//////////////////// IMPORTS ////////////////////

In [3]:
```
import numpy as np
import urllib
import math
import sys
import pandas
```

//////////////////// FUNCTIONS ////////////////////

/////////////////// ACCURACY FUNCTIONS ///////////////////

In [25]:
```
def accuracy(predicted,actual):
    correctList = []
    nData = len(actual)
    #print(nData)
    for i in range(0,nData):
        if predicted[i] == actual[i]:
            correct = 1
        else:
            correct = 0
        #print(correct)
        correctList.append(correct)
    nCorrect = np.sum(np.array(correctList))
    #print(nCorrect)
    #print(nData)
    pCorrect = (nCorrect/nData)*100


    return [nCorrect, pCorrect]
    #print(nIncorrect)
    #print(pIncorrect)
    #print(pCorrect)
```

In [26]:
```
def getConfusion(predicted, actual, n=10):
    cMat = np.zeros((n, n))
    for p, a in zip(predicted, actual):
        cMat[a][p] += 1
    return cMat.astype(int)
```

////////// SHARED FUNCTIONS //////////

In [20]:
```
def sigmoid(x):
    return 1/(1 + np.exp(-1 * x))
```

In [146…
```
def getBinaryY(yOneHot):
    yBinary = []
    for col in yOneHot.T:
        yBinary.append(np.where(col < 1, -1, col).astype(int))
        #print(yBinary)
    return yBinary
```

////////// SVM FUNCTIONS //////////

In [255…
```
def svmPredict(w, x):
    p = np.dot(x,w)
    #print(p)
    return sigmoid(p)
    #if p > 0:
    #    return 1
    #return 0
```

In [167…
```
def pegasos(X, y, lamda, T = 100):
    ns = X.shape[0]
    nf = X.shape[1]
    thekMap = np.zeros((ns,ns))
    w = np.zeros(nf)
    for t in range(T):
        eta = 1/(lamda * (t + 1))
        #print(eta)
        factor = w * (1 - (eta * lamda))
        #print(factor)
        shuffled = np.arange(ns)
        np.random.shuffle(shuffled)
        for i in shuffled:
            #if t > 0:
            #    pred = kPredict(w, X[i], X, y, i, thekMap, True)
            #else:
            #    pred = kPredict(w, X[i], X, y, i, thekMap)
            pred = np.dot(x[i],w)
            #print(pred)
            if pred * y[i] < 1:
                w = factor + (eta * y[i] * X[i])
            else:
                w = factor
    return w
```

In [220…
```
def svmTrainNB(X, yBinary, lamda, T = 100):
    wList = []
    n = 0
    for yCat in yBinary:
        print(n)
        n += 1
        wList.append(pegasos(X, yCat, lamda, T))
    return wList
```

In [178…
```
def svmPredictNB(wList, X): # yBinary is the training y set
    predList = []
    for sample in X:
        predPerCat = []
        for weight in wList:
            predPerCat.append(svmPredict(weight, sample))
        #print(predPerCat)
        predList.append(predPerCat.index(max(predPerCat)))
    return predList
```

////////// KERNEL PERCEPTRON FUNCTIONS //////////

In [144…
```
def kern(x1,x2,d=2,c=1):
    pre = c + np.dot(x1.T,x2)
    #print(pre)
    ret = pre**d
    return ret
```

In [145…
```
def kPredict(alpha,xj,xData,yData,j=0,kMap=None,mapped=False):
    #print(type(kMap))
    #print("kPredict")
    yp = 0
    for i in range(yData.size - 1):
        if not mapped:
                k = kern(xData[i],xj)
                if not type(kMap) == type(None):
                        kMap[i][j]=k
                #print("calculated k")
        else:
                k = kMap[i,j]
                #print("k from map")
        #print("k: ", k)
        yp += alpha[i] * yData[i] * k
        #print("yp: ", yp)
    if yp > 0:
            yPredicted = 1
    else:
            yPredicted = -1
    return yPredicted
```

