Neural Networks

Instructor: Lei Wu [1]

Mathematical Introduction to Machine Learning

Peking University, Fall 2024

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

Outline

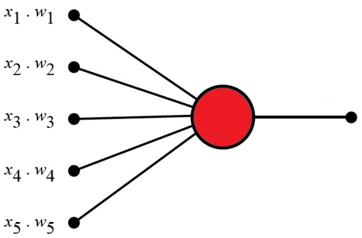
1. Fully connected networks (aka MLP)
2. Convolution neural networks (CNN)
3. Recurrent neural networks (RNN)
4. Symmetry-preserving neural networks

The perceptron model: 1943-1957

- In 1943, Warren McCulloch and Walter Pitts developed the **perceptron algorithm**:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}.$$

  The perceptron is a simplified model of a biological neuron

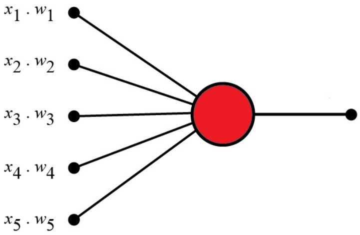


The perceptron model: 1943-1957

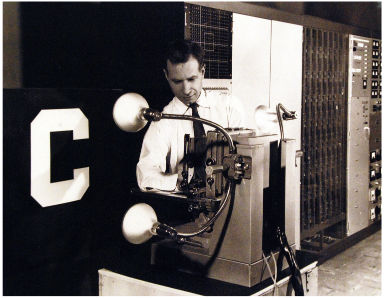
- In 1943, Warren McCulloch and Walter Pitts developed the **perceptron algorithm**:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}.$$

The perceptron is a simplified model of a biological neuron



- In 1957, the first implementation was a machine built in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt, funded by the United States Office of Naval Research.



Two-layer neural networks

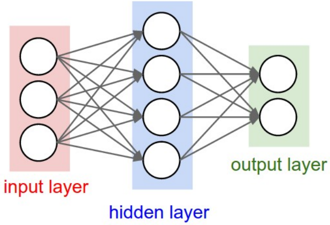
In 1965, Alexey Ivakhnenko and Valentin Lapa developed the Multiplayer Perceptron (MLP).

Two-layer neural networks

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

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

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



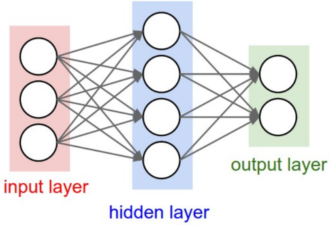
where $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times k}, \boldsymbol{c} \in \mathbb{R}^m$. Here, $\theta = \{A, B, \boldsymbol{c}\}$ are the trainable parameters.

Two-layer neural networks

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

- A two-layer network defines function define 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 $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times k}, \boldsymbol{c} \in \mathbb{R}^m$. Here, $\theta = \{A, B, \boldsymbol{c}\}$ are the trainable parameters.

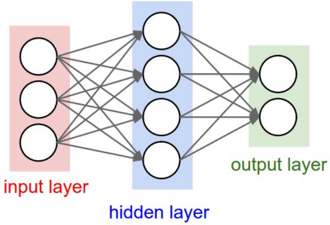
- $\sigma : \mathbb{R} \mapsto \mathbb{R}$ is the (nonlinear) activation function, e.g. $\sigma(z) = e^z/(1+e^z)$ (sigmoid). When z is a vector or matrix, $\sigma(z)$ should be understood in an **element-wise** manner.

Two-layer neural networks

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

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

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



  where $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times k}, \boldsymbol{c} \in \mathbb{R}^m$. 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

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

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.

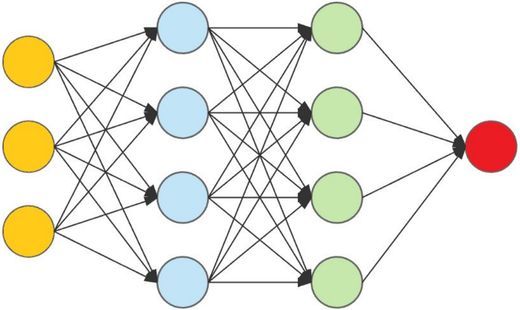


Figure 1: Play with MLP: https://playground.tensorflow.org.

