Lecture_14_pytorch_computation_graph_gpu_code

October 11, 2024

PyTorch: Computation Graph and GPU

1 1. Computation Graph

1.1 1.1 Simple Example

Computation graph is the internal mechanism that PyTorch uses to compute gradient. The way it works is that it will record all tensor computations, and for each tensor, it will ''remember" where the tensor is computed from.

[1]:
```
import torch

x = torch.tensor(2.0, requires_grad = True)
z = x*x # conduct some tensor computation

print(x)
print(z)
```

```
tensor(2., requires_grad=True)
tensor(4., grad_fn=<MulBackward0>)
```

You can see that the `z` tensor has value `4.0` as expected. Moreover, the `z` tensor includes `grad_fn=<MulBackward0>`. This is an internal object that remembers that `z` is computed from a multiplication of `x`. Such connections betweens tensors collectively form a computation graph.

When you call `z.backward()`, it will look at where `z` is computed from, which in our case is $z = x^2$. It will then apply the appropriate differentiation rule $\frac{dz}{dx} = 2x$ to compute what is the gradient of z with respect to x, which in our case is 2*2 = 4. The result of the gradient computation is stored in `x.grad`.

[2]:
```
z.backward()

print("The gradient of z = x*x respect to x is :", x.grad)
```

```
The gradient of z = x*x respect to x is : tensor(4.)
```

1.2 1.2 Revisit: Linear Regression Example

[3]:
```
from torch import nn

class MyLinearRegressionModel(nn.Module):
    def __init__(self,d): # d is the dimension of the input
        super(MyLinearRegressionModel,self).__init__()   # call the init␣
 ↪function of super class
        # we usually create variables for all our model parameters (w and b in␣
 ↪our case) in __init__ and give them initial values.
        # need to create them as nn.Parameter so that the model knows it is an␣
 ↪parameter that needs to be trained
        self.w = nn.Parameter(torch.zeros(1,d, dtype=torch.float))
        self.b = nn.Parameter(torch.zeros(1,dtype=torch.float))
    def forward(self,x):
        # The main purpose of the forward function is to specify given input x,␣
 ↪how the output is calculated.
        return torch.inner(x,self.w) + self.b
```

After creating the model, you will find that the "w" and "b" will have `requires_grad = True`. This is because we created w and b as `nn.Parameter`, a special type of tensor for model's trainable parameters. By default, `nn.Parameter` will have `requires_grad = True` as `nn.Parameter` is supposed to be "trainable", that is, we want to do gradient descent on these parameters.

[4]:
```
mymodel = MyLinearRegressionModel(1) # creating a model instance with input␣
 ↪dimension 1
print(mymodel.w)
print(mymodel.b)
```

```
Parameter containing:
tensor([[0.]], requires_grad=True)
Parameter containing:
tensor([0.], requires_grad=True)
```

The above is in contrast to the tensor for the dataset. For example, recall we created the `x, y` tensor as the data set to train on. By default, such non-parameter tensors have `requires_grad = False`. The reason is we don't expect to do gradient descent on `x, y` (which are just some input and output data).

[5]:
```
x = torch.arange(0,10,.1,dtype=torch.float)
x = x[:,None]
y = x*3+torch.randn(x.shape)

print(f"x.requires_grad = {x.requires_grad}")
print(f"y.requires_grad = {y.requires_grad}")
```

```
x.requires_grad = False
```

`y.requires_grad = False`

As a result, after the forward and backward, the gradient is only computed for `mymodel.w` and `mymodel.b` which has `requires_grad = True`. The gradient is not computed for `x` and `y`.

[6]:
```
prediction = mymodel(x)
loss = torch.mean((prediction - y)**2)

loss.backward()

print(f"mymodel.w.grad = {mymodel.w.grad}, mymodel.b.grad = {mymodel.b.grad}")
print(f"x.grad = {x.grad}, y.grad = {y.grad}")
```

```
mymodel.w.grad = tensor([[-195.5122]]), mymodel.b.grad = tensor([-29.5709])
x.grad = None, y.grad = None
```

A special case is "non-leaf" tensors, which are tensors resulting from some computation from other tensors. As an example,the `prediction = mymodel(x)` tensor is from some computation from `x`, and `mymodel.w`, `mymodel.b`.

