LECTURE 4

LEARNING FEATURE REPRESENTATIONS

From Linear Regression to Multi-Layer Perceptron

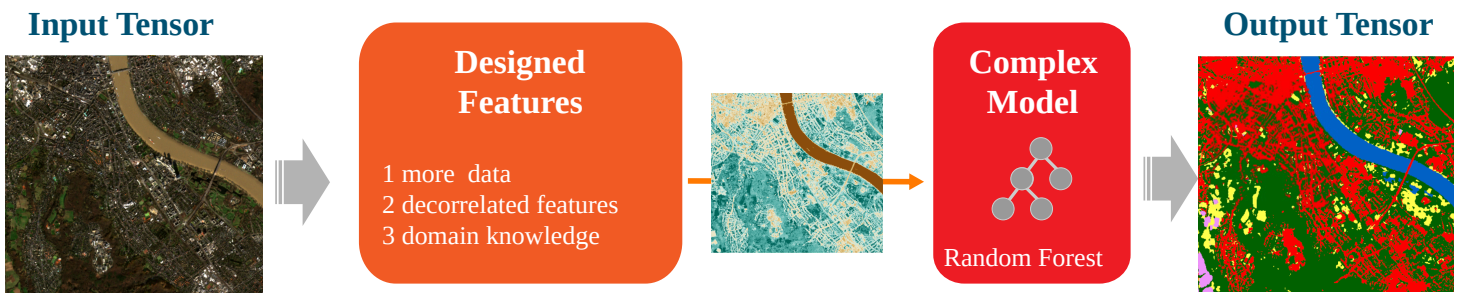
PRESS → OR SPACE.

© Marc Rußwurm
Licensed under **Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0)**.
You may share and adapt these slides for teaching, research, and other non-commercial educational purposes with attribution.
Commercial training, paid workshops, consulting seminars, or incorporation into commercial course products require prior permission.

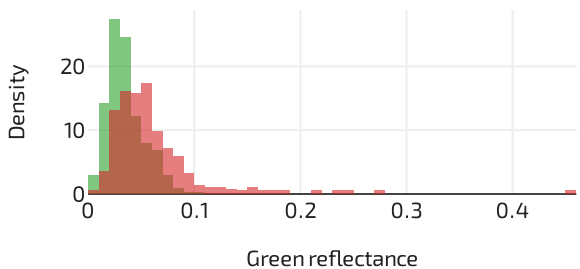
UNIVERSITÄT BONN

Recap Lecture 3 - Feature Design

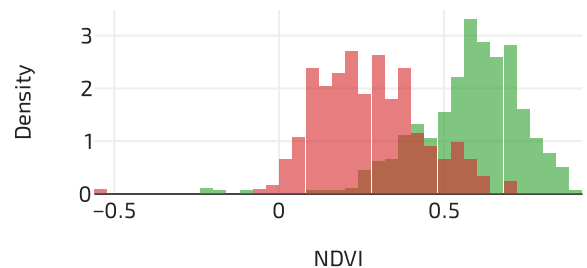
By Data, Decorrelation, and Domain Knowledge



Raw Band: Green



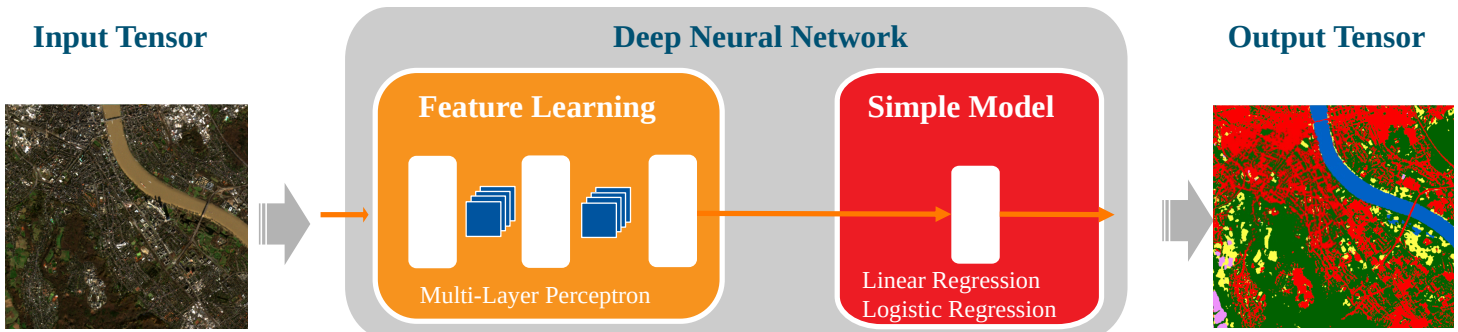
Engineered Feature: NDVI



Vegetation Urban/Built-up

Today's Lecture - Feature Learning

From Linear Models to Multi-Layer Perceptrons



BLOCK 1
Linear Regression

BLOCK 2
Logistic Regression

BLOCK 3
Multilayer Perceptrons

Each Block: Model -> Loss -> Optimization

BLOCK 1
Linear Regression

MODEL

$\hat{y} = xw + b$

LOSS

MAE or MSE

OPTIMIZATION

Closed-form solution (MSE)

BLOCK 2
Logistic Regression

MODEL

$\sigma(xw + b)$ or $\text{softmax}(Wx + b)$

LOSS

BCE or CE

OPTIMIZATION

Gradient descent

BLOCK 3
Multilayer Perceptrons

MODEL

Linear layers + activations

LOSS

MAE / MSE or BCE / CE

OPTIMIZATION

Backpropagation + gradient descent

Learning outcomes

Explain linear regression, from one to many inputs and outputs, and how minimizing MSE fits its parameters.

Not comfortable — Very comfortable

Click to respond

Explain how sigmoid and softmax turn linear outputs into class probabilities.

Not comfortable — Very comfortable

Click to respond

Explain how MSE and cross-entropy losses and gradient descent guide parameter learning.

Not comfortable — Very comfortable

Click to respond

Describe artificial neurons and layers as affine maps followed by nonlinear activations.

Not comfortable — Very comfortable

Click to respond

Explain how MLPs learn intermediate representations.

Not comfortable — Very comfortable

Click to respond

Explain how backpropagation computes the gradient of the loss for every weight.

Not comfortable — Very comfortable

Click to respond

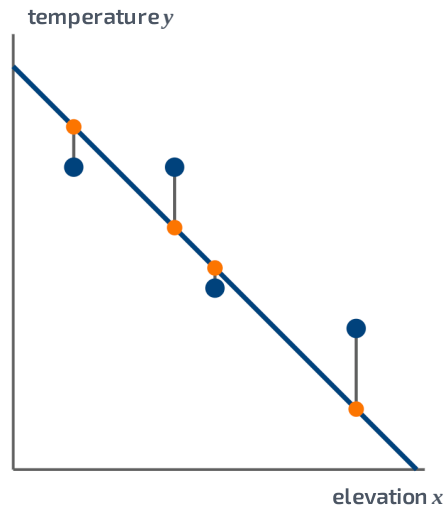
Linear Regression



Photo: Luca Galuzzi / Wikimedia Commons, CC BY-SA 2.5.

How much does the temperature change when you hike a mountain?





A simple linear model

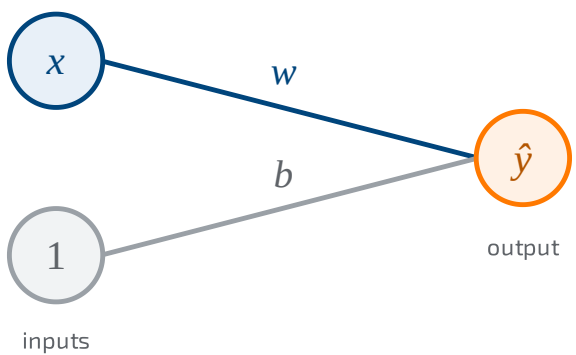
$$\hat{y} = w\,x + b$$

maps an elevation x to an estimated temperature $\hat{y}$, with the **weight** w (slope) and the **bias** b (y-intercept).

Photo: Luca Galuzzi / Wikimedia Commons, CC BY-SA 2.5. Figure: M. Rußwurm & D. Tuia, *Linear Regression*, ENV-408 course materials, EPFL, 2023.

Univariate Linear Regression

$$\hat{y} = w\,x + b \;=\; \begin{pmatrix} x & 1 \end{pmatrix} \begin{pmatrix} w \\ b \end{pmatrix}$$




```python
import torch.nn as nn

# 1 input (elevation) → 1 output (temperature)
model = nn.Linear(1, 1)

print(model.weight)  # w
print(model.bias)    # b

n = sum(p.numel() for p in model.parameters())
print(n)
```

From Uni- to Multivariate Linear Regression



Same elevation, different temperature

☀️ sun-facing slope: warmer

☁️ shaded slope: colder

Slope and **aspect** (the direction a slope faces) control how much sun a location gets — extra covariates beyond elevation.

Photo: Luca Galuzzi / Wikimedia Commons, CC BY-SA 2.5.

Univariate linear regression: elevation only, two parameters

$$y = w_1 \overbrace{x_1}^{\text{elevation}} + b$$

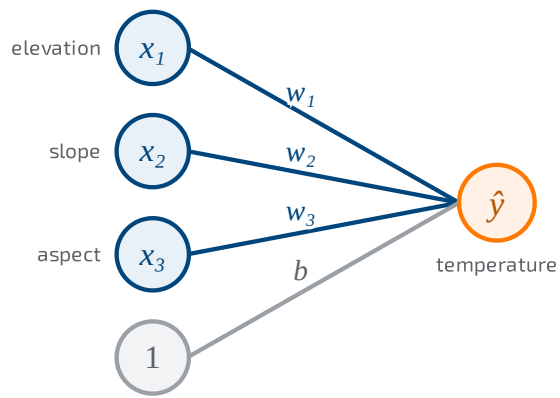
Multivariate linear regression

- one input sample $\mathbf{x}_i = (x_1, x_2, x_3, \ldots, x_p)$ of p features
- corresponding target y_i
- one weight per feature $\mathbf{w} = (w_1, w_2, \ldots, w_p)$, plus one bias b

$$y_i = w_1 \overbrace{x_1}^{\text{elevation}} + w_2 \overbrace{x_2}^{\text{slope}} + w_3 \overbrace{x_3}^{\text{aspect}} + \cdots + b = \mathbf{w}^\top \mathbf{x}_i + b$$

Multivariate Linear Regression

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$
$$= \begin{pmatrix} x_1 & x_2 & x_3 & 1 \end{pmatrix} \begin{pmatrix} w_1 & w_2 & w_3 & b \end{pmatrix}^\top$$




```python
import torch.nn as nn

# elevation, slope, aspect → temperature
model = nn.Linear(3, 1)

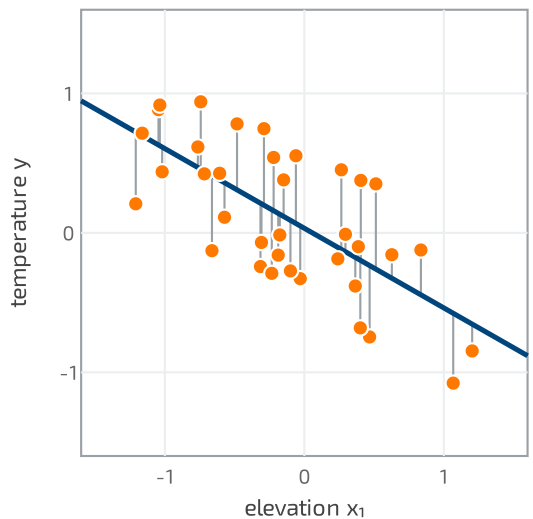
print(model.weight.shape)  # (1, 3): w1, w2, w3
print(model.bias.shape)    # (1,):   b

n = sum(p.numel() for p in model.parameters())
print(n)                   # 4
```

Same graph, more input nodes: one weight per feature, still one bias. Only `in_features` changes.

