Neural Networks

Rishi Raj Vishwakarma

November 6, 2025

Contents

Part I

Neural Networks

Chapter 1

Feed Forward Neural Networks

1.1 Feed-forward Network Functions

The linear models for regression and classification seen previously are based on linear combinations of **fixed nonlinear basis functions** $\phi_j(\mathbf{x})$. These models take the form:

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right) \tag{1.1}$$

In this model, only the coefficients $\{w_j\}$ are adapted during training. The key idea of a neural network is to make the basis functions $\phi_j(\mathbf{x})$ themselves adaptive by parameterizing them.

1.1.1 A Series of Functional Transformations

A neural network model can be described as a series of functional transformations. We will build this up in steps, from the input layer to the output layer.

Step 1: First Layer (Input to Hidden)

First, we construct M linear combinations of the input variables $x_1, \ldots, x_D$. These are called **activations**, a_j.

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{1.2}$$

where $j = 1, \ldots, M$.

- The superscript $^{(1)}$ indicates these are parameters for the **first layer** of the network.
- $w_{ji}^{(1)}$ are the **weights**.
- $w_{j0}^{(1)}$ are the **biases**.

Example 1.1 (3 Inputs, 4 Hidden Units)**.** If we have $D = 3$ inputs ($\mathbf{x} = (x_1, x_2, x_3)^T$) and $M = 4$ hidden units, we would compute 4 separate activations:

$$\begin{aligned}
a_1 &= w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + w_{10}^{(1)} \\
a_2 &= w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3 + w_{20}^{(1)} \\
a_3 &= w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3 + w_{30}^{(1)} \\
a_4 &= w_{41}^{(1)} x_1 + w_{42}^{(1)} x_2 + w_{43}^{(1)} x_3 + w_{40}^{(1)}
\end{aligned}$$

This first layer has $(3 \times 4) = 12$ weights and 4 biases, for a total of 16 parameters.

Step 2: Hidden Unit Activation Function

Each activation a_j is then transformed by a differentiable, nonlinear **activation function**, $h(\cdot)$.

$$z_j = h(a_j) \tag{1.3}$$

These z_j are the outputs of the **hidden units**. These $z_j = h(\sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)})$ are the adaptive basis functions. They correspond to the $\phi_j(\mathbf{x})$ in Equation (5.1), but now their parameters $(w_{ji}^{(1)}, w_{j0}^{(1)})$ are learned. Common choices for $h(\cdot)$ are sigmoidal (S-shaped) functions, such as:

- **Logistic Sigmoid:** $\sigma(a) = \frac{1}{1+\exp(-a)}$
- **Hyperbolic Tangent (tanh):** $\tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$

The nonlinearity of $h(\cdot)$ is crucial; if $h(\cdot)$ were linear, the entire multi-layer network would collapse into a simple linear model.

Step 3: Second Layer (Hidden to Output)

The outputs of the hidden units, z_j, are then linearly combined to give **output unit activations**, a_k.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{1.4}$$

where $k = 1, \ldots, K$, and K is the total number of outputs.

- The superscript $^{(2)}$ indicates these are parameters for the **second layer**.
- $w_{kj}^{(2)}$ are the second-layer weights.
- $w_{k0}^{(2)}$ are the second-layer biases.

Step 4: Output Unit Activation Function

Finally, the output unit activations a_k are transformed by an output activation function to give the final network outputs y_k. The choice of this function depends on the problem:

- **Regression:** For standard regression, we use the **identity** function, $y_k = a_k$.
- **Binary Classification:** We use the **logistic sigmoid** function, $y_k = \sigma(a_k)$, where $\sigma(a) = (1 + \exp(-a))^{-1}$.
- **Multiclass Classification:** We use the **softmax** function, $y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$.

1.1.2 Overall Network Function

We can combine all these stages to write the complete network function. For a two-layer network with D inputs, M hidden units, and K outputs, using $h(\cdot)$ for the hidden layer and $\sigma(\cdot)$ for the output layer (e.g., for binary classification), the function is:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{1.5}$$

Here, $\mathbf{w}$ represents the set of all weight and bias parameters from both layers. This entire process of calculating the outputs from the inputs is known as **forward propagation**.

Example 1.2 (3-4-2 Network Diagram)**.** Let's complete our 3-input, 4-hidden-unit example, assuming $K = 2$ outputs (e.g., for a 2-class problem using 1-of-K coding, or two separate binary classifications).

1. **Input:** $\mathbf{x} = (x_1, x_2, x_3)^T$
2. **Hidden Layer ($M = 4$):**

$$\begin{aligned} a_j &= w_{j1}^{(1)} x_1 + w_{j2}^{(1)} x_2 + w_{j3}^{(1)} x_3 + w_{j0}^{(1)} && \text{(for } j = 1, \ldots, 4\text{)} \\ z_j &= h(a_j) && \text{(e.g., } z_j = \tanh(a_j)\text{)} \end{aligned}$$

3. **Output Layer ($K = 2$):**

$$\begin{aligned} a_k &= w_{k1}^{(2)} z_1 + w_{k2}^{(2)} z_2 + w_{k3}^{(2)} z_3 + w_{k4}^{(2)} z_4 + w_{k0}^{(2)} && \text{(for } k = 1, 2\text{)} \\ y_k &= \sigma(a_k) && \text{(e.g., } y_k = \text{softmax}(a_k)\text{)} \end{aligned}$$

This network structure would be a specific instance of the diagram in Figure 5.1. It has $(3 \times 4) + 4 = 16$ parameters in the first layer and $(4 \times 2) + 2 = 10$ parameters in the second layer, for a total of 26 adaptive parameters.

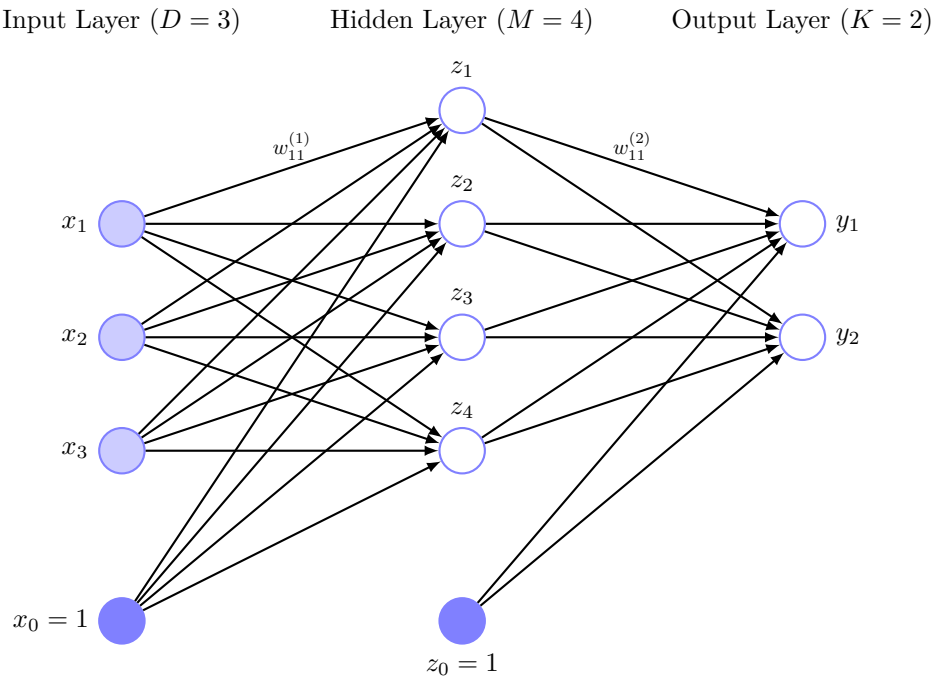


Figure 1.1: A 3-4-2 feed-forward network. Bias units x_0 and z_0 are included.

1.1.3 Differentiability of the Network Function

A key property of the multilayer perceptron, which distinguishes it from the original perceptron, is that its function is differentiable with respect to its parameters.

Proposition 1.3. *The network output function $y_k(\mathbf{x}, \mathbf{w})$ is differentiable with respect to all network parameters (weights $\mathbf{w}$ and biases).*

Proof. This property is a direct consequence of the fact that the network function is a **composition of differentiable functions**. The original perceptron used a non-differentiable step function, but the MLP uses continuous, differentiable activation functions (like 'tanh' or logistic sigmoid) for its hidden units.

We can prove this by constructing the partial derivative for an arbitrary weight in any layer using the **chain rule** of calculus.

Let's use the full network function from Equation (5.7) [cite: 91] and define its stages:

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \qquad \text{(1st layer activation)}$$

$$z_j = h(a_j^{(1)}) \qquad \text{(Hidden unit output)}$$

$$a_k^{(2)} = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \qquad \text{(2nd layer activation)}$$

$$y_k = \sigma(a_k^{(2)}) \qquad \text{(Final network output)}$$

We assume both $h(\cdot)$ and $\sigma(\cdot)$ are differentiable functions. We must show that $\frac{\partial y_k}{\partial w}$ exists for any weight w.

Case 1: A second-layer weight $w_{kj}^{(2)}$ This weight connects hidden unit j to output unit k. The derivative is found by applying the chain rule:

$$\frac{\partial y_k}{\partial w_{kj}^{(2)}} = \frac{\partial y_k}{\partial a_k^{(2)}} \cdot \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}}$$

We can solve for each term:

- $\frac{\partial y_k}{\partial a_k^{(2)}} = \sigma'(a_k^{(2)})$ (This exists because σ is differentiable).
- $\frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = \frac{\partial}{\partial w_{kj}^{(2)}} \left(\sum_{j'=1}^{M} w_{kj'}^{(2)} z_{j'} + w_{k0}^{(2)} \right) = z_j$

Combining these gives:

$$\frac{\partial y_k}{\partial w_{kj}^{(2)}} = \sigma'(a_k^{(2)}) \cdot z_j = \sigma'(a_k^{(2)}) \cdot h(a_j^{(1)}) \tag{1.6}$$

This derivative is well-defined.

Case 2: A first-layer weight $w_{ji}^{(1)}$ This weight connects input i to hidden unit j. This requires a longer chain rule, as the weight affects $a_j^{(1)} \to z_j \to a_k^{(2)} \to y_k$.

$$\frac{\partial y_k}{\partial w_{ji}^{(1)}} = \frac{\partial y_k}{\partial a_k^{(2)}} \cdot \frac{\partial a_k^{(2)}}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}$$

We can solve for each term:

- $\frac{\partial y_k}{\partial a_k^{(2)}} = \sigma'(a_k^{(2)})$
- $\frac{\partial a_k^{(2)}}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\sum_{j'=1}^{M} w_{kj'}^{(2)} z_{j'} + w_{k0}^{(2)}\right) = w_{kj}^{(2)}$
- $\frac{\partial z_j}{\partial a_j^{(1)}} = h'(a_j^{(1)})$ (This exists because h is differentiable [cite: 50]).
- $\frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = \frac{\partial}{\partial w_{ji}^{(1)}}\left(\sum_{i'=1}^{D} w_{ji'}^{(1)} x_{i'} + w_{j0}^{(1)}\right) = x_i$

Combining these gives:

$$\frac{\partial y_k}{\partial w_{ji}^{(1)}} = \sigma'(a_k^{(2)}) \cdot w_{kj}^{(2)} \cdot h'(a_j^{(1)}) \cdot x_i \tag{1.7}$$

This derivative is also well-defined.

Since we can construct a well-defined partial derivative for any weight in any layer, the entire network function $y_k(\mathbf{x}, \mathbf{w})$ is differentiable with respect to all its parameters $\mathbf{w}$. This property is what enables training via gradient descent, which is formalized by the **backpropagation** algorithm (see Section 5.3). □

1.2 Important Results and Conditions

1.2.1 Universal Approximation Theorem

A key property of feed-forward networks is that they are **universal approximators**[cite: 149].

Theorem 1.4 (Universal Approximation - Informal)**.** *A two-layer network with a sufficient number of hidden units (M) can approximate any continuous function $f(\mathbf{x})$ on a compact (i.e., closed and bounded) input domain to any desired degree of accuracy[cite: 150].*

Mathematical Intuition (Proof Sketch). A full proof is complex and relies on functional analysis. However, the intuition is as follows:

1. A single hidden unit is a "soft step function". Consider a hidden unit j with a logistic sigmoid activation $h(a_j) = \sigma(a_j)$. The activation is:

$$z_j = \sigma(\mathbf{w}_j^T \mathbf{x} + w_{j0})$$

By adjusting the bias w_{j0}, we can shift this "step" left or right. By scaling the weights $\mathbf{w}_j$, we can control the steepness of the step. As the weights $\to \infty$, the sigmoid $\sigma(a_j)$ approaches a hard step function.

2. Two hidden units can create a "bump" function. We can create a function that is non-zero only in a small region (a "bump") by subtracting two step functions. Let's imagine we have two hidden units, j_1 and j_2, with very steep activations (large weights).

- Let z_1 be a step function that "turns on" at $x = c_1$.
- Let z_2 be a step function that "turns on" at $x = c_2$, with $c_2 > c_1$.

Now, in the second layer, if we combine them with weights $w_1^{(2)} = +1$ and $w_2^{(2)} = -1$, the output y will be:

$$y \approx z_1 - z_2$$

This output y will be approximately 0 for $x < c_1$, then 1 for $c_1 < x < c_2$, and then 0 again for $x > c_2$. This creates a rectangular "bump".

3. Any function can be approximated by "bump" functions. Any continuous function $f(\mathbf{x})$ can be approximated as a sum of many "bump" functions. This is the same principle behind a histogram approximating a

probability density. By adding together many sigmoidal bumps (using the weights $w_{kj}^{(2)}$ of the output layer as their heights), we can "build" an approximation of any continuous function.

Conclusion: Each hidden unit j gives the network a "soft step function" $h(a_j)$. The network can combine these steps in the second layer to create "bumps." Given a sufficient number of hidden units (M), the network has enough "bumps" to approximate any continuous function.

The formal proofs (e.g., by Cybenko, Hornik, et al.) show that this is not just a loose analogy, but a mathematically rigorous fact for a wide range of activation functions[cite: 148, 151]. □

1.2.2 Feed-Forward Architecture

The network architecture is not limited to the simple two-layer structure. More complex mappings can be created by considering more general network diagrams. However, these architectures must adhere to a critical constraint.

Definition 1.5 (Feed-Forward)**.** A network must have a **feed-forward architecture**. This is defined as a network having **no closed directed cycles**. This constraint ensures that information flows in only one direction—from the inputs, through any hidden units, to the outputs.

Remark 1.6 (Deterministic Function). The feed-forward rule is essential because it guarantees that the network outputs are **deterministic functions of the inputs**. For any given input vector $\mathbf{x}$, the network performs a fixed sequence of calculations, and the resulting output $\mathbf{y}$ is uniquely determined.

This is in contrast to *recurrent* networks, which (by definition) contain cycles. In a recurrent network, the output can depend on an internal state that was set by previous inputs, meaning the function is no longer a static map from $\mathbf{x}$ to $\mathbf{y}$.

For a general feed-forward network, the activation z_k of any hidden or output unit k is computed by:

$$z_k = h\left(\sum_j w_{kj} z_j\right) \tag{1.8}$$

where the sum runs over all units j (which could be inputs or other hidden units) that send a connection *to* unit k. Because there are no cycles, these activations can be evaluated in a specific order, starting from the inputs and moving forward through the network until all output unit activations are evaluated.

1.3 Weight-space Symmetries

A key property of a feed-forward network is that multiple distinct choices for the weight vector $\mathbf{w}$ can all produce the same input-output mapping function $y(\mathbf{x}, \mathbf{w})$. This is a result of symmetries in the network's structure.

We will demonstrate this mathematically with a minimal example network:

- $D = 1$ input (x)
- $M = 2$ hidden units (z_1, z_2)
- $K = 1$ linear output (y)
- Hidden activation function $h(a) = \tanh(a)$

1.3.1 Baseline Network (Network A)

First, let $\mathbf{w}_A$ be the set of all weights and biases. The full network function is:

$$\begin{aligned} z_1 &= \tanh(w_{11}^{(1)} x + w_{10}^{(1)}) \\ z_2 &= \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) \\ y_A &= w_{11}^{(2)} z_1 + w_{12}^{(2)} z_2 + w_{10}^{(2)} \end{aligned}$$

Substituting gives the complete function:

$$y(\mathbf{x}, \mathbf{w}_A) = w_{11}^{(2)} \tanh(w_{11}^{(1)} x + w_{10}^{(1)}) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \tag{1.9}$$

1.3.2 Symmetry 1: Sign-Flip

This symmetry relies on the hidden activation function being **odd**, i.e., $h(-a) = -h(a)$. The $\tanh(a)$ function has this property: $\tanh(-a) = -\tanh(a)$.

We create a new **Network B** by applying a sign-flip to all weights and biases entering and leaving **hidden unit 1**.

- **New weights $\mathbf{w}_B$:**
  - $\tilde{w}_{11}^{(1)} = -w_{11}^{(1)}$ (Incoming weight)
  - $\tilde{w}_{10}^{(1)} = -w_{10}^{(1)}$ (Incoming bias)
  - $\tilde{w}_{11}^{(2)} = -w_{11}^{(2)}$ (Outgoing weight)
  - All other weights are unchanged.

The new network function y_B is:

$$\begin{aligned} y(\mathbf{x}, \mathbf{w}_B) &= \tilde{w}_{11}^{(2)} \tanh(\tilde{w}_{11}^{(1)} x + \tilde{w}_{10}^{(1)}) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \\ &= (-w_{11}^{(2)}) \tanh(-w_{11}^{(1)} x - w_{10}^{(1)}) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \\ &= (-w_{11}^{(2)}) \tanh\left(-\left[w_{11}^{(1)} x + w_{10}^{(1)}\right]\right) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \end{aligned}$$

Using $\tanh(-a) = -\tanh(a)$:

$$= (-w_{11}^{(2)}) \left[-\tanh(w_{11}^{(1)} x + w_{10}^{(1)})\right] + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)}$$

The negatives cancel:

$$\begin{aligned} &= w_{11}^{(2)} \tanh(w_{11}^{(1)} x + w_{10}^{(1)}) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \\ &= y(\mathbf{x}, \mathbf{w}_A) \end{aligned}$$

Proposition 1.7. *The network function is unchanged. Thus $\mathbf{w}_A$ and $\mathbf{w}_B$ are two different weight vectors that produce the same function. For M hidden units, there are 2^M such equivalent weight vectors from sign-flips.*

1.3.3 Symmetry 2: Interchange

This symmetry relies on the fact that the units in the hidden layer are summed in the second layer, and addition is commutative ($A + B = B + A$).

We create a new **Network C** by **interchanging units 1 and 2**. This means we swap all their incoming and outgoing weights.

- **New weights $\mathbf{w}_C$:**
  - Incoming: $\hat{w}_{11}^{(1)} = w_{21}^{(1)}$, $\hat{w}_{10}^{(1)} = w_{20}^{(1)}$
  - Incoming: $\hat{w}_{21}^{(1)} = w_{11}^{(1)}$, $\hat{w}_{20}^{(1)} = w_{10}^{(1)}$
  - Outgoing: $\hat{w}_{11}^{(2)} = w_{12}^{(2)}$
  - Outgoing: $\hat{w}_{12}^{(2)} = w_{11}^{(2)}$

The new network function y_C is:

$$\begin{aligned} y(\mathbf{x}, \mathbf{w}_C) &= \hat{w}_{11}^{(2)} \tanh(\hat{w}_{11}^{(1)} x + \hat{w}_{10}^{(1)}) + \hat{w}_{12}^{(2)} \tanh(\hat{w}_{21}^{(1)} x + \hat{w}_{20}^{(1)}) + w_{10}^{(2)} \\ &= (w_{12}^{(2)}) \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + (w_{11}^{(2)}) \tanh(w_{11}^{(1)} x + w_{10}^{(1)}) + w_{10}^{(2)} \end{aligned}$$

By commutativity of addition:

$$\begin{aligned} &= w_{11}^{(2)} \tanh(w_{11}^{(1)} x + w_{10}^{(1)}) + w_{12}^{(2)} \tanh(w_{21}^{(1)} x + w_{20}^{(1)}) + w_{10}^{(2)} \\ &= y(\mathbf{x}, \mathbf{w}_A) \end{aligned}$$

Proposition 1.8. *The network function is unchanged. For M hidden units, there are $M!$ (M-factorial) permutations of the units, leading to $M!$ equivalent weight vectors.*

Corollary 1.9 (Total Symmetry)**.** *For a two-layer network with M hidden units (using an odd activation function), any given weight vector $\mathbf{w}$ belongs to a set of $\mathbf{M!2^M}$ equivalent weight vectors that all produce the identical input-output mapping.*

Remark 1.10 (Activation Function Dependant Symmetries). The **interchange symmetry** (the $M!$ factor) is general and applies to any hidden unit activation function. However, the **sign-flip symmetry** (the 2^M factor) is specific to **odd** activation functions, such as $\tanh(a)$, where $h(-a) = -h(a)$.

Chapter 2

Network training

2.1 Fundamentals

We provide a probabilistic interpretation for the network outputs, which will help motivate the choice of error functions.

2.1.1 Regression Problems

We first consider regression, assuming a single target variable t. We assume t has a Gaussian distribution, with the mean given by the network's output.

Definition 2.1 (Conditional Distribution for Regression)**.** The conditional distribution of t given an input $\mathbf{x}$ is:

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) \tag{2.1}$$

where β is the precision (inverse variance) of the Gaussian noise. For this model, the output unit activation function is the identity, $y(\mathbf{x}, \mathbf{w}) = a(\mathbf{x}, \mathbf{w})$, as this allows the network to approximate any continuous function.

The Error Function (Maximum Likelihood)

Given a training set of N i.i.d. observations, $\{\mathbf{x}_n, t_n\}$, the likelihood function is:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta) \tag{2.2}$$

The negative log-likelihood (our error function) is:

$$\begin{aligned}
E(\mathbf{w}, \beta) &= -\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) \\
&= -\sum_{n=1}^{N} \ln \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}) \\
&= -\sum_{n=1}^{N} \left\{ \ln \left(\frac{\beta}{2\pi} \right)^{1/2} - \frac{\beta}{2}(y(\mathbf{x}_n, \mathbf{w}) - t_n)^2 \right\} \\
&= -\sum_{n=1}^{N} \left\{ \frac{1}{2} \ln \beta - \frac{1}{2} \ln(2\pi) - \frac{\beta}{2}(y_n - t_n)^2 \right\} \\
&= \frac{\beta}{2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi)
\end{aligned}$$

To find the maximum likelihood solution for the weights, $\mathbf{w}_{ML}$, we minimize this function with respect to $\mathbf{w}$. The terms that do not depend on $\mathbf{w}$ can be discarded, leaving us with the sum-of-squares error.

Proposition 2.2. *Maximizing the likelihood for* $\mathbf{w}$ *is equivalent to minimizing the sum-of-squares error function:*

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 \tag{2.3}$$

Determining the Noise Precision β

Once $\mathbf{w}_{ML}$ is found, we can find β_{ML} by minimizing the negative log-likelihood $E(\mathbf{w}_{ML}, \beta)$ with respect to β.

$$\frac{\partial E(\mathbf{w}_{ML}, \beta)}{\partial \beta} = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n\}^2 - \frac{N}{2\beta}$$

$$\text{Setting to 0:} \quad \frac{N}{2\beta} = \frac{1}{2}\sum_{n=1}^{N}\{y_n - t_n\}^2$$

$$\implies \frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n\}^2$$

The maximum likelihood solution for the variance $(1/\beta)$ is the average of the squared errors.

Multiple Target Variables (K outputs)

If we assume K independent target variables, each with a Gaussian distribution sharing the same precision β:

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \beta^{-1}\mathbf{I}) \tag{2.4}$$

The negative log-likelihood (ignoring constants) becomes:

$$E(\mathbf{w}, \beta) = \frac{\beta}{2}\sum_{n=1}^{N}\|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2 - \frac{NK}{2}\ln\beta$$

The maximum likelihood solution for $\mathbf{w}$ is again found by minimizing the sum-of-squares error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2 \tag{2.5}$$

The solution for the precision is found similarly, by averaging over all N patterns and K outputs:

$$\frac{1}{\beta_{ML}} = \frac{1}{NK}\sum_{n=1}^{N}\|\mathbf{y}(\mathbf{x}_n, \mathbf{w}_{ML}) - \mathbf{t}_n\|^2 \tag{2.6}$$

Derivative of the Error Function

For regression, we pair the identity activation function $y_k = a_k$ with the sum-of-squares error function $E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(y_{nk} - t_{nk})^2$.

This pairing has a mathematically simple and useful property. When we calculate the derivative of the error for a single pattern n with respect to the activation a_k of an output unit k, we get a very clean result.

Proposition 2.3. *The derivative of the sum-of-squares error E_n with respect to an output unit's activation a_k is simply the error: (output - target).*

$$\frac{\partial E_n}{\partial a_k} = y_k - t_k \tag{2.7}$$

Proof. The error E_n for a single pattern n is $E_n = \frac{1}{2}\sum_{j=1}^{K}(y_j - t_j)^2$. We find the partial derivative with respect to a specific activation a_k:

$$\frac{\partial E_n}{\partial a_k} = \frac{\partial}{\partial a_k}\left[\frac{1}{2}\sum_{j=1}^{K}(y_j - t_j)^2\right]$$

$$\text{Only the } j = k \text{ term in the sum depends on } a_k:$$

$$= \frac{\partial}{\partial a_k}\left[\frac{1}{2}(y_k - t_k)^2\right]$$

$$\text{Using the chain rule:} \quad = \frac{1}{2}\cdot 2(y_k - t_k)\cdot\frac{\partial(y_k - t_k)}{\partial a_k}$$

$$= (y_k - t_k)\cdot\left(\frac{\partial y_k}{\partial a_k} - \frac{\partial t_k}{\partial a_k}\right)$$

$$\text{Since } t_k \text{ is a fixed target, } \frac{\partial t_k}{\partial a_k} = 0. \text{ Since } y_k = a_k, \frac{\partial y_k}{\partial a_k} = 1.$$

$$= (y_k - t_k)\cdot(1 - 0)$$

$$= y_k - t_k$$

This simple "error" signal $(y_k - t_k)$ is what is used in the first step of backpropagation, which we will see in Section 5.3. □

Remark 2.4 (Why the Identity Activation Works for Regression). The use of the identity function $y_k = a_k$ for the output layer in regression is a necessary choice.

- **The Goal:** The goal of regression is to predict a target t_k that can be any real value (e.g., $t_k \in (-\infty, +\infty)$).
- **The Math:** The activation a_k is computed as a linear combination $a_k = \sum_j w_{kj}^{(2)} z_j + w_{k0}^{(2)}$. This sum a_k can already produce any real value.
- **The Logic:** By setting $y_k = a_k$, we allow the network's final output y_k to also be any real value. If we were to use a non-linear function, such as $y_k = \tanh(a_k)$, we would *restrict* the network's output to the range $(-1, 1)$, making it impossible to predict any target values outside this range.

