Lecture 12: Neural Network (Back-Propagation, Activation Functions, Advanced Architectures)

Tao LIN

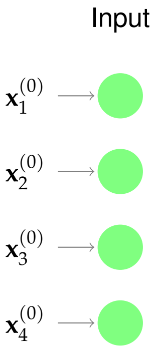
SoE, Westlake University

December 2, 2025

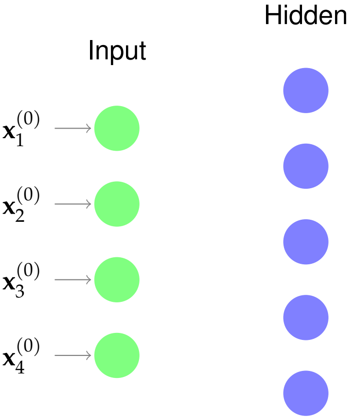
Table of Contents

Table of Contents

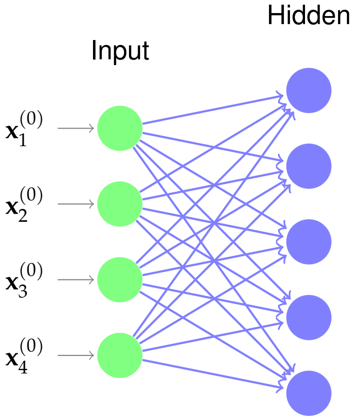
Step-by-Step: MLP Forward Pass



Step-by-Step: MLP Forward Pass

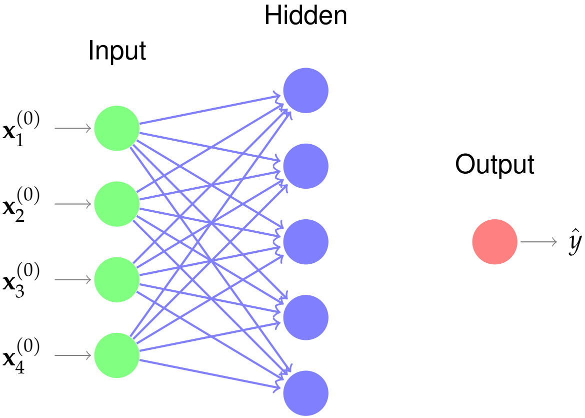


Step-by-Step: MLP Forward Pass



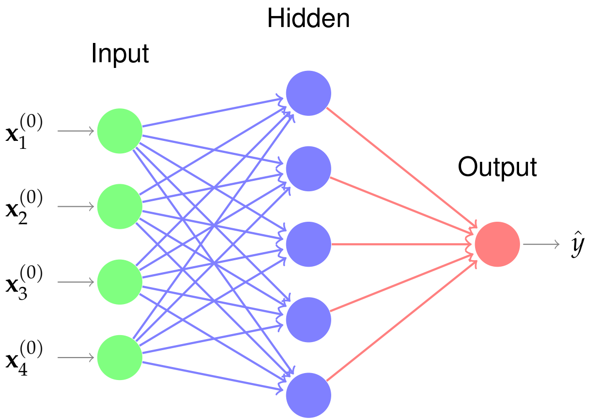
Compute: $\mathbf{z}^{(1)} = (\mathbf{W}^{(1)})^\top \mathbf{x}^{(0)} + \mathbf{b}^{(1)}$

Step-by-Step: MLP Forward Pass



Compute: $\mathbf{z}^{(1)} = (\mathbf{W}^{(1)})^\top \mathbf{x}^{(0)} + \mathbf{b}^{(1)}$

Step-by-Step: MLP Forward Pass



Final: $\hat{y} = \phi((\mathbf{W}^{(2)})^\top \mathbf{x}^{(1)} + b^{(2)})$

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation.

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation. This function is defined by

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation. This function is defined by
    - The biases $\{\mathbf{b}^{(l)}\}_{l\in[L]}$ and weights $\{\mathbf{W}^{(l)}\}_{l\in[L]}$

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation. This function is defined by
  - The biases $\{\mathbf{b}^{(l)}\}_{l\in[L]}$ and weights $\{\mathbf{W}^{(l)}\}_{l\in[L]}$
  - The activation function σ we pick

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation. This function is defined by
  - The biases $\{\mathbf{b}^{(l)}\}_{l\in[L]}$ and weights $\{\mathbf{W}^{(l)}\}_{l\in[L]}$
  - The activation function σ we pick

In practice: both L and K are large — over-parameterized NNs.

NNs extract suitable features from the input

A NN can be decomposed into a feature extractor and the output layer.

- Feature extractor $\mathbb{R}^d \to \mathbb{R}^K$: It transforms data into a suitable representation. This function is defined by
  - The biases $\{\mathbf{b}^{(l)}\}_{l\in[L]}$ and weights $\{\mathbf{W}^{(l)}\}_{l\in[L]}$
  - The activation function σ we pick

  In practice: both L and K are large — over-parameterized NNs.

- The last layer $\mathbb{R}^K \to \mathbb{R}$: It performs the desired ML task, either linear regression or classification.

Table of Contents

Training loss for a regression problem with $S_{\text{train}} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$:

$$\mathcal{L}(f) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n))^2 \,, \tag{1}$$

where

Training loss for a regression problem with $S_{\text{train}} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$:

$$\mathcal{L}(f) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n))^2 \,, \tag{1}$$

where

- f is the function represented by a NN.

Training loss for a regression problem with $S_{\text{train}} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$:

$$\mathcal{L}(f) = \frac{1}{2N}\sum_{n=1}^{N}(y_n - f(\mathbf{x}_n))^2\,, \tag{1}$$

where

- f is the function represented by a NN.
- The overall function $y = f(\mathbf{x}^{(0)})$ can then be written as the composition:

$$f(\mathbf{x}^{(0)}) = f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}^{(0)}).$$

Compact description of output

Compact description of output

- The function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})\,. \tag{2}$$

Compact description of output

- The function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})\,. \tag{2}$$

- Let $\mathbf{W}^{(l)}$ denote the *weight* matrix that connects layer $l-1$ to layer l.

Compact description of output

- The function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})\,. \tag{2}$$

- Let $\mathbf{W}^{(l)}$ denote the *weight* matrix that connects layer $l-1$ to layer l.
- The matrix $\mathbf{W}^{(1)}$ is of dimension $D \times K$, the matrices $\mathbf{W}^{(l)}$, $2 \leq l \leq L$, are of dimension $K \times K$, and the matrix $\mathbf{W}^{(L+1)}$ is of dimension $K \times 1$.

Compact description of output

- The function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})\,. \tag{2}$$

- Let $\mathbf{W}^{(l)}$ denote the *weight* matrix that connects layer $l-1$ to layer l.
- The matrix $\mathbf{W}^{(1)}$ is of dimension $D \times K$, the matrices $\mathbf{W}^{(l)}$, $2 \leq l \leq L$, are of dimension $K \times K$, and the matrix $\mathbf{W}^{(L+1)}$ is of dimension $K \times 1$.
- The entries of each matrix $\mathbf{W}$ are given by

$$\mathbf{W}_{i,j}^{(l)} = w_{i,j}^{(l)}\,, \tag{3}$$

Compact description of output

- The function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})\,. \tag{2}$$

- Let $\mathbf{W}^{(l)}$ denote the *weight* matrix that connects layer $l-1$ to layer l.
- The matrix $\mathbf{W}^{(1)}$ is of dimension $D \times K$, the matrices $\mathbf{W}^{(l)}$, $2 \leq l \leq L$, are of dimension $K \times K$, and the matrix $\mathbf{W}^{(L+1)}$ is of dimension $K \times 1$.
- The entries of each matrix $\mathbf{W}$ are given by

$$\mathbf{W}_{i,j}^{(l)} = w_{i,j}^{(l)}\,, \tag{3}$$

where $w_{i,j}^{(l)}$ is the edge weight that connects node i on layer $l-1$ to node j on layer l.

The back-propagation algorithm

Cost function:

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n^{(0)}) \right)^2 ,$$

where $\mathbf{x}_n^{(l)} = f^{(l)}(\mathbf{x}_n^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}_n^{(l-1)} + \mathbf{b}^{(l)})$.

The back-propagation algorithm

Cost function:

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n^{(0)})\right)^2 ,$$

where $\mathbf{x}_n^{(l)} = f^{(l)}(\mathbf{x}_n^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}_n^{(l-1)} + \mathbf{b}^{(l)})$.

Recall that we aim to compute:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \qquad l = 1, \cdots, L+1,$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, \qquad l = 1, \cdots, L+1.$$

Let's use two quantities (i.e., $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$) to aid the computation:

Let's use two quantities (i.e., $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$) to aid the computation:

- Quantity computed in the **forward pass**:

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)} \tag{4}$$

  be the input at the l-th layer before applying the activation function, where $\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$.

Let's use two quantities (i.e., $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$) to aid the computation:

- Quantity computed in the **forward pass**:

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)} \tag{4}$$

  be the input at the l-th layer before applying the activation function, where $\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$.

- Quantity computed in the **backward pass**:

$$\begin{aligned}
\delta_j^{(l)} &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} && (5)\\
&= \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} && (6)\\
&= \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}) \,, && (7)
\end{aligned}$$

Let's use two quantities (i.e., $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$) to aid the computation:

- Quantity computed in the **forward pass**:

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)} \tag{4}$$

  be the input at the l-th layer before applying the activation function, where $\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$.

- Quantity computed in the **backward pass**:

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} && (5) \\ &= \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} && (6) \\ &= \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}) \,, && (7) \end{aligned}$$

  In vector form, we can write this as

$$\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}) \,, \tag{8}$$

  where $\odot$ denotes the Hadamard product (the point-wise multiplication of vectors).

Now that we have both $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$ let us get back to our initial goal.

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}}}_{\mathbf{x}_i^{(l-1)}} = \delta_j^{(l)} \mathbf{x}_i^{(l-1)}$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}_{1} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)} .$$

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.
- We are given in addition a sample $(\mathbf{x}_n, y_n)$.

