Foundations of AI **Fall 2025**
Professor Eric Ewing[1] **Machine Learning Unit**

Perceptrons

1 Introduction

Perceptrons represent one of the earliest models of artificial neural networks, introduced in the 1950s and 1960s as a simplified model of biological neurons. Despite their simplicity, perceptrons provide fundamental insights into neural computation and serve as the building blocks for more complex neural network architectures.

In this lecture, we study perceptrons as binary classifiers that learn to separate data using linear decision boundaries. We begin with the structure of neural units, then examine the perceptron learning rule, and finally discuss the fundamental limitations that motivated the development of multi-layer networks.

2 Neural Units

Motivated by the design of biological learning systems, artificial neural networks are complex webs of simple *neural units* interconnected in parallel structures. The connections among the units are typically described using a directed graph, with edges that are labeled by weights. Each processing unit is equipped with a very simple program, which (i) computes a weighted sum of the input data it receives from those units which feed into it, and (ii) outputs a single value, which is some function of the weighted sum of its inputs.

2.1 Binary Threshold Units

In 1943, McCulloch and Pitts proved that artificial neural networks composed of binary threshold units are universal machines capable of computing any boolean function. These networks take as input binary vector $\vec{x} = (x_1, \ldots, x_n)$ and output binary vector $\vec{z} = (z_1, \ldots, z_m)$ such that:

$$z_k = g\left(\sum_{i=1}^{n} w_{ik} x_i - \theta_k\right) \tag{1}$$

where the *transfer function* g is the step function, defined as follows:

$$g(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In Equation 1, w_{ik} denotes the strength of the connection from the ith input to the kth output. If $w_{ik} > 0$, the connection is *excitatory*, providing a positive influence on the output. If $w_{ik} < 0$, the connection is *inhibitory*, providing a negative influence. Finally, if $w_{ik} = 0$, there is no connection at all between that input and output.

[1] This notes were compiled in conjunction with Professor Amy Greenwald.

2.2 Incorporating the Threshold

The parameter θ_k is the threshold value. This parameter can be absorbed into the summation by assuming an additional input whose value is fixed at either -1 or $+1$. In the former case, this yields $\theta_k = w_{0k}$, whereas in the latter, this yields $\theta_k = -w_{0k}$. This gives us the more compact notation:

$$z_k = g\left(\sum_{i=0}^{n} w_{ik} x_i\right) = g(h_k) \tag{3}$$

where

$$h_k = \sum_{i=0}^{n} w_{ik} x_i = \vec{w}_k \cdot \vec{x} \tag{4}$$

This representation treats the threshold as just another weight, simplifying both notation and implementation.

3 Perceptron Architecture

Perceptrons are feed-forward, layered networks. Feed-forward implies that such networks form directed acyclic graphs (DAGs). An N-layered network has N layers of connections; specifically, such a network has a single input layer, and N additional layers of nodes, where the final layer is called the output layer, and the $N-1$ internal layers are called hidden layers.

3.1 Simple Perceptrons

The *simple perceptron* has no hidden layers; it is a 1-layer network with only an output layer. This is the model we focus on in this lecture. For a simple perceptron performing binary classification, we use threshold units based on the sign function:

$$g(h) = \text{sgn}(h) = \begin{cases} +1 & \text{if } h \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{5}$$

Since there is no dependence among the outputs in a simple perceptron, it suffices to consider them one at a time. We therefore drop the dependence on k and write:

$$z = \text{sgn}(\vec{w} \cdot \vec{x}) \tag{6}$$

4 The Perceptron Learning Rule

Given a data set D consisting of examples of the form $\langle \vec{x}, y \rangle$, with $\vec{x} \in \mathbb{R}^n$ and $y \in \{+1, -1\}$, the perceptron rule updates weight w_i as follows:

$$\Delta w_i = \alpha(y - z)x_i \tag{7}$$

where α is the learning rate.

4.1 Intuition Behind the Rule

The perceptron learning rule is a *mistake-driven* rule. If the output is correct (meaning $y = z$), the algorithm makes no changes to the weights. However, when the perceptron makes an error ($y \neq z$), the rule adjusts the weights to move the output in the correct direction.

When $z = -1$ but the true label is $y = +1$, we need to increase the weighted sum $h = \vec{w} \cdot \vec{x}$ to push the output toward the positive class. The rule accomplishes this by increasing weights on positive inputs (making their positive contribution stronger) and decreasing weights on negative inputs (making their negative contribution weaker).

