Foundations of AI from Scratch to PyTorch

A Practical Guide to Perceptrons, Backpropagation, Gradient Descent, and Neural Networks

Minor In AI, IIT Ropar
2nd May, 2025

Welcome Message

Welcome, future AI enthusiasts! This guide is designed to gently lead you through the fascinating world of perceptron learning, backpropagation, gradient descent, and neural networks. We'll be using Python and the powerful PyTorch library to make our journey practical and hands-on.

Contents

1. Predicting House Prices - A Simple Start

Imagine you want to predict the price of a house based on its area. Let's say you see a house that's 1000 square feet and you *know* (somehow!) that its *true* price should be 75 lakhs. Your goal is to build a model that can accurately predict prices based on square footage.

Where do we even begin? We'll start with an *initial guess*. Let's assume that for every square foot of area, the house costs Rs. 0.01 lakh. That's our starting point, our initial *weight*.

So, a 1000 sq ft house, according to our guess, costs $1000 \times 0.01 = 10$ lakhs! That's our *predicted price*.

But we know the real price is 75 lakhs. So we are off by quite a bit. How much are we off? We calculate the *error*. A simple way to do this is to find the squared difference between the actual price and the predicted price $(75 - 10)^2 = 4225$

Now we have a problem, an error. In the upcoming sections, we'll explore different techniques, including gradient descent and backpropagation, to minimize this error and achieve better predictions.

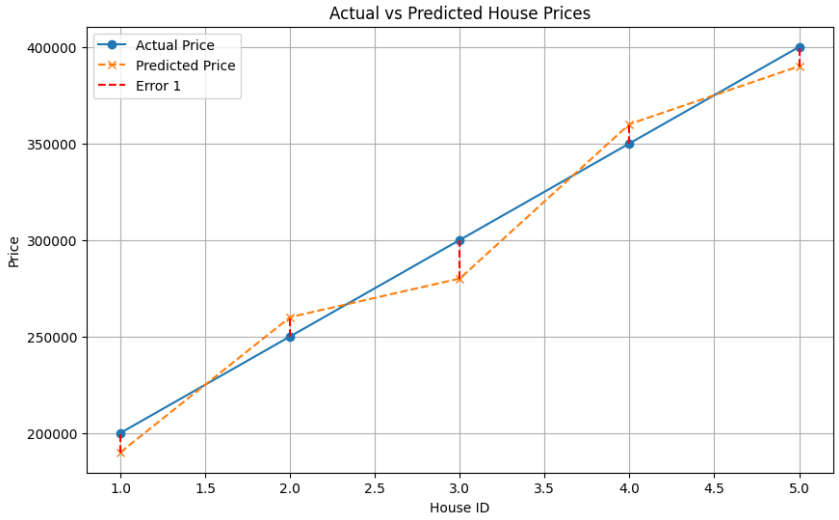

Figure 1: A simple graph showing the predicted price vs the actual price, highlighting the error.

2. The Perceptron Learning Algorithm: A Closer Look

The core idea behind the perceptron learning algorithm is to adjust the *weights* based on the error in each prediction. We saw a glimpse of this in the house price example, where we started with an initial guess for the weight and then adjusted it based on the difference between the predicted price and the actual price.

Let's revisit the "AND gate" example from the lecture. An AND gate takes two inputs (0 or 1) and outputs 1 only if *both* inputs are 1. Otherwise, it outputs 0.

Key Concept

The lecture highlighted a crucial point about how perceptron learning *really* works: **update weights after every iteration (every single input), not just after each epoch (full cycle through the data)!!!**

This is key to understanding the perceptron learning algorithm, which updates weights after every turn, not epoch.

Imagine the following scenario:

1. **Input:** 0, 0. Expected Output: 0
2. **Input:** 0, 1. Expected Output: 0
3. **Input:** 1, 0. Expected Output: 0
4. **Input:** 1, 1. Expected Output: 1

In a classical approach, you consider constant weight and bias for all, but with the perceptron approach, the next one is affected by weight changes in the previous iteration.

In the previous example, if the values change, it doesn't wait for the next Epoch, it implements it in the same sequence.

3. Linear Separability and XOR

The perceptron learning algorithm has a limitation: it can only solve problems that are *linearly separable*.

What does that even *mean*?

Think of it like this: imagine you have two groups of data points, and you want to draw a straight line to separate them. If you can draw a straight line (or a plane in higher dimensions) that perfectly divides the two groups, the data is linearly separable.

