Neural Networks: Architecture Design

Instructor: Lei Wu [1]

Mathematical Introduction to Machine Learning

Peking University, Fall 2025

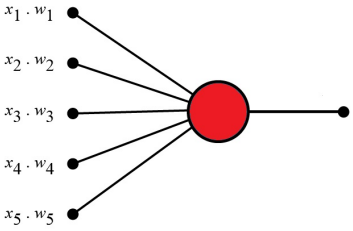
[1]School of Mathematical Sciences and Center for Machine Learning Research

Outline

McCulloch–Pitts Neuron (1943)

The first mathematical model of a neuron

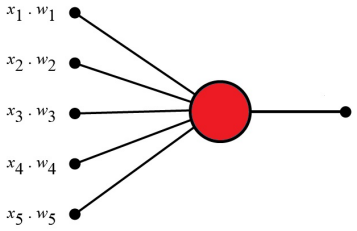
- Introduced a formal computational neuron: $f(\boldsymbol{x}) = \mathbf{1}\{\boldsymbol{w} \cdot \boldsymbol{x} + b > 0\}$.
- It can implement logical operations (e.g., AND, OR).
- Initiated the viewpoint that the brain can be modeled as a **computational system**.



McCulloch–Pitts Neuron (1943)

The first mathematical model of a neuron

- Introduced a formal computational neuron: $f(\boldsymbol{x}) = \mathbf{1}\{\boldsymbol{w} \cdot \boldsymbol{x} + b > 0\}$.
- It can implement logical operations (e.g., AND, OR).
- Initiated the viewpoint that the brain can be modeled as a **computational system**.



Impact:

- Provided the basic architectural template for modern neural networks.
- Logic-based model with **no learning rule**.

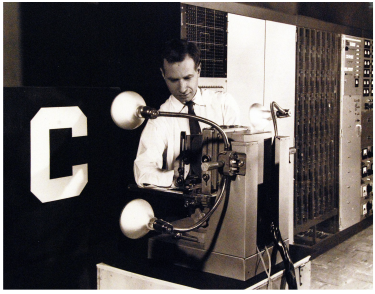
Rosenblatt's Perceptron (1957)

From neuron model to learning algorithm

- Introduced the **perceptron learning rule**:

$$w \leftarrow w + \eta(y - \hat{y})\, x.$$

- Built the first hardware neural net: **Mark I Perceptron** (the right figure).
- Established early theoretical results:
  - linear separability for classification;
  - convergence under separability (Perceptron Convergence Theorem).



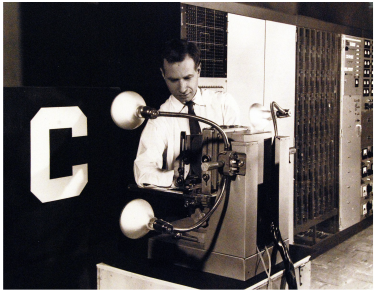
Rosenblatt's Perceptron (1957)

From neuron model to learning algorithm

- Introduced the **perceptron learning rule**:

$$w \leftarrow w + \eta(y - \hat{y})\, x.$$

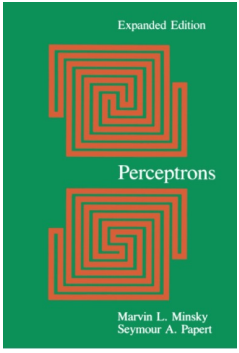
- Built the first hardware neural net: **Mark I Perceptron** (the right figure).
- Established early theoretical results:
  - linear separability for classification;
  - convergence under separability (Perceptron Convergence Theorem).

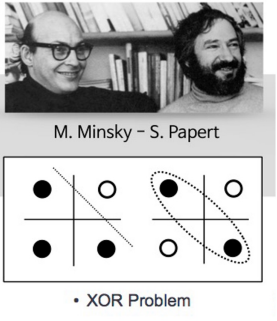


Impact:

- Turned neuron models into a **trainable machine learning algorithm**.
- Marked the beginning of learning theory.

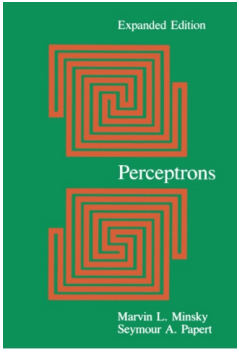
Minsky's Book: The Perceptrons (1963-1969)

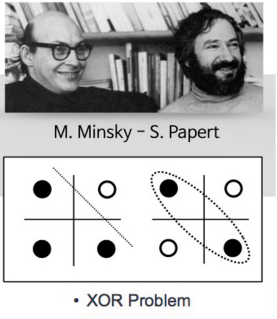




- Perceptron cannot represent the XOR function.

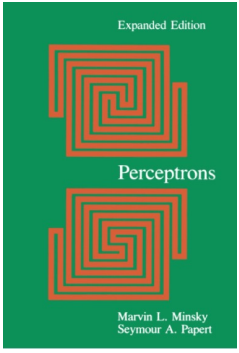
Minsky's Book: The Perceptrons (1963-1969)

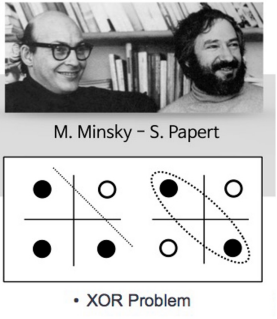




- Perceptron cannot represent the XOR function.
- Contribute to the first "AI winter".

Minsky's Book: The Perceptrons (1963-1969)





- Perceptron cannot represent the XOR function.
- Contribute to the first "AI winter".
- Motivate the development of **multi-layer perceptrons (MLPs)**, which became the basis of modern deep learning.

Two-layer Neural Networks

In 1965, Alexey Ivakhnenko and Valentin Lapa developed the Multilayer Perceptron (MLP).

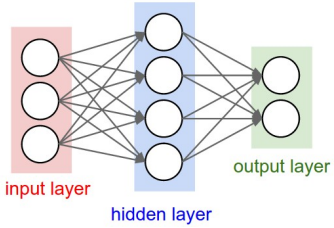
Two-layer Neural Networks

In 1965, Alexey Ivakhnenko and Valentin Lapa developed the Multilayer Perceptron (MLP).

- A two-layer network defines a map from $\mathbb{R}^d$ to $\mathbb{R}^k$

$$f_m(\boldsymbol{x};\theta) = \sum_{j=1}^{m} \boldsymbol{a}_j \sigma(\boldsymbol{b}_j \cdot \boldsymbol{x} + c_j)$$
$$= A\sigma(B\boldsymbol{x} + \boldsymbol{c}),$$

where $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times d}, \boldsymbol{c} \in \mathbb{R}^m$. Here, $\theta = \{A, B, \boldsymbol{c}\}$ are the trainable parameters.

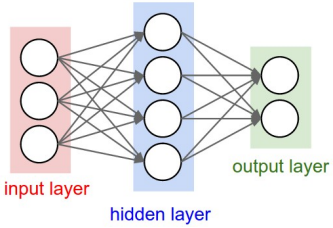


Two-layer Neural Networks

In 1965, Alexey Ivakhnenko and Valentin Lapa developed the Multilayer Perceptron (MLP).

- A two-layer network defines a map from $\mathbb{R}^d$ to $\mathbb{R}^k$

$$f_m(\boldsymbol{x};\theta) = \sum_{j=1}^{m} \boldsymbol{a}_j \sigma(\boldsymbol{b}_j \cdot \boldsymbol{x} + c_j)$$
$$= A\sigma(B\boldsymbol{x} + \boldsymbol{c}),$$



where $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times d}, \boldsymbol{c} \in \mathbb{R}^m$. Here, $\theta = \{A, B, \boldsymbol{c}\}$ are the trainable parameters.

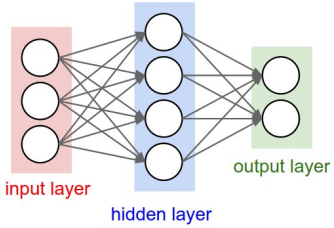
- $\sigma : \mathbb{R} \mapsto \mathbb{R}$ is the (nonlinear) activation function, e.g. $\sigma(z) = e^z/(1+e^z)$ (sigmoid). When z is a vector or matrix, $\sigma(z)$ should be understood in an **element-wise** manner.

Two-layer Neural Networks

In 1965, Alexey Ivakhnenko and Valentin Lapa developed the Multilayer Perceptron (MLP).

