

# **CS 4644-DL / 7643-A: LECTURE 14**

## **DANFEI XU**

Topics:

- Deep Learning Hardware and Software

# Administrative

- Time to work on the project
  - Read papers in more details, implement baselines, process data, verify hypotheses, etc.
- We will release the milestone guideline soon
- Start on PS3/HW3 if you haven't
  - Coding: If you passed individual testing cases but are failing end-to-end testing, double check your Multi-Headed Attention. The unit test doesn't catch all errors.
  - DO NOT MODIFY YOUR TEST CODE

# Today

- Finishing Attention, Transformers
- Deep learning hardware
  - CPU, GPU
- Deep learning software
  - PyTorch and TensorFlow
  - Static and Dynamic computation graphs

# Attention Layer

## Inputs:

Query vectors:  $\mathbf{Q}$  (Shape:  $N_Q \times D_Q$ )

Input vectors:  $\mathbf{X}$  (Shape:  $N_X \times D_X$ )

Key matrix:  $\mathbf{W}_K$  (Shape:  $D_X \times D_Q$ )

Value matrix:  $\mathbf{W}_V$  (Shape:  $D_X \times D_V$ )

## Computation:

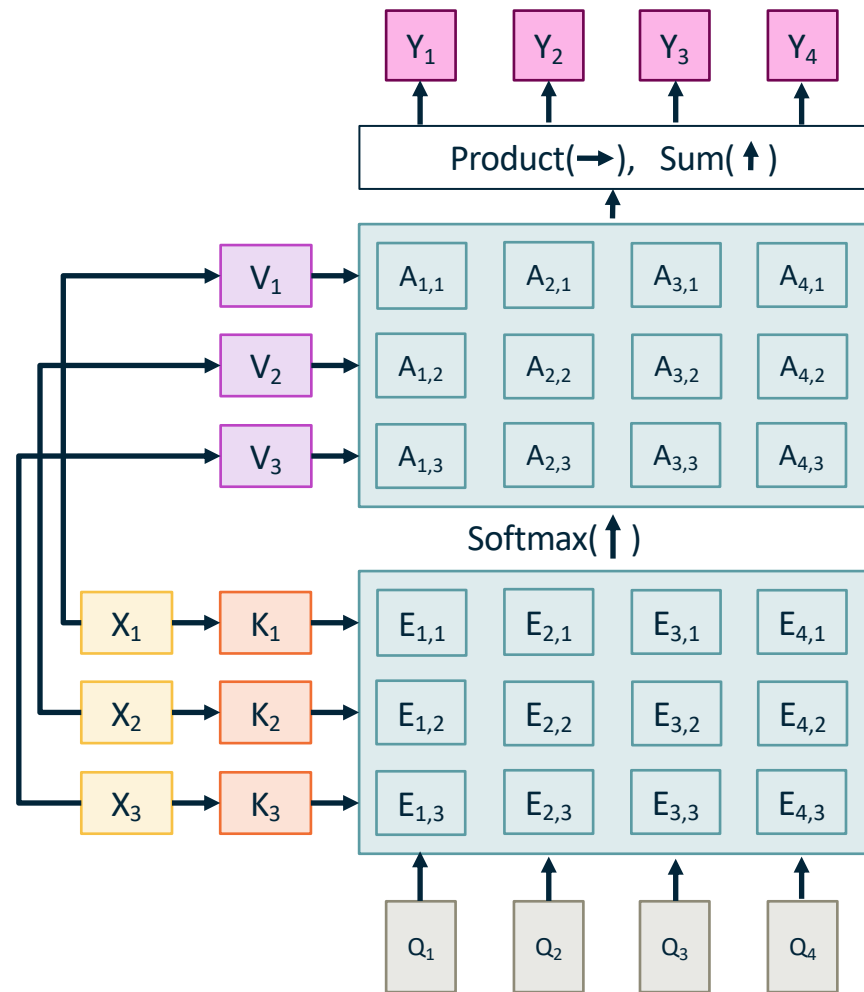
Key vectors:  $\mathbf{K} = \mathbf{XW}_K$  (Shape:  $N_X \times D_Q$ )

Value vectors:  $\mathbf{V} = \mathbf{XW}_V$  (Shape:  $N_X \times D_V$ )

Similarities:  $\mathbf{E} = \mathbf{QK}^T$  (Shape:  $N_Q \times N_X$ )  $E_{i,j} = \mathbf{Q}_i \cdot \mathbf{K}_j / \text{sqrt}(D_Q)$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  (Shape:  $N_Q \times N_X$ )

→ Output vectors:  $\mathbf{Y} = \mathbf{AV}$  (Shape:  $N_Q \times D_V$ )  $Y_i = \sum_j A_{i,j} \mathbf{V}_j$



# Attention Layer

## Inputs:

**Query vectors:**  $\mathbf{Q}$  (Shape:  $N_Q \times D_Q$ )

**Input vectors:**  $\mathbf{X}$  (Shape:  $N_X \times D_X$ )

**Key matrix:**  $\mathbf{W}_K$  (Shape:  $D_X \times D_Q$ )

**Value matrix:**  $\mathbf{W}_V$  (Shape:  $D_X \times D_V$ )

## Computation:

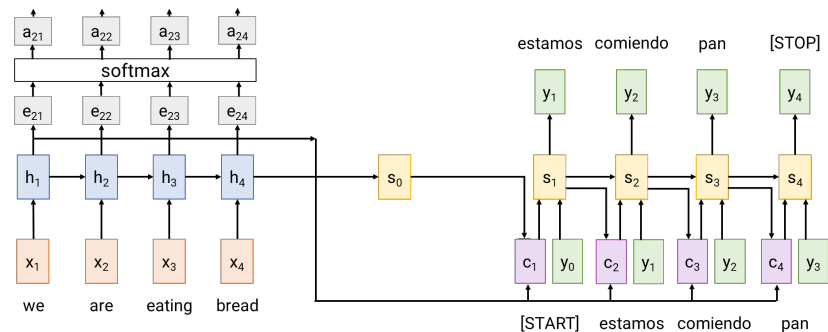
**Key vectors:**  $\mathbf{K} = \mathbf{XW}_K$  (Shape:  $N_X \times D_Q$ )

**Value vectors:**  $\mathbf{V} = \mathbf{XW}_V$  (Shape:  $N_X \times D_V$ )

**Similarities:**  $\mathbf{E} = \mathbf{QK}^T$  (Shape:  $N_Q \times N_X$ )  $E_{i,j} = \mathbf{Q}_i \cdot \mathbf{K}_j / \text{sqrt}(D_Q)$

**Attention weights:**  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  (Shape:  $N_Q \times N_X$ )

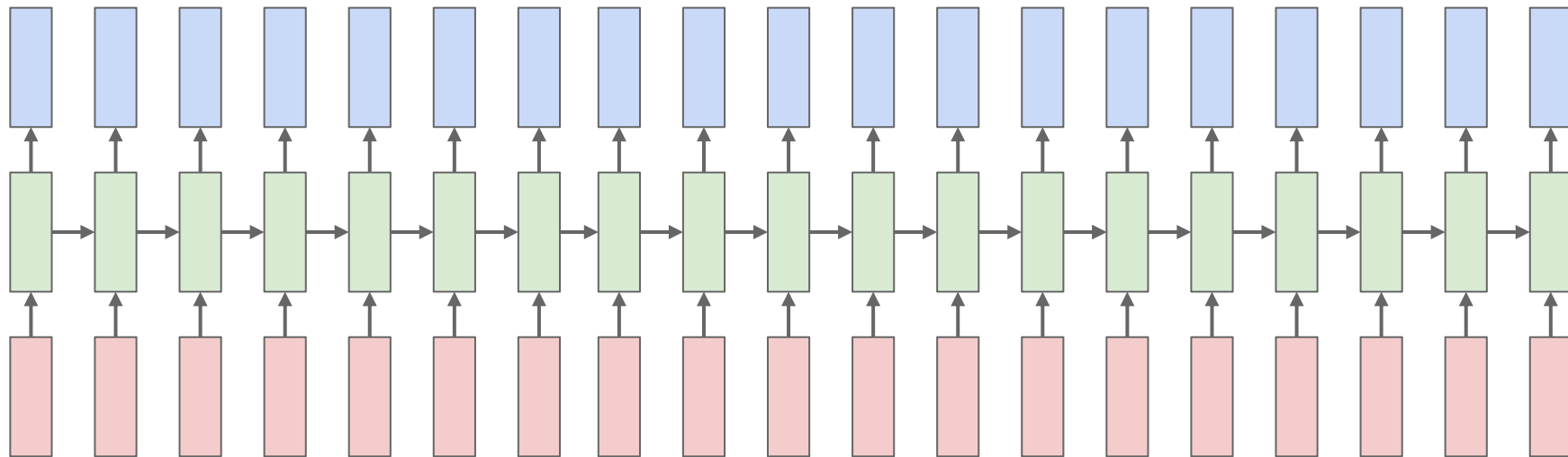
**Output vectors:**  $\mathbf{Y} = \mathbf{AV}$  (Shape:  $N_Q \times D_V$ )  $Y_i = \sum_j A_{i,j} \mathbf{V}_j$



Attention seems to be really powerful ...

Do we still need RNN?

# RNN is bad at encoding long-range relationships!



Recurrent update can easily “forget” information

# Attention Layer

## Inputs:

Query vectors:  $\mathbf{Q}$  (Shape:  $N_Q \times D_Q$ )

Input vectors:  $\mathbf{X}$  (Shape:  $N_X \times D_X$ )

Key matrix:  $\mathbf{W}_K$  (Shape:  $D_X \times D_Q$ )

Value matrix:  $\mathbf{W}_V$  (Shape:  $D_X \times D_V$ )

## Computation:

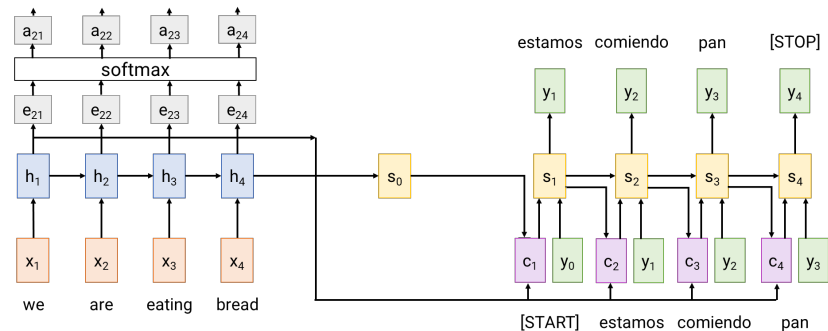
Key vectors:  $\mathbf{K} = \mathbf{XW}_K$  (Shape:  $N_X \times D_Q$ )

Value vectors:  $\mathbf{V} = \mathbf{XW}_V$  (Shape:  $N_X \times D_V$ )

Similarities:  $\mathbf{E} = \mathbf{QK}^T$  (Shape:  $N_Q \times N_X$ )  $E_{i,j} = \mathbf{Q}_i \cdot \mathbf{K}_j / \text{sqrt}(D_Q)$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  (Shape:  $N_Q \times N_X$ )

Output vectors:  $\mathbf{Y} = \mathbf{AV}$  (Shape:  $N_Q \times D_V$ )  $Y_i = \sum_j A_{i,j} \mathbf{V}_j$



Attention seems to be really powerful ...  
Do we still need RNN?

Can we use **only attention layers** to encode an entire sequence?

---

# Attention Is All You Need

---

**Ashish Vaswani\***  
Google Brain  
avaswani@google.com

**Noam Shazeer\***  
Google Brain  
noam@google.com

**Niki Parmar\***  
Google Research  
nikip@google.com

**Jakob Uszkoreit\***  
Google Research  
usz@google.com

**Llion Jones\***  
Google Research  
llion@google.com

**Aidan N. Gomez\* †**  
University of Toronto  
aidan@cs.toronto.edu

**Łukasz Kaiser\***  
Google Brain  
lukaszkaizer@google.com

**Illia Polosukhin\* ‡**  
illia.polosukhin@gmail.com

“The Transformer Paper”



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Query vectors:**  $\mathbf{Q}$  (Shape:  $N_Q \times D_Q$ )

**Input vectors:**  $\mathbf{X}$  (Shape:  $N_X \times D_X$ )

**Key matrix:**  $\mathbf{W}_K$  (Shape:  $D_X \times D_Q$ )

**Value matrix:**  $\mathbf{W}_V$  (Shape:  $D_X \times D_V$ )

## Computation:

**Query vectors:**  $\mathbf{Q} = \mathbf{XW}_Q$

**Key vectors:**  $\mathbf{K} = \mathbf{XW}_K$  (Shape:  $N_X \times D_Q$ )

**Value vectors:**  $\mathbf{V} = \mathbf{XW}_V$  (Shape:  $N_X \times D_V$ )

**Similarities:**  $\mathbf{E} = \mathbf{QK}^T$  (Shape:  $N_X \times N_X$ )  $E_{i,j} = \mathbf{Q}_i \cdot \mathbf{K}_j / \text{sqrt}(D_Q)$

**Attention weights:**  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  (Shape:  $N_X \times N_X$ )

**Output vectors:**  $\mathbf{Y} = \mathbf{AV}$  (Shape:  $N_X \times D_V$ )  $Y_i = \sum_j A_{i,j} \mathbf{V}_j$

Still need to somehow represent inter-token connection in the input sequence.

Goal: encode the input sequence with only attention, without a recurrent network.

$X_1$

$X_2$

$X_3$

# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Query vectors:**  $\mathbf{Q}$  (Shape:  $N_x \times D_Q$ )

**Input vectors:**  $\mathbf{X}$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $\mathbf{W}_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $\mathbf{W}_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $\mathbf{W}_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $\mathbf{Q} = \mathbf{XW}_Q$

**Key vectors:**  $\mathbf{K} = \mathbf{XW}_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $\mathbf{V} = \mathbf{XW}_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $\mathbf{E} = \mathbf{QK}^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = \mathbf{Q}_i \cdot \mathbf{K}_j / \text{sqrt}(D_Q)$

**Attention weights:**  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $\mathbf{Y} = \mathbf{AV}$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} \mathbf{V}_j$

Goal: encode the input sequence with only attention, without a recurrent network.

Encoding only -> no external queries

Use each element to query other elements

$X_1$

$X_2$

$X_3$

# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

→ **Query vectors:**  $Q = XW_Q$

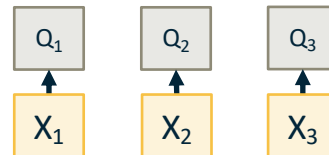
**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

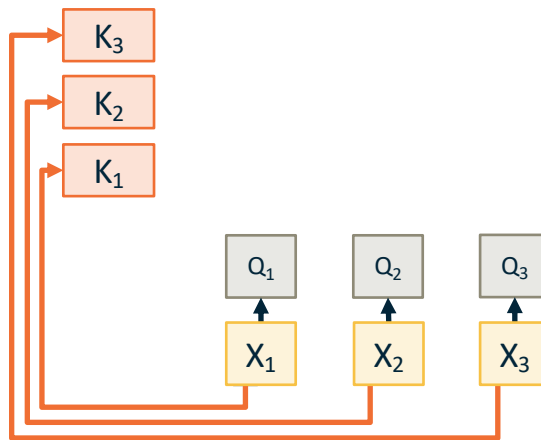
→ **Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

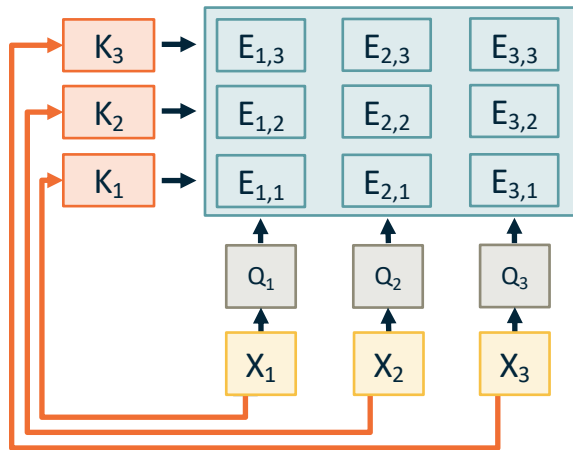
**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

→ **Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

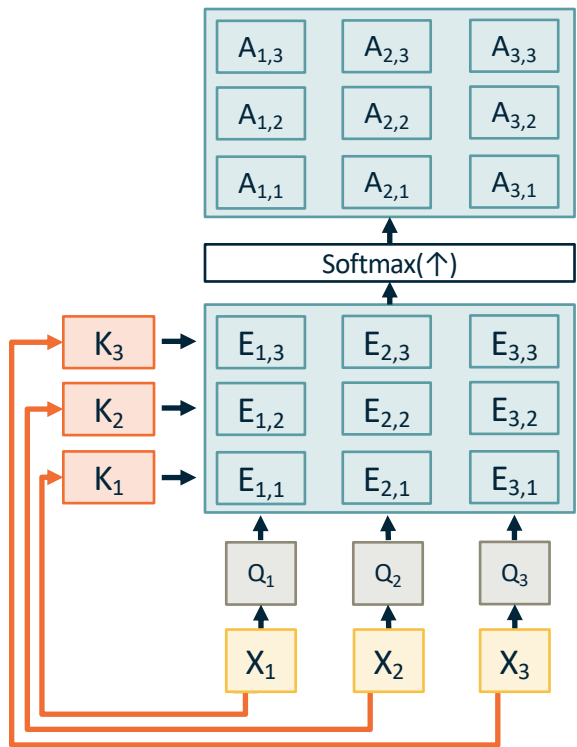
**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

→ **Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

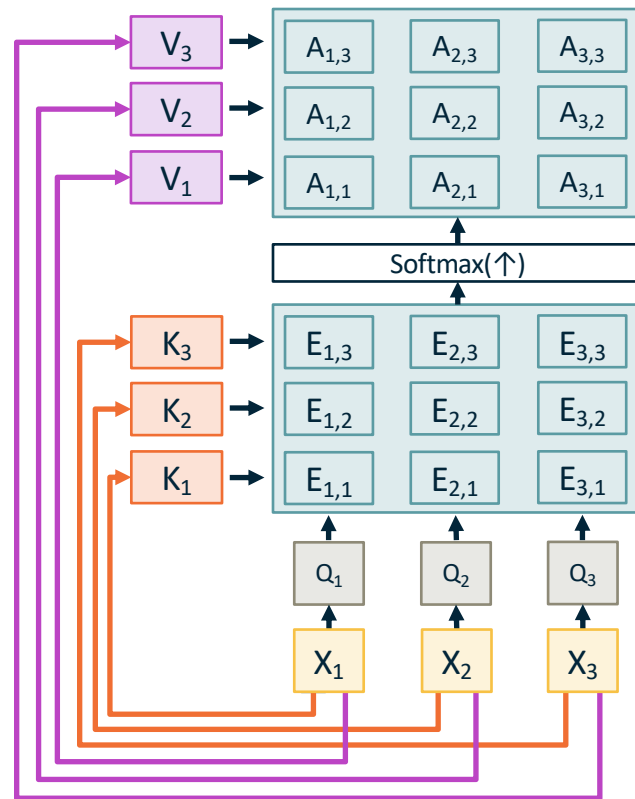
→ **Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Q,K,V are all generated from X!



# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

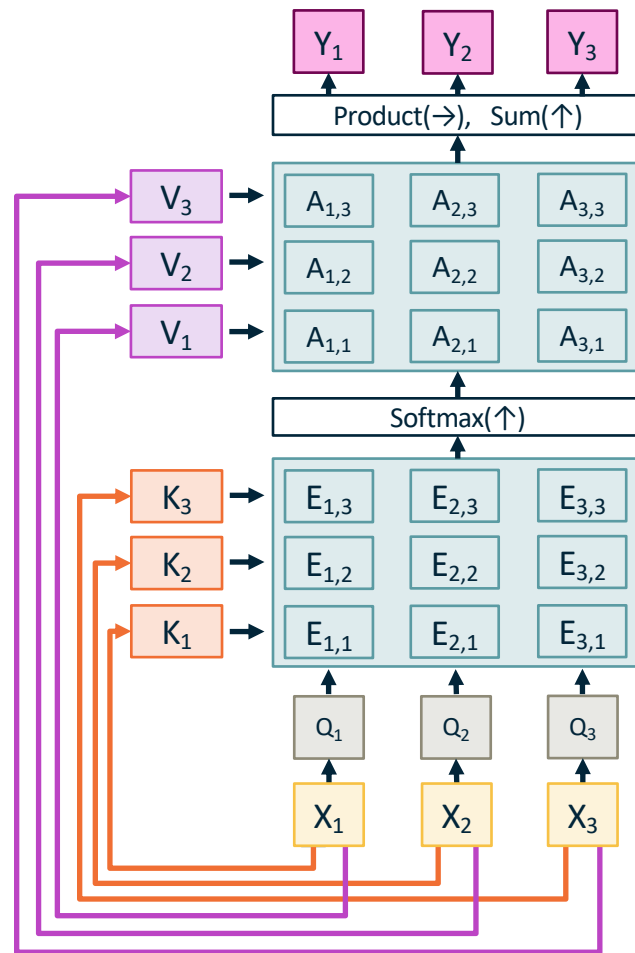
**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Q,K,V are all generated from X!





