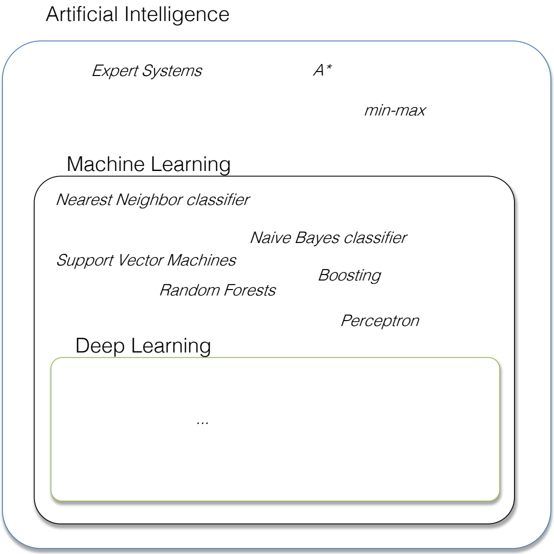
[GEIA-21B] Deep Learning: Introduction

Vincent Lepetit

September 12, 2021

AI / Machine Learning / Deep Learning



Schedule

- Today: Neural networks, architectures, optimization, intuitions.
- Tomorrow: Hands-on, deep learning for computer vision problems.
- Next session: Deep Learning in practice, selected topics.
- Last session: Project on medical imagery.

Why Deep Learning is Currently so Popular?

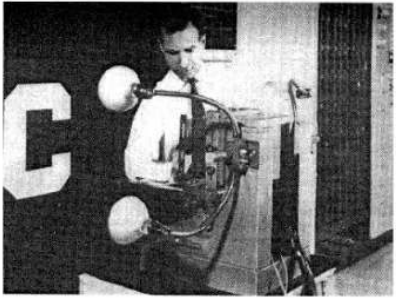
- Very general: Computer Vision, Speech Recognition, Natural Language Processing, Graphs, Chemistry, Mechanical Engineering, Computer Graphics, etc.

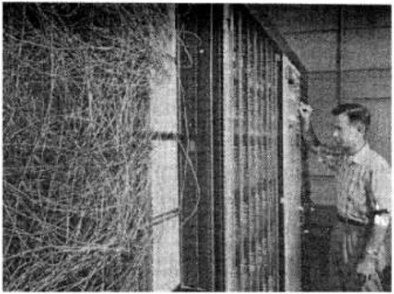
- No need to engineer features;

- Very flexible framework. Originally developed for supervised learning, but can be extended to many other problems.

- *Why now?*
  - Faster computers (with GPUs); More training data; Better optimization algorithms; Easy to use and powerful libraries in Python; People are now convinced it works.

Perceptron (1958, Frank Rosenblatt)





Perceptron

The perceptron was developed for supervised binary classification.

Input data are represented as vectors $\mathbf{x}$.

We are looking for a function $f(\mathbf{x}; \mathbf{w}, b)$ such that
$f(.; \mathbf{w}) : \mathbf{x} \in \mathbb{R} \rightarrow \{+1, -1\}$. $\mathbf{w}$ and b are the parameters of f.

Training set: We have examples $\{(\mathbf{x}_i, y_i)\}$ of vectors $\mathbf{x}$ annotated with the expected values for $f(\mathbf{x})$.

Perceptron

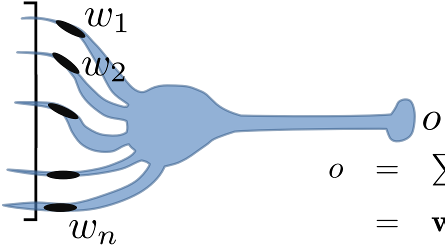
Linear model:

$$f(\mathbf{x};\, \mathbf{w}, b) = \begin{cases} +1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0 \;, \\ -1 & \text{otherwise }. \end{cases} \tag{1}$$

[we need to find parameters $\mathbf{w}$ and b. We will see that later]

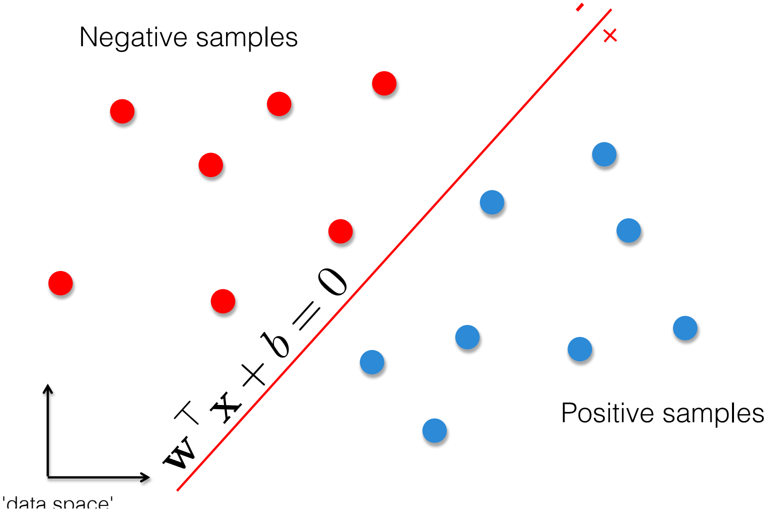
Inspired by works from neuroscientists such as Donald Hebb (see Hebbian theory and Hebb's rule).

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$



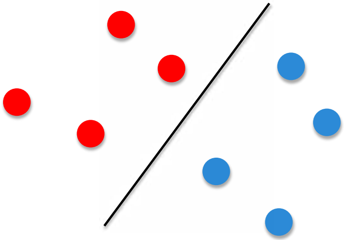
$$\begin{aligned} o &= \textstyle\sum_i w_i x_i + b \\ &= \mathbf{w}^\top \mathbf{x} + b \end{aligned}$$

Perceptron: Geometric Interpretation

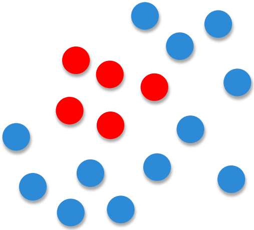


Perceptron: Limitation

A perceptron can only correctly classify data points that are linearly separable:



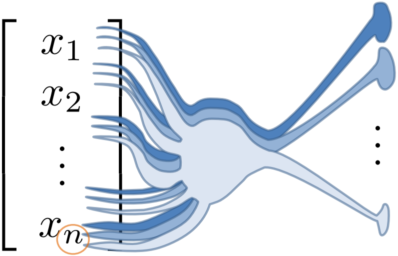
linearly separable



nonlinearly separable

- The Perceptron: A 1-layer “network”;
- A 2-layer network (1-hidden layer network)

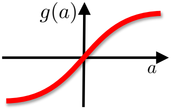
A Two-Layer Network: First Layer

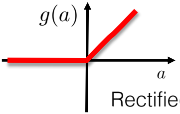


$$h_1 = g\left(\sum_i W_{1,i} x_i + b_1\right)$$

$g(\;)$: non-linear function

$$h_m = g\left(\sum_i W_{m,i} x_i + b_m\right)$$

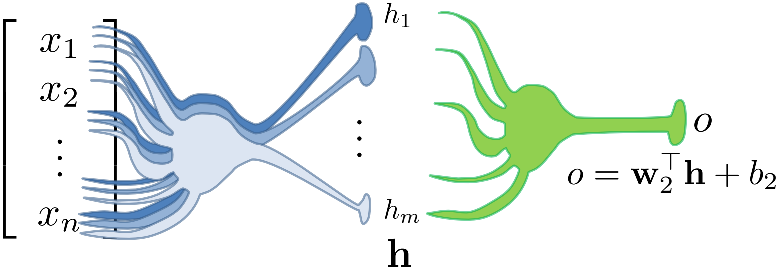




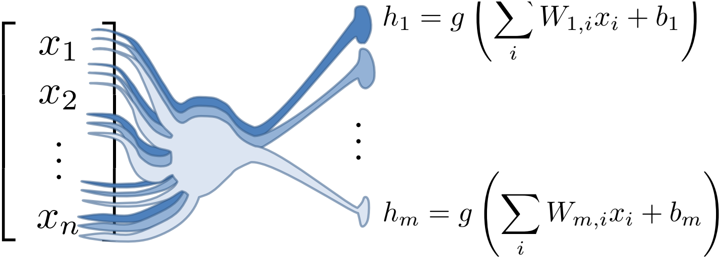
Rectified Linear Unit (ReLU)

$$g(a) = \max(0, a)$$

A Two-Layer Network: Second Layer



$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}_2^\top \mathbf{h} + b_2 \geq 0 \ , \\ -1 & \text{otherwise} \ . \end{cases}$$

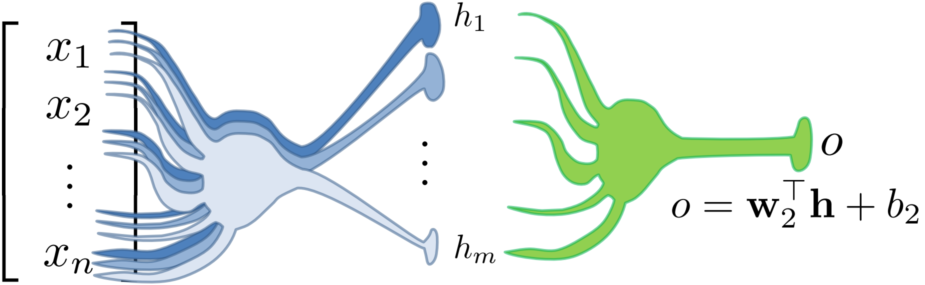


$$\mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2}$$

where $g(\mathbf{x})$ is a non-linear function.

$\mathbf{h}$ can be seen as a learned feature vector. Its dimension is a hyper-parameter.

Two-Layer Network



$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$

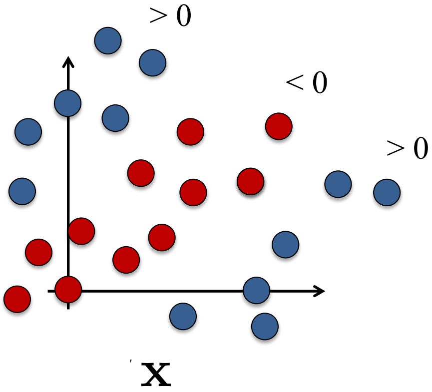
$g(\ )$: non-linear function

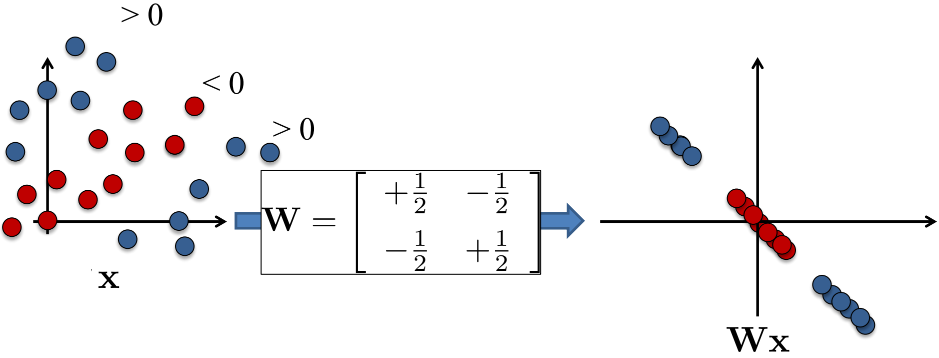
The coefficients of matrix $\mathbf{W}$, vectors $\mathbf{b}$ and $\mathbf{w_2}$, scalar b_2 are the *parameters* of the network

$$\boxed{\begin{array}{l} \left\{ \begin{array}{l} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{array} \right. \\ \\ g(\ )\text{: non-linear function} \end{array}}$$

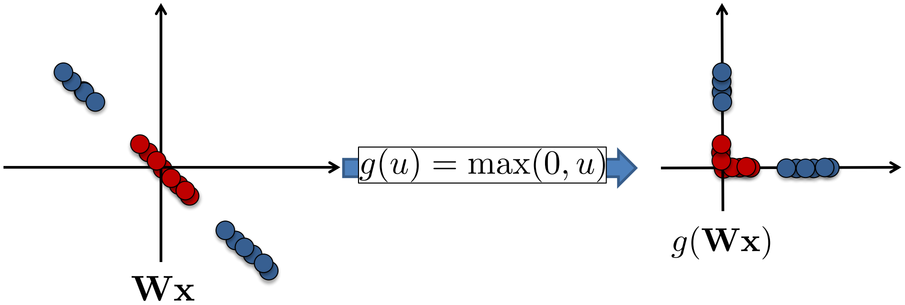
- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?

