**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 :

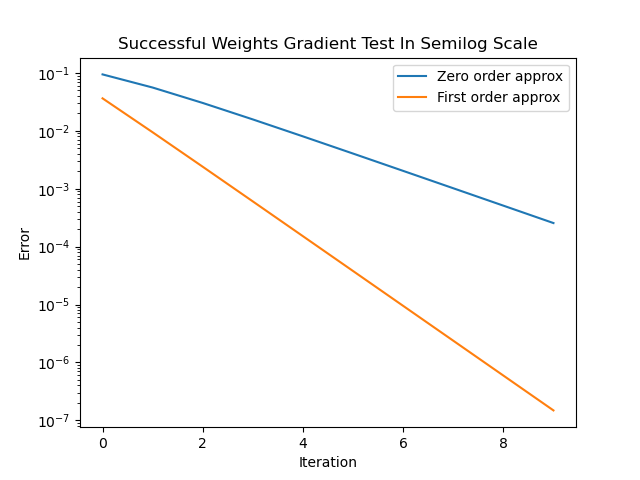
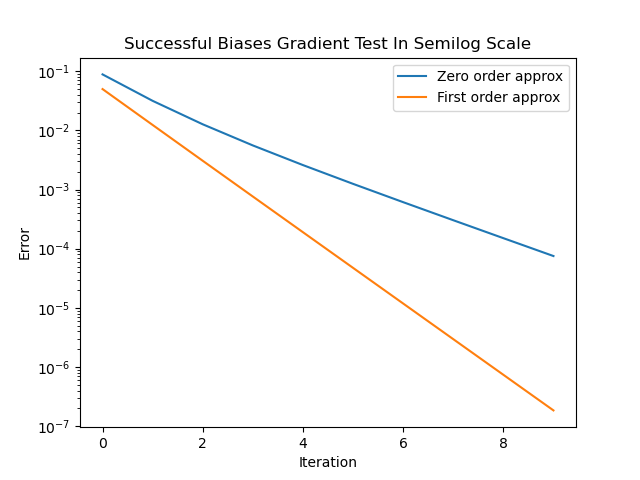
import numpy as np  
  
def softmax\_regression(X\_L, W, bias, C):  
 probs = sigmoid(X\_L, W, bias)  
 m = X\_L.shape[1]  
 cost = (-1 / m) \* (np.sum(C.T \* np.log(probs)))  
 return cost, probs  
  
  
def sigmoid(X\_L, W, bias):  
 output\_layer = X\_L.T @ W + bias  
 output\_layer -= np.max(output\_layer)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

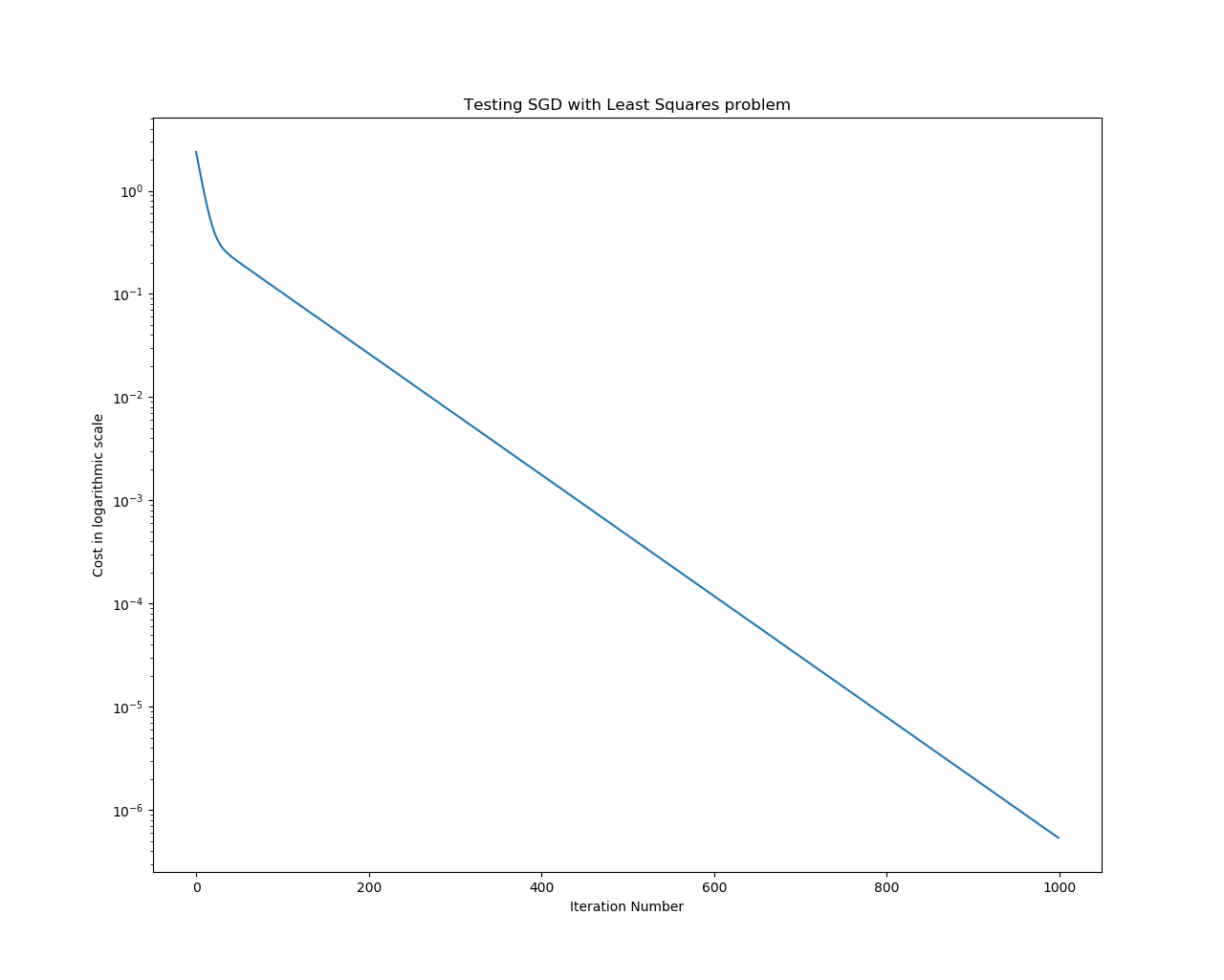
if \_\_name\_\_ == '\_\_main\_\_':  
 X\_batches, W, bias, C\_batches = extract\_frad\_test\_data("PeaksData.mat", 100)  
 zero\_order\_W, first\_order\_W = grad\_W\_test(X\_batches[0], W, bias, C\_batches[0])  
 draw\_results(zero\_order\_W, first\_order\_W)  
 zero\_order\_bias, first\_order\_bias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

