

Supervised Learning

Deep Dive

Linear & Logistic Regression

Nipun Batra | IIT Gandhinagar

Learning Goals

By the end of this lecture, you will:

1. **Understand** how linear regression finds the best line
2. **Learn** how to find optimal weights (optimization!)
3. **Apply** logistic regression for classification
4. **Connect** sklearn to PyTorch for neural networks

Recap: Supervised Learning

We have:

- Features (\mathbf{X}): What we know about each example
- Labels (\mathbf{y}): What we want to predict

Goal: Learn a function f where $f(\mathbf{X}) \approx \mathbf{y}$

If \mathbf{y} is...	Task	Example
A number	Regression	Predict house price
A category	Classification	Spam or not spam

Part 1: Linear Regression

Finding the Best Line

The Simplest Prediction Problem

Scenario: You're a real estate agent. A client asks:

"I'm looking at a 1750 sqft house. What should I expect to pay?"

You have data from recent sales:

Size (sqft)	Price (₹ lakhs)
1000	40
1500	60
2000	80
2500	100

Can you see the pattern?

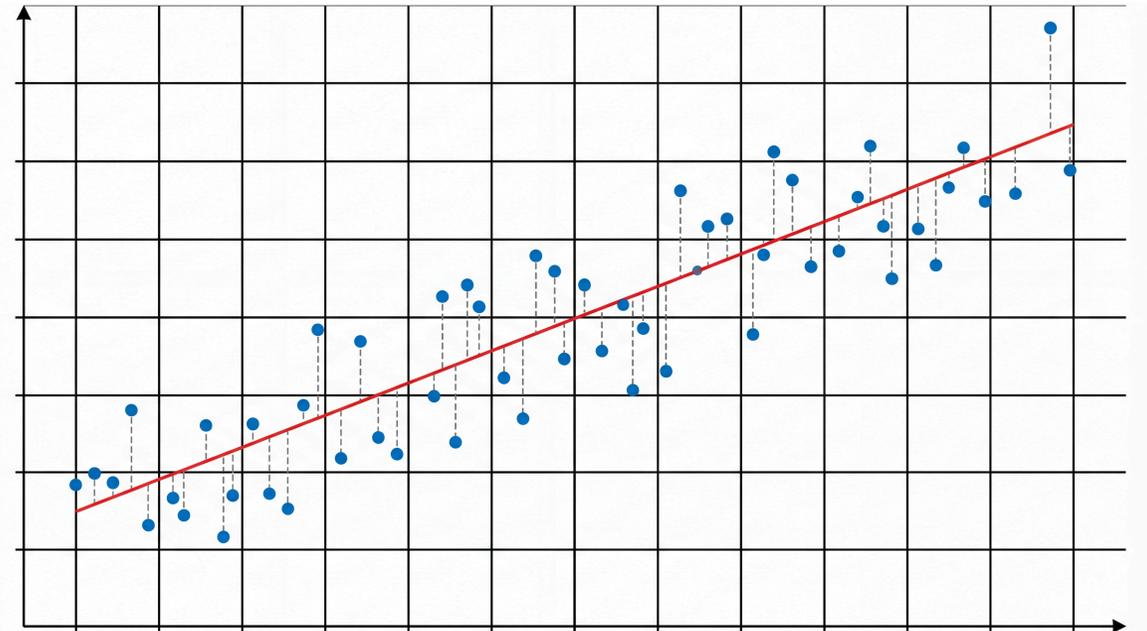
Visualizing the Data

When we plot the data:

- X-axis: House size
- Y-axis: Price

The points seem to follow a line!

Linear regression = finding the best line through the points



The Pattern is Clear!

Every 500 sqft adds ₹20 lakhs.

Size	Price	Pattern
1000	40	
1500	60	+500 sqft → +₹20 lakhs
2000	80	+500 sqft → +₹20 lakhs
2500	100	+500 sqft → +₹20 lakhs

So 1750 sqft should cost... ₹70 lakhs!

You just did linear regression in your head.

The Equation of a Line

$$\hat{y} = w \cdot x + b$$

Symbol	Name	Meaning	Our Example
x	Input	Feature value	Size (sqft)
\hat{y}	Output	Predicted value	Price
w	Weight	Slope	0.04
b	Bias	Intercept	0

The "hat" on y means it's our prediction!

What Does the Weight Mean?

Weight $w = 0.04$ means:

"For every 1 sqft increase, price increases by ₹0.04 lakhs"

Or equivalently:

"For every 100 sqft increase, price increases by ₹4 lakhs"

The weight tells you the **sensitivity** — how much does output change when input changes?

What Does the Bias Mean?

Bias $b = 0$ means:

"A 0 sqft house would cost ₹0"

In reality, bias captures the **baseline** cost:

- Land value
- Permits and fees
- Minimum construction cost

If $b = 10$, then even a tiny house costs at least ₹10 lakhs.

Multiple Features: The General Form

What if price depends on more than just size?

$$\hat{y} = w_1x_1 + w_2x_2 + \dots + w_dx_d + b$$

Or in **vector form** (both notations are equivalent):

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

Symbol	Shape	Example
\mathbf{x}	(d,)	[1500, 3, 2] — size, beds, baths
\mathbf{w}	(d,)	[0.03, 5.0, 8.0] — learned weights
b	scalar	-10

Note: $\mathbf{w}^\top \mathbf{x}$ and $\langle \mathbf{w}, \mathbf{x} \rangle$ both mean **dot product** (sum of element-wise products)

Notation: Absorbing Bias into θ

Going forward, we combine weights and bias into one vector θ :

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots = \boldsymbol{\theta}^\top \mathbf{x}$$

Trick: Add a column of 1s to \mathbf{X} , so $\theta_0 \cdot 1 = \theta_0$ (the bias)

Original \mathbf{x}	Augmented \mathbf{x}
$[x_1, x_2]$	$[1, x_1, x_2]$

Now $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \dots]^\top$ contains bias + weights!