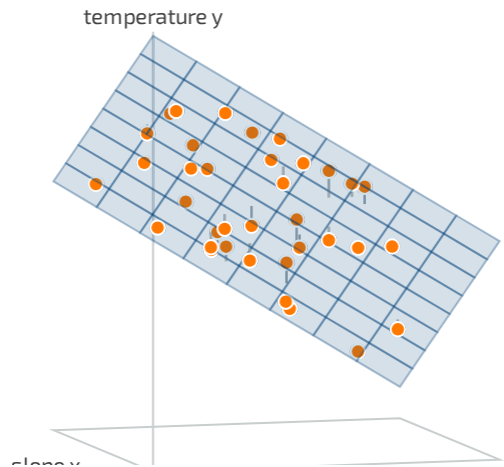
Geometry: From a Line to a Plane

1 feature → a line in 2D



$\hat{y} = 0.03 - 0.57\, x_1 \cdot R^2 = 0.48$

2 features → a plane in 3D

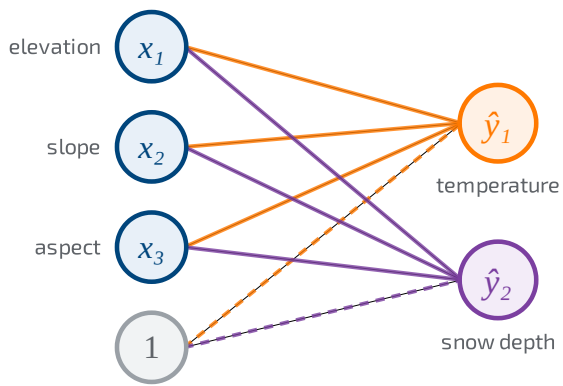


$\hat{y} = 0.10 - 0.53\, x_1 + 0.45\, x_2 \cdot R^2 = 0.97$

p features → a hyperplane in $p+1$ dimensions. The second feature (slope) explains much of the spread that elevation alone leaves in the residuals.

Many Regressions in Parallel: $\mathbb{R}^N \to \mathbb{R}^M$

$$\underbrace{\begin{pmatrix}\hat{y}_1\\ \hat{y}_2\end{pmatrix}}_{\hat{\mathbf{y}}\in\mathbb{R}^M} = \underbrace{\begin{pmatrix}w_{11} & w_{12} & w_{13}\\ w_{21} & w_{22} & w_{23}\end{pmatrix}}_{\mathbf{W}\in\mathbb{R}^{M\times N}} \underbrace{\begin{pmatrix}x_1\\ x_2\\ x_3\end{pmatrix}}_{\mathbf{x}\in\mathbb{R}^N} + \underbrace{\begin{pmatrix}b_1\\ b_2\end{pmatrix}}_{\mathbf{b}\in\mathbb{R}^M}$$




```python
import torch.nn as nn

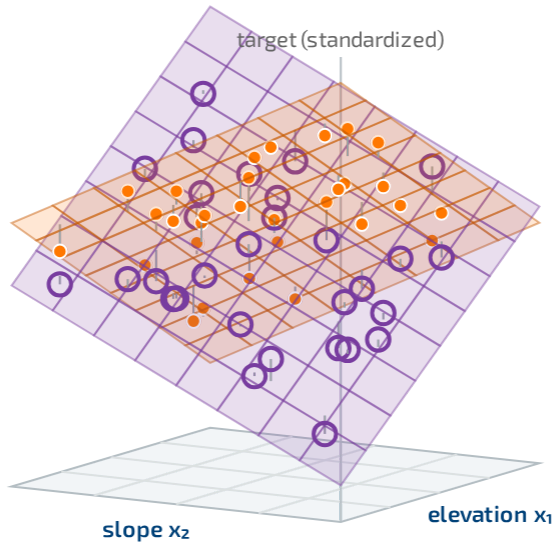
# 3 features → 2 targets
model = nn.Linear(3, 2)

print(model.weight.shape)  # (2, 3) = (M, N)
print(model.bias.shape)    # (2,)

n = sum(p.numel() for p in model.parameters())
print(n)                   # 2*3 + 2 = 8
```

Each row of $\mathbf{W}$ is one multivariate regression; M run in parallel on the same input. Together: a linear transformation with $M \cdot N + M$ parameters.

Geometry: One Plane per Target



elevation view | top view: $x_1 \times x_2$

rotate

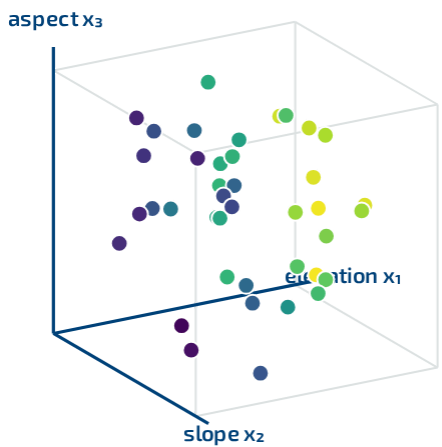
- row 1 of W: temperature $\hat{y}_1$
  $\hat{y}_1 = 0.13 - 0.46\, x_1 + 0.40\, x_2$
- row 2 of W: snow depth $\hat{y}_2$
  $\hat{y}_2 = -0.02 + 0.50\, x_1 - 0.38\, x_2$

same inputs x_1, x_2 (the floor) — two planes, two sets of weights

Same inputs, two targets: each row of $\mathbf{W}$ fits its own plane — M targets, M hyperplanes.

Linear Projection: From N to M Dim Spaces

input space $\mathbb{R}^3$



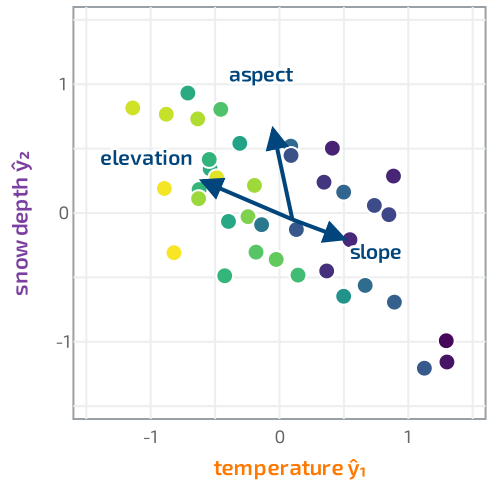
drag to rotate · hover a point
colour = elevation: low high

$$\hat{y} = Wx + b$$

$$\begin{vmatrix} -0.70 & 0.40 & -0.15 \\ 0.30 & -0.15 & 0.70 \end{vmatrix}$$

$W \in \mathbb{R}^{2\times3}$

output space $\mathbb{R}^2$



arrows: where each input axis lands (columns of W)

$\mathbf{Wx} + \mathbf{b}$ maps every location from the input space to a point in the output space; each input axis lands on a column of $\mathbf{W}$.

Quiz: How Much Colder per 100 m?



Back to our hiker: $\hat{y} = w\,x + b$ with elevation x and temperature $\hat{y}$.

The slope w answers the question: **how many degrees does it get colder for every 100 m of ascent?**

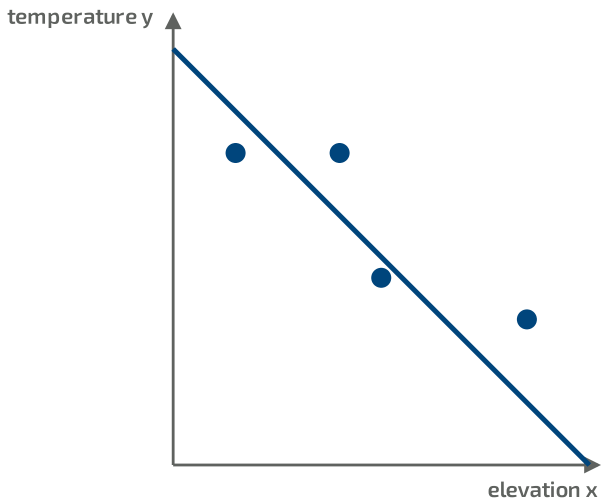
On average about **0.65 °C per 100 m**, i.e. $w \approx -6.5\,^{\circ}\mathrm{C}/\mathrm{km}$ — the slope our regression should recover from data.



Click on the line: temperature drop per 100 m of ascent

Photo: Luca Galuzzi / Wikimedia Commons, CC BY-SA 2.5.

How to estimate w and b from data?



We have a dataset of data samples (x_i, y_i).

Problem: how do we find good model parameters w, b for our linear model $f(x; w, b) = w\,x + b$?

Why Don't the Points Lie on the Line?

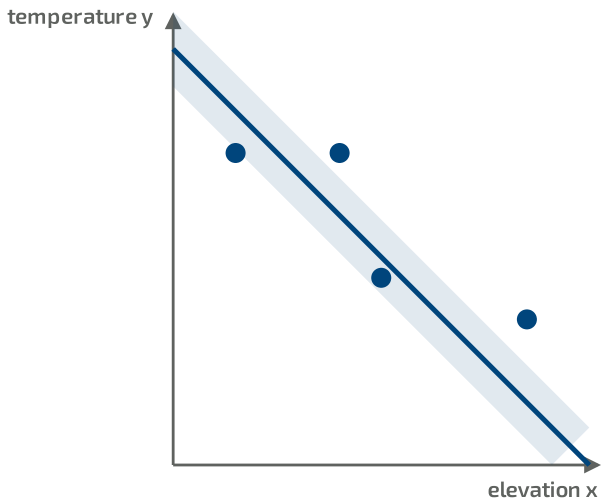


Photo: Hustvedt / Wikimedia Commons, CC BY-SA 3.0

$$y_i = w\,x_i + b \pm \overbrace{\varepsilon}^{\text{error}}$$

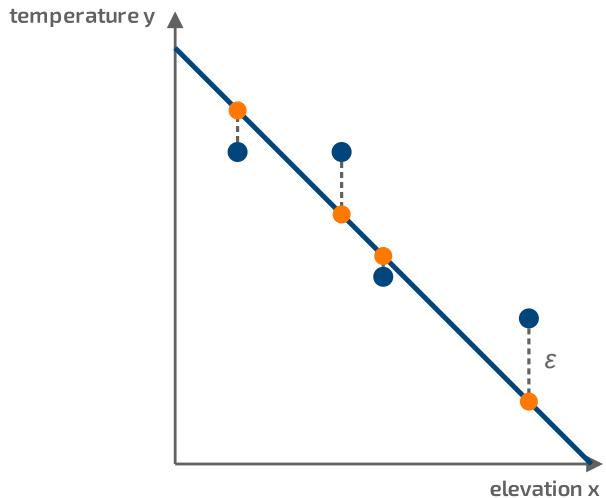
with the error ε as a random variable — the model never fits the data perfectly.

Measurement noise is one source: not every thermometer is precise.



The hiker took a bad thermometer

Residuals and Mean Absolute Error



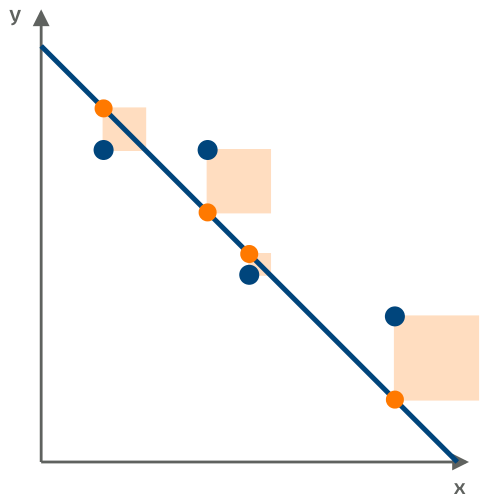
Concretely, the error — the "residual" ε_i — for each sample i:

$$\varepsilon_i = \hat{y}_i - y_i$$

We want small residuals over the whole dataset. Positive and negative residuals would cancel, so we average their absolute values: the **Mean Absolute Error**

$$\mathrm{MAE}(\mathcal{D}; w, b) = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i|$$

Mean Squared Error (MSE)



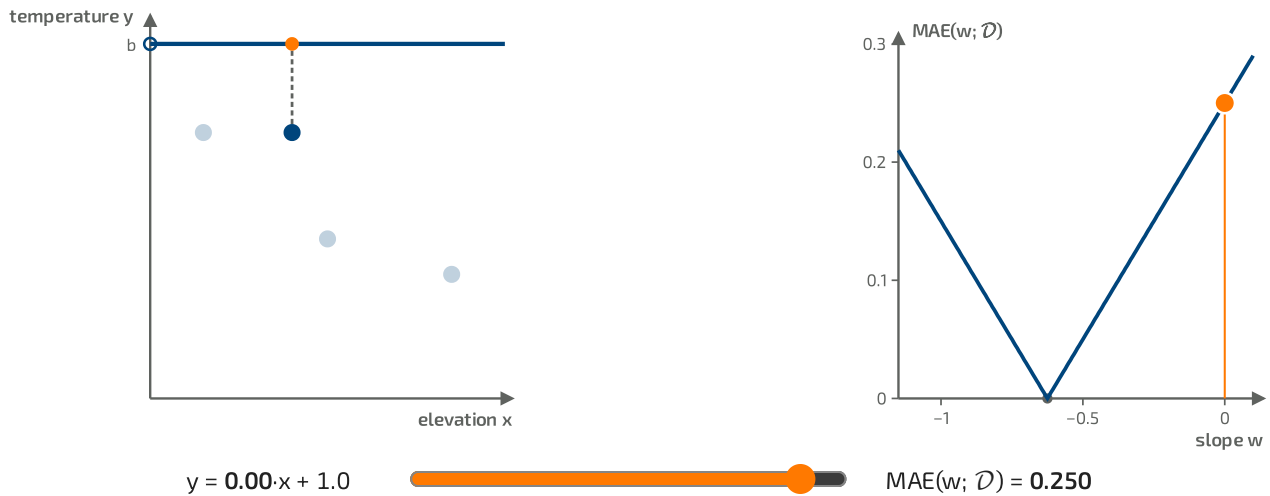
$$\mathrm{MSE}(\mathcal{D}; w, b) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Punishes large errors with a square penalty.

