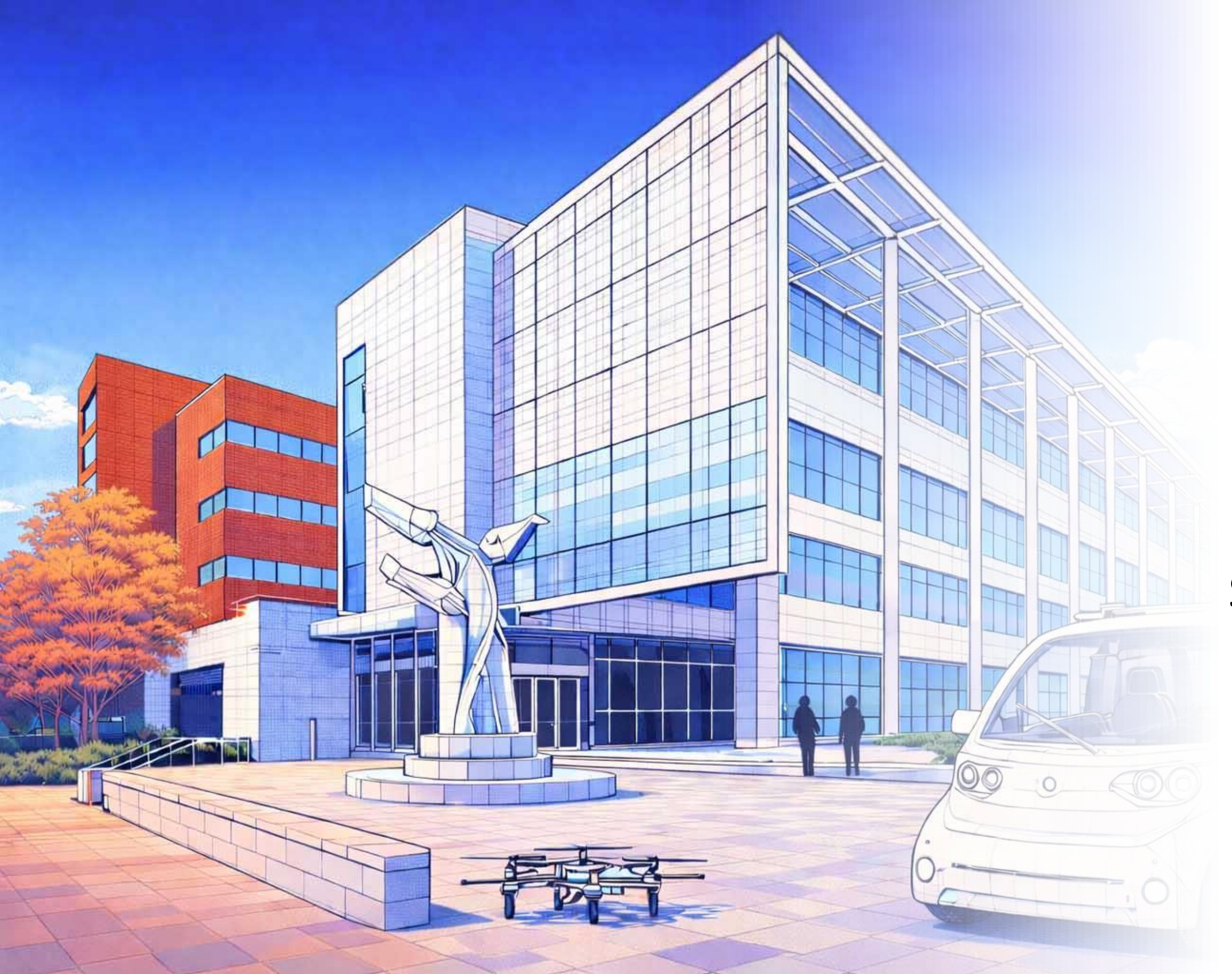
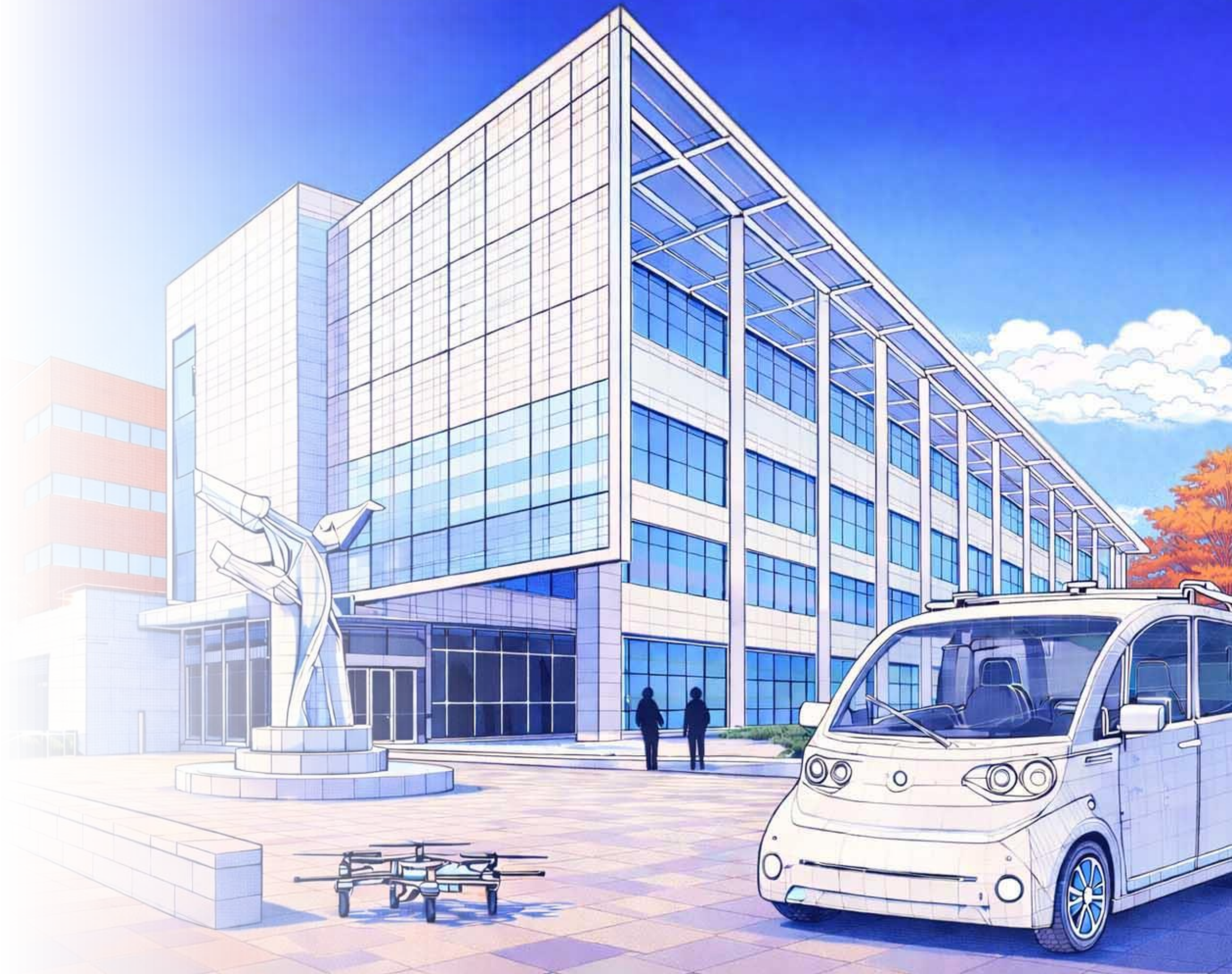