# Self-Attention Layer

Sequence encode -> use each input element as query!

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

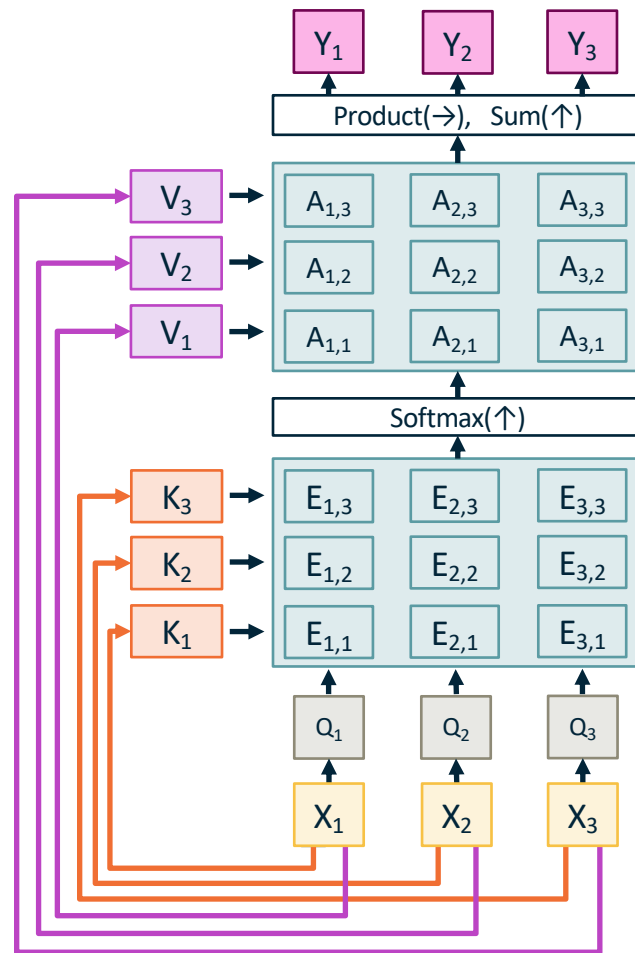
**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Q: Can we use self-attention to encode an input with specific sequential ordering?

Q,K,V are all generated from X!



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

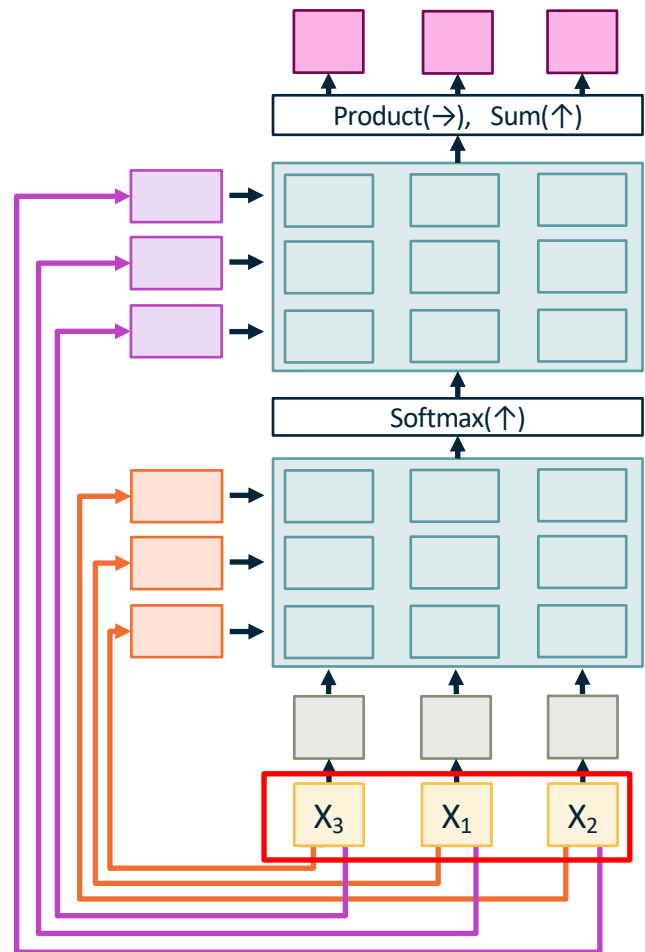
Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_X \times D_X$ )

Key matrix:  $W_K$  (Shape:  $D_X \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_X \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_X \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_X \times D_Q$ )

Value Vectors:  $V = XW_V$  (Shape:  $N_X \times D_V$ )

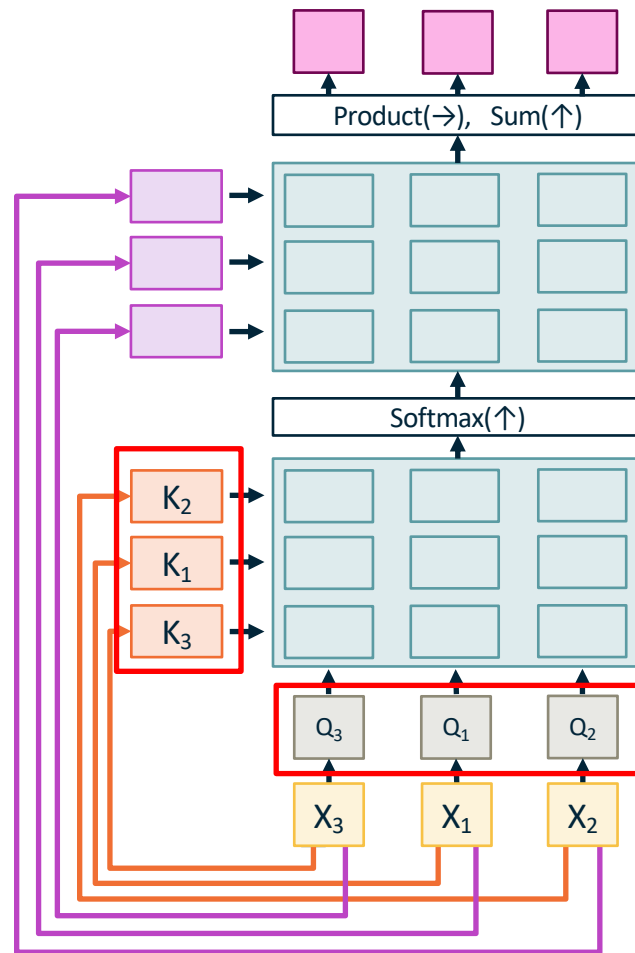
Similarities:  $E = QK^T$  (Shape:  $N_X \times N_X$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_X \times N_X$ )

Output vectors:  $Y = AV$  (Shape:  $N_X \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Queries and Keys will be  
the same, but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

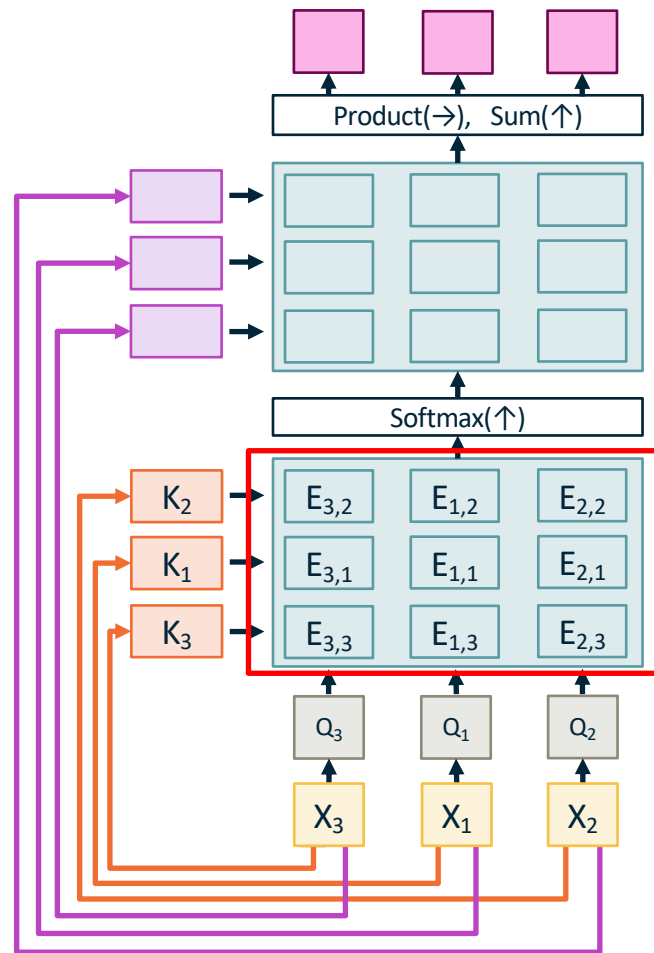
Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Similarities will be the  
same, but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

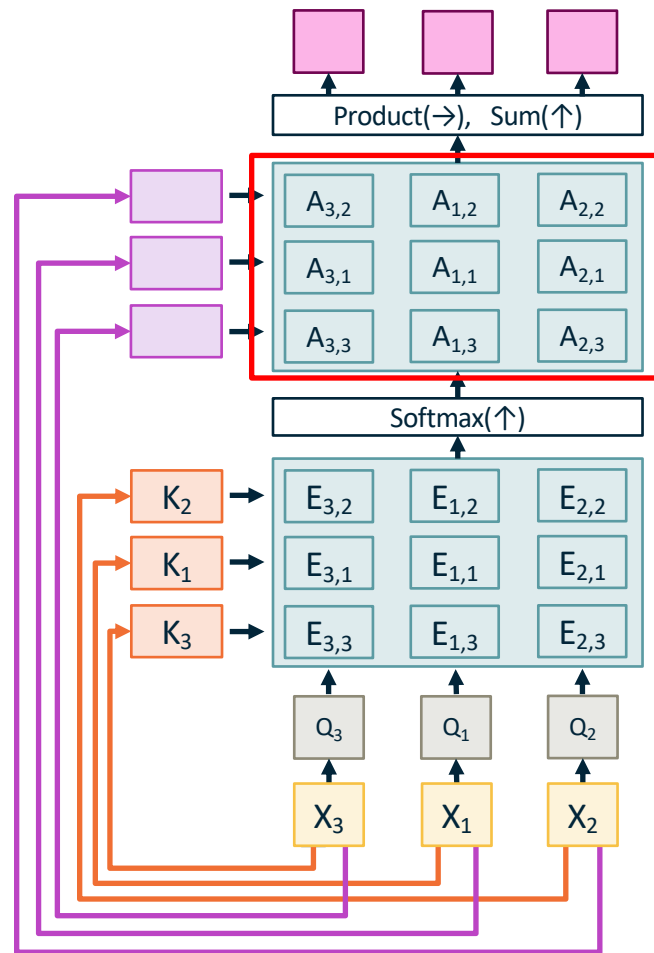
Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Attention weights will be  
the same, but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

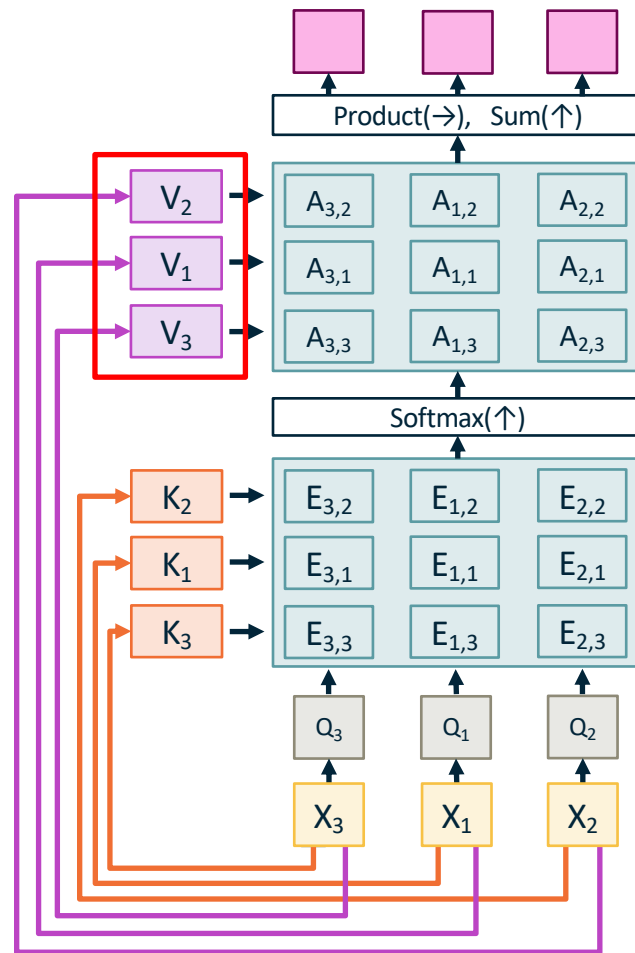
Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Values will be the  
same, but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

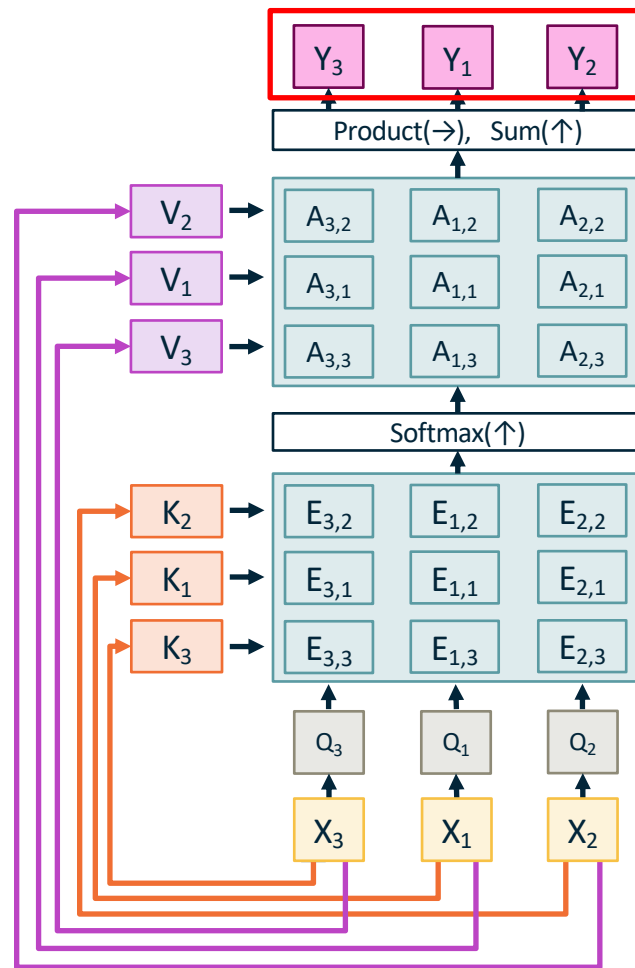
Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Outputs will be the  
same, but permuted



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

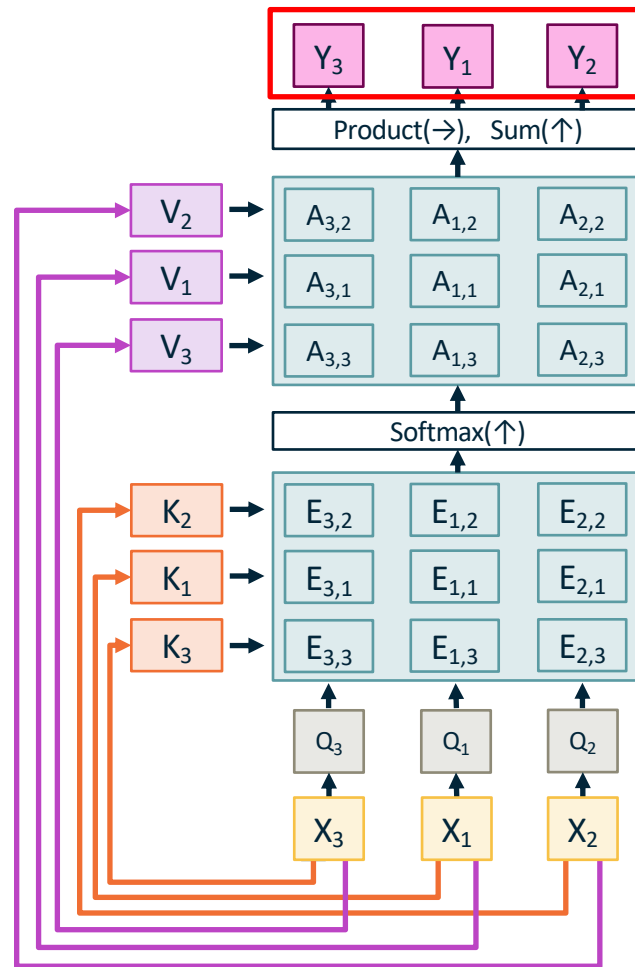
**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting**  
the input vectors:

Outputs will be the  
same, but permuted

Self-attention layer is  
**Permutation Equivariant**  
 $f(s(x)) = s(f(x))$





# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

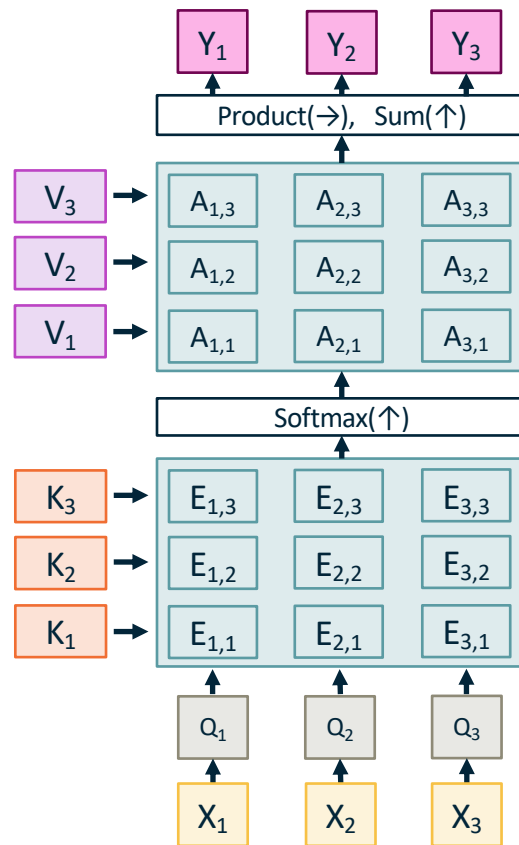
**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Self attention doesn't "know" the order of the vectors it is processing! Not good for sequence encoding.



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

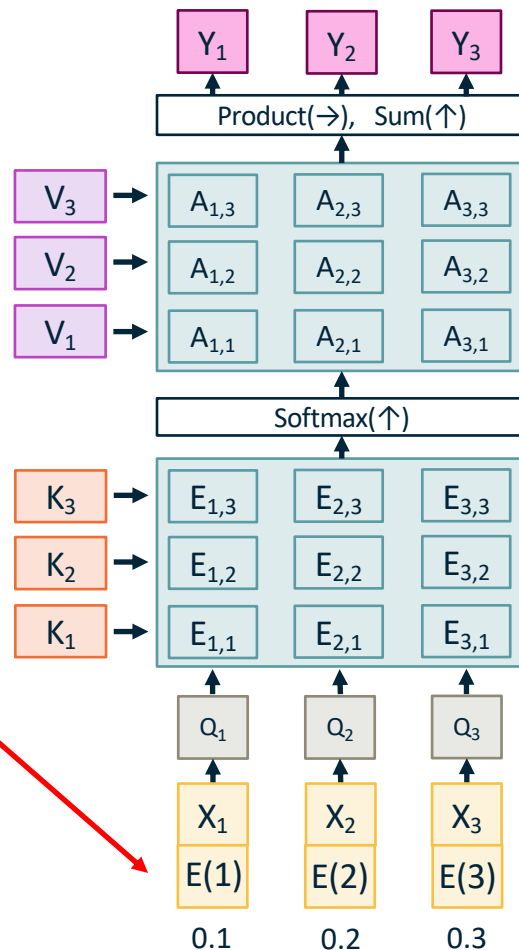
**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

In order to make processing position-aware, concatenate input with **positional encoding**  $E$

$E(i)$  encodes the position of the  $i$ -th element in a sequence

$E()$  can be a simple function (e.g., linear or sin functions) or a learned lookup table.



## Aside: Positional Encoding (PE) for Self-Attention

**Motivation:** Maintain the order of input data since attention mechanisms are permutation invariant. PEs are shared across all input sequences.

**Linear Positional Encoding:**  $PE(pos) = a \cdot pos + b$ .

Problem: encoding increases with the sequence length, causing gradient problem for long sequences.

**Sin/cos Positional Encoding (Default):**

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

PE for each dimension (i) repeats periodically, combine different waveforms at each dimension to get a unique embedding.

**Learned Positional Encoding:**  $PE_{\theta}(pos, i)$ .

Learn the most suitable position embedding for the training set.