Geometric intuition: find a model that minimizes the total area of these squares.

The Loss $\mathcal{L}(w;\ \mathcal{D})$ as a Function of w

MAE MSE | one point all points



y = **0.00**·x + 1.0

MAE(w; $\mathcal{D}$) = **0.250**

① Big errors matter most: squared penalty — a little off is fine, very off is fatal.

② Smooth gradients: MAE has constant slopes and kinks; MSE is smooth — needed to find good parameters (later).

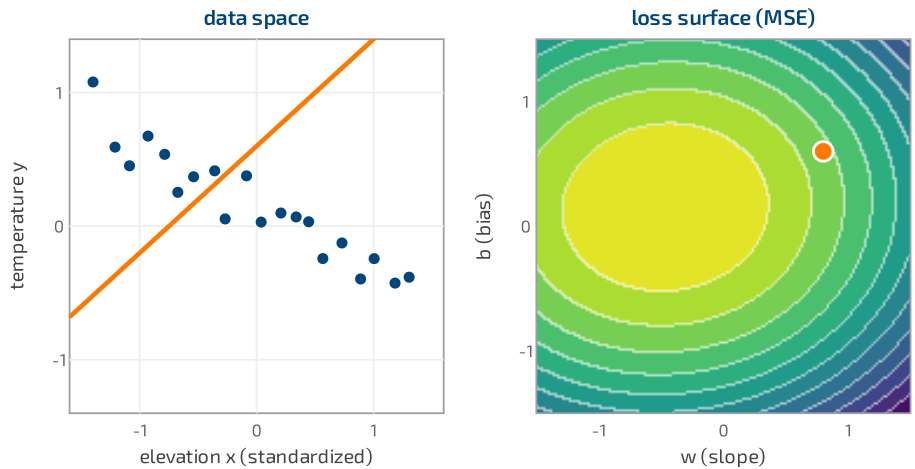
Searching over b and w spans a 2D Loss Surface

The loss $\mathcal{L}(\mathcal{D}; w, b)$ depends on

- the **model** $f(x) = w\,x + b$
- the **loss function**, e.g. MSE
- the **dataset** $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$

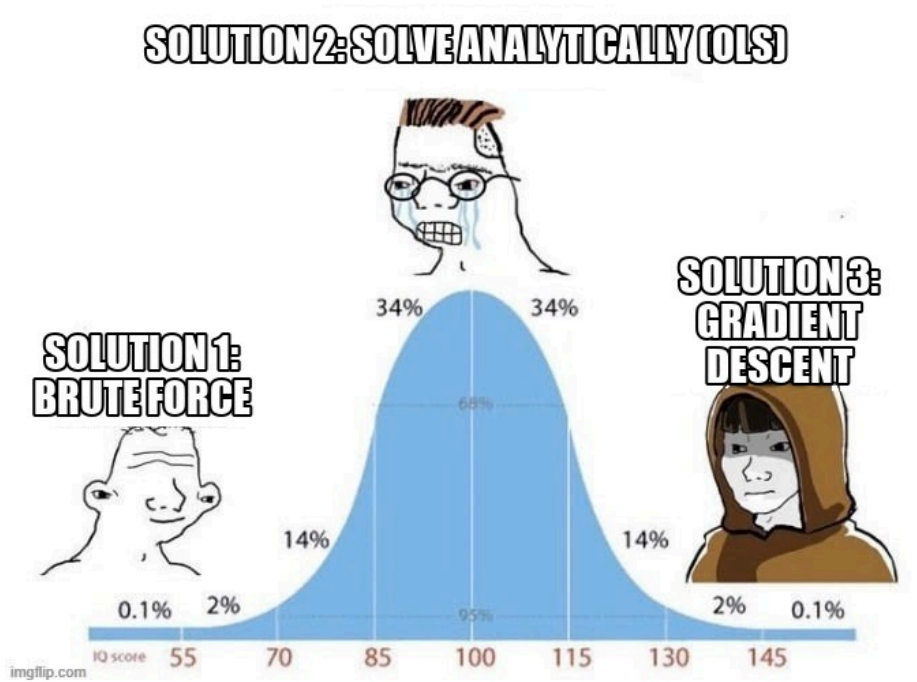
For a fixed dataset, every parameter pair (w, b) gets one loss value: a **surface over parameter space**.

> **Each point on the right is one line on the left.**



hover the surface · w = 0.80, b = 0.60 · **L = 0.613**

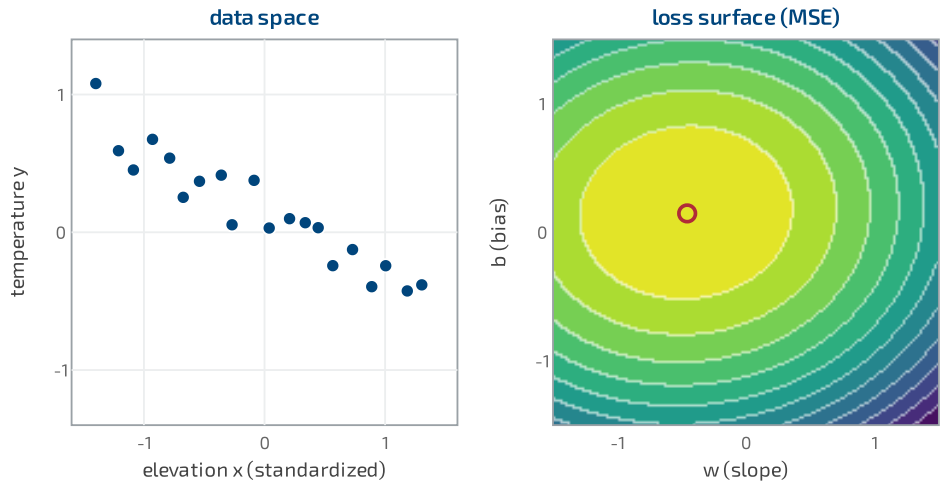
How to find the optimal w and b parameters?



Meme template: "bell curve / IQ distribution" via imgflip.com, labels modified.



Brute Force: Guess the Parameters ¯_(□)_/¯



guess | 10 guesses | 100 guesses | reset | 0 guesses

Works for 2 parameters — but it is expensive and never exact.

100 values per parameter: 2 parameters → 10^4 evaluations; a small MLP with 10^4 parameters → $100^{10\,000}$. The curse of dimensionality again.



Smart Solution: Ordinary Least Square (OLS)

Task: find $\hat{w}, \hat{b}$ that minimize the Mean Squared Error

$$\hat{w}, \hat{b} = \arg\min_{w,b} \frac{1}{2N} \sum_{i=1}^{N} \left(\overbrace{w\, x_i + b}^{\hat{y}_i} - y_i \right)^2$$

Question: how do we minimize a function with respect to its parameters?

- calculate the partial derivatives $\frac{\partial \text{MSE}}{\partial w}$ and $\frac{\partial \text{MSE}}{\partial b}$
- set the derivatives to zero: $\frac{\partial \text{MSE}}{\partial w} = 0$, $\frac{\partial \text{MSE}}{\partial b} = 0$
- solve for w and b



OLS Step by Step

$$\mathcal{L}(w,b) = \frac{1}{2N}\sum_{i=1}^{N}\left(w\,x_i + b - y_i\right)^2 \qquad \bar{x} = \frac{1}{N}\sum_i x_i,\;\; \bar{y} = \frac{1}{N}\sum_i y_i$$

① derivative with respect to b

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N}\left(w\,x_i + b - y_i\right) \overset{!}{=} 0$$

$$\Leftrightarrow\; w\,\bar{x} + b - \bar{y} = 0$$

$$\Leftrightarrow\; b = \bar{y} - w\,\bar{x}$$

② derivative with respect to w

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{N}\sum_{i=1}^{N}\left(w\,x_i + b - y_i\right)x_i \overset{!}{=} 0$$

$$\overset{\text{insert } b}{\Leftrightarrow}\; \textstyle\sum_i \left(w\,(x_i - \bar{x}) - (y_i - \bar{y})\right)x_i = 0$$

$$\Leftrightarrow\; w\textstyle\sum_i (x_i - \bar{x})\,x_i = \sum_i (y_i - \bar{y})\,x_i$$

$$\Leftrightarrow\; w = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

last step: $\sum_i (x_i - \bar{x}) = \sum_i (y_i - \bar{y}) = 0$, so x_i may be replaced by $(x_i - \bar{x})$



Ordinary Least Squares (OLS) — Solution

Model $y = w\,x + b$ on a dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^{N}$

Ordinary Least Squares objective

$$\hat{w}, \hat{b} = \arg\min_{w,b} \frac{1}{2N} \sum_{i=1}^{N} \left(\overbrace{w\,x_i + b}^{\hat{y}_i} - y_i \right)^2$$

Closed-form solution

$$\hat{w} = \frac{\mathrm{Cov}[x, y]}{\mathrm{Var}[x]} = \frac{\frac{1}{N}\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{N}\sum_i (x_i - \bar{x})^2}$$

$$\hat{b} = \frac{1}{N}\sum_{i=1}^{N} y_i - \hat{w}\frac{1}{N}\sum_{i=1}^{N} x_i = \bar{y} - \hat{w}\,\bar{x}$$

See Ordinary Least Squares, Wikipedia.



Multivariate OLS — Closed-form Derivation

Data: n locations, $p = 3$ features, one target

$$\mathbf{X} = \begin{pmatrix} & \overbrace{}^{\text{elev.}} & \overbrace{}^{\text{slope}} & \overbrace{}^{\text{aspect}} \\ 1 & x_{11} & x_{12} & x_{13} \\ 1 & x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n3} \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

one row per location: $\mathbf{X} \in \mathbb{R}^{n \times (p+1)}$, temperatures $\mathbf{y} \in \mathbb{R}^n$

Model for all n locations at once:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}, \qquad \mathbf{w} = (b, w_1, w_2, w_3)^\top$$

The column of ones absorbs the bias b into $\mathbf{w}$.

