COMPUTER SCIENCE DEPARTMENT

Practical Deep Learning

Assignment 1 Report

Authors:
Almog Zemach: 205789001
Raz Monsonego: 313292120

May 2022

Contents

1 Part I: The classifier and optimizer

1.1 Gradient test for softmax regression

Initially, we implement the loss-function using the soft-max regression function, and its gradient with respect to the weights and biases (see code attached).
Then, we make sure we calculated the derivatives correctly by using the gradient-test. The code that performs the gradient test and the results are shown bellow:

Gradient test code:

```
def gradient_test(X: np.array, W: np.array, C: np.array, b: np.array, policy):
    """
    Gradient test with respect for W.
    :param X matrix.
    :param W matrix.
    :param C matrix.
    :param b matrix.
    :param policy. Indicates to which parameter we are doing the test.
    :return matplotlib graph which shows the gradiant test.
    """
    dict_num = {"W": 1, "X": 2, "b": 3}
    dict_param = {"W": W, "X": X, "b": b}
    dict_name = {"W": "$\delta W$", "X": "$\delta X$", "b": "$\delta b$"}
    V = np.random.rand(dict_param[policy].shape[0], dict_param[policy].shape[1])
    d = (V / np.linalg.norm(V))
    d_vector = d.reshape(-1, 1)
    err_1 = []
    err_2 = []
    ks = []
    params = soft_max_regression(X, W, C, b)
    grad = params[dict_num[policy]].reshape(-1, 1)
    for k in range(1, 20):
        epsilon = 0.5 ** k
        new = dict_param[policy] + epsilon * d
        dict_args = {"W": (X, new, C, b), "X": (new, W, C, b), "b": (X, W, C, new)}
        f_x_d, _, _, _ = soft_max_regression(*dict_args[policy])
        err_1.append(abs(f_x_d - params[0]))
        err_2.append(abs(f_x_d - params[0] - (epsilon * d_vector.T @ grad)[0][0]))
        ks.append(k)
    print_test(ks, err_1, err_2, "Gradiant Test: " + dict_name[policy])
```

Gradient test results (for the soft-max function):

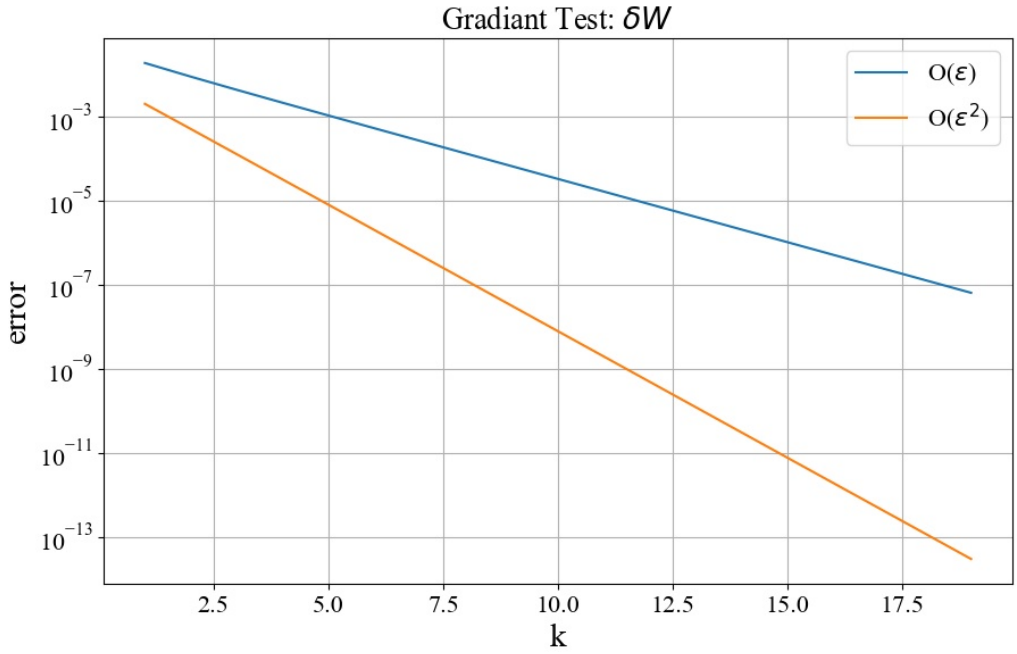

Figure 1.1: Gradient Test with respect to W. Iterating over different k's which represent the power of ε. The initial value is $\varepsilon = 0.5$.

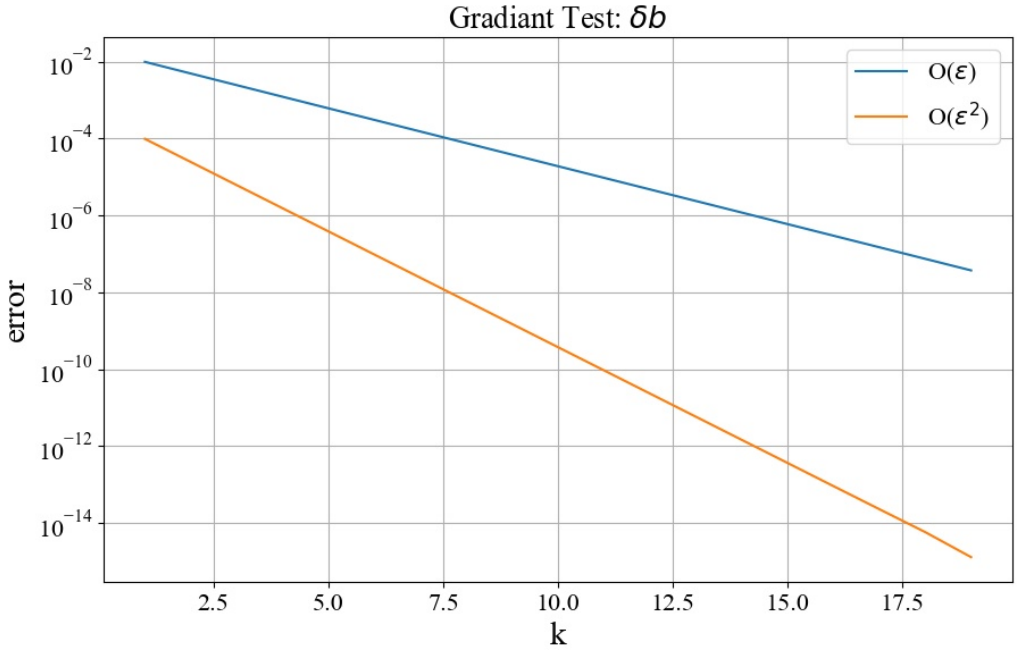

Figure 1.2: Gradient Test with respect to b (the biases). Iterating over different k's which represent the power of ε. The initial value is $\varepsilon = 0.5$.

The figures ensure that we are computing the gradients correctly.

1.2 SGD for Least Squares

In this section we show the Stochastic Gradient Descent (SGD) we implemented for the least squares loss function.
Note: This implementation is **specific** for the Least Squares function (and so is the whole example). We later generalize it for the network to come.

SGD for least squares function's code:

```
import numpy as np

xs = np.arange(-1, 1, 0.1)
ys = []


def f(x):
    return 5 * (x ** 2)


data = [np.transpose([x ** 2, x, 1]) for x in xs]
expectations = [f(x) for x in xs]


def loss(f_, data_, weights):
    data_list = list(data_)
    return sum([f_(x_i, weights, y_i) for (x_i, y_i) in data_list])
    / len(data_list)


def lls_func(x, w_, y):
    return (1 / 2) * (np.transpose(x) @ w_ - y) ** 2


# The gradient.
def LLS_grad(x, w_, y):
    return (np.transpose(x) @ w_ - y) * x


def SGD(mini_loss_func, f_grad, data, expectations, mb_size,
   max_epochs, lr):
    loss_hist = []
    weights = [1, 1, 1]  # initial weights.
    for k in range(max_epochs):
        # Devide data to mini-batches. TBD
        num_of_mbs = int(len(data) / mb_size)
        for j in range(num_of_mbs):
            x_j = data[j]  # Array of 3 - x^2, X, 1.
            y_j = expectations[j]  # Expectation value.
            grad = f_grad(x_j, weights, y_j)
            weights = weights - lr * grad
        loss_hist += [loss(mini_loss_func, zip(data, expectations),
    weights)]
    return weights, loss_hist


```

```
epochs = 1000
w, l = SGD(lls_func, LLS_grad, data, expectations, 1, epochs, 0.01)
```

The result of the optimization is shown bellow:

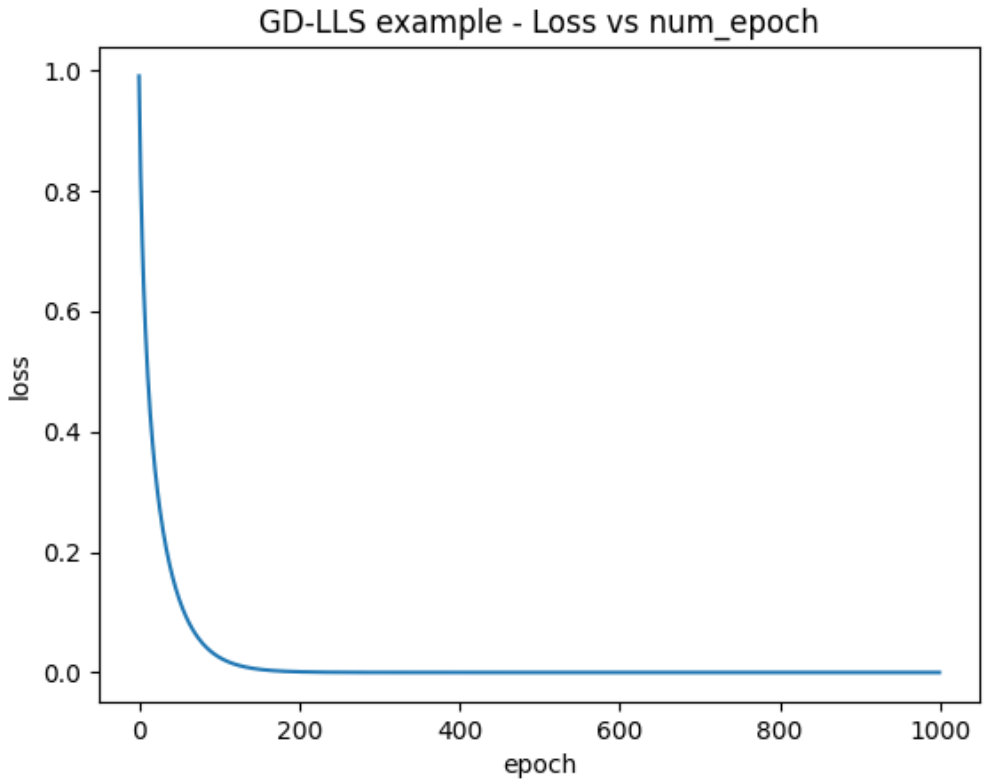

Figure 1.3: SGD for least squares. Iterating over 1000 epochs.

As we can see, the minimization of the Least-Squares function using the SGD algorithm indeed works.

1.3 SGD for softmax

We implemented the SGD for the softmax regression function as a method in the *util* module. We measured the loss and success percentage for different mini-batch sizes and different learning rates, on the Swiss-Roll data-set. We wrote the following code for the Softmax function:

Softmax regression code:

```
def soft_max_regression(X: np.array, W: np.array, C: np.array, b: np.array):
    """
    Computing the loss function 'Soft-Max regression'.
    :param X. The data input as a matrix of size nXm
    :param W. The weights, size of lXn (where l is the amount of labels)
    :param C. Indicators matrix. size of mXl.
    :return the loss function, and the gradients with respect to X, W.
    """
    expr = (W @ X + b).T  # m X l
    arg = expr - etta(expr)  # m X l
    prob = np.exp(arg) / np.sum(np.exp(arg), axis=1).reshape(-1, 1)
    m = len(X.T)
    F = - (1 / m) * np.sum(C * np.log(prob))
    grad_W = (1 / m) * (X @ (prob - C)).T
    grad_X = (1 / m) * (W.T @ (prob - C).T)
    grad_b = (1 / m) * np.sum((prob - C).T, axis=1).reshape(-1, 1)
    return F, grad_W, grad_X, grad_b
```

SGD for softmax code:

```
def SGD_for_Softmax(X, W, b, C, mb_size, max_epochs, lr):
    """
    :param loss_func: loss function to be evaluated.
    :param loss_func_grad: gradient of loss function.
    :param X: X matrix.
    :param W: W matrix.
    :param b: biases.
    :param C: Indicators matrix.
    :param mb_size: batches size.
    :param max_epochs: Number of epochs.
    :param lr: learning rate.
    :return: the value of W after the GD, and the loss for each epoch.
    """
    loss = []
    prob = []
    for k in range(max_epochs):
        bchs = util.generate_batches(X.T, C, mb_size)
        # Partition the data to random mini-batches of size mb_size.
        for curr_Mb, curr_Ind in bchs:
            # curr_Mb is a matrix of size n X mb_size.
```

```
            # curr_Ind is a matrix of size mb_size X 1.
            _, grad, _, _ = util.soft_max_regression(curr_Mb, W,
    curr_Ind, b)
            W -= lr * grad
        l, p = util.calc_probs(X, W, C, b)
        loss += [l]
        prob += [util.get_accuracy(p, C)]
    return W, loss, prob
```

** Auxiliary codes (such as 'generate_batches') are attached in section 4

The results are shown bellow:

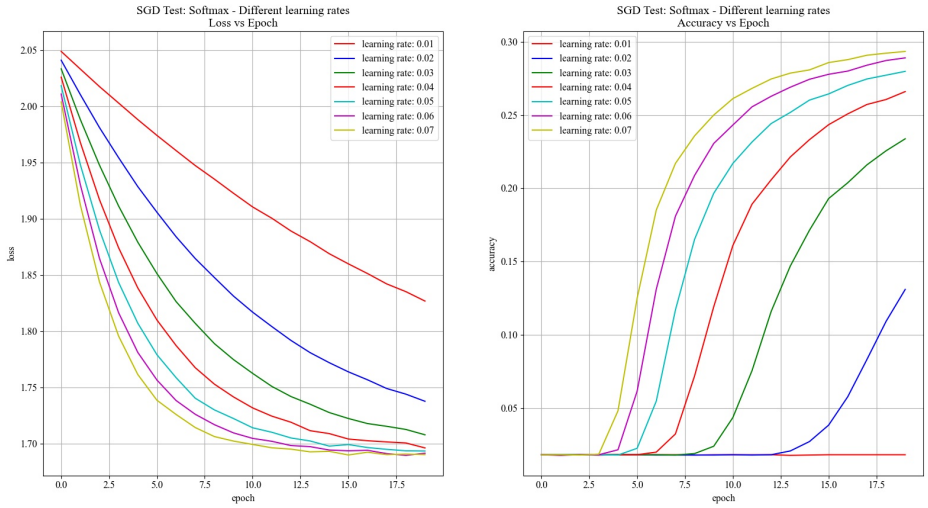

Figure 1.4: SGD for Softmax, on the training data. We iterated over 20 epochs for different learning rates, with mini-batch size of 30.

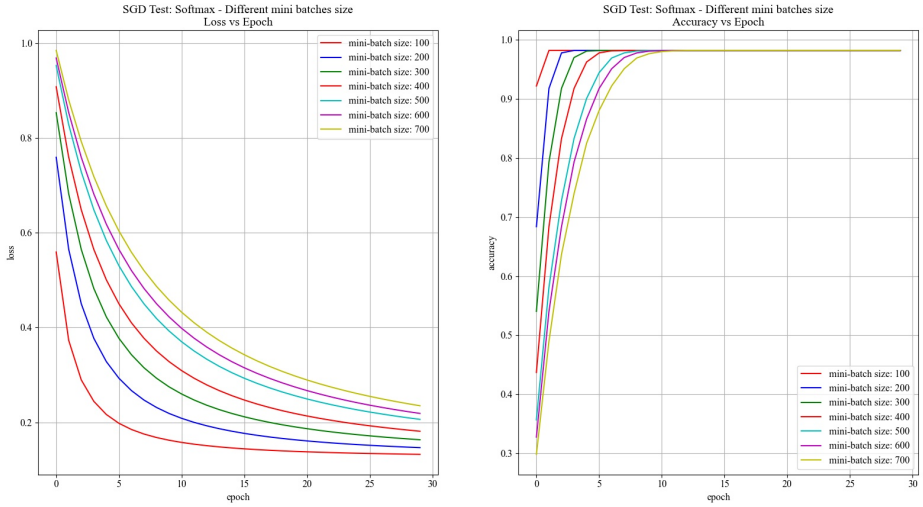

Figure 1.5: SGD for Softmax, on the training data. We iterated over 20 epochs for different mini-batches size, with a learning rate of 0.05.

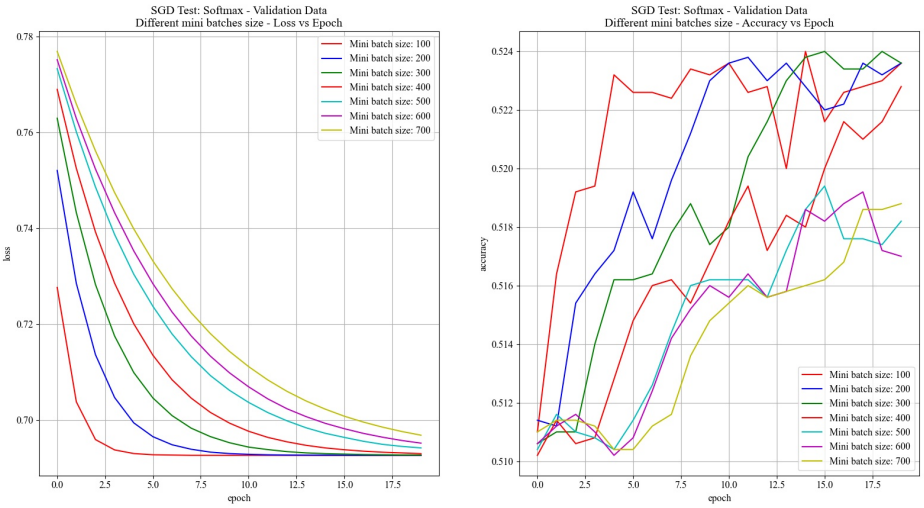

Figure 1.6: SGD for Softmax, on the validation data. We iterated over 20 epochs for different mini-batches size, with a learning rate of 0.05.

