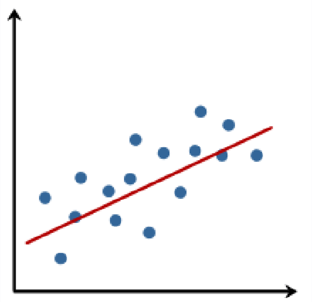
The story we will complete: Cab Fare Prediction

- Create numeric data
- Train model (w, b) to reduce loss
- Test Performance

If we make error/loss go down and test looks reasonable, we succeeded.

Session Outcomes

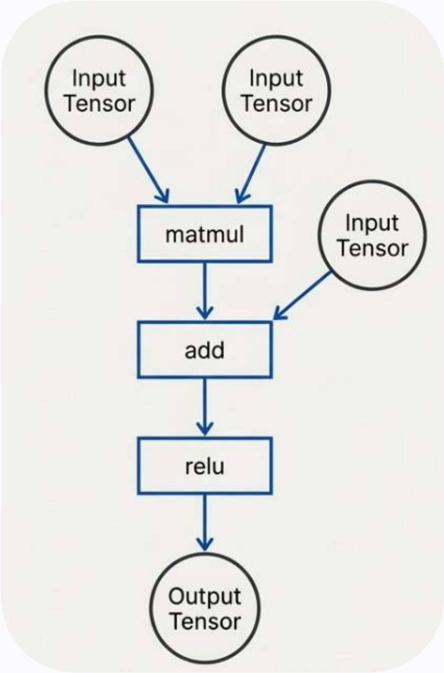
Build a Simple Linear Regression Model w/ PyTorch



$$y = Xw + b$$

- Tensors: The core data structure
- Matmul & Broadcasting: The mechanics of the forward pass
- Autograd: The engine for learning
- Training & Testing: The complete model lifecycle

PyTorch Mental Model



The Static Computation Graph (blueprint)

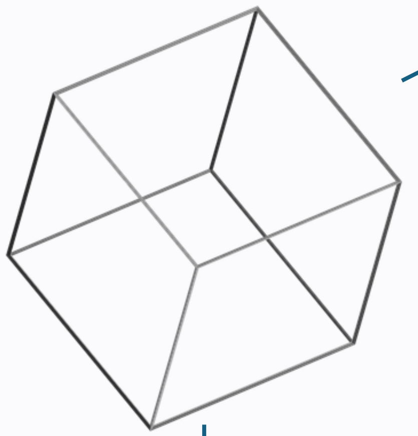
Build Graph First, Run Later

- Fast!
- Cannot handle conditions
- Difficult to Debug

- **Eager Execution**: ops run immediately (no “compile step”)
- **Debugging = inspection**: print **shape / dtype / device** early
- **Core Primitives**: **Tensor + Autograd** → everything else builds on this

The Core Primitive: Tensors

- A Tensor is more than just data, it data + critical metadata that governs all operations



When something breaks, the first and fastest diagnostic is to print these three things:

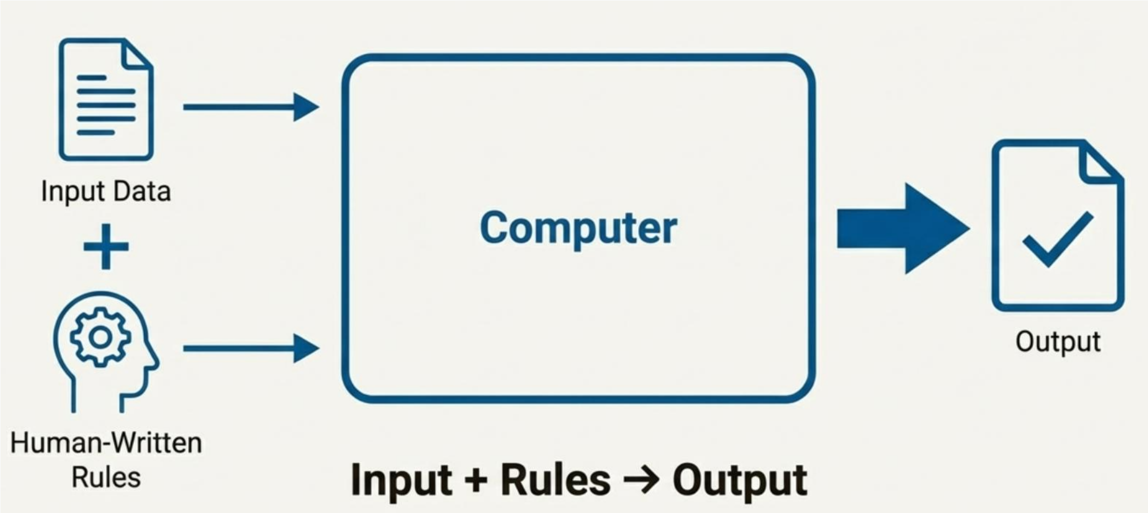
1. x.shape
2. x.dtype
3. x.device

Lab: Tensors

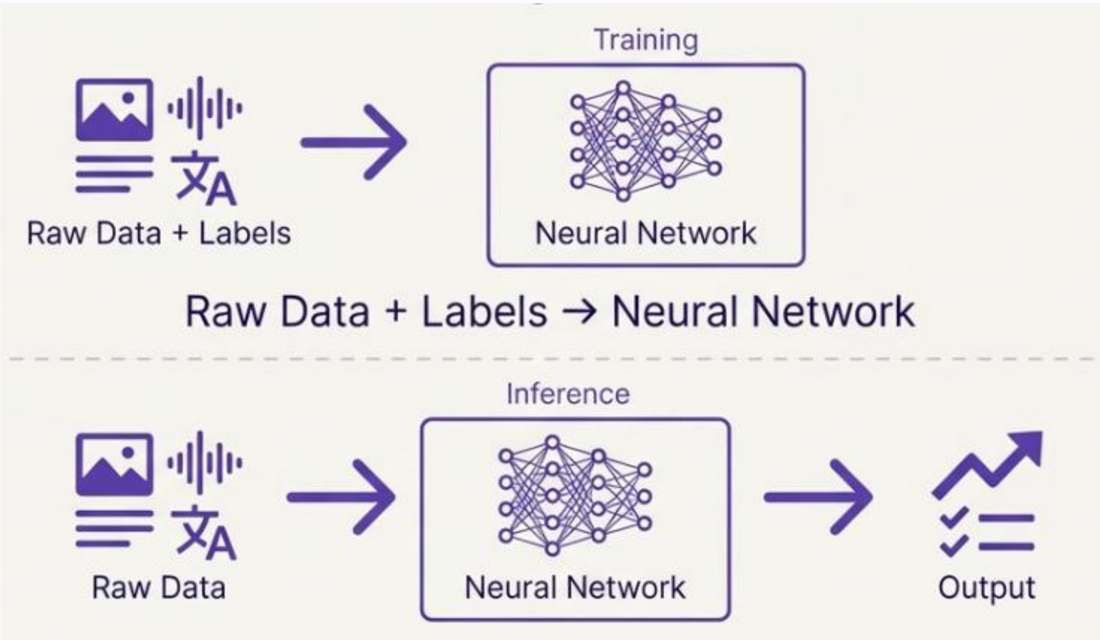
Lab: Toy Dataset

Revisiting Machine Learning and Deep Learning

Traditional Programming



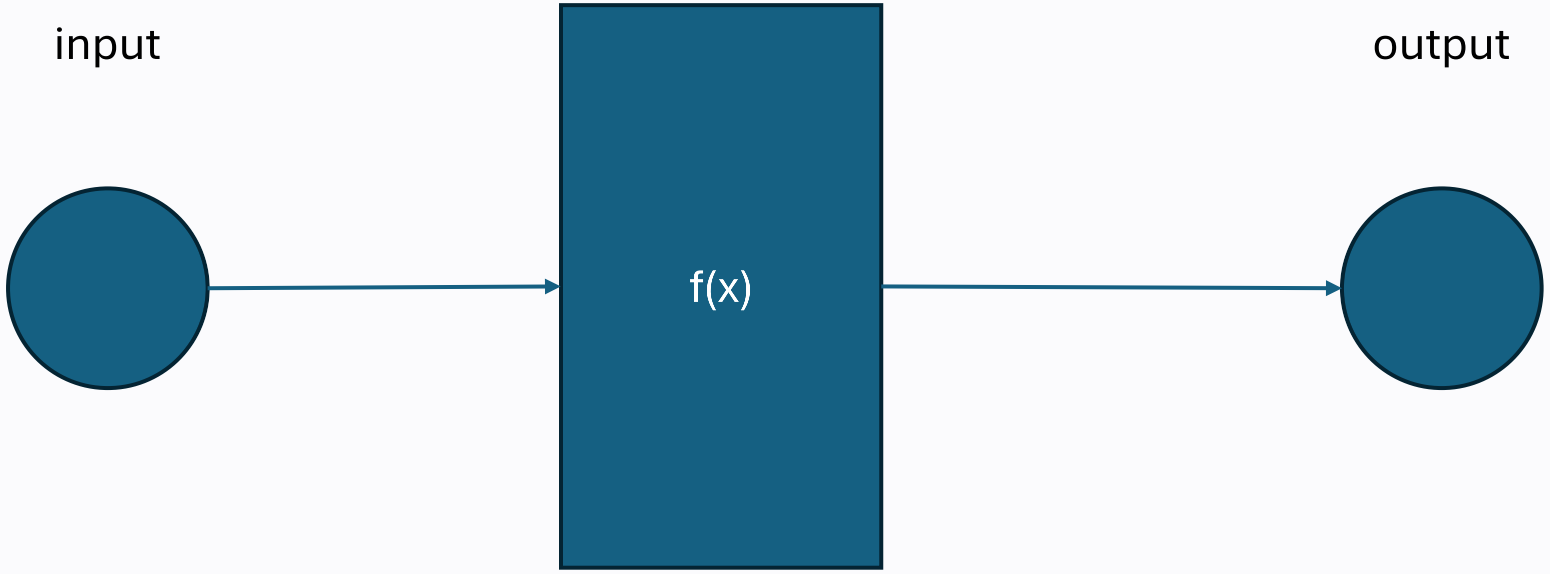
Deep Learning



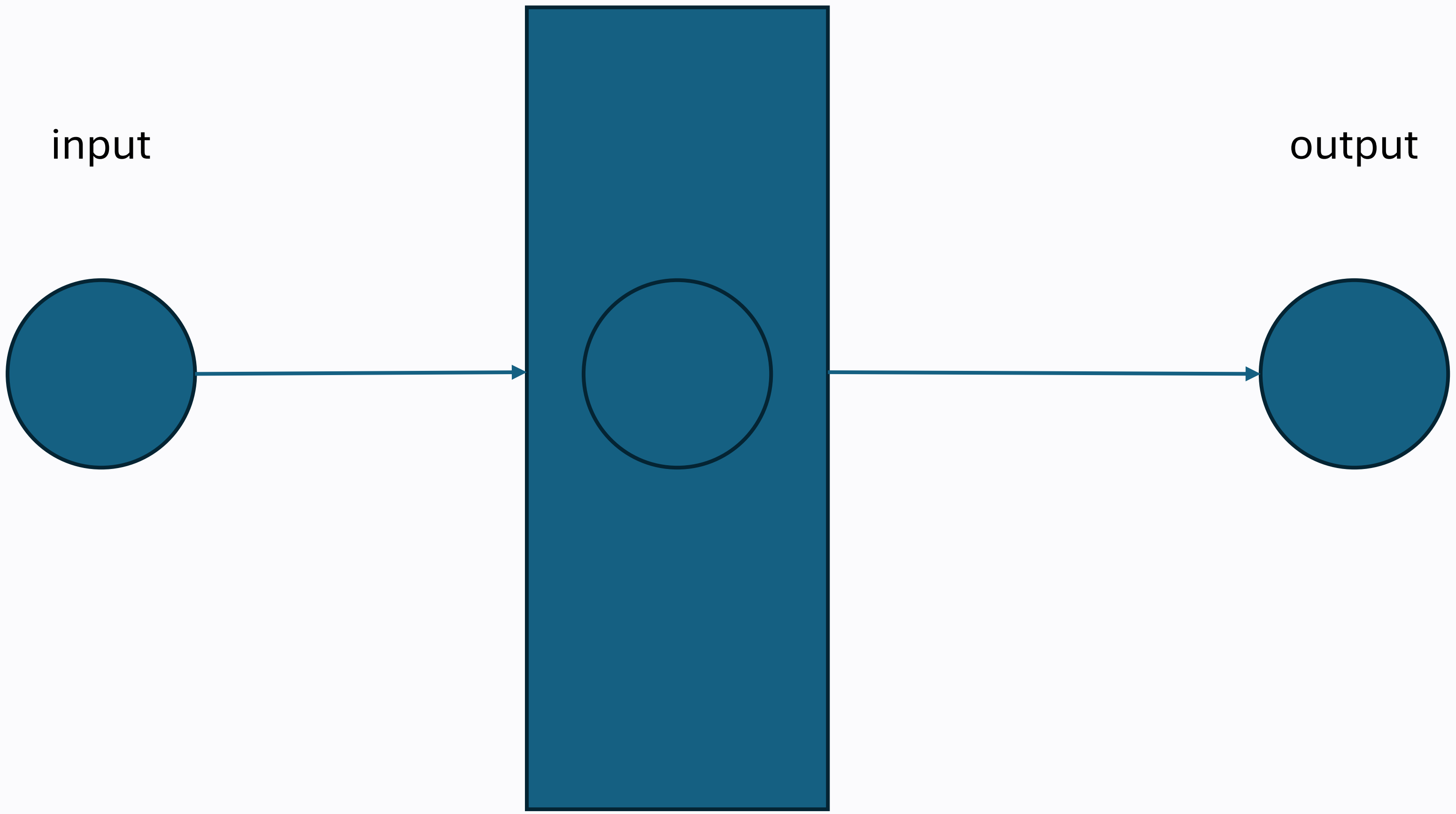
Comparison

	Traditional Programming	Machine Learning	Deep Learning
Rules	Written by humans	Learned from data	Learned from data
Features	Explicit logic	Manually engineered	Automatically learned
Data Need	Low	Medium	High
Interpretability	Very High	Medium	Low
Compute Cost	Low	Medium	High
Best For	Clear logic	Structured data	Unstructured data

