

# **Final Review**

**DS 4400 | Machine Learning and Data Mining I**

**Zohair Shafi**

**Spring 2026**

**Wednesday | April 15th, 2026**

# Models Seen So Far

## Supervised

Linear Regression

Naive Bayes

Logistic  
Regression

Decision Trees

k-NN

Neural Networks

LDA

CNNs

## Unsupervised

PCA

k-Means  
Clustering

Hierarchical  
Clustering

Autoencoders &  
VAE

# Models Seen So Far

## Supervised

Linear Regression

Logistic  
Regression

k-NN

LDA

Naive Bayes

Decision Trees

Neural Networks

CNNs

## Unsupervised

PCA

k-Means  
Clustering

Hierarchical  
Clustering

Autoencoders &  
VAE

# Naive Bayes Classifier

- Like LDA, Naive Bayes is a generative classifier using Bayes' theorem

$$\mathbb{P}(Y = k | X = x) = \frac{\mathbb{P}(X = x | Y = k) \cdot \mathbb{P}(Y = k)}{\mathbb{P}(X = x)}$$

*Bayes Rule*

- The challenge:

- Estimating  $\mathbb{P}(X | Y = k)$  for high-dimensional  $X$  is difficult.

# Naive Bayes Classifier

Naive Bayes assumes features are conditionally independent given the class

- Like LDA, Naive Bayes is a generative classifier using Bayes' theorem

$$\mathbb{P}(Y = k | X = x) = \frac{\mathbb{P}(X = x | Y = k) \cdot \mathbb{P}(Y = k)}{\mathbb{P}(X = x)}$$

- The challenge:
  - Estimating  $\mathbb{P}(X | Y = k)$  for high-dimensional  $X$  is difficult.
  - With  $d$  features, each taking  $v$  possible values, we'd need to estimate  $v^d$  parameters per class - **exponential in dimensionality**.

# Naive Bayes Classifier

- Example:
  - For spam classification with words “free” and “money”
    - Reality:  $\mathbb{P}(\text{"free", "money" | spam}) \neq \mathbb{P}(\text{"free" | spam}) \cdot \mathbb{P}(\text{"money" | spam})$ 
      - These words are correlated in spam emails
      - Naive Bayes pretends they're independent

# Naive Bayes Classifier

Want to predict:

$$\mathbb{P}(Y = k | X = x) = \frac{\mathbb{P}(X = x | Y = k) \cdot \mathbb{P}(Y = k)}{\mathbb{P}(X = x)}$$

Naive Assumption:

$$\mathbb{P}(X = x | Y = k) = \mathbb{P}(x_1, x_2, x_3, \dots, x_d | Y = k) = \prod_{j=1}^d \mathbb{P}(x_j | Y = k)$$

multiplication  
operator.

# Naive Bayes Classifier

How do you find  $\mathbb{P}(x_j | Y = k)$ ?

# Naive Bayes Classifier

$$\mu_{kj} = \frac{1}{N_k} \sum_{i:y_i=k} x_{ij}$$

$$\sigma_{kj}^2 = \frac{1}{N_k} \sum_{i:y_i=k} (x_{ij} - \mu_{kj})^2$$

## Gaussian Naive Bayes

For continuous features, assume each feature follows a Gaussian Distribution

$$\mathbb{P}(x_j | Y = k) = \frac{1}{\sqrt{2\pi\sigma_{kj}^2}} \cdot e^{-\frac{(x_j - \mu_{kj})^2}{2\sigma_{kj}^2}}$$

Each class-feature combination has its own mean  $\mu_{kj}$  and variance  $\sigma_{kj}^2$

# Naive Bayes Classifier

## Multinomial Naive Bayes

For discrete features/count data like word frequencies

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j}$$

Where  $\theta_{kj}$  is the probability of feature  $j$  in class  $k$  and  $x_j$  is the count of feature  $j$

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

# Naive Bayes Classifier

## Bernoulli Naive Bayes

For binary data like word frequencies

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j} \cdot (1 - \theta_{kj})^{1-x_j}$$

Where  $\theta_{kj}$  is the probability of feature  $j$  in class  $k$  and  $x_j$  is the count of feature  $j$

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total samples in class } k}$$

# Naive Bayes Classifier

## Example

### Task

Classify emails as spam or not spam

### Training Data

*4 dimensional features.*

Document	x_1 = "free"	x_2 = "money"	x_3 = "meeting"	x_4 = "lunch"	Class
1	3	2	0	0	spam
2	2	1	0	0	spam
3	0	0	2	1	not spam
4	0	0	1	2	not spam

# Naive Bayes Classifier

## Example

### Task

Classify emails as spam or not spam

### Training Data

Document	x_1 = “free”	x_2 = “money”	x_3 = “meeting”	x_4 = “lunch”	Class
1	3	2	0	0	spam
2	2	1	0	0	spam
3	0	0	2	1	not spam
4	0	0	1	2	not spam

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(\text{spam}) = \frac{2}{4} = 0.5$$

$$\mathbb{P}(\text{not spam}) = \frac{2}{4} = 0.5$$

# Naive Bayes Classifier

## Example

### Task

Classify emails as spam or not spam

### Training Data

Document	x_1 = "free"	x_2 = "money"	x_3 = "meeting"	x_4 = "lunch"	Class
1	3	2	0	0	spam
2	2	1	0	0	spam
3	0	0	2	1	not spam
4	0	0	1	2	not spam

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(\text{spam}) = \frac{2}{4} = 0.5$$

$$\mathbb{P}(\text{not spam}) = \frac{2}{4} = 0.5$$

For spam class:

### Laplace Smoothing

$$\mathbb{P}(\text{"free"}|\text{spam}) = \frac{3 + 1}{8} = \frac{4}{8} = 0.5$$

# Naive Bayes Classifier

## Example

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

### Task

Classify emails as spam or not spam

### Training Data

Document	x_1 = "free"	x_2 = "money"	x_3 = "meeting"	x_4 = "lunch"	Class
1	3	2	0	0	spam
2	2	1	0	0	spam
3	0	0	2	1	not spam
4	0	0	1	2	not spam

$$\mathbb{P}(\text{spam}) = \frac{2}{4} = 0.5$$

$$\mathbb{P}(\text{not spam}) = \frac{2}{4} = 0.5$$

For **spam** class:

$$\bullet \mathbb{P}(\text{"free"}|\text{spam}) = \frac{(5+1)}{(8)} = \frac{6}{8} = 0.75$$

$$\bullet \mathbb{P}(\text{"money"}|\text{spam}) = \frac{(3+1)}{(8)} = \frac{4}{8} = 0.5$$

$$\bullet \mathbb{P}(\text{"meeting"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$$

$$\bullet \mathbb{P}(\text{"lunch"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$$

# Naive Bayes Classifier

## Example

### Task

Classify emails as spam or not spam

### Training Data

Document	x_1 = "free"	x_2 = "money"	x_3 = "meeting"	x_4 = "lunch"	Class
1	3	2	0	0	spam
2	2	1	0	0	spam
3	0	0	2	1	not spam
4	0	0	1	2	not spam

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(\text{spam}) = \frac{2}{4} = 0.5$$

$$\mathbb{P}(\text{not spam}) = \frac{2}{4} = 0.5$$

For **not spam** class:

- $\mathbb{P}(\text{"free"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"money"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"meeting"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$
- $\mathbb{P}(\text{"lunch"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$

# Naive Bayes Classifier

## Example

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j}$$

For **spam** class:

- $\mathbb{P}(\text{"free"}|\text{spam}) = \frac{(5+1)}{(8)} = \frac{6}{8} = 0.75$
- $\mathbb{P}(\text{"money"}|\text{spam}) = \frac{(3+1)}{(8)} = \frac{4}{8} = 0.5$
- $\mathbb{P}(\text{"meeting"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$
- $\mathbb{P}(\text{"lunch"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$

**New Email:**  $x = \text{"free money"}$

For **not spam** class:

- $\mathbb{P}(\text{"free"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"money"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"meeting"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$
- $\mathbb{P}(\text{"lunch"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$

# Naive Bayes Classifier

## Example

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j}$$

For **spam** class:

- $\mathbb{P}(\text{"free"}|\text{spam}) = \frac{(5+1)}{(8)} = \frac{6}{8} = 0.75$
- $\mathbb{P}(\text{"money"}|\text{spam}) = \frac{(3+1)}{(8)} = \frac{4}{8} = 0.5$
- $\mathbb{P}(\text{"meeting"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$
- $\mathbb{P}(\text{"lunch"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$

**New Email:**  $x = \text{"free money"}$

$$\mathbb{P}(\text{spam} | x) = \mathbb{P}(Y = \text{spam}) \cdot \prod_{j=1}^d \mathbb{P}(x_j | Y = \text{spam})$$

For **not spam** class:

- $\mathbb{P}(\text{"free"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"money"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"meeting"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$
- $\mathbb{P}(\text{"lunch"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$

# Naive Bayes Classifier

## Example

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j}$$

For **spam** class:

- $\mathbb{P}(\text{"free"}|\text{spam}) = \frac{(5+1)}{(8)} = \frac{6}{8} = 0.75$
- $\mathbb{P}(\text{"money"}|\text{spam}) = \frac{(3+1)}{(8)} = \frac{4}{8} = 0.5$
- $\mathbb{P}(\text{"meeting"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$
- $\mathbb{P}(\text{"lunch"}|\text{spam}) = \frac{(0+1)}{(8)} = \frac{1}{8} = 0.125$

**New Email:**  $x = \text{"free money"}$

$$\mathbb{P}(\text{spam} | x) = \mathbb{P}(Y = \text{spam}) \cdot \prod_{j=1}^d \mathbb{P}(x_j | Y = \text{spam})$$

$$\mathbb{P}(\text{spam} | x) = 0.5 \cdot (0.75)^1 \cdot (0.5)^1 = 0.1875$$

$$\mathbb{P}(\text{not spam} | x)$$

For **not spam** class:

- $\mathbb{P}(\text{"free"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"money"}|\text{not spam}) = \frac{(0+1)}{(6)} = \frac{1}{6} = 0.166$
- $\mathbb{P}(\text{"meeting"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$
- $\mathbb{P}(\text{"lunch"}|\text{not spam}) = \frac{(3+1)}{(6)} = \frac{4}{6} = 0.66$

# Naive Bayes Classifier

## Example

$$\theta_{kj} = \frac{\text{count of feature } j \text{ in class } k}{\text{total count of all features in class } k}$$

$$\mathbb{P}(x_j | Y = k) = \theta_{kj}^{x_j}$$

$\mathbb{P}(\text{spam} | x)$  —  
 $\mathbb{P}(\text{not spam} | x)$  —

For **spam** class:

$$\bullet \mathbb{P}(\text{"free"} | \text{spam}) = \frac{(5 + 1)}{(8)} = \frac{6}{8} = 0.75$$

$$\bullet \mathbb{P}(\text{"money"} | \text{spam}) = \frac{(3 + 1)}{(8)} = \frac{4}{8} = 0.5$$

$$\bullet \mathbb{P}(\text{"meeting"} | \text{spam}) = \frac{(0 + 1)}{(8)} = \frac{1}{8} = 0.125$$

$$\bullet \mathbb{P}(\text{"lunch"} | \text{spam}) = \frac{(0 + 1)}{(8)} = \frac{1}{8} = 0.125$$

**New Email:**  $x = \text{"free money"}$

$$\mathbb{P}(\text{spam} | x) = \mathbb{P}(Y = \text{spam}) \cdot \prod_{j=1}^d \mathbb{P}(x_j | Y = \text{spam})$$

$$\mathbb{P}(\text{spam} | x) = 0.5 \cdot (0.75)^1 \cdot (0.5)^1 = 0.1875$$

$$\mathbb{P}(\text{not spam} | x) = \mathbb{P}(Y = \text{not spam}) \cdot \prod_{j=1}^d \mathbb{P}(x_j | Y = \text{not spam})$$

$$\mathbb{P}(\text{not spam} | x) = 0.5 \cdot (0.166)^1 \cdot (0.166)^1 = 0.013$$

For **not spam** class:

$$\bullet \mathbb{P}(\text{"free"} | \text{not spam}) = \frac{(0 + 1)}{(6)} = \frac{1}{6} = 0.166$$

$$\bullet \mathbb{P}(\text{"money"} | \text{not spam}) = \frac{(0 + 1)}{(6)} = \frac{1}{6} = 0.166$$

$$\bullet \mathbb{P}(\text{"meeting"} | \text{not spam}) = \frac{(3 + 1)}{(6)} = \frac{4}{6} = 0.66$$

$$\bullet \mathbb{P}(\text{"lunch"} | \text{not spam}) = \frac{(3 + 1)}{(6)} = \frac{4}{6} = 0.66$$

