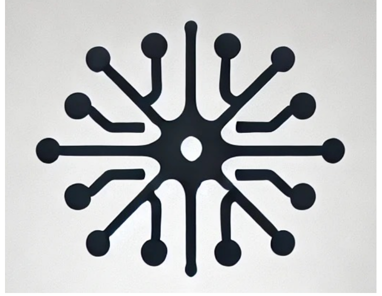
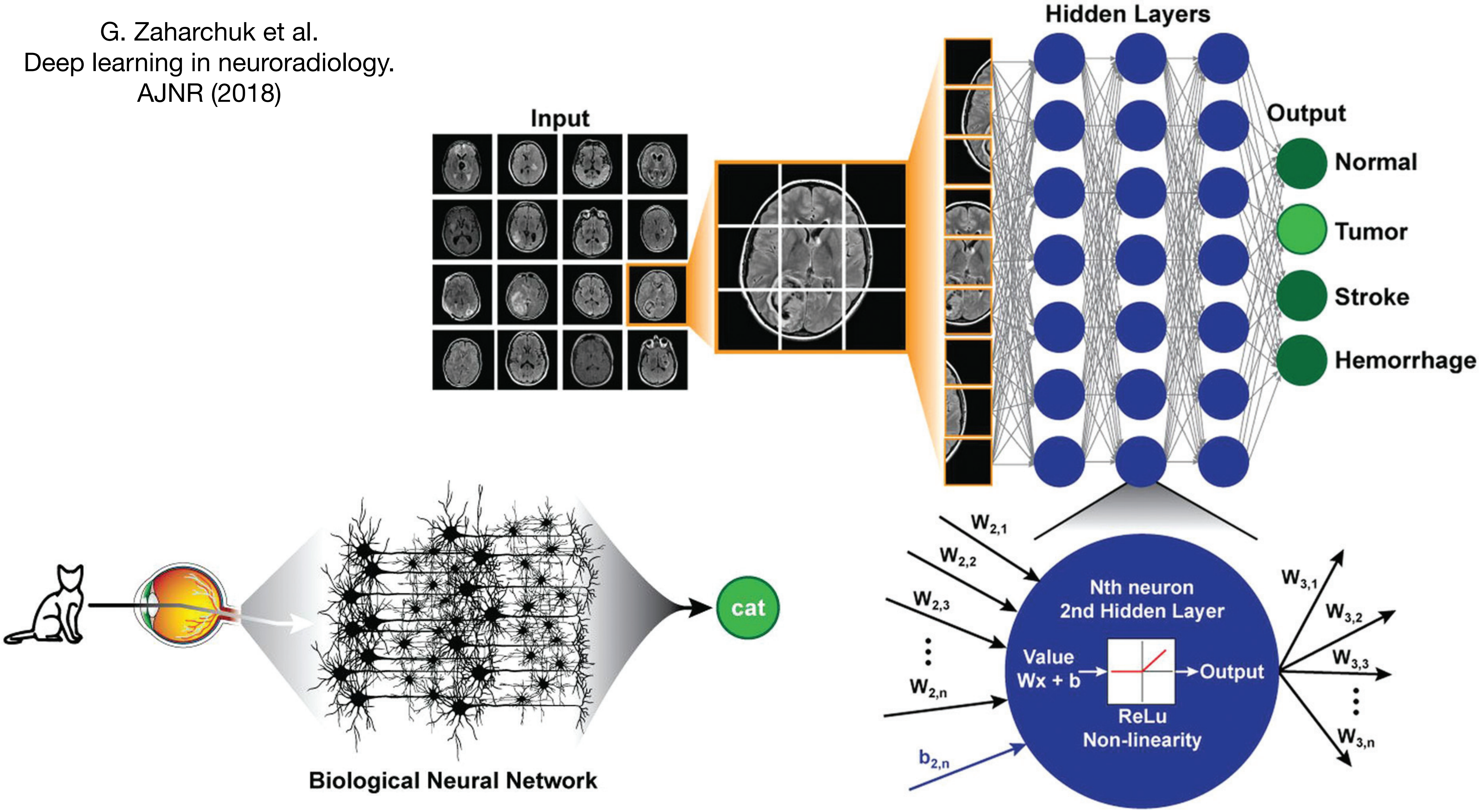


3. Convolutional Neural Network

“Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence.

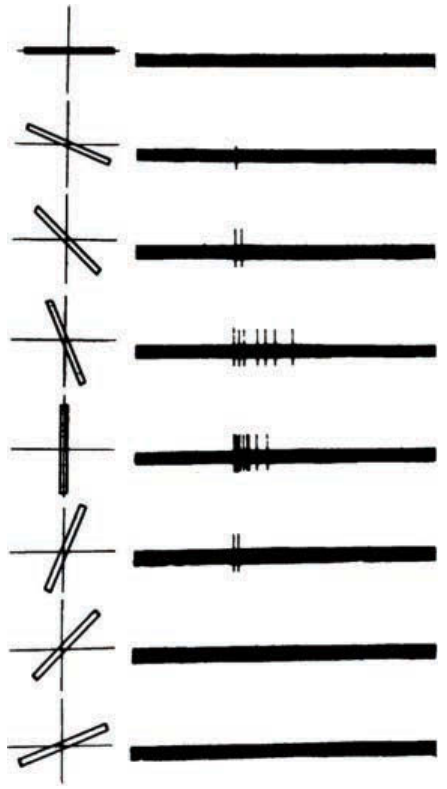
Some of the key design principles of neural networks were drawn from neuroscience.” - Ian Goodfellow

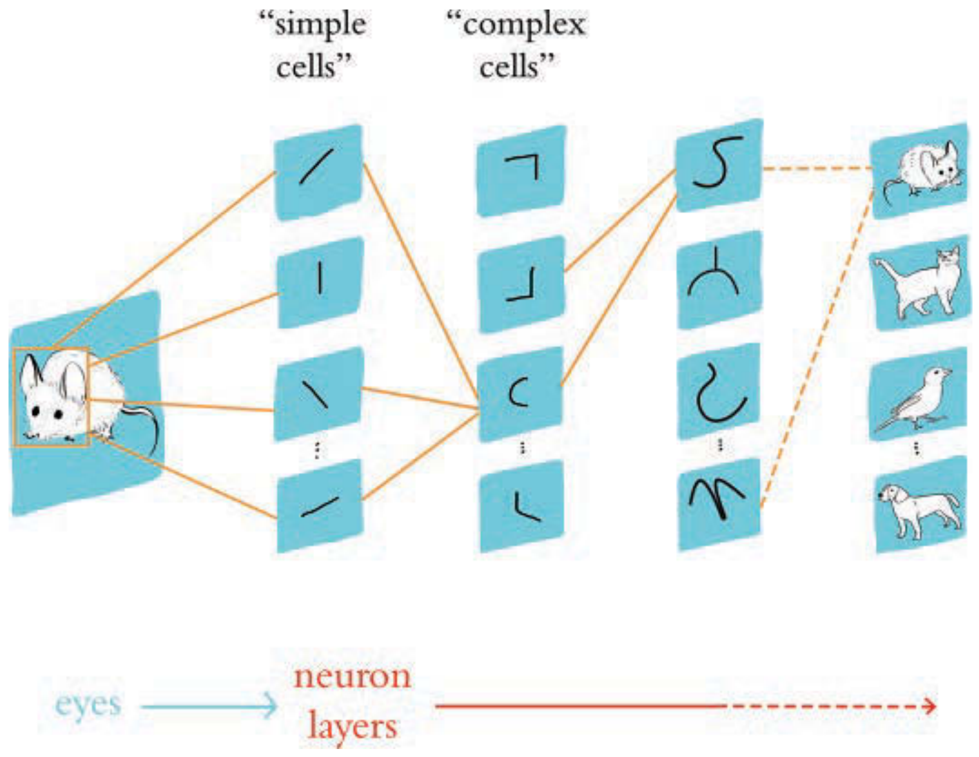
G. Zaharchuk et al.
Deep learning in neuroradiology.
AJNR (2018)