The complex, non-linear "learning" is done by the hidden layers. The output layer for regression is kept linear (using the identity function) so as not to limit the final prediction.

2.1.2 Classification Problems

For classification, the target variable t represents a class label. We must adapt the network's output and the error function accordingly.

Binary Classification

We consider a two-class problem with $t = 1$ for class $\mathcal{C}_1$ and $t = 0$ for class $\mathcal{C}_2$. The network output $y(\mathbf{x}, \mathbf{w})$ must represent a probability, $p(\mathcal{C}_1|\mathbf{x})$, and thus must be in the range $[0, 1]$. We achieve this by using the **logistic sigmoid** activation function for the single output unit:

$$y = \sigma(a) \equiv \frac{1}{1 + \exp(-a)} \tag{2.8}$$

The conditional distribution of the target is a Bernoulli distribution:

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \{1 - y(\mathbf{x}, \mathbf{w})\}^{1-t} \tag{2.9}$$

We aim to find $\mathbf{w}$ by maximizing the likelihood. For N i.i.d. data points, the likelihood function is:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

where $y_n = y(\mathbf{x}_n, \mathbf{w})$. Maximizing the likelihood is equivalent to minimizing the negative log-likelihood, which defines the **cross-entropy error function**.

Proposition 2.5. *The negative log-likelihood for binary classification is the cross-entropy error function.*

Proof.

$$\begin{aligned}
E(\mathbf{w}) &= -\ln p(\mathbf{t}|\mathbf{w}) \\
&= -\ln \left(\prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \right) \\
&= -\sum_{n=1}^{N} \ln \left(y_n^{t_n} \{1 - y_n\}^{1-t_n} \right) \\
&= -\sum_{n=1}^{N} \left\{ \ln(y_n^{t_n}) + \ln(\{1 - y_n\}^{1-t_n}) \right\} \\
E(\mathbf{w}) &= -\sum_{n=1}^{N} \{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \}
\end{aligned}$$

□

This is the error function given in Equation (5.21).

Multiple Independent Binary Classifications

We can extend this to K separate binary classifications. We use a network with K output units, where each unit k has a logistic sigmoid activation function, $y_k = \sigma(a_k)$. Each y_k represents an independent probability $p(t_k = 1|\mathbf{x})$. The conditional distribution for a single data point $\mathbf{x}_n$ with target vector $\mathbf{t}_n$ is:

$$p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) = \prod_{k=1}^{K} y_{nk}^{t_{nk}}[1 - y_{nk}]^{1-t_{nk}} \tag{2.10}$$

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$. The full likelihood for N data points is $p(\mathbf{T}|\mathbf{w}) = \prod_{n=1}^{N} p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w})$. The negative log-likelihood is:

$$\begin{aligned}
E(\mathbf{w}) &= -\ln p(\mathbf{T}|\mathbf{w}) \\
&= -\sum_{n=1}^{N} \ln p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}) \\
&= -\sum_{n=1}^{N} \ln \left(\prod_{k=1}^{K} y_{nk}^{t_{nk}}[1 - y_{nk}]^{1-t_{nk}} \right) \\
&= -\sum_{n=1}^{N} \sum_{k=1}^{K} \ln \left(y_{nk}^{t_{nk}}[1 - y_{nk}]^{1-t_{nk}} \right) \\
E(\mathbf{w}) &= -\sum_{n=1}^{N} \sum_{k=1}^{K} \{t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk})\}
\end{aligned}$$

This is the error function from Equation (5.23).

Derivative of the Error Function

This "canonical pairing" of the logistic sigmoid output and the cross-entropy error function results in a very simple derivative, just as in the regression case.

Proposition 2.6. *The derivative of the cross-entropy error E with respect to the activation a_{nj} of a specific output unit j for a pattern n is given by:*

$$\frac{\partial E}{\partial a_{nj}} = y_{nj} - t_{nj}$$

This is the form of Equation (5.18).

Proof. The total error E is a sum over all patterns n' and all outputs k'. We only need to consider the terms that depend on a_{nj}.

$$E = \sum_{n'=1}^{N} \sum_{k=1}^{K} E_{n'k} = \sum_{n'=1}^{N} \sum_{k=1}^{K} -\{t_{n'k} \ln y_{n'k} + (1 - t_{n'k}) \ln(1 - y_{n'k})\}$$

The activation a_{nj} only influences y_{nj}. Thus, all terms in the sum where $n' \neq n$ or $k \neq j$ have a derivative of zero.

$$\frac{\partial E}{\partial a_{nj}} = \frac{\partial E_{nj}}{\partial a_{nj}} = \frac{\partial E_{nj}}{\partial y_{nj}} \cdot \frac{\partial y_{nj}}{\partial a_{nj}} \quad \text{(by the chain rule)}$$

First, we find $\frac{\partial E_{nj}}{\partial y_{nj}}$:

$$\begin{aligned}
\frac{\partial E_{nj}}{\partial y_{nj}} &= \frac{\partial}{\partial y_{nj}} \left[-\{t_{nj} \ln y_{nj} + (1 - t_{nj}) \ln(1 - y_{nj})\} \right] \\
&= -\left[\frac{t_{nj}}{y_{nj}} + (1 - t_{nj}) \frac{1}{1 - y_{nj}} \cdot (-1) \right] \\
&= -\left[\frac{t_{nj}}{y_{nj}} - \frac{1 - t_{nj}}{1 - y_{nj}} \right] \\
&= -\left[\frac{t_{nj}(1 - y_{nj}) - y_{nj}(1 - t_{nj})}{y_{nj}(1 - y_{nj})} \right] \\
&= -\left[\frac{t_{nj} - t_{nj}y_{nj} - y_{nj} + t_{nj}y_{nj}}{y_{nj}(1 - y_{nj})} \right] \\
&= -\left[\frac{t_{nj} - y_{nj}}{y_{nj}(1 - y_{nj})} \right] = \frac{y_{nj} - t_{nj}}{y_{nj}(1 - y_{nj})}
\end{aligned}$$

Second, we find the derivative of the logistic sigmoid $y_{nj} = \sigma(a_{nj})$:

$$\frac{\partial y_{nj}}{\partial a_{nj}} = \sigma'(a_{nj}) = \sigma(a_{nj})(1 - \sigma(a_{nj})) = y_{nj}(1 - y_{nj})$$

Finally, we multiply the two parts:

$$\frac{\partial E}{\partial a_{nj}} = \left(\frac{y_{nj} - t_{nj}}{y_{nj}(1 - y_{nj})} \right) \cdot (y_{nj}(1 - y_{nj}))$$
$$\frac{\partial E}{\partial a_{nj}} = y_{nj} - t_{nj}$$

□

2.1.3 Numerical Example: Binary Classification

Let's consider a simple network with $D = 2$ inputs, $M = 1$ hidden unit, and $K = 1$ output unit.

- **Activation (Hidden):** $h(a) = \tanh(a)$
- **Activation (Output):** $y = \sigma(a) = (1 + \exp(-a))^{-1}$
- **Error Function:** $E(\mathbf{w}) = -\{t \ln y + (1 - t) \ln(1 - y)\}$

Parameters and Data

We define a single data point $(\mathbf{x}, t)$ and a set of weights $\mathbf{w}$.

- **Input x:** $x_1 = 0.5$, $x_2 = 1.0$ (We use $x_0 = 1$ for the bias)
- **Target t:** $t = 1$
- **Layer 1 Weights:** $w_{11}^{(1)} = 0.8$, $w_{12}^{(1)} = -0.5$, $w_{10}^{(1)} = 0.2$ (bias)
- **Layer 2 Weights:** $w_{11}^{(2)} = 1.0$, $w_{10}^{(2)} = -0.3$ (bias)

Step 1: Forward Propagation

We now calculate the network's output y for the input $\mathbf{x}$.

1. **Calculate Hidden Unit Activation $a_1^{(1)}$**

$$a_1^{(1)} = \left(w_{11}^{(1)} \cdot x_1\right) + \left(w_{12}^{(1)} \cdot x_2\right) + w_{10}^{(1)}$$
$$a_1^{(1)} = (0.8 \cdot 0.5) + (-0.5 \cdot 1.0) + 0.2$$
$$a_1^{(1)} = 0.4 - 0.5 + 0.2 = 0.1$$

2. **Calculate Hidden Unit Output z_1**

$$z_1 = h(a_1^{(1)}) = \tanh(0.1)$$
$$z_1 \approx 0.09967 \approx 0.10$$

3. **Calculate Output Unit Activation $a_1^{(2)}$**

$$a_1^{(2)} = \left(w_{11}^{(2)} \cdot z_1\right) + w_{10}^{(2)}$$
$$a_1^{(2)} = (1.0 \cdot 0.10) - 0.3$$
$$a_1^{(2)} = 0.10 - 0.3 = -0.20$$

4. **Calculate Final Network Output y_1 (Prediction)**

$$y_1 = \sigma(a_1^{(2)}) = \sigma(-0.20)$$
$$y_1 = \frac{1}{1 + \exp(-(-0.20))} = \frac{1}{1 + \exp(0.20)}$$
$$y_1 \approx \frac{1}{1 + 1.2214} = \frac{1}{2.2214}$$
$$y_1 \approx 0.450$$

So, the network's prediction is $y_1 = 0.45$. It predicts a 45% probability that the class is 1.

Step 2: Calculate the Error

Now we compute the cross-entropy error for this single pattern, given $t = 1$.

$$\begin{aligned} E(\mathbf{w}) &= -\{t \ln y_1 + (1-t)\ln(1-y_1)\} \\ E(\mathbf{w}) &= -\{1 \cdot \ln(0.450) + (1-1)\cdot \ln(1-0.450)\} \\ E(\mathbf{w}) &= -\{\ln(0.450) + 0 \cdot \ln(0.550)\} \\ E(\mathbf{w}) &= -\ln(0.450) \\ E(\mathbf{w}) &\approx -(-0.7985) \\ E(\mathbf{w}) &\approx 0.7985 \end{aligned}$$

The error for this pattern is ≈ 0.7985. The goal of training would be to adjust the 6 weights to make y_1 closer to $t = 1$, which would, in turn, reduce this error value towards 0.

2.2 Parameter Optimisation

The goal of training is to find a weight vector $\mathbf{w}$ that minimizes the error function $E(\mathbf{w})$. Because $E(\mathbf{w})$ is a smooth continuous function, its smallest value (a local or global minimum) will occur at a **stationary point** where the gradient of the error function vanishes.

$$\nabla E(\mathbf{w}) = 0 \tag{2.11}$$

2.2.1 Challenges: Non-Convexity and Multiple Minima

Finding a solution to $\nabla E(\mathbf{w}) = 0$ is highly non-trivial. The error function's highly nonlinear dependence on $\mathbf{w}$ creates two major challenges:

1. **Inequivalent Local Minima:** The error surface has many *inequivalent* local minima. An algorithm may converge to a local minimum that is not the global minimum.
2. **Equivalent Minima:** Due to the **weight-space symmetries** (Section 5.1.1), any single minimum $\mathbf{w}^*$ is part of a large family (of size $M!2^M$ for a two-layer tanh network) of equivalent minima that all share the same error value.

2.2.2 Impossibility of an Analytic Solution

We cannot solve $\nabla E(\mathbf{w}) = 0$ analytically (i.e., with a "closed-form" solution).

- **The Reason:** An analytic solution would require us to algebraically isolate $\mathbf{w}$.
- **The Math:** The gradient ∇E is a complex sum over all data points. The gradient for a single first-layer weight $w_{ji}^{(1)}$ has the form (using our previous results):

$$\frac{\partial E}{\partial w_{ji}^{(1)}} \propto \sum_{n,k} (y_{nk} - t_{nk}) \cdot \sigma'(a_{nk}^{(2)}) \cdot w_{kj}^{(2)} \cdot h'(a_{nj}^{(1)}) \cdot x_{ni}$$

  The weight $w_{ji}^{(1)}$ is "trapped" inside the non-linear function $h'(a_{nj}^{(1)}) = h'(\sum_i w_{ji}^{(1)} x_i + \dots)$. This creates a **transcendental equation** (not a simple polynomial) that cannot be solved for $w_{ji}^{(1)}$ using standard algebra.

2.2.3 Iterative Numerical Solution

Because no analytic solution exists, we must use iterative numerical procedures. These methods start with an initial guess $\mathbf{w}^{(0)}$ and move through weight space in a series of steps:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{2.12}$$

where τ is the iteration step. Different optimization algorithms (like gradient descent) are defined by different choices for the update $\Delta\mathbf{w}^{(\tau)}$, which generally makes use of **gradient information** $\nabla E(\mathbf{w})$.

2.3 Local Quadratic Approximation *Optional*

To understand the optimization problem and the algorithms that solve it, we can gain insight by approximating the complex $E(\mathbf{w})$ landscape with a simple quadratic function.

2.3.1 Primer on the Mathematics

Definition 2.7 (Taylor Expansion)**.** The **Taylor Expansion** is a mathematical tool for approximating a complex, smooth function in the local neighborhood of a point. The quadratic approximation is like "zooming in" on a point on a complex, hilly landscape until it looks like a simple, smooth "bowl" (a parabola in 2D, or a paraboloid in 3D).

Definition 2.8 (Gradient (∇E))**.** The **Gradient** (denoted **b** in this section) is a vector. It represents the **slope and direction** of the error landscape at a specific point.

- **Direction:** It always points in the direction of the *steepest uphill* path.
- **Magnitude:** Its length represents how steep that slope is.
- **In the approximation:** This is the *linear* term and defines the "tilt" of the bowl. At the very bottom of a minimum, the ground is flat, so the gradient is a zero vector.

Definition 2.9 (Hessian (H))**.** The **Hessian** is a matrix (a 2D grid) of all possible second-order partial derivatives. It represents the **curvature** of the error landscape.

- **Meaning:** It describes the *shape* of the bowl, i.e., how the slope is changing in every direction.
- **Example:** A perfectly round bowl has a simple Hessian. A long, narrow, elliptical valley (a "ravine") has a complex Hessian: high curvature across the valley (steep walls) and low curvature along its length (gentle slope).
- **In the approximation:** This is the *quadratic* term and defines the bowl's shape.

2.3.2 The Quadratic Approximation

We consider a Taylor expansion of $E(\mathbf{w})$ around a point $\hat{\mathbf{w}}$:

$$E(\mathbf{w}) \simeq E(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})^T \mathbf{b} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^T H (\mathbf{w} - \hat{\mathbf{w}}) \tag{2.13}$$

where cubic and higher terms are omitted. The vector **b** and matrix H are:

- The **gradient b** evaluated at $\hat{\mathbf{w}}$:
$$\mathbf{b} \equiv \nabla E|_{\mathbf{w}=\hat{\mathbf{w}}} \tag{2.14}$$
- The **Hessian** H evaluated at $\hat{\mathbf{w}}$, with elements:
$$(H)_{ij} \equiv \left.\frac{\partial^2 E}{\partial w_i \partial w_j}\right|_{\mathbf{w}=\hat{\mathbf{w}}} \tag{2.15}$$

Using this, the local approximation for the gradient itself is:

$$\nabla E \simeq \mathbf{b} + H(\mathbf{w} - \hat{\mathbf{w}}) \tag{2.16}$$

2.3.3 Analysis at a Minimum

We are particularly interested in the case where our point $\hat{\mathbf{w}}$ is a local minimum, which we call $\mathbf{w}^*$.

1. At a minimum, the gradient is zero, so $\mathbf{b} = 0$.
2. The quadratic approximation (5.28) simplifies to the equation for a perfect "bowl" centered at $\mathbf{w}^*$:
$$E(\mathbf{w}) \simeq E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T H (\mathbf{w} - \mathbf{w}^*) \tag{2.17}$$
3. The Hessian H is symmetric and has a set of eigenvectors $\mathbf{u}_i$ and corresponding eigenvalues λ_i defined by:
$$H\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{2.18}$$
4. If we express the vector $(\mathbf{w} - \mathbf{w}^*)$ in terms of the eigenvectors (a change of coordinates), the error function simplifies to:
$$E(\mathbf{w}) \simeq E(\mathbf{w}^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2 \tag{2.19}$$

For $\mathbf{w}^*$ to be a minimum, all directions must curve upwards. This means all eigenvalues must be positive, $\lambda_i > 0$. A matrix with this property is called **positive definite**.

2.3.4 Significance in Neural Network Training

- **Why we study this:** This approximation is the key to creating optimization algorithms that are much more powerful than simple gradient descent.
- **The Problem with Gradient Descent:** Gradient descent only uses the gradient (**b**). If it is in a long, narrow "ravine" (an elliptical bowl where one λ is very small and another is very large), the gradient will point almost directly at the steep side wall. The algorithm will oscillate back and forth, making very slow progress down the valley.
- **The Solution (Advanced Optimizers):** Advanced methods (like Newton's method) use the Hessian (H) to get a full picture of the "bowl's" shape. They can rescale the step in each direction (using the eigenvalues λ_i) to aim directly for the true minimum, converging much faster.
- **Foundation for Bayesian Methods:** This quadratic approximation (also called the **Laplace approximation**) is the foundation for Bayesian neural networks (Section 5.7), where it is used to approximate the complex posterior distribution.

2.4 Gradient Descent Optimization

The simplest iterative optimization algorithms use the gradient information by taking small steps in the direction of the negative gradient.

2.4.1 Batch Gradient Descent

This algorithm, also known as **steepest descent**, uses the entire training set to calculate the gradient $\nabla E(\mathbf{w})$ at each iteration. The weight update $\Delta\mathbf{w}^{(\tau)}$ from Equation (5.27) is given by:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E(\mathbf{w}^{(\tau)}) \tag{2.20}$$

where $\eta > 0$ is the **learning rate**.

- At each step, the weight vector is moved in the direction of the greatest rate of decrease of the error function.
- This is a **batch method** because the error function $E(\mathbf{w})$ is a sum over the entire training set, and the gradient must be computed for all N patterns.
- This simple gradient descent approach is often an inefficient algorithm.

More powerful batch optimization methods, which are much faster and more robust, exist. These include **conjugate gradients** and **quasi-Newton methods**.

2.4.2 On-line (Stochastic) Gradient Descent

An alternative, on-line version of gradient descent, also known as **sequential gradient descent** or **stochastic gradient descent (SGD)**, updates the weight vector using one data point at a time.

The error function $E(\mathbf{w})$ is a sum of terms for each data point:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{2.21}$$

The SGD update is based only on the gradient of one of these terms, E_n:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E_n(\mathbf{w}^{(\tau)}) \tag{2.22}$$

This update is repeated by cycling through the data, either in sequence or by selecting points at random. SGD has several advantages over batch methods:

- It handles redundancy in large datasets much more efficiently.
- It has the possibility of escaping from local minima, since a stationary point for the full error $E(\mathbf{w})$ will generally not be a stationary point for an individual $E_n(\mathbf{w})$.

Chapter 3

Error Backpropagation

3.1 Backpropagation Example: A 1-1-1-1 Network

To understand the derivation of the backpropagation algorithm, we will use a simple "deep" network with one input, two hidden layers (each with one neuron), and one output neuron.

3.1.1 1. Network & Problem Setup

- **Input:** x (and a bias $x_0 = 1$)
- **Hidden Layer 1 (H1):** 1 neuron.
  - Activation: $a_1 = w_{11}^{(1)} x + w_{10}^{(1)}$
  - Output: $z_1 = \tanh(a_1)$
- **Hidden Layer 2 (H2):** 1 neuron.
  - Activation: $a_2 = w_{11}^{(2)} z_1 + w_{10}^{(2)}$
  - Output: $z_2 = \tanh(a_2)$
- **Output Layer (O1):** 1 neuron.
  - Activation: $a_3 = w_{11}^{(3)} z_2 + w_{10}^{(3)}$
  - Output: $y_1 = a_3$ (Identity function for regression)
- **Error Function:** We use the sum-of-squares error for a single pattern n:

$$E = \frac{1}{2}(y_1 - t)^2 \tag{3.1}$$

Our Goal: Find the derivative of E for all 6 parameters: $\{w_{11}^{(3)}, w_{10}^{(3)}, w_{11}^{(2)}, w_{10}^{(2)}, w_{11}^{(1)}, w_{10}^{(1)}\}$.

3.1.2 2. The Forward Propagation Pass

First, we perform a forward pass. We take our input x and compute all unit outputs. We assume these values are now known:

1. $z_1 = \tanh(w_{11}^{(1)} x + w_{10}^{(1)})$
2. $z_2 = \tanh(w_{11}^{(2)} z_1 + w_{10}^{(2)})$
3. $y_1 = w_{11}^{(3)} z_2 + w_{10}^{(3)}$

3.1.3 3. The Backward Propagation Pass (The Derivatives)

We will now calculate the gradients, starting from the output layer and moving backward.

Step 3a: Gradients for Layer 3 (Output Layer)

Theory: We use the core rule from Eq. (5.53): $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$. Here, $j = 3$ (our output neuron) and the input z_i is z_2 (from H2). For the bias, the input is $z_0 = 1$.

$$\frac{\partial E}{\partial w_{11}^{(3)}} = \delta_3 z_2$$
$$\frac{\partial E}{\partial w_{10}^{(3)}} = \delta_3 \cdot 1$$

Calculation: We need to find δ_3. By definition, δ_3 is the error-derivative with respect to the *activation* a_3.

$$\delta_3 \equiv \frac{\partial E}{\partial a_3}$$

We use the chain rule: $\frac{\partial E}{\partial a_3} = \frac{\partial E}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_3}$.

1. $\frac{\partial E}{\partial y_1} = \frac{\partial}{\partial y_1}\left[\frac{1}{2}(y_1 - t)^2\right] = (y_1 - t)$
2. $\frac{\partial y_1}{\partial a_3} = \frac{\partial}{\partial a_3}[a_3] = 1$ (since it's the identity function)

So, the error signal for the output layer is simply:

$$\delta_3 = (y_1 - t) \tag{3.2}$$

Final Gradients (Layer 3):

$$\frac{\partial E}{\partial w_{11}^{(3)}} = (y_1 - t) z_2$$
$$\frac{\partial E}{\partial w_{10}^{(3)}} = (y_1 - t)$$

Step 3b: Gradients for Layer 2 (Hidden Layer H2)

Theory: We use the same rule: $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$. Here, $j = 2$ (H2 neuron) and the input z_i is z_1 (from H1). For the bias, the input is $z_0 = 1$.

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \delta_2 z_1$$
$$\frac{\partial E}{\partial w_{10}^{(2)}} = \delta_2 \cdot 1$$

Calculation: We need to find δ_2. By definition, $\delta_2 = \frac{\partial E}{\partial a_2}$. We use the chain rule, noting that a_2 affects E *only* by affecting a_3 (via z_2).

$$\delta_2 = \frac{\partial E}{\partial a_2} = \frac{\partial E}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_2}$$

Let's group these terms:

$$\delta_2 = \left(\frac{\partial E}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_2}\right) \cdot \frac{\partial z_2}{\partial a_2}$$

1. The first part, $\left(\frac{\partial E}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_2}\right)$, is "the error signal from above, propagated back".
   - We know $\frac{\partial E}{\partial a_3} = \delta_3$.
   - We know $a_3 = w_{11}^{(3)} z_2 + w_{10}^{(3)}$, so $\frac{\partial a_3}{\partial z_2} = w_{11}^{(3)}$.
   - This part is therefore $\delta_3 w_{11}^{(3)}$.
2. The second part, $\frac{\partial z_2}{\partial a_2}$, is the derivative of the local activation function.
   - $z_2 = \tanh(a_2)$, so $\frac{\partial z_2}{\partial a_2} = 1 - \tanh^2(a_2) = 1 - z_2^2$.

This is the general backpropagation rule: $\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$. In our simple case, $j = 2$, $h' = 1 - z_2^2$, and the sum over k is just the single term for $k = 3$.

$$\delta_2 = (1 - z_2^2) \cdot w_{11}^{(3)}\delta_3 \tag{3.3}$$

Final Gradients (Layer 2):

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \delta_2 z_1 = \Big((1 - z_2^2)w_{11}^{(3)}(y_1 - t)\Big)\, z_1$$
$$\frac{\partial E}{\partial w_{10}^{(2)}} = \delta_2 \cdot 1 = \Big((1 - z_2^2)w_{11}^{(3)}(y_1 - t)\Big)$$

Step 3c: Gradients for Layer 1 (Hidden Layer H1)

Theory: We use the same rule: $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$. Here, $j = 1$ (H1 neuron) and the input z_i is x (the network input). For the bias, the input is $x_0 = 1$.

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \delta_1 x$$
$$\frac{\partial E}{\partial w_{10}^{(1)}} = \delta_1 \cdot 1$$

Calculation: We need to find δ_1. By definition, $\delta_1 = \frac{\partial E}{\partial a_1}$. We use the chain rule. a_1 affects E only through a_2.

$$\delta_1 = \frac{\partial E}{\partial a_1} = \frac{\partial E}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial a_1}$$

We group these just as before:

$$\delta_1 = \left(\frac{\partial E}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_1}\right) \cdot \frac{\partial z_1}{\partial a_1}$$

1. The first part is the "error signal from above" (δ_2), propagated back.
   - We know $\frac{\partial E}{\partial a_2} = \delta_2$.
   - We know $a_2 = w_{11}^{(2)} z_1 + w_{10}^{(2)}$, so $\frac{\partial a_2}{\partial z_1} = w_{11}^{(2)}$.
   - This part is therefore $\delta_2 w_{11}^{(2)}$.
2. The second part is the local derivative $h'(a_1)$.
   - $z_1 = \tanh(a_1)$, so $\frac{\partial z_1}{\partial a_1} = 1 - \tanh^2(a_1) = 1 - z_1^2$.

This is the backpropagation rule for $j = 1$.

$$\delta_1 = (1 - z_1^2) \cdot w_{11}^{(2)}\delta_2 \tag{3.4}$$

Final Gradients (Layer 1):

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \delta_1 x = \Big((1 - z_1^2)w_{11}^{(2)}\delta_2\Big)\, x$$
$$\frac{\partial E}{\partial w_{10}^{(1)}} = \delta_1 \cdot 1 = \Big((1 - z_1^2)w_{11}^{(2)}\delta_2\Big)$$

...where δ_2 is the value we calculated in the previous step.

3.1.4 4. Summary of the Algorithm

1. **Forward Pass:** Compute and store z_1, z_2, y_1.
2. **Backward Pass (Init):** Compute $\delta_3 = y_1 - t$.
3. **(Layer 3 Grad):** $\frac{\partial E}{\partial w_{11}^{(3)}} = \delta_3 z_2$
4. **Backward Pass (H2):** Compute $\delta_2 = (1 - z_2^2)w_{11}^{(3)}\delta_3$.
5. **(Layer 2 Grad):** $\frac{\partial E}{\partial w_{11}^{(2)}} = \delta_2 z_1$
6. **Backward Pass (H1):** Compute $\delta_1 = (1 - z_1^2)w_{11}^{(2)}\delta_2$.
7. **(Layer 1 Grad):** $\frac{\partial E}{\partial w_{11}^{(1)}} = \delta_1 x$

This shows the "chain" of error, starting at δ_3 and propagating backward to δ_2, then δ_1.

