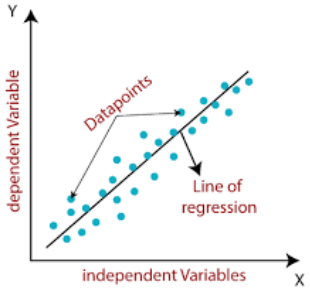
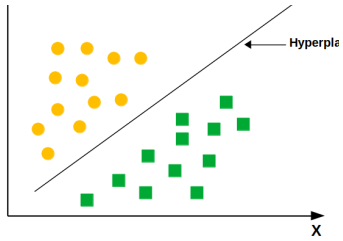
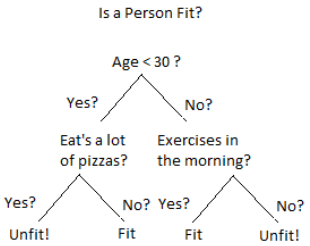
Machine Learning to Deep Learning

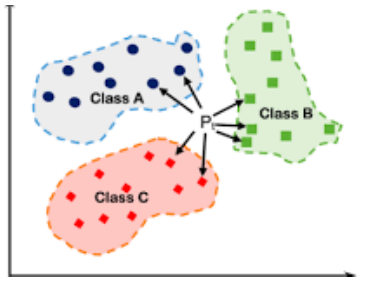
AI overview

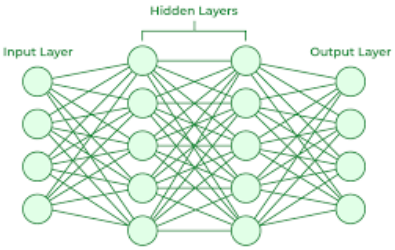






- Artificial Intelligence
  - Old school (1950~80)
    - Rule-based AI (Expert systems) – *programmed intelligence*
    - Search & Planning (A*/minimax) - *optimization*
  - Machine Learning - *learn from data*
    - Traditional ML – *manual feature selection*
      - Linear/Logistic Regressions
      - 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: $P(\text{spam}|\text{words}) \propto P(\text{words}|\text{spam}) \times P(\text{spam})$ — 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.

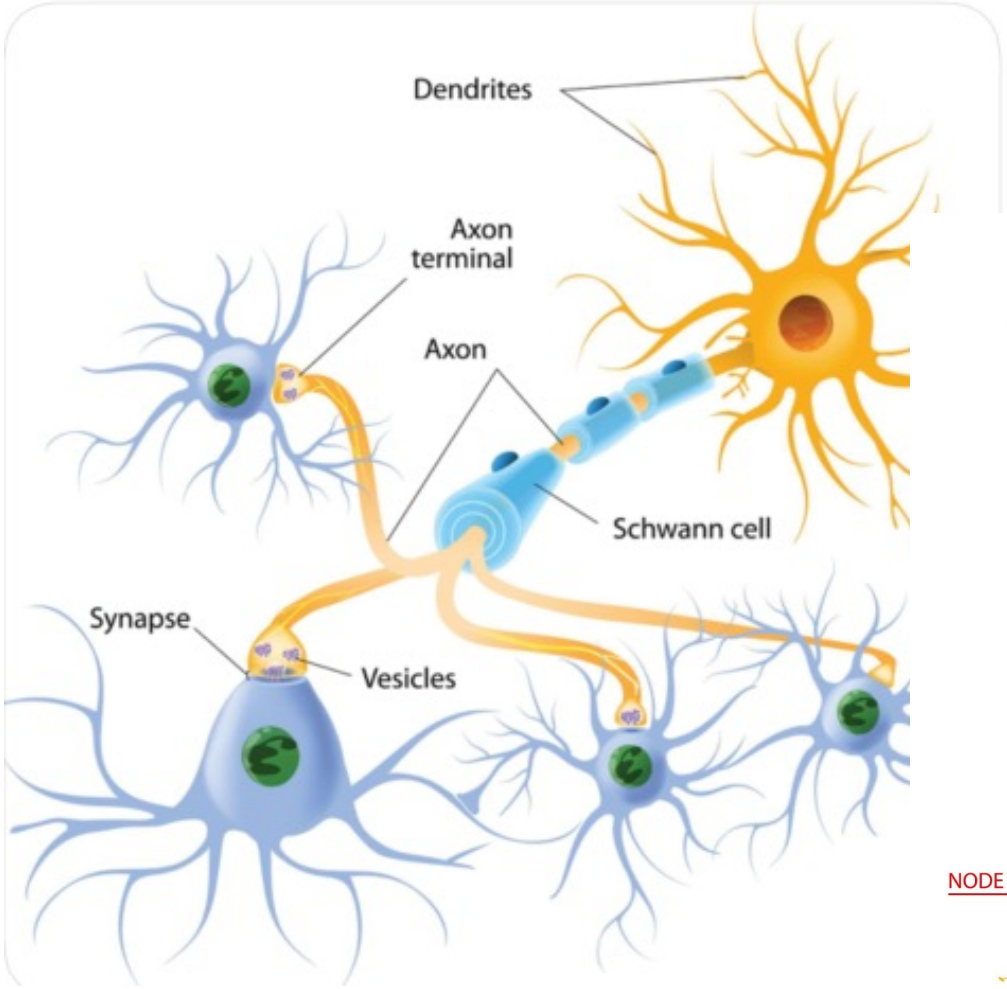
What can go wrong:

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

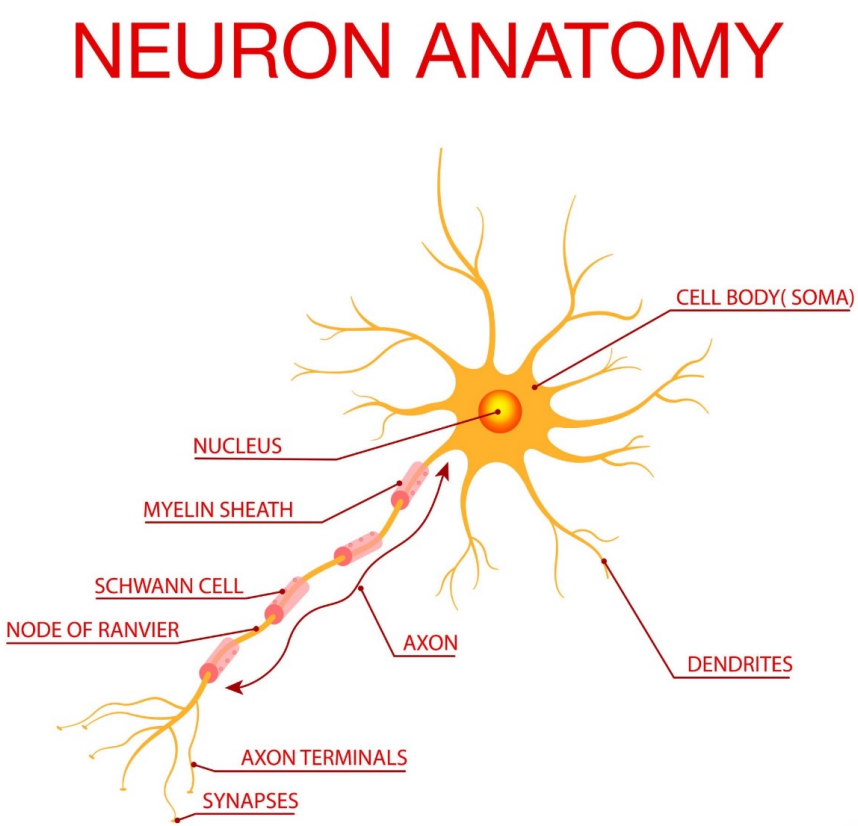
Why This Works:

- 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

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

Simple Example: XOR Problem

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

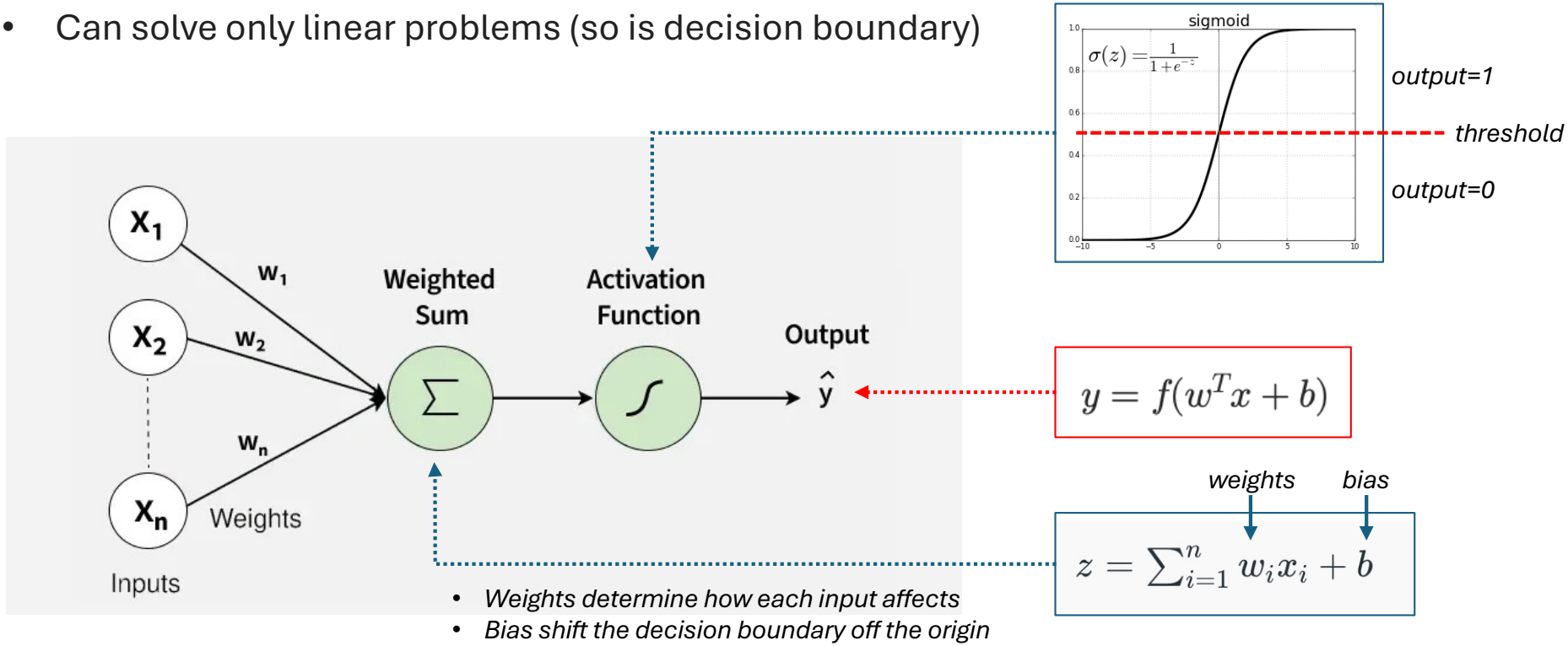
Input: (1,0) → Output: 1 | Input: (1,1) → Output: 0

❌ 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)



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

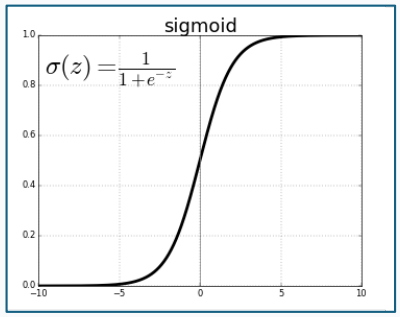
Activation Functions: Adding Non-linearity

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

Sigmoid

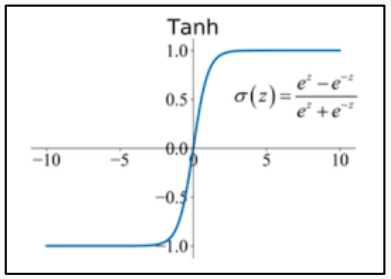
- Output: (0, 1)
- Smooth, probabilistic
- Vanishing gradients

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



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

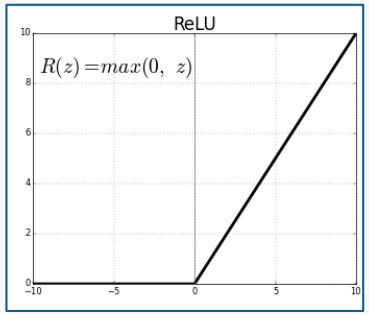
$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



ReLU

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

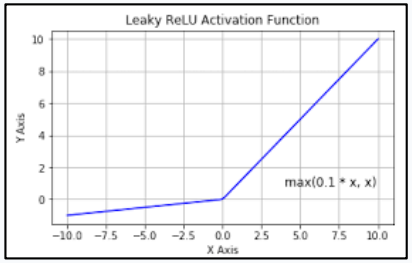
$$\max(0, x)$$



Leaky ReLU

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

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



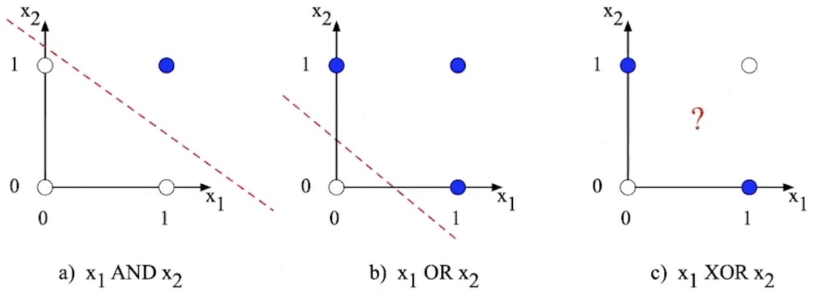
Single perceptron cannot solve XOR

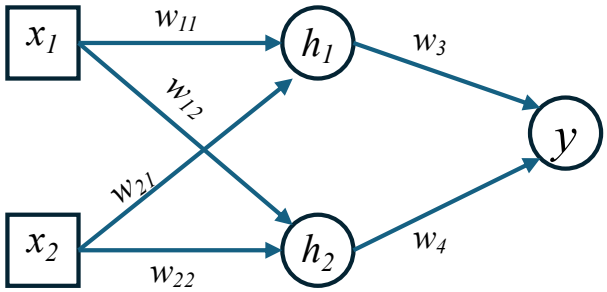
- XOR decision cannot be expressed by a linear boundary
- Mathematical contradiction
  - $f(0,0)=0: b < 0$
  - $f(0,1)=1: w2+b > 0$
  - $f(1,0)=1: w1+b > 0$
  - $f(1,1)=0: w1+w2+b < 0$
  - $w1+w2 > -2b$
  - $w1+w2 < -b$
  - $-2b < -b$
  - $b > 0$: *contradiction*
- Solution
  - Add one hidden layer