In [ ]:
```
def train(x, y, T=100):
    ns = x.shape[0]
    nf = x.shape[1]
    #print("ns: ", nf)
    a = np.zeros(ns)
    iRange = range(T)
    jRange = range(ns)
    aMax = 0
    thekMap = np.zeros((a.size,len(jRange)))

    for i in iRange:
        for j in jRange:
            if i <= 0:
                yHat = kPredict(a,x[j,:],x,y,j,thekMap)
            else:
                yHat = kPredict(a,x[j,:],x,y,j,thekMap,True)

            if yHat != y[j]:
                a[j] += 1
            if a[j] > aMax:
                aMax = a[j]
        if aMax < i:
            print("Stoped training early as alpha is static")
            break
            #print("a[j]: ", a[j])
        #print(a)
    return a
```

In [ ]:
```
def kernelTrainNB(X, yBinary, T = 100):
    aList = []
    for yCat in yBinary:
        aList.append(train(X, yCat, T))
    return aList
```

In [ ]:
```
def predictNB(aList, X, Xtrain, yBinary):# yBinary is the training y set
    predList = []
    for sample in X:
        predPerCat = []
        for yCat, alpha in zip(yBinary, aList):
            predPerCat.append(kPredict(alpha, sample, Xtrain, yCat))
        predList.append(predPerCat.index(max(predPerCat)))
    return predList
```

////////// NEURAL NETWORK FUNCTIONS //////////

In [4]:
```
def initWeight(sz):
    return np.random.randn(sz)
```

In [5]:
```
def initNN(nIn, nHid, nOut):
    net = []
    hidLayer = []
    for i in range(nHid):
        hidLayer.append({'weights':initWeight(nIn)})
    net.append(hidLayer)
    outLayer = []
    for j in range(nOut):
        outLayer.append({'weights':initWeight(nHid)})
    net.append(outLayer)
    return net
```

In [6]:
```
def activate(wt,xData):
    #print("wt: ", wt)
    #print("xData: ", xData)
    wSum = np.dot(wt,xData)
    #print("wSum", wSum)
    s = sigmoid(wSum)
    #print("s: ", s)
    return s
```

In [7]:
```
def fwdProp(net, sample):
    sampleIn = sample
    for layer in net:
        inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], sampleIn)
            #print(activation)
            neuron['out'] = activation
            inputs.append(neuron['out'])
        sampleIn = inputs
    return sampleIn
```

In [8]:
```
def derive_activation(pred):
    return pred * (1 - pred)
```

In [9]:
```
def getCost(pred, actual):
    cList = []
    for p, a in zip(pred, actual):
        cList.append(pow((p - a), 2))
    return sum(cList)
```

In [10]:
```
def backProp(net, actual):
    for i in reversed(range(len(net))):
        layer = net[i]
        errs = []
        if i == len(net) - 1:
            for j, neuron in enumerate(layer):
                #print(neuron['out'])
                errs.append(actual[j] - neuron['out'])
                #print(errs[j])
        else:
            for j in range(len(layer)):
                err = 0.0
                for neuron in net[i + 1]:
                    err += (neuron['weights'][j] * neuron['offset'])
                errs.append(err)
        for j, neuron in enumerate(layer):
            #print(neuron['out'])
            neuron['offset'] = errs[j] * derive_activation(neuron['out'])
```

In [11]:
```
def updateWeights(net, sample, rate):
    for i, layer in enumerate(net):
        inputs = sample[:-1]
        if not i == 0:
            inputs = []
            for neuron in net[i - 1]:
                inputs.append(neuron['out'])
        for neuron in layer:
            for j, anInput in enumerate(inputs):
                neuron['weights'][j] += rate * neuron['offset'] * anInput
```

In [12]:
```
def trainNN(net, X, y, rate, T, nOut):
    for t in range(T):
        sys.stdout.write("\r" + str(t) + "  ")
        sys.stdout.flush()
        for i, sample in enumerate(X):
            outs = fwdProp(net, sample)
            backProp(net, y[i])
            updateWeights(net, sample, rate)
```

In [13]:
```
def predictNN(net, sample):
    pred = fwdProp(net, sample)
    return pred.index(max(pred))
```

