Neural Networks

Minds, Machines, and Matrices Workshop

Cedric Kiplimo

Centre for Data Science and Artificial Intelligence

Thursday, 9 July 2026

Objectives

What we will explore:

- The transition from linear models to adaptive, non-linear neural networks

Objectives

What we will explore:

- The transition from linear models to adaptive, non-linear neural networks
- The early biological inspirations for neural networks

Objectives

What we will explore:

- The transition from linear models to adaptive, non-linear neural networks
- The early biological inspirations for neural networks
- The design of Multi-Layer Perceptrons (MLPs)

Objectives

What we will explore:

- The transition from linear models to adaptive, non-linear neural networks
- The early biological inspirations for neural networks
- The design of Multi-Layer Perceptrons (MLPs)
- How MLPs (and neural networks in general) learn

Linear Models

- A typical linear regression model takes an input vector:

$$\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$$

Linear Models

- A typical linear regression model takes an input vector:

$$\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$$

- It predicts a target value using a linear combination of inputs:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b$$

Linear Models

- A typical linear regression model takes an input vector:

$$\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$$

- It predicts a target value using a linear combination of inputs:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b$$

- Where:
  - $\mathbf{w} = [w_1, w_2, \ldots, w_D]^T$ is the **weight vector**

Linear Models

- A typical linear regression model takes an input vector:

$$\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$$

- It predicts a target value using a linear combination of inputs:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b$$

- Where:
  - $\mathbf{w} = [w_1, w_2, \ldots, w_D]^T$ is the **weight vector**
  - b is the **bias parameter**

The Limitations of Linear Models

- **Advantage:** Convex loss surface makes optimisation straightforward.

The Limitations of Linear Models

- **Advantage:** Convex loss surface makes optimisation straightforward.
- **Limitation:** Can only form linear decision boundaries or linear prediction surfaces.

The Limitations of Linear Models

- **Advantage:** Convex loss surface makes optimisation straightforward.
- **Limitation:** Can only form linear decision boundaries or linear prediction surfaces.
- **High Bias (Underfitting):** Occurs if the true underlying relationship is non-linear

Fixed Basis Functions

- Basis functions extend the capability of linear models to handle non-linear data. These basis functions are non-linear transformations.

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) + b$$

Fixed Basis Functions

- Basis functions extend the capability of linear models to handle non-linear data. These basis functions are non-linear transformations.

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) + b$$

- Using vector notation: $\boldsymbol{\phi}(\mathbf{x}) = [\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_{M-1}(\mathbf{x})]^T$:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + b$$

Fixed Basis Functions

- Basis functions extend the capability of linear models to handle non-linear data. These basis functions are non-linear transformations.

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) + b$$

- Using vector notation: $\boldsymbol{\phi}(\mathbf{x}) = [\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_{M-1}(\mathbf{x})]^T$:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + b$$

Crucial Structural Property

This model is non-linear with respect to $\mathbf{x}$, but it remains **linear with respect to the parameters $\mathbf{w}$**.

Some Common Basis Functions

- **Polynomial basis functions:**

$$\phi_j(x) = x^j$$

Some Common Basis Functions

- **Polynomial basis functions:**

$$\phi_j(x) = x^j$$

- **Gaussian basis functions:**

$$\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$$

Some Common Basis Functions

- **Polynomial basis functions:**

$$\phi_j(x) = x^j$$

- **Gaussian basis functions:**

$$\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$$

- **Sigmoidal basis functions:**

$$\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right)$$

Some Common Basis Functions

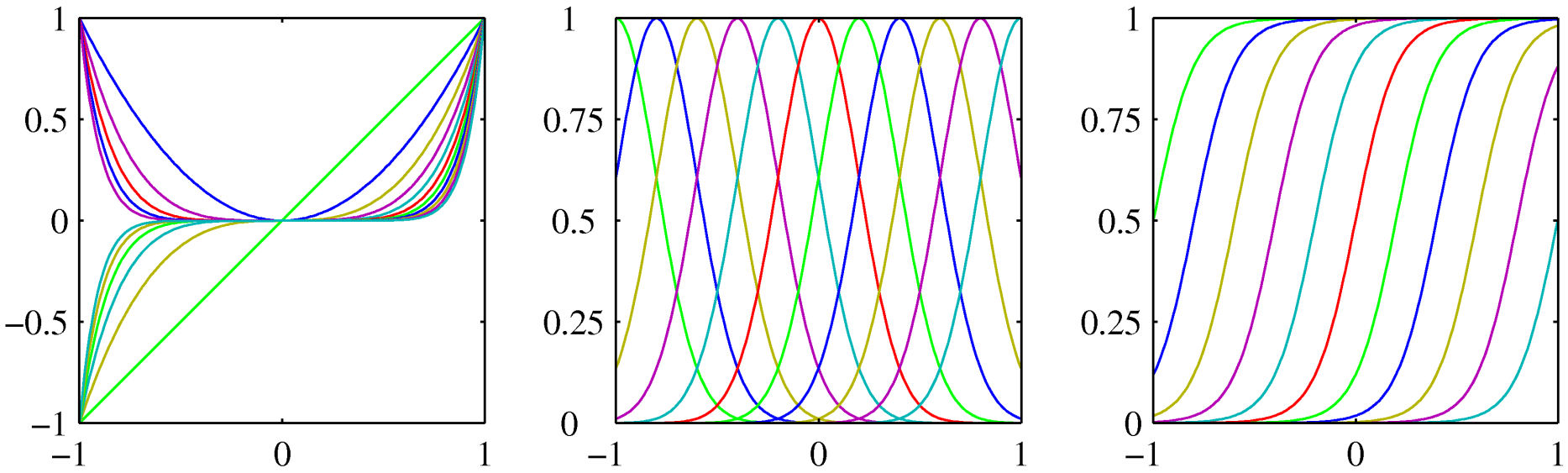

Figure: Polynomial, Gaussian, and Sigmoid basis functions

Some Common Basis Functions

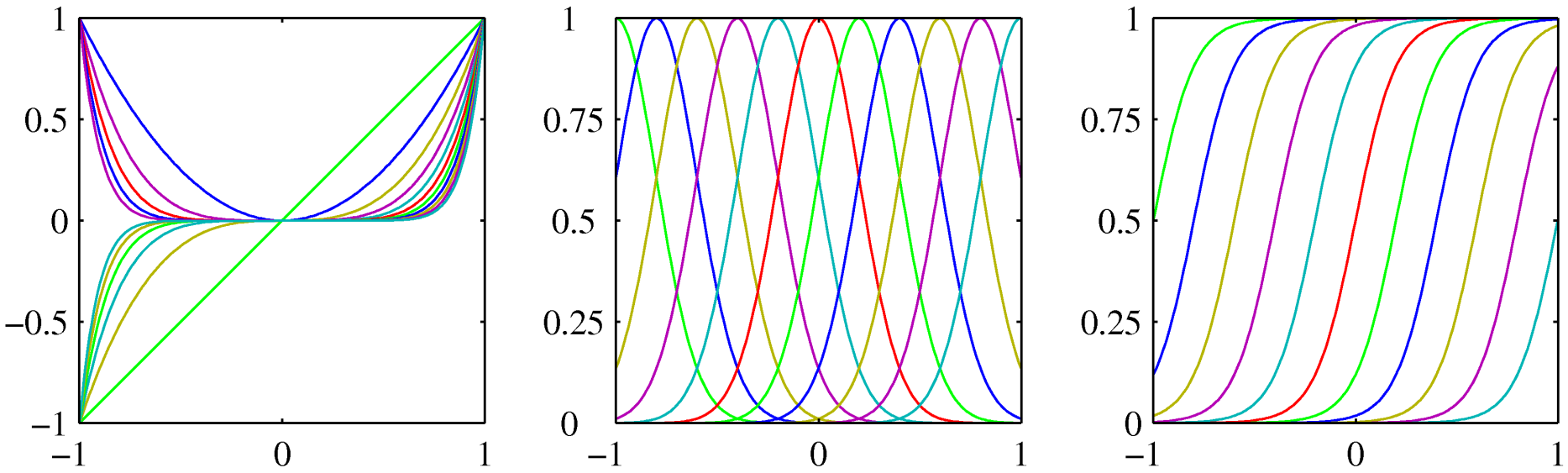

Figure: Polynomial, Gaussian, and Sigmoid basis functions

- These transformations manually re-map the inputs into spaces where a linear boundary might work.

Where Fixed Basis Functions Break Down

- Why not simply use this approach for all machine learning problems?

Where Fixed Basis Functions Break Down

- Why not simply use this approach for all machine learning problems?
- The issue lies in the **curse of dimensionality**.

Where Fixed Basis Functions Break Down

- Why not simply use this approach for all machine learning problems?
- The issue lies in the **curse of dimensionality**.
- If input space $\mathbf{x}$ has high dimensionality D, the number of required basis functions M grows **exponentially** with D.

Where Fixed Basis Functions Break Down

- Place 10 basis functions along each dimension:

Where Fixed Basis Functions Break Down

- Place 10 basis functions along each dimension:
  - For a 3D input space: $10^3 = 1,000$ basis functions.

Where Fixed Basis Functions Break Down

- Place 10 basis functions along each dimension:
  - For a 3D input space: $10^3 = 1,000$ basis functions.
  - For a 100D input space (e.g., small 10×10 image): 10^{100} basis functions required!

Where Fixed Basis Functions Break Down

- Place 10 basis functions along each dimension:
  - For a 3D input space: $10^3 = 1,000$ basis functions.
  - For a 100D input space (e.g., small 10×10 image): 10^{100} basis functions required!
- 10^{100} vastly exceeds the number of atoms in the observable universe.

Where Fixed Basis Functions Break Down

- Place 10 basis functions along each dimension:
  - For a 3D input space: $10^3 = 1,000$ basis functions.
  - For a 100D input space (e.g., small 10×10 image): 10^{100} basis functions required!
- 10^{100} vastly exceeds the number of atoms in the observable universe.
- **Conclusion:** Manually designing or pre-allocating fixed basis functions becomes completely intractable for high-dimensional data.

Fixed vs. Adaptive Basis Functions

- Neural networks resolve this limitation by making the basis functions **adaptive**.

Fixed vs. Adaptive Basis Functions

- Neural networks resolve this limitation by making the basis functions **adaptive**.
- Parameters defining the basis functions are learned simultaneously with output weights.