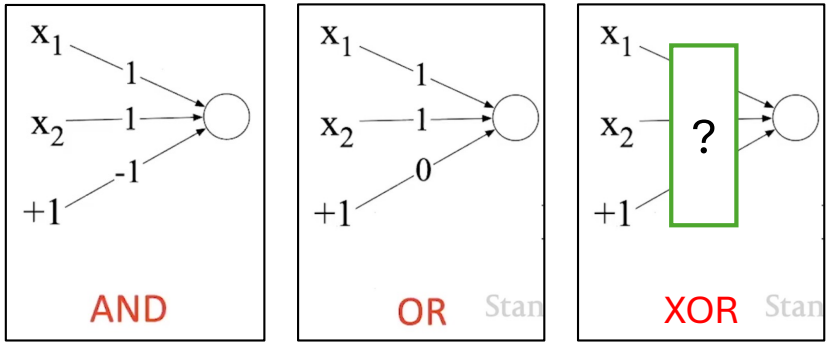
$$y = \text{sign}(w_1 x_1 + w_2 x_2 + b)$$

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

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

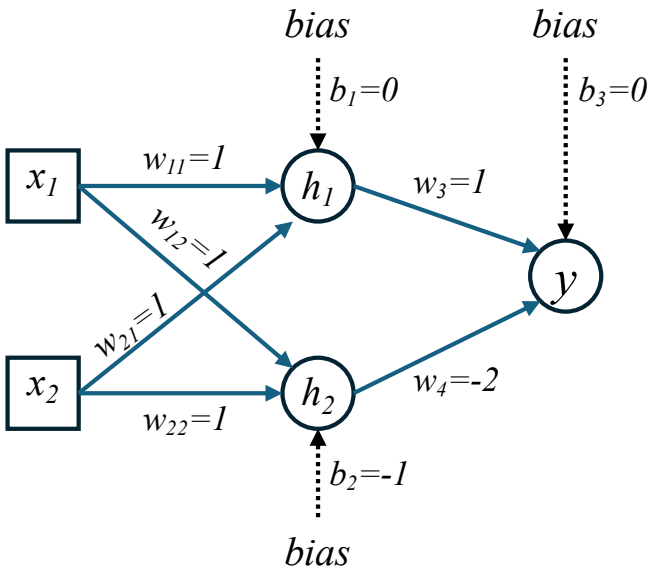






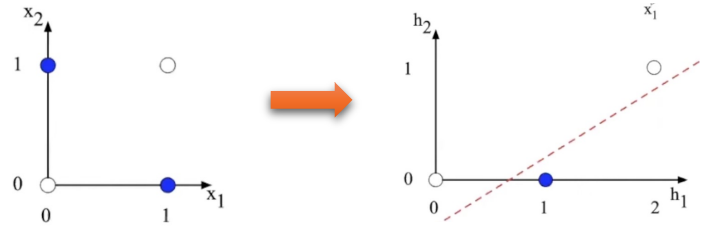
Add a hidden layer to solve XOR

- 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)

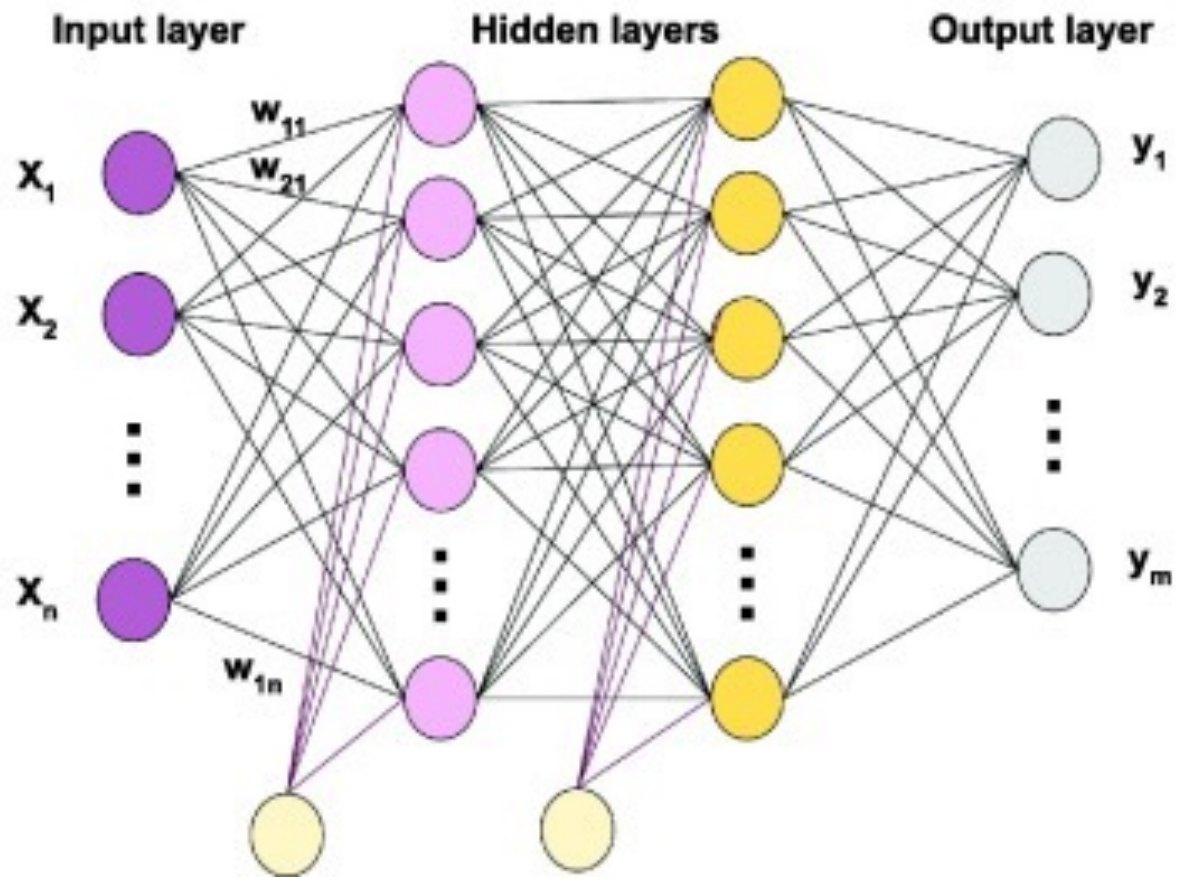
hidden space (linearly-seperable)

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

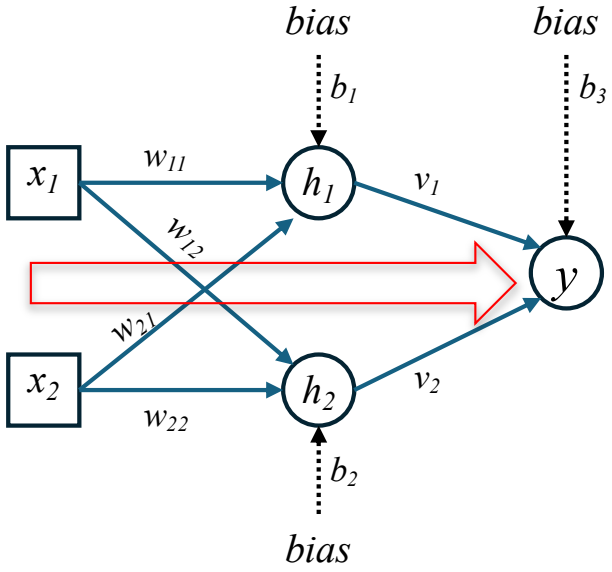
Multi-layer Neural Network

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

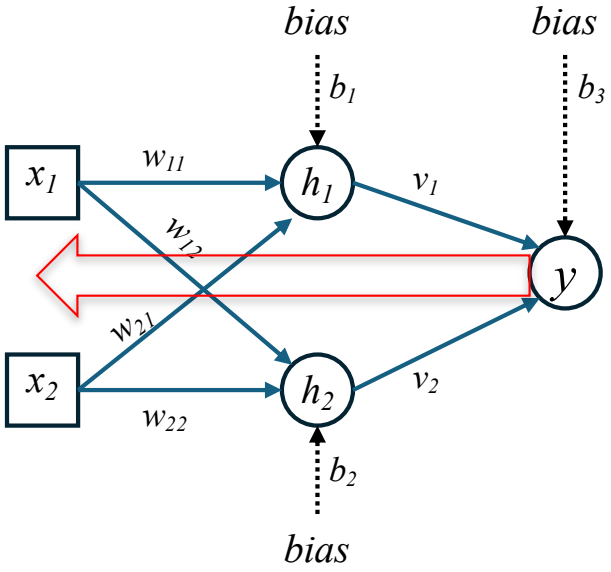


How can we find the weights/biases?

- 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



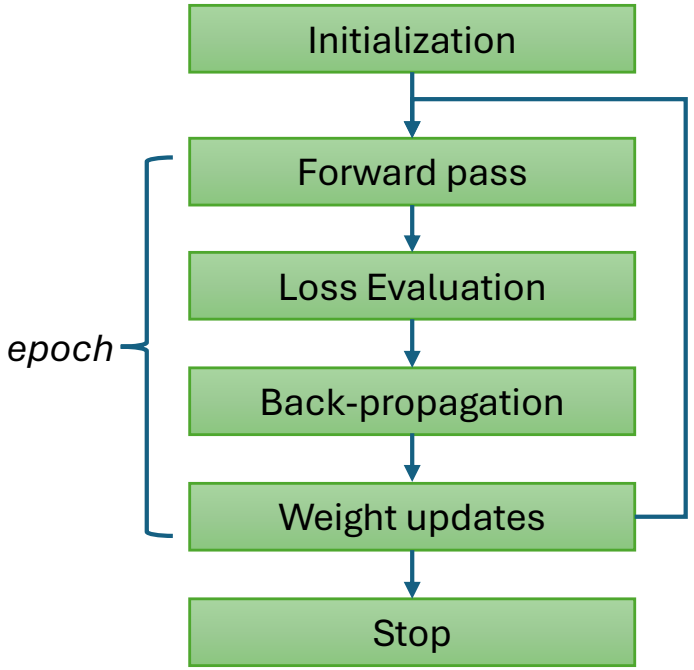
Forward pass

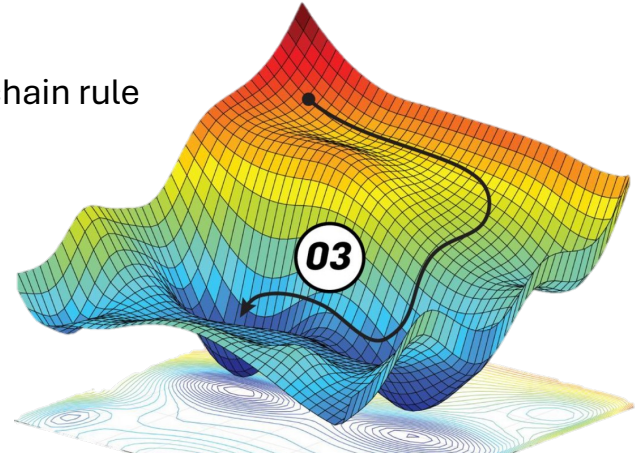


Backward propagation

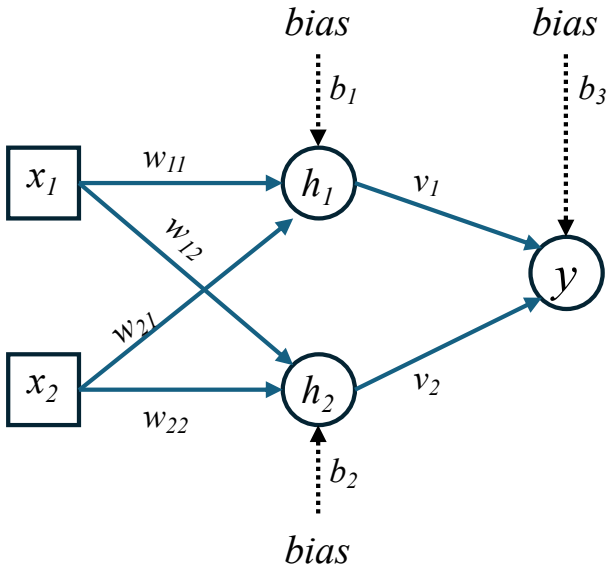
How a Neural Network Learns

- *"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





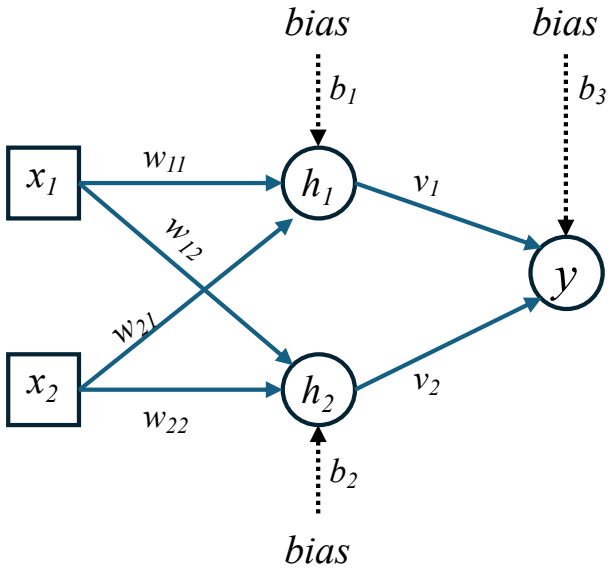
XOR: Design

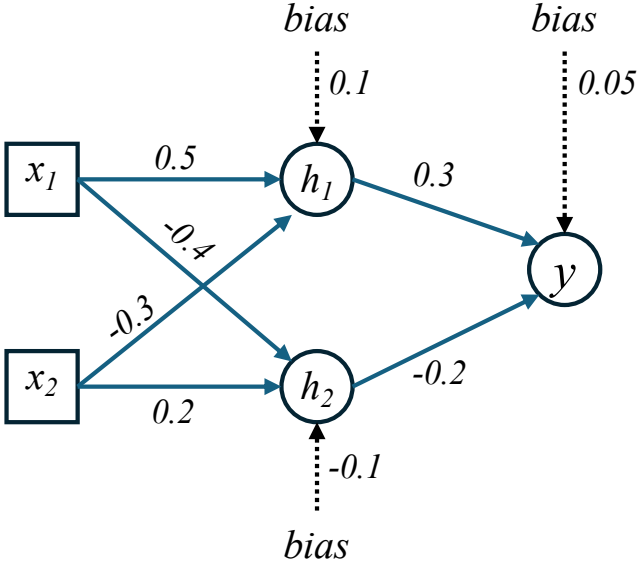


- Architecture
  - 2 inputs
  - 2 hidden neurons
  - 1 output neuron
  - (2-2-1) neural network
  - Activation function: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Transforms:
  - Params: $\theta = \{w_{11}, w_{12}, w_{21}, w_{22}, b_1, b_2, v_1, v_2, b_3\}$
- Loss function
  - Use binary cross entropy: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

