The deep learning training workflow

As illustrated with the simple case of linear regression

If you understand linear regression deeply you are well on your way to understanding deep learning

The “canonical” deep learning training workflow

- **Data** → collect, clean, **supervise**, split (train/val/test), augment
  - Garbage in, garbage out: this is where most projects live or die
- **Model Architecture** → choose a model that well suited to the data
  - MLP (simple data), CNN (grids/Images), RNN/Transformer (sequences), GNN (graphs)
- **Objective** (prediction quality scoring mechanism) → define a loss $\mathcal{L}(\theta)$ that encodes "good"
  - Regression: Mean square error MSE
  - Binary classification: Binary Cross-entropy
  - Multi-class classification: Categorical Cross-entropy
- **Optimization** (learning) → back-prop + SGD/Adam, LR schedule, regularization
  - We are just rolling downhill on a very high-dimensional surface
- **Evaluation** → validation metrics, error analysis, ablations
  - Tune on val, touch test *once*
- **Deployment** → export, serve, monitor drift, iterate

Aside: "Canonical" means conforming to a standard, authoritative, or generally accepted form the version of something that serves as the reference point against which variants are judged (e.g., the canonical form of a matrix, or the canonical version of a text).

Raw Data

Collect data on houses

Resultant cleaned Pandas data-frame

s a p → Temporary variable name

Sample (batch) dimension

i	sqft 1000 ft²	age years	price $1000s
0	1.45	12	365.4
1	0.98	45	219.8
2	2.10	3	498.6
3	1.32	28	308.1
4	2.85	8	625.7
5	1.70	35	366.2
⋮			

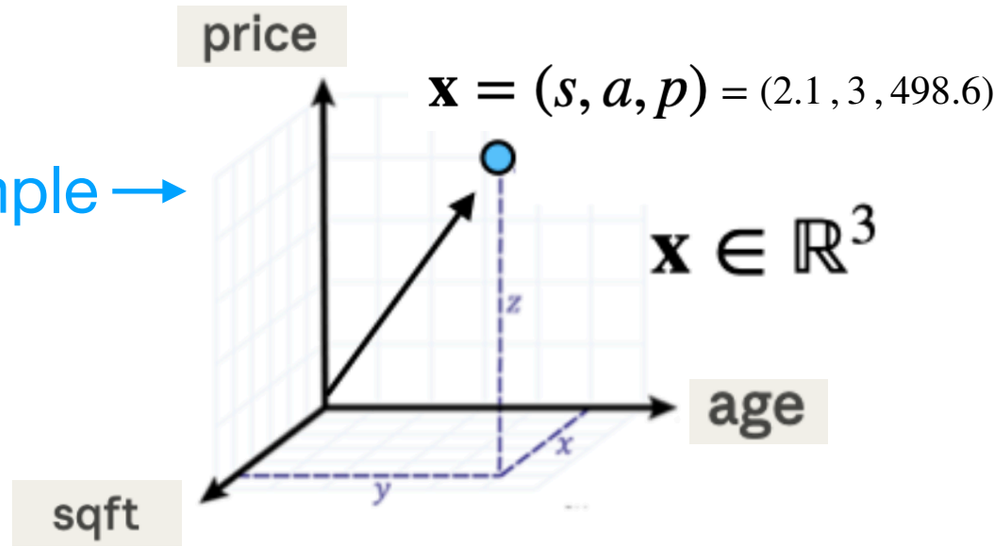
i^{th} sample →

Unit of observation = house

Let's assume there are 100 rows

Full Data shape: (100, 3)

Data Space



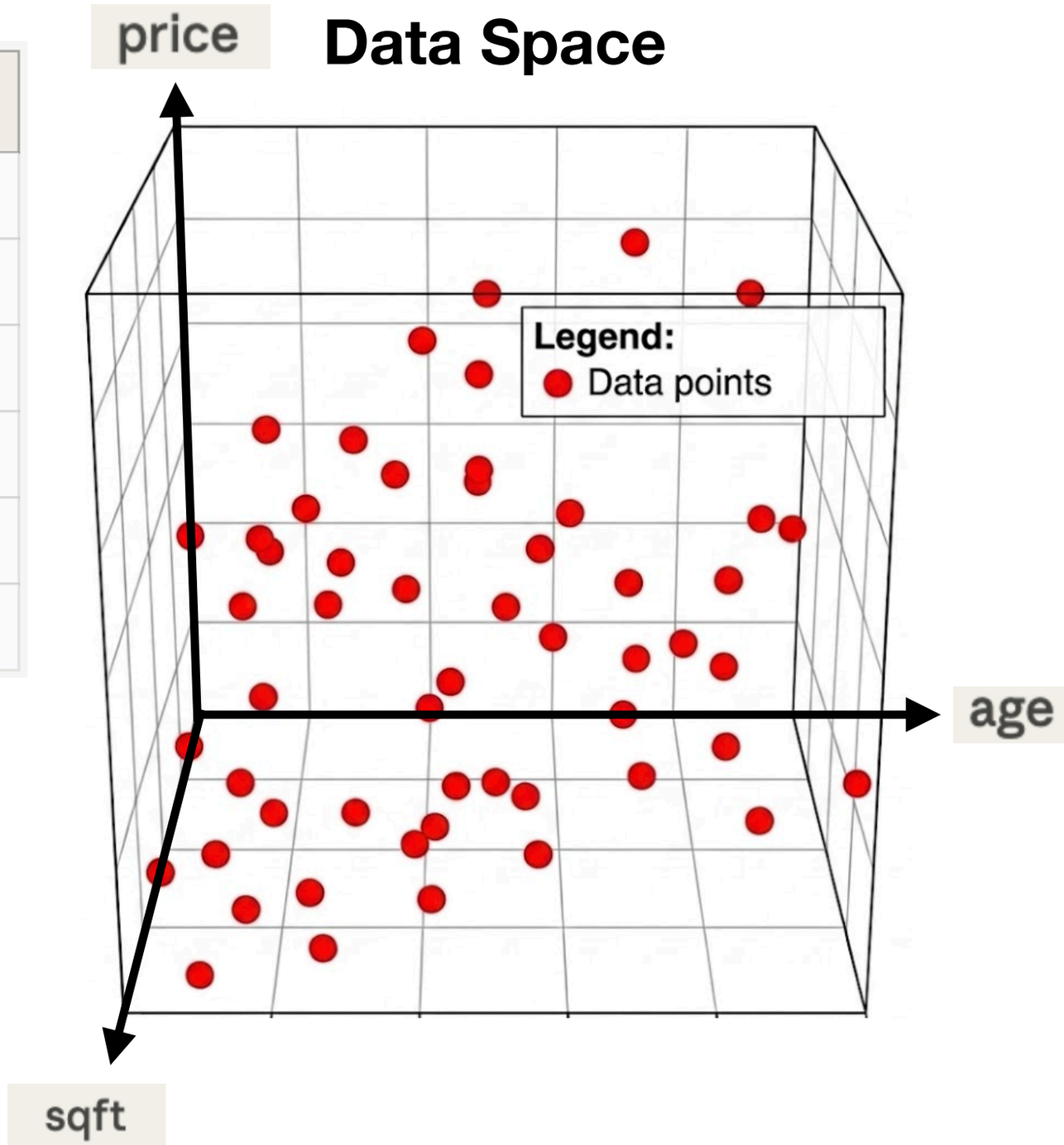
Full data-set lives in $\mathbb{R}^3$

Basic Exploratory data analysis (EDA)

Since the data is so simplistic, there's not much to do here other than visualizing a standard point cloud.

Sample (batch) dimension

i	sqft 1000 ft²	age years	price $1000s
0	1.45	12	365.4
1	0.98	45	219.8
2	2.10	3	498.6
3	1.32	28	308.1
4	2.85	8	625.7
5	1.70	35	366.2

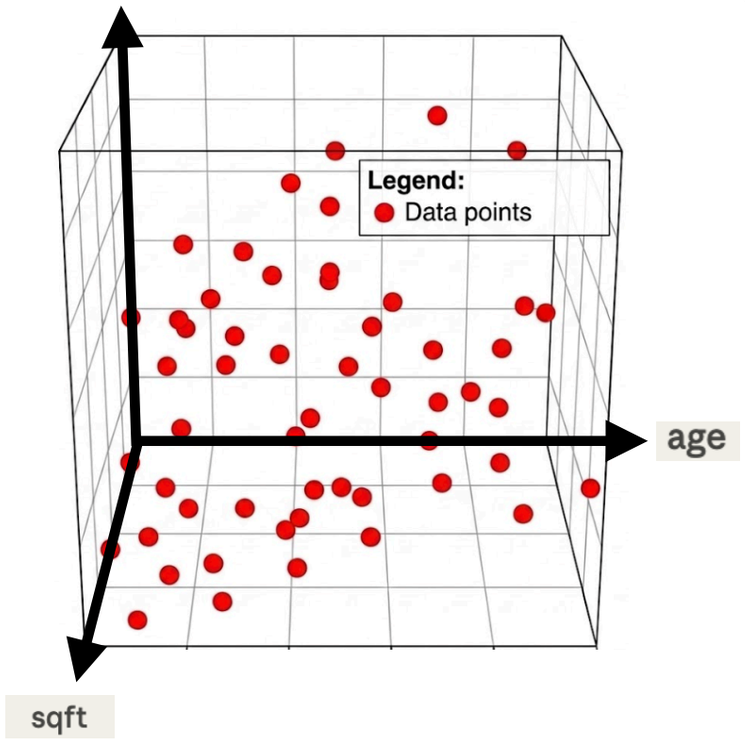


Clearly there is a numeric relationship between the variables.

Aside: Normalization

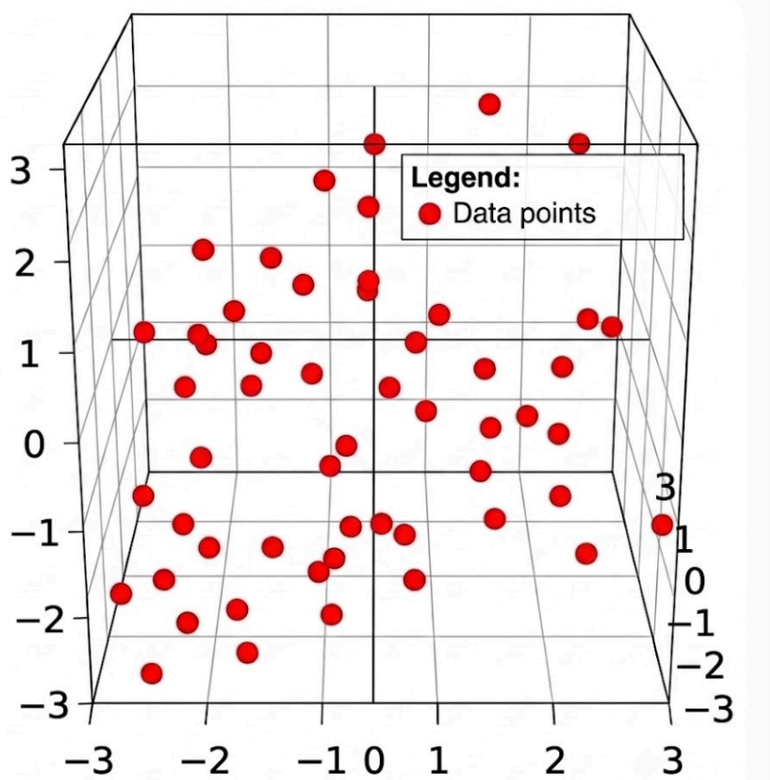
Normalization is very important in deep learning. It is a general procedure. It makes the models train more efficiently. They tend to work better with numbers on the order of negative ten to ten centered around the origin.

Data with real units (e.g. years)



$$z = \frac{x - \mu}{\sigma}$$

Data on standarized scale



i	sqft 1000 ft²	age years	price $1000s
0	1.45	12	365.4
1	0.98	45	219.8
2	2.10	3	498.6
3	1.32	28	308.1
4	2.85	8	625.7
5	1.70	35	366.2

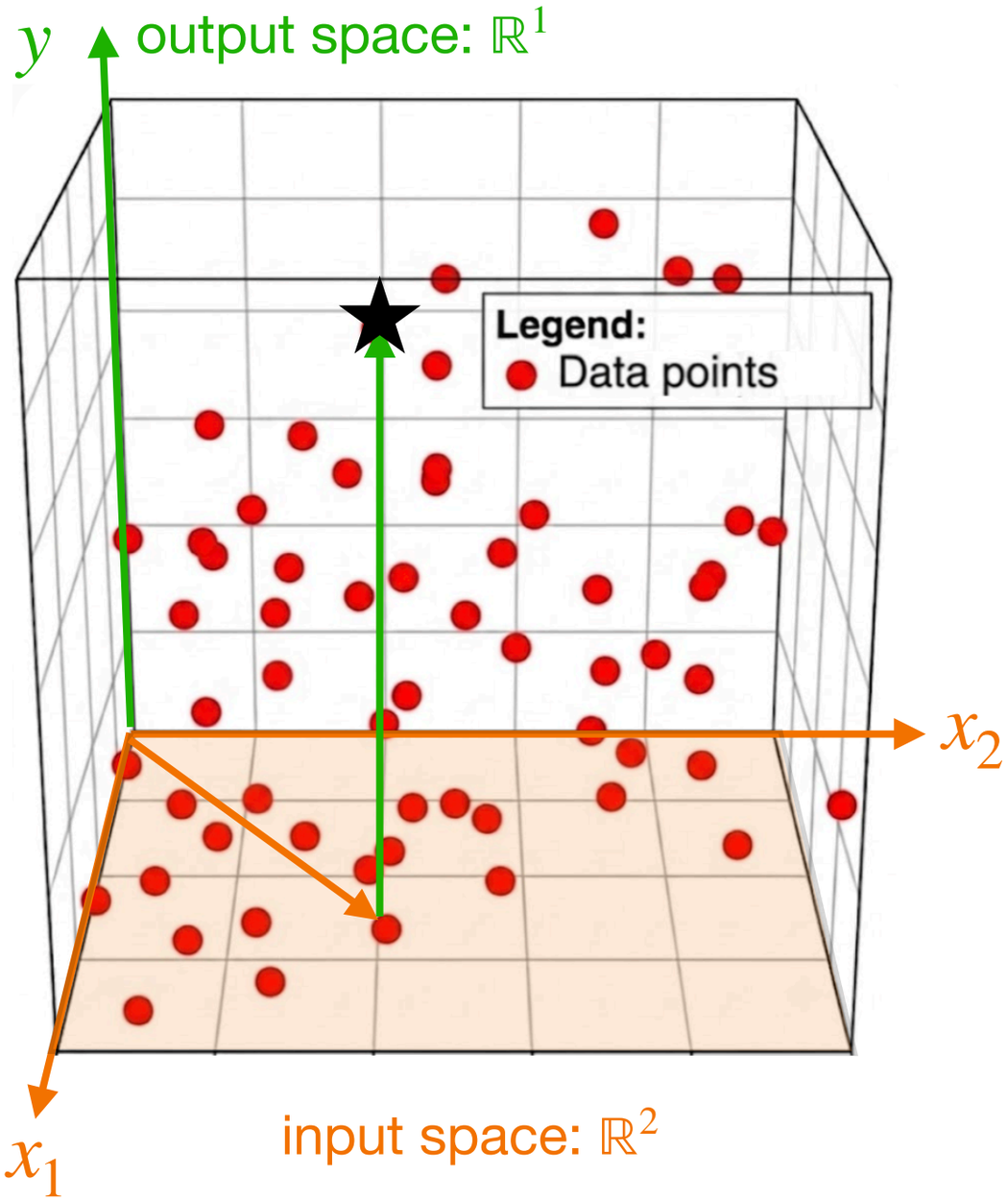
i	sqft z units	age z units	price z units
0	−0.47	−0.65	−0.24
1	−1.24	1.52	−1.35
2	0.61	−1.24	0.77
3	−0.68	0.41	−0.68
4	1.84	−0.91	1.74
5	−0.05	0.86	−0.24

μ	1.733	21.83	397.30
σ	0.606	15.23	131.58

below the column mean above the column mean

Supervision

In this case the supervision lies in the act of establishing or hypothesizing the relationship between the input space and the output space.



input space: x_1 x_2 — output space: y

	input space: $\mathbb{R}^2$		output space: $\mathbb{R}^1$
i	sqft $x_1 \cdot 1000\ ft^2$	age $x_2 \cdot$ years	price $y \cdot \$1000s$
0	1.45	12	365.4
1	0.98	45	219.8
2	2.10	3	498.6
3	1.32	28	308.1
4	2.85	8	625.7
5	1.70	35	366.2

★

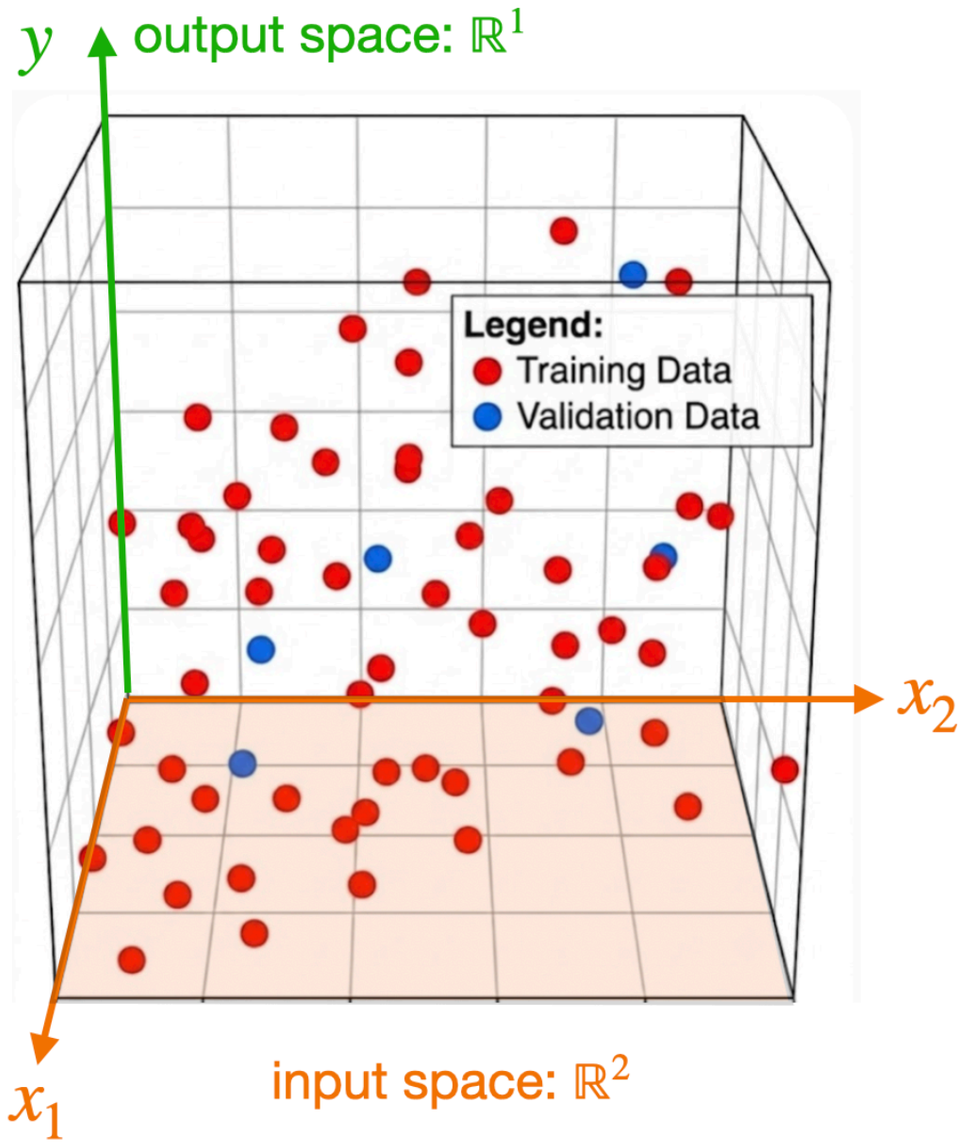
position on the floor — height above the floor

$$\mathbf{x} = (x_1, x_2) \xrightarrow{\text{functional relationship}} y$$

Note that in reality the data is generally much higher dimensional and the existence of the relationship cannot be inferred visually and instead hypothesized and verified during training.

Data split

Deep learning models are very prone to overfitting, so Splitting our data into tra tra tra tra tra tra tra tra tra tra tra training and validation or training test and validation is a non negotiable practice.



x_1 x_2 y

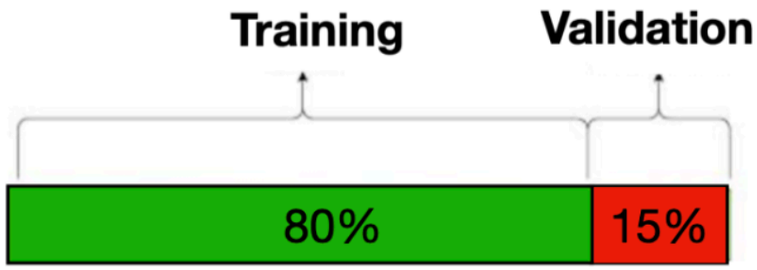
	input space: $\mathbb{R}^2$		output space: $\mathbb{R}^1$	not a feature
i	sqft x_1 · 1000 ft²	age x_2 · years	price y · $1000s	split
0	1.45	12	365.4	train
1	0.98	45	219.8	train
2	2.10	3	498.6	validation
3	1.32	28	308.1	train
4	2.85	8	625.7	train
5	1.70	35	366.2	validation

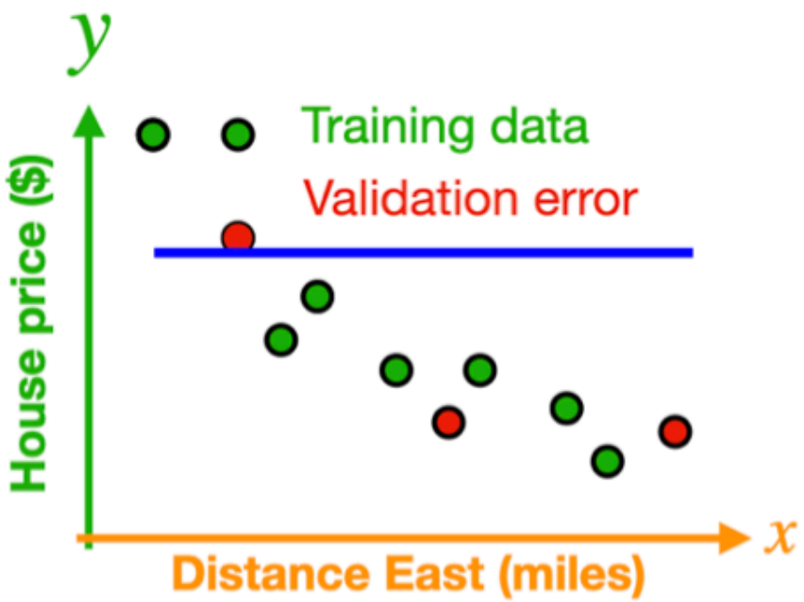
position on the floor height above the floor red / blue dots in the figure

Note that in reality the data is generally much higher dimensional and the existence of the relationship cannot be inferred visually and instead hypothesized and established during training.

Recall why we need the validation set!

monitor and avoid overfitting!



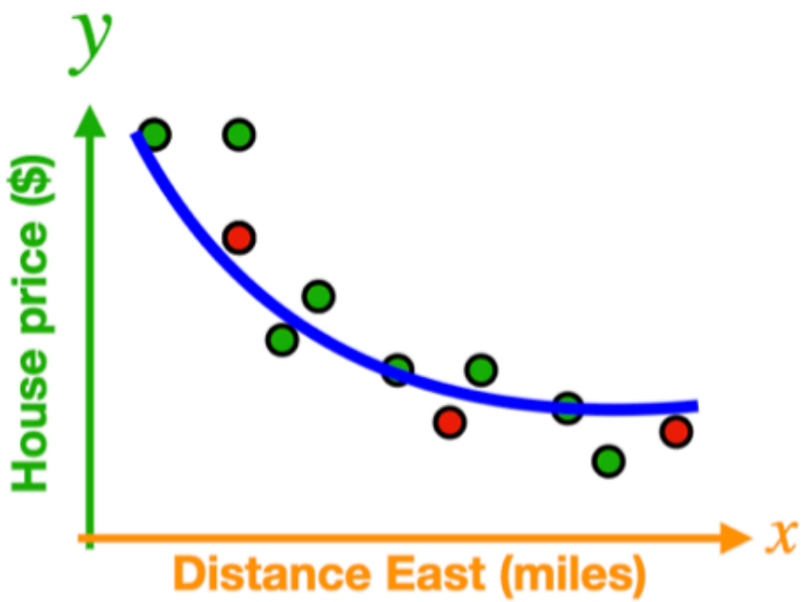


under-fitting
model complexity is too low

High training error
High Validation error

high bias

low variance

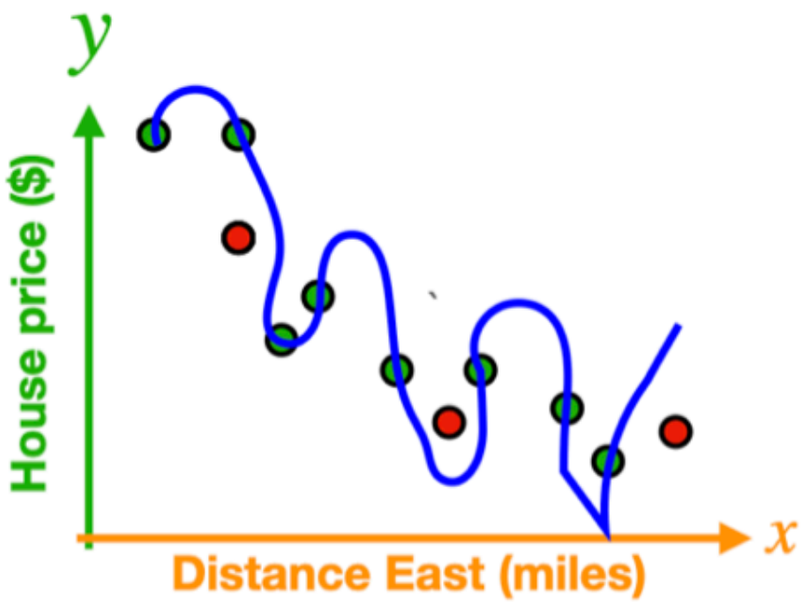


optimal
correct level of model complexity

Training and Validation error
are equally low

low bias

low variance



over-fitting
model complexity is too high

Very Low training error
Very high Validation error

low bias

high variance

The “canonical” deep learning training workflow

- **Data** → collect, clean, **supervise**, split (train/val/test), augment
  - Garbage in, garbage out: this is where most projects live or die
- **Model Architecture** → choose a model that well suited to the data
  - MLP (simple data), CNN (grids/Images), RNN/Transformer (sequences), GNN (graphs)
- **Objective** (prediction quality scoring mechanism) → define a loss $\mathcal{L}(\theta)$ that encodes "good"
  - Regression: Mean square error MSE
  - Binary classification: Binary Cross-entropy
  - Multi-class classification: Categorical Cross-entropy
- **Optimization** (learning) → back-prop + SGD/Adam, LR schedule, regularization
  - We are just rolling downhill on a very high-dimensional surface
- **Evaluation** → validation metrics, error analysis, ablations
  - Tune on val, touch test *once*
- **Deployment** → export, serve, monitor drift, iterate

Aside: "Canonical" means conforming to a standard, authoritative, or generally accepted form the version of something that serves as the reference point against which variants are judged (e.g., the canonical form of a matrix, or the canonical version of a text).