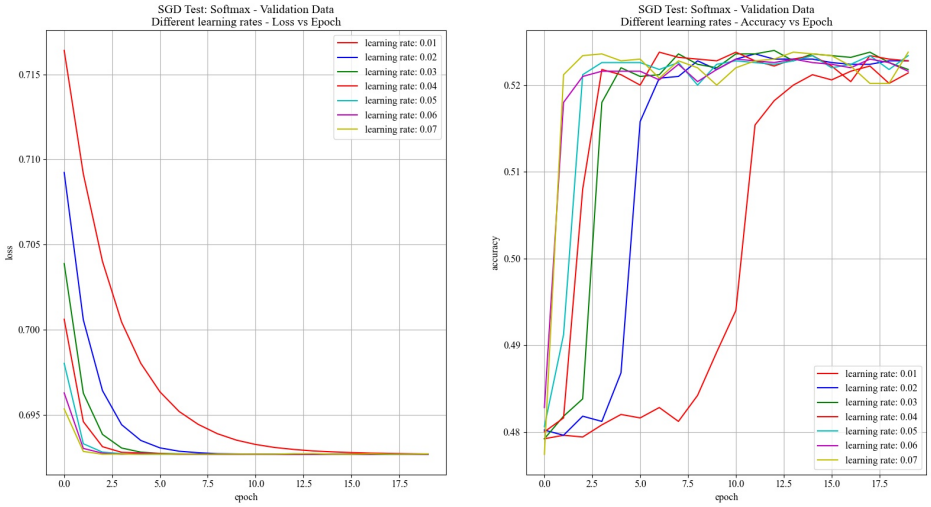

Figure 1.7: SGD for Softmax, on the validation data. We iterated over 20 epochs for different learning rates, with a mini-batches size of 30.

As we can see from the graphs above, the smaller mini-batches sizes and higher learning rates lead for higher accuracy. The latter is partly due to the range of learning rates we tested, as it is clear the for very high learning rates - SGD works badly (the zigzag effect).

2 Part II: The neural network

2.1 Jacobian test - regular layer

We now implement the whole neural-network (initialization, forward-pass, backward-pass, optimization loop).
We present here the Jacobian test we did for W and b for the regular (feed-forward) layer. As mentioned in the assignment we used the Tanh activation function, as it is differentiable.
The code for the test is shown bellow:

Jacobian Test code:

```
def JacTMV(X: np.array, W: np.array, b: np.array, V: np.array, active_func: ActiveFunc, policy):
    temp = W @ X + b
    arg = active_func.deriv(temp) * V
    dict_jac = {"W": arg @ X.T, "X": W.T @ arg, "b": np.sum(arg, axis=1).reshape(-1, 1)}
    return dict_jac[policy]

def jacobian_test(X: np.array, W: np.array, b: np.array, active_func: ActiveFunc, policy):
    """"""
    Gradient test with respect for W.
    :param X matrix.
    :param W matrix.
    :param b matrix.
    :param active_func. The activation function (can be Relu or Tanh).
    :param policy. Indicates to which parameter we are doing the test.
    :return matplotlib graph which shows the jacobian test.
    """"""
    dict_param = {"W": W, "X": X, "b": b}
    dict_name = {"W": "$\delta W$", "X": "$\delta X$", "b": "$\delta b$"}
    d = np.random.randn(*dict_param[policy].shape)

    V = np.random.randn(W.shape[0], X.shape[1])

    d_vector = d.reshape(-1, 1)
    err_1 = []
    err_2 = []
    ks = []
    g_x = sum([arr[i] for i, arr in enumerate(active_func.activ(W @ X + b) @ V.T)])
    dict_args = {"W": lambda: (W + epsilon * d) @ X + b, "X": lambda: W @ (X + epsilon * d) + b,
                 "b": lambda: W @ X + (b + epsilon * d)}
    for k in range(1, 20):
        epsilon = 0.5 ** k
        jac_v = JacTMV(X, W, b, V, active_func, policy).reshape(-1, 1)
```

```
33          g_x_d = sum([arr[i] for i, arr in enumerate(active_func.
    activ(dict_args[policy]()) @ V.T)])
34          err_1.append(abs(g_x_d - g_x))
35          err_2.append(abs(g_x_d - g_x - (epsilon * d_vector.T @
    jac_v)[0][0]))
36          ks.append(k)
37      print_test(ks, err_1, err_2, "Jacobian Test: " + dict_name[
    policy])
```

The results we receive:

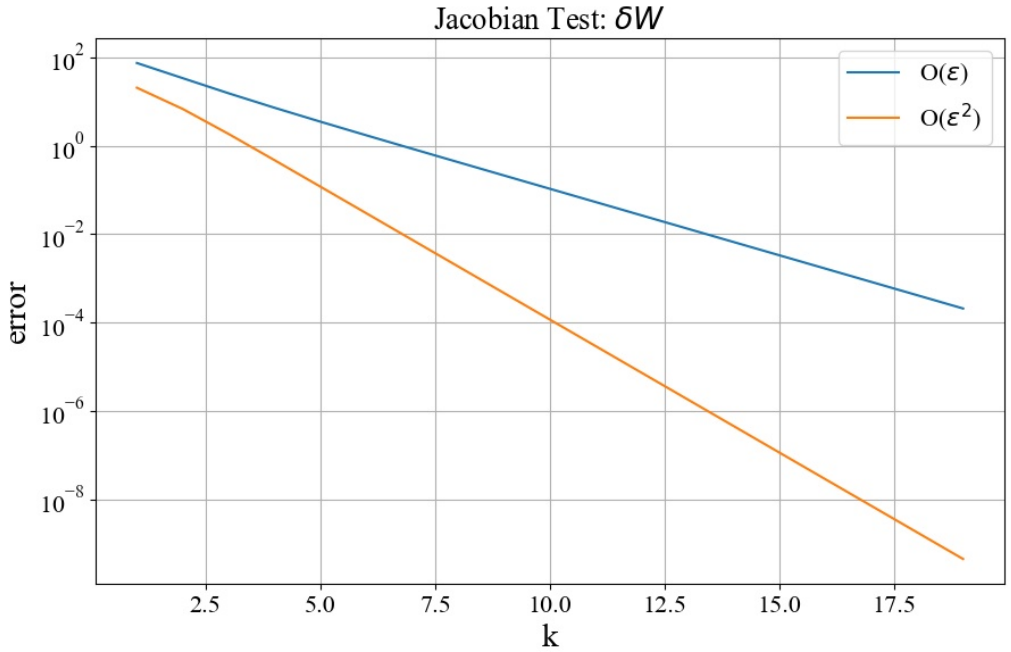

Figure 2.1: Jacobian test with respect to W. Iterating over different k's which represent the power of ε. The initial value is $\varepsilon = 0.5$.

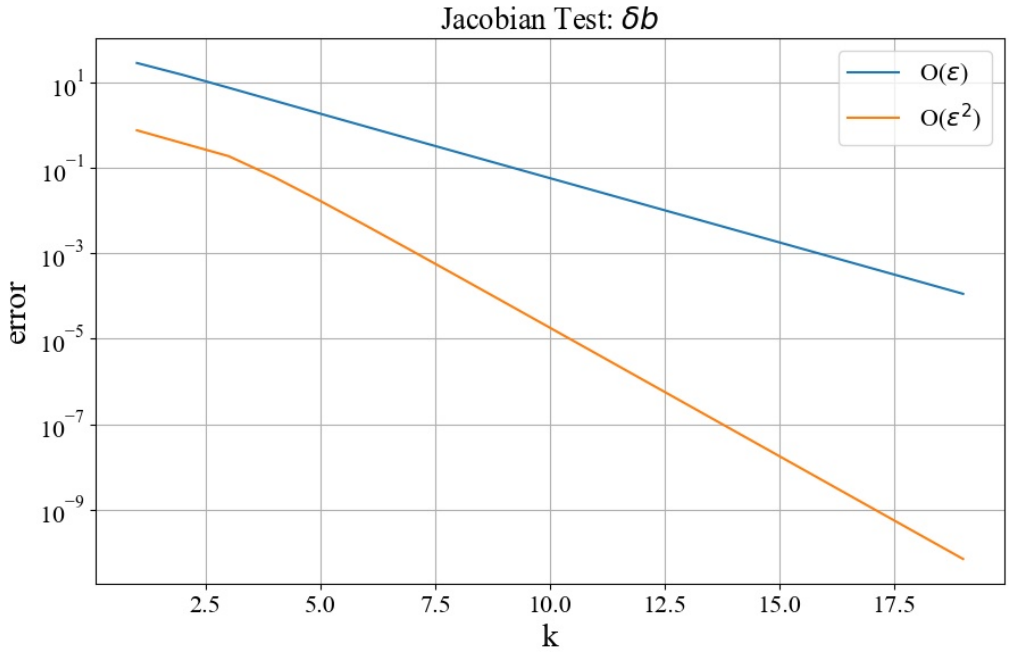

Figure 2.2: Jacobian test with respect to b. Iterating over different k's which represent the power of ε. The initial value is $\varepsilon = 0.5$.

