# Lecture 16 – Recurrent Neural Networks

Prof. Makar

# Today:

- Recap: Convolutions and Max Pooling
- Final layer
- Training CNNs

⚠ This lecture is low on TL;DPAs

- Recurrent neural networks (RNNs)
  - Example tasks
  - Motivation: why do we need yet another architecture?
  - Vanilla RNN
  - LSTMs
  - Transformers and self attention

# 2D example: Max pooling

Output from convolutional layer & ReLU:

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Max pooling: returns max of its arguments

- E.g size $3 \times 3$
- E.g stride 3

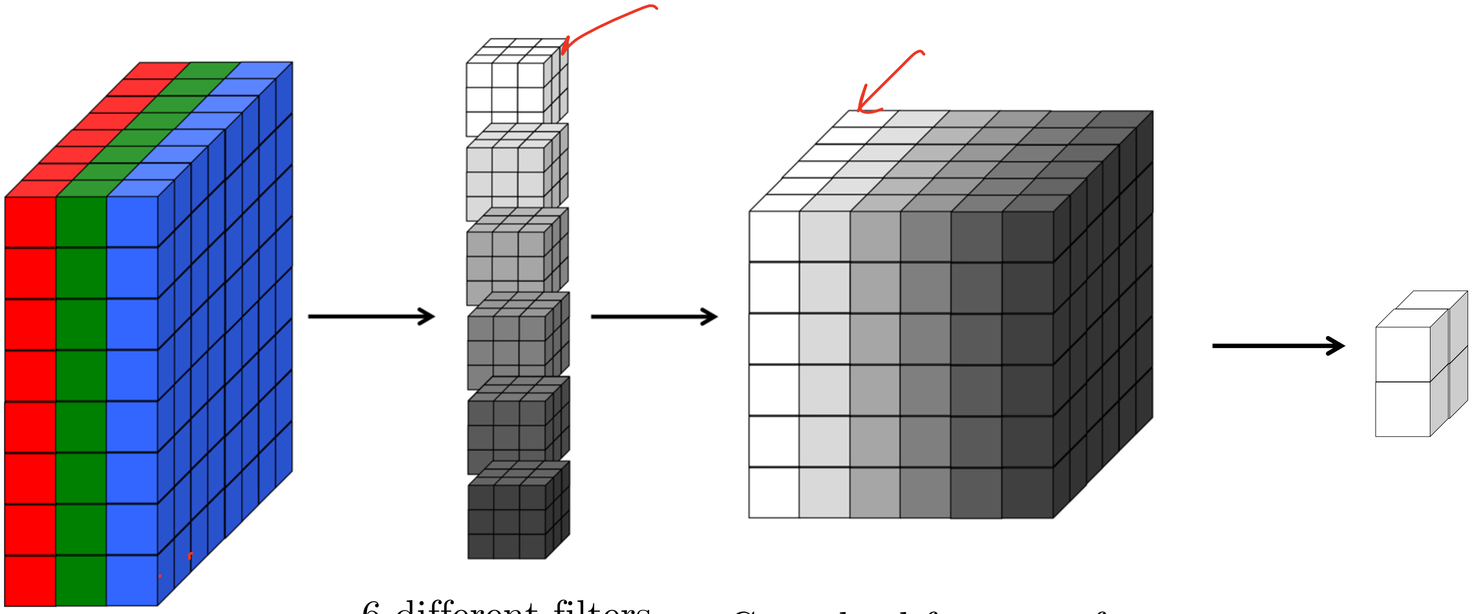
After max pooling:

| | |
|---|---|
| 0 | 1 |
| 1 | 0 |

No learnable weights in this layer

Depth remains the same, but size shrinks

# Max pooling leaves depth unchanged

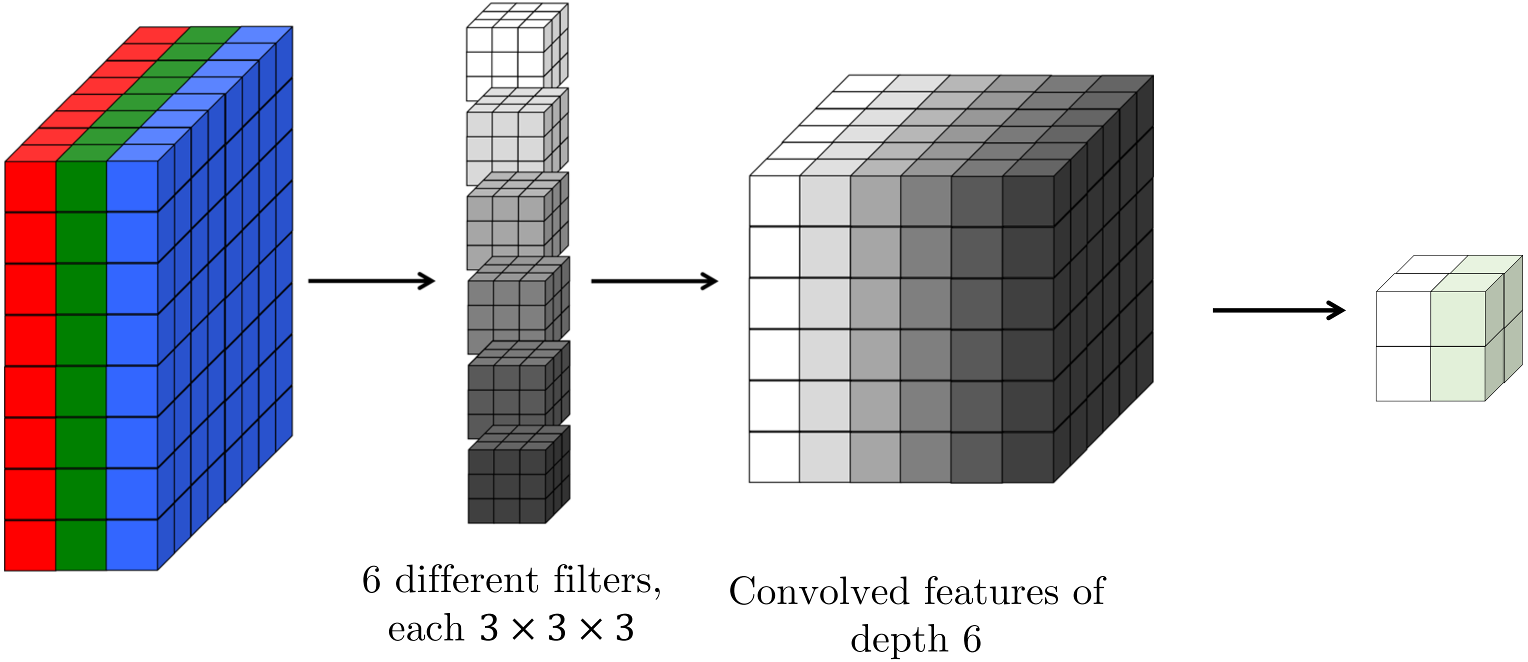


$3 \times 8 \times 8$

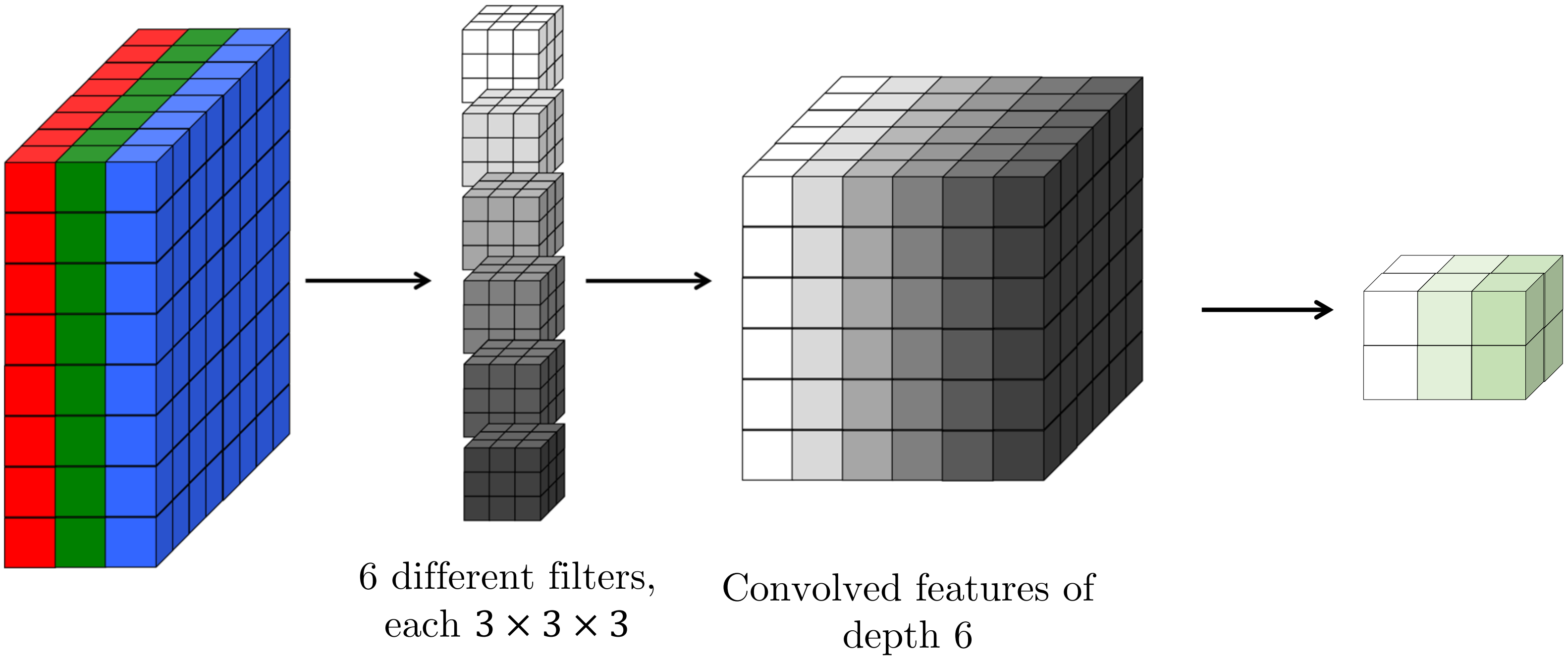
6 different filters, each $3 \times 3 \times 3$