# Masked Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  (Shape:  $N_x \times D_x$ )

**Key matrix:**  $W_K$  (Shape:  $D_x \times D_Q$ )

**Value matrix:**  $W_V$  (Shape:  $D_x \times D_V$ )

**Query matrix:**  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

**Query vectors:**  $Q = XW_Q$

**Key vectors:**  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

**Value vectors:**  $V = XW_V$  (Shape:  $N_x \times D_V$ )

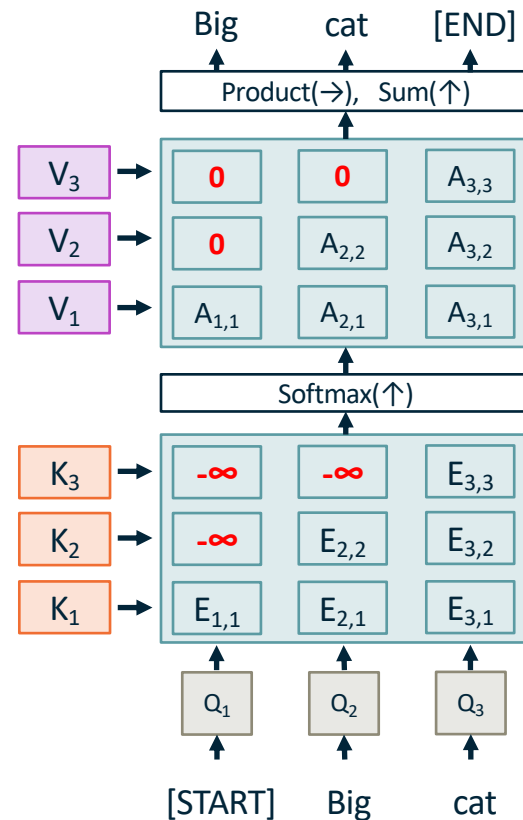
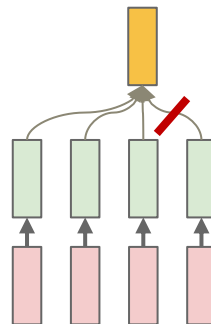
**Similarities:**  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

**Output vectors:**  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Don't let vectors "look ahead" in the sequence

Used for sequence decoding  
(predict next word)



# Multi-headed Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

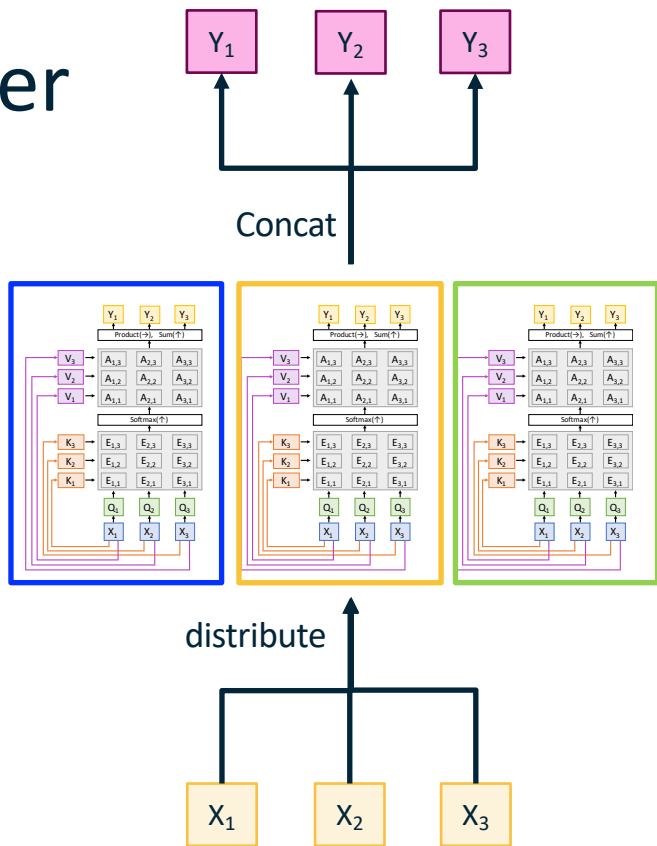
Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

Use  $H$  independent  
“Attention Heads” in  
parallel



# Multi-headed Self-Attention Layer

## Inputs:

Input vectors:  $X$  (Shape:  $N_x \times D_x$ )

Key matrix:  $W_K$  (Shape:  $D_x \times D_Q$ )

Value matrix:  $W_V$  (Shape:  $D_x \times D_V$ )

Query matrix:  $W_Q$  (Shape:  $D_x \times D_Q$ )

## Computation:

Query vectors:  $Q = XW_Q$

Key vectors:  $K = XW_K$  (Shape:  $N_x \times D_Q$ )

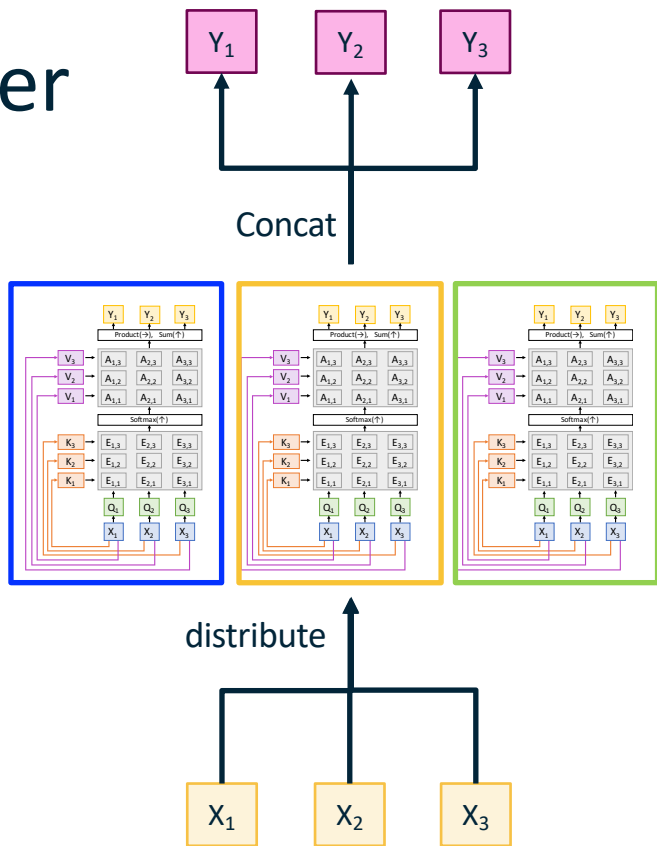
Value vectors:  $V = XW_V$  (Shape:  $N_x \times D_V$ )

Similarities:  $E = QK^T$  (Shape:  $N_x \times N_x$ )  $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  (Shape:  $N_x \times N_x$ )

Output vectors:  $Y = AV$  (Shape:  $N_x \times D_V$ )  $Y_i = \sum_j A_{i,j} V_j$

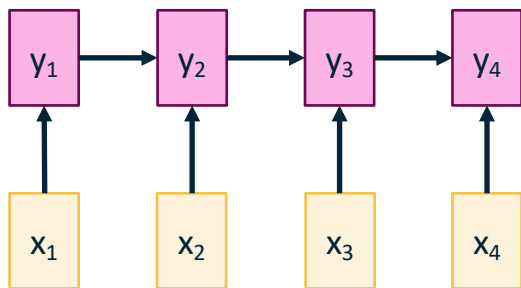
Use  $H$  independent  
“Attention Heads” in  
parallel



Highly parallelizable: Can compute attentions for all input element from all head in parallel!

# Three Ways of Processing Sequences

## Recurrent Neural Network



Works on **Ordered Sequences**

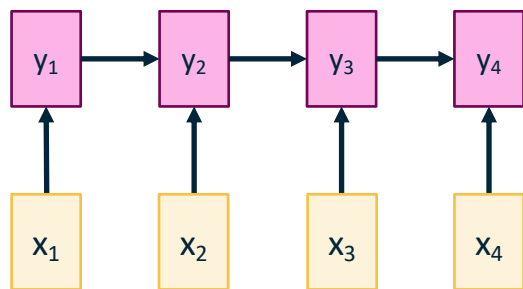
**(+) Natural sequential processing:**  
“sees” the input sequence in its  
original ordering

**(-) Forgetful: difficult to handle**  
long-range dependencies.

**(-) Not parallelizable: need to**  
compute hidden states sequentially

# Three Ways of Processing Sequences

## Recurrent Neural Network



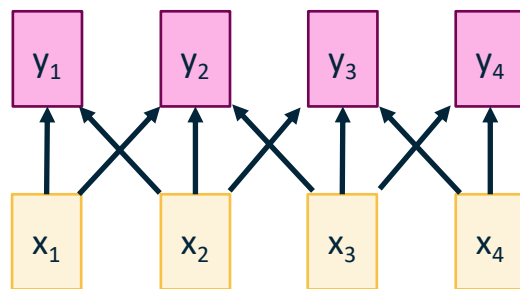
Works on **Ordered Sequences**

**(+) Natural sequential processing:** “sees” the input sequence in its original ordering

**(-) Forgetful:** difficult to handle long-range dependencies.

**(-) Not parallelizable:** need to compute hidden states sequentially

## 1D Convolution



Works on **Multidimensional Grids**

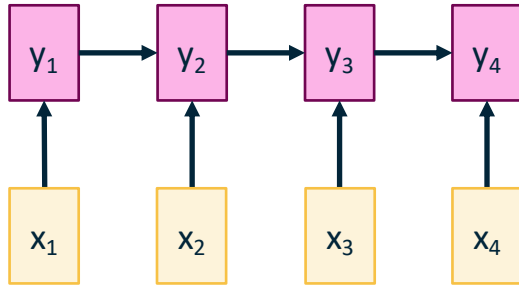
**(-) Bad at long sequences:** Need to stack many conv layers for outputs to “see” the whole sequence

**(+) Highly parallel:** Each output can be computed in parallel



# Three Ways of Processing Sequences

## Recurrent Neural Network



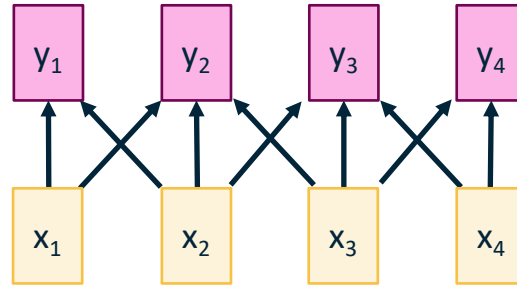
Works on **Ordered Sequences**

(+) **Natural sequential processing:** “sees” the input sequence in its original ordering

(-) **Forgetful:** difficult to handle long-range dependencies.

(-) **Not parallelizable:** need to compute hidden states sequentially

## 1D Convolution

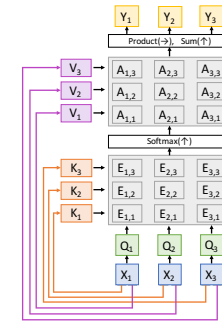


Works on **Multidimensional Grids**

(-) **Bad at long sequences:** Need to stack many conv layers for outputs to “see” the whole sequence

(+) **Highly parallel:** Each output can be computed in parallel

## Self-Attention



Works on **Sets of Vectors**

(+) **Good at long sequences:** after one self-attention layer, each output “sees” all inputs!

(+) **Highly parallel:** Each output can be computed in parallel

(-) **Very memory intensive**

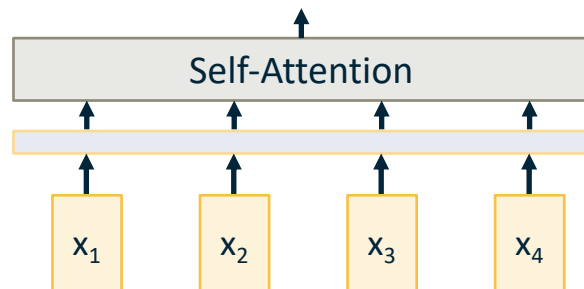
(-) **Requires positional encoding**

# The Transformer Block



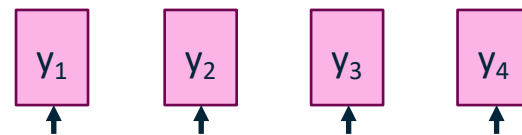
# The Transformer Block

All vectors interact  
with each other

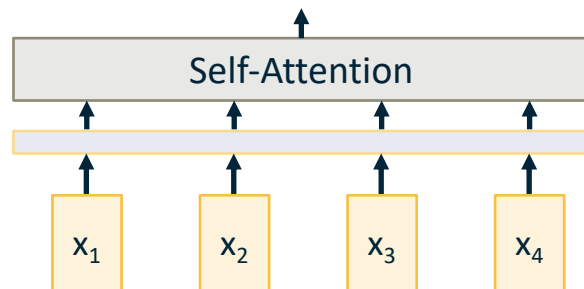


# The Transformer Block

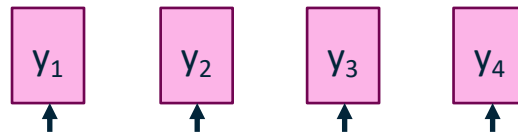
MLP independently on each vector



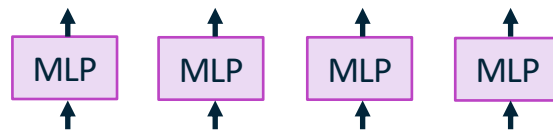
All vectors interact with each other



# The Transformer Block

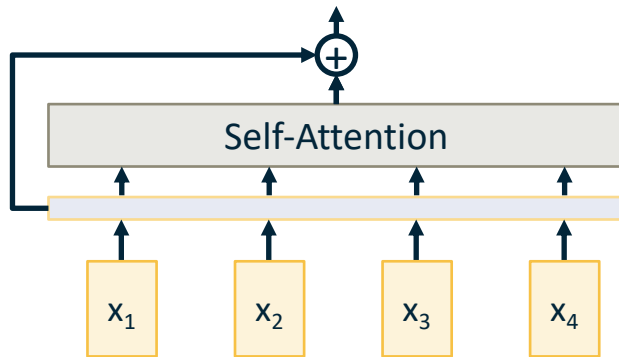


MLP independently on each vector



Residual connection

All vectors interact with each other



# The Transformer Block

## Recall **Layer Normalization**:

Given  $h_1, \dots, h_N$  (shape: D)

scale:  $\gamma$  (shape: D)

shift:  $\beta$  (shape: D)

$\mu_i = (1/D)\sum_j h_{i,j}$  (scalar)

$\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2)^{1/2}$  (scalar)

$z_i = (h_i - \mu_i) / \sigma_i$  (shape: D)

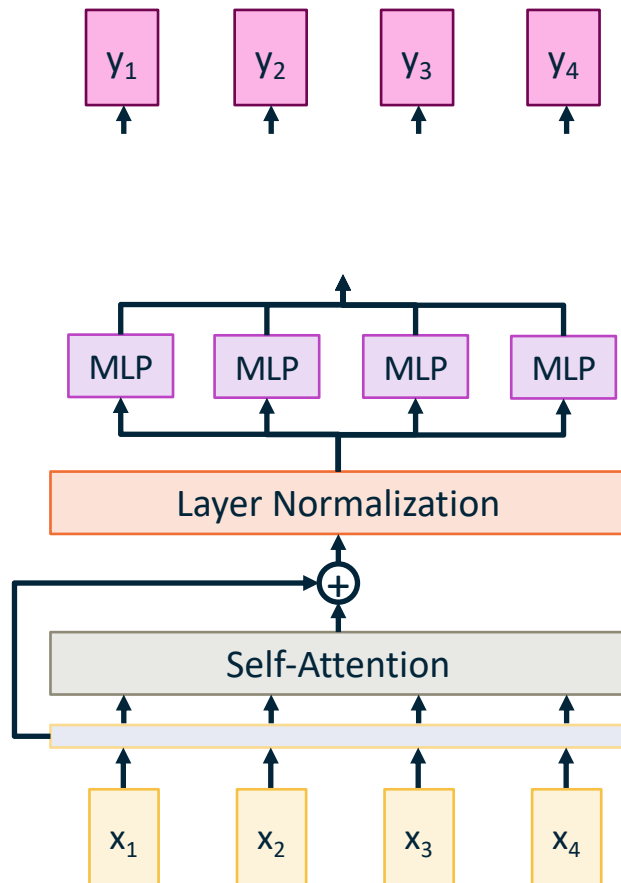
$y_i = \gamma * z_i + \beta$  (shape: D)

Applied **per element**, not across the sequence

MLP independently on each vector

Residual connection

All vectors interact with each other



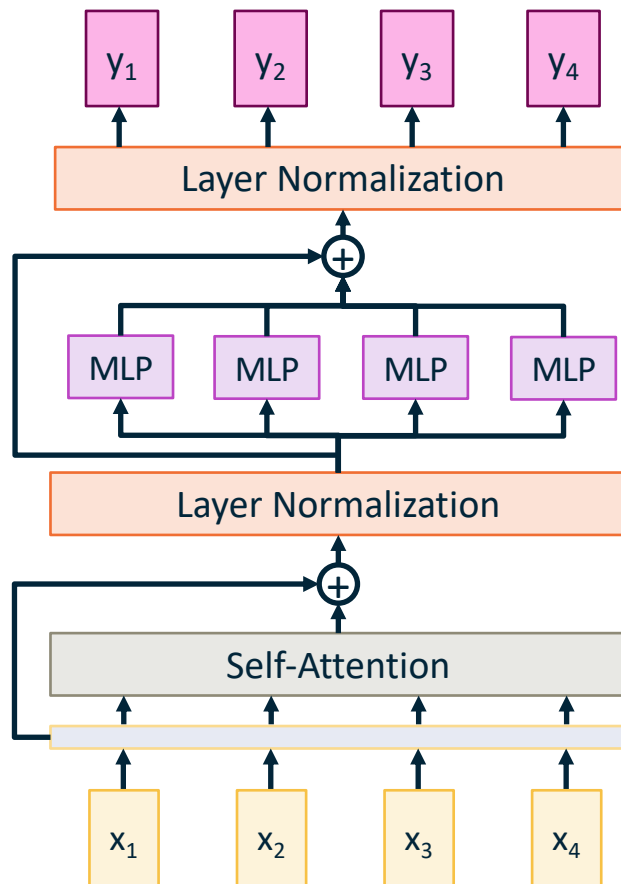
# The Transformer Block

Residual connection

MLP independently on each vector

Residual connection

All vectors interact with each other



# The Transformer Block

## Transformer Block:

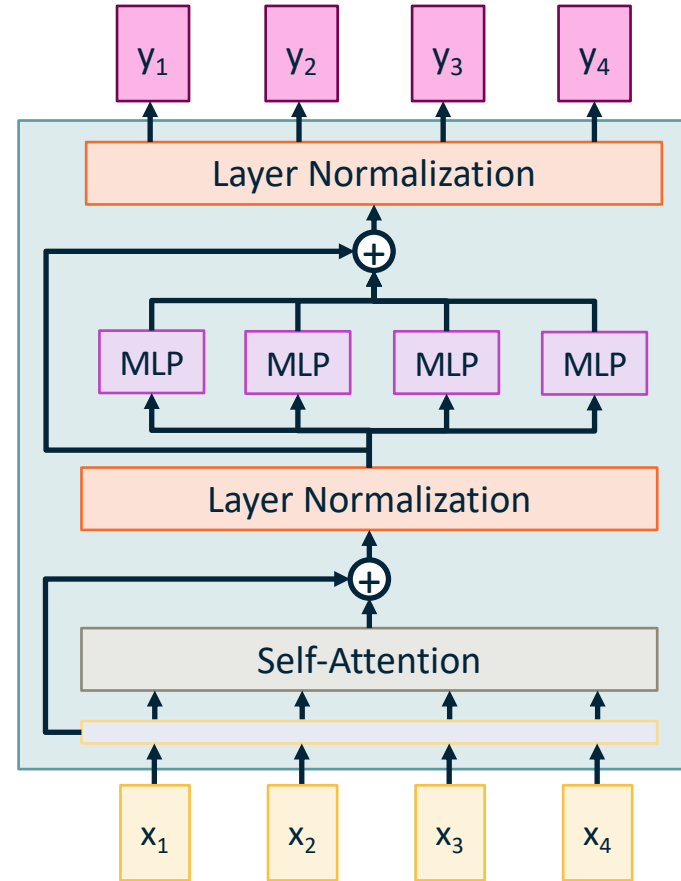
**Input:** Set of vectors  $x$

**Output:** Set of vectors  $y$

Self-attention is the only interaction among vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable





# The Transformer

## Transformer Block:

**Input:** Set of vectors  $x$

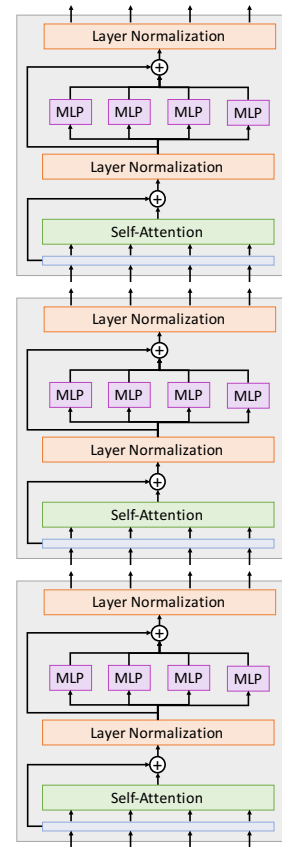
**Output:** Set of vectors  $y$

Self-attention is the only interaction among vectors!

