CS498: Algorithmic Engineering

Lecture 23: PyTorch in Practice, CNNs & Language Modeling

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 13

Outline

1. PyTorch in Practice
2. Convolutional Neural Networks
3. (Only if there is time) Beyond Images: Sequence Data

1 PyTorch in Practice

2 Convolutional Neural Networks

3 (Only if there is time) Beyond Images: Sequence Data

Where We Left Off

Last time: neural networks, nonlinearity, activations, depth, residual connections, LayerNorm, dropout.

Where We Left Off

Last time: neural networks, nonlinearity, activations, depth, residual connections, LayerNorm, dropout.

You already know `autograd` and `.backward()` from the convex optimization lectures.

Where We Left Off

Last time: neural networks, nonlinearity, activations, depth, residual connections, LayerNorm, dropout.

You already know `autograd` and `.backward()` from the convex optimization lectures.

The question is: how do we actually *build and train* a neural network in PyTorch?

Where We Left Off

Last time: neural networks, nonlinearity, activations, depth, residual connections, LayerNorm, dropout.

You already know `autograd` and `.backward()` from the convex optimization lectures.

The question is: how do we actually *build and train* a neural network in PyTorch?

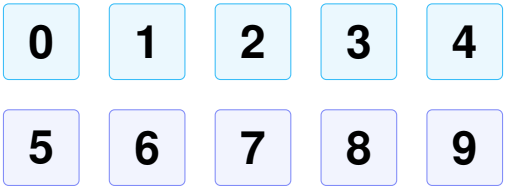
Today: practical APIs (`nn.Module`, `DataLoader`, training loop), then CNNs for images, then language modeling.

Our First Real Problem: MNIST

The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).

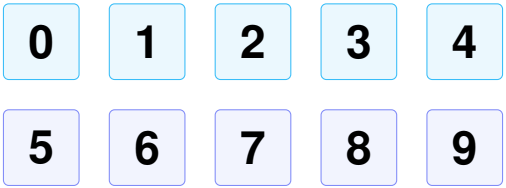
Our First Real Problem: MNIST

The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).



Our First Real Problem: MNIST

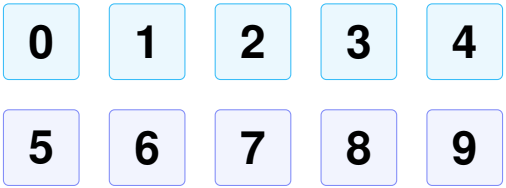
The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).



60,000 training images, 10,000 test images.

Our First Real Problem: MNIST

The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).

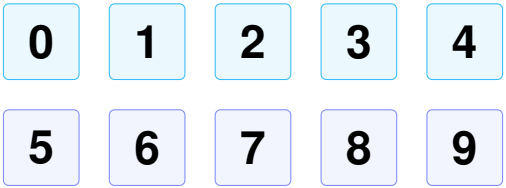


60,000 training images, 10,000 test images.

Each image: 28×28 pixels, single channel (grayscale). Pixel values in $[0, 255]$.

Our First Real Problem: MNIST

The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).



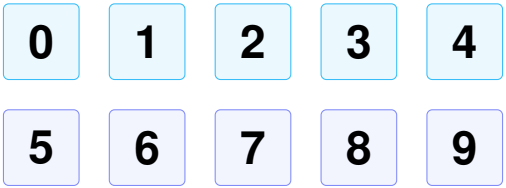
60,000 training images, 10,000 test images.

Each image: 28×28 pixels, single channel (grayscale). Pixel values in $[0, 255]$.

The task: given a 28×28 image, predict which digit (0 through 9) it shows.

Our First Real Problem: MNIST

The MNIST dataset: 70,000 grayscale images of handwritten digits (0 through 9).



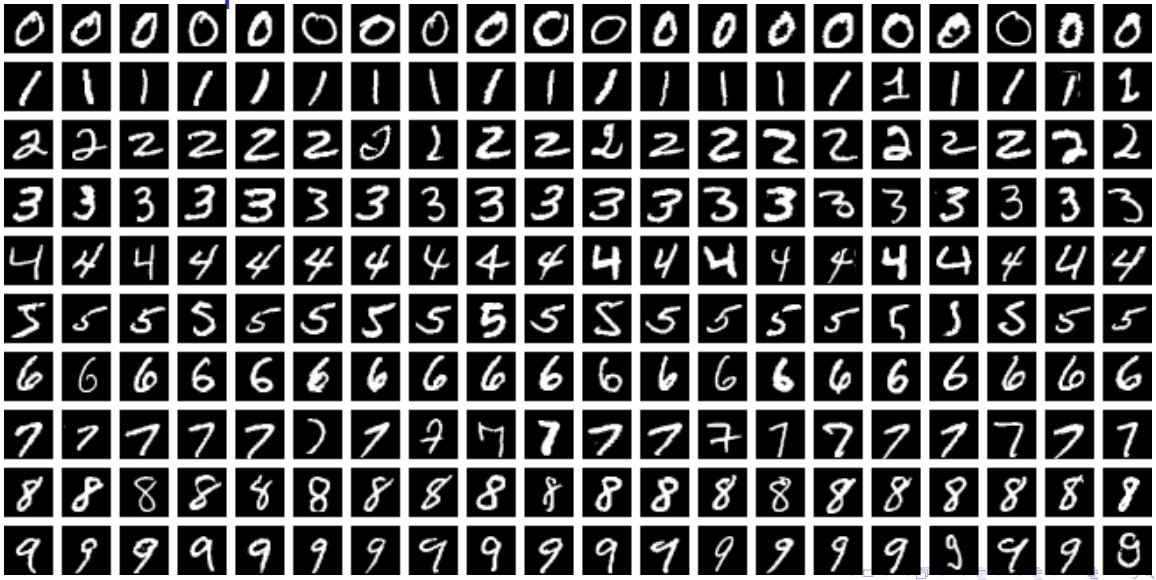
60,000 training images, 10,000 test images.

Each image: 28×28 pixels, single channel (grayscale). Pixel values in $[0, 255]$.

The task: given a 28×28 image, predict which digit (0 through 9) it shows.

This is a **10-class classification** problem.

Some Samples



The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

PyTorch adds a *channel* dimension: `[1, 28, 28]` (1 = grayscale; RGB would be 3)

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

PyTorch adds a *channel* dimension: `[1, 28, 28]` (1 = grayscale; RGB would be 3)

A batch of 64 images: `[64, 1, 28, 28]` (batch, channels, height, width)

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

PyTorch adds a *channel* dimension: `[1, 28, 28]` (1 = grayscale; RGB would be 3)

A batch of 64 images: `[64, 1, 28, 28]` (batch, channels, height, width)

For an MLP, we **flatten** the spatial dimensions: $28 \times 28 = 784$

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

PyTorch adds a *channel* dimension: `[1, 28, 28]` (1 = grayscale; RGB would be 3)

A batch of 64 images: `[64, 1, 28, 28]` (batch, channels, height, width)

For an MLP, we **flatten** the spatial dimensions: $28 \times 28 = 784$

After flattening: `[64, 784]`

The Shape of MNIST Data

An image is a 2D grid of pixel intensities.