# Naive Bayes Classifier

## Pros

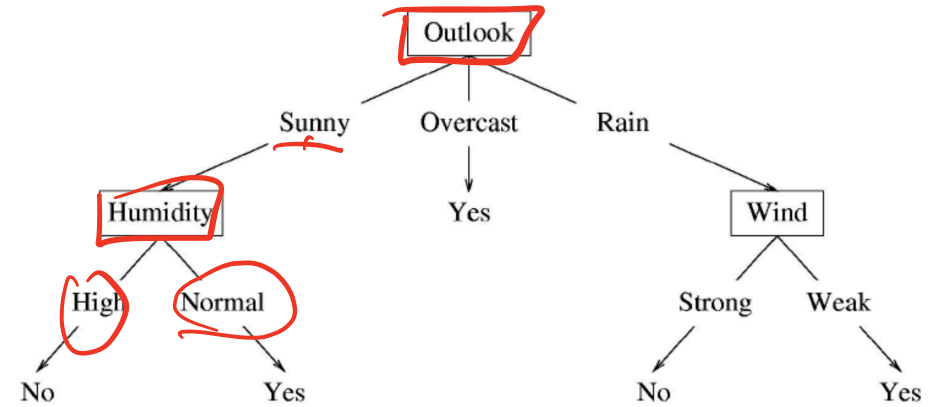
- Extremely fast training (just counting)
- Fast prediction
- Handles high-dimensional data well
- Works with **small training** sets
- Handles missing features naturally
- Easy to implement and interpret
- Often surprisingly accurate

## Cons

- Independence assumption is usually wrong
- Probability estimates are unreliable
- Cannot learn feature interactions
- Continuous features require distributional assumptions
- Correlated features are “double-counted”

# Decision Trees

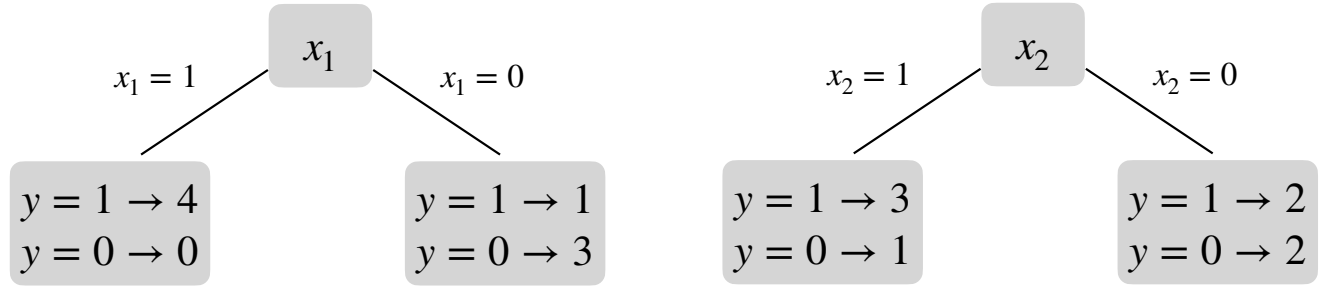
Play Outside? (y)	Humidity (X1)	Wind (X2)	Outlook (X3)
Yes	Normal	Strong	Sunny
Yes	Low	Weak	Overcast
No	High	Weak	Rain



# Decision Trees

*Impurity*  
*Entropy* ←  
*Measures*  
*Gain*

$y$	$x_1$	$x_2$
1	1	1
1	1	0
1	1	1
1	1	0
1	0	1
0	0	0
0	0	1
0	0	0



How do we decide which is better?

## Entropy Measures - Information Gain

Use counts at leaf nodes to define probabilities, so we can measure uncertainty

*Impurity*

# Entropy

- Entropy measures uncertainty or **surprise**.
- The **more uncertain** you are about an outcome, the **higher the entropy**.
- Entropy is maximized when all outcomes are equally likely.
- For outcomes with probabilities  $p_1, p_2, \dots, p_n$ :

$$H = - \sum_i p_i \log_2(p_i)$$

# Entropy

- Why Entropy Matters in Decision Trees
  - We want splits that reduce uncertainty about the class label.
  - A split that creates pure nodes (all one class) reduces entropy to zero.
    - **Before split:** Mixed classes, high entropy
    - **After good split:** Purer nodes, lower entropy
  - Information gain = entropy reduction

# Entropy

- Node 1 has 75% class A, 25% class B:

$$H = -0.75\log_2(0.75) - 0.25\log_2(0.25) = 0.811 \text{ bits}$$

- Node 2 has 50% class A and 50% class B:

$$H = -0.5\log_2(0.5) - 0.5\log_2(0.5) = 1 \text{ bit}$$

- Node 1 has lower entropy (less uncertainty) than node 2.

# Measuring Split Quality: Impurity Functions

- A good split separates classes.
- We measure node **impurity** - how mixed the classes are - and choose splits that maximize impurity reduction.

# Measuring Split Quality: Impurity Functions

## Gini Impurity

- Let  $p_k$  be the proportion of class  $k$  samples in a node

$$Gini(D) = 1 - \sum_{k=1}^K p_k^2 = \sum_{k=1}^K p_k(1 - p_k)$$

- Interpretation:** Probability of misclassifying a randomly chosen sample if labeled according to the class distribution.

- Properties:**

- Minimum = 0 when node is pure (all one class)
- Maximum =  $1 - \frac{1}{K}$  when classes are equally distributed
- For binary: max = 0.5 at  $p = 0.5$

- Example (binary classification):**

- Node with 100% class A: Gini =  $1 - 1^2 = 0$  (pure)
- Node with 50% each: Gini =  $1 - 0.5^2 - 0.5^2 = 0.5$  (maximum impurity)
- Node with 90% class A: Gini =  $1 - 0.9^2 - 0.1^2 = 0.18$

# Measuring Split Quality: Impurity Functions

## Information Gain

- We want splits that reduce **impurity** (i.e., Gini, Entropy).
- Information gain measures this reduction:

$$Gain(D, split) = \text{Impurity}(D) - \sum_{k \in \text{classes}} \frac{|D_k|}{|D|} \cdot \text{Impurity}(D_k)$$

# Measuring Split Quality: Impurity Functions

## Information Gain

- We want splits that reduce **impurity** (i.e., Gini, Entropy).
- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{classes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

weighted average impurity of child nodes


# Measuring Split Quality: Impurity Functions

## Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

100 Days  
60 Play, 40 Don't Play



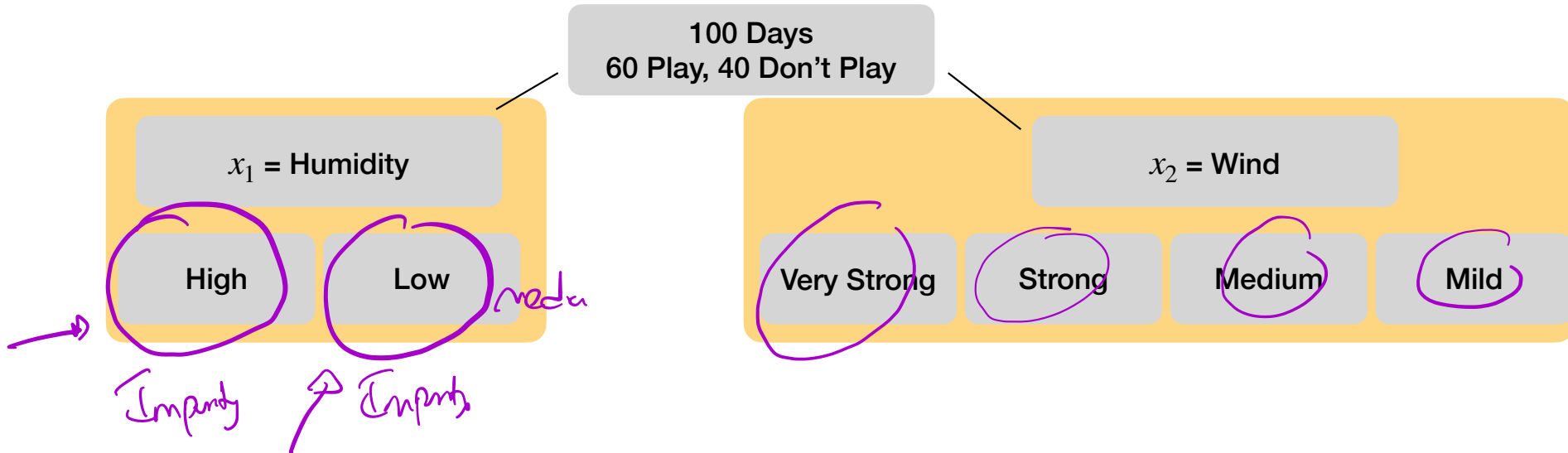
# Measuring Split Quality: Impurity Functions

## Information Gain

Step 1: Decide between Wind and Humidity

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

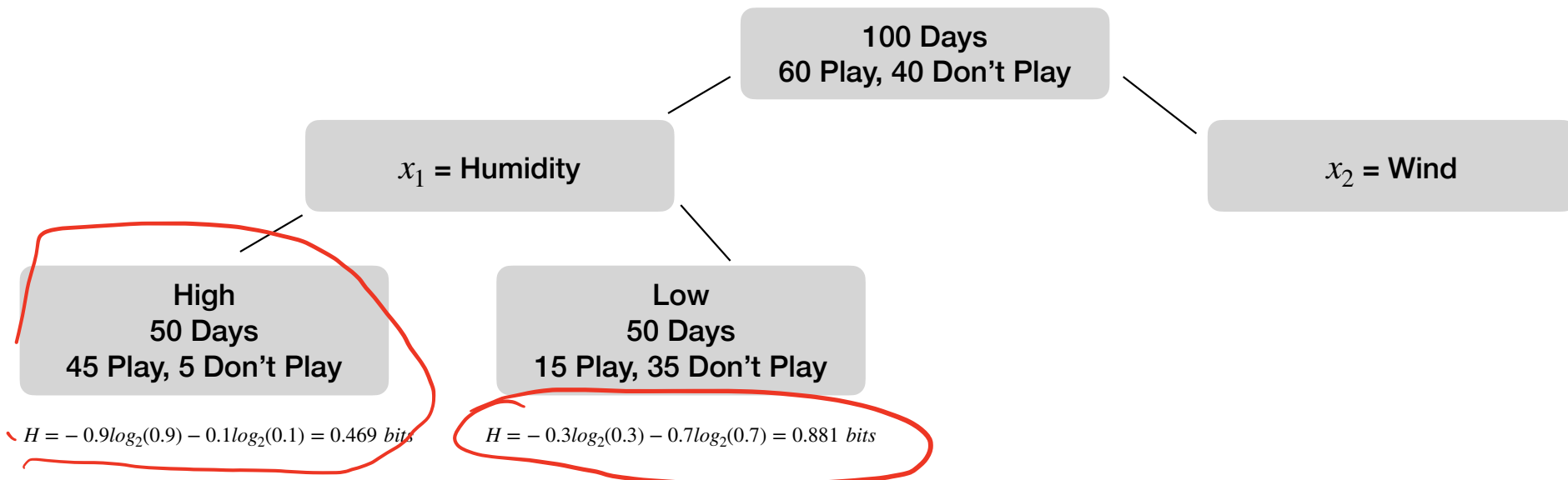


# Measuring Split Quality: Impurity Functions

## Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$

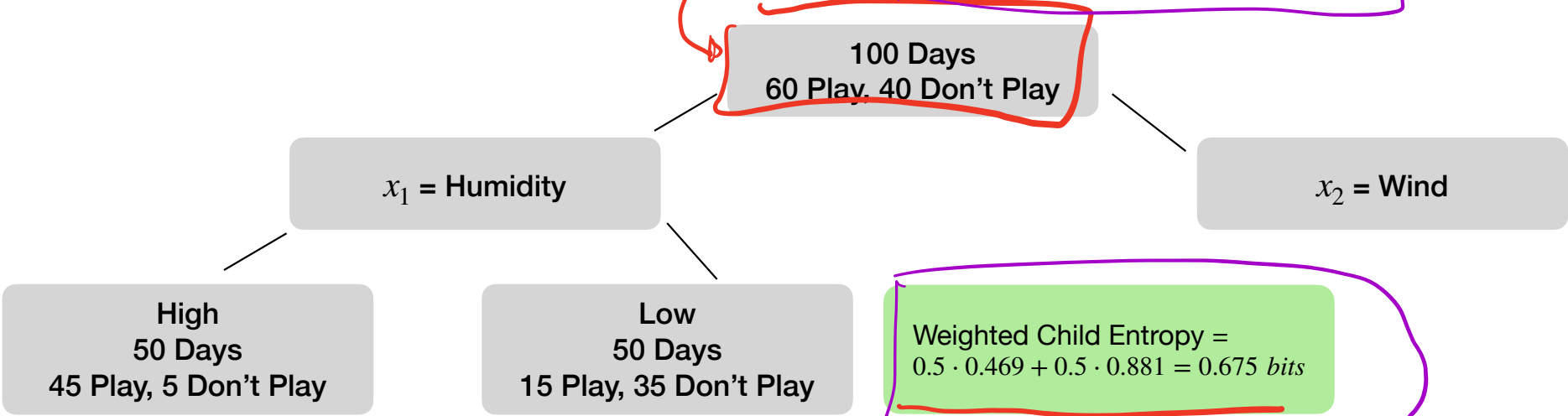


# Measuring Split Quality: Impurity Functions

## Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = Impurity(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot Impurity(D_k)$$



