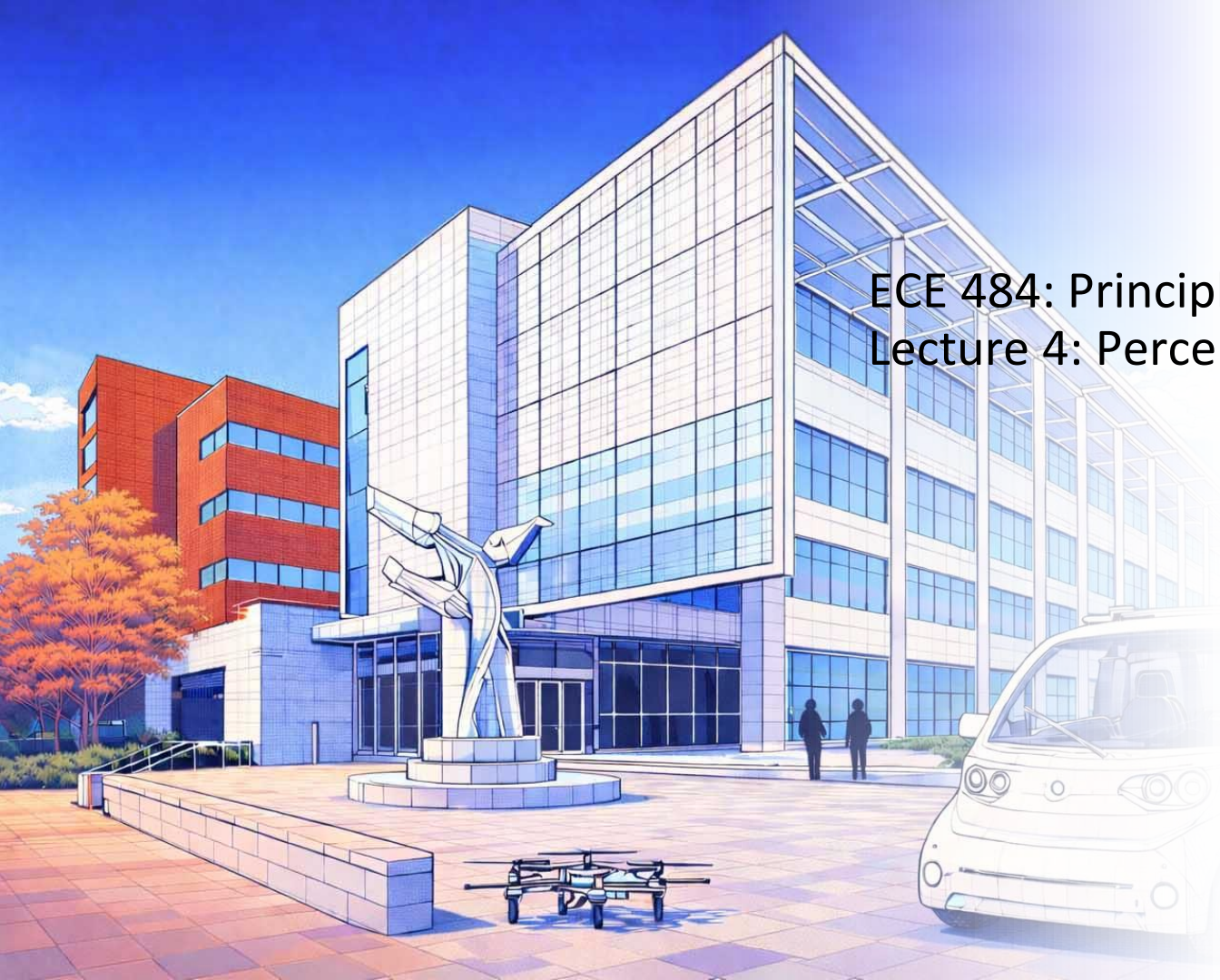


ECE 484: Principles of Safe Autonomy
Lecture 4: Perception: Neural Networks

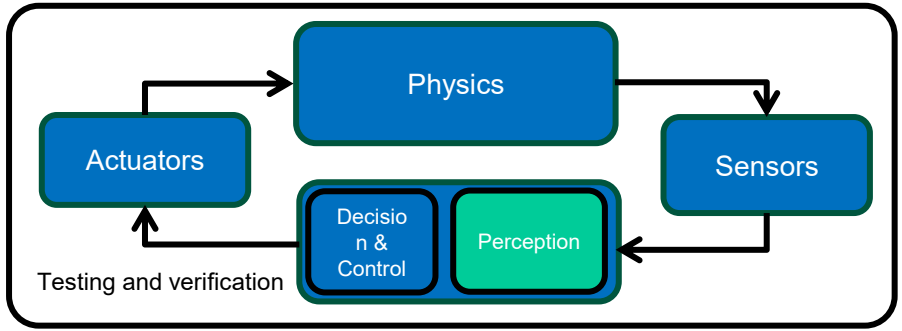
Prof. Huan Zhang

Fall 2026

Slides adapted in part from materials by Prof. Sayan Mitra.

Perception in an autonomous system

Perception converts sensor signals to state estimates for decision making and control



Examples of state estimates:

- **Type** or **Class** of lead vehicle, traffic sign
- **Pose**: position of ego on map, relative to lanes, relative to lead vehicle; attitude of drone relative to marker
- **Speed,** angular speed of ego and others
- Acceleration, intention of the pedestrian
- Semantic estimates are labels (type, sign, intention); geometric estimates are numbers (position, speed, pose)

Fundamental questions

- What can be estimated? Observability
- What resources (bits) are needed for achieving certain quality of estimation?
- How to build estimators?