Fixed vs. Adaptive Basis Functions

- Neural networks resolve this limitation by making the basis functions **adaptive**.
- Parameters defining the basis functions are learned simultaneously with output weights.
- Each basis function $\phi_j(\mathbf{x})$ is parameterised by its own internal weights and biases:

$$\phi_j(\mathbf{x}) = \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

Fixed vs. Adaptive Basis Functions

- Neural networks resolve this limitation by making the basis functions **adaptive**.
- Parameters defining the basis functions are learned simultaneously with output weights.
- Each basis function $\phi_j(\mathbf{x})$ is parameterised by its own internal weights and biases:

$$\phi_j(\mathbf{x}) = \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

- The data determines the most optimal features to extract, bypassing uniform space-tiling.

Historical Context: The Birth of Artificial Neurons

- **1943:** Warren McCulloch and Walter Pitts introduce a highly simplified mathematical abstraction of a biological brain cell.

Historical Context: The Birth of Artificial Neurons

- **1943:** Warren McCulloch and Walter Pitts introduce a highly simplified mathematical abstraction of a biological brain cell.
- **Biological Inspiration:** Receives electrochemical signals through dendrites $\rightarrow$ if total signal exceeds threshold $\rightarrow$ fires electrical impulse down axon.

The McCulloch-Pitts Neuron Model

- Takes binary inputs $x_i \in \{0, 1\}$, sums them, and checks if the sum exceeds threshold θ:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{D} x_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{D} x_i < \theta \end{cases}$$

The McCulloch-Pitts Neuron Model

- Takes binary inputs $x_i \in \{0, 1\}$, sums them, and checks if the sum exceeds threshold θ:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{D} x_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{D} x_i < \theta \end{cases}$$

Limitation

This early model lacked a mechanism for learning; thresholds and connections had to be designed by hand.

Rosenblatt's Perceptron

- **1958:** Frank Rosenblatt extends the concept by introducing adjustable weights.

Rosenblatt's Perceptron

- **1958:** Frank Rosenblatt extends the concept by introducing adjustable weights.
- Forms a linear combination of real-valued inputs, adds a bias, and passes the result through a rigid step activation function:

$$y = f(\mathbf{w}^T \mathbf{x} + b)$$

Rosenblatt's Perceptron

- **1958:** Frank Rosenblatt extends the concept by introducing adjustable weights.
- Forms a linear combination of real-valued inputs, adds a bias, and passes the result through a rigid step activation function:

$$y = f(\mathbf{w}^T \mathbf{x} + b)$$

- Where:

$$f(a) = \begin{cases} +1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$$

Perceptron Learning Algorithm

- Rosenblatt developed an iterative learning algorithm to automatically adjust weights $\mathbf{w}$ based on errors.

Perceptron Learning Algorithm

- Rosenblatt developed an iterative learning algorithm to automatically adjust weights $\mathbf{w}$ based on errors.
- If a training example is misclassified, the weights are adjusted in the direction of the error.

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.
- Classic example: The **Exclusive OR (XOR)** logical function.
  - $\text{XOR}(0, 0) = 0$

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.
- Classic example: The **Exclusive OR (XOR)** logical function.
  - $\text{XOR}(0, 0) = 0$
  - $\text{XOR}(1, 1) = 0$

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.
- Classic example: The **Exclusive OR (XOR)** logical function.
  - $\text{XOR}(0, 0) = 0$
  - $\text{XOR}(1, 1) = 0$
  - $\text{XOR}(1, 0) = 1$

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.
- Classic example: The **Exclusive OR (XOR)** logical function.
  - $\text{XOR}(0, 0) = 0$
  - $\text{XOR}(1, 1) = 0$
  - $\text{XOR}(1, 0) = 1$
  - $\text{XOR}(0, 1) = 1$

Minsky & Papert's Critique

- **1969:** Marvin Minsky and Seymour Papert publish a rigorous mathematical critique of the Perceptron.
- **The Proof:** A single perceptron is fundamentally a linear classifier; it is entirely incapable of solving problems where classes are not linearly separable.
- Classic example: The **Exclusive OR (XOR)** logical function.
  - $\text{XOR}(0, 0) = 0$
  - $\text{XOR}(1, 1) = 0$
  - $\text{XOR}(1, 0) = 1$
  - $\text{XOR}(0, 1) = 1$

Minsky & Papert's Critique

- No straight line can separate zeroes from ones in a 2D plane.

Minsky & Papert's Critique

- No straight line can separate zeroes from ones in a 2D plane.
- Minsky and Papert noted that adding hidden layers could solve this, but argued there was no computationally efficient way to train them.

Minsky & Papert's Critique

- No straight line can separate zeroes from ones in a 2D plane.
- Minsky and Papert noted that adding hidden layers could solve this, but argued there was no computationally efficient way to train them.
- Resulted in the first **"Artificial Intelligence Winter"**.

Transition to Continuous Activation Functions

- The field was revitalised by replacing rigid, discontinuous step functions with **continuous, differentiable activation functions**.

Transition to Continuous Activation Functions

- The field was revitalised by replacing rigid, discontinuous step functions with **continuous, differentiable activation functions**.
- Step functions have a derivative of zero everywhere (except at the undefined boundary), offering no direction for weight updates.

Transition to Continuous Activation Functions

- The field was revitalised by replacing rigid, discontinuous step functions with **continuous, differentiable activation functions**.
- Step functions have a derivative of zero everywhere (except at the undefined boundary), offering no direction for weight updates.
- Smooth functions like the logistic sigmoid:

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

- Make the model fully differentiable $\rightarrow \frac{\partial y}{\partial \mathbf{w}}$ can be computed via the chain rule. This enables backpropagation.

Feedforward Neural Network or Multi-Layer Perceptron

- Multiple layers of computational units
- Information flows strictly in one direction:

$$\text{Input Layer} \rightarrow \text{Hidden Layers} \rightarrow \text{Output Layer}$$

Feedforward Neural Network or Multi-Layer Perceptron

- Multiple layers of computational units
- Information flows strictly in one direction:

$$\text{Input Layer} \rightarrow \text{Hidden Layers} \rightarrow \text{Output Layer}$$

- **Feedforward vs. Recurrent:** In a feedforward network, there are no feedback loops.

Feedforward Neural Network or Multi-Layer Perceptron

- Multiple layers of computational units
- Information flows strictly in one direction:

$$\text{Input Layer} \rightarrow \text{Hidden Layers} \rightarrow \text{Output Layer}$$

- **Feedforward vs. Recurrent:** In a feedforward network, there are no feedback loops.
- The output of any given layer cannot loop back to influence itself or any preceding layer.

Directed Acyclic Graphs (DAGs)

- Mathematically, a feedforward network can be formalized as a DAG:
  - **Graph:** A collection of nodes (neurons) connected by edges (weights).

Directed Acyclic Graphs (DAGs)

- Mathematically, a feedforward network can be formalized as a DAG:
  - **Graph:** A collection of nodes (neurons) connected by edges (weights).
  - **Directed:** Each edge has a specific direction, indicating that node i passes its output value to node j.

Directed Acyclic Graphs (DAGs)

- Mathematically, a feedforward network can be formalized as a DAG:
  - **Graph:** A collection of nodes (neurons) connected by edges (weights).
  - **Directed:** Each edge has a specific direction, indicating that node i passes its output value to node j.
  - **Acyclic:** There are no paths or closed loops that allow an information signal to travel from a node and eventually return to it.

Directed Acyclic Graphs (DAGs)

- Mathematically, a feedforward network can be formalized as a DAG:
  - **Graph:** A collection of nodes (neurons) connected by edges (weights).
  - **Directed:** Each edge has a specific direction, indicating that node i passes its output value to node j.
  - **Acyclic:** There are no paths or closed loops that allow an information signal to travel from a node and eventually return to it.
- Breaking computation down as a DAG is crucial for modern automatic differentiation libraries.

MLP Layer Taxonomy

1. **Input Layer:** No mathematical transformations. Represents the raw environment feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$.

MLP Layer Taxonomy

1. **Input Layer:** No mathematical transformations. Represents the raw environment feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$.
2. **Hidden Layers:** Not directly observed in the training dataset. Extract increasingly abstract features from representations provided by previous layers.

MLP Layer Taxonomy

1. **Input Layer:** No mathematical transformations. Represents the raw environment feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$.
2. **Hidden Layers:** Not directly observed in the training dataset. Extract increasingly abstract features from representations provided by previous layers.
3. **Output Layer:** Transforms the final hidden representations into predictions $\hat{\mathbf{y}}$. Structure is dictated by the specific ML task.

Multi-Layer Perceptron (MLP)

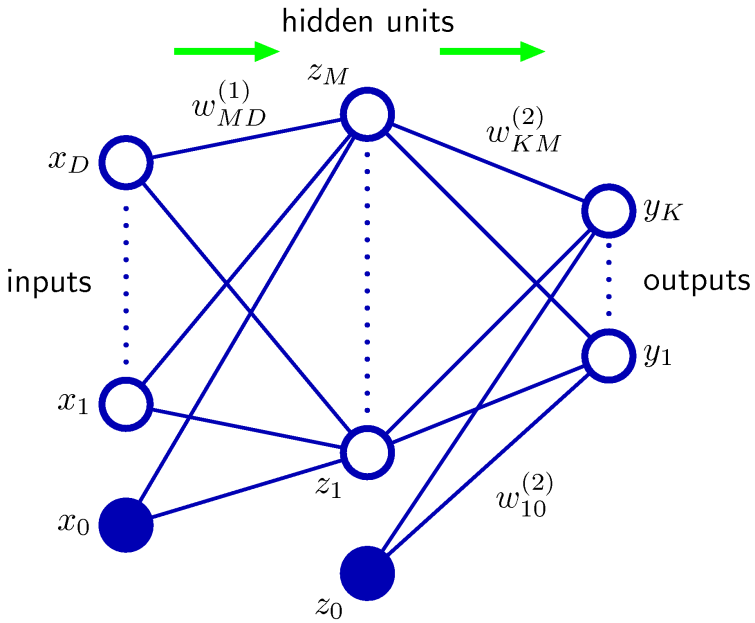

Figure: Network diagram for the classic two-layer MLP (Source: Bishop PRML)

The Single-Neuron Functional Unit

- The individual processing node (artificial neuron) performs two successive mathematical transformations on its inputs:
  1. A linear combination

The Single-Neuron Functional Unit

- The individual processing node (artificial neuron) performs two successive mathematical transformations on its inputs:
  1. A linear combination
  2. A non-linear activation

The Linear Combination Step

- Let a neuron receive an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$:

The Linear Combination Step

- Let a neuron receive an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$:
- The unit computes a weighted sum and adds a scalar bias b to produce the **pre-activation** a:

$$a = \sum_{i=1}^{D} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

The Linear Combination Step

- Let a neuron receive an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$:
- The unit computes a weighted sum and adds a scalar bias b to produce the **pre-activation** a:

$$a = \sum_{i=1}^{D} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

- Where:
  1. $\mathbf{w}$ determines the orientation of the neuron's decision plane.

