q1.py

```python
#Q1
# (A)
# # Generate training data and labels
# training_w1_data = multivariate_normal(mean_w1, covar_w1, size=(50))
# training_w1_labels = np.zeros(training_w1_data.shape[0])
# training_w2_data = multivariate_normal(mean_w2, covar_w2, size=(50))
# training_w2_labels = np.ones(training_w2_data.shape[0])
#
# # Generate test data and labels
# test_w1_data = multivariate_normal(mean_w1, covar_w1, size=(20))
# test_w1_labels = np.zeros(test_w1_data.shape[0])
# test_w1_labels = test_w1_labels.reshape((-1, 1))
# test_w2_data = multivariate_normal(mean_w2, covar_w2, size=(20))
# test_w2_labels = np.ones(test_w2_data.shape[0])
# test_w2_labels = test_w2_labels.reshape((-1, 1))
# training_data = np.r_[training_w1_data, training_w2_data]
# training_labels = np.r_[training_w1_labels, training_w2_labels]
# training = unison_shuffled_copies(training_data, training_labels)
# training[1] = training[1].reshape((-1, 1))
# test_data = np.r_[test_w1_data, test_w2_data]
# test_labels = np.r_[test_w1_labels, test_w2_labels]
# test = unison_shuffled_copies(test_data, test_labels)
# test[1] = test[1].reshape((-1, 1))

# (B) Best validation error achieved: 22%
#     n*_H = 4
# (C) 10%
# (D) In part B, we fit the data too strongly to the training data. In doing so,
#     the validation accuracy decreased.
#     In Part C, we stopped training at the minimum value, causing us to avoid
#     overfitting.


import numpy as np, math
import argparse
import pickle
import matplotlib

matplotlib.use('agg')
from matplotlib import pyplot as plt

import os, sys
from tensorflow.examples.tutorials.mnist import input_data

alpha = 0.1


class NeuralNetwork:
    def __init__(self, *args, dropout=False):
        # Layers of the neural network (doesn't include an input layer)
        self.loss = []
        self.layers = []

        self.image_count = 0

        current_wire = None
        for i, (layerType, units) in enumerate(*args):
            # For each layer defined, create the layer and attach it to the output wires
            # of the preceding layer

            if i == 0:
                # Create wires going from inputs to first hidden layer
                current_wire = Wire(units)
            else:
                new_layer = {
                    'sigmoid': sigmoidLayer(current_wire, units, dropout=dropout),
                    'softmax': softmaxLayer(current_wire, units, dropout=dropout),
                    'relu': reluLayer(current_wire, units, dropout=dropout)
                }.get(layerType, None)
                current_wire = new_layer.outputWire
                self.layers.append(new_layer)

    def train_and_validate(self, train_data, validation_data, epochs):
        global alpha
        alpha = 0.1
        for epoch in range(epochs):
            print("Epoch:", epoch)
            self.train(train_data)
            self.test(validation_data)
            alpha *= 1

    def train(self, data):
        train_loss = []
        for i, (input, label) in enumerate(zip(data[0], data[1])):
            output = self.forward_pass(input, training=True)
            errorGradient = output-label  # TODO: Modify this when gradient calculation is resolved
            errorGradient = errorGradient.reshape((-1, 1))
            print("training", "output:", output, "label:", label, "errorGradient:", errorGradient)
            train_loss.append(-label.T.dot(output))
            self.backward_pass(errorGradient)
```

```
        self.loss.append(sum(train_loss) / float(len(train_loss)))
        plt.clf()
        plt.plot(self.loss)
        #         plt.savefig('images/training_' + str(self.image_count) + '.png')
        self.image_count += 1

    def test(self, data):
        correct_count = 0
        for i, (input, label) in enumerate(zip(data[0], data[1])):

            output = self.forward_pass(input)
            print(input, output, label)
            if np.round(output) == label:
                correct_count += 1

        print(correct_count / data[0].shape[0])
        print()

    def forward_pass(self, input, training=False):
        current_output = input
        for i, layer in enumerate(self.layers):
            current_output = layer.forward(current_output, training=training)