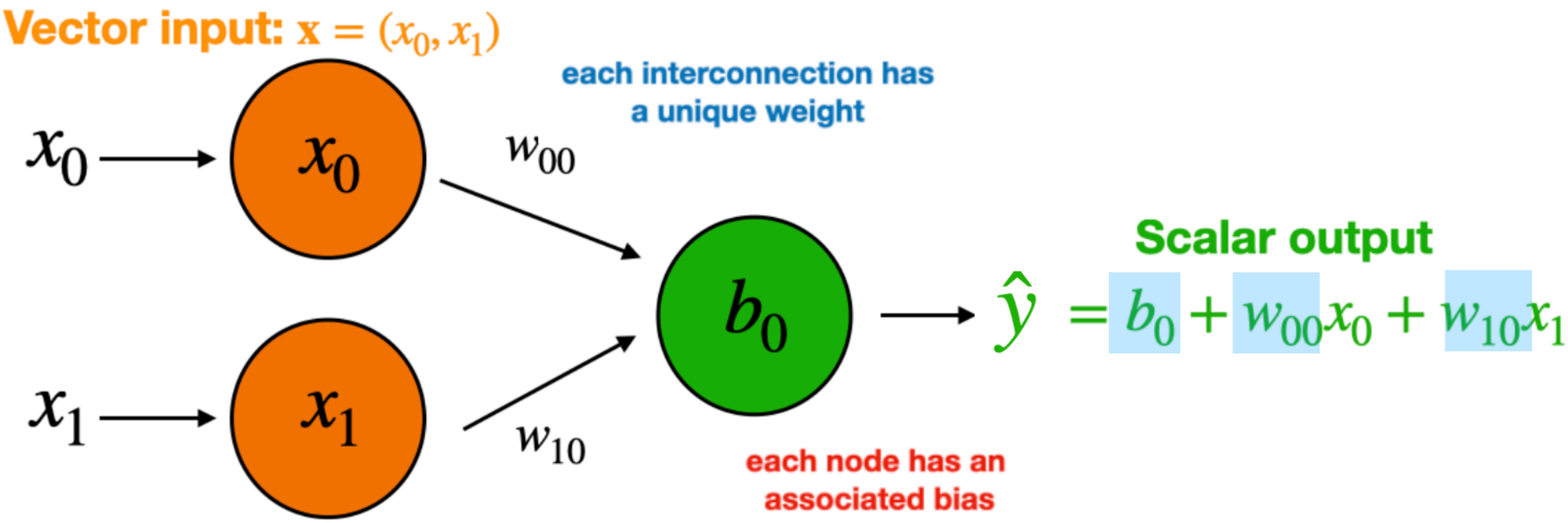
Choosing a model: For now consider a linear model

model=prediction machine: $\hat{y} = M(\mathbf{x}|\mathbf{w})$

- For example, consider a **two** variable $\mathbf{x} = (x_0, x_1)$ linear scalar regression model

$$\hat{y} = M(\mathbf{x}|\mathbf{w}) = b_0 + w_0 x_0 + w_1 x_1$$

- We can represent this as a "computational graph"



- **Problem Type:** Scalar-regression
- **Mapping:** $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ (input layer) (output layer)
- **Hidden layers**: None
- **Hidden-layer activation**: N/A
- **Output-layer activation**: linear
- **Number of fitting parameters**: 3

Bivariate linear regression = Our first "neural network" model

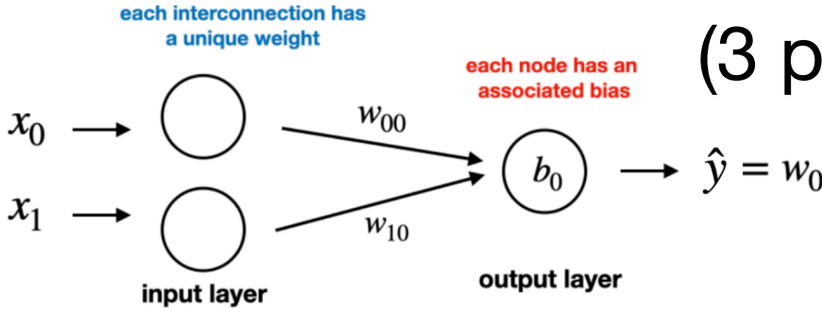
For educational purposes let's drop a feature to simplify the math

	input space: $\mathbb{R}^2$		output space: $\mathbb{R}^1$	not a feature
i	sqft x_1 · 1000 ft²	age x_2 · years	price y · $1000s	split
0	1.45	12	365.4	train
1	0.98	45	219.8	train
2	2.10	3	498.6	validation
3	1.32	28	308.1	train
4	2.85	8	625.7	train
5	1.70	35	366.2	validation

position on the floor height above the floor red / blue dots in the figure

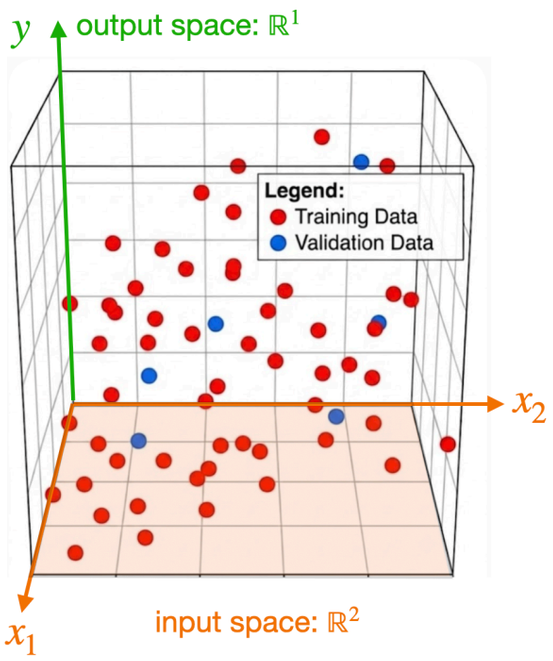
x_1 y

i	sqft x_1 · 1000 ft²	price y · $1000s	split
0	1.45	365.4	train
1	0.98	219.8	train
2	2.10	498.6	validation
3	1.32	308.1	train
4	2.85	625.7	train
5	1.70	366.2	validation

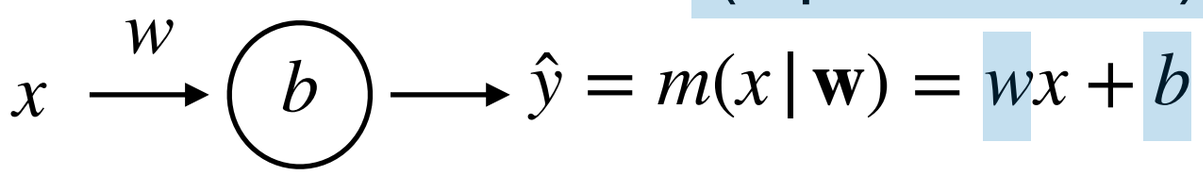


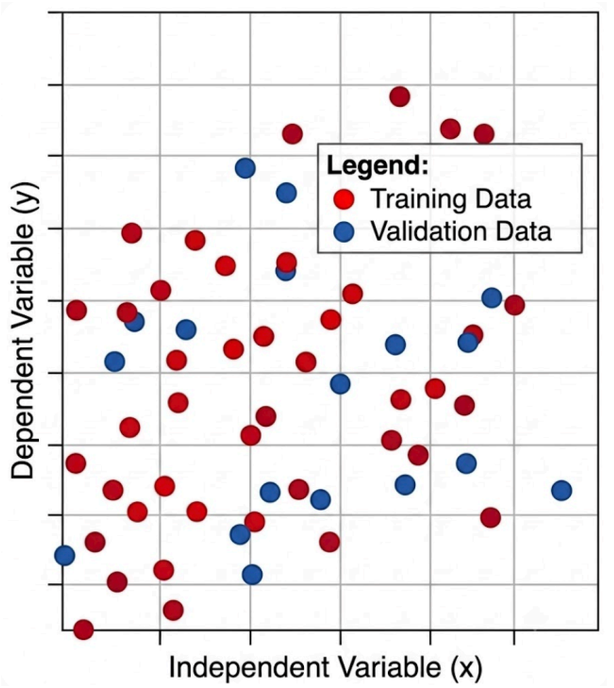
(3 parameters)

$$\hat{y} = w_{00}x_0 + w_{10}x_1 + b_0$$



(2 parameters)





The “canonical” deep learning training workflow

- **Data** → collect, clean, **supervise**, split (train/val/test), augment
  - Garbage in, garbage out: this is where most projects live or die
- **Model Architecture** → choose a model that well suited to the data
  - MLP (simple data), CNN (grids/Images), RNN/Transformer (sequences), GNN (graphs)
- **Objective** (prediction quality scoring mechanism) → define a loss $\mathcal{L}(\theta)$ that encodes "good"
  - Regression: Mean square error MSE
  - Binary classification: Binary Cross-entropy
  - Multi-class classification: Categorical Cross-entropy
- **Optimization** (learning) → back-prop + SGD/Adam, LR schedule, regularization
  - We are just rolling downhill on a very high-dimensional surface
- **Evaluation** → validation metrics, error analysis, ablations
  - Tune on val, touch test *once*
- **Deployment** → export, serve, monitor drift, iterate

Aside: "Canonical" means conforming to a standard, authoritative, or generally accepted form the version of something that serves as the reference point against which variants are judged (e.g., the canonical form of a matrix, or the canonical version of a text).

Parameters are everything in deep learning!!

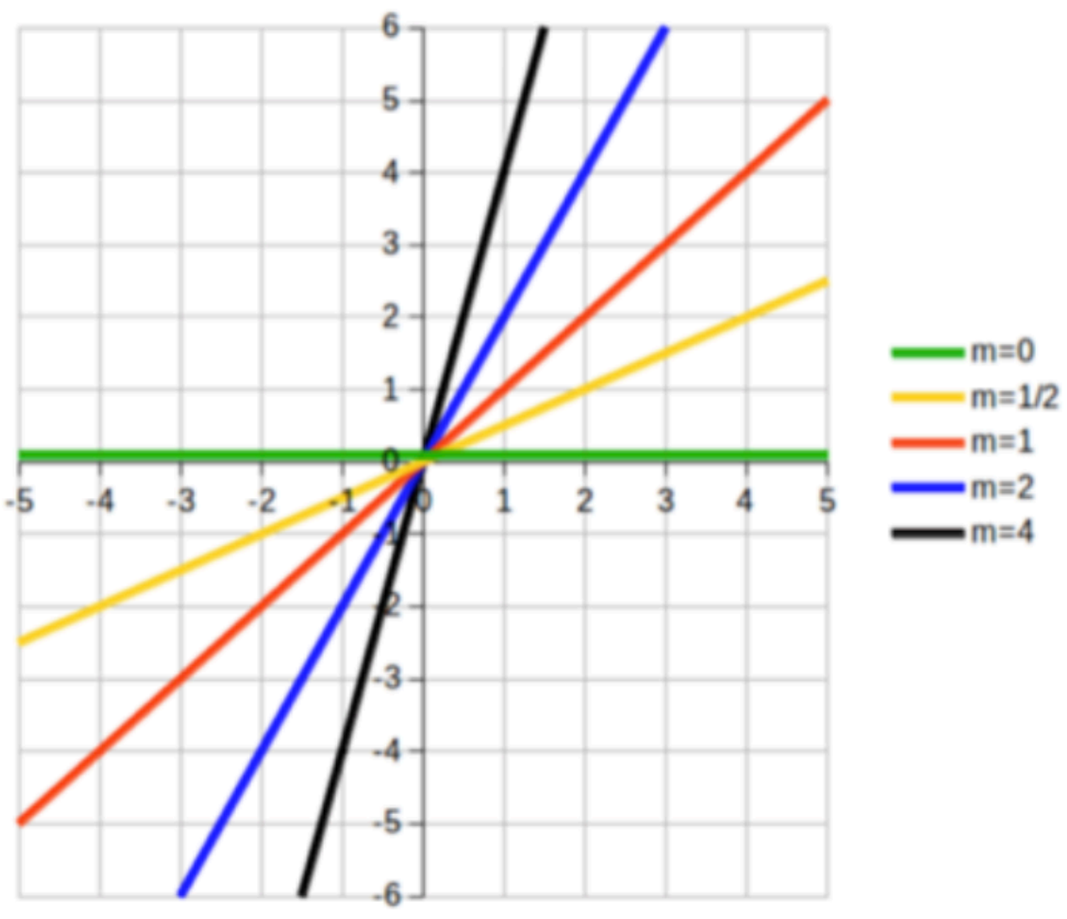
There are an infinite number of possible parameterizations of this plane

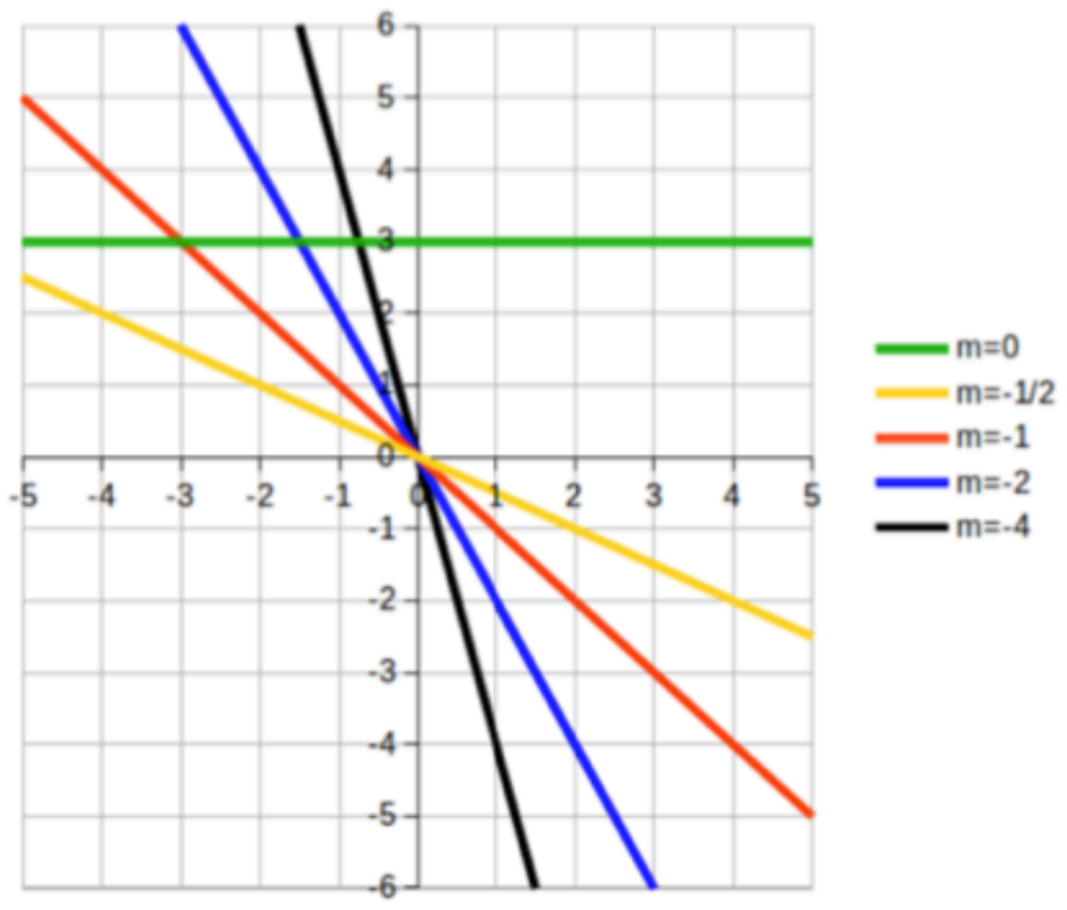
$$x \xrightarrow{w} \boxed{b} \longrightarrow \hat{y} = m(x \mid \mathbf{w}) = w_1 x + w_0 = mx + b$$

(2 parameters)

This model represents all possible lines in a two dimensional plane.

A particular choice of parameters instantiates a particular line.



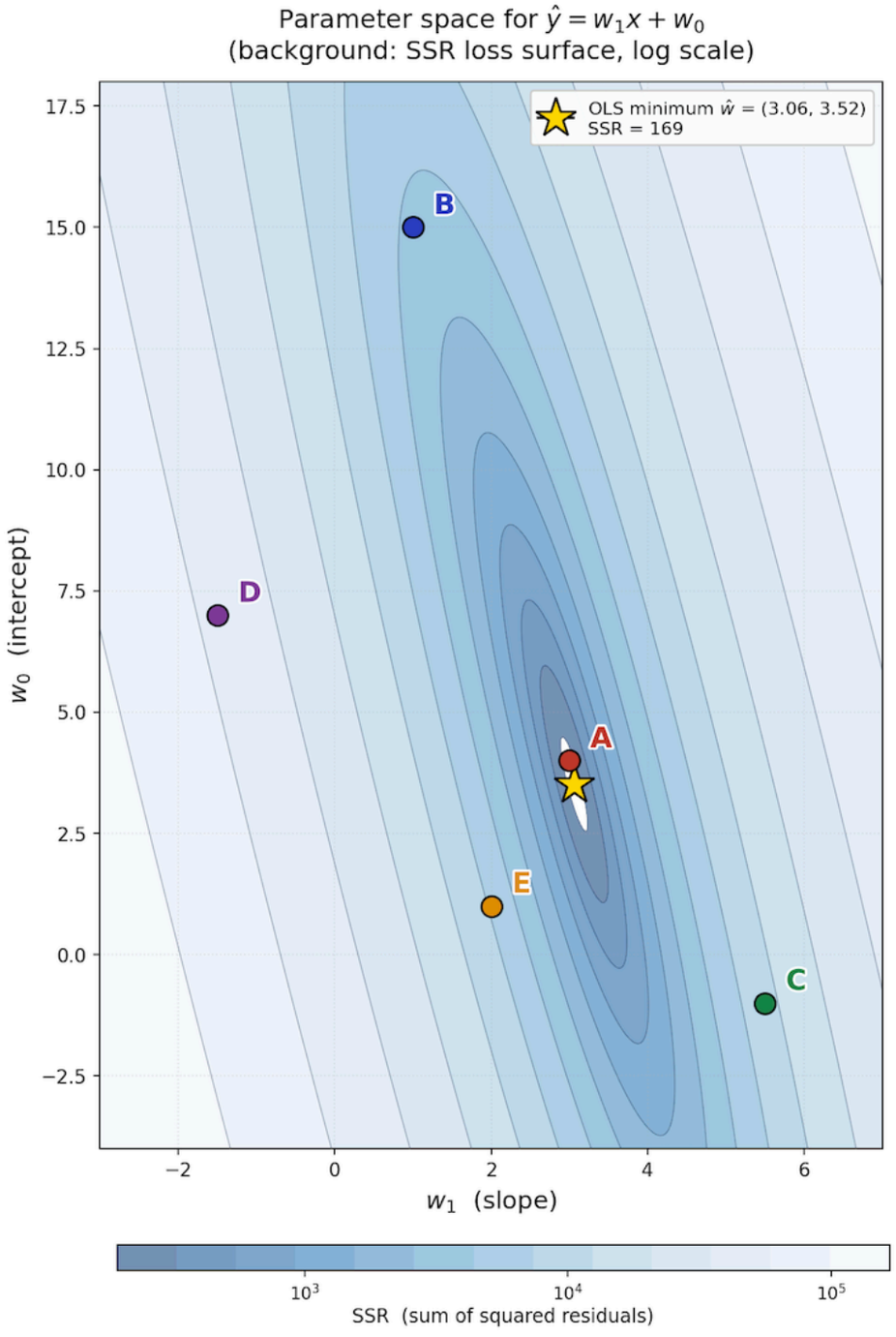


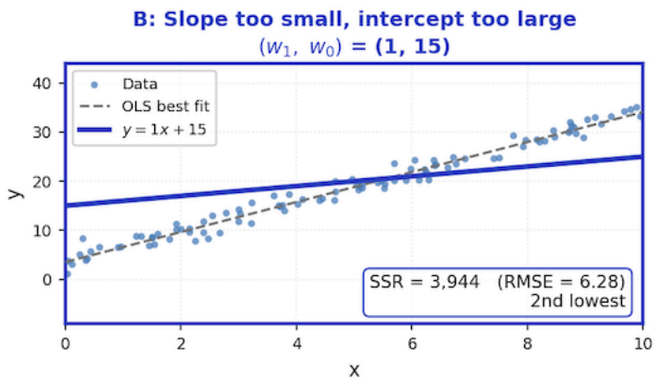
Parameters are what we learn in deep learning

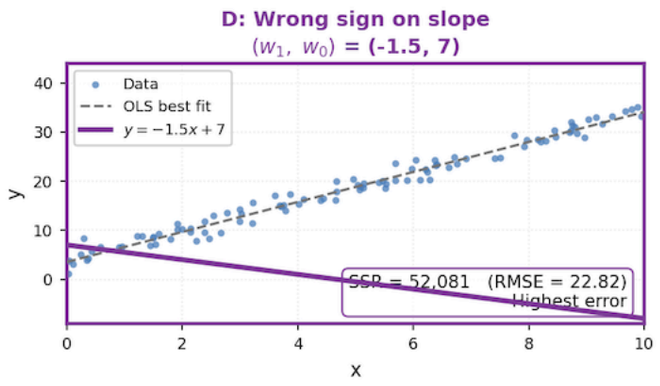
Parameter space

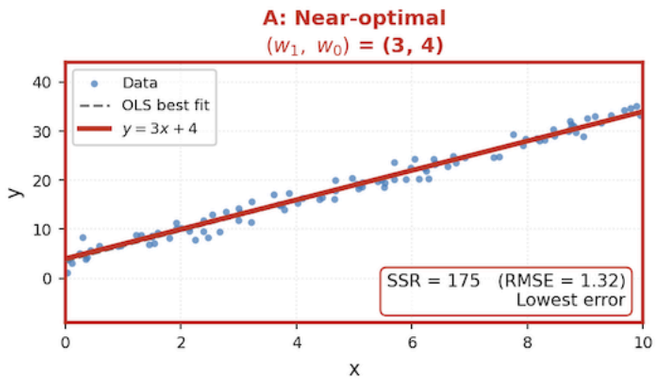
Data space

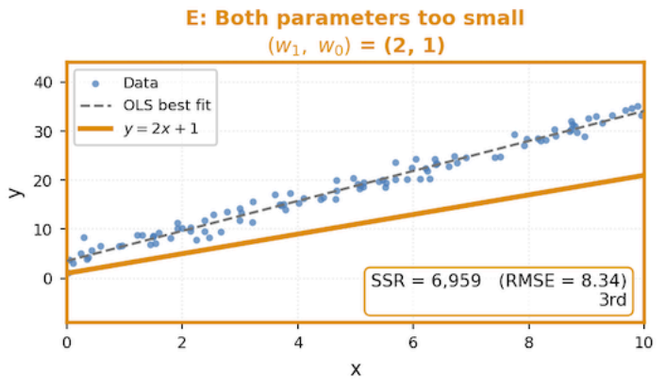
Parameter space ↔ data space: every (w_1, w_0) pair is one candidate line

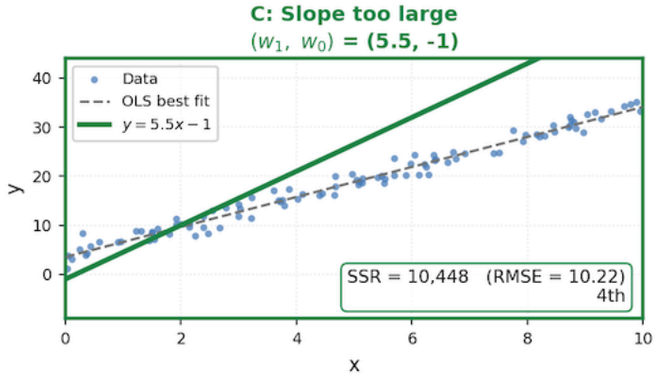













```
HOW TO READ THIS FIGURE

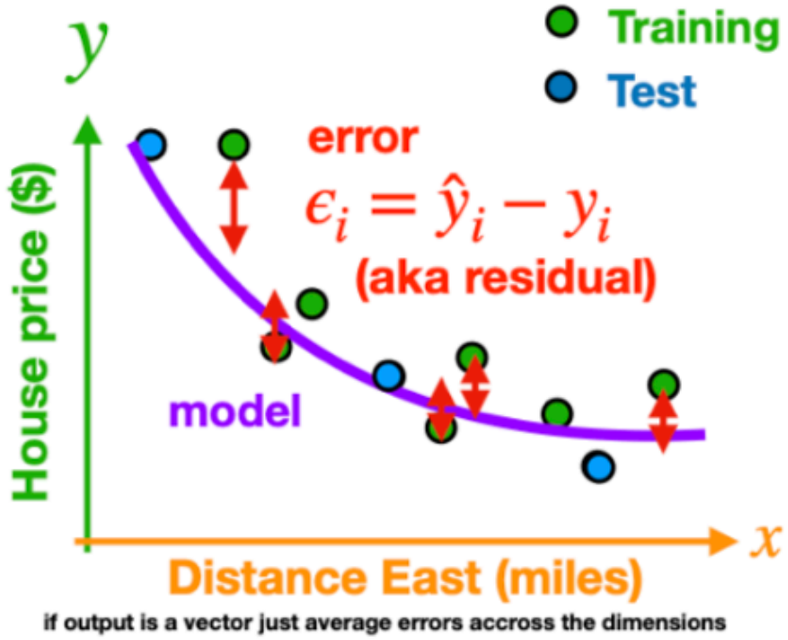
One point in parameter space (left)
= one candidate line in data space.

The contours are the SSR loss surface:
the farther a point sits from the gold
star (the OLS minimum), the larger its
SSR and the worse its line fits.

Data:  n = 100,  y = 3x + 4 + N(0, 1.5²)
SSR floor ≈ nσ² = 225

SSR ranking:
   A: SSR =      175
   B: SSR =    3,944
   E: SSR =    6,959
   C: SSR =   10,448
   D: SSR =   52,081
```