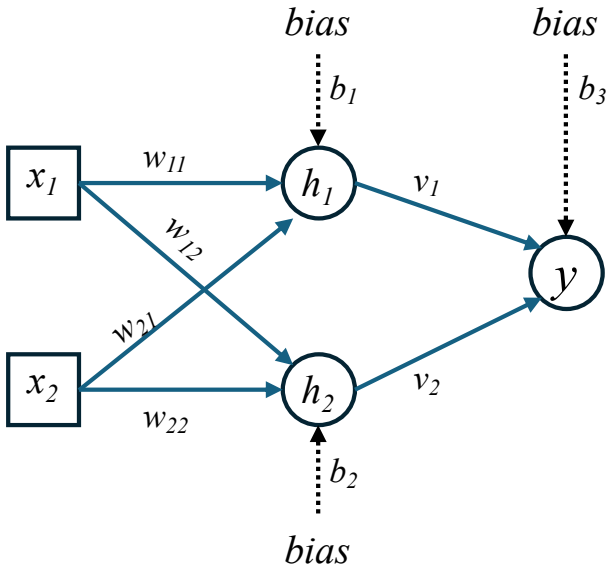
XOR: Initialization

- 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



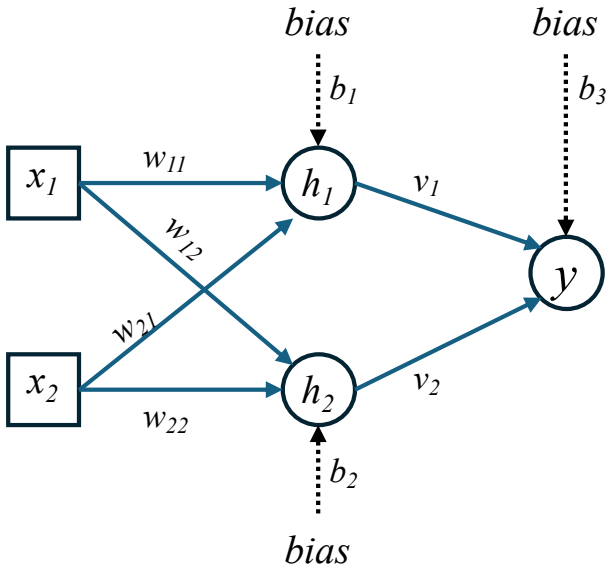


XOR: Forward Pass



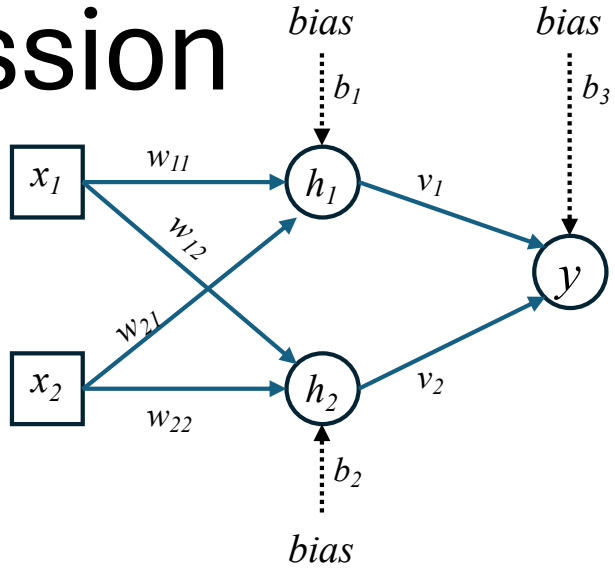
- (x1, x2) = (1, 0) → y_{target}=1
- z1=
- h1=
- z2=
- h2=
- z3=
- y=
- y_{target}=1

XOR: Back Propagation and weight update

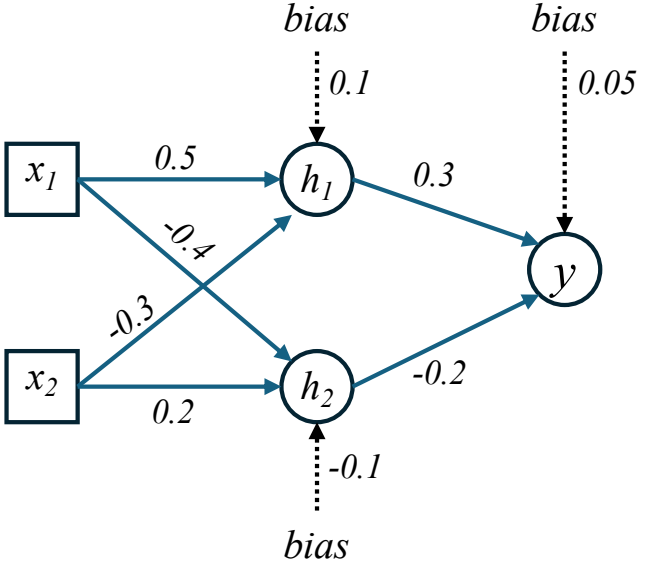


- v1’ = v1 – 0.1 * dL/dv1
- v2’ = v2 – 0.1 * dL/dv2
- w11→z1→h1→z3→y→L
- …
- dL/dw11 = (y^-y)v1h1(1-h1)x1
- w11’ = w11-0.1*dL/dw11
- …

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

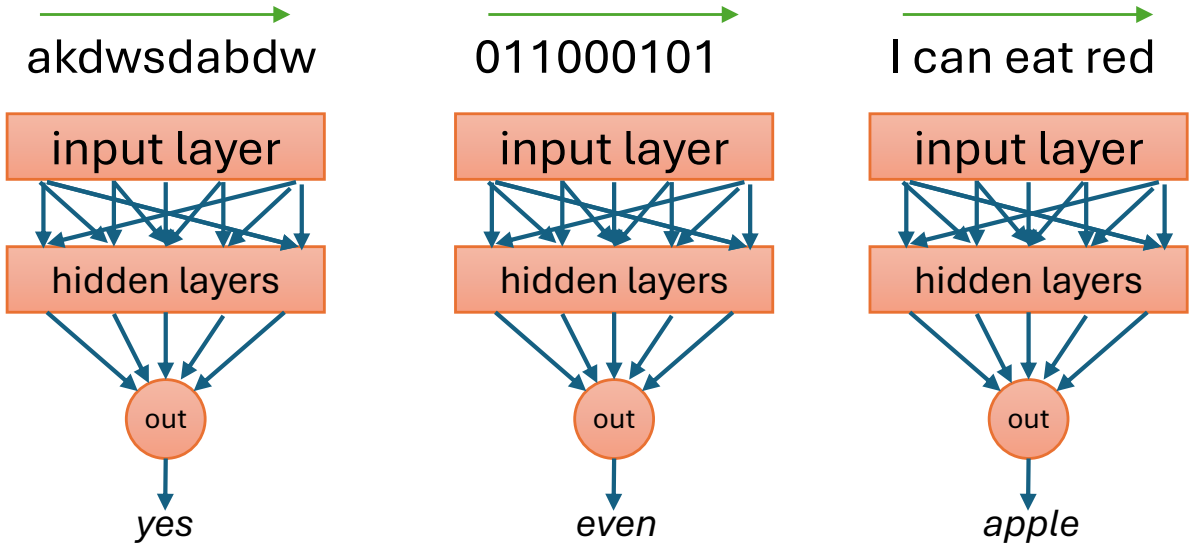


Worksheet

- 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

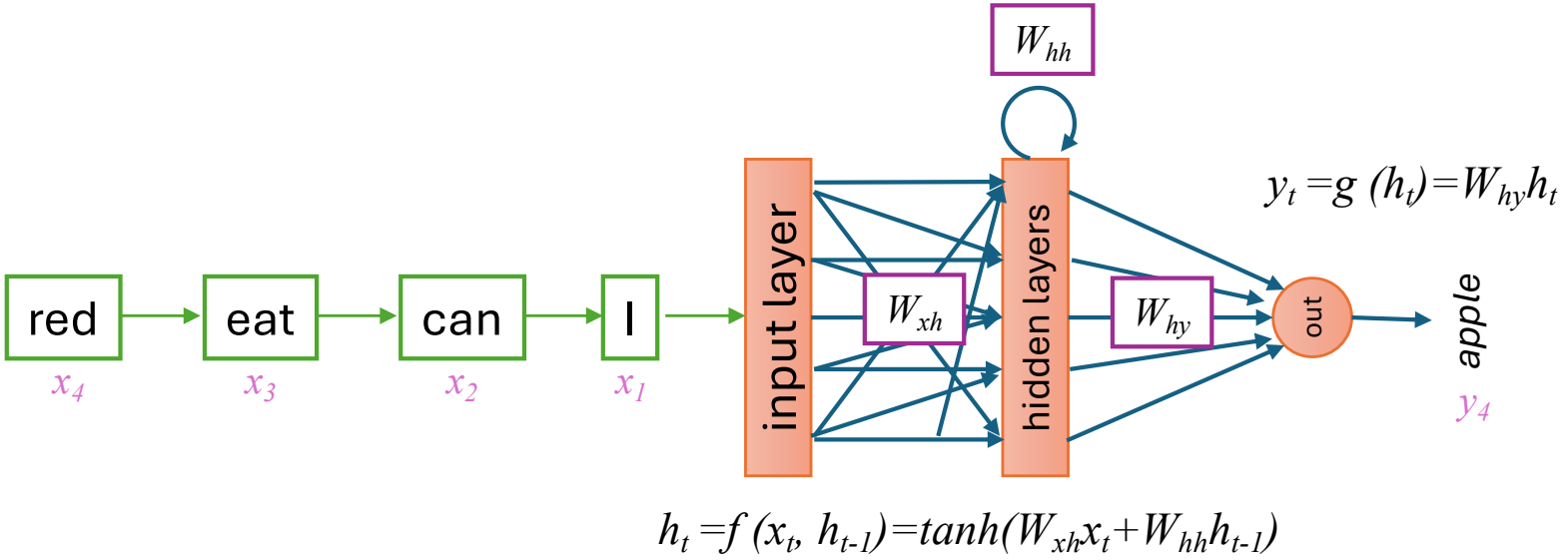
Limitation of Feedforward ANN

- Can it handle a series of input ?
  - From a stream of alphabets, tell if b comes after a
  - Given a stream of bits, compute paritiy bit
  - Given a series of words, which word is the most likely to come?
- Use the long input or sliding-window input
- What is we want to output based on a variable length intput stream?
- ANN can only optimize compute $y=f(x)$ with static x

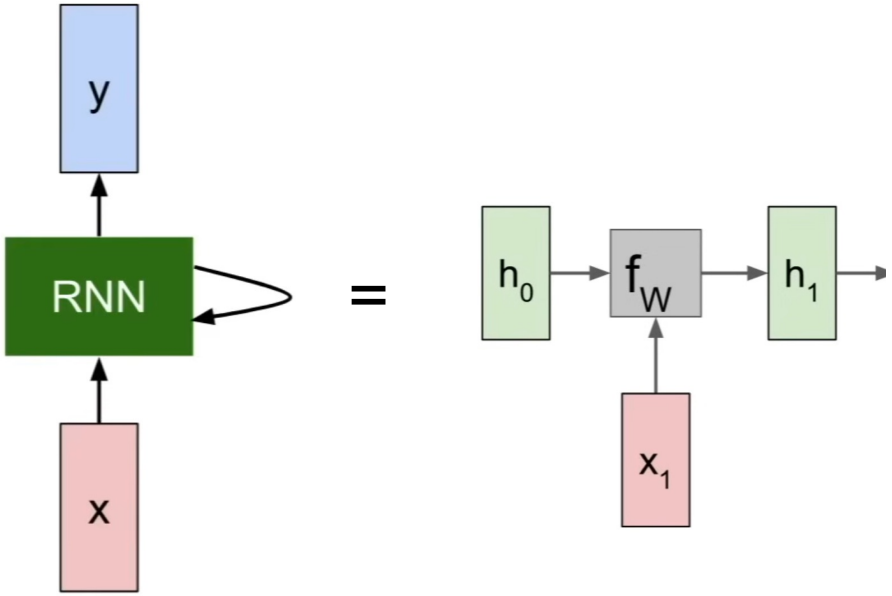


What is RNN (Recurrent)?

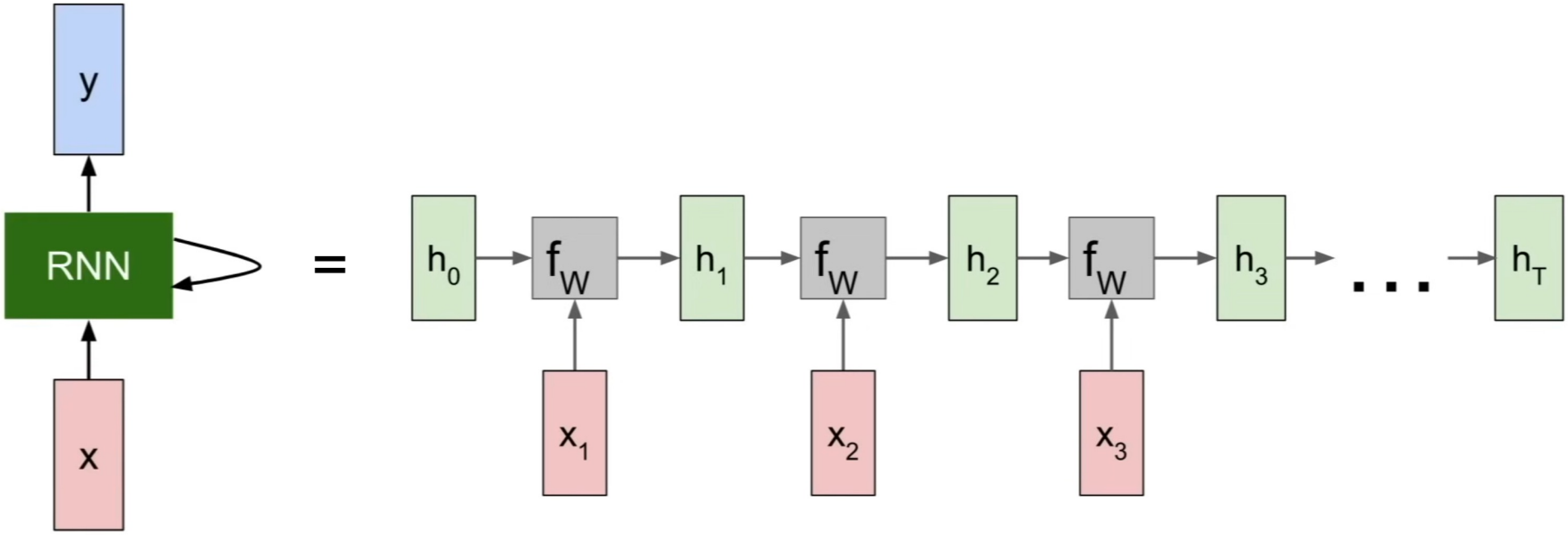
- At each time t, compute
  - $h_t = f(x_t, h_{t-1})$ and $y_t = g(h_t)$
- Current state depends on the previous state(s)
  - Memory of the past
  - Order sensitivity
  - Variable-length input
- Feedforward ANN: “Classify this image.”
- RNN: “Predict the next word in this sentence.”