- A two-layer network defines a map from $\mathbb{R}^d$ to $\mathbb{R}^k$

$$f_m(\boldsymbol{x};\theta) = \sum_{j=1}^{m} \boldsymbol{a}_j \sigma(\boldsymbol{b}_j \cdot \boldsymbol{x} + c_j)$$
$$= A\sigma(B\boldsymbol{x} + \boldsymbol{c}),$$



where $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times d}, \boldsymbol{c} \in \mathbb{R}^m$. Here, $\theta = \{A, B, \boldsymbol{c}\}$ are the trainable parameters.

- $\sigma : \mathbb{R} \mapsto \mathbb{R}$ is the (nonlinear) activation function, e.g. $\sigma(z) = e^z/(1+e^z)$ (sigmoid). When z is a vector or matrix, $\sigma(z)$ should be understood in an **element-wise** manner.
- m denotes the number of neurons, which is also called the network **width**.

An Adaptive Feature Perspective

- Let $\varphi(\boldsymbol{x}; \boldsymbol{b}, c) = \sigma(\boldsymbol{b} \cdot \boldsymbol{x} + c)$. Two-layer neural networks can be written as

$$f_m(\boldsymbol{x}; \theta) = \sum_{j=1}^{m} \boldsymbol{a}_j \sigma(\boldsymbol{b}_j \cdot \boldsymbol{x} + c_j) = \sum_{j=1}^{m} \boldsymbol{a}_j \varphi(\boldsymbol{x}; \boldsymbol{b}_j, c_j)$$

  If $\{(\boldsymbol{b}_j, c_j)\}_{j=1}^m$ keep fixed after the (random) initialization and only train the outer coefficients $\{\boldsymbol{a}_j\}_{j=1}^m$, we obtain a random feature model.

An Adaptive Feature Perspective

- Let $\varphi(\boldsymbol{x}; \boldsymbol{b}, c) = \sigma(\boldsymbol{b} \cdot \boldsymbol{x} + c)$. Two-layer neural networks can be written as

$$f_m(\boldsymbol{x}; \theta) = \sum_{j=1}^{m} \boldsymbol{a}_j \sigma(\boldsymbol{b}_j \cdot \boldsymbol{x} + c_j) = \sum_{j=1}^{m} \boldsymbol{a}_j \varphi(\boldsymbol{x}; \boldsymbol{b}_j, c_j)$$

  If $\{(\boldsymbol{b}_j, c_j)\}_{j=1}^m$ keep fixed after the (random) initialization and only train the outer coefficients $\{\boldsymbol{a}_j\}_{j=1}^m$, we obtain a random feature model.
- However, for neural networks, $\{(\boldsymbol{b}_j, c_j)\}_{j=1}^m$ are learned from data. Thus, two-layer neural networks can be interpreted as a specific type of adaptive feature methods.

Multilayer Fully-Connected Networks

- A L-layer network is defined as $f(x;\theta) = \boldsymbol{x}^L$, with $\boldsymbol{x}^0 = \boldsymbol{x}$ and

$$\boldsymbol{x}^{\ell+1} = \sigma(W^\ell \boldsymbol{x}^\ell + b^\ell), \quad \ell = 0, 1, \ldots, L-1. \tag{1}$$

Multilayer Fully-Connected Networks

- A L-layer network is defined as $f(x;\theta) = \boldsymbol{x}^L$, with $\boldsymbol{x}^0 = \boldsymbol{x}$ and

$$\boldsymbol{x}^{\ell+1} = \sigma(W^\ell \boldsymbol{x}^\ell + b^\ell), \quad \ell = 0, 1, \ldots, L-1. \tag{1}$$

- It is also common to write $f(\cdot;\theta)$ in a compositional form:

$$f(x;\theta) = \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(\boldsymbol{x}),$$

with $\mathcal{A}^{(\ell)}(\boldsymbol{z}) = W^\ell \boldsymbol{z} + \boldsymbol{b}^\ell$.

Multilayer Fully-Connected Networks

- A L-layer network is defined as $f(x;\theta) = \boldsymbol{x}^L$, with $\boldsymbol{x}^0 = \boldsymbol{x}$ and

$$\boldsymbol{x}^{\ell+1} = \sigma(W^\ell \boldsymbol{x}^\ell + b^\ell), \quad \ell = 0, 1, \ldots, L-1. \tag{1}$$

- It is also common to write $f(\cdot;\theta)$ in a compositional form:

$$f(x;\theta) = \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(\boldsymbol{x}),$$

  with $\mathcal{A}^{(\ell)}(\boldsymbol{z}) = W^\ell \boldsymbol{z} + \boldsymbol{b}^\ell$.
- $\theta = \{W^\ell, \boldsymbol{b}^\ell\}_\ell$ are the trainable parameters. $W^\ell \in \mathbb{R}^{m_{\ell+1} \times m_\ell}$ and $\boldsymbol{b}^\ell \in \mathbb{R}^{m_{\ell+1}}$ are called the **weight** and **bias** of ℓ-layer, respectively.

Multilayer Fully-Connected Networks

- A L-layer network is defined as $f(x;\theta) = \boldsymbol{x}^L$, with $\boldsymbol{x}^0 = \boldsymbol{x}$ and

$$\boldsymbol{x}^{\ell+1} = \sigma(W^\ell \boldsymbol{x}^\ell + b^\ell), \quad \ell = 0, 1, \ldots, L-1. \tag{1}$$

- It is also common to write $f(\cdot;\theta)$ in a compositional form:

$$f(x;\theta) = \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(\boldsymbol{x}),$$

  with $\mathcal{A}^{(\ell)}(\boldsymbol{z}) = W^\ell \boldsymbol{z} + \boldsymbol{b}^\ell$.
- $\theta = \{W^\ell, \boldsymbol{b}^\ell\}_\ell$ are the trainable parameters. $W^\ell \in \mathbb{R}^{m_{\ell+1} \times m_\ell}$ and $\boldsymbol{b}^\ell \in \mathbb{R}^{m_{\ell+1}}$ are called the **weight** and **bias** of ℓ-layer, respectively.
- Layers $1, 2, \ldots, L$ are the hidden layers, and 0 and L are called the input and output layer, respectively. L and $\max\{m_1, \ldots, m_{L-1}\}$ are the depth and width, respectively.

Multilayer Fully-Connected Networks (Cont'd)

- We call $f(\cdot;\theta)$ a **fully-connected** neural networks since $\{W^\ell\}$ are dense matrices.
- They are also called multilayer perceptron (**MLP**) networks due to historical reasons.

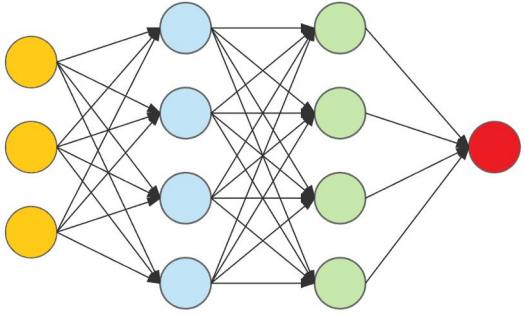


Figure 1: Play with MLP: https://playground.tensorflow.org.

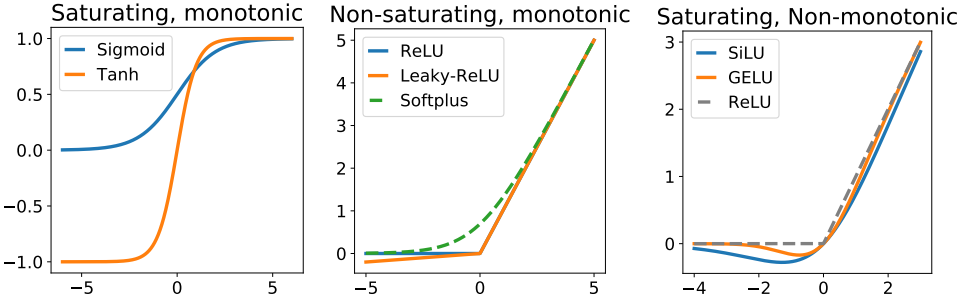
Activation Functions

Saturating	Sigmoid	$\frac{1}{1+e^{-x}}$
	Tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$
Non-saturating	ReLU	$\max(0, x)$
	Leaky ReLU	$\max(ax, x)$, where a is a small value, e.g. 0.01
	Parametric ReLU	$\max(ax, x)$, with a **learnable**
	Softplus	$\ln(1 + e^x)$
	GELU	$x\Phi(x)$
	SiLU	$x\sigma_{\text{sigmoid}}(\beta x)$

Table 1: Commonly used activation functions. ReLU stands for rectified linear unit. $\Phi(\cdot)$ is the CDF of $\mathcal{N}(0, 1)$. GELU and SiLU (aka Swish) belongs to the **self-gated family**: $x\phi(x)$ with ϕ being a CDF.