Loss: squared error of all residuals

$$\mathrm{MSE}(\hat{\mathbf{w}}) = \|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|^2 = \mathbf{y}^\top\mathbf{y} - 2\hat{\mathbf{w}}^\top\mathbf{X}^\top\mathbf{y} + \hat{\mathbf{w}}^\top\mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}}$$

Derivative set to zero:

$$\frac{\partial \mathrm{MSE}(\hat{\mathbf{w}})}{\partial \hat{\mathbf{w}}} = -2\mathbf{X}^\top\mathbf{y} + 2\mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}} = 0$$

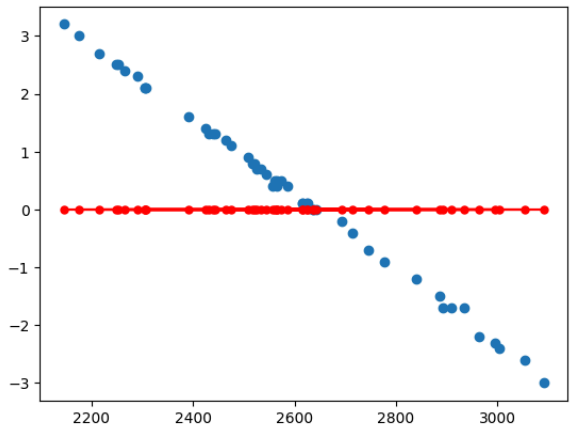
$$\mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^\top\mathbf{y}$$

$$\hat{\mathbf{w}} = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{y}$$

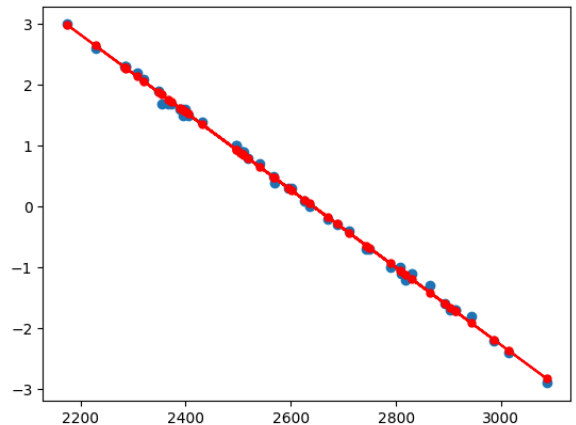
> The normal equation — a closed-form solution, no iterative optimization required.



Let's Appreciate the Elegance



$\mathbf{w} = (0, 0, 0, 0, \dots)$



$\mathbf{w} = (-.1, .2, .5, -.5, \dots)$

$\hat{\mathbf{w}} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$ — a single line of code, and blazing fast.

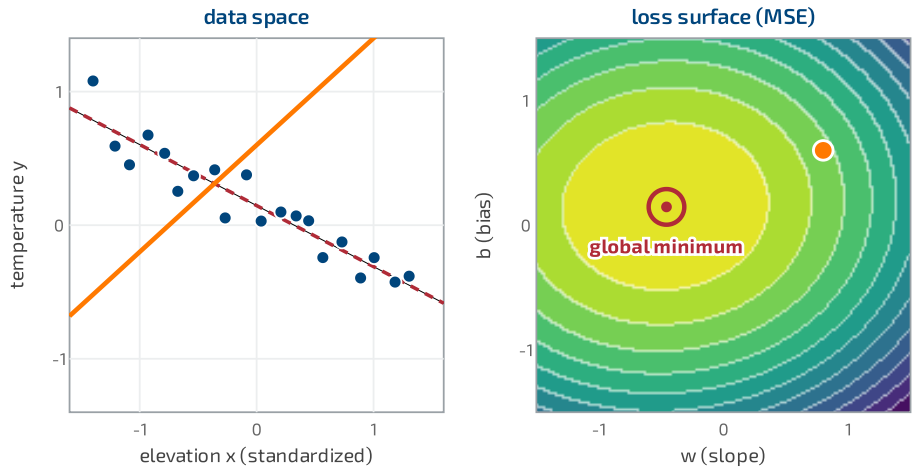


Why Does OLS Work for Linear Regression?

The MSE of a linear model is a **quadratic bowl** in (w, b):

- it is **convex**: no local minima to get stuck in
- it has exactly **one global minimum**
- at that point, the slope is zero in every direction

> So "derivatives $= 0$" has exactly one solution — and that solution is the global optimum.



closed form (OLS) hover the surface · w = 0.80, b = 0.60 · **L = 0.613**

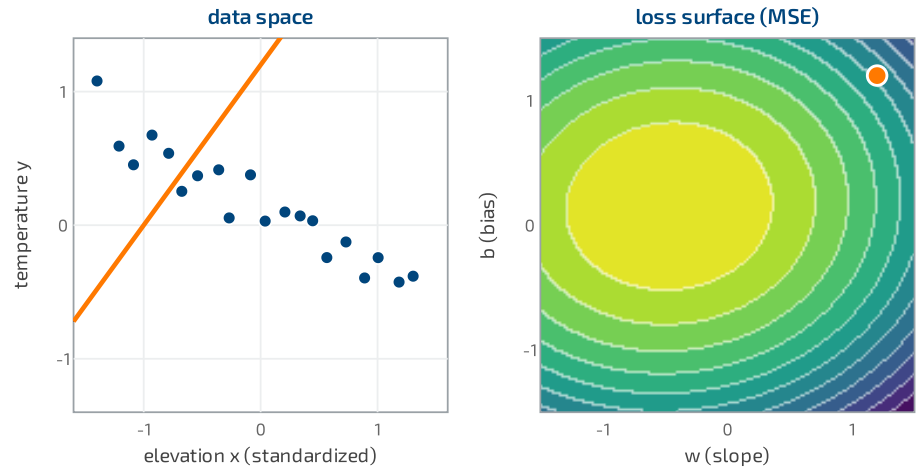


Gradient Descent — Guess Once, Then Improve

Gradient descent

1. initialize w, b (e.g. randomly)
2. update $w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}$ and $b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$
3. repeat 2 until convergence

- the gradient $\nabla \mathcal{L}$ points **uphill**; we step the other way
- η is the **step size** or **learning rate**





The Gradient of the MSE

Loss for $\hat{y}_i = w\,x_i + b$:

$$\mathcal{L}(w, b) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Partial derivatives (chain rule, the $\frac{1}{2}$ cancels):

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{N} \sum_i (\hat{y}_i - y_i)\, x_i \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N} \sum_i (\hat{y}_i - y_i)$$

> **Each sample pushes the parameters by its residual — weighted by its input.**

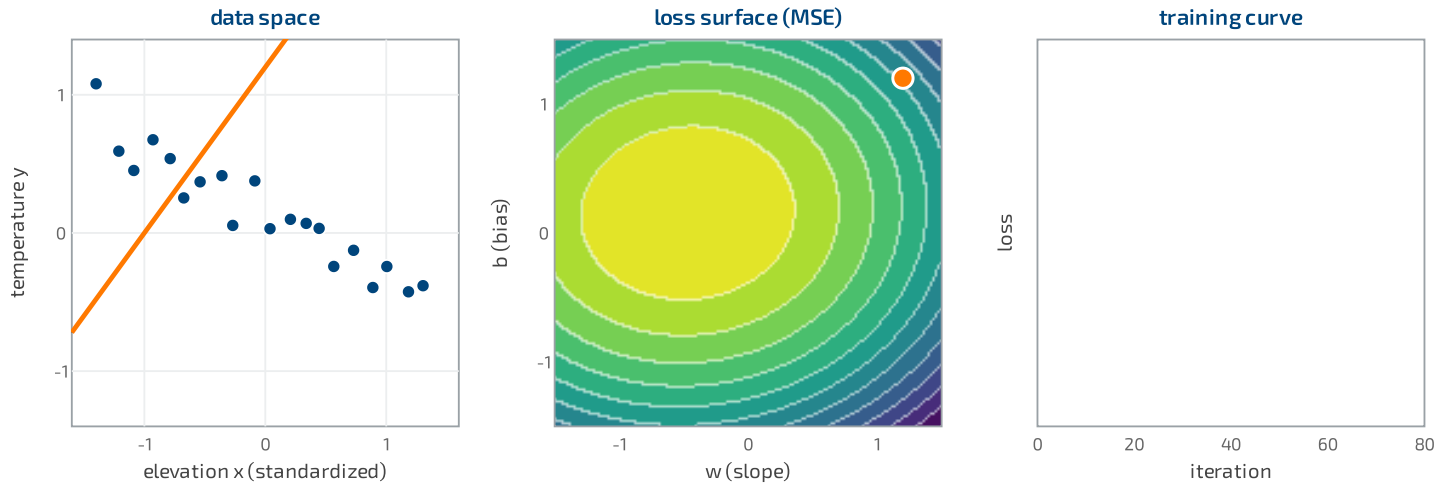
```python
import numpy as np

w, b, lr = 1.2, 1.2, 0.3    # init, step size
for it in range(100):
    y_hat = w * x + b        # prediction
    r = y_hat - y            # residuals
    w -= lr * (r * x).mean() # dL/dw
    b -= lr * r.mean()       # dL/db
```

Setting both derivatives to zero and solving gives the OLS solution. Gradient descent only needs to *evaluate* them.



Gradient Descent: Role of the Learning Rate



η = 0.05: safe, but slow — still not converged after 80 steps

η ≈ 0.3–1: few steps to the optimum

η ≥ 2: every step overshoots more — the loss diverges

Recap: Linear Regression

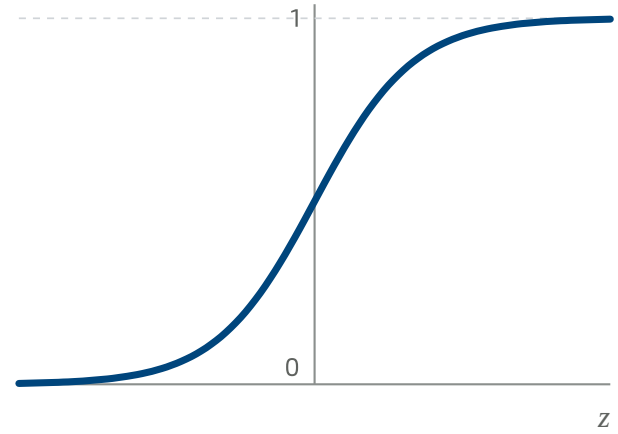
- **Model:** from univariate $\hat{y} = w\,x + b$ (a line) to multivariate $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$ (a plane), to M regressions in parallel $\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}$: a **linear transformation** from input to output space
- **Loss:** residuals measured by MAE or MSE; we usually prefer **MSE**: large errors are penalized more, and its smooth, convex bowl has one global minimum
- **Fitting:** three ways to find the parameters — brute force (does not scale), OLS in closed form $\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ (special cases only), and gradient descent (steps downhill, works everywhere)

	brute force	closed form	gradient descent
linear regression	✓	✓	✓

Limitation: linear regression predicts continuous values (temperature, snow depth). What about categories — snow or no snow, forest or field?

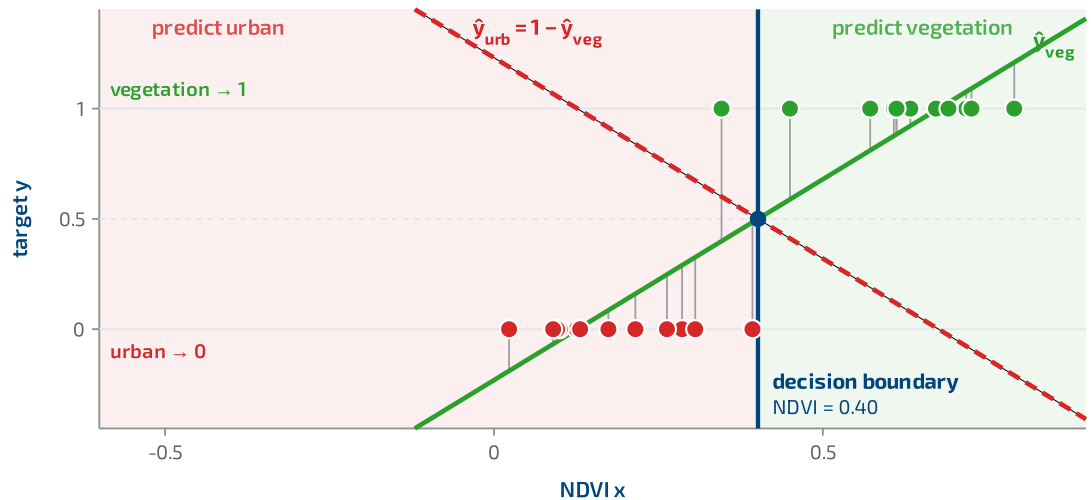
Next: classification with the same linear transformation — new output function, new loss, and the closed form disappears.

Logistic Regression



1D Land Cover

From the NDVI histograms of lecture 3 to a **regression problem** with 0/1 targets.



