉    | 974848/1648877 [00:01<00:01, 628264.23it/s]
 68%|██████▊   | 1114112/1648877 [00:01<00:00, 707715.96it/s]
 76%|███████▌  | 1253376/1648877 [00:01<00:00, 779105.93it/s]
 85%|████████▍ | 1400832/1648877 [00:01<00:00, 850785.19it/s]
1654784it [00:01, 842170.38it/s]

0it [00:00, ?it/s]
8192it [00:00, 55022.33it/s]
```

In [1]:
```
#how to use google colab: https://pytorch.org/tutorials/beginner/co
#pytorch tutorial: https://pytorch.org/tutorials/beginner/deep_lear
import torch
import torch.nn.functional as F
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# define the network structure
class fc_net(torch.nn.Module):
 def __init__(self, num_in, num_out):
   super(fc_net, self).__init__()
   # Initialize two linear neural networks, one as the input layer a
   self.h1 = torch.nn.Linear(in_features=num_in, out_features=256)
   self.h2 = torch.nn.Linear(in_features=256, out_features=num_out)
 def forward(self, inputs):
   # We use relu as the activation function as the first layer
   a1 = F.relu(self.h1(inputs))
   # We use softmax as the activation function as the second layer
   a2 = F.softmax(self.h2(a1),dim=-1)
   return a2
```

In [3]:
```
# use data_loader to load_in data
train_data = datasets.MNIST('./', train=True, download=True, transf
# We use DataLoader to load the train data.
# We specify the batch_size and the DataLoader will return the spli
# We use shuffle = False so we won't shuffle the data.
train_loader = torch.utils.data.DataLoader(train_data, batch_size=1
batch_size = 10
# Model is a fully-connected net.
model = fc_net(num_in=28*28, num_out=10)
# We use Cross Entropy as our loss.
loss = torch.nn.CrossEntropyLoss()
# We use SGD as our optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

In [4]:
```
epoch = 10
epoch_accuracies = []
for j in range (epoch):
    loader = iter(train_loader)
    epoch_accuracy = 0.0
    print (j)
    for i in range (1, len(train_loader)):
        # cur_s, cur_y gets the data and label for the next iterati
        cur_x, cur_y = next(loader)
        # We unsqueeze the data for one batch.
        cur_x = torch.reshape(cur_x, (10, 28*28))
        # We get the predictions, which are probabilities, after th
        preds = model.forward(cur_x)
        # Use the predictions and labels to compute loss.
        cur_loss = loss(preds, cur_y)
        optimizer.zero_grad()
        # Backward
        cur_loss.backward()
        optimizer.step()
        preds_numpy = preds.detach().numpy()
        preds_label = np.argmax(preds_numpy, axis=1)
        cur_y_numpy = cur_y.detach().numpy()
        acc_iter = np.sum(1*(preds_label)==(cur_y_numpy))/batch_siz
        epoch_accuracy += acc_iter
    epoch_accuracy =  epoch_accuracy/len(train_loader)
    epoch_accuracies.append(epoch_accuracy)
    print (epoch_accuracy)
#         new_preds = model.forward(cur_x)
#         print(new_preds[0])
```

```
0
0.8604666666666702
1
0.9342666666666593
2
0.9492333333333249
3
0.9586333333333246
4
0.965449999999991
5
0.9702833333333245
6
0.9743333333333243
7
0.9771666666666581
8
0.9798166666666586
9
[5 0 4 1 9 2 1 3 1 4]
```

In [ ]:
```
```

In [1]:
```
#how to use google colab: https://pytorch.org/tutorials/beginner/co
#pytorch tutorial: https://pytorch.org/tutorials/beginner/deep_lear
import torch
import torch.nn.functional as F
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# define the network structure
class fc_net(torch.nn.Module):
  def __init__(self, num_in, num_out):
    super(fc_net, self).__init__()
    # Initialize two linear neural networks, one as the input layer
    self.h1 = torch.nn.Linear(in_features=num_in, out_features=256)
    self.h2 = torch.nn.Linear(in_features=256, out_features=num_out
  def forward(self, inputs):
    # We use relu as the activation function as the first layer
    a1 = F.relu(self.h1(inputs))
    # We use softmax as the activation function as the second layer
    a2 = F.softmax(self.h2(a1),dim=-1)
    return a2
```

In [3]:
```
# use data_loader to load_in data
train_data = datasets.MNIST('./', train=True, download=True, transf
# We use DataLoader to load the train data.
# We specify the batch_size and the DataLoader will return the spli
# We use shuffle = False so we won't shuffle the data.
train_loader = torch.utils.data.DataLoader(train_data, batch_size=1
batch_size = 10
# Model is a fully-connected net.
model = fc_net(num_in=28*28, num_out=10)
# We use Cross Entropy as our loss.
loss = torch.nn.CrossEntropyLoss()
# We use SGD as our optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
```

In [4]:
```
epoch = 10
epoch_accuracies = []
for j in range (epoch):
    loader = iter(train_loader)
    epoch_accuracy = 0.0
    print (j)
    for i in range (1, len(train_loader)):
        # cur_s, cur_y gets the data and label for the next iterati
        cur_x, cur_y = next(loader)
        # We unsqueeze the data for one batch.
        cur_x = torch.reshape(cur_x, (10, 28*28))
        # We get the predictions, which are probabilities, after th
        preds = model.forward(cur_x)
        # Use the predictions and labels to compute loss.
        cur_loss = loss(preds, cur_y)
        optimizer.zero_grad()
        # Backward
        cur_loss.backward()
        optimizer.step()
        preds_numpy = preds.detach().numpy()
        preds_label = np.argmax(preds_numpy, axis=1)
        cur_y_numpy = cur_y.detach().numpy()
        acc_iter = np.sum(1*(preds_label)==(cur_y_numpy))/batch_siz
        epoch_accuracy += acc_iter
    epoch_accuracy =  epoch_accuracy/len(train_loader)
    epoch_accuracies.append(epoch_accuracy)
    print (epoch_accuracy)
#         new_preds = model.forward(cur_x)
#         print(new_preds[0])
```

```
0
0.8963166666666643
1
0.9503499999999919
2
0.9609499999999929
3
0.9667333333333245
4
0.9705333333333254
5
0.9737666666666573
6
0.9759499999999905
7
0.9772666666666581
8
0.9795333333333224
9
0.9796499999999911
```

In [ ]:
```

```

# Chapter 4

# Memo for Previous Projects

## 4.1 Homework01-Sampling

### 4.1.1 Problem Description

- Write R code for the following random number generators:

  1. Uniform[0, 1], using the linear congruential method.
  2. Exponential(1), using the inversion method.
  3. Normal(0, 1), using the Polar method.

  For (a), generate a sequence $X_t$, $t = 0, 1, ..., t, ...$ and draw the histogram of $X_t$ , and the scatterplot of $(X_t, X_{t+1})$ (Fig 2).

  For (b), draw the histogram (Fig 3).

  For (c), after generating $(X_t, Y_t)$ using the Polar method, draw the scatterplot of $(X_t, Y_t)$ (Fig 4). Draw the histogram of $T = R^2/2$ (Fig 5).

- Write R code for Monte Carlo computation of pi, by generating $(X_t, Y_t)$ from unit square $[0, 1]^2$, and computing the frequency that the points fall below $x^2 + y^2 = 1$. Please also use Monte Carlo method to compute the volume of d-dimensional unit ball, for d = 5 and 10.

### 4.1.2 Uniform Using Linear Congruential Method

Computers are deterministic. There is no way to generate true random numbers. In practice, people use algorithms to generate psudorandom numbers, which appears to be random and uniformly distribution if you don't know the seed.

One of the most common generator is the **linear congruential generator**:

- Given $X_0$ as a seed.
- Iteratively generate $X_{t+1} = (aX_t + b) \bmod M \in 0, 1, \ldots, M$.
- Obtain $U_t = \frac{X_t}{M} \sim \text{unif}[0, 1]$. $U_t$ is discrete.
- We can do this iteratively. So from seed $X_0$ we get $X_1$, from $X_1$ we get $X_2$...Finally we get $X_T$ so we have T random numbers.

Here $a, b, M$ are some carefully chosen integers; $M$ is chosen to be a large number to avoid the sequence repeating itself too soon.

R code for generating uniform random numbers is as below.

```
sample_uniform <- function(low=0, high=1){
 random = rep(0,10000)
 randomy = rep (0,10000)

 m = 2 ** 31 - 1
 a = 7 ** 5
 c = 12345

   # Set the seed using the current system time in microseconds
 d = as.numeric(Sys.time()) * 1000

 for (i in 1:10000) {
     d = (a * d + c) %% m
     random[i] = d / m * (high - low) + low
 }

 for (i in 1:10000) {
  randomy[i] = random[i+1]
 }

 hist (random, main="the Histogram of X_t", xlab="X_t")
 plot (random, randomy, main="Scatterplot of (X_t, X_{t+1})",
 xlab="X_t", ylab="X_{t+1}")
}
```

We also have the python version.

```
def sample_uniform(low=0, high=1):
 random = np.zeros(10000)
 randomy = np.zeros(10000)

 m = 2**31 - 1
 a = 7**5
 c = 12345

   # Set the seed using the current system time in microseconds
 import time
 d = int(time.time())

 for i in range (0,10000):
  d = (a * d + c) % m
  random[i] = d / m * (high - low) + low


 for i in range (0,9999):
  randomy[i] = random[i+1]

 import matplotlib.pyplot as plt
 fig1 = plt.figure('Figure1',figsize = (6,4)).add_subplot(111)
 fig1.hist(random)
 fig2 = plt.figure('Figure2',figsize = (6,4)).add_subplot(111)
 fig2.scatter(random,randomy)
 plt.show()
```

Here, the **input** is low and high, which specifies the range of the Uniform random numbers, i.e.$U_t$ = unif[low, high]

### 4.1.3 Exponential Using Inversion Method

We have the Cumulative density function

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(x)\, dx.$$

.

When given a percentage $u = F(x) \in [0, 1]$, we can find a percentile $x = F^{-1}(u)$ such that $x \sim f(x)$.

To draw a sample $X \sim f(x)$ using the random number generator, we can do the following two steps:

- Generate a random number $u \sim \text{unif}[0, 1]$,
- Find out the $x \sim F^{-1}(u)$ by the inverse of CDF function.