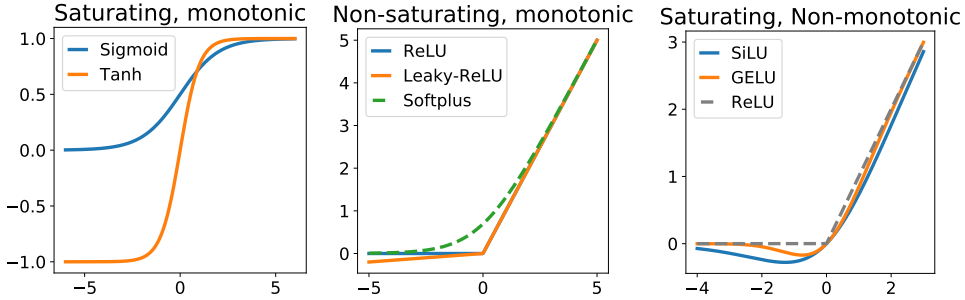
- The Gaussian error linear unit (GELU) and sigmoid linear unit (SiLU) becomes popular recently.
- **Question:** Why is ReLU not good choice for solving scientific computing problems?

Comparison of Activation Functions



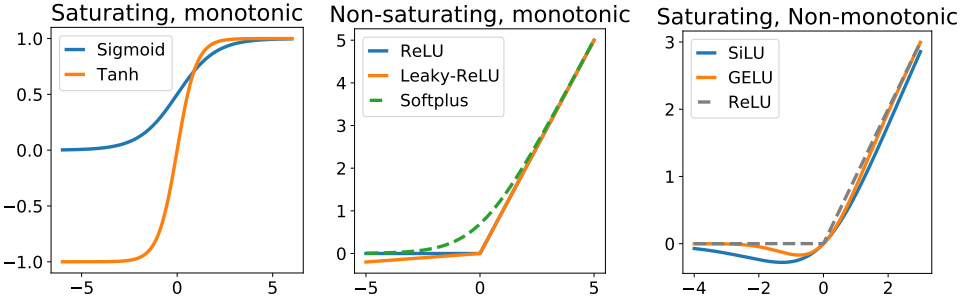
- Softplus, GELU, and SiLU can be viewed as smoothed versions of ReLU. Currently, ReLU and ReLU variants are the most popular ones.

Comparison of Activation Functions



- Softplus, GELU, and SiLU can be viewed as smoothed versions of ReLU. Currently, ReLU and ReLU variants are the most popular ones.
- The non-monotonic GELU and SiLU become very popular very recently.

Comparison of Activation Functions



- Softplus, GELU, and SiLU can be viewed as smoothed versions of ReLU. Currently, ReLU and ReLU variants are the most popular ones.
- The non-monotonic GELU and SiLU become very popular very recently.
- For saturating activation functions, $\sigma'(z) \approx 0$ when $|z|$ is relatively large. This is bad for training.

Universal Approximation Theorem (UAT)

Theorem 1 (Cybenko 1989)

Let Ω be a compact subset in $\mathbb{R}^d$. Assume that σ is sigmoidal, i.e.

$$\sigma(t) \to \begin{cases} 1 & t \to +\infty \\ 0 & t \to -\infty. \end{cases}$$

For any $f \in C(\Omega)$ and $\varepsilon > 0$, there exist a two-layer neural network $f_m(\boldsymbol{x}; \theta) = \sum_{j=1}^{m} a_j \sigma(\boldsymbol{b}_j^\top \boldsymbol{x} + c_j)$ such that

$$\sup_{\boldsymbol{x} \in \Omega} |f(\boldsymbol{x}) - f_m(\boldsymbol{x})| \leq \varepsilon.$$

- The above theorem says that two-layer neural networks can approximate any continuous function.
- UAT is an analog of Weierstrass Theorem which asserts that on compact domains, **continuous functions can be approximated by polynomials**.

Universal Approximation Theorem (UAT)

- By itself, it does not explain the success of neural network approximations over polynomial approximations (in high dimensions).
- UAT demonstrated the **universality** of neural network's approximation—**making neural networks a subject of interest across disciplines.**

George Cybenko

FOLLOW

Thayer School of Engineering at Dartmouth
Verified email at dartmouth.edu

Disclosure risk cyber security behavior analysis adversarial dynamics covert channels

TITLE	CITED BY	YEAR
Approximation by superpositions of a sigmoidal function G Cybenko Mathematics of control, signals and systems 2 (4), 303-314	22850	1989

Convolutional Neural Networks

Question:

- Why are MLPs not well-suited for processing image or video inputs?

Convolutional neural networks

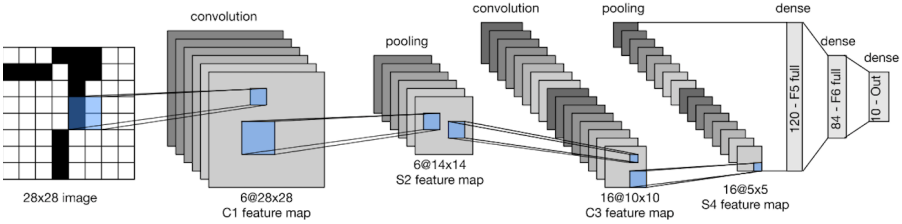


Figure 2: LeNet-5 for MNIST dataset

- Convolutional networks are similar to fully connected networks,

$$f(x) = \mathcal{A}^{(L)} \circ \sigma \circ \mathcal{A}^{(L-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)} x.$$

The only difference is that $\mathcal{A}^{(\ell)} \boldsymbol{z} = \boldsymbol{z} * \boldsymbol{w}^{\ell} + \boldsymbol{b}^{\ell}$ is a convolutional transformation.

History of CNNs

- In the **1950s-1960s, Hubel and Wiesel** demonstrated that cat **visual cortices** contain neurons responsive to specific small regions of the visual field (**receptive filed**).

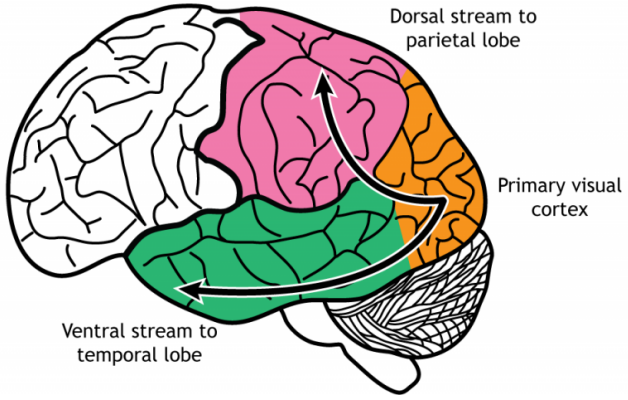
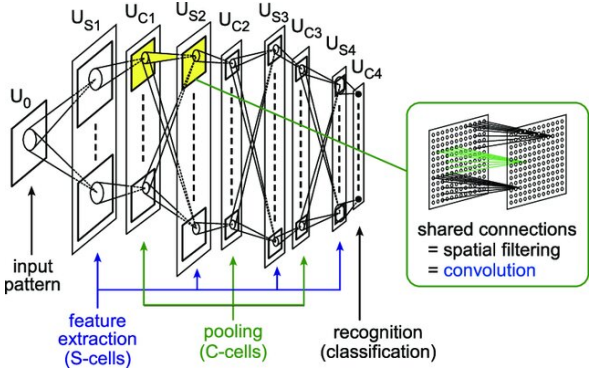


Figure 3: If you are interested in learning how the human brain processes visual signals, we recommend visiting this link.

History of CNNs

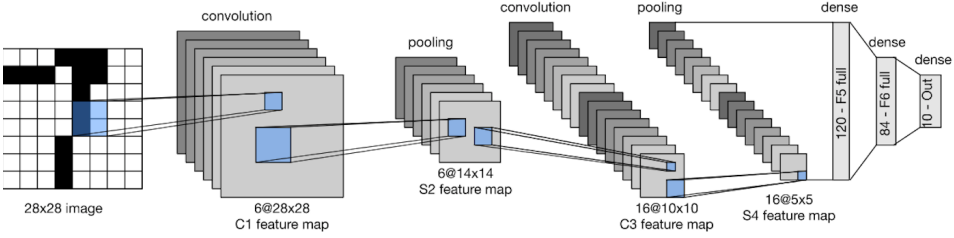
- **In 1969, Kunihiko Fukushima** introduced the first deep ReLU CNN, called **Neocognitron**, featuring fixed filters:
  - The "S-layer": a weight-shared receptive field layer, later termed conv. layers.
  - The "C-layer": a downsampling layer.

  But the **filters are not learnable**.



