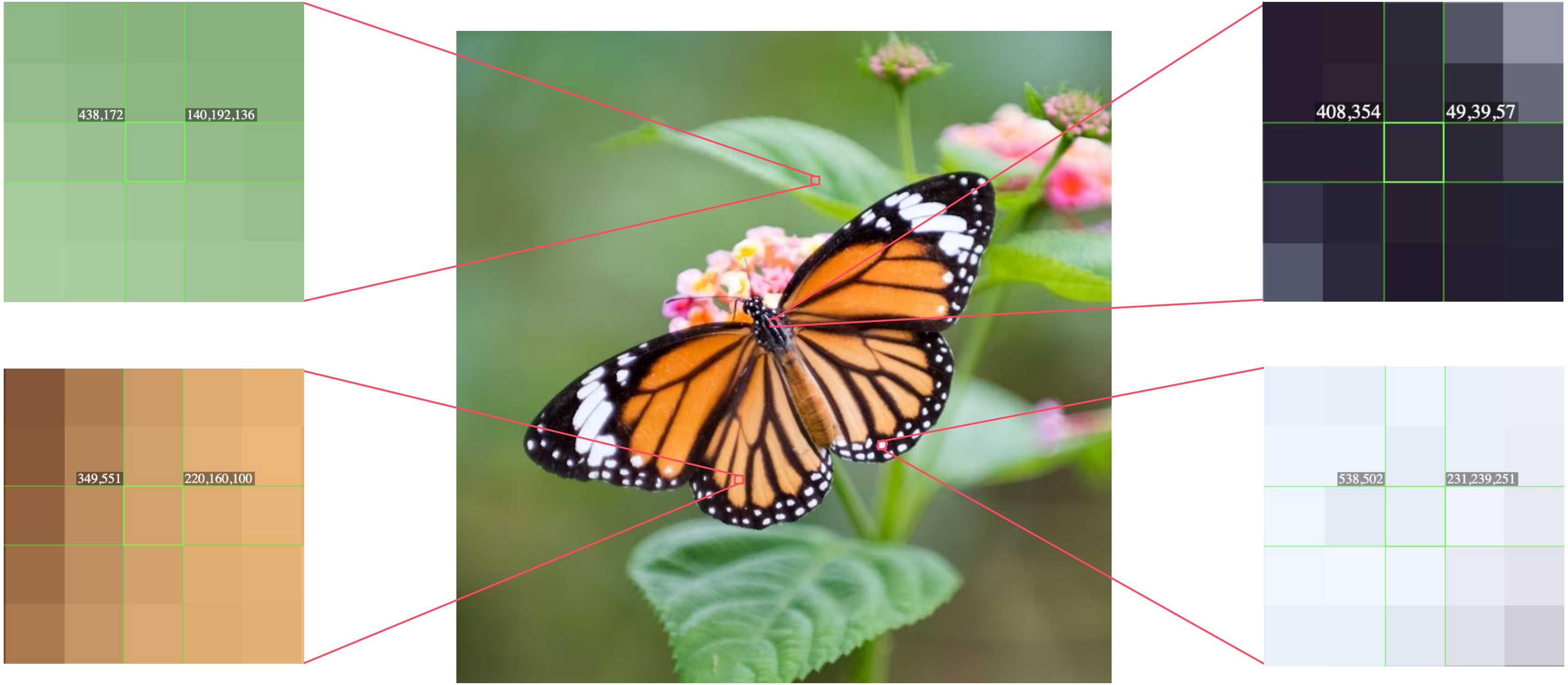
DeepLearning.AI

Convolutional Neural Networks

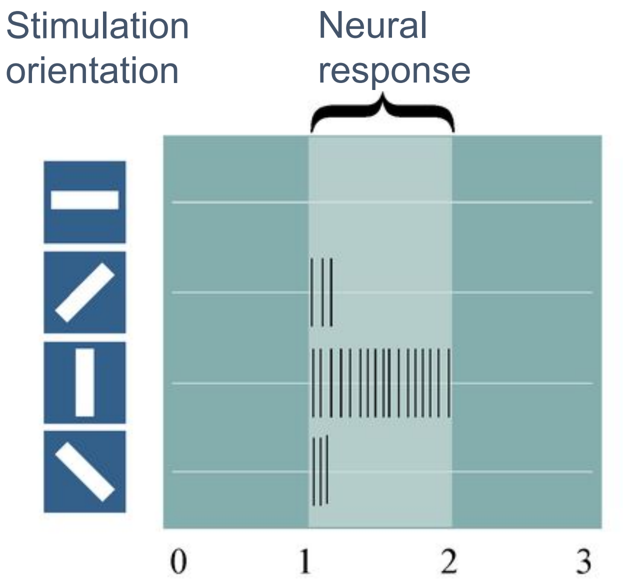
Part 1: Filters, Patterns, and Feature Maps

Core Neural Network Components

Linear layers treat every pixel as independent



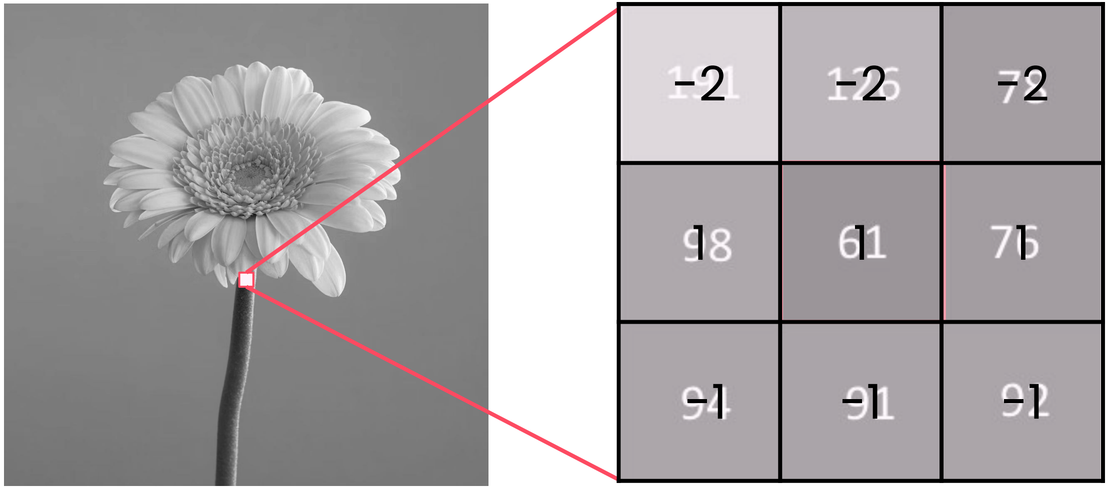
CNNs mimic biological vision



Hubel and Wiesel (1962)

Laurence Moroney

Convolutional Filters



−2	−2	−2
1	1	1
−1	−1	−1

Convolutional Filters

191*(-2)	126*(-2)	78*(-2)
98*(1)	61*(1)	76*(1)
94*(-1)	91*(-1)	92*(-1)

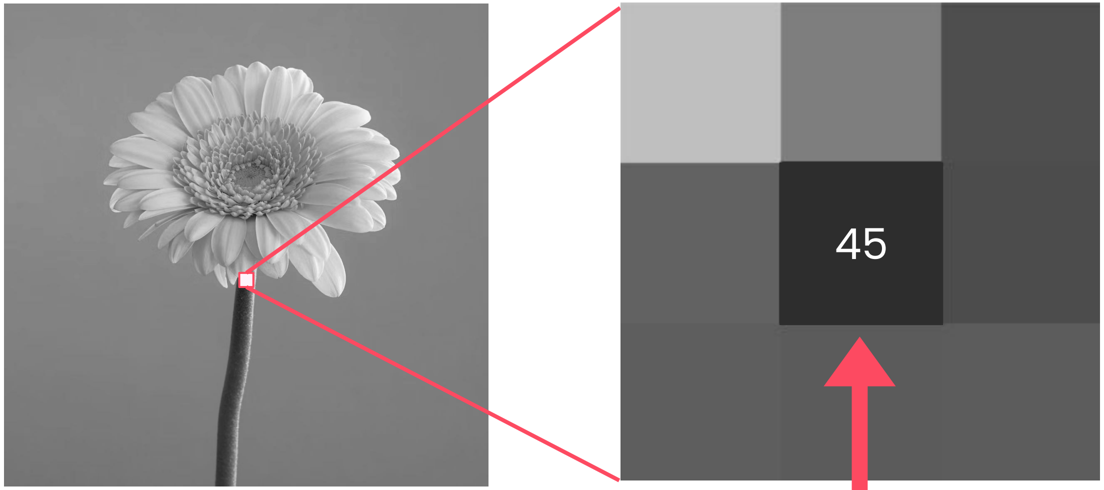
Convolutional Filters


```
New_value = sum(-382,-252,-156,98,122,152,188,91,184) + bias
```

DeepLearning.AI

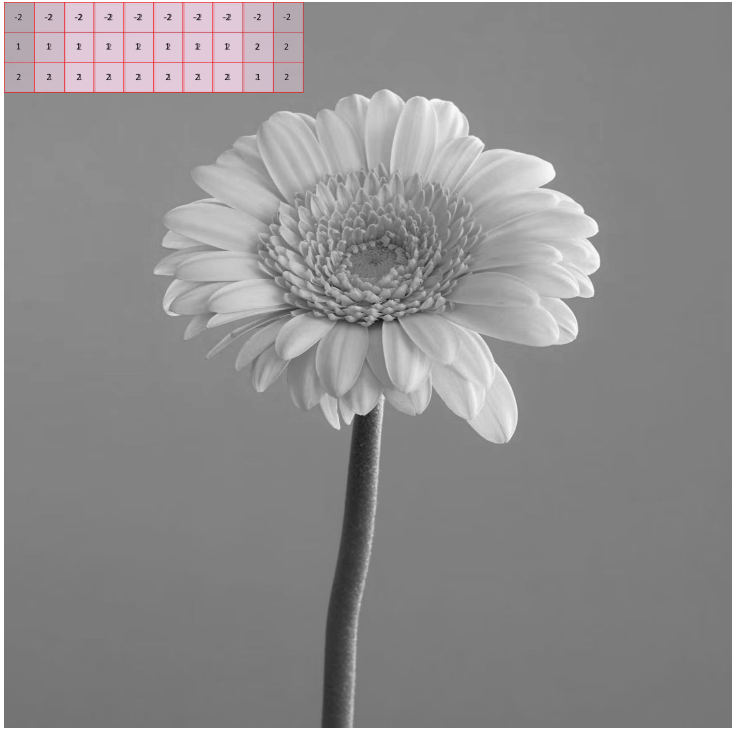
Laurence Moroney

Convolutional Filters



`New_value = 45`

How does convolution work?

