Practical deep learning
Assignment 1

Names: Dan Yarden 316611854 and Ofer Moses 311452171

GitHub project:

Part 1: the classifier and optimizer

1. We implemented the soft max regression function as type of layer in the neural network, called “SoftMaxLayer”,which extends the basic Layer class:

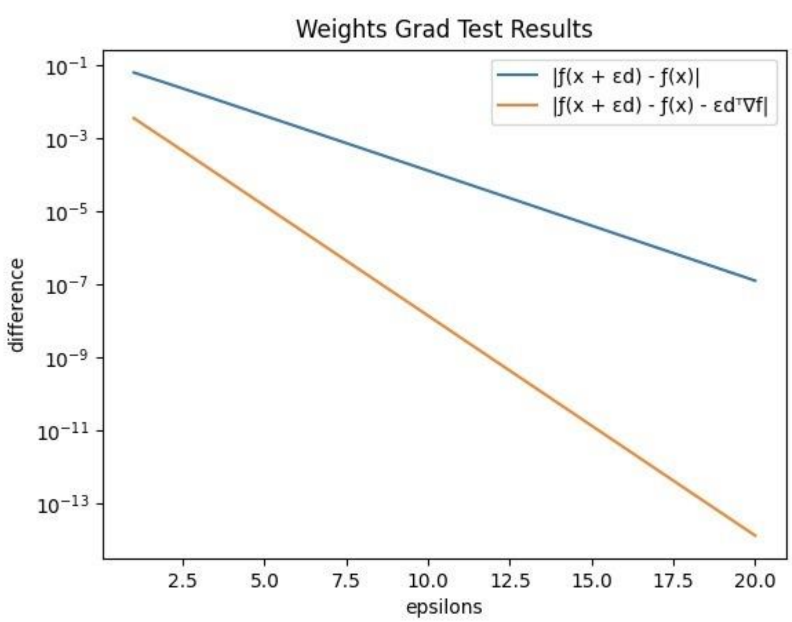
```
class Layer:
    def __init__(self, in_dimensions, out_dimensions, activation=ReLU):
        """

        :param in_dimensions: dimensions of input
        :param out_dimensions: dimensions of output
        :param activation: activation function used by the layer
        """
        self.X = None
        np.random.seed(0)
        self.W = np.random.uniform(-1, 1, size=(out_dimensions,
in_dimensions))
        self.b = np.zeros((out_dimensions, 1))
        self.activation = activation.activate
        self.activation_derivative = activation.deriviative
        self.dX = None  # derivative with respect to input
        self.dW = None  # derivative with respect to Weight
        self.db = None  # derivative with respect to bias
        self.train = True
```

```python
class SoftMaxLayer(Layer):
    def __init__(self, in_dimensions, num_of_classes):
        super(SoftMaxLayer, self).__init__(in_dimensions, num_of_classes)
        self.activation = lambda X: X

    def forward(self, X):
        self.X = X.copy()
        out = self.activation(self.W @ X + self.b)

        return out

    def backward(self, V=None):
        pass

    def soft_max(self, net_out, Y):
        """
        description..
        :param net_out: a matrix of size nxm, output of  forward layer
        :param Y: a matrix of size lxm,
         where Y[i,:] is c_i (indicator vector for label i)
        :return: loss score, and probabilities matrix for each class,x
        """
        W = self.W.T
        n = self.X.shape[0]
        if len(self.X.shape) > 1:
            m = self.X.shape[1]
        else:
            m = 1
        l = self.W.shape[0]
        self.dW = np.zeros((l, n))
        self.db = np.zeros((l, 1))

        ettas_vector = get_ettas(self.X, W, m, self.b)
        scores = exp(net_out - ettas_vector)
        right_sum = np.sum(scores, axis=0)
        probabilities = scores / right_sum
        loss = np.sum(Y * np.log(probabilities))

        # if during training, calculate gradients of loss and save
        if self.train:
            self.dW = (1 / m) * (self.X @ (probabilities - Y).T).T
            self.db = (1 / m) * np.sum((probabilities - Y), axis=1).reshape(-
1, 1)
            self.dX = (1 / m) * (W @ (probabilities - Y))

        return -loss / m, probabilities
```

Gradient Tests for loss:

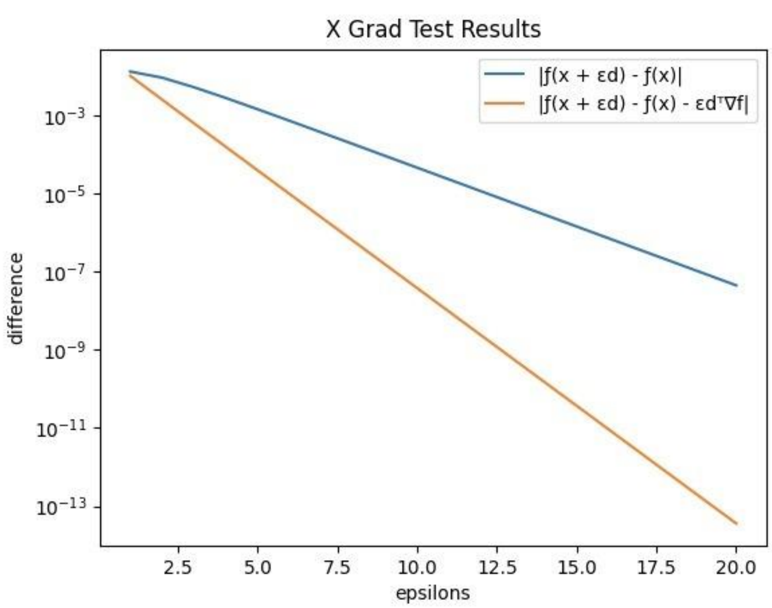
<u>Weights:</u>

```
def grad_test_soft_max_weights(X: np.array, Y: np.array):
    iter_num = 20
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    n = X.shape[0]
    l = Y.shape[0]
    if len(X.shape) > 1:
        m = X.shape[1]
    else:
        m = 1
    soft_max_layer = SoftMaxLayer(n, l)
    d = normalize(np.random.rand(*soft_max_layer.W.shape))
    W_orig = soft_max_layer.W.copy()
    out = soft_max_layer.forward(X)
    fw, _  = soft_max_layer.soft_max(out, Y)
    grad_w = soft_max_layer.dW
    for i, epsilon in enumerate(epsilons):
        W_diff = W_orig.copy()
        W_diff += d * epsilon
        soft_max_layer.W = W_diff
        out_epsilon = soft_max_layer.forward(X)
        fw_epsilon, _  = soft_max_layer.soft_max(out_epsilon, Y)
        diff[i] = abs(fw_epsilon - fw)
        d_flat = d.reshape(-1, 1)
        grads_flat = grad_w.reshape(-1, 1)
        diff_grad[i] = abs(fw_epsilon - fw - epsilon * d_flat.T @ grads_flat)

    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('weights Grad Test Results')
    plt.legend(("diff without grad", "diff with grad"))
    plt.show()
```

