Intro to ML Homework4

Haotian Yi N18800809

April 2, 2020

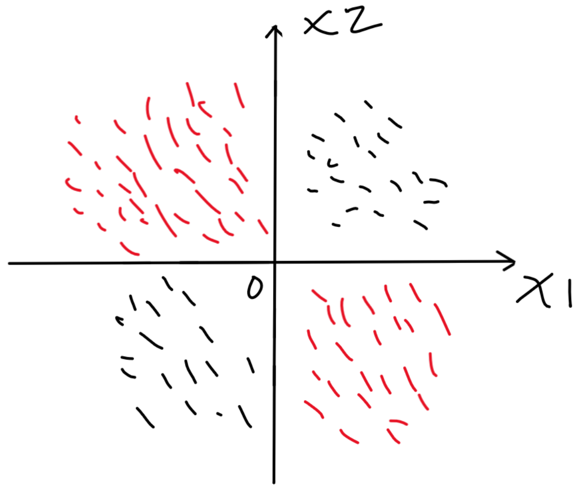
1 Question 1

(a)

A single perception is:

$$y = sign(\langle w, x_i \rangle + b)$$

Thus we can see that output y will also depend on w and b, and labels will be divided into two parts by a straight decision boundary. However the XOR function we want to classify is like:



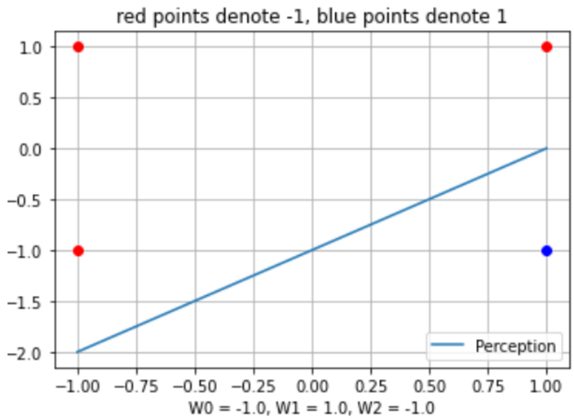
(black points denote "-1" label, red points denote "1" label)

So we can't classify XOR by a single perception's straight decision boundary. For instance, there could be situation that $y = 1$ when $x_1 = x_2$ in perception which is not in consistency with a standard two-variable XOR function.

(b) To solve this question, I consider -1 in perception equivalent to 0 in logical calculus, and in graph I plot, it will use 1 and -1 to show result of perception. Besides, I used perception in colab to help getting decision boundary. **X axis is x_1, y axis is x_2. Weights I got are at the bottom of each graph.**

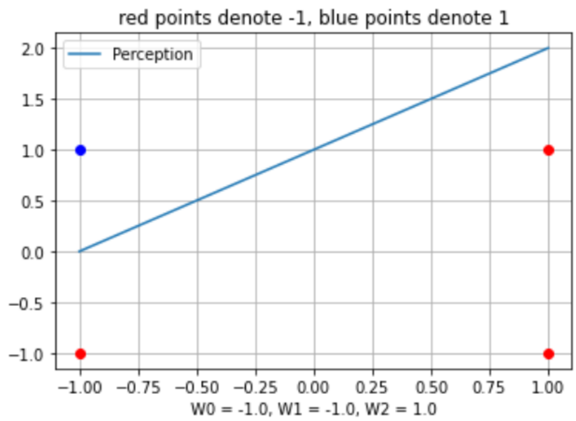
(i) For x1(AND)(NOT(x2)) there are four situations in **logical calculus**:
(1) $x_1 = 1, x_2 = 1, label = 0$ (2) $x_1 = 1, x_2 = 0, label = 1$
(3) $x_1 = 0, x_2 = 1, label = 0$ (4) $x_1 = 0, x_2 = 0, label = 0$

