**Deep Learning – Assignment 1**

**Part I**

1) Below is the code implemented using Python for calculating the loss function “soft-max regression” and its gradient with respect to W and its gradient with respect to the biases. For efficiency purposes, we are calculating the gradient for all and () using matrix multiplication, and return the Jacobian with respect to W (which is – row of the output would be ) and the Jacobian with respect to the biases (which is – row of the output would be :

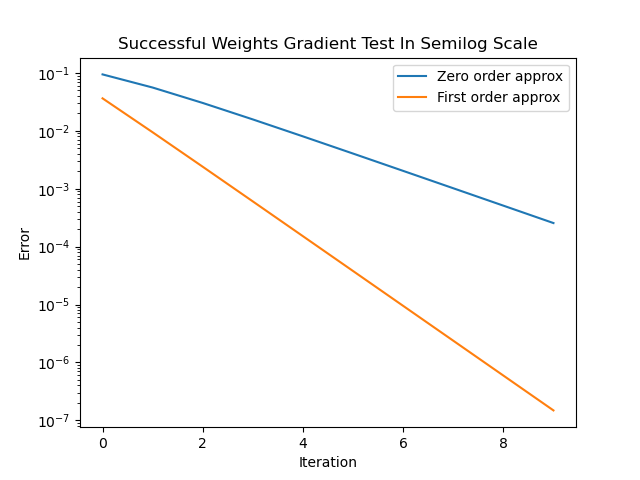
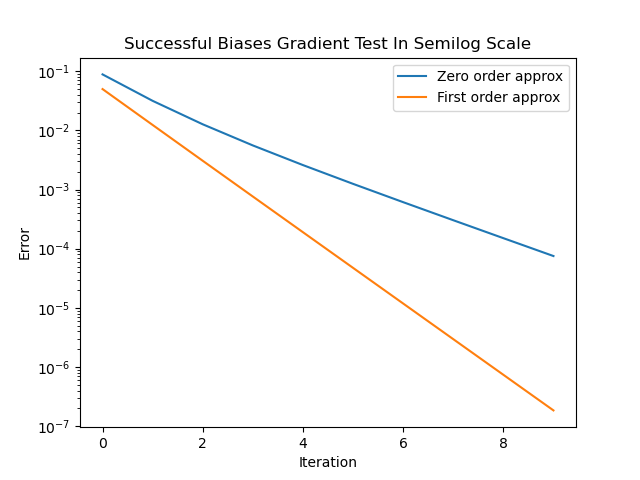
import numpy as np  
  
def softmax\_regression(X\_L, W, bias, C):  
 probs = sigmoid(X\_L, W, bias)  
 m = X\_L.shape[1]  
 cost = (-1 / m) \* (np.sum(C.T \* np.log(probs)))  
 return cost, probs  
  
  
def sigmoid(X\_L, W, bias):  
 output\_layer = X\_L.T @ W + bias  
 output\_layer -= np.max(output\_layer)  
 probs = (np.exp(output\_layer).T / np.sum(np.exp(output\_layer), axis=1)).T  
 return probs  
  
  
def softmax\_grad(X, W, bias, C):  
 probs = sigmoid(X, W, bias)  
 m = X.shape[1]  
 grad\_W = (-1 / m) \* (X @ (C.T - probs))  
 grad\_b = (-1 / m) \* (np.sum(C.T - probs, axis=0)).T  
 return grad\_W, grad\_b

To verify that the derivatives are correct, we implemented a gradient test for the gradient with respect to the weights and the biases (and a plot function for the result):

import numpy as np  
import matplotlib.pyplot as plt  
from softmax import softmax\_regression, softmax\_grad  
from loadData import extract\_frad\_test\_data  
  
num\_of\_iterations = 10  
  
  
def grad\_W\_test(X, W, bias, C):  
 # calculate F(W) and the gradient w.r.t W  
 F\_0, \_ = softmax\_regression(X, W, bias, C)  
 grad\_W, \_ = softmax\_grad(X, W, bias, C)  
 grad\_W\_flat = grad\_W.flatten()  
  
