ECE 484: Principles of Safe Autonomy Fall 2025 Lecture 4: Perception Part I: Neural Networks

Professor: Huan Zhang

Sep 4, 2025

https://publish.illinois.edu/safe-autonomy/

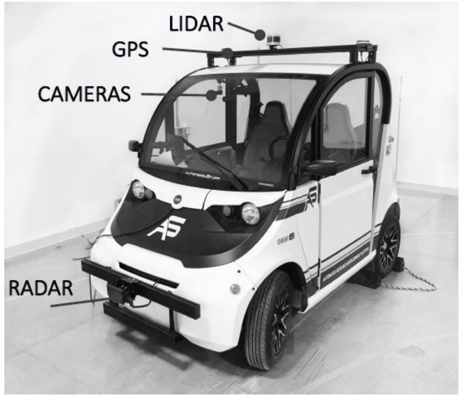
https://huan-zhang.com

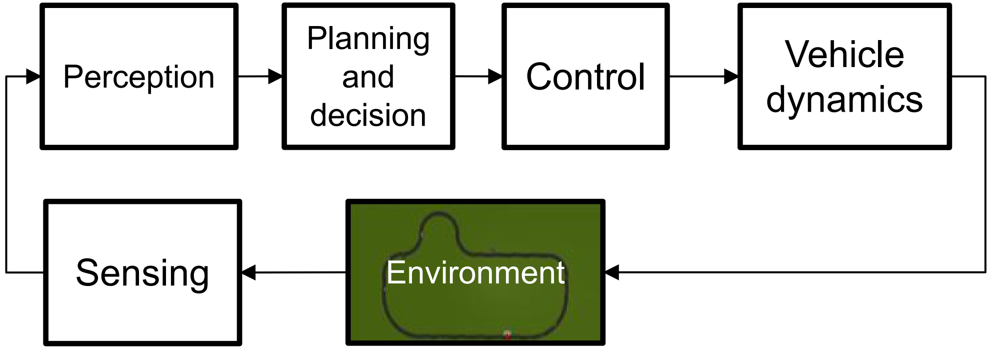
huanz@illinois.edu

Announcements

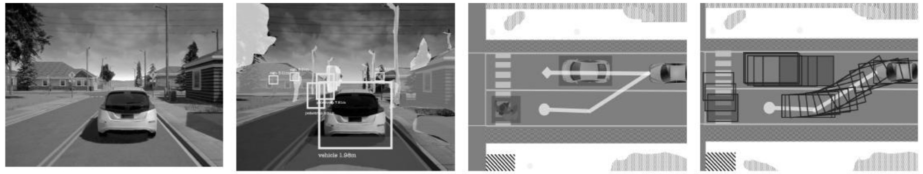
- Check CampusWire daily! Important announcements will be posted there.
- Fill out the form to create groups by 11:59 pm CT Friday (9/5)
- There have been students who keep requesting access, and it's because they need to activate their Illinois Google account

Review: How does an autonomous vehicle work?





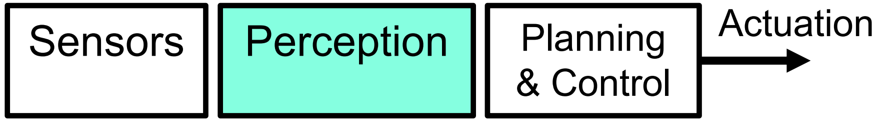
Lidar / vision



Sensing	Perception	Decisions and planning	Control
Physics-based models of cameras, LIDAR, radar, GPS, and so on.	Programs for object tracking, scene understanding, and so on.	Programs and multi-agent models of pedestrians, cars, and so on.	Dynamical models of vehicle engine, powertrain, steering, tires, and so on.

Role of Perception in Autonomy

Perception module converts **signals** from the environment to **state estimates** for the autonomous agent and its environment



Examples of state estimates:

- Type of lead vehicle, traffic sign
- Position of ego on the map, relative to the lane, distance to the leading vehicle
- Position of lead vehicle, speed, intention of the pedestrian

Types of estimates:

- Semantic: E.g., type/class of vehicle, sign
- Geometric: E.g., position, speed

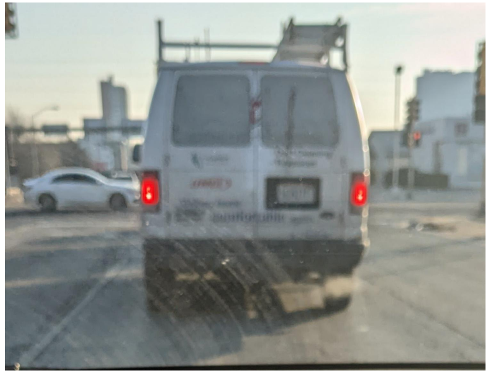
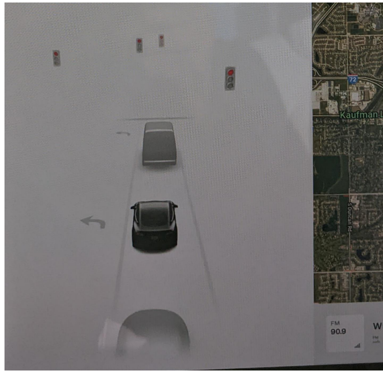
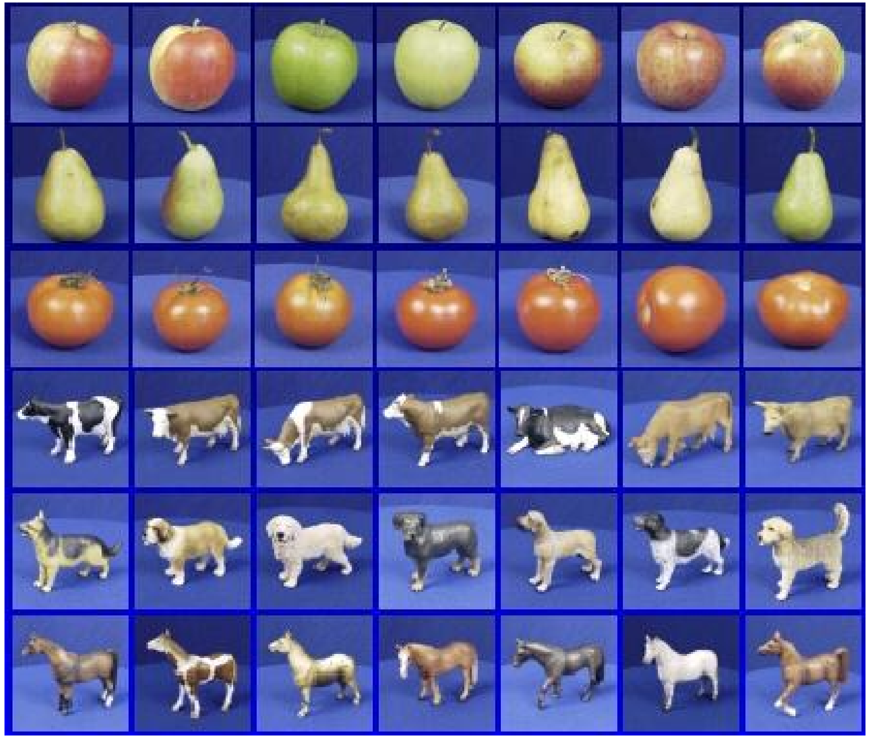
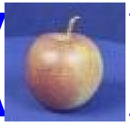



