Lab 2: Overfitting and Underfitting with Neural Networks

CSC 592: Machine Learning Security and Privacy

**Background**

In machine learning, understanding underfitting and overfitting is important for building machine learning models that perform well on new (unseen) data. Underfitting occurs when a model is too simple to capture the underlying patterns in the data, leading to poor performance on both training and test sets. On the other hand, overfitting occurs when a model memorizes the training data rather than the true patterns in the data. Overfitting occurs due to over training which results in excellent performance on training data but poor performance to the test set. Overfitting is common when models are too complex relative to the amount of training data available. To achieve an optimal balance, techniques like regularization are used to ensure the model does not overfit.

In this lab, you will create an object-oriented implementation of a deep linear network (from scratch) using the provided starter code. You will then experiment with model complexity to see the effects of underfitting and overfitting when training a machine learning model.

**Step by Step Guide**

1. Create a Python application called NNDecisionBoundary. Add a file called Utils.py with the following code in it.

import numpy as np

from sklearn import datasets, linear\_model

import matplotlib.pyplot as plt

**class Utils(object):**

def init\_data(self):

# generate a random dataset and plot it

np.random.seed(0)

X, y = datasets.make\_moons(200, noise=0.20)

plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)

plt.show()

return X, y

def normalize\_data(self,X):

min = np.min(X, axis = 0)

max = np.max(X, axis = 0)

normX = 1 - ((max - X)/(max-min))

return normX

2. Add a class called ActivationType.py with the following code in it:

**class ActivationType(object):**

NONE = 0

SIGMOID = 1

TANH = 2

RELU = 3

SOFTMAX = 4

3. Add a class called GradDescType.py with the following code in it:

**class GradDescType(object):**

STOCHASTIC = 1

BATCH = 2

MINIBATCH = 3

4. Add a class called Layer.py with the following code in it. This class contains the neurons, their activation functions, biases, and weights for a single layer in the neural network. Examine the code carefully to see how a single layer in a neural network is put together. The forward function does the forward pass through the layer.

import numpy as np

from ActivationType import ActivationType

**class Layer(object):** # represents one layer of neurons in an NN

def \_\_init\_\_(self,num\_neurons,num\_neurons\_prev\_layer, last\_layer= False,drop\_out = 0.2,activation\_type=ActivationType.SIGMOID):

self.num\_neurons = num\_neurons

self.last\_layer = last\_layer

self.numNeurons\_pre\_Layer = num\_neurons\_prev\_layer

self.activation\_function = activation\_type

self.drop\_out = drop\_out

self.delta = np.zeros((num\_neurons,1))

self.a = np.zeros((num\_neurons,1)) # actual output from layer

self.derivAF = np.zeros((num\_neurons,1)) # derivative of Activation function

self.W = np.random.randn(num\_neurons, num\_neurons\_prev\_layer)/ np.sqrt(num\_neurons\_prev\_layer) # initialize weight matrix randomly

self.b = np.zeros((num\_neurons,1))

self.WGrad = np.zeros((num\_neurons,num\_neurons\_prev\_layer))

self.bGrad = np.zeros((num\_neurons,1)) # gradient for biases

**def forward(self,input\_data):**

sum = np.dot(self.W,input\_data) + self.b

aa = 0

if (self.activation\_function == ActivationType.NONE):

self.a = sum

self.derivAF = 1

if (self.activation\_function == ActivationType.SIGMOID):

self.a = self.sigmoid(sum)

self.derivAF = self.a \* (1 - self.a)

if (self.activation\_function == ActivationType.TANH):

self.a = self.TanH(sum)

self.derivAF = (1 - self.a\*self.a)

if (self.activation\_function == ActivationType.RELU):

self.a = self.Relu(sum)

self.derivAF = 1.0 \* (self.a > 0)

if (self.activation\_function == ActivationType.SOFTMAX):

self.a = self.Softmax(sum)

self.derivAF = None # we do delta computation in Softmax layer

if (self.last\_layer == False):

zeroout = np.random.binomial(1,self.drop\_out,(self.num\_neurons,1))/self.drop\_out

# which neurons to zero out

self.a = self.a \* zeroout

self.derivAF = self.derivAF \* zeroout

return self.a

**def Linear(self,x):**

return x # output same as input

**def sigmoid(self,x):**

return 1 / (1 + np.exp(-x)) # np.exp makes it operate on entire array

**def TanH(self, x):**

return np.tanh(x)

