Introduction to Deep Learning

Lecture 2, March 27, 2025

CS25800 Adv. ML

PyTorch Info Session

- This Friday 3/28, Two Sessions: 2:30-3:30, 3:30-4:30pm
- Location: JCL 298
- Three topics:
  - Quick PyTorch overview;
  - How to install/run PyTorch on the CS dept server; SSH overview;
  - How to install/run PyTorch on local Macbooks; Jupyter notebook overview.

For Now, Focusing on Discriminatory AI

Classification tasks rather than generation tasks

Lecture Goal

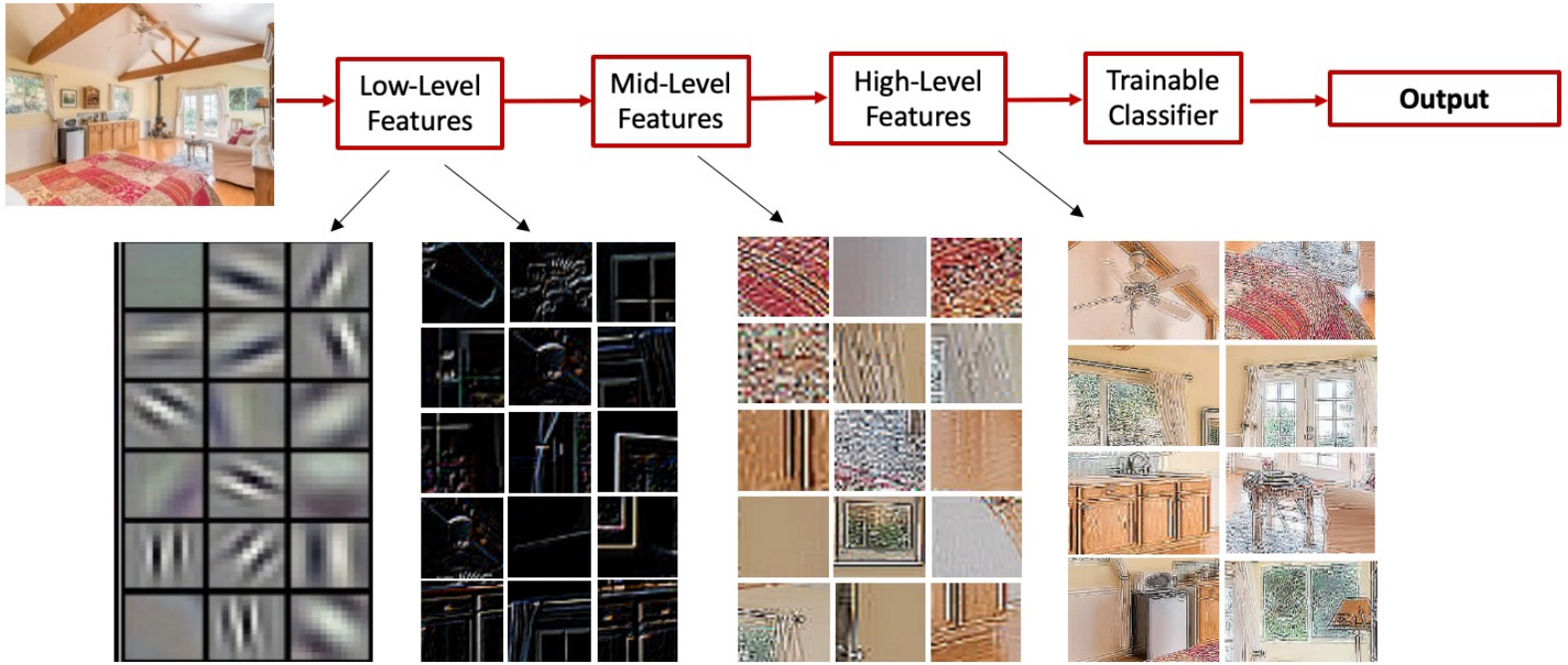
- Key elements of deep neural networks (DNN)
- How to define, train, and evaluate a DNN model
- Key elements of CNN (time permits)

→ Challenges facing training a "good" DNN model for classification

→Vulnerabilities of DNN models

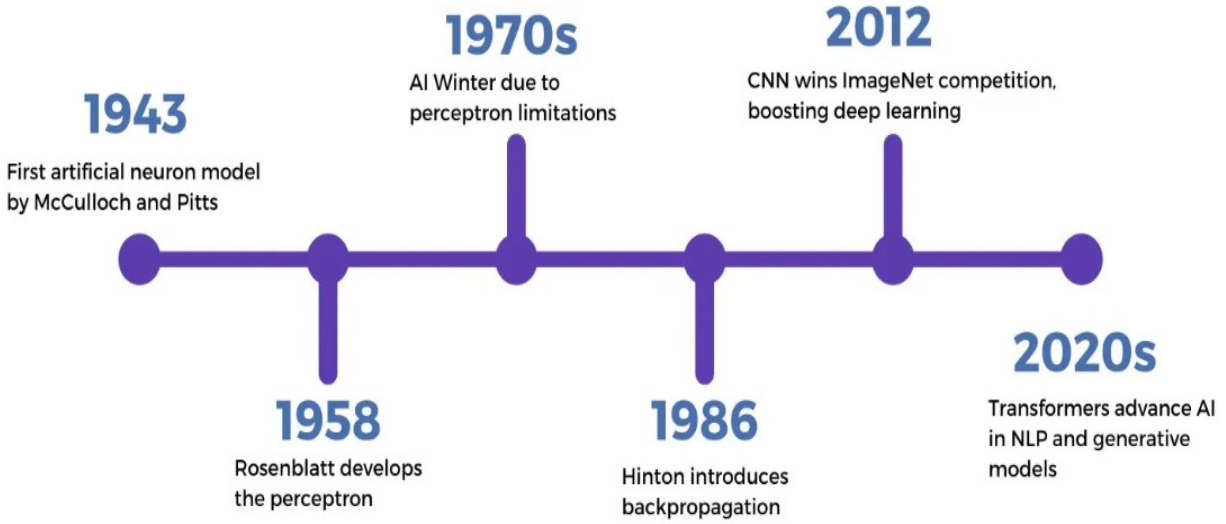
Deep Learning (Deep Neural Networks)

- **Multi-layer Neural Network** for Learning Data Representations **Directly** from Data



History of Neural Networks

- Perceptron 1958
- Backpropagation (multi-layer perceptron) 1986
- Convolutional Neural Networks (CNN) 1995



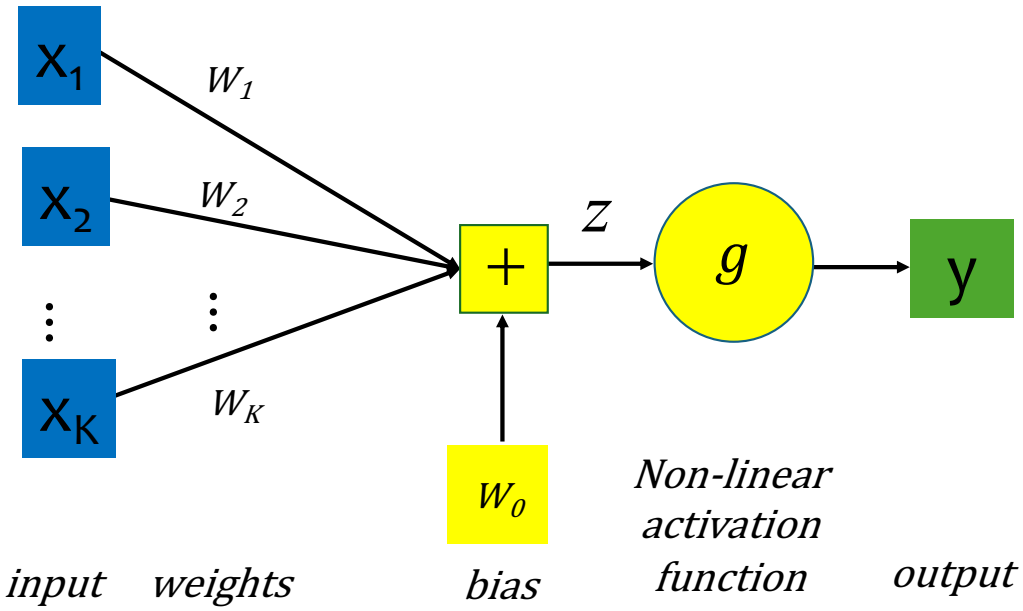
Today's Schedule

A. Perceptron (a single neuron)
B. Multilayer Neural Networks
C. Training a Multilayer Neural Network
D. Evaluating a Model

E. Convolution NN (CNN)

A. Perceptron (a single neuron)

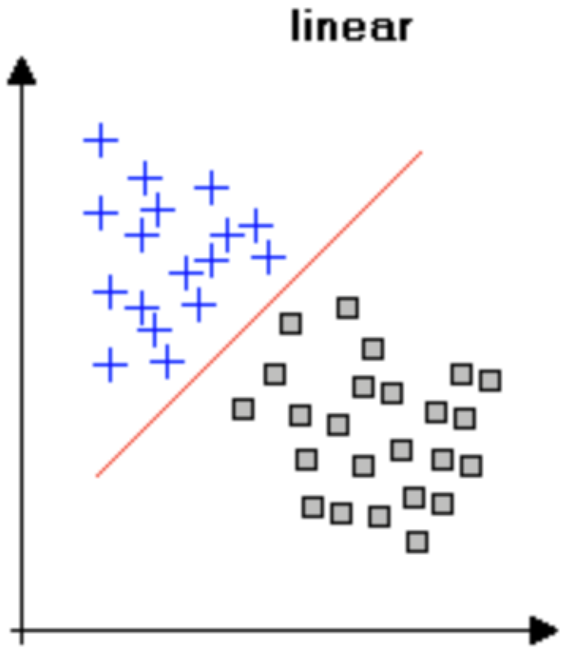
Perceptron: A Single Neuron

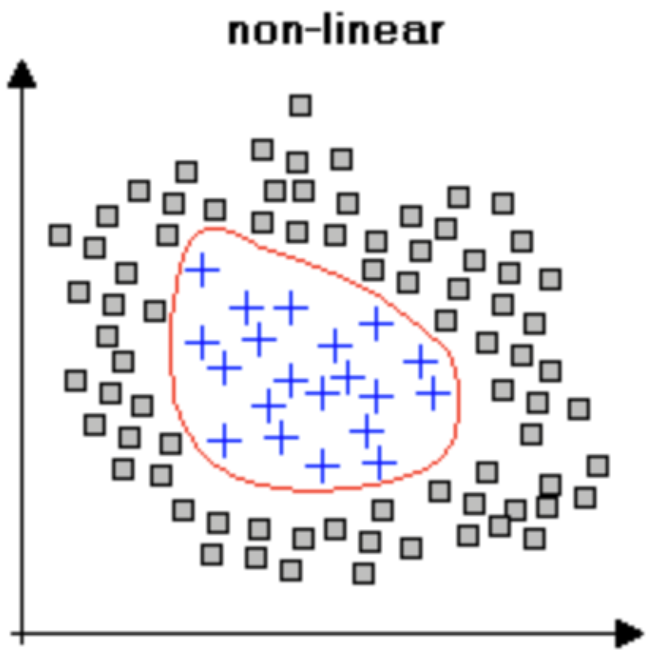


z: linear combination of inputs

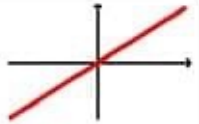
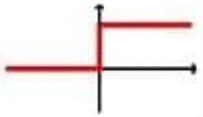
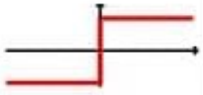
$$y = g(w_0 + x_1 w_1 + \ldots + x_K w_K)$$
$$= g(w_0 + \mathbf{X}^T \mathbf{W})$$

Why Non-Linear Activation?





