USC
University of
Southern California

ECE 105: Introduction to Electrical Engineering

Lab 4.1

Neural network training on MNIST

Yasser Khan

Rehan Kapadia

Objective

USC University of Southern California

Build, train and test neural networks in PyTorch, one part at a time.

By the end of this lab you can:

- Construct a simple **artificial neuron** and **perceptrons** in PyTorch
- **Train** neural networks on the **MNIST** handwritten digits
- Run **inference on a custom image** of your own digit
- Read the training curves to tell **underfitting** from **overfitting**



MNIST digits: 28 × 28 gray pixels, white on black.



SETUP

USC University of Southern California

No setup files to run: unzip the lab kit and open the notebook.

Windows and Mac

1. Unzip **mod-4-nn-labs.zip**
2. Open **lab4.1-neural-networks.ipynb** in Jupyter
3. Select **EE105** as the kernel
4. Run the first cell: it installs **torch** (PyTorch)

No board today.

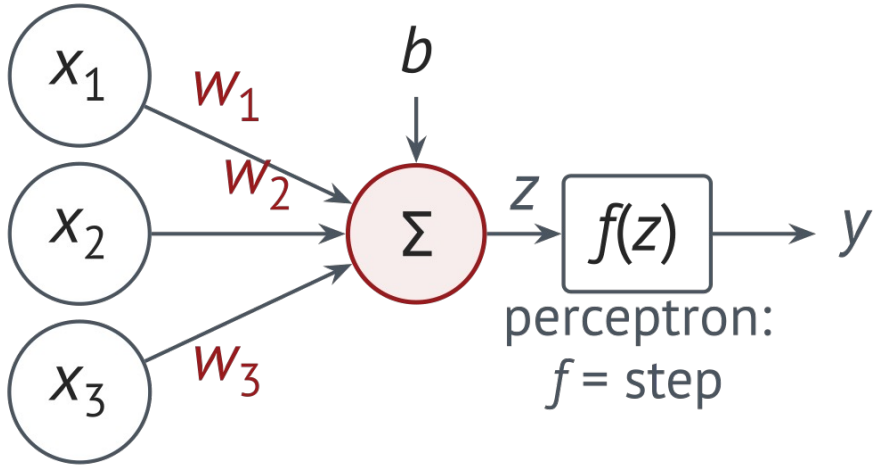
Parts B and C

A neuron multiplies, adds and applies an activation; a perceptron uses a step.

PARTS B, C

Goal: build one neuron in PyTorch and choose perceptron weights by hand.

1. **Forward pass** through one neuron: z, then step, ReLU, sigmoid, tanh
2. **B1, B2**: check z and σ(z) by hand
3. **Understand perceptrons**: the Coachella decision, then AND and OR
4. **Perceptron challenge** (C1): fix w and b in the NOR cells



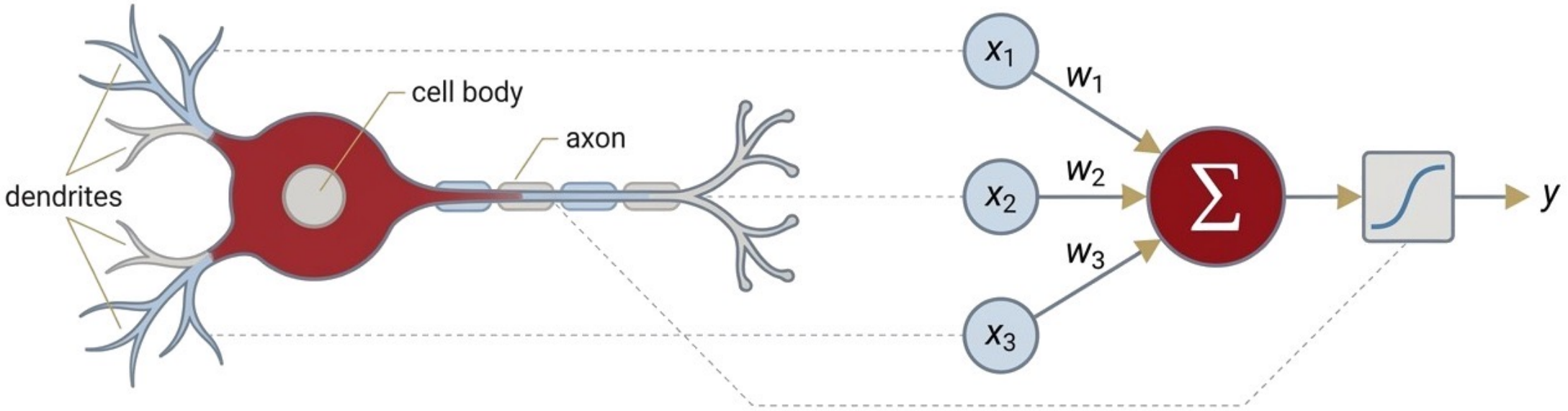
x_1	x_2	AND	OR	NAND	NOR
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Check (Checkpoint 1): your NAND and NOR tables both print “correct”, and the NOR table is right.

Fundamental Unit of a Neural Network

USC University of Southern California

A neuron multiplies each input by a weight, adds a bias, and passes the sum through a function.



A biological neuron (left) and the artificial neuron it inspired (right): weighted inputs, a sum, an activation, one output.

