Minor in AI

Perceptrons & Neural Network Fundamentals

1 The Decision-Making Motivation

Every day, we make countless decisions based on simple conditions: Do my homework? Is my room tidy? For a child pleading to watch their favorite movie—say, *The Avengers*—parents often set clear requirements: ”Finish your homework AND clean your room!”

This real-world scenario mirrors a fundamental concept in machine learning: decision boundaries. Just as a strict parent insists on both tasks being completed, an algorithm can be trained to follow a precise rule that switches from “no” to “yes” when certain criteria are met.

In this section, we'll explore:

- How basic logical rules govern simple yes/no decisions.
- Why modeling these rules is the first step toward building more complex learners.
- How translating everyday choices into algorithmic conditions reveals the power and limitations of early neural architectures.

By framing the parental example as a logical gate, we bridge intuition with computational models. This lays the groundwork for understanding how perceptrons—a precursor to modern neural networks—can learn to replicate decision-making patterns, from natural language parsing to image recognition.

Key Components

- **Inputs**: Binary states (e.g., 0/1 for tasks)
- **Weights**: Importance of each input
- **Bias**: Flexibility factor (e.g., allowing 1 unfinished task)
- **Threshold**: Activation condition

2 Anatomy of a Perceptron

Perceptrons are the simplest units of a neural network. They classify inputs into two categories by computing a weighted sum of features, adding a bias, and then applying a threshold function.

2.1 Decision Formula

A perceptron computes an activation score z:

$$z = \sum_i x_i w_i + b \quad \Longrightarrow \quad y_{\text{pred}} = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Term Definitions:

- x_i: Input feature i, representing observed data (e.g., homework done = 1, not done = 0).

- w_i: Weight for feature i, quantifying its importance in the decision.
- b: Bias term, allowing the activation boundary to shift away from the origin.
- z: Activation score, the raw decision signal before thresholding.
- y_{pred}: Model prediction (0 or 1) after applying the threshold.

2.2 Learning Through Error Correction

To teach a perceptron, we compare its prediction y_{pred} to the true label y_{true} and update parameters to reduce errors. The classic update rule is:

Parameter Update

$$w_{i,\text{new}} = w_{i,\text{old}} + \alpha\,(y_{\text{true}} - y_{\text{pred}})\,x_i, \quad b_{\text{new}} = b_{\text{old}} + \alpha\,(y_{\text{true}} - y_{\text{pred}}).$$

Term Definitions:

- α: Learning rate, a small positive constant that controls how much we adjust on each mistake.
- y_{true}: Actual label (0 or 1) for the current example.
- y_{pred}: Perceptron's current prediction.
- $(y_{\text{true}} - y_{\text{pred}})$: Error signal indicating under- or over-prediction.
- $w_{i,\text{new}}, b_{\text{new}}$: Updated weight and bias after correction.

2.3 Practical Implementation: AND Gate

```python
import numpy as np

# Define input patterns for the AND truth table
X = np.array([[0,0], [0,1], [1,0], [1,1]])
# Corresponding target outputs: only (1,1)     1
y = np.array([0, 0, 0, 1])

# Initialize weights and bias to zero
weights = np.zeros(2)
bias = 0

# Hyperparameters: how fast to learn, and how many passes over the data
learning_rate = 1
epochs = 10

def step(x):
    """Step activation: returns 1 if x >= 0, else 0."""
    return 1 if x >= 0 else 0

# Training loop
for epoch in range(epochs):
    for i in range(len(X)):
        # Compute the weighted sum plus bias
        z = np.dot(X[i], weights) + bias
        # Apply step function to get prediction
        y_pred = step(z)
        # Calculate error (difference between true label and prediction)
        error = y[i] - y_pred
        # Update weights and bias based on error
        weights += learning_rate * error * X[i]
        bias += learning_rate * error
```

Listing 1: Perceptron Learning Code

Code Explanation

- **Data Setup:**
  - `X` contains all binary input combinations for an AND gate.
  - `y` holds the target outputs (only `[1,1]` maps to 1).
- **Initialization:**
  - `weights = [0, 0]` and `bias = 0` start the model with no initial preference.
- **Hyperparameters:**
  - α (`learning_rate`) controls update magnitude.
  - `epochs` specifies how many passes over the entire dataset.
- **Activation Function:**
  - `step(x)` returns 1 if the input $x \geq 0$, else 0, implementing a binary threshold.
- **Training Loop:**
  1. Compute the activation $z = \mathbf{x} \cdot \mathbf{w} + b$.
  2. Predict $\hat{y} = \text{step}(z)$.
  3. Calculate the error $e = y_{\text{true}} - \hat{y}$.
  4. Update each weight $w_i \leftarrow w_i + \alpha\, e\, x_i$ and bias $b \leftarrow b + \alpha\, e$.