A Multilayer Network Can Solve Non-Linearly Separable Problems

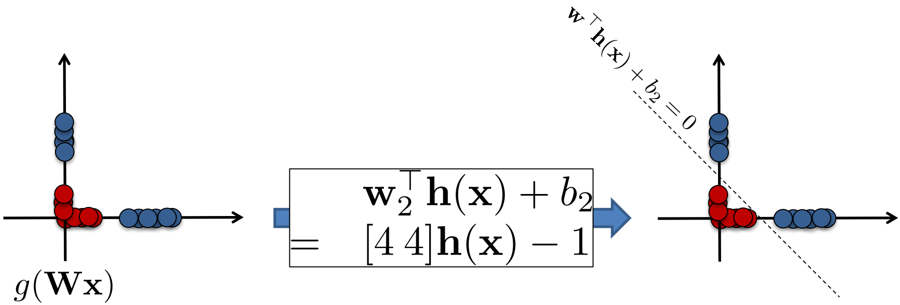




$$
\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x}) \text{ with } g(u) = \max(0, u) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}
$$



$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x}) \text{ with } g(u) = \max(0, u) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$



$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x}) \text{ with } g(u) = \max(0, u) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$

- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;

Universal Approximation Theory for Two-Layer Networks

Proves that any continuous function can be approximated under mild conditions as closely as wanted by a two-layer network:

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ o(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$

See

K. Hornik, M. Stinchcombe, and H. White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* (1989).

H. N. Mhaskar. "Neural Networks for Optimal Approximation of Smooth and Analytic Functions". In: *Neural Computing* (1996).

A. Pinkus. "Approximation Theory of the MLP Model in Neural Networks". In: *Acta Numerica* (1999).

Universal Approximation Theory for Two-Layer Networks

A two-layer network can be written as:

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \hat{f}(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$

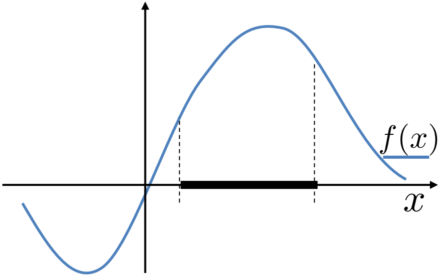
or

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^{N} c_j g(\mathbf{W}_{(j)}^\top \mathbf{x} - b_j)\,,$$

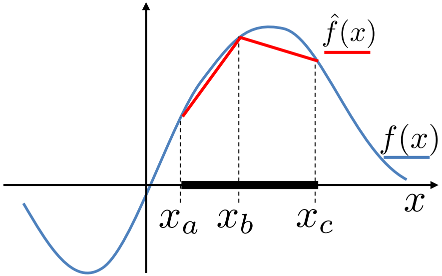
where $g : \mathbb{R} \to \mathbb{R}$ is an activation function and N is the number of hidden units.

As $N \to \infty$, any continuous function f can be approximated by some neural network $\hat{f}$, because each component $g(\mathbf{W}_{(j)}^\top \mathbf{x} - b_j)$ behaves like a basis function and functions in a suitable space admits a basis expansion.

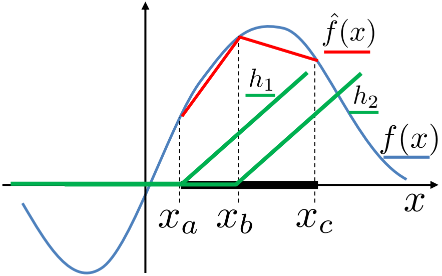
Universal Approximation Theorem: Intuition in 1D

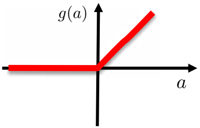


Universal Approximation Theorem: Intuition in 1D



Universal Approximation Theorem: Intuition in 1D

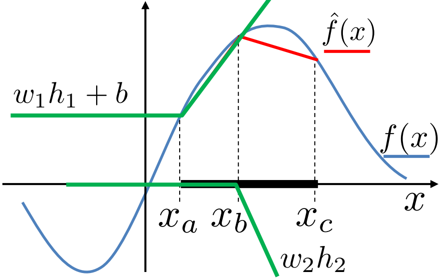


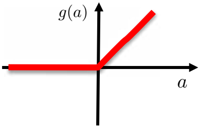


Introducing: $\begin{cases} h_1 = g(x - x_a) \\ h_2 = g(x - x_b) \\ g(x) = \max(0, x) \end{cases}$

$$\mathbf{h} = g\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} x + \begin{bmatrix} -x_a \\ -x_b \end{bmatrix}\right)$$

Universal Approximation Theorem: Intuition in 1D





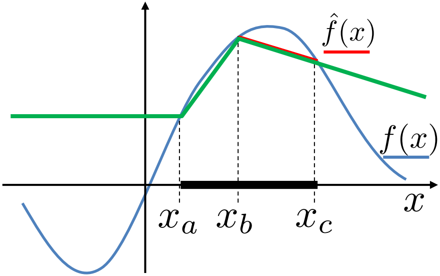
Introducing: $\begin{cases} h_1 = g(x - x_a) \\ h_2 = g(x - x_b) \\ g(x) = \max(0, x) \end{cases}$ $\quad \mathbf{h} = g\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} x + \begin{bmatrix} -x_a \\ -x_b \end{bmatrix} \right)$

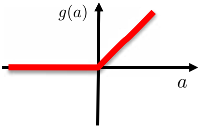
there exist w_1, w_2, b_2 such that: $\hat{f}(x) = w_1 h_1 + w_2 h_2 + b = \mathbf{w}^\top \mathbf{h} + b_2$

By introducing more x_i and h_i, $\hat{f}(x)$ can approximate $f(x)$ more closely.

Side note: Deep networks with ReLU activation functions extrapolate poorly.

Universal Approximation Theorem: Intuition in 1D





Introducing: $\begin{cases} h_1 = g(x - x_a) \\ h_2 = g(x - x_b) \\ g(x) = \max(0, x) \end{cases}$ $\quad \mathbf{h} = g(\begin{bmatrix} 1 \\ 1 \end{bmatrix} x + \begin{bmatrix} -x_a \\ -x_b \end{bmatrix})$

there exist w_1, w_2, b_2 such that: $\hat{f}(x) = w_1 h_1 + w_2 h_2 + b = \mathbf{w}^\top \mathbf{h} + b_2$

By introducing more x_i and h_i, $f(x)$ can approximate $\hat{f}(x)$ more closely.

- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;
- The structure of a 2-layer network function;

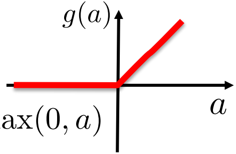
The Topology of the Function Learned by a Two-Layer Network

A two-layer network:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad \text{with } g(a) = \max(a, 0)$$
$$y = \mathbf{w}_2 \mathbf{h} + b_2$$

or, more compactly:

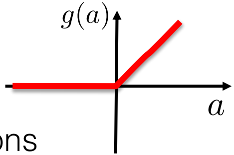
$$y = \mathbf{w}_2 g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b_2$$



$$g(a) = \max(0, a)$$

Rectified Linear Unit (ReLU)

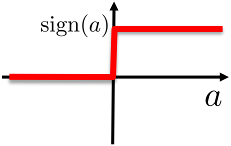
$$y = \mathbf{w}_2 g(\mathbf{W}\mathbf{x} + \mathbf{b}) + b_2$$



$y(\mathbf{x})$ is a composition of continuous functions and is therefore **continuous.**

Introduce the matrix

$$\mathbf{B}(\mathbf{x}) = \mathrm{diag}(\ldots, \mathrm{sign}(\mathbf{W}^{(i)}\mathbf{x} + \mathbf{b}^{(i)}), \ldots)$$

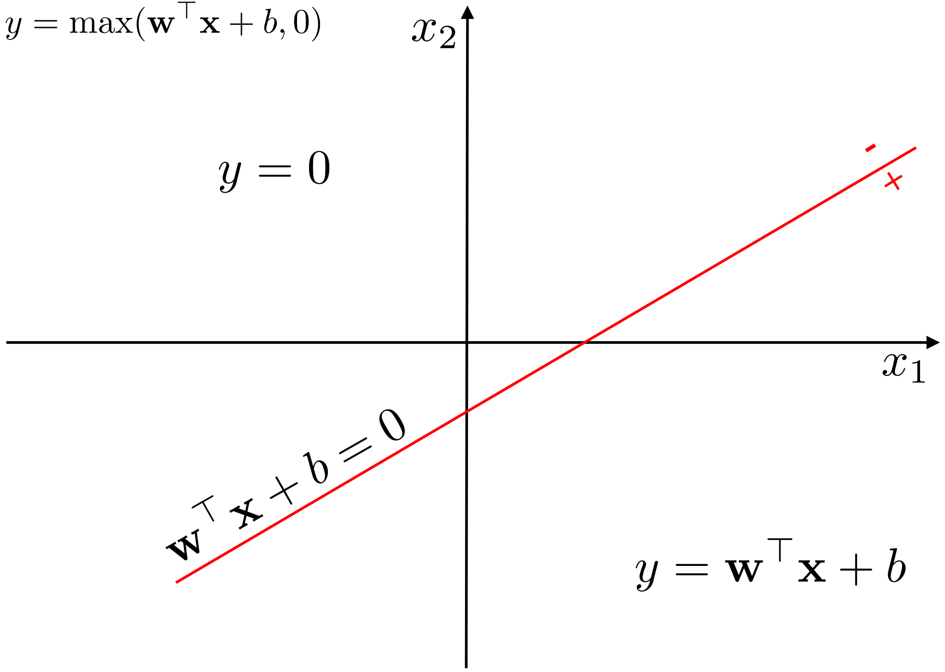


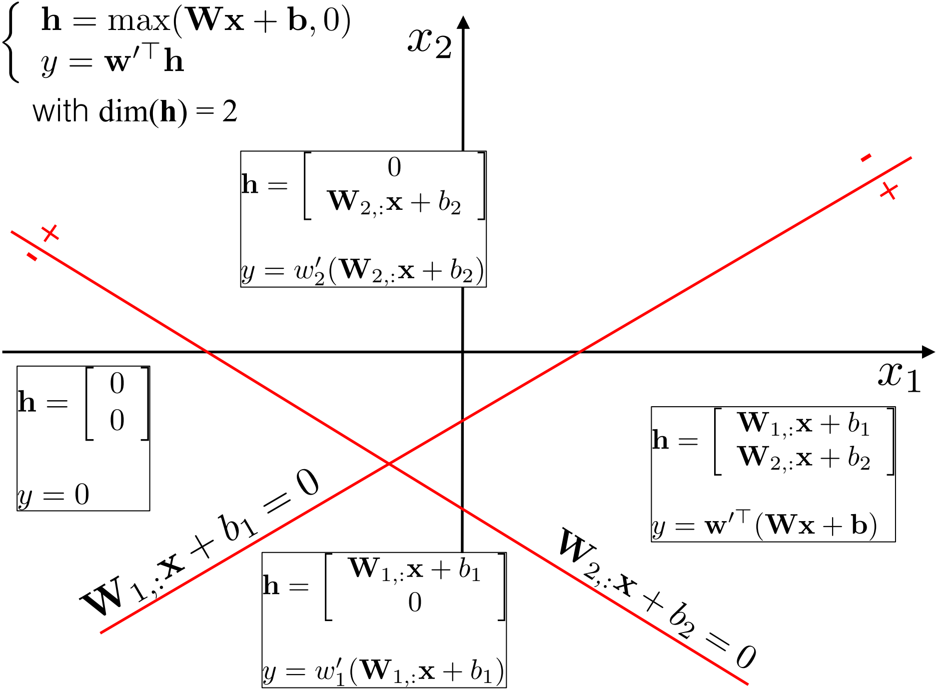
$y(\mathbf{x})$ can be rewritten:

$$y = \mathbf{w}_2 \mathbf{B}(\mathbf{x})(\mathbf{W}\mathbf{x} + \mathbf{b})$$

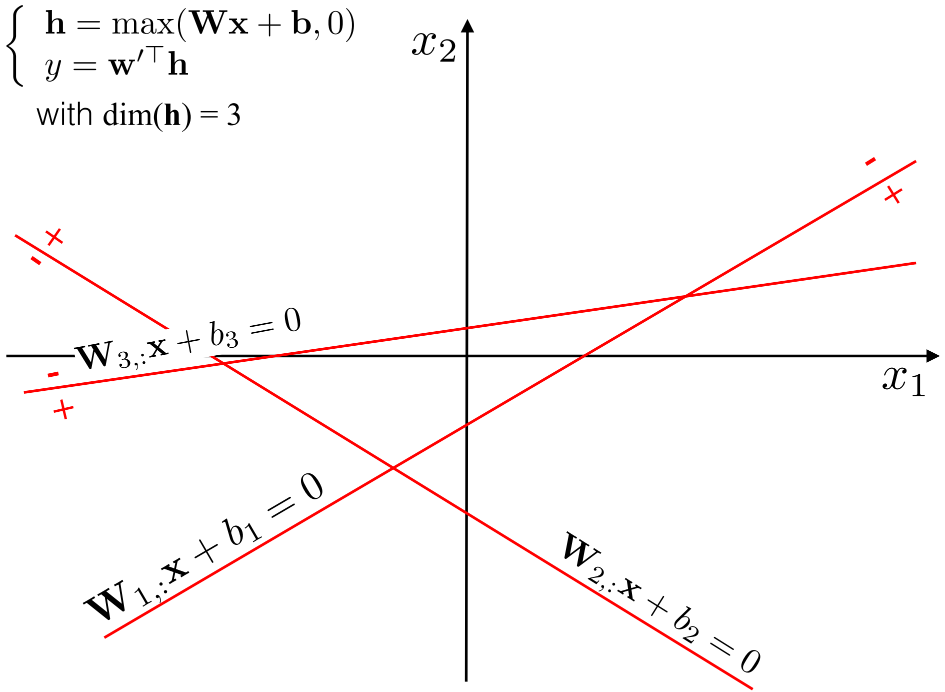
The function $\mathbf{x} \mapsto \mathbf{B}(\mathbf{x})$ is piecewise constant.