For example, we have:

$$X \sim f(x) = \begin{cases} e^{-x} & : x \geq 0 \\ 0 & : x < 0 \end{cases}$$

$$F(x) = \int_{-\infty}^{x} e^{-x} dx = -e^{-x}|_0^x = 1 - e^{-x} = u$$

$$x = -log(1 - u)$$

$$x = -logu$$

R code for generating exponential random numbers using inversion Method:

```
sample_exponential <- function(k=1){
 randomu = rep(0,10000)
 randomx = rep(0,10000)
 r = rep(0,10000)

 m = 2 ** 31 - 1
 a = 7 ** 5
 c = 12345

   # Set the seed using the current system time in microseconds
 d = as.numeric(Sys.time()) * 1000

 for (i in 1:10000) {
     d = (a * d + c) %% m
     randomu[i] = d / m
     randomx[i] = - (log(randomu[i])) / k
 }

 hist (randomx, main="Histogram of X_t", xlab="X_t")

}
```

We also have the python version:

```
def sample_exponential(k=1):
 import math
 randomu = np.zeros(10000)
 randomx = np.zeros(10000)
 r = np.zeros(10000)

 m = 2 ** 31 - 1
 a = 7 ** 5
 c = 12345

   # Set the seed using the current system time in microseconds
 import time
 d = int(time.time())

 for i in range(0,10000):
     d = (a * d + c) % m
     randomu[i] = d / m
     randomx[i] = - (math.log(randomu[i])) / k

 import matplotlib.pyplot as plt
 plt.hist(randomx)
 plt.show()
```

The **input** is k. Which is the coefficient of x, i.e., $e^{-x}$

### 4.1.4 Normal using Polar Method

We want to generate samples $X \sim N(0,1)$ with density function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

People usually perform a polar transformation and generate two independent copies $(X,Y) \sim N(0,1)$ at the same time.

Consider the joint density

$$f(x,y) = f(x) \cdot f(y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}}.$$

Let's parameterize $x = r\cos\theta, y = r\sin\theta$ for $\theta \in [0, 2\pi)$, then the density only depends on $r$.

Since the density $f(x,y) = \frac{\text{number of points in } D_{xy}}{\text{aera of } D_{xy}}$ for a region $D_{xy}$, where the aera is $r \cdot dr \cdot d\theta$. Hence the number of points in region $D_{xy}$ is $f(x,y)rdrd\theta$.

Suppose $R$ and $\Theta$ are random variables. We have $\Theta \sim$ uniform$[0, 2\pi)$, i.e. $\Theta = 2\pi \cdot U$.

$$P(R \in (r, r+\delta r)) = f(x,y)r\ drd\theta$$
$$F(r) = P(R \le r) = \int_0^{2\pi}\int_0^r \frac{1}{2\pi} e^{-\frac{r^2}{2}} r\ drd\theta = 1 - e^{-\frac{r^2}{2}}$$

Let $F(r) = u$, we have $r = \sqrt{-2\log u}$.

To sum up, we can generate two independent samples with two random numbers:

$$\Theta = 2\pi \cdot U_1$$
$$R = \sqrt{-2\log U_2}$$
$$X = R\cos\Theta$$
$$Y = R\sin\Theta.$$

R code for generating normal with polar method:

```
sample_normal <- function(mean=0, var=1){
 d1 = as.numeric(Sys.time()) * 1000
 d2 = as.numeric(Sys.time()) * 500


 randomu1 = rep(0,10000)
 randomu2 = rep(0,10000)
 theta = rep(0,10000)
 t = rep(0,10000)
 r = rep(0,10000)
 x = rep(0,10000)
 y = rep(0,10000)

 m = 2 ** 31 - 1
 a = 7 ** 5
 c = 12345

 for (i in 1:10000) {
     d1 = (a * d1 + c) %% m
     d2 = (a * d2 + c) %% m
     randomu1[i] = d1 / m
     randomu2[i] = d2 / m
     theta[i] = 2 * pi * randomu1[i]
     t[i] = - log(randomu2[i])
     r[i] = (2*var*t[i]) ^ (1/2)
     x[i] = r[i]*cos(theta[i]) + mean
     y[i] = r[i]*sin(theta[i]) + mean
 }
```

```
29
30  plot (x,y,main="Scatterplot of (X_t, Y_t)", xlab="X_t", ylab="Y_t")
31  hist (t, main="Histogram of T = R^2/2",xlab="T")
32
33 }
```

We also have the python version.

```
1  def sample_normal(mean=0, var=1):
2   import math
3   import time
4   d1 = int(time.time() * 500 * math.exp(2))
5   d2 = int(time.time() * 500 * math.exp(1))
6   randomu1 = np.zeros(10000)
7   randomu2 = np.zeros(10000)
8   theta = np.zeros(10000)
9   t = np.zeros(10000)
10  r = np.zeros(10000)
11  x = np.zeros(10000)
12  y = np.zeros(10000)
13
14  m = 2 ** 31 - 1
15  a = 7 ** 5
16  c = 12345
17
18  for i in range (0,10000):
19      d1 = (a * d1 + c) % m
20      d2 = (a * d2 + c) % m
21      randomu1[i] = d1 / m
22      randomu2[i] = d2 / m
23      theta[i] = 2 * math.pi * randomu1[i]
24      t[i] = - math.log(randomu2[i])
25      r[i] = (2 * var * t[i]) ** (1/2)
26      x[i] = r[i]*math.cos(theta[i]) + mean
27      y[i] = r[i]*math.sin(theta[i]) + mean
28
29  import matplotlib.pyplot as plt
30  fig1 = plt.figure('Figure4',figsize = (6,6)).add_subplot(111)
31  fig1.scatter(x,y)
32  fig2 = plt.figure('Figure5',figsize = (6,4)).add_subplot(111)
33  fig2.hist(t)
34  plt.show()
```

The **input** mean and var serve as:

$$f(x,y) = \frac{1}{2\pi} e^{-\frac{(x-mean)^2+(y-mean)^2}{2*var}}.$$

### 4.1.5 Monte Carlo

The Monte Carlo method essentially refers to the techniques which uses computer-generated random number to draw iid (independent and identically distributed) samples from probability distributions.

Suppose we want to calculate an integral

$$I = \int h(x)f(x)\,dx = E(h(X)),$$

where random variable follows a distribution described by a probability density function $X \sim f(x)$.

The Monte Carlo method samples $X_1, X_2, \ldots, X_n \sim f(x)$ and approximates the integral by the sample average

$$\hat{I} = \frac{1}{n} \sum_{i=1}^{n} h(X_i).$$

By the law of large numbers, we know $\hat{I} \to I$ as

$$E(\hat{I}) = I, \quad Var(\hat{I}) = \frac{Var(h(X))}{n} \xrightarrow{n \to \infty} 0.$$

This gets around the curse of dimensionality. Here the random variable $X$ can be high dimensional, but the variance of $\hat{I}$ is just a constant divided by $n$, which has nothing to do with the dimenionality.

The question becomes how do we generate iid samples using computers. It roots to generate uniformly distributed random numbers.

R code for Monte Carlo computation of pi and calculation of the volume of the ball is as below:

```
monte_carlo <- function(d=2){
 ds = vector()
 random = matrix(0,d,1000000)
 n = 0

 m = 2 ** 31 - 1
 a = 7 ** 5
 c = 12345

 for (i in 1:d){
  ds[i] = as.numeric(Sys.time()) * 500 * exp(i)
 }
 for (i in 1:1000000){
  squaresum = 0
  for (j in 1:d) {
   ds[j] = (a * ds[j] + c) %% m
   random[j,i] = ds[j]/m   #j is dimension, i is iteration
   squaresum = squaresum + random[j,i]**2 # sum of squares
  }
  if (squaresum <= 1){
   n = n + 1
  }
 }

 volume = n * (2**d) / 1000000
 print ("volume=")
 print (volume)

 if (d == 2){
  pi1 = 4 * n / 1000000
  print ("pi=")
  print (pi1)
 }
}
```

We also have the python version:

```
def monte_carlo(d=2):
 ds = []
 random = []
 n = 0
```

```
5
6  m = 2 ** 31 - 1
7  a = 7 ** 5
8  c = 12345
9
10 for i in range (1, d+1):
11  ds.append(int(time.time() * 500 * math.exp(i)))
12  random.append(np.zeros(1000000))
13 for i in range (0,1000000):
14  squaresum = 0
15  for j in range (0, d):
16   ds[j] = (a * ds[j] + c) % m
17   random[j][i] = ds[j]/m   #j is dimension, i is iteration
18   squaresum += random[j][i]**2 # sum of squares
19  if squaresum <= 1:
20   n = n + 1
21 volume = n * (2**d) / 1000000
22 print ("volume="+str(volume))
23 if d == 2:
24  pi1 = 4 * n / 1000000
25  print ("pi="+str(pi1))
```

Here, the **input** is the dimension of our ball, to either compute pi or compute the volume.

## 4.2 Homework02-Sampling2

### 4.2.1 Problem Description

- Use the Metropolis algorithm to sample from $N(0,1)$. The proposal distribution at x is $y$ $Uniform[x-c, x+c]$.
- Run 1000 parallel chains with $x0$ $uniform[a,b]$. Show a movie for the change of histogram of $x_t$.
- Experiment with different $[a,b]$ and $c$.
- Use the Gibbs sampler to sample from Bivariate normal with correlation rho.
- Run 1000 parallel chains from the same starting point. Show the movie of scatterplot of $(x_t, y_t)$.
- Run a single chain for T steps, discard the first B steps, and show the scatterplot of the footsteps for the rest of the steps.
- Experiment with different rho.

### 4.2.2 Metropolis

A Markov process is uniquely defined by its transition probabilities $P(x'|x)$, the probability of transitioning from any given state $x$to any other given state $x'$. It has a unique stationary distribution $\pi(x)$ when the following two conditions are met.

Existence of stationary distribution: there must exist a stationary distribution $\pi(x)$. A sufficient but not necessary condition is detailed balance, which requires that each transition $x \to x'$ is reversible: for every pair of states $x, x'$, the probability of being in state $x$ and transitioning to state $x'$ must be equal to the probability of being in state $x'$ and transitioning to state $x$, $\pi(x)P(x'|x) = \pi(x')P(x|x')$. Uniqueness of stationary distribution: the stationary distribution $\pi(x)$ must be unique.

This is guaranteed by ergodicity of the Markov process, which requires that every state must (1) be aperiodic—the system does not return to the same state at fixed intervals; and (2) be positive recurrent—the expected number of steps for returning to the same state is finite. The Metropolis–Hastings algorithm involves designing a Markov process (by constructing transition probabilities) that fulfills the two above conditions, such that its stationary distribution $\pi(x)$ is chosen to be $P(x)$. The derivation of the algorithm starts with the condition of detailed balance:

$P(x'|x)P(x) = P(x|x')P(x')$, which is re-written as

$\frac{P(x'|x)}{P(x|x')} = \frac{P(x')}{P(x)}$. The approach is to separate the transition in two sub-steps; the proposal and the acceptance-rejection. The proposal distribution $g(x'|x)$ is the conditional probability of proposing a state $x'$ given $x$, and the acceptance ratio $A(x', x)$ is the probability to accept the proposed state $x'$. The transition probability can be written as the product of them:

$P(x'|x) = g(x'|x)A(x', x)$. Inserting this relation in the previous equation, we have

$\frac{A(x',x)}{A(x,x')} = \frac{P(x')}{P(x)} \frac{g(x|x')}{g(x'|x)}$. The next step in the derivation is to choose an acceptance ratio that fulfills the condition above. One common choice is the Metropolis choice:

$A(x', x) = \min\left(1, \frac{P(x')}{P(x)} \frac{g(x|x')}{g(x'|x)}\right)$. For this Metropolis acceptance ratio $A$, either $A(x', x) = 1$ or $A(x, x') = 1$ and, either way, the condition is satisfied.

