ECE472 – Project 2

Jonathan Lam

September 16, 2020

Notes on model development

Initialization Like in the last project, initialization was done with Gaussians. The only problems I had were that if I initialized the ReLUs with zero-centered bias matrices, then the convergence appeared to be slower (most likely due to many "dead" ReLUs), so I initialized them with a largely-positive bias (`layers[i]['bias']`). I also realized that as the number of layers increased, I had to downscale the initial weights for the activation functions so that there weren't convergence issues (`layers[i]['coef']`).

Layer widths At first, I had relatively low layer widths. Following the example of the textbook, which used a two-layer perceptron to model XOR, where the hidden layer had a width of 2, I tried widths of 2, 4, and 8. This didn't classify all of the points correctly (it seemed to converge, but not near zero), so at this point I added another layer, which only helped a little bit. Only when I had five layers (four with width 10, the last one with width 1) was I able to make the model converge.

After speaking with some classmates (Derek Lee), I learnt that even this was considered a low number, and I experimented with larger sizes, which in general tended to converge faster. The final values I chose (32, 64, 32, 1) are somewhat arbitrary but seem to converge quickly ($<$ 1000 iterations with Adam). Using smaller values (e.g., 16, 16, 16, 1) I can still classify all of the points correctly, but occasionally there are classification errors with only 1000 iterations and would benefit with more iterations, and the boundary is not as smooth.

Layer count Initially, I began with three ReLU layers and one sigmoid layer (this is the same as my final). I only tried up to five layers (adding one additional ReLU), which helped a little when the widths of the layers were small.

Update rule, iteration count, and dynamic step sizes Most of my experimentation worked with a basic gradient descent algorithm with a step size that was manually attenuated with higher iterations, e.g., $\alpha \leftarrow \alpha/2$ every 1000 iterations. This eventually converged (most of the time) without problems, but there were generally many irregular spikes on the loss vs. iterations plot. Typical convergence took 5000-10000 iterations using this scheme.

After reading the paper on Adam, I wanted to see how it would affect my model. I wasn't expecting much, but this did wonders right away on the convergence (both the smoothness and convergence rate): the same model only required 500-1000 iterations to converge. The loss vs. iteration count plot immediately became smooth.

The end result is a four-layer perceptron using batch gradient descent with the Adam update rule, with three ReLU layers with widths 32, 64, and 32, followed by a sigmoid layer to map onto the probability space. The loss function is a binary cross-entropy function with L2 penalties for the weight matrices W_i (but not the bias matrices b_i). 1000 iterations are sufficient to train this model, which runs in approximately 6s on an i7-7500U processor with integrated graphics.

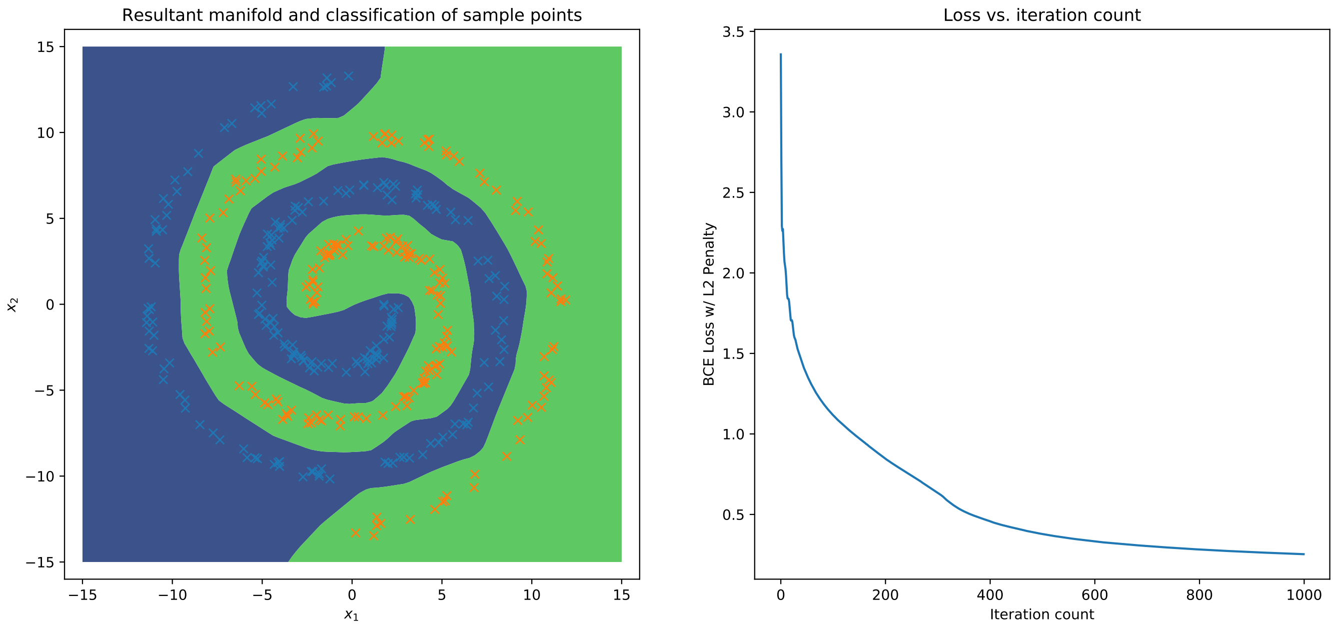
Source code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt


### DEFINE MODEL

# activation function definitions
def relu(X, W, b):
    return tf.nn.relu(tf.transpose(W) @ X + b)

def sigmoid(X, W, b):
    return tf.math.sigmoid(tf.transpose(W) @ X + b)

# multi-layer perceptron classifier model definition
class MultiLayerPerceptronClassifier:

    def __init__(self,
                 sampleDim,          # dimensionality of input
                 layers,             # layer definitions
                 lmbda=0.01,         # l2 penalty coefficient
                 beta1=0.9,          # decay factor for first moment (adam)
                 beta2=0.999,        # decay factor for second moment (adam)
                 alpha=0.001):       # upper bound on learning rate (adam)

        self._lmbda = lmbda
        self._beta1 = beta1
        self._beta2 = beta2
        self._alpha = alpha
        self._layers = layers

        # model variables to optimize; initialize with normal distribution
        self._modelVars = [
            {
                'W': tf.Variable(tf.random.normal(
                    (layers[i-1]['width'] if i > 0 else sampleDim,
                    layer['width']), dtype=tf.float64) * layer['coef']),
                'b': tf.Variable(tf.random.normal((layer['width'], 1),
                    dtype=tf.float64) * layer['coef'] + layer['bias'])
            } for i, layer in enumerate(layers)
        ]

        # adam variables: weighted first and second moments of gradients
        self._adamVars = [
            {
                'zW': tf.zeros_like(layerVars['W']),
                'zb': tf.zeros_like(layerVars['b']),
                'zW2': tf.zeros_like(layerVars['W']),
                'zb2': tf.zeros_like(layerVars['b']),
            } for layerVars in self._modelVars
        ]

    def f(self, X):
        for layer, layerVars in zip(self._layers, self._modelVars):
            X = layer['actFn'](X, layerVars['W'], layerVars['b'])
        return X

    # binary cross-entropy loss with L2 penalty
    def bceLossL2Penalty(self, y, yhat):
        loss = -y * tf.math.log(yhat) - (1 - y) * tf.math.log(1 - yhat)
        for layerVars in self._modelVars:
            loss += self._lmbda * tf.nn.l2_loss(layerVars['W'])
        return loss

    def step(self, X, y):
        # don't do random batches, just use entire input on every step
        with tf.GradientTape() as tape:
            loss = self.bceLossL2Penalty(y, self.f(X))

        # calculate gradient, store loss
        grad = tape.gradient(loss, self._modelVars)
        self._losses.append(tf.math.reduce_mean(loss))

        # update model and adam variables
        for adam, layerVars, layerGrad \
            in zip(self._adamVars, self._modelVars, grad):

            # update adam moments
            adam['zW'] = self._beta1 * adam['zW'] \
                + (1 - self._beta1) * layerGrad['W']
            adam['zb'] = self._beta1 * adam['zb'] \
                + (1 - self._beta1) * layerGrad['b']
            adam['zW2'] = self._beta2 * adam['zW2'] \
                + (1 - self._beta2) * layerGrad['W']**2
            adam['zb2'] = self._beta2 * adam['zb2'] \
                + (1 - self._beta2) * layerGrad['b']**2