Thus $y(\mathbf{x})$ is piecewise affine.


$y = \max(\mathbf{w}^\top \mathbf{x} + b, 0)$
x_2
$y = 0$
-
+
x_1
$\mathbf{w}^\top \mathbf{x} + b = 0$
$y = \mathbf{w}^\top \mathbf{x} + b$



$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ y = \mathbf{w}'^\top \mathbf{h} \end{cases}$
with $\dim(\mathbf{h}) = 2$
x_2
x_1
$\mathbf{h} = \begin{bmatrix} 0 \\ \mathbf{W}_{2,:}\mathbf{x} + b_2 \end{bmatrix}$
$y = w'_2(\mathbf{W}_{2,:}\mathbf{x} + b_2)$
$\mathbf{h} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$y = 0$
$\mathbf{h} = \begin{bmatrix} \mathbf{W}_{1,:}\mathbf{x} + b_1 \\ \mathbf{W}_{2,:}\mathbf{x} + b_2 \end{bmatrix}$
$y = \mathbf{w}'^\top(\mathbf{W}\mathbf{x} + \mathbf{b})$
$\mathbf{h} = \begin{bmatrix} \mathbf{W}_{1,:}\mathbf{x} + b_1 \\ 0 \end{bmatrix}$
$y = w'_1(\mathbf{W}_{1,:}\mathbf{x} + b_1)$
$\mathbf{W}_{1,:}\mathbf{x} + b_1 = 0$
$\mathbf{W}_{2,:}\mathbf{x} + b_2 = 0$
+
-
+
-



$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ y = \mathbf{w}'^\top \mathbf{h} \end{cases}$
with $\dim(\mathbf{h}) = 3$
x_2
x_1
$\mathbf{W}_{1,:}\mathbf{x} + b_1 = 0$
$\mathbf{W}_{2,:}\mathbf{x} + b_2 = 0$
$\mathbf{W}_{3,:}\mathbf{x} + b_3 = 0$
+
-

- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;
- The structure of a 2-layer network function;
- The limitations of 2-layer networks, and the motivation for multi-layer networks;

Universal Approximation Theory for Two-Layer Networks

A two-layer network can be written as:

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \hat{f}(\mathbf{x}) = \mathbf{w}_2^\top \mathbf{h}(\mathbf{x}) + b_2 \end{cases}$$

or

$$\hat{f}(\mathbf{x}) = \sum_{j=1}^{N} c_j g(\mathbf{W}_{(j)}^\top \mathbf{x} - b_j)\,,$$

where $g : \mathbb{R} \to \mathbb{R}$ is an activation function and N is the number of hidden units.

N (or equivalently $\mathbf{h}$) may need to be large.

Deeper networks can mitigate this problem.

Multi-Layer Networks

2-layer network:

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{o}(\mathbf{x}) = \mathbf{W}_2^\top \mathbf{h}(\mathbf{x}) + \mathbf{b}_2 \end{cases}$$

3-layer network:

$$\begin{cases} \mathbf{h}_1(\mathbf{x}) = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2(\mathbf{h}_1) = g(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \\ \mathbf{o}(\mathbf{x}) = \mathbf{W}_3^\top \mathbf{h}_2 + \mathbf{b}_3 \end{cases}$$

etc.

Universal Approximation Theory for Deep Networks

The approximation theory for multilayer neural nets is less understood compared with neural nets with one hidden layer.

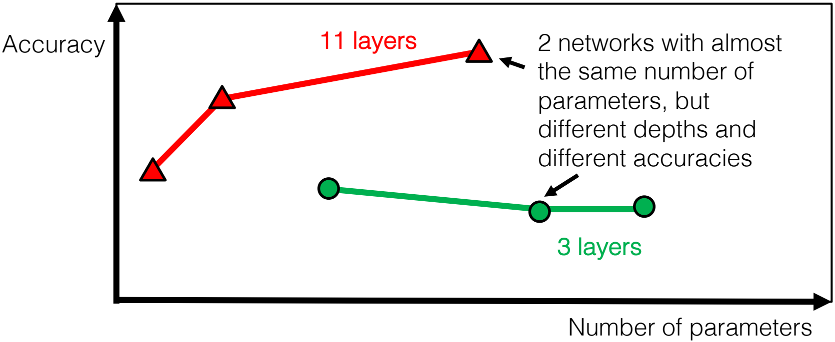
Deep neural nets excel at representing a composition of functions.

D. Rolnick and M. Tegmark. "The Power of Deeper Networks for Expressing Natural Functions". In: *arXiv Preprint*. 2017.

T. Poggio et al. "Why and When Can Deep-But Not Shallow-Networks Avoid the Curse of Dimensionality: A Review". In: *International Journal of Automation and Computing* (2017).

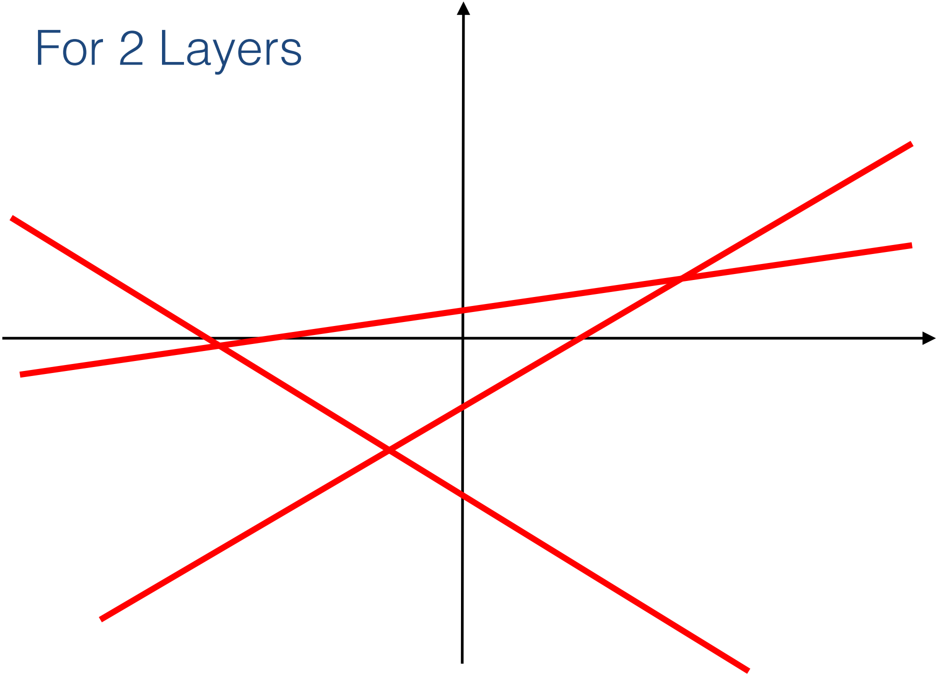
D. Rolnick and M. Tegmark. "The Power of Deeper Networks for Expressing Natural Functions". In: *arXiv Preprint*. 2017.

Deeper Networks Perform Better for a Given Number of Parameters in Practice

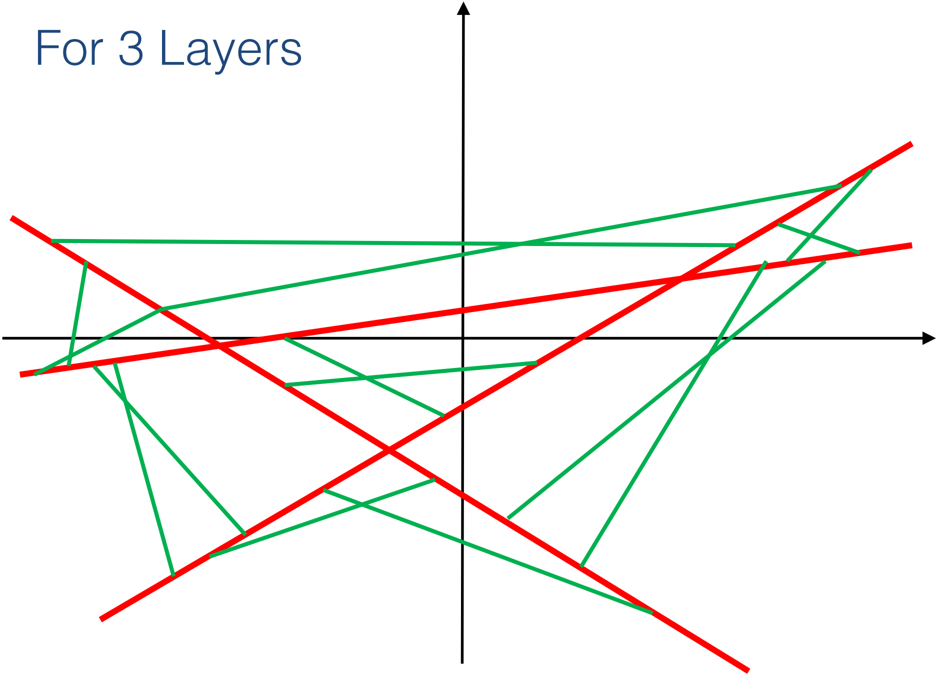


I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. 2016.

For 2 Layers



For 3 Layers



Number of Regions Generated by a Two-Layer Network

Maximum number of pieces into which r hyperplanes disconnect the space $\mathbb{R}^n$:

$$\sum_{i=0}^{n} \binom{r}{i}.$$

A 2-layer network:

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{y}(\mathbf{x}) = \mathbf{W}_2^\top \mathbf{h}(\mathbf{x}) + \mathbf{b}_2 \end{cases}$$

generates at most:

$$\sum_{i=0}^{|\mathbf{x}|} \binom{|\mathbf{h}|}{i}.$$

G. Montufar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *NIPS*. 2014.

Number of Regions Generated by a Three-Layer Network

A 3-layer network:

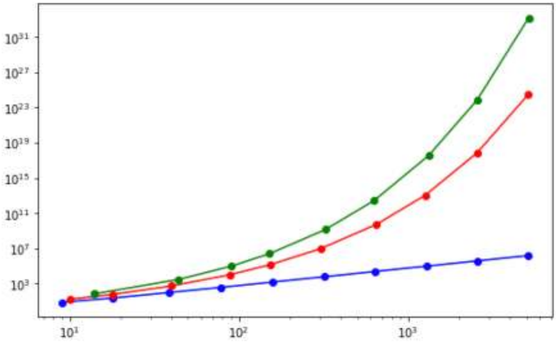
$$\begin{cases} \mathbf{h}_1(\mathbf{x}) = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2(\mathbf{h}_1) = g(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \\ \mathbf{y}(\mathbf{x}) = \mathbf{W}_3^\top \mathbf{h}_2 + \mathbf{b}_3 \end{cases}$$

generates at most:

$$\left(\sum_{i=0}^{|\mathbf{x}|} \binom{|\mathbf{h}_1|}{i} \right) \left(\sum_{i=0}^{|\mathbf{h}_1|} \binom{|\mathbf{h}_2|}{i} \right).$$

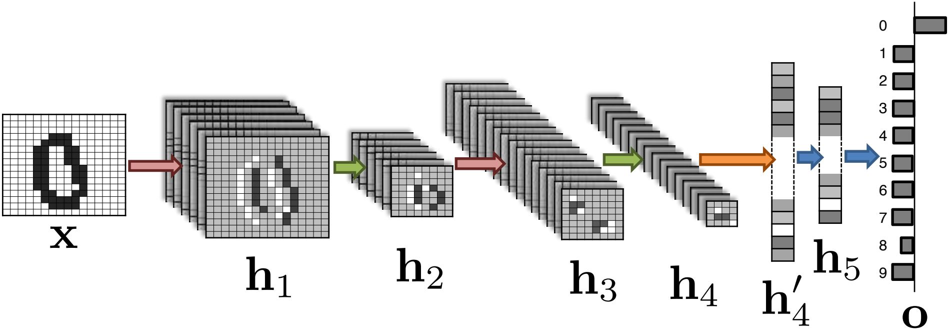
Number of Regions as a Function of the Number of Network Parameters for Different Depths