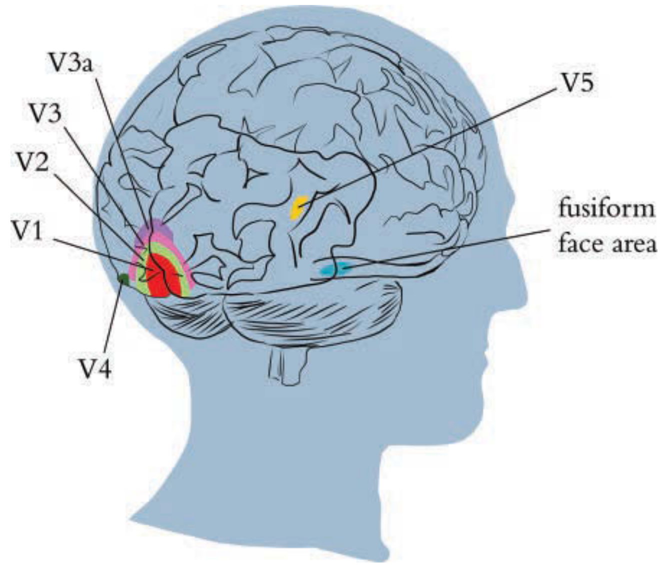


Biological Vision & Hierarchical feature extraction

Krohn Ch1

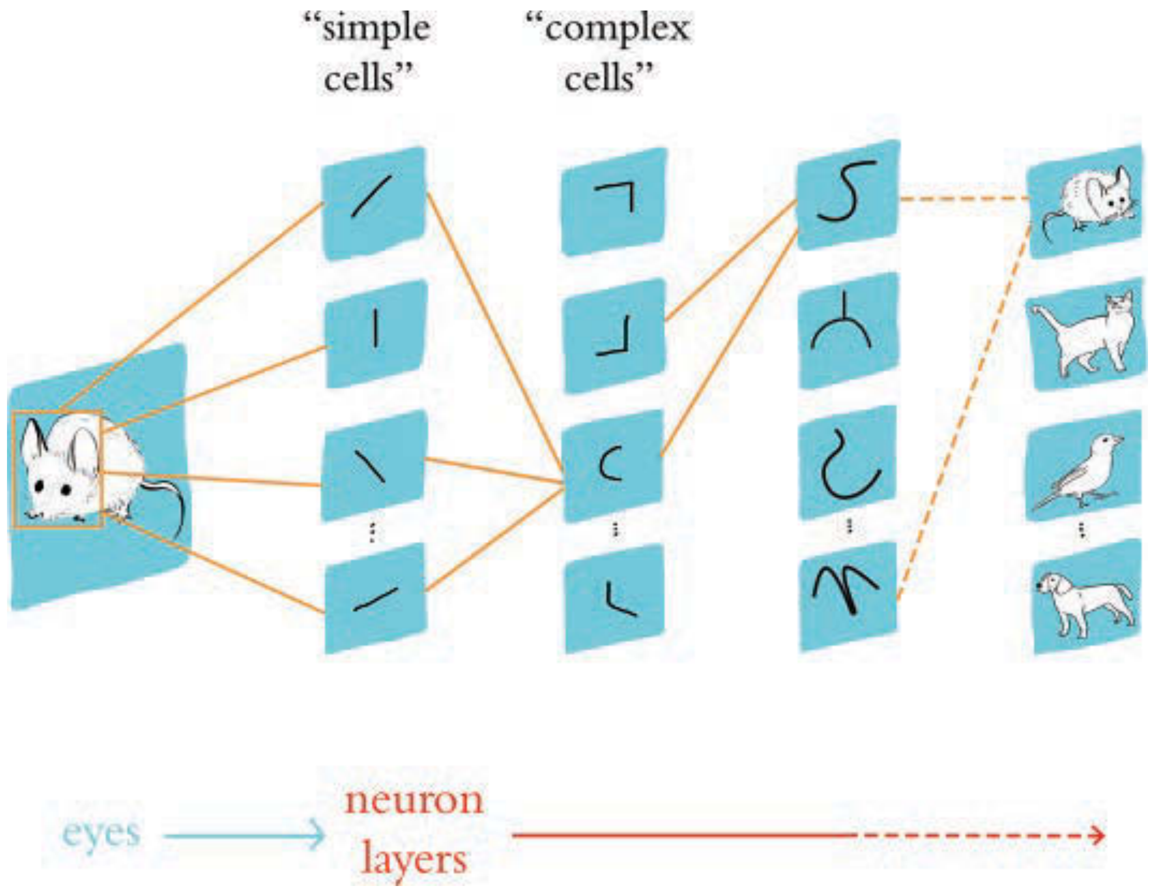


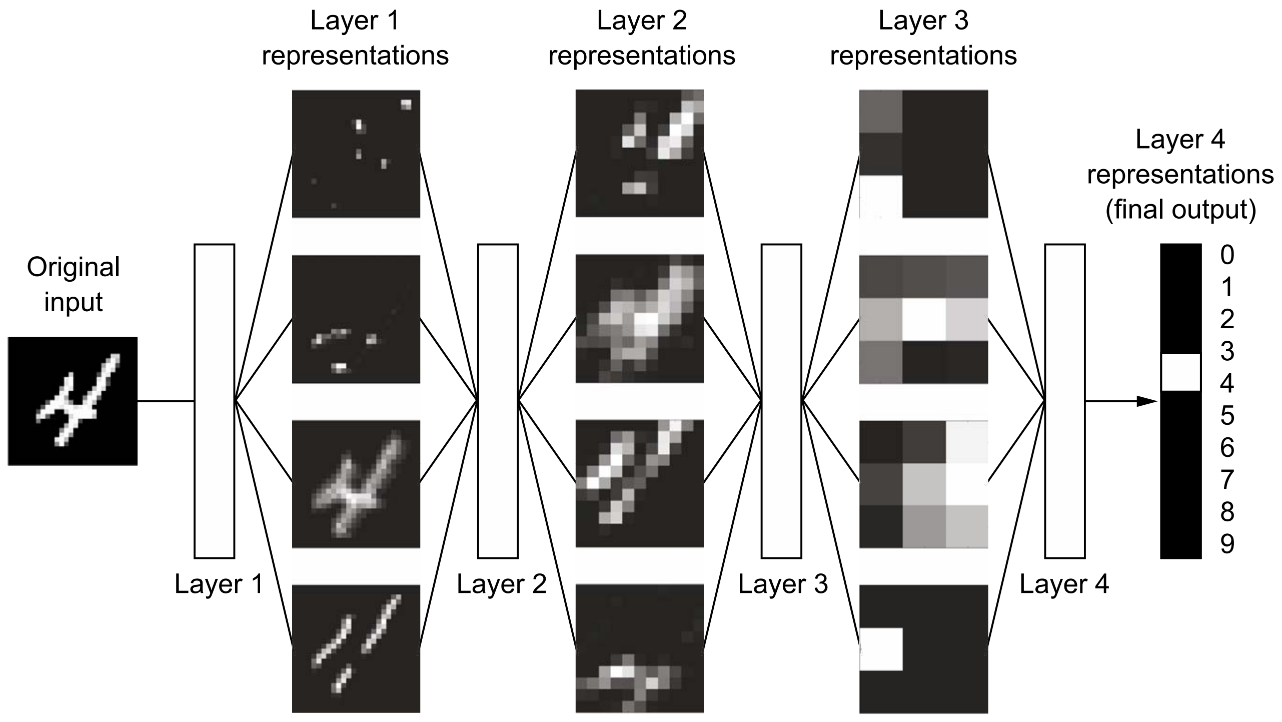




- A simple cell in the visual cortex fires at different rates depending on the orientation of a line.
- Hierarchical representations in human brain: **V1** region detects edges based on orientation. Subsequent layers represent increasing abstract info. **V2** combines edges into contours. **V4**: sensitive to color and complex shape, **V5**: motion detection, **fusiform face area**: face recognition

From Biological vision to Machine vision





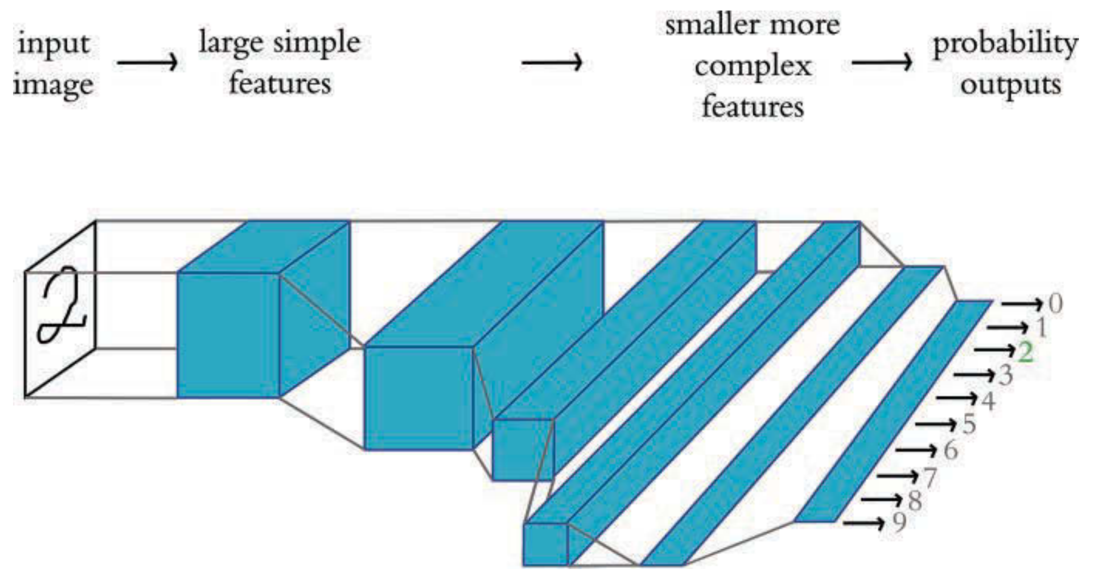
F. Chollet

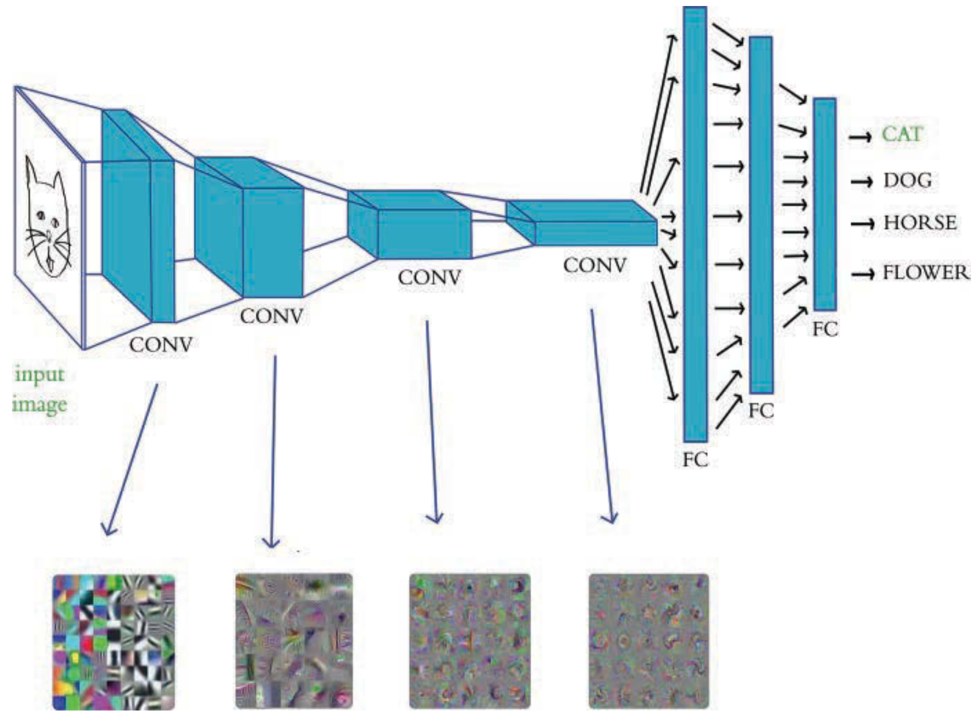
Similarly, in a CNN trained on MNIST digit classification, early layers detect edges and textures, while deeper layers recognize higher-order patterns.

By learning feature representations like the brain, CNNs work well for machine vision.