Single image: shape `[28, 28]`

PyTorch adds a *channel* dimension: `[1, 28, 28]` (1 = grayscale; RGB would be 3)

A batch of 64 images: `[64, 1, 28, 28]` (batch, channels, height, width)

For an MLP, we **flatten** the spatial dimensions: $28 \times 28 = 784$

After flattening: `[64, 784]`

This is the input vector to our MLP. Each of the 784 entries is one pixel.

nn.Sequential for Simple Architectures

If you do not need skip connections/advanced architectures, `nn.Sequential` is the easiest way to define a neural network in PyTorch:

nn.Sequential for Simple Architectures

If you do not need skip connections/advanced architectures, `nn.Sequential` is the easiest way to define a neural network in PyTorch:

```
simple_mlp = nn.Sequential(
    nn.Flatten(), #(batch_size x 784)
    nn.Linear(784, 256), #(784 x 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 256), #(256, 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 10) #(256, 10) --> output
)
```

nn.Sequential for Simple Architectures

If you do not need skip connections/advanced architectures, `nn.Sequential` is the easiest way to define a neural network in PyTorch:

```
simple_mlp = nn.Sequential(
    nn.Flatten(), #(batch_size x 784)
    nn.Linear(784, 256), #(784 x 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 256), #(256, 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 10) #(256, 10) --> output
)
```

Layers execute in order.

nn.Sequential for Simple Architectures

If you do not need skip connections/advanced architectures, `nn.Sequential` is the easiest way to define a neural network in PyTorch:

```
simple_mlp = nn.Sequential(
    nn.Flatten(), #(batch_size x 784)
    nn.Linear(784, 256), #(784 x 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 256), #(256, 256)
    nn.ReLU(), #Relu activation
    nn.LayerNorm(256), #Add layer normalization, can also use nn.RMSNorm
    nn.Dropout(0.1), #Add dropout
    nn.Linear(256, 10) #(256, 10) --> output
)
```

Layers execute in order.

Trade-off: simpler code, but no residual connections or branching. Use `nn.Sequential` for simple stacks.

nn.Module: Building a Network

`nn.Module` is the base class for all neural network models in PyTorch.

nn.Module: Building a Network

`nn.Module` is the base class for all neural network models in PyTorch.

Two methods to define: `__init__` (create layers) and `forward` (define computation).

nn.Module: Building a Network

`nn.Module` is the base class for all neural network models in PyTorch.

Two methods to define: `__init__` (create layers) and `forward` (define computation).

```python
import torch.nn as nn

class CustomSuperDuperMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.block1 = nn.Sequential(nn.Linear(784, 256), nn.ReLU(),nn.LayerNorm(256), nn.Dropout(0.1))
        self.block2 = nn.Sequential(nn.Linear(256, 256), nn.ReLU(),nn.LayerNorm(256), nn.Dropout(0.1))
        self.head = nn.Linear(256, 10)

    def forward(self, x):
        x = self.flatten(x)          # [B, 1, 28, 28] -> [B, 784]
        h = self.block1(x)           # [B, 784] -> [B, 256]
        h = self.block2(h) + h       # residual connection!
        return self.head(h)          # [B, 256] -> [B, 10]
```

nn.Module: Building a Network

`nn.Module` is the base class for all neural network models in PyTorch.

Two methods to define: `__init__` (create layers) and `forward` (define computation).

```python
import torch.nn as nn

class CustomSuperDuperMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.block1 = nn.Sequential(nn.Linear(784, 256), nn.ReLU(),nn.LayerNorm(256), nn.Dropout(0.1))
        self.block2 = nn.Sequential(nn.Linear(256, 256), nn.ReLU(),nn.LayerNorm(256), nn.Dropout(0.1))
        self.head = nn.Linear(256, 10)

    def forward(self, x):
        x = self.flatten(x)          # [B, 1, 28, 28] -> [B, 784]
        h = self.block1(x)           # [B, 784] -> [B, 256]
        h = self.block2(h) + h       # residual connection!
        return self.head(h)          # [B, 256] -> [B, 10]
```

Notice: **LayerNorm**, **Dropout**, and a **residual connection**.

nn.Module: What Happens Under the Hood

When you assign an `nn.Module` or `nn.Parameter` as an attribute in `CustomSuperDuperMLP`, PyTorch *registers* it.

nn.Module: What Happens Under the Hood

When you assign an `nn.Module` or `nn.Parameter` as an attribute in `CustomSuperDuperMLP`, PyTorch *registers* it.

Meaning, for `model = CustomSuperDuperMLP()`, this means:

- `model.parameters()` returns all learnable weights
- `model.to(device)` moves the model's parameters to GPU
- `model.train()` / `model.eval()` toggles Dropout and LayerNorm behavior

nn.Module: What Happens Under the Hood

When you assign an `nn.Module` or `nn.Parameter` as an attribute in `CustomSuperDuperMLP`, PyTorch *registers* it.

Meaning, for `model = CustomSuperDuperMLP()`, this means:

- `model.parameters()` returns all learnable weights
- `model.to(device)` moves the model's parameters to GPU
- `model.train()` / `model.eval()` toggles Dropout and LayerNorm behavior

Calling `model(x)` invokes `model.forward(x)` automatically.

Parameter Count

How many learnable parameters does our MLP have?

Parameter Count

How many learnable parameters does our MLP have?

```
total = sum(p.numel() for p in model.parameters())
print(f"Parameters: {total:,}")   # Parameters: 270,346
```

Parameter Count

How many learnable parameters does our MLP have?

```
total = sum(p.numel() for p in model.parameters())
print(f"Parameters: {total:,}")   # Parameters: 270,346
```

Breaking it down:

- `block1`: $784 \times 256 + 256 = 200{,}960$ (Linear) $+512$ (LayerNorm)
- `block2`: $256 \times 256 + 256 = 65{,}792$ (Linear) $+512$ (LayerNorm)
- `head`: $256 \times 10 + 10 = 2{,}570$

Parameter Count

How many learnable parameters does our MLP have?

```
total = sum(p.numel() for p in model.parameters())
print(f"Parameters: {total:,}")   # Parameters: 270,346
```

Breaking it down:

- `block1`: $784 \times 256 + 256 = 200{,}960$ (Linear) $+512$ (LayerNorm)
- `block2`: $256 \times 256 + 256 = 65{,}792$ (Linear) $+512$ (LayerNorm)
- `head`: $256 \times 10 + 10 = 2{,}570$

$\sim$270K parameters. Tiny by modern standards.

Parameter Count

How many learnable parameters does our MLP have?

```
total = sum(p.numel() for p in model.parameters())
print(f"Parameters: {total:,}")   # Parameters: 270,346
```

Breaking it down:

- `block1`: $784 \times 256 + 256 = 200{,}960$ (Linear) $+512$ (LayerNorm)
- `block2`: $256 \times 256 + 256 = 65{,}792$ (Linear) $+512$ (LayerNorm)
- `head`: $256 \times 10 + 10 = 2{,}570$

$\sim$270K parameters. Tiny by modern standards.

For reference: GPT-3 has 175B parameters. GPT-4 is estimated at $\sim$1.8 trillion.