Interpreting Multiple Weights

```
# After training on multiple features:  
# coef_ = [0.03, 5.0, 8.0]  
# intercept_ = -10
```

Feature	Weight	Interpretation
Size (sqft)	0.03	+100 sqft → +₹3 lakhs
Bedrooms	5.0	+1 bedroom → +₹5 lakhs
Bathrooms	8.0	+1 bathroom → +₹8 lakhs

Each weight shows that feature's **independent contribution** to price!

Part 2: Finding the Best Weights

The Optimization Problem

But What if Data Isn't Perfect?

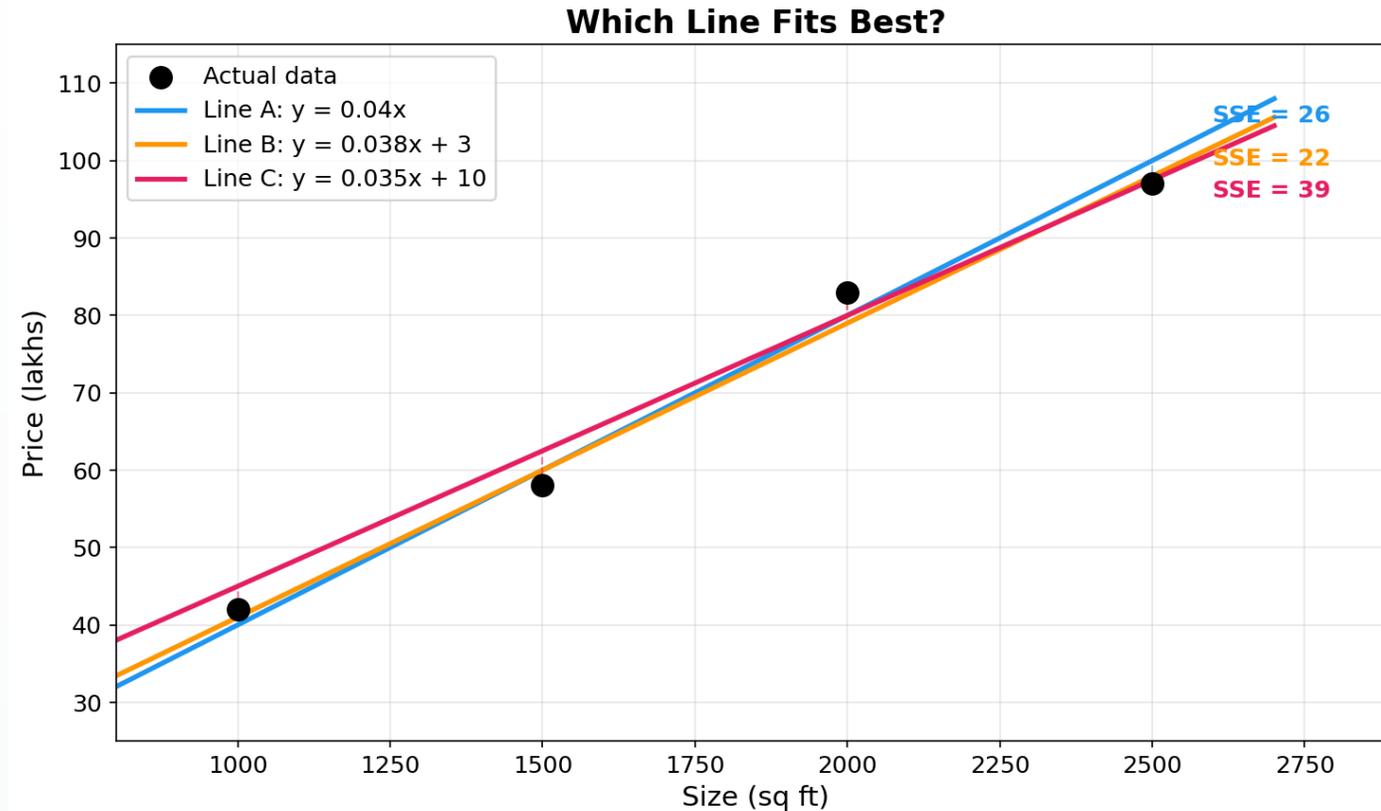
Real data has **noise** — points don't fall exactly on a line.

Three candidate lines:

Line	Equation
A	$y = 0.04x$
B	$y = 0.038x + 3$
C	$y = 0.035x + 10$

Which line is "best"?

The one with the smallest total error!



The Goal: Minimize Errors

Residual = Actual - Predicted = $y - \hat{y}$

Size	Actual	Predicted	Residual	Residual ²
1000	42	40	+2	4
1500	58	60	-2	4
2000	83	80	+3	9
2500	97	100	-3	9

Goal: Find θ that minimizes $\sum(\text{residual})^2 = 4 + 4 + 9 + 9 = 26$

Why Squared Errors?

We minimize **Sum of Squared Errors** (SSE):

$$\text{SSE} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Why Square?	Reason
Errors don't cancel	+3 and -3 both contribute positively
Penalizes big errors more	Error of 10 costs 100, not 10
Has nice math properties	Differentiable, convex

Mean Squared Error (MSE)

More commonly, we use MSE (average of squared errors):

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

This is also called the **Loss Function** or **Cost Function**:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (y_i - \boldsymbol{\theta}^\top \mathbf{x}_i)^2$$

Our goal: Find $\boldsymbol{\theta}$ that minimizes \mathcal{L}

Two Ways to Find the Best Weights

Method	How It Works	When to Use
Normal Equation	Direct formula, one-shot	Small datasets
Gradient Descent	Iterative, step-by-step	Large datasets, neural nets

Let's learn both!