Evolution of CNN: From LeNet to AlexNet

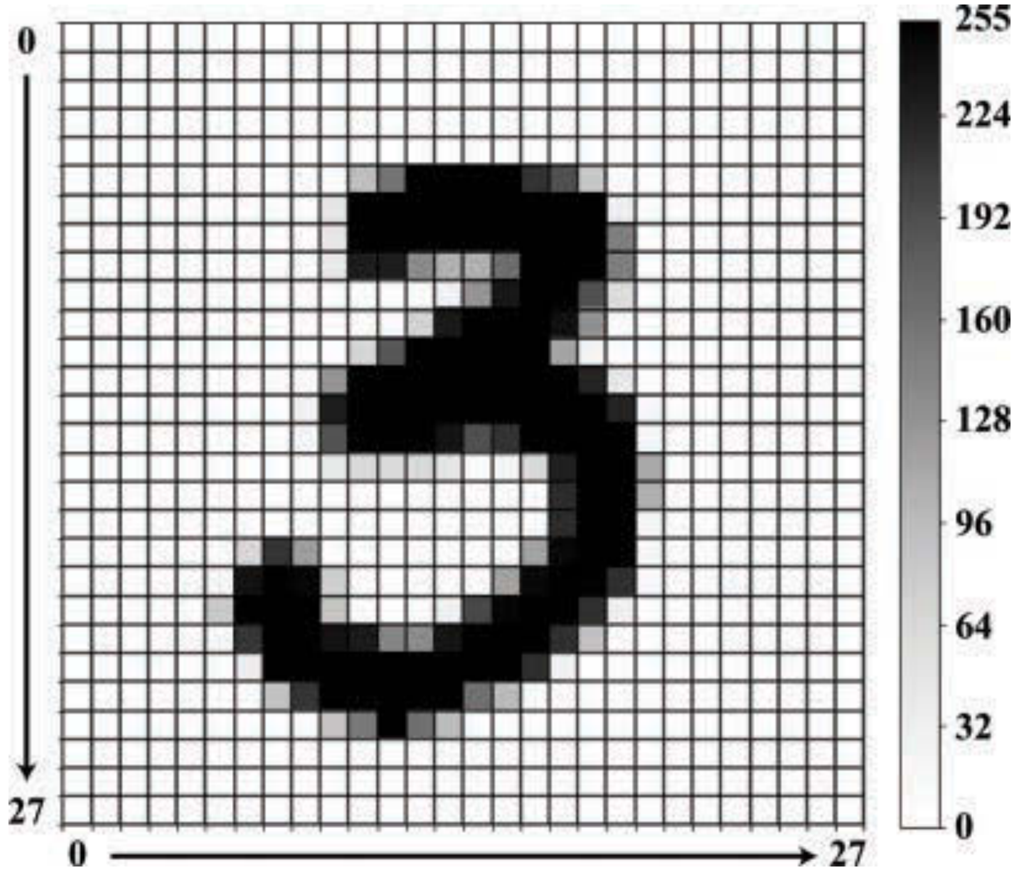
Krohn Ch1





- (Left) LeNet-5 retains the hierarchical architecture in the visual cortex. The first layer (leftmost) represents simple edges, while successive layers represent increasing complex features.
- (Right) AlexNet's hierarchical architecture is similar to LeNet, but AlexNet had more layers, was trained with larger datasets, and used GPU acceleration
- Breakthrough: AlexNet's success in the 2012 ImageNet competition shows the power of deep learning

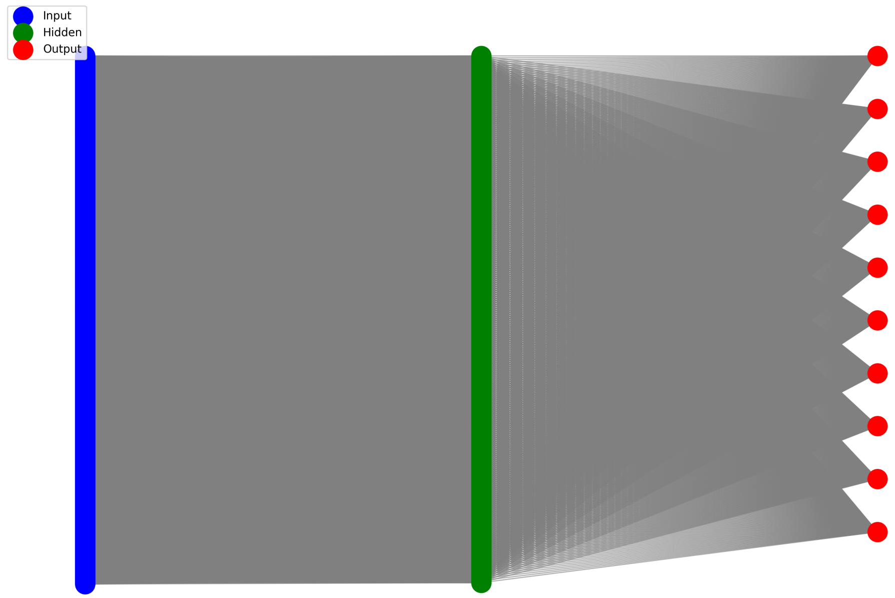
MNIST classification



- 28 x 28 pixel image (total of 28 x 28 = 784 values)
- Data for each pixel is stored as integer from 0 to 255

Naive approach: Fully connected layers for MNIST

scalability of dense layers



- Let’s connect each pixel in the image to every neuron in the next layer.

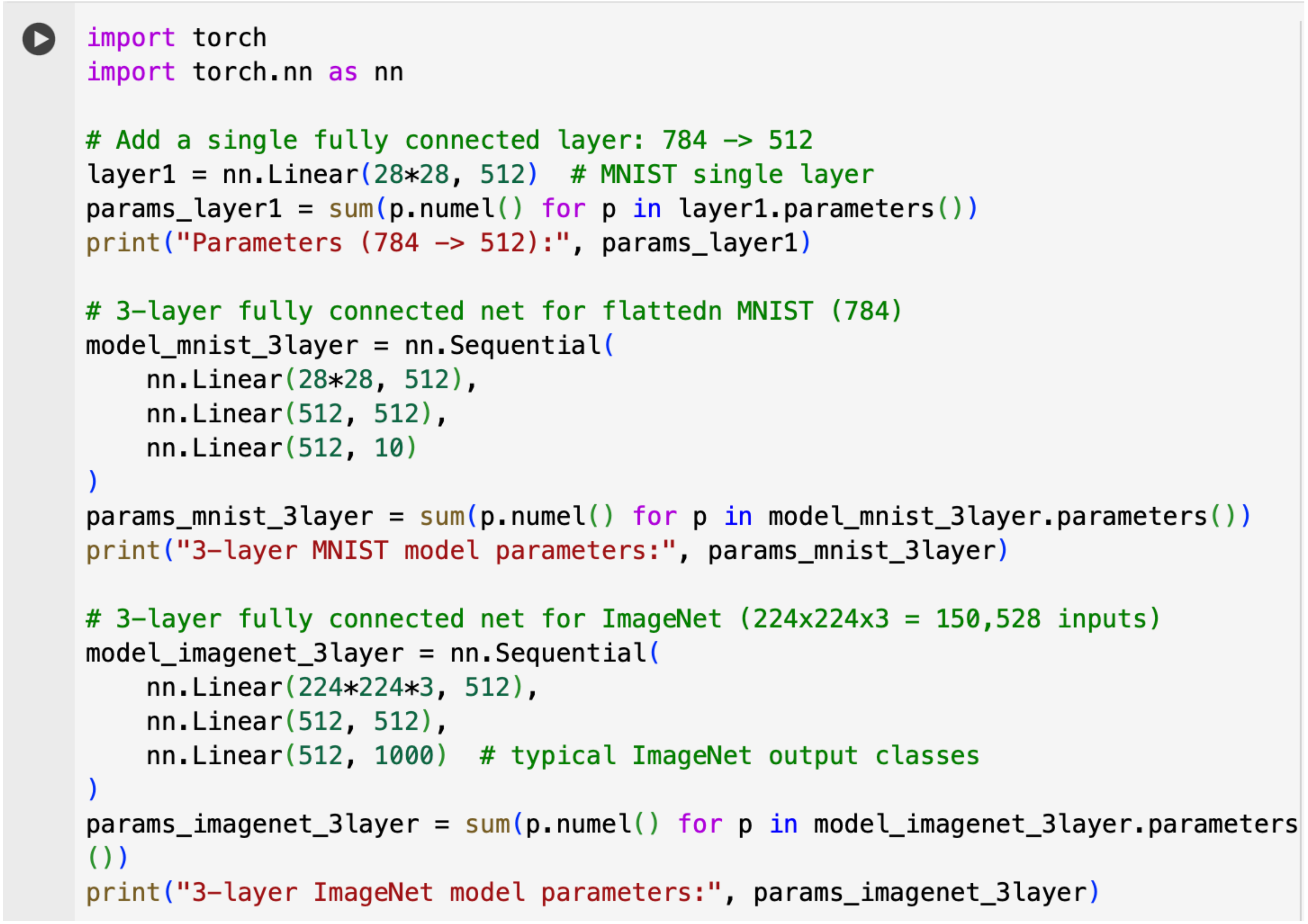
- For an MNIST image of 28x28 pixels (784 input nodes), a fully connected layer with 512 neurons requires (   ) parameters.