The Linear Combination Step

- Let a neuron receive an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$:
- The unit computes a weighted sum and adds a scalar bias b to produce the **pre-activation** a:

$$a = \sum_{i=1}^{D} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

- Where:
  1. $\mathbf{w}$ determines the orientation of the neuron's decision plane.
  2. b shifts the activation threshold away from the origin.

The Activation Function Step

- To convert pre-activation a into an adaptive feature, it is passed through a non-linear activation function $\sigma(\cdot)$ to yield post-activation value z:

$$z = \sigma(a) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

The Activation Function Step

- To convert pre-activation a into an adaptive feature, it is passed through a non-linear activation function $\sigma(\cdot)$ to yield post-activation value z:

$$z = \sigma(a) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

- Must be continuous and differentiable to permit gradient-based learning.

Various Activation Functions: The Sigmoid

- The logistic sigmoid function squashes input into a bounded range between 0 and 1:

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

Various Activation Functions: The Sigmoid

- The logistic sigmoid function squashes input into a bounded range between 0 and 1:

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

- **Mathematical Property:** Compact derivative:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a))$$

Various Activation Functions: The Sigmoid

- The logistic sigmoid function squashes input into a bounded range between 0 and 1:

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

- **Mathematical Property:** Compact derivative:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a))$$

- **Limitation (Saturation):** When a is very large or very small, the curve flattens. The gradient $\sigma'(a)$ approaches zero, leading to the **vanishing gradient problem**.

Various Activation Functions: Hyperbolic Tangent

- The tanh function is a scaled version of the sigmoid, mapping inputs into a range between -1 and 1:

$$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} = 2\sigma(2a) - 1$$

Various Activation Functions: Hyperbolic Tangent

- The tanh function is a scaled version of the sigmoid, mapping inputs into a range between -1 and 1:

$$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} = 2\sigma(2a) - 1$$

- **Advantage:** It is **zero-centered**. Outputs have an expected value close to zero when inputs are symmetrically distributed, stabilizing optimization.

Various Activation Functions: Hyperbolic Tangent

- The tanh function is a scaled version of the sigmoid, mapping inputs into a range between -1 and 1:

$$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} = 2\sigma(2a) - 1$$

- **Advantage:** It is **zero-centered**. Outputs have an expected value close to zero when inputs are symmetrically distributed, stabilizing optimization.
- **Limitation:** It still suffers from saturation at extreme values of a.

Various Activation Functions: ReLU

- The Rectified Linear Unit (ReLU) is defined as a simple piecewise linear function:

$$f(a) = \max(0, a)$$

Various Activation Functions: ReLU

- The Rectified Linear Unit (ReLU) is defined as a simple piecewise linear function:

$$f(a) = \max(0, a)$$

- **Advantage:** For any positive pre-activation ($a > 0$), the derivative is exactly 1 ($f'(a) = 1$). This completely eliminates vanishing gradients for positive paths.

Various Activation Functions: ReLU

- The Rectified Linear Unit (ReLU) is defined as a simple piecewise linear function:

$$f(a) = \max(0, a)$$

- **Advantage:** For any positive pre-activation ($a > 0$), the derivative is exactly 1 ($f'(a) = 1$). This completely eliminates vanishing gradients for positive paths.
- Exceptionally cheap to compute.

Various Activation Functions: ReLU

- The Rectified Linear Unit (ReLU) is defined as a simple piecewise linear function:

$$f(a) = \max(0, a)$$

- **Advantage:** For any positive pre-activation ($a > 0$), the derivative is exactly 1 ($f'(a) = 1$). This completely eliminates vanishing gradients for positive paths.
- Exceptionally cheap to compute.
- **Limitation (Dying ReLU):** For $a < 0$, output and derivative are exactly zero. If a neuron always receives negative inputs, its gradient stays zero forever $\rightarrow$ "dead neuron".

Various Activation Functions: Leaky ReLU and GeLU

- Variants to mitigate the dying ReLU problem:
  1. **Leaky ReLU:** Introduces a small constant slope $\alpha \approx 0.01$ when $a < 0$:

$$f(a) = \max(\alpha a, a)$$

Various Activation Functions: Leaky ReLU and GeLU

- Variants to mitigate the dying ReLU problem:
  1. **Leaky ReLU:** Introduces a small constant slope $\alpha \approx 0.01$ when $a < 0$:

$$f(a) = \max(\alpha a, a)$$

  2. **Gaussian Error Linear Unit (GeLU):** Used in transformers and deep MLPs. Weights input by standard normal CDF:

$$f(a) = a \cdot \Phi(a)$$

Various Activation Functions: Leaky ReLU and GeLU

- Variants to mitigate the dying ReLU problem:
  1. **Leaky ReLU:** Introduces a small constant slope $\alpha \approx 0.01$ when $a < 0$:

$$f(a) = \max(\alpha a, a)$$

  2. **Gaussian Error Linear Unit (GeLU):** Used in transformers and deep MLPs. Weights input by standard normal CDF:

$$f(a) = a \cdot \Phi(a)$$

- Create a smooth approximation of ReLU preserving slight curvature for negative values.

Various Activation Functions

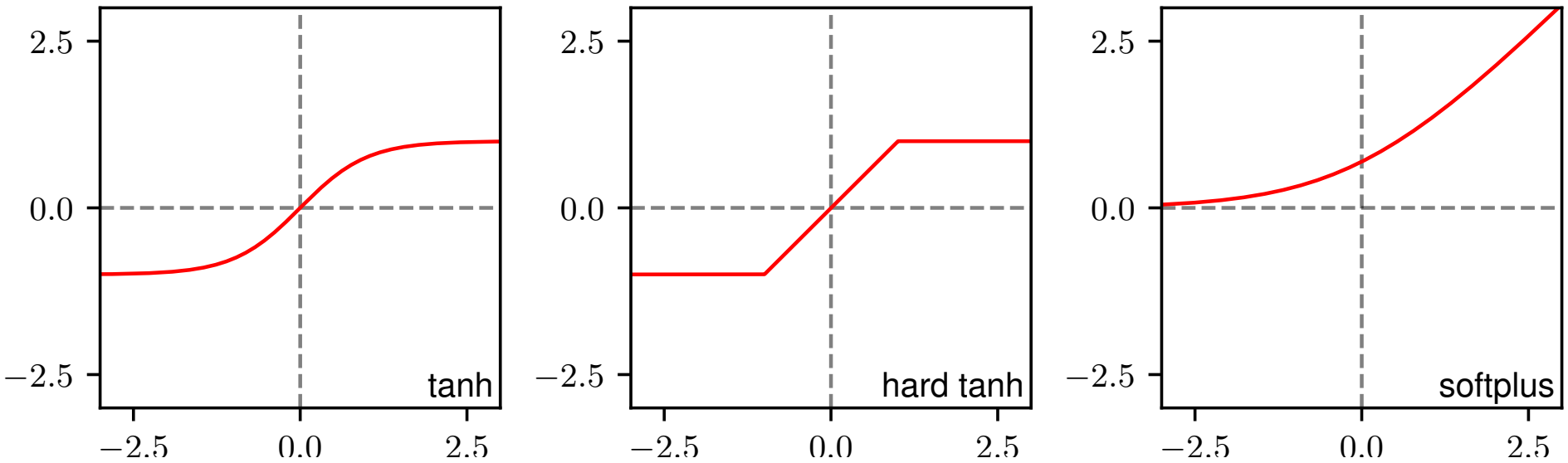

Figure: Tanh, Hard tanh, and Softplus activation functions

Various Activation Functions

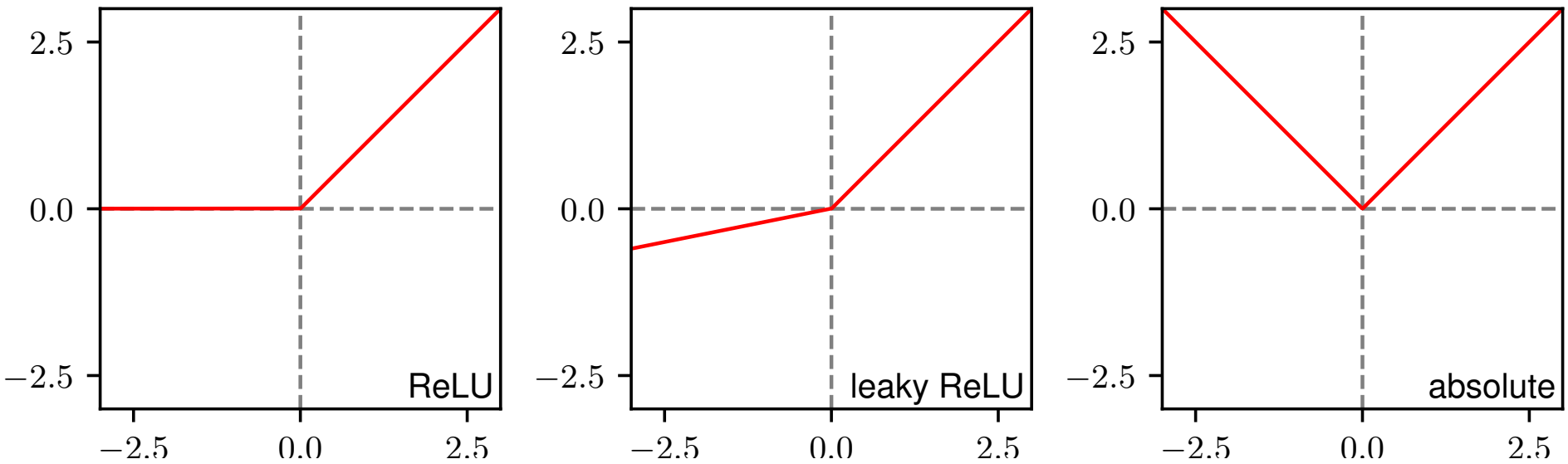

Figure: ReLU, Leaky ReLU, and Absolute activation functions

MLP: Layer Matrix Form

- To avoid scalar iterations, we aggregate individual neurons into a layer using linear algebra. For a single hidden layer (M units) and input $\mathbf{x} \in \mathbb{R}^D$:

MLP: Layer Matrix Form

- To avoid scalar iterations, we aggregate individual neurons into a layer using linear algebra. For a single hidden layer (M units) and input $\mathbf{x} \in \mathbb{R}^D$:
  - Collect weights into matrix $\mathbf{W}^{(1)}$ of size $M \times D$.