Quick Review: Derivatives and Gradients

Concept	What it means	Example
Derivative	Rate of change (1 variable)	$f(x) = x^2 \Rightarrow \frac{df}{dx} = 2x$
Partial Derivative	Rate of change w.r.t. one variable (others fixed)	$f(x, y) = x^2 + y^2 \Rightarrow \frac{\partial f}{\partial x} = 2x$
Gradient	Vector of all partial derivatives	$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right] = [2x, 2y]$

Key insight: Gradient points in direction of steepest **increase**. To minimize, go **opposite** to gradient!

Gradient Example: $f(x, y) = x^2 + y^2$

At point $(1, 2)$:

$$\nabla f = [2x, 2y] = [2, 4]$$

Interpretation:

- Moving in $+x$ direction increases f at rate 2
- Moving in $+y$ direction increases f at rate 4
- To **decrease** f , move in direction $[-2, -4]$

Setting $\nabla f = 0$: Gives us $x = 0, y = 0$ — the minimum!

Can We Solve It Directly?

For simple functions like $f(x) = x^2$, we know the trick:

Step	What we do	Result
1. Write the function	$f(x) = x^2$	
2. Take derivative	$f'(x) = 2x$	
3. Set = 0	$2x = 0$	
4. Solve	$x = 0$	Minimum!

Can we do the same for linear regression?

Yes! Our loss is $\mathcal{L}(\boldsymbol{\theta}) = \sum (y_i - \mathbf{x}_i \cdot \boldsymbol{\theta})^2$

Take derivative, set to 0, solve...

The Normal Equation

Setting $\nabla \mathcal{L} = 0$ and solving gives us:

$$\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

How to read this: "The best parameters $\hat{\theta}$ = some matrix math on X and y"

```
import numpy as np

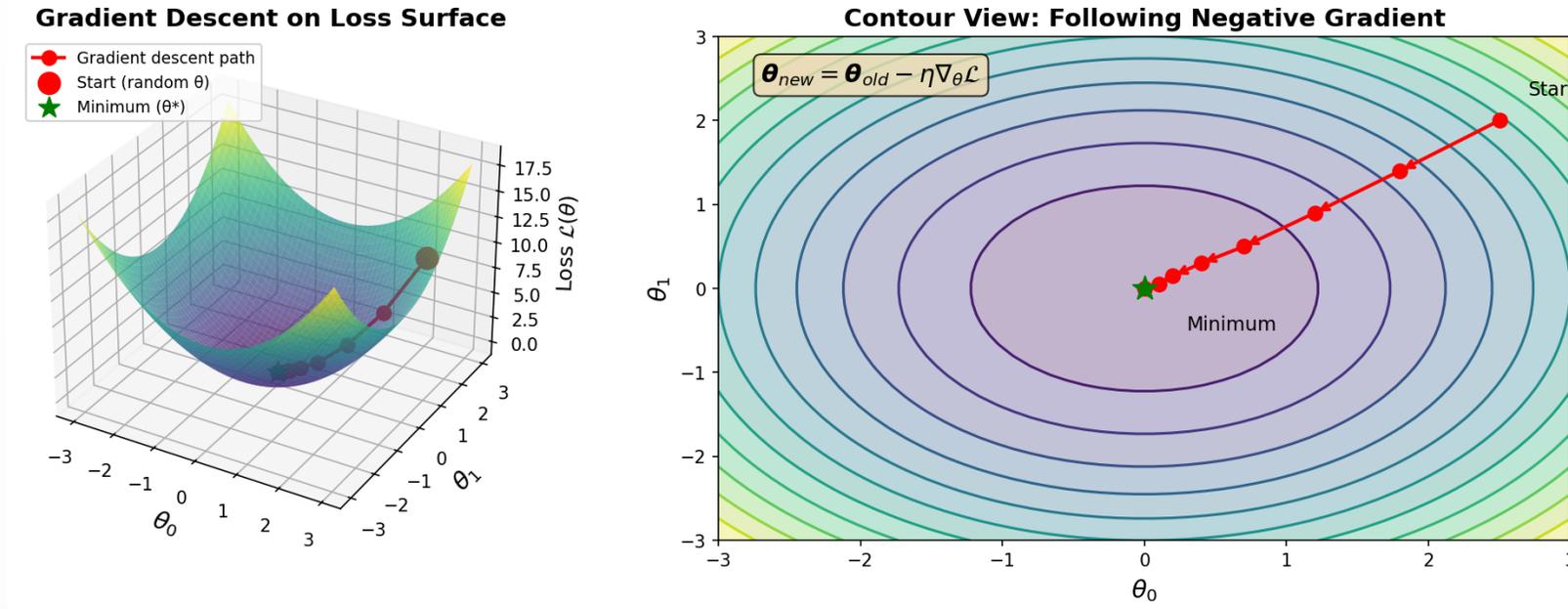
X = np.array([[1, 1000], [1, 1500], [1, 2000], [1, 2500]]) # column of 1s for bias
y = np.array([42, 58, 83, 97])

theta = np.linalg.inv(X.T @ X) @ X.T @ y
print(f"bias = {theta[0]:.1f}, weight = {theta[1]:.4f}")
# bias = 1.0, weight = 0.038 → Line B from our plot!
```

Limitation: Requires matrix inversion — too slow for millions of features. We need gradient descent!

Gradient Descent

The idea: Take small steps downhill until you reach the minimum!



1. Start with random θ → 2. Compute gradient → 3. Step opposite → 4. Repeat!

Gradient Descent: The Algorithm

Update rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_{\theta} \mathcal{L}$$

Symbol	Name	Meaning
η	Learning rate	How big each step is
$\nabla_{\theta} \mathcal{L}$	Gradient	Direction of steepest increase
$-\nabla_{\theta} \mathcal{L}$		Direction of steepest decrease

Gradient Descent: A Worked Example

Let's walk through ONE step:

Current	Value
θ (weight)	0.01
Loss	150
Gradient	-80
Learning rate η	0.001

The update:

$$\theta_{\text{new}} = 0.01 - 0.001 \times (-80) = 0.01 + 0.08 = 0.09$$

Gradient was negative → we moved weight UP!