Mathematical definition

$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

- x_i: the **inputs**; y: the **output**
- w_i: the **weights**, also called synaptic weights
- b: the **bias**, a constant added to the sum
- f: the **neuron function** that maps inputs to output

Part B: torch

nn.Module holds a model; nn.Parameter marks the values that training may change.

```
# Define the neuron class
class SimpleNeuron(nn.Module):
    def __init__(self):
        super(SimpleNeuron, self).__init__()
        # Initialize weights and bias
        self.weights = nn.Parameter(torch.randn(3))  # w1..w3
        self.bias = nn.Parameter(torch.randn(1))     # b

    def forward(self, x):
        # Weighted sum of the inputs plus the bias
        return torch.sum(x * self.weights) + self.bias
torch.manual_seed(105)              # same weights for everyone
neuron = SimpleNeuron()
inputs = torch.tensor([1.0, 2.0, 3.0])
output = neuron(inputs)             # raw output z
activated = nn.ReLU()(output)       # also nn.Sigmoid, nn.Tanh
```

One neuron

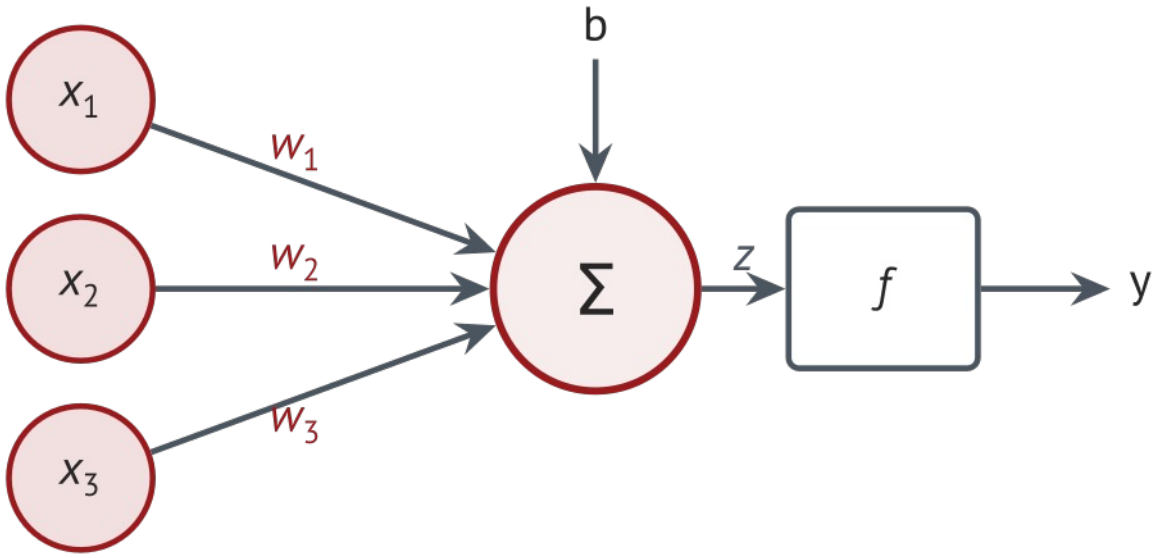
$$y = f(z), \qquad z = \sum_i w_i x_i + b$$

CHECK BY HAND

1. Notebook prints w = (−0.2055, 0.0008, 1.0546), b = −0.2668; x = (1, 2, 3)
2. z = −0.2055 + 0.0016 + 3.1638 − 0.2668 = **2.693** (matches z)
3. ReLU: max(0, z) = **2.693**; sigmoid 0.937; tanh 0.991

Fundamental Unit of a Neural Network

Many functions can serve as the neuron's f; the three most popular are sigmoid, ReLU and tanh.



Pre-activation, then activation

$$z = \mathbf{w} \cdot \mathbf{x} + b, \qquad y = f(z)$$

Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Rectified linear unit (ReLU)

$$f(z) = \max(0, z)$$

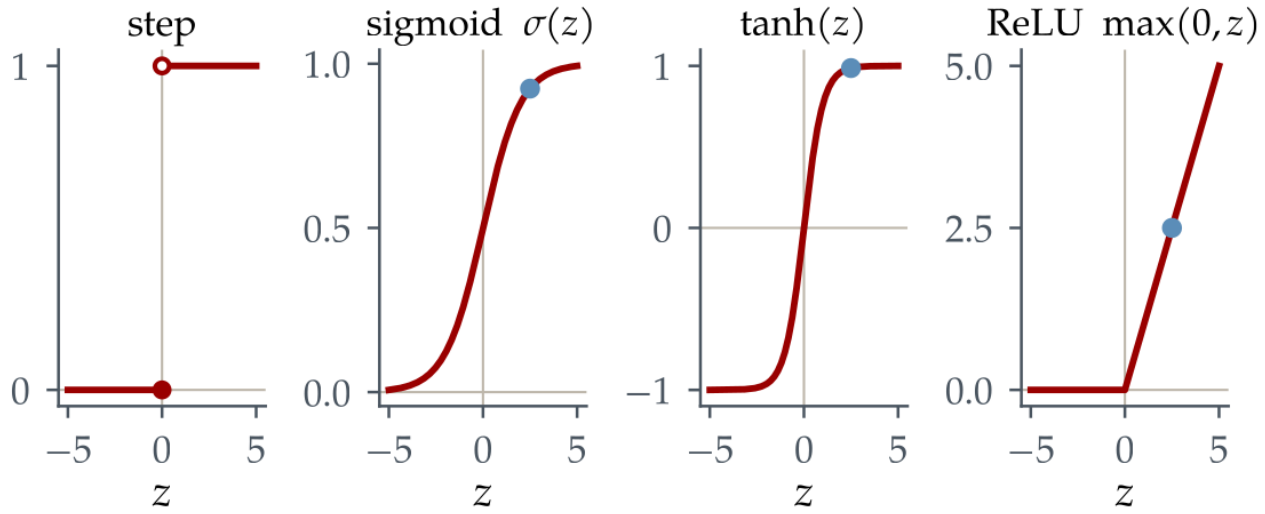
Hyperbolic tangent

$$\tanh(z) = 2\,\sigma(2z) - 1$$

Activation functions of a Neural Network

USC University of Southern California

Each activation squashes or clips the weighted sum z; its shape sets how well a network learns.