 # define a random d for the test  
 d = np.random.rand(grad\_W.shape[0], grad\_W.shape[1])  
 d /= np.linalg.norm(d)  
 d\_flat = d.flatten()  
  
 zero\_order = []  
 first\_order = []  
 print('\nGradient test w.r.t W results:')  
 print('k\t\terror order 0\t\terror order 1')  
 for k in range(num\_of\_iterations):  
 epsilon = 0.5 \*\* k  
 F\_k, \_ = softmax\_regression(X, W + epsilon \* d, bias, C)  
 F\_1 = F\_0 + epsilon \* d\_flat.T @ grad\_W\_flat  
 zero\_order.append(abs(F\_k - F\_0))  
 first\_order.append(abs(F\_k - F\_1))  
 print(k, '\t', abs(F\_k - F\_0), '\t', abs(F\_k - F\_1))  
 return zero\_order, first\_order  
  
  
def grad\_bias\_test(X, W, bias, C):  
 # calculate F(W) and the gradient w.r.t bias  
 F\_0, \_ = softmax\_regression(X, W, bias, C)  
 \_, grad\_bias = softmax\_grad(X, W, bias, C)  
  
 # define a random d for the test  
 d = np.random.rand(grad\_bias.shape[0])  
 d /= np.linalg.norm(d)  
  
 zero\_order = []  
 first\_order = []  
 print('\nGradient test w.r.t bias results:')  
 print('k\t\terror order 0\t\terror order 1')  
 for k in range(num\_of\_iterations):  
 epsilon = 0.5 \*\* k  
 F\_k, \_ = softmax\_regression(X, W, bias + epsilon \* d, C)  
 F\_1 = F\_0 + epsilon \* d.T @ grad\_bias  
 zero\_order.append(abs(F\_k - F\_0))  
 first\_order.append(abs(F\_k - F\_1))  
 print(k, '\t', abs(F\_k - F\_0), '\t', abs(F\_k - F\_1))  
 return zero\_order, first\_order  
  
  
def draw\_results(y\_0, y\_1, result\_for='Weights'):  
 plt.semilogy([i for i in range(num\_of\_iterations)], y\_0)  
 plt.semilogy([i for i in range(num\_of\_iterations)], y\_1)  
 plt.legend(["Zero order approx", "First order approx"])  
 plt.title("Successful {res\_for} Gradient Test In Semilog Scale".format(res\_for=result\_for))  
 plt.xlabel("Iteration")  
 plt.ylabel("Error")  
 plt.show()

To test our code, we ran a small experiment over a small batch of data points (0.01 of the dataset) from the provided dataset “Peaks”. The code for the test is as follows:

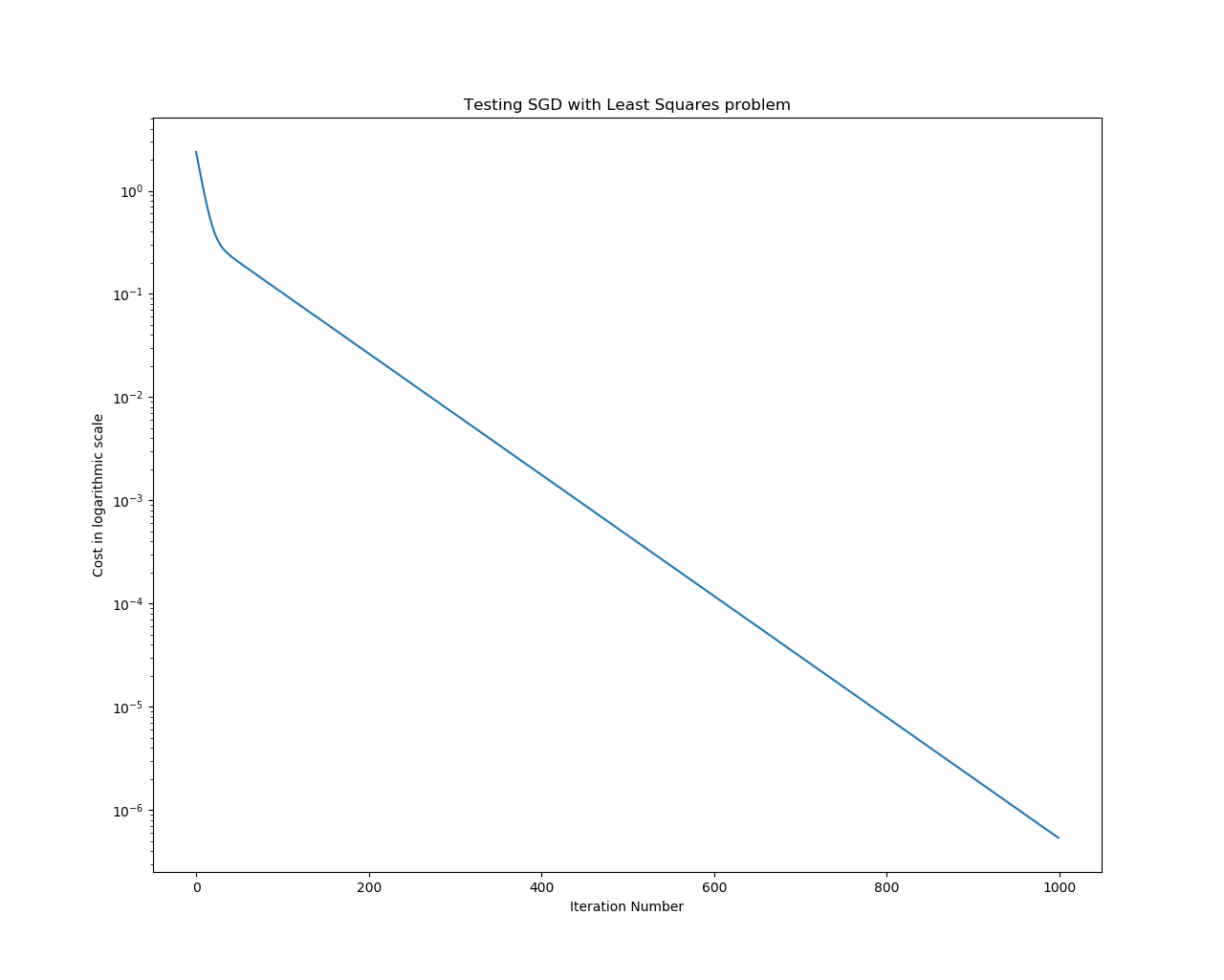
if \_\_name\_\_ == '\_\_main\_\_':  
 X\_batches, W, bias, C\_batches = extract\_frad\_test\_data("PeaksData.mat", 100)  
 zero\_order\_W, first\_order\_W = grad\_W\_test(X\_batches[0], W, bias, C\_batches[0])  
 draw\_results(zero\_order\_W, first\_order\_W)  
 zero\_order\_bias, first\_order\_bias = grad\_bias\_test(X\_batches[0], W, bias, C\_batches[0])  
 draw\_results(zero\_order\_bias, first\_order\_bias, result\_for='Biases')