The Gradient for MSE: Intuition

How should we nudge θ to reduce the error?

Think about one data point (x_i, y_i) with prediction \hat{y}_i :

Situation	Error	What to do with θ ?
Predicted too low	$y_i - \hat{y}_i > 0$	Increase θ (to predict higher)
Predicted too high	$y_i - \hat{y}_i < 0$	Decrease θ (to predict lower)
Predicted correctly	$y_i - \hat{y}_i = 0$	Don't change!

The gradient captures exactly this: **error** \times **feature value**

$$\nabla_{\theta} \mathcal{L} = -\frac{2}{n} \mathbf{X}^{\top} (\mathbf{y} - \hat{\mathbf{y}})$$

Big error + big feature = big update. Zero error = no update.

Gradient Descent in NumPy

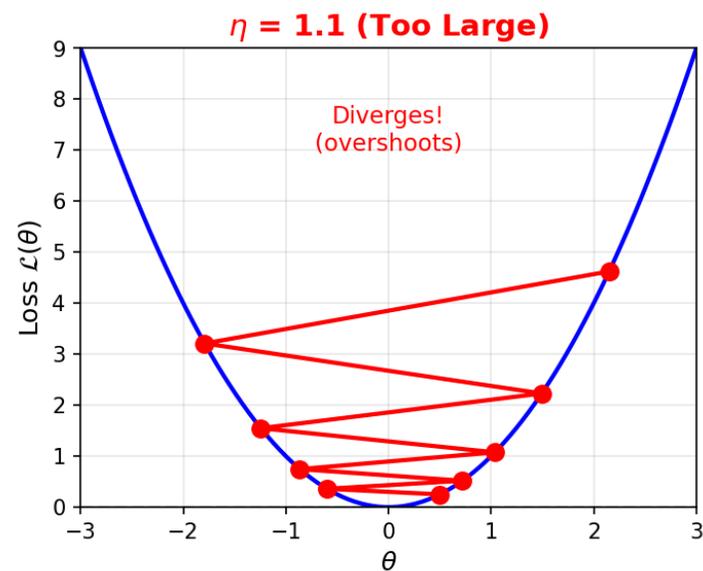
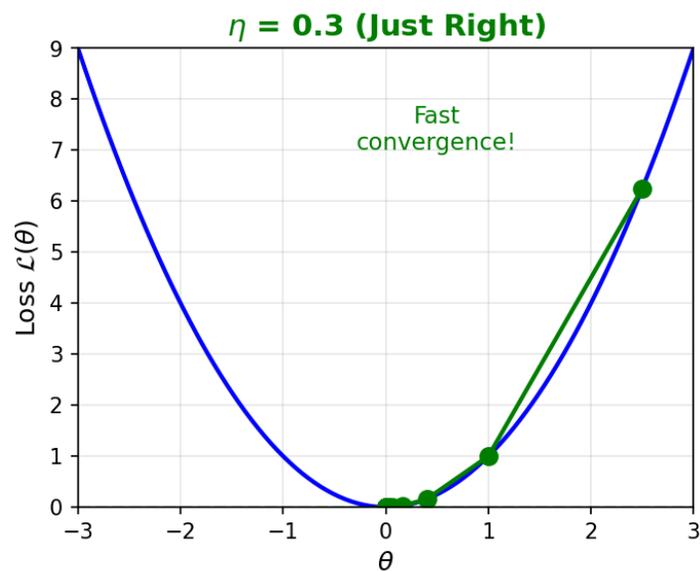
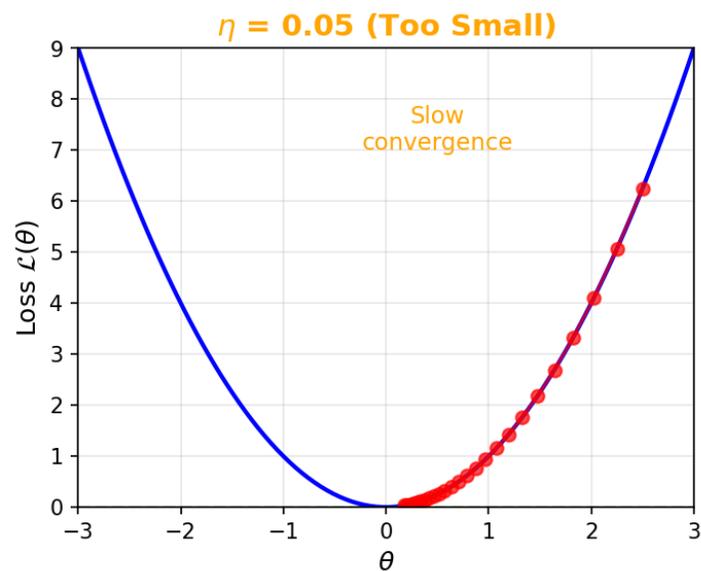
```
def gradient_descent(X, y, lr=0.01, epochs=1000):
    theta = np.zeros(X.shape[1]) # Start with zeros

    for epoch in range(epochs):
        y_pred = X @ theta          # Predictions
        error = y - y_pred         # Residuals
        gradient = (-2/len(y)) * (X.T @ error)
        theta = theta - lr * gradient # Update!

    return theta
```

Just 8 lines of code! This is all of gradient descent.

Learning Rate: The Key Hyperparameter



Too small ($\eta = 0.05$)	Just right ($\eta = 0.3$)	Too large ($\eta = 1.1$)
Slow convergence	Fast convergence ✓	Diverges!