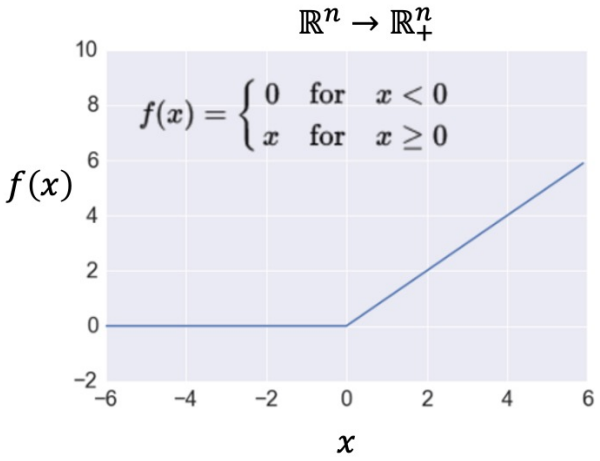
Potential Activation Functions

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

ReLU

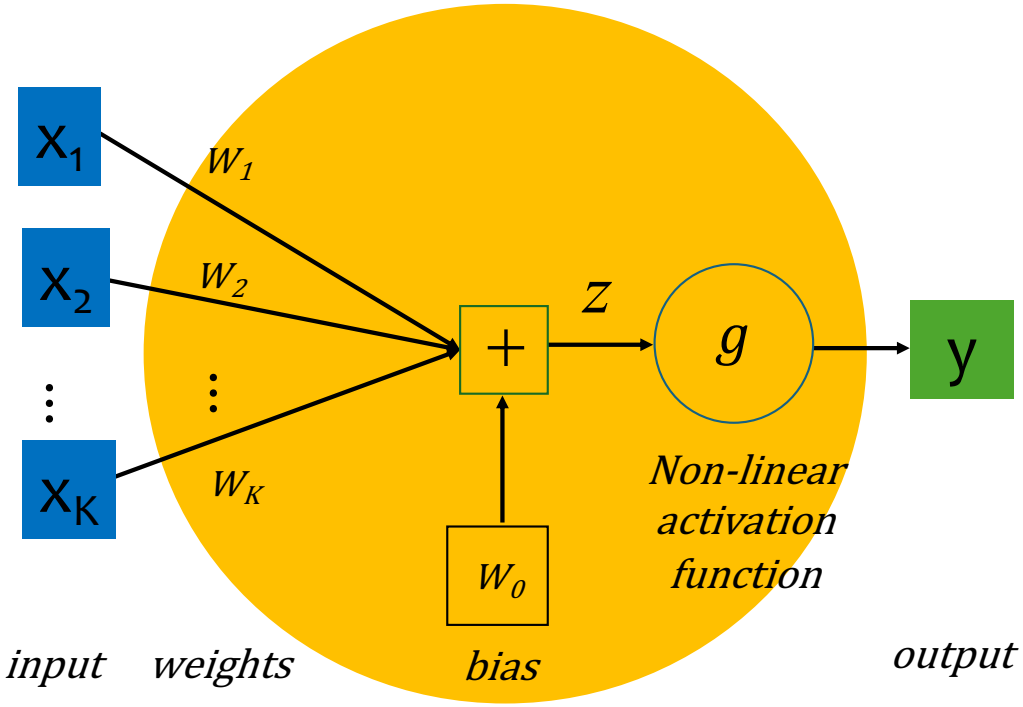
- ***ReLU*** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

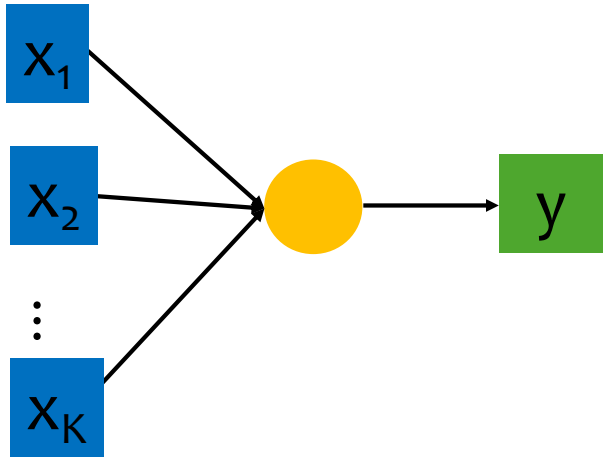
$$f(x) = \max(0, x)$$



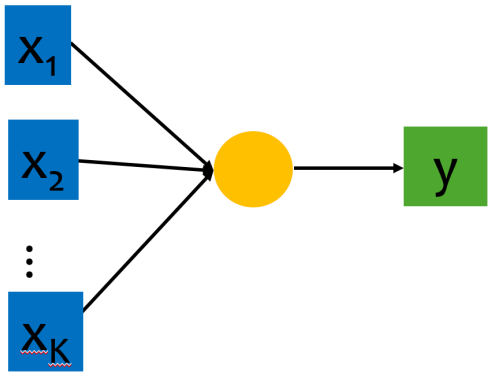
- Most modern deep NNs use ReLU activations
  - ReLU is fast to compute
    - Compared to sigmoid, tanh
    - Simply threshold a matrix at zero
  - Accelerates the convergence of gradient descent

Perceptron: A Single Neuron

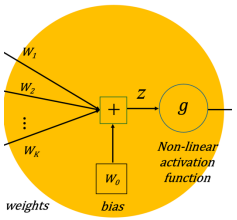


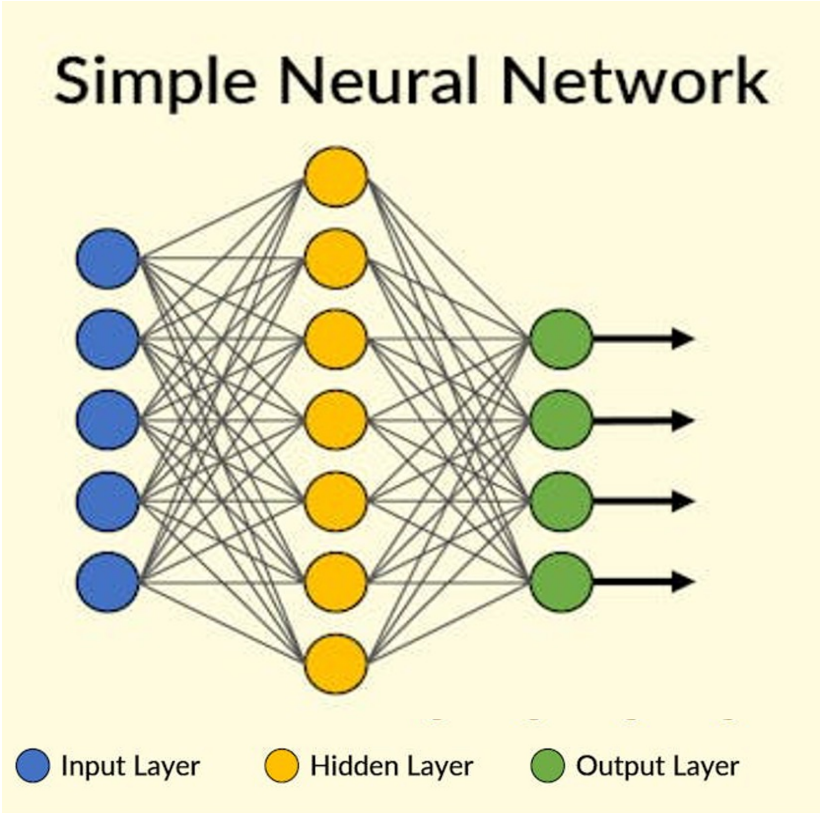


Single-layer Neural Network (NN)