        return current_output

    def calculate_error(self, prediction, label):
        return 0.5 * (prediction - label) ** 2  # TODO: Modify when gradient issue resolved

    def backward_pass(self, errorGradient):
        for i, layer in enumerate(self.layers[::-1]):
            if i == 0:
                layer.outputWire.gradients = errorGradient
                layer.backward()  # TODO: Modify when gradient issue resolved
            else:
                layer.backward()

    def saveModel(self):
        pickle.dump(self, open("model", "wb"))

    @staticmethod
    def load_model():
        return pickle.load(open("model", "rb"))


class Wire:
    def __init__(self, input_dimensions):
        self.input_dimensions = input_dimensions

    def initialize(self, units, fn):
        self.dropout_proba = 0.5
        self.units = units
        self.weights = fn(self.input_dimensions + 1, units)
        self.gradients = None
        self.velocity = np.zeros(self.weights.shape)

    def generate_dropout_variables(self):
        self.dropout_vector = np.random.choice([0, 1], size=(self.units,),
                                               p=[self.dropout_proba, 1 - self.dropout_proba])
        self.dropout_matrix = np.diag(self.dropout_vector)


class Layer:
    def __init__(self, inputWire, units, dropout=False, weight_fn=lambda x, y: np.random.randn(x, y) / np.sqrt(x)):
        if not inputWire:
            raise TypeError("Must initialize layer with Wire")
        self.dropout = dropout
        self.inputWire = inputWire
        self.inputWire.initialize(units, fn=weight_fn)

        self.outputWire = Wire(units)

    def forward(self, x, training=False):
        raise NotImplementedError("Must implement forward-pass function -- forward( self,  )")

    def backward(self):
        raise NotImplementedError("Must implement backward-pass function -- backward( self,  )")


class sigmoidLayer(Layer):
    def __init__(self, inputWire, units, dropout=False):
        super().__init__(inputWire, units, dropout)

    def forward(self, x, training=False):
        self.inputs = x
        # Add input value of "1" to the input array for the bias weight (w_0*x_0 + ...)
        self.inputs = np.insert(x, 0, 1.0, 0)
        self.linear_outputs = self.inputWire.weights.T.dot(self.inputs)

        self.outputs = self.sigmoid(self.linear_outputs)
        if training and self.dropout:
            self.inputWire.generate_dropout_variables()
            # self.outputs = self.outputs.dot(self.inputWire.dropout_matrix)
            self.outputs = self.outputs * self.inputWire.dropout_vector / self.inputWire.dropout_proba
        return self.outputs
```

```python
    def sigmoid(self, x):
        # Calculate sigmoid on vector cells
        self.sig_values = 1 / (1 + np.exp(-x))
        return self.sig_values

    def backward(self):
        d_p_wrt_f = self.sig_values * (1 - self.sig_values)
        if self.dropout:
            d_p_wrt_f = d_p_wrt_f * self.inputWire.dropout_vector
        self.d_E_wrt_f = np.sum(self.outputWire.gradients) * d_p_wrt_f if self.outputWire.gradients.shape[
                                                                                0] > 1 else self.outputWire.gradients * d_p_wrt_f
        d_E_wrt_x = self.inputWire.weights.dot(
            np.diag(self.d_E_wrt_f))  # Multiply outgoing gradients by the dE/df of their respective node in this layer
        self.inputWire.gradients = d_E_wrt_x
        # print("backward gradient:", d_E_wrt_x)

        self.update()

    def update(self):
        global alpha
        # if self.dropout:
        #     self.inputWire.weights -= alpha*(np.outer(self.inputs, self.inputWire.dropout_matrix.dot(self.d_E_wrt_f)))
        # self.inputWire.velocity = self.inputWire.momentum*self.inputWire.velocity - alpha * (np.outer(self.inputs, self.d_E_wrt_f))
        # self.inputWire.weights += self.inputWire.velocity
        self.inputWire.weights -= alpha * (np.outer(self.inputs, self.d_E_wrt_f))


class reluLayer(Layer):
    def __init__(self, inputWire, units, dropout=False):
        super().__init__(inputWire, units, dropout, weight_fn=lambda x, y: np.random.randn(x, y) / np.sqrt(x / 2))

    def forward(self, x, training=False):