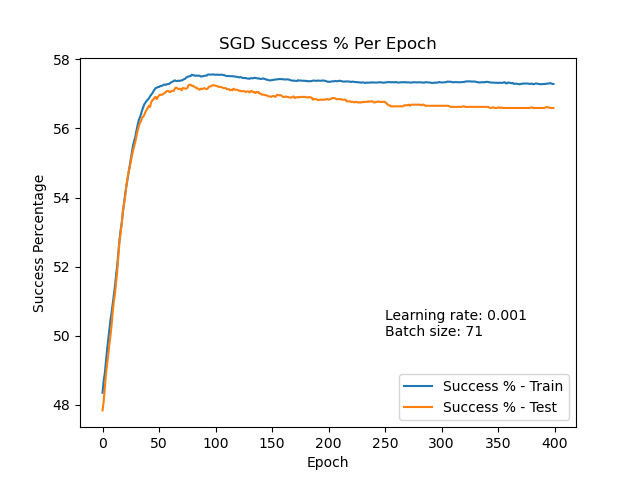
The graph at the plot is the result of testing the code on a simple least squares problem. We chose a random matrix , random result vector and a random initial guess . Also we implemented the cost function and gradient for LS. We took the learning rate to be 1 and run SGD for 1000 iterations. As you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

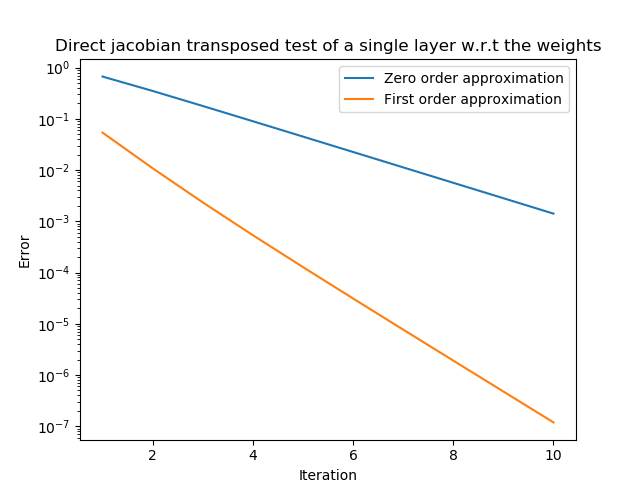
With these parameters, SGD converged and gave a **success rate** of approximately **58%** for both train and validation data. The following plot demonstrate the execution of SGD algorithm with the above parameters, and displays the success rate (in percentage) per epoch (we show how the success rate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

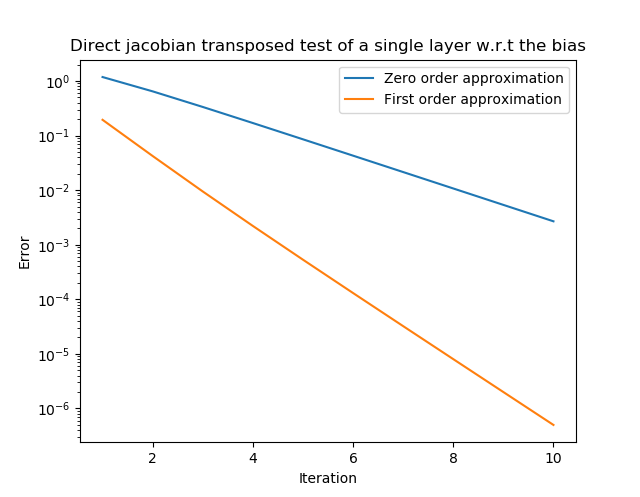
**Part II**

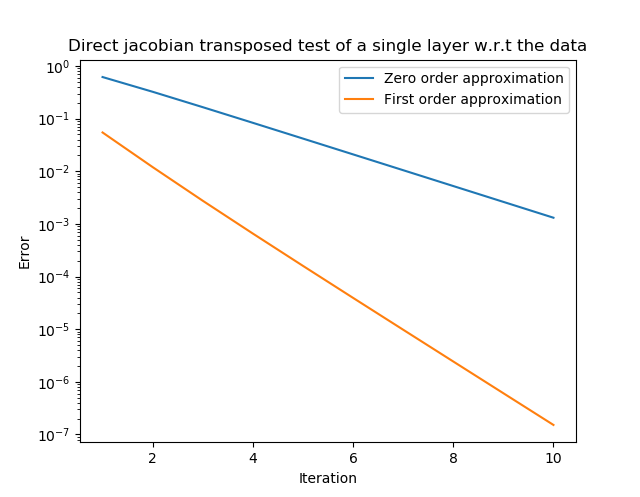
1) Below is the code for the full neural network:

class NN:  
 *"""  
 NN is a class which represents the neural network.  
 """* def \_\_init\_\_(self, network\_layers\_list, activation, activation\_gradient):  
 self.weights = []  
 self.biases = []  
 self.activation = activation  
 self.activation\_gradient = activation\_gradient  
  
 np.random.seed(0)  
 scale = 1 / max(1., (2 + 2) / 2.)  
 limit = math.sqrt(3.0 \* scale)  
  
 for i in range(len(network\_layers\_list) - 1):  
 W\_i = np.random.uniform(-limit, limit, size=(network\_layers\_list[i + 1], network\_layers\_list[i]))  
 W\_i /= np.linalg.norm(W\_i)  
 bias\_i = np.zeros([network\_layers\_list[i + 1], 1])  
 self.weights.append(W\_i)  
 self.biases.append(bias\_i)  
  