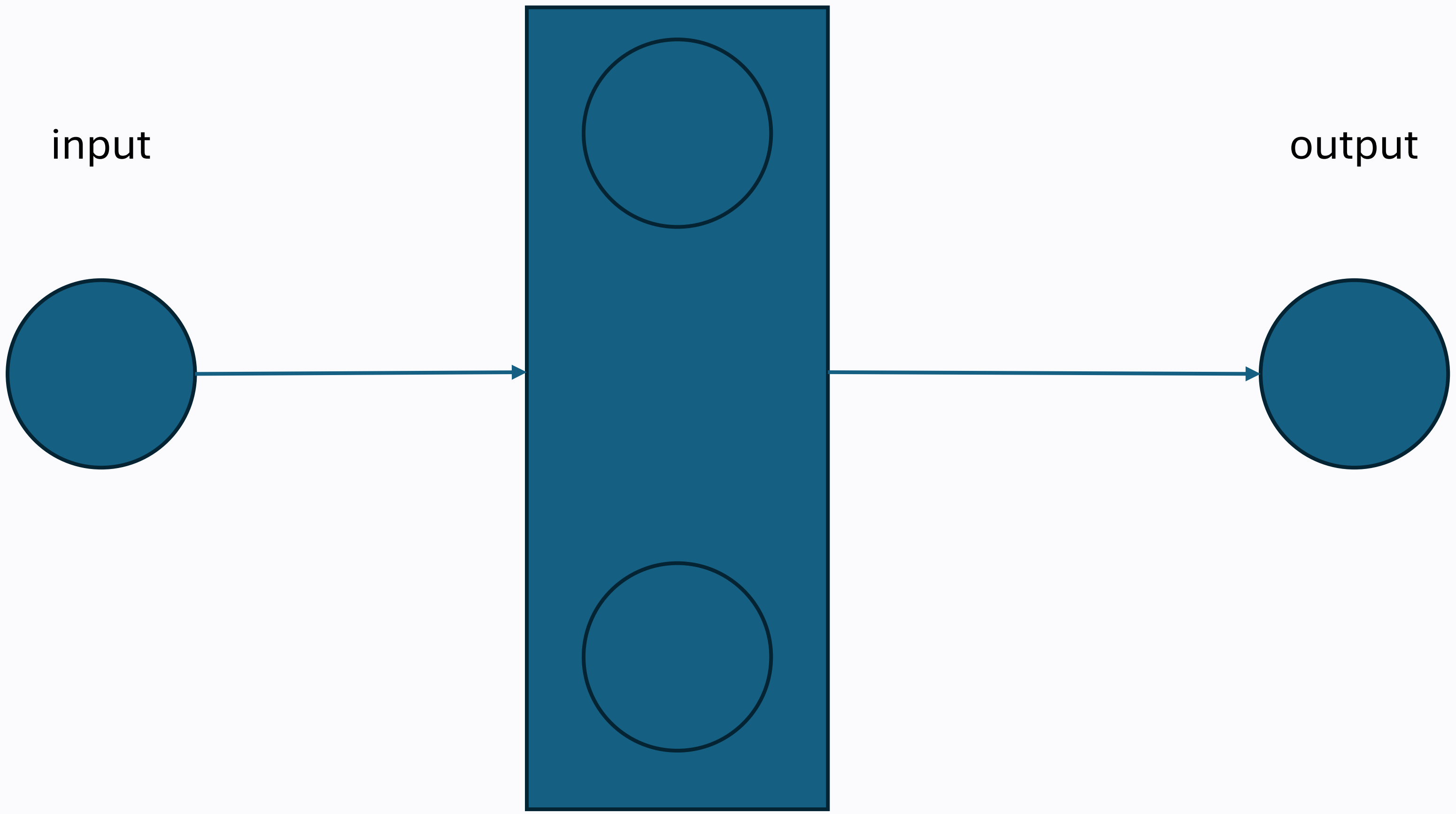
Neural Networks



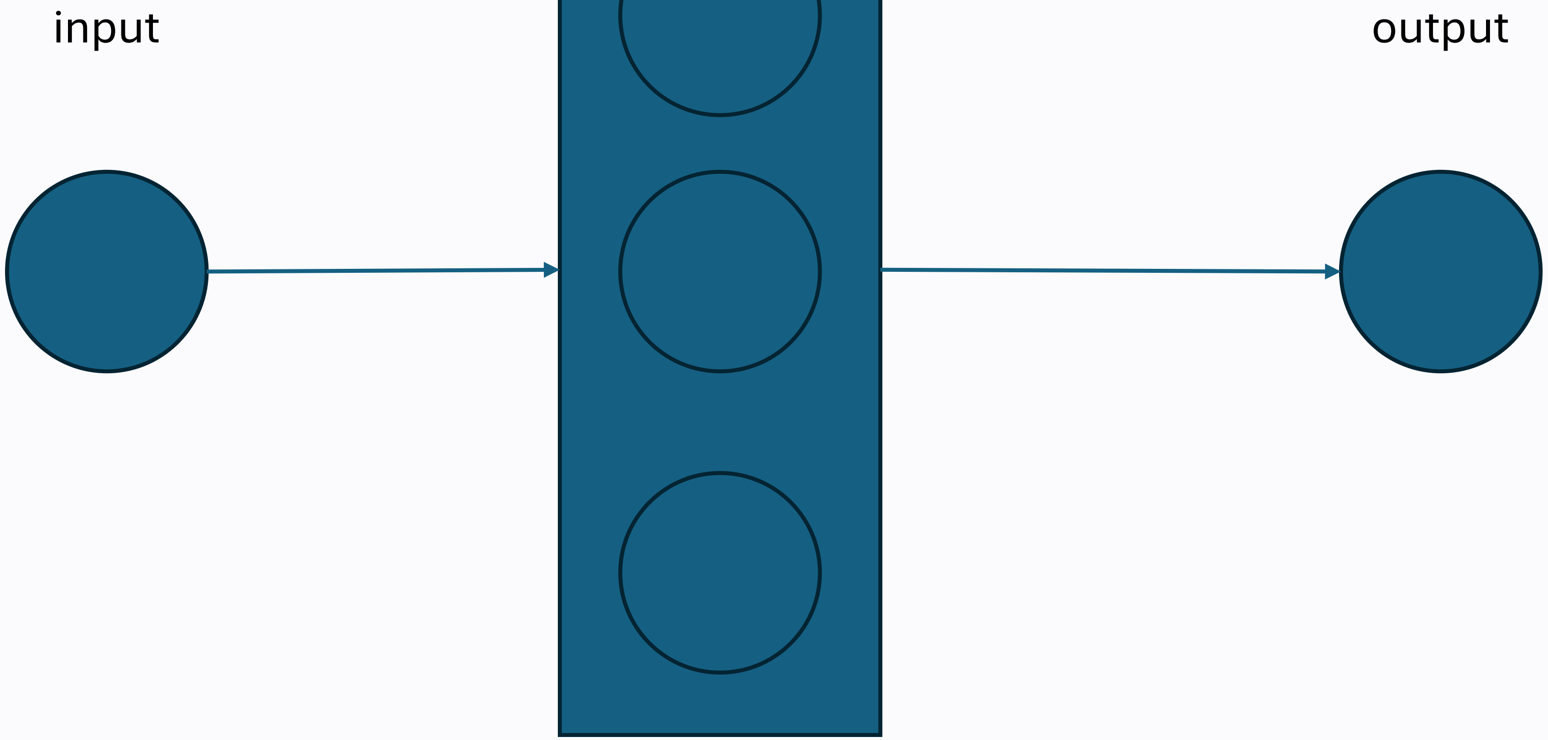
Neural Networks



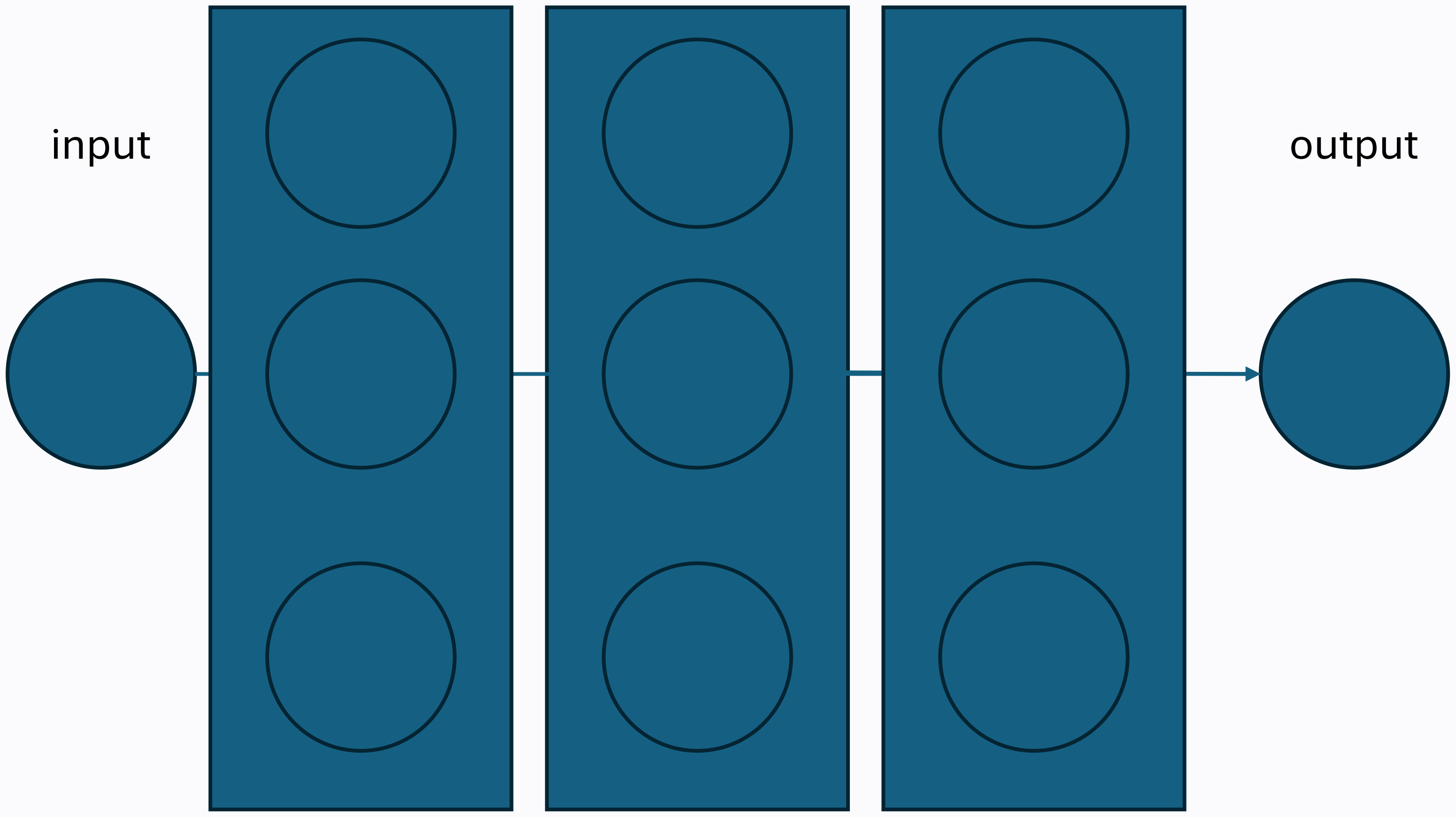
Neural Networks



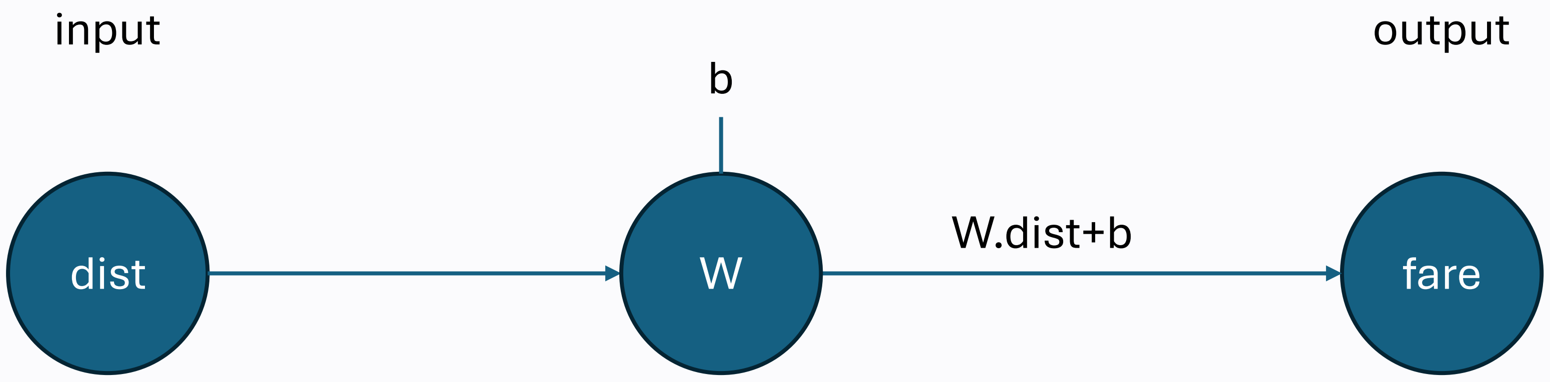
Neural Networks