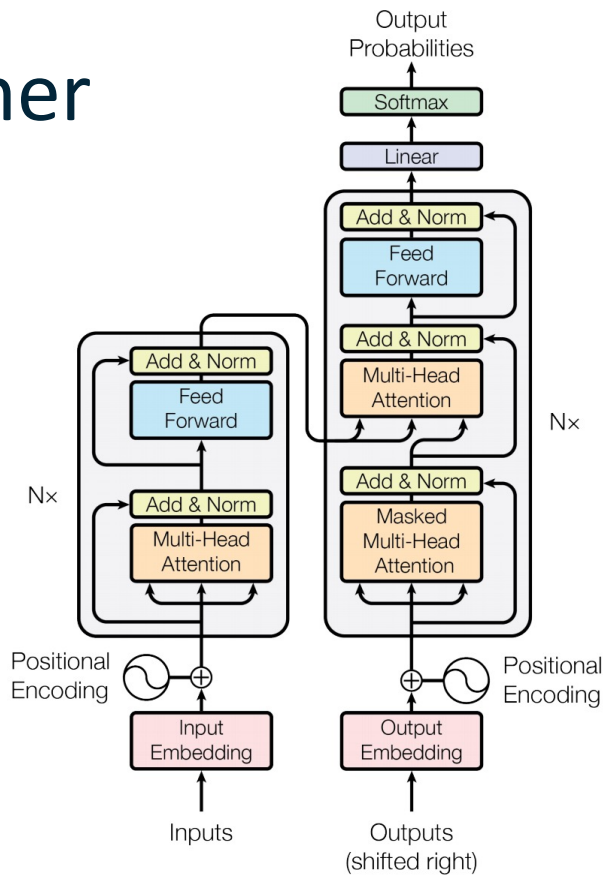
Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

A **Transformer** is a sequence of transformer blocks

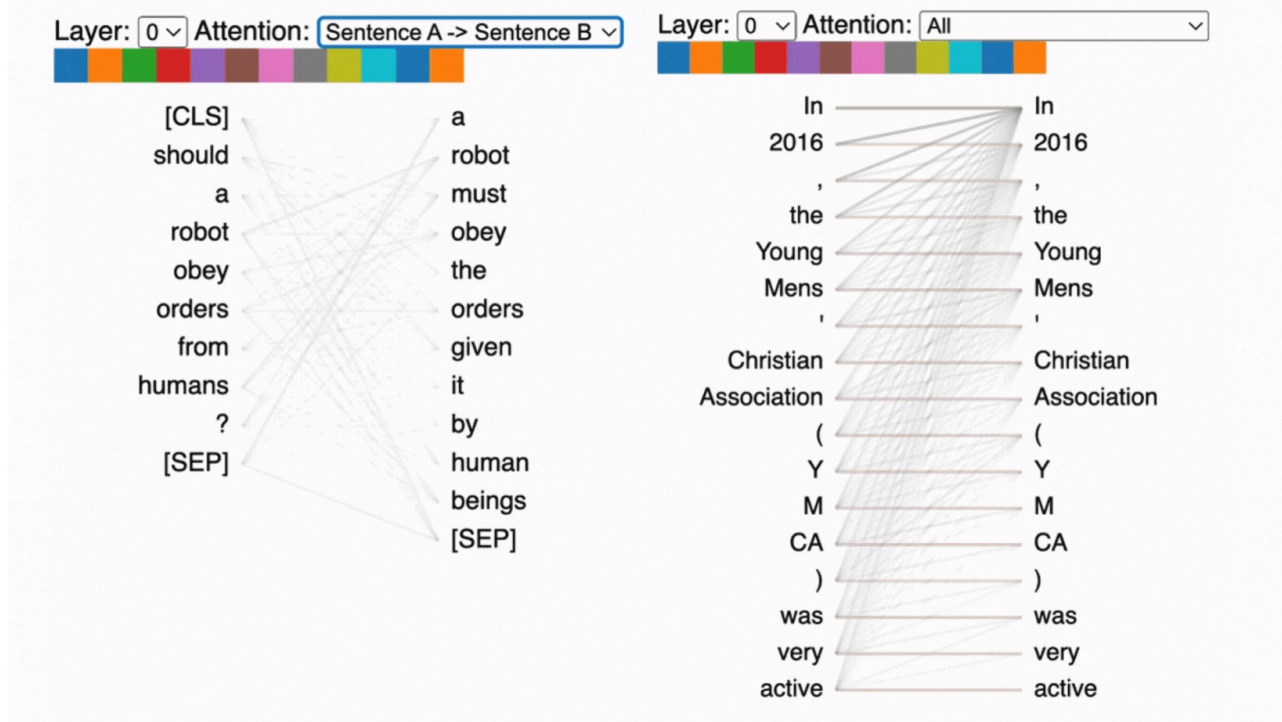


# The Transformer



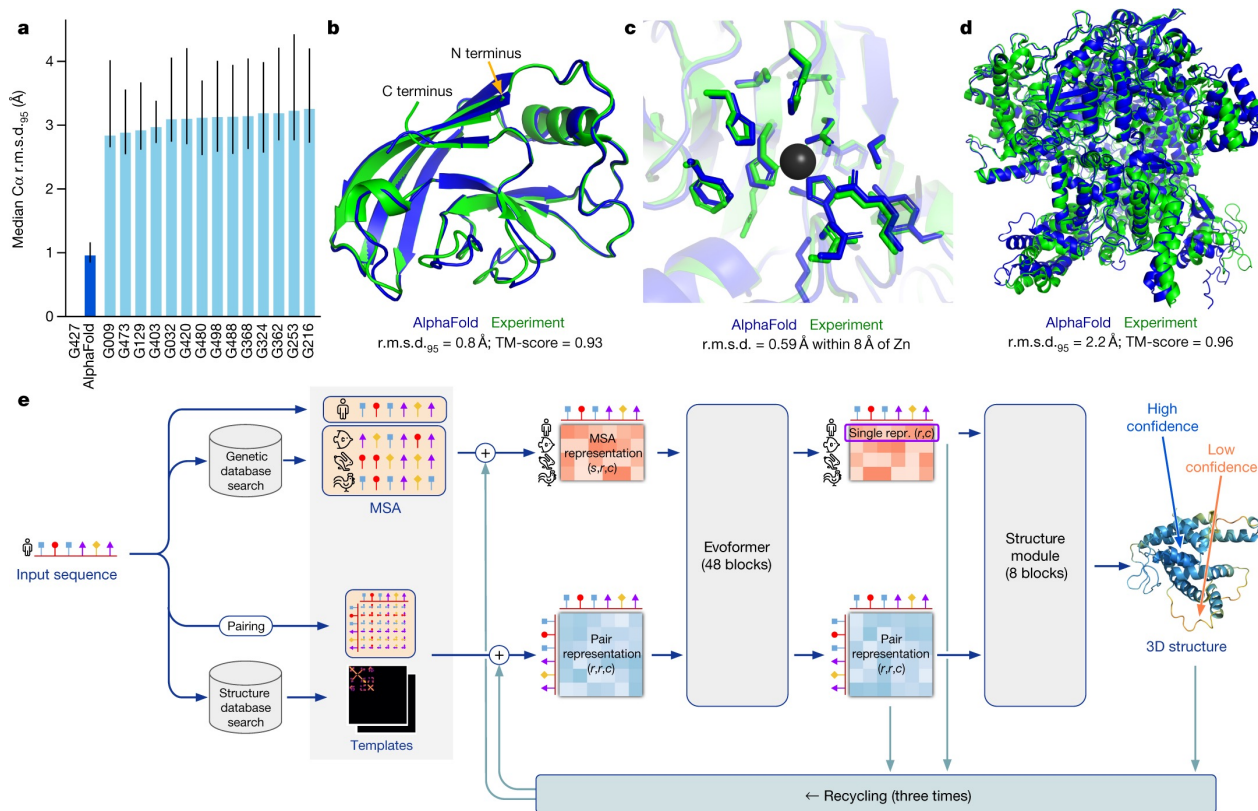
## Encoder-Decoder

# Visualizing Transformer Attentions

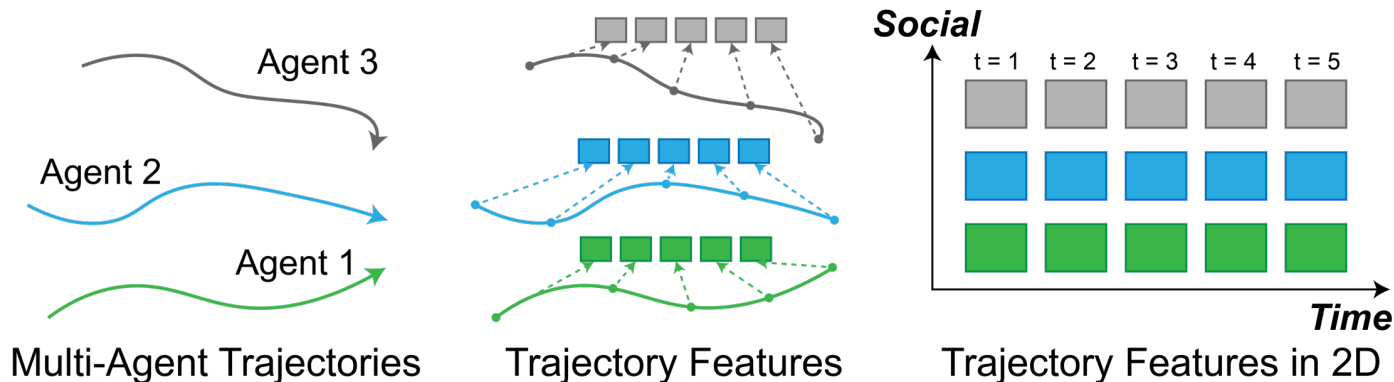


Can Attention/Transformers be used from  
more than text processing?

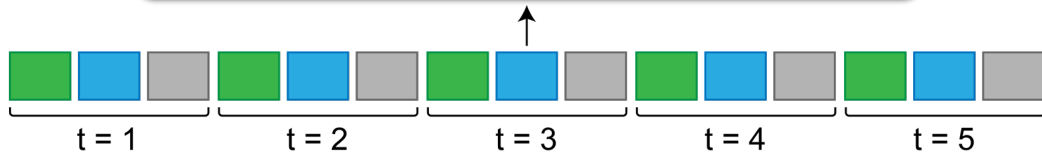
# Encoding/Decoding Protein Structures (AlphaFold)



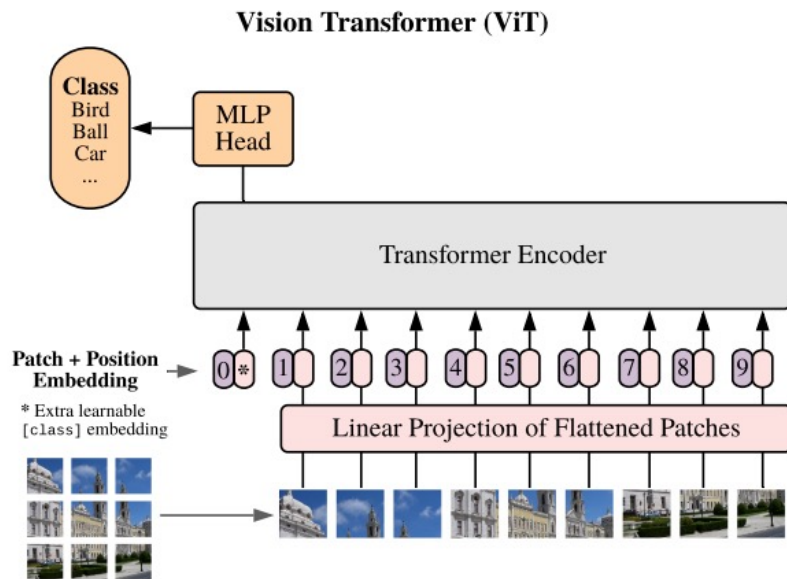
# Predicting Multi-agent Behaviors



**Agent-Aware Transformer**  
(Joint Social & Temporal Modeling + Preserve **Time** & **Agent** Information)

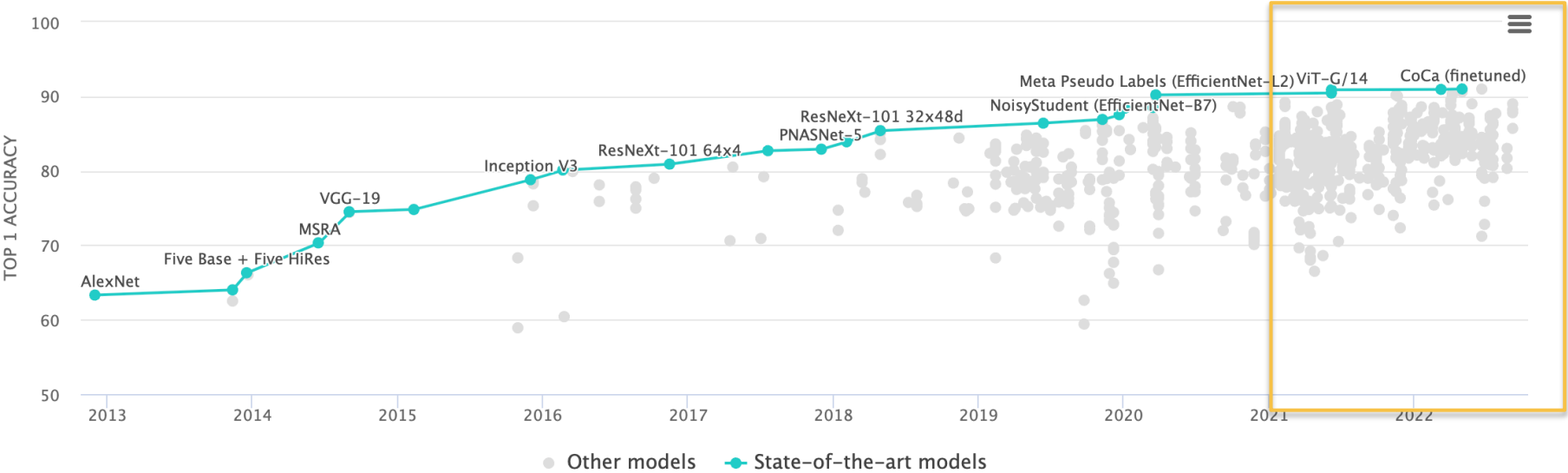


# ViT: Vision Transformer



An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale  
(Dosovitskiy *et al.*, 2021)

# ViT: Vision Transformer



Generally more expensive to train and execute than ConvNets-based models



# Formal Algorithms for Transformers

Mary Phuong<sup>1</sup> and Marcus Hutter<sup>1</sup>

<sup>1</sup>DeepMind

This document aims to be a self-contained, mathematically precise overview of transformer architectures and algorithms (*not* results). It covers what transformers are, how they are trained, what they are used for, their key architectural components, and a preview of the most prominent models. The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

*Keywords:* formal algorithms, pseudocode, transformers, attention, encoder, decoder, BERT, GPT, Gopher, tokenization, training, inference.

## Contents

|   |                                       |    |
|---|---------------------------------------|----|
| 1 | Introduction                          | 1  |
| 2 | Motivation                            | 1  |
| 3 | Transformers and Typical Tasks        | 3  |
| 4 | Tokenization: How Text is Represented | 4  |
| 5 | Architectural Components              | 4  |
| 6 | Transformer Architectures             | 7  |
| 7 | Transformer Training and Inference    | 8  |
| 8 | Practical Considerations              | 9  |
| A | References                            | 9  |
| B | List of Notation                      | 16 |

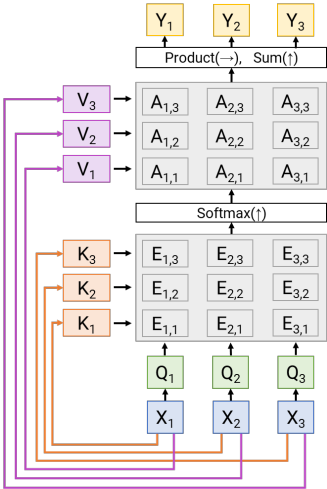
*A famous colleague once sent an actually very well-written paper he was quite proud of to a famous complexity theorist. His answer: “I can’t find a theorem in the paper. I have no idea what this*

plete, precise and compact overview of transformer architectures and formal algorithms (but *not* results). It covers what Transformers are (Section 6), how they are trained (Section 7), what they’re used for (Section 3), their key architectural components (Section 5), tokenization (Section 4), and a preview of practical considerations (Section 8) and the most prominent models.

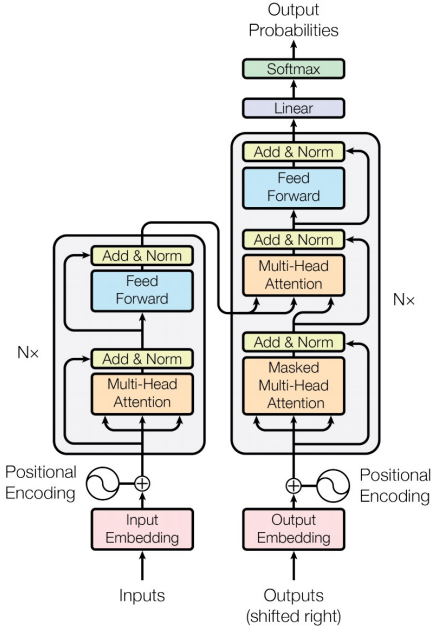
The essentially complete pseudocode is about 50 lines, compared to thousands of lines of actual real source code. We believe these formal algorithms will be useful for theoreticians who require compact, complete, and precise formulations, experimental researchers interested in implementing a Transformer from scratch, and

# Summary

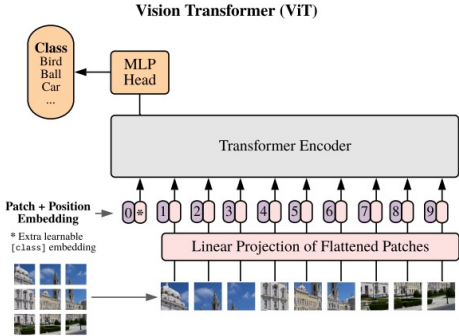
## Self-Attention



## Transformer Model



## Beyond Language



# A Lecture on Large Language Models

Nov 5th by William Held (GT, Stanford)

Fully-remote

# Today

- Finishing Attention, Transformers
- Deep learning hardware
  - CPU, GPU
- Deep learning software
  - PyTorch and TensorFlow
  - Static and Dynamic computation graphs

# Deep Learning Hardware

# Inside a computer

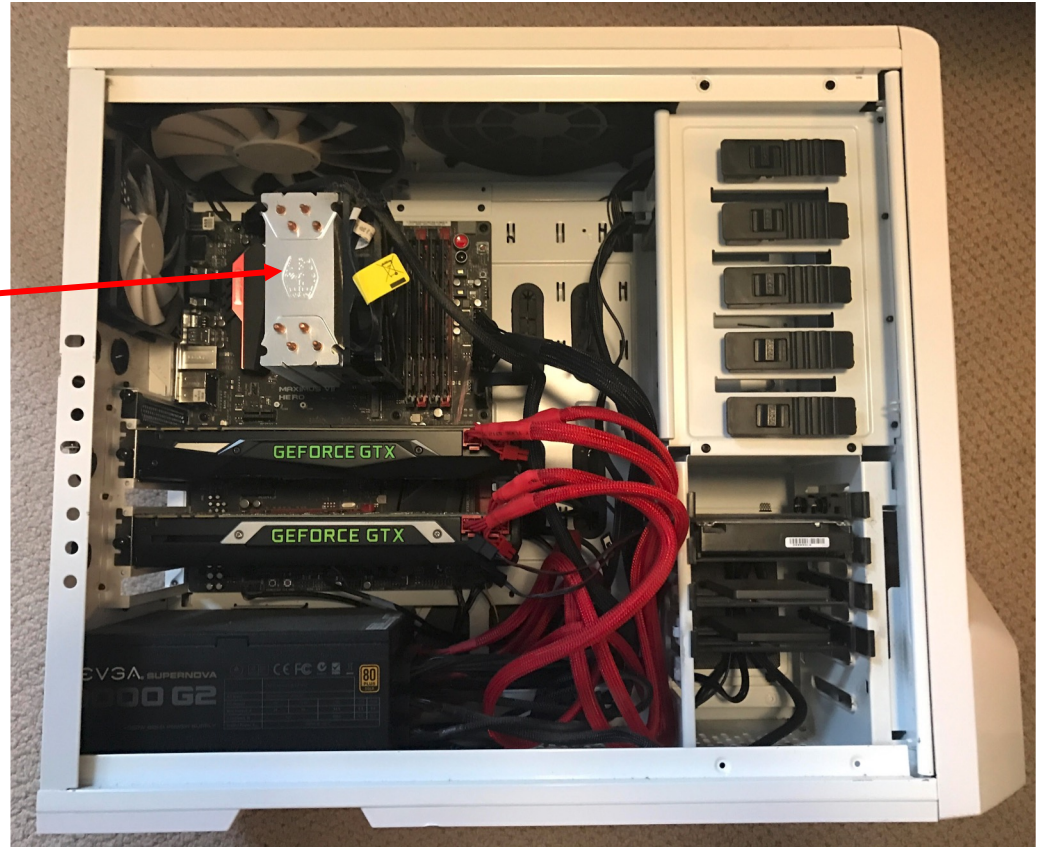


# Spot the CPU!

(central processing unit)