RNN in time-series form

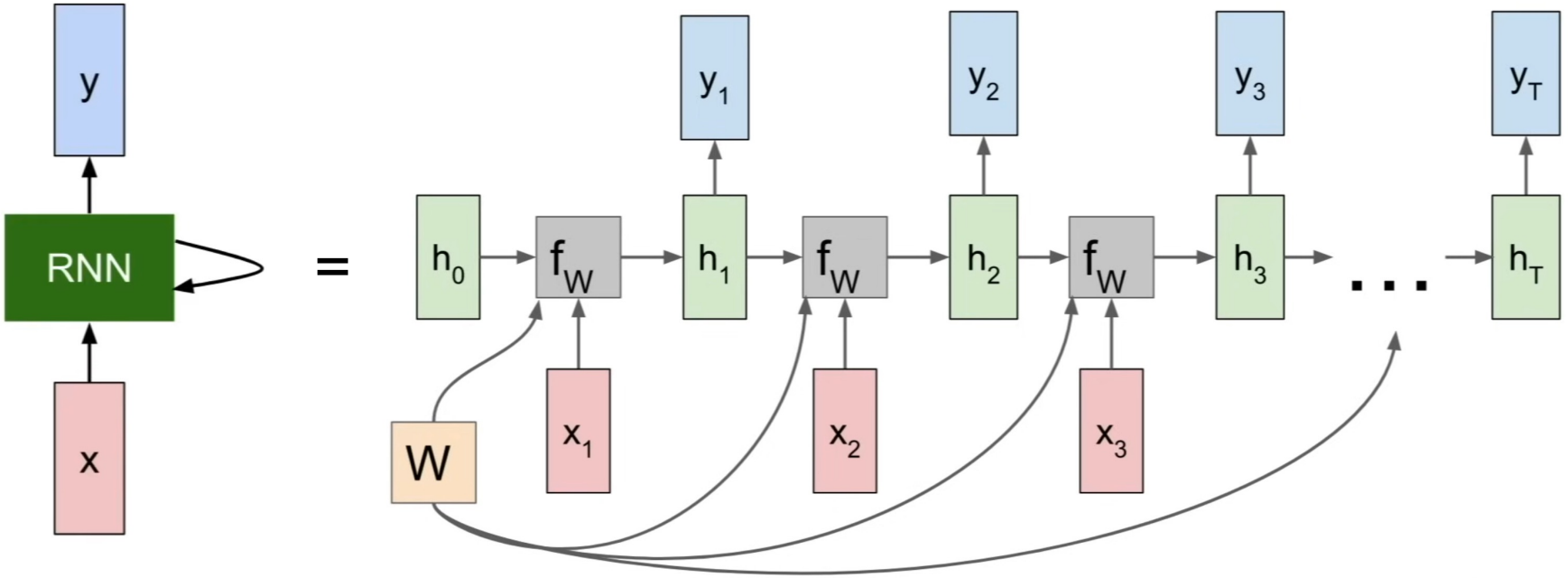


RNN in time-series form



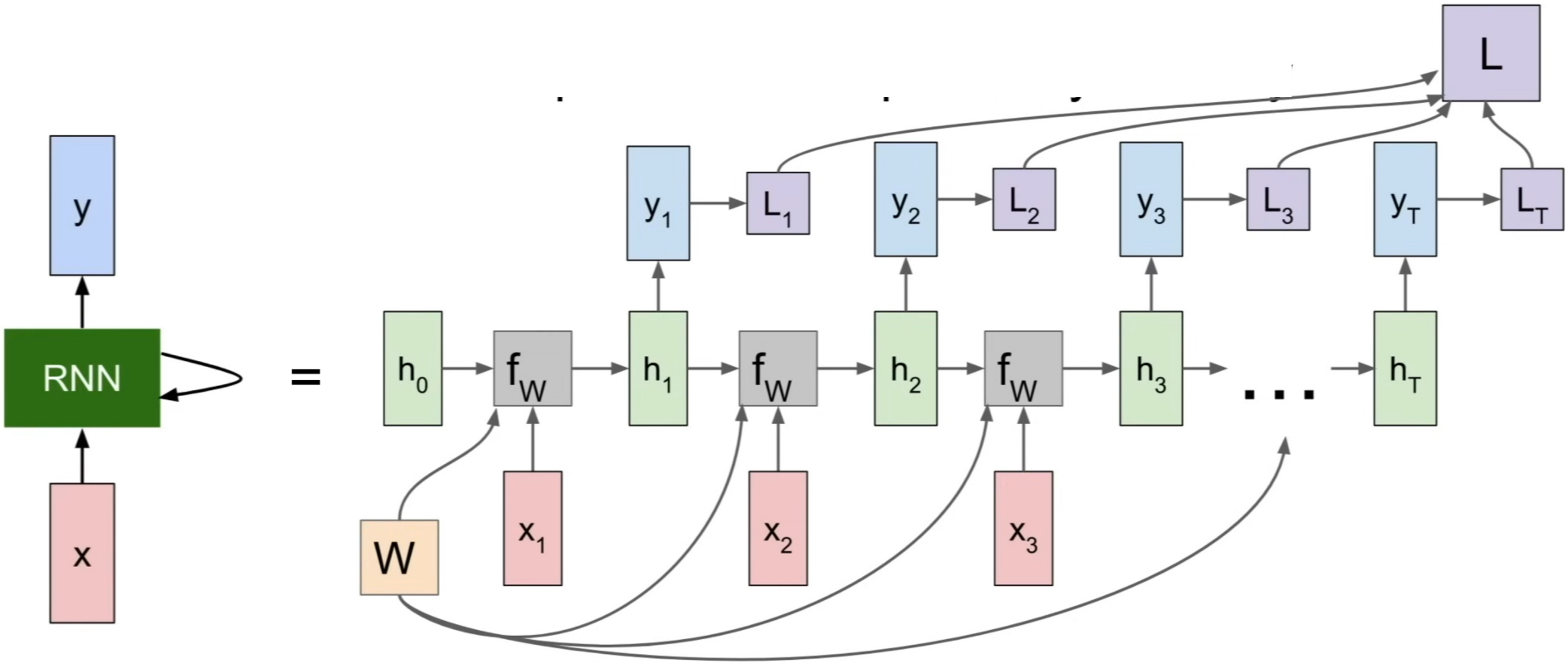
Hidden layers are chained over time

RNN in time-series form



Each step shares the same weight parameter

RNN in time-series form



Aggregation of loss: sum, average, weighted sum, masking, ...

BPTT Backpropagation Through Time

$dL/dW_2 = (\text{change via } y_2 + \text{change via } h_3)/dh_2 * dh_2/dW_2$

$= \{ (dL/dy_2 * dy_2/dh_2) + (\text{via } y_3 + \text{via } h_4)/dh_3 * dh_3/dh_2)\} * dh_2/dW_2$

$= \{ \ldots + (dL/dy_3 * dy_3/dh_3 + (\text{via } y_4 + \text{via } h_5)/dh_4 * dh_4/dh_3)*dh_3/dh_2*dh_2/dW_2$

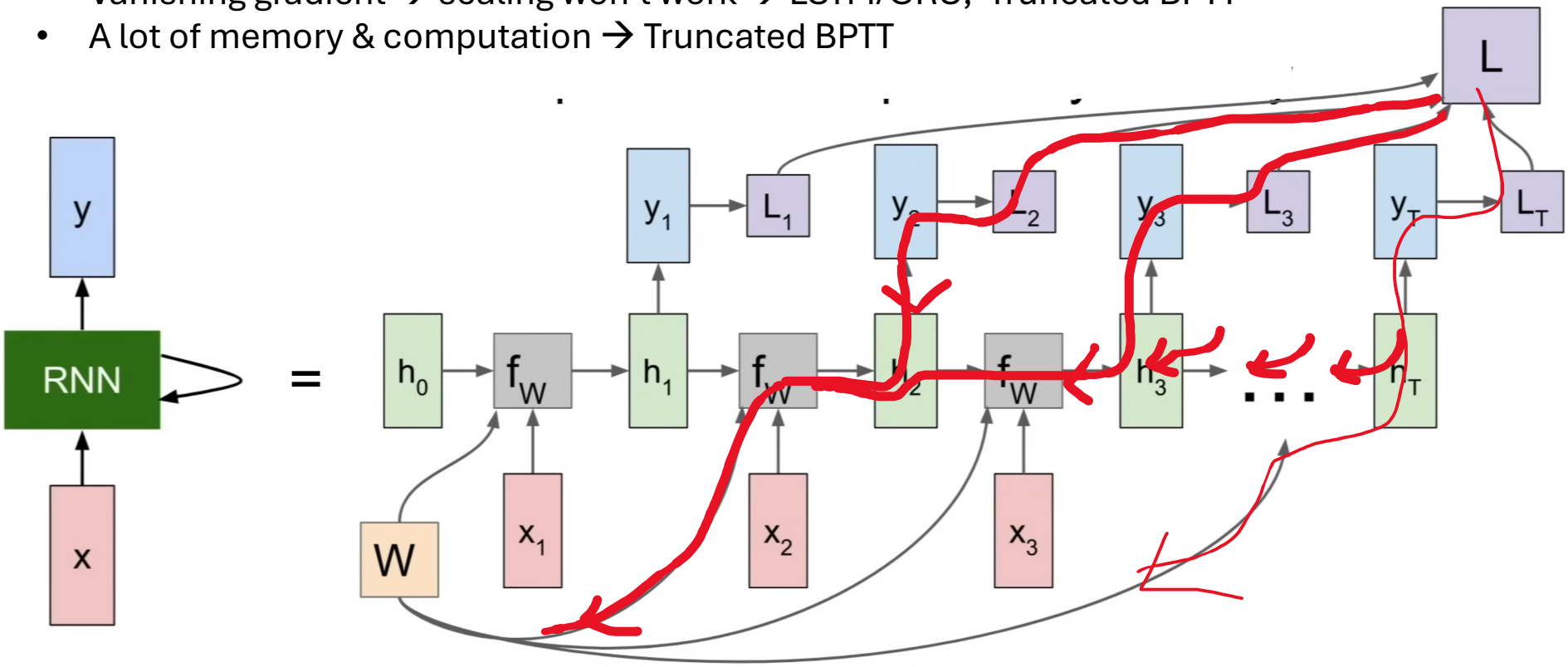
$= \ldots + dh_T/dh_{T-1}*dh_{T-1}/dh_{T-2}*\ldots.*dh_4/dh_3*dh_3/dh_2*dh_2/dh_1*dh_1/dW_1 + \ldots$

$dh_t/dh_{t-1} = (1-\tanh^2(z))*W_{hh}$

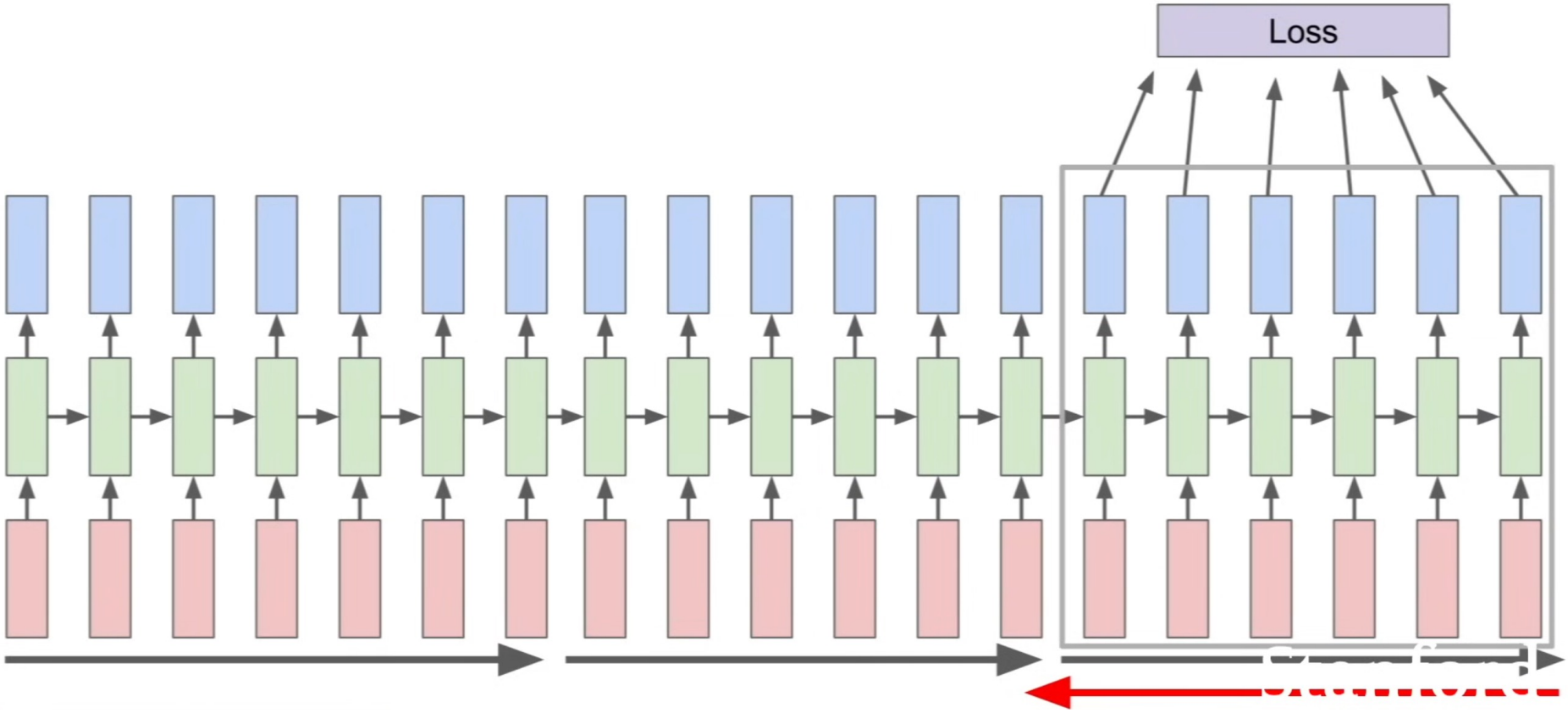
$[*] = (1-\tanh^2(z_1))*\ldots*(1-\tanh^2(z_T))*W_{hh}^T$