 def softmax\_layer(self, X\_L, W, bias, C):  
 *"""  
 The softmax function fot the last layer in the network* ***:param*** *X\_L: the data from step n-1* ***:param*** *W: the weights from step n-1* ***:param*** *bias: the bias from step n-1* ***:param*** *C: the indicators* ***:return****: cost value (scalar) and the probabilities matrix (for each data point, a vector of probabilities to get  
 each label.  
 """* batch\_size = C.shape[1]  
 scores = W @ X\_L + bias  
 scores -= np.max(scores)  
 probs = (np.exp(scores) / np.sum(np.exp(scores), axis=0))  
 cost = (-1 / batch\_size) \* (np.sum(np.log(probs) \* C))  
 return cost, probs  
  
 def forward\_step(self, x, W, bias):  
 *"""  
 A forward step in the network, alters the data in transition from layer k to layer k+1* ***:param*** *x: the data in layer k* ***:param*** *W: the weights in layer k* ***:param*** *bias: the bias in layer k* ***:return****: the linear result of W@x + bias, and f(linear) where f is the configured activation function  
 """* linear = W @ x + bias  
 nonlinear = self.activation(linear)  
 return linear, nonlinear  
  
 def forward(self, x\_0, C):  
 *"""  
 Forward routine of the whole network, starting from layer 1 and performing n forward steps (including the last  
 step of the softmax function)* ***:param*** *x\_0: the initial data* ***:param*** *C: the indicators* ***:return****: cost of the entire network, probabilities matrix, linear (W@x+bias) and non-linear (activation(linear))  
 results for each layer (as lists).  
 """* linear\_layers = [x\_0.copy()]  
 nonlinear\_layers = [x\_0.copy()]  
 current\_x = x\_0  
 for i in range(len(self.weights) - 1):  
 linear, current\_x = self.forward\_step(current\_x, self.weights[i], self.biases[i])  
 linear\_layers.append(linear.copy())  
 nonlinear\_layers.append(current\_x.copy())  
  
 cost, probs = self.softmax\_layer(current\_x, self.weights[-1], self.biases[-1], C)  
 nonlinear\_layers.append(probs.copy())  
 return cost, probs, linear\_layers, nonlinear\_layers  
  
 def softmax\_gradient(self, X, W, C, v):  
 *"""  
 Calculate the softmax gradients w.r.t the weights, the bias and the data.* ***:param*** *X: the data* ***:param*** *W: the weights* ***:param*** *C: the indicators* ***:param*** *v: a vector v* ***:return****: softmax gradient w.r.t weights, bias and data  
 """* batch\_size = C.shape[1]  
 dl\_dy = (1 / batch\_size) \* (X - C)  
 dl\_W = dl\_dy @ v.T  
 dl\_db = np.sum(dl\_dy, axis=1, keepdims=True)  
 new\_v = W.T @ dl\_dy  
 return dl\_W, dl\_db, new\_v  
  
 def hidden\_layer\_grad(self, X, W, b, v):  
 *"""  
 Calculate the gradient of an individual hidden layer.* ***:param*** *X: the current layer's data* ***:param*** *W: the current layer's weights* ***:param*** *b: the current layer's bias* ***:param*** *v: a vector v (from the next layer)* ***:return****: The hidden layer gradient w.r.t weights, bias and the new v vector to use in the previous layer  
 """* linear = W @ X + b  
 batch\_size = linear.shape[1]  
 grad\_activation = self.activation\_gradient(linear)  
 common = grad\_activation \* v  
 grad\_W = (1 / batch\_size) \* common @ X.T  
 grad\_b = (1 / batch\_size) \* np.sum(common, axis=1, keepdims=True)  
 grad\_X = W.T @ common  
 return grad\_W, grad\_b, grad\_X  
  
 def backpropagation(self, X\_list, C):  
 *"""  
 The backpropagation process of the network.* ***:param*** *X\_list: a list of the data X in each layer* ***:param*** *C: the indicators* ***:return****: gradients list of each layer w.r.t weights and bias  
 """* layer\_number = len(X\_list)  
 x\_grads = []  
 weights\_grads = []  
 biases\_grads = []  
  
 # last layer gradient  
 W\_i\_grad, b\_i\_grad, v\_i = self.backward\_last\_layer(X\_list, C)  
 x\_grads.insert(0, v\_i)  
 weights\_grads.insert(0, W\_i\_grad)  
 biases\_grads.insert(0, b\_i\_grad)  
  
 # hidden layer grads  
 for i in range(layer\_number - 2, 0, -1):  
 W\_i\_grad, b\_i\_grad, v\_i = self.backward\_hidden\_layer(X\_list, i, v\_i)  
 x\_grads.insert(0, v\_i)  
 weights\_grads.insert(0, W\_i\_grad)  
 biases\_grads.insert(0, b\_i\_grad)  
 return x\_grads, weights\_grads, biases\_grads  
  
 def backward\_last\_layer(self, X\_list, C):  
 *"""  
 Backward step for the last layer (softmax layer)* ***:param*** *X\_list: list of X (data) of each layer (from the forward pass)* ***:param*** *C: the indicators* ***:return****: Vector v\_i for the next backward step  
 """* W\_grad, b\_grad, x\_grad = self.softmax\_gradient(X\_list[-1], self.weights[-1], C, X\_list[-2])  
 v\_i = x\_grad.copy()  
 return W\_grad, b\_grad, v\_i  
  
 def backward\_hidden\_layer(self, X\_list, i, v):  
 *"""  
 Backward step for the hidden layers* ***:param*** *X\_list: list of X (data) of each layer (from the forward pass)* ***:param*** *i: index of the current layer* ***:param*** *v: vector v from the previous backward step* ***:return****: New vector v for the next backward step to use  
 """* F\_grad\_W\_i, F\_grad\_b\_i, F\_grad\_X\_i = self.hidden\_layer\_grad(X\_list[i - 1], self.weights[i - 1], self.biases[i - 1], v)  
 return F\_grad\_W\_i, F\_grad\_b\_i, F\_grad\_X\_i  
  
 def update\_thetas(self, weights\_grads, biases\_grads, learning\_rate):  
 *"""  
 Update the weights and biases of the network* ***:param*** *W\_grad\_list: list of gradients w.r.t weights* ***:param*** *bias\_grad\_list: list of gradients w.r.t bias* ***:param*** *learning\_rate: the learning rate  
 """* for i in range(len(self.weights)):  
 self.weights[i] = self.weights[i] - learning\_rate \* weights\_grads[i]  
 self.biases[i] = self.biases[i] - learning\_rate \* biases\_grads[i]

In order to check the correctness of the hidden layer gradient we made used the Jacobian verification. We defined the function and made a regular grad test for it, for we know that

We made the verification w.r.t weights, bias and x for a single layer.