The results ensure that our derivatives are indeed exact.

2.2 ResNet Jacobian test

We present here the Jacobian test we did for the **ResNet** layer with respect to W and b.
The code for the test is shown bellow:

ResNet Jacobian Test code:

```
def resnetJacTMV(X: np.array, W1: np.array, W2: np.array, b: np.array, V: np.array, active_func: ActiveFunc, policy):
    arg = active_func.der(W1 @ X + b) * (W2.T @ V)
    dict_jac = {"W1": arg @ X.T, "W2": V @ (active_func.act(W1 @ X + b)).T, "X": V + W1.T @ arg,
                "b": np.sum(arg, axis=1).reshape(-1, 1)}
    return dict_jac[policy]

def resnet_jacobian_test(X: np.array, W1: np.array, W2: np.array, b: np.array, active_func: ActiveFunc, policy):
    """
    :param X: data matrix.
    :param W1: first weights matrix.
    :param W2: second weights matrix.
    :param b: bias matrix.
    :param active_func: activation function.
    :param policy: Indicates to which parameter we are doing the test.
    :return: matplotlib graph which shows the jacobian test.
    """
    dict_param = {"W1": W1, "W2": W2, "X": X, "b": b}
    dict_name = {"W1": "$\delta W_1$", "W2": "$\delta W_2$", "X": "$\delta X$", "b": "$\delta b$"}
    d = np.random.randn(*dict_param[policy].shape)

    V = np.random.randn(W2.shape[0], X.shape[1])

    d_vector = d.reshape(-1, 1)
    err_1 = []
    err_2 = []
    ks = []
    g_x = sum([arr[i] for i, arr in enumerate((X + W2 @ active_func.activ(W1 @ X + b)) @ V.T)])
    dict_args = {"W1": lambda: X + W2 @ active_func.activ((W1 + epsilon * d) @ X + b),
                 "W2": lambda: X + (W2 + epsilon * d) @ active_func.activ(W1 @ X + b),
                 "X": lambda: (X + epsilon * d) + W2 @ active_func.activ(W1 @ (X + epsilon * d) + b),
                 "b": lambda: X + W2 @ active_func.activ(W1 @ X + (b + epsilon * d))}
    for k in range(1, 20):
        epsilon = 0.5 ** k
        jac_v = resnetJacTMV(X, W1, W2, b, V, active_func, policy).reshape(-1, 1)

        g_x_d = sum([arr[i] for i, arr in enumerate(dict_args[policy]() @ V.T)])
```

```
        err_1.append(abs(g_x_d - g_x))
        err_2.append(abs(g_x_d - g_x - (epsilon * d_vector.T @
jac_v)[0][0]))
        ks.append(k)
    print_test(ks, err_1, err_2, "Resnet Jacobian Test: " +
dict_name[policy])
```

The results are shown bellow:

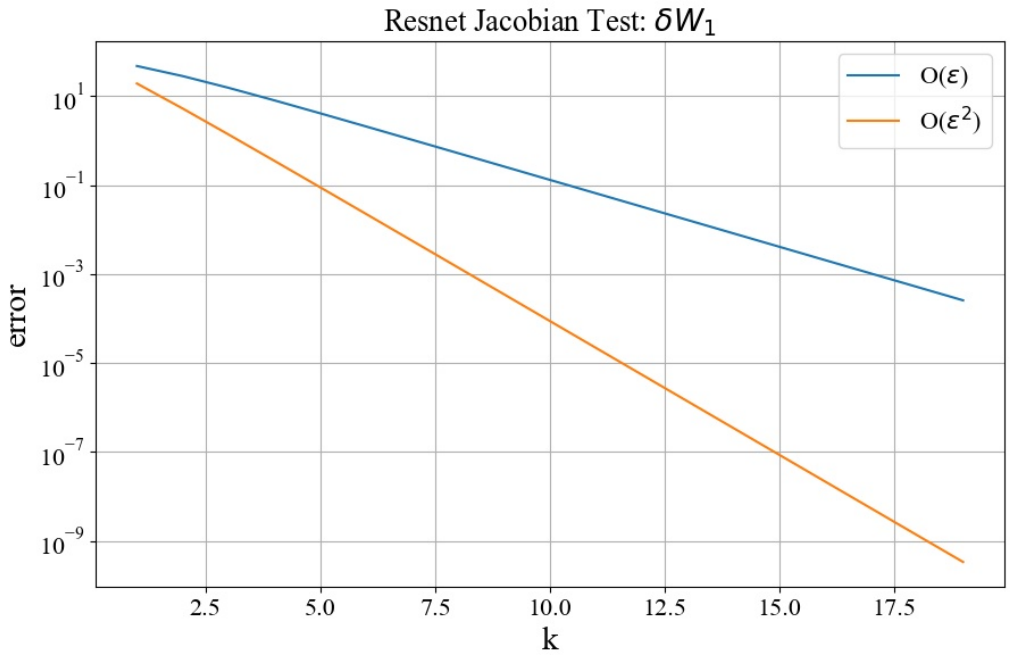

Figure 2.3: ResNet Jacobian test with respect to W_1. Iterating over different ks which represent the power of ε. The initial value is $\varepsilon = 0.5$.

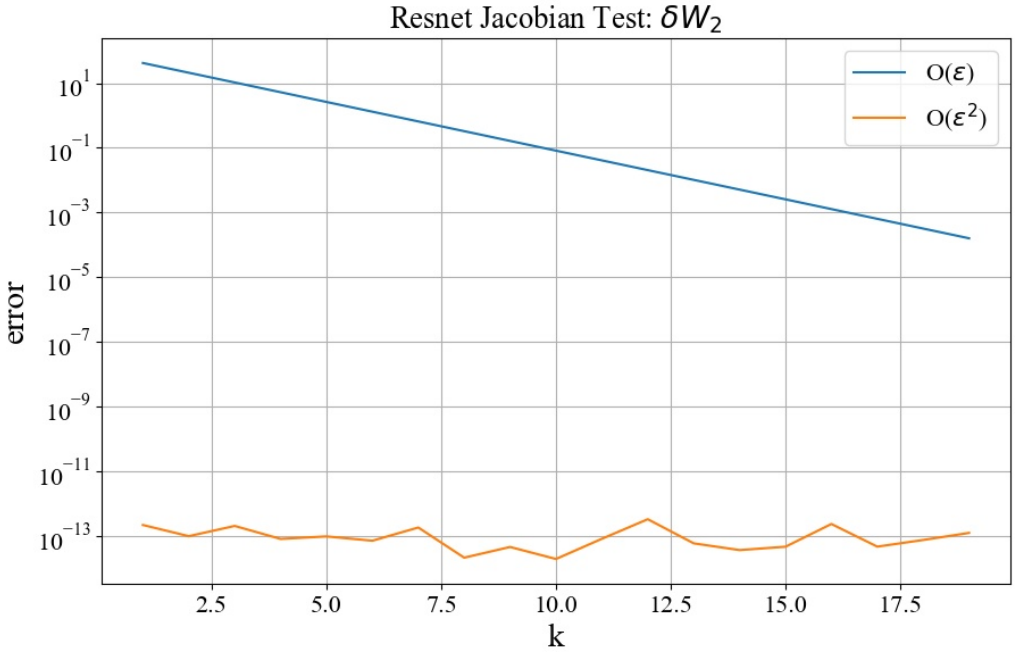

Figure 2.4: ResNet Jacobian test with respect to W_2. Iterating over different ks which represent the power of ε. The initial value is $\varepsilon = 0.5$.

