Building a PyTorch Model Class: A Step-by-Step Guide

January 5, 2025

In the previous blog post, we explored three main aspects of model training in PyTorch:

1. **Data**: Methods to load your data and make it available for PyTorch.
2. **Model**: A class describing your model structure and forward pass. PyTorch simplifies this by requiring only the forward pass definition, leveraging its built-in gradient tracking for the backward pass.
3. **Training Loop**: The process where data is loaded, the model is trained, loss is calculated, gradients are propagated, and parameters are updated.

Previously, we focused on the **Data** part, discussing the `Dataset` and `DataLoader` classes. With your training and validation datasets prepared and wrapped in `DataLoader` objects, the next step is to build the **Model** class.

If you've worked with other deep learning frameworks before, you'll find PyTorch's approach to model building refreshingly transparent. Rather than configuring models through abstract APIs, PyTorch lets you define neural networks in a way that closely resembles how you'd implement them mathematically, with clear operations and transformations applied to tensors.

This post will guide you through creating a model class in PyTorch, including its essential components, practical implementation, and testing. We will also introduce concepts for future exploration, such as PyTorch's gradient tracking and backpropagation.

Key Requirements for a PyTorch Model Class

A custom model class in PyTorch must:

1. **Inherit from `torch.nn.Module`** : This is the base class for all neural network modules in PyTorch.
2. **Implement an `__init__` method**: This method defines the model's components, such as layers and activation functions.
3. **Implement a `forward` method**: This method defines the forward pass, specifying how input data propagates through the model components.

Components Available in PyTorch

PyTorch provides a comprehensive library of layers and activation functions, such as:

- **Convolutional Layers**: For processing image data.
- **Linear Layers**: General matrix multiplication.
- **Transformer Layers**: For sequence modeling.
- **Activation Functions**: ReLU, ELU, etc.

You can explore the full list of components in the PyTorch documentation.

Example: Understanding the `Linear` Layer

The `Linear` layer performs a general matrix multiplication:

$$Y = XW^T + b$$

Where:

- $X \in \mathbb{R}^{k \times i}$: Input matrix.
- $W \in \mathbb{R}^{o \times i}$: Weight matrix.
- $b \in \mathbb{R}^{o}$: Bias term.
- $Y \in \mathbb{R}^{k \times o}$: Output matrix.

Here, k is the batch size, i is the input dimension, and o is the output dimension. Note that `Linear` layers do not apply activation functions internally, unlike TensorFlow's `Dense` layers. You must explicitly add activation functions, such as `ReLU`, after the `Linear` layer if needed.

Constructing the Forward Method

The `forward` method describes the data flow through the model, akin to building a computational graph:

- **Nodes**: Defined in the `__init__` method (e.g., layers).
- **Edges**: Defined in the `forward` method, specifying the flow between nodes.

For example, consider the FashionMNIST dataset, where each image is a single-channel tensor of shape ((1, 28, 28)), and the goal is to classify the image into one of 10 categories. A simple model could include:

1. **Flatten Layer**: Converts the ((1, 28, 28)) tensor into a 1D tensor of size 784.
2. **Linear Layers**: Two hidden layers (512 units each) and one output layer (10 logits).
3. **ReLU Activation**: Applied after each hidden layer.
4. **SoftMax (Optional)**: Converts logits to probabilities.

Implementation

Here is the implementation of a simple neural network for the FashionMNIST dataset:

```python
import torch
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.l1 = nn.Linear(28 * 28, 512)
        self.l2 = nn.Linear(512, 512)
        self.l3 = nn.Linear(512, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.flatten(x)
        x = self.l1(x)
        x = self.relu(x)
        x = self.l2(x)
        x = self.relu(x)
        logits = self.l3(x)
        return logits

model = NeuralNetwork()
print(model)
```

Output:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (l1): Linear(in_features=784, out_features=512, bias=True)
  (l2): Linear(in_features=512, out_features=512, bias=True)
  (l3): Linear(in_features=512, out_features=10, bias=True)
  (relu): ReLU()
)
```

Note that the PyTorch quick start tutorial implements the model above using the `nn.Sequential` interface, which is similar to the Keras Sequential API. This approach simplifies defining your model when it has a straightforward, linear topology. However, for more complex architectures with branching or multiple inputs/outputs, using the functional approach shown above (where we call layers as functions on tensors) is preferred and offers more customization.

Inspecting Model Parameters

To better understand your model, you can inspect its layers, parameter shapes, and total parameter count. This is particularly useful for debugging and optimizing your model.

Listing Model Parameters

PyTorch provides the `.parameters()` method to access all parameters in the model. To print layer-wise details:

```
for name, param in model.named_parameters():
    print(f"Layer: {name} | Shape: {param.shape} | Requires Grad: {param.
```

This prints the name of each parameter (e.g., weights and biases), their shape, and whether they require gradients.

Output:

```
Layer: l1.weight | Shape: torch.Size([512, 784]) | Requires Grad: True
Layer: l1.bias   | Shape: torch.Size([512])     | Requires Grad: True
Layer: l2.weight | Shape: torch.Size([512, 512]) | Requires Grad: True
Layer: l2.bias   | Shape: torch.Size([512])     | Requires Grad: True
Layer: l3.weight | Shape: torch.Size([10, 512]) | Requires Grad: True
Layer: l3.bias   | Shape: torch.Size([10])      | Requires Grad: True
```

Here, `l1`, `l2`, and `l3` are the three `Linear` layers defined in the model class. Since `ReLU` and `Flatten` layers do not have parameters, they are not included in `model.named_parameters()`. The `requires_grad` attribute indicates whether gradients are computed for a parameter, which we will discuss in detail in the next blog post.

Counting Total Parameters

To calculate the total number of parameters in the model:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params}")
```

Output:

```
Total parameters: 669706
```

You can also split this into trainable and non-trainable parameters, which is useful in transfer learning scenarios:

```
trainable_params = sum(p.numel() for p in model.parameters() if p.require
non_trainable_params = total_params - trainable_params
print(f"Trainable parameters: {trainable_params}")
print(f"Non-trainable parameters: {non_trainable_params}")
```

Output:

```
Trainable parameters: 669706
Non-trainable parameters: 0
```

Non-trainable parameters are those whose `requires_grad` attribute is set to `False`. These often appear in frozen layers when fine-tuning pre-trained models.

Inspecting Individual Layers

For a specific layer, you can directly access its parameters. For example, to inspect the first linear layer:

```
print(f"Weights of l1: {model.l1.weight.shape}")
print(f"Bias of l1: {model.l1.bias.shape}")
```

Output:

```
Weights of l1: torch.Size([512, 784])
Bias of l1: torch.Size([512])
```

This approach helps validate the dimensions and ensures that the data flows correctly through the network.

Testing the Model

Before deploying your model for training, it's always a good practice to perform a quick test to verify the dimensions flow correctly through your architecture. Let's generate a random tensor with the expected input dimensions and pass it through the model:

```
x_test = torch.rand(1, 28, 28)  # One sample with shape matching FashionM
print(model(x_test))
```

Output:

```
tensor([[-0.0179,  0.0328,  0.0971, -0.0456, -0.0692, -0.0543,  0.0189, -
         -0.0626,  0.0481]], grad_fn=<AddmmBackward0>)
```

This confirms the model produces an output tensor of size 10 (our number of classes). The values are random at this point since the model weights are initialized randomly. During training, these values will be optimized so that the largest logit value corresponds to the correct class prediction.

This testing step may seem trivial for a simple model, but it becomes crucial when dealing with complex architectures where tensor shape mismatches can be hard to debug later in the training process.

Understanding `grad_fn`

The `grad_fn=<AddmmBackward0>` in the output indicates PyTorch has created a computational graph for this tensor, recording the operation that generated it (matrix multiplication and addition in this case). We will delve deeper into computational graphs and gradient tracking in the next blog post.

Conclusion

In this post, we covered the essential components of a PyTorch model class, how to define its structure, and how to implement and test the forward pass.

One thing I particularly appreciate about PyTorch's approach is how it balances simplicity and flexibility. The basic model structure is straightforward, but there's virtually no limit to the complex architectures you can build by extending these concepts. Whether you're implementing a standard neural network like we did here or designing something cutting-edge, the same principles apply.

Key takeaways include:

- PyTorch simplifies model implementation by handling gradient computations automatically
- The `forward` method defines the computational graph, specifying how data flows through the model
- PyTorch's object-oriented design makes it easy to organize, inspect, and debug your models
- Testing ensures the model operates as expected and produces the desired output dimensions

In the upcoming post, we will explore PyTorch's gradient tracking and computational graphs in greater detail, including implementing backpropagation for a `Linear` layer from scratch. This will give you a deeper understanding of what's happening "under the hood" during model training.

References

- PyTorch Quick Start Tutorial

↗ rss

↗ source code

↗ github

↗ linkedin

↗ resume

↗ google scholar