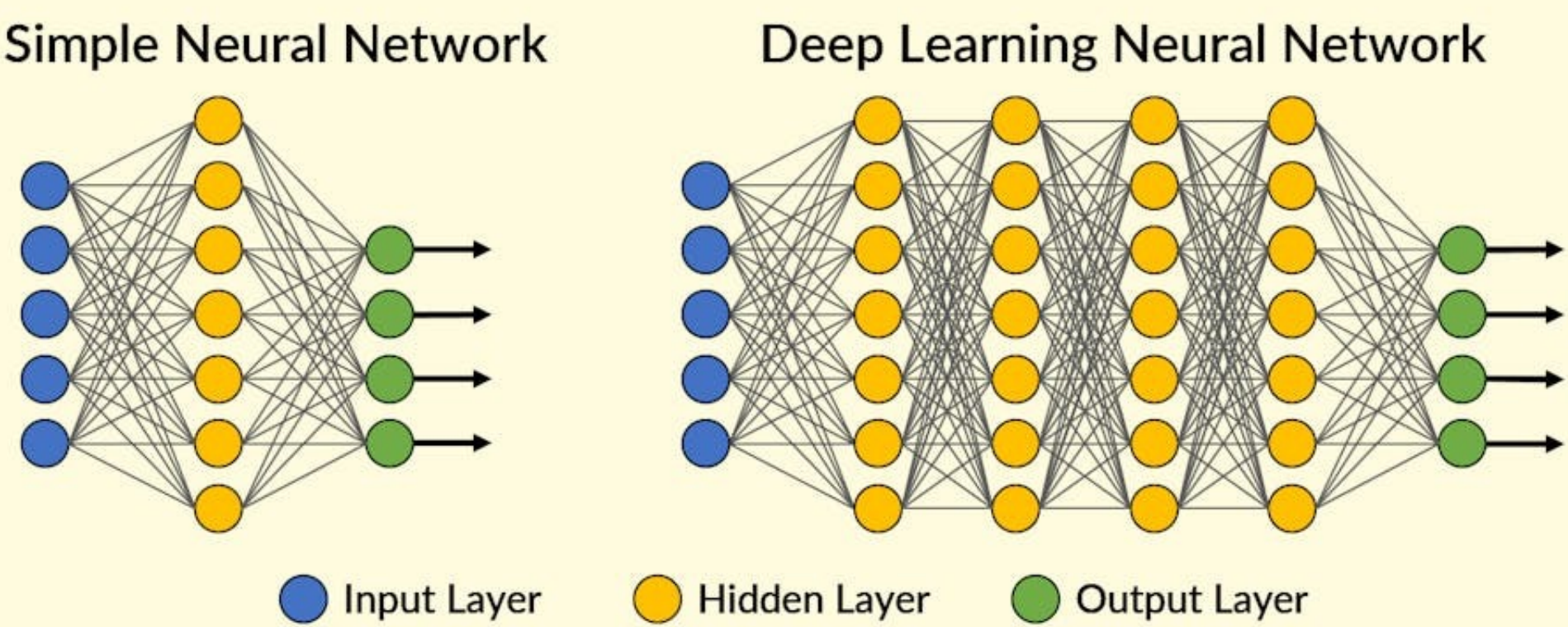
Hidden layer





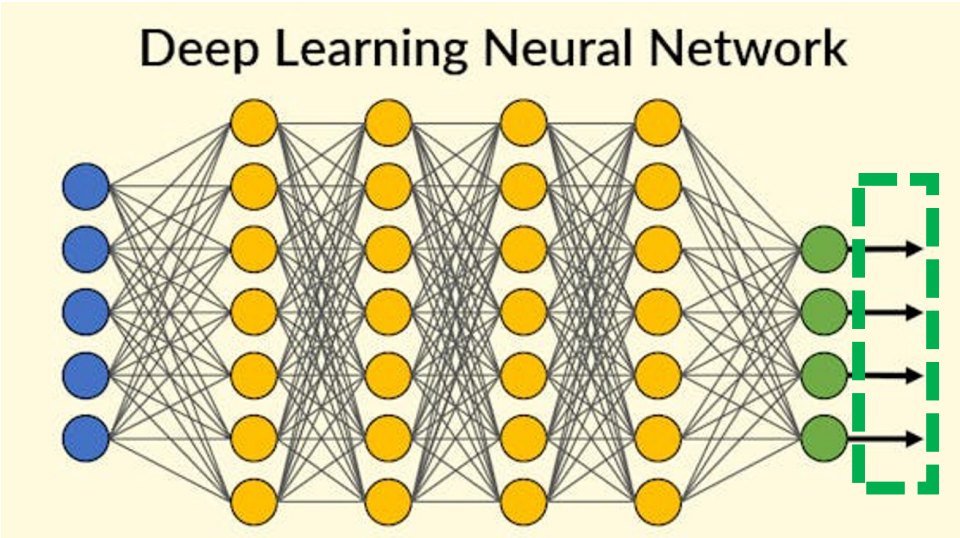
B. Multilayer Neural Networks

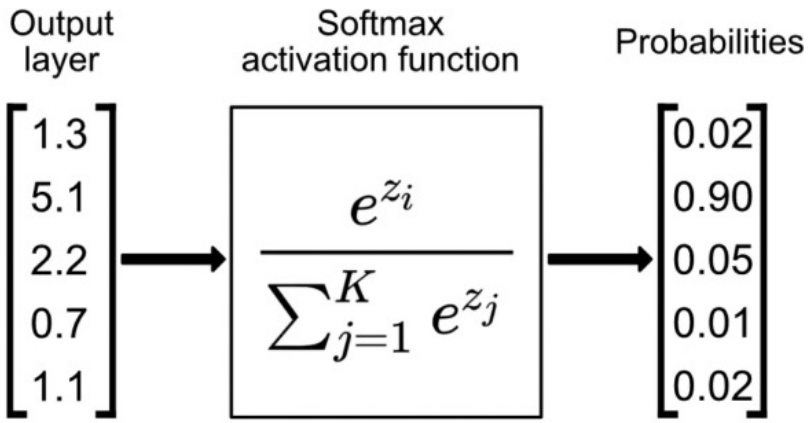
Multi-layer NN



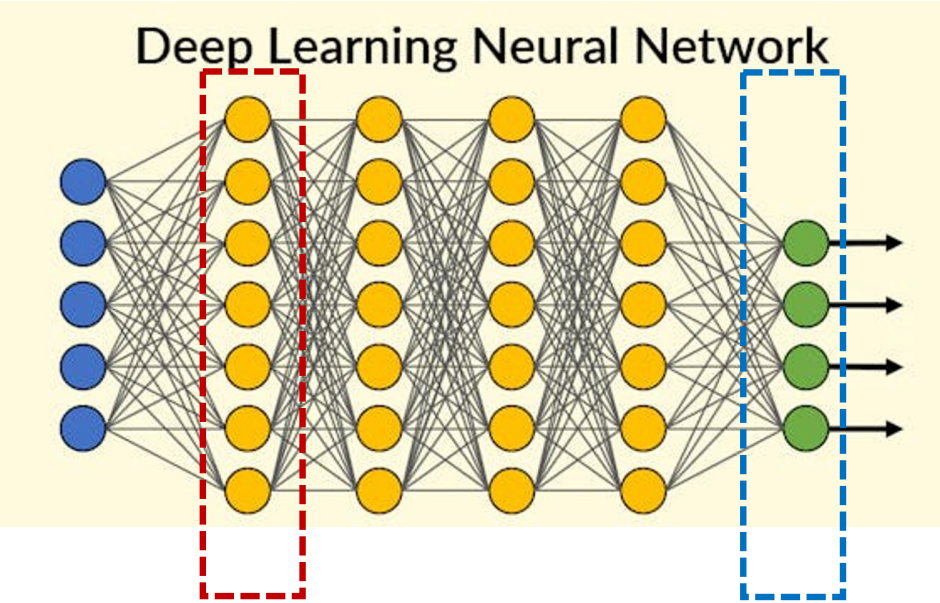
Softmax

- The final operation of a multi-classification NN model
  - Converting raw scores into a probability distribution over possible classes
  - Ensuring that the probabilities for all classes sum up to 1





PyTorch samples for Defining a Deep NN



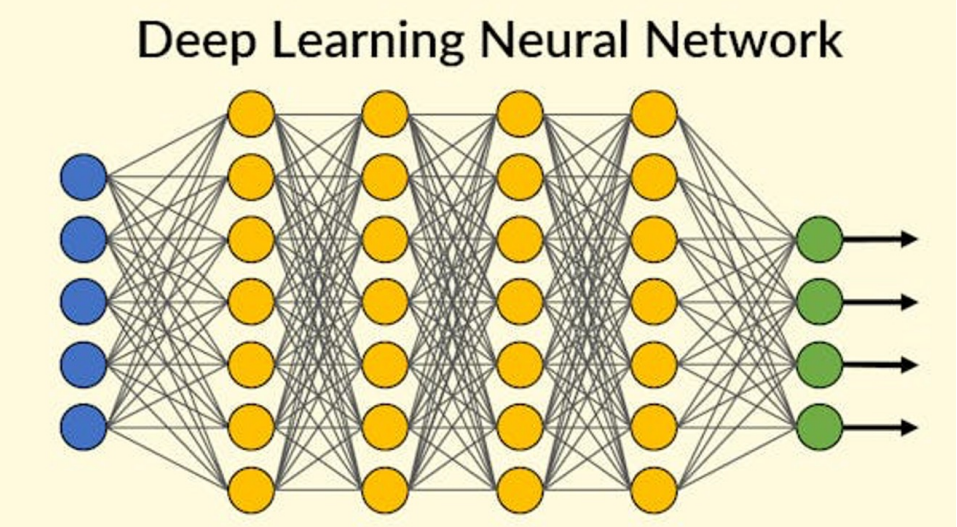

```
import torch.nn as nn
model = nn.Sequential(
    nn.Linear(5, n_1),
    nn.ReLU(),

    nn.Linear(n_1, n_2),
    nn.ReLU(),
    ...
    nn.Linear(n_k, 4)
)
```

** Does not include SoftMax computation

C. Training a Multilayer Neural Network

Training NNs

- Supervised learning
- Input: Training data (X), and labels (Y)
- **Optimize:** Model weights (***W***)
- **Loss-based optimization**



Loss based Optimization

- Loss $J(\boldsymbol{W})$ measures the cost incurred from **incorrect model predictions**
- Given training data: **N** pairs of (input, label): $x^{(i)}, y^{(i)}$

$$J(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(x^{(i)}; \boldsymbol{W}), y^{(i)})$$

Model predicted probability

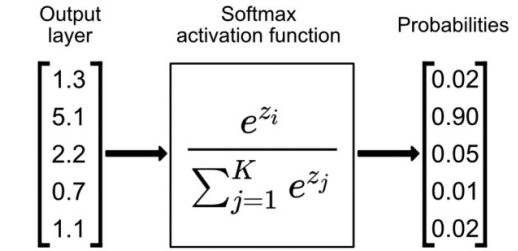
True label

- Find the model weights ($\mathbf{W}^*$) that achieve the lowest loss

$$\boldsymbol{W}^* = \underset{W}{\operatorname{argmin}}\ J(\boldsymbol{W})$$

Cross Entropy Loss

- A popular loss function for classification models
- Measures the difference between the predicted probability and the true probability
  - predicted probability = softmax probability of x belong to a specific class
  - true probability = the label of x



$$f\left(x^{(i)}; \boldsymbol{W}\right)$$

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Multiclass Cross Entropy Loss