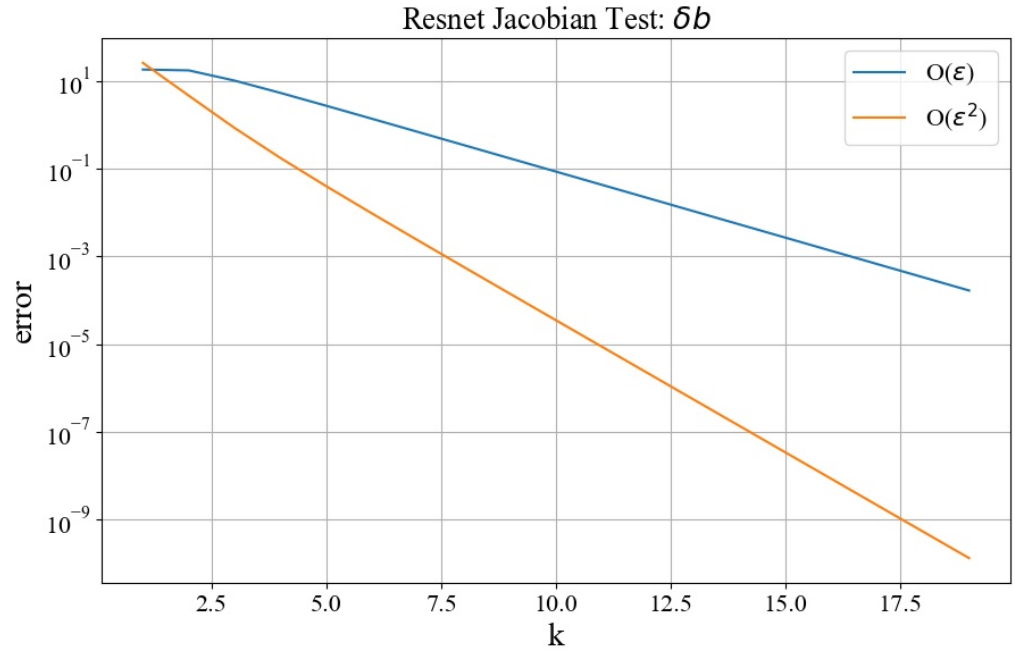

Figure 2.5: ResNet Jacobian test with respect to b. Iterating over different ks which represent the power of ε. The initial value is $\varepsilon = 0.5$.

It seems that the Jacobian test for W_2 in Fig. 2.4 has an anomaly, but in fact it is the value we expected to see for W_2 because it's second derivative is zero (and in our program, zero represented as the smallest value we can reach with 64 bit

Floating-Point representation). For the rest of the tests, as before, we conclude that our derivatives are correct.

2.3 Neural network Gradient test

We present here the Gradient test we did for the whole network with respect to W and b combined. The code for the test is shown bellow:

Network Gradient Test code:

```
def nn_gradient_test(nn, X, C):
    """
    :param nn: The Neural Network.
    :param X: Data matrix.
    :param C: Indicators matrix.
    :return: matplotlib graph which shows the gradiant test.
    """
    Ws = []
    dWs = []
    bs = []
    dbs = []

    nn.forward(X)
    nn.backward(C)
    for layer in nn.layers:
        Ws.append(layer.W.copy())
        dWs.append(layer.grad_W.copy())
        bs.append(layer.b.copy())
        dbs.append(layer.grad_b.copy())
    f_x, _ = nn.calc_loss_probs()

    ds_W = [np.random.randn(*w.shape) for w in Ws]
    ds_b = [np.random.randn(*b.shape) for b in bs]

    err_1 = []
    err_2 = []
    ks = []

    sum_grad = 0
    for i, layer in enumerate(nn.layers):
        sum_grad += (ds_W[i].reshape(1, -1) @ layer.grad_W.reshape
(-1, 1))[0][0]
    for i, layer in enumerate(nn.layers):
        sum_grad += (ds_b[i].reshape(1, -1) @ layer.grad_b.reshape
(-1, 1))[0][0]

    for k in range(1, 10):
        epsilon = 0.5 ** k
        for i, layer in enumerate(nn.layers):
            layer.set_W(Ws[i] + epsilon * ds_W[i])
        for i, layer in enumerate(nn.layers):
            layer.set_b(bs[i] + epsilon * ds_b[i])
        nn.forward(X)
        f_x_d, _ = nn.calc_loss_probs()

```

```
        err_1.append(abs(f_x_d - f_x))
        err_2.append(abs(f_x_d - f_x - epsilon * sum_grad))
        ks.append(k)
    print_test(ks, err_1, err_2, "Network Gradiant Test")
```

The results are shown bellow:

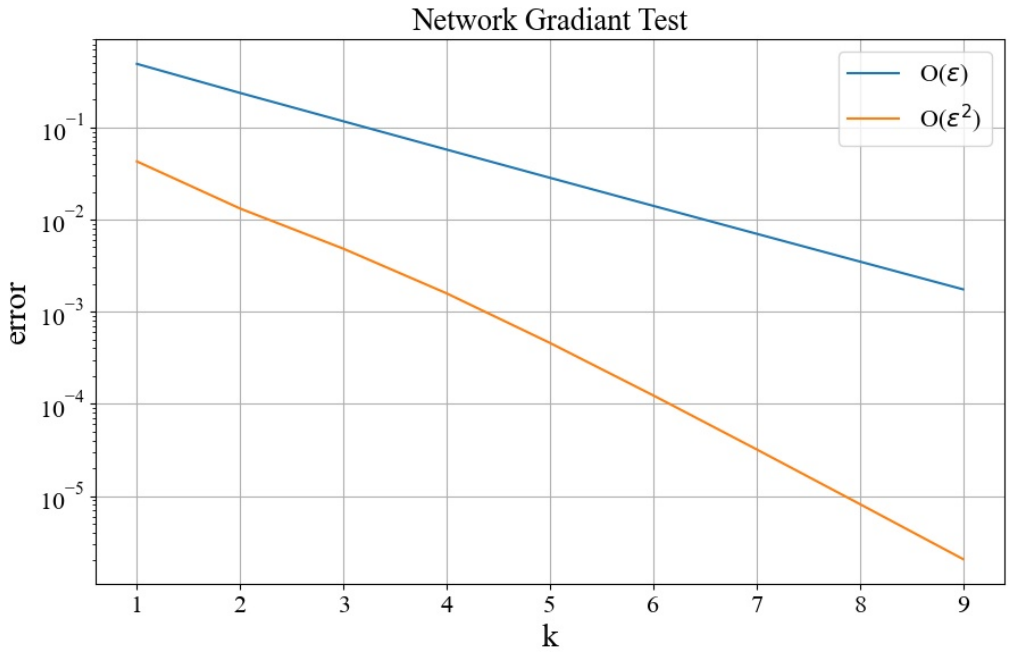

Figure 2.6: ResNet Jacobian test with respect to W_1. Iterating over different ks which represent the power of ε. The initial value is $\varepsilon = 0.5$.

The gradient test for the whole network ensures that we are computing the gradient correctly.

2.4 Learning of the whole neural network

2.4.1 Training

We show here the results we received for different hyper-parameters on the **training data** of the **Swiss roll** data-set.

We define the following architectures:

- Regular net: 8 layers with sizes: $[n, 3 \cdot n, 3 \cdot n, 5 \cdot n, 5 \cdot n, 3 \cdot n, 3 \cdot n, n]$.
  Note: Regular net 2 is in the figures below has the exact same architecture as the above net. The difference is the mini-batch size we use.
- ResNet: A net of 15 layers, of the sizes: $[n, 2 \cdot n, 3 \cdot n, 4 \cdot n, 3 \cdot n, 2 \cdot n, n, n, n, 2 \cdot n, 3 \cdot n, 4 \cdot n, 3 \cdot n, 2 \cdot n, n]$ where the middle layer (of size n) is a Residual layer.
- Big Net: 8 layers with sizes: $[n, 12 \cdot n, 12 \cdot n, 144 \cdot n, 144 \cdot n, 12 \cdot n, 12 \cdot, n]$.
- Full ResNet: 8 layers, all of the same size, all layers except the first and last are Residual layers.

Note: n refers to the input data dimension.

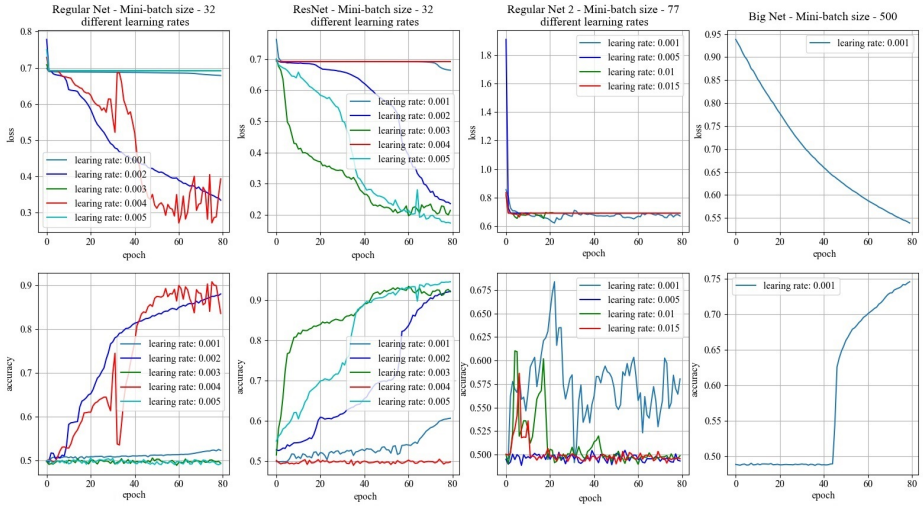

Figure 2.7: Loss and accuracy of the whole network, for the Swiss-Roll training data-set. Layer's size are fixed, varying the learning rate, mini-batch size and adding Resnet layers.