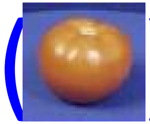
Image classification problem

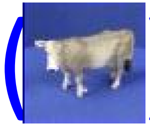


Goal: Learn a function that predicts the object in an image

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

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

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

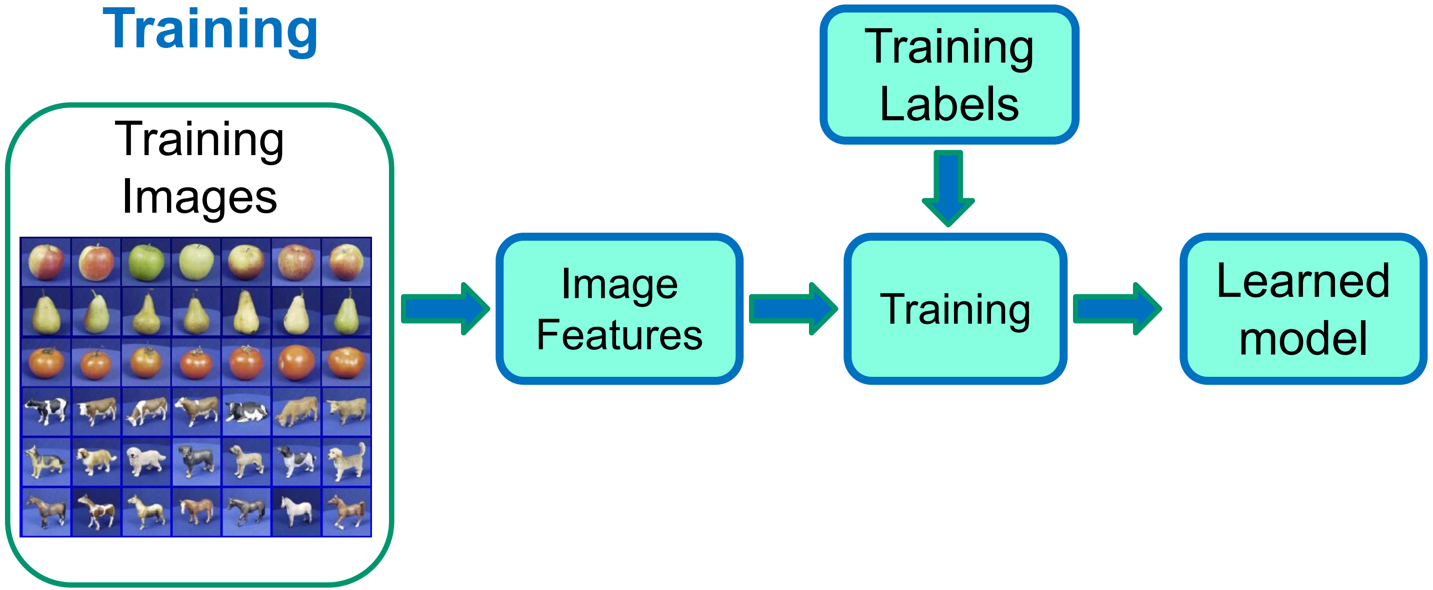
Linear classifiers

Neural networks

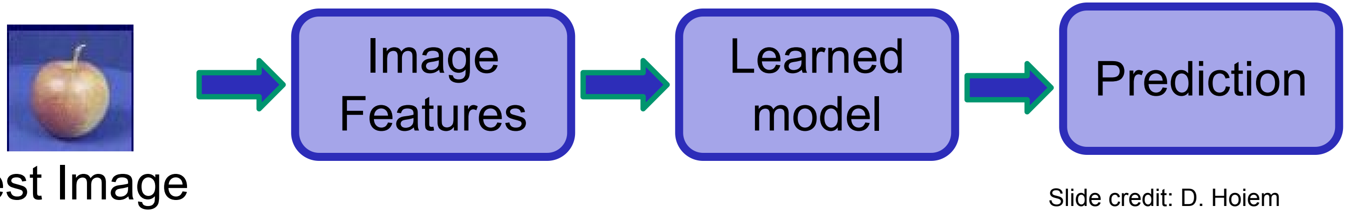
- Universal approximators
- Forward pass
- Backpropagation; Gradient descent
- Common neural network architectures
- *Exploding and vanishing gradients

Best practices

Training



Testing

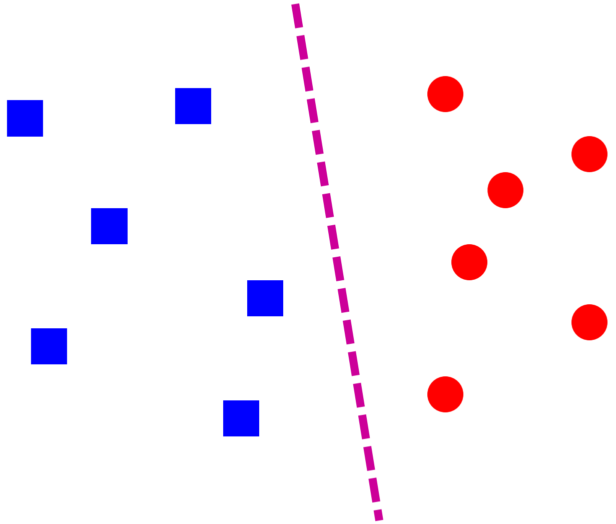


Slide credit: D. Hoiem

Linear classifiers

If interested, read Support vector machines (SVM) https://greitemann.dev/svm-demo

Images or feature representation of images

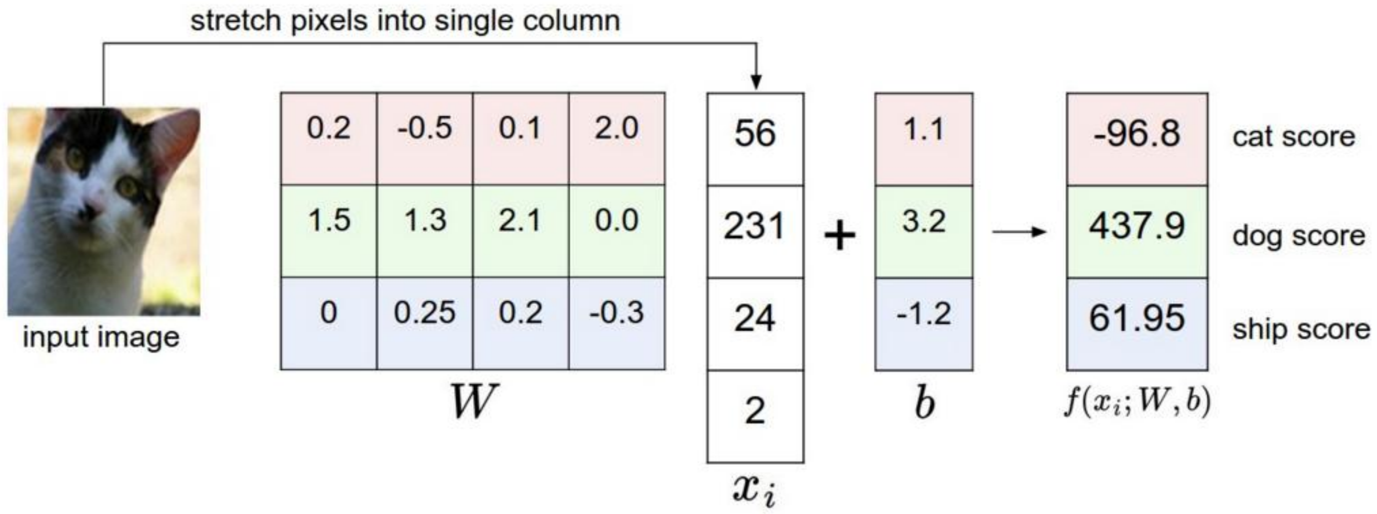


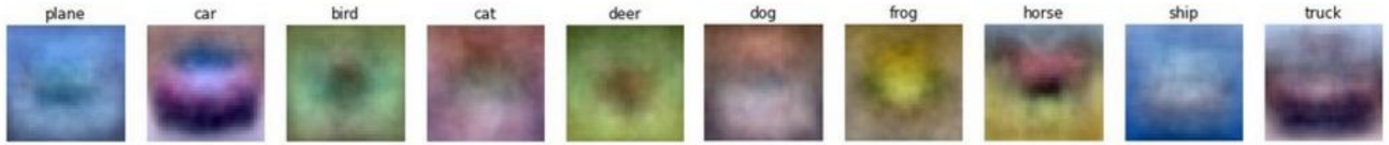
Find a *linear function (w,b)* to separate the classes:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$$