**def Relu(self, x):**

return np.maximum(0,x)

**def Softmax(self, x):**

ex = np.exp(x)

return ex/ex.sum()

**def clear\_WBgradients(self):** # zero out weight and bias gradients

self.WGrad = 0

self.bGrad = 0

5. Add a class called Network to the project with the following code in it. The Network class assembles the layers and provides the train forward function. The forward function passes the data through all the layers.

import math

import numpy as np

from Layer import \*

from GradDescType import \*

from sklearn.utils import shuffle

**class Network(object):**

def \_\_init\_\_(self,X,Y,num\_layers,drop\_out = 0.0, activationF=ActivationType.SIGMOID,last\_layerAF= ActivationType.SOFTMAX):

self.X = X

self.Y = Y

self.num\_layers = num\_layers

self.Layers = [] # network contains list of layers

self.last\_layerAF = last\_layerAF

for i in range(len(num\_layers)):

if (i == 0): # first layer

layer = Layer(num\_layers[i],X.shape[1],False,drop\_out, activationF)

elif (i == len(num\_layers)-1): # last layer

layer = Layer(Y.shape[1],num\_layers[i-1],True,drop\_out, last\_layerAF)

else: # intermediate layers

layer = Layer(num\_layers[i],num\_layers[i-1],False,drop\_out, activationF)

self.Layers.append(layer)

def forward(self,input\_data): # evaluates all layers

out\_layer = self.Layers[0].forward(input\_data)

for i in range(1,len(self.num\_layers)):

out\_layer = self.Layers[i].forward(out\_layer)

return out\_layer

**def Train(self, epochs,learningRate, lambda1, gradDescType, batchSize=1):**

for j in range(epochs):

loss = 0

self.X, self.Y = shuffle(self.X, self.Y, random\_state=0)

zz = self.X.shape[0]

for i in range(self.X.shape[0]):

aout = self.forward(self.X[i]) # actual output of network

if (self.last\_layerAF == ActivationType.SOFTMAX):

loss += -(self.Y[i] \* np.log(aout)).sum()

else:

loss += ((aout - self.Y[i]) \* \

(aout - self.Y[i])).sum()

lnum = len(self.num\_layers)-1 # last layer number

# compute deltas, grads on all layers

while(lnum >= 0):

if (lnum == len(self.num\_layers)-1): # last layer

if (self.last\_layerAF == ActivationType.SOFTMAX):

self.Layers[lnum].delta = -self.Y[i]+ self.Layers[lnum].a

else:

self.Layers[lnum].delta = -(self.Y[i]-self.Layers[lnum].a) \* self.Layers[lnum].derivAF

else: # intermediate layer

self.Layers[lnum].delta = np.dot(self.Layers[lnum+1].W.T,self.Layers[lnum+1].delta) \* self.Layers[lnum].derivAF

if (lnum > 0): #previous output

prevOut = self.Layers[lnum-1].a

else:

prevOut = self.X[i]

self.Layers[lnum].WGrad += np.dot(self.Layers[lnum].delta,prevOut.T)

self.Layers[lnum].bGrad += self.Layers[lnum].delta

lnum = lnum - 1

if (gradDescType == GradDescType.MINIBATCH):

if (i % batchSize == 0):

self.UpdateGradsBiases(learningRate,lambda1, batchSize)

if (gradDescType == GradDescType.STOCHASTIC):

self.UpdateGradsBiases(learningRate,lambda1, 1)

if (gradDescType == GradDescType.BATCH):

self.UpdateGradsBiases(learningRate,lambda1, self.X.shape[0])

print("Iter = " + str(j) + " Loss = "+ str(loss))

**def UpdateGradsBiases(self, learningRate, lambda1, batchSize):**

# update weights and biases for all layers

for ln in range(len(self.num\_layers)):

self.Layers[ln].W = self.Layers[ln].W - learningRate \* (1/batchSize) \* self.Layers[ln].WGrad

- learningRate \* lambda1 \* self.Layers[ln].W

self.Layers[ln].b = self.Layers[ln].b - learningRate \* (1/batchSize) \* self.Layers[ln].bGrad

self.Layers[ln].clear\_WBgradients()

