Foundations of Neural Networks

September 28, 2025

Contents

1 Artificial Neuron Model (Perceptron)

The perceptron is the simplest artificial neuron. It computes a linear combination of inputs and passes the result through a step function to obtain a binary output. Formally, for input vector $\mathbf{x} = [x_1, \ldots, x_d]^\top$, weight vector $\mathbf{w} = [w_1, \ldots, w_d]^\top$, and bias b, the perceptron output is

$$y = \operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} + b\right), \tag{1}$$

where $\operatorname{sign}(z) = 1$ if $z \geq 0$ and -1 otherwise. The hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$ partitions the input space into two classes.

1.1 Learning Rule

The classic perceptron learning algorithm performs stochastic gradient descent on the hinge-like loss by updating weights when a mistake occurs on a labeled example $(\mathbf{x}, t)$ with $t \in \{-1, 1\}$. The update reads

$$\mathbf{w} \leftarrow \mathbf{w} + \eta t \mathbf{x}, \quad b \leftarrow b + \eta t, \tag{2}$$

with learning rate $\eta > 0$. The update encourages the decision boundary to move toward correctly classifying the mispredicted point. Convergence is guaranteed if the data are linearly separable.

1.2 Geometric Intuition

Figure **??** illustrates the perceptron decision boundary and the signed distances of points from the separating hyperplane.

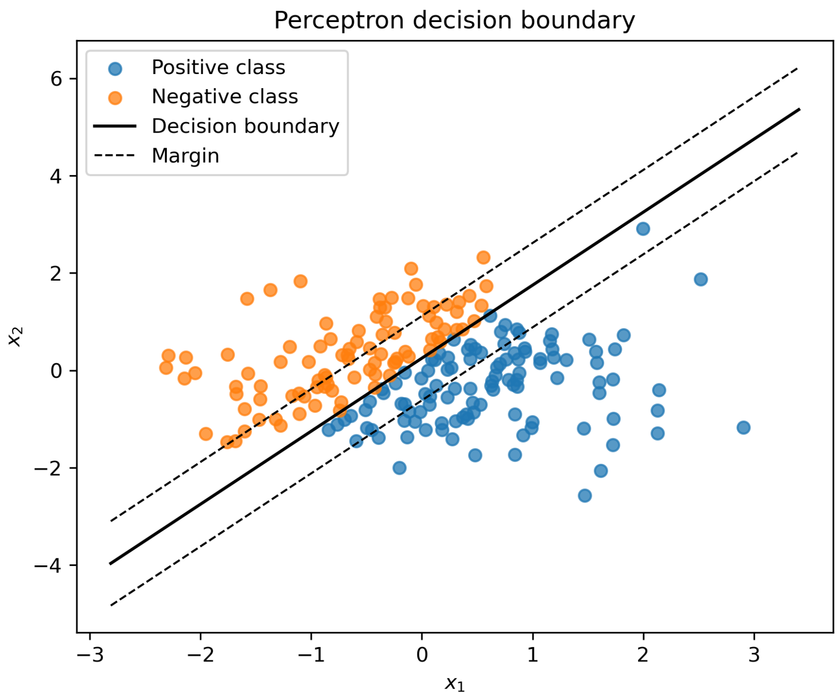

Figure 1: Perceptron decision boundary separating two classes with margin intuition.

2 Multilayer Perceptron (MLP) and Forward Propagation

An MLP stacks layers of perceptrons with differentiable activation functions, enabling the model to approximate complex nonlinear mappings. Given an L-layer MLP, the computation proceeds layer by layer:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \qquad \mathbf{h}^{(1)} = \phi^{(1)}\big(\mathbf{a}^{(1)}\big), \tag{3}$$

$$\mathbf{a}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{h}^{(\ell)} = \phi^{(\ell)}\big(\mathbf{a}^{(\ell)}\big), \tag{4}$$

$$\hat{\mathbf{y}} = \mathbf{h}^{(L)}, \tag{5}$$

where $\phi^{(\ell)}$ denotes the element-wise activation of layer ℓ.

2.1 Forward Propagation Algorithm

Forward propagation evaluates the network efficiently by reusing intermediate results.

Listing 1: Forward pass for a dense MLP.

```python
import numpy as np

def forward_pass(weights, biases, activations, x):
    h = x
    for W, b, act in zip(weights, biases, activations):
        a = W @ h + b
        h = act(a)
    return h
```

2.2 Expressive Power

The universal approximation theorem states that a feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$ given appropriate activation functions. Deeper networks reduce the number of neurons required by reusing intermediate abstractions.