3.2 Backpropagation Example: A 2-3-2-1 Network

We now show a more complex example to illustrate the full backpropagation algorithm, especially how error signals are summed.

3.2.1 Network & Problem Setup

- **Architecture:** 2-3-2-1
- **Inputs:** $\mathbf{x} = (x_1, x_2)^T$. We also use bias inputs $x_0 = 1$.
- **Hidden Layer 1 (H1, $j = 1, 2, 3$):** 3 neurons.
  - Activation: $a_j^{(1)} = \sum_{i=0}^{2} w_{ji}^{(1)} x_i = w_{j1}^{(1)} x_1 + w_{j2}^{(1)} x_2 + w_{j0}^{(1)}$
  - Output: $z_j^{(1)} = \tanh(a_j^{(1)})$
- **Hidden Layer 2 (H2, $k = 1, 2$):** 2 neurons.
  - Activation: $a_k^{(2)} = \sum_{j=0}^{3} w_{kj}^{(2)} z_j^{(1)} = w_{k1}^{(2)} z_1^{(1)} + w_{k2}^{(2)} z_2^{(1)} + w_{k3}^{(2)} z_3^{(1)} + w_{k0}^{(2)}$
  - Output: $z_k^{(2)} = \sigma(a_k^{(2)})$ (Logistic Sigmoid)
- **Output Layer (O1, $l = 1$):** 1 neuron.
  - Activation: $a_l^{(3)} = \sum_{k=0}^{2} w_{lk}^{(3)} z_k^{(2)} = w_{11}^{(3)} z_1^{(2)} + w_{12}^{(3)} z_2^{(2)} + w_{10}^{(3)}$
  - Output: $y_1 = a_1^{(3)}$ (Identity function for regression)
- **Error Function:** Sum-of-squares error for a single pattern:

$$E = \frac{1}{2}(y_1 - t)^2 \tag{3.5}$$

Goal: Find the gradient $\frac{\partial E}{\partial w}$ for all weights in layers 3, 2, and 1.

Network Diagram

3.2.2 1. Forward Propagation

We assume a forward pass is completed, and all activation (a) and output (z, y) values are computed and stored.

3.2.3 2. Backward Propagation (The Derivatives)

We use the rule $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$ for all weights. The entire problem is to find the δ (error) for each neuron, starting from the output and moving backward.

Step 2a: Gradients for Layer 3 (Output)

Goal: Find $\frac{\partial E}{\partial w_{1k}^{(3)}}$ for $k \in \{0, 1, 2\}$. **Rule:** $\frac{\partial E}{\partial w_{1k}^{(3)}} = \delta_1^{(3)} z_k^{(2)}$ (where $z_0^{(2)} = 1$).

Calculate $\delta_1^{(3)}$: $\delta_1^{(3)}$ is the error w.r.t. the output activation $a_1^{(3)}$.

$$\delta_1^{(3)} \equiv \frac{\partial E}{\partial a_1^{(3)}} = \frac{\partial E}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_1^{(3)}}$$

- $\frac{\partial E}{\partial y_1} = \frac{\partial}{\partial y_1}\left[\frac{1}{2}(y_1 - t)^2\right] = (y_1 - t)$
- $\frac{\partial y_1}{\partial a_1^{(3)}} = \frac{\partial}{\partial a_1^{(3)}}[a_1^{(3)}] = 1$ (Identity function)

$$\delta_1^{(3)} = (y_1 - t) \tag{3.6}$$

Final Gradients (Layer 3):

$$\frac{\partial E}{\partial w_{11}^{(3)}} = \delta_1^{(3)} z_1^{(2)} = (y_1 - t) z_1^{(2)}$$
$$\frac{\partial E}{\partial w_{12}^{(3)}} = \delta_1^{(3)} z_2^{(2)} = (y_1 - t) z_2^{(2)}$$
$$\frac{\partial E}{\partial w_{10}^{(3)}} = \delta_1^{(3)} \cdot 1 = (y_1 - t)$$

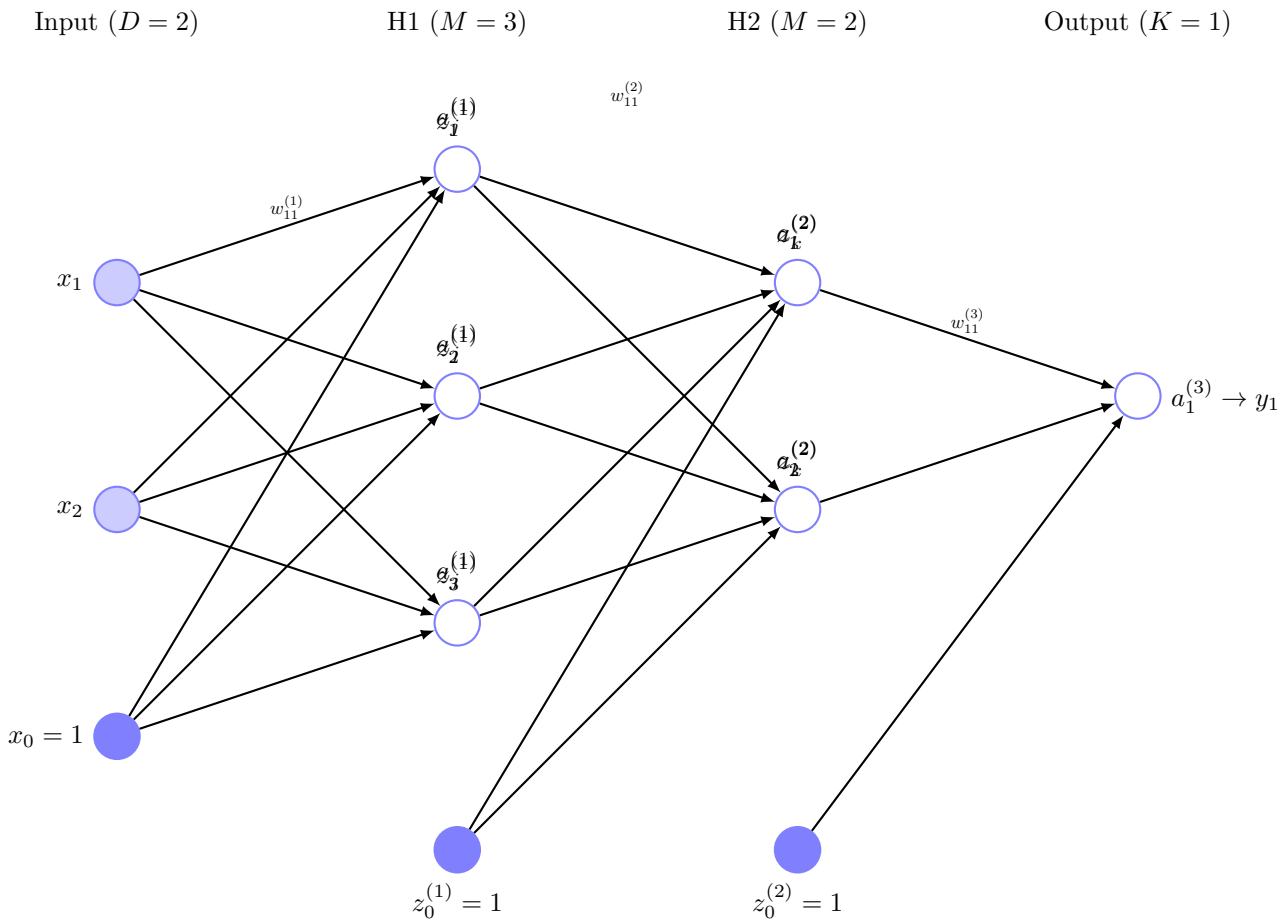


Figure 3.1: A 2-3-2-1 feed-forward network with bias units for each layer.

Step 2b: Gradients for Layer 2 (H2)

Goal: Find $\frac{\partial E}{\partial w_{kj}^{(2)}}$ for $k \in \{1, 2\}$, $j \in \{0, 1, 2, 3\}$. **Rule:** $\frac{\partial E}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} z_j^{(1)}$ (where $z_0^{(1)} = 1$).

Calculate $\delta_k^{(2)}$ (for $k = 1, 2$): This is the error $\delta_k^{(2)} = \frac{\partial E}{\partial a_k^{(2)}}$. We use the backpropagation formula (Eq. 5.56):

$$\delta_k^{(2)} = h'(a_k^{(2)}) \sum_l w_{lk}^{(3)} \delta_l^{(3)}$$

- $h'(a_k^{(2)})$ is the derivative of H2's activation function, $z_k^{(2)} = \sigma(a_k^{(2)})$. The derivative is $\sigma'(a) = \sigma(a)(1 - \sigma(a)) = z_k^{(2)}(1 - z_k^{(2)})$.
- The sum $\sum_l$ is over all units in the *next* layer (O1) that this unit connects to. Here, the sum is just over $l = 1$.

For $\delta_1^{(2)}$ (H2 neuron 1):

$$\delta_1^{(2)} = \left(z_1^{(2)}(1 - z_1^{(2)})\right) \cdot \left(w_{11}^{(3)} \delta_1^{(3)}\right)$$

For $\delta_2^{(2)}$ (H2 neuron 2):

$$\delta_2^{(2)} = \left(z_2^{(2)}(1 - z_2^{(2)})\right) \cdot \left(w_{12}^{(3)} \delta_1^{(3)}\right)$$

Final Gradients (Layer 2): The 8 gradients are calculated by multiplying the δ of the neuron ($k = 1$ or $k = 2$) by the input $z_j^{(1)}$ that feeds into it.

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \delta_1^{(2)} z_1^{(1)} \qquad \frac{\partial E}{\partial w_{21}^{(2)}} = \delta_2^{(2)} z_1^{(1)}$$
$$\frac{\partial E}{\partial w_{12}^{(2)}} = \delta_1^{(2)} z_2^{(1)} \qquad \frac{\partial E}{\partial w_{22}^{(2)}} = \delta_2^{(2)} z_2^{(1)}$$
$$\frac{\partial E}{\partial w_{13}^{(2)}} = \delta_1^{(2)} z_3^{(1)} \qquad \frac{\partial E}{\partial w_{23}^{(2)}} = \delta_2^{(2)} z_3^{(1)}$$
$$\frac{\partial E}{\partial w_{10}^{(2)}} = \delta_1^{(2)} \cdot 1 \qquad \frac{\partial E}{\partial w_{20}^{(2)}} = \delta_2^{(2)} \cdot 1$$

Step 2c: Gradients for Layer 1 (H1)

Goal: Find $\frac{\partial E}{\partial w_{ji}^{(1)}}$ for $j \in \{1,2,3\}$, $i \in \{0,1,2\}$. **Rule:** $\frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} x_i$ (where $x_0 = 1$).

Calculate $\delta_j^{(1)}$ (for $j = 1,2,3$): This is where the error **sums up**.

$$\delta_j^{(1)} = h'(a_j^{(1)}) \sum_k w_{kj}^{(2)} \delta_k^{(2)}$$

- $h'(a_j^{(1)})$ is the derivative of H1's activation function, $z_j^{(1)} = \tanh(a_j^{(1)})$. The derivative is $1 - \tanh^2(a_j^{(1)}) = 1 - (z_j^{(1)})^2$.
- The sum $\sum_k$ is over all units in the *next* layer (H2) that this unit connects to. Here, the sum is over $k = 1, 2$.

For $\delta_1^{(1)}$ (H1 neuron 1): This neuron ($j = 1$) connects to H2 neuron 1 (via $w_{11}^{(2)}$) and H2 neuron 2 (via $w_{21}^{(2)}$). We sum the error contributions from both.

$$\delta_1^{(1)} = \left(1 - (z_1^{(1)})^2\right) \cdot \left(w_{11}^{(2)} \delta_1^{(2)} + w_{21}^{(2)} \delta_2^{(2)}\right)$$

For $\delta_2^{(1)}$ (H1 neuron 2):

$$\delta_2^{(1)} = \left(1 - (z_2^{(1)})^2\right) \cdot \left(w_{12}^{(2)} \delta_1^{(2)} + w_{22}^{(2)} \delta_2^{(2)}\right)$$

For $\delta_3^{(1)}$ (H1 neuron 3):

$$\delta_3^{(1)} = \left(1 - (z_3^{(1)})^2\right) \cdot \left(w_{13}^{(2)} \delta_1^{(2)} + w_{23}^{(2)} \delta_2^{(2)}\right)$$

Final Gradients (Layer 1): The 9 gradients are calculated by multiplying the δ of the neuron ($j = 1, 2, \text{or } 3$) by the input x_i that feeds into it.

$$\begin{array}{lll}
\frac{\partial E}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1 & \frac{\partial E}{\partial w_{21}^{(1)}} = \delta_2^{(1)} x_1 & \frac{\partial E}{\partial w_{31}^{(1)}} = \delta_3^{(1)} x_1 \\
\frac{\partial E}{\partial w_{12}^{(1)}} = \delta_1^{(1)} x_2 & \frac{\partial E}{\partial w_{22}^{(1)}} = \delta_2^{(1)} x_2 & \frac{\partial E}{\partial w_{32}^{(1)}} = \delta_3^{(1)} x_2 \\
\frac{\partial E}{\partial w_{10}^{(1)}} = \delta_1^{(1)} \cdot 1 & \frac{\partial E}{\partial w_{20}^{(1)}} = \delta_2^{(1)} \cdot 1 & \frac{\partial E}{\partial w_{30}^{(1)}} = \delta_3^{(1)} \cdot 1
\end{array}$$

3.2.4 3. Summary of the Algorithm

1. **Forward Pass:** Compute and store all a and z values, and the final output y_1.
2. **Backward Pass:**
   (a) **Layer 3:** Compute $\delta_1^{(3)} = y_1 - t$.
   (b) **Layer 2:** Compute $\delta_1^{(2)}$ and $\delta_2^{(2)}$ using $\delta_1^{(3)}$ and weights $w^{(3)}$.
   (c) **Layer 1:** Compute $\delta_1^{(1)}$, $\delta_2^{(1)}$, and $\delta_3^{(1)}$ using $\delta_1^{(2)}$, $\delta_2^{(2)}$, and weights $w^{(2)}$.
3. **Gradient Calculation:** For every weight w_{ji}, multiply the δ_j at the output of the weight by the z_i at the input of the weight.

3.3 The General Rules of Backpropagation

The entire algorithm for calculating the gradient ∇E_n (for a single data point n) is built on two core rules, which are applied repeatedly.

We first run a **forward pass** to compute all unit activations a_j and outputs z_j, which are stored.

Then, we run a **backward pass** to compute the "error signal" δ_j for every unit.

Definition 3.1 (The Error Signal, δ_j)**.** The error signal δ_j for a unit j is defined as the partial derivative of the total error E_n with respect to the *activation* (the weighted sum) a_j of that unit.

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{3.7}$$

3.3.1 Rule 1: Calculating the δ for an Output Unit

(The Starting Point)

This rule is how we *start* the backpropagation. The error δ_k for an output unit k is found by taking the derivative of the error function E_n with respect to the output activation a_k.

$$\delta_k = \frac{\partial E_n}{\partial a_k} \tag{3.8}$$

For the "canonical" error/activation pairs we've discussed, this derivative is always:

$$\delta_k = y_k - t_k \tag{3.9}$$

- This is true for regression (Identity $y_k = a_k$ + Sum-of-Squares error).
- This is true for binary classification (Sigmoid $y_k = \sigma(a_k)$ + Cross-Entropy error).
- This is true for multiclass classification (Softmax + Cross-Entropy error).

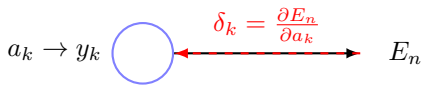


Figure 3.2: Rule 1: The error signal δ_k for an output unit is the derivative of the final error E_n with respect to the unit's own activation a_k.

3.3.2 Rule 2: Calculating the δ for a Hidden Unit

(The "Join" or "Split" Rule)

This is the core recursive step. It tells us how to find the error δ_j for a hidden unit j *given the δ_k's from the layer in front of it*.

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{3.10}$$

This rule has two parts. To find the error δ_j for hidden unit j:

1. **Sum the Propagated Errors:** $\sum_k w_{kj}\delta_k$ This is the "join" part of the backward pass, which corresponds to the "split" in the forward pass.

   A hidden unit j "splits" its output and influences the error E_n through *all* the units k it is connected to in the next layer.

   To find its total error, δ_j, we must "join" all these error contributions. We sum the error signals δ_k from the next layer, but we weight each one by the w_{kj} that connects j to k. This is the multivariate chain rule in action.

2. **Multiply by Local Derivative:** $h'(a_j) \cdot (\dots)$ We multiply the total propagated error by the derivative of our *local* activation function, $h'(a_j)$. This scales the error, accounting for the hidden unit's own contribution.

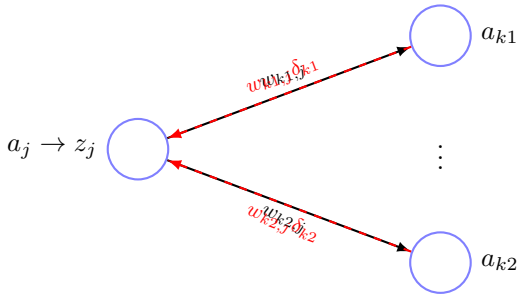


Figure 3.3: Rule 2: The error δ_j is the **sum** of all incoming error signals from the next layer ($\sum_k w_{kj}\delta_k$), multiplied by the local derivative $h'(a_j)$.

3.3.3 Rule 3: The Gradient Calculation Rule

(Finding the Final Answer)

This is the rule we saw in Eq. (5.53). Once we have computed all the δ_j values (using Rule 1 and Rule 2), we can find the gradient for any weight w_{ji} in the network.

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{3.11}$$

This is the "local computation." The gradient for the weight w_{ji} (connecting unit i to unit j) is simply:

$$\text{(Error at destination unit } j\text{)} \times \text{(Output from source unit } i\text{)}$$

(Note: z_i is the output of unit i. If i is an input unit, z_i is just the input x_i).

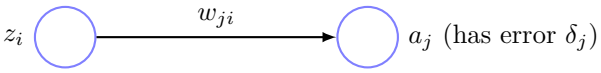


Figure 3.4: Rule 3: The gradient for the weight w_{ji} is the product of the output from the source unit (z_i) and the error signal at the destination unit (δ_j).

3.3.4 Summary of the Backpropagation Algorithm

This gives us the complete four-step algorithm:

1. **Forward Pass:** Give the network an input $\mathbf{x}_n$. Compute and store all unit activations a_j and outputs z_j (and final output y_k).
2. **Output δ:** For all output units k, calculate δ_k using **Rule 1**.
3. **Backpropagate δ:** For each hidden layer, moving from output to input, compute the δ_j for each hidden unit using **Rule 2**.
4. **Calculate Gradients:** For every weight w_{ji} in the network, calculate the gradient using **Rule 3**.

3.4 Backpropagation Practice Questions

This section contains sample questions for calculating gradients in a neural network. Solutions are to be compiled later.

Question 1 (2-2-1 Regression Network). Consider a 2-2-1 neural network for a regression task.

- **Inputs:** $\mathbf{x} = [x_1, x_2]^T = [1.0, 0.5]^T$.
- **Hidden Layer (2 neurons, $j = 1, 2$):**
  - Activation: $a_j^{(1)} = w_{j1}^{(1)} x_1 + w_{j2}^{(1)} x_2 + w_{j0}^{(1)}$
  - Output: $z_j = \tanh(a_j^{(1)})$
- **Output Layer (1 neuron, $k = 1$):**
  - Activation: $a_k^{(2)} = w_{k1}^{(2)} z_1 + w_{k2}^{(2)} z_2 + w_{k0}^{(2)}$

  - Output: $y_k = a_k^{(2)}$ (Identity function)

- **Target:** $t = 1.0$
- **Error Function:** $E = \frac{1}{2}(y_1 - t)^2$
- **Weights (Layer 1):**
  - $w_{11}^{(1)} = 0.5$, $w_{12}^{(1)} = 0.2$, $w_{10}^{(1)} = 0.1$
  - $w_{21}^{(1)} = -0.3$, $w_{22}^{(1)} = 0.8$, $w_{20}^{(1)} = 0.0$
- **Weights (Layer 2):**
  - $w_{11}^{(2)} = 1.0$, $w_{12}^{(2)} = -0.5$, $w_{10}^{(2)} = 0.2$

Task:

1. Perform a full forward pass and calculate the final output y_1 and the error E.
2. Perform a full backward pass, calculating $\delta_k^{(2)}$ and $\delta_j^{(1)}$ for all units.
3. Calculate the gradient $\frac{\partial E}{\partial w_{11}^{(2)}}$ and $\frac{\partial E}{\partial w_{11}^{(1)}}$.

(Note: You will need the derivative $f'(a)$ for $f(a) = \tanh(a)$, which is $1 - \tanh^2(a)$.)

Question 2 (2-2-1 Binary Classification Network). Consider a 2-2-1 network for a binary classification task.

- **Inputs:** $\mathbf{x} = [x_1, x_2]^T = [1.0, 2.0]^T$.
- **Hidden Layer (2 neurons, $j = 1, 2$):**
  - Activation: $a_j^{(1)} = w_{j1}^{(1)} x_1 + w_{j2}^{(1)} x_2 + w_{j0}^{(1)}$
  - Output: $z_j = \sigma(a_j^{(1)})$ (Logistic Sigmoid)
- **Output Layer (1 neuron, $k = 1$):**
  - Activation: $a_k^{(2)} = w_{k1}^{(2)} z_1 + w_{k2}^{(2)} z_2 + w_{k0}^{(2)}$
  - Output: $y_k = \sigma(a_k^{(2)})$ (Logistic Sigmoid)
- **Target:** $t = 1$
- **Error Function:** Cross-Entropy, $E = -[t \ln y_1 + (1 - t) \ln(1 - y_1)]$
- **Weights (Layer 1):**
  - $w_{11}^{(1)} = 0.1$, $w_{12}^{(1)} = -0.2$, $w_{10}^{(1)} = 0.0$
  - $w_{21}^{(1)} = 0.4$, $w_{22}^{(1)} = 0.3$, $w_{20}^{(1)} = -0.1$
- **Weights (Layer 2):**
  - $w_{11}^{(2)} = 0.5$, $w_{12}^{(2)} = -0.4$, $w_{10}^{(2)} = 0.2$

Task:

1. Perform a full forward pass and calculate the final prediction y_1.
2. Calculate the error E.
3. Perform a full backward pass, calculating $\delta_k^{(2)}$ and $\delta_j^{(1)}$ for all units. (Hint: δ for the output layer is $y_1 - t$).
4. Calculate the gradient for all 6 weights in the first layer: $\frac{\partial E}{\partial w_{ji}^{(1)}}$.

(Note: You will need the derivative $f'(a)$ for $f(a) = \sigma(a)$, which is $\sigma(a)(1 - \sigma(a))$.)

Question 3 (Error Splitting and Summing). Consider the 2-3-2-1 network from our notes (Section 5.3.2). Assume that after a forward and partial backward pass, you have computed the following values:

- **From Layer 2 (H2):**
  - $\delta_1^{(2)} = 0.5$ (Error signal for H2 neuron 1)

  - $\delta_2^{(2)} = -0.2$ (Error signal for H2 neuron 2)

- **Weights from H1 to H2:**
  - $w_{12}^{(2)} = 0.8$ (Weight from H1-neuron 2 to H2-neuron 1)
  - $w_{22}^{(2)} = 0.1$ (Weight from H1-neuron 2 to H2-neuron 2)
- **From Layer 1 (H1):**
  - $z_2^{(1)} = 0.7$ (Output of H1-neuron 2)
  - $a_2^{(1)} = 0.887$ (Activation of H1-neuron 2, $\tanh(0.887) \approx 0.7$)
- **Activation Function:** H1 uses $h(a) = \tanh(a)$.

Task:

1. Calculate the error signal $\delta_2^{(1)}$ (for H1-neuron 2). You must show how the δ signals from Layer H2 are combined.
2. Using your result from (1), calculate the gradient $\frac{\partial E}{\partial w_{21}^{(1)}}$ assuming the input was $x_1 = 1.5$.

Chapter 4

Regularization In Neural Networks

4.1 Introduction

The number of hidden units, M, is a free parameter that controls the total number of parameters (weights and biases) in the network. In a maximum likelihood setting, we might expect an optimal value of M that gives the best generalization by balancing under-fitting and over-fitting.

However, the generalization error is not a simple function of M, because the error function $E(\mathbf{w})$ is non-convex and has many local minima. As shown in Figure 5.10, training a network with a given M from different random initializations can lead to different solutions with different validation errors. A practical approach to choosing M is to train several networks for various M values, each with multiple random initializations, and pick the specific model with the best validation performance.

An alternative and often preferred approach is to choose a relatively large value for M and then control the model's complexity by adding a **regularization term** to the error function.

4.1.1 Weight Decay

The simplest regularizer is a quadratic term, also known as **weight decay**. The new regularized error function $\tilde{E}(\mathbf{w})$ is:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} \tag{4.1}$$

where $E(\mathbf{w})$ is the original error function (e.g., sum-of-squares or cross-entropy) and λ is the **regularization coefficient** that controls the trade-off between the original error and the penalty term.

This regularizer can be interpreted from a probabilistic (Bayesian) perspective as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$. The effective complexity of the model is then governed by the choice of λ.

Proposition 4.1. *The weight decay regularization term $\frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ is mathematically equivalent to the negative logarithm of a zero-mean Gaussian prior distribution on the weights $\mathbf{w}$.*

Proof. This is a central concept in the Bayesian interpretation of machine learning, known as MAP (Maximum A Posteriori) estimation.

1. Define the Goal (MAP Estimation) In a Bayesian framework, we don't just maximize the likelihood $p(\mathcal{D}|\mathbf{w})$; we maximize the **posterior probability** $p(\mathbf{w}|\mathcal{D})$. Using Bayes' theorem:

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})}$$

Since $p(\mathcal{D})$ is a constant, maximizing the posterior is equivalent to maximizing the numerator:

$$\mathbf{w}_{\text{MAP}} = \arg\max_{\mathbf{w}} [p(\mathcal{D}|\mathbf{w})p(\mathbf{w})]$$

This is equivalent to maximizing the log of the posterior, or *minimizing* its negative log:

$$\mathbf{w}_{\text{MAP}} = \arg\min_{\mathbf{w}} [-\ln p(\mathcal{D}|\mathbf{w}) - \ln p(\mathbf{w})]$$

We recognize the two terms:

- $-\ln p(\mathcal{D}|\mathbf{w}) = E(\mathbf{w})$, the original error function (e.g., sum-of-squares).
- $-\ln p(\mathbf{w})$ is the **regularization term**.