DataLoader: Batching and Shuffling

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),             # PIL image -> tensor, scales to [0,1]
])

train_data = datasets.MNIST('data', train=True,  download=True, transform=transform)
test_data  = datasets.MNIST('data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_data,  batch_size=1000, shuffle=False)
```

DataLoader: Batching and Shuffling

```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),              # PIL image -> tensor, scales to [0,1]
])

train_data = datasets.MNIST('data', train=True,  download=True, transform=transform)
test_data  = datasets.MNIST('data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_data,  batch_size=1000, shuffle=False)
```

`transforms.ToTensor()`: converts a PIL image to a `[C, H, W]` float tensor in $[0, 1]$.

DataLoader: Batching and Shuffling

```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),             # PIL image -> tensor, scales to [0,1]
])

train_data = datasets.MNIST('data', train=True,  download=True, transform=transform)
test_data  = datasets.MNIST('data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_data,  batch_size=1000, shuffle=False)
```

`transforms.ToTensor()`: converts a PIL image to a `[C, H, W]` float tensor in $[0, 1]$.

`Normalize((mean,), (std,))`: centers and scales pixel values. `DataLoader` handles: batching (64 images per batch), shuffling (randomize each epoch), parallel loading (`num_workers`).

The Training Loop

The core pattern (same for *every* supervised learning task):

The Training Loop

The core pattern (same for *every* supervised learning task):

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)  # move model parameters to GPU (if available)

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

model.train()
for epoch in range(10):
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)  # data to GPU
        logits = model(images)             # 1. forward pass
        loss = loss_fn(logits, labels)     # 2. compute loss
        optimizer.zero_grad()              # 3. clear old gradients
        loss.backward()                    # 4. compute gradients
        optimizer.step()                   # 5. update parameters
```

The Training Loop

The core pattern (same for *every* supervised learning task):

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)  # move model parameters to GPU (if available)

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

model.train()
for epoch in range(10):
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)  # data to GPU
        logits = model(images)             # 1. forward pass
        loss = loss_fn(logits, labels)     # 2. compute loss
        optimizer.zero_grad()              # 3. clear old gradients
        loss.backward()                    # 4. compute gradients
        optimizer.step()                   # 5. update parameters
```

Five lines that do all the work: forward, loss, zero_grad, backward, step.

The Training Loop

The core pattern (same for *every* supervised learning task):

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)  # move model parameters to GPU (if available)

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

model.train()
for epoch in range(10):
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)  # data to GPU
        logits = model(images)             # 1. forward pass
        loss = loss_fn(logits, labels)     # 2. compute loss
        optimizer.zero_grad()              # 3. clear old gradients
        loss.backward()                    # 4. compute gradients
        optimizer.step()                   # 5. update parameters
```

Five lines that do all the work: forward, loss, zero_grad, backward, step.

We use `AdamW` (from Lecture 21). `model.train()` enables Dropout and sets LayerNorm to training mode.

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Intuition: if $p_{\text{correct}} = 0.95$, loss $= -\log(0.95) = 0.05$.

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Intuition: if $p_{\text{correct}} = 0.95$, loss $= -\log(0.95) = 0.05$.

Numerical example: logits $= [2.0,\ 0.5,\ -1.0]$, true class $= 0$.

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Intuition: if $p_{\text{correct}} = 0.95$, loss $= -\log(0.95) = 0.05$.

Numerical example: logits $= [2.0,\ 0.5,\ -1.0]$, true class $= 0$.

Softmax: $[\frac{e^2}{e^2+e^{0.5}+e^{-1}}, \frac{e^{0.5}}{e^2+e^{0.5}+e^{-1}}, \frac{e^{-1}}{e^2+e^{0.5}+e^{-1}}] \approx [0.79, 0.17, 0.04]$

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Intuition: if $p_{\text{correct}} = 0.95$, loss $= -\log(0.95) = 0.05$.

Numerical example: logits $= [2.0,\ 0.5,\ -1.0]$, true class $= 0$.
Softmax: $[\frac{e^2}{e^2+e^{0.5}+e^{-1}}, \frac{e^{0.5}}{e^2+e^{0.5}+e^{-1}}, \frac{e^{-1}}{e^2+e^{0.5}+e^{-1}}] \approx [0.79, 0.17, 0.04]$
Loss $= -\log(0.79) \approx 0.241$

Cross-Entropy Loss

The model outputs **logits**: raw scores, one per class $(z_0, z_1, \ldots, z_9)$.

Softmax converts logits to probabilities:

$$p_i = \frac{\exp(z_i)}{\sum_{j=0}^{9} \exp(z_j)}$$

Cross-entropy loss: $L = -\log(p_{\text{correct}})$

Intuition: if $p_{\text{correct}} = 0.95$, loss $= -\log(0.95) = 0.05$.

Numerical example: logits $= [2.0,\ 0.5,\ -1.0]$, true class $= 0$.
Softmax: $[\frac{e^2}{e^2+e^{0.5}+e^{-1}}, \frac{e^{0.5}}{e^2+e^{0.5}+e^{-1}}, \frac{e^{-1}}{e^2+e^{0.5}+e^{-1}}] \approx [0.79, 0.17, 0.04]$
Loss $= -\log(0.79) \approx 0.241$

`nn.CrossEntropyLoss` does softmax + log + negate in one **numerically stable** operation.

Evaluation

At test time, disable Dropout:

Evaluation

At test time, disable Dropout:

```
model.eval()
correct, total = 0, 0
with torch.no_grad():              # don't track gradients
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        logits = model(images)
        preds = logits.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

print(f"Accuracy: {correct/total:.2%}")   # ~97.8%
```

Evaluation

At test time, disable Dropout:

```
model.eval()
correct, total = 0, 0
with torch.no_grad():              # don't track gradients
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        logits = model(images)
        preds = logits.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

print(f"Accuracy: {correct/total:.2%}")   # ~97.8%
```

`model.eval()`: switches Dropout off, LayerNorm uses running stats.
`torch.no_grad()`: saves memory by not building the computation graph.

Evaluation

At test time, disable Dropout:

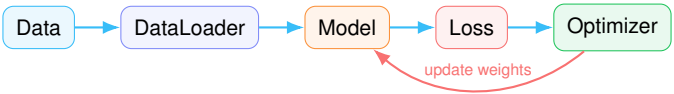
```
model.eval()
correct, total = 0, 0
with torch.no_grad():              # don't track gradients
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        logits = model(images)
        preds = logits.argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

print(f"Accuracy: {correct/total:.2%}")   # ~97.8%
```

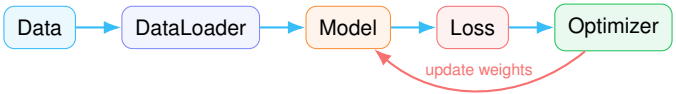
`model.eval()`: switches Dropout off, LayerNorm uses running stats.
`torch.no_grad()`: saves memory by not building the computation graph.

Result: $\sim$**98%** accuracy on MNIST with our simple MLP.

What We Just Built

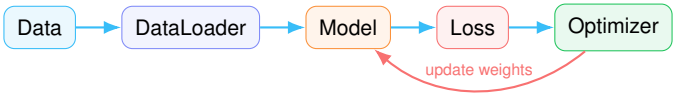


What We Just Built



This pattern is **identical for every supervised learning task**.

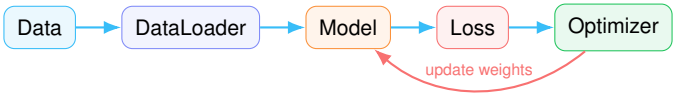
What We Just Built



This pattern is **identical for every supervised learning task**.

Only the **model** and **data** change. The loop stays the same.

What We Just Built



This pattern is **identical for every supervised learning task**.

Only the **model** and **data** change. The loop stays the same.

Want to classify cats vs. dogs? Swap the dataset and adjust the model. Want to predict stock prices? Same thing.