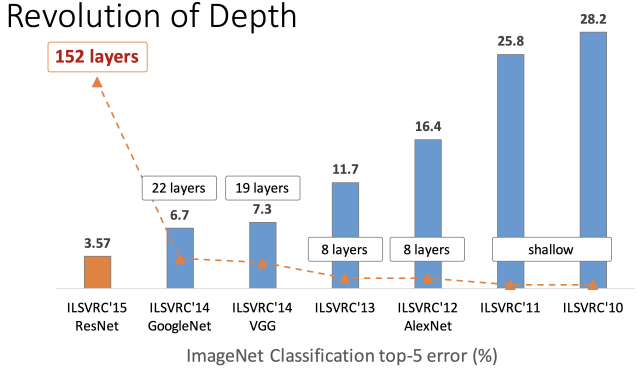
History of CNNs

- In **1989, Yann LeCun** et al. **utilized backpropagation to learn convolutional filters** for handwritten digit classification.
- In **1995, Yann LeCun** introduced LeNet-5, a 7-layer CNN designed for classifying **high-resolution** "32x32" handwritten digit images, which was adopted by NCR for its check reading system.



History of CNNs

- In **2012, AlexNet**, developed by Alex Krizhevsky and Geoffrey Hinton, won the ImageNet challenge with images of size 224x224x3. **This ignited the era of deep learning**.
- In **2015, ResNet**, developed by Kaiming He et al., enabled the training of very deep (hundreds layers) CNNs.



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

1D Convolution Transform

- Consider the 1-D signal $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$.

1D Convolution Transform

- Consider the 1-D signal $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$.
- Given a filter $\boldsymbol{w} \in \mathbb{R}^k$, a "valid" convolutional transform, $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, defines a linear map: $\mathbb{R}^n \mapsto \mathbb{R}^{n-k+1}$ as follows

$$y_s = \sum_{i=1}^{k} x_{s+i-1} w_i, \quad \forall s = 1, \cdots, n-k+1.$$

1D Convolution Transform

- Consider the 1-D signal $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$.
- Given a filter $\boldsymbol{w} \in \mathbb{R}^k$, a "valid" convolutional transform, $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, defines a linear map: $\mathbb{R}^n \mapsto \mathbb{R}^{n-k+1}$ as follows

$$y_s = \sum_{i=1}^{k} x_{s+i-1} w_i, \quad \forall s = 1, \cdots, n-k+1.$$

- **Matrix Form:** The convolutional transform can be written in a matrix form. For example, if $\boldsymbol{w} = (w_1, w_2, w_3)^\top \in \mathbb{R}^3$, we have

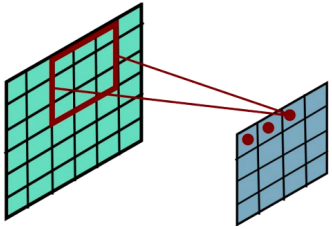
$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-3+1} \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & w_3 & \cdots & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & \cdots & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & w_1 & w_2 & w_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}.$$

The matrix corresponds to general $\boldsymbol{w} \in \mathbb{R}^k$ is given similarly.

2D convolutional transform

We can similarly define the "valid" convolutional transform for $x \in \mathbb{R}^{d \times d}$. Then, the filter $w \in \mathbb{R}^{k \times k}$ is a small matrix. Let $y = x * w \in \mathbb{R}^{(n-k+1) \times (n-k+1)}$, then

$$y_{s,t} = \sum_{i,j=1}^{k} x_{s+i,t+j} w_{i,j}.$$



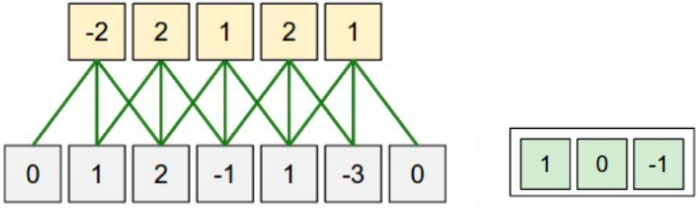
- **Sliding window!**
- **Small filter size!**

Padding

- **Padding:** To simplify the design of architecture of networks, we usually hope *the output has the same dimension as the input*. We can first appropriately pad zeros in the boundary, then perform "valid" convolutional transform

Padding

- **Padding:** To simplify the design of architecture of networks, we usually hope *the output has the same dimension as the input*. We can first appropriately pad zeros in the boundary, then perform "valid" convolutional transform
- **Visualization**: $x = (1, 2, -1, 1, -3) \in \mathbb{R}^5, w = (1, 0, -1)^T \in \mathbb{R}^3$. Then $y = x * w = (-2, 2, 1, 2, 1) \in \mathbb{R}^5$.



Motivation to Use Convolution Transforms

- Convolutional transforms are widely used for data with spatial structures, such as audio (1-D), image(2-D), video(3-D).

Motivation to Use Convolution Transforms

- Convolutional transforms are widely used for data with spatial structures, such as audio (1-D), image(2-D), video(3-D).
- We usually choose a small filter size k, e.g. $3, 5.$, to better capture the local correlation (see, e.g., the following example). The global structures are captured by stacking many layers of convolutional transforms.

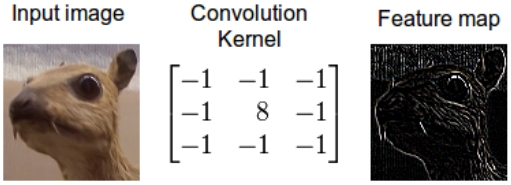


Figure 3: Taken from https://developer.nvidia.com/discover/convolution.

Motivation to Use Convolution Transforms

- Convolutional transforms are widely used for data with spatial structures, such as audio (1-D), image(2-D), video(3-D).
- We usually choose a small filter size k, e.g. $3, 5.$, to better capture the local correlation (see, e.g., the following example). The global structures are captured by stacking many layers of convolutional transforms.

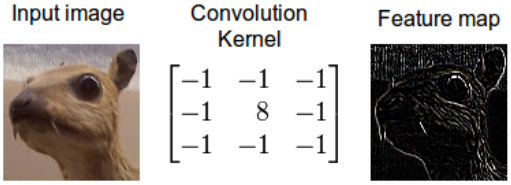


Figure 3: Taken from https://developer.nvidia.com/discover/convolution.

- Fully-connected layers ignore the spatial locality and translation structure of images.

Motivation to use convolutional transforms (Cont'd)

- Translation invariance.
- The number of parameters to be learned for convolutional transforms are much smaller than that of fully-connected linear transforms. It is also much efficient to compute former than the latter.

Channels

Assume the input is an image.

- Let h^ℓ denote output of the ℓ-th layer. $h^\ell \in \mathbb{R}^{W_\ell \times H_\ell \times C_l}$ is a 3-order tensor. h^ℓ is called a **feature map** with shape (width W_ℓ) $\times$ (height H_ℓ) $\times$ (channels C_ℓ).

Channels

Assume the input is an image.

- Let h^ℓ denote output of the ℓ-th layer. $h^\ell \in \mathbb{R}^{W_\ell \times H_\ell \times C_l}$ is a 3-order tensor. h^ℓ is called a **feature map** with shape (width W_ℓ) $\times$ (height H_ℓ) $\times$ (channels C_ℓ).
- Consider the input h^0. $C_0 = 1$ for a grayscale image; $C_0 = 3$ for a color image. The different channels store different information.

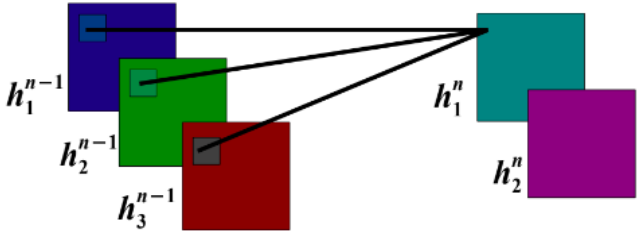
Channels

Assume the input is an image.

- Let h^ℓ denote output of the ℓ-th layer. $h^\ell \in \mathbb{R}^{W_\ell \times H_\ell \times C_l}$ is a 3-order tensor. h^ℓ is called a **feature map** with shape (width W_ℓ) $\times$ (height H_ℓ) $\times$ (channels C_ℓ).
- Consider the input h^0. $C_0 = 1$ for a grayscale image; $C_0 = 3$ for a color image. The different channels store different information.
- It is expected that as we go deeper, the information stored at different channels becomes eventually "disentangled". For example, when extracting features from an image of human, we would like that channel 1 represents "eye"; channel 2 represents "leg"; channel 3 represents "hand", etc.

A Convolutional Layer

A convolutional layer performs the convolution transform along the width and height dimensions and the **fully-connected** transform along the channel dimension.