- Without flattening, 3-layer fully connected network for MNIST would require how many parameters?

- For ImageNet (224x224 pixels), a 3-layer fully connected network would result in how many parameters?

Naive approach: Fully connected layers for MNIST

Before executing PyTorch code, try to answer analytically.



```
import torch
import torch.nn as nn

# Add a single fully connected layer: 784 -> 512
layer1 = nn.Linear(28*28, 512)  # MNIST single layer
params_layer1 = sum(p.numel() for p in layer1.parameters())
print("Parameters (784 -> 512):", params_layer1)

# 3-layer fully connected net for flattedn MNIST (784)
model_mnist_3layer = nn.Sequential(
    nn.Linear(28*28, 512),
    nn.Linear(512, 512),
    nn.Linear(512, 10)
)
params_mnist_3layer = sum(p.numel() for p in model_mnist_3layer.parameters())
print("3-layer MNIST model parameters:", params_mnist_3layer)

# 3-layer fully connected net for ImageNet (224x224x3 = 150,528 inputs)
model_imagenet_3layer = nn.Sequential(
    nn.Linear(224*224*3, 512),
    nn.Linear(512, 512),
    nn.Linear(512, 1000)  # typical ImageNet output classes
)
params_imagenet_3layer = sum(p.numel() for p in model_imagenet_3layer.parameters
())
print("3-layer ImageNet model parameters:", params_imagenet_3layer)
```

The need for convolutional neural networks (CNN)

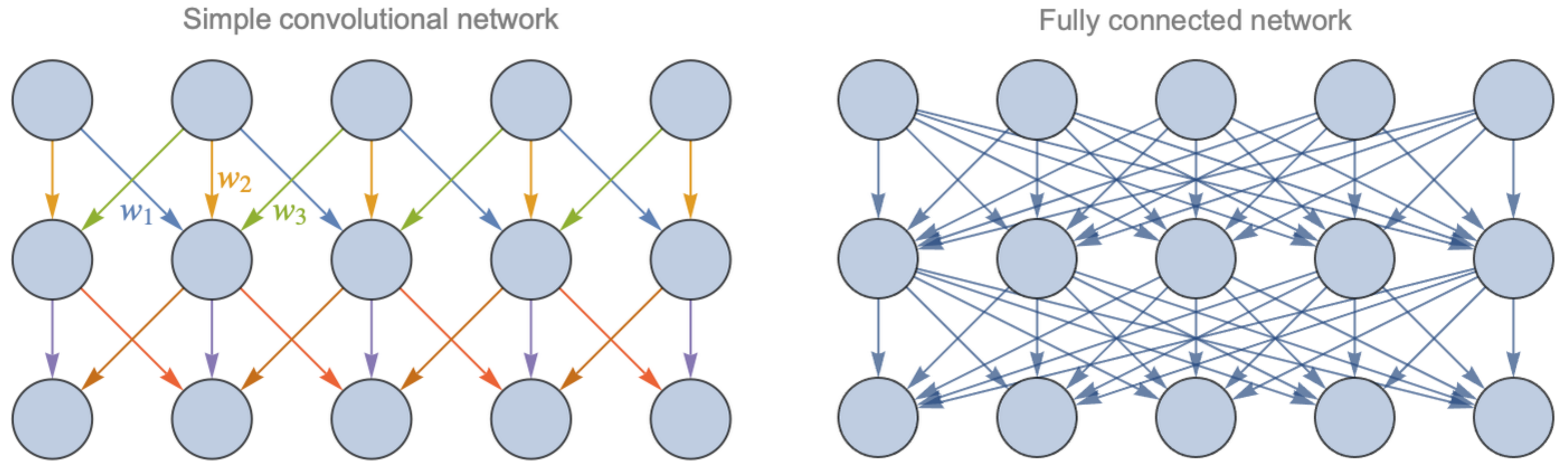
Why fully connected (FC) networks fail for images

- # of parameters in FC nets scale poorly with images, i.e. # parameter explodes as image resolution increases.

- FC nets don’t capture spatial relationship: Treats pixels independently ignoring the natural structure of images (i.e. there’s no hypothesis about the world).

- Overfitting becomes problematic => Large parameters counts leads to memorization rather than generalization.

Why CNN solves these issues



- Parameter Efficiency: CNNs drastically reduce the number of learnable parameters by sharing weights. CNNs also preserve spatial structure:

1. Locality: Nearby pixels are more likely to be related, allowing CNNs to capture patterns.

2. Translation Invariance: Features like edges or textures are recognized regardless of their position in the image.

3. Hierarchical Feature Learning: Similar to the human visual cortex, CNNs build complex representations (faces, objects) from simpler features (edges, corners).