So, the regularized error is $\tilde{E}(\mathbf{w}) = E(\mathbf{w}) - \ln p(\mathbf{w})$.

2. Define the Prior Distribution Let's define our prior $p(\mathbf{w})$ as a zero-mean Gaussian distribution with a precision (inverse variance) parameter α. The covariance matrix is $\alpha^{-1}\mathbf{I}$.

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$$

The formula for a W-dimensional multivariate Gaussian is:

$$\begin{aligned} p(\mathbf{w}|\alpha) &= \frac{1}{(2\pi)^{W/2}|\alpha^{-1}\mathbf{I}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{w}-\mathbf{0})^T(\alpha^{-1}\mathbf{I})^{-1}(\mathbf{w}-\mathbf{0})\right) \\ &= \frac{1}{(2\pi/\alpha)^{W/2}} \exp\left(-\frac{1}{2}\mathbf{w}^T(\alpha\mathbf{I})\mathbf{w}\right) \\ &= \left(\frac{\alpha}{2\pi}\right)^{W/2} \exp\left(-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right) \end{aligned}$$

3. Take the Negative Logarithm Now we take the negative logarithm of this prior:

$$\begin{aligned} -\ln p(\mathbf{w}|\alpha) &= -\ln\left[\left(\frac{\alpha}{2\pi}\right)^{W/2} \exp\left(-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right)\right] \\ &= -\left[\ln\left(\left(\frac{\alpha}{2\pi}\right)^{W/2}\right) + \ln\left(\exp\left(-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right)\right)\right] \\ &= -\left[\frac{W}{2}\ln\left(\frac{\alpha}{2\pi}\right) - \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right] \\ &= \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} - \frac{W}{2}\ln\left(\frac{\alpha}{2\pi}\right) \end{aligned}$$

This simplifies to:

$$-\ln p(\mathbf{w}|\alpha) = \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + \text{Constant}$$

The constant term does not depend on $\mathbf{w}$, so it does not affect the optimization.

4. Conclusion The total error we minimize is:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) - \ln p(\mathbf{w}) = E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + \text{Constant}$$

This is identical in form to the weight decay regularized error:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

Therefore, the regularization term $\frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ is the negative log of a zero-mean Gaussian prior, where the regularization coefficient λ is equivalent to the prior's precision α. □

4.2 Consistent Gaussian Priors (and limitations of Weight Decay)

A significant limitation of the simple weight decay regularizer (Eq. 5.112) is that it is **inconsistent with the scaling properties** of the network. A consistent model should not favor one network over another equivalent network that just uses rescaled inputs or outputs.

To demonstrate this, we first define a two-layer network with linear output units:

$$z_j = h\left(\sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) = h(a_j) \tag{4.2}$$

$$y_k = \sum_j w_{kj}^{(2)} z_j + w_{k0}^{(2)} = a_k \tag{4.3}$$

4.2.1 Invariance to Input Transformations

First, consider a linear transformation of the input variables:

$$x_i \to \tilde{x}_i = a x_i + b \tag{4.4}$$

We can find a corresponding transformation of the first-layer weights and biases that leaves the network's final output y_k completely unchanged.

Proposition 4.2. *The hidden unit outputs z_j (and thus the final output y_k) are unchanged by the input transformation (5.115) if the first-layer weights and biases are transformed as follows:*

$$w_{ji}^{(1)} \to \tilde{w}_{ji}^{(1)} = \frac{1}{a} w_{ji}^{(1)} \tag{4.5}$$

$$w_{j0}^{(1)} \to \tilde{w}_{j0}^{(1)} = w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)} \tag{4.6}$$

Proof. We compute the new hidden unit activation $\tilde{a}_j$ using the transformed inputs and weights.

$$\tilde{a}_j = \sum_i \tilde{w}_{ji}^{(1)} \tilde{x}_i + \tilde{w}_{j0}^{(1)}$$

Substitute the transformations:

$$\tilde{a}_j = \sum_i \left(\frac{1}{a} w_{ji}^{(1)} \right) (a x_i + b) + \left(w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)} \right)$$

Expand the first term:

$$\tilde{a}_j = \sum_i \left(\frac{1}{a} w_{ji}^{(1)} a x_i + \frac{1}{a} w_{ji}^{(1)} b \right) + w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)}$$

$$\tilde{a}_j = \sum_i (w_{ji}^{(1)} x_i) + \sum_i \left(\frac{b}{a} w_{ji}^{(1)} \right) + w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)}$$

Group the terms:

$$\tilde{a}_j = \left(\sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + \left(\frac{b}{a} \sum_i w_{ji}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)} \right)$$

The second group cancels to zero:

$$\tilde{a}_j = \sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

$$\tilde{a}_j = a_j$$

Since the hidden unit activation a_j is unchanged, its output $z_j = h(a_j)$ is also unchanged. Because all z_j are unchanged, the final network output y_k (which depends only on z_j) is also unchanged. □

4.2.2 Invariance to Output Transformations

Second, consider a linear transformation of the output variables:

$$y_k \to \tilde{y}_k = c y_k + d \tag{4.7}$$

We can achieve this transformation by transforming the second-layer weights and biases.

Proposition 4.3. *We can produce the transformed output $\tilde{y}_k$ by transforming the second-layer weights and biases as follows:*

$$w_{kj}^{(2)} \to \tilde{w}_{kj}^{(2)} = c w_{kj}^{(2)} \tag{4.8}$$

$$w_{k0}^{(2)} \to \tilde{w}_{k0}^{(2)} = c w_{k0}^{(2)} + d \tag{4.9}$$

Proof. We compute the new output $\tilde{y}_k$ from the new second-layer weights, using the *original* hidden unit outputs z_j

(which we assume are fixed).

$$\tilde{y}_k = \sum_j \tilde{w}_{kj}^{(2)} z_j + \tilde{w}_{k0}^{(2)}$$

Substitute the transformations:

$$\tilde{y}_k = \sum_j (c w_{kj}^{(2)}) z_j + (c w_{k0}^{(2)} + d)$$

Factor out the constant c :

$$\tilde{y}_k = c \left(\sum_j w_{kj}^{(2)} z_j \right) + c w_{k0}^{(2)} + d$$

$$\tilde{y}_k = c \left(\sum_j w_{kj}^{(2)} z_j + w_{k0}^{(2)} \right) + d$$

The term in parentheses is the original output y_k :

$$\tilde{y}_k = c y_k + d$$

This proves that the new network computes the desired transformed output. □

Remark 4.4 (Why Simple Weight Decay Fails). If we use simple weight decay $\tilde{E} = E + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$, the regularization term $\frac{\lambda}{2}\sum w^2$ sums over *all* weights and biases. However, the transformations (e.g., 5.117 and 5.120) for the biases are **shifts**, not scales. A bias $w_{j0}^{(1)}$ changes to $w_{j0}^{(1)} -$ constant. The $\mathbf{w}^T\mathbf{w}$ regularizer is *not* invariant to this shift. It will assign a different penalty to the two equivalent networks, arbitrarily favoring one over the other. A consistent regularizer should not penalize biases, or at least should treat them separately.

4.2.3 Weight Decay lacks Invariance

Proposition 4.5. *The simple weight decay regularizer $R(\mathbf{w}) = \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ is not invariant under the linear transformations (5.115) and (5.118).*

Proof. The total regularization penalty $R(\mathbf{w})$ is the sum of the squared magnitudes of all weights and biases in the network.

$$R(\mathbf{w}) = \frac{\lambda}{2} \left(\sum_{j,i} (w_{ji}^{(1)})^2 + \sum_j (w_{j0}^{(1)})^2 + \sum_{k,j} (w_{kj}^{(2)})^2 + \sum_k (w_{k0}^{(2)})^2 \right)$$

We now find the penalty $R(\tilde{\mathbf{w}})$ for an equivalent network and show $R(\tilde{\mathbf{w}}) \neq R(\mathbf{w})$.

Case 1: Input Transformation ($x_i \to a x_i + b$) From (5.116) and (5.117), the new first-layer weights $\tilde{\mathbf{w}}^{(1)}$ are:

$$\tilde{w}_{ji}^{(1)} = \frac{1}{a} w_{ji}^{(1)}$$

$$\tilde{w}_{j0}^{(1)} = w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)}$$

The second-layer weights are unchanged. The new penalty term for the first-layer weights is:

$$R(\tilde{\mathbf{w}}^{(1)}) = \frac{\lambda}{2} \left(\sum_{j,i} (\tilde{w}_{ji}^{(1)})^2 + \sum_j (\tilde{w}_{j0}^{(1)})^2 \right)$$

Let's analyze the bias term $\sum_j (\tilde{w}_{j0}^{(1)})^2$:

$$\sum_j (\tilde{w}_{j0}^{(1)})^2 = \sum_j \left(w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)} \right)^2$$

Unless $b = 0$, this new term is completely different. The transformation (5.117) is a **shift**, not a **scale**.

$$\sum_j \left(w_{j0}^{(1)} - \frac{b}{a} \sum_i w_{ji}^{(1)} \right)^2 \neq \sum_j (w_{j0}^{(1)})^2$$

Because the penalty for the biases changes, the total penalty $R(\tilde{\mathbf{w}})$ for the transformed network is different from the original penalty $R(\mathbf{w})$. The regularizer is not invariant to this transformation.

Case 2: Output Transformation ($y_k \to cy_k + d$) From (5.119) and (5.120), the new second-layer weights $\tilde{\mathbf{w}}^{(2)}$ are:

$$\tilde{w}_{kj}^{(2)} = cw_{kj}^{(2)}$$
$$\tilde{w}_{k0}^{(2)} = cw_{k0}^{(2)} + d$$

The new penalty term for the second-layer biases is:

$$\sum_k (\tilde{w}_{k0}^{(2)})^2 = \sum_k (cw_{k0}^{(2)} + d)^2$$

Unless $d = 0$, this transformation is also a shift.

$$\sum_k (cw_{k0}^{(2)} + d)^2 \neq \sum_k (w_{k0}^{(2)})^2$$

Again, the total penalty $R(\tilde{\mathbf{w}})$ is not equal to $R(\mathbf{w})$.

Conclusion: Simple weight decay, which treats all weights and biases identically, is not invariant. It arbitrarily favors one set of parameters over another, even if both networks compute the exact same function (or a simple linear transformation of it). This is a sign of a poorly chosen prior. □

4.3 The Right One.....

The simple weight decay regularizer $R(\mathbf{w}) = \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ is inconsistent because it is not invariant to the simple shift and scaling transformations that produce equivalent networks. We must "engineer" a regularizer that respects these invariances.

4.3.1 Logical Deduction of a Consistent Regularizer

We arrive at the correct regularizer by fixing the problems we identified.

Problem 1: The Bias Invariance

- **Problem:** The transformations for bias terms are **shifts** (e.g., $\tilde{w}_{k0}^{(2)} = cw_{k0}^{(2)} + d$).
- **Analysis:** A quadratic penalty $R_{bias} = \frac{\lambda}{2}(w_{k0}^{(2)})^2$ is *not* invariant to a shift. The penalty for the new bias $\tilde{w}_{k0}^{(2)}$ would be $\frac{\lambda}{2}(cw_{k0}^{(2)} + d)^2$, which is completely different.
- **Solution:** The simplest way to make a penalty invariant to a shift is to **not penalize the shifted term at all**.
- **Conclusion 1:** A consistent regularizer should omit the bias terms.

Problem 2: The Weight Invariance

- **Problem:** The transformations for weight terms are **scalings**, and they scale differently for each layer (e.g., $\tilde{w}_{ji}^{(1)} = \frac{1}{a}w_{ji}^{(1)}$ and $\tilde{w}_{kj}^{(2)} = cw_{kj}^{(2)}$).
- **Analysis:** A single penalty λ for all weights is not invariant. The penalty for the first layer would change by $\frac{1}{a^2}$, while the penalty for the second layer would change by c^2.
- **Solution:** Since the weights in each layer (let's call them $\mathcal{W}_1, \mathcal{W}_2, \dots$) scale differently, they must be treated as separate groups.
- **Conclusion 2:** A consistent regularizer must have a separate regularization coefficient $(\lambda_1, \lambda_2, \dots)$ for each layer's weight group.

4.3.2 The Generalised Regularizer

By combining these two solutions, we arrive at the form of a consistent regularizer for a general network with L layers. We define L separate groups of weights $\mathcal{W}_k$ (for $k = 1 \dots L$), which explicitly exclude all bias terms.

The regularized error function is:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \underbrace{\frac{\alpha_1}{2}\sum_{w\in\mathcal{W}_1} w^2}_{\text{Layer 1 Penalty}} + \underbrace{\frac{\alpha_2}{2}\sum_{w\in\mathcal{W}_2} w^2}_{\text{Layer 2 Penalty}} + \cdots + \underbrace{\frac{\alpha_L}{2}\sum_{w\in\mathcal{W}_L} w^2}_{\text{Layer L Penalty}} \tag{4.10}$$

This can be written compactly as:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{1}{2}\sum_{k=1}^{L} \alpha_k ||\mathbf{w}||_k^2 \tag{4.11}$$

where $||\mathbf{w}||_k^2 = \sum_{j \in \mathcal{W}_k} w_j^2$ is the sum of squared weights *only* in layer k, and α_k is the regularization coefficient for that layer.

This corresponds to a Gaussian prior where the weights for each layer are drawn from their own zero-mean Gaussian with its own precision α_k, and the biases are given an improper (non-informative) prior.

4.4 Early Stopping

4.4.1 Definition

Early Stopping is a practical and widely used form of regularization that operates as a **stopping criterion** rather than as a modification to the error function.

The procedure is as follows:

1. **Split Data:** The training data is split into a *training set* and a *validation set*.
2. **Train:** The network parameters $\mathbf{w}$ are optimized by minimizing the error $E(\mathbf{w})$ on the *training set*.
3. **Monitor:** After each iteration τ (or every few iterations), the error of the current network $\mathbf{w}^{(\tau)}$ is evaluated on the *validation set*.
4. **Stop:** As training progresses, the training error will almost always decrease. The validation error, however, will typically decrease at first and then, as the network begins to over-fit, it will hit a minimum and start to increase. The training is halted at this "sweet spot"—the iteration τ that corresponds to the lowest validation set error.

4.4.2 Significance as a Regularizer

The key idea is that the optimizer (like gradient descent) learns the "simple," large-scale structures of the error surface first. Only in later iterations does it begin to fit the "complex" noise in the data.

By stopping early, we prevent the weights from moving into these complex, over-fitted configurations. The number of training iterations τ itself acts as a regularization parameter, similar to λ in weight decay.

4.4.3 Mathematical Justification (Proof of Equivalence)

We can prove this equivalence by analyzing the behavior of gradient descent on the **local quadratic approximation** of the error function near a minimum.

Setup

1. **Error Function:** We approximate the error near a minimum $\mathbf{w}^*$ as a quadratic bowl:

$$E(\mathbf{w}) \simeq E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T H (\mathbf{w} - \mathbf{w}^*) \tag{4.12}$$

2. **Optimizer:** We use the batch gradient descent update rule:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{4.13}$$

3. **Gradient:** The gradient of our quadratic error function is:

$$\nabla E(\mathbf{w}) = H(\mathbf{w} - \mathbf{w}^*) \tag{4.14}$$

Analysis of Gradient Descent Update

We analyze the optimization by changing to a coordinate system aligned with the eigenvectors $\mathbf{u}_j$ of the Hessian H. In this system, H is a diagonal matrix with eigenvalues λ_j. The update for the j-th component of the weight vector, $w_j = \mathbf{w}^T \mathbf{u}_j$, becomes decoupled:

$$w_j^{(\tau+1)} = w_j^{(\tau)} - \eta \lambda_j (w_j^{(\tau)} - w_j^*)$$

This is a recurrence relation. If we assume training starts at $\mathbf{w}^{(0)} = \mathbf{0}$ (so $w_j^{(0)} = 0$), we can solve for w_j after τ steps.

Let's track the difference from the minimum: $\Delta_j^{(\tau)} = w_j^{(\tau)} - w_j^*$. The update rule for the difference is:

$$\Delta_j^{(\tau+1)} + w_j^* = (\Delta_j^{(\tau)} + w_j^*) - \eta \lambda_j \Delta_j^{(\tau)} \implies \Delta_j^{(\tau+1)} = (1 - \eta\lambda_j)\Delta_j^{(\tau)}$$

After τ steps, the solution is $\Delta_j^{(\tau)} = (1 - \eta\lambda_j)^\tau \Delta_j^{(0)}$. The initial difference is $\Delta_j^{(0)} = w_j^{(0)} - w_j^* = -w_j^*$.

$$w_j^{(\tau)} - w_j^* = (1 - \eta\lambda_j)^\tau (-w_j^*)$$

Rearranging gives the weight value for the **early-stopped solution**:

$$w_j^{(\tau)} = w_j^* \left[1 - (1 - \eta\lambda_j)^\tau\right] \tag{4.15}$$

Analysis of Weight Decay Solution

Now we find the solution for training with a weight decay regularizer, $\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$. The regularized error is:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

We train this to convergence, i.e., $\nabla\tilde{E}(\mathbf{w}) = 0$.

$$\begin{aligned} \nabla\left(E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right) &= 0 \\ \nabla E(\mathbf{w}) + \alpha\mathbf{w} &= 0 \\ H(\mathbf{w} - \mathbf{w}^*) + \alpha\mathbf{w} &= 0 \end{aligned}$$

Solving for the j-th component w_j^{reg} of this regularized solution:

$$\begin{aligned} \lambda_j(w_j^{reg} - w_j^*) + \alpha w_j^{reg} &= 0 \\ (\lambda_j + \alpha)w_j^{reg} &= \lambda_j w_j^* \\ w_j^{reg} &= w_j^*\left[\frac{\lambda_j}{\lambda_j + \alpha}\right] \end{aligned}$$

This is the final **regularized solution**.

4.4.4 Conclusion: A Comparison

We have found two solutions for the j-th weight component:

1. **Early-Stopped Solution (after τ steps):** $w_j^{(\tau)} = w_j^*\left[1 - (1 - \eta\lambda_j)^\tau\right]$
2. **Regularized Solution (at convergence):** $w_j^{reg} = w_j^*\left[\frac{\lambda_j}{\lambda_j + \alpha}\right]$

These two equations have the exact same qualitative behavior:

- **When λ_j is large** (high curvature, "simple" component):
  - $w_j^{(\tau)}$: The $(1 - \eta\lambda_j)^\tau$ term (assuming $\eta\lambda_j < 1$) goes to zero very fast. $w_j^{(\tau)}$ converges quickly to w_j^*.
  - w_j^{reg}: The $\frac{\lambda_j}{\lambda_j + \alpha}$ term is ≈ 1. The solution is allowed to be w_j^*.
- **When λ_j is small** (low curvature, "complex/noise" component):
  - $w_j^{(\tau)}$: The $(1 - \eta\lambda_j)^\tau$ term stays close to 1. $w_j^{(\tau)}$ stays small and far from w_j^*.
  - w_j^{reg}: The $\frac{\lambda_j}{\lambda_j + \alpha}$ term is ≈ 0. The solution is suppressed toward 0.

This proves that early stopping is mathematically analogous to weight decay. The number of iterations τ acts as an inverse regularization parameter, $1/\alpha$.

4.5 Invariances in Neural Networks

In many pattern recognition problems, the classification of an input should be **invariant** (unchanged) under certain transformations of that input.

Definition 4.6 (Invariance)**.** An invariance is a property where a transformation of the input vector $\mathbf{x}$ does not change the desired output (the target t).

- **Example:** In handwritten digit recognition, the identity of a digit "6" is invariant to:
  - **Translation:** Shifting the digit left, right, up, or down.
  - **Scaling:** Making the digit larger or smaller.
  - **Rotation:** Slightly rotating the digit.
  - **Deformation:** Minor changes in shape or stroke (e.g., elastic deformations).

4.5.1 The Problem

A standard feed-forward network is *not* inherently invariant. A translated input $\tilde{\mathbf{x}}$ is a completely different vector from the original $\mathbf{x}$, and the network will produce a different output.

While a network can *learn* invariance if shown a sufficiently large and diverse training set (e.g., digits at every possible position), this is often impractical, especially if there are multiple invariances (e.g., all positions *and* all sizes).

4.5.2 Four Approaches to Achieve Invariance

There are four main strategies to encourage a model to learn the desired invariances.

1. Augmenting the Training Set

This is the most straightforward approach. The training set is "augmented" by creating multiple transformed copies of the original data.

- **Example:** For each digit image, create 10 new copies, each one slightly shifted, scaled, or rotated.
- **Method:** For sequential (SGD) training, a new random transformation can be applied to an input pattern each time it is presented to the network. For batch training, the original data set is expanded with these transformed replicas.
- **Effect:** This teaches the network by example that $\mathbf{x}$ and its transformed version $\tilde{\mathbf{x}}$ should both map to the same target t.

2. Regularization

A regularization term Ω is added to the error function $E(\mathbf{w})$ to penalize the network if its output changes when the input is transformed.

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\Omega$$

This leads to the technique of **Tangent Propagation** (Section 5.5.4), which specifically penalizes the derivative of the output with respect to the transformation.

3. Invariant Feature Pre-processing

In this approach, invariance is achieved *before* the data reaches the neural network.

- **Method:** A pre-processing step extracts a set of "hand-crafted" features from the raw input $\mathbf{x}$. These features are designed to be invariant to the desired transformations.
- **Example:** Using "moment invariants" from image processing, which produce the same feature values even if an object is scaled or rotated.
- **Effect:** Any classifier that uses these features as its input will automatically inherit the invariance. The challenge is designing good features that are invariant but still discriminative.

4. Building Invariance into the Architecture

This is the most powerful and modern approach. The network's structure is explicitly designed to be invariant.

- **Method:** The network architecture uses specific mechanisms to enforce the invariance.
- **Example: Convolutional Neural Networks (CNNs)** (Section 5.5.6) use:
  - **Local Receptive Fields:** Neurons only look at small patches of the image.
  - **Weight Sharing:** A group of neurons (a "feature map") is forced to use the *exact same* weights. This means they are all detectors for the same feature (e.g., a horizontal edge).
- **Effect:** If the input image is translated, the activations in the feature map will also translate, but their values will be the same. This "equivariance" at the feature level leads to invariance in the final classification.

Chapter 5

Implementing Logical Functions with Neural Networks

5.1 Introduction: The Perceptron Model

We can represent logical functions using a simple artificial neuron model (a Perceptron). This neuron computes a weighted sum of its inputs, adds a bias, and passes the result through an activation function.

Linear Combination The weighted sum, z, is calculated by including a bias unit $x_0 = 1$ with weight $w_0 = b$:

$$z = \sum_{i=0}^{n} w_i x_i = w_0 x_0 + \sum_{i=1}^{n} w_i x_i = b + \mathbf{w}^T \mathbf{x} \tag{5.1}$$

where x_i are inputs, w_i are weights, and b is the bias.

Activation Function For these examples, we will use a **Heaviside step function** as our activation function, $f(z)$. This is the original, non-differentiable activation function of the Perceptron.

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{5.2}$$

The neuron "fires" (outputs 1) if the weighted sum is greater than or equal to zero.

Truth Table Convention Inputs (x_1, x_2) and outputs will use 0 for **False** and 1 for **True**.

5.2 Basic Logic Functions (Linearly Separable)

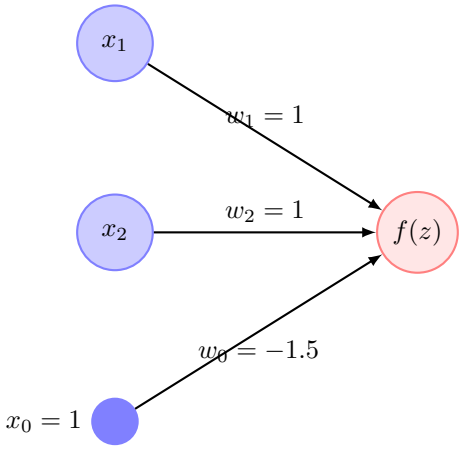
These functions can all be implemented with a single neuron, as their truth tables are *linearly separable*.

5.2.1 AND Logic

Outputs 1 only if $x_1 = 1$ and $x_2 = 1$.

x_1	x_2	$f(z)$
0	0	0
0	1	0
1	0	0
1	1	1

We set $\mathbf{w} = [1, 1]$ and the bias $w_0 = -1.5$.



Verification:

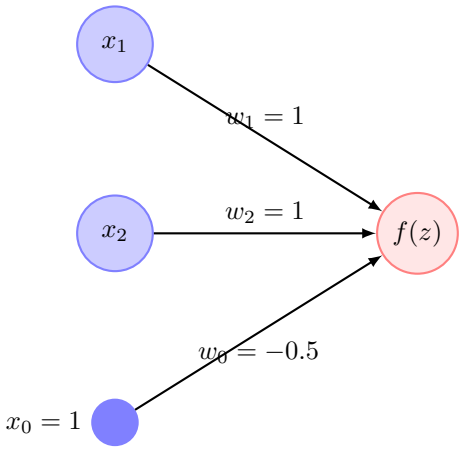
- $x = [0, 0] : z = (1 \cdot 0) + (1 \cdot 0) - 1.5 = -1.5 \implies f(z) = 0$
- $x = [0, 1] : z = (1 \cdot 0) + (1 \cdot 1) - 1.5 = -0.5 \implies f(z) = 0$
- $x = [1, 0] : z = (1 \cdot 1) + (1 \cdot 0) - 1.5 = -0.5 \implies f(z) = 0$
- $x = [1, 1] : z = (1 \cdot 1) + (1 \cdot 1) - 1.5 = +0.5 \implies f(z) = 1$

5.2.2 OR Logic

Outputs 1 if $x_1 = 1$ or $x_2 = 1$ (or both).

x_1	x_2	$f(z)$
0	0	0
0	1	1
1	0	1
1	1	1

We set $\mathbf{w} = [1, 1]$ and the bias $w_0 = -0.5$.



Verification:

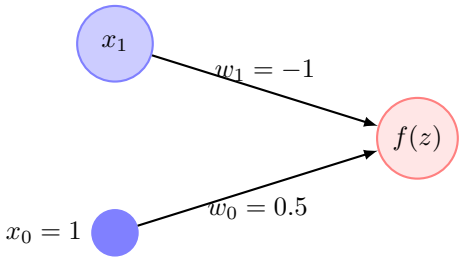
- $x = [0, 0] : z = (1 \cdot 0) + (1 \cdot 0) - 0.5 = -0.5 \implies f(z) = 0$
- $x = [0, 1] : z = (1 \cdot 0) + (1 \cdot 1) - 0.5 = +0.5 \implies f(z) = 1$
- $x = [1, 0] : z = (1 \cdot 1) + (1 \cdot 0) - 0.5 = +0.5 \implies f(z) = 1$
- $x = [1, 1] : z = (1 \cdot 1) + (1 \cdot 1) - 0.5 = +1.5 \implies f(z) = 1$

5.2.3 NOT Logic

A 1-input gate that inverts the input.

x_1	$f(z)$
0	1
1	0

We set $\mathbf{w} = [-1]$ and the bias $w_0 = 0.5$.