$$H = -0.9 \log_2(0.9) - 0.1 \log_2(0.1) = 0.469 \text{ bits}$$

$$H = -0.3 \log_2(0.3) - 0.7 \log_2(0.7) = 0.881 \text{ bits}$$

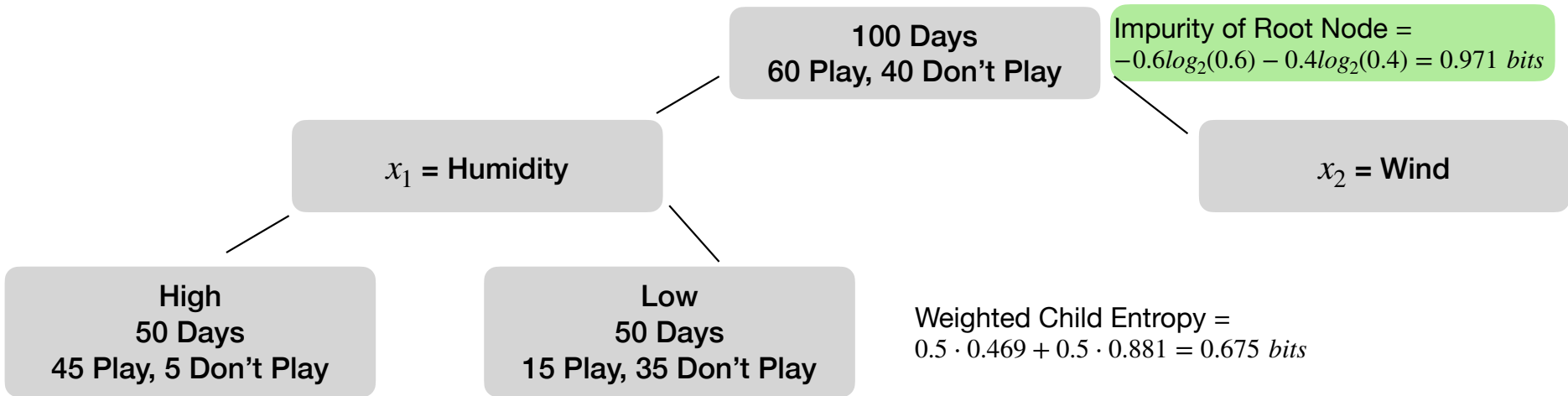
$$\text{Weighted Child Entropy} = 0.5 \cdot 0.469 + 0.5 \cdot 0.881 = 0.675 \text{ bits}$$

# Measuring Split Quality: Impurity Functions

## Information Gain

- Information gain measures this reduction:

$$Gain(D, split) = \text{Impurity}(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot \text{Impurity}(D_k)$$



$$H = -0.9 \log_2(0.9) - 0.1 \log_2(0.1) = 0.469 \text{ bits}$$

$$H = -0.3 \log_2(0.3) - 0.7 \log_2(0.7) = 0.881 \text{ bits}$$

# Measuring Split Quality: Impurity Functions

## Information Gain

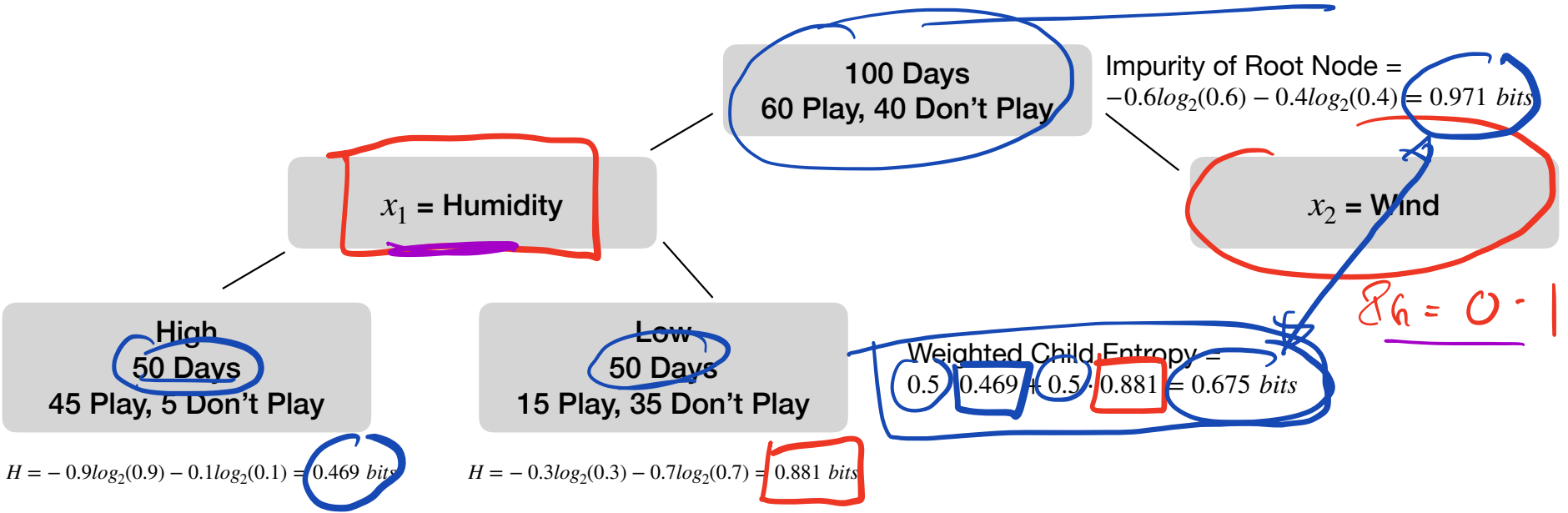
Information Gain = 0.971 - 0.675 = **0.296 bits**

Gini or Entropy.

Gini or Entropy

- Information gain measures this reduction:

$$Gain(D, split) = \text{Impurity}(D) - \sum_{k \in \text{outcomes}} \frac{|D_k|}{|D|} \cdot \text{Impurity}(D_k)$$



# Stopping Criteria

## When to stop splitting and create a leaf

Hyperparameters

- **Maximum depth reached:**  $\text{depth} \geq \text{max\_depth}$
- **Minimum samples:** node has fewer than **min\_samples\_split** samples
- **Minimum samples per leaf:** split would create child with  $< \text{min\_samples\_leaf}$
- **Pure node:** all samples belong to same class
- **No information gain:** best split doesn't improve impurity
- **Maximum leaf nodes:** tree already has **max\_leaf\_nodes** leaves

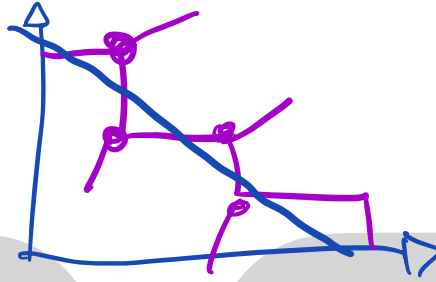
# Stopping Criteria

## Overfitting & Pruning

- Deep trees overfit - they memorize training data, **creating leaves with very few samples.**
- Signs of overfitting:
  - Training accuracy  $\approx 100\%$
  - Test accuracy much lower
  - **Very deep tree with many leaves**

# Decision Trees

## Pros and Cons



### Pros

- Highly interpretable (visualize as flowchart)
- No feature scaling required
- Handles numerical and categorical features
- Handles missing values
- Captures non-linear relationships
- Automatic feature selection
- Fast prediction:  $O(\text{depth})$

### Cons

- High variance (unstable)
- Prone to overfitting
- Greedy algorithm (no global optimum)
- Axis-aligned splits only (can't capture diagonal boundaries efficiently)
- Biased toward high-cardinality features

# Ensemble Methods

## The Fundamental Idea

- A single decision tree is unstable and prone to overfitting.
- Ensemble methods combine multiple trees to create a more robust model.
- **Key Insight:**
  - Combining many “weak” learners can create a “strong” learner.
- **Analogy:**

Asking 100 people to estimate something and averaging their answers is often more accurate than asking one expert.

# Ensemble Methods

## The Fundamental Idea

Method	Strategy	Trees Trained	Key Idea
Bagging	<u>Parallel</u>	<u>Independently</u>	Reduce variance via averaging
Random Forest	Parallel	Independently + High Depth	Bagging + <u>random feature subsets</u>
<u>Boosting</u>	Sequential	Each corrects previous	Reduce bias via focusing on errors

# Bagging

## Bootstrap Aggregating

diff. Cross validation.

- Bootstrap sampling:
  - Sample  $N$  points with replacement from a dataset of  $N$  points.
- **Bagging Algorithm**
  - Create  $B$  bootstrap samples from the training data
  - Train one decision tree on each bootstrap sample
  - Combine predictions:
    - Classification: Majority vote
    - Regression: Average

# Bagging

## Evaluation - Out-of-Bag (OOB)

- For each sample  $x_i$ , find all trees that **did not** include it in their bootstrap sample
- Get predictions from **only those trees**
- Aggregate these predictions
- Compute error across all samples
- OOB error approximates test error without needing a separate validation set.

# Random Forests

## Pros and Cons

### Pros

- Excellent accuracy out-of-the-box
- Robust to overfitting
- Handles high-dimensional data
- Provides feature importance
- Built-in OOB validation
- Parallelizable (trees are independent)
- Minimal tuning required

### Cons

- Less interpretable than single tree
- Can be slow for very large datasets
- Not optimal for sparse, high-dimensional data (like text)

# Boosting

- Unlike bagging (parallel, independent trees), boosting trains trees **sequentially**
  - Each subsequent tree trying to **correct the errors of the previous trees.**
- **Intuition:** Each tree focuses on the samples that previous trees got wrong.

# AdaBoost - Adaptive Boosting

- **Key idea:**
  - Maintain weights on samples.
  - Increase weights on misclassified samples so the next tree focuses on them.

# AdaBoost - Adaptive Boosting Algorithm

- Initialize weights:  $w_i = \frac{1}{N}$  for all samples ✓/1000
- For  $t = 1 \rightarrow T$ :
  - Train weak learner  $h_t$  on weighted data
  - Compute weighted error:  $\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$
  - Compute tree weight:  $\alpha_t = \frac{0.5 \cdot \ln(1 - \epsilon_t)}{\epsilon_t}$
  - Update sample weights:  $w_i \leftarrow w_i \cdot \exp(-\alpha_t \cdot y_i \cdot h_t(x_i))$  (Normalize so weights sum to 1)
- Final prediction:  $H(x) = \text{sign}(\sum \alpha_t \cdot h_t(x))$

# AdaBoost - Adaptive Boosting

- Intuition Behind the Updates

- Tree weight  $\alpha_t$  :

- If error  $\epsilon_t = 0.5$  (random):

- $\alpha_t = 0$  (ignore this tree)

- If error  $\epsilon_t \rightarrow 0$  (perfect):

- $\alpha_t \rightarrow \infty$  (trust this tree completely)

- If error  $\epsilon_t \rightarrow 1$  (anti-correlated)

- $\alpha_t \rightarrow -\infty$  (flip its predictions)

$$\epsilon_t = \frac{\sum w_i \cdot I(y_i \neq h_t(x_i))}{\sum w_i}$$

$$\alpha_t = \frac{0.5 \cdot \ln(1 - \epsilon_t)}{\epsilon_t}$$

# When to use what?

## Random Forest

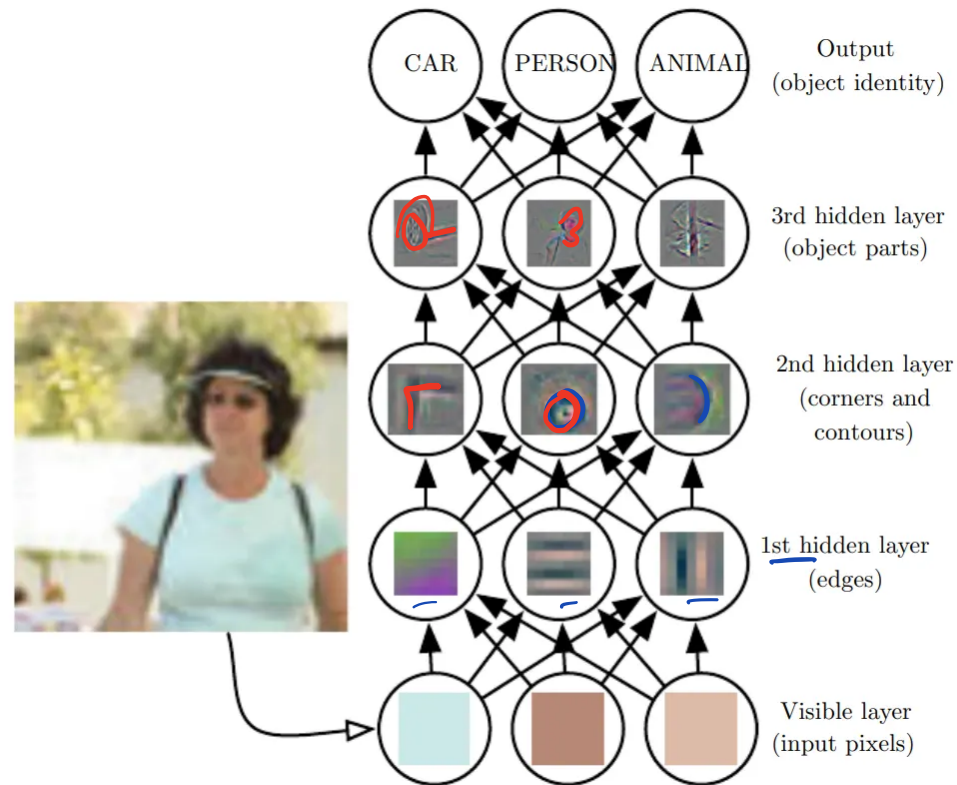
- Good default choice
- When interpretability (feature importance) matters
- When you want robustness without tuning
- When you have noisy labels

