NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

BS6207 ASSIGNMENT 1

LONG JINGYU
10/12/2021

QUESTION 1

GIVEN A FULLY CONNECTED NEURAL NETWORK AS FOLLOWS:

A. INPUT (X1,X2,...,XD): D–NODES
B. K–HIDDEN FULLY CONNECTED LAYERS WITH BIAS OF 2D+1 NODES
C. OUTPUT (PREDICT): 1 NODE
D. USE RELU ACTIVATION FUNCTION FOR ALL LAYERS

In [1]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    # a fully connected Neural Network
    def __init__(self,K=10,d=10):
        super(Net,self).__init__()
        hidden_size = 2*d+1
        # Input (x1,x2,…,xd): d-nodes
        self.input = nn.Sequential(
            nn.Linear(d,hidden_size),
            nn.ReLU()
        )
        # K-hidden fully connected layers with bias of 2d+1 nodes
        hidden_list = []
        for i in range(K):
            hidden_list.append(nn.Linear(hidden_size,hidden_size,bias=True))
            hidden_list.append(nn.ReLU())
        self.hidden = nn.Sequential(*hidden_list)
        # Output (predict): 1 node
        self.output = nn.Sequential(
            nn.Linear(hidden_size,1),
        )

    def forward(self,x):
        x = self.input(x)
        x = self.hidden(x)
        x = self.output(x)
        return x

model = Net()
model
```

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

In [2]:

```
# 2.Generate the input data (x1,x2,..xd) \in [0,1] drawn from a uniform random distr
n_sample=20
d=10
x=torch.Tensor(n_sample,d).uniform_(0,1)
x
```

```
tensor([
[0.2659, 0.1186, 0.0292, 0.8546, 0.1721, 0.2413, 0.5650, 0.2015, 0.5293,
         0.0882],
[0.3239, 0.5718, 0.2037, 0.6626, 0.3914, 0.9792, 0.1123, 0.4600, 0.7393,
         0.8573],
[0.2705, 0.9003, 0.0257, 0.3053, 0.4876, 0.3443, 0.7383, 0.6315, 0.1882,
         0.4509],
[0.9394, 0.3523, 0.4443, 0.6567, 0.9696, 0.5028, 0.8928, 0.8165, 0.3917,
         0.3599],
[0.9202, 0.0873, 0.9021, 0.8899, 0.3689, 0.9817, 0.9215, 0.9948, 0.0733,
         0.8841],
[0.9664, 0.0805, 0.7113, 0.2430, 0.9132, 0.5585, 0.2394, 0.4683, 0.8358,
         0.3403],
[0.8715, 0.4915, 0.0427, 0.9939, 0.9656, 0.1677, 0.0042, 0.5592, 0.9243,
         0.4736],
[0.2664, 0.0666, 0.8331, 0.9408, 0.7673, 0.0162, 0.2848, 0.8458, 0.2953,
         0.9762],
```

```
[0.6480, 0.6877, 0.7188, 0.3692, 0.6990, 0.4124, 0.5635, 0.9833, 0.3476,
         0.4854],
[0.3594, 0.9829, 0.9270, 0.5396, 0.3106, 0.8589, 0.9491, 0.2278, 0.0850,
         0.1688],
[0.4126, 0.4560, 0.9065, 0.1810, 0.3570, 0.3038, 0.7928, 0.3371, 0.7833,
         0.2799],
[0.7660, 0.9750, 0.1065, 0.3620, 0.7240, 0.2197, 0.8177, 0.3172, 0.6708,
         0.3936],
[0.0215, 0.1569, 0.6460, 0.4436, 0.5005, 0.4277, 0.9192, 0.8981, 0.2241,
         0.0376],
[0.3497, 0.5647, 0.0587, 0.2005, 0.0353, 0.0631, 0.1507, 0.5241, 0.9131,
         0.4331],
[0.8549, 0.2526, 0.1021, 0.1957, 0.7489, 0.0675, 0.0358, 0.2221, 0.6795,
         0.1739],
[0.0317, 0.0379, 0.8072, 0.6807, 0.3497, 0.5227, 0.4399, 0.0731, 0.8531,
         0.3566],
[0.8361, 0.9884, 0.3201, 0.6694, 0.5516, 0.4993, 0.1611, 0.8955, 0.8815,
         0.3811],
[0.1963, 0.9188, 0.3009, 0.0495, 0.6863, 0.3006, 0.9740, 0.4434, 0.2179,
         0.7639],
[0.6403, 0.5198, 0.9669, 0.1393, 0.6807, 0.0363, 0.6929, 0.2176, 0.5752,
         0.4124],
[0.4935, 0.3548, 0.6310, 0.5814, 0.3914, 0.4296, 0.5931, 0.7116, 0.9248,
         0.1524]])
```

In [3]:

```
# 3.Generate the labels y = (x1*x1+x2*x2+…+xd*xd)/d
y=(x**2).sum(dim=1)/d
y
```

Out[3]:

```
tensor([0.1552, 0.3530, 0.2517, 0.4575, 0.6186, 0.3742, 0.4342, 0.408
0, 0.3845,
        0.4068, 0.2887, 0.3627, 0.2776, 0.1809, 0.1951, 0.2567, 0.453
0, 0.3313,
        0.3127, 0.3180])
```

In [4]:

```
# 4.Implement a loss function L = (predict-y)^2
```

In [5]:

```
# 5.Use batch size of 1, that means feed data one point at a time into network and c
model_state_dict = model.state_dict()
batch_size=1
x_batch=x[:batch_size]
y_batch=y[:batch_size]
l=loss(model(x_batch),y_batch)
print('loss:',l)
```

```
loss: tensor([[0.0783]], grad_fn=<PowBackward0>)
```

In [6]:

```python
# 6.Compute the gradients using pytorch autograd:
# a. dL/dw, dL/db
# b. Print these values into a text file: torch_autograd.dat
l.backward()
for layer,param in zip(model.state_dict().keys(),model.parameters()):
    print(layer,param.grad)
    with open('torch_autograd.dat','a') as f:
        f.write(layer+':\n')
        f.write(str(param.grad)+'\n\n')
```

Open 'torch_autograd.dat' in text:

```
hidden.18.bias:
tensor([ 0.0000,  0.0000, -0.1646,  0.0391,  0.0000, -0.0528, -0.1227,
0.1546, 0.1251, -0.1609, -0.1057,  0.0132,  0.0946,  0.0000, -0.0511,
0.0292, 0.0073, -0.1562,  0.0240,  0.0000,  0.0000])

output.0.weight:
tensor([[ 0.0000,  0.0000, -0.2263, -0.0141,  0.0000, -0.0655, -0.0854,
-0.0124, -0.0443, -0.1538, -0.0348, -0.1015, -0.1281,  0.0000, -0.0499,
-0.1618, -0.0769, -0.1245, -0.0116,  0.0000,  0.0000]])

output.0.bias:
tensor([-0.7625])
```

In [7]:

```python
# 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
from torch.autograd.variable import Variable
from torch.autograd.function import Function, NestedIOFunction
from torch.autograd.gradcheck import gradcheck, gradgradcheck
from torch.autograd.grad_mode import no_grad, enable_grad, set_grad_enabled
from torch.autograd.anomaly_mode import detect_anomaly, set_detect_anomaly
from torch.autograd import profiler
```

```python
def _make_grads(outputs, grads):
    new_grads = []
    for out, grad in zip(outputs, grads):
        if isinstance(grad, torch.Tensor):
            new_grads.append(grad)
        elif grad is None:
            if out.requires_grad:
                if out.numel() != 1:
                    raise RuntimeError("grad can be implicitly created only for scalar outputs")
                new_grads.append(torch.ones_like(out))
            else:
                new_grads.append(None)
        else:
            raise TypeError("gradients can be either Tensors or None, but got " +
                            type(grad).__name__)
    return tuple(new_grads)
```

```python
def my_backward(tensors, grad_tensors=None, retain_graph=None, create_graph=False, grad_variables=None):
    if grad_variables is not None:
        warnings.warn("'grad_variables' is deprecated. Use 'grad_tensors' instead.")
        if grad_tensors is None:
            grad_tensors = grad_variables
        else:
            raise RuntimeError("'grad_tensors' and 'grad_variables' (deprecated) "
                               "arguments both passed to backward(). Please only "
                               "use 'grad_tensors'.")

    tensors = (tensors,) if isinstance(tensors, torch.Tensor) else tuple(tensors)

    if grad_tensors is None:
        grad_tensors = [None] * len(tensors)
    elif isinstance(grad_tensors, torch.Tensor):
        grad_tensors = [grad_tensors]
    else:
        grad_tensors = list(grad_tensors)

    grad_tensors = _make_grads(tensors, grad_tensors)
    if retain_graph is None:
        retain_graph = create_graph