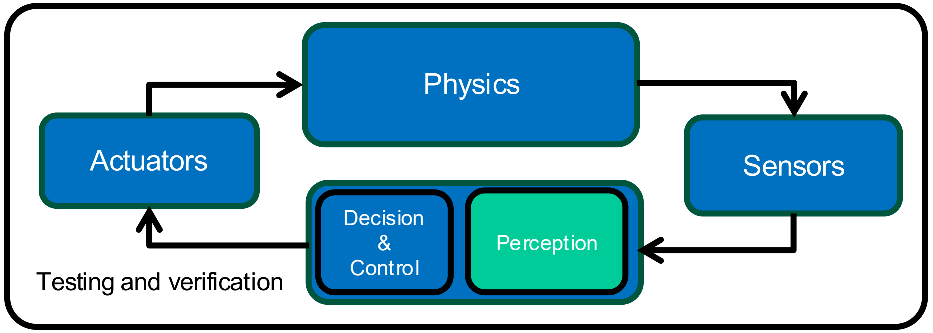
ECE 484 Lecture 4 Perception: Neural Networks

Sayan Mitra



Perception in an autonomous system

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

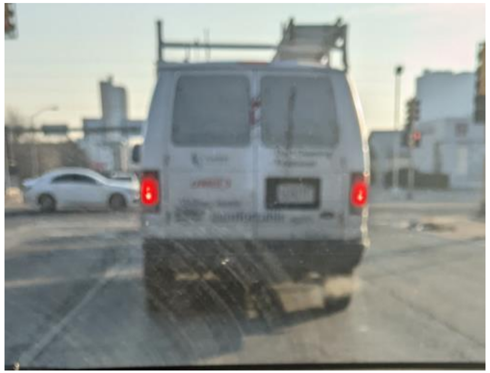


Examples:

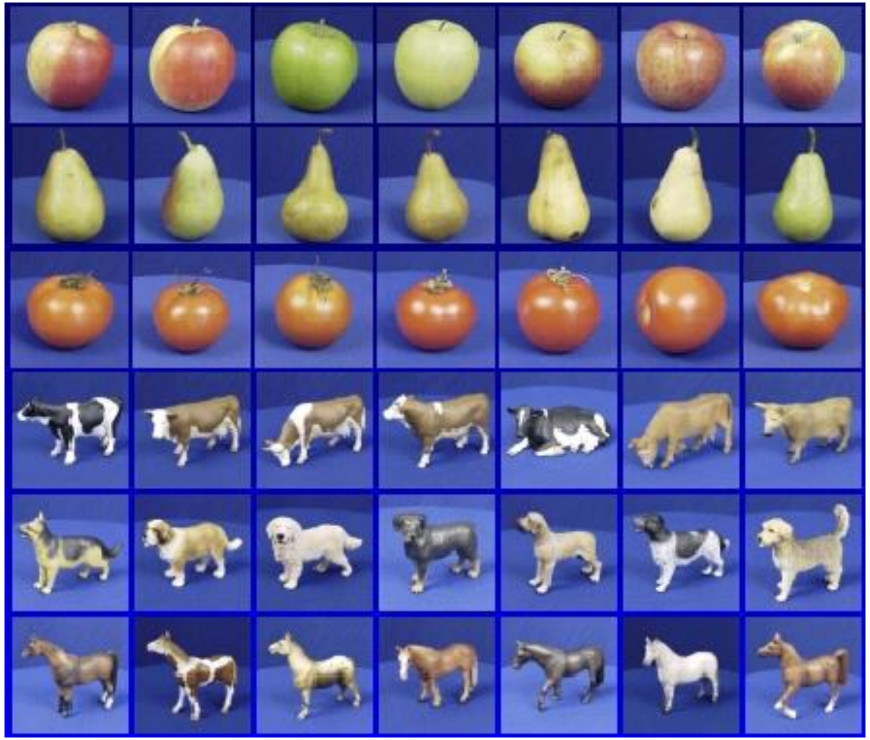
- **Type** or **Class** of lead vehicle, traffic sign
- **Pose** position of ego on map, relative to lanes, relative to lead vehicle, attitude of drone relative to marker
- **Speed,** angular speed of ego and others
- Acceleration, intention of the pedestrian

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

$$\underset{\text{output}}{\underset{\uparrow}{y}} = \underset{\substack{\text{prediction}\\ \text{function}}}{\underset{\uparrow}{f}}(\underset{\substack{\text{feature}\\ \text{representation}}}{\underset{\nwarrow}{\mathbf{x}}})$$

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: tune (hyper)parameters in f, learning rate

