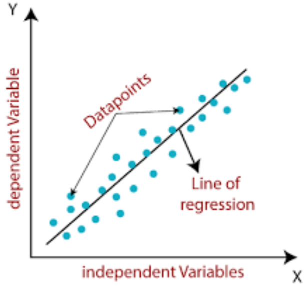
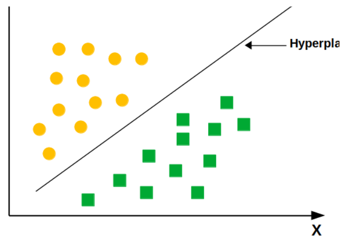
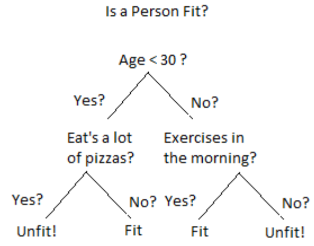
Machine Learning to Deep Learning

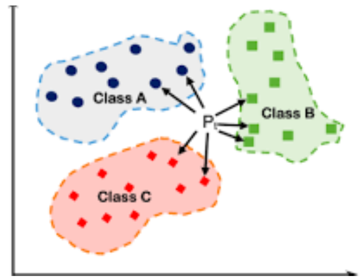
AI overview

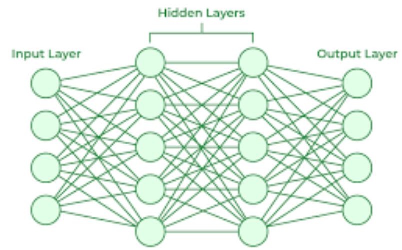






- Artificial Intelligence
  - Old school (1950~80)
    - Rule-based AI (Expert systems) – *programmed intelligence*
    - Search & Planning (A*/minimax) - *optimization*
  - Machine Learning - *learn from data*
    - Traditional ML – *manual feature selection*
      - Linear/Logistic Regressions
      - Support Vector Machine (SVM)
      - Decision Trees/Random Forrests
      - k-Nearest Neighbors (KNN)
      - (Shallow) Neural Network
    - Deep Learning - *automatically learn features* (2012~)
      - Feed-forward Neural Network (ANN)
      - Convolutional Neural Network (CNN)
      - Recurrent Neural Network (RNN)
      - Transformers





- *Training paradigms*
  - *Supervised learning*
  - *Unsupervised learning*
  - *Reinforcement learning*

Training Paradigms

- Supervised learning
  - learns from labeled examples (label=correct answer)
- Unsupervised learning
  - finds patterns in data without labels
- Reinforcement learning
  - learns to make decisions by interacting with an environment to maximize cumulative rewards over time.

Supervised Learning Pipeline

1. Data Collection & Preparation
2. Data Splitting
   - Training set/ Validation (tuning) set/ Test set
3. Feature Engineering
   - Feature selection & transformation
4. Model Selection
5. Define Loss function
   - Cross-entropy/ Hinge loss/ MSE/ MAE
6. Model training
7. Tuning
   - Learning rate, # of layers, Tree depth, # of neighbors
8. Model Evaluation & Validation & Error analysis
9. Deploy,ent

Naïve Bayes Spam Classifier

- Classify an email as spam or ham (legitimate mail). -- Binary classification.
- Learn statistical patterns from labeled examples, then use those patterns to label new, unseen examples.
- Math

$$P(\text{Spam} \mid \text{Words}) \propto P(\text{Spam}) \cdot P(\text{Words} \mid \text{Spam})$$

ID	Email Content	Label
1	"Money free money"	**Spam**
2	"Click free link"	**Spam**
3	"Meeting today money"	**Ham (Normal)**
4	"Lunch today"	**Ham (Normal)**

Attack #1: Poisoning a Spam Filter

Scenario:

Uses a Naive Bayes classifier to filter spam emails. The model learns that certain words indicate spam ("FREE", "WINNER", "CLICK HERE").

An attacker start adding random dictionary words to their spam emails: "the cat sat on the mat FREE MONEY CLICK HERE..."

What can go wrong:

- The filter retrains on user feedback (legitimate emails marked as spam by mistake)
- Good words like "cat", "mat" now associated with spam
- Future legitimate emails with these words get filtered!

Why This Works:

- Naive Bayes: P(spam|words) ∝ P(words|spam) × P(spam) — learns word-spam associations
- Attacker adds benign words → model learns benign words correlate with spam
- Real-world example: Microsoft's 2007 spam filter was attacked this way

Attack #2: Fooling a Self-Driving Car

Scenario:

An autonomous vehicle uses a CNN to recognize traffic signs. It's trained on thousands of stop signs and achieves 99.9% accuracy.

Attacker places small stickers on a stop sign in a specific pattern. To humans, it still clearly looks like a stop sign.

What can go wrong:

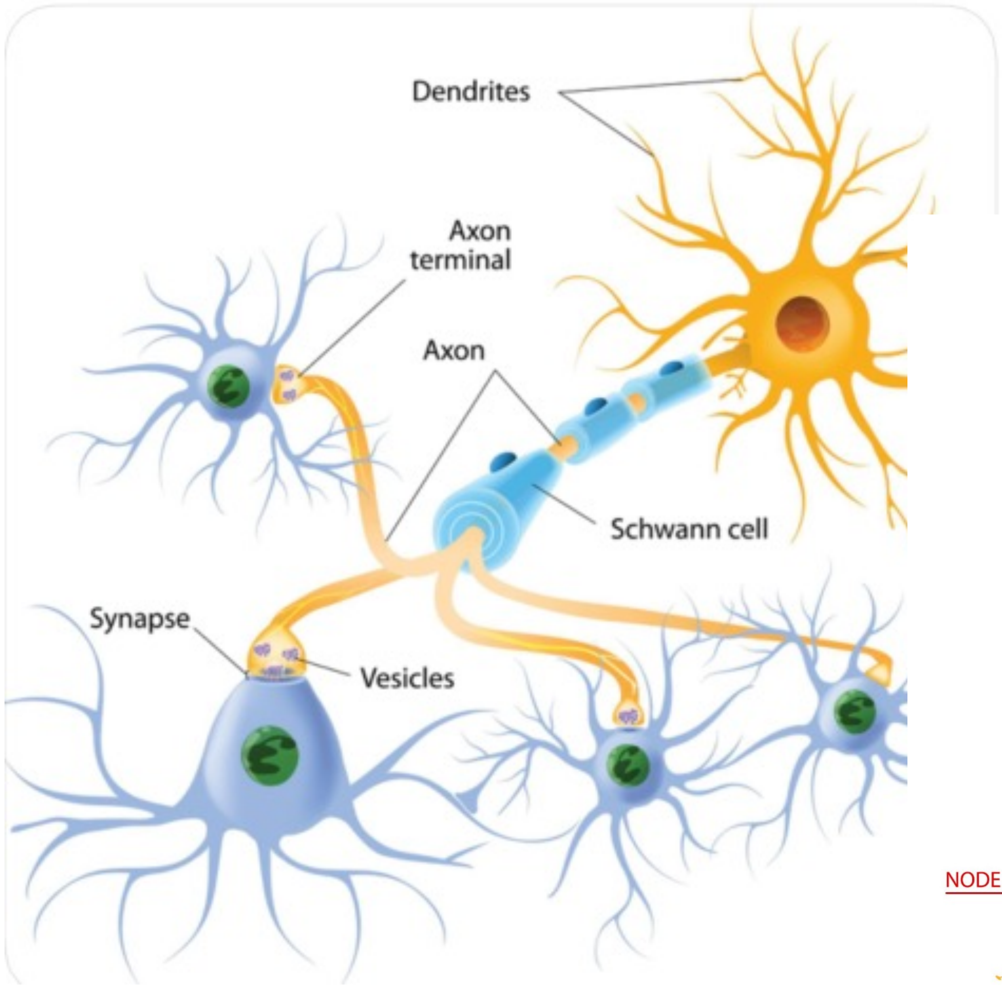
- The CNN classifies it as "Speed Limit 45" with 95% confidence
- The car doesn't stop at the intersection

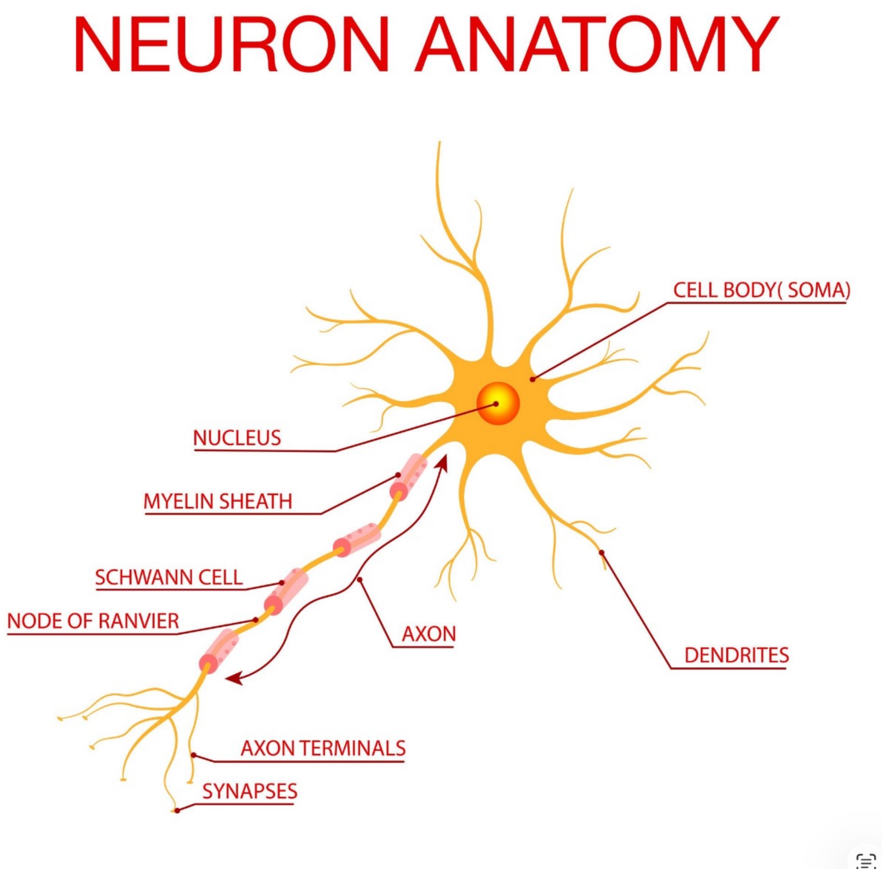
Why This Works:

- CNNs learn pixel patterns, not semantic understanding — stickers change patterns in ways humans don't notice
- Decision boundaries are close to data — small perturbations can cross boundaries
- https://openaccess.thecvf.com/content_cvpr_2018/papers/Eykholt_Robust_Physical-World_Attacks_CVPR_2018_paper.pdf?utm_source=chatgpt.com

How human brain works

- **86 billion neurons**
- each neuron connected to **thousands of others**





Why Neural Network

The Problem

- Many ML algorithms are linear (Regression, SVM)
- Linear models can't capture complex patterns
- Manual feature selection is time-consuming
- Real-world data is high-dimensional non-linear
- We need models that learn representations automatically