## Gradient Boosting (XGBoost / LightGBM / CatBoost)

- When you need maximum accuracy
- Structured/tabular data competitions
- When you have time to tune hyperparameters
- When training data is clean

# Deep Learning

- Why is deep learning needed?
  - Learnable transformations of data
- Limitations of Linear Models - can only learn **linear** decision boundaries



# Deep Learning

## Composing Functions

$$\left. \begin{array}{l} h = (w_1 x + b_1) \quad \text{Lin Reg} \quad \textcircled{1} \\ y = w_2 h + b_2 \quad \text{Lin Reg} \quad \textcircled{2} \end{array} \right\}$$

Solution: Add **non-linearity** between compositions

$$h = \sigma(w_1 x + b_1) \quad \textcircled{1}$$

$$\hat{y} = w_2 h + b_2 \quad \textcircled{2}$$

$$\begin{aligned} y &= w_2 (w_1 x + b_1) + b_2 \\ &= \underbrace{w_2 w_1}_w x + \dots + b_2 \end{aligned}$$

# Deep Learning

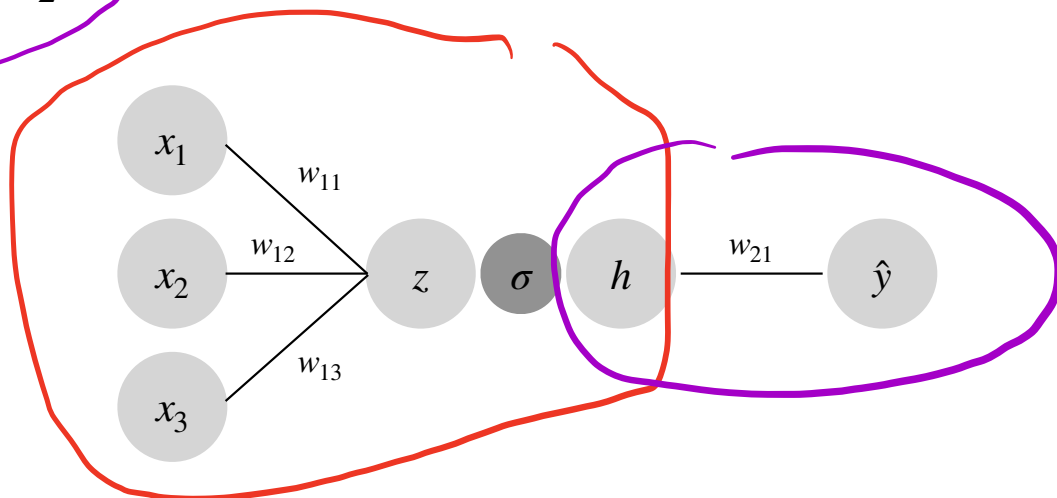
## Composing Functions

$$x = [x_1 \quad x_2 \quad x_3]$$

$$h = \sigma(W_1 x + b_1)$$

$$W_1 = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix} \quad W_2 = [w_{21}]$$

$$\hat{y} = W_2 h + b_2$$

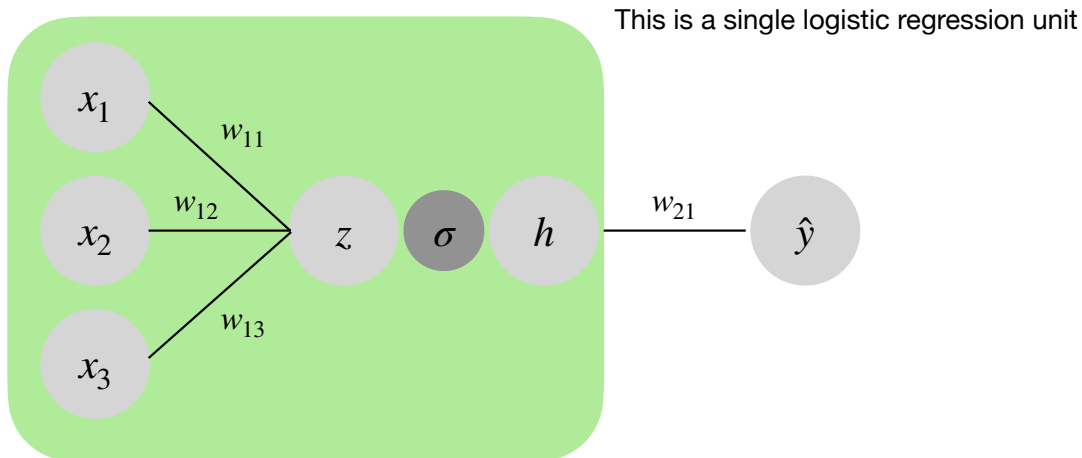


# Deep Learning

## Composing Functions

$$x = [x_1 \quad x_2 \quad x_3]$$

$$h = \sigma(W_1x + b_1)$$
$$\hat{y} = W_2h + b_2$$
$$W_1 = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix} \quad W_2 = [w_{21}]$$



# Deep Learning

## Composing Functions

$$h = \sigma(W_1x + b_1)$$

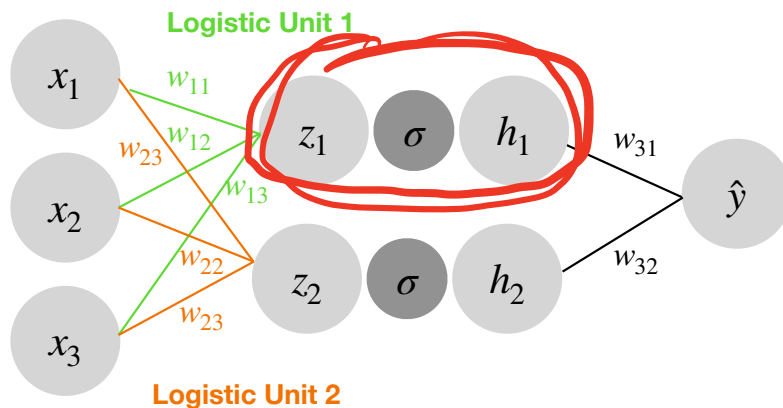
$$\hat{y} = W_2h + b_2$$

$x = [x_1 \ x_2 \ x_3]$

$W_1 = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}$

$W_2 = \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix}$

*3x2*



# Deep Learning

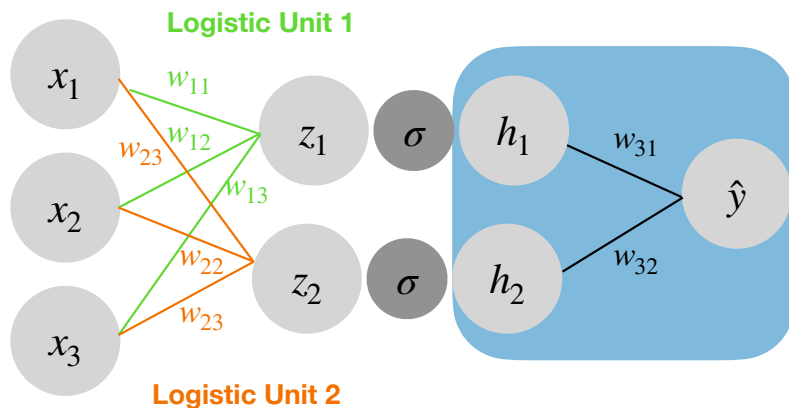
## Composing Functions

$$x = [x_1 \quad x_2 \quad x_3]$$

$$h = \sigma(W_1 x + b_1)$$

$$W_1 = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \quad W_2 = \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix}$$

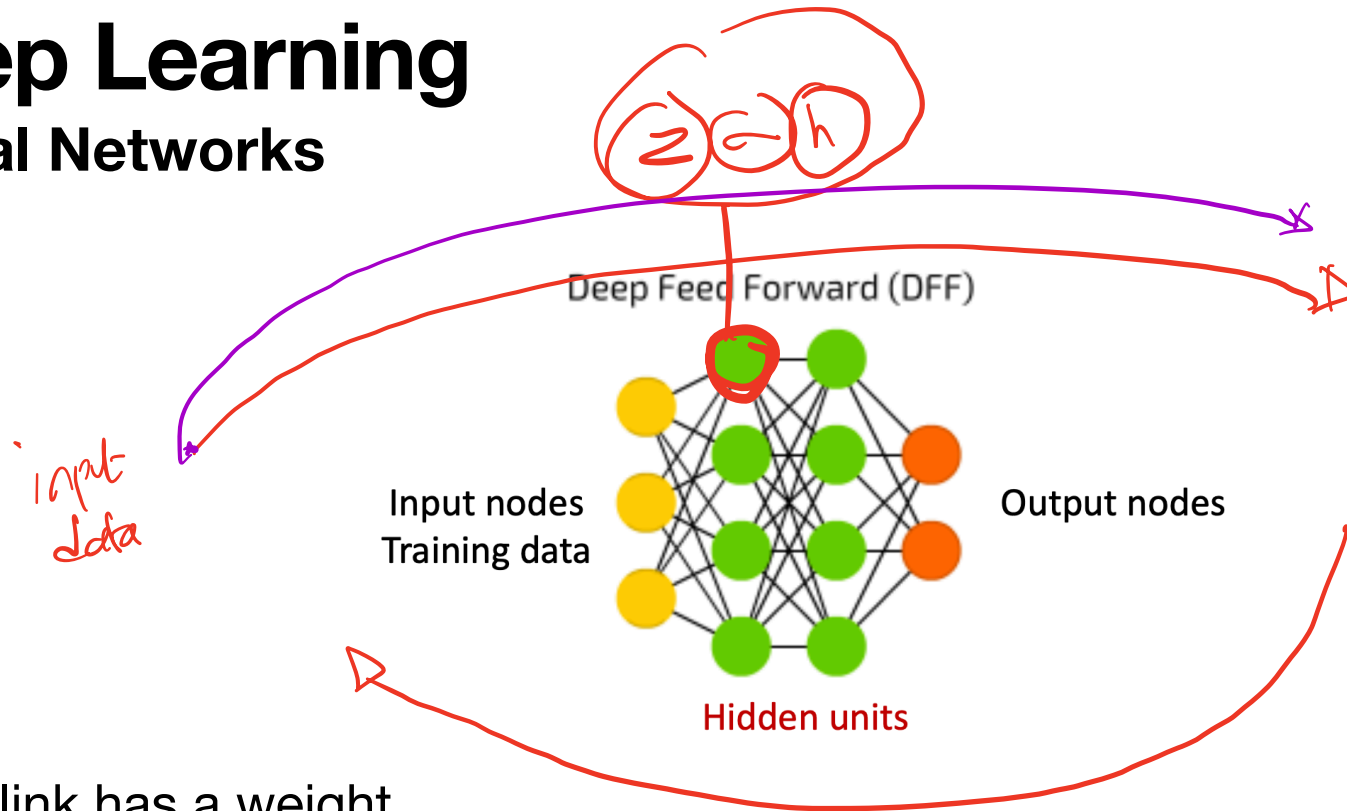
$$\hat{y} = W_2 h + b_2$$



**This is also a Logistic Unit!**  
It takes as **input** the outputs of the previous layer

# Deep Learning

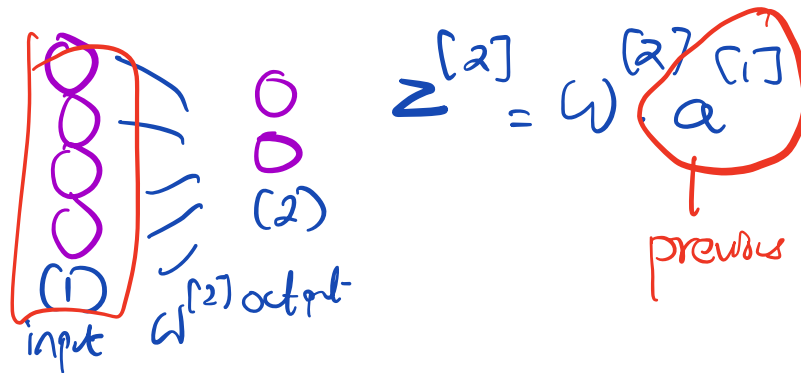
## Neural Networks



- Each link has a weight
- Each hidden unit has an activation function
- Training data is input in the left, output is generated on the right

# Deep Learning

## Multi Layer Perceptron (MLP)



Number of layers  $L = 3$

Computation:

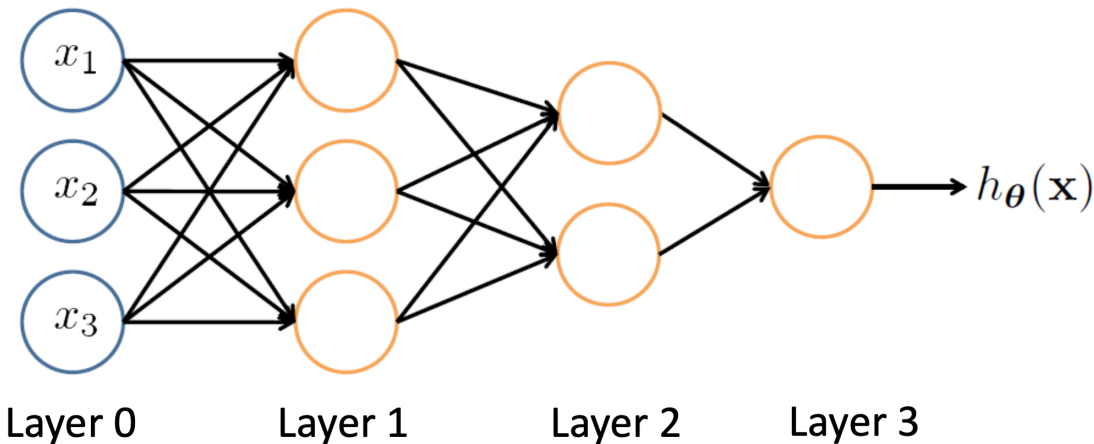
$$a^{[0]} = x \text{ (input)}$$

For each layer  $l = 1, 2, \dots, L$ :

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \text{ (linear transform)}$$


$$a^{[l]} = \sigma^{[l]}(z^{[l]}) \text{ (activation)}$$

Final Output:  $h_{\theta}(x) = \hat{y} = a^{[L]}$



# Deep Learning

## Activation Functions

- Why Activation Functions Matter
  - Introduce non-linearity (**essential for learning complex patterns**)
  - Control the range of outputs 
  - Affect gradient flow during training

# Deep Learning

## Activation Functions

- Sigmoid Function

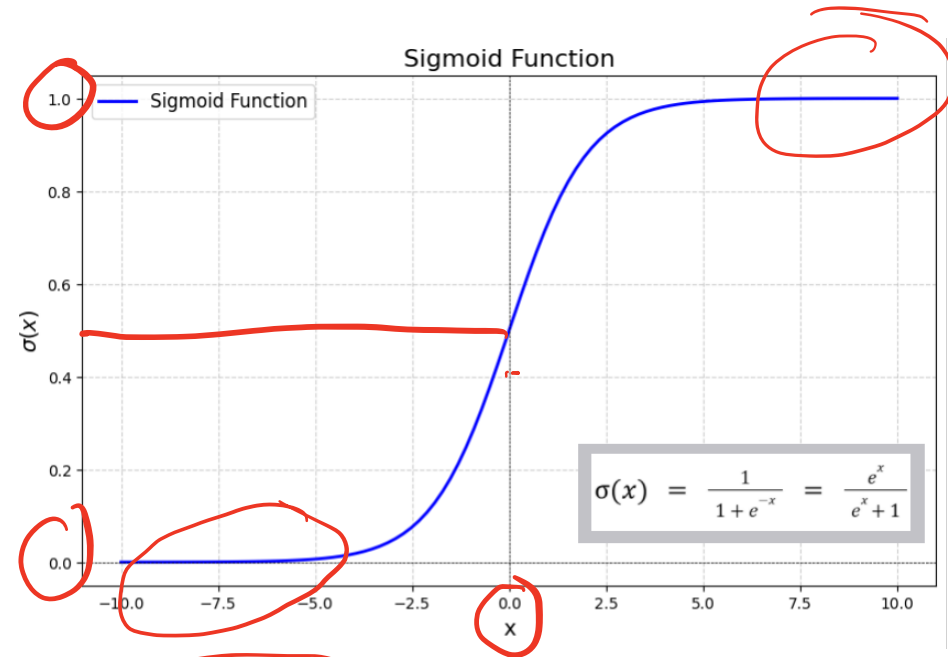
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

**Range:** [0, 1]

**Pros:** Smooth, bounded, interpretable as probability

**Cons:** Vanishing gradients (saturates at extremes), not zero-centered, expensive exponential operation, Slow due to exponentiation operation.

**Use:** Output layer for binary classification



# Deep Learning

## Activation Functions

- Tanh Function

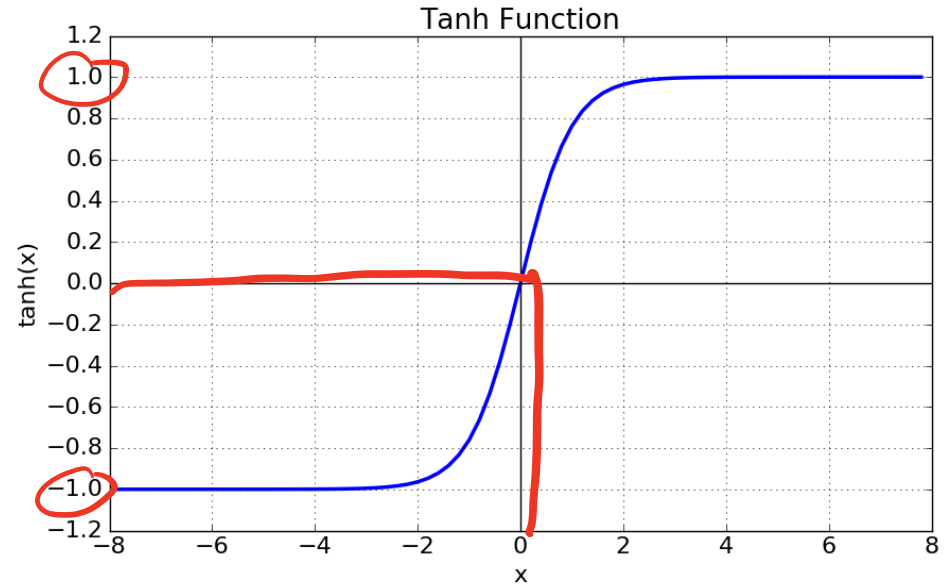
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

**Range:** [-1, 1]

**Pros:** Zero-centered (better than sigmoid), stronger gradients

**Cons:** Still has vanishing gradient problem. Slow due to exponentiation operation.

**Use:** Hidden layers (historically), RNNs



# Deep Learning

## Activation Functions

- ReLU Function (Rectified Linear Unit)

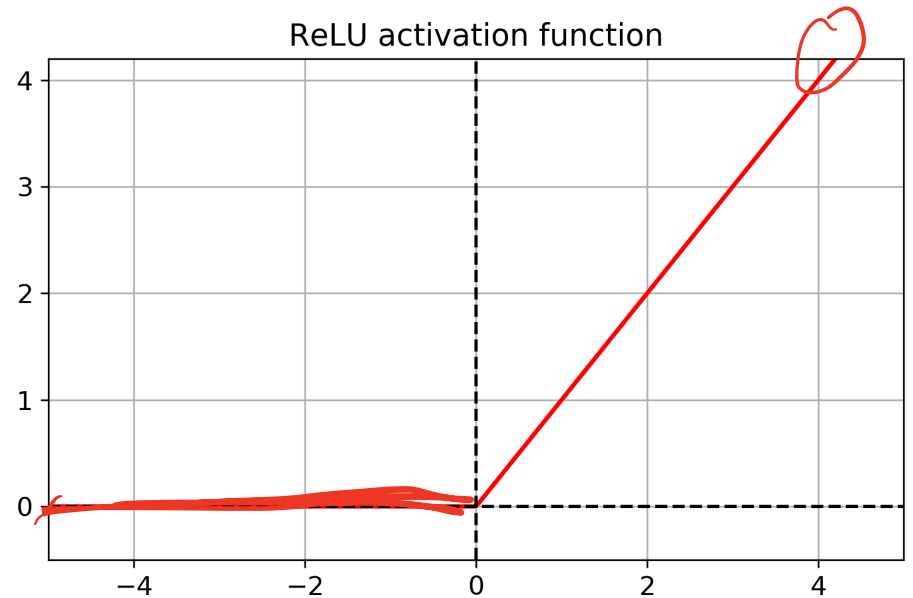
$$ReLU(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

**Range:**  $[0, \infty)$

**Pros:** No vanishing gradient (for  $x > 0$ ), computationally efficient, sparse activation

**Cons:** “Dying ReLU” problem (neurons stuck at 0), not zero-centered

**Use:** Default for hidden layers in most networks



# Deep Learning

## Activation Functions

- Leaky ReLU Function

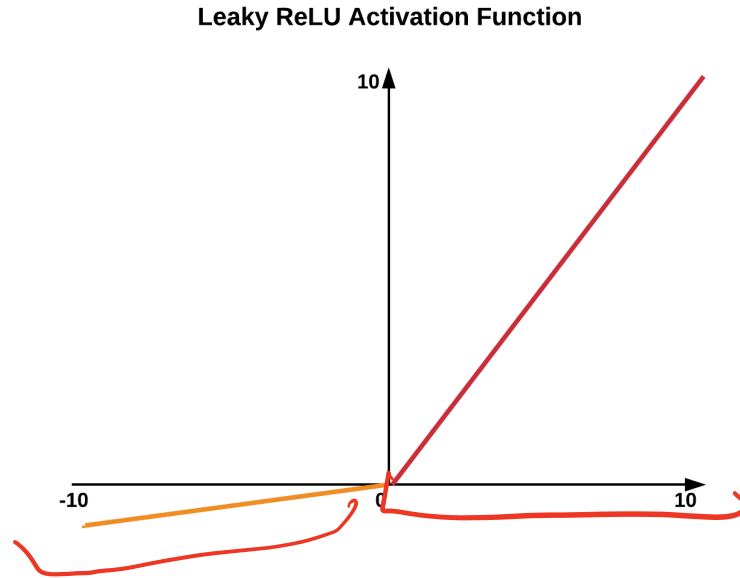
$$\text{ReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$

**Range:**  $(-\infty, \infty)$

**Pros:** Prevents dying ReLU

**Cons:** Extra hyperparameter ( $\alpha = \underline{0.01}$ )

**Use:** Default for hidden layers in most networks



# Deep Learning

## Activation Functions

If we have multi-class probability, what loss function can we use?

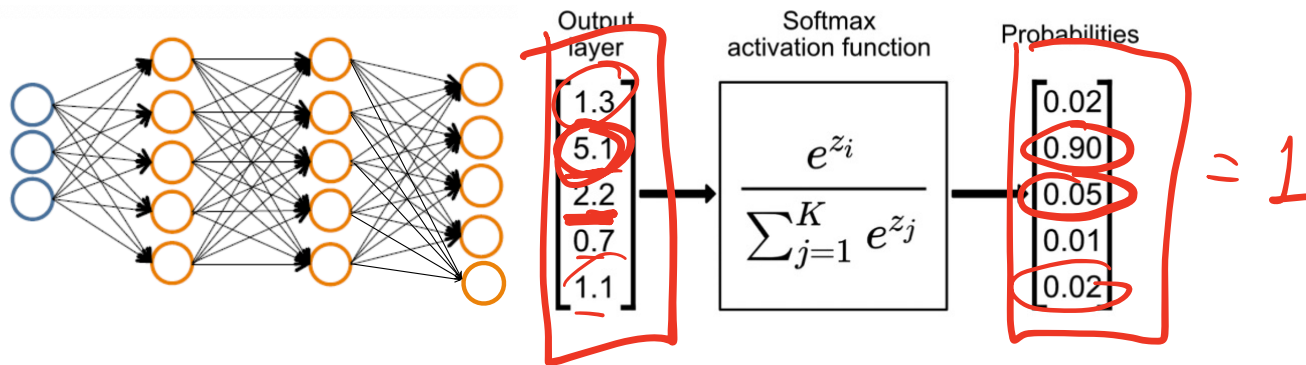
Categorical Cross Entropy Loss:

$$\ell_{\theta}(x) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log(\hat{y}_{ik})$$

- Softmax (**Output Layer**)

$$\text{softmax}(x)_k = \frac{e^{x_k}}{\sum_{j=1}^K e^{x_j}}$$

**Output:** Probability distribution over K classes (sums to 1)



# Deep Learning

## Activation Functions

If we have multi-class probability, what loss function can we use?