And the plots we got for the gradient tests are:

Note that we defined , where , and ran the test for 10 iterations (. We can clearly see that the test for both the weights and the biases is successful, as the error for the first order taylor expansion approximation decreases quadratically compared to the zero order taylor expansion approximation.

2( The code below is our implementation for the SGD:

def sgd(grad\_function, cost\_function, X, W, C, bias, batch\_size, learning\_rate, iter\_num):  
 costs = []  
 for iter in range(iter\_num):  
 shuffler = np.random.permutation(X.shape[1])  
 X = X[shuffler]  
 C = C[shuffler]  
 m = X.shape[1]  
 for i in range(batch\_size):  
 X\_i = X[i \* batch\_size:i \* batch\_size + batch\_size, :]  
 C\_i = C[i \* batch\_size:i \* batch\_size + batch\_size]  
 grad = grad\_function(X\_i, W, C\_i, bias)  
 W = W - (1 / batch\_size) \* learning\_rate \* (grad)  
 costs.append(cost\_function(X, W, C, bias))  
 return W, costs

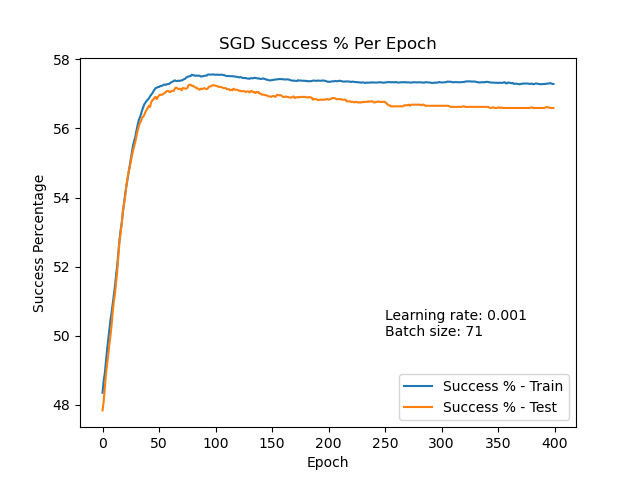
The graph at the plot is the result of testing the code on a simple least squares problem. We chose a random matrix , random result vector and a random initial guess . Also we implemented the cost function and gradient for LS. We took the learning rate to be 1 and run SGD for 1000 iterations. As you can see in the plot, the cost decreases in each iteration:

def least\_squares\_gradient(A, x, b, bias=None):  
 grad = 2 \* A.T @ A @ x - 2 \* A.T @ b  
 return grad  
  
  
def least\_squares\_cost\_function(A, x, b, bias=None):  
 cost = np.linalg.norm(A @ x - b)  
 return cost  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 iter\_num = 1000  
 A = np.random.rand(10, 6)  
 b = np.random.rand(10, 1)  
 x = np.random.rand(6, 1)  
 x, costs = sgd(least\_squares\_gradient, least\_squares\_cost\_function, A, x, b, None, 10, 1, iter\_num)  
 plt.semilogy([i for i in range(iter\_num)], costs)  
 plt.title("Testing SGD with Least Squares problem")  
 plt.xlabel("Iteration Number")  
 plt.ylabel("Cost in logarithmic scale")  
 plt.show()