Learning Rate: The Hill Analogy

Imagine walking down a hill blindfolded:

Learning Rate	What Happens
Tiny steps (0.0001)	Safe, but takes forever to reach bottom
Normal steps (0.01)	Good progress, reach bottom reasonably
Giant leaps (1.0)	Overshoot, end up on the other side!

How to choose?

- Start with 0.001 or 0.01
- If loss doesn't decrease → try smaller
- If loss explodes (NaN) → definitely smaller!

Why Gradient Descent Matters

Normal Equation	Gradient Descent
One-shot computation	Iterative process
Exact solution	Approximate (but close enough)
$O(n^3)$ complexity	$O(n)$ per iteration
Only works for linear models	Works for ANY differentiable model!

This is the foundation of neural network training!

Part 3: Feature Scaling

Why Scale Matters

The Scaling Problem

Different features have different scales:

Feature	Range	Scale
House size	500 - 5000 sqft	~1000s
Bedrooms	1 - 6	~1s
Age	0 - 100 years	~10s

Problem: Large-scale features dominate gradient descent!

Why Scaling Helps Gradient Descent

Without scaling:

- Size weight needs tiny updates (large values)
- Bedroom weight needs large updates (small values)
- Gradient descent zigzags inefficiently!

With scaling:

- All features contribute equally
- Gradient descent converges faster
- More stable training

Scaling: The Currency Analogy

Imagine training with mixed currencies:

Feature	Value	Scale
Price in rupees	50,00,000	Millions
Number of rooms	3	Single digits

Without scaling: The model thinks rupees matter MORE (bigger numbers!).

Standardization: Convert everything to "standard units"

- "This house is 2 std devs above average in price"
- "This house has 1 std dev above average rooms"

Now both features speak the same language!

Two Common Scaling Methods

Method	Formula	Result
Standardization	$\frac{x-\mu}{\sigma}$	Mean=0, Std=1
Min - Max Scaling	$\frac{x-x_{min}}{x_{max}-x_{min}}$	Range [0, 1]

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Standardization (most common)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Min-Max (when you need bounded range)
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Important: Fit on Train, Transform Both!

```
# CORRECT way:  
scaler = StandardScaler()  
X_train_scaled = scaler.fit_transform(X_train) # Fit AND transform  
X_test_scaled = scaler.transform(X_test)      # Only transform!  
  
# WRONG (data leakage!):  
X_scaled = scaler.fit_transform(X) # Fitting on all data
```

Never fit the scaler on test data! It would leak information.

When to Scale?

Algorithm	Needs Scaling?	Why
Linear/Logistic Regression	Yes	Gradient descent
Neural Networks	Yes	Gradient descent
Decision Trees	No	Split-based, scale-invariant
K-Nearest Neighbors	Yes	Distance-based
Random Forest	No	Tree-based

Part 4: From sklearn to PyTorch

Building the Bridge

Linear Regression in sklearn

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Our data
X = np.array([[1000], [1500], [2000], [2500]])
y = np.array([40, 60, 80, 100])

# Create and train model
model = LinearRegression()
model.fit(X, y)

# Now predict!
model.predict([[1750]]) # → 70.0 (₹70 lakhs)
```

Understanding What sklearn Learned

```
print(f"Weight (w): {model.coef_[0]}")      # 0.04  
print(f"Intercept (b): {model.intercept_}") # 0.0
```

The equation it learned:

$$\text{Price} = 0.04 \times \text{Size} + 0$$

Verify:

- 1750 sqft $\rightarrow 0.04 \times 1750 + 0 =$ ₹70 lakhs

The Same Thing in PyTorch!

```
import torch
import torch.nn as nn

# Data as tensors
X = torch.tensor([[1000.], [1500.], [2000.], [2500.]])
y = torch.tensor([[40.], [60.], [80.], [100.]])

# Normalize for stable training
X_norm = X / 1000

# Linear model:  $y = wx + b$ 
model = nn.Linear(1, 1) # 1 input, 1 output
```

Training in PyTorch

```

criterion = nn.MSELoss() # Loss function
optimizer = torch.optim.SGD(model.parameters(), lr=0.1) # Optimizer