    Variable._execution_engine.run_backward(
        tensors, grad_tensors, retain_graph, create_graph,
        allow_unreachable=True)
```

In [8]:

```python
model.zero_grad()
model.load_state_dict(model_state_dict)
my_l=loss(model(x_batch),y_batch)
my_backward(my_l)
for layer,param in zip(model.state_dict().keys(),model.parameters()):
    print(layer,param.grad)
    with open('my_autograd.dat','a') as f:
        f.write(layer+':\n')
        f.write(str(param.grad)+'\n\n')
```

Open 'my_autograd.dat' in text:

```
hidden.18.bias:
tensor([ 0.0000,  0.0000, -0.1646,  0.0391,  0.0000, -0.0528, -0.1227,
0.1546, 0.1251, -0.1609, -0.1057,  0.0132,  0.0946,  0.0000, -0.0511,
0.0292, 0.0073, -0.1562,  0.0240,  0.0000,  0.0000])

output.0.weight:
tensor([[ 0.0000,  0.0000, -0.2263, -0.0141,  0.0000, -0.0655, -0.0854,
-0.0124, -0.0443, -0.1538, -0.0348, -0.1015, -0.1281,  0.0000, -0.0499,
-0.1618, -0.0769, -0.1245, -0.0116,  0.0000,  0.0000]])

output.0.bias:
tensor([-0.7625])
```

There is no different between he two files torch_autograd.dat and my_autograd.dat.

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)

```python
import torch
import torch.nn as nn
from torchviz import make_dot
import graphviz
```

```python
def print_compute_tree(name,node):
    dot = make_dot(node)
    #print(dot)
    dot.render(name)
```