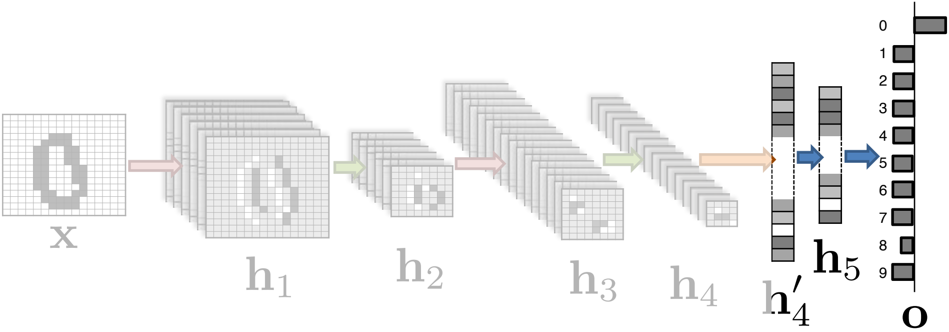
For a given number of layers D, and a given number of parameters P, set dims of all the $\mathbf{h}$ vectors to $\left\lceil \frac{P}{D} \right\rceil$



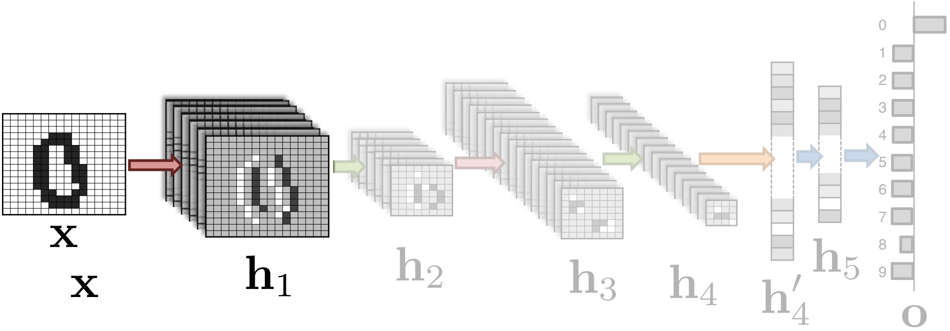
- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;
- The structure of a 2-layer network function;
- The limitations of 2-layer networks, and multi-layer networks;
- Dealing with images;

Dealing with Images

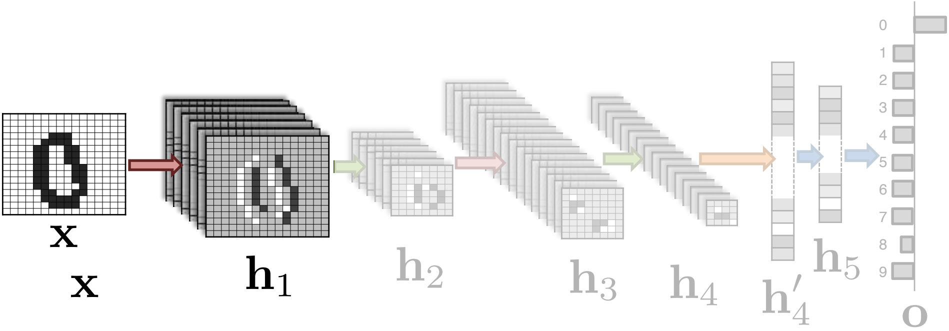




$$\mathbf{h}_5 = g(\mathbf{W}_5\mathbf{h}_4' + \mathbf{b}_5)$$
$$\mathbf{o} = \mathbf{W}_6\mathbf{h}_5 + \mathbf{b}_6$$



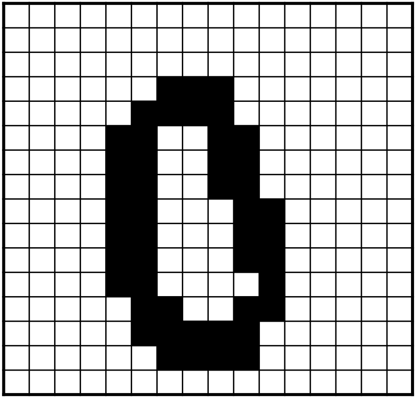
$$\cancel{\mathbf{h}_1 = g(\mathbf{W}\mathbf{x} + \mathbf{b})}$$



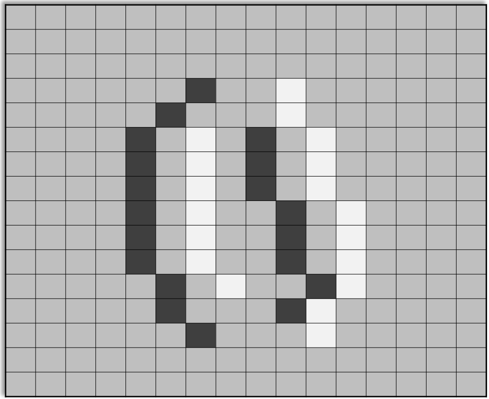
Product of convolution: $\mathbf{h}_{1,1} = g(\mathbf{f}_{1,1} * \mathbf{x} + \mathbf{b}_{1,1})$

$$\mathbf{h}_1 = [g(\mathbf{f}_{1,1} * \mathbf{x}), \ldots, g(\mathbf{f}_{1,m} * \mathbf{x})]$$

Convolution: Example



$\mathbf{x}$

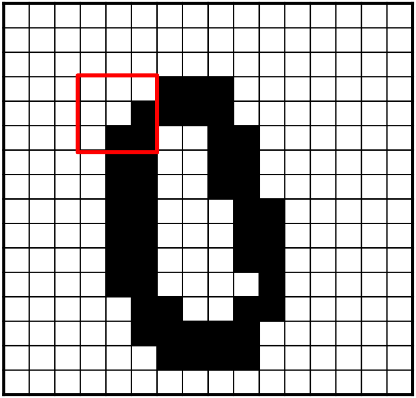


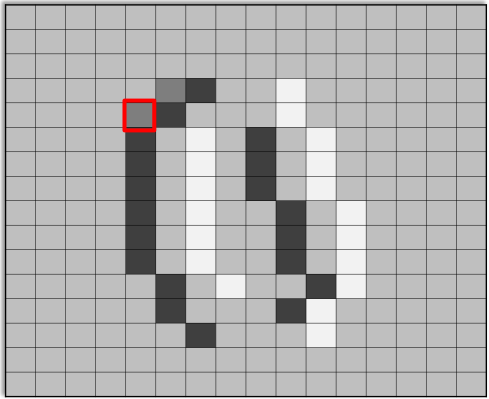
$\mathbf{h}_{1,1}$

$$\mathbf{f}_{1,1} = \begin{array}{|c|c|c|} \hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \end{array}$$

$$\mathbf{h}_{1,1} = g(\mathbf{f}_{1,1} * \mathbf{x} + \mathbf{b}_{1,1})$$

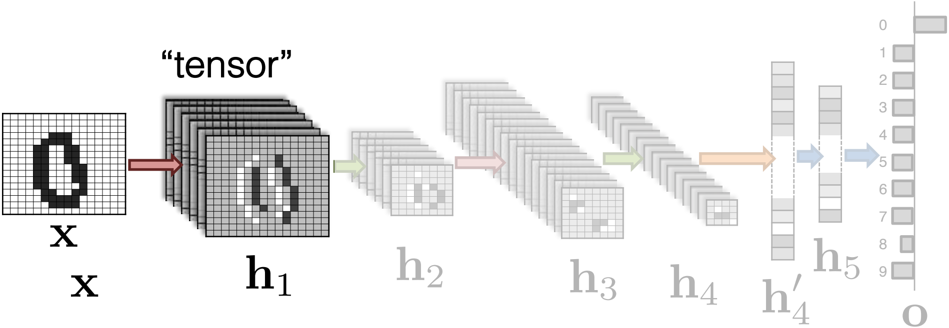
Numerical Example





$$\mathbf{f}_{1,1} = \begin{array}{|c|c|c|}\hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \mathbf{-1} & \mathbf{0} & \mathbf{+1} \\ \hline \end{array}$$

$$\begin{aligned} h &= (-1) \times 255 + 0 \times 255 + (+1) \times 255 + \\ &\quad (-1) \times 255 + 0 \times 255 + (+1) \times 0 + \\ &\quad (-1) \times 255 + 0 \times 0 + (+1) \times 0 \\ &= -255 + 0 + 255 \\ &\quad -255 + 0 + 0 \\ &\quad -255 + 0 + 0 \\ &= -510 \end{aligned}$$

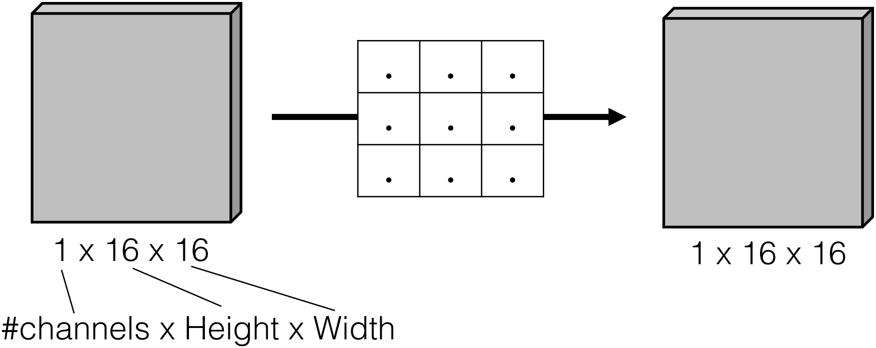


Product of convolution: $\mathbf{h}_{1,1} = g(\mathbf{f}_{1,1} * \mathbf{x} + \mathbf{b}_{1,1})$

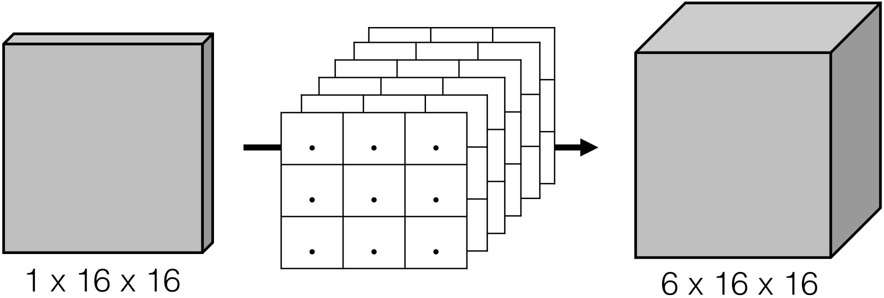
$$\mathbf{h}_1 = [g(\mathbf{f}_{1,1} * \mathbf{x}), \ldots, g(\mathbf{f}_{1,m} * \mathbf{x})]$$

"tensor"

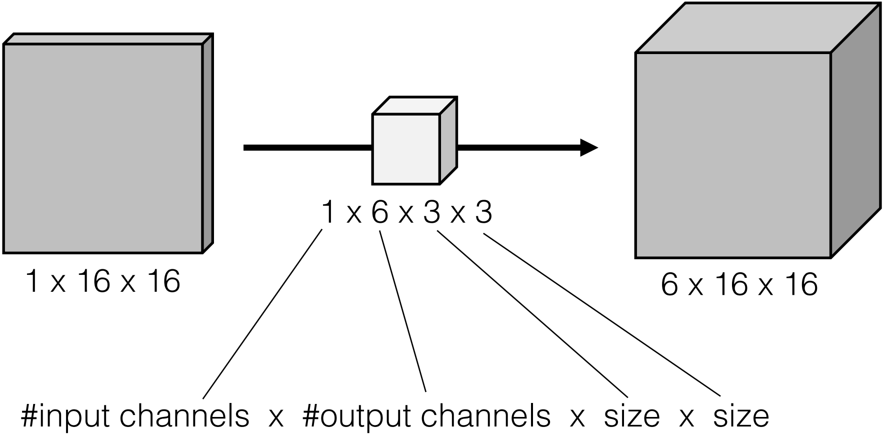
Tensors



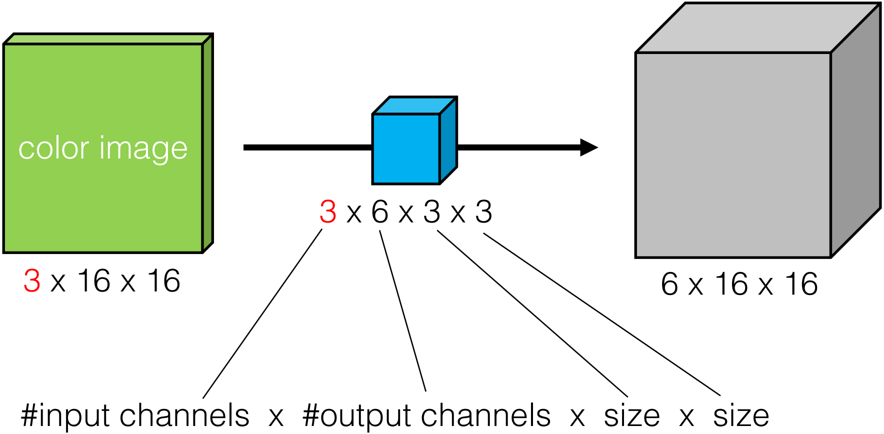
Tensors

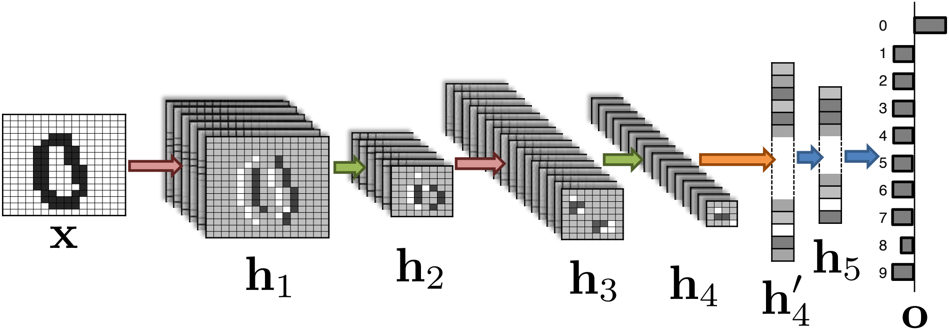


Tensors


1 x 16 x 16
1 x 6 x 3 x 3
6 x 16 x 16
#input channels x #output channels x size x size