Equation for decision boundary in **perception**: $-1 + x_1 - x_2 = 0$



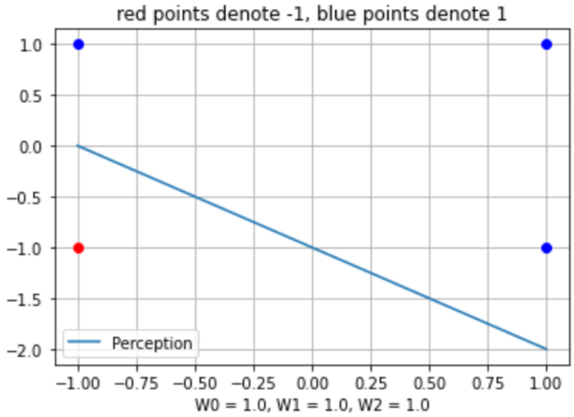
(ii) For (NOT(x1))(AND)x2 there are four situations in **logical calculus**:
(1) $x_1 = 1, x_2 = 1, label = 0$ (2) $x_1 = 1, x_2 = 0, label = 0$
(3) $x_1 = 0, x_2 = 1, label = 1$ (4) $x_1 = 0, x_2 = 0, label = 0$

Equation for decision boundary in **perception**: $1 + x_1 + x_2 = 0$



(iii) For x1(OR)x2 there are four situations in **logical calculus**:
(1) $x_1 = 1$,$x_2 = 1$,$label = 1$ (2) $x_1 = 1$,$x_2 = 0$,$label = 1$
(3) $x_1 = 0$,$x_2 = 1$,$label = 1$ (4) $x_1 = 0$,$x_2 = 0$,$label = 0$

Equation for decision boundary in **perception**: $-1 - x_1 + x_2 = 0$



(c) In XOR, it can be considered that points of 1, 3 quadrant are of same class and the same for points in 2, 4 quadrant. So we can combine x1(AND)(NOT(x2)) and (NOT(x1))(AND)x2, because they can respectively distinguish points at 4 quadrant, 2 quadrant.

$$y = sign(w_0 + w_1 * sign(-1 + x_1 - w_2) + w_2 * sign(-1 - x_1 + x_2))$$

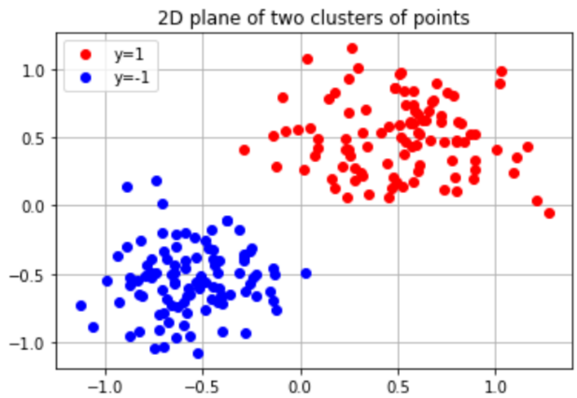
2 Question 2

(a) plot 2-D plane of points

(a)

```
[ ] import numpy as np
    import matplotlib.pyplot as plt
    meanx1 = (0.5, 0.5)
    covx1 = [[0.07, 0], [0, 0.07]]
    x1 = np.random.multivariate_normal(meanx1, covx1, 100)
    meanx2 = (-0.5, -0.5)
    covx2 = [[0.07, 0], [0, 0.07]]
    x2 = np.random.multivariate_normal(meanx2, covx2, 100)
    print("x1.shape:" ,x1.shape)
    print("x2.shape:" ,x2.shape)
    plt.figure(1)
    plt.title("2D plane of two clusters of points")
    plt.plot(x1[:,0],x1[:,1],'ro',label="y=1")
    plt.plot(x2[:,0],x2[:,1],'bo',label="y=-1")
    plt.legend()
    plt.grid()

    y1 = np.random.normal(loc=1.0, scale=0.0, size=100)
    y2 = np.random.normal(loc=-1.0, scale=0.0, size=100)
    y1 = y1.reshape(y1.shape[0],1)
    y2 = y2.reshape(y2.shape[0],1)
    print("y1.shape:" ,y1.shape)
    print("y2.shape:" ,y2.shape)
```

```
x1.shape: (100, 2)
x2.shape: (100, 2)
y1.shape: (100, 1)
y2.shape: (100, 1)
```