Convolved features of depth 6

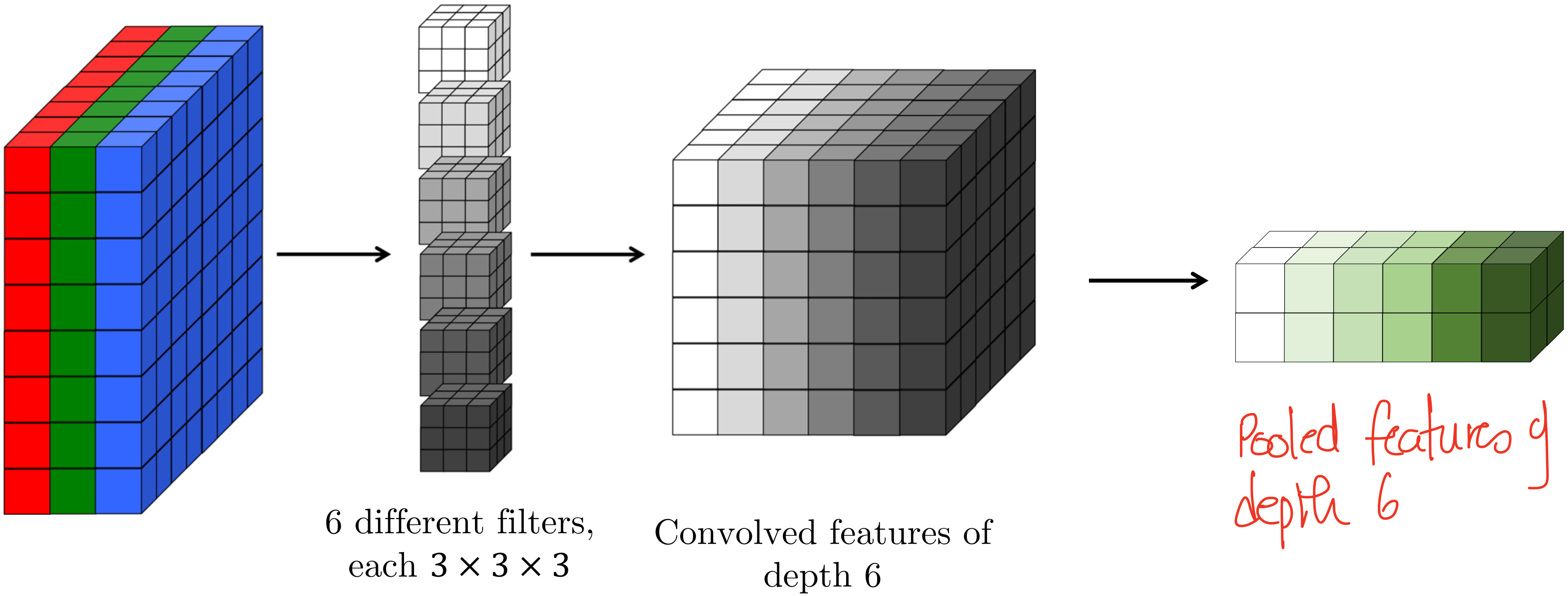
# Max pooling leaves depth unchanged



# Max pooling leaves depth unchanged

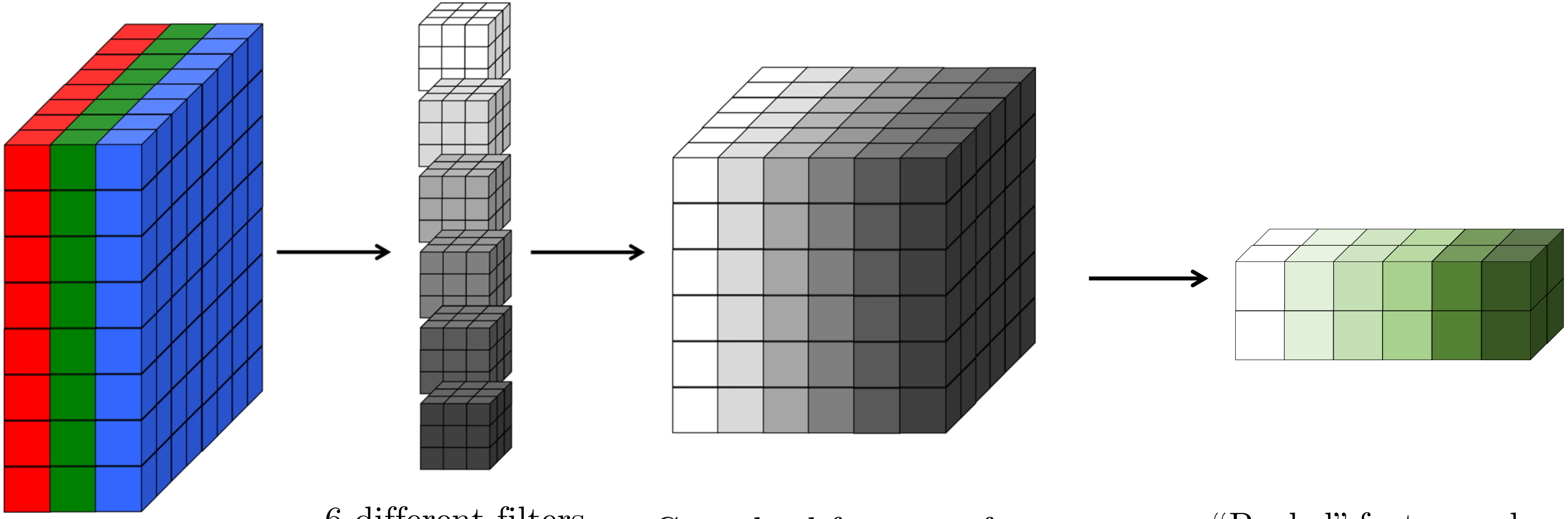


# Max pooling leaves depth unchanged




$6 \times \underline{W} \times \underline{H}$

# Max pooling leaves depth unchanged



6 different filters, each $3 \times 3 \times 3$

Convolved features of depth 6

“Pooled” features also of depth 6

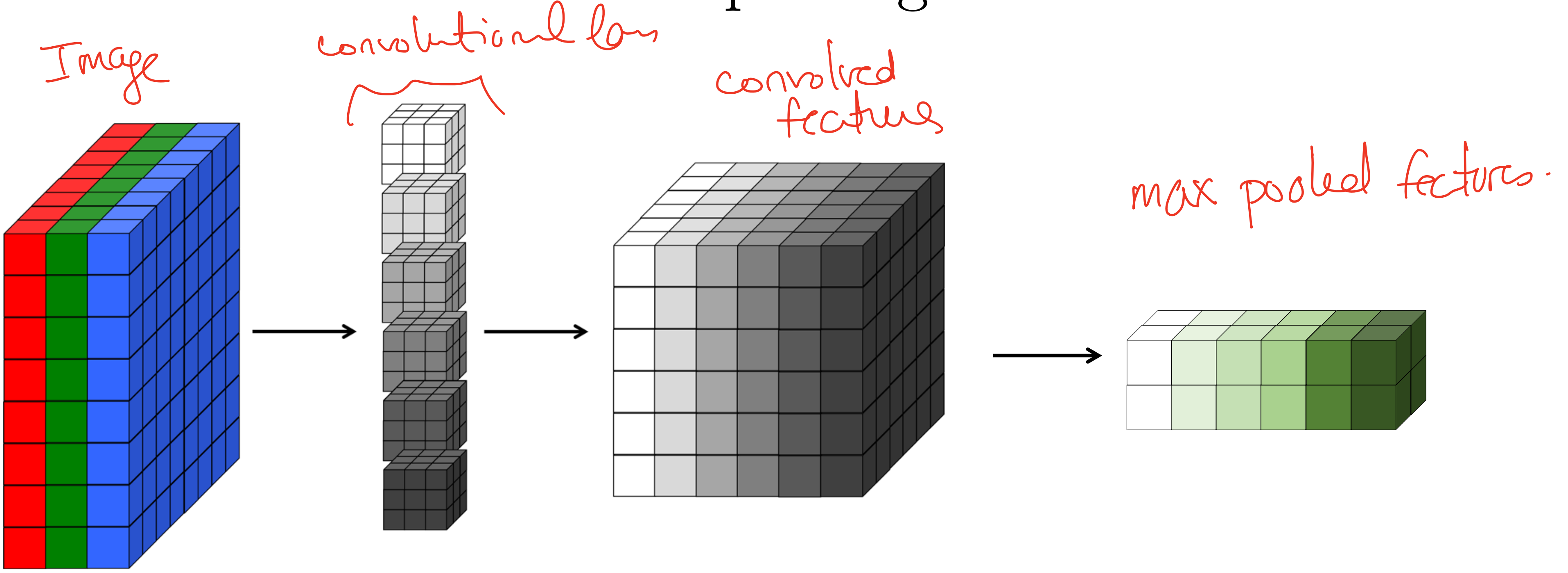
**TL;DPA:**

1. Max pooling allows us to consolidate/sharpen what we've learned from the convolved features

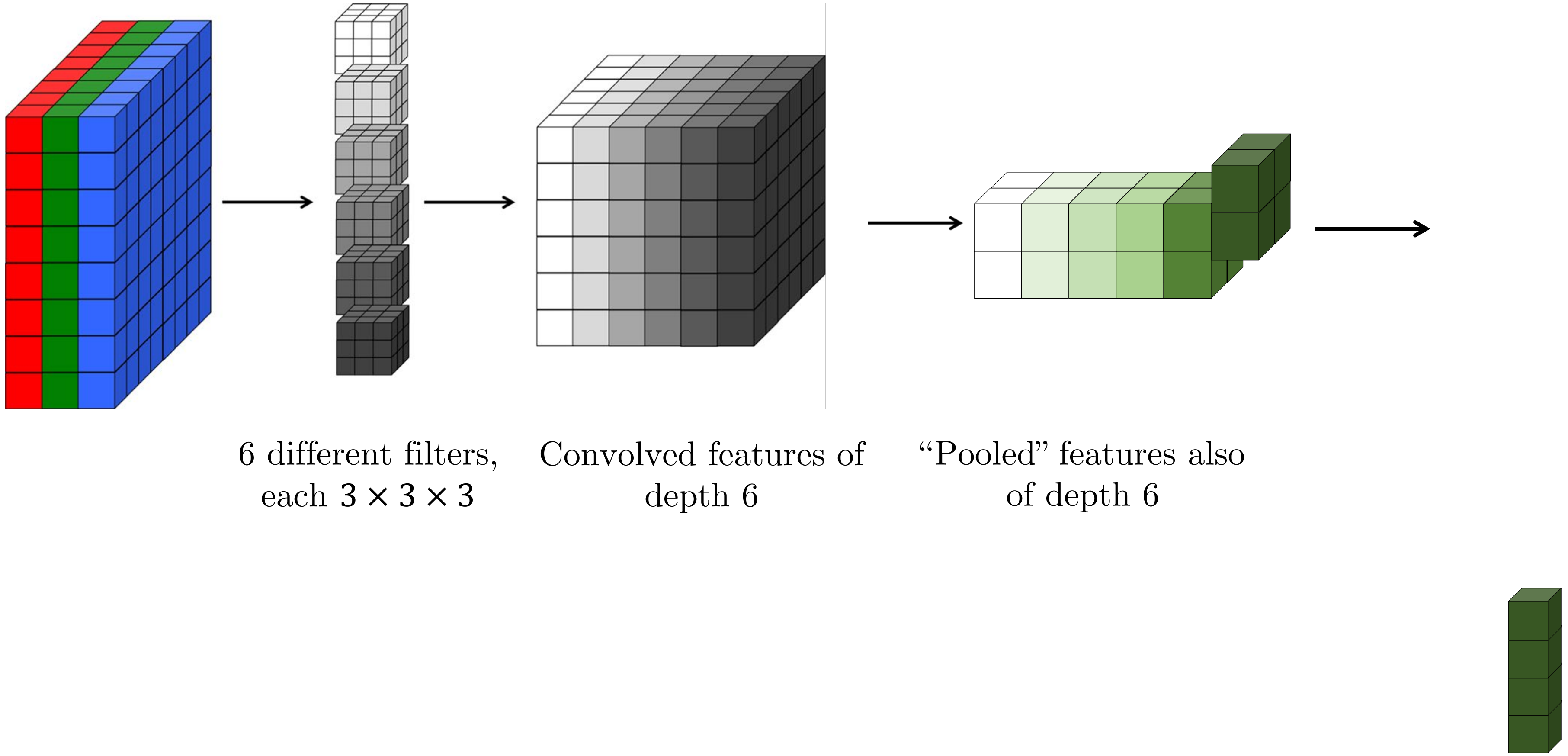
2. It's the main mechanism that gives us invariance (e.g., to translations)

