Supervised Machine Learning and Learning Theory

Lecture 12: Convolutional neural networks

October 15, 2024

Warm-up questions

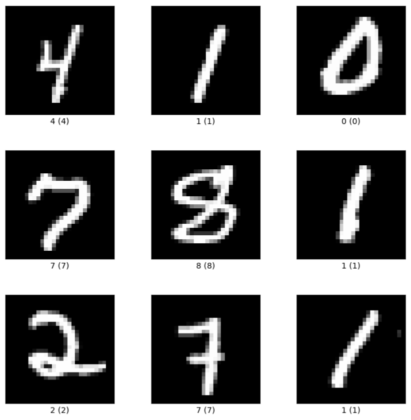
- What is the difference between random forests and gradient boosting?
- For random forests, how many features are usually selected to fit each tree?
- Name several choices of activation functions for designing an artificial neuron
- Describe the difference between sigmoid activation and perceptron

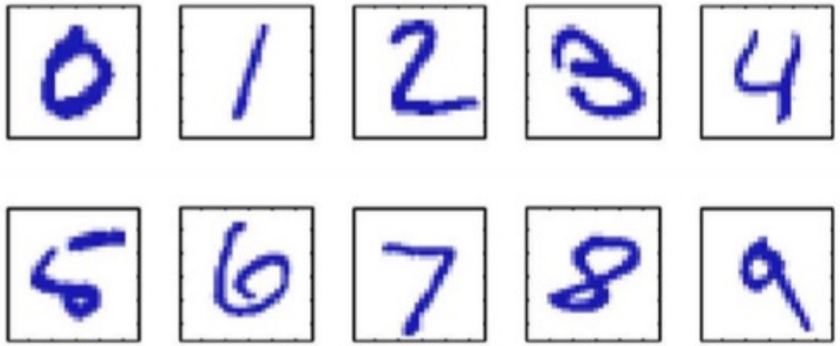
Lecture plan

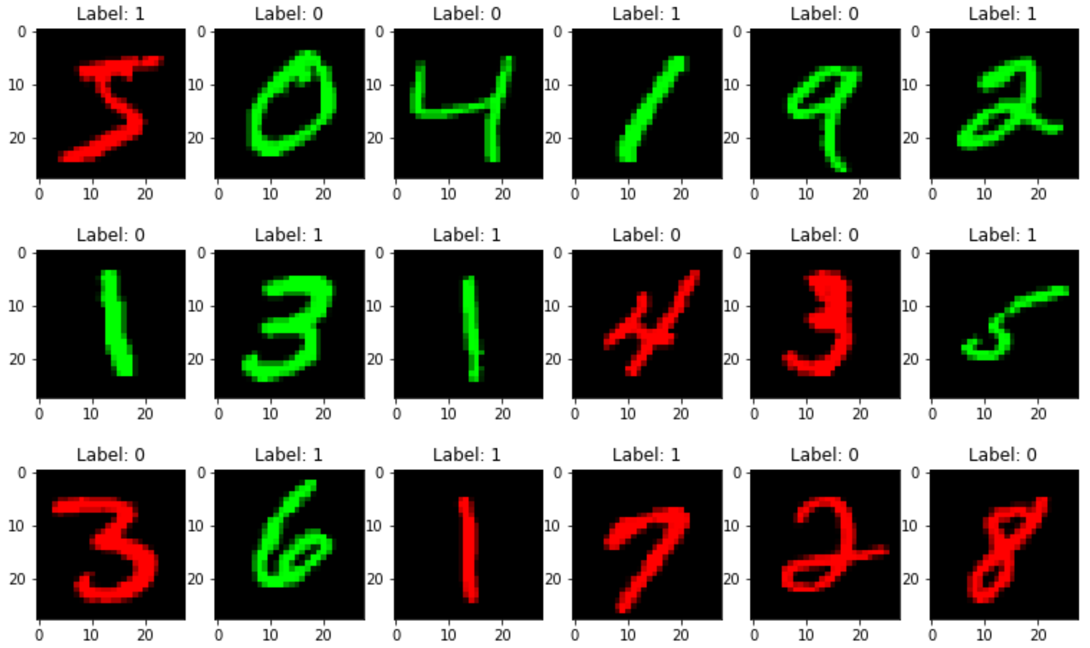
- **Convolutional layers**

Handwritten digit classification

- Classifying handwritten digits

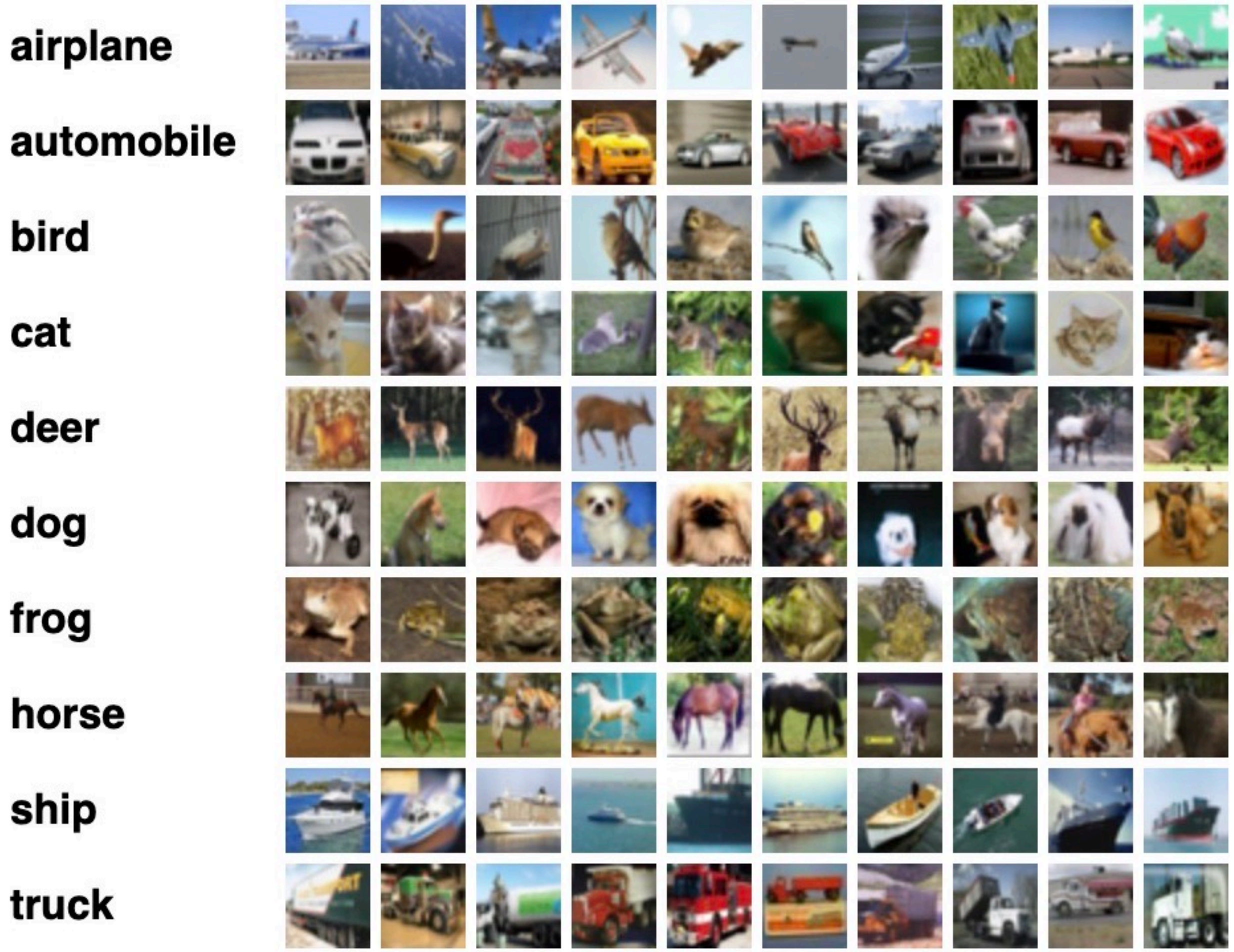






Object recognition

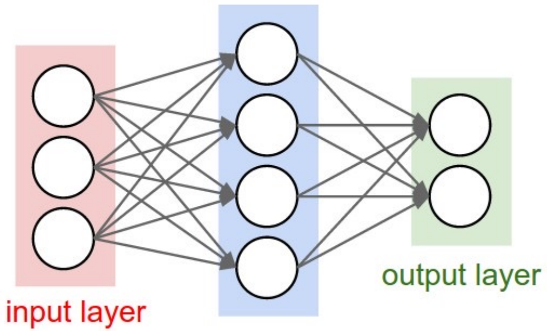
- CIFAR-10



Object recognition

- ImageNet (1000 classes)

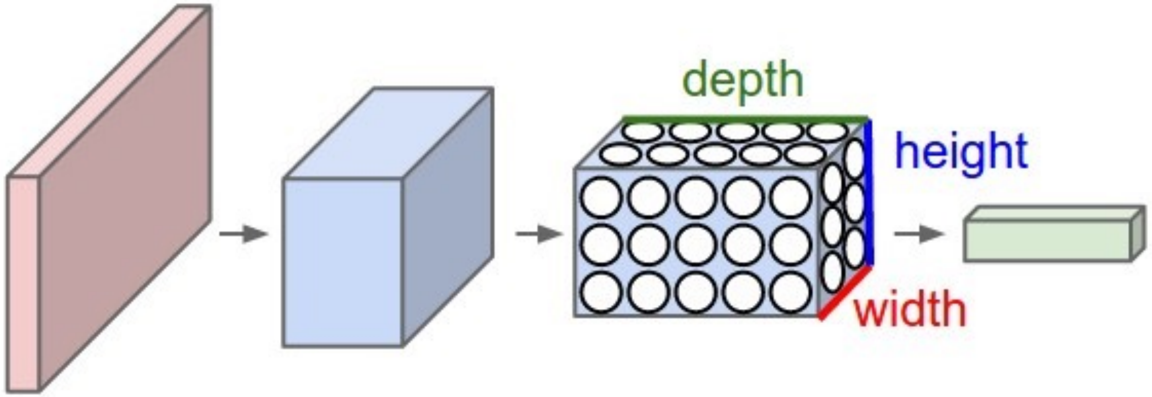
Issues of using feedforward neural networks for large images