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

Visualizing linear classifiers with many classes

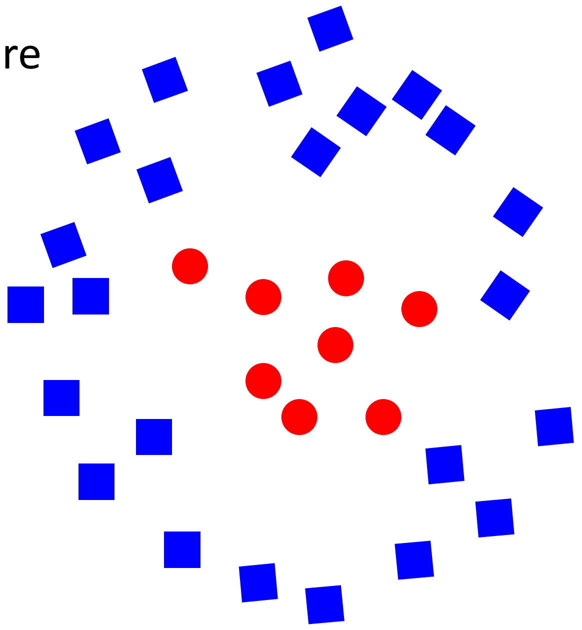




Source: Andrej Karpathy, http://cs231n.github.io/linear-classify/

Limitations of Linear Classifiers

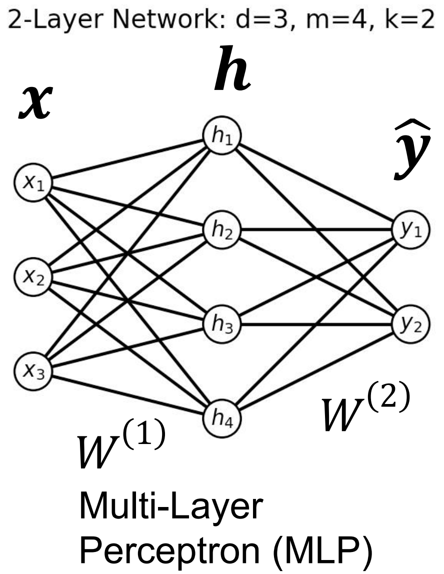
- Input: A feature vector $\boldsymbol{x} \in \mathbb{R}^d$.
- Weights and bias: $\boldsymbol{w} \in \mathbb{R}^d$, $b \in \mathbb{R}$.
- Prediction (binary classification example): $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$
- Limitations: Linear decision boundaries may not capture complex relationships between classes



- How to address this limitation and build more practical verifiers?
  - How about "stacking" multiple linear functions? Will that work?
  - Is just using linear functions enough?

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.



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

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

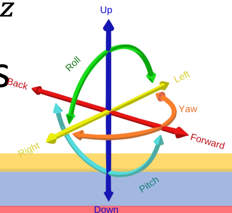
$\widehat{\boldsymbol{y}} = W^{(2)}\sigma\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}$

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

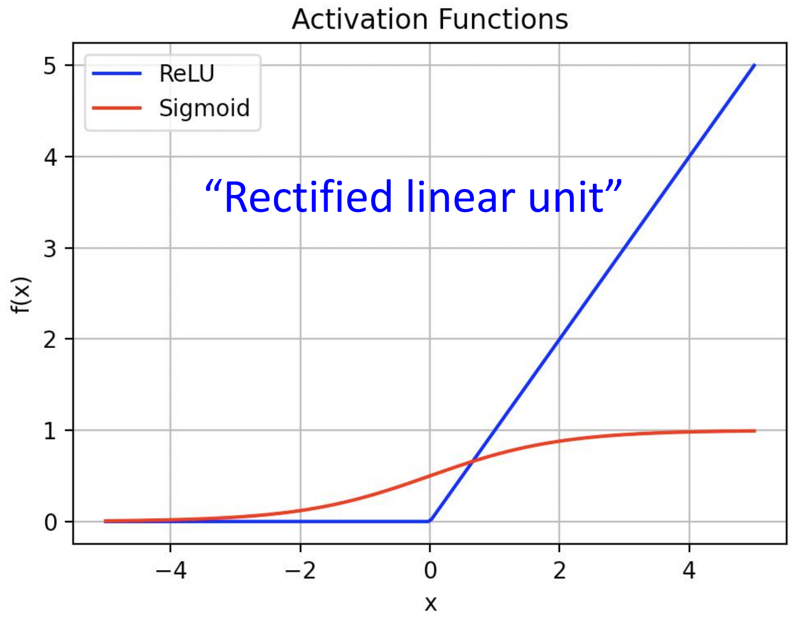
$b^{(1)} \in \mathbb{R}^{m}$ $b^{(2)} \in \mathbb{R}^{k}$ are the **biases**

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

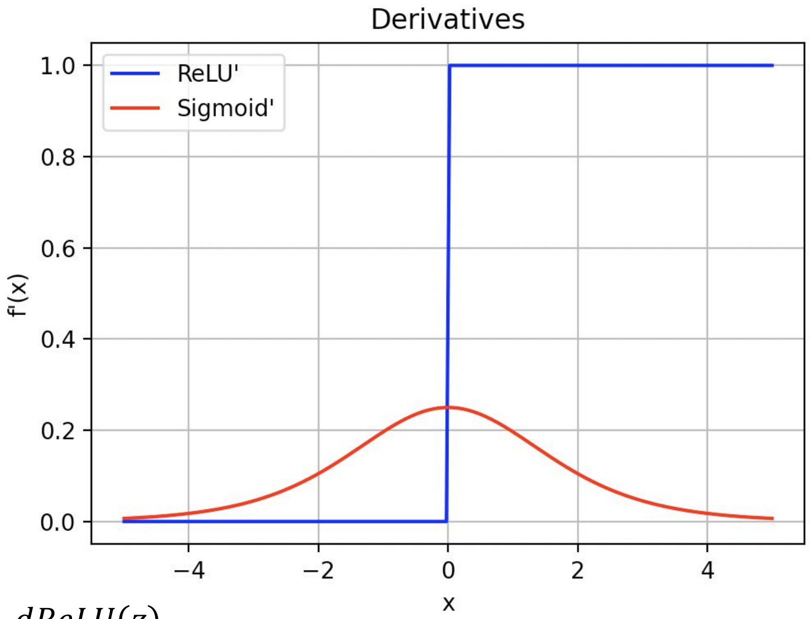
Example k=2 for lane boundary parameters, k=6 for pose components



Activation functions and their derivatives



"Rectified linear unit"



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

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

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

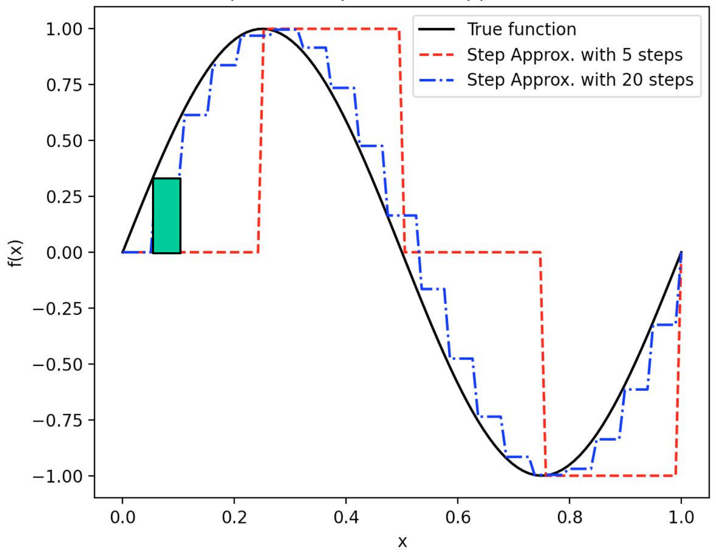
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Universal Approximation Theorem [Cybenko, G. 1989]

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

- f can be approximated arbitrarily by a sum of towers
- A tower function can be represented by a network w a single hidden layer (think of large w and b)
- 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



