Colab Notebook GitHub

Build the Neural Network

Created On: Feb 09, 2021 | Last Updated: Jan 24, 2025 | Last Verified: Not Verified

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 an accelerator such as CUDA, MPS, MTIA, or XPU. If the current accelerator is available, we will use it. Otherwise, we use the CPU.

```
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_ava
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().__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 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output. 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([6], 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([[ 0.0838, -0.1596,  0.2112,  0.2359, -0.3761,  0.4043, -0.401
          0.2180, -0.2139, -0.5868,  0.3046, -0.1642,  0.0262,  0.5605, -0.1890,
          0.5140, -0.2630,  0.4404,  0.1834],
        [-0.0791, -0.3621, -0.0149,  0.4168, -0.0165,  0.3271, -0.0582, -0.1739,
         -0.0459, -0.4254, -0.4844,  0.1458, -0.0997,  0.2241,  0.2173,  0.0705,
          0.2485, -0.2096,  0.1545,  0.0299],
        [ 0.0156, -0.0276,  0.1354,  0.2339, -0.4241,  0.3049, -0.4130, -0.7753,
          0.2701, -0.4634, -0.8258, -0.1060, -0.2186, -0.2815,  0.4303, -0.2589,
          0.6219, -0.0349,  0.3395,  0.0192]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.0838, 0.0000, 0.2112, 0.2359, 0.0000, 0.4043, 0.0000, 0.000
         0.0000, 0.0000, 0.3046, 0.0000, 0.0262, 0.5605, 0.0000, 0.5140, 0.0000,
         0.4404, 0.1834],
        [0.0000, 0.0000, 0.0000, 0.4168, 0.0000, 0.3271, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.1458, 0.0000, 0.2241, 0.2173, 0.0705, 0.2485, 0.0000,
         0.1545, 0.0299],
        [0.0156, 0.0000, 0.1354, 0.2339, 0.0000, 0.3049, 0.0000, 0.0000, 0.2701,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.4303, 0.0000, 0.6219, 0.0000,
         0.3395, 0.0192]], 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
```