From Logistic Regression to (Recurring) Neural Networks

Thiemo Fetzer

University of Warwick & University of Bonn & CEPR & LSE & AEAI

May 8, 2025

Plan

Structuring the unstructured

Logistic Regression as a Neural Network

Training a Simple Neural Network

From Neural Networks to Recurring Neural Networks

Applied Machine Learning

- Key focus area of applied machine learning and economics is to structure data
- Control versus Chaos
- Typical tasks in any form of AI or machine learning:
  - Generate output conditional on context – *prediction*
    we have seen N-gram language model
  - Structuring large data conditional or unconditional on input
    Clustering and complexity reduction (e.g. topic modeling)
  - Retrieve (relevant) information
    Embeddings powerful in enabling semantic search

$\rightarrow$ large language modeling may be able to carry out all three

Plan

Structuring the unstructured

Logistic Regression as a Neural Network

Training a Simple Neural Network

From Neural Networks to Recurring Neural Networks

Recap Logistic Regression: A Generative Likelihood Model

We model the conditional probability of binary labels:

$$P(y_i = 1 \mid \mathbf{x}_i) = \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})$$

Assuming a Bernoulli likelihood:

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[\sigma(\mathbf{x}_i^\top \boldsymbol{\beta})\right]^{y_i} \left[1 - \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})\right]^{1-y_i}$$

We maximize the log-likelihood:

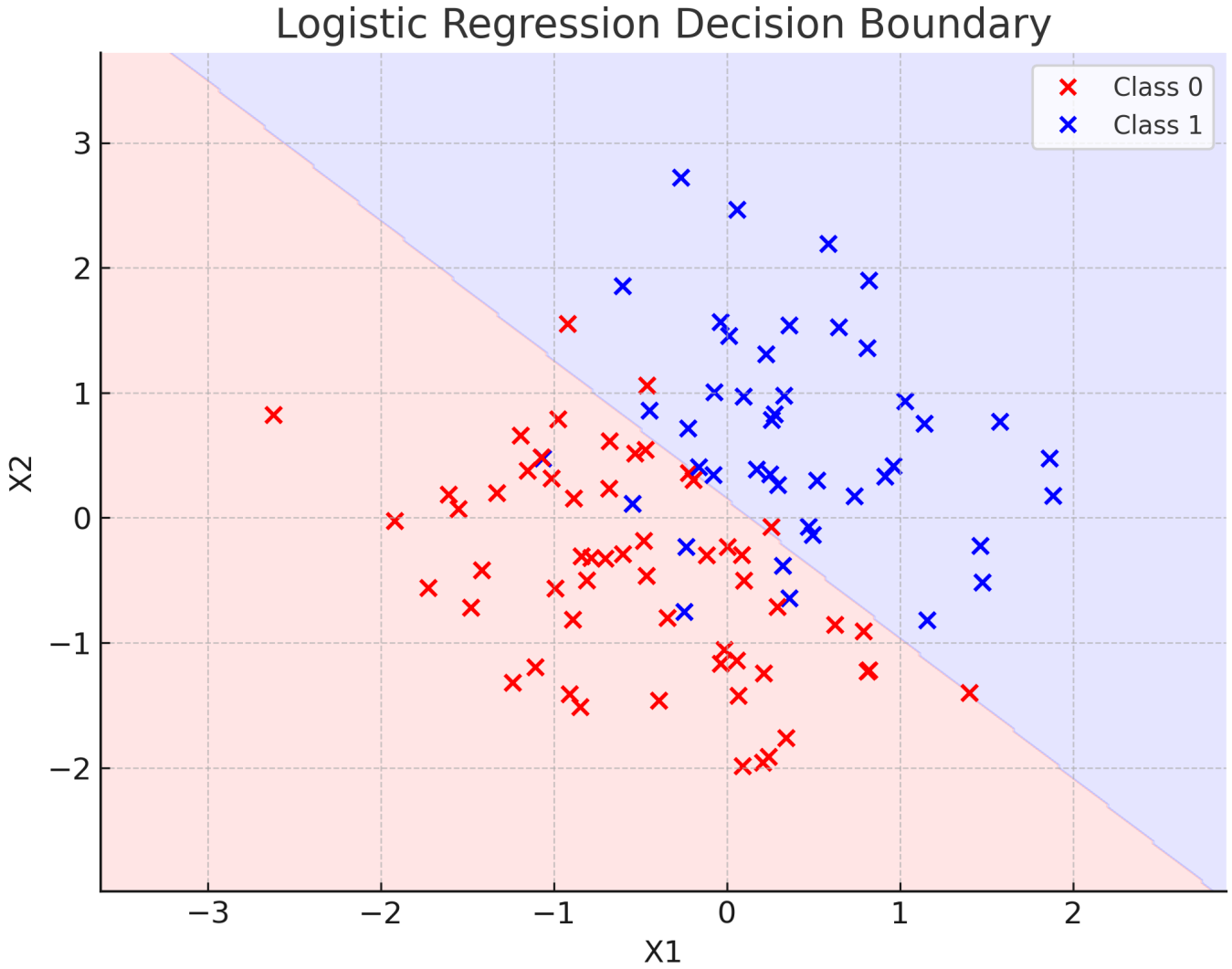
$$\log \mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \log \sigma(\mathbf{x}_i^\top \boldsymbol{\beta}) + (1 - y_i) \log(1 - \sigma(\mathbf{x}_i^\top \boldsymbol{\beta}))$$

This is equivalent to minimizing the **binary cross-entropy loss**:

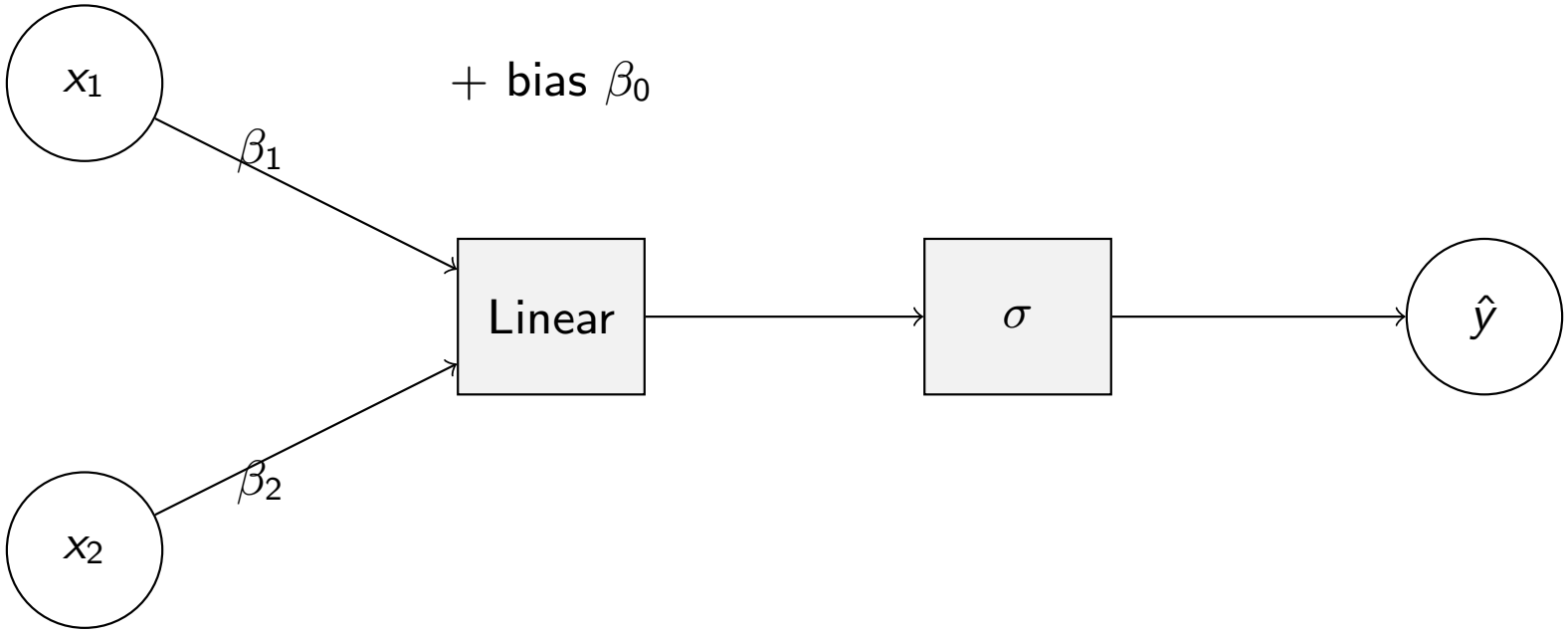
$$\mathcal{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log \sigma(\mathbf{x}_i^\top \boldsymbol{\beta}) + (1 - y_i) \log \left(1 - \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})\right)\right]$$

Logistic Regression Fitting

A typical logistic regression decision boundary that is learned e.g. via MLE



Visualizing Logistic Regression as a Neural Network



- General form:

$$\hat{y} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2)$$

- Example (fitted):

$$\hat{y} = \sigma(-0.82 + 1.10 x_1 + 0.85 x_2)$$

- Logistic regression is equivalent to a single-layer neural network.

What about the σ function?