for epoch in range(100):
    y_pred = model(X_norm) # 1. Forward pass
    loss = criterion(y_pred, y) # 2. Compute loss
    optimizer.zero_grad() # 3. Clear gradients
    loss.backward() # 4. Compute gradients
    optimizer.step() # 5. Update weights

```

After training: `model.weight` ≈ 0.04 , `model.bias` ≈ 0 (same as sklearn!)

The PyTorch Training Loop

The 5-step training cycle (memorize this!):

Step	Code	What it does
1	<code>y_pred = model(X)</code>	Forward pass: compute predictions
2	<code>loss = criterion(y_pred, y)</code>	Measure error
3	<code>optimizer.zero_grad()</code>	Clear old gradients
4	<code>loss.backward()</code>	Compute new gradients
5	<code>optimizer.step()</code>	Update θ using gradients

This exact loop works for ANY neural network - from linear regression to GPT!

Comparing sklearn vs PyTorch

Aspect	sklearn	PyTorch
Simplicity	2 lines of code	10+ lines
Method	Closed-form (SVD)	Gradient descent
Customization	Limited	Full control
Neural nets	Basic only	Full support
GPU support	No	Yes!

Start with sklearn, move to PyTorch when you need more power!

Part 4: Logistic Regression

From Numbers to Categories

A Different Problem

Scenario: You're building a spam filter.

Email	Exclamation marks	Has "FREE"	Is Spam?
1	5	Yes	Spam
2	0	No	Not Spam
3	3	Yes	Spam
4	1	No	Not Spam

The output is a category, not a number!

Why Can't We Use Linear Regression?

If we use linear regression:

$$\text{Spam Score} = \theta_1 \times \text{Exclamations} + \theta_2 \times \text{HasFREE} + \theta_0$$

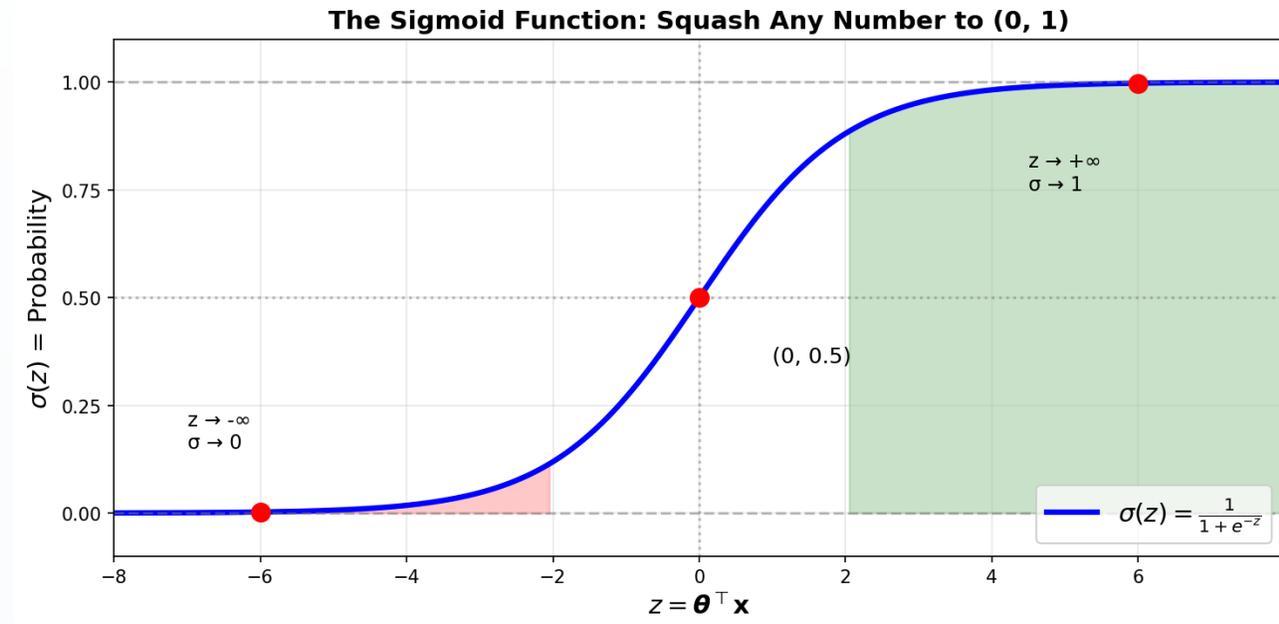
Problem: This gives any number ($-\infty$ to $+\infty$)

Score	What does it mean?
-2.5	???
0.3	???
1.5	???
147	???

We need something between 0 and 1 (a probability)!

The Sigmoid Function

Solution: Squash any number to range (0, 1)



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

The Sigmoid Shape

The sigmoid is an **S-curve**:

Region	Behavior
Very negative z	Output ≈ 0
z near 0	Output changes rapidly (decision boundary)
Very positive z	Output ≈ 1

Key insight: It converts any number to a probability!

Why Sigmoid? Let's Verify!

Plug in some numbers:

Linear score z	$\sigma(z) = \frac{1}{1+e^{-z}}$	Meaning
$z = -5$	$\frac{1}{1+148.4} = 0.007$	~0% chance
$z = 0$	$\frac{1}{1+1} = 0.5$	50/50
$z = +5$	$\frac{1}{1+0.007} = 0.993$	~100% chance

No matter what z is, output is always between 0 and 1!

Logistic Regression Model

Two steps:

1. **Linear:** Compute a score (same as linear regression!)

$$z = \theta_1 x_1 + \theta_2 x_2 + \theta_0 = \boldsymbol{\theta}^\top \mathbf{x}$$

2. **Sigmoid:** Convert to probability

$$P(\text{spam}) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

A Concrete Example

Email features: 5 exclamation marks, has "FREE" (=1)

Learned weights: $w_1 = 0.5$, $w_2 = 2.0$, $b = -1.0$

Step 1: Linear score

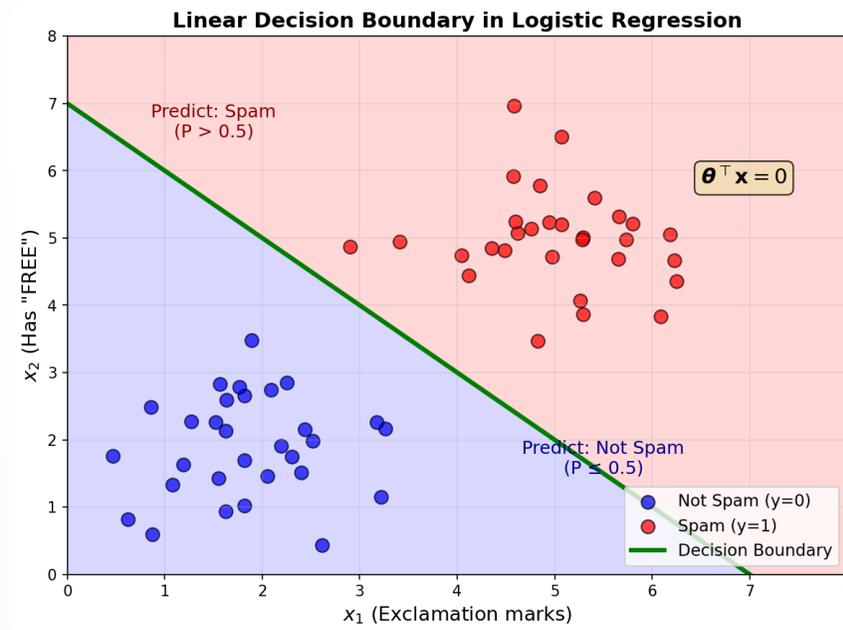
$$z = 0.5 \times 5 + 2.0 \times 1 + (-1.0) = 3.5$$

Step 2: Sigmoid

$$P(\text{spam}) = \sigma(3.5) = \frac{1}{1 + e^{-3.5}} = 0.97$$

Decision: 97% → This is spam!

The Decision Rule



If $P(\text{spam})$	Decision	Threshold can be tuned!
> 0.5	Predict SPAM	Lower \rightarrow catch more spam
≤ 0.5	Predict NOT SPAM	Higher \rightarrow fewer false alarms

Logistic Regression in sklearn