        self.inputs = x
        # Add input value of "1" to the input array for the bias weight (w_0*x_0 + ...)
        self.inputs = np.insert(x, 0, 1.0, 0)
        self.linear_outputs = self.inputWire.weights.T.dot(self.inputs)

        self.outputs = self.relu(self.linear_outputs)
        if training and self.dropout:
            self.inputWire.generate_dropout_variables()
            # self.outputs = self.outputs.dot(self.inputWire.dropout_matrix)
            self.outputs = self.outputs * self.inputWire.dropout_vector / self.inputWire.dropout_proba
        return self.outputs

    def relu(self, x):
        # Calculate sigmoid on vector cells
        self.relu_values = np.maximum(0, x)
        return self.relu_values

    def backward(self):
        d_p_wrt_f = np.zeros(self.relu_values.shape)
        d_p_wrt_f[self.relu_values > 0] = 1
        if self.dropout:
            d_p_wrt_f = d_p_wrt_f * self.inputWire.dropout_vector
        self.d_E_wrt_f = np.sum(self.outputWire.gradients[1:], 1) * d_p_wrt_f
        d_E_wrt_x = self.inputWire.weights.dot(
            np.diag(self.d_E_wrt_f))  # Multiply outgoing gradients by the dE/df of their respective node in this layer
        self.inputWire.gradients = d_E_wrt_x
        # print("backward gradient:", d_E_wrt_x)

        self.update()

    def update(self):
        global alpha
        # if self.dropout:
        #     self.inputWire.weights -= alpha*(np.outer(self.inputs, self.inputWire.dropout_matrix.dot(self.d_E_wrt_f)))
        # self.inputWire.velocity = self.inputWire.momentum*self.inputWire.velocity - alpha * (np.outer(self.inputs, self.d_E_wrt_f))
        # self.inputWire.weights += self.inputWire.velocity
        self.inputWire.weights -= alpha * (np.outer(self.inputs, self.d_E_wrt_f))


class softmaxLayer(Layer):
    def __init__(self, inputWire, units, dropout=False):
        super().__init__(inputWire, units, dropout)

    def forward(self, x, training=False):
        self.inputs = x
        # Add input value of "1" to the input array for the bias weight (w_0*x_0 + ...)
        self.inputs = np.insert(x, 0, 1, 0)
        self.outputs = self.softmax(self.inputWire.weights.T.dot(self.inputs))
        return self.outputs

    def softmax(self, x):
        # Calculate sigmoid on vector cells
        exponentials = np.exp(x - np.max(x))
        self.softmax_values = exponentials / np.sum(exponentials)
        return self.softmax_values

    def backward(self):
        self.inputWire.gradients = None

        d_p_wrt_f = self.softmax_values * (1 - self.softmax_values)
        self.d_E_wrt_f = self.outputWire.gradients * d_p_wrt_f  # single incoming gradient
        d_E_wrt_x = self.inputWire.weights.dot(
```

```
            np.diag(self.d_E_wrt_f))  # Multiply outgoing gradients by the dE/df of their respective node in this layer
        self.inputWire.gradients = d_E_wrt_x

        self.update()

    def backward(self, y):
        self.inputWire.gradients = None

        self.d_E_wrt_f = self.outputs - y  # single incoming gradient
        d_E_wrt_x = self.inputWire.weights.dot(
            np.diag(self.d_E_wrt_f))  # Multiply outgoing gradients by the dE/df of their respective node in this layer
        self.inputWire.gradients = d_E_wrt_x

        self.update()

    def update(self):
        global alpha
        # TODO: Refactor away from the outer() function to allow for mini-batch
        self.inputWire.weights -= alpha * (np.outer(self.inputs, self.d_E_wrt_f))

from numpy.random import multivariate_normal

def unison_shuffled_copies(a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return [a[p], b[p]]

mean_w1 = np.array([0, 0])
covar_w1 = np.array([[1,0], [0,1]])
mean_w2 = np.array([1, 0.5])
covar_w2 = np.array([[3, 1],[1, 2]])

# Generate training data and labels
training_w1_data = multivariate_normal(mean_w1, covar_w1, size=(50))
training_w1_labels = np.zeros(training_w1_data.shape[0])
training_w2_data = multivariate_normal(mean_w2, covar_w2, size=(50))
training_w2_labels = np.ones(training_w2_data.shape[0])

# Generate test data and labels
test_w1_data = multivariate_normal(mean_w1, covar_w1, size=(20))
test_w1_labels = np.zeros(test_w1_data.shape[0])
test_w1_labels = test_w1_labels.reshape((-1, 1))
test_w2_data = multivariate_normal(mean_w2, covar_w2, size=(20))
test_w2_labels = np.ones(test_w2_data.shape[0])
test_w2_labels = test_w2_labels.reshape((-1, 1))
training_data = np.r_[training_w1_data, training_w2_data]
training_labels = np.r_[training_w1_labels, training_w2_labels]
training = unison_shuffled_copies(training_data, training_labels)
training[1] = training[1].reshape((-1, 1))
test_data = np.r_[test_w1_data, test_w2_data]
test_labels = np.r_[test_w1_labels, test_w2_labels]
test = unison_shuffled_copies(test_data, test_labels)
test[1] = test[1].reshape((-1, 1))

EPOCHS = 100
nn_architecture = [("input", 2), ("sigmoid", 5), ("sigmoid", 1)]
nn = NeuralNetwork(nn_architecture, dropout=False)
nn.train_and_validate(training, training, EPOCHS)
```