[This image](#) is licensed under [CC-BY 2.0](#)

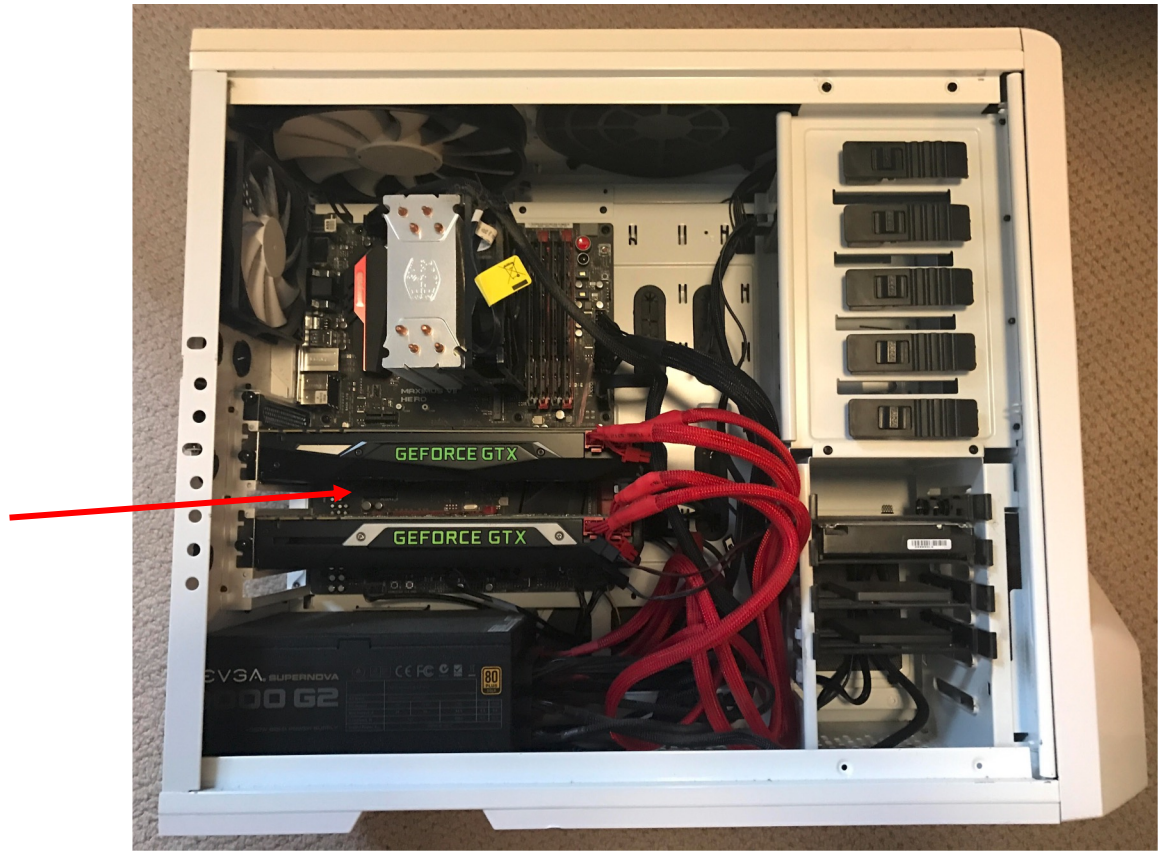


# Spot the GPUs!

(graphics processing unit)



This image is licensed under [CC-BY 2.0](https://creativecommons.org/licenses/by/2.0/)





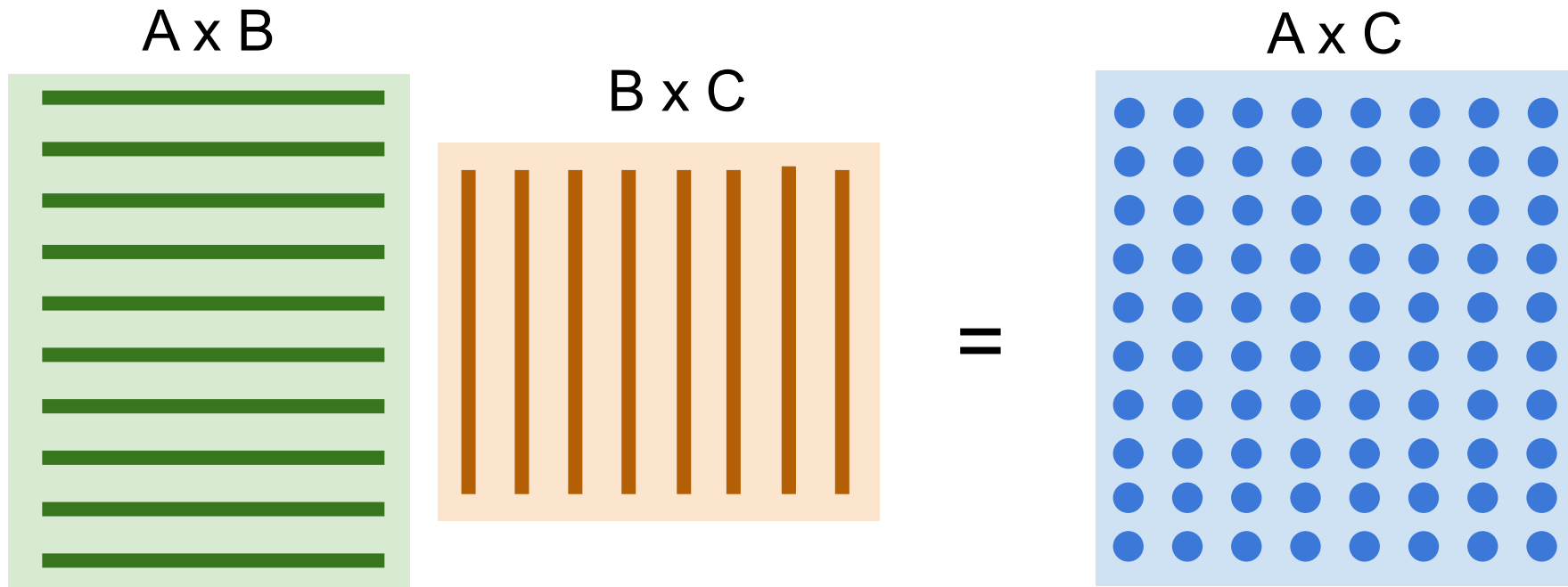
# CPU vs GPU

|                                     | Cores | Clock Speed | Memory       | Price  | Speed (throughput) |
|-------------------------------------|-------|-------------|--------------|--------|--------------------|
| <b>CPU</b><br>(Intel Core i9-7900k) | 10    | 4.3 GHz     | System RAM   | \$385  | ~640 GFLOPS FP32   |
| <b>GPU</b><br>(NVIDIA RTX 3090)     | 10496 | 1.6 GHz     | 24 GB GDDR6X | \$1499 | ~35.6 TFLOPS FP32  |

**CPU:** Fewer cores, but each core is much faster and much more capable; great at sequential tasks

**GPU:** More cores, but each core is much slower and “dumber”; great for parallel tasks

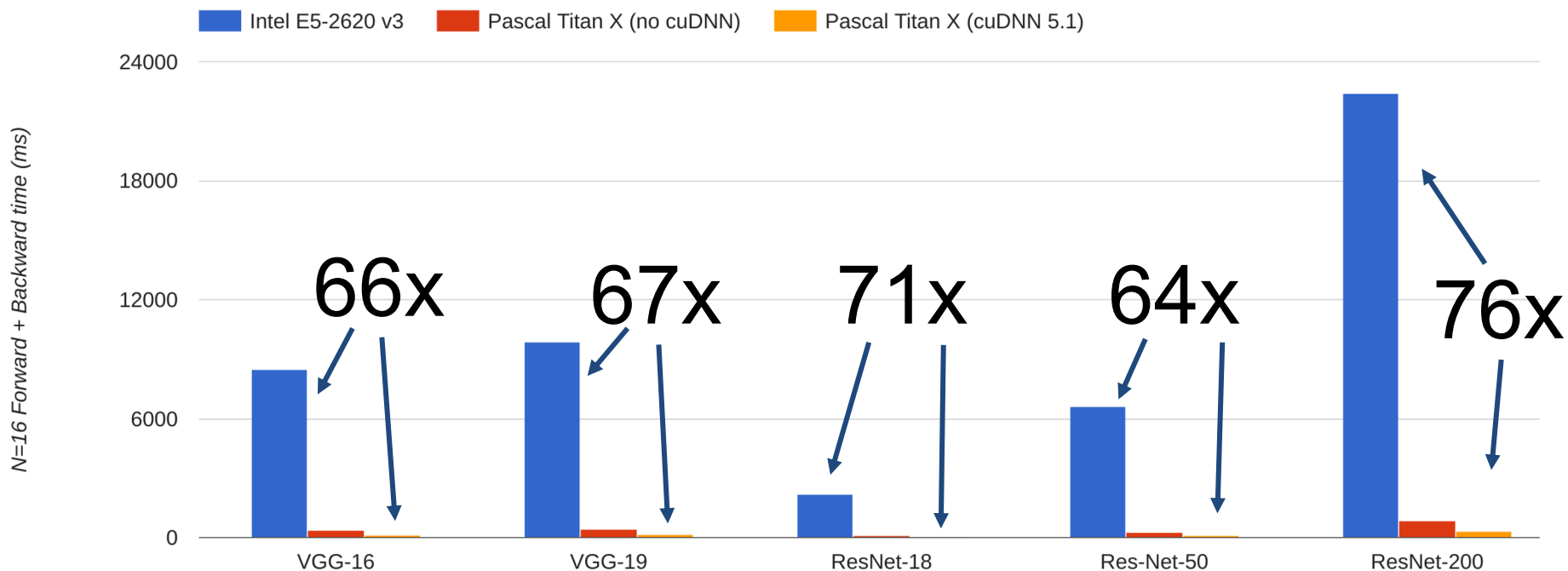
# Example: Matrix Multiplication



cuBLAS::GEMM (General Matrix-to-matrix Multiply)

# CPU vs GPU in practice

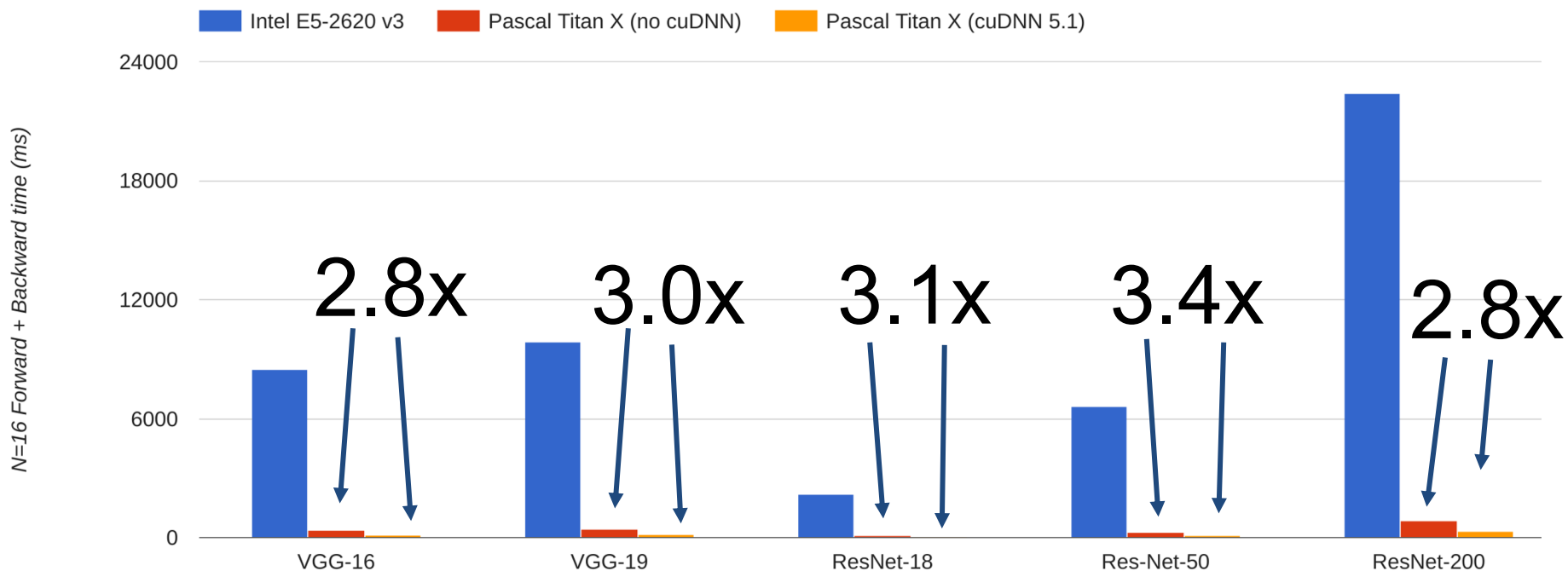
(CPU performance not well-optimized, a little unfair)



Data from <https://github.com/jcjohnson/cnn-benchmarks>

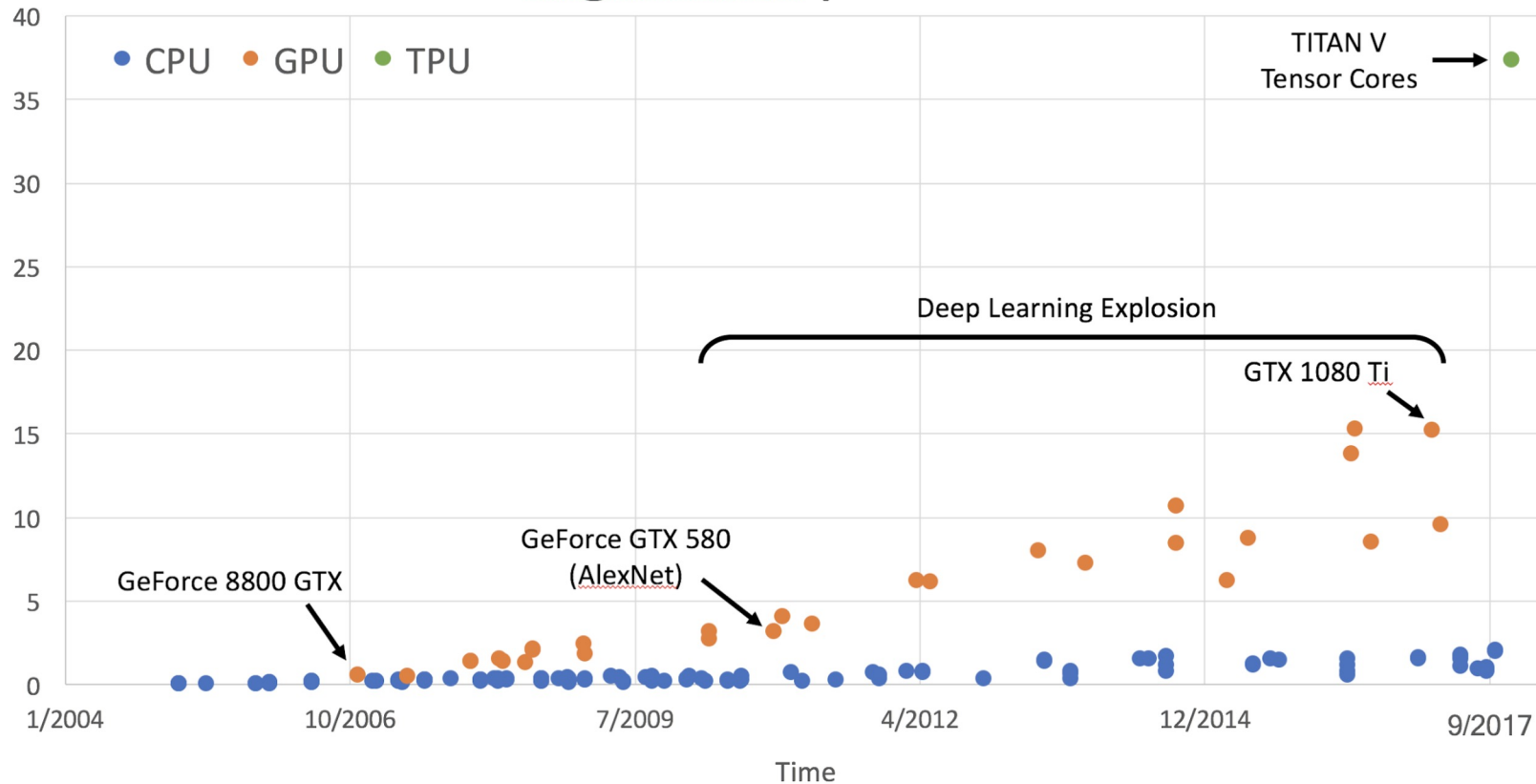
# CPU vs GPU in practice

cuDNN much faster than  
“unoptimized” CUDA

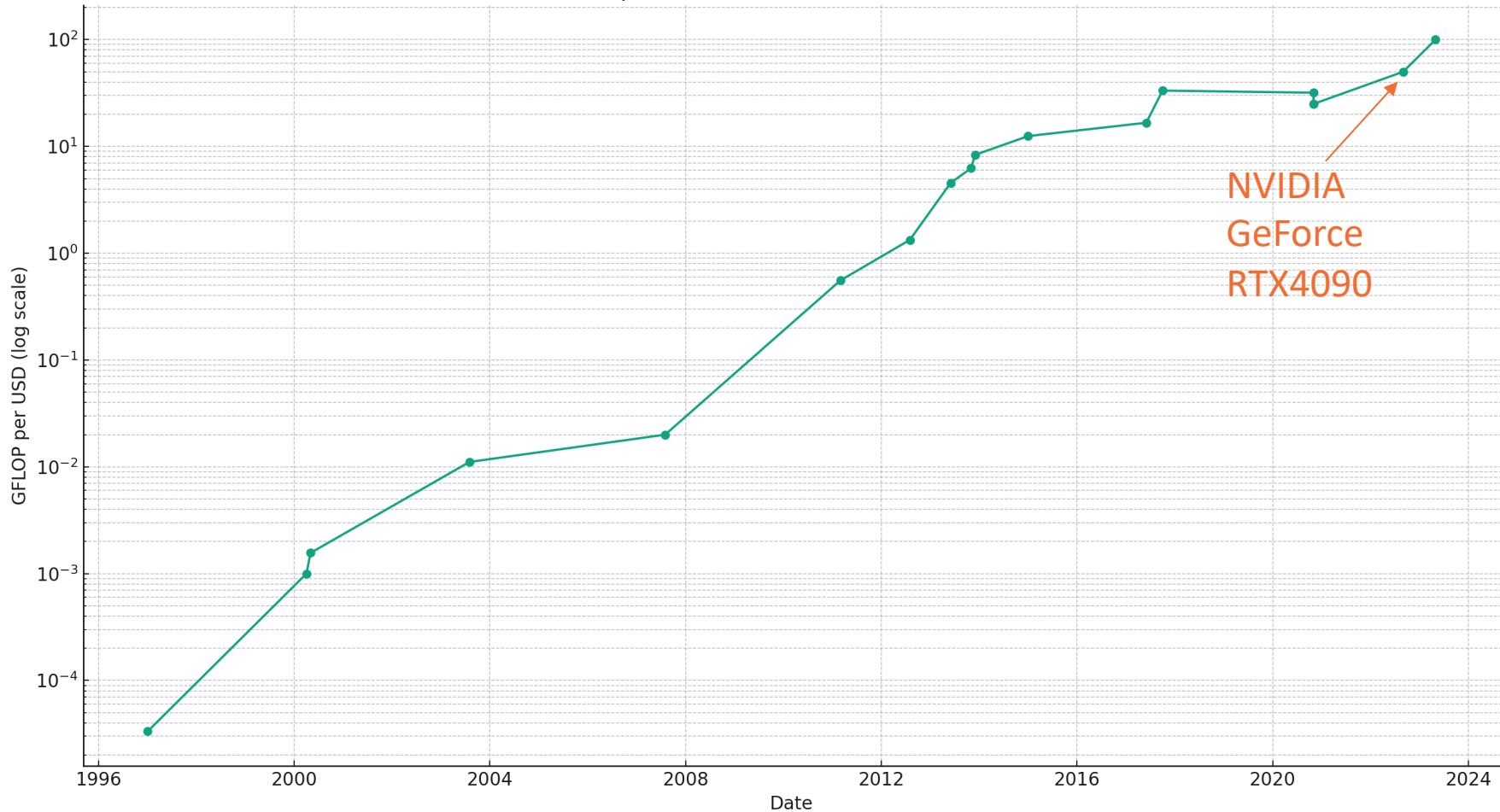


Data from <https://github.com/jcjohnson/cnn-benchmarks>

# GigaFLOPs per Dollar



GFLOP per USD Over Time (1990 onwards)