Regression cost functions



if output is a vector just average errors accross the dimensions

$$\underset{data}{\mathbf{y}} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \quad \underset{predictions}{\mathbf{m}(x \mid \mathbf{p}) = \hat{\mathbf{y}}} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{pmatrix} \quad \underset{errors}{\mathbf{y} - \hat{\mathbf{y}}} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{pmatrix}$$

Sum of square error (SSE)

$$SSE = \sum_i^N \epsilon_i^2 = \sum_i^N (y_i - \hat{y}_i)^2$$

no $\frac{1}{N}$

Mean square error (MSE) (variance of error distribution)

$$MSE = \frac{1}{N} \sum_i^N \epsilon_i^2 = \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2$$

Root mean square error (RMSE) (standard deviation of error distribution)

$$RMSE = \left(\frac{1}{N} \sum_i^N \epsilon_i^2 \right)^{\frac{1}{2}} = \left(\frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2 \right)^{\frac{1}{2}}$$

Mean absolute error (MAE) Less sensitive to outliers

$$MAE = \frac{1}{N} \sum_i^N |\epsilon_i| = \frac{1}{N} \sum_i^N |y_i - \hat{y}_i|$$

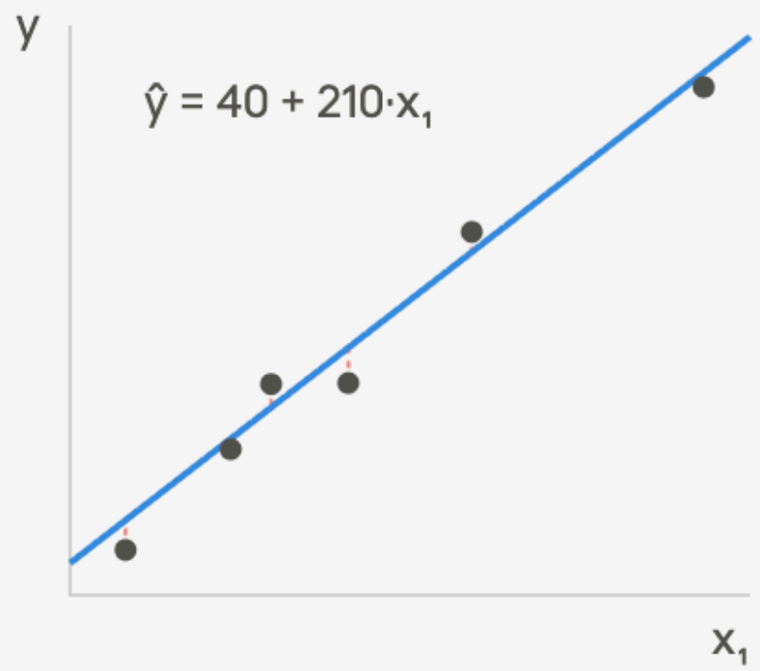
Scoring the quality of fit: SSE

$$\hat{y} = w_0 + w_1 \cdot x_1 \text{ with } p = (40, 210)$$

everything to the right of the dashed line depends on p

i	x_1 sqft	y price	$\hat{y}$ predicted	ε residual	ε^2 sq. error	split
0	1.45	365.4	344.5	+20.9	436.8	● train
1	0.98	219.8	245.8	−26.0	676.0	● train
2	2.10	498.6	481.0	+17.6	309.8	○ validation
3	1.32	308.1	317.2	−9.1	82.8	● train
4	2.85	625.7	638.5	−12.8	163.8	● train
5	1.70	366.2	397.0	−30.8	948.6	○ validation

given data | computed from p



436.8
676.0
309.8
82.8
163.8
948.6

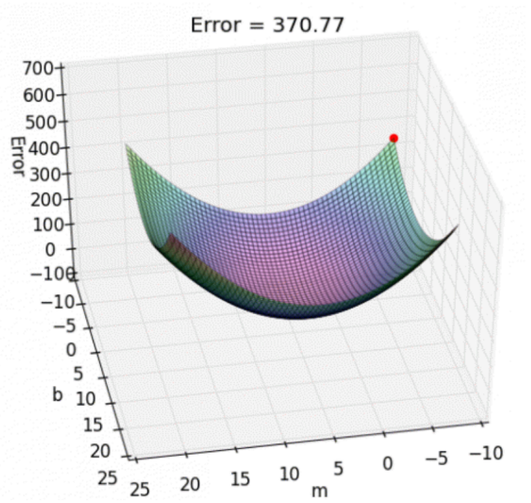
Σ → L(p) **2617.8**

one number per choice of p

Loss surface: Ordinary Least squares (OLS)

- Assume we have supervised training data $x_i \in (x_1, x_2 \dots x_N) \rightarrow y_i \in (y_1, y_2 \dots y_N)$
- Lets use a single variable linear model $y_{pred} = M(x|\mathbf{w}) = M(x|m, b) = mx + b$
- Let's also use mean square error (MSE) as our loss function:

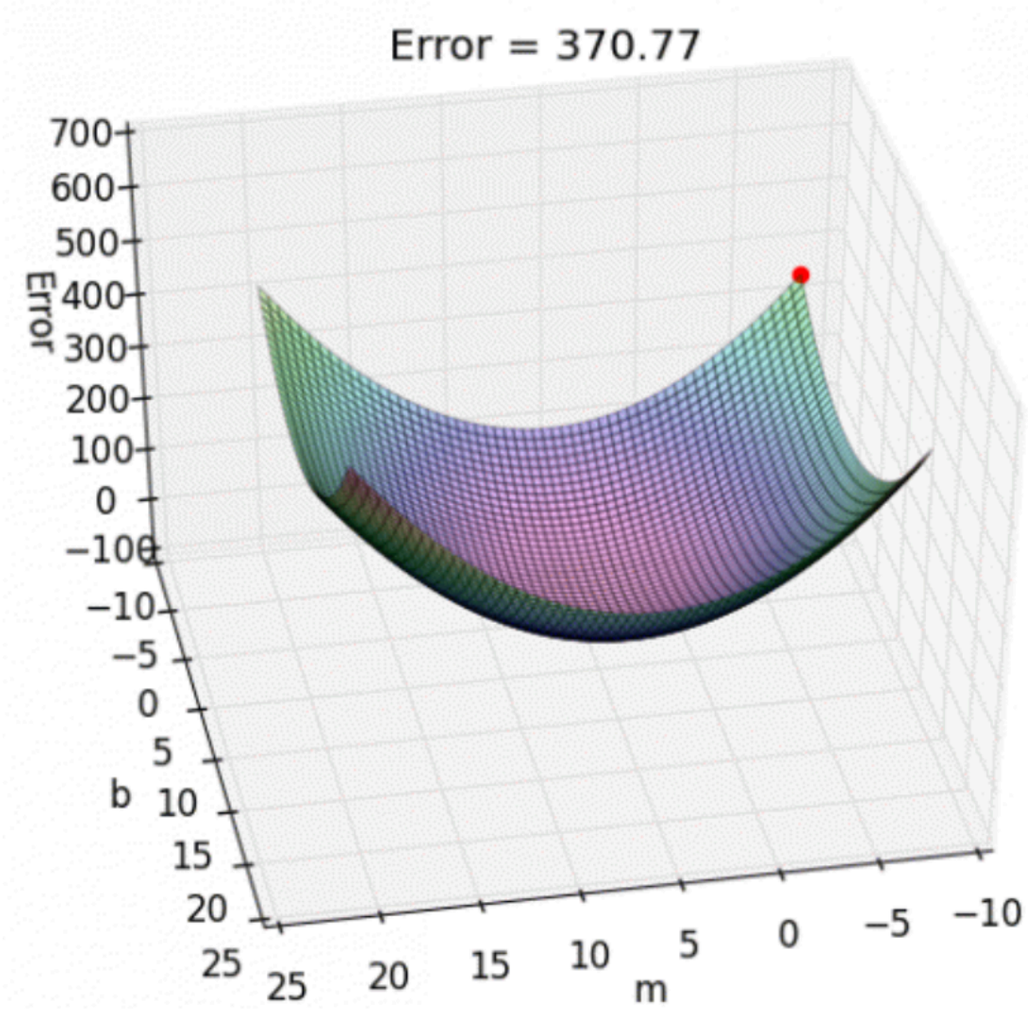
$$
\begin{aligned}
L(m,b) &= \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \quad (MSE) \\
&= \frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b))^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}(y_i - mx_i - b)(y_i - mx_i - b) \\
&= \frac{1}{N}\sum_{i=1}^{N} y_i^2 - mx_iy_i - y_ib - mx_iy_i + (mx_i)^2 + mx_ib - by_i + bmx_i + b^2 \\
&= \frac{1}{N}\sum_{i=1}^{N} y_i^2 - 2x_iy_im - 2y_ib + 2mx_ib + (mx_i)^2 + b^2 \\
&= \overline{y^2} + \overline{x^2}m^2 - 2\overline{xy}m + 2\overline{x}mb - 2\overline{y}b + b^2
\end{aligned}
$$

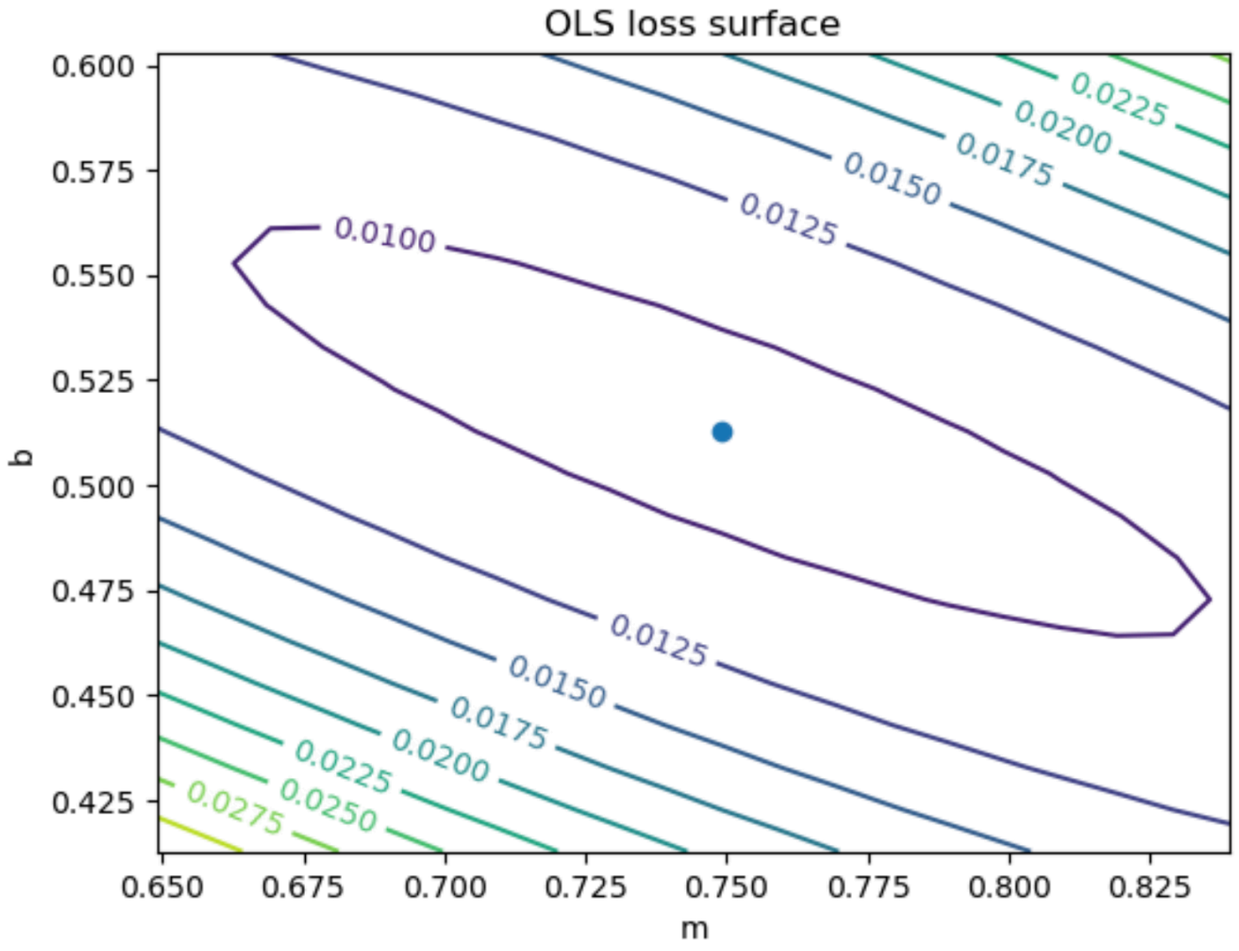


- $\boxed{L(m,b) = \overline{y^2} + \overline{x^2}m^2 - 2\overline{xy}m + 2\overline{x}mb - 2\overline{y}b + b^2}$ (where bar denotes the average)

Loss surface: Ordinary Least squares (OLS)

$$L(m,b) = \overline{y^2} + \overline{x^2}m^2 - 2\overline{xy}m + 2\overline{x}mb - 2\overline{y}b + b^2$$





The “canonical” deep learning training workflow

- **Data** → collect, clean, **supervise**, split (train/val/test), augment
  - Garbage in, garbage out: this is where most projects live or die
- **Model Architecture** → choose a model that well suited to the data
  - MLP (simple data), CNN (grids/Images), RNN/Transformer (sequences), GNN (graphs)
- **Objective** (prediction quality scoring mechanism) → define a loss $\mathcal{L}(\theta)$ that encodes "good"
  - Regression: Mean square error MSE
  - Binary classification: Binary Cross-entropy
  - Multi-class classification: Categorical Cross-entropy
- **Optimization** (learning) → back-prop + SGD/Adam, LR schedule, regularization
  - We are just rolling downhill on a very high-dimensional surface
- **Evaluation** → validation metrics, error analysis, ablations
  - Tune on val, touch test *once*
- **Deployment** → export, serve, monitor drift, iterate

Aside: "Canonical" means conforming to a standard, authoritative, or generally accepted form the version of something that serves as the reference point against which variants are judged (e.g., the canonical form of a matrix, or the canonical version of a text).

Ordinary least squares: closed form

Stack the n observations into a **design matrix** with a column of ones for the intercept:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \in \mathbb{R}^{n \times 2}, \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \qquad \mathbf{y} \in \mathbb{R}^n$$

so that the model is simply $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$.

Loss (squared error, a quadratic form in $\mathbf{w}$):

$$\mathcal{L}(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

Gradient $\rightarrow$ set to zero $\rightarrow$ **normal equations**:

$$\nabla_{\mathbf{w}} \mathcal{L} = -2\,\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) = \mathbf{0} \quad \Longrightarrow \quad \mathbf{X}^\top \mathbf{X}\,\mathbf{w} = \mathbf{X}^\top \mathbf{y}$$

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

Analytic solution

- Using the loss surface above, we can analytically obtain the solution to the OLS problem
- OLS is simply enough that we can apply analytic optimization techniques, similar to the previous slide
- The global minima of $L(m, b) = \overline{y^2} + \overline{x^2}m^2 - 2\overline{xy}m + 2\overline{x}mb - 2\overline{y}b + b^2$ satisfy the condition

$$\nabla L(\mathbf{w}) = \left(\frac{\partial L}{\partial m}, \frac{\partial L}{\partial b} \right) = (0, 0)$$

- gonna do the derivation here because it is part of your homework but it is quite straightforward
- This forms a system of two equations which can be solved algebraically
- Which yields the closed form analytical solution for the best fit parameters

$$m_{opt} = \frac{Cov(x_i, y_i)}{var(x_i)} \qquad b_{opt} = \overline{y} - m_{opt}\overline{x} = \overline{y} - \overline{x}\frac{Cov(x_i, y_i)}{var(x_i)}$$

Looking ahead:

$$y_{pred} = M(x|\mathbf{w}) = M(x|m,b) = mx + b$$

new value = old value – step

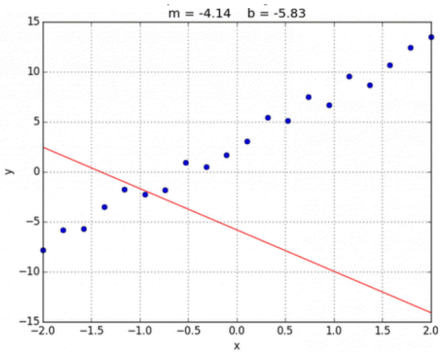
$$\mathbf{w}_{\text{next}} \leftarrow \mathbf{w}_{\text{current}} - \eta \nabla_{\mathbf{w}} L$$

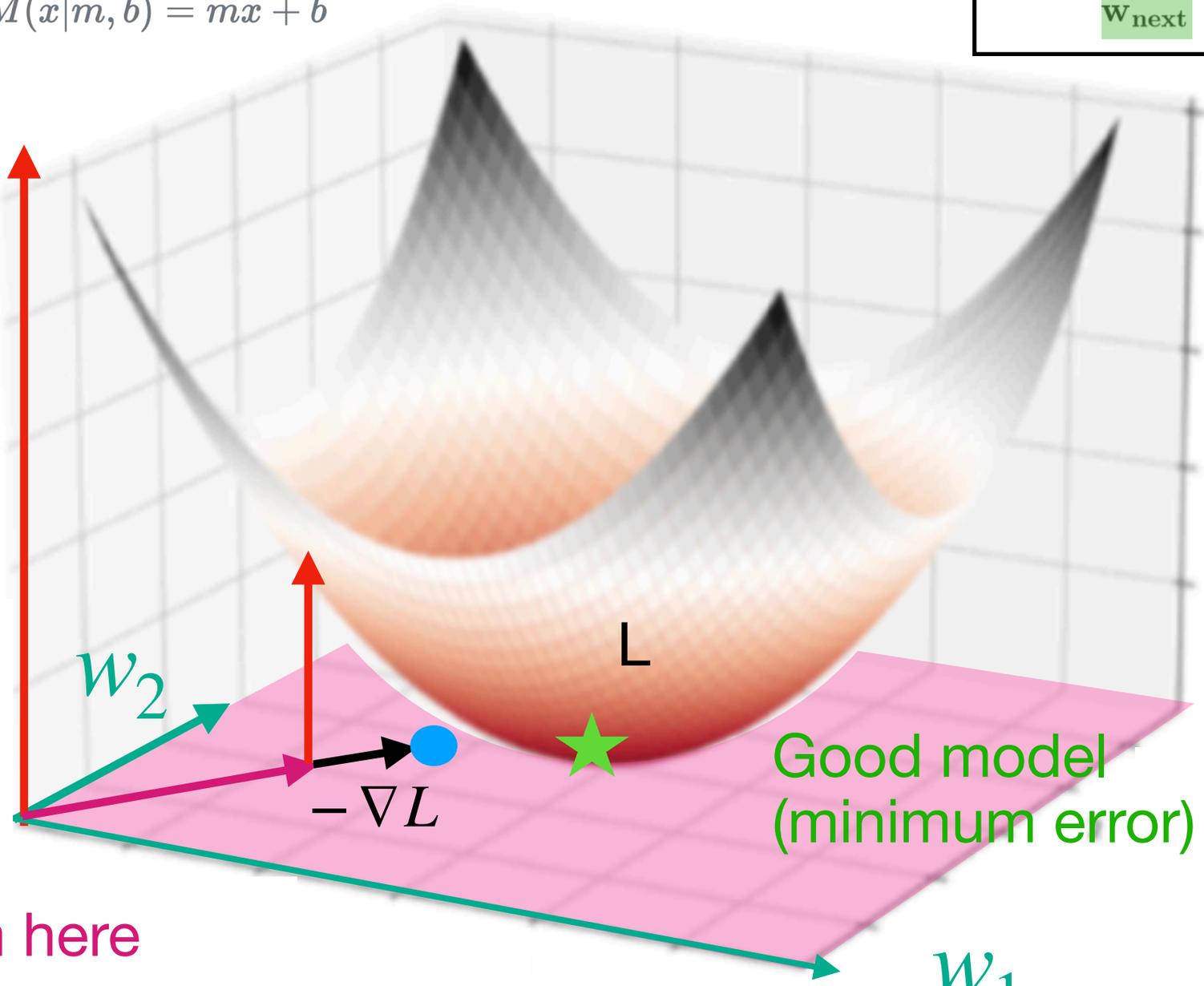


Model error

$l = l(w_1, w_2)$

(Loss)



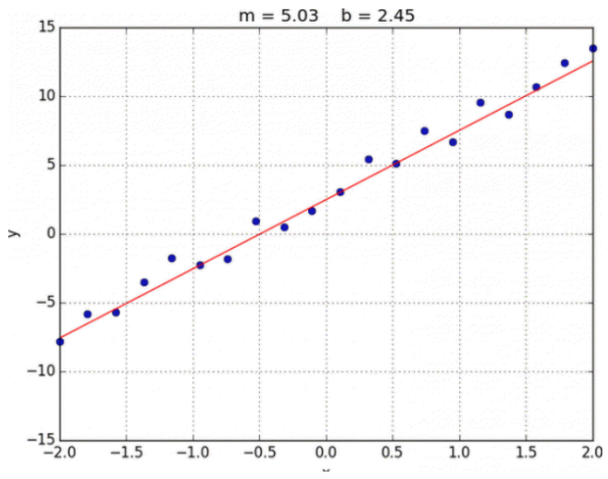


Imagine I'm here

$\mathbf{w} = (w_1, w_2) = (-4.1, -5.8)$

Model parameter space

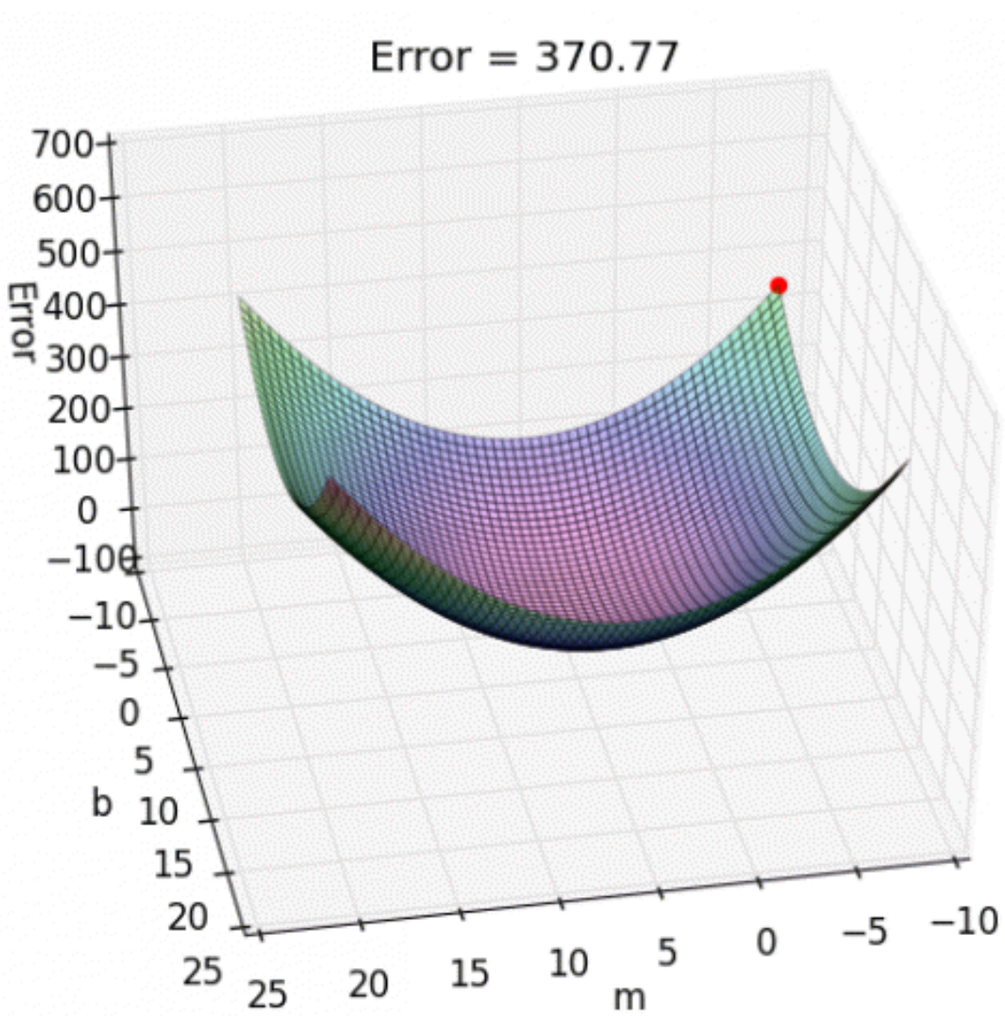
$\mathbf{w}^* = (5\ , 2.5)$

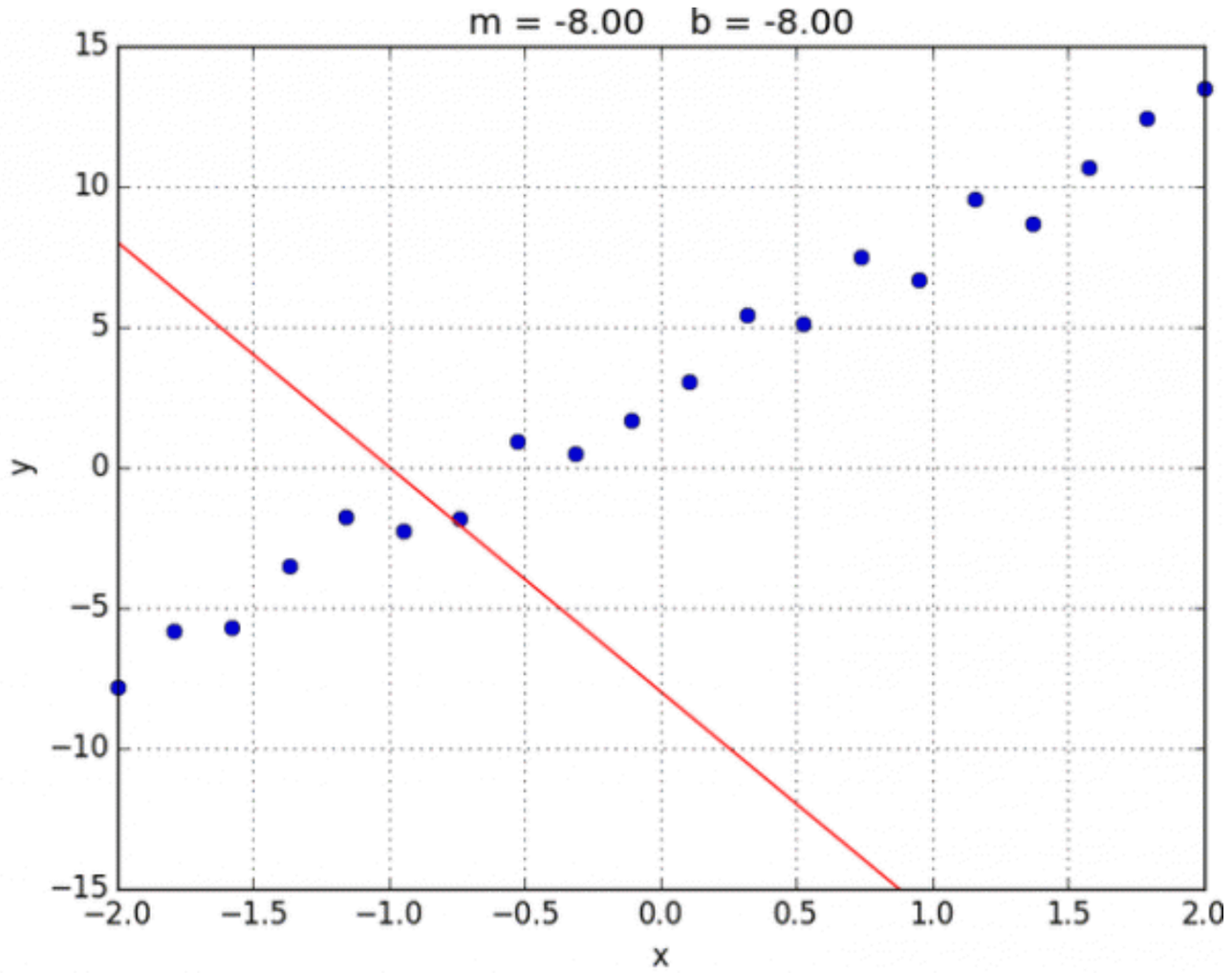


Computer calculates gradient vector for us via back-propagation (don't ask)

