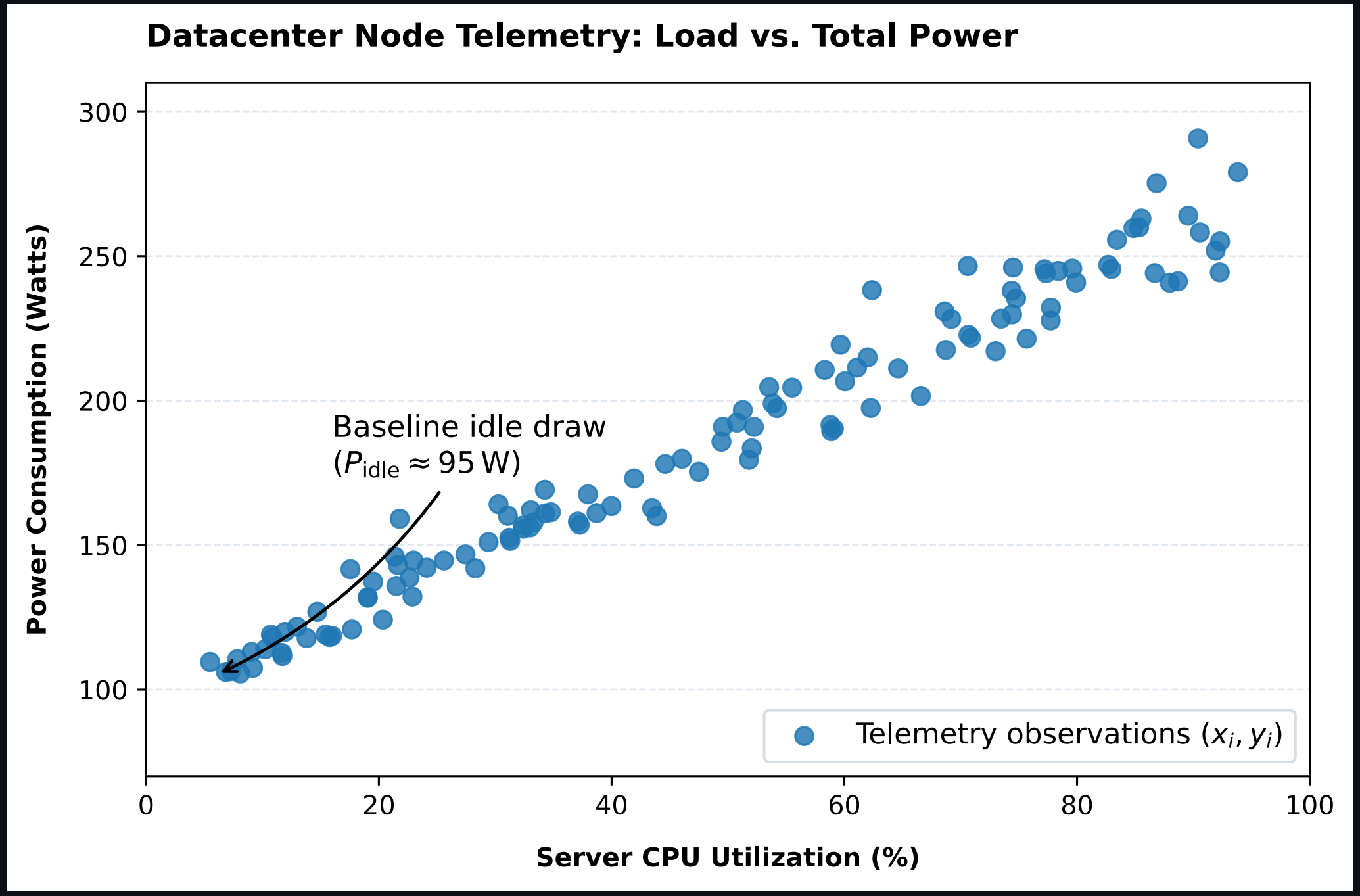
Machine Learning

Imperial College London

Nuri Cingillioglu

Part 2: Statistical Foundations of ML

The glory of curve-fitting awaits you all...


Datacenter Node Telemetry: Load vs. Total Power
Power Consumption (Watts)
300
250
200
150
100
Baseline idle draw
($P_{idle} \approx 95$ W)
Telemetry observations (x_i, y_i)
0
20
40
60
80
100
Server CPU Utilization (%)

One step back

Recall the artificial neuron (perceptron) model:

$$h(\mathbf{w}^T\mathbf{x} + b) = h(\mathbf{w}^T[\mathbf{x}, 1])$$

now let's remove the activation:

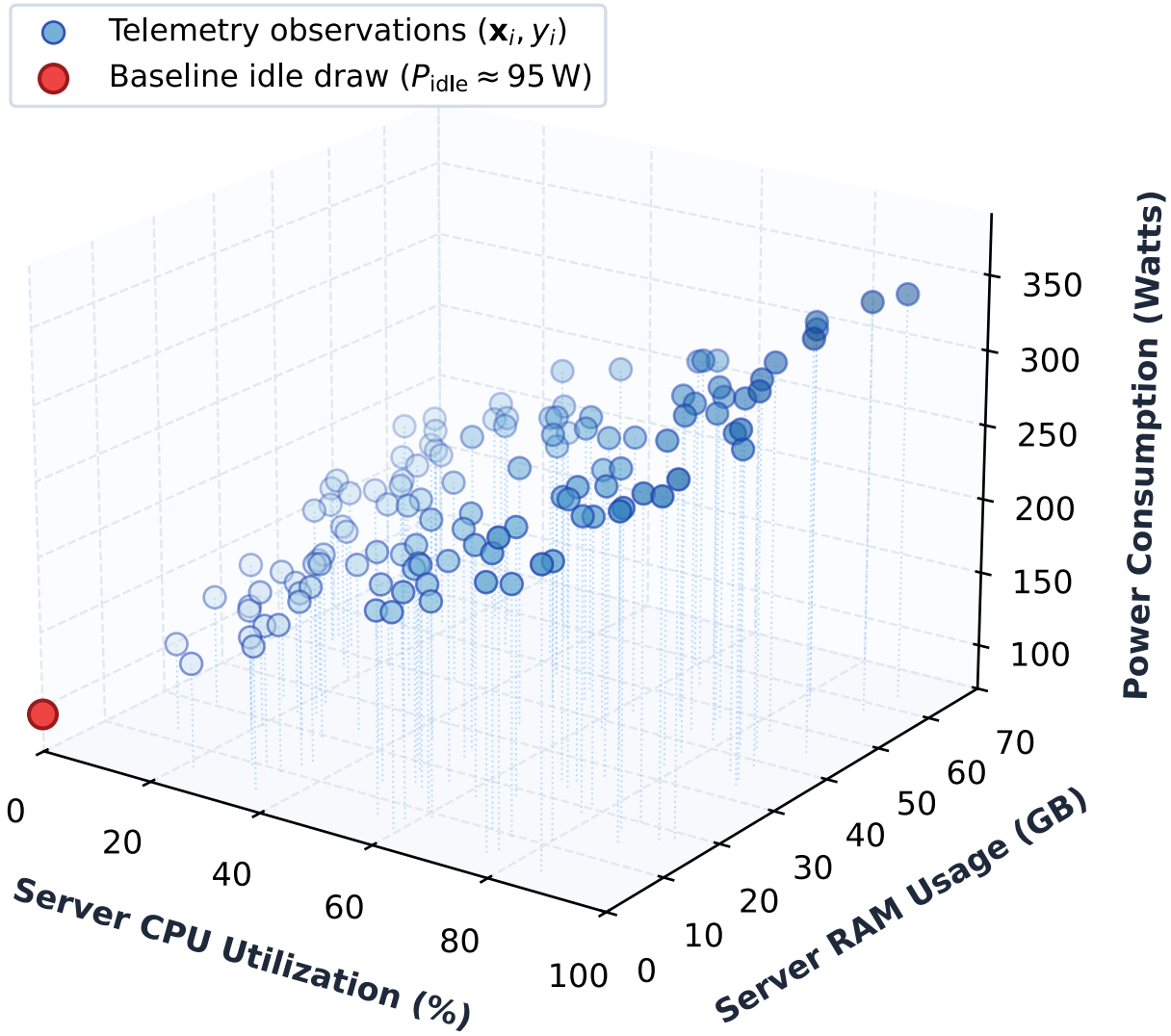
$$y = \mathbf{w}^T\mathbf{x} + b$$

oh gosh, this looks like a line, doesn't it??? We have effectively come back to the gooooood old

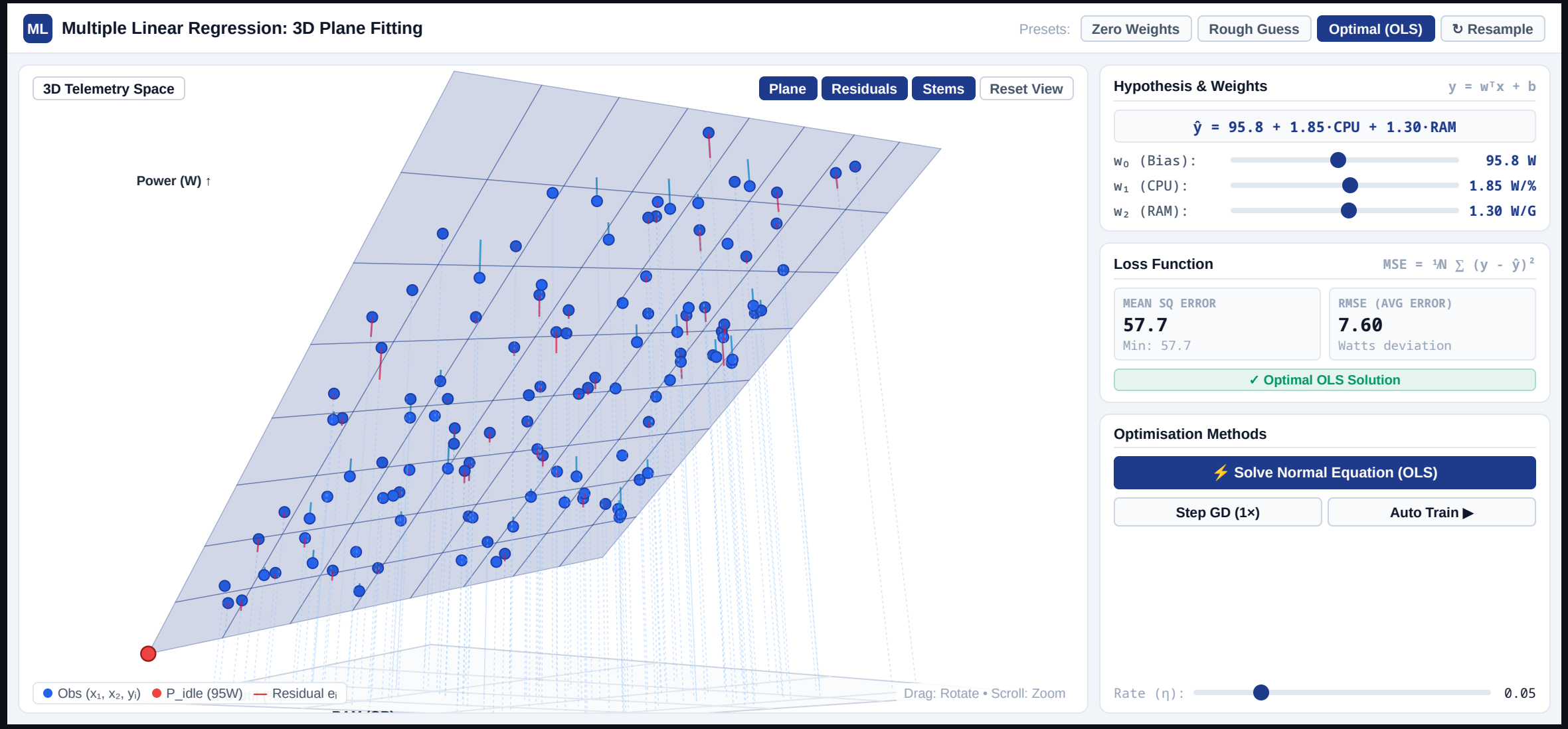
$$y = ax + b$$

> **Let's machine learn this then**

Datacenter Node Telemetry: CPU & RAM Load vs. Total Power



Fitting a plane in action



Linear Regression

For some given inputs $\mathbf{x} \in \mathbb{R}^d$ we have an associated output $y \in \mathbb{R}$ modelled as:

$$y = \mathbf{w}^T \mathbf{x} + b = \sum_{i=1}^{d} w_i x_i + b$$

For convention, we'll let $x_0 = 1$ and $w_0 = b$. Then, our model becomes:

$$y = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{d} w_i x_i$$

> So how do we find the best parameters? What does best mean?

Notation abuse

You'll see, even in this course, that the notation may change as we go along. There is no right or wrong, just different ways people write things down. I've seen all these:

$$f(\mathbf{x}; \theta) = f_\theta(\mathbf{x}) = f_w(\mathbf{x}) = h_\mathbf{w}(\mathbf{x}) = \theta^T \mathbf{x} = \mathbf{w}^T \mathbf{x} = \dots$$

- It's all the same thing, some parameterised function that we are going to optimise.
- There is some baseline similarity which is f, g, h etc are used for functions, θ, $\mathbf{w}$, b etc are used for parameters and x is input, y is output.

Least Squares

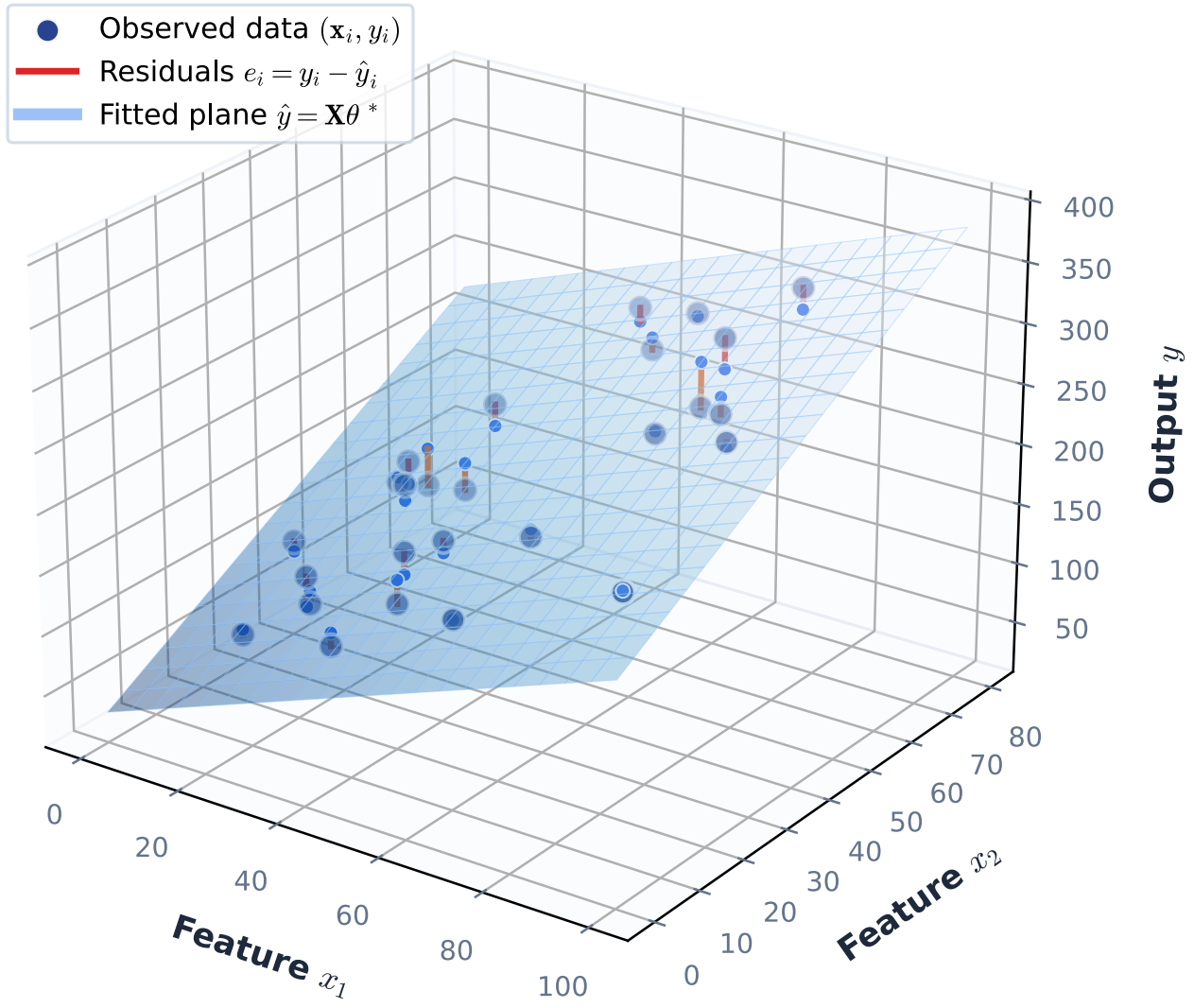
We need a way to measure how well our model is doing. Consider the *distance* between the predictions $\hat{y}_i$ and true values y_i:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \frac{1}{2}\|\mathbf{X}\theta - \mathbf{y}\|_2^2$$

Now, we can minimise the error by minimising $J(\theta)$:

$$\theta^* = \arg\min_{\theta} J(\theta)$$

Least Squares Regression: Minimising Squared Residuals $\frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$



Ordinary Least Squares (OLS)

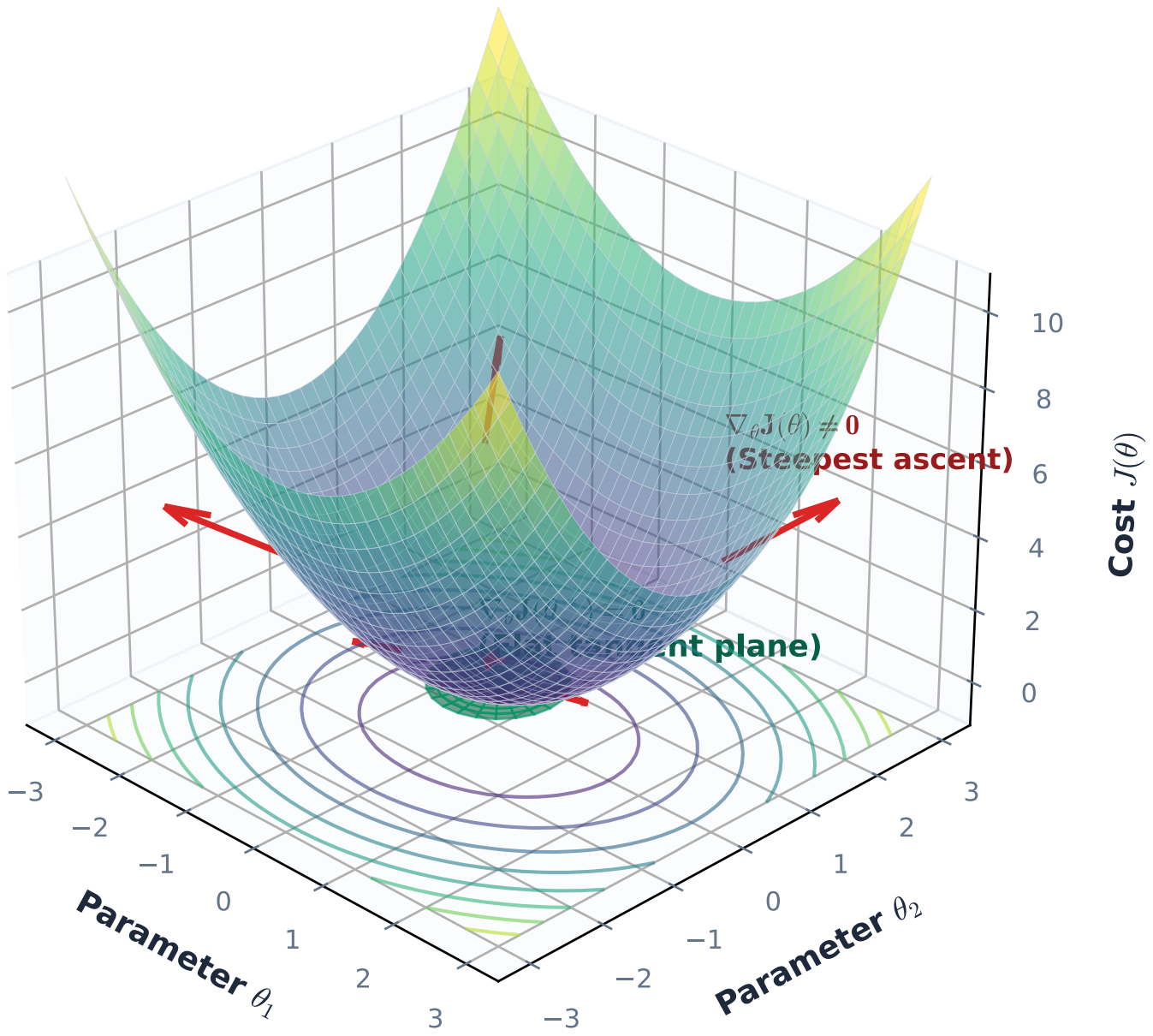
From calculus we know that the minimum of $J(\theta)$ occurs when its derivative is zero. Given N training samples with features $\mathbf{X} \in \mathbb{R}^{N \times d}$ and outputs $\mathbf{y} \in \mathbb{R}^N$ (assuming d includes the bias parameter and $x_0 = 1$):

$$\nabla_\theta J(\theta) = \left[\frac{dJ}{d\theta_0}, \frac{dJ}{d\theta_1}, \ldots, \frac{dJ}{d\theta_d} \right]^T = \mathbf{0}$$

- This is a system of d equations in d unknowns written in matrix form.