- Feedforward neural networks use fully-connected layers to transform the input
- Fully-connected layers do not scale to large images
  - A black-and-white digit in MNIST has size 28 by 28. A colored image in CIFAR-10 has size 32 by 32 by 3
  - For MNIST, a fully-connected neuron needs $28 \times 28 = 784$ weights
  - For CIFAR-10, a fully-connected neuron needs $32 \times 32 \times 3 = 3{,}072$ weights
  - Processing larger images requires more parameters

CNN only uses local connections

- In convolutional neural networks (CNN), a neuron only connects to a small local region of the image
  - Example: A colored (2D) image is specified by width, height, and depth



Types of layers

- A CNN involves a combination of the following types of layers

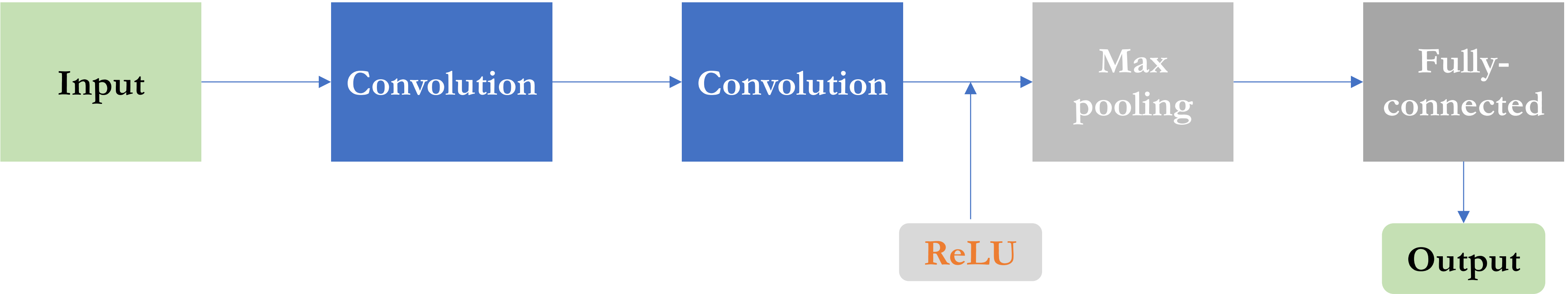
  - **Input layer:** Raw pixel values of the image

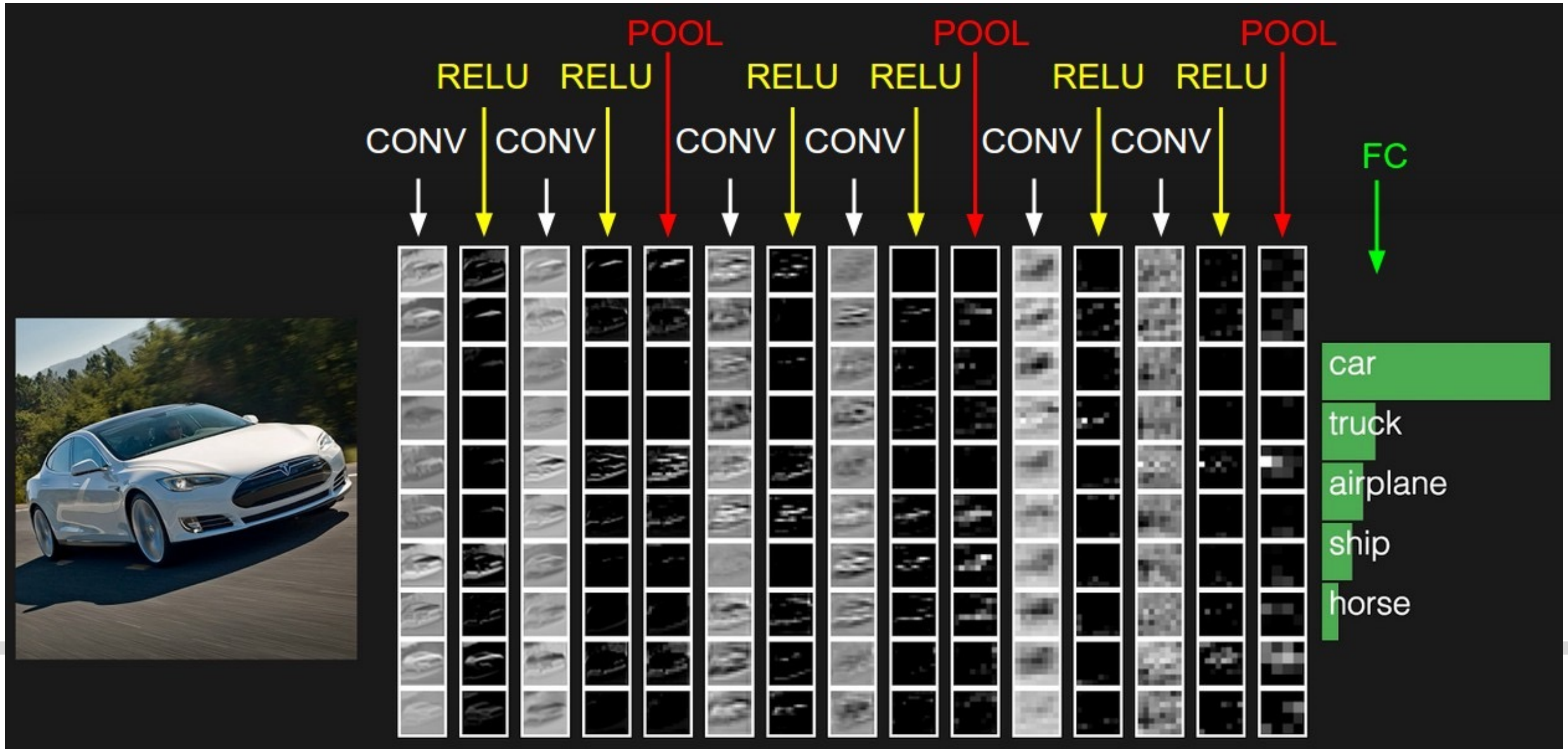
  - **Convolution layer:** Combine pixel values in a local region

  - **Pooling layer:** Down sample pixels

  - **Fully-connected layers:** Classification/prediction

Illustration of CNN architectures



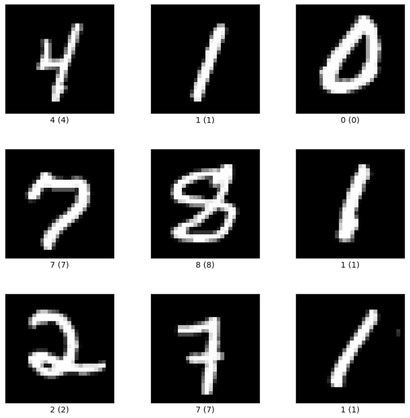


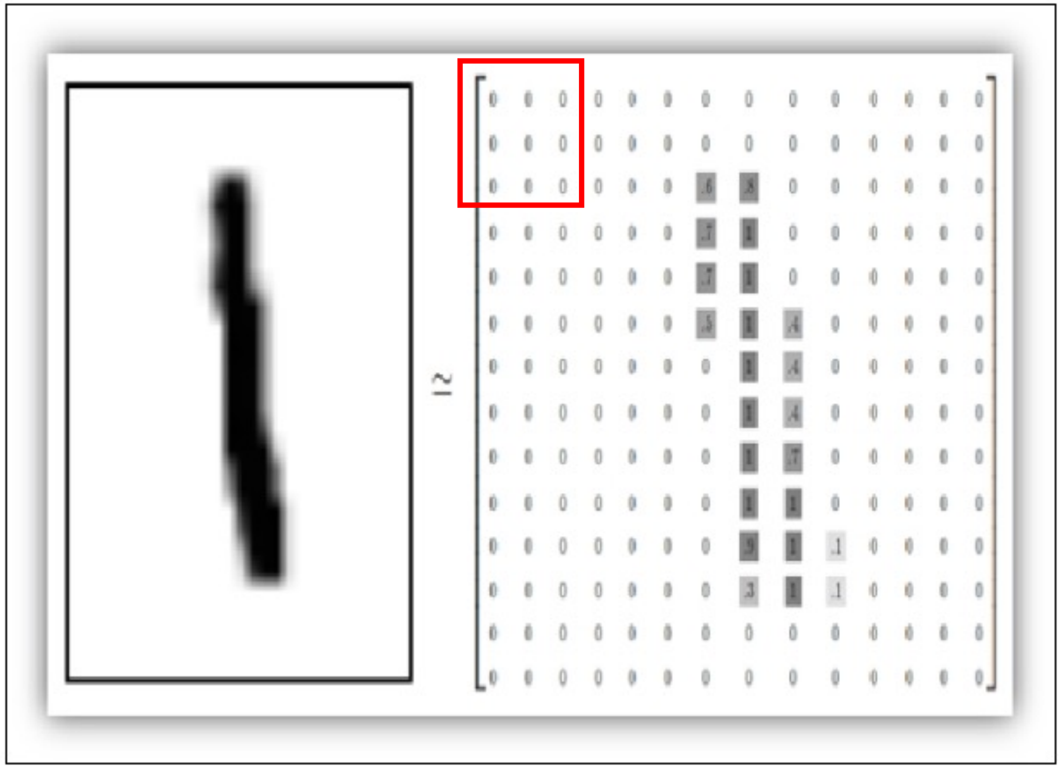
Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28