MLP: Layer Matrix Form

- To avoid scalar iterations, we aggregate individual neurons into a layer using linear algebra. For a single hidden layer (M units) and input $\mathbf{x} \in \mathbb{R}^D$:
  - Collect weights into matrix $\mathbf{W}^{(1)}$ of size $M \times D$.
  - Collect biases into vector $\mathbf{b}^{(1)} \in \mathbb{R}^M$.

MLP: Layer Matrix Form

- To avoid scalar iterations, we aggregate individual neurons into a layer using linear algebra. For a single hidden layer (M units) and input $\mathbf{x} \in \mathbb{R}^D$:
  - Collect weights into matrix $\mathbf{W}^{(1)}$ of size $M \times D$.
  - Collect biases into vector $\mathbf{b}^{(1)} \in \mathbb{R}^M$.

MLP: Layer Matrix Form

- To avoid scalar iterations, we aggregate individual neurons into a layer using linear algebra. For a single hidden layer (M units) and input $\mathbf{x} \in \mathbb{R}^D$:
  - Collect weights into matrix $\mathbf{W}^{(1)}$ of size $M \times D$.
  - Collect biases into vector $\mathbf{b}^{(1)} \in \mathbb{R}^M$.
- The vector of pre-activations $\mathbf{a}^{(1)} \in \mathbb{R}^M$ is:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

MLP: Forward Propagation Equations

- Applying element-wise non-linear activation yields hidden output vector $\mathbf{z} \in \mathbb{R}^M$:

$$\mathbf{z} = \sigma\left(\mathbf{a}^{(1)}\right)$$

MLP: Forward Propagation Equations

- Applying element-wise non-linear activation yields hidden output vector $\mathbf{z} \in \mathbb{R}^M$:

$$\mathbf{z} = \sigma\left(\mathbf{a}^{(1)}\right)$$

- This hidden representation is passed to the output layer (K units) with weight matrix $\mathbf{W}^{(2)}$ $(K \times M)$ and bias vector $\mathbf{b}^{(2)} \in \mathbb{R}^K$:

$$\mathbf{a}^{(2)} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$

MLP: Forward Propagation Equations

- Applying element-wise non-linear activation yields hidden output vector $\mathbf{z} \in \mathbb{R}^M$:

$$\mathbf{z} = \sigma\left(\mathbf{a}^{(1)}\right)$$

- This hidden representation is passed to the output layer (K units) with weight matrix $\mathbf{W}^{(2)}$ $(K \times M)$ and bias vector $\mathbf{b}^{(2)} \in \mathbb{R}^K$:

$$\mathbf{a}^{(2)} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$

- Final raw predictions $\hat{\mathbf{y}} \in \mathbb{R}^K$ are:

$$\hat{\mathbf{y}} = f\left(\mathbf{a}^{(2)}\right)$$

Generalisation to L Layers

- For any intermediate hidden layer $l \in \{1, 2, \ldots, L\}$, the operations scale recursively:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$$

Generalisation to L Layers

- For any intermediate hidden layer $l \in \{1, 2, \ldots, L\}$, the operations scale recursively:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(\mathbf{a}^{(l)}\right)$$

Generalisation to L Layers

- For any intermediate hidden layer $l \in \{1, 2, \ldots, L\}$, the operations scale recursively:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(\mathbf{a}^{(l)}\right)$$

- Where $\mathbf{z}^{(0)} = \mathbf{x}$ and $\hat{\mathbf{y}} = \mathbf{z}^{(L)}$.

Generalisation to L Layers

- For any intermediate hidden layer $l \in \{1, 2, \ldots, L\}$, the operations scale recursively:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(\mathbf{a}^{(l)}\right)$$

- Where $\mathbf{z}^{(0)} = \mathbf{x}$ and $\hat{\mathbf{y}} = \mathbf{z}^{(L)}$.
- Written completely as a nested functional composition:

$$f(\mathbf{x}) = f^{(L)}\left(\sigma^{(L-1)}\left(\mathbf{W}^{(L-1)}\ldots\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)\cdots + \mathbf{b}^{(L-1)}\right)\right)$$

MLP Output Layer

- The choice of final layer activation function $f\left(\mathbf{a}^{(L)}\right)$ is dictated entirely by the statistical nature of target variables.

MLP Output Layer

- The choice of final layer activation function $f\left(\mathbf{a}^{(L)}\right)$ is dictated entirely by the statistical nature of target variables.
- For **continuous target variables**, use linear activation.

MLP Output Layer

- The choice of final layer activation function $f\left(\mathbf{a}^{(L)}\right)$ is dictated entirely by the statistical nature of target variables.
- For **continuous target variables**, use linear activation.
- For **binary classification tasks**, use sigmoid activation.

MLP Output Layer

- The choice of final layer activation function $f\left(\mathbf{a}^{(L)}\right)$ is dictated entirely by the statistical nature of target variables.
- For **continuous target variables**, use linear activation.
- For **binary classification tasks**, use sigmoid activation.
- For **multi-class classification tasks**, use softmax activation.

MLP Output Layer

- For continuous, unconstrained target values $\mathbf{t} \in \mathbb{R}^K$:

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)}$$

MLP Output Layer

- For continuous, unconstrained target values $\mathbf{t} \in \mathbb{R}^K$:

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)}$$

- For binary classification target $t \in \{0, 1\}$, the network outputs a single scalar conditional probability $p(t = 1 \mid \mathbf{x})$:

$$\hat{y} = \sigma\left(a^{(L)}\right) = \frac{1}{1 + e^{-a^{(L)}}}$$

MLP Output Layer

- For continuous, unconstrained target values $\mathbf{t} \in \mathbb{R}^K$:

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)}$$

- For binary classification target $t \in \{0, 1\}$, the network outputs a single scalar conditional probability $p(t = 1 \mid \mathbf{x})$:

$$\hat{y} = \sigma\left(a^{(L)}\right) = \frac{1}{1 + e^{-a^{(L)}}}$$

MLP Output Layer

- When mutually exclusive classification involves $K > 2$ distinct classes, target is a one-hot encoded vector $\mathbf{t} \in \{0, 1\}^K$.

MLP Output Layer

- When mutually exclusive classification involves $K > 2$ distinct classes, target is a one-hot encoded vector $\mathbf{t} \in \{0, 1\}^K$.
- Apply the **softmax function** to output pre-activations:

$$\hat{y}_k = \frac{e^{a_k^{(L)}}}{\sum_{j=1}^{K} e^{a_j^{(L)}}}$$

MLP Output Layer

- When mutually exclusive classification involves $K > 2$ distinct classes, target is a one-hot encoded vector $\mathbf{t} \in \{0, 1\}^K$.
- Apply the **softmax function** to output pre-activations:

$$\hat{y}_k = \frac{e^{a_k^{(L)}}}{\sum_{j=1}^{K} e^{a_j^{(L)}}}$$

  - Exponentiates unconstrained real values ("logits") to ensure positivity.

MLP Output Layer

- When mutually exclusive classification involves $K > 2$ distinct classes, target is a one-hot encoded vector $\mathbf{t} \in \{0, 1\}^K$.
- Apply the **softmax function** to output pre-activations:

$$\hat{y}_k = \frac{e^{a_k^{(L)}}}{\sum_{j=1}^{K} e^{a_j^{(L)}}}$$

  - Exponentiates unconstrained real values ("logits") to ensure positivity.
  - Normalises by dividing by the sum of all exponentiated values.

Probabilistic Interpretation of Softmax Outputs

- Maps directly to a Multinomial (Categorical) distribution over classes:

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} \hat{y}_k^{t_k}$$

Probabilistic Interpretation of Softmax Outputs

- Maps directly to a Multinomial (Categorical) distribution over classes:

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} \hat{y}_k^{t_k}$$

- Given target class j where $t_j = 1$ and $t_{k \neq j} = 0$:

$$\prod_{k=1}^{K} \hat{y}_k^{t_k} = \hat{y}_j^1 \times \prod_{k \neq j}^{K} \hat{y}_k^0 = \hat{y}_j \times 1 \times \cdots \times 1 = \hat{y}_j$$

Probabilistic Interpretation of Softmax Outputs

- Maps directly to a Multinomial (Categorical) distribution over classes:

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} \hat{y}_k^{t_k}$$

- Given target class j where $t_j = 1$ and $t_{k \neq j} = 0$:

$$\prod_{k=1}^{K} \hat{y}_k^{t_k} = \hat{y}_j^1 \times \prod_{k \neq j}^{K} \hat{y}_k^0 = \hat{y}_j \times 1 \times \cdots \times 1 = \hat{y}_j$$

- The probability of the observed class is exactly the model's predicted probability for that label.

The Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

A feedforward neural network with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on a compact subset of $\mathbb{R}^D$ to any desired degree of accuracy, provided the activation function is continuous, bounded, and non-linear.

The Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

A feedforward neural network with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on a compact subset of $\mathbb{R}^D$ to any desired degree of accuracy, provided the activation function is continuous, bounded, and non-linear.

- Guarantees that a shallow network can theoretically represent any smooth mapping.

The Curse of Dimensionality for MLPs

- If a single hidden layer is theoretically sufficient, why use deep networks?

The Curse of Dimensionality for MLPs

- If a single hidden layer is theoretically sufficient, why use deep networks?
  - **Existence Proof Only:** The theorem guarantees that a network *exists*, but does not specify size or ensure gradient descent can find it.

The Curse of Dimensionality for MLPs

- If a single hidden layer is theoretically sufficient, why use deep networks?
  - **Existence Proof Only:** The theorem guarantees that a network *exists*, but does not specify size or ensure gradient descent can find it.
  - **Exponential Width:** Approximating complex, highly oscillating functions can require an exponentially large number of hidden units ($M \to \infty$).

The Curse of Dimensionality for MLPs

- If a single hidden layer is theoretically sufficient, why use deep networks?
  - **Existence Proof Only:** The theorem guarantees that a network *exists*, but does not specify size or ensure gradient descent can find it.
  - **Exponential Width:** Approximating complex, highly oscillating functions can require an exponentially large number of hidden units ($M \to \infty$).
  - **Hierarchical Feature Extraction:** Depth allows composition. Early layers extract simple edges; later layers compose them into shapes, then complex objects. Deep networks use far fewer parameters than shallow counterparts.

