

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

**Tao LIN**

SoE, Westlake University

December 9, 2025



① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

- The Basic Structure of MLP
- Training of NNs and BP

② Neural Networks

- Activation Function

③ Neural Networks for Images

④ Neural Networks for Sequences

# Table of Contents

## ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

- The Basic Structure of MLP
- Training of NNs and BP

## ② Neural Networks

## ③ Neural Networks for Images

## ④ Neural Networks for Sequences

# Table of Contents

## ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

- The Basic Structure of MLP
- Training of NNs and BP

## ② Neural Networks

- Activation Function

## ③ Neural Networks for Images

## ④ Neural Networks for Sequences

# Step-by-Step: MLP Forward Pass

Input

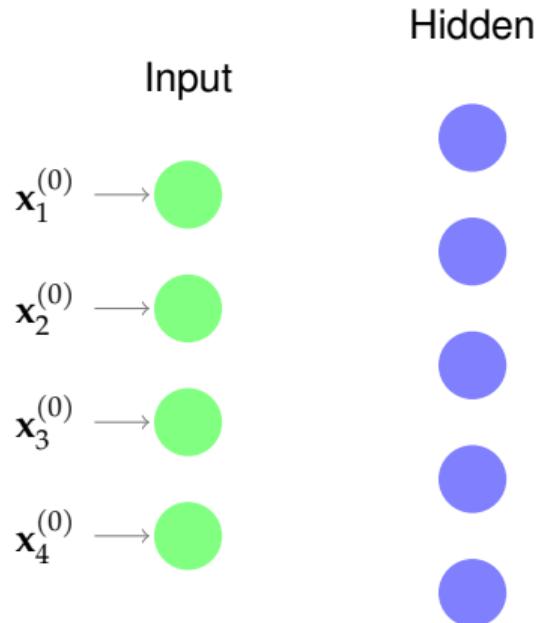
$$\mathbf{x}_1^{(0)} \longrightarrow \text{circle}$$

$$\mathbf{x}_2^{(0)} \longrightarrow \text{circle}$$

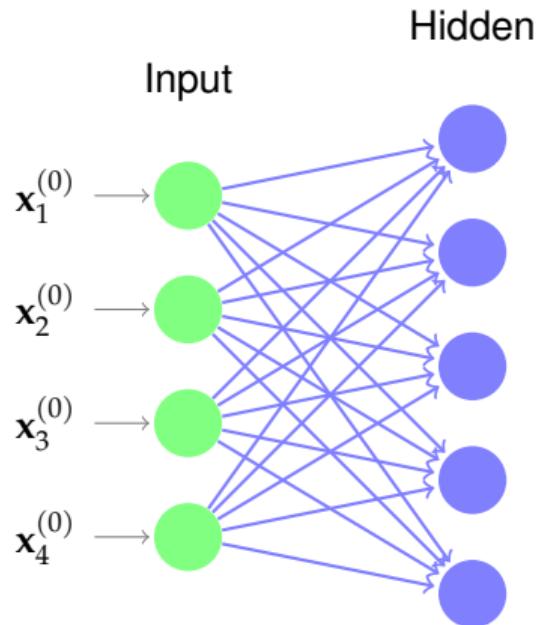
$$\mathbf{x}_3^{(0)} \longrightarrow \text{circle}$$

$$\mathbf{x}_4^{(0)} \longrightarrow \text{circle}$$

# Step-by-Step: MLP Forward Pass

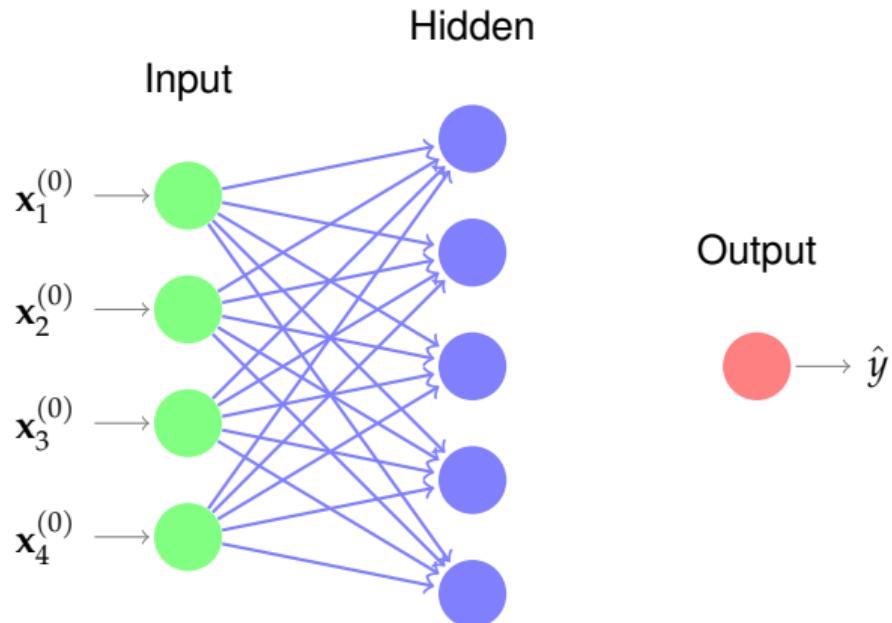


# Step-by-Step: MLP Forward Pass



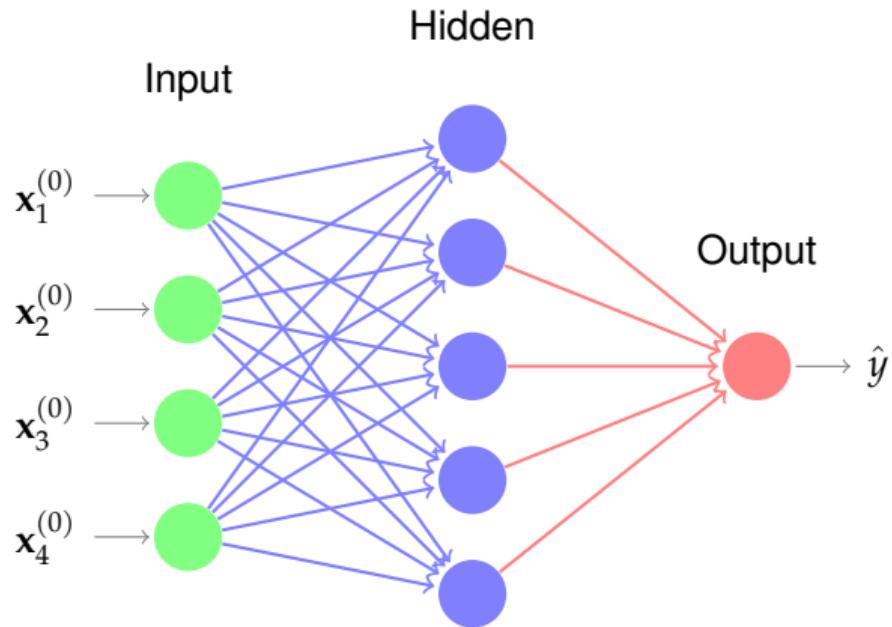
Compute:  $\mathbf{z}^{(1)} = (\mathbf{W}^{(1)})^\top \mathbf{x}^{(0)} + \mathbf{b}^{(1)}$

# 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



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

# 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  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\sigma$  we pick

# NNs extract suitable features from the input

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

- Feature extractor  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\sigma$  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  $\mathbb{R}^d \rightarrow \mathbb{R}^K$ : 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  $\sigma$  we pick

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

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

# Table of Contents

## ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

- The Basic Structure of MLP
- Training of NNs and BP

## ② Neural Networks

- Activation Function

## ③ Neural Networks for Images

## ④ Neural Networks for Sequences

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

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

where

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

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

where

- $f$  is the function represented by a NN.

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

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

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

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

# Compact description of output

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

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

- 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

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

- 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

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

- 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

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

## Compact description of output

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

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

- 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

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

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:

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

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

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

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

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

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

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  $\delta^{(l)}$ ) to aid the computation:

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

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

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**:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \quad (5)$$

$$= \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \quad (6)$$

$$= \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}) , \quad (7)$$

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

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

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

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**:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \quad (5)$$

$$= \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \quad (6)$$

$$= \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}) , \quad (7)$$

In vector form, we can write this as

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

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

Now that we have both  $\mathbf{z}^{(l)}$  and  $\delta^{(l)}$  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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , 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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , 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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , are fixed.
- We are given in addition a sample  $(\mathbf{x}_n, y_n)$ .
- We want to compute the derivatives

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

where

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

## 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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , are fixed.
- We are given in addition a sample  $(\mathbf{x}_n, y_n)$ .
- We want to compute the derivatives

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

where

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

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

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

## 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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , are fixed.
- We are given in addition a sample  $(\mathbf{x}_n, y_n)$ .
- We want to compute the derivatives

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

where

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

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

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

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

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

## 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  $\mathbf{b}^{(l)}$ ,  $l = 1, \dots, L + 1$ , are fixed.
- We are given in addition a sample  $(\mathbf{x}_n, y_n)$ .
- We want to compute the derivatives

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

where

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

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

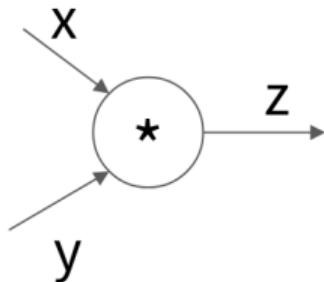
$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