The Metropolis–Hastings algorithm thus consists in the following:

- Pick an initial state $x_0$.
- Set $t = 0$.
- Generate a random candidate state $x'$ according to $g(x'|x_t)$.
- Calculate the acceptance probability $A(x', x_t) = \min\left(1, \frac{P(x')}{P(x_t)} \frac{g(x_t|x')}{g(x'|x_t)}\right)$;.
- generate a uniform random number $u \in [0, 1]$;
- if $u \leq A(x', x_t)$, then accept the new state and set $x_{t+1} = x'$;
- if $u > A(x', x_t)$, the state, and copy the old state forward $x_{t+1} = x_t$.
- Increment: set $t = t + 1$.

Here is the R code for Metropolis:

```
metropolis_simulation <- function(num_chain=1000, chain_length=100,
    mean=0, var=1){
 install.packages("animation")
 library(animation)
 list_a = c(0, -0.1) # Add other value as you want.
 list_b = c(1, 0.1)
 list_c = c(1, 2)
 for (a in list_a){
  for (b in list_b){
   for (c in list_c){
   x0 = sample_uniform(num_chain, a, b)
   # Run Metropolis
   X_normal = sample_normal_chain(x0, c, chain_length, mean, var)
   saveGIF(for(i in 1:chain_length) hist(X_normal[,i], main=NULL, xlab=
    NULL), movie.name = paste('R_Metropolis_a,b,c=',a,b,c,'.gif'),
    interval=0.2)
   }
  }
```

```
 }
}
```

We also have the python version.

```
def metropolis_simulation(num_chain=1000, chain_length=100, mean=0, var
    =1):
 import matplotlib.pyplot as plt
 import matplotlib.animation as animation
 """
 Function 1C: Simulate metropolis with different setting.
 Detail :  Try different setting and output movie of histgrams.
 """

 list_a = [0, -0.1, -1, -10] # Add other value as you want.
 list_b = [1, 0.1, 1, 10]
 list_c = [1, 2]

 for a, b, c in [(a, b, c) for a in list_a for b in list_b for c in
    list_c]:

  # Sample num_chain x0 from uniform[a,b]
  x0 = sample_uniform(num_chain, a, b)

  # Run Metropolis
  X_normal = sample_normal_chain(x0, c, chain_length, mean, var)

  # Define the graph for i-th frame
  def animate(i):
      plt.cla()
      plt.title("Metropolis Sampling (a=%s_b=%s_c=%s) Time: %d" % (a,b,
    c,i))
      plt.hist(X_normal[:,i], bins=30)

  # Define the graph for first frame
  fig = plt.figure()
  hist = plt.hist(X_normal[:,0], bins=30)

  # Define animation class and save
  ani = animation.FuncAnimation(fig, animate, frames=chain_length,
    repeat_delay=3000, repeat=True)
  ani.save('Python_Metropolis_a=%s_b=%s_c=%s.gif' % (a, b, c), writer='
    imagemagick')

  # Plot movie and save
  # Here plot chain_length graphs, each of them is a histogram of
    num_chain point.
  # You may use matplotlib.animation and matplotlib.rc to save graphs
    into gif movies.
```

THis function takes as inputs the number of chians, the length of the chain, and the mean and variance.

### 4.2.3 Gibbs Sampling

Gibbs sampling, in its basic incarnation, is a special case of the Metropolis–Hastings algorithm. The point of Gibbs sampling is that given a multivariate distribution it is simpler to sample from a conditional distribution than to marginalize by integrating over a joint distribution. Suppose we want to obtain $k$. samples of $\mathbf{X} = (x_1, \ldots, x_n)$ from a joint distribution $p(x_1, \ldots, x_n)$. Denote the $i$th sample by $\mathbf{X}^{(i)} = \left(x_1^{(i)}, \ldots, x_n^{(i)}\right)$. We proceed as follows:

- We begin with some initial value $\mathbf{X}^{(i)}$.
- We want the next sample. Call this next sample $\mathbf{X}^{(i+1)}$. Since $\mathbf{X}^{(i+1)} = \left(x_1^{(i+1)}, x_2^{(i+1)}, \ldots, x_n^{(i+1)}\right)$ is a vector, we sample each component of the vector, $x_j^{(i+1)}$, from the distribution of that component conditioned on all other components sampled so far. But there is a catch: we condition on $\mathbf{X}^{(i+1)}$'s components up to $x_{j-1}^{(i+1)}$, and thereafter condition on $\mathbf{X}^{(i)}$'s components, starting from $x_{j+1}^{(i)}$ to $x_n^{(i)}$. To achieve this, we sample the components in order, starting from the first component. More formally, to sample $x_j^{(i+1)}$, we update it according to the distribution specified by $p\left(x_j^{(i+1)} | x_1^{(i+1)}, \ldots, x_{j-1}^{(i+1)}, x_{j+1}^{(i)}, \ldots, x_n^{(i)}\right)$. Note that we use the value that the $(j+1)$th component had in the $i$th sample, not the $(i+1)$th sample.
- Repeat the above step $k$ times.

Here is the R code for Gibbs.

```
gibbs_sample <- function(x0, y0, rho, num_chain=1000, chain_length=100,
    mean=0, var=1){
 X = array(0,dim=c(num_chain, chain_length, 2))
 random_normal = rnorm(num_chain * chain_length * 2, mean = mean, sd =
   var)
 X[,1,1] = x0
 X[,1,2] = y0
 n = 1
 for (i in 1: num_chain){
  for (j in 2: chain_length){
   X[i,j,1] = rho * X[i,j-1,2] + (1-rho**2)**(1/2)*random_normal[n]
   X[i,j,2] = rho * X[i,j,1] + (1-rho**2)**(1/2)*random_normal[n+1]
   n = n + 2
  }
 }
    return(X)
}

# ################################################
# ## Function 2B                                ##
# ################################################

gibbs_simulation <- function(){
 install.packages("animation")
 library(animation)
 list_rho = c(0, 0.5)
 for (rho in list_rho){
  X_gibbs = gibbs_sample(1, 1, rho)
  saveGIF(for(i in 1:100) plot(X_gibbs[,i,1],X_gibbs[,i,2]), movie.name
    = paste('R_Gibbs_1000_Chain_rho=',rho,'.gif'),interval=0.2)
 }
 X_gibbs2 = gibbs_sample(1, 1, 0.5)
 plot (X_gibbs2[,50:100,1], X_gibbs2[,50:100,2], main = "discard the
   first 50 steps, the scatterplot of the footsteps for the rest of
   the steps", xlab = "X[50:100]", ylab="Y[50:100]")

}
```

We also have the python code.

```
def gibbs_sample(x0, y0, rho, num_chain=1000, chain_length=100, mean=0,
    var=1):
```

```
 """
 Function 2A: Bivariate normal with correlation rho
 Detail :  This function return multiple chains by Gibbs sampling
    The input x0, y0, rho, num_chain is a number. This time, we use
    same starting point.
    Return a np.ndarray X with size num_chain * chain_length * 2. In X,
     each row X[i]
    should be a chain and t-th column X[:, t] corresponding to all
    different pair (X_t, Y_t).
 """
 X = np.zeros(((num_chain, chain_length, 2)))
 random_normal = np.random.normal(mean, var, (num_chain, chain_length,
    2))
 X[:,0,0] = x0
 X[:,0,1] = y0
 for i in range (0, num_chain):
  for j in range (1, chain_length):
   X[i][j][0] = rho * X[i][j-1][1] + (1-rho**2)**(1/2)*random_normal[i
    ][j][0]
   X[i][j][1] = rho * X[i][j][0] + (1-rho**2)**(1/2)*random_normal[i][j
    ][1]
 return X


def gibbs_simulation():
 import matplotlib.pyplot as plt
 import matplotlib.animation as animation
 """
 Function 2B: Simulate Gibbs with different rho and plot
 Detail :  Try different setting and output movie of histgrams.
    Discard first 50 steps and output 50~100 steps only.
 """
 list_rho = [0, -1, 1, -0.5, 0.5, -0.25, 0.25, -0.75, 0.75] # Add other
     value as you want.
 for rho in list_rho:
  X_gibbs = gibbs_sample(1, 1, rho)
  fig = plt.figure('Gibbs_Sample_Sccaterplot_rho=%s' % rho, figsize =
    (6,6)).add_subplot(111)
  fig.scatter(X_gibbs[1,50:,0],X_gibbs[1,50:,1])
  plt.savefig ("Python_Gibbs Single Chain Scatterplot(rho=%s).png" % (
    rho))

  def animate(i):
      plt.cla()
      plt.title("Gibbs 1000 Chain (rho=%s) Time: %d" % (rho,i))
      plt.scatter(X_gibbs[:,i,0],X_gibbs[:,i,1])

  # Define the graph for first frame
  fig = plt.figure()
  hist = plt.scatter(X_gibbs[:,0,0],X_gibbs[:,0,1])

  # Define animation class and save
  ani = animation.FuncAnimation(fig, animate, frames=100, repeat_delay
    =3000, repeat=True)
  ani.save('Python_Gibbs_rho=%s.gif' % (rho), writer='imagemagick')
  # Run Gibbs Sampling
```

Here, the inputs $x_0, y_0$ are the initial numbers of the chain. We return X which is the whole chains.

## 4.3 Homework03-Sweep

### 4.3.1 Problem Description

- Write R code for the Sweep operator. The input K means we sweep for k in 1 : K.
- Write Rcpp code for the for-loops in the Sweep operator. A tutorial will be given in the next lecture.
- Write R code for linear model, with X and Y being the inputs, using the Sweep operator as the engine. Output beta. Compare the results of your R code with the built-in lm (Y X) function.

### 4.3.2 Sweep Operator

Where $A = (a_{ij})$ is a $n \times n$ square matrix, the SWEEP operator on $A$ and $k$ is defined as $\text{SWEEP}(A, k) = \tilde{A}$, where

$\tilde{a}_{ij} = a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}}$, for all $i, j \neq k$,

$\tilde{a}_{kj} = \frac{a_{kj}}{a_{kk}}$, for all $j \neq k$,

$\tilde{a}_{ik} = \frac{a_{ik}}{a_{kk}}$, for all $i \neq k$,

$\tilde{a}_{kk} = -\frac{1}{a_{kk}}$.

We can apply it via matrix form. Suppose we split a $n \times n$ matrix $A$ into four blocks

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where $A_{11}$ is a $m \times m$ matrix.

Using SWEEP operator repeatedly with $k = 1, 2, \ldots, m$, we get

$$\text{SWEEP}(A, 1 \cdots m) = \begin{bmatrix} -A_{11}^{-1} & A_{11}^{-1}A_{12} \\ A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}.$$

As a special case, $\text{SWEEP}(A, 1 \cdots n) = -A^{-1}$.