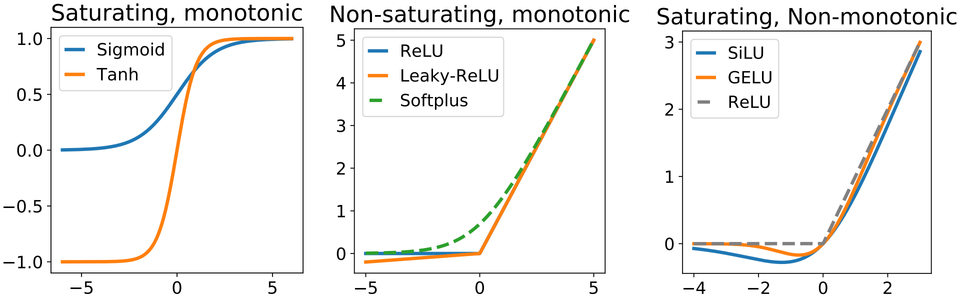
Activation Functions

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

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.

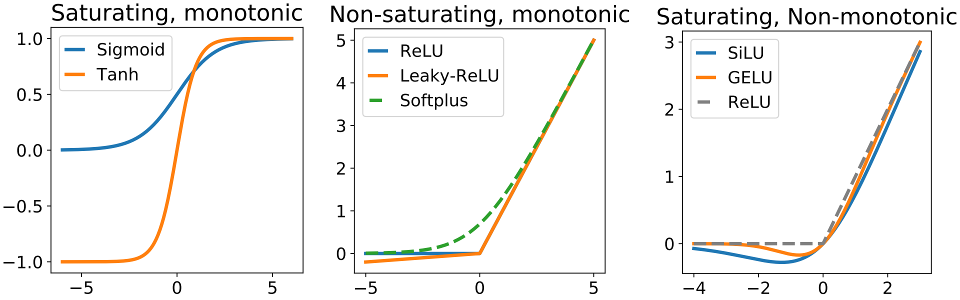
- The Gaussian error linear unit (GELU) and sigmoid linear unit (SiLU) becomes popular recently.
- **Question:** Why is ReLU not good choice for solving scientific computing problems?

Comparison of activation functions



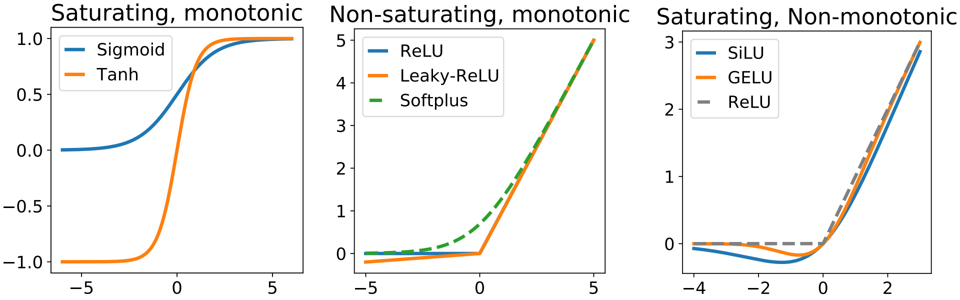
- Softplus, GELU, and SiLU can be viewed as smoothed versions of ReLU. Currently, ReLU and ReLU variants are the most popular ones.

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 Property (UAP)

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 $f_m(\boldsymbol{x};\theta) = \sum_{j=1}^{m} a_j \sigma(\boldsymbol{b}_j^T \boldsymbol{x} + c_j)$ such that

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

- The above theorem can be extended to general non-polynomial activation functions, including all the commonly-used activation functions.
- The above theorem says that two-layer neural networks can approximate any continuous function.
- Here, we only state theorem with the proof deferred to the advanced topics.

Remarks

The universal approximation theorem is an analog of Weierstrass Theorem in mathematical analysis which asserts that on compact domains, **continuous functions can be approximated by polynomials**.

By itself, it does not explain the success of neural network approximations over polynomial approximations (in high dimensions).