LINEAR REGRESSION

$$\hat{y}_{veg} = w\,x + b$$

$$\hat{y}_{urb} = 1 - \hat{y}_{veg}$$

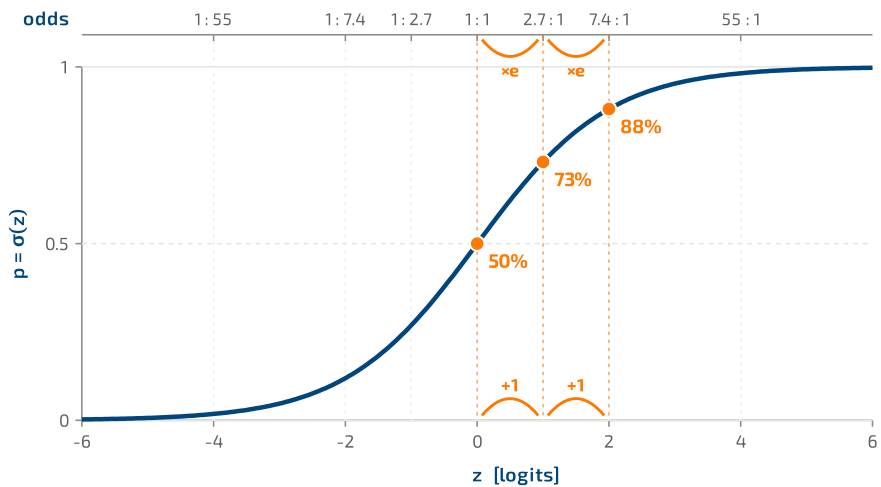
decision boundary:

$\hat{y}_{veg} = \hat{y}_{urb}$

The Logistic Function and the Logit

Logistic function
also called the **sigmoid**: S-shaped, like the Greek letter sigma (ς)

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



INVERSE: LOGIT = LOGISTIC UNIT

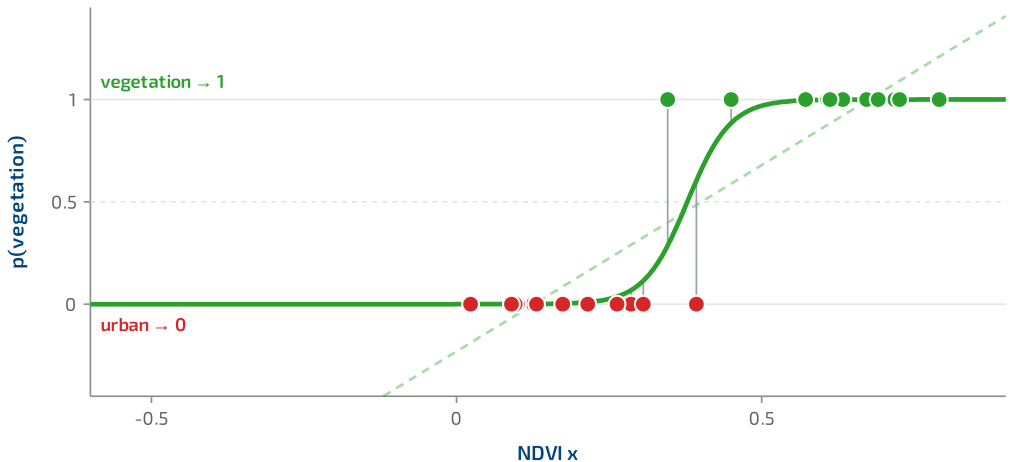
$$z = \text{logit}(p) = \ln \frac{p}{1-p}$$

term coined by J. Berkson (1944) · z measures p on the log-odds scale

$+1 \text{ logit} \Rightarrow \text{odds} \times e$

equal steps in z, equal factors in the odds, but shrinking steps in p: 50% → 73% → 88%

From Linear to Logistic Regression



LINEAR REGRESSION

$$\hat{y}_{\text{veg}} = w\,x + b$$

LOGISTIC REGRESSION

$$p(\text{vegetation}) = \sigma(w\,x + b)$$

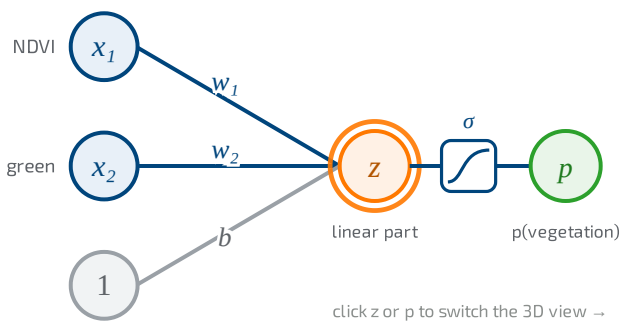
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

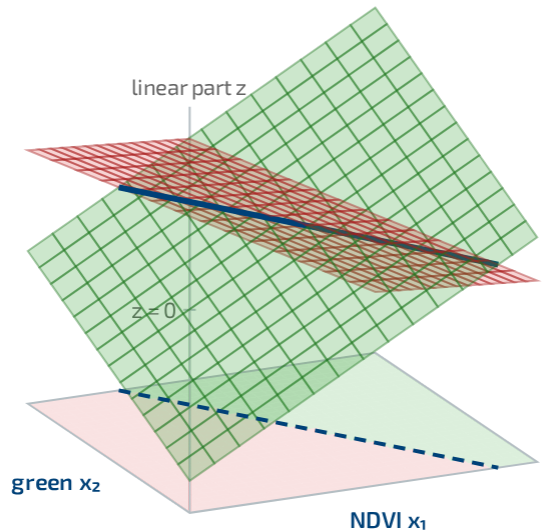
steepness: w × 1.0

flatter — sharper

Logistic Regression with Multiple Inputs

$$p(\text{vegetation}) = \sigma(w_1 x_1 + w_2 x_2 + b)$$
$$p(\text{not vegetation}) = 1 - p(\text{vegetation})$$

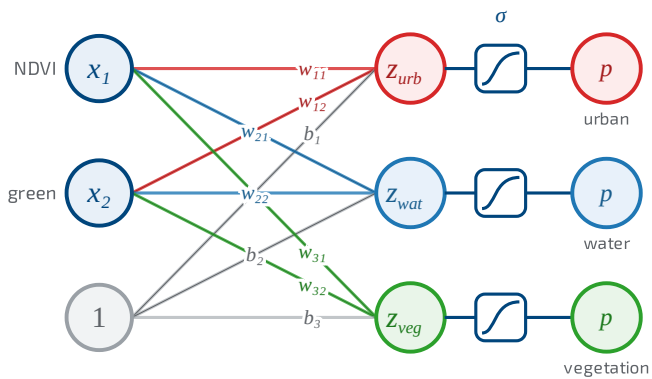




z and −z: two planes, crossing at z = 0

look along boundary | top view | rotate

From Binary Classification to Multi-Class



One linear output **per class**, all from the same inputs:

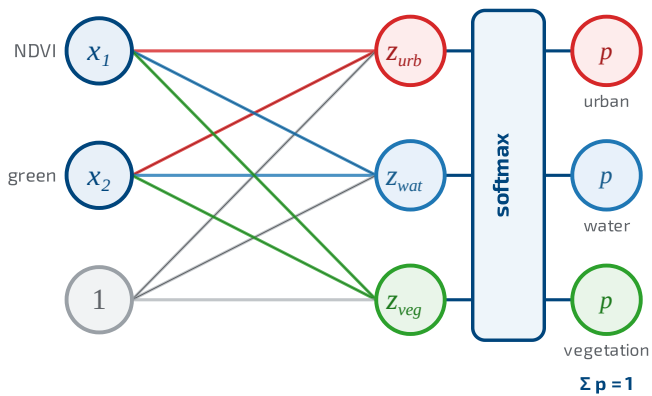
$$\underbrace{\begin{pmatrix} z_{\text{urb}} \\ z_{\text{wat}} \\ z_{\text{veg}} \end{pmatrix}}_{\mathbf{z}} = \underbrace{\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}}_{\mathbf{W}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \mathbf{b}$$

Three independent logistic regressions, each one class vs. all others (*one-vs-rest*):

Softmax Functions

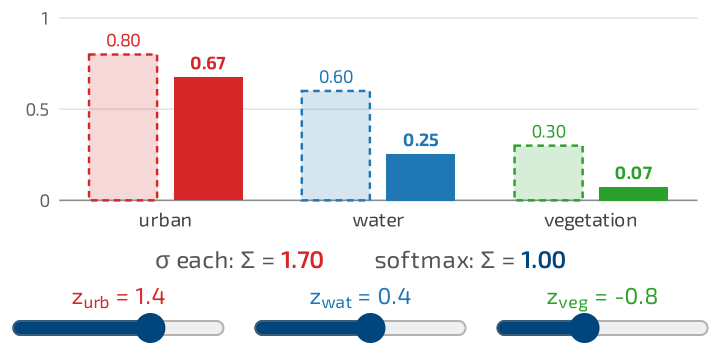
Softmax
K logits → K probabilities that sum to 1

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_j e^{z_j}} \quad \text{e.g.} \quad p_{\text{veg}} = \frac{e^{z_{\text{veg}}}}{e^{z_{\text{urb}}} + e^{z_{\text{wat}}} + e^{z_{\text{veg}}}}$$



e^z makes every score **positive** · the shared denominator makes them **sum to 1**

dashed one sigmoid per class (one-vs-rest) solid softmax



Log-Probabilities in Practice

$$\log p_k = \log \frac{e^{z_k}}{\sum_j e^{z_j}} = z_k - \log \sum_j e^{z_j}$$

the log turns the **division** into a **subtraction**: logits minus one shared number

Example (urban, water, vegetation):

$$\mathbf{z} = (2,\ 1,\ -1), \qquad \log \textstyle\sum_j e^{z_j} = 2.35$$
$$\log \mathbf{p} = \mathbf{z} - 2.35 = (-0.35,\ -1.35,\ -3.35)$$
$$\mathbf{p} = (0.71,\ 0.26,\ 0.04)$$

Why work with $\log p$?

- **Numerically stable:** no overflow of e^z, no underflow of tiny products
- **The loss needs it:** we will minimize $-\log p$ of the true class

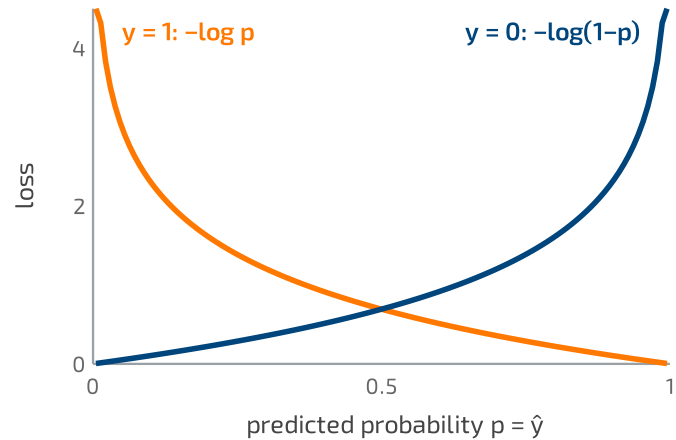
```
log_p = torch.log_softmax(z, dim=1)
```

Binary Cross Entropy

How wrong is $p_i = p(\text{vegetation})$ for label $y_i \in \{0, 1\}$?