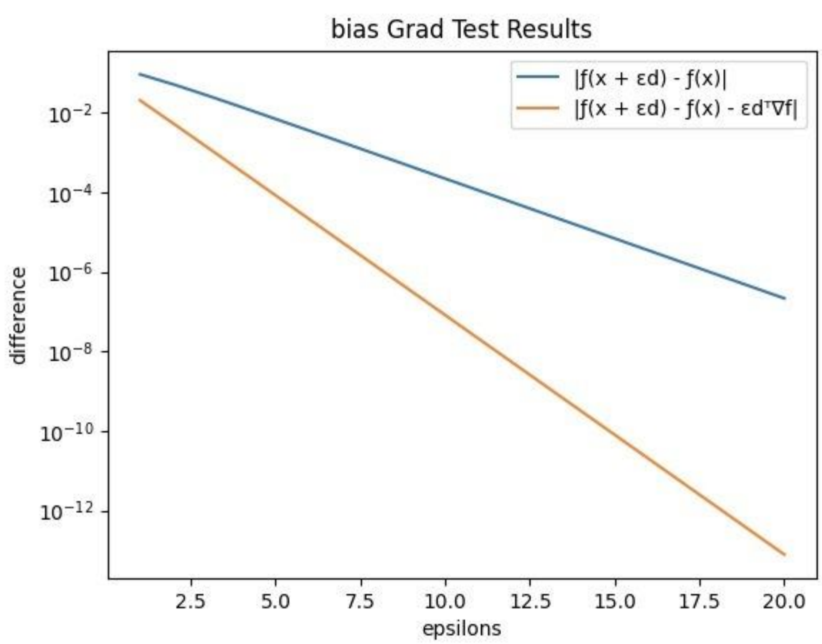


<u>Input(X):</u>

```
def grad_test_soft_max_X(X: np.array, Y: np.array):
    iter_num = 20
    X_orig = X.copy()
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    n = X.shape[0]
    l = Y.shape[0]
    soft_max_layer = SoftMaxLayer(n, l)
    d = normalize(np.random.rand(*X.shape))
    out = soft_max_layer.forward(X)
    fx, _  = soft_max_layer.soft_max(out, Y)
    grad_x = soft_max_layer.dX.copy()
    for i, epsilon in enumerate(epsilons):
        X_diff = X_orig.copy()
        X_diff += d * epsilon
        out_epsilon = soft_max_layer.forward(X_diff)
        fx_epsilon, _ = soft_max_layer.soft_max(out_epsilon, Y)
        diff[i] = abs(fx_epsilon - fx)
        d_flat = d.reshape(-1, 1)
        grads_flat = grad_x.reshape(-1, 1)
        diff_grad[i] = abs(fx_epsilon - fx - epsilon * d_flat.T @ grads_flat)

    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('X Grad Test Results')
    plt.legend(("diff without grad", "diff with grad"))
    plt.show()
```



```
def grad_test_soft_max_bias(X: np.array, Y: np.array):
    iter_num = 20
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    n, m = X.shape
    l = Y.shape[0]
    soft_max_layer = SoftMaxLayer(n, l)
    b_original = soft_max_layer.b.copy()
    d = normalize(np.random.rand(*soft_max_layer.b.shape))
    out = soft_max_layer.forward(X)
    fb, _ = soft_max_layer.soft_max(out, Y)
    grad_b = soft_max_layer.db
    for i, epsilon in enumerate(epsilons):
        b_diff = b_original.copy()
        b_diff += d * epsilon
        soft_max_layer.b = b_diff
        out_epsilon = soft_max_layer.forward(X)
        fb_epsilon, _ = soft_max_layer.soft_max(out_epsilon, Y)
        diff[i] = abs(fb_epsilon - fb)
        diff_grad[i] = abs(fb_epsilon - fb - epsilon * d.T @ grad_b)

    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('bias Grad Test Results')
    plt.legend(("diff without grad", "diff with grad"))
    plt.show()
```



2.SGD Optimizer:

We implemented the SGD optimizer as a class which keeps a pointer to the network, and uses it to update the parameters during the step() call using the gradients.

```python
from network import NeuralNetwork


class SGD:
    def __init__(self, net: NeuralNetwork, lr=0.001):
        self.net = net
        self.lr = lr

    def step(self):
        for layer in self.net.layers:
            layer.W = layer.W - self.lr * layer.dW
            layer.b = layer.b - self.lr * layer.db
```

For the train code itself, we used the generic code we wrote for training a neural network, but created the network with the policy = “loss” parameter indicating it will only have 1 layer, the soft max regression:

```
def train_network(data_path: str, num_layers=1, batch_size: int = 32, lr: int
= 0.001, epochs: int = 60):
    data = sio.loadmat(data_path)
    X_train = data["Yt"]
    Y_train = data["Ct"]
    X_test = data["Yv"]
    Y_test = data["Cv"]
    input_size = X_train.shape[0]
    m = X_train.shape[1]
    num_of_classes = Y_train.shape[0]
    net = NeuralNetwork(input_size, num_layers, num_of_classes,
policy='loss')
    optimizer = SGD(net, lr)
    losses = np.zeros(epochs)
    validation_accuracy = np.zeros(epochs)
    training_accuracy = np.zeros(epochs)

    for epoch in range(epochs):
        net.train_mode()
        perm_indices = np.random.permutation(m)
        for j in range(0, m, batch_size):
            X_batch = X_train[:, perm_indices[j:j + batch_size]]
            Y_batch = Y_train[:, perm_indices[j:j + batch_size]]

            out = net.forward_pass(X_batch)
            loss, probabilities = net.soft_max_layer.soft_max(out, Y_batch)
            net.backward_pass()
            optimizer.step()
            losses[epoch] += loss
            training_accuracy[epoch] += get_acc(probabilities, Y_batch)

        losses[epoch] /= (m // batch_size)
        training_accuracy[epoch] /= (m // batch_size)
        validation_accuracy[epoch] = validate(net, X_test, Y_test)

        print(f"epochs = {epoch}, loss = {losses[epoch]}, validation_accuracy
= {validation_accuracy[epoch]}"
              f" train_accuracy = {training_accuracy[epoch]}")

    plt.plot(np.arange(0, epochs, 1), validation_accuracy)
    plt.xlabel("epochs")
    plt.ylabel("score")
    plt.legend("accuracy")
    plt.title(f"accuracy : batchsize = {batch_size} lr = {lr}")
    plt.show()

    plt.plot(np.arange(0, epochs, 1), losses)
    plt.xlabel("epochs")
    plt.ylabel("score")
```