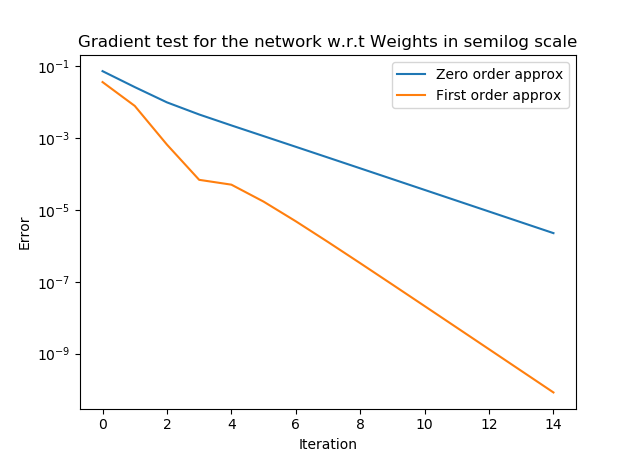
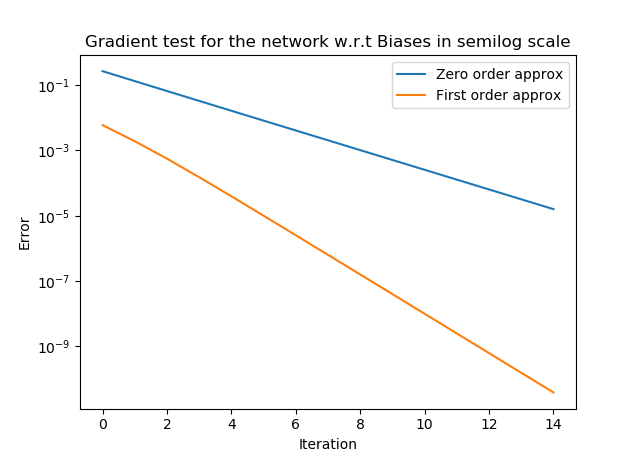
3) After running a few experiments with different batch sizes, learning rates and maximum iterations number, we noticed the following observations:

|  |  |  |
| --- | --- | --- |
|  | Smaller | Bigger |
| Batch size | As we decreased the batch size, we noticed that it took the SGD algorithm less time to converge (as the computation of the gradient in each step takes less time), but we get noisier (or less steady) success rate as the gradient calculation depended on less data points on each iteration. | As we increased the batch size, we noticed that it took the SGD algorithm more time to converge (harder to compute the gradient each step), but we got a “cleaner” success rate, as the gradient approximation was closer to the real gradient each iteration. |
| Learning rate | As we decreased the learning rate, it took the SGD algorithm more time to converge (as we’re making smaller “steps” each iteration until reaching the minimum), but we also decrease the risk of moving too far and not converge | As we increased the learning rate, we noticed that it took us less time to converge, but there’s a limit – if we increase it too much we might not converge at all |

After a few experiments, we observed that we get the best result with a **learning rate of**  and a **batch size of 71**. We let the SGD algorithm run for 400 iterations (epochs), but we saw that we get to the maximum success rate after about 75 iterations, and after that we maintained that success rate (approximately) for the rest of the iterations.

With these parameters, SGD converged and gave a **success rate** of approximately **58%** for both train and validation data. The following plot demonstrate the execution of SGD algorithm with the above parameters, and displays the success rate (in percentage) per epoch (we show how the success rate is maintained throughout the entire run of 400 epochs):

**Part II**

3) The graphs below are the results of the gradient test for the whole network. Note that we defined , where , and ran the test for 20 iterations. We can see that as the error for the first order Taylor expansion approximation decreases quadratically compared to the zero order Taylor expansion approximation, meaning the gradient was calculated correctly for both weights and biases.