1 PyTorch in Practice

2 Convolutional Neural Networks

3 (Only if there is time) Beyond Images: Sequence Data

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

1. Locality

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

1. Locality
Nearby pixels are related. An edge is defined by *neighboring* pixels, not distant ones. A fully-connected layer connects every pixel to every neuron. Most of those connections are wasted.

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

1. Locality
Nearby pixels are related. An edge is defined by *neighboring* pixels, not distant ones. A fully-connected layer connects every pixel to every neuron. Most of those connections are wasted.

2. Translation invariance

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

1. Locality
Nearby pixels are related. An edge is defined by *neighboring* pixels, not distant ones. A fully-connected layer connects every pixel to every neuron. Most of those connections are wasted.

2. Translation invariance
A cat is a cat regardless of where it appears in the image. An MLP must re-learn the concept of “cat” for every possible position.

98% on MNIST. But...

Our MLP **flattens** the 28×28 image into a 784-dimensional vector. Two properties that MLPs completely ignore:

1. Locality
Nearby pixels are related. An edge is defined by *neighboring* pixels, not distant ones. A fully-connected layer connects every pixel to every neuron. Most of those connections are wasted.

2. Translation invariance
A cat is a cat regardless of where it appears in the image. An MLP must re-learn the concept of "cat" for every possible position.

Idea: build these two properties *into the architecture itself*.

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1, 3, 2, 4, 1]$ Kernel: $[1, 0, -1]$

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1, 3, 2, 4, 1]$ Kernel: $[1, 0, -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1,\ 3,\ 2,\ 4,\ 1]$ Kernel: $[1,\ 0,\ -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

Position 0: $1 \cdot 1 + 3 \cdot 0 + 2 \cdot (-1) \quad = -1$

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1,\ 3,\ 2,\ 4,\ 1]$ Kernel: $[1,\ 0,\ -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

Position 0: $1 \cdot 1 + 3 \cdot 0 + 2 \cdot (-1) = -1$

Position 1: $3 \cdot 1 + 2 \cdot 0 + 4 \cdot (-1) = -1$

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1,\ 3,\ 2,\ 4,\ 1]$ Kernel: $[1,\ 0,\ -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

Position 0: $1 \cdot 1 + 3 \cdot 0 + 2 \cdot (-1) = -1$

Position 1: $3 \cdot 1 + 2 \cdot 0 + 4 \cdot (-1) = -1$

Position 2: $2 \cdot 1 + 4 \cdot 0 + 1 \cdot (-1) = 1$

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1,\ 3,\ 2,\ 4,\ 1]$ Kernel: $[1,\ 0,\ -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

$$\begin{aligned}
\text{Position 0:} \quad & 1 \cdot 1 + 3 \cdot 0 + 2 \cdot (-1) &&= -1 \\
\text{Position 1:} \quad & 3 \cdot 1 + 2 \cdot 0 + 4 \cdot (-1) &&= -1 \\
\text{Position 2:} \quad & 2 \cdot 1 + 4 \cdot 0 + 1 \cdot (-1) &&= \ \ 1
\end{aligned}$$

Output: $[-1,\ -1,\ 1]$

The Convolution Operation (1D)

Start simple: a 1D signal and a small **kernel** (also called a filter).

Signal: $[1,\ 3,\ 2,\ 4,\ 1]$ Kernel: $[1,\ 0,\ -1]$

Slide the kernel across the signal. At each position: **element-wise multiply, then sum**.

$$\begin{aligned}
\text{Position 0:} \quad & 1 \cdot 1 + 3 \cdot 0 + 2 \cdot (-1) &&= -1 \\
\text{Position 1:} \quad & 3 \cdot 1 + 2 \cdot 0 + 4 \cdot (-1) &&= -1 \\
\text{Position 2:} \quad & 2 \cdot 1 + 4 \cdot 0 + 1 \cdot (-1) &&= \ \ 1
\end{aligned}$$

Output: $[-1,\ -1,\ 1]$

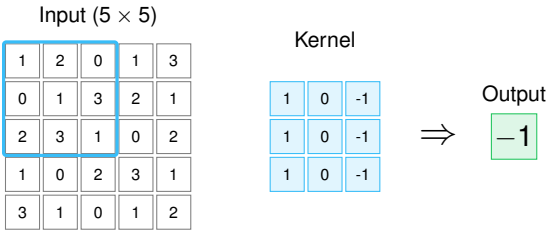
This kernel computes: (left value) $-$ (right value). It is a **difference detector**. Negative output $\Rightarrow$ values increasing. Positive $\Rightarrow$ values decreasing.

2D Convolution

The same idea, now on a 2D image.

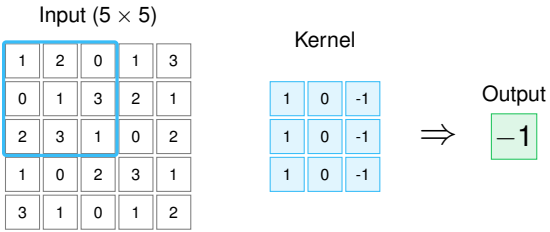
2D Convolution

The same idea, now on a 2D image.



2D Convolution

The same idea, now on a 2D image.

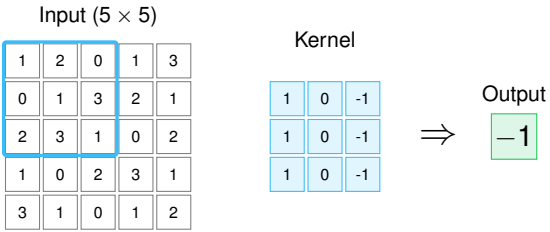


At this position:

$$(1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1)) + (0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)) + (2 \cdot 1 + 3 \cdot 0 + 1 \cdot (-1)) = -1$$

2D Convolution

The same idea, now on a 2D image.



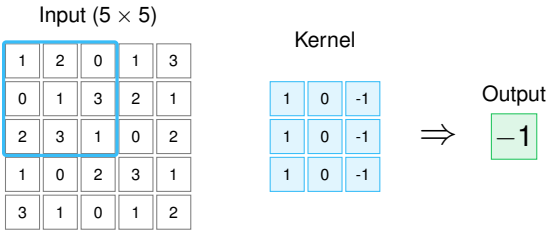
At this position:

$$(1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1)) + (0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)) + (2 \cdot 1 + 3 \cdot 0 + 1 \cdot (-1)) = -1$$

Slide the kernel to every valid position $\Rightarrow$ output is a 3×3 **feature map**.

2D Convolution

The same idea, now on a 2D image.



At this position:

$$(1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1)) + (0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)) + (2 \cdot 1 + 3 \cdot 0 + 1 \cdot (-1)) = -1$$

Slide the kernel to every valid position $\Rightarrow$ output is a 3×3 **feature map**.

A 5×5 input with a 3×3 kernel $\Rightarrow (5 - 3 + 1) \times (5 - 3 + 1) = 3 \times 3$ output.

What Do Kernels Detect?

Different kernels detect different patterns:

What Do Kernels Detect?

Different kernels detect different patterns:

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

Vertical edges

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

Horizontal edges

0	-1	0
-1	5	-1
0	-1	0

Sharpen

What Do Kernels Detect?