Here's the code for sweep operator.

```
mySweep <- function(A, k){
  # Perform a SWEEP operation on A with the pivot element A[k,k].
  #
  # A: a square matrix.
  # m: the pivot element is A[k, k].
  # Returns a swept matrix B (which is k by k).
  ############################################
  ## FILL IN THE BODY OF THIS FUNCTION BELOW ##
  ############################################
    n <- dim(A)[1]
    for (m in 1:k){
      for (i in 1:n){
        for (j in 1:n){
          if (i!=m & j!=m){
            A[i,j] = A[i,j] - (A[i,m]%*%A[m,j]/A[m,m])
          }
        }
      }
```

```
      for (i in 1:n){
        if (i!=m) {
          A[m,i] = A[m,i]/A[m,m]
          A[i,m] = A[i,m]/A[m,m]
        }
      }
      A[m,m] = - 1/A[m,m]
    }
  ## The function outputs the matrix B
  return(A)
}
```

The input is A, which is the original input matrix, and k, which is the pivot element from which we do the sweep. The output is the transformed matrix A.

We also have a cpp version.

```
NumericMatrix mySweepC(const NumericMatrix A, int k){
  /*
  Perform a SWEEP operation on A with the pivot element A[k,k].

  A: a square matrix (mat).
  m: the pivot element is A[k, k].
  Returns a swept matrix B (which is k by k).
  Note the "const" in front of mat A; this is so you
  don't accidentally change A inside your code.
  ##############################################
  ## FILL IN THE BODY OF THIS FUNCTION BELOW ##
  ##############################################
  */
  NumericMatrix B = clone(A);
  int n = B.nrow();
  for (int m = 0; m < k; m++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if ((i!=m) & (j!=m)){
          B(i,j) = B(i,j) - (B(i,m)*B(m,j)/B(m,m));
        }
      }
    }
    for (int i = 0; i < n; i++){
      if (i!=m ){
        B(m,i) = B(m,i)/B(m,m);
        B(i,m) = B(i,m)/B(m,m);
      }
    }
    B(m,m) = -(1/B(m,m));
  }
  // Return swept matrix B
  return(B);

}
```

### 4.3.3 Solving Linear Regression via SWEEP Operator

In linear regression $Y = X^T\beta + \epsilon$, where $\epsilon_i \sim N(0, \sigma^2)$.

When solving the least square estimator

$$\hat{\beta} = \arg\min_{\beta} |Y - X^T\beta|^2 = (X^TX)^{-1}X^TY,$$

we construct a matrix $Z = \begin{bmatrix} X & Y \end{bmatrix}$,

$$A = Z^T Z = \begin{bmatrix} X^T X & X^T Y \\ Y^T X & Y^T Y \end{bmatrix}.$$

Using SWEEP operator repeatedly with $k = 1, 2, \ldots, p$,

$$\text{SWEEP}(A, 1 \cdots p) = \begin{bmatrix} -(X^T X)^{-1} & (X^T X)^{-1} X^T Y \\ Y^T X (X^T X)^{-1} & Y^T Y - Y^T X (X^T X)^{-1} X^T Y \end{bmatrix} = \begin{bmatrix} -\frac{\text{Var}[\hat{\beta}]}{\sigma^2} & \hat{\beta} \\ \hat{\beta}^T & \text{RSS} \end{bmatrix},$$

where $\text{RSS} = Y^T Y - \hat{Y}^T \hat{Y}$ is the residual sum of squares.

Here's code for solving linear regression via SWEEP.

```
myLinearRegression <- function(X, Y){

  # Find the regression coefficient estimates beta_hat
  # corresponding to the model Y = X * beta + epsilon
  # Your code must use the sweep operator you coded above.
  # Note: we do not know what beta is. We are only
  # given a matrix X and a vector Y and we must come
  # up with an estimate beta_hat.
  #
  # X: an 'n row' by 'p column' matrix of input variables.
  # Y: an n-dimensional vector of responses
  ###############################################
  ## FILL IN THE BODY OF THIS FUNCTION BELOW ##
  ###############################################
    library(Rcpp)
    sourceCpp("HW3_Sweep.cpp")
    append = matrix(1, nrow = nrow(X))
    X = cbind(append,X)
    A = rbind (cbind (t(X)%*%X, t(X)%*%Y), cbind(t(Y)%*%X, t(Y)%*%Y))
    n <- nrow(X)
    p <- ncol(X)
    B =  mySweep(A, p)
    #B =  mySweepC(A, p)
    c = c(1:p)
    beta_hat = B[c,p+1]
  ## Function returns the (p+1)-dimensional vector
  ## beta_hat of regression coefficient estimates
  return(beta_hat)
}
```

The input is X and Y, which is the data and result of the linear regression. The function returns the estimated $\beta$

## 4.4 Homework04-QR

### 4.4.1 Problem Description

- Write R code for QR decomposition. The input is a matrix A. The outputs are Q and R, with R being an upper triangular matrix.
- Write R code for linear model based on QR. The inputs are $(X, Y)$, The outputs are beta-hat and $|e|^2$. Please do not use the built-in matrix inverse function.

### 4.4.2 QR Decomposition

QR decomposition is to decompose a matrix $A$ into a product $A = QR$ of an orthogonal matrix $Q$ and a upper triangular matrix $R$.

Let $Q = (q_1, q_2, \ldots, q_n)$ be an orthogonal matrix, $Q^TQ = QQ^T = I$

* For each vector $q_i$, $\langle q_i, q_i \rangle = 1$. i.e $|q_i| = 1$. * For any two of the vectors $q_i$ and $q_j$, the inner product $\langle q_i, q_j \rangle = 0$, which means they are orthogonal.

$Q$ consists of $n$ orthogonal basis. Given a vector $v$ in original space, the coordinates in the new space defined by $Q$ is

$$u = Q^T v.$$

To get a QR decomposition, we can construct householder reflections repeatedly. Given a matrix, as the first step, we want to find a orthogonal transformation $H_1$ such that only the first element in the first column is non-zero after the transformation.

$$\begin{bmatrix} x_{11} & x_{12} & \ldots & y_1 \\ x_{21} & x_{22} & \ldots & y_2 \\ \ldots & \ldots & \ldots & \ldots \\ x_{n1} & x_{n2} & \ldots & y_n \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} x_{11}^* & x_{12}^* & \ldots & y_1^* \\ 0 & x_{22}^* & \ldots & y_2^* \\ \ldots & \ldots & \ldots & \ldots \\ 0 & x_{n2}^* & \ldots & y_n^* \end{bmatrix}$$

Note that since the orthogonal transformation preserves the length of vectors, we know

$$|x_1^*| = |x_1| = \sqrt{(x_{11}^2 + x_{12}^2 + \ldots + x_{1n}^2)},$$

which means the value of $x_{11}$ is determined

$$x_{11}^* = \pm |x_1|.$$

In actual the implementation, the sign is chosen as the opposite of $x_{11}$ for numerical stability.

To find a transformation $H$ which can rotate vector $x_1$ to $x_1^*$, one simple way is to construct an isosceles triangle where $x_1^*$ is just a reflection of $x_1$.

$$x_1^* = x_1 - 2\langle x_1, u \rangle u = x_1 - 2uu^T x_1 = H_1 x_1,$$

where

$$u = \frac{x_1 \pm x_1^*}{|x_1 \pm x_1^*|}, \quad H_1 = I - 2uu^T.$$

To transform the original matrix $A_{n \times p}$ into a upper triganle matrix, we can repeat this procedure $p$ times, the orthogonal matrix $H = H_p \ldots H_2 H_1$. Hence, $HA = R$. let $Q = H^T$, we obtain the QR decomposition of $A$

$$A = QR.$$

Here is my code for QR decomposition.

```
myQR <- function(A){

  ## Perform QR decomposition on the matrix A
  ## Input:
  ## A, an n x m matrix

  ########################
  ## FILL IN CODE BELOW ##
  ########################

  n = dim(A)[1]
  m = dim(A)[2]

```

```
  R = A
  Q = diag(n)

  for (k in 1 : (m-1)){
    x = matrix(0,n,1)
    x[k:n,1] = R[k:n, k]
    v = x
    v[k] = x[k] + sign(x[k,1])*norm(x, type = "F")
    s = norm(v, "F")

    if(s != 0){
      u = v/s
      R = R - 2*(u %*% (t(u) %*% R))
      Q = Q - 2*(u %*% (t(u) %*% Q))
    }
  }
  ## Function should output a list with Q.transpose and R
  ## Q is an orthogonal n x n matrix
  ## R is an upper triangular n x m matrix
  ## Q and R satisfy the equation: A = Q %*% R
  return(list("Q" = t(Q), "R" = R))
}
```

The input is the matrix for decomposition. We return the Q and R after decomposition.

### 4.4.3 QR Decomposition for Linear Regression

Consider rotate the matrix $(XY)$ by some orthogonal matrix $Q$,

$$\begin{bmatrix} X & Y \end{bmatrix} \xrightarrow{Q^T} \begin{bmatrix} R & Y^* \end{bmatrix} = \begin{bmatrix} R_1 & Y_1^* \\ 0 & Y_2^* \end{bmatrix},$$

where $R_1$ is a upper triangular matrix.

To solve the least square,

$$\min_\beta |Y^* - R\beta|^2 = \min_\beta \left( |Y_1^* - R_1\beta|^2 + |Y_2^*|^2 \right).$$

So the solution $\hat{\beta} = R_1^{-1} Y_1^*$ and RSS $= |Y_2^*|^2$.

To solve $\hat{\beta}$ is essentially easy, since $R_1$ is an upper triangular matrix. We can solve the elements of $\hat{\beta}$ in reverse order $\hat{\beta}_p, \hat{\beta}_{p-1}, \ldots, \hat{\beta}_1$. It is numerically stable and efficient.

Here is the code.

```
myLinearRegression <- function(X, Y){

  ## Perform the linear regression of Y on X
  ## Input:
  ## X is an n x p matrix of explanatory variables
  ## Y is an n dimensional vector of responses
  ## Do NOT simulate data in this function. n and p
  ## should be determined by X.
  ## Use myQR inside of this function

  ##########################
  ## FILL IN CODE BELOW ##
  ##########################

  n = dim(X)[1]
```

```
  p = dim(X)[2]

  Z = cbind(rep(1,n), X, Y)

  R = myQR(Z)$R

  R1 = R[1:(p + 1), 1:(p + 1)]
  Y1 = R[1:(p + 1), p + 2]
  error_list = matrix(0, (n - p - 1), 1)
  error_list[,1] = R[(p + 2):n, p + 2]
  error = norm(error_list, type = "F")

  beta_hat = solve(R1) %*% Y1

  ## Function returns the 1 x (p + 1) vector beta_ls,
  ## the least squares solution vector
  return(list(beta_hat=beta_hat, error=error))
}
```

The inputs are X, an n x p matrix of explanatory variables; Y, an n dimensional vector of response. We return the estimated $\beta$ and the error.