The AND gate is linearly separable. You can draw a line that separates the inputs that produce a 0 output from the input that produces a 1 output.

However, the XOR (exclusive OR) gate is **not** linearly separable. XOR outputs 1 only if *one* of the inputs is 1, but not *both*. Try as you might, you cannot draw a single straight line to separate the XOR inputs that produce 0 from those that produce 1.

If the values are linearly separable, your data is considered to be converging. If they are not, that means they're not converging.

This limitation of the perceptron led to the development of more complex models like multi-layer perceptrons, which can handle non-linearly separable data.

To deal with XOR, or other linearly-inseparable functions, there's a need to implement other equations such as Sigmoid.

4. Chain Rule and Gradient Descent

Before diving into PyTorch, we need to understand two crucial concepts: the chain rule and gradient descent.

4.1. The Chain Rule: Breaking Down Complexity

Imagine you eat a lot of burgers. This leads to weight gain. Weight gain leads to increased body fat. Increased body fat leads to health problems.

Instead of saying "eating burgers causes health problems," the chain rule helps us break down this complex relationship into a chain of simpler relationships:

- Burgers → Weight Gain
- Weight Gain → Increased Fat
- Increased Fat → Health Problems

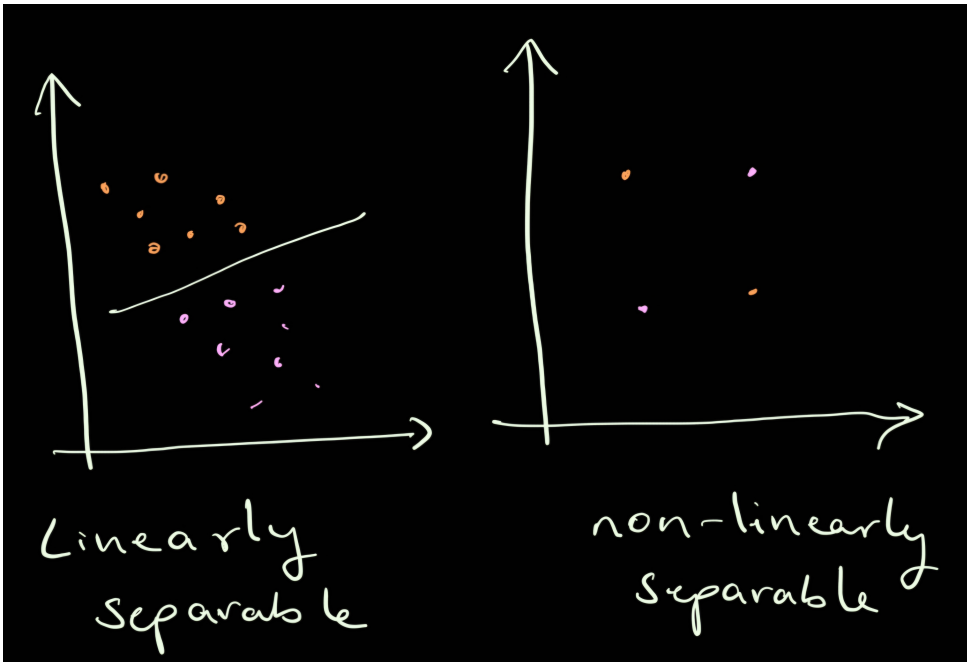

Figure 2: Diagram showing linearly separable data vs. non-linearly separable data (e.g., XOR). Illustration of drawing a line.

The chain rule helps you find the root of all evil, but breaking it down into understandable sequences.

4.2. Gradient Descent: Finding the Lowest Point

Gradient descent is an optimization algorithm used to find the minimum of a function. Think of it like this: imagine you're standing on a hill, and you want to get to the lowest point in the valley.

You can't see the entire valley, but you can feel the slope around you. Gradient descent tells you to take a step in the direction of the steepest descent (the direction where the ground is sloping downwards most rapidly). You repeat this process until you reach the bottom of the valley.

To understand the math, consider these examples:

1. $X = 2, W = 3$

2. $Y = W \times X$ (Linear Equation)

3. $Z = Y^2$ (Loss Function)

Let's use the chain rule in code to differentiate the Loss Function:

```python
# Simplified Gradient Descent Example

x = 2
w = 3
y = w * x
z = y**2

# Want to find dz/dw

# Chain Rule: dz/dw = dz/dy * dy/dw

# dz/dy = 2*y
dz_dy = 2 * y

# dy/dw = x
dy_dw = x

# dz/dw = dz/dy * dy/dw
dz_dw = dz_dy * dy_dw

print(f"dz/dw: {dz_dw}")
```

Listing 1: Simplified Gradient Descent Example

The *gradient* is nothing but the slope of the function at a given point. The *learning rate* determines the size of the step you take in the direction of the steepest descent.