The solution: Neural Network

- Compose simple non-linear functions
- Learn features from data automatically
- Universal function approximators
  - single-layer NN can approximate any cont's f(x)
- Scale to massive datasets and complexity

$$y = w^T x \longrightarrow y = f_L(f_{L-1}(\dots f_1(x)))$$

Simple Example: XOR Problem

Input: (0,0) → Output: 0 | Input: (0,1) → Output: 1

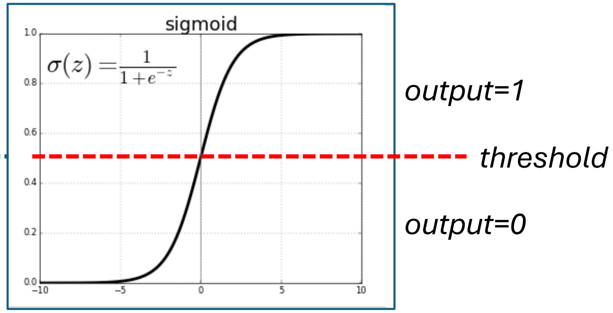
Input: (1,0) → Output: 1 | Input: (1,1) → Output: 0

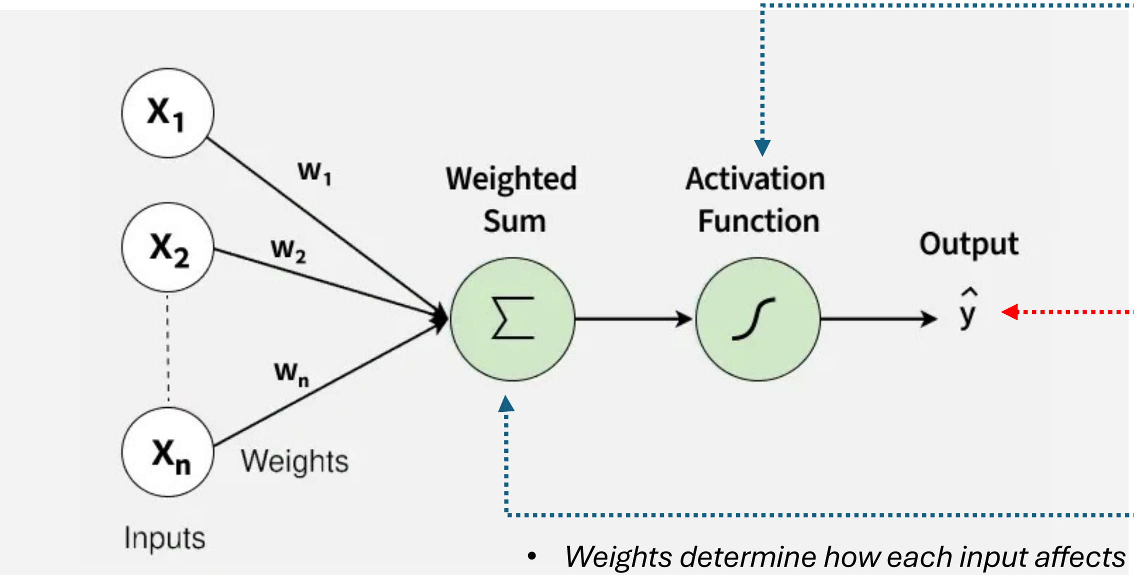
❌ Linear classifier (like logistic regression): Cannot solve this! No single line separates the classes.

✅ Neural network with 1 hidden layer: Easily solves it by learning non-linear decision boundary.

The Perceptron (Single Neuron)

- The simplest form of a neural network
- Can be used for binary classification
- Basic building block of neural network
- Takes multiple inputs and assigns weights
- Computes a weighted sum and applies a threshold (activation function)
- Outputs either 0 or 1 (binary outcome)
- Can solve only linear problems (so is decision boundary)





$$y = f(w^T x + b)$$

weights   bias

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- *Weights determine how each input affects*
- *Bias shift the decision boundary off the origin*

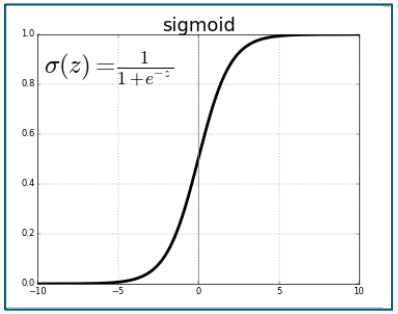
Activation Functions: Adding Non-linearity

- Without non-linear activations, stacking layers just gives you a linear function!
- Activation functions introduce the non-linearity making neural networks powerful.

Sigmoid

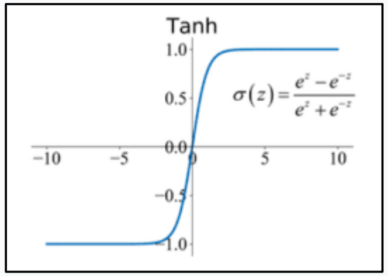
- Output: (0, 1)
- Smooth, probabilistic
- Vanishing gradients

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



- **Tanh**
- Output: (-1, 1)
- Zero-centered
- Vanishing gradients

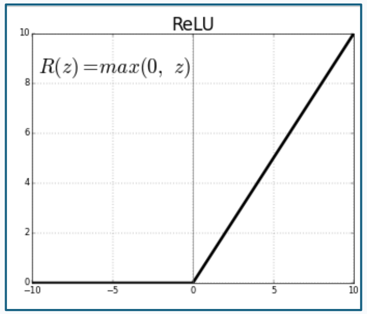
$$\text{Tanh}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