Verification:

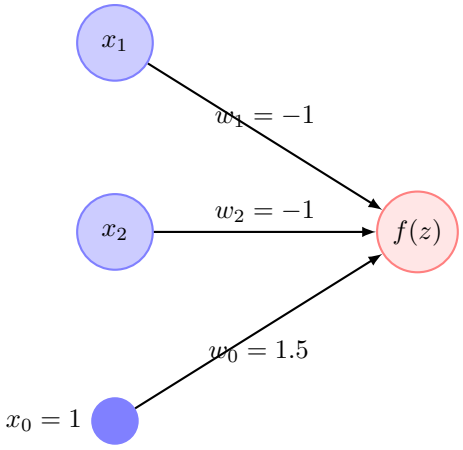
- $x = [0] : z = (-1 \cdot 0) + 0.5 = +0.5 \implies f(z) = 1$
- $x = [1] : z = (-1 \cdot 1) + 0.5 = -0.5 \implies f(z) = 0$

5.2.4 NAND Logic (NOT AND)

The inverse of AND. Outputs 0 only if $x_1 = 1$ and $x_2 = 1$.

x_1	x_2	$f(z)$
0	0	1
0	1	1
1	0	1
1	1	0

We set $\mathbf{w} = [-1, -1]$ and the bias $w_0 = 1.5$.



Verification:

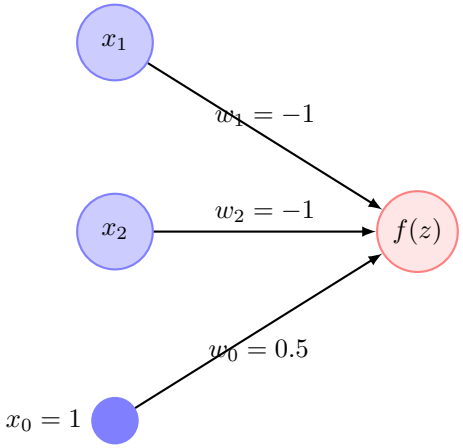
- $x = [0, 0] : z = (-1 \cdot 0) + (-1 \cdot 0) + 1.5 = +1.5 \implies f(z) = 1$
- $x = [0, 1] : z = (-1 \cdot 0) + (-1 \cdot 1) + 1.5 = +0.5 \implies f(z) = 1$
- $x = [1, 0] : z = (-1 \cdot 1) + (-1 \cdot 0) + 1.5 = +0.5 \implies f(z) = 1$
- $x = [1, 1] : z = (-1 \cdot 1) + (-1 \cdot 1) + 1.5 = -0.5 \implies f(z) = 0$

5.2.5 NOR Logic (NOT OR)

The inverse of OR. Outputs 1 only if $x_1 = 0$ and $x_2 = 0$.

x_1	x_2	$f(z)$
0	0	1
0	1	0
1	0	0
1	1	0

We set $\mathbf{w} = [-1, -1]$ and the bias $w_0 = 0.5$.



Verification:

- $x = [0, 0] : z = (-1 \cdot 0) + (-1 \cdot 0) + 0.5 = +0.5 \implies f(z) = 1$
- $x = [0, 1] : z = (-1 \cdot 0) + (-1 \cdot 1) + 0.5 = -0.5 \implies f(z) = 0$
- $x = [1, 0] : z = (-1 \cdot 1) + (-1 \cdot 0) + 0.5 = -0.5 \implies f(z) = 0$
- $x = [1, 1] : z = (-1 \cdot 1) + (-1 \cdot 1) + 0.5 = -1.5 \implies f(z) = 0$

5.3 XOR Logic (Non-Linearly Separable)

Outputs 1 only if x_1 and x_2 are different.

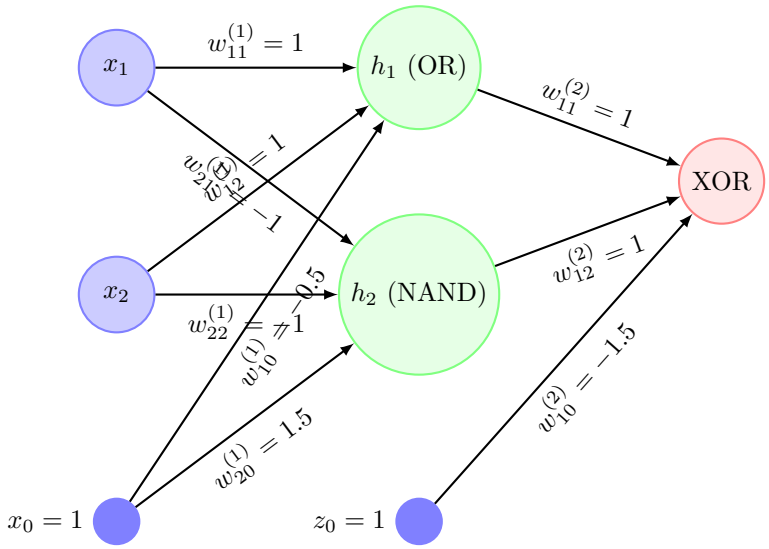
x_1	x_2	$f(z)$
0	0	0
0	1	1
1	0	1
1	1	0

Remark 5.1 (Linear Separability). The XOR function is not linearly separable. It is impossible to find a single set of weights w_1, w_2, w_0 (i.e., a single line) that can separate the $\{0\}$ outputs from the $\{1\}$ outputs. Therefore, it cannot be solved by a single neuron.

We must use a multi-layer network. We can build XOR by combining the gates we already know. A common construction is:

$$\text{XOR} = (\text{A OR B}) \text{ AND } (\text{A NAND B})$$

We will use a network with one hidden layer containing two neurons: one for OR (h_1) and one for NAND (h_2). The output layer will be a single neuron that performs an AND on the hidden layer's outputs. This architecture is identical in principle to the 3-4-2 network in Figure 5.1.



Verification: We trace the values h_1 (OR output) and h_2 (NAND output), and then compute the final output (AND).

- $\mathbf{x} = [\mathbf{0}, \mathbf{0}]$:
  - $h_1 = f((0 \cdot 1) + (0 \cdot 1) - 0.5) = f(-0.5) = 0$
  - $h_2 = f((0 \cdot -1) + (0 \cdot -1) + 1.5) = f(+1.5) = 1$
  - Out $= f((0 \cdot 1) + (1 \cdot 1) - 1.5) = f(-0.5) = \mathbf{0}$
- $\mathbf{x} = [\mathbf{0}, \mathbf{1}]$:
  - $h_1 = f((0 \cdot 1) + (1 \cdot 1) - 0.5) = f(+0.5) = 1$
  - $h_2 = f((0 \cdot -1) + (1 \cdot -1) + 1.5) = f(+0.5) = 1$
  - Out $= f((1 \cdot 1) + (1 \cdot 1) - 1.5) = f(+0.5) = \mathbf{1}$
- $\mathbf{x} = [\mathbf{1}, \mathbf{0}]$:
  - $h_1 = f((1 \cdot 1) + (0 \cdot 1) - 0.5) = f(+0.5) = 1$
  - $h_2 = f((1 \cdot -1) + (0 \cdot -1) + 1.5) = f(+0.5) = 1$
  - Out $= f((1 \cdot 1) + (1 \cdot 1) - 1.5) = f(+0.5) = \mathbf{1}$
- $\mathbf{x} = [\mathbf{1}, \mathbf{1}]$:
  - $h_1 = f((1 \cdot 1) + (1 \cdot 1) - 0.5) = f(+1.5) = 1$
  - $h_2 = f((1 \cdot -1) + (1 \cdot -1) + 1.5) = f(-0.5) = 0$
  - Out $= f((1 \cdot 1) + (0 \cdot 1) - 1.5) = f(-0.5) = \mathbf{0}$

Chapter 6

Building and Generalizing Logic Functions

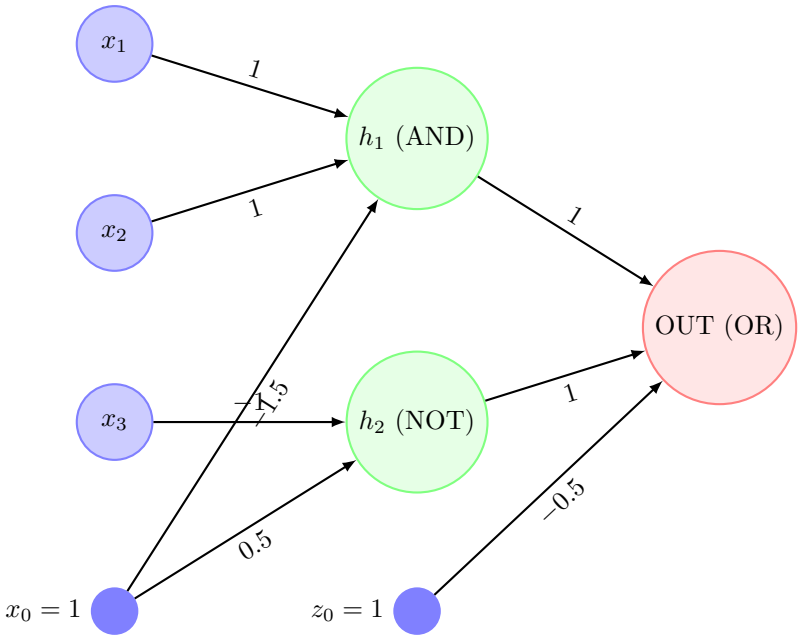
6.1 Composing Functions

The XOR example demonstrates the core principle of the Universal Approximation Theorem. Any complex Boolean expression can be built by composing these basic gates in a network. The expression's structure dictates the network's architecture.

Example 6.1 (Expression: (x_1 AND x_2) OR (NOT x_3))**.** This function takes 3 inputs. The hidden layer will have two neurons:

1. h_1: Implements x_1 AND x_2.
2. h_2: Implements NOT x_3.

The output layer will have one neuron that implements h_1 OR h_2.



6.2 Generalization to n Inputs

We can generalize the weights and biases for the simple gates to work for any number of inputs n.

6.2.1 n-Input AND Gate

Outputs 1 only if all n inputs are 1.

- **Weights:** $w_i = 1$ for $i = 1...n$
- **Bias:** $w_0 = -n + 0.5$

Verification:

- All n inputs are 1: $z = (\sum_{i=1}^{n} 1 \cdot 1) - n + 0.5 = n - n + 0.5 = +0.5 \geq 0 \implies 1$
- At most $n-1$ inputs are 1: $z \leq (1 \cdot (n-1)) - n + 0.5 = n - 1 - n + 0.5 = -0.5 < 0 \implies 0$

6.2.2 n-Input OR Gate

Outputs 1 if at least one input is 1.

- **Weights:** $w_i = 1$ for $i = 1...n$
- **Bias:** $w_0 = -0.5$

Verification:

- All n inputs are 0: $z = (\sum_{i=1}^{n} 1 \cdot 0) - 0.5 = 0 - 0.5 = -0.5 < 0 \implies 0$
- At least one input is 1: $z \geq (1 \cdot 1) - 0.5 = +0.5 \geq 0 \implies 1$

6.2.3 n-Input NAND Gate

Outputs 0 only if all n inputs are 1.

- **Weights:** $w_i = -1$ for $i = 1...n$
- **Bias:** $w_0 = n - 0.5$

Verification:

- All n inputs are 1: $z = (\sum_{i=1}^{n} -1 \cdot 1) + n - 0.5 = -n + n - 0.5 = -0.5 < 0 \implies 0$
- At most $n-1$ inputs are 1: $z \geq (-1 \cdot (n-1)) + n - 0.5 = -n + 1 + n - 0.5 = +0.5 \geq 0 \implies 1$

6.2.4 n-Input NOR Gate

Outputs 1 only if all n inputs are 0.

- **Weights:** $w_i = -1$ for $i = 1...n$
- **Bias:** $w_0 = 0.5$

Verification:

- All n inputs are 0: $z = (\sum_{i=1}^{n} -1 \cdot 0) + 0.5 = 0 + 0.5 = +0.5 \geq 0 \implies 1$
- At least one input is 1: $z \leq (-1 \cdot 1) + 0.5 = -0.5 < 0 \implies 0$

6.3 General Method for Arbitrary n-Input Functions

The methods above show how to build specific gates. We can now show a general-purpose method to construct a two-layer neural network for *any* n-input Boolean function, $f : \{0,1\}^n \to \{0,1\}$.

This method is a constructive proof of the Universal Approximation Theorem for Boolean functions and is based on the **Disjunctive Normal Form (DNF)**.

Proposition 6.2 (Disjunctive Normal Form)**.** *Any Boolean function (excluding the trivial function that always outputs 0) can be expressed as an **OR** of one or more **minterms**.*

- *A **minterm** is an **AND** of all n input variables, where each variable x_i is either in its original form (x_i) or its negated form ($\neg x_i$).*
- *Each minterm corresponds to exactly one row in the truth table that outputs a 1.*

6.3.1 The DNF Network Architecture

This DNF representation maps directly to a two-layer neural network:

1. **Hidden Layer:** Each neuron in the hidden layer implements one **minterm (an AND gate)**. We will have one hidden neuron for each row in the truth table that outputs a 1.
2. **Output Layer:** A single output neuron implements an **OR gate**, taking the outputs of all hidden layer neurons as its input.

This network will output 1 if *any* of its hidden layer minterm-neurons fire, which is exactly the definition of the DNF.

6.3.2 Rule for Minterm Neurons

A hidden neuron h_j that implements a specific minterm (e.g., $x_1 \wedge \neg x_2 \wedge x_3$) can be constructed with the following weights and bias:

- **Weights (w_{ji}):** For each input x_i:
  - If the term is x_i (un-negated), set weight $w_{ji} = +1$.
  - If the term is $\neg x_i$ (negated), set weight $w_{ji} = -1$.
- **Bias (w_{j0}):** Set the bias to $b = -(\text{number of un-negated inputs}) + 0.5$.

Verification of Minterm Rule. Let the number of un-negated inputs be k. The bias is $b = -k + 0.5$.

- **Correct Input Row:** For the one correct input row, all un-negated x_i are 1 and all negated x_i are 0.

$$z = \sum(w_i x_i) + b = \sum_{\text{un-negated}} (1 \cdot 1) + \sum_{\text{negated}} (-1 \cdot 0) + (-k + 0.5)$$

$$z = (k) + (0) - k + 0.5 = +0.5 \geq 0 \implies \text{Output is 1}$$

- **Any Other Row:** Any other input row must differ by at least one bit.
  - *Case 1: An un-negated x_i flips to 0.* The sum loses 1. $z \leq (k-1) - k + 0.5 = -0.5 < 0 \implies$ Output is 0.
  - *Case 2: A negated x_i flips to 1.* The sum gains $(-1 \cdot 1) = -1$. $z \leq k - 1 - k + 0.5 = -0.5 < 0 \implies$ Output is 0.

  In all other cases, the output is 0. The rule is correct.

□

6.3.3 Example: 3-Input Parity Function (3-XOR)

Let's build a network for a function that outputs 1 if an *odd* number of inputs are 1.

1. Truth Table and Minterms

x_1	x_2	x_3	$f(x)$	Minterm
0	0	0	0	
0	0	1	1	$h_1 = (\neg x_1 \wedge \neg x_2 \wedge x_3)$
0	1	0	1	$h_2 = (\neg x_1 \wedge x_2 \wedge \neg x_3)$
0	1	1	0	
1	0	0	1	$h_3 = (x_1 \wedge \neg x_2 \wedge \neg x_3)$
1	0	1	0	
1	1	0	0	
1	1	1	1	$h_4 = (x_1 \wedge x_2 \wedge x_3)$

2. DNF (Function) The function is the OR of the four minterms that output 1.

$$f(x) = h_1 \vee h_2 \vee h_3 \vee h_4$$

3. Network Architecture (3-4-1)

- **Input Layer:** 3 neurons (x_1, x_2, x_3) + bias (x_0).
- **Hidden Layer:** 4 neurons (h_1, h_2, h_3, h_4) + bias (z_0).
- **Output Layer:** 1 neuron (y).

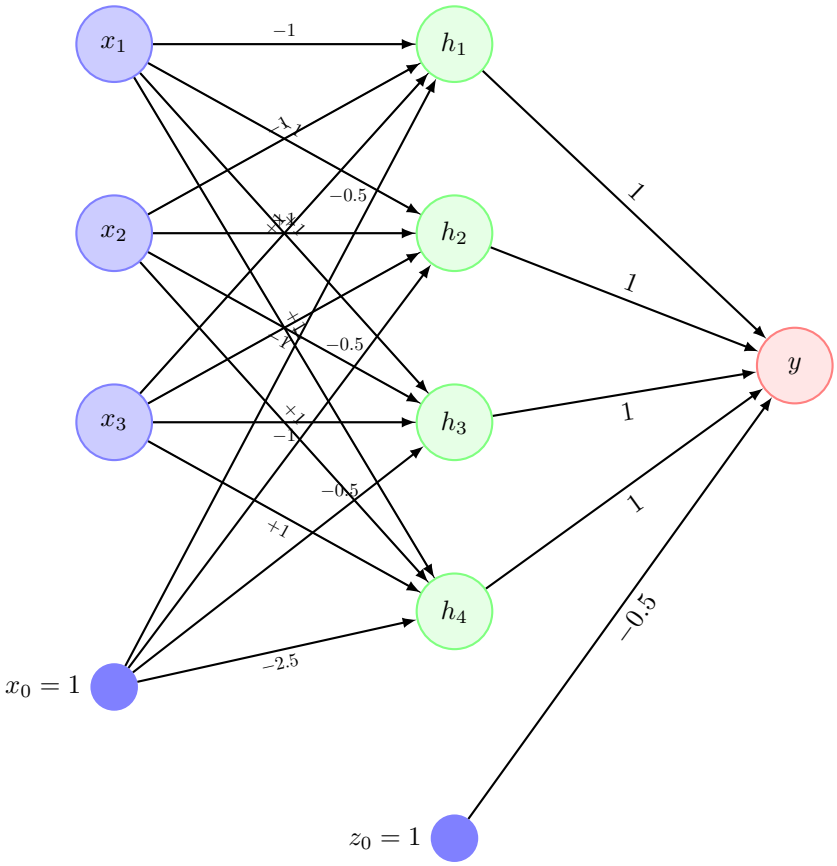
4. Weights and Biases Hidden Layer (Minterm Neurons):

- $\mathbf{h_1} = (\neg \mathbf{x_1} \wedge \neg \mathbf{x_2} \wedge \mathbf{x_3})$: (1 un-negated term) $\implies \mathbf{w}_1^{(1)} = [-1, -1, +1]$, $w_{10}^{(1)} = -1 + 0.5 = -0.5$
- $\mathbf{h_2} = (\neg \mathbf{x_1} \wedge \mathbf{x_2} \wedge \neg \mathbf{x_3})$: (1 un-negated term) $\implies \mathbf{w}_2^{(1)} = [-1, +1, -1]$, $w_{20}^{(1)} = -1 + 0.5 = -0.5$
- $\mathbf{h_3} = (\mathbf{x_1} \wedge \neg \mathbf{x_2} \wedge \neg \mathbf{x_3})$: (1 un-negated term) $\implies \mathbf{w}_3^{(1)} = [+1, -1, -1]$, $w_{30}^{(1)} = -1 + 0.5 = -0.5$
- $\mathbf{h_4} = (\mathbf{x_1} \wedge \mathbf{x_2} \wedge \mathbf{x_3})$: (3 un-negated terms) $\implies \mathbf{w}_4^{(1)} = [+1, +1, +1]$, $w_{40}^{(1)} = -3 + 0.5 = -2.5$

Output Layer (4-input OR gate):

- $\mathbf{y} = (\mathbf{h_1} \vee \mathbf{h_2} \vee \mathbf{h_3} \vee \mathbf{h_4})$: (General $n = 4$ OR gate) $\implies \mathbf{w}_1^{(2)} = [1, 1, 1, 1]$, $w_{10}^{(2)} = -0.5$

5. Final Network Diagram



Part II

Convolutional Neural Networks

Chapter 7

Convolutional Filters

7.1 Introduction

7.1.1 Motivation and Inductive Bias

A standard fully connected network is a poor choice for high-dimensional data like images for two main reasons:

1. **Vast Number of Parameters:** A modest $10^3 \times 10^3$ color image (3 million inputs) connecting to a 1,000-unit hidden layer would have 3×10^9 weights in the first layer alone, making it computationally infeasible.
2. **Requires Huge Datasets:** Such a network would have to learn all invariances (like translation or scale) by example, which would require an enormous training set.

By designing an architecture that incorporates our **inductive bias** (prior knowledge) about the structure of images, we can dramatically reduce the data requirements and improve generalization.

7.1.2 Four Key Concepts for Image Structure

We exploit the 2D structure of images using four interrelated concepts:

- **Hierarchy:** Natural images have a hierarchical structure (e.g., a face contains eyes, which contain an iris, which is defined by edges). More complex features are built by composing simpler features from previous levels.
- **Locality:** A low-level feature, like an edge, can be detected by a neuron using only a small, local region of the image (a small subset of pixels).
- **Equivariance & Invariance:** These are properties the architecture is designed to have (discussed later).

The goal of a deep learning model is to allow the network to *learn* the details of this feature hierarchy from the data, rather than requiring us to hand-code the features. The entire system is trained *end-to-end*.

7.2 Feature Detectors

A core concept of CNNs is to move away from fully connected layers. Instead, units in the first hidden layer are designed as **feature detectors** that operate on small, local regions of the input image.

7.2.1 Locality and Receptive Fields

A unit in the first layer does not connect to the entire input image. It only takes inputs from a small rectangular patch of the image.

Definition 7.1 (Receptive Field)**.** The **receptive field** of a unit is the small patch of pixels in the input image that it is connected to. This design choice enforces the inductive bias of **locality**.

7.2.2 Mathematical Definition

The unit's output is calculated as a weighted sum of the pixel values in its receptive field, plus a bias, which is then passed through a non-linear activation function.

Filter / Kernel The set of weights $\mathbf{w}$ for a single unit can be visualized as a 2D grid, matching the shape of the receptive field (e.g., 3×3). This grid of weights is called a **filter** or **kernel**.

Vector Notation If we "flatten" the receptive field patch (e.g., a 3×3 patch becomes a 9×1 vector $\mathbf{x}$) and flatten the kernel (a 3×3 filter becomes a 9×1 vector $\mathbf{w}$), the output z of the unit is:

$$z = f(\mathbf{w}^T\mathbf{x} + w_0) \tag{7.1}$$

A common activation function $f(\cdot)$ is the Rectified Linear Unit (ReLU):

$$z = \text{ReLU}(\mathbf{w}^T\mathbf{x} + w_0) \tag{7.2}$$

2D Convolutional Notation More formally, we can write this operation as a 2D sum. Let the input image be I, the kernel be K, and the output of the single unit be z_{jk} (representing its position in the next layer). The unit's activation a_{jk} is:

$$a_{jk} = \left(\sum_{l=1}^{M} \sum_{m=1}^{M} I(j+l, k+m) K(l, m) \right) + w_0 \tag{7.3}$$

where M is the size of the filter (e.g., $M = 3$ for a 3×3 filter). The final output of the unit is then:

$$z_{jk} = f(a_{jk}) = \text{ReLU}(a_{jk}) \tag{7.4}$$

7.2.3 How it Works

The unit acts as a detector for the feature represented by its filter. The term $\mathbf{w}^T\mathbf{x}$ (the 2D sum) will have the largest positive value when the input patch $\mathbf{x}$ "looks like" the filter $\mathbf{w}$.

The ReLU activation $z = \text{ReLU}(\mathbf{w}^T\mathbf{x} + w_0)$ will be non-zero only if this "match" is sufficiently strong to overcome the negative bias $-w_0$. Therefore, the unit "fires" (outputs a non-zero value) when it detects a feature it is looking for.

Part III

Recurrent Neural Networks

Chapter 8

Foundations of Sequence Modelling

8.1 The Problem: Sequential Data

Our goal is to model sequential data. A standard data point $\mathbf{x}$ is a vector in $\mathbb{R}^D$. A **sequence** X is an ordered set of such vectors indexed by time t.

Definition 8.1 (Sequential Data)**.**

- **Input Sequence:** $X = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)})$
- **Input Vector at time t:** $\mathbf{x}^{(t)} \in \mathbb{R}^{D_x}$, where D_x is the input feature dimension.
- **Sequence Length:** T, which can be variable.

A standard Multilayer Perceptron (MLP) computes a function $y = f(\mathbf{x}, \mathbf{W})$ and is designed for data where order and structure are irrelevant. It fails for sequential data for three primary mathematical reasons.

8.1.1 Failure 1: Fixed-Size Input and Output

An MLP architecture is rigid. Its input layer has a fixed number of neurons, D_{in}, and its output layer has a fixed number, D_{out}.

- **The Problem:** Sequences have variable lengths. For example:
  - Sentence 1 ($T = 2$): "I read."
  - Sentence 2 ($T = 4$): "I read the book."
- **The Limitation:** If we "flatten" these sequences into single vectors (assuming $D_x = 100$):
  - Input 1 has dimension $T \times D_x = 2 \times 100 = 200$.
  - Input 2 has dimension $T \times D_x = 4 \times 100 = 400$.

  An MLP's first weight matrix $\mathbf{W}^{(1)}$ has a fixed shape (e.g., $D_{hidden} \times 200$). It is mathematically impossible for this matrix to multiply with the 400-dimensional input from Sentence 2.

  Common "hacks" like padding (adding zeros) or truncating (cutting off data) are computationally wasteful and lead to information loss.

8.1.2 Failure 2: No Parameter Sharing Across Time

This is the most significant theoretical failure. Even if we pad all sequences to a "max length" T_{max}, the MLP treats the problem incorrectly.

- **The Problem:** An MLP has independent weights for each input position.
- **Mathematical Form:** Let the flattened input be $\mathbf{x}_{flat} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}]$. The first hidden layer activation $\mathbf{a}$ is $\mathbf{a} = \mathbf{W}\mathbf{x}_{flat} + \mathbf{b}$.

  The weight matrix $\mathbf{W}$ can be seen as blocks: $\mathbf{W} = [\mathbf{W}_{\text{time\_1}} | \mathbf{W}_{\text{time\_2}} | \ldots | \mathbf{W}_{\text{time\_T}}]$

  The weights $\mathbf{W}_{\text{time\_1}}$ that process the first word $\mathbf{x}^{(1)}$ are completely independent from the weights $\mathbf{W}_{\text{time\_2}}$ that process the second word $\mathbf{x}^{(2)}$.
- **The Consequence:** The rules of language or physics are the same at $t = 1$ and $t = 2$. An MLP, however, must re-learn the *exact same rules* at every single time step using a completely different set of parameters. This is statistically inefficient and fails to generalize.