A linear probability model has:

$$\sigma = \mathbf{x}_i^\top \boldsymbol{\beta}$$

But this can produce fitted probabilities $P(y_i = 1 \mid \mathbf{x}_i)$ that are negative or above 1.
A common replacement is a sigmoid or *softmax* representation. The sigmoid function maps real numbers into the interval $(0, 1)$:

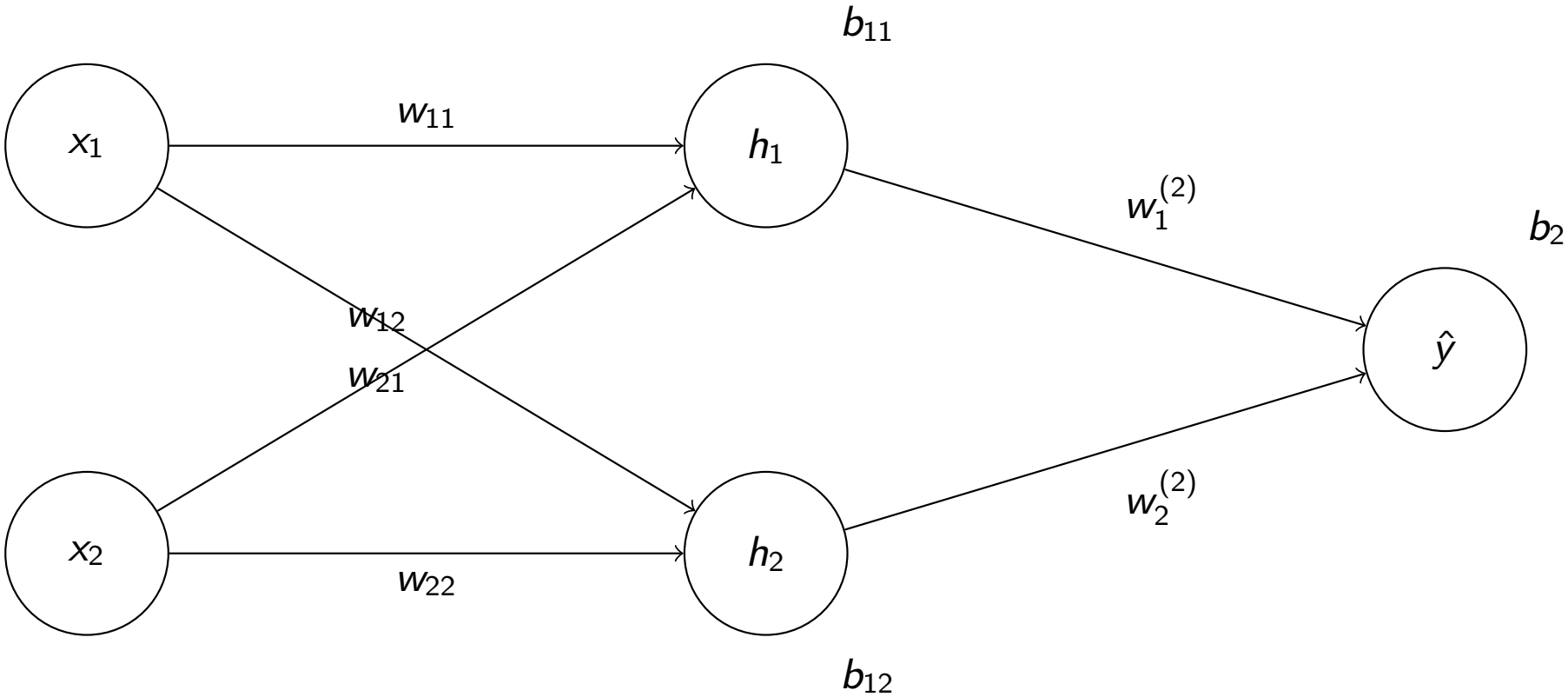
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Applied to our model:

$$P(y_i = 1 \mid \mathbf{x}_i) = \frac{1}{1 + \exp(-\mathbf{x}_i^\top \boldsymbol{\beta})}$$

Plan

Example Neural Network with One Hidden Layer With Two Neurons



Each edge is labeled with its corresponding weight symbol. Biases are annotated next to each layer.

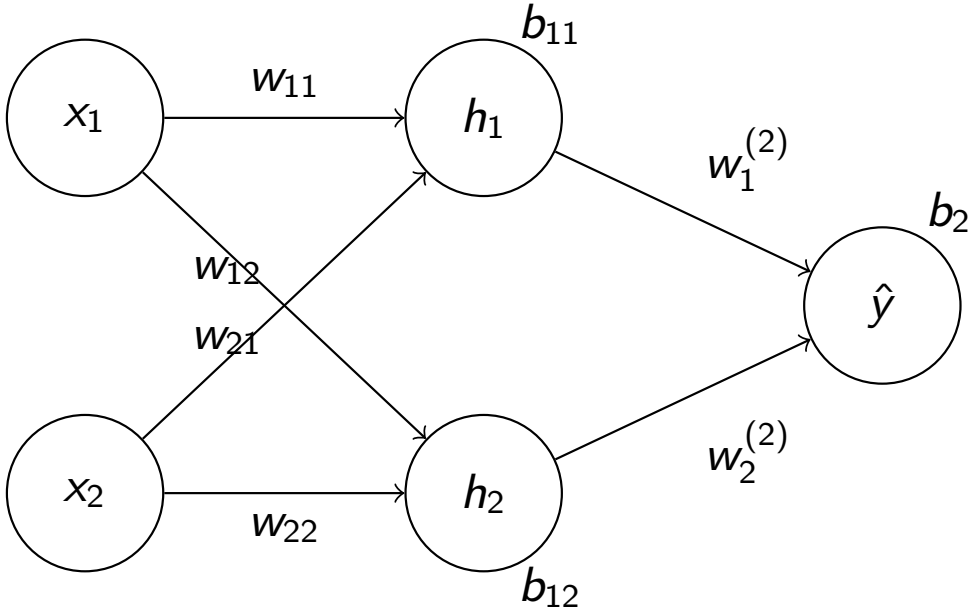
An Example Neural Network

Model Definition:

$$\hat{y} = \sigma\left(\mathbf{W}_2^\top \cdot \mathrm{f}(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + b_2\right)$$

Parameters:

- $\mathbf{W}_1 \in \mathbb{R}^{2\times 2}$ input hidden weights
- $\mathbf{b}_1 \in \mathbb{R}^2$: biases hidden neurons
- $\mathbf{W}_2 \in \mathbb{R}^{2\times 1}$ hidden output weights
- $b_2 \in \mathbb{R}$: output bias



Each edge is labeled with its corresponding parameter. Biases are shown next to the hidden/output layers.

Forward Pass:

$$\mathbf{z}_1 = \mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h}_1 = \mathrm{f}(\mathbf{z}_1)$$
$$z_2 = \mathbf{W}_2^\top \cdot \mathbf{h}_1 + b_2$$
$$\hat{y} = \sigma(z_2)$$

Forward Pass Equations Spelled Out

Hidden Layer (Pre-activations):

$$z_1 = w_{11}x_1 + w_{21}x_2 + b_{11}$$
$$z_2 = w_{12}x_1 + w_{22}x_2 + b_{12}$$

Hidden Layer (Activations):

$$h_1 = f(z_1)$$
$$h_2 = f(z_2)$$

Output Layer (Pre-activation):

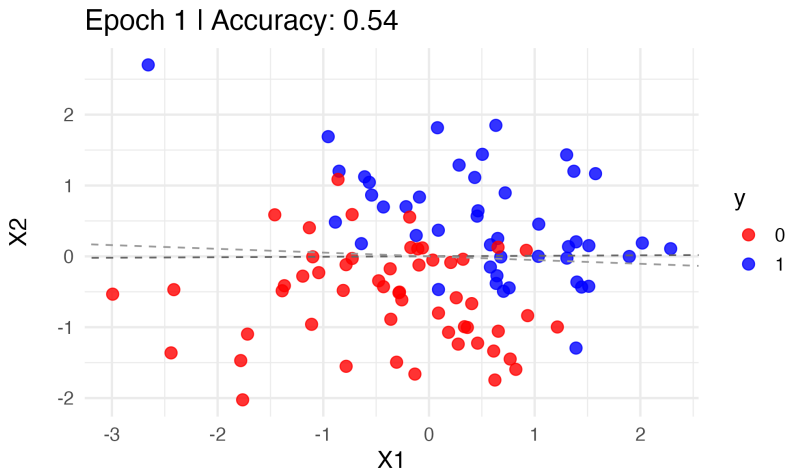
$$z_{\text{out}} = w_1^{(2)} h_1 + w_2^{(2)} h_2 + b_2$$

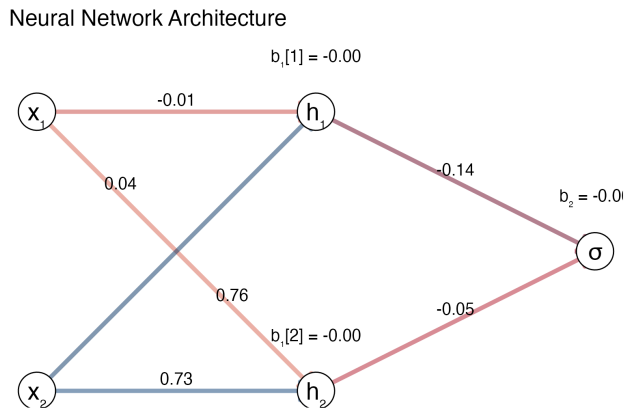
Predicted Output:

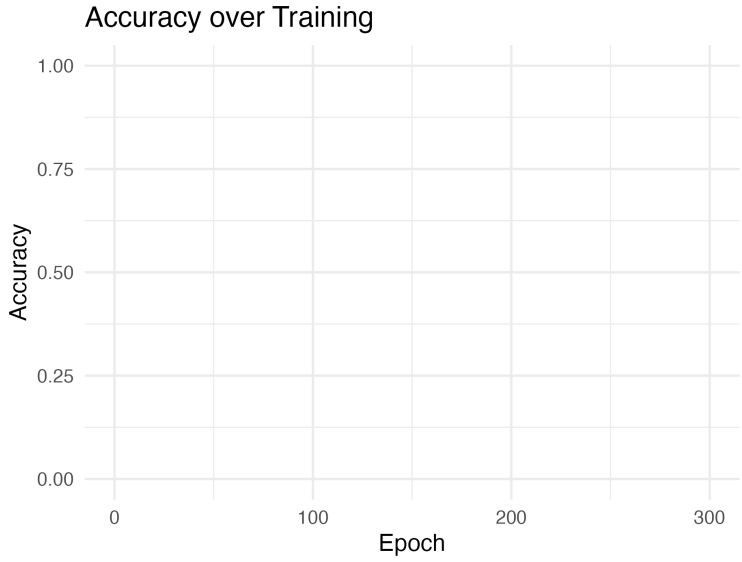
$$\hat{y} = \psi(z_{\text{out}})$$

Where $f(\cdot)$ is the activation function (e.g. ReLU or sigmoid), and $\psi(\cdot)$ is the output activation (e.g. identity or sigmoid).