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

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

# Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code



(x,y,z are scalars)

```
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

① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

② Neural Networks

- Activation Function

③ Neural Networks for Images

④ Neural Networks for Sequences

# Table of Contents

## ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)

- The Basic Structure of MLP
- Training of NNs and BP

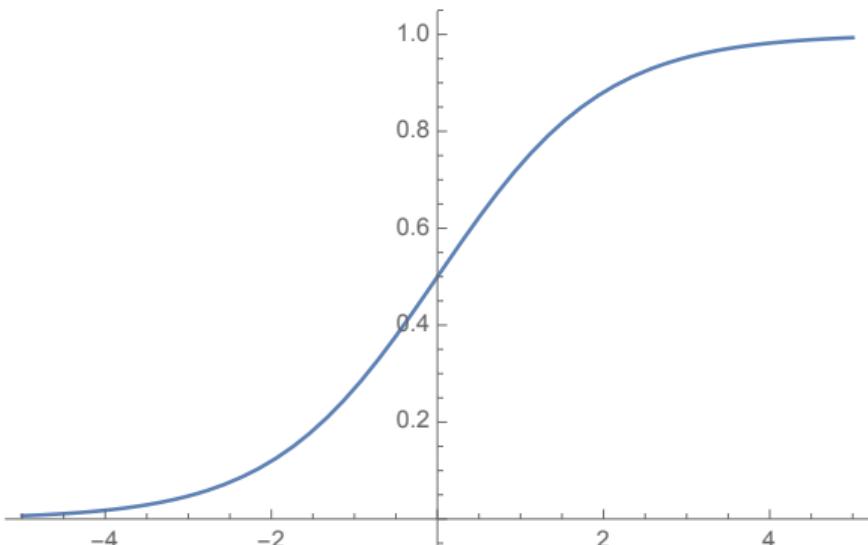
## ② Neural Networks

- Activation Function

## ③ Neural Networks for Images

## ④ Neural Networks for Sequences

# The sigmoid



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

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

- 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

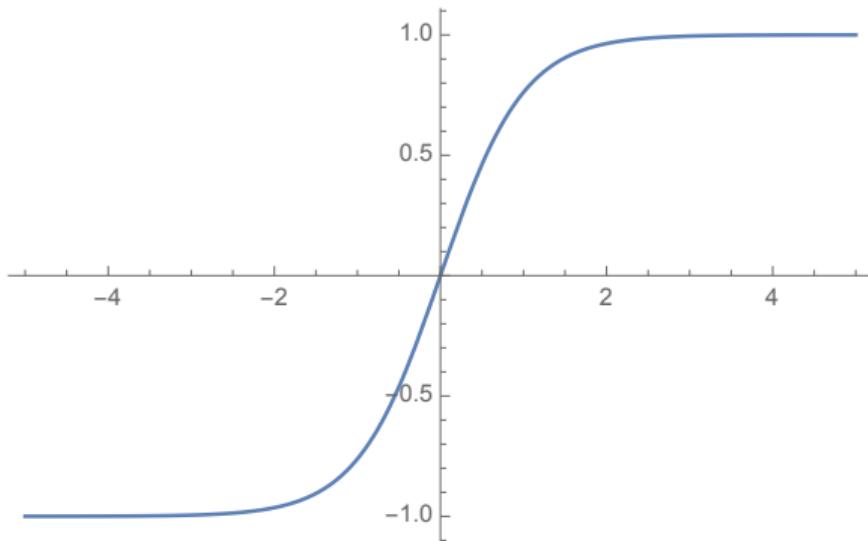
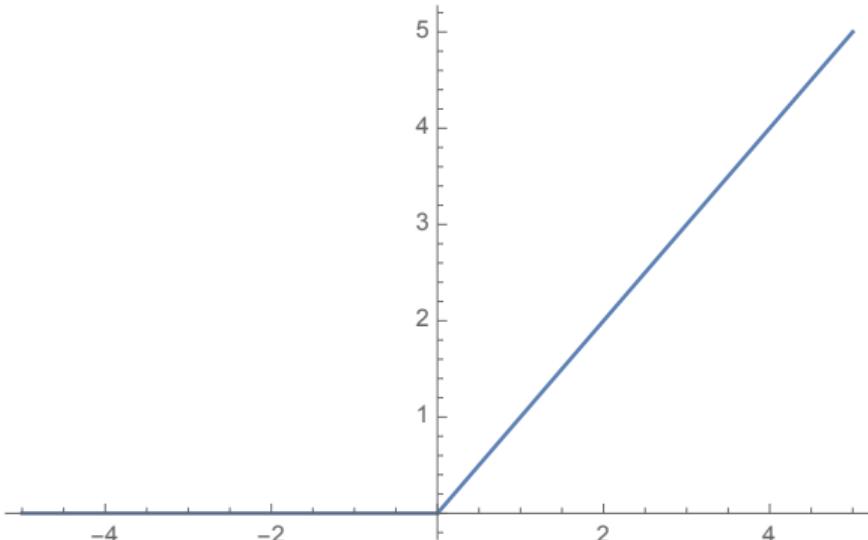


Figure:  $\tanh(x)$ .

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

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

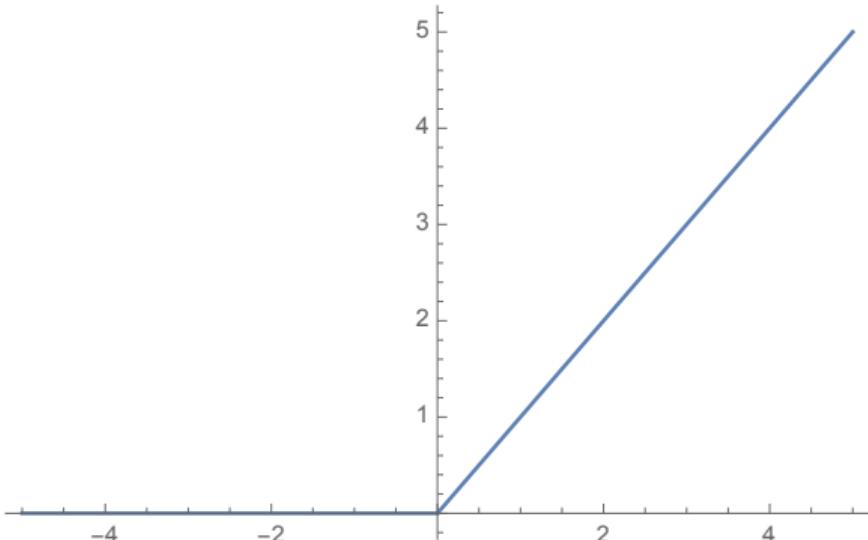
# Rectified linear Unit – ReLU



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

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

# Rectified linear Unit – ReLU

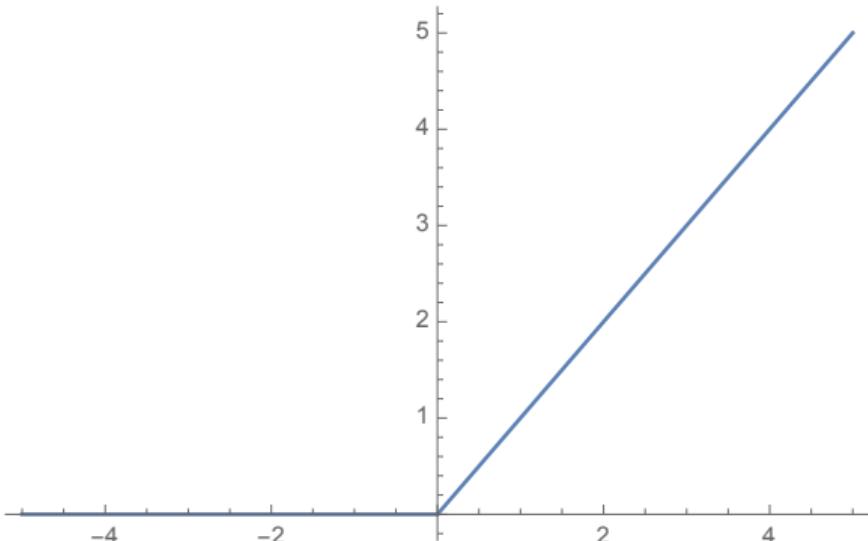


$$(x)_+ = \max\{0, x\}, \quad (11)$$

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

- ReLU is always positive and is unbounded.

# Rectified linear Unit – ReLU

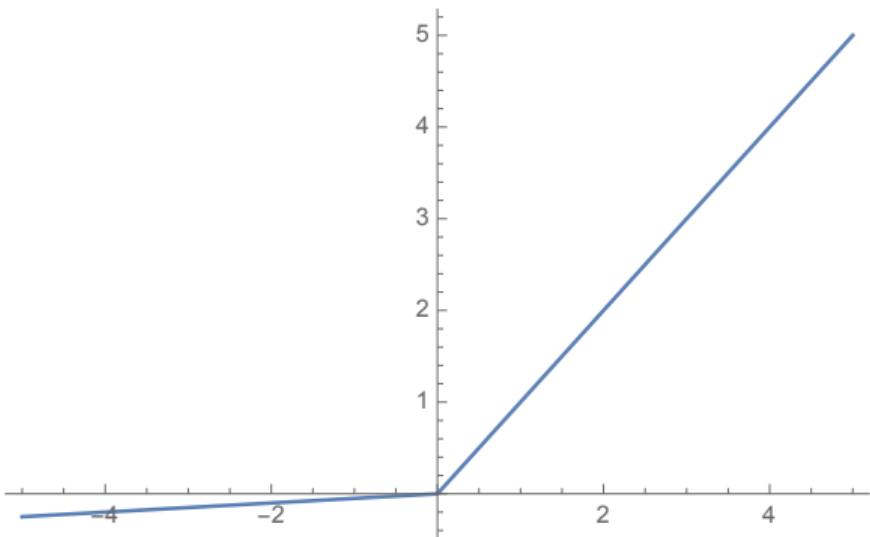


$$(x)_+ = \max\{0, x\}, \quad (11)$$

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

- 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

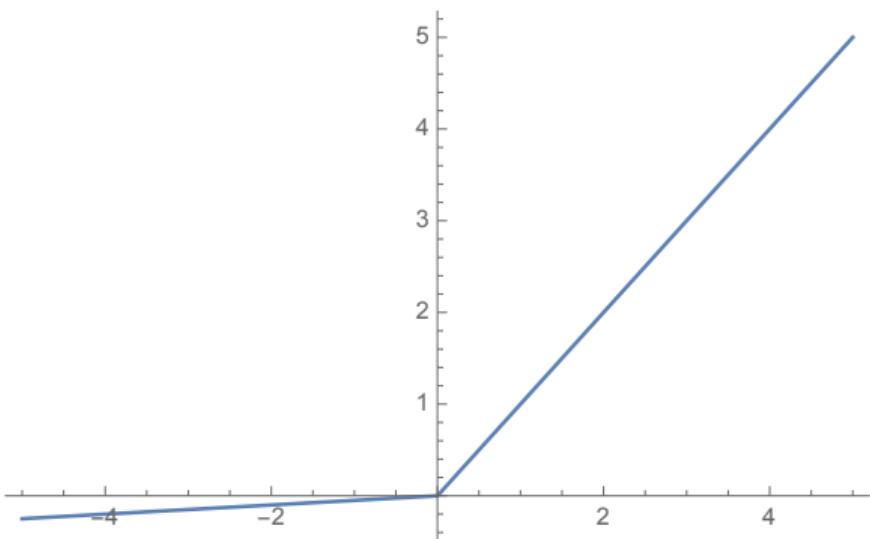


In order to solve the 0-derivative problem of the ReLU (for negative values of  $x$ ) one can add a very small slope  $\alpha$  in the negative part.

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

Figure: LReLU with  $\alpha = 0.05$

# Leaky ReLU



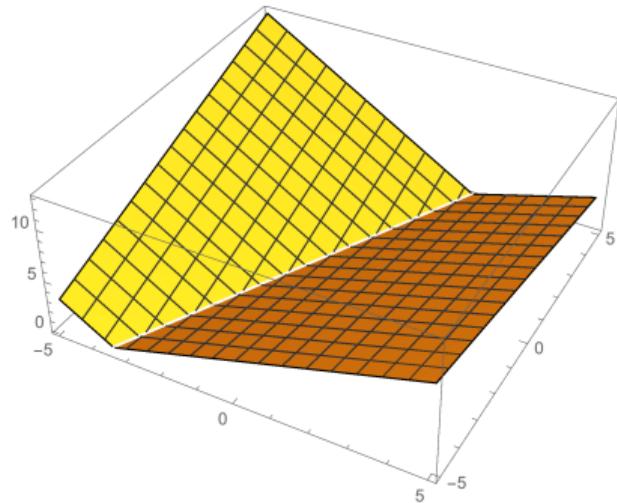
In order to solve the 0-derivative problem of the ReLU (for negative values of  $x$ ) one can add a very small slope  $\alpha$  in the negative part.

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

Figure: LReLU with  $\alpha = 0.05$

- The constant  $\alpha$  is of course a hyper-parameter that can be optimized.

# Maxout

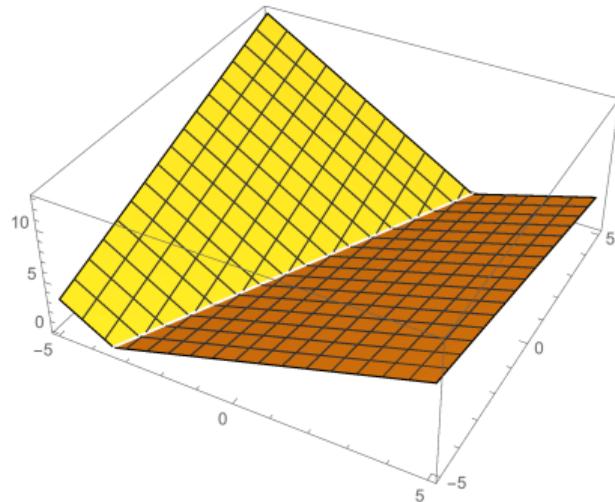


The maxout generalizes ReLU and LReLU.

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

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

# Maxout



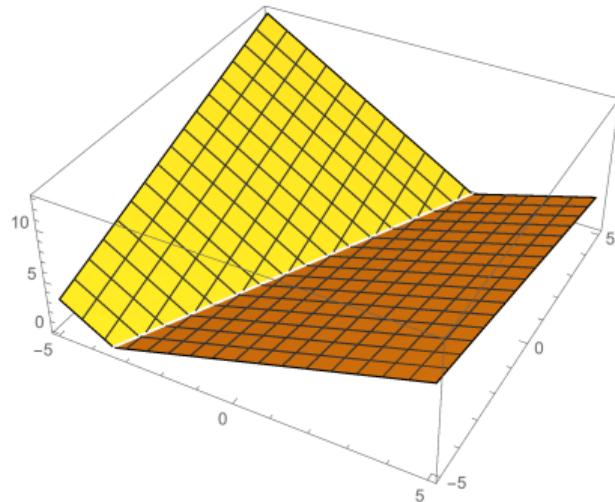
The maxout generalizes ReLU and LReLU.

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

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

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

# Maxout



The maxout generalizes ReLU and LReLU.

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

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

- 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

- ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)
- ② Neural Networks
- ③ Neural Networks for Images
- ④ Neural Networks for Sequences

# Introduction: Why Not MLPs for Images?

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

# Introduction: Why Not MLPs for Images?

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

# Introduction: Why Not MLPs for Images?

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

# Introduction: Why Not MLPs for Images?

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

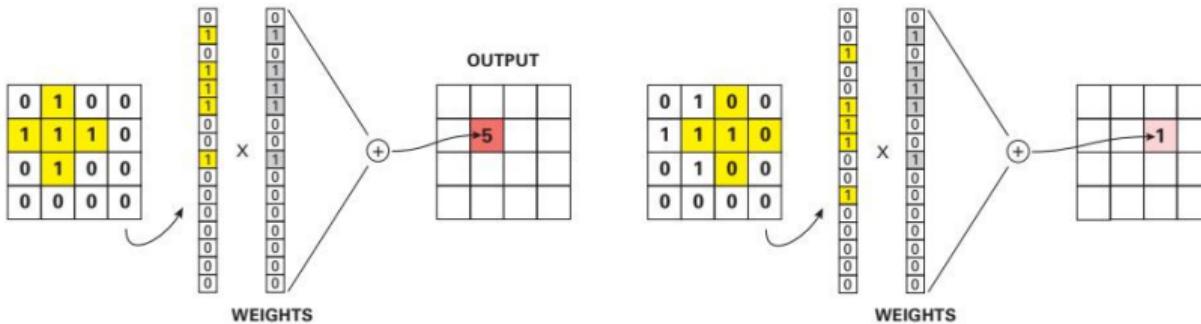


Figure: Detecting patterns with MLPs lacks translation invariance. A matched filter (weight vector) gives a strong response when the pattern aligns perfectly (left) but a weak response when shifted (right).

# 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  $w$ , vector  $x$ ):

# Common Layers: Convolution in 1D

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

# Common Layers: Convolution in 1D

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

# Common Layers: Convolution in 1D

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

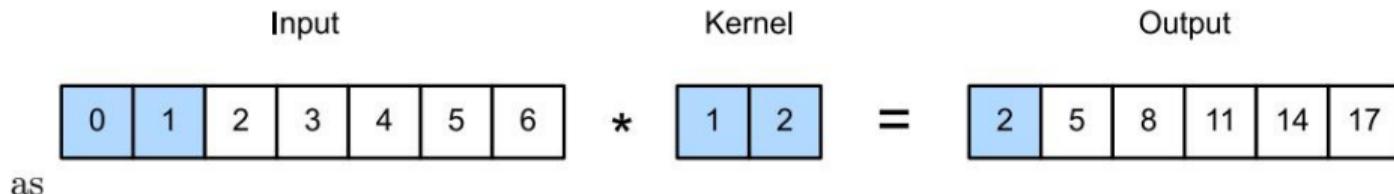


Figure: 1D cross-correlation: Sliding a filter (bottom) over an input sequence (top) to produce an output sequence (middle).

*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  $\mathbf{X}$ .

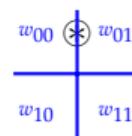
# Convolution in 2D

- 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 X**

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

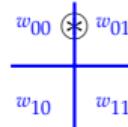
**Filter W****Feature Map Y**


=

# Step-by-Step: 2D Convolution Operation

**Input 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 W****Feature Map Y**

$y_{00}$		

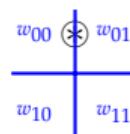
=

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

# Step-by-Step: 2D Convolution Operation

**Input 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 W****Feature Map Y**

	$y_{01}$	

=

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

# Step-by-Step: 2D Convolution Operation

**Input 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 W**

$$\begin{array}{c} w_{00} \otimes w_{01} \\ \hline w_{10} & w_{11} \end{array}$$

**Feature Map Y**

$y_{10}$		

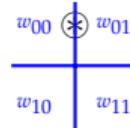
=

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

# Step-by-Step: 2D Convolution Operation

**Input 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 W****Feature Map Y**

$y$	$y$	$y$
$y$	$y$	$y$
$y$	$y$	$y$

 $=$ 

Process repeats for all spatial locations.

## 2D Convolution as Feature Detection

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

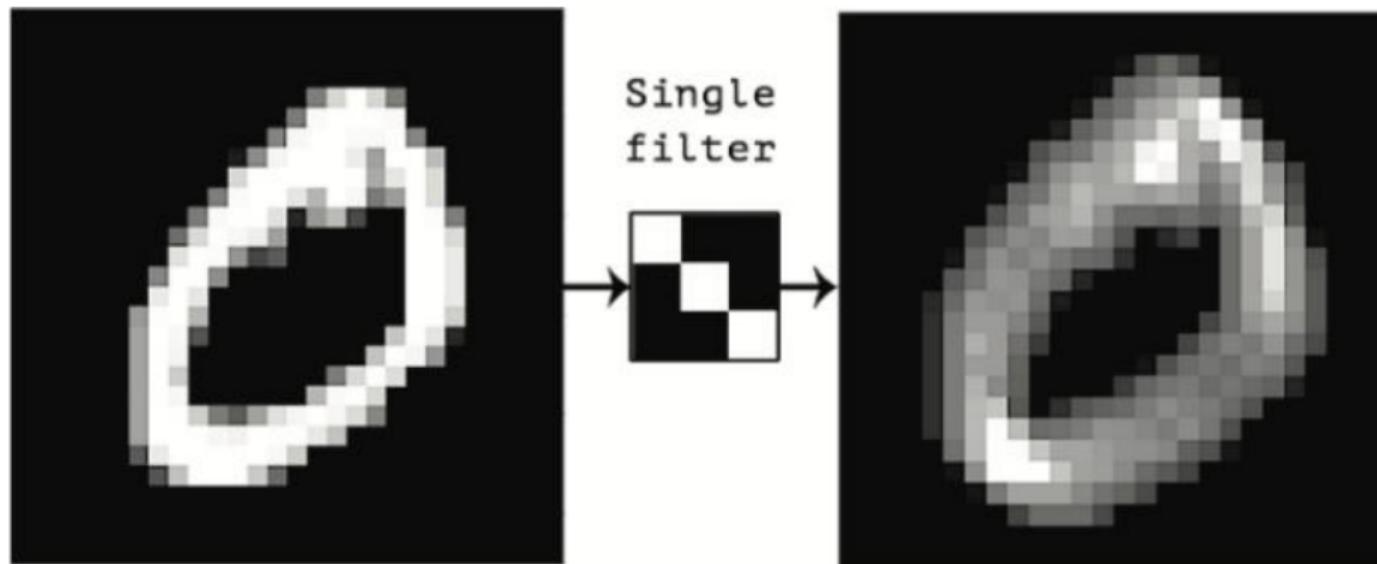


Figure: Convolving an image (left) with a 3x3 filter detecting diagonal lines (middle) produces a feature map (right) highlighting those features.

# 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  $x$  by a specially structured matrix  $C$  derived from the filter  $W$ .

# Convolution as Matrix Multiplication

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

# Convolution as Matrix Multiplication

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

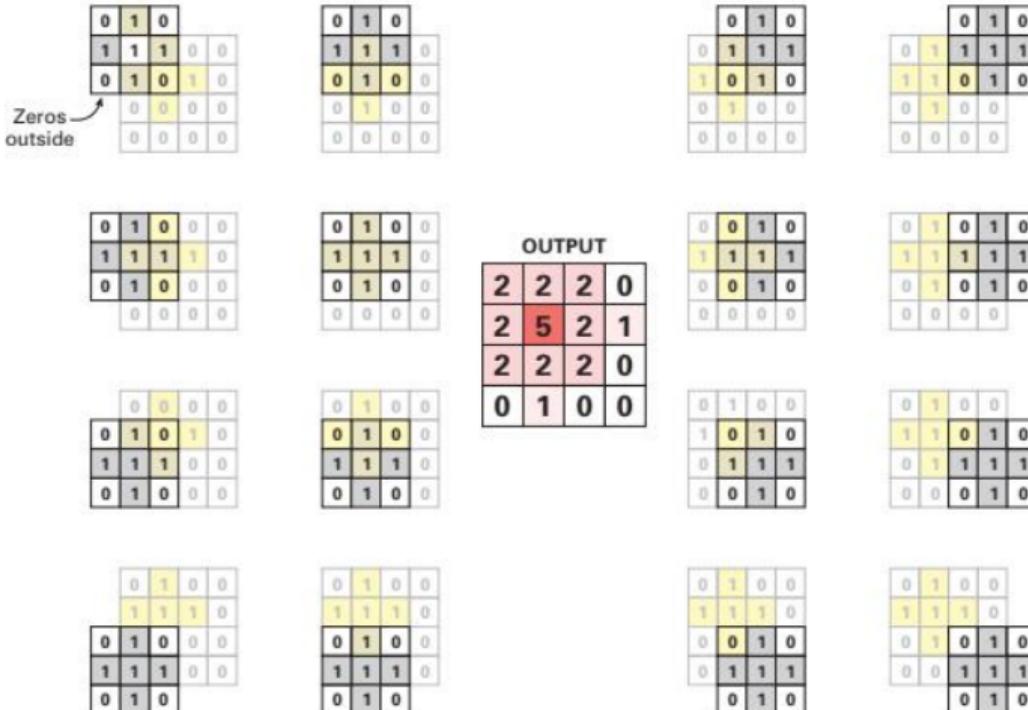


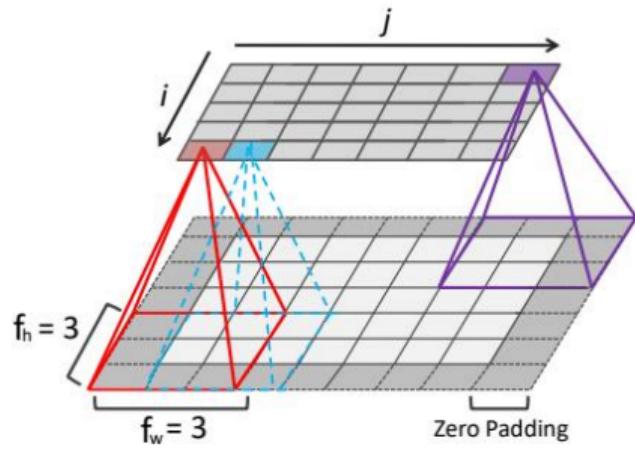
Figure: 'Same' convolution uses zero-padding to keep output size equal to input size.

# Strided Convolution

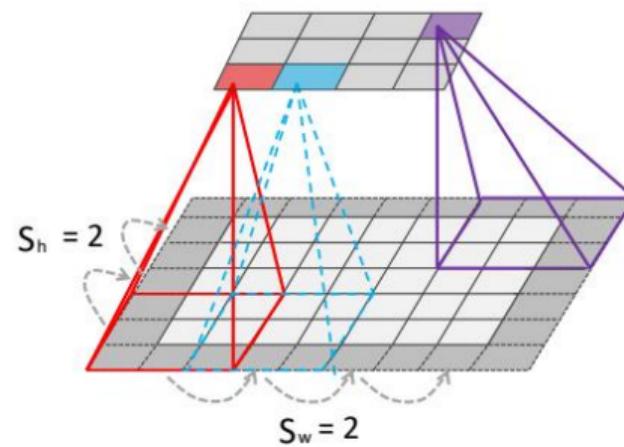
- Problem: Neighboring outputs in feature maps are often redundant due to overlapping input patches.
- Solution: **Strided Convolution** skips inputs by a step size (stride)  $s$ .
- Reduces output size and computation.
- Output size with stride  $s^h, s^w$  and padding  $p^h, p^w$ :

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

# Padding and Stride Example



(a)



(b)

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

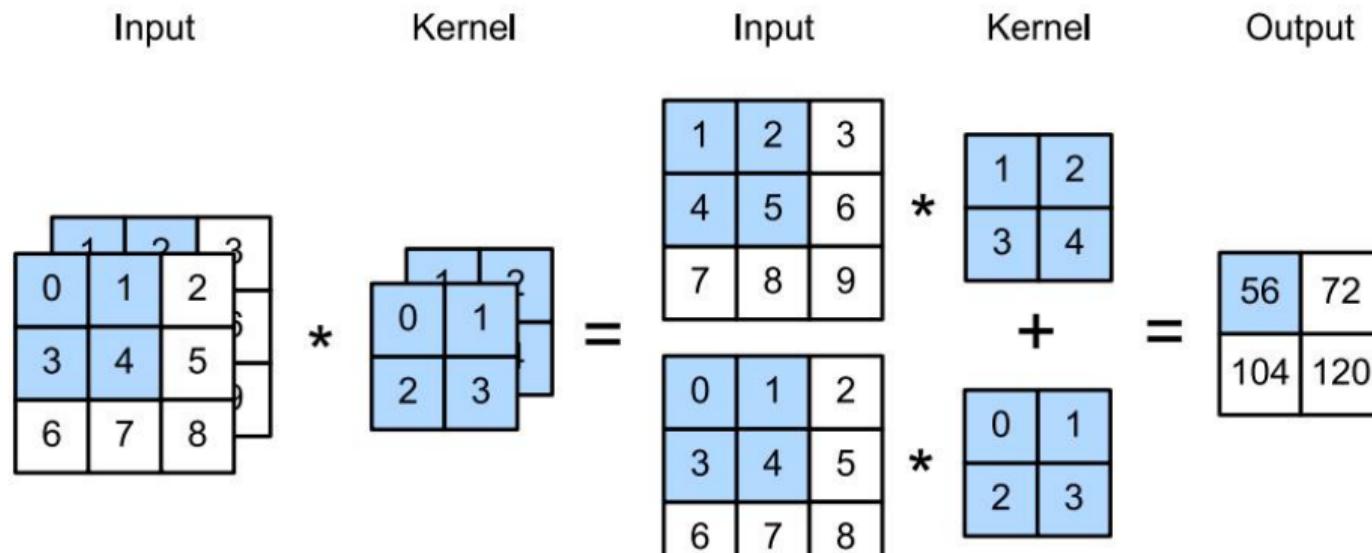


Figure: 2D convolution with a 2-channel input. Each input channel is convolved with a corresponding 2D filter slice, and the results are summed.