Recognizing object classes from camera images: Classification

Goal: Learn/build a function that predicts the object in an image

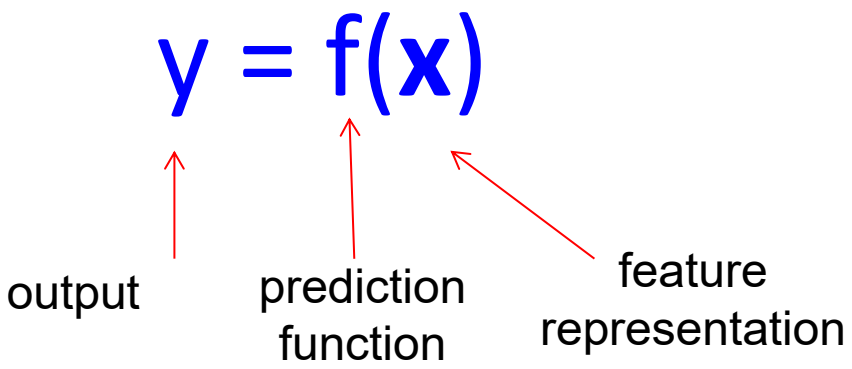
Apply a prediction function to a **representation** of the image to get the desired output:

f() = “apple”

f() = “tomato”

f() = “cow”

Statistical learning framework: Train classifier from training data



Training: given a *training set* of **labeled** examples $\{(\mathbf{x}_1,y_1), \ldots, (\mathbf{x}_N,y_N)\}$, estimate the prediction function f by minimizing the prediction error on the training set

Validation: choose hyperparameters (e.g. model size, learning rate) by the error on a separate validation set

Testing: apply f to a never-before-seen *test input* **x** and output the predicted value $y = f(\mathbf{x})$

Training

Training Images

Training Labels

Image Features

Training

Learned model

Testing

Test Image

Image Features

Learned model

Prediction

Slide credit: D. Hoiem

Linear classifiers

Images or feature representations of images $\boldsymbol{x} \in \mathbb{R}^d$

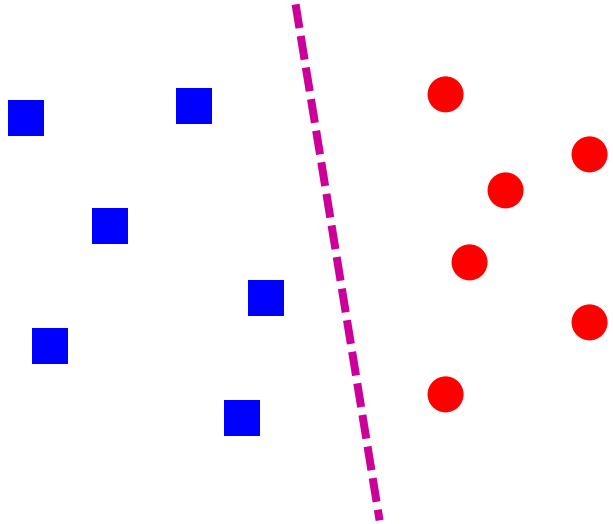
A constant weight vector $\boldsymbol{w} \in \mathbb{R}^d$ and a scalar bias $b \in \mathbb{R}$ define a binary classifier

$$f(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}^T \boldsymbol{x} + b)$$

The training task is to find the weight and the bias $(\boldsymbol{w}, b)$ that define a linear separator for the given data set

The separators are also called decision boundaries

SVM classifiers choose the linear separator that maximizes the margin, i.e. the distance from the separator to the closest training points

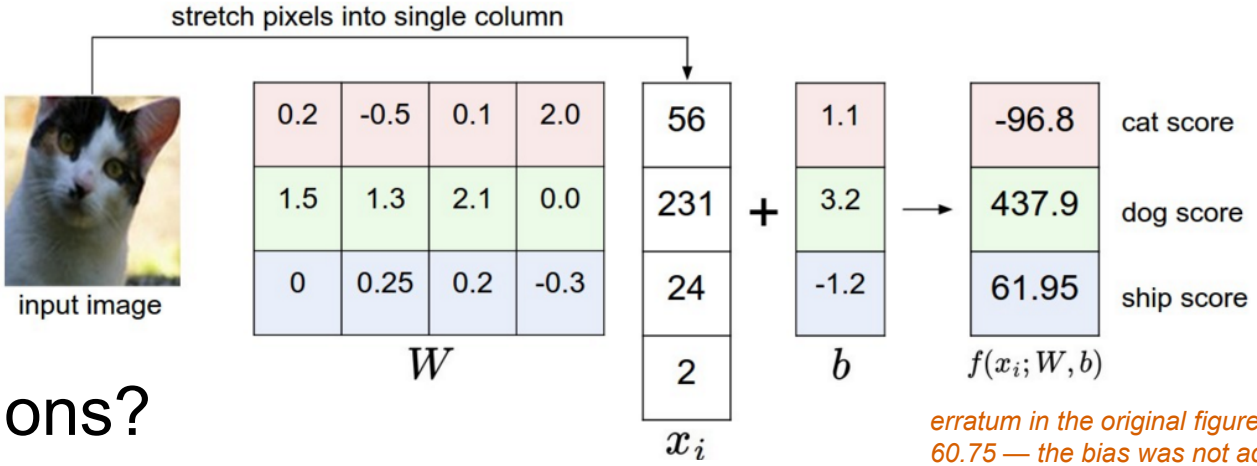


Read support vector machines (SVM) https://greitemann.dev/svm-demo

Read about SIFT, HOG, bag of visual words to learn about image features.

Visualizing linear classifiers with many classes

K classes: scores $\boldsymbol{s} = W\boldsymbol{x} + \boldsymbol{b}$, $W \in \mathbb{R}^{K \times d}$ (one row per class), $\boldsymbol{b} \in \mathbb{R}^{K}$; predicted class $\mathrm{argmax}_k s_k$;
softmax $p_k = e^{s_k} / \sum_j e^{s_j}$ turns scores into probabilities



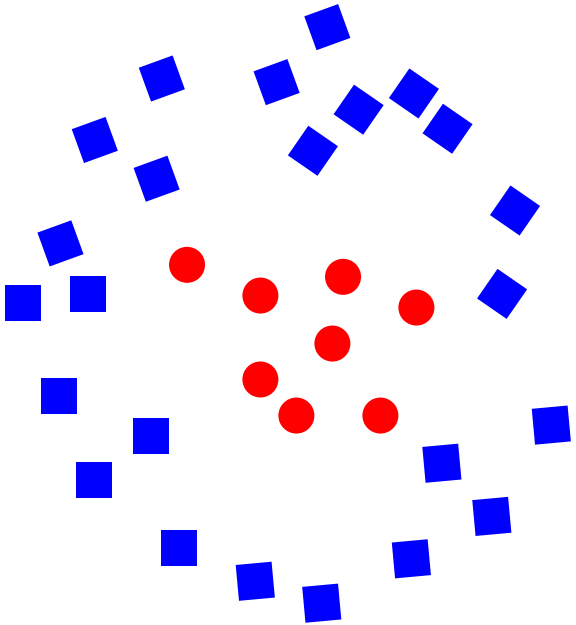
erratum in the original figure: ship score = 61.95 + (−1.2) = 60.75 — the bias was not added

Limitations?