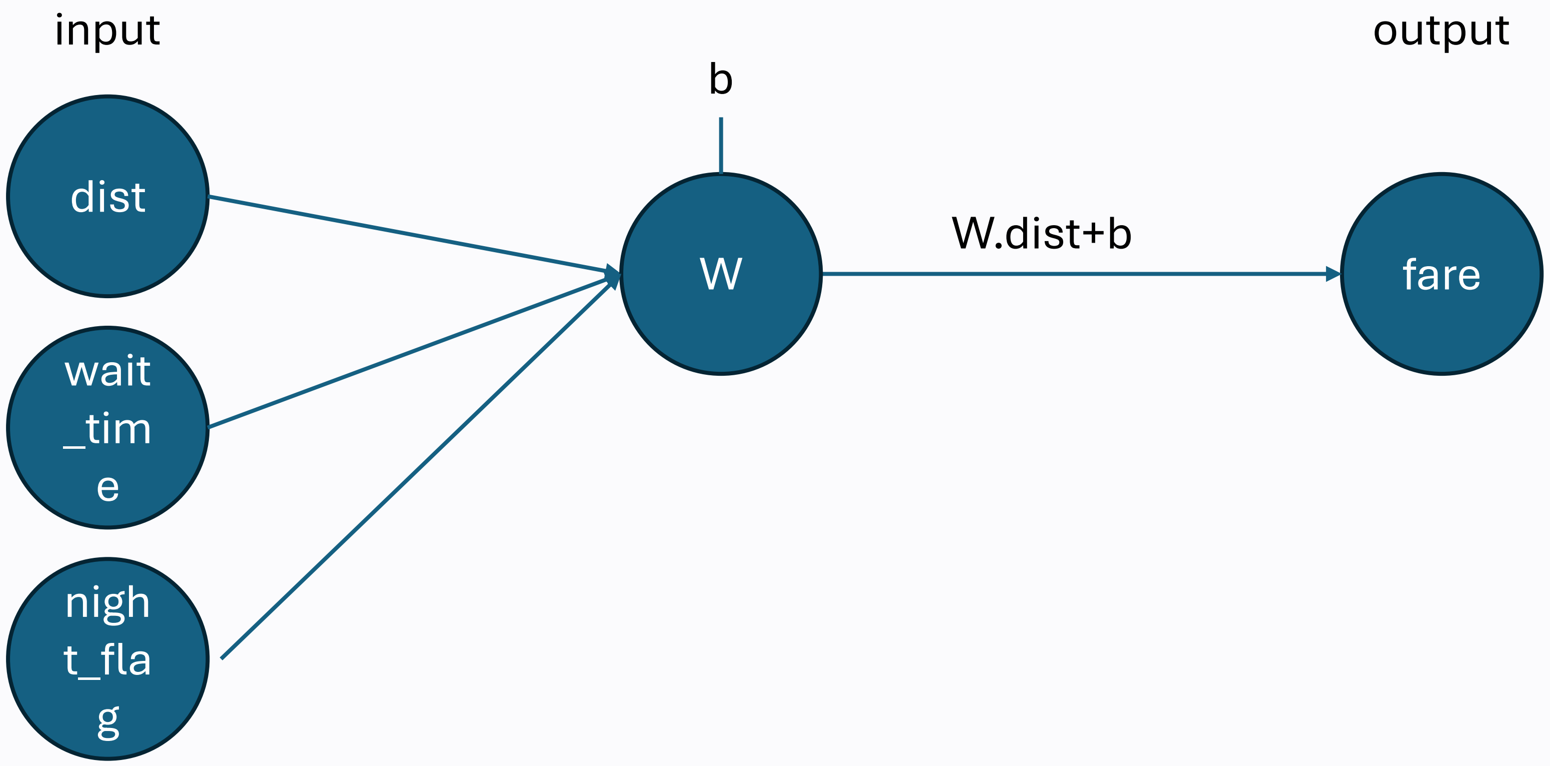
Neural Networks



Neural Networks: Cab Fare 1



Neural Networks: Cab Fare 2



Lab: Defining a Neural Network

Linear Regression

We can approximate the relationship with a straight line.
This approach is called **Linear Regression**.

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

The **predicted** output.

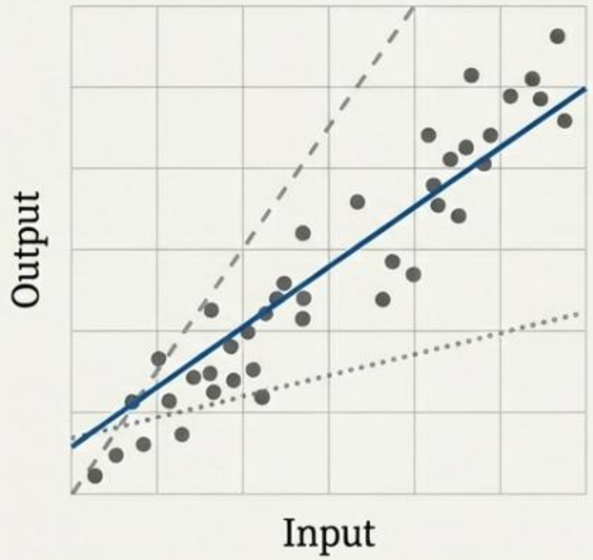
The **weight** (or slope), representing the input's importance.