Problem

- Exploding gradient → scaling down works → Gradient clipping
- Vanishing gradient → scaling won't work → LSTM/GRU, Truncated BPTT
- A lot of memory & computation → Truncated BPTT

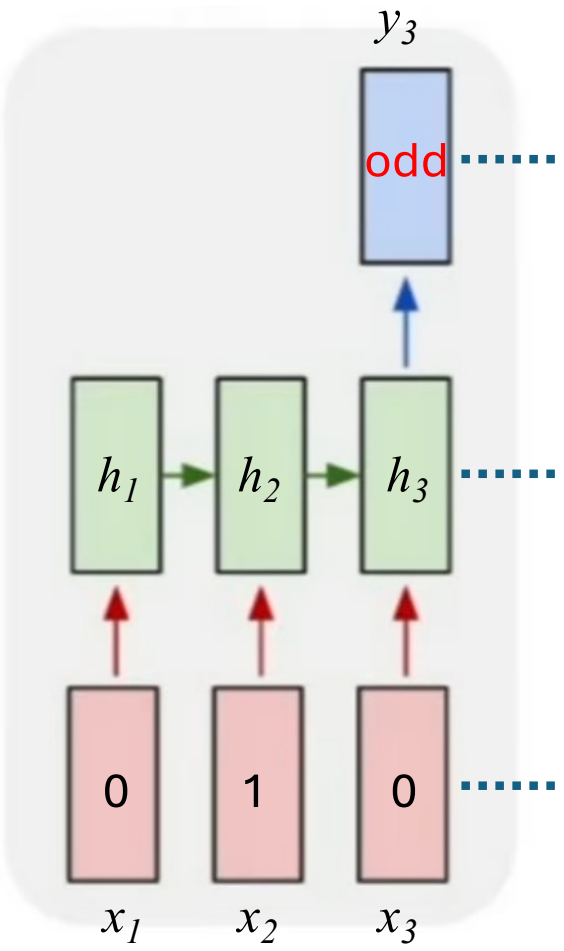


Trauncated BPTT



Example: Bit-stream XOR (Parity)

- Input: bit-stream
- Output: XOR of all bits so far
  - 1 if odd # of bits so far
  - 0 if even # of bits so far
- Actually RNN is not good for this
  - Need LSTM



RNN for variable-size input/output

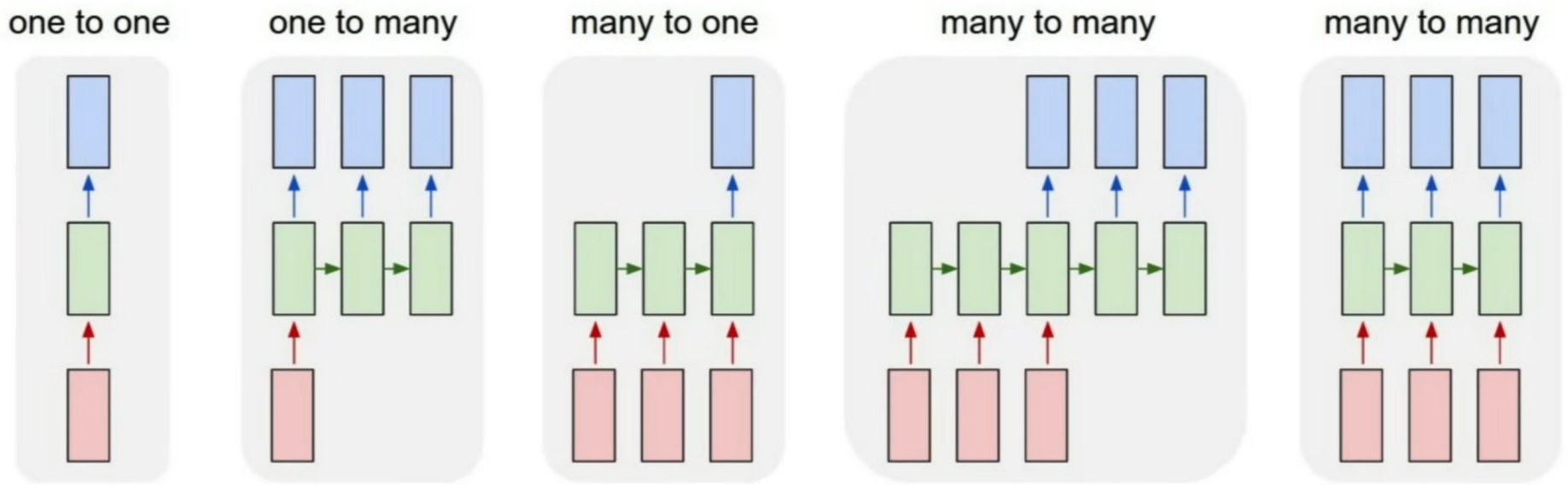


Image captioning
image → sentence

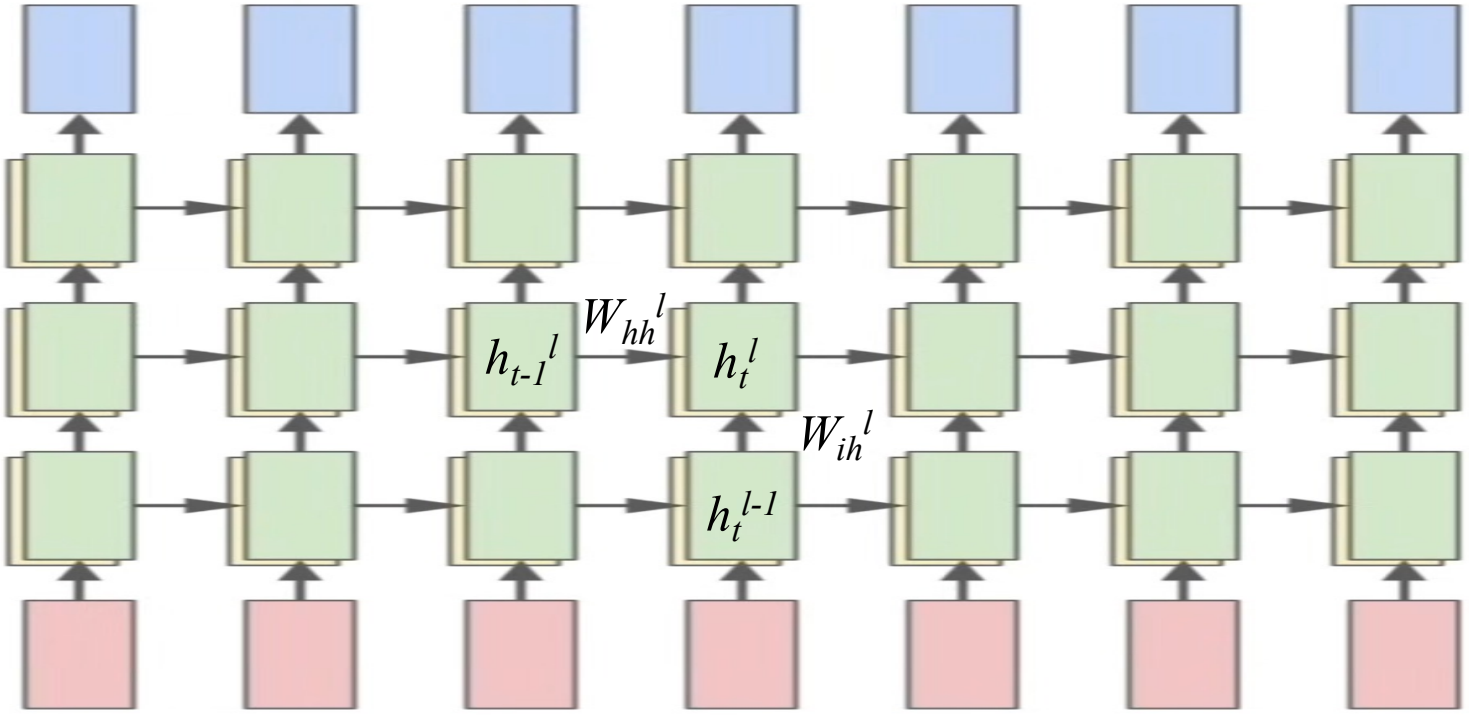
Translation
Seq of words → seq of words

Sentiment classification
Seq of words → sentiment
seq of frames → activity

Frame-level classification
seq of frames → seq of classes

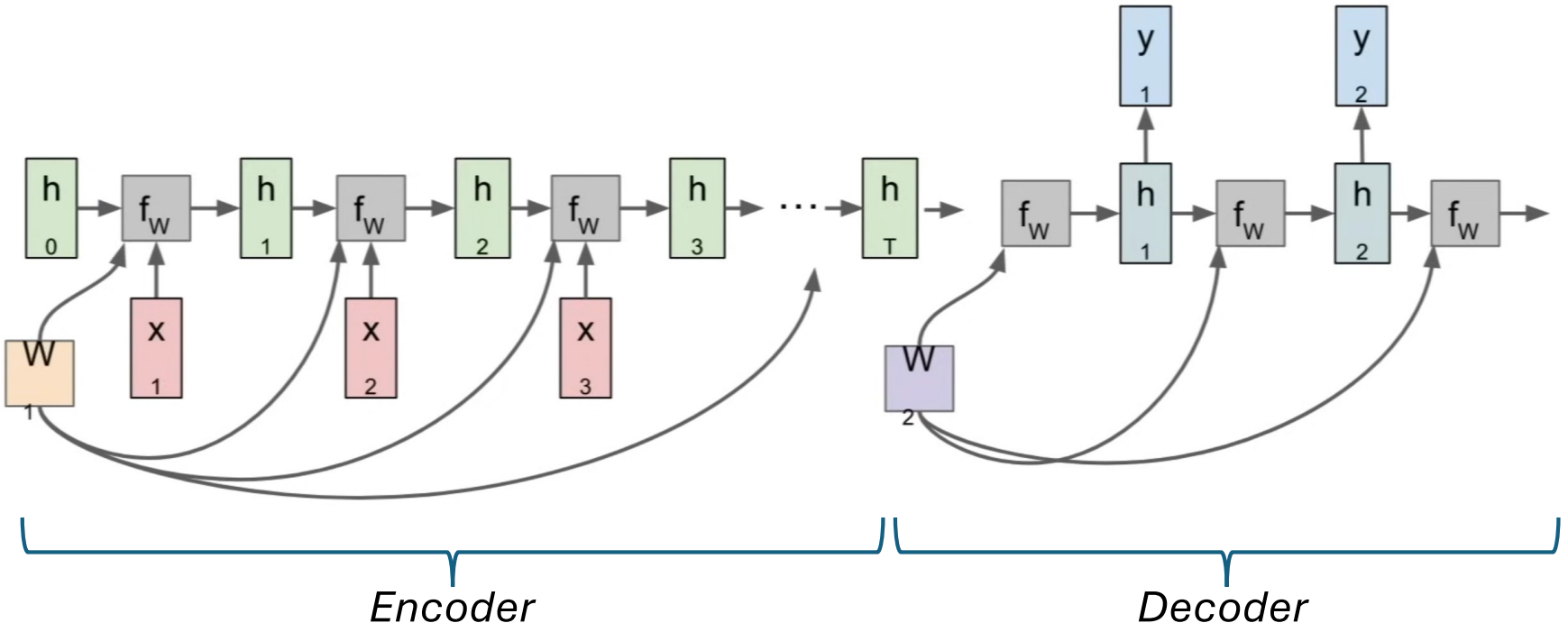
Multilayer RNN

- $h_t^l = tanh(W_{hh}^l * h_{t-1}^l + W_{ih}^l * h_t^{l-1})$



Encoder/Decoder RNN (Seq2Seq)

- Many-to-Many = Many2One (Encoder) + One2Many (Decoder)
- Encoder: Encode input sequence into a single vector
- Decoder: Decode a single vector into an output sequence



Success story of RNN (2010-2017)

- RNNs shine when processing sequential data where order matters and context accumulates over time.
- Language Modeling & test generation
  - Predict the next word (or character) given previous words.
  - Smartphone autocomplete, shakespeare-like text, linux kernel code, wikipedia, …
  - Limitation: short term memory
    - The chef who made the soup in the morning were…
- Machine Translation
  - Use Encoder/Decoder architecture
  - Encoder RNN: Reads French sentence, produces fixed-size context vector
  - Decoder RNN: Generates English words one at a time
  - Limitation: the context vector is a bottleneck (loss of information)
    - The man who wrote the book that I read yesterday was…
- Time series forcasting
  - Predict future values from historical sequences (stock prices, energy demand, medical data)
  - Capture temporal dependencies and trends
  - Limitation: Cannot extrapolate beyond training distribution, Sensitive to outliers and distribution shift, easy to disturb by adding noise

Limitations of Vanilla RNN

- The Vanishing/Exploding Gradient Problem
  - Resulting in short term memory
  - "Alice went to the store. She met Bob there. Later, she told Carol about it. [50 more words…] She was happy." → RNN forgets who 'she' refers to after 50+ words.
- Sequential Processing = Slow Training
  - Cannot parallelize across time steps (cf. Transformer)
- Fixed Context Window (Seq2Seq Models)
  - In encoder-decoder RNNs, the entire input must compress into a fixed-size vector.
  - led to Attention mechanism (Decoder looks back at ALL encoder hidden states)
  - → Transformer ("Attention Is All You Need", 2017)
- Difficulty with Bidirectional Context
  - "The bank near the river…" ← Need to see "river" to know it's a riverbank
  - Bidirectional RNN: Process forward and backward
  - Transformer solves by self-attention