Different kernels detect different patterns:

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

Vertical edges

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

Horizontal edges

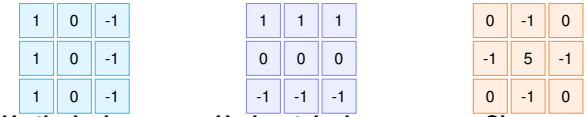
0	-1	0
-1	5	-1
0	-1	0

Sharpen

In classical computer vision, engineers *hand-designed* these kernels (Sobel, Laplacian, Gaussian).

What Do Kernels Detect?

Different kernels detect different patterns:

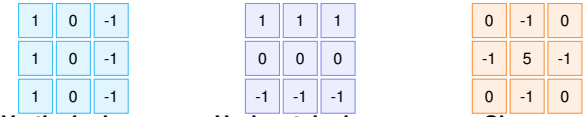


In classical computer vision, engineers *hand-designed* these kernels (Sobel, Laplacian, Gaussian).

Key insight: in a CNN, the kernel values are *learned parameters*.

What Do Kernels Detect?

Different kernels detect different patterns:



In classical computer vision, engineers *hand-designed* these kernels (Sobel, Laplacian, Gaussian).

Key insight: in a CNN, the kernel values are *learned parameters*.

The network discovers which patterns are useful for the task through gradient descent.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Compare with an MLP: a $32 \times 32 \times 3$ image flattened is 3,072 inputs. One hidden layer of size 256 needs $3{,}072 \times 256 = 786{,}432$ parameters.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Compare with an MLP: a $32 \times 32 \times 3$ image flattened is 3,072 inputs. One hidden layer of size 256 needs $3{,}072 \times 256 = 786{,}432$ parameters.

A conv layer with 1 kernel of size 3×3 on 3-channel input: $1 \times 3 \times 3 \times 3 + 1 = 28$ parameters.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Compare with an MLP: a $32 \times 32 \times 3$ image flattened is 3,072 inputs. One hidden layer of size 256 needs $3{,}072 \times 256 = 786{,}432$ parameters.

A conv layer with 1 kernel of size 3×3 on 3-channel input: $1 \times 3 \times 3 \times 3 + 1 = 28$ parameters.

That is $\approx 28086\times$ fewer parameters.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Compare with an MLP: a $32 \times 32 \times 3$ image flattened is 3,072 inputs. One hidden layer of size 256 needs $3{,}072 \times 256 = 786{,}432$ parameters.

A conv layer with 1 kernel of size 3×3 on 3-channel input: $1 \times 3 \times 3 \times 3 + 1 = 28$ parameters.

That is $\approx 28086\times$ fewer parameters.

Weight sharing *is* translation invariant: if a vertical edge exists at position $(5, 5)$, the same kernel detects it at $(9, 9)$.

Weight Sharing: The Key Property

The **same** kernel slides to every position in the image.

A 3×3 kernel has only **9 parameters**, regardless of image size.

Compare with an MLP: a $32 \times 32 \times 3$ image flattened is 3,072 inputs. One hidden layer of size 256 needs $3{,}072 \times 256 = 786{,}432$ parameters.

A conv layer with 1 kernel of size 3×3 on 3-channel input: $1 \times 3 \times 3 \times 3 + 1 = 28$ parameters.

That is $\approx 28086\times$ fewer parameters.

Weight sharing *is* translation invariant: if a vertical edge exists at position $(5, 5)$, the same kernel detects it at $(9, 9)$.

The network does not need to re-learn the same pattern at every location.

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

That means one kernel can detect **one kind of pattern** across the image.

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

That means one kernel can detect **one kind of pattern** across the image.

Want to detect many different patterns?

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

That means one kernel can detect **one kind of pattern** across the image.

Want to detect many different patterns?
Use **multiple kernels**.

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

That means one kernel can detect **one kind of pattern** across the image.

Want to detect many different patterns?
Use **multiple kernels**.

Example:

$$\text{32 kernels} \Rightarrow \text{32 output feature maps}$$

Feature Maps: Multiple Kernels

One kernel $\Rightarrow$ one output feature map.

That means one kernel can detect **one kind of pattern** across the image.

Want to detect many different patterns?
Use **multiple kernels**.

Example:

$$32 \text{ kernels } \Rightarrow 32 \text{ output feature maps}$$

Each kernel learns to respond to a different kind of visual pattern (for example: an edge, a corner, or a texture).

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

```
[64, 3, 32, 32]
```

meaning: 64 images, 3 input channels, height 32, width 32.

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

```
[64, 3, 32, 32]
```

meaning: 64 images, 3 input channels, height 32, width 32.

Each kernel must span **all input channels**.

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

```
[64, 3, 32, 32]
```

meaning: 64 images, 3 input channels, height 32, width 32.

Each kernel must span **all input channels**.
So a 3×3 kernel on an RGB image is really `[3, 3, 3]`.

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

```
[64, 3, 32, 32]
```

meaning: 64 images, 3 input channels, height 32, width 32.

Each kernel must span **all input channels**.
So a 3×3 kernel on an RGB image is really `[3, 3, 3]`.

At each spatial location, one kernel combines information from all 3 input channels and produces **one number**.

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

```
[64, 3, 32, 32]
```

meaning: 64 images, 3 input channels, height 32, width 32.

Each kernel must span **all input channels**.
So a 3×3 kernel on an RGB image is really `[3, 3, 3]`.

At each spatial location, one kernel combines information from all 3 input channels and produces **one number**.

So:

$$1 \text{ kernel } \Rightarrow 1 \text{ output channel}$$

Multiple Kernels: What Happens to the Channels?

Suppose the input has shape

`[64, 3, 32, 32]`

meaning: 64 images, 3 input channels, height 32, width 32.

Each kernel must span **all input channels**.
So a 3×3 kernel on an RGB image is really `[3, 3, 3]`.

At each spatial location, one kernel combines information from all 3 input channels and produces **one number**.

So:

$$1 \text{ kernel } \Rightarrow 1 \text{ output channel}$$

With 32 kernels of size 3×3: `[64, 3, 32, 32]` $\Rightarrow$ `[64, 32, 30, 30]`

Pooling: Reducing Spatial Size

After convolution, spatial dimensions can be large. **Pooling** reduces them.

Pooling: Reducing Spatial Size

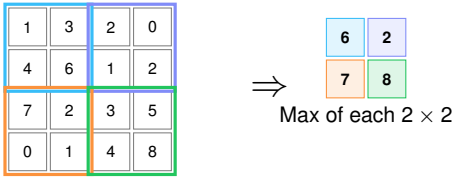
After convolution, spatial dimensions can be large. **Pooling** reduces them.

Max Pooling (`MaxPool2d(2)`): take the maximum in each 2×2 region.

Pooling: Reducing Spatial Size

After convolution, spatial dimensions can be large. **Pooling** reduces them.

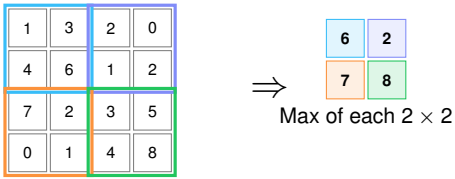
Max Pooling (`MaxPool2d(2)`): take the maximum in each 2×2 region.



Pooling: Reducing Spatial Size

After convolution, spatial dimensions can be large. **Pooling** reduces them.

Max Pooling (`MaxPool2d(2)`): take the maximum in each 2×2 region.