In vector calculus, the gradient of a scalar function with respect to a vector or matrix is the vector or matrix of the partial derivatives.

Cost Surface $J(\theta)$: Gradient $\nabla_\theta J(\theta) = 0$ at Minimum



In matrix form

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1d} \\ 1 & x_{21} & x_{22} & \dots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & \dots & x_{Nd} \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

Now, since $f(\mathbf{x}; \theta) = \theta^T \mathbf{x}$ for a single input vector, we write in matrix form:

$$\mathbf{X}\theta = \mathbf{y}$$

> Shape check: $(N, d)(d, 1) = (N, 1)$.

Using the fact that for any vector z we have $z^T z = \sum_i z_i^2 = ||z||_2^2$, we can write our cost function as:

$$J(\theta) = \frac{1}{2}(\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y}) = \frac{1}{2}\sum_i (\mathbf{x}_i^T\theta - y_i)^2$$

Normal equations

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y}) \\
&= \frac{1}{2}\nabla_\theta \left((\mathbf{X}\theta)^T\mathbf{X}\theta - (\mathbf{X}\theta)^T\mathbf{y} - \mathbf{y}^T(\mathbf{X}\theta) + \mathbf{y}^T\mathbf{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T(\mathbf{X}^T\mathbf{X})\theta - \mathbf{y}^T(\mathbf{X}\theta) - \mathbf{y}^T(\mathbf{X}\theta)\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T(\mathbf{X}^T\mathbf{X})\theta - 2(\mathbf{X}^T\mathbf{y})^T\theta\right) \\
&= \frac{1}{2}\left(2\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{X}^T\mathbf{y}\right) \\
&= \mathbf{X}^T\mathbf{X}\theta - \mathbf{X}^T\mathbf{y}
\end{aligned}
$$

Recall that: $(AB)^T = B^T A^T$, $a^T b = b^T a$, $\nabla_x b^T x = b$ and $\nabla_x x^T A x = 2Ax$ for symmetric matrix A.

Normal equations

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

The closed-form solution $\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ requires that $\mathbf{X}^T\mathbf{X}$ is invertible. That inversion breaks down whenever $\mathbf{X}$ lacks sufficient independent features:

- **Too few samples ($N < d$):** There are fewer training examples than parameters to learn.
- **Redundant features:** One or more features can be written as a linear combination of others.

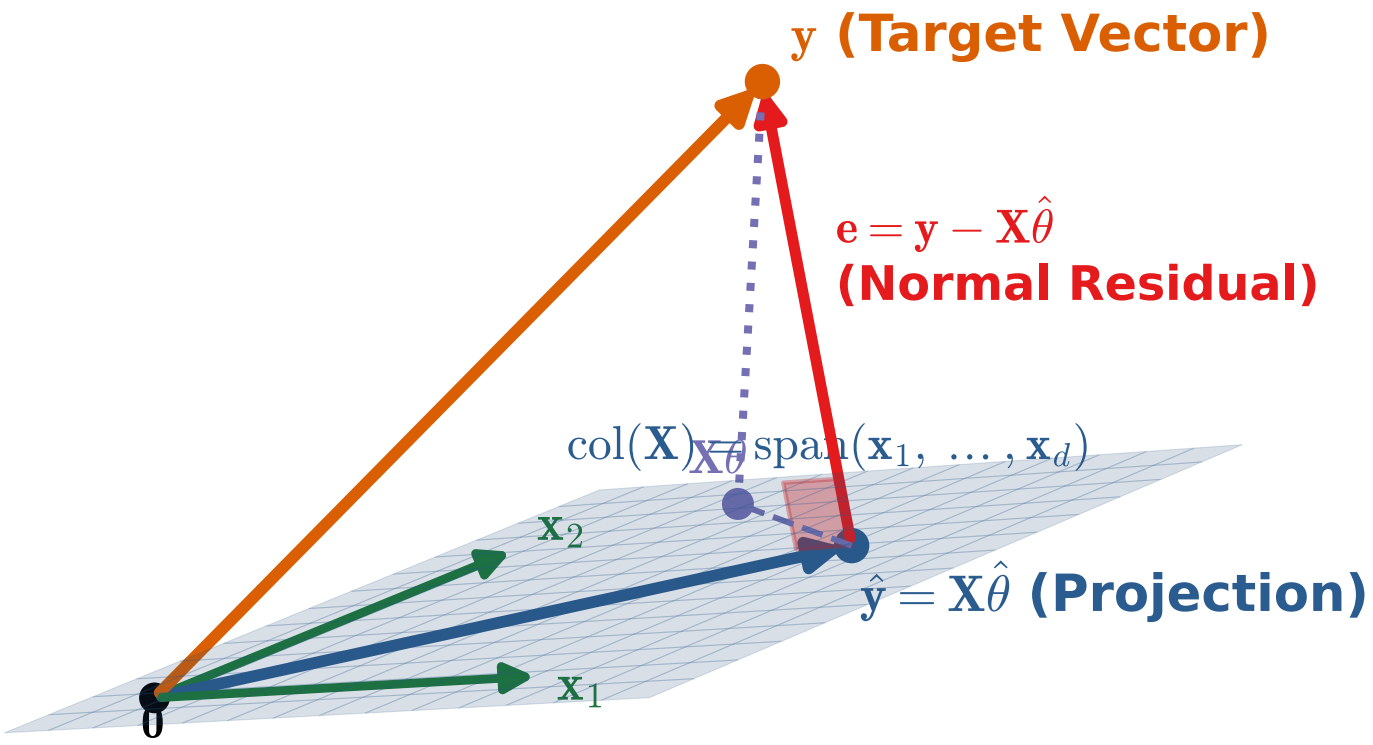
In either case, $\mathrm{rank}(\mathbf{X}) < d$, rendering $\mathbf{X}^T\mathbf{X}$ singular (non-invertible) and leaving infinitely many parameter vectors that minimise the loss.

Why call it normal equations?

$$\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y}) = \mathbf{0}$$
$$\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y}$$

This equation is called the **normal equation** because it represents the condition that the residual $\mathbf{X}\theta - \mathbf{y}$ is orthogonal (or “normal”) to the column space of $\mathbf{X}$.

Geometric Intuition of the Normal Equation: $\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\theta}) = \mathbf{0} \Rightarrow \mathbf{X}^T\mathbf{X}\hat{\theta} = \mathbf{X}^T\mathbf{y}$



Mini-break

When we removed the activation function from the perceptron, we ended up with linear regression. *Doesn't look like our artificial brain...*

- This is an equation / problem with a rich history.
- We are now at the mercy of statistics, linear algebra, calculus and probability theory...

> Please hang in there.

Gradient descent

I don't want to invert matrices, I want to crank the handle.

We know that the minimum occurs when the **derivative is zero**. So, we can start with an initial guess for θ and iteratively update it in the direction of the negative gradient:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

where $\alpha > 0$ is the **learning rate**. This update occurs simultaneously for all $j = 0, \ldots, d$:

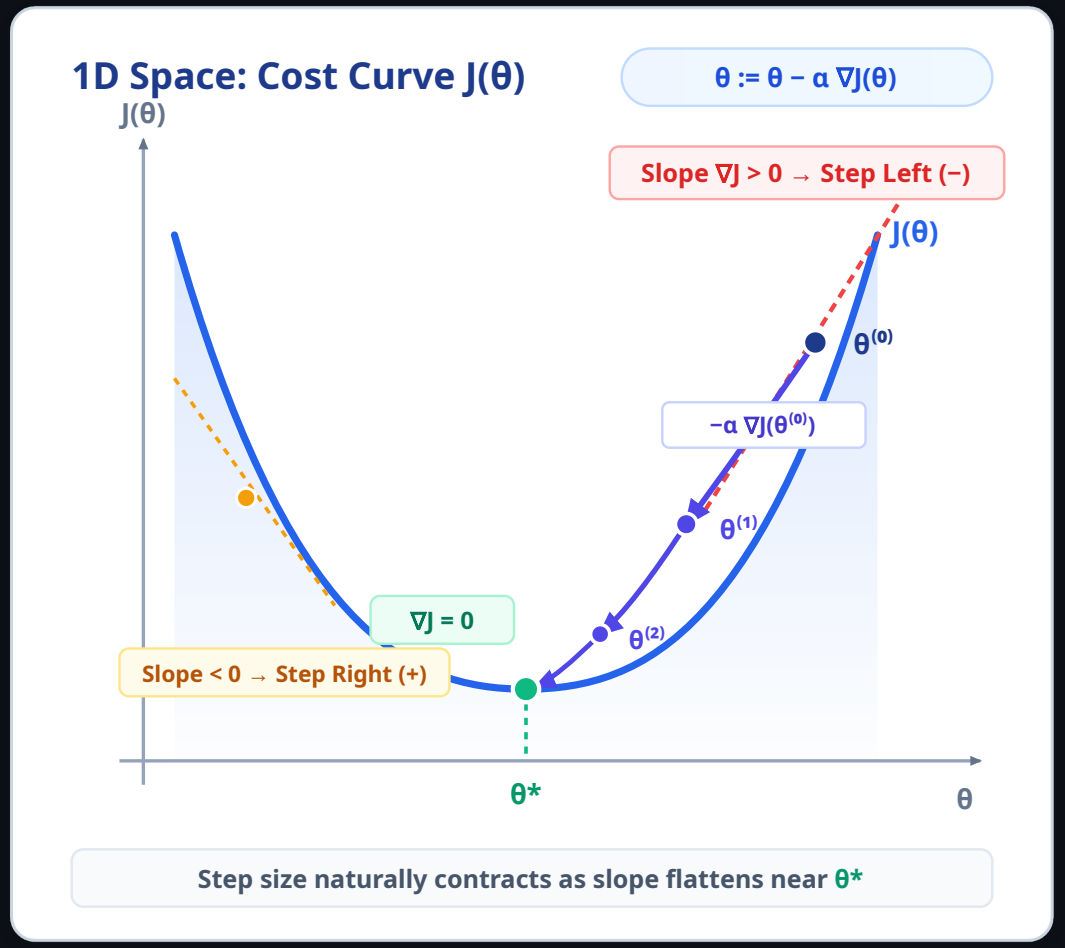
$$\theta := \theta - \alpha \nabla_\theta J(\theta)$$

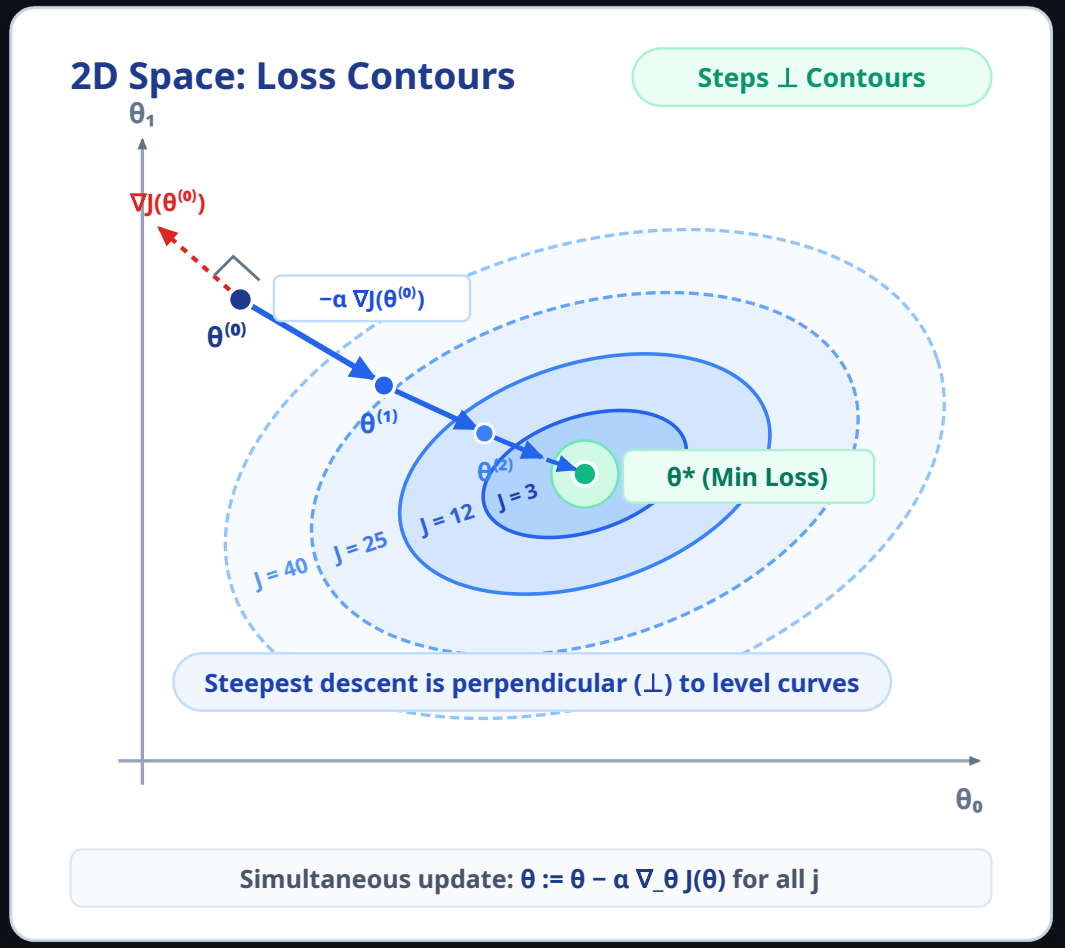
Intuition

1. The gradient $\nabla_\theta J(\theta)$ tells us how each parameter θ_j changes the cost function J.
2. If we go in the direction of the gradient, we increase the cost function. (Gradient Ascent)
3. If we go in the direction of the negative gradient, we decrease the cost function. (Gradient Descent)

This is like walking on a hill blind folded. You take a step in the direction where you feel has the biggest drop until you reach the bottom.

Gradient descent





Least Mean Squares

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (\hat{y} - y)^2 \\
&= 2 \cdot \frac{1}{2} (\hat{y} - y) \cdot \frac{\partial}{\partial \theta_j} (\hat{y} - y) \\
&= (\hat{y} - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{d} \theta_i x_i - y \right) \\
&= (\hat{y} - y) x_j
\end{aligned}
$$

Thus the LMS update rule with gradient descent is:

$$\theta_j := \theta_j - \alpha (\hat{y} - y) x_j$$

This is also known as the **Widrow-Hoff learning rule** (1960).

LMS: Vectorised Form

Given design matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$, targets $\mathbf{y} \in \mathbb{R}^N$, and parameters $\theta \in \mathbb{R}^d$:

$$J(\theta) = \frac{1}{2}\|\mathbf{X}\theta - \mathbf{y}\|_2^2 = \frac{1}{2}(\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y})$$

Gradient Derivation

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbf{X}^T\mathbf{X}\theta - \mathbf{X}^T\mathbf{y} \\ &= \mathbf{X}^T(\mathbf{X}\theta - \mathbf{y})\end{aligned}$$

> Shape check: $(N, d)^T\,((N, d)(d, 1) - (N, 1)) = (d, N)(N, 1) = (d, 1)$

Batch Update Rule

$$\theta := \theta - \alpha\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y})$$

Probabilistic Interpretation

> Is there any other way to look at this? That perhaps gives us why the data, the cost function etc work together?

