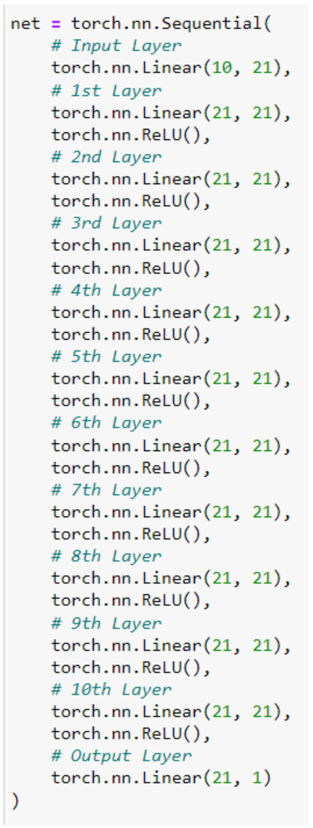
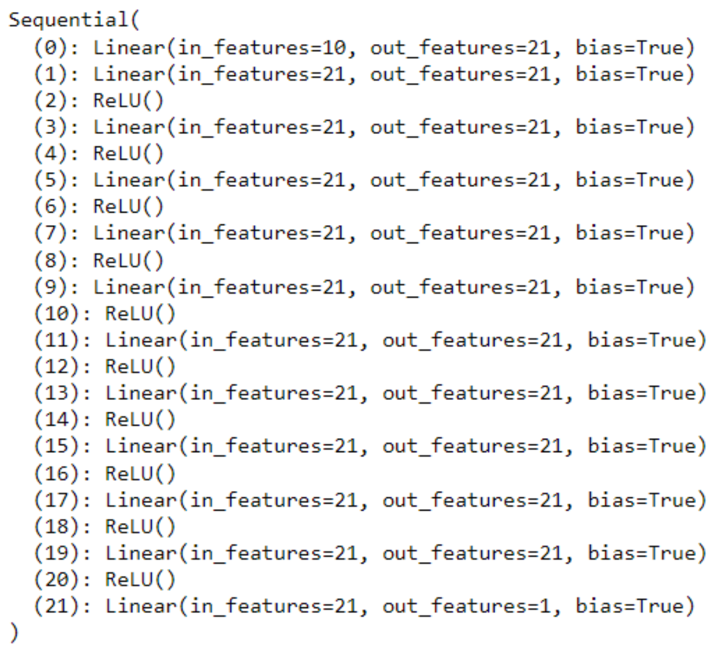
BS 6207

Q1:

Following the question, I first calculate the node of the hidden layer is 21; I create a sequential network with one input layer, ten hidden layers with Relu Activativation Function, and one output layer.

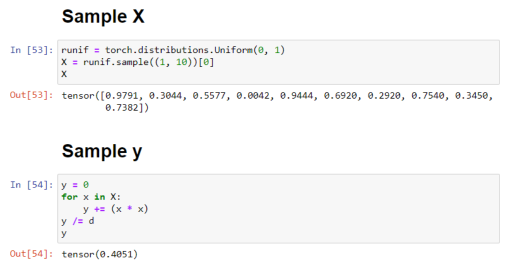


```
net = torch.nn.Sequential(
    # Input Layer
    torch.nn.Linear(10, 21),
    # 1st Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 2nd Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 3rd Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 4th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 5th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 6th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 7th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 8th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 9th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # 10th Layer
    torch.nn.Linear(21, 21),
    torch.nn.ReLU(),
    # Output Layer
    torch.nn.Linear(21, 1)
)
```



```
Sequential(
  (0): Linear(in_features=10, out_features=21, bias=True)
  (1): Linear(in_features=21, out_features=21, bias=True)
  (2): ReLU()
  (3): Linear(in_features=21, out_features=21, bias=True)
  (4): ReLU()
  (5): Linear(in_features=21, out_features=21, bias=True)
  (6): ReLU()
  (7): Linear(in_features=21, out_features=21, bias=True)
  (8): ReLU()
  (9): Linear(in_features=21, out_features=21, bias=True)
  (10): ReLU()
  (11): Linear(in_features=21, out_features=21, bias=True)
  (12): ReLU()
  (13): Linear(in_features=21, out_features=21, bias=True)
  (14): ReLU()
  (15): Linear(in_features=21, out_features=21, bias=True)
  (16): ReLU()
  (17): Linear(in_features=21, out_features=21, bias=True)
  (18): ReLU()
  (19): Linear(in_features=21, out_features=21, bias=True)
  (20): ReLU()
  (21): Linear(in_features=21, out_features=1, bias=True)
)
```

After that, I used Pytorch's build-in uniform distribution to generate X and y.