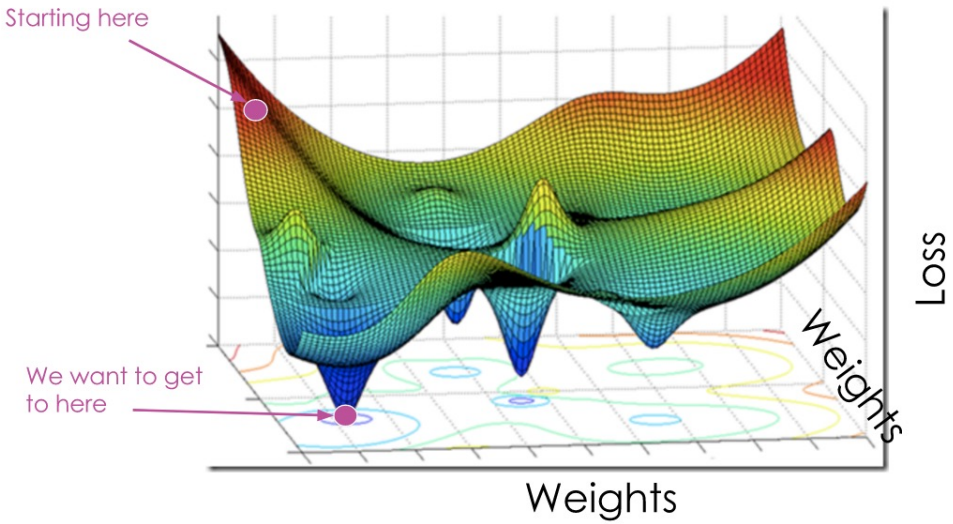
- Training Data: N pairs of (input, label): $\mathbf{x^{(i)}}, \mathbf{y^{(i)}}$
- **C: # of classes**

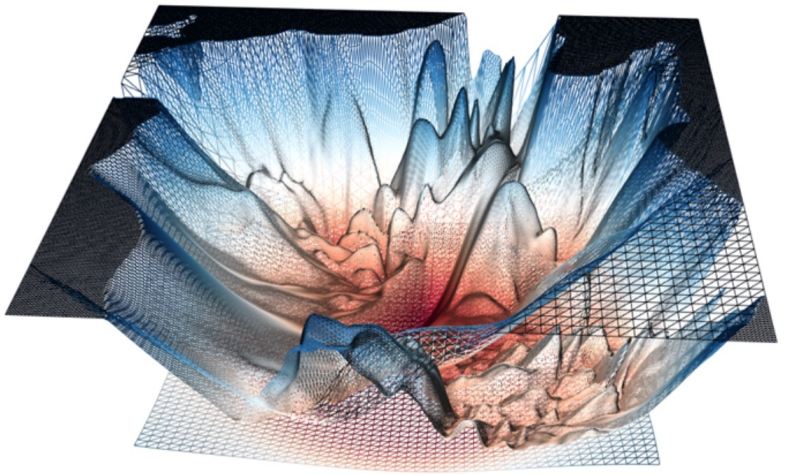
$$J(\boldsymbol{W}) = -\frac{1}{N}\sum_{j=1}^{C}\sum_{i=1}^{N} y^{(i,j)}\log(p^{(i,j)})$$

Per class label

Per class predicted probability


```
loss = torch.nn.functional.cross_entropy(outputs, labels)
```





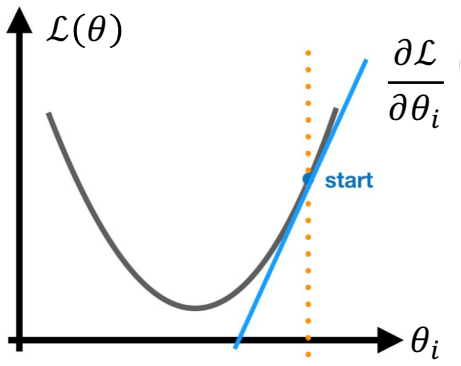
How to find the optimal W that minimizes the loss J(W)?

Image source: https://www.telesens.co/2019/01/16/neural-network-loss-visualization/

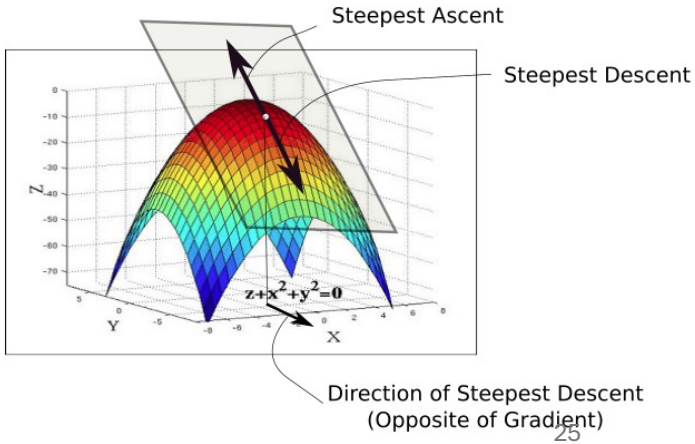
$$\boldsymbol{\theta} == \boldsymbol{W}$$

Gradient Descent (GD)

- An optimization algorithm for finding θ that minimizes $\mathcal{L}(\theta)$
- Walks down the hill in the steepest direction until it gets to a bottom



  - **Direction** defined by $-\frac{\partial \mathcal{L}(\theta)}{\partial\, \theta_i}$
  - **Movement amount** defined by η (learning rate)



Gradient Decent based Model Training

Pseudo code

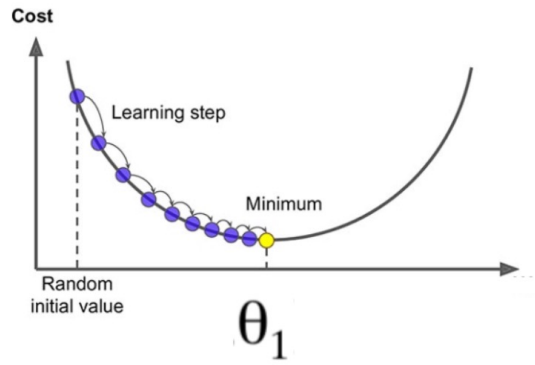
Initialize weights **W** randomly $N(0,\sigma^2)$
Loop until converge:

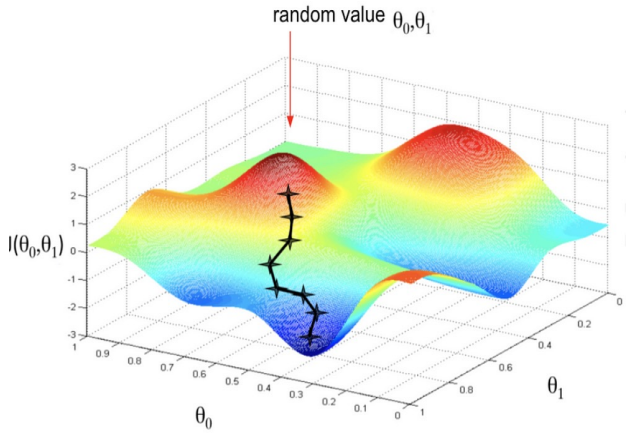
Compute loss $J(W)$

Compute gradient $\frac{\partial J(W)}{\partial W}$

Update weight: W = W - η $\frac{\partial J(W)}{\partial W}$

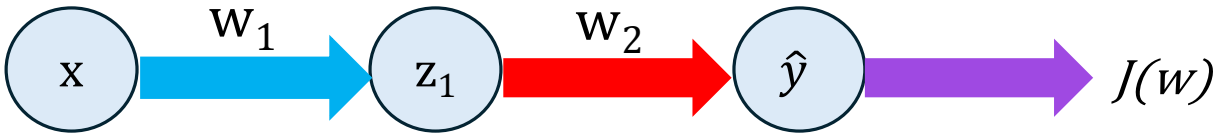
Return weights **W**





How to Compute Gradient $\frac{\partial J(W)}{\partial W}$

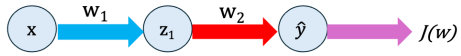
Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

Backpropagation traverses the model in reverse order, from the output y backward toward the input x to calculate the gradients: $\frac{\partial J(W)}{\partial W}$

Forward & Backward Propagation



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

- Forward propagation = passing the input x through the hidden layers to obtain the model prediction $\hat{y}$
  - Calculate the loss $\mathbf{J(W)}$ based on the model prediction and the label
- Backpropagation traverses the model in reverse order, from the output $\hat{y}$ backward toward x to calculate the gradient $\frac{\partial J(W)}{\partial W}$
- Each update of W takes one forward pass and one backward pass

Pseudo code

Initialize weights **W** randomly $N(0,\sigma^2)$

Loop until converge:

Compute loss $J(W)$

Compute gradient $\frac{\partial J(W)}{\partial W}$

Update weight: W=W - η $\frac{\partial J(W)}{\partial W}$

Return weights **W**

$$J(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f\left(x^{(i)};\boldsymbol{W}\right), y^{(i)})$$

$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$ **depends on the training data**

Perform a single parameter update == compute the loss *J(W)* and its gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$, on the entire training dataset ??