Source: Andrej Karpathy, http://cs231n.github.io/linear-classify/

Limitations of Linear Classifiers

- Input: A feature vector $\boldsymbol{x} \in \mathbb{R}^d$.
- Weights and bias: $\boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}$.
- Prediction (binary): score $\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b$, class $\text{sgn}(\hat{y})$
- Limitations: The data may not be linearly separable
- Linear decision boundaries may not capture nonlinear relationships between classes
- How can we address this limitation and build more expressive classifiers?
- "Stacking" multiple linear functions — will that work?



Nonlinearity via Neural Networks

A **neural network** is a function $f_{NN}: \mathbb{R}^d \to \mathbb{R}^k$ defined as a composition of layers of linear and nonlinear transformations.

A single layer performs an affine transformation (linear map + bias) followed by a nonlinear activation

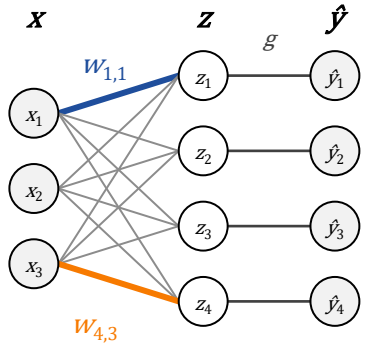
$\boldsymbol{w}^T\boldsymbol{x} + b \in \mathbb{R}$ (one unit: a scalar output)

$\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^m$ weight matrix $W \in \mathbb{R}^{m\times d}$ bias vector $\boldsymbol{b} \in \mathbb{R}^m$

$\hat{\boldsymbol{y}} = g(\boldsymbol{z}) = g(W\boldsymbol{x} + \boldsymbol{b})$ nonlinear activation $g: \mathbb{R}^m \to \mathbb{R}^m$ applied elementwise: $g(\boldsymbol{z})_j = g(z_j)$

g: activation function e.g. $\mathrm{ReLU}(z) = \max(z, 0)$, $\mathrm{sigmoid}(z) = \frac{e^z}{1+e^z}$

1-layer network, d = 3, m = 4; bias not drawn



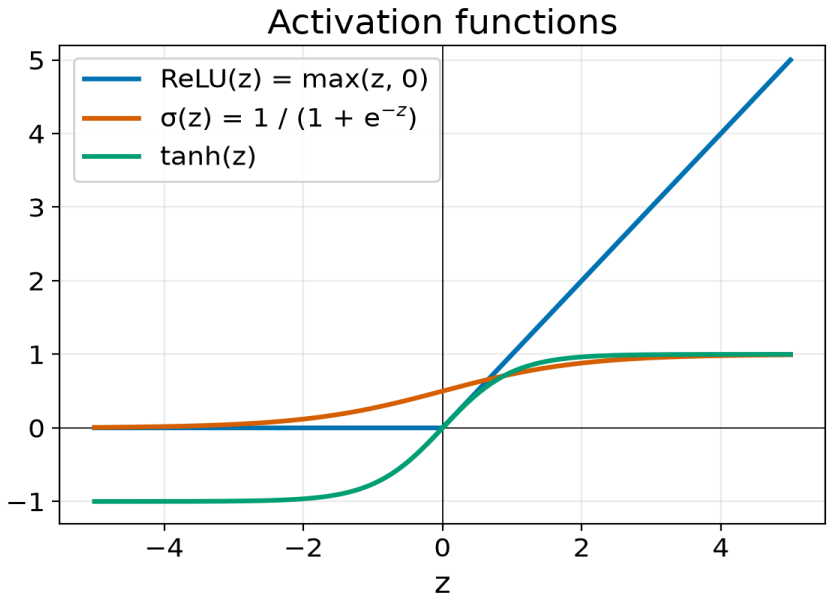
W: 4×3 = 12 lines

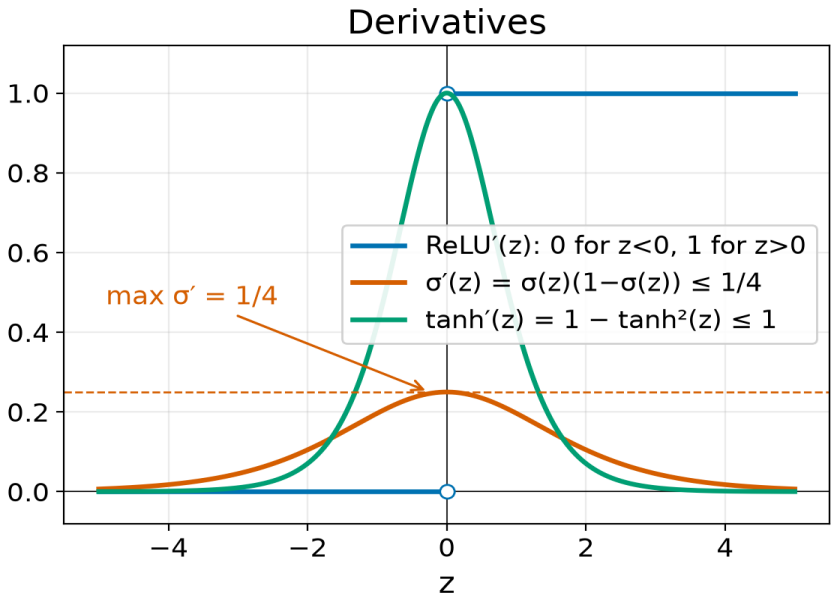
each line = one weight (an entry of W)

$\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b}$, $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$ elementwise