Code for the tests:

iter\_num = 10  
  
def jacobian\_test\_layer\_X(nn: NN, X\_0):  
 *"""  
 Direct jacobian transposed test for a single layer w.r.t to the data* ***:param*** *nn: the neural network object* ***:param*** *X\_0: initial data (we init with random data)* ***:return****: shows a plot of the zero order vs. first order approximation  
 """* W\_0 = nn.weights[0].copy()  
 b\_0 = nn.biases[0].copy()  
 \_, X\_1 = nn.forward\_step(X\_0, W\_0, b\_0)  
 n, m = X\_1.shape  
 out\_dimensions = b\_0.shape[0]  
 u = np.random.rand(out\_dimensions, m)  
 d = np.random.rand(\*X\_0.shape)  
 d = (1 / np.linalg.norm(d)) \* d  
  
 g\_x = np.dot(X\_1.T, u).item()  
 \_, \_, JtU\_X = nn.backward\_hidden\_layer([X\_0, X\_1], 1, u)  
  
 zero\_order = np.zeros(iter\_num)  
 first\_order = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 for i, epsilon in enumerate(epsilons):  
 X\_diff = X\_0.copy()  
 X\_diff += d \* epsilon  
 \_, X\_eps\_forward = nn.forward\_step(X\_diff, W\_0, b\_0)  
 X\_eps\_forward\_T = X\_eps\_forward.T  
 gx\_epsilon = np.dot(X\_eps\_forward\_T, u).item()  
 d\_flat = d.reshape(-1, 1)  
 zero\_order[i] = abs(gx\_epsilon - g\_x)  
 first\_order[i] = abs(gx\_epsilon - g\_x - epsilon \* d\_flat.T @ JtU\_X)  
 draw\_results(zero\_order, first\_order)  
  
  
def jacobian\_test\_layer\_W(nn: NN, X\_0):  
 *"""  
 Direct jacobian transposed test for a single layer w.r.t to the weights* ***:param*** *nn: the neural network object* ***:param*** *X\_0: initial data (we init with random data)* ***:return****: shows a plot of the zero order vs. first order approximation  
 """* W\_0 = nn.weights[0].copy()  
 b\_0 = nn.biases[0].copy()  
 \_, X\_1 = nn.forward\_step(X\_0, W\_0, b\_0)  
 n, m = X\_1.shape  
 out\_dimensions = b\_0.shape[0]  
 u = np.random.rand(out\_dimensions, m)  
 d = np.random.rand(\*W\_0.shape)  
 d = (1 / np.linalg.norm(d)) \* d  
  
 g\_x = np.dot(X\_1.T, u).item()  
 JtU\_W, \_, \_ = nn.backward\_hidden\_layer([X\_0, X\_1], 1, u)  
  
 zero\_order = np.zeros(iter\_num)  
 first\_order = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 for i, epsilon in enumerate(epsilons):  
 W\_diff = W\_0.copy()  
 W\_diff += d \* epsilon  
 \_, X\_eps\_forward = nn.forward\_step(X\_0, W\_diff, b\_0)  
 X\_eps\_forward\_T = X\_eps\_forward.T  
 gx\_epsilon = np.dot(X\_eps\_forward\_T, u).item()  
 d\_flat = d.reshape(-1, 1)  
 JtU\_W\_flat = JtU\_W.reshape(-1, 1)  
 zero\_order[i] = abs(gx\_epsilon - g\_x)  
 first\_order[i] = abs(gx\_epsilon - g\_x - epsilon \* d\_flat.T @ JtU\_W\_flat)  
 draw\_results(zero\_order, first\_order, 'weights')  
  
  
def jacobian\_test\_layer\_b(nn: NN, X\_0):  
 *"""  
 Direct jacobian transposed test for a single layer w.r.t to biases* ***:param*** *nn: the neural network object* ***:param*** *X\_0: initial data (we init with random data)* ***:return****: shows a plot of the zero order vs. first order approximation  
 """* W\_0 = nn.weights[0].copy()  
 b\_0 = nn.biases[0].copy()  
 \_, X\_1 = nn.forward\_step(X\_0, W\_0, b\_0)  
 n, m = X\_1.shape  
 out\_dimensions = b\_0.shape[0]  
 u = np.random.rand(out\_dimensions, m)  
 d = np.random.rand(\*b\_0.shape)  
 d = (1 / np.linalg.norm(d)) \* d  
  
 g\_x = np.dot(X\_1.T, u).item()  
 \_, JtU\_b, \_ = nn.backward\_hidden\_layer([X\_0, X\_1], 1, u)  
  
 zero\_order = np.zeros(iter\_num)  
 first\_order = np.zeros(iter\_num)  
 epsilons = [0.5 \*\* i for i in range(iter\_num)]  
 for i, epsilon in enumerate(epsilons):  
 b\_diff = b\_0.copy()  
 b\_diff += d \* epsilon  
 \_, X\_eps\_forward = nn.forward\_step(X\_0, W\_0, b\_diff)  
 X\_eps\_forward\_T = X\_eps\_forward.T  
 gx\_epsilon = np.dot(X\_eps\_forward\_T, u).item()  
 zero\_order[i] = abs(gx\_epsilon - g\_x)  
 first\_order[i] = abs(gx\_epsilon - g\_x - epsilon \* d.T @ JtU\_b)  
 draw\_results(zero\_order, first\_order, 'bias')  
  
  
def draw\_results(zero\_order, first\_order, wrt='data'):  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), zero\_order)  
 plt.semilogy(np.arange(1, iter\_num + 1, 1), first\_order)  
 plt.xlabel('Iteration')  
 plt.ylabel('Error')  
 plt.title(f'Direct jacobian transposed test of a single layer w.r.t the {wrt}')  
 plt.legend(("Zero order approximation", "First order approximation"))  
 plt.show()