Convolution Neural Networks

Question:

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

Convolutional neural networks

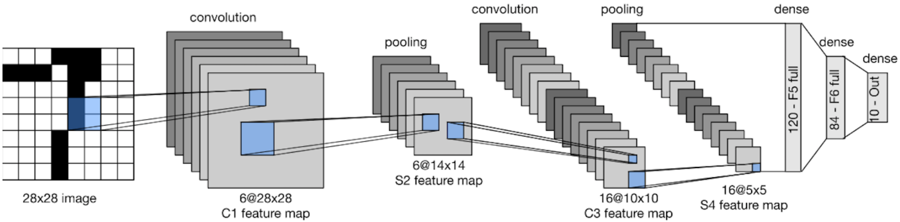


Figure 2: LeNet-5 for MNIST dataset

- 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**).

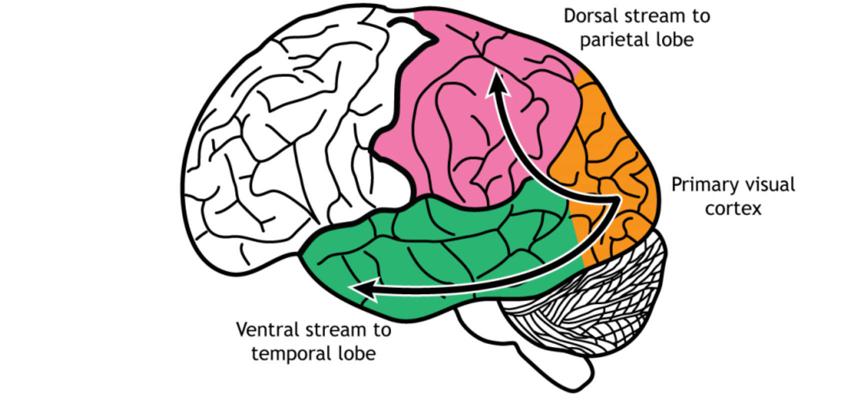
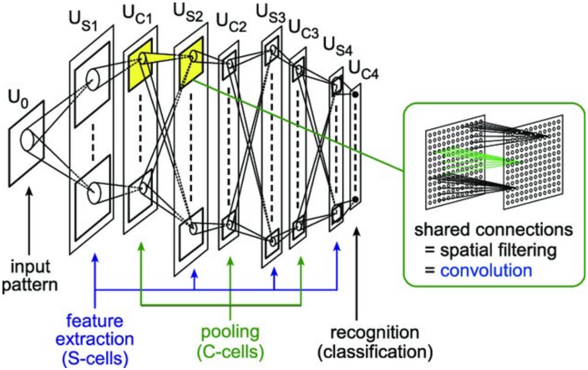


Figure 3: If you are interested in learning how the human brain processes visual signals, we recommend visiting this link.

History of CNNs

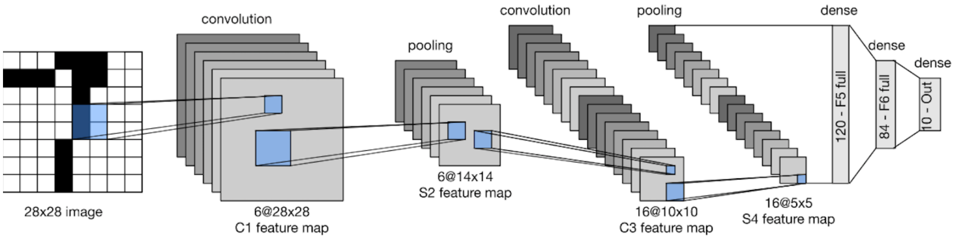
- **In 1969, Kunihiko Fukushima** introduced the first deep ReLU CNN, called **Neocognitron**, featuring fixed filters:
  - The "S-layer": a weight-shared receptive field layer, later termed conv. layers.
  - The "C-layer": a downsampling layer.

But the **filters are not learnable**.