1-Layer Perceptron

Activation functions and their derivatives





ReLU (rectified linear unit): $\mathrm{ReLU}(z) = \max(z, 0)$

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

tanh: $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} = 2\sigma(2z) - 1$

ReLU′: $\mathrm{ReLU}'(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \end{cases}$ (undefined at 0; use 0)

sigmoid′: $\sigma'(z) = \sigma(z)\,(1 - \sigma(z)) \leq \frac{1}{4}$

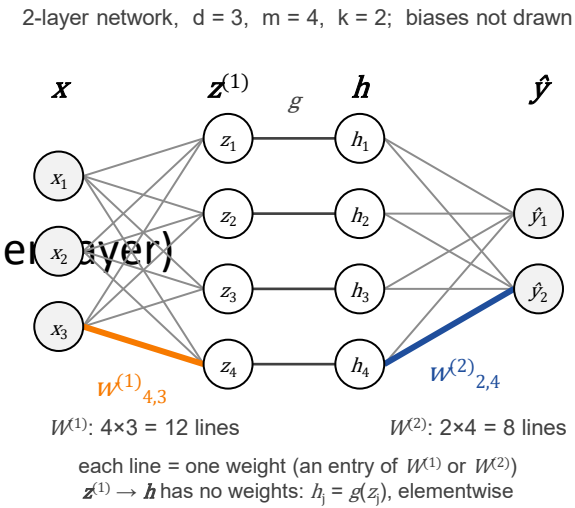
tanh′: $\tanh'(z) = 1 - \tanh^2(z) \leq 1$

Nonlinearity via Neural Networks

A **neural network** is a function $f_{NN}: \mathbb{R}^d \to \mathbb{R}^k$ defined as a composition of layers of linear and nonlinear transformations.

2-layer network with one hidden layer and input $\boldsymbol{x} \in \mathbb{R}^d$

- $\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \ \boldsymbol{h} = g(\boldsymbol{z}^{(1)})$ (hidden layer)
- $\hat{\boldsymbol{y}} = W^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$ (output layer)



$$\hat{\boldsymbol{y}} = W^{(2)}g\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}$$

$W^{(1)} \in \mathbb{R}^{m\times d}, W^{(2)} \in \mathbb{R}^{k\times m}$ are the weights (m = number of hidden units)

$\boldsymbol{b}^{(1)} \in \mathbb{R}^m, \boldsymbol{b}^{(2)} \in \mathbb{R}^k$ are the biases

g: activation function e.g. $\mathrm{ReLU}(z) = \max(z, 0)$, $\mathrm{sigmoid}(z) = \frac{e^z}{1+e^z}$

Multi-Layer Perceptron (MLP)

Example k=2 for lane boundary parameters, k=6 for pose components

Universal Approximation Theorem

Theorem (Cybenko 1989; Hornik 1991; Leshno et al. 1993). Let f be continuous on a compact set $K \subset \mathbb{R}^d$. For every $\varepsilon > 0$ there is a network with ONE hidden layer, enough hidden units, activation g and a LINEAR output layer such that $|f(\boldsymbol{x}) - f_{NN}(\boldsymbol{x})| < \varepsilon$ for all $\boldsymbol{x} \in K$ — for every continuous sigmoidal g (Cybenko), and in fact for every continuous non-polynomial g, e.g. ReLU (Leshno et al.).

1-D intuition: f ≈ a sum of towers (bumps). A steep step ≈ $\sigma(w(x-a))$ for large w; a tower = difference of two steps; more hidden units ⇒ more towers ⇒ smaller error.

With ReLUs, two units make a step: $\mathrm{ReLU}(w(x-a)+1) - \mathrm{ReLU}(w(x-a))$.

Caveat: an existence result — no bound on the width, nothing about finding the weights by training, nothing about inputs outside K.

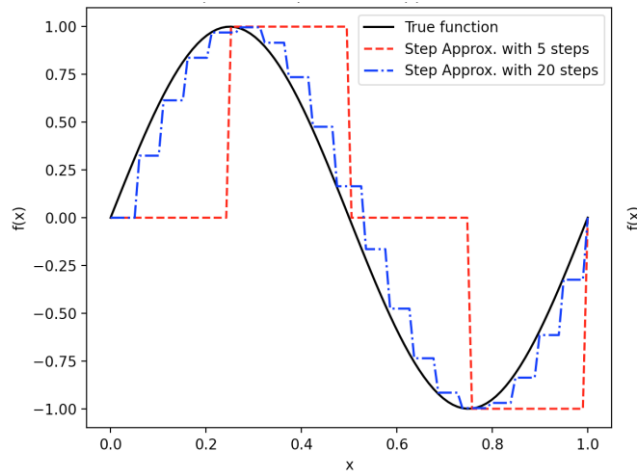
Neural networks are expressive enough to infer complex state estimates from raw pixels — but expressive ≠ works better ≠ safe (later slide).

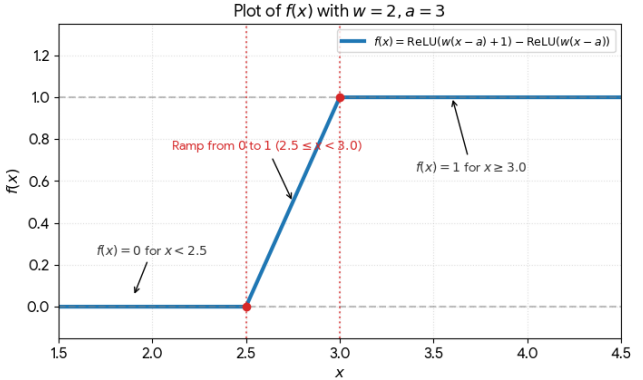
Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function." Mathematics of Control, Signals and Systems, 2(4), 303–314.

Hornik, K. (1991). "Approximation capabilities of multilayer feedforward networks." Neural Networks 4(2), 251–257.

Leshno, Lin, Pinkus, Schocken (1993). "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function." Neural Networks 6(6), 861–867.

https://www.youtube.com/watch?v=Ijqkc7OLenI





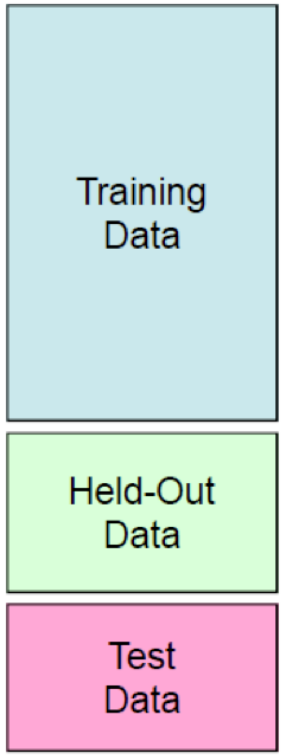
Best practices for training classifiers

Goal: obtain a classifier with **good generalization** or performance on never before seen data

1. Learn *parameters* on the ***training set***
2. Tune *hyperparameters* on the *held out* ***validation set***
3. Evaluate performance on the ***test set***

Always keep an eye on your training and validation losses!

Crucial: do not peek at the test set when iterating steps 1 and 2!