  We need an architecture that uses the *same* set of weights at every time step. This is **parameter sharing**.

8.1.3 Failure 3: No "Memory" or Internal State

An MLP is a stateless function. Its output for a given input is computed without any "memory" of past inputs.

- **The Problem:** The meaning at time t often depends on information from much earlier time steps (a **long-range dependency**).
- **Example:** "The **cat**, which had been chasing birds all day, finally **sat** down."

  To predict that "sat" is the verb for "cat," the model must have a memory of the word "cat" from many steps prior.
- **The Limitation:** An MLP has no mechanism to store and pass this information. The calculation for "sat" is performed by a different set of weights than the calculation for "cat," and no information is passed between them.

  We need an architecture that maintains an internal **state** or "memory" $\mathbf{h}^{(t)}$ that is updated at each time step and carries information forward.

8.2 Building the RNN Equations

To understand the general equations, we will build them up layer by layer, starting from the simplest possible recurrent network.

8.2.1 Case 1: The Simplest RNN (1-1-1 Network)

Let's analyze a network with 1 input, 1 hidden neuron, and 1 output neuron.

- **Input ($D_x = 1$):** The input $\mathbf{x}^{(t)}$ is a single scalar $x^{(t)}$.
- **Hidden Layer ($D_h = 1$):** The hidden state $\mathbf{h}^{(t)}$ is a single scalar $h^{(t)}$.
- **Output Layer ($D_y = 1$):** The output $\hat{y}^{(t)}$ is a single scalar.

Weight "Matrices" as Scalars

Because all our layers have dimension 1, all the "matrices" are just 1×1 scalars.

- $\mathbf{W}_{xh} \in \mathbb{R}^{1\times 1} \implies$ a single scalar weight w_{xh}
- $\mathbf{W}_{hh} \in \mathbb{R}^{1\times 1} \implies$ a single scalar weight w_{hh}
- $\mathbf{W}_{hy} \in \mathbb{R}^{1\times 1} \implies$ a single scalar weight w_{hy}
- $\mathbf{b}_h \in \mathbb{R}^1 \implies$ a single scalar bias b_h
- $\mathbf{b}_y \in \mathbb{R}^1 \implies$ a single scalar bias b_y

The Forward Pass Equations (Scalar Form)

The general matrix equations now become simple scalar arithmetic.

1. Hidden Activation $a^{(t)}$: The activation $a^{(t)}$ is the sum of the signal from the input and the signal from the previous hidden state, plus a bias.

$$\begin{aligned} a^{(t)} &= (\text{Signal from memory}) + (\text{Signal from input}) + (\text{Bias}) \\ a^{(t)} &= (w_{hh} \cdot h^{(t-1)}) + (w_{xh} \cdot x^{(t)}) + b_h \end{aligned}$$

This is the simplest form of the recurrent equation.

2. Hidden State $h^{(t)}$: The activation is passed through the non-linearity $f(\cdot)$ (e.g., tanh).

$$h^{(t)} = f(a^{(t)}) = \tanh\left(w_{hh}h^{(t-1)} + w_{xh}x^{(t)} + b_h\right)$$

This $h^{(t)}$ is the "memory" that gets passed to time step $t+1$.

3. Output Activation $o^{(t)}$: The new hidden state is used to make a prediction.

$$o^{(t)} = (w_{hy} \cdot h^{(t)}) + b_y$$

4. Final Prediction $\hat{y}^{(t)}$: The output activation is passed through the output function $g(\cdot)$ (e.g., identity).

$$\hat{y}^{(t)} = g(o^{(t)}) = o^{(t)}$$

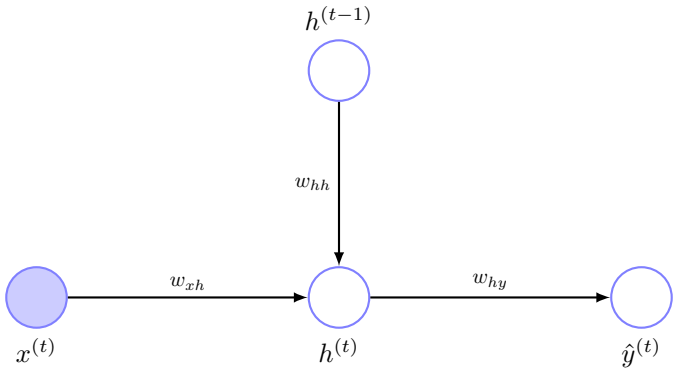


Figure 8.1: A 1-1-1 network, showing the scalar weights for input (w_{xh}), hidden state (w_{hh}), and output (w_{hy}). Biases are omitted for clarity.

8.2.2 Case 2: 1 Input, D_h Hidden Neurons (1-D_h-1 Network)

Now we step up the complexity. The input is still a single scalar $x^{(t)}$, but the hidden layer now has D_h neurons (e.g., $D_h = 3$). The hidden state $\mathbf{h}^{(t)}$ is now a vector.

- **Input ($D_x = 1$):** A scalar $x^{(t)}$.
- **Hidden Layer ($D_h > 1$):** A vector $\mathbf{h}^{(t)} = [h_1^{(t)}, \dots, h_{D_h}^{(t)}]^T$.
- **Output Layer ($D_y = 1$):** A scalar $\hat{y}^{(t)}$.

Weight Matrices (from Scalars to Vectors/Matrices)

This is the most important step. Let's see how the weights change.

- $\mathbf{W}_{xh} \in \mathbb{R}^{D_h \times 1}$: The connection from the 1 input to all D_h hidden neurons is now a **column vector**.

$$\mathbf{W}_{xh} = \begin{pmatrix} w_{1,1}^{(xh)} \\ w_{2,1}^{(xh)} \\ \vdots \\ w_{D_h,1}^{(xh)} \end{pmatrix}$$

- $\mathbf{W}_{hh} \in \mathbb{R}^{D_h \times D_h}$: The connections *between* the D_h hidden neurons at $t-1$ and the D_h hidden neurons at t is now a full **square matrix**.

$$\mathbf{W}_{hh} = \begin{pmatrix} w_{1,1}^{(hh)} & w_{1,2}^{(hh)} & \dots & w_{1,D_h}^{(hh)} \\ w_{2,1}^{(hh)} & w_{2,2}^{(hh)} & \dots & w_{2,D_h}^{(hh)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_h,1}^{(hh)} & w_{D_h,2}^{(hh)} & \dots & w_{D_h,D_h}^{(hh)} \end{pmatrix}$$

- $\mathbf{W}_{hy} \in \mathbb{R}^{1 \times D_h}$: The connections from all D_h hidden neurons to the 1 output neuron is now a **row vector**.

$$\mathbf{W}_{hy} = \begin{pmatrix} w_{1,1}^{(hy)}, & w_{1,2}^{(hy)}, & \dots, & w_{1,D_h}^{(hy)} \end{pmatrix}$$

- $\mathbf{b}_h \in \mathbb{R}^{D_h}$: A vector of D_h biases, one for each hidden neuron.
- $\mathbf{b}_y \in \mathbb{R}^1$: A scalar bias for the output.

The Forward Pass Equations (Vector Form)

1. Hidden Activation $\mathbf{a}^{(t)}$: The activation for the *entire* hidden layer $\mathbf{a}^{(t)}$ (a $D_h \times 1$ vector) is computed.

$$\begin{pmatrix} a_1^{(t)} \\ \vdots \\ a_{D_h}^{(t)} \end{pmatrix} = \underbrace{\begin{pmatrix} w_{1,1}^{(hh)} & \dots & w_{1,D_h}^{(hh)} \\ \vdots & \ddots & \vdots \\ w_{D_h,1}^{(hh)} & \dots & w_{D_h,D_h}^{(hh)} \end{pmatrix}}_{\mathbf{W}_{hh}} \underbrace{\begin{pmatrix} h_1^{(t-1)} \\ \vdots \\ h_{D_h}^{(t-1)} \end{pmatrix}}_{\mathbf{h}^{(t-1)}} + \underbrace{\begin{pmatrix} w_{1,1}^{(xh)} \\ \vdots \\ w_{D_h,1}^{(xh)} \end{pmatrix}}_{\mathbf{W}_{xh}} \underbrace{(x^{(t)})}_{x^{(t)}} + \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_{D_h} \end{pmatrix}}_{\mathbf{b}_h}$$

In compact notation, this is our first full vector equation:

$$\mathbf{a}^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}x^{(t)} + \mathbf{b}_h \tag{8.1}$$

2. Hidden State $\mathbf{h}^{(t)}$: The non-linearity $f(\cdot)$ is applied **element-wise** to the activation vector $\mathbf{a}^{(t)}$.

$$\mathbf{h}^{(t)} = f(\mathbf{a}^{(t)}) = \tanh(\mathbf{a}^{(t)}) = \begin{pmatrix} \tanh(a_1^{(t)}) \\ \vdots \\ \tanh(a_{D_h}^{(t)}) \end{pmatrix}$$

3. Output Activation $o^{(t)}$: The new hidden state vector $\mathbf{h}^{(t)}$ is multiplied by the row vector $\mathbf{W}_{hy}$ to produce a single scalar output activation.

$$o^{(t)} = \begin{pmatrix} w_{1,1}^{(hy)}, & \dots, & w_{1,D_h}^{(hy)} \end{pmatrix} \begin{pmatrix} h_1^{(t)} \\ \vdots \\ h_{D_h}^{(t)} \end{pmatrix} + b_y$$

$$o^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + b_y$$

4. Final Prediction $\hat{y}^{(t)}$:

$$\hat{y}^{(t)} = g(o^{(t)})$$

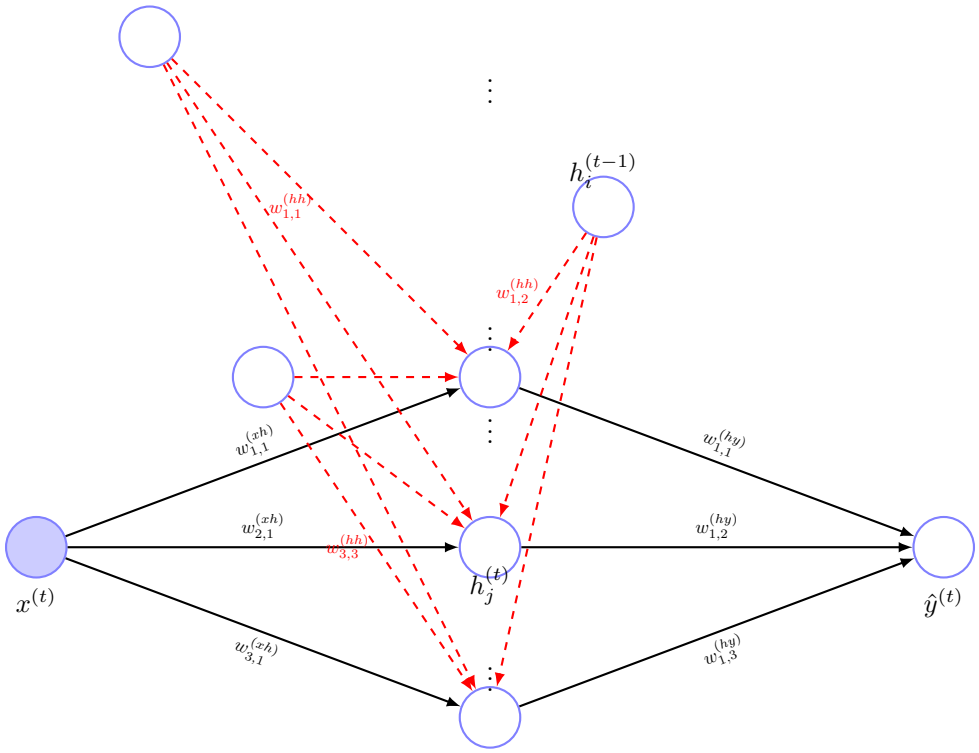


Figure 8.2: A 1-D_h-1 network. $\mathbf{W}_{xh}$ is a vector (left), $\mathbf{W}_{hy}$ is a vector (right), and $\mathbf{W}_{hh}$ is a full matrix (red, dashed) connecting all D_h previous states to all D_h current states.

8.2.3 Case 3: D_x Inputs, D_h Hidden Neurons (The General Case)

Finally, we arrive at the general case used in practice. We have multiple inputs (e.g., a D_x-dimensional word vector) and multiple hidden neurons (a D_h-dimensional memory). We will keep the output $D_y = 1$ for simplicity.

- **Input ($D_x > 1$):** A vector $\mathbf{x}^{(t)} = [x_1^{(t)}, \dots, x_{D_x}^{(t)}]^T$.
- **Hidden Layer ($D_h > 1$):** A vector $\mathbf{h}^{(t)} = [h_1^{(t)}, \dots, h_{D_h}^{(t)}]^T$.
- **Output Layer ($D_y = 1$):** A scalar $\hat{y}^{(t)}$.

Weight Matrices (Full Matrix Form)

This is the final form of the weight matrices.

- $\mathbf{W}_{xh} \in \mathbb{R}^{D_h \times D_x}$: The connection from D_x inputs to D_h hidden neurons is now a full **rectangular matrix**.

$$\mathbf{W}_{xh} = \begin{pmatrix} w_{1,1}^{(xh)} & w_{1,2}^{(xh)} & \dots & w_{1,D_x}^{(xh)} \\ w_{2,1}^{(xh)} & w_{2,2}^{(xh)} & \dots & w_{2,D_x}^{(xh)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_h,1}^{(xh)} & w_{D_h,2}^{(xh)} & \dots & w_{D_h,D_x}^{(xh)} \end{pmatrix}$$

  Each *row* j of this matrix contains the weights connecting all D_x inputs to the single hidden neuron j.

- $\mathbf{W}_{hh} \in \mathbb{R}^{D_h \times D_h}$: As in Case 2, this is a full **square matrix** connecting all D_h previous hidden states to all D_h current hidden states.

$$\mathbf{W}_{hh} = \begin{pmatrix} w_{1,1}^{(hh)} & \dots & w_{1,D_h}^{(hh)} \\ \vdots & \ddots & \vdots \\ w_{D_h,1}^{(hh)} & \dots & w_{D_h,D_h}^{(hh)} \end{pmatrix}$$

- $\mathbf{W}_{hy} \in \mathbb{R}^{1 \times D_h}$: As in Case 2, this is a **row vector** connecting all D_h hidden states to the single output neuron.
- $\mathbf{b}_h \in \mathbb{R}^{D_h}$: A vector of D_h biases.
- $\mathbf{b}_y \in \mathbb{R}^1$: A scalar bias.

The Forward Pass Equations (Final Vector Form)

1. Hidden Activation $\mathbf{a}^{(t)}$: The activation $\mathbf{a}^{(t)}$ (a $D_h \times 1$ vector) is the sum of the two matrix-vector products plus the bias vector.

$$\begin{pmatrix} a_1^{(t)} \\ \vdots \\ a_{D_h}^{(t)} \end{pmatrix} = \underbrace{\begin{pmatrix} w_{1,1}^{(hh)} & \dots & w_{1,D_h}^{(hh)} \\ \vdots & \ddots & \vdots \\ w_{D_h,1}^{(hh)} & \dots & w_{D_h,D_h}^{(hh)} \end{pmatrix}}_{\mathbf{W}_{hh}\ (D_h \times D_h)} \underbrace{\begin{pmatrix} h_1^{(t-1)} \\ \vdots \\ h_{D_h}^{(t-1)} \end{pmatrix}}_{\mathbf{h}^{(t-1)}\ (D_h \times 1)} + \underbrace{\begin{pmatrix} w_{1,1}^{(xh)} & \dots & w_{1,D_x}^{(xh)} \\ \vdots & \ddots & \vdots \\ w_{D_h,1}^{(xh)} & \dots & w_{D_h,D_x}^{(xh)} \end{pmatrix}}_{\mathbf{W}_{xh}\ (D_h \times D_x)} \underbrace{\begin{pmatrix} x_1^{(t)} \\ \vdots \\ x_{D_x}^{(t)} \end{pmatrix}}_{\mathbf{x}^{(t)}\ (D_x \times 1)} + \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_{D_h} \end{pmatrix}}_{\mathbf{b}_h\ (D_h \times 1)}$$

This gives us the final, general equation for the hidden layer activation:

$$\mathbf{a}^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h \tag{8.2}$$

2. Hidden State $\mathbf{h}^{(t)}$: The non-linearity $f(\cdot)$ is applied **element-wise** to the activation vector $\mathbf{a}^{(t)}$.

$$\mathbf{h}^{(t)} = f(\mathbf{a}^{(t)}) = \tanh(\mathbf{a}^{(t)}) \tag{8.3}$$

3. Output Activation $o^{(t)}$ and Prediction $\hat{y}^{(t)}$: This part is the same as in Case 2.

$$o^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + b_y \tag{8.4}$$

$$\hat{y}^{(t)} = g(o^{(t)}) \tag{8.5}$$

8.2.4 The General Case: RNN Equations

From our step-by-step construction, we can now define the general equations for a recurrent neural network at a single time step t.

Definitions

- Input vector at time t: $\mathbf{x}^{(t)} \in \mathbb{R}^{D_x}$
- Hidden state at time $t-1$: $\mathbf{h}^{(t-1)} \in \mathbb{R}^{D_h}$
- Hidden state at time t: $\mathbf{h}^{(t)} \in \mathbb{R}^{D_h}$
- Output prediction at time t: $\hat{\mathbf{y}}^{(t)} \in \mathbb{R}^{D_y}$

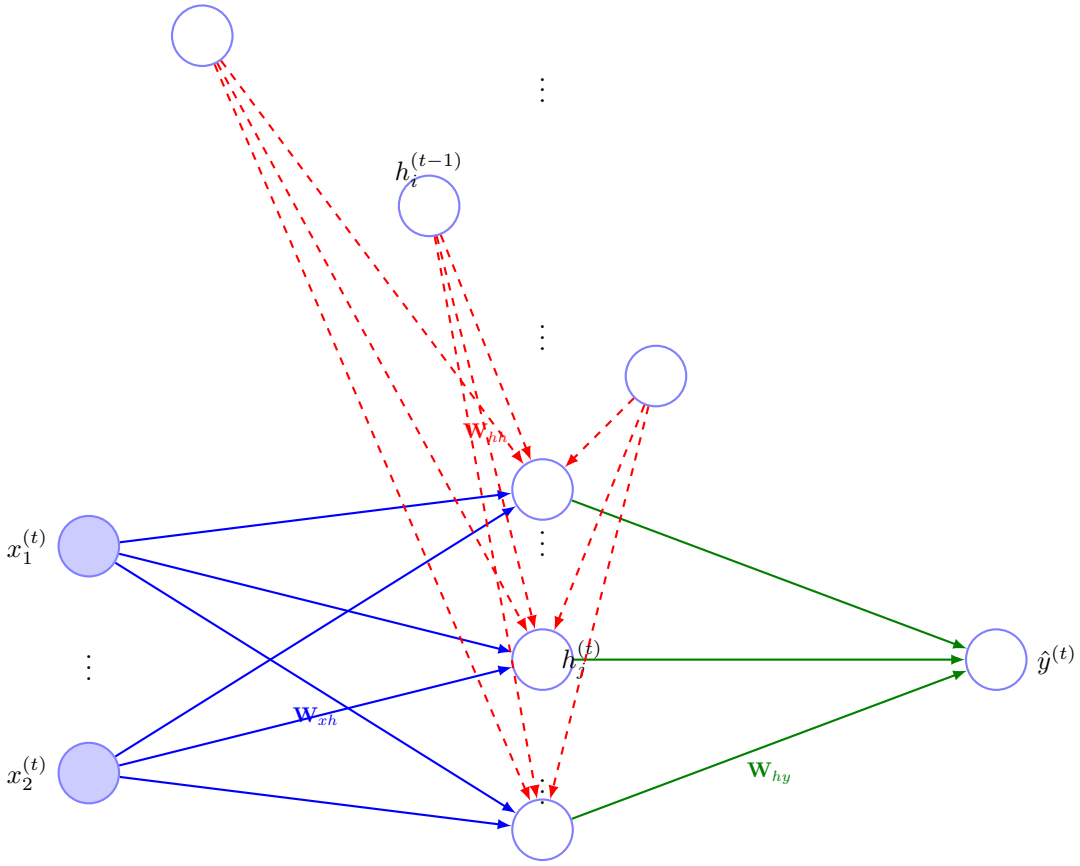


Figure 8.3: The general RNN case ($D_x > 1, D_h > 1$). $\mathbf{W}_{xh}$ (blue) and $\mathbf{W}_{hh}$ (red) are now both full matrices, and their outputs are added at each hidden neuron.

Shared Parameters (The "Rule")

These parameters are *the same* for all time steps $t = 1, \ldots, T$.

- Input-to-Hidden weights: $\mathbf{W}_{xh} \in \mathbb{R}^{D_h \times D_x}$
- Hidden-to-Hidden weights: $\mathbf{W}_{hh} \in \mathbb{R}^{D_h \times D_h}$
- Hidden-to-Output weights: $\mathbf{W}_{hy} \in \mathbb{R}^{D_y \times D_h}$
- Hidden bias vector: $\mathbf{b}_h \in \mathbb{R}^{D_h}$
- Output bias vector: $\mathbf{b}_y \in \mathbb{R}^{D_y}$

Forward Pass Equations

The model is defined by two primary equations: the hidden state update and the output calculation.

1. **The Hidden State Update** First, the activation $\mathbf{a}^{(t)}$ is computed by summing the weighted contributions from the current input and the previous hidden state, plus a bias.

$$\mathbf{a}^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h \tag{8.6}$$

   This activation vector is then passed through an element-wise non-linear function $f(\cdot)$ (typically tanh) to produce the new hidden state.

$$\mathbf{h}^{(t)} = f(\mathbf{a}^{(t)}) \tag{8.7}$$

2. **The Output Calculation** The new hidden state $\mathbf{h}^{(t)}$ is used to make a prediction. An output activation $\mathbf{o}^{(t)}$ is computed.

$$\mathbf{o}^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y \tag{8.8}$$

   This is passed through an output activation function $g(\cdot)$ (e.g., softmax for classification, identity for regression) to get the final prediction.

$$\hat{\mathbf{y}}^{(t)} = g(\mathbf{o}^{(t)}) \tag{8.9}$$

(Note: The initial state $\mathbf{h}^{(0)}$ is typically initialized as a zero vector).

Remark 8.2. These equations define the computation for a single time step. To process a sequence, these equations are applied in a loop for $t = 1 \dots T$. This sequential computation is the basis for the "unfolding in time" visualization, which is the next logical step (Module 1.3) and the key to deriving the training algorithm.

8.3 The "Unfolding in Time" Computational Graph

The recurrent equation $\mathbf{h}^{(t)} = f(\mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \dots)$ creates a cycle, which makes it impossible to apply standard backpropagation (which requires a Directed Acyclic Graph, or DAG).

To derive the training algorithm, we visualize the network's computation by **"unfolding"** or "unrolling" the loop in time. This creates a deep, feed-forward network that is mathematically equivalent to the RNN for a specific, finite sequence of length T.

8.3.1 Mathematical Definition of the Unfolded Graph

For a sequence of length T, the unfolded graph is a T-layer network where each "layer" t corresponds to a time step. The computation at each time step t (from $t = 1$ to T) is defined as follows:

Time Step $t = 1$:

$$\begin{aligned}\mathbf{a}^{(1)} &= \mathbf{W}_{hh}\mathbf{h}^{(0)} + \mathbf{W}_{xh}\mathbf{x}^{(1)} + \mathbf{b}_h \\ \mathbf{h}^{(1)} &= f(\mathbf{a}^{(1)}) \\ \mathbf{o}^{(1)} &= \mathbf{W}_{hy}\mathbf{h}^{(1)} + \mathbf{b}_y \\ \hat{\mathbf{y}}^{(1)} &= g(\mathbf{o}^{(1)})\end{aligned}$$

Time Step $t = 2$:

$$\begin{aligned}\mathbf{a}^{(2)} &= \mathbf{W}_{hh}\mathbf{h}^{(1)} + \mathbf{W}_{xh}\mathbf{x}^{(2)} + \mathbf{b}_h \\ \mathbf{h}^{(2)} &= f(\mathbf{a}^{(2)}) \\ \mathbf{o}^{(2)} &= \mathbf{W}_{hy}\mathbf{h}^{(2)} + \mathbf{b}_y \\ \hat{\mathbf{y}}^{(2)} &= g(\mathbf{o}^{(2)})\end{aligned}$$

$\vdots$

Time Step $t = T$:

$$\begin{aligned}\mathbf{a}^{(T)} &= \mathbf{W}_{hh}\mathbf{h}^{(T-1)} + \mathbf{W}_{xh}\mathbf{x}^{(T)} + \mathbf{b}_h \\ \mathbf{h}^{(T)} &= f(\mathbf{a}^{(T)}) \\ \mathbf{o}^{(T)} &= \mathbf{W}_{hy}\mathbf{h}^{(T)} + \mathbf{b}_y \\ \hat{\mathbf{y}}^{(T)} &= g(\mathbf{o}^{(T)})\end{aligned}$$

8.3.2 Diagram of the Unfolded Graph (for $T = 3$)

Remark 8.3 (Significance of Unfolding). This unfolded computational graph is the key to training RNNs.

1. **It is a DAG:** The graph is now a deep feed-forward network with no cycles. We can now apply the standard backpropagation algorithm to it.
2. **It visualizes Parameter Sharing:** The diagram makes it clear that $\mathbf{W}_{hh}$ (red), $\mathbf{W}_{xh}$ (blue), and $\mathbf{W}_{hy}$ (green) are the *same* parameters used in each layer.
3. **It defines the Gradient:** When we run backpropagation, the total gradient for a shared weight (like $\mathbf{W}_{hh}$) is the **sum** of the individual gradients computed at each time step.

$$\frac{\partial E_{\text{total}}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}} \tag{8.10}$$

This specific application of backpropagation to an unfolded recurrent graph is known as **Backpropagation Through Time (BPTT)**.