(b) do perception on separated points

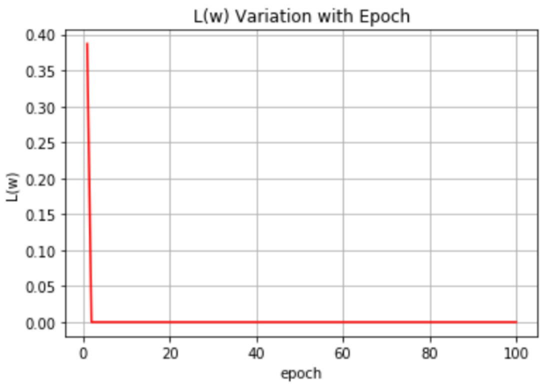
(b)

```
def perception(x,y,epoch,learning_rate):
  import numpy as np
  import matplotlib.pyplot as plt
  pairs = np.concatenate((x, y), axis=1)
  #print("pairs.shape: ",pairs.shape,"pairs:")
  np.random.shuffle(pairs)
  #print(pairs)
  X = pairs[:,0:pairs.shape[1]-1]
  y = pairs[:,pairs.shape[1]-1:pairs.shape[1]]
  #print(X)
  #print(y)
  n = X.shape[0]
  d = X.shape[1]
  x0 = np.ones((n,1))
  X = np.hstack((x0,X))# add a column of 1 to the left of x matching for w0
  w = np.zeros(d+1)
  print("X.shape: ",X.shape," y.shape: ",y.shape," w.shape: ",w.shape)

  cost = []
  converge = False;
  for T in range(1,epoch+1):
    Lw = 0
    for i in range(n):
      if np.matmul(w,X[i].T)*y[i] <= 0 :
        Lw = Lw - np.matmul(w,X[i].T)*y[i]
        w = w - learning_rate*(-1)*y[i]*X[i]
```

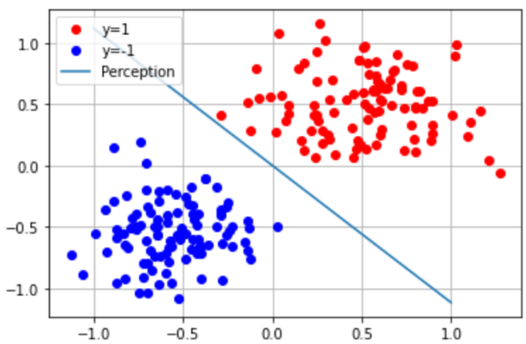
```
    cost.append(Lw)
    if (Lw == 0 and converge == False) :
      converge = True
      print("no change in epoch {} ".format(T))

  plt.figure()
  plt.title("L(w) Variation with Epoch")
  iterations = range(1,epoch+1)
  plt.plot(iterations,cost,'r')
  plt.xlabel("epoch")
  plt.ylabel("L(w)")
  plt.grid()
  print("w: ",w)
  print("Last Loss L(w): ",cost[-1])

  y_pred = X@w.T
  y_pred[y_pred<0]=-1
  y_pred[y_pred>=0]=1

  acc = 0
  for i in range(n):
    if y_pred[i] == y[i]:
      acc += 1
  print("final accuracy: ", acc/200)
  return w
```

```
[ ] x = np.concatenate((x1, x2), axis=0)
    y = np.concatenate((y1, y2), axis=0)

    perception(x,y,100,1)
```

```
X.shape:  (200, 3)  y.shape:  (200, 1)  w.shape:  (3,)
no change in epoch 2
w:  [0.          1.0992005   1.29950527]
Last Loss L(w):  0
final accuracy:  1.0
array([0.         , 1.0992005 , 1.29950527])
```




```
plt.figure()

plt.plot(x1[:,0],x1[:,1],'ro',label="y=1")
plt.plot(x2[:,0],x2[:,1],'bo',label="y=-1")
linex1 = [-1 , 1]
#boundary w0 + w1*x1 + w2*x2 = 0
linex2 = [(0.0 - w[0] - w[1]*linex1[0]) / w[2],(0.0 - w[0] - w[1]*linex1[1]) / w[2]]
plt.plot(linex1,linex2,label = "Perception")
plt.legend()
plt.grid()
```