Convolution as matrix operation

feature extraction with a sliding window

$$\begin{bmatrix} 2 & 1 & 3 & 1 & 4 \\ 4 & 1 & 0 & 1 & 3 \\ 1 & 3 & 1 & 1 & 1 \\ 2 & 6 & 1 & 1 & 1 \\ 3 & 3 & 1 & 0 & 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} \boxed{3} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

2x1 + 1x2 +
4x(-1) + 1x3
=3

Prewitt Kernels

Hand-designed kernel

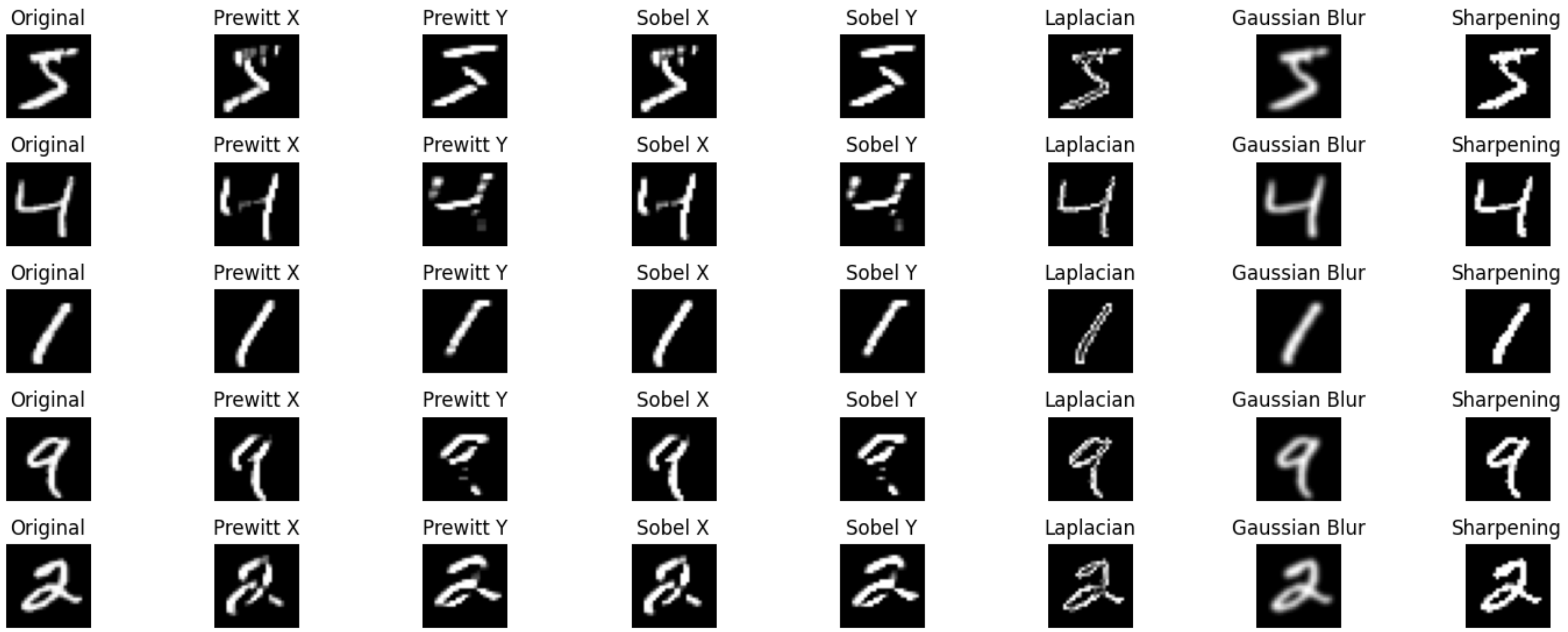
-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

- Connects to the idea of ‘receptive fields’. Each neuron is paying attention to only a particular part of the image.
- Each neuron has its own region of interest (ROI) in the image, and that is its receptive field.

Kernels

Prewitt Kernel edge detection



What about complex images? Design backward

How do we hand-design such kernels? Feature engineering? no way

- Yann LeCun realized he could train a neural net to learn these kernels.
- Training a network with backpropagation helps it find the appropriate kernels.
- This is a general concept in deep learning. No hand-crafted ‘forward design’, or ‘feature engineering’, but backward design via BP.
- All we need is to set up the architecture that captures the hypothesis or unique characteristics of the system (e.g. translational invariance for images, recurrent nature of time series, etc.).
- The rest is rather empirical. If the ‘frame’ is good, then BP will find efficient parameters. If the frame is bad, hard to find converging solution. We’ll use this approach for time series as well.

Deep Learning approach: Automatic feature learning

Why is it so revolutionary?

- Remove the burden of human-designed features.
- Instead, give the model the architecture and ‘freedom’ to discover the best way to solve the problem and let the backpropagation find the weights.

Spatial Learning
CNN: Recognize structure in images

Temporal Learning
LSTM: Detect sequences in data

Representation Learning
VAE: Compress and represent data

Linguistic Learning
LLM: Learn languages directly from texts

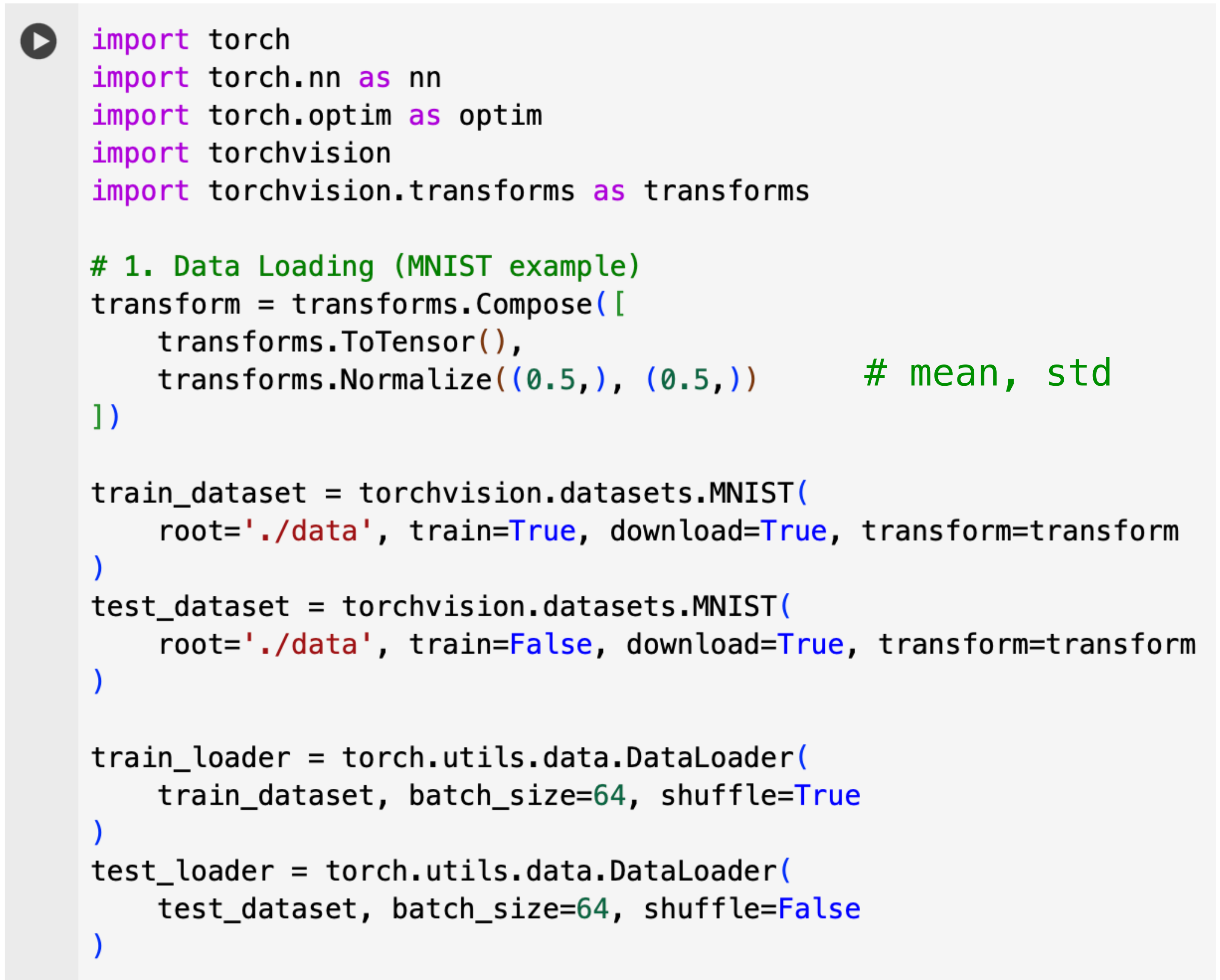
What is pooling? Why?