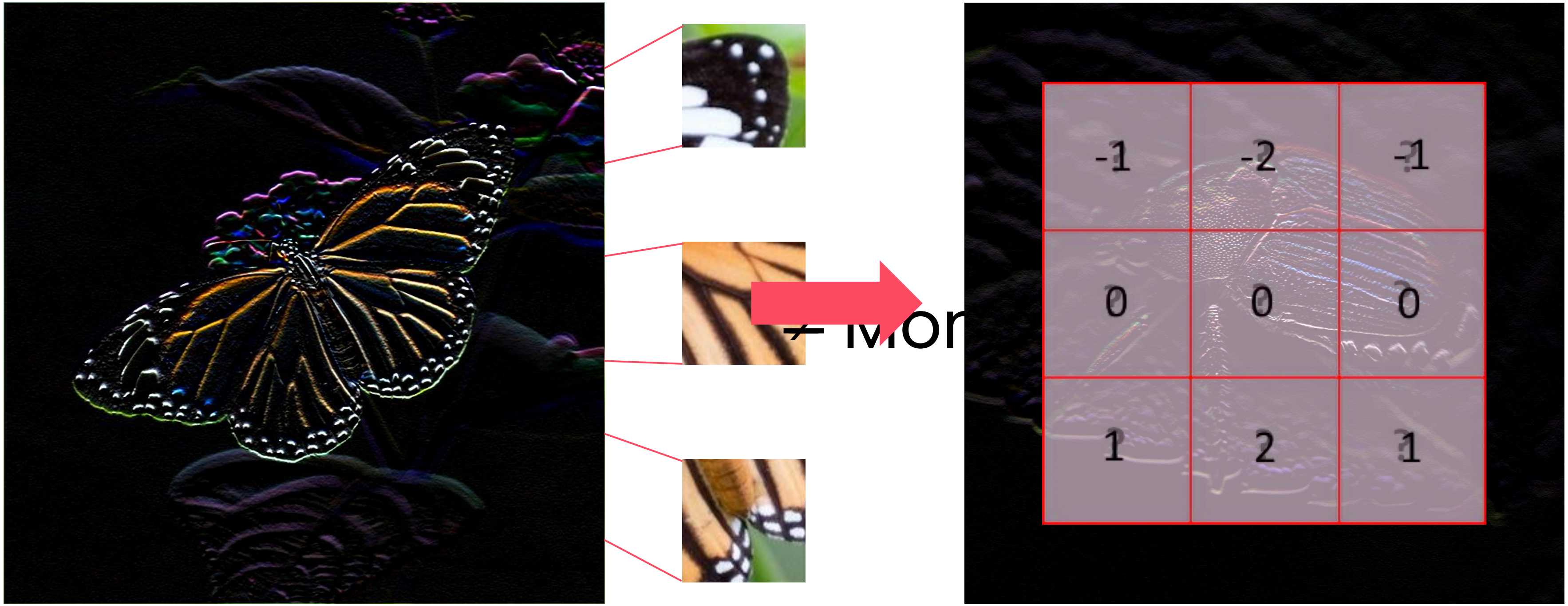


What does this filter do?

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Filters highlight patterns to identify objects



How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,          # Number of input channels (e.g., RGB has 3)
    out_channels=16,        # Number of output channels (number of filters)
    kernel_size=3,          # Size of the convolutional kernel (3x3)
    stride=1,               # Step size of the convolution
    padding=1               # Zero-padding around the edges
)
```

How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,       # Number of input channels (e.g., RGB has 3)
    out_channels=16,     # Number of output channels (number of filters)
    kernel_size=3,       # Size of the convolutional kernel (3x3)
    stride=1,            # Step size of the convolution
    padding=1            # Zero-padding around the edges
)
```

How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,       # Number of input channels (e.g., RGB has 3)
    out_channels=16,     # Number of output channels (number of filters)
    kernel_size=3,       # Size of the convolutional kernel (3x3)
    stride=1,            # Step size of the convolution
    padding=1            # Zero-padding around the edges
)
```

How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,        # Number of input channels (e.g., RGB has 3)
    out_channels=16,      # Number of output channels (number of filters)
    kernel_size=3,        # Size of the convolutional kernel (3x3)
    stride=1,             # Step size of the convolution
    padding=1             # Zero-padding around the edges
)
```

How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,      # Number of input channels (e.g., RGB has 3)
    out_channels=16,    # Number of output channels (number of filters)
    kernel_size=3,      # Size of the convolutional kernel (3x3)
    stride=1,           # Step size of the convolution
    padding=1           # Zero-padding around the edges
)
```

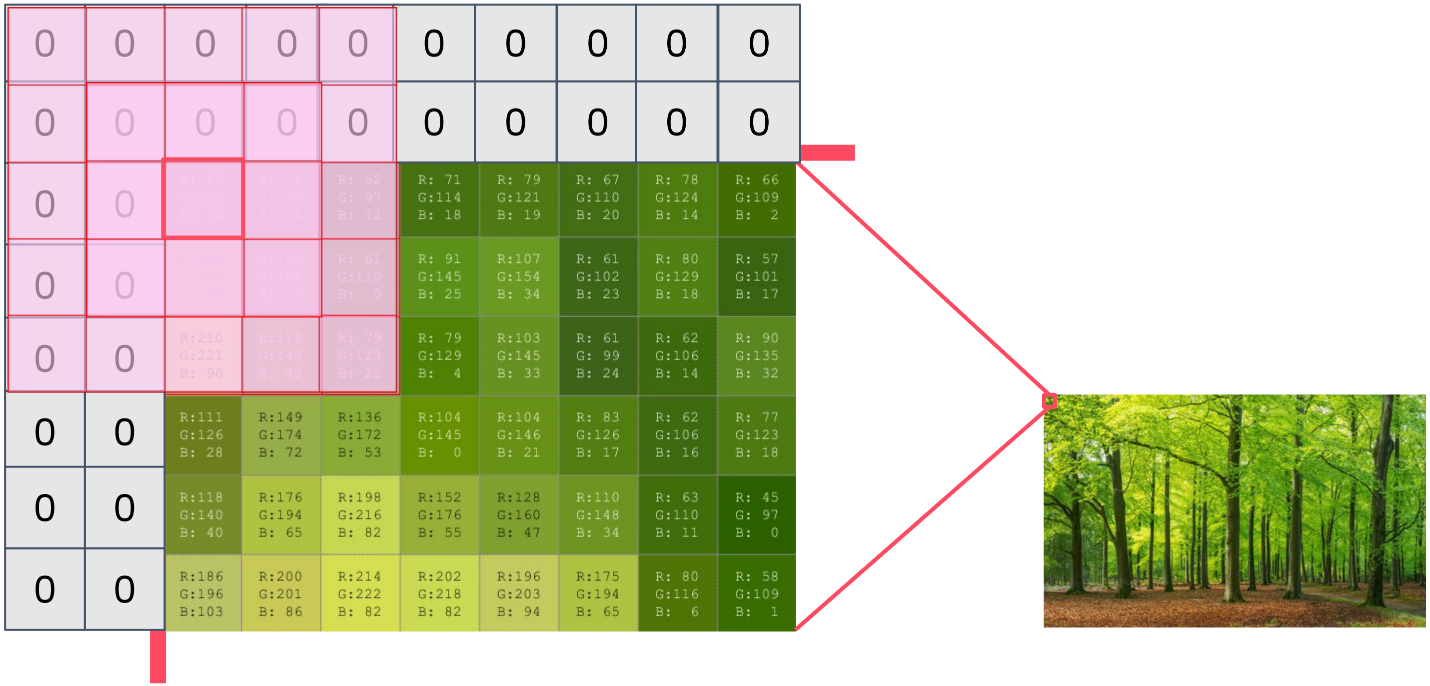
How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,       # Number of input channels (e.g., RGB has 3)
    out_channels=16,     # Number of output channels (number of filters)
    kernel_size=3,       # Size of the convolutional kernel (3x3)
    stride=1,            # Step size of the convolution
    padding=1            # Zero-padding around the edges
)
```

How to create CNNs in PyTorch with nn.Module

```
# Basic convolutional layer
conv_layer = nn.Conv2d(
    in_channels=3,       # Number of input channels (e.g., RGB has 3)
    out_channels=16,     # Number of output channels (number of filters)
    kernel_size=3,       # Size of the convolutional kernel (3x3)
    stride=1,            # Step size of the convolution
    padding=1            # Zero-padding around the edges
)
```

How does padding work?