  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0

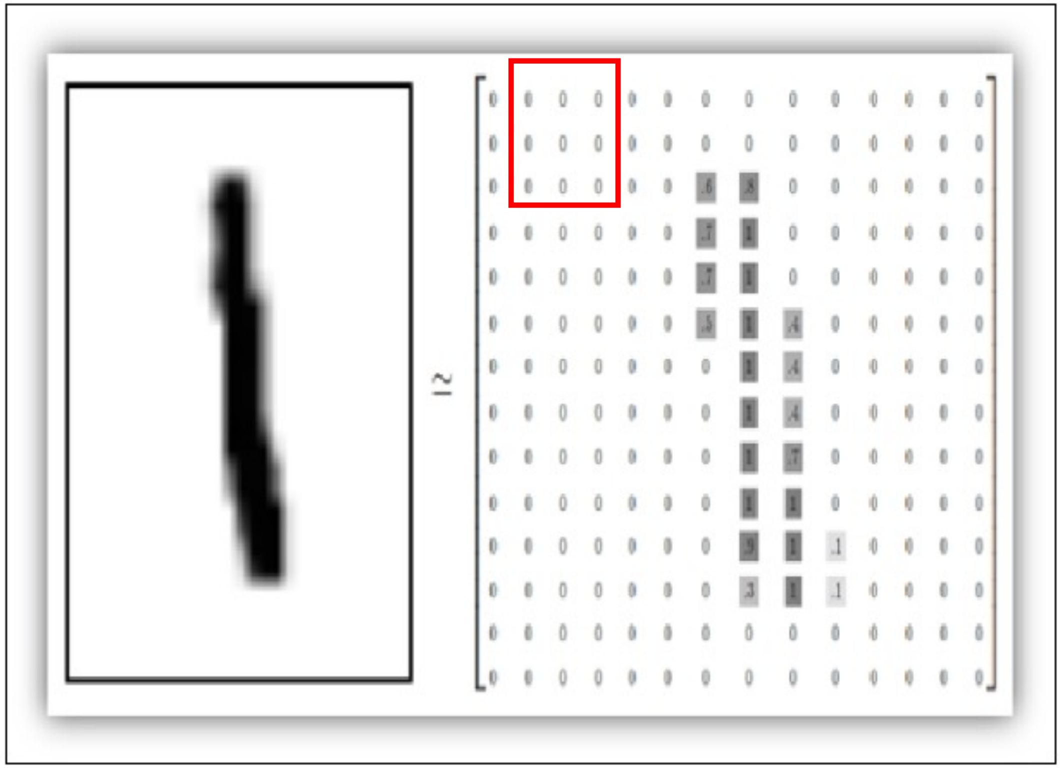
  - **First row, first patch**





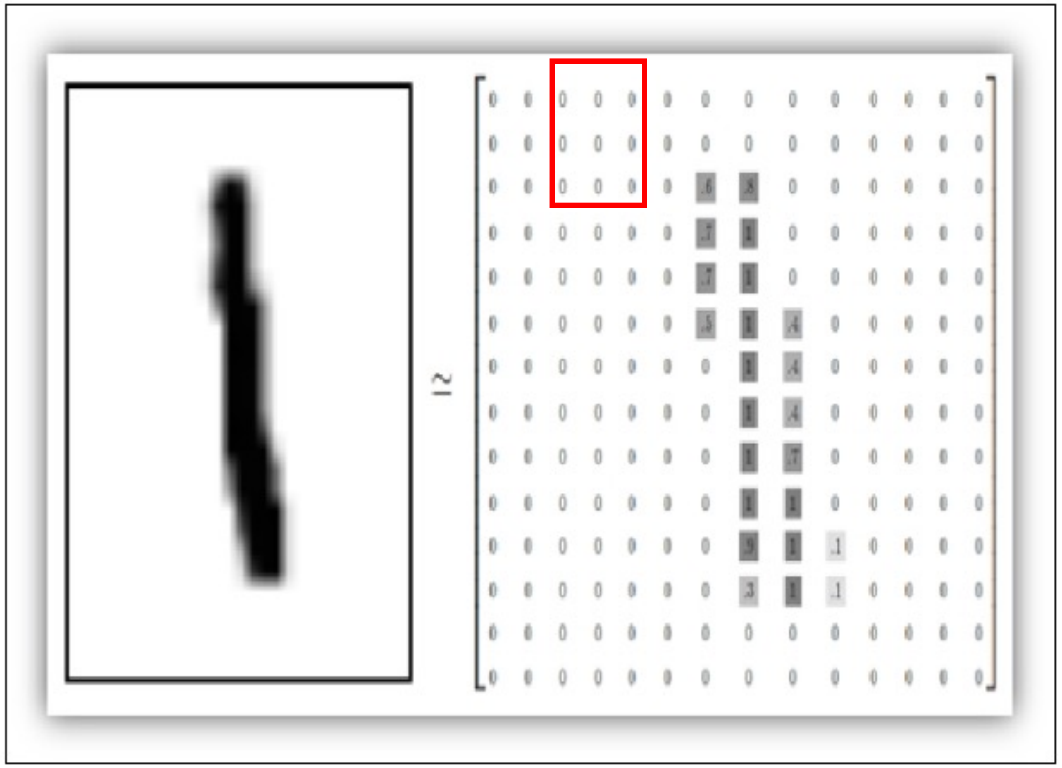
Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0
  - **First row, second patch**



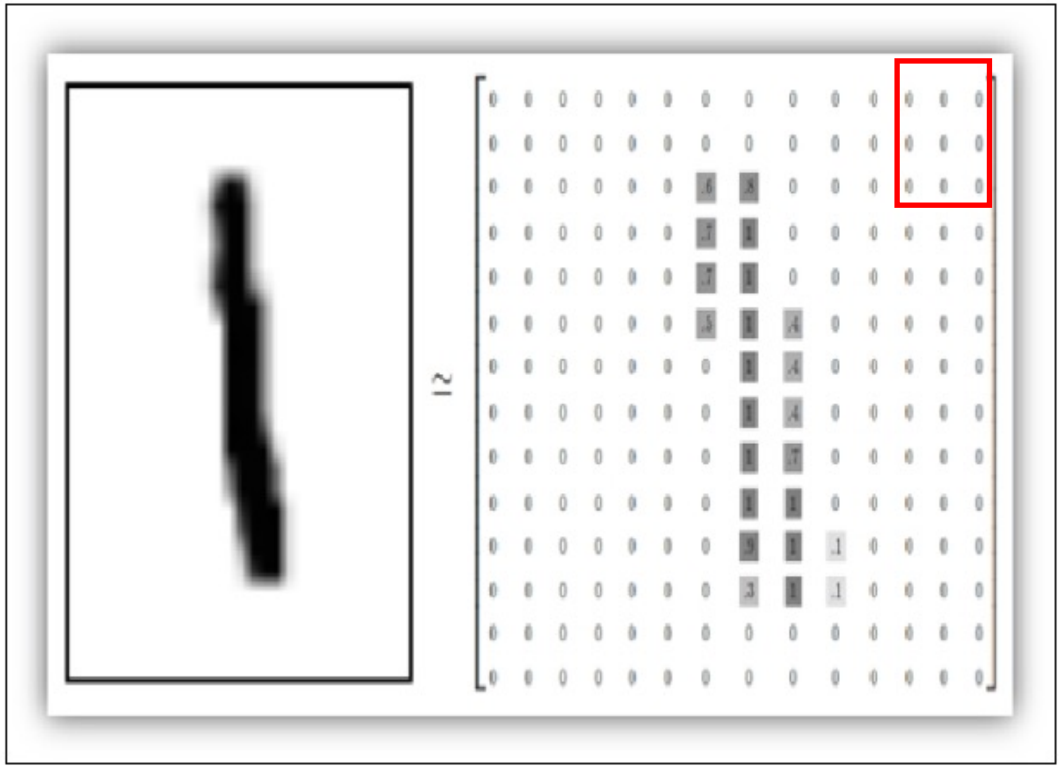
Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0
  - **First row, third patch**



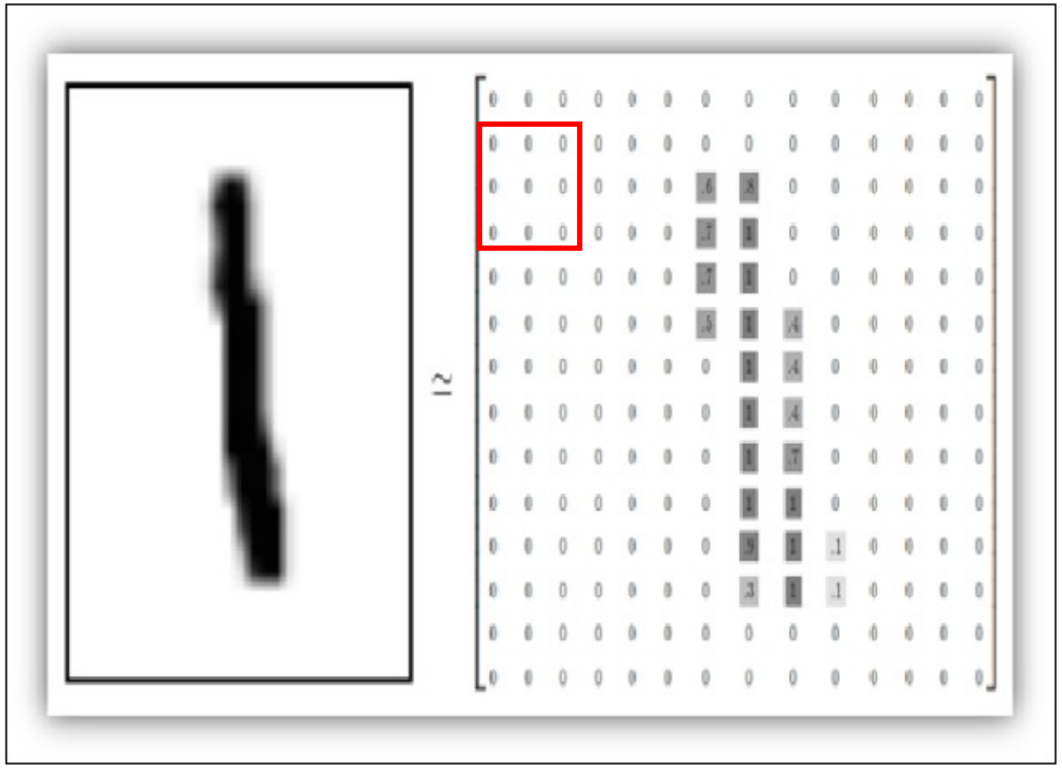
Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0
  - **First row, last patch**