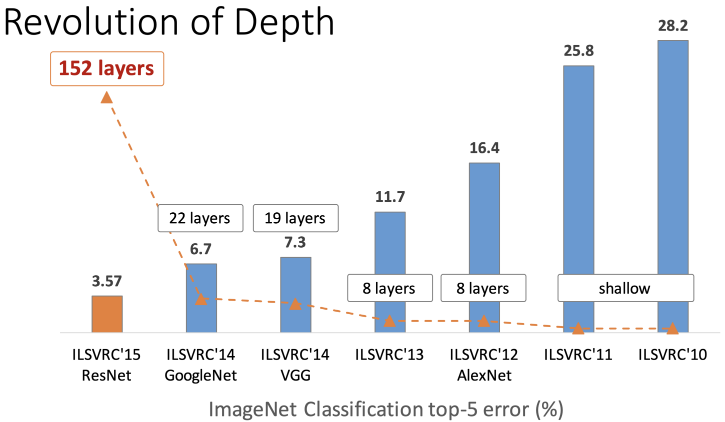
History of CNNs

- 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.



1D Convolutional transform

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

1D Convolutional 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

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

1D Convolutional 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

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

- **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

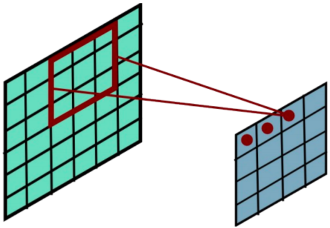
$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-3+1} \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & w_3 & \cdots & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & \cdots & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & w_1 & w_2 & w_3 \end{pmatrix}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

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

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



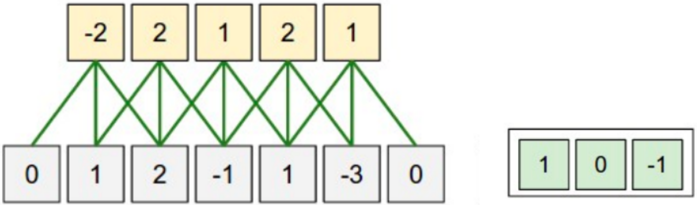
- **Sliding window!**
- **Small filter size!**

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 $y = x * w = (-2, 2, 1, 2, 1) \in \mathbb{R}^5$.

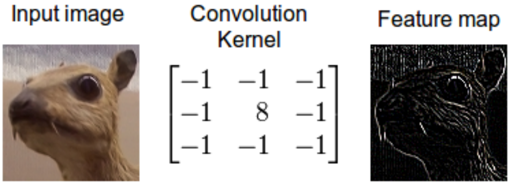


Motivation to use convolutional transforms

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

Motivation to use convolutional 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.

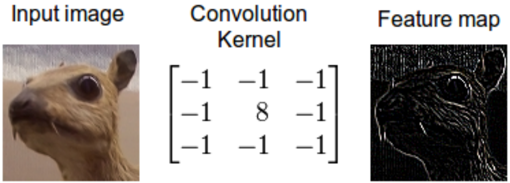


$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

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

Motivation to use convolutional 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.



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

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

- The fully-connected linear transform: $W\boldsymbol{x} + b$, is not easy to capture the local structures.

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.

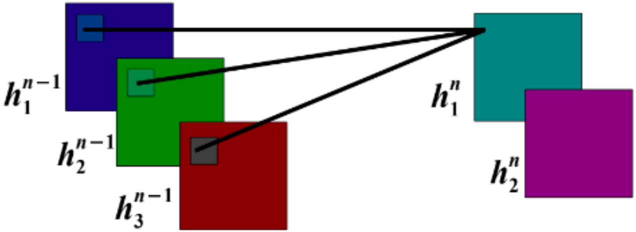
Channels

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.



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 $b \in \mathbb{R}^{C_o}$.

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

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.

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.

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.

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.

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.

Convolutional layer (Cont'd)

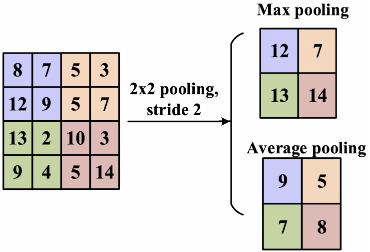
- Let $h^i \in \mathbb{R}^{W_i \times H_i \times C_i}$ and $h^o \in \mathbb{R}^{W_o \times H_o \times C_o}$ denote the input and output feature map, respectively. The filter $w \in \mathbb{R}^{k \times k \times C_i \times C_o}$ is 4-**order tensor** and bias $b \in \mathbb{R}^{C_o}$.
- Mathematically,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

$$h^o_t = \sum_{s=1}^{C_i} h^i_s * w^{s,t} + b^t,$$

  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^o_t is the t-th channel of output feature map.
  - h^i_s 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

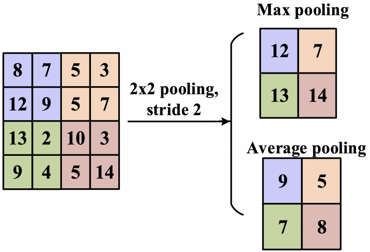
- **Pooling (aka down-sampling):** There are two types of pooling: max pooling and average pooling.