# 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

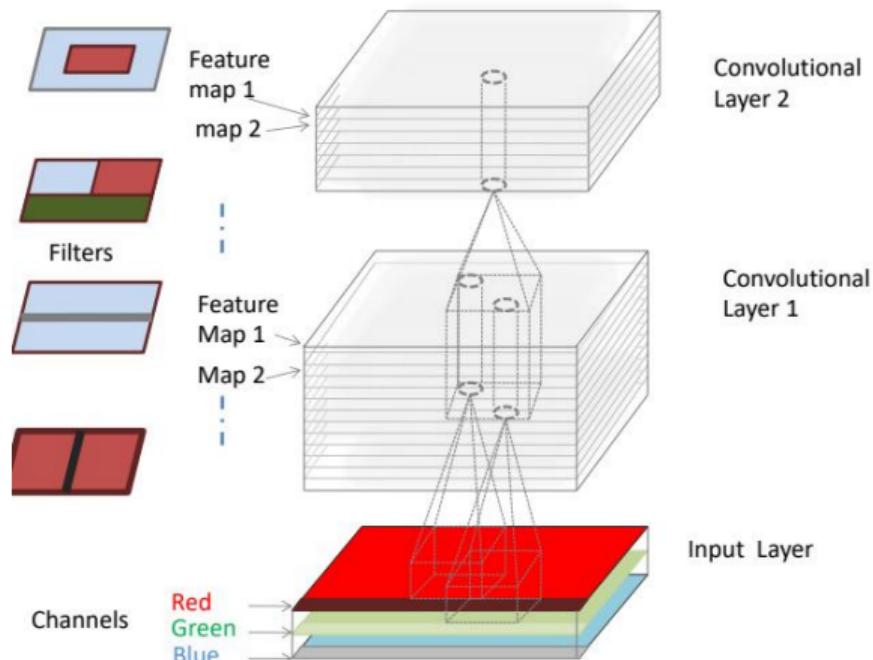


Figure: CNN with multiple channels. Input (3 channels) -> Conv Layer 1 (multiple channels) -> Conv Layer 2 (more channels). Cylinders represent feature vectors (hypercolumns) at specific locations.

# 1x1 Convolution (Pointwise Convolution)

- A special case with filter size  $1 \times 1$ .
- 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 ( $C \rightarrow D$ ) 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

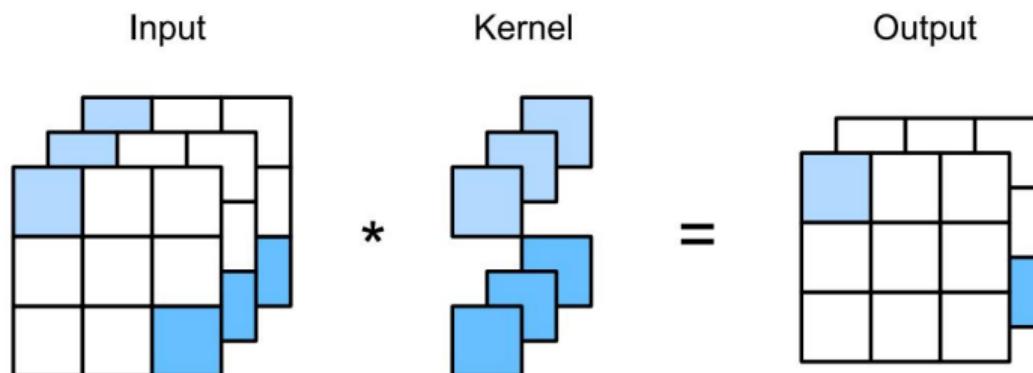


Figure: Mapping 3 input channels to 2 output channels using 1x1 convolution (filter size 1x1x3x2).

# 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.

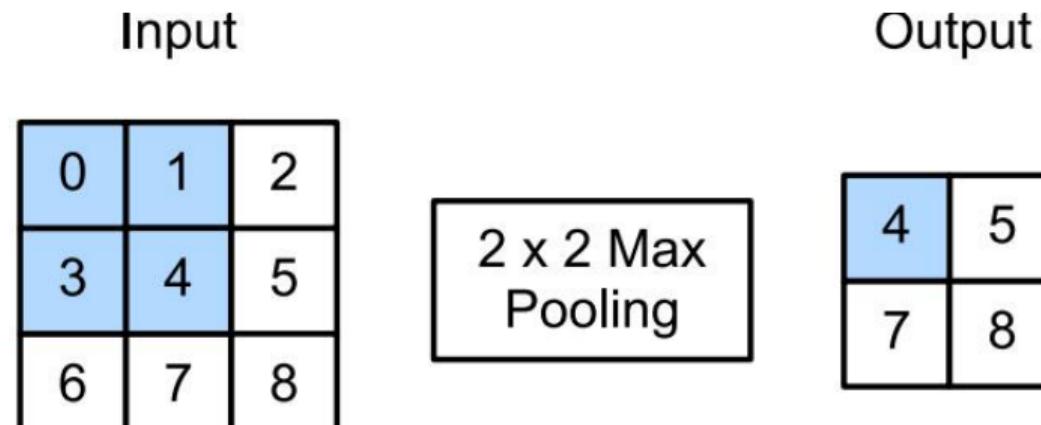


Figure: Max pooling with a 2x2 filter and stride 2.

# Pooling Layers: Max Pooling

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

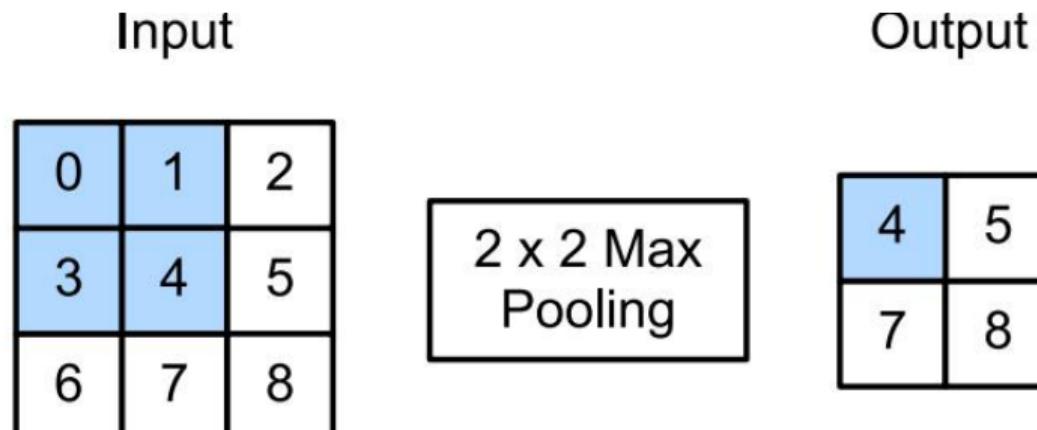


Figure: Max pooling with a 2x2 filter and stride 2.

# 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).

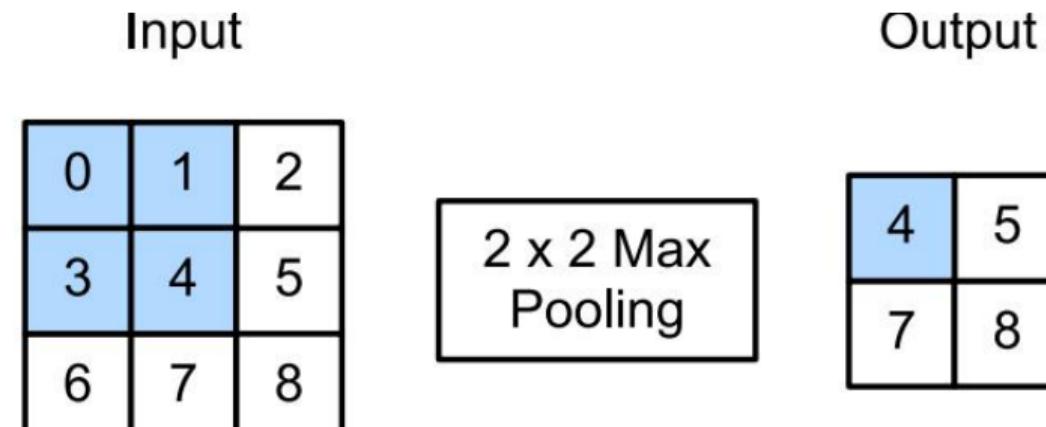


Figure: Max pooling with a 2x2 filter and stride 2.

# 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.

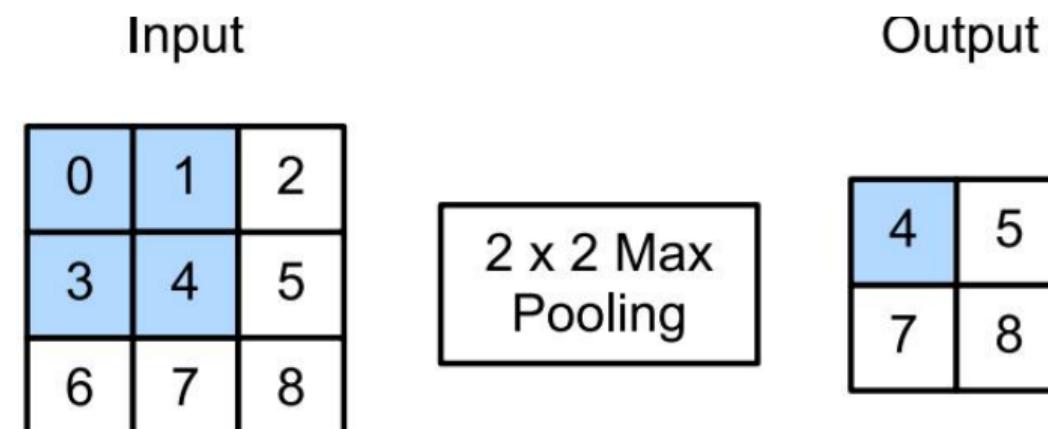


Figure: Max pooling with a 2x2 filter and stride 2.

# 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.

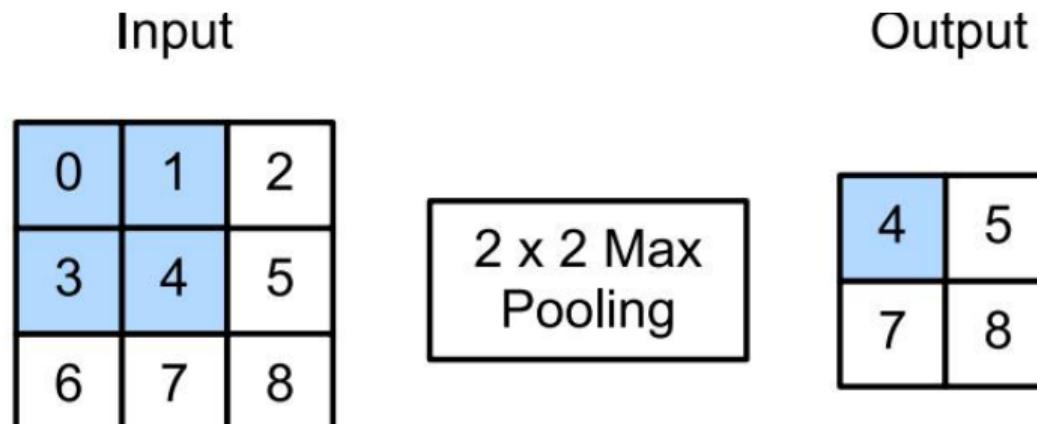


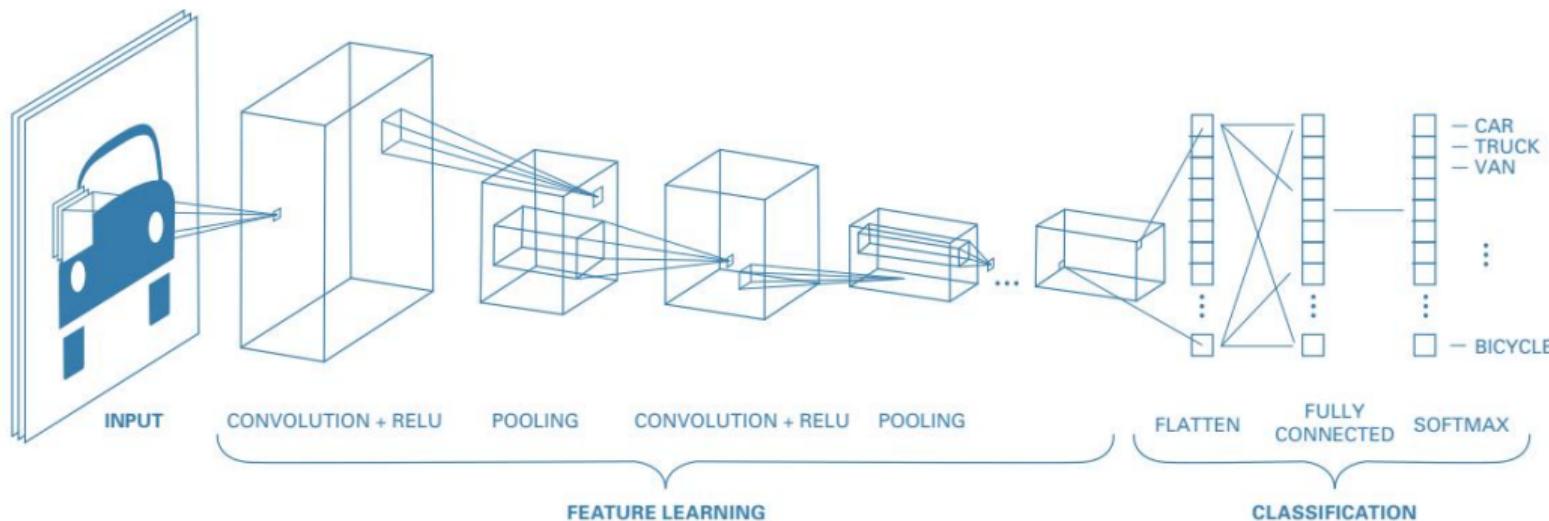
Figure: Max pooling with a 2x2 filter and stride 2.

# 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]  $\times N$  -> 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  $\sigma_{\mathcal{B}}^2$ .

# 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  $\sigma_{\mathcal{B}}^2$ .
  - ② Normalize:  $\hat{z}_n = (z_n - \mu_{\mathcal{B}}) / \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$

# 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  $\sigma_{\mathcal{B}}^2$ .
  - ② Normalize:  $\hat{z}_n = (z_n - \mu_{\mathcal{B}}) / \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$
  - ③ Scale and Shift:  $\tilde{z}_n = \gamma \odot \hat{z}_n + \beta$  ( $\gamma, \beta$  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  $\sigma_{\mathcal{B}}^2$ .
  - ② Normalize:  $\hat{z}_n = (z_n - \mu_{\mathcal{B}}) / \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$
  - ③ Scale and Shift:  $\tilde{z}_n = \gamma \odot \hat{z}_n + \beta$  ( $\gamma, \beta$  are learnable parameters).
- Applied after CONV/DENSE layers, often before activation function.

# Batch Normalization: Training vs. Test Time

- **Training:** Use mini-batch statistics  $(\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}^2)$ . Learn  $\gamma, \beta$ .
- **Testing:** Mini-batch statistics are unreliable (batch size might be 1).
  - Use population statistics (mean  $\mu$ , variance  $\sigma^2$ ) 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

- ① Multi-Layer Perceptron (MLP) and Back-Propagation (BP)
- ② Neural Networks
- ③ Neural Networks for Images
- ④ Neural Networks for Sequences

# Why Sequence Models?

**Real-World Sequential Data is Everywhere:**

# Why Sequence Models?

**Real-World Sequential Data is Everywhere:**

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

# Why Sequence Models?

## Real-World Sequential Data is Everywhere:

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

- **Speech & Audio:**

- Speech Recognition
- Music Generation
- Audio Classification

# Why Sequence Models?

## Real-World Sequential Data is Everywhere:

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

- **Speech & Audio:**

- Speech Recognition
- Music Generation
- Audio Classification

- **Time Series:**

- Stock Price Prediction
- Weather Forecasting
- Sensor Data Analysis

# Why Sequence Models?

## Real-World Sequential Data is Everywhere:

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

- **Speech & Audio:**

- Speech Recognition
- Music Generation
- Audio Classification

- **Time Series:**

- Stock Price Prediction
- Weather Forecasting
- Sensor Data Analysis

- **Video:**

- Action Recognition
- Video Captioning

# Why Sequence Models?

## Real-World Sequential Data is Everywhere:

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

- **Speech & Audio:**

- Speech Recognition
- Music Generation
- Audio Classification

- **Time Series:**

- Stock Price Prediction
- Weather Forecasting
- Sensor Data Analysis

- **Video:**

- Action Recognition
- Video Captioning

### Key Challenge

Sequential data has:

- **Variable length  $T$**
- **Temporal dependencies**
- **Order matters!**

# Why Sequence Models?

## Real-World Sequential Data is Everywhere:

- **Natural Language:**

- Machine Translation
- Question Answering
- Text Generation

- **Speech & Audio:**

- Speech Recognition
- Music Generation
- Audio Classification

- **Time Series:**

- Stock Price Prediction
- Weather Forecasting
- Sensor Data Analysis

- **Video:**

- Action Recognition
- Video Captioning

### Key Challenge

Sequential data has:

- **Variable length  $T$**
- **Temporal dependencies**
- **Order matters!**

**Standard MLPs cannot handle this!**

they expect fixed-size input and treat features  
independently

# Introduction to Sequence Modeling

- **The Data:** Sequential data  $\mathbf{x} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$ .

# Introduction to Sequence Modeling

- **The Data:** Sequential data  $\mathbf{x} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$ .
- **Examples:**
  - Text: “The” → “cat” → “sat”.
  - Time Series: Stock prices over days.
  - Audio: Waveform samples over time.

# Introduction to Sequence Modeling

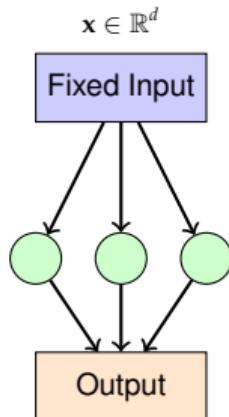
- **The Data:** Sequential data  $\mathbf{x} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$ .
- **Examples:**
  - Text: “The” → “cat” → “sat”.
  - Time Series: Stock prices over days.
  - Audio: Waveform samples over time.
- **The Challenge:**
  - Variable length  $T$ .
  - Long-term dependencies (e.g., “The **boy** who wore a red hat ... **was** happy”).

# Introduction to Sequence Modeling

- **The Data:** Sequential data  $\mathbf{x} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$ .
- **Examples:**
  - Text: “The” → “cat” → “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.

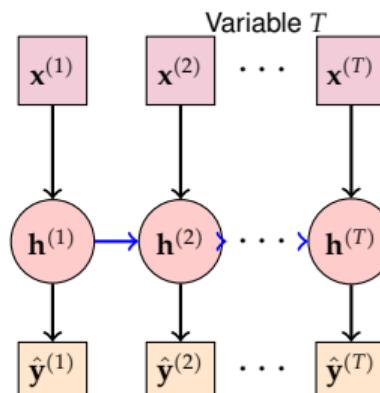
# Fixed-Size (MLP) vs. Variable-Length (RNN)

Standard MLP



**Problem:** Cannot handle sequences of different lengths!

RNN



**Solution:** Process one step at a time, **share weights** across time!

# The Language Modeling Task

- **Goal:** Predict what word comes next.

# The Language Modeling Task

- **Goal:** Predict what word comes next.
- **Example:** “the students opened their \_\_\_\_\_”

# The Language Modeling Task

- **Goal:** Predict what word comes next.
- **Example:** “the students opened their \_\_\_\_\_”
- Possible continuations:
  - books ( $P = 0.4$ )
  - laptops ( $P = 0.3$ )
  - exams ( $P = 0.1$ )

# The Language Modeling Task

- **Goal:** Predict what word comes next.
- **Example:** “the students opened their \_\_\_\_\_”
- Possible continuations:
  - books ( $P = 0.4$ )
  - laptops ( $P = 0.3$ )
  - exams ( $P = 0.1$ )
- **Formal Definition:**