**NVIDIA**

**vs**

**AMD**

**NVIDIA**

**vs**

**AMD**



# CPU vs GPU

|   | Cores                                   | Clock Speed | Memory        | Price        | Speed   |
|---|---|-------------|---------------|--------------|---|
| <b>CPU</b><br>(Intel Core i7-7700k)     | 10                                      | 4.3 GHz     | System RAM    | \$385        | ~640 GFLOPs FP32  |
| <b>GPU</b><br>(NVIDIA RTX 3090)         | 10496                                   | 1.6 GHz     | 24 GB GDDR6X  | \$1499       | ~35.6 TFLOPs FP32                                       |
| <b>GPU (Data Center)</b><br>NVIDIA A100 | 6912 CUDA, 432 Tensor                   | 1.5 GHz     | 40/80 GB HBM2 | \$3/hr (GCP) | ~9.7 TFLOPs FP64<br>~20 TFLOPs FP32<br>~312 TFLOPs FP16 |
| <b>TPU</b><br>Google Cloud TPUv3        | 2 Matrix Units (MXUs) per core, 4 cores | ?           | 128 GB HBM    | \$8/hr (GCP) | ~420 TFLOPs (non-standard FP)                           |

**CPU:** Fewer cores, but each core is much faster and much more capable; great at sequential tasks

**GPU:** More cores, but each core is much slower and “dumber”; great for parallel tasks

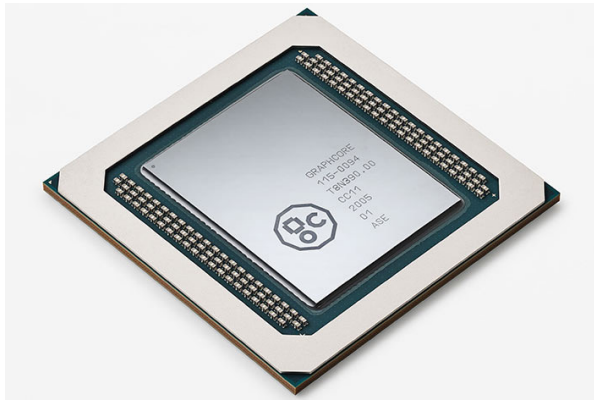
**TPU:** Specialized hardware for deep learning

# Aside: NPUs

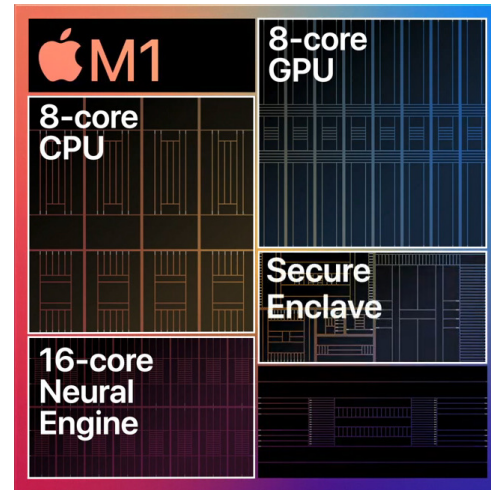
Neural Processing Units (NPUs) are specialized hardware designed for Deep Learning applications. Example: GraphCore IPU

**General pros:** larger on-device memory, lower power consumption

**General cons:** specialized computation units (compared to GPU and CPUs). Smaller instruction sets. Less supported by popular platforms (PyTorch, TensorFlow)



Graphcore M2000



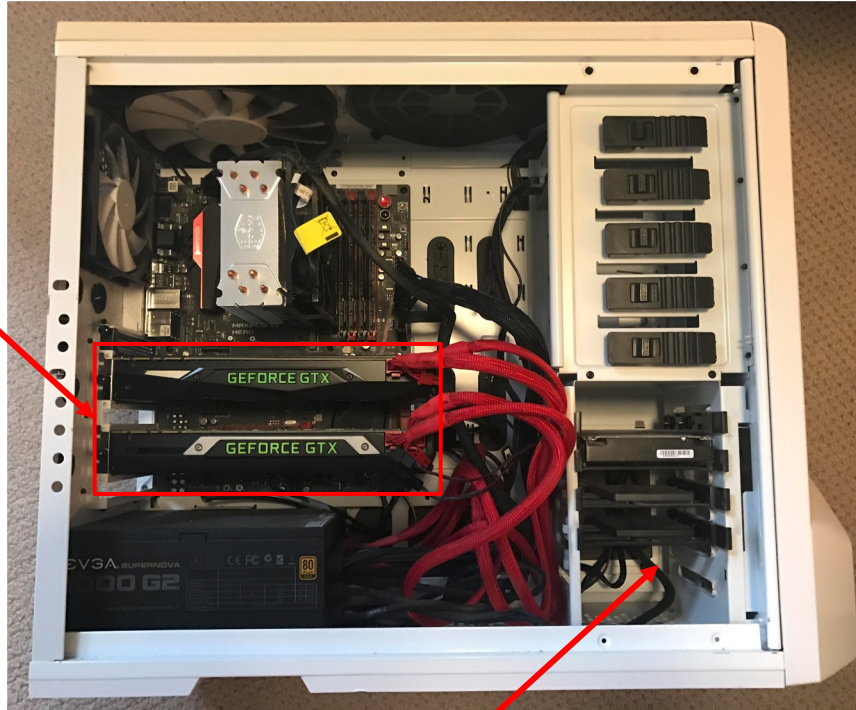
Apple M1

# Programming GPUs

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower on NVIDIA hardware
- HIP <https://github.com/ROCm-Developer-Tools/HIP>
  - New project that automatically converts CUDA code to something that can run on AMD GPUs
- CS 8803 – GPU at GaTech
  - Taught by Prof. Hyesoon Kim

# CPU / GPU Communication

Model  
is here

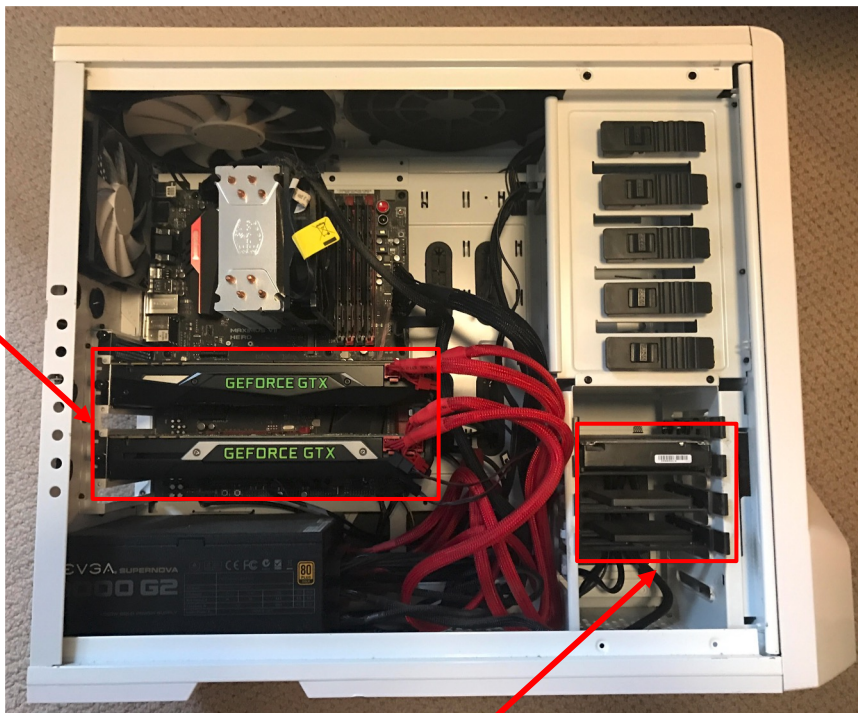


Data is here

Data access rate: RAM and the GPU over PCIe lanes is about **16 GB/s**. GPU's internal memory (like GDDR6) is about **448 GB/s**.

# CPU / GPU Communication

Model  
is here



Data is here

Data access rate: RAM and the GPU over PCIe lanes is about **16 GB/s**. GPU's internal memory (like GDDR6) is about **448 GB/s**.

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

## Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

# Deep Learning Software

# A zoo of frameworks!

Caffe  
(UC Berkeley)



Caffe2  
(Facebook)  
mostly features absorbed  
by PyTorch



Torch  
(NYU / Facebook)



PyTorch  
(Facebook)

Theano  
(U Montreal)



TensorFlow  
(Google)

PaddlePaddle  
(Baidu)

Chainer  
(Preferred Networks)  
The company has officially migrated its research  
infrastructure to PyTorch

MXNet  
(Amazon)

Developed by U Washington, CMU, MIT,  
Hong Kong U, etc but main framework of  
choice at AWS

CNTK  
(Microsoft)

JAX  
(Google)

And others...

# A zoo of frameworks!

Caffe  
(UC Berkeley)



Caffe2  
(Facebook)  
mostly features absorbed  
by PyTorch



Torch  
(NYU / Facebook)



PyTorch  
(Facebook)

Theano  
(U Montreal)



TensorFlow  
(Google)

We'll focus on these

PaddlePaddle  
(Baidu)

Chainer  
(Preferred Networks)  
The company has officially migrated its research infrastructure to PyTorch

MXNet  
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

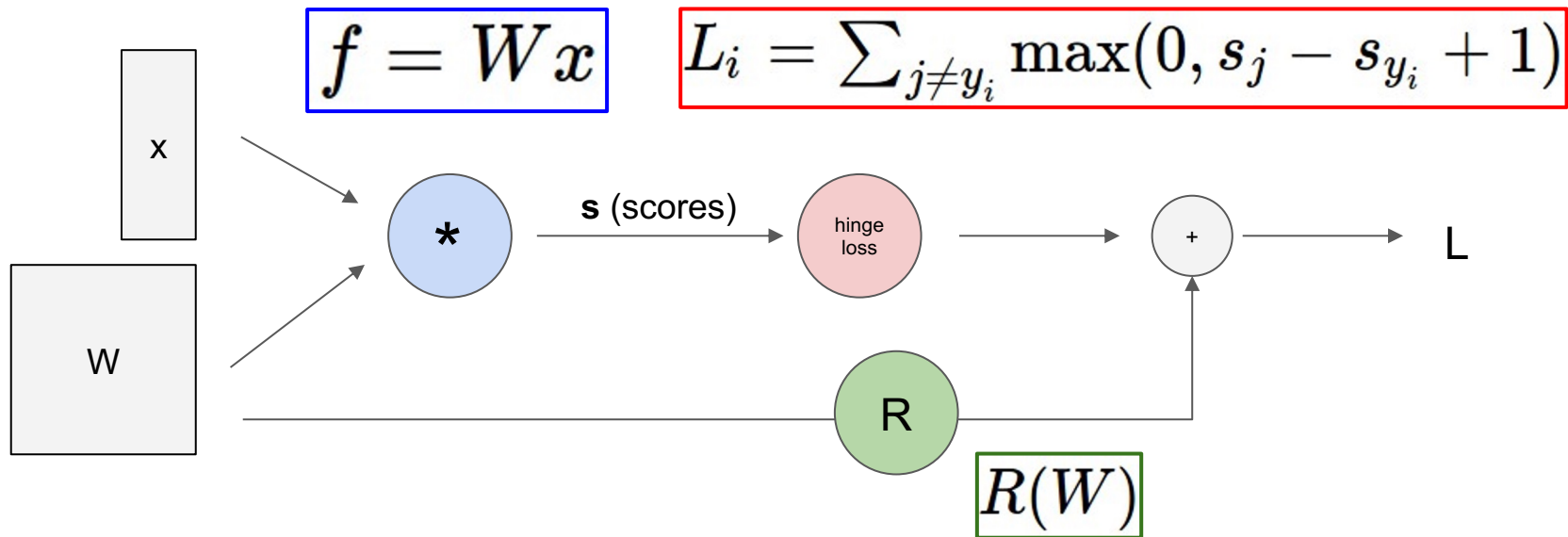
CNTK  
(Microsoft)

JAX  
(Google)

And others...



# Recall: Computational Graphs



# Recall: Computational Graphs

input image

weights

loss

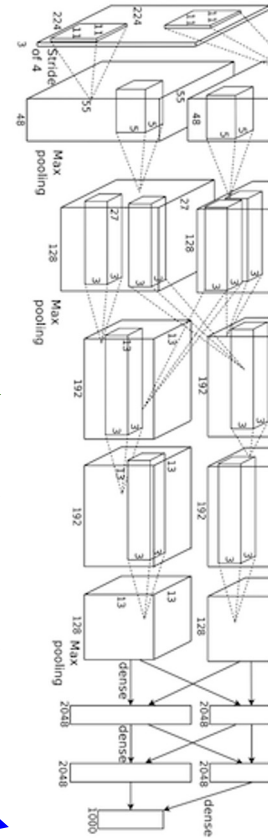


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Recall: Computational Graphs

input image

loss

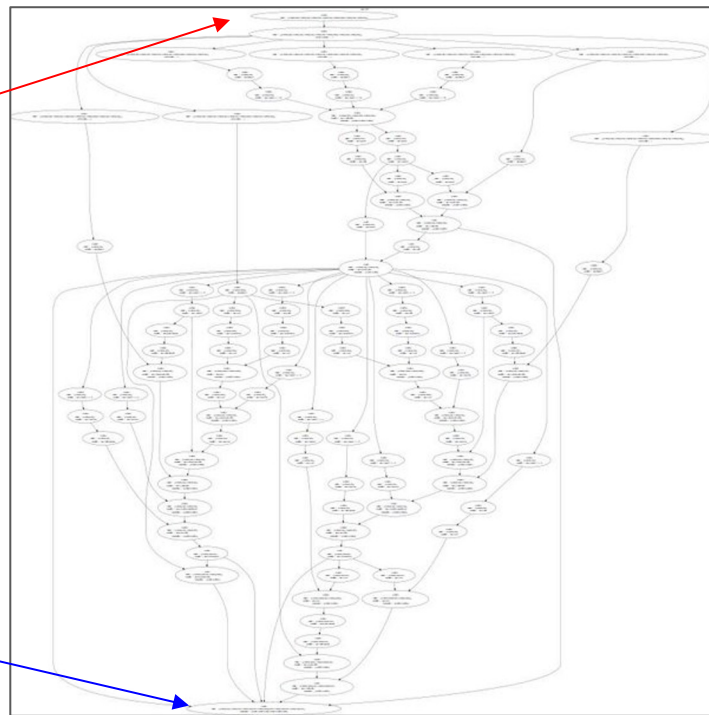


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

# The point of deep learning frameworks

- (1) Quick to develop and test new ideas
- (2) Automatically compute gradients
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, OpenCL, etc)

# Computational Graphs

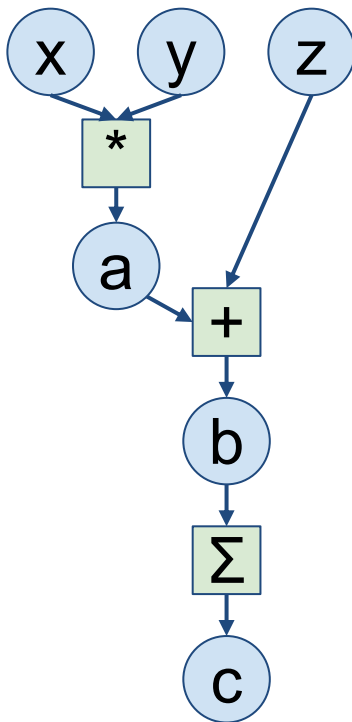
## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



# Computational Graphs

## Numpy

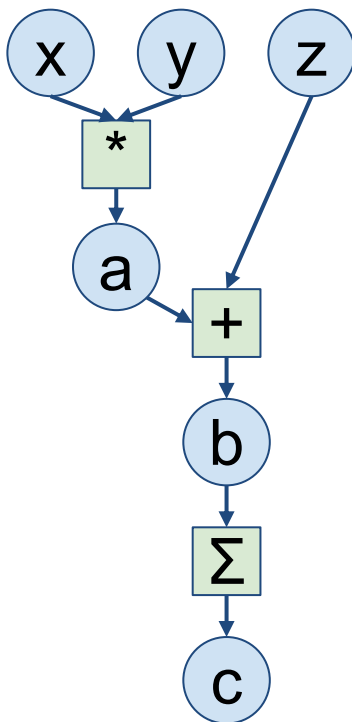
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



# Computational Graphs

## Numpy

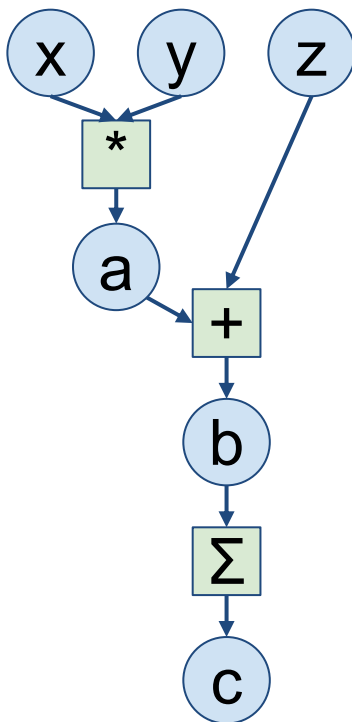
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



## Good:

Clean API, easy to write numeric code

## Bad:

- Have to compute our own gradients
- Can't run on GPU

# Computational Graphs

## Numpy

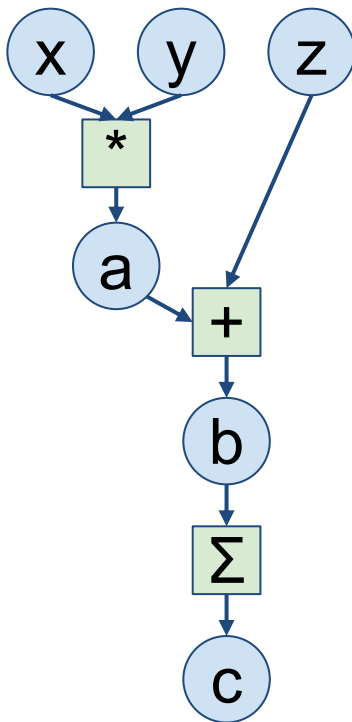
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

```
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



## PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!



# Computational Graphs

## Numpy

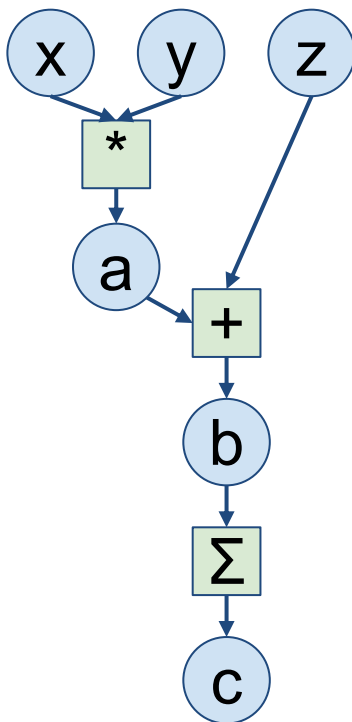
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



## PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

# Computational Graphs

## Numpy

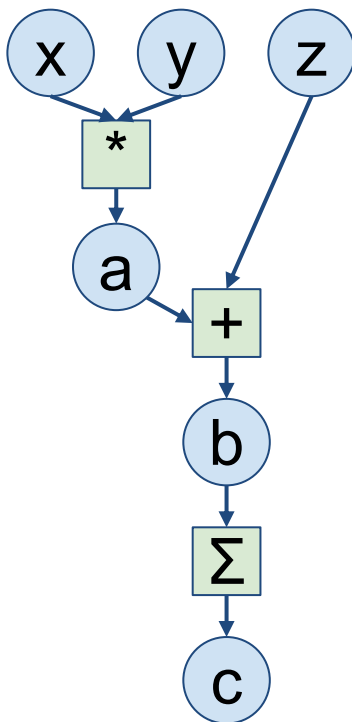
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



## PyTorch

```
import torch
device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

# PyTorch

(More details)

# PyTorch: Fundamental Concepts

**torch.Tensor:** Like a numpy array, but can run on GPU

**torch.autograd:** Package for building computational graphs out of Tensors, and automatically computing gradients

**torch.nn.Module:** A neural network layer; may store state or learnable weights

# PyTorch: Versions

For this class we are using **PyTorch version  $\geq 2.0.0$**   
**(newest is v2.1.0)**

Major API change in release 1.0

Be careful if you are looking at older PyTorch code ( $<1.0$ )!