You can then make code that slowly and iteratively increases the weight until you get to zero.

```python
# Simplified Gradient Descent with Weight Update

learning_rate = 0.01
w = w - learning_rate * dz_dw

print(f"Updated w: {w}")
```

Listing 2: Simplified Gradient Descent with Weight Update

5. PyTorch: Tensors and Basic Operations

PyTorch simplifies the implementation of neural networks. A key concept in PyTorch is the *tensor*. Think of a tensor as a multi-dimensional array. It's like NumPy's arrays, but with the added benefit of being able to run on GPUs for faster computation.

```python
import torch

# Create a tensor
tensor = torch.rand(3, 3) # 3x3 tensor with random values
print(tensor)
```

Listing 3: Creating a PyTorch Tensor

PyTorch supports 1D (vectors), 2D (matrices), and higher-dimensional tensors. You can perform various arithmetic operations on tensors, such as addition, subtraction, multiplication, and division.

```
import torch

# Addition
x = torch.tensor([2.0, 4.0])
y = torch.tensor([1.0, 3.0])
print(x + y) # Element-wise addition
```

Listing 4: Basic PyTorch Tensor Operations

Matrix multiplication is also straightforward:

```
import torch

A = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
B = torch.tensor([[5.0, 6.0], [7.0, 8.0]])

# Matrix multiplication
C = torch.matmul(A, B)
print(C)
```

Listing 5: Matrix Multiplication in PyTorch

6. House Price Prediction with PyTorch

Let's revisit the house price prediction example, but this time using PyTorch!

```
import torch

# Data
area = torch.tensor(1.0)
true_price = torch.tensor(3.0)

# Initialize weight with gradient tracking
weight = torch.tensor(0.1, requires_grad=True)

# Learning rate
lr = 0.01

# Model function
def model(x):
    return x * weight

# Training loop
for epoch in range(1000):
    # Forward pass
    predicted_price = model(area)
    loss = (true_price - predicted_price) ** 2

    # Backward pass
    loss.backward()

    # Update weight (without tracking gradients)
    with torch.no_grad():
```

```
        weight -= lr * weight.grad

    # Zero gradients after updating
    weight.grad.zero_()

    # Print progress
    if epoch % 100 == 0 or loss.item() < 1e-6:
        print(f"Epoch {epoch}: Weight = {weight.item():.4f},
    Predicted = {predicted_price.item():.4f}, Loss = {loss.item():.6f
    }")

    if loss.item() < 1e-6:
        print("Converged.")
        break

# Final prediction
print("\nFinal Trained Weight:", weight.item())
print("Prediction for area=2:", model(torch.tensor(2.0)).item())
```

Listing 6: House Price Prediction with PyTorch

Key points in this example:

- `requires_grad=True`: This tells PyTorch to track the gradients for this tensor. Gradients are used to update the weights during backpropagation.
- `loss.backward()`: This is the magic function! It calculates the gradients of the loss function with respect to all tensors that have `requires_grad=True`.
- `weight.grad.zero_()`: After updating the weights, it's important to reset the gradients to zero. Otherwise, they'll accumulate from previous iterations.

You are setting the true function, but you set a different scalar value for weight. requires_grad means that the backward function will be implemented during the training loop.

7. Forward and Backward Propagation

Now that we've seen PyTorch in action, let's formalize the concepts of forward and backward propagation:

Feature	Forward Propagation	Backward Propagation
Goal	Make a prediction (compute output)	Learn from the error (adjust weight
Flow Direction	Input → Output	Loss → Input
Math	Matrix Multiplications, Activation Functions	Chain Rule, Derivatives
Purpose	Check how good the model is	Adjust weights to reduce error
Code	`y = model(x)`	`loss.backward()`

Table 1: Comparison of Forward and Backward Propagation

In essence, forward propagation calculates how the model performs, and backward propagation helps the model learn from its mistakes.

The key difference is forward propagation takes the input to the output, while backward propagation takes the loss and figures out the weights.

With this foundation, you're well-equipped to explore more complex neural network architectures and deep learning concepts. This marks just the beginning of your journey. Keep experimenting, keep learning, and enjoy the exciting world of AI!