☹ Expensive and Wasteful, e.g. ImageNet has 14M images

Mini-batch Gradient Descent

- Assumption: Gradient from a mini-batch is a good approximation of gradient from the entire training set
  - Randomly divide training data into a set of mini-batches
  - Compute the loss on a mini-batch of input data, update the parameters; repeat until all training input data are used
  - At the next epoch, shuffle the training data, repeat the above process
  - Typical mini-batch size: 32 to 256 images
- Stochastic gradient descent (SGD) == Mini-batch size of 1

PyTorch has a built-in function for creating batches:

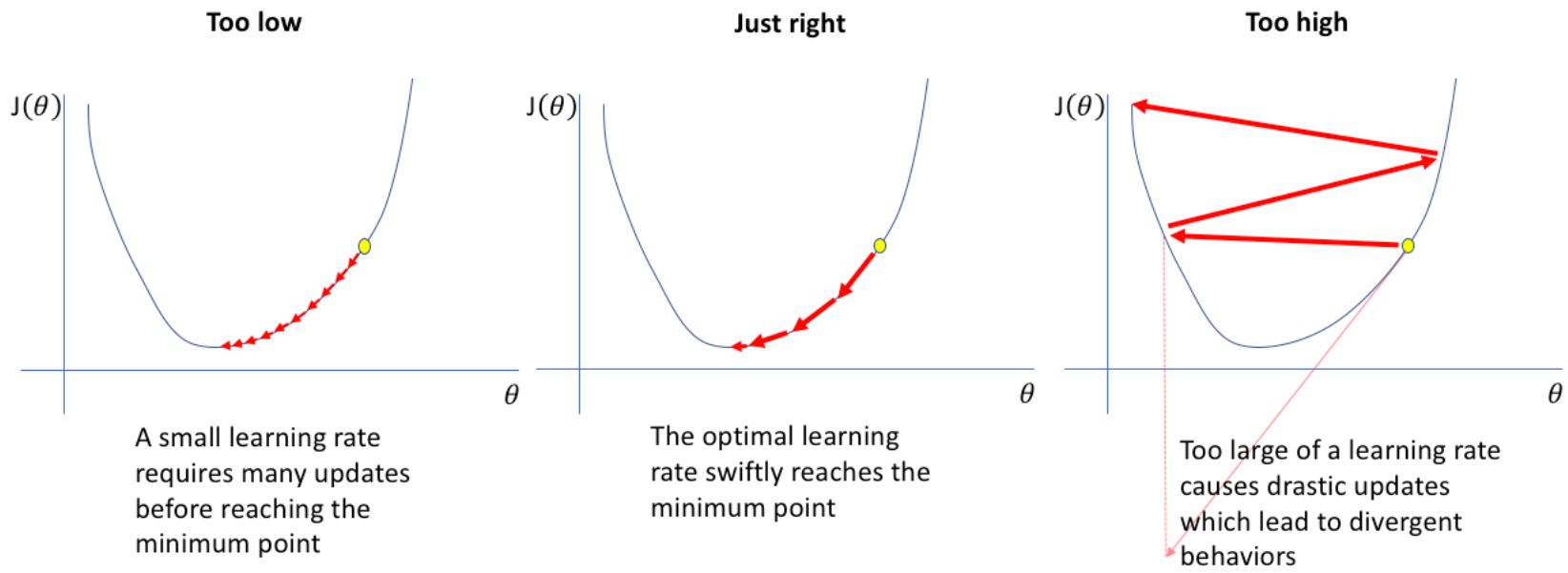
```
train_loader = torch.utils.data.DataLoader(training_set, batch_size=32, shuffle=True)
```

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

How to Set Learning Rate

Choice of Learning Rate

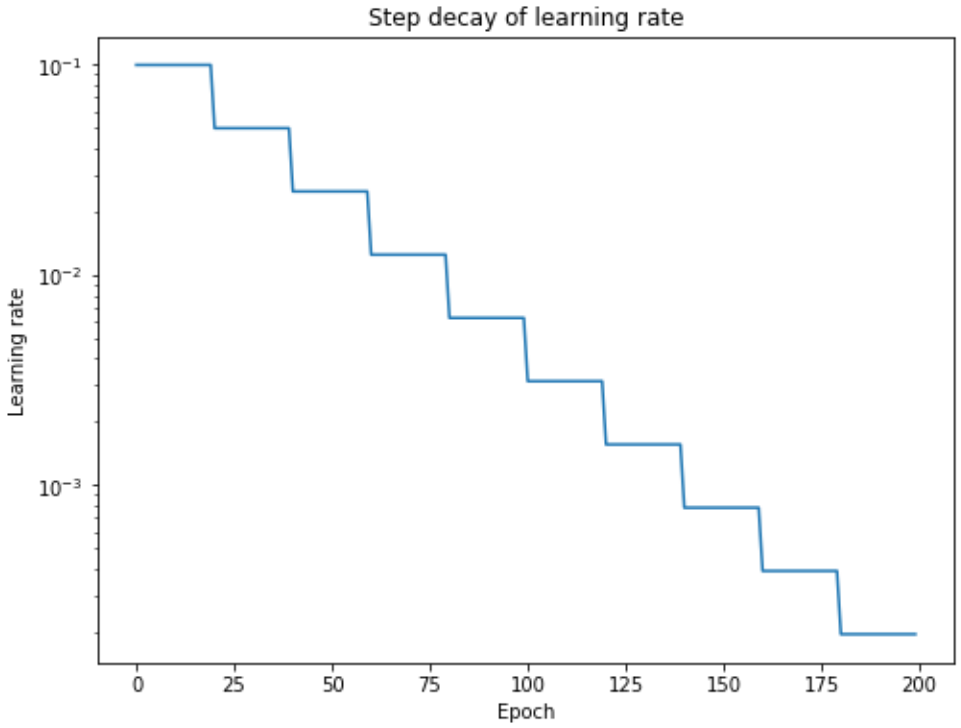
- Choices of learning rate depend on architecture, batch size, dataset



Learning Rate Scheduling

- Adapting learning rate during model training
  - Learning rate decay

- *Example: Step decay:* reducing the learning rate by some percentage after a set number of training epochs.



Hyper-parameter Tuning

- Training NNs can involve setting many ***hyper-parameters***
- The most common hyper-parameters include:
  - Number of layers, and number of neurons per layer
  - Initial learning rate
  - Learning rate decay schedule (e.g., decay constant)
  - Optimizer type
  - Batch size
- Other hyper-parameters may include:
  - Regularization parameters (ℓ_2 penalty, dropout rate)
  - Activation functions
  - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Typical Optimizers

- SGD (mostly mini-batch GD)
  - The basic optimization algorithm that updates model parameters using the gradient of the loss function
  - Uses a fixed learning rate throughout training.
- Adam
  - Adapts the learning rate for each parameter based on the first and second moments of the gradients
  - Often converges faster than SGD

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

PyTorch Samples for Training a Deep NN

Key reference: https://pytorch.org/docs/stable/nn.html

```
import torch

# specify the model
        model = torch.nn.Sequential[.........]

# pick an optimizer
        optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# define mini-batch shuffler
        train_loader = torch.utils.data.DataLoader(training_set, batch_size=32, shuffle=True)
        [.........]
# set x = input data of a training mini-batch, labels = ground truth labels of x
        [..........]
# forward pass through the model to compute loss
        outputs = model(x)
        loss = compute_loss(outputs, labels)
# initialize gradient as 0
        optimizer.zero_grad()
# back propagation to compute loss gradient over w
        loss.backward()
# update the model (i.e. update w)
        optimizer.step()
```

D. Evaluating a Classification Model

Typical Metrics

- **Classification accuracy:** $\frac{\#\text{ of correct predictions}}{\#\text{ of input samples}}$

- **Precision (P):** for a given class k, ratio of correctly predicted samples (of class k) to the total input samples being predicted as class k.
  - Macro averaged precision: calculate precision for all classes individually and average them

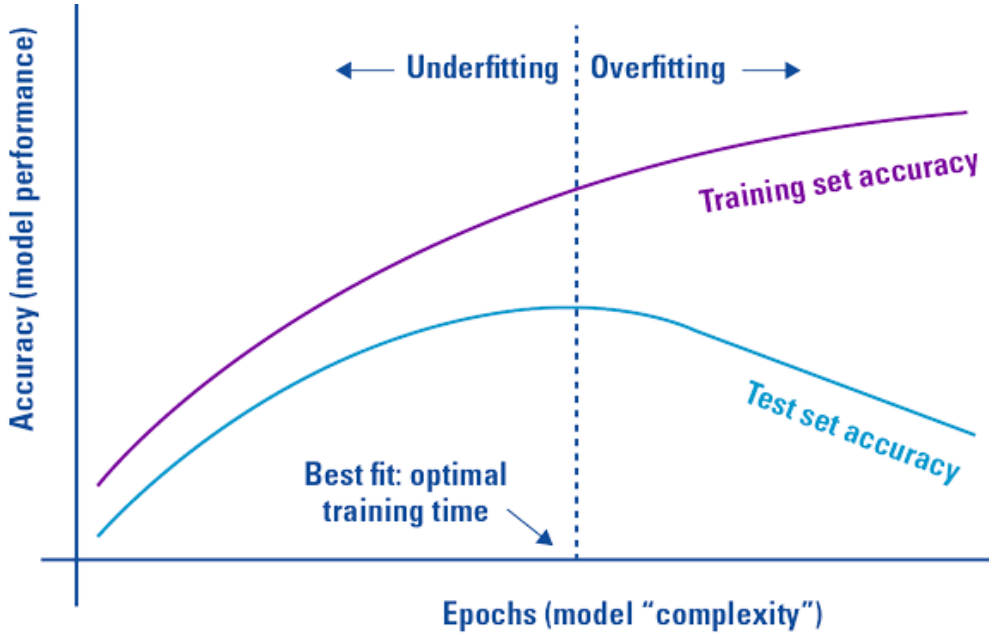
- **Recall (R):** for a given class k, the ratio of correctly predicted samples (of class k) to the total input samples whose true label is class k.
  - Macro averaged recall

- **F1 Score:** weighted average of precision and recall: $F1 = 2PR/(P+R)$

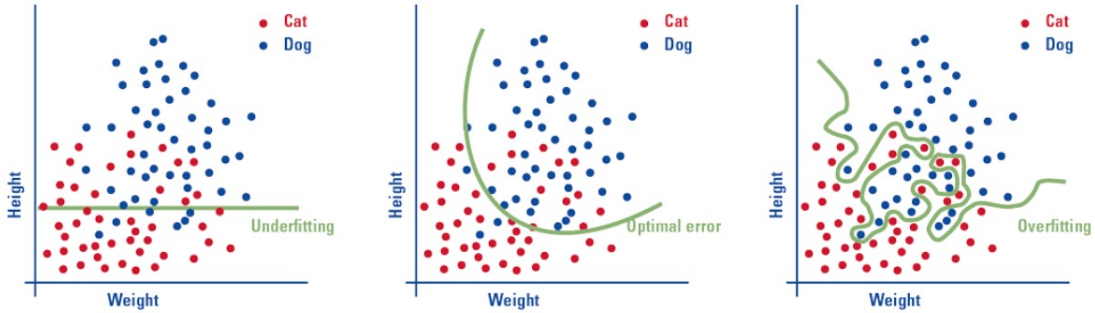
Metrics for Multi-Class Classification: an Overview

Measuring Model Accuracy

- Depending on data being used
- Accuracy on training data vs. Accuracy on validation/test data