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[\, y_i \log p_i + (1 - y_i)\log(1 - p_i)\Big]$$

- $-\log$ of the probability of the **true** class
- confident & right → 0, confident & wrong → ∞



Multi-Class Cross Entropy

Binary: two classes, vegetation and not vegetation

$$\mathcal{L}_{\text{BCE}} = -\big[\, y \log p + (1-y) \log(1-p) \big]$$

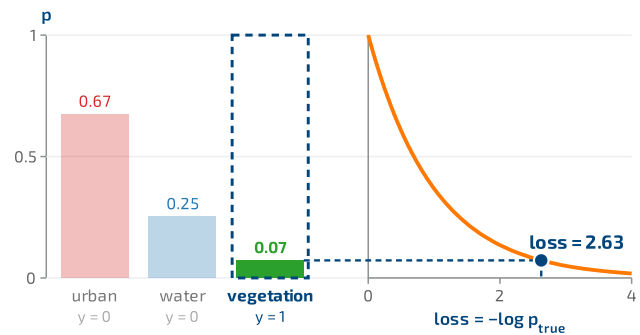
Same thing as a sum over the 2 classes, one-hot $\mathbf{y} = (y,\, 1-y)$, $\mathbf{p} = (p,\, 1-p)$:

$$\mathcal{L}_{\text{BCE}} = -\sum_{k=1}^{2} y_k \log p_k$$

Multi-class: just sum over all K classes, $\mathbf{p} = \text{softmax}(\mathbf{z})$:

$$\mathcal{L}_{\text{CE}} = -\sum_{k=1}^{K} y_k \log p_k = -\log p_{\text{true}}$$

Example: a **vegetation** pixel, $\mathbf{y} = (0, 0, 1)$



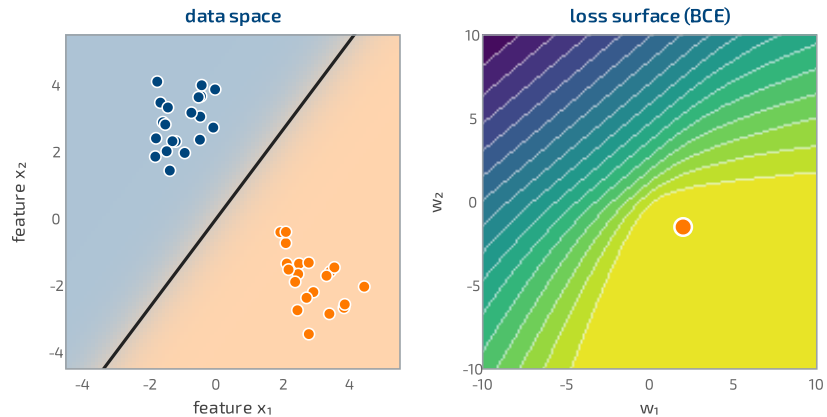
A: confident, wrong | unsure | B: confident, right

Logistic Regression in Weight Space

$$p = \sigma(w_1 x_1 + w_2 x_2)$$

- every model is a **point** (w_1, w_2)
- each point → one decision boundary
- which point has the lowest loss?

> **Note:** no bias $(b = 0)$, so every boundary $w_1 x_1 + w_2 x_2 = 0$ passes through the **origin** $(0, 0)$. A simplification: with only two weights, the whole loss surface fits in one 2D plot.

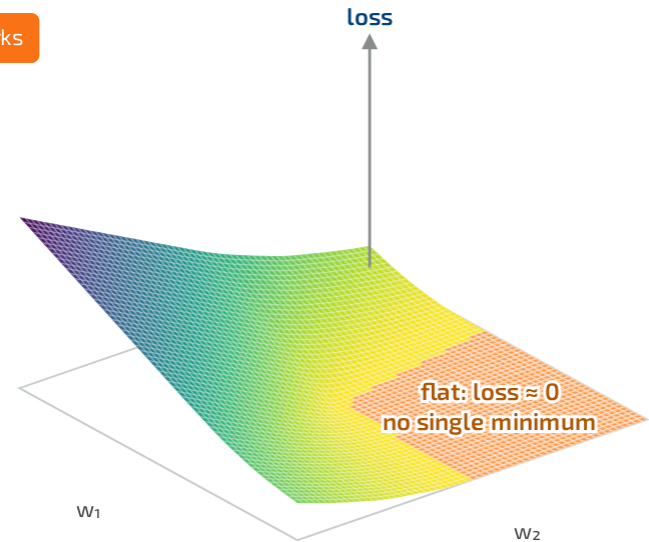


hover the surface · $w_1 = 2.0$, $w_2 = -1.5$ · **L = 0.002**

The Loss Surface of Logistic Regression

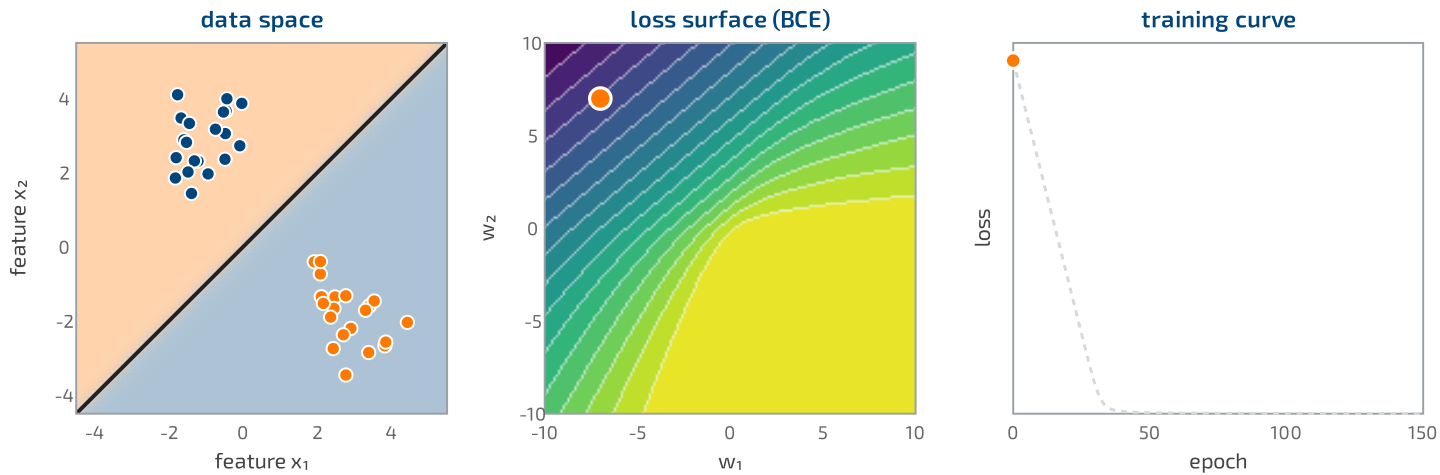
no global minimum → no closed form → Gradient Descent still works

- **convex**: no bad valleys
- **flat region**: larger weights keep lowering the loss a little
- in practice: stop early or use weight decay



Let's Try Gradient Descent

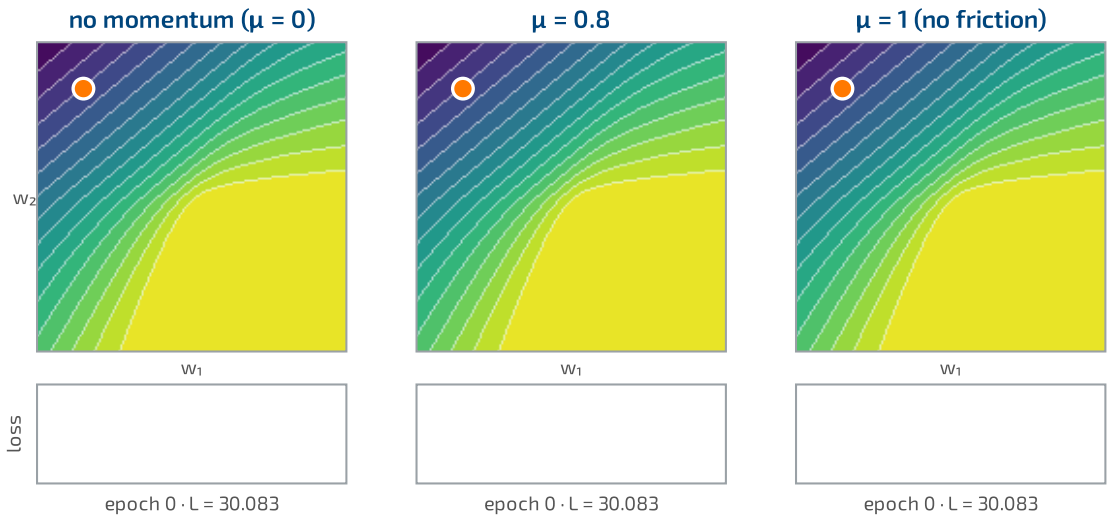
Start with a wrong boundary at $\mathbf{w} = (-7, 7)$ and press **run**.



Steep wall → fast progress. Flat plateau → slow progress; the loss never reaches zero.

Momentum accelerates Gradient Descent

Keep a running velocity $\mathbf{v}$ and let it carry the steps: $\mathbf{v} \leftarrow \mu\,\mathbf{v} - \eta\,\nabla\mathcal{L}, \quad \mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$

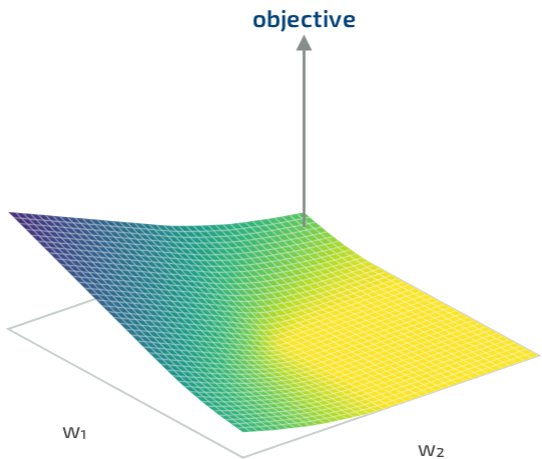


Momentum crosses flat areas quickly; with μ = 1 nothing slows it down — it never stops.

Regularization through Weight Decay

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_{\text{BCE}}(\mathbf{w}) + \lambda\, R(\mathbf{w}), \qquad R(\mathbf{w}) = \tfrac{1}{2}\|\mathbf{w}\|^2$$

the penalty removes the flat plateau: there is exactly **one** optimum



LAYERS

BCE loss

penalty λ/2 · |w|²

loss + penalty

λ = 0.05

FITTING

run gradient descent
with momentum

reset



Occam's razor: prefer the simpler model, here: **smaller weights**

William of Ockham, stained glass. Moscarlop, CC BY-SA 3.0, Wikimedia Commons

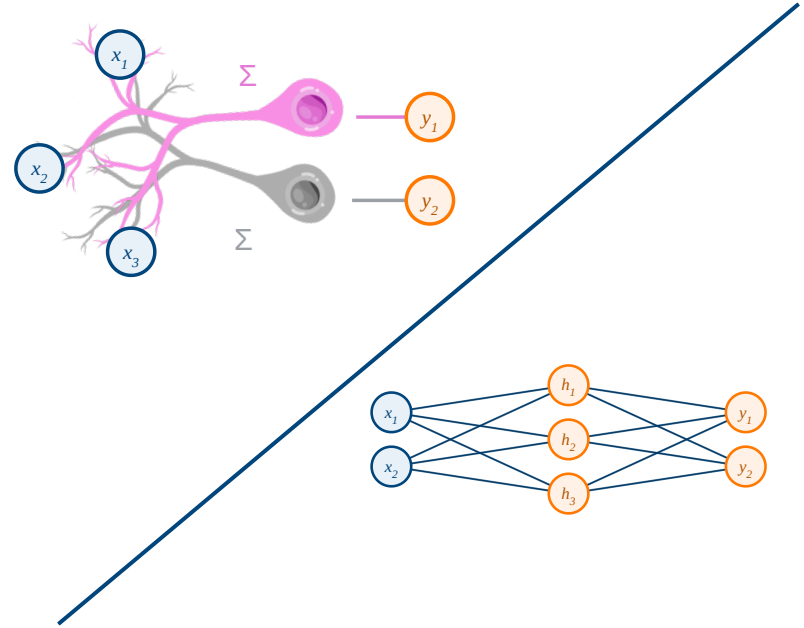
Recap: Logistic Regression

	Linear regression	Logistic regression
Model	$\mathbf{W}\mathbf{x} + \mathbf{b}$	$\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$
Output	continuous value	class probability
Typical loss	MSE	cross-entropy
Finding the weights	closed-form solution in one step	gradient descent (+ regularization)

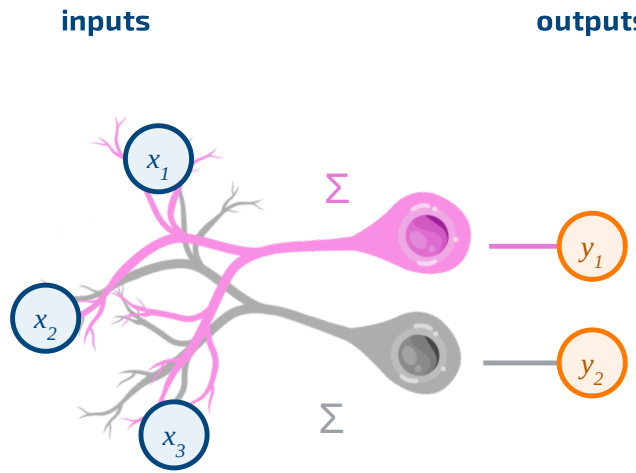
The linear transformation has not changed. What changes is the interpretation of its output, the output function and the loss — and with them, how we must optimize.