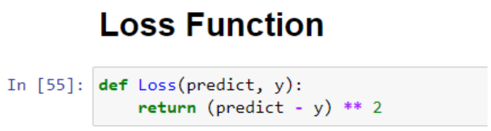
Sample X

```
In [53]: runif = torch.distributions.Uniform(0, 1)
         X = runif.sample((1, 10))[0]
         X

Out[53]: tensor([0.9791, 0.3044, 0.5577, 0.0042, 0.9444, 0.6920, 0.2920, 0.7540, 0.3450,
                 0.7382])
```

Sample y

```
In [54]: y = 0
         for x in X:
             y += (x * x)
         y /= d
         y

Out[54]: tensor(0.4051)
```

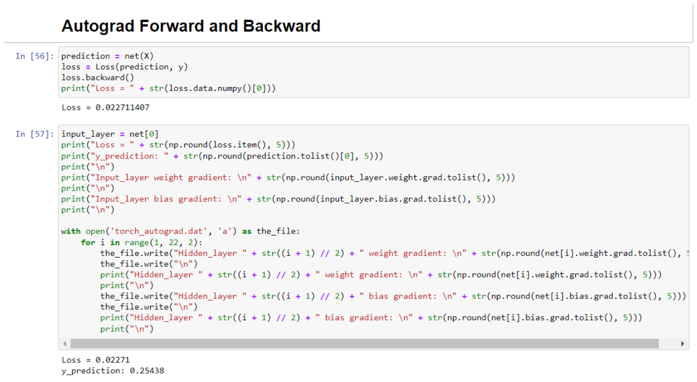
Then I implant the loss function as described.



Loss Function

```
In [55]: def Loss(predict, y):
             return (predict - y) ** 2
```

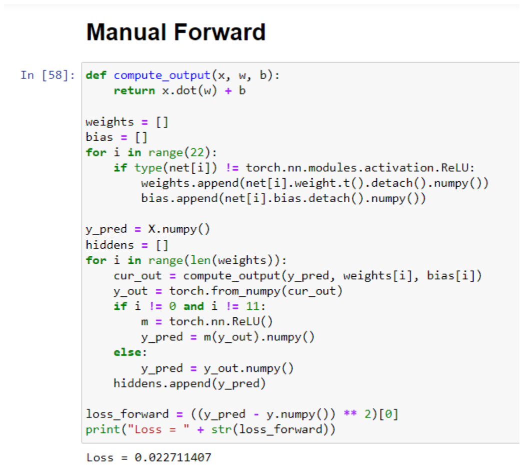
After that, I used the sequential network for prediction and stored each layer's weight and bias gradient into the text file.



Autograd Forward and Backward

```
In [56]: prediction = net(X)
         loss = Loss(prediction, y)
         loss.backward()
         print("Loss = " + str(loss.data.numpy()[0]))

         Loss = 0.022711407

In [57]: input_layer = net[0]
         print("Loss = " + str(np.round(loss.item(), 5)))
         print("y_prediction: " + str(np.round(prediction.tolist()[0], 5)))
         print("\n")
         print("Input_layer weight gradient: \n" + str(np.round(input_layer.weight.grad.tolist(), 5)))
         print("\n")
         print("Input_layer bias gradient: \n" + str(np.round(input_layer.bias.grad.tolist(), 5)))
         print("\n")

         with open('torch_autograd.dat', 'a') as the_file:
             for i in range(1, 22, 2):
                 the_file.write("Hidden_layer " + str((i + 1) // 2) + " weight gradient: \n" + str(np.round(net[i].weight.grad.tolist(),
                 the_file.write("\n")
                 print("Hidden_layer " + str((i + 1) // 2) + " weight gradient: \n" + str(np.round(net[i].weight.grad.tolist(), 5)))
                 print("\n")
                 the_file.write("Hidden_layer " + str((i + 1) // 2) + " bias gradient: \n" + str(np.round(net[i].bias.grad.tolist(), 5)))
                 the_file.write("\n")
                 print("Hidden_layer " + str((i + 1) // 2) + " bias gradient: \n" + str(np.round(net[i].bias.grad.tolist(), 5)))
                 print("\n")

         Loss = 0.02271
         y_prediction: 0.25438
```

