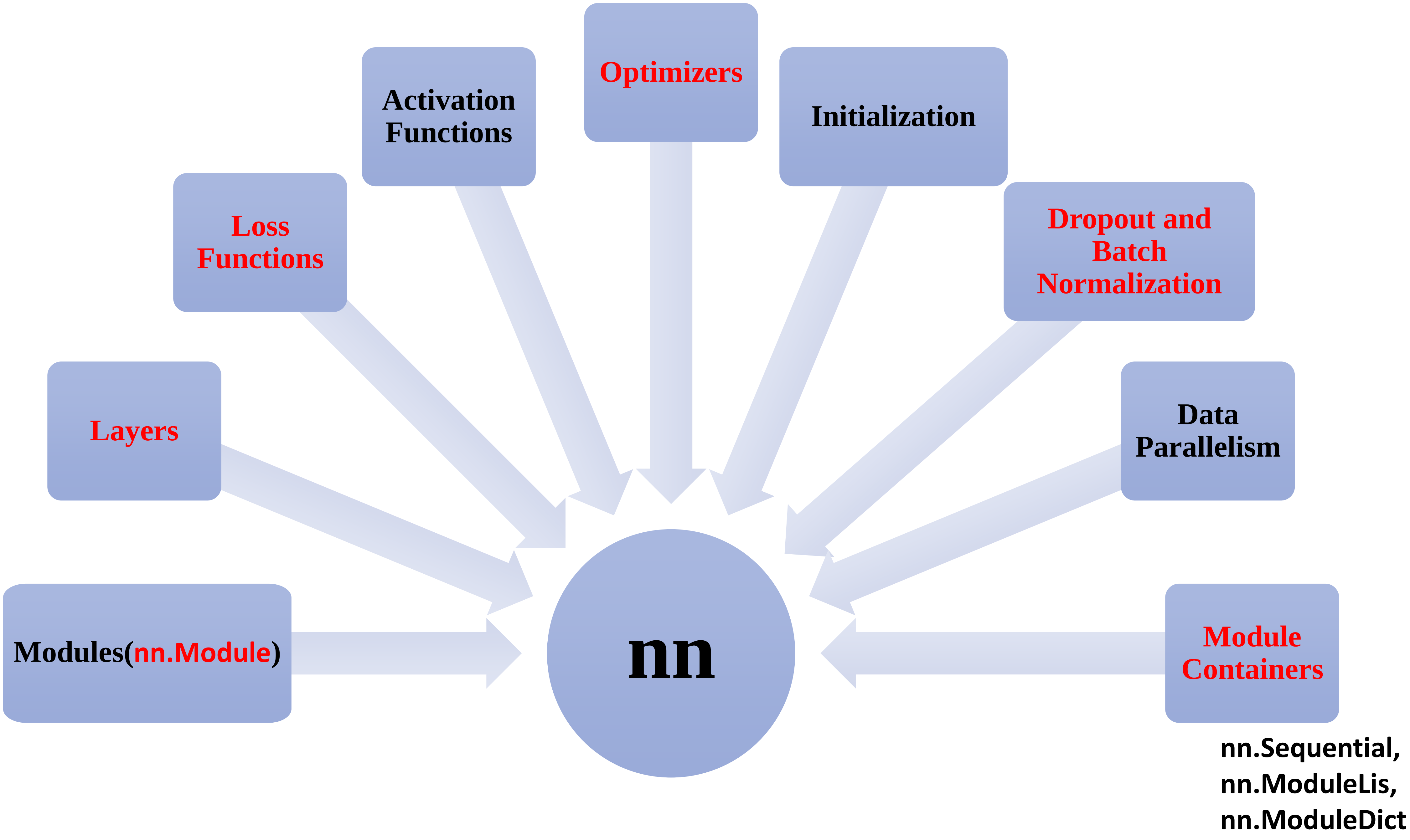

Optimizers
Activation Functions
Initialization
Loss Functions
Dropout and Batch Normalization
Layers
Data Parallelism
Modules(nn.Module)
nn
Module Containers
nn.Sequential,
nn.ModuleLis,
nn.ModuleDict

nn.Module provides a structured and organized way to create and manage neural network architectures in PyTorch

nn.Module

Parameter Management: Layers defined as attributes within an **nn.Module** **automatically register their parameters**

Nested Modules: You can nest multiple **nn.Module** instances inside each other, **creating complex architectures**

Device Handling: The to() method allows you to move the entire model to a specified device (CPU or GPU).

Saving and Loading: nn.Module makes it easier to save and load models and their trained weights

Training and Evaluation: By defining the forward pass in the **forward** method, **model.train()** and **model.eval()**.