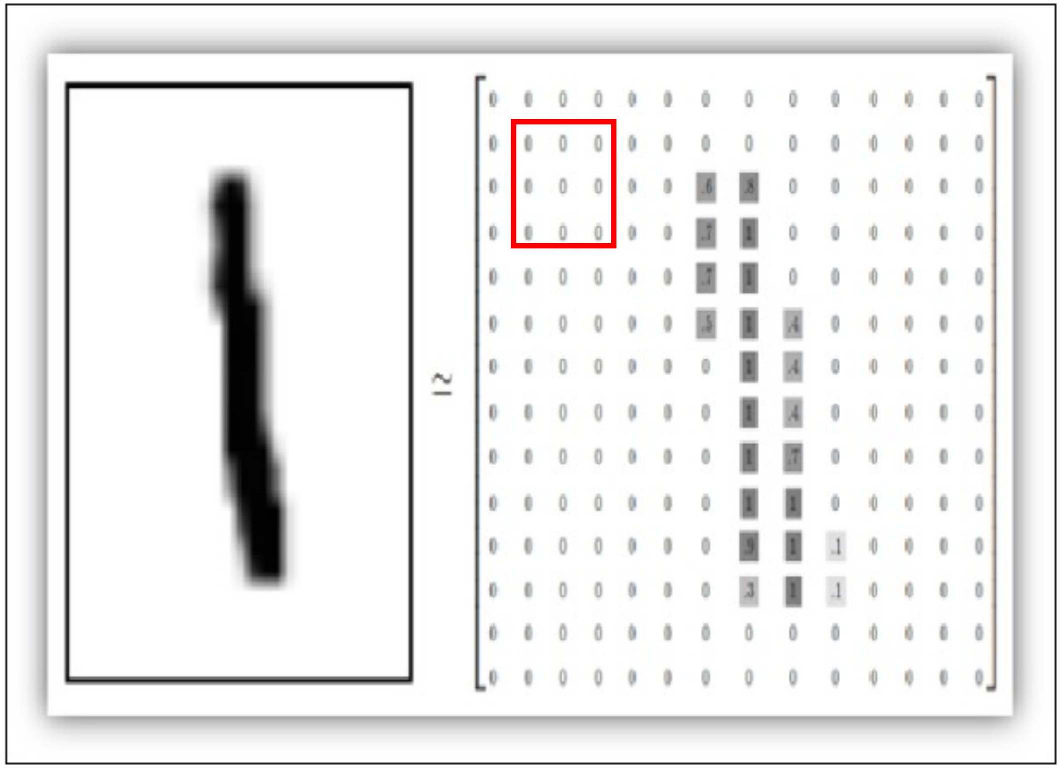
Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 1
  - **Second row, first patch**



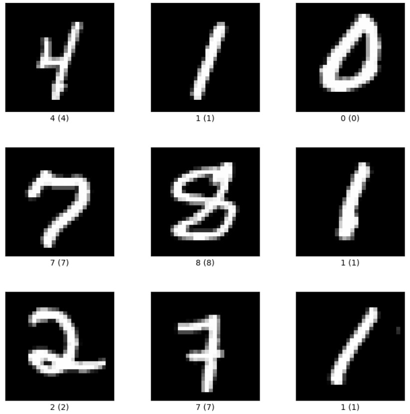
Convolution layer

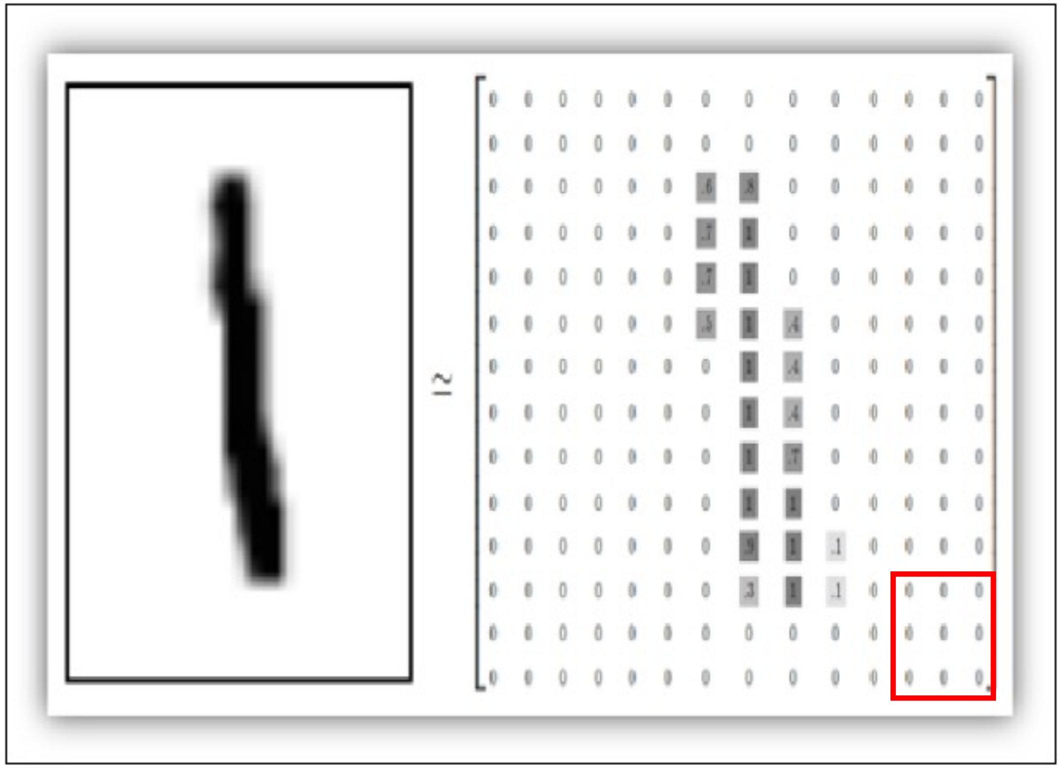
- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0
  - **Second row, second patch**



Convolution layer

- **Example** (MNIST)
  - **Input** size: 28 by 28
  - **Convolutional layer:**
    - Filter size: (3, 3)
    - Stride: (1, 1)
    - Zero padding size: 0
  - **Last row, last patch**





- **Question:** What is the final output size?

Convolution layer

- **Filter (depth times width):** Larger filter captures coarser spatial patterns, while smaller filters capture finer spatial patterns

- **Stride (depth times width):** How often do we slide the filter? For example, when the stride is 1, we slide the filter one pixel at a time

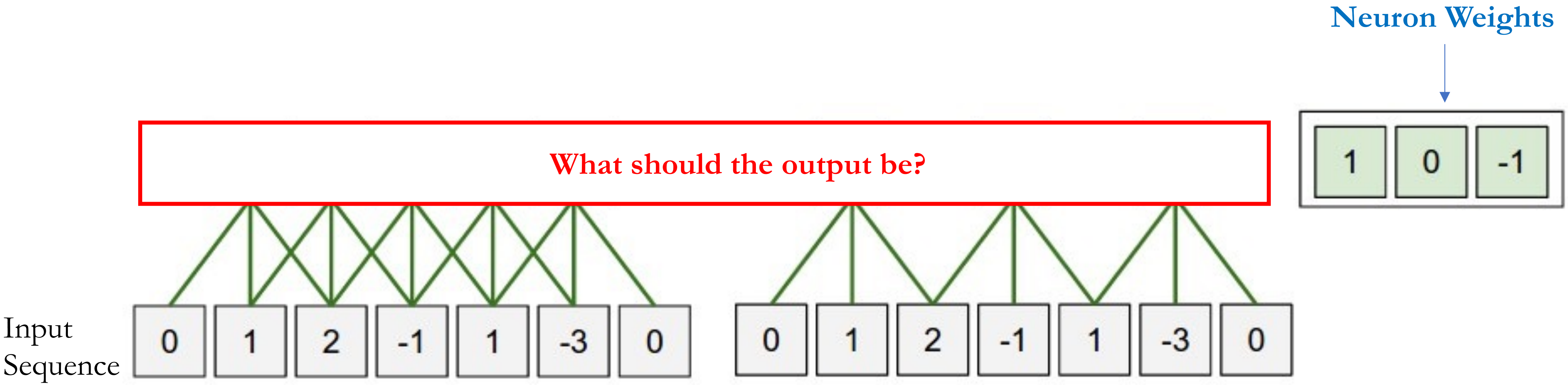
- **Zero padding:** Pad the input with zeros around the border

- MNIST example: filter size (3, 3), stride size (1, 1), zero padding size 0
  - Question: Suppose we want to preserve the spatial size of the input so that the input and output have the same size. What should we set as the zero padding size?

