Assignment 5 - Scott Berry

Importing the libraries

In [1]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable
```

Importing the dataset

The y values are the binary yes/no application acceptance

The shape of the dataset is set to the num variables

In [2]:
```
dataset = pd.read_csv('Credit_Card_Applications.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

num_applications = dataset.shape[0]
num_categories = dataset.shape[1]
```

Feature Scaling

This normalizes the data to put all values between 0 and 1

In [3]:
```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0,1))
X = sc.fit_transform(X)
```

Split into train/test and convert to Torch tensors

Data is split 90/10 into Torch tensors

In [4]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
training_set = np.hstack((X_train, y_train.reshape((y_train.shape[0],1))))
test_set = np.hstack((X_test, y_test.reshape((y_test.shape[0],1))))
training_set = torch.FloatTensor(training_set)
test_set = torch.FloatTensor(test_set)
```

Create AutoEncoder Neural Network

This model inherits from Torch Neural Network with some modified values/methods

Different optimizers affect model loss due to their effect on learning rate and weights

The example on Canvas used the RMSprop optimizer (a gradient descent variant) due to ability to increase learning rate reliably and as such will be used in this notebook

The Adam optimizer is the ideal choice in most datasets, however, the main reasons being the faster compute time which can increase the number of epochs and easier parameter tuning

In [5]:
```
class SAE(nn.Module):
    def __init__(self, ):
        super(SAE, self).__init__()
        self.fc1 = nn.Linear(num_categories, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, num_categories)
        self.activation = nn.Sigmoid()
    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
        return x
sae = SAE()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay = 0.5)
```

Train the AutoEncoder

AE model is trained over 200 epochs for each value in the train set

With too few epochs loss is not minimized, too many and the model will be over-fitted

