BUILD THE NEURAL NETWORK

Neural networks comprise of layers/modules that perform operations on data. The [torch.nn](https://pytorch.org/docs/stable/nn.html) namespace provides all the building blocks you need to build your own neural network. Every module in PyTorch subclasses the [nn.Module](https://pytorch.org/docs/stable/generated/torch.nn.Module.html). A neural network is a module itself that consists of other modules (layers). This nested structure allows for building and managing complex architectures easily.

In the following sections, we’ll build a neural network to classify images in the FashionMNIST dataset.

**import** os

**import** torch

**from** torch **import** **nn**

**from** torch.utils.data **import** **DataLoader**

**from** torchvision **import** **datasets,** **transforms**

Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let’s check to see if [torch.cuda](https://pytorch.org/docs/stable/notes/cuda.html) is available, else we continue to use the CPU.

**device** **=** 'cuda' **if** **torch.cuda.is\_available()** **else** 'cpu'

print**(**f'Using {**device**} device'**)**

Out:

Using cuda device

Define the Class

We define our neural network by subclassing nn.Module, and initialize the neural network layers in \_\_init\_\_. Every nn.Module subclass implements the operations on input data in the forward method.

**class** **NeuralNetwork(nn.Module):**

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

super**(NeuralNetwork,** self**).**\_\_init\_\_**()**

self**.flatten** **=** **nn.Flatten()**

self**.linear\_relu\_stack** **=** **nn.Sequential(**

**nn.Linear(**28**\***28**,** 512**),**

**nn.ReLU(),**

**nn.Linear(**512**,** 512**),**

**nn.ReLU(),**

**nn.Linear(**512**,** 10**),**

**)**

**def** **forward(**self**,** **x):**

**x** **=** self**.flatten(x)**

**logits** **=** self**.linear\_relu\_stack(x)**

**return** **logits**

We create an instance of NeuralNetwork, and move it to the device, and print its structure.

**model** **=** **NeuralNetwork().to(device)**

print**(model)**

Out:

NeuralNetwork(

(flatten): Flatten(start\_dim=1, end\_dim=-1)

(linear\_relu\_stack): Sequential(

(0): Linear(in\_features=784, out\_features=512, bias=True)

(1): ReLU()

(2): Linear(in\_features=512, out\_features=512, bias=True)

(3): ReLU()

(4): Linear(in\_features=512, out\_features=10, bias=True)

)

)

To use the model, we pass it the input data. This executes the model’s forward, along with some [background operations](https://github.com/pytorch/pytorch/blob/270111b7b611d174967ed204776985cefca9c144/torch/nn/modules/module.py#L866). Do not call model.forward() directly!

Calling the model on the input returns a 10-dimensional tensor with raw predicted values for each class. We get the prediction probabilities by passing it through an instance of the nn.Softmax module.

**X** **=** **torch.rand(**1**,** 28**,** 28**,** **device=device)**

**logits** **=** **model(X)**

**pred\_probab** **=** **nn.Softmax(dim=**1**)(logits)**

**y\_pred** **=** **pred\_probab.argmax(**1**)**

print**(**f"Predicted class: {**y\_pred**}"**)**

Out:

Predicted class: tensor([9], device='cuda:0')

Model Layers

Let’s break down the layers in the FashionMNIST model. To illustrate it, we will take a sample minibatch of 3 images of size 28x28 and see what happens to it as we pass it through the network.

**input\_image** **=** **torch.rand(**3**,**28**,**28**)**

print**(input\_image.size())**

Out:

torch.Size([3, 28, 28])

### nn.Flatten

We initialize the [nn.Flatten](https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html) layer to convert each 2D 28x28 image into a contiguous array of 784 pixel values ( the minibatch dimension (at dim=0) is maintained).

**flatten** **=** **nn.Flatten()**

**flat\_image** **=** **flatten(input\_image)**

print**(flat\_image.size())**

Out:

torch.Size([3, 784])

### nn.Linear

The [linear layer](https://pytorch.org/docs/stable/generated/torch.nn.Linear.html) is a module that applies a linear transformation on the input using its stored weights and biases.

**layer1** **=** **nn.Linear(in\_features=**28**\***28**,** **out\_features=**20**)**

**hidden1** **=** **layer1(flat\_image)**

print**(hidden1.size())**

Out:

torch.Size([3, 20])

### nn.ReLU

Non-linear activations are what create the complex mappings between the model’s inputs and outputs. They are applied after linear transformations to introduce nonlinearity, helping neural networks learn a wide variety of phenomena.