            # adam update rule
            ep = 0.0001             # epsilon to prevent division by zero
            layerVars['W'].assign_sub(self._alpha * adam['zW'] \
                / (tf.math.sqrt(adam['zW2']) + ep))
            layerVars['b'].assign_sub(self._alpha * adam['zb'] \
                / (tf.math.sqrt(adam['zb2']) + ep))

    def train(self, X, y, iterations):
        self._losses = []
        for i in range(iterations):
            self.step(X, y)
        return self._losses


### GENERATE SAMPLE DATA

# spiral definition
offset = 2                          # offset of spirals from center (radially)
```

```
N = 200                                # sample count
noiseStd = 0.25                        # sample noise (radially)
spiralEnd = 3.5 * np.pi                # spiral end (radians)

# generate spirals
t = tf.random.uniform((N,), 0., spiralEnd, dtype=tf.float64)
noise = tf.random.normal((2*N,), 0, noiseStd, dtype=tf.float64)
x1 = (t + noise[:N] + offset) * tf.math.cos(-t)
y1 = (t + noise[:N] + offset) * tf.math.sin(-t)
x2 = (t + noise[N:] + offset) * tf.math.cos(-t + np.pi)
y2 = (t + noise[N:] + offset) * tf.math.sin(-t + np.pi)

# samples of zeroes and ones; sample matrices include both inputs and labels
S0 = tf.concat((x1[tf.newaxis], y1[tf.newaxis],
                tf.zeros((1, N), dtype=tf.float64)), axis=0)
S1 = tf.concat((x2[tf.newaxis], y2[tf.newaxis],
                tf.ones((1, N), dtype=tf.float64)), axis=0)
S = tf.concat((S0, S1), axis=1)

# samples predictor matrices, label matrices
X = S[0:2, :]
y = S[2, :][tf.newaxis]


### CREATE AND RUN MODEL

# configure layers, widths, functions; width is the number of outputs for a fn,
# number of inputs inferred from last layer's width (or sample's dimensions)
layers = [
    {'actFn': relu, 'width': 32, 'coef': 0.25, 'bias': 0.5},
    {'actFn': relu, 'width': 64, 'coef': 0.25, 'bias': 0.5},
    {'actFn': relu, 'width': 32, 'coef': 0.25, 'bias': 0.5},
    {'actFn': sigmoid, 'width': 1, 'coef': 0.25, 'bias': 0.},
]
classifier = MultiLayerPerceptronClassifier(2, layers)
losses = classifier.train(X, y, 1000)


### PLOT RESULTS

# plot spirals
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

res = classifier.f(X)[0]
x1 = X[0][res<0.5]
y1 = X[1][res<0.5]
x2 = X[0][res>=0.5]
y2 = X[1][res>=0.5]

# plot original sample points
ax1.plot(x1, y1, 'x', x2, y2, 'x')

# plot manifold
x1, x2 = np.meshgrid(np.linspace(-15, 15, 100), np.linspace(-15, 15, 100))
yhat = np.reshape(classifier.f(np.vstack((x1.flatten(), x2.flatten()))),
                  (100, 100))
ax1.contourf(x1, x2, yhat, 1, vmin=0, vmax=1)
ax1.set_ylim([-16, 16])
```

```
ax1.set_xlim([-16, 16])
ax1.set_ylabel('$x_2$')
ax1.set_xlabel('$x_1$')
ax1.set_title('Resultant manifold and classification of sample points')

# plot losses vs. iteration count
ax2.plot(losses)
ax2.set_ylabel('BCE Loss w/ L2 Penalty')
ax2.set_xlabel('Iteration count')
ax2.set_title('Loss vs. iteration count')
plt.show()
```

Plots



The blue crosses are randomly generated samples from one class, and the orange crosses are randomly generated samples from the other class. The dark blue and green regions are where the model predicts that points will fall into those respective classes. All of the points are correctly classified. The model does not look too overfitted, and there is a reasonable margin between the two sample sets. The loss vs. iteration count is smooth and monotonically decreasing, which shows a nice convergence.