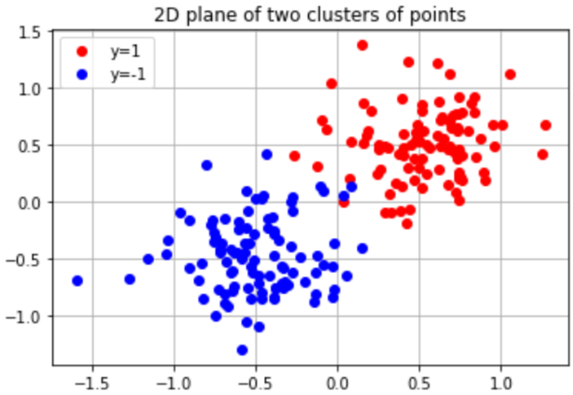
Result is shown above. Perception on this case converge at epoch 2 which is very fast.

(c) Perform perception on overlapping points

(c)

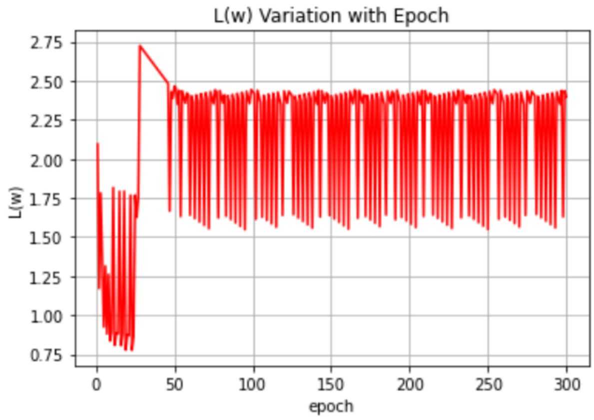
```
[3]  import numpy as np
     import matplotlib.pyplot as plt
     mean1 = (0.5, 0.5)
     cov1 = [[0.09, 0], [0, 0.09]]
     X1 = np.random.multivariate_normal(mean1, cov1, 100)
     mean2 = (-0.5, -0.5)
     cov2 = [[0.09, 0], [0, 0.09]]
     X2 = np.random.multivariate_normal(mean2, cov2, 100)
     print("X1.shape:" ,X1.shape)
     print("X2.shape:" ,X2.shape)
     plt.figure(1)
     plt.title("2D plane of two clusters of points")
     plt.plot(X1[:,0],X1[:,1],'ro',label="y=1")
     plt.plot(X2[:,0],X2[:,1],'bo',label="y=-1")
     plt.legend()
     plt.grid()

     Y1 = np.random.normal(loc=1.0, scale=0.0, size=100)
     Y2 = np.random.normal(loc=-1.0, scale=0.0, size=100)
     Y1 = Y1.reshape(Y1.shape[0],1)
     Y2 = Y2.reshape(Y2.shape[0],1)
     print("Y1.shape:" ,Y1.shape)
     print("Y2.shape:" ,Y2.shape)
```

```
X1.shape: (100, 2)
X2.shape: (100, 2)
Y1.shape: (100, 1)
Y2.shape: (100, 1)
```

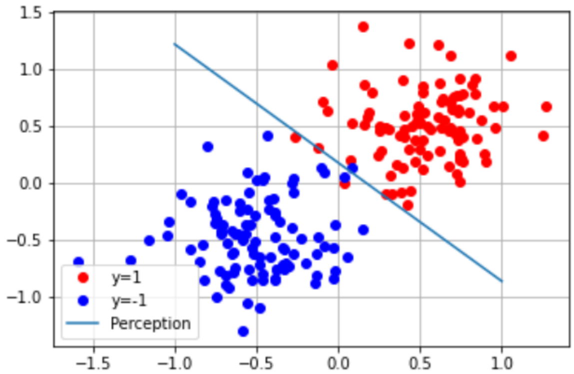


```
[4] x = np.concatenate((X1, X2), axis=0)
    y = np.concatenate((Y1, Y2), axis=0)
    w = perception(x,y,300,1)
```

```
X.shape:  (200, 3)  y.shape:  (200, 1)  w.shape:  (3,)
w:  [-1.          5.90306856  5.65967229]
Last Loss L(w):  [2.39681691]
final accuracy:  0.985
```




```
[5] plt.figure()

    plt.plot(X1[:,0],X1[:,1],'ro',label="y=1")
    plt.plot(X2[:,0],X2[:,1],'bo',label="y=-1")
    linex1 = [-1 , 1]
    #boundary w0 + w1*x1 + w2*x2 = 0
    linex2 = [(0.0 - w[0] - w[1]*linex1[0]) / w[2],(0.0 - w[0] - w[1]*linex1[1]) / w[2]]
    plt.plot(linex1,linex2,label = "Perception")
    plt.legend()
    plt.grid()
```