Below is the python code where we implemented the tests:

import numpy as np  
from section2.NN.NN import NN  
from section2.activations import tanh, tanh\_grad  
import matplotlib.pyplot as plt  
  
num\_of\_iterations = 15  
  
  
def nn\_grad\_test\_W(nn: NN, X):  
 *"""  
 Test the gradient of the whole network w.r.t the weights* ***:param*** *nn: the neural network object* ***:param*** *X: the data to test with (we send random data)* ***:return****: shows a plot of the zero order vs. first order approximations  
 """* C = np.zeros((10, 1))  
 C[0] = 1  
 F\_0, \_, \_, nonlinear\_layers = nn.forward(X, C)  
 weight\_grads, biases\_grads = nn.backpropagation(nonlinear\_layers, C)  
 grad\_W = np.concatenate([w.flatten() for w in weight\_grads])  
  
 # define a random d for the test and align dimensions  
 ds\_w = [np.random.rand(w.shape[0], w.shape[1]) for w in nn.weights]  
 ds\_w = [d / np.linalg.norm(d) for d in ds\_w]  
 d\_flat = np.concatenate([d.flatten() for d in ds\_w])  
 nn\_weights\_original = nn.weights.copy()  
  
 zero\_order = []  
 first\_order = []  
 print('\nGradient test w.r.t bias results:')  
 print('k\t\terror order 0\t\terror order 1')  
 for k in range(num\_of\_iterations):  
 epsilon = 0.5 \*\* k  
 nn.weights = [nn.weights[i] + epsilon \* ds\_w[i] for i in range(len(nn.weights))]  
 F\_k, \_, \_, \_ = nn.forward(X, C)  
 F\_1 = F\_0 + epsilon \* d\_flat.T @ grad\_W  
 zero\_order.append(np.linalg.norm(F\_k - F\_0))  
 first\_order.append(np.linalg.norm((F\_k - F\_1)))  
 print(k, '\t', np.linalg.norm(F\_k - F\_0), '\t', np.linalg.norm(F\_k - F\_1))  
 nn.weights = nn\_weights\_original  
 draw\_results(zero\_order, first\_order)  
  
  
def nn\_grad\_test\_b(nn: NN, X):  
 *"""  
 Test the gradient of the whole network w.r.t the biases* ***:param*** *nn: the neural network object* ***:param*** *X: the data to test with (we send random data)* ***:return****: shows a plot of the zero order vs. first order approximations  
 """* C = np.zeros((10, 1))  
 C[0] = 1  
 F\_0, \_, \_, nonlinear\_layers = nn.forward(X, C)  
 \_, biases\_grads = nn.backpropagation(nonlinear\_layers, C)  
 grad\_b = np.concatenate([b.flatten() for b in biases\_grads])  
  
 # define a random d for the test  
 ds\_b = [np.random.rand(b.shape[0], b.shape[1]) for b in nn.biases]  
 ds\_b = [d / np.linalg.norm(d) for d in ds\_b]  
 d\_flat = np.concatenate([d.flatten() for d in ds\_b])  
 nn\_biases\_original = nn.biases.copy()  
  
 zero\_order = []  
 first\_order = []  
 print('\nGradient test w.r.t bias results:')  
 print('k\t\terror order 0\t\terror order 1')  
 for k in range(num\_of\_iterations):  
 epsilon = 0.5 \*\* k  
 nn.biases = [nn.biases[i] + epsilon \* ds\_b[i] for i in range(len(nn.biases))]  
 F\_k, \_, \_, \_ = nn.forward(X, C)  
 F\_1 = F\_0 + epsilon \* d\_flat.T @ grad\_b  
 zero\_order.append(np.linalg.norm(F\_k - F\_0))  
 first\_order.append(np.linalg.norm((F\_k - F\_1)))  
 print(k, '\t', np.linalg.norm(F\_k - F\_0), '\t', np.linalg.norm(F\_k - F\_1))  
 nn.biases = nn\_biases\_original  
 draw\_results(zero\_order, first\_order, "Biases")  
  
  
def draw\_results(y\_0, y\_1, result\_for='Weights'):  
 plt.semilogy([i for i in range(num\_of\_iterations)], y\_0)  
 plt.semilogy([i for i in range(num\_of\_iterations)], y\_1)  
 plt.legend(["Zero order approx", "First order approx"])  
 plt.title(f"Gradient test for the network w.r.t {result\_for} in semilog scale")  
 plt.xlabel("Iteration")  
 plt.ylabel("Error")  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 nn = NN([5, 7, 8, 10, 10], tanh, tanh\_grad)  
 X = np.random.rand(5, 1)  
 nn\_grad\_test\_W(nn, X)  
 nn\_grad\_test\_b(nn, X)