Given a sequence of words  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$ , compute the probability distribution of the next word  $\mathbf{x}^{(t+1)}$ :

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$$

# The Language Modeling Task

- **Goal:** Predict what word comes next.
- **Example:** “the students opened their \_\_\_\_\_”
- Possible continuations:
  - books ( $P = 0.4$ )
  - laptops ( $P = 0.3$ )
  - exams ( $P = 0.1$ )
- **Formal Definition:**

Given a sequence of words  $x^{(1)}, \dots, x^{(t)}$ , compute the probability distribution of the next word  $x^{(t+1)}$ :

$$P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$$

- A system that does this is called a **Language Model (LM)**.

# The Language Modeling Task

- **Goal:** Predict what word comes next.
- **Example:** “the students opened their \_\_\_\_\_”
- Possible continuations:
  - books ( $P = 0.4$ )
  - laptops ( $P = 0.3$ )
  - exams ( $P = 0.1$ )
- **Formal Definition:**

Given a sequence of words  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$ , compute the probability distribution of the next word  $\mathbf{x}^{(t+1)}$ :

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$$

- A system that does this is called a **Language Model (LM)**.
- **Connection to Probability Theory:** By the chain rule, the probability of a sequence is:

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) = \prod_{t=1}^T P(\mathbf{x}^{(t)} | \mathbf{x}^{<t})$$

# Why not N-gram Language Models?

- **N-grams:** Count statistics of  $n$  consecutive words.

$$P(\text{books}|\text{opened their}) \approx \frac{\text{count(opened their books)}}{\text{count(opened their)}}$$

# Why not N-gram Language Models?

- **N-grams:** Count statistics of  $n$  consecutive words.

$$P(\text{books}|\text{opened their}) \approx \frac{\text{count(opened their books)}}{\text{count(opened their)}}$$

- **Sparsity issue:** What if “students opened their  $w$ ” never occurred in the training data?
  - Then probability is 0 (or requires smoothing).
  - If the context “students opened their” never occurred, we can’t calculate anything.

# Why not N-gram Language Models?

- **N-grams:** Count statistics of  $n$  consecutive words.

$$P(\text{books}|\text{opened their}) \approx \frac{\text{count(opened their books)}}{\text{count(opened their)}}$$

- **Sparsity issue:** What if “students opened their  $w$ ” never occurred in the training data?
  - Then probability is 0 (or requires smoothing).
  - If the context “students opened their” never occurred, we can’t calculate anything.
- **Storage issue:** Must store counts for all observed n-grams. Model size grows with corpus size.

# Why not N-gram Language Models?

- **N-grams:** Count statistics of  $n$  consecutive words.

$$P(\text{books}|\text{opened their}) \approx \frac{\text{count(opened their books)}}{\text{count(opened their)}}$$

- **Sparsity issue:** What if “students opened their  $w$ ” never occurred in the training data?
  - Then probability is 0 (or requires smoothing).
  - If the context “students opened their” never occurred, we can’t calculate anything.
- **Storage issue:** Must store counts for all observed n-grams. Model size grows with corpus size.
- **Solution:**

Neural LMs solve it by using distributed representations (embeddings) and sharing weights.

# N-gram vs. Neural Language Models

Feature	N-gram LM	Neural LM
Parameters	$O(V^n)$	$O(Vd + d^2)$
Storage	Grows with corpus	Fixed (weight matrices)
Unseen sequences	Zero probability	Can generalize
Long context	Limited by $n$	Can use arbitrary $T$
Similarity	No notion	Embeddings capture similarity
Training	Count & normalize	Gradient descent

**Key Insight:** Neural LMs use **distributed representations** (word embeddings) to share statistical strength between similar words and contexts.

where  $V$  = vocabulary size,  $n$  = n-gram size,  $d$  = embedding dimension

# Recurrent Neural Networks (RNNs): The Core Idea

- **Core Concept:**

Process the sequence one step at a time,  
maintaining an internal “memory” or **Hidden State** ( $h^{(t)}$ ).

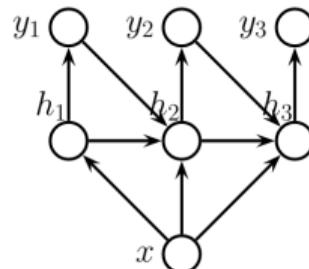


Figure: The hidden state  $h$  passes information forward through time.

# 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:**

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

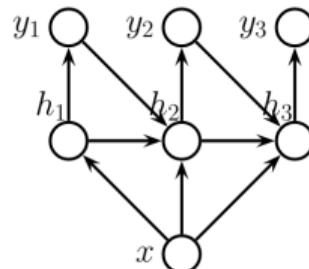


Figure: The hidden state  $\mathbf{h}$  passes information forward through time.

# 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  $\hat{\mathbf{y}}^{(t)}$  from  $\mathbf{h}^{(t)}$ .

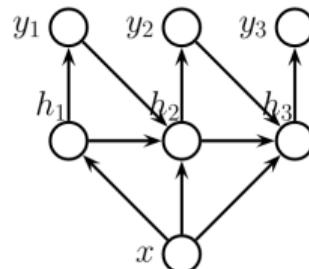


Figure: The hidden state  $\mathbf{h}$  passes information forward through time.

# RNN: Mathematical Formulation

- A vanilla RNN typically uses the tanh activation function:

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

# RNN: Mathematical Formulation

- A vanilla RNN typically uses the tanh activation function:

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

- **Dimensions:**

- Input:  $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$  (e.g., word embedding)
- Hidden:  $\mathbf{h}^{(t)} \in \mathbb{R}^{d_h}$  (captures context)
- Output:  $\hat{\mathbf{y}}^{(t)} \in \mathbb{R}^{d_y}$  (e.g., vocabulary size  $V$ )

# RNN: Mathematical Formulation

- A vanilla RNN typically uses the tanh activation function:

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

- **Dimensions:**

- Input:  $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$  (e.g., word embedding)
- Hidden:  $\mathbf{h}^{(t)} \in \mathbb{R}^{d_h}$  (captures context)
- Output:  $\hat{\mathbf{y}}^{(t)} \in \mathbb{R}^{d_y}$  (e.g., vocabulary size  $V$ )

- **Weight Matrices:**

- $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d_x}$ : Input to hidden
- $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ : Hidden to hidden (**recurrence**)
- $\mathbf{W}_{hy} \in \mathbb{R}^{d_y \times d_h}$ : Hidden to output

# RNN: Mathematical Formulation

- A vanilla RNN typically uses the tanh activation function:

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

- **Dimensions:**

- Input:  $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$  (e.g., word embedding)
- Hidden:  $\mathbf{h}^{(t)} \in \mathbb{R}^{d_h}$  (captures context)
- Output:  $\hat{\mathbf{y}}^{(t)} \in \mathbb{R}^{d_y}$  (e.g., vocabulary size  $V$ )

- **Weight Matrices:**

- $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d_x}$ : Input to hidden
- $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ : Hidden to hidden (**recurrence**)
- $\mathbf{W}_{hy} \in \mathbb{R}^{d_y \times d_h}$ : Hidden to output

- **Key:** **Same weights**  $\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}$  **used at every time step!**

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

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

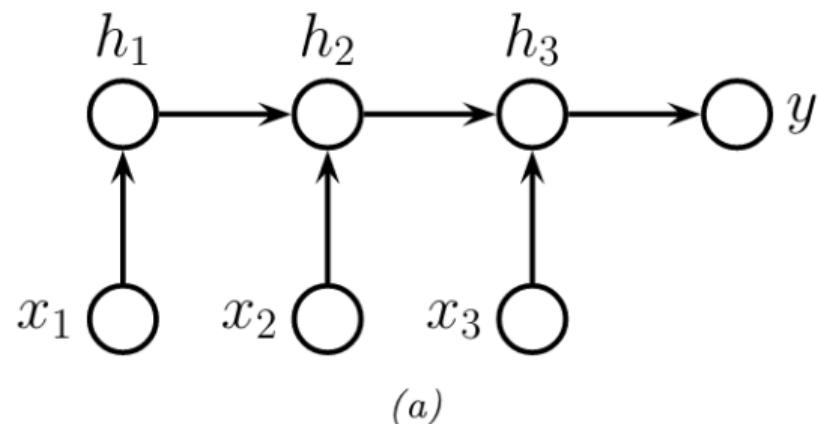


Figure: Basic RNN for sequence classification where only the final output is used.

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

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

- **Process:**

- Read entire sequence  $x^{(1)}, \dots, x^{(T)}$ .