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

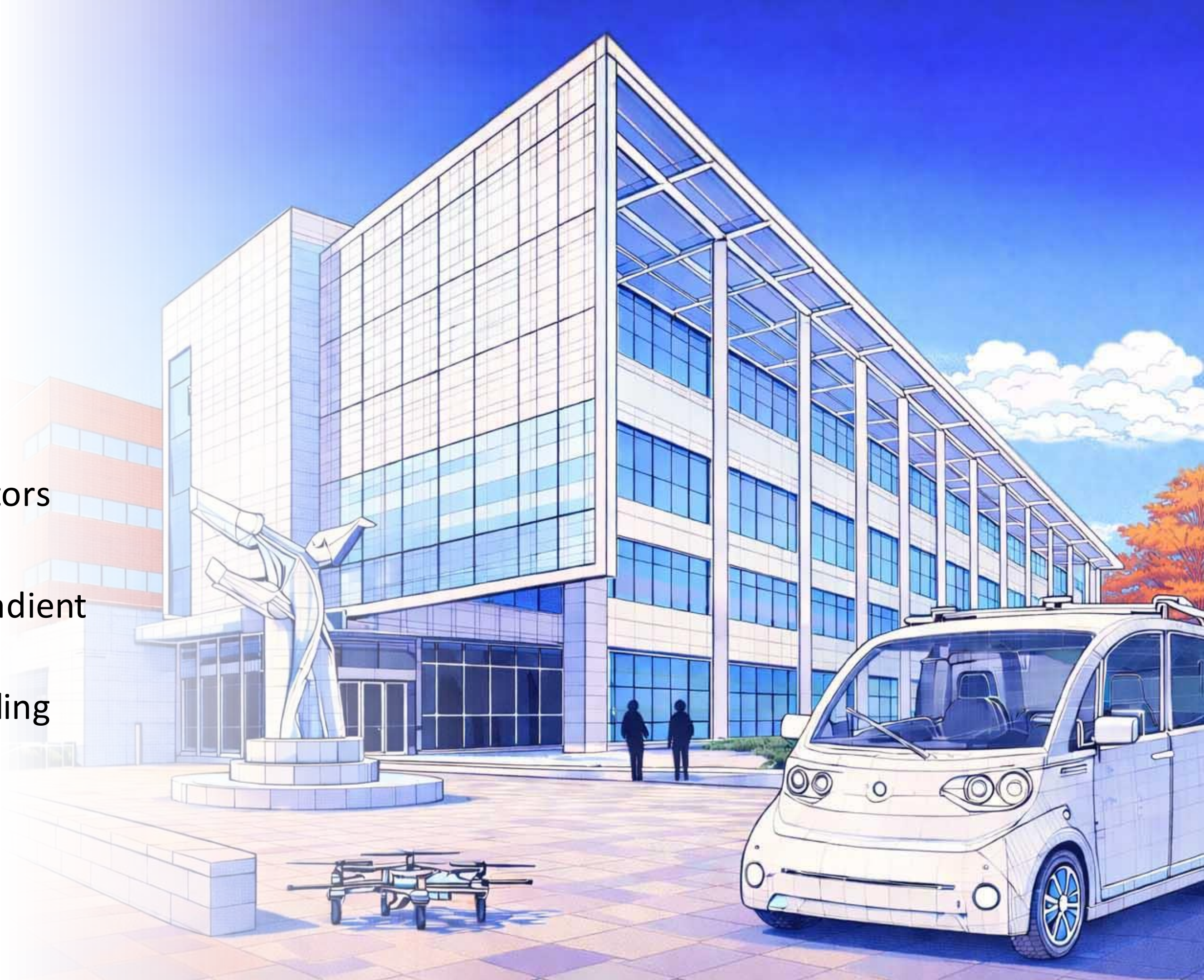
Outline

Linear classifiers

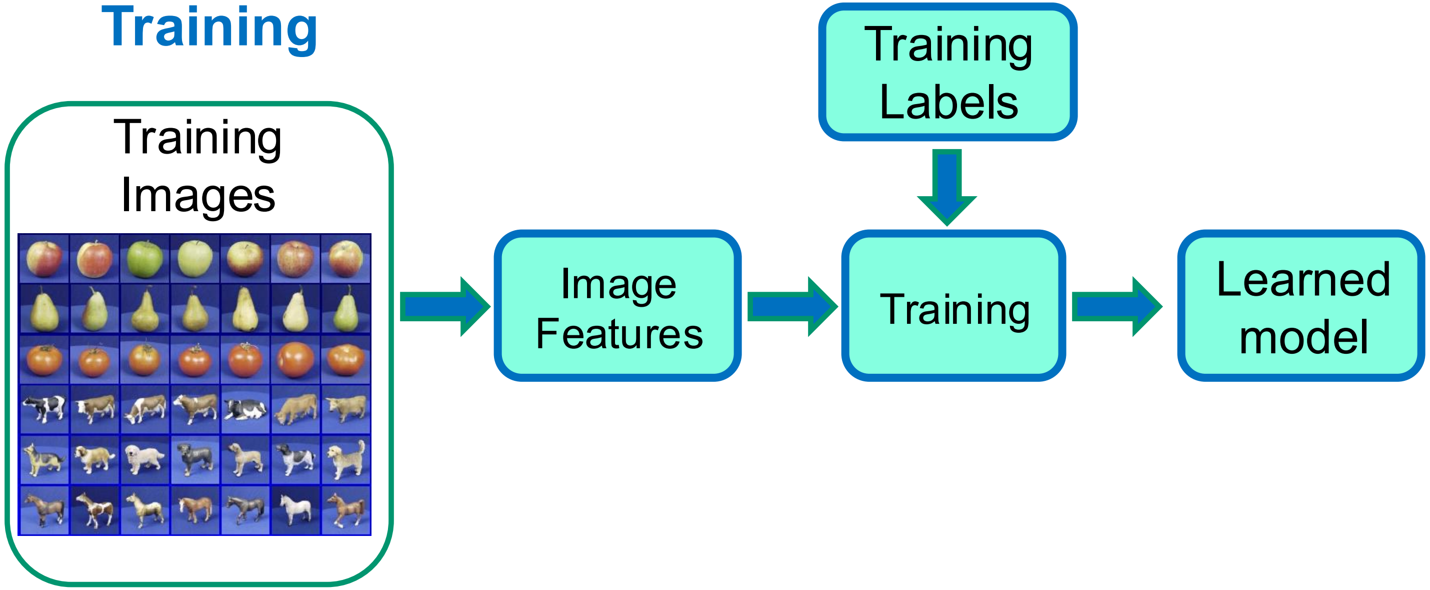
Neural networks

- Universal approximators
- Forward pass
- Backpropagation; Gradient descent
- Exploding and imploding gradients

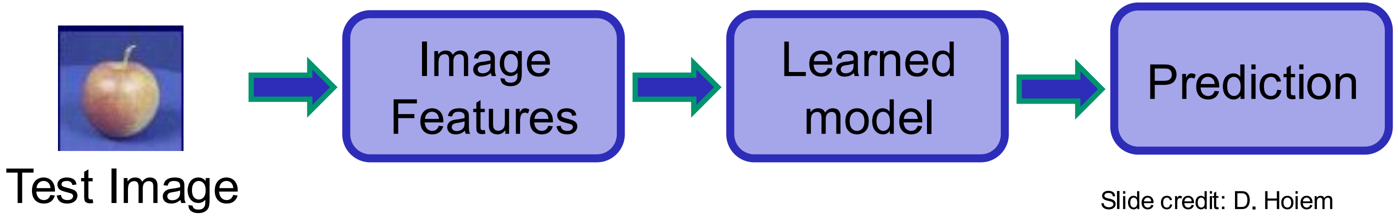
Best practices



Training



Testing



Slide credit: D. Hoiem

Linear classifiers

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

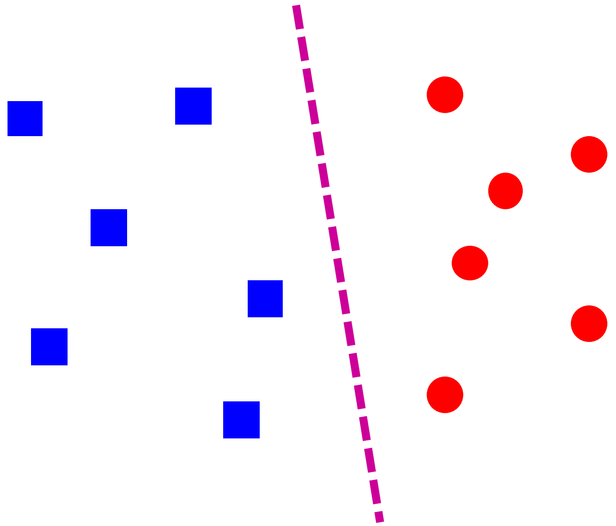
A constant weight vector $\mathrm{w} \in \mathbb{R}^d$ and a scalar bias $b \in \mathbb{R}$ define a binary classifier

$$f(\boldsymbol{x}) = \operatorname{sgn}(w.\boldsymbol{x} + \boldsymbol{b})$$

The classification task is to find the weight and the bias (w, b) that define a linear separator for the given data set