## 4.5 Homework05-PCA,Eigen

### 4.5.1 Problem Description

Function 1/ 2: Fill the QR decomposition and linear regression you've done last week.

Function 3: Implement eigen decomposition based on QR.

Function 4: Implement PCA based on eigen decomposition.

### 4.5.2 PCA&Eigen

Suppose $X = (X_1, X_2, \cdots, X_n)$ is a p-dimentional random variable, its variance (in high dimension form) is the covariance matrix

$$\text{var}[X] = \Sigma = \text{E}\left[(X - \text{E}(X))(X - \text{E}(X)^T)\right],$$

where element on the diagonal are $\text{var}(X_i)$, other elements are covariance $\text{cov}(X_i, X_j)$.

Suppose we have a transformed version $Z = AX$, its variance

$$\begin{aligned}
\text{var}(Z) =& \text{E}\left[(Z - \text{E}(Z))(Z - \text{E}(Z)^T)\right] && (4.1)\\
=& \text{E}\left[(AX - A\mu_X)(AX - A\mu_X)^T\right] && (4.2)\\
=& \text{E}\left[A(X - \mu_X)(X - \mu_X)^T A^T\right] = A\text{var}(X)A^T. && (4.3)
\end{aligned}$$

The essense of PCA is to find a transformation of $X$ such that in a new corridinate system $Q$, the covariance (and therefore correlation) become zero between any two dimensions.

Let $X = QZ$, which says $Z$ is the coordinate of $X$ in the space of $Q$. For $Z = Q^T X$, we want to find the $Q$ so that its covariance matrix can be writen as a diagonal matrix $\Lambda$, so that the covariance and correlations between any two dimensions are removed

in this new system.

$$\text{var}(Z) = Q^T \Sigma Q = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_p \end{bmatrix} = \Lambda,$$

where $\Sigma$ is the covariance matrix of $X$.

Reorganize we have $\Sigma = Q\Lambda Q^T$, therefore $\Sigma Q = Q\Lambda$. It is clear that column vectors in $Q$ and diagonal elementsin $\Lambda$ are eigenvectors and eigenvalues of $\Sigma$. Hence the eigen decomposition of the covariance matrix $\Sigma$ of $X$ gives us the priciple directions $q_i$ and the variability $\lambda_i$ of data along the corresponding direction.

If top $k$ basis in $Q$ are selected according to $\lambda_i$ from biggest to smallest, we captured the top $k$ pricinple directions of data along which the variability are most significant. The choice of $k$ in practice can be choosen by keeping the reconstruction error of data in an acceptable range.

Here we use the power method. In Vector form:

For a vector $v$, let $u$ be the transformed coordinate in system $Q$, i.e. $v = Qu$.

$$\Sigma v = Q\Lambda Q^T Qu = Q \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_p \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_p \end{bmatrix} = Q \begin{bmatrix} \lambda_1 u_1 \\ \lambda_2 u_2 \\ \vdots \\ \lambda_p u_p \end{bmatrix}.$$

This says the vector $\Sigma v$ becomes $(\lambda_1 u_1, \lambda_2 u_2, \cdots, \lambda_p u_p)^T$ in system $Q$.

If we repeat this process $n$ times,

$$v \xrightarrow{\Sigma^n} (\lambda_1^n u_1, \lambda_2^n u_2, \cdots, \lambda_p^n u_p)^T.$$

Suppose $\lambda_1$ has the greatest magnitude, this procedure will converge to $v = (1, 0, \cdots, 0)$ in the space of $Q$, i.e. $v = q_1$.

* Repeat the following steps: - Compute normalized vector $\tilde{v} = \frac{v}{|v|}$. - Update $v = \Sigma\tilde{v}$.

To get $q_2$ using this method, we choose a vector $v \perp q_1$ that is perpendicular to $q_1$. To get $q_3$, choose $v$ to be perpendicular to both $q_1$ and $q_2$, i.e. $v \perp q_1$ and $v \perp q_2$. So on and so forth, we can get all the vectors in $Q$.

And for matrix form:

Suppose initially we have a random matrix $V = (v_1, v_2, \cdots, v_p)$, whose orthogonalized version is $\tilde{V} = (\tilde{v}_1, \tilde{v}_2, \cdots, \tilde{v}_p)$ where each of these column vectors is a unit vector and every two of them are orthogonal to each other.

Power method is a iterative algorithm to compute eigen decomposition.

* Repeat the following steps: - Compute $\tilde{V}$, the orthogonalized $V$. - Update $V = \Sigma\tilde{V}$.

In the following implementation, we use QR decomposition to perform orthogonalization.

Here is my code for eigen:

```
myEigen_QR <- function(A, numIter = 1000) {

  ## Perform PCA on matrix A using your QR function, myQR or Rcpp myQRC
    .
  ## Input:
```

```
  ## A: Square matrix
  ## numIter: Number of iterations

  ########################
  ## FILL IN CODE BELOW ##
  ########################
  r = dim(A)[1]
  c = dim(A)[2]

  V = matrix(runif(r*r), nrow = r)

  for (i in 1 : numIter){
    op = myQR(V)
    Q = op$Q
    V  = A %*% Q
  }

  op = myQR(V)

  Q = op$Q
  R = op$R
  ## Function should output a list with D and V
  ## D is a vector of eigenvalues of A
  ## V is the matrix of eigenvectors of A (in the
  ## same order as the eigenvalues in D.)

  return(list("D" = diag(R), "V" = Q))
}
```

The function takes as input the matrix for PCA and the number of iterations. The function returns the eigenvalues and eigenvectors of A.

Here is the code for PCA based on eigen:

```
myPCA <- function(X) {

  ## Perform PCA on matrix A using your eigen decomposition.
  ## Input:
  ## X: Input Matrix with dimension n * p

  n1=nrow(X)
  p1=ncol(X)

  X1 = X
  for(i in 1:p1)
  {
    X1[,i]=X[,i]-mean(X[,i])
  }
  epsilon = t(X1) %*% X1 / (n1 - 1)
  pca_answer <- myEigen_QR (epsilon)
  Q = pca_answer$V
  Z = X %*% Q

  ## Output :
  ## Q : basis matrix, p * p which is the basis system.
  ## Z : data matri x with dimension n * p based on the basis Q.
  ## It should match X = Z %*% Q.T. Please follow lecture notes.

  return(list("Q" = Q, "Z" = Z))
}
```

## 4.6 Homework06&07-Logistic, Boosting

### 4.6.1 Problem Description

- Write R code for logistic regression, based on QR code for linear regression. Return beta and compare with built in R function. Take a screenshot of your comparison.
- Write R code for extreme gradient boosting, using one layer tree as base function.
- Write R code for adaboost, using one layer tree as base classifier.

### 4.6.2 Logistic Regression

Logistic regression is an importatn model for classification. Consider a dataset with $n$ samples, where $x_i^T = (x_{i1}, \cdots, x_{ip})$ is the features vector, $y_i \in \{0, 1\}$ is the class label for $i = 1, 2, \ldots, n$.

We assume the label follows a Bernoulli distribution $y_i \sim \text{Bernoulli}(p_i)$, $p(y_i = 1) = p_i$. And we assume the logarithm of probability odd-ratio is a linear function

$$\log \frac{p_i}{1 - p_i} = x_i^T \beta.$$

Let $\eta_i = x_i^T \beta$ be the score, then the probability $p_i$ is simply a mapping from $\mathbb{R}$ to $(0, 1)$

$$p_i = f(\eta_i) = \frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} = \frac{1}{1 + e^{-x_i^T \beta}},$$

where the function $f(\eta_i) = \dfrac{1}{1 + e^{-\eta_i}}$ is the sigmoid function.

To estimate $\beta$, we find a maximum likelihood estimator. Consider the likelihood function

$$L(\beta) = \prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1 - y_i} = \prod_{i=1}^{n} \frac{e^{y_i x_i^T \beta}}{1 + e^{x_i^T \beta}},$$

we take the logarithm

$$\ell(\beta) = \log L(\beta) = \sum_{i=1}^{n} y_i x_i^T \beta - \log(1 + e^{x_i^T \beta}).$$

There is no close form solution for $\beta$. To find the maximum, let's first take derivative

$$\ell'(\beta) = \sum_{i=1}^{n} y_i x_i - \frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} x_i = \sum_{i=1}^{n} (y_i - p_i) x_i.$$

And we use gradient ascent to iteratively update $\beta$ along the gradient direction.

$$\beta_0 = 0,$$
$$\beta_{t+1} = \beta_t + \gamma \sum_{i=1}^{n} (y_i - p_i) x_i,$$

A more efficient ways is to update $\beta$ using Newton-Raphson method.

As the first case, to solve the equation $h(x) = 0$, we take the first order Taylor expansion

$$h(x) \doteq h(x_t) + h'(x_t)(x - x_t)$$
$$x_{t+1} = x_t - \frac{h(x_t)}{h'(x_t)}.$$

Each iteration, it solves a linear surrogate function to the original one. where $\gamma$ is usually called learning rate. Intuitively, the idea is to accumulatively memorizes the training examples on which the learner is not doing well, i.e. $y_i \neq p_i$. It is very much like getting your homework graded and learning from the mistakes.

In this case, we actually want to solve $f'(x) = 0$. Using the equation above, use Newton-Raphson method, we have

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}.$$

It essentially maximize a quadratic approximation of the original function at $x_t$.

$$f(x) \doteq f(x_t) + f'(x_t)(x - x_t) + \frac{1}{2}f''(x_t)(x - x_t)^2.$$

When the variable is a vector $x = (x_1, x_2, \cdots, x_n)^T$, to solve $h(x) = 0$,

$$x_{t+1} = x_t - h'(x_t)^{-1}h(x_t).$$

If function $h(x) = (h_1(x), h_2(x), \cdots, h_n(x))^T$ is n-dimensional, the derivative is a $n \times n$ matrix:

$$h'(x_t) = \left(\frac{\partial h_i(x)}{\partial x_j}\right)_{i,j}.$$

In the case of maximization, suppose $f(x) = f(x_1, x_2, \ldots, x_n)$,

$$f'(x) = \left(\frac{\partial f}{\partial x_i}\right)_i, \quad f''(x) = \left(\frac{\partial f}{\partial x_i \partial x_j}\right)_{i,j},$$