If such "non-leaf" tensor is computed from at least one tensor with `requires_grad = True`, then this tensor will also have `requires_grad = True`. For the `prediction` tensor, since `mymodel.w`, `mymodel.b` have `requires_grad = True`, so `prediction` will also have `requires_grad = True`. This is because of the nature of the back-propagation, the algorithm underlying `backward()`. When calling `backward()`, the gradient for `prediction` has to be computed first before the gradient for `mymodel.w` and `mymodel.b` can be computed, so `prediction` will also have `requires_grad = True`. That being said, after the `backward()`, the gradient for such non-leaf tensors will be discarded as they are only an intermediary result and not useful.

[7]:
```
print(f"prediction.requires_grad = {prediction.requires_grad}, prediction.grad␣
↪= {prediction.grad}" )
```

```
prediction.requires_grad = True, prediction.grad = None
```

```
/var/folders/g6/z18mrwfd6xvc5_xsmtvk3yym0000gn/T/ipykernel_78835/1494579461.py:1
: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is
being accessed. Its .grad attribute won't be populated during
autograd.backward(). If you indeed want the .grad field to be populated for a
non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the
non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See
github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered
internally at /Users/runner/work/pytorch/pytorch/pytorch/build/aten/src/ATen/cor
e/TensorBody.h:494.)
  print(f"prediction.requires_grad = {prediction.requires_grad}, prediction.grad
= {prediction.grad}" )
```

1.2.1 Detach method

The `detach()` method detaches a tensor from a computation graph - it will become an independent tensor with `requires_grad = False`, and are not connected to other tensors any more.

[8]:
```
print("loss = ", loss, f"loss.requires_grad = {loss.requires_grad}")
loss_detached = loss.detach()
print("loss_detached = ", loss_detached, f"loss_detached.requires_grad =␣
 ↪{loss_detached.requires_grad}")
```

```
loss =  tensor(291.9410, grad_fn=<MeanBackward0>) loss.requires_grad = True
loss_detached =  tensor(291.9410) loss_detached.requires_grad = False
```

Detaching a tensor is necessary to convert a tensor with `requres_grad = True` to a numpy array.

[9]:
```
print("loss_detached.numpy() = ", loss_detached.numpy())

print("loss.numpy() = ", loss.numpy())
```

```
loss_detached.numpy() =  291.941
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/Users/coolq/Library/CloudStorage/Box-Box/Teaching/Tool Chain/Toolchain 2023␣
 ↪Fall/notebooks/Lecture_18_pytorch_computation_graph.ipynb Cell 21 line 3

      <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
 ↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
 ↪Lecture_18_pytorch_computation_graph.ipynb#Y105sZmlsZQ%3D%3D?line=0'>1</a>␣
 ↪print("loss_detached.numpy() = ", loss_detached.numpy())
----> <a href='vscode-notebook-cell:/Users/coolq/Library/CloudStorage/Box-Box/
 ↪Teaching/Tool%20Chain/Toolchain%202023%20Fall/notebooks/
 ↪Lecture_18_pytorch_computation_graph.ipynb#Y105sZmlsZQ%3D%3D?line=2'>3</a>␣
 ↪print("loss.numpy() = ", loss.numpy())

RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.
 ↪detach().numpy() instead.
```

1.2.2 torch.no_grad()

Sometimes, you only want to do forward without doing the backward, e.g. in validation and testing. In this case, it would be a waste of resources if torch still builds the computation graph, as the computation graph will never be used.

In this case, placing your forward pass under `with torch.no_grad()` will temporarily disable the construction of computation graph and will save computation/memory.