$$-\nabla L(\mathbf{w}) = -(-0.5, -0.5) = (0.5, 0.5)$$

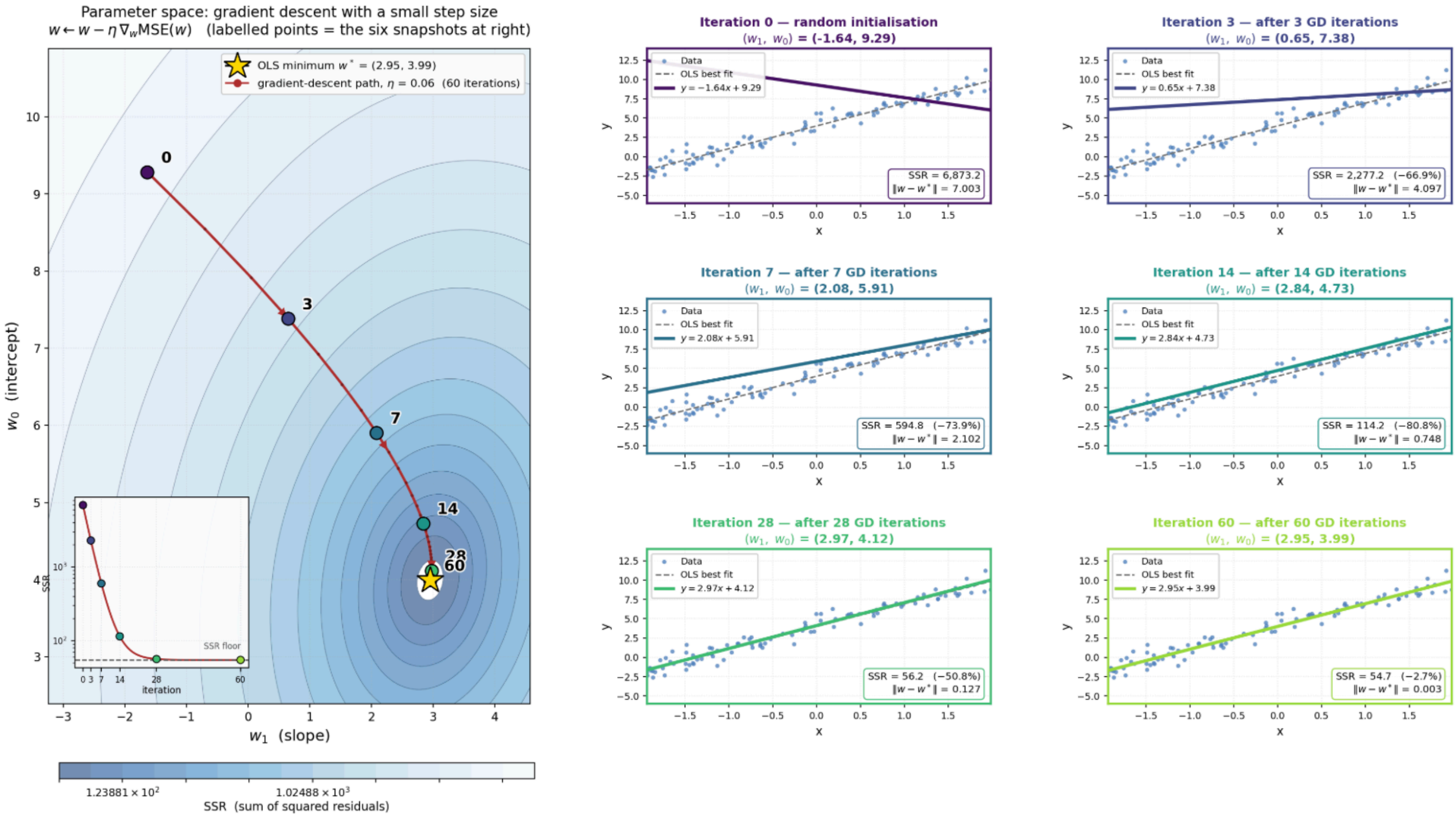
Gradient descent applied to linear regressor:



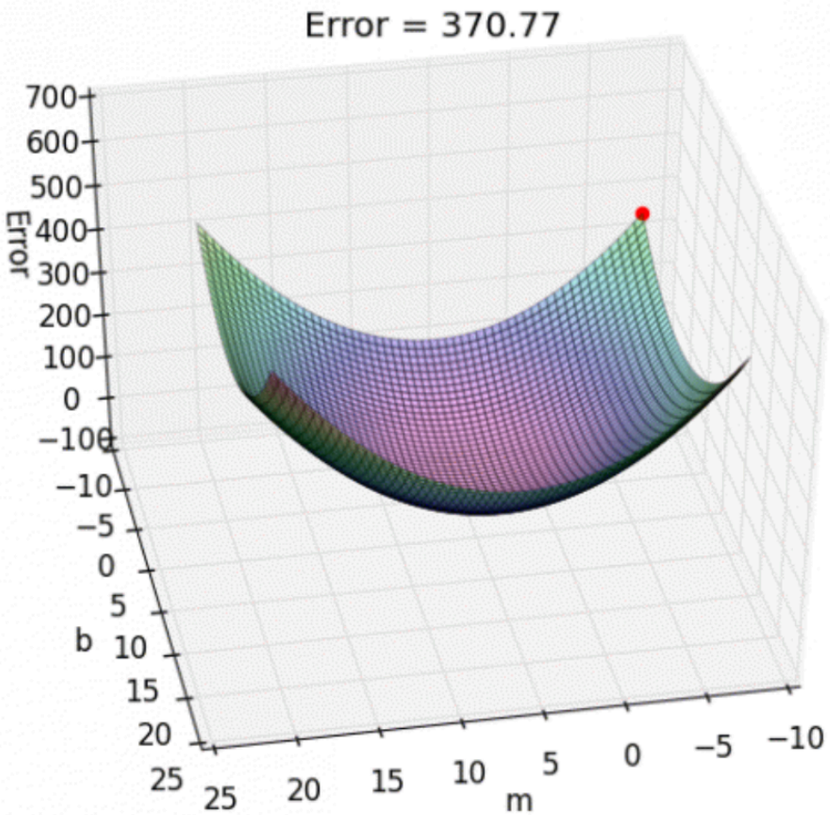


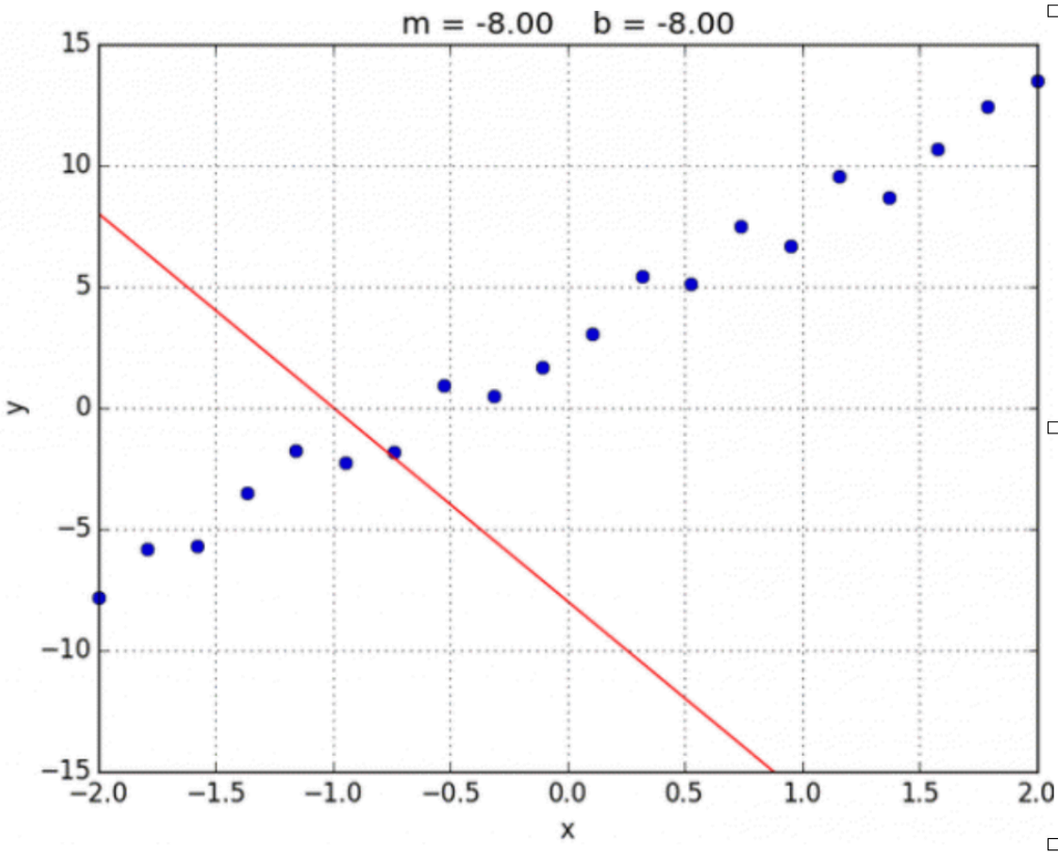
Parameter space makes a search space that allows us to systematically search for better and better models (better parameterizations)

Gradient descent applied to linear regressor:

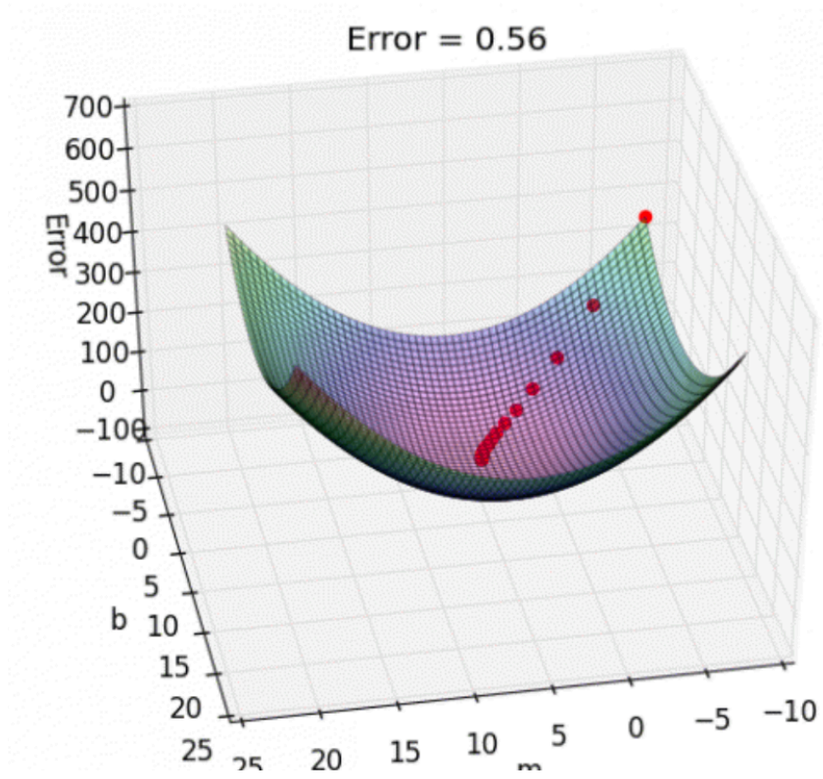


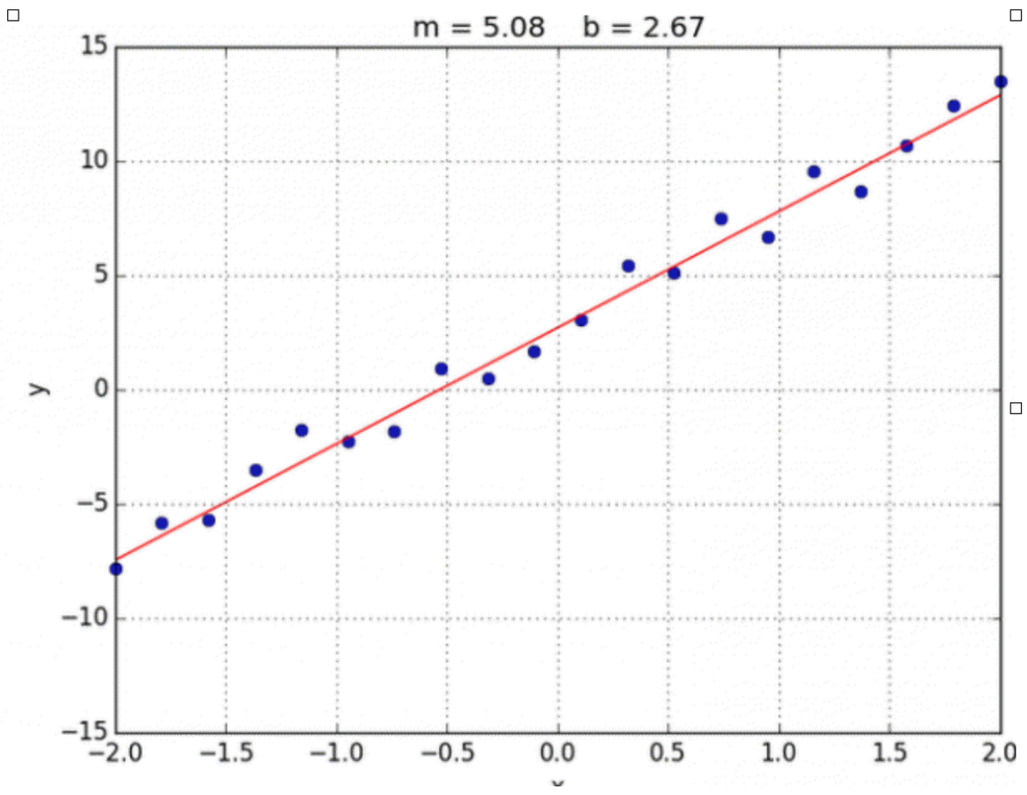
Random initial parameter guess:



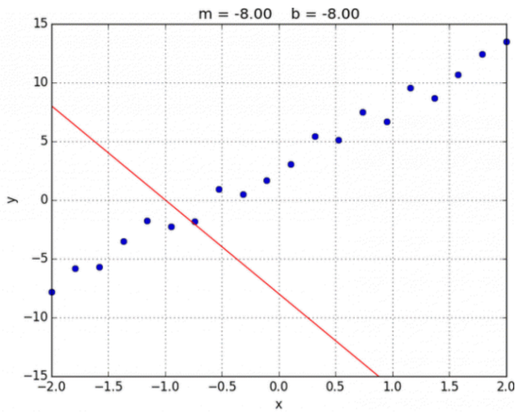


Final solution after optimization





Training loop for a model



$\mathbf{w}_{t=0} \rightarrow$ e.g. np.rand() $w = 0.1, b = 0.4$

Single training run:

Random parameters guess

Input X
(Training data features)

Parametric Model

$x \xrightarrow{w} (b) \longrightarrow \hat{y} = m(x \mid \mathbf{w}) = w_1 x + w_0 = mx + b$ (2 parameters)

model prediction at iteration t for i^{th} training sample
for parameterization $\mathbf{w}_t$ at t^{th} iteration of GD

prediction $= \hat{y}_t^i = m(\mathbf{x}_i \mid \mathbf{w}_t)$ evaluate

(Repeat)

True targets Y
(Training data targets)

Predictions $\hat{y}_t$

Exit criteria
(convergence or early stopping)

Trained model
$\mathbf{w}^*$ = optimal model parameters

Sum of square error (SSE)

$$SSE = \sum_i^N \epsilon_i^2 = \sum_i^N (y_i - \hat{y}_i)^2$$

Loss function
Compare targets and predictions

$\mathbf{w}_{t+1}$

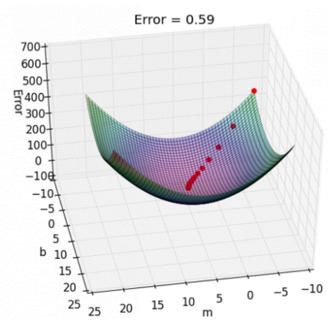
Updated Parameters

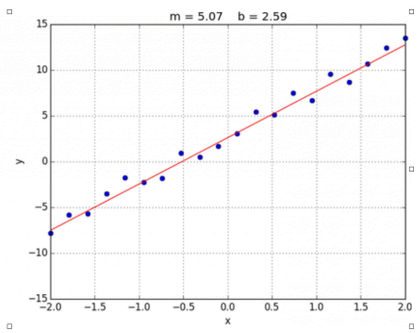
(*sum or mean*) reduction

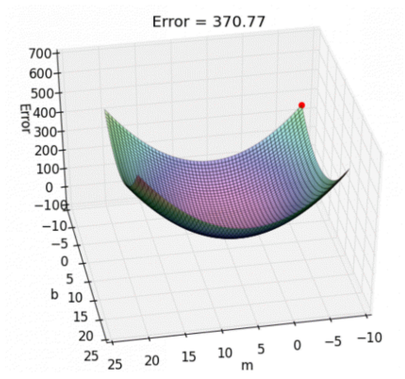
Training loop
(iterations of gradient descent)

Monitor

Loss $L_t = L(\mathbf{w}_t)$







Loss score
(AKA loss surface)

Optimizer

The gradient vector $\nabla_w L\big|_{\mathbf{w}_t}$,at the t^{th} time-step, is computed numerically via back-propagation (i.e. automatic differentiation) using specialized libraries such as Tensor-flow or PyTorch. Alternatively the gradient can also be approximated via finite difference in limited simplistic cases

Gradient descent update rule

Vector equation (updates all fitting parameters)

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \nabla_w L \Big|_{\mathbf{w}_t}$$

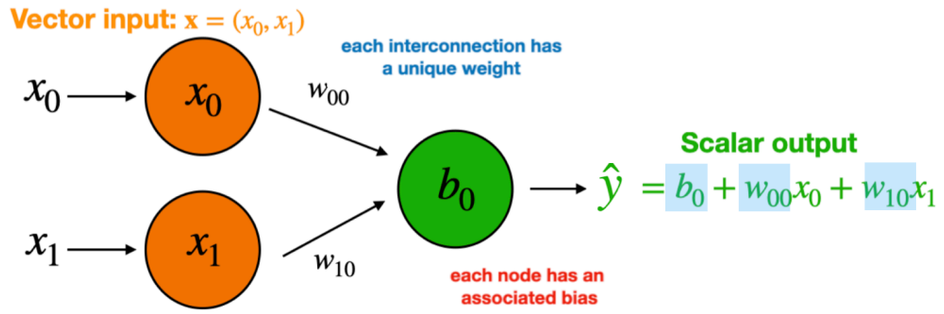
λ = learning rate

(For the j^{th} fitting parameter) $\longrightarrow$

$$w_{t+1}^j = w_t^j - \lambda \frac{\partial L}{\partial w^j}\Big|_{w_t^j}$$

$\longrightarrow$ Read vertical line as "Evaluated at"

Dimensionality of the optimization space = number of fitting parameters

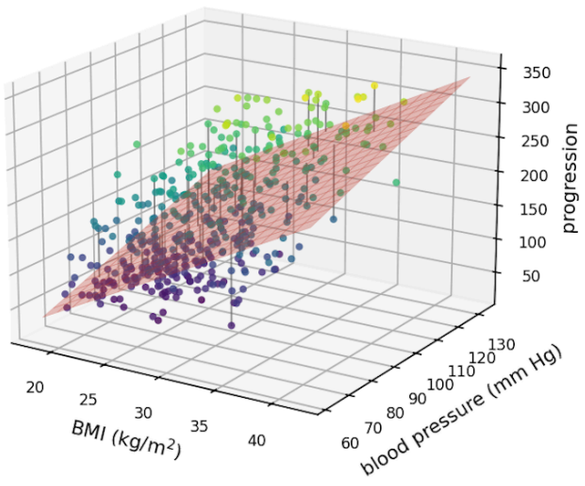


- **Number of fitting parameters: 3**

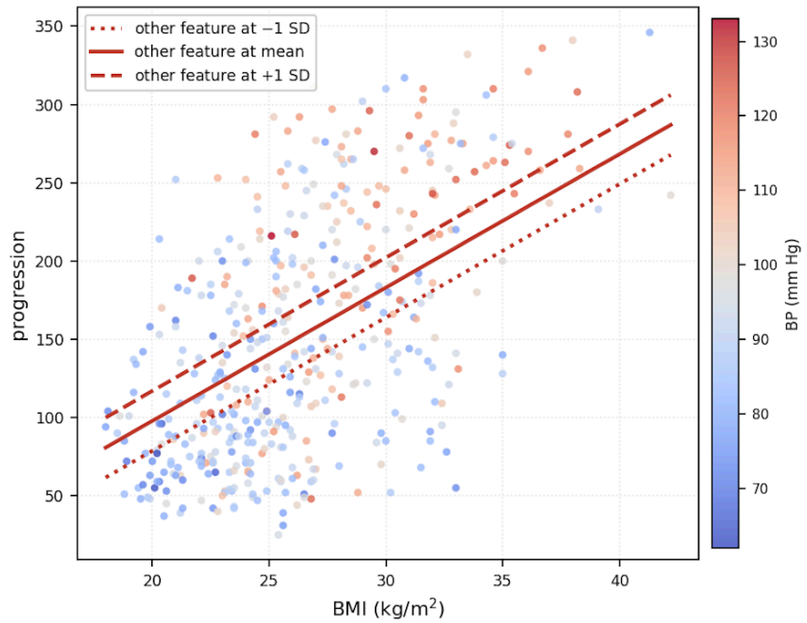
A three-parameter loss space: progression $\approx w_0 + w_1 \cdot \mathrm{BMI}_z + w_2 \cdot \mathrm{BP}_z$ (diabetes study, n = 442)

SSR(w^*) = 1,583,105 | R^2 = 0.396 | w^* = (152.1, 37.6, 19.1) | standard errors = (2.9, 3.1, 3.1) | corr(w_1, w_2) = -0.40

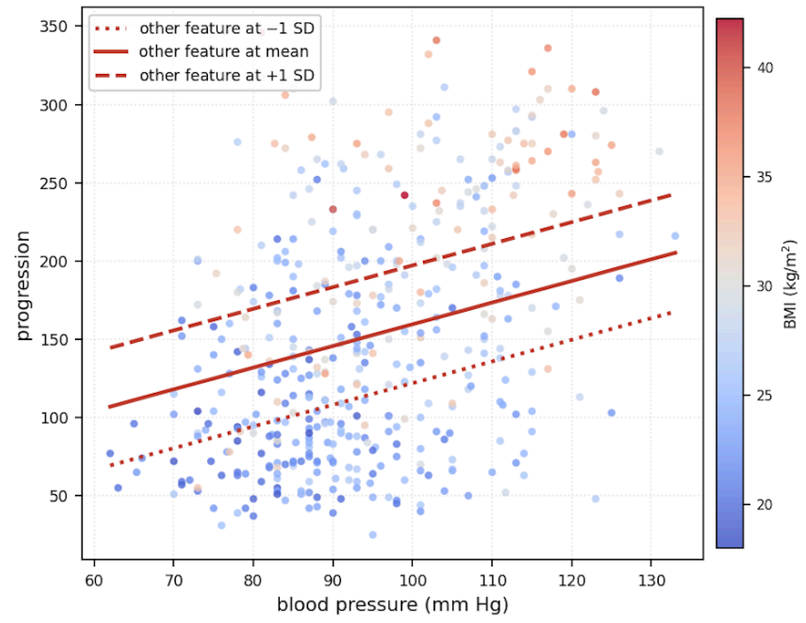
(a) Data cloud + fitted plane
n = 442; grey sticks = residuals, SSR sums their squares



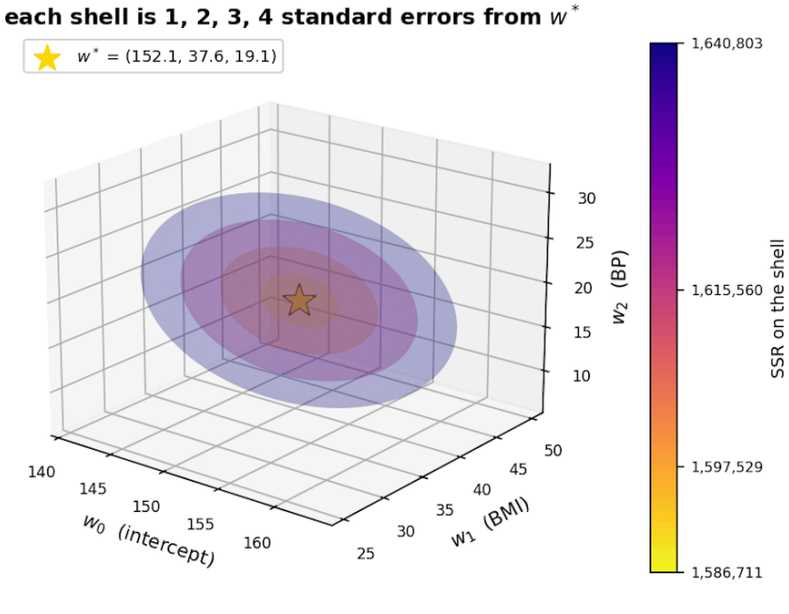
(b) The same plane, sliced along BMI
colour = the other feature