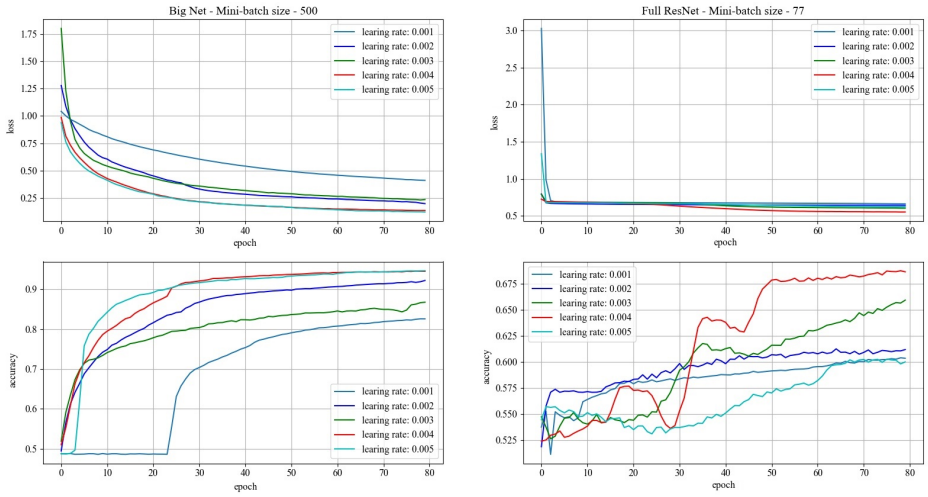

Figure 2.8: Training data of Swiss roll. Loss and accuracy for the whole network. The left side focus's on mini-batch size of 500 with large layers size (as described in the beginning of this section). In the right side we have 8 layers, six of which are ResNet layers (all but first and last).

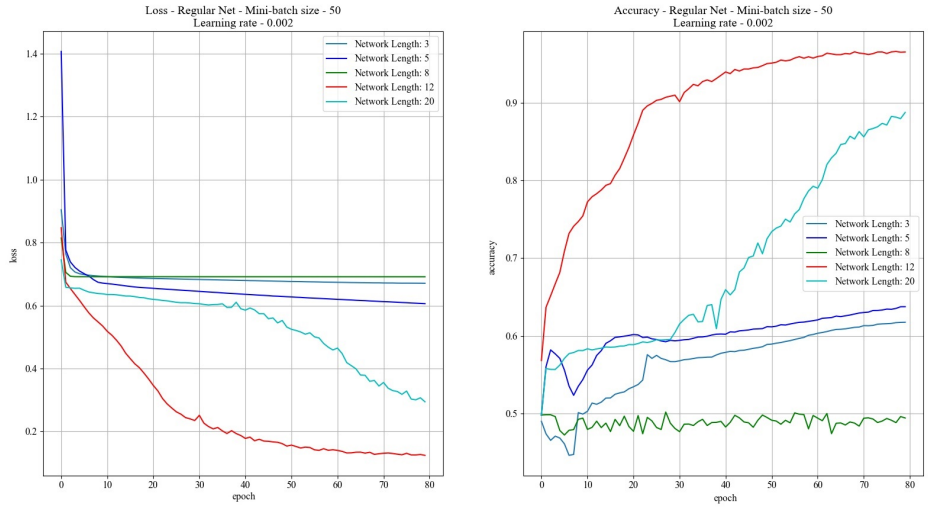

Figure 2.9: Loss and accuracy for the whole network. Different layers length.
For all network lengths, we monotonically increase the layer size as we get to the middle of the net, and then we decrease it back as we get the end (as done for the Regular Net described above).

From the above research, we conclude that the best network hyper-parameters and length are:

- Network length: 12.
- Learning rate: 0.03.
- Mini-batch size: 50.

We shall use these parameter values for the learning to come on the data-sets provided.

2.4.2 Swiss Roll data

After finding the best hyper-parameters - we now **train with them** and check the resulting accuracy on the validation data.
As mentioned above, the hyper-parameters we use are network length of 12 (layers), a learning rate of 0.003 and mini-batch size of 50.
We emphasize that although the accuracy and loss are calculated for the validation data as well as for the training data, learning is performed only on the training data, i.e., the weights are not changed due to the values from the validation-data.

The results are shown bellow:

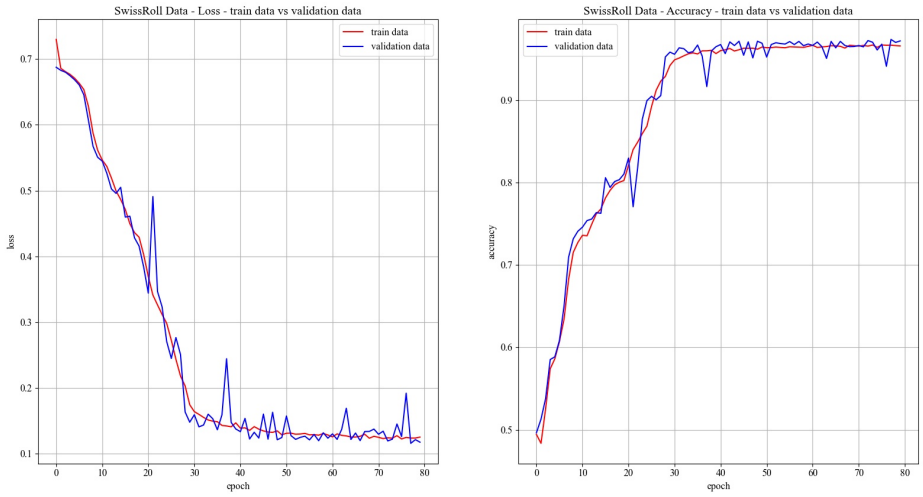

Figure 2.10: Swiss Roll validation data vs training data. After iterating over the whole data-set (in each epoch), we evaluated the loss and probability matrix of the validation data without learning from them.

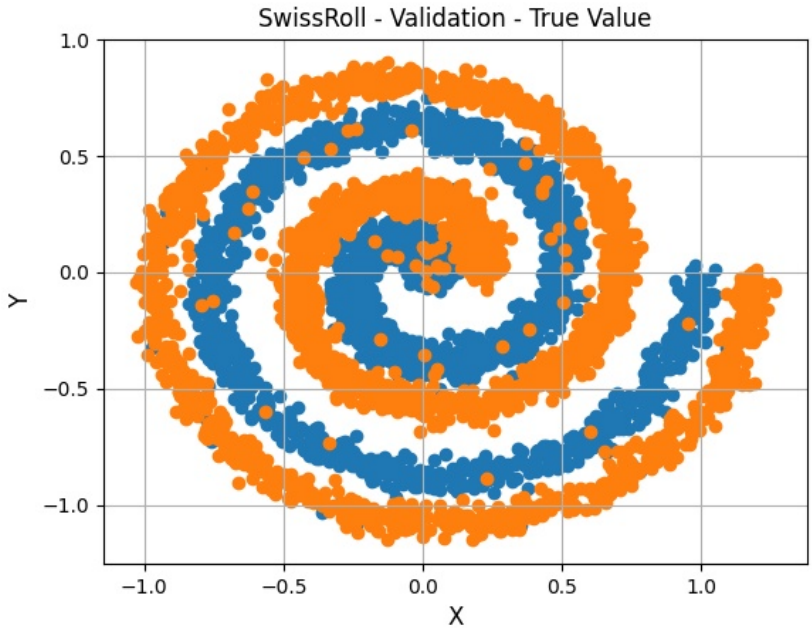

Figure 2.11: Swiss Roll raw validation data.

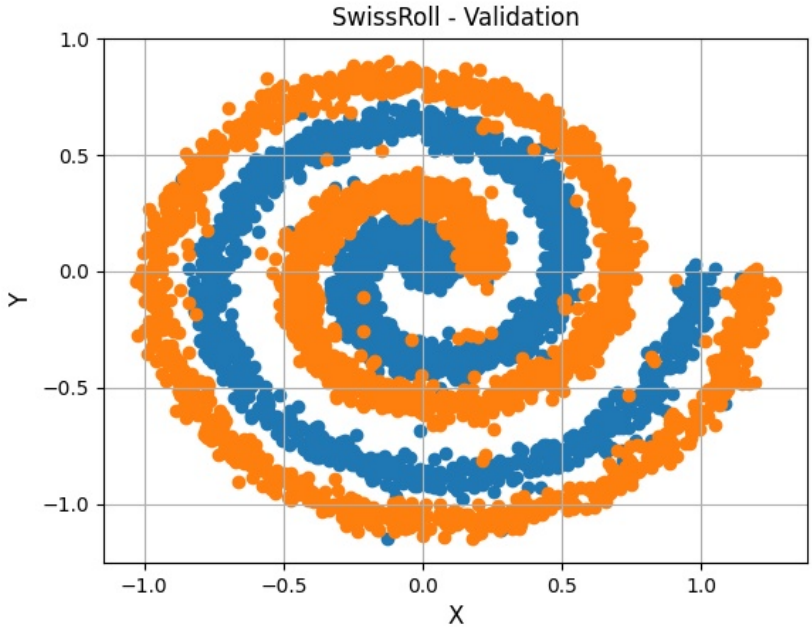

Figure 2.12: Our network's classification of the Swiss Roll validation data. This image represents the values of the points our network learned (classified). The network was trained on 12 layers with mini-batch size of 50 and learning rate of 0.002.