A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.
- h_t^o is the t-th channel of output feature map.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.
- h_t^o is the t-th channel of output feature map.
- h_s^i is the s-th channel of input feature map.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.
- h_t^o is the t-th channel of output feature map.
- h_s^i is the s-th channel of input feature map.
- "$*$" denotes the convolution transform with an appropriate padding.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.
- h_t^o is the t-th channel of output feature map.
- h_s^i is the s-th channel of input feature map.
- "$*$" denotes the convolution transform with an appropriate padding.
- b^t is the bias corresponding to t-th channel of output feature map.

A Convolutional Layer (Cont'd)

- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically, a convolutional layer makes the following transform:

$$h_t^o = \sum_{s=1}^{C_i} h_s^i * w^{s,t} + b^t,$$

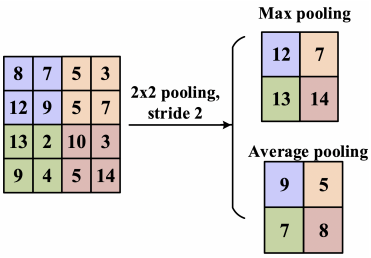
where

- $w^{s,t} \in \mathbb{R}^{k \times k}$ denotes the filter from the s-th channel of input to the t-th channel of output.
- h_t^o is the t-th channel of output feature map.
- h_s^i is the s-th channel of input feature map.
- "$*$" denotes the convolution transform with an appropriate padding.
- b^t is the bias corresponding to t-th channel of output feature map.

- Note that (w, b) will be learned from the data.

Pooling Layer

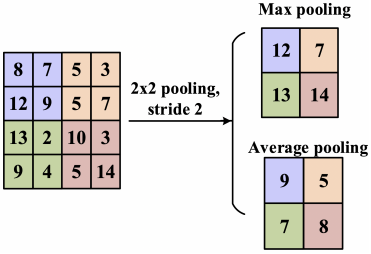
- **Pooling (aka down-sampling):** There are two types of pooling: max pooling and average pooling.



- **Pooling layer:** $\mathbb{R}^{W \times H \times C} \mapsto \mathbb{R}^{\frac{W}{k} \times \frac{H}{k} \times C}$.
  - Pooling is performed for each channel, with no across-channel mixing.
  - No learnable parameters.

Pooling Layer

- **Pooling (aka down-sampling):** There are two types of pooling: max pooling and average pooling.



- **Pooling layer:** $\mathbb{R}^{W \times H \times C} \mapsto \mathbb{R}^{\frac{W}{k} \times \frac{H}{k} \times C}$.
  - Pooling is performed for each channel, with no across-channel mixing.
  - No learnable parameters.
- **Motivation:**
  - Decreasing the spatial dimension can reduce the memory usage. Hence, we can increase the number of channels without running out of the GPU memory.
  - For image classification problems, coarse graining does not lose too much category information.

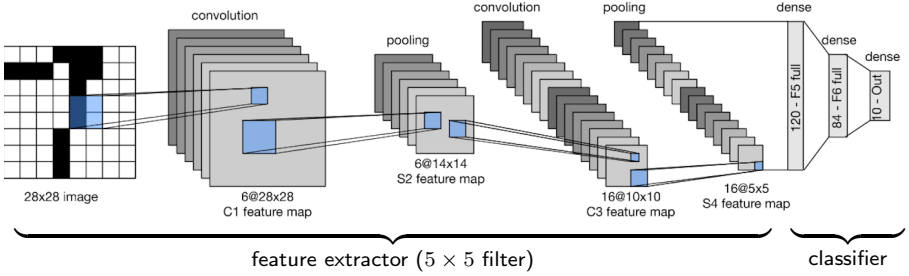
A Closer Look at LeNet-5

- **MNIST:** Handwritten Digits, $60,000$ training examples, $10,000$ test examples. Each sample is a 28×28 grayscale image.

- **Task:** build a classifier: $f(x) : \mathbb{R}^{28\times28\times1} \mapsto \mathbb{R}^{10}$, with $f_i(x) \in [0, 1]$ and $\sum_{i=1}^{10} f_i(x) = 1$.

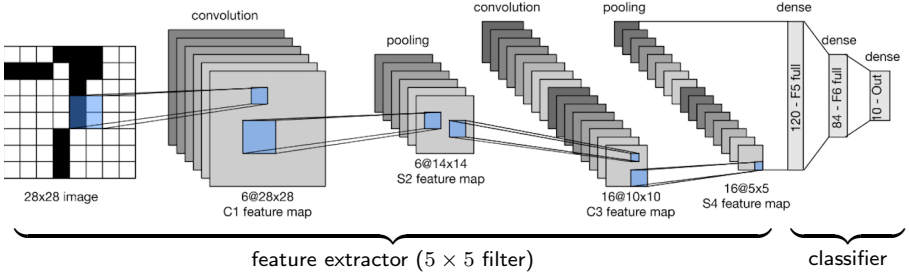
A Closer Look at LeNet-5

- **LeNet-5:** Convolutional layers + Fully-connected layers + Softmax.



A Closer Look at LeNet-5

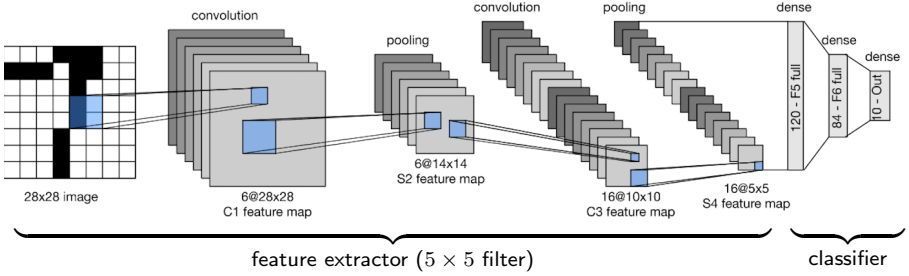
- **LeNet-5:** Convolutional layers + Fully-connected layers + Softmax.



- The outputs before the softmax layer are usually called logits. Then, **softmax layer** converts logits to a probability: $\mathbb{R}^k \mapsto \mathbb{R}^k$ $p_i(x) = \frac{e^{x_i}}{\sum_{i=1}^{k} e^{x_i}}$, which gives the predicted probability over the classes.

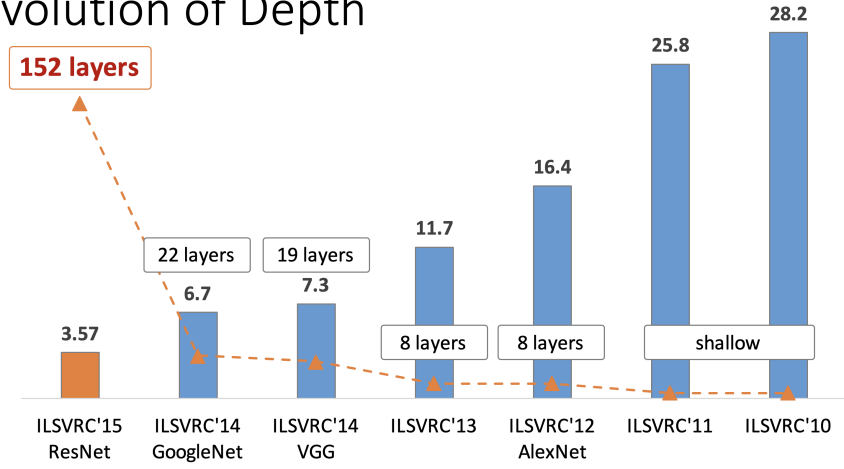
A Closer Look at LeNet-5

- **LeNet-5:** Convolutional layers + Fully-connected layers + Softmax.



- The outputs before the softmax layer are usually called logits. Then, **softmax layer** converts logits to a probability: $\mathbb{R}^k \mapsto \mathbb{R}^k$ $p_i(x) = \frac{e^{x_i}}{\sum_{i=1}^{k} e^{x_i}}$, which gives the predicted probability over the classes.
- **One useful principle:** While decreasing the spatial dimension, increase the number of channels.

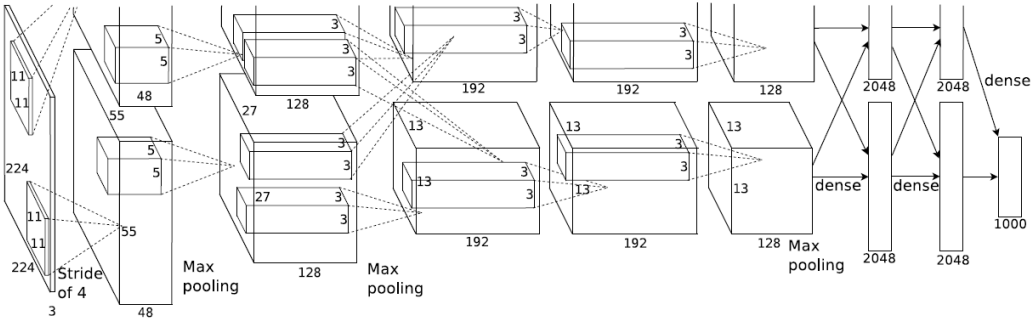
Revolution of Depth



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Figure 4: Taken from Kaiming He's slide.