Illustration

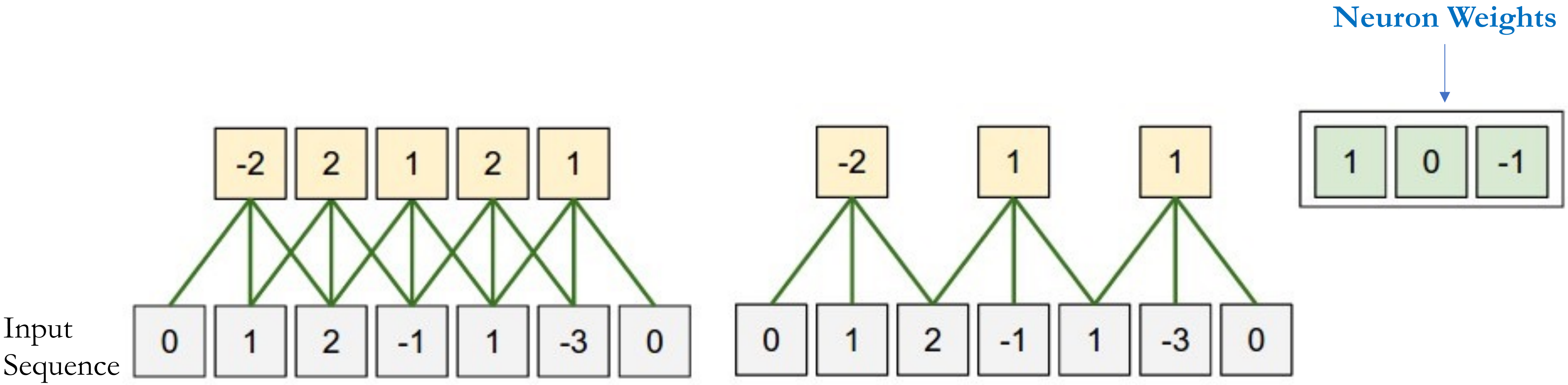
- Input dimension is one, filter size is (3), stride is (1)

- Multiply the input with the neuron weights pixel-by-pixel



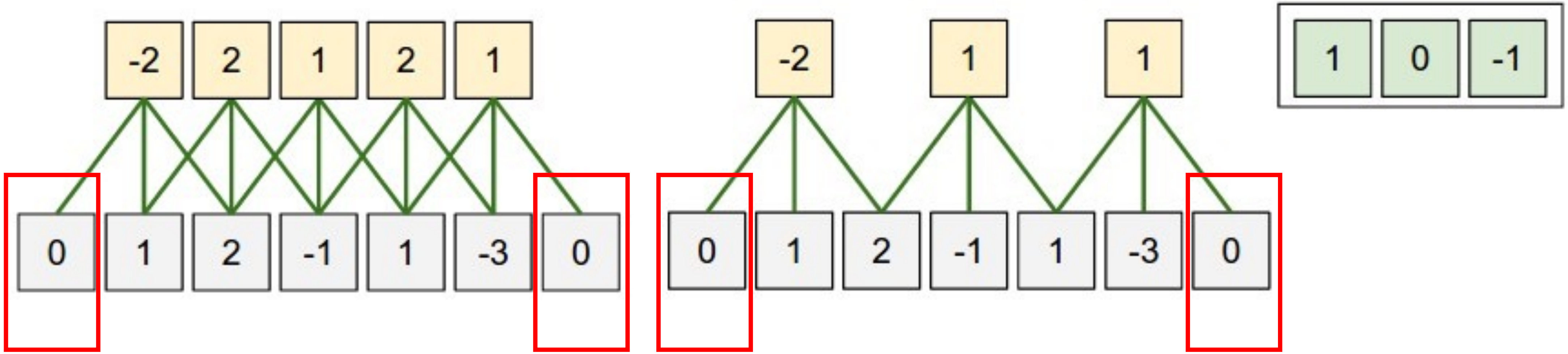
Illustration

- Illustration of spatial arrangement with a simplified example
  - Filter size is (3)
  - Stride is (1)



Explaining zero padding size

- This example uses **a single zero padding** on **both left and right**



- We can use zero padding to adjust the output dimension, e.g., in sentence classification, use zero padding for fixed (max) length sentences

Stride size

- **Constraints**

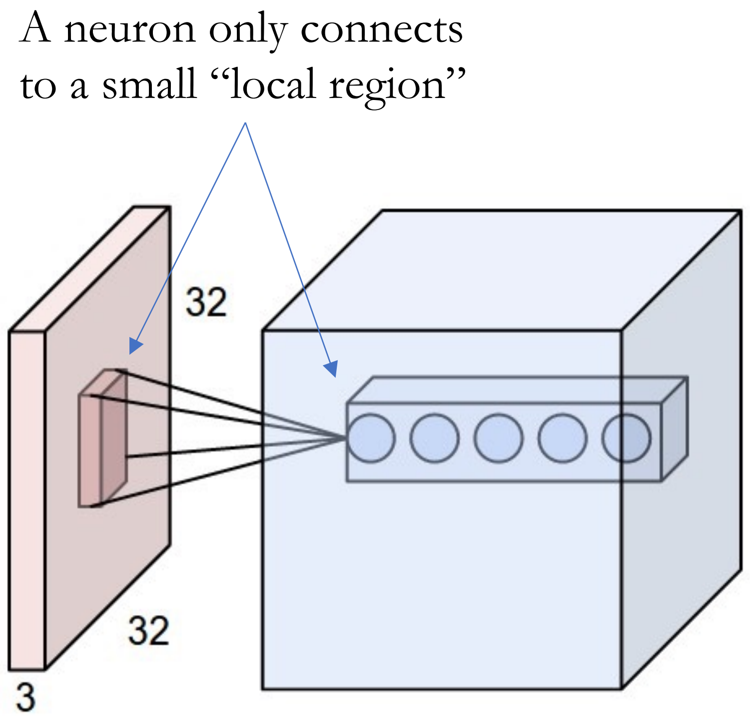
  - Filter size and stride size must satisfy that: (image width – filter size) should be divisible by (stride size)

  - **Otherwise, add zero padding**

  - **Question:** What goes wrong if this constraint is not satisfied?

Example (CIFAR-10)

- Illustrating the convolution operation for an image of size $(32, 32, 3)$



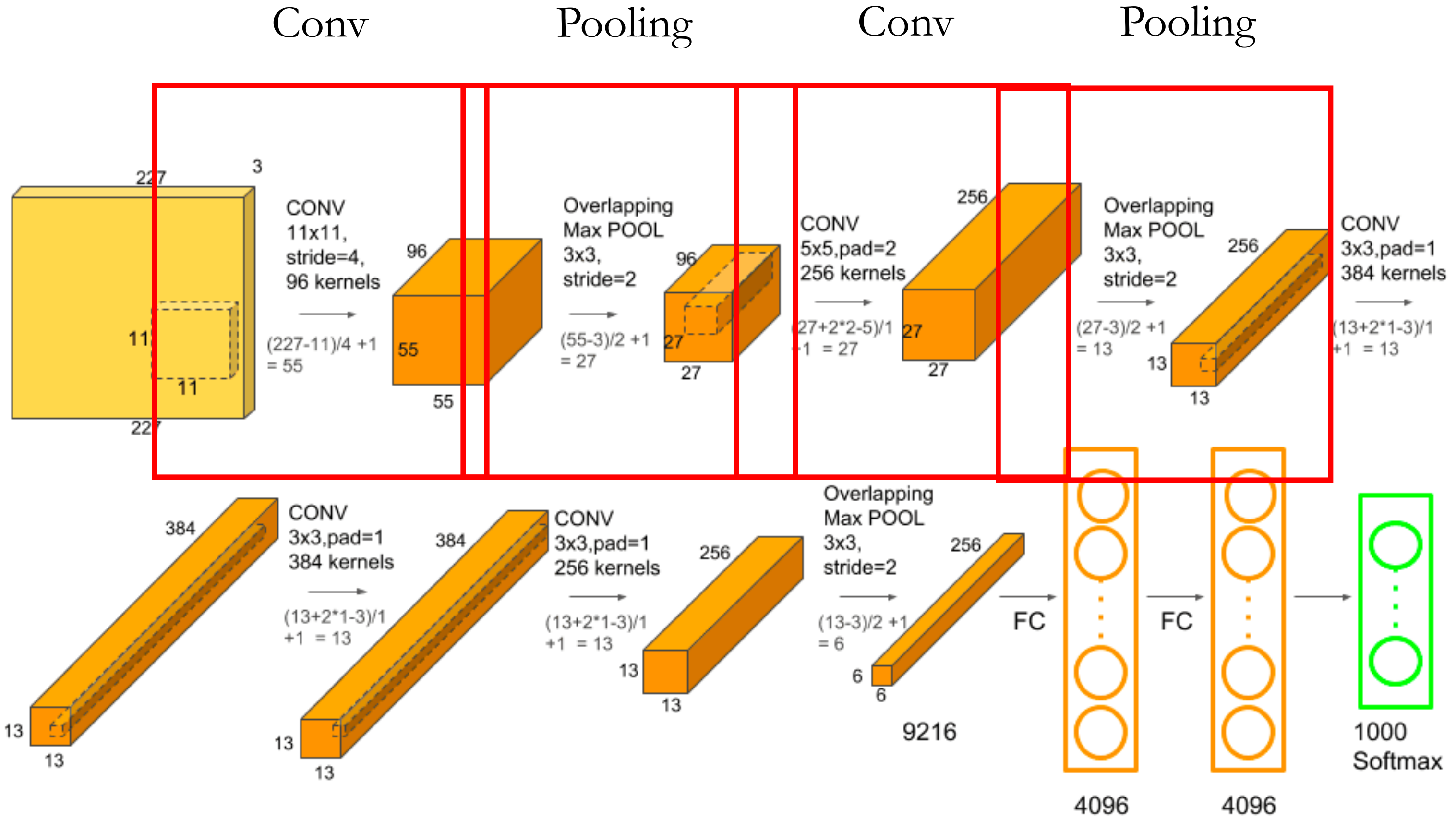
  - Within each neuron, perform convolution with possible nonlinear activation
- **Question:** can you specify a convolution layer configuration for CIFAR-10?