In [6]:
```
num_epoch = 200
for epoch in range(1, num_epoch + 1):
    train_loss = 0
    s = 0.
    for id_user in range(num_applications):
        try:
            input = Variable(training_set[id_user]).unsqueeze(0)
            target = input.clone()
            if torch.sum(target.data > 0) > 0:
                output = sae(input)
                target.require_grad = False
                output[target == 0] = 0
                loss = criterion(output, target)
                mean_corrector = num_categories / float(torch.sum(target.data > 0) + 1e-10)
                loss.backward()
                train_loss += np.sqrt(loss.data*mean_corrector)
                s += 1.
                optimizer.step()
        except IndexError:
            s += 1.
            optimizer.step()
    print('epoch: '+str(epoch)+' loss: '+ str(train_loss/s))
```

```
epoch: 1 loss: tensor(0.1534)
epoch: 2 loss: tensor(0.1406)
epoch: 3 loss: tensor(0.1396)
epoch: 4 loss: tensor(0.1395)
epoch: 5 loss: tensor(0.1396)
epoch: 6 loss: tensor(0.1396)
epoch: 7 loss: tensor(0.1396)
epoch: 8 loss: tensor(0.1396)
epoch: 9 loss: tensor(0.1396)
epoch: 10 loss: tensor(0.1395)
epoch: 11 loss: tensor(0.1394)
epoch: 12 loss: tensor(0.1393)
epoch: 13 loss: tensor(0.1390)
epoch: 14 loss: tensor(0.1387)
epoch: 15 loss: tensor(0.1380)
epoch: 16 loss: tensor(0.1371)
epoch: 17 loss: tensor(0.1358)
epoch: 18 loss: tensor(0.1343)
epoch: 19 loss: tensor(0.1332)
epoch: 20 loss: tensor(0.1321)
epoch: 21 loss: tensor(0.1311)
epoch: 22 loss: tensor(0.1301)
epoch: 23 loss: tensor(0.1289)
epoch: 24 loss: tensor(0.1279)
epoch: 25 loss: tensor(0.1271)
epoch: 26 loss: tensor(0.1262)
epoch: 27 loss: tensor(0.1255)
epoch: 28 loss: tensor(0.1250)
epoch: 29 loss: tensor(0.1247)
epoch: 30 loss: tensor(0.1244)
epoch: 31 loss: tensor(0.1243)
epoch: 32 loss: tensor(0.1241)
epoch: 33 loss: tensor(0.1240)
epoch: 34 loss: tensor(0.1239)
epoch: 35 loss: tensor(0.1238)
epoch: 36 loss: tensor(0.1237)
epoch: 37 loss: tensor(0.1237)
epoch: 38 loss: tensor(0.1236)
epoch: 39 loss: tensor(0.1235)
epoch: 40 loss: tensor(0.1235)
epoch: 41 loss: tensor(0.1234)
epoch: 42 loss: tensor(0.1233)
epoch: 43 loss: tensor(0.1233)
epoch: 44 loss: tensor(0.1232)
epoch: 45 loss: tensor(0.1231)
epoch: 46 loss: tensor(0.1231)
epoch: 47 loss: tensor(0.1230)
epoch: 48 loss: tensor(0.1230)
epoch: 49 loss: tensor(0.1229)
epoch: 50 loss: tensor(0.1228)
epoch: 51 loss: tensor(0.1228)
epoch: 52 loss: tensor(0.1227)
epoch: 53 loss: tensor(0.1227)
epoch: 54 loss: tensor(0.1226)
epoch: 55 loss: tensor(0.1226)
epoch: 56 loss: tensor(0.1225)
epoch: 57 loss: tensor(0.1225)
epoch: 58 loss: tensor(0.1224)
epoch: 59 loss: tensor(0.1224)
epoch: 60 loss: tensor(0.1223)
epoch: 61 loss: tensor(0.1223)
epoch: 62 loss: tensor(0.1222)
epoch: 63 loss: tensor(0.1222)
epoch: 64 loss: tensor(0.1221)
epoch: 65 loss: tensor(0.1221)
epoch: 66 loss: tensor(0.1221)
epoch: 67 loss: tensor(0.1220)
epoch: 68 loss: tensor(0.1220)
epoch: 69 loss: tensor(0.1219)
epoch: 70 loss: tensor(0.1219)
epoch: 71 loss: tensor(0.1219)
epoch: 72 loss: tensor(0.1218)
epoch: 73 loss: tensor(0.1218)
epoch: 74 loss: tensor(0.1218)
epoch: 75 loss: tensor(0.1217)
epoch: 76 loss: tensor(0.1217)
epoch: 77 loss: tensor(0.1217)
epoch: 78 loss: tensor(0.1217)
epoch: 79 loss: tensor(0.1216)
epoch: 80 loss: tensor(0.1216)
epoch: 81 loss: tensor(0.1216)
epoch: 82 loss: tensor(0.1216)
epoch: 83 loss: tensor(0.1215)
epoch: 84 loss: tensor(0.1215)
epoch: 85 loss: tensor(0.1215)
epoch: 86 loss: tensor(0.1215)
epoch: 87 loss: tensor(0.1215)
epoch: 88 loss: tensor(0.1214)