3. It's not a real layer (no trainable parameters)

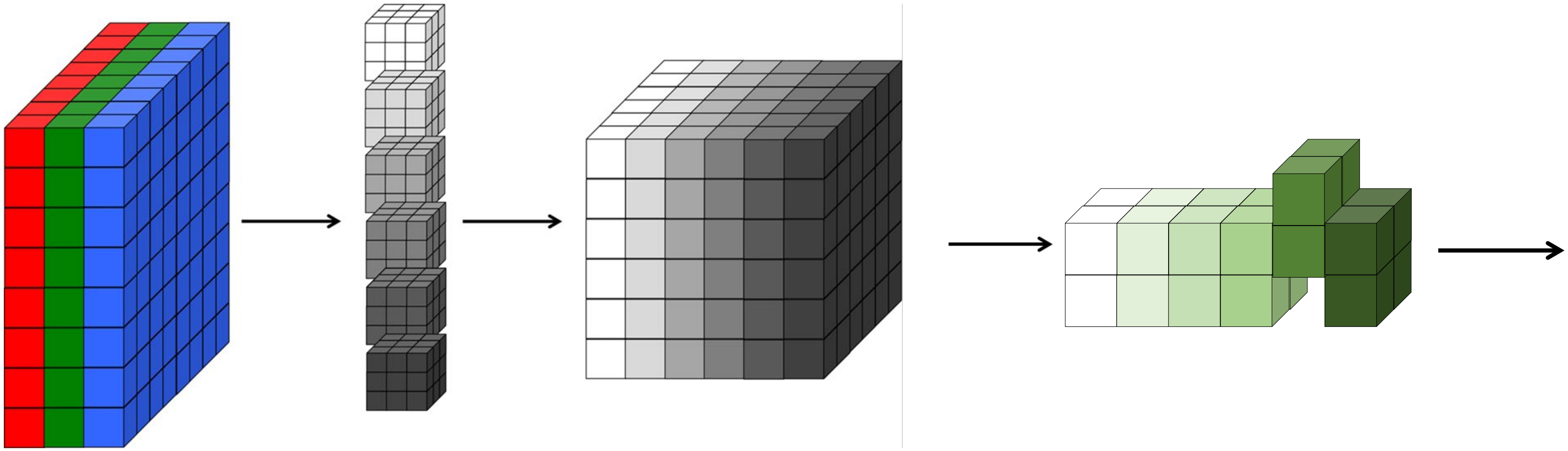
# 3D convolution and max pooling



# Flattening


6 different filters,
each $3 \times 3 \times 3$
Convolved features of
depth 6
"Pooled" features also
of depth 6

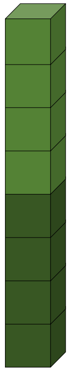
# Flattening



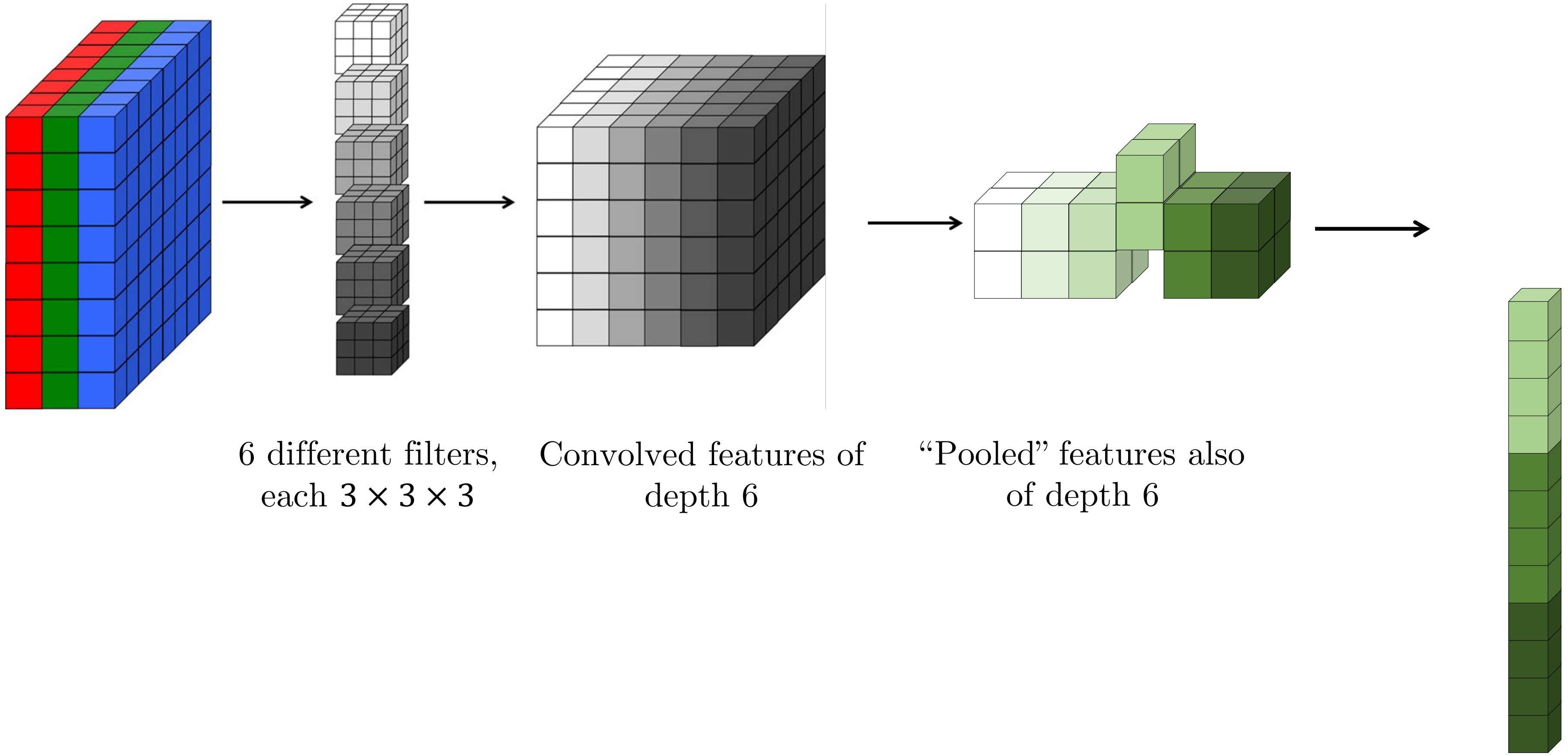
6 different filters, each $3 \times 3 \times 3$

