# Build the Neural Network

pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

Neural networks comprise of layers/modules that perform operations on data. The torch.nn namespace provides all the building blocks you need to build your own neural network. Every module in PyTorch subclasses the nn.Module. 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 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. 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([7], 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 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 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 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([[ 0.0243,  0.1528,  0.3291, -0.3993,  0.0261,  0.5131, -0.0781,
0.0009,
         -0.0701,  0.7571, -0.3199, -0.2240, -0.1630, -0.4166,  0.3677,  0.2397,
         -0.3969, -0.0211, -0.1858, -0.0602],
        [-0.0572,  0.0168,  0.0232,  0.0168, -0.2071, -0.1296, -0.2205,  0.0526,
         -0.1163,  0.7483, -0.1032,  0.0233, -0.1022, -0.2331, -0.0169,  0.6094,
         -0.6465, -0.1062, -0.1283, -0.1658],
        [-0.2706, -0.0502,  0.4091, -0.1694, -0.0021, -0.0069, -0.0837, -0.1120,
         -0.1444,  0.7707, -0.5680,  0.0765,  0.2619, -0.5335,  0.1178,  0.2009,
         -0.2483,  0.1838, -0.0406,  0.1116]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.0243, 0.1528, 0.3291, 0.0000, 0.0261, 0.5131, 0.0000, 0.0009,
0.0000,
         0.7571, 0.0000, 0.0000, 0.0000, 0.0000, 0.3677, 0.2397, 0.0000, 0.0000,
         0.0000, 0.0000],
        [0.0000, 0.0168, 0.0232, 0.0168, 0.0000, 0.0000, 0.0000, 0.0526, 0.0000,
         0.7483, 0.0000, 0.0233, 0.0000, 0.0000, 0.0000, 0.6094, 0.0000, 0.0000,
         0.0000, 0.0000],
        [0.0000, 0.0000, 0.4091, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.7707, 0.0000, 0.0765, 0.2619, 0.0000, 0.1178, 0.2009, 0.0000, 0.1838,
         0.0000, 0.1116]], grad_fn=<ReluBackward0>)
```

## nn.Sequential

nn.Sequential 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 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(f"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([[
0.0059, -0.0219, -0.0003,  ..., -0.0088,  0.0087,  0.0024],
        [ 0.0018, -0.0215, -0.0305,  ..., -0.0132,  0.0101,  0.0164]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0355,
-0.0062], device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values :
tensor([[-0.0094, -0.0400,  0.0406,  ..., -0.0222,  0.0327,  0.0073],
        [ 0.0222, -0.0188, -0.0322,  ...,  0.0048, -0.0146, -0.0175]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([0.0002,
0.0354], device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values :
tensor([[-0.0121,  0.0316,  0.0260,  ...,  0.0105,  0.0363, -0.0346],
        [ 0.0234, -0.0189,  0.0089,  ..., -0.0048, -0.0343, -0.0133]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([0.0032,
0.0333], device='cuda:0', grad_fn=<SliceBackward0>)
```

---