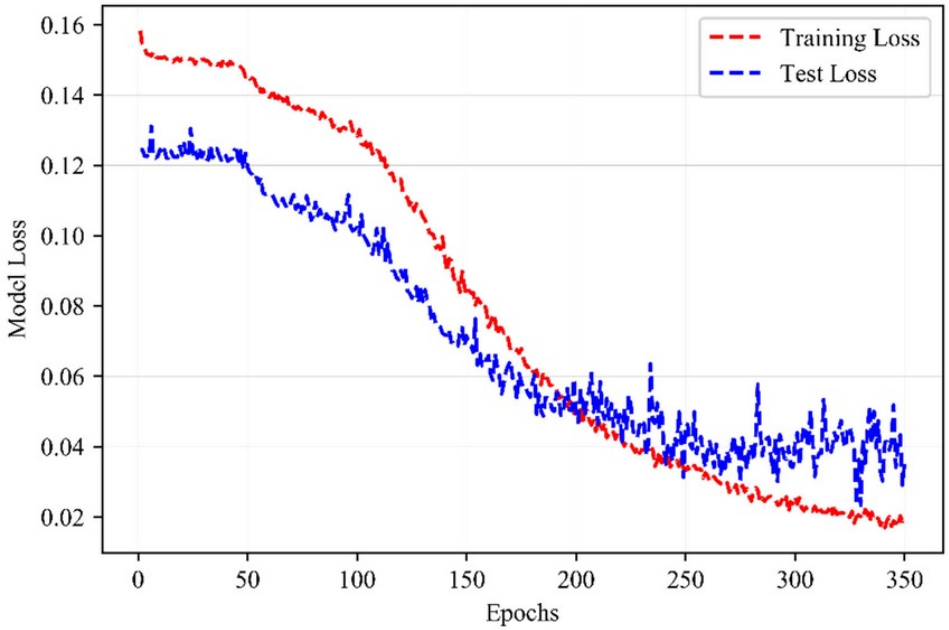
- During model training, use validation data to avoid overfitting



Images from https://www.compact.nl/articles/deep-learning-finding-that-perfect-fit/

Model Loss J(W)

- Model loss vs. model accuracy
  - Accuracy (0-100%)
  - Model loss (lower the better, but when to stop?)

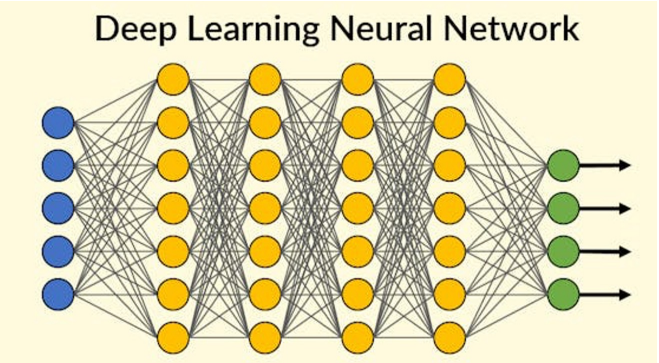


Goals of Today's Lecture

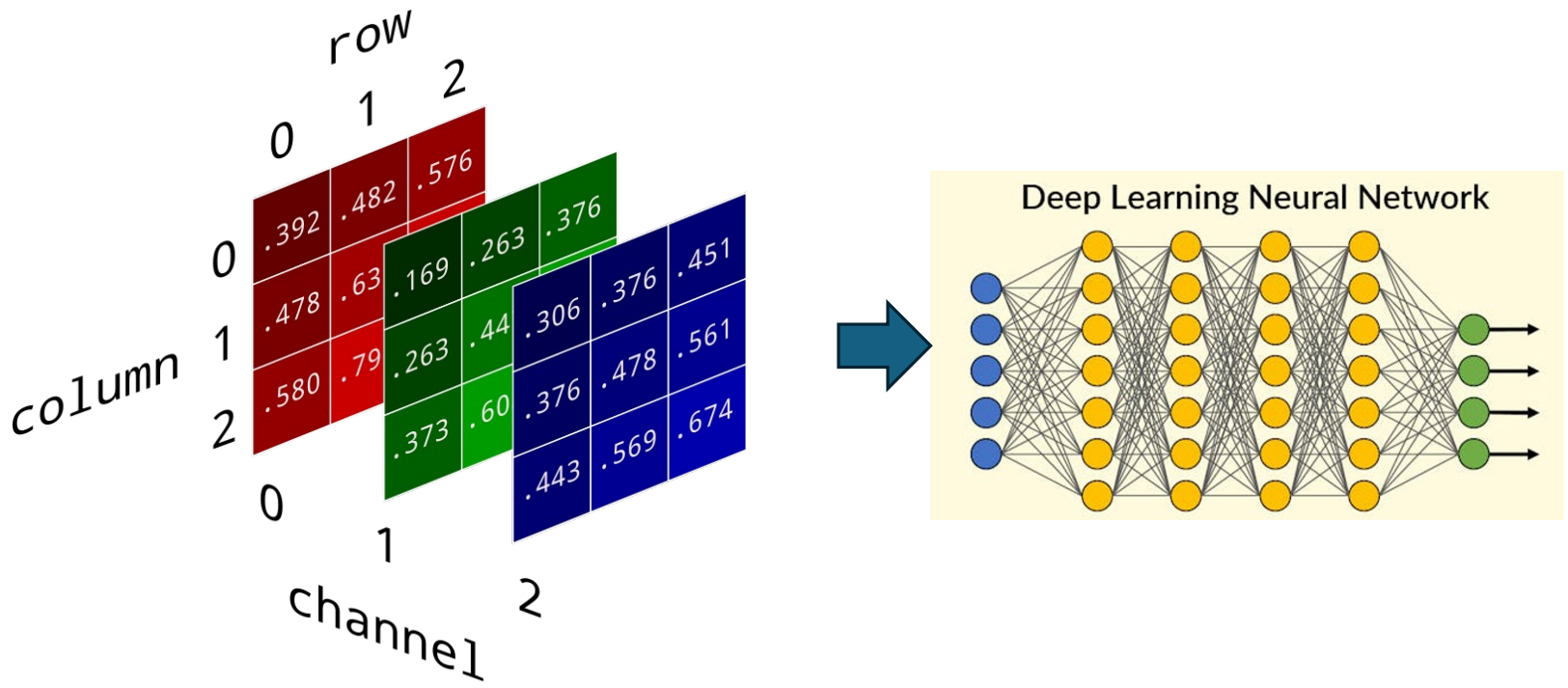
- **Key elements of deep neural networks (DNN)**
- **How to specify, train, and evaluate a DNN model**
- Key elements of CNN

E. Convolution NN Models that take an image as input

Ways to input an image into this NN

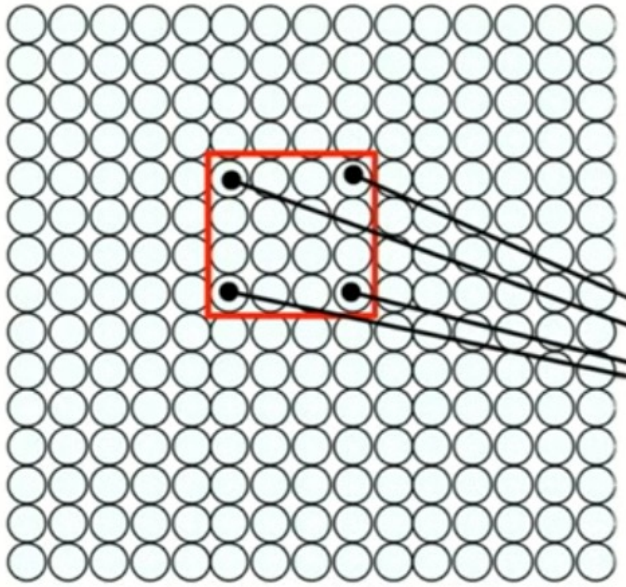


Ways to input an image into this NN



Learning the Spatial Structure

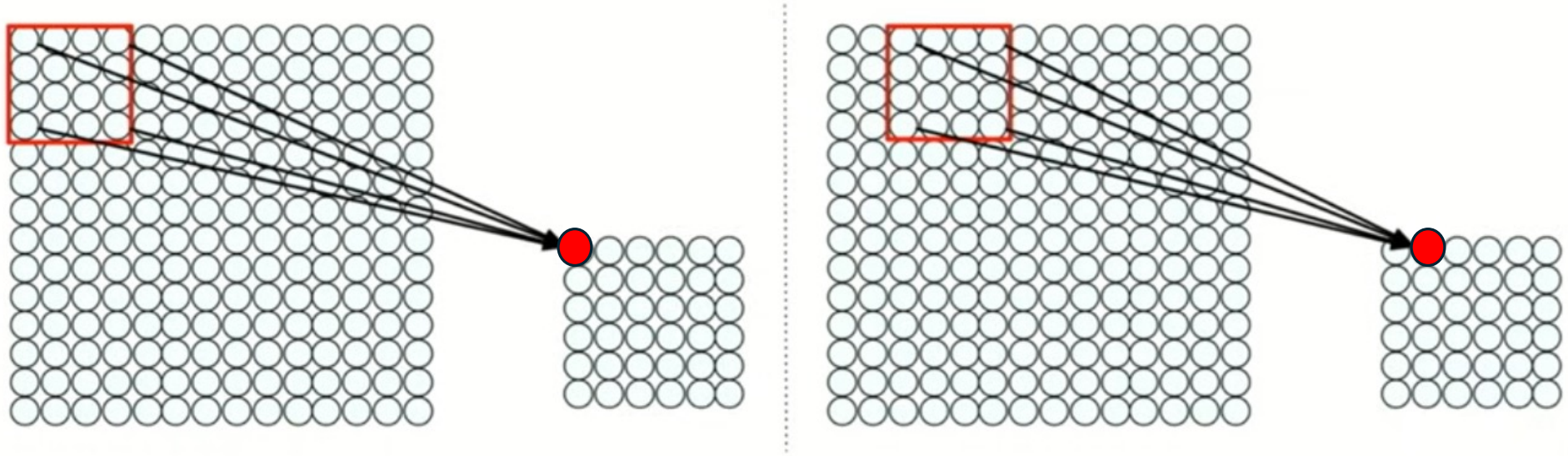
Input: 2D image.
Array of pixel values



Idea: connect patches of input to neurons in hidden layer.