Under and overfitting

- **Underfitting:** training and test error are both *high*
  - Model does an equally poor job on the training and the test set
  - The model is too "simple" to represent the data or the model is not trained well
- **Overfitting:** Training error is *low* but test error is *high*
  - Model fits irrelevant characteristics (noise) in the training data
  - Model is too complex or amount of training data is insufficient

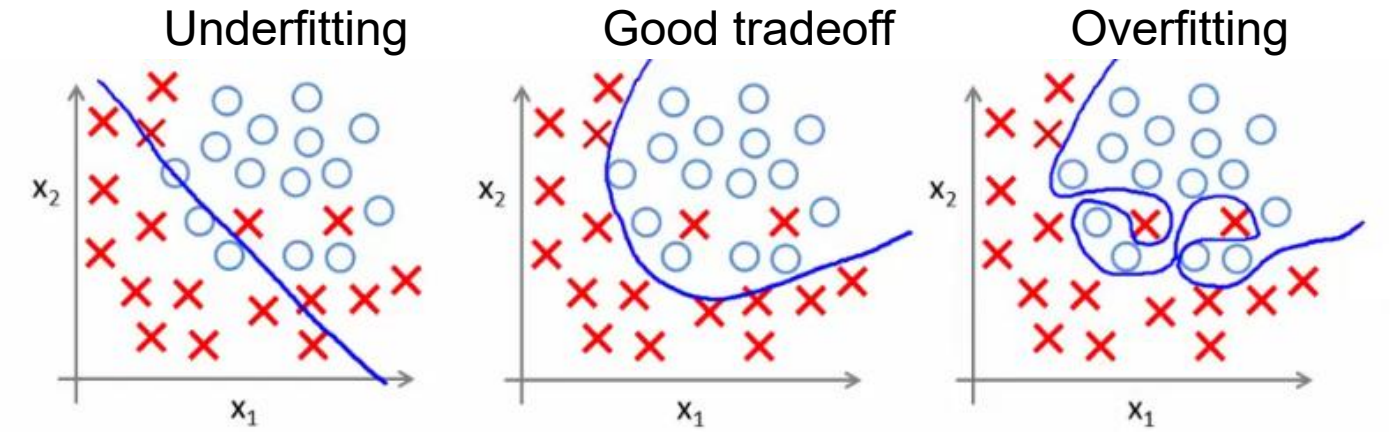
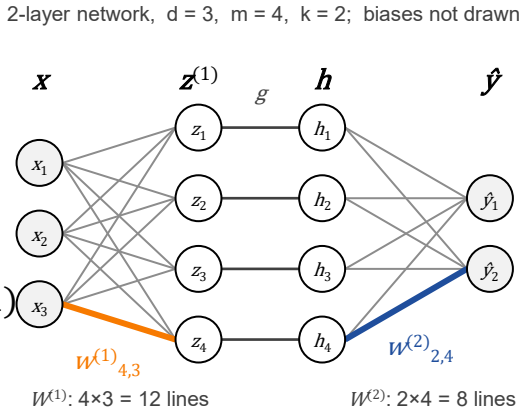


Figure source

Neural Network Forward Propagation



For a given input $\boldsymbol{x}_i$:

- Compute hidden layer **pre-activation**: $\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x}_i + \boldsymbol{b}^{(1)}$
- Apply activation: $\boldsymbol{h} = g(\boldsymbol{z}^{(1)})$.
- Compute output layer: $\hat{\boldsymbol{y}}_i = W^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$.

This results in a prediction $\hat{\boldsymbol{y}}_i$ for input $\boldsymbol{x}_i$, e.g.:

- For lane estimation: $\hat{\boldsymbol{y}} \in \mathbb{R}^{H \times W}$ if predicting per-pixel segmentation,
- For 6DOF pose: $\hat{\boldsymbol{y}} \in \mathbb{R}^6$, representing $(x, y, z, \alpha, \beta, \gamma)$ or other parameterization of rotation and translation.

Loss functions

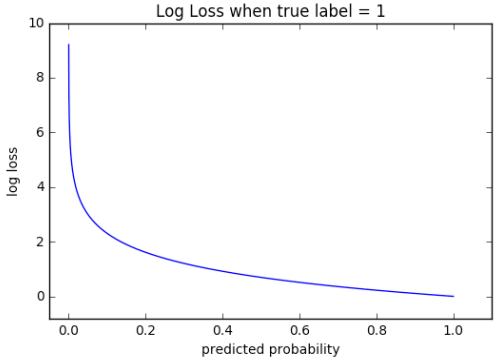
Loss function: A scalar function $L(W, \{\boldsymbol{y}_i, \boldsymbol{x}_i, \hat{\boldsymbol{y}}_i\})$ that measures how well a prediction $\hat{\boldsymbol{y}}_i$ matches the ground truth $\boldsymbol{y}_i$.

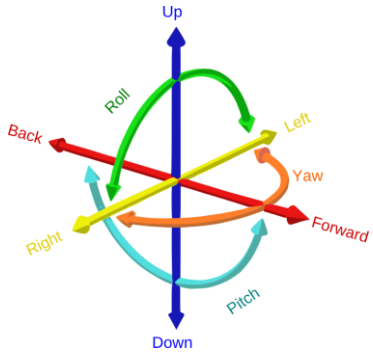
Requires knowledge of ground truth/labels for a batch of data: supervised learning

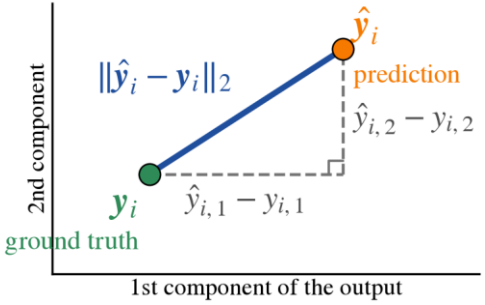
For lane segmentation (classification per pixel) cross-entropy loss for a binary label $y_i \in \{0,1\}$ and a predicted probability $\hat{y}_i \in (0,1)$

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \quad \text{(i indexes pixels)}$$

For pose regression, squared L2 distance: $L = \frac{1}{N}\sum_{i=1}^{N}\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i\|_2^2$







Pythagoras:

$$\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i\|_2^2 = (\hat{y}_{i,1} - y_{i,1})^2 + (\hat{y}_{i,2} - y_{i,2})^2$$