Conversely, when $z = 1$ but the true label is $y = -1$, we need to decrease the weighted sum to push the output toward the negative class. The rule decreases weights on positive inputs and increases weights on negative inputs.

4.2 Mathematical Verification

Let us verify that Equation 7 achieves its intended purpose by considering several cases.

First, assume input $x_i > 0$. If the true label is $y = +1$ but the perceptron outputs $z = -1$, then $y - z = 2$ is positive, which makes $\Delta w_i > 0$. This increases the weight, which is correct since this positive input should contribute more positively to push the output toward $+1$. On the other hand, if $y = -1$ but $z = +1$, then $y - z = -2$ is negative, so $\Delta w_i < 0$. This decreases the weight, which is again correct since we want to reduce this positive input's contribution.

Next, assume input $x_i < 0$. If $y = +1$ and $z = -1$, then $y - z = 2$ is positive, but since $x_i < 0$, we have $\Delta w_i < 0$. This decreases the weight on a negative input, which effectively reduces the negative contribution and helps push the output toward $+1$. Similarly, if $y = -1$ and $z = +1$, then $y - z = -2$ is negative, but with $x_i < 0$, we get $\Delta w_i > 0$. This increases the weight on a negative input, making the negative contribution stronger and helping push the output toward -1.

Finally, if $x_i = 0$, then $\Delta w_i = 0$ regardless of the error, which is appropriate since this input makes no contribution to the output.

4.3 The Perceptron Algorithm

The complete perceptron learning algorithm is presented below:

PERCEPTRON(D, ϵ, α)

Inputs:	$D = (D_{\text{train}}, D_{\text{val}})$ data set
	ϵ convergence condition
	α learning rate
Output:	trained weights $\vec{w}$
Initialize:	weights w_i randomly (often to small values near 0)

`while` ($\Delta\text{error}(D_{\text{val}}) > \epsilon$) `do`

1. For each training example $\langle \vec{x}, y \rangle \in D_{\text{train}}$:
   (a) Compute output: $z = \text{sgn}(\vec{w} \cdot \vec{x})$
   (b) If $y = z$, continue to next example
   (c) Else, for all weights w_i:
      i. $\Delta w_i = \alpha(y - z)x_i$
      ii. $w_i \leftarrow w_i + \Delta w_i$
2. Compute $\Delta\text{ERROR}(D_{\text{val}})$ on validation set

`return` weights $\vec{w}$

5 Convergence Theorem

Theorem [Minsky and Papert, 1969]
The perceptron rule converges to weights that correctly classify all training examples, provided the given data set can be separated by a linear function.

This is a remarkable result: if the data is linearly separable, the perceptron algorithm is *guaranteed* to find a solution in finite time.

5.1 Linear Separability

The set of vectors $\{\vec{x} \mid \vec{w} \cdot \vec{x} = 0\}$ determines a *hyperplane*, which is a generalization of a line to multiple dimensions. Recall that $g(h) = +1$ if and only if $h \geq 0$ if and only if $\vec{w} \cdot \vec{x} \geq 0$.

In other words, the network output is positive for all $\vec{x}$ that lie on the non-negative side of the hyperplane, and negative otherwise. A function is said to be *linearly separable* if it can be separated in this way by a hyperplane.

5.2 Examples of Linearly Separable Functions

The Boolean function AND is linearly separable and can be represented by a simple perceptron. We can use inputs x_1 and x_2 with weights $w_1 = w_2 = 1$, along with a bias input $x_0 = -1$ with weight $w_0 = 1.5$. This gives $z = \text{sgn}(x_1 + x_2 - 1.5)$, which correctly computes the AND function. The decision boundary requires both inputs to be positive (contributing a total of $+2$) to overcome the negative bias of -1.5 and produce a positive output.

Similarly, the function OR is linearly separable. We can represent it using inputs x_1 and x_2 with weights $w_1 = w_2 = 1$ and a bias input $x_0 = -1$ with weight $w_0 = 0.5$. This gives $z = \text{sgn}(x_1 + x_2 - 0.5)$, which correctly computes OR. The lower bias threshold means that either input being positive is sufficient to produce a positive output.

6 The XOR Problem

The function XOR (exclusive OR), however, is *not* linearly separable, and hence cannot be represented by a simple perceptron. This fundamental limitation became widely known through Minsky and Papert's 1969 book *Perceptrons*, which analyzed the computational capabilities and limitations of these models.