```
from sklearn.linear_model import LogisticRegression

X = [[5, 1], [0, 0], [3, 1], [1, 0]] # [exclamations, has_FREE]
y = [1, 0, 1, 0]                    # 1=spam, 0=not spam

model = LogisticRegression()
model.fit(X, y)

model.predict([[4, 1]])              # → [1] (spam)
model.predict_proba([[4, 1]])       # → [[0.12, 0.88]] = [P(not spam), P(spam)]
```

Logistic Regression in PyTorch

```
import torch
import torch.nn as nn

# Model: Linear + Sigmoid
class LogisticRegression(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))

model = LogisticRegression(input_dim=2)
```

Training Logistic Regression

```
# Binary Cross-Entropy Loss (for classification)
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(100):
    # Forward pass
    y_pred = model(X)

    # Compute loss (cross-entropy, not MSE!)
    loss = criterion(y_pred, y)

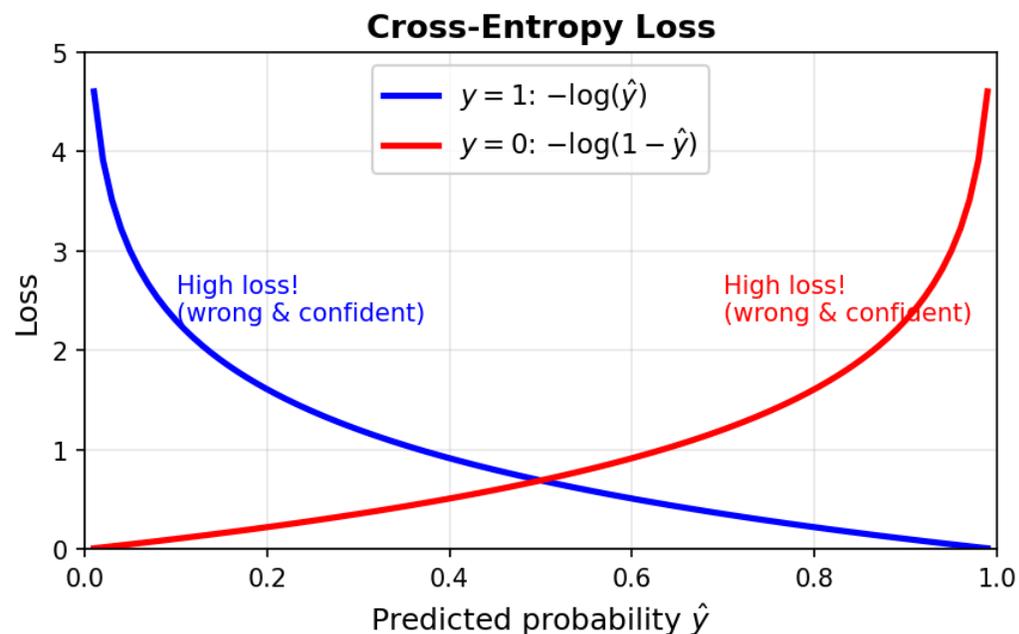
    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Why Cross-Entropy Loss?

For classification, MSE doesn't work well — Cross-Entropy penalizes **confident wrong predictions** severely!

$$\mathcal{L} = -\frac{1}{n} \sum [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Cross-Entropy: The 4 Cases



The 4 Cases: Confident & Wrong = High Loss!

Case	True (y)	Pred (\hat{y})	Loss	Interpretation
✓ Correct	1 (Spam)	0.95	0.05	Low loss - good!
✓ Correct	0 (Not)	0.05	0.05	Low loss - good!
✗ Wrong	1 (Spam)	0.10	2.30	High loss - BAD!
✗ Wrong	0 (Not)	0.90	2.30	High loss - BAD!

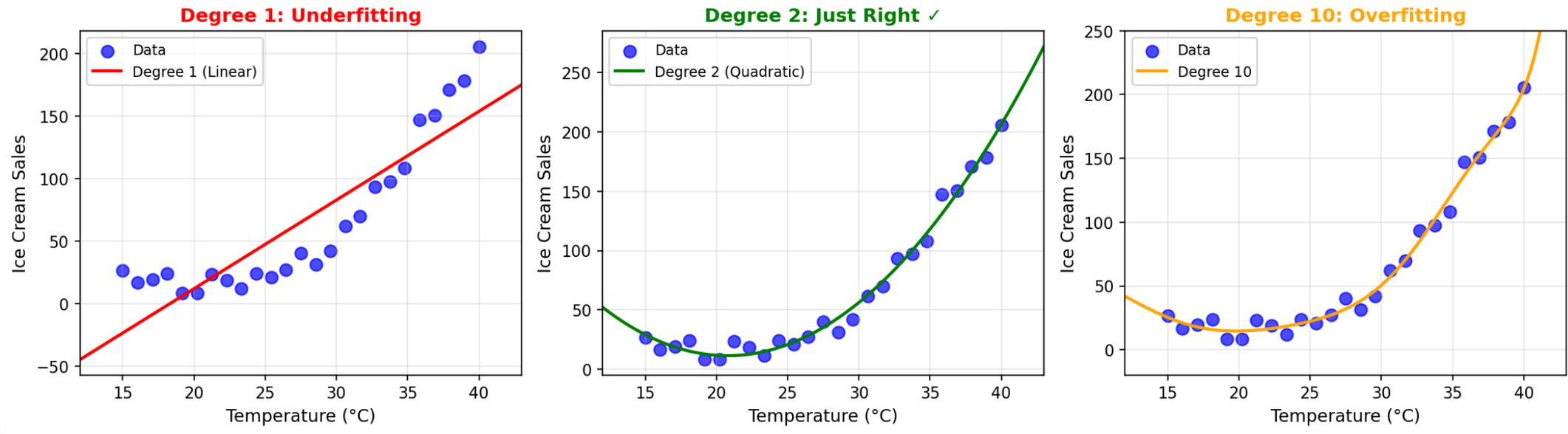
Key insight: Being confident AND wrong = very high loss!