Biological insight: Downsampling

- Geoff Hinton drew inspiration for pooling layers from the visual cortex.
- The brain doesn't process every detail of an image with equal precision; instead, it focuses on key features while abstracting away finer details. Pooling layers mimic this process by downsampling information, reducing spatial resolution but retaining important features.
- Reduces spatial resolution: Make the model more computationally efficient.
- Translation Invariance: The model becomes more robust to small shifts or distortions in the input image.
- Without Pooling: Models may overfit or struggle to generalize due to too much reliance on local details.
- With Pooling: CNN learns abstract representation that’s less sensitive to exact position.

End-to-end MNIST classification in PyTorch

Part 1: MNIST data loading

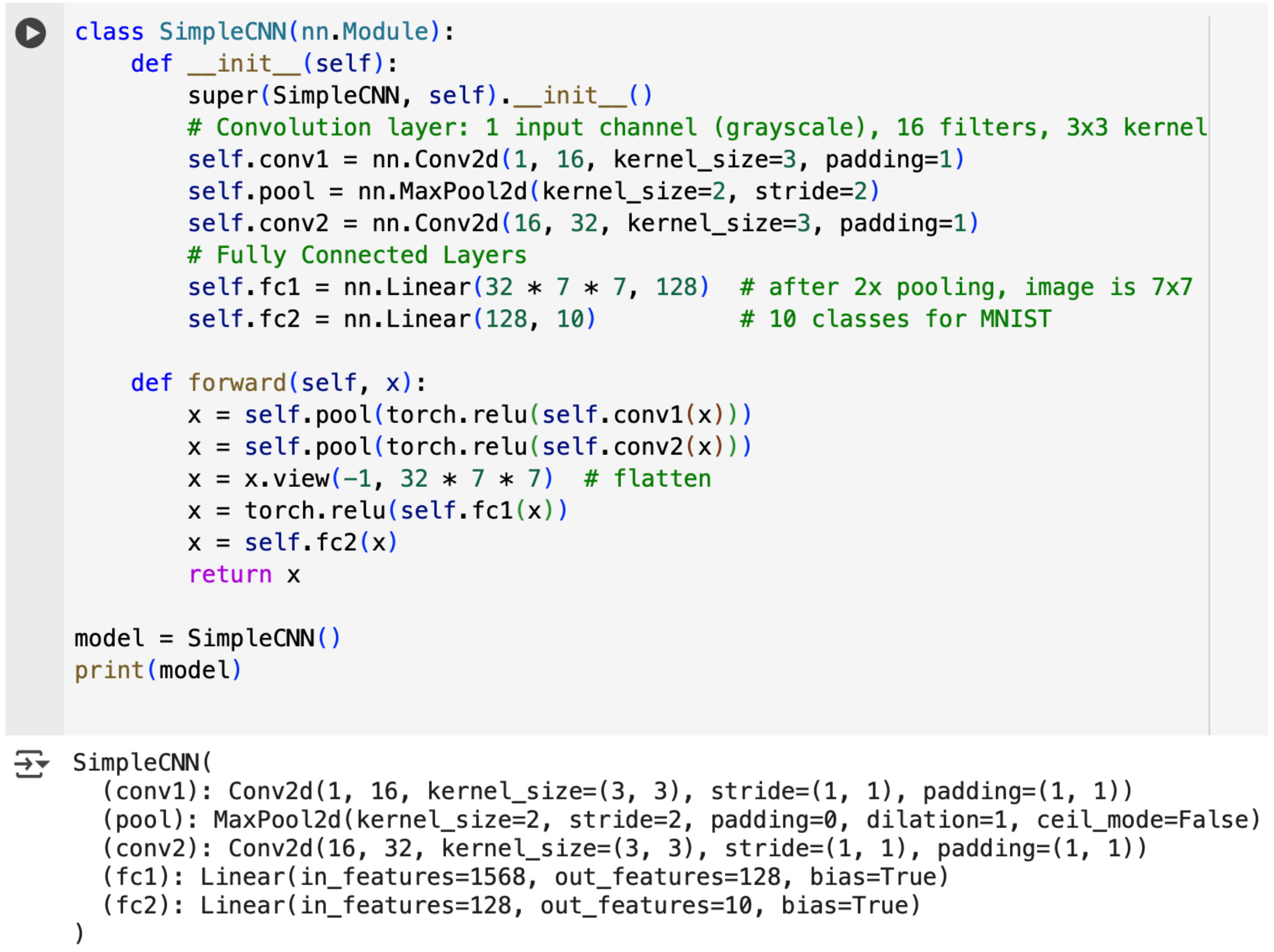


```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# 1. Data Loading (MNIST example)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))      # mean, std
])

train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform
)

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=64, shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=64, shuffle=False
)
```