2.4.3 Peaks Data

We show here the result we achieved for the same network architecture in the Swiss Roll case (12 layers, learning rate 0.003 and 50 mini-batch size). Same as in Sec. 2.4.2, we trained our network only on the train data-set and evaluated the performance of the validation data-set compared to the train data-set (see Fig. 2.13).

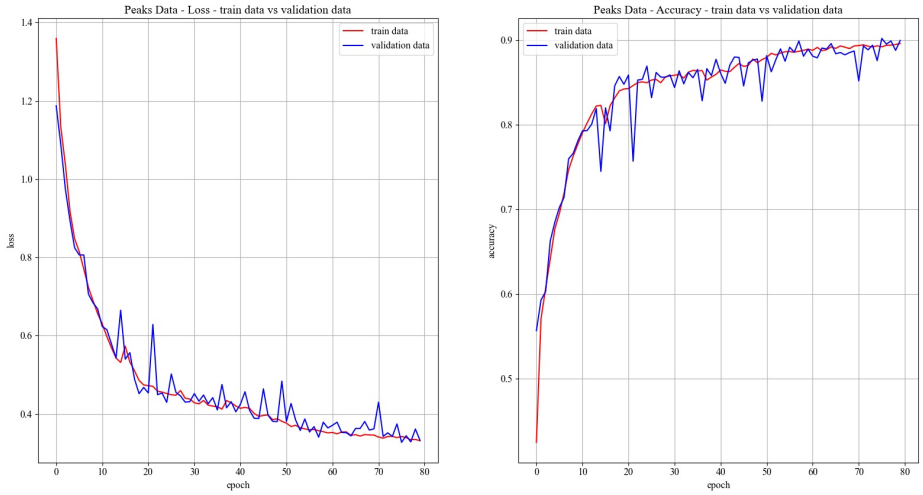

Figure 2.13: Peaks validation data vs training data. We evaluated in each epoch the loss and probability matrix of the validation data and plot it in the graph.

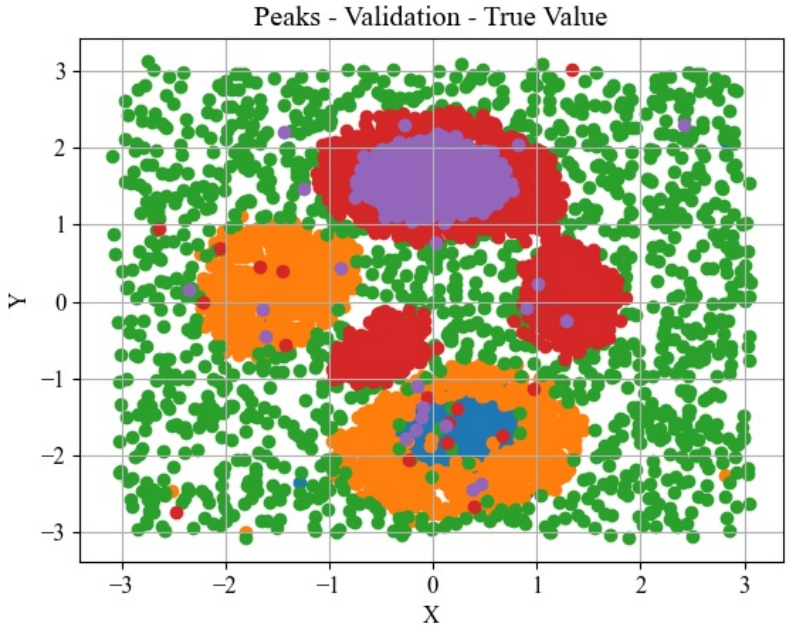

Figure 2.14: Peaks validation raw data.

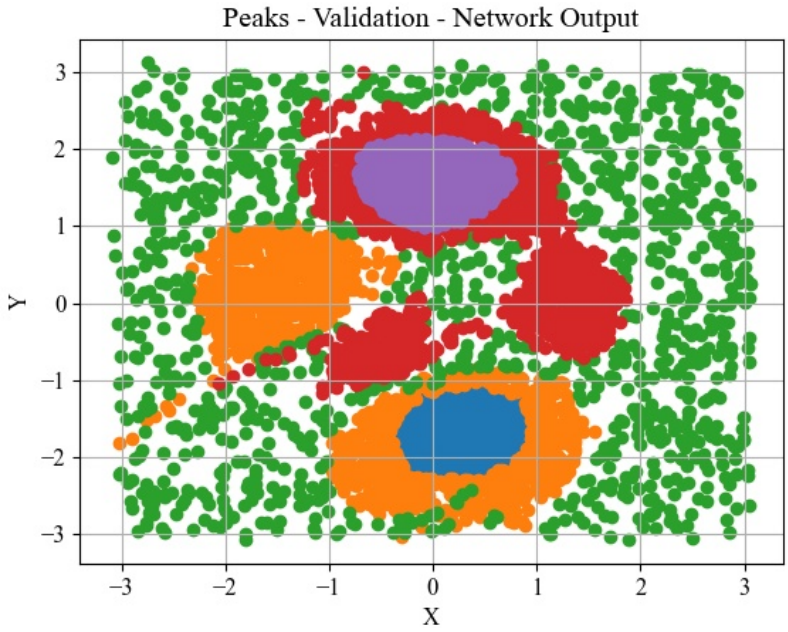

Figure 2.15: Peaks validation data. This graph represent the values of the points (labels) our network learned. The network was trained on 12 layers network with mini-batch size of 50 and learning rate of 0.003.

2.4.4 GMM Data

We show here the results we achieved for the same configuration as the Swiss Roll and the Peaks data (with 12 layers, 0.003 learning rate and 50 mini-batch size). Further explanation in Sec. 2.10.

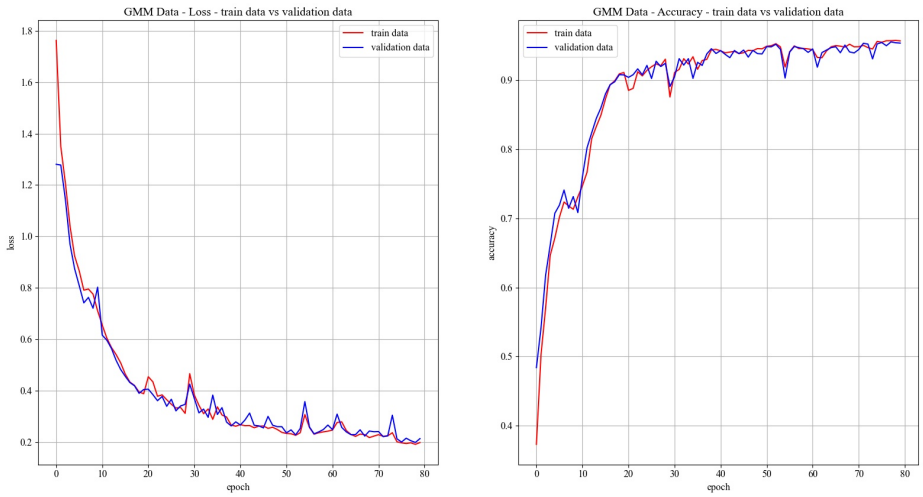

Figure 2.16: GMM validation data vs training data. We evaluated in each epoch the loss and probability matrix of the validation data.

2.4.5 Learning with 200 data points

We repeated the same procedure as in Sec. 2.4.2, 2.4.3, 2.4.4, with a training data set of 200 points. The results are shown bellow (Fig. 2.17, 2.18 and 2.19):

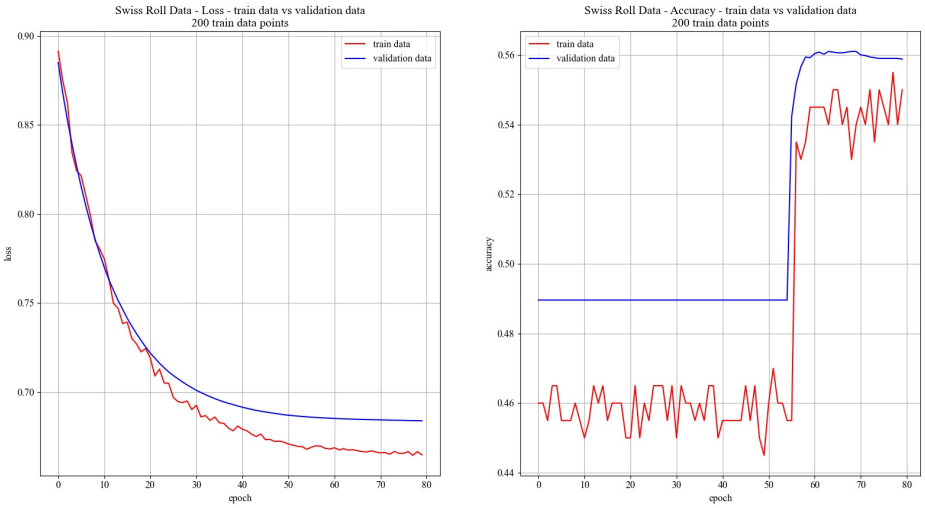

Figure 2.17: Swiss Roll validation data vs training data of 200 data points. We evaluated in each epoch the loss and probability matrix of the validation data.