- **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.

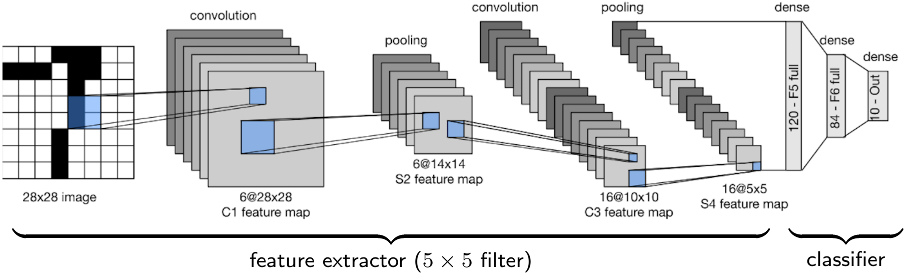
A Closer Look at LeNet-5

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

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

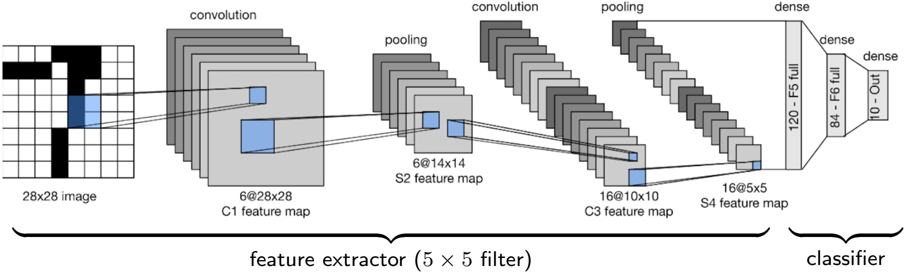
A Closer Look at LeNet-5

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



A Closer Look at LeNet-5

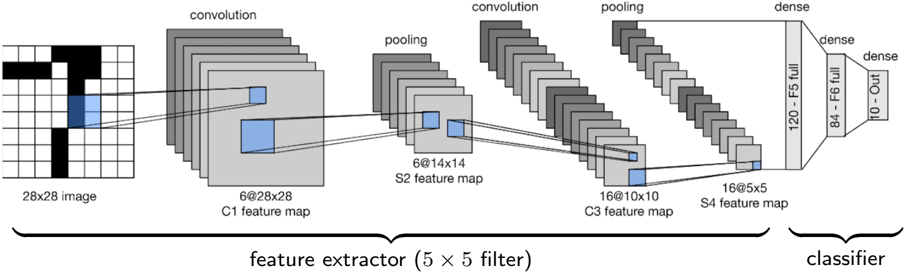
- **LeNet-5:** Convolutional layers + Fully-connected layers + Softmax.



- 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.