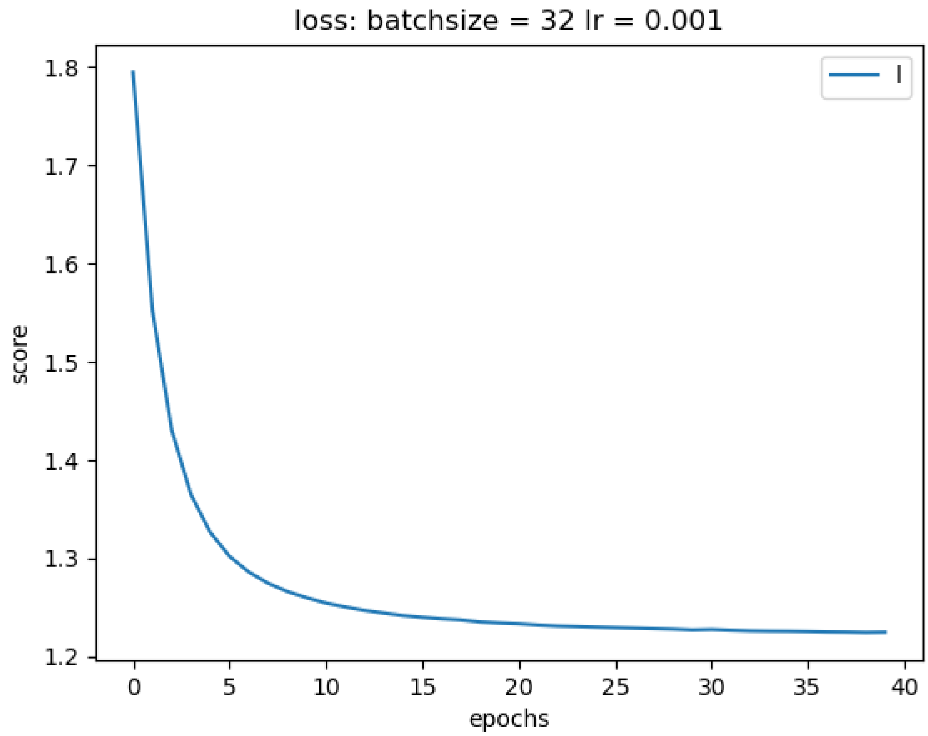
```
    plt.legend("loss")
    plt.title(f"loss: batchsize = {batch_size} lr = {lr}")
    plt.show()


def validate(net: NeuralNetwork, X_test, Y_test):
    net.eval_mode()
    out = net.forward_pass(X_test)
    _, probabilities = net.soft_max_layer.soft_max(out, Y_test)
    acc = get_acc(probabilities, Y_test)
    return acc
```

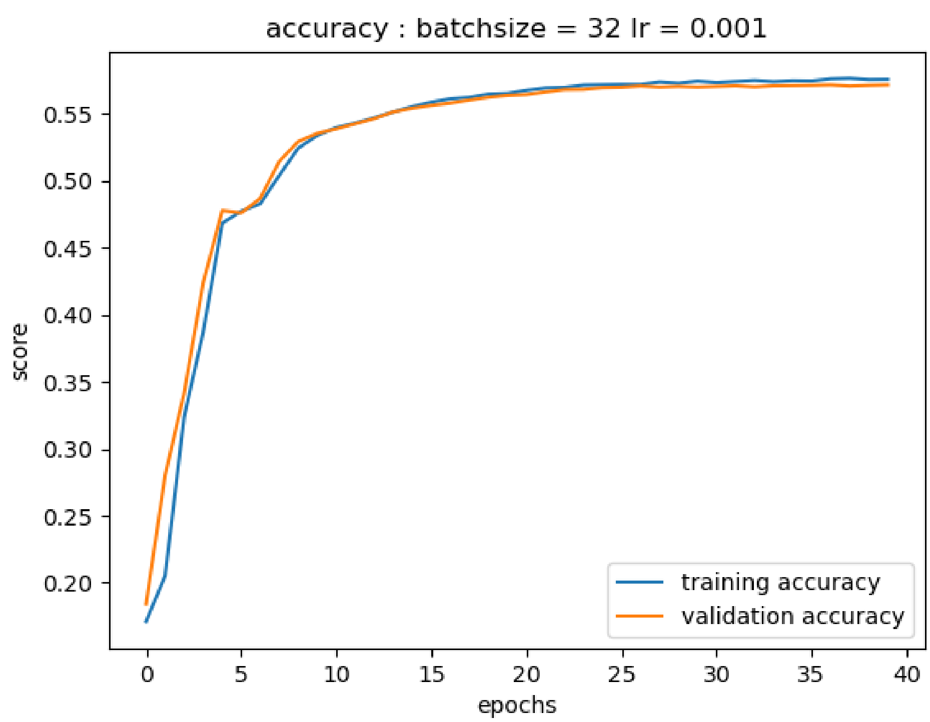
we tried any combination of the parameters : batch sizes = [32,64,128], learning rates = [0.0001, 0.001, 0.05].

the results:

Learning rate/ batch size	Loss	Train accuracy	Validate accuracy
0.0001,32	1.344	0.42	0.42
0.0001,64	1.48	0.27	0.28
0.0001,128	1.65	0.18	0.18
0.001,32	1.22	0.57	0.57
0.001,64	1.23	0.56	0.56
0.001,128	1.26	0.53	0.53
0.05,32	1.22	0.57	0.56
0.05,64	1.22	0.57	0.57
0.05,128	1.22	0.57	0.57