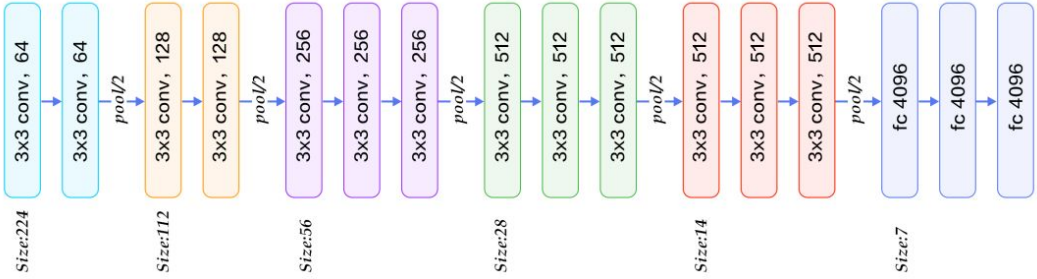
AlexNet: 2012



Contribution:

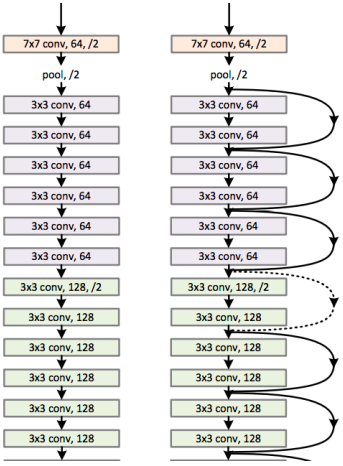
- BIG LeNet!
- deep CNN, GPU Acceleration. (Jürgen Schmidhuber team did the same thing in 2011, but unfortunately their CNNs are trained for a small-scale dataset.)
- ReLU and ImageNet.

VGG: 2014



- Small (3×3) convolutional layer.
- Better architecture-design principles.

Residual Networks (ResNets): 2015



Vanilla net

$$x^{\ell+1} = h(x^\ell; \theta^\ell)$$

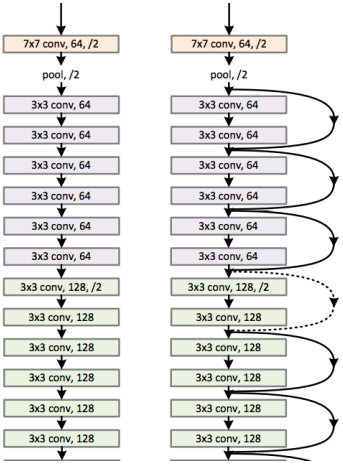
Residual net

$$x^{\ell+1} = h(x^\ell; \theta^\ell) + x^\ell$$

$h(\cdot; \theta^l)$ can be a fully-connected or convolutional neural network.

- In ResNets, we learn the residual $h(\cdot; \theta^\ell)$ instead of the full map $\mathrm{Id} + h(\cdot; \theta^\ell)$.

Residual Networks (ResNets): 2015



Vanilla net

$$x^{\ell+1} = h(x^\ell; \theta^\ell)$$

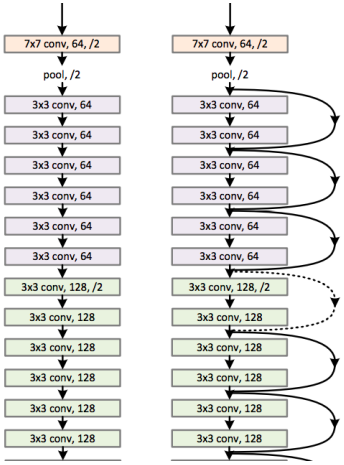
Residual net

$$x^{\ell+1} = h(x^\ell; \theta^\ell) + x^\ell$$

$h(\cdot; \theta^l)$ can be a fully-connected or convolutional neural network.

- In ResNets, we learn the residual $h(\cdot; \theta^\ell)$ instead of the full map $\mathrm{Id} + h(\cdot; \theta^\ell)$.
- Residual and vanilla nets have the same expressivity: $\boldsymbol{x} = \mathtt{ReLU}(\boldsymbol{x}) - \mathtt{ReLU}(-\boldsymbol{x})$.

Residual Networks (ResNets): 2015



Vanilla net

$$x^{\ell+1} = h(x^\ell; \theta^\ell)$$

Residual net

$$x^{\ell+1} = h(x^\ell; \theta^\ell) + x^\ell$$

$h(\cdot; \theta^l)$ can be a fully-connected or convolutional neural network.

- In ResNets, we learn the residual $h(\cdot; \theta^\ell)$ instead of the full map $\mathrm{Id} + h(\cdot; \theta^\ell)$.
- Residual and vanilla nets have the same expressivity: $\boldsymbol{x} = \mathtt{ReLU}(\boldsymbol{x}) - \mathtt{ReLU}(-\boldsymbol{x})$.
- Skip connections can be more general, e.g. connecting the input to the output directly.

Recurrent Neural Networks

Motivation

Many tasks involve sequential data:

- Speech-to-text and text-to-speech.
- Machine translation.
- Sentiment analysis.
- video / image captioning.

Core Challenge: Long-range Dependence.
Important information at time t may depend on inputs hundreds or thousands of steps earlier:

$$y_t = H(x_1, x_2, \ldots, x_t).$$

Capturing these long-distance relations is fundamentally difficult for models with finite memory.

Abstract Formulation of Sequence Modeling

- Input: $\boldsymbol{x} = (x_1, \ldots, x_T)$, $x_t \in \mathbb{R}^{d_x}$.
- Output: $\boldsymbol{y} = (y_1, \ldots, y_T)$, $y_t \in \mathbb{R}^{d_y}$.

Long-range dependence:

$$y_t = H_t(x_1, \ldots, x_t)$$

where the relevant information for y_t may lie far back in the input sequence.

The challenge is to extract and preserve useful information from the distant past.

State-Space Models (SSMs): A Unifying Abstraction

- Directly modeling $H_t(x_1, \ldots, x_t)$ is infeasible for long sequences.
- Instead, we assume that all relevant information about the past can be summarized in a **latent state**

$$S_t \in \mathbb{R}^m.$$

State update (general, possibly nonlinear):

$$S_t = G_\theta(x_t, S_{t-1}).$$

Output / readout:

$$y_t = O_\theta(S_t).$$

Here we use "state-space model" purely as an *abstract computational framework*, rather than the classical linear dynamical systems from control theory: a hidden state is updated over time by a shared rule G_θ and mapped to outputs by O_θ.

State-Space View of RNNs

In this abstract framework:

- **State:** S_t is the hidden representation at time t.
- **Update rule:** G_θ describes how we combine the new input x_t with the previous state S_{t-1}.
- **Readout:** O_θ maps the state to the output y_t.

Key perspective:

- RNNs, LSTMs and GRUs are all concrete parameterizations of (G_θ, O_θ) in this state-space view.
- Later, encoder–decoder architectures can be seen as *compositions* of such state-space modules.

Vanilla RNN as a SSM

A **vanilla RNN** chooses a specific form of the state update and readout.

Update formulation:

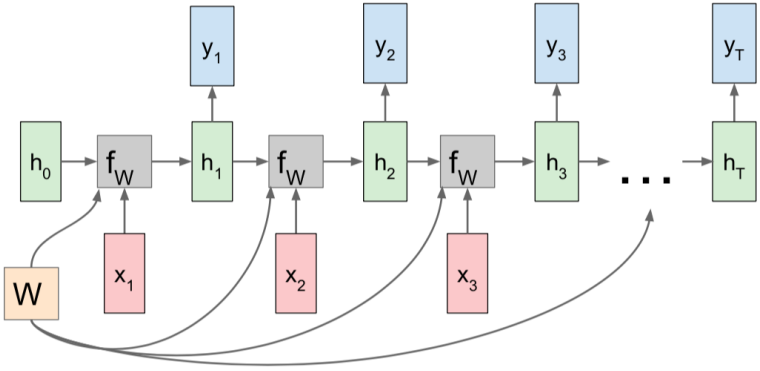
$$h_t = \tanh(W_{hh} h_{t-1} + W_{hx} x_t + b_h),$$
$$y_t = W_{yh} h_t + b_y.$$

In the state-space notation:

$$S_t = h_t, \quad G_\theta(x_t, S_{t-1}) = \tanh(W_{hh} S_{t-1} + W_{hx} x_t + b_h), \quad O_\theta(S_t) = W_{yh} S_t + b_y.$$

Parameters are shared across time, enabling learning over arbitrarily long sequences (in principle).