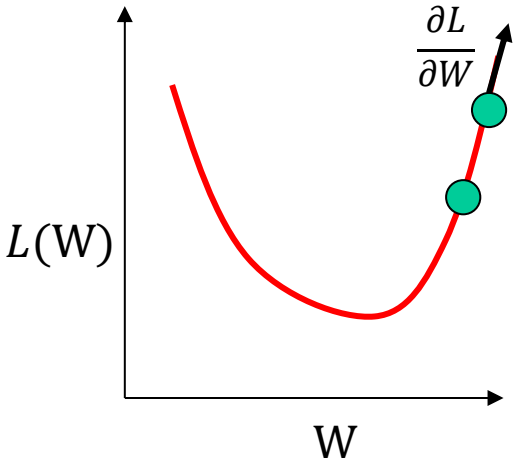
(pose: 6 components)

Forward pass and gradient descent

Evaluating loss involves computing $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i)$ forward pass of NN

Training: algorithm/process of minimizing loss $L(W)$ by changing the weights (W) and the biases (b) of the neural network (f)

E.g. gradient descent with back propagation



Gradient descent example

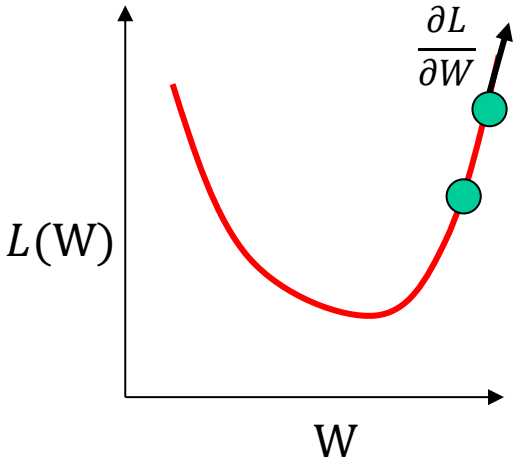
Core idea (picture: ball rolling downhill on a surface):

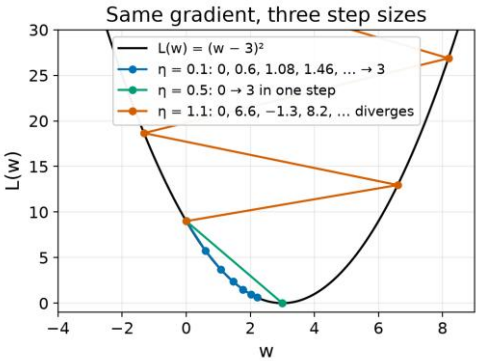
- Gradient $\nabla_{\boldsymbol{\theta}} L$ points in the direction of steepest increase of L.
- So to decrease loss, step in the opposite direction.

Update rule: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L$, η is learning rate (step size)

Example (1D): $L(w) = (w-3)^2$, so $\frac{dL}{dw} = 2(w-3)$.

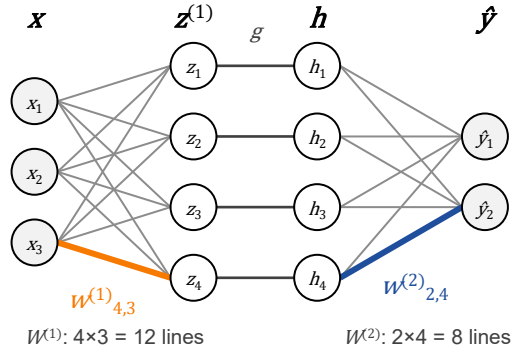
- Start $w_0 = 0, \eta = 0.1$:
- $w_1 = 0 - 0.1 * 2(0-3) = 0.6$
- $w_2 = 0.6 - 0.1 * 2(0.6-3) = 1.08$
- $w_3 = 1.08 - 0.1 * 2(1.08-3) = 1.464$
- $w_4 = 1.464 - 0.1 * 2(1.464-3) \approx 1.771$
- $w_5 \approx 2.017, w_6 \approx 2.214, w_7 \approx 2.371, w_8 \approx 2.497, \ldots$
- $w_{10} \approx 2.678, w_{20} \approx 2.965, w_{50} \approx 2.99996$ — the gap to 3 keeps shrinking
- loss $(w_t - 3)^2$: $9, 5.76, 3.69, 2.36, 1.51, 0.97, \ldots \to 0$ — smaller after every step
- Each step closes 20% of the remaining gap: $3 - w_{t+1} = 0.8\,(3 - w_t)$
- so $w_t = 3\,(1 - 0.8^t) \to 3$ without overshooting
- What happens for a larger η?





How to get gradients? Backpropagation!

2-layer network, d = 3, m = 4, k = 2; biases not drawn



Forward, one example: $\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},\ \boldsymbol{h} = g(\boldsymbol{z}^{(1)}),\ \hat{\boldsymbol{y}} = W^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$

Shapes (d inputs, m hidden units, k outputs; in the figure $d = 3, m = 4, k = 2$):
$\boldsymbol{x} \in \mathbb{R}^d,\ W^{(1)} \in \mathbb{R}^{m\times d},\ \boldsymbol{b}^{(1)}, \boldsymbol{z}^{(1)}, \boldsymbol{h} \in \mathbb{R}^m,\ W^{(2)} \in \mathbb{R}^{k\times m},\ \boldsymbol{b}^{(2)}, \hat{\boldsymbol{y}}, \boldsymbol{y} \in \mathbb{R}^k$

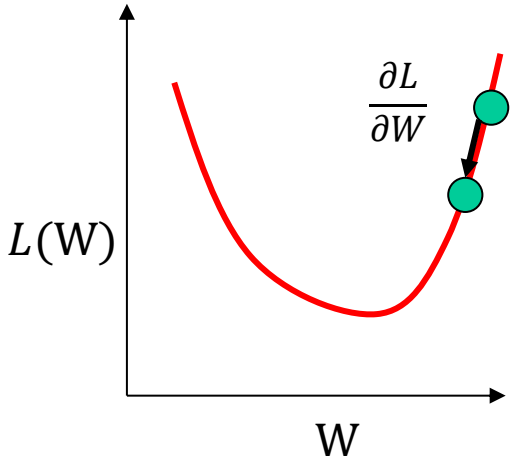
Loss (regression, one example): $L = \frac{1}{2}\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2 = \frac{1}{2}\sum_{j=1}^{k}(\hat{y}_j - y_j)^2$
($\hat{y}_j, y_j$: j-th output; for a batch average over the examples)