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.
- We are given in addition a sample $(\mathbf{x}_n, y_n)$.
- We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \qquad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, \qquad l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n)\right)^2.$$

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.
- We are given in addition a sample $(\mathbf{x}_n, y_n)$.
- We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \qquad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, \qquad l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n)\right)^2.$$

Forward pass: Set $\mathbf{x}^{(0)} = \mathbf{x}_n$. Compute for $l = 1, \cdots, L+1$,

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \qquad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.
- We are given in addition a sample $(\mathbf{x}_n, y_n)$.
- We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \qquad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, \qquad l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n)\right)^2.$$

Forward pass: Set $\mathbf{x}^{(0)} = \mathbf{x}_n$. Compute for $l = 1, \cdots, L+1$,

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \qquad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

Backward pass: Set $\delta^{(L+1)} = -2(y_n - \mathbf{x}^{(L+1)})\phi'(z^{(L+1)})$. Compute for $l = L, \cdots 1$,

$$\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}).$$

Summary: Backpropagation Algorithm for Computing the Derivatives

Settings: We are given a NN with L hidden layers

- All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed.
- We are given in addition a sample $(\mathbf{x}_n, y_n)$.
- We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \qquad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, \qquad l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = \left(y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n)\right)^2.$$

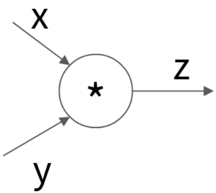
Forward pass: Set $\mathbf{x}^{(0)} = \mathbf{x}_n$. Compute for $l = 1, \cdots, L+1$,

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \qquad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

Backward pass: Set $\delta^{(L+1)} = -2(y_n - \mathbf{x}^{(L+1)})\phi'(z^{(L+1)})$. Compute for $l = L, \cdots 1$,

$$\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}).$$

Modularized implementation: forward / backward API



(x,y,z are scalars)

Gate / Node / Function object: Actual PyTorch code

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to cache some values for use in backward

Upstream gradient

Multiply upstream and local gradients

Table of Contents

Table of Contents

The sigmoid

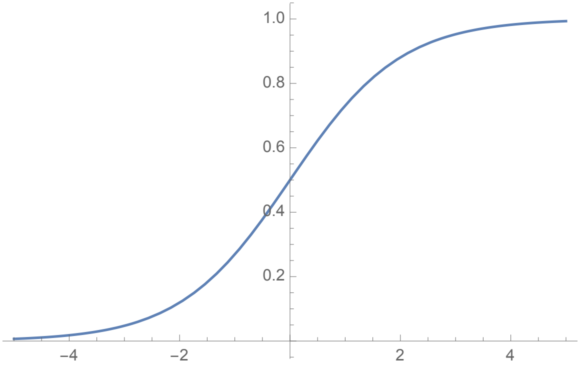


Figure: The sigmoid function $\phi(x)$.

$$\phi(x) = \frac{1}{1+e^{-x}} \tag{9}$$

- The sigmoid is always positive (not really an issue) and that it is bounded.
- For $|x|$ large, $\phi'(x) \sim 0$. This can cause the gradient to become very small ("vanishing gradient problem"), sometimes making learning slow.

Tanh

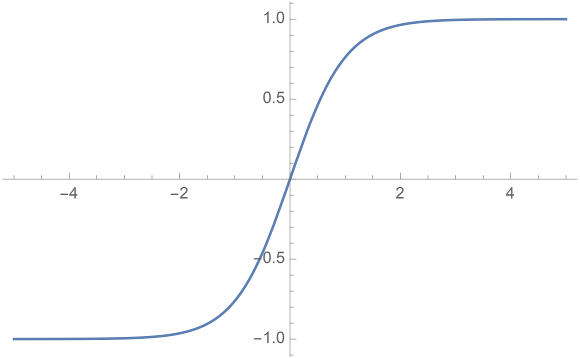


Figure: $\tanh(x)$.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\phi(2x) - 1 \qquad (10)$$

- $\tanh(x)$ is "balanced" (positive and negative) and that it is bounded.
- It has the same problem as the sigmoid function, namely for $|x|$ large, $\tanh'(x) \sim 0$.

Rectified linear Unit – ReLU

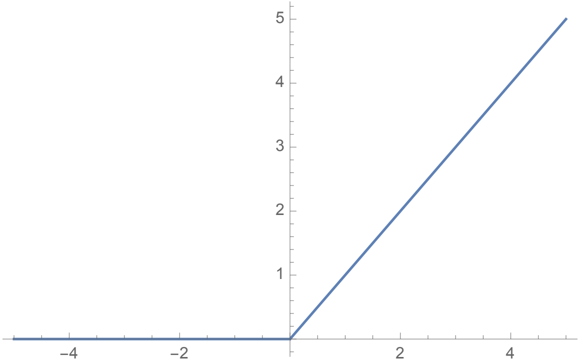


Figure: The ReLU $(x)_+$.

$$(x)_+ = \max\{0, x\}, \tag{11}$$

Rectified linear Unit – ReLU

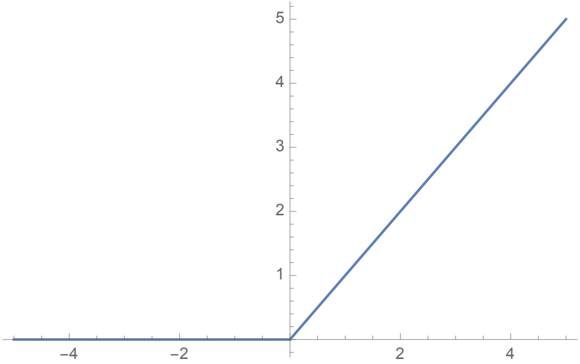


Figure: The ReLU $(x)_+$.

$$(x)_+ = \max\{0, x\}, \tag{11}$$

- ReLU is always positive and is unbounded.

Rectified linear Unit – ReLU

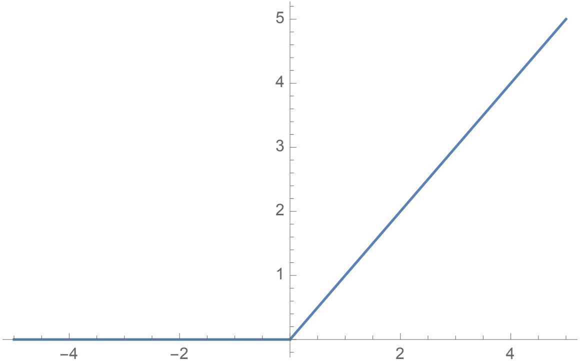


Figure: The ReLU $(x)_+$.

$$(x)_+ = \max\{0, x\}, \tag{11}$$

- ReLU is always positive and is unbounded.
- Its derivative is 1 (and does not vanish) for positive values of x (it has 0 derivative for negative values of x though)

Leaky ReLU

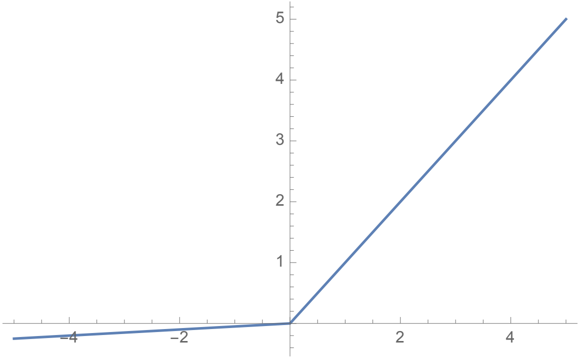


Figure: LReLU with $\alpha = 0.05$

In order to solve the 0-derivative problem of the ReLU (for negative values of x) one can add a very small slope α in the negative part.

$$f(x) = \max\{\alpha x, x\} \tag{12}$$

Leaky ReLU

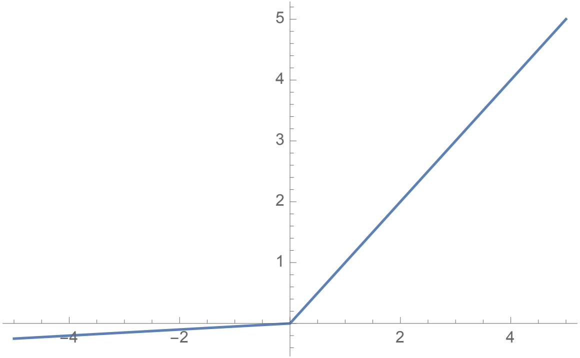


Figure: LReLU with $\alpha = 0.05$

In order to solve the 0-derivative problem of the ReLU (for negative values of x) one can add a very small slope α in the negative part.

$$f(x) = \max\{\alpha x, x\} \tag{12}$$

- The constant α is of course a hyper-parameter that can be optimized.

Maxout

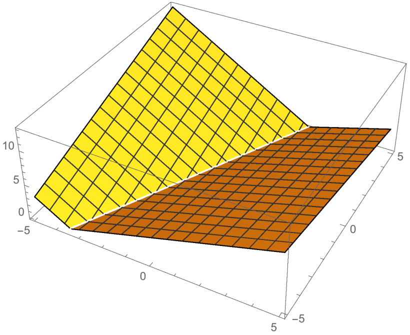


Figure: Maxout function with two terms, max$\{x_1 - 0.5x_2 + 1, -2x_1 + x_2 - 2\}$.

The maxout generalizes ReLU and LReLU.

$$f(\mathbf{x}) = \max\{\mathbf{x}^\top \mathbf{w}_1 + b_1, \cdots, \mathbf{x}^\top \mathbf{w}_k + b_k\} \tag{13}$$

Maxout

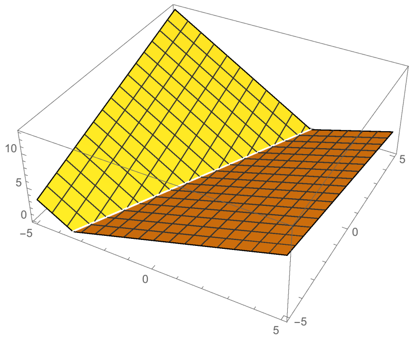


Figure: Maxout function with two terms, max$\{x_1 - 0.5x_2 + 1, -2x_1 + x_2 - 2\}$.

The maxout generalizes ReLU and LReLU.

$$f(\mathbf{x}) = \max\{\mathbf{x}^\top \mathbf{w}_1 + b_1, \cdots, \mathbf{x}^\top \mathbf{w}_k + b_k\} \quad (13)$$

- This activation function is quite different from the previous cases.

Maxout

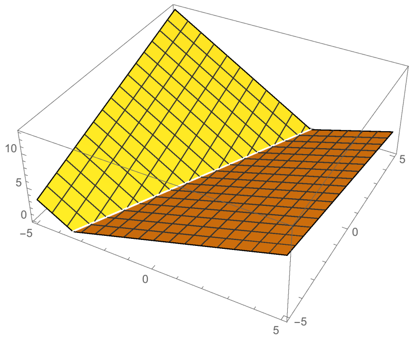


Figure: Maxout function with two terms, max$\{x_1 - 0.5x_2 + 1, -2x_1 + x_2 - 2\}$.