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

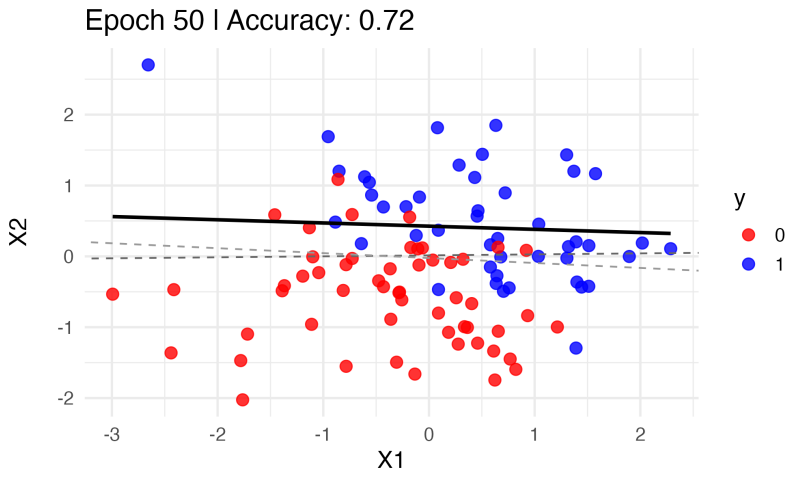
$$W_1 = [[-0.01, 0.04], [0.76, 0.73]]$$

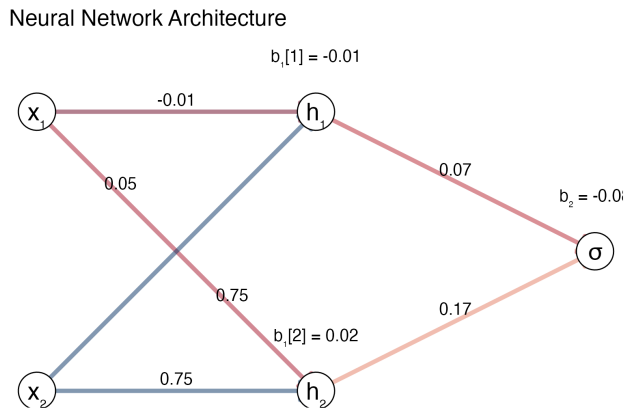
$$b_1 = [-0.00, -0.00]$$

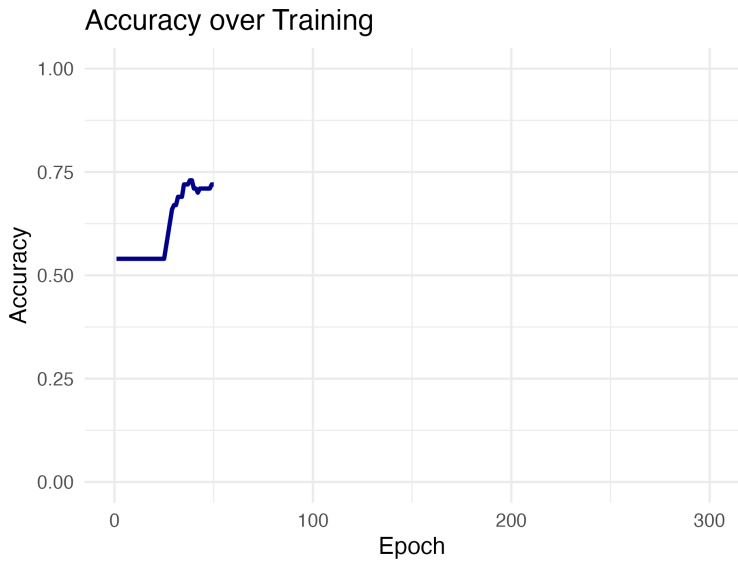
$$W_2 = [-0.14, -0.05]$$

$$b_2 = -0.00$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

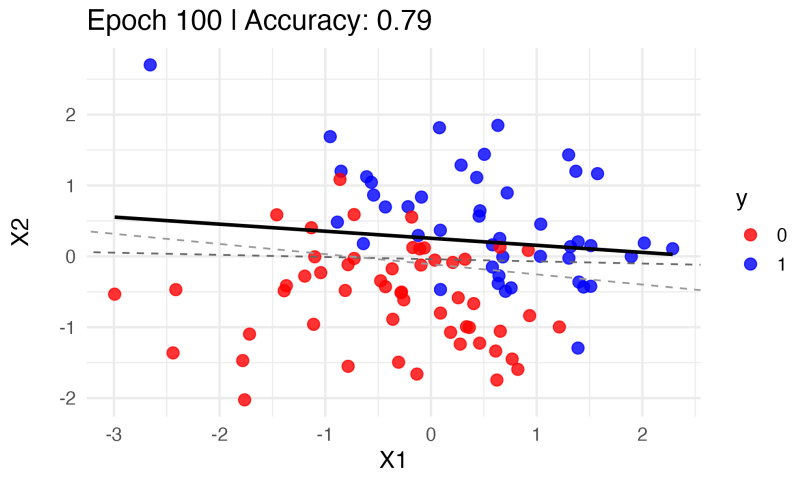
$$W_1 = [[-0.01, 0.05], [0.75, 0.75]]$$

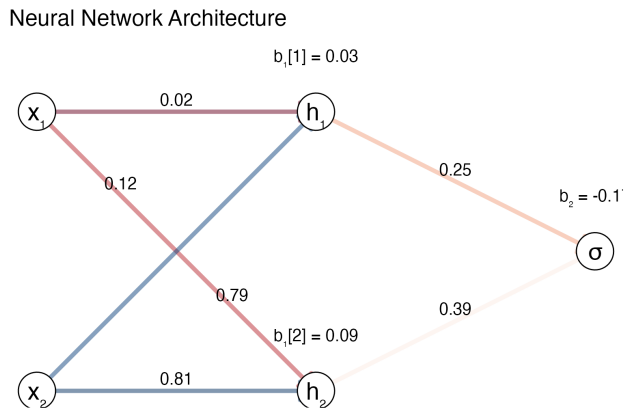
$$b_1 = [-0.01, 0.02]$$

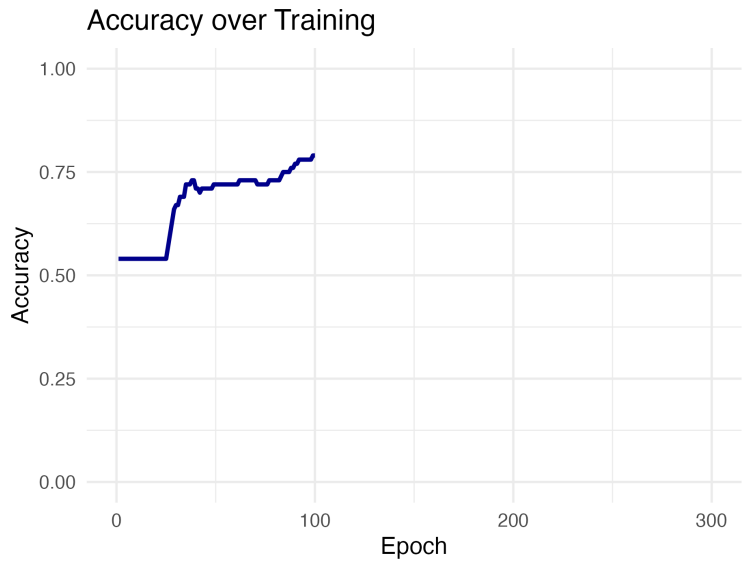
$$W_2 = [0.07, 0.17]$$

$$b_2 = -0.08$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

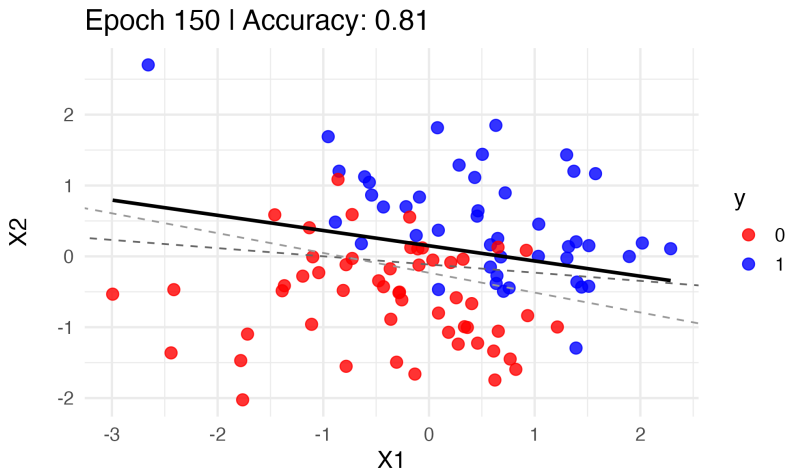
$$W_1 = [[0.02, 0.12], [0.79, 0.81]]$$

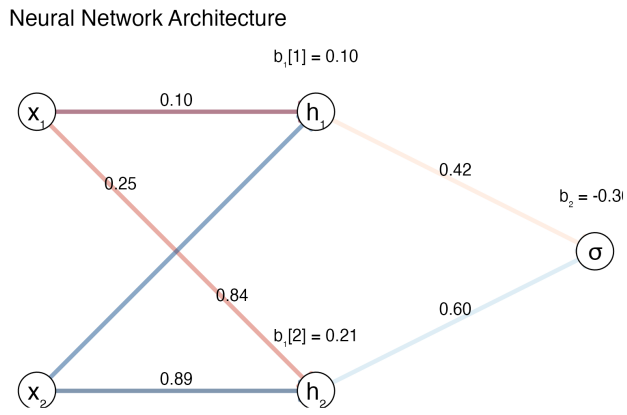
$$b_1 = [0.03, 0.09]$$

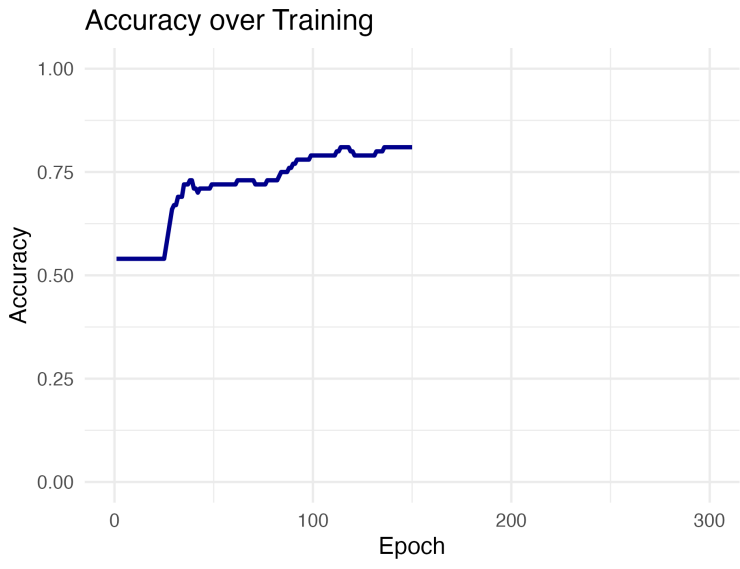
$$W_2 = [0.25, 0.39]$$

$$b_2 = -0.17$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

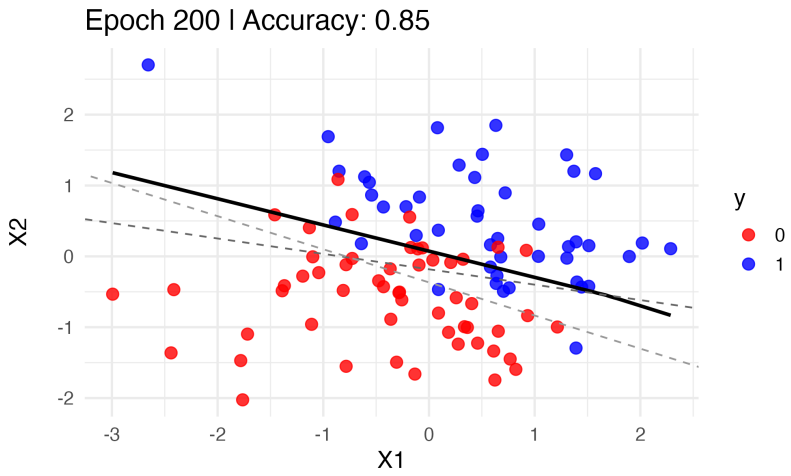
$$W_1 = [[0.10, 0.25], [0.84, 0.89]]$$

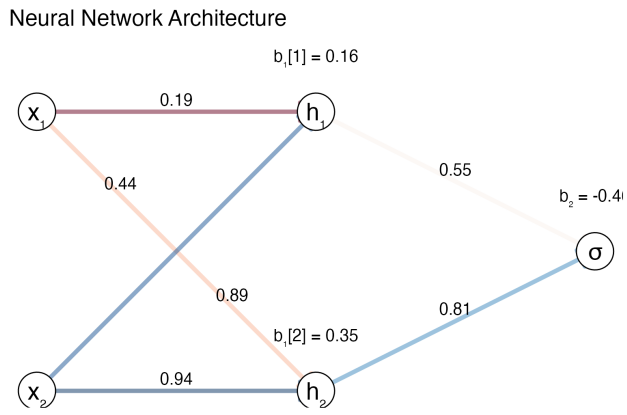
$$b_1 = [0.10, 0.21]$$

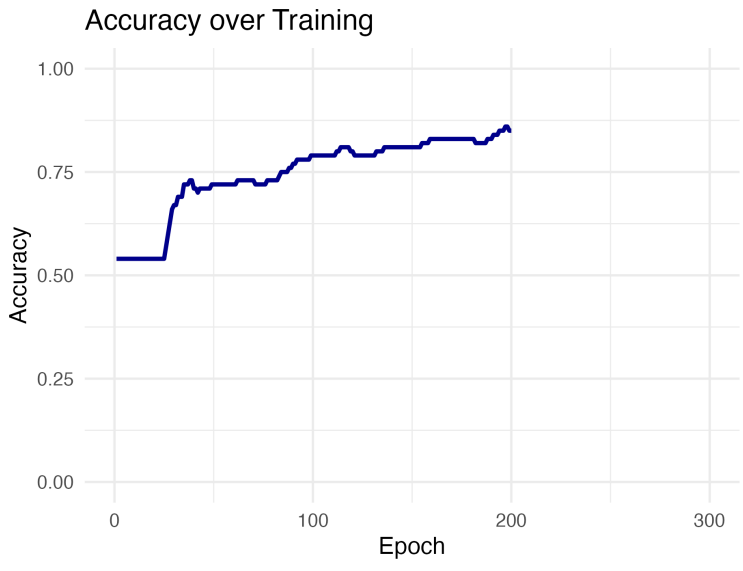
$$W_2 = [0.42, 0.60]$$

$$b_2 = -0.30$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

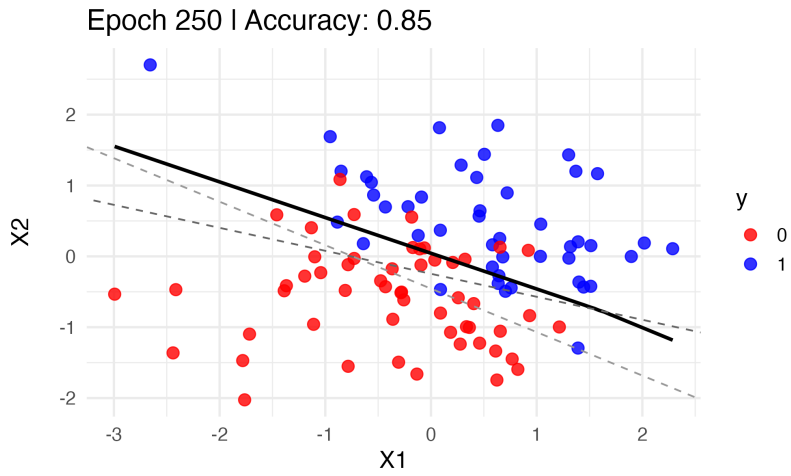
$$W_1 = [[0.19, 0.44], [0.89, 0.94]]$$

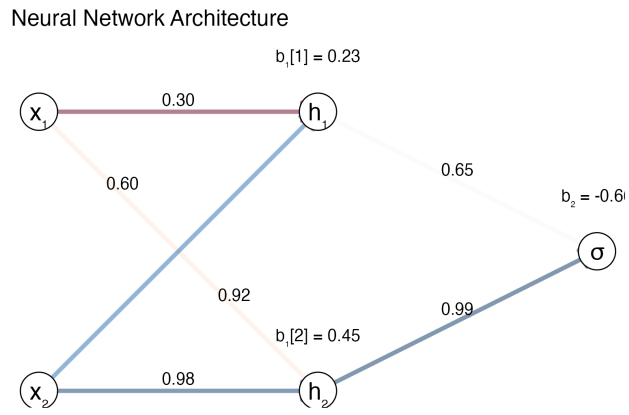
$$b_1 = [0.16, 0.35]$$

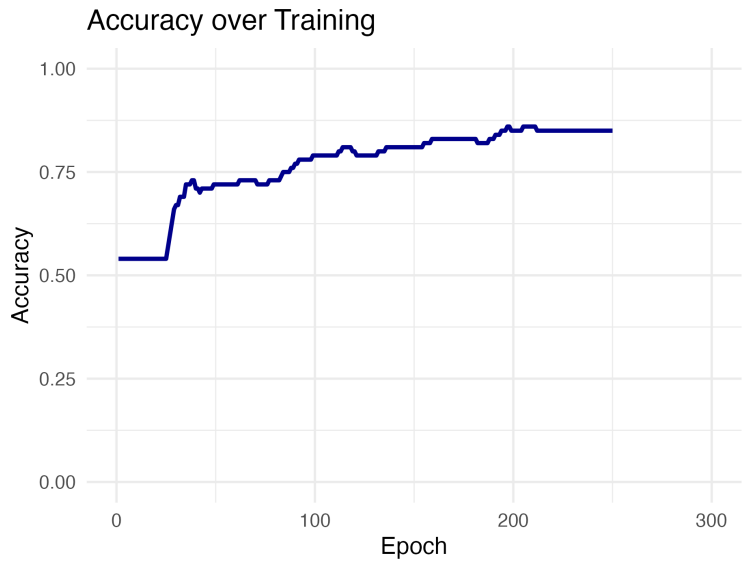
$$W_2 = [0.55, 0.81]$$

$$b_2 = -0.46$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

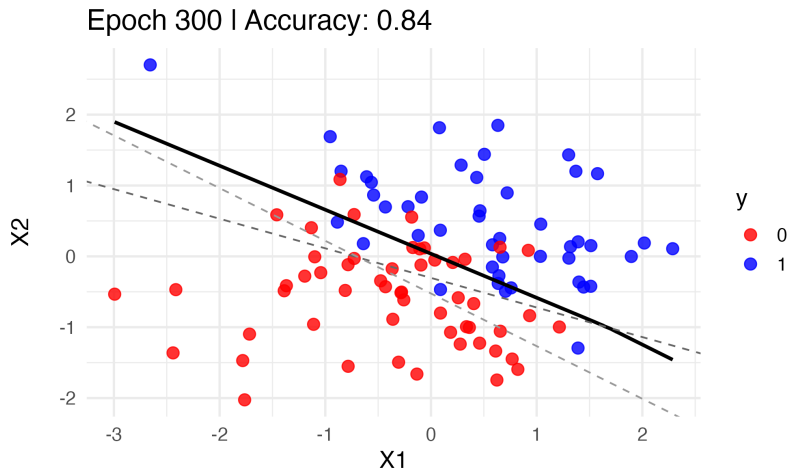
$$W_1 = [[0.30, 0.60], [0.92, 0.98]]$$

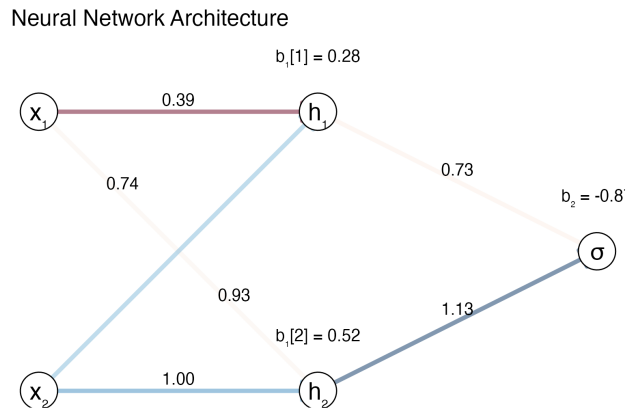
$$b_1 = [0.23, 0.45]$$

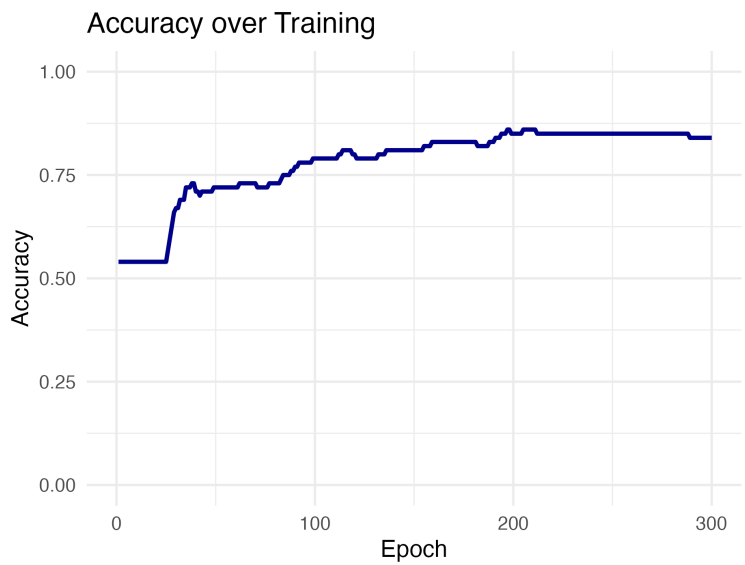
$$W_2 = [0.65, 0.99]$$

$$b_2 = -0.66$$

Neural Network Training: Gradient Descent in Action







$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot X + b_1) + b_2)$$

$$W_1 = [[0.39, 0.74], [0.93, 1.00]]$$

$$b_1 = [0.28, 0.52]$$

$$W_2 = [0.73, 1.13]$$

$$b_2 = -0.87$$

Forward Pass - Prediction Step

In each epoch, the forward pass takes the current weights and biases and produces predictions for all input samples.

For each input example $\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$, the neural network computes:

$$\mathbf{z}_1^{(i)} = \mathbf{W}_1 \cdot \mathbf{x}^{(i)} + \mathbf{b}_1 \qquad \text{(shape: } 2 \times 1\text{) - linear combination}$$

$$\mathbf{h}_1^{(i)} = \mathrm{f}(\mathbf{z}_1^{(i)}) \qquad \text{(hidden activation)}$$

$$z_2^{(i)} = \mathbf{W}_2^{\top} \cdot \mathbf{h}_1^{(i)} + b_2 \qquad \text{(shape: scalar)}$$