ReLU

- Output: [0, ∞)
- Fast
- no vanishing gradient
- "Dying ReLU" problem

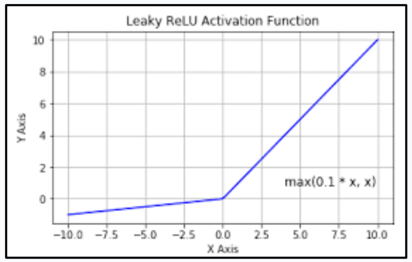
$$\max(0, x)$$



Leaky ReLU

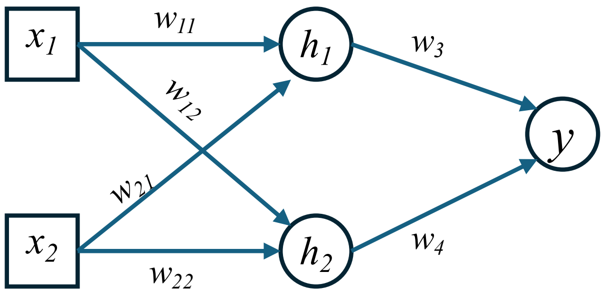
- Output: (-∞, ∞)
- Fixes dying ReLU
- Extra hyperparameter

$$\max(0.1x, x)$$



Single perceptron cannot solve XOR

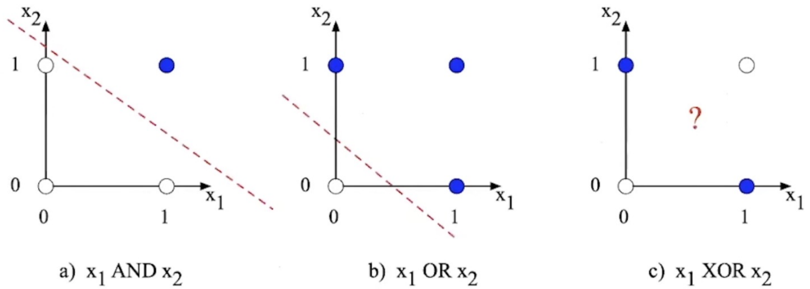
- XOR decision cannot be expressed by a linear boundary
- Mathematical contradiction
  - *f(0,0)=0: b < 0*
  - *f(0,1)=1: w2+b > 0*
  - *f(1,0)=1: w1+b > 0*
  - *f(1,1)=0: w1+w2+b < 0*
  - *w1+w2 > -2b*
  - *w1+w2 < -b*
  - *-2b < -b*
  - *b > 0 : contradiction*
- Solution
  - Add one hidden layer

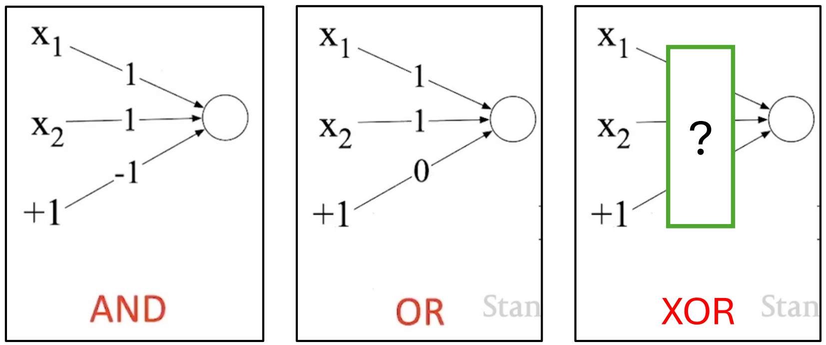


$$y = \text{sign}(w_1 x_1 + w_2 x_2 + b)$$

sign(z) = 1 if z>0, 0 o.w.

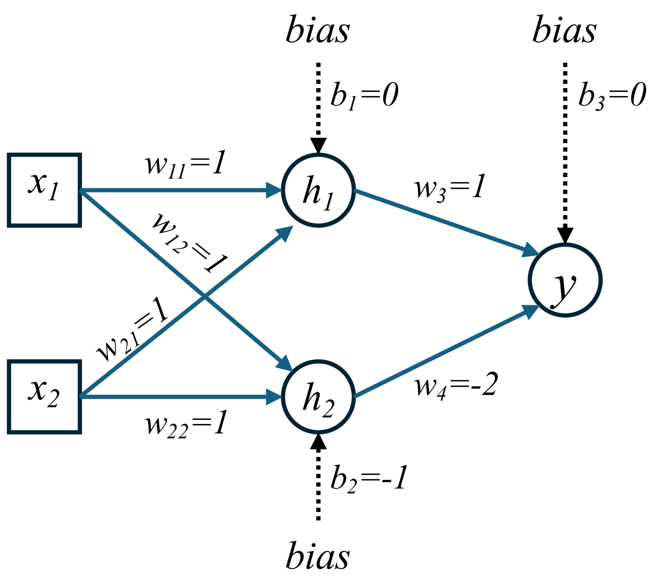
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0





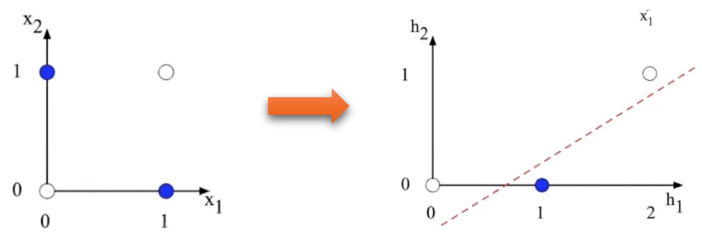
Add a hidden layer to solve XOR

- 2-2-1 Neural Network with ReLU activation function



$$h_1 = ReLU(w_{11}x_1+w_{21}x_2+b_1)$$
$$= ReLU(x_1+x_2)$$
$$h_2 = ReLU(w_{12}x_1+w_{22}x_2+b_2)$$
$$= ReLU(x_1+x_2-1)$$
$$y = ReLU(w_3h_1+w_4h_2+b_3)$$
$$= ReLU(h_1-2h_2)$$
$$= ReLU(ReLU(x_1+x_2)-2ReLU(x_1+x_2-1))$$

- Calculations
  - 0 xor 0: y = ReLU(ReLU(0) -2*ReLU(-1)) = 0
  - 0 xor 1: y = ReLU(ReLU(1) -2*ReLU(0)) = 1
  - 1 xor 0: y = ReLU(ReLU(1) -2*ReLU(0)) = 1
  - 1 xor 1: y = ReLU(ReLU(2) -2*ReLU(1)) = 0
- Hidden layer
  - (x1, x2) → (h1, h2): new representation of data



input space (Non-linear seperable)