The maxout generalizes ReLU and LReLU.

$$f(\mathbf{x}) = \max\{\mathbf{x}^\top \mathbf{w}_1 + b_1, \cdots, \mathbf{x}^\top \mathbf{w}_k + b_k\} \tag{13}$$

- This activation function is quite different from the previous cases.
- In the previous cases we computed a weighted sum and then applied the activation function to it, whereas here we compute two or more different weighted sums and then choose the maximum.

Table of Contents

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.
- Images have 2D spatial structure ($\mathbf{x} \in \mathbb{R}^{W \times H \times C}$).

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.
- Images have 2D spatial structure ($\mathbf{x} \in \mathbb{R}^{W \times H \times C}$).
- Applying MLPs directly to images is problematic:

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.
- Images have 2D spatial structure ($\mathbf{x} \in \mathbb{R}^{W \times H \times C}$).
- Applying MLPs directly to images is problematic:
    - Variable image sizes need different weight matrices $\mathbf{W}$.

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.
- Images have 2D spatial structure ($\mathbf{x} \in \mathbb{R}^{W \times H \times C}$).
- Applying MLPs directly to images is problematic:
  - Variable image sizes need different weight matrices $\mathbf{W}$.
  - Fixed-size images still lead to huge weight matrices ($(W \times H \times C) \times D$ parameters).

Introduction: Why Not MLPs for Images?

- MLPs map unstructured vectors $\mathbf{x} \in \mathbb{R}^D$ to outputs.
- Images have 2D spatial structure ($\mathbf{x} \in \mathbb{R}^{W \times H \times C}$).
- Applying MLPs directly to images is problematic:
  - Variable image sizes need different weight matrices $\mathbf{W}$.
  - Fixed-size images still lead to huge weight matrices ($(W \times H \times C) \times D$ parameters).
  - Lack of translation invariance: Pattern recognized in one location may not be recognized if shifted.

Lack of Translation Invariance in MLPs

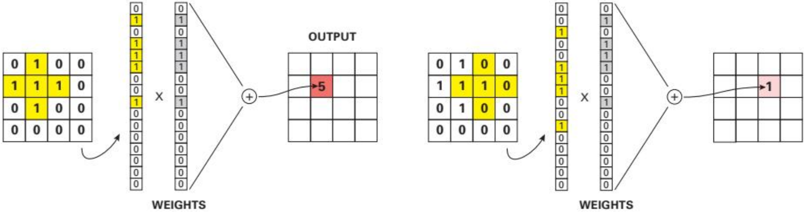


Figure: Detecting patterns with MLPs lacks translation invariance. A matched filter (weight vector) gives a strong response when the pattern aligns perfectly (left) but a weak response when shifted (right).

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.
  - Compare each patch with small weight matrices (filters/kernels).

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.
  - Compare each patch with small weight matrices (filters/kernels).
  - Filters act as learnable templates for parts of objects.

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.
  - Compare each patch with small weight matrices (filters/kernels).
  - Filters act as learnable templates for parts of objects.
- Advantages:

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.
  - Compare each patch with small weight matrices (filters/kernels).
  - Filters act as learnable templates for parts of objects.
- Advantages:
  - Reduced parameters (small filters, e.g., 3x3, 5x5).

The CNN Solution: Convolution

- Convolutional Neural Networks (CNNs) replace matrix multiplication with convolution.
- Basic Idea:
  - Divide input into overlapping 2D patches.
  - Compare each patch with small weight matrices (filters/kernels).
  - Filters act as learnable templates for parts of objects.
- Advantages:
  - Reduced parameters (small filters, e.g., 3x3, 5x5).
  - Translation invariance (filters applied across all locations).

Convolution as Template Matching

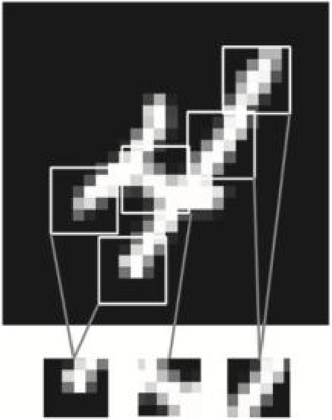


Figure: Classifying a digit by matching discriminative features (templates) in specific relative locations.

Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.

Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.
- Discrete convolution (vector $\mathbf{w}$, vector $\mathbf{x}$):

Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.
- Discrete convolution (vector $\mathbf{w}$, vector $\mathbf{x}$):
  - Flip the weight vector $\mathbf{w}$.

Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.
- Discrete convolution (vector $\mathbf{w}$, vector $\mathbf{x}$):
  - Flip the weight vector $\mathbf{w}$.
  - Slide $\mathbf{w}$ over $\mathbf{x}$.

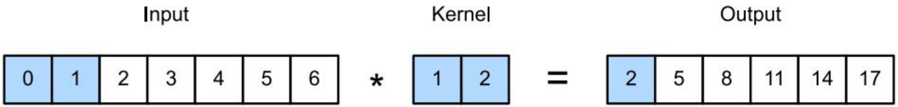
Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.
- Discrete convolution (vector $\mathbf{w}$, vector $\mathbf{x}$):
  - Flip the weight vector $\mathbf{w}$.
  - Slide $\mathbf{w}$ over $\mathbf{x}$.
  - Compute element-wise product sum at each position.

Common Layers: Convolution in 1D

- Continuous convolution: $[f \oplus g](z) = \int f(u)g(z-u)du$.
- Discrete convolution (vector $\mathbf{w}$, vector $\mathbf{x}$):
  - Flip the weight vector $\mathbf{w}$.
  - Slide $\mathbf{w}$ over $\mathbf{x}$.
  - Compute element-wise product sum at each position.
- Example: $[\mathbf{w} \circledast \mathbf{x}](i) = \sum_{u=0}^{L-1} w_u x_{i+u}$ (often means cross-correlation in DL).

1D Convolution / Cross-Correlation Example



as

Figure: 1D cross-correlation: Sliding a filter (bottom) over an input sequence (top) to produce an output sequence (middle).

Note: Deep learning libraries often implement cross-correlation but call it convolution.

Convolution in 2D

- Extends 1D concept to 2D inputs (images) and 2D filters (kernels).

Convolution in 2D

- Extends 1D concept to 2D inputs (images) and 2D filters (kernels).
- Formula: $[\mathbf{W} \circledast \mathbf{X}](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$

Convolution in 2D

- Extends 1D concept to 2D inputs (images) and 2D filters (kernels).
- Formula: $[\mathbf{W} \circledast \mathbf{X}](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$
- Filter $\mathbf{W}$ slides over the input image $\mathbf{X}$.

Convolution in 2D

- Extends 1D concept to 2D inputs (images) and 2D filters (kernels).
- Formula: $[\mathbf{W} \circledast \mathbf{X}](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$
- Filter $\mathbf{W}$ slides over the input image $\mathbf{X}$.
- Output at (i,j) is the weighted sum of the input patch centered at (i,j).

Step-by-Step: 2D Convolution Operation

Input $\mathbf{X}$

x_{03}	x_{13}	x_{23}	x_{33}
x_{02}	x_{12}	x_{22}	x_{32}
x_{01}	x_{11}	x_{21}	x_{31}
x_{00}	x_{10}	x_{20}	x_{30}

Filter $\mathbf{W}$

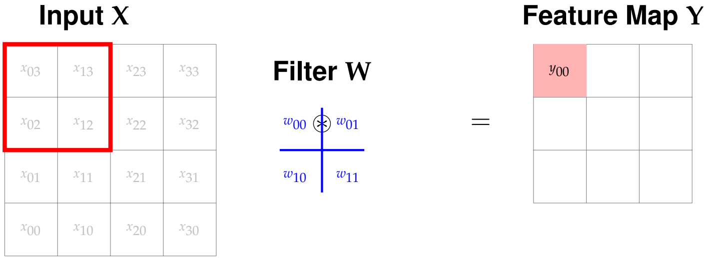
$\circledast$

w_{00}	w_{01}
w_{10}	w_{11}

$=$

Feature Map $\mathbf{Y}$

Step-by-Step: 2D Convolution Operation



Step 1: $$\mathbf{y}_{00} = \sum \mathbf{W} \odot \mathbf{X}_{0:2,0:2}$$

Step-by-Step: 2D Convolution Operation

Input $\mathbf{X}$

x_{03}	x_{13}	x_{23}	x_{33}
x_{02}	x_{12}	x_{22}	x_{32}
x_{01}	x_{11}	x_{21}	x_{31}
x_{00}	x_{10}	x_{20}	x_{30}

Filter $\mathbf{W}$

$\circledast$

w_{00}	w_{01}
w_{10}	w_{11}

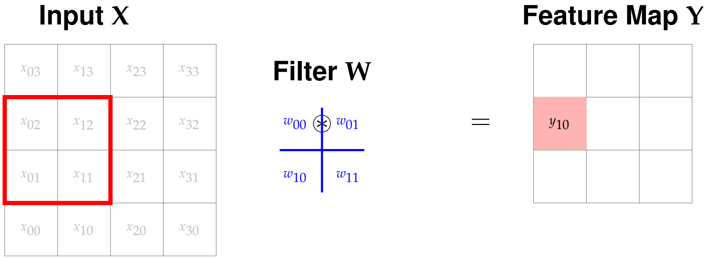
$=$

Feature Map $\mathbf{Y}$

	y_{01}	

Step 2: Slide stride=1. $\mathbf{y}_{01} = \sum \mathbf{W} \odot \mathbf{X}_{0:2,1:3}$

Step-by-Step: 2D Convolution Operation



Step 3: Next row. $\mathbf{y}_{10} = \sum \mathbf{W} \odot \mathbf{X}_{1:3,0:2}$

Step-by-Step: 2D Convolution Operation

Input $\mathbf{X}$

x_{03}	x_{13}	x_{23}	x_{33}
x_{02}	x_{12}	x_{22}	x_{32}
x_{01}	x_{11}	x_{21}	x_{31}
x_{00}	x_{10}	x_{20}	x_{30}

Filter $\mathbf{W}$

$$\circledast \quad \begin{array}{c|c} w_{00} & w_{01} \\ \hline w_{10} & w_{11} \end{array} \quad =$$

Feature Map $\mathbf{Y}$

y	y	y
y	y	y
y	y	y

Process repeats for all spatial locations.

2D Convolution as Feature Detection

- Output $\mathbf{Y} = \mathbf{W} \circledast \mathbf{X}$ is called a **feature map**.
- Output is large where the image patch matches the filter $\mathbf{W}$.
- Example: Filter matching a diagonal line.