Vanilla RNN: Computation Graph



- The hidden state h_t is updated recurrently over time.
- Information flows along the time axis through the hidden states.

Why RNNs Struggle with Long-range Dependencies?

The state update

$$h_t = \tanh(W_{hh} h_{t-1} + U x_t)$$

repeatedly mixes new information with old.

1. Information is repeatedly overwritten (finite state).

- h_t is a fixed-dimensional summary of $(x_1, \ldots, x_t)$.
- As t grows, older information becomes increasingly difficult to retain.

2. Gradient propagation = information propagation:

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{j=1}^{k} \frac{\partial h_{t-j+1}}{\partial h_{t-j}} \approx (W'_{hh})^k.$$

If $\rho(W'_{hh}) < 1 \rightarrow$ gradients decay exponentially; If $\rho(W'_{hh}) > 1 \rightarrow$ gradients explode.

Result: RNNs struggle to learn long-range dependencies.

Long Short-Term Memory (LSTM)

LSTM introduces a more flexible state update to *control* how much past information is carried forward.

- **Gate update:**

$$\begin{pmatrix} f_t \\ i_t \\ o_t \end{pmatrix} = \texttt{sigmoid} \begin{pmatrix} W_f x_t + U_f h_{t-1} + b_f \\ W_i x_t + U_i h_{t-1} + b_i \\ W_o x_t + U_o h_{t-1} + b_o \end{pmatrix}$$

- **Memory update:**

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \texttt{tanh}\left(W_c x_t + U_c h_{t-1} + b_c\right) \\ h_t &= o_t \odot c_t \end{aligned}$$

where $o_t, f_t, i_t \in [0, 1]$ represent the output gate, forget gate and input gate, respectively. $\odot$ denotes the hadamard product.

Key idea: the cell state c_t can preserve information over long time horizons by keeping $f_t \approx 1$ and i_t small when needed.

LSTM in the State-Space View

$$S_t = (c_t, h_t), \quad S_t = G_\theta(x_t, S_{t-1}), \quad y_t = O_\theta(S_t).$$

- Still a state-space model.
- But the state update G_θ now includes gates that control:
  - how much of the old state is kept,
  - how much new information is written,
  - how much information is exposed.
- This enables LSTMs to maintain much longer-range dependencies than RNNs.

Fundamental Limits of Finite-state SSMs

Even with gating, any model of the form

$$S_t = G_\theta(x_t, S_{t-1})$$

has intrinsic difficulty with **very long-range dependence**.

Fundamental Limits of Finite-state SSMs

Even with gating, any model of the form

$$S_t = G_\theta(x_t, S_{t-1})$$

has intrinsic difficulty with **very long-range dependence**.

1. Finite state = inevitable information bottleneck. Older information must eventually be compressed or forgotten.

Fundamental Limits of Finite-state SSMs

Even with gating, any model of the form

$$S_t = G_\theta(x_t, S_{t-1})$$

has intrinsic difficulty with **very long-range dependence**.

1. Finite state = inevitable information bottleneck. Older information must eventually be compressed or forgotten.

2. Repeated updates distort old information.

$$\frac{\partial S_t}{\partial x_{t-k}} = \prod_{j=0}^{k-1} \frac{\partial G_\theta}{\partial S_{t-j-1}} \frac{\partial G_\theta}{\partial x_{t-k}}.$$

Exponential decay makes long-range signals very weak.

Therefore: general SSMs (RNN/LSTM/GRU) have built-in limitations for extremely long-range dependencies.

Encoder-Decoder Architectures

What if input and output have different lengths?

Encoder-Decoder Architectures

What if input and output have different lengths?

- **Encoder:** a state-space model that reads the input sequence and compresses it into a final state (or a sequence of states).
- **Decoder:** another state-space model that generates the output sequence conditioned on the encoder state.

Perspective: encoder–decoder architectures can be viewed as *compositions of state-space models*, and later Transformers will provide a different way to model sequence-to-sequence maps without relying on a single recurrent state.

Looking Ahead: Beyond Recurrent SSMs

Recurrent SSMs compress the entire history into a *single evolving state*:

$$S_1 \to S_2 \to \cdots \to S_T.$$

Transformers take a fundamentally different approach:

- no single recurrent state,
- each token can attend to any previous token,
- information flow is global rather than recurrent.

This avoids the memory bottleneck and gradient decay intrinsic to finite-dimensional recurrent models.

Geometric Deep Learning: Symmetry-Preserving Neural Networks

Consider an invariance group G, e.g., the permutation, translation, and rotation groups. For any $x \in \mathcal{X}$, suppose $\sigma \cdot x \in \Omega$ for any $\sigma \in G$.

- **Invariance:** $f : \mathcal{X}^d \to \mathbb{R}$ is said to be G-invariant if $f(\sigma \cdot x) = f(x)$ for any $\sigma \in G$.

Geometric Deep Learning: Symmetry-Preserving Neural Networks

Consider an invariance group G, e.g., the permutation, translation, and rotation groups. For any $x \in \mathcal{X}$, suppose $\sigma \cdot x \in \Omega$ for any $\sigma \in G$.

- **Invariance:** $f : \mathcal{X}^d \to \mathbb{R}$ is said to be G-invariant if $f(\sigma \cdot x) = f(x)$ for any $\sigma \in G$.
- **Equivariance:** $F : \mathcal{X}^d \to \mathcal{X}^d$ is said to be G-equivariant if $F(\sigma \cdot x) = \sigma \cdot F(x)$ for any $\sigma \in G$.

Geometric Deep Learning: Symmetry-Preserving Neural Networks

Consider an invariance group G, e.g., the permutation, translation, and rotation groups. For any $x \in \mathcal{X}$, suppose $\sigma \cdot x \in \Omega$ for any $\sigma \in G$.

- **Invariance:** $f : \mathcal{X}^d \to \mathbb{R}$ is said to be G-invariant if $f(\sigma \cdot x) = f(x)$ for any $\sigma \in G$.
- **Equivariance:** $F : \mathcal{X}^d \to \mathcal{X}^d$ is said to be G-equivariant if $F(\sigma \cdot x) = \sigma \cdot F(x)$ for any $\sigma \in G$.

Geometric Deep Learning: Symmetry-Preserving Neural Networks

Consider an invariance group G, e.g., the permutation, translation, and rotation groups. For any $x \in \mathcal{X}$, suppose $\sigma \cdot x \in \Omega$ for any $\sigma \in G$.

- **Invariance:** $f : \mathcal{X}^d \to \mathbb{R}$ is said to be G-invariant if $f(\sigma \cdot x) = f(x)$ for any $\sigma \in G$.
- **Equivariance:** $F : \mathcal{X}^d \to \mathcal{X}^d$ is said to be G-equivariant if $F(\sigma \cdot x) = \sigma \cdot F(x)$ for any $\sigma \in G$.

We will focus on constructing networks satisfying certain invariances.

Permutation Symmetry

- A function $f : \mathbb{R}^{n \times d} \to \mathbb{R}$ is said to be permutation invariant if

$$f(\boldsymbol{x}_{\sigma(1)}, \ldots, \boldsymbol{x}_{\sigma(n)}) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \tag{2}$$

  for any permutation $\sigma \in S_n$ and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$.
- We can also understand f as a function over the **set** $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$.

Permutation Symmetry

- A function $f : \mathbb{R}^{n \times d} \to \mathbb{R}$ is said to be permutation invariant if

$$f(\boldsymbol{x}_{\sigma(1)}, \ldots, \boldsymbol{x}_{\sigma(n)}) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \tag{2}$$

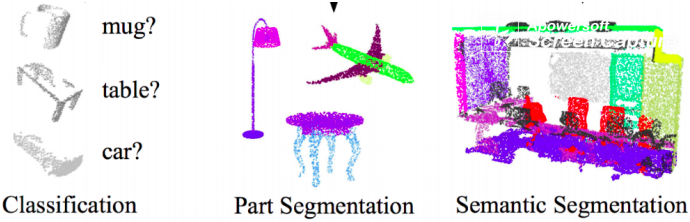
  for any permutation $\sigma \in S_n$ and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$.
- We can also understand f as a function over the **set** $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$.

Example:

- $f(x_1, \ldots, x_n) = \max\{x_1, \ldots, x_n\}$.
- $f(x_1, \ldots, x_n) = \sum_{i=1}^n x_i$.