We chose lr = 0.001, batch size =32, as it provided us with the best results:





```
class NeuralNetwork:
    def __init__(self, input_size, num_of_layers, num_of_classes,
                 policy="constant"):

        if policy == "constant" or num_of_layers == 1
        # creating a constant sized network with num_of_layers layers
            self.layers = [Layer(input_size, input_size) for i in
            range(num_of_layers - 1)] +
            [SoftMaxLayer(input_size, num_of_classes)]

        elif policy == "loss":
            self.layers = [SoftMaxLayer(input_size, num_of_classes)]

        else:
       # creating a list of layers, where we increase in dimensions each time
         until the middle layer
       # then we start decreasing again until the final loss layer with
         output size num_of_classes
                self.layers = [Layer(input_size, 6)] +
                [Layer(2 * (i + 2), 2 * (i + 3))
                for i in range(1, (num_of_layers) // 2)] \
                + [Layer(2 * (i + 3), 2 * (i + 2))
                for i in range((num_of_layers) // 2 - 1, 0, -1)] \
                + [SoftMaxLayer(6, num_of_classes)]

        self.soft_max_layer = self.layers[-1]

    def forward_pass(self, X):
        out = X
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def backward_pass(self):
        prev_dx = None
        for layer in self.layers[::-1]:
            layer.backward(prev_dx)
            prev_dx = layer.dX.copy()

    def train_mode(self):
        for layer in self.layers:
            layer.train_mode()

    def eval_mode(self):
        for layer in self.layers:
            layer.eval_mode()




class Layer:
    def __init__(self, in_dimensions, out_dimensions, activation=ReLU):
```

```python
        """
        :param in_dimensions: dimensions of input
        :param out_dimensions: dimensions of output
        :param activation: activation function used by the layer
        """
        self.X = None
        np.random.seed(0)
        self.W = np.random.uniform(-1, 1,
                size=(out_dimensions, in_dimensions))
        self.b = np.zeros((out_dimensions, 1))
        self.activation = activation.activate
        self.activation_derivative = activation.deriviative
        self.dX = None  # derivative with respect to input
        self.dW = None  # derivative with respect to Weight
        self.db = None  # derivative with respect to bias
        self.train = True

    def forward(self, X):
        self.X = X.copy()
        out = self.activation(self.W @ X + self.b)
        return out

    def backward(self, V):
        temp = self.activation_derivative(self.W @ self.X + self.b) * V
        self.dX = self.W.T @ temp
        self.dW = temp @ self.X.T
        self.db = np.sum(temp, axis=1).reshape(-1, 1)

    def train_mode(self):
        self.train = True

    def eval_mode(self):
        self.train = False



class SoftMaxLayer(Layer):

        :param Y: a matrix of size lxm,
         where Y[i,:] is c_i (indicator vector for label i)
        :return: loss score, and probabilities matrix for each class,x
        """
(Included Above In Part 1)
```

Activation functions:

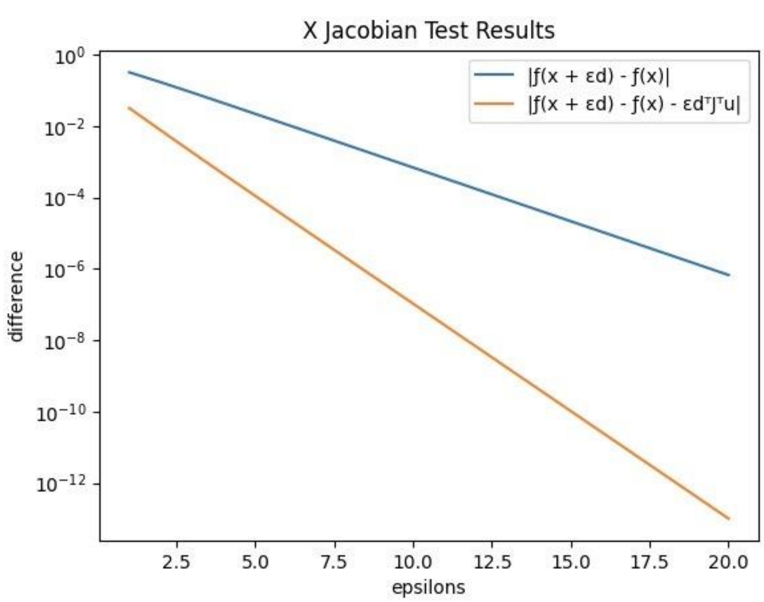
```
class ReLU:
    @staticmethod
    def activate(x):
        return np.maximum(0, x)

    @staticmethod
    def deriviative(x):
        f = lambda t: 1 if t >= 0 else 0
        vfunc = np.vectorize(f)

        return vfunc(x)

class tanh:
    @staticmethod
    def activate(x):
        return np.tanh(x)

    @staticmethod
    def deriviative(x):
        f = lambda X: 1 - np.tanh(X) ** 2
        return f(x)
```

Jacobian Tests (conducted with tanh)

we used the shortcut version, defining a function $g(x) = < f(x), u >$ and performing a grad test for it, as $\nabla g(x) = J^T u$.

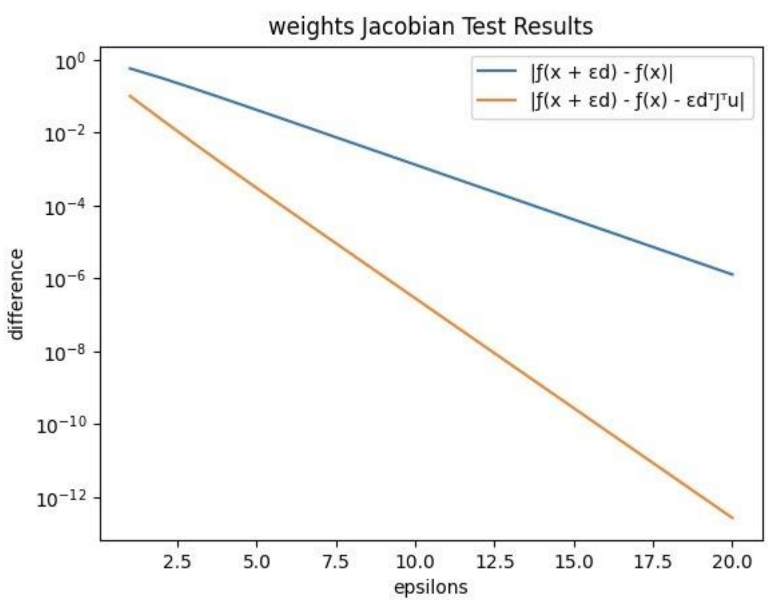
Input(X):

```
def jacobian_test_layer_X(X):
    layer = Layer(2, 3)
    n, m = X.shape
    out_dimensions = layer.b.shape[0]
    U = normalize(np.random.rand(out_dimensions, m))

    iter_num = 20
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    d = normalize(np.random.rand(*X.shape))

    fx = np.dot(layer.forward(X).T, U).item()
    layer.backward(U)
    JacTu_X = layer.dX

    for i, epsilon in enumerate(epsilons):
        X_diff = X.copy()
        X_diff += d * epsilon
        fx_epsilon = np.dot(layer.forward(X_diff).T, U).item()
        d_flat = d.reshape(-1, 1)
        JacTu_X_flat = JacTu_X.reshape(-1, 1)

        diff[i] = abs(fx_epsilon - fx)
        diff_grad[i] = abs(fx_epsilon - fx - epsilon * d_flat.T @
JacTu_X_flat)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('X Jacobian Test Results')
    plt.legend(("diff without grad", "diff with grad"))
```