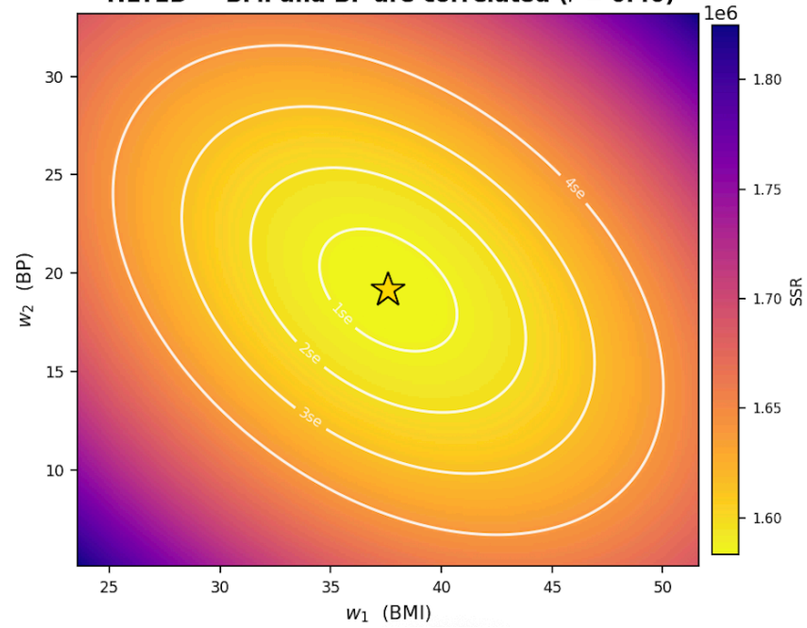
(c) The same plane, sliced along blood pressure
colour = the other feature



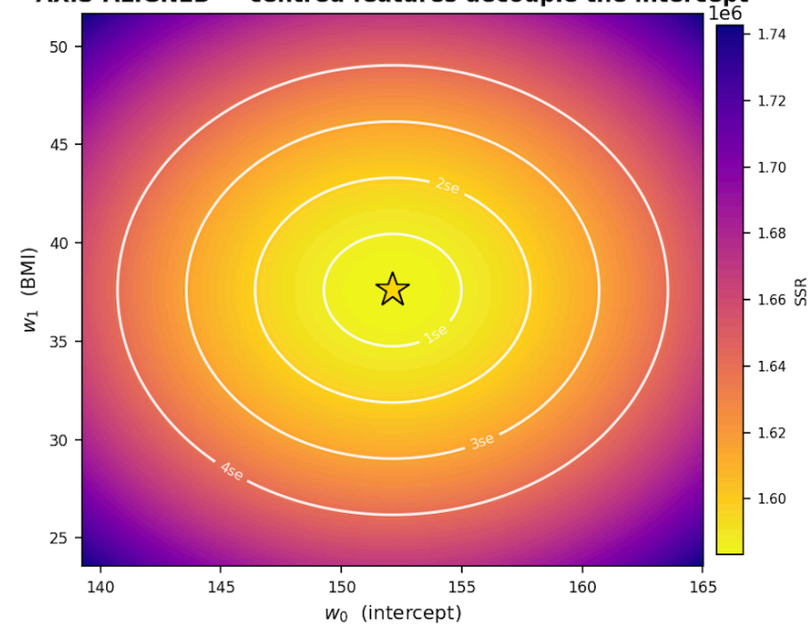
(d) Loss space is 3-D: nested SSR isosurfaces
each shell is 1, 2, 3, 4 standard errors from w^*



(e) Slice at $w_0 = w_0^*$: the (w_1, w_2) plane
TILTED — BMI and BP are correlated (r = 0.40)



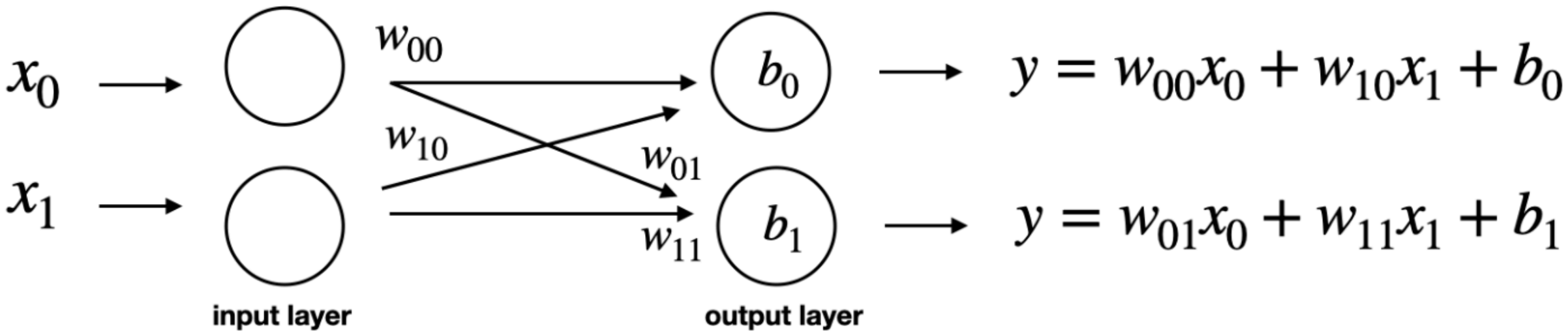
(f) Slice at $w_2 = w_2^*$: the (w_0, w_1) plane
AXIS-ALIGNED — centred features decouple the intercept



Linear vector regression:

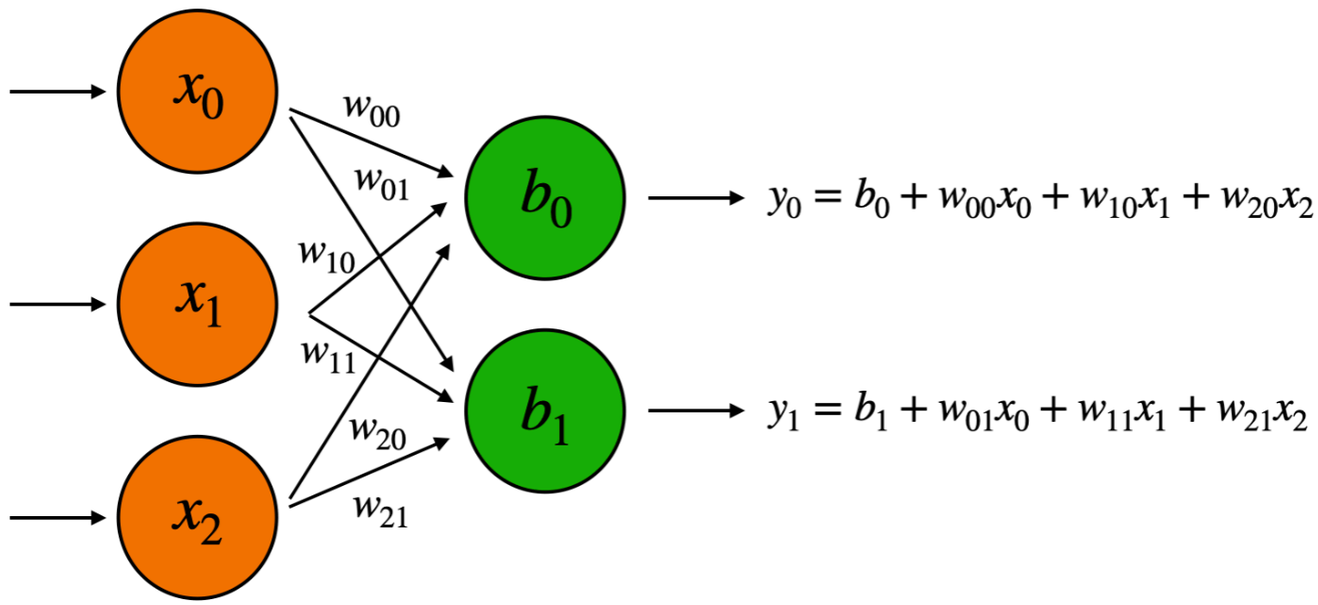
- **Problem Type:** Vector-regression
- **Mapping:** $\overset{\text{(input layer)}}{\mathbb{R}^2} \rightarrow \overset{\text{(output layer)}}{\mathbb{R}^2}$
- **Hidden layers**: None
- **Hidden-layer activation**: N/A
- **Output-layer activation**: linear
- **Number of fitting parameters**: 6

Computational graph



Scaling up: We can easily extend the problem by considering more dimensions.

	input space: $\mathbb{R}^3$			output space: $\mathbb{R}^2$	
i	**sqft** $x_1 \cdot 1000\ ft^2$	**age** $x_2 \cdot years$	**beds** $x_3 \cdot count$	**price** $y_1 \cdot \$1000s$	**rent** $y_2 \cdot \$/month$
0	1.45	12	3	365.4	2150
1	0.98	45	2	219.8	1480
2	2.10	3	4	498.6	2890
3	1.32	28	3	308.1	1975
4	2.85	8	5	625.7	3420
5	1.70	35	3	366.2	2240



For a single input feature vector $\mathbf{x}^T = (x_0, x_1, x_2)$ the model is

$$\mathbf{y}_{pred} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix}_{pred} = \mathbf{m}(\mathbf{x}|\mathbf{w}, \mathbf{b}) = \mathbf{b} + \mathbf{w}\mathbf{x} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

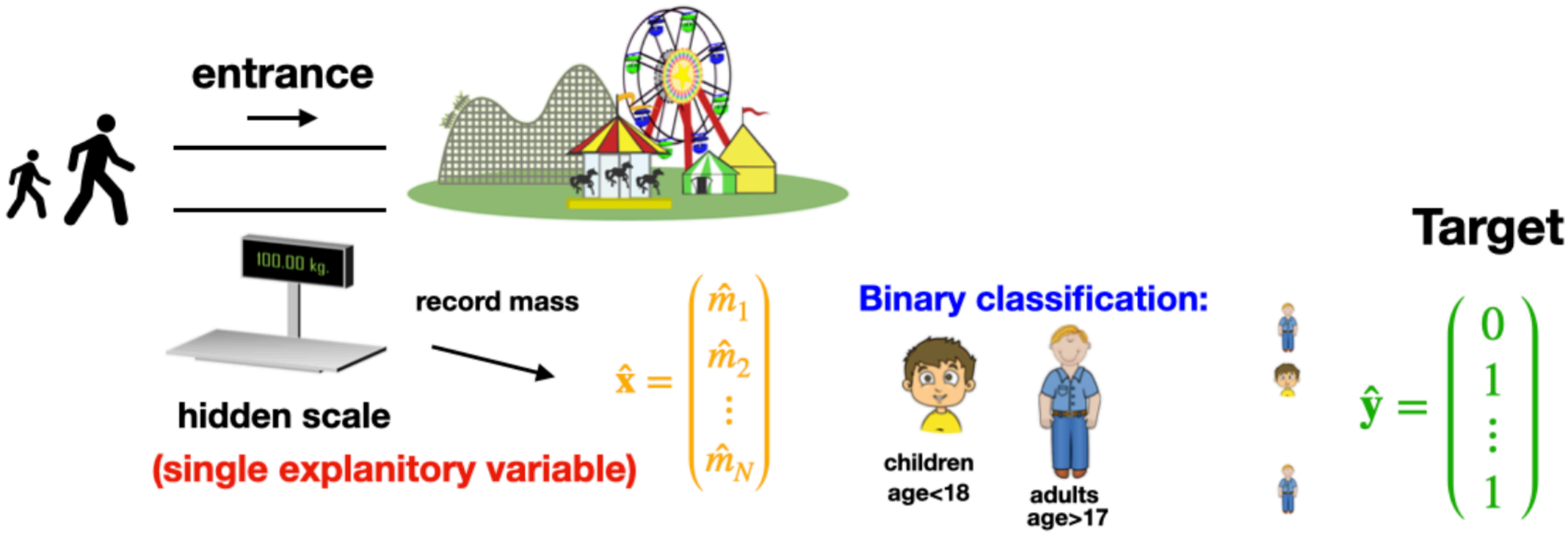
In which case the model can be rewritten in a mathematically identical form

$$\mathbf{y}_{pred} = \mathbf{m}(\tilde{\mathbf{x}}|\tilde{\mathbf{w}}) = \tilde{\mathbf{w}}\tilde{\mathbf{x}} = \begin{pmatrix} b_0 & w_{00} & w_{01} & w_{02} \\ b_1 & w_{10} & w_{11} & w_{12} \end{pmatrix} \begin{pmatrix} 1 \\ x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

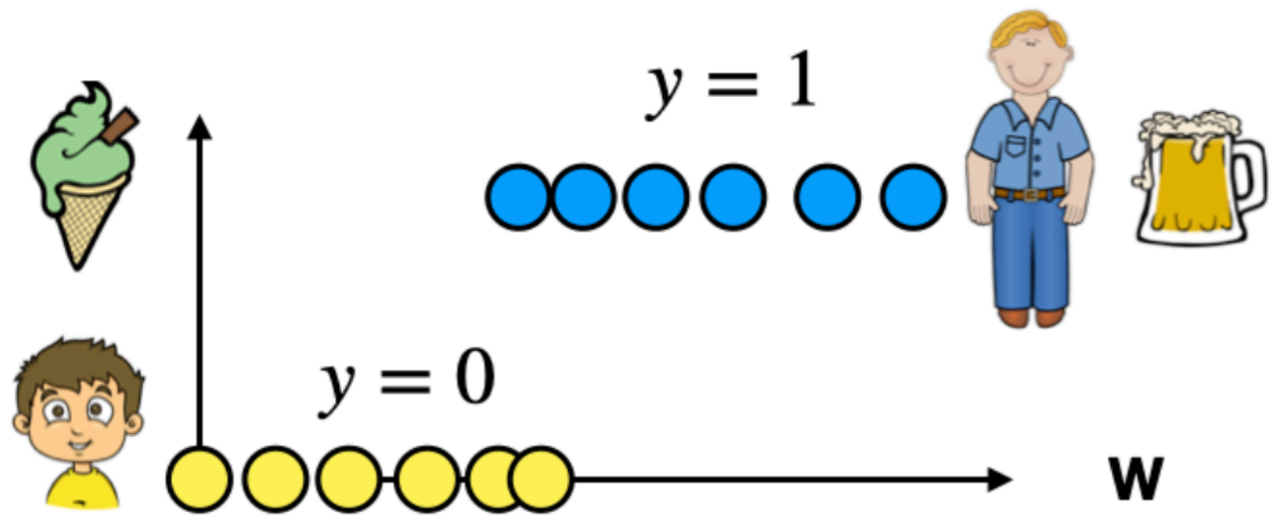
Generalizing the model as a classifier (Logistic regression)

Example: Dataset

- Consider the following scenario at an amusement park gathering data.
- **Goal:** Classify qualitative age y (0=child vs 1=adult) based on weight x descriptor
    - This data could be used to help predict inventory prediction (ice-cream vs beer).



Make prediction: given an arbitrary mass, is it adult or child?



Logistic function

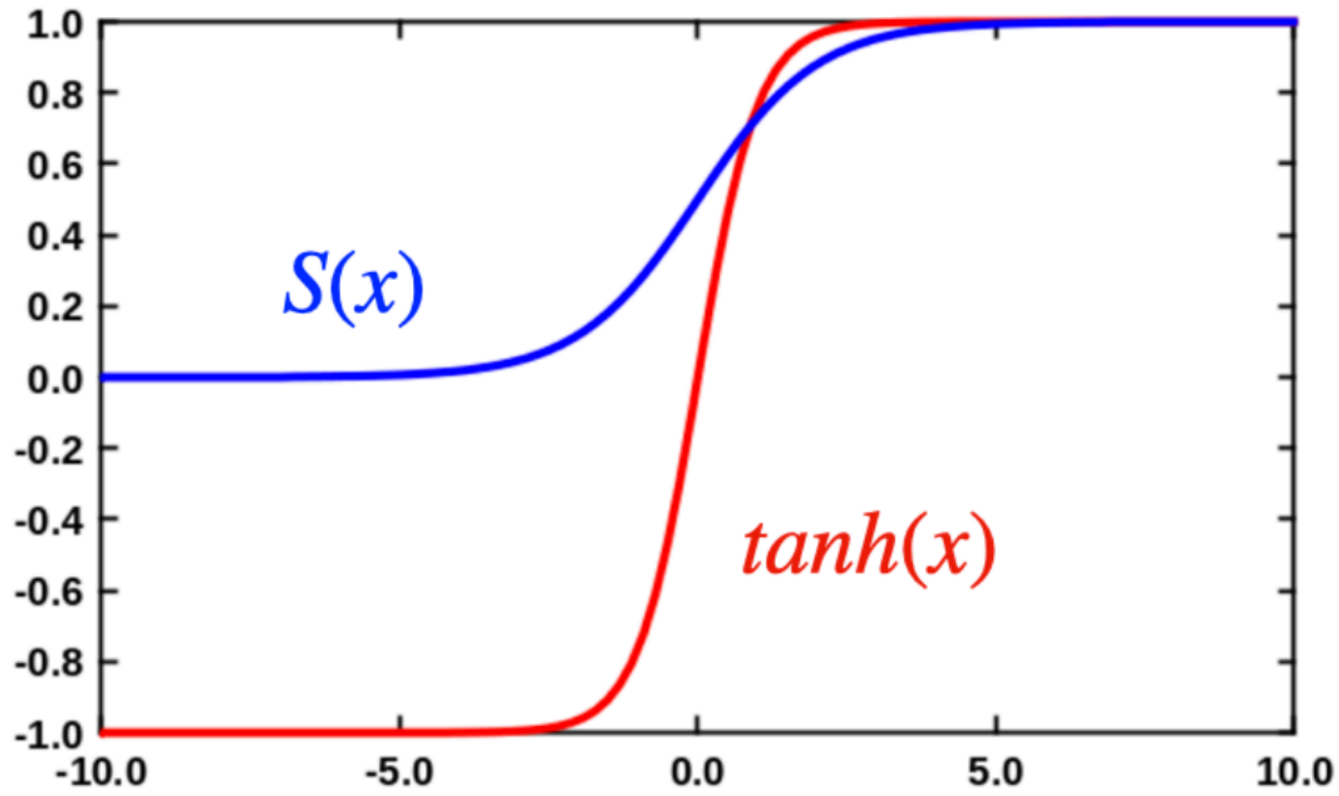
- A much better option, is to model the class-probability of y (0=child vs 1=adult) as a non-linear logistic function

Sigmoid (aka. logistic) function

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} = 1 - S(-x)$$

Hyperbolic Tangent function

$$\tanh(x) = 2\left(S(2x) - 0.5\right)$$



Parameterized

$$y = S(x\,|\,\mathbf{p}) = \frac{1}{1+e^{-(wx+b)}}$$

$$\mathbf{p} = (w, b)$$

Logistic regression

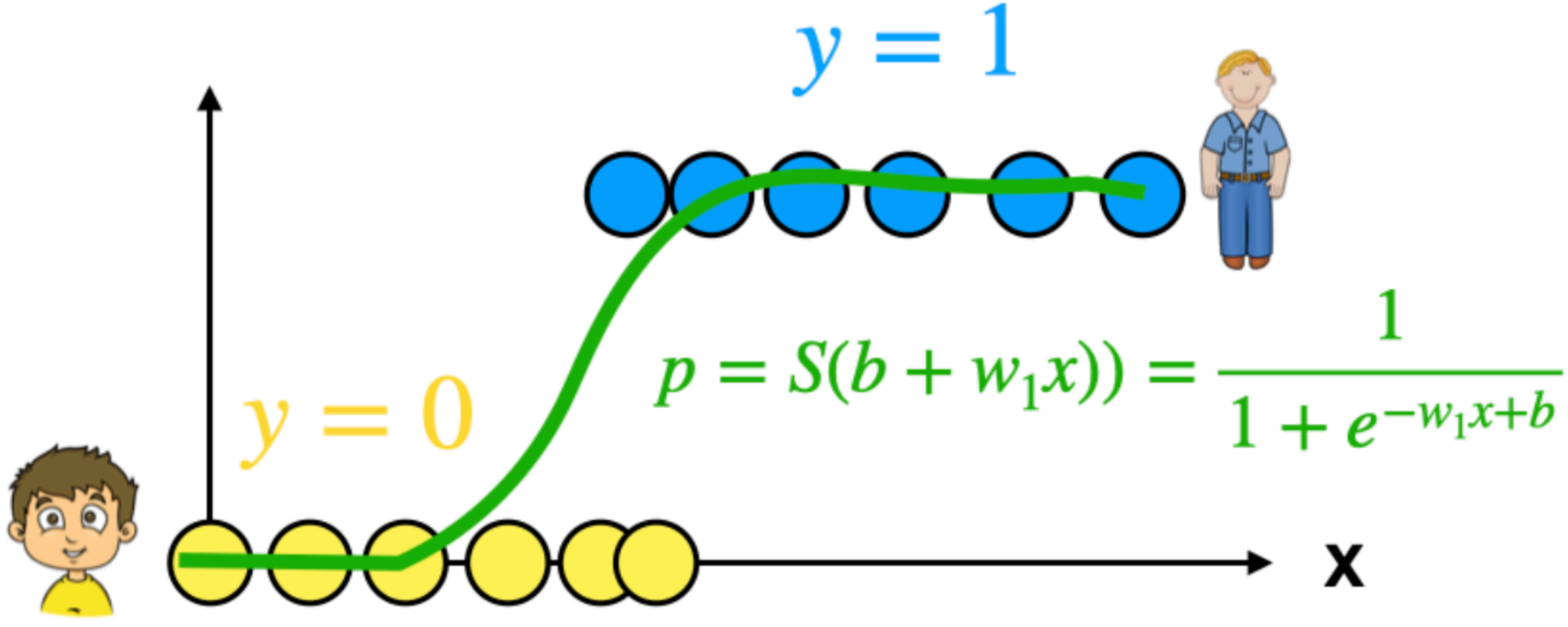
- It is common to assume a `linear logit function` in binary logistic regression .
- In this case the class-probability obeys a sigmoid (logistic function)

$$\ell = \log_e \frac{p}{1-p} = b + w_1 x$$ log-odds (also called logit)

$$\text{odds} \quad \frac{p}{1-p} = e^{b+w_1 x} \longrightarrow$$

probablility that outcome=true (Y=1)

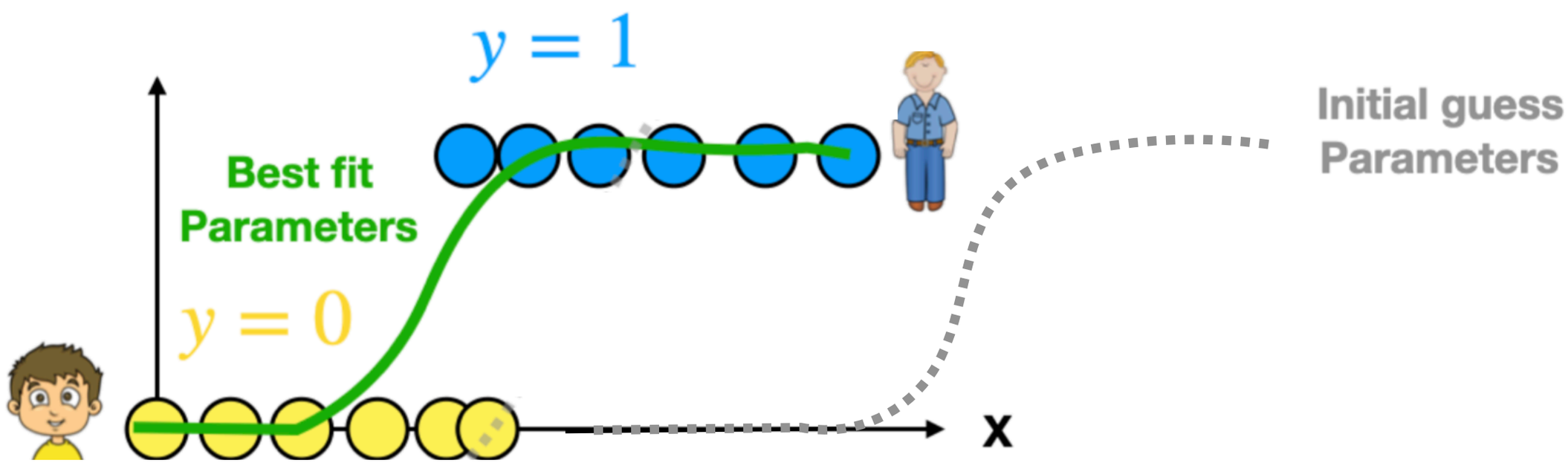
$$p = S(b + w_1 x)) = \frac{1}{1 + e^{-w_1 x + b}}$$



- **Important**: Even though probability $p(x)$ is non-linear, we still call this a `linear logistic regression model`, because the is assumed to be linear $logit = w_1 x + b$

Model fitting-1

- In theory, we could fit this using numerical optimization, similar to in module-1.
- The model is simple enough that we can do it analytically (computationally faster)



(weight and bias)

Model: $M(x \mid \mathbf{w}) = \sigma(w_1 x + w_0)$ $\mathbf{w} = (w_1,\ w_0)$ $\sigma(z) = \frac{1}{1+e^{-z}},$ $z = w_1 x + w_0$

Minimize binary cross-entropy = maximize log-likelihood

Loss: ~~$\sum_{i=1}^{N} \epsilon_i^2$~~

$$L(\mathbf{w}) = -N^{-1}\sum_{i=1}^{N}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

$$\hat{y}_i = M(x_i \mid \mathbf{w}) = \Pr(y_i = 1 \mid x_i; \mathbf{w})$$

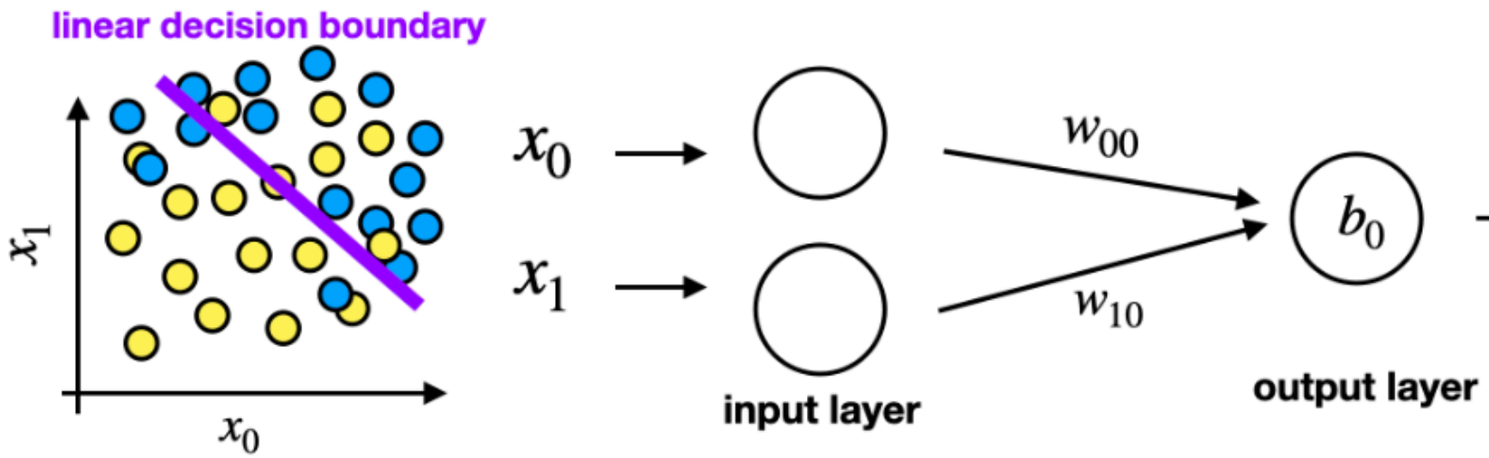
squared error (regression)

Numerical Optimization: $\nabla L(\mathbf{w}) = 0 = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_0}\right)$ $\mathbf{w}_{n+1} = \mathbf{w}_n - \eta \nabla L(\mathbf{w}_n)$

logistic regression:

- **Problem Type:** Binary-classification
- **Mapping:** $\overset{\text{(input layer)}}{\mathbb{R}^2} \rightarrow \overset{\text{(output layer)}}{\mathbb{R}^1}$
- **Hidden layers**: None
- **Hidden-layer activation**: N/A
- **Output-layer activation**: Sigmoidal (logistic)
- **Number of fitting parameters**: 3

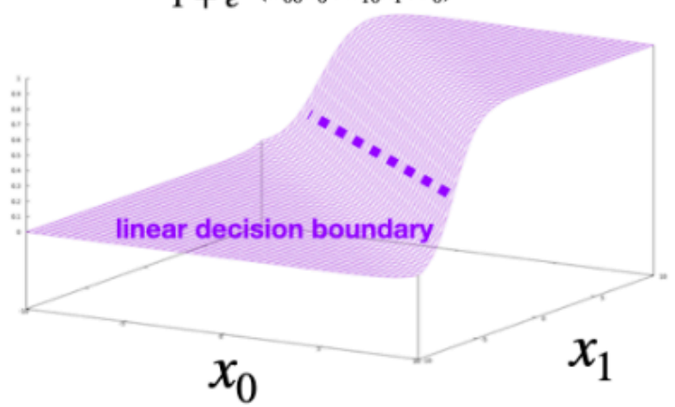
Computational graph



$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

$$\rightarrow S(w_{00}x_0 + w_{10}x_1 + b_0)$$

probablitly: $p = \frac{1}{1+e^{-(w_{00}x_0+w_{10}x_1+b_0)}} = S(w_{00}x_0 + w_{10}x_1 + b_0)$



The multi-layer perceptron model

Transformations

Linear transformations are constructed from basic geometric actions like rotation, scaling, contraction, skewing, and projection.