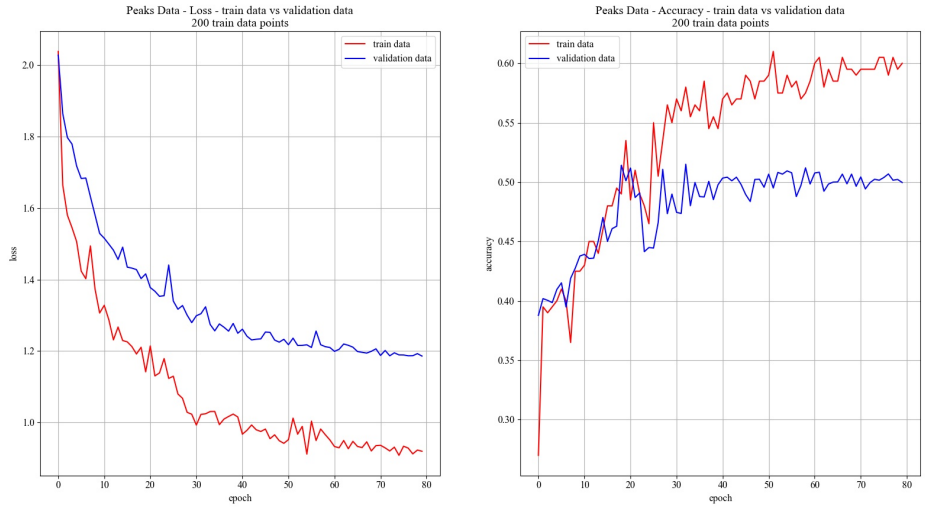

Figure 2.18: Peaks validation data vs training data of 200 data points. We evaluated in each epoch the loss and probability matrix of the validation data.

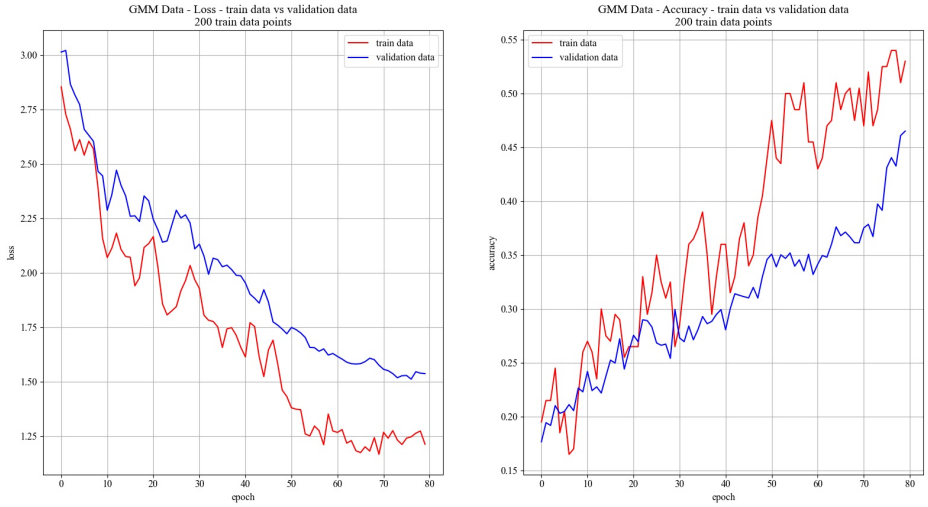

Figure 2.19: GMM validation data vs training data of 200 data points. We evaluated in each epoch the loss and probability matrix of the validation data.

3 Conclusions

1. We see that when we make sure of all our calculations using the gradient and Jacobian test (see Sec. 2.3), we indeed get a learning network, as wanted (emphasizes the importance of checking our derivatives).

2. As we can see in Fig. 2.7, as we add layers and residual layers to our network, we get smoother accuracy curves (then the regular network). This strengthens the claim that when we add more layers and neurons to our network, we can calculate more complex functions (such as the classifier for the Swiss-Roll data-set).

3. As we tested our network for different hyper-parameters for different networks, we saw that their effect on the loss and accuracy varies as we change the network architecture. Their inconsistent effect makes us understand that we need to analyze their effect in an architecture-specific manner (for example, fixing the randomness seed).

4. The results we receive are highly inconsistent from execution to execution, even with the same hyper-parameters (from very 'good' learning to very 'bad'). This emphasizes the effect of the randomness of our initial weights, and our randomly picked seed.

5. From Fig. 2.9 we can see that the best results achieved with 12 layers network. We chose this network rather than the big sized network which also preformed well (see Fig. 2.8) due to the shorter time complexity of this network (big matrices cost longer running time).

6. As we can see from Fig. 2.12 and 2.15, the network seems to 'fix' the raw data, i.e., its fluctuations vanish in the network output.

7. ResNet seems to have good loss reduction but it's accuracy is fluctuated and therefore we did not use it.

8. For all data samples, the comparison between the train network output and validation network output seems consistent (see Fig. 2.10, 2.13 and 2.16), with slightly small fluctuations of the validation graph.

9. We noticed that the correlation between the loss and the accuracy (see Fig. 2.7, 2.8) is not as strong as we initially thought. Smooth and exponentially decaying loss curve does not guarantee exponentially arising accuracy curve. Nevertheless, consistent behaviour of those two parameters increase the confidence and reliability of the architecture that we are choosing.

10. For all 3 data-sets we used (Swiss-Roll, Peaks, GMM) our network was able to achive accuracy of at least 90%!

11. Upon learning on 200 data-points only (see Sec. 2.4.5), we deduce that the extensive data-set (i.e., a lot of data-points) is a crucial ingredient for sufficient learning.

Furthermore, we witness larger fluctuations in the values of the loss and accuracy, as well as smaller correlations between them, relative to the case of a large (extensive) data-set.

4 Auxiliary code

Generate mini-batch code:

```
def generate_batches(X, C, mb_size):
    """
    :param X: data matrix.
    :param C: indicators matrix.
    :param mb_size: batches size.
    :return: array with all the mini batches and their correspondent indicators.
    """
    data = []
    mb = []
    mbs = []
    # Generate 'data' - An array containing 2-component arrays of [data, indicator].
    for i in range(len(X)):
        data.append([X[i], C[i]])  # C[i] is the i'th row, corresponding to the i'th data-sample (it's indicator).
    indices = list(range(len(data)))
    random.shuffle(indices)
    while len(indices) > mb_size:
        for i in range(mb_size):
            mb.append(data[indices.pop()])
        """
        Mb: Array of size nXmb_size.
        Indicator: Matrix of size 1Xmb_size
        """
        Mb, Indicator = functools.reduce(lambda acc, curr: [acc[0] + [curr[0]], acc[1] + [curr[1]]], mb, [[], []])
        mb = []
        mbs += [(np.array(Mb).T, np.array(Indicator))]

    return mbs
```

Accuracy calculation code:

```
def get_accuracy(Prob: np.array, Indicator: np.array):
    """
    Get the probabilities matrix and indicator matrix and calculate the accuracy
    :param Prob:
    :type Prob:
    :param C:
    :type C:
    :return:
    :rtype:
    """
    size = len(Prob)
    probs = np.argmax(Prob, axis=1)
    indicators = np.argmax(Indicator, axis=1)
    counter = sum(probs == indicators)
    return counter / size
```

Probability and Softmax code:

```
def soft_max_regression(X: np.array, W: np.array, C: np.array, b: np.array):
    """
    Computing the loss function 'Soft-Max regression'.
    :param X. The data input as a matrix of size nXm
    :param W. The weights, size of lXn (where l is the amount of labels)
    :param C. Indicators matrix. size of mXl.
    :return the loss function, and the gradients with respect to X, W.
    """
    expr = (W @ X + b).T  # m X l
    arg = expr - etta(expr)  # m X l
    prob = np.exp(arg) / np.sum(np.exp(arg), axis=1).reshape(-1, 1)
    m = len(X.T)
    F = - (1 / m) * np.sum(C * np.log(prob))
    grad_W = (1 / m) * (X @ (prob - C)).T
    grad_X = (1 / m) * (W.T @ (prob - C).T)
    grad_b = (1 / m) * np.sum((prob - C).T, axis=1).reshape(-1, 1)
    return F, grad_W, grad_X, grad_b


def calc_probs(X, W, C, b):
    expr = (W @ X + b).T
    arg = expr - etta(expr)
    prob = np.exp(arg) / np.sum(np.exp(arg), axis=1).reshape(-1, 1)
    m = len(X.T)
    F = - (1 / m) * np.sum(C * np.log(prob))
    return F, prob

def etta(A: np.array):
    """
    This method calculate the etta vector that required to reduce from A in order to prevent numerical overflow.
    :return etta vector. this vector is the column with the maximal norm from A.
    """
    etta = np.array([])
    for a in A:
        etta = np.append(etta, max(a))
    return etta.reshape(-1, 1)
```