For the test we used as an activation between layers:

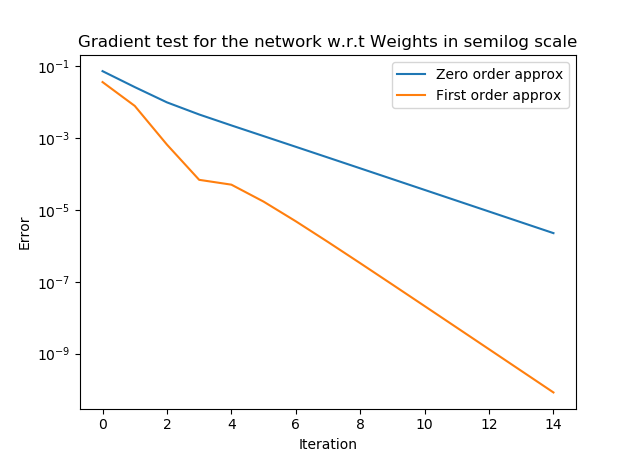
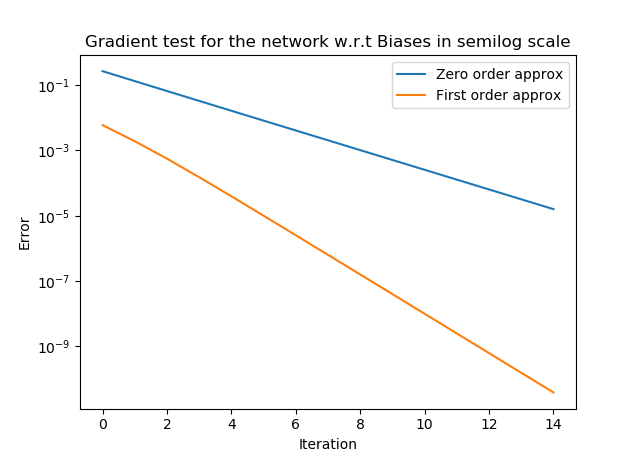
def tanh(x):  
 *"""* ***:return****: tanh activation function  
 """* return np.tanh(x)  
  
  
def tanh\_grad(x):  
 *"""  
 tanh gradient function* ***:param*** *x: point where we calculate the gradient of tanh* ***:return****: gradient of tanh in point x  
 """* return np.ones(x.shape) - (np.tanh(x)) \*\* 2

For the tests we run the following:

if \_\_name\_\_ == '\_\_main\_\_':  
 nn = NN([3, 5, 7], tanh, tanh\_grad)  
 X = np.random.rand(3, 1)  
 jacobian\_test\_layer\_b(nn, X)  
 jacobian\_test\_layer\_X(nn, X)  
 jacobian\_test\_layer\_W(nn, X)

2) Below is the code for the full residual neural network. In order to change the dimension of the input we added an optional parameter which increasing the dimension of the first layer with a regular net forward step .

class ResNet:  
 *"""  
 ResNet is a class which represents the neural network.  
 """* def \_\_init\_\_(self, input\_size, num\_of\_layers, num\_of\_labels, activation, activation\_gradient, first\_layer=8):  
 self.weights = []  
 self.biases = []  
 self.activation = activation  
 self.activation\_gradient = activation\_gradient  
  
 np.random.seed(0)  
 scale = 1 / max(1., (2 + 2) / 2.)  
 limit = math.sqrt(3.0 \* scale)  
 W\_1 = np.random.uniform(-limit, limit, size=(first\_layer, input\_size))  
 W\_1 /= np.linalg.norm(W\_1)  
 bias\_1 = np.zeros([first\_layer, 1])  
 self.weights.append(W\_1)  
 self.biases.append(bias\_1)  
  
 for i in range(1, num\_of\_layers - 2):  
 W1\_i = np.random.uniform(-limit, limit, size=(first\_layer, first\_layer))  
 W1\_i /= np.linalg.norm(W1\_i)  
 bias\_i = np.zeros([first\_layer, 1])  
 W2\_i = np.random.uniform(-limit, limit, size=(first\_layer, first\_layer))  
 W2\_i /= np.linalg.norm(W2\_i)  
 self.weights.append([W1\_i, W2\_i])  
 self.biases.append(bias\_i)  
  
 W\_n = np.random.uniform(-limit, limit, size=(num\_of\_labels, first\_layer))  
 bias\_n = np.zeros([num\_of\_labels, 1])  
 self.weights.append(W\_n)  
 self.biases.append(bias\_n)  
  
 def softmax\_layer(self, X\_L, W, bias, C):  
 *"""  
 The softmax function fot the last layer in the network* ***:param*** *X\_L: the data from step n-1* ***:param*** *W: the weights from step n-1* ***:param*** *bias: the bias from step n-1* ***:param*** *C: the indicators* ***:return****: cost value (scalar) and the probabilities matrix (for each data point, a vector of probabilities to get  
 each label.  
 """* batch\_size = C.shape[1]  
 scores = W @ X\_L + bias  
 scores -= np.max(scores)  
 probs = (np.exp(scores) / np.sum(np.exp(scores), axis=0))  
 cost = (-1 / batch\_size) \* (np.sum(np.log(probs) \* C))  
 return cost, probs  
  
 def res\_forward\_step(self, x, Ws, bias):  
 *"""  
 A forward step in the residual network, alters the data in transition from layer k to layer k+1* ***:param*** *x: the data in layer k* ***:param*** *Ws: the weights in layer k (we have W1 and W2)* ***:param*** *bias: the bias in layer k* ***:return****: the linear result of W1@x + bias, and x + W2 @ f(linear) where f is the configured activation function  
 """* linear = Ws[0] @ x + bias  
 nonlinear = x + Ws[1] @ self.activation(linear)  
 return linear, nonlinear  
  