epoch: 89 loss: tensor(0.1214)
epoch: 90 loss: tensor(0.1214)
epoch: 91 loss: tensor(0.1214)
epoch: 92 loss: tensor(0.1214)
epoch: 93 loss: tensor(0.1213)
epoch: 94 loss: tensor(0.1213)
epoch: 95 loss: tensor(0.1213)
epoch: 96 loss: tensor(0.1213)
epoch: 97 loss: tensor(0.1213)
epoch: 98 loss: tensor(0.1213)
epoch: 99 loss: tensor(0.1212)
epoch: 100 loss: tensor(0.1212)
epoch: 101 loss: tensor(0.1212)
epoch: 102 loss: tensor(0.1212)
epoch: 103 loss: tensor(0.1212)
epoch: 104 loss: tensor(0.1212)
epoch: 105 loss: tensor(0.1212)
epoch: 106 loss: tensor(0.1211)
epoch: 107 loss: tensor(0.1211)
epoch: 108 loss: tensor(0.1211)
epoch: 109 loss: tensor(0.1211)
epoch: 110 loss: tensor(0.1211)
epoch: 111 loss: tensor(0.1211)
epoch: 112 loss: tensor(0.1211)
epoch: 113 loss: tensor(0.1211)
epoch: 114 loss: tensor(0.1211)
epoch: 115 loss: tensor(0.1211)
epoch: 116 loss: tensor(0.1210)
epoch: 117 loss: tensor(0.1210)
epoch: 118 loss: tensor(0.1210)
epoch: 119 loss: tensor(0.1210)
epoch: 120 loss: tensor(0.1210)
epoch: 121 loss: tensor(0.1210)
epoch: 122 loss: tensor(0.1210)
epoch: 123 loss: tensor(0.1210)
epoch: 124 loss: tensor(0.1210)
epoch: 125 loss: tensor(0.1209)
epoch: 126 loss: tensor(0.1209)
epoch: 127 loss: tensor(0.1209)
epoch: 128 loss: tensor(0.1209)
epoch: 129 loss: tensor(0.1209)
epoch: 130 loss: tensor(0.1209)
epoch: 131 loss: tensor(0.1209)
epoch: 132 loss: tensor(0.1209)
epoch: 133 loss: tensor(0.1209)
epoch: 134 loss: tensor(0.1208)
epoch: 135 loss: tensor(0.1208)
epoch: 136 loss: tensor(0.1208)
epoch: 137 loss: tensor(0.1208)
epoch: 138 loss: tensor(0.1208)
epoch: 139 loss: tensor(0.1208)
epoch: 140 loss: tensor(0.1208)
epoch: 141 loss: tensor(0.1208)
epoch: 142 loss: tensor(0.1207)
epoch: 143 loss: tensor(0.1207)
epoch: 144 loss: tensor(0.1207)
epoch: 145 loss: tensor(0.1207)
epoch: 146 loss: tensor(0.1207)
epoch: 147 loss: tensor(0.1207)
epoch: 148 loss: tensor(0.1207)
epoch: 149 loss: tensor(0.1207)
epoch: 150 loss: tensor(0.1206)
epoch: 151 loss: tensor(0.1206)
epoch: 152 loss: tensor(0.1206)
epoch: 153 loss: tensor(0.1206)
epoch: 154 loss: tensor(0.1206)
epoch: 155 loss: tensor(0.1206)
epoch: 156 loss: tensor(0.1206)
epoch: 157 loss: tensor(0.1206)
epoch: 158 loss: tensor(0.1205)
epoch: 159 loss: tensor(0.1205)
epoch: 160 loss: tensor(0.1205)
epoch: 161 loss: tensor(0.1205)
epoch: 162 loss: tensor(0.1205)
epoch: 163 loss: tensor(0.1205)
epoch: 164 loss: tensor(0.1204)
epoch: 165 loss: tensor(0.1204)
epoch: 166 loss: tensor(0.1204)
epoch: 167 loss: tensor(0.1204)
epoch: 168 loss: tensor(0.1204)
epoch: 169 loss: tensor(0.1203)
epoch: 170 loss: tensor(0.1203)
epoch: 171 loss: tensor(0.1203)
epoch: 172 loss: tensor(0.1203)
epoch: 173 loss: tensor(0.1203)
epoch: 174 loss: tensor(0.1202)
epoch: 175 loss: tensor(0.1202)
epoch: 176 loss: tensor(0.1202)
epoch: 177 loss: tensor(0.1202)
epoch: 178 loss: tensor(0.1201)
epoch: 179 loss: tensor(0.1201)
epoch: 180 loss: tensor(0.1201)
epoch: 181 loss: tensor(0.1200)
epoch: 182 loss: tensor(0.1200)
epoch: 183 loss: tensor(0.1200)
epoch: 184 loss: tensor(0.1199)
epoch: 185 loss: tensor(0.1199)
epoch: 186 loss: tensor(0.1199)
epoch: 187 loss: tensor(0.1198)
epoch: 188 loss: tensor(0.1198)
epoch: 189 loss: tensor(0.1197)
epoch: 190 loss: tensor(0.1197)
epoch: 191 loss: tensor(0.1196)
epoch: 192 loss: tensor(0.1196)
epoch: 193 loss: tensor(0.1195)
epoch: 194 loss: tensor(0.1194)
epoch: 195 loss: tensor(0.1194)
epoch: 196 loss: tensor(0.1193)
epoch: 197 loss: tensor(0.1192)
epoch: 198 loss: tensor(0.1191)
epoch: 199 loss: tensor(0.1190)
epoch: 200 loss: tensor(0.1189)
```