$f''(x_t)$ is called the Hessian matrix, we have

$$x_{t+1} = x_t - f''(x_t)^{-1}f'(x_t).$$

Using Newton-Raphson method, to maximize $\ell(\beta)$, therefore we take second derivative of the log likelihood function,

$$\ell''(\beta) = -\sum_{i=1}^{n} p_i(1 - p_i)x_i x_i^T.$$

We then can update $\beta$ with a larger step-size:

$$\beta_{t+1} = \beta_t + \ell''(\beta)^{-1}\ell'(\beta).$$

Let $w_i = p_i(1 - p_i)$, we can rewrite the update equation as

$$\begin{aligned}\beta_{t+1} &= \beta_t + \left[\sum_{i=1}^{n} p_i(1-p_i)x_i x_i^T\right]^{-1}(y_i - p_i)x_i \\ &= \left(\sum_{i=1}^{n} w_i x_i x_i^T\right)^{-1}\left[\sum_{i=1}^{n} w_i x_i x_i^T \beta_t + (y_i - p_i)x_i\right] \\ &= \left(\sum_{i=1}^{n} w_i x_i x_i^T\right)^{-1}\left[\sum_{i=1}^{n} w_i x_i \left(x_i^T\beta_t + \frac{y_i - p_i}{w_i}\right)\right].\end{aligned}$$

Let $z_i = x_i^T\beta_t + \frac{y_i - p_i}{w_i}, \tilde{x}_i = \frac{x_i}{\sqrt{w_i}}, \tilde{z}_i = \frac{z_i}{\sqrt{w_i}}$, we can further rewrite the equation above as follows.

$$\begin{aligned}\beta_{t+1} &= \left(\sum_{i=1}^{n} w_i x_i x_i^T\right)^{-1}\left(\sum_{i=1}^{n} w_i x_i z_i\right) \\ &= \left(\sum_{i=1}^{n} \tilde{x}_i \tilde{x}_i{}^T\right)^{-1}\left(\sum_{i=1}^{n} \tilde{x}_i \tilde{z}_i\right).\end{aligned}$$

This means we can actually view $\beta_{t+1}$ as the solution to the weighted least squares problem

$$\texttt{WLS}(z_i, x_i, w_i),$$

or the ordinary least squares with

$$\tilde{z}_i = \tilde{x}_i{}^T\beta_{t+1}.$$

Using Newton-Raphson method to solve logistic regression can be summarized as follows.

* Start with $\beta_t$, * $\eta_i = x_i^T\beta_t$. * $p_i = \text{sigmoid}(\eta_i)$, * $w_i = p_i(1-p_i)$, * $z_i = \eta_i + \frac{y_i - p_i}{w_i}$, * $\tilde{x}_i = \frac{x_i}{\sqrt{w_i}}, \tilde{z}_i = \frac{z_i}{\sqrt{w_i}}$, * $\beta_{t+1} = \texttt{solve}(\tilde{z}_i, \tilde{x}_i)$.

Here is the R code for Logistic Regression.

```
myLogisticSolution <- function(X, Y){

  ########################
  ## FILL IN CODE BELOW ##
  ########################
  r= dim(X)[1]
  c= dim(X)[2]

  Xcopy=X
  Ycopy=Y

  beta <- rep(0,c)
  epsilon = 10^(-6)

  err=10
  while(err>epsilon){
    eta <- Xcopy %*% beta
    p <- expit(eta)
    w= p *(1-p)
```

```
    z=eta+(Ycopy-p)/w
    sw=sqrt(w)
    mw=rep(sw,c)

    x_new=mw*Xcopy
    y_new=sw*z

    new_beta <- myLM(x_new, y_new)
    err=sum(abs(new_beta-beta))
    beta=new_beta
  }

  return(beta)
}
```

In the code, MyLM is the linear regression. We take as input the data X and result Y.

### 4.6.3 Adaboost

Let's first examine AdaBoost from the perspective of loss function. Recall that we have seen loss functions for logisitic regression.

$$\ell_{logistic} = \sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T \beta}), \tag{4.4}$$

$$\ell_{hinge} = \sum_{i=1}^{n} \max(0, 1 - y_i x_i^T \beta). \tag{4.5}$$

In AdaBoost, the loss function is in the exponential form:

$$\ell_{AdaBoost} = \sum_{i=1}^{n} e^{-y_i \sum_{k=1}^{m} \beta_k h_k(x_i)},$$

where $h_k(x_i) \in \{+, -\}$ are weak classifiers (aka. weak learners or base learners).

The following code segment plots these loss functions.

Examine the loss function

$$\ell_{AdaBoost} = \sum_{i=1}^{n} e^{-y_i \sum_{k=1}^{m} \beta_k h_k(x_i)}.$$

Intuitively, the set of $h_k()$'s forms a committee, each member has a voting weight $\beta_k$. The classification decision is made based on the voting of committee members, sign $(\sum_k \beta_k h_k(x_i))$.

When training a AdaBoost classifier, we usually use a sequential method where we select committee one after another so that training examples are getting separated gradually.

Suppose a committee with $k$ classifiers, we want to add a new member $h_{\text{new}}$. The votes are

$$\text{current committee:} \quad \sum_{i=1}^{m} \beta_k h_k(x_i), \tag{4.6}$$

$$\text{add a new member:} \quad \sum_{i=1}^{m} \beta_k h_k(x_i) + \beta_{\text{new}} h_{\text{new}}(x_i). \tag{4.7}$$

After adding a member, the loss function becomes:

$$\ell_{AdaBoost} = \sum_{i=1}^{n} e^{-y_i(\sum_{k=1}^{m} \beta_k h_k(x_i) + \beta_{\text{new}} h_{\text{new}}(x_i))},$$

Take derivative

$$\frac{\partial \ell}{\partial \beta_{\text{new}}} = \sum_{i=1}^{n} e^{-y_i(\sum_{k=1}^{m} \beta_k h_k(x_i) + \beta_{\text{new}} h_{\text{new}}(x_i))} \cdot \left(-y_i h_{\text{new}}(x_i)\right).$$

At the point we haven't choose a new member, $\beta_{\text{new}} = 0$, the above gradient can be written as

$$\frac{\partial \ell}{\partial \beta_{\text{new}}}\Big|_{\beta_{\text{new}}=0} = \sum_{i=1}^{n} e^{-y_i(\sum_{k=1}^{m} \beta_k h_k(x_i))} \cdot \left(-y_i h_{\text{new}}(x_i)\right) = -\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i),$$

where

$$w_i = e^{-y_i(\sum_{k=1}^{m} \beta_k h_k(x_i))}.$$

Here we want to choose the weak classifier $h_{\text{new}}$ which gives the maximum drop in loss, i.e. we want to maximize $\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i)$.

Intuitively, we can consider the original training examples being weighted by $w_i$'s (voting from current committee). The correctly classified examples receive lower weights, whereas the wrongly classified examples get higher weights. Then the ideal $h_{\text{new}}$ is the one which can do well in this weighted examples among all candidates. This new weak classifier therefore comes to resolve the issues that the current committee cannot handle well.

To find $\beta_{\text{new}}$, we set the derivative to 0,

$$\frac{\partial \ell}{\partial \beta_{\text{new}}} = 0, \tag{4.8}$$

$$\sum_{i=1}^{n} w_i e^{-y_i \beta_{\text{new}} h_{\text{new}}(x_i)} \cdot y_i h_{\text{new}}(x_i) = 0, \tag{4.9}$$

$$\sum_{i \in \text{correct}} w_i e^{-\beta_{\text{new}}} = \sum_{i \in \text{wrong}} w_i e^{\beta_{\text{new}}}, \tag{4.10}$$

$$\sum_{i \in \text{correct}} w_i = \sum_{i \in \text{wrong}} w_i e^{2\beta_{\text{new}}}. \tag{4.11}$$

If we define error rate as $\epsilon = \dfrac{\sum_{i \in \text{wrong}} w_i}{\sum_i w_i}$, $\beta_{\text{new}}$ can be written as

$$\beta_{\text{new}} = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}.$$

It says that the weight is determined by how much error $h_{\text{new}}$ made on the weighted data. This also explains the name of AdaBoost, where "Ada" means adpative.

Here is the R code for Adaboost:

```
myAdaboost <- function(x1, x2, y) {
  num_samples = length(x1)
  num_training <- round(num_samples*0.8)
  indices = sample(1:num_samples, size = num_samples)
  training_idx <- indices[1:num_training]
  testing_idx <- indices[-(1:num_training)]

  x1_train = x1[training_idx]
  x2_train = x2[training_idx]
  y_train = y[training_idx]
  y_test = y[testing_idx]

  num_iterations=100

  w = rep(1/num_samples, num_samples)
  s=rep(0,num_samples)

  train_error=rep(0,num_iterations)
  test_error=rep(0,num_iterations)

  for (iter in 1:num_iterations){
    wtrain = w[training_idx]
    direction = 0 #column or row
    cutpoint = 0
    t=100 #number of slices
    x1list=rep(0,t-1)
    x2list=rep(0,t-1)

    for (i in 1:(t-1)) {
      x1list[i]=sum(wtrain*(y_train != as.integer(x1_train<=(i/t))))
      x2list[i]=sum(wtrain*(y_train != as.integer(x2_train<=(i/t))))
    }

    num=which.min(c(x1list,x2list))
    error=0
    h=rep(0,num_samples)

    if(num<=(t-1)){
      cutpoint=num*(1/t)
      error = x1list[num]
      h = as.integer(x1<=(cutpoint))*2-1
    }
    else{
      cutpoint=(num-(t-1))*(1/t)
      error = x2list[num-(t-1)]
      h=as.integer(x2<=(cutpoint))*2-1
    }

    beta=1/2*log((1-error)/error)

    s=s+beta*h
    yi=y*2-1
    w=exp(-yi*s)/sum(exp(-yi*s))

    final = (s > 0)

    train_error[iter]=sum(y_train!=final[training_idx])/num_training
```

```
    test_error[iter]=sum(y_test!=final[testing_idx])/(num_samples-
    num_training)

  }

  plot(train_error,type = "o",col = "red",ylim = c(0,0.3),
       xlab = "epoch",ylab = "error rate",main="Adaboost training error
     and testing error")
  lines(test_error,type = "o",col = "green")
  legend(60, 0.2, legend=c("Training", "Testing"),
       col=c("red", "green"), lty=1:2, cex=0.8)

  plot(x1,x2,col = ifelse(final==1, "Red", ifelse(final==0, "#0073C2FF
    ", "Black")),pch = 20,main="classification result of Adaboost")

}
```

### 4.6.4 XGBoost

We have $s_i = F(x_i) = F_m(x_i) + h_{m+1}(x_i)$

$$h_{m+1} = argmin \sum_{1}^{n} w_i (r_i - h(x_i))^2$$

$$L(R_1, R_2) = min_{C_1,C_2} \sum_{i \in R_1} w_i (r_i - \hat{c_1})^2 + \sum_{i \in R_2} w_i (r_i - \hat{c_2})^2$$

