**Question #1**

Given a fully connected Neural Network as follows:

1. Input (x1,x2,…,xd): d-nodes
2. K-hidden fully connected layers with bias of 2d+1 nodes
3. Output (predict): 1 node
4. Use Relu activation function for all layers
5. Implement this neural network in pytorch

# https://pytorch.org/tutorials/recipes/recipes/defining\_a\_neural\_network.html

# https://www.cnblogs.com/denny402/p/7593301.html

import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.layers = nn.Sequential()

# define input layer

self.layers.add\_module("input", nn.Linear(10, 21, bias = True))

self.layers.add\_module("hidden0\_relu", torch.nn.ReLU())

# define fc layers

for i in range(1, 10):

self.layers.add\_module("hidden"+str(i)+"\_fc", nn.Linear(21, 21, bias = True))

self.layers.add\_module("hidden"+str(i)+"\_relu",torch.nn.ReLU())

# output layer

self.layers.add\_module("output", nn.Linear(21, 1, bias = True))

def forward(x):

self.layers(x)

model = Net()

print(model)

1. Generate the input data (x1,x2,..xd) \in [0,1] drawn from a uniform random distribution

X = (0 - 1) \* torch.rand(10) + 1

1. Generate the labels y = (x1\*x1+x2\*x2+…+xd\*xd)/d

y = X.pow(2).sum()/2

1. Implement a loss function L = (predict-y)^2

def Loss(predict, y): return (predict - y).pow(2).sum()

1. 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.

# https://stackoverflow.com/questions/52241680/pytorch-notimplementederror-in-forward

optimizer = torch.optim.SGD(model.layers.parameters(), lr=0.1)

optimizer.zero\_grad()

y\_pred = model.layers(X)

loss = Loss(y\_pred, y)

loss.backward()

1. Compute the gradients using pytorch autograd: a. dL/dw, dL/db b. Print these values into a text file: torch\_autograd.dat

with open("torch.autograd.dat", "w") as f:

f.write("Loss = " + str(np.round(loss.item(),5))+"\n")

f.write("y\_pred: \n" + str(np.round(y\_pred.tolist(),5))+"\n")

for i in model.layers:

if "Linear" in str(i):

f.write("\n"+str(i)+"\n")

f.write("w\_gradient: \n" + str(np.round(i.weight.grad.tolist(),5))+"\n")

f.write("b\_gradient: \n" + str(np.round(i.bias.grad.tolist(),5))+"\n")

1. 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

Please see GitHub repo for detail. To help implement the computation, we can also generate a computational tree as suggested in Question 2.



…….



1. Compare the two files torch\_autograd.dat and my\_autograd.dat and show that they give the same values up to 5 significant numbers.

They are the same

1. Use K=10,d=10

**Question #2**

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

The graph is a computational graph of the model defined in the code, which shows how the reverse mode of automatic differentiation works in the model.[[1]](#footnote-1) So we can tell from the output how this calculate to the final gradients are done. The annotations in the graph below shows the forward mode that helps us understand the model architecture.

The model starts with the a random tensor with the shape (1, 1, 10) with autograd, which we can call x0, as shown in the up left corner of the graph. Then the tensor is put into the convolutional layer (Conv Layer) to get c=conv(x0) as the output. The output c is later been added up with flatten(x0) to generate x1. This is made the input of the first fully connected layer (FC1 Layer), where the output is designated as x2 and likewise to the second fully connected layer (FC2 Layer). Eventually the model ends up with the green box representing the loss function.

Each grey box here means the backward propagation methods, i.e. how to differentiate the corresponding forward method. Thus, the documentation of the each forward step can be found in PyTorch documentation. For example, for the box annotated with UnsqueezeBackward0, its forward function is torch.unsqueeze, of which the documentation can be found within the official PyTorch documentation.[[2]](#footnote-2) Thus, we can know what each forward step is and then make annotations as follows.

Diagram

Description automatically generated

1. <https://pytorch.org/blog/overview-of-pytorch-autograd-engine/> [↑](#footnote-ref-1)
2. <https://pytorch.org/docs/stable/generated/torch.unsqueeze.html> [↑](#footnote-ref-2)