Categorical Cross Entropy Loss:

$$\ell_{\theta}(x) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log(\hat{y}_{ik})$$



Pedestrian



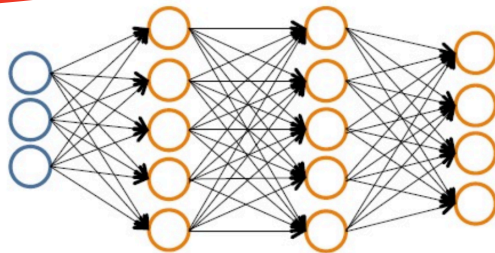
Car



Motorcycle



Truck



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

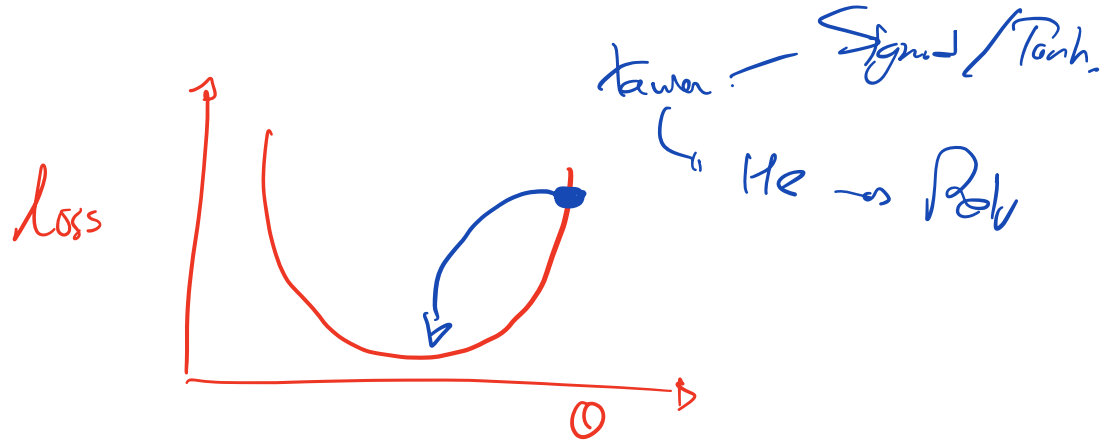
when motorcycle

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

# Deep Learning

## Weight Initialization



• Poor initialization causes:

• **Vanishing signals:** Activations shrink to zero through layers

• **Exploding signals:** Activations grow unboundedly

• **Symmetry:** If all weights are equal, all neurons compute the same thing

• **All Zeros**

• Problem: All neurons compute the same thing. Gradients are identical. Symmetry is never broken. Network cannot learn.

• **All Same Value**

• Same problem as zeros - symmetry is not broken.

# Deep Learning

## Regularization

- Deep networks have **millions** of parameters - they can **memorize** training data.
- We need regularization to improve generalization.

# Deep Learning

## Regularization

- $L_2$  Regularization
  - Add penalty on weight magnitude to loss

$$L_{\theta}(x) = L_{\theta}(x) + \lambda \sum_l \|W^{[l]}\|_F^2$$

- **Effect:** Pushes weights toward zero, preferring simpler models.

# Deep Learning

## Regularization

- $L_1$  Regularization
  - Add penalty on weight magnitude to loss

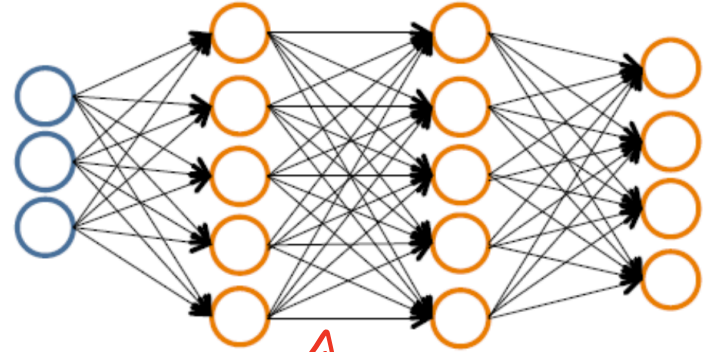
$$L_{\theta}(x) = L_{\theta}(x) + \lambda \sum_l \|W^{[l]}\|_1$$

- **Effect:** Encourages **sparsity** (many weights become exactly zero)

# Deep Learning

## Regularization

$$p = 0.2$$



- Dropout Regularization
  - During training: randomly set activations to zero with probability  $p$
  - During inference: Use all neurons, no dropout
  - **Intuition:**
    - Forces network to not rely on any single neuron
    - Approximately trains an ensemble of  $2^n$  sub-networks
    - Prevents co-adaptation of neurons

11-D dropout ( $p = 0.2$ )

# Deep Learning

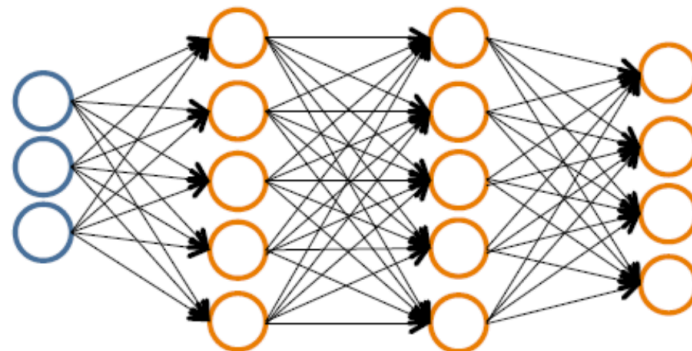
## Regularization

- Batch Normalization
  - Normalize **activations** within each mini-batch

$$\hat{z} = \frac{z - \mu_B}{\sqrt{\sigma_B^2}}$$

$$\tilde{z} = \gamma \cdot \hat{z} + \beta$$

- **Benefits:**
  - Reduces internal covariate shift
  - Allows higher learning rates
  - Acts as regularization (noise from batch statistics)
  - Reduces sensitivity to initialization



# Deep Learning

## Backpropagation

**Layer 1 (Hidden Layer):**

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \text{ReLU}(z^{[1]})$$

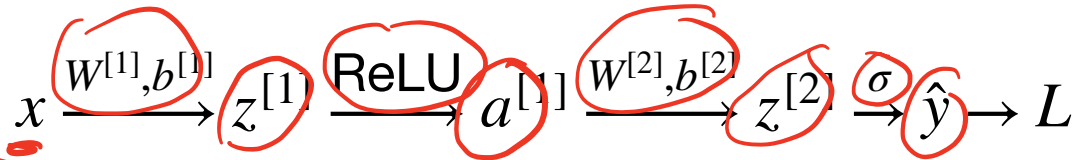
**Layer 2 (Output Layer):**

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$\hat{y} = \sigma(z^{[2]})$$

**Loss Function:**  $L = - [y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$

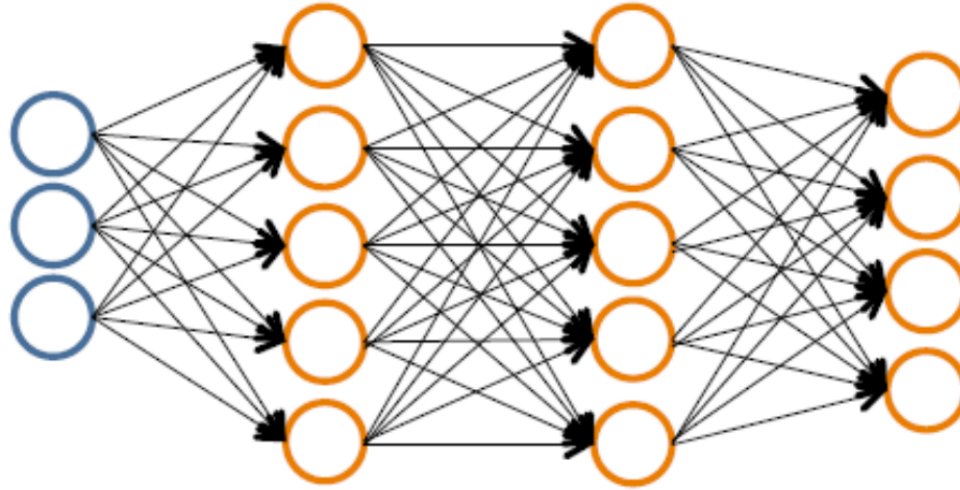
**Forward Pass:** Compute and store all intermediate values:



# Deep Learning

## Backpropagation

### 1. Forward Propagation



### 2. Backward Propagation



3. ?

Equations -

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$

$$(2 \times 4) \times (4 \times m)$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$(1 \times 2) \times (2 \times m)$$

$$a^{[2]} = \sigma(z^{[2]})$$

$$L = \frac{1}{2} (y - \hat{y})^2$$

Final Equations -

$$da^{[2]} = a^{[2]} - y$$

$$dz^{[2]} = da^{[2]} g'(z^{[2]})$$

$$d\omega^{[2]} = dz^{[2]} a^{[1]}$$

$$da^{[1]} = dz^{[2]} w^{[2]}$$

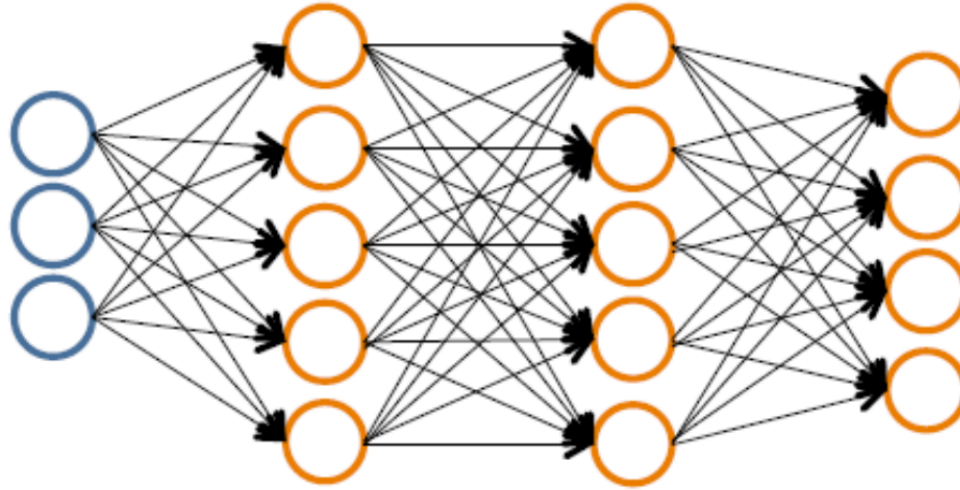
$$dz^{[1]} = da^{[1]} g'(z^{[1]})$$

$$d\omega^{[1]} = dz^{[1]} a^{[0]}$$

# Deep Learning

## Backpropagation

### 1. Forward Propagation



### 2. Backward Propagation



### 3. Update Weights - Gradient Descent

Equations -

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

$$\mathcal{L} = \frac{1}{2} (y - \hat{y})^2$$

Final Equations -

$$da^{[2]} = a^{[2]} - y$$

$$dz^{[2]} = da^{[2]} g'(z^{[2]})$$

$$da^{[1]} = dz^{[2]} w^{[2]}$$

$$dz^{[1]} = da^{[1]} g'(z^{[1]})$$

$$d\omega^{[1]} = dz^{[1]} a^{[0]}$$

# Convolutional Neural Networks

- An image is represented as a grid of pixel values
- Lets say this image is 224x224 RGB image
  - ~ 150 million parameters in the first layer alone
  - Why is this a problem?
    - **Too many parameters:** Overfitting, memory issues
    - **No spatial structure:** Flattening destroys 2D relationships
    - **No translation invariance:** A cat in the corner vs. center looks completely different to an MLP

# Convolutional Neural Networks

- CNNs exploit these insights through:
  - **Local receptive fields**: Each neuron sees only a small region
  - **Weight sharing**: Same detector (filter) applied across entire image
  - **Pooling**: Reduces spatial dimensions, adds invariance

# Convolutional Neural Networks

- 2D filters “slide” over the entire dataset
- Each filter learns something about the data

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ -- Vertical Edge Filter}$$

The diagram illustrates the convolution of a 6x6 input grid with a 3x3 vertical edge filter. The input grid is:

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix}$$

The filter is:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

The resulting output grid is:

$$\begin{bmatrix} 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \end{bmatrix}$$

Hand-drawn colored circles and lines connect the filter elements to the corresponding elements in the input and output grids. A blue line connects the filter to the first row of the input grid. A purple line connects the filter to the first row of the output grid. A red line connects the filter to the second row of the output grid. A green line connects the filter to the third row of the output grid.