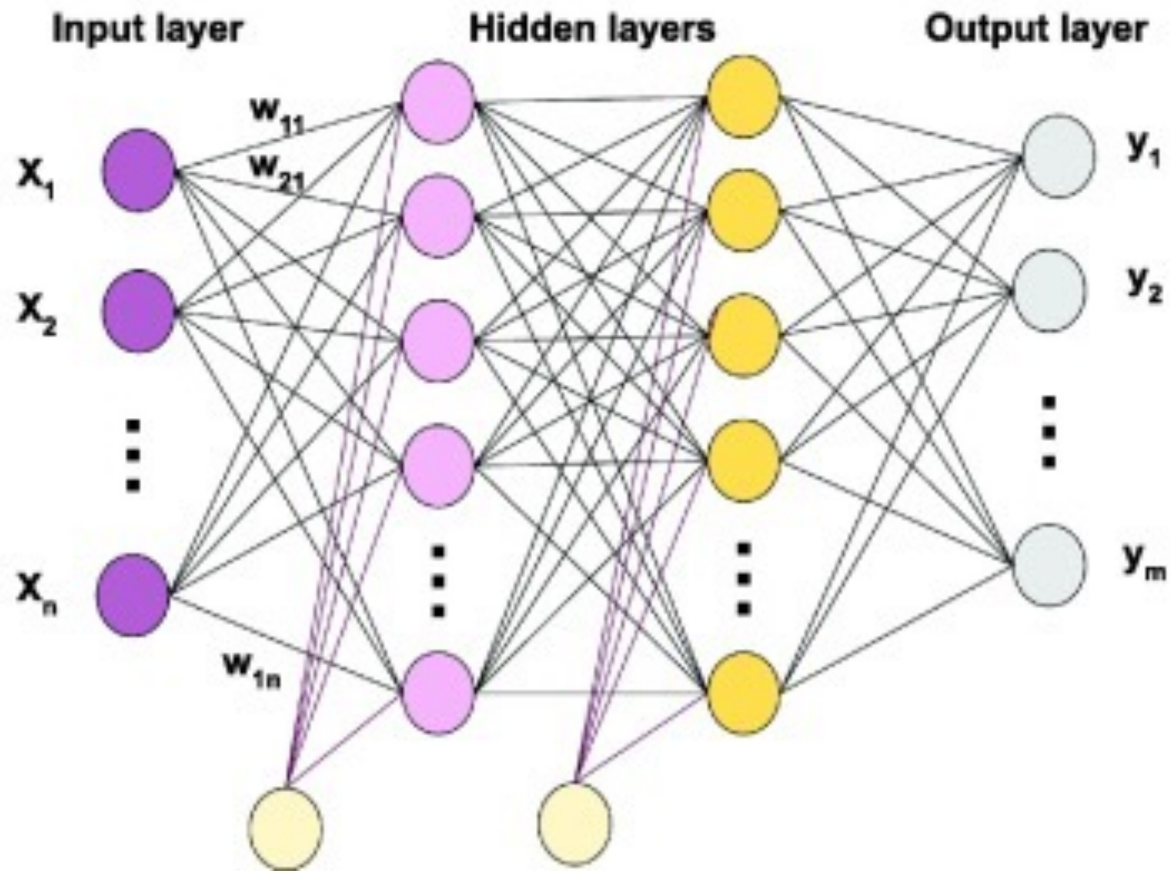
hidden space (linearly-seperable)

From Single-layer to Multi-layer NN

- What single-layer NN can do (no hidden layer)
  - $y=f(Wx+b)$ with linear/non-linear activation function f
  - Decision boundary is linear (hyperplane)
    - Only linear separation problem can be solved
    - Limisted representational power
- What additional Hidden-layers can do?
  - Hidden layer transforms input space to new representation space by $h = f(W_1 x + b_1)$
  - Output $y = f(W_2 h + b_2)$
  - Thus, decision boundary in input space becomes
    - $W_2 f(W_1 x + b_1) + b_2$ -- *non-linear in x*
  - So, hidden-layer transforms input space by bending/forlding/stretching so that data, not linear-separable, become linear separable
    - eg: speech, vision, language
  - Learns features
  - Build complex functions
  - Expressive efficiency
    - Theoretically only one hidden-layer is needed (Universal Approximation Theorem)
    - But some functions require exponentially many neurons in shallow network
    - Alternatively, only polynomial number of hidden layers are needed