//////////////////// ALL FUNCTIONS ABOVE ////////////////////

//////////////////// DATA IMPORT AND FORMATTING ////////////////////

In [14]:
```
import numpy as np
import urllib

nF = 780
dta = []
y = []
with open("../input/mnistscale/mnist.scale") as f:
    for line in f:
        lineArray = np.zeros(nF)
        ln = line.replace(" \n","")
        rawLine = ln.split(" ")
        #print(rawLine)
        for pixel in rawLine:
            if ":" in pixel:
                breakdown = pixel.split(":")
                #print(breakdown)
                lineArray[int(breakdown[0]) - 1] = float(breakdown[1])
            else:
                #print(pixel)
                #lineArray[0] = int(pixel)
                y.append(int(pixel))
        #print(lineArray)
        dta.append(lineArray)
        #print(len(dta))
```

In [15]:
```
x = np.array(dta)
ynp = np.array(y)
XtrainSmall = x[0:500]
yTrainSmall = ynp[0:500]

yTrainSmallArrays = []
for yVal in yTrainSmall:
    yArr = np.zeros(10)
    yArr[yVal] = 1
    yTrainSmallArrays.append(yArr)
yTrainSmallArrays = np.array(yTrainSmallArrays)

yOneHot = []
for yVal in ynp:
    yArr = np.zeros(10)
    yArr[yVal] = 1
    yOneHot.append(yArr)
yOneHot = np.array(yOneHot)

Xtrain = x[0:42000]
Xtest = x[42001:]
yOneHotTrain = yOneHot[0:42000]
yOneHotTest = yOneHot[42001:]
ynpTest = ynp[42001:]
```

In [17]:
```
print(len(Xtrain))
```

```
42000
```

In [16]:
```
nSplit = 840
XtrainBatches = np.split(Xtrain, nSplit)
yOneHotTrainBatches = np.split(yOneHotTrain, nSplit)
```

In [18]:
```
print(len(XtrainBatches))
```

```
840
```

///////// SVM IMPLEMENTATION //////////

In [217…
```
yTrainBinary = getBinaryY(yOneHotTrain)
```

In [221…
```
weightList = svmTrainNB(Xtrain, yTrainBinary, 0.001, 300)
#print(weightList)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [256…
```
predictions = svmPredictNB(weightList, Xtest)
#print(predictions)
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning: over
flow encountered in exp
```

In [261…
```
errSVM = accuracy(predictions, ynpTest)

print("SVM Accuracy:")
print("Number correct: ", errSVM[0], " out of ", len(predictions))
print("Percentage correct: ", errSVM[1], "%")
```

```
SVM Accuracy:
Number correct:  13424  out of  17999
Percentage correct:  74.58192121784543 %
```

In [272…
```
confusionSVM = getConfusion(predictions, ynpTest)
label = np.arange(10)
cmSVM = pandas.DataFrame(confusionSVM, columns = label, index = label)
print("SVM Confusion Matrix: ")
cmSVM
```

SVM Confusion Matrix:

Out[272…

	0	1	2	3	4	5	6	7	8	9
0	1700	0	24	22	3	16	2	1	8	4
1	1	1783	54	49	3	13	2	6	31	0
2	50	10	1594	19	19	7	27	38	28	19
3	55	21	279	1274	4	123	5	31	50	21
4	27	27	68	68	1277	18	6	18	35	199
5	79	31	186	179	61	976	12	14	47	52
6	120	15	500	7	55	62	972	0	6	1
7	201	23	40	28	27	3	1	1528	52	32
8	22	85	235	167	15	89	3	19	1143	33
9	60	38	41	40	71	28	1	191	92	1177

////////// KERNEL PERCEPTRON IMPLEMENTATION //////////

In [273…
```
yBinary = getBinaryY(yTrainSmallArrays)
XtestSmall = Xtest[0:500]
yTestSmall = ynpTest[0:500]
```

In [274…
```
#Smaller training sets were used for final report due to time constraints.
alphaList = kernelTrainNB(XtrainSmall, yBinary)
```

```
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
Stoped training early as alpha is static
```

In [275…
```
predictedList = predictNB(alphaList, XtestSmall, XtrainSmall, yBinary)
print(predictedList)
```