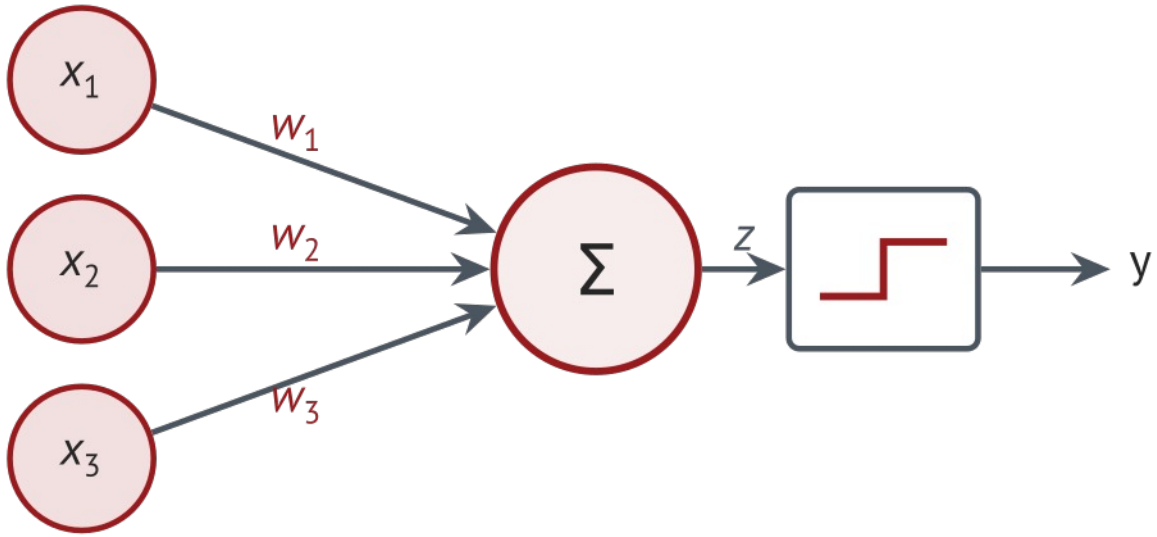


Blue dots: outputs at z = 2.5, where σ = 0.924, tanh = 0.987 and ReLU = 2.5.

Function	Range	Shape	Pros	Cons
Step	0 or 1	jumps from 0 to 1 at z = 0	clean yes/no decision	zero slope: cannot learn by small steps
Sigmoid σ(z)	(0, 1)	S-curve, σ(0) = 0.5	smooth; reads as a probability	saturates for large |z|; not zero-centered
tanh(z)	(−1, 1)	S-curve centered on 0	zero-centered	still saturates for large |z|
ReLU max(0, z)	[0, ∞)	0 for z < 0, z for z > 0	cheapest; no saturation for z > 0	"dead" neurons stuck at 0

Perceptron

A perceptron uses a step: it outputs 1 only when the weighted sum is above a threshold.



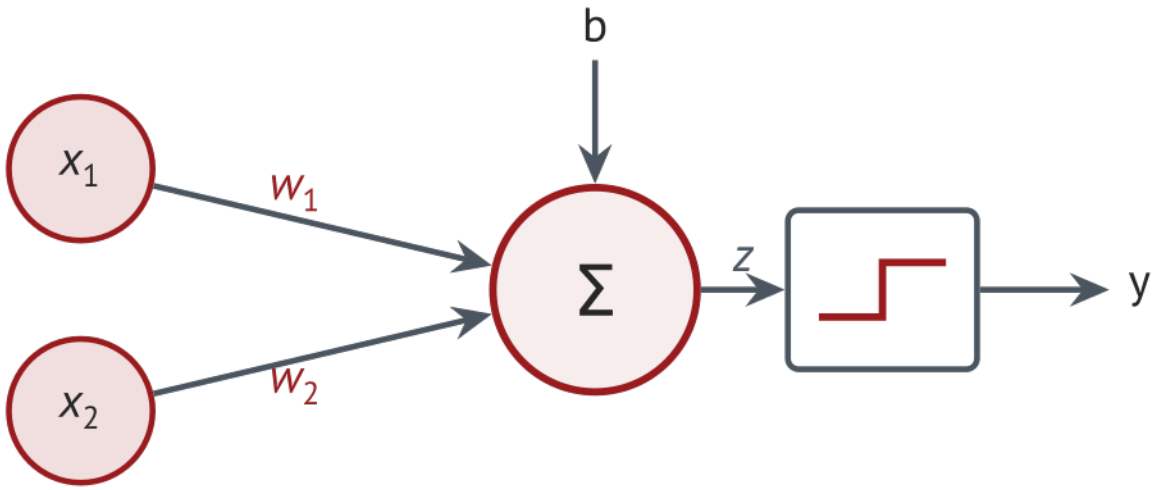
- Same unit as before, with a **step** function
- Inputs x_i and output y are 0 or 1
- The weights set how much each input counts
- The **threshold** sets how much evidence is enough

Perceptron rule

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases}$$

Perceptrons for logical functions?

A two-input perceptron can act as a logic gate: we only need the right weights and bias.