$$\hat{y}^{(i)} = \sigma(z_2^{(i)}) \qquad \text{(predicted probability)}$$

Where $\mathbf{x}^{(i)} \in \mathbb{R}^{2\times 1}$, $\mathbf{W}_1 \in \mathbb{R}^{2\times 2}$, $\mathbf{b}_1 \in \mathbb{R}^{2\times 1}$, $\mathbf{W}_2 \in \mathbb{R}^{2\times 1}$, $b_2 \in \mathbb{R}$

Intuition: the forward pass is applying a recipe: the ingredients (inputs and parameters) are used to produce a prediction (the output) for each input. This prediction will later be compared to the true label during the backward pass.

Backward Pass - How the Network Learns

The backward pass computes how each weight and bias contributed to the prediction error, and how to adjust them to reduce future errors.

1. **Loss Function (per observation):**

$$\mathcal{L}^{(i)} = -\left[y^{(i)} \log(\sigma(z_2^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z_2^{(i)}))\right]$$

2. **Total Loss (Objective Function):**

$$\mathcal{L}_{\text{total}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}^{(i)}$$

Loss is computed for each observation, but the gradient is taken with respect to the mean (or sum) across all data.

3. **Core Idea:** Use **chain rule** to compute gradients of the loss vis-a-vis all parameters: weights and biases in both hidden and output layers.

Backpropagation: This is the systematic application of the chain rule, layer by layer in reverse, to compute gradients efficiently.

Gradient Computation via Chain Rule for one $(\mathbf{x}^{(i)}, y^{(i)})$

Output layer:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial z_2^{(i)}} = \hat{y}^{(i)} - y^{(i)} \quad \text{(simplified from sigmoid + cross-entropy)}$$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{W}_2} = (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{h}_1^{(i)} \qquad \frac{\partial \mathcal{L}^{(i)}}{\partial b_2} = \hat{y}^{(i)} - y^{(i)}$$

Backpropagation into hidden layer:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}_1^{(i)}} = (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{W}_2$$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{z}_1^{(i)}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}_1^{(i)}} \circ \mathrm{f}'(\mathbf{z}_1^{(i)})$$

Weight Update Rule (Gradient Descent)

At each step:

$$\mathbf{W}_1 \leftarrow \mathbf{W}_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$$

$$\mathbf{b}_1 \leftarrow \mathbf{b}_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1}$$

$$\mathbf{W}_2 \leftarrow \mathbf{W}_2 - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}$$