Part 5: Feature Engineering

Making Linear Models More Powerful

The Limitation of Linear Models

Problem: What if the relationship isn't linear?



Example: Ice cream sales vs temperature — clearly not a straight line!

Solution: Transform the inputs using **basis functions**

Basis Functions: The Key Idea

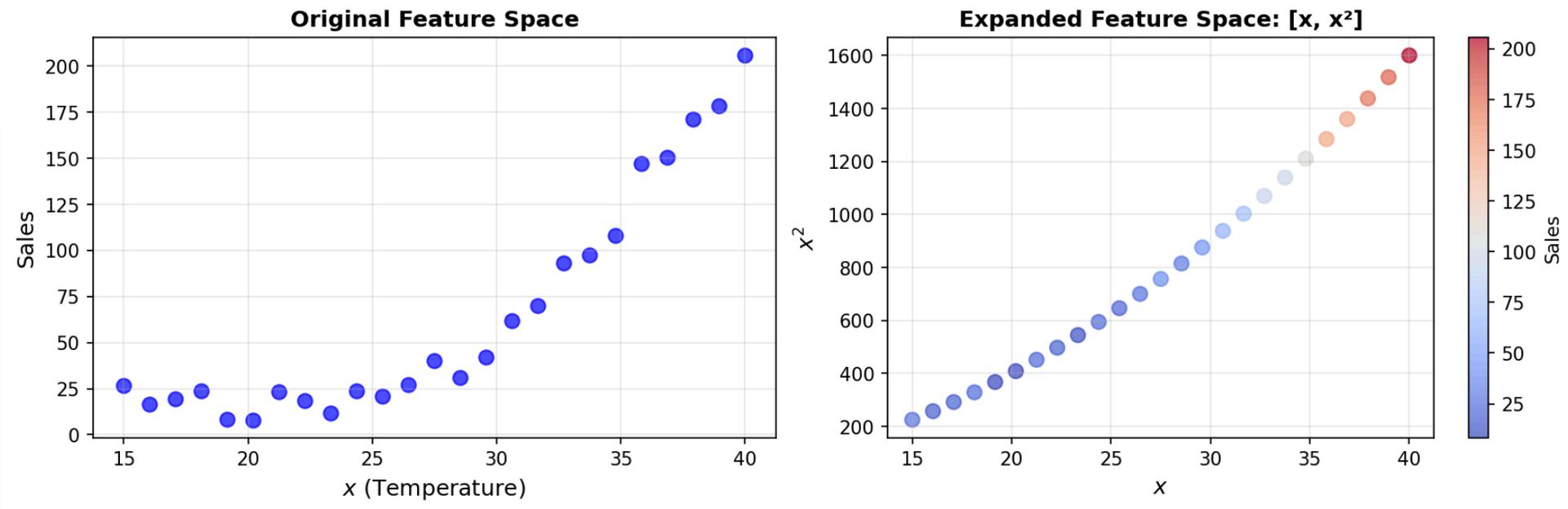
Instead of: $\hat{y} = \theta_0 + \theta_1 x$

Use: $\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2$

Original Feature	Basis - Expanded Features
x	$[1, x, x^2]$
x	$[1, x, x^2, x^3]$
x	$[1, \sin(x), \cos(x)]$

The model is still linear in θ ! (just not in x)

Feature Expansion Visualized



By adding x^2 as a feature, we give the model more "raw material" to work with!

Polynomial Features in sklearn

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

X = np.array([[15], [20], [25], [30], [35]]) # Temperature
y = np.array([15, 10, 20, 55, 120]) # Ice cream sales

poly = PolynomialFeatures(degree=2) # Transform x → [1, x, x2]
X_poly = poly.fit_transform(X)

model = LinearRegression()
model.fit(X_poly, y) # Now it can fit curves!
```

Works for Classification Too!

Same idea with Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

model = LogisticRegression()
model.fit(X_poly, y) # Now decision boundary can be curved!
```

Key insight: Basis functions let linear models learn nonlinear patterns!

Summary: The Big Picture

Concept	Key Takeaway
Linear Regression	$\hat{y} = \boldsymbol{\theta}^\top \mathbf{x}$
Loss Function	MSE measures how wrong we are
Gradient Descent	$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla \mathcal{L}$
Logistic Regression	Linear + Sigmoid for classification
Cross-Entropy	Loss for classification
Basis Functions	Transform inputs for nonlinear patterns

Key Takeaways

1. **Linear Regression** fits a line through data

- Weight = sensitivity (how much output changes per input)
- Minimize squared errors

2. **Two ways to find optimal weights**

- Normal equation (direct)
- Gradient descent (iterative) — **foundation of deep learning!**

3. **Logistic Regression** classifies using the sigmoid

- Converts any score to probability (0-1)

4. **sklearn** → **PyTorch** uses the same concepts!

You Now Understand the Basics!

What You Learned	Why It Matters
Linear Regression	Foundation of all neural networks
Gradient Descent	How we train ANY model
Logistic Regression	Classification with probabilities
Basis Functions	Make linear models powerful

The training loop you learned today is the SAME loop used to train GPT-4!

Next: Model Selection & Evaluation — How do we know if our model is good?