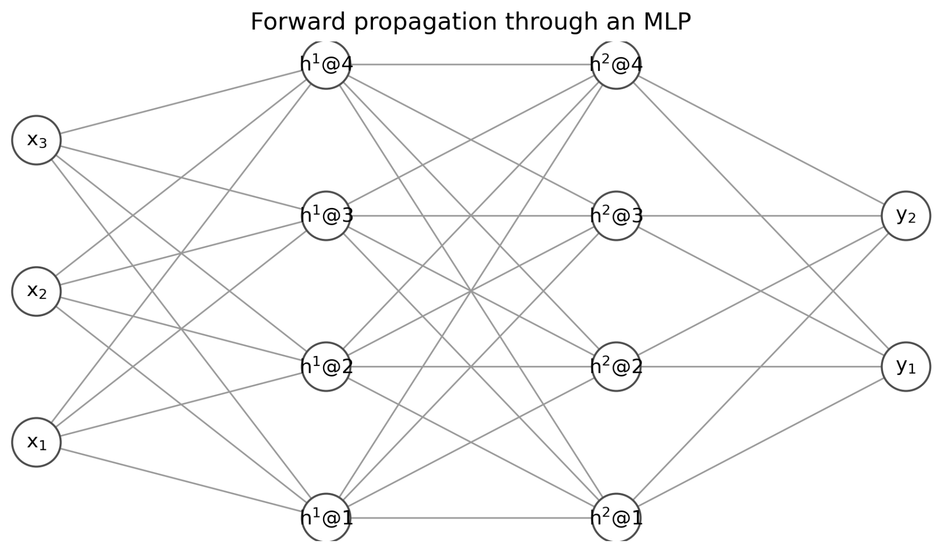

Figure 2: Forward propagation through an MLP with highlighted linear maps and activations.

3 Activation Functions

Activation functions inject nonlinearity, enabling neural networks to model complex relationships. Figure **??** compares several common activations.

3.1 Sigmoid

The logistic sigmoid maps real numbers to $(0, 1)$:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)\big(1-\sigma(z)\big). \tag{6}$$

It saturates for large $|z|$, which can slow training.

3.2 Hyperbolic Tangent

The tanh activation rescales the sigmoid to $(-1, 1)$ and is zero-centered:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z). \tag{7}$$

3.3 Rectified Linear Unit (ReLU)

ReLU is defined as

$$\text{ReLU}(z) = \max(0, z), \quad \text{ReLU}'(z) = \begin{cases} 1, & z > 0, \\ 0, & z < 0. \end{cases} \tag{8}$$

It accelerates convergence but suffers from "dead" neurons when $z < 0$ persistently.

3.4 Leaky ReLU

Leaky ReLU mitigates dead neurons by allowing a small slope for negative inputs:

$$\mathrm{LeakyReLU}(z) = \begin{cases} z, & z \geq 0, \\ \alpha z, & z < 0, \end{cases} \tag{9}$$

with $\alpha \approx 0.01$.

3.5 Gaussian Error Linear Unit (GELU)

GELU weights inputs by their magnitude and probability under a standard Gaussian:

$$\mathrm{GELU}(z) = z\Phi(z) = \frac{z}{2}\left[1 + \mathrm{erf}\left(\frac{z}{\sqrt{2}}\right)\right], \tag{10}$$

where Φ is the Gaussian cumulative distribution function and erf is the error function. GELU maintains smooth derivatives, facilitating optimization in transformer architectures.

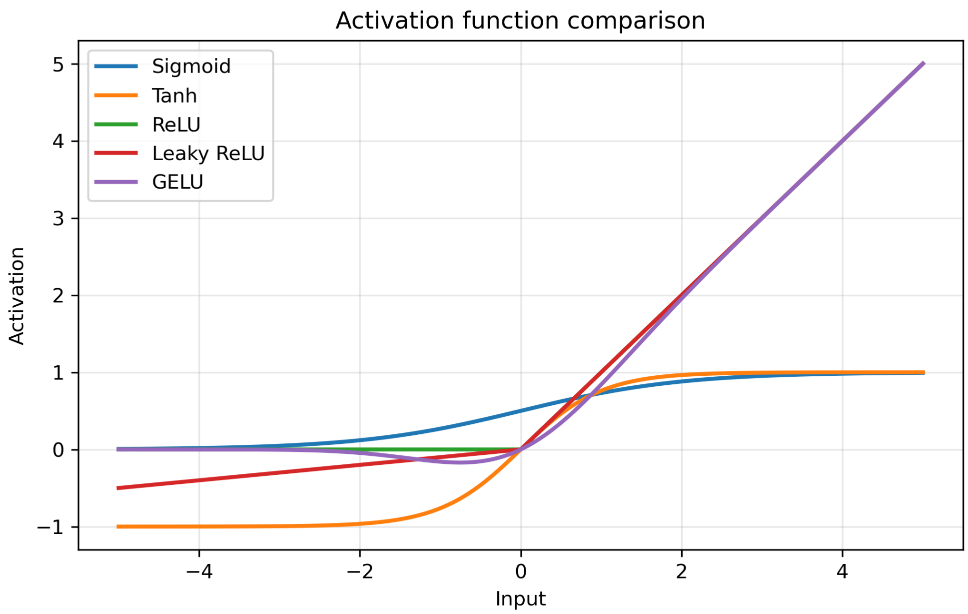

Figure 3: Comparison of common activation functions.