Error (Loss) Functions

- Let a dataset consist of N independent and identically distributed (*i.i.d.*) observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with target labels $\mathbf{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_N\}$.

Error (Loss) Functions

- Let a dataset consist of N independent and identically distributed (*i.i.d.*) observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with target labels $\mathbf{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_N\}$.
- We use different loss functions to compute the error in the network's predictions depending on the task.

Error (Loss) Functions

- For regression, we use the sum of squares error function:

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

Error (Loss) Functions

- For regression, we use the sum of squares error function:

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

- For binary classification where $t_n \in \{0, 1\}$ and $\hat{y}_n = \sigma(a_n)$, minimizing the negative log-likelihood results in:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} [t_n \ln \hat{y}_n + (1 - t_n) \ln(1 - \hat{y}_n)]$$

Error (Loss) Functions

- For multi-class classification with K mutually exclusive classes (one-hot target $\mathbf{t}_n$, softmax output $\hat{\mathbf{y}}_n$):

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln \hat{y}_{nk}$$

Geometry of the Error Surface

- An error surface spans a high-dimensional space of W parameters. Unlike linear models (convex paraboloids), MLPs yield non-convex landscapes with:

Geometry of the Error Surface

- An error surface spans a high-dimensional space of W parameters. Unlike linear models (convex paraboloids), MLPs yield non-convex landscapes with:
  - **Global Minima:** Absolute lowest possible error value.

Geometry of the Error Surface

- An error surface spans a high-dimensional space of W parameters. Unlike linear models (convex paraboloids), MLPs yield non-convex landscapes with:
  - **Global Minima:** Absolute lowest possible error value.
  - **Local Minima:** Error is lower than in the immediate neighborhood, but higher than the global minimum.

Geometry of the Error Surface

- An error surface spans a high-dimensional space of W parameters. Unlike linear models (convex paraboloids), MLPs yield non-convex landscapes with:
  - **Global Minima:** Absolute lowest possible error value.
  - **Local Minima:** Error is lower than in the immediate neighborhood, but higher than the global minimum.
  - **Saddle Points:** Surface slopes upward along some parameter dimensions but downward along others.

Geometry of the Error Surface

- Local geometry is analyzed using the eigenvalues of the Hessian matrix H od the error function $E\mathbf{w}$:

Geometry of the Error Surface

- Local geometry is analyzed using the eigenvalues of the Hessian matrix H od the error function $E\mathbf{w}$:
  - If all eigenvalues of H are strictly positive $\rightarrow$ **Local Minimum**.

Geometry of the Error Surface

- Local geometry is analyzed using the eigenvalues of the Hessian matrix H od the error function $E\mathbf{w}$:
  - If all eigenvalues of H are strictly positive $\rightarrow$ **Local Minimum**.
  - If all eigenvalues of H are strictly negative $\rightarrow$ **Local Maximum**.

Geometry of the Error Surface

- Local geometry is analyzed using the eigenvalues of the Hessian matrix H od the error function $E\mathbf{w}$:
  - If all eigenvalues of H are strictly positive $\rightarrow$ **Local Minimum**.
  - If all eigenvalues of H are strictly negative $\rightarrow$ **Local Maximum**.
  - If there is a mix of positive and negative eigenvalues $\rightarrow$ **Saddle Point**.

Visualizing the Loss Surface

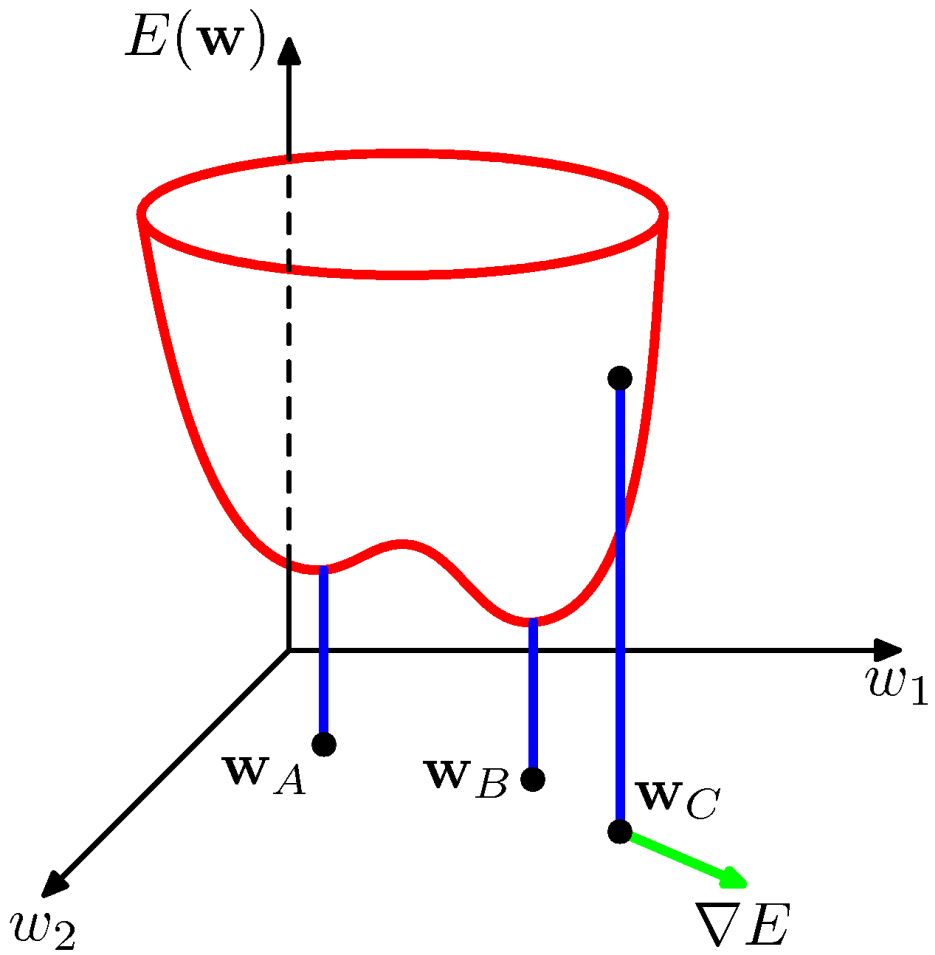

Figure: Geometrical view of the error function E(w) as a surface sitting over weight space. (Source: Bishop PRML)

High-Dimensionality Reality

- Early literature focused heavily on the risk of getting trapped in poor local minima.

High-Dimensionality Reality

- Early literature focused heavily on the risk of getting trapped in poor local minima.
- In modern networks where W spans millions of dimensions, a true local minimum requires the slope to curve upward in *every single one* of the W dimensions simultaneously.

High-Dimensionality Reality

- Early literature focused heavily on the risk of getting trapped in poor local minima.
- In modern networks where W spans millions of dimensions, a true local minimum requires the slope to curve upward in *every single one* of the W dimensions simultaneously.
- The probability of this is exceptionally low.

High-Dimensionality Reality

- Early literature focused heavily on the risk of getting trapped in poor local minima.
- In modern networks where W spans millions of dimensions, a true local minimum requires the slope to curve upward in *every single one* of the W dimensions simultaneously.
- The probability of this is exceptionally low.
- **Takeaway:** The vast majority of critical points ($\nabla E = 0$) are **saddle points**. Optimisers must traverse these flat regions efficiently without stalling.

Gradient Descent Algorithms

- To find a parameter configuration that minimizes $E(\mathbf{w})$, we iterate using the gradient vector $\nabla E(\mathbf{w})$ for directions of steepest descent.

Gradient Descent Algorithms

- To find a parameter configuration that minimizes $E(\mathbf{w})$, we iterate using the gradient vector $\nabla E(\mathbf{w})$ for directions of steepest descent.

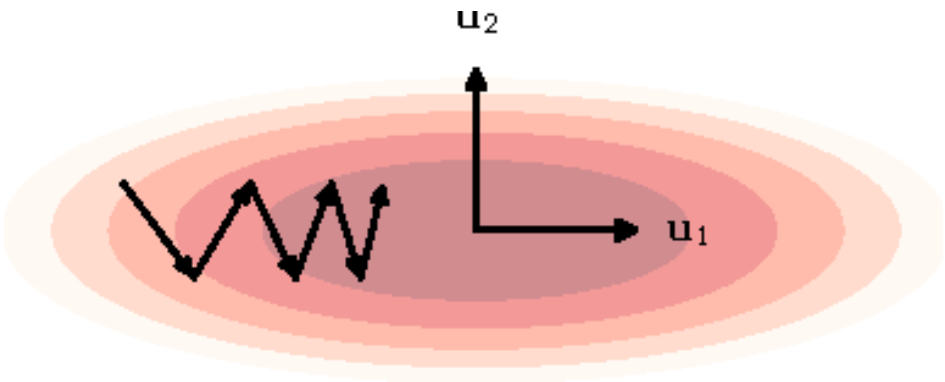

Figure: Schematic of fixed-step gradient descent on an error surface with highly different curvatures (a long valley shown by ellipses). The local negative gradient $-\delta E$ usually doesn't point to the minimum, so steps can oscillate across the valley and make slow progress toward the minimum. u1 and u2 are the Hessian eigenvectors. (Source: Bishop DLFC)

Batch Gradient Descent (BGD)

- Computes exact gradient of total error by evaluating every sample before an update:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) = \mathbf{w}^{(\tau)} - \eta \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$

Batch Gradient Descent (BGD)

- Computes exact gradient of total error by evaluating every sample before an update:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) = \mathbf{w}^{(\tau)} - \eta \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$

- Where $\eta > 0$ is the learning rate and τ is the iteration index.

Batch Gradient Descent (BGD)

- Computes exact gradient of total error by evaluating every sample before an update:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) = \mathbf{w}^{(\tau)} - \eta \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{(\tau)})$$

- Where $\eta > 0$ is the learning rate and τ is the iteration index.

Limitation

If the dataset contains millions of samples, tracking every sample for a single update is computationally slow and memory-intensive.

Stochastic Gradient Descent (SGD)

- Updates parameters immediately after processing a single, randomly selected example $\mathbf{x}_n$:

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

Stochastic Gradient Descent (SGD)

- Updates parameters immediately after processing a single, randomly selected example $\mathbf{x}_n$:

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

  - **Advantage:** Updates are quick; data can be processed on the fly.