 def forward\_step(self, x, W, bias):  
 *"""  
 The first forward step of the ResNet, used to change dimensions at the beginning of the network.* ***:param*** *x: the data* ***:param*** *W: the weights* ***:param*** *bias: the bias* ***:return****: the linear result of W@x + bias, and f(linear) where f is the configured activation function  
 """* linear = W @ x + bias  
 nonlinear = self.activation(linear)  
 return linear, nonlinear  
  
 def forward(self, x\_0, C):  
 *"""  
 Forward routine of the whole residual network, starting from layer 1 and performing n forward steps (including  
 the last step of the softmax function)* ***:param*** *x\_0: the initial data* ***:param*** *C: the indicators* ***:return****: cost of the entire residual network, probabilities matrix, linear (W@x+bias) and non-linear  
 (activation(linear)) results for each layer (as lists).  
 """* linear\_layers = [x\_0.copy()]  
 nonlinear\_layers = [x\_0.copy()]  
 linear, current\_x = self.forward\_step(x\_0, self.weights[0], self.biases[0])  
 linear\_layers.append(linear.copy())  
 nonlinear\_layers.append(current\_x.copy())  
 for i in range(1, len(self.weights) - 1):  
 linear, current\_x = self.res\_forward\_step(current\_x, self.weights[i], self.biases[i])  
 linear\_layers.append(linear.copy())  
 nonlinear\_layers.append(current\_x.copy())  
  
 cost, probs = self.softmax\_layer(current\_x, self.weights[-1], self.biases[-1], C)  
 nonlinear\_layers.append(probs.copy())  
 return cost, probs, linear\_layers, nonlinear\_layers  
  
 def softmax\_gradient(self, X, W, C, v):  
 *"""  
 Calculate the softmax gradients w.r.t the weights, the bias and the data.* ***:param*** *X: the data* ***:param*** *W: the weights* ***:param*** *C: the indicators* ***:param*** *v: a vector v* ***:return****: softmax gradient w.r.t weights, bias and data  
 """* batch\_size = C.shape[1]  
 dl\_dy = (1 / batch\_size) \* (X - C)  
 dl\_W = dl\_dy @ v.T  
 dl\_db = np.sum(dl\_dy, axis=1, keepdims=True)  
 new\_v = W.T @ dl\_dy  
 return dl\_W, dl\_db, new\_v  
  
 def res\_hidden\_layer\_grad(self, X, Ws, b, v):  
 *"""  
 Calculate the gradient of an individual hidden layer.* ***:param*** *X: the current layer's data* ***:param*** *Ws: the current layer's weights (W1 and W2)* ***:param*** *b: the current layer's bias* ***:param*** *v: a vector v (from the next layer)* ***:return****: The hidden layer gradient w.r.t weights (W1 and W2), bias and the new v vector to use in the previous  
 layer.  
 """* linear = Ws[0] @ X + b  
 batch\_size = linear.shape[1]  
 grad\_activation = self.activation\_gradient(linear)  
 grad\_W1 = (1 / batch\_size) \* (grad\_activation \* (Ws[1].T @ v)) @ X.T  
 grad\_W2 = (1 / batch\_size) \* v @ (self.activation(linear)).T  
 grad\_b = (1 / batch\_size) \* np.sum((grad\_activation \* (Ws[1].T @ v)), axis=1, keepdims=True)  
 grad\_X = v + (Ws[0].T @ (grad\_activation \* (Ws[1].T @ v)))  
 return grad\_W1, grad\_W2, grad\_b, grad\_X  
  
 def hidden\_layer\_grad(self, X, W, b, v):  
 *"""  
 Calculate the gradient of the first layer.* ***:param*** *X: the first layer's data* ***:param*** *W: the first layer's weights* ***:param*** *b: the first layer's bias* ***:param*** *v: a vector v (from the second layer)* ***:return****: The first layer gradient w.r.t weights, bias and the new v vector to use in the previous layer  
 """* linear = W @ X + b  
 batch\_size = linear.shape[1]  
 grad\_activation = self.activation\_gradient(linear)  
 common = grad\_activation \* v  
 grad\_W = (1 / batch\_size) \* common @ X.T  
 grad\_b = (1 / batch\_size) \* np.sum(common, axis=1, keepdims=True)  
 grad\_X = W.T @ common  
 return grad\_W, grad\_b, grad\_X  
  
 def backpropagation(self, X\_list, C):  
 *"""  
 The backpropagation process of the network.* ***:param*** *X\_list: a list of the data X in each layer* ***:param*** *C: the indicators* ***:return****: gradients list of each layer w.r.t weights and bias  
 """* layer\_number = len(X\_list)  
 x\_grads = []  
 weight\_grads = []  
 bias\_grads = []  
  
 # last layer gradient  
 W\_grad, b\_grad, x\_grad = self.backward\_last\_layer(X\_list, C)  
 x\_grads.insert(0, x\_grad.copy())  
 weight\_grads.insert(0, W\_grad.copy())  
 bias\_grads.insert(0, b\_grad.copy())  
 v\_i = x\_grad.copy()  
  