The separators are also called decision boundaries

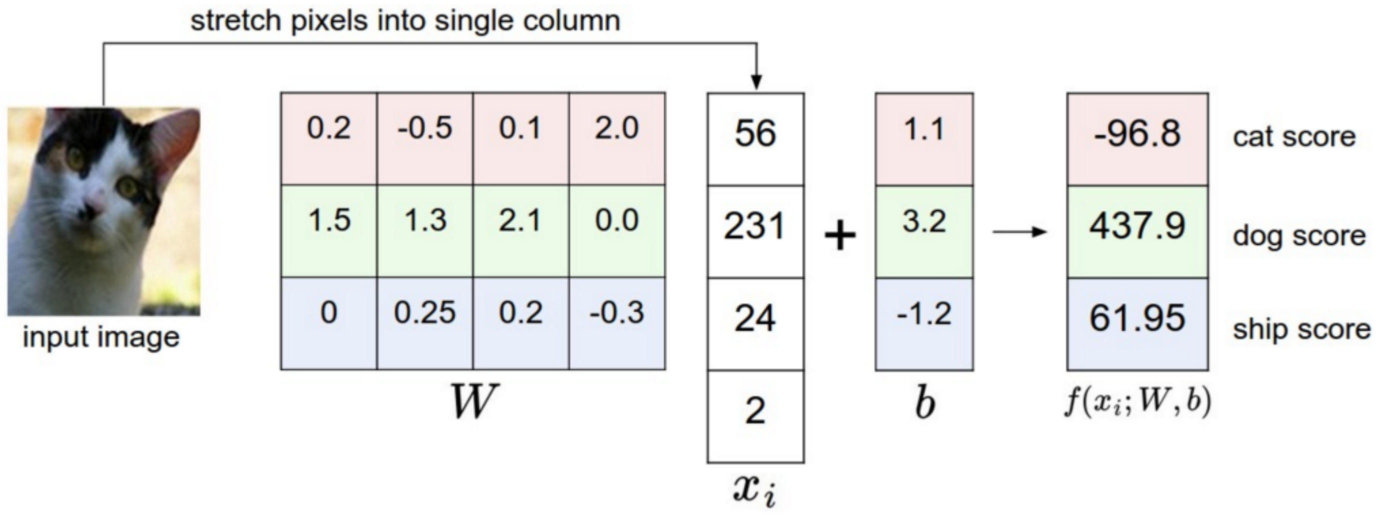
SVM classifiers try to find a linear separator that maximizes the distance of the separator to the points

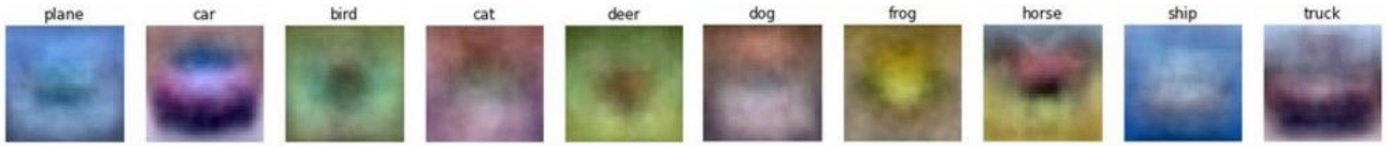


Read support vector machines (SVM) https://greitemann.dev/svm-demo

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

Visualizing linear classifiers with many classes

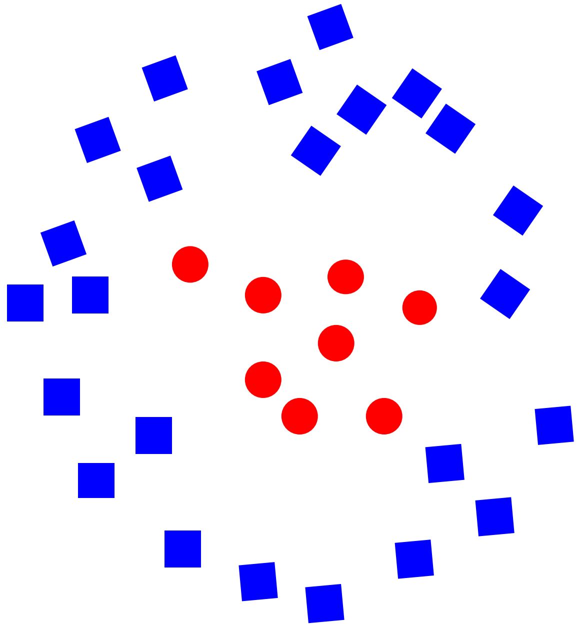




Source: Andrej Karpathy, http://cs231n.github.io/linear-classify/

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 classification example): $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$
- Limitations: The data may not be linearly separable
- Linear separators decision boundaries may not capture nonlinear relationships between classes



Nonlinearity via Neural Networks

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

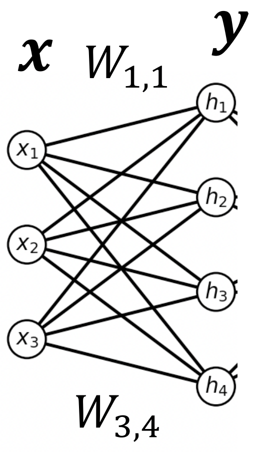
A single layer performs a linear transformation followed by a nonlinear activation

$w.\boldsymbol{x} + b \in \mathbb{R}$

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

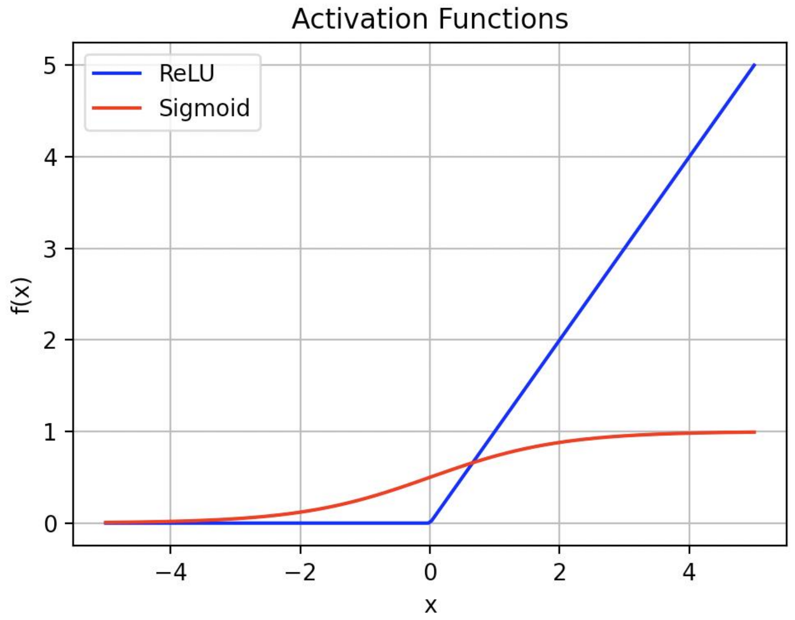
$\boldsymbol{y} = g(W\boldsymbol{x} + \boldsymbol{b})$ nonlinear activation $g \in \mathbb{R}^m \rightarrow \mathbb{R}^m$