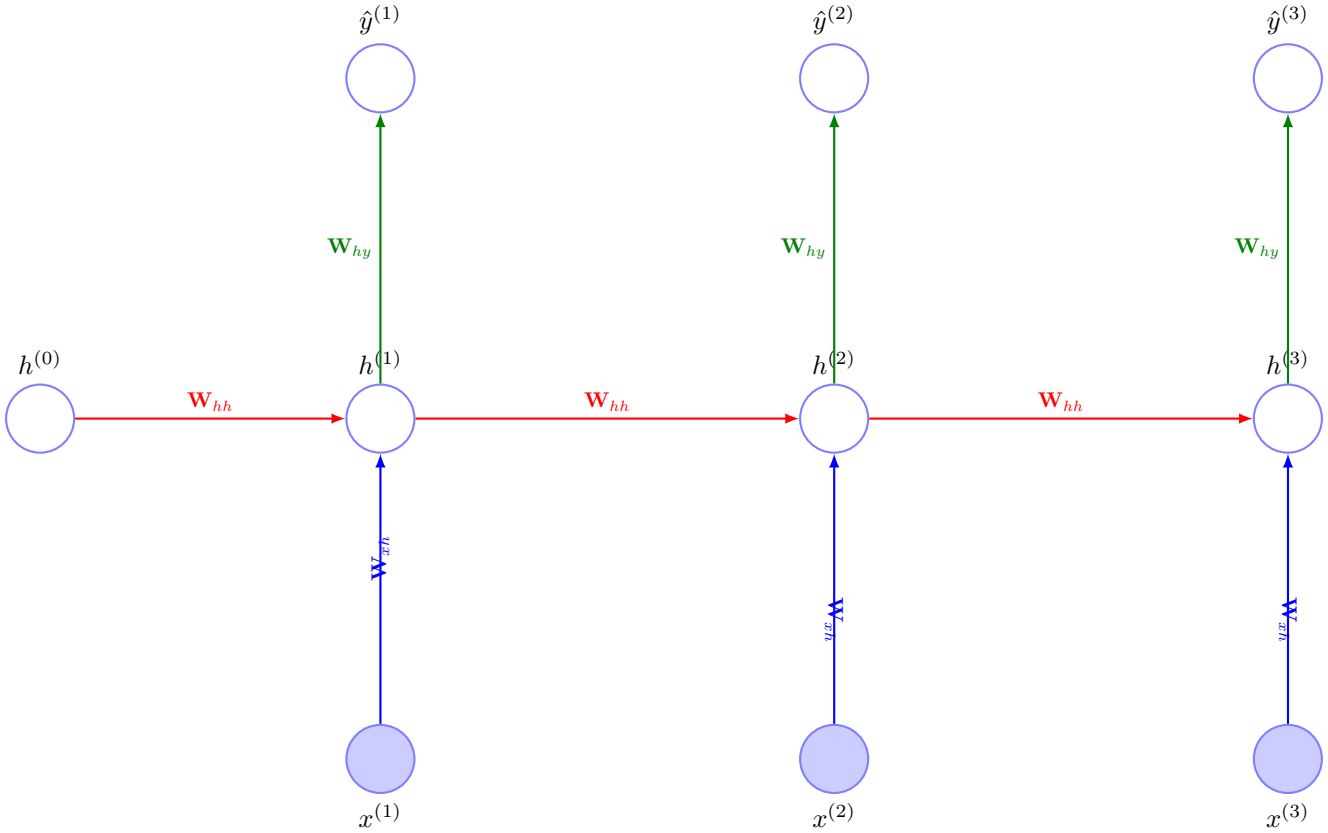


Figure 8.4: An RNN with 1 hidden layer, "unfolded" for $T = 3$ time steps. This is now a deep feed-forward network with 3 layers. Critically, the weights for each connection type ($\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hy}$) are **shared** across all time steps.

Chapter 9

Ndtwork Training

9.1 Defining the Loss Function

Training an RNN requires a mathematical objective to minimize. Because an RNN produces an output $\hat{\mathbf{y}}^{(t)}$ and has a target $\mathbf{y}^{(t)}$ at *every* time step, the loss function is typically a sum over all time steps.

9.1.1 The Per-Time-Step Loss: $E^{(t)}$

First, we define a "local" loss $E^{(t)}$ at a single time step t. The formula for $E^{(t)}$ depends on the task.

Case 1: Regression

For predicting continuous values, we use the **Sum-of-Squares Error**.

- **Target $\mathbf{y}^{(t)}$:** A vector of real numbers.
- **Prediction $\hat{\mathbf{y}}^{(t)}$:** The network's output (typically from a linear/identity activation $g(\mathbf{o}^{(t)}) = \mathbf{o}^{(t)}$).
- **Loss at t:**

$$E^{(t)} = \frac{1}{2}||\hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}||_2^2 \tag{9.1}$$

  This is the squared Euclidean distance between the prediction and the target.

Case 2: Classification

For predicting a class from a vocabulary of D_y classes, we use the **Cross-Entropy Error**.

- **Target $\mathbf{y}^{(t)}$:** A one-hot vector (e.g., $[0, \dots, 1, \dots, 0]^T$).
- **Prediction $\hat{\mathbf{y}}^{(t)}$:** A vector of probabilities (from a $g(\cdot) =$ softmax activation).
- **Loss at t:**

$$E^{(t)} = L(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)}) = -\sum_{k=1}^{D_y} y_k^{(t)} \log \hat{y}_k^{(t)} \tag{9.2}$$

  where k is the index for the output dimension (e.g., the k-th word in the vocabulary).

9.1.2 The Total Loss: E

The total loss E for the entire sequence is the **sum of the per-time-step losses**.

$$E = \sum_{t=1}^{T} E^{(t)} \tag{9.3}$$

where T is the total length of the sequence. This single scalar value is the final objective function that we will minimize.

9.1.3 Significance for Gradient Calculation

This summation structure is the key to deriving the training algorithm (BPTT). When we compute the gradient for a shared weight matrix (like $\mathbf{W}_{hh}$), we must use the sum rule of differentiation:

$$\frac{\partial E}{\partial \mathbf{W}_{hh}} = \frac{\partial}{\partial \mathbf{W}_{hh}} \left(\sum_{t=1}^{T} E^{(t)} \right) = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}} \tag{9.4}$$

This means the total gradient for a shared weight is the **sum of its individual gradient contributions** from each time step.

9.2 Module 2.2: Deriving the Gradients (BPTT)

We derive the gradients for the shared weight matrices by applying the chain rule to the unfolded graph (Module 1.3) and the total loss function (Module 2.1).

The total loss is $E = \sum_{t=1}^{T} E^{(t)}$. The total gradient for any shared weight matrix $\mathbf{W}$ is the sum of its gradient contributions from each time step:

$$\frac{\partial E}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{W}} \tag{9.5}$$

9.2.1 Case 1: Gradient for Output Weights ($\mathbf{W}_{hy}$)

- **Logic:** The weights $\mathbf{W}_{hy}$ are used at time t to compute $\mathbf{o}^{(t)}$ from $\mathbf{h}^{(t)}$. These weights *only* influence the loss at the same time step, $E^{(t)}$. The gradient calculation is therefore "local in time" and does not require recursive backpropagation.

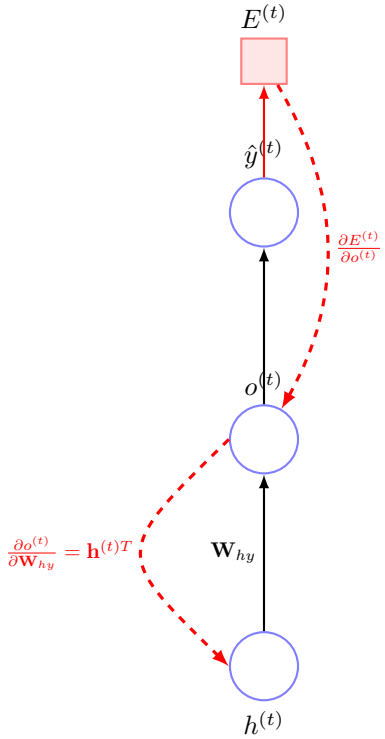


Figure 9.1: The gradient for $\mathbf{W}_{hy}$ is local. It only depends on the error at time t ($\frac{\partial E^{(t)}}{\partial o^{(t)}}$) and the hidden state at time t ($\mathbf{h}^{(t)}$).

Mathematical Derivation (per time step t)

We find the gradient $\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}}$ using the chain rule.

$$\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{W}_{hy}}$$

Let's analyze the two terms:

1. **Error w.r.t. Output Activation:** This first term, $\frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}}$, is the "error signal" from the top.
   - For softmax + cross-entropy: $\frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} = \hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}$

   This is a $D_y \times 1$ column vector.
2. **Activation w.r.t. Weights:** The forward equation is $\mathbf{o}^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y$. This is a standard linear transformation. The derivative of a linear function $\mathbf{y} = \mathbf{W}\mathbf{x}$ w.r.t. $\mathbf{W}$ is $\mathbf{x}^T$.

$$\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{W}_{hy}} = \mathbf{h}^{(t)T}$$

   This is a $1 \times D_h$ row vector.

Final Gradient Calculation

The gradient $\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}}$ is the **outer product** of these two terms, resulting in a $D_y \times D_h$ matrix, which matches the shape of $\mathbf{W}_{hy}$.

$$\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} = \underbrace{\left(\frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}}\right)}_{\text{dim } D_y \times 1} \underbrace{\left(\mathbf{h}^{(t)T}\right)}_{\text{dim } 1 \times D_h} \tag{9.6}$$

The **total gradient** for $\mathbf{W}_{hy}$ is the sum of these outer products over all T time steps:

$$\frac{\partial E}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \left(\frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}}\right)\left(\mathbf{h}^{(t)T}\right) \tag{9.7}$$

Remark 9.1. The gradient for the output bias $\mathbf{b}_y$ is even simpler, as $\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{b}_y} = 1$. $\frac{\partial E}{\partial \mathbf{b}_y} = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}}$

9.3 Learn with example

9.3.1 Part 1: The Problem Definition

We will derive all gradients for a standard Recurrent Neural Network on a sequence of length $T = 2$.

1.1: Architecture

- **Input Dimension (D_x):** 2 (e.g., $\mathbf{x} = [x_1, x_2]^T$)
- **Hidden Dimension (D_h):** 2 (e.g., $\mathbf{h} = [h_1, h_2]^T$)
- **Output Dimension (D_y):** 1 (e.g., $\hat{y}$ is a scalar)
- **Sequence Length (T):** 2 (i.e., time steps $t = 1$ and $t = 2$)

1.2: Mathematical Equations

- **Activation Functions:**
  - Hidden layer: $f(\mathbf{a}) = \tanh(\mathbf{a})$ (applied element-wise)
  - Output layer: $g(\mathbf{o}) = \mathbf{o}$ (Identity, for regression)
- **Forward Pass Equations (General):**

$$\mathbf{a}^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h \tag{9.8}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \tag{9.9}$$

$$\mathbf{o}^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y \tag{9.10}$$

$$\hat{\mathbf{y}}^{(t)} = \mathbf{o}^{(t)} \tag{9.11}$$

- **Loss Function (Total):** The total error E is the sum of the per-time-step Sum-of-Squares errors $E^{(t)}$.

$$E^{(t)} = \frac{1}{2}(\hat{y}^{(t)} - y^{(t)})^2 \tag{9.12}$$

$$E = E^{(1)} + E^{(2)} = \sum_{t=1}^{2} E^{(t)} \tag{9.13}$$

1.3: The Unfolded Computational Graph (T=2)

This is the "deep" feed-forward network we will use for backpropagation.

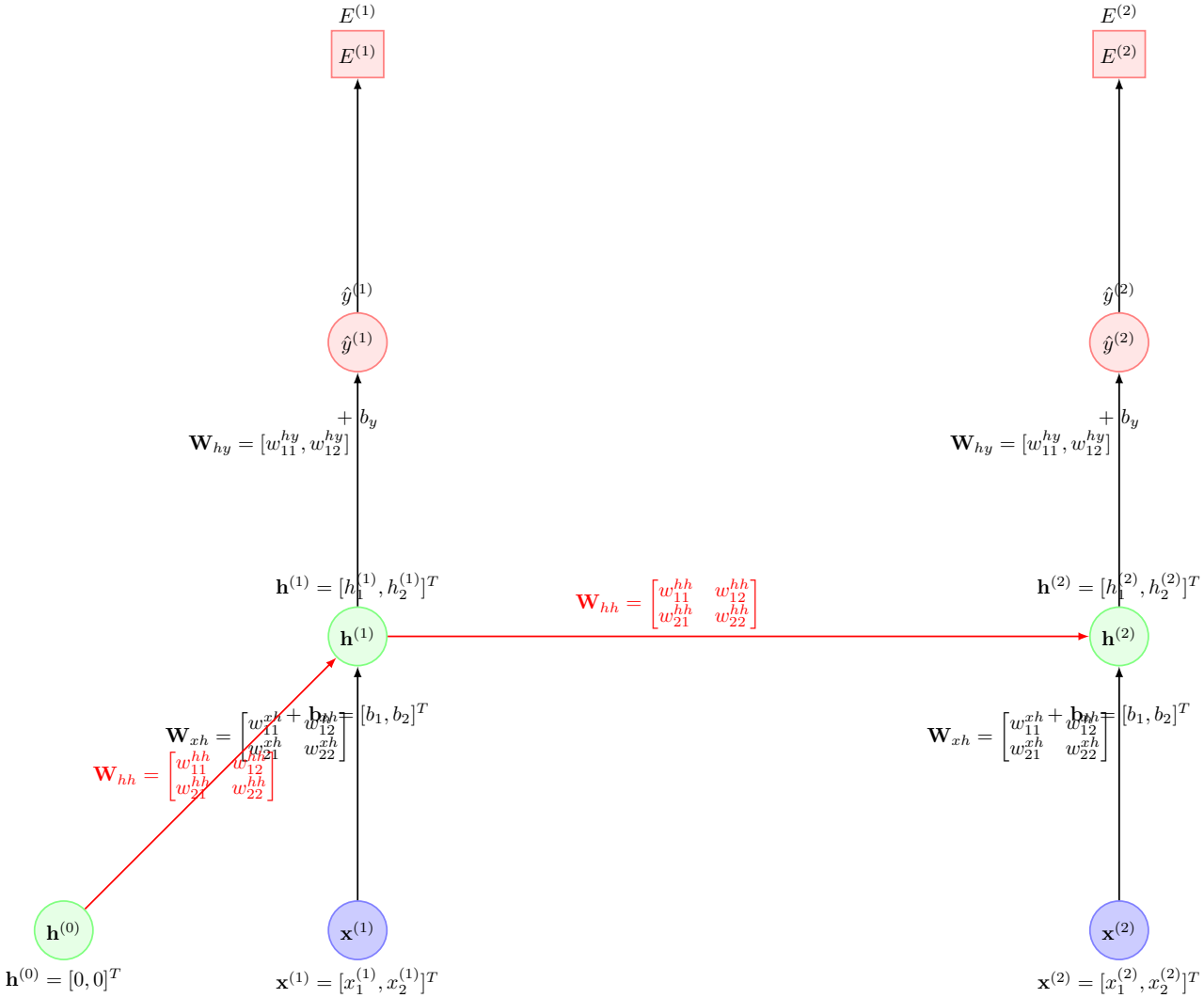


Figure 9.2: The 2-2-1 RNN, unfolded for $T = 2$. Note the parameter sharing: the *same* matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{hy}$ are used at both time steps.

9.3.2 Part 2: The Numerical Example

2.1: Initial Values

- **Data (T=2):**
  - $t = 1$: $\mathbf{x}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $y^{(1)} = 0.5$
  - $t = 2$: $\mathbf{x}^{(2)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $y^{(2)} = 0.9$
- **Initial State:** $\mathbf{h}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- **Weights:**
  - $\mathbf{W}_{xh} = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.4 \end{bmatrix}$
  - $\mathbf{W}_{hh} = \begin{bmatrix} 0.3 & 0.6 \\ 0.7 & 0.1 \end{bmatrix}$
  - $\mathbf{W}_{hy} = \begin{bmatrix} 0.8 & 0.4 \end{bmatrix}$
- **Biases:**
  - $\mathbf{b}_h = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$
  - $b_y = 0.0$

- **Learning Rate:** $\eta = 0.1$
- **Needed Derivatives:**
  - $f(a) = \tanh(a) \implies f'(a) = 1 - \tanh^2(a) = 1 - (f(a))^2$
  - $g(o) = o \implies g'(o) = 1$

9.3.3 Part 3: Forward Pass (Numerical Calculation)

3.1: Time Step $t = 1$

$$\begin{aligned}\mathbf{a}^{(1)} &= \mathbf{W}_{hh}\mathbf{h}^{(0)} + \mathbf{W}_{xh}\mathbf{x}^{(1)} + \mathbf{b}_h \\ &= \begin{bmatrix}0.3 & 0.6\\ 0.7 & 0.1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0.5 & 0.1\\ 0.2 & 0.4\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0.1\\0.1\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0.5\\0.2\end{bmatrix} + \begin{bmatrix}0.1\\0.1\end{bmatrix} = \begin{bmatrix}0.6\\0.3\end{bmatrix}\end{aligned}$$

$$\mathbf{h}^{(1)} = \tanh(\mathbf{a}^{(1)}) = \begin{bmatrix}\tanh(0.6)\\ \tanh(0.3)\end{bmatrix} = \begin{bmatrix}0.537\\0.291\end{bmatrix}$$

$$\begin{aligned}\mathbf{o}^{(1)} &= \mathbf{W}_{hy}\mathbf{h}^{(1)} + b_y \\ &= \begin{bmatrix}0.8 & 0.4\end{bmatrix}\begin{bmatrix}0.537\\0.291\end{bmatrix} + 0 = (0.8 \cdot 0.537) + (0.4 \cdot 0.291) = 0.4296 + 0.1164 = 0.546\end{aligned}$$

$$\hat{y}^{(1)} = \mathbf{o}^{(1)} = 0.546$$

$$E^{(1)} = \frac{1}{2}(\hat{y}^{(1)} - y^{(1)})^2 = \frac{1}{2}(0.546 - 0.5)^2 = \frac{1}{2}(0.046)^2 = 0.001058$$

3.2: Time Step $t = 2$

$$\begin{aligned}\mathbf{a}^{(2)} &= \mathbf{W}_{hh}\mathbf{h}^{(1)} + \mathbf{W}_{xh}\mathbf{x}^{(2)} + \mathbf{b}_h \\ &= \begin{bmatrix}0.3 & 0.6\\ 0.7 & 0.1\end{bmatrix}\begin{bmatrix}0.537\\0.291\end{bmatrix} + \begin{bmatrix}0.5 & 0.1\\ 0.2 & 0.4\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}0.1\\0.1\end{bmatrix} \\ &= \begin{bmatrix}(0.3 \cdot 0.537) + (0.6 \cdot 0.291)\\ (0.7 \cdot 0.537) + (0.1 \cdot 0.291)\end{bmatrix} + \begin{bmatrix}0.1\\0.4\end{bmatrix} + \begin{bmatrix}0.1\\0.1\end{bmatrix} = \begin{bmatrix}0.1611 + 0.1746\\ 0.3759 + 0.0291\end{bmatrix} + \begin{bmatrix}0.1\\0.4\end{bmatrix} + \begin{bmatrix}0.1\\0.1\end{bmatrix} \\ &= \begin{bmatrix}0.3357\\0.405\end{bmatrix} + \begin{bmatrix}0.2\\0.5\end{bmatrix} = \begin{bmatrix}0.5357\\0.905\end{bmatrix}\end{aligned}$$

$$\mathbf{h}^{(2)} = \tanh(\mathbf{a}^{(2)}) = \begin{bmatrix}\tanh(0.5357)\\ \tanh(0.905)\end{bmatrix} = \begin{bmatrix}0.490\\0.719\end{bmatrix}$$

$$\begin{aligned}\mathbf{o}^{(2)} &= \mathbf{W}_{hy}\mathbf{h}^{(2)} + b_y \\ &= \begin{bmatrix}0.8 & 0.4\end{bmatrix}\begin{bmatrix}0.490\\0.719\end{bmatrix} + 0 = (0.8 \cdot 0.490) + (0.4 \cdot 0.719) = 0.392 + 0.2876 = 0.6796\end{aligned}$$

$$\hat{y}^{(2)} = \mathbf{o}^{(2)} = 0.6796$$

$$E^{(2)} = \frac{1}{2}(\hat{y}^{(2)} - y^{(2)})^2 = \frac{1}{2}(0.6796 - 0.9)^2 = \frac{1}{2}(-0.2204)^2 = 0.02429$$

3.3: Total Loss

$$E = E^{(1)} + E^{(2)} = 0.001058 + 0.02429 = \mathbf{0.025348}$$

9.3.4 Part 4: Backward Pass (BPTT Derivations & Calculations)

Our goal is to compute $\frac{\partial E}{\partial \mathbf{W}_{hy}}$, $\frac{\partial E}{\partial \mathbf{W}_{hh}}$, $\frac{\partial E}{\partial \mathbf{W}_{xh}}$, and the bias gradients. We define our "delta" error signals as:

- $\delta_o^{(t)} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}}$ (Error at the output activation)
- $\delta_h^{(t)} = \frac{\partial E}{\partial \mathbf{a}^{(t)}}$ (Total error at the hidden activation)

4.1: Gradient for Output Weights $\mathbf{W}_{hy}$ and $\mathbf{b}_y$

$$\frac{\partial E}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{2} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} \quad \text{and} \quad \frac{\partial E}{\partial \mathbf{b}_y} = \sum_{t=1}^{2} \frac{\partial E^{(t)}}{\partial \mathbf{b}_y}$$

Step 4.1.1: General Derivation (per time step t)

$$\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{W}_{hy}} \quad \text{and} \quad \frac{\partial E^{(t)}}{\partial \mathbf{b}_y} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{b}_y}$$

- $\delta_o^{(t)} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} = \frac{\partial}{\partial \mathbf{o}^{(t)}} \left[\frac{1}{2}(\hat{y}^{(t)} - y^{(t)})^2\right] = (\hat{y}^{(t)} - y^{(t)}) \cdot \frac{\partial \hat{y}^{(t)}}{\partial \mathbf{o}^{(t)}} = (\hat{y}^{(t)} - y^{(t)}) \cdot 1 = (\hat{y}^{(t)} - y^{(t)})$
- From $\mathbf{o}^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y$:
  - $\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{W}_{hy}} = (\mathbf{h}^{(t)})^T$
  - $\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{b}_y} = 1$

So, $\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hy}} = \delta_o^{(t)}(\mathbf{h}^{(t)})^T$ (outer product) and $\frac{\partial E^{(t)}}{\partial \mathbf{b}_y} = \delta_o^{(t)}$.

Step 4.1.2: Numerical Calculation

- $\delta_o^{(1)} = \hat{y}^{(1)} - y^{(1)} = 0.546 - 0.5 = 0.046$
- $\delta_o^{(2)} = \hat{y}^{(2)} - y^{(2)} = 0.6796 - 0.9 = -0.2204$

Gradients for $t = 1$:

- $\frac{\partial E^{(1)}}{\partial \mathbf{W}_{hy}} = \delta_o^{(1)}(\mathbf{h}^{(1)})^T = (0.046) \begin{bmatrix} 0.537 & 0.291 \end{bmatrix} = \begin{bmatrix} 0.0247 & 0.0134 \end{bmatrix}$
- $\frac{\partial E^{(1)}}{\partial \mathbf{b}_y} = \delta_o^{(1)} = 0.046$

Gradients for $t = 2$:

- $\frac{\partial E^{(2)}}{\partial \mathbf{W}_{hy}} = \delta_o^{(2)}(\mathbf{h}^{(2)})^T = (-0.2204) \begin{bmatrix} 0.490 & 0.719 \end{bmatrix} = \begin{bmatrix} -0.108 & -0.1585 \end{bmatrix}$
- $\frac{\partial E^{(2)}}{\partial \mathbf{b}_y} = \delta_o^{(2)} = -0.2204$

Total Gradients (Final Answer):

$$\frac{\partial E}{\partial \mathbf{W}_{hy}} = \begin{bmatrix} 0.0247 & 0.0134 \end{bmatrix} + \begin{bmatrix} -0.108 & -0.1585 \end{bmatrix} = \begin{bmatrix} -0.0833 & -0.1451 \end{bmatrix}$$

$$\frac{\partial E}{\partial \mathbf{b}_y} = 0.046 + (-0.2204) = \mathbf{-0.1744}$$

4.2: Gradients for Hidden Weights $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$, $\mathbf{b}_h$

This is the core of BPTT. We need to find $\delta_h^{(t)} = \frac{\partial E}{\partial \mathbf{a}^{(t)}}$ for $t = 1, 2$. We compute this *recursively*, starting from $t = T$ (i.e., $t = 2$) and moving backward.

Step 4.2.1: General Derivation of $\delta_h^{(t)}$ The error E is affected by $\mathbf{a}^{(t)}$ through two paths:

1. The output at the same step: $\mathbf{a}^{(t)} \to \mathbf{h}^{(t)} \to \mathbf{o}^{(t)} \to E^{(t)}$
2. The hidden state at the *next* step: $\mathbf{a}^{(t)} \to \mathbf{h}^{(t)} \to \mathbf{a}^{(t+1)} \to \cdots \to E$

$$\delta_h^{(t)} = \frac{\partial E}{\partial \mathbf{a}^{(t)}} = \frac{\partial E}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{a}^{(t)}}$$
$$\text{where } \frac{\partial E}{\partial \mathbf{h}^{(t)}} = \frac{\partial E^{(t)}}{\partial \mathbf{h}^{(t)}} + \frac{\partial E}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}}$$

Let's find the derivatives of the paths:

- **Path 1 (Vertical):** $\frac{\partial E^{(t)}}{\partial \mathbf{h}^{(t)}} = \frac{\partial E^{(t)}}{\partial \mathbf{o}^{(t)}} \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} = \delta_o^{(t)} \mathbf{W}_{hy}$
- **Path 2 (Horizontal):** $\frac{\partial E}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} = \left(\frac{\partial E}{\partial \mathbf{a}^{(t+1)}} \frac{\partial \mathbf{a}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right) = (\delta_h^{(t+1)})^T \mathbf{W}_{hh}$ (Note: Transposes needed for correct matrix-vector alignment)
- **Local Derivative:** $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{a}^{(t)}} = f'(\mathbf{a}^{(t)}) = \text{diag}(1 - (\mathbf{h}^{(t)})^2)$ (element-wise square)

Combining these gives the BPTT recurrence relation (written for element-wise operation):

$$\delta_h^{(t)} = \left((\mathbf{W}_{hy})^T \delta_o^{(t)} + (\mathbf{W}_{hh})^T \delta_h^{(t+1)} \right) \odot f'(\mathbf{a}^{(t)}) \tag{9.14}$$

where $\odot$ is element-wise multiplication.