Target truth tables

x_1	x_2	AND	OR	NAND
0	0	0	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

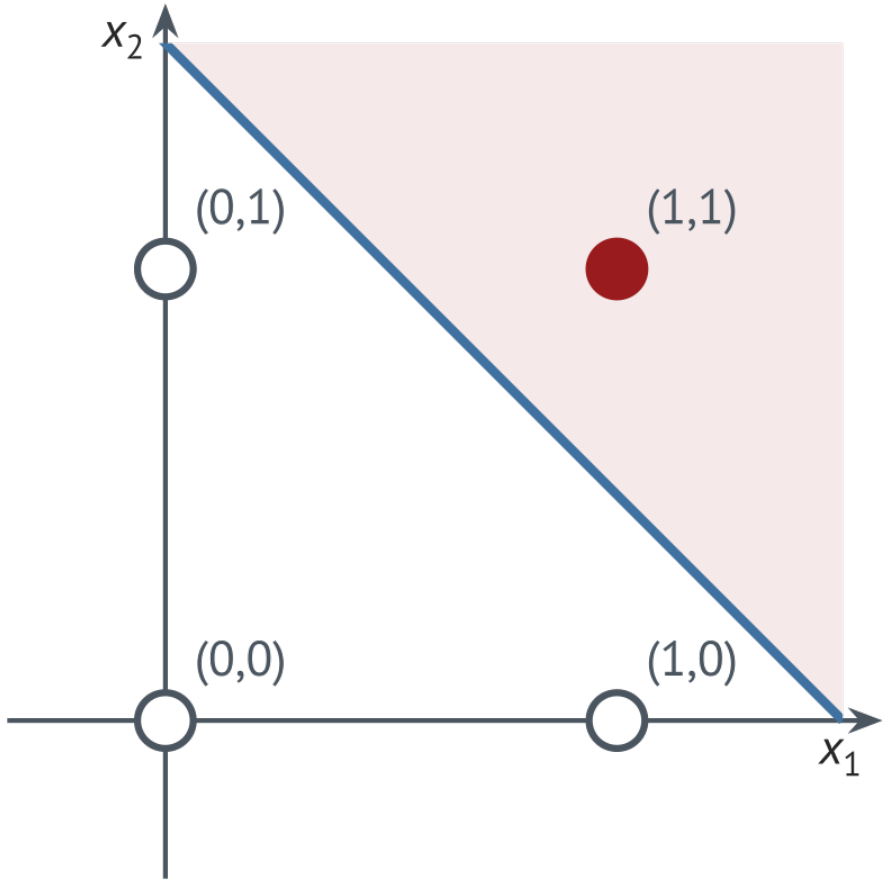
- How could we implement **AND**, **OR** and **NAND**?
- Find w_1, w_2 and b for each gate

TRY IT

Start with AND: which inputs need $z > 0$?

AND gate

With w = (1, 1) and b = −1.5, only (1, 1) lands on the positive side of the line.



Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

AND gate: weights and bias

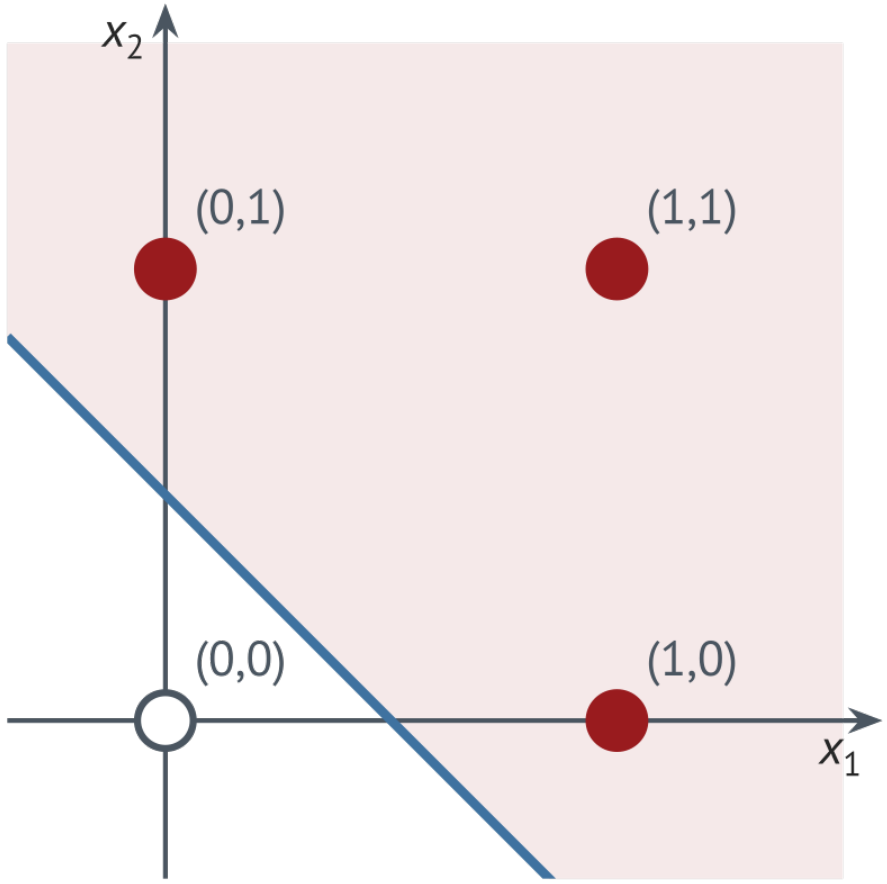
$$w_1 = 1, \quad w_2 = 1, \quad b = -1.5$$

x_1	x_2	$z = w_1x_1 + w_2x_2 + b$	y
0	0	0 + 0 − 1.5 = −1.5	0
0	1	0 + 1 − 1.5 = −0.5	0
1	0	1 + 0 − 1.5 = −0.5	0
1	**1**	**1 + 1 − 1.5 = 0.5**	**1**

Decision boundary: $x_1 + x_2 = 1.5$. Any b with $-2 < b \leq -1$ works; −1.5 is the middle.

OR gate

Lower the bias to −0.5 and the line moves down: any input with a 1 now gives y = 1.



Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

OR gate: weights and bias

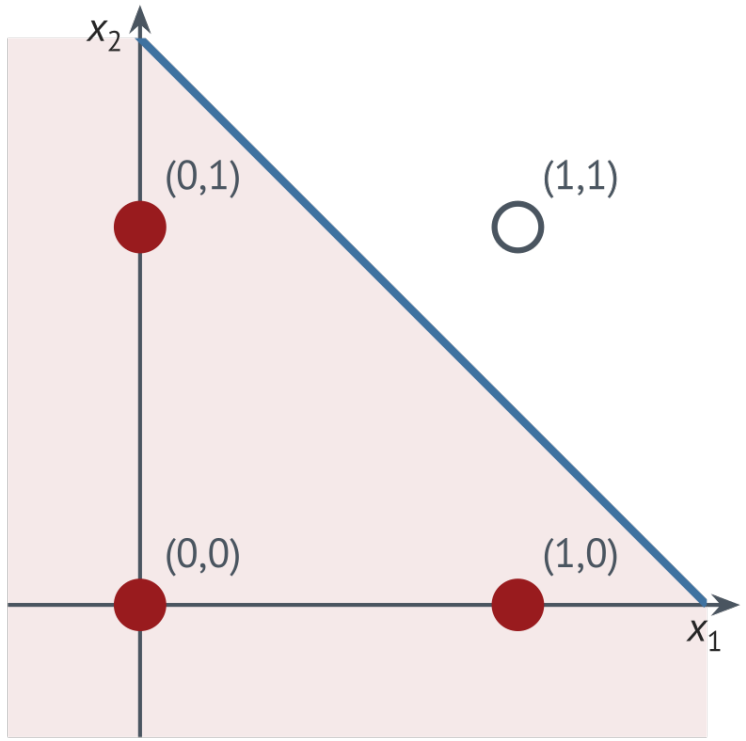
$$w_1 = 1, \quad w_2 = 1, \quad b = -0.5$$

x_1	x_2	$z = w_1x_1 + w_2x_2 + b$	y
0	0	0 + 0 − 0.5 = −0.5	0
0	**1**	**0 + 1 − 0.5 = 0.5**	**1**
1	**0**	**1 + 0 − 0.5 = 0.5**	**1**
1	**1**	**1 + 1 − 0.5 = 1.5**	**1**

Decision boundary: $x_1 + x_2 = 0.5$. Same weights as AND; only the bias moved.

NAND gate

Flip the sign of every weight and of the bias of AND, and every output flips: NAND.



Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

NAND gate: weights and bias

$$w_1 = -1, \quad w_2 = -1, \quad b = 1.5$$

x_1	x_2	$z = w_1x_1 + w_2x_2 + b$	y
0	**0**	**0 + 0 + 1.5 = 1.5**	**1**
0	**1**	**0 − 1 + 1.5 = 0.5**	**1**
1	**0**	**−1 + 0 + 1.5 = 0.5**	**1**
1	1	−1 − 1 + 1.5 = −0.5	0

NOT of AND: 1 except at (1, 1). Same line $x_1 + x_2 = 1.5$, with the sides swapped.

TRY IT **Now a NOR gate:** in the notebook's NOR cell, choose w_1, w_2 and b so the output is 1 only at (0, 0).

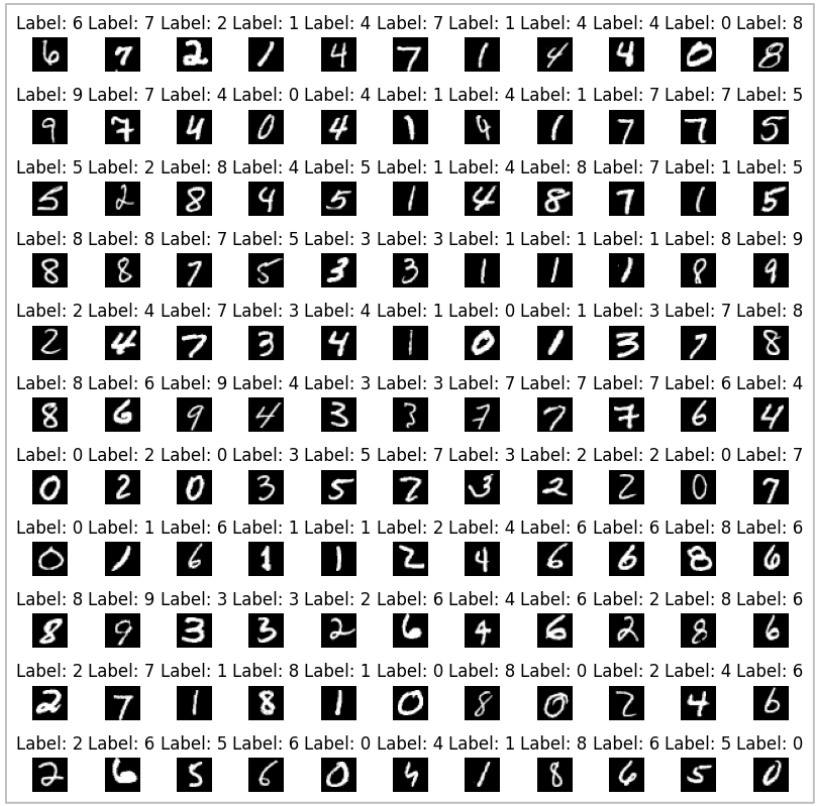
Part D

Mini-batch gradient descent: estimate the gradient from 128 images, update, repeat.

PART D

Goal: load MNIST in shuffled mini-batches and look at one batch.

1. **Data loading**: torchvision downloads **MNIST**, 60,000 training and 10,000 test images (fallback: MNIST_ORG.zip)
2. Each pixel is scaled to [0, 1], then (x − 0.5)/0.5 maps it to [−1, 1]
3. Display one shuffled **mini-batch** of 128 as an 11 × 11 grid



One mini-batch of 128 (the notebook shows 121 in an 11 × 11 grid)

Check: one batch has the shape **[128, 1, 28, 28]** (128 images, 1 channel, 28 × 28 pixels), and an epoch has 469 batches.