- Update state:  $h^{(0)} \rightarrow h^{(1)} \rightarrow \dots \rightarrow h^{(T)}$ .
- Use **final state**  $h^{(T)}$  to predict label  $y$ .

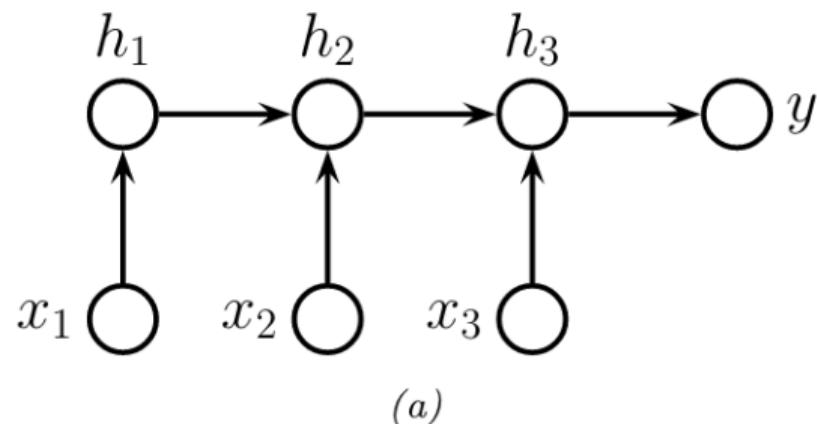


Figure: Basic RNN for sequence classification where only the final output is used.

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

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

- **Process:**

- Read entire sequence  $x^{(1)}, \dots, x^{(T)}$ .
- Update state:  $h^{(0)} \rightarrow h^{(1)} \rightarrow \dots \rightarrow h^{(T)}$ .
- Use **final state**  $h^{(T)}$  to predict label  $y$ .

- **Intuition:**  $h^{(T)}$  is a vector summary of the whole sentence.

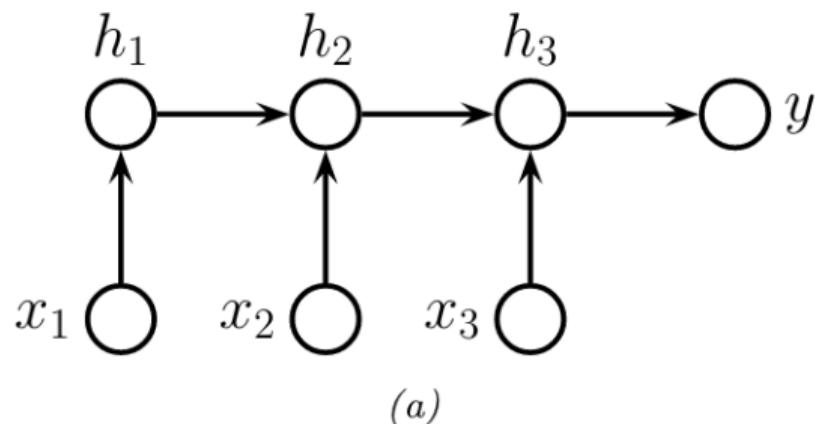
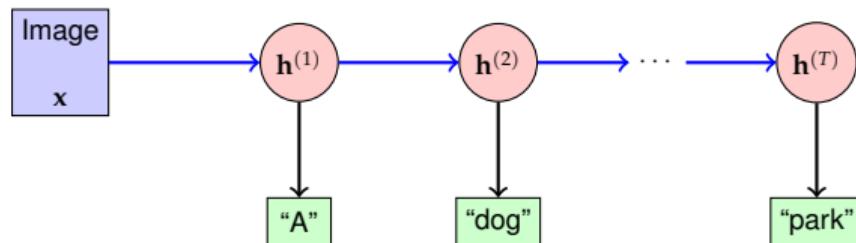


Figure: Basic RNN for sequence classification where only the final output is used.

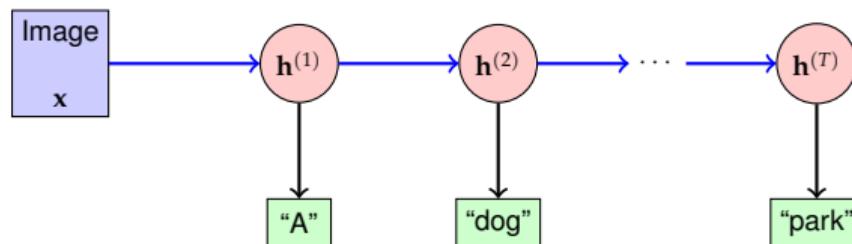
## Architecture 2: One-to-Many (Sequence Generation)

- **Task:** Image Captioning, Music Generation.



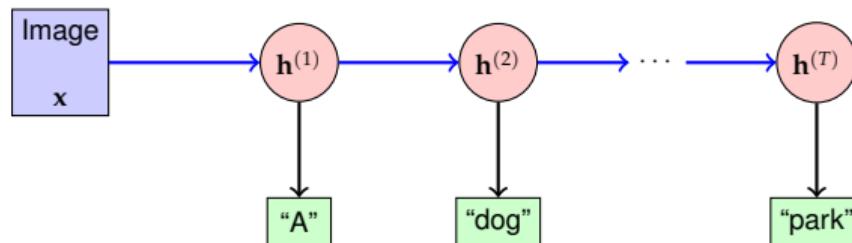
## Architecture 2: One-to-Many (Sequence Generation)

- **Task:** Image Captioning, Music Generation.
- **Process:**
  - **Input:** Single fixed input (e.g., image embedding  $x$ ).
  - **Initialize:**  $\mathbf{h}^{(0)} = f(x)$  (encode input into initial state).
  - **Generate:** Produce sequence  $y^{(1)}, y^{(2)}, \dots, y^{(T)}$  autoregressively.



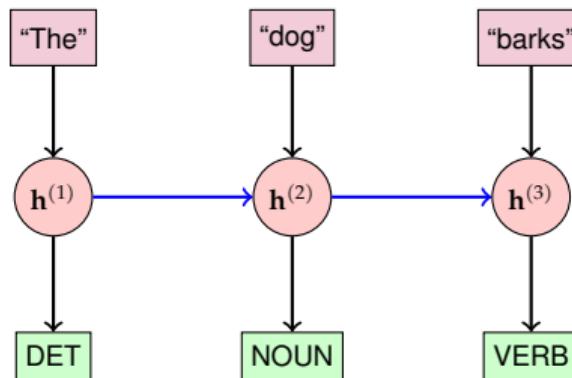
## Architecture 2: One-to-Many (Sequence Generation)

- **Task:** Image Captioning, Music Generation.
- **Process:**
  - **Input:** Single fixed input (e.g., image embedding  $x$ ).
  - **Initialize:**  $\mathbf{h}^{(0)} = f(x)$  (encode input into initial state).
  - **Generate:** Produce sequence  $y^{(1)}, y^{(2)}, \dots, y^{(T)}$  autoregressively.
- **Example:** Image  $\rightarrow$  “A dog playing in the park”



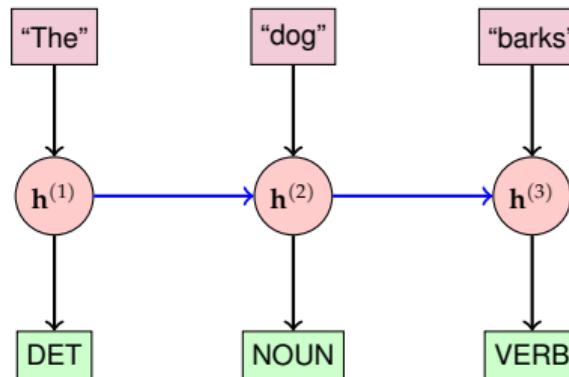
## Architecture 3: Many-to-Many (Aligned)

- **Task:** Part-of-Speech Tagging, Named Entity Recognition, Video Frame Labeling.



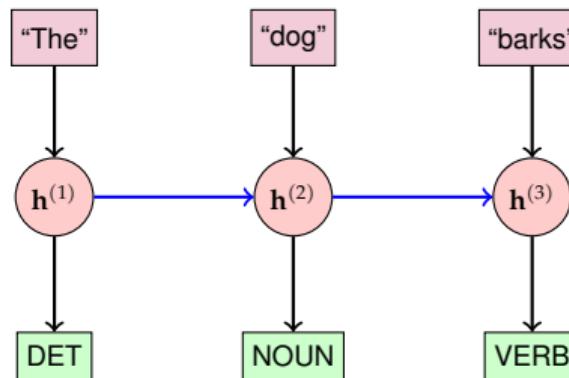
## Architecture 3: Many-to-Many (Aligned)

- **Task:** Part-of-Speech Tagging, Named Entity Recognition, Video Frame Labeling.
- **Process:**
  - **Input:** Sequence  $x^{(1)}, \dots, x^{(T)}$ .
  - **Output:** Aligned sequence  $y^{(1)}, \dots, y^{(T)}$  (same length).
  - Each output  $y^{(t)}$  depends on input  $x^{(t)}$  and context from  $h^{(t)}$ .



## Architecture 3: Many-to-Many (Aligned)

- **Task:** Part-of-Speech Tagging, Named Entity Recognition, Video Frame Labeling.
- **Process:**
  - **Input:** Sequence  $x^{(1)}, \dots, x^{(T)}$ .
  - **Output:** Aligned sequence  $y^{(1)}, \dots, y^{(T)}$  (same length).
  - Each output  $y^{(t)}$  depends on input  $x^{(t)}$  and context from  $h^{(t)}$ .
- **Example:** “The dog barks” → [DET, NOUN, VERB]



## Architecture 4: Many-to-Many (Seq2Seq)

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

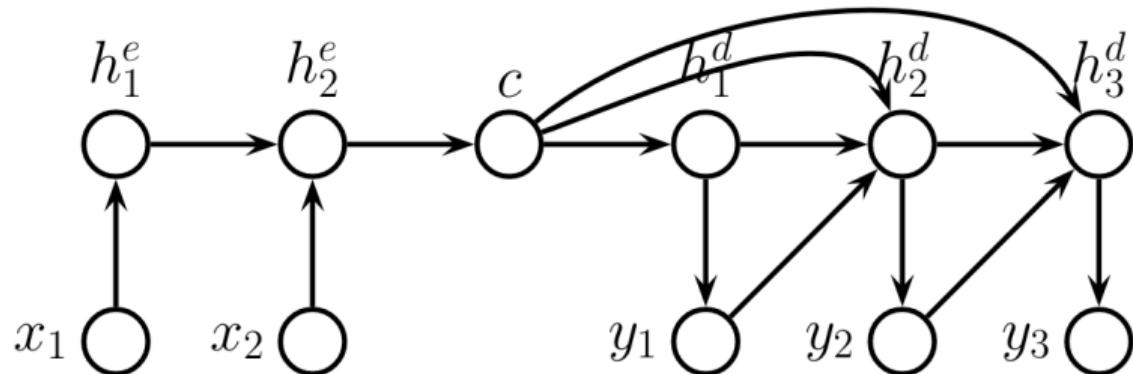


Figure: The context vector  $c$  is the bottleneck passing info from Encoder to Decoder.

## Architecture 4: Many-to-Many (Seq2Seq)

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

- **Encoder-Decoder Architecture:**

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

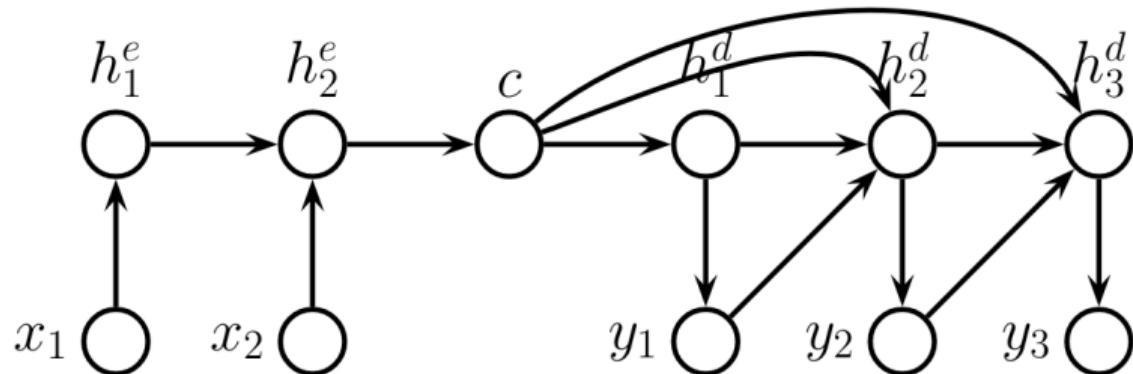


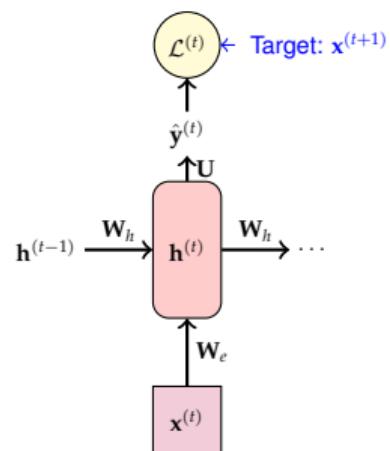
Figure: The context vector  $c$  is the bottleneck passing info from Encoder to Decoder.

# RNN Language Model Training

- Let's consider input sequence:  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ .

# RNN Language Model Training

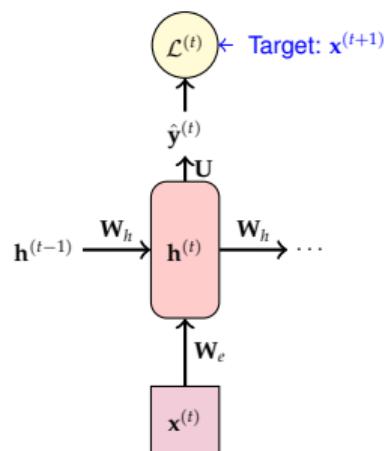
- Let's consider input sequence:  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ .
- At every step  $t$ , the RNN predicts the probability distribution  $\hat{\mathbf{y}}^{(t)}$  for the *next* word  $\mathbf{x}^{(t+1)}$ .



# RNN Language Model Training

- Let's consider input sequence:  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ .
- At every step  $t$ , the RNN predicts the probability distribution  $\hat{\mathbf{y}}^{(t)}$  for the *next word*  $\mathbf{x}^{(t+1)}$ .
- Loss Function:** Cross-entropy between predicted  $\hat{\mathbf{y}}^{(t)}$  and actual next word  $\mathbf{y}^{(t)}$  (one-hot of  $\mathbf{x}^{(t+1)}$ ).

$$\mathcal{L}^{(t)}(\theta) = -\log \hat{\mathbf{y}}_{\mathbf{x}^{(t+1)}}^{(t)}$$

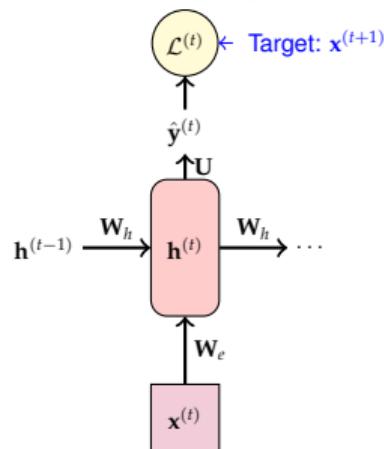


# RNN Language Model Training

- Let's consider input sequence:  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ .
- At every step  $t$ , the RNN predicts the probability distribution  $\hat{\mathbf{y}}^{(t)}$  for the *next word*  $\mathbf{x}^{(t+1)}$ .
- Loss Function:** Cross-entropy between predicted  $\hat{\mathbf{y}}^{(t)}$  and actual next word  $\mathbf{y}^{(t)}$  (one-hot of  $\mathbf{x}^{(t+1)}$ ).

$$\mathcal{L}^{(t)}(\theta) = -\log \hat{\mathbf{y}}_{\mathbf{x}^{(t+1)}}^{(t)}$$

- Total Loss:** Average over the sequence:  $\mathcal{L}(\theta) = \frac{1}{T} \sum_{t=1}^T \mathcal{L}^{(t)}(\theta)$

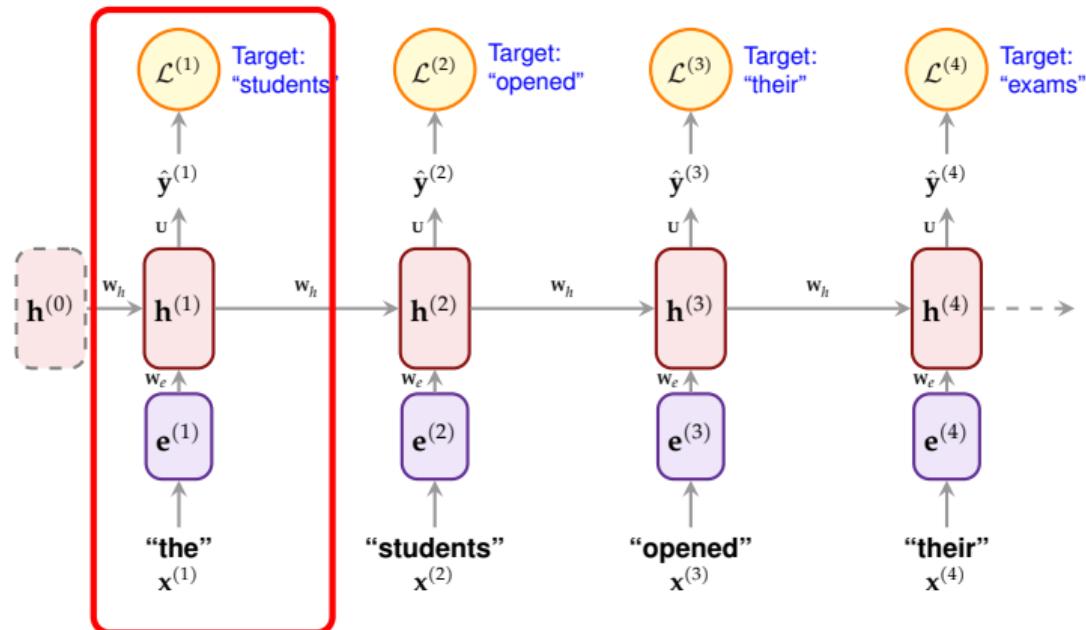


# Training an RNN Language Model (Step-by-Step)

**Step 1:** Compute loss  $\mathcal{L}^{(1)}$

Input: "the"

Target: "students"

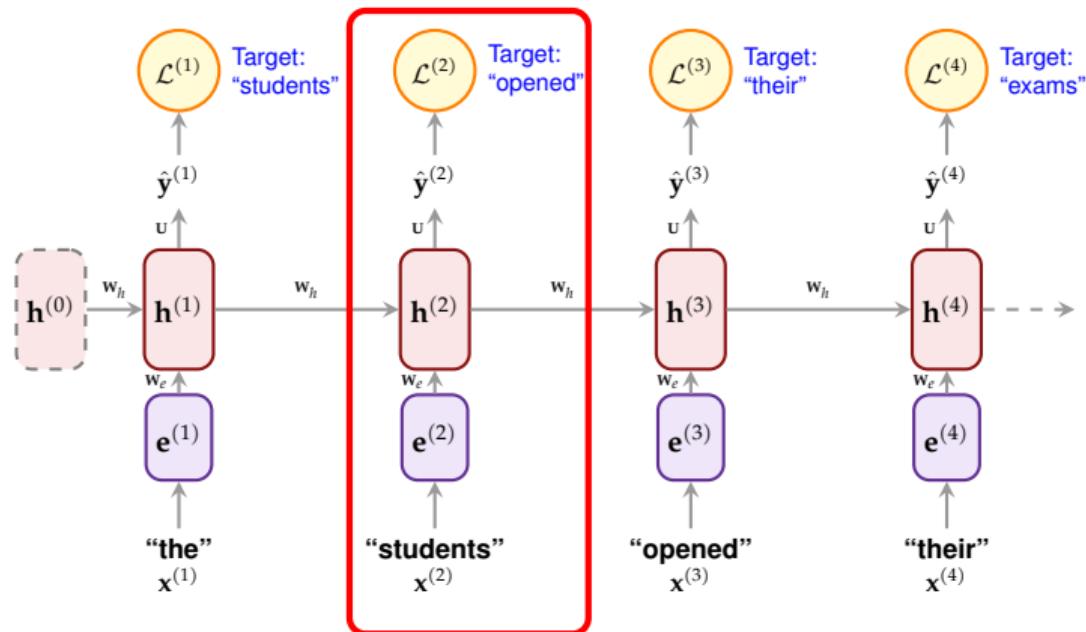


# Training an RNN Language Model (Step-by-Step)

**Step 2:** Compute loss  $\mathcal{L}^{(2)}$

Input: "students"

Target: "opened"

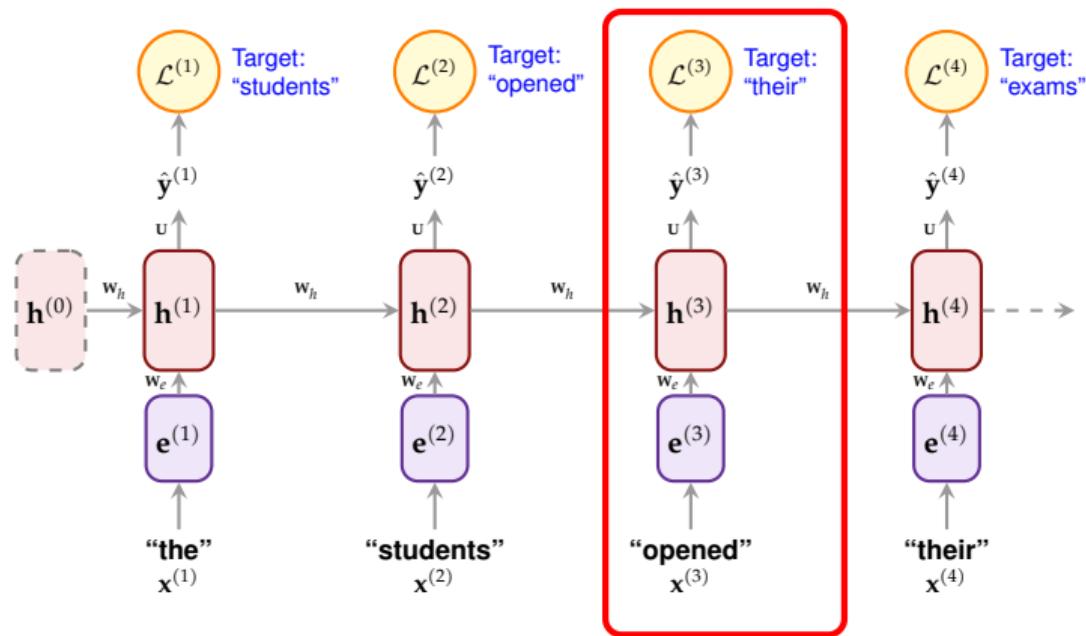


# Training an RNN Language Model (Step-by-Step)

**Step 3:** Compute loss  $\mathcal{L}^{(3)}$

Input: "opened"

Target: "their"

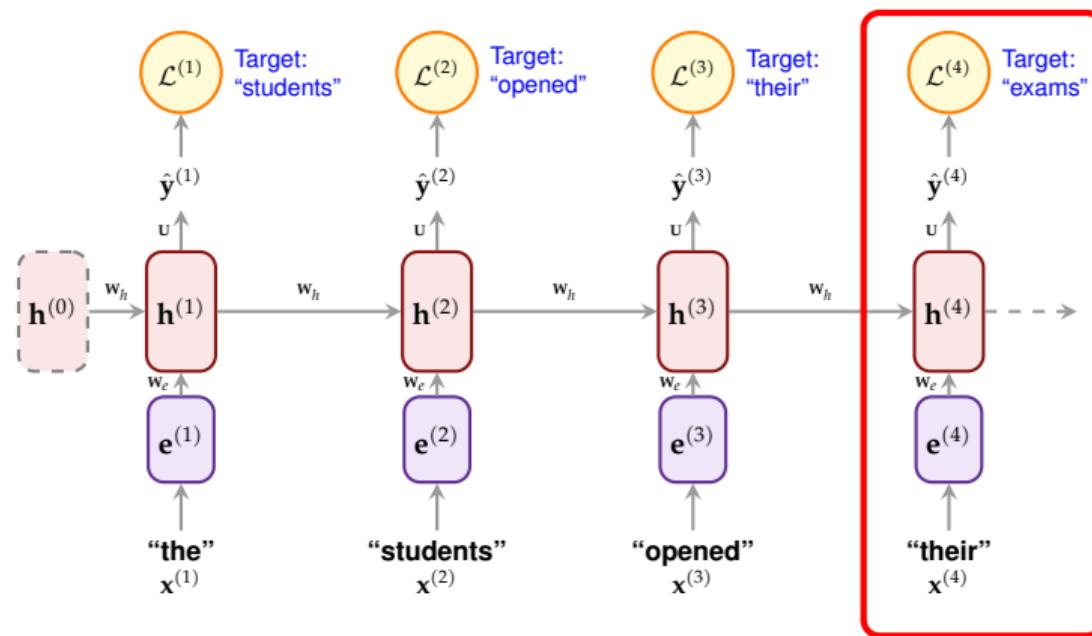


# Training an RNN Language Model (Step-by-Step)

**Step 4:** Compute loss  $\mathcal{L}^{(4)}$

Input: "their"

Target: "exams"



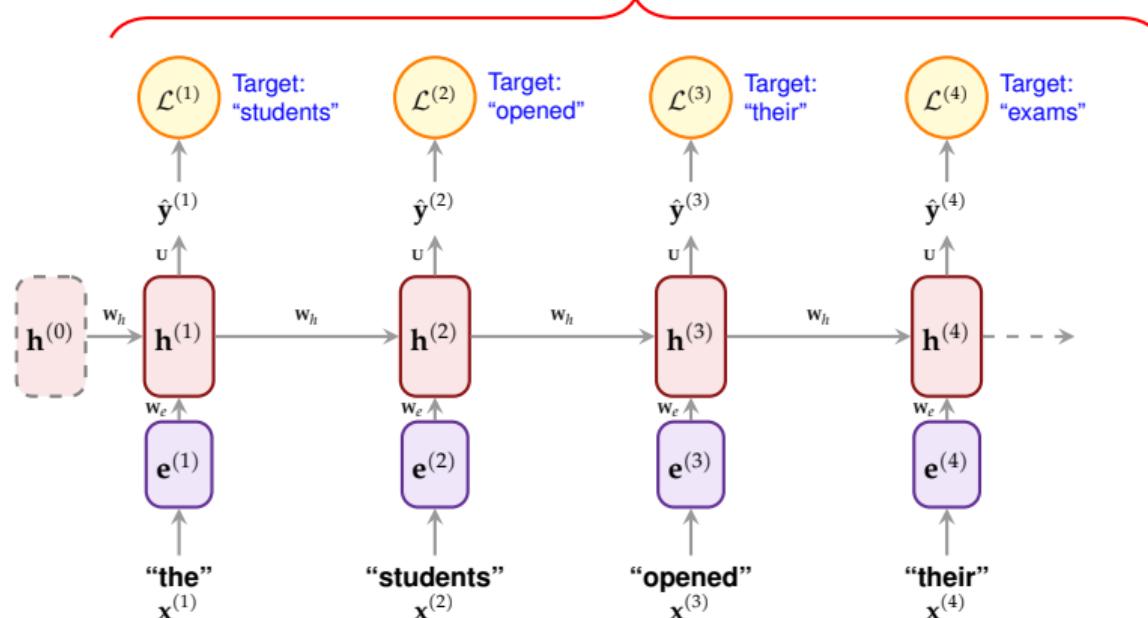
# Training an RNN Language Model (Step-by-Step)

**Final Step:** Backpropagate

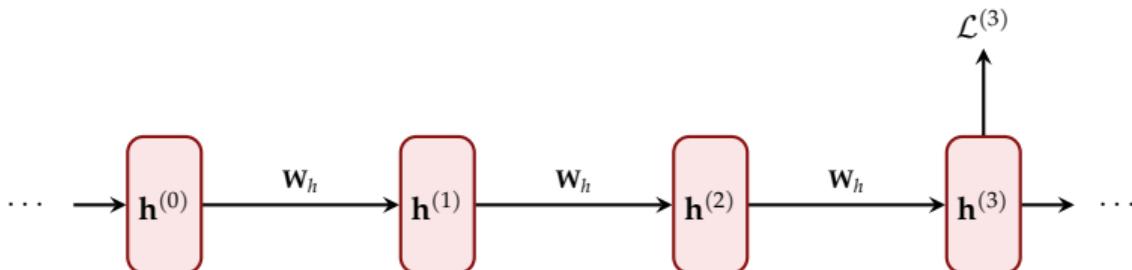
Average total loss across sequence.

Total Loss Over Sequence

$$\mathcal{L}(\theta) = \frac{1}{T} \sum_{t=1}^T \mathcal{L}^{(t)}(\theta)$$



# Backpropagation for RNNs



**Question:** What is the derivative of the loss  $\mathcal{L}^{(t)}$  with respect to the **repeated** weight matrix  $W_h$ ?

**Answer:** The gradient w.r.t. a repeated weight is the **sum of the gradients** w.r.t. each time it appears.

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}_h} = \sum_{i=1}^t \left. \frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}_h} \right|_{(i)}$$

Why? → Multivariable Chain Rule

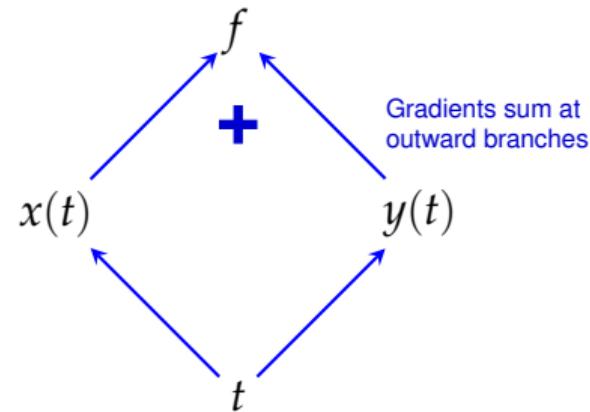
# Mathematical Basis: Multivariable Chain Rule

Given a multivariable function  $f(x, y)$ , where  $x$  and  $y$  depend on  $t$ :

$$f(x(t), y(t))$$

The derivative is the **sum** of derivatives along all paths:

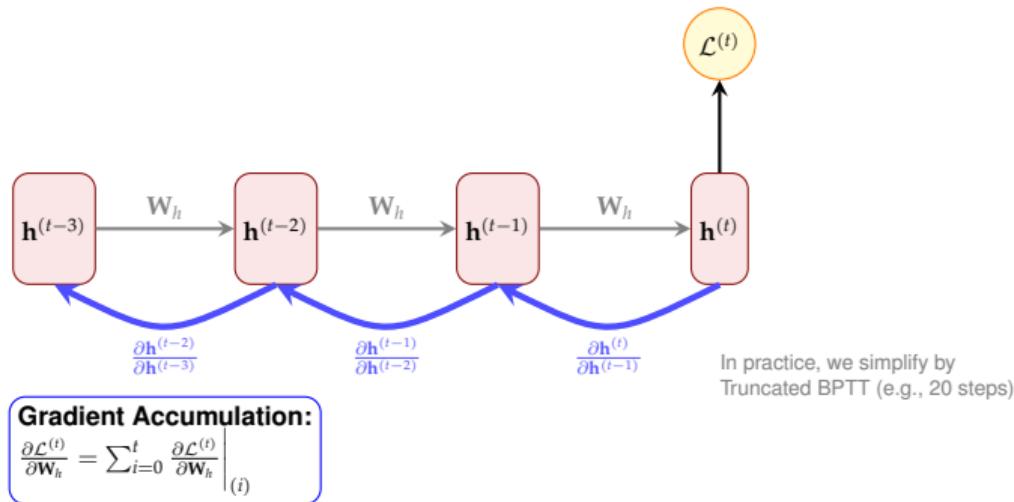
$$\frac{d}{dt} f = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$



**Example:** If  $a = x + y$ ,  $b = \max(y, z)$ , and  $f = ab$ :

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial y}$$

# Backpropagation Through Time (BPTT)



- We backpropagate gradients from the loss through the unrolled network.
- At each step, we add the gradient contribution to  $\mathbf{W}_h$ .