Example (Image

- ImageNet: Each image has size (227, 227, 3)
- AlexNet (2012), led by Goeff Hinton at Google
  - First convolution layer uses
    - **Filter size**: 11 by 11 by 3
    - **Stride**: 4 by 4
    - **Zero-padding**: 0
    - (227 – 11) is divisible by 4
  - Number of different filters is 96
  - **Question: Final output size?**
    - (227 – 11) / 4 + 1 = 55: 55 by 55 by 96

Nobel prize in physics 2024!!

Example (ImageNet)



Comparison of number of parameters

- In ImageNet, each image has size (227, 227, 3)

  - If we use a fully-connected layer: Suppose there are 100 filters, the total number of parameters is 227*227*3*100; this is very large

  - If we use a convolution layer: 11*11*3*100=36,300

- **Key idea:** parameter sharing, i.e., we use the same parameters in every filter
  - Leverages the geometry already present in visual images

Summary

- Input: A 3D image of size (W_1, H_1, D_1)
- Convolution layer:
  - Number of filters K
  - Filter size F $(F\times F\times D_1)$
  - Stride size S
  - Zero padding size P
- Produces an output of size (W_2, H_2, D_2). What is it?
  - $W_2 = \frac{W_1 - F + 2P}{S} + 1$
  - $H_2 = \frac{H_1 - F + 2P}{S} + 1$
  - $D_2 = K$
- With parameter sharing, $F\times F\times D_1$ weights per filter, for a total of $(F^2\times D_1)\times K$ weights

Numpy example

- **Input: numpy array $\boldsymbol{X}$**
  - $X.shape = (11,11,4)$
- **Convolution layer**
  - Number of filters: $K = 2$
  - Filter size: $5\times5\times4$
  - Stride size: 2×2
  - Zero padding size: 0
- **Output:** Denote as V
  - Output width and height: $\frac{11-5}{2} + 1 = 4$
  - Depth: 2

Numpy example

- First depth slice, along the first column: Filter parameters W_0, Bias b_0.
  $W_0.shape = (5, 5, 4)$

  - $V[0,0,0] = np.sum(X[:5, :5, :] * W_0) + b_0$

  - $V[1,0,0] = np.sum(X[2:7, :5, :] * W_0) + b_0$

  - $V[2,0,0] = np.sum(X[4:9, :5, :] * W_0) + b_0$

  - $V[3,0,0] = np.sum(X[6:11, :5, :] * W_0) + b_0$

Numpy example

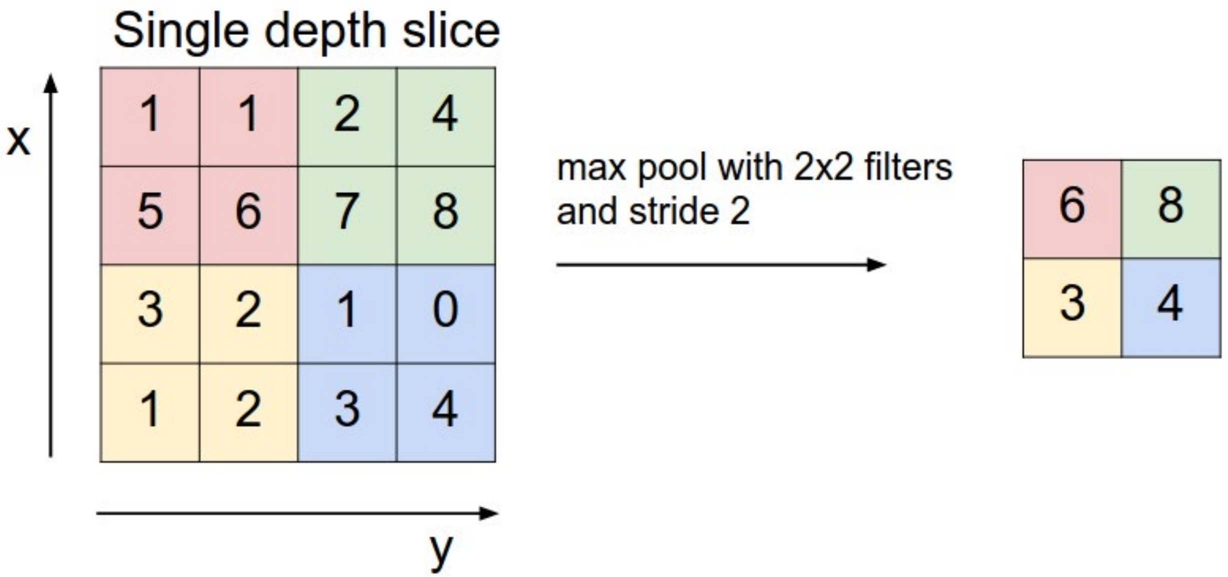
- For a different neuron: Filter parameters W_1, bias b_1
  - $V[0,0,1] = np.sum(X[:5, :5, :] * W_1) + b_1$
  - $V[1,0,1] = np.sum(X[2:7, :5, :] * W_1) + b_1$
  - $V[2,0,1] = np.sum(X[4:9, :5, :] * W_1) + b_1$
  - $V[3,0,1] = np.sum(X[6:11, :5, :] * W_1) + b_1$
  - Question: how do we calculate $V[0,1,1]$ and $V[2,3,1]$?

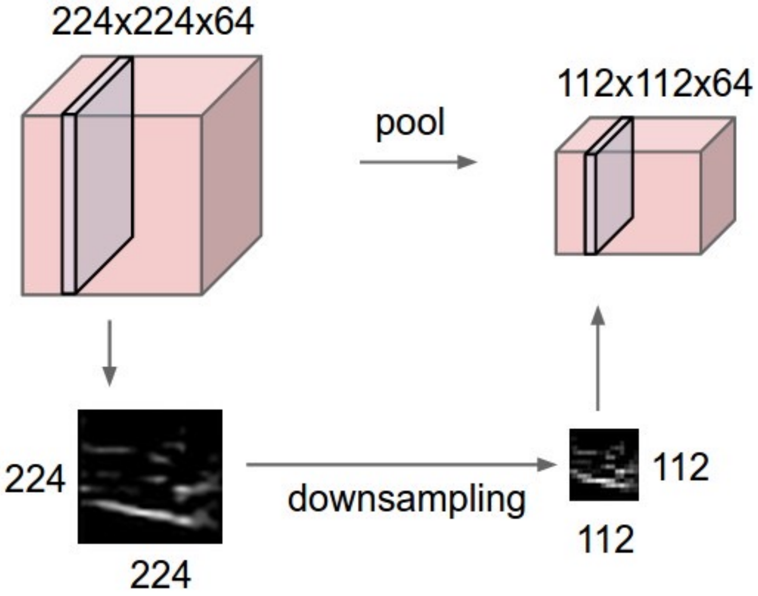
Lecture plan

- **Pooling layers**

Pooling layer

- **Pooling** reduces the spatial size of the input: Insert a pooling layer between convolution layers





Pooling layer

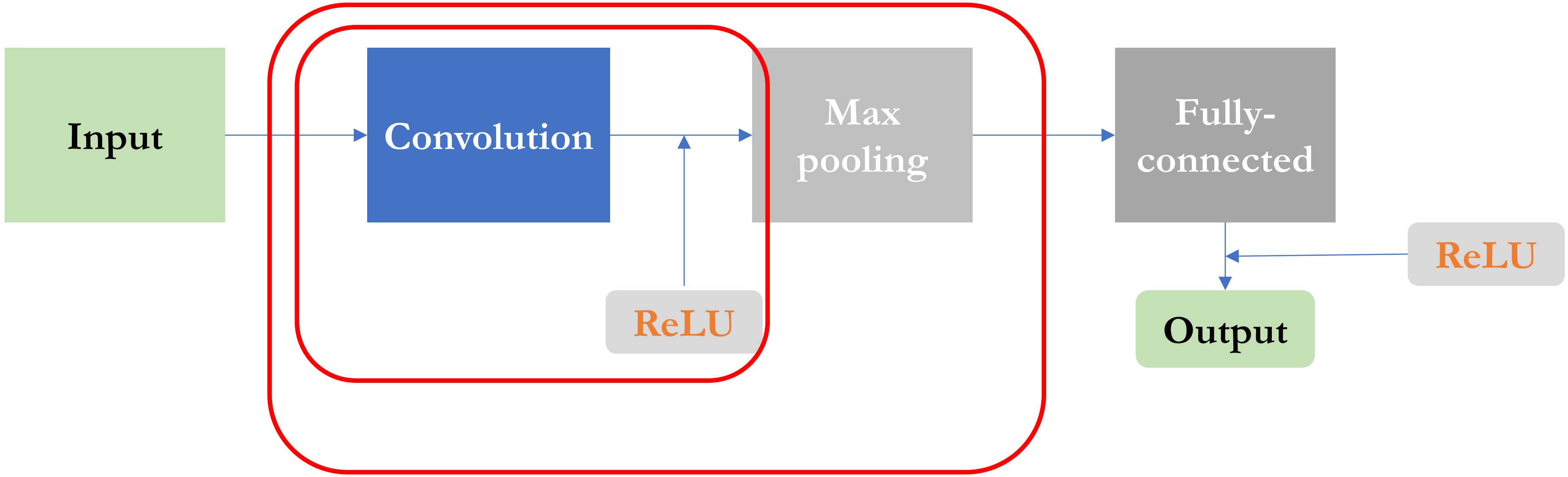
- Input: An image of size (W_1, H_1, D_1)