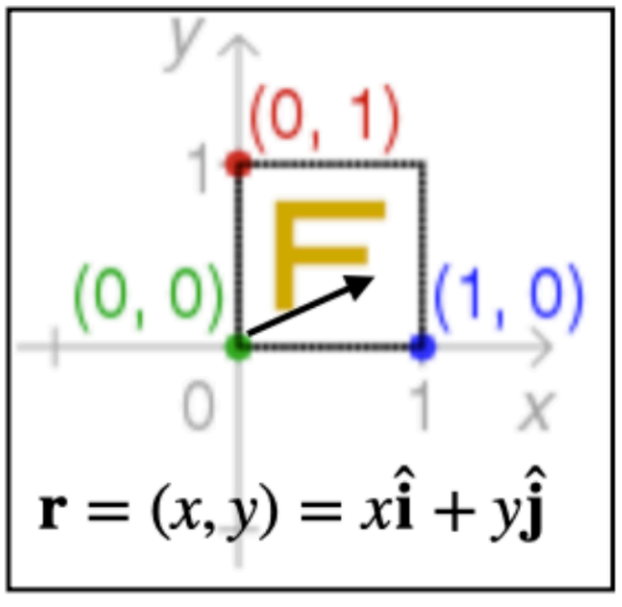
$$\mathbf{x} \rightarrow \mathbf{x}' = \mathbf{A}\mathbf{x}$$

Transformations

matrices

$$\mathbf{x}' = \mathbf{A}\mathbf{x} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix}$$

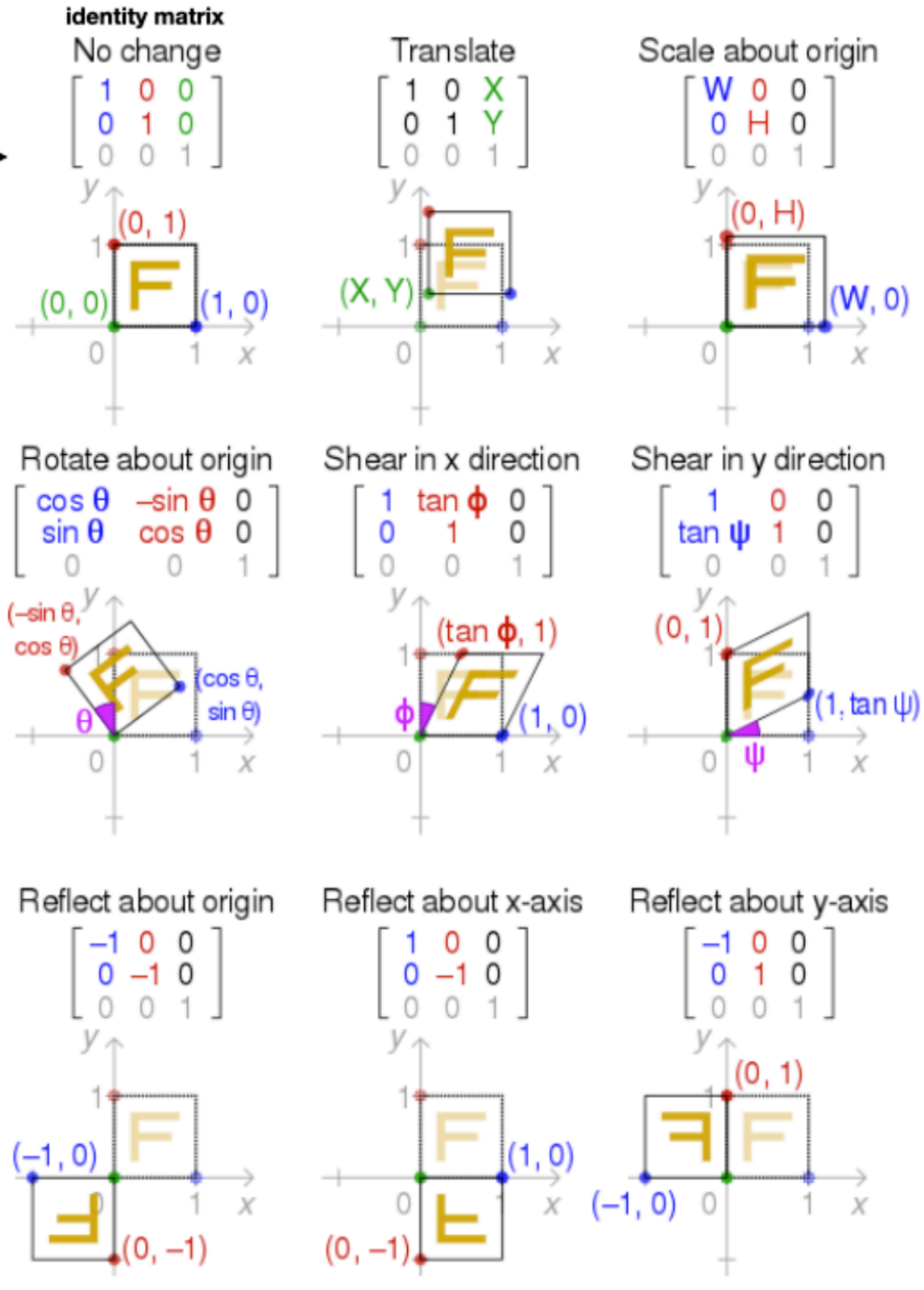
Original image



Inverse tranformation:

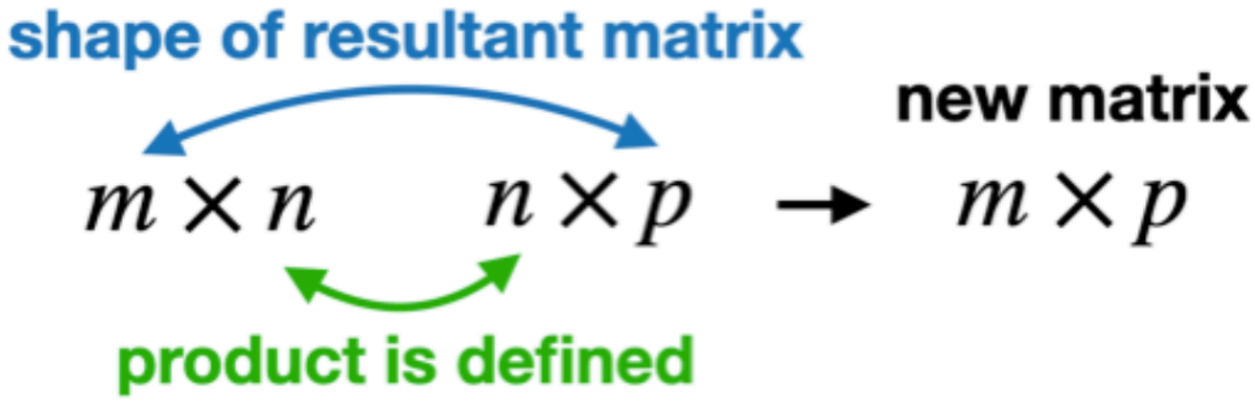
$$\mathbf{x}' \rightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{x}' \qquad \mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{1}$$

inverse of A



Matrix multiplication

- Matrix multiplication is basically repeated dot products: The new entry for row-i column-j is A_row_i DOT B_column_j
- Two matrices can be multiplied if they obey the following rule.



- **Example-1:** (matrix-vector product).

$$\mathbf{Ax} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{pmatrix}$$

$$2 \times 2 \quad 2 \times 1 \qquad 2 \times 1$$

$$\mathbf{x}^T\mathbf{A} = (x_1\ x_2) \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = (x_1a_{11} + x_2a_{21} \quad x_1a_{12} + x_2a_{22})$$

$$1 \times 2 \quad 2 \times 2 \qquad 1 \times 2$$

Example

- **Example-2:** (matrix-matrix product).

$$2\times 3 \quad \mathbf{A} \quad \mathbf{B} \quad 3\times 2$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

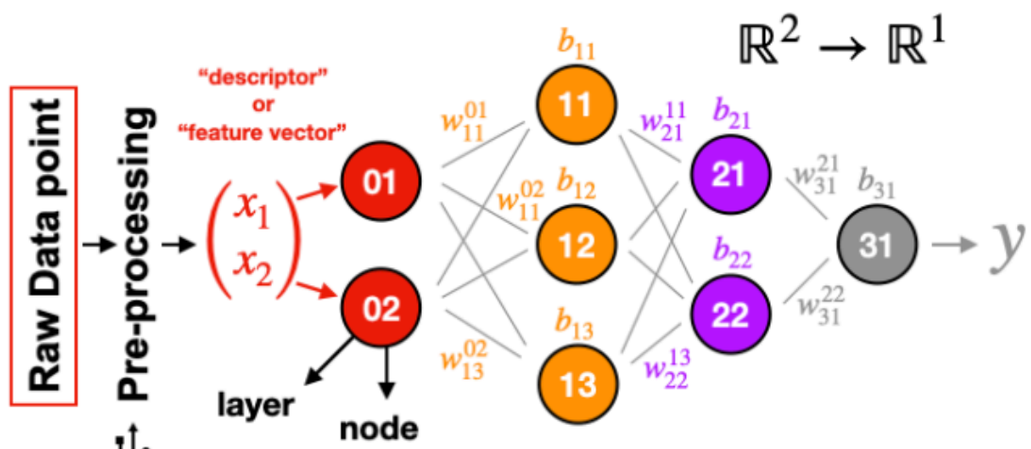
$$= \begin{bmatrix} 1x10 + 2x20 + 3x30 & 1x11 + 2x21 + 3x31 \\ 4x10 + 5x20 + 6x30 & 4x11 + 5x21 + 6x31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

$$2\times 2$$

ANN model: introductory example

- **Problem Type:** Scalar regression (un-constrained output)
- **Mapping:** $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ (input layer) (output layer)
- **Hidden layers:** $N_{hidden} = 2 \rightarrow [3,2]$
- **Hidden-layer activation**: Sigmoid
- **Output-layer activation**: Linear
- **Number of fitting parameters**: 20



NN architecture: 2-3-2-1

Number of fitting parameters:

$$N_{fit} = (3 \times 2 + 3) + (2 \times 3 + 2) + (2 \times 1 + 1) = 20$$

$$\begin{pmatrix} w_{11}^{01} \\ w_{12}^{01} \\ w_{13}^{01} \\ w_{11}^{02} \\ w_{12}^{02} \\ w_{13}^{02} \\ b_{11} \\ b_{12} \\ b_{13} \\ w_{21}^{11} \\ w_{22}^{11} \\ w_{21}^{12} \\ w_{22}^{12} \\ w_{21}^{13} \\ w_{22}^{13} \\ b_{21} \\ b_{22} \\ w_{31}^{21} \\ w_{31}^{22} \\ b_{33} \end{pmatrix}$$

Input Layer $\rightarrow A(x) \rightarrow A(x) \rightarrow$ **Output layer**

Activation function:

$$A(x) = \frac{1}{1 + e^{-x}}$$

apply activation function (non-linearity) Apply component wise

Hidden layer-1 output

$$\begin{pmatrix} O_{11} \\ O_{12} \\ O_{13} \end{pmatrix} = A\left(\begin{pmatrix} w_{11}^{01} & w_{11}^{02} \\ w_{12}^{01} & w_{12}^{02} \\ w_{13}^{01} & w_{13}^{02} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_{11} \\ b_{12} \\ b_{13} \end{pmatrix} \right)$$

Hidden layer-2 output

$$\begin{pmatrix} O_{21} \\ O_{22} \end{pmatrix} = A\left(\begin{pmatrix} w_{21}^{11} & w_{21}^{12} & w_{21}^{13} \\ w_{22}^{11} & w_{22}^{12} & w_{22}^{13} \end{pmatrix} \begin{pmatrix} O_{11} \\ O_{12} \\ O_{13} \end{pmatrix} + \begin{pmatrix} b_{21} \\ b_{22} \end{pmatrix} \right)$$

Neural network output

$$y = (O_{31}) = \begin{pmatrix} w_{31}^{21} & w_{31}^{22} \end{pmatrix} \begin{pmatrix} O_{21} \\ O_{22} \end{pmatrix} + (b_{31})$$

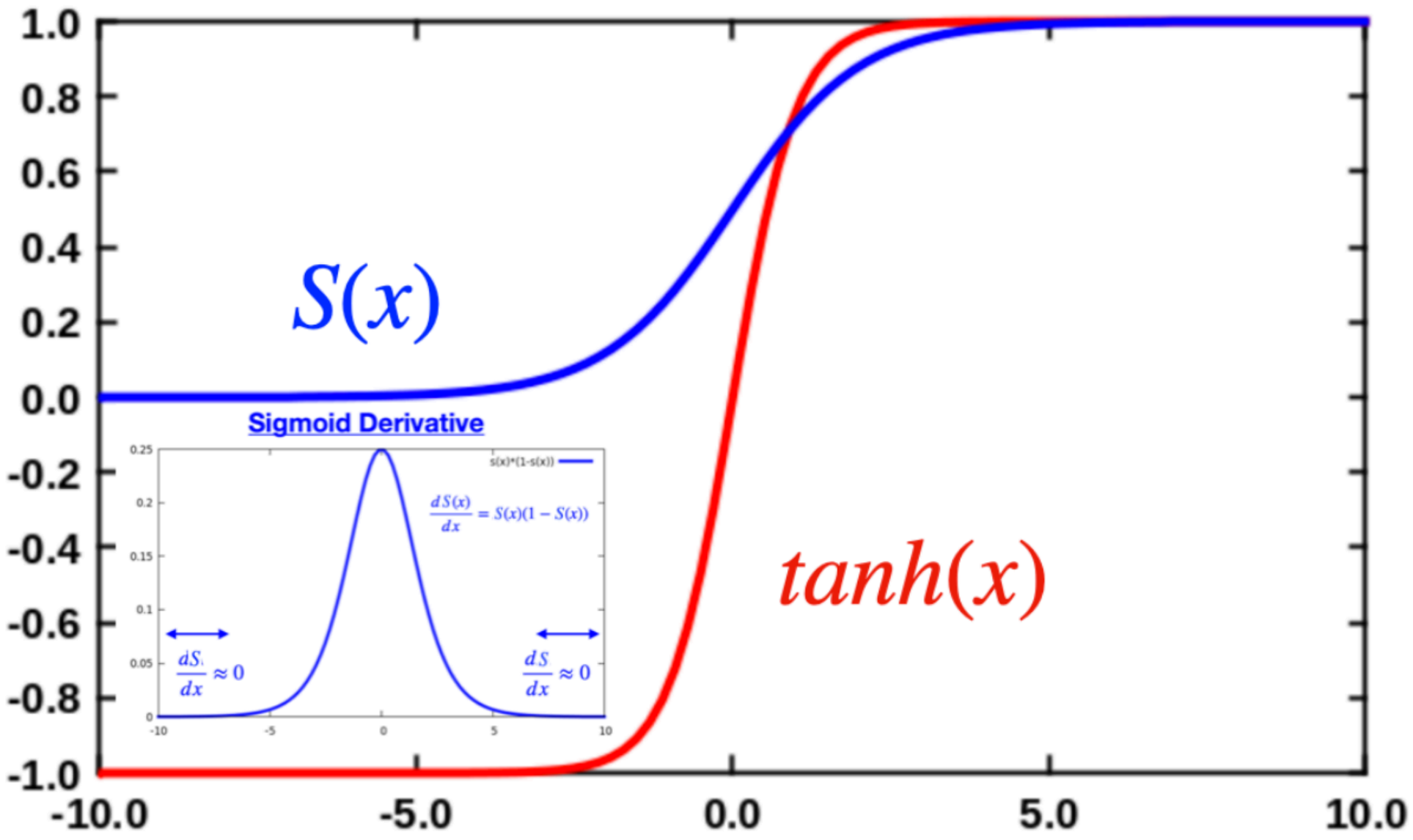
Don't apply activation function on output layer

Matrix form:

$$y = \mathbf{w}_{23} A\left(\mathbf{w}_{12} A\left(\mathbf{w}_{01}\mathbf{x} + \mathbf{b}_1 \right) + \mathbf{b}_2 \right) + \mathbf{b}_3$$

Activation: Sigmoid (i.e. logistic) and Tanh

- Activation functions, aka. non-linearities, make the ANN a non-linear interpolator
- Historically, these were very popular choices for activation functions.



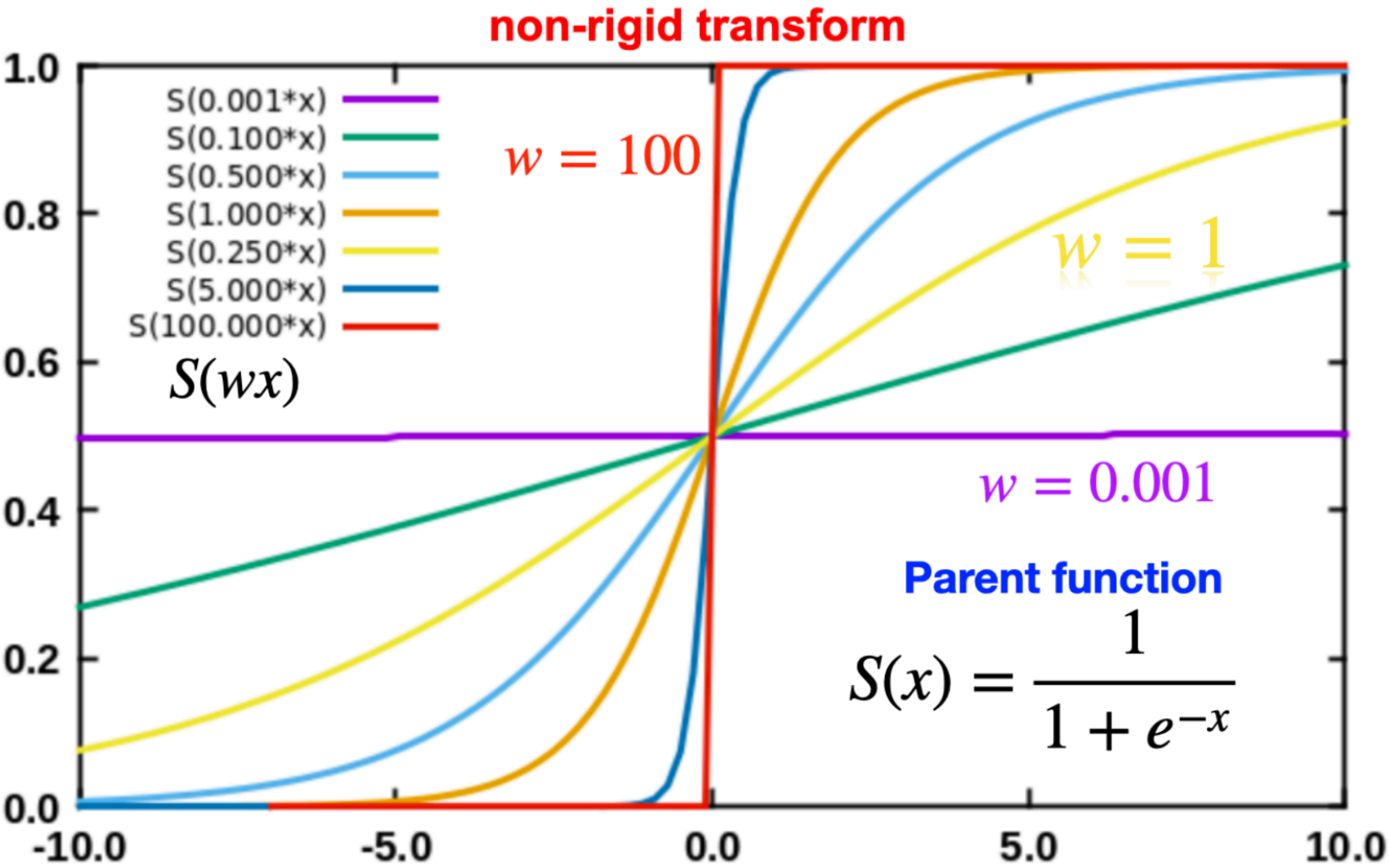
Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Hyperbolic Tangent function

$$\tanh(x) = 2\left(S(2x) - 0.5\right)$$

Logistic function: Width parameter



Small weights —> less overfitting

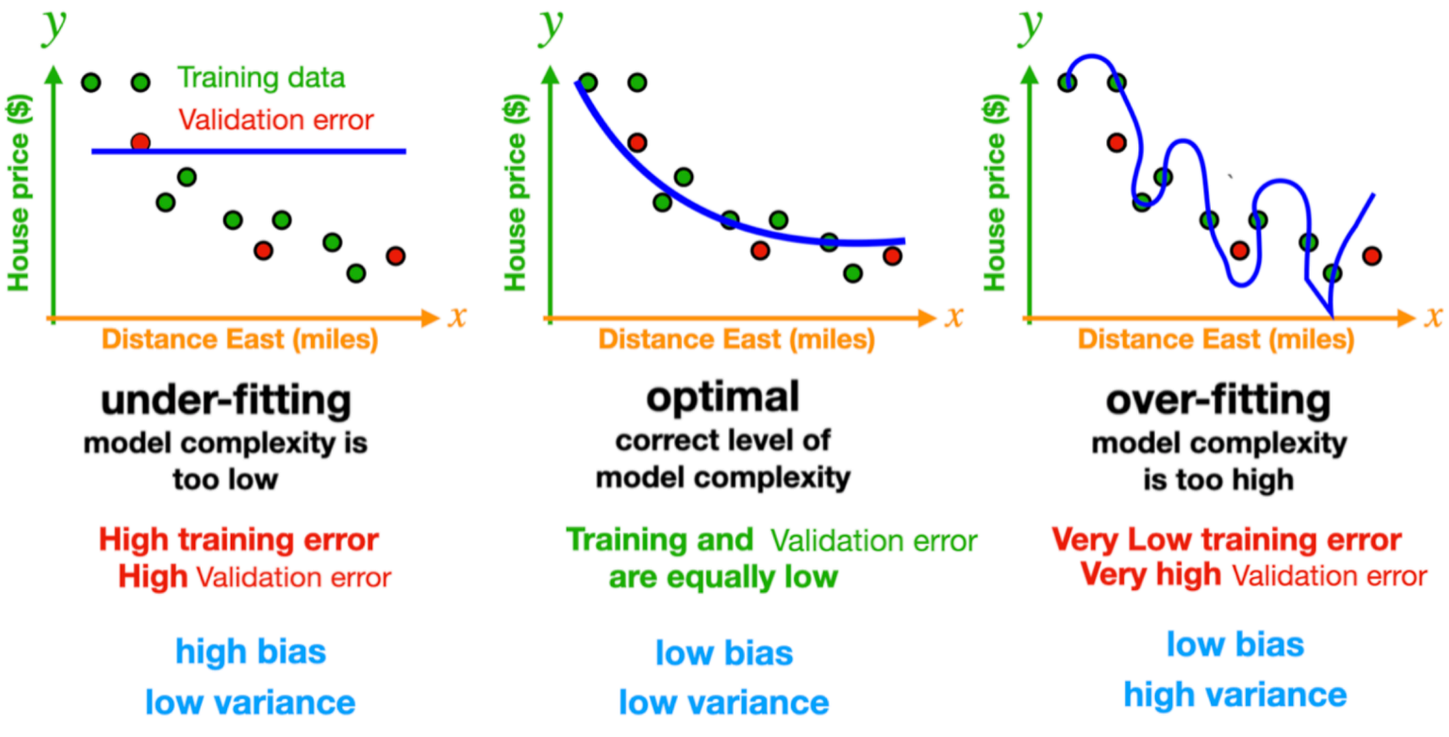
Regularization: constrain model flexibility

Deep nets have enough capacity to memorize the training set.