Customization: custom methods, properties, and behaviors in your subclass to extend the functionality of your model

Layers in Pytorch

Fully Connected Layer `nn.Linear()`

Convolutional Layer `nn.Conv2d()`

ReLu Layer `nn.ReLU()`

MaxPooling Layer `nn.MaxPool2d()`

Dropout Layer `nn.Dropout(p)`

Batch Normalization Layer `nn.BatchNorm2d()`

```
conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

```
batchnorm_layer=nn.BatchNorm2d(num_features)
```

```
relu_layer = nn.ReLU()
```

```
maxpool_layer = nn.MaxPool2d(kernel_size, stride)
```

```
Linear_layer=nn.Linear(in_features,out_features)
```

```
dropout_layer = nn.Dropout(p)
```

1. **In PyTorch, a neural network is constructed using the concept of layers**.

2. Layers are the building blocks of a neural network, and they define the operations that transform the input data into meaningful representations.

3. PyTorch provides a variety of pre-defined layers that you can use to build your neural network architecture.

Linear Layer (Fully Connected Layer): This layer performs a linear transformation on the input data by multiplying it with a weight matrix and adding a bias term.

Layer=Wx+b

```
import torch.nn as nn

Linear_layer=nn.Linear(in_features,out_features)
```

Convolutional Layer: **Convolutional layers are used for processing grid-like data, such as images. They apply a set of learnable filters to the input data.**

```
import torch.nn as nn
conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

ReLu Layer: **The Rectified Linear Unit (ReLU) layer applies the activation function ReLU(x) = max(0, x) element-wise to the input.**

```
import torch.nn as nn
relu_layer = nn.ReLU()
```

Convolutional Layer: Convolutional layers are used for processing grid-like data, such as images. They apply a set of learnable filters to the input data.

```
import torch.nn as nn
conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

ReLu Layer: The Rectified Linear Unit (ReLU) layer applies the activation function ReLU(x) = max(0, x) element-wise to the input.

```
import torch.nn as nn
relu_layer = nn.ReLU()
```

MaxPooling Layer: MaxPooling layers downsample the spatial dimensions of the input by selecting the maximum value from each local region.

```
import torch.nn as nn
maxpool_layer = nn.MaxPool2d(kernel_size, stride)
```

Dropout Layer: Dropout is a regularization technique that randomly sets a fraction of input units to zero during each forward pass.

```
import torch.nn as nn
dropout_layer = nn.Dropout(p)
```

Batch Normalization Layer: Batch normalization normalizes the activations of a layer across a batch of data, helping with faster training and better generalization.

```
import torch.nn as nn
batchnorm_layer = nn.BatchNorm2d(num_features)
```

2D convolutional layer

The 2D convolutional layer, often denoted as nn.Conv2d in PyTorch, is a fundamental building block in convolutional neural networks (CNNs). This layer performs a 2D convolution operation on the input data, which is particularly useful for image-related tasks.

nn.Conv2d Module

Parameters:

in_channels: The number of input channels (e.g., **for an RGB image, in_channels would be 3**).
out_channels: The number of output channels (i.e., the number of filters/kernels to be applied).
kernel_size: The size of the convolutional kernel (filter).
stride: The stride of the convolution.
padding: The zero-padding added to both sides of the input.
bias: If True, a bias term is added to the output.

Attributes:

weight: The learnable weights (filters/kernels) of the convolutional layer.
bias: The learnable bias term.

Forward Method:

- The forward method implements the forward pass of the convolutional layer. It takes an input tensor and applies the convolution operation using the specified parameters.

2D convolutional layer

```
import torch
import torch.nn as nn

# Example Conv2d layer
conv_layer = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1)

# Accessing parameters
weights = conv_layer.weight
biases = conv_layer.bias

# Forward pass
input_data = torch.randn(1, 3, 32, 32)  # Batch size of 1, 3 channels, 32x32 image
output_data = conv_layer(input_data)

# Print details
print("Conv2d Layer:")
print(conv_layer)
print("\nWeights Shape:", weights.shape)
print("Biases Shape:", biases.shape)
print("\nInput Shape:", input_data.shape)
print("Output Shape:", output_data.shape)
```

Weights and Biases:
The **weight tensor has dimensions** (out_channels, in_channels, kernel_size[0], kernel_size[1]).
The **bias** tensor has dimensions (out_channels).

output