Part D: Data

torchvision downloads MNIST; Normalize maps the pixels to [−1, 1]; a DataLoader serves batches.

```
transform = transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])  # [0, 1] -> [-1, 1]

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

trainloader = DataLoader(train_dataset,
                  batch_size=128, shuffle=True)
```



60,000 training + 10,000 test images

- **MNIST**: LeCun et al. (1998), handwritten digits 0 to 9
- **torchvision.datasets.MNIST**: downloads the digits to ./data (fallback: MNIST_ORG.zip)
- **torch.utils.data.DataLoader**: used everywhere, gives shuffled batches of the training set

Epoch: one pass through the training set

$$\frac{60{,}000 \text{ images}}{128 \text{ per batch}} \approx 469 \text{ batches}$$

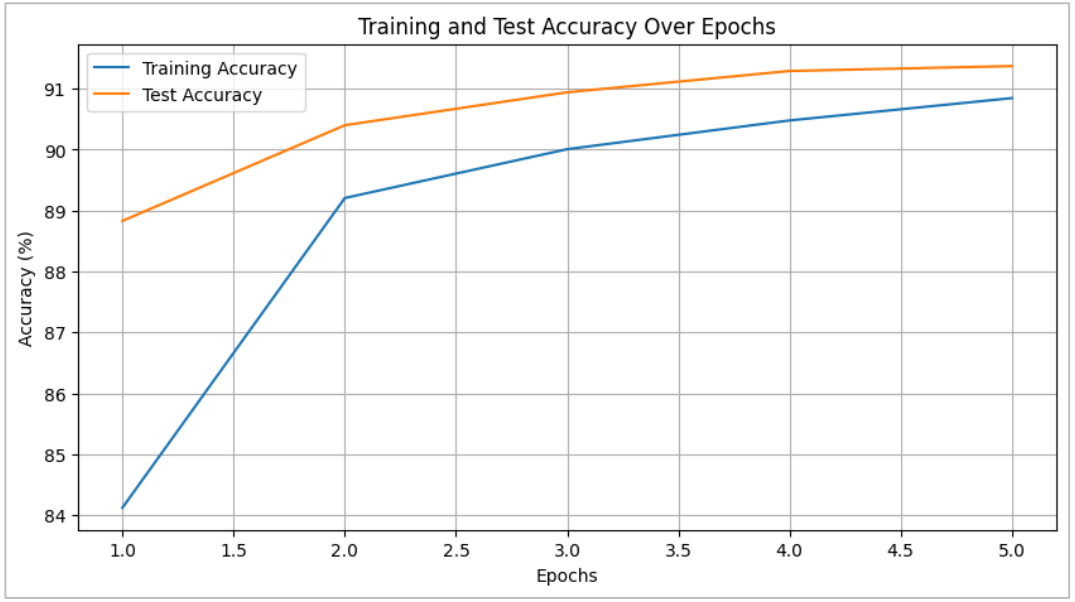
Part E

Define the network, pick a loss and an optimizer, then train and evaluate.

PART E

Goal: train a one-layer network on MNIST to about 91% test accuracy.

1. **Define the neural network**: SimpleNet, one nn.Linear(784, 10) layer
2. **Loss function and optimizer**: nn.CrossEntropyLoss and optim.SGD with lr = 0.02
3. **Training and eval**: 5 epochs with train().

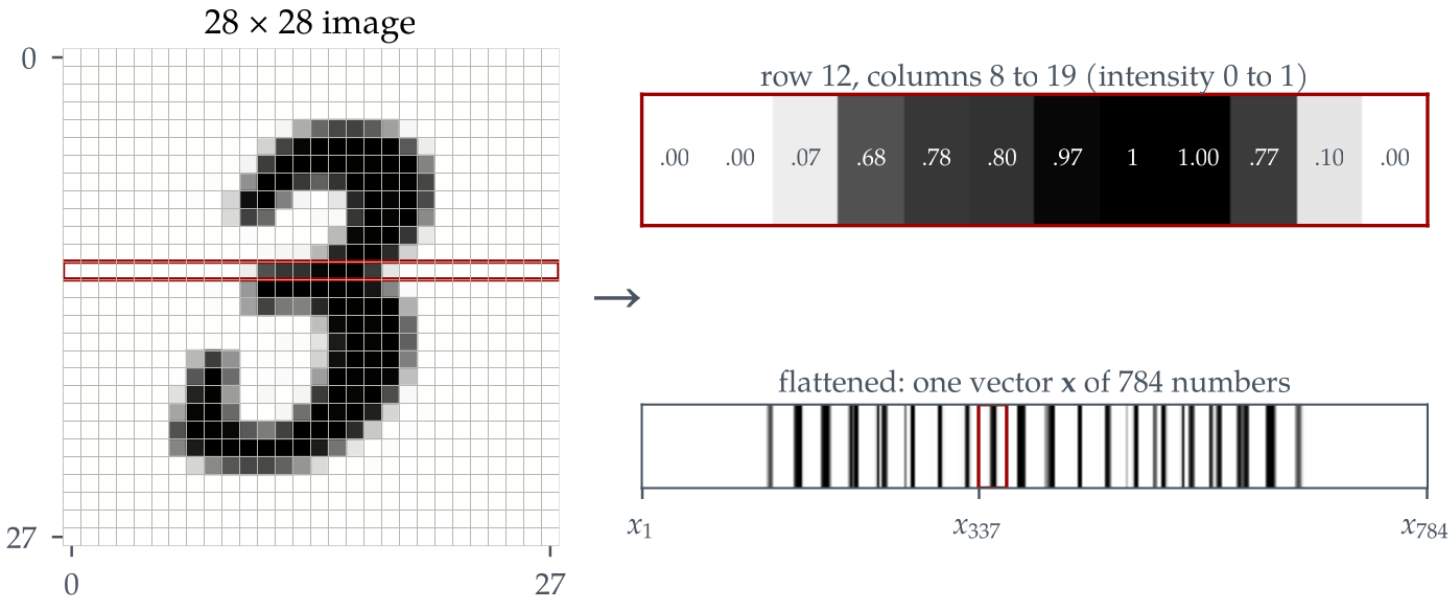


One run: accuracy over 5 epochs

Check (Checkpoint 2): the one-layer network reaches about 91% on the test set.