Computing gradients (backprop): we need $\frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial \boldsymbol{b}^{(2)}}, \frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial \boldsymbol{b}^{(1)}}$
all from the chain rule

- output: $\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T$, shape $1 \times k$: one entry per output, $\frac{\partial L}{\partial \hat{y}_j} = \hat{y}_j - y_j$
- $\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}} \in \mathbb{R}^{k\times m}$ — the same shape as $W^{(2)}$
  likewise, $\frac{\partial L}{\partial W^{(1)}} \in \mathbb{R}^{m\times d}$; $\frac{\partial L}{\partial \boldsymbol{b}^{(\ell)}}$ has the shape of $\boldsymbol{b}^{(\ell)}$

Intermediate results are reused: compute once, propagate backwards

Gradient descent update: $W^{(\ell)} \leftarrow W^{(\ell)} - \eta\frac{\partial L}{\partial W^{(\ell)}},\ \boldsymbol{b}^{(\ell)} \leftarrow \boldsymbol{b}^{(\ell)} - \eta\frac{\partial L}{\partial \boldsymbol{b}^{(\ell)}},\ \ell = 1,2$



Example ($k = 2$): $L = \frac{1}{2}\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2 = \frac{1}{2}[(\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2]$, solve $\frac{\partial L}{\partial \hat{\boldsymbol{y}}}$

Notation: $\hat{\boldsymbol{y}}, \boldsymbol{y} \in \mathbb{R}^2$ are vectors (bold); $\hat{y}_1, \hat{y}_2$ and y_1, y_2 are their entries (the two outputs of the network, not two examples).

$\frac{\partial L}{\partial \hat{\boldsymbol{y}}}$ has one entry per output, shape 1×2: $\left(\frac{\partial L}{\partial \hat{y}_1}, \frac{\partial L}{\partial \hat{y}_2}\right)$

$\frac{\partial L}{\partial \hat{y}_1} = \hat{y}_1 - y_1, \quad \frac{\partial L}{\partial \hat{y}_2} = \hat{y}_2 - y_2$ (the $\frac{1}{2}$ cancels the 2 from differentiating the square)

So $\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = [\hat{y}_1 - y_1, \hat{y}_2 - y_2] = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T$ — the prediction error

Exercise: $\hat{\boldsymbol{y}} = W^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$, $W^{(2)} \in \mathbb{R}^{2\times 4}$ $(m = 4)$, solve $\frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}}$

$$\hat{\boldsymbol{y}} = \begin{bmatrix}\hat{y}_1\\ \hat{y}_2\end{bmatrix} = \begin{bmatrix} w_{11}^{(2)}h_1 + w_{12}^{(2)}h_2 + w_{13}^{(2)}h_3 + w_{14}^{(2)}h_4 + b_1^{(2)} \\ w_{21}^{(2)}h_1 + w_{22}^{(2)}h_2 + w_{23}^{(2)}h_3 + w_{24}^{(2)}h_4 + b_2^{(2)} \end{bmatrix}$$

($w_{ij}^{(2)}$: entry (i,j) of $W^{(2)}$, the weight from h_j to $\hat{y}_i$)

Number of elements in $\frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}}$: $2 \times 2 \times 4 = 16$ (two outputs, 2×4 elements in $W^{(2)}$)

For simplicity, we can "vectorize" $W^{(2)}$ into a vector with 8 elements (row by row). Then we calculate the 2×8 Jacobian matrix:

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \mathrm{vec}(W^{(2)})} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial w_{11}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{12}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{13}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{14}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{21}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{22}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{23}^{(2)}} & \frac{\partial \hat{y}_1}{\partial w_{24}^{(2)}} \\ \frac{\partial \hat{y}_2}{\partial w_{11}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{12}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{13}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{14}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{21}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{22}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{23}^{(2)}} & \frac{\partial \hat{y}_2}{\partial w_{24}^{(2)}} \end{bmatrix}$$

$$= \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$

(row i: $\frac{\partial \hat{y}_i}{\partial w_{ij}^{(2)}} = h_j$, every other entry is 0)

Abusing the notation a little bit for simplicity — we can call this $\frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}}$

Backprop worked example (3 of 4): the output-layer weight gradient

Shape of a Jacobian: a function with m inputs and k outputs ($\boldsymbol{f}: \mathbb{R}^m \to \mathbb{R}^k$) has a $k \times m$ Jacobian $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$: one row per output, one column per input ($k \cdot m$ entries).

Here: L has 2 inputs ($\hat{y}_1, \hat{y}_2$), 1 output → 1×2; $\hat{\boldsymbol{y}}$ has 8 inputs (the entries of $W^{(2)}$), 2 outputs → 2×8; product $(1 \times 2)(2 \times 8) = 1 \times 8$.

Shape 1x8

$$\text{Finally, } \frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}} = [\hat{y}_1 - y_1, \hat{y}_2 - y_2] \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$

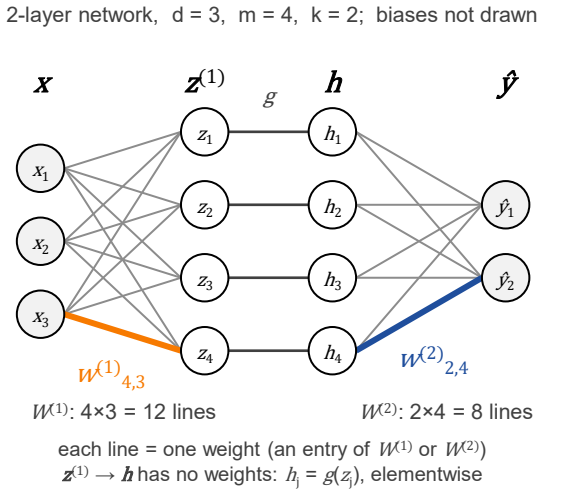
Shape 1x2

Shape 2x8

Backprop worked example (4 of 4): the first-layer chain

$$\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$
$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}^{(1)})$$
$$\hat{\boldsymbol{y}} = W^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$$

Now how about $\frac{\partial L}{\partial W^{(1)}}$? How many elements in this gradient?