Next: what if the features are not good enough for a linear boundary? Combine several transformations and let the model learn the features.

Multilayer Perceptrons



Biological Interpretation of Adding Signals



In **1943**, Warren McCulloch and Walter Pitts proposed the **binary artificial neuron** as a logical model of biological neural networks.

- **dendrites** collect the input signals x_i
- the **cell body** adds them up: $\sum_i w_i x_i$
- the neuron **fires** ($y = 1$) if the sum exceeds a threshold

Each output of $\mathbf{y} = \mathbf{W}\mathbf{x}$ is one such neuron, without the threshold.

Neuron drawings adapted from Wikipedia by Hariadhi, own work.
W. S. McCulloch and W. Pitts (1943): A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5, 115–133.

The McCulloch & Pitts Neuron (1943)

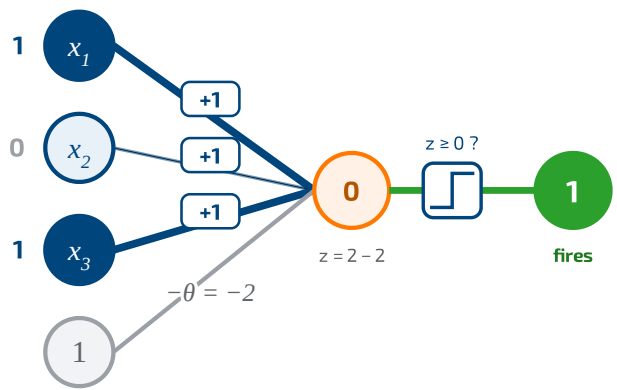
Networks like neurons can compute any logical function.



Warren McCulloch
1898–1969



Walter Pitts
1923–1969



AND OR **majority**

McCulloch & Pitts: binary inputs, a weighted sum and a **threshold** θ

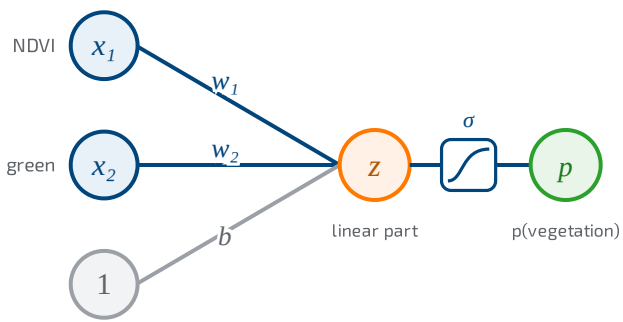
$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

- the threshold acts as a bias $b = -\theta$
- weights set **by hand** (e.g. AND / OR), no learning

McCulloch: University of Illinois Archives. Pitts (1954): Francis Bello, public domain, Wikimedia Commons.

Recap Logistic Regression

Logistic regression is the same neuron with a smooth activation.



Logistic regression (block 2): the same neuron, with a **sigmoid** instead of the step

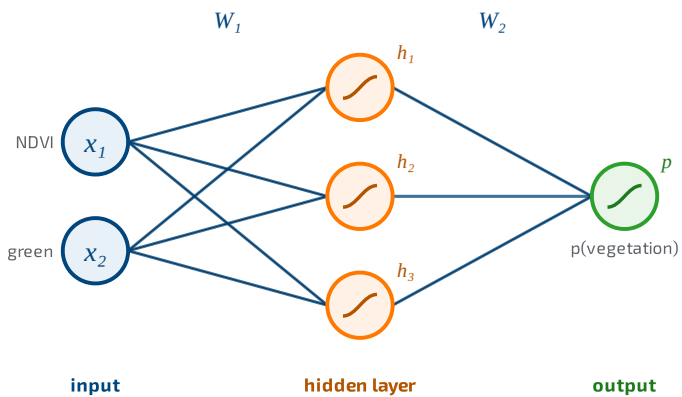
$$p = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

- same weighted sum, the threshold is the bias b
- at $z = 0$: step jumps, sigmoid gives $p = 0.5$
- output: a **probability** instead of 0 or 1
- weights **learned** by gradient descent, not set by hand

The sigmoid has a gradient, the step does not: that is what makes learning possible.

Going Deeper with Logistic Regressions

Stack logistic regressions: the outputs of one layer are the inputs of the next.



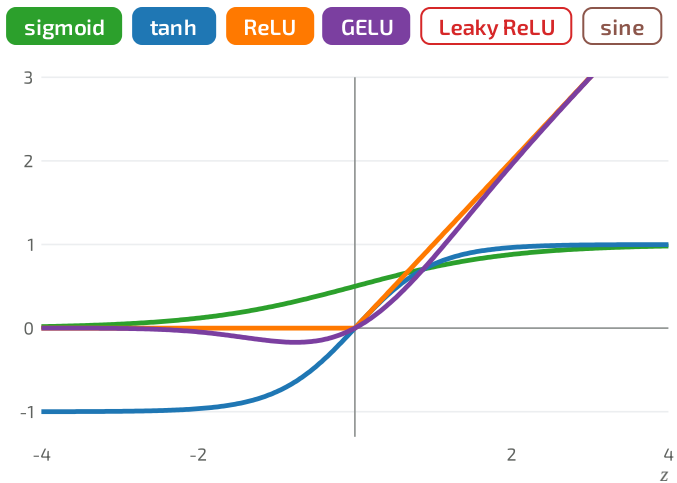
Hidden layer: $\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x})$, logistic regressions on $\mathbf{x}$

Output layer: $p = \sigma(\mathbf{w}_2^\top \mathbf{h})$, a logistic regression on $\mathbf{h}$

PyTorch: add nn.Linear / nn.Sigmoid lines, the graph follows reset

```
model = nn.Sequential(
    nn.Linear(2, 3),
    nn.Sigmoid(),
    nn.Linear(3, 1),
    nn.Sigmoid(),
)
```

Activation Functions beyond Sigmoid/Logistic

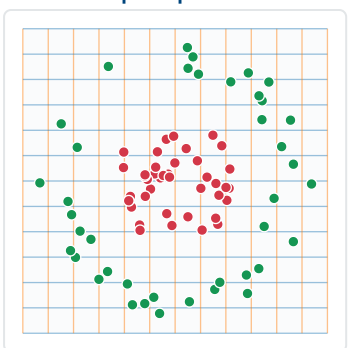


	$f(z)$	PyTorch	used in
sigmoid	$\frac{1}{1+e^{-z}}$	`nn.Sigmoid()`	outputs
tanh	$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	`nn.Tanh()`	RNNs
ReLU	$\max(0, z)$	`nn.ReLU()`	CNNs
GELU	$z\,\Phi(z)$	`nn.GELU()`	Transformers
Leaky ReLU	$\max(0.1z, z)$	`nn.LeakyReLU(0.1)`	GANs, CNNs
sine	$\sin(\omega_0 z)$	`torch.sin(30*z)`	implicit neural repr.

Why Non-Linear Activations?

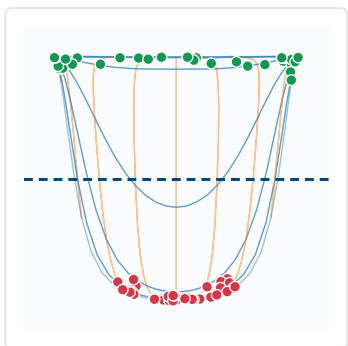
Without activations, stacked layers stay linear.

input space x



2 layers

after the network



Without activation: layers collapse into one

$$\mathbf{W}_2\mathbf{W}_1\mathbf{x} = \mathbf{W}\mathbf{x}$$

With activation: the space **bends**

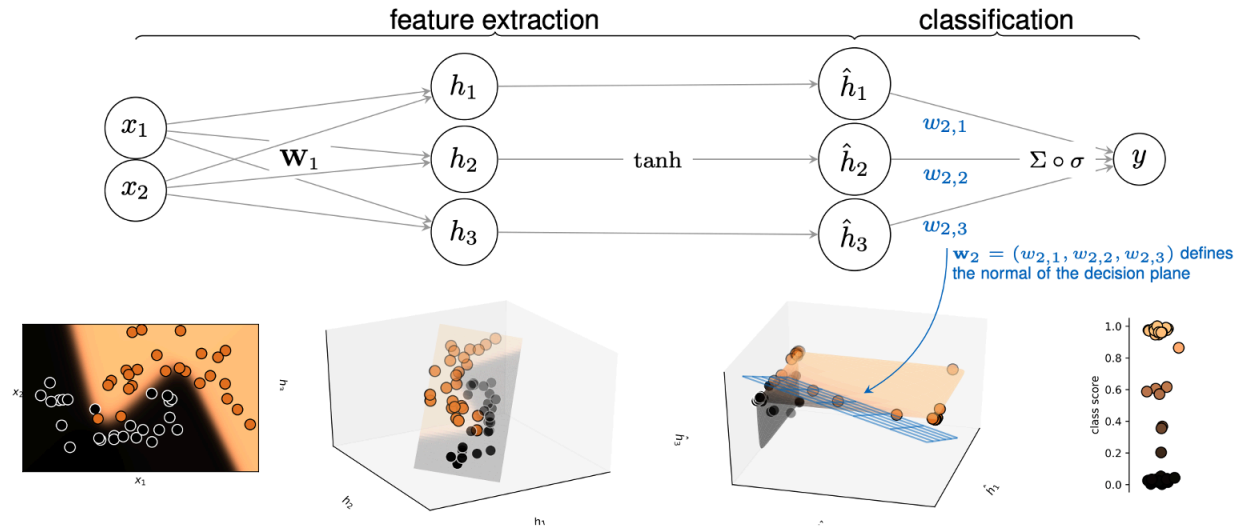
Bending makes the classes linearly separable.

none | tanh | ReLU

transform

the grid bends: a line now separates the classes

More layers → data becomes linearly separable



(a) Input space $\mathbf{x} = (x_1, x_2)$

(b) linear projection $\mathbf{w}_1^T\mathbf{x}$ into $\mathbb{R}^3$

(c) non-linear distortion with $\tanh(\mathbf{w}_1^T\mathbf{x})$

(d) Output class probabilities $y \in [0, 1]$

Demo: Multi-Layer Perceptron Feature Space

Circle | XOR | Two clusters

Train | Step | Reset model

Rate 0.03 Step 0 · loss 0.716

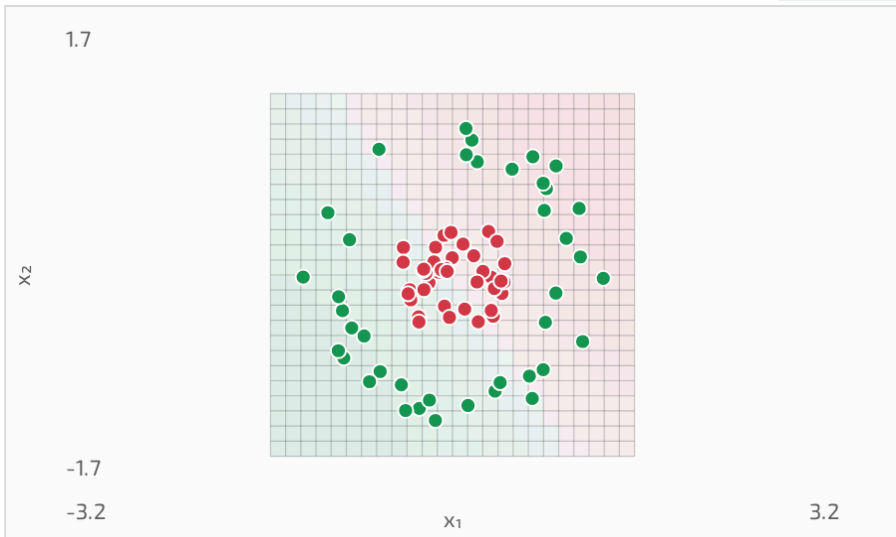
PyTorch model (edit me) reset

```
model = nn.Sequential(
    nn.Linear(2, 6),
    nn.Tanh(),
    nn.Linear(6, 2),
    nn.Tanh(),
    nn.Linear(2, 2),
)
```