Operation of a neural network

An image is a grid of pixels; each pixel becomes one input, connected to every hidden neuron.



Row 12 of a 28 × 28 digit with its values; reading all 28 rows gives the input vector x of 784 numbers.

- An image is made of a set number of pixels: 28 × 28 = **784**
- **Each pixel becomes an input**
- 8-bit greyscale: each pixel is 0 to 255, scaled to 0 to 1
- Each input connects to **all** the neurons of the hidden layer
- Adjusting the weights eventually lets the network recognize the digits

Pixel value to input

$$x_i = \frac{p_i}{255}, \qquad p_i = 0, 1, \ldots, 255$$

Source: becominghuman.ai

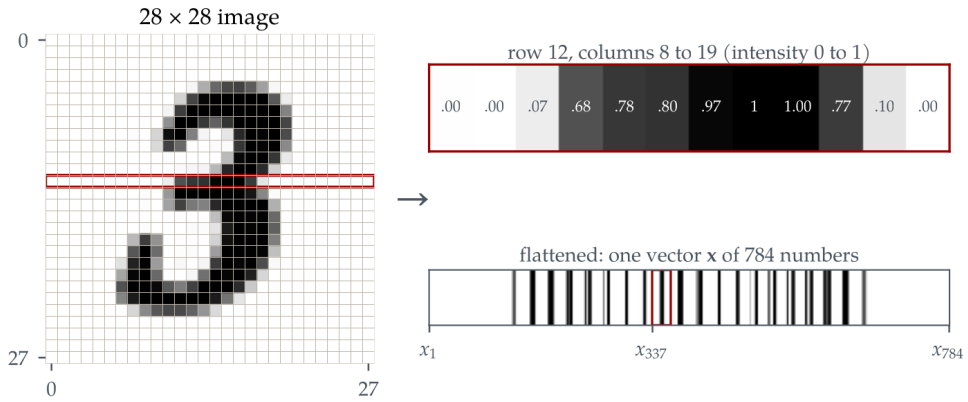
Part E: NN setup

SimpleNet flattens each image to 784 numbers and maps them to 10 class scores.

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.linear = nn.Linear(28 * 28, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)    # flatten to 784
        return self.linear(x)      # 10 scores

model = SimpleNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.02)
```



x.view(-1, 28 * 28): rows laid end to end

Parameters in nn.Linear(784, 10)

784 × 10 weights + 10 biases = 7,850

- **nn.Linear**: both weights and biases included
- **nn.CrossEntropyLoss**: used for multi-class classification
- **optim.SGD**: stochastic gradient descent

Part E: NN training

Every batch: clear the gradients, forward pass, loss, backward pass, update.

```
for epoch in range(epochs):                    # epochs = 5
    model.train()
    for images, labels in loader:              # 469 batches
        optimizer.zero_grad()      # clear previous gradients
        outputs = model(images)                # forward pass
        loss = criterion(outputs, labels)    # compute the loss
        loss.backward()            # backward pass
        optimizer.step()           # update weights and biases

    model.eval()
    with torch.no_grad():          # testing: no gradients
        outputs = model(X_test)    # all 10,000 test images
        _, predicted = torch.max(outputs, 1)
    test_acc = (predicted == y_test).float().mean()
```

Epoch	Loss	Training	Test
1	0.613	84.1%	88.8%
2	0.386	89.2%	90.4%
3	0.352	90.0%	90.9%
4	0.335	90.5%	91.3%
5	0.324	90.8%	91.4%

SimpleNet, SGD, lr = 0.02: one run. Results vary by about a point between runs.

Loss falls and accuracy levels off near **91%**: one linear layer has reached its limit.

Part F

A network only knows its training data; a photo of your handwriting looks different.

PART F

Goal: classify a photo of your own handwritten digit with the trained SimpleNet.

1. Write a digit in thick dark ink, photograph it, crop it to the paper
2. Save it as a JPEG in the notebook's folder and put its file name in **image_path**
3. Run **predict_image**: grayscale, 28 × 28, normalize, invert, gamma, **prediction**
4. Try another digit: change the file name in image_path

Distribution shift

MNIST training digit	Your phone photo
White digit on black	Dark ink on white paper
Thick, smooth strokes	Thin pen lines
Centered, fills the frame	Any size, any position
Even, clean background	Shadows and paper texture

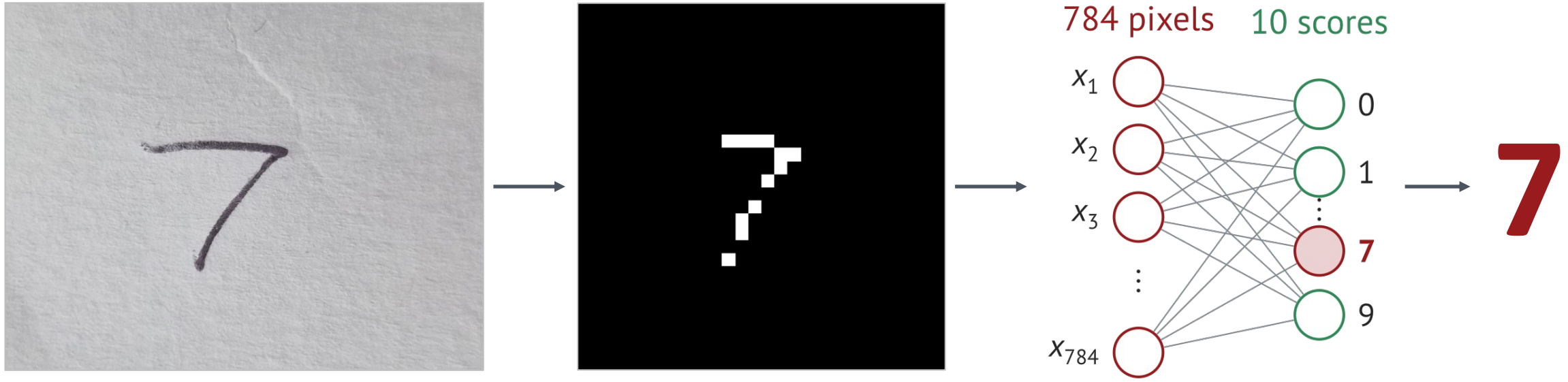
Every row is a difference the network never saw in training.

Check: the notebook prints the predicted label; if it is wrong, explain why with the table.

Part F: custom NN inference

USC University of Southern California

Photo, then a 28 × 28 input, then the network: the highest of 10 scores is the answer.



Phone photo of a digit | 28 × 28 input | SimpleNet, 784 to 10 | Prediction

predict_image

Grayscale, resize to 28 × 28, scale to [−1, 1], invert, gamma correction

Distribution shift

The thin pen line breaks into a few pixels; MNIST strokes are thick and solid

Parts G and H

USC University of Southern California

A hidden layer and a convolutional network beat the one-layer model; then tune them.

PARTS G, H

Goal: train deeper networks and find which parameter change helps most.

1. **Train a 1-hidden-layer NN**: SimpleNetWithHiddenLayer (784-128-10), SGD, 10 epochs
2. **Train a CNN**: SimpleCNN, Adam, 10 epochs
3. **Eval both networks** on your own digit with predict_image
4. **Experiment with different parameters**: hidden units, learning rate, epochs

Experiment	Hidden units	Learning rate	Epochs	Test acc.
Baseline	128	0.001	5	96.7%
Higher learning rate	128	0.01	5	93.5%
More hidden units	256	0.001	5	95.5%
More epochs	128	0.001	10	96.1%

Typical final test accuracy (Adam, all 60,000 training images). Yours will be within about a point.

Check (Checkpoint 3): at least one digit you wrote is predicted correctly, and you can say which parameter change helped most.

Part G: Better architectures

USC University of Southern California

Both models end in nn.Linear layers; the CNN first finds local features with 3 × 3 filters.

784-128-10: **101,770** parameters