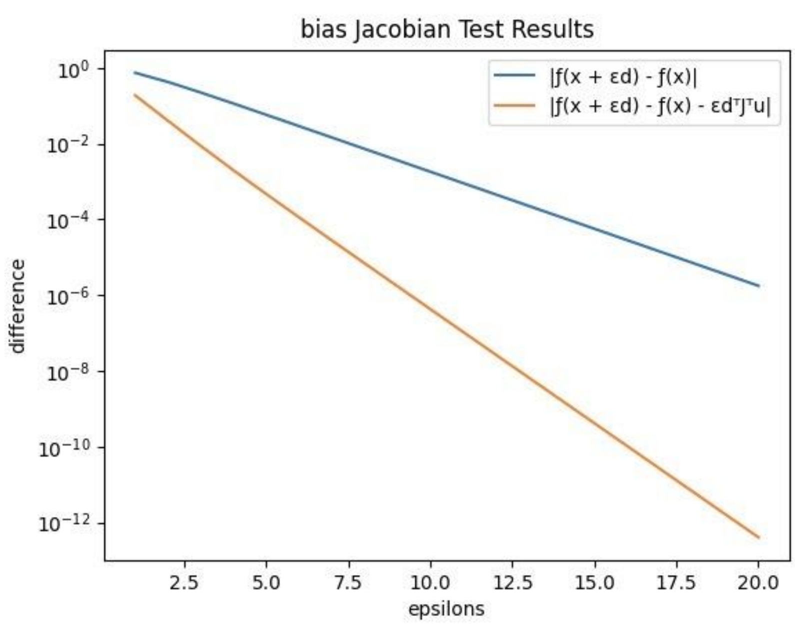
Weights:

```
def jacobian_test_layer_W(X):
    layer = Layer(2, 3)
    n, m = X.shape
    out_dimensions = layer.b.shape[0]
    U = normalize(np.random.rand(out_dimensions, m))
    original_W = layer.W.copy()

    iter_num = 20
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    d = normalize(np.random.rand(*layer.W.shape))
    fw = np.dot(layer.forward(X).T, U).item()
    layer.backward(U)
    JacTu_W = layer.dW

    for i, epsilon in enumerate(epsilons):
        W_diff = original_W.copy()
        W_diff += d * epsilon
        layer.W = W_diff
        fw_epsilon = np.dot(layer.forward(X).T, U).item()
        diff[i] = abs(fw_epsilon - fw)
        d_flat = d.reshape(-1, 1)
        JacTu_W_flat = JacTu_W.reshape(-1, 1)
        diff_grad[i] = abs(fw_epsilon - fw - epsilon * d_flat.T @
JacTu_W_flat)

    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('weights Jacobian Test Results')
    plt.legend(("diff without grad", "diff with grad"))
    plt.show()
```



Bias:

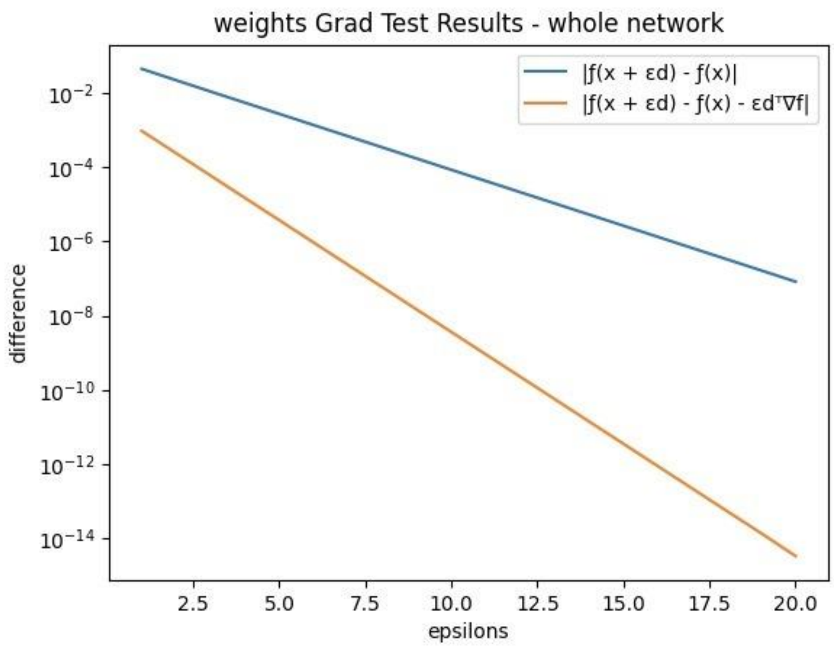
```
def jacobian_test_layer_b(X):
    layer = Layer(2, 3)
    n, m = X.shape
    out_dimensions = layer.b.shape[0]
    U = normalize(np.random.rand(out_dimensions, m))
    original_b = layer.b.copy()

    iter_num = 20
    diff = np.zeros(iter_num)
    diff_grad = np.zeros(iter_num)
    epsilons = [0.5 ** i for i in range(iter_num)]
    d = normalize(np.random.rand(*layer.b.shape))
    fb = np.dot(layer.forward(X).T, U).item()
    layer.backward(U)
    JacTu_b = layer.db
    for i, epsilon in enumerate(epsilons):
        b_diff = original_b.copy()
        b_diff += d * epsilon
        layer.b = b_diff
        fb_epsilon = np.dot(layer.forward(X).T, U).item()
        diff[i] = abs(fb_epsilon - fb)
        diff_grad[i] = abs(fb_epsilon - fb - epsilon * d.T @ JacTu_b)

    plt.semilogy(np.arange(1, iter_num + 1, 1), diff)
    plt.semilogy(np.arange(1, iter_num + 1, 1), diff_grad)
    plt.xlabel('epsilons')
    plt.ylabel('difference')
    plt.title('bias Jacobian Test Results')
    plt.legend(("diff without grad", "diff with grad"))
    plt.show()
```