```
Conv2d Layer: Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Weights Shape: torch.Size([64, 3, 3, 3])                note w=(out_channels, in_channels, kernel_size[0], kernel_size[1]).
Biases Shape: torch.Size([64])
Input Shape: torch.Size([1, 3, 32, 32])
Output Shape: torch.Size([1, 64, 32, 32])
```

2D convolutional layer

Weights and Biases:

The weight tensor has dimensions (out_channels, in_channels, kernel_size[0], kernel_size[1]).
The bias tensor has dimensions (out_channels).

Input and Output Shapes:

The input shape is (batch_size, in_channels, height, width).
The output shape is (batch_size, out_channels, height_out, width_out).

Parameters Initialization:

The learnable parameters (weights and biases) are initialized during the instantiation of the nn.Conv2d module.

Forward Pass:

The forward method performs the convolution operation on the input tensor.

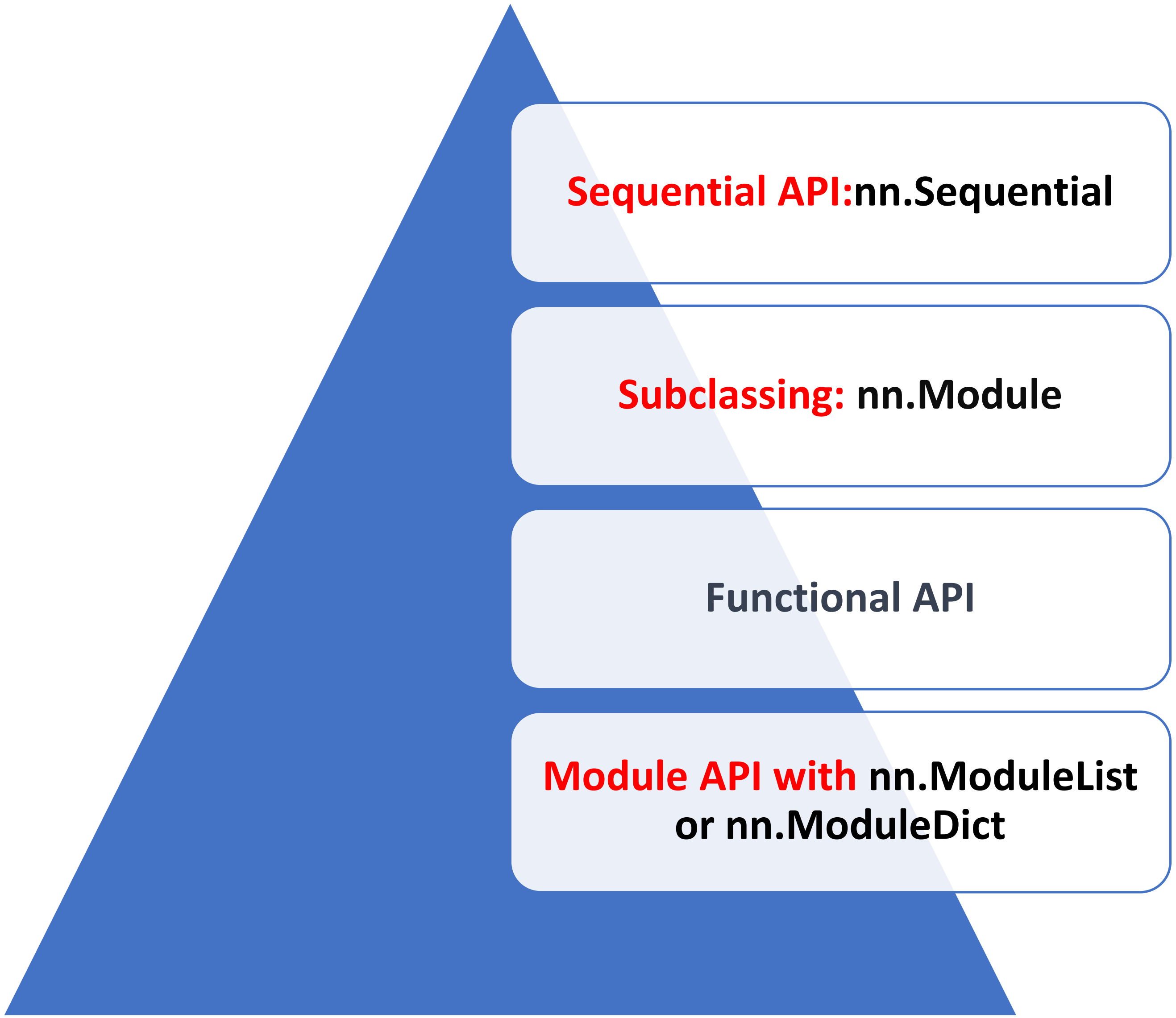
Padding and Stride:

Padding and stride can be adjusted to control the spatial dimensions of the output tensor.

Activation Function:

The convolutional layer itself does not include an activation function. Typically, it is followed by a separate activation layer like nn.ReLU.

Model Design in Pytorch


Sequential API:nn.Sequential
Subclassing: nn.Module
Functional API
Module API with nn.ModuleList
or nn.ModuleDict

1. Sequential API:

The **nn.Sequential class in PyTorch allows** you to create a neural network by stacking layers sequentially. It's a convenient way to define networks when the architecture follows a linear structure without branching or complex connections.