The input feature you know.

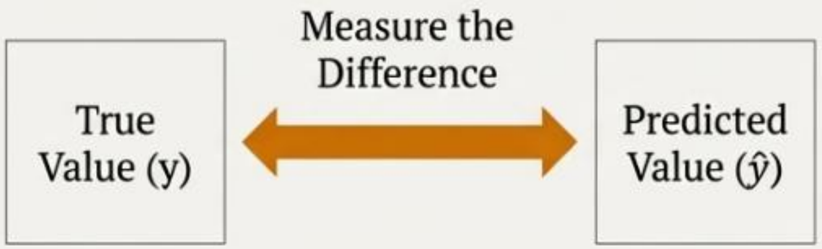
The **bias** (or intercept), a baseline value.

For multiple features, the idea is the same, just in higher dimensions: $\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + b$

Loss Function



This raises a critical question: What does *best fit* actually mean?



Measure the Error with MSE

Mean Squared Error (MSE) measures the average of the squared differences between the predicted values and the actual values.

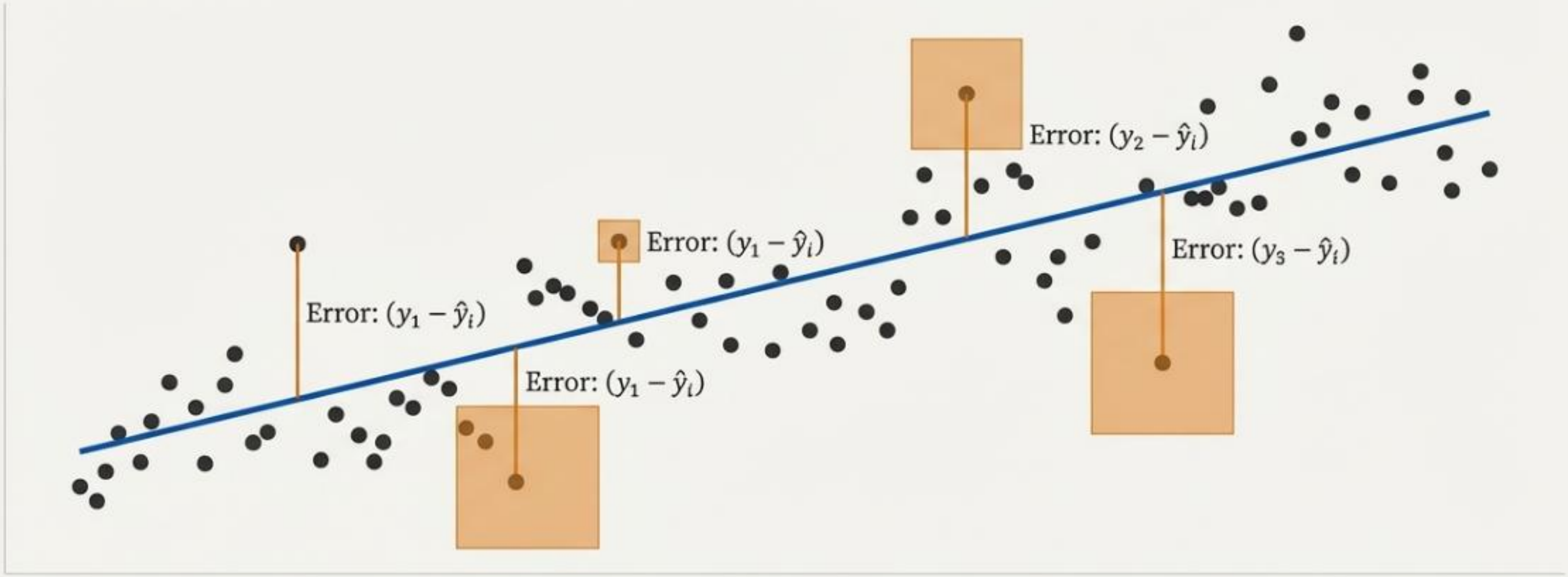
$$\mathrm{MSE} = \frac{1}{n} \sum \overbrace{(y_i - \hat{y}_i)^2}^{\text{The error for a single prediction.}}$$

The total number of samples.

The true value for the i-th sample.

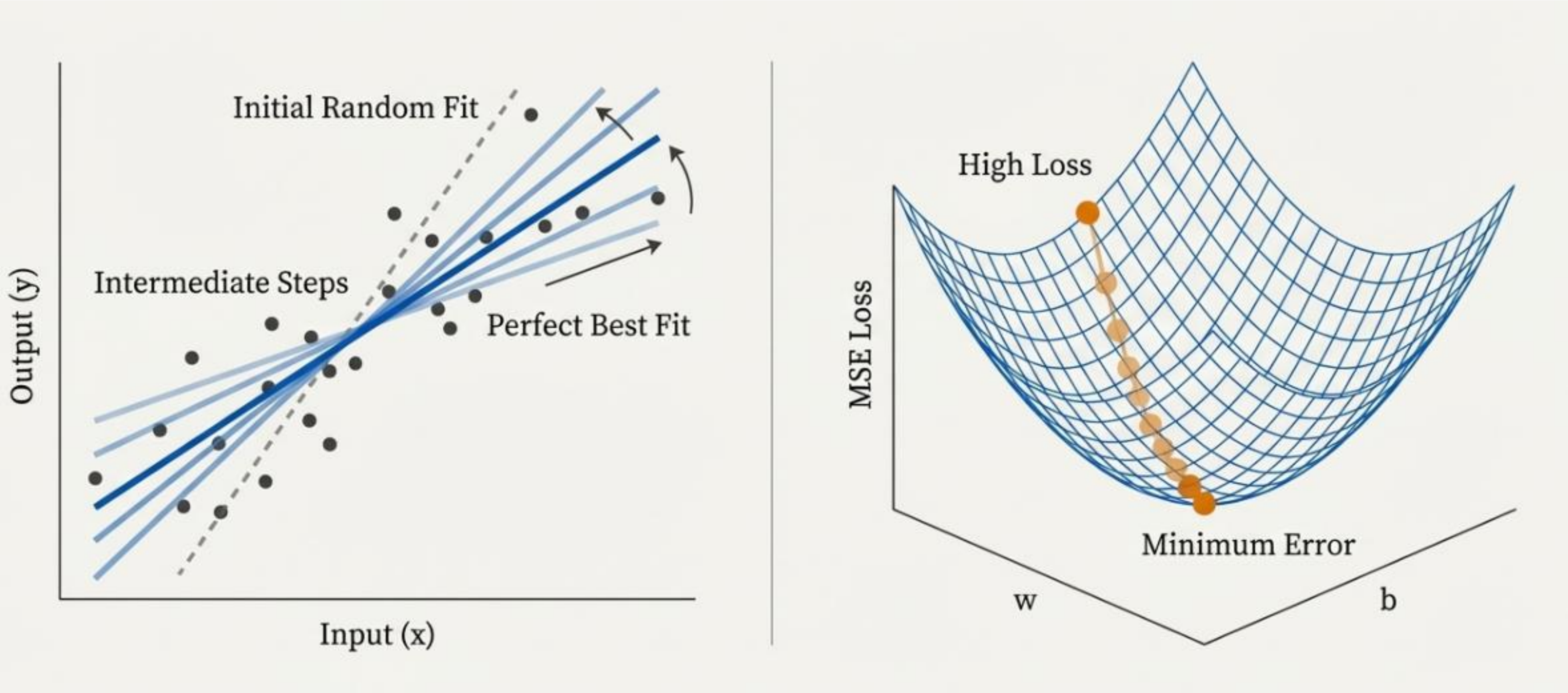
The predicted value for the i-th sample.