4 Loss Functions

Loss functions quantify the discrepancy between predictions and targets, guiding gradient-based optimization.

4.1 Mean Squared Error (MSE)

For regression with targets t_i and predictions $\hat{y}_i$, the MSE is

$$\mathcal{L}_{\mathrm{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - t_i)^2. \tag{11}$$

The gradient with respect to $\hat{y}_i$ is $\frac{2}{N}(\hat{y}_i - t_i)$.

4.2 Cross-Entropy Loss

For binary classification with targets $t_i \in \{0, 1\}$ and predicted probabilities p_i, the binary cross-entropy loss reads

$$\mathcal{L}_{\mathrm{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[t_i \log p_i + (1-t_i)\log(1-p_i)\right]. \tag{12}$$

When combined with the logistic sigmoid, this loss aligns the gradient with the negative log-likelihood of a Bernoulli distribution.

For multi-class classification with softmax outputs $p_{i,k}$ over classes k, the categorical cross-entropy is

$$\mathcal{L}_{\mathrm{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} t_{i,k}\log p_{i,k}, \tag{13}$$

where $t_{i,k}$ is the one-hot target.

4.3 Huber Loss

The Huber loss interpolates between L1 and L2 losses, offering robustness to outliers while retaining differentiability at the origin:

$$\mathcal{L}_\delta(r) = \begin{cases} \frac{1}{2}r^2, & |r| \le \delta, \\ \delta(|r| - \frac{1}{2}\delta), & |r| > \delta, \end{cases} \tag{14}$$

where $r = \hat{y} - t$ and δ controls the transition point.

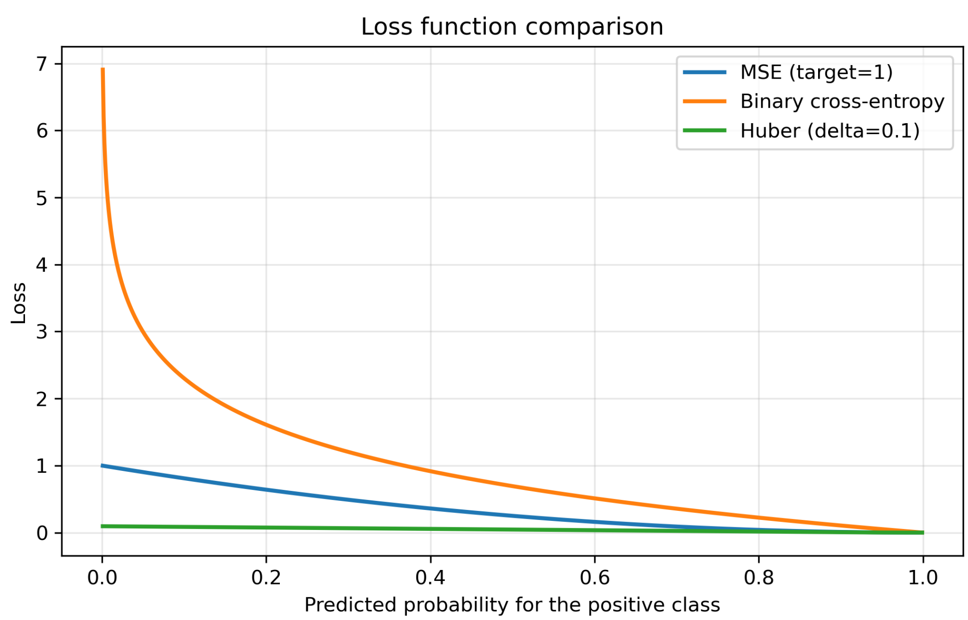

Figure 4: Shapes of MSE, cross-entropy (binary log-loss), and Huber losses.

5 Practical Considerations

- **Initialization:** Proper weight initialization (e.g., Xavier or He) prevents activations from vanishing or exploding.
- **Normalization:** Batch or layer normalization stabilizes training by controlling activation statistics.

- **Optimization:** Adaptive optimizers (Adam, RMSprop) adjust learning rates per parameter.
- **Regularization:** Techniques such as dropout, weight decay, or early stopping combat overfitting.