Consider the XOR function with truth table:

x_1	x_2	XOR
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

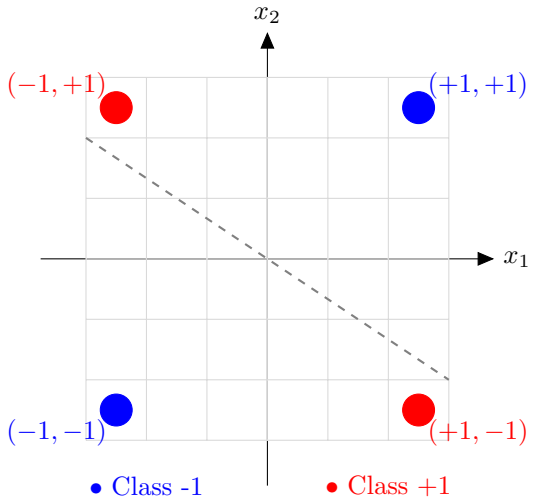

Figure 1: The XOR problem visualized. Blue points (class -1) and red points (class $+1$) cannot be separated by any single straight line. This demonstrates that XOR is not linearly separable.

As shown in Figure 1, there is no way to draw a single straight line (or hyperplane in higher dimensions) that correctly separates the positive examples from the negative examples.

This realization (among other factors as well) stagnated neural network research for about a decade in the 1970s. However, it was known even then that XOR *can* be represented by a multi-layer perceptron with hidden units. This insight eventually led to the development of backpropagation and the resurgence of neural network research in the 1980s.

7 Geometric Interpretation

The perceptron defines a linear decision boundary in the input space. The weight vector $\vec{w}$ is perpendicular to this decision boundary and points in the direction of the positive class. The magnitude of $\vec{w} \cdot \vec{x}$ represents the distance from point $\vec{x}$ to the decision boundary, up to a scaling factor determined by the magnitude of $\vec{w}$.

When the perceptron makes an error, the weight update has a clear geometric interpretation. If the perceptron predicts -1 but the true label is $+1$, the point is on the wrong side of the decision boundary. The

update rule rotates the weight vector toward $\vec{x}$, moving the decision boundary to bring this point onto the correct side. Conversely, if it predicts $+1$ but the true label is -1, the update rule rotates the weight vector away from $\vec{x}$, again moving the boundary to correct the classification.

This geometric interpretation explains why the algorithm converges for linearly separable data. Each update moves the decision boundary in a direction that reduces the number of misclassified points. Since the data is linearly separable, there exists some orientation of the hyperplane that correctly classifies all points, and the algorithm systematically rotates toward this correct orientation.

8 Practical Considerations

8.1 Learning Rate

The learning rate α controls how much we adjust weights in response to errors. A large learning rate leads to faster learning but may cause the algorithm to oscillate around the optimal solution or even overshoot it entirely. A small learning rate produces slower but more stable convergence, as each weight update makes only a small change to the decision boundary. In practice, it is common to use a decaying learning rate that starts relatively large and decreases over time, allowing rapid initial progress followed by fine-tuning adjustments.

8.2 Initialization

Weights are typically initialized to small random values, often uniformly distributed in an interval like $[-0.5, 0.5]$. This random initialization serves two important purposes. First, it breaks symmetry among the weights, ensuring that different weights can learn different patterns. Second, it provides a reasonable starting point for the learning process without biasing the algorithm toward any particular solution.

9 Summary and Looking Ahead

Perceptrons provide a foundational model for understanding neural computation. They efficiently learn linear decision boundaries through simple weight updates based on classification errors. The perceptron convergence theorem guarantees that the algorithm will find a solution whenever the data is linearly separable. However, perceptrons cannot learn non-linearly separable functions like XOR, which fundamentally limits their expressiveness.

This fundamental limitation motivates the study of multi-layer perceptrons (MLPs) and more sophisticated neural network architectures. By adding hidden layers with non-linear activation functions, MLPs can learn arbitrary decision boundaries and represent any continuous function to arbitrary precision. The challenge that faced early researchers was how to train these more complex networks, since the perceptron rule only applies to single-layer networks.

In the next lecture, we will explore how gradient descent and backpropagation enable learning in these more expressive multi-layer networks. These techniques extend the basic ideas of the perceptron rule to networks with hidden layers, opening up the full power of neural network learning.