Laurence Moroney

DeepLearning.AI

Convolutional Neural Networks

Part 2: The Full Architecture

Core Neural Network Components

```python
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

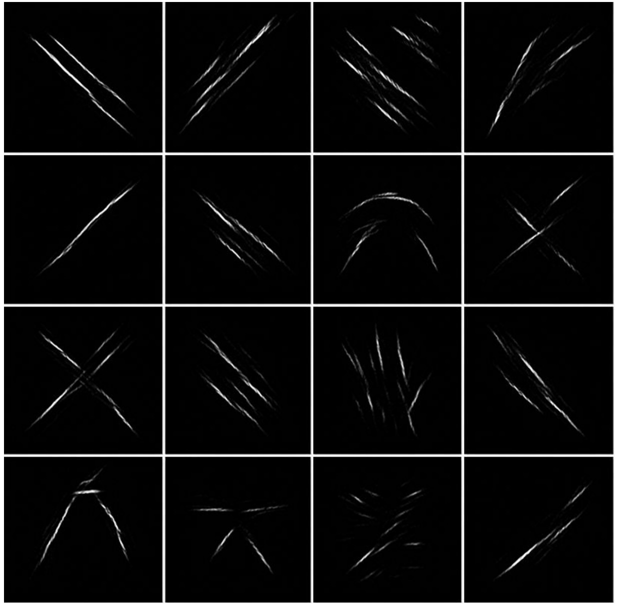
```
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

```
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

```
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

Feature maps or activation maps



0	64	128	128
48	192	144	144
142	226	168	0
255	0	0	64

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

Pooling in action

```
self.pool1 = nn.MaxPool2d(kernel_size=2)
```

0	64	128	128
48	192	144	144
142	226	168	0
255	0	0	64

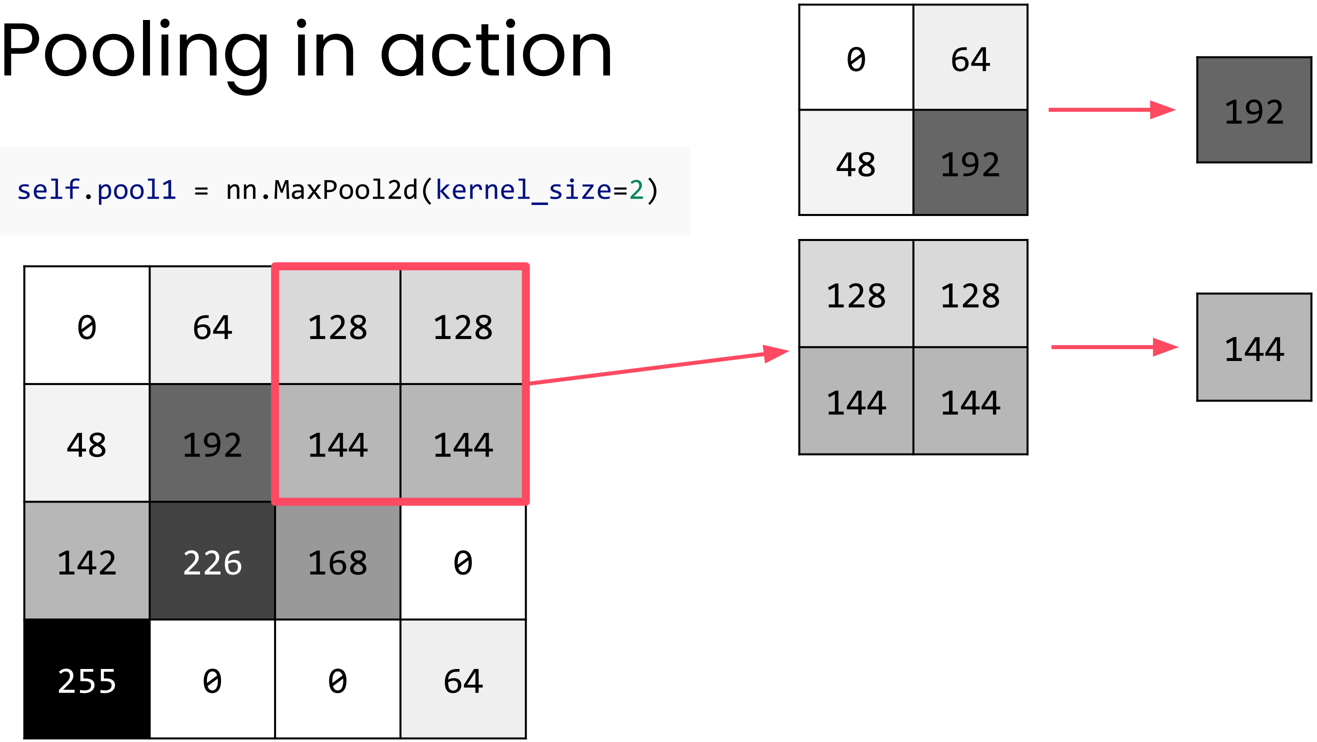
0	64
48	192

192

Laurence Moroney

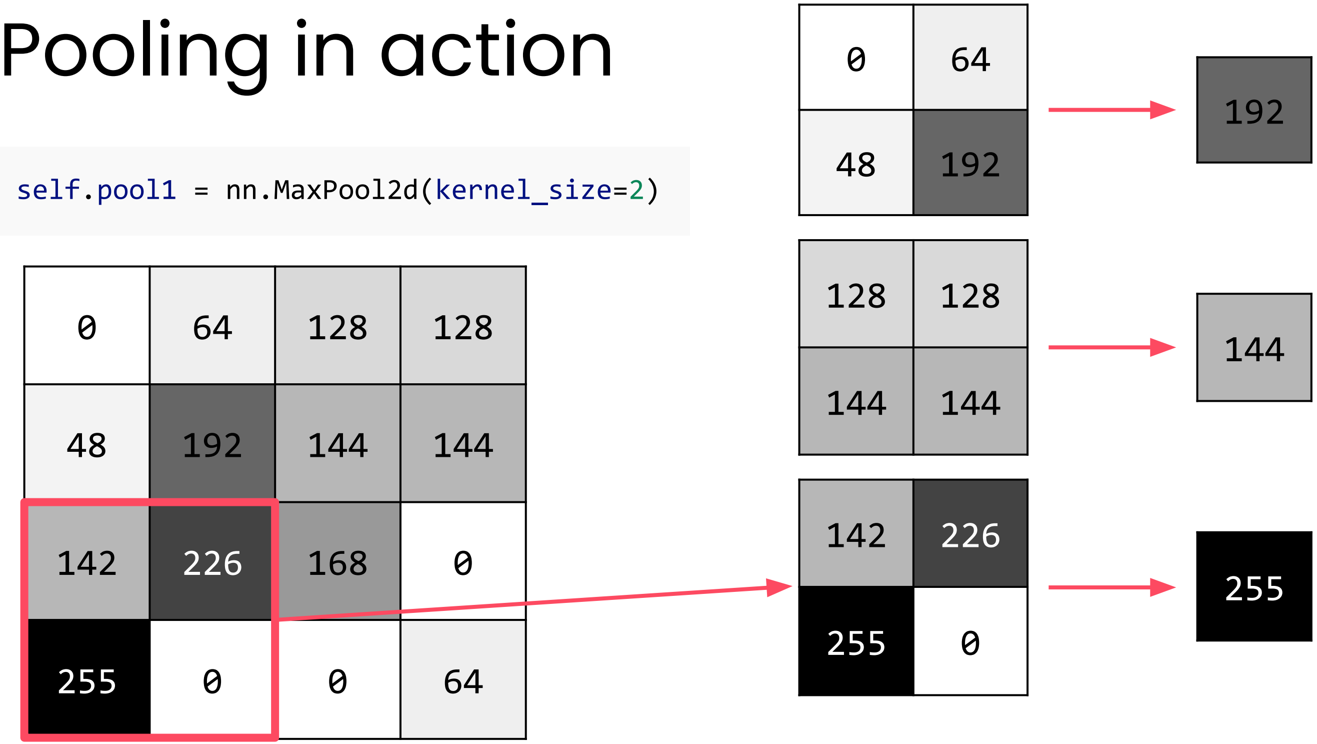
Pooling in action

```
self.pool1 = nn.MaxPool2d(kernel_size=2)
```



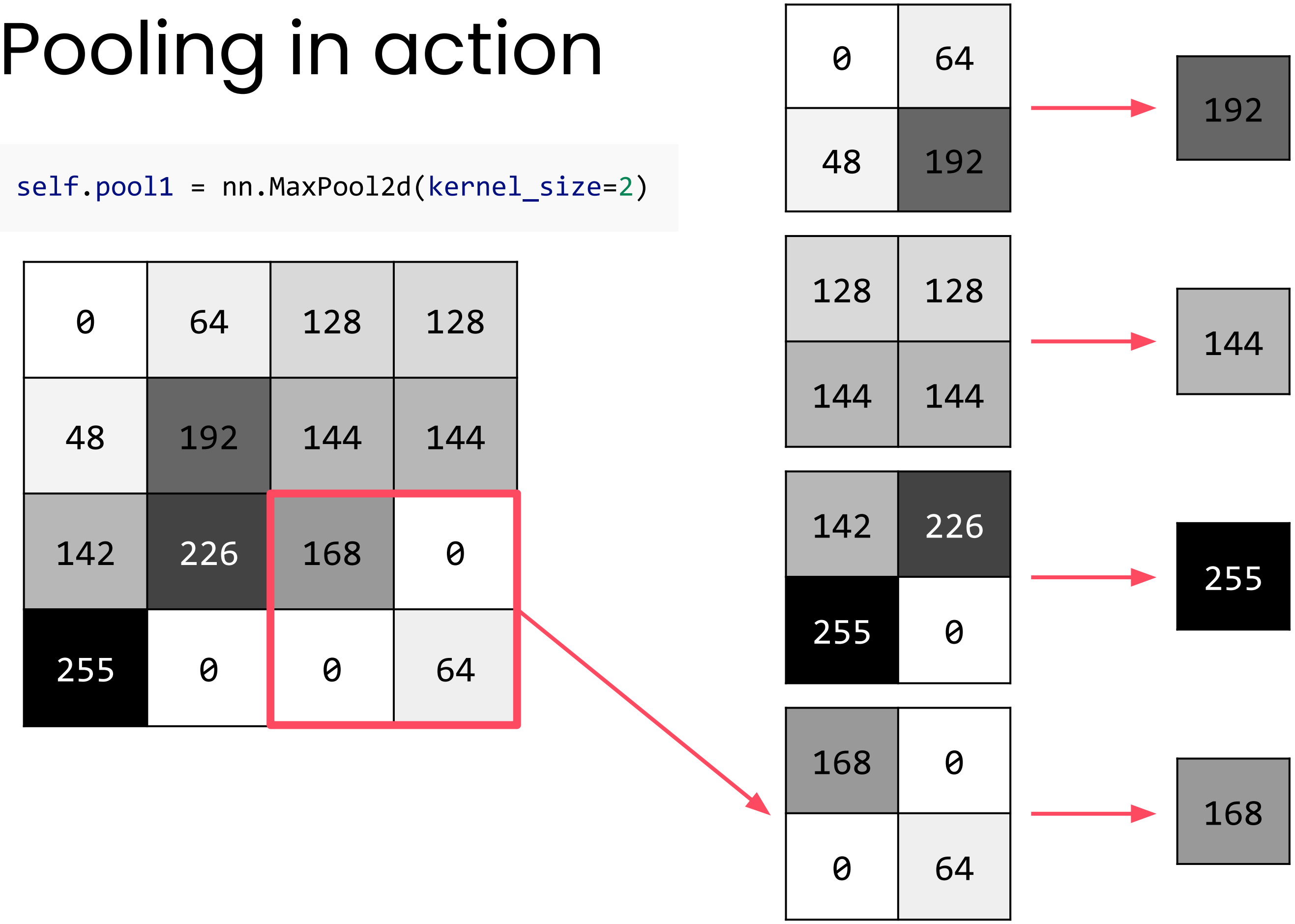
Pooling in action

```
self.pool1 = nn.MaxPool2d(kernel_size=2)
```