# The Vanishing Gradient Problem: Intuition

Why does the gradient vanish? Consider the chain rule for 4 steps:

$$\frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(1)}} = \underbrace{\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}}}_{\text{Jacobian}} \times \frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(4)}}$$

# The Vanishing Gradient Problem: Intuition

Why does the gradient vanish? Consider the chain rule for 4 steps:

$$\frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(1)}} = \underbrace{\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}}}_{\text{Jacobian}} \times \frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(4)}}$$

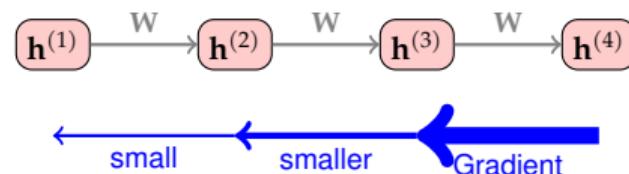
- If the singular values of the weight matrices are small (or  $\tanh$  derivative  $< 1$ ), the product shrinks exponentially.

# The Vanishing Gradient Problem: Intuition

Why does the gradient vanish? Consider the chain rule for 4 steps:

$$\frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(1)}} = \underbrace{\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}}}_{\text{Jacobian}} \times \frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(4)}}$$

- If the singular values of the weight matrices are small (or tanh derivative  $< 1$ ), the product shrinks exponentially.
- **Visual:**

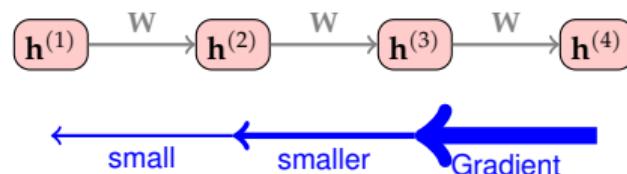


# The Vanishing Gradient Problem: Intuition

Why does the gradient vanish? Consider the chain rule for 4 steps:

$$\frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(1)}} = \underbrace{\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}}}_{\text{Jacobian}} \times \frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(4)}}$$

- If the singular values of the weight matrices are small (or tanh derivative  $< 1$ ), the product shrinks exponentially.
- **Visual:**



- **Result:** The model cannot learn long-range dependencies (e.g., matching “The **tickets**” ... to ... “**tickets**” at the end).

# The Vanishing Gradient Problem

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

# 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:**

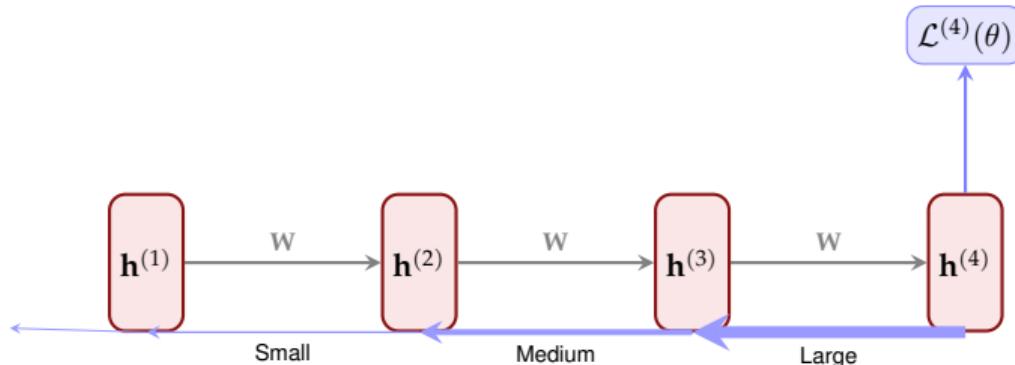
The model stops learning from early inputs.

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).

# The Vanishing Gradient Problem: Intuition



Gradient flow via Chain Rule:

$$\frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(1)}} = \underbrace{\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}}}_{\text{Jacobian}} \times \underbrace{\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}}}_{\text{Jacobian}} \times \frac{\partial \mathcal{L}^{(4)}}{\partial \mathbf{h}^{(4)}}$$

**Problem:** If the weight matrices (or gradients of activation) are small, the gradient signal **shrinks exponentially** as it backpropagates.

# Vanishing Gradient Proof Sketch (Linear Case)

- Recall the recurrence:  $\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1)$ .

# Vanishing Gradient Proof Sketch (Linear Case)

- Recall the recurrence:  $\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1)$ .
- Assume linear activation  $\sigma(x) = x$  (identity function).

# Vanishing Gradient Proof Sketch (Linear Case)

- Recall the recurrence:  $\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1)$ .
- Assume linear activation  $\sigma(x) = x$  (identity function).
- The Jacobian simplifies to the weight matrix itself:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \mathbf{I} \cdot \mathbf{W}_h = \mathbf{W}_h$$

# Vanishing Gradient Proof Sketch (Linear Case)

- Recall the recurrence:  $\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1)$ .
- Assume linear activation  $\sigma(x) = x$  (identity function).
- The Jacobian simplifies to the weight matrix itself:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \mathbf{I} \cdot \mathbf{W}_h = \mathbf{W}_h$$

- The gradient of loss  $\mathcal{L}^{(i)}$  w.r.t. a previous state  $\mathbf{h}^{(j)}$  involves a product of  $l = i - j$  matrices:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(j)}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(i)}} (\mathbf{W}_h)^l$$

# Vanishing Gradient Proof Sketch (Linear Case)

- Recall the recurrence:  $\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b}_1)$ .
- Assume linear activation  $\sigma(x) = x$  (identity function).
- The Jacobian simplifies to the weight matrix itself:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \mathbf{I} \cdot \mathbf{W}_h = \mathbf{W}_h$$

- The gradient of loss  $\mathcal{L}^{(i)}$  w.r.t. a previous state  $\mathbf{h}^{(j)}$  involves a product of  $l = i - j$  matrices:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(j)}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}^{(i)}} (\mathbf{W}_h)^l$$

- If eigenvalues  $\lambda < 1$ , then  $(\mathbf{W}_h)^l \rightarrow 0$  exponentially as  $l$  grows.

# Why is Vanishing Gradient a Problem?

**Task:** “The **tickets** ... [long sequence] ... **tickets**.”



## Consequence:

Weights are updated only w.r.t. **near effects**.

The model cannot learn **long-term dependencies**.

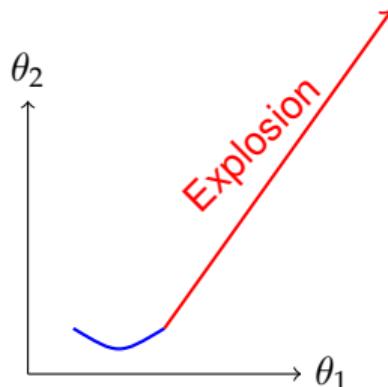
# Exploding Gradients

- Sometimes gradients can become too **big** (e.g., if eigenvalues  $> 1$ ).

# Exploding Gradients

- Sometimes gradients can become too **big** (e.g., if eigenvalues  $> 1$ ).
- **Result:** SGD update step becomes huge.

$$\theta^{new} = \theta^{old} - \alpha \underbrace{\nabla_{\theta} \mathcal{L}(\theta)}_{\text{Huge!}}$$

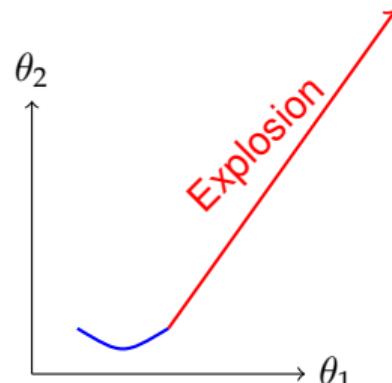


# Exploding Gradients

- Sometimes gradients can become too **big** (e.g., if eigenvalues  $> 1$ ).
- **Result:** SGD update step becomes huge.

$$\theta^{new} = \theta^{old} - \alpha \underbrace{\nabla_{\theta} \mathcal{L}(\theta)}_{\text{Huge!}}$$

- This causes "bad updates": parameters shoot off to infinity (NaN) or weird regions.

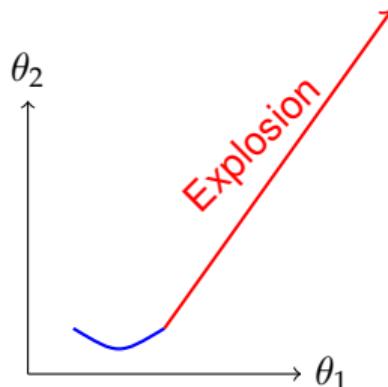


# Exploding Gradients

- Sometimes gradients can become too **big** (e.g., if eigenvalues  $> 1$ ).
- **Result:** SGD update step becomes huge.

$$\theta^{new} = \theta^{old} - \alpha \underbrace{\nabla_{\theta} \mathcal{L}(\theta)}_{\text{Huge!}}$$

- This causes "bad updates": parameters shoot off to infinity (NaN) or weird regions.



Solution: Gradient Clipping

If norm  $\|\hat{g}\| >$  threshold, scale it down:

$$\hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g}$$

# Long Short-Term Memory (LSTM)

- **Problem:** Vanilla RNNs cannot learn long-term dependencies due to vanishing gradients.

# Long Short-Term Memory (LSTM)

- **Problem:** Vanilla RNNs cannot learn long-term dependencies due to vanishing gradients.
- **Solution:** LSTM introduces a **Cell State  $c^{(t)}$**  that acts as a “memory highway”.

# Long Short-Term Memory (LSTM)

- **Problem:** Vanilla RNNs cannot learn long-term dependencies due to vanishing gradients.
- **Solution:** LSTM introduces a **Cell State  $c^{(t)}$**  that acts as a “memory highway”.
- **Key Idea:** Use **gates** to control information flow:
  - **Forget Gate ( $f^{(t)}$ )**: What to remove from memory
  - **Input Gate ( $i^{(t)}$ )**: What new information to add
  - **Output Gate ( $o^{(t)}$ )**: What to output from memory