Let us assume that for a single data point we have:

$$y = \theta^T x + \epsilon$$

where ϵ is an error term that captures the noise in the model. Furthermore, let's assume that these ϵ values are independent and identically distributed with zero mean and variance σ^2 Gaussian distribution: $\epsilon \sim \mathcal{N}(0, \sigma^2)$:

$$p(\epsilon) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon - 0)^2}{2\sigma^2}\right)$$

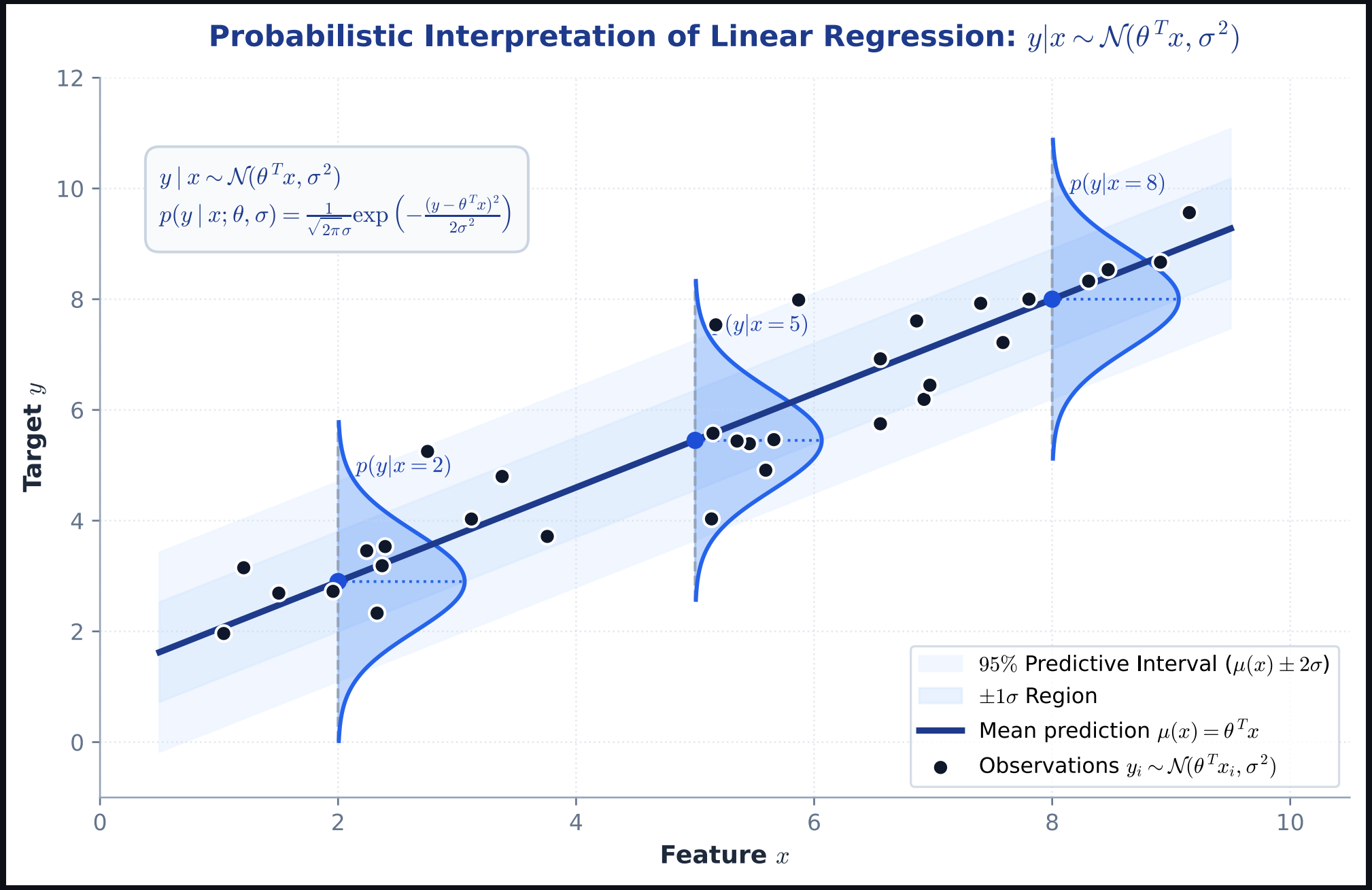
Probabilistic Interpretation cont.

If we replace ϵ with $y - \theta^T x$ in the equation, we get:

$$p(y|x;\theta) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y-\theta^T x)^2}{2\sigma^2}\right)$$

Take a very good look at this, what does it mean?

- It looks like our predictions $\theta^T x$ are the mean of a Gaussian distribution.
- $y|x \sim \mathcal{N}(\theta^T x, \sigma^2)$


Probabilistic Interpretation of Linear Regression: $y|x \sim \mathcal{N}(\theta^T x, \sigma^2)$
$y \mid x \sim \mathcal{N}(\theta^T x, \sigma^2)$
$p(y \mid x; \theta, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y-\theta^T x)^2}{2\sigma^2}\right)$
$p(y|x=2)$
$p(y|x=5)$
$p(y|x=8)$
95% Predictive Interval ($\mu(x) \pm 2\sigma$)
$\pm 1\sigma$ Region
Mean prediction $\mu(x) = \theta^T x$
Observations $y_i \sim \mathcal{N}(\theta^T x_i, \sigma^2)$
Target y
Feature x
0
2
4
6
8
10
12

Likelihood

Surely, we would like to find the parameters θ that maximise the probability of observing $\mathbf{y}$ given $\mathbf{X} \in \mathbb{R}^{N \times d}$ and the parameters θ:

$$L(\theta) = L(\theta; \mathbf{X}, \mathbf{y}) = p(\mathbf{y}|\mathbf{X}; \theta)$$

This is our likelihood function. Notice how it is a function of θ now. Using IID assumption:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{N} p(y_i|\mathbf{x}_i; \theta) \\ &= \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \theta^T \mathbf{x}_i)^2}{2\sigma^2}\right) \end{aligned}$$

Here $\mathbf{x}_i$ denotes the i-th feature vector and y_i its target scalar.

Maximum Likelihood Estimation (MLE)

Maximum likelihood estimation is about finding parameters θ that maximise $L(\theta)$.

> What are the best parameters θ that maximise the probability of observing $\mathbf{y}$ given $\mathbf{X}$?

To avoid dealing with a huge product of terms that can underflow (multiply to zero), we can instead maximise the logarithm of the likelihood function:

$$\arg\max_{\theta} L(\theta) = \arg\max_{\theta} \log(L(\theta))$$

since logarithm is a monotonically increasing function.

> This is really it. Behind the scenes of AI, we maximise some likelihood.

MLE Cont.

$$
\begin{aligned}
\log(L(\theta)) &= \log\left(\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y_i-\theta^T\mathbf{x}_i)^2}{2\sigma^2}\right)\right) \\
&= \sum_{i=1}^{N} \log\left(\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y_i-\theta^T\mathbf{x}_i)^2}{2\sigma^2}\right)\right) \\
&= \sum_{i=1}^{N} \left(\log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \log\left(\exp\left(-\frac{(y_i-\theta^T\mathbf{x}_i)^2}{2\sigma^2}\right)\right)\right) \\
&= \sum_{i=1}^{N} \left(-\log(\sqrt{2\pi}\sigma) - \frac{(y_i-\theta^T\mathbf{x}_i)^2}{2\sigma^2}\right) \\
&= -N\log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i-\theta^T\mathbf{x}_i)^2
\end{aligned}
$$

Connecting MLE to Least Squares

Maximising the log-likelihood with respect to θ:

$$\begin{aligned}
\arg\max_{\theta} \log(L(\theta)) &= \arg\max_{\theta} \left[-N \log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \theta^T \mathbf{x}_i)^2 \right] \\
&= \arg\max_{\theta} \left[-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \theta^T \mathbf{x}_i)^2 \right] \\
&= \arg\min_{\theta} \frac{1}{2} \sum_{i=1}^{N} (y_i - \theta^T \mathbf{x}_i)^2 \\
&= \arg\min_{\theta} J(\theta)
\end{aligned}$$

which is our original least squares cost function. Also notice how σ^2 was not important in the final result...

Connecting MLE to Least Squares

$$\arg\max_{\theta} \log(L(\theta)) = \arg\min_{\theta} J(\theta)$$

Under the assumptions we made, least squares regression corresponds to maximum likelihood estimation.

- This sort of jiggery pockery is not *necessary* although you will come across it a fair amount...
- A geometric interpretation from our visuals is an equally valid, good and rational procedure and there may be indeed other assumptions and / or interpretations.

> But the Bayesian conjugate prior of the normal posterior estimation is another normal...
> **This is proof by intimidation.** Don't be scared.

Where we are so far?

- We started with a biological neuron and how we can model it computationally.
- We looked at the Perceptron and struggled with XOR
- We then briefly covered k-nearest neighbours
- We removed the activation function to discover linear regression and saw how it related to maximum likelihood estimation.

Questions?

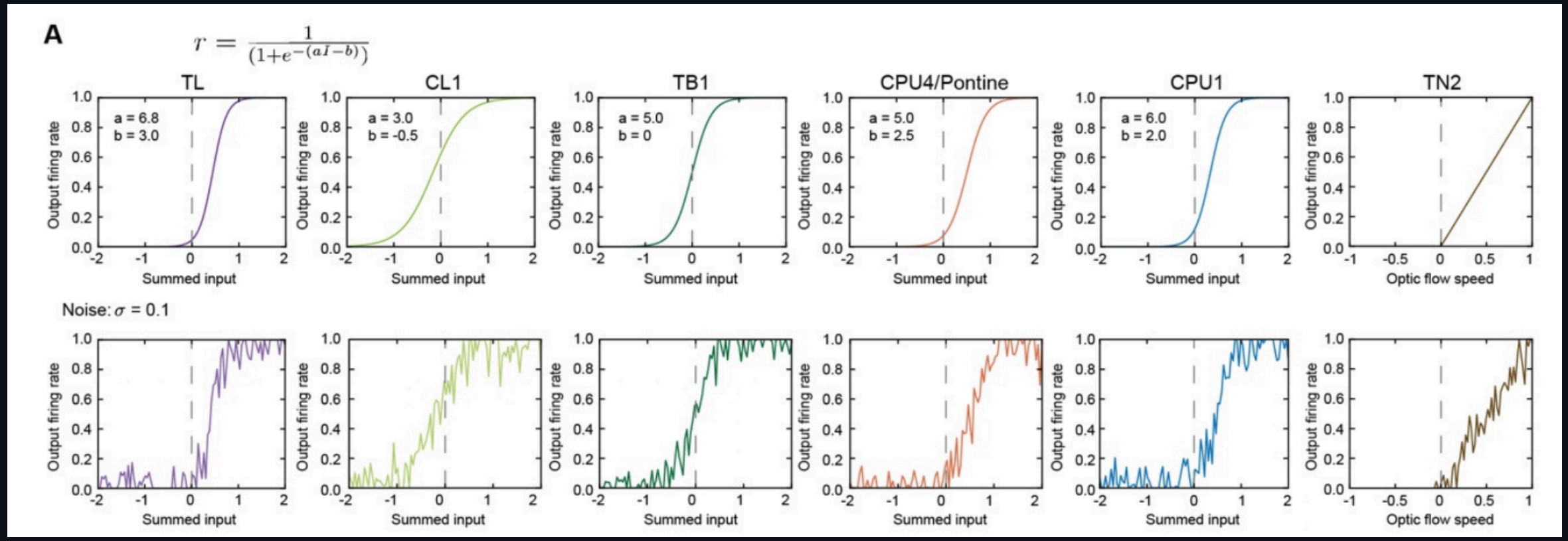
Linear Regression for Classification?

Suppose our task is binary classification: $y \in \{0, 1\}$. Why can't we simply fit a linear regression model $y = \mathbf{w}^T\mathbf{x} + b$ and threshold predictions at $\hat{y} \geq 0.5$?

1. **Unbounded Predictions:** A linear model produces $\hat{y} \in (-\infty, +\infty)$. Outputs like $\hat{y} = 2.4$ or $\hat{y} = -0.8$ cannot represent valid probabilities.
2. **Extreme Sensitivity to Outliers:** An observation with a large feature value (even if correctly classified) pulls the linear regression line, shifting the entire decision boundary.
3. **Violates Error Assumptions:** Binary targets mean errors $\epsilon = y - \hat{y} = y - (\mathbf{w}^T\mathbf{x} + b)$ cannot be normally distributed with constant variance σ^2.

> We need a smooth, non-linear activation that squashes $(-\infty, +\infty)$ into valid probabilities $[0, 1]$!

Bring back the activation



Stone T, Webb B, Adden A, et al.

An Anatomically Constrained Model for Path Integration in the Bee Brain

Current Biology, 2017; 27, 3069-3085.e11

Logistic function

It seems like some of these are modelling on or off like behaviour following S shape?

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

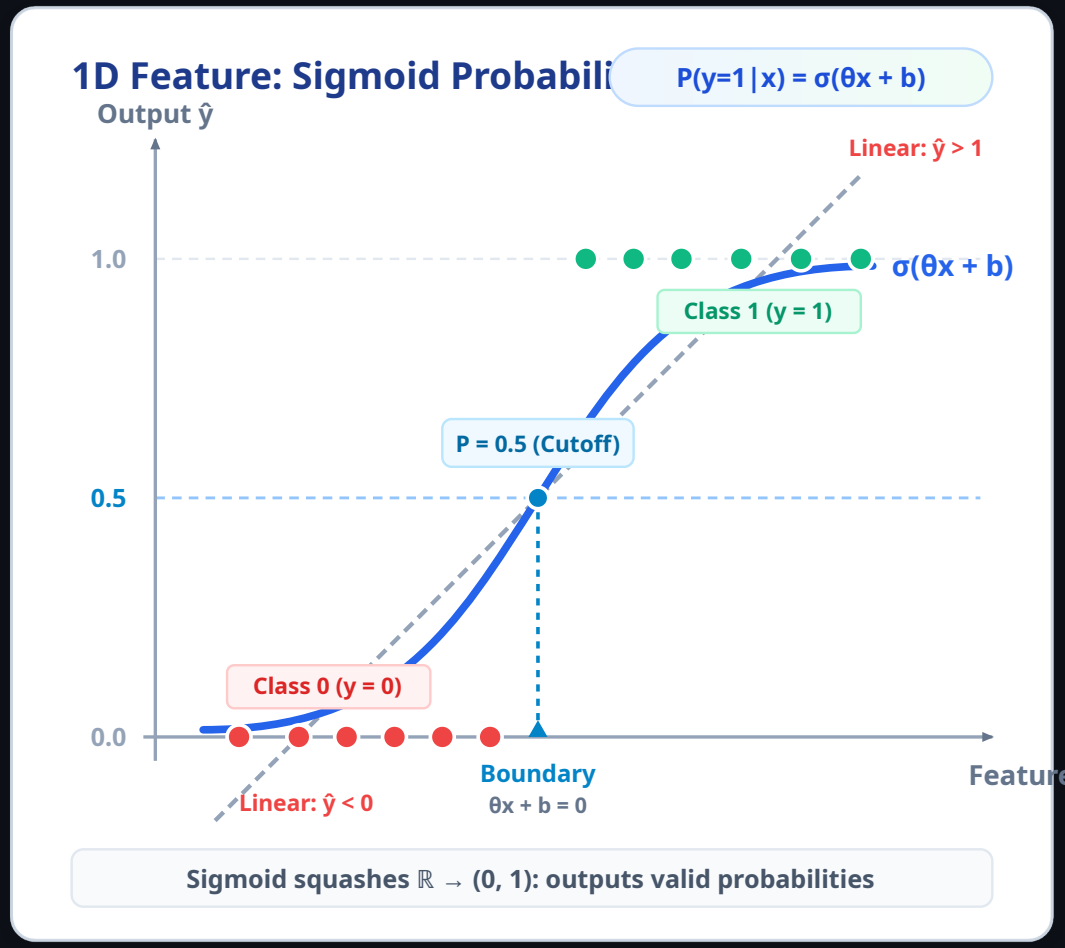
Let's plug this into our previous linear model:

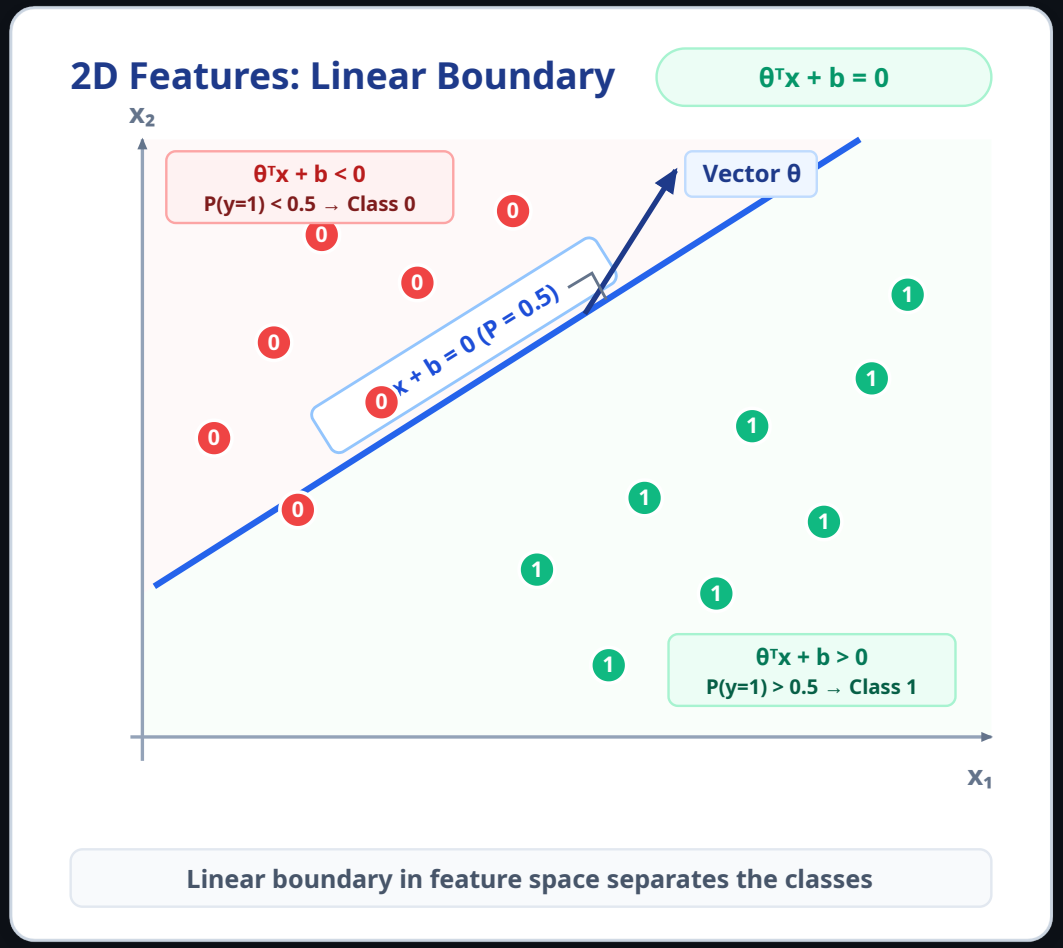
$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{x} + b) = \sigma(\theta^T \mathbf{x})$$

we're back in stats land with *logistic regression*!

> One day, we will have a discussion about artificial brains; but today is not that day.

Logistic Regression





Logistic Regression

The aim is now to classify data rather than predict continuous values. The regression in "logistic regression" is a historical naming quirk - it is a classification algorithm.

- We now have the tools to make sense of this setup and try and optimise it using gradient descent.

Let us assume:

$$P(y = 1|x; \theta) = \sigma(\theta^T x)$$
$$P(y = 0|x; \theta) = 1 - \sigma(\theta^T x)$$

we can rewrite this more compactly:

$$p(y|x; \theta) = \sigma(\theta^T x)^y (1 - \sigma(\theta^T x))^{1-y}$$

MLE again

The last equation looks like a Bernoulli distribution. So our assumption is that our model produces probabilities! Bring in MLE to the rescue:

$$
\begin{aligned}
L(\theta) &= p(\mathbf{y}|\mathbf{X};\theta) \\
&= \prod_{i=1}^{N} p(y_i|\mathbf{x}_i;\theta) \\
&= \prod_{i=1}^{N} \sigma(\theta^T\mathbf{x}_i)^{y_i}(1-\sigma(\theta^T\mathbf{x}_i))^{1-y_i}
\end{aligned}
$$

take the log:

$$
\log(L(\theta)) = \sum_{i=1}^{N}(y_i \log(\sigma(\theta^T\mathbf{x}_i)) + (1-y_i)\log(1-\sigma(\theta^T\mathbf{x}_i)))
$$

Negative Log Likelihood

Maximising log-likelihood is equivalent to minimising the negative log-likelihood:

$$J(\theta) = -\log(L(\theta)) = -\sum_{i=1}^{N}(y_i \log(\sigma(\theta^T \mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma(\theta^T \mathbf{x}_i)))$$

Gradient of Sigmoid

The gradient / derivative of the sigmoid function has a really nice property:

$$\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$$

we can use this and the chain rule of calculus to find the gradient of the cost function with respect to the parameters θ.