Pooling in action

```
self.pool1 = nn.MaxPool2d(kernel_size=2)
```



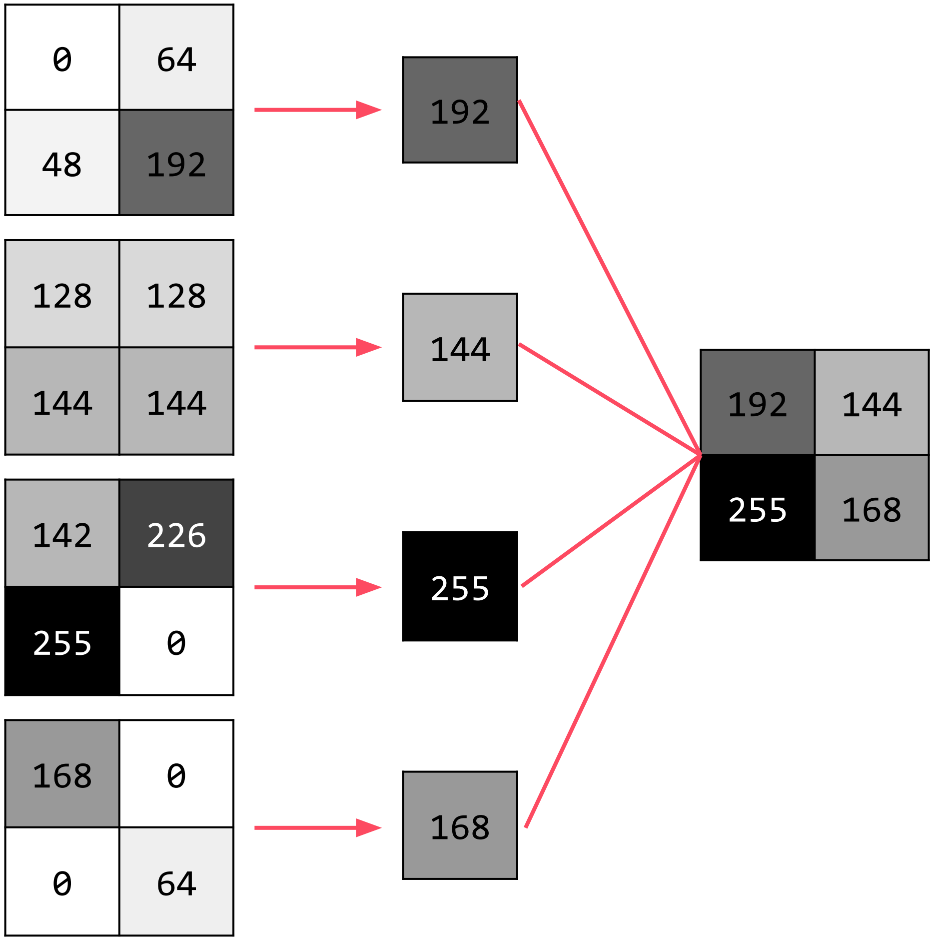
DeepLearning.AI

Laurence Moroney

Pooling in action

```
self.pool1 = nn.MaxPool2d(kernel_size=2)
```

0	64	128	128
48	192	144	144
142	226	168	0
255	0	0	64



```
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```python
class SimpleCNN(nn.Module):
   def __init__(self):
       super(SimpleCNN, self).__init__()
       # First convolutional layer
       self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
       self.relu1 = nn.ReLU()
       self.pool1 = nn.MaxPool2d(kernel_size=2)

       # Second convolutional layer
       self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
       self.relu2 = nn.ReLU()
       self.pool2 = nn.MaxPool2d(kernel_size=2)

       # Flatten layer (no parameters, just reshaping)
       self.flatten = nn.Flatten()

       # Fully connected layer
       self.fc = nn.Linear(64 * 7 * 7, 10)
```

28x28

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

28x28

14x14

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

28x28

14x14

7x7

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

28x28

14x14

7x7

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)

        # Flatten layer (no parameters, just reshaping)
        self.flatten = nn.Flatten()

        # Fully connected layer
        self.fc = nn.Linear(64 * 7 * 7, 10)
```

```
    def forward(self, x):
        # First conv block
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Second conv block
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # Flatten before the fully connected layer
        x = self.flatten(x)

        # Fully connected layer
        x = self.fc(x)
        return x

# Create an instance of our CNN
model = SimpleCNN()
print(model)
```

```python
def forward(self, x):
        # First conv block
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Second conv block
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # Flatten before the fully connected layer
        x = self.flatten(x)

        # Fully connected layer
        x = self.fc(x)
        return x

# Create an instance of our CNN
model = SimpleCNN()
print(model)
```

```python
def forward(self, x):
        # First conv block
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Second conv block
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # Flatten before the fully connected layer
        x = self.flatten(x)

        # Fully connected layer
        x = self.fc(x)
        return x

# Create an instance of our CNN
model = SimpleCNN()
print(model)
```

DeepLearning.AI

Train a CNN for Image Classification

Core Neural Network Components

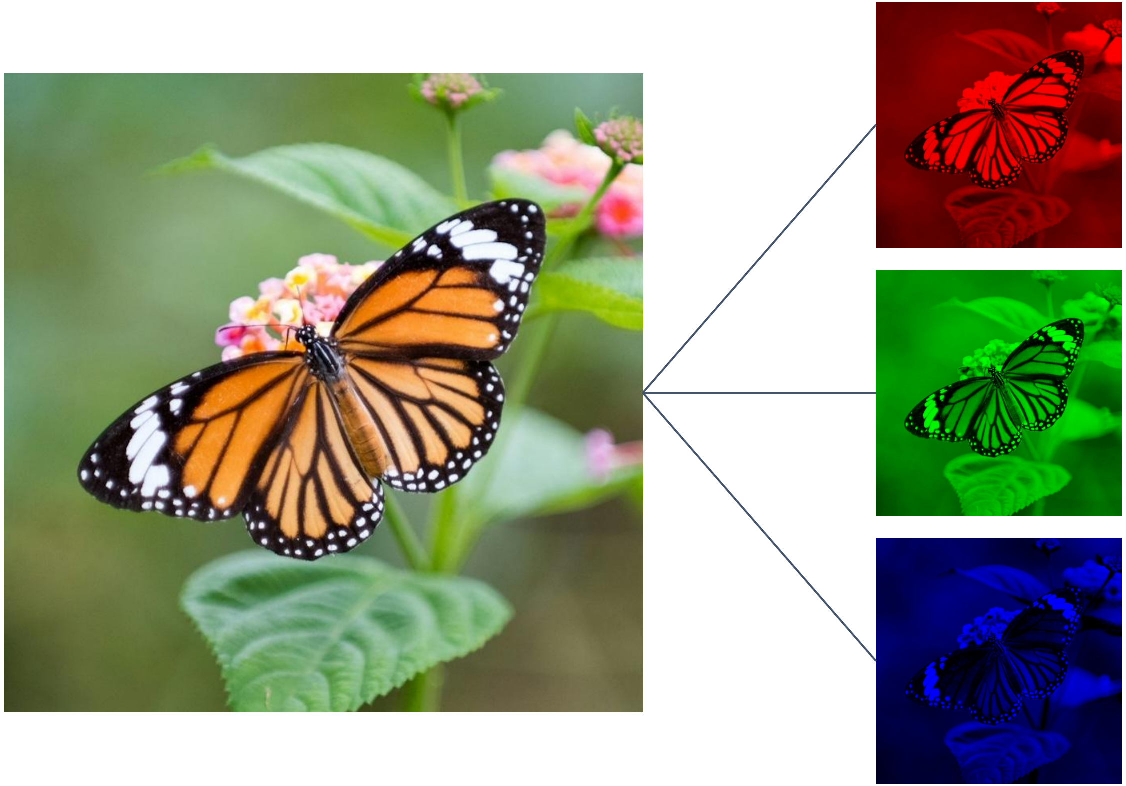
The dataset: 32x32 color images

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

Three input channels: red, green, and blue



DeepLearning.AI

Laurence Moroney

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

32x32

16x16

8x8

4x4

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

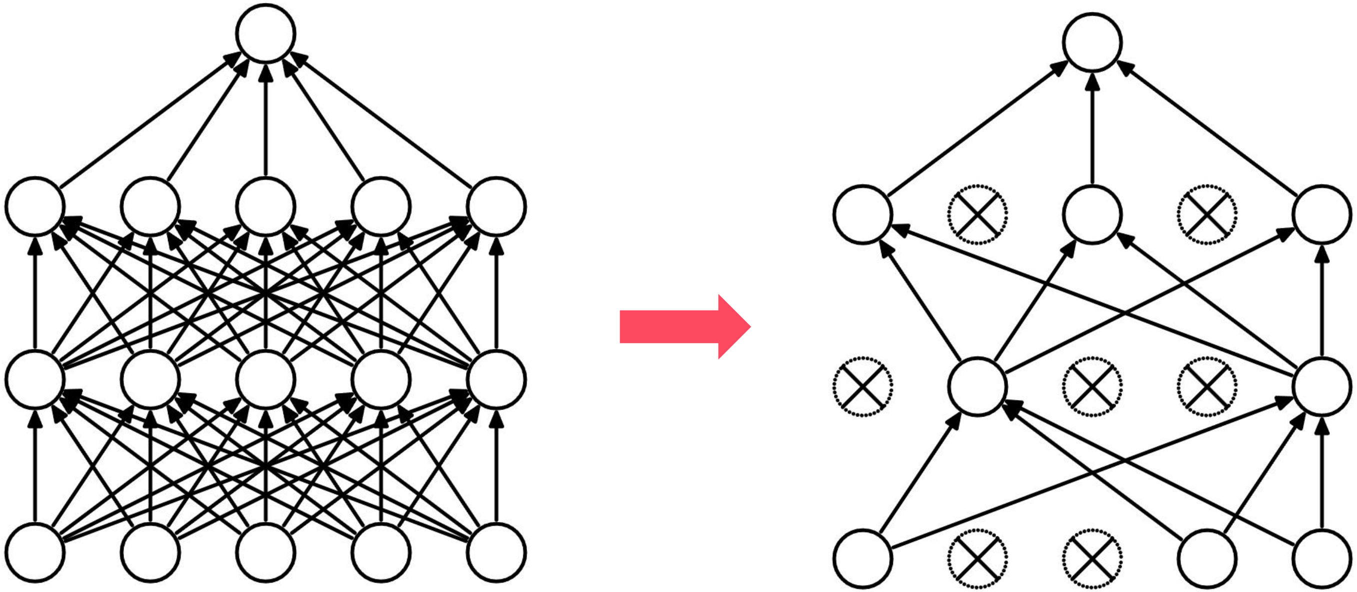
This husky was misclassified as a wolf. Why?

Ribeiro, Singh & Guestrin (2016)

This husky was misclassified as a wolf. Why?

Co-adaptation: neurons learn to rely on shortcuts.

Dropout breaks co-adaptation



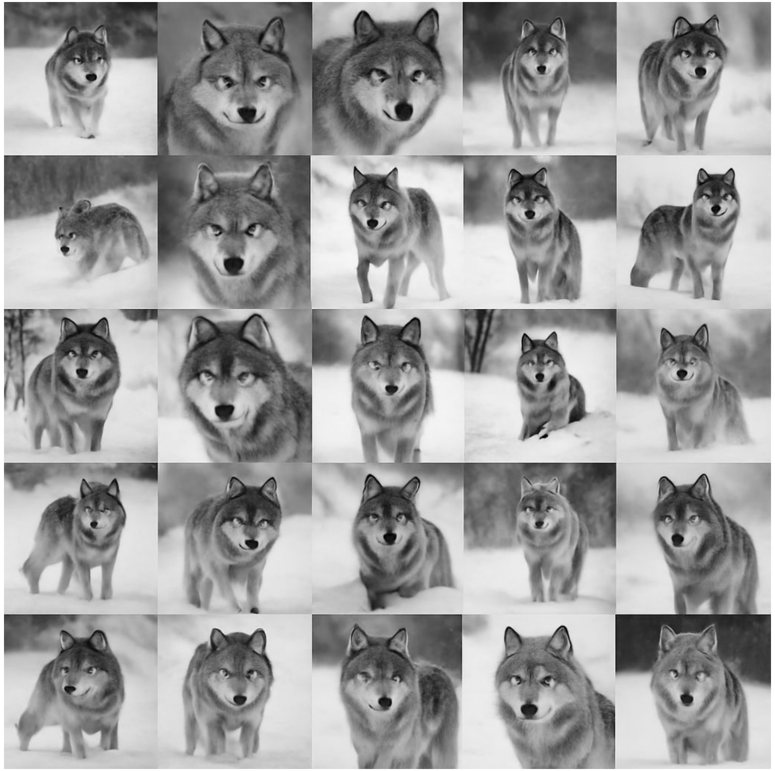
Srivastava et al. (2014)

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

Dataset Problems vs. Co-Adaptation

Dataset Problems vs. Co-Adaptation

Dropout helps learn features that generalize



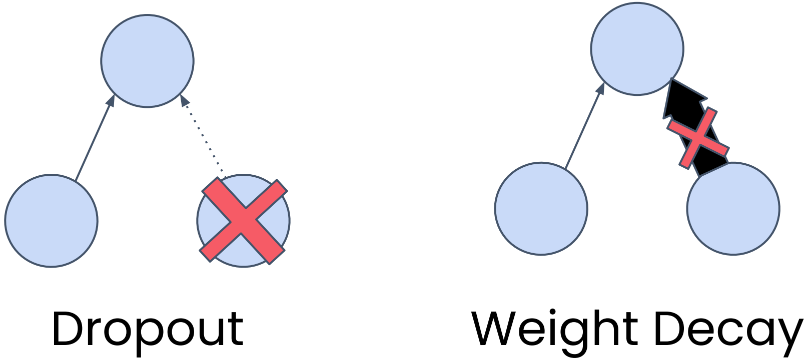
```
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolutional block
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolutional block
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)  # 15 classes in the dataset
```