$$b_2 \leftarrow b_2 - \eta \cdot \frac{\partial \mathcal{L}}{\partial b_2}$$

$\rightarrow$ here is where the learning rate η comes in (check out the R code)

Our Simple Single Layer Neural Network with 2-Input, 2-Hidden Neurons Network

Parameter	Label	Gradient Expression	Plain English Explanation
Output pre-activation	z_{out}	$\frac{\partial \mathcal{L}}{\partial z_{\text{out}}} = \hat{y} - y$	Error term from output (prediction minus target)
Output weights	$w_1^{(2)}, w_2^{(2)}$	$\frac{\partial \mathcal{L}}{\partial w_j^{(2)}} = (\hat{y} - y) \cdot h_j$	Gradient w.r.t. weight from hidden unit h_j to output
Output bias	b_2	$\frac{\partial \mathcal{L}}{\partial b_2} = \hat{y} - y$	Same as the output error term
Hidden pre-activations	z_1, z_2	$\frac{\partial \mathcal{L}}{\partial z_j} = (\hat{y} - y) \cdot w_j^{(2)} \cdot f'(z_j)$	Backpropagate output error to each hidden neuron via chain rule
Hidden weights	w_{ij}	$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial z_j} \cdot x_i$	Gradient of loss w.r.t. weights from input x_i to hidden unit j
Hidden biases	b_{11}, b_{12}	$\frac{\partial \mathcal{L}}{\partial b_1[j]} = \frac{\partial \mathcal{L}}{\partial z_j}$	Same as hidden neuron error term

Notes:

- $\hat{y} = \sigma(z_{\text{out}})$ is the predicted probability.
- $f'(z_j)$ is the derivative of the hidden layer activation.
- $w_j^{(2)}$ is the weight from hidden neuron j to output.
- Chain rule is applied to backpropagate error from output to inputs.

What is f and Why Do We Need It?

In neural networks, **nonlinear activation functions** are essential for modeling complex patterns. One of the most widely used is the Rectified

Linear Unit activation:

$$\mathrm{f}(z) = \max(0, z)$$

That is:

- If $z > 0$, output is z
- If $z \leq 0$, output is 0

This is explicitly non-linear function.

Why Do We Need Activation Functions f?

Without a non-linear functions like f, a neural network with multiple layers reduces to a **linear transformation**:

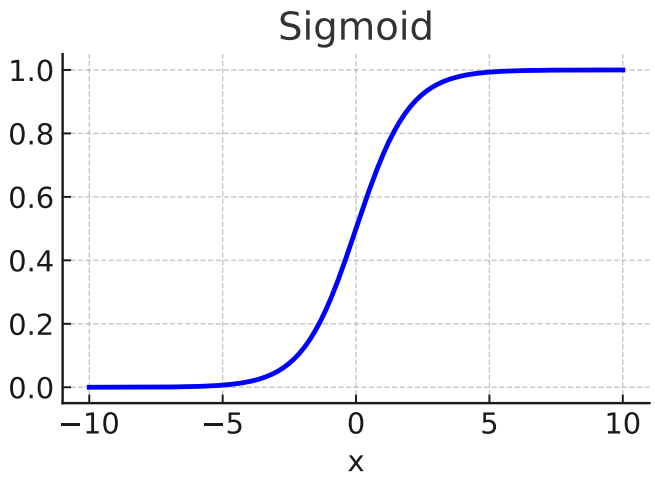
$$\sigma(\mathbf{W}_2 \cdot (\mathbf{W}_1 \cdot \mathbf{x})) = \sigma(\mathbf{W}_{\text{combined}} \cdot \mathbf{x})$$

Since this is just a matrix multiplication of 2 x 2 matrix W_1 with a 2 x 1 matrix W_2 yielding a 2 x 1 vector.

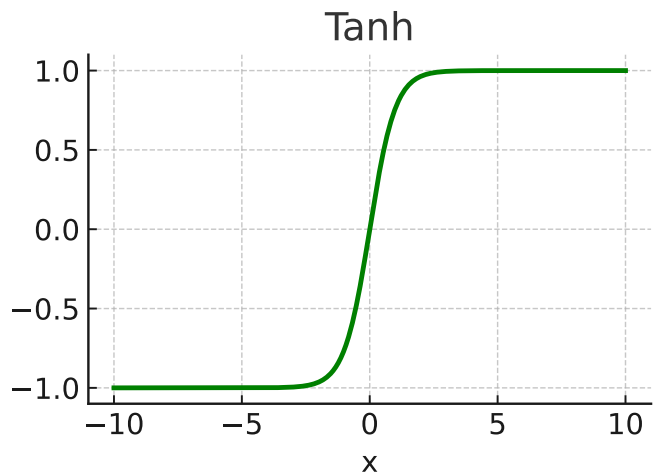
This is functionally the same as logistic regression!

$\rightarrow$ f introduces **non-linearity**, enabling the network to **approximate complex decision boundaries**.