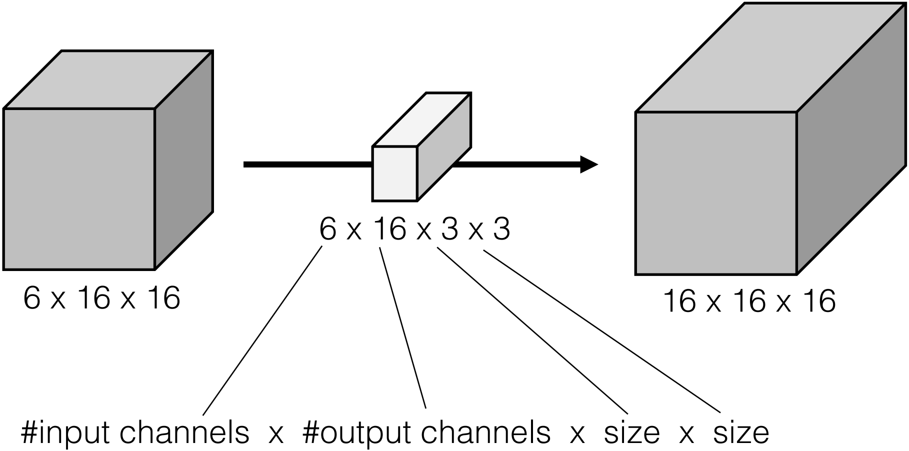
Tensors


color image
3 x 16 x 16
3 x 6 x 3 x 3
6 x 16 x 16
#input channels x #output channels x size x size

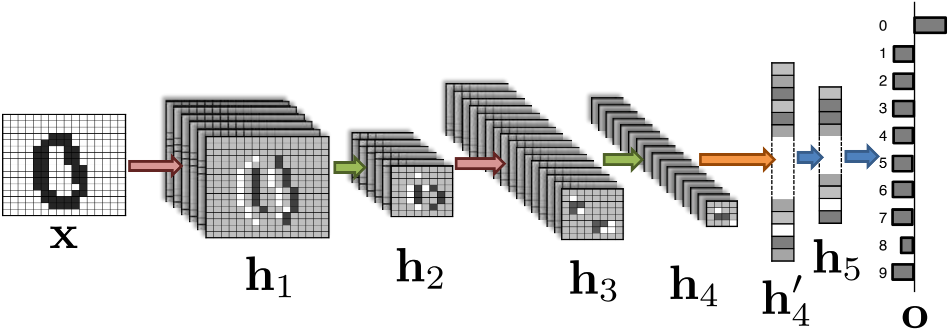


x
h1
h2
h3
h4
h'4
h5
o
0
1
2
3
4
5
6
7
8
9

Tensors

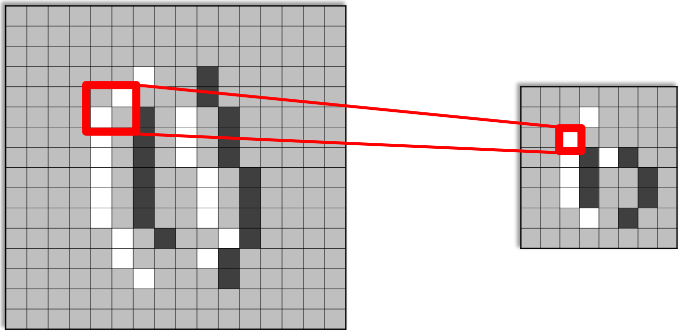


6 x 16 x 16
6 x 16 x 3 x 3
16 x 16 x 16
#input channels x #output channels x size x size



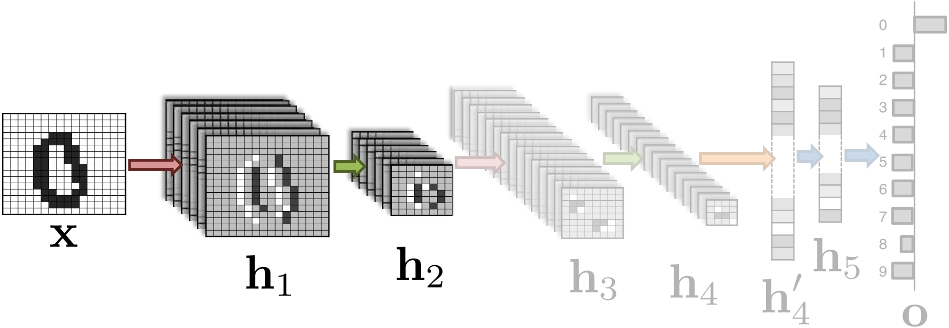
x
h1
h2
h3
h4
h'4
h5
o
0
1
2
3
4
5
6
7
8
9

Subsampling / Pooling



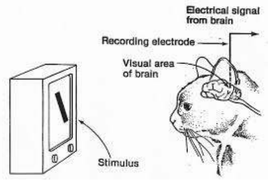
For example, max-pooling:

$$\mathbf{h}_i[u,v] = \max\{ \quad \mathbf{h}_{i-1}[2u, \quad 2v], \quad \mathbf{h}_{i-1}[2u, \quad 2v+1], \quad \mathbf{h}_{i-1}[2u+1, \quad 2v], \quad \mathbf{h}_{i-1}[2u+1, \quad 2v+1] \quad \}$$



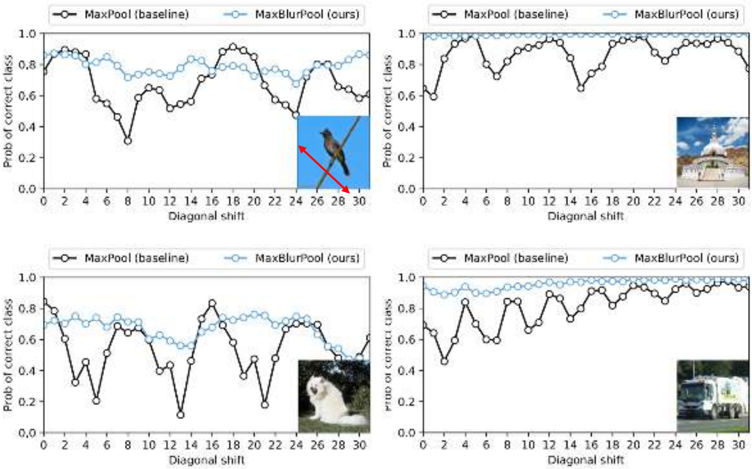
$$\mathbf{h}_1 = [g(\mathbf{f}_{1,1} * \mathbf{x}), \ldots, g(\mathbf{f}_{1,m} * \mathbf{x})]$$
$$\mathbf{h}_2 = \mathrm{pooling}(\mathbf{h}_1)$$

Inspired by the theory of Hubel and Wiesel on the visual cortex (Nobel prize in 1981)



Making Convolutional Networks Shift-Invariant Again

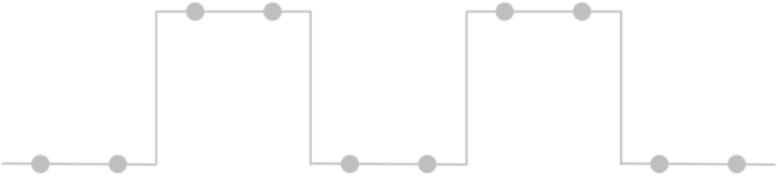
The Max Pooling operation, while popular, is actually not very robust to small shifts:



Making Convolutional Networks Shift-Invariant Again. Richard Zhang. ICML 2019.

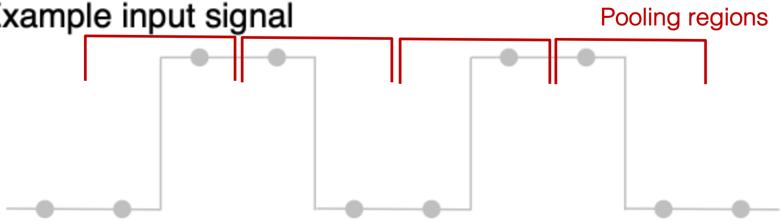
Shift-[In]variance: 1D Toy Problem

Example input signal

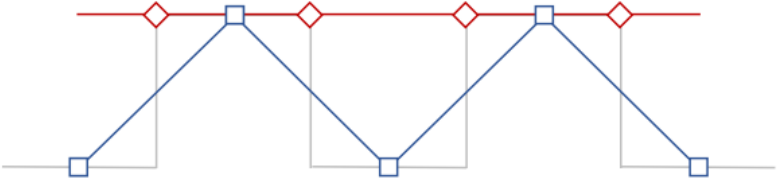


Shift-[In]variance: 1D Toy Problem

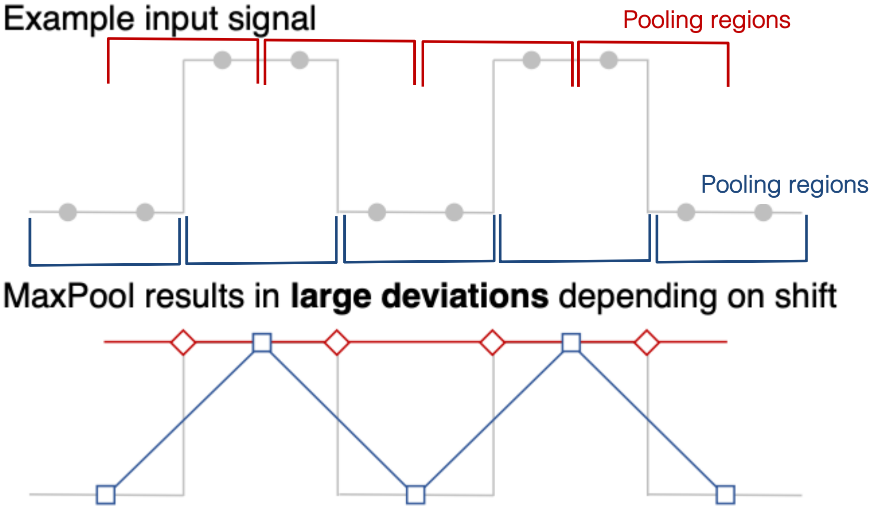
Example input signal



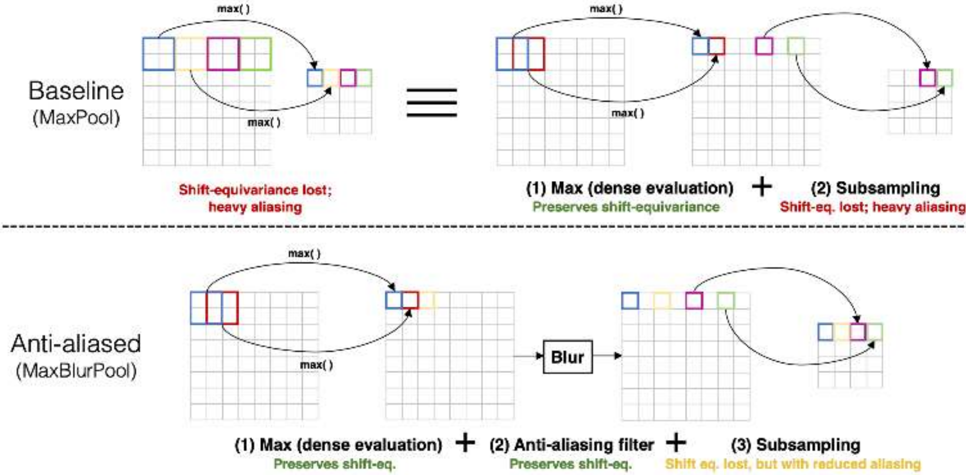
MaxPool results in **large deviations** depending on shift