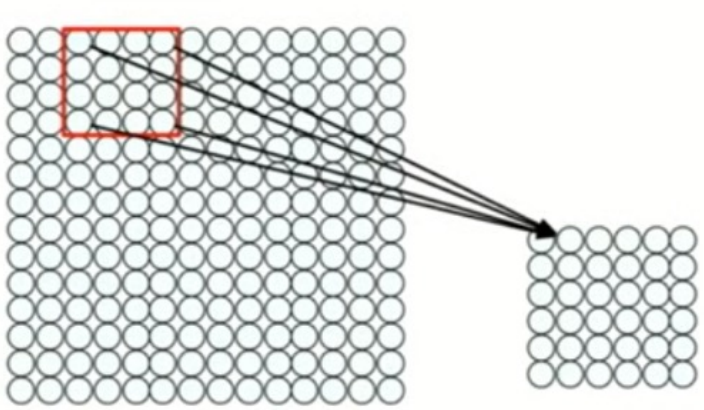
Neuron connected to region of input. Only "sees" these values.

Extracting Features from Visual Data



Connect patch in input layer to a single neuron in subsequent layer.
Use a sliding window to define connections.
How can we **weight** *the patch to detect particular features?*

Extracting Features via Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

3x3 filter, shift by 1

1	0	1
0	1	0
1	0	1

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

feature extracted by this filter

4		

Extracting Different Features by Applying Different Filters

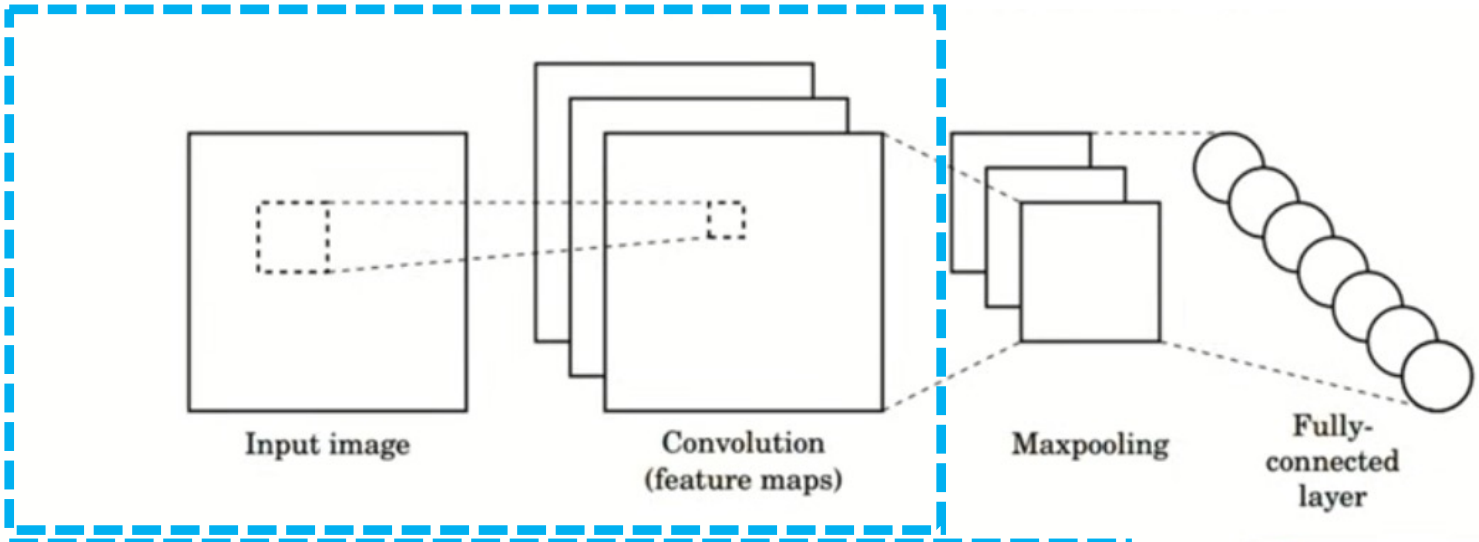
Original

Sharpen

Edge Detect

"Strong" Edge Detect

CNNs for Image Classification

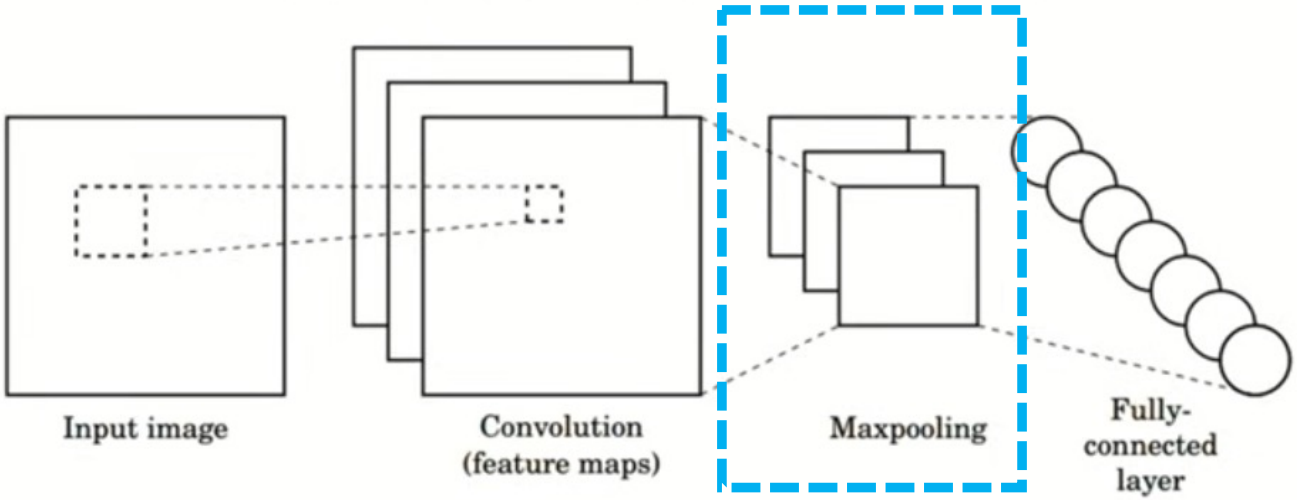


1. **Convolution**: Apply filters to generate feature maps.
2. **Non-linearity**: Often ReLU.
3. **Pooling**: Downsampling operation on each feature map.

```
torch.nn.Conv2d()
torch.nn.ReLU()
torch.nn. MaxPool2d()
```

Train model with image data.
Learn weights of filters in convolutional layers.

CNNs for Image Classification



1. **Convolution**: Apply filters to generate feature maps.
2. **Non-linearity**: Often ReLU.
3. **Pooling**: Downsampling operation on each feature map.

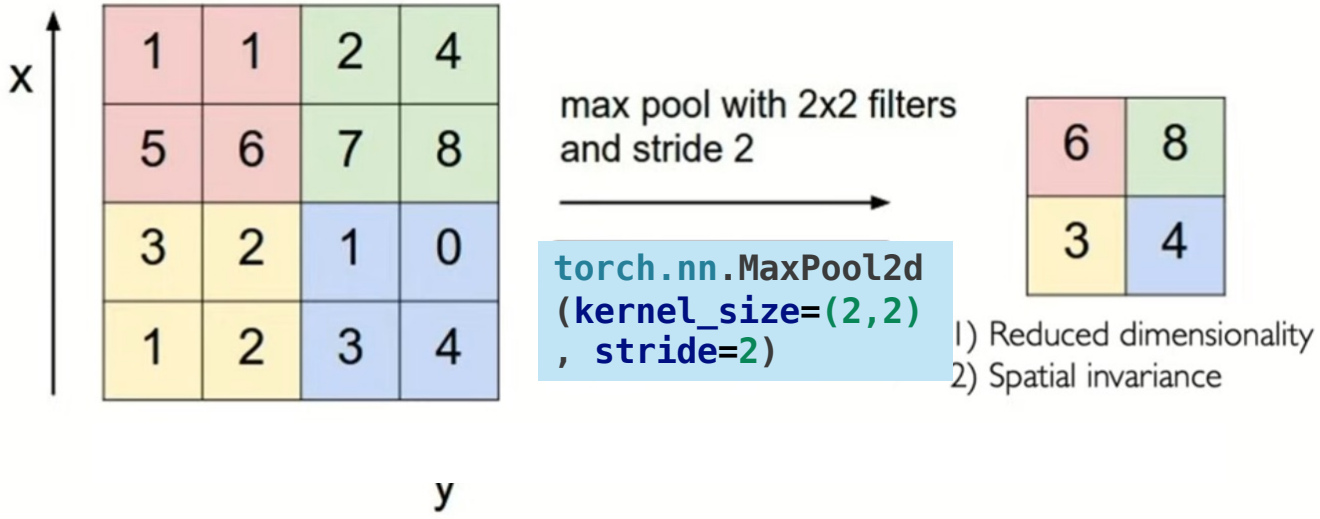
```
torch.nn.Conv2d()
torch.nn.ReLU()
torch.nn. MaxPool2d()
```

Train model with image data.
Learn weights of filters in convolutional layers.