```python
if __name__=='__main__':
    torch.manual_seed(2317)
    #Sets the seed for generating random numbers. Returns a torch.Generator object.
    x = torch.randn([1,1,10],requires_grad=True)
    #Returns a tensor filled with random numbers from a standard normal distribution
    cn1 = nn.Conv1d(1,1,3,padding=1)
    #Applies a 1D convolution over an input signal composed of several input planes.
    fc1 = nn.Linear(10,10)
    #  First fully connected layer
    fc2 = nn.Linear(10,1)
    # Second fully connected layer
    y = torch.sum(x)
    #Returns the sum of all elements in the input tensor.
    c = cn1(x)
    x = torch.flatten(x)+torch.flatten(c)
    x = fc1(x)
    x = fc2(x)
    loss = torch.sum((x-y)*(x-y))
    print_compute_tree('./tree_ex' ,loss)
```

And we have to install graphviz first, which is a little complex on Mac OS. Then, we will get the tree.pdf in file

Install graphviz on Mac OSX

JULY 25, 2021MAC APP STORE

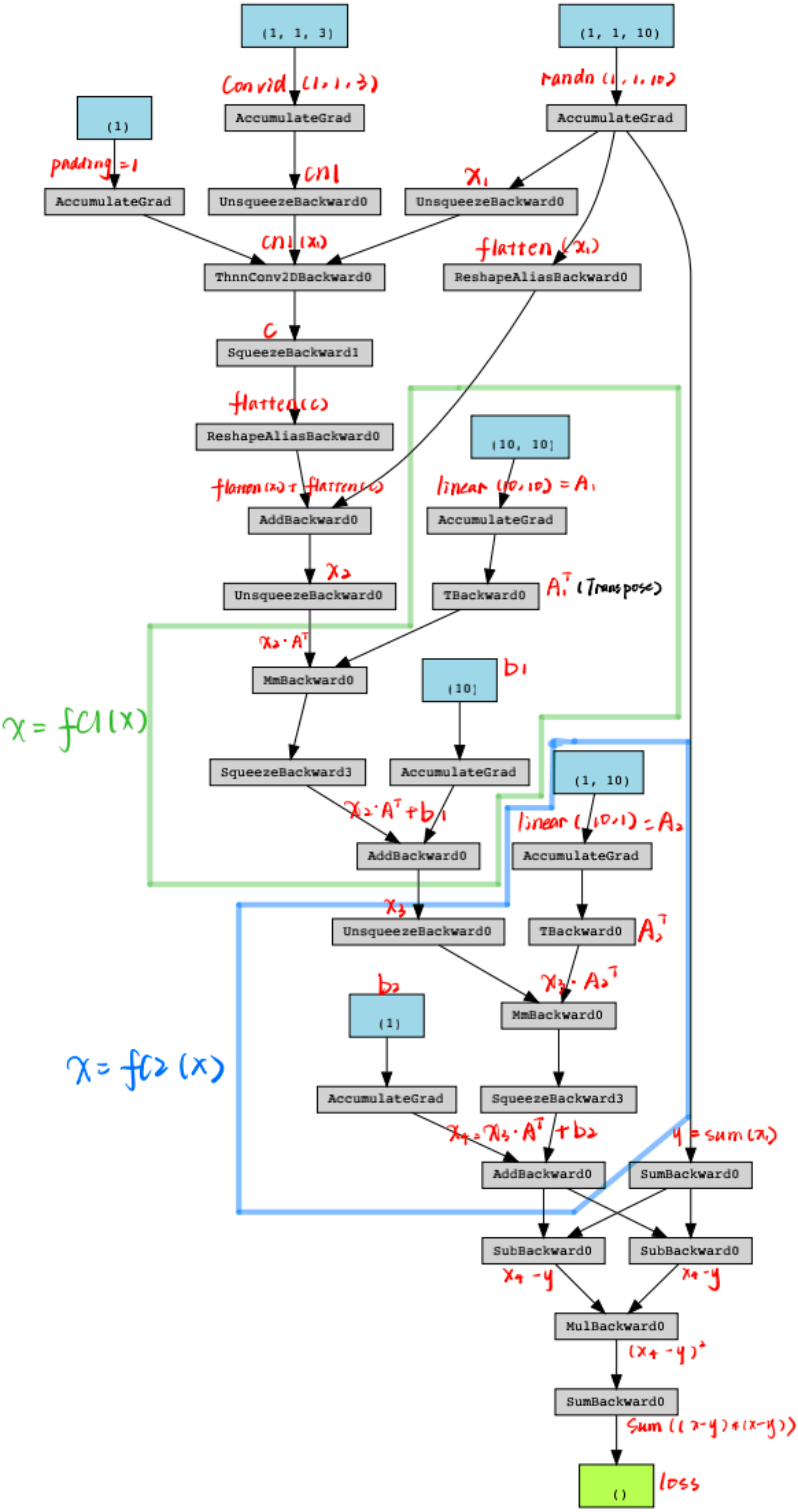
About the App

- **App name**: graphviz
- **App description**: Graph visualization software from AT&T and Bell Labs
- **App website**: http://graphviz.org/

Install the App

1. Press `Command+Space` and type ***Terminal*** and press ***enter/return*** key.
2. Run in Terminal app:
   `ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" 2> /dev/null`
   and press ***enter/return*** key. If you are prompted to enter your Mac's user password, enter it (when you type it, you wont see it on your screen/terminal.app but it would accept the input; this is to ensure no one can see your password on your screen while you type it. So just type password and press enter, even if you dont see it on your screen). Then wait for the command to finish.
3. Run:
   `brew install graphviz`

Done! You can now use `graphviz`.


(1, 1, 3)
(1, 1, 10)
Conv1d (1, 1, 3)
randn (1, 1, 10)
(1)
AccumulateGrad
AccumulateGrad
padding = 1
cn1
x1
AccumulateGrad
UnsqueezeBackward0
UnsqueezeBackward0
cn1 (x1)
flatten (x1)
ThnnConv2DBackward0
ReshapeAliasBackward0
c
SqueezeBackward1
flatten (c)
ReshapeAliasBackward0
(10, 10)
flatten (x1) + flatten (c)
linear (10, 10) = A1
AddBackward0
AccumulateGrad
x2
UnsqueezeBackward0
TBackward0
A1^T (Transpose)
x2 · A^T
MmBackward0
(10)
b1
x = fc1 (x)
SqueezeBackward3
AccumulateGrad
(1, 10)
x2 · A^T + b1
linear (10, 1) = A2
AddBackward0
AccumulateGrad
x3
UnsqueezeBackward0
TBackward0
A2^T
x3 · A2^T
b2
MmBackward0
(1)
x = fc2 (x)
AccumulateGrad
SqueezeBackward3
x4 = x3 · A^T + b2
y = sum (x1)
AddBackward0
SumBackward0
SubBackward0
SubBackward0
x4 - y
x4 - y
MulBackward0
(x4 - y)^2
SumBackward0
sum ((x-y) * (x-y))
()
loss