Stochastic Gradient Descent (SGD)

- Updates parameters immediately after processing a single, randomly selected example $\mathbf{x}_n$:

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

  - **Advantage:** Updates are quick; data can be processed on the fly.
  - **Limitation:** Single sample gradient is noisy. The optimization path fluctuates significantly. This helps escape shallow saddle points, but causes continuous oscillation around the minimum.

Mini-batch Stochastic Gradient Descent

- Strikes a balance between BGD and SGD. Partitions dataset into subsets $\mathcal{B}$ containing B samples (e.g., $B = 32, 64, 256$):

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \frac{\eta}{B} \sum_{n \in \mathcal{B}} \nabla E_n(\mathbf{w}^{(\tau)})$$

Mini-batch Stochastic Gradient Descent

- Strikes a balance between BGD and SGD. Partitions dataset into subsets $\mathcal{B}$ containing B samples (e.g., $B = 32, 64, 256$):

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \frac{\eta}{B} \sum_{n \in \mathcal{B}} \nabla E_n(\mathbf{w}^{(\tau)})$$

- Reduces noise compared to pure SGD, while leveraging parallel hardware matrix processing.

The Backpropagation Goal

- To update parameters, we need partial derivatives for every single weight:

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

The Backpropagation Goal

- To update parameters, we need partial derivatives for every single weight:

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

- We must calculate exactly how a tiny change in a specific weight w_{ji} impacts total error E:

$$\frac{\partial E}{\partial w_{ji}}$$

The Backpropagation Goal

- To update parameters, we need partial derivatives for every single weight:
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

- We must calculate exactly how a tiny change in a specific weight w_{ji} impacts total error E:
$$\frac{\partial E}{\partial w_{ji}}$$

- Backpropagation calculates these derivatives efficiently by applying the **Chain Rule**.

The Chain Rule

Single-Variable Chain Rule

If y depends on u, and u depends on x:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$

The Chain Rule

Single-Variable Chain Rule

If y depends on u, and u depends on x:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$

Multi-Variable Chain Rule

If x influences E through multiple independent paths (a_1 and a_2):

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial a_1}\frac{\partial a_1}{\partial x} + \frac{\partial E}{\partial a_2}\frac{\partial a_2}{\partial x} = \sum_j \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial x}$$

Derivation: Single Connection Path

- Consider weight w_{ji} connecting output of node i (z_i) to node j in the next layer.

Derivation: Single Connection Path

- Consider weight w_{ji} connecting output of node i (z_i) to node j in the next layer.
- The sequence is:
  1. **Pre-activation:** $a_j = \sum_i w_{ji} z_i + b_j$

Derivation: Single Connection Path

- Consider weight w_{ji} connecting output of node i (z_i) to node j in the next layer.
- The sequence is:
  1. **Pre-activation:** $a_j = \sum_i w_{ji} z_i + b_j$
  2. **Activation function:** $z_j = \sigma(a_j)$

Derivation: Single Connection Path

- Consider weight w_{ji} connecting output of node i (z_i) to node j in the next layer.
- The sequence is:
  1. **Pre-activation:** $a_j = \sum_i w_{ji} z_i + b_j$
  2. **Activation function:** $z_j = \sigma(a_j)$
  3. **Forward flow:** Signals travel forward to final error E.

Derivation: Single Connection Path

- Consider weight w_{ji} connecting output of node i (z_i) to node j in the next layer.
- The sequence is:
  1. **Pre-activation:** $a_j = \sum_i w_{ji} z_i + b_j$
  2. **Activation function:** $z_j = \sigma(a_j)$
  3. **Forward flow:** Signals travel forward to final error E.
- Applying the chain rule:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}}$$

Derivation Part 1: Evaluating $\frac{\partial a_j}{\partial w_{ji}}$

- The term a_j is expanded as:

$$a_j = w_{j1}z_1 + w_{j2}z_2 + \cdots + w_{ji}z_i + \cdots + b_j$$

Derivation Part 1: Evaluating $\frac{\partial a_j}{\partial w_{ji}}$

- The term a_j is expanded as:

$$a_j = w_{j1}z_1 + w_{j2}z_2 + \cdots + w_{ji}z_i + \cdots + b_j$$

- Taking the derivative with respect to the specific weight w_{ji}, all other terms drop out:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

Derivation Part 1: Evaluating $\frac{\partial a_j}{\partial w_{ji}}$

- The term a_j is expanded as:

$$a_j = w_{j1}z_1 + w_{j2}z_2 + \cdots + w_{ji}z_i + \cdots + b_j$$

- Taking the derivative with respect to the specific weight w_{ji}, all other terms drop out:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

- The first part of our derivative is simply the incoming signal (z_i).

Derivation Part 2: The Local Error Signal (δ_j)

- The remaining part measures how a change in pre-activation a_j affects total error. We define this as the **local error signal** or **delta value**:

$$\delta_j \equiv \frac{\partial E}{\partial a_j}$$

Derivation Part 2: The Local Error Signal (δ_j)

- The remaining part measures how a change in pre-activation a_j affects total error. We define this as the **local error signal** or **delta value**:

$$\delta_j \equiv \frac{\partial E}{\partial a_j}$$

- Substituting both components back into the chain rule formula gives:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$$

Derivation Part 2: The Local Error Signal (δ_j)

- The remaining part measures how a change in pre-activation a_j affects total error. We define this as the **local error signal** or **delta value**:

$$\delta_j \equiv \frac{\partial E}{\partial a_j}$$

- Substituting both components back into the chain rule formula gives:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$$

"Derivative = Local Error at receiving node $\times$ Input Signal from sending node."

Derivative of Bias Parameters

- For bias parameter b_j, since:

$$\frac{\partial a_j}{\partial b_j} = 1$$

Derivative of Bias Parameters

- For bias parameter b_j, since:

$$\frac{\partial a_j}{\partial b_j} = 1$$

- Its derivative simplifies directly to the local error signal:

$$\frac{\partial E}{\partial b_j} = \delta_j$$

Computing Deltas Backward: Output Layer (L)

- For node k in final layer L, its pre-activation a_k directly influences output $\hat{y}_k$:

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial a_k}$$

Computing Deltas Backward: Output Layer (L)

- For node k in final layer L, its pre-activation a_k directly influences output $\hat{y}_k$:

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial a_k}$$

- Example with MSE loss ($E = \frac{1}{2}(\hat{y}_k - t_k)^2$) and linear activation ($\hat{y}_k = a_k$):

Computing Deltas Backward: Output Layer (L)

- For node k in final layer L, its pre-activation a_k directly influences output $\hat{y}_k$:

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial a_k}$$

- Example with MSE loss ($E = \frac{1}{2}(\hat{y}_k - t_k)^2$) and linear activation ($\hat{y}_k = a_k$):
  - $\frac{\partial E}{\partial \hat{y}_k} = (\hat{y}_k - t_k)$

Computing Deltas Backward: Output Layer (L)

- For node k in final layer L, its pre-activation a_k directly influences output $\hat{y}_k$:

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial a_k}$$

- Example with MSE loss ($E = \frac{1}{2}(\hat{y}_k - t_k)^2$) and linear activation ($\hat{y}_k = a_k$):
  - $\frac{\partial E}{\partial \hat{y}_k} = (\hat{y}_k - t_k)$
  - $\frac{\partial \hat{y}_k}{\partial a_k} = 1$

Computing Deltas Backward: Output Layer (L)

- For node k in final layer L, its pre-activation a_k directly influences output $\hat{y}_k$:

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial a_k}$$

- Example with MSE loss ($E = \frac{1}{2}(\hat{y}_k - t_k)^2$) and linear activation ($\hat{y}_k = a_k$):
  - $\frac{\partial E}{\partial \hat{y}_k} = (\hat{y}_k - t_k)$
  - $\frac{\partial \hat{y}_k}{\partial a_k} = 1$
- Multiplying gives the output delta:

$$\delta_k = (\hat{y}_k - t_k)$$

Computing Deltas Backward: Hidden Layers

- To track indices across consecutive layers $(l-1,\ l,\ l+1)$, let:
  1. i **(Layer $l-1$):** Sending node outputs signal z_i across weight w_{ji}.

Computing Deltas Backward: Hidden Layers

- To track indices across consecutive layers $(l-1,\ l,\ l+1)$, let:
  1. i **(Layer $l-1$):** Sending node outputs signal z_i across weight w_{ji}.
  2. j **(Layer l):** Middle neuron under evaluation. We need to calculate its δ_j.

Computing Deltas Backward: Hidden Layers

- To track indices across consecutive layers $(l-1,\ l,\ l+1)$, let:
  1. i **(Layer $l-1$):** Sending node outputs signal z_i across weight w_{ji}.
  2. j **(Layer l):** Middle neuron under evaluation. We need to calculate its δ_j.
  3. k **(Layer $l+1$):** Downstream receiving nodes. Node j impacts multiple receiving neurons k.

Computing Deltas Backward: Hidden Layers

- To track indices across consecutive layers $(l-1,\ l,\ l+1)$, let:
  1. i **(Layer $l-1$):** Sending node outputs signal z_i across weight w_{ji}.
  2. j **(Layer l):** Middle neuron under evaluation. We need to calculate its δ_j.
  3. k **(Layer $l+1$):** Downstream receiving nodes. Node j impacts multiple receiving neurons k.

$$\text{Layer } (l-1) \xrightarrow{w_{ji}} \text{Layer } (l) \xrightarrow{w_{kj}} \text{Layer } (l+1)$$

The Recurrence Relation

- Applying the multi-variable chain rule to sum over all receiving nodes k:

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial a_j}$$

The Recurrence Relation

- Applying the multi-variable chain rule to sum over all receiving nodes k:

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial a_j}$$

- Where:

$$a_k = w_{kj} z_j + \sum_{i \neq j} w_{ki} z_i + b_k \quad \text{and} \quad z_j = \sigma(a_j)$$

The Recurrence Relation

- Applying the multi-variable chain rule to sum over all receiving nodes k:

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial a_j}$$

- Where:

$$a_k = w_{kj} z_j + \sum_{i \neq j} w_{ki} z_i + b_k \quad \text{and} \quad z_j = \sigma(a_j)$$

- Breaking down $\frac{\partial a_k}{\partial a_j}$:

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial a_j} = w_{kj} \cdot \sigma'(a_j)$$

The Recurrence Relation

- Plugging $\delta_k = \frac{\partial E}{\partial a_k}$ and $\frac{\partial a_k}{\partial a_j} = w_{kj}\sigma'(a_j)$ back into the summation:

$$\delta_j = \sigma'(a_j) \sum_k w_{kj}\delta_k$$

The Recurrence Relation

- Plugging $\delta_k = \frac{\partial E}{\partial a_k}$ and $\frac{\partial a_k}{\partial a_j} = w_{kj}\sigma'(a_j)$ back into the summation:

$$\delta_j = \sigma'(a_j) \sum_k w_{kj}\delta_k$$

Intuition

To find hidden local error (δ_j), take all errors from the next layer (δ_k), multiply them by connecting weights (w_{kj}), sum them up, and scale by the slope of the current activation function ($\sigma'(a_j)$).