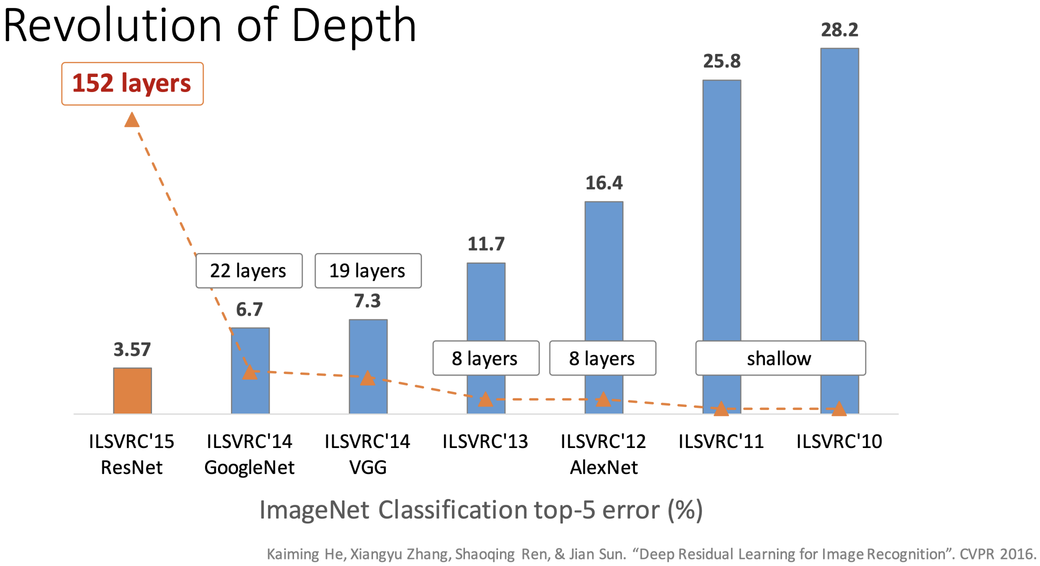
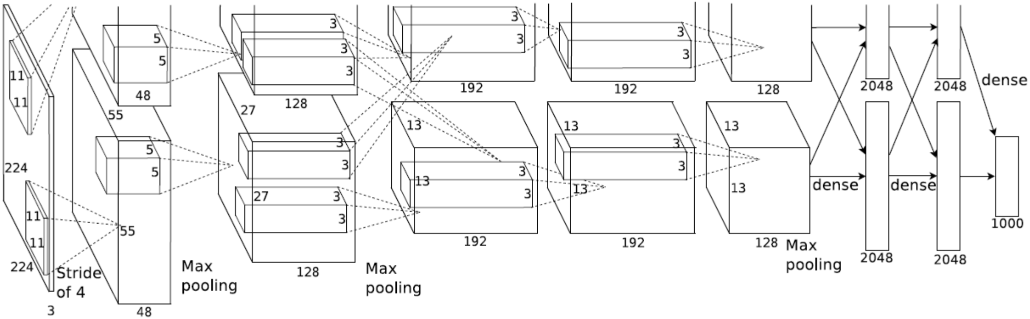


Figure 4: Taken from Kaiming He's slide.

AlexNet: 2012



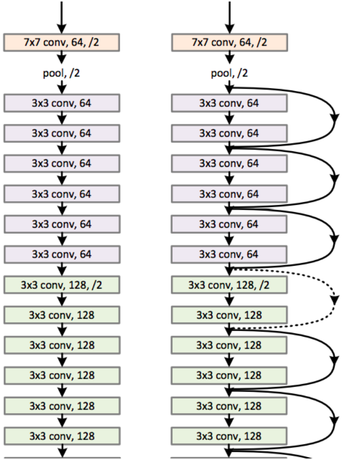
Contribution:

- 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

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

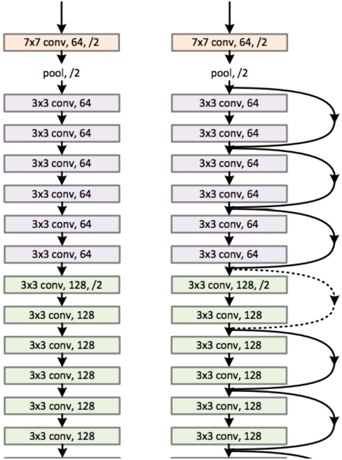
Residual net

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

$h(\cdot; \theta^{l})$ 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

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

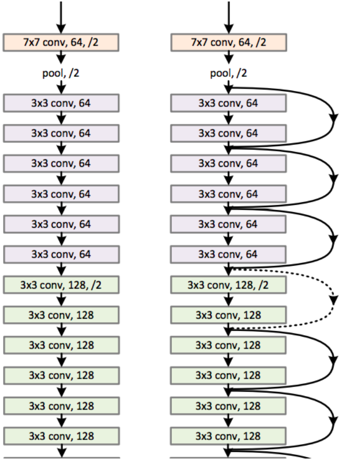
Residual net

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

$h(\cdot; \theta^l)$ 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

Sequence predictions:

- Speech-to-text and text-to-speech.
- Machine translation.
- Sentiment analysis.
- Caption generalization.

When both input and output are sequence, this task is called **sequence-to-sequence** prediction.