```
[2, 1, 8, 9, 0, 8, 6, 0, 8, 0, 7, 2, 9, 2, 1, 7, 9, 9, 2, 0, 6, 6, 7, 3, 6, 1, 6, 1,
0, 7, 7, 0, 4, 8, 0, 7, 0, 0, 1, 1, 3, 9, 3, 1, 0, 0, 5, 9, 5, 0, 0, 9, 0, 3, 3, 3,
7, 1, 0, 8, 7, 7, 0, 7, 2, 6, 7, 2, 0, 2, 1, 6, 3, 0, 6, 9, 7, 9, 8, 6, 0, 3, 0, 0,
0, 0, 3, 0, 6, 9, 0, 2, 1, 7, 2, 8, 0, 1, 9, 0, 0, 1, 6, 0, 7, 9, 8, 0, 0, 5, 0, 8,
1, 1, 0, 4, 0, 9, 0, 3, 0, 5, 0, 9, 0, 2, 8, 4, 9, 8, 0, 1, 0, 7, 2, 0, 0, 2, 8, 0,
0, 2, 0, 0, 7, 7, 8, 9, 9, 0, 6, 9, 0, 0, 8, 1, 0, 2, 2, 6, 0, 6, 0, 7, 9, 6, 9, 2,
7, 0, 1, 6, 9, 6, 1, 1, 7, 0, 1, 0, 9, 1, 0, 1, 0, 3, 4, 2, 1, 6, 7, 9, 8, 1, 0, 8,
1, 4, 0, 7, 0, 7, 2, 7, 1, 2, 0, 8, 0, 3, 9, 3, 8, 7, 3, 9, 0, 8, 2, 9, 5, 9, 2, 0,
6, 8, 0, 2, 5, 0, 9, 5, 9, 6, 0, 4, 0, 8, 0, 2, 8, 1, 0, 5, 0, 0, 0, 3, 9, 0, 3, 5,
0, 2, 0, 6, 0, 8, 8, 1, 1, 0, 0, 0, 0, 8, 4, 1, 7, 0, 0, 3, 2, 5, 0, 1, 0, 2, 1, 2,
7, 7, 7, 3, 9, 9, 4, 1, 9, 9, 8, 9, 9, 0, 6, 7, 8, 7, 8, 4, 2, 0, 2, 4, 6, 9, 1, 2,
2, 2, 9, 0, 8, 1, 9, 1, 4, 9, 2, 7, 4, 5, 2, 0, 6, 1, 2, 2, 9, 0, 8, 1, 0, 5, 0, 2,
0, 5, 1, 0, 2, 0, 3, 7, 4, 4, 0, 1, 0, 6, 1, 7, 2, 0, 1, 7, 4, 0, 0, 9, 6, 0, 3, 0,
0, 2, 3, 7, 0, 6, 1, 5, 2, 0, 3, 9, 4, 0, 5, 1, 6, 4, 0, 0, 9, 6, 5, 8, 0, 0, 3, 5,
1, 2, 6, 6, 0, 3, 4, 9, 1, 0, 7, 1, 6, 1, 0, 8, 2, 7, 4, 1, 2, 5, 1, 7, 8, 7, 3, 0,
4, 1, 3, 1, 4, 0, 2, 0, 2, 7, 5, 3, 0, 7, 1, 5, 9, 0, 2, 1, 3, 0, 2, 0, 3, 0, 5, 0,
5, 6, 7, 6, 8, 2, 4, 0, 0, 9, 9, 1, 7, 1, 1, 4, 9, 2, 4, 7, 3, 8, 4, 3, 8, 2, 6, 3,
3, 9, 8, 5, 2, 0, 0, 4, 1, 1, 0, 0, 6, 6, 4, 9, 3, 1, 8, 8, 7, 0, 0, 9]
```

In [276…
```
erKernel = accuracy(predictedList, yTestSmall)

print("Kernel Perceptron Accuracy:")
print("Number correct: ", erKernel[0], " out of ", len(predictedList))
print("Percentage correct: ", erKernel[1], "%")
```