Applications

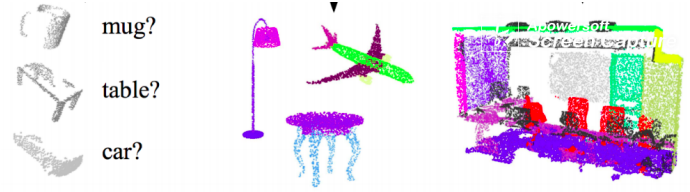
- Point cloud.



Classification | Part Segmentation | Semantic Segmentation

Applications

- Point cloud.

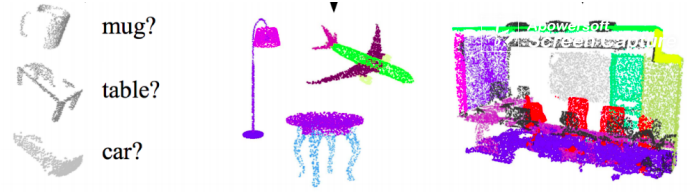


Classification | Part Segmentation | Semantic Segmentation

- Wave functions of bosons in quantum physics.

Applications

- Point cloud.



Classification | Part Segmentation | Semantic Segmentation

- Wave functions of bosons in quantum physics.
- Energy function of a molecule. The energy should keep unchanged if we swap two identical atoms.

Deep Sets

Given the one-particular feature extractor $g : \mathbb{R}^d \to \mathbb{R}^m$ and $\phi : \mathbb{R}^m \to \mathbb{R}^1$, the deep set model is given by

$$(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \mapsto \phi \left(\sum_{j=1}^{n} g(\boldsymbol{x}_j) \right)$$

In practice, we can replace g and ϕ with neural nets. The corresponding models are called **deep sets**).

Approximation of Permutation-Invariant Functions

- UAT guarantees that any continuous permutation-invariant function can be approximated by neural networks. But the networks are not permutation invariant.
- Can we construct models that has UAT while preserving the symmetry?

[2] Universal approximation of symmetric and anti-symmetric functions

Approximation of Permutation-Invariant Functions

- UAT guarantees that any continuous permutation-invariant function can be approximated by neural networks. But the networks are not permutation invariant.
- Can we construct models that has UAT while preserving the symmetry?

The following theorem shows deep sets are universal [2].

Theorem 2 (Han et al. 2019)

Let $f : \mathbb{R}^{n\times d} \to \mathbb{R}$ be a permutation invariant and continuous differentiable function. Let Ω be a compact subset of $\mathbb{R}^d$. For any $\varepsilon \in (0, \sqrt{nd}n^{-1/d})$, there exits $g : \mathbb{R}^d \to \mathbb{R}^m$, $\phi : \mathbb{R}^m \to \mathbb{R}$ such that

$$\sup_{\boldsymbol{x}\in\Omega} \left| f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) - \phi\Big(\sum_{j=1}^{n} g(\boldsymbol{x}_j)\Big) \right| \leq \varepsilon,$$

where m, the number of feature variables, satisfies that $m \geq O\left(\frac{2^n (nd)^{\frac{nd}{2}}}{\varepsilon^{nd} n!}\right)$

[2] Universal approximation of symmetric and anti-symmetric functions

Translation and Rotation Invariance

- Let $X = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^\top \in \mathbb{R}^{n \times d}$. A function $f : \mathbb{R}^{n \times d} \to \mathbb{R}$ is said to be translation invariant if

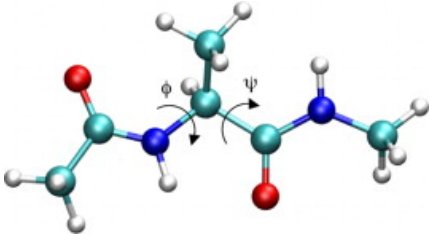
$$f(\boldsymbol{x}_1 + \boldsymbol{b}, \ldots, \boldsymbol{x}_n + \boldsymbol{b}) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \qquad \forall \boldsymbol{b} \in \mathbb{R}^d,$$

  and to be rotational invariant if

$$f(U\boldsymbol{x}_1, \ldots, U\boldsymbol{x}_n) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n),$$

  for any rotational matrix U.

Note that the the translation and rotation are applied to each "particle". The most important application is molecular modeling:



Network Designing

- Let r_c be a pre-specified cut-off radius. Define the neighbor of atom i by

$$\mathcal{N}_i = \{j \in [n] : \|\boldsymbol{x}_j - \boldsymbol{x}_i\| \leq r_c\},$$

and $n_i = |\mathcal{N}_i|$.

Network Designing

- Let r_c be a pre-specified cut-off radius. Define the neighbor of atom i by

$$\mathcal{N}_i = \{j \in [n] : \|\boldsymbol{x}_j - \boldsymbol{x}_i\| \leq r_c\},$$

  and $n_i = |\mathcal{N}_i|$.
- For each $\mathcal{N}_i$, define

$$R_i := (\boldsymbol{x}_{j_1} - \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{j_{n_i}} - \boldsymbol{x}_i)^T \in \mathbb{R}^{n_i \times d}$$

  for $j_k \in \mathcal{N}_i$. Then, the matrix

$$\Omega_i = R_i^T R_i$$

  is invariant with respect to both translation and rotation.

Network Designing

- Let r_c be a pre-specified cut-off radius. Define the neighbor of atom i by

$$\mathcal{N}_i = \{j \in [n] : \|\boldsymbol{x}_j - \boldsymbol{x}_i\| \leq r_c\},$$

  and $n_i = |\mathcal{N}_i|$.
- For each $\mathcal{N}_i$, define

$$R_i := (\boldsymbol{x}_{j_1} - \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{j_{n_i}} - \boldsymbol{x}_i)^T \in \mathbb{R}^{n_i \times d}$$

  for $j_k \in \mathcal{N}_i$. Then, the matrix

$$\Omega_i = R_i^T R_i$$

  is invariant with respect to both translation and rotation.
- Consider the function of the following form

$$f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \sum_{i=1}^{n} h_i(\Omega_i).$$

  It is obvious that f is invariant to translation and rotation.

Network Designing

- Let r_c be a pre-specified cut-off radius. Define the neighbor of atom i by

$$\mathcal{N}_i = \{j \in [n] : \|\boldsymbol{x}_j - \boldsymbol{x}_i\| \leq r_c\},$$

  and $n_i = |\mathcal{N}_i|$.
- For each $\mathcal{N}_i$, define

$$R_i := (\boldsymbol{x}_{j_1} - \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{j_{n_i}} - \boldsymbol{x}_i)^T \in \mathbb{R}^{n_i \times d}$$

  for $j_k \in \mathcal{N}_i$. Then, the matrix

$$\Omega_i = R_i^T R_i$$

  is invariant with respect to both translation and rotation.
- Consider the function of the following form

$$f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \sum_{i=1}^{n} h_i(\Omega_i).$$

  It is obvious that f is invariant to translation and rotation.
- Parameterize $\{h_i\}$ with neural network models.

The Effect of Symmetry Preservation

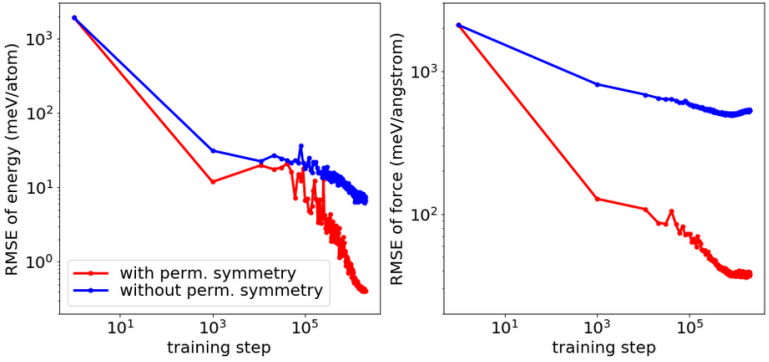


Figure 5: The effect of symmetry preservation on testing accuracy.

We refer to https://geometricdeeplearning.com/ for more resources on this topic.

Summary

- Fully-connected networks
- Convolutional networks
- Recurrent neural networks.
- Residual neural networks.
- Symmetry-preserving is in crucial in practice.

Other important but uncovered architectures: **Transformer** (we will discuss it later), **Graph neural network**.

Reading:

- MLP: https://www.deeplearningbook.org/contents/mlp.html
- CNN:
  - https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/
  - https://www.deeplearningbook.org/contents/convnets.html
- RNN: https://www.deeplearningbook.org/contents/rnn.html
- Geometric Deep Learning: https://geometricdeeplearning.com.