70,000 grayscale images

28x28 pixels

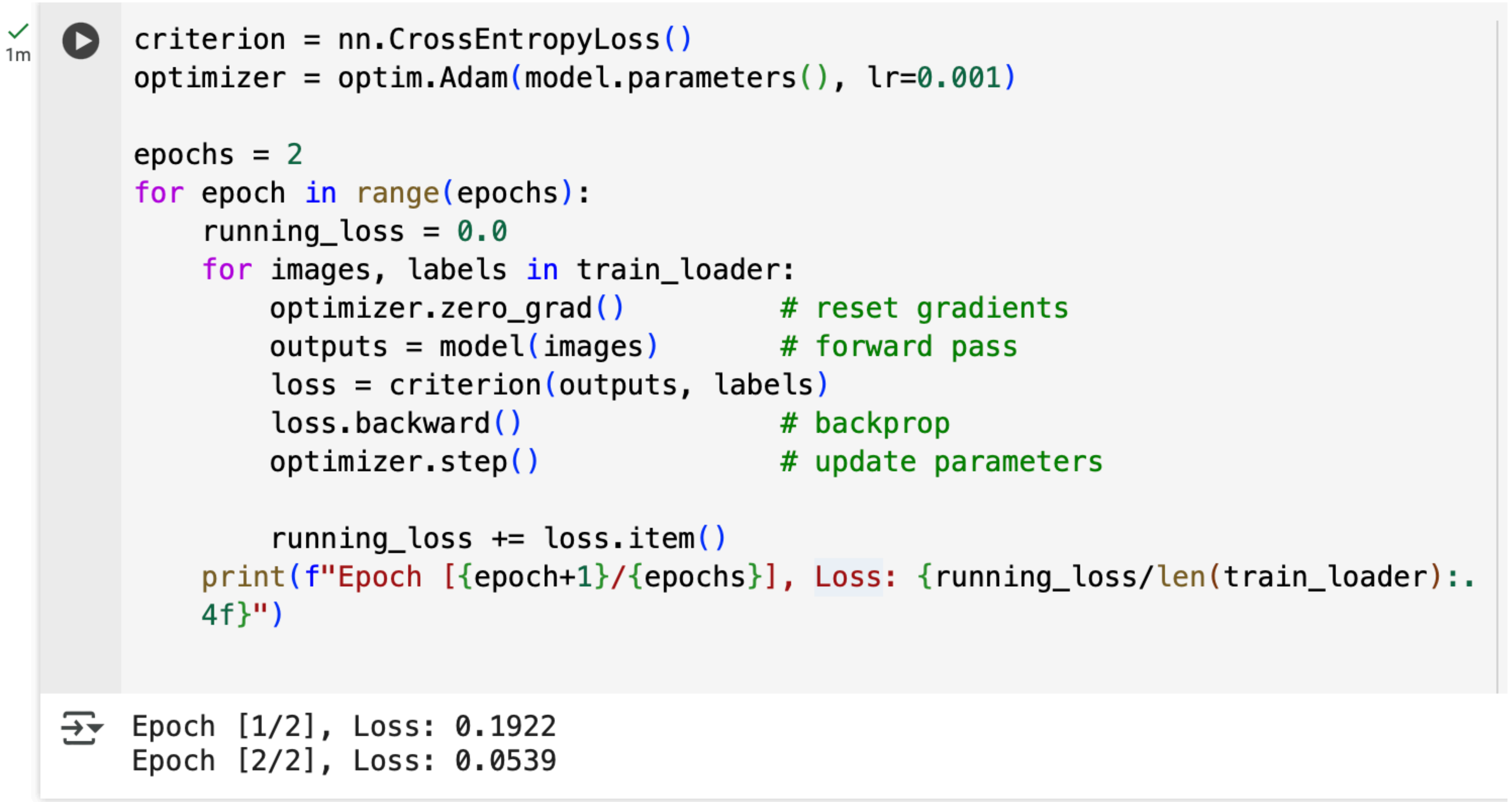
Part 2: Define CNN architecture



```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # Convolution layer: 1 input channel (grayscale), 16 filters, 3x3 kernel
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        # Fully Connected Layers
        self.fc1 = nn.Linear(32 * 7 * 7, 128)  # after 2x pooling, image is 7x7
        self.fc2 = nn.Linear(128, 10)          # 10 classes for MNIST

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)  # flatten
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN()
print(model)
```

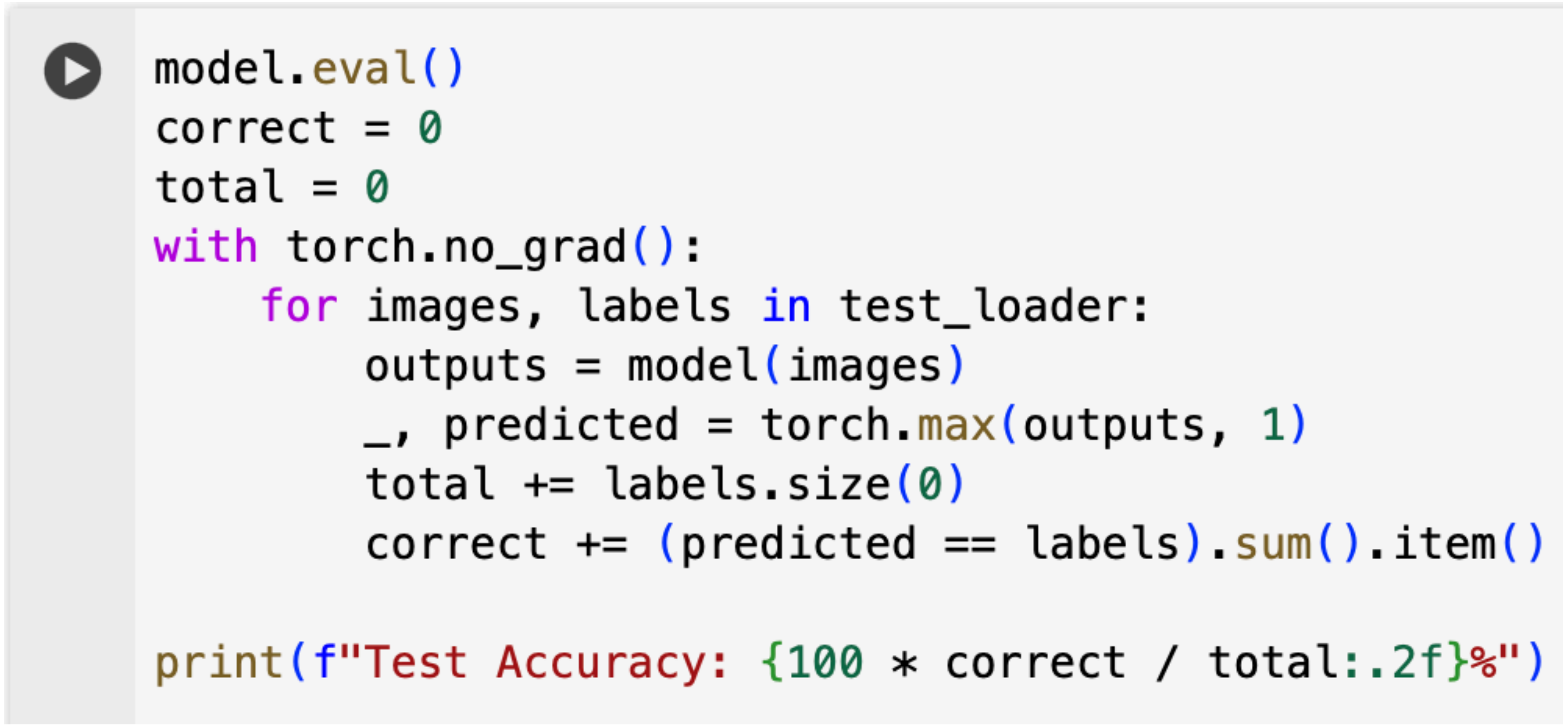
```
SimpleCNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1568, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

Loss function and optimizer



```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 2
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()          # reset gradients
        outputs = model(images)        # forward pass
        loss = criterion(outputs, labels)
        loss.backward()                # backprop
        optimizer.step()               # update parameters

        running_loss += loss.item()
    print(f"Epoch [{epoch+1}/{epochs}], Loss: {running_loss/len(train_loader):.
    4f}")
```

```
Epoch [1/2], Loss: 0.1922
Epoch [2/2], Loss: 0.0539
```



```python
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```