[10]:
```
prediction = mymodel(x)
loss = torch.mean((prediction - y)**2)
print(f"prediction.requires_grad = {prediction.requires_grad}", f"prediction.
 ↪grad_fn = {prediction.grad_fn}")
print(f"loss.requires_grad = {loss.requires_grad}", f"loss.grad_fn = {loss.
 ↪grad_fn}")
```

```
with torch.no_grad():
    prediction = mymodel(x)
    loss = torch.mean((prediction - y)**2)
    print(f"prediction.requires_grad = {prediction.requires_grad}",␣
 ↪f"prediction.grad_fn = {prediction.grad_fn}")
    print(f"loss.requires_grad = {loss.requires_grad}", f"loss.grad_fn = {loss.
 ↪grad_fn}")
    # if you try to do loss.backward() here, an error will occur
    # loss.backward()
```

```
prediction.requires_grad = True prediction.grad_fn = <AddBackward0 object at
0x7faa29906190>
loss.requires_grad = True loss.grad_fn = <MeanBackward0 object at
0x7faa2972e1c0>
prediction.requires_grad = False prediction.grad_fn = None
loss.requires_grad = False loss.grad_fn = None
```

1.3 1.3 Fine Tuning

[22]:
```
import torchvision.models as models

resnet18 = models.resnet18(pretrained=True)

print(resnet18)
```

```
/Users/coolq/opt/anaconda3/envs/sparktest2/lib/python3.9/site-
packages/torchvision/models/_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the future, please
use 'weights' instead.
  warnings.warn(
/Users/coolq/opt/anaconda3/envs/sparktest2/lib/python3.9/site-
packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed
in the future. The current behavior is equivalent to passing
`weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
```

```
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

[12]:
```python
# Freeze all layers
for param in resnet18.parameters():
    param.requires_grad = False


# Unfreeze last layer
for param in resnet18.fc.parameters():
    param.requires_grad = True
```

[13]:
```python
# Create random data
inputs = torch.randn(5, 3, 224, 224)
labels = torch.randint(0, 10, (5,))

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(resnet18.fc.parameters(), lr=0.001, momentum=0.9)

# Training loop
for epoch in range(5):
    optimizer.zero_grad()
    outputs = resnet18(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch+1}/5, Loss: {loss.item()}')
```

```
Epoch 1/5, Loss: 7.9002814292907715
Epoch 2/5, Loss: 7.787904262542725
Epoch 3/5, Loss: 7.574479579925537
Epoch 4/5, Loss: 7.270318508148193
```

```
Epoch 5/5, Loss: 6.884873867034912
```

2 2. GPU acceleration

The forward/zero_grad/backward/step procedures in the training loop can be parallized on a GPU.

2.1 2.1 Loading torch.cuda

To check whether a GPU is available on your computer, you can run `torch.cuda.is_available()`

[14]:
```
import torch.cuda

torch.cuda.is_available()
```

[14]:
```
False
```

You can also check how many GPUs are available.

[15]:
```
torch.cuda.device_count()
```

[15]:
```
0
```

2.2 2.2 Define a neural network

For illustration purpose, we will train a LeNet to learn how to recognize handwritten digits for the MNIST dataset.

LeNet is a convolutional neural network proposed by Yann Lecun in the 80s. The code of LeNet is as below. The details of how a convolutional neural network works is beyond the scope of this course.

[2]:
```
# Realization of LeNet
import torch.nn as nn

class LeNet(nn.Module):
    def __init__(self):
        super().__init__()

        # convolution layers
        self._body = nn.Sequential(
            # First convolution Layer
            # input size = (32, 32), output size = (28, 28)

            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5),
            # ReLU activation

            nn.ReLU(),

            # Max pool 2-d
```

```
            nn.MaxPool2d(kernel_size=2),

            # Second convolution layer
            # input size = (14, 14), output size = (10, 10)
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            # output size = (5, 5)
        )

        # Fully connected layers
        self._head = nn.Sequential(
            # First fully connected layer
            # in_features = total number of weights in last conv layer = 16 * 5␣
↪* 5
            nn.Linear(in_features=256, out_features=120),