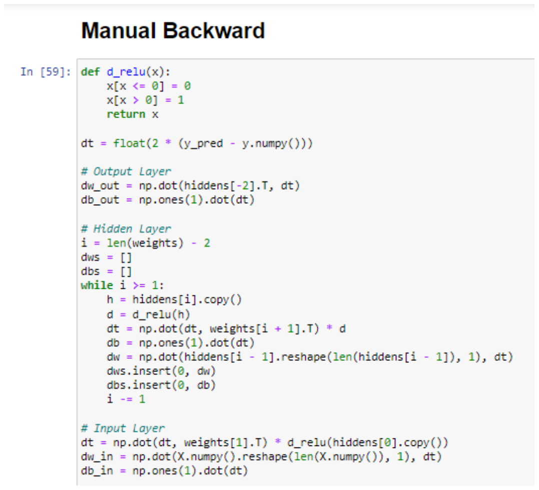
For forwarding propagation, I applied a loop on the initial X and used saved weight and bias gradient on X by the formula $W^T X + b$ to compute the X_i at each layer.



Manual Forward

```
In [58]: def compute_output(x, w, b):
             return x.dot(w) + b

         weights = []
         bias = []
         for i in range(22):
             if type(net[i]) != torch.nn.modules.activation.ReLU:
                 weights.append(net[i].weight.t().detach().numpy())
                 bias.append(net[i].bias.detach().numpy())

         y_pred = X.numpy()
         hiddens = []
         for i in range(len(weights)):
             cur_out = compute_output(y_pred, weights[i], bias[i])
             y_out = torch.from_numpy(cur_out)
             if i != 0 and i != 11:
                 m = torch.nn.ReLU()
                 y_pred = m(y_out).numpy()
             else:
                 y_pred = y_out.numpy()
             hiddens.append(y_pred)

         loss_forward = ((y_pred - y.numpy()) ** 2)[0]
         print("Loss = " + str(loss_forward))

         Loss = 0.022711407
```

For backward propagation, I loop through the layers reversely, use the prediction as input and get the gradient of weight and bias at each layer by using the formula from the slides with Relu’s derivative.