Test the AutoEncoder

The testing computes a total test loss value by comparing the test set to output of the AE model

The relatively low test loss result indicates that this classifier can reliably predict which applications will be approved/disapproved

The frauds in the test set are outputted based on approvals that SHOULD have been failures set at a threshold of 0.08 loss

In [7]:
```
test_loss = 0
s = 0.
frauds = []
for id_user in range(num_applications):
    try:
        input = Variable(training_set[id_user]).unsqueeze(0)
        target = Variable(test_set[id_user]).unsqueeze(0)
        if torch.sum(target.data > 0) > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            if loss > 0.08:
                frauds.append(id_user)
            mean_corrector = num_categories / float(torch.sum(target.data > 0) + 1e-10)
            test_loss += np.sqrt(loss.data*mean_corrector)
            s += 1.
    except IndexError:
        s += 1.
print('test loss: '+str(test_loss/s))
print("frauds: " + str(frauds))
```

```
test loss: tensor(0.0179)
frauds: [19]
```

Create the Boltzmann Machine

This RBM class is created by specifying weights, hidden and visible nodes

Further, the training method is specified here with batches of 100

In [8]:
```
class RBM:
    def __init__(self, nv, nh):
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)
    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)
    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)
    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)).t()
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)
nv = len(training_set[0])
nh = 100
batch_size = 100
rbm = RBM(nv, nh)
```

Train the Boltzmann Machine

The RBM model is trained over the batches of the training set for the length of the training set

In [9]:
```
num_epoch = 7
for epoch in range(1, num_epoch + 1):
    train_loss = 0
    s = 0.
    for id_user in range(0, num_applications - batch_size, batch_size):
        vk = training_set[id_user : id_user + batch_size]
        v0 = training_set[id_user : id_user + batch_size]
        ph0,_ = rbm.sample_h(v0)
        for k in range(10):
            _,hk = rbm.sample_h(vk)
            _,vk = rbm.sample_v(hk)
            vk[v0<0] = v0[v0<0]
        phk,_ = rbm.sample_h(vk)
        rbm.train(v0, vk, ph0, phk)
        train_loss += torch.mean(torch.abs(v0[v0 >= 0] - vk[v0 >= 0]))
        s += 1.
    print('epoch: '+str(epoch)+' loss: '+str(train_loss/s))
```

```
epoch: 1 loss: tensor(0.4082)
epoch: 2 loss: tensor(0.3777)
epoch: 3 loss: tensor(0.3775)
epoch: 4 loss: tensor(0.3653)
epoch: 5 loss: tensor(0.3651)
epoch: 6 loss: tensor(0.3731)
epoch: 7 loss: tensor(0.3726)
```

Test the Boltzmann Machine

The testing set is compared to the RBM model

The high test loss result is indicative of the model not performing as well as the AE model

Each fraud is printed when loss is found at a threshold of 0.50

In [10]:
```
test_loss = 0
s = 0.
frauds = []
for id_user in range(num_applications):
    v = training_set[id_user:id_user+1]
    vt = test_set[id_user:id_user+1]
    if len(vt[vt>=0]) > 0:
        _,h = rbm.sample_h(v)
        _,v = rbm.sample_v(h)
        if torch.mean(torch.abs(vt[vt>=0] - v[vt>=0])) > 0.50:
            frauds.append(id_user)
        test_loss += torch.mean(torch.abs(vt[vt>=0] - v[vt>=0]))
        s += 1.
print('test loss: '+str(test_loss/s))
print("frauds: " + str(frauds))
```

```
test loss: tensor(0.3447)
frauds: [1, 23, 43]
```