Show layer:

input | Linear 1 | tanh 1 | Linear 2 | tanh 2 | logits

Input — neurons 1 and 2 of 2 Cycle pair

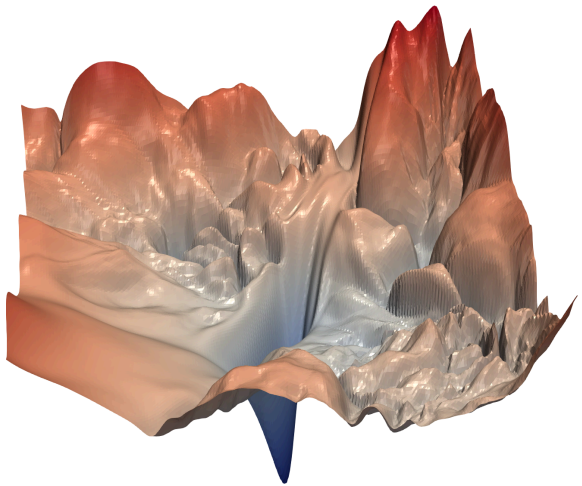


Hover a sample to follow it through the network.
Click: add A · Shift+click: add B · Ctrl/⌘+click: remove nearest
● A ● B · Background: final prediction · Lines: transformed input grid

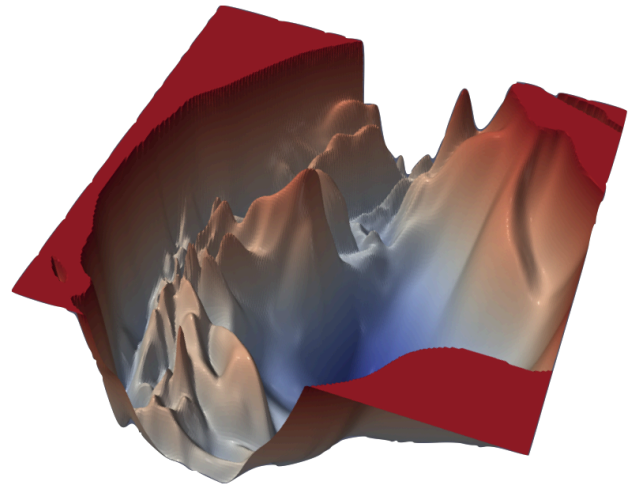
Inspired by Andrej Karpathy's ConvNetJS Classify2D demo. Implemented with Claude and Codex.

More Parameters → Harder Optimization

Deep networks have rugged loss landscapes: many local minima, saddle points and plateaus.



ResNet-56 without skip connections



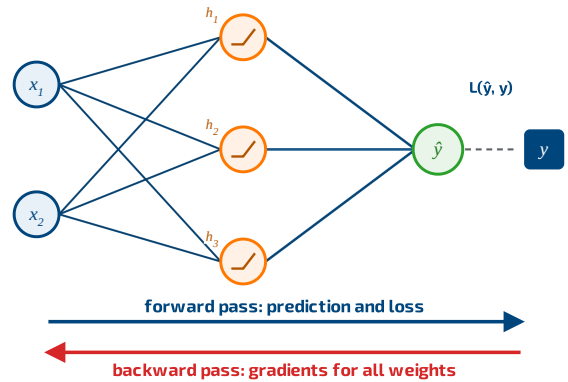
ResNet-110 without skip connections

Figures: H. Li, Z. Xu, G. Taylor, C. Studer, T. Goldstein, *Visualizing the Loss Landscape of Neural Nets*, NeurIPS 2018.

Backpropagation and Parameter Update

Backpropagation computes the gradients, gradient descent uses them.

		PyTorch
1 · Forward pass	$\hat{y} = f_{\mathbf{W}}(\mathbf{x})$	`y_pred = model(x)`
2 · Loss	$L(\hat{y}, y)$	`loss = mse(y_pred, y)`
3 · Backward pass	$\partial L / \partial \mathbf{W}$	`loss.backward()`
4 · Update	$\mathbf{W} \leftarrow \mathbf{W} - \eta \, \partial L / \partial \mathbf{W}$	`optimizer.step()`



Backpropagation: Step by Step

1 · Forward pass
2 · Loss
3a · Backward: output weight w_1
3b · Backward: to hidden neuron h_1
3c · Backward: input weight W_{11}
3d · Backward: all weights

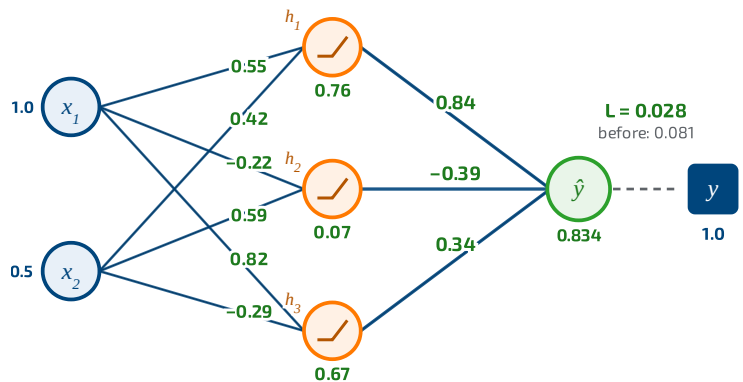
4 · Update

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}, \qquad \eta = 0.1$$

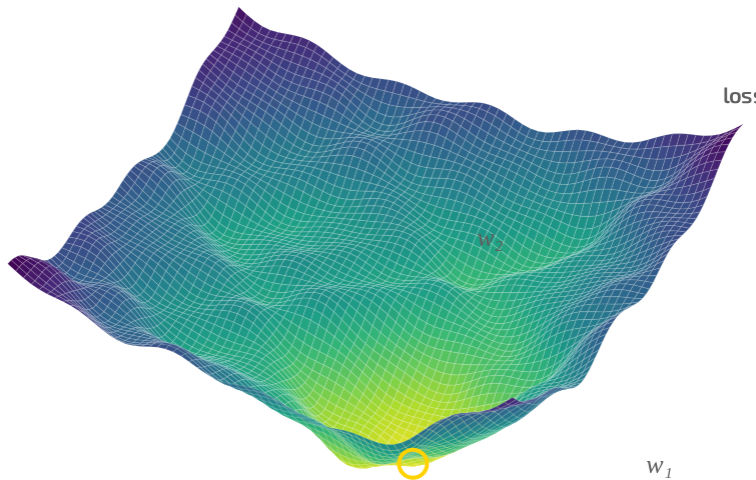
$$w_1 : \; 0.80 - 0.1 \cdot (-0.40) = 0.84$$

$$W_{11} : \; 0.50 - 0.1 \cdot (-0.46) = 0.55$$

Forward again: L: $0.081 \rightarrow 0.028$



Tricks to Navigate Complex Loss Surfaces



optimizers

GD | Momentum | RMSProp | Adam

learning rate **0.050**

momentum β **0.90**

gradient noise (mini-batch) **0.00**

dropout p **0.00**

random start | 20 random starts | clear

click the surface: start · drag: rotate · ○ global minimum

Checklist: Getting a Network to Train

Do this	Why (loss surface)	PyTorch
Normalise the inputs (per band)	round bowl, not a narrow valley	`x = (x - mean) / std`
Use Adam, or SGD with momentum	rolls over bumps and saddles	`optim.Adam(model.parameters(), lr=1e-3)`
Tune the learning rate first	too high: oscillates, too low: stuck	try `lr` = 1e-2, 1e-3, 1e-4
Lower the learning rate later	explore first, then settle	`lr_scheduler.ReduceLROnPlateau(opt)`
Train on mini-batches	noisy gradients escape saddles	`DataLoader(ds, batch_size=64, shuffle=True)`
Several seeds or **pretrained weights**	other or better starting points	`torch.manual_seed(1)`
Regularise	small weights, robust features	`AdamW( … , weight_decay=1e-4)` , `nn.Dropout(0.2)`
Plot train and validation loss	spot divergence and overfitting	stop when validation loss rises
First overfit one small batch	loss → 0, otherwise: a bug	train on 10 samples only

Takeaways - Multi-Layer Perceptrons

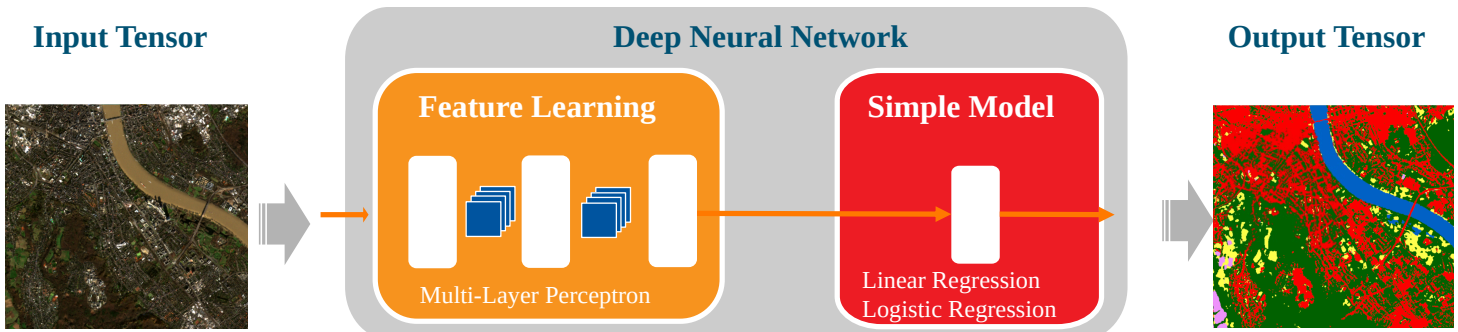
	Linear regression	Logistic regression	Multi-layer perceptron
Model	$\mathbf{W}\mathbf{x} + \mathbf{b}$	$\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$	$\mathrm{head}\big(\mathbf{W}_2\,\mathrm{act}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2\big)$
Features	hand-designed	hand-designed	**learned** in the hidden layers
Typical loss	MSE	cross-entropy	MSE or cross-entropy
Finding the weights	closed form	gradient descent	backpropagation + gradient descent

An MLP stacks linear transformations with non-linear activations. The hidden layers bend the feature space until a simple linear or logistic head is enough — with enough neurons, any function can be approximated.

The price: a rugged, non-convex loss surface. Backpropagation gives all gradients in one sweep; the optimizer, learning rate, initialization and regularization decide where we end up.

Lecture 4 - Feature Learning

From Linear Models to Multi-Layer Perceptrons



BLOCK 1
Linear Regression

BLOCK 2
Logistic Regression

BLOCK 3
Multilayer Perceptrons

Takeaways - Lecture 4

- **Linear regression:** $\mathbf{W}\mathbf{x} + \mathbf{b}$ predicts values
- **Logistic regression:** add a sigmoid / softmax to predict classes
- **MLPs:** stack layers with activations to **learn the features**
- **Training:** backpropagation computes the gradients, gradient descent uses them

> **No more hand-designed features: feed in the raw data and let the model learn the most discriminative features — trained end-to-end.**

	brute force	closed form	gradient descent
linear regression	✓	✓	✓
logistic regression	✓	✗	✓
MLP	✗	✗	✓

Revisit the learning outcomes

Light gray fill: your group's before responses today · Click to respond again.

Explain linear regression, from one to many inputs and outputs, and how minimizing MSE fits its parameters.

Not comfortable — Very comfortable

Click to respond

Explain how sigmoid and softmax turn linear outputs into class probabilities.

Not comfortable — Very comfortable

Click to respond

Explain how MSE and cross-entropy losses and gradient descent guide parameter learning.

Not comfortable — Very comfortable

Click to respond

Describe artificial neurons and layers as affine maps followed by nonlinear activations.

Not comfortable — Very comfortable

Click to respond

Explain how MLPs learn intermediate representations.

Not comfortable — Very comfortable

Click to respond

Explain how backpropagation computes the gradient of the loss for every weight.

Not comfortable — Very comfortable

Click to respond