```python
# Define the loss function
loss_function = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=0.0005)
```

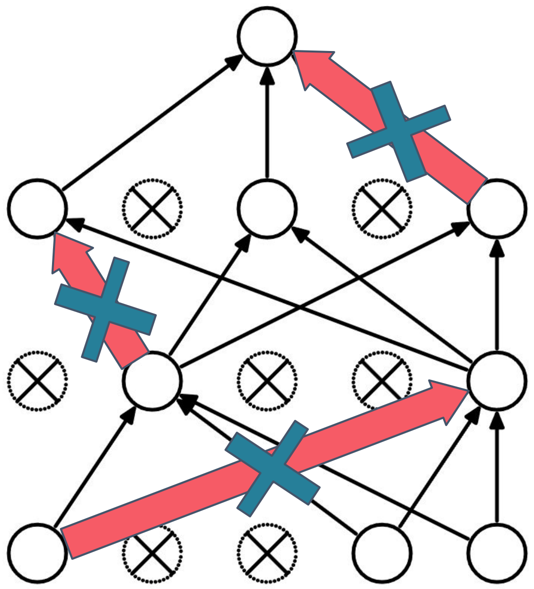
```python
# Define the loss function
loss_function = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=0.0005)
```

Dropout and Weight Decay

Regularization Techniques

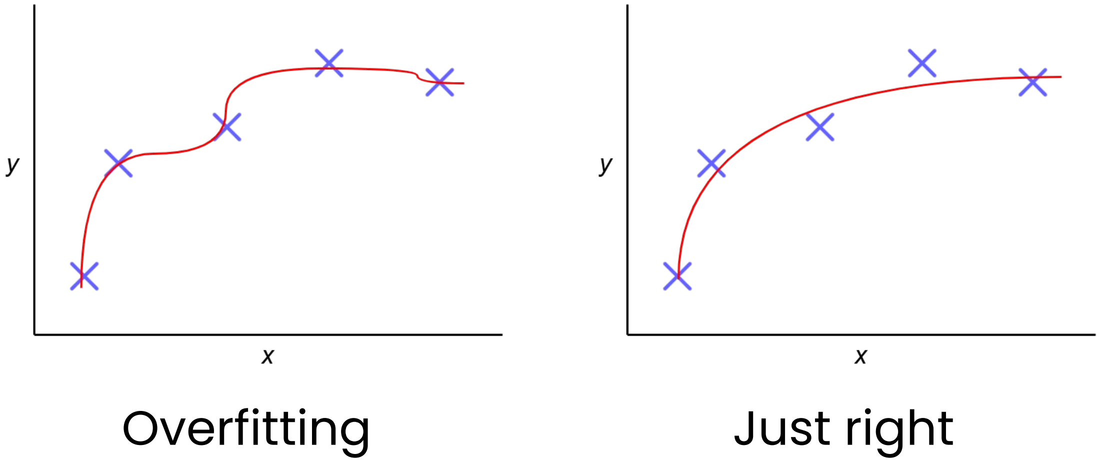


Regularization Techniques



Weight Decay

Penalize large weights



Mahmud (2021)

Model output:

```
C:>
Input shape: torch.Size([1, 3, 32, 32])
After conv1: torch.Size([1, 32, 32, 32])
After pool1: torch.Size([1, 32, 16, 16])
After conv2: torch.Size([1, 64, 16, 16])
After pool2: torch.Size([1, 64, 8, 8])
After conv3: torch.Size([1, 128, 8, 8])
After pool3: torch.Size([1, 128, 4, 4])
After flatten: torch.Size([1, 2048])
After fc1: torch.Size([1, 512])
Output shape: torch.Size([1, 10])
```

Model output:

```
C:>
Input shape: torch.Size([1, 3, 32, 32])
After conv1: torch.Size([1, 32, 32, 32])
After pool1: torch.Size([1, 32, 16, 16])
After conv2: torch.Size([1, 64, 16, 16])
After pool2: torch.Size([1, 64, 8, 8])
After conv3: torch.Size([1, 128, 8, 8])
After pool3: torch.Size([1, 128, 4, 4])
After flatten: torch.Size([1, 2048])
After fc1: torch.Size([1, 512])
Output shape: torch.Size([1, 10])
```

Model output:

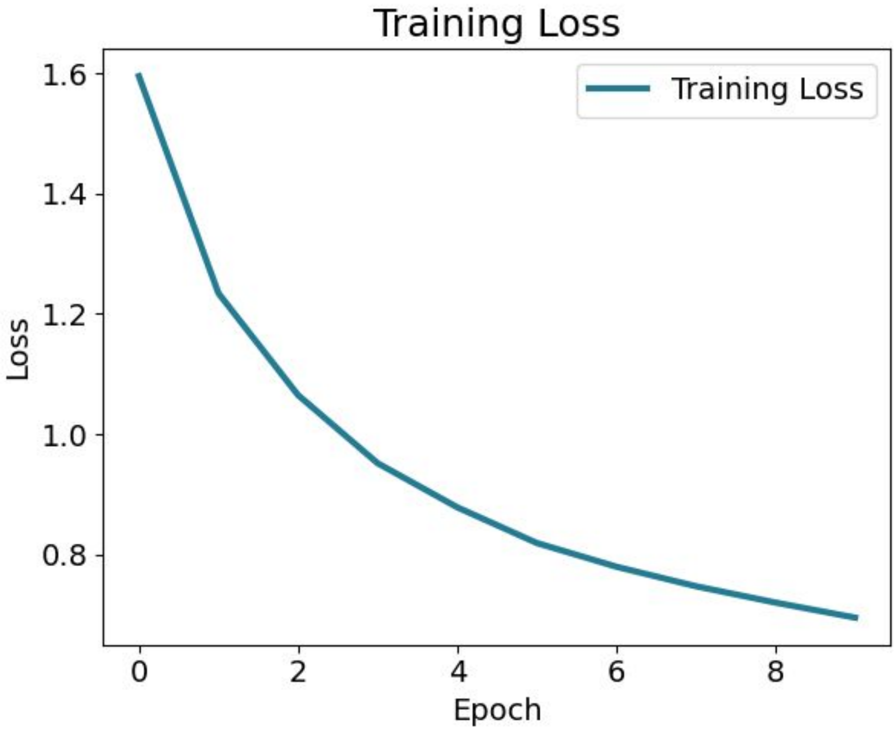
```
C:>
Input shape: torch.Size([1, 3, 32, 32])
After conv1: torch.Size([1, 32, 32, 32])
After pool1: torch.Size([1, 32, 16, 16])
After conv2: torch.Size([1, 64, 16, 16])
After pool2: torch.Size([1, 64, 8, 8])
After conv3: torch.Size([1, 128, 8, 8])
After pool3: torch.Size([1, 128, 4, 4])
After flatten: torch.Size([1, 2048])
After fc1: torch.Size([1, 512])
Output shape: torch.Size([1, 15])
```

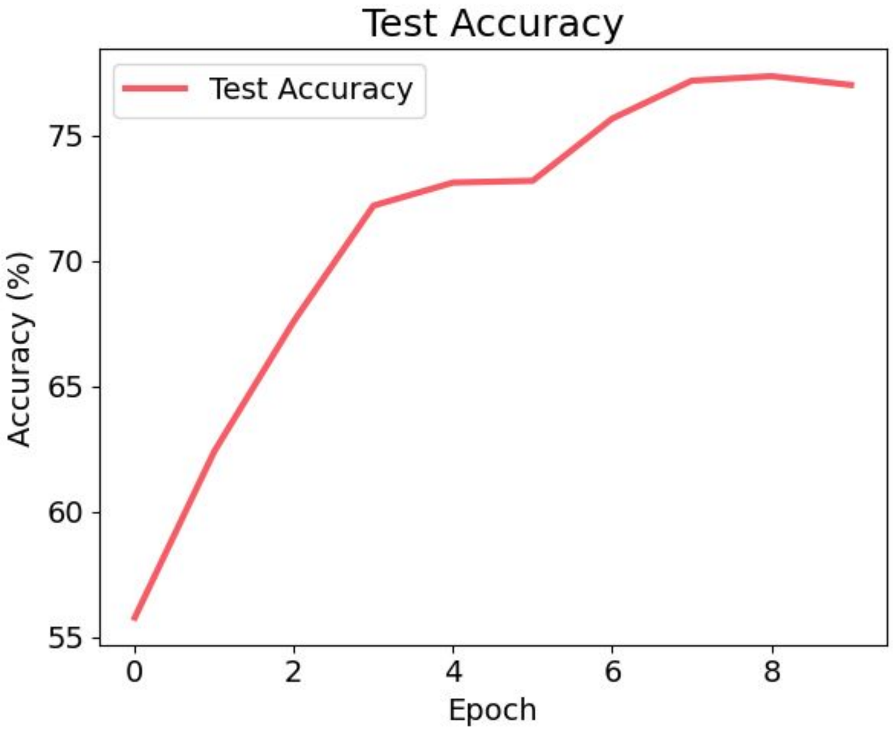
Model output:

```
C:>
Input shape: torch.Size([1, 3, 32, 32])
After conv1: torch.Size([1, 32, 32, 32])
After pool1: torch.Size([1, 32, 16, 16])
After conv2: torch.Size([1, 64, 16, 16])
After pool2: torch.Size([1, 64, 8, 8])
After conv3: torch.Size([1, 128, 8, 8])
After pool3: torch.Size([1, 128, 4, 4])
After flatten: torch.Size([1, 2048])
After fc1: torch.Size([1, 512])
Output shape: torch.Size([1, 15])
```

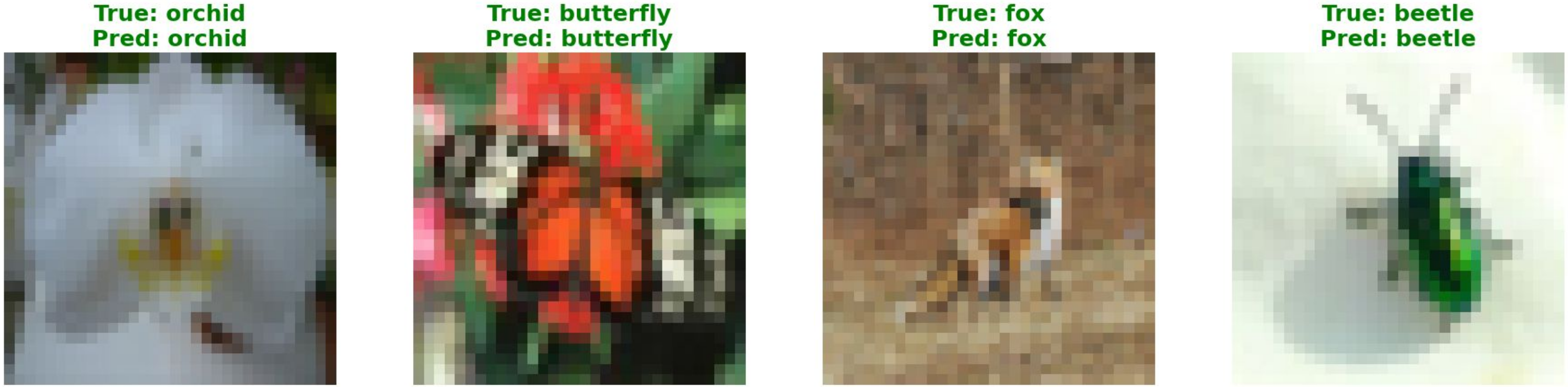

Laurence Moroney

Training and testing results





Model predictions accurate!



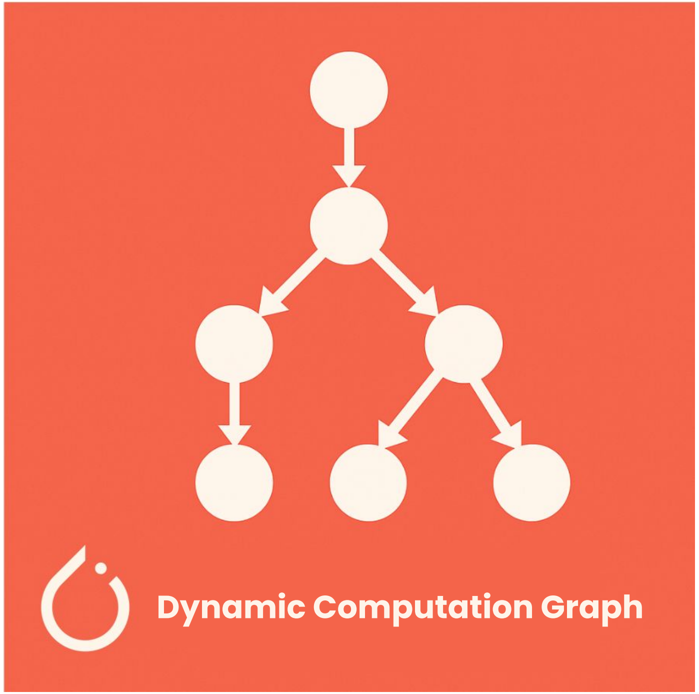
DeepLearning.AI

Dynamic Graphs

Core Neural Network Components

```python
# Define a simple CNN with only Conv2d layers
class SimpleCNN(nn.Module):

    def forward(self, x):
        # First convolutional block
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        # Second convolutional block
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        # Third convolutional block
        x = self.conv3(x)
        x = self.relu3(x)
        x = self.pool3(x)

        # Fully connected layers
        # Input image is 32x32, after 3 pooling layers: 4x4
        x = self.fc1(x)
        x = self.relu4(x)
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

Dynamic Computation Graph



Early Frameworks

```
# A chain of operations, one feeding into the next
result1 = do_math(input)
result2 = do_math(result1)
result3 = do_math(result2)
result4 = do_math(result3)
result5 = do_math(result4)
# ... and so on, down the line
```

nn.Sequential

```
model = nn.Sequential(
    nn.Conv2d(3, 32, 3),
    nn.ReLU(),
    nn.Conv2d(32, 64, 3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 26 * 26, 10)
)
```

~~for a in b:~~

~~if a:~~

~~print()~~

What is a computation graph?

```
model = nn.Sequential(
    nn.Conv2d(3, 32, 3),
    nn.ReLU(),
    nn.Conv2d(32, 64, 3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 26 * 26, 10)
)
```

`nn.Conv2d()`

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

What is a computation graph?

```
model = nn.Sequential(
    nn.Conv2d(3, 32, 3),
    nn.ReLU(),
    nn.Conv2d(32, 64, 3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 26 * 26, 10)
)
```

`nn.Conv2d()`

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

`nn.ReLU()`

$$ReLU(x) = (x)^+ = max(0, x)$$

What is a computation graph?

```python
model = nn.Sequential(
    nn.Conv2d(3, 32, 3),
    nn.ReLU(),
    nn.Conv2d(32, 64, 3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 26 * 26, 10)
)
```

`nn.Conv2d()`

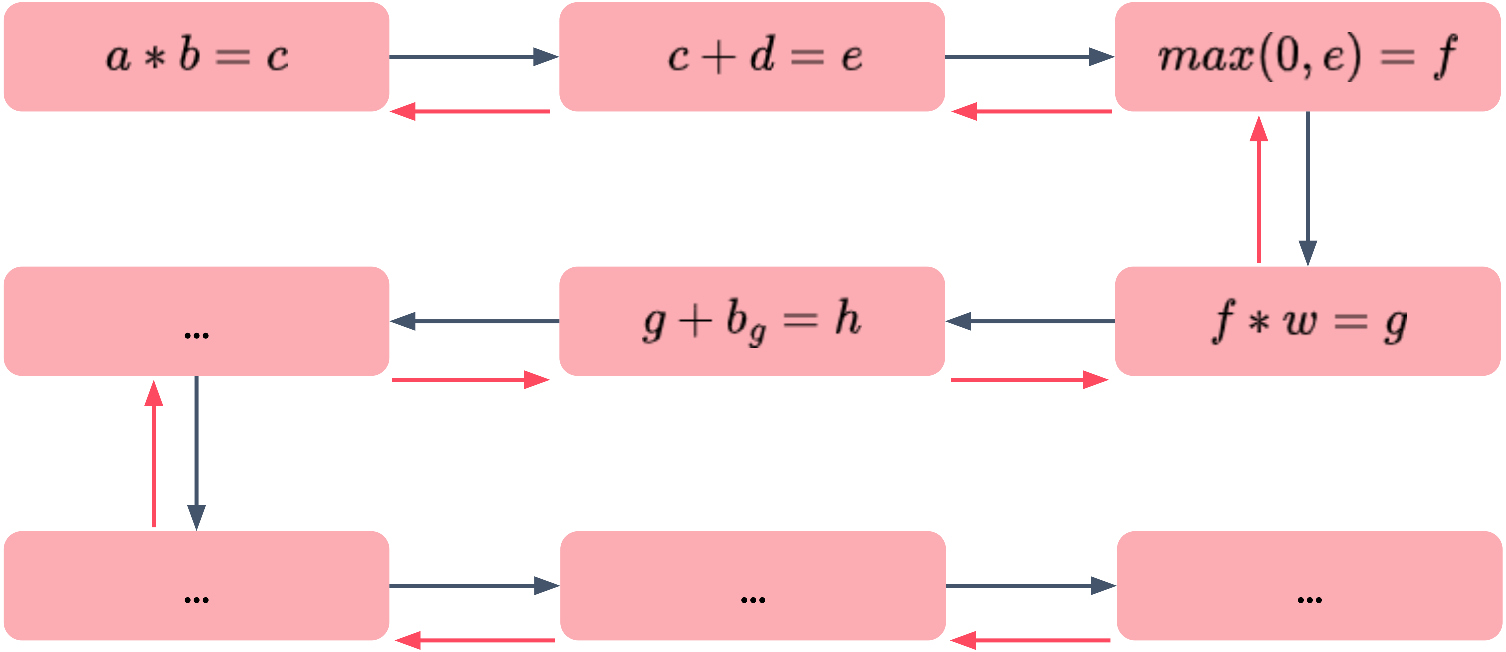
$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

$\downarrow$

`nn.ReLU()`

$$ReLU(x) = (x)^+ = max(0, x)$$

What is a computation graph?