- Pooling layer
  - Filter size F
  - Stride size S

- Output size: (W_2, H_2, D_2)
  - $W_2 = \frac{W_1 - F}{S} + 1$
  - $H_2 = \frac{H_1 - F}{S} + 1$
  - $D_2 = D_1$

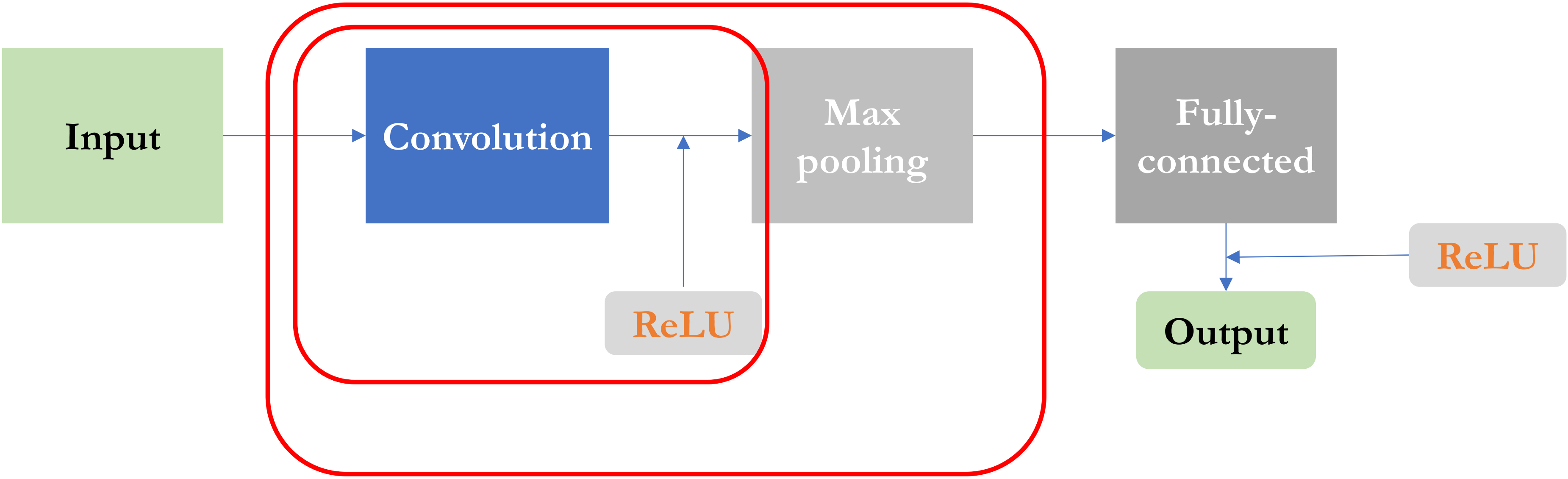
- **Previous example: $\boldsymbol{F = 2}$ and $\boldsymbol{S = 2}$**

CNN architecture



- A deep CNN involves multiple convolution and pooling layers
- Input -> [[Conv -> ReLU]*N -> Pool?]*M -> [FC -> ReLU]*K -> FC

Summary of CNN architecture



- Input -> FC: Linear classifier
- Input -> FC -> ReLU: Non-linear classifier
- Input -> (Conv -> ReLU -> Pool)*2 -> FC -> ReLU -> FC: A simple CNN architecture
- Input -> (Conv -> ReLU -> Conv -> ReLU -> Pool) -> FC -> ReLU -> FC: Suitable for large images

Lecture plan

- **Implementation of a simple CNN in PyTorch**

Implementation in PyTorch

- Loading dependencies

Implement a convolutional neural network to recognize handwritten digits

Before you start, make sure to read the problem description in the handout pdf.

```
# Uncomment the below line and run to install required packages if you have not done so

# !pip install torch torchvision matplotlib tqdm
```

```
# Setup
import torch
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets, transforms
from tqdm import trange

%matplotlib inline
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# Set random seed for reproducibility
seed = 1234
# cuDNN uses nondeterministic algorithms, set some options for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.manual_seed(seed)
```

Loading dataset

Get MNIST Data

The `torchvision` package provides a wrapper to download MNIST data. The cell below downloads the training and test datasets and creates dataloaders for each.

```
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])
#torchvision.datasets.MNIST(root=root_dir,download=True)
root_dir = './data'
train_data = datasets.MNIST(root_dir, train=True, download=False, transform=transform)
test_data = datasets.MNIST(root_dir, train=False, download=False, transform=transform)

train_data.transform = transform
test_data.transform = transform

batch_size = 64
torch.manual_seed(seed)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=True)
```

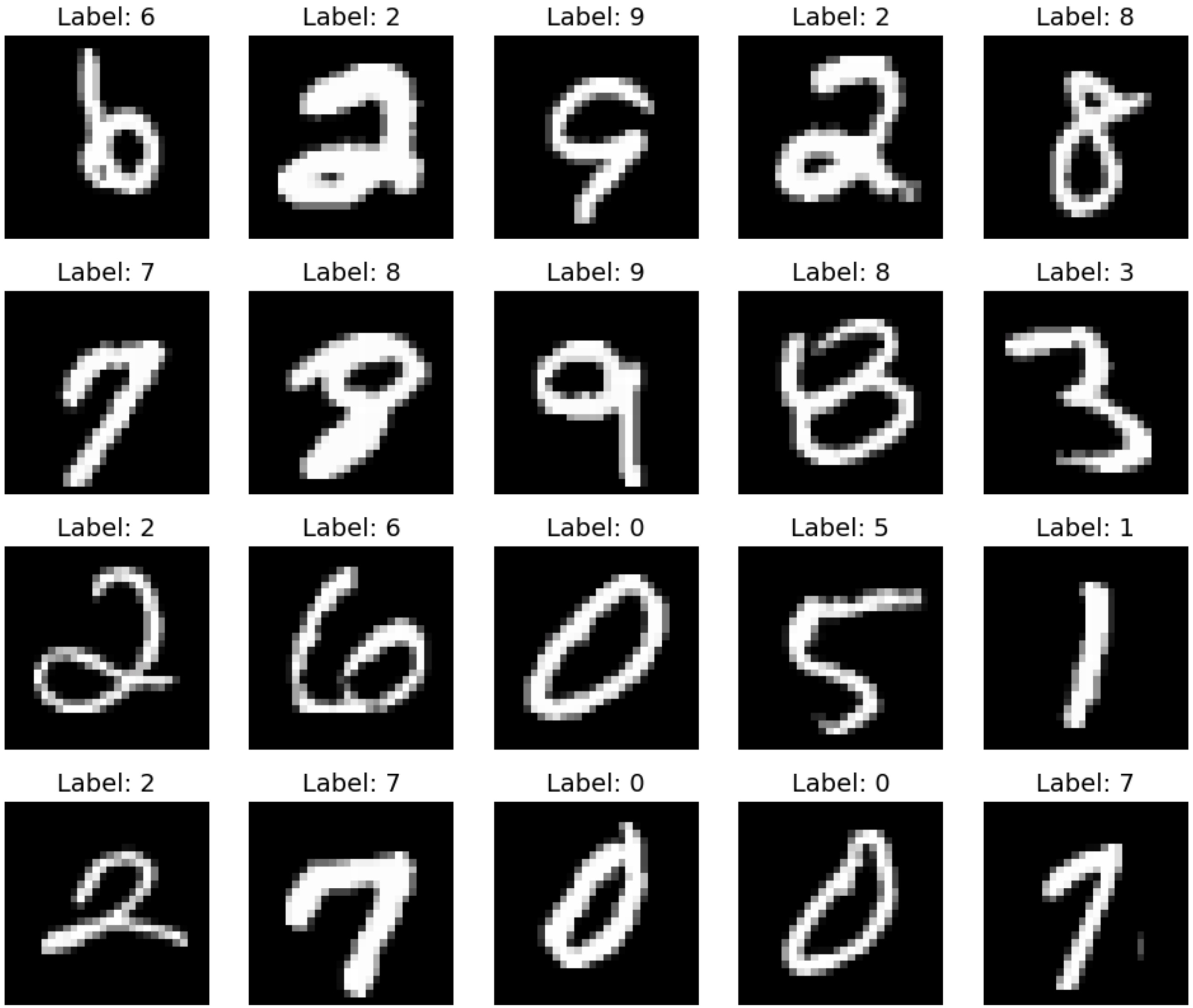
Inspect dataset

```
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Print information and statistics of the first batch of images
print("Images shape: ", images.shape)
print("Labels shape: ", labels.shape)
print(f'Mean={images.mean()}, Std={images.std()}')

fig = plt.figure(figsize=(12, 10))
for i in range(20):
    plt.subplot(4, 5, i+1)
    plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
    plt.title(f'Label: {labels[i]}', fontsize=14)
    plt.axis('off')
```

Visualization



Defining network architecture

Implement a two-layer neural network

Write a class that constructs a two-layer neural network as specified in the handout. The class consists of two methods, an initialization that sets up the architecture of the model, and a forward pass function given an input feature.

```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = torch.nn.Sequential(
```

```
model = CNN().to(DEVICE)

# sanity check
print(model)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Linear(in_features=320, out_features=10, bias=True)
  (act): ReLU()
)
```

```
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        # print(x.shape)
        x = x.view(x.size(0), -1)
        # print(x.shape)
        x = self.fc(x)
#        x = self.log_softmax(x)
        y_output = x

        return y_output
```