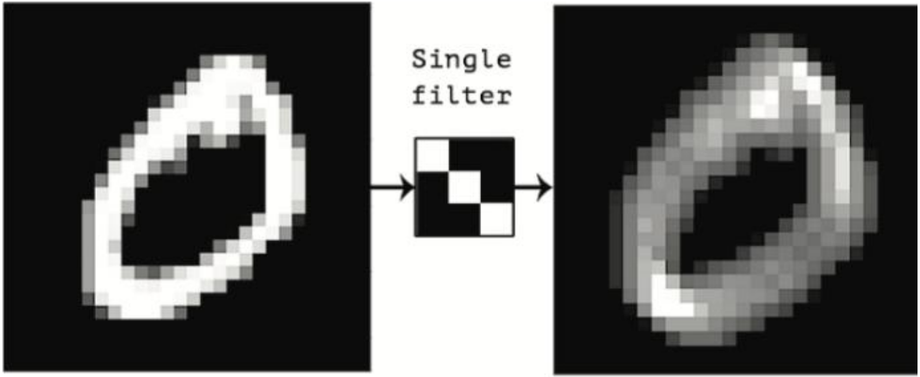


Figure: Convolving an image (left) with a 3x3 filter detecting diagonal lines (middle) produces a feature map (right) highlighting those features.

Convolution as Matrix Multiplication

- Convolution is a linear operation.

Convolution as Matrix Multiplication

- Convolution is a linear operation.
- Can be represented by multiplying a flattened input vector $\mathbf{x}$ by a specially structured matrix $\mathbf{C}$ derived from the filter $\mathbf{W}$.

Convolution as Matrix Multiplication

- Convolution is a linear operation.
- Can be represented by multiplying a flattened input vector $\mathbf{x}$ by a specially structured matrix $\mathbf{C}$ derived from the filter $\mathbf{W}$.
- $\mathbf{y} = \mathbf{C}\mathbf{x}$

Convolution as Matrix Multiplication

- Convolution is a linear operation.
- Can be represented by multiplying a flattened input vector $\mathbf{x}$ by a specially structured matrix $\mathbf{C}$ derived from the filter $\mathbf{W}$.
- $\mathbf{y} = \mathbf{C}\mathbf{x}$
- This matrix $\mathbf{C}$ is sparse and has repeated elements (tied weights).

Convolution as Matrix Multiplication

- Convolution is a linear operation.
- Can be represented by multiplying a flattened input vector $\mathbf{x}$ by a specially structured matrix $\mathbf{C}$ derived from the filter $\mathbf{W}$.
- $\mathbf{y} = \mathbf{C}\mathbf{x}$
- This matrix $\mathbf{C}$ is sparse and has repeated elements (tied weights).
- Shows CNNs are like MLPs with structured, sparse, weight-tied matrices.

Convolution as Matrix Multiplication

- Convolution is a linear operation.
- Can be represented by multiplying a flattened input vector $\mathbf{x}$ by a specially structured matrix $\mathbf{C}$ derived from the filter $\mathbf{W}$.
- $\mathbf{y} = \mathbf{C}\mathbf{x}$
- This matrix $\mathbf{C}$ is sparse and has repeated elements (tied weights).
- Shows CNNs are like MLPs with structured, sparse, weight-tied matrices.
- Achieves translation invariance and parameter reduction.

Boundary Conditions: Padding

- Problem: Convolution reduces output size. Convolving $f^h \times f^w$ filter on $x^h \times x^w$ image yields $(x^h - f^h + 1) \times (x^w - f^w + 1)$ output ('valid' convolution).

Boundary Conditions: Padding

- Problem: Convolution reduces output size. Convolving $f^h \times f^w$ filter on $x^h \times x^w$ image yields $(x^h - f^h + 1) \times (x^w - f^w + 1)$ output ('valid' convolution).
- Solution: Zero-padding adds a border of 0s around the input image.

Boundary Conditions: Padding

- Problem: Convolution reduces output size. Convolving $f^h \times f^w$ filter on $x^h \times x^w$ image yields $(x^h - f^h + 1) \times (x^w - f^w + 1)$ output ('valid' convolution).
- Solution: Zero-padding adds a border of 0s around the input image.
- **Same Convolution:** Choose padding p such that output size matches input size.

Boundary Conditions: Padding

- Problem: Convolution reduces output size. Convolving $f^h \times f^w$ filter on $x^h \times x^w$ image yields $(x^h - f^h + 1) \times (x^w - f^w + 1)$ output ('valid' convolution).
- Solution: Zero-padding adds a border of 0s around the input image.
- **Same Convolution:** Choose padding p such that output size matches input size.
  - Typically $p = (f - 1)/2$ for odd filter sizes.

Boundary Conditions: Padding

- Problem: Convolution reduces output size. Convolving $f^h \times f^w$ filter on $x^h \times x^w$ image yields $(x^h - f^h + 1) \times (x^w - f^w + 1)$ output ('valid' convolution).
- Solution: Zero-padding adds a border of 0s around the input image.
- **Same Convolution:** Choose padding p such that output size matches input size.
  - Typically $p = (f - 1)/2$ for odd filter sizes.
  - Output size with padding p^h, p^w: $(x^h + 2p^h - f^h + 1) \times (x^w + 2p^w - f^w + 1)$.

Padding Example: Same Convolution

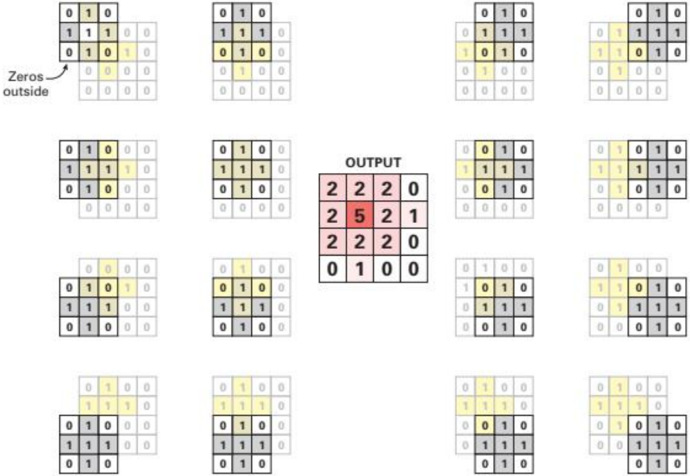


Figure: 'Same' convolution uses zero-padding to keep output size equal to input size.

Strided Convolution

- Problem: Neighboring outputs in feature maps are often redundant due to overlapping input patches.
- Solution: **Strided Convolution** skips inputs by a step size (stride) s.
- Reduces output size and computation.
- Output size with stride s^h, s^w and padding p^h, p^w:

$$\lfloor \frac{x^h + 2p^h - f^h + s^h}{s^h} \rfloor \times \lfloor \frac{x^w + 2p^w - f^w + s^w}{s^w} \rfloor$$

Padding and Stride Example

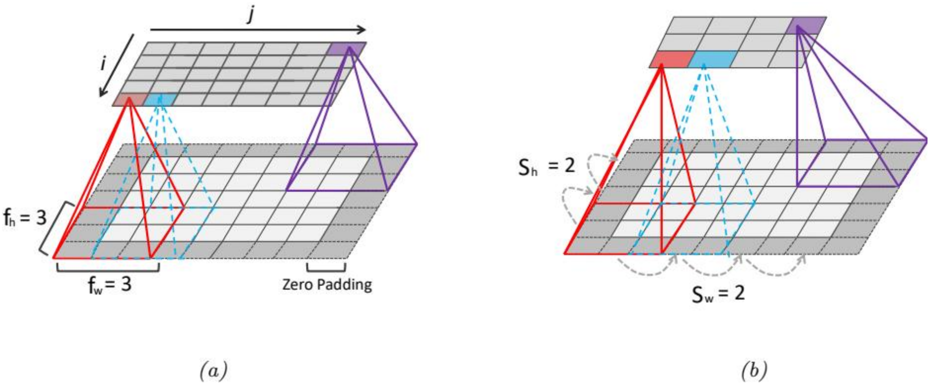


Figure: (a) 'Same' convolution (padding=1, stride=1) on 5x7 input with 3x3 filter gives 5x7 output. (b) Stride=2 gives 3x4 output.

Multiple Input Channels

- Real images often have multiple channels (e.g., RGB, $C = 3$).

Multiple Input Channels

- Real images often have multiple channels (e.g., RGB, $C = 3$).
- Filter $\mathbf{W}$ becomes 3D: $H \times W \times C$.

Multiple Input Channels

- Real images often have multiple channels (e.g., RGB, $C = 3$).
- Filter $\mathbf{W}$ becomes 3D: $H \times W \times C$.
- Each input channel c is convolved with its corresponding filter slice $\mathbf{W}_{:,:,c}$.

Multiple Input Channels

- Real images often have multiple channels (e.g., RGB, $C = 3$).
- Filter $\mathbf{W}$ becomes 3D: $H \times W \times C$.
- Each input channel c is convolved with its corresponding filter slice $\mathbf{W}_{:,:,c}$.
- Results are summed across channels (plus bias b) to produce a single output channel:

$$z_{i,j} = b + \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \sum_{c=0}^{C-1} x_{si+u,sj+v,c} w_{u,v,c}$$

Multiple Input Channels Visualization

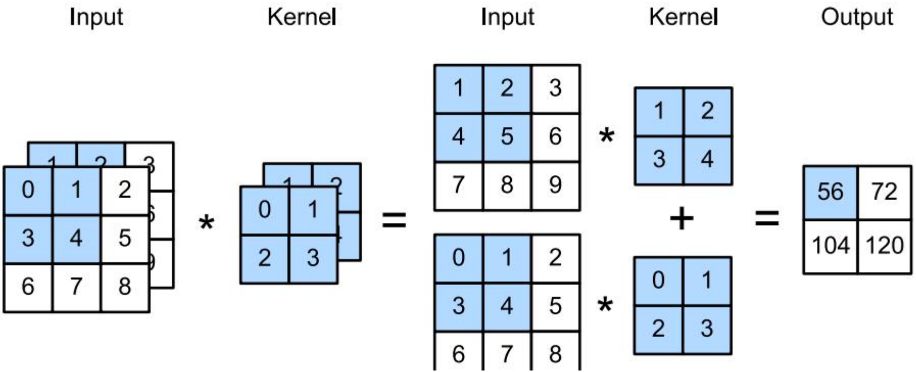


Figure: 2D convolution with a 2-channel input. Each input channel is convolved with a corresponding 2D filter slice, and the results are summed.