`nn.Sequential` constraints:

- Every operation locked in order.
- No print statements to debug intermediate values.
- No if statements that conditionally branch.
- No loops that adjust to your data.

`nn.Sequential` tradeoffs:

✔

- Knows all operations ahead of time
- Can optimize aggressively

✘

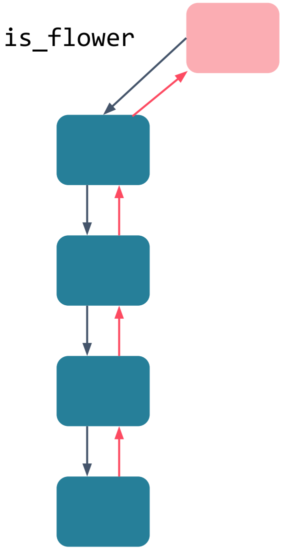
- Lose Python's power
- No debugging with print statements
- No conditional logic
- No experimentation on the fly

Dynamic Graphs

```
def forward(self, x):
    if self.is_flower(x):
        return self.flower_layers(x)
    else:
        return self.butterfly_layers(x)
```

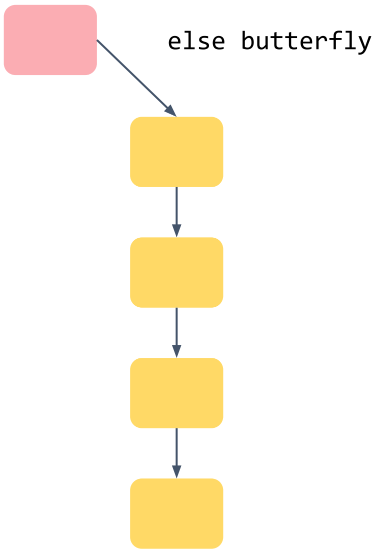
Dynamic Graphs

```python
def forward(self, x):
    if self.is_flower(x):
        return self.flower_layers(x)
    else:
        return self.butterfly_layers(x)
```



Dynamic Graphs

```
def forward(self, x):
    if self.is_flower(x):
        return self.flower_layers(x)
    else:
        return self.butterfly_layers(x)
```



DeepLearning.AI

Laurence Moroney

Static vs. Dynamic Frameworks

Static

Spell out every possible execution path.

Dynamic

Adapt as you go.

Dynamic Graphs

```python
import torch.nn as nn

class MyModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)


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

Defines Model

Defines Flow of Dynamic Graph

Flexible Input Shapes

```
def forward(self, sentences):

    # Graph builds on the fly - no max length required
    for sentence in sentences:
        # sentence might be 3 words
        # sentence might be 50 words
        process(sentence)  # Each is processed dynamically
```

Debugging in PyTorch is like regular Python

```
def forward(self, x):
    x = self.conv1(x)

    if x.std() < 0.1:
        print(f"Problem detected! Variance: {x.std()}")

    x = self.conv2(x)
    return x
```

Adaptive Processing

```
def forward(self, x):

    if is_easy_case(x):
        # Easy image - use small network (2 layers)
        return self.small_network(x)
    else:
        # Hard image - use big network (50 layers)
        return self.full_network(x)
```

Adaptive Processing

```python
def forward(self, x):

    if is_easy_case(x):
        # Easy image - use small network (2 layers)
        return self.small_network(x)
    else:
        # Hard image - use big network (50 layers)
        return self.full_network(x)
```

DeepLearning.AI

Modular Architectures

Core Neural Network Components

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

        # Block 1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Block 2
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Block 3
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

`__init__`

Defines Architecture

`forward`

Defines Flow

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Block 4
    self.conv4 = nn.Conv2d(...)
    self.relu4 = nn.ReLU()
    self.pool4 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Block 4
    self.conv4 = nn.Conv2d(...)
    self.relu4 = nn.ReLU()
    self.pool4 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Block 4
    self.conv4 = nn.Conv2d(...)
    self.relu5 = nn.ReLU()
    self.pool4 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Block 4
    self.conv4 = nn.Conv2d(...)
    self.relu5 = nn.ReLU()
    self.pool4 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)

    # Block 4
    x = self.conv4(x)
    x = self.relu5(x)    # Or was it 4?
    x = self.pool4(x)
```

```python
def __init__(self):
    super().__init__()
    # Block 1
    self.conv1 = nn.Conv2d(...)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(...)

    # Block 2
    self.conv2 = nn.Conv2d(...)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(...)

    # Block 3
    self.conv3 = nn.Conv2d(...)
    self.relu3 = nn.ReLU()
    self.pool3 = nn.MaxPool2d(...)

    # Block 4
    self.conv4 = nn.Conv2d(...)
    self.relu5 = nn.ReLU()
    self.pool4 = nn.MaxPool2d(...)

    # Fully connected layers
    self.fc1 = nn.Linear(128 * 4 * 4, 512)
    self.relu4 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)
    self.fc2 = nn.Linear(512, 10)
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)

    # Block 4
    x = self.conv4(x)
    x = self.relu5(x)
    x = self.pool4(x)
```

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
def forward(self, x):

    # Block 1
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    # Block 2
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)

    # Block 3
    x = self.conv3(x)
    x = self.relu3(x)
    x = self.pool3(x)

    # Block 4
    x = self.conv4(x)
    x = self.relu5(x)
    x = self.pool4(x)
```

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
def forward(self, x):

    x = self.block1(x)
    x = self.block2(x)
```

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
def forward(self, x):

    x = self.block1(x)
    x = self.block2(x)
```


Laurence Moroney

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))



    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
def forward(self, x):

    x = self.block1(x)
    x = self.block2(x)
```

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

No branching

No looping

One input / One output

```python
def forward(self, x):

    x = self.block1(x)
    y = self.block2(x)

    return x, y
```

```
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
def __init__(self):
    super().__init__()

    self.block1 = nn.Sequential(
        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...),

        nn.Conv2d(...),
        nn.ReLU(),
        nn.MaxPool2d(...))


    self.block2 = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))
```

```python
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

    def forward(self, x):
        return self.block(x)
```

```python
def __init__(self):
    super().__init__()

    self.features = nn.Sequential(
        ConvBlock(3, 32),
        ConvBlock(32, 64),
        ConvBlock(64, 128))

    self.classifier = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))

def forward(self, x):
    x = self.features(x)
    return self.classifier(x)
```

```python
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

    def forward(self, x):
        return self.block(x)
```

```python
def __init__(self):
    super().__init__()

    self.features = nn.Sequential(
        ConvBlock(3, 32),
        ConvBlock(32, 64),
        ConvBlock(64, 128),
        ConvBlock(128, 256),)

    self.classifier = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))

def forward(self, x):
    x = self.features(x)
    return self.classifier(x)
```

```python
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

    def forward(self, x):
        return self.block(x)
```

```python
def __init__(self):
    super().__init__()

    self.features = nn.Sequential(
        ConvBlock(3, 32),
        ConvBlock(32, 64),
        ConvBlock(64, 128),
        ConvBlock(128, 256),)

    self.classifier = nn.Sequential(
        nn.Linear(128 * 4 * 4, 512),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(512, 10))

def forward(self, x):
    x = self.features(x)
    return self.classifier(x)
```

```python
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.block = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

    def forward(self, x):
        return self.block(x)
```

Workflow Tips

- Start explicit - write it all out
- Look for patterns
- Refactor

DeepLearning.AI

Model Inspecting and Debugging

Core Neural Network Components

How do you see what's in your model?

```
print(model)
```

Output:

```
SimpleCNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

What’s missing?

- How many parameters does each layer have?
- What shapes are the tensors?
- What’s actually inside those Sequential blocks?

How many parameters does your model have?

```
print(model.parameters())
```

Output:

```
<generator object Module.parameters at 0x7f8b8c0a5f90>
```

How many parameters does your model have?

```
for param in model.parameters():
    print(param.shape)
```

Output:

```
torch.Size([32, 3, 3, 3])
torch.Size([32])
torch.Size([64, 32, 3, 3])
torch.Size([64])
...
```

How many parameters does your model have?

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

Output:

```
Total parameters: 1147466
```

How can you see inside each layer?

```
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")
```

How can you see inside each layer?

```python
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")
```

```
conv1.weight: torch.Size([32, 3, 3, 3])
conv1.bias: torch.Size([32])
conv2.weight: torch.Size([64, 32, 3, 3])
conv2.bias: torch.Size([64])
conv3.weight: torch.Size([128, 64, 3, 3])
conv3.bias: torch.Size([128])
fc1.weight: torch.Size([512, 2048])
fc1.bias: torch.Size([512])
fc2.weight: torch.Size([10, 512])
fc2.bias: torch.Size([10])
...
```

How can you see inside each layer?