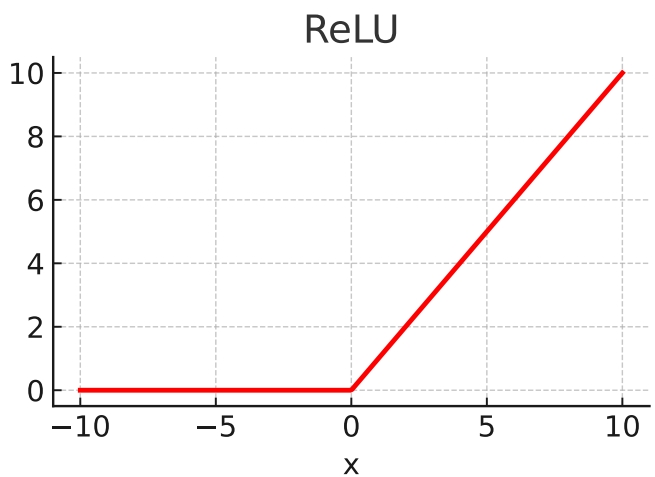
Alternative Activation Functions



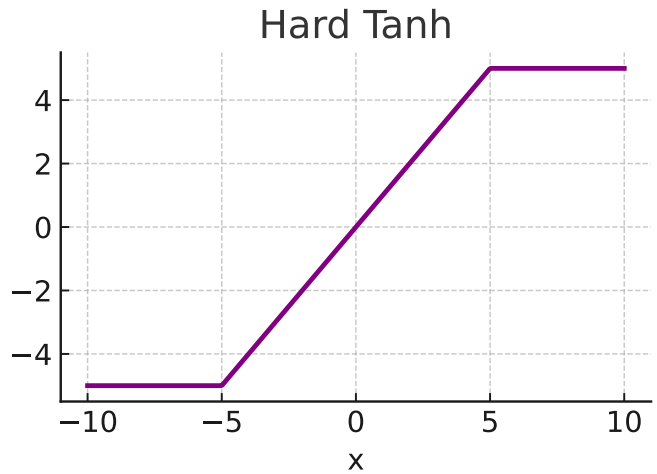
Sigmoid
Smooth S-curve; saturates at 0 and 1.



Tanh
Zero-centered; maps to $(-1, 1)$.



ReLU
Zero for $x < 0$; linear for $x > 0$.



Hard Tanh
Linearly bounded between -5 and $+5$.

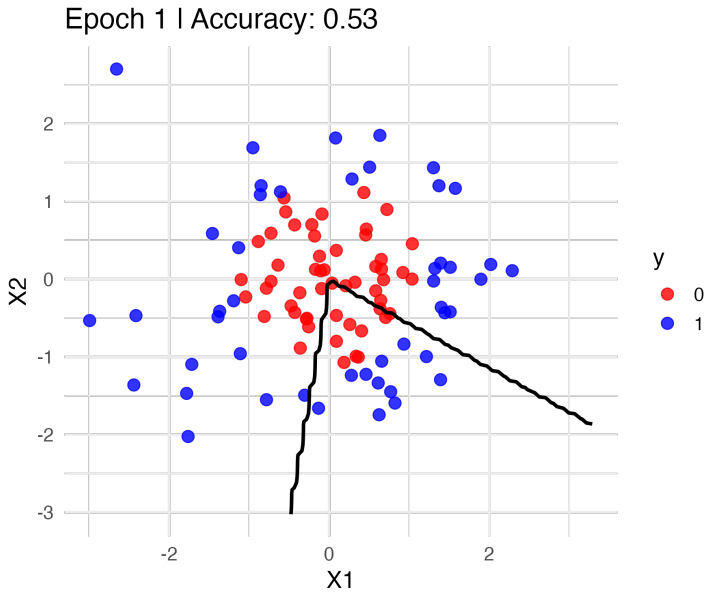
Why do we like RELU or hard tanh?

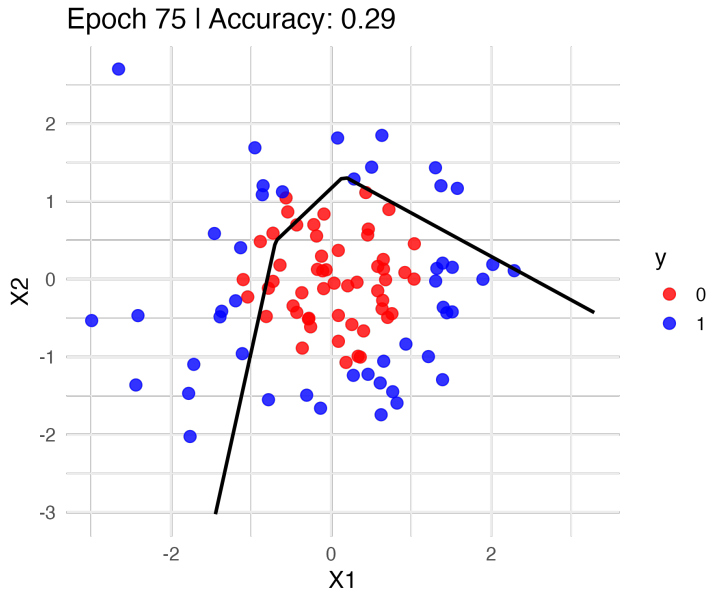
$\rightarrow$ strong preference for RELU and Hard TanH, why?

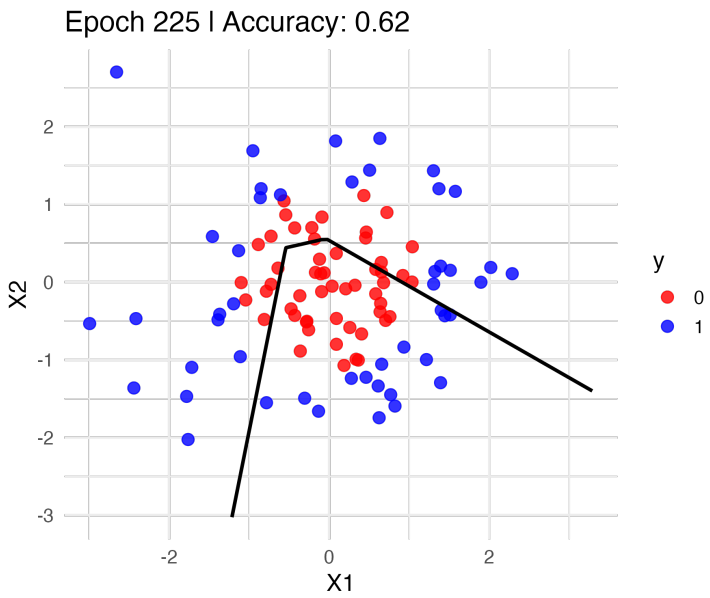
- **Simple**: Fast to compute, easy to differentiate
- **Sparse activation**: Most neurons output zero - makes networks efficient
- **Avoids vanishing gradients** (compared to sigmoid or tanh)

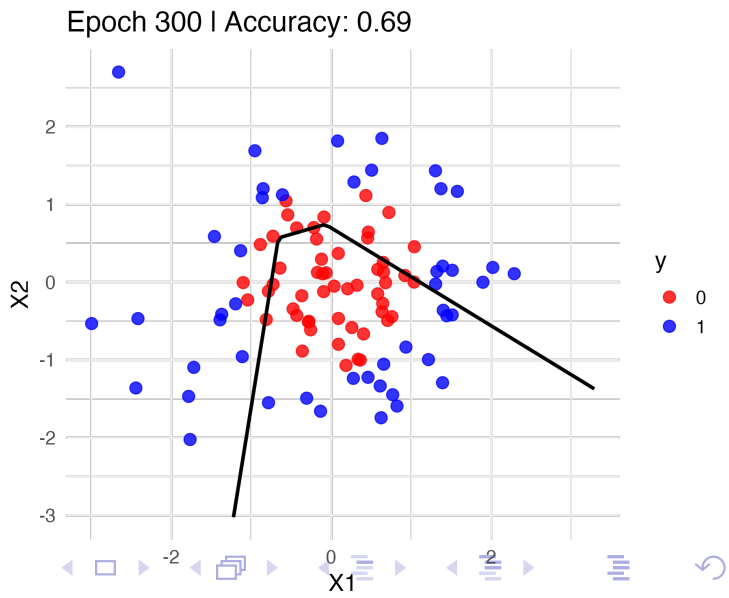
Each f "unit" acts like a **hinge** or **kink** that bends the decision surface. That is why RELU is often also referred to as "hinge Loss".
Using multiple f neurons in a layer allows a network to **combine piecewise linear functions** - forming nonlinear, expressive mappings from inputs to outputs.

Example with non-linearly separable data









Plan

Structuring the unstructured

Logistic Regression as a Neural Network

Training a Simple Neural Network

From Neural Networks to Recurring Neural Networks

What We Have Looked at So Far?

- We started from logistic regression:

$$\hat{y} = \sigma(\mathbf{x}^\top \boldsymbol{\beta})$$

- Then generalized to a single-layer neural network:

$$\hat{y} = \sigma(W_2 \cdot \mathrm{f}(W_1 \mathbf{x} + b_1) + b_2)$$

- This allows us to approximate non-linear functions from inputs to predictions.
- But: All of this assumes the input $\mathbf{x}$ is a **fixed-size vector**.
- What if $\mathbf{x}$ is a sequence?

A sequence in language modeling terms? N-gram Models

- N-gram language models predict the next word based on the last $n-1$:

$$P(w_t \mid w_{t-1}, w_{t-2}, \ldots, w_{t-n+1})$$

- They work well when:
  - We have lots of data to count observed sequences.
  - Dependencies are short-range.
- But they struggle with:
  - Long-term context
  - Unseen combinations
  - Data sparsity

Why Do We Need Recurrent Neural Networks?

Context matters. Just like an **n-gram model** captures context to predict the next word, we want a model that:

- Keeps track of what it has seen so far
- Uses this memory to make future predictions

Recurrent Neural Networks do this by maintaining a hidden state:

$$\boxed{h_t = \tanh(W_x x_t + W_h h_{t-1} + b)}$$

$$\boxed{\hat{y}_t = \text{softmax}(W_y h_t)}$$

- *Same weights shared across all time steps*:

$$W_x, W_h, W_y \quad \text{do not change over } t$$

- This allows generalization to **variable-length input sequences**
- RNNs introduce **time** into the neural network computation graph

Worked Example: Step-by-Step RNN

Let's walk through an RNN processing a sequence of 3 word vectors.