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ — Blur (Averaging)}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \text{ — Horizontal Edge Filter}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \text{ — Sharpen}$$

# Convolutional Neural Networks

- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).
- Instead of having a fixed vertical edge filter like

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

- You learn filters like

$$\begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} \\ \theta_{13} & \theta_{14} & \theta_{15} \\ \theta_{16} & \theta_{17} & \theta_{18} \end{bmatrix}$$

# Convolutional Neural Networks

- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).
- Instead of having a fixed vertical edge filter like

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

- You learn filters like

$$\text{Filter 1: } \begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} \\ \theta_{13} & \theta_{14} & \theta_{15} \\ \theta_{16} & \theta_{17} & \theta_{18} \end{bmatrix} \quad \text{Filter 2: } \begin{bmatrix} \theta_{20} & \theta_{21} & \theta_{22} \\ \theta_{23} & \theta_{24} & \theta_{25} \\ \theta_{26} & \theta_{27} & \theta_{28} \end{bmatrix}$$

# Convolutional Neural Networks

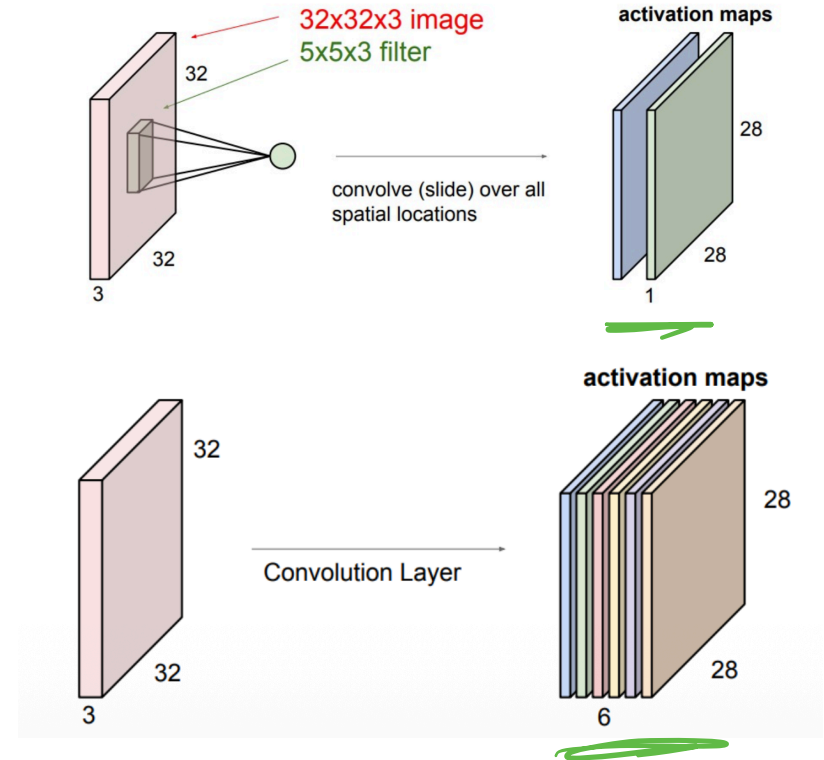
- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).
- Input:  $H_{in} \times W_{in} \times C_{in}$  (height x width x channels)
- Output:  $H_{out} \times W_{out} \times C_{out}$  (one channel per filter)
- Each filter has shape  $k \times k \times C_{in}$

# Convolutional Neural Networks

- Input:  $H_{in} \times W_{in} \times C_{in}$  (height x width x channels)
- Output:  $H_{out} \times W_{out} \times C_{out}$  (one channel per filter)
- Each filter has shape  $k \times k \times C_{in}$

## • Example

- Input Image:  $32 \times 32 \times 3$  (RGB Image), 6 filters
- Each filter:  $5 \times 5 \times 3 = 75$  parameters
- Output Shape:  $28 \times 28 \times 6$  (why 28?)
- Total parameters =  $75 \times 6 = 450$ , compared to  $32 \times 32 \times 3 \times 1000 = 3M$  parameters for MLP



# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 2

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 2

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 2

Next Stride?

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

0	10	10	10	0	0	0
0	10	10	10	0	0	0
0	10	10	10	0	0	0
0	10	10	10	0	0	0
0	10	10	10	0	0	0
0	10	10	10	0	0	0
0	10	10	10	0	0	0

Stride = 2

$$* \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Padding

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step
- Stride and padding change the shape of the output matrix
- With no padding, input size  $n$ , filter size  $k$  and stride  $s$  :

- Output Size =  $\lfloor \frac{n - k}{s} \rfloor + 1$

- With padding  $p$ :

- Output Size =  $\lfloor \frac{n + 2p - k}{s} \rfloor + 1$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

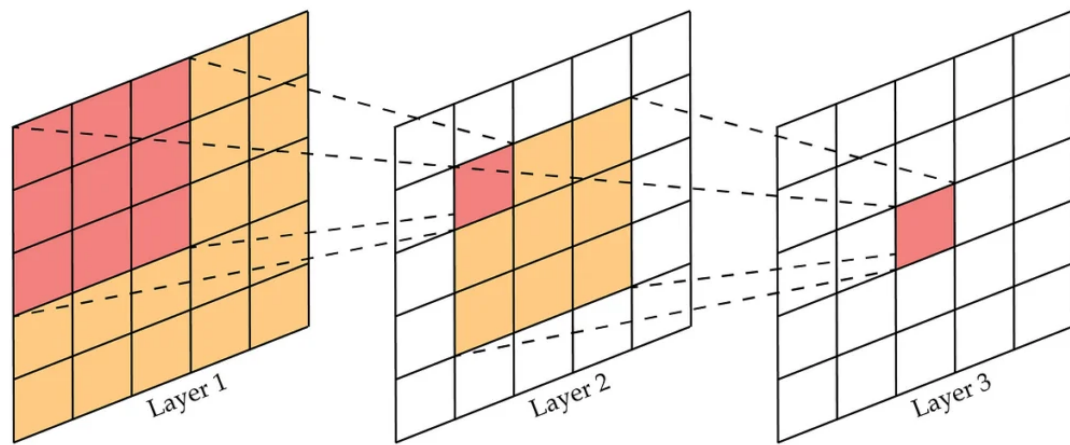
# Convolutional Neural Networks

## Pooling

- The receptive field is the **region of the input** that influences a particular output neuron.
- With stacking, receptive fields grow:
  - Layer 1 (3×3 filter): Each filter sees 3×3 input region
  - Layer 2 (3×3 filter): Each filter sees 5×5 input region
  - Layer 3 (3×3 filter): Each filter sees 7×7 input region

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

Receptive Field in Convolutional Networks



# Dimensionality Reduction

Metric	PCA	t-SNE	UMAP
Type	Linear	Non-Linear	Non-Linear
Preserves	Global Variance	Local Neighborhoods	Local + Global
Speed	Very Fast	Slow	Fast
Deterministic	Yes	No	Consistent but not deterministic
Interpretable	Somewhat	No	No
New Data	Yes	No	Yes
Use Cases	Preprocessing, quick visualization	Visualization	Visualization

# Principal Component Analysis (PCA)

## Motivation

- PCA finds new axes (principal components) such that:
  - PC1: Direction of maximum variance
  - PC2: Direction of maximum remaining variance, **orthogonal** to PC1
  - PC3: Direction of maximum remaining variance, **orthogonal to PC1 and PC2**

# Principal Component Analysis (PCA)

## Formulation

- **Goal:** Find a linear transformation that projects data onto directions of maximum variance.
- Given data matrix  $X \in \mathbb{R}^{n \times d}$  ( $n$  samples,  $d$  features)
  - Compute covariance matrix

$$C = \frac{1}{n-1} X^T X$$

- Compute Eigen-decomposition of  $C$

$$Cv_i = \lambda_i v_i$$

# Principal Component Analysis (PCA)

## Formulation

- Given data matrix  $X \in \mathbb{R}^{n \times d}$  ( $n$  samples,  $d$  features)
  - Compute covariance matrix

$$C = \frac{1}{n-1} X^T X$$

- Compute Eigen-decomposition of  $C$

$$Cv_i = \lambda_i v_i$$

- Sort Eigenvalues  $\lambda_1 \geq \lambda_2 \dots \geq \lambda_n$
- Project into generated dimensions

$$Z = XW_k \text{ where } W_k = [v_1, v_2, \dots, v_k]$$

# Principal Component Analysis (PCA)

## Features, Pros and Cons

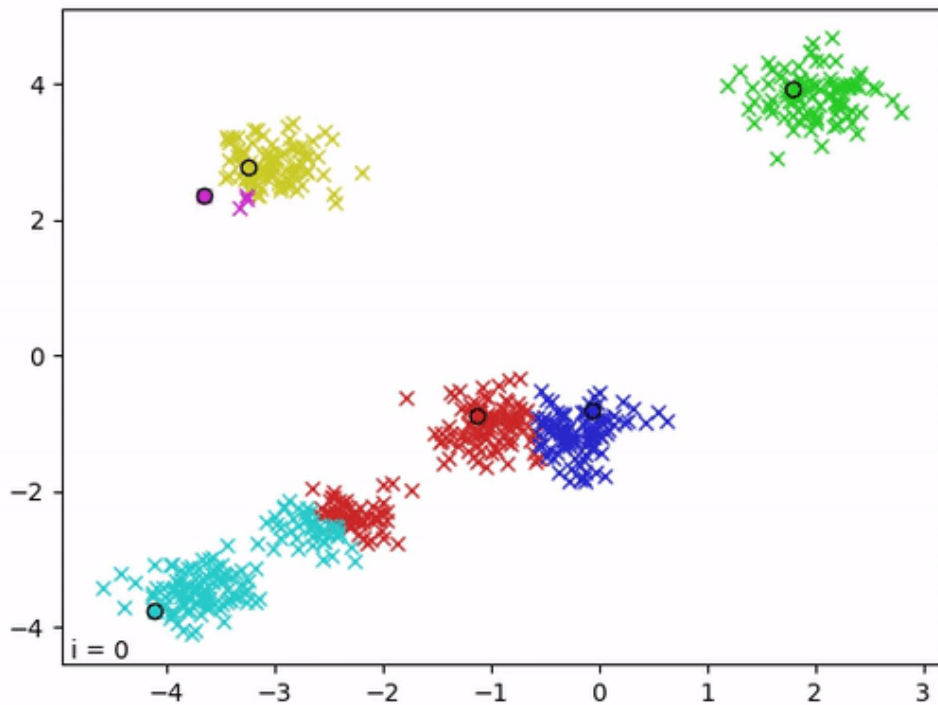
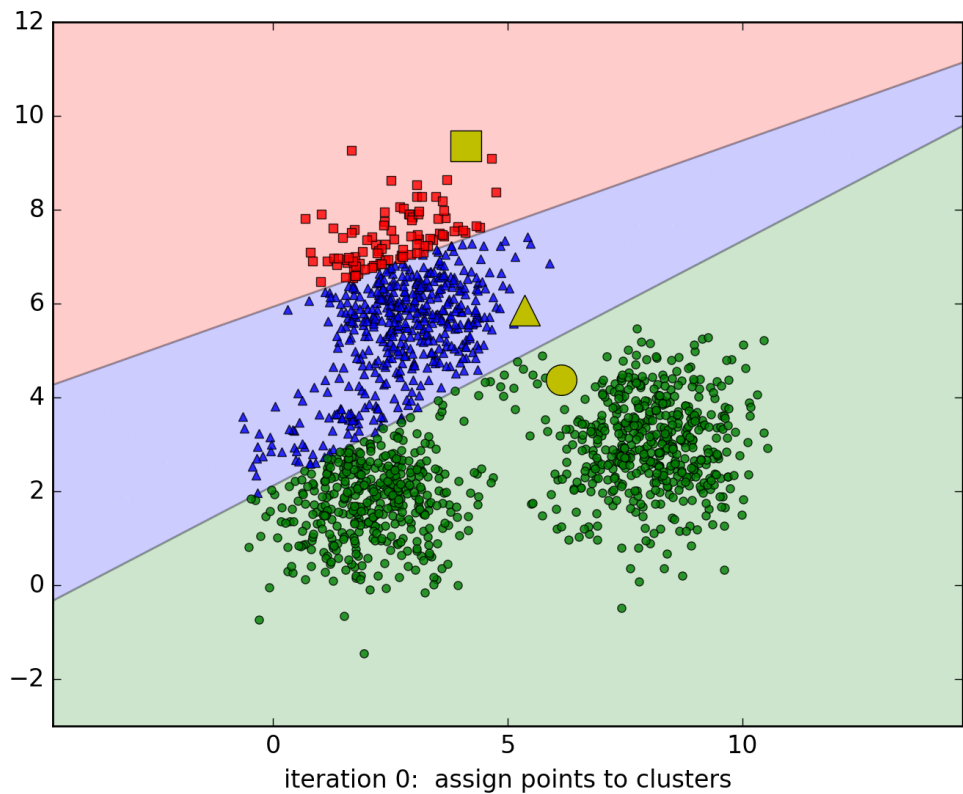
- Always make sure to normalize input features (say mean-variance normalization)
  - Without normalization: Features with larger variance dominate
  - With normalization: All features contribute equally
- Pros:
  - Multicollinearity Handling: Creates new, uncorrelated variables to address issues when **original features are highly correlated**.
  - Noise Reduction: Eliminates components with low variance
  - Data Compression: Represents data with fewer components reduce storage needs and speeding up processing.
  - Outlier Detection: Identifies unusual data points by showing which ones deviate significantly in the reduced space.
- Cons:
  - Assumes linear relationships, can't capture non-linear structure
  - Maximum variance direction may not be most informative - variance  $\neq$  importance
  - Outliers can dominate variance
  - Lack of interpretability - PCs are mixtures of original features

# Clustering

- **Goal:** Partition data into groups (clusters) such that:
  - Points within a cluster are similar
  - Points in different clusters are dissimilar
  - Unlike classification, **clusters are discovered**, not predefined.