Visualizing Mean Squared Error in Action

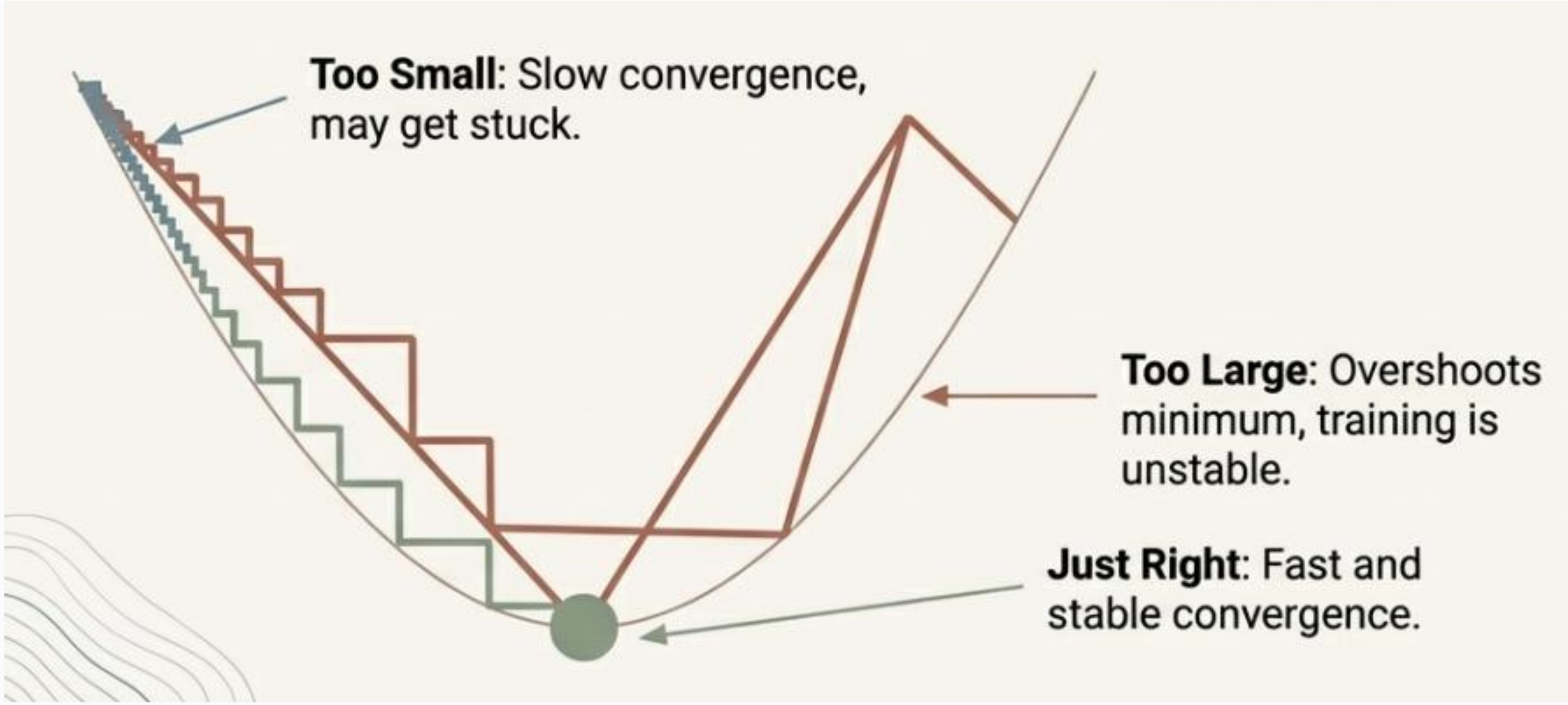


The 'best fit' line is the one that makes the total area of all these squares as small as possible. This is what MSE minimizes.