According to results shown above, algorithm did not converge but it still generate a decision boundary with good accuracy of 98.5%.

3 Question 3

(a)

$$SE = \sum_{i=1}^{n} (\phi(x_i)w - y_i)^2$$

(b) Loss function with a L2 regularizer as shown below, and derive closed form expression for w.

$$\begin{aligned}
&L(w) = \frac{1}{2}\|\Phi w - y\| + \frac{\alpha}{2}\|w\|_2^2 \\
&\frac{\partial L}{\partial w} = \Phi^T(\Phi w - y) + \alpha w = 0 \\
&(\Phi^T\Phi + \alpha I)w = \Phi^T y \\
&w = (\Phi^T\Phi + \alpha I)^{-1}\Phi^T y
\end{aligned}$$

(c)

$$f(z) = \langle w, \phi(x_i)\rangle = \phi(z) * w = \phi(z) * (\Phi^T\Phi + \alpha I)^{-1}\Phi^T y$$

(d) For w in (b), use Sherman-Morrison-Woodbury identity for matrix to transform:

$$(A^{-1} + B^T C^{-1} B)^{-1} B^T C^{-1} = AB^T(BAB^T + C)^{-1}$$

Consider w, $(\alpha I)^{-1}$ as A, Φ as B, I as C, so:

$$\begin{aligned}
w &= (((\alpha I)^{-1})^{-1} + \Phi^T I^{-1}\Phi)^{-1}\Phi^T I^{-1} * y \\
&= (\alpha I)^{-1}\Phi^T(\Phi(\alpha I)^{-1}\Phi^T + I^{-1})^{-1} * y \\
&= (\alpha I)^{-1}\Phi^T((\alpha I)^{-1}\Phi\Phi^T + I^{-1})^{-1} * y \\
&= (\alpha)^{-1}\Phi^T((\alpha)^{-1}M + I)^{-1} * y
\end{aligned} \tag{1}$$

Matrix M above is : (use d^r represent dimension after lifting for input)

$$M = \Phi\Phi^T = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(d^r)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(d^r)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(d^r)} \end{bmatrix} * \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(d^r)} & x_2^{(d^r)} & \dots & x_n^{(d^r)} \end{bmatrix}$$

$$M = \begin{bmatrix} K(x_1,x_1) & K(x_1,x_2) & \dots & K(x_1,x_n) \\ K(x_2,x_1) & K(x_2,x_2) & \dots & K(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n,x_1) & K(x_n,x_2) & \dots & K(x_n,x_n) \end{bmatrix}$$

Then for $f(z)$ in (c):

$$\begin{aligned}
f(z) = \langle w, \phi(x_i) \rangle &= \phi(z) * w = \phi(z) * (\alpha)^{-1} \Phi^T ((\alpha)^{-1} M + I)^{-1} * y \\
&= (\alpha)^{-1} \phi(z) \Phi^T ((\alpha)^{-1} M + I)^{-1} * y \\
&= (\alpha)^{-1} N ((\alpha)^{-1} M + I)^{-1} * y \\
&= N(M + \alpha I)^{-1} * y
\end{aligned}$$

Matrix N above is: (d^r represent dimension after lifting for input)

$$N = \phi(z)\Phi^T = \begin{bmatrix} z_1^{(1)} & z_1^{(2)} & \dots & z_1^{(d^r)} \end{bmatrix} * \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(d^r)} & x_2^{(d^r)} & \dots & x_n^{(d^r)} \end{bmatrix}$$

$$N = \begin{bmatrix} K(z_1, x_1) & K(z_1, x_2) & \dots & K(z_1, x_n) \end{bmatrix}$$

So $f(z) = N(M + \alpha I)^{-1} y$ has no $\phi(x)$.