```
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")
```

```
conv1.weight: torch.Size([32, 3, 3, 3])
conv1.bias: torch.Size([32])
conv2.weight: torch.Size([64, 32, 3, 3])
conv2.bias: torch.Size([64])
conv3.weight: torch.Size([128, 64, 3, 3])
conv3.bias: torch.Size([128])
fc1.weight: torch.Size([512, 2048])
fc1.bias: torch.Size([512])
fc2.weight: torch.Size([10, 512])
fc2.bias: torch.Size([10])
...
```

Methods for exploring inside nested blocks:

```
for name, module in model.named_children():
    print(name, module)
```

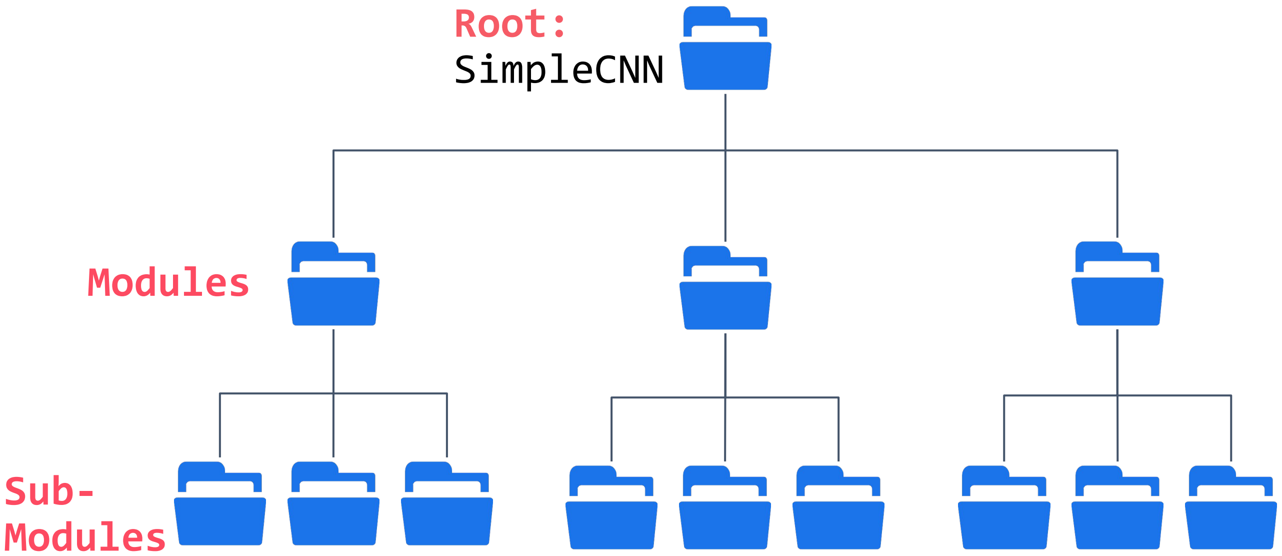
```
for name, module in model.named_modules():
    if name:  # Skip the model itself
        print(name)
```

Methods for exploring inside nested blocks:

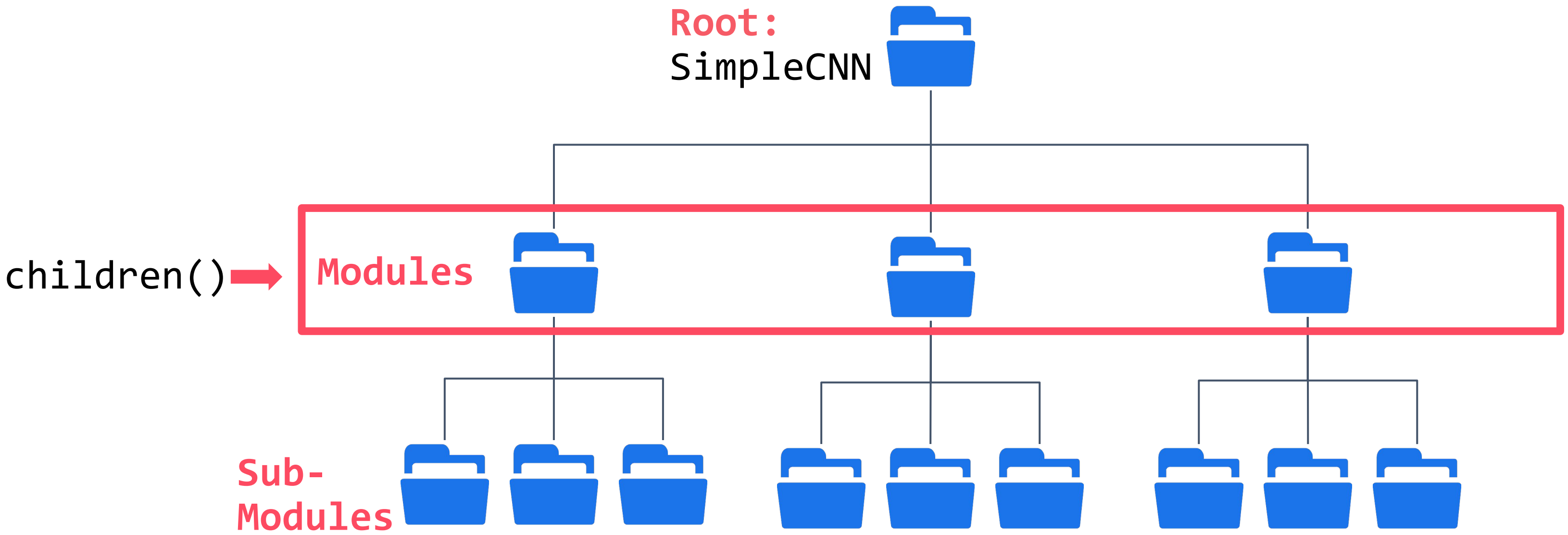
```
for name, module in model.named_children():
    print(name, module)
```

```
for name, module in model.named_modules():
    if name:  # Skip the model itself
        print(name)
```

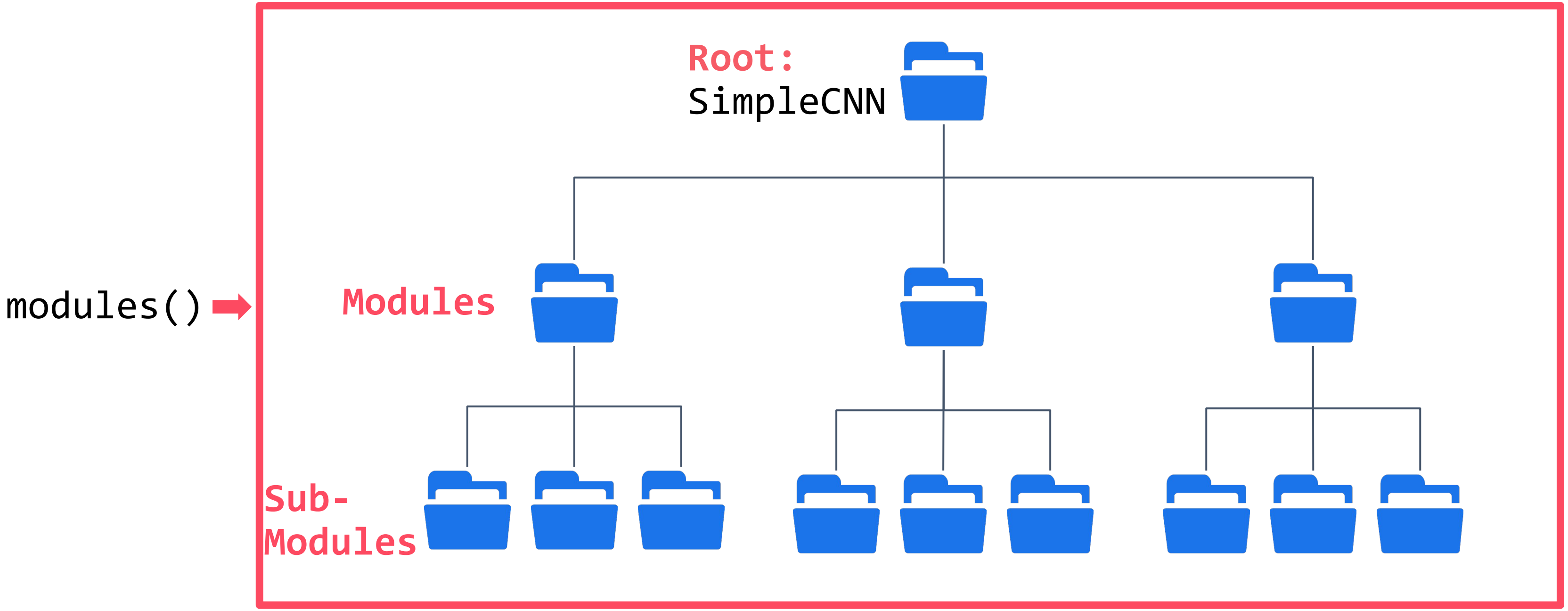
Module Organization



Module Organization



Module Organization



What happens when something goes wrong?

Error:

```
RuntimeError: mat1 and mat2 shapes cannot be multiplied (32x2048 and 1024x512)
```

Print the weight shape:

```
print(model.fc1.weight.shape)

> fc1.weight.shape: torch.Size([512, 1024])
```

What happens when something goes wrong?

```
for name, param in model.named_parameters():
    if 'fc1.weight' in name:
        print(f"{name}: {param.shape}")
```

```
fc1.weight: torch.Size([512, 1024])
```

```
RuntimeError: mat1 and mat2 shapes cannot be multiplied (32x2048 and 1024x512)
```

Trace the shape through your model:

```
def forward(self, x):
    print(f"Input: {x.shape}")
    x = self.features(x)
    print(f"After features: {x.shape}")
    x = x.flatten(1)
    print(f"Flattened: {x.shape}")
    x = self.classifier(x)
    return x
```

Trace the shape through your model:

```
def forward(self, x):
    print(f"Input: {x.shape}")
    x = self.features(x)
    print(f"After features: {x.shape}")
    x = x.flatten(1)
    print(f"Flattened: {x.shape}")
    x = self.classifier(x)
    return x
```

You’ve now mastered the basics of PyTorch

- Creating data pipelines
- Building and training models
- Evaluating performance
- Learning to inspect what’s under the hood

What’s next?

- Using advanced PyTorch tools.
- Working with visual data using Torchvision.
- Explore text models.
- Optimizing hyperparameters
- Building real-world machine learning pipelines.