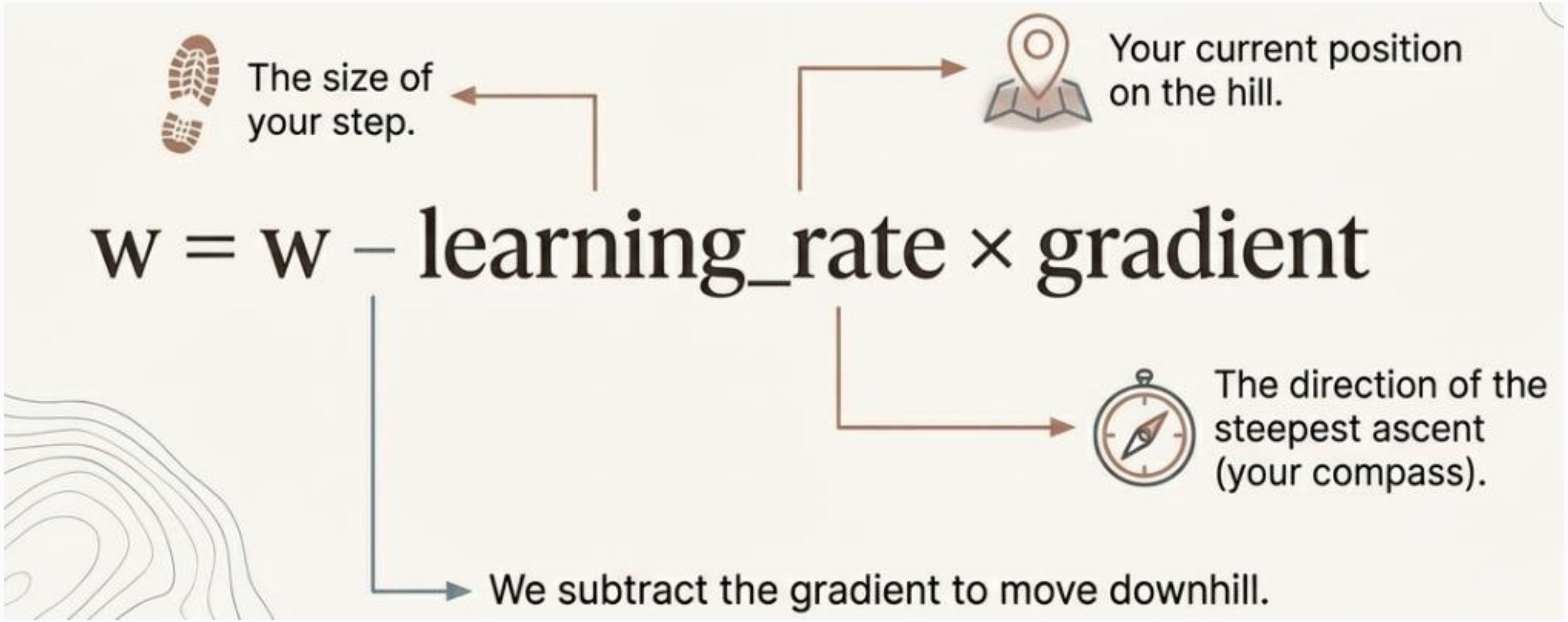
Gradient Descend



Learning Rate



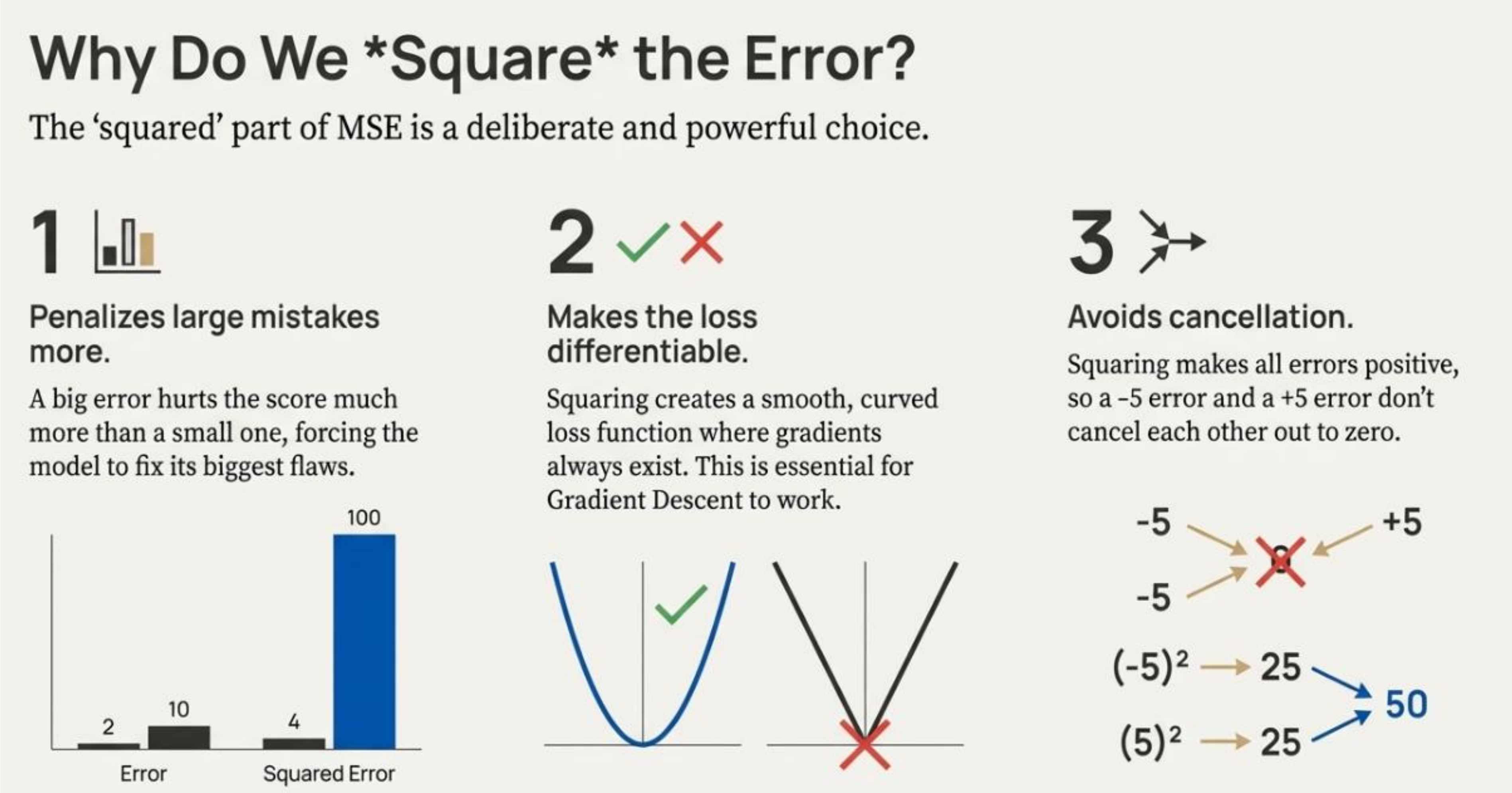
Weight Update



Training Loop Skeleton

Putting it all together

Appendix



Why Do We *Square* the Error?
The ‘squared’ part of MSE is a deliberate and powerful choice.
1
Penalizes large mistakes more.
A big error hurts the score much more than a small one, forcing the model to fix its biggest flaws.
100
2
10
4
Error
Squared Error
2
Makes the loss differentiable.
Squaring creates a smooth, curved loss function where gradients always exist. This is essential for Gradient Descent to work.
3
Avoids cancellation.
Squaring makes all errors positive, so a -5 error and a +5 error don’t cancel each other out to zero.
-5
+5
-5
(-5)²
25
50
(5)²
25