Deriving the Gradient: Chain Rule Setup

Let $z_i = \theta^T \mathbf{x}_i$ and $\hat{y}_i = \sigma(z_i)$.

For a single data point i, the negative log-likelihood is:

$$J_i(\theta) = -\left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right]$$

Using the chain rule, we decompose the gradient with respect to θ:

$$\frac{\partial J_i}{\partial \theta} = \frac{\partial J_i}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial \theta}$$

Evaluating the Derivatives

We evaluate each of the three terms separately:

1. **Loss with respect to prediction (deriv. of log, don't panic):**

$$\frac{\partial J_i}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} = \frac{-y_i(1 - \hat{y}_i) + (1 - y_i)\hat{y}_i}{\hat{y}_i(1 - \hat{y}_i)} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)}$$

2. **Prediction with respect to z (Sigmoid derivative):**

$$\frac{\partial \hat{y}_i}{\partial z_i} = \hat{y}_i(1 - \hat{y}_i)$$

3. **Linear combination with respect to θ:**

$$\frac{\partial z_i}{\partial \theta} = \mathbf{x}_i$$

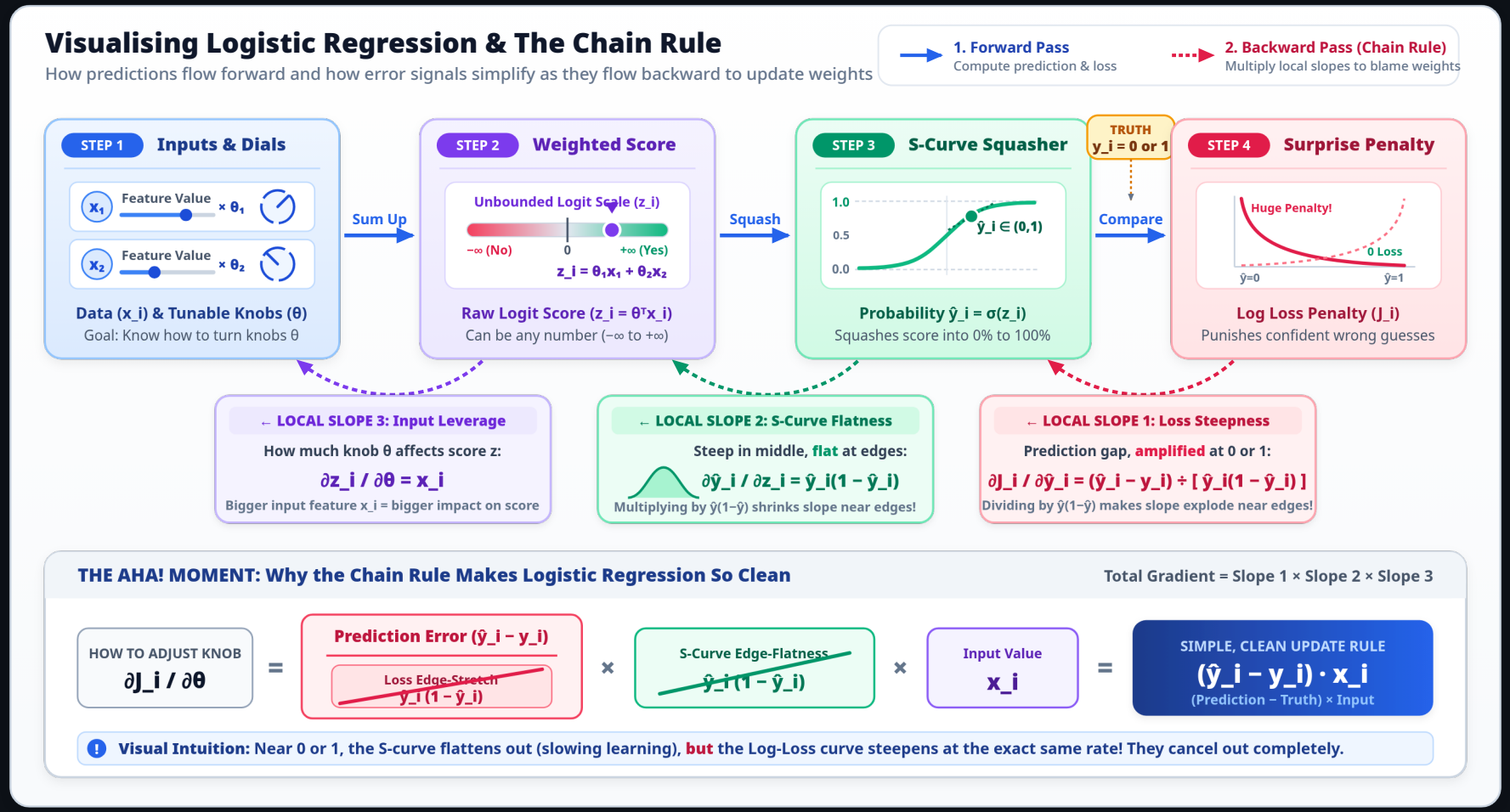
The Chained Derivative

Multiplying the three factors together, the $\hat{y}_i(1-\hat{y}_i)$ terms cancel out cleanly:

$$\begin{aligned}\frac{\partial J_i}{\partial \theta} &= \left(\frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}\right) \cdot \big(\hat{y}_i(1-\hat{y}_i)\big) \cdot \mathbf{x}_i \\ &= (\hat{y}_i - y_i)\mathbf{x}_i \\ &= (\sigma(\theta^T \mathbf{x}_i) - y_i)\mathbf{x}_i\end{aligned}$$

Summing across all N data points, the total gradient of $J(\theta)$ is:

$$\nabla_\theta J(\theta) = \sum_{i=1}^{N} (\sigma(\theta^T \mathbf{x}_i) - y_i)\mathbf{x}_i$$



I asked the best LLM I had access to make a diagram for you, after thinking for a while it gave me this. *Please ignore it*. The LLM just took my beautiful hand written equations and copied it. Where is my PhD level AI assistant now?

Gradient Descent Update

To minimise the cost $J(\theta)$, we take steps in the direction opposite to the gradient:

$$\theta := \theta - \alpha \nabla_\theta J(\theta)$$

Substituting our derived gradient:

$$\theta := \theta - \alpha \sum_{i=1}^{N} (\sigma(\theta^T \mathbf{x}_i) - y_i)\mathbf{x}_i$$

where $\alpha > 0$ is the learning rate.

(Note the exact same functional form as the Least Mean Squares update in linear regression, but here $\sigma(\theta^T \mathbf{x}_i)$ is non-linear!)

Why No Closed-Form Solution?

In linear regression (OLS), setting the gradient to zero gives the normal equations:

$$\nabla_\theta J(\theta) = \mathbf{X}^T(\mathbf{X}\theta - \mathbf{y}) = 0 \implies \theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

For logistic regression, we attempt the same:

$$\nabla_\theta J(\theta) = \sum_{i=1}^{N} \left(\frac{1}{1 + e^{-\theta^T \mathbf{x}_i}} - y_i \right) \mathbf{x}_i = 0$$

The Problem: Non-Linearity

- The parameter vector θ is trapped inside the non-linear sigmoid term $\frac{1}{1+e^{-\theta^T \mathbf{x}_i}}$.
- This forms a system of **coupled non-linear equations** across all N data points.
- You cannot factor out or isolate θ algebraically via matrix inversion.

The Solution: Numerical Optimisation

Because no closed-form solution exists, we must rely on iterative numerical methods:

- **First-order:** Gradient Descent / Stochastic Gradient Descent (SGD)

(Fortunately, in this simple case $J(\theta)$ is convex, so numerical methods are guaranteed to converge to the global minimum.)

> Turns out, this optimisation via gradient *descent* is *decent* enough! You crank it up to large neural networks, with tricks and tweaks, it's what we will be doing in later parts of this course 🫠

Beyond Binary: Multi-Class

So far, our classification world has been strictly binary: $y \in \{0, 1\}$. But the real world rarely offers only two choices:

- Classifying handwritten digits: $0, 1, 2, \ldots, 9$ (MNIST)
- Medical diagnoses: multiple distinct disease categories
- Object recognition: cats, dogs, birds, cars...

Why Not Just Train Multiple Binary Classifiers?

- **One-vs-Rest (OvR):** Train K independent binary classifiers separating class k from all other classes.
- **Problem:** Probabilities are uncalibrated and don't sum to 1. What happens if multiple classifiers output $P > 0.5$, or all output $P < 0.1$?

Multi-class Classification

The Goal: A single unified probabilistic model producing a valid categorical distribution:

$$\mathbf{p} = [p_1, p_2, \ldots, p_K]^T \quad \text{where} \quad p_k \geq 0 \quad \text{and} \quad \sum_{k=1}^{K} p_k = 1$$

We want our model to output this vector $\mathbf{p}$:

$$\mathbf{p} = \begin{bmatrix} P(y = 1|\mathbf{x}; \theta) \\ P(y = 2|\mathbf{x}; \theta) \\ \vdots \\ P(y = K|\mathbf{x}; \theta) \end{bmatrix} = \begin{bmatrix} P(y = \text{cat}|\mathbf{x}) \\ P(y = \text{dog}|\mathbf{x}) \\ \vdots \\ P(y = \text{bird}|\mathbf{x}) \end{bmatrix}$$

From Linear Scores to Logits

For K classes, instead of a single parameter vector θ, each class gets its own weight vector $\mathbf{w}_k \in \mathbb{R}^d$ and bias $b_k \in \mathbb{R}$:

$$z_k = \mathbf{w}_k^T \mathbf{x} + b_k \quad \text{for } k = 1, \ldots, K$$

In compact matrix notation:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^K \quad \text{where } \mathbf{W} \in \mathbb{R}^{K \times d}$$

We call $\mathbf{z} = [z_1, \ldots, z_K]^T$ the **logits** (unconstrained linear scores).

The Challenge

- Each logit z_k can be any real number: $-\infty < z_k < +\infty$.
- Simple division $\frac{z_k}{\sum_j z_j}$ fails: logits can be negative, and the sum can be zero!
- How do we map an arbitrary vector $\mathbf{z} \in \mathbb{R}^K$ into a valid probability distribution?

The Softmax Function

To convert unbounded logits $\mathbf{z}$ into valid probabilities, we take exponents and normalise:

$$\hat{y}_k = p_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Essential Properties

1. **Strictly Positive:** $e^{z_k} > 0 \implies p_k \in (0, 1)$ for all k.
2. **Sums to One:** $\sum_{k=1}^{K} p_k = \frac{\sum_k e^{z_k}}{\sum_j e^{z_j}} = 1$.
3. **Monotonic:** If $z_i > z_j$, then $p_i > p_j$. Larger logits receive exponentially more probability.

What's up with this softmax?

"Soft" Maximum: A smooth, differentiable approximation to argmax. Keeps gradient descent happy.

> If we scale logits by a **temperature** hyperparameter $\tau > 0$ (computing $e^{z_k/\tau}$): as $\tau \to 0^+$, the distribution sharpens into a hard one-hot argmax; as $\tau \to \infty$, it flattens into a uniform distribution ($1/K$).

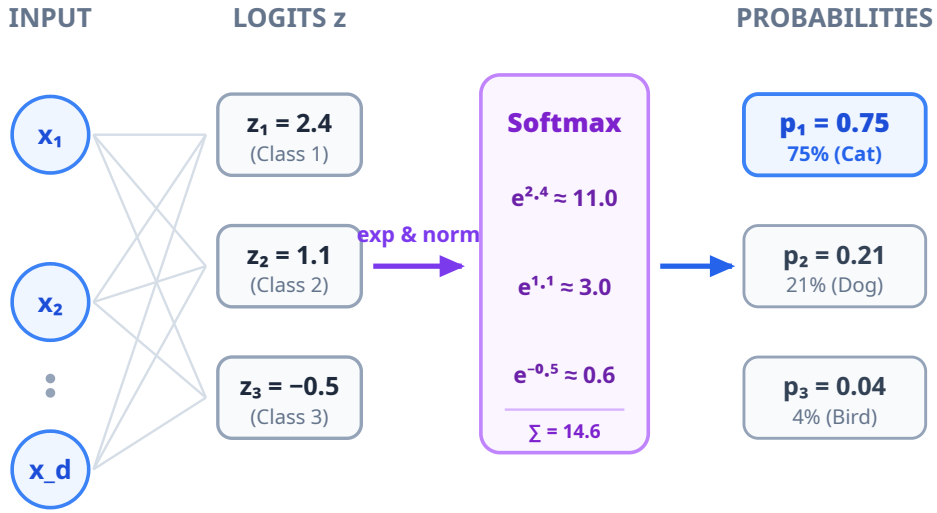
Have you heard of temperature before? Where? We will revisit it later.

Sanity check for $K = 2$: When there are only two classes, $\frac{e^{z_1}}{e^{z_1}+e^{z_2}} = \frac{1}{1+e^{-(z_1-z_2)}} = \sigma(z_1 - z_2)$. Softmax is the exact multi-class generalisation of sigmoid.

Softmax Classification

Softmax: Logits to Probabilities

$p_k = e^{z_k} / \sum e^{z_j}$

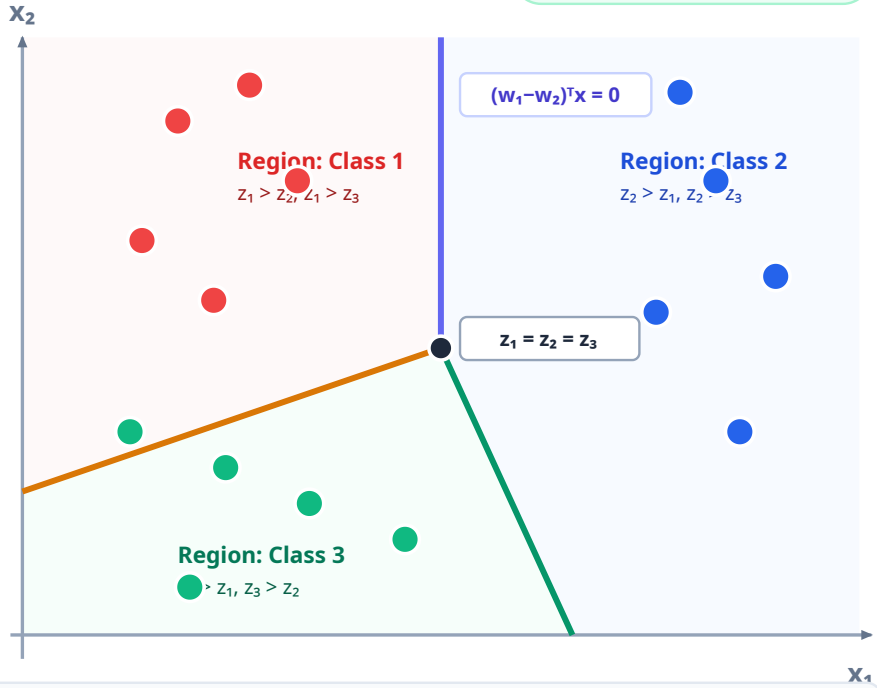


Exponentials guarantee $p_k > 0$ • Normalisation enforces $\sum p_k = 1$

Loss: Cross-Entropy $J = -\sum y_k \log(p_k) = -\log(p_true)$

3-Class Decision Boundaries

Piecewise Linear



Decision boundaries occur where top logits tie: $z_i = z_j$

$(w_i - w_j)^T x + (b_i - b_j) = 0 \rightarrow$ hyperplanes divide $\mathbb{R}^d$ into convex polyhedra

Multi-Class Likelihood

To represent targets across K classes, we use **one-hot encoding**:
If sample i belongs to class c, then $\mathbf{y}_i = [0, \ldots, 1, \ldots, 0]^T$ with $y_{ic} = 1$ and $y_{ik} = 0$ for $k \neq c$
.

Under the **Categorical distribution** (generalised Bernoulli), the probability of observing $\mathbf{y}_i$ is:

$$p(\mathbf{y}_i|\mathbf{x}_i; \mathbf{W}) = \prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}} = \hat{y}_{ic}$$

Because only the true class index c has $y_{ic} = 1$, all other terms are $\hat{y}_{ik}^{0} = 1$. It cleanly **selects** the probability of the true label.

Assuming N independent observations, the total likelihood is:

$$L(\mathbf{W}) = \prod_{i=1}^{N} p(\mathbf{y}_i|\mathbf{x}_i; \mathbf{W}) = \prod_{i=1}^{N} \prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}}$$

Multi-Class Likelihood Example

Consider our 3-class animal classifier ($K = 3$): $\text{cat}(k = 1)$, $\text{dog}(k = 2)$, $\text{bird}(k = 3)$.

For an input $\mathbf{x}$, our model outputs predicted probabilities:

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} = \begin{bmatrix} 0.75 \\ 0.21 \\ 0.04 \end{bmatrix} \begin{matrix} (\text{cat}) \\ (\text{dog}) \\ (\text{bird}) \end{matrix}$$

Suppose the true label is a **cat** ($c = 1$), so the one-hot target is $\mathbf{y} = [1, 0, 0]^T$:

$$\begin{aligned} p(\mathbf{y}|\mathbf{x}; \mathbf{W}) &= \prod_{k=1}^{3} \hat{y}_k^{y_k} \\ &= (\hat{y}_1)^{y_1} \cdot (\hat{y}_2)^{y_2} \cdot (\hat{y}_3)^{y_3} \\ &= (0.75)^1 \cdot (0.21)^0 \cdot (0.04)^0 \\ &= 0.75 \cdot 1 \cdot 1 = \mathbf{0.75} \end{aligned}$$

Categorical Cross-Entropy

Taking the negative log-likelihood gives our cost function:

$$J(\mathbf{W}) = -\log L(\mathbf{W}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

For a single data point (omitting the sample index i):

$$J(\mathbf{W}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k = -\log \hat{y}_{\text{true}}$$

because only the true label has $y_k = 1$. This is the famous **Categorical Cross-Entropy Loss**.

Intuition

- If the model is confident and correct ($\hat{y}_{\text{true}} \to 1$): Loss $\to -\log(1) = 0$.
- If the model is wrong ($\hat{y}_{\text{true}} \to 0$): Loss $\to -\log(0) \to +\infty$ (severely penalised!).

Multi-Class Gradient Update

In vector form, the gradient with respect to the logits $\mathbf{z}$ is (see appendix for gory details):

$$\nabla_{\mathbf{z}} J = \hat{\mathbf{y}} - \mathbf{y}$$

And applying the chain rule to the weight vector $\mathbf{w}_k$ for class k:

$$\nabla_{\mathbf{w}_k} J = \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \mathbf{w}_k} = (\hat{y}_k - y_k)\mathbf{x}$$

Summing over all N data points gives the gradient descent update:

$$\mathbf{w}_k := \mathbf{w}_k - \alpha \sum_{i=1}^{N} (\hat{y}_{ik} - y_{ik})\mathbf{x}_i$$

Look Familiar?

- **Linear Regression (OLS):** $\nabla_\theta J_i = (\theta^T \mathbf{x} - y)\mathbf{x} = (\hat{y} - y)\mathbf{x}$
- **Logistic Regression:** $\nabla_\theta J_i = (\sigma(\theta^T \mathbf{x}) - y)\mathbf{x} = (\hat{y} - y)\mathbf{x}$
- **Softmax Regression:** $\nabla_{\mathbf{w}_k} J_i = (\text{softmax}(\mathbf{W}\mathbf{x})_k - y_k)\mathbf{x} = (\hat{y}_k - y_k)\mathbf{x}$

The gradient has this general pattern: **(Prediction** $-$ **Target)** $\times$ **Input**! It is not coincidence this happened.