```
class SimpleNetWithHiddenLayer(nn.Module):
    def __init__(self):
        super(SimpleNetWithHiddenLayer,
              self).__init__()
        self.hidden = nn.Linear(28 * 28, 128)
        self.relu = nn.ReLU()
        self.output = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)    # 784
        x = self.relu(self.hidden(x))
        x = self.output(x)         # 10
        return x
```

SimpleCNN: **206,922** parameters

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):                          # 1x28x28
        x = self.pool(self.relu(self.conv1(x)))    # 16x14x14
        x = self.pool(self.relu(self.conv2(x)))    # 32x7x7
        x = x.view(-1, 32 * 7 * 7)                 # 1568
        x = self.relu(self.fc1(x))
        return self.fc2(x)
```

Typical: hidden layer **96.1%** after 10 epochs (SGD); CNN **96.9%** after 10 epochs (Adam). SimpleNet: 91.4%.

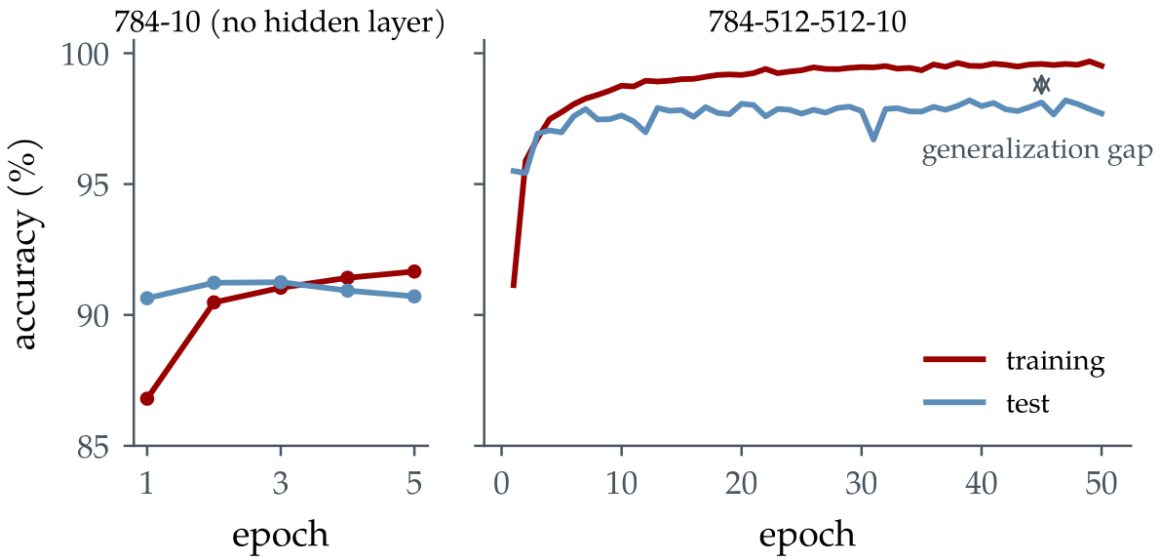
Part I

Underfit: both accuracies low. Overfit: training keeps rising while test stalls.

PART I

Goal: see underfitting and overfitting in the training and test curves.

1. **Underfitting**: a "simpler" model, fewer epochs (784-10, 5 epochs)
2. **Overfitting**: a "more complicated" model, 20 epochs on all 60,000 images (784-512-512-10, Adam)
3. Plot accuracy for both; find the best epoch (I1)



Notebook run, all 60,000 images. Left: both near 91%. Right: training above 99%, test near 98%.

Check (Checkpoint 4): both plots and an early-stopping epoch. Then submit your work through the task tracker.