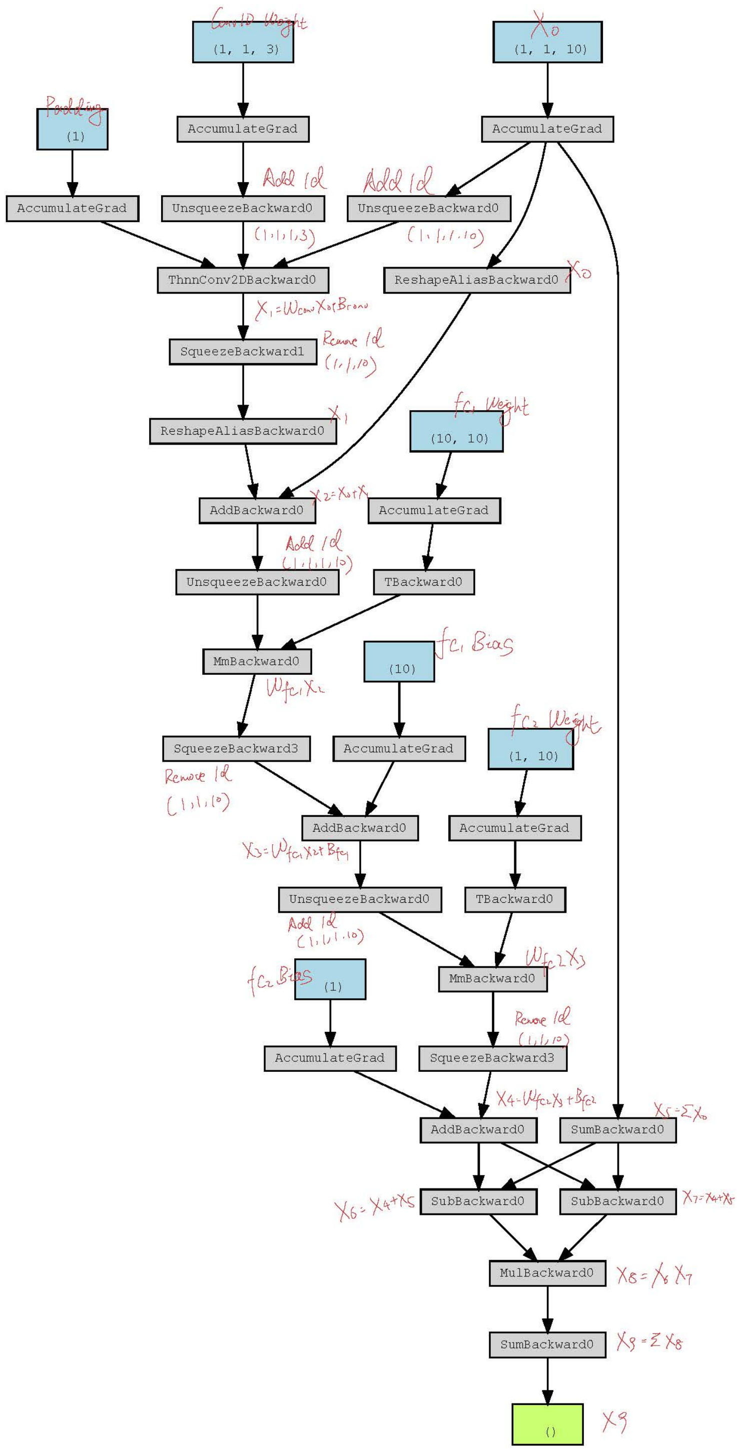


Manual Backward

```
In [59]: def d_relu(x):
             x[x <= 0] = 0
             x[x > 0] = 1
             return x

         dt = float(2 * (y_pred - y.numpy()))

         # Output Layer
         dw_out = np.dot(hiddens[-2].T, dt)
         db_out = np.ones(1).dot(dt)

         # Hidden Layer
         i = len(weights) - 2
         dws = []
         dbs = []
         while i >= 1:
             h = hiddens[i].copy()
             d = d_relu(h)
             dt = np.dot(dt, weights[i + 1].T) * d
             db = np.ones(1).dot(dt)
             dw = np.dot(hiddens[i - 1].reshape(len(hiddens[i - 1]), 1), dt)
             dws.insert(0, dw)
             dbs.insert(0, db)
             i -= 1

         # Input Layer
         dt = np.dot(dt, weights[1].T) * d_relu(hiddens[0].copy())
         dw_in = np.dot(X.numpy().reshape(len(X.numpy()), 1), dt)
         db_in = np.ones(1).dot(dt)
```

Lastly, I compare the result, which is the same as auto-graded.

Q2:



The definitions of individual nodes are labelled in the graph. Some general nodes include AccumulateGrad, TBackWard, UnsqueezeBackward, SqueezeBackward, and ReshapeAliasBackward represent:

- AccumulateGrad: ask each node to save gradients accumulated into their .grad fields.
- TBackWard: Trace Backward Propagation to save weight, bias and function gradient.
- ReshapeAliasBackward: Reshape the node to a specific dimension.
- UnsqueezeBackward: Add one dimension of the previous node.
- SqueezeBackward: Remove one dimension of the previous node.

Github Link:

https://github.com/Ironarrow98/Zhang_Chenxi_BS6207_A1