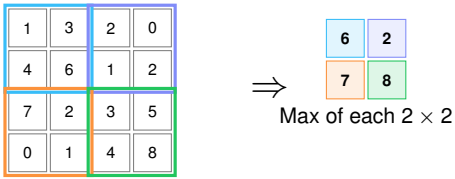


Input `[batch, C, 16, 16]` $\Rightarrow$ Output `[batch, C, 8, 8]`

Pooling: Reducing Spatial Size

After convolution, spatial dimensions can be large. **Pooling** reduces them.

Max Pooling (`MaxPool2d(2)`): take the maximum in each 2×2 region.



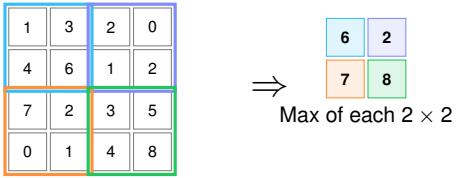
Input `[batch, C, 16, 16]` $\Rightarrow$ Output `[batch, C, 8, 8]`

Pooling **halves** spatial dimensions. Channel count stays the same.

Pooling: Reducing Spatial Size

After convolution, spatial dimensions can be large. **Pooling** reduces them.

Max Pooling (`MaxPool2d(2)`): take the maximum in each 2×2 region.



Input `[batch, C, 16, 16]` $\Rightarrow$ Output `[batch, C, 8, 8]`

Pooling **halves** spatial dimensions. Channel count stays the same.

Purpose: reduces computation, adds some translation tolerance. If a feature was detected *anywhere* in the 2×2 region, the max captures it.

Stride and Padding

Two important hyperparameters of a conv layer:

Stride and Padding

Two important hyperparameters of a conv layer:

Stride: how far the kernel moves at each step.

Stride and Padding

Two important hyperparameters of a conv layer:

Stride: how far the kernel moves at each step.

Stride 1: kernel moves 1 pixel at a time (default).
Stride 2: moves 2 pixels $\Rightarrow$ output is **half the size** (like pooling).

Stride and Padding

Two important hyperparameters of a conv layer:

Stride: how far the kernel moves at each step.

Stride 1: kernel moves 1 pixel at a time (default).
Stride 2: moves 2 pixels $\Rightarrow$ output is **half the size** (like pooling).

Padding: add zeros around the border of the input.

Stride and Padding

Two important hyperparameters of a conv layer:

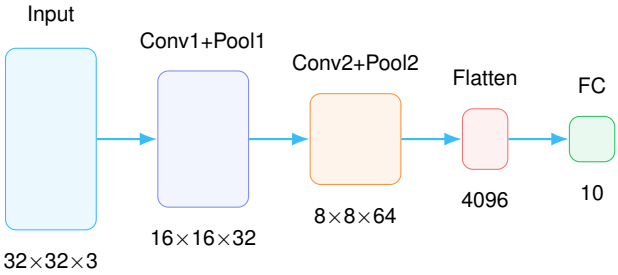
Stride: how far the kernel moves at each step.

Stride 1: kernel moves 1 pixel at a time (default).
Stride 2: moves 2 pixels $\Rightarrow$ output is **half the size** (like pooling).

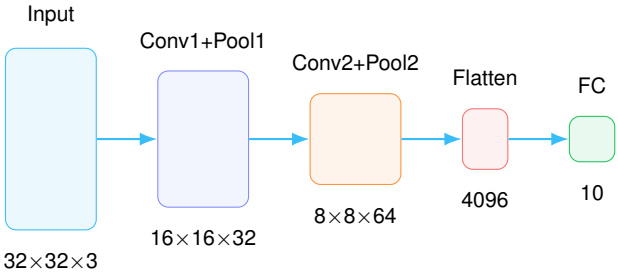
Padding: add zeros around the border of the input.

`padding=0`: output shrinks (default).
`padding=1` with a 3×3 kernel: output size = input size.

A CNN Architecture

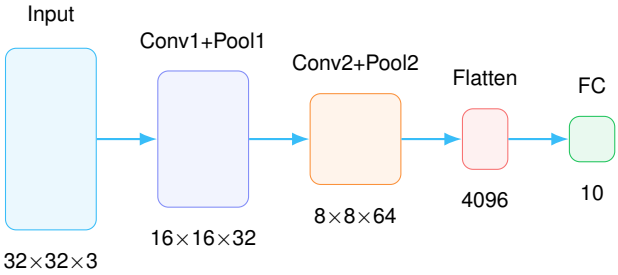


A CNN Architecture



Assume each convolution uses a 3×3 kernel with stride 1 and padding 1, and each pooling layer is 2×2 max-pooling with stride 2.

A CNN Architecture



Assume each convolution uses a 3×3 kernel with stride 1 and padding 1, and each pooling layer is 2×2 max-pooling with stride 2.

- Conv1: 32 kernels, 3×3, stride 1, padding 1 $\Rightarrow 32 \times 32 \times 32$
- Pool1: 2×2, stride 2 $\Rightarrow 16 \times 16 \times 32$
- Conv2: 64 kernels, 3×3, stride 1, padding 1 $\Rightarrow 16 \times 16 \times 64$
- Pool2: 2×2, stride 2 $\Rightarrow 8 \times 8 \times 64$

CNN in PyTorch

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # [B,3,32,32] -> [B,32,32,32]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,32,32,32] -> [B,32,16,16]
            nn.Conv2d(32, 64, kernel_size=3, padding=1), # [B,32,16,16] -> [B,64,16,16]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,64,16,16] -> [B,64,8,8]
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),                                # [B,64,8,8] -> [B,4096]
            nn.Linear(64 * 8 * 8, 256), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(256, 10)
        )

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

CNN in PyTorch

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # [B,3,32,32] -> [B,32,32,32]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,32,32,32] -> [B,32,16,16]
            nn.Conv2d(32, 64, kernel_size=3, padding=1), # [B,32,16,16] -> [B,64,16,16]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,64,16,16] -> [B,64,8,8]
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),                                # [B,64,8,8] -> [B,4096]
            nn.Linear(64 * 8 * 8, 256), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(256, 10)
        )

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

`Conv2d(in_channels, out_channels, kernel_size, padding)`

CNN in PyTorch

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # [B,3,32,32] -> [B,32,32,32]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,32,32,32] -> [B,32,16,16]
            nn.Conv2d(32, 64, kernel_size=3, padding=1), # [B,32,16,16] -> [B,64,16,16]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,64,16,16] -> [B,64,8,8]
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),                                # [B,64,8,8] -> [B,4096]
            nn.Linear(64 * 8 * 8, 256), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(256, 10)
        )

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

`Conv2d(in_channels, out_channels, kernel_size, padding)`

`MaxPool2d(2)` halves spatial dimensions. `Flatten()` collapses [C, H, W] into one dim.

CNN in PyTorch

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # [B,3,32,32] -> [B,32,32,32]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,32,32,32] -> [B,32,16,16]
            nn.Conv2d(32, 64, kernel_size=3, padding=1), # [B,32,16,16] -> [B,64,16,16]
            nn.ReLU(),
            nn.MaxPool2d(2),                             # [B,64,16,16] -> [B,64,8,8]
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),                                # [B,64,8,8] -> [B,4096]
            nn.Linear(64 * 8 * 8, 256), nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(256, 10)
        )

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

`Conv2d(in_channels, out_channels, kernel_size, padding)`

`MaxPool2d(2)` halves spatial dimensions. `Flatten()` collapses [C, H, W] into one dim.

Shapes are tracked in comments. Always verify your shapes!