$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}^{(1)}}\frac{\partial \boldsymbol{z}^{(1)}}{\partial W^{(1)}}$$

$\boldsymbol{x} \in \mathbb{R}^3,\ W^{(1)} \in \mathbb{R}^{4\times 3},\ \boldsymbol{b}^{(1)} \in \mathbb{R}^4$
$\boldsymbol{z}^{(1)} \in \mathbb{R}^4,\ \boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}^{(1)}) \in \mathbb{R}^4$
$W^{(2)} \in \mathbb{R}^{2\times 4},\ \boldsymbol{b}^{(2)} \in \mathbb{R}^2,\ \hat{\boldsymbol{y}}, \boldsymbol{y} \in \mathbb{R}^2,\ L \in \mathbb{R}$

Exercises:

1. What is the shape of each Jacobian matrix? 1x2, 2x4, 4x4, 4x(4x3)
2. What is $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}^{(1)}}$?

A Diagonal matrix with 0 or 1 on its diagonal, depending on the value of $\boldsymbol{z^{(1)}}$ during forward propagation

Worked example: $\frac{\partial L}{\partial W^{(1)}}$ in closed form (1 of 2) — the four Jacobians

Same network ($d = 3$, $m = 4$, $k = 2$), one example, $L = \frac{1}{2}\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$, $\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}^{(1)})$. Chain rule:

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}^{(1)}}\frac{\partial \boldsymbol{z}^{(1)}}{\partial W^{(1)}}$$ — write down each factor with its shape:

1. $\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = [\hat{y}_1 - y_1, \hat{y}_2 - y_2]$ (shape 1×2)

2. $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}} = W^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} & w_{14}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} & w_{23}^{(2)} & w_{24}^{(2)} \end{bmatrix}$ (shape 2×4), because $\hat{y}_i = \sum_{j=1}^{4} w_{ij}^{(2)} h_j + b_i^{(2)}$ gives $\frac{\partial \hat{y}_i}{\partial h_j} = w_{ij}^{(2)}$

3. $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}^{(1)}} = \mathrm{diag}(s_1, s_2, s_3, s_4)$ (4×4; slide 30), $s_j = 1$ if $z_j^{(1)} > 0$ else 0 — $h_j = \mathrm{ReLU}(z_j^{(1)})$ depends on $z_j^{(1)}$ only

4. $\frac{\partial \boldsymbol{z}^{(1)}}{\partial \mathrm{vec}(W^{(1)})}$: $z_j^{(1)} = w_{j1}^{(1)}x_1 + w_{j2}^{(1)}x_2 + w_{j3}^{(1)}x_3 + b_j^{(1)}$, so $\frac{\partial z_j^{(1)}}{\partial w_{jn}^{(1)}} = x_n$, and row j of $W^{(1)}$ feeds no other $z_j^{(1)}$:

$$\frac{\partial \boldsymbol{z}^{(1)}}{\partial \mathrm{vec}(W^{(1)})} = \begin{bmatrix} x_1 & x_2 & x_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & x_2 & x_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x_1 & x_2 & x_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_1 & x_2 & x_3 \end{bmatrix}$$ (4×12; $W^{(1)}$ vectorized row by row)

Shape check: $(1 \times 2)(2 \times 4)(4 \times 4)(4 \times 12) = 1 \times 12$ → reshaped to 4×3, the shape of $W^{(1)}$: 12 numbers, one per weight.

Worked example: $\frac{\partial L}{\partial W^{(1)}}$ in closed form (2 of 2) — multiply, then reshape

Step 1 — through the output layer $(1 \times 2)(2 \times 4) = 1 \times 4$ — each output error, weighted by the outgoing weights of h_j:

$$\frac{\partial L}{\partial \boldsymbol{h}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}} = [\hat{y}_1 - y_1, \hat{y}_2 - y_2]\, W^{(2)} = [u_1, u_2, u_3, u_4], \quad u_j = (\hat{y}_1 - y_1)\, w_{1j}^{(2)} + (\hat{y}_2 - y_2)\, w_{2j}^{(2)}$$

Step 2 — through the ReLU $(1 \times 4)(4 \times 4) = 1 \times 4$ — an inactive unit ($s_j = 0$) passes no gradient:

$$\frac{\partial L}{\partial \boldsymbol{z}^{(1)}} = [u_1, u_2, u_3, u_4]\, \mathrm{diag}(s_1, s_2, s_3, s_4) = [s_1 u_1, s_2 u_2, s_3 u_3, s_4 u_4] =: [\delta_1, \delta_2, \delta_3, \delta_4]$$

Step 3 — through the first layer $(1 \times 4)(4 \times 12) = 1 \times 12$:

$$\frac{\partial L}{\partial \mathrm{vec}(W^{(1)})} = [\delta_1, \delta_2, \delta_3, \delta_4] \cdot (4 \times 12 \text{ matrix on the previous slide})$$
$$= [\delta_1 x_1, \delta_1 x_2, \delta_1 x_3, \delta_2 x_1, \delta_2 x_2, \delta_2 x_3, \delta_3 x_1, \delta_3 x_2, \delta_3 x_3, \delta_4 x_1, \delta_4 x_2, \delta_4 x_3]$$

Step 4 — reshape row by row into the shape of $W^{(1)}$ (4×3):

$$\frac{\partial L}{\partial W^{(1)}} = \begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & \delta_1 x_3 \\ \delta_2 x_1 & \delta_2 x_2 & \delta_2 x_3 \\ \delta_3 x_1 & \delta_3 x_2 & \delta_3 x_3 \\ \delta_4 x_1 & \delta_4 x_2 & \delta_4 x_3 \end{bmatrix} = \boldsymbol{\delta}^{(1)}\, \boldsymbol{x}^T \text{ with } \boldsymbol{\delta}^{(1)} = [\delta_1, \delta_2, \delta_3, \delta_4]^T$$

i.e. $\frac{\partial L}{\partial w_{jn}^{(1)}} = \delta_j\, x_n = s_j[(\hat{y}_1 - y_1)w_{1j}^{(2)} + (\hat{y}_2 - y_2)w_{2j}^{(2)}]\, x_n$

Closed form: $\frac{\partial L}{\partial W^{(1)}} = ((W^{(2)})^T(\hat{\boldsymbol{y}} - \boldsymbol{y}) \odot 1[\boldsymbol{z}^{(1)} > 0])\, \boldsymbol{x}^T$; likewise $\frac{\partial L}{\partial \boldsymbol{b}^{(1)}} = \boldsymbol{\delta}^{(1)}$ (since $\frac{\partial z_j^{(1)}}{\partial b_j^{(1)}} = 1$).