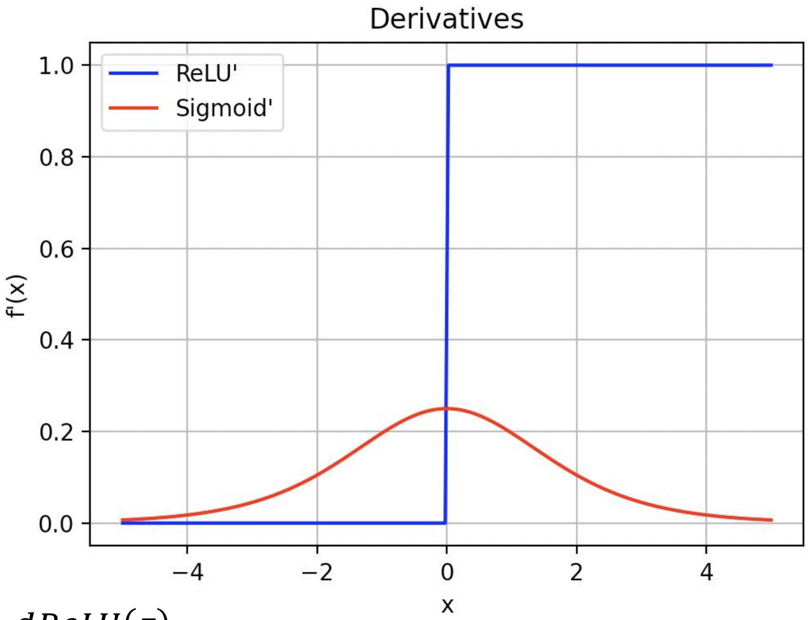
g: activation function e.g. $ReLU(z) = \max(z, 0)$, $sigmoid(z) = \frac{e^z}{1+e^z}$



1-Layer Perceptron

Activation functions and their derivatives





$$ReLU(z) = \max(z, 0)$$

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

$$\frac{dReLU(z)}{dz} = \{0,1\}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

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 $\boldsymbol{x} \in \mathbb{R}^d$

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

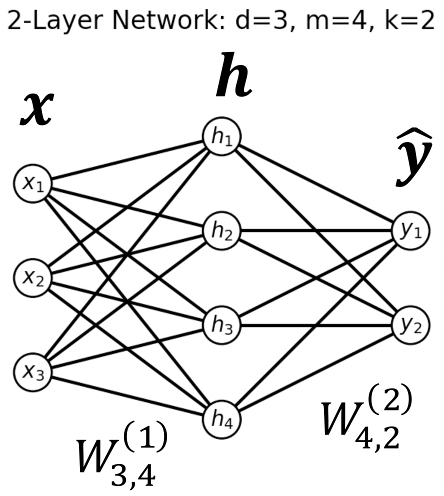
$$\hat{\boldsymbol{y}} = W^{(2)}g\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}$$

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

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

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

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



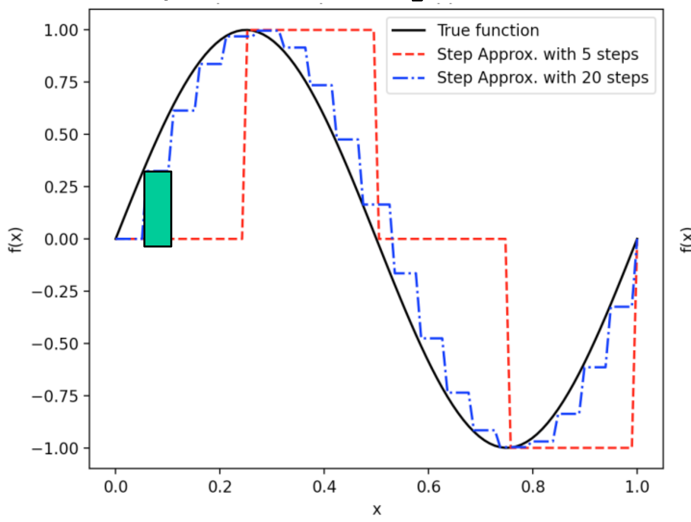
Multi-Layer Perceptron (MLP)

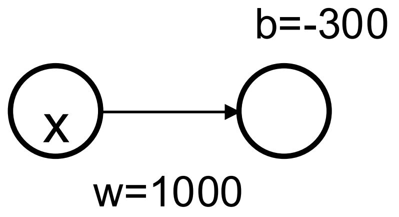
Universal Approximation Theorem [Cybenko, G. 1989]

Any continuous function f on a compact domain can be approximated to arbitrary precision with a sufficiently large (but finite) single-hidden-layer feedforward network with a suitable activation function.

- f can be approximated arbitrarily by a sum of towers
- **Exercise.** A tower function can be approximated by a network with 1 layer (2 ReLUs ~= _|)
- Sum of towers can be created by adding more elements in the hidden layer

Neural networks are expressive enough to infer complex state estimates from raw pixels



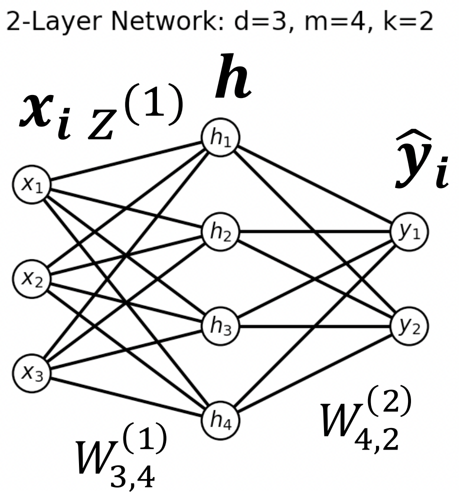


y=$ReLU$(1000x-300)

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

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

Neural Network Forward Propagation



For a given input $\boldsymbol{x_i}$:

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

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

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

Backpropagation and Gradient-Based Training

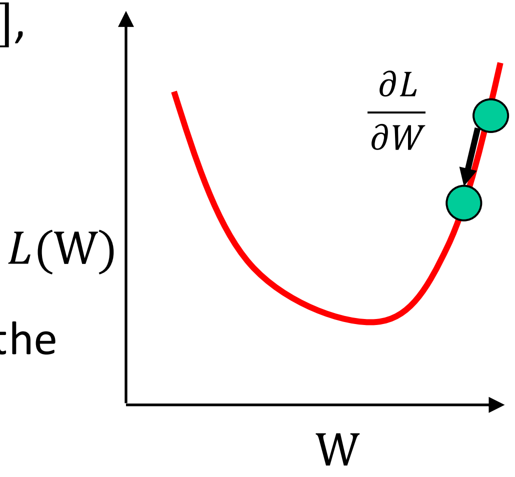
Loss function: A scalar function $L(W,\{y_i,x_i,\hat{y}_i\})$ that measures how well a prediction $\hat{\boldsymbol{y}}_i$ matches the ground truth $\boldsymbol{y_i}$.

Requires knowledge of ground truth/labels: supervised learning

For lane segmentation (classification per pixel) cross-entropy loss suitable when predictions are probabilities $y_i \in \{0,1\}$ $\hat{y}_{\boldsymbol{i}} \in [0,1]$

$$L(W,\{y_i,x_i,\hat{y}_i\}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)],$$

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



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

Backpropagation Background: Vector Calculus refresher

Differentiating scalars, vectors, and matrices:

$\boldsymbol{y} = \left(\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(n)}\right)^T;\ \frac{\partial}{\partial \boldsymbol{x}} = (\frac{\partial}{\partial \boldsymbol{x}^{(1)}}, \frac{\partial}{\partial \boldsymbol{x}^{(2)}}, \frac{\partial}{\partial \boldsymbol{x}^{(3)}})$ Gradient operator ∇

$\frac{\partial y}{\partial x}, \frac{\partial \boldsymbol{y}}{\partial x} = \begin{bmatrix} \frac{\partial \boldsymbol{y}^{(1)}}{\partial x} \\ \dots \\ \frac{\partial \boldsymbol{y}^{(n)}}{\partial x} \end{bmatrix}, \frac{\partial Y}{\partial x} = \begin{bmatrix} \frac{\partial Y^{(1,1)}}{\partial x} & \dots \\ \dots & \frac{\partial Y^{(i,j)}}{\partial x} \end{bmatrix}$ Element-wise differentiation of vectors and matrices

$\frac{\partial y}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}} . y = \left(\frac{\partial}{\partial \boldsymbol{x}^{(1)}}, \frac{\partial}{\partial \boldsymbol{x}^{(2)}}, \frac{\partial}{\partial \boldsymbol{x}^{(3)}}\right) . y$ Gradient of y; row vector of partials

$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}} \quad . \boldsymbol{y} = \left(\frac{\partial}{\partial \boldsymbol{x}^{(1)}}, \frac{\partial}{\partial \boldsymbol{x}^{(2)}}, \frac{\partial}{\partial \boldsymbol{x}^{(3)}}\right) . \begin{bmatrix} \boldsymbol{y}^{(1)} \\ \boldsymbol{y}^{(2)} \\ \boldsymbol{y}^{(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial \boldsymbol{y}^{(1)}}{\partial \boldsymbol{x}^{(1)}} & \frac{\partial \boldsymbol{y}^{(1)}}{\partial \boldsymbol{x}^{(2)}} & \frac{\partial \boldsymbol{y}^{(1)}}{\partial \boldsymbol{x}^{(3)}} \\ & & \\ & & \frac{\partial \boldsymbol{y}^{(3)}}{\partial \boldsymbol{x}^{(3)}} \end{bmatrix}$ Jacobian; . is Kronecker product (also ⨀)

$\frac{\partial Y}{\partial \boldsymbol{x}} = \left(\frac{\partial}{\partial \boldsymbol{x}^{(1)}}, \frac{\partial}{\partial \boldsymbol{x}^{(2)}}, \frac{\partial}{\partial \boldsymbol{x}^{(3)}}\right) . Y = \left[\frac{\partial}{\partial \boldsymbol{x}^{(1)}} Y \ \frac{\partial}{\partial \boldsymbol{x}^{(2)}} Y \ \frac{\partial}{\partial \boldsymbol{x}^{(3)}} Y\right]$

Backpropagation Background: Vector Calculus notations

$$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot y$$

$$\frac{\partial \boldsymbol{y}}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \boldsymbol{y} = \left[\begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \boldsymbol{y}^{(1)} \quad \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \boldsymbol{y}^{(2)} \quad \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \boldsymbol{y}^{(3)} \right]$$

$$\frac{\partial Y}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot Y = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \begin{bmatrix} Y^{(1,1)} & Y^{(1,2)} \\ Y^{(2,1)} & Y^{(2,2)} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \frac{\partial Y^{(1,1)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(1,1)}}{\partial X^{(1,2)}} \\ \frac{\partial Y^{(1,1)}}{\partial X^{(2,1)}} & \frac{\partial Y^{(1,1)}}{\partial X^{(2,2)}} \end{bmatrix} & \begin{bmatrix} \frac{\partial Y^{(1,2)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(1,2)}}{\partial X^{(1,2)}} \\ \frac{\partial Y^{(1,2)}}{\partial X^{(2,1)}} & \frac{\partial Y^{(1,2)}}{\partial X^{(2,2)}} \end{bmatrix} \\ \cdots & \begin{bmatrix} \frac{\partial Y^{(2,2)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(2,2)}}{\partial X^{(1,2)}} \\ \frac{\partial Y^{(2,2)}}{\partial X^{(2,1)}} & \frac{\partial Y^{(2,2)}}{\partial X^{(2,2)}} \end{bmatrix} \end{bmatrix}$$

Backpropagation and Gradient-Based Trainin

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

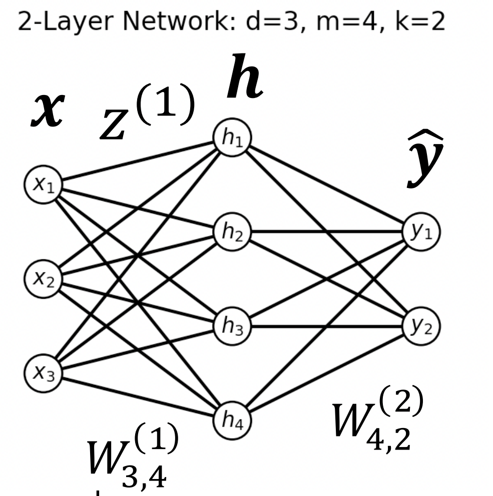
For pose regression: $L = -\frac{1}{N}\sum_{i=1}^{N} |\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i|_2^2$

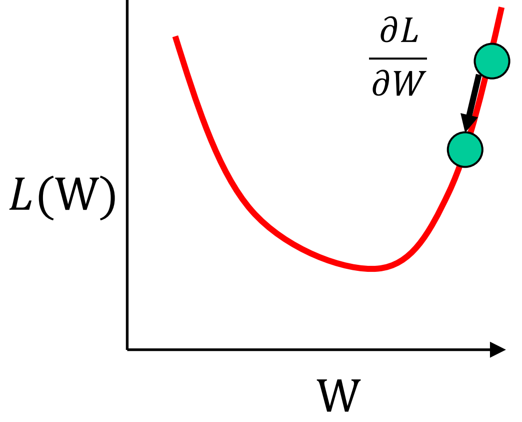
Computing Gradients (Backprop):

- We compute $\frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial b^{(2)}}, \frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial b^{(1)}}$ using chain rule
- For example: $\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}} = \left(\frac{\partial L}{\partial \hat{\boldsymbol{y}}}\right) h^{\top}$

Gradient Descent Update: With learning rate η

$$W^{(l)} := W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}, \quad \boldsymbol{b}^{(l)} := \boldsymbol{b}^{(l)} - \eta \frac{\partial L}{\partial \boldsymbol{b}^{(l)}}$$





$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 \\ w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial}{\partial W^{(1,1)}} & \frac{\partial}{\partial W^{(1,2)}} & \frac{\partial}{\partial W^{(1,3)}} \\ \frac{\partial}{\partial W^{(2,1)}} & \frac{\partial}{\partial W^{(2,2)}} & \frac{\partial}{\partial W^{(2,3)}} \end{bmatrix} \odot y^T$$

$$\left[\begin{bmatrix} \frac{\partial}{\partial W^{(1,1)}} & \frac{\partial}{\partial W^{(1,2)}} & \frac{\partial}{\partial W^{(1,3)}} \\ \frac{\partial}{\partial W^{(2,1)}} & \frac{\partial}{\partial W^{(2,2)}} & \frac{\partial}{\partial W^{(2,3)}} \end{bmatrix} w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 \begin{bmatrix} \frac{\partial}{\partial W^{(1,1)}} & \frac{\partial}{\partial W^{(1,2)}} & \frac{\partial}{\partial W^{(1,3)}} \\ \frac{\partial}{\partial W^{(2,1)}} & \frac{\partial}{\partial W^{(2,2)}} & \frac{\partial}{\partial W^{(2,3)}} \end{bmatrix} w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2\right].$$

$$\left[\begin{bmatrix} h_1 & h_2 & h_3 \\ 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 0 & 0 & 0 \\ h_1 & h_2 & h_3 \end{bmatrix}\right] = I \odot h^T$$

Vanishing gradient problem

Consider MLP with:

- Input $x \in \mathbb{R}^d$
- 3 hidden layers, each with **sigmoid**.
- 1 output $\hat{y} \in \mathbb{R}$ for a regression or binary classification.

That is

1. $z^{(1)} = W^{(1)}x + b^{(1)} \in \mathbb{R}^m$

2. $a^{(1)} = \sigma(z^{(1)}) \quad \in \mathbb{R}^m$

3. $z^{(i)} = W^{(i)}a^{(i-1)} + b^{(i)} \in \mathbb{R}^m, i = 2,3,4$

4. $a^{(i)} = \sigma\left(z^{(i)}\right) \quad \in \mathbb{R}^m, i = 2,3$

5. $\hat{y} = \sigma(z^{(4)}) \ \in \mathbb{R}^m$

Loss $\mathrm{L} = \frac{1}{2}(\hat{y} - y)^2$

As the number of layers increase gradient can vanish

$$\frac{\partial L}{\partial z^{(4)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z^{(4)}} = (\hat{y}-y)\frac{\partial \hat{y}}{\partial z^{(4)}} = (\hat{y}-y)\hat{y}(1-\hat{y})$$

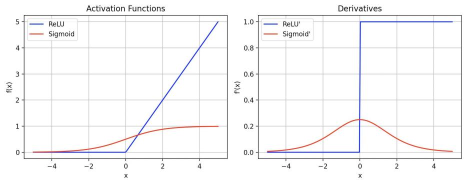
$$\frac{\partial L}{\partial a^{(3)}} = \frac{\partial L}{\partial z^{(4)}}\frac{\partial z^{(4)}}{\partial a^{(3)}} = (\hat{y}-y)\hat{y}(1-\hat{y}).W^{(4)}$$

$$\frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial a^{(3)}}\frac{\partial a^{(3)}}{\partial z^{(3)}} = (\hat{y}-y)\hat{y}(1-\hat{y}).W^{(4)} \odot \sigma'(z^{(3)})$$

$$\frac{\partial L}{\partial z^{(\ell)}} = \frac{\partial L}{\partial z^{(\ell+1)}} W^{\ell+1}\sigma'\left(z^{(\ell)}\right)$$

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = \sigma\left(z^{(1)}\right)$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$a^{(2)} = \sigma\left(z^{(2)}\right)$$
$$z^{(3)} = W^{(3)}a^{(2)} + b^{(3)}$$
$$a^{(3)} = \sigma\left(z^{(3)}\right)$$
$$z^{(4)} = W^{(4)}a^{(3)} + b^{(4)}$$
$$\hat{y} = \sigma(z^{(4)})$$
$$L = \frac{1}{2}(\hat{y}-y)^2$$

If each $\sigma'\left(z^{(\ell)}\right) \leq 0.25$ the gradient vanishes



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

Exploding gradient problem

Consider MLP with:

- Input $x \in \mathbb{R}^d$
- 3 hidden layers, each with **RELU**.
- 1 output $\hat{y} \in \mathbb{R}$ for a regression or binary classification.

That is

1. $z^{(1)} = W^{(1)}x + b^{(1)} \in \mathbb{R}^m = \alpha I x$ for simplicity $b^{(i)} = 0$ $W^{(i)} = \alpha I$
2. $a^{(1)} = ReLU\left(z^{(1)}\right) = \alpha x$ if $a^{(1)}$ is positive in each component
3. $z^{(i)} = \alpha I a^{(i-1)} = \alpha^i x, i = 2,3,4$
4. $a^{(i)} = ReLU\left(z^{(i)}\right) = \alpha^i x, i = 2,3$
5. $\hat{y} = ReLU\left(z^{(4)}\right) = \alpha^4 x$

Loss $\mathrm{L} = \frac{1}{2}(\hat{y} - y)^2$

Exploding gradient continued

$$\frac{\partial L}{\partial z^{(4)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z^{(4)}} = \alpha^4 x \ - y.1$$

$$\frac{\partial L}{\partial a^{(3)}} = \frac{\partial L}{\partial z^{(4)}}\frac{\partial z^{(4)}}{\partial a^{(3)}} = (\alpha^4 x \ - y.1)\alpha I$$

$$\frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial a^{(3)}}\frac{\partial a^{(3)}}{\partial z^{(3)}} = (\alpha^4 x \ - y.1)\alpha.1$$

$$\frac{\partial L}{\partial a^{(2)}} = \frac{\partial L}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial a^{(2)}} = (\alpha^4 x \ - y.1)\alpha\alpha.I = \alpha^2(\alpha^4 x \ - y.1)$$

…

$$\frac{\partial L}{\partial a^{(1)}} = \alpha^3(\alpha^4 x \ - y.1)$$

$$z^{(1)} = W^{(1)}x + b^{(1)} = \alpha I x$$

$$a^{(1)} = ReLU\left(z^{(1)}\right) \ = \alpha x$$

$$z^{(i)} = \alpha I a^{(i-1)}$$

$$a^{(i)} = ReLU\left(z^{(i)}\right)$$

$$\hat{y} = ReLU\left(z^{(4)}\right)$$

$$L = \frac{1}{2}(\hat{y} - y)^2$$

If $\alpha \gg 1$ the factor α^3 explodes in the gradient computation

Caution: Sigmoid activations clip the gradient and can lead to vanishing gradients

ReLU can make the gradients large

Further readings on NN architectures

MLP: $x_{\ell+1} = \sigma(W_\ell x_\ell + b_\ell)$

ResNets: $x_{\ell+1} = x_\ell + \frac{1}{L} U_\ell^T \sigma(V_\ell x_\ell + b_\ell), U_\ell, V_\ell$ are the weight matrices and L is the number of layers, can avoid issues with gradient collapse

Neural ODEs: As the number of layers approaches $L \to \infty$ ResNets can be seen as a discretization of $\frac{dx_s}{ds} = U_s^T \sigma(V_s x_s + b_s)$

He, Zhang, Ren, and Sun. Deep residual learning for image recognition. CVPR, 2016.

Chen, Rubanova, Bettencourt, and K Duvenaud. Neural ordinary differential equations. NeuRIPs, 2018.

Example NN training in Python: Setup

```
# Dataset: a circle N = 200 # samples

X = np.random.randn(N, 2) # shape: (N, 2) r = 1.0
Y = (X[:,0]**2 + X[:,1]**2 < r**2).astype(np.float32) shape: (N, 1) 0 or 1

# Define NN architecture
input_dim = 2 hidden_dim = 8 output_dim = 1 # binary classification

# Initialize NN: small random values for weights, and zeros for biases

W1 = 0.01 * np.random.randn(input_dim, hidden_dim) # shape: (2, 8)
b1 = np.zeros((1, hidden_dim)) # shape: (1, 8)
W2 = 0.01 * np.random.randn(hidden_dim, output_dim) # shape: (8, 1)
b2 = np.zeros((1, output_dim)) # shape: (1, 1)
```

```
def compute_loss(Y_pred, Y_true): # Cross-entropy loss: $L = -\frac{1}{N} \sum_i y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$
return -1/N*sum(Y_true*log(Y_pred+epsilon) + (1-Y_true)*log(1 - Y_pred+epsilon))

# Training loop
learning_rate = 0.05 num_iterations = 1000
for i in range(num_iterations):
    # Forward pass Layer 1: Z1 = XW1 + b1; Layer 2: Z2 = A1W2 + b2
    Z1 = np.dot(X, W1) + b1 # shape: (N, 8)
    A1 = relu(Z1) # shape: (N, 8)
    Z2 = np.dot(A1, W2) + b2 # shape: (N, 1)
    A2 = sigmoid(Z2) # Y_pred
    loss = compute_loss(A2, Y)

    # Backward pass; For cross-entropy and sigmoid, dL/dZ2 = A2 - Y
    dZ2 = A2 - Y # shape: (N, 1)
    dW2 = np.dot(A1.T, dZ2) # shape: (8, 1)
    dB2 = np.sum(dZ2, axis=0, keepdims=True) # shape: (1,1)
    dA1 = np.dot(dZ2, W2.T) # shape: (N,8)
    dZ1 = dA1 * relu_derivative(Z1) # shape: (N,8)
    dW1 = np.dot(X.T, dZ1) # shape: (2,8)
    dB1 = np.sum(dZ1, axis=0, keepdims=True) # shape: (1,8)
```

```
# Update parameters
W1 -= learning_rate * dW1
b1 -= learning_rate * dB1
W2 -= learning_rate * dW2
b2 -= learning_rate * dB2
```

Example in Pytorch

```
class TwoLayerNet(nn.Module):
     def __init__(self, input_dim=2, hidden_dim=8, output_dim=1):
        super(TwoLayerNet, self).__init__()
    self.layer1 = nn.Linear(input_dim, hidden_dim) # W1, b1
    self.layer2 = nn.Linear(hidden_dim, output_dim) # W2, b2

  def forward(self, x): # x shape: (N, 2)
   z1 = self.layer1(x)    a1 = self.relu(z1)   z2 = self.layer2(a1)
   y_hat = self.sigmoid(z2)
   return y_hat

# 2layer NN Binary Cross Entropy ; SGD
model = TwoLayerNet() criterion = nn.BCELoss()  optimizer = optim.SGD(...)
for i in range(num_iterations):                   # Training loop
  optimizer.zero_grad()                           # 1. Zero the parameter gradients
  y_pred = model(X_torch)                         # 2. Forward pass
  loss = criterion(y_pred, Y_torch)               # 3. Compute loss
  loss.backward()                                                 # 4. Backward pass (compute gradients)
  optimizer.step()                                                # 5. Update parameters
```

Bias-variance tradeoff

Training loss/ error of learning algorithms has two main components:

- **Bias:** error due to simplifying model assumptions
- **Variance:** error due to randomness of training set

Bias-variance tradeoff can be tuned with hyperparameters during the **validation phase**

- E.g. hyperparameters: number of layers, activation type, network architecture, etc.

High bias, low variance Low bias, high variance

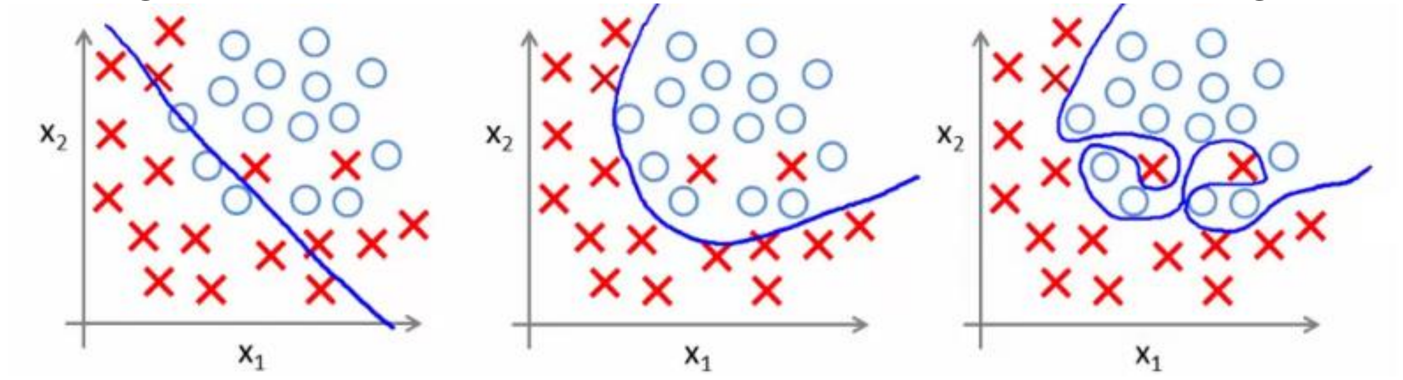


Figure source

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

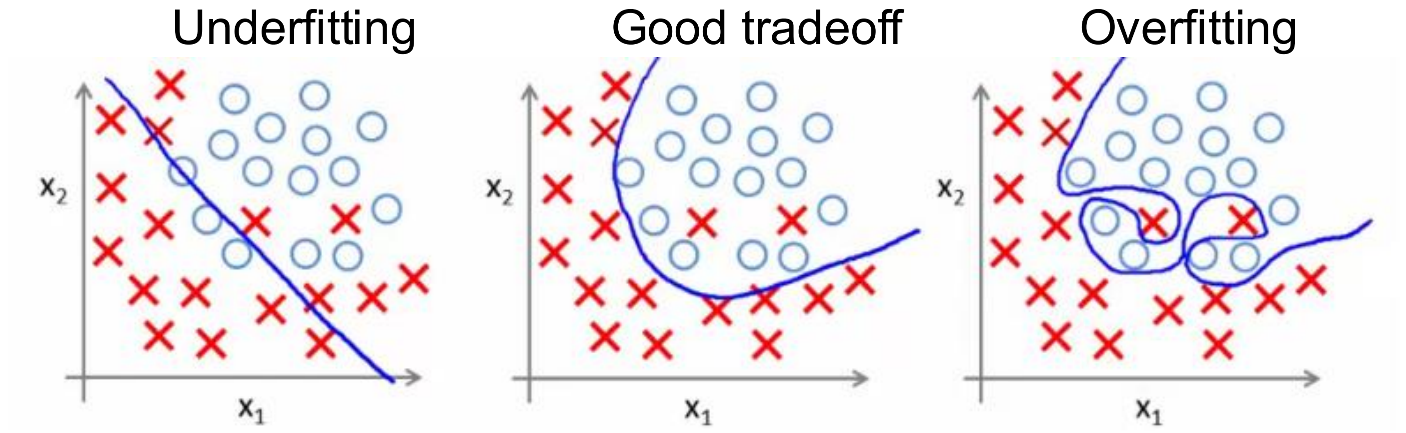


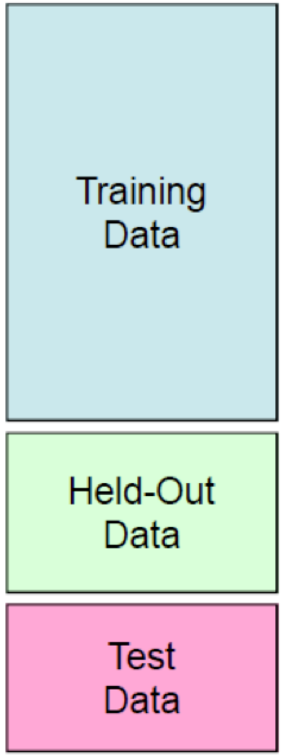
Figure source

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***

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


Training Data
Held-Out Data
Test Data