Thus the calculations in (b) and (c) can be performed by invoking the kernel dot product alone without explicitly writing down $\phi(x)$ ever.

4 Question 4

(a) Load data set and display 10 representatives from each class

(a)

load dataset

```
[6] import tensorflow as tf
    from tensorflow import keras
    fashion_mnist = keras.datasets.fashion_mnist
    (train_images, train_labels),(test_images, test_labels) = fashion_mnist.load_data()
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

find representatives' indices for 10 classes

```
[2] print(train_images.shape)
    print(test_images.shape)
```

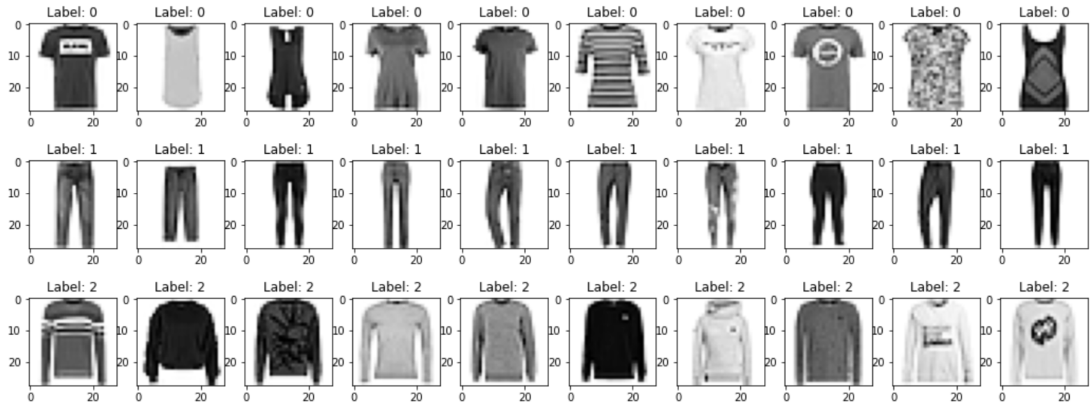
```
(60000, 28, 28)
(10000, 28, 28)
```

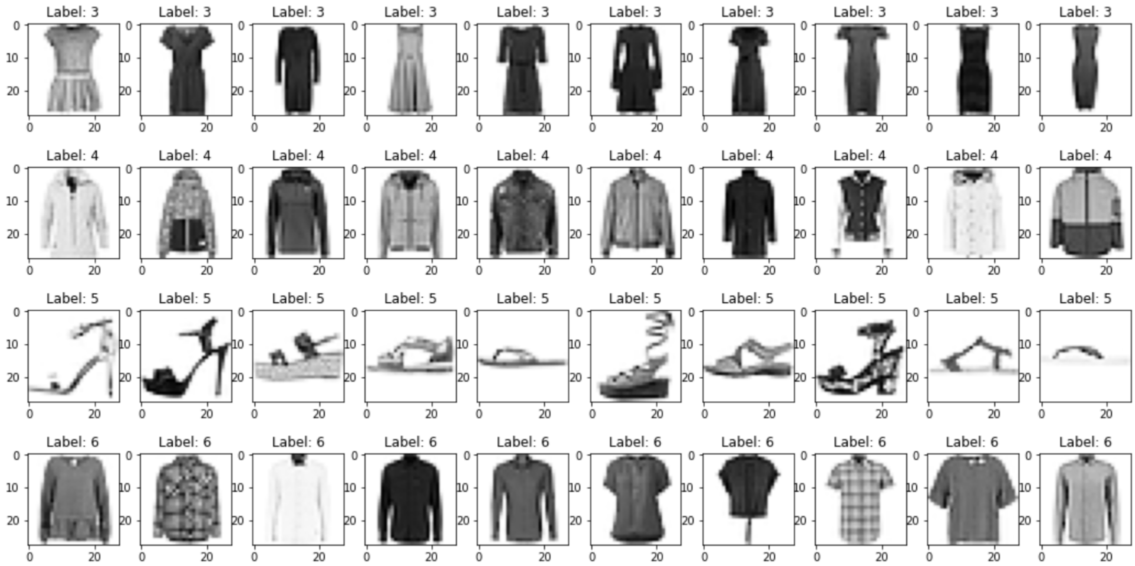
```
[3] import numpy as np
    labels = []
    for clas in range(10):
      count = 0
      for pic in range(60000):
        if train_labels[pic] == clas :
          count = count + 1
          labels.append(pic)
          if (count == 10):
            break
```

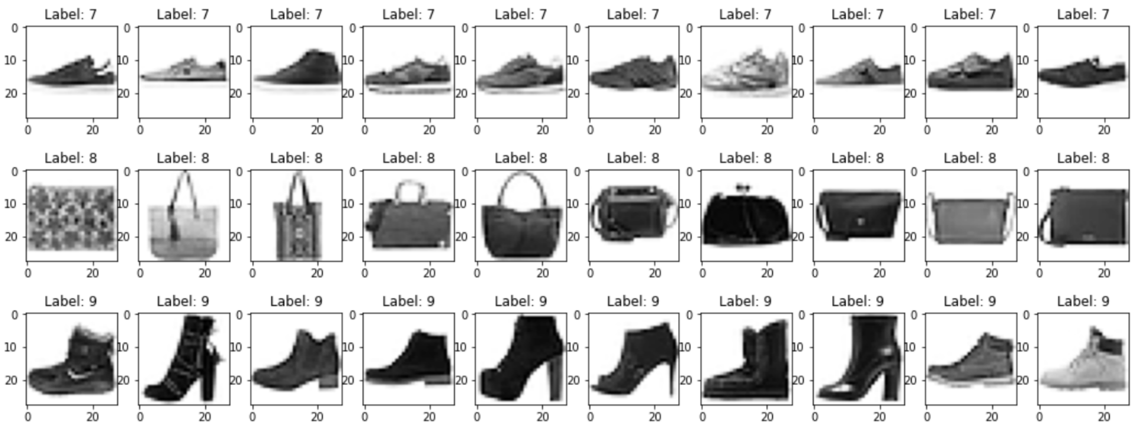
plot representatives for 10 classes