6. The main code will be in NNDecisionBoundary.py. Type the following code in it:

import sys

from Network import Network

from GradDescType import \*

from ActivationType import \*

from Utils import Utils

import numpy as np

import matplotlib.pyplot as plt

def plot\_decision\_boundary(pred\_func, X, y):

# Set min and max values and give it some padding

x\_min, x\_max = X[:, 0].min() - .5, X[:, 0].max() + .5

y\_min, y\_max = X[:, 1].min() - .5, X[:, 1].max() + .5

h = 0.01

# Generate a grid of points with distance h between them

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

# Predict the function value for the whole gid

xdata = np.c\_[xx.ravel(), yy.ravel()]

xdatanp = xdata.reshape(xdata.shape[0],xdata.shape[1],1)

#print(xdatanp.shape)

Z = [pred\_func(xdatanp[x]) for x in range(0,len(xdatanp))]

Z = np.array(Z)

exp\_scores = np.exp(Z)

probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)

Z = np.argmax(probs, axis=1)

Z = Z.reshape(xx.shape)

# Plot the contour and training examples

plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)

plt.show()

**def main():**

utils = Utils()

X, Y = utils.init\_data() # initialize data

#X = utils.normalizeData(X)

trainX = X.reshape((X.shape[0],X.shape[1],1))

trainY = np.zeros((len(Y),2))

for i in range(0,len(Y)):

if Y[i] == 1:

trainY[i,0] = 1

trainY[i,1] = 0

else:

trainY[i,0] = 0

trainY[i,1]= 1

trainY = trainY.reshape((X.shape[0],X.shape[1],1))

numLayers = [30,2]

NN = Network(trainX,trainY,numLayers,1.0,ActivationType.TANH, ActivationType.SOFTMAX) # try different activation functions

NN.Train(1000,0.10,0.0, GradDescType.MINIBATCH,1)

#------------ compute accuracy----------

accuracy = 0

for i in range(len(trainX)):

pred = NN.forward(trainX[i])

if (pred.argmax() == 0 and trainY[i,0,0] == 1) or \

(pred.argmax() == 1 and trainY[i,0,0] == 0):

accuracy = accuracy + 1

accuracy\_percent = accuracy/len(trainX)

print('accuracy =', accuracy\_percent)

plot\_decision\_boundary(lambda x: NN.forward(x), X, Y)

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(int(main() or 0))

As you can see from the above code, the main function creates a network and specifies the neurons via a list called numLayers. For example, the above code sets the number of neurons to 30 in the first layer, and 2 in the last layer. For the moon dataset, the input and output is two dimensional so that we can visualize the decision boundaries.

For this assignment, your goal is to understand the effect of number of layers, number of neurons in the hidden layers, different activation types, effect of stochastic vs minibatch, and overfitting, underfitting of the model.

**Lab Assignment**

**Exercise 1:** Set the number of neurons in the hidden layer of the network to be 1. Train the network. Take a screenshot of the figure for submission. Answer the following question: based on the figure, is this model overfitting, underfitting or just right for the data? Hint: the code output should look similar to what is shown below.

A screen shot of a graph

Description automatically generated

A black screen with a black border

Description automatically generated

**Exercise 2:** Set the number of neurons in the hidden layer of the network to be 3. Train the network. Take a screenshot of the figure for submission. Answer the following question: based on the figure, is this model overfitting, underfitting or just right for the data? Hint: the code output should look similar to what is shown below.

A screen shot of a graph

Description automatically generated

A screenshot of a computer

Description automatically generated

**Exercise 3:** Set the number of neurons in the hidden layer of the network to be 30. Train the network. Take a screenshot of the figure for submission. Answer the following question: based on the figure, is this model overfitting, underfitting or just right for the data? Hint: the code output should look similar to what is shown below.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

**Exercise 4:** In the previous exercises we did not reach 100% training accuracy. Train the network for 5000 epochs with 30 neurons in the hidden layer. Take a screenshot of the figure for submission. Answer the following question: based on the figure, is this model overfitting, underfitting or just right for the data? Hint: the code output should look similar to what is shown below.

A screenshot of a computer screen

Description automatically generated

A black rectangular object with a black background

Description automatically generated

**Deliverables**

Submit the following two documents on Brightspace:

Deliverable #1: A screenshot of the output for each of the four exercises running on your machine. In this document also include an answer to each of the exercise questions about underfitting/overfitting.

Deliverable #2: A copy of your code (the .py files).