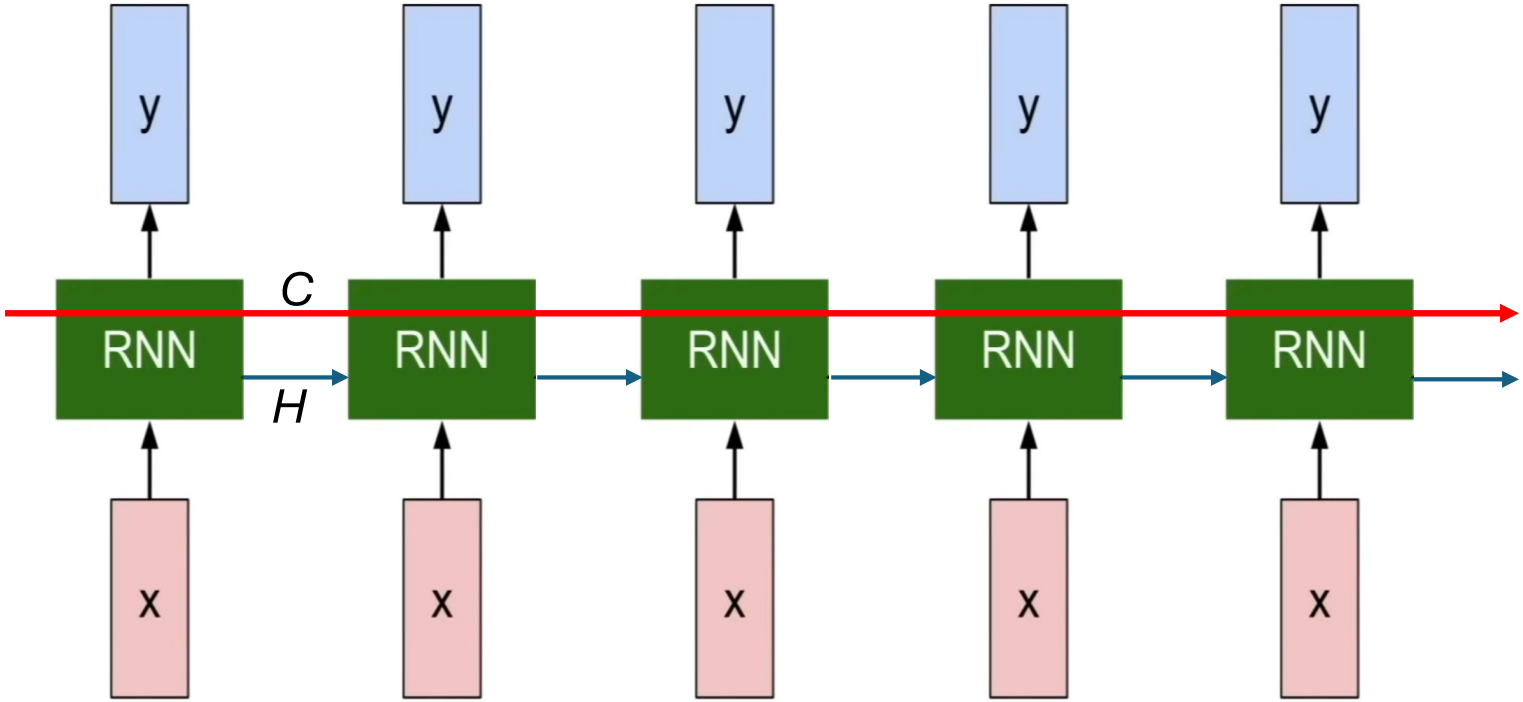
Vanishing Gradient with BPTT

- This is the structural problem: At every step, hidden states are overwritten
  - Information at step 1 must actively preserved by the weight matrix !

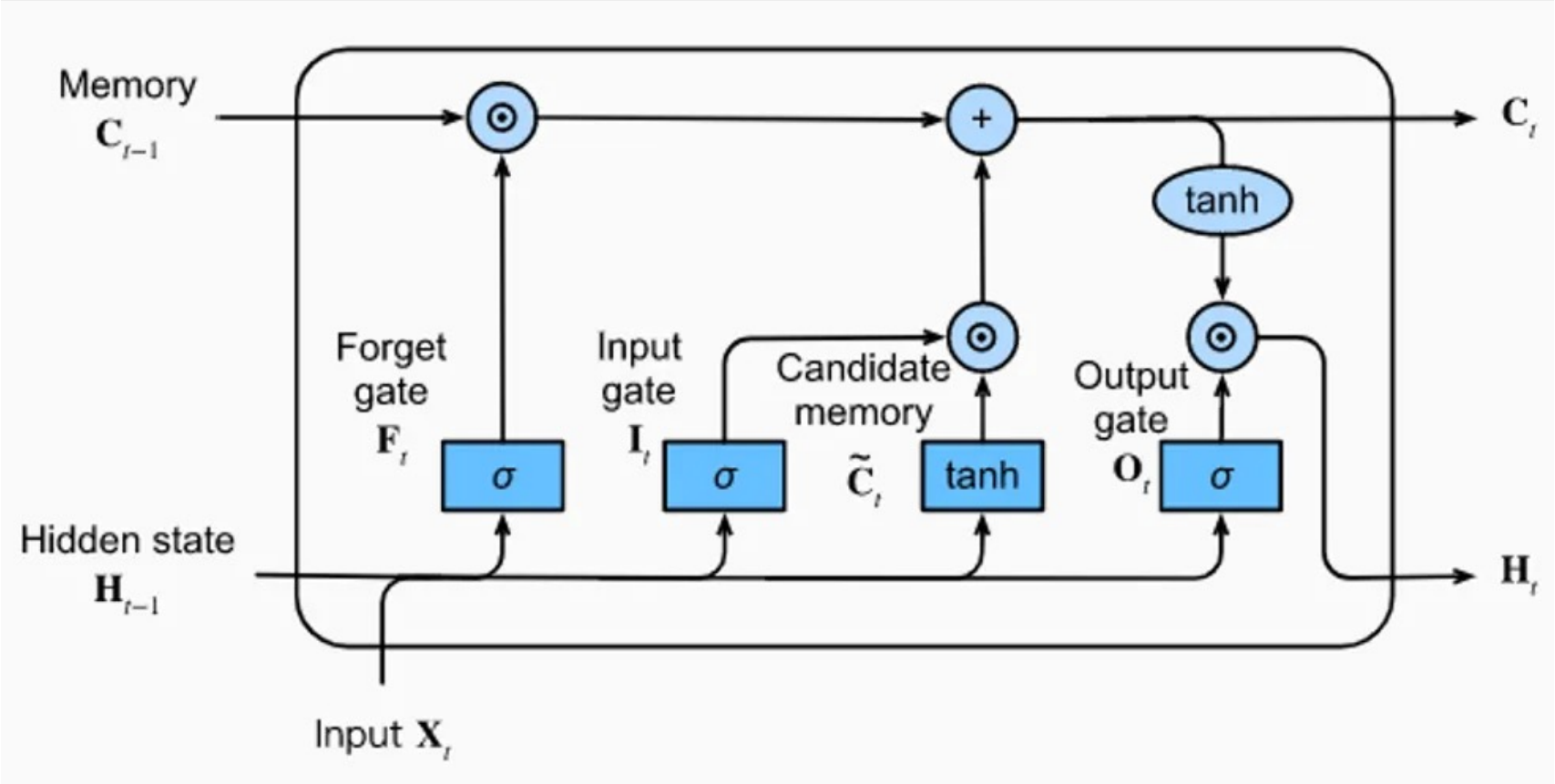
$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} W_{hh}^T \mathrm{diag}(f'(z_i))$$

Long Short-Term Memory

- Separate memory from computation
  - "what if we had a dedicated memory lane that information could flow through with minimal transformation?" → ***cell state*** c_t → *additively* update (i.e., linearly)



Long Shot-Term Memory



Long Shot-Term Memory

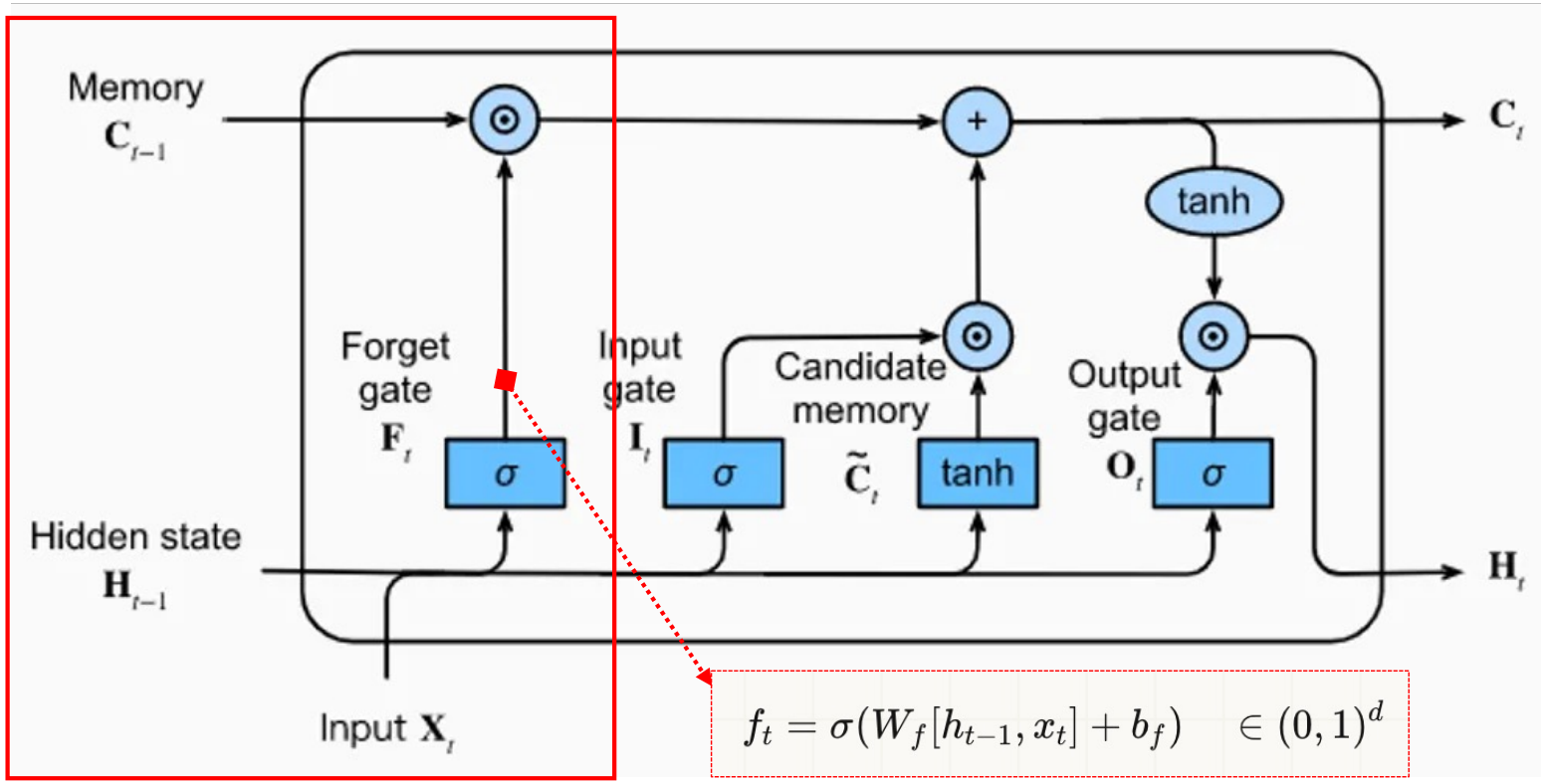
- Gates
  - A fully-connected neural network
  - Control what goes in and out of memory by sigmoid
  - Takes input (x_t)and previous hidden state (h_{t-1}) as input
  - Generates a selection vector = σ([x, h]) = (s1, s2, … ,sN)
    - if s_i ~= 0 : remove the effect of ith element
    - if s_i ~= 1 : keep the effect of ith element
  - Multiplied with a cell state
    - Gate selectively keeps/removes the effect of ith element

Steps

- Step 1: determine what to forget from long-term memory
- Step 2: determine which new (short-term) memory to add to long-term memory
- Step 3: determine what to output from long-term memory (mixed with new short-term memory)

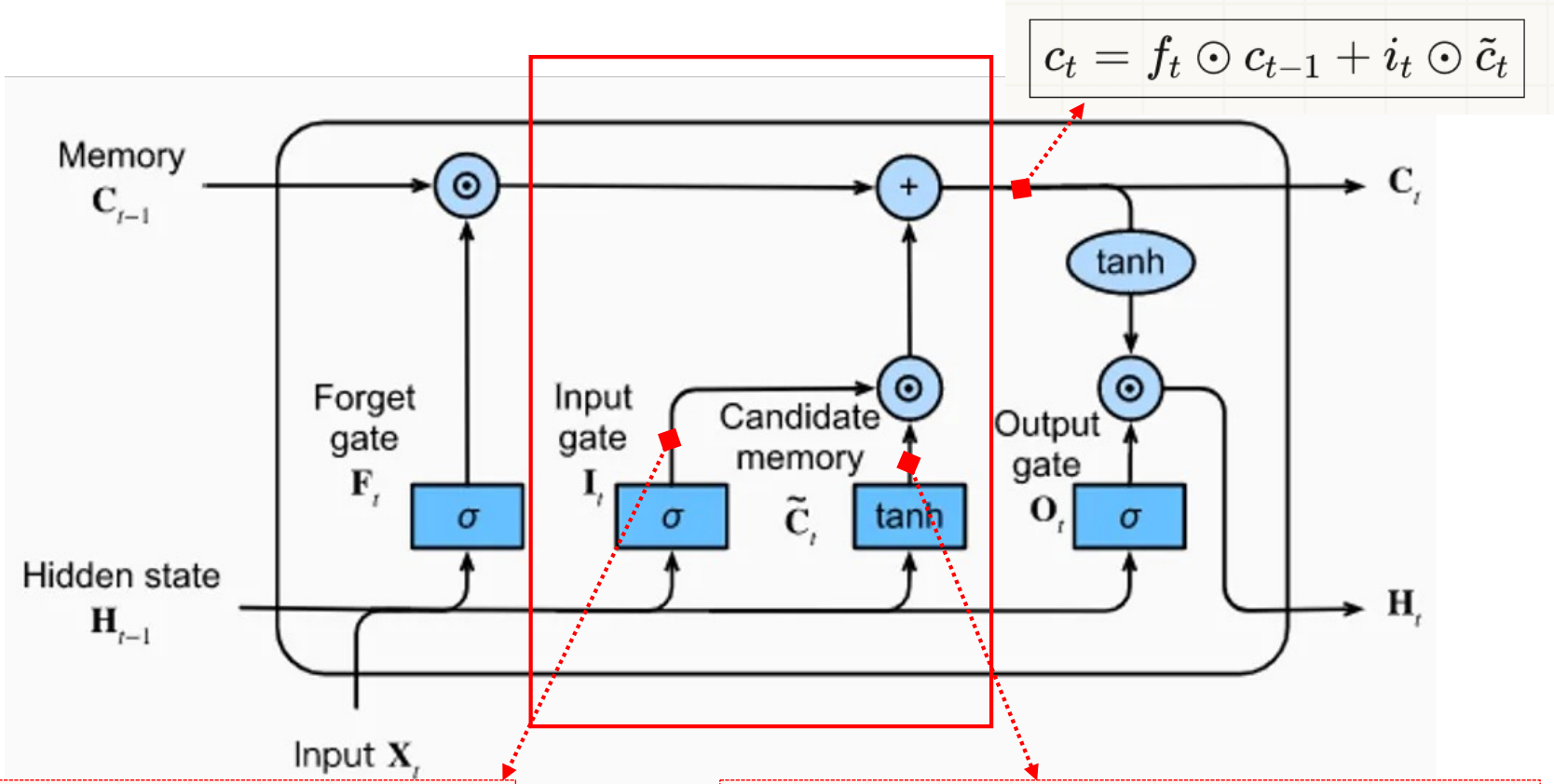
Step 1: Forget gate

- decides (based on input *(x_t)* and previous hidden state *(H[t−1]))* what information to remove from the cell state vector coming from time *t− 1 (C[t-1])*.