```
[15] import matplotlib.pyplot as plt
     %matplotlib inline
     # Only use this if using iPython
     plt.figure(figsize = (20,4)) # set area of plot 20 inch width 4 inch height

     for i in range(10):
       no = 1
       f = plt.figure()
       f.set_figheight(18)
       f.set_figwidth(18)
       for index in range(10):
         plt.subplot(10,10,no) #subplot(row,column,index)
         plt.imshow(np.reshape(train_images[labels[i*10 + index]],(28,28)), cmap='Greys')#cmap=plt.cm.gray
         plt.title('Label: {}'.format(train_labels[labels[i*10 + index]]))
         no += 1
```

```
<Figure size 1440x288 with 0 Axes>
```







(b)&(c) Implement the following classification methods: k-NN, logistic regression, and support vector machines (with linear and rbf kernels).Report best possible test-error performances by tuning hyperparameters in each of your methods.

(b)

Extract 5000 for train and 500 for test to make program running time not too long, then flatten images and normalize data.

```
train_flat = np.zeros(shape=(5000,784))
test_flat = np.zeros(shape=(500,784))
for i in range(5000):
  train_flat[i] = train_images[i].flatten()
for i in range(500):
  test_flat[i] = test_images[i].flatten()
#print(train_flat[1])

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()#normalize data
train_flat = scaler.fit_transform(train_flat)
test_flat = scaler.fit_transform(test_flat)

train_labels = train_labels[0:5000]
test_labels = test_labels[0:500]

print("train_flat.shape: ",train_flat.shape," test_flat.shape: ",test_flat.shape)
print("train_labels.shape: ",train_labels.shape," test_labels.shape: ",test_labels.shape)
```

```
train_flat.shape:  (5000, 784)  test_flat.shape:  (500, 784)
train_labels.shape:  (5000,)  test_labels.shape:  (500,)
```

K-NN

```
[34] from sklearn import neighbors
     clf = neighbors.KNeighborsClassifier(9)
     %time clf.fit(train_flat, train_labels)
```

```
CPU times: user 341 ms, sys: 2 ms, total: 343 ms
Wall time: 346 ms
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=9, p=2,
                     weights='uniform')
```

```
[35] %time knnscore = clf.score(test_flat, test_labels)
     print(knnscore)
```

```
CPU times: user 3.93 s, sys: 1.99 ms, total: 3.93 s
Wall time: 3.94 s
0.83
```

After trying several K, K = 9 is best for this KN-N. As we can see, accuracy is 83.00%, which is acceptable.

Logistic Regression

```
[21] from sklearn.linear_model import LogisticRegression

     logisticReg = LogisticRegression(penalty = 'l2', tol = 0.01,solver = 'saga',C=0.05)
     %time logisticReg.fit(train_flat,train_labels)
```

```
CPU times: user 8.35 s, sys: 17 µs, total: 8.35 s
Wall time: 8.37 s
LogisticRegression(C=0.05, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='saga', tol=0.01, verbose=0,
                   warm_start=False)
```

```
[22] %time lrscore = logisticReg.score(test_flat,test_labels)
     print(lrscore)
```

```
CPU times: user 2.85 ms, sys: 997 µs, total: 3.84 ms
Wall time: 2.62 ms
0.85
```

After tuning the parameters, penalty = 'l2', tol = 0.01, solver = 'saga', C=0.05 is best for this method. As we can see, accuracy is 85.00%, which is even better than KN-N.

SVMs(linear kernel)

```
from sklearn import svm
svc = svm.SVC(kernel = 'linear',C=0.001,probability=False)
%time svc.fit(train_flat,train_labels)
```

```
CPU times: user 7.17 s, sys: 8.97 ms, total: 7.18 s
Wall time: 7.2 s
SVC(C=0.001, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

```
[30] %time lscore = svc.score(test_flat, test_labels)
     print(lscore)
```

```
CPU times: user 1.27 s, sys: 1 ms, total: 1.27 s
Wall time: 1.28 s
0.856
```

After tuning the parameter, C = 0.001 is best for this linear-kernel SVMs. As we can see, accuracy is 85.60%, which higher than logistic regression.

SVMs(RBF kernel)

```
[53] svmrbf  = svm.SVC(probability=False,kernel='rbf',C=5)
     %time svmrbf.fit(train_flat,train_labels)
```

```
CPU times: user 9.66 s, sys: 3.02 ms, total: 9.67 s
Wall time: 9.68 s
SVC(C=5, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

```
%time rbfscore = svmrbf.score(test_flat, test_labels)
print(rbfscore)
```

```
CPU times: user 1.41 s, sys: 1.01 ms, total: 1.41 s
Wall time: 1.41 s
0.874
```

After tuning the parameter, C = 5 (less regularization strength compared to linear kernel) is good for this linear-kernel SVMs. As we can see, accuracy is 87.40%, which is higher than any others.

(d) Report train- and test-running time of each of your methods in the form of a table, and comment on the relative tradeoffs across the different methods.

Method	Train Time	Test Time	Accuracy
KN-N	0.346s	3.94s	83.00%
Logistic Regression	8.42s	0.0102s	85.00%
SVMs(Linear)	7.20s	1.28s	85.60%
SVMs(RBF)	9.68s	1.41s	87.40%

Method	Tradeoffs
KN-N	As we can see that KN-N takes very quick to train but longest time to test, in practical use we often want test to be quick.
Logistic Regression	Logistic regression has an advantage that it tests fastest among these methods although it need long time to train.
SVMs(Linear)	SVMs(Linear) is not as fast as logistic regression on test time but more accurate than logistic regression.
SVMs(RBF)	SVMs(RBF) has longest time to train but highest accuracy.