            # ReLU activation
            nn.ReLU(inplace=True),

            # second fully connected layer
            # in_features = output of last linear layer = 120
            nn.Linear(in_features=120, out_features=84),

            # ReLU activation
            nn.ReLU(inplace=True),

            # Third fully connected layer which is also output layer
            # in_features = output of last linear layer = 84
            # and out_features = number of classes = 10 (MNIST data 0-9)
            nn.Linear(in_features=84, out_features=10)
        )

    def forward(self, x):
        # apply feature extractor
        x = self._body(x)
        # flatten the output of conv layers
        # dimension should be batch_size * number_of weight_in_last conv_layer
        x = x.view(x.size()[0], -1)
        # apply classification head
        x = self._head(x)
        return x
```

2.3 2.3 Training with GPU acceleration

To conduct the forward/zero_grad/backward/step procedures with GPU acceleration, you need to move the model and the data to the GPU device.

For example, suppose the neural network model is named `mynn`, then run `mynn = mynn.to(device = device)` to move the model to a given device. For GPU device, set `device = torch.device('cuda:0')`, where `cuda:0` means the default GPU.

Similarly, upon loading `x_batch, y_batch` from the dataloader, run

```
x_batch = x_batch.to(device)
y_batch = y_batch.to(device)
```

to move `x_batch, y_batch` to GPU.

Once the model and the data are on the GPU, all the subsequent forward/zero_grad/backward/step procedures will automatically be implemented in a parallized manner on GPU.

You may not be able to run the following code on your local computer because you may not have GPU on your computer. You can run the code on CoLab with GPU enabled, and for convenience, we have created a seperate notebook `lecture_20_gpu_demo.ipynb` for you to upload to CoLab.

[4]:
```
import torch,torchvision
from torch.utils.data import Dataset,DataLoader
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np
import time


# Choose between CPU or GPU (cuda:0)
device = torch.device('cpu')
# device = torch.device('cuda:0')

# move model to device
mynn = LeNet()
mynn = mynn.to(device = device)

# get dataset


# Three hyper parameters for training
lr = .04
batch_size = 32
N_epochs = 5

# Create dataloaders for training and validation
mydataset = datasets.FashionMNIST(
    root="data",
    train=True,
```

```
    download=True,
    transform=torchvision.transforms.ToTensor()
)
train_dataloader = DataLoader(mydataset, batch_size = batch_size, shuffle =␣
 ↪True)

# Create optimizer
optimizer = torch.optim.SGD(mynn.parameters(), lr = lr) # this line creates a␣
 ↪optimizer, and we tell optimizer we are optimizing the parameters in mymodel

losses = [] # training losses of each epoch
num_batches = len(train_dataloader)

for epoch in range(N_epochs):
    batch_loss = []
    per_batch_time = 0.0
    for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
        start_time = time.time()
        # data to device
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        # pass input data to get the prediction outputs by the current model
        prediction = mynn(x_batch)

        # compare prediction and the actual output label and compute the loss
        loss = nn.functional.cross_entropy(prediction,y_batch)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()

        end_time = time.time()
        per_batch_time += (end_time - start_time)

        # add this loss to batch_loss for later computation
        batch_loss.append(loss.detach().numpy())

    losses.append(np.mean(np.array(batch_loss)))
    per_batch_time = per_batch_time/num_batches
    print(f"Epoch = {epoch}, device = {device}, per_batch_time =␣
 ↪{per_batch_time}, train_loss = {losses[-1]}")
```

```
Epoch = 0, device = cpu, per_batch_time = 0.0032194700876871747, train_loss =
```

```
0.8680732250213623
Epoch = 1, device = cpu, per_batch_time = 0.003127245203653971, train_loss =
0.4853483736515045
Epoch = 2, device = cpu, per_batch_time = 0.003118114344278971, train_loss =
0.4081261157989502
Epoch = 3, device = cpu, per_batch_time = 0.003064979298909505, train_loss =
0.36758360266685486
Epoch = 4, device = cpu, per_batch_time = 0.0033786468505859375, train_loss =
0.34055203199386597
```