Conv2d Parameters Explained

```
nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)
```

Conv2d Parameters Explained

`nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`

`in_channels`	Number of channels in the input (3 for RGB, 32 for a previous conv output)
`out_channels`	Number of kernels = number of output feature maps
`kernel_size`	Size of each kernel (3 means 3×3)
`stride`	Step size when sliding the kernel (default 1)
`padding`	Zeros added to each border (1 preserves spatial size for 3×3)

Conv2d Parameters Explained

`nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`

`in_channels`	Number of channels in the input (3 for RGB, 32 for a previous conv output)
`out_channels`	Number of kernels = number of output feature maps
`kernel_size`	Size of each kernel (3 means 3×3)
`stride`	Step size when sliding the kernel (default 1)
`padding`	Zeros added to each border (1 preserves spatial size for 3×3)

Number of parameters: out_channels $\times$ in_channels $\times$ $k \times k$ + out_channels (bias)

Conv2d Parameters Explained

`nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`

`in_channels`	Number of channels in the input (3 for RGB, 32 for a previous conv output)
`out_channels`	Number of kernels = number of output feature maps
`kernel_size`	Size of each kernel (3 means 3×3)
`stride`	Step size when sliding the kernel (default 1)
`padding`	Zeros added to each border (1 preserves spatial size for 3×3)

Number of parameters: out_channels $\times$ in_channels $\times$ $k \times k$ + out_channels (bias)

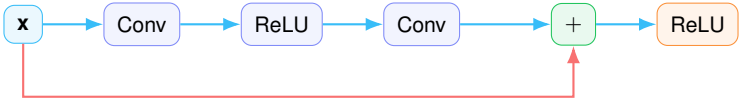
Example: `Conv2d(3, 32, 3, padding=1)` has $32 \times 3 \times 3 \times 3 + 32 = 896$ parameters.

Residual Connections in CNNs: ResNets

Skip connections from Lecture 22, applied to conv blocks:

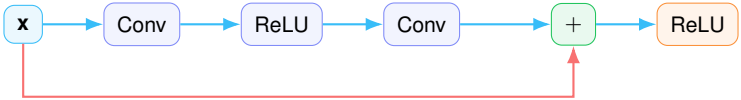
Residual Connections in CNNs: ResNets

Skip connections from Lecture 22, applied to conv blocks:



Residual Connections in CNNs: ResNets

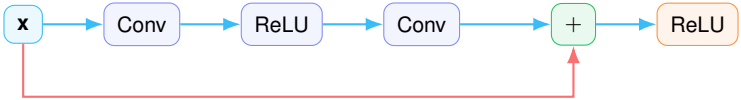
Skip connections from Lecture 22, applied to conv blocks:



Output = $F(\mathbf{x}) + \mathbf{x}$, where F is two conv layers.

Residual Connections in CNNs: ResNets

Skip connections from Lecture 22, applied to conv blocks:

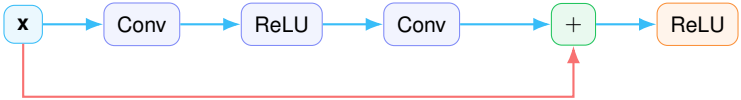


Output = $F(\mathbf{x}) + \mathbf{x}$, where F is two conv layers.

ResNet (He et al., 2015): stack many residual blocks.

Residual Connections in CNNs: ResNets

Skip connections from Lecture 22, applied to conv blocks:

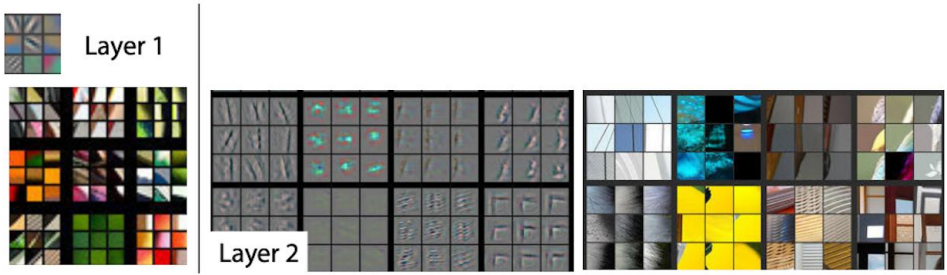


Output = $F(\mathbf{x}) + \mathbf{x}$, where F is two conv layers.

ResNet (He et al., 2015): stack many residual blocks.

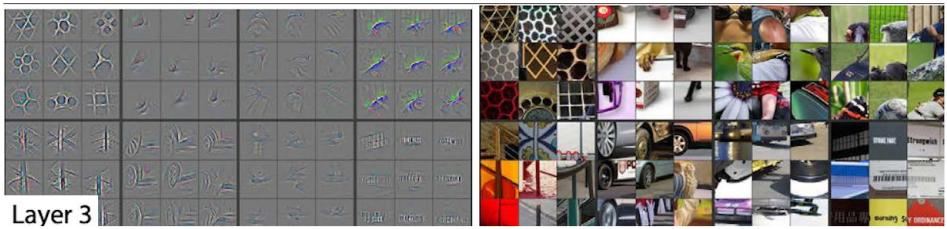
Model	Depth	ImageNet Top-5
ResNet-18	18 layers	10.9% error
ResNet-50	50 layers	7.1% error
ResNet-152	152 layers	5.7% error

What CNNs Learn: Early Layers



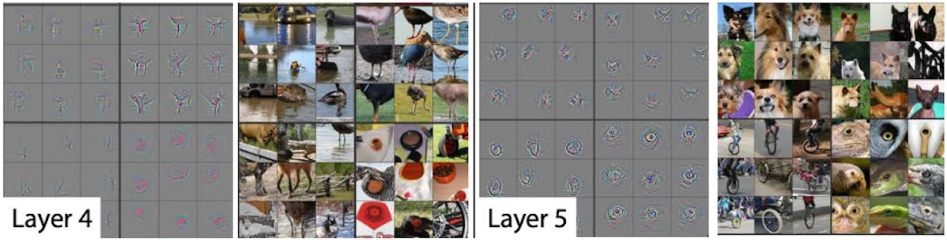
Early layers learn simple local detectors: **edges, color contrasts, and blobs.**

What CNNs Learn: Middle Layers



Middle layers combine simple detectors into **textures, corners, and repeated patterns.**

What CNNs Learn: Later Layers



Later layers respond to larger structures: **parts, motifs, and more semantic patterns.**

Takeaway

CNNs learn a **hierarchy of features**:

Takeaway

CNNs learn a **hierarchy of features**:

- Early layers: **edges, gradients, simple color contrasts**

Takeaway

CNNs learn a **hierarchy of features**:

- Early layers: **edges, gradients, simple color contrasts**
- Middle layers: **corners, textures, repeated motifs**

Takeaway

CNNs learn a **hierarchy of features**:

- Early layers: **edges, gradients, simple color contrasts**
- Middle layers: **corners, textures, repeated motifs**
- Later layers: **parts and higher-level visual structure**

Takeaway

CNNs learn a **hierarchy of features**:

- Early layers: **edges, gradients, simple color contrasts**
- Middle layers: **corners, textures, repeated motifs**
- Later layers: **parts and higher-level visual structure**

Takeaway

CNNs learn a **hierarchy of features**:

- Early layers: **edges, gradients, simple color contrasts**
- Middle layers: **corners, textures, repeated motifs**
- Later layers: **parts and higher-level visual structure**

Key point: we do not manually specify these detectors. We choose the architecture, and **training learns the useful features automatically**.