Abstraction:

- **Input:** $\boldsymbol{x} = (x_1, x_2, \ldots, x_T)$ with $x_t \in \mathbb{R}^{d_x}$.
- **Output:** $\boldsymbol{y} = (y_1, y_2, \ldots, y_T)$ with $y_t \in \mathbb{R}^{d_y}$.
- **Target:**

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

  Non-Markovian process!

Recurrent neural networks

- **Code/Feature:** $\boldsymbol{h} = (h_1, h_2, \ldots, h_T)$, with $h_t \in \mathbb{R}^{d_h}$ encodes the information of $(x_1, x_2, \ldots, x_t)$ through
$$h_t = f(x_t, h_{t-1}).$$

Recurrent neural networks

- **Code/Feature:** $\boldsymbol{h} = (h_1, h_2, \dots, h_T)$, with $h_t \in \mathbb{R}^{d_h}$ encodes the information of $(x_1, x_2, \dots, x_t)$ through
$$h_t = f(x_t, h_{t-1}).$$
- **Model output**:
$$y_t = g(y_{t-1}, h_t)$$

Recurrent neural networks

- **Code/Feature:** $\boldsymbol{h} = (h_1, h_2, \ldots, h_T)$, with $h_t \in \mathbb{R}^{d_h}$ encodes the information of $(x_1, x_2, \ldots, x_t)$ through
$$h_t = f(x_t, h_{t-1}).$$
- **Model output**:
$$y_t = g(y_{t-1}, h_t)$$
- **Parameterization:** Use fully or convolutional networks to parameterize f and g.

Recurrent neural networks

- **Code/Feature:** $\boldsymbol{h} = (h_1, h_2, \ldots, h_T)$, with $h_t \in \mathbb{R}^{d_h}$ encodes the information of $(x_1, x_2, \ldots, x_t)$ through
$$h_t = f(x_t, h_{t-1}).$$
- **Model output**:
$$y_t = g(y_{t-1}, h_t)$$
- **Parameterization:** Use fully or convolutional networks to parameterize f and g.
- Note that f and g are shared among all time t's.

Vanilla RNN

- **Update Formulation:**

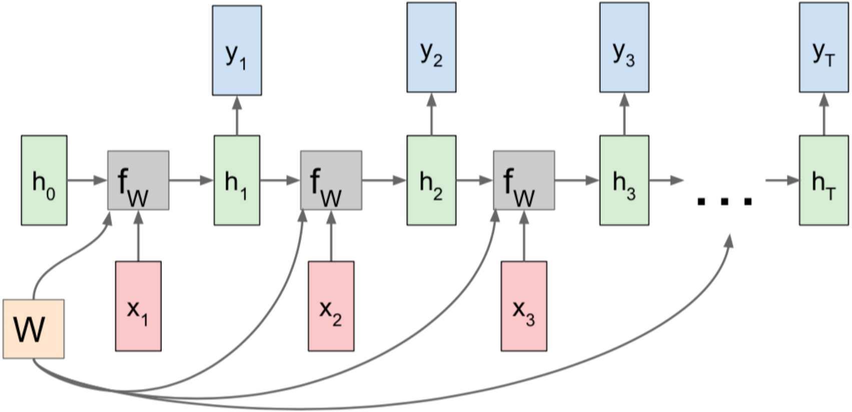
$$h_t = \texttt{tanh}(W_{hh} h_{t-1} + W_{hx} x_t)$$
$$y_t = W_{yh} h_t$$

Vanilla RNN

- **Update Formulation:**

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

- **Visualization:**



Long Short Term Memory (LSTM)

- **Gate update:**

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

Long Short Term Memory (LSTM)

- **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:**

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

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.

Long Short Term Memory (LSTM)

- **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:**

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

  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 Factors:**

Long Short Term Memory (LSTM)

- **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:**

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

  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 Factors:**
  - The extra state c_t (aka cell) is used to store long-time memory. In contrast, h_t store short-time memory.

Long Short Term Memory (LSTM)

- **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:**

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

  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 Factors:**
  - The extra state c_t (aka cell) is used to store long-time memory. In contrast, h_t store short-time memory.
  - Gate mechanism.

Encoder-decoder structures

What if the output and input have different lengths?