 # hidden layer grads  
 for i in range(layer\_number - 2, 1, -1):  
 F\_grad\_W1\_i, F\_grad\_W2\_i, F\_grad\_b\_i, v\_i = self.backward\_hidden\_layer(X\_list, i, v\_i)  
 x\_grads.insert(0, v\_i)  
 weight\_grads.insert(0, [F\_grad\_W1\_i.copy(), F\_grad\_W2\_i.copy()])  
 bias\_grads.insert(0, F\_grad\_b\_i.copy())  
  
 grad\_first\_W, grad\_first\_b, grad\_first\_X = self.hidden\_layer\_grad(X\_list[0], self.weights[0], self.biases[0], v\_i)  
 x\_grads.insert(0, grad\_first\_X)  
 weight\_grads.insert(0, grad\_first\_W.copy())  
 bias\_grads.insert(0, grad\_first\_b.copy())  
 return x\_grads, weight\_grads, bias\_grads  
  
 def backward\_last\_layer(self, X\_list, C):  
 *"""  
 Calculate the gradient of the last layer.* ***:param*** *X\_list: all nonlinear x's collected at forward steps* ***:param*** *C: an indicator matrix for labeling* ***:return****: The last layer gradient w.r.t weights, bias and x  
 """* W\_grad, b\_grad, x\_grad = self.softmax\_gradient(X\_list[-1], self.weights[-1], C, X\_list[-2])  
 return W\_grad, b\_grad, x\_grad  
  
 def backward\_hidden\_layer(self, X\_list, i, v):  
 *"""  
 Backward step for the first hidden layer* ***:param*** *X\_list: list of X (data) of each layer (from the forward pass)* ***:param*** *i: index of the current layer* ***:param*** *v: vector v from the previous backward step* ***:return****: New vector v for the next backward step to use  
 """* F\_grad\_W1\_i, F\_grad\_W2\_i, F\_grad\_b\_i, grad\_X\_i = self.res\_hidden\_layer\_grad(X\_list[i - 1], self.weights[i - 1], self.biases[i - 1], v)  
 return F\_grad\_W1\_i, F\_grad\_W2\_i, F\_grad\_b\_i, grad\_X\_i  
  
 def update\_thetas(self, W\_grad\_list, bias\_grad\_list, learning\_rate):  
 *"""  
 Update the weights (W1 and W2) and biases of the network* ***:param*** *W\_grad\_list: list of gradients w.r.t weights* ***:param*** *bias\_grad\_list: list of gradients w.r.t bias* ***:param*** *learning\_rate: the learning rate  
 """* self.weights[0] = self.weights[0] - learning\_rate \* W\_grad\_list[0]  
 self.biases[0] = self.biases[0] - learning\_rate \* bias\_grad\_list[0]  
 for i in range(1, len(self.weights)):  
 self.weights[i][0] = self.weights[i][0] - learning\_rate \*W\_grad\_list[i][0]  
 self.weights[i][1] = self.weights[i][1] - learning\_rate \*W\_grad\_list[i][1]  
 self.biases[i] = self.biases[i] - learning\_rate \* bias\_grad\_list[i]

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:

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)

4) Code of the SGD for the whole network:

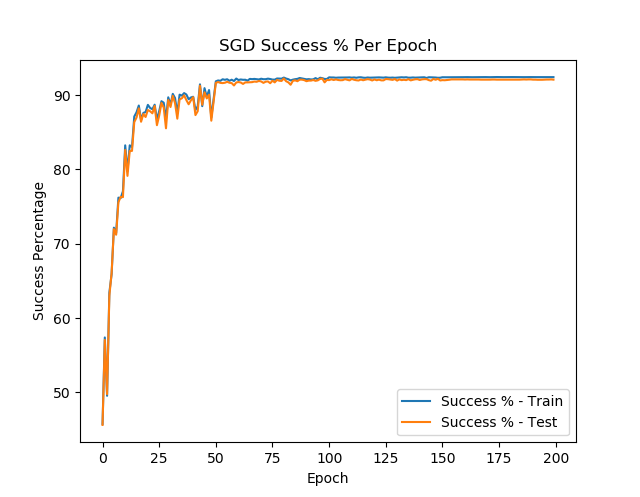
def sgd(nn: NN, X\_train, X\_test, W, C\_train, C\_test, batch\_size, learning\_rate, epoch\_num, divide\_lr=50):  
 *"""  
 SGD for the neural network. each SGD batch iteration, the network learns using forward pass. then, backpropagation  
 occurs to calculate the gradients and update the network params.* ***:param*** *nn: the neural network object* ***:param*** *X\_train: train data* ***:param*** *X\_test: test data* ***:param*** *W: weights* ***:param*** *C\_train: train indicators* ***:param*** *C\_test: test indicators* ***:param*** *batch\_size: the desired batch size* ***:param*** *learning\_rate: the desired learning rate* ***:param*** *epoch\_num: number of epochs to perform* ***:param*** *divide\_lr: how many epochs until dividing the learning rate by 10* ***:return****: Optimized weights, costs through the optimization, accuracy lists for the train and test  
 """* costs = []  
 accuracy\_train = []  
 accuracy\_test = []  
 m = X\_train.shape[1]  
 for epoch in range(epoch\_num):  
 cur\_costs = []  
 if epoch % divide\_lr == 0:  
 learning\_rate /= 10  
 shuffler = np.random.permutation(X\_train.shape[1])  
 print(epoch)  
 X\_shuffled = X\_train.T[shuffler].T  
 C\_shuffled = C\_train.T[shuffler].T  
 for i in range(int(m / batch\_size)):  
 X\_batch = X\_shuffled[:, i \* batch\_size:i \* batch\_size + batch\_size]  
 C\_batch = C\_shuffled[:, i \* batch\_size:i \* batch\_size + batch\_size]  
 cost, probs, linear\_layers, nonlinear\_layers = nn.forward(X\_batch, C\_batch)  
 \_, weights\_grads, biases\_grads = nn.backpropagation(nonlinear\_layers, C\_batch)  
 nn.update\_thetas(weights\_grads, biases\_grads, learning\_rate)  
 cur\_costs.append(cost)  
 costs.append(sum(cur\_costs) / len(cur\_costs))  
 accuracy\_train.append(success\_percentage(nn, X\_shuffled, C\_shuffled))  
 accuracy\_test.append(success\_percentage(nn, X\_test, C\_test))  
 return W, costs, accuracy\_train, accuracy\_test