We take the derivative, and get:

$$\hat{c_1} = \frac{\sum_{R_1} w_i r_i}{\sum_{R_1} w_{ii}}$$

Here is the R code for XGBoost:

```
myXGBoost <- function(x1, x2, y) {
  num_samples = length(x1)
  num_training <- round(num_samples*0.8)
  indices = sample(1:num_samples, size = num_samples)
  training_idx <- indices[1:num_training]
  testing_idx <- indices[-(1:num_training)]

  x1_train = x1[training_idx]
  x2_train = x2[training_idx]
  y_train = y[training_idx]
  y_test = y[testing_idx]

  num_iterations=50
  w = rep(0, num_samples)
  s=rep(0,num_samples)
  train_error=rep(0,num_iterations)
  test_error=rep(0,num_iterations)

  for (iter in 1:num_iterations){
    p=exp(s)/(1+exp(s))
    w=p*(1-p)
    r=(y-p)/w

    w_train = w[training_idx]
    r_train = r[training_idx]

    t=100
```

```
28    x1list_c1=rep(0,t-1)
29    x1list_c2=rep(0,t-1)
30
31    x2list_c1=rep(0,t-1)
32    x2list_c2=rep(0,t-1)
33
34    R_columns=rep(0,t-1)
35    R_rows=rep(0,t-1)
36
37    for (i in 1:(t-1)){
38      x1list_c1[i]=sum(w_train*r_train*(x1_train<=(i*(1/t))))/sum(
      w_train*(x1_train<=(i*(1/t))))
39      x1list_c2[i]=sum(w_train*r_train*(x1_train>(i*(1/t))))/sum(
      w_train*(x1_train>(i*(1/t))))
40
41      x2list_c1[i]=sum(w_train*r_train*(x2_train<=(i*(1/t))))/sum(
      w_train*(x2_train<=(i*(1/t))))
42      x2list_c2[i]=sum(w_train*r_train*(x2_train>(i*(1/t))))/sum(
      w_train*(x2_train>(i*(1/t))))
43
44      x1list_c=x1list_c1[i]*(x1_train<=(i*(1/t)))+x1list_c2[i]*(
      x1_train>(i*(1/t)))
45      R_columns[i]=sum(w_train*((r_train-x1list_c)^2))
46
47      x2list_c=x2list_c1[i]*(x2_train<=(i*(1/t)))+x2list_c2[i]*(
      x2_train>(i*(1/t)))
48      R_rows[i]=sum(w_train*((r_train-x2list_c)^2))
49    }
50
51    #compare R and find the best cut
52
53    num=which.min(c(R_columns,R_rows))
54    if(num<=(t-1)){
55      cutpoint=num*(1/t)
56      direction = 0
57      c = x1list_c1[num]*(x1<=(cutpoint))+x1list_c2[num]*(x1>(cutpoint)
      )
58      s = s + c
59    }
60    else{
61      cutpoint=(num-(t-1))*(1/t)
62      direction = 1
63      c = x2list_c1[num-(t-1)]*(x2<=(cutpoint))+x2list_c2[num-(t-1)]*(
      x2>(cutpoint))
64      s = s + c
65    }
66
67    #running on test data
68    final=(s>0)
69    sum(y!=final)/length(y)
70    train_error[iter]=sum(y[training_idx]!=final[training_idx])/(
      num_samples)
71    test_error[iter]=sum(y[testing_idx]!=final[testing_idx])/(
      num_samples-num_training)
72  }
73
74  plot(train_error,type = "o",col = "red",ylim = c(0,0.3),
75       xlab = "epoch",ylab = "error rate",main="XGboost training error
      and testing error")
76  lines(test_error,type = "o",col = "green")
77 legend(30, 0.2, legend=c("Training", "Testing"),
78    col=c("red", "green"), lty=1:2, cex=0.8)
79
```

```
  plot(x1,x2,col = ifelse(final==1, "Red", ifelse(final==0, "#0073C2FF
    ", "Black")),pch = 20)

}
```

We also have the python code in Assignment7.

```
# -*- coding: utf-8 -*-
"""

 Stat 202A 2019 Fall - Homework 07
 Author:
 Date :

 INSTRUCTIONS: Please fill in the corresponding function. Do not change
     function names,
 function inputs or outputs. Do not write anything outside the function
    .

"""

import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import KFold, GridSearchCV
from matplotlib import pyplot as plt
import xgboost as xgb

cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

#
    ##############################################################################################

# TODO (1) Perform 5-fold validation for cancer data.
# You may consider use KFold function in sklearn.model_selection
# Print the mean and std of the 5-fold validation accuracy
#
    ##############################################################################################

kfold = KFold(5)
print
acc = []
i = 0
for m,n in kfold.split(X):
    i += 1
    X_train, X_test = X[m], X[n]
    y_train, y_test = y[m], y[n]
    xgb_model = xgb.XGBClassifier(objective="binary:logistic",
    random_state=42)
    xgb_model.fit(X_train, y_train)

    y_pred = xgb_model.predict(X_test)
    Xpos = X_test[y_pred == 1.0]
    Xneg = X_test[y_pred == 0.0]
    plt.figure()
    plt.scatter(Xpos[:, 0], Xpos[:, 1], c = 'r', label = 'Postive')
    plt.scatter(Xneg[:, 0], Xneg[:, 1], c = 'b', label = 'Negative')
    plt.legend()
    plt.title("Graph of 5-fold: split" + str(i))
    plt.savefig("Graph of 5-fold: split" + str(i) + ".png")

```

```
    acc.append(sum(y_pred == y_test)/len(y_test))
acc = np.array(acc)
print("Q1: 5-fold validation")
print("Mean: {}, Std: {}".format(acc.mean(), acc.std()))
#
    ##############################################################################################

# TODO (2) Perform Grid Search for parameter max_depth in [3,5,7] and
    min_child_weight in [0.1, 1, 5]
# For each combination use 5-Fold validation
# You may consider use GridSearchCV function in sklearn.modelselection
# Print the grid search mean test score for each paramter combination (
    use cv_results_)
#
    ##############################################################################################

d_candidate = [3,6,9,10]
w_candidate = [0,2,4,6]
parameters = {"max_depth":d_candidate, "min_child_weight":w_candidate}
xgb_model = xgb.XGBClassifier(objective="binary:logistic", random_state
    =42)
clf = GridSearchCV(xgb_model, parameters, cv=5)
clf.fit(X, y)
print('Q2:', clf.cv_results_['params'])
print('Q2:', clf.cv_results_['mean_test_score'])

params = clf.cv_results_['params']
for param in params:
    xgb_model = xgb.XGBClassifier(objective="binary:logistic",
    random_state=42, **param)
    xgb_model.fit(X, y)
    y_pred = xgb_model.predict(X)
    Xpos = X[y_pred == 1.0]
    Xneg = X[y_pred == 0.0]
    plt.figure()
    plt.scatter(Xpos[:, 0], Xpos[:, 1], c = 'r', label = 'Postive')
    plt.scatter(Xneg[:, 0], Xneg[:, 1], c = 'b', label = 'Negative')
    plt.legend()
    plt.title("Graph of parameter:"+str(param))
    plt.savefig("Graph of parameter:"+str(param)+".png")


#
    ##############################################################################################

# TODO (3) Plot the feature importance of the best model
# You may fit a new xgboost model using all the data and then plot the
    importance using xgb.plot_importance()
#
    ##############################################################################################

best_ind = np.argsort(clf.cv_results_['mean_test_score'])[-1]
best_params = clf.cv_results_['params'][best_ind]
print('Q3: best param', best_params)
print('Q3: best result', clf.cv_results_['mean_test_score'][best_ind])
best_xgb_model = xgb.XGBClassifier(objective="binary:logistic",
    random_state=42, **best_params)
best_xgb_model.fit(X, y)
plt.bar(range(len(best_xgb_model.feature_importances_)),
        best_xgb_model.feature_importances_)
plt.show()
```

### 4.6.5 Homework08-SVM

### 4.6.6 Problem Description

1. Write 2 version, (1) Gradient version. (2) Dual coordinate ascent version.
2. Try different kernel : 0 – linear: u'*v 1 – polynomial: $(gamma * u' * v + coef0)^2$ 2 – radial basis function: $exp(-gamma * |u - v|^2)$ 3 – sigmoid: $tanh(gamma * u' * v + coef0)$
3. Print accuracy.
4. Use part of mnist as data. There are 10 digit in mnist. You only need to train 2-class classifier and classify mnist 0 and 1 (for both training and testing). There will be about 10K data. If you computer cannot handle such big kernel. You may consider to use only part of the data to train. (at least 2k data)

### 4.6.7 SVM

First we consider the primal form. Perceptron is an algorithm which separates two classes by the sign of transformed data, i.e. $y_i = \text{sign}(x_i^T \beta)$. However, there can be multiple $\beta$'s which separates the data. The fundamental idea of SVM is to find the separating hyperplan with a large margin.

The goal is to find the $\beta$, so that * for positive examples $y = +$, $x^T \beta \geq 1$, * for negative examples $y = -$, $x^T \beta \leq -1$.

Here we use +1 and -1, becasuse we can always scale $\beta$ to make the margin 1.

The decision boundary is essentially decided by the training examples that lies on the margin. Let $u$ be an unit vector who has the same direction as $\beta$.

$$u = \frac{\beta}{|\beta|}.$$

Suppose $x$ is an example on the margin (a.k.a. support vector), the projection of $x$ on $u$ is

$$\langle x, u \rangle = \langle x, \frac{\beta}{|\beta|} \rangle = \frac{x^T \beta}{|\beta|} = \frac{1}{|\beta|}.$$

Hence, the SVM can be formulated as an optimization problem as follows:

$$\text{minimize} \quad \frac{1}{2}|\beta|^2, \tag{4.12}$$

$$\text{subject to} \quad y_i x_i^T \beta \geq 1, \forall i. \tag{4.13}$$

When the data is not separatable, SVM can be defined as follows which allows limited cross margin $\xi$.

We have the hinge loss.

$$\text{minimize} \quad \frac{1}{2}|\beta|^2 + c \sum_{i=1}^{n} \xi_i, \tag{4.14}$$

$$\text{subject to} \quad y_i x_i^T \beta \geq 1 - \xi_i, \forall i. \tag{4.15}$$

This is equivalent to

$$\text{minimize} \quad \frac{1}{2}|\beta|^2 + c \sum_{i=1}^{n} \max(0, 1 - y_i x_i^T \beta),$$

where $\sum_{i=1}^{n} \max(0, 1 - y_i x_i^T \beta)$ is usually called the **hinge loss**.

From this perspective, SVM can be think of an approximation to the loss of logistic regression hinge loss with a L-2 regularization.

$$\ell_{\text{logistic}}(\beta) = \sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T \beta}), \tag{4.16}$$

$$\ell_{\text{svm}}(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i x_i^T \beta) + \frac{\lambda}{2}|\beta|^2. \tag{4.17}$$

Given the loss function $\ell_(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i x_i^T \beta) + \frac{\lambda}{2}|\beta|^2$, we can solve $\beta$ by gradient descent.

Take derivative, we have

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{n} \mathbb{1}(1 - y_i x_i^T \beta < 1) \cdot (-y_i x_i) + \lambda \beta,$$

where $\mathbb{1}(\cdot)$ is an indicator function.

The code from logistic regression can be reused with a minor change in the gradient calculation.