3 The XOR Limitation

XOR Truth Table

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Although the perceptron handles AND and OR easily, it fails on XOR. The reason lies in *linear separability*:

- A single-layer perceptron computes a decision boundary of the form
$$w_1x_1 \;+\; w_2x_2 \;+\; b \;=\; 0,$$
which geometrically is a line in the two-dimensional (x_1, x_2) plane.
- For XOR, the positive examples $(0, 1)$ and $(1, 0)$ lie in opposite quadrants, while the negative examples $(0, 0)$ and $(1, 1)$ occupy the other two. No single straight line can separate these classes.

Overcoming the Limitation

To solve XOR, we need to introduce nonlinearity via a hidden layer:

- **Multi-Layer Perceptron (MLP):** Adds one or more hidden neurons, each learning its own weighted sum and activation.
- **Nonlinear Activation:** Functions like sigmoid or ReLU allow the network to carve out complex regions in input space.
- **Universal Approximation:** With sufficient hidden units, an MLP can approximate any Boolean function, including XOR.

Thus, while a single perceptron is limited to linearly separable tasks, deeper architectures unlock the full power of neural networks.“‘

4 Neural Network Evolution

4.1 Gradient Descent

Gradient descent is the fundamental optimization algorithm used to adjust network parameters so as to minimize a chosen loss function. At each iteration, we compute how the loss changes with respect to each weight and then move the weight in the direction that most rapidly decreases the loss:

$$w \;\leftarrow\; w \;-\; \alpha\,\frac{\partial \mathcal{L}}{\partial w}$$

Term Definitions:

- $\mathcal{L}$: *Loss function*, e.g. mean squared error or cross-entropy, measuring discrepancy between predictions and true labels.
- $\frac{\partial \mathcal{L}}{\partial w}$: *Gradient*, the rate of change of the loss with respect to weight w, computed via backpropagation.
- α: *Learning rate*, a small positive constant controlling the step size of each update.
- w: A generic network parameter (weight or bias) being optimized.

Variants of Gradient Descent

- **Batch GD:** Compute gradients over the entire training set before each update. Stable but can be slow.
- **Stochastic GD:** Update after each training example. Faster convergence on large datasets but more noisy.
- **Mini-batch GD:** A compromise using small batches for each update, balancing speed and stability.

4.2 Activation Functions

Activation functions introduce nonlinearity, enabling networks to learn complex mappings.

- **Step Function**

  $$\text{step}(z) = \begin{cases} 1 & z \geq 0, \\ 0 & z < 0. \end{cases}$$

  Use: Early perceptrons. *Pros:* Simple thresholding. *Cons:* No gradient for learning; cannot train deep networks.

- **Sigmoid (Logistic) Function**

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

  Use: Binary classification outputs. *Pros:* Smooth, differentiable everywhere. *Cons:* Vanishing gradients for large $|z|$, outputs not zero-centered.

- **Hyperbolic Tangent (tanh)**

  $$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

  Pros: Zero-centered outputs in $[-1, 1]$, stronger gradients near zero. *Cons:* Still suffers from saturation at extremes.

- **ReLU (Rectified Linear Unit)**

  $$\text{ReLU}(z) = \max(0, z).$$

  Use: Hidden layers in modern deep networks. *Pros:* Simple, efficient gradients when $z > 0$, promotes sparse activations. *Cons:* "Dying ReLU" problem when neurons get stuck at zero.

5 Case Studies

Cat Recognition

- **Input Layer**: Raw pixel values capture visual cues—fur texture, whisker outlines, ear shape.
- **Hidden Layer**: Neurons learn to detect subfeatures such as edges, spots, and curves, combining them into higher-order patterns like whisker groupings or ear geometry.
- **Output Layer**: A threshold neuron aggregates hidden signals to decide "cat" vs. "not cat."

Explanation: In this example, training adjusts hidden-layer weights so that primitive detectors (edges, blobs) become specialized for feline features. Over epochs, the network learns to fire the "cat" neuron strongly when patterns like pointed ears and distinctive whiskers are present.

6 Golden Insights

- **Bias**: Shifts each neuron's activation threshold—analogous to a model's default level of "strictness" before considering inputs.
- **Learning Rate**: Controls update magnitude per error. A gentle rate (e.g., 0.1) yields smooth convergence; a harsh rate (e.g., 1.0) accelerates learning but may overshoot minima.
- **Epochs**: Full passes through the data; more epochs refine weights but risk overfitting if excessive.
- **Layers**: Depth increases representational capacity. Hidden layers enable hierarchical feature abstraction, making it possible to solve non-linear tasks (e.g., XOR, image recognition).

Together, these case studies and insights underscore how architectural choices and hyperparameter settings shape a neural network's ability to learn and generalize.“‘

Key Proverb

"Just as a master builder relies on crews to erect great structures, a neural network needs many layers to construct complex patterns from simple signals."