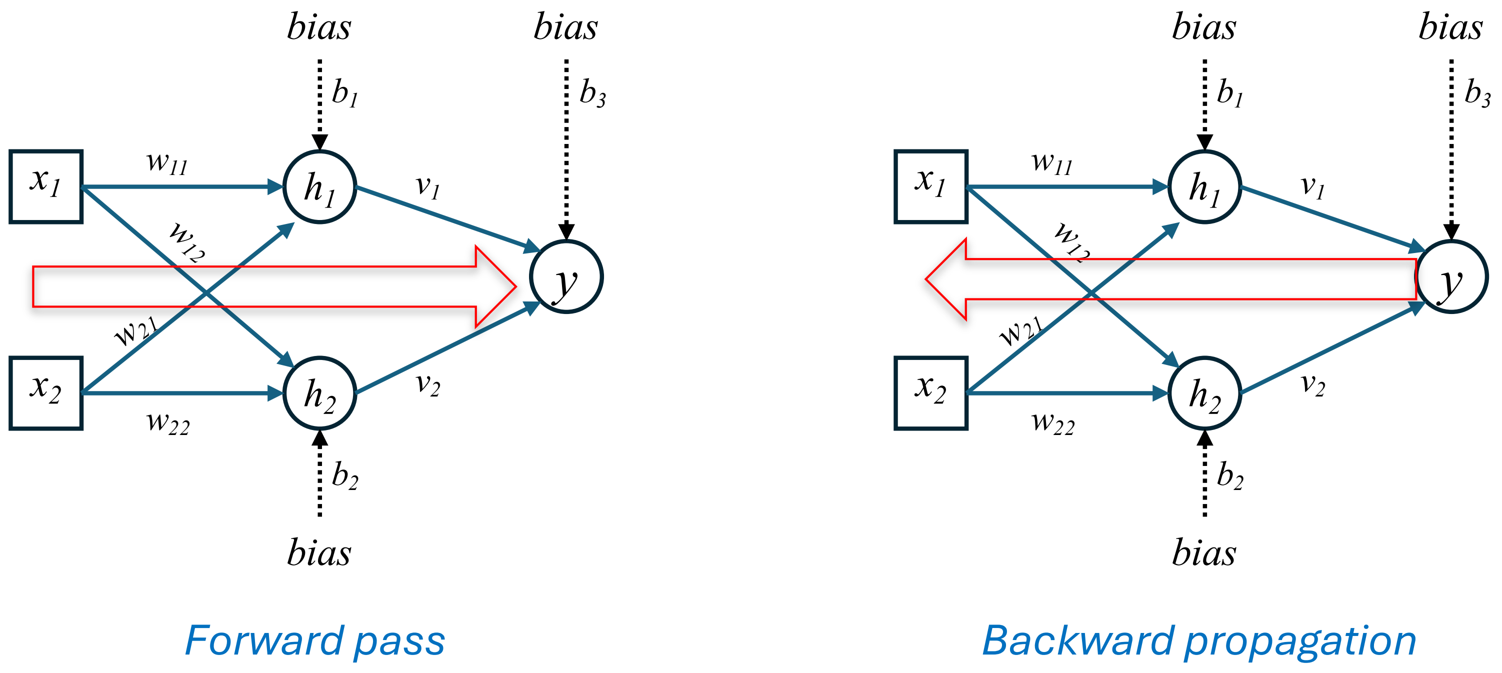
Multi-layer Neural Network

- Multi-Layer Perceptron (MLP)
- Input layer
- Hidden layers
- Output layer



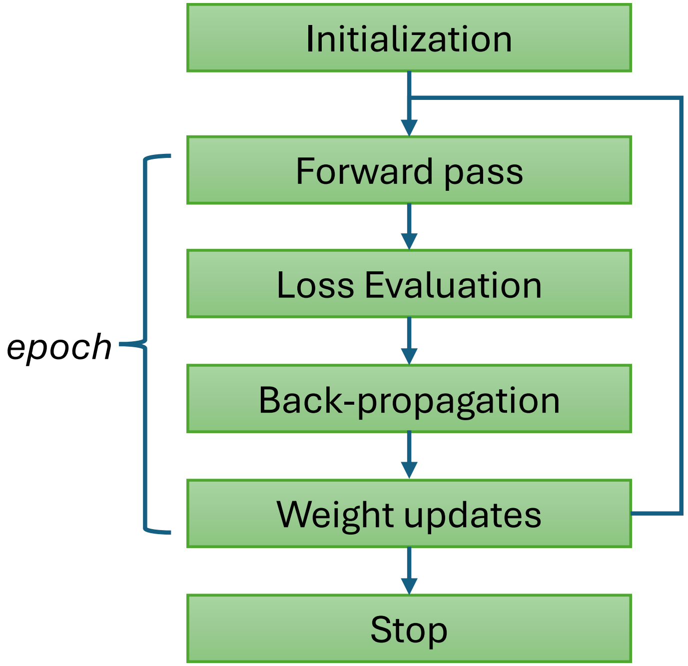
How can we find the weights/biases?

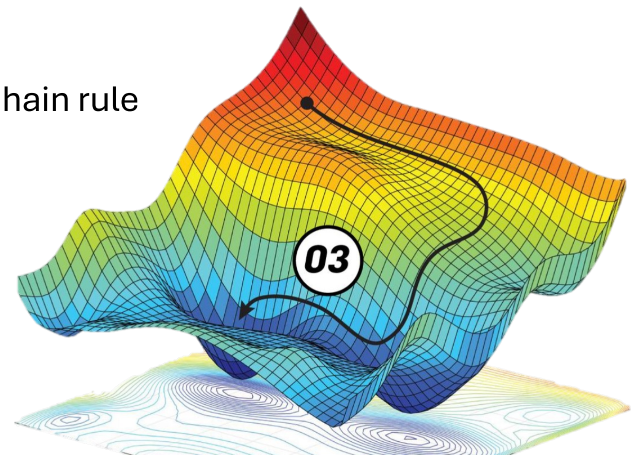
- 2-2-1 Neural Network for XOR
  - Training = Model building = find the best weights & biases
  - Model = Network + weights + biases
  - Predict (Forward) → Evaluate → Update (Backward) → Repeat