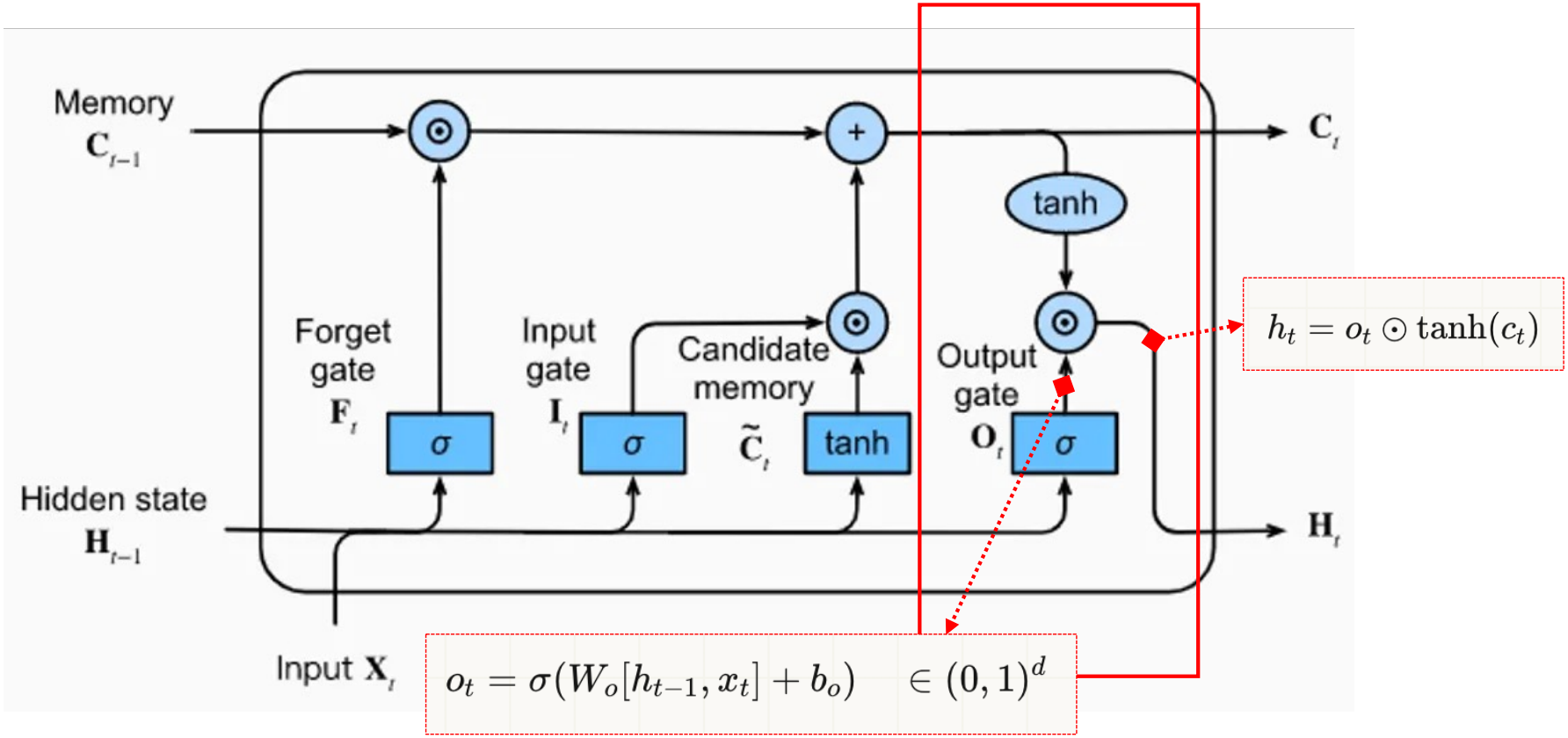
Step 2: Input gate * Candidate memory

- Candidate memory: a vector of information that is candidate to be added to the cell state, normalized by tanh
- Input gate: determine which elements of candidate memory are **added** to the cell state vector after step 1



Step 3: Output gate

- determines H[t], the value of the hidden state outputted by the LSTM (in instant *t*)
- by multiplying a selector vectore to the normalized cell state



Benefit of LSTM

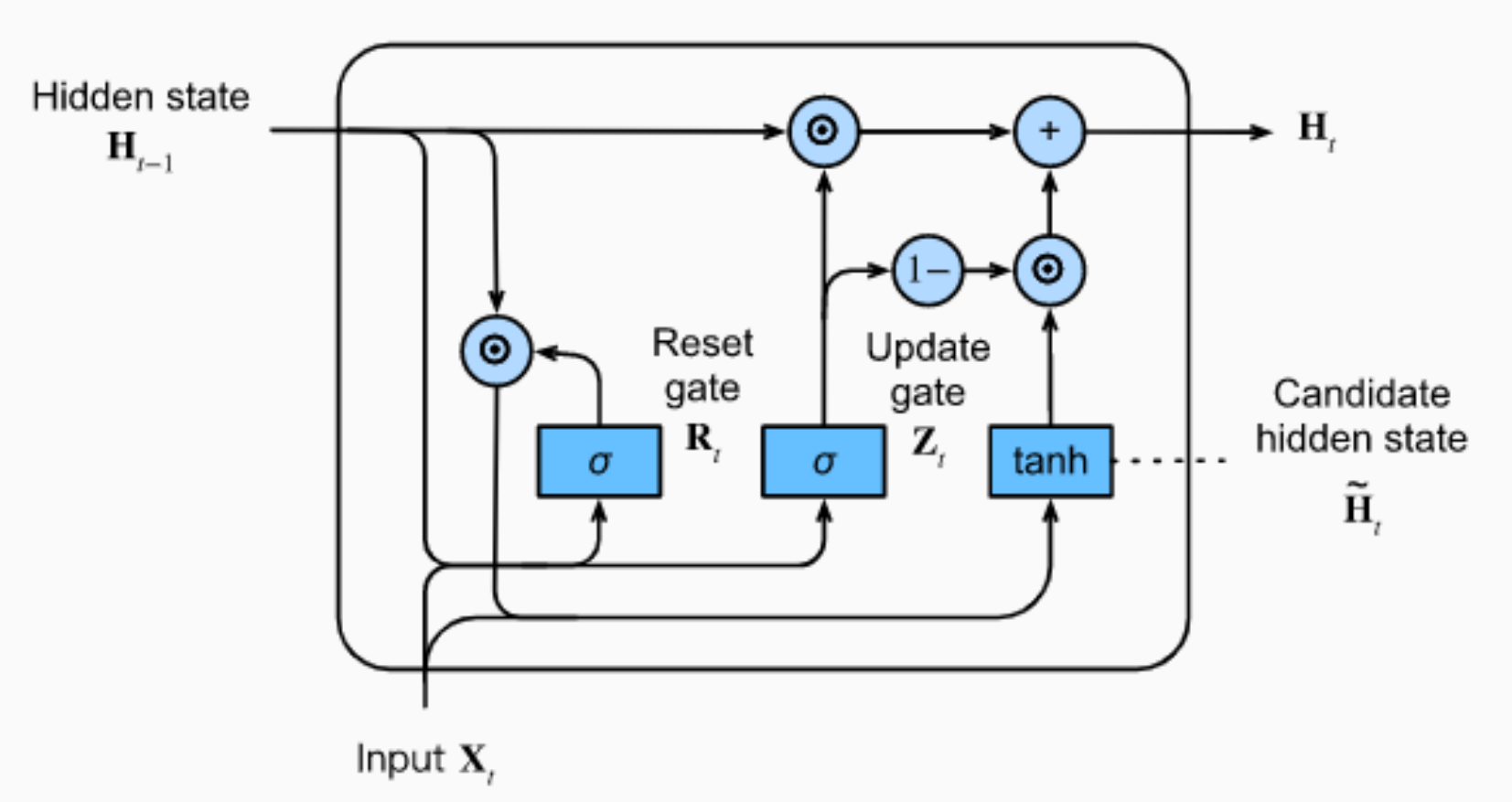
- Removes vanishing gradient problem of RNN
  - RNN: $\prod_{t=k}^{T-1} W_h^\top \cdot \mathrm{diag}(\tanh'(\cdot))$
  - LSTM: $\frac{\partial \mathcal{L}}{\partial c_k} = \frac{\partial \mathcal{L}}{\partial c_T} \cdot \prod_{t=k}^{T-1} \mathrm{diag}(f_{t+1})$
- If the network learns to keep f_i close to 1 (it's better to keep long term memory), cell state creates a "highway" for gradients (Constant Error Carousel)
- Hidden states still have vanishing gradient problem (only short term memory survives)

Limitations of LSTM

- Complexity
  - Each step contains four neural networks (x4)
  - Number of parameters is four times larger
- Computation cost
- Redundant architecture
  - separate forget gate, input gate, output gate

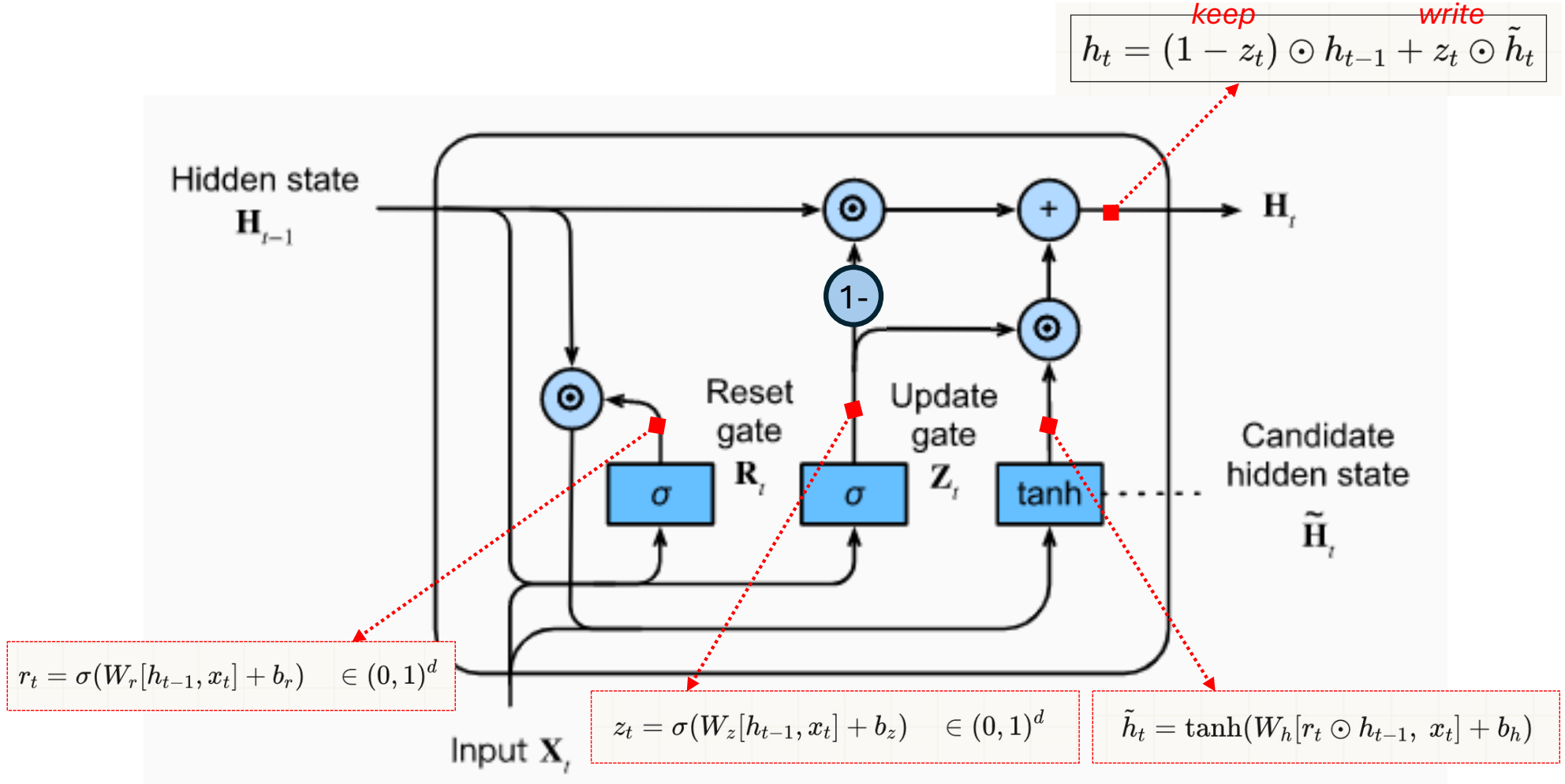
GRU (Gated Recurrent Unit)

- Use only two gates: Reset gate / Update gate (forget + input)
- Uses 3 neural networks
- No cell state: only hidden
- Decides how much of the **old hidden state** to keep, and how much **new information** to write.