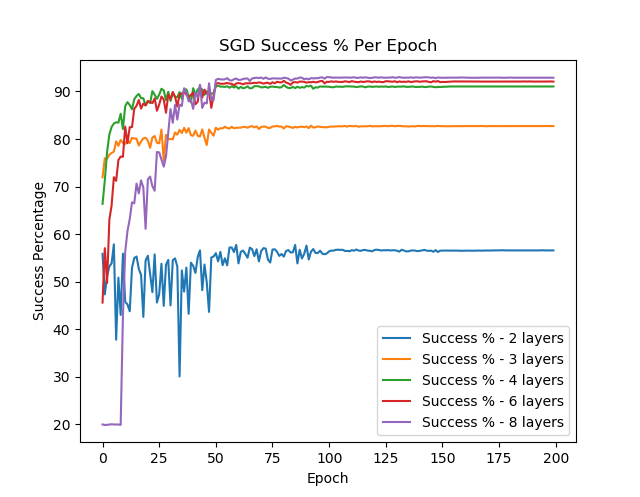
For all the experiments below we used ReLU function as an activation:

def ReLU(x):  
 *"""  
 ReLU activation function* ***:param*** *x: point on which to calculate the ReLU* ***:return****: point/vector in which each entry is either 0 or positive  
 """* return np.maximum(0, x)  
  
  
def ReLU\_grad(x):  
 *"""  
 ReLU gradient function* ***:param*** *x: point where we calculate the gradient of ReLU* ***:return****: gradient of ReLU in point x  
 """* vfunc = np.vectorize(lambda t: 1 if t >= 0 else 0)  
 return vfunc(x)

Peaks Data

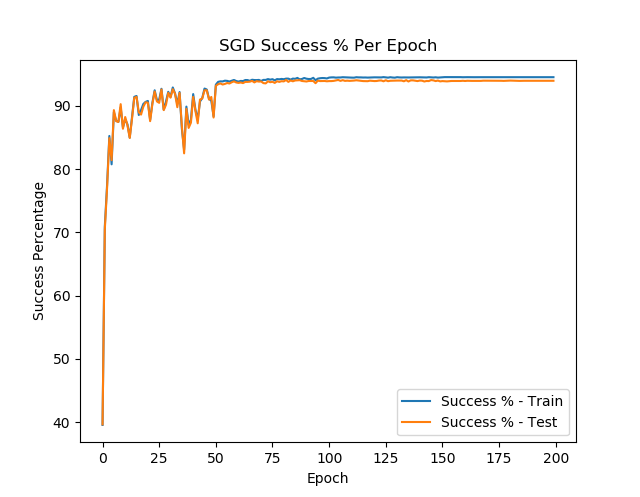
Below is the graph of success % for test and train sets by iteration. We managed to achieve fast convergence at the beginning using a relatively large learning rate = 1, and then dividing it by 10 every 50 iterations. Also we find that the best results were given when batch size = 60. After 70 iterations our model achieved almost 95% accuracy and it barely got better, even when reducing the learning rate. We choose an architecture of 4 hidden layers of size 5,6,8,10.

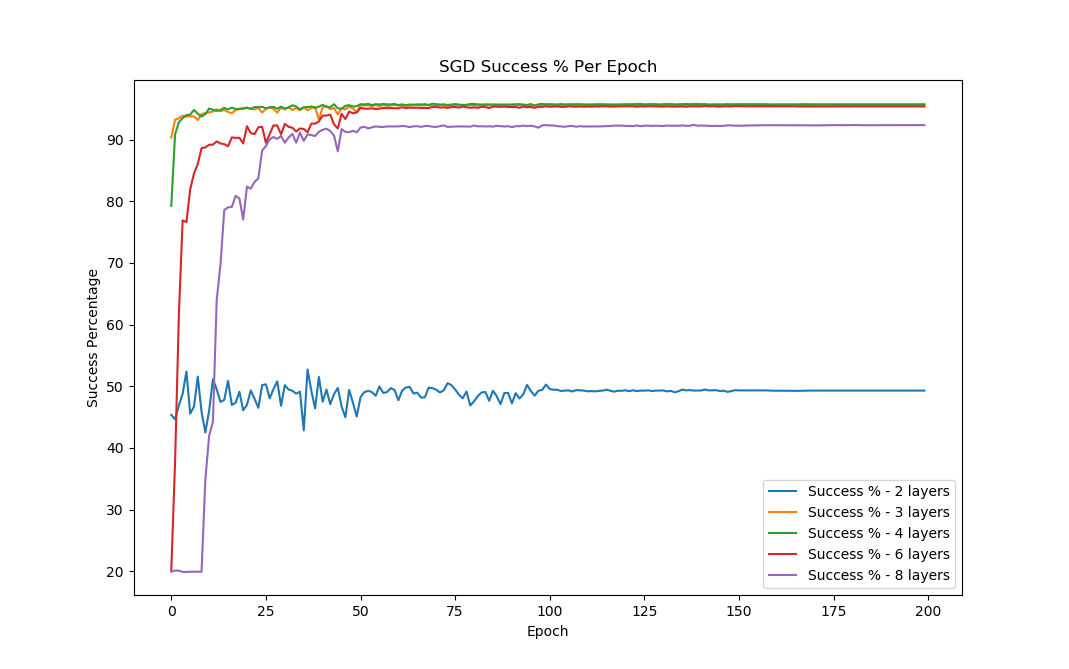


Here are the results of the test accuracy comparing different architectures with the same learning rate and batch size:

We can see that as the number of layers gets larger, it takes more iterations to achieve convergence, which make sense due to more parameters the model should learn. Also, there is a correlation between accuracy after 75 iteration to the number of layers, more layers achieve better results, but the difference is not significant.

GMM Data

 The results were quite similar to the peaks data results. Again we started with a high learning rate to achieve fast convergence at the beginning and dividing it by 10 every 50 iterations. This time we used batch size = 32. We choose an architecture of 4 hidden layers with size 5,6,8,10.

Here are the results of the test accuracy comparing different architectures with the same learning rate and batch size:

This time we can see that 3,4 and 6 layers achieved similar results while 8 layers accuracy was a little smaller for the same parameters. Together with the previous results we can see that a dipper network does not necessarily achieve a much better results for the same parameters.

Swiss Roll

SwissRoll dataset was much more challenging for our model to learn. This time we choose a constant learning rate = 0.001 and batch size = 20, the architecture consists 5 hidden layers of size 5, 5, 6, 6, 4. Eventually we were able to achieve 95% accuracy, but it took our model around 300 iterations in order to get there for the first time.