Initial weights:

$$W_x = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}, \quad W_h = \begin{bmatrix} 0.4 & 0.1 \\ 0.1 & 0.5 \end{bmatrix}, \quad h_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Input sequence:

$$x_1 = [1, 0], \quad x_2 = [0, 1], \quad x_3 = [1, 1]$$

At time step 1:

$$h_1 = \tanh(W_x x_1 + W_h h_0) = \tanh\left(\begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix}\right) \approx [0.46, 0.66]$$

You can repeat this for h_2 and h_3, showing how memory evolves.

Step 1: Input "The"

- Input embedding: $x_1 = [1, 0]$
- Initial hidden state: $h_0 = [0, 0]$
- Compute:

$$z_1 = W_x x_1 + W_h h_0 + b = [0.5, 0.8] \Rightarrow h_1 = \tanh(z_1) = [0.46, 0.66]$$

- Predict:

$$\hat{y}_1 = \text{softmax}(W_y h_1) = [0.1, \mathbf{0.7}, 0.2]$$

- Model predicts: "cat"

Step 2: Input "cat"

- Input embedding: $x_2 = [0, 1]$
- Previous state: $h_1 = [0.46, 0.66]$
- Compute:

$$z_2 = W_x x_2 + W_h h_1 + b = [-0.07, 0.6] \Rightarrow h_2 = \tanh(z_2) = [-0.07, 0.54]$$

- Predict:

$$\hat{y}_2 = \text{softmax}(W_y h_2) = [0.05, 0.1, \mathbf{0.85}]$$

- Model predicts: "sat"

Step 3: Input "sat"

- Input embedding: $x_3 = [1, 1]$
- Previous state: $h_2 = [-0.07, 0.54]$
- Compute:

$$z_3 = W_x x_3 + W_h h_2 + b = [0.26, 1.26] \Rightarrow h_3 = \tanh(z_3) = [0.25, 0.85]$$

- Predict:

$$\hat{y}_3 = \text{softmax}(W_y h_3) = [0.1, 0.15, \mathbf{0.75}]$$

- Model predicts: "on"

Final Sequence Prediction

- The model has seen: "The cat sat"
- It generated the prediction sequence:

$$[\hat{y}_1, \hat{y}_2, \hat{y}_3] = \text{"cat"}, \text{"sat"}, \text{"on"}$$

- The final sequence probability:

$$P(w_1, w_2, w_3) = \hat{y}_1(\text{"cat"}) \cdot \hat{y}_2(\text{"sat"}) \cdot \hat{y}_3(\text{"on"})$$

- Each $\hat{y}_t$ is a distribution over vocabulary - we take the relevant entry

From Feedforward to Recurrent Thinking

- In feedforward networks:

$$\hat{y} = f(x) \qquad \text{(direct mapping)}$$

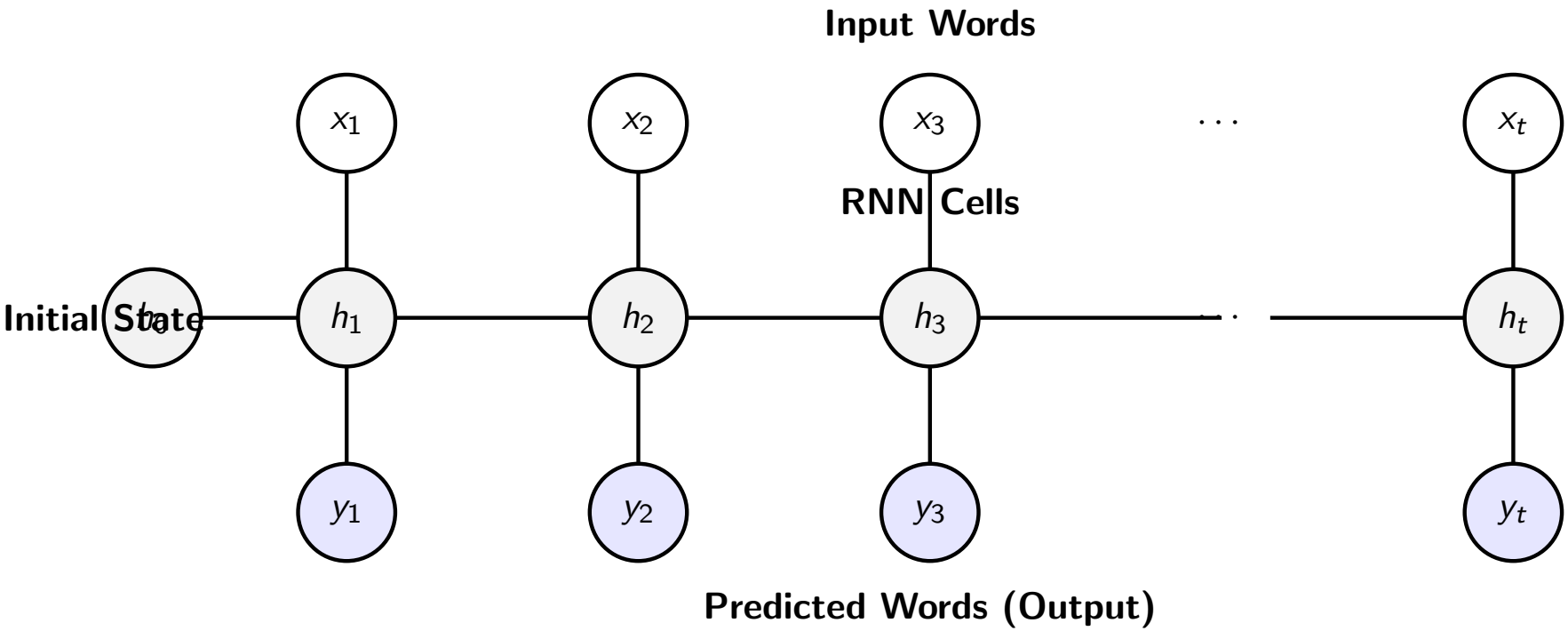
- In recurrent networks:

$$\hat{y}_t = f(x_1, x_2, \ldots, x_t) \qquad \text{(accumulated memory)}$$

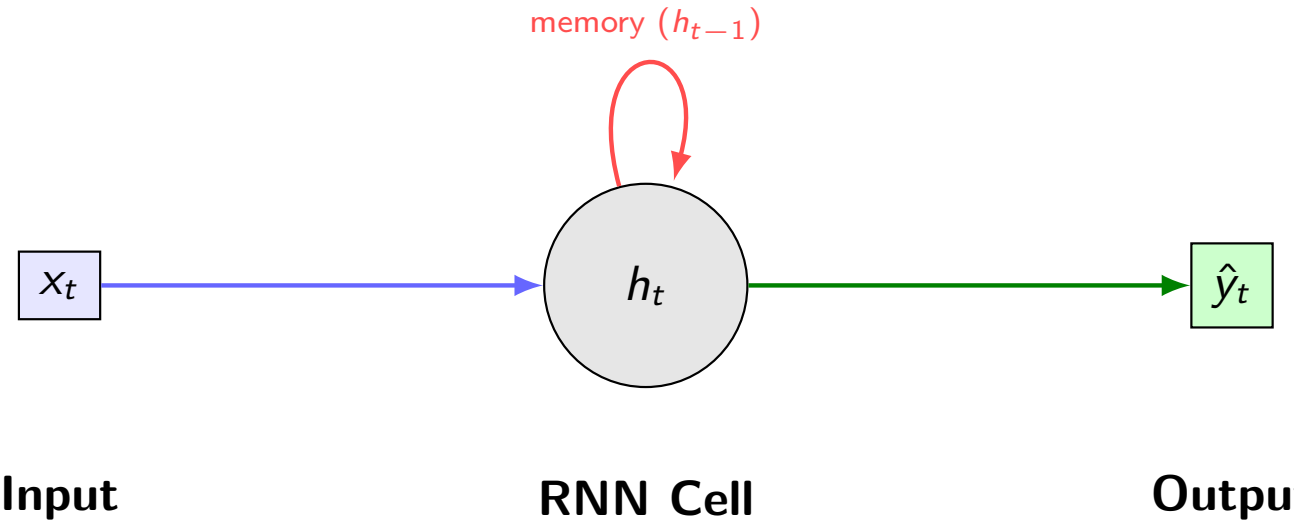
- Instead of memorizing sequences (like n-grams), we learn to represent and compress them into a hidden state.
- This opens the door to language models:

$$P(w_t \mid w_{<t}) \approx \text{softmax}(W_y h_t)$$

Unrolled RNN Architecture



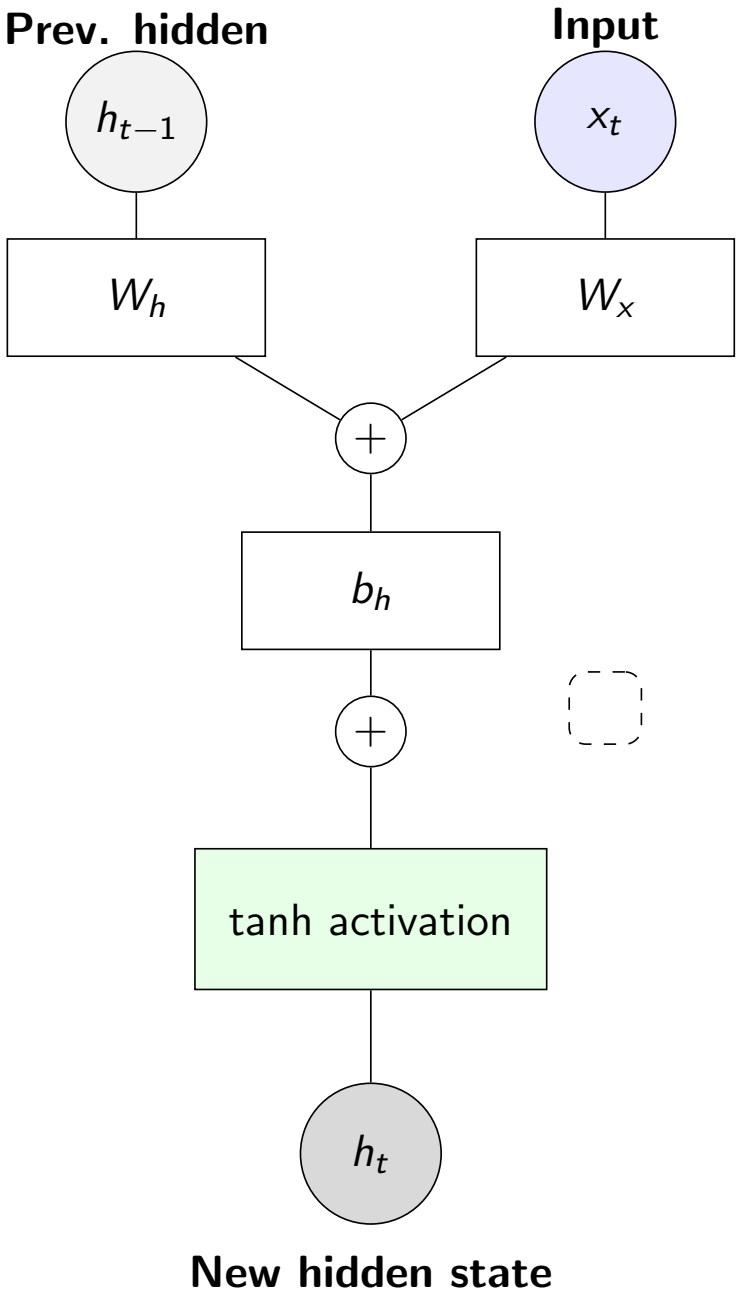
Rolled-Up View of an RNN



This compact representation shows that each RNN cell receives:

- A new input x_t
- A hidden state h_{t-1} from the previous time step
- It updates and returns a new state h_t, and an output $\hat{y}_t$

RNN Cell Computation



Computation Steps:

Input: $x_t \in \mathbb{R}^V$

Prev: $h_{t-1} \in \mathbb{R}^H$

Weights: $W_x \in \mathbb{R}^{H \times V}$,
$W_h \in \mathbb{R}^{H \times H}$,
$b_h \in \mathbb{R}^H$

Step 1: $W_x x_t$ (input projection)

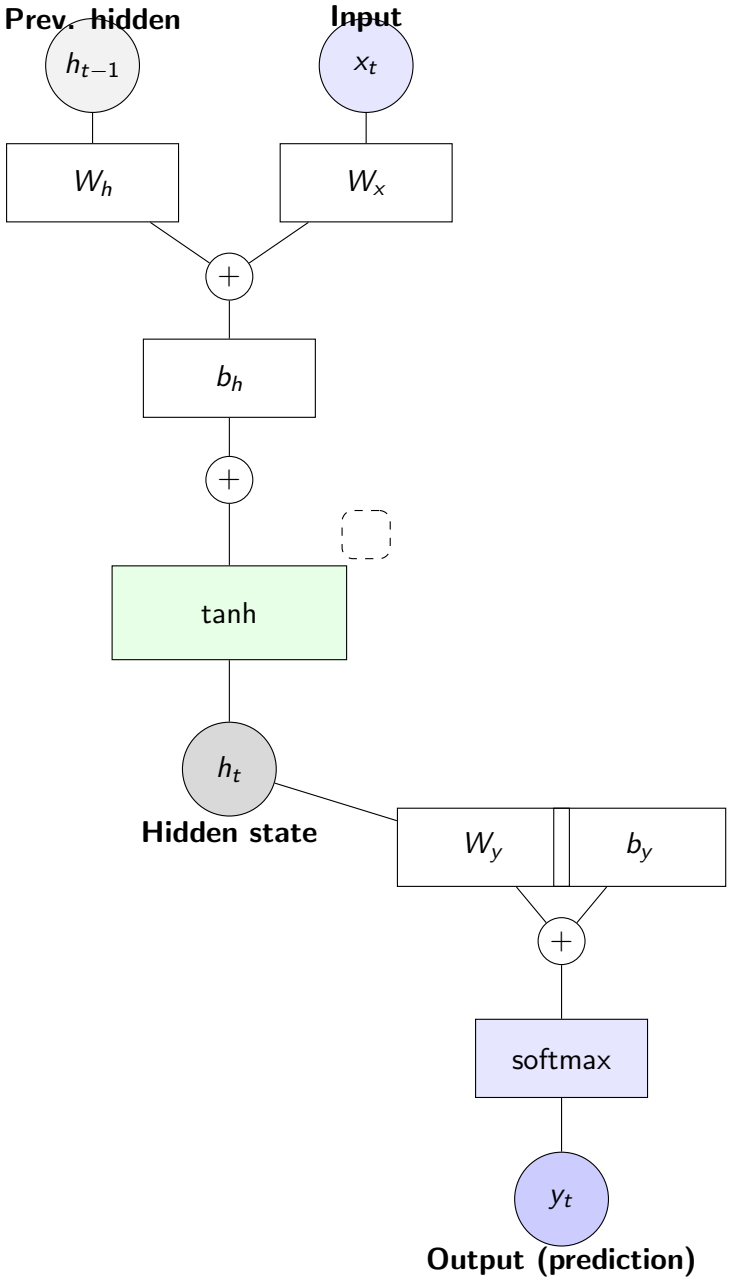
Step 2: $W_h h_{t-1}$ (hidden projection)

Step 3: $a_t = W_x x_t + W_h h_{t-1} + b_h$

Step 4: $h_t = f(a_t)$

Output: $h_t \in \mathbb{R}^H$

RNN Cell: Forward Pass



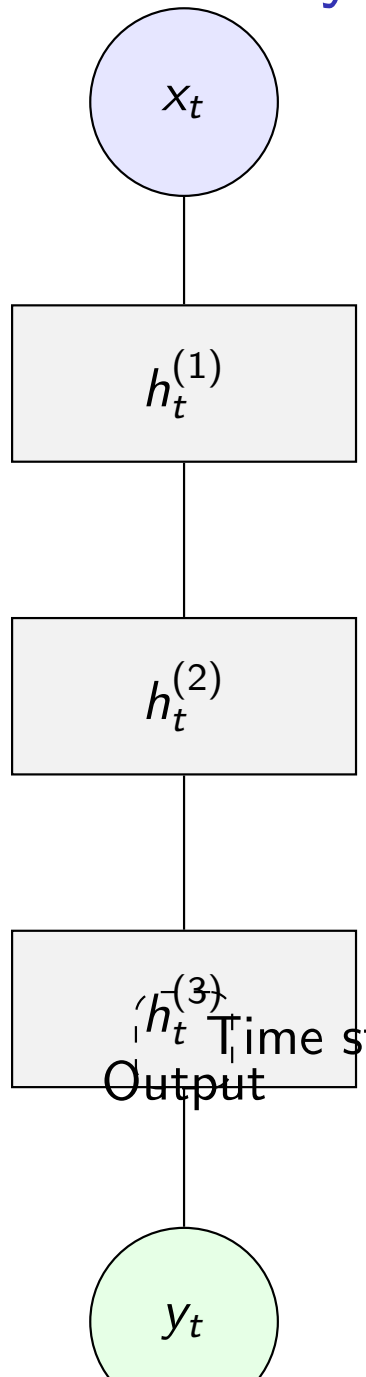
Forward Pass:

Input: $x_t \in \mathbb{R}^V, \quad h_{t-1} \in \mathbb{R}^H$
Weights:
$W_x \in \mathbb{R}^{H \times V}, \quad W_h \in \mathbb{R}^{H \times H}, \quad b_h \in \mathbb{R}^H$

Step 1: $z_x = W_x x_t$ (input projection)
Step 2: $z_h = W_h h_{t-1}$ (hidden projection)
Step 3: $a_t = z_x + z_h + b_h$ (pre-activation)
Step 4: $h_t = \tanh(a_t)$ (new hidden state)

Output Layer: $W_y \in \mathbb{R}^{V \times H}, \quad b_y \in \mathbb{R}^V$
Step 5: $o_t = W_y h_t + b_y$ (logits)
Step 6: $y_t = \text{softmax}(o_t)$ (predicted output)

Stacked RNN with 3 Hidden Layers (Single Time Step)



Why RNNs Struggle with Long-Term Dependencies

Vanishing and Exploding Gradients

- In RNNs, gradients are backpropagated through time via repeated multiplication.
- This leads to:
  - **Vanishing gradients:** early layers learn very slowly $\rightarrow$ short memory
  - **Exploding gradients:** instability in training
- Mathematically, we have:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_T} \prod_{k=t+1}^{T} \frac{\partial h_k}{\partial h_{k-1}}$$

- If $\left\| \frac{\partial h_k}{\partial h_{k-1}} \right\| < 1$, this product vanishes; if > 1, it explodes.

Why RNNs Struggle with Long-Term Dependencies

Vanishing and Exploding Gradients

- In RNNs, gradients are backpropagated through time via repeated multiplication.
- This leads to:
  - **Vanishing gradients:** early layers learn very slowly $\rightarrow$ short memory
  - **Exploding gradients:** instability in training
- Mathematically, we have:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_T} \prod_{k=t+1}^{T} \frac{\partial h_k}{\partial h_{k-1}}$$

- If $\left\| \frac{\partial h_k}{\partial h_{k-1}} \right\| < 1$, this product vanishes; if > 1, it explodes.

$\rightarrow$ Gate-based architectures

Variants: LSTM and GRU

Long Short-Term Memory (LSTM) and **Gated Recurrent Units (GRU)** are designed to mitigate vanishing gradients:

- They introduce **gates** to control:
  - What to forget
  - What to remember
  - What to output
- Enable modeling of longer-range dependencies

However, they still require sequential processing: $\Rightarrow$ **no parallelization!**

Motivates attention-based models...

Why RNNs Can't Scale

- RNNs process tokens **step-by-step**, maintaining a hidden state h_t
- This introduces a strict **temporal dependency**: to compute h_t, you must compute h_{t-1}
  - Cannot parallelize across tokens
  - Training and inference become slow and memory-inefficient
- At each time step t, the RNN updates a hidden state h_t:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

  - This state must represent all relevant history and so is a **compressed memory** adding more hidden layers (deep RNNs) helps
  - But there is no way to **look back** at earlier hidden states directly

Idea: What if each token could look at *any* previous token it wants? $\rightarrow$ This is the core idea behind **Attention**.