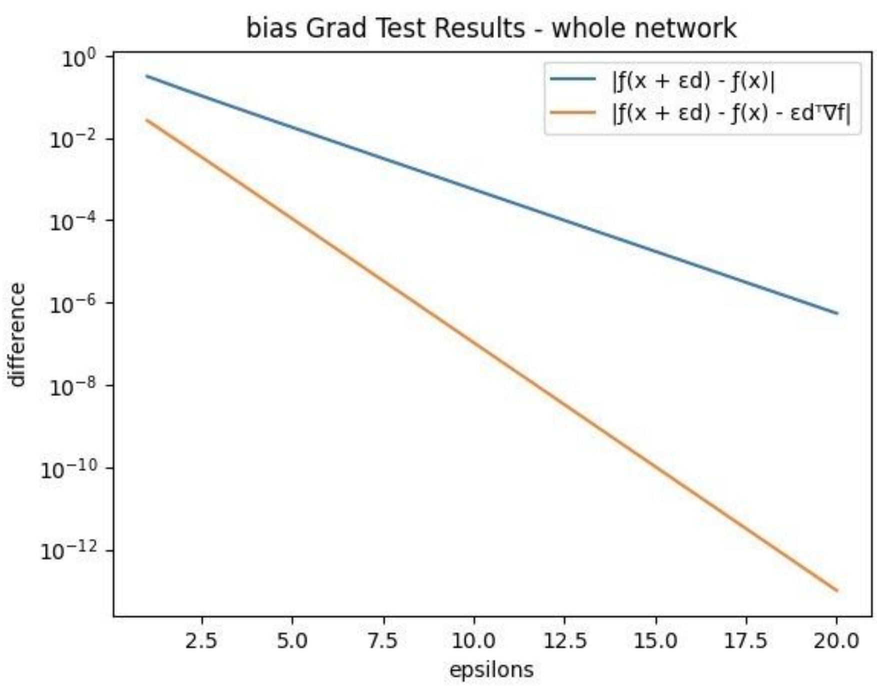
2.2.3 check if should add more output

<u>Grad Tests for the whole network:</u>

Weights:



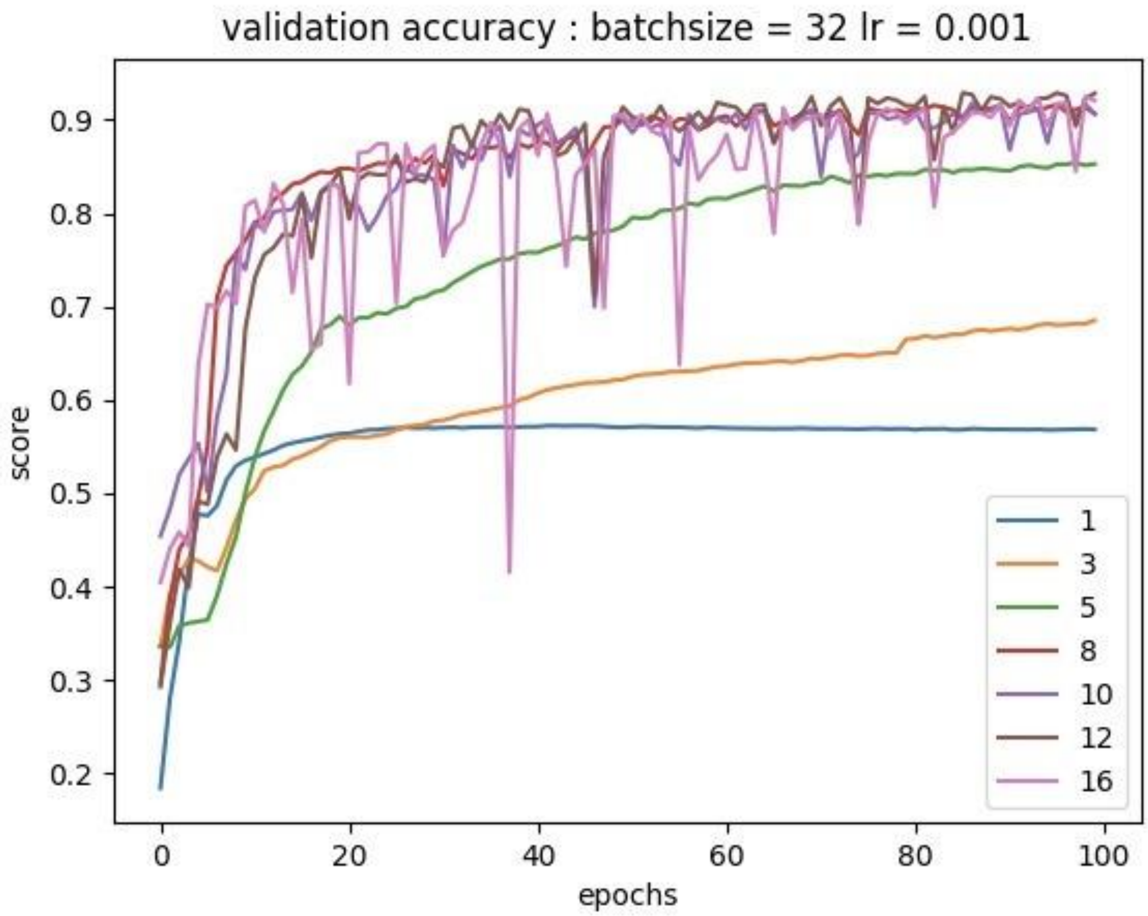
Bias:

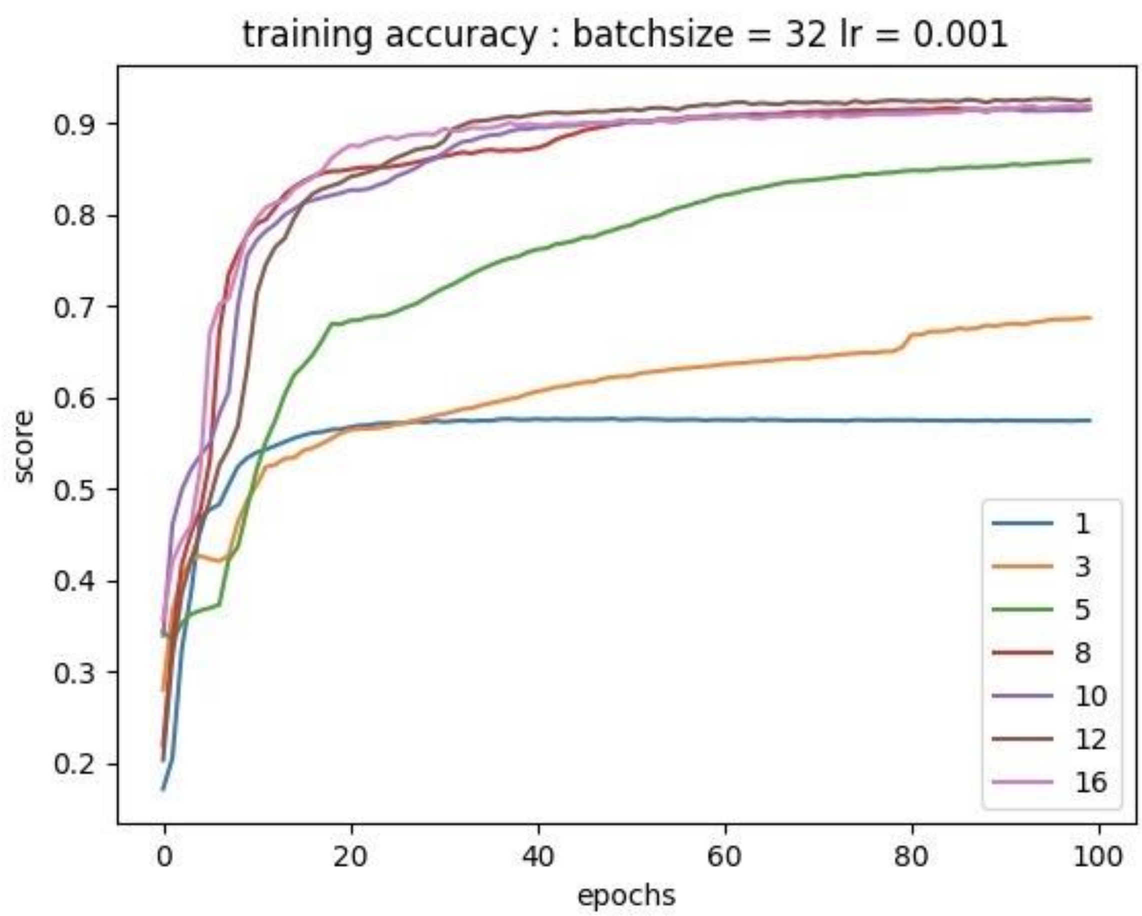


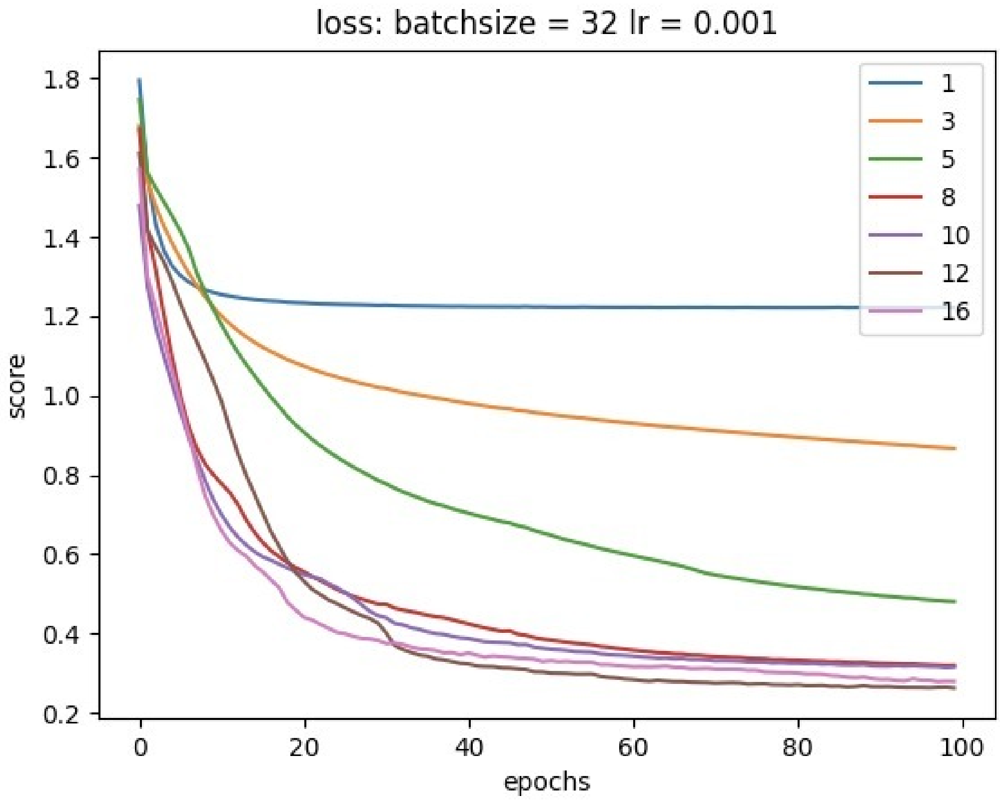
3.Expiriments:

We experimented with a few layer sizes ranging between 1-16, where in the first l/2 layers the dimensions increase by 2 in each forward pass, then in the last l/2 layers we decrease back by 2 each pass.

