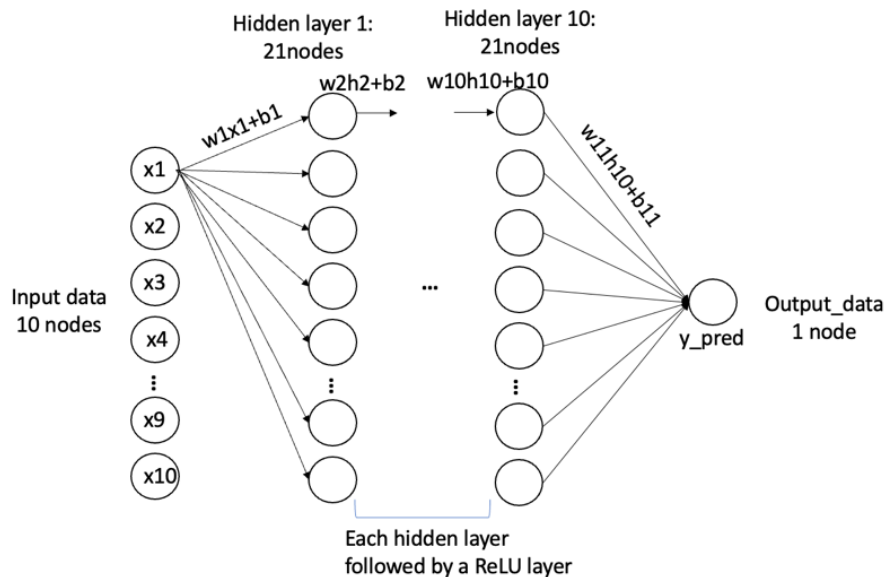


Question #1

1. Implement this neural network in pytorch

Overview of the neural network



2. Generate the input data (x_1, x_2, \dots, x_d) in $[0, 1]$ drawn from a uniform random distribution.

Using uniform distribution to generate random value with interval $[0, 1]$.

```
runif = torch.distributions.Uniform(0,1) #[0,1]
x=runif.sample((batch_size,input_data))
```

3. Generate the labels $y = (x_1*x_1+x_2*x_2+\dots+x_d*x_d)/d$

```
y = x.pow(2).sum()/2
```

4. Implement a loss function $L = (\text{predict}-y)^2$

```
#Loss Function
loss = (y_auto_pred-y).pow(2).sum()
```

```
batch_size = 1
```

```
x=runif.sample((batch_size,input_data))
```

5. Use batch size of 1, that means feed data one point at a time into network and compute the loss. Do one-time forward propagation with one data point.
6. Compute the gradients using pytorch autograd:
 - a. $dL/dw, dL/db$
 - b. Print these values into a text file: torch_autograd.dat
7. Implement the forward propagation and backpropagation algorithm from scratch, without using pytorch autograd, compute the gradients using your implementation.

a. dL/dw , dL/db

b. Print these values into a text file: my_autograd.dat

Backpropagation algorithm:

```
loss = (y_pred-y)**2/2
grad_y_pred = loss' = 2*(y_pred - y)
y_pred = relu( h10 * w11 + b11)
b11' = relu'(x) * x' = 1*1* grad_y_pred = grad_b11
w11' = 1*1*h10* grad_y_pred
w11' = h10.T* grad_b11 = grad_w11
y_pred = relu( relu(h9*w10 + b10) * w11 + b11)
b10' = w11.T* grad_b11
w10' = grad_w11 * h9.T
```

```
#Back-propagation
grad_y_pred = 2*(y_pred - y)
w_grad = [0]*11
b_grad = [0]*11

for i in range(11):
    if i == 0:
        b_grad[10 - i] = grad_y_pred
        w_grad[10 - i] = (b_grad[10-i] * h_list[9-i].T )
    elif i == 10:
        b_grad[10 - i] = torch.mm(b_grad[11-i],w_list[11 - i].T)
        b_grad[10 - i][h_list[10 - i]<=0] = 0
        w_grad[10 - i] = torch.mm(x.T,b_grad[10-i])
    else:
        b_grad[10 - i] = torch.mm(b_grad[11-i],w_list[11 - i].T)
        b_grad[10 - i][h_list[10 - i]<=0] = 0
        w_grad[10 - i] = torch.mm(h_list[9-i].T,b_grad[10-i])
```

8. Compare the two files torch_autograd.dat and my_autograd.dat and show that they give the same values up to 5 significant numbers

I exported w and b in different files, so I got four files.

My_grad_w vs. torch_grad_w

my_grad_w.txt	torch_grad_w.txt
tensor([[-5.53375e+08, 0.00000e+00, 7.07888e+08, 1.68588e+08, -4.91744e+08, 0.00000e+00, 0.00000e+00, 0.00000e+00, 4.88407e+08, 3.73732e+05, 1.09556e+09, 0.00000e+00, 0.00000e+00, -1.30155e+08, 1.18425e+09, 0.00000e+00, -2.17313e+08, -3.46116e+08, 0.00000e+00, 0.00000e+00, 6.44276e+06],	tensor([[-5.53375e+08, 0.00000e+00, 7.07888e+08, 1.68588e+08, -4.91744e+08, 0.00000e+00, 0.00000e+00, 0.00000e+00, 4.88407e+08, 3.73732e+05, 1.09556e+09, 0.00000e+00, 0.00000e+00, -1.30155e+08, 1.18425e+09, 0.00000e+00, -2.17313e+08, -3.46116e+08, 0.00000e+00, 0.00000e+00, 6.44276e+06],

y_grad_b vs. torch_grad_b

torch_grad_b.txt	my_grad_b.txt
tensor([[-9.28489e+08, 0.00000e+00, 1.18774e+09, 2.82868e+08, -8.25080e+08, 0.00000e+00, 0.00000e+00, 0.00000e+00, 8.19480e+08, 6.27072e+05, 1.83820e+09, 0.00000e+00, 0.00000e+00, -2.18382e+08, 1.98701e+09, 0.00000e+00, -3.64622e+08, -5.80736e+08, 0.00000e+00, 0.00000e+00, 1.08101e+07]])	tensor([[-9.28489e+08, 0.00000e+00, 1.18774e+09, 2.82868e+08, -8.25080e+08, 0.00000e+00, 0.00000e+00, 0.00000e+00, 8.19480e+08, 6.27072e+05, 1.83820e+09, 0.00000e+00, 0.00000e+00, -2.18382e+08, 1.98701e+09, 0.00000e+00, -3.64622e+08, -5.80736e+08, 0.00000e+00, 0.00000e+00, 1.08101e+07]])

These files show the result of my grad and torch grad are all same. It means that the implementation of my algorithm is correct.

9. Use $K=10, d=10$

Question #2

Run the following code, generate the computational graph, label and explain **all** nodes (all nodes mean not just the leave nodes, all intermediate nodes should be explained):

This graph shows a tree that implementing pytorch operation. It builds during **forward propagation** and showing which operations will be called on **backward**. It didn't mention the subgraph which **do not require gradient**.

Blue boxes:

These correspond to the tensors we use as parameters, the ones we want use PyTorch to compute gradients.

Gray boxes:

A Python operation that involves a gradient-computing tensor or its dependencies.

Green box:

It is the starting point for the computation of gradients (assuming the backward () method is called from the variable used to visualize the graph) — they are computed from the **bottom-up** in a graph. (<https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e>)

The attached pdf shows the label of all nodes.