**Introduction to pytorch**

Pytorch is a framework developed by Facebook. This is a giant in technology that invests a lot of resources in the development of Artificial Intelligence. Pytorch is developed with an open source license so it makes for a very large community. A large community means a lot of resources to learn and your problems may have already been solved by someone else. share with the community. Pytorch along with Tensorflow and Keras is one of the popular frameworks used in Deep Learning problems today. In particular, in research fields, almost all authors use pytorch to develop their problems. Pytorch shows its advantage in the research field because it is very easy for you to debug and visualize, in addition it follows the Dynamic Graphs mechanism that allows to reduce the time to train the model.

**Tensor in pytorch**

Tensor is like Numpy array but converted to tensor for use on GPU. Numpy's math, transformation, and basic operations are all possible on Tensor. Here, I will give some basic syntax on Tensor so that you can understand the basics

#Tạo một Tensor từ list cho trước sử dụng torch.Tensor

t = torch.Tensor([[1,2,3],[3,4,5]])

#Tạo một Tensor với kích thước (2, 3) cho trước và có giá trị ngẫu nhiên tuân theo phân phối chuẩn với trung vị bằng 0 và phương sai bằng 1

t = torch.randn(2, 3)

# Tạo một Tensor với kích thước (2, 3) cho trước và tất cả phần tử có giá trị đều bằng 1

t = torch.ones(2, 3)

# Tạo một Tensor với kích thước (2, 3) cho trước và tất cả phần tử có giá trị đều bằng 0

t = torch.zeros(2, 3)

#Tạo một tensor có kích thước (2,3) với giá trị nằm trong khoảng từ 0->10

t = torch.randint(low = 0,high = 10,size = (2,3))

#Sử dụng torch.from\_numpy để chuyển đổi từ Numpy array sang Tensor

a = np.array([[1,2,3],[3,4,5]])

t = torch.from\_numpy(a)

# Sử dụng .numpy() để chuyển đổi từ Tensor sang Numpy array

t = t.numpy()

Numpy arrays vs PyTorch tensors.

Numpy is a Python library for storing and processing calculations with real numeric data. However, Numpy is written in C / C ++, so processing and computation speed is very fast.

PyTorch tensors are similar in function and purpose to Numpy arrays, but with a few more advantages:

* Perform fast computation on GPU, because DL models that want to be accelerated need to be processed through GPUs, so tensors that support fast computation on GPUs are essential.
* Pytorch tensors can save computational graphs, so they can calculate derivatives quickly, serving the backpropagation algorithm in Deep Learning, I will talk about it in the next lesson about autograd.

Torch Properties.

Torch tensors contain only numeric and bool (True/False) data. Each torch tensor belongs to a data type, in the dtype attribute. Here is a list of the data types torch tensors can contain:

torch.float32 or torch.float: 32-bit floating-point

torch.float64 or torch.double: 64-bit, double-precision floating-point

torch.float16 or torch.half: 16-bit, half-precision floating-point

torch.int8: signed 8-bit integers

torch.uint8: unsigned 8-bit integers

torch.int16 or torch.short: signed 16-bit integers

torch.int32 or torch.int: signed 32-bit integers

torch.int64 or torch.long: signed 64-bit integers

torch.bool: Boolean

**Torch Storage**

**Storage**

Actually the values in the tensor will be stored in a contiguous area of memory, managed by torch.Storage. Storage is a 1-dimensional array of numbers with the same data type.

x = torch.tensor([[1,2,3],[4,5,6]]) x.storage() # output: 1,2,3,4,5,6 x[1][2] == x.storage()[5] # output: True

Tensor metadata: Size, offset, and stride

In order for the tensor to get the value from storage, we need some information: size, offset and stride.

* Offset is the starting position to store the value of the tensor in storage.
* Size is the size of the tensor.
* Stride has the same dimension as Size, which means how many elements it needs to jump in storage to get to the next element in that dimension.

Torch GPU

Torch allows tensor to be stored in GPU for parallel computation as well as speed up processing.