Multiple Output Channels

- Goal: Detect multiple types of features at each location.
- Use multiple filters, one for each desired output feature map d.
- Filter $\mathbf{W}$ becomes 4D: $H \times W \times C \times D$.
- $\mathbf{W}_{:,:,c,d}$ is the 2D filter for output channel d and input channel c.
- Output $z_{i,j,d}$ for feature map d is computed by summing convolutions across all input channels C:

$$z_{i,j,d} = b_d + \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \sum_{c=0}^{C-1} x_{si+u,sj+v,c} w_{u,v,c,d}$$

Multiple Input/Output Channels Visualization

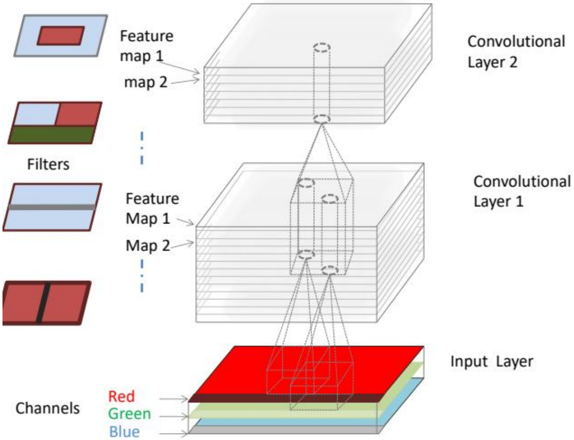


Figure: CNN with multiple channels. Input (3 channels) -> Conv Layer 1 (multiple channels) -> Conv Layer 2 (more channels). Cylinders represent feature vectors (hypercolumns) at specific locations.

1x1 Convolution (Pointwise Convolution)

- A special case with filter size 1×1.
- Computes a weighted combination of input channels *at the same location*.

$$z_{i,j,d} = b_d + \sum_{c=0}^{C-1} x_{i,j,c} w_{0,0,c,d}$$

- Changes the number of channels ($C \to D$) without changing spatial dimensions (H, W).
- Equivalent to applying a small MLP (Dense layer) independently to each spatial location's feature vector.
- Used in modern architectures (e.g., bottleneck layers, network-in-network).

1x1 Convolution Visualization

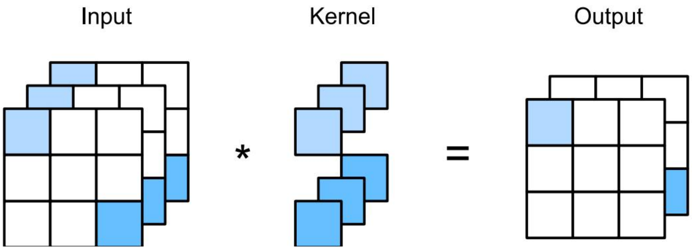


Figure: Mapping 3 input channels to 2 output channels using 1x1 convolution (filter size 1x1x3x2).

Pooling Layers: Motivation

- Convolution is *equivariant* (feature location changes if input shifts).

Pooling Layers: Motivation

- Convolution is *equivariant* (feature location changes if input shifts).
- Often desire *invariance* (output doesn't change if input shifts slightly).

Pooling Layers: Motivation

- Convolution is *equivariant* (feature location changes if input shifts).
- Often desire *invariance* (output doesn't change if input shifts slightly).
- Example: Image classification - presence of an object matters more than exact location.

Pooling Layers: Motivation

- Convolution is *equivariant* (feature location changes if input shifts).
- Often desire *invariance* (output doesn't change if input shifts slightly).
- Example: Image classification - presence of an object matters more than exact location.
- Pooling layers reduce spatial resolution and introduce local invariance.

Pooling Layers: Max Pooling

- Most common pooling method.

Pooling Layers: Max Pooling

- Most common pooling method.
- Slides a window over the feature map.

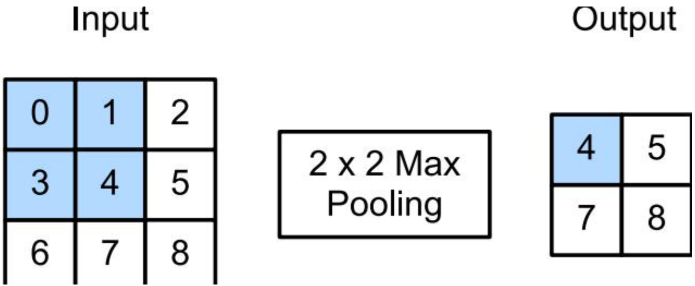


Figure: Max pooling with a 2x2 filter and stride 2.

Pooling Layers: Max Pooling

- Most common pooling method.
- Slides a window over the feature map.
- Outputs the *maximum* value within each window.

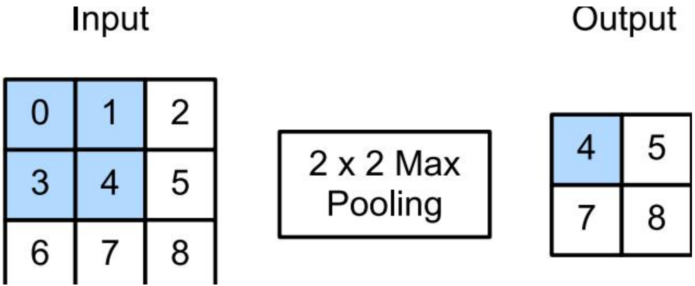


Figure: Max pooling with a 2x2 filter and stride 2.

Pooling Layers: Max Pooling

- Most common pooling method.
- Slides a window over the feature map.
- Outputs the *maximum* value within each window.
- Typically uses a small window (e.g., 2x2) and stride (e.g., 2).

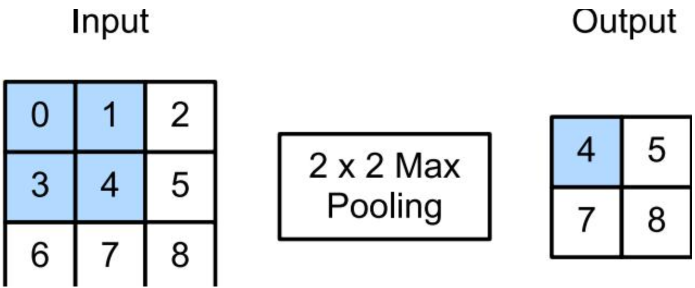


Figure: Max pooling with a 2x2 filter and stride 2.

Pooling Layers: Max Pooling

- Most common pooling method.
- Slides a window over the feature map.
- Outputs the *maximum* value within each window.
- Typically uses a small window (e.g., 2x2) and stride (e.g., 2).
- Reduces dimensionality, introduces robustness to small translations.

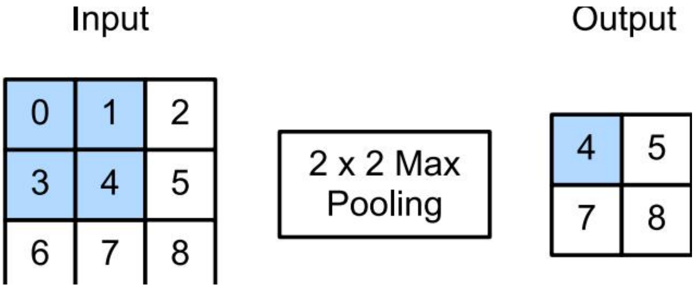


Figure: Max pooling with a 2x2 filter and stride 2.

Pooling Layers: Max Pooling

- Most common pooling method.
- Slides a window over the feature map.
- Outputs the *maximum* value within each window.
- Typically uses a small window (e.g., 2x2) and stride (e.g., 2).
- Reduces dimensionality, introduces robustness to small translations.
- Applied independently to each channel.

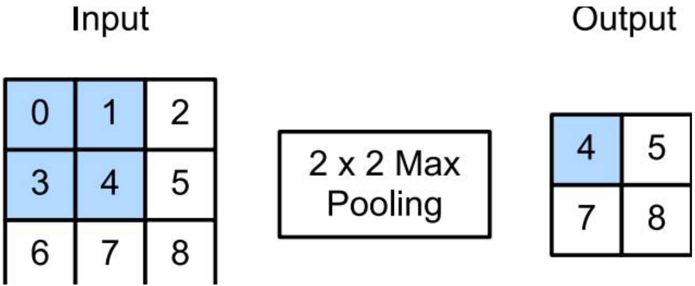


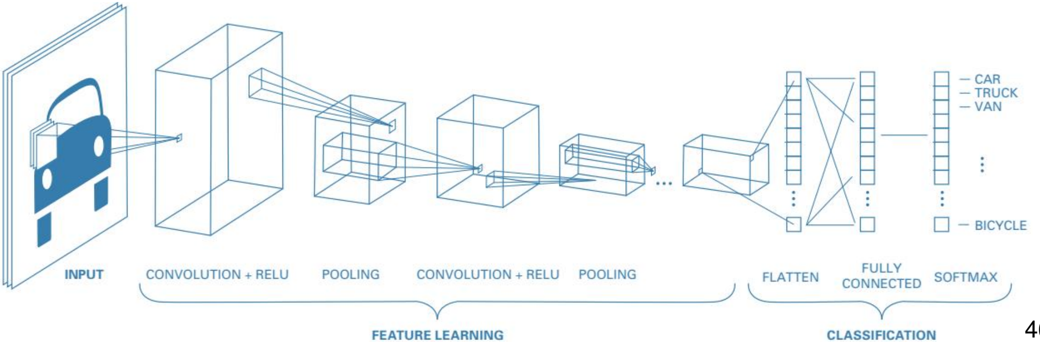
Figure: Max pooling with a 2x2 filter and stride 2.

Pooling Layers: Average Pooling & Global Average Pooling

- **Average Pooling:** Computes the *average* value within the window instead of the max.
- **Global Average Pooling (GAP):** Averages over the *entire* spatial dimension of a feature map.
  - Converts an $H \times W \times D$ feature map to a $1 \times 1 \times D$ (or D-dimensional) vector.
  - Often used before the final classification layer.
  - Allows the network to handle variable input image sizes.

Putting It Together: Simple CNN Architecture

- Common pattern: [CONV -> ReLU -> POOL] x N -> FLATTEN -> DENSE -> SOFTMAX
- Convolutional layers extract features.
- Pooling layers reduce dimensionality and add invariance.
- Final dense layers perform classification based on high-level features.



Historical Context: LeNet

- Early successful CNN architecture by Yann LeCun et al. (1998) [LeC+98].
- Designed for digit recognition (MNIST).
- Similar pattern: CONV -> POOL -> CONV -> POOL -> DENSE -> DENSE -> OUTPUT.
- Used backpropagation and SGD for training.
- Inspired by earlier Neocognitron [Fuk75] and biological vision models [HW62].