Network Training Step-by-Step

1. **Forward Pass:** Input $\mathbf{x}$, compute activations layer-by-layer $(\mathbf{a}^{(l)}, \mathbf{z}^{(l)})$. Keep intermediate values in memory.

Network Training Step-by-Step

1. **Forward Pass:** Input $\mathbf{x}$, compute activations layer-by-layer $(\mathbf{a}^{(l)}, \mathbf{z}^{(l)})$. Keep intermediate values in memory.
2. **Output Error Evaluation:** Compute initial signals $\delta_k = (\hat{y}_k - t_k)$ at the final layer.

Network Training Step-by-Step

1. **Forward Pass:** Input $\mathbf{x}$, compute activations layer-by-layer $(\mathbf{a}^{(l)}, \mathbf{z}^{(l)})$. Keep intermediate values in memory.
2. **Output Error Evaluation:** Compute initial signals $\delta_k = (\hat{y}_k - t_k)$ at the final layer.
3. **Backward Pass:** Propagate errors using $\delta_j = \sigma'(a_j) \sum w_{kj}\delta_k$ to get delta for every neuron.

Network Training Step-by-Step

1. **Forward Pass:** Input $\mathbf{x}$, compute activations layer-by-layer $(\mathbf{a}^{(l)}, \mathbf{z}^{(l)})$. Keep intermediate values in memory.
2. **Output Error Evaluation:** Compute initial signals $\delta_k = (\hat{y}_k - t_k)$ at the final layer.
3. **Backward Pass:** Propagate errors using $\delta_j = \sigma'(a_j) \sum w_{kj}\delta_k$ to get delta for every neuron.
4. **Gradient Calculation:** Calculate $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$ and update weights via gradient descent.

Computational Efficiency

- Forward pass operations are proportional to number of weights, $\mathcal{O}(W)$.

Computational Efficiency

- Forward pass operations are proportional to number of weights, $\mathcal{O}(W)$.
- Backpropagation allows gradients for *every single weight* to be calculated in roughly the same backward duration, $\mathcal{O}(W)$.

Computational Efficiency

- Forward pass operations are proportional to number of weights, $\mathcal{O}(W)$.
- Backpropagation allows gradients for *every single weight* to be calculated in roughly the same backward duration, $\mathcal{O}(W)$.
- This linear scale makes deep networks computationally viable.

Computational Efficiency

- Forward pass operations are proportional to number of weights, $\mathcal{O}(W)$.
- Backpropagation allows gradients for *every single weight* to be calculated in roughly the same backward duration, $\mathcal{O}(W)$.
- This linear scale makes deep networks computationally viable.

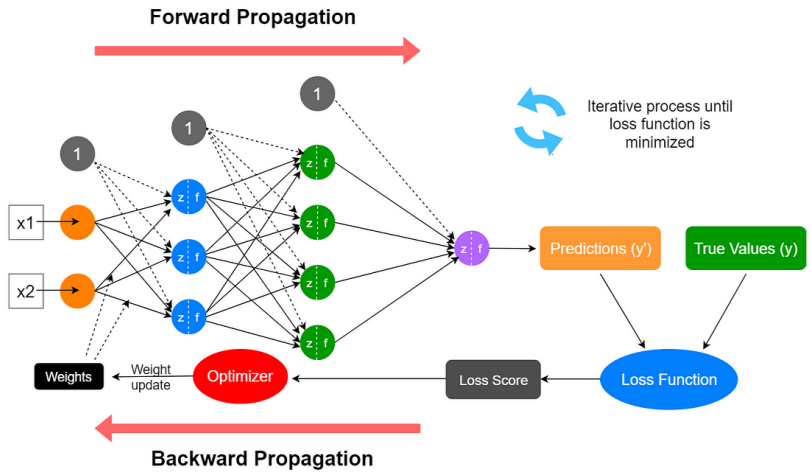

Figure: Forward pass and backward propagation in neural networks. (Source: Suresh Bikhani, LinkedIn)

Generalisation vs. Overfitting

- Minimising training error is not our ultimate goal.

Generalisation vs. Overfitting

- Minimising training error is not our ultimate goal.
- We want our network to perform well on **unseen data** (Generalization).

Generalisation vs. Overfitting

- Minimising training error is not our ultimate goal.
- We want our network to perform well on **unseen data** (Generalization).
- **Overfitting:** Occurs when a network performs exceptionally well on training data but fails on new, unseen test data.

The Bias-Variance Trade-off

Generalization error is broken down into two main competing components:

The Bias-Variance Trade-off

Generalization error is broken down into two main competing components:

1. **Bias (Underfitting):** Model architecture is too simple; predictions systematically differ from true values. (e.g., fitting curves with a straight line).

The Bias-Variance Trade-off

Generalization error is broken down into two main competing components:

1. **Bias (Underfitting):** Model architecture is too simple; predictions systematically differ from true values. (e.g., fitting curves with a straight line).
2. **Variance (Overfitting):** Model has high capacity but memorizes specific noise, quirks, and random fluctuations of small training samples. Performance plummets on new data.

Overfitting in Neural Networks

- MLPs have high expressive power and are highly prone to overfitting.

Overfitting in Neural Networks

- MLPs have high expressive power and are highly prone to overfitting.
- Left unchecked, they bend decision boundaries excessively to classify training points perfectly.

Overfitting in Neural Networks

- MLPs have high expressive power and are highly prone to overfitting.
- Left unchecked, they bend decision boundaries excessively to classify training points perfectly.
- **Regularization:** A collection of techniques used to control model capacity and prevent overfitting.

L_2 Regularization

- Modifies the error function $E(\mathbf{w})$ to penalize complexity by adding squared weight magnitudes:

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

L_2 Regularization

- Modifies the error function $E(\mathbf{w})$ to penalize complexity by adding squared weight magnitudes:

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

- Where:
  - $\|\mathbf{w}\|^2 = \sum_i w_i^2$ is the sum of squared network weights.

L_2 Regularization

- Modifies the error function $E(\mathbf{w})$ to penalize complexity by adding squared weight magnitudes:

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

- Where:
  - $\|\mathbf{w}\|^2 = \sum_i w_i^2$ is the sum of squared network weights.
  - $\lambda > 0$ is the regularisation coefficient hyperparameter.

L_2 Regularization

- The gradient of the regularized objective with respect to a weight w is:

$$\frac{\partial \tilde{E}}{\partial w} = \frac{\partial E}{\partial w} + \lambda w$$

L_2 Regularization

- The gradient of the regularized objective with respect to a weight w is:

$$\frac{\partial \tilde{E}}{\partial w} = \frac{\partial E}{\partial w} + \lambda w$$

- Plugging this into the gradient descent rule:

$$w^{(\tau+1)} = w^{(\tau)} - \eta \left(\frac{\partial E}{\partial w} + \lambda w^{(\tau)} \right)$$

L_2 Regularization

- The gradient of the regularized objective with respect to a weight w is:

$$\frac{\partial \tilde{E}}{\partial w} = \frac{\partial E}{\partial w} + \lambda w$$

- Plugging this into the gradient descent rule:

$$w^{(\tau+1)} = w^{(\tau)} - \eta \left(\frac{\partial E}{\partial w} + \lambda w^{(\tau)} \right)$$

- Rearranging highlights the decay:

$$w^{(\tau+1)} = (1 - \eta\lambda) w^{(\tau)} - \eta \frac{\partial E}{\partial w}$$

Weight Decay

- Since $(1 - \eta\lambda) < 1$, every step automatically shrinks weights toward zero before adding standard updates.

Weight Decay

- Since $(1 - \eta\lambda) < 1$, every step automatically shrinks weights toward zero before adding standard updates.
- Known as **weight decay**.

Weight Decay

- Since $(1 - \eta\lambda) < 1$, every step automatically shrinks weights toward zero before adding standard updates.
- Known as **weight decay**.
- Keeping weights small prevents any single neuron from exerting a dominant, sensitive influence, producing smoother functions.

L_1 Regularization

- Penalizes absolute values of weights rather than their squares:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_i |w_i|$$

L_1 Regularization

- Penalizes absolute values of weights rather than their squares:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_i |w_i|$$

- The derivative involves sign(w), yielding update rule:

$$w^{(\tau+1)} = w^{(\tau)} - \eta\lambda \cdot \text{sign}(w^{(\tau)}) - \eta \frac{\partial E}{\partial w}$$

L_1 Regularization

- Penalizes absolute values of weights rather than their squares:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_i |w_i|$$

- The derivative involves sign(w), yielding update rule:

$$w^{(\tau+1)} = w^{(\tau)} - \eta\lambda \cdot \text{sign}(w^{(\tau)}) - \eta \frac{\partial E}{\partial w}$$

  - Subtracts a constant force $\eta\lambda$ regardless of size.

L_1 Regularization

- Penalizes absolute values of weights rather than their squares:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_i |w_i|$$

- The derivative involves sign(w), yielding update rule:

$$w^{(\tau+1)} = w^{(\tau)} - \eta\lambda \cdot \text{sign}(w^{(\tau)}) - \eta \frac{\partial E}{\partial w}$$

  - Subtracts a constant force $\eta\lambda$ regardless of size.
  - Drives less important weights to **exactly zero**, creating a **sparse model**.

Algorithmic Regularization: Early Stopping

- Monitor error on both the training set and a separate validation set.

Algorithmic Regularization: Early Stopping

- Monitor error on both the training set and a separate validation set.
  1. Training error steadily decreases as parameters learn details.

Algorithmic Regularization: Early Stopping

- Monitor error on both the training set and a separate validation set.
  1. Training error steadily decreases as parameters learn details.
  2. Initially, validation error decreases alongside it (general patterns).

Algorithmic Regularization: Early Stopping

- Monitor error on both the training set and a separate validation set.
  1. Training error steadily decreases as parameters learn details.
  2. Initially, validation error decreases alongside it (general patterns).
  3. Eventually, overfitting starts $\rightarrow$ training error drops further, but validation error begins to **rise**.