> Hold on to this thought until we get to neural networks.

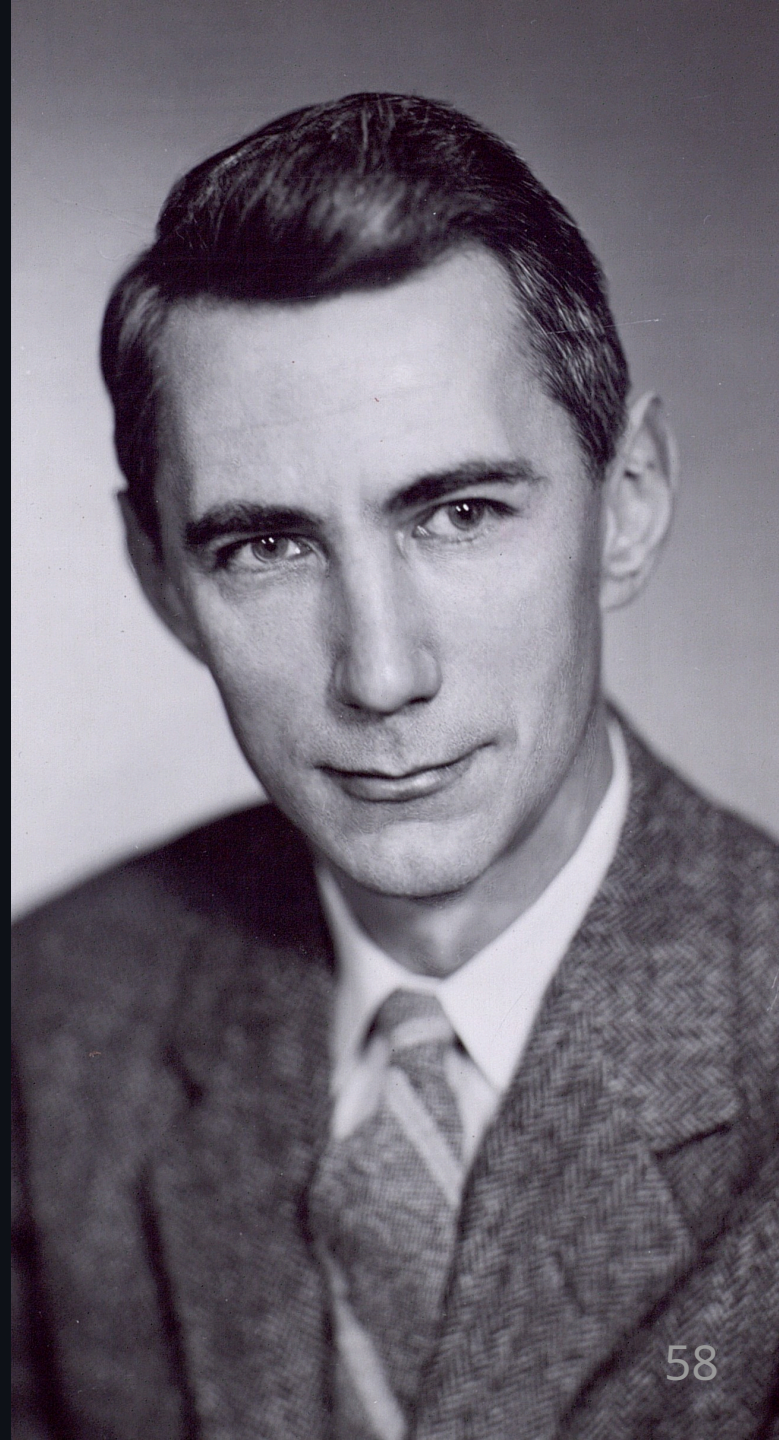
Information Theory: Surprise

Why is this called "Cross-Entropy"?

This brings us to information theory and how we interpret our model outputs. Similarly to the probabilistic framework, this will give us a new perspective.

A Mathematical Theory of Communication

Claude E. Shannon. The Bell System Technical Journal 1948-07: Vol 27 Iss 3. AT & T Bell Laboratories. 1948-07-01.



How Much Information Does an Event Carry?

Imagine hearing two news reports:

1. *"The sun rose this morning."* $\rightarrow$ Completely expected. Zero surprise.
2. *"A massive meteor landed in Hyde Park!"* $\rightarrow$ Highly improbable! Enormous surprise!

Shannon formalised this intuition as **Information Content** (or **Surprise**):

$$I(x) = -\log_2 P(x) = \log_2 \frac{1}{P(x)} \quad (\text{measured in bits})$$

- If $P(x) = 1 \implies I(x) = -\log_2(1) = 0$ bits (no new information).
- If $P(x) \to 0 \implies I(x) \to \infty$ bits (infinite information / shock).

Shannon Entropy

What is the **average surprise** you expect when sampling from a distribution P?

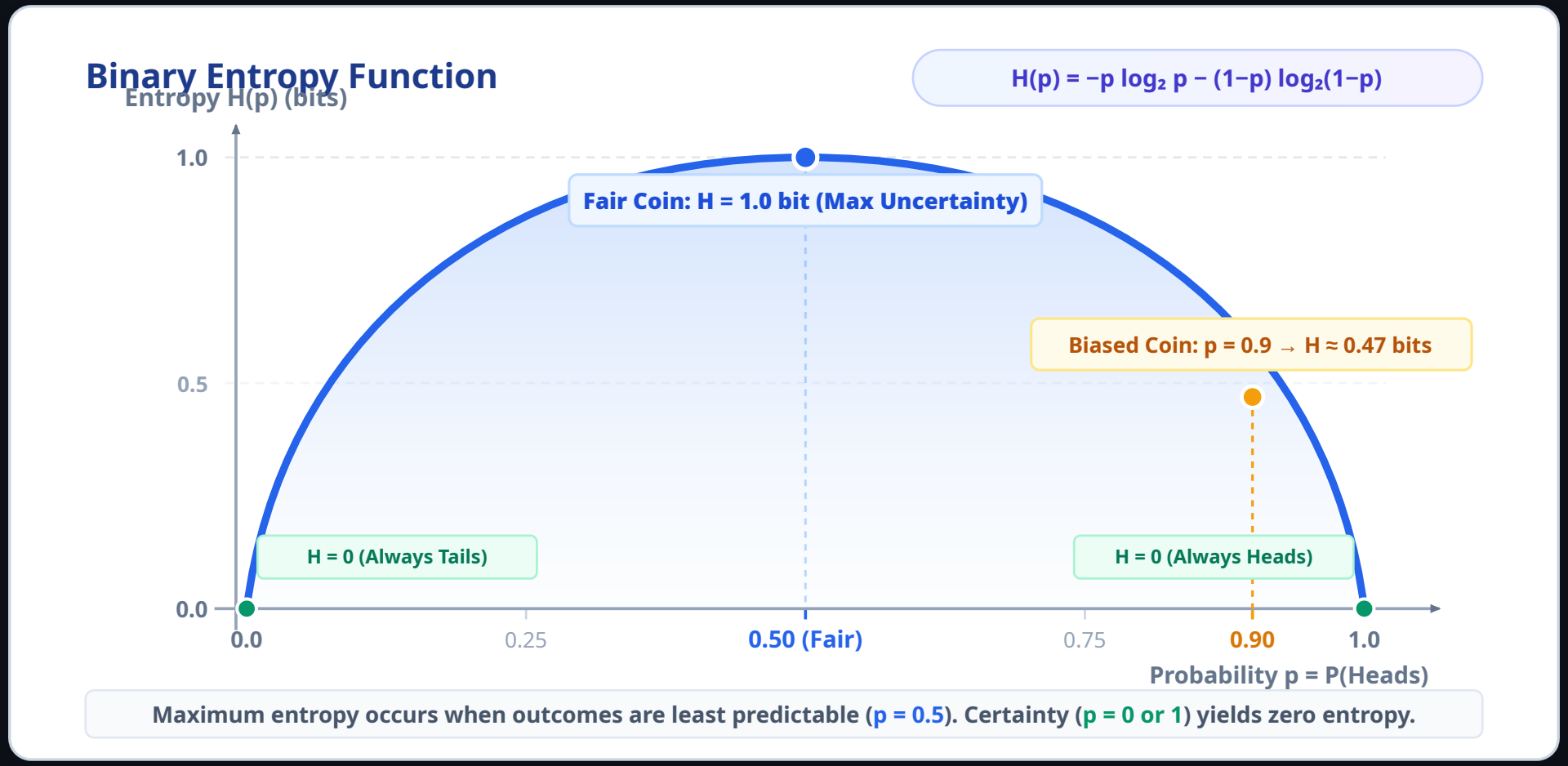
This expected surprise is the **Shannon Entropy** $H(P)$:

$$H(P) = \mathbb{E}_{x \sim P}[I(x)] = -\sum_{k=1}^{K} P(k) \log_2 P(k)$$

Physical & Computational Meaning

- **Measure of Uncertainty:** High entropy means outcomes are uniform and unpredictable; zero entropy means the outcome is completely deterministic.
- **Theoretical Compression Limit:** Shannon's source coding theorem shows that $H(P)$ is the absolute minimum number of bits needed, on average, to encode outcomes from P without loss.

Entropy in Action: Binary Entropy Curve



Fair vs. Biased Coin

- **Fair Coin ($p = 0.5$): Maximum Uncertainty**
  - Surprise: $I(\mathrm{H}) = -\log_2(0.5) = 1 \text{ bit}$, $I(\mathrm{T}) = 1 \text{ bit}$.
  - Expected Entropy: $H = 1.0 \text{ bit/flip}$. Unpredictable and incompressible: strictly requires 1 full bit per flip to store or transmit.
- **Biased Coin ($p = 0.9$): Low Uncertainty**
  - Common outcome (H): $I(\mathrm{H}) = -\log_2(0.9) \approx 0.15 \text{ bits}$ (low surprise, little new information).
  - Rare outcome (T): $I(\mathrm{T}) = -\log_2(0.1) \approx 3.32 \text{ bits}$ (high surprise!).
  - Expected Entropy: $H = 0.9(0.15) + 0.1(3.32) \approx 0.47 \text{ bits/flip}$.
  - Highly compressible: on average, takes less than half a bit per flip!
- **Deterministic Coin ($p = 1.0$): Zero Uncertainty**
  - Surprise: $I(\mathrm{H}) = -\log_2(1.0) = 0 \text{ bits} \implies H = 0.0 \text{ bits/flip}$.

Visual and modern deep dive by 3B1B

There is a fantastic playlist by 3B1B on Information Theory. I would recommend you watch it! Especially for visual learners who find the mathematical derivations challenging.

- Reinventing Entropy | Compression is Intelligence Part 1
- But what is cross-entropy? | Compression is Intelligence Part 2

A Note on Logarithms: Bits vs Nats

You will see both base 2 and base e used across literature:

- **Information Theory (Bits):** Shannon originally defined entropy using $\log_2$:

$$I(x) = -\log_2 P(x) \quad (\text{measured in } \textbf{bits} \text{ / } \textbf{shannons})$$

- **Machine Learning (Nats):** In ML, we almost always use the natural logarithm ($\ln$, base e):

$$H(P, Q) = -\sum_{k=1}^{K} y_k \ln \hat{y}_k \quad (\text{measured in } \textbf{nats})$$

  Because $\frac{d}{dz}\ln(z) = \frac{1}{z}$, base e is preferred.

- **Equivalence:** Since $\log_2(z) = \frac{\ln(z)}{\ln 2} \approx 1.443 \ln(z)$, the two differ only by a constant positive multiplier. Minimising nats is strictly identical to minimising bits.

Cross-Entropy & KL Divergence

Suppose nature generates data from the **true distribution** P, but we believe nature follows our **model's distribution** Q denoted as $\hat{\mathbf{y}}$.

- **Cross-Entropy $H(P, Q)$:** The average code length needed to transmit events from P using an optimal code designed for Q:

$$H(P, Q) = -\sum_{k=1}^{K} P(k) \log Q(k) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- **KL Divergence $D_{\mathrm{KL}}(P \parallel Q)$:** The "wasted bits" (inefficiency) incurred by using distribution Q instead of true P:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{k=1}^{K} P(k) \log \frac{P(k)}{Q(k)} = H(P, Q) - H(P)$$

The Fundamental Link

Since the training labels P are fixed, the true entropy $H(P)$ is a constant! Therefore:

$$\arg\min_{Q} H(P, Q) \equiv \arg\min_{Q} D_{\mathrm{KL}}(P \parallel Q) \equiv \arg\max_{Q} L(\mathbf{W})$$

Minimising Cross-Entropy $\equiv$ Minimising KL Divergence $\equiv$ Maximum Likelihood Estimation.

The Grand Unification

We have now journeyed across the statistical foundations of supervised learning:

Model	Target / Noise	Loss Function	Information / Likelihood View
OLS	$y \in \mathbb{R}$, Gaussian Noise	Mean Squared Error	MLE under $\mathcal{N}(\mu, \sigma^2)$
Logistic	$y \in \{0, 1\}$, Bernoulli	Binary Cross-Entropy	MLE under $\text{Bernoulli}(p)$
Softmax	$y \in \{1, \ldots, K\}$, Categorical	Categorical Cross-Entropy	Min $D_{\text{KL}}(P \parallel Q)$ / Categorical MLE

> There are bigger unifications than this but it is out of scope for this course and depends on how you want to specialise across statistics, information theory or probabilistic views.

But wait... Remember XOR?

We have built linear regression, logistic regression, and softmax regression. All mathematically sound, all derived from first statistical principles.

Now, take a careful look at the **decision boundary** of logistic regression:

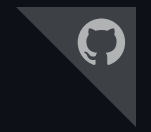
$$P(y=1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) = 0.5 \iff \mathbf{w}^T\mathbf{x} + b = 0$$

The decision boundary is strictly a **flat hyperplane**:

- In 2D: a straight line ($w_1 x_1 + w_2 x_2 + b = 0$).
- In 3D: a flat two-dimensional plane.
- In d-D: a $(d-1)$-dimensional hyperplane.

> Can this separate the XOR function? **No!** We are trapped at the exact same geometric wall that Minsky and Papert raised against the Perceptron in 1969.

Playground (Remove Hidden Layers)



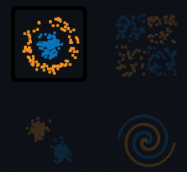
(https://gi

Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,000 | Learning rate 0.03 | Activation Tanh | Regularization None | Regularization rate 0 | Problem type Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 50%

FEATURES

Which properties do you want to feed in?

X_1
X_2
X_1^2

2 HIDDEN LAYERS

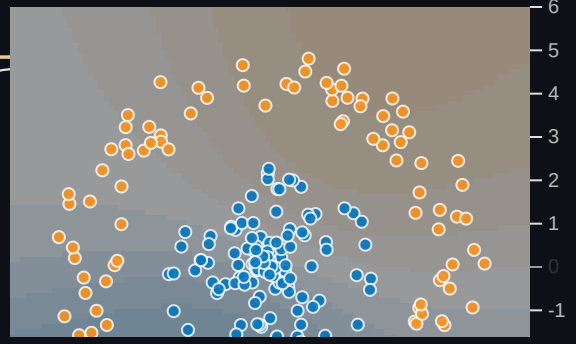
4 neurons

2 neurons

The outputs are mixed with varying

OUTPUT

Test loss 0.511
Training loss 0.510



The Escape: Basis Expansion

If our statistical model is linear in its parameters, how can we possibly learn non-linear boundaries?

- The model must be linear in its **parameters $\mathbf{w}$**, but it does **not** have to be linear in the original **input features $\mathbf{x}$**.

Let us define an explicit non-linear feature mapping:

$$\phi : \mathbb{R}^d \to \mathbb{R}^D \quad \text{where } D > d$$

Instead of feeding $\mathbf{x}$ directly, we feed the transformed feature vector $\phi(\mathbf{x})$:

$$y = \sigma(\mathbf{w}^T \phi(\mathbf{x}) + b)$$

We project the data into a higher-dimensional space where previously entangled points become **linearly separable**.

Cracking XOR (finally)

Let the input be $\mathbf{x} = [x_1, x_2]^T \in \{0, 1\}^2$:

Now, augment the input with a simple polynomial **interaction term**:

$$\phi(\mathbf{x}) = [x_1,\ x_2,\ x_1 x_2]^T \in \mathbb{R}^3$$

Notice what happens to the third feature value for $(1, 1)$:

- For $(0, 0) \rightarrow [0, 0, 0]$
- For $(1, 0) \rightarrow [1, 0, 0]$
- For $(0, 1) \rightarrow [0, 1, 0]$
- For $(1, 1) \rightarrow [1, 1, \mathbf{1}] \leftarrow$ *lifted off the base plane!*

Cracking XOR: The Solution

With this new feature space, let our parameter vector be:

$$\mathbf{w} = [1,\ 1,\ -2]^T, \quad b = -0.5$$

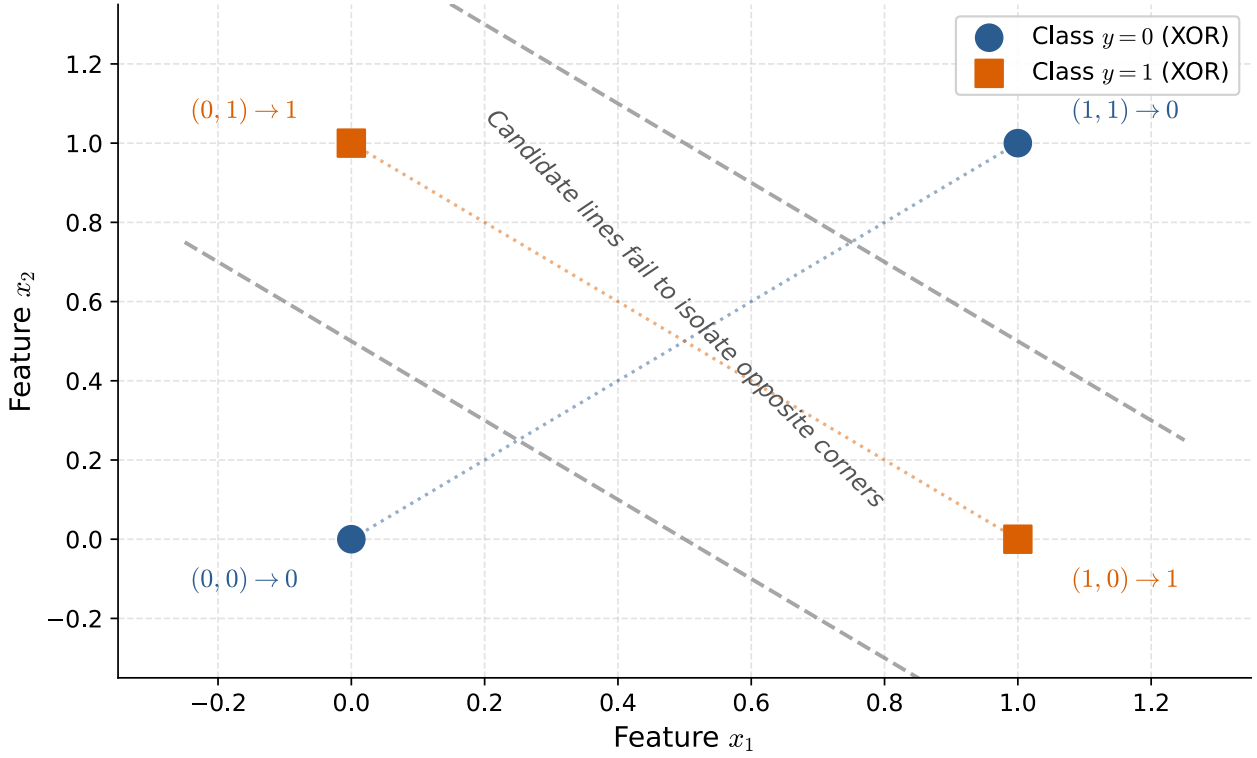
Let us evaluate the linear score $z = \mathbf{w}^T \phi(\mathbf{x}) + b = x_1 + x_2 - 2x_1x_2 - 0.5$:

- **Sample** $(0, 0)$: $0 + 0 - 2(0) - 0.5 = \mathbf{-0.5} < 0 \implies \hat{y} = 0$ *(Correct)*
- **Sample** $(1, 0)$: $1 + 0 - 2(0) - 0.5 = \mathbf{+0.5} > 0 \implies \hat{y} = 1$ *(Correct)*
- **Sample** $(0, 1)$: $0 + 1 - 2(0) - 0.5 = \mathbf{+0.5} > 0 \implies \hat{y} = 1$ *(Correct)*
- **Sample** $(1, 1)$: $1 + 1 - 2(1) - 0.5 = \mathbf{-0.5} < 0 \implies \hat{y} = 0$ *(Correct)*

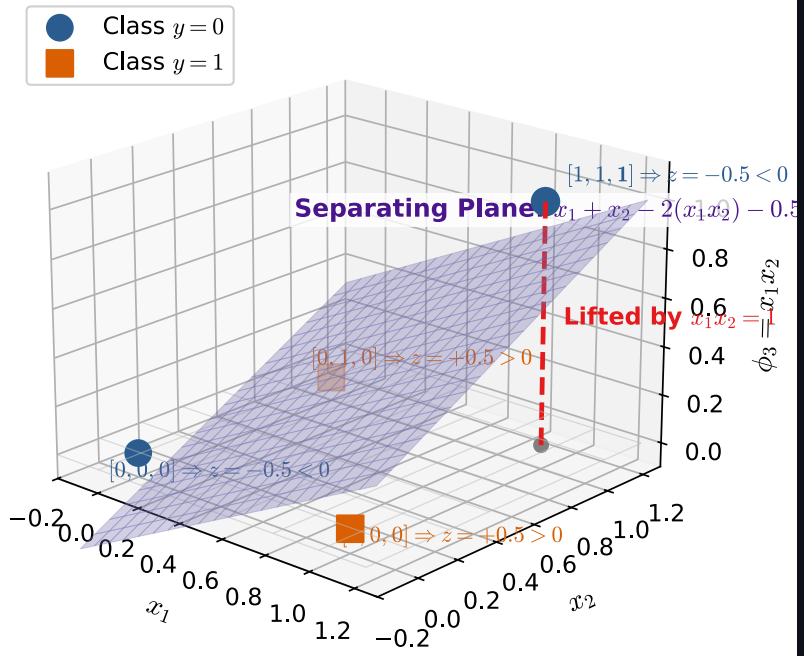
The non-linear interaction term x_1x_2 pulled the point $(1, 1)$ into an orthogonal dimension, allowing a flat plane to **slice** cleanly between the classes.

Cracking XOR: Basis Expansion $\phi(\mathbf{x}) = [x_1,\ x_2,\ x_1x_2]^T$ with $\mathbf{w} = [1,\ 1,\ -2]^T,\ b = -0.5$

1. Original Input Space: $\mathbf{x} = [x_1, x_2]^T$
(XOR is NOT linearly separable)



2. Expanded Space: $\phi(\mathbf{x}) = [x_1, x_2, x_1x_2]^T$
(Linearly Separated by Hyperplane $\mathbf{w}^T\phi(\mathbf{x}) + b = 0$)



General Polynomial Basis Expansion

We can extend this principle to arbitrary degree M. For a scalar input $x \in \mathbb{R}$:

$$\phi(x) = [1,\ x,\ x^2,\ x^3,\ \ldots,\ x^M]^T \in \mathbb{R}^{M+1}$$

Our regression model becomes:

$$f(x) = \mathbf{w}^T \phi(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \cdots + w_M x^M$$

The Crucial Property

Because $f(x)$ remains **strictly linear with respect to the weights $\mathbf{w}$**:

- All previous least-squares derivations apply without modification.
- We simply replace the design matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ with the expanded matrix $\mathbf{\Phi} \in \mathbb{R}^{N \times D}$:

$$\mathbf{\Phi}_{ij} = \phi_j(\mathbf{x}_i) \implies \mathbf{w}^* = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{y}$$

And for logistic regression, the loss function works like before.

The Curse of Combinatorics

If polynomial expansions give us non-linear power, why not expand everything to degree 20?

For an input vector $\mathbf{x} \in \mathbb{R}^d$ expanded to polynomial degree M, the number of features is:

$$D = \binom{d + M}{M} = \frac{(d + M)!}{d!\, M!}$$

Consider a modest feature space:

- A small greyscale icon of size 16×16 pixels has $d = 256$ features.
- If we want pairwise interactions ($M = 2$):

$$D = \binom{256 + 2}{2} = \frac{258 \times 257}{2} = 33,153 \text{ features}$$

> Hand-crafting basis functions goes BOOM 💥. Forget vision, audio, or language…

SVMs & The Kernel Trick

Before deep learning, how did machine learning deal with the combinatorial explosion of polynomial expansions?

In the 1990s and 2000s, the dominant paradigm was the **Support Vector Machine (SVM)** and the **Kernel Trick**:

- **The Idea:** Many algorithms (including linear regression and SVMs) can be expressed purely in terms of inner products between data points: $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$.
- **The Kernel Trick:** Compute this inner product implicitly using a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ in original input space, without ever calculating the massive vector $\phi(\mathbf{x})$ directly.

> Kernel methods scale with number of samples N. For modern datasets with millions of samples, this is intractable. Deep learning eventually destroyed all this...

The Danger of High Capacity

Even when we can compute the expansion, should we?

Suppose we collect N noisy observations from an underlying natural phenomenon:

$$y = f_{\text{true}}(x) + \epsilon$$

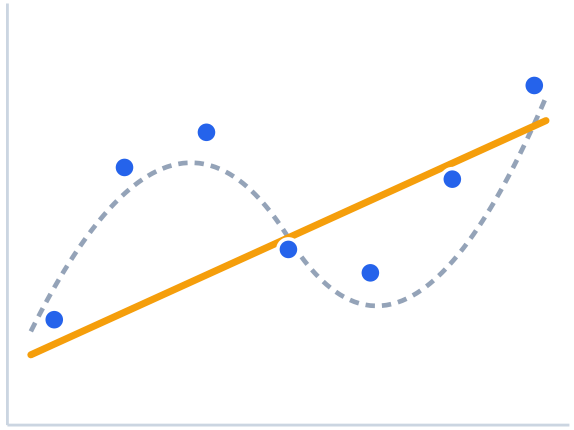
Let us fit polynomials of varying degree M to this data:

1. **Degree $M = 1$ (Underfitting):** Too rigid. Cannot capture curvature.
2. **Degree $M = 3$ (Appropriate Fit):** Balances flexibility with simplicity. Captures the true generative pattern.
3. **Degree $M = 15$ (Overfitting):** Excess capacity. Snaking through every single noisy sample with zero training error, but exploding into wild oscillations between points.

Degree 1 (Linear)

Underfitting

Model is too rigid / high bias



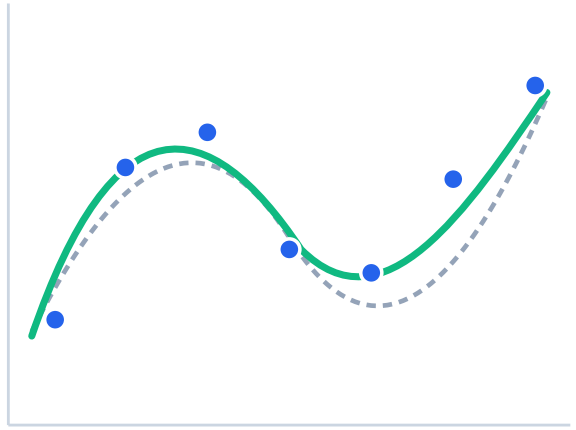
Train Error: High **Test Error: High**

Fails to capture underlying structure

Degree 3 (Cubic)

Just Right

Optimal complexity / generalises



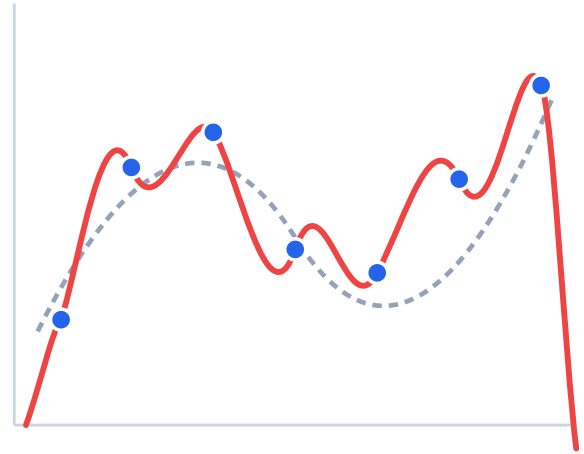
Train Error: Low **Test Error: Low (Best)**

Captures signal, ignores random noise

Degree 15 (Complex)

Overfitting

Memorises noise / high variance



Train Error: ≈ 0.0 **Test Error: Catastrophic**

Violent oscillations explode on new inputs

Memorisation vs Generalisation

This brings us to the foundational objective of machine learning:

$$\textbf{Machine Learning is NOT Optimisation!}$$

- **Pure Optimisation:** Minimise the empirical risk / loss on observed data:

$$\min_{\mathbf{w}} J_{\text{train}}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}_i; \mathbf{w}), y_i)$$

- **Machine Learning:** Minimise the expected risk / loss on **unseen future data** drawn from the underlying distribution $\mathcal{D}$:

$$R_{\text{true}}(f) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathcal{L}(f(\mathbf{x}), y)]$$

A model with infinite capacity (e.g. high degree polynomial or kNN) can achieve $J_{\text{train}} = 0$ by memorising the training data and any noise in it, while going BOOM 💥 in the real world.

All Possible Datasets

Assume the true data-generating process is:

$$y = f(\mathbf{x}) + \epsilon, \quad \text{where } \mathbb{E}[\epsilon] = 0 \text{ and } \mathrm{Var}(\epsilon) = \sigma^2$$

Suppose we collect a training set $\mathcal{D}$ and train an estimator $\hat{f}(\mathbf{x}; \mathcal{D})$.

What is the expected squared test error at a fixed query point $\mathbf{x}$, averaged across all possible training sets $\mathcal{D}$ and test noise ϵ?

$$\text{Expected Error} = \mathbb{E}_{\mathcal{D},\epsilon}\left[(y - \hat{f}(\mathbf{x}))^2\right]$$

A Useful Probability Identity

Recall from introductory probability that for any random variable Z:

$$\text{Var}(Z) = \mathbb{E}[Z^2] - (\mathbb{E}[Z])^2$$

Rearranging this gives a clean shortcut for the expected squared value:

$$\mathbb{E}[Z^2] = (\mathbb{E}[Z])^2 + \text{Var}(Z)$$

Key Idea: Let our random error at query point $\mathbf{x}$ be $Z = y - \hat{f}$.

$$\mathbb{E}_{\mathcal{D},\epsilon}\left[(y - \hat{f})^2\right] = \left(\mathbb{E}_{\mathcal{D},\epsilon}[y - \hat{f}]\right)^2 + \text{Var}_{\mathcal{D},\epsilon}(y - \hat{f})$$

Step 1: Mean of the Error

Substitute $y = f + \epsilon$ and apply linearity of expectation:

$$\begin{aligned}\mathbb{E}_{\mathcal{D},\epsilon}[y - \hat{f}] &= \mathbb{E}_{\mathcal{D},\epsilon}[f + \epsilon - \hat{f}] \\ &= f + \mathbb{E}_{\epsilon}[\epsilon] - \mathbb{E}_{\mathcal{D}}[\hat{f}]\end{aligned}$$

Since f is deterministic at $\mathbf{x}$ and the noise has zero mean ($\mathbb{E}_{\epsilon}[\epsilon] = 0$):

$$\mathbb{E}_{\mathcal{D},\epsilon}[y - \hat{f}] = f - \mathbb{E}_{\mathcal{D}}[\hat{f}] = \mathbf{Bias}(\hat{f})$$

Step 2: Variance of the Error

Now find the variance of $Z = y - \hat{f} = f + \epsilon - \hat{f}$:

$$\mathrm{Var}_{\mathcal{D},\epsilon}(y - \hat{f}) = \mathrm{Var}_{\mathcal{D},\epsilon}(f + \epsilon - \hat{f})$$

1. f is a constant, so $\mathrm{Var}(f) = 0$.
2. Test noise ϵ is **independent** of the training set $\mathcal{D}$ (and thus independent of $\hat{f}$).
3. Using $\mathrm{Var}(A - B) = \mathrm{Var}(A) + (-1)^2\mathrm{Var}(B) = \mathrm{Var}(A) + \mathrm{Var}(B)$ for independent A, B.

$$\mathrm{Var}_{\mathcal{D},\epsilon}(y - \hat{f}) = \mathrm{Var}_{\epsilon}(\epsilon) + \mathrm{Var}_{\mathcal{D}}(\hat{f}) = \sigma^2 + \mathrm{Var}_{\mathcal{D}}(\hat{f})$$

Step 3: Putting It All Together

Substitute the **Mean** (Step 1) and **Variance** (Step 2) back into our identity:

$$\mathbb{E}_{\mathcal{D},\epsilon}\left[(y-\hat{f})^2\right] = \left(\mathbb{E}_{\mathcal{D},\epsilon}[y-\hat{f}]\right)^2 + \mathrm{Var}_{\mathcal{D},\epsilon}(y-\hat{f})$$

$$\mathbb{E}_{\mathcal{D},\epsilon}\left[(y-\hat{f})^2\right] = \underbrace{\left(f-\mathbb{E}_{\mathcal{D}}[\hat{f}]\right)^2}_{\mathrm{Bias}^2} + \underbrace{\mathrm{Var}_{\mathcal{D}}(\hat{f})}_{\mathrm{Variance}} + \underbrace{\sigma^2}_{\mathrm{Irreducible\ Error}}$$

- Bias^2: How far the average model $\mathbb{E}_{\mathcal{D}}[\hat{f}]$ is from the true function f.
- $\mathrm{Variance}$: How sensitive $\hat{f}$ is to the specific choice of training set $\mathcal{D}$.
- σ^2: Inherent noise in the target y that no model can predict.

The Bias-Variance Decomposition

You'll also see it written with $\mathbf{x}$ explicitly:

$$\mathbb{E}_{\mathcal{D},\epsilon}\left[(y-\hat{f}(\mathbf{x}))^2\right] = \underbrace{(f(\mathbf{x})-\mathbb{E}[\hat{f}(\mathbf{x})])^2}_{\textbf{Bias}^2} + \underbrace{\mathbb{E}_{\mathcal{D}}[(\hat{f}(\mathbf{x})-\mathbb{E}[\hat{f}(\mathbf{x})])^2]}_{\textbf{Variance}} + \underbrace{\sigma^2}_{\textbf{Noise}}$$

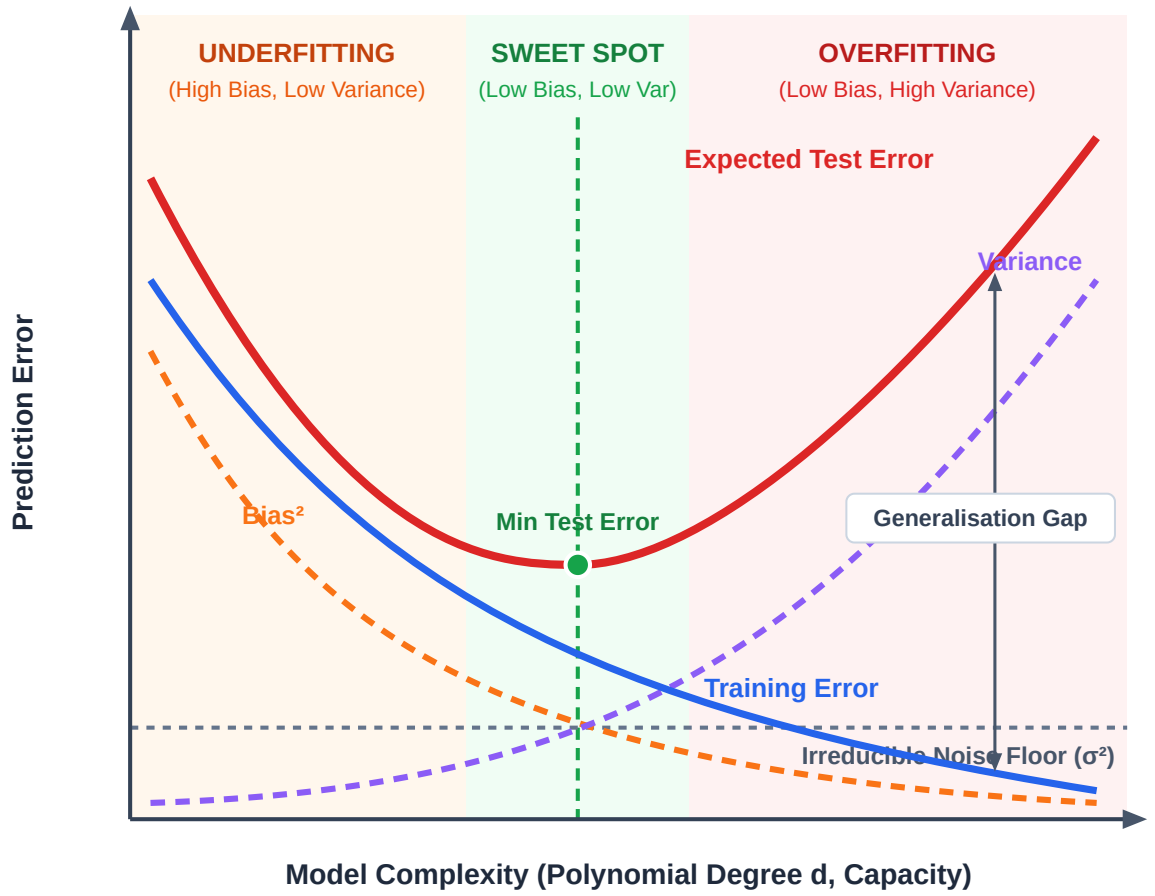
Let's try again

- Bias^2 **(Systematic Error):** How much the *average* model over all possible datasets deviates from the true underlying function.
- Variance **(Instability):** How much the model's predictions fluctuate when trained on different random samples from the same population.
- σ^2 **(Irreducible Noise):** Unavoidable variance in the measurement or universe. No model can eliminate this.