Normalization Layers: Why?

- Training deep networks is hard (Vanishing/Exploding Gradients - Ch 13).
- Normalization layers help stabilize training.
- Idea: Standardize the statistics (mean, variance) of activations within layers.
- Analogy: Standardizing input features.

Batch Normalization (BN) [IS15]

- Most popular normalization technique.

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.
- For each activation z_n:

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.
- For each activation z_n:
  1. Calculate mini-batch mean $\mu_{\mathcal{B}}$ and variance $\sigma^2_{\mathcal{B}}$.

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.
- For each activation z_n:
  1. Calculate mini-batch mean $\mu_{\mathcal{B}}$ and variance $\sigma^2_{\mathcal{B}}$.
  2. Normalize: $\hat{z}_n = (z_n - \mu_{\mathcal{B}})/\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}$

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.
- For each activation z_n:
  1. Calculate mini-batch mean $\mu_{\mathcal{B}}$ and variance $\sigma^2_{\mathcal{B}}$.
  2. Normalize: $\hat{z}_n = (z_n - \mu_{\mathcal{B}})/\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}$
  3. Scale and Shift: $\tilde{z}_n = \gamma \odot \hat{z}_n + \beta$ (γ, β are learnable parameters).

Batch Normalization (BN) [IS15]

- Most popular normalization technique.
- Normalizes activations within a mini-batch $\mathcal{B}$.
- For each activation z_n:
  1. Calculate mini-batch mean $\mu_{\mathcal{B}}$ and variance $\sigma^2_{\mathcal{B}}$.
  2. Normalize: $\hat{z}_n = (z_n - \mu_{\mathcal{B}}) / \sqrt{\sigma^2_{\mathcal{B}} + \epsilon}$
  3. Scale and Shift: $\tilde{z}_n = \gamma \odot \hat{z}_n + \beta$ (γ, β are learnable parameters).
- Applied after CONV/DENSE layers, often before activation function.

Batch Normalization: Training vs. Test Time

- **Training:** Use mini-batch statistics ($\mu_{\mathcal{B}}, \sigma^2_{\mathcal{B}}$). Learn γ, β.
- **Testing:** Mini-batch statistics are unreliable (batch size might be 1).
  - Use population statistics (mean μ, variance σ^2) estimated from the entire training set (often using moving averages during training).
  - Freeze $\mu, \sigma^2, \gamma, \beta$.
  - The BN layer becomes a simple linear transform.
- BN layer behaves differently during training and inference.

Benefits of Batch Normalization

- Speeds up training significantly.

Benefits of Batch Normalization

- Speeds up training significantly.
- Stabilizes training, allows higher learning rates.

Benefits of Batch Normalization

- Speeds up training significantly.
- Stabilizes training, allows higher learning rates.
- Reduces sensitivity to initialization.

Benefits of Batch Normalization

- Speeds up training significantly.
- Stabilizes training, allows higher learning rates.
- Reduces sensitivity to initialization.
- Acts as a regularizer, sometimes reducing need for Dropout.

Benefits of Batch Normalization

- Speeds up training significantly.
- Stabilizes training, allows higher learning rates.
- Reduces sensitivity to initialization.
- Acts as a regularizer, sometimes reducing need for Dropout.
- Smoother optimization landscape [San+18b].

Benefits of Batch Normalization

- Speeds up training significantly.
- Stabilizes training, allows higher learning rates.
- Reduces sensitivity to initialization.
- Acts as a regularizer, sometimes reducing need for Dropout.
- Smoother optimization landscape [San+18b].
- Mechanism still debated ("Internal Covariate Shift" is likely not the full story).

Conclusion & Next Steps (CNNs)

- CNNs are essential for image data due to convolution's properties (parameter sharing, translation invariance).
- Key Layers: Convolution, Pooling, Normalization (esp. Batch Norm).
- Standard architectures combine these layers effectively.
- Modern CNNs (ResNet, EfficientNet) use advanced techniques but follow these core principles.

Table of Contents

Introduction to Sequence Modeling

- **The Data:** Sequential data $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$.

Introduction to Sequence Modeling

- **The Data:** Sequential data $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$.
- **Examples:**
  - Text: "The" $\rightarrow$ "cat" $\rightarrow$ "sat".
  - Time Series: Stock prices over days.
  - Audio: Waveform samples over time.

Introduction to Sequence Modeling

- **The Data:** Sequential data $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$.
- **Examples:**
  - Text: "The" $\rightarrow$ "cat" $\rightarrow$ "sat".
  - Time Series: Stock prices over days.
  - Audio: Waveform samples over time.
- **The Challenge:**
  - Variable length T.
  - Long-term dependencies (e.g., "The **boy** who wore a red hat ... **was** happy").

Introduction to Sequence Modeling

- **The Data:** Sequential data $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$.
- **Examples:**
  - Text: "The" $\rightarrow$ "cat" $\rightarrow$ "sat".
  - Time Series: Stock prices over days.
  - Audio: Waveform samples over time.
- **The Challenge:**
  - Variable length T.
  - Long-term dependencies (e.g., "The **boy** who wore a red hat ... **was** happy").
- **Standard MLPs fail** because they expect fixed-size input and treat features as independent.

Recurrent Neural Networks (RNNs): The Core Idea

- **Core Concept:** Process the sequence one step at a time, maintaining an internal "memory" or **Hidden State** $(\mathbf{h}_t)$.

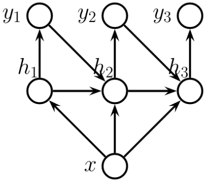


Figure: The hidden state $\mathbf{h}$ passes information forward through time.

Recurrent Neural Networks (RNNs): The Core Idea

- **Core Concept:** Process the sequence one step at a time, maintaining an internal "memory" or **Hidden State** ($\mathbf{h}_t$).
- **Recurrence Relation:**

$$\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

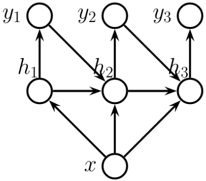


Figure: The hidden state $\mathbf{h}$ passes information forward through time.

Recurrent Neural Networks (RNNs): The Core Idea

- **Core Concept:** Process the sequence one step at a time, maintaining an internal "memory" or **Hidden State** ($\mathbf{h}_t$).
- **Recurrence Relation:**

$$\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

- **Step-by-Step Update:**
  1. **Input:** Current token $\mathbf{x}_t$.
  2. **Context:** Previous state $\mathbf{h}_{t-1}$ (summary of the past).
  3. **Update:** Compute new state $\mathbf{h}_t$ using shared weights $\mathbf{W}$.
  4. **Output:** Compute prediction $\mathbf{y}_t$ from $\mathbf{h}_t$.

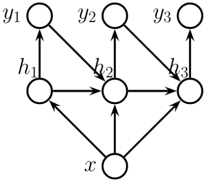


Figure: The hidden state $\mathbf{h}$ passes information forward through time.

RNN: Mathematical Formulation

- A vanilla RNN typically uses the $\tanh$ activation function:

$$
\begin{aligned}
\mathbf{h}_t &= \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \\
\mathbf{y}_t &= \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y
\end{aligned}
$$

RNN: Mathematical Formulation

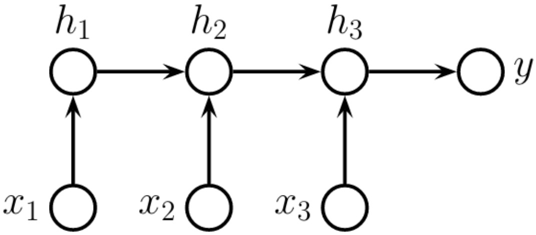
- A vanilla RNN typically uses the $\tanh$ activation function:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

- **Parameters (Shared across all time steps):**
  - $\mathbf{W}_{xh}$: Weights mapping input to hidden.
  - $\mathbf{W}_{hh}$: Weights mapping hidden to hidden (recurrence).
  - $\mathbf{W}_{hy}$: Weights mapping hidden to output.

Architecture 1: Many-to-One (Sequence Classification)

- **Task:** Sentiment Analysis, Intent Classification.



(a)

Figure: Basic RNN for sequence classification where only the final output is used.

Architecture 1: Many-to-One (Sequence Classification)

- **Task:** Sentiment Analysis, Intent Classification.
- **Process:**
  - Read entire sequence $\mathbf{x}_1, \ldots, \mathbf{x}_T$.
  - Update state: $\mathbf{h}_0 \to \mathbf{h}_1 \to \cdots \to \mathbf{h}_T$.
  - Use **final state** $\mathbf{h}_T$ to predict label y.

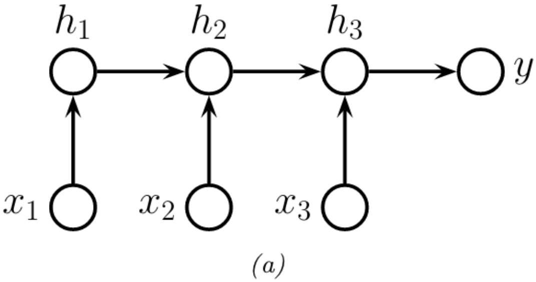


Figure: Basic RNN for sequence classification where only the final output is used.

Architecture 1: Many-to-One (Sequence Classification)

- **Task:** Sentiment Analysis, Intent Classification.
- **Process:**
  - Read entire sequence $\mathbf{x}_1, \ldots, \mathbf{x}_T$.
  - Update state: $\mathbf{h}_0 \to \mathbf{h}_1 \to \cdots \to \mathbf{h}_T$.
  - Use **final state** $\mathbf{h}_T$ to predict label y.
- **Intuition:** $\mathbf{h}_T$ is a vector summary of the whole sentence.

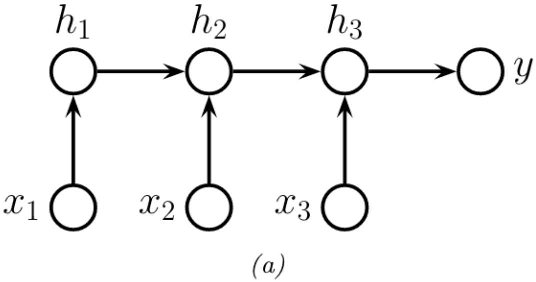


Figure: Basic RNN for sequence classification where only the final output is used.

Architecture 2: Many-to-Many (Seq2Seq)

- **Task:** Machine Translation, Summarization.