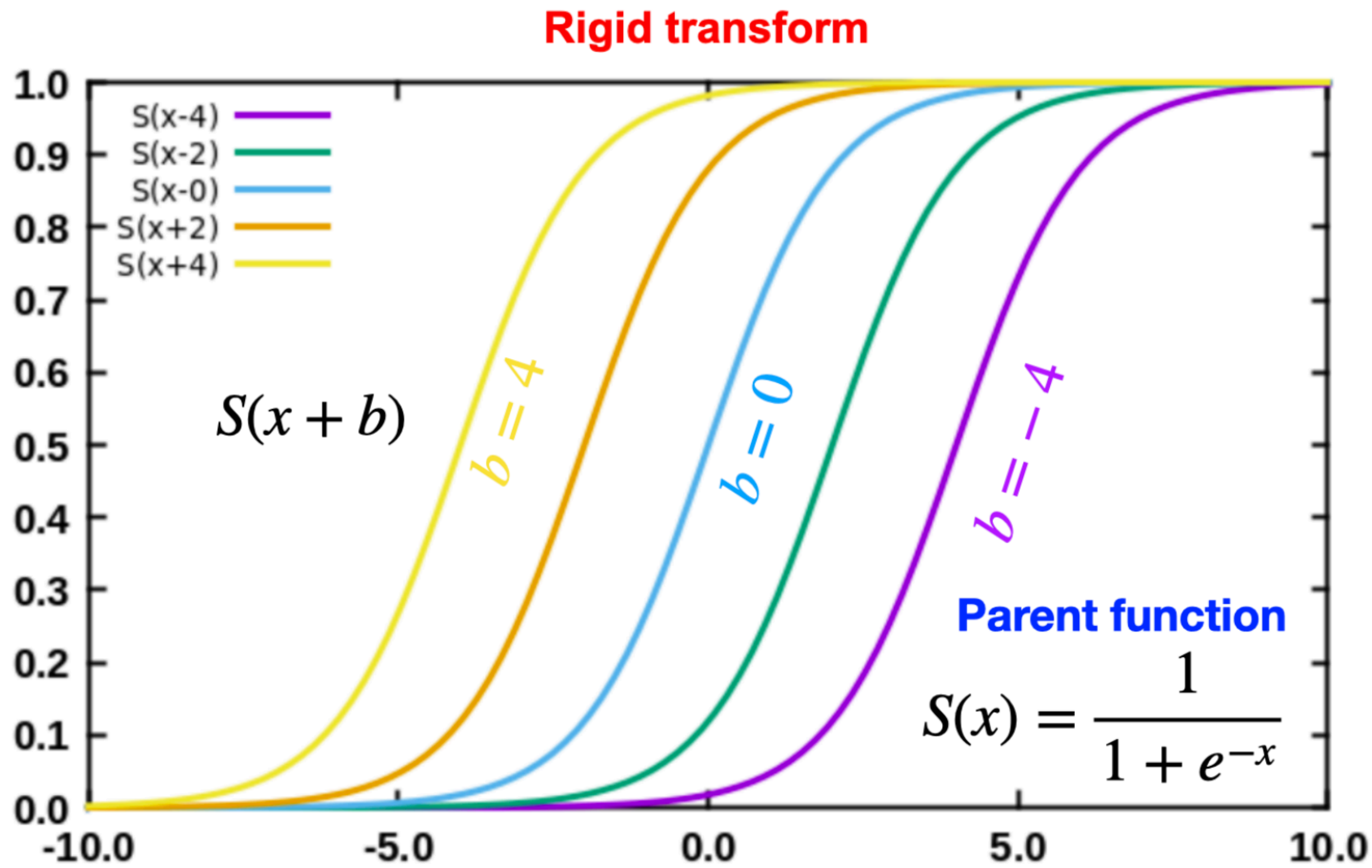
Regularization methods = tricks to PREVENT OVER FITTING

Method	Mechanism	Keras
L2 / weight decay	Penalize $\|w\|_2^2 \rightarrow$ shrink weights toward 0	`kernel_regularizer=l2(1e-4)`
L1	Penalize $\|w\|_1 \rightarrow$ drives weights *to* 0 (sparse)	`kernel_regularizer=l1(1e-5)`
Dropout	Randomly zero units each step → ensemble of subnets	`layers.Dropout(0.5)`
Early stopping	Halt when val loss turns up	`callbacks.EarlyStopping()`
Data augmentation	Enlarge the effective training set	`layers.RandomFlip()`, etc.
Batch norm	Noisy batch statistics act as a regularizer	`layers.BatchNormalization()`

All of these say the same thing in different dialects: **prefer the simpler function** when the data can't tell two models apart.

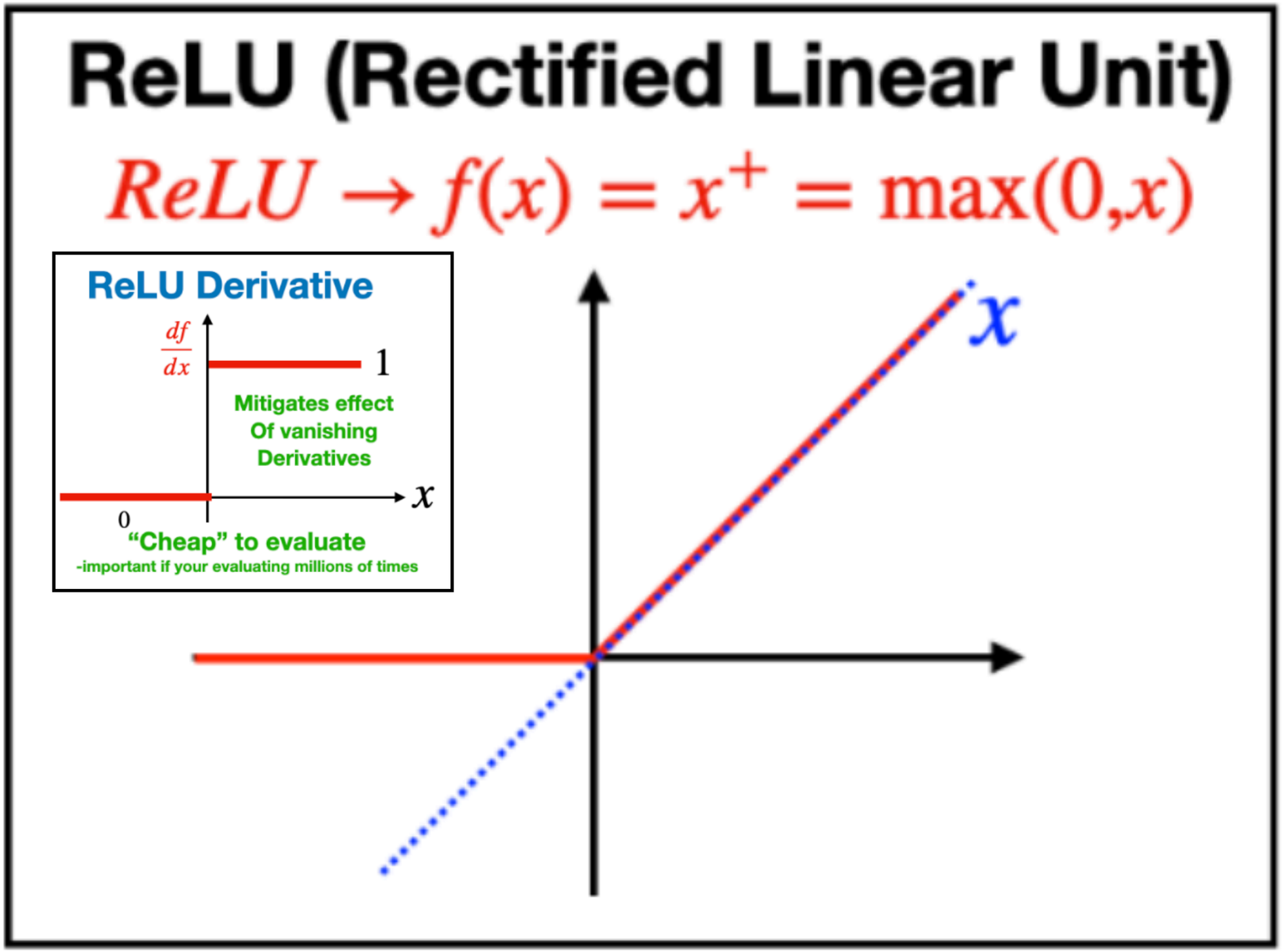


Logistic function: Shift parameter



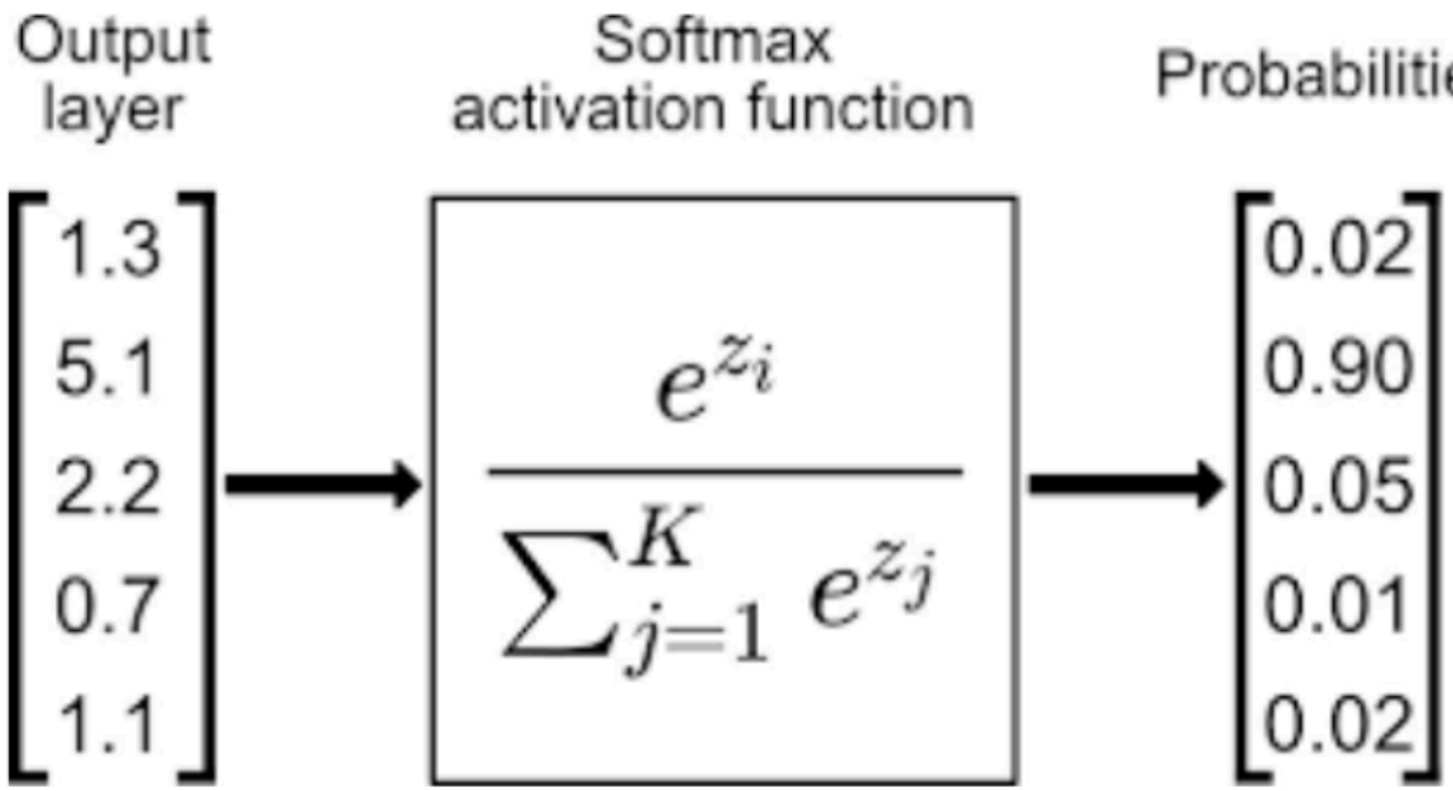
Activation: RELU

- A rectified linear unit (ReLU) is an activation function that introduces non-linearity
- Arguably, the popular common activation function, since it mitigates the vanishing gradient effect.



Activation: Softmax

- The `softmax function` converts a vector of numbers into a probability distribution by exponentiating each element and dividing by the sum of exponentiated values.
    - The division normalizes the vector so that the sum is equal to one, since $\sum_i p_i = 1$.
- Softmax returns an array of K probability scores (summing to 1). Each score is the probability that the sample belongs to one of our K classes.
- The standard (unit) soft-max function is defined by the formula



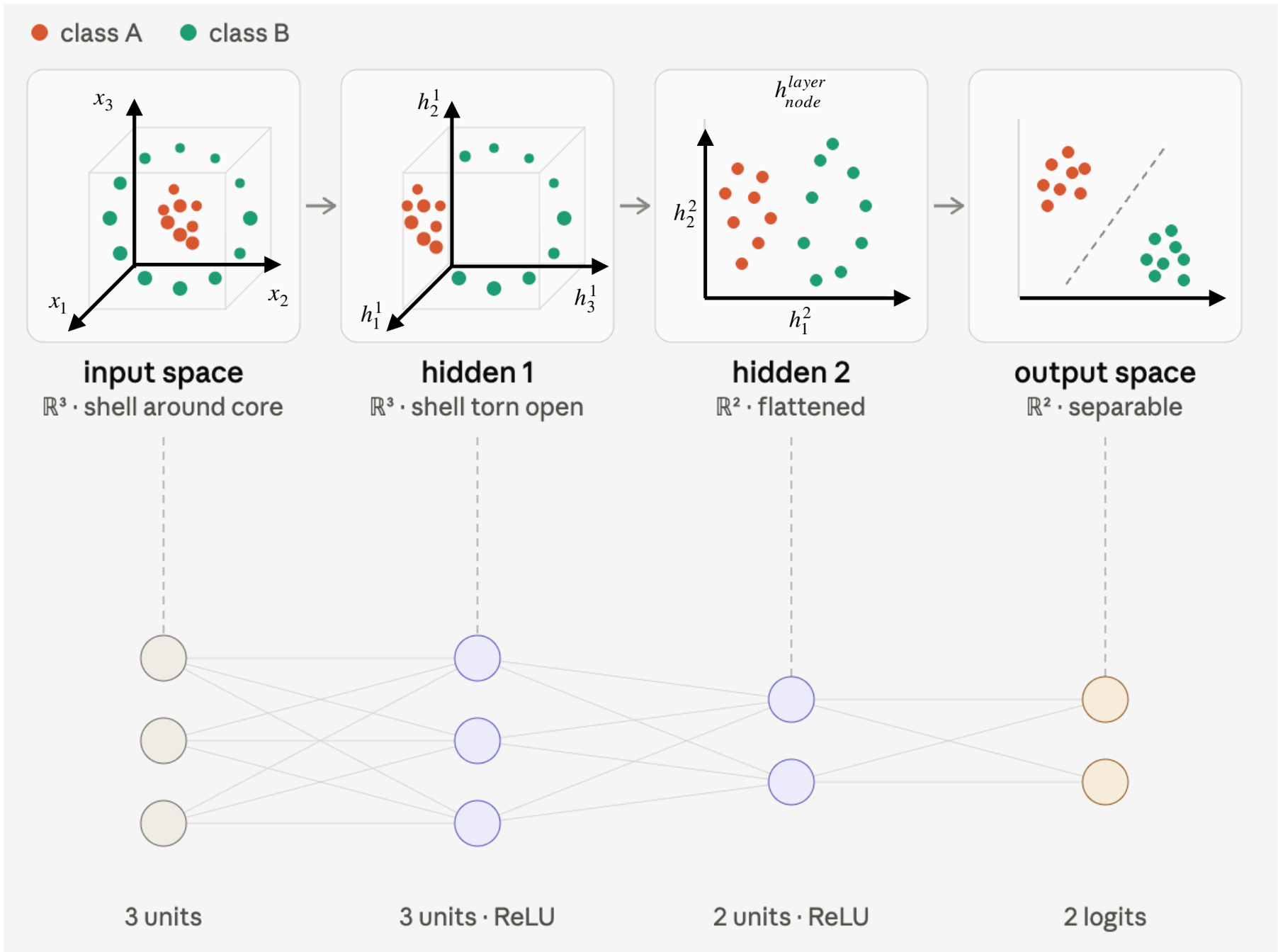
- This is very useful for `Multi-class classification`

Representation learning: Core idea in deep learning

Deep learning models repeatedly transform data into better and better representations spaces.

You can envision this as the data flowing through the network resulting in various intermediate representation spaces.

learned data transformation



What is better: In this context better means more suitable for a particular machine learning task.

Common Loss Functions

Regression

- **MSE (L2)** — default; smooth gradients, but heavily penalizes outliers.
  $\mathcal{L} = \frac{1}{N}\sum_i (\hat{y}_i - y_i)^2$
- **MAE (L1)** — robust to outliers; constant-magnitude gradient.
  $\mathcal{L} = \frac{1}{N}\sum_i |\hat{y}_i - y_i|$
- **Huber** — L2 near zero, L1 in the tails; best of both when outliers exist but you want smooth gradients.

Binary Classification

- **Binary cross-entropy** — $\hat{y} \in (0, 1)$ from a **sigmoid**; the standard for one-vs-rest and multi-label.
  $\mathcal{L} = -\frac{1}{N}\sum_i \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i) \right]$

Multi-Class Classification

- **Categorical cross-entropy** — $\hat{\mathbf{y}}$ from a **softmax** over C classes; $\mathbf{y}$ is one-hot.
  $\mathcal{L} = -\frac{1}{N}\sum_i \sum_{c=1}^{C} y_{ic} \log \hat{y}_{ic}$
- **Sparse categorical CE** — identical math, integer labels instead of one-hot (memory saver).

Training jargon: what actually counts as a "step"

- **Iteration** (*step*): one forward pass, one backward pass, **one gradient update**.
- **Gradient update**: the optimizer touching the weights once: $w \leftarrow w - \eta\,\hat{g}$.
  Synonymous with iteration; the two are 1:1.
- **Epoch**: one full sweep over all N training examples.

$$\text{iterations per epoch} = \lceil N/B \rceil$$

The three regimes

Regime	batch size B.	Updates / epoch	Gradient $\hat{g}$
Full batch	N	1	stable ∇L (Smoother training)
Mini-batch	$1 < B < N$	$\lceil N/B \rceil$	Unbiased estimate, some noise
Stochastic (SGD)	1	N	Unbiased, very noisy

Worked example — MNIST

$N = 60{,}000$, `batch_size=128`, `epochs=10`

- $\lceil 60000/128 \rceil = \mathbf{469}$ iterations per epoch (last batch holds 96)
- $469 \times 10 = \mathbf{4{,}690}$ gradient updates total
- Full batch would give **10** updates. Same data, ~470× less learning.

Noise can be good isn't a defect to be minimized: it's what lets mini-batch escape sharp minima and saddle points. Full batch is *more accurate per step* and *worse per epoch*.

Keras

1. **Load** — `keras.datasets.mnist.load_data()`
2. **Explore** — shapes, ranges, class balance, sample images
3. **Preprocess** — flatten to 784, rescale to [0, 1]
4. **Split** — train / validation / (locked) test
5. **Build & compile** — `Sequential` + `Dense` layers; loss, optimizer, metrics
6. **Train** — `fit()`, watching validation each epoch
7. **Evaluate** — training curves, test accuracy, confusion matrix, error inspection

Deep learning software tools

MOST POPULAR TOOLKITS

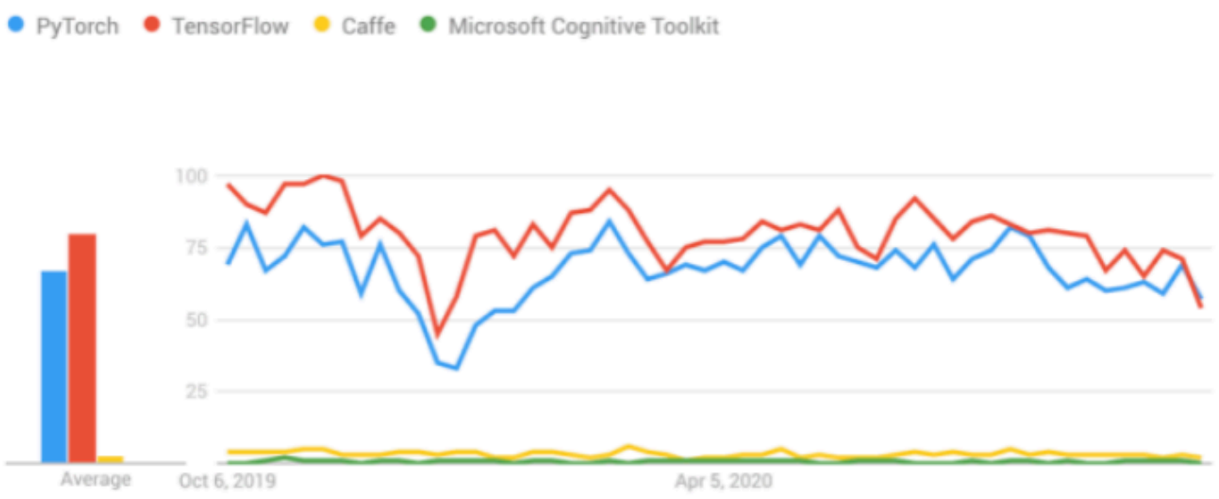
- **TENSOR-FLOW**
- **PYTORCH**
- **THEANO (MILA U-Montral)**
- **COGNITIVE TOOLKIT CNTK**
  - **(MICROSOFT)**

Keras

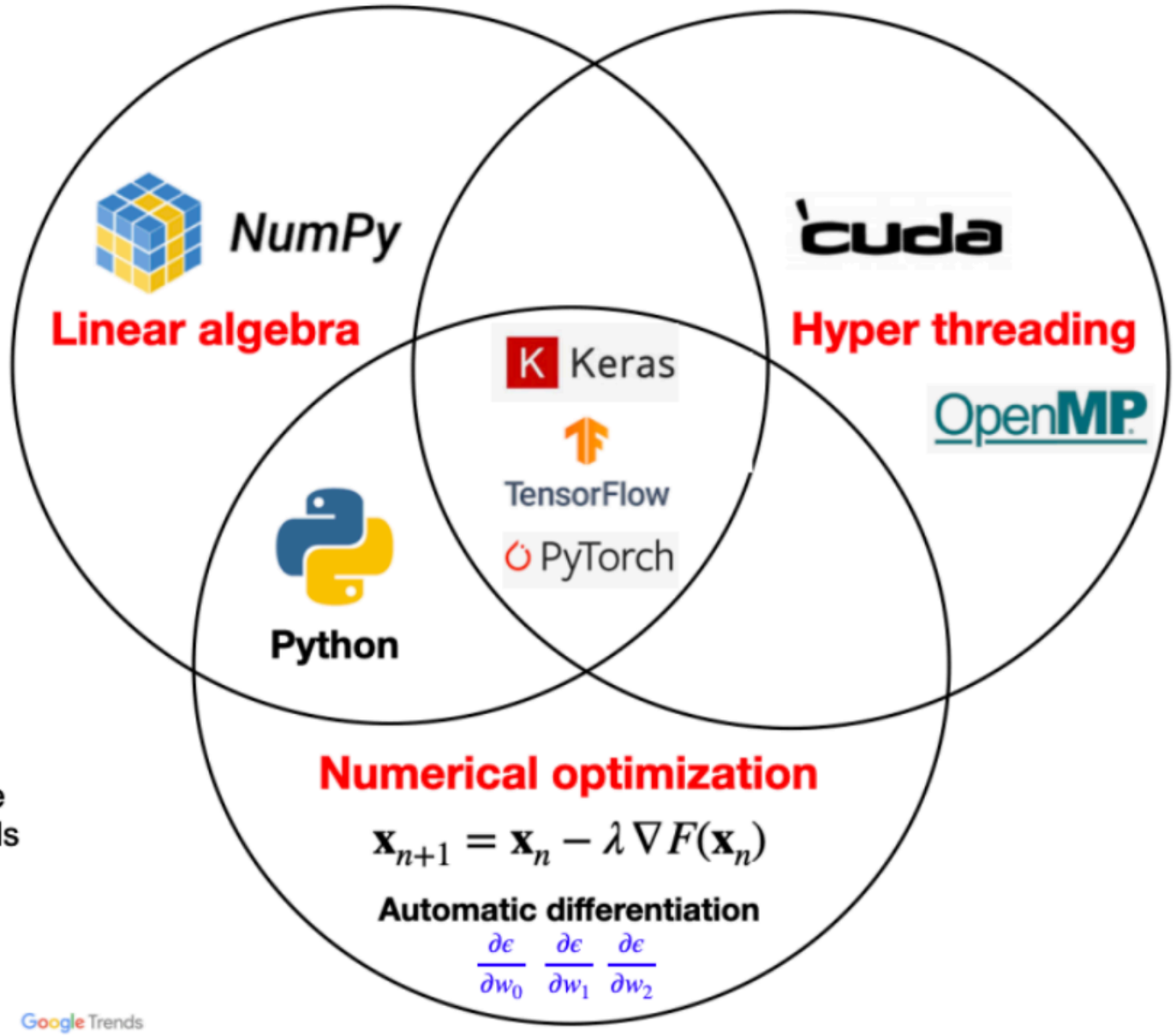
- Keras is an **open-source** software library
- **High-level API**: Application Programming Interface
- Provides a **python interface** for deep learning tools
- **Runs on top of TensorFlow,**
  - or Microsoft CNTK, or Theano
- **Easy to use**

Interest over time

Google Trends



United States. Past 12 months. Web Search. https://www.tooploox.com/blog/pytorch-vs-tensorflow-a-detailed-comparison



Keras + TensorFlow

PyTorch

- provide exposure
- Add extra line to vour resume

KERAS SUMMARY

Keras API reference

Keras

- Models API
- Layers API
- Callbacks API
- Data preprocessing
- Optimizers
- Metrics
- Losses
- Built-in small datasets
- Keras Applications
- Utilities
- Keras Tuner