# Long Short-Term Memory (LSTM)

- **Problem:** Vanilla RNNs cannot learn long-term dependencies due to vanishing gradients.
- **Solution:** LSTM introduces a **Cell State  $c^{(t)}$**  that acts as a “memory highway”.
- **Key Idea:** Use **gates** to control information flow:
  - **Forget Gate ( $f^{(t)}$ ):** What to remove from memory
  - **Input Gate ( $i^{(t)}$ ):** What new information to add
  - **Output Gate ( $o^{(t)}$ ):** What to output from memory
- Gates are computed using sigmoid ( $\sigma$ ), giving values in  $[0, 1]$ :
  - $0 \rightarrow$  “block completely”
  - $1 \rightarrow$  “let everything through”

# LSTM: Mathematical Formulation

**Forget Gate:**  $\mathbf{f}^{(t)} = \sigma(\mathbf{W}_f[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_f)$

**Input Gate:**  $\mathbf{i}^{(t)} = \sigma(\mathbf{W}_i[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_i)$

**Candidate:**  $\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{W}_c[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_c)$

**Cell Update:**  $\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)}$

**Output Gate:**  $\mathbf{o}^{(t)} = \sigma(\mathbf{W}_o[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_o)$

**Hidden State:**  $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)})$

**Key Insight:** Cell state  $\mathbf{c}^{(t)}$  flows with only element-wise operations, creating a **gradient highway** through time!

# LSTM: How Gates Work

## 1. Forget Gate $f^{(t)}$

- Decides what to discard from  $c^{(t-1)}$
- If  $f_i^{(t)} \approx 0$ : forget dimension  $i$
- If  $f_i^{(t)} \approx 1$ : keep dimension  $i$

## 2. Input Gate $i^{(t)}$

- Decides what new info to add
- Works with candidate  $\tilde{c}^{(t)}$
- Filters what gets written to memory

## 3. Cell State Update

$$c^{(t)} = \underbrace{f^{(t)} \odot c^{(t-1)}}_{\text{forget old}} + \underbrace{i^{(t)} \odot \tilde{c}^{(t)}}_{\text{add new}}$$

## 4. Output Gate $o^{(t)}$

- Filters cell state for output
- Hidden state:  $h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$

# LSTM vs. Vanilla RNN

Feature	Vanilla RNN	LSTM
<b>State</b>	Hidden $\mathbf{h}^{(t)}$	Hidden $\mathbf{h}^{(t)}$ + Cell $\mathbf{c}^{(t)}$
<b>Gates</b>	None	Forget, Input, Output
<b>Gradient Flow</b>	Multiplicative (decays)	Additive highway
<b>Long Dependencies</b>	Poor	Good
<b>Parameters</b>	$\sim 3d^2$	$\sim 12d^2$ ( $4\times$ more)
<b>Training Speed</b>	Fast	Slower

**Trade-off:** LSTM has more parameters and is slower, but much better at capturing long-range dependencies.

# Gated Recurrent Units (GRU): Intuition

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

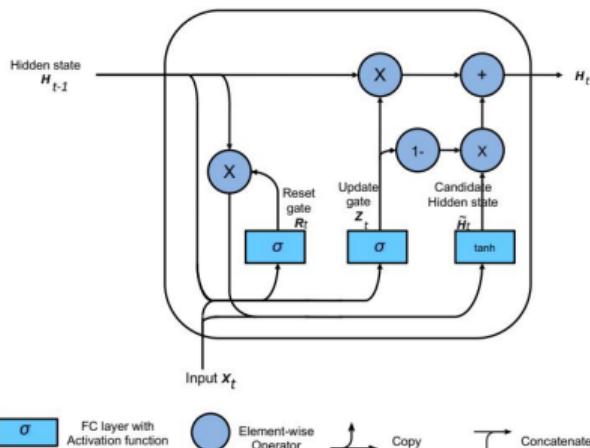


Figure: The GRU adds gates to control information flow.

# 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?”

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

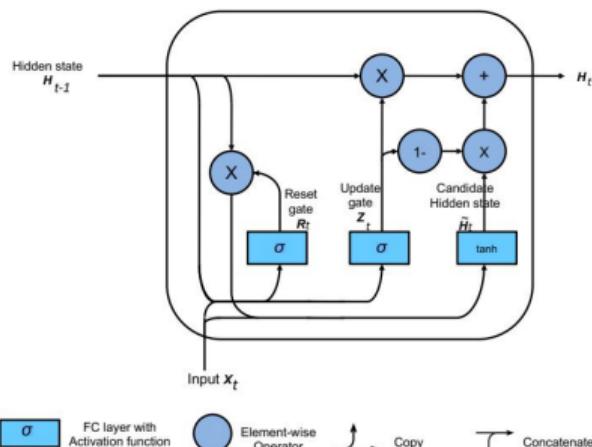


Figure: The GRU adds gates to control information flow.

# 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  $\mathbf{z}^{(t)} \approx 0$ , then  $\mathbf{h}^{(t)} \approx \mathbf{h}^{(t-1)}$ . The gradient passes through unchanged!

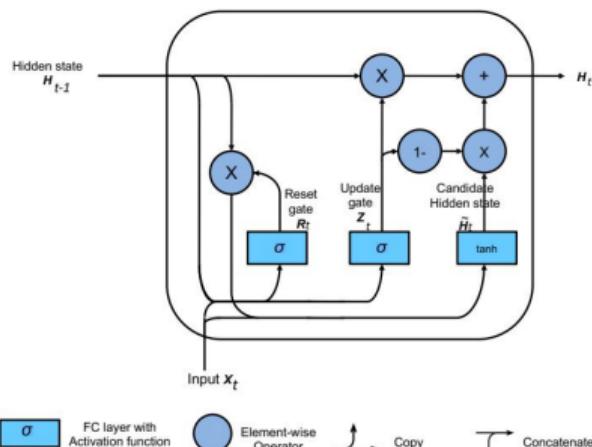


Figure: The GRU adds gates to control information flow.

# 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  $\mathbf{z}^{(t)} \approx 0$ , 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.

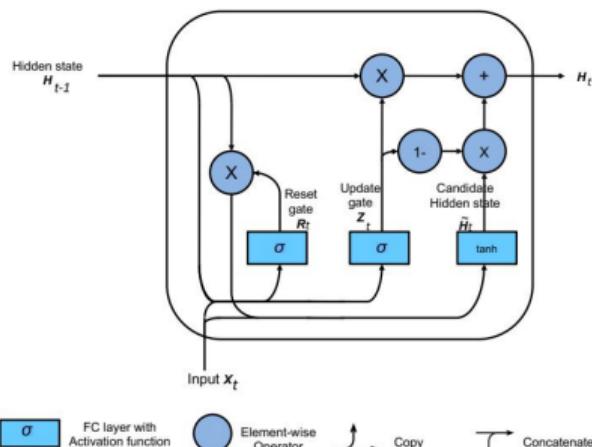


Figure: The GRU adds gates to control information flow.

# 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  $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(T)}$  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  $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(T)}$  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  $\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(T)}$  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

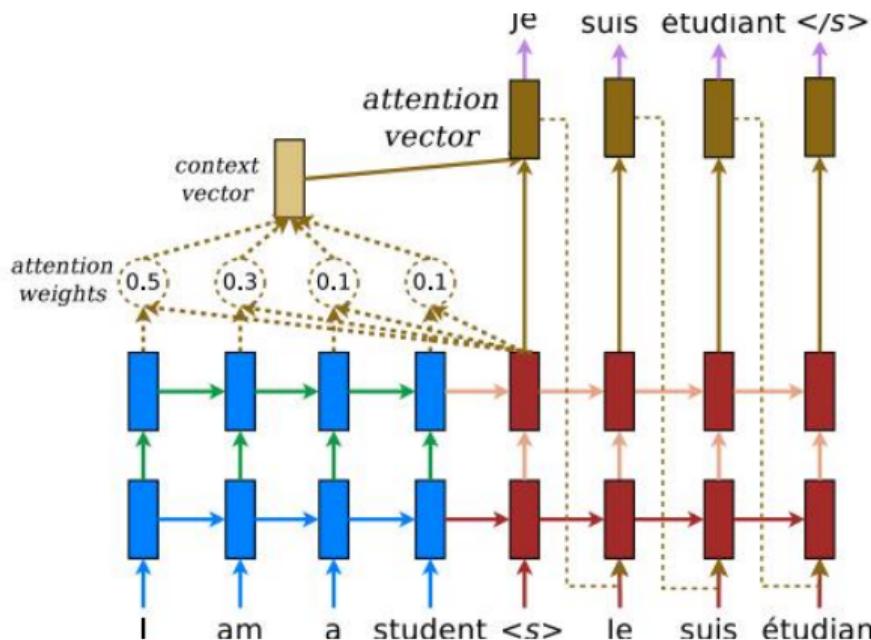


Figure: The decoder (blue) attends to relevant encoder states (red) to generate the next word.

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

# The Transformer: Attention Is All You Need

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

# The Transformer: Attention Is All You Need

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

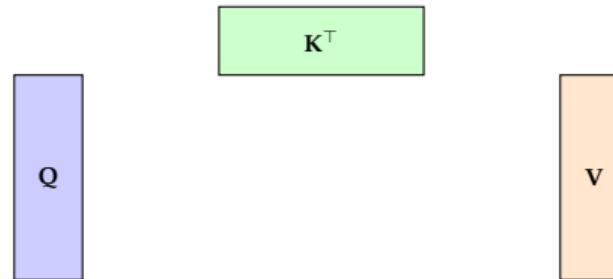
# The Transformer: Attention Is All You Need

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

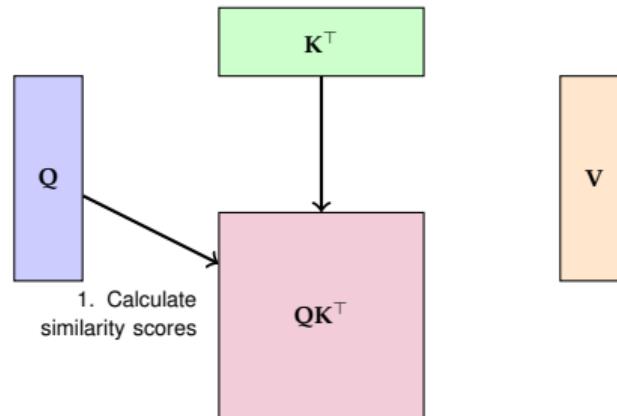
# The Transformer: Attention Is All You Need

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

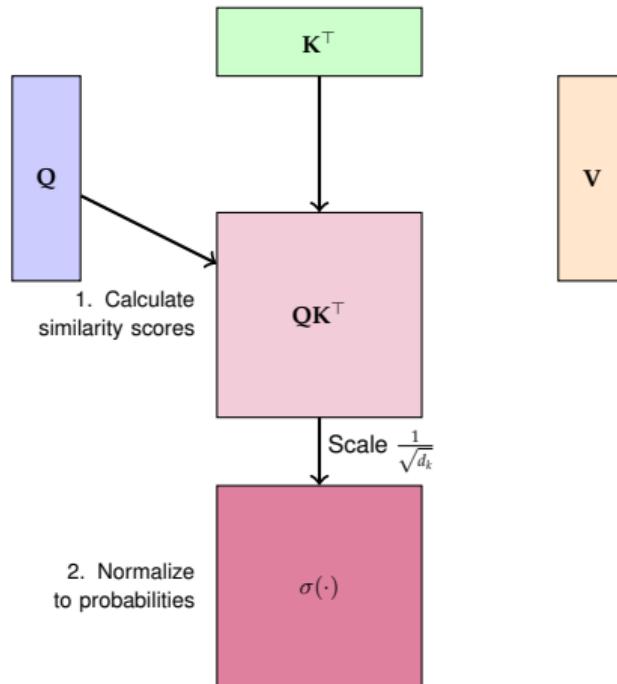
# Step-by-Step: Self-Attention Mechanism



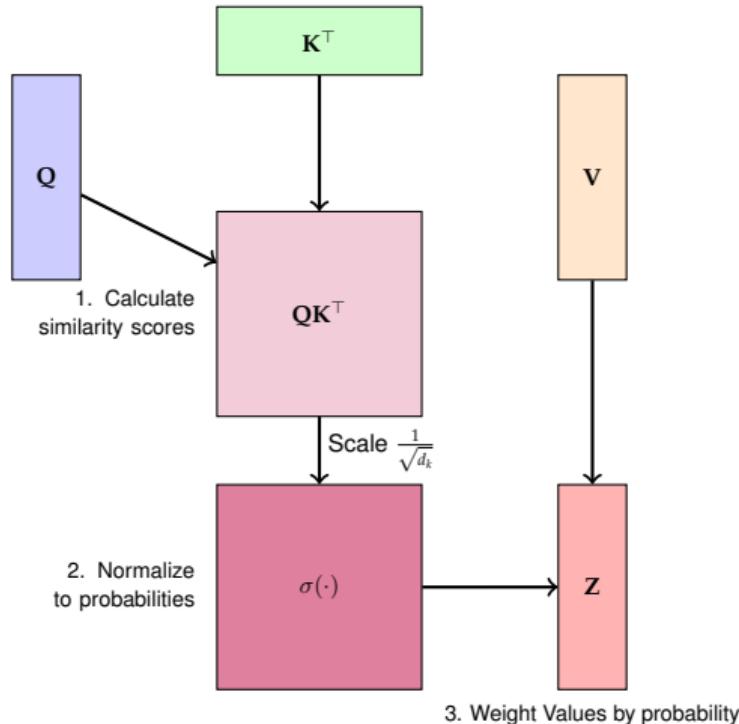
# Step-by-Step: Self-Attention Mechanism



# Step-by-Step: Self-Attention Mechanism



# Step-by-Step: Self-Attention Mechanism



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

# Multi-Head Attention

- A single attention layer might focus only on syntax.

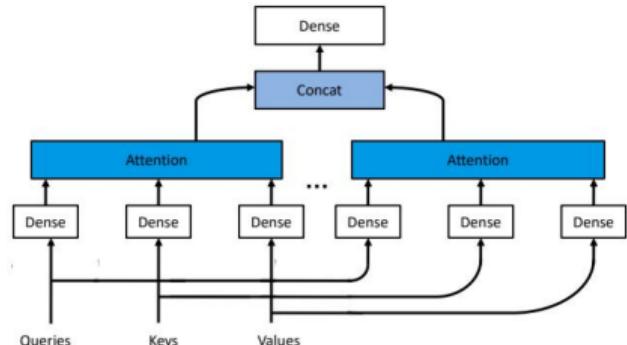


Figure: Multi-Head Attention schematic.

# Multi-Head Attention

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

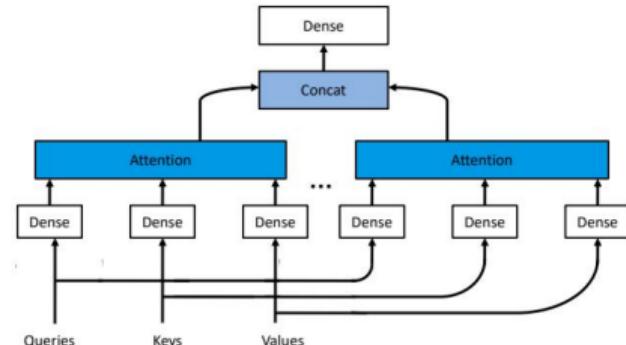


Figure: Multi-Head Attention schematic.

# 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.

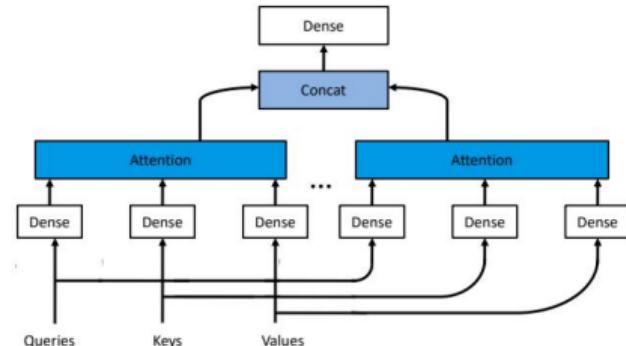


Figure: Multi-Head Attention schematic.

# 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:

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

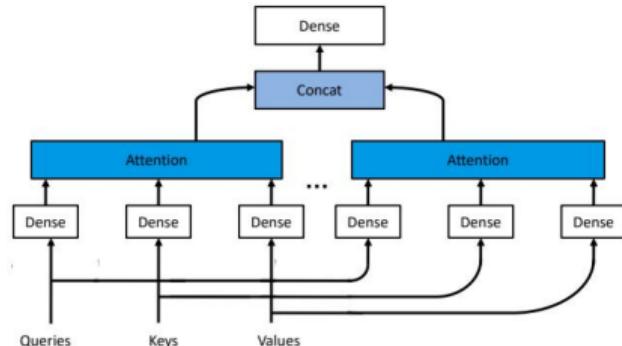


Figure: Multi-Head Attention schematic.

# 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.

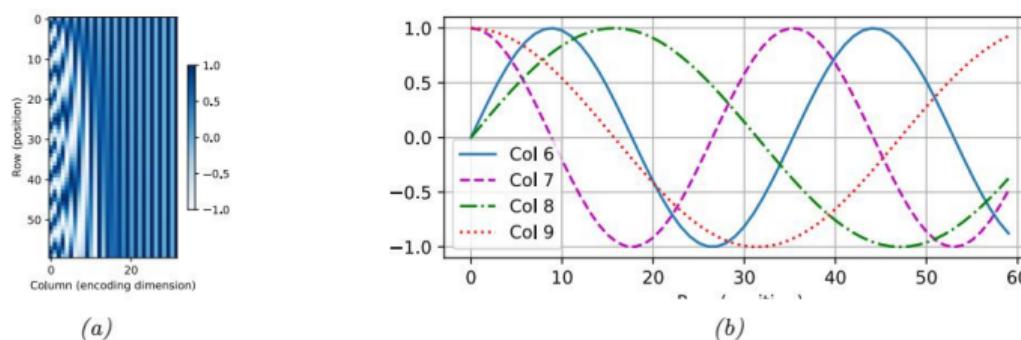


Figure: Sinusoidal Positional Encodings.

# 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.

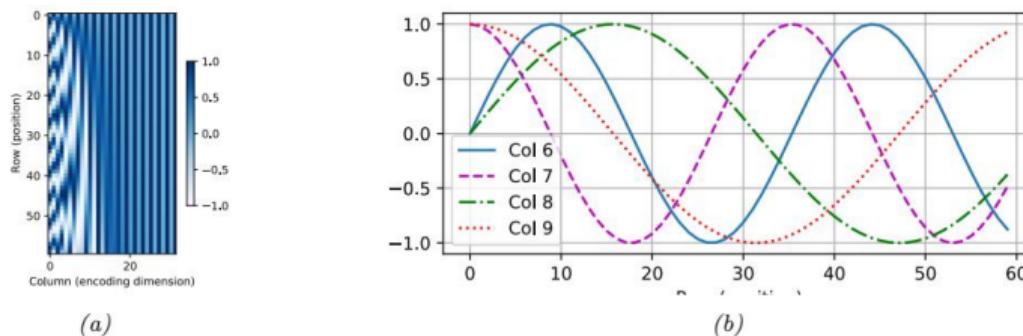


Figure: Sinusoidal Positional Encodings.

# 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:

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

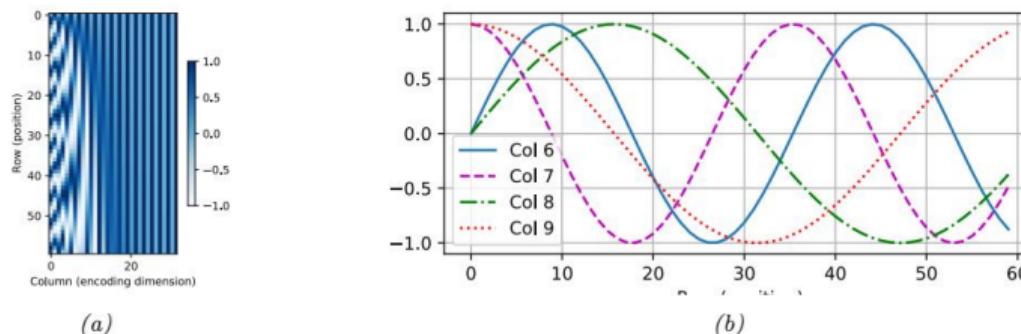


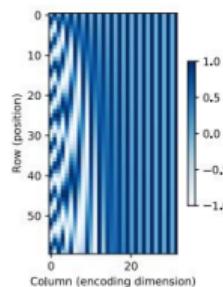
Figure: Sinusoidal Positional Encodings.

# Positional Encoding: Adding Order

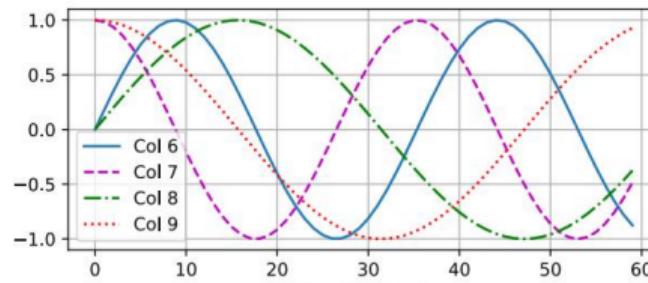
- **Problem:** Self-attention is permutation invariant. “The cat bit the dog” looks the same as “The dog bit the cat” to the math above.
- **Solution:** Inject information about position.
- Add a vector  $PE$  to the input embeddings:

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

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



(a)

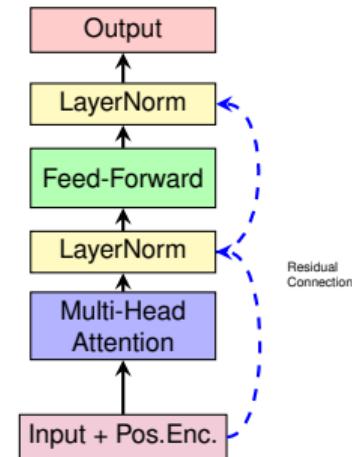


(b)

Figure: Sinusoidal Positional Encodings.

# Transformer Architecture: The Full Picture

Each Transformer Layer Contains:

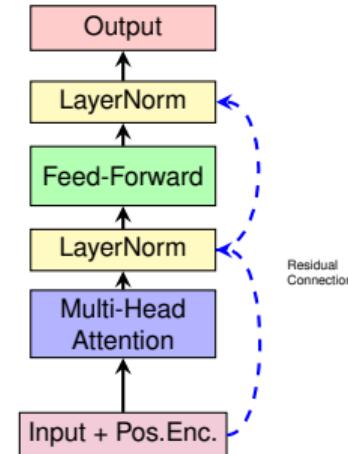


# Transformer Architecture: The Full Picture

Each Transformer Layer Contains:

- **Multi-Head Self-Attention**

- Captures relationships between all tokens
- Parallel processing (not sequential!)



# Transformer Architecture: The Full Picture

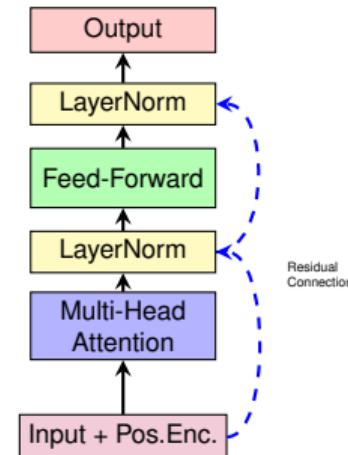
Each Transformer Layer Contains:

- **Multi-Head Self-Attention**

- Captures relationships between all tokens
- Parallel processing (not sequential!)