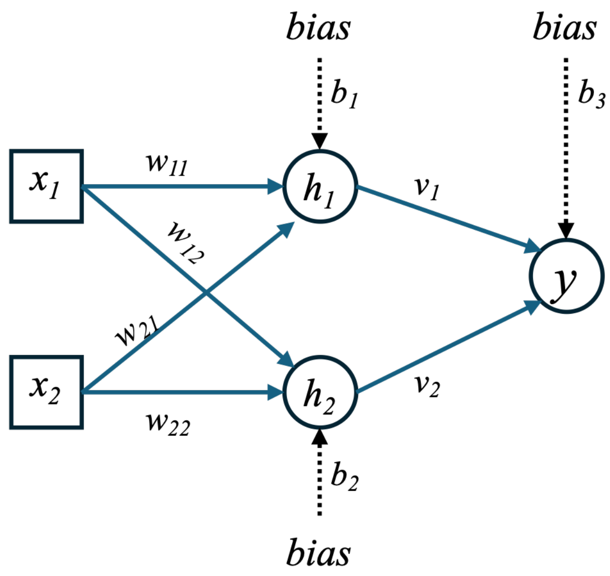
How a Neural Network Learns

- *"Optimization over a loss surface via gradient descent"*
- Training = Optimization: adjust weights to minimize the loss
- Design step
  - Architecture: define functions to learn
  - Loss function: define goals, differentiable by weights
  - Optimizer: How to descend the loss surface
- Initialization step
  - Set weights to small random values
- Forward pass
  - Flow from input→hidden→output (prediction)
- Loss evaluation
  - Binary classification: Binary Cross-Entropy
  - Multi-class: Categorical Cross Entropy
  - Regression: Mean Square Error
- Back-propagation (Gradient descending)
  - Compute change rate of loss by weight (partial deriviation) using chain rule
- Weight update (optimizer)
  - Vanilla Gradient Descent
  - SGD + Momentum – accumulates velocity, dampens oscillation
  - Adam – Adaptive per-weight learning rates
- If loss is not improving → stop





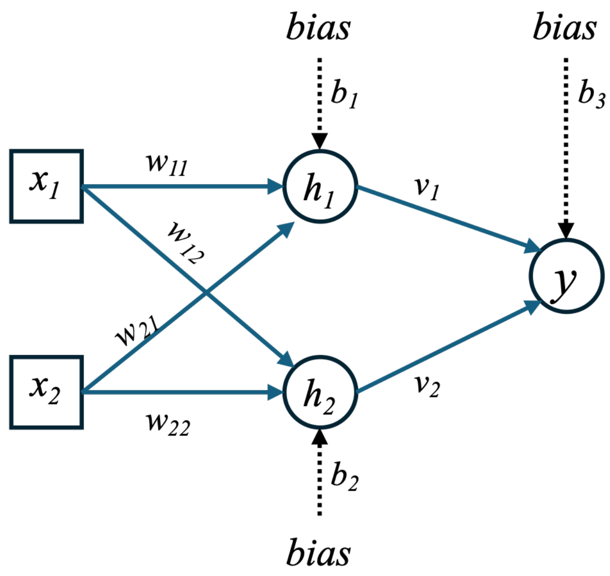
XOR: Design

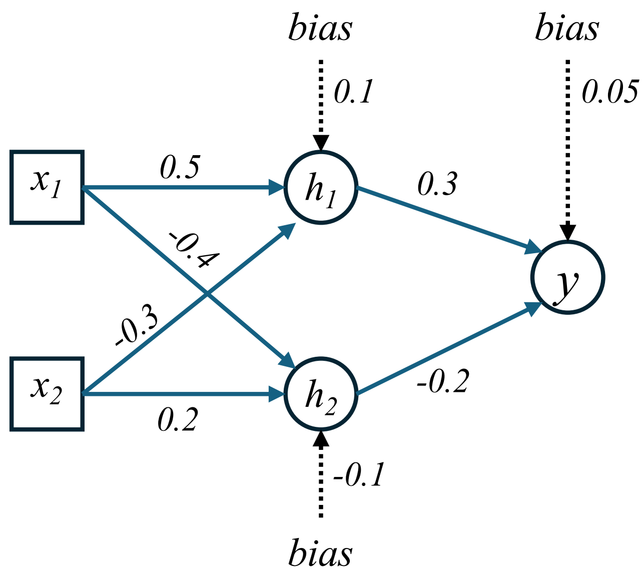


- Architecture
  - 2 inputs
  - 2 hidden neurons
  - 1 output neuron
  - (2-2-1) neural network
  - Activation function: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Transforms:
  - Params: $\theta = \{w_{11}, w_{12}, w_{21}, w_{22}, b_1, b_2, v_1, v_2, b_3\}$
- Loss function
  - Use binary cross entropy: $L = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$
- Optimizer
  - Gradient descent $\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$

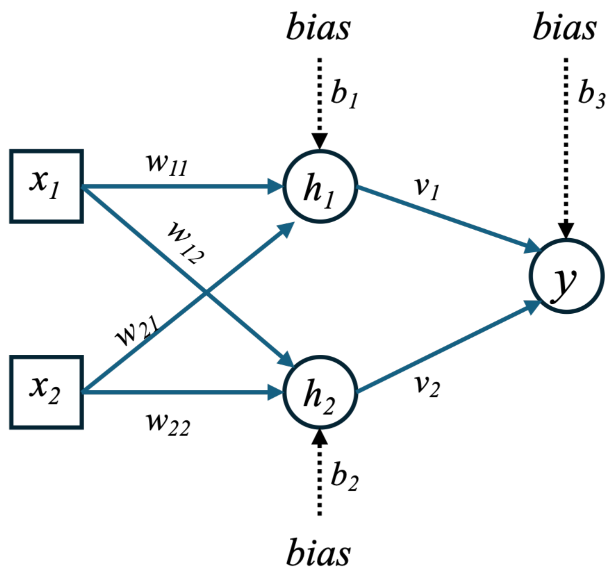
XOR: Initialization

- w11 = 0.5   w12 = -0.4
- w21 = -0.3 w22 = 0.2
- b1 = 0.1   b2 = -0.1
- v1 = 0.3   v2 = -0.2
- b3 = 0.05
- Learning rate: 0.1
- Training data
    - (x1, x2) = (1, 0) → y=1