Convolved features of depth 6

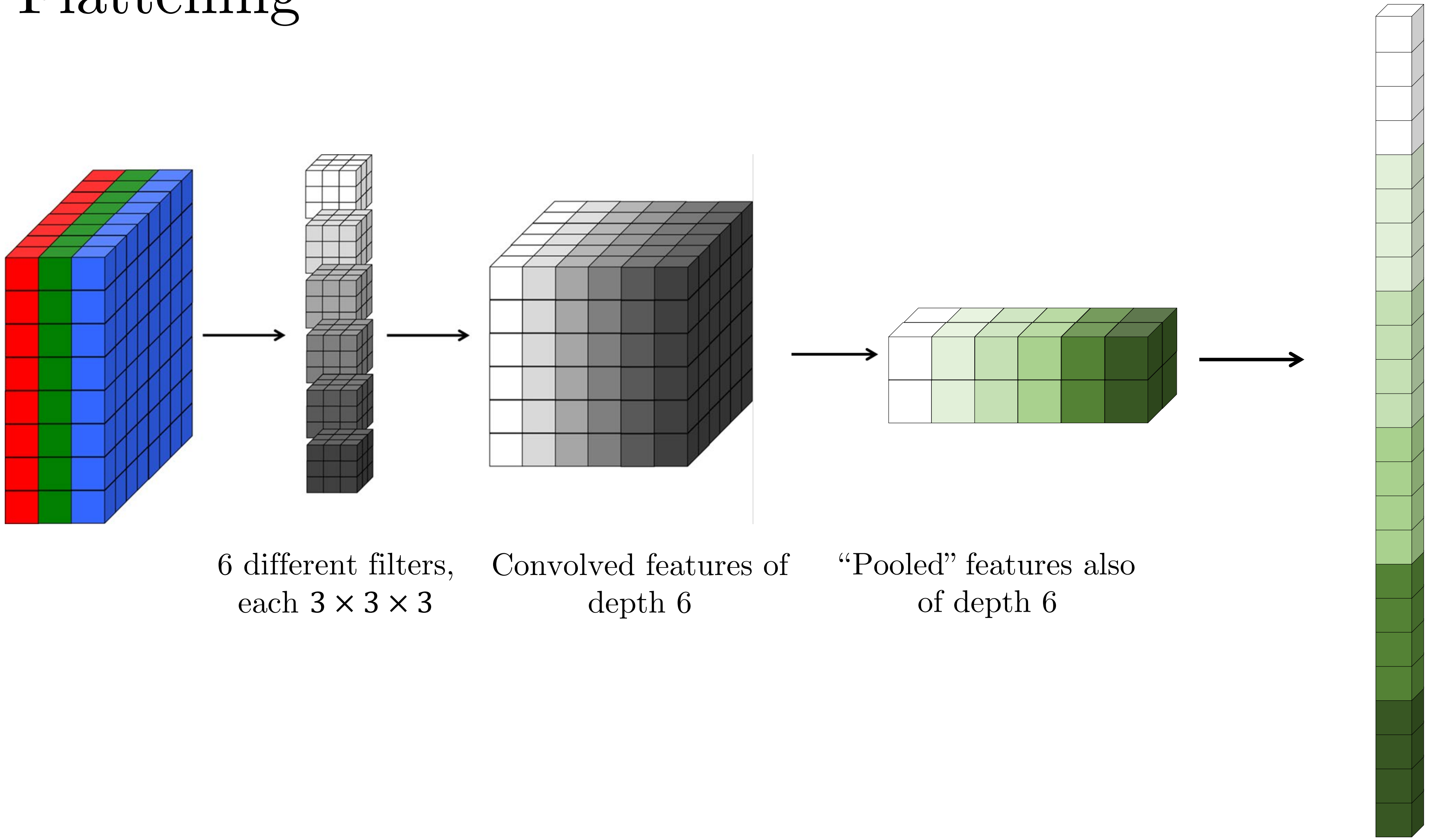
"Pooled" features also of depth 6



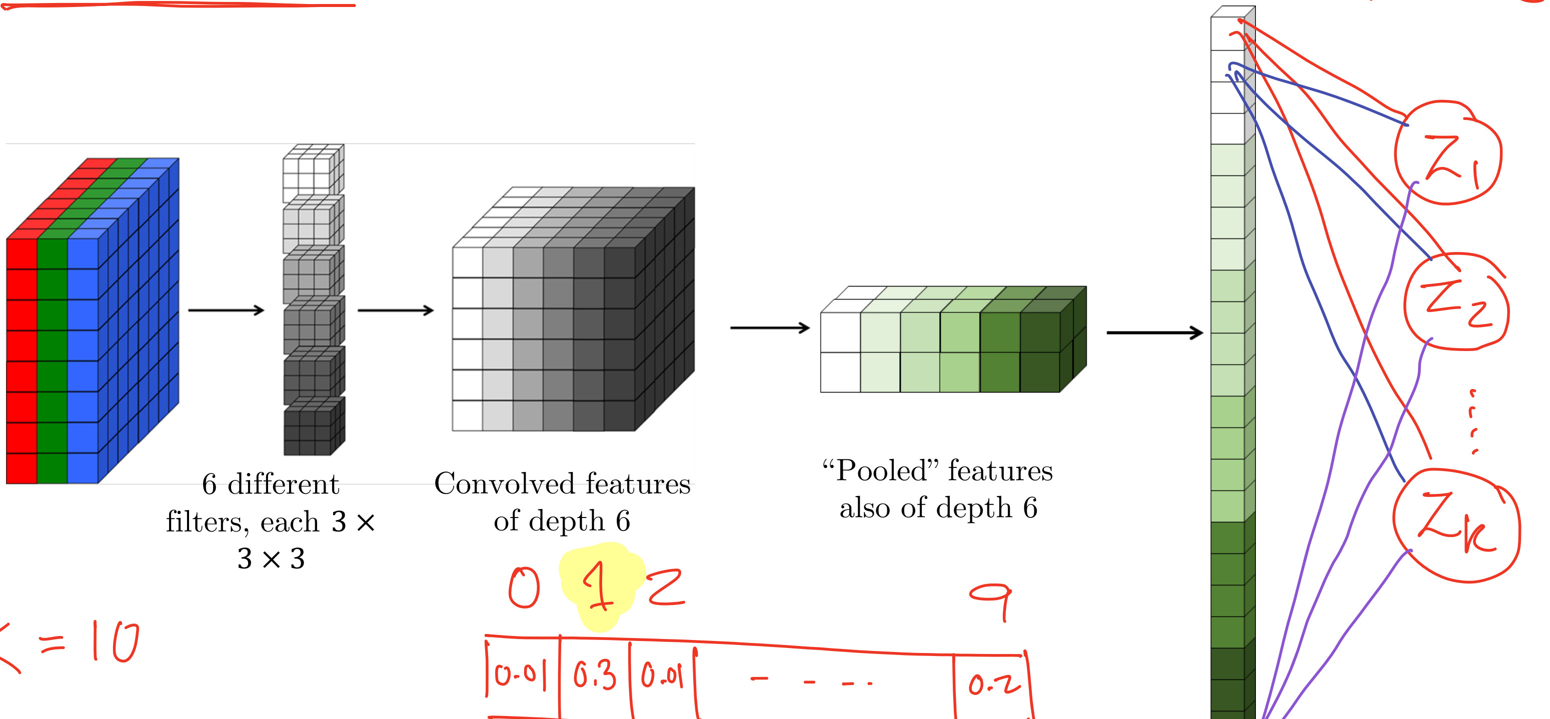
# Flattening


6 different filters,
each $3 \times 3 \times 3$
Convolved features of
depth 6
"Pooled" features also
of depth 6

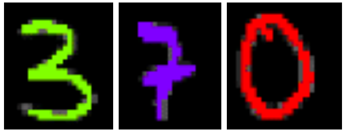
# Flattening


6 different filters,
each $3 \times 3 \times 3$
Convolved features of
depth 6
"Pooled" features also
of depth 6

# Multi-class classification



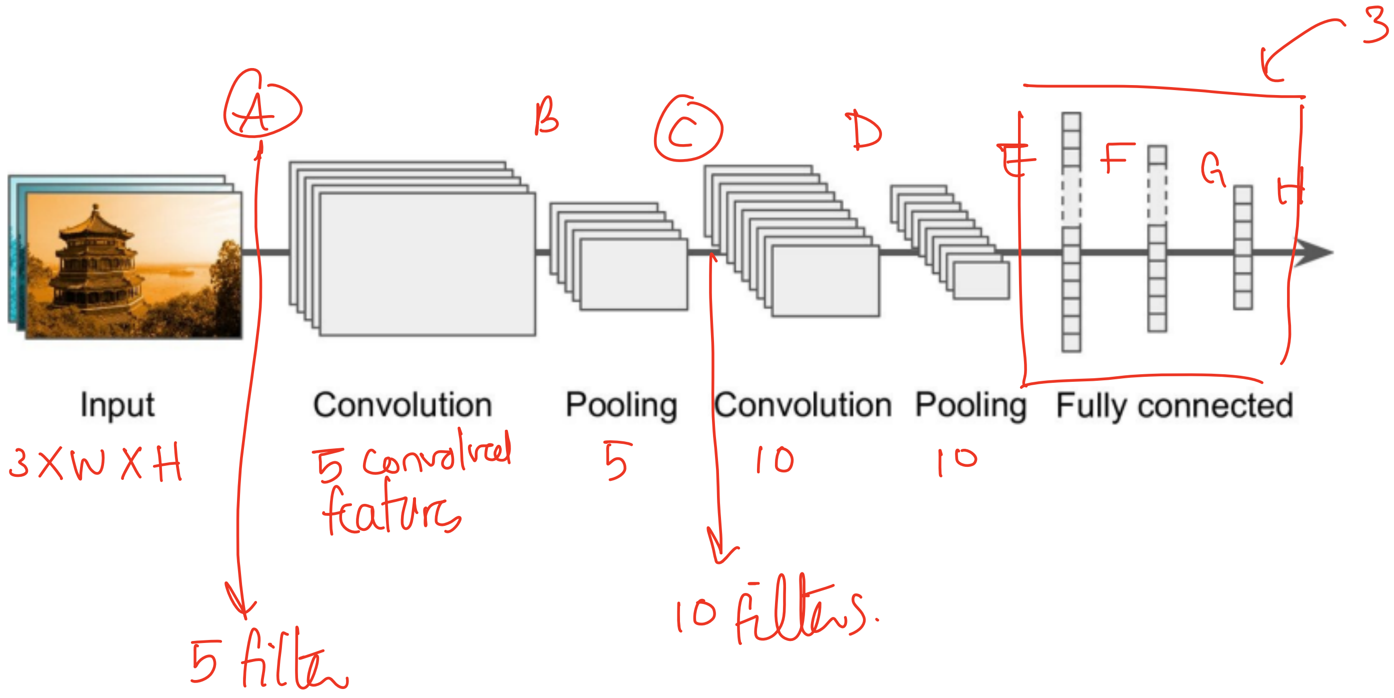

K = 10



softmax function

$$\hat{y}_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

# An example of a CNN architecture



# Check your understanding

- Consider a CNN composed of three convolutional layers, each with 3x3 filters (+bias terms), a stride of 2, and zero (SAME) padding. The lowest layer outputs 100 feature maps, and the middle outputs 200. The top one outputs 400. The input images are colored, with 200x300 pixels. What is the total number of parameters in this CNN?

W x H

a) 6,300 b) 7,000 c) 902,700 d) 902,800 e) 903,400

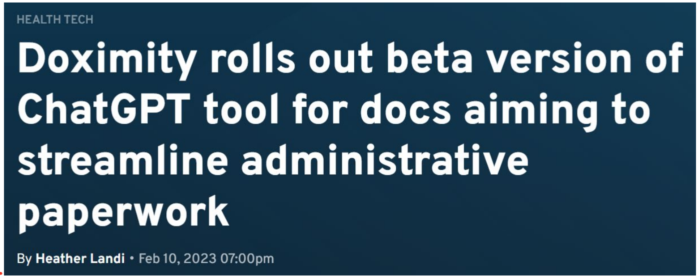
- One filter in the first layer = 3x3x3 + 1 = 28
- All filters in the first layer = 28 x 100 = **2,800** ←
- One filter in the second layer = 100x3x3 + 1 = 901
- All filters in the second layer = 901x200 = **180,200** ←
- One filter in the third layer = 200x3x3 + 1 = 1,801
- All filters in the third layer = 1,801x400 = **720,400** ←
- Total = 2,800 +180,200 + 720,400 = **903,400**

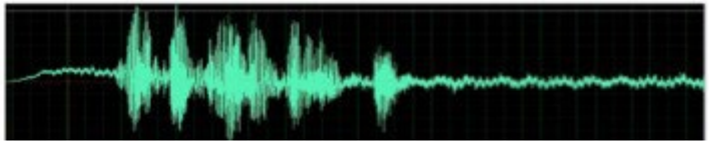
**But if we used a single hidden layer with only 100 neurons, this would be >18M params**

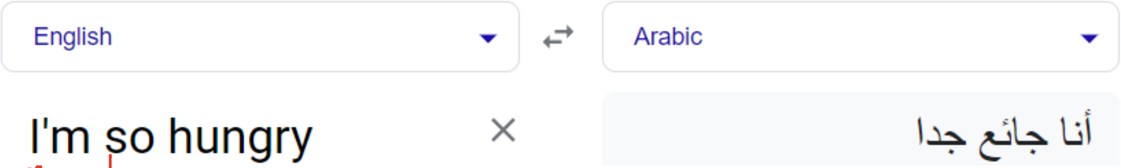
# CNNs: Training

- Initialize parameters → filters + bias terms.
- Forward propagate a training example (image)
  - i.e., convolution, ReLU, pooling and Fully Connected layers
  - compute class probabilities
- Calculate the loss at the output layer
- Use backprop to calculate error contribution of each layer
- Update parameter values to minimize the output error.

# Sequence models

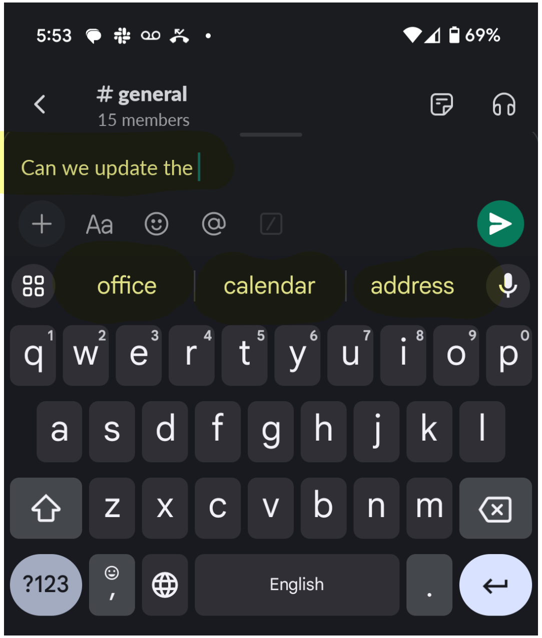



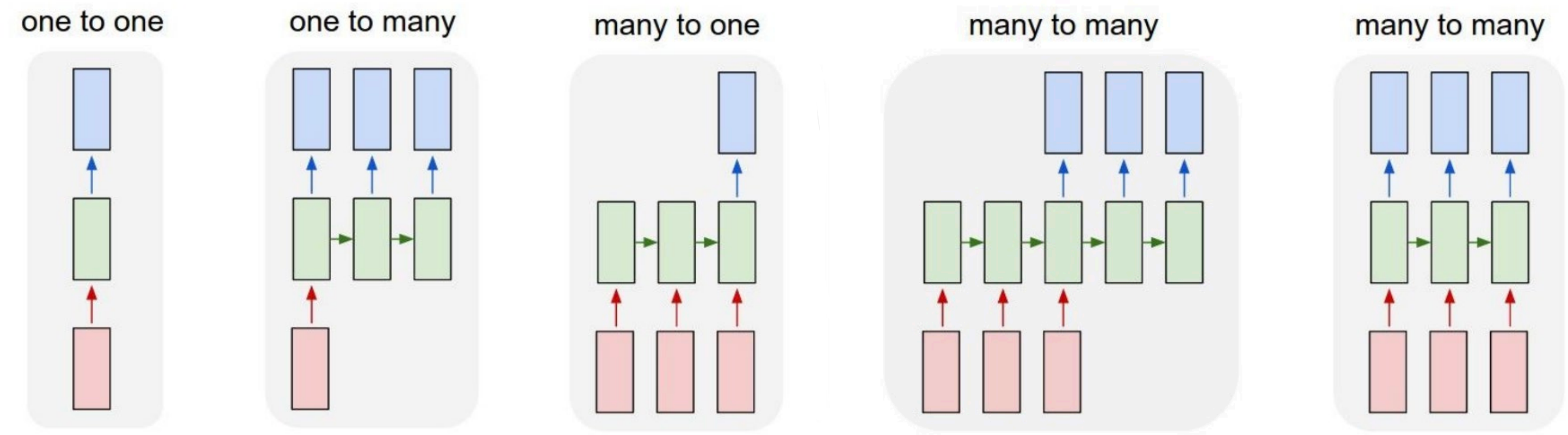



4

3 words



# Sequence Modeling: Types of Tasks




Vanilla classification/ regression

e.g. **image captioning**
Image → sequence of words

e.g. **sentiment classification**
Text → positive/negative

e.g. **translation** English → Arabic

e.g. **video captioning**
Sequence of frames → text sequence

# We need a different kind of NN structure to...

1. Efficiently address varying input size

→ Who is the murderer in "And then there were none" by Jane Austin?
→ Why are barns painted red?