Training procedure

Implement an optimizer to train the neural net model

Write a method called `train_one_epoch` that runs one step using the optimizer.

```
def train_one_epoch(train_loader, model, device, optimizer, log_interval, epoch):
    model.train()
    losses = []
    counter = []

    for i, (img, label) in enumerate(train_loader):
        img, label = img.to(device), label.to(device)

        # ------------------
        optimizer.zero_grad()
        output = model(img)
        criterion = torch.nn.CrossEntropyLoss()
        loss = criterion(output, label)

        loss.backward()
        optimizer.step()
        # ------------------

        # Record training loss every log_interval and keep counter of total training images seen
        if (i+1) % log_interval == 0:
            losses.append(loss.item())
            counter.append(
                (i * batch_size) + img.size(0) + epoch * len(train_loader.dataset))

    return losses, counter
```

- Stochastic gradient descent
  - Let w_t be the parameters of a neural network
  - Let f_{w_t} be the neural network
  - Let $\nabla \hat{L}(f_{w_t})$ be the gradient of the training loss at w_t
  - Let η be a learning rate parameter, and B be the number of batches
  - For $i = 0, 1, \ldots, B-1$

$$w_t \leftarrow w_t - \eta \cdot \nabla \hat{L}_i(f_{w_t}),$$

where the loss is evaluated on the i-th batch

Test accuracy

```python
# Hyperparameters
lr = 0.001
max_epochs=10
gamma = 0.95

# Recording data
log_interval = 100

# Instantiate optimizer (model was created in previous cell)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

# Use for CNN model
# optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
train_counter = []
test_losses = []
test_correct = []
for epoch in trange(max_epochs, leave=True, desc='Epochs'):
    train_loss, counter = train_one_epoch(train_loader, model, DEVICE, optimizer, log_interval, epoch)
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

    # Record results
    train_losses.extend(train_loss)
    train_counter.extend(counter)
    test_losses.append(test_loss)
    test_correct.append(num_correct)

    print(train_loss, test_loss, num_correct)

print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
```

```
67, 0.7128437161445618, 0.4861321151256561 3] tensor(0.3032) 8908

Epochs:  80%|████████████████████████████████████████        | 8/10 [01:19<0
0:19,  9.96s/it]
[0.4310001730918884, 0.2578464150428772, 0.390159547328949, 0.2206697016954422, 0.3051441013813019, 0.22070705890655518, 0.659205794334411
6, 0.4572473466396332, 0.41547641158103943] tensor(0.2518) 8995

Epochs:  90%|█████████████████████████████████████████████   | 9/10 [01:30<0
0:10, 10.06s/it]
[0.6253061890602112, 0.3636443614959717, 0.2863709330558777, 0.3423950672149658, 0.3142278790473938, 0.2135738581418991, 0.291072398424148
56, 0.47620293498039246, 0.3207015097141266] tensor(0.2235) 9060

Epochs: 100%|██████████████████████████████████████████████████| 10/10 [01:40<0
0:00, 10.03s/it]
[0.32710516452789307, 0.376585990190506, 0.4734595715999603, 0.47056400775909424, 0.17729906737804413, 0.25048649311065674, 0.188878461718
55927, 0.30020228028297424, 0.3596789538860321] tensor(0.2127) 9115
Test accuracy: 0.9115
```

Training and test loss curves

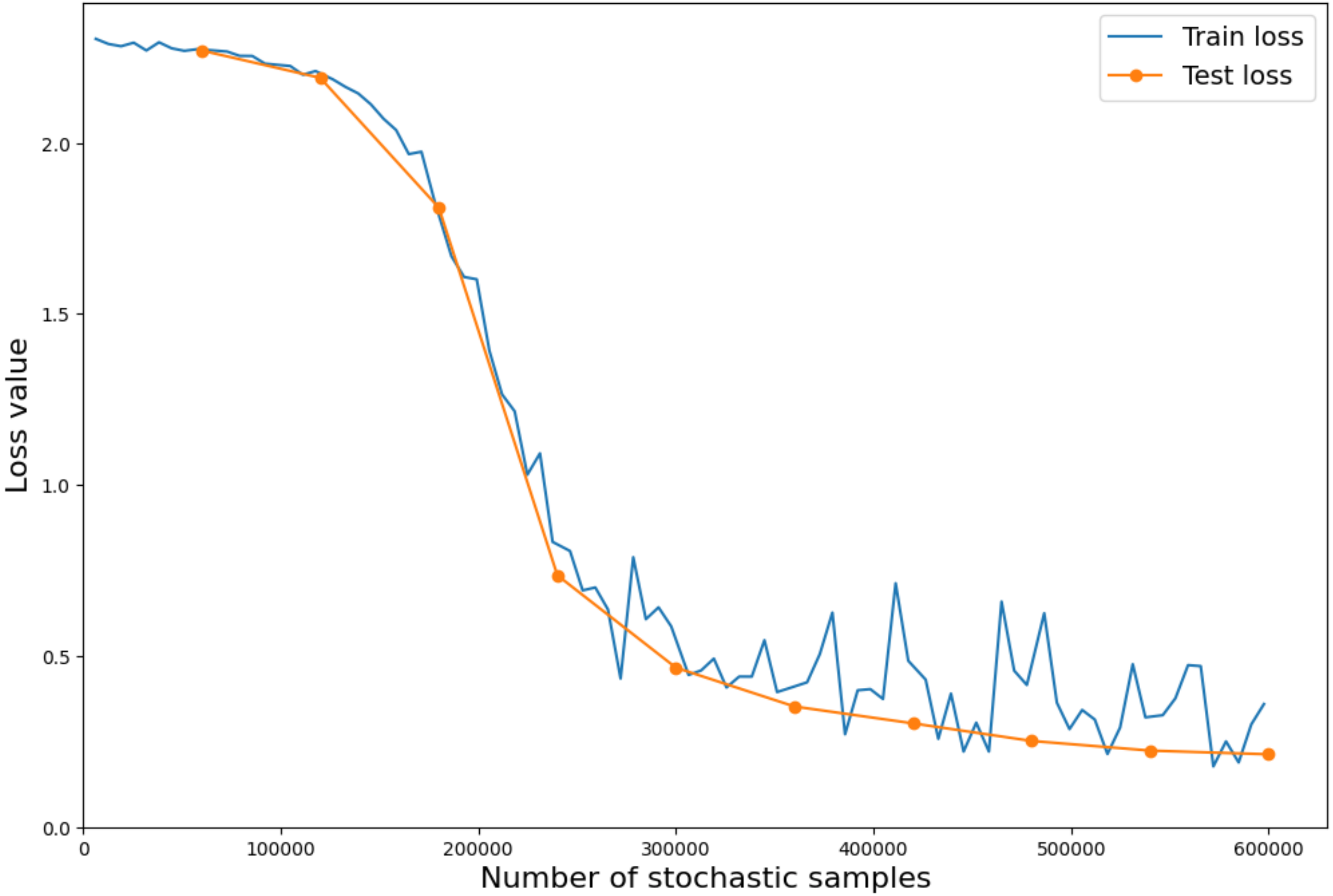
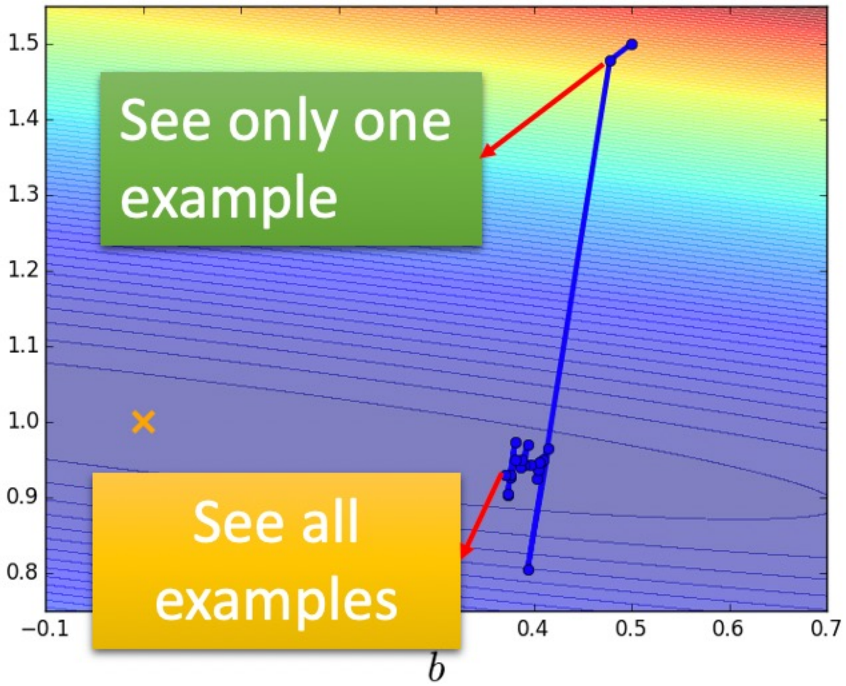


Illustration of stochastic gradient descent

- **Stochastic Gradient Descent** updates for each example, whereas gradient descent updates for all examples



A more sophisticated CNN architecture

```
VGG (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU (inplace)
    (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU (inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU (inplace)
    (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU (inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU (inplace)
    (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU (inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU (inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU (inplace)
    (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU (inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU (inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU (inplace)
    (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
```

```python
from torchvision import models
model = models.vgg16()
print(model)
```

```
  (classifier): Sequential (
    (0): Dropout (p = 0.5)
    (1): Linear (25088 -> 4096)
    (2): ReLU (inplace)
    (3): Dropout (p = 0.5)
    (4): Linear (4096 -> 4096)
    (5): ReLU (inplace)
    (6): Linear (4096 -> 1000)
  )
)
```

More suitable for large-sized, colored images (e.g., ImageNet)

Announcements

- HW2 is due

- Submit regrade requests on gradescope or drop by TA office hours to double check grading doubts

- We will release HW3 later today (this homework will be lighter than HW2)