```
import torch.nn as nn

model = nn.Sequential(nn.Linear(input_size, hidden_size),
        nn.ReLU(),
        nn.Linear(hidden_size, output_size))
```

Subclassing nn.Module:

Creating custom classes by subclassing **nn.Module** gives you more flexibility to define complex architectures, branching structures, and non-sequential networks. This approach is useful when you need more control over how the layers are connected and how data flows through the network.

2. Subclassing nn.Module:

Creating custom classes by subclassing **nn.Module** gives you more flexibility to define complex architectures, branching structures, and non-sequential networks. This approach is useful when you need more control over how the layers are connected and how data flows through the network.

```
import torch.nn as nn

class CustomNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CustomNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
model = CustomNet(input_size, hidden_size, output_size)
```

3.Functional API:

The functional API allows you to define complex architectures using PyTorch functions directly. **This approach is useful when you want to perform operations that are not layer-based, such as concatenation or element-wise operations.**

```
import torch.nn.functional as F

class FunctionalNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(FunctionalNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = FunctionalNet(input_size, hidden_size, output_size)
```

4. Module API with nn.ModuleList or nn.ModuleDict:
When your model includes multiple layers that are not defined in a sequential order, you can use **nn.ModuleList** or **nn.ModuleDict** to manage them.

```
class ComplexModel(nn.Module):
    def __init__(self):
        super(ComplexModel, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(input_size, hidden_size),
            nn.ReLU(),nn.Linear(hidden_size, num_classes)])

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

model = ComplexModel()
```

4. Module API with nn.ModuleDict:

nn.ModuleDict is a PyTorch module that can be used to hold a collection of PyTorch modules as attributes. It is a way to organize and manage multiple sub-modules within a larger module.

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        # Define a ModuleDict to hold sub-modules
        self.module_dict = nn.ModuleDict({
            'conv1': nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1),
            'batch_norm': nn.BatchNorm2d(64),
            'relu': nn.ReLU(),
            'fc': nn.Linear(64 * 28 * 28, 10)  # Example fully connected layer
        })
    def forward(self, x):
        # Access and apply sub-modules in the forward pass
        x = self.module_dict['conv1'](x)
        x = self.module_dict['batch_norm'](x)
        x = self.module_dict['relu'](x)
        # Reshape for fully connected layer
        x = x.view(x.size(0), -1)
        # Apply fully connected layer
        x = self.module_dict['fc'](x)
        return x
```

Parameters:

1. In PyTorch, parameters refer to the learnable weights and biases in a neural network.

2. These parameters are associated with instances of the nn.Parameter class, which is a subclass of torch.Tensor.

3. Parameters are tensors that are marked as parameters of a module and can be updated during the training process using optimization algorithms.

1. Creating Parameters:

Parameters are typically created as attributes of a custom neural network module (a class that inherits from nn.Module).

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.weight = nn.Parameter(torch.randn(5, 5))  # Example parameter

    def forward(self, x):
        # Use the parameter in the forward pass
        output = torch.mm(x, self.weight)
        return output
```

Parameters:

2. Accessing Parameters:

Parameters are accessible through the parameters() method of an nn.Module instance.

```
model = MyModel()
for param in model.parameters():
    print(param)
```

This loop prints all the parameters of the MyModel instance.

Parameters:

3. Optimizer and Parameter Updates:

During training, an optimizer is used to update the **parameters based on the gradients computed during backpropagation**.

```
import torch.optim as optim

model = MyModel()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Inside the training loop
for input_data, target in training_data:
    optimizer.zero_grad()  # Zero the gradients
    output = model(input_data)
    loss = loss_function(output, target)
    loss.backward()  # Compute gradients
    optimizer.step()  # Update parameters
```

In this example, optimizer.step() updates the parameters of the model based on the **computed gradients**.

Parameters:

4. Freezing Parameters:

You can freeze specific parameters during training by setting their **requires_grad** attribute to **False**.

```
for param in model.parameters():
    param.requires_grad = False  # Freeze all parameters

model.weight.requires_grad = True  # Unfreeze specific parameter
```

This is useful when you want to fine-tune only certain parts of a pre-trained model.

register parameters in pytorch

In PyTorch, you can use the **register_parameter** method to manually register a parameter that is not a direct attribute of an **nn.Module subclass**. This method allows you to add a parameter to the module and specify its name. Here's an example to illustrate how to use **register_parameter**

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        # Manually register a parameter
        self.register_parameter('custom_param', nn.Parameter(torch.randn(5, 5)))
    def forward(self, x):
        # Use the registered parameter in the forward pass
        output = torch.mm(x, self.custom_param)
        return output
# Create an instance of the model
model = MyModel()
# Access the registered parameter
print("Custom Parameter:")
print(model.custom_param)
# Access all parameters using parameters()
print("\nAll Parameters:")
for name, param in model.named_parameters():
    print(name, param.size())
```