To initialize a tensor and save it on the gpu, I use the device.

x\_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')

Or I can copy 1 tensor from CPU to GPU

x = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])

x\_gpu = x.to(device='cuda')

Each tensor is only stored on a certain GPU, so if there are multiple GPUs, you must specify which GPU to save on, the GPU index also starts from 0.

x\_gpu = x.to(device='cuda:0')

# hoặc

x\_gpu = x.cuda(0)

x\_gpu = x\_gpu + 4 # Thực hiện phép tính trên GPU

To convert from GPU to CPU, I use

x\_cpu = x\_gpu.to(device='cpu')

# hoặc

x\_cpu = x\_gpu.cpu()

Torch Tensor to Numpy Array

Torch allows converting tensors to Numpy arrays. The properties of size and shape will be preserved, and the type will change from Torch to Numpy.

x = torch.tensor([1,2,3])

x\_np = x.numpy()

If the tensor is stored on the CPU, the Torch tensor and the Numpy array will share the same memory, so changing the value in one variable will also change the value of the other variable.

x[1] = 0

print(x) # output: [1, 0, 3]

print(x\_np) # output: [1, 0, 3]

If tensor is stored on GPU, people will not be able to convert tensor to Numpy array directly, but we need to copy the content of tensor to CPU first and then convert to Numpy array. Therefore, the two variables on gpu and np do not share the same memory and modifying one does not affect the other.

x\_gpu = torch.tensor([1, 2, 3], device='cuda')

x\_np = x\_gpu.numpy() # Error

x\_np = x\_gpu.cpu().numpy() # ok

x\_gpu[1] = 0 print(x\_gpu) # output: [1, 0, 3]

print(x\_np) # output: [1, 2, 3]

Similarly, I can convert Numpy array to Torch tensor. Torch tensor will be stored in CPU and 2 variables on np and cpu will share memory.

x\_np = np.array([1, 2, 3])

x\_cpu = torch.from\_numpy(x\_np)

Autograd

The derivative calculation mechanism in Pytorch is called Autograd (AUTOMATIC DIFFERENTIATION PACKAGE)

Requires\_grad

In order for pytorch to save and calculate the derivative of a tensor, I assign the requires\_grad = True attribute to that tensor and Pytorch only allows float tensor to be assigned the requires\_grad = True attribute.

when initializing tensor, the default requires\_grad = False.

a = torch.tensor([2., 3.])

a.requires\_grad # False

# Gán thuộc tính requires\_grad = True

a = torch.tensor([2., 3.], requires\_grad=True)

a.requires\_grad # True

# Hoặc

a = torch.tensor([2., 3.])

a.requires\_grad = True

The requires\_grad property is spread, i.e. if a has requires\_grad=True, then other tensors computed from a will also have requires\_grad=True.

a = torch.tensor([2., 3.], requires\_grad=True)

a.requires\_grad # True

b = a\*\*2 c = 2\*b

print(b.requires\_grad) # True

print(c.requires\_grad) # True

Even if the calculation of a tensor with requires\_grad and a tensor without requires\_grad, the output still outputs a tensor with requires\_grad

a = torch.tensor([2., 3.], requires\_grad=True)

b = torch.tensor([1., 1.])

print(a.requires\_grad) # True

print(b.requires\_grad) # False

c = a + b # [3., 4.]

print(c.requires\_grad) # True

When tensor has requires\_grad, Pytorch will build computational graph.

x = torch.tensor([3])

y = torch.tensor([10])

a = torch.tensor([1.], requires\_grad=True)

b = torch.tensor([2.], requires\_grad=True)

y\_hat = a\*x + b

z = y\_hat - y

L = z\*\*2

Backward

When calling y.backward, I pass a tensor vector with the same size as y, the meaning can be understood as the derivative of loss with y.

x = torch.tensor([1., 2., 3.], requires\_grad=True)

y = 2\*x + 1

y.backward(gradient=torch.tensor([1, 2, 1]))

print(x.grad) # 2, 4, 2

Neural Network with Pytorch

Pytorch supports the torch.nn library for building neural networks. It includes the necessary blocks to build a complete neural network. Each layer in the network is called a module and inherits from nn.Module. Each module will have a Parameter property (eg W, b in Linear Regression) to be optimized during the modeling process.

Linear regression model

# nn.Linear(số\_node\_input, số node\_output) trong layer đó linear\_model = nn.Linear(1, 1)

I can get the parameters in linear\_model

list(linear\_model.parameters()) # [Parameter containing: tensor([[14.9464]], requires\_grad=True), Parameter containing: tensor([1.1261], requires\_grad=True)]

In each layer, there will be 2 Parameters, W and b respectively. I see that the requires\_grad attribute of the Parameters are all True to be able to calculate backward loss and use gradient descent.

