DL Demystified 1 - Single Neuron Fundamentals - The Perceptron

January 28, 2025

[8]:
```
################################################################
#                                                              #
#  CS435 Generative AI: Security, Ethics and Governance        #
#                                                              #
#  Instructor: Dr. Adnan Masood                                #
#  Contact:    adnanmasood@gmail.com                           #
#                                                              #
#  Notebook is MIT Licensed                                    #
################################################################
```

1 Perceptron Explained

Welcome, everyone! In this Jupyter notebook, we will explore the Perceptron algorithm in detail We will cover its **history**, the **math** behind it, **intuitive examples**, **code demonstrations**, practical problem-solving applications, and more.

Let's jump right in!

1.1 Building the Intuition

- Think of the Perceptron as a simple decision maker. It takes some numbers (like test scores), adds them up, and then decides if the result is above or below a certain mark (like a passing grade). If it's above, it says "Yes"; if it's below, it says "No."
- A Perceptron is a small machine that takes inputs (like features about something), multiplies each input by a number (called a weight), adds them up, then adds a little extra number (called bias). Finally, it passes the result through a step function which outputs either 0 or 1.
- This means the Perceptron is deciding whether something meets a condition or not, much like flipping a switch.
- A Perceptron is one of the earliest forms of a neural network unit. It takes an input vector $\mathbf{x} = (x_1, x_2, ..., x_n)$. Each input x_i is multiplied by a weight w_i, and a bias term b is added. We then apply an activation function ϕ, typically a step function:

$$\text{output} = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

- If the sum is greater than some threshold (often 0 when using a bias), the Perceptron outputs 1; otherwise, 0.
- The Perceptron is a linear binary classifier. It is trained using the Perceptron Learning Algorithm (PLA), which updates weights based on misclassifications:

  $$w_i \leftarrow w_i + \eta(t-o)x_i, \quad b \leftarrow b + \eta(t-o)$$

  where η is the learning rate, t is the target label, o is the output, and x_i is the input.
- Convergence is guaranteed if the data is linearly separable.
- The Perceptron can be seen as a foundational model for computational learning theory. Under the Probably Approximately Correct (PAC) framework, the Perceptron algorithm can learn any linearly separable function in a finite number of steps proportional to the margin and data dimension (Novikoff bound).
- Extensions of the Perceptron, such as the kernel Perceptron, leverage kernel functions to deal with non-linearly separable data in a higher-dimensional feature space. This forms the bedrock of many advanced techniques in modern machine learning.

1.2 2. Brief History and Invention

The Perceptron was invented by **Frank Rosenblatt** in 1957 at the Cornell Aeronautical Laboratory. It was inspired by the earlier work of Warren McCulloch and Walter Pitts on simplified neurons. Rosenblatt's Perceptron gained significant attention as a promising step toward "thinking machines." However, it was later criticized (notably by Minsky and Papert) for its inability to solve certain problems (like the XOR problem) due to its linear nature.

Despite setbacks, the Perceptron laid the groundwork for future developments in **artificial neural networks** and **deep learning**.

1.3 3. Underlying Technology (How It Works)

Structure: - Inputs: $x_1, x_2, ..., x_n$ - Weights: $w_1, w_2, ..., w_n$ - Bias: b - Activation function: typically a step function

Working: 1. Compute weighted sum: $z = \sum_{i=1}^{n} w_i x_i + b$. 2. Pass z through an activation function (often a Heaviside step function):

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

3. The output is a **binary** decision (0 or 1, or sometimes -1 or +1).

Intuitive Explanation: - Each weight w_i tells us how important the corresponding input x_i is. - The bias b is like a baseline that shifts the decision boundary. - If the total influence (weighted sum + bias) is large enough, the Perceptron outputs 1 (true/positive). Otherwise, it outputs 0 (false/negative).

1.4 4. Math Behind the Perceptron (ELI5)

Equation:

$$\text{Output} = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \quad \text{where} \quad \phi(z) = \begin{cases} 1 & z \geq 0, \\ 0 & z < 0. \end{cases}$$

- The weights w_i and bias b are parameters that we adjust during **training**.
- If we think about a 2D plane (just 2 inputs x_1 and x_2), the decision boundary is a line given by $w_1 x_1 + w_2 x_2 + b = 0$.
- Training involves moving this line around until the Perceptron correctly classifies the training data.

1.5 5. A Quick Illustrative Example

Let's consider a super-simple dataset: - We want the Perceptron to learn the **AND** logical function. - (0, 0) -> 0 - (0, 1) -> 0 - (1, 0) -> 0 - (1, 1) -> 1

Below is some sample code (ELI5 style) demonstrating how to train a Perceptron on this small dataset.