1. PyTorch in Practice
2. Convolutional Neural Networks
3. (Only if there is time) Beyond Images: Sequence Data

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

Key properties of sequences:

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

Key properties of sequences:

Order matters: “Dog bites man” $\neq$ “Man bites dog”

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

Key properties of sequences:

Order matters: "Dog bites man" $\neq$ "Man bites dog"

Length varies: sentences have different numbers of words.

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

Key properties of sequences:

Order matters: "Dog bites man" $\neq$ "Man bites dog"

Length varies: sentences have different numbers of words.

A CNN with a fixed kernel size can only see **local context** (a small window).

CNNs Are Great for Images. But...

Many important problems involve **sequences**, not grids:

- Text: sentences, documents, code
- Audio: waveforms, speech
- Time series: stock prices, sensor readings
- DNA: sequences of nucleotides

Key properties of sequences:

Order matters: "Dog bites man" $\neq$ "Man bites dog"

Length varies: sentences have different numbers of words.

A CNN with a fixed kernel size can only see **local context** (a small window).

For language, we often need to connect words that are far apart: "The cat *that I saw yesterday at the park near the river* **was** orange."

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: “The students opened their _____”

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: “The students opened their _____”

$\Rightarrow$ “books” (high probability), “laptops” (medium), “elephants” (low)

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: “The students opened their _____”

$\Rightarrow$ “books” (high probability), “laptops” (medium), “elephants” (low)

Why does this matter?

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their _____"

$\Rightarrow$ "books" (high probability), "laptops" (medium), "elephants" (low)

Why does this matter?

- Text generation: ChatGPT, Claude, Gemini
- Autocomplete: Gmail Smart Compose, GitHub Copilot

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their _____"

$\Rightarrow$ "books" (high probability), "laptops" (medium), "elephants" (low)

Why does this matter?

- Text generation: ChatGPT, Claude, Gemini
- Autocomplete: Gmail Smart Compose, GitHub Copilot
- Translation, summarization, question answering, code generation

Language Modeling: The Task

Language modeling: given previous words, predict the next one.

Formally: learn $p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their _____"

$\Rightarrow$ "books" (high probability), "laptops" (medium), "elephants" (low)

Why does this matter?

- Text generation: ChatGPT, Claude, Gemini
- Autocomplete: Gmail Smart Compose, GitHub Copilot
- Translation, summarization, question answering, code generation

Nearly all modern NLP is built on language modeling.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

"the cat" appears 50 times, "the dog" appears 30 times, "the" appears 1000 times.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

"the cat" appears 50 times, "the dog" appears 30 times, "the" appears 1000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05, \quad P(\text{dog} \mid \text{the}) = 30/1000 = 0.03$

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

"the cat" appears 50 times, "the dog" appears 30 times, "the" appears 1000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05, \quad P(\text{dog} \mid \text{the}) = 30/1000 = 0.03$

Trigram: $P(w_t \mid w_{t-2}, w_{t-1})$. Bigger table, more context.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

"the cat" appears 50 times, "the dog" appears 30 times, "the" appears 1000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05, \quad P(\text{dog} \mid \text{the}) = 30/1000 = 0.03$

Trigram: $P(w_t \mid w_{t-2}, w_{t-1})$. Bigger table, more context.

N-gram: $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$. Even bigger table, even more context.

The Simplest Approach: N-Grams

Bigram model: predict the next word using only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1},\ w_t)}{\text{count}(w_{t-1})}$$

Just a lookup table built by counting word pairs in a corpus.

Example from training data:

"the cat" appears 50 times, "the dog" appears 30 times, "the" appears 1000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05, \quad P(\text{dog} \mid \text{the}) = 30/1000 = 0.03$

Trigram: $P(w_t \mid w_{t-2}, w_{t-1})$. Bigger table, more context.

N-gram: $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$. Even bigger table, even more context.

Simple, fast, and they actually work reasonably well for short contexts.

N-Grams Cannot Generalize

The Sparsity Problem

If “the purple elephant” never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

N-Grams Cannot Generalize

The Sparsity Problem

If “the purple elephant” never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

Zero probability, not low probability. **Zero.**

N-Grams Cannot Generalize

The Sparsity Problem

If “the purple elephant” never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

Zero probability, not low probability. **Zero.**

The vocabulary is huge (tens of thousands of words). Most n-gram combinations are **unseen**.

N-Grams Cannot Generalize

The Sparsity Problem

If "the purple elephant" never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

Zero probability, not low probability. **Zero.**

The vocabulary is huge (tens of thousands of words). Most n-gram combinations are **unseen**.

For trigrams with a 50,000-word vocabulary: $50{,}000^3 = 1.25 \times 10^{14}$ possible entries. Even a massive corpus only covers a tiny fraction.

N-Grams Cannot Generalize

The Sparsity Problem

If "the purple elephant" never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

Zero probability, not low probability. **Zero.**

The vocabulary is huge (tens of thousands of words). Most n-gram combinations are **unseen**.

For trigrams with a 50,000-word vocabulary: $50{,}000^3 = 1.25 \times 10^{14}$ possible entries. Even a massive corpus only covers a tiny fraction.

N-grams **memorize** co-occurrences. They do not **generalize** to new contexts.

N-Grams Cannot Generalize

The Sparsity Problem

If "the purple elephant" never appeared in training, $P(\text{elephant} \mid \text{the purple}) = 0$.

Zero probability, not low probability. **Zero.**

The vocabulary is huge (tens of thousands of words). Most n-gram combinations are **unseen**.

For trigrams with a 50,000-word vocabulary: $50{,}000^3 = 1.25 \times 10^{14}$ possible entries. Even a massive corpus only covers a tiny fraction.

N-grams **memorize** co-occurrences. They do not **generalize** to new contexts.

The question: can we learn a function that generalizes to contexts never seen in training?

What We Need for Sequence Modeling

We need a model that:

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)
3. **Can be trained with gradient descent** (end-to-end differentiable)
4. **Parallelizes well on GPUs** (fast training on large datasets)

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)
3. **Can be trained with gradient descent** (end-to-end differentiable)
4. **Parallelizes well on GPUs** (fast training on large datasets)

N-grams fail at (1), (2), and (3).

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)
3. **Can be trained with gradient descent** (end-to-end differentiable)
4. **Parallelizes well on GPUs** (fast training on large datasets)

N-grams fail at (1), (2), and (3).

CNNs handle (3) and (4) but are limited on (1) and (2) due to fixed receptive fields.

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)
3. **Can be trained with gradient descent** (end-to-end differentiable)
4. **Parallelizes well on GPUs** (fast training on large datasets)

N-grams fail at (1), (2), and (3).

CNNs handle (3) and (4) but are limited on (1) and (2) due to fixed receptive fields.

Next lectures:

What We Need for Sequence Modeling

We need a model that:

1. **Handles variable-length sequences** (sentences have different lengths)
2. **Captures long-range dependencies** (subject-verb agreement across many words)
3. **Can be trained with gradient descent** (end-to-end differentiable)
4. **Parallelizes well on GPUs** (fast training on large datasets)

N-grams fail at (1), (2), and (3).

CNNs handle (3) and (4) but are limited on (1) and (2) due to fixed receptive fields.

Next lectures:

Tokenization: how to represent text as numbers.

The attention mechanism: every token can look at every other token simultaneously.