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

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

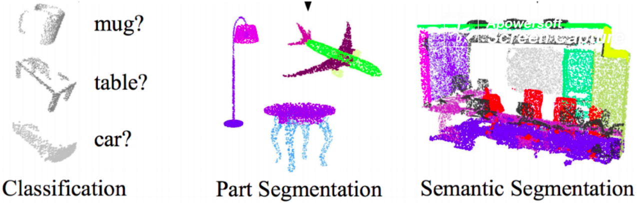
  for any permutation $\sigma \in S_n$ and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$.
- We can also understand f as a function over the **set** $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$.

Example:

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

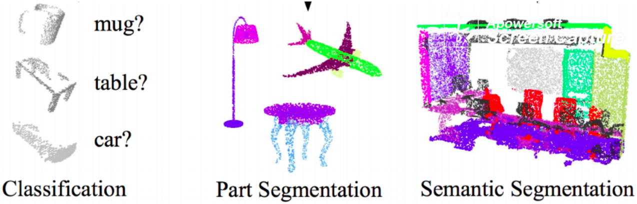
Applications

- Point cloud.



Applications

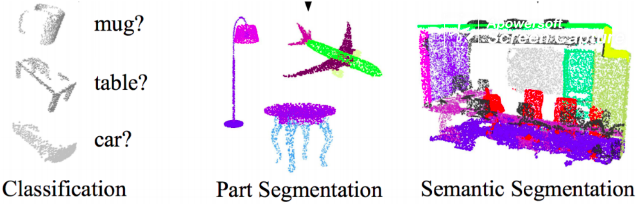
- Point cloud.



- Wave functions of bosons in quantum physics.

Applications

- Point cloud.



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

Deep set models

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

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

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

Approximation of permutation-invariant functions

- UAP 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 UAP while preserving the symmetry?

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

Approximation of permutation-invariant functions

- UAP 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 UAP while preserving the symmetry?

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

Theorem 2 (Han et al. 2019)

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

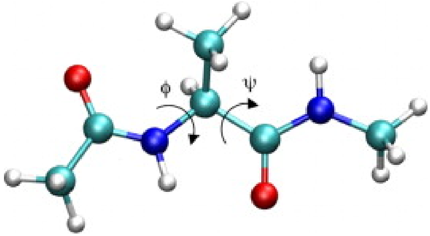
$$f(\boldsymbol{x}_1 + \boldsymbol{b}, \dots, \boldsymbol{x}_n + \boldsymbol{b}) = f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n), \qquad \forall \boldsymbol{b} \in \mathbb{R}^d,$$

  and to be rotational invariant if

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

  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
$$\mathcal{N}_i = \{j \in [n] : \|\boldsymbol{x}_j - \boldsymbol{x}_i\| \le r_c\},$$
and $n_i = |\mathcal{N}_i|$.
- For each $\mathcal{N}_i$, define
$$R_i := (\boldsymbol{x}_{j_1} - \boldsymbol{x}_i, \dots, \boldsymbol{x}_{j_{n_i}} - \boldsymbol{x}_i)^T \in \mathbb{R}^{n_i \times d}$$
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
$$f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n) = \sum_{i=1}^{n} h_i(\Omega_i).$$
It is obvious that f is invariant to translation and rotation.
- Parameterize $\{h_i\}$ with neural network models.

The effect of symmetry preservation

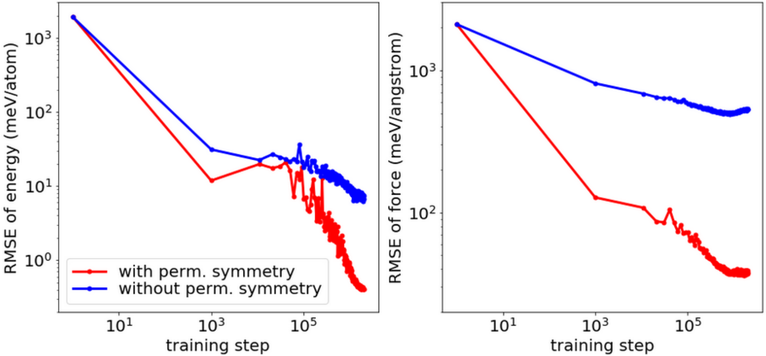


Figure 5: The effect of symmetry preservation on testing accuracy.

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 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.