In this model, we use [nn.ReLU](https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html) between our linear layers, but there’s other activations to introduce non-linearity in your model.

print**(**f"Before ReLU: {**hidden1**}\n\n"**)**

**hidden1** **=** **nn.ReLU()(hidden1)**

print**(**f"After ReLU: {**hidden1**}"**)**

Out:

Before ReLU: tensor([[ 0.0394, 0.4881, -0.2893, 0.4827, 0.5349, -0.5114, 0.2003, -0.0861,

0.0211, -0.0266, -0.6103, 0.0533, -0.1037, -0.1142, 0.7322, 0.2388,

0.1105, 0.0577, -0.5333, -0.2239],

[ 0.2532, 0.0914, -0.4058, 0.2776, 0.8942, -0.7504, 0.0648, 0.1197,

0.1227, -0.0314, -0.2471, 0.0224, -0.2912, -0.2891, 0.5831, -0.0054,

0.3160, -0.0549, -0.3157, -0.1836],

[ 0.3184, 0.3533, -0.0850, 0.4517, 1.1259, -0.4292, 0.1215, 0.2031,

0.2627, -0.4550, -0.2749, 0.0108, -0.0942, 0.0040, 0.3829, -0.0461,

-0.2068, -0.0583, -0.4131, -0.5878]], grad\_fn=<AddmmBackward0>)

After ReLU: tensor([[0.0394, 0.4881, 0.0000, 0.4827, 0.5349, 0.0000, 0.2003, 0.0000, 0.0211,

0.0000, 0.0000, 0.0533, 0.0000, 0.0000, 0.7322, 0.2388, 0.1105, 0.0577,

0.0000, 0.0000],

[0.2532, 0.0914, 0.0000, 0.2776, 0.8942, 0.0000, 0.0648, 0.1197, 0.1227,

0.0000, 0.0000, 0.0224, 0.0000, 0.0000, 0.5831, 0.0000, 0.3160, 0.0000,

0.0000, 0.0000],

[0.3184, 0.3533, 0.0000, 0.4517, 1.1259, 0.0000, 0.1215, 0.2031, 0.2627,

0.0000, 0.0000, 0.0108, 0.0000, 0.0040, 0.3829, 0.0000, 0.0000, 0.0000,

0.0000, 0.0000]], grad\_fn=<ReluBackward0>)

### nn.Sequential

[nn.Sequential](https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html) is an ordered container of modules. The data is passed through all the modules in the same order as defined. You can use sequential containers to put together a quick network like seq\_modules.

**seq\_modules** **=** **nn.Sequential(**

**flatten,**

**layer1,**

**nn.ReLU(),**

**nn.Linear(**20**,** 10**)**

**)**

**input\_image** **=** **torch.rand(**3**,**28**,**28**)**

**logits** **=** **seq\_modules(input\_image)**

### nn.Softmax

The last linear layer of the neural network returns logits - raw values in [-infty, infty] - which are passed to the [nn.Softmax](https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html) module. The logits are scaled to values [0, 1] representing the model’s predicted probabilities for each class. dim parameter indicates the dimension along which the values must sum to 1.

**softmax** **=** **nn.Softmax(dim=**1**)**

**pred\_probab** **=** **softmax(logits)**

Model Parameters

Many layers inside a neural network are *parameterized*, i.e. have associated weights and biases that are optimized during training. Subclassing nn.Module automatically tracks all fields defined inside your model object, and makes all parameters accessible using your model’s parameters() or named\_parameters() methods.

In this example, we iterate over each parameter, and print its size and a preview of its values.

print**(**"Model structure: "**,** **model,** "\n\n"**)**

**for** **name,** **param** **in** **model.named\_parameters():**

print**(**f"Layer: {**name**} | Size: {**param.size()**} | Values : {**param[:**2**]**} \n"**)**

Out:

Model structure: NeuralNetwork(

(flatten): Flatten(start\_dim=1, end\_dim=-1)

(linear\_relu\_stack): Sequential(

(0): Linear(in\_features=784, out\_features=512, bias=True)

(1): ReLU()

(2): Linear(in\_features=512, out\_features=512, bias=True)

(3): ReLU()

(4): Linear(in\_features=512, out\_features=10, bias=True)

)

)

Layer: linear\_relu\_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[ 0.0323, 0.0152, 0.0083, ..., -0.0112, -0.0318, -0.0196],

[-0.0094, 0.0084, -0.0083, ..., 0.0014, -0.0171, 0.0075]],

device='cuda:0', grad\_fn=<SliceBackward0>)

Layer: linear\_relu\_stack.0.bias | Size: torch.Size([512]) | Values : tensor([0.0068, 0.0231], device='cuda:0', grad\_fn=<SliceBackward0>)

Layer: linear\_relu\_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[-0.0020, 0.0114, -0.0166, ..., 0.0026, 0.0099, 0.0258],

[ 0.0275, 0.0008, -0.0290, ..., -0.0293, -0.0044, 0.0330]],

device='cuda:0', grad\_fn=<SliceBackward0>)

Layer: linear\_relu\_stack.2.bias | Size: torch.Size([512]) | Values : tensor([ 0.0434, -0.0340], device='cuda:0', grad\_fn=<SliceBackward0>)

Layer: linear\_relu\_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[ 0.0269, -0.0220, -0.0351, ..., -0.0392, 0.0343, 0.0249],

[-0.0248, -0.0286, 0.0430, ..., 0.0346, 0.0092, -0.0256]],

device='cuda:0', grad\_fn=<SliceBackward0>)

Layer: linear\_relu\_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0143, 0.0374], device='cuda:0', grad\_fn=<SliceBackward0>)

Further Reading

* [torch.nn API](https://pytorch.org/docs/stable/nn.html)

**Total running time of the script:** ( 0 minutes 6.346 seconds)