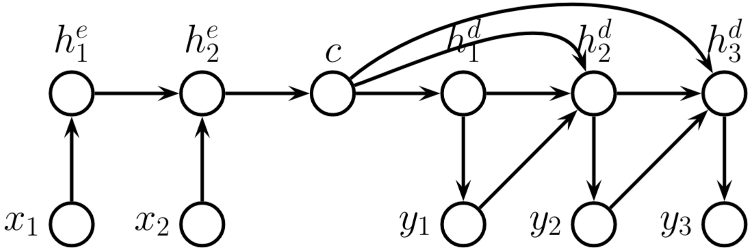


Figure: The context vector $\mathbf{c}$ is the bottleneck passing info from Encoder to Decoder.

Architecture 2: Many-to-Many (Seq2Seq)

- **Task:** Machine Translation, Summarization.
- **Encoder-Decoder Architecture:**
  1. **Encoder:** Process input $\mathbf{x}$ into context vector $\mathbf{c}$ (usually final state $\mathbf{h}_T$).
  2. **Decoder:** Generate output $\mathbf{y}$ one word at a time, conditioned on $\mathbf{c}$.

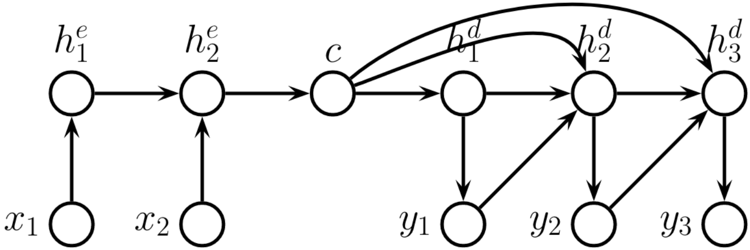


Figure: The context vector $\mathbf{c}$ is the bottleneck passing info from Encoder to Decoder.

Training: Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" it over time and treat it like a very deep MLP.

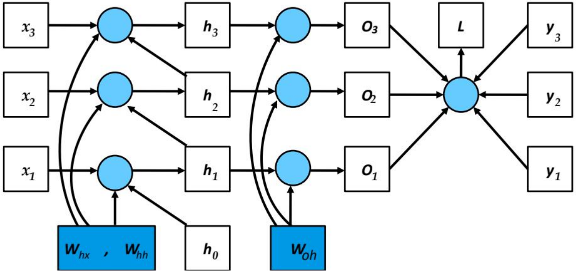


Figure: RNN unrolled for BPTT makes the dependency chain visible.

Training: Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" it over time and treat it like a very deep MLP.
- **The Chain Rule Problem:** Gradient of loss L_T w.r.t. initial weight $\mathbf{W}$ involves a product of partial derivatives:

$$\frac{\partial L_T}{\partial \mathbf{h}_1} = \frac{\partial L_T}{\partial \mathbf{h}_T} \cdot \prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$$

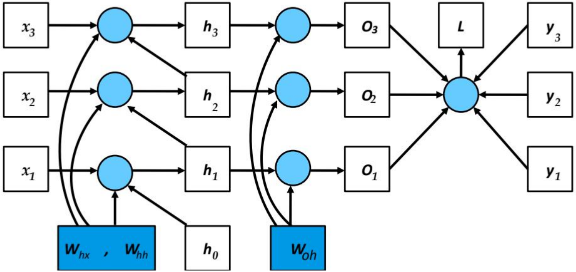


Figure: RNN unrolled for BPTT makes the dependency chain visible.

Training: Backpropagation Through Time (BPTT)

- To train an RNN, we "unroll" it over time and treat it like a very deep MLP.
- **The Chain Rule Problem:** Gradient of loss L_T w.r.t. initial weight $\mathbf{W}$ involves a product of partial derivatives:

$$\frac{\partial L_T}{\partial \mathbf{h}_1} = \frac{\partial L_T}{\partial \mathbf{h}_T} \cdot \prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$$

- Since $\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \dots)$, the term $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ depends on $\mathbf{W}$.

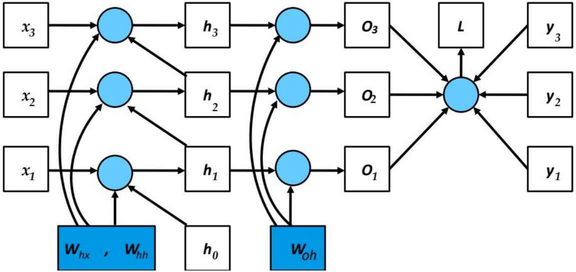


Figure: RNN unrolled for BPTT makes the dependency chain visible.

The Vanishing Gradient Problem

- Recall the chain rule product: $\prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$.

The Vanishing Gradient Problem

- Recall the chain rule product: $\prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$.
- If the dominant eigenvalue of $\mathbf{W}_{hh}$ is < 1, the gradient shrinks exponentially as T grows.

The Vanishing Gradient Problem

- Recall the chain rule product: $\prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$.
- If the dominant eigenvalue of $\mathbf{W}_{hh}$ is < 1, the gradient shrinks exponentially as T grows.
- **Consequence:** The model stops learning from early inputs. It "forgets" the beginning of the sentence by the time it reaches the end.

The Vanishing Gradient Problem

- Recall the chain rule product: $\prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$.
- If the dominant eigenvalue of $\mathbf{W}_{hh}$ is < 1, the gradient shrinks exponentially as T grows.
- **Consequence:** The model stops learning from early inputs. It "forgets" the beginning of the sentence by the time it reaches the end.
- **Solution:** Gated Architectures (LSTM, GRU).

Gated Recurrent Units (GRU): Intuition

- Standard RNN overwrites $\mathbf{h}_t$ at every step. GRUs typically decide *how much* to update.

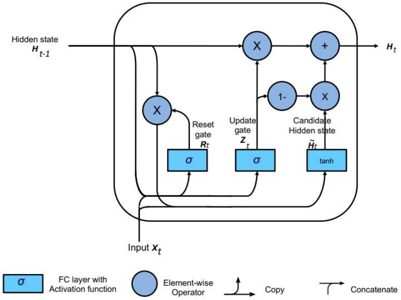


Figure: The GRU adds gates to control information flow.

Gated Recurrent Units (GRU): Intuition

- Standard RNN overwrites $\mathbf{h}_t$ at every step. GRUs typically decide *how much* to update.
- **Update Gate ($\mathbf{z}_t$):** "Should I copy the old state or write a new one?"

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

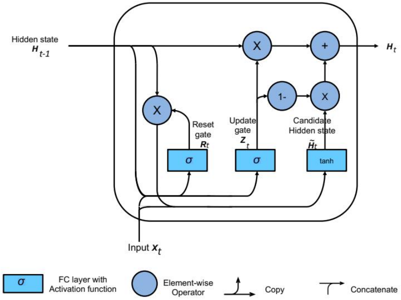


Figure: The GRU adds gates to control information flow.

Gated Recurrent Units (GRU): Intuition

- Standard RNN overwrites $\mathbf{h}_t$ at every step. GRUs typically decide *how much* to update.
- **Update Gate ($\mathbf{z}_t$):** "Should I copy the old state or write a new one?"

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

- If $\mathbf{z}_t \approx \mathbf{0}$, then $\mathbf{h}_t \approx \mathbf{h}_{t-1}$. The gradient passes through unchanged!

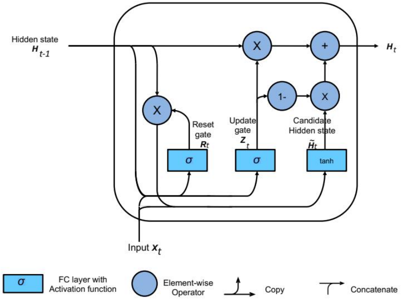


Figure: The GRU adds gates to control information flow.

Gated Recurrent Units (GRU): Intuition

- Standard RNN overwrites $\mathbf{h}_t$ at every step. GRUs typically decide *how much* to update.
- **Update Gate ($\mathbf{z}_t$):** "Should I copy the old state or write a new one?"

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

- If $\mathbf{z}_t \approx \mathbf{0}$, then $\mathbf{h}_t \approx \mathbf{h}_{t-1}$. The gradient passes through unchanged!
- This creates a "gradient superhighway" back through time, solving vanishing gradients.

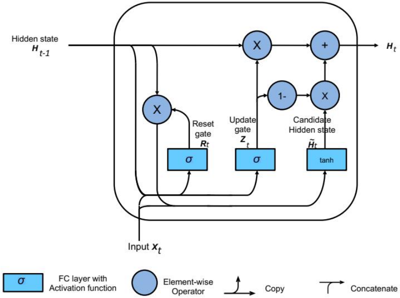


Figure: The GRU adds gates to control information flow.

The Attention Mechanism: Step-by-Step

Problem: Encoding a long sentence into a single vector $\mathbf{c}$ loses information. **Solution:** Let the decoder "look" at all encoder states $\mathbf{h}_1, \ldots, \mathbf{h}_T$ dynamically.

- **Step 1 (Score):** Compare current decoder state $\mathbf{s}_{t-1}$ with every encoder state $\mathbf{h}_i$.

$$\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i) = \mathbf{s}_{t-1}^\top \mathbf{h}_i$$

The Attention Mechanism: Step-by-Step

Problem: Encoding a long sentence into a single vector $\mathbf{c}$ loses information. **Solution:** Let the decoder "look" at all encoder states $\mathbf{h}_1, \ldots, \mathbf{h}_T$ dynamically.

- **Step 1 (Score):** Compare current decoder state $\mathbf{s}_{t-1}$ with every encoder state $\mathbf{h}_i$.

$$\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i) = \mathbf{s}_{t-1}^\top \mathbf{h}_i$$

- **Step 2 (Normalize):** Turn scores into probabilities (weights) using Softmax.

$$\alpha_{t,i} = \frac{\exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_j \exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))}$$

The Attention Mechanism: Step-by-Step

Problem: Encoding a long sentence into a single vector $\mathbf{c}$ loses information. **Solution:** Let the decoder "look" at all encoder states $\mathbf{h}_1, \ldots, \mathbf{h}_T$ dynamically.