MaxPooling

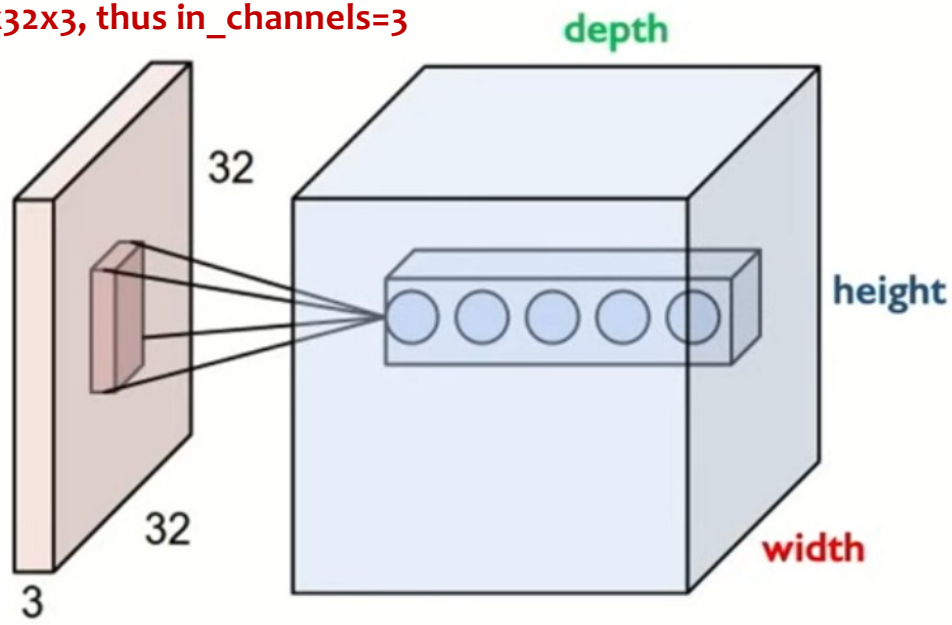
Pooling seeks to reduce the spatial size of the feature maps

- reduce # of model parameters and prevent overfitting
- provide translational or spatial invariance, making the CNN model invariant to "small" translations



Spatial Arrangement for Convolution

Input image is defined by RGB values of each pixel
32x32x3, thus in_channels=3



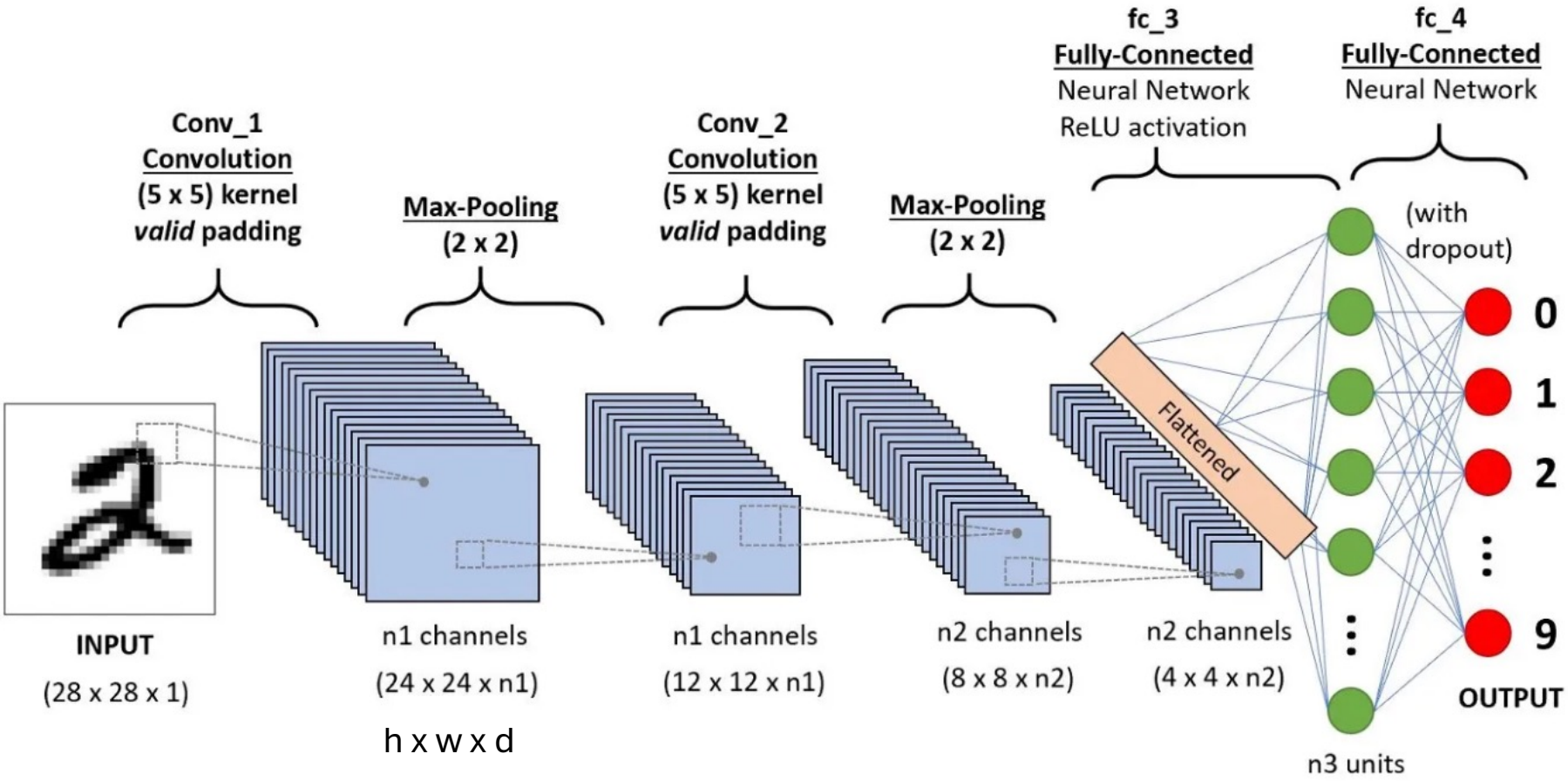
Layer Dimensions:

$h \times w \times d$

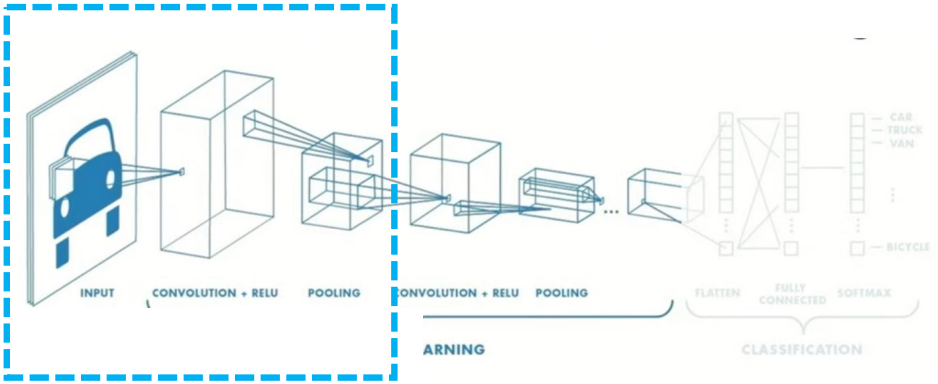
where h and w are spatial dimensions
d (depth) = number of filters

Stride:
Filter step size

```
RGB Images:      torch.nn.Conv2d(in_channels=3, out_channels=d, kernel_size=(h,w), stride=s)
Grayscale Imgs:  torch.nn.Conv2d(in_channels=1, out_channels=d, kernel_size=(h,w), stride=s)
```


Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
(with
dropout)
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
n3 units
0
1
2
9
OUTPUT
h x w x d

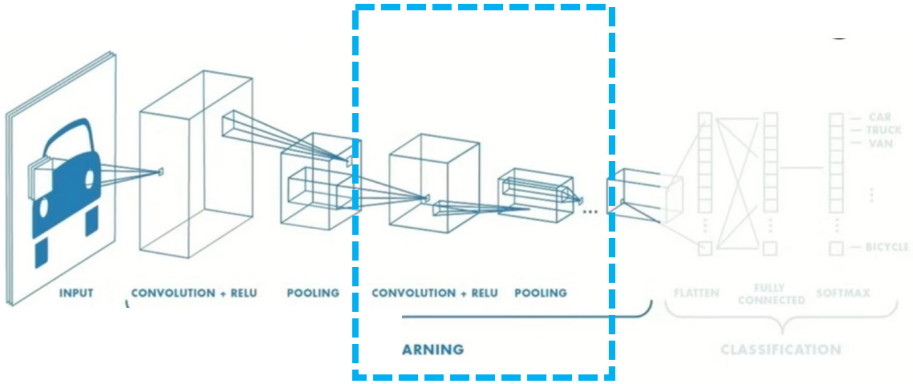
PyTorch code to specify a CNN




```
import torch.nn as nn

class Model(nn.module):
    def __init__(self):
        # first convolution layer
        self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # second convolution layer
        self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # fully connected layers
        self.fc1 = nn.Linear(in_features=16*4*4, out_features=120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

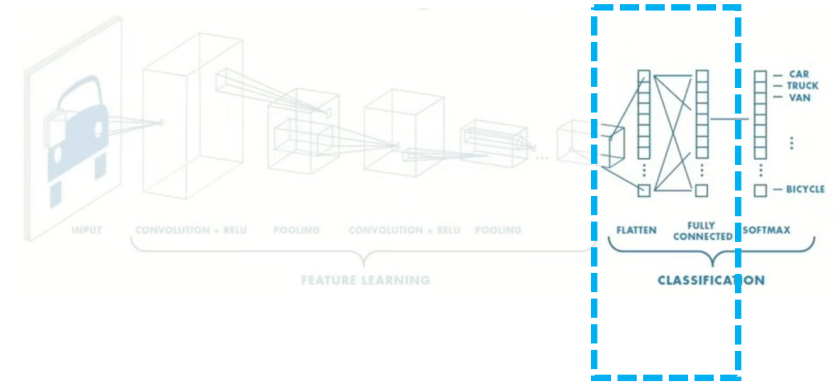
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

PyTorch code to specify a CNN




```
import torch.nn as nn

class Model(nn.module):
    def __init__(self):
        # first convolution layer
        self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # second convolution layer
        self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # fully connected layers
        self.fc1 = nn.Linear(in_features=16*4*4, out_features=120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

PyTorch code to specify a CNN




```
import torch.nn as nn

class Model(nn.module):
    def __init__(self):
        # first convolution layer
        self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # second convolution layer
        self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)
        )
        # fully connected layers
        self.fc1 = nn.Linear(in_features=16*4*4, out_features=120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

Does not include softmax

Goals of Today's Lecture

Key elements of DNN

How to specify, train, and evaluate a DNN model

Key elements of CNN

Challenges facing training a "good" DNN model

Short-comings/vulnerabilities of DNN models

Key Issues Facing DNN Models

- Model behavior depends heavily on the training data
  - More likely to make a wrong decision on unseen data
    - What is ”unseen” data
  - Bad (adversarial) training data → bad (adversarial) model behavior
    - Training data = (input sample, its label)
    - Who decides the label?

- Model training cost
  - The amount, quality, and complexity of training data
  - The size/architecture of the model
  - How to select hyperparameters
  - When to stop training

Reminder: PyTorch Info Session

- This Friday 3/28, Two Sessions: 2:30-3:30, 3:30-4:30pm
- Location: JCL 298
- Three topics:
  - Quick PyTorch overview;
  - How to install/run PyTorch on the CS dept server; SSH overview;
  - How to install/run PyTorch on local Macbooks; Jupyter notebook overview.