[9]:
```
import numpy as np

# Define our input data (X) and labels (y)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 0, 0, 1])  # AND function labels

 # Initialize weights and bias
np.random.seed(42)  # For reproducibility
w = np.random.randn(2)  # random weights
b = np.random.randn()   # random bias

learning_rate = 0.1
epochs = 10

def step_function(z):
    return 1 if z >= 0 else 0

for epoch in range(epochs):
    for i, x in enumerate(X):
        # Compute weighted sum
        z = np.dot(w, x) + b
        # Perceptron output
        prediction = step_function(z)
        # Calculate error
        error = y[i] - prediction
        # Update weights and bias
        w += learning_rate * error * x
        b += learning_rate * error
```

```python
print("Trained weights:", w)
print("Trained bias:", b)

# Test the perceptron
for i, x in enumerate(X):
    z = np.dot(w, x) + b
    prediction = step_function(z)
    print(f"Input: {x}, Predicted: {prediction}, True: {y[i]}")
```

```
Trained weights: [0.19671415 0.0617357 ]
Trained bias: -0.2523114618993075
Input: [0 0], Predicted: 0, True: 0
Input: [0 1], Predicted: 0, True: 0
Input: [1 0], Predicted: 0, True: 0
Input: [1 1], Predicted: 1, True: 1
```

1.5.1 Mock Calculation Example

Let's do a tiny mock calculation for a single input from the AND dataset, say (x = (1, 1)), with **imagined** weights and bias: - Suppose (w = (0.5, 0.5)) and (b = -0.8). - Weighted sum: (z = 0.5*1 + 0.5*1 - 0.8 = 1 - 0.8 = 0.2). - Since (z = 0.2 0), the step function returns 1. - If the correct label was 1, there's no error and no update. If the label was 0, we'd need to adjust weights.

1.6 6. Step-by-Step Perceptron Creation from Scratch

1. **Gather Data**: Create or collect labeled training examples (e.g., (features) -> (label)).
2. **Initialize Parameters**: Choose random weights and a random bias.
3. **Define an Activation Function**: Often a step function.
4. **Compute Output**: For each training example, compute $z = w \cdot x + b$. Output $\phi(z)$.
5. **Compare Output with True Label**: If there's a mismatch (error), update weights and bias:
$$w_i = w_i + \eta(t - o)x_i, \quad b = b + \eta(t - o)$$
6. **Repeat**: Keep looping over the dataset until the errors are minimized or you reach a max number of epochs.

This is the **Perceptron Learning Algorithm**!

[10]:
```python
import numpy as np

class PerceptronScratch:
    def __init__(self, learning_rate=0.1, epochs=10):
        self.lr = learning_rate
        self.epochs = epochs
        self.w = None
        self.b = None

    def step_function(self, z):
        return 1 if z >= 0 else 0

    def fit(self, X, y):
        # X is shape (num_samples, num_features)
        # y is shape (num_samples,)
        num_samples, num_features = X.shape
        # Initialize weights and bias randomly
        np.random.seed(0)
        self.w = np.random.randn(num_features)
        self.b = np.random.randn()

        for _ in range(self.epochs):
            for i in range(num_samples):
                z = np.dot(self.w, X[i]) + self.b
                prediction = self.step_function(z)
                error = y[i] - prediction
                # Update rule
                self.w += self.lr * error * X[i]
                self.b += self.lr * error

    def predict(self, X):
        z = np.dot(X, self.w) + self.b
        # Vectorized step function
        return np.array([1 if val >= 0 else 0 for val in z])

# Let's test our from-scratch Perceptron with the AND data again.
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

perceptron = PerceptronScratch(learning_rate=0.1, epochs=10)
perceptron.fit(X, y)
print("Learned weights:", perceptron.w)
print("Learned bias:", perceptron.b)

predictions = perceptron.predict(X)
print("Predictions on AND data:", predictions)
print("True labels:", y)
```

```
Learned weights: [0.86405235 0.10015721]
Learned bias: -0.7212620158942606
Predictions on AND data: [0 0 1 1]
True labels: [0 0 0 1]
```

1.7 7. Illustrative Problem It Solves

- **Logical Gates**: As demonstrated, a Perceptron can learn logical operations like AND, OR, etc.
- **Linear Classification**: For instance, determining if a point is above or below a line in 2D.

1.8 8. Practical Problem It Solves

- **Spam Detection (Simple Version)**: Suppose we have features like `contains_word_free = 1/0`, `contains_link = 1/0`, etc. A Perceptron can combine these features with weights. If the sum is above a threshold, label the email as spam.
- **Image Classification (Basic)**: If your images are small and data is linearly separable, a Perceptron can do a crude classification.

1.9 9. Solving a Real World Problem

If we had a labeled dataset of small images or textual features, the Perceptron could be quickly trained to do a binary classification (e.g., "Cat vs. Not Cat"). Of course, for complex tasks, we now use deeper neural networks, but the Perceptron is the conceptual building block behind them.