# PyTorch: Tensors

Running example: Train  
a two-layer ReLU  
network on random data  
with L2 loss

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)


learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# PyTorch: Tensors

Create random tensors  
for data and weights



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)


learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# PyTorch: Tensors

Forward pass: compute predictions and loss



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```



# PyTorch: Tensors

Backward pass:  
manually compute  
gradients

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# PyTorch: Tensors

Gradient descent  
step on weights

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# PyTorch: Tensors

To run on GPU, just use a different device!

```
import torch
```

```
device = torch.device('cuda:0')
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in, device=device)
```

```
y = torch.randn(N, D_out, device=device)
```

```
w1 = torch.randn(D_in, H, device=device)
```

```
w2 = torch.randn(H, D_out, device=device)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    h = x.mm(w1)
```

```
    h_relu = h.clamp(min=0)
```

```
    y_pred = h_relu.mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    grad_y_pred = 2.0 * (y_pred - y)
```

```
    grad_w2 = h_relu.t().mm(grad_y_pred)
```

```
    grad_h_relu = grad_y_pred.mm(w2.t())
```

```
    grad_h = grad_h_relu.clone()
```

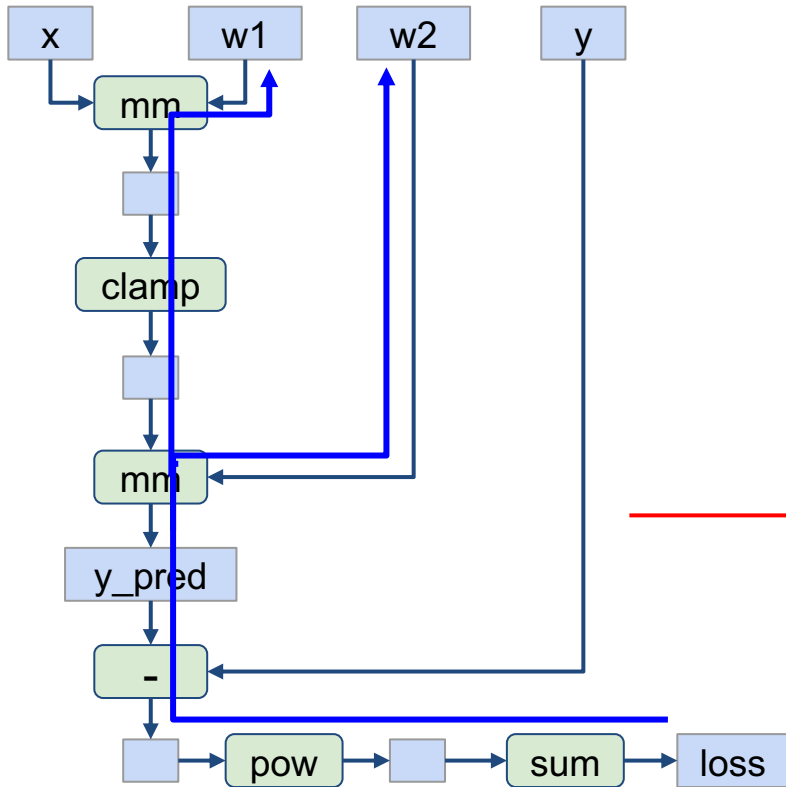
```
    grad_h[h < 0] = 0
```

```
    grad_w1 = x.t().mm(grad_h)
```

```
    w1 -= learning_rate * grad_w1
```

```
    w2 -= learning_rate * grad_w2
```

# PyTorch: Autograd



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

```
    with torch.no_grad():
```

```
        w1 -= learning_rate * w1.grad
```

```
        w2 -= learning_rate * w2.grad
```

```
        w1.grad.zero_()
```

```
        w2.grad.zero_()
```

# PyTorch: Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

Make gradient step on weights, then zero them. Torch.no\_grad means “don’t build a computational graph for this part”

# PyTorch: Autograd

PyTorch methods that end in underscore modify the Tensor in-place; methods that don't return a new Tensor

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

# PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass

```
class MyReLU(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input
```

# PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass

Define a helper function to make it easy to use the new function

```
class MyReLU(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input  
  
def my_relu(x):  
    return MyReLU.apply(x)
```



# PyTorch: New Autograd Functions

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```

Can use our new autograd function in the forward pass

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

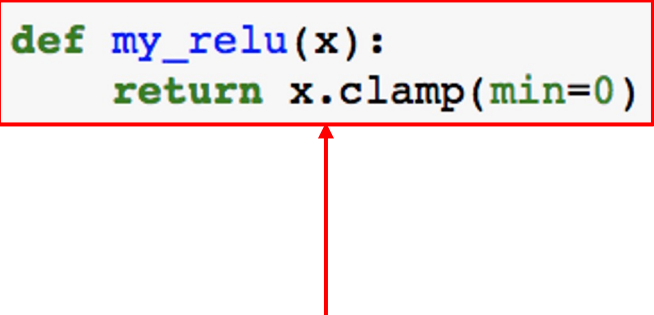
learning_rate = 1e-6
for t in range(500):
    y_pred = my_relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

# PyTorch: New Autograd Functions

```
def my_relu(x):  
    return x.clamp(min=0)
```



In practice you almost never need to define new autograd functions! Only do it when you need custom backward. In this case we can just use a normal PyTorch function

```
N, D_in, H, D_out = 64, 1000, 100, 10  
  
x = torch.randn(N, D_in)  
y = torch.randn(N, D_out)  
w1 = torch.randn(D_in, H, requires_grad=True)  
w2 = torch.randn(H, D_out, requires_grad=True)  
  
learning_rate = 1e-6  
for t in range(500):  
    y_pred = my_relu(x.mm(w1)).mm(w2)  
    loss = (y_pred - y).pow(2).sum()  
  
    loss.backward()  
  
    with torch.no_grad():  
        w1 -= learning_rate * w1.grad  
        w2 -= learning_rate * w2.grad  
        w1.grad.zero_()  
        w2.grad.zero_()
```

# PyTorch: Computational Graphs

input image

loss

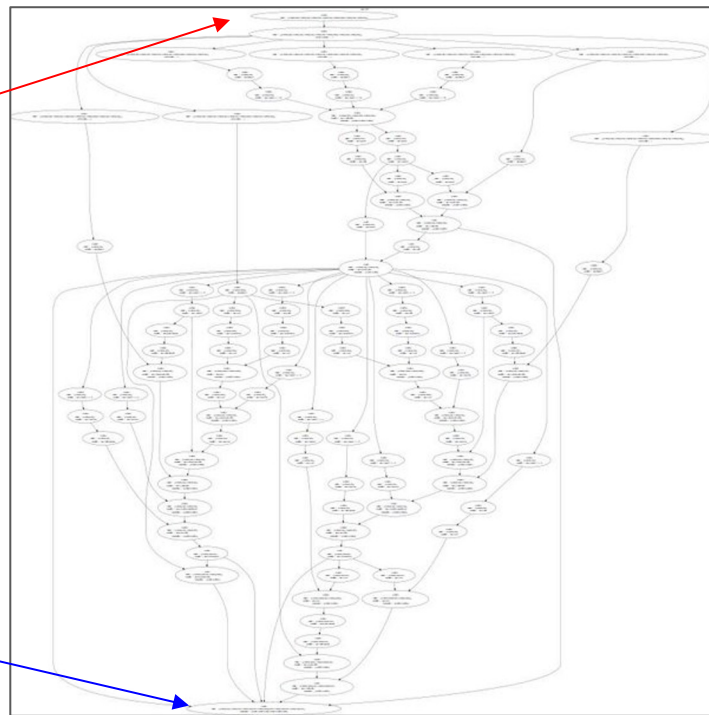


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

# PyTorch: Dynamic Computation Graphs

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

# PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

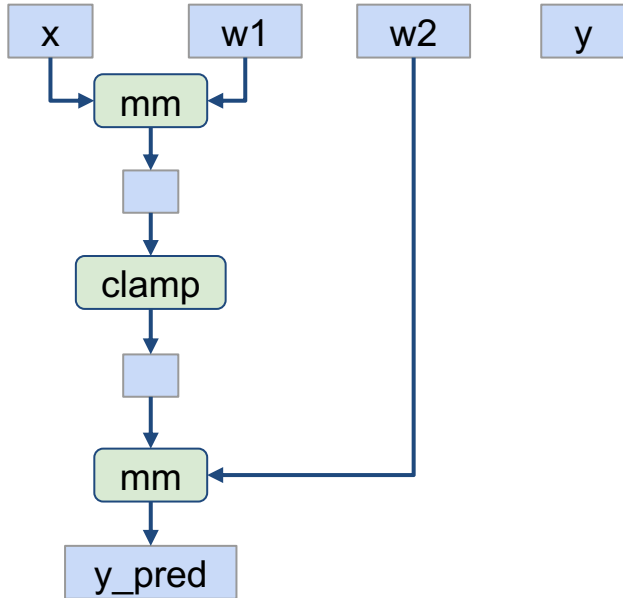
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Create Tensor objects

# PyTorch: Dynamic Computation Graphs



```
import torch

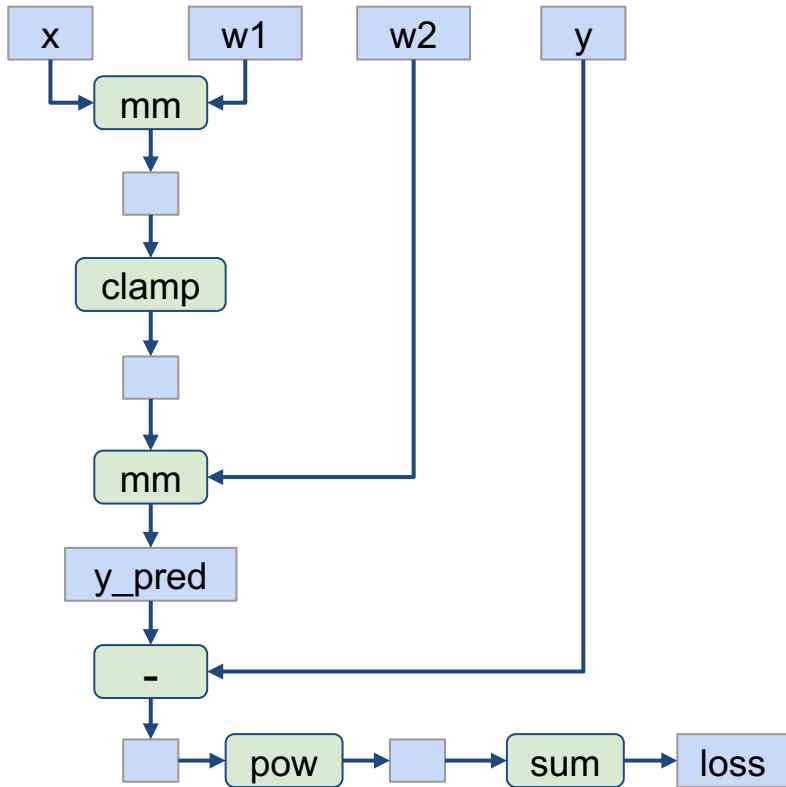
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND  
perform computation

# PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

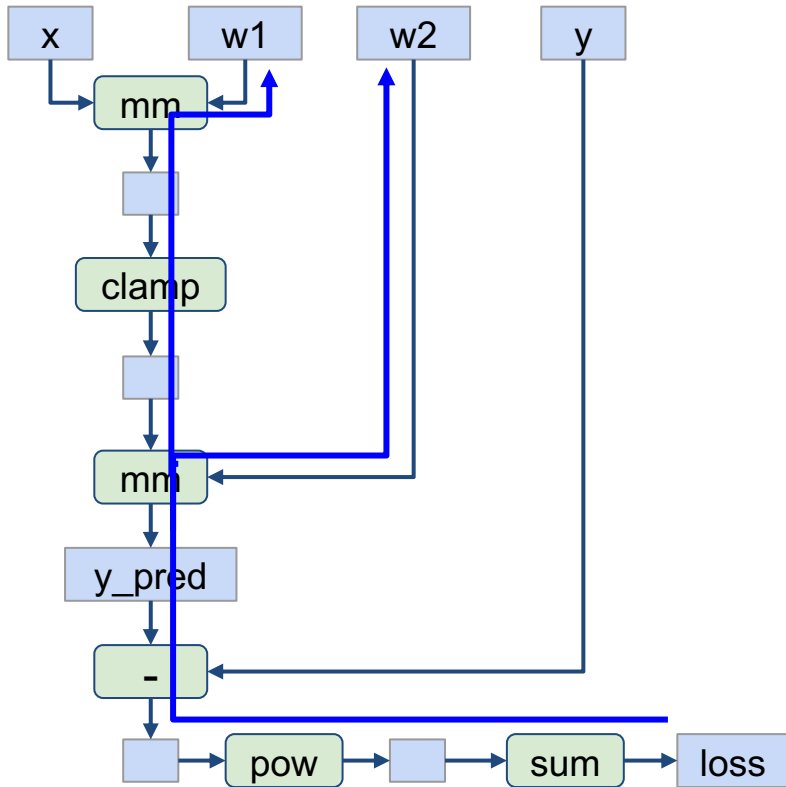
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Build graph data structure AND  
perform computation

# PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Search for path between loss and `w1`, `w2`  
(for backprop) AND perform computation



# PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

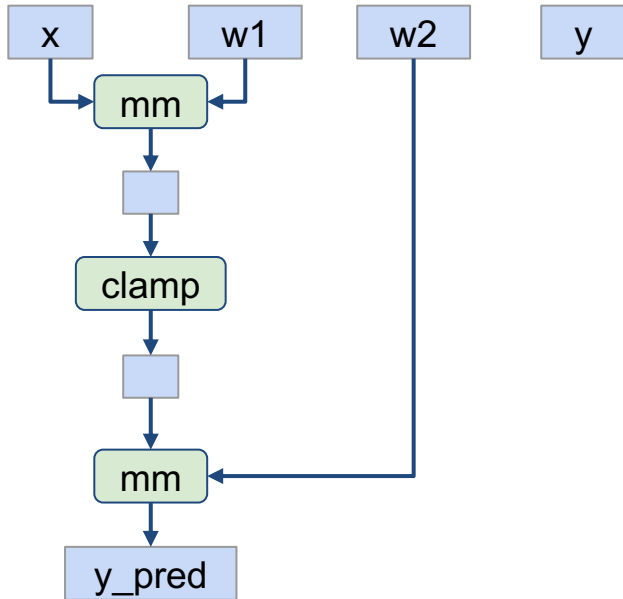
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

# PyTorch: Dynamic Computation Graphs



```
import torch

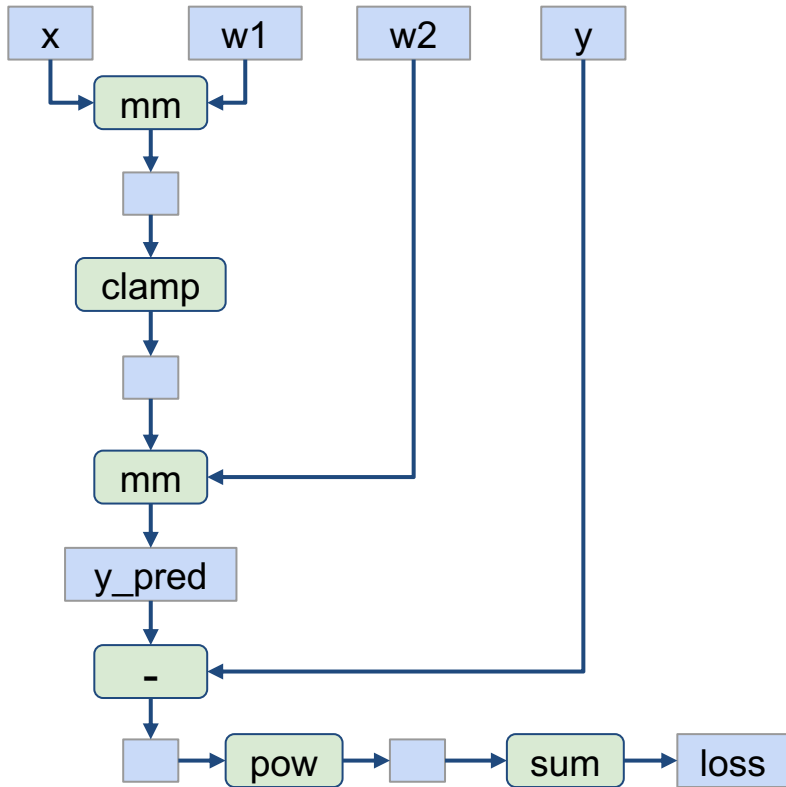
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND  
perform computation

# PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

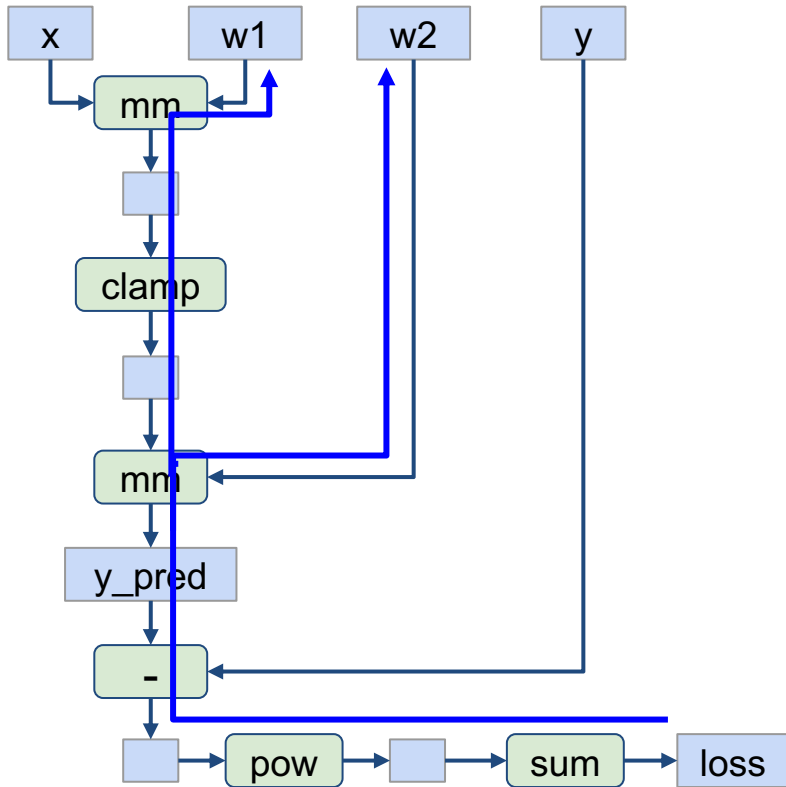
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Build graph data structure AND  
perform computation

# PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Search for path between loss and `w1`, `w2`  
(for backprop) AND perform computation

# PyTorch: Dynamic Computation Graphs

**Building** the graph and **computing** the graph happen at the same time.

Seems inefficient, especially if we are building the same graph over and over again...

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

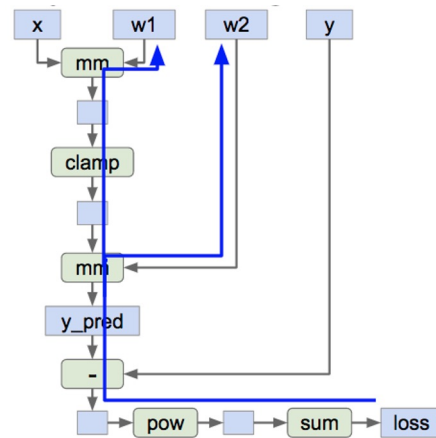
    loss.backward()
```

# Static Computation Graphs

Alternative: **Static** graphs

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration



```
graph = build_graph()
```

```
for x_batch, y_batch in loader:  
    run_graph(graph, x=x_batch, y=y_batch)
```

TensorFlow

# TensorFlow Versions

## Pre-2.0 (1.14 latest)

Default static graph,  
optionally dynamic  
graph (eager mode).

## 2.0+

**Default dynamic graph,**  
optionally static graph.



# TensorFlow: Neural Net (Pre-2.0)

```
import numpy as np
import tensorflow as tf
```

(Assume imports at the top of each snippet)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                    feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

# TensorFlow: Neural Net (Pre-2.0)

First **define**  
computational graph



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Then **run** the graph  
many times



```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                    feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

# TensorFlow: 2.0+ vs. pre-2.0

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
```

Tensorflow 2.0+:

“Eager” Mode by default

```
assert(tf.executing_eagerly())
```

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                    feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 1.13

# TensorFlow: 2.0+ vs. pre-2.0

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2])
```

Tensorflow 2.0+:

“Eager” Mode by default

```
assert(tf.executing_eagerly())
```

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))
```

```
h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                    feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 1.13

# TensorFlow: 2.0+ vs. pre-2.0

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
```

Tensorflow 2.0+:  
“Eager” Mode by default  
`assert(tf.executing_eagerly())`

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                    feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 1.13

# TensorFlow: Neural Net

Convert input numpy  
arrays to TF **tensors**.  
Create weights as  
tf.Variable

```
N, D, H = 64, 1000, 100
```

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
w1 = tf.Variable(tf.random.uniform((D, H))) # weights  
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:  
    h = tf.maximum(tf.matmul(x, w1), 0)  
    y_pred = tf.matmul(h, w2)  
    diff = y_pred - y  
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))  
gradients = tape.gradient(loss, [w1, w2].)
```

# TensorFlow: Neural Net

Use `tf.GradientTape()`  
context to build  
**dynamic** computation  
graph.