# Clustering

## K-Means Clustering



# Clustering

## K-Mean Clustering

- **Input:** Data  $X$ , number of clusters  $k$
- Randomly initialize  $k$  cluster centroids  $\mu_1, \mu_2, \dots, \mu_k$
- Repeat until convergence:

- **ASSIGN:** For each point  $x_i$ , assign to nearest centroid  $\mu$

$$c_i = \arg \min_j \|x_i - \mu_j\|^2$$

- **UPDATE:** Recompute centroids as cluster means

$$\mu_j = \frac{1}{|C_j|} \sum_{i \in C_j} x_i$$

- **Return** cluster assignments and centroids

# Clustering

## K-Mean Clustering - When to use?

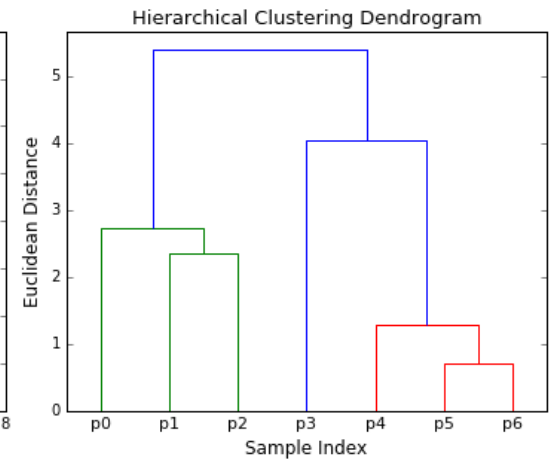
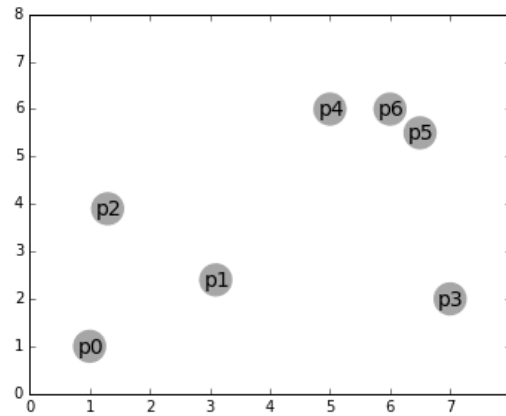
- Good for:
  - Large datasets (scales well:  $O(nkd)$  per iteration)
  - Roughly spherical, similar-sized clusters
  - When you know approximate number of clusters
- Not good for:
  - Non-convex cluster shapes
  - Clusters of very different sizes/densities
  - When number of clusters is unknown

# Clustering

## Hierarchical Clustering

- Agglomerative Algorithm

1. Start: Each point is its own cluster (n clusters)
2. Compute distance between all pairs of clusters
3. Merge the two closest clusters
4. Repeat steps 2-3 until only one cluster remains
5. Record the merge history (dendrogram)



# Clustering

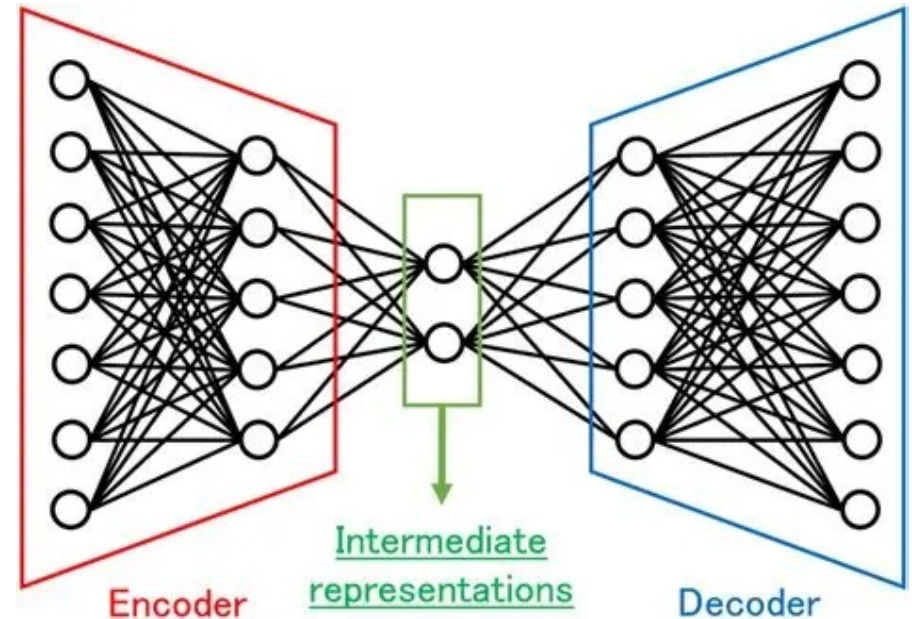
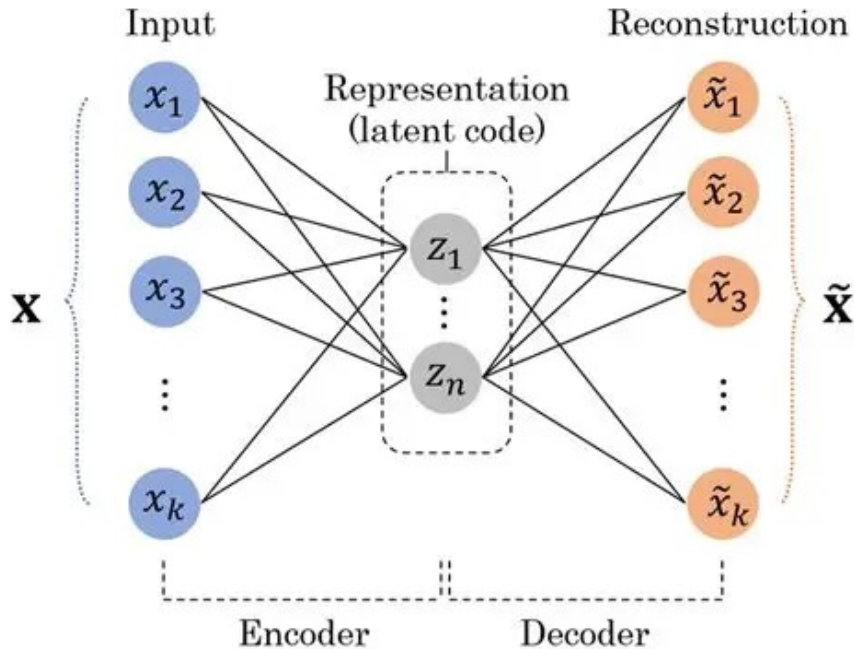
## Hierarchical Clustering - Pros and Cons

- Advantages:
  - No need to specify  $k$  in advance
  - Produces hierarchy (useful for taxonomy)
  - Dendrogram is informative
  - Any distance metric can be used
  - Deterministic (unlike K-Means)
- Disadvantages:
  - Slow:  $O(n^3)$  time,  $O(n^2)$  space
  - Cannot “undo” a merge (greedy)
  - Sensitive to noise and outliers
  - Not suitable for large datasets

# Autoencoders

## Dimensionality Reduction using Deep Learning

- An autoencoder is a neural network trained to **reconstruct** its input.



# Autoencoders

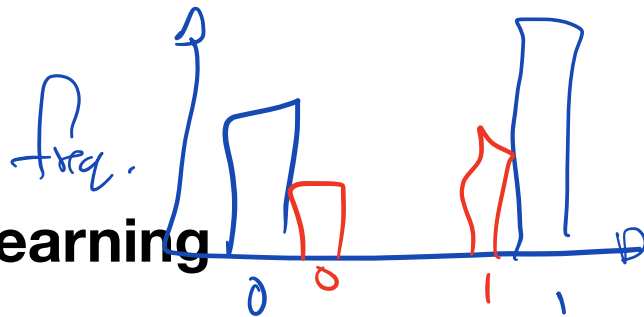
## Dimensionality Reduction using Deep Learning

- An autoencoder is a neural network trained to **reconstruct** its input.
- **Encoder:** Compresses input to lower-dimensional latent representation
- **Latent space:** Compressed representation (bottleneck)
- **Decoder:** Reconstructs input from latent code
- **Training objective:** Minimize reconstruction error

$$L = \|x - \hat{x}\|^2$$

# Variational Autoencoders

## Dimensionality Reduction using Deep Learning



- Standard autoencoders learn a **deterministic** mapping to latent space (this is true for all neural networks in general)

- The latent space may have “holes”, i.e., regions that don't correspond to valid data.

- You can't sample from it to generate new data.

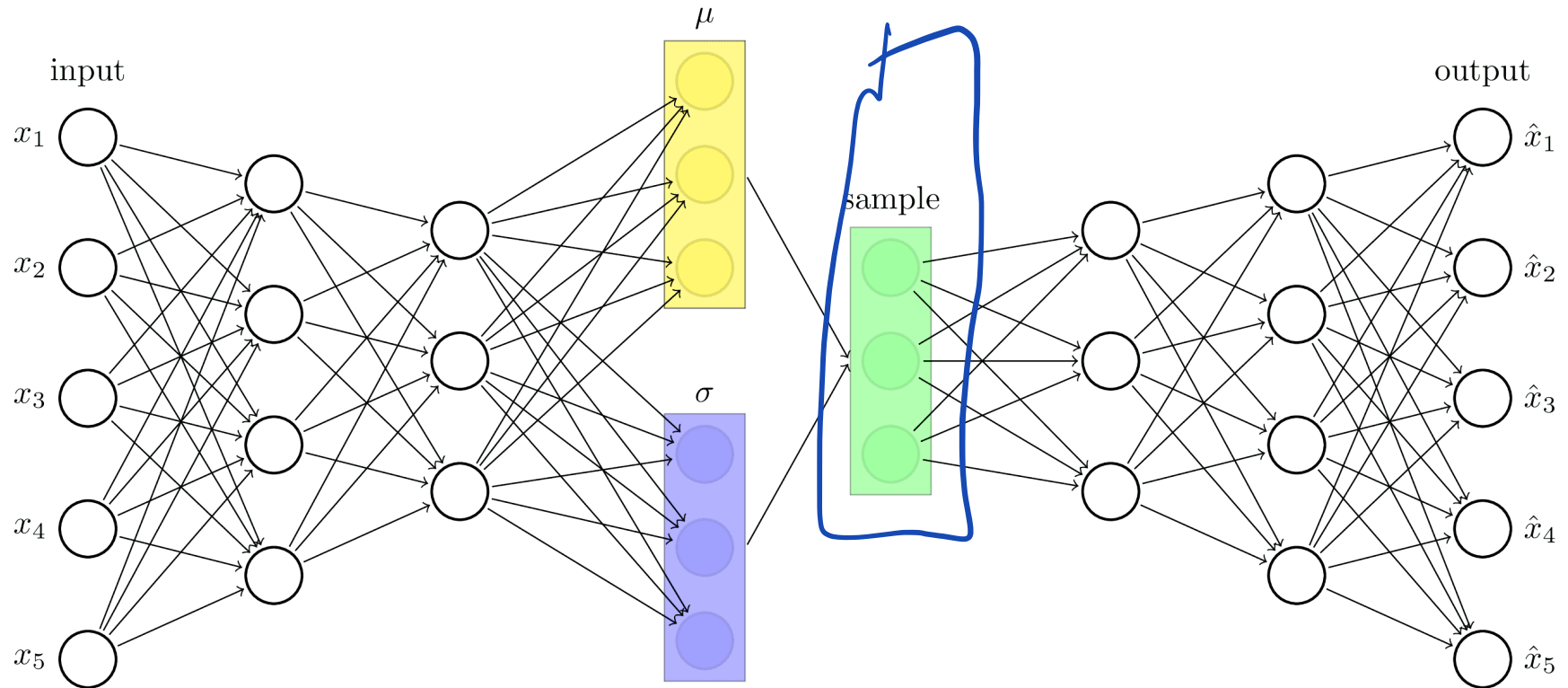
- **VAE Key Idea**

- Instead of encoding to a point, encode to a **probability distribution**. The encoder outputs **parameters of a Gaussian distribution**, and we sample from it



# Variational Autoencoders

## Dimensionality Reduction using Deep Learning



# Variational Autoencoders

## Dimensionality Reduction using Deep Learning

- The Reparameterization Trick

$$\begin{aligned}\mu &= w \cdot a^{[l]} + b^{[l]} \\ \sigma &= w \cdot a^{[l]} + b^{[l]}\end{aligned}$$

Sampled

$$z \sim \mathcal{N}(\mu, \sigma)$$

- **Problem:** Can't backpropagate through random sampling

- **Solution:** Reparameterize the sampling

$$z = \mu + \sigma(\epsilon)$$

- $z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0,1)$

- The randomness from sampling is now in  $\epsilon$  and gradients can flow through  $\mu$  and  $\sigma$

# Variational Autoencoders

## Dimensionality Reduction using Deep Learning

- VAE Loss Function

- Two Components

- Reconstruction Loss - How well can the model reconstruct the input?

$$L_{recon} = \|x - \hat{x}\|^2$$

- KL Divergence - How close is the learned distribution to the original?

$$L_{KL} = KL(q(z|x) || p(z))$$

- Total Loss:  $L = L_{recon} + L_{KL}$

The KL divergence **regularizes** the latent space:

- Encourages latent distributions to be close to  $\mathcal{N}(0,1)$
- Creates a smooth, continuous latent space
- Enables **generation**: Sample  $z \sim \mathcal{N}(0,1)$ , decode to generate new data
- Without KL term, the encoder could **learn very narrow distributions** (essentially points), losing the generative property.

5	7	8
3	7	9
1	7	2

$$\begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$