Algorithmic Regularization: Early Stopping

- Monitor error on both the training set and a separate validation set.
  1. Training error steadily decreases as parameters learn details.
  2. Initially, validation error decreases alongside it (general patterns).
  3. Eventually, overfitting starts $\rightarrow$ training error drops further, but validation error begins to **rise**.
- Save parameters at the exact point where validation error is lowest; stop training early.

Architectural Regularization: Dropout

- For each forward pass during training, dropout randomly forces a subset of hidden units to output **zero**.

Architectural Regularization: Dropout

- For each forward pass during training, dropout randomly forces a subset of hidden units to output **zero**.
- Each neuron has a deactivation probability $p \approx 0.5$.

Architectural Regularization: Dropout

- For each forward pass during training, dropout randomly forces a subset of hidden units to output **zero**.
- Each neuron has a deactivation probability $p \approx 0.5$.
- Forces hidden units to learn robust features independent of co-adapted neighbors.

Algorithmic Regularization: Data Augmentation

- Increases effective dataset size synthetically to counter high capacity.

Algorithmic Regularization: Data Augmentation

- Increases effective dataset size synthetically to counter high capacity.
- For images, apply minor transformations that preserve core identities:
  - Random horizontal flips

Algorithmic Regularization: Data Augmentation

- Increases effective dataset size synthetically to counter high capacity.
- For images, apply minor transformations that preserve core identities:
  - Random horizontal flips
  - Small rotations or crops

Algorithmic Regularization: Data Augmentation

- Increases effective dataset size synthetically to counter high capacity.
- For images, apply minor transformations that preserve core identities:
  - Random horizontal flips
  - Small rotations or crops
  - Brightness/contrast/color shifts

Algorithmic Regularization: Data Augmentation

- Increases effective dataset size synthetically to counter high capacity.
- For images, apply minor transformations that preserve core identities:
  - Random horizontal flips
  - Small rotations or crops
  - Brightness/contrast/color shifts
- Forces adaptive basis functions to learn invariants.

Algorithmic Regularization: Data Augmentation

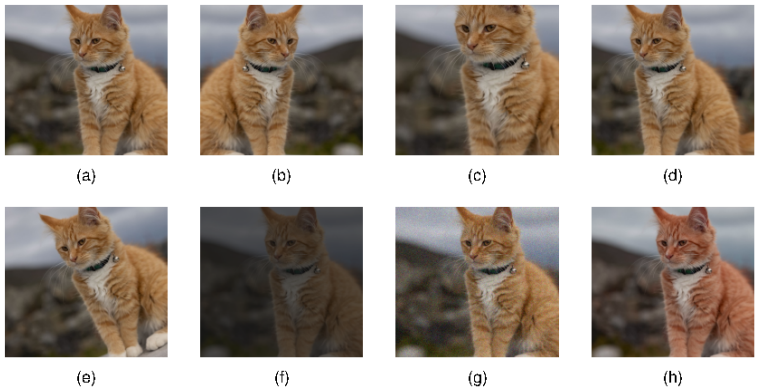

Figure: Illustration of data set augmentation, showing (a) the original image, (b) horizontal inversion, (c) scaling, (d) translation, (e) rotation, (f) brightness and contrast change, (g) additive noise, and (h) colour shift. (Source: Bishop DLFC)

Architectural Regularization: Dropout

- Effectively trains a different sub-network at every step.

Architectural Regularization: Dropout

- Effectively trains a different sub-network at every step.
- During test time inference, all neurons are turned back on.

Architectural Regularization: Dropout

- Effectively trains a different sub-network at every step.
- During test time inference, all neurons are turned back on.
- Scale weights down by factor $(1 - p)$ to match expected signal strengths.

Architectural Regularization: Dropout

- Effectively trains a different sub-network at every step.
- During test time inference, all neurons are turned back on.
- Scale weights down by factor $(1 - p)$ to match expected signal strengths.
- Acts as an **approximate ensemble** over thousands of unique subnetworks, boosting generalization.

Momentum

- Standard mini-batch gradient descent struggles with irregular curvatures.

Momentum

- Standard mini-batch gradient descent struggles with irregular curvatures.
- The optimisation process is like a heavy ball rolling down a hilly terrain toward a valley

Momentum

- Standard mini-batch gradient descent struggles with irregular curvatures.
- The optimisation process is like a heavy ball rolling down a hilly terrain toward a valley
- If it hits steep, narrow ravine, it bounces back and forth violently between side walls, making slow progress along the valley floor.

Momentum

- Introduces physical simulation by remembering previous update velocity.

Momentum

- Introduces physical simulation by remembering previous update velocity.
- Gathers speed down consistent slopes, creating larger updates.

Momentum

- Introduces physical simulation by remembering previous update velocity.
- Gathers speed down consistent slopes, creating larger updates.
- Back-and-forth erratic updates cancel out, damping oscillations.

Momentum

- Introduces physical simulation by remembering previous update velocity.
- Gathers speed down consistent slopes, creating larger updates.
- Back-and-forth erratic updates cancel out, damping oscillations.
- Helps push straight through flat saddle points where standard GD stalls.

The Need for Adaptive Learning Rates

- A single learning rate applies across all network weights.

The Need for Adaptive Learning Rates

- A single learning rate applies across all network weights.
- Some weights might have massive gradients (steep cliffs) whiles others have tiny gradients (flat plains).

The Need for Adaptive Learning Rates

- A single learning rate applies across all network weights.
- Some weights might have massive gradients (steep cliffs) whiles others have tiny gradients (flat plains).
- One rate may be too large for the steep sections cliffs (causing crashes) and too small for plains (causing stagnation).

Adaptive Optimizers: RMSprop

- Tracks a running average of gradient volatility individually per weight.

Adaptive Optimizers: RMSprop

- Tracks a running average of gradient volatility individually per weight.
- If a weight has massive, erratic updates $\rightarrow$ scale down individual learning rate to stabilize.

Adaptive Optimizers: RMSprop

- Tracks a running average of gradient volatility individually per weight.
- If a weight has massive, erratic updates $\rightarrow$ scale down individual learning rate to stabilize.
- If a weight has tiny, sluggish updates $\rightarrow$ scales up individual rate to accelerate.

Adaptive Optimizers: Adam

Adaptive Moment Estimation (Adam)

Combines the strengths of both Momentum and RMSprop.

Adaptive Optimizers: Adam

Adaptive Moment Estimation (Adam)

Combines the strengths of both Momentum and RMSprop.

- Remembers past directional travel (Momentum).

Adaptive Optimizers: Adam

Adaptive Moment Estimation (Adam)

Combines the strengths of both Momentum and RMSprop.

- Remembers past directional travel (Momentum).
- Dynamically scales step sizes per parameter based on recent volatility (RMSprop).

Adaptive Optimizers: Adam

Adaptive Moment Estimation (Adam)

Combines the strengths of both Momentum and RMSprop.

- Remembers past directional travel (Momentum).
- Dynamically scales step sizes per parameter based on recent volatility (RMSprop).
- Default, go-to optimizer choice for most modern deep learning projects.

The Problem of Scale

- Weights cannot start at zero: neurons would compute identical outputs and updates, failing to differentiate to learn unique features.

The Problem of Scale

- Weights cannot start at zero: neurons would compute identical outputs and updates, failing to differentiate to learn unique features.
- Random initialization without care causes two extreme signal disasters across deep layers.
  1. **Vanishing Gradients:** If starting weights are too small, signals shrink at every layer, fading to practically zero in early layers. Learning stalls.

The Problem of Scale

- Weights cannot start at zero: neurons would compute identical outputs and updates, failing to differentiate to learn unique features.
- Random initialization without care causes two extreme signal disasters across deep layers.
  1. **Vanishing Gradients:** If starting weights are too small, signals shrink at every layer, fading to practically zero in early layers. Learning stalls.
  2. **Exploding Gradients:** If weights are too large, signals multiply exponentially layer by layer, growing too massive for hardware limits and crashing the model.

Smart Initialization Schemes

- Designed to keep signal variances stable across layers:

Smart Initialization Schemes

- Designed to keep signal variances stable across layers:
  - **Xavier (Glorot) Initialization:** Tailored for sigmoidal or tanh layers. Scales weights based on count of input and output connections.

Smart Initialization Schemes

- Designed to keep signal variances stable across layers:
  - **Xavier (Glorot) Initialization:** Tailored for sigmoidal or tanh layers. Scales weights based on count of input and output connections.
  - **He (Kaiming) Initialization:** Tailored for ReLU layers. Accounts for ReLU turning off half of the neurons on average, preventing signal fade.

Batch Normalization

- As weights shift during training, input distributions to deeper layers fluctuate wildly, demanding constant baseline adaptation.

Batch Normalization

- As weights shift during training, input distributions to deeper layers fluctuate wildly, demanding constant baseline adaptation.
- **Batch Normalization** adds a calibration step between layers.

Batch Normalization

- As weights shift during training, input distributions to deeper layers fluctuate wildly, demanding constant baseline adaptation.
- **Batch Normalization** adds a calibration step between layers.
- For every mini-batch, it calculates average and flattens ranges, centering data around zero with a predictable standard width.

Batch Normalization

- Prevents variance variations from compounding across deep layers.

Batch Normalization

- Prevents variance variations from compounding across deep layers.
- Makes the network highly resilient.

Batch Normalization

- Prevents variance variations from compounding across deep layers.
- Makes the network highly resilient.
- Permits much higher learning rates safely.

Batch Normalization

- Prevents variance variations from compounding across deep layers.
- Makes the network highly resilient.
- Permits much higher learning rates safely.
- Significantly speeds up total training time.

Summary of Neural Network Journey

- We have covered:
  1. Adaptive basis functions avoiding the curse of dimensionality

Summary of Neural Network Journey

- We have covered:
  1. Adaptive basis functions avoiding the curse of dimensionality
  2. Matrix formulation of forward and backward passes

Summary of Neural Network Journey

- We have covered:
  1. Adaptive basis functions avoiding the curse of dimensionality
  2. Matrix formulation of forward and backward passes
  3. Loss surface geometry and calculus derivation of backprop

Summary of Neural Network Journey

- We have covered:
  1. Adaptive basis functions avoiding the curse of dimensionality
  2. Matrix formulation of forward and backward passes
  3. Loss surface geometry and calculus derivation of backprop
  4. Regularization and advanced optimization tricks for successful training