2. Incorporate dependencies over sequences/time

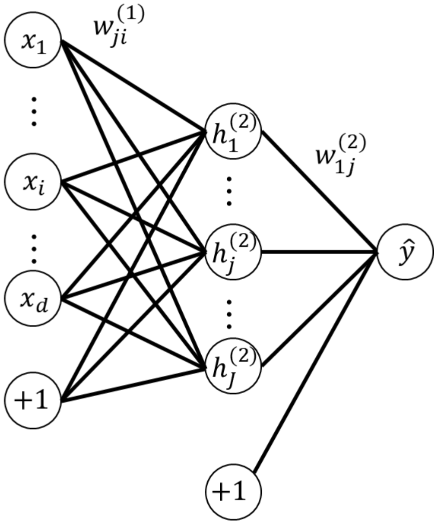
**I was born and raised in Egypt.** Growing up, I had a pet camel called Joe. I had to leave him behind when I came to the US as an undergrad. After graduating with a degree in Math and Economics....I could understand him because **my native language is** ___

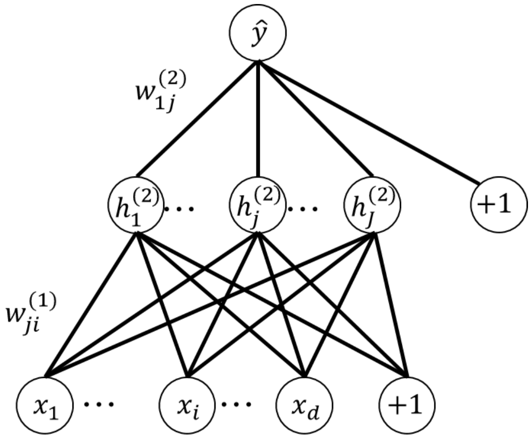
3. Allow shared parameters over time

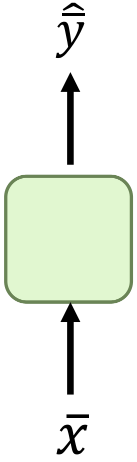
Jiali plays the guitar
Hugh ate a sandwich

- Of course I've been to a football game! I live in Ann ___
- We live in Ann ___ because we work for UM

# Building up to RNNs



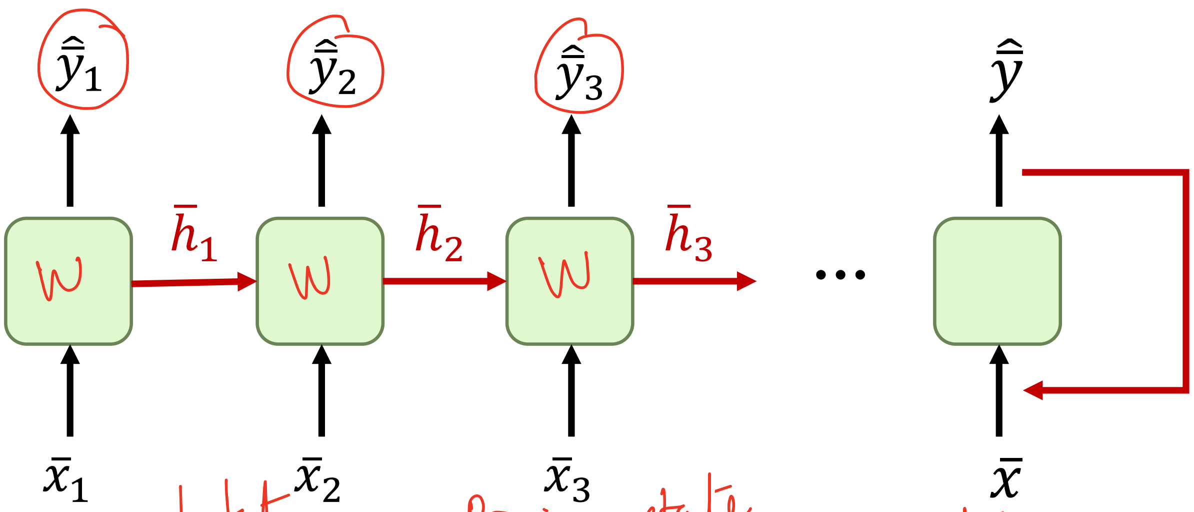



# RNNs (many to many)

1. Efficiently address varying input size
2. Incorporate dependencies over sequences/time
3. Allow shared parameters over time

$\bar{h}_t$: state




current state

Previous state

current input

$$\bar{h}_t = f(\bar{h}_{t-1}, \bar{x}_t)$$

current state

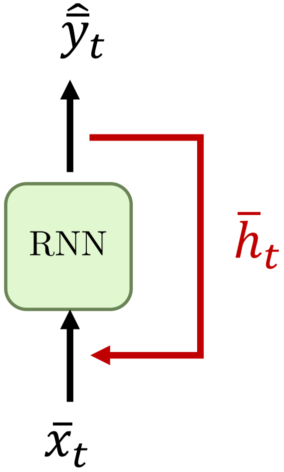
current pred label

$$\hat{\bar{y}}_t = g(\bar{h}_t, \bar{x}_t)$$

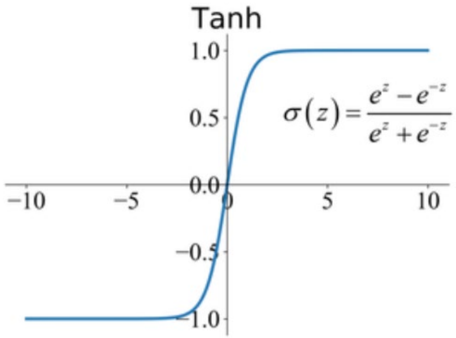
current input

# Vanilla RNN

$\hat{\bar{y}}_t = \hat{\bar{h}}_t$




$$\bar{h}_t = \tanh(W_{hh}\bar{h}_{t-1} + W_{xh}\bar{x}_t)$$
$$\hat{\bar{y}}_t = W_{hy}\bar{h}_t$$



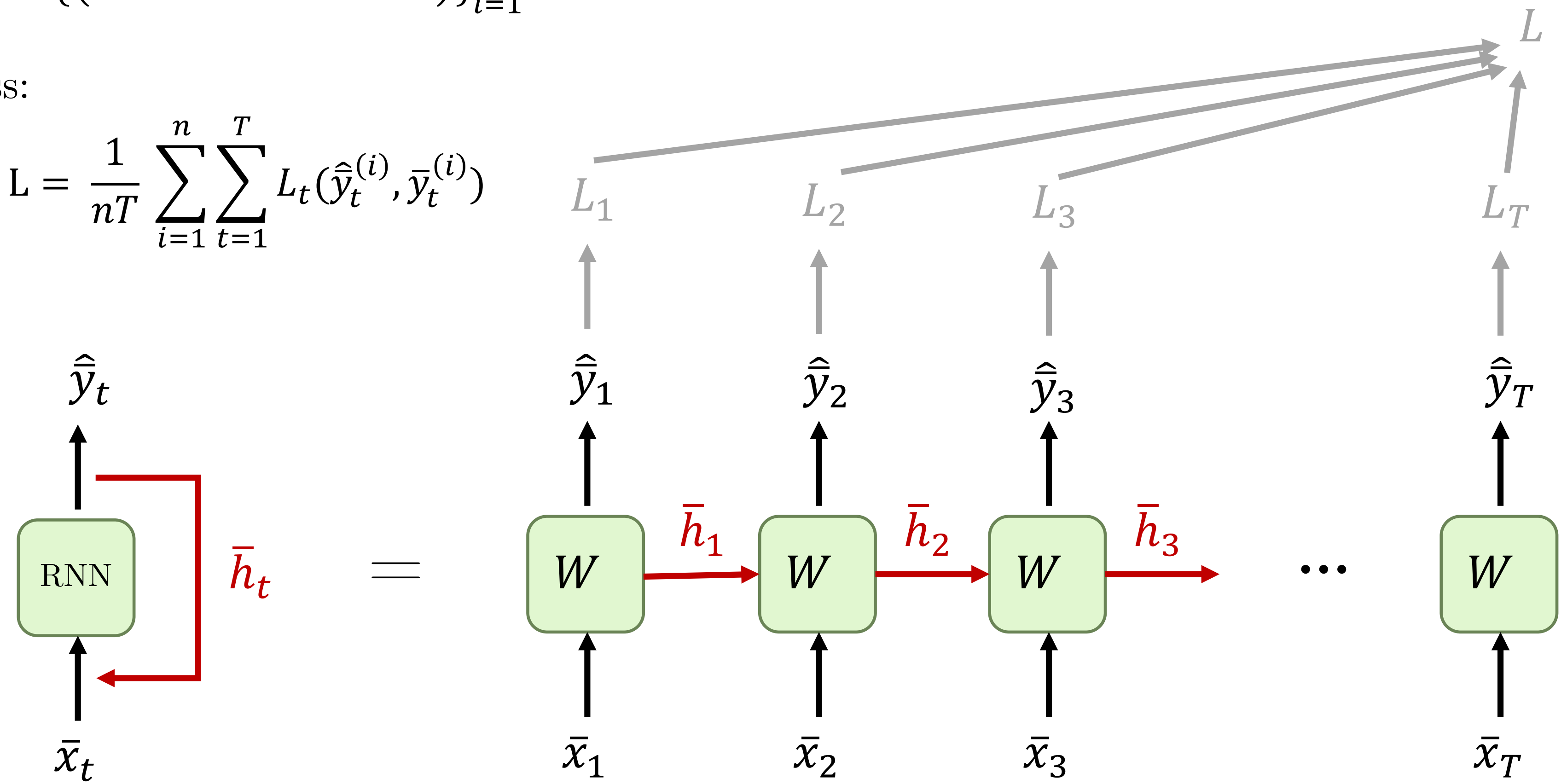

**Note:** The same set of parameters $W_{hh}$, $W_{xh}$, and $W_{hy}$ are used at every time step

# RNN loss: Many to Many

Data: $\left\{\left(\bar{x}_1^{(i)}, \bar{y}_1^{(i)}, \ldots, \bar{x}_T^{(i)}, \bar{y}_T^{(i)}\right)\right\}_{i=1}^{n}$

Loss:

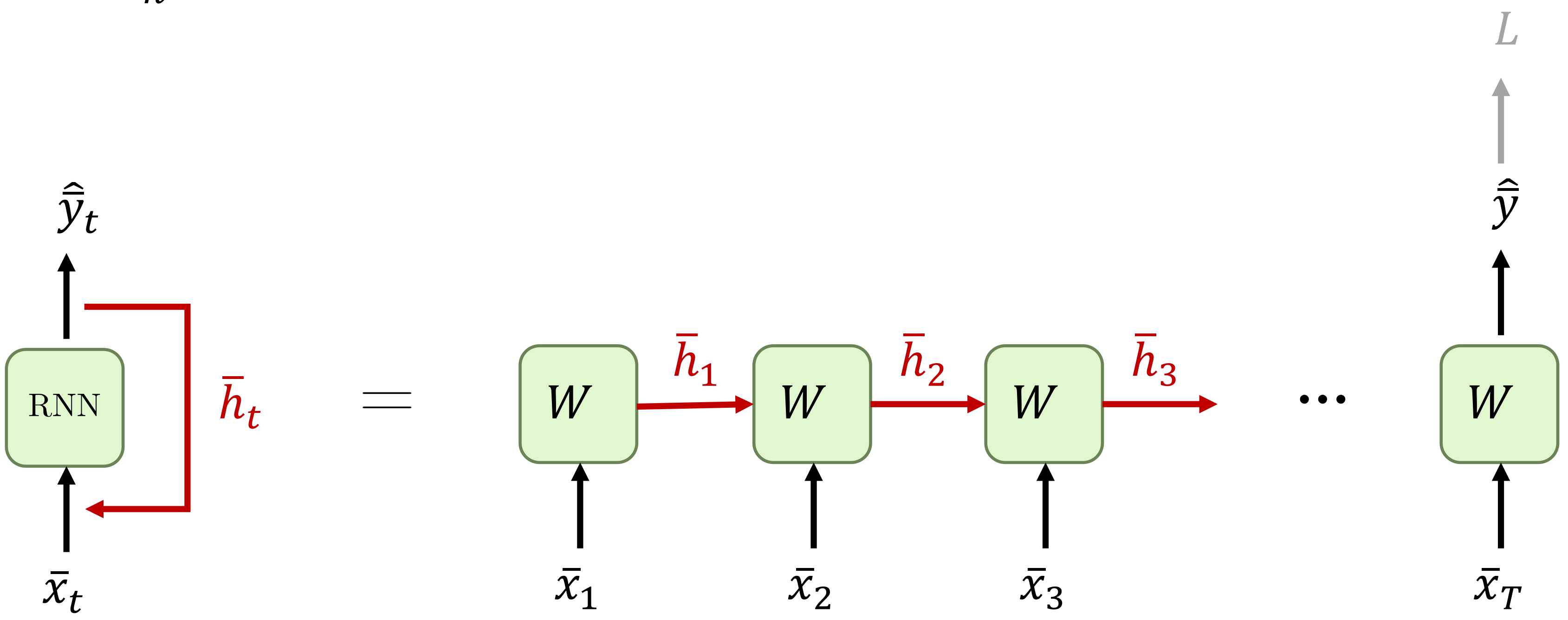
$$\mathrm{L} = \frac{1}{nT}\sum_{i=1}^{n}\sum_{t=1}^{T} L_t(\hat{\bar{y}}_t^{(i)}, \bar{y}_t^{(i)})$$



# RNN loss: Many to One

Data: $\left\{\left(\bar{x}_1^{(i)}, \dots, \bar{x}_T^{(i)}, \bar{y}^{(i)}\right)\right\}_{i=1}^{n}$

Loss: $\mathrm{L} = \frac{1}{n} \sum_{i=1}^{n} L(\hat{\bar{y}}^{(i)}, \bar{y}^{(i)})$

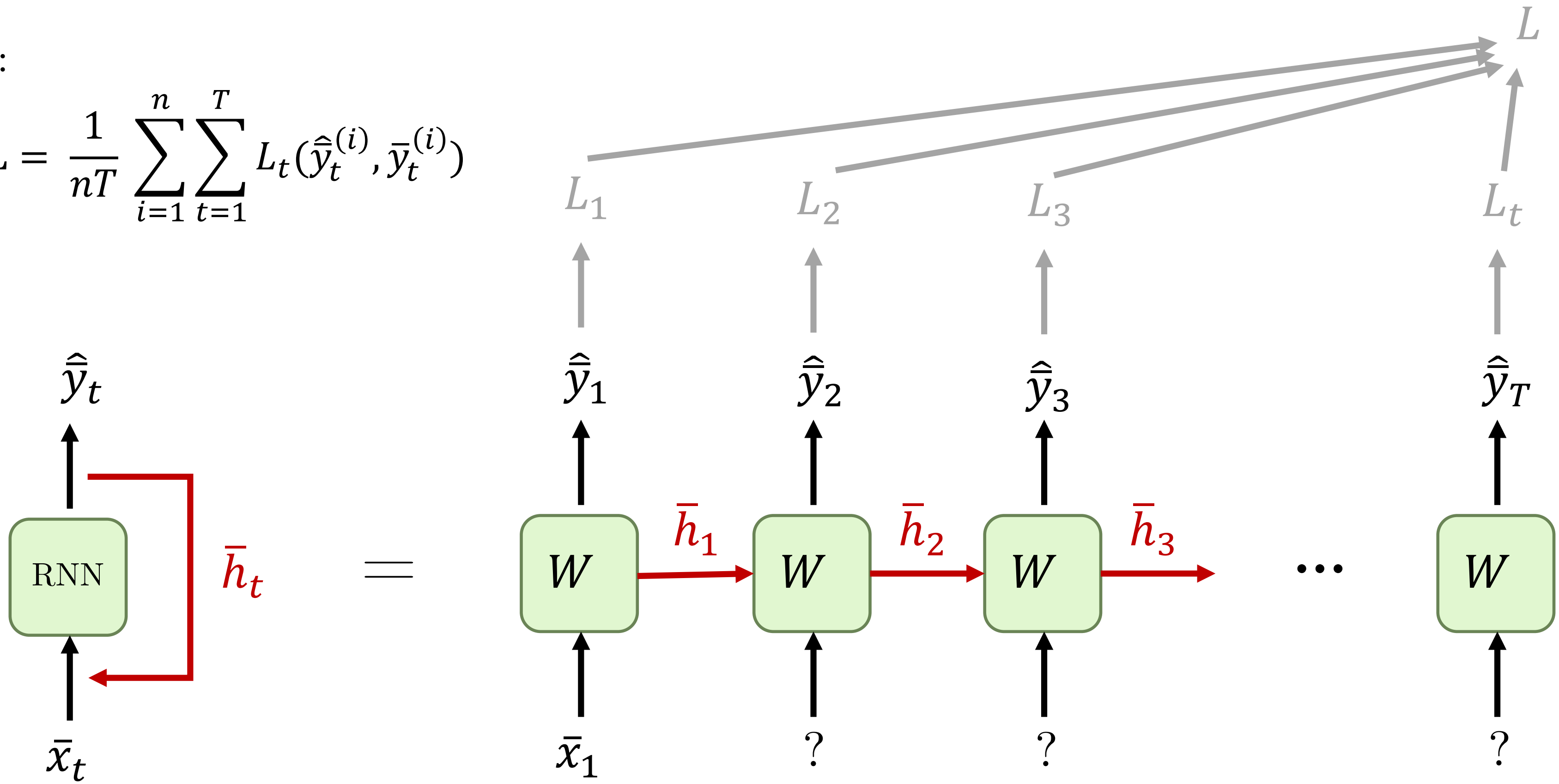


# RNN loss: One to Many

Data: $\left\{\left(\bar{x}^{(i)}, \bar{y}_1^{(i)}, \dots, \bar{y}_T^{(i)}\right)\right\}_{i=1}^{n}$

Loss:

$$L = \frac{1}{nT}\sum_{i=1}^{n}\sum_{t=1}^{T} L_t(\hat{\bar{y}}_t^{(i)}, \bar{y}_t^{(i)})$$

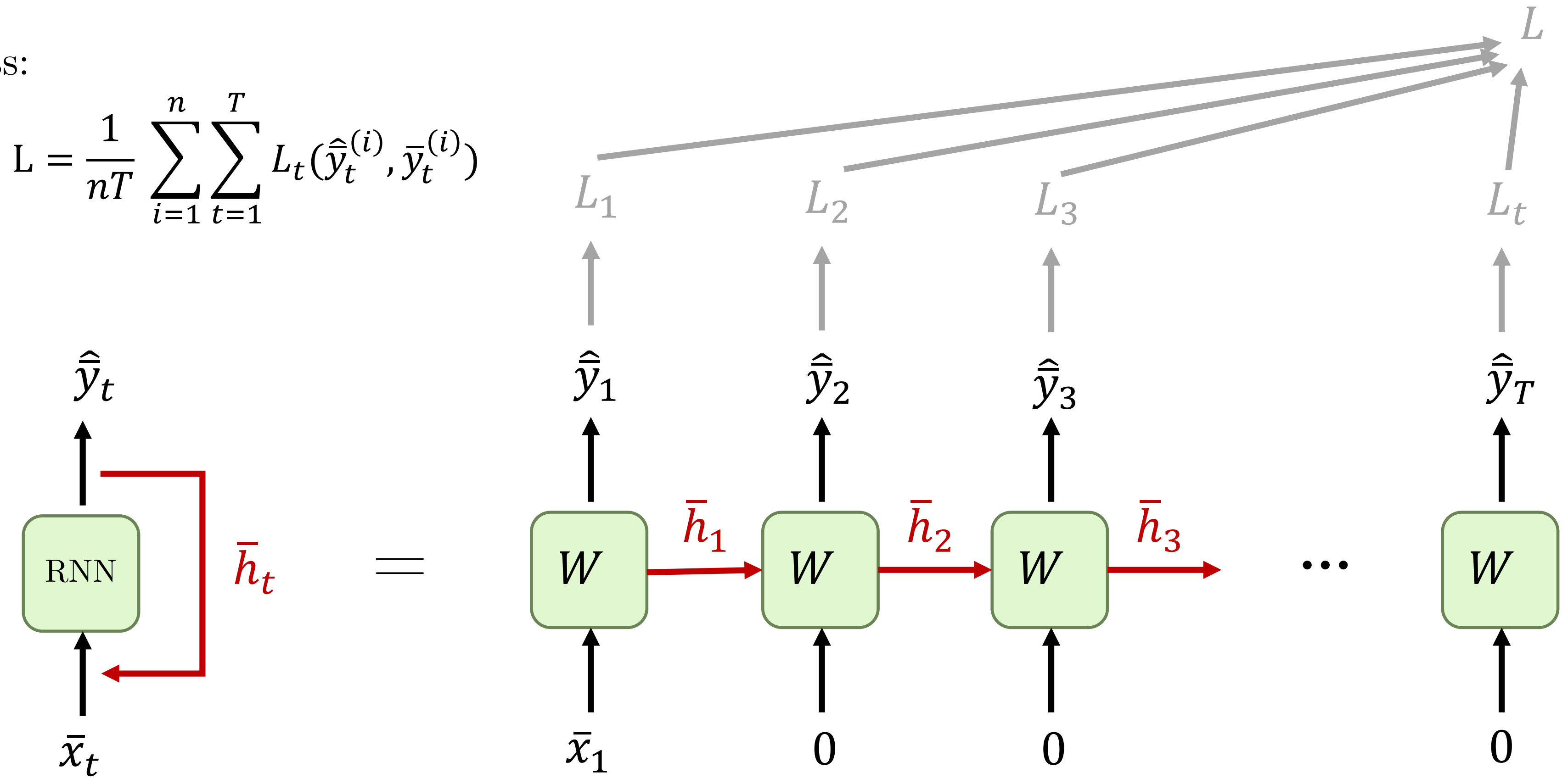


# RNN loss: One to Many

Data: $\left\{\left(\bar{x}^{(i)}, \bar{y}_1^{(i)}, \ldots, \bar{y}_T^{(i)}\right)\right\}_{i=1}^{n}$

Loss:

$$L = \frac{1}{nT}\sum_{i=1}^{n}\sum_{t=1}^{T} L_t(\hat{\bar{y}}_t^{(i)}, \bar{y}_t^{(i)})$$

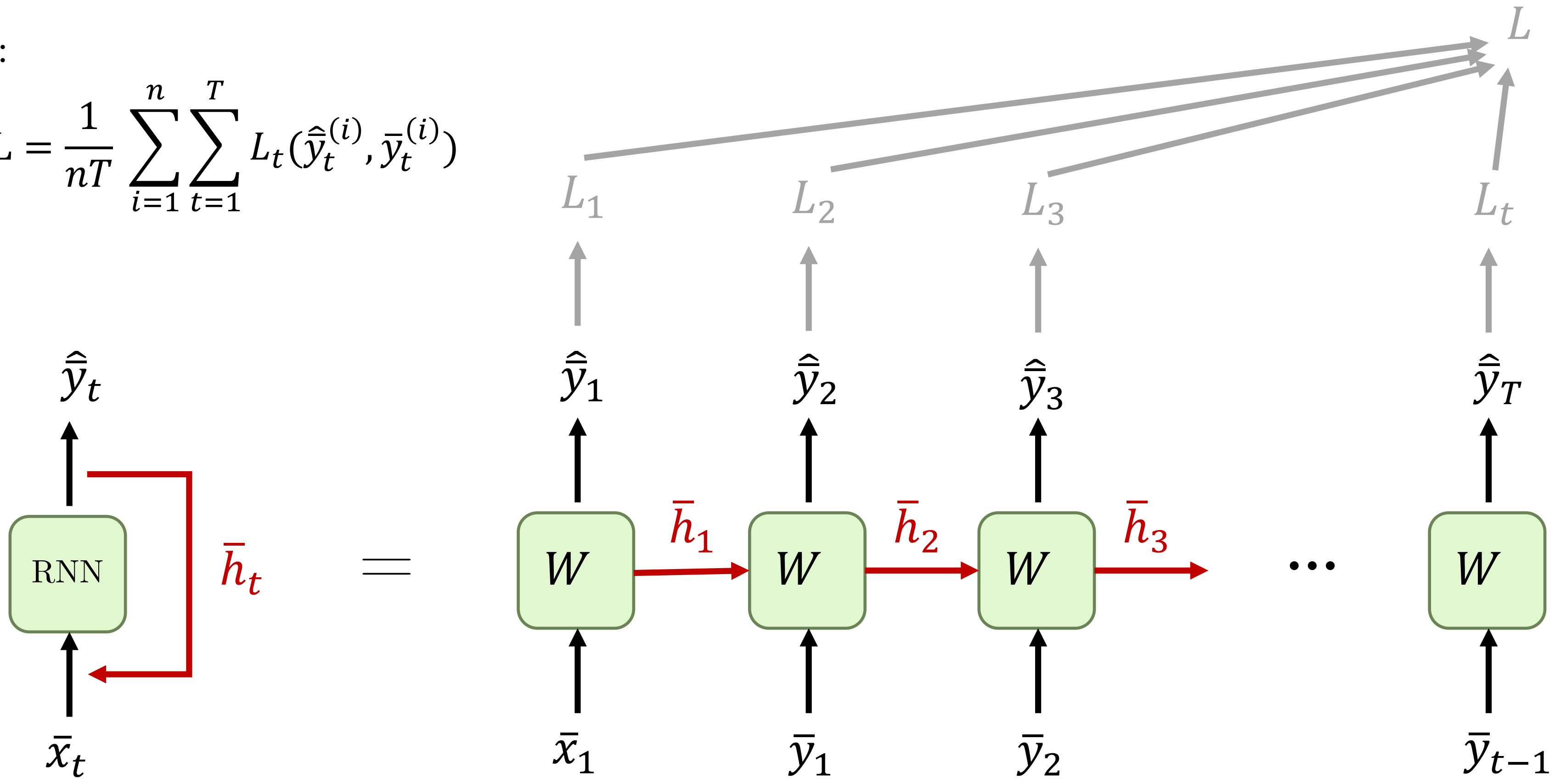


# RNN loss: One to Many

Data: $\left\{\left(\bar{x}^{(i)}, \bar{y}_1^{(i)}, \ldots, \bar{y}_T^{(i)}\right)\right\}_{i=1}^{n}$

Loss:

$$\mathrm{L} = \frac{1}{nT}\sum_{i=1}^{n}\sum_{t=1}^{T} L_t(\hat{\bar{y}}_t^{(i)}, \bar{y}_t^{(i)})$$

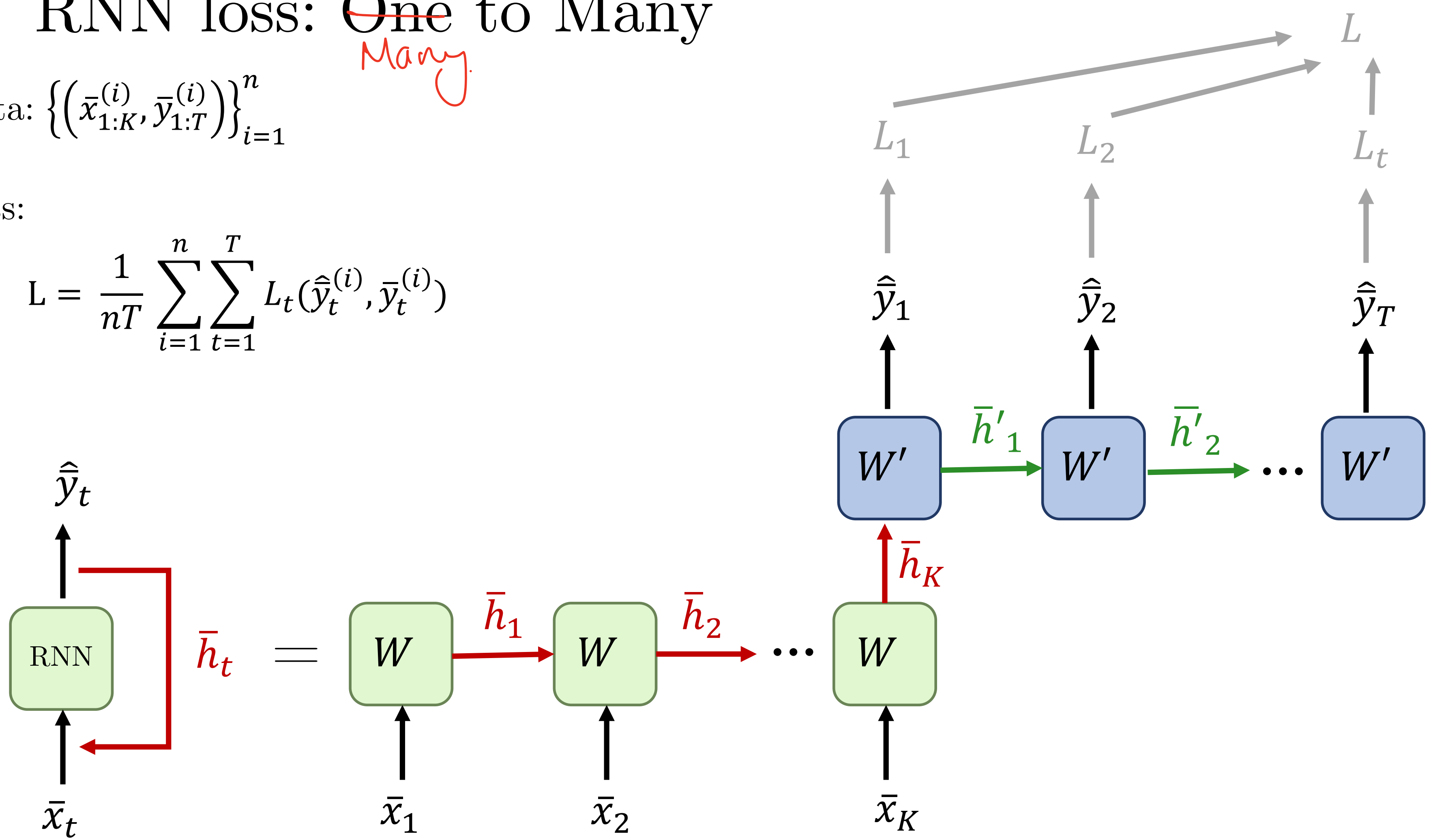


# RNN loss: ~~One~~ Many to Many

Data: $\left\{\left(\bar{x}_{1:K}^{(i)}, \bar{y}_{1:T}^{(i)}\right)\right\}_{i=1}^{n}$

Loss:

$$\mathrm{L} = \frac{1}{nT}\sum_{i=1}^{n}\sum_{t=1}^{T} L_t(\hat{\bar{y}}_t^{(i)}, \bar{y}_t^{(i)})$$



**TL;DPA**

1. RNNs allow for varying input length, sharing of parameters across time and incorporate dependencies over long sequences

2. Core idea: the RNN learns a feature that summarizes the previous state.

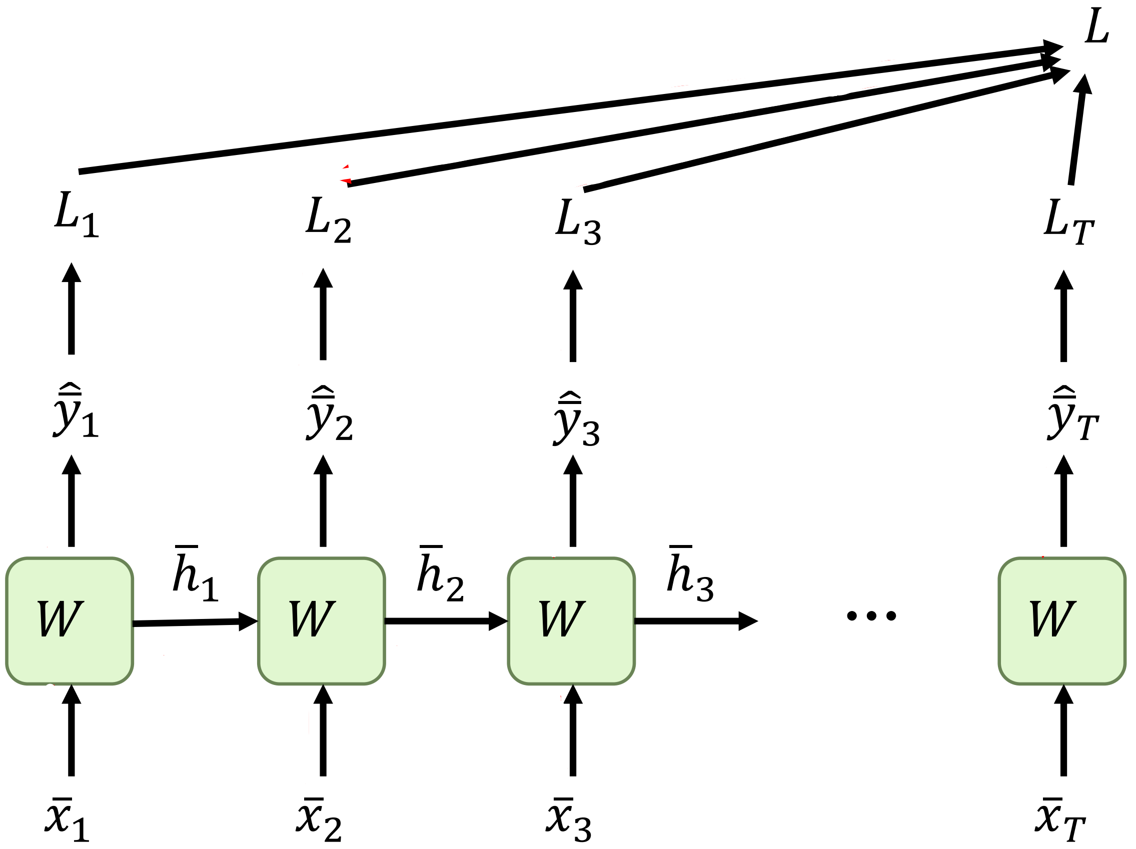
3. How we calculate the loss in an RNN depends on the task (one → many, many → many, many → one, etc)

# An important limitation of vanilla RNNs

- Challenge: the vanilla RNN might “forget” information as the sequence grows longer
- This happens because of vanishing gradients

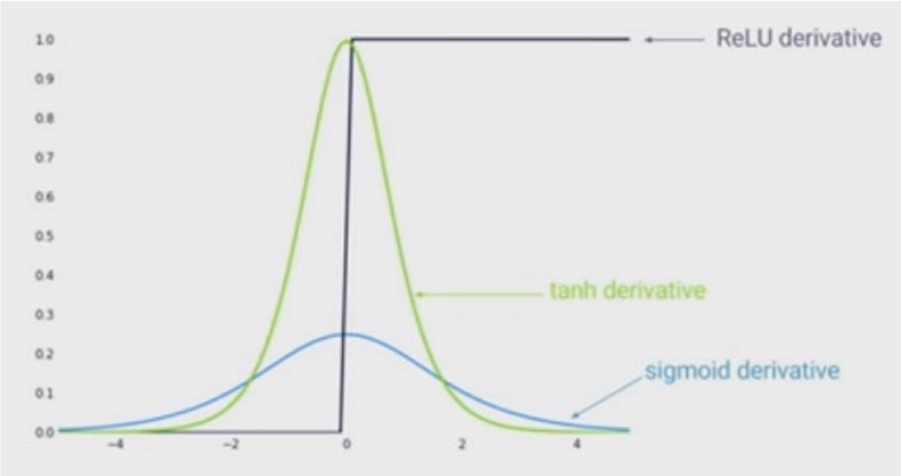
# Vanishing gradients (in 2 minutes)

- Forward pass and loss calculation
-
-



# Solutions to vanishing gradients

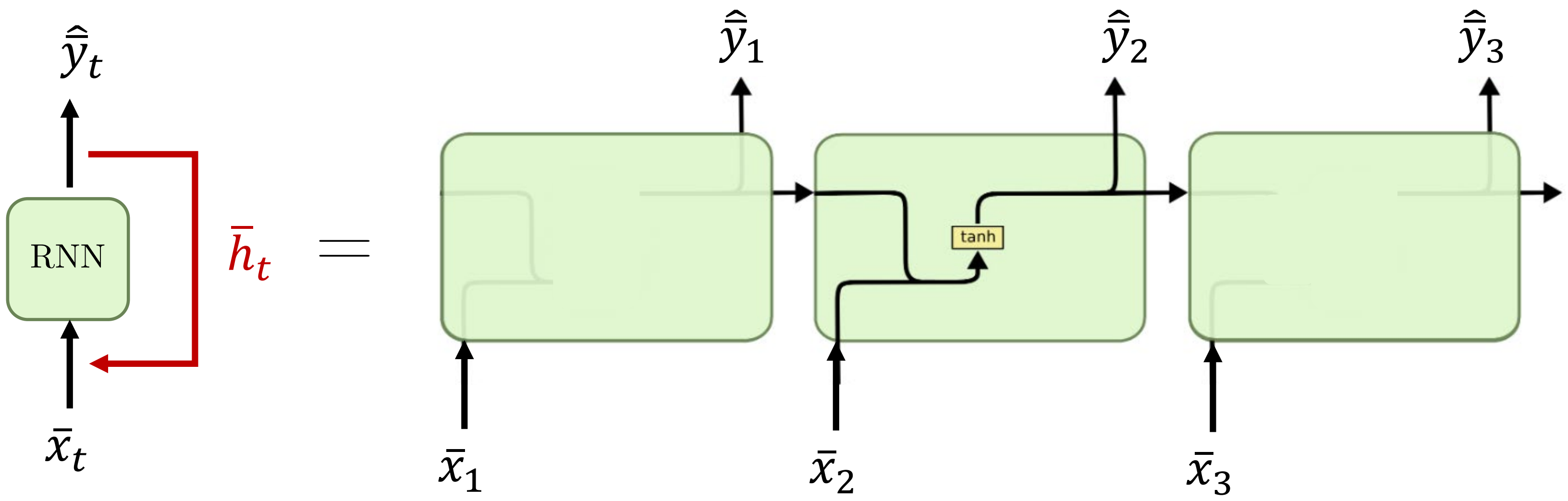
- Better activation functions
- Different network architectures
- Careful weight initializations



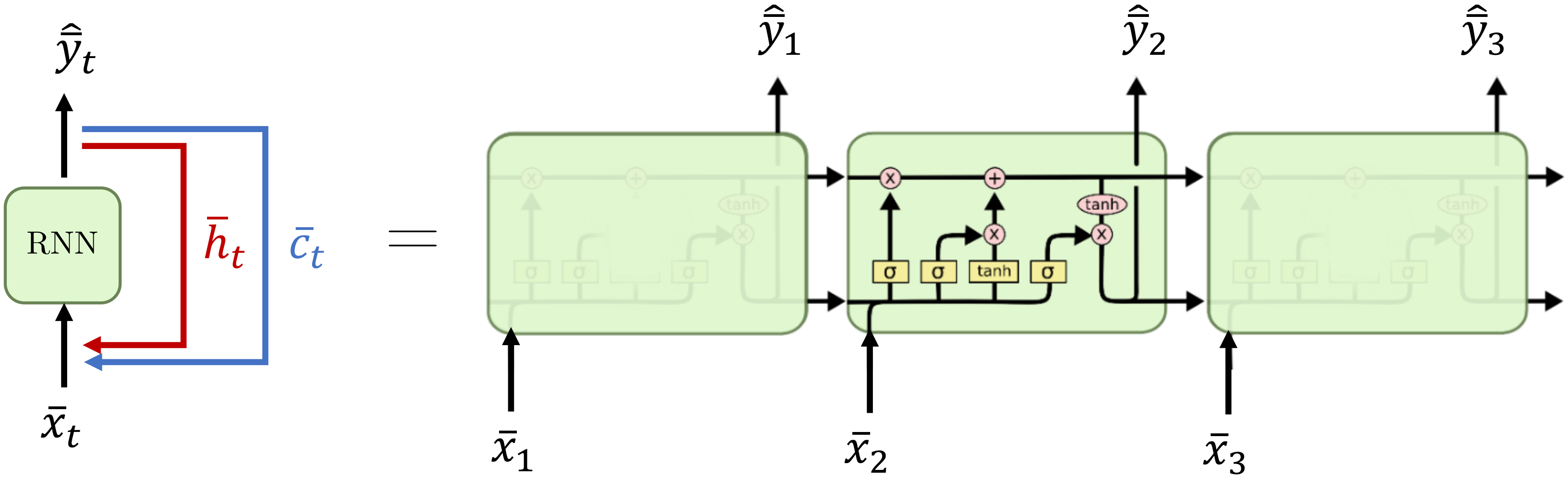
**TL;DPA:**

LSTMs keep track of an additional state. This additional state keeps track of "longer term memory"

# Recall the vanilla RNN unit

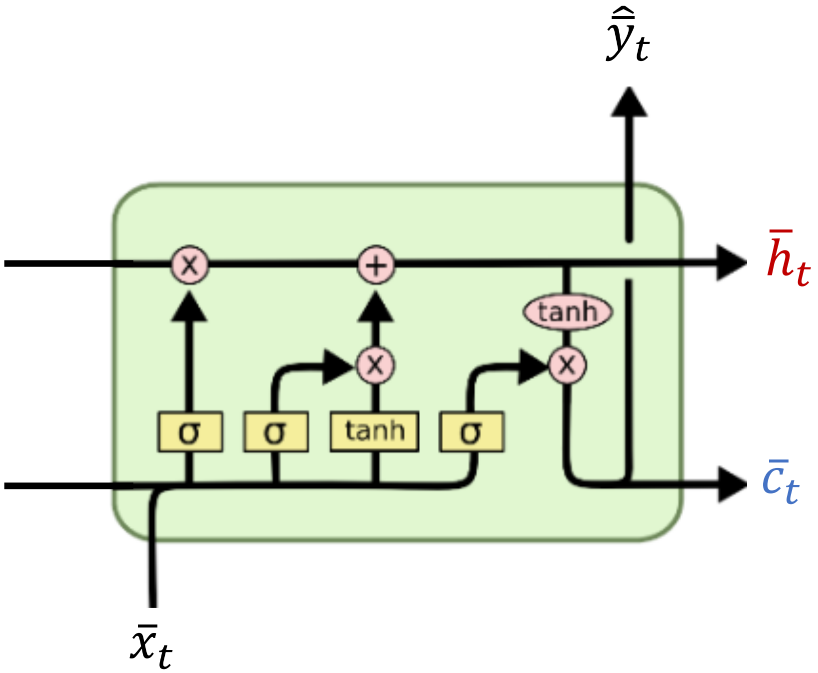


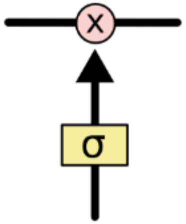
# LSTM unit



$\hat{\bar{y}}_t$
RNN
$\bar{h}_t$
$\bar{c}_t$
$\bar{x}_t$
=
$\hat{\bar{y}}_1$
$\hat{\bar{y}}_2$
$\hat{\bar{y}}_3$
σ
σ
tanh
σ
tanh
$\bar{x}_1$
$\bar{x}_2$
$\bar{x}_3$

# LSTM unit




The "long term state" depends on

1. **Forget gate:** which parts of the long-term memory should I forget?

2. **Input (store) gate:** which parts of the new input should I pay attention to (= encode in the current long-term memory)?

3. **Exposure gate:** which parts of my current long-term memory are relevant in the short-term?

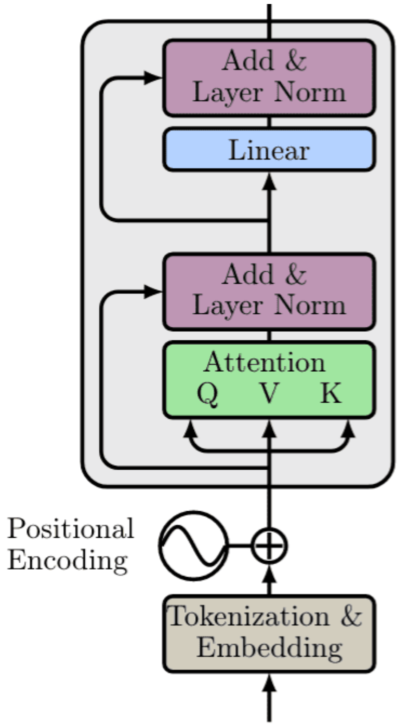
# Limitations of RNNs (Vanilla and LSTM)

- Sequential nature: slow + no parallelization
- Hard to capture long term memory
- Encoding bottleneck