Loss, I use the function nn.MSELoss(), without re-implementing it.

loss\_fn = nn.MSELoss()

Pytorch supports optimizers in nn.optim to help people implement gradient descent

optimizer = optim.SGD(linear\_model.parameters(), lr=0.00004)

Then at each epoch:

for epoch in range(1, n\_epochs + 1):

y\_hat = model(x)

loss = loss\_fn(y\_hat, y)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

When I put data into the model to learn, instead of learning data one by one, I will let many data points be learned at the same time in batches, the data I pass in will have the size (batch\_size, 1) or in the field. The general synthesis with NN is (batch\_size, num\_features), abbreviated (N\*d).

x = torch.tensor(data[:,0], dtype=torch.float32) # x.shape (30)

# Thêm chiều vào vị trí tương ứng, ví dụ (30) -> (30, 1). x = x.unsqueeze(1) # x.shape (30, 1)

The torchvision library supports many models

from torchvision import models

dir(models)

# output:

['AlexNet', 'DenseNet', 'GoogLeNet', 'GoogLeNetOutputs', 'Inception3', 'InceptionOutputs', 'MNASNet', 'MobileNetV2', 'MobileNetV3', 'ResNet', 'ShuffleNetV2', 'SqueezeNet', 'VGG', 'alexnet', 'densenet', 'densenet121', 'densenet161', 'densenet169', 'densenet201', 'resnet', 'resnet101', 'resnet152', 'resnet18', 'resnet34', 'resnet50', 'detection', 'googlenet', 'inception', 'inception\_v3', ]

From torchvision.models I can load models, for example I use resnet50. Inside there is a pretrained = True parameter, which means that I used the pretrained model resnet50 with the weight trained with the ImageNet dataset. And if pretrained = False, then I will use resnet50 model with randomly initialized weight.

resnet = models.resnet50(pretrained=True)

To see the layers in the resnet model I can print out

print(resnet)

What is state\_dict?

The model's state\_dict is a Python dict, with the key being the name of the layer and the value being the parameter of that layer, including weight and bias. Besides the model, the optimizer (torch.optim) also has a state\_dict, which contains information about the optimizer's state, as well as associated hyperparameters.

Since state\_dict is a Python dict, it can be easily updated, changed, saved, and uploaded. For example, the CNN model I built in the article Neural Network

class Net(nn.Module):

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

super().\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, kernel\_size=3, padding=1) self.conv2 = nn.Conv2d(16, 8, kernel\_size=3, padding=1) self.fc1 = nn.Linear(8 \* 8 \* 8, 32)

# bài toán phân loại 10 lớp nên output ra 10 nodes self.fc2 = nn.Linear(32, 10)

def forward(self, x):

out = F.max\_pool2d(torch.tanh(self.conv1(x)), 2)

out = F.max\_pool2d(torch.tanh(self.conv2(out)), 2)

# flatten về dạng vector để cho vào neural network

out = out.view(-1, 8 \* 8 \* 8)

out = torch.tanh(self.fc1(out))

out = self.fc2(out) return out # Initialize model

model = Net() # Initialize optimizer

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

I can print the state\_dict of the model and optimizer.

# Print model's state\_dict

print("Model's state\_dict:")

for param\_tensor in model.state\_dict():

print(param\_tensor, "\t", model.state\_dict()[param\_tensor].size())

'''

Output: Model's state\_dict:

conv1.weight torch.Size([16, 3, 3, 3])

conv1.bias torch.Size([16])

conv2.weight torch.Size([8, 16, 3, 3])

conv2.bias torch.Size([8])

fc1.weight torch.Size([32, 512]) fc1.bias torch.Size([32]) fc2.weight torch.Size([10, 32]) fc2.bias torch.Size([10])

You can see that the state\_dict of the model has the key as the layer name such as conv1.weight, conv1.bias,…,fc2.weight, fc2.bias and the value is the coefficient tensor corresponding to those layers.

Store model's state\_dict

torch.save(model.state\_dict(), PATH)

When loading the model, I need to rebuild the model's architecture first, then call the function to load the state\_dict into the model.

model = Net()

model.load\_state\_dict(torch.load(PATH))

Save the whole model

Normally Pytorch will save the model as .pt or .pth

torch.save(model, PATH)

Save and load checkpoint to continue training

When I save the model to continue training, besides the model state\_dict, I also have to save additional information such as optimizer state\_dict, number of epochs, current loss.

torch.save({ 'epoch': epoch, 'model\_state\_dict': model.state\_dict(), 'optimizer\_state\_dict': optimizer.state\_dict(), 'loss': loss }, PATH

Next time I train again, I will upload the saved parameters to continue training

checkpoint = torch.load(PATH) model.load\_state\_dict(checkpoint['model\_state\_dict']) optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

epoch = checkpoint['epoch']

loss = checkpoint['loss']

# train model

model.train()

# tiếp tục train model