- **Step 1 (Score):** Compare current decoder state $\mathbf{s}_{t-1}$ with every encoder state $\mathbf{h}_i$.

$$\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i) = \mathbf{s}_{t-1}^\top \mathbf{h}_i$$

- **Step 2 (Normalize):** Turn scores into probabilities (weights) using Softmax.

$$\alpha_{t,i} = \frac{\exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_j \exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))}$$

- **Step 3 (Context):** Compute weighted average of encoder states.

$$\mathbf{c}_t = \sum_i \alpha_{t,i} \mathbf{h}_i$$

Visualizing Attention

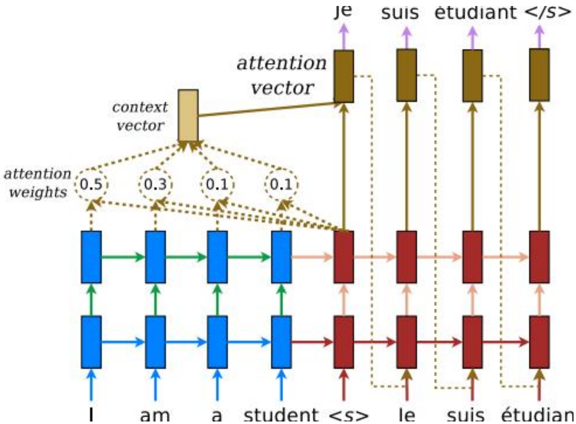


Figure: The decoder (blue) attends to relevant encoder states (red) to generate the next word.

- The model learns alignment automatically (e.g., aligning "European" with "Européenne").

The Transformer: Attention Is All You Need

- RNNs process sequentially $t = 1, 2, \ldots$ (Slow, hard to parallelize).

The Transformer: Attention Is All You Need

- RNNs process sequentially $t = 1, 2, \ldots$ (Slow, hard to parallelize).
- Transformers process the whole sequence **at once** using Self-Attention.

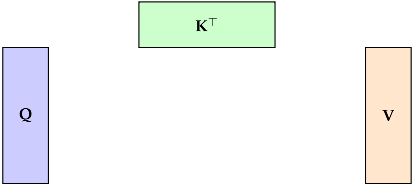
The Transformer: Attention Is All You Need

- RNNs process sequentially $t = 1, 2, \ldots$ (Slow, hard to parallelize).
- Transformers process the whole sequence **at once** using Self-Attention.
- **Self-Attention Analogy (Database Lookup):**
  - **Query ($\mathbf{Q}$):** What am I looking for?
  - **Key ($\mathbf{K}$):** What defines this item?
  - **Value ($\mathbf{V}$):** What is the content of this item?

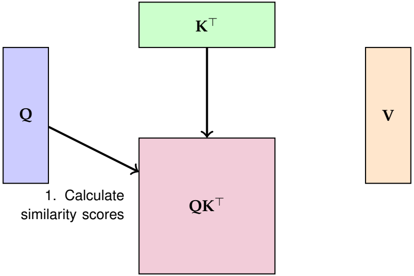
The Transformer: Attention Is All You Need

- RNNs process sequentially $t = 1, 2, \ldots$ (Slow, hard to parallelize).
- Transformers process the whole sequence **at once** using Self-Attention.
- **Self-Attention Analogy (Database Lookup):**
  - **Query ($\mathbf{Q}$):** What am I looking for?
  - **Key ($\mathbf{K}$):** What defines this item?
  - **Value ($\mathbf{V}$):** What is the content of this item?
- In Self-Attention, every word generates its own $\mathbf{Q}, \mathbf{K}$, and $\mathbf{V}$ vectors.

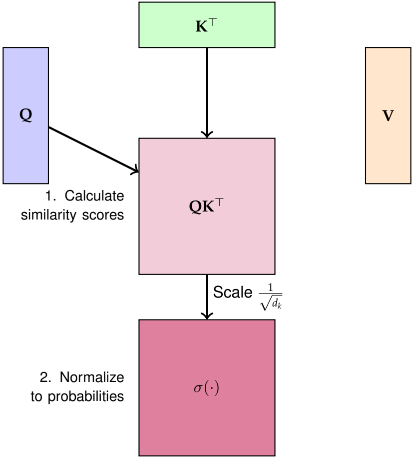
Step-by-Step: Self-Attention Mechanism



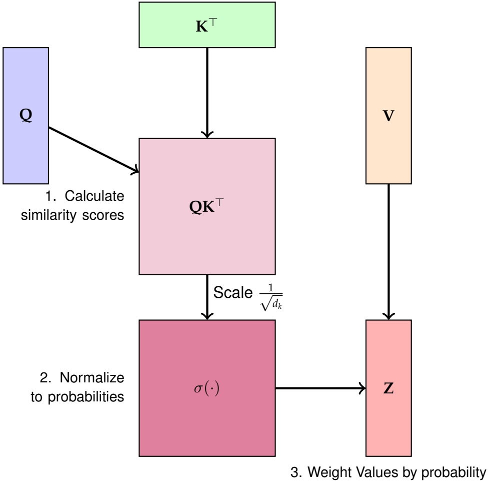
Step-by-Step: Self-Attention Mechanism



Step-by-Step: Self-Attention Mechanism



Step-by-Step: Self-Attention Mechanism



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Multi-Head Attention

- A single attention layer might focus only on syntax.

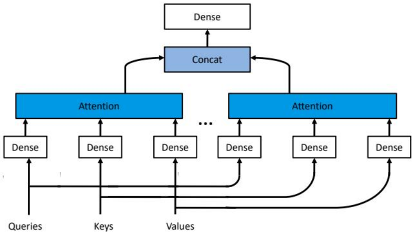


Figure: Multi-Head Attention schematic.

Multi-Head Attention

- A single attention layer might focus only on syntax.
- We want to attend to multiple aspects (syntax, semantics, relationships) simultaneously.

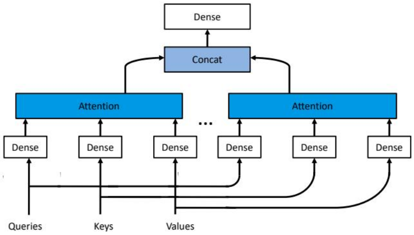


Figure: Multi-Head Attention schematic.

Multi-Head Attention

- A single attention layer might focus only on syntax.
- We want to attend to multiple aspects (syntax, semantics, relationships) simultaneously.
- **Multi-Head Attention:** Run h attention layers in parallel with different projection matrices.

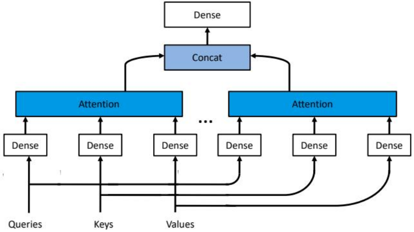


Figure: Multi-Head Attention schematic.

Multi-Head Attention

- A single attention layer might focus only on syntax.
- We want to attend to multiple aspects (syntax, semantics, relationships) simultaneously.
- **Multi-Head Attention:** Run h attention layers in parallel with different projection matrices.
- Concatenate the results:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O$$

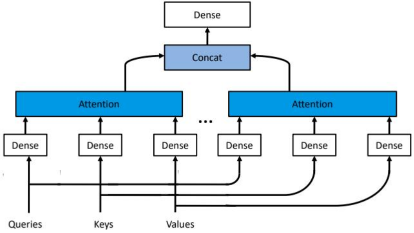
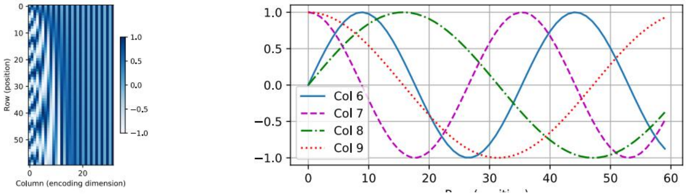


Figure: Multi-Head Attention schematic.

Positional Encoding: Adding Order

- **Problem:** Self-attention is permutation invariant. "The cat bit the dog" looks the same as "The dog bit the cat" to the math above.

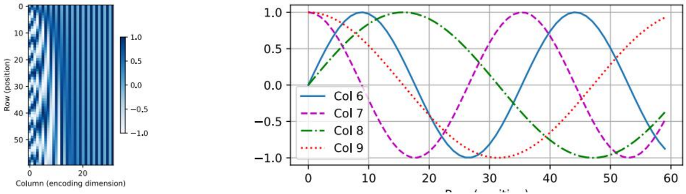


(a) (b)

Figure: Sinusoidal Positional Encodings.

Positional Encoding: Adding Order

- **Problem:** Self-attention is permutation invariant. "The cat bit the dog" looks the same as "The dog bit the cat" to the math above.
- **Solution:** Inject information about position.



(a) (b)

Figure: Sinusoidal Positional Encodings.

Positional Encoding: Adding Order

- **Problem:** Self-attention is permutation invariant. "The cat bit the dog" looks the same as "The dog bit the cat" to the math above.
- **Solution:** Inject information about position.
- Add a vector PE to the input embeddings:

$$\mathbf{x}_{\text{input}} = \mathbf{x}_{\text{word\_embedding}} + PE_{\text{position}}$$

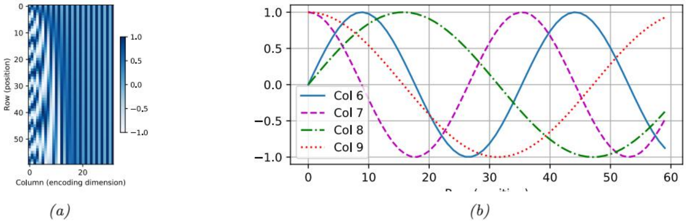


Figure: Sinusoidal Positional Encodings.

Positional Encoding: Adding Order

- **Problem:** Self-attention is permutation invariant. "The cat bit the dog" looks the same as "The dog bit the cat" to the math above.
- **Solution:** Inject information about position.
- Add a vector PE to the input embeddings:

$$\mathbf{x}_{\text{input}} = \mathbf{x}_{\text{word\_embedding}} + PE_{\text{position}}$$

- Transformer uses fixed Sinusoidal functions so the model can learn relative positions easily.

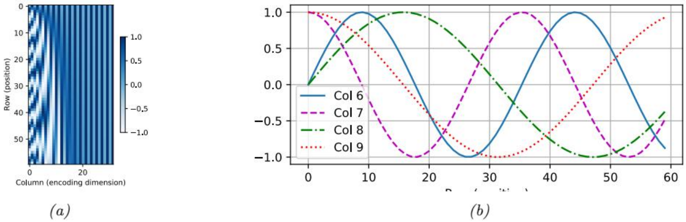


Figure: Sinusoidal Positional Encodings.

Summary: RNN vs Transformer

Feature	**RNN / LSTM**	**Transformer**
Processing	Sequential ($O(N)$)	Parallel ($O(1)$)
Long Distance	Hard (Vanishing Grad)	Easy (Direct Attention)
Complexity	$O(N)$	$O(N^2)$ (Heavy for long seq)
Inductive Bias	Recency	Global Interaction

- Transformers are now the state-of-the-art for NLP (BERT, GPT) and increasingly for Computer Vision (ViT).