Step 4.2.2: Numerical Calculation of $\delta_h^{(t)}$ **For $t = 2$ (Base Case):** The recursion stops here. $\delta_h^{(3)}$ (from the "future") is $\mathbf{0}$.

$$f'(\mathbf{a}^{(2)}) = 1 - (\mathbf{h}^{(2)})^2 = 1 - \begin{bmatrix} 0.490 \\ 0.719 \end{bmatrix}^{\odot 2} = 1 - \begin{bmatrix} 0.240 \\ 0.517 \end{bmatrix} = \begin{bmatrix} 0.760 \\ 0.483 \end{bmatrix}$$
$$\begin{aligned} \delta_h^{(2)} &= \left((\mathbf{W}_{hy})^T \delta_o^{(2)} + \mathbf{0} \right) \odot f'(\mathbf{a}^{(2)}) \\ &= \left(\begin{bmatrix} 0.8 \\ 0.4 \end{bmatrix} (-0.2204) \right) \odot \begin{bmatrix} 0.760 \\ 0.483 \end{bmatrix} \\ &= \begin{bmatrix} -0.1763 \\ -0.0882 \end{bmatrix} \odot \begin{bmatrix} 0.760 \\ 0.483 \end{bmatrix} = \begin{bmatrix} -0.1339 \\ -0.0426 \end{bmatrix} \end{aligned}$$

For $t = 1$ (Recursive Step): Now we use $\delta_h^{(2)}$ to find $\delta_h^{(1)}$.

$$f'(\mathbf{a}^{(1)}) = 1 - (\mathbf{h}^{(1)})^2 = 1 - \begin{bmatrix} 0.537 \\ 0.291 \end{bmatrix}^{\odot 2} = 1 - \begin{bmatrix} 0.288 \\ 0.085 \end{bmatrix} = \begin{bmatrix} 0.712 \\ 0.915 \end{bmatrix}$$
$$\delta_h^{(1)} = \left((\mathbf{W}_{hy})^T \delta_o^{(1)} + (\mathbf{W}_{hh})^T \delta_h^{(2)} \right) \odot f'(\mathbf{a}^{(1)})$$

First, the term in parentheses:

$$\begin{aligned} (\dots) &= \begin{bmatrix} 0.8 \\ 0.4 \end{bmatrix} (0.046) + \begin{bmatrix} 0.3 & 0.7 \\ 0.6 & 0.1 \end{bmatrix} \begin{bmatrix} -0.1339 \\ -0.0426 \end{bmatrix} \\ &= \begin{bmatrix} 0.0368 \\ 0.0184 \end{bmatrix} + \begin{bmatrix} (0.3 \cdot -0.1339) + (0.7 \cdot -0.0426) \\ (0.6 \cdot -0.1339) + (0.1 \cdot -0.0426) \end{bmatrix} \\ &= \begin{bmatrix} 0.0368 \\ 0.0184 \end{bmatrix} + \begin{bmatrix} -0.0402 - 0.0298 \\ -0.0803 - 0.0043 \end{bmatrix} \\ &= \begin{bmatrix} 0.0368 \\ 0.0184 \end{bmatrix} + \begin{bmatrix} -0.0700 \\ -0.0846 \end{bmatrix} = \begin{bmatrix} -0.0332 \\ -0.0662 \end{bmatrix} \end{aligned}$$

Now, the final element-wise multiplication:

$$\delta_h^{(1)} = \begin{bmatrix} -0.0332 \\ -0.0662 \end{bmatrix} \odot \begin{bmatrix} 0.712 \\ 0.915 \end{bmatrix} = \begin{bmatrix} -0.0236 \\ -0.0606 \end{bmatrix}$$

Step 4.2.3: Final Gradient Calculation (W, b) Now we have $\delta_h^{(1)}$ and $\delta_h^{(2)}$, we can find the gradients.

Total Gradient for $\mathbf{W}_{xh}$:

$$\begin{aligned}\frac{\partial E}{\partial \mathbf{W}_{xh}} &= \sum_{t=1}^{2} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{2} \delta_h^{(t)} (\mathbf{x}^{(t)})^T = \delta_h^{(1)} (\mathbf{x}^{(1)})^T + \delta_h^{(2)} (\mathbf{x}^{(2)})^T \\ &= \begin{bmatrix} -0.0236 \\ -0.0606 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} -0.1339 \\ -0.0426 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -0.0236 & 0 \\ -0.0606 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -0.1339 \\ 0 & -0.0426 \end{bmatrix} = \begin{bmatrix} -0.0236 & -0.1339 \\ -0.0606 & -0.0426 \end{bmatrix}\end{aligned}$$

Total Gradient for $\mathbf{W}_{hh}$:

$$\begin{aligned}\frac{\partial E}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^{2} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{2} \delta_h^{(t)} (\mathbf{h}^{(t-1)})^T = \delta_h^{(1)} (\mathbf{h}^{(0)})^T + \delta_h^{(2)} (\mathbf{h}^{(1)})^T \\ &= \begin{bmatrix} -0.0236 \\ -0.0606 \end{bmatrix} \begin{bmatrix} 0 & 0 \end{bmatrix} + \begin{bmatrix} -0.1339 \\ -0.0426 \end{bmatrix} \begin{bmatrix} 0.537 & 0.291 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} -0.1339 \cdot 0.537 & -0.1339 \cdot 0.291 \\ -0.0426 \cdot 0.537 & -0.0426 \cdot 0.291 \end{bmatrix} \\ &= \begin{bmatrix} -0.0719 & -0.0390 \\ -0.0229 & -0.0124 \end{bmatrix}\end{aligned}$$

Total Gradient for $\mathbf{b}_h$:

$$\frac{\partial E}{\partial \mathbf{b}_h} = \sum_{t=1}^{2} \frac{\partial E^{(t)}}{\partial \mathbf{b}_h} = \sum_{t=1}^{2} \delta_h^{(t)} = \begin{bmatrix} -0.0236 \\ -0.0606 \end{bmatrix} + \begin{bmatrix} -0.1339 \\ -0.0426 \end{bmatrix} = \begin{bmatrix} -0.1575 \\ -0.1032 \end{bmatrix}$$

9.3.5 Part 5: Weight Update (Example)

We use the gradients from Part 4 and the learning rate $\eta = 0.1$ to update the shared weights.

Update for $\mathbf{W}_{hh}$:

$$\begin{aligned}\mathbf{W}_{hh}^{(\text{new})} &= \mathbf{W}_{hh}^{(\text{old})} - \eta \frac{\partial E}{\partial \mathbf{W}_{hh}} \\ &= \begin{bmatrix}0.3 & 0.6 \\ 0.7 & 0.1\end{bmatrix} - (0.1)\begin{bmatrix}-0.0719 & -0.0390 \\ -0.0229 & -0.0124\end{bmatrix} \\ &= \begin{bmatrix}0.3 & 0.6 \\ 0.7 & 0.1\end{bmatrix} + \begin{bmatrix}0.0072 & 0.0039 \\ 0.0023 & 0.0012\end{bmatrix} \\ &= \begin{bmatrix}0.3072 & 0.6039 \\ 0.7023 & 0.1012\end{bmatrix}\end{aligned}$$

Update for $\mathbf{W}_{hy}$:

$$\begin{aligned}\mathbf{W}_{hy}^{(\text{new})} &= \mathbf{W}_{hy}^{(\text{old})} - \eta \frac{\partial E}{\partial \mathbf{W}_{hy}} \\ &= \begin{bmatrix}0.8 & 0.4\end{bmatrix} - (0.1)\begin{bmatrix}-0.0833 & -0.1451\end{bmatrix} \\ &= \begin{bmatrix}0.8 & 0.4\end{bmatrix} + \begin{bmatrix}0.0083 & 0.0145\end{bmatrix} \\ &= \begin{bmatrix}0.8083 & 0.4145\end{bmatrix}\end{aligned}$$

The same update step is applied to $\mathbf{W}_{xh}$ and the biases. This completes one full iteration of training.

9.4 The Recursive Jacobian Product

This module analyzes the mathematical properties of the BPTT gradients we derived in Module 2.2.

The total gradient for the recurrent weights $\mathbf{W}_{hh}$ is a sum over time:

$$\frac{\partial E}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}} \tag{9.15}$$

Let's analyze the contribution from a single loss $E^{(t)}$ (at time t) to the weights used at a much earlier time step k (where $k < t$). To do this, the chain rule must propagate the gradient all the way back from t to k.

$$\left.\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}}\right|_{\text{at time } k} \propto \frac{\partial E^{(t)}}{\partial \mathbf{h}^{(t)}} \underbrace{\left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \frac{\partial \mathbf{h}^{(t-1)}}{\partial \mathbf{h}^{(t-2)}} \cdots \frac{\partial \mathbf{h}^{(k+1)}}{\partial \mathbf{h}^{(k)}}\right)}_{\text{The Problem: A Long Product of Jacobians}} \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \tag{9.16}$$

The core of the problem lies in the long product of matrices. Let's define this product:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} \tag{9.17}$$

9.4.1 The Jacobian of the State Transition

We must first define the single-step Jacobian matrix, $\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$.

$$\begin{aligned}\text{Recall:} \quad \mathbf{h}^{(i)} &= f(\mathbf{a}^{(i)}) \\ \mathbf{a}^{(i)} &= \mathbf{W}_{hh}\mathbf{h}^{(i-1)} + \mathbf{W}_{xh}\mathbf{x}^{(i)} + \mathbf{b}_h\end{aligned}$$

Using the chain rule:

$$\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} = \underbrace{\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{a}^{(i)}}}_{\text{Term 1}} \cdot \underbrace{\frac{\partial \mathbf{a}^{(i)}}{\partial \mathbf{h}^{(i-1)}}}_{\text{Term 2}}$$

1. **Term 1:** The derivative of the element-wise activation function f w.r.t. its input $\mathbf{a}^{(i)}$ is a diagonal matrix of the individual derivatives:
$$\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{a}^{(i)}} = \text{diag}(f'(\mathbf{a}^{(i)}))$$

2. **Term 2:** The derivative of the linear activation $\mathbf{a}^{(i)}$ w.r.t. the previous state $\mathbf{h}^{(i-1)}$ is simply the recurrent weight matrix:

$$\frac{\partial \mathbf{a}^{(i)}}{\partial \mathbf{h}^{(i-1)}} = \mathbf{W}_{hh}$$

Substituting these back, the long product of Jacobians becomes:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \left(\text{diag}(f'(\mathbf{a}^{(i)})) \mathbf{W}_{hh} \right) \tag{9.18}$$

This is a product of $(t-k)$ matrices. This long product is the mathematical source of the two major problems in training RNNs.

—

9.5 The Problem of Long-Range Dependencies

The long product of Jacobians is mathematically unstable. It leads to two related problems.

9.5.1 3.1 Vanishing Gradients

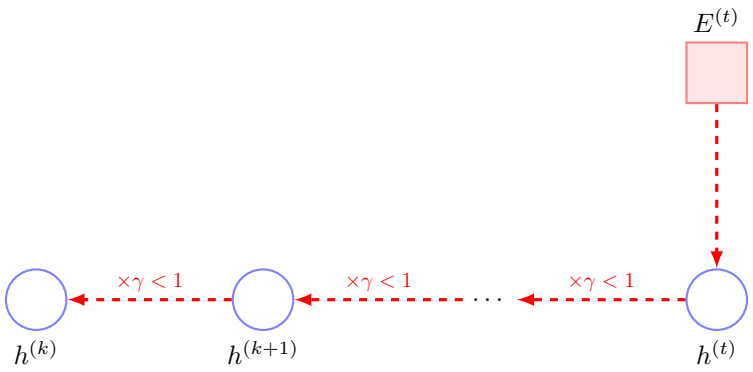
- **The Cause:** The derivative of the tanh activation function is $f'(a) = 1 - \tanh^2(a)$, which is *always* in the range $(0, 1]$.
- The $\text{diag}(f'(\mathbf{a}^{(i)}))$ matrix will have values ≤ 1 on its diagonal.
- If the weights in $\mathbf{W}_{hh}$ are "small" (e.g., its largest singular value $\gamma_1 < 1$), the norm (or magnitude) of the entire Jacobian $\left\| \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} \right\|$ will be less than 1.
- **The Math:** When you multiply a number less than 1 by itself many times, it vanishes exponentially:

$$\text{e.g., } (0.9)^{100} \approx 0.000026$$

Therefore, for a large time gap $(t-k)$:

$$\lim_{(t-k)\to\infty} \left\| \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \right\| = 0 \tag{9.19}$$

- **The Consequence:** The gradient from a future loss $E^{(t)}$ cannot flow back to a distant past state $\mathbf{h}^{(k)}$. The gradient $\frac{\partial E^{(t)}}{\partial \mathbf{W}_{hh}}\Big|_{\text{at time } k}$ becomes zero. The network becomes mathematically **unable to learn long-range dependencies**.



Gradient signal decays exponentially to zero

Figure 9.3: The Vanishing Gradient Problem. The error signal from $E^{(t)}$ is repeatedly multiplied by a Jacobian matrix with a norm < 1, causing it to vanish to zero before it reaches $h^{(k)}$.

9.5.2 3.2 Exploding Gradients

- **The Cause:** If the weights in $\mathbf{W}_{hh}$ are "large" (e.g., its largest singular value $\gamma_1 > 1$), the norm of the Jacobian $\left\| \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} \right\|$ can be greater than 1.
- **The Math:** When you multiply a number greater than 1 by itself many times, it explodes exponentially:

  $$\text{e.g., } (1.1)^{100} \approx 13780$$

  Therefore, for a large time gap $(t-k)$:

  $$\lim_{(t-k)\to\infty} \left\| \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \right\| = \infty \tag{9.20}$$

- **The Consequence:** The gradients become massive, resulting in numerical overflow ('Inf' or 'NaN'). The optimization process becomes completely unstable and collapses.

9.5.3 3.3 Solution for Exploding Gradients: Gradient Clipping

While vanishing gradients are the harder problem (requiring a new architecture), exploding gradients have a simple, robust solution.

Definition 9.2 (Gradient Clipping)**.** We cap the *magnitude* of the total gradient vector $\mathbf{g} = \frac{\partial E}{\partial \mathbf{W}}$ before the update step. We choose a maximum threshold value, θ.

$$\text{If } ||\mathbf{g}|| > \theta, \quad \text{then set} \quad \mathbf{g} \leftarrow \frac{\theta}{||\mathbf{g}||}\mathbf{g} \tag{9.21}$$

Remark 9.3. This operation does not change the *direction* of the gradient; it only rescales its *magnitude* (its length) to be no larger than θ. This prevents the "jumping" that leads to 'NaN' values and stabilizes training.

Chapter 10

Gated RNNs

10.1 The Gating Mechanism

10.1.1 The Problem: The Unstable Gradient Path

From Module 3, we established that the gradient of the loss E at time t with respect to a hidden state at time k ($k < t$) requires a long product of Jacobian matrices:

$$\frac{\partial E^{(t)}}{\partial \mathbf{h}^{(k)}} \propto \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} \tag{10.1}$$

Where the Jacobian of the state transition is:

$$\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} = \text{diag}\left(f'(\mathbf{a}^{(i)})\right) \mathbf{W}_{hh} \tag{10.2}$$

This repeated *matrix multiplication* by $\mathbf{W}_{hh}$ is mathematically unstable and leads to:

- **Vanishing Gradients:** If the singular values of the Jacobian are < 1, the gradient norm shrinks exponentially to zero.
- **Exploding Gradients:** If the singular values are > 1, the gradient norm grows exponentially to infinity.

The network is therefore unable to learn long-range dependencies.

10.1.2 The Conceptual Solution: An Additive "Cell State"

The core problem is that the "memory" $\mathbf{h}^{(t)}$ and the "computation" (multiplication by $\mathbf{W}_{hh}$) are inseparably mixed. We can solve this by creating a separate "memory superhighway," called the **Cell State** $\mathbf{C}^{(t)}$, that operates via simple *addition*.

Consider a hypothetical, simple cell state update:

$$\mathbf{C}^{(t)} = \mathbf{C}^{(t-1)} + \text{some new information}$$

Let's analyze the gradient flow for this state:

$$\frac{\partial \mathbf{C}^{(t)}}{\partial \mathbf{C}^{(t-1)}} = \mathbf{I} \quad \text{(The Identity Matrix)} \tag{10.3}$$

The long product of Jacobians for this path would be:

$$\frac{\partial \mathbf{C}^{(t)}}{\partial \mathbf{C}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{C}^{(i)}}{\partial \mathbf{C}^{(i-1)}} = \prod_{i=k+1}^{t} \mathbf{I} = \mathbf{I}$$

The gradient flows perfectly, without vanishing or exploding. However, this state can only add; it can never forget information. We need a mechanism to control this additive highway.

10.1.3 The Gating Mechanism: A Learnable "Soft Switch"

A **gate** is a neural network component that learns to control the flow of information. It is a "soft" switch that can be open (pass all information), closed (block all information), or partially open.

Mathematical Definition of a Gate

A gate is composed of two parts: a gate vector $\mathbf{g}$ and an element-wise multiplication $\odot$.

1. The Gate Vector g: The gate is a vector of values in the range $(0, 1)$. We compute it using a standard linear layer followed by the **logistic sigmoid** activation function, $\sigma(\cdot)$.

$$\mathbf{g} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{10.4}$$

where $\mathbf{x}$ is the input that "decides" whether the gate should be open or closed.

- If $g_i \to 0$, the i-th gate is **closed**.
- If $g_i \to 1$, the i-th gate is **open**.

2. The Gating Operation $\odot$: To apply the gate, we use **element-wise multiplication** (the Hadamard product, $\odot$) with the data vector we wish to control.

$$\mathbf{y}_{\text{gated}} = \mathbf{g} \odot \mathbf{y}_{\text{data}} \tag{10.5}$$

Numerical Example: Let $\mathbf{g}$ be a gate vector and $\mathbf{y}_{\text{data}}$ be a data vector.

$$\mathbf{g} \odot \mathbf{y}_{\text{data}} = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.7 \end{bmatrix} \odot \begin{bmatrix} 123.4 \\ 567.8 \\ 90.0 \end{bmatrix} = \begin{bmatrix} 0.0 \times 123.4 \\ 1.0 \times 567.8 \\ 0.7 \times 90.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 567.8 \\ 63.0 \end{bmatrix}$$

The gate has successfully "blocked" the first element, "passed" the second, and "attenuated" the third. Since $\mathbf{W}$ and $\mathbf{b}$ are learnable, the network can learn to control this flow.

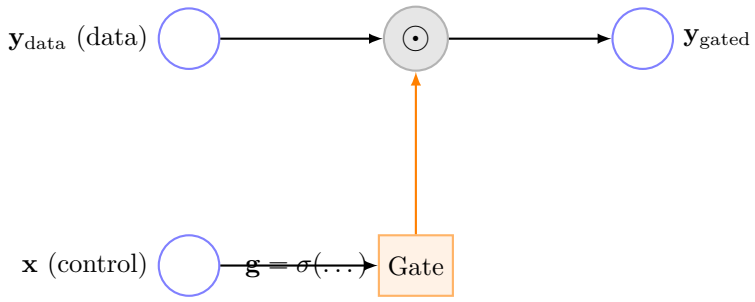


Figure 10.1: The Gating Mechanism. The input $\mathbf{x}$ is fed into a sigmoid function to create a gate vector $\mathbf{g}$. This gate $\mathbf{g}$ controls which elements of $\mathbf{y}_{\text{data}}$ are allowed to pass through via element-wise multiplication.

10.1.4 The Gated Gradient Superhighway

We now apply this gating mechanism to our ideal cell state. We introduce two gates:

- **Forget Gate ($f^{(t)}$):** Decides what to *forget* from the old cell state $C^{(t-1)}$.
- **Input Gate ($i^{(t)}$):** Decides what to *add* from the new candidate information $\tilde{C}^{(t)}$.

The simple additive equation $C^{(t)} = C^{(t-1)} + \tilde{C}^{(t)}$ becomes:

$$\mathbf{C}^{(t)} = \left(\mathbf{f}^{(t)} \odot \mathbf{C}^{(t-1)}\right) + \left(\mathbf{i}^{(t)} \odot \tilde{\mathbf{C}}^{(t)}\right) \tag{10.6}$$

This is the core equation for an LSTM cell (Module 4.2).

Gradient Analysis of the Gated Highway

This architecture solves the vanishing gradient problem. Let's analyze the new gradient path for BPTT. We are interested in the derivative of the cell state at time t with respect to the cell state at time $t-1$:

$$\frac{\partial \mathbf{C}^{(t)}}{\partial \mathbf{C}^{(t-1)}} = \frac{\partial}{\partial \mathbf{C}^{(t-1)}} \left[\left(\mathbf{f}^{(t)} \odot \mathbf{C}^{(t-1)}\right) + \left(\mathbf{i}^{(t)} \odot \tilde{\mathbf{C}}^{(t)}\right)\right]$$

(Note: $f^{(t)}, i^{(t)}$, and $\tilde{C}^{(t)}$ depend on $\mathbf{h}^{(t-1)}$, not $\mathbf{C}^{(t-1)}$ directly. We ignore these for the main path.)

$$\frac{\partial \mathbf{C}^{(t)}}{\partial \mathbf{C}^{(t-1)}} = \mathbf{f}^{(t)} \quad \text{(The Forget Gate vector)}$$

The long product of Jacobians from the simple RNN, $\prod \left(\text{diag}(f'(\mathbf{a}^{(k)}))\mathbf{W}_{hh}\right)$, is now replaced by:

$$\frac{\partial \mathbf{C}^{(t)}}{\partial \mathbf{C}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{C}^{(i)}}{\partial \mathbf{C}^{(i-1)}} = \prod_{i=k+1}^{t} \mathbf{f}^{(i)} \tag{10.7}$$

Remark 10.1 (The Solution). This is the solution to the vanishing/exploding gradient problem.

1. **No Matrix Multiplication:** The gradient path is now a series of element-wise multiplications. The unstable $\mathbf{W}_{hh}$ matrix is gone from this path.
2. **Learnable Control:** The network can *learn* to control the gradient flow. If a piece of information from time k is important at time t, the network will learn to set its forget gates $f^{(i)}$ for $i = k + 1 \dots t$ to 1.0.
3. **Gradient Flow:** When $f^{(i)} = 1.0$, the gradient from that path is $1.0 \times 1.0 \times \cdots = 1.0$. The gradient flows perfectly, allowing the network to learn long-range dependencies.

Part IV

Transformers

Chapter 11

Attention

11.1 Introduction to Attention

This section introduces the Transformer architecture, which is based on the concept of "attention."

11.1.1 Key Takeaways from the Introduction

- **What is a Transformer?** It is a neural network architecture that transforms a set of input vectors (e.g., word embeddings) into a new set of output vectors of the same dimension. The goal is to create a richer, more contextual representation of the data.
- **What is the Core Mechanism?** The core idea is **attention**. This is a mechanism that allows the network to assign different "weights" (importance) to different parts of the input. Crucially, these weights are **data-dependent**; they are calculated "on the fly" based on the input itself.
- **Why is this a Big Deal?**
  - It has **surpassed RNNs** in Natural Language Processing (NLP).
  - **Vision Transformers (ViTs)** now often outperform CNNs in image processing.
  - The architecture is highly flexible and works on text, images, audio, or any combination (**multimodal**).
- **What is its Key Advantage over RNNs?**
  - **Parallelization**. RNNs are inherently sequential (you must calculate $h^{(t)}$ before $h^{(t+1)}$), which is slow.
  - Transformers process all input tokens at once, making them "especially well suited to massively parallel processing hardware such as... GPUs".
- **How are they Trained? (The "Scaling Hypothesis")**
  - **Self-Supervision:** They are perfect for self-supervised learning, allowing them to be trained on *massive, unlabelled datasets* (like the entire internet).
  - **Scaling Laws:** The "scaling hypothesis" is a key finding: Transformers get better and better simply by *increasing their size* (more parameters) and *training on more data*, even with no other architectural changes.
  - **Foundation Models:** This process creates massive, pre-trained "foundation models" (like GPT) that can then be easily "fine-tuned" for many different downstream tasks.
- **The Result: LLMs and Emergent Properties**
  - This combination of a parallel architecture and self-supervised scaling has enabled the creation of **Large Language Models (LLMs)** with over a trillion parameters.
  - These massive models show **emergent properties**—surprising capabilities that weren't explicitly trained for, which are described as "early signs of artificial general intelligence".

11.2 Attention Coefficients

The goal of an attention layer is to map a set of N input tokens (vectors) $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ to a new set of N output tokens $\{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$, where the new representation $\mathbf{y}_n$ is richer and captures context from the entire input set.

The output vector $\mathbf{y}_n$ should depend on all input vectors, not just $\mathbf{x}_n$. The simplest way to achieve this is to define $\mathbf{y}_n$ as a linear combination of all input vectors:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m \tag{11.1}$$

where the coefficients a_{nm} are the **attention weights**. These weights define how much "attention" output token n pays to input token m.

11.2.1 Constraints on Attention Weights

To ensure this operation is a stable weighted average (a "partition of unity"), the attention weights a_{nm} must satisfy two constraints for each output n:

1. **Non-negativity:** We require all weights to be non-negative.

$$a_{nm} \geq 0 \tag{11.2}$$

   This prevents large positive and large negative coefficients from compensating for each other.

2. **Normalization:** We constrain the coefficients for a given output n to sum to unity.

$$\sum_{m=1}^{N} a_{nm} = 1 \tag{11.3}$$

   This ensures that if the model pays more attention to one input, it must pay less attention to the others.

These two constraints together imply that $0 \leq a_{nm} \leq 1$. It is important to note that this is a *different* set of N weights for *each* of the N output tokens.

The key challenge, which we will address next, is how to calculate these data-dependent attention weights a_{nm}.

Appendix: Common Activation Functions and Derivatives

This section compiles the activation functions and their derivatives used in neural network theory. The derivative $f'(a)$ is what's used in the backpropagation step.

Point-wise Activation Functions

These functions are applied element-wise to an activation a.

Identity Function

Used for regression output units.

$$f(a) = a$$
$$f'(a) = 1$$

Logistic Sigmoid

Used for binary classification output units.

$$f(a) = \sigma(a) = \frac{1}{1 + \exp(-a)}$$
$$f'(a) = \sigma(a)(1 - \sigma(a))$$

Remark .1. Range is $(0, 1)$. Not zero-centered.

Hyperbolic Tangent (tanh)

A common choice for hidden units.

$$f(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$
$$f'(a) = 1 - \tanh^2(a)$$

Remark .2. Range is $(-1, 1)$. It is an **odd function** ($f(-a) = -f(a)$) and is zero-centered, which can be beneficial for training.

Rectified Linear Unit (ReLU)

The modern default choice for hidden units (though not featured in this chapter, it is the most common in practice).

$$f(a) = \max(0, a) = \begin{cases} a, & a \geq 0 \\ 0, & a < 0 \end{cases}$$
$$f'(a) = \begin{cases} 1, & a > 0 \\ 0, & a < 0 \end{cases}$$

Remark .3. The derivative is undefined at $a = 0$, but in practice, it is set to 0 or 1.

Vector Activation Functions

This function takes the entire vector of output activations $\mathbf{a}$ as input.

Softmax

Used for K-class classification output layers.

$$y_k(\mathbf{a}) = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)}$$

The derivative is a Jacobian matrix. The partial derivative $\frac{\partial y_k}{\partial a_j}$ depends on whether $k = j$ or $k \neq j$:

$$\frac{\partial y_k}{\partial a_j} = \begin{cases} y_k(1 - y_k), & \text{if } k = j \\ -y_k y_j, & \text{if } k \neq j \end{cases}$$

or more compactly:

$$\frac{\partial y_k}{\partial a_j} = y_k(\delta_{kj} - y_j)$$

where δ_{kj} is the Kronecker delta ($\delta_{kj} = 1$ if $k = j$, 0 otherwise).