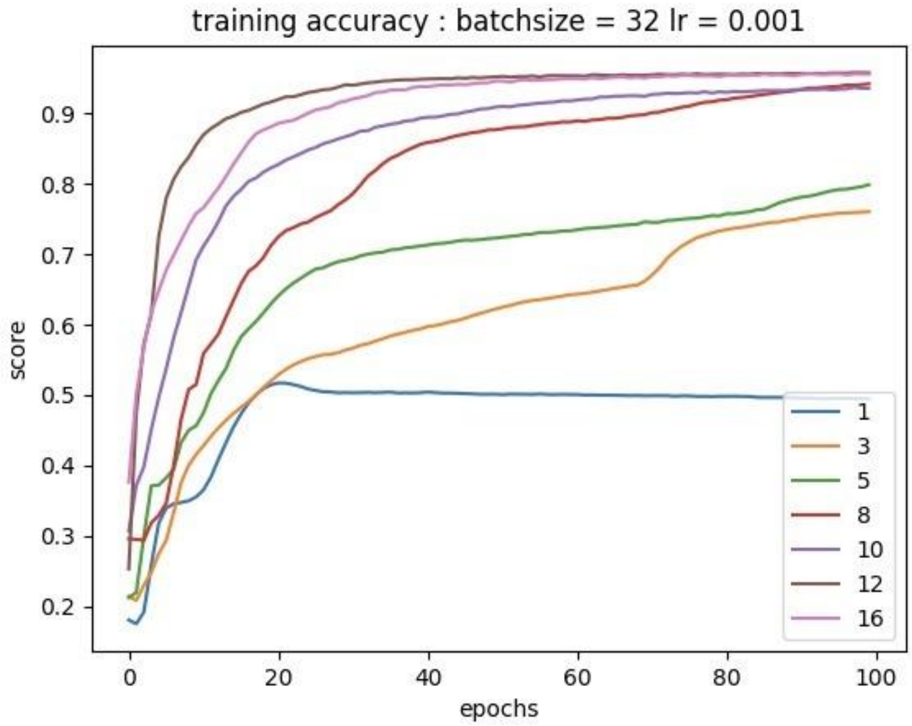
Peaks Data

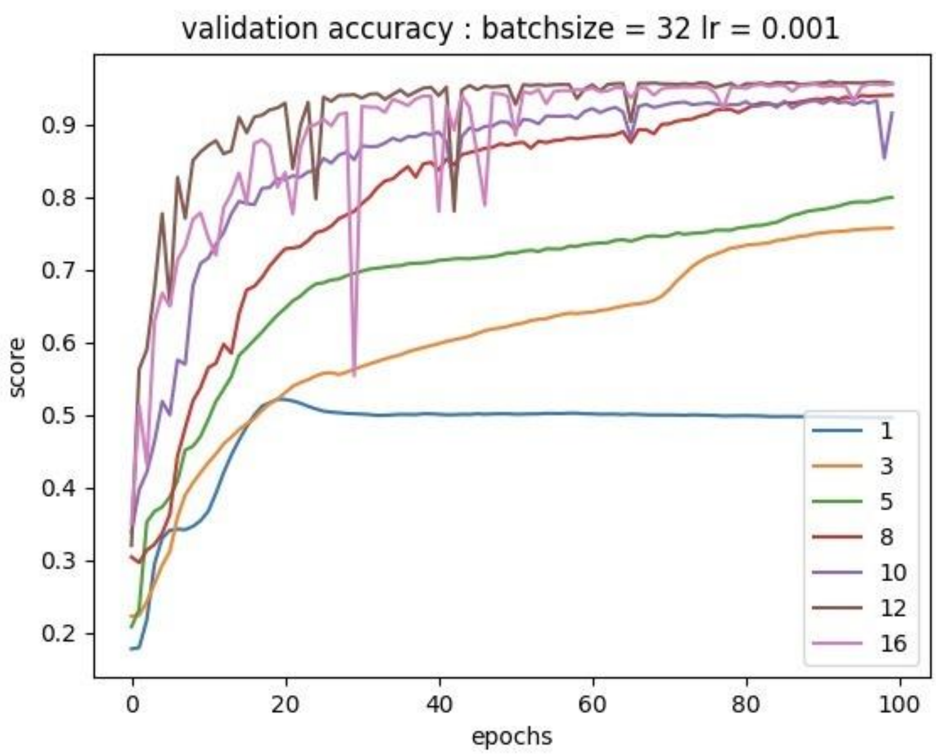

validation accuracy : batchsize = 32 lr = 0.001
score
epochs
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
20
40
60
80
100
1
3
5
8
10
12
16

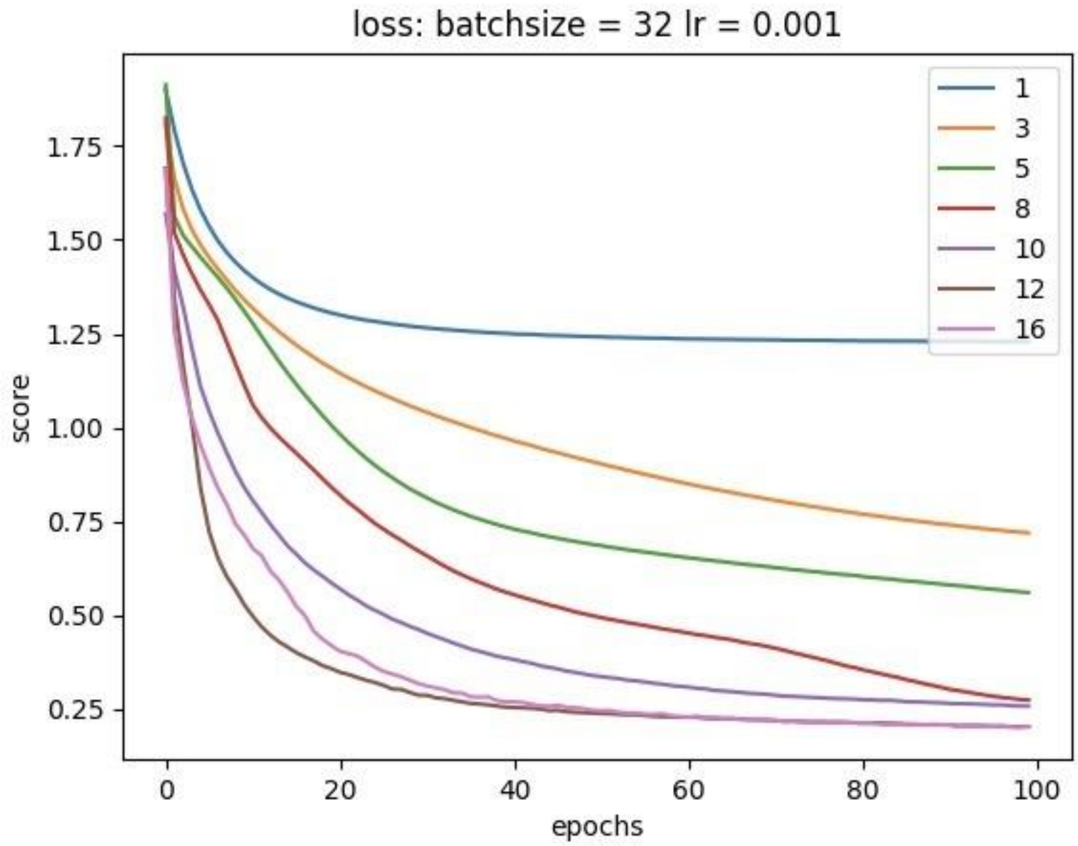

training accuracy : batchsize = 32 lr = 0.001
score
epochs
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
20
40
60
80
100
1
3
5
8
10
12
16


loss: batchsize = 32 lr = 0.001
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
score
0
20
40
60
80
100
epochs
1
3
5
8
10
12
16

GMM Data


training accuracy : batchsize = 32 lr = 0.001
score
epochs
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100
1
3
5
8
10
12
16


validation accuracy : batchsize = 32 lr = 0.001
score
epochs
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100
1
3
5
8
10
12
16



loss: batchsize = 32 lr = 0.001
1
3
5
8
10
12
16
score
epochs
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
20
40
60
80
100

Conclusions:

As we can see from the graphs above, in both the data sets of GMM and Peaks, the best results were achieved using a network with 12 layers (

In GMM: best epoch achieved 95.8% train accuracy, 95.9% validation accuracy, and 0.2 loss score.
In Peaks: best epoch achieved 92.6% train accuracy, 92.9% validation accuracy, and 0.26 loss score.

Between 1 - 12 layers we can the results (both accuracy and loss) improve as the layer number increase, and in the 16 layers version we witness a decrease in performance.

Also, as we increase in the layers number (especially with 12 and 16) we can see that the validation accuracy becomes less stable, with substantial changes between epochs.