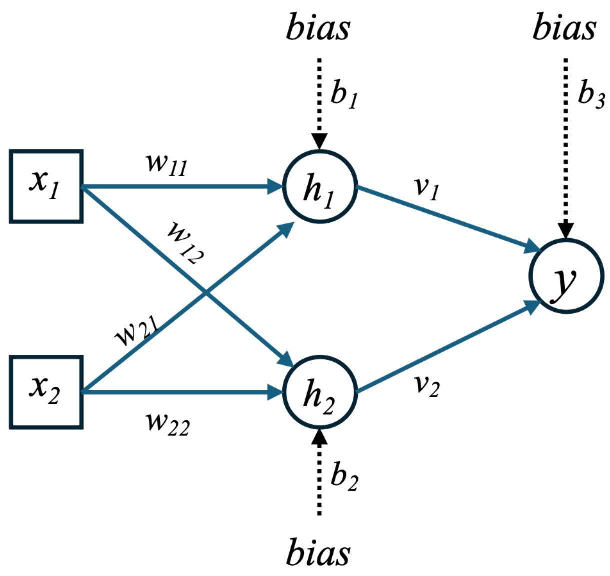


XOR: Forward Pass



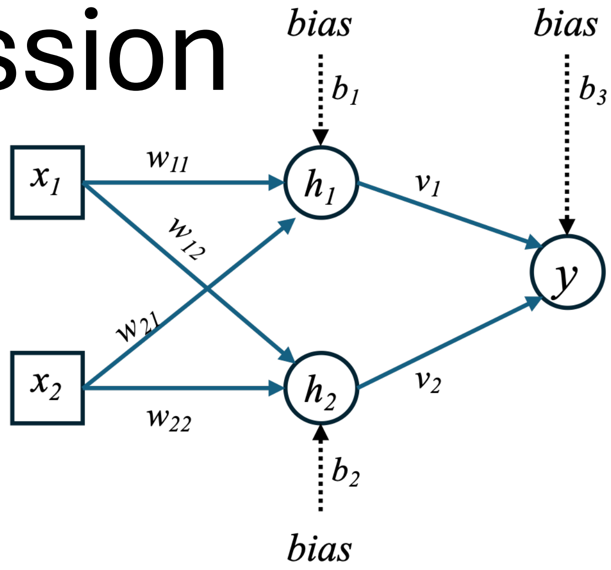
- (x1, x2) = (1, 0) → y_{target}=1
- z1=
- h1=
- z2=
- h2=
- z3=
- y=
- y_{target}=1

XOR: Back Propagation and weight update

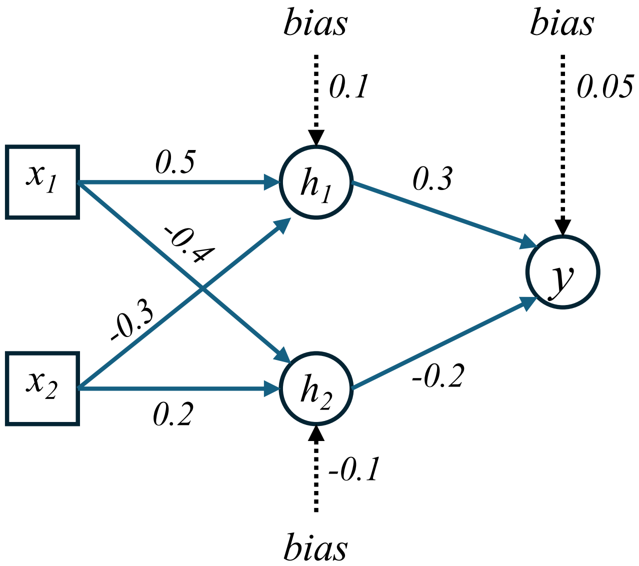


- v1’ = v1 – 0.1 * dL/dv1
- v2’ = v2 – 0.1 * dL/dv2
- w11→z1→h1→z3→y→L
- …
- dL/dw11 = (y^-y)v1h1(1-h1)x1
- w11’ = w11-0.1*dL/dw11
- …

Another example: Regression



- Learn how to multiply
  - Compute x1 * x2
  - Use 2-2-1 network
  - Output activation function: none (linear)
  - Hidden-layer activation function: sigmoid
  - Loss function: MSE: $(y^*-y)^2/2$
- Training data
  - 0.2 * 0.5 = 0.1
  - 0.8 * 0.4 = 0.32
  - 0.6 * 0.9 = 0.54
  - 0.3 * 0.7 = 0.21
- Normalization?
- z1=w11x1+w21x2+b1; h1=σ(z1)
- z2=w12x1+w22x2+b2; h2=σ(z2)
- zo=v1h1+v2h2+b3; y*=zo



Worksheet

- For x1=0.2, x2=0.5, y=0.1
- Forward pass: y*=   ; Loss=   ;
- Backpropagation:
  - $v1 \rightarrow_{h1} zo \rightarrow_{1} y^* \rightarrow L$
  - $v2 \rightarrow_{h2} zo \rightarrow_{1} y^* \rightarrow L$
  - $b3 \rightarrow_{1} zo \rightarrow_{1} y^* \rightarrow L$
  - $w11 \rightarrow_{x1} z1 \rightarrow_{\sigma} h1 \rightarrow_{v1} zo \rightarrow_{1} y^* \rightarrow L$
  - $w21 \rightarrow_{x2} z1 \rightarrow_{\sigma} h1 \rightarrow_{v1} zo \rightarrow_{1} y^* \rightarrow L$
  - $b1 \rightarrow_{1} z1 \rightarrow_{\sigma} h1 \rightarrow_{v1} zo \rightarrow_{1} y^* \rightarrow L$
  - $w12 \rightarrow_{x1} z2 \rightarrow_{\sigma} h2 \rightarrow_{v2} zo \rightarrow_{1} y^* \rightarrow L$
  - $w22 \rightarrow_{x2} z2 \rightarrow_{\sigma} h2 \rightarrow_{v2} zo \rightarrow_{1} y^* \rightarrow L$
  - $b2 \rightarrow_{1} z2 \rightarrow_{\sigma} h2 \rightarrow_{v2} zo \rightarrow_{1} y^* \rightarrow L$
  - Find dL/v1, .., dL/b2
  - Update v1’ = v1-rdL/dv1, …
  - Compute new L’ and compare with L