Overfitting & Underfitting

Regime	Bias^2	Variance	Train Error	Test Error	What's Happening?	Concrete Example
Underfitting	**High**	Low	High	High	Model is too simple to capture f; insensitive to the choice of $\mathcal{D}$.	Fitting a straight line ($d = 1$) to a sine wave.
Overfitting	Low	**High**	Very Low	High	Model captures f on average, but chases random noise ϵ in each specific $\mathcal{D}$.	Unregularized degree-20 polynomial on $N = 25$ noisy points.
Sweet Spot	Low	Low	Low	Low	Model capacity matches the complexity of f and dataset size $\lvert\mathcal{D}\rvert$.	Degree-3 polynomial or Ridge regression with tuned λ.

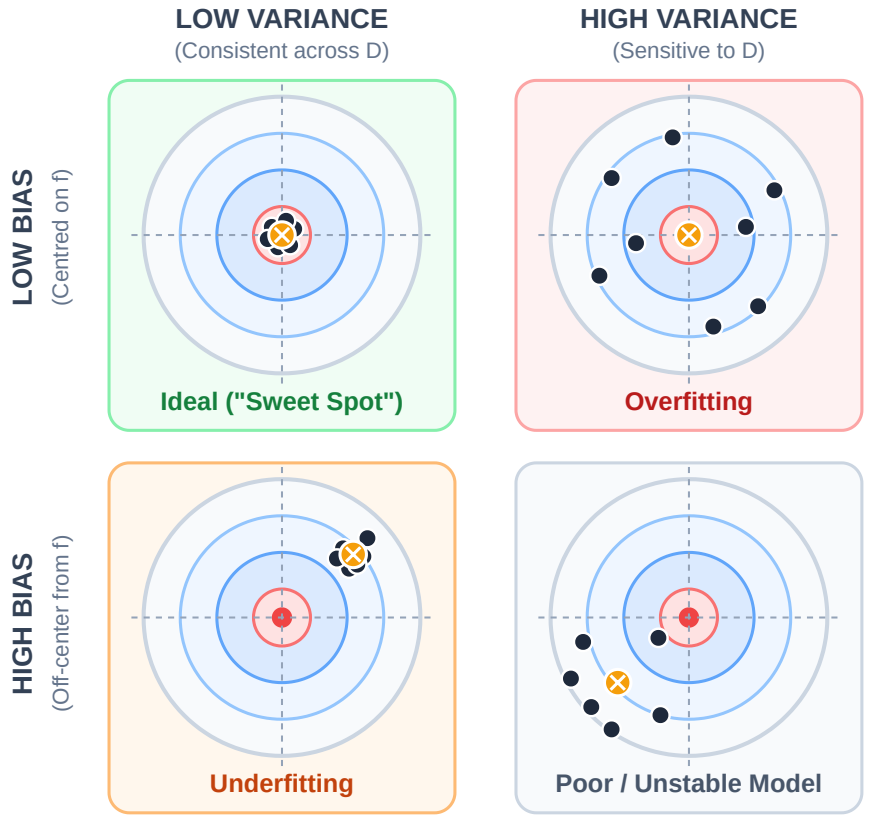
Bias-Variance Tradeoff & Generalisation Error



Expected Test Error = Bias² + Variance + σ² (Irreducible Noise)

Target Analogy Across Datasets D

Each dark dot is a model trained on a different sampled dataset D



True Target f(x) · Single Fit f_hat(x; D) · Average E[f_hat(x)]

How to reduce variance?

Our foundational goal: **minimise total expected test risk**.

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \sigma^2$$

When models have high capacity (e.g. high-degree polynomials or huge neural networks), variance dominates. How can we reduce variance?

1. **Collect more training data ($N \uparrow$):** As $N \to \infty$, variance shrinks towards zero. But acquiring more data is often expensive or impossible.
2. **Remove features / reduce degree ($d \downarrow$):** Eliminates variance, but risks underfitting (causing bias to spike).
3. **Constrain parameter magnitudes (Regularisation):** Retain all expressive features, but penalise large weights.

Taming Overfitting: Regularisation

When an $M = 15$ polynomial oscillates violently to pass through noisy samples, its weight vector behaves in a telltale way:

$$\mathbf{w} = [2.1,\ -4.8 \times 10^{5},\ +1.2 \times 10^{7},\ -8.3 \times 10^{6},\ \ldots]^{T}$$

The coefficients explode into **massive positive and negative values** that cancel each other out precisely on the training points, but diverge violently elsewhere.

$\rightarrow$ Regularisation

Penalise the model for having large weights:

$$\mathcal{L}_{\text{reg}}(\mathbf{w}) = \mathcal{L}_{\text{data}}(\mathbf{w}) + \lambda \cdot \Omega(\mathbf{w})$$

where $\lambda \geq 0$ is the **regularisation hyperparameter** controlling the trade-off between fitting the training data and keeping model capacity restricted.

L_2 Regularisation

In **Ridge Regression**, we penalise the squared Euclidean norm (L_2 norm) of the weight vector:

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \sum_{j=1}^{d} w_j^2$$

The total cost function is:

$$J_{\text{Ridge}}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

(Convention: We regularise the weights $w_1, \ldots, w_d$, but typically leave the bias term w_0 unpenalised so the model can freely shift the baseline.)

Ridge: Closed-Form Solution

Let us take the gradient of $J_{\text{Ridge}}(\mathbf{w})$ with respect to $\mathbf{w}$ and set it to zero:

$$\begin{aligned}\nabla_{\mathbf{w}} J_{\text{Ridge}}(\mathbf{w}) &= \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w} \\ &= \mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} + \lambda\mathbf{I}\mathbf{w} \\ &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\mathbf{w} - \mathbf{X}^T\mathbf{y} = \mathbf{0}\end{aligned}$$

Solving for $\mathbf{w}$:

$$\mathbf{w}^*_{\text{Ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

A small note

Recall that OLS broke down when $\mathbf{X}^T\mathbf{X}$ was non-invertible ($N < d$ or collinear features).

For any $\lambda > 0$, the matrix $(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})$ shifts all eigenvalues by $+\lambda$. It is **strictly positive definite** and **guaranteed to be invertible**. *But who cares...*

L_1 Regularisation

What if we penalise the sum of absolute values (L_1 norm) instead?

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{j=1}^{d} |w_j|$$

This gives **Lasso Regression** (*Least Absolute Shrinkage and Selection Operator*):

$$J_{\text{Lasso}}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha\|\mathbf{w}\|_1$$

- **Ridge (L_2):** Shrinks all weights smoothly towards zero, but rarely become exactly zero.
- **Lasso (L_1):** Drives many weights to **exactly zero** ($w_j = 0$).

> Lasso performs automatic **feature selection**, yielding **sparse models** that retain only the most explanatory inputs. In 2026, it is used in mechanistic interpretability of LLMs.

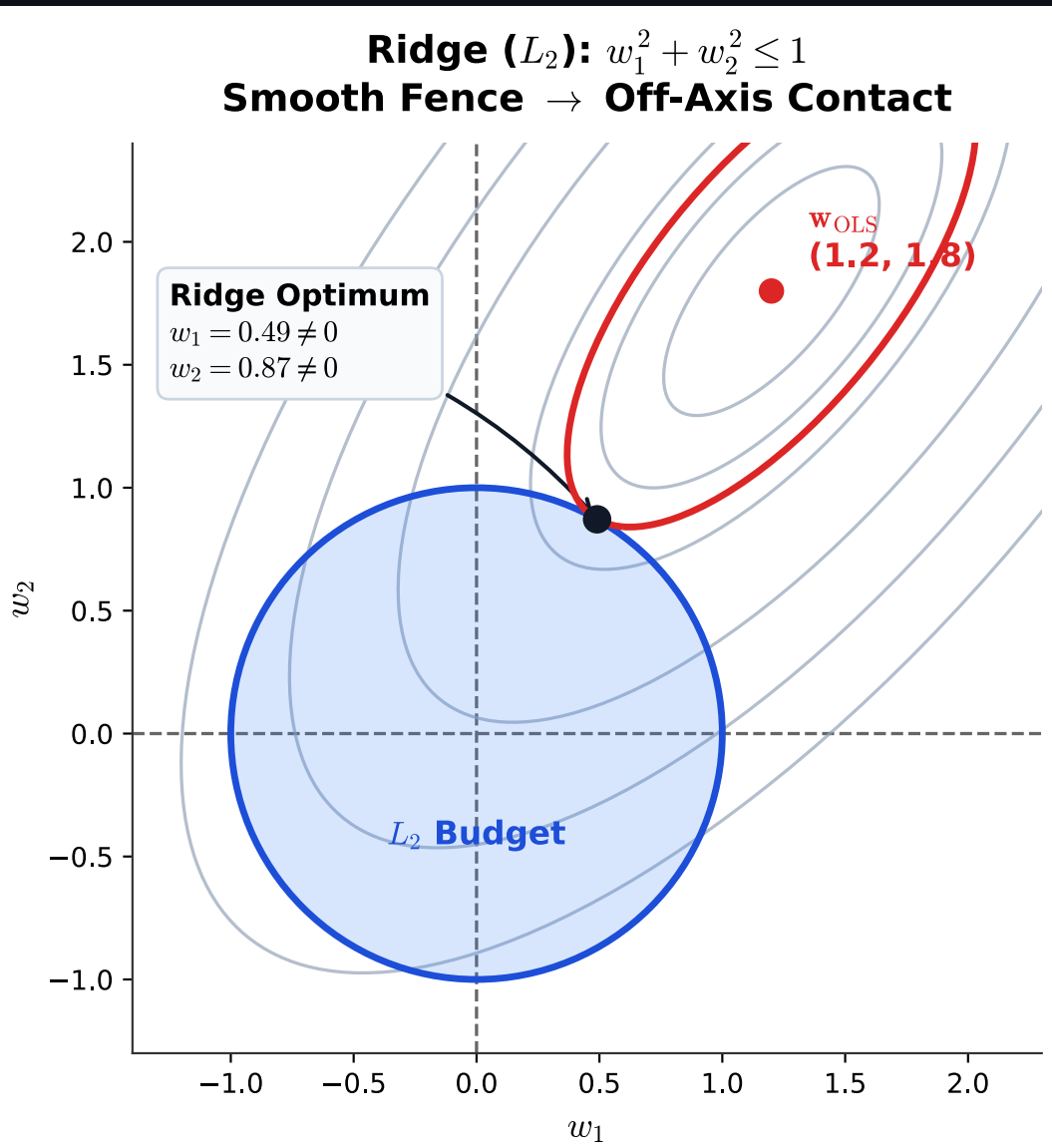
Ridge (L_2) vs. Lasso (L_1)

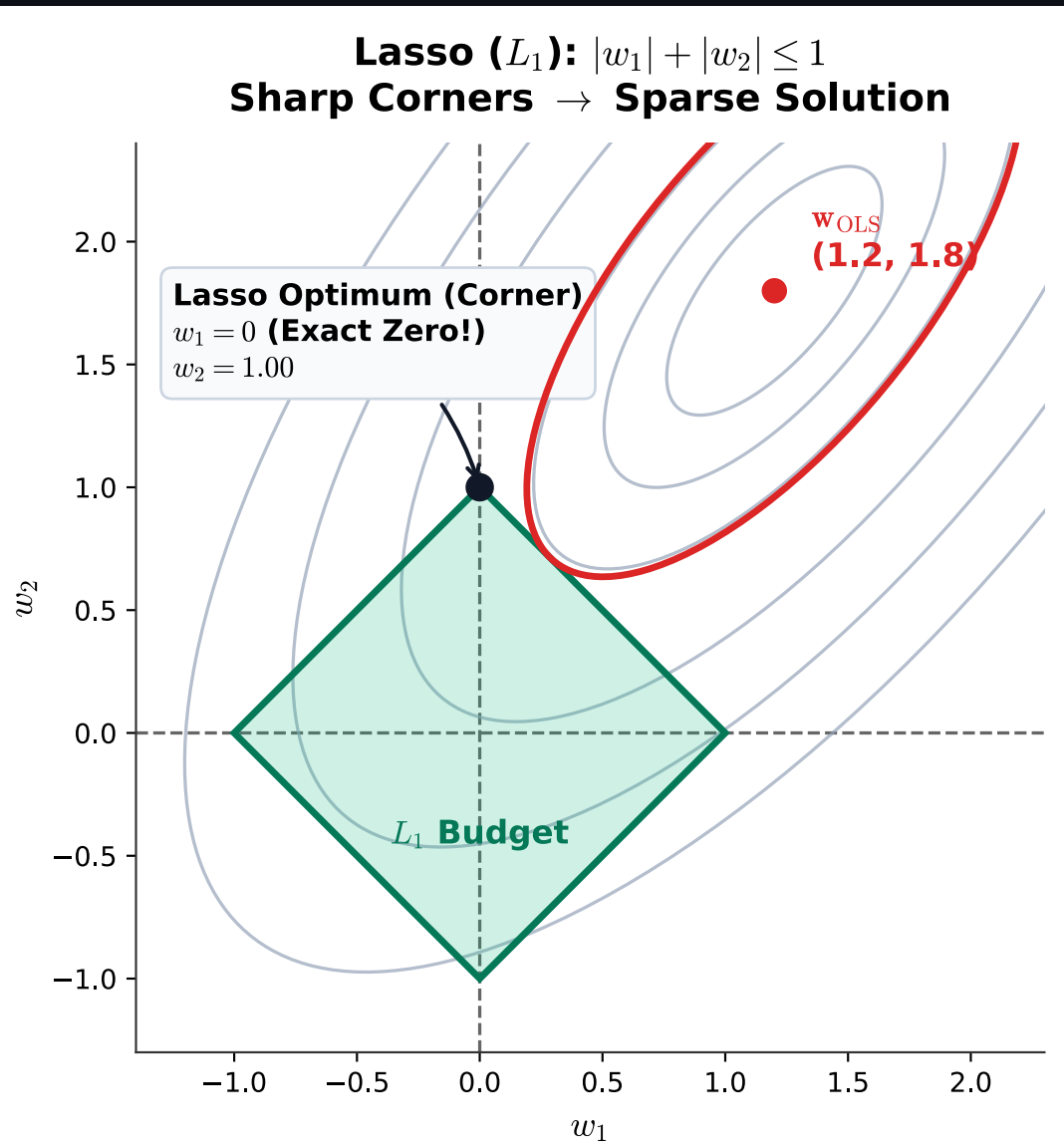
Why does L_1 produce exact zeros?

We effectively have a constrained optimisation problem with some budget C:

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \quad \text{subject to} \quad \Omega(\mathbf{w}) \leq C$$

- **Ridge (L_2 Circle: $w_1^2 + w_2^2 \leq C$):** Smooth boundary $\rightarrow$ tangential contact happens off-axis ($w_1, w_2 \neq 0$) with best solution w_{OLS}.
- **Lasso (L_1 Diamond: $|w_1| + |w_2| \leq C$):** Sharp corners sit directly on the axes (e.g., $w_1 = C, w_2 = 0$) $\rightarrow$ the ellipse hits a corner first, zeroing out w_2.


Ridge (L_2): $w_1^2 + w_2^2 \leq 1$
Smooth Fence $\rightarrow$ Off-Axis Contact
$\mathbf{w}_{\text{OLS}}$
(1.2, 1.8)
Ridge Optimum
$w_1 = 0.49 \neq 0$
$w_2 = 0.87 \neq 0$
L_2 Budget
w_1
w_2


Lasso (L_1): $|w_1| + |w_2| \leq 1$
Sharp Corners $\rightarrow$ Sparse Solution
$\mathbf{w}_{\text{OLS}}$
(1.2, 1.8)
Lasso Optimum (Corner)
$w_1 = 0$ (Exact Zero!)
$w_2 = 1.00$
L_1 Budget
w_1
w_2

The way I think about it

How hard does each penalty push a small weight (say, $w = 0.01$) towards zero?

Penalty Type	Penalty Term	Derivative (Push towards 0)	At $w = 0.01$	Behaviour near 0
Ridge (L_2)	$\frac{1}{2}w^2$	w	0.01	Push weakens as $w \to 0$
Lasso (L_1)	$\mid w \mid$	$\text{sign}(w)$	1.00	Constant push all the way to **exactly** 0

The Bayesian MAP Perspective

Can we understand Ridge and Lasso from a probabilistic perspective? *Because we're bored.*

Recall **Bayes' Theorem**, and apply to our model parameters $\mathbf{w}$:

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})\,p(\mathbf{w})}{p(\mathcal{D})} \propto \underbrace{p(\mathcal{D}|\mathbf{w})}_{\text{Likelihood}} \times \underbrace{p(\mathbf{w})}_{\text{Prior}}$$

- **MLE (Frequentist):** Finds the single $\mathbf{w}$ that maximises data likelihood:

$$\mathbf{w}_{\text{MLE}} = \arg\max_{\mathbf{w}} \log p(\mathcal{D}|\mathbf{w})$$

- **MAP (Bayesian):** Finds the mode of the posterior distribution, incorporating prior beliefs about $\mathbf{w}$:

$$\mathbf{w}_{\text{MAP}} = \arg\max_{\mathbf{w}} \left[\log p(\mathcal{D}|\mathbf{w}) + \log p(\mathbf{w})\right]$$

Ridge: MAP with a Gaussian Prior

Assume the standard Gaussian likelihood from linear regression:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \mathcal{N}(\mathbf{y}; \mathbf{X}\mathbf{w}, \sigma^2\mathbf{I})$$

Now place an independent zero-mean **Gaussian prior** on the weights: $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \tau^2\mathbf{I})$:

$$p(\mathbf{w}) = \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}\tau} \exp\left(-\frac{w_j^2}{2\tau^2}\right) = \left(\frac{1}{2\pi\tau^2}\right)^{d/2} \exp\left(-\frac{\|\mathbf{w}\|_2^2}{2\tau^2}\right)$$

Take the almighty log:

$$\log p(\mathbf{w}) = -\frac{d}{2}\log(2\pi\tau^2) - \frac{1}{2\tau^2}\|\mathbf{w}\|_2^2$$

Maximising the posterior $\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) + \log p(\mathbf{w})$ is equivalent to minimising:

$$J(\mathbf{w}) = \frac{1}{2\sigma^2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \frac{1}{2\tau^2}\|\mathbf{w}\|_2^2$$

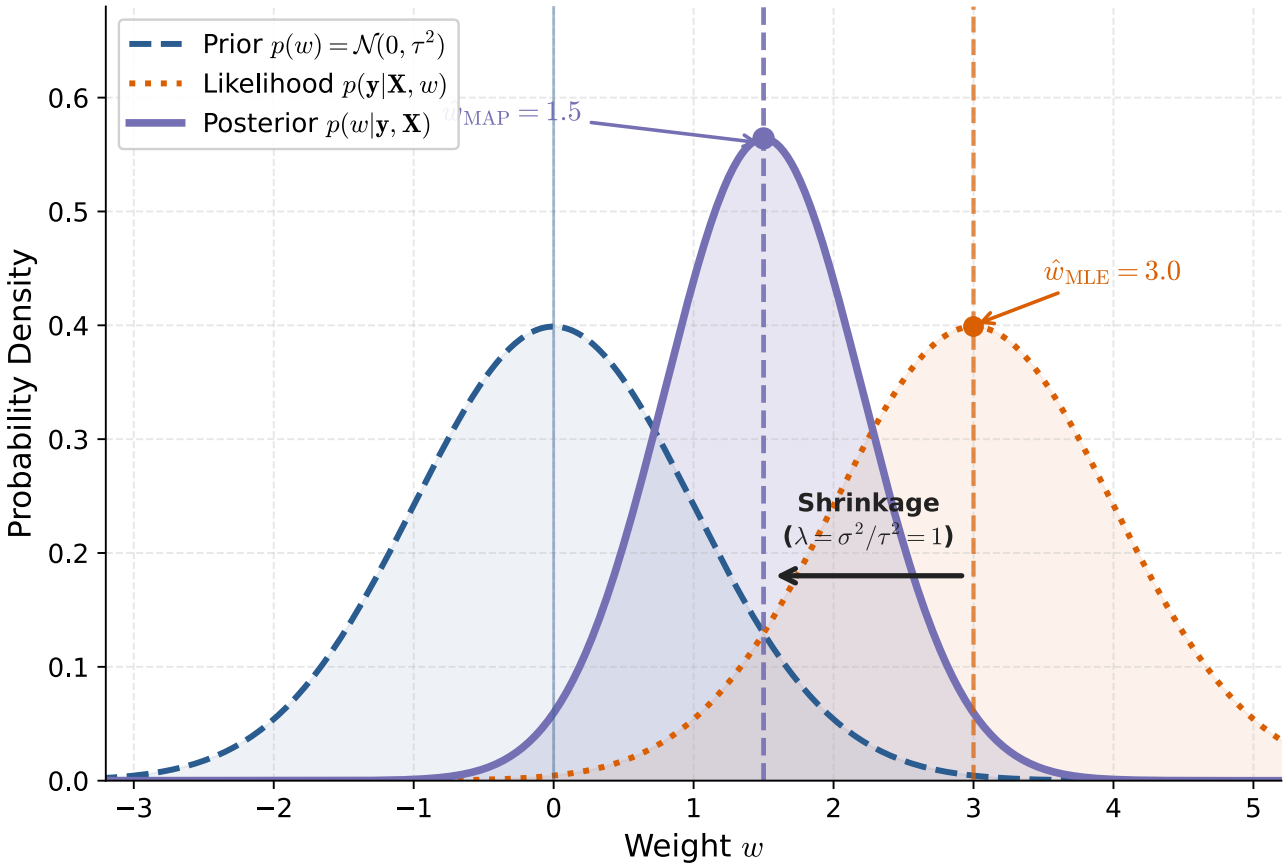
Multiplying through by σ^2 gives **Ridge Regression** with $\lambda = \frac{\sigma^2}{\tau^2}$ 😲

Wait da hell is $\lambda = \frac{\sigma^2}{\tau^2}$?