We also have the dual form. Consider the convex hull of the positive and negative population, the margin between the two population is therefore defined by the minimum distance betweent two. Let $x_+ = \sum_{i \in +} c_i x_i$ and $x_- = \sum_{i \in -} c_i x_i$ ($c_i \geq 0, \sum_{i \in +} c_i = 1, \sum_{i \in -} c_i = 1$) be two points in the positive and negative population (in the region of the convex hull). The margin is

$$\min |x_+ - x_-|^2 .$$

It can be rewriten as

$$|x_+ - x_-|^2 = \left| \sum_{i \in +} c_i x_i - \sum_{i \in -} c_i x_i \right|^2 \tag{4.18}$$

$$= \left| \sum_{i} y_i c_i x_i \right|^2 \tag{4.19}$$

$$= \sum_{i,j} c_i, c_j y_i y_j \langle x_i, x_j \rangle, \tag{4.20}$$

$$\text{subject to} \quad c_i \geq 0, \sum_{i \in +} c_i = 1, \sum_{i \in -} c_i = 1. \tag{4.21}$$

After we solve for $c$, non-zeros $c_i$'s are support vectors, i.e. data examples on the boundary.

Here is the R code for SVM:

```
install.packages(data.table)
library(data.table) # allows us to use function fread


mySVM=function(X,Y,Xtest,Ytest,kernel='linear',method='gradient',
    num_iterations=75,learning_rate=1e-1,lamda=0.01)
{ X = X[1:2000,]
  Y = Y[1:2000]
```

```
  Xtest = Xtest[1:500,]
  Ytest = Ytest[1:500]
  training_acc = rep(0,num_iterations)

  n=dim(X)[1]
  p=dim(X)[2]+1
  X1 = cbind(rep(1, n), X)#intercept is included

  Y=2*Y-1# y=1--->1 y=0--->-1

  ntest = dim(Xtest)[1]
  Xtest1 = cbind(rep(1, ntest), Xtest)
  Ytest=2*Ytest-1
  score_test = matrix(rep(0,ntest),nrow=ntest)

  K = matrix(0,n,n)
  alpha = matrix(rep(0,n),nrow=n)
 for (i in 1:n) {
  for (j in 1:(n)) {
    if (kernel == 'polynomial'){
      K[i,j] = (X1[i,]%*%c(X1[j,]))**2
    }
    else if (kernel == 'sigmoid'){
      K[i,j] = tanh(0.0000001*(X1[i,]%*%c(X1[j,])))
    }
    else if(kernel == 'radial'){
      K[i,j] = exp(-0.0000005*(X1[i,]-X1[j,])%*%c(X1[i,]-X1[j,]))
    }
    else {
      K[i,j] = (X1[i,]%*%c(X1[j,]))
    }
  }
}

  K2 = matrix(0,n,ntest)
  for (i in 1:n) {
    for (j in 1:ntest) {
      if (kernel == 'polynomial'){
        K2[i,j] = (X1[i,]%*%c(Xtest1[j,]))**2
      }
      else if (kernel == 'sigmoid'){
        K2[i,j] = tanh(0.0000001*(X1[i,]%*%c(Xtest1[j,])))
      }
      else if(kernel == 'radial'){
        K2[i,j] = exp(-0.0000005*(X1[i,]-Xtest1[j,])%*%c(X1[i,]-Xtest1[
    j,]))
      }
      else {
        K2[i,j] = (X1[i,]%*%c(Xtest1[j,]))
      }
    }
  }

  if (method=='gradient'){

    for(it in 1:num_iterations){
      score = matrix(rep(0,n),nrow=n)
      for (i in 1:n){
        for (j in 1:n){
          score[j] = score[j] + alpha[i]*Y[i]*K[i,j]
        }
      }

```

```
    dalpha = matrix(rep(0,n),nrow=n)

    for (i in 1:n){
      for (j in 1:n){
        if (score[j]*Y[j]<1){
          dalpha[i] = dalpha[i] + Y[j]*Y[i]*K[i,j]
        }
      }
    }

    dalpha = dalpha/n

    alpha=alpha-lamda*alpha
    alpha=alpha+learning_rate*dalpha

    for (i in 1:n){
        for (j in 1:n){
          score[j] = score[j] + alpha[i]*Y[i]*K[i,j]
        }
    }

    predict_trainning_temp=sign(score)
    train_acc_temp=100*mean(predict_trainning_temp*Y>0)
    training_acc[it] = train_acc_temp
    #cat(it,"traning accuracy: ",train_acc_temp,"%\n")
  }

  for (j in 1:ntest){
      for (i in 1:n){
        score_test[j] = score_test[j] + alpha[i]*Y[i]*K2[i,j]
      }
  }
}

  if (method=='dual'){
    for (it in 1:num_iterations){
      score = matrix(rep(0,n),nrow=n)
      is = sample(1:n, 3, replace=TRUE)
      for (i in is){
        sum = 0
        for (j in 1:n){
          if (i!=j) {
            sum = sum + alpha[j]*Y[i]*Y[j]*K[i,j]
          }
        }
        alpha[i] = max((1-sum)/(Y[i]*Y[i]*K[i,i]),0)
      }
      for (i in 1:n){
        for (j in 1:n){
          score[j] = score[j] + alpha[i]*Y[i]*K[i,j]
        }
      }

    predict_trainning_temp=sign(score)
    train_acc_temp=100*mean(predict_trainning_temp*Y>0)
    training_acc[it] = train_acc_temp
    #cat(it,"traning accuracy: ",train_acc_temp,"%\n")
    }
    for (j in 1:ntest){
        for (i in 1:n){
          score_test[j] = score_test[j] + alpha[i]*Y[i]*K2[i,j]
        }
    }
```

```
  }


  predict_trainning_final=sign(score_test)
  test_acc=100*mean(predict_trainning_final*Ytest>0)
  #plot (training_acc,type="l",main=paste("training accuracy",kernel,
    method),xlabel="iteration",ylabel="accuracy")
  #return(beta)
  cat("Final traning accuracy: ",train_acc_temp,"%","\nFinal testing
    accuracy: ",test_acc,"%")
  return (list(training_acc, test_acc))
}




# load data
prepare_data <- function() {

  #load train data
  df = fread("/Users/apple/Desktop/202A/Assignment8/mnist-in-csv/
    mnist_train.csv")
  df = as.matrix(df)

  ## only keep the numbers we want.
  l_col <- df[,1]
  index = NULL

  subset = c(0,1)

  for (i in 1:length(subset)){
    number = subset[i]
    index = c(index,which(l_col == number))
  }

  index = sort(index)
  df = df[index,]

  # convert to numpy arrays.
  digits = df[,2:dim(df)[2]]
  labels = df[,1]

  #load test data
  df2 = fread("/Users/apple/Desktop/202A/Assignment8/mnist-in-csv/
    mnist_test.csv")
  df2 = as.matrix(df2)

  l_col <- df2[,1]
  index = NULL

  for (i in 1:length(subset)){
    number = subset[i]
    index = c(index,which(l_col == number))
  }

  index = sort(index)
  df2 = df2[index,]


  # convert to numpy arrays.s
  digits2 = df2[,2:dim(df2)[2]]
  labels2 = df2[,1]
```

```
192
193   return(list(digits, labels, digits2, labels2))
194
195 }
196
197 test <- function(){
198   result = prepare_data()
199   train_data = result[[1]]
200   train_label = result[[2]]
201   test_data = result[[3]]
202   test_label = result[[4]]
203
204
205   linear_gradient = mySVM(train_data,train_label,test_data,test_label,
        kernel='linear',method='gradient')
206   train_acc_linear_gradient = linear_gradient[[1]]
207   test_acc_linear_gradient = linear_gradient[[2]]
208   print (paste("test_acc_linear_gradient", test_acc_linear_gradient))
209   linear_dual =  mySVM(train_data,train_label,test_data,test_label,
        kernel='linear',method='dual')
210   train_acc_linear_dual = linear_dual[[1]]
211   test_acc_linear_dual = linear_dual[[2]]
212   print (paste("test_acc_linear_dual", test_acc_linear_dual))
213   poly_gradient = mySVM(train_data,train_label,test_data,test_label,
        kernel='polynomial',method='gradient')
214   train_acc_poly_gradient = poly_gradient[[1]]
215   test_acc_poly_gradient = poly_gradient[[2]]
216   print (paste("test_acc_poly_gradient", test_acc_poly_gradient))
217   poly_dual = mySVM(train_data,train_label,test_data,test_label,kernel=
        'polynomial',method='dual')
218   train_acc_poly_dual = poly_dual[[1]]
219   test_acc_poly_dual = poly_dual[[2]]
220   print (paste("test_acc_poly_dual", test_acc_poly_dual))
221   radial_gradient = mySVM(train_data,train_label,test_data,test_label,
        kernel='radial',method='gradient')
222   train_acc_radial_gradient = radial_gradient[[1]]
223   test_acc_radial_gradient = radial_gradient[[2]]
224   print (paste("test_acc_radial_gradient", test_acc_radial_gradient))
225   radial_dual = mySVM(train_data,train_label,test_data,test_label,
        kernel='radial',method='dual')
226   train_acc_radial_dual = radial_dual[[1]]
227   test_acc_radial_dual = radial_dual[[2]]
228   print (paste("test_acc_radial_dual", test_acc_radial_dual))
229   sigmoid_gradient = mySVM(train_data,train_label,test_data,test_label,
        kernel='sigmoid',method='gradient')
230   train_acc_sigmoid_gradient = sigmoid_gradient[[1]]
231   test_acc_sigmoid_gradient = sigmoid_gradient[[2]]
232   print (paste("test_acc_sigmoid_gradient", test_acc_sigmoid_gradient))
233   sigmoid_dual = mySVM(train_data,train_label,test_data,test_label,
        kernel='sigmoid',method='dual')
234   train_acc_sigmoid_dual = sigmoid_dual[[1]]
235   test_acc_sigmoid_dual = sigmoid_dual[[2]]
236   print (paste("test_acc_sigmoid_dual", test_acc_sigmoid_dual))
237
238
239   cl = rainbow(8)
240   plot(train_acc_linear_gradient,type = "l",col = cl[1],
241        xlab = "iteration",ylab = "accuracy",main="training accuracy of
        different settings")
242   lines(train_acc_linear_dual,type = "l",col = cl[2])
243   lines(train_acc_poly_gradient,type = "l",col = cl[3])
244   lines(train_acc_poly_dual,type = "l",col = cl[4])
245   lines(train_acc_radial_gradient,type = "l",col = cl[5])
```

```
246   lines(train_acc_radial_dual,type = "l",col = cl[6])
247   lines(train_acc_sigmoid_gradient,type = "l",col = cl[7])
248   lines(train_acc_sigmoid_dual,type = "l",col = cl[8])
249   legend(50, 70, legend=c("inear_gradient", "linear_dual","
        poly_gradient","poly_dual","radial_gradient","radial_dual","
        sigmoid_gradient","sigmoid_dual"),
250       col=cl, lty=1:2, cex=0.8)
251
252 }
253
254 test()
```