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2])
```

# TensorFlow: Neural Net

All forward-pass operations in the contexts (including function calls) gets traced for computing gradient later.

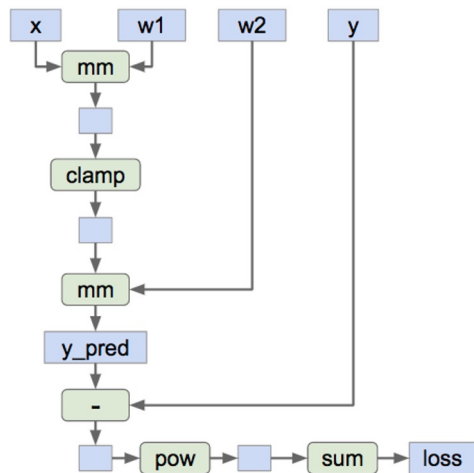
```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2]).
```



# TensorFlow: Neural Net



Forward pass

```
N, D, H = 64, 1000, 100
```

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:
```

```
h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2]).
```

# TensorFlow: Neural Net

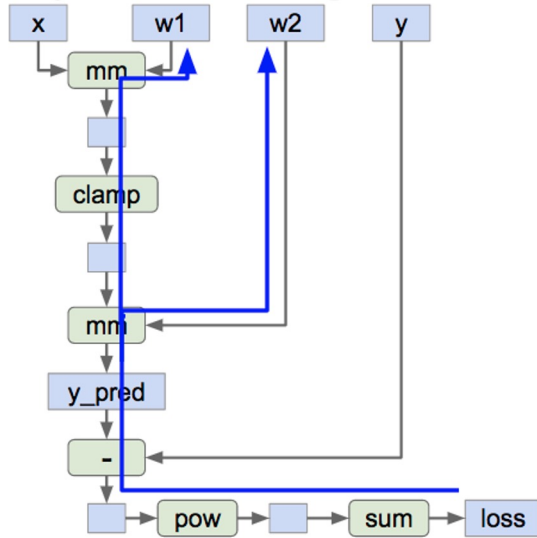
tape.gradient() uses the  
traced computation  
graph to compute  
gradient for the weights

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
```

# TensorFlow: Neural Net



Backward pass

```
N, D, H = 64, 1000, 100
```

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
```

# TensorFlow: Neural Net

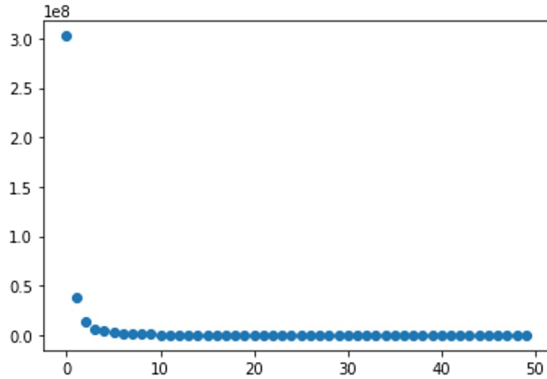
```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
        gradients = tape.gradient(loss, [w1, w2])
        w1.assign(w1 - learning_rate * gradients[0])
        w2.assign(w2 - learning_rate * gradients[1])
```

**Train the network:** Run the training step over and over, use gradient to update weights

# TensorFlow: Neural Net



**Train the network:** Run the training step over and over, use gradient to update weights

```
N, D, H = 64, 1000, 100
```

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
        gradients = tape.gradient(loss, [w1, w2])
        w1.assign(w1 - learning_rate * gradients[0])
        w2.assign(w2 - learning_rate * gradients[1])
```

# TensorFlow: Optimizer

Can use an **optimizer** to compute gradients and update weights

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

optimizer = tf.optimizers.SGD(1e-6)

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.matmul(x, w1), 0
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
        gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2])).
```

# @tf.function: compile static graph

tf.function decorator  
(implicitly) compiles  
python functions to  
static graph for better  
performance

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_func(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

for t in range(50):
    with tf.GradientTape() as tape:
        y_pred, loss = model_func(x, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

# @tf.function: compile static graph

Here we compare the forward-pass time of the same model under dynamic graph mode and static graph mode

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
```

```
print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))
```

```
dynamic graph: 0.02520249200000535
static graph: 0.03932226699998864
```



# @tf.function: compile static graph

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))
```

```
dynamic graph: 0.02520249200000535
static graph: 0.03932226699998864
```

# @tf.function: compile static graph

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

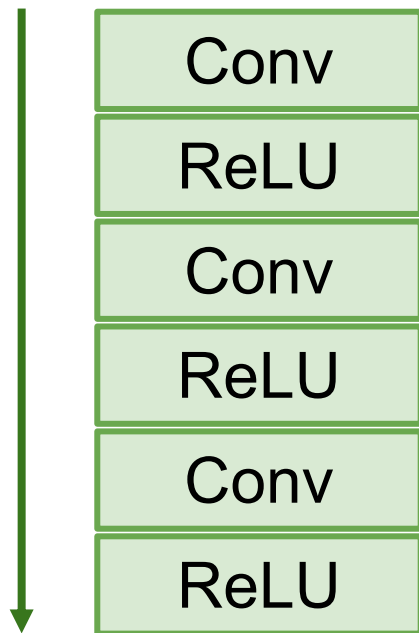
print("dynamic graph:", timeit.timeit(lambda: model_dynamic(x, y), number=1000))
print("static graph:", timeit.timeit(lambda: model_static(x, y), number=1000))
```

```
dynamic graph: 2.3648411540000325
static graph: 1.1723986679999143
```

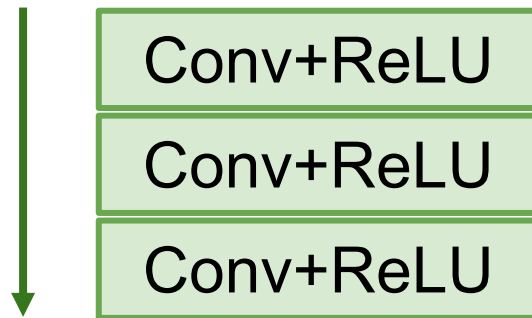
# Static vs Dynamic: Optimization

With static graphs, framework can **optimize** the graph for you before it runs!

The graph you wrote



Equivalent graph with **fused operations**



# Static PyTorch: TorchScript

```
graph(%self.1 :
  __torch__.torch.nn.modules.module.__torch_mangle_4.Module,
    %input : Float(3, 4),
    %h : Float(3, 4)):
  %19 :
  __torch__.torch.nn.modules.module.__torch_mangle_3.Module =
  prim::GetAttr[name="linear"](%self.1)
  %21 : Tensor =
  prim::CallMethod[name="forward"](%19, %input)
  %12 : int = prim::Constant[value=1]() #
  <ipython-input-40-26946221023e>:7:0
  %13 : Float(3, 4) = aten::add(%21, %h, %12) #
  <ipython-input-40-26946221023e>:7:0
  %14 : Float(3, 4) = aten::tanh(%13) #
  <ipython-input-40-26946221023e>:7:0
  %15 : (Float(3, 4), Float(3, 4)) =
  prim::TupleConstruct(%14, %14)
  return (%15)
```

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

my_cell = MyCell()
x, h = torch.rand(3, 4), torch.rand(3, 4)
traced_cell = torch.jit.trace(my_cell, (x, h))
print(traced_cell.graph)
traced_cell(x, h)
```

Build static graph with `torch.jit.trace`

# Static PyTorch: torch.compile()

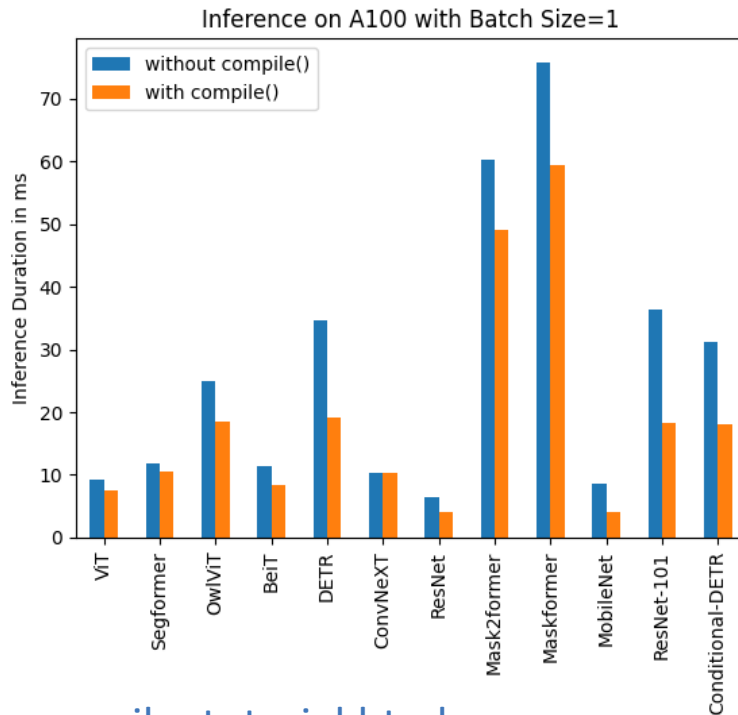
Applies a suite of kernel optimization techniques by analyzing your computation graph. Optimizations include CUDA graphs, kernel fusion, and pattern matching (e.g., flash attention).

```
def foo(x, y):  
    a = torch.sin(x)  
    b = torch.cos(y)  
    return a + b  
opt_foo1 = torch.compile(foo)  
print(opt_foo1(torch.randn(10, 10), torch.randn(10, 10)))
```

Curious? Read more here:

[https://pytorch.org/tutorials/intermediate/torch\\_compile\\_tutorial.html](https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html)

<https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/>



# PyTorch vs TensorFlow, Static vs Dynamic

## **PyTorch**

Dynamic Graphs

Static: TorchScript,  
torch.compile()

## **TensorFlow**

Dynamic Graphs

Static: @tf.function

# Static vs Dynamic: Serialization

## **Static**

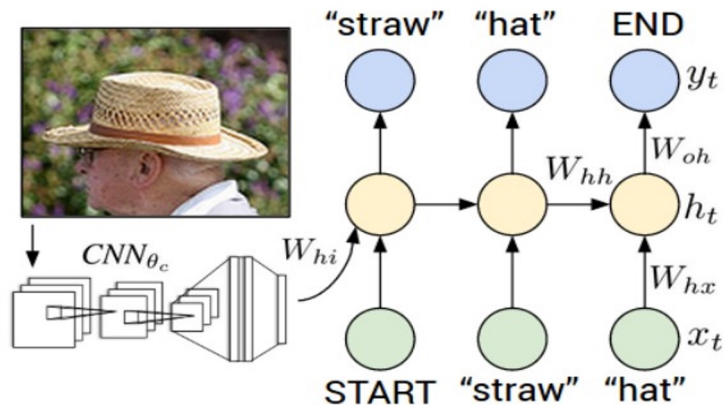
Once graph is built, can **serialize** it and run it without the code that built the graph!

## **Dynamic**

Graph building and execution are intertwined, so always need to keep code around

# Dynamic Graph Applications

- Recurrent networks

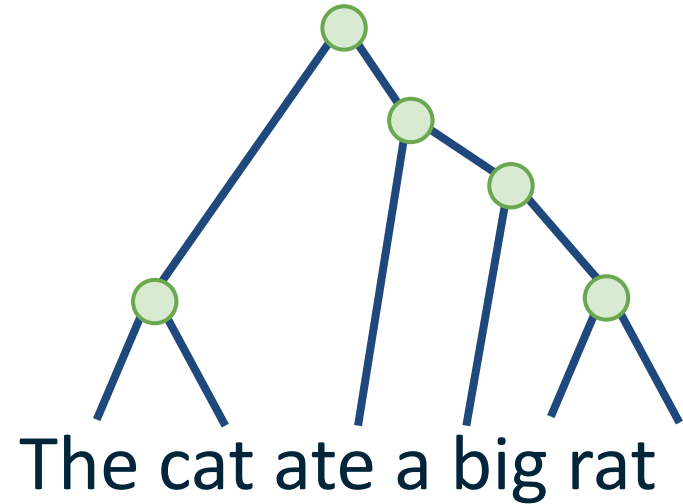


Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015  
Figure copyright IEEE, 2015. Reproduced for educational purposes.



# Dynamic Graph Applications

- Recurrent networks
- Recursive networks



# Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular networks

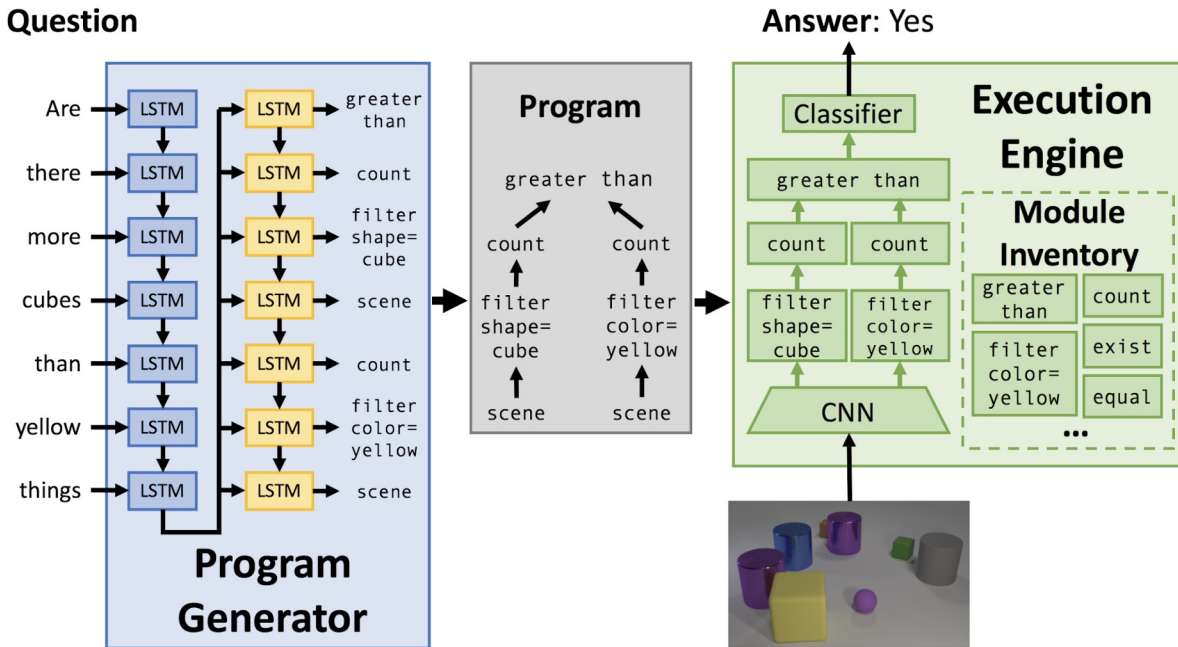


Figure copyright Justin Johnson, 2017. Reproduced with permission.

Andreas et al, "Neural Module Networks", CVPR 2016

Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

Johnson et al, "Inferring and Executing Programs for Visual Reasoning", ICCV 2017

# Dynamic Graph Applications

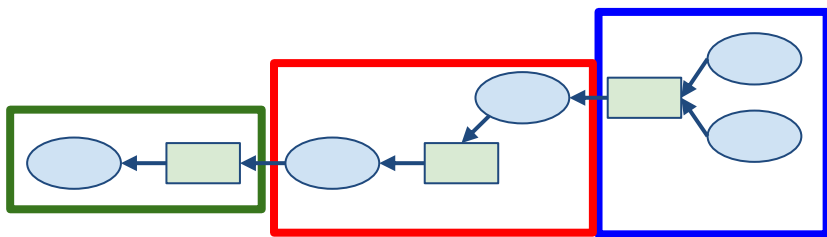
- Recurrent networks
- Recursive networks
- Modular Networks
- (Your creative idea here)

# Model Parallel vs. Data Parallel

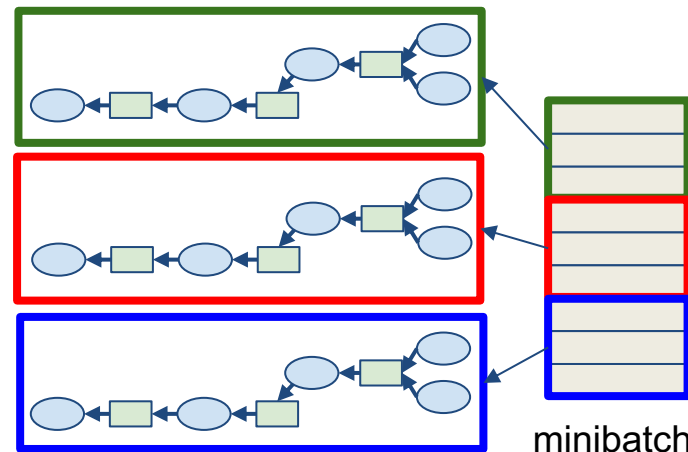
Model parallelism:  
split computation  
graph into parts &  
distribute to GPUs/  
nodes



Data parallelism: split  
minibatch into chunks &  
distribute to GPUs/ nodes



Model Parallel



Data Parallel

# PyTorch: Data Parallel

`nn.DataParallel`

Pro: Easy to use (just wrap the model and run training script as normal)

Con: Single process & single node. Can be bottlenecked by CPU with large number of GPUs (8+).

`nn.DistributedDataParallel`

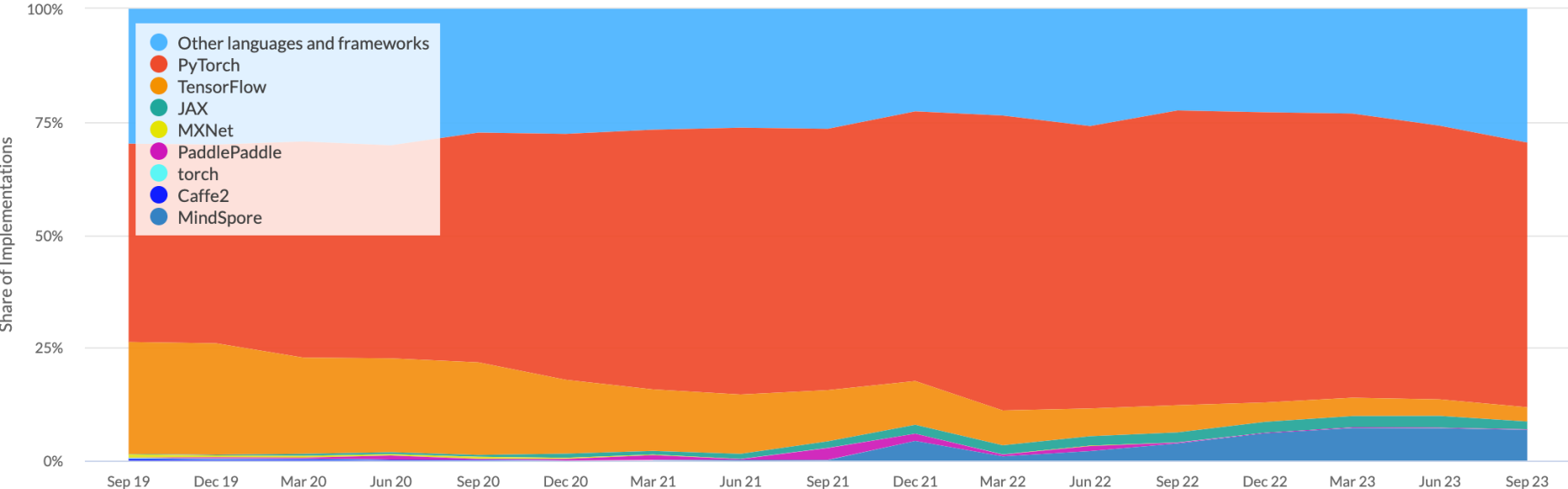
Pro: Multi-nodes & multi-process training

Con: Need to hand-designate device and manually launch training script for each process / nodes.

Horovod (<https://github.com/horovod/horovod>): Supports both PyTorch and TensorFlow

<https://pytorch.org/docs/stable/nn.html#dataparallel-layers-multi-gpu-distributed>

# PyTorch vs. TensorFlow



# My Advice:

**PyTorch** is my personal favorite. Clean API, native dynamic graphs make it very easy to develop and debug. Can build model using the default API then compile static graph using JIT. Almost all academic research uses PyTorch

**TensorFlow's** syntax became a lot more intuitive after 2.0. Not perfect but still has a wide industry usage. Can use same framework for research and production.

Explore other frameworks such as **JAX** if you are curious!