1.10 **10. Questions to Illustrate the Use

1. **What happens if the data is not linearly separable?**
2. **How does the Perceptron update its weights?**
3. **Why does the bias term matter?**
4. **What is the decision boundary for a Perceptron with two inputs?**
5. **How many epochs are enough?**
6. **Can we use a different activation function?**

1.11 11. Answers

1. **Non-linearly separable data**: The plain Perceptron can't perfectly classify data that's not linearly separable. It will keep trying to update weights forever (or until you stop). We need multiple perceptrons (multi-layer) or other methods (like kernels or advanced architectures) for non-linear boundaries.
2. **Weight update**: If an example is misclassified, weights are nudged in a direction that makes the output more correct. The update rule is $w_i = w_i + \eta(t - o)x_i$, and $b = b + \eta(t - o)$.
3. **Bias term**: The bias shifts the decision boundary away from the origin. Without bias, your decision boundary always has to pass through the origin (when all inputs are zero).
4. **Decision boundary (2 inputs)**: For x_1 and x_2, the boundary is the line $w_1x_1+w_2x_2+b = 0$.
5. **How many epochs?**: There's no fixed rule. Often, we set a maximum number of epochs. If data is linearly separable, theoretically it will converge. Practically, we stop when misclassifications are zero or below some threshold.
6. **Different activation function**: Yes! We can replace the step function with sigmoid, ReLU, or others. This transforms the Perceptron into different forms of neural network units (like logistic regression or modern neuron units).

[11]:
```python
# 12. Code Example with TODO items
# Let's create a small exercise for students:

import numpy as np

class SimplePerceptron:
    def __init__(self, lr=0.01, epochs=5):
        self.lr = lr
        self.epochs = epochs
        self.w = None
        self.b = None

    def step_function(self, z):
        return 1 if z >= 0 else 0

    def fit(self, X, y):
        # TODO 1: Initialize self.w and self.b
        # hint: use np.random.randn for random initialization
        num_features = X.shape[1]
        self.w = np.random.randn(X.shape[1])
        self.b = np.random.randn()

        for _ in range(self.epochs):
            for i, x in enumerate(X):
                # TODO 2: Compute the weighted sum z
                # z = ?
                z = np.dot(self.w, x) + self.b

                # TODO 3: Apply the step function to get the prediction
                # pred = ?
                pred = self.step_function(z)

                # TODO 4: Compute the error
                # error = ?
                error = y[i] - pred

                # TODO 5: Update the weights and bias using the perceptron rule
                # self.w += ?
                # self.b += ?
                self.w += self.lr * error * x
                self.b += self.lr * error


    def predict(self, X):
        # We'll do the step function for each input in X
        predictions = []
        for x in X:
            z = np.dot(self.w, x) + self.b
            predictions.append(self.step_function(z))
        return np.array(predictions)

# Let's create a small dataset for the exercise - the OR function.
X_todo = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y_todo = np.array([0, 1, 1, 1])  # OR function

perceptron_todo = SimplePerceptron(lr=0.1, epochs=10)

############################################
# Students will fill in the TODOs above.   #
# After filling in, they can run:          #
############################################
perceptron_todo.fit(X_todo, y_todo)
preds = perceptron_todo.predict(X_todo)
print("Predictions:", preds)
print("True labels:", y_todo)

print("TODO exercise is set up. Please open the SimplePerceptron class and fill␣
 ↪in the missing parts!")
```

```
Predictions: [0 1 1 1]
True labels: [0 1 1 1]
TODO exercise is set up. Please open the SimplePerceptron class and fill in the
missing parts!
```

1.12 13. Glossary

- **Weights ((w_i))**: The parameters that scale the importance of each input.
- **Bias ((b))**: A constant term added to shift the decision boundary.
- **Activation Function**: A function (like step, sigmoid, ReLU) that decides the output based on the weighted sum.
- **Linearly Separable**: A dataset is linearly separable if there exists a hyperplane (line in 2D, plane in 3D, etc.) that can perfectly separate the classes.
- **Epoch**: One full pass over the entire training dataset.
- **Learning Rate (( ))**: A small number controlling how much we adjust weights/bias on each error.
- **Perceptron Learning Algorithm**: The procedure to update weights/bias to reduce misclassification.
- **Decision Boundary**: In 2D, it's a line; in nD, it's a hyperplane where (w x + b = 0).
- **Neural Network**: A collection of interconnected perceptrons (neurons) in multiple layers (multi-layer perceptrons, deep networks, etc.).

Happy coding!

[12]:
```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in␣
 ↪reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-27 15:13:03.599349
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```

```
[ ]:
```