```
Kernel Perceptron Accuracy:
Number correct:  366  out of  500
Percentage correct:  73.2 %
```

In [277…
```
confusionKernel = getConfusion(predictedList, ynpTest)
label = np.arange(10)
cmKernel = pandas.DataFrame(confusionKernel, columns = label, index = label)
print("Kernel Perceptron Confusion Matrix: ")
cmKernel
```

Kernel Perceptron Confusion Matrix:

Out[277…

	0	1	2	3	4	5	6	7	8	9
0	50	0	0	0	0	0	0	0	0	0
1	2	56	0	0	0	0	0	0	0	0
2	4	0	43	1	1	0	0	0	1	0
3	15	2	4	30	0	0	0	1	0	1
4	7	0	0	1	21	0	0	0	2	11
5	13	0	0	0	4	22	0	0	1	0
6	10	0	2	0	0	0	37	0	0	0
7	6	1	2	1	0	0	0	40	0	5
8	10	1	0	0	1	1	0	0	34	2
9	13	0	1	1	0	0	0	5	1	33

////////// NEURAL NETWORK IMPLEMENTATION //////////

In [29]:
```
#Neural Network Training by batches
#originallly I trained all batches but this took too long to complete
#I have reduced the amount of batches due to time constraints
nPerLayer = 5
nLastLayer = 10
myNetwork = None
i = 1
for xBatch, yBatch in zip(XtrainBatches[0:200], yOneHotTrainBatches[0:200]):
    sys.stdout.write("\r       " + str(i) + "  ")
    i += 1
    sys.stdout.flush()
    nsX = xBatch.shape[0]
    nfX = xBatch.shape[1]
    #print(yTrainSmallArrays)
    if type(myNetwork) == type(None):
        myNetwork = initNN(nfX, nPerLayer, nLastLayer)
    trainNN(myNetwork, xBatch, yBatch, 0.5, 100, 10)
```

```
99     200
```

In [30]:
```
for i in range(600,700):
    fp = fwdProp(myNetwork, x[i])
    prediction = predictNN(myNetwork, x[i])
    actual = ynp[i]
    #print("\nfp: ", fp)
    #print("Predicted: ", prediction, "\tActual: ", actual)
```

In [31]:
```
preds = []
for sample in Xtest:
    preds.append(predictNN(myNetwork, sample))

er = accuracy(preds, ynpTest)
print("Neural Network Accuracy:")
print("Number correct: ", er[0], " out of ", len(preds))
print("Percentage correct: ", er[1], "%")
```

```
Neural Network Accuracy:
Number correct:  12720  out of  17999
Percentage correct:  70.67059281071171 %
```

In [32]:
```
confusionNN = getConfusion(preds, ynpTest)
label = np.arange(10)
cmNN = pandas.DataFrame(confusionNN, columns = label, index = label)
print("Neural Network Confusion Matrix: ")
cmNN
```

Neural Network Confusion Matrix:

Out[32]:

	0	1	2	3	4	5	6	7	8	9
0	1555	0	138	7	7	22	6	12	33	0
1	0	1627	48	15	4	1	12	17	110	108
2	59	15	1389	43	29	2	114	28	128	4
3	30	16	78	1486	11	60	20	34	85	43
4	0	0	12	2	1461	17	30	1	30	190
5	178	2	61	324	116	800	26	22	87	21
6	5	11	68	0	186	12	1365	9	48	34
7	3	34	56	40	37	205	127	1395	4	34
8	45	30	42	190	181	50	20	4	1093	156
9	8	16	29	116	734	125	97	25	40	549

////////// RAW DATA TO IMAGE //////////
only used for personal curiosity

In [ ]:
```
print(y[1])
print("0")
d = np.array(dta[1])
#print(d)
s = 0
imArr = []
for i in range(1,28):
    e = i*28
    #print(d[s:e])
    imArr.append((d[s:e]*255).astype(int))
    s = e + 1
imnpArr = np.array(imArr)
```

In [ ]:
```
d2 = (d*255).astype(int)
d2.resize((28,28))
#print(d2)
```

In [ ]:
```
from PIL import Image
im = Image.fromarray(d2.astype(np.uint8), "L")
im.show()
```