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

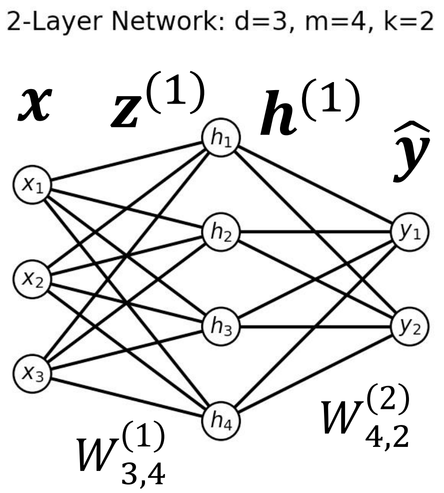
Neural Network Forward Propagation

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

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

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

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



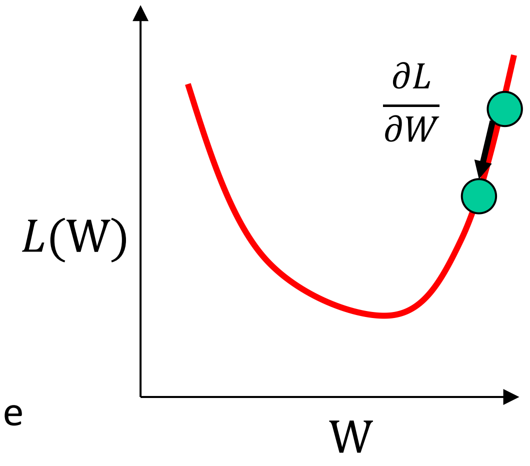
Backpropagation and Gradient-Based Training

Loss Function: A scalar function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ measures how well predictions match the ground truth $\boldsymbol{y}$. (lower is better)

For lane segmentation (classification per pixel) cross-entropy loss

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right],\ y_i \in \{0,1\}\ \hat{y}_{\boldsymbol{i}} \in [0,1]$$

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



The **loss function** (L) function is minimized during training by changing the weights (W) and the biases (b) of the neural network (f) using **back propagation + gradient descent**

Backpropagation and Gradient-Based Trainin

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

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

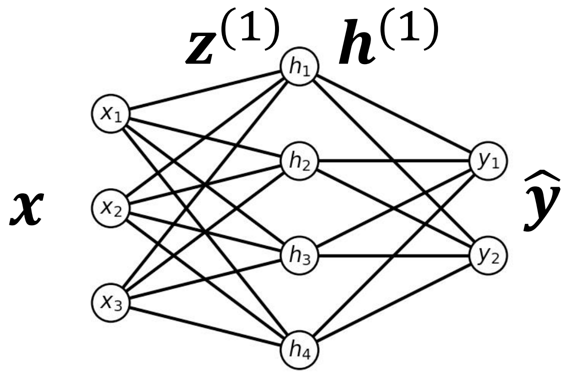
Computing Gradients (Backprop):

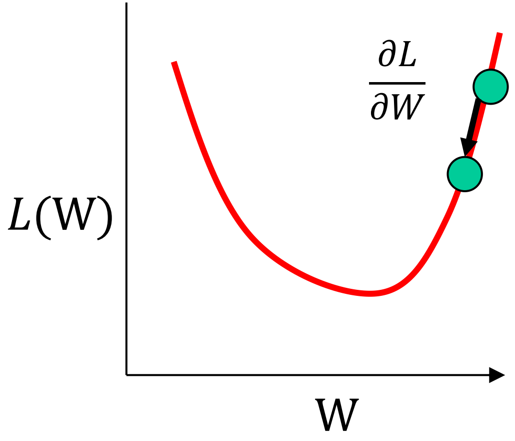
- We compute $\frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial b^{(2)}}, \frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial b^{(1)}}$ using chain rule
- For example: $\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \widehat{\boldsymbol{y}}} \frac{\partial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

Gradient Descent Update: With learning rate η

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

2-Layer Network: d=3, m=4, k=2





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

$\frac{\partial L}{\partial \hat{\boldsymbol{y}}}$ is a vector with two elements: $(\frac{\partial L}{\partial \hat{\boldsymbol{y}}_1}, \frac{\partial L}{\partial \hat{\boldsymbol{y}}_{\mathbf{2}}})$

$\frac{\partial L}{\partial \hat{\boldsymbol{y}}_1} = (\hat{\boldsymbol{y}}_1 - \boldsymbol{y}_{\mathbf{1}}), \quad \frac{\partial L}{\partial \hat{\boldsymbol{y}}_{\mathbf{2}}} = (\hat{\boldsymbol{y}}_2 - \boldsymbol{y}_{\mathbf{2}})$

Exercise: $\hat{\boldsymbol{y}} = W\boldsymbol{h} + \boldsymbol{b}^{(2)}$, solve $\frac{\partial \hat{\boldsymbol{y}}}{\partial W}$

$$\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{\boldsymbol{y}}_1 \\ \hat{\boldsymbol{y}}_2 \end{bmatrix} = \begin{bmatrix} w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 \\ w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2 \end{bmatrix}$$

Number of elements in $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{W}}$: 2 × 2 × 3 (two outputs, 2 × 3 elements in W)

For simplicity, we can "vectorize" W into a vector with 6 elements. Then we calculate the 2x6 Jacobian matrix:

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

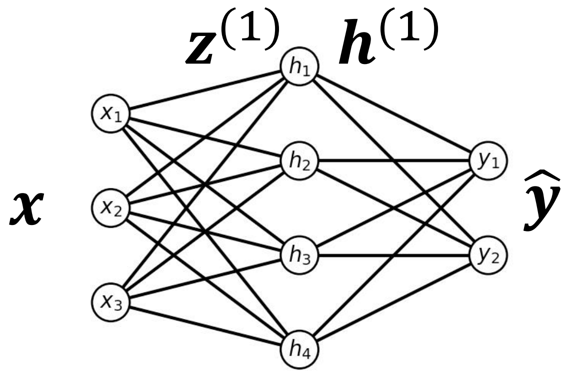
$$= \begin{bmatrix} h_1 & h_2 & h_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & h_1 & h_2 & h_3 \end{bmatrix}$$

Abusing the notation a little bit for simplicity - we can call this $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{W}}$

Finally, $$\overset{\text{Shape 1x6}}{\frac{\partial L}{\partial W^{(2)}}} = \underset{\text{Shape 1x2}}{\frac{\partial L}{\partial \hat{\boldsymbol{y}}}}\,\underset{\text{Shape 2x6}}{\frac{\partial \hat{\boldsymbol{y}}}{\partial W^{(2)}}} = [\hat{\boldsymbol{y}}_1 - \boldsymbol{y}_1, \hat{\boldsymbol{y}}_2 - \boldsymbol{y}_2]\begin{bmatrix} h_1 & h_2 & h_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & h_1 & h_2 & h_3 \end{bmatrix}$$

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

2-Layer Network: d=3, m=4, k=2



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

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

Exercises:

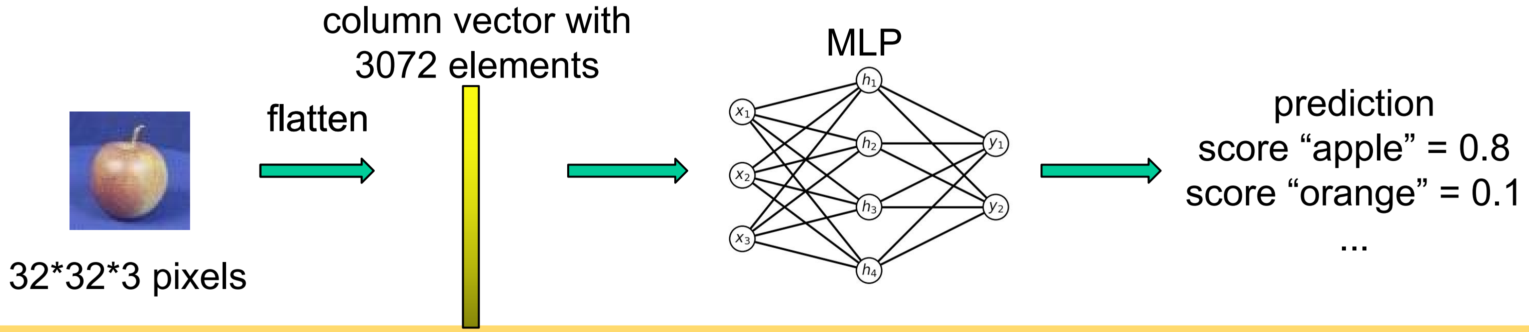
1. What is the shape of each Jacobian matrix? 1x2, 2x4, 4x4, 4x(4x3)
2. What is $\frac{\partial h^{(1)}}{\partial z^{(1)}}$? A Diagonal matrix with 0 or 1 on its diagonal, depending on the value of $z^{(1)}$ during forward propagation

NN architectures

In the above example, we used a simple matrix W in neural network:

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

$W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$ is called a fully-connected (FC) layer, and a neural network with multiple layers of FC layers with nonlinear function is called a "Multilayer perceptron" (MLP) network



NN architectures: convolutional layers

Fully connected layer $W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$ can be replaced by other linear or nonlinear operators.

For images, flatten then into a 1-d vector may not be ideal - we lose the spacial insight of the input.

Layers that are often used in NNs for perception:

1. Convolutional layers
2. Pooling layers (maxpool, avgpool)
3. Transposed convolution layers
4. Normalization layers
5. Dropout

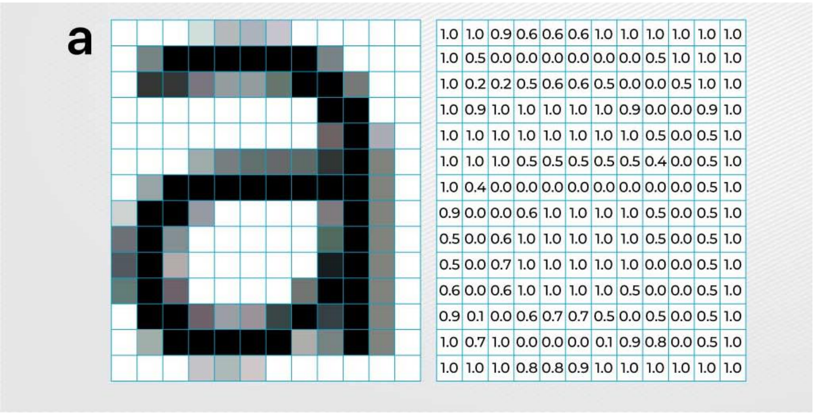
...

NN architectures: convolutional layers

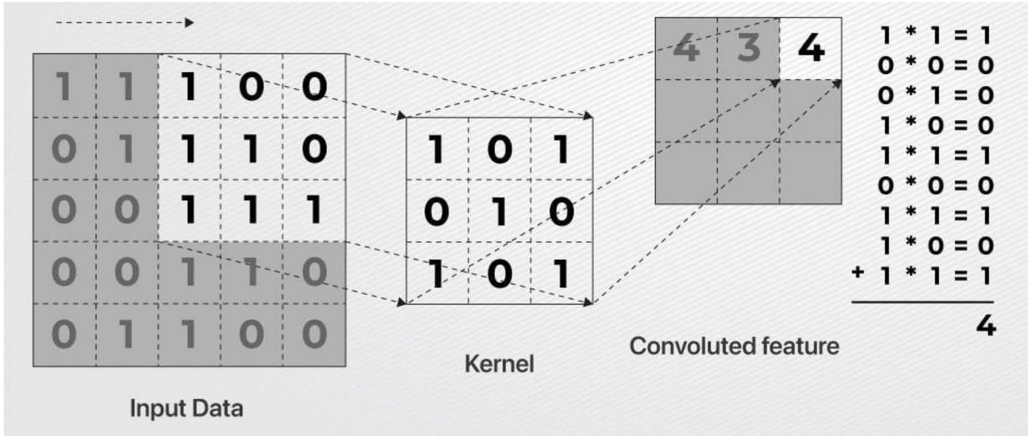
The convolution operator instead operate on a 1D, 2D, or 3D inputs directly, focusing on input features that are sptially adjacent

“Kernel” or “filter” *K* replaces the weights *W*

Typically, spatial dimension is reduced after convolution



1.0	1.0	0.9	0.6	0.6	0.6	1.0	1.0	1.0	1.0	1.0	1.0
1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	1.0	1.0
1.0	0.2	0.2	0.5	0.6	0.6	0.5	0.0	0.0	0.5	1.0	1.0
1.0	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.9	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.4	0.0	0.5	1.0
1.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0
0.9	0.0	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.6	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.7	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.5	1.0
0.6	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.0	0.5	1.0
0.9	0.1	0.0	0.6	0.7	0.7	0.5	0.0	0.5	0.0	0.5	1.0
1.0	0.7	1.0	0.0	0.0	0.0	0.1	0.9	0.8	0.0	0.5	1.0
1.0	1.0	1.0	0.8	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0



$\boldsymbol{x}$ (5x5) $\boldsymbol{K}$ (3x3) $\boldsymbol{x} * \boldsymbol{K}\ (3x3)$

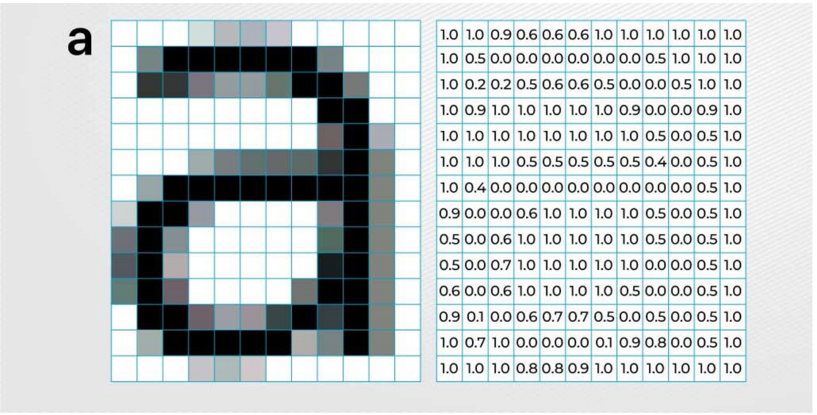
https://www.analytixlabs.co.in/blog/convolutional-neural-network/

NN architectures: convolutional layers

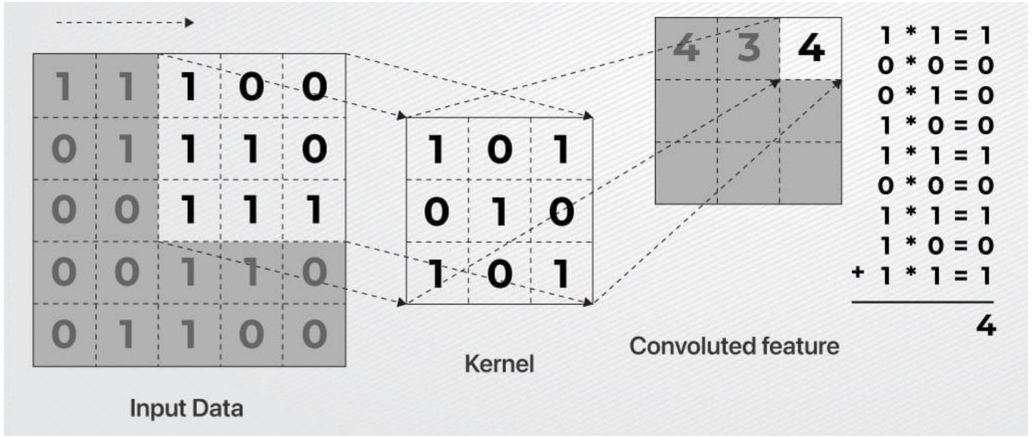
The convolution operator instead operate on a 1D, 2D, or 3D inputs directly, focusing on input features that are sptially adjacent

“Kernel” or “filter” K replaces the weights W

Typically, spatial dimension is reduced after convolution



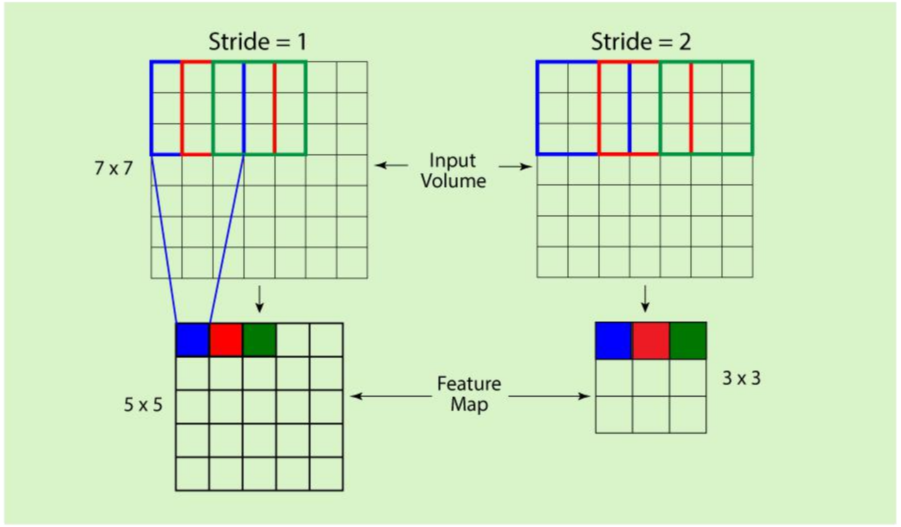
1.0	1.0	0.9	0.6	0.6	0.6	1.0	1.0	1.0	1.0	1.0	1.0
1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	1.0	1.0
1.0	0.2	0.2	0.5	0.6	0.6	0.5	0.0	0.0	0.5	1.0	1.0
1.0	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.9	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.4	0.0	0.5	1.0
1.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0
0.9	0.0	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.6	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.7	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.5	1.0
0.6	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.0	0.5	1.0
0.9	0.1	0.0	0.6	0.7	0.7	0.5	0.0	0.5	0.0	0.5	1.0
1.0	0.7	1.0	0.0	0.0	0.0	0.1	0.9	0.8	0.0	0.5	1.0
1.0	1.0	1.0	0.8	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0



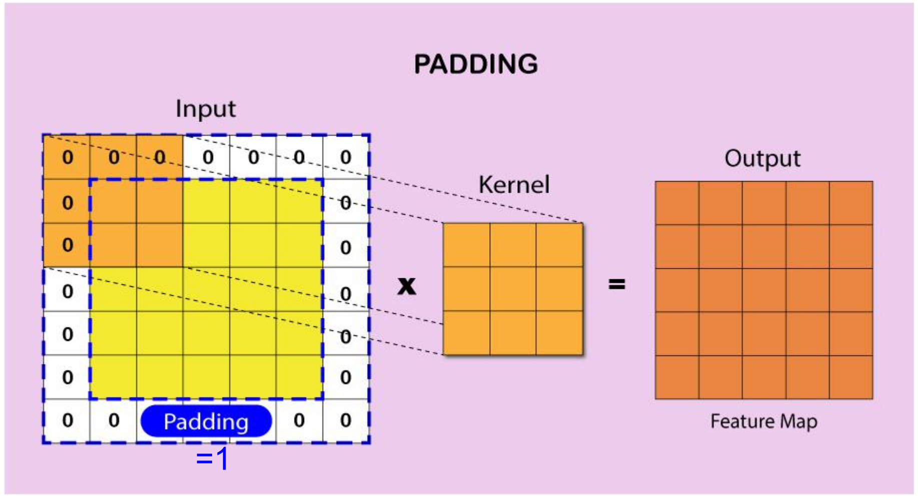
$\boldsymbol{x}$ (5x5) $\boldsymbol{K}$ (3x3) $\boldsymbol{x} * \boldsymbol{K}\ (3x3)$

https://www.analytixlabs.co.in/blog/convolutional-neural-network/

Convolutional layers with stride and padding



(how many columns/rows to skip when moving the kernel)



(how many extra rows/columns of zeros added to all sides of inputs)

https://developersbreach.com/convolution-neural-network-deep-learning/

Convolutional layers with dilation

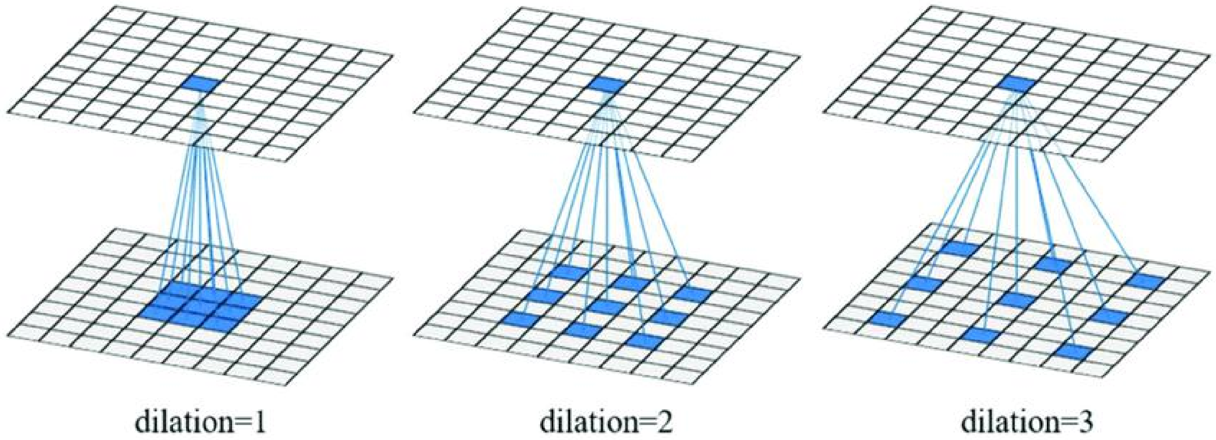


Image from https://www.mdpi.com/2072-4292/11/19/2220

Convolutional layers with multiple input channels and output channels

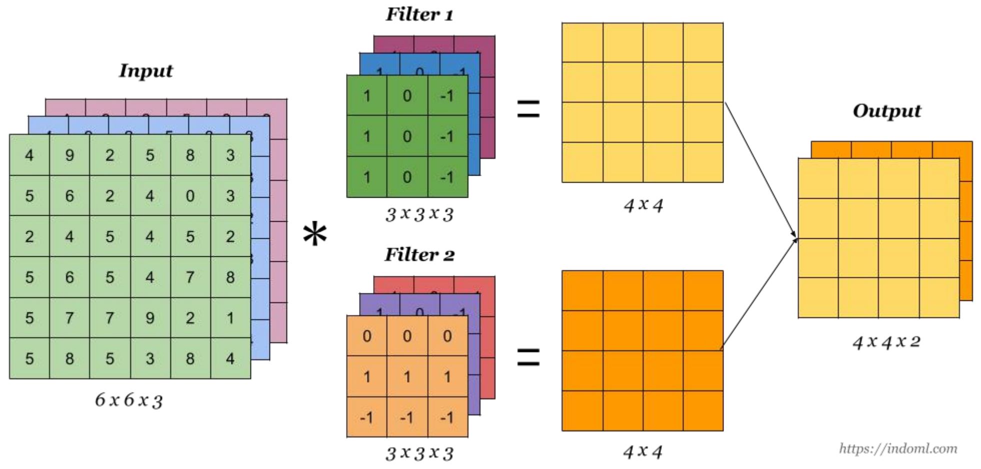
Input dimension: 6x6x3

Kernel (filter) size: 3*3*3

Output channel: 2

Stride: 1

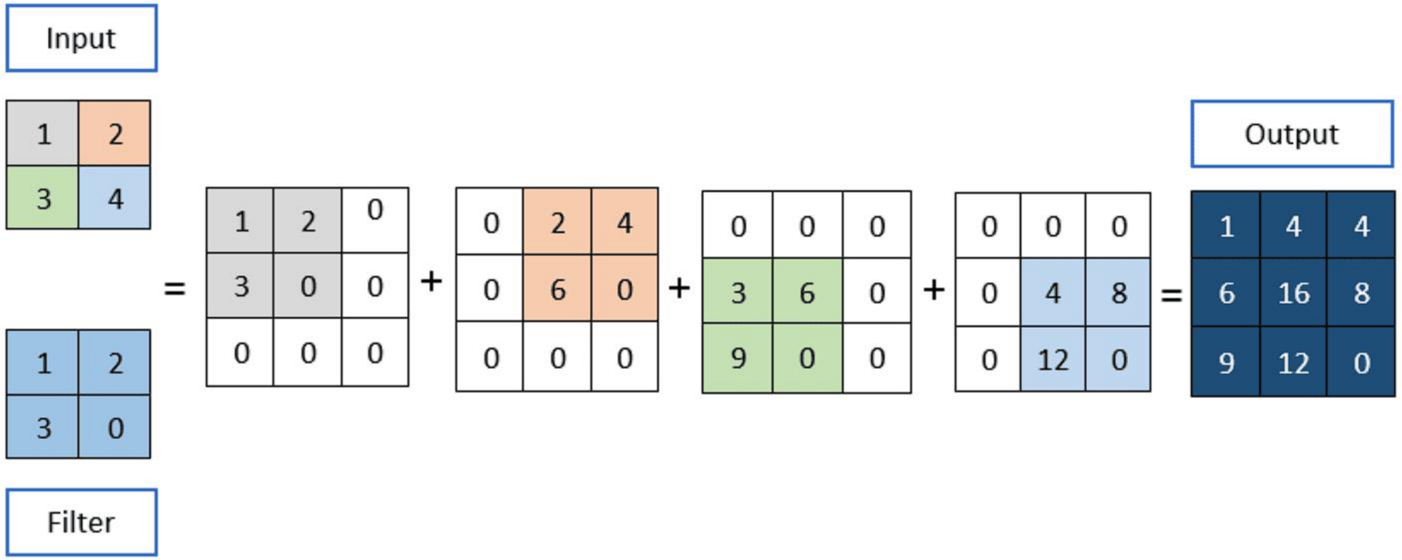
Padding: 0



HW: given input dimension, kernel size, stride, padding, dilation, calculate output dimension

Transposed convolution

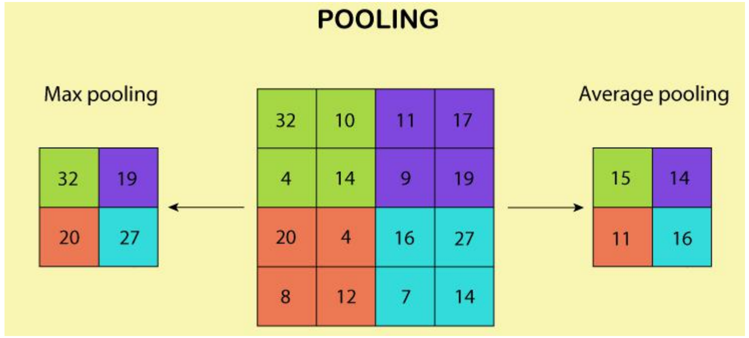
- Transposed convolution can increase the spatial dimension (2x2 -> 3x3 in this example)
- Useful for upsampling



https://viso.ai/deep-learning/convolution-operations/

Pooling layers

Take max of all numbers in the 2x2 square of the same color



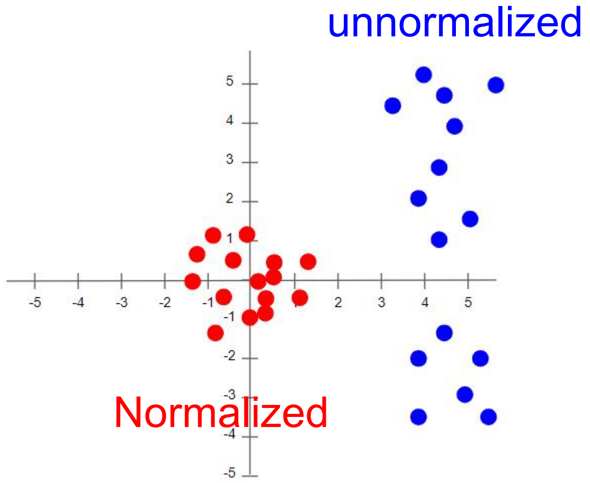
Take the average of all numbers in the 2x2 square of the same color

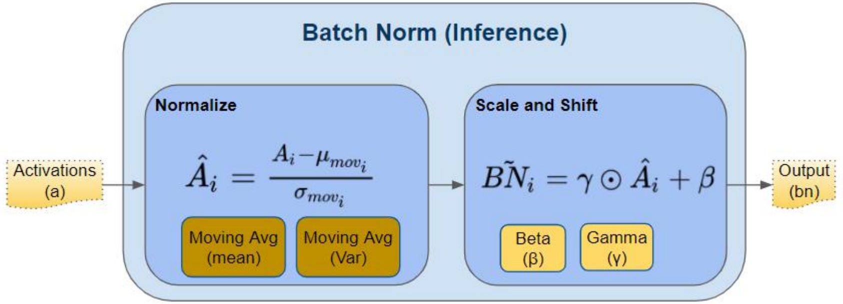
- Pooling parameters: kernel size, stride, padding (typically, kernel size = stride)
- Pooling layer has no parameters to learn (it is a fixed operation)
- Average pooling is a special case of convolution (why?)

https://developersbreach.com/convolution-neural-network-deep-learning/

Batch normalization

- Simple idea: let the network always processes features that have zero mean and a variance of 1, so normalize the features





mean and average calculated using activations during training

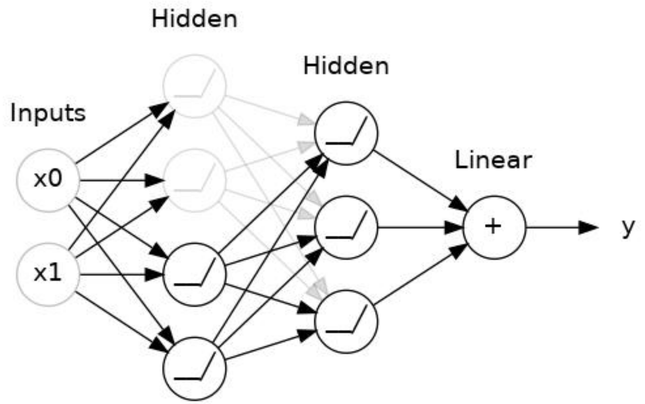
Further learnable scale and shift is optional

https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739/

Read about different types of normalization layers: https://arxiv.org/pdf/1803.08494

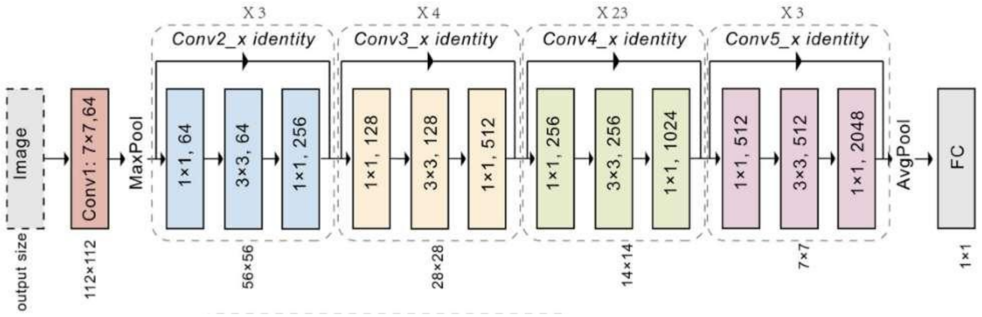
Dropout

- During training, randomly remove some connections in the NN
- Avoid learning spurious patterns in training data, and tend to obtain a more robust and generalizable NN
- During testing, no neuron is removed and Dropout becomes identity

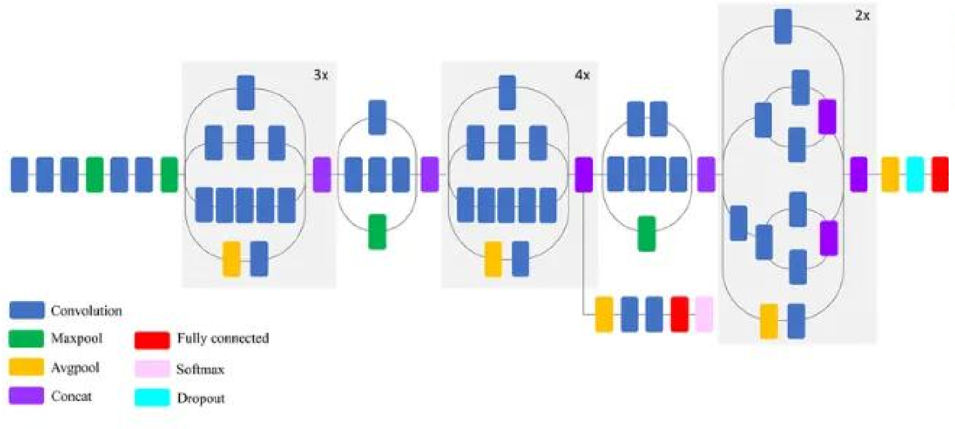


https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization

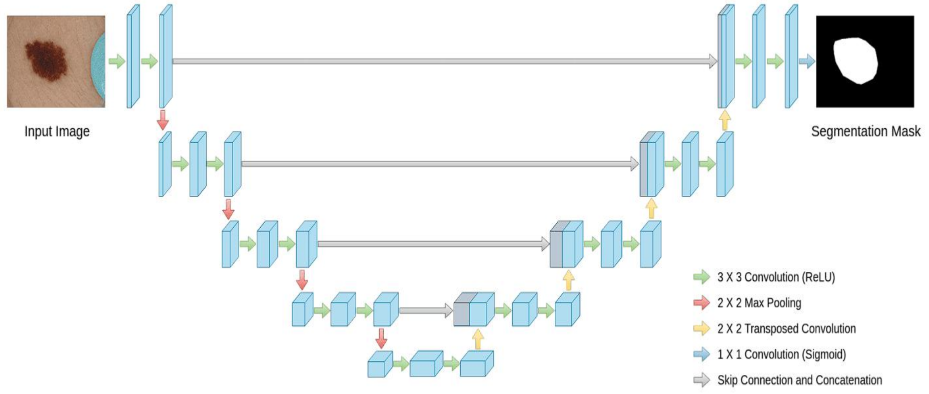
More general NN architecutres



ResNet (residual network)



Inception-v3


Input Image
Segmentation Mask
3 X 3 Convolution (ReLU)
2 X 2 Max Pooling
2 X 2 Transposed Convolution
1 X 1 Convolution (Sigmoid)
Skip Connection and Concatenation

U-Net

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

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

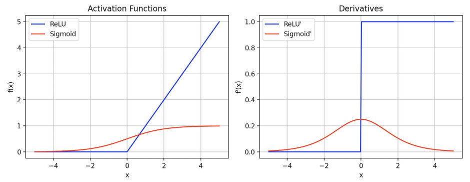
$$\frac{\partial L}{\partial a^{(3)}} = \frac{\partial L}{\partial z^{(4)}}\frac{\partial z^{(4)}}{\partial a^{(3)}} = (\hat{y}-y)\hat{y}(1-\hat{y}).W^{(4)}$$

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

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

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

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



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

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

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
```

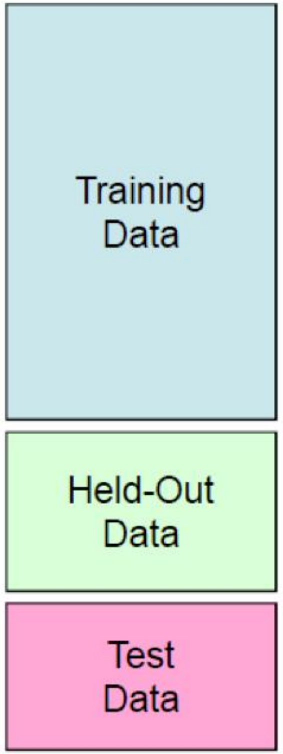
Best practices for training classifiers

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 keey 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

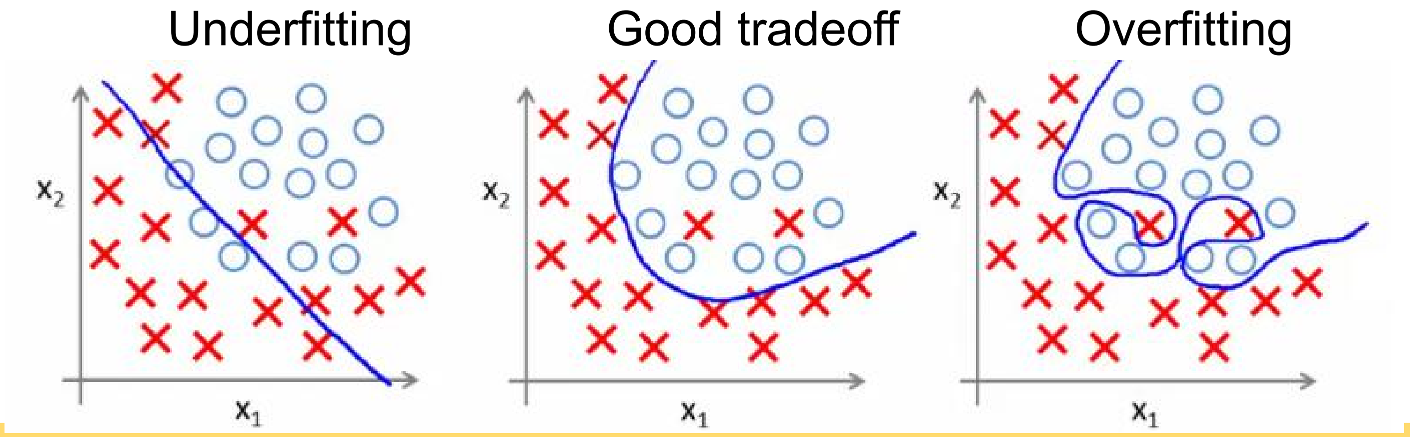


Figure source

Bias-variance tradeoff

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

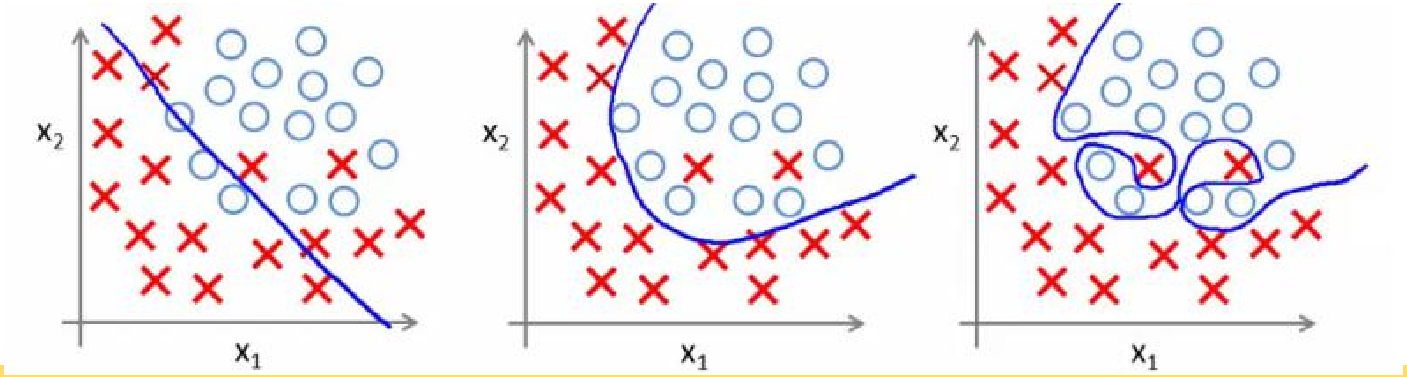
- **Bias:** error due to simplifying model assumptions
- **Variance:** error due to randomness of training set

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



x_2
x_1
x_2
x_1
x_2
x_1