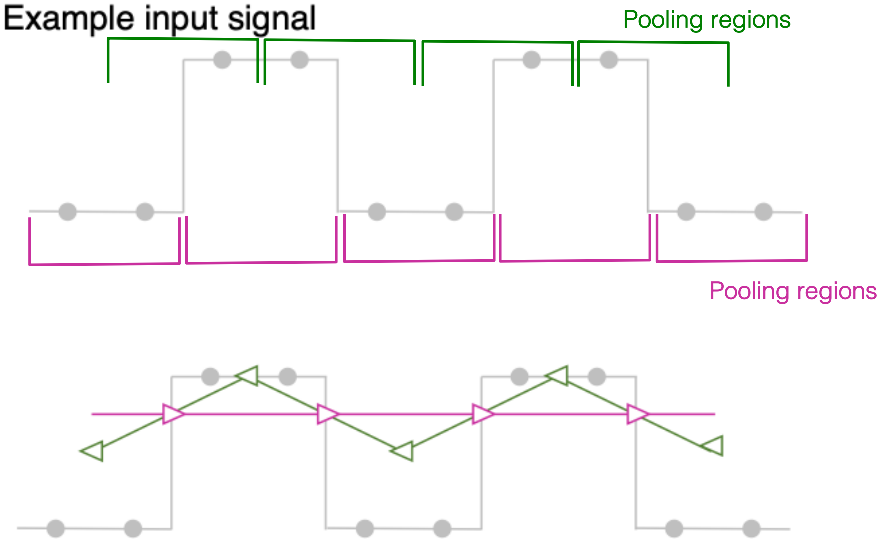
Shift-[In]variance: 1D Toy Problem

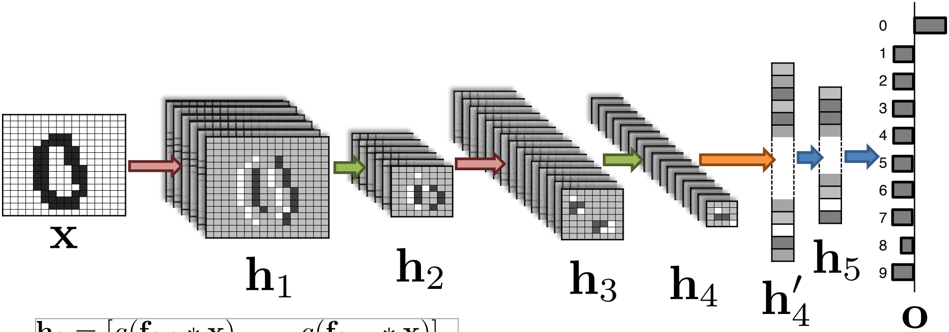


Antialising MaxPooling



Antialising MaxPooling



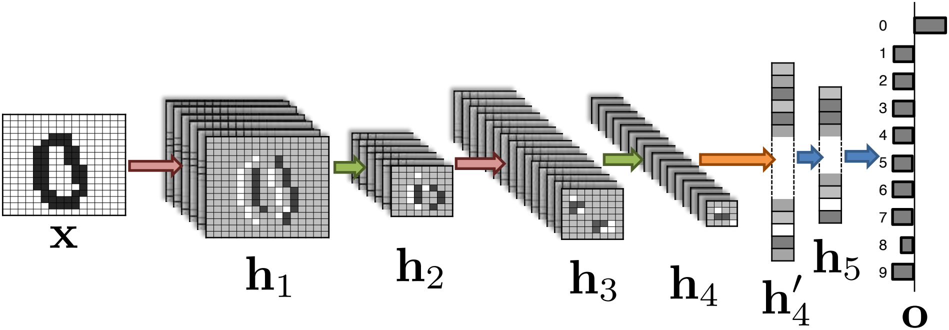


$$
\begin{aligned}
\mathbf{h}_1 &= [g(\mathbf{f}_{1,1} * \mathbf{x}), \ldots, g(\mathbf{f}_{1,m} * \mathbf{x})] \\
\mathbf{h}_2 &= \mathrm{pooling}(\mathbf{h}_1) \\
\mathbf{h}_3 &= [g(\mathbf{f}_{3,1} * \mathbf{h}_2), \ldots, g(\mathbf{f}_{3,n} * \mathbf{h}_2)] \\
\mathbf{h}_4 &= \mathrm{pooling}(\mathbf{h}_3) \\
\mathbf{h}'_4 &= \mathrm{Vec}(\mathbf{h}_4) \\
\mathbf{h}_5 &= g(\mathbf{W}_5 \mathbf{h}'_4 + \mathbf{b}_5) \\
\mathbf{o} &= \mathbf{W}_6 \mathbf{h}_5 + \mathbf{b}_6
\end{aligned}
$$

convolutional layers

pooling layers

fully-connected layers



By contrast with other Computer Vision models:

- CNNs retain spatial information (compare with Bags-of-Words for example);
- CNNs do not need engineered features (compare with Histograms of Gradients for example).

- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;
- The structure of a 2-layer network function;
- The limitations of 2-layer networks, and multi-layer networks;
- Dealing with images;
- How do we find the parameters of a Deep Network?

Finding the Parameters

$$\begin{cases} \mathbf{h}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{o}(\mathbf{x}) = \mathbf{W}_2\mathbf{h}(\mathbf{x}) + \mathbf{b}_2 \end{cases}$$

How can we find $\mathbf{W}$, $\mathbf{b}$, $\mathbf{W}_2$, *and* $\mathbf{b}_2$?

→ By minimizing a loss function. The loss function can be adapted to the problem.

$$(\widehat{\mathbf{W}}, \widehat{\mathbf{b}}, \widehat{\mathbf{W}_2}, \widehat{\mathbf{b}_2}) = \arg\min_{(\mathbf{W}, \mathbf{b}, \mathbf{W}_2, \mathbf{b}_2)} \mathcal{L}(\mathbf{W}, \mathbf{b}, \mathbf{W}_2, \mathbf{b}_2)$$

Optimisation: (Variants of) gradient descent.

Example of Loss Function

For example:

$$\mathcal{L}(\mathbf{W}, \mathbf{b}, \mathbf{W}_2, \mathbf{b}_2) = - \sum_{(\mathbf{x},d)\in\mathcal{T}} \log \left(\frac{\exp(\mathbf{o}(\mathbf{x})_d)}{\sum_i \exp(\mathbf{o}(\mathbf{x})_i)} \right)$$

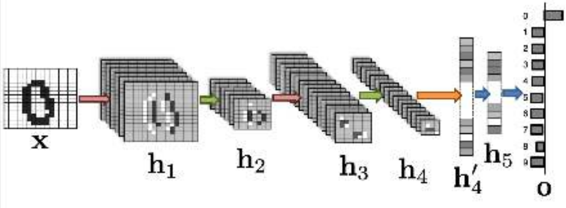
Training sample

( , 2)

Predicted output for the training sample

Training set $\mathcal{T}$

How to Choose the HyperParameters (Number of Layers, Number of Filters, Sizes of the Filters)?

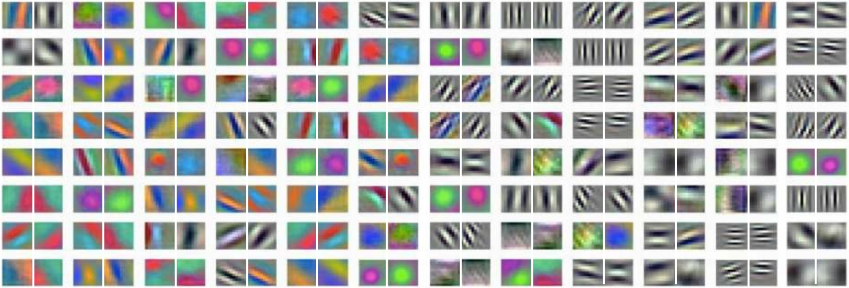


- In practice, often from previous experience...;
- Automatically, using 'AutoML'.

Xin He, Kaiyong Zhao, Xiaowen Chu. AutoML: A Survey of the State-of-the-Art. arXiv 2019.

- The Perceptron: A 1-layer “network”;
- A 2-layer network;
- How does a 2-layer network “work”?
- The power of 2-layer networks;
- The structure of a 2-layer network function;
- The limitations of 2-layer networks, and multi-layer networks;
- Dealing with images;
- How do we find the parameters of a Deep Network?
- What does a Deep Network compute internally?

Learned Filters for the First Layer for Natural Images



$$\{\mathbf{f}_{1,j}\}_j$$

Images that Generate High
Values for a Neuron in Layer 2

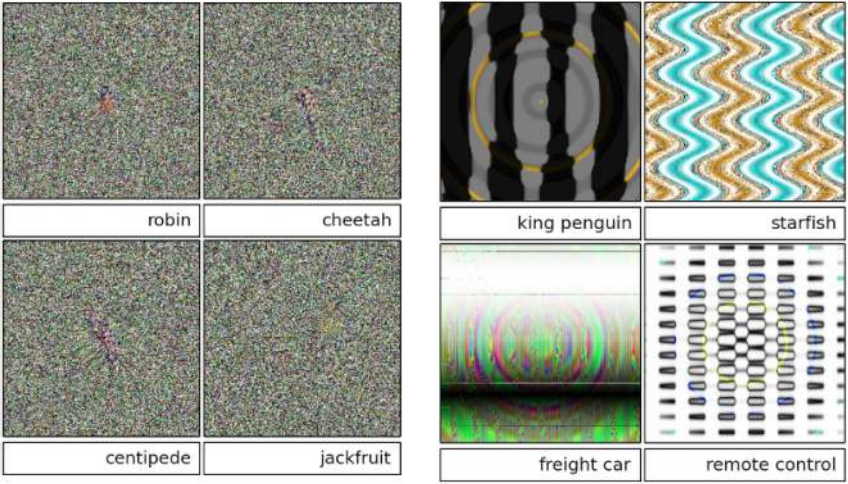
Images that Generate High
Values for a Neuron in Layer 3

Current Limits of Deep Learning

... "the inconvenient truth" is that at present the algorithms that feature prominently in research literature are in fact not, for the most part, executable at the frontlines of clinical practice.

Panch19.

Images with High Confidence Predictions



Adversarial Examples

A. Nguyen, J. Yosinski, and J. Clune. *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images*. CVPR 2015.

Other Adversarial Examples

$$\min_{\mathbf{r}} \|\mathbf{r}\|_2 \text{ subject to } c(\mathbf{x}+\mathbf{r}) \neq c(\mathbf{x})$$

where

- $\mathbf{x}$ is an image and
- $c(.)$ is the class predicted by an already trained network.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Pascal Frossard. *DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks*. CVPR 2016.

Other Adversarial Examples

$$\min_{\mathbf{r}} \|\mathbf{r}\|_2 \text{ subject to } c(\mathbf{x}+\mathbf{r}) \neq c(\mathbf{x})$$

where

- $\mathbf{x}$ is an image and
- $c(.)$ is the class predicted by an already trained network.

$\mathbf{r}$

$\mathbf{x}+\mathbf{r}$

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Pascal Frossard. *DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks*. CVPR 2016.

Other Adversarial Examples

$$\min_{\mathbf{r}} \|\mathbf{r}\|_2 \text{ subject to } c(\mathbf{x}+\mathbf{r}) \neq c(\mathbf{x})$$

where

- $\mathbf{x}$ is an image and
- $c(.)$ is the class predicted by an already trained network.

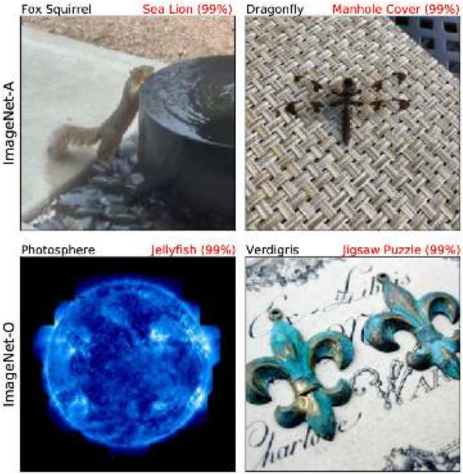
$\mathbf{r}$

$\mathbf{x}+\mathbf{r}$

Predicted class: 'Indian elephant'

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Pascal Frossard. *DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks*. CVPR 2016.