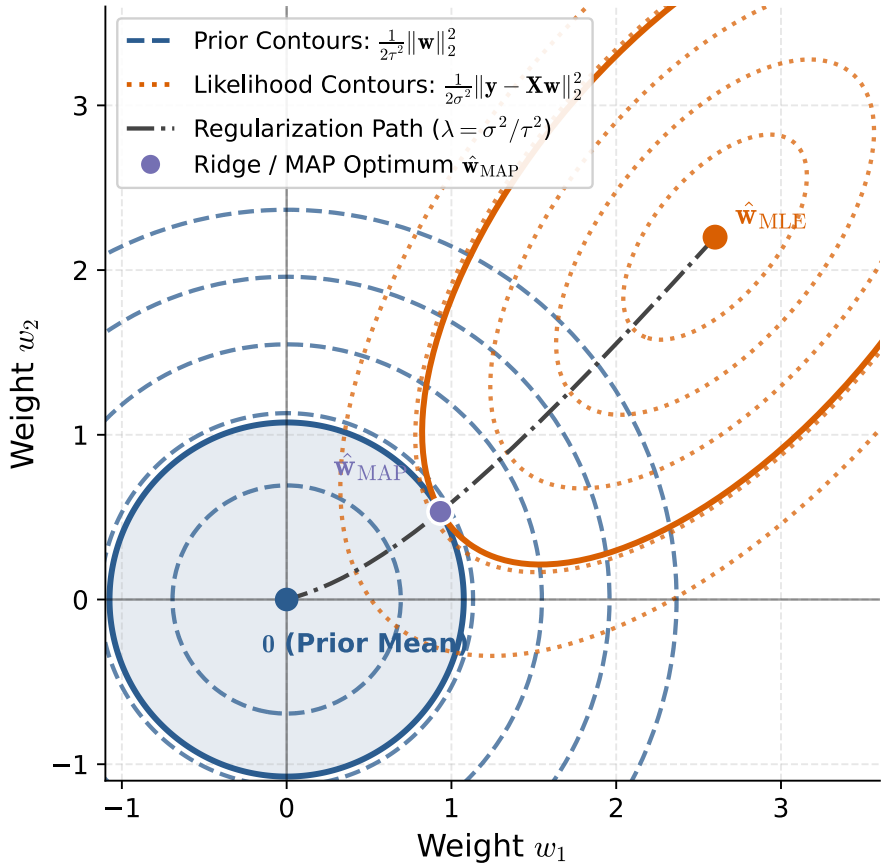
- High data noise ($\sigma^2 = 10$) or tight prior ($\tau^2 = 0.1$) $\implies$ Large $\lambda = 100$ (trust the prior, shrink $\mathbf{w}$).
- Low data noise ($\sigma^2 = 0.1$) or broad prior ($\tau^2 = 10$) $\implies$ Small $\lambda = 0.01$ (trust the data).

Ridge Regression as MAP Estimation: $\hat{\mathbf{w}}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left(\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \frac{\sigma^2}{\tau^2} \|\mathbf{w}\|_2^2 \right)$

1D View: Prior Pulls Likelihood Toward Zero



2D Weight Space: Contours of $J(\mathbf{w})$



Lasso: MAP with a Laplace Prior

I believe that most features are irrelevant, and weights should be concentrated at zero.

Laplace prior on the weights: $w_j \sim \text{Laplace}(0, b)$:

$$p(w_j) = \frac{1}{2b}\exp\left(-\frac{|w_j - 0|}{b}\right) \implies p(\mathbf{w}) = \left(\frac{1}{2b}\right)^d \exp\left(-\frac{\|\mathbf{w}\|_1}{b}\right)$$

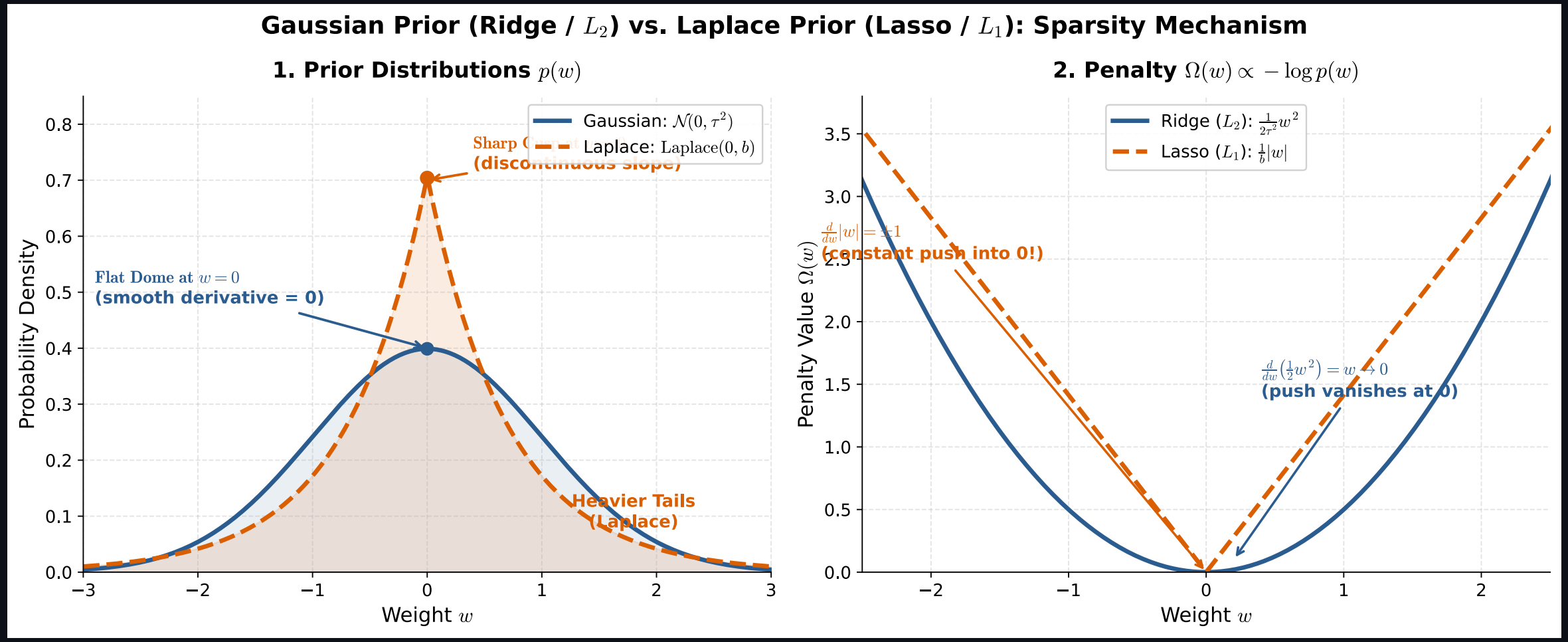
Take the almighty log:

$$\log p(\mathbf{w}) = -d\log(2b) - \frac{1}{b}\|\mathbf{w}\|_1$$

Maximising the log-posterior corresponds to minimising:

$$J(\mathbf{w}) = \frac{1}{2\sigma^2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \frac{1}{b}\|\mathbf{w}\|_1$$

Multiplying through by σ^2 gives **Lasso Regression** with $\alpha = \frac{\sigma^2}{b}$.


Gaussian Prior (Ridge / L_2) vs. Laplace Prior (Lasso / L_1): Sparsity Mechanism
1. Prior Distributions $p(w)$
Gaussian: $\mathcal{N}(0, \tau^2)$
Laplace: $\mathrm{Laplace}(0, b)$
Sharp C
(discontinuous slope)
Flat Dome at $w = 0$
(smooth derivative = 0)
Heavier Tails
(Laplace)
Probability Density
Weight w
2. Penalty $\Omega(w) \propto -\log p(w)$
Ridge (L_2): $\frac{1}{2\tau^2} w^2$
Lasso (L_1): $\frac{1}{b}|w|$
$\frac{d}{dw}|w| = \pm 1$
(constant push into 0!)
$\frac{d}{dw}\left(\frac{1}{2}w^2\right) = w \to 0$
(push vanishes at 0)
Penalty Value $\Omega(w)$
Weight w

The Regularisation Summary

Property	Ordinary Least Squares	Ridge Regression (L_2)	Lasso Regression (L_1)
Penalty $\Omega(\mathbf{w})$	None (0)	$\sum_{j=1}^{d} w_j^2$	$\sum_{j=1}^{d} \mid w_j \mid$
Bayesian Prior	Uniform (No prior)	Gaussian $\mathcal{N}(0, \tau^2\mathbf{I})$	Laplace $\mathrm{Laplace}(0, b)$
Weight Effect	Unconstrained, can explode	Shrunk smoothly towards 0	Sparse: driven to exact 0
Feature Selection	None	Retains all features	Automatic selection
Solution Method	$(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$	$(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$	Gradient descent
Collinearity Handling	Fails (Singular $\mathbf{X}^T\mathbf{X}$)	Robust (Always invertible)	Selects one feature arbitrarily

"Double Descent"

For a long time, the bias-variance tradeoff was king:

$$\text{Parameters } P \uparrow \implies \text{Variance} \uparrow \implies \text{Test Error} \to \infty$$

But modern deep neural networks have hundreds of billions of parameters, way more than training samples ($P \gg N$), and they generalise remarkably well.

Double Descent (Belkin et al., 2019)

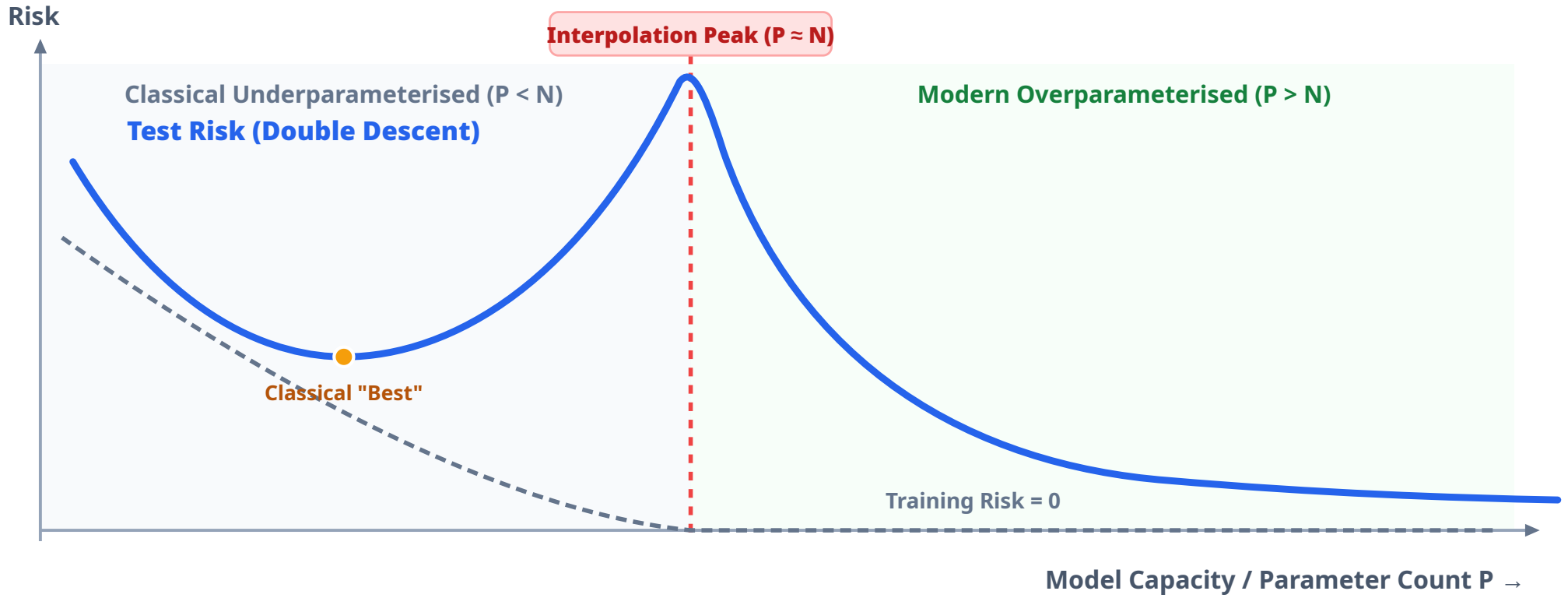
- **Classical Regime ($P < N$):** Test error follows the traditional U-shaped curve.
- **Interpolation Threshold ($P \approx N$):** The model has just enough capacity to interpolate all training points exactly.
- **Overparameterised Regime ($P > N$):** Test error **descends a second time**!

> *We do not formally examine double descent in this course, but it covers why the classical fear of overparameterisation does not destroy modern deep learning.*

The "Double Descent" Phenomenon

Belkin et al. (PNAS 2019)

When parameter count P exceeds sample count N, test error drops again



Belkin et al., Reconciling modern machine-learning practice and the classical bias–variance trade-off, PNAS, https://doi.org/10.1073/pnas.1903070116 (2019).

Where is our artificial brain?

Phew! I'm tired, you're tired of playing around with all this. Where is our artificial brain?

But

We are now only **1 step away from deep learning**. So close, yet so far...

I personally prefer things that are elegant rather than wrangled together to make it work. What did we experience so far?

Here is what elegant looks looks like: $e^{i\pi} + 1 = 0$

Appendix

AKA The Cabinet of Mysteries

$$
\begin{aligned}
\frac{d\sigma}{dz} &= \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{d}{dz}(1+e^{-z})^{-1} \\
&= -(1+e^{-z})^{-2} \cdot \frac{d}{dz}(1+e^{-z}) \\
&= -(1+e^{-z})^{-2} \cdot (-e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}} \cdot \left(\frac{1+e^{-z}-1}{1+e^{-z}}\right) \\
&= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}
$$

Deriving the Gradient: Softmax

To optimise $J(\mathbf{W})$ via gradient descent, we need $\frac{\partial J}{\partial z_k}$. But unlike sigmoid, every logit z_j affects **every** output probability $\hat{y}_i$ via the denominator $\sum_m e^{z_m}$.

We need to use the quotient rule:

$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

to handle the denominator:

$$\hat{y}_i = p_i = \text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{m=1}^{K} e^{z_m}}$$

Let:

$$\hat{y}_i = \frac{e^{z_i}}{S} \quad \text{where } S = \sum_{m=1}^{K} e^{z_m}$$

Case 1: $i = j$

$$\frac{\partial \hat{y}_i}{\partial z_i} = \frac{e^{z_i} S - e^{z_i} e^{z_i}}{S^2} = \frac{e^{z_i}}{S}\left(1 - \frac{e^{z_i}}{S}\right) = \hat{y}_i(1 - \hat{y}_i)$$

Case 2: $i \neq j$

$$\frac{\partial \hat{y}_i}{\partial z_j} = \frac{0 \cdot S - e^{z_i} e^{z_j}}{S^2} = -\frac{e^{z_i}}{S}\frac{e^{z_j}}{S} = -\hat{y}_i \hat{y}_j$$

Softmax Derivative Cont.

We can express both cases concisely using the Kronecker delta δ_{ij}:

$$\frac{\partial \hat{y}_i}{\partial z_j} = \hat{y}_i(\delta_{ij} - \hat{y}_j)$$

where δ_{ij} is:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

- If $i = j$: $\delta_{ii} = 1 \implies \hat{y}_i(1 - \hat{y}_i)$ (Case 1)
- If $i \neq j$: $\delta_{ij} = 0 \implies \hat{y}_i(0 - \hat{y}_j) = -\hat{y}_i\hat{y}_j$ (Case 2)

It unifies both terms into a single expression in a fancy way.

Now, using the multivariable chain rule to find $\frac{\partial J}{\partial z_k}$:

$$\begin{aligned}
\frac{\partial J}{\partial z_k} &= -\sum_{i=1}^{K} \frac{y_i}{\hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_k} \\
&= -\frac{y_k}{\hat{y}_k}\left[\hat{y}_k(1-\hat{y}_k)\right] - \sum_{i \neq k} \frac{y_i}{\hat{y}_i}\left[-\hat{y}_i \hat{y}_k\right] \\
&= -y_k(1-\hat{y}_k) + \sum_{i \neq k} y_i \hat{y}_k \\
&= -y_k + \hat{y}_k \underbrace{\sum_{i=1}^{K} y_i}_{=1} \\
&= \hat{y}_k - y_k
\end{aligned}$$

Counting Features

Why is the number of features up to degree M given by $\binom{d+M}{M}$?

Introduce bias variable as $x_0 = 1$. Now every term of degree $\leq M$ can be written as a product of **exact** degree M:

$$x_0^{k_0} x_1^{k_1} \dots x_d^{k_d} \quad \text{where} \sum_{i=0}^{d} k_i = M$$

Finding all features is equivalent to distributing M degree units into $d+1$ buckets:

- Example ($d = 2, M = 2$): $\bullet \mid \bullet \mid \longrightarrow x_0^1 x_1^1 x_2^0 = x_1$
- Total slots to fill: d dividers + M units = $d + M$ slots.
- Choosing which M out of $d + M$ slots:

$$D = \binom{d+M}{M} = \frac{(d+M)!}{d!\,M!}$$