GRU Gates

- Reset gate: how much of the previous hidden state to use when computing the candidate:
- Update Gate: how much of the previous hidden state to keep vs. replace:



GRU vs LSTM

- GRU Update gate == LSTM forget gate + input gate
  - LSTM: forget & input are independent (f_t + i_t != 1)
  - GRU: forget & input are tied (keep + write = 1)
    - New information replacees old information
- GRU reset gate
  - Control how much old information contributes to the candidate hidden state
- GRU
  - keep long term memory with z_t ~= 0
  - ignore past with r_t ~= 0
- GRU Parameter
- Gradient flow:

$$\frac{\partial h_t}{\partial h_{t-1}} = \mathrm{diag}(1 - z_t) + \mathrm{diag}(z_t) \cdot \frac{\partial \tilde{h}_t}{\partial h_{t-1}}$$

  - if z_t ~= 0: long term dominates → CEC
  - if z_t ~= 1: short term dominates → fresh candidate

Comparison

Property	Vanilla RNN	LSTM	GRU
States	h_t	h_t, c_t	h_t
Gates	0	3 (f, i, o)	2 (r, z)
Parameters	$d(d+n+1)$	$4d(d+n+1)$	$3d(d+n+1)$
Long-range deps	Poor	Excellent	Good–Excellent
Training speed	Fast	Slow	Medium
Forget/input independence	N/A	Yes	No (tied)

Examples for LSTM

- LSTM
  - Language Modeling / Text Generation
    - Given tokens $x_1, \ldots, x_{t-1}$, predict x_t.
  - Named Entity Recognition in Long Documents
    - Tag each token as PERSON, LOCATION, etc.
  - Handwriting Recognition
    - Predict characters from a sequence of coordinates.j
  - Speech Recognition
    - Map acoustic feature sequences to character sequences

Examples for GRU

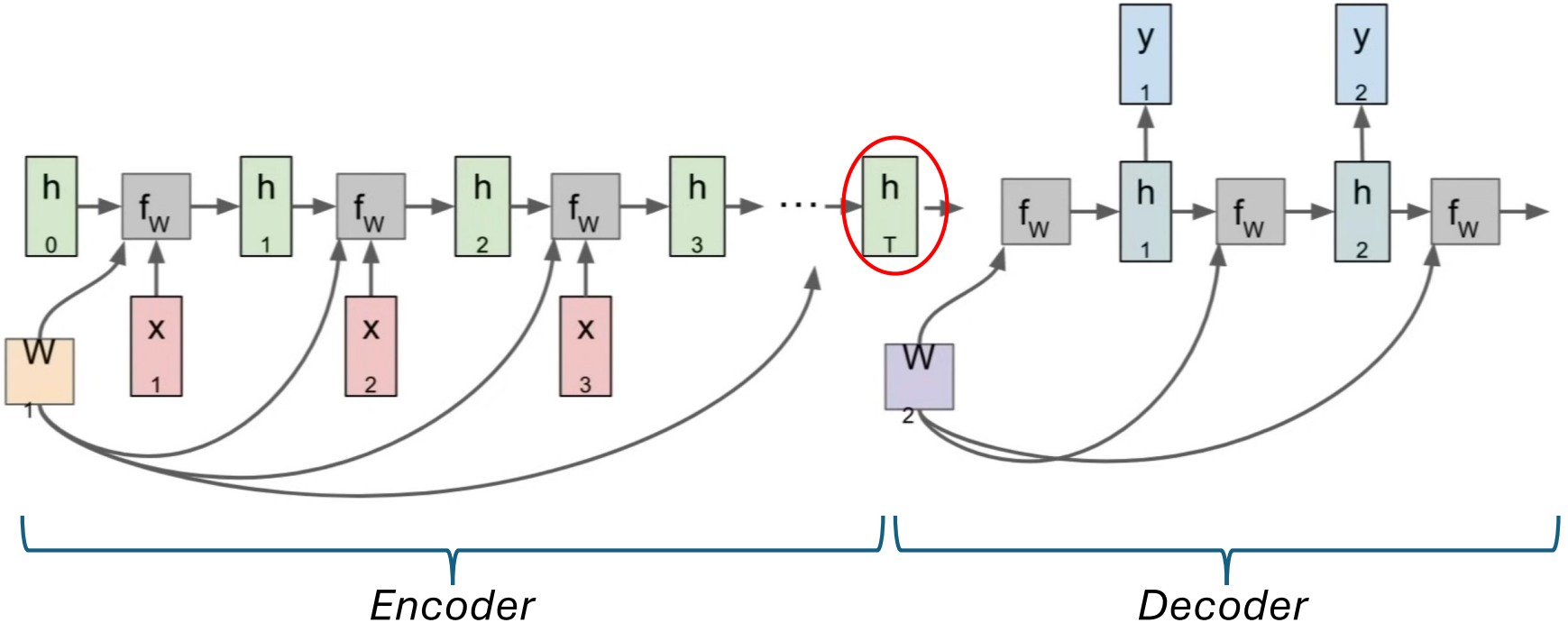
- GRU
  - Sentiment Classification on Reviews
    - Given a product review of ~100–200 words, output positive/negative/neutral.
  - Machine Translation with Shorter Sentences
    - Encoder-decoder translation (e.g., English → French).
  - Real-Time Anomaly Detection in Sensor Streams
    - Given a stream of sensor readings, flag anomalies in near real-time.
  - Music Generation
    - Model a sequence of (pitch, duration, velocity) events and sample new music.

Attention

- Model to **look back at the most relevant parts of previous information instead of compressing everything into one fixed-size hidden state**.
- In RNN, LSTM, GRU, model reads tokens one by one and carries information forward in hidden states.
- The past must be packed into a limited hidden representation
- Distant information can still become hard to preserve
- When generating an output, the model may need one specific earlier word, but the hidden state is only a summary
- Attention fixes this by letting the model say:
- “At this moment, which earlier positions should I focus on most?”

Encoder/Decoder RNN (Seq2Seq)

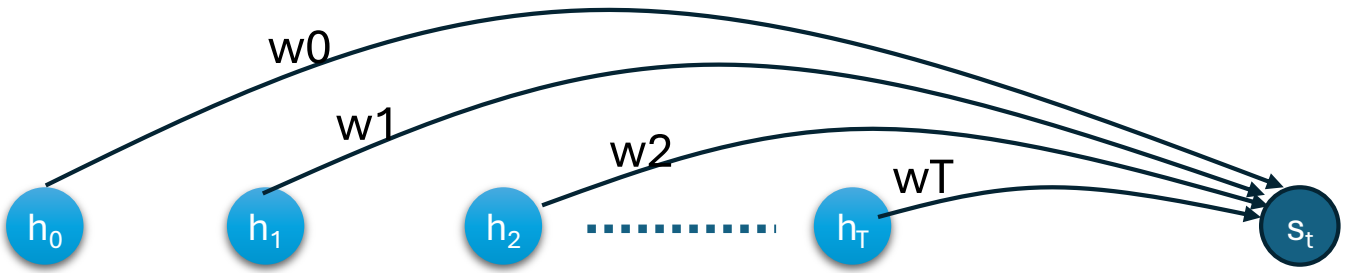
- sentence → (encoder) → fixed-size vector → (decoder) oracion
- problem: long sentence compressed into small vector
  - long sentence can lose information
- Solution: The decoder, at each output step, look at all encoder states and decide which ones matter most.



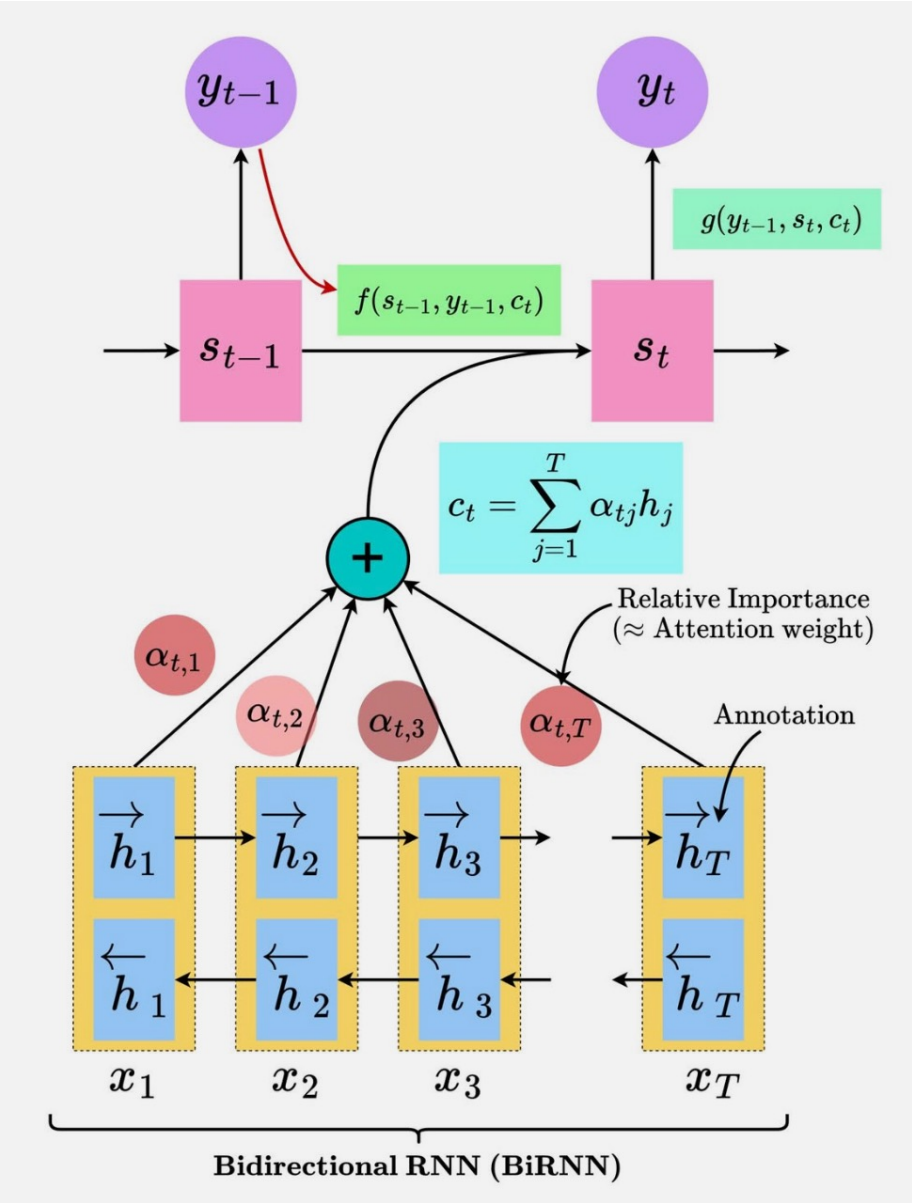
Encoder-Decoder-Cross-Attention

- Determine which encoder hidden states affects the output of the current hidden state?
  - Find such weights
  - decoder attends to encoder states.

context c_t = weighted sum $w_i h_i$
$Output_t = f(s_t, c_t)$



Architecture



Decoder: GRU

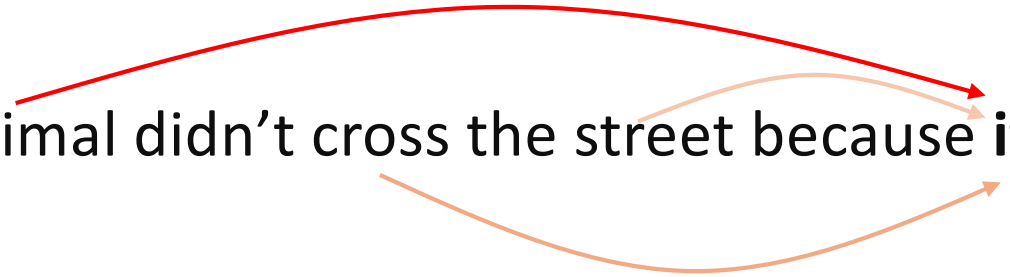
Alignment model

Encoder: BiRNN

$$\overrightarrow{h}_j = \mathrm{RNN}_{\mathrm{fwd}}(\overrightarrow{h}_{j-1}, x_j)$$

$$\overleftarrow{h}_j = \mathrm{RNN}_{\mathrm{bwd}}(\overleftarrow{h}_{j+1}, x_j)$$

Self-Attention



- The animal didn’t cross the street because **it** was too tired.

- Each token looks at all other tokens in the same sequence and decides which ones matter.

Attention Is All You Need (Vaswani, 2017)

- We don't need recurrence (RNN/LSTM) at all. Pure attention mechanisms are sufficient for sequence modeling.
- That architecture is called the Transformer.
- self-attention is the core operation, and the Transformer is the full architecture built around it.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

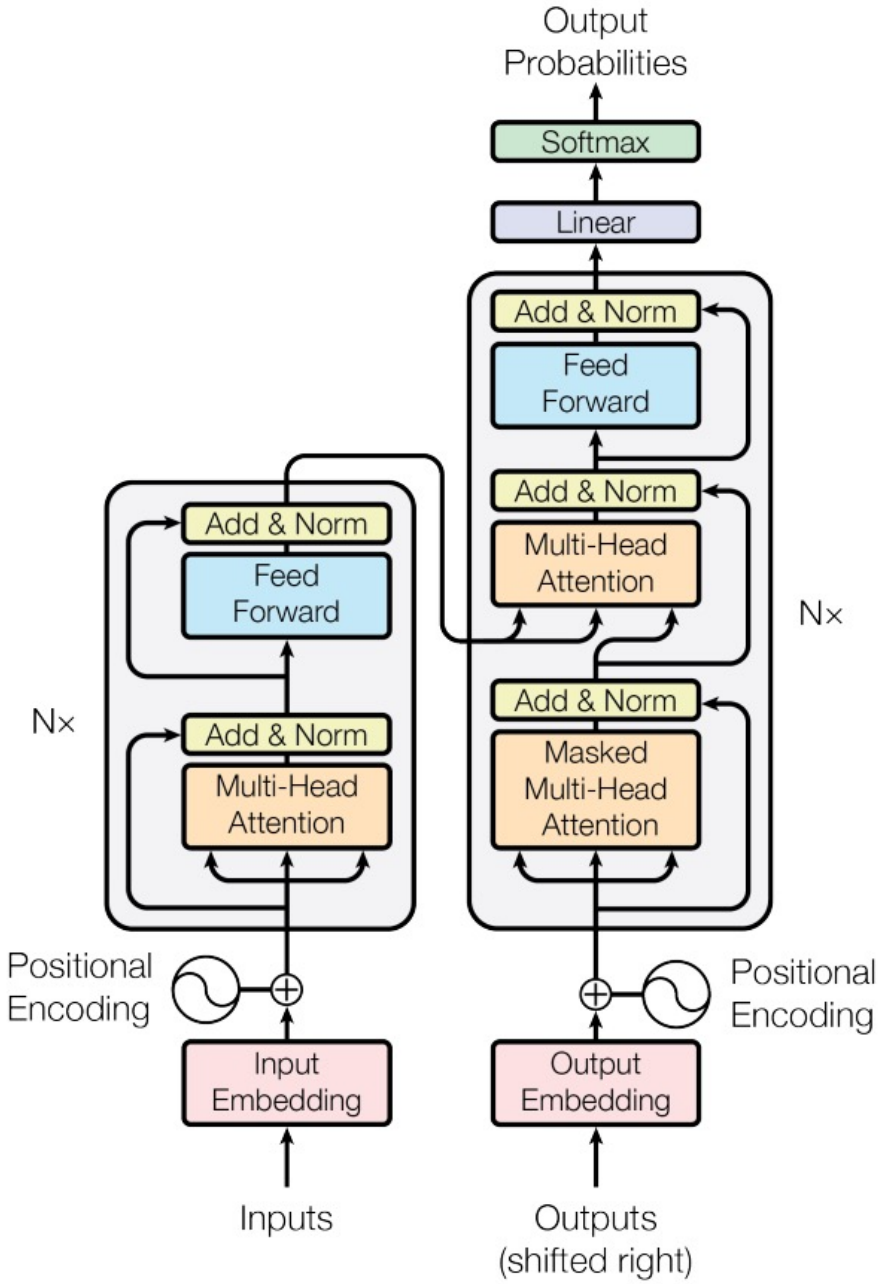


Figure 1: The Transformer - model architecture.