- **Feed-Forward Network**

- Applied to each position independently
- Two linear layers with ReLU



# Transformer Architecture: The Full Picture

Each Transformer Layer Contains:

- **Multi-Head Self-Attention**

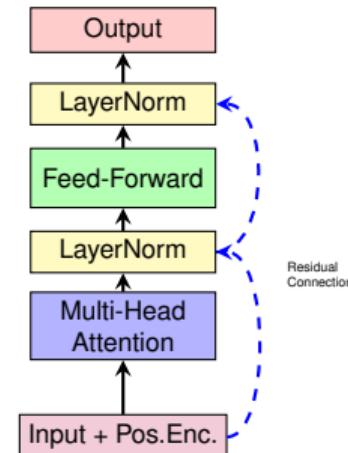
- Captures relationships between all tokens
- Parallel processing (not sequential!)

- **Feed-Forward Network**

- Applied to each position independently
- Two linear layers with ReLU

- **Residual Connections**

- $\text{Output} = \text{Layer}(x) + x$
- Helps gradient flow



# Transformer Architecture: The Full Picture

Each Transformer Layer Contains:

- **Multi-Head Self-Attention**

- Captures relationships between all tokens
- Parallel processing (not sequential!)

- **Feed-Forward Network**

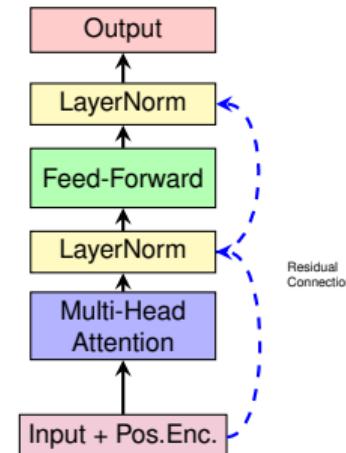
- Applied to each position independently
- Two linear layers with ReLU

- **Residual Connections**

- $\text{Output} = \text{Layer}(x) + x$
- Helps gradient flow

- **Layer Normalization**

- Stabilizes training
- Normalizes across features



# Feed-Forward Network & Layer Normalization

## Feed-Forward Network (FFN):

- Applied to each position  $i$  independently:

$$\text{FFN}(\mathbf{x}_i) = \max(0, \mathbf{x}_i \mathbf{W}_1 + \mathbf{b}_1) \mathbf{W}_2 + \mathbf{b}_2$$

- Typical dimensions:  $d_{\text{model}} = 512$ ,  $d_{\text{ff}} = 2048$
- Same weights shared across all positions (like 1D convolution)

## Layer Normalization:

- Normalize activations across features for each sample:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

- $\mu, \sigma$ : mean and std computed over feature dimension
- $\gamma, \beta$ : learned affine parameters
- Stabilizes training and allows higher learning rates

# Why Residual Connections Matter

Without Residuals:

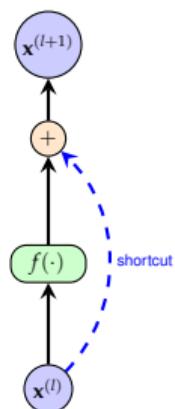
$$\mathbf{x}^{(l+1)} = f(\mathbf{x}^{(l)})$$

- Information must flow through  $f$
- Gradient can vanish through many layers
- Harder to train deep models

With Residuals:

$$\mathbf{x}^{(l+1)} = f(\mathbf{x}^{(l)}) + \mathbf{x}^{(l)}$$

- Identity shortcut path
- Gradient flows directly backwards
- Model can learn incremental changes



# Modern Transformer Variants

- **Original Transformer (2017):** Encoder-Decoder for machine translation

# Modern Transformer Variants

- **Original Transformer (2017)**: Encoder-Decoder for machine translation
- **BERT (2018) - Encoder-Only:**
  - Bidirectional encoding
  - Pre-trained with masked language modeling
  - Best for: classification, NER, question answering

# Modern Transformer Variants

- **Original Transformer (2017)**: Encoder-Decoder for machine translation
- **BERT (2018) - Encoder-Only**:
  - Bidirectional encoding
  - Pre-trained with masked language modeling
  - Best for: classification, NER, question answering
- **GPT (2018+) - Decoder-Only**:
  - Autoregressive generation
  - Pre-trained with next-token prediction
  - Best for: text generation, completion
  - GPT-3, GPT-4: scaled to billions/trillions of parameters

# Modern Transformer Variants

- **Original Transformer (2017)**: Encoder-Decoder for machine translation
- **BERT (2018) - Encoder-Only**:
  - Bidirectional encoding
  - Pre-trained with masked language modeling
  - Best for: classification, NER, question answering
- **GPT (2018+) - Decoder-Only**:
  - Autoregressive generation
  - Pre-trained with next-token prediction
  - Best for: text generation, completion
  - GPT-3, GPT-4: scaled to billions/trillions of parameters
- **Other Variants**:
  - T5, BART (encoder-decoder)
  - Vision Transformer (ViT) for images
  - Efficient Transformers (Lformer, Performer) to reduce  $O(n^2)$  complexity

# Summary: RNN vs Transformer

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

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

# When to Use What? A Practical Guide

Model	Best For	Avoid When
<b>Vanilla RNN</b>	Short sequences, simple patterns, resource-constrained	Long-term dependencies, complex tasks
<b>LSTM/GRU</b>	Long sequences with dependencies, time series, moderate data	Very long sequences ( $>1000$ ), large datasets
<b>Transformer</b>	Long-range dependencies, large datasets, parallel training	Small datasets, real-time/streaming

## Key Considerations:

- **Dataset size:** Transformers need more data to train effectively
- **Sequence length:** Transformers  $O(n^2)$  vs. RNN  $O(n)$
- **Parallelization:** Transformers can leverage GPUs better
- **Interpretability:** Attention weights provide some interpretability

# Key Takeaways

- ① **Sequential Data is Everywhere:** RNNs, LSTMs, and Transformers are fundamental for language, time series, video, etc.
- ② **The Gradient Problem:** Vanilla RNNs suffer from vanishing/exploding gradients. LSTMs/GRUs use gates to solve this.
- ③ **Attention is Powerful:** Attention mechanisms allow models to focus on relevant inputs dynamically.
- ④ **Transformers Dominate Modern NLP:** Self-attention + parallelization = state-of-the-art performance (BERT, GPT, T5).
- ⑤ **Historical Progression:**

RNN → LSTM/GRU → Attention → Transformer

Each step addressed limitations of the previous approach.

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)

## Future Directions:

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length

## Future Directions:

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training

## Future Directions:

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training
- **Lack of Built-in Inductive Biases:** No inherent notion of order or locality

## Future Directions:

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training
- **Lack of Built-in Inductive Biases:** No inherent notion of order or locality

## Future Directions:

- **Efficient Transformers:** Linear attention, sparse attention, etc.

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training
- **Lack of Built-in Inductive Biases:** No inherent notion of order or locality

## Future Directions:

- **Efficient Transformers:** Linear attention, sparse attention, etc.
- **Very Long Sequences:** Models that can handle 100K+ tokens efficiently

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training
- **Lack of Built-in Inductive Biases:** No inherent notion of order or locality

## Future Directions:

- **Efficient Transformers:** Linear attention, sparse attention, etc.
- **Very Long Sequences:** Models that can handle 100K+ tokens efficiently
- **Multi-modal Models:** Unified models for text, images, audio (e.g., CLIP, Flamingo)

# Limitations & Future Directions

## Current Limitations:

- **Computational Cost:** Transformers require massive compute (GPT-3: ~\$4M to train)
- **Long Sequences:**  $O(n^2)$  self-attention limits sequence length
- **Data Hunger:** Need large datasets for effective pre-training
- **Lack of Built-in Inductive Biases:** No inherent notion of order or locality

## Future Directions:

- **Efficient Transformers:** Linear attention, sparse attention, etc.
- **Very Long Sequences:** Models that can handle 100K+ tokens efficiently
- **Multi-modal Models:** Unified models for text, images, audio (e.g., CLIP, Flamingo)
- **Better Sample Efficiency:** Learning with less data