The Model class

- Model class
- summary method
- get_layer method

The Sequential class

- Sequential class
- add method
- pop method

Model training APIs

- compile method
- fit method
- evaluate method
- predict method
- train_on_batch method
- test_on_batch method
- predict_on_batch method
- run_eagerly property

Keras layers API

- Creating custom layers
- Layers API overview

The base Layer class
Layer activations
Layer weight initializers
Layer weight regularizers
Layer weight constraints
Core layers
Convolution layers
Pooling layers
Recurrent layers
Preprocessing layers
Normalization layers
Regularization layers
Attention layers
Reshaping layers
Merging layers
Locally-connected layers
Activation layers

Optimizers

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl

Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Model saving & serialization APIs

- save method
- save_model function
- load_model function
- get_weights method
- set_weights method
- save_weights method
- load_weights method
- get_config method
- from_config method
- model_from_config function
- to_json method
- model_from_json function
- clone_model function

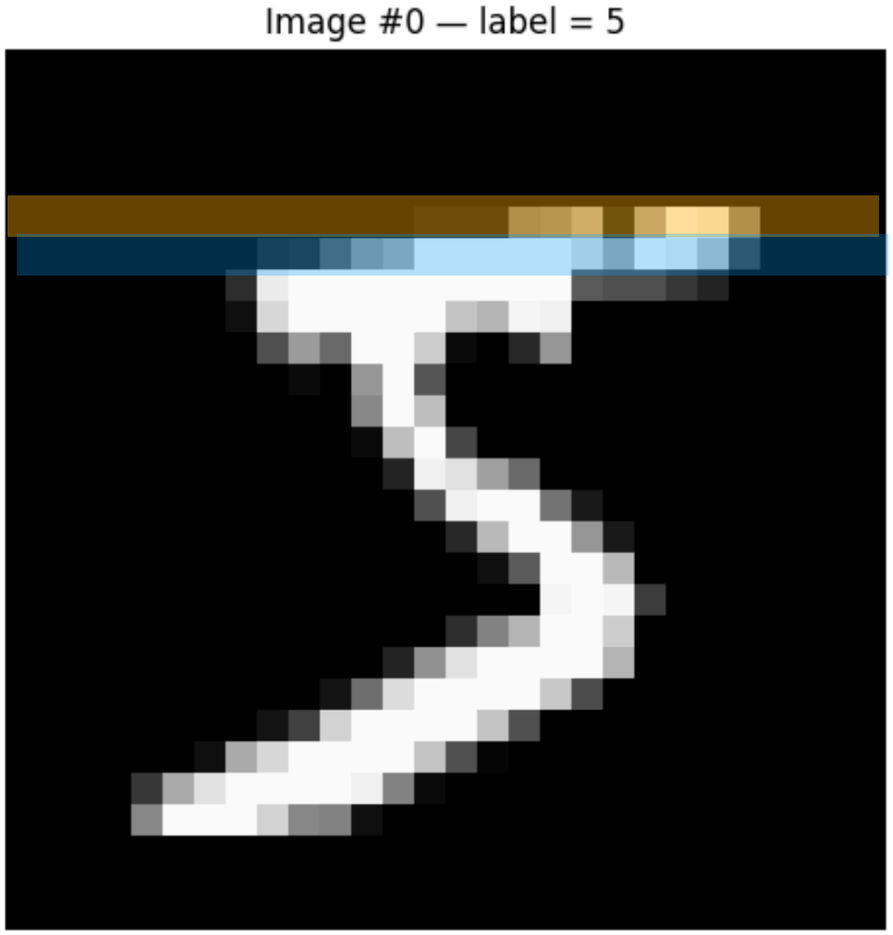
https://keras.io/api/models/

The “canonical” deep learning training workflow

- **Data** → collect, clean, **supervise**, split (train/val/test), augment
  - Garbage in, garbage out: this is where most projects live or die
- **Model Architecture** → choose a model that well suited to the data
  - MLP (simple data), CNN (grids/Images), RNN/Transformer (sequences), GNN (graphs)
- **Objective** (prediction quality scoring mechanism) → define a loss $\mathcal{L}(\theta)$ that encodes "good"
  - Regression: Mean square error MSE
  - Binary classification: Binary Cross-entropy
  - Multi-class classification: Categorical Cross-entropy
- **Optimization** (learning) → back-prop + SGD/Adam, LR schedule, regularization
  - We are just rolling downhill on a very high-dimensional surface
- **Evaluation** → validation metrics, error analysis, ablations
  - Tune on val, touch test *once*
- **Deployment** → export, serve, monitor drift, iterate

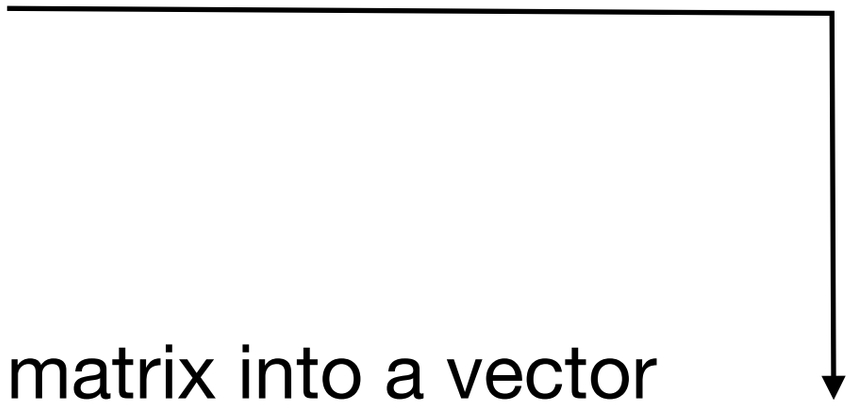
Aside: "Canonical" means conforming to a standard, authoritative, or generally accepted form the version of something that serves as the reference point against which variants are judged (e.g., the canonical form of a matrix, or the canonical version of a text).

Data: MNIST



28x28

Flatten



Flatten: unwraps (reshapes) matrix into a vector



$x \rightarrow (1,784)$

Keras Models:

Keras →

`Model` class

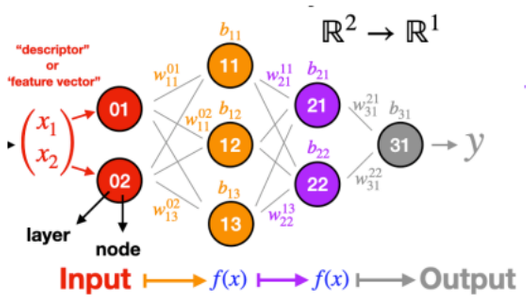
```
tf.keras.Model()
```

`Model` groups layers into an object with training and inference features.

Arguments

- **inputs**: The input(s) of the model: a `keras.Input` object or list of `keras.Input` objects.
- **outputs**: The output(s) of the model. See Functional API example below.
- **name**: String, the name of the model.

Neural network



- *Layers*, which are combined into a *network* (or *model*)
- The *input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

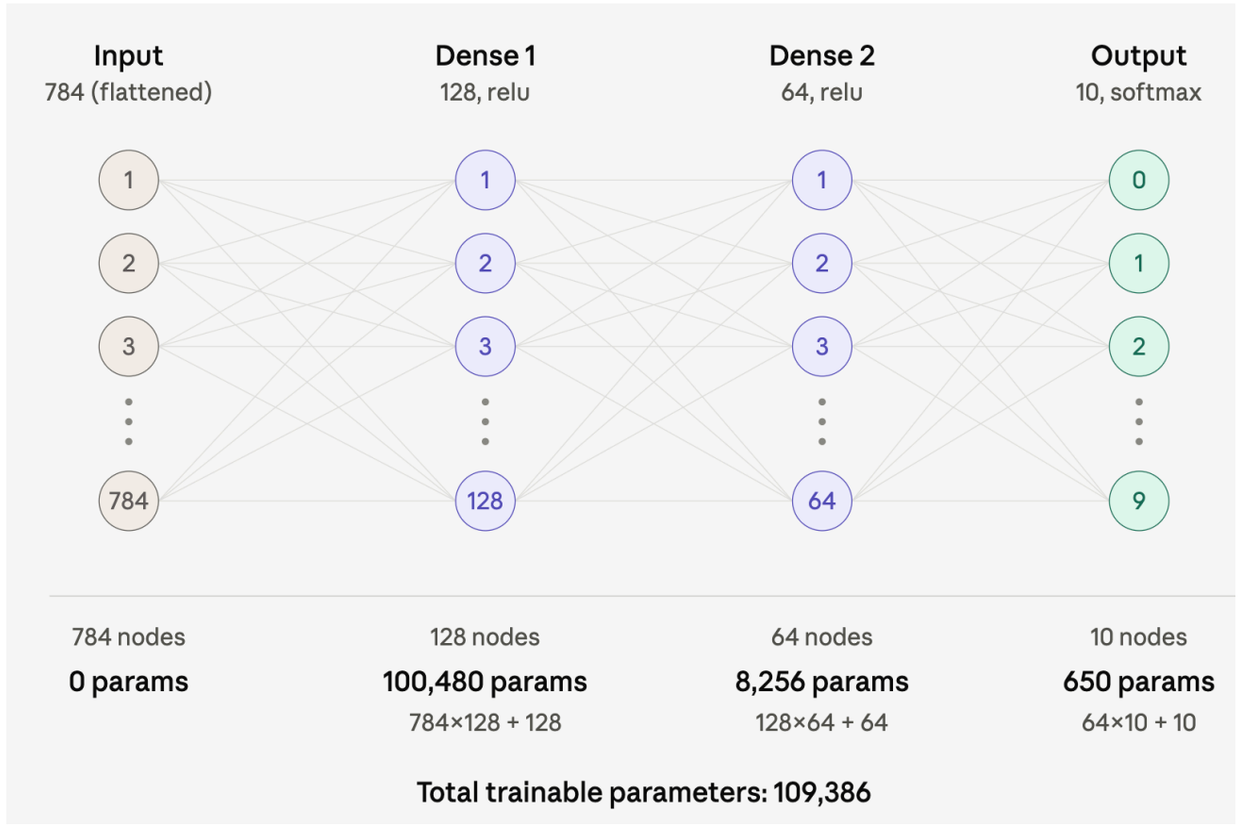
```
model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation="relu"),
    layers.Dense(64,  activation="relu"),
    layers.Dense(10,  activation="softmax"),
])

model.summary()
```

Keras Models:

```
model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation="relu"),
    layers.Dense(64,  activation="relu"),
    layers.Dense(10,  activation="softmax"),
])

model.summary()
```




```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

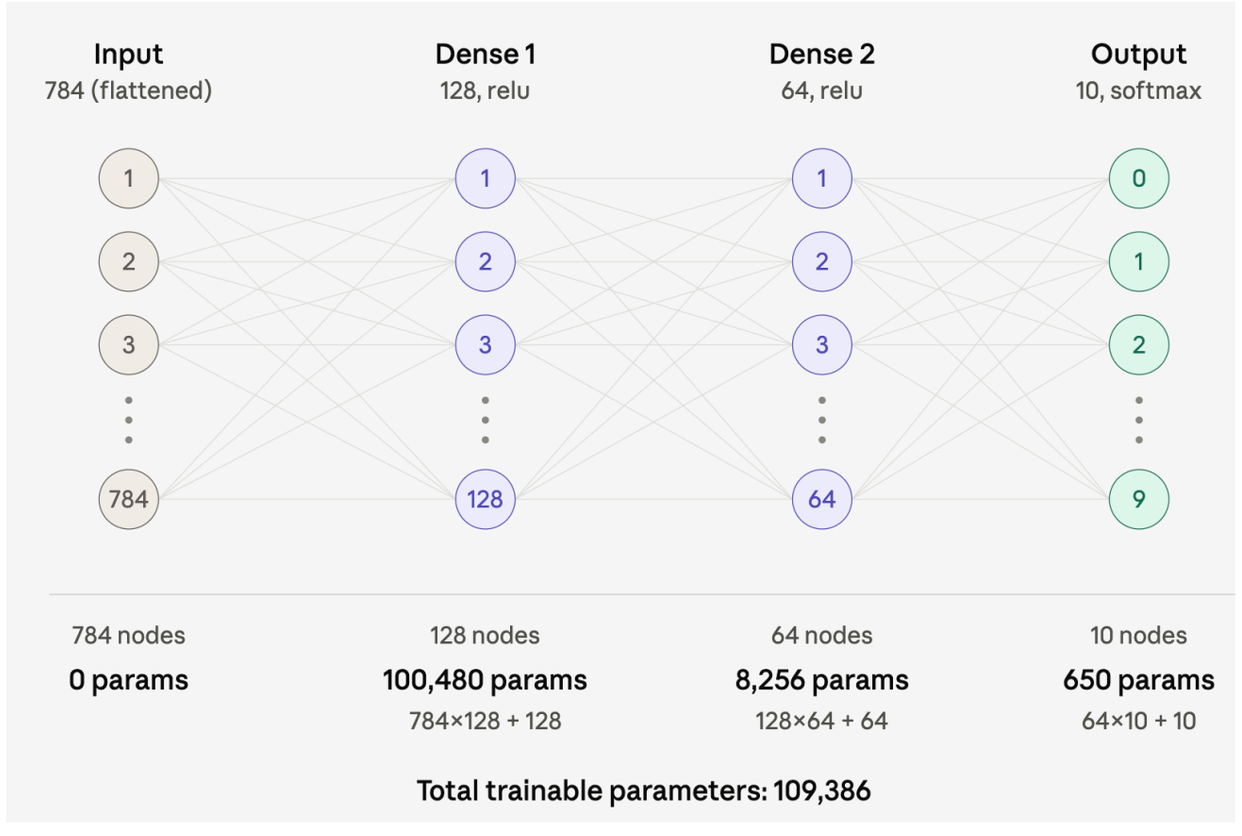
=================================================================
Total params: 109386 (427.29 KB)
Trainable params: 109386 (427.29 KB)
Non-trainable params: 0 (0.00 Byte)
```

Keras Models:

Build model

```
model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation="relu"),
    layers.Dense(64,  activation="relu"),
    layers.Dense(10,  activation="softmax"),
])

model.summary()
```




```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 109386 (427.29 KB)
Trainable params: 109386 (427.29 KB)
Non-trainable params: 0 (0.00 Byte)
```

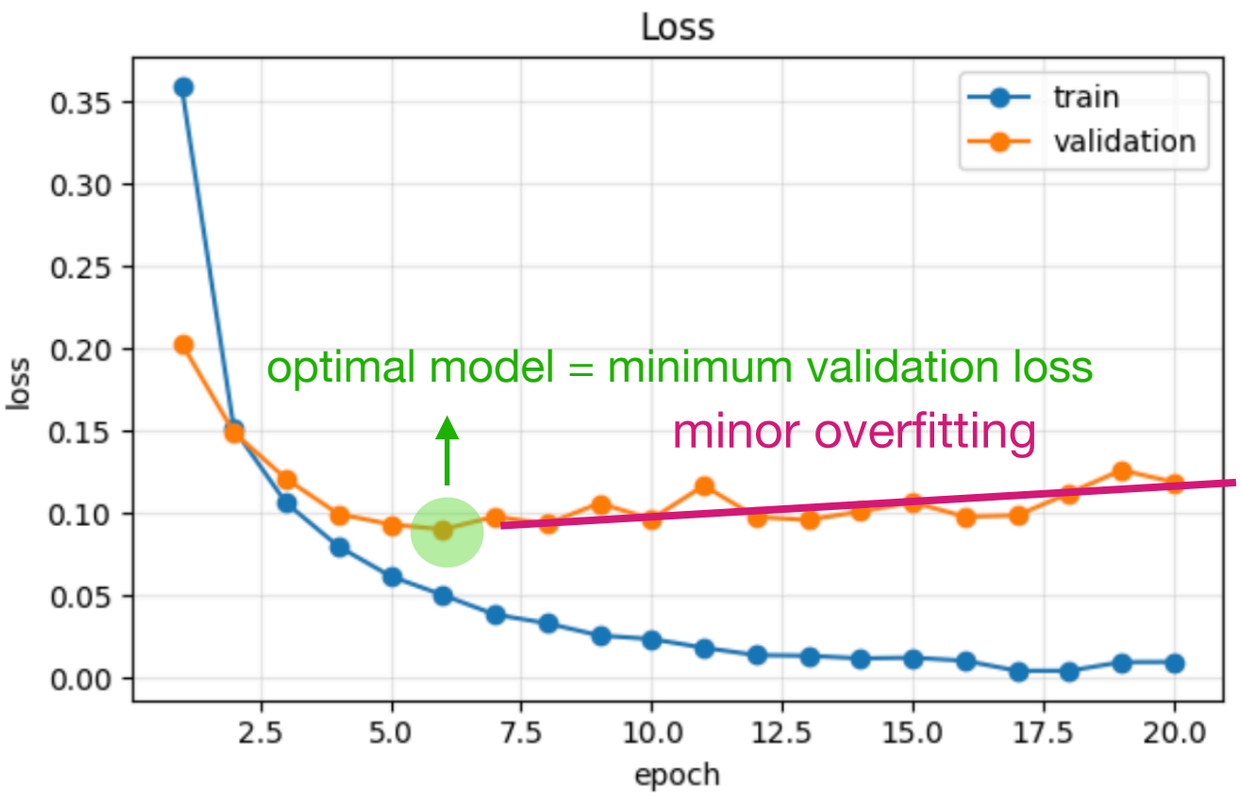
Compile model

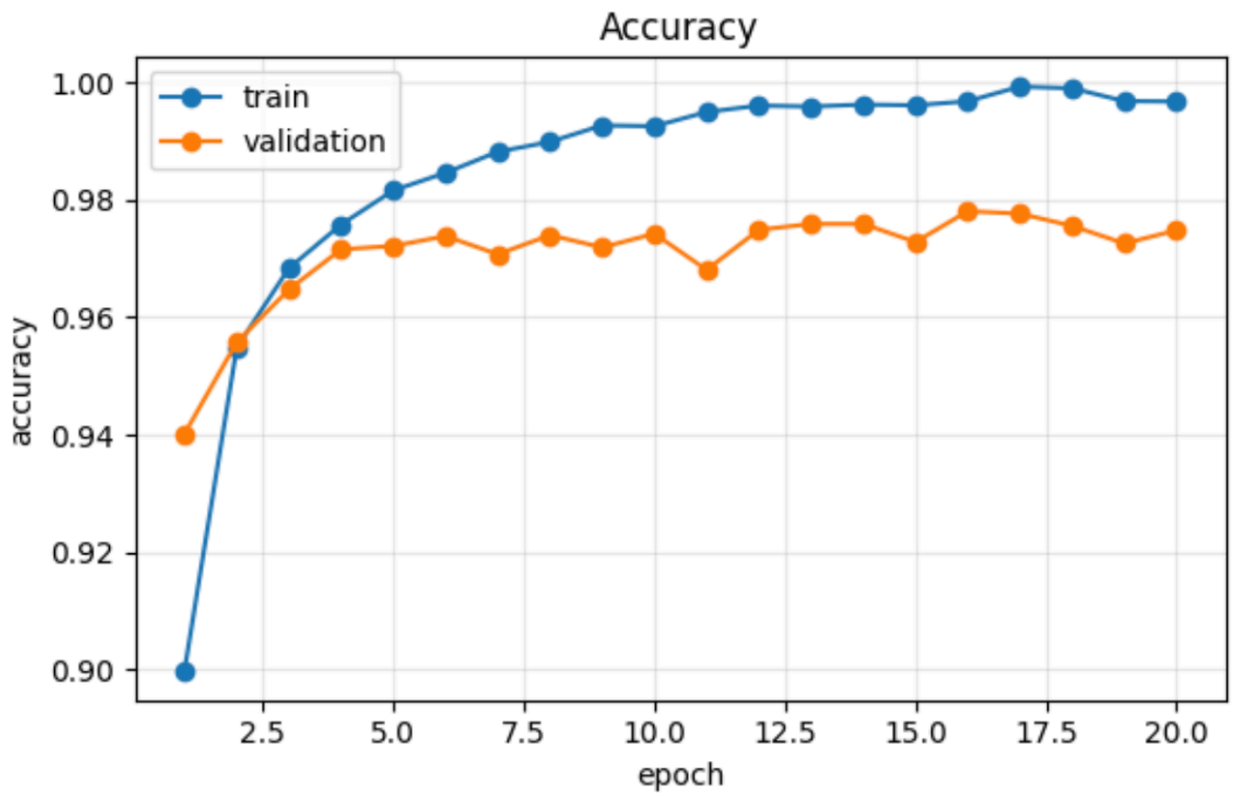
```
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Remember: Parameters are learned directly by the optimizer during training, **Hyper-parameters** (e.g. learning rate) are choices made by humans that **CONTROL** training

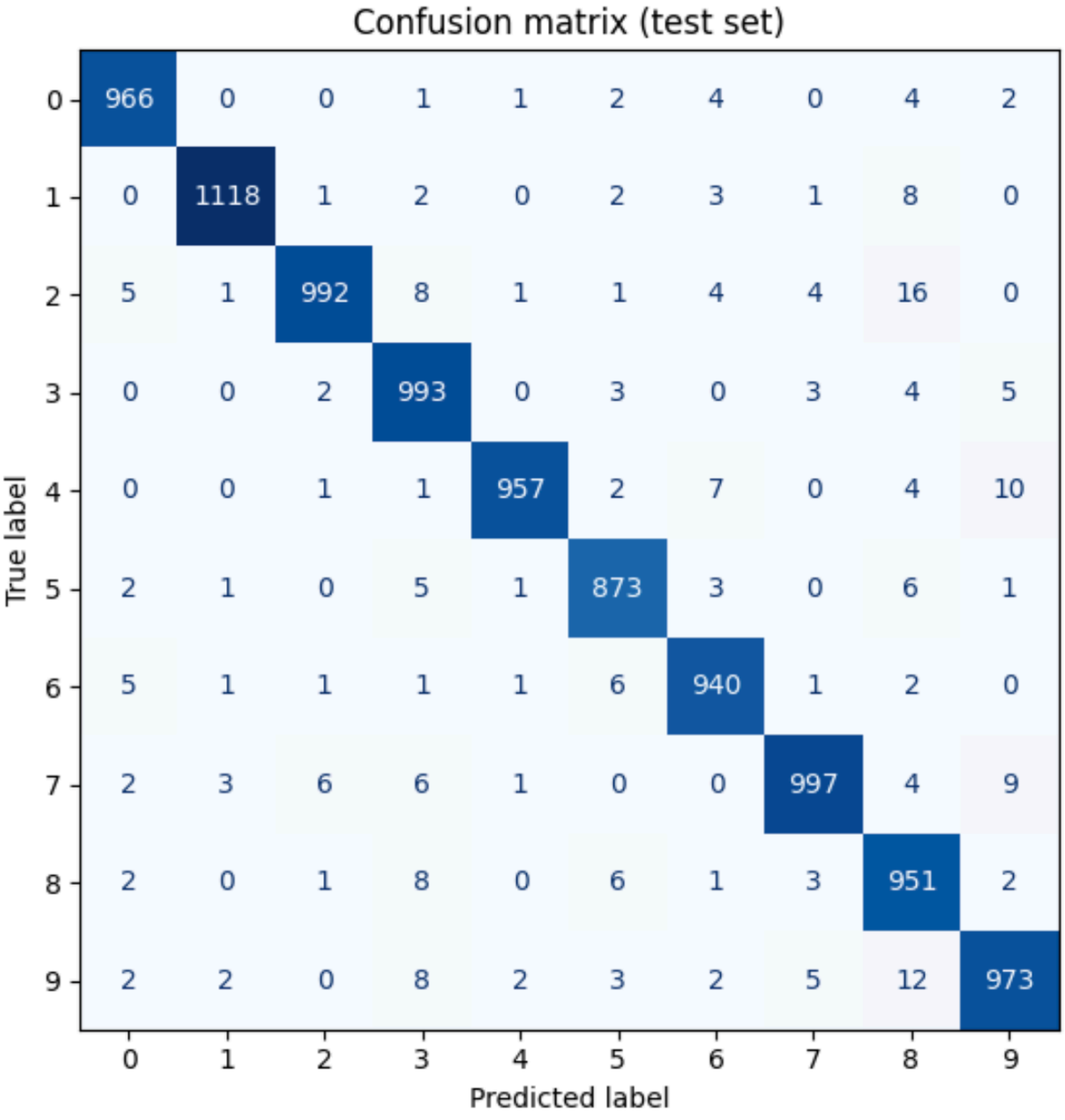
Train model

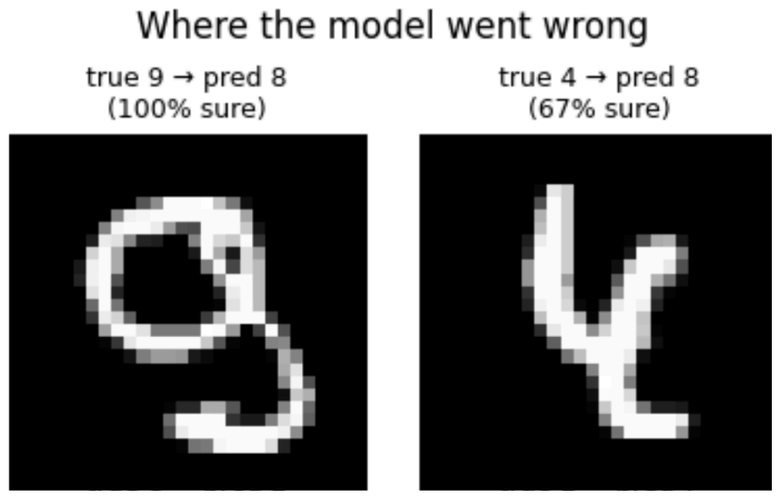
```
history = model.fit(
    x_train, y_train,
    validation_data=(x_val, y_val),
    epochs=20,
    batch_size=128,
    verbose=2,
)
```





Results


Confusion matrix (test set)
True label
Predicted label
0 966 0 0 1 1 2 4 0 4 2
1 0 1118 1 2 0 2 3 1 8 0
2 5 1 992 8 1 1 4 4 16 0
3 0 0 2 993 0 3 0 3 4 5
4 0 0 1 1 957 2 7 0 4 10
5 2 1 0 5 1 873 3 0 6 1
6 5 1 1 1 1 6 940 1 2 0
7 2 3 6 6 1 0 0 997 4 9
8 2 0 1 8 0 6 1 3 951 2
9 2 2 0 8 2 3 2 5 12 973
0 1 2 3 4 5 6 7 8 9


Where the model went wrong
true 9 → pred 8
(100% sure)
true 4 → pred 8
(67% sure)