Natural Adversarial Examples



Natural Adversarial Examples. Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, Dawn Song. CVPR 2021.

recognizing objects

Natural Adversarial Examples. Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, Dawn Song. CVPR 2021.

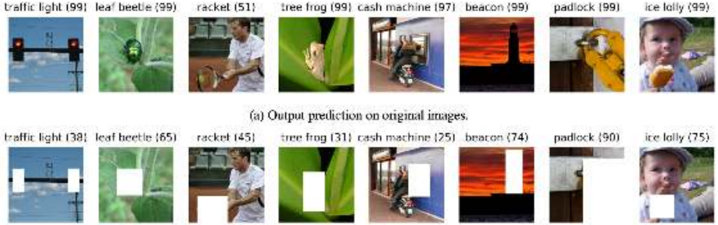
recognizing objects

(a) Output prediction on original images.

(b) Prediction when foreground is whitened.

Natural Adversarial Examples. Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, Dawn Song. CVPR 2021.

recognizing objects



(a) Output prediction on original images.

(b) Prediction when foreground is whitened.

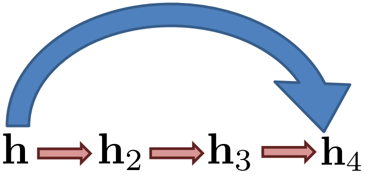
Natural Adversarial Examples. Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, Dawn Song. CVPR 2021.

“Modern” Deep Learning

Skip Connections

For example (Residual module):

$$\mathbf{h}_2 = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$$
$$\mathbf{h}_4 = g(\mathbf{h} + \mathbf{h}_3)$$



Skip Connections

For example (Residual module):

$$\begin{aligned}\mathbf{h}_2 &= g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)\\ \mathbf{h}_3 &= \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3\\ \mathbf{h}_4 &= g(\mathbf{h} + \mathbf{h}_3)\end{aligned}$$

$$\begin{tikzcd} \mathbf{h} \arrow[r] \arrow[rrr, bend left] & \mathbf{h}_2 \arrow[r] & \mathbf{h}_3 \arrow[r] & \mathbf{h}_4 \end{tikzcd}$$

Does not create a larger set of neural networks. The following network has the standard structure and computes the same output:

$$\begin{aligned}\mathbf{h}'_2 &= g\left(\begin{bmatrix}\mathbf{W}_2\\ \mathbf{I}\\ -\mathbf{I}\end{bmatrix}\mathbf{h} + \begin{bmatrix}\mathbf{b}_2\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}\right)\\ &= \begin{bmatrix}g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)\\ g(\mathbf{h})\\ g(-\mathbf{h})\end{bmatrix}\end{aligned}$$

because

$$g(a) - g(-a) = a$$

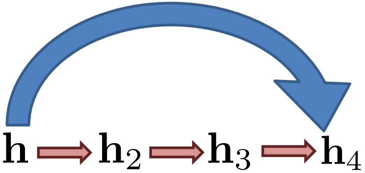
(for the case $g(a) = \max(0, a)$)

$$\begin{aligned}\mathbf{h}'_3 &= \begin{bmatrix}\mathbf{W}_3 & \mathbf{I} & -\mathbf{I}\end{bmatrix}\mathbf{h}'_2 + \mathbf{b}_3\\ &= \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3 + g(\mathbf{h}) - g(-\mathbf{h})\\ &= \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3 + \mathbf{h} = \mathbf{h}_3 + \mathbf{h}\\ \mathbf{h}'_4 &= g(\mathbf{h}'_3) = \mathbf{h}_4\end{aligned}$$

Skip Connections

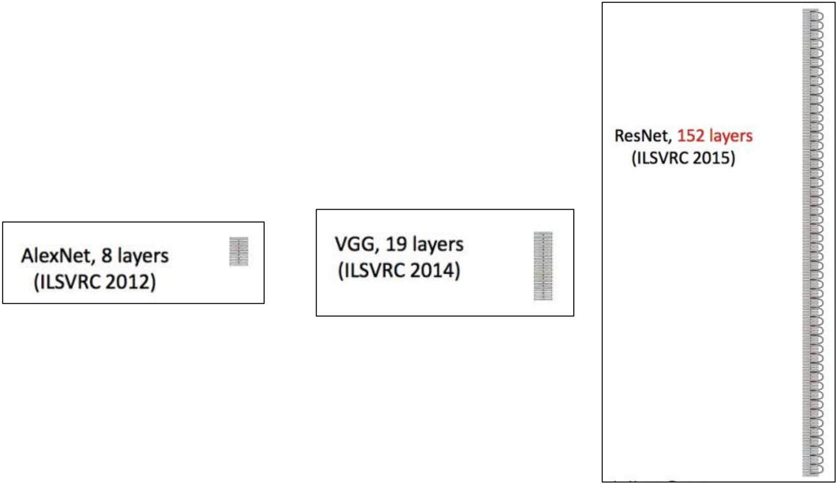
For example (Residual module):

$$\mathbf{h}_2 = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$$
$$\mathbf{h}_4 = g(\mathbf{h} + \mathbf{h}_3)$$

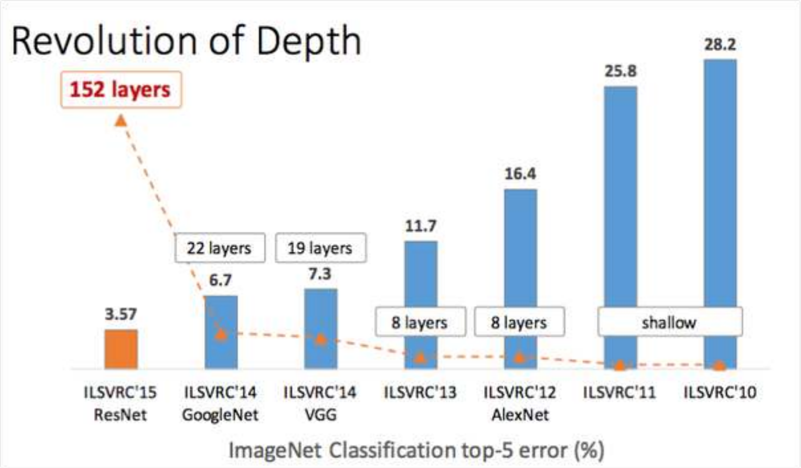


- Limits vanishing and exploding gradients.

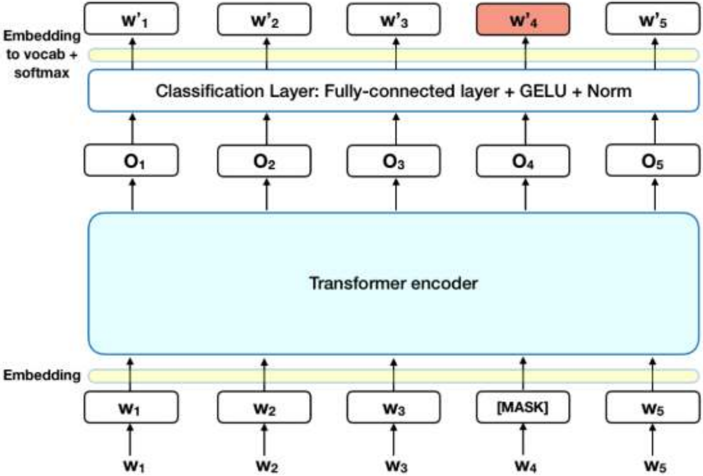
ResNet [He et al, CVPR 2016]



ResNet [He et al, CVPR 2016]



Transformers



BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. 2019.

Optimization Algorithms and Tricks

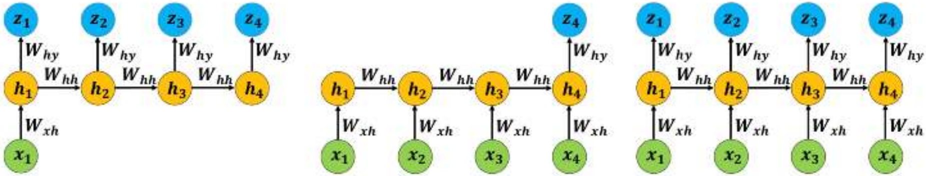
- Stochastic Gradient Descent, momentum, Adam, etc.
- Batch normalization, DropOut, etc.
- Data augmentation, etc.
- Multi task training, ...

Python Libraries

TensorFlow, Keras, PyTorch, ..

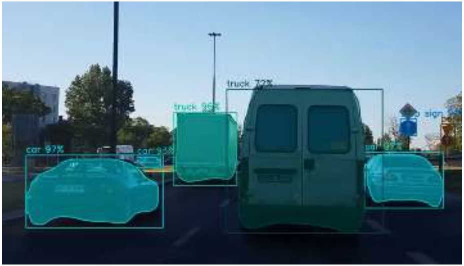
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
from keras.layers import Flatten
model.add(Flatten())

from keras.layers import Dense
model.add(Dense(128, activation='relu'))

model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X_train, Y_train, batch_size=32, epochs=10, verbose=1)
```

Recurrent Networks



(a) One-to-many (b) Many-to-one (c) Many-to-many

Smart Use of Deep Learning to Solve Specific Problems



How can we formalize these problems in order to solve them with Deep Learning?

Beyond Supervised Learning

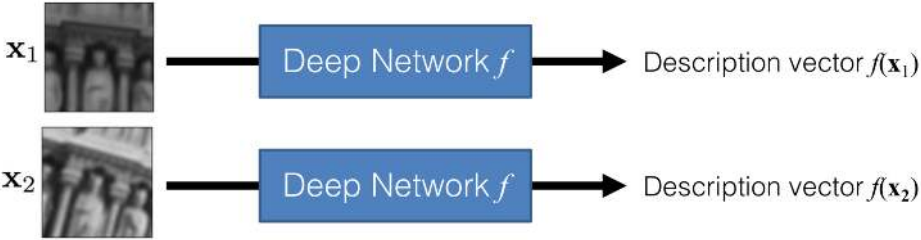
- We can use a Deep Network to approximate any continuous function;

$$\mathbf{x} \longrightarrow \boxed{\text{Deep Network } f} \longrightarrow \mathbf{o}$$

- We can use any loss function as long as it is differentiable;

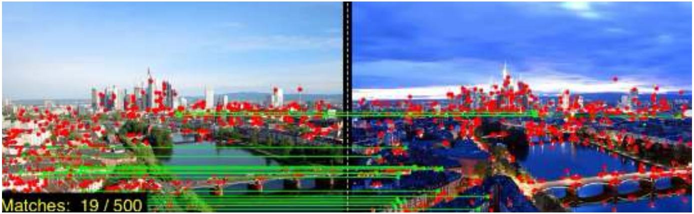
→ very flexible!

Siamese Networks



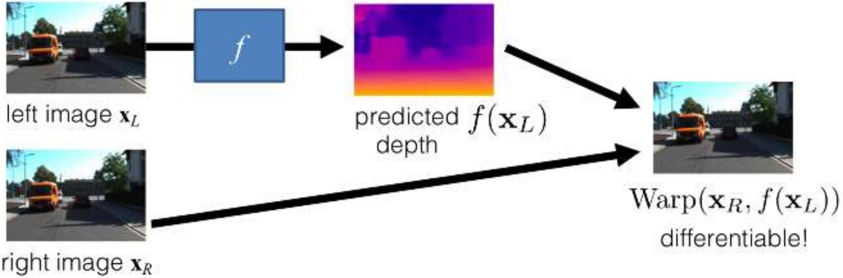
Loss function:

- minimize the distance $\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|$ for samples $\mathbf{x}_1$, $\mathbf{x}_2$ that correspond to each other;
- maximize the distance $\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|$ for samples $\mathbf{x}_1$, $\mathbf{x}_2$ that don't.



Self Learning: Case of Unsupervised Depth Prediction

Unsupervised Depth Estimation

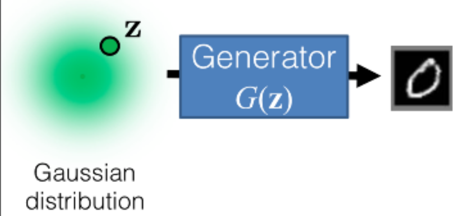


Loss function:

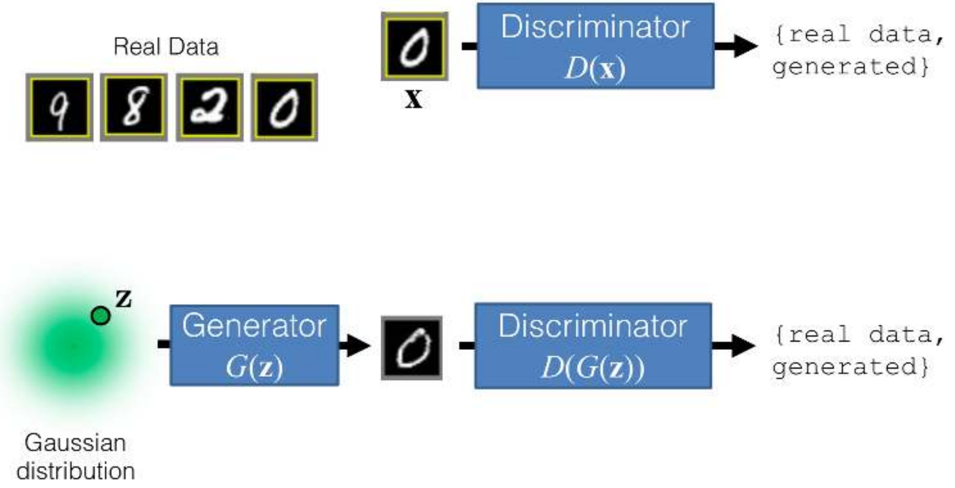
$$\mathcal{L} = \sum_{(\mathbf{x}_L, \mathbf{x}_R)} \|\mathbf{x}_L - \mathrm{Warp}(\mathbf{x}_R, f(\mathbf{x}_L))\|^2$$

Generative Adversarial Networks

We would like to train a network G to generate images of digits from noise vectors $\mathbf{z}$:



Generative Adversarial Networks



Generative Adversarial Networks

Generated Images

DeepFakes are an extension of GANs

Sometimes used to generate training data

transfer learning / domain transfer

Transfer learning:

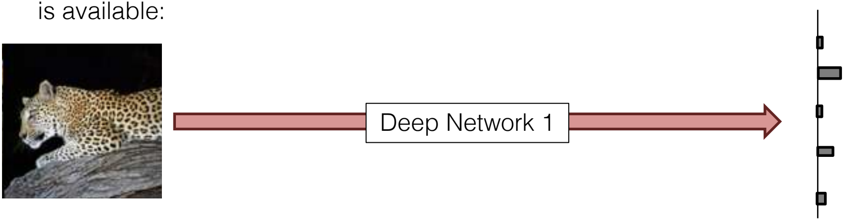
- we have few training data on our problem, but
- we have a lot of training data for a similar problem.

transfer learning / domain transfer

A simple method for transfer learning:

$\mathbf{x}$

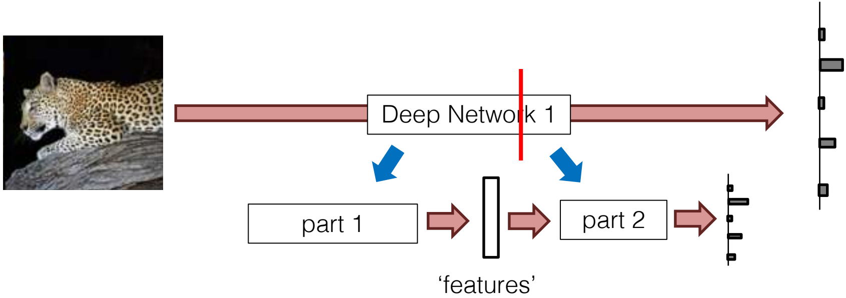
1. Training a deep network on a problem where a large amount of training data is available:



transfer learning / domain transfer

A simple method for transfer learning:

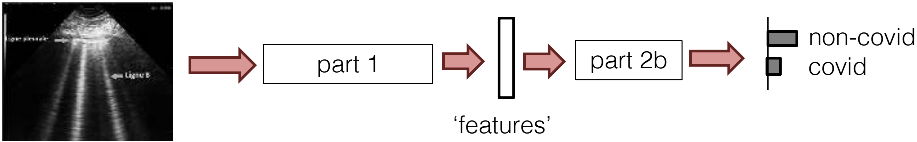
2. Cut this network into two parts (after training):



transfer learning / domain transfer

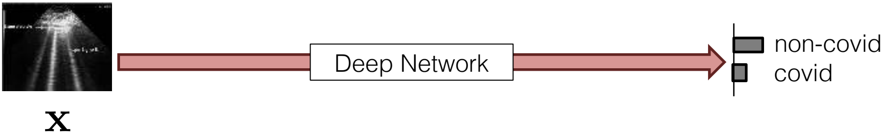
A simple method for transfer learning:

3. Keep the parameters of Part 1, initialize randomly Part 2b with the new number of classes

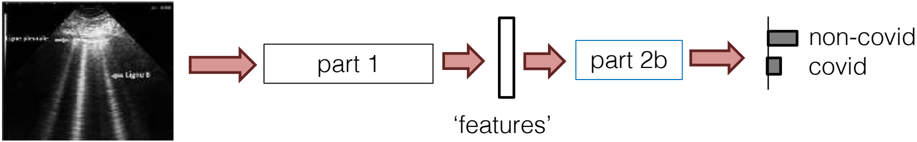


transfer learning / domain transfer

A simple method for transfer learning:

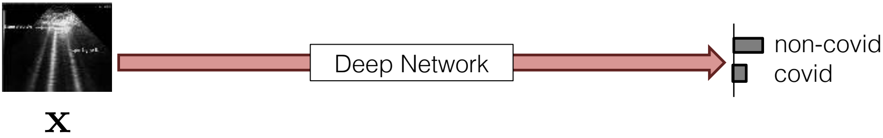


4. Keep the parameters of Part 1, optimize only the parameters of Part 2b on the available data
Alternatively, we can ‘fine-tune‘ the parameters of Part 1.



transfer learning / domain transfer

A simple method for transfer learning:



Part 1 and Part 2b form a deep network:

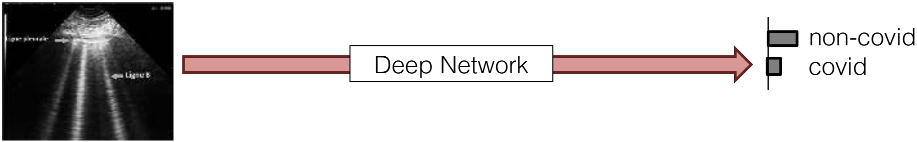
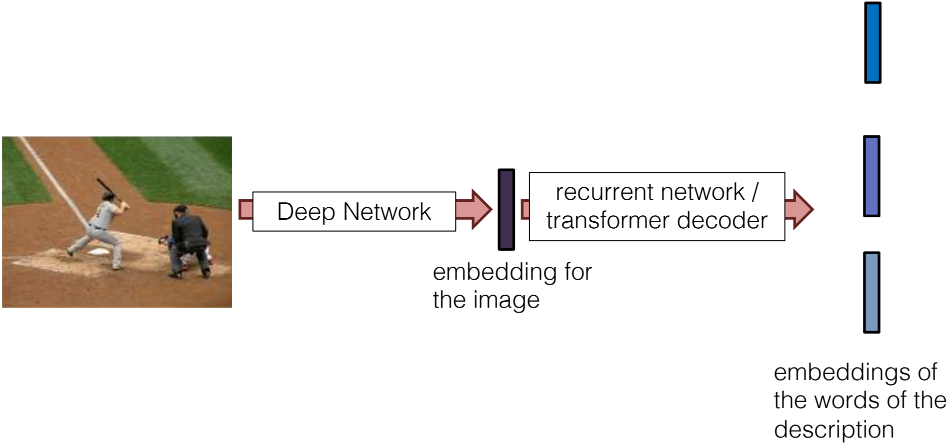
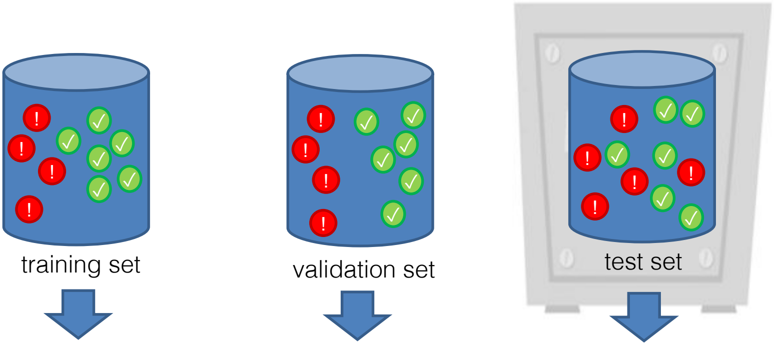


Image Captioning



How to Evaluate a Deep Network

training set, validation set, test set



use it to find the classifier (ie the separation between the classes)

use it to find the classifier’s hyperparameters

the performance on the validation set is an estimate of the performance on the test sett

use it to evaluate the classifier

Positive and negative samples

positive: A sample from the “positive” class (eg ‘at risk’);
negative: A sample from the “negative” class (eg ‘not at risk’);

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

The classification error rate considers the costs of false positives and false negatives to be the same.

This is not necessarily true, for example for a medical test.

→ We need finer metrics

new metrics

positive: A sample from the “positive” class;
negative: A sample from the “negative” class;

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

Classification error rate
= (# false positives + # false negatives) / # samples

new metrics

positive: A sample from the “positive” class;
negative: A sample from the “negative” class;

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

True Positive rate (TP) = # true positives / # positives

a number between 0 (worst) and 1 (best)

new metrics

positive: A sample from the “positive” class;
negative: A sample from the “negative” class;

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

False Positive rate (FP) = # false positives / # negatives

a number between 0 (best) and 1 (worst)

new metrics

positive: A sample from the “positive” class;
negative: A sample from the “negative” class;

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

True Negative rate (TN) = # true negatives / # negatives

a number between 0 (worst) and 1 (best)

new metrics

positive: A sample from the “positive” class;
negative: A sample from the “negative” class;

true positive: A positive sample classified as positive
false positive: A negative sample classified as positive

true negative: A negative sample classified as negative
false negative: A positive sample classified as negative

False Negative rate (FN) = # false negatives / # positives

a number between 0 (best) and 1 (worst)

new metrics

True Positive rate (TP) = # true positives / # positives
False Positive rate (FP) = # false positives / # negatives

True Negative rate (TN) = # true negatives / # negatives
False Negative rate (FN) = # false negatives / # positives

It is easy to have a True Positive rate equal to 1 (how?)

It is easy to have a True Negative rate equal to 1 (how?)

It is almost impossible to have a True Positive rate and a True Negative rate both equal to 1.

new metrics

True Positive rate (TP) = # true positives / # positives
False Positive rate (FP) = # false positives / # negatives

True Negative rate (TN) = # true negatives / # negatives
False Negative rate (FN) = # false negatives / # positives

It is easy to have a True Positive rate equal to 1 (how?)

It is easy to have a True Negative rate equal to 1 (how?)

It is almost impossible to have a True Positive rate and a True Negative rate both equal to 1.

Finding a good classifier is a balance between a good True Positive rate and a good True Negative rate. The acceptable values for TP and TN (or FP and FN) depend on the target application.

new metrics

True Positive rate (TP) = # true positives / # positives
False Positive rate (FP) = # false positives / # negatives

True Negative rate (TN) = # true negatives / # negatives
False Negative rate (FN) = # false negatives / # positives

It is easy to have a True Positive rate equal to 1 (how?)

It is easy to have a True Negative rate equal to 1 (how?)

It is almost impossible to have a True Positive rate and a True Negative rate both equal to 1.

We would like to have metrics that capture the balance between the different errors (and success) of the classifier.

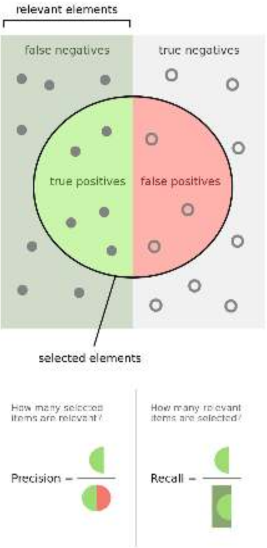
recall and precision

Additional metrics:

precision = # true positives / (# true positives + # false positives)

recall = True Positive rate
= # true positives / # positives
= # true positives / (# true positives + # false negatives)

recall and precision



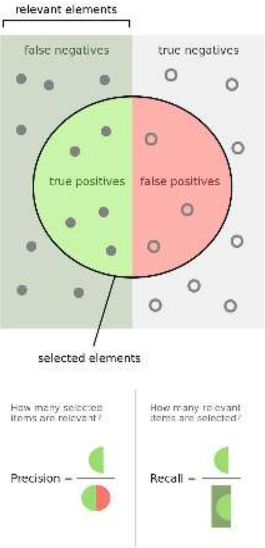
- precision =
true positives / (# true positives + # false positives)

- recall = True Positive rate

precision: proportion of samples predicted positive that are actually positive (between 0 and 1)

recall: proportion of the samples actually positive that are predicted positive (between 0 and 1)

recall and precision



- precision =
true positives / (# true positives + # false positives)

- recall = True Positive rate

precision: proportion of samples predicted positive that are actually positive (between 0 and 1)

recall: proportion of the samples actually positive that are predicted positive (between 0 and 1)

If the precision is high, we can trust the classifier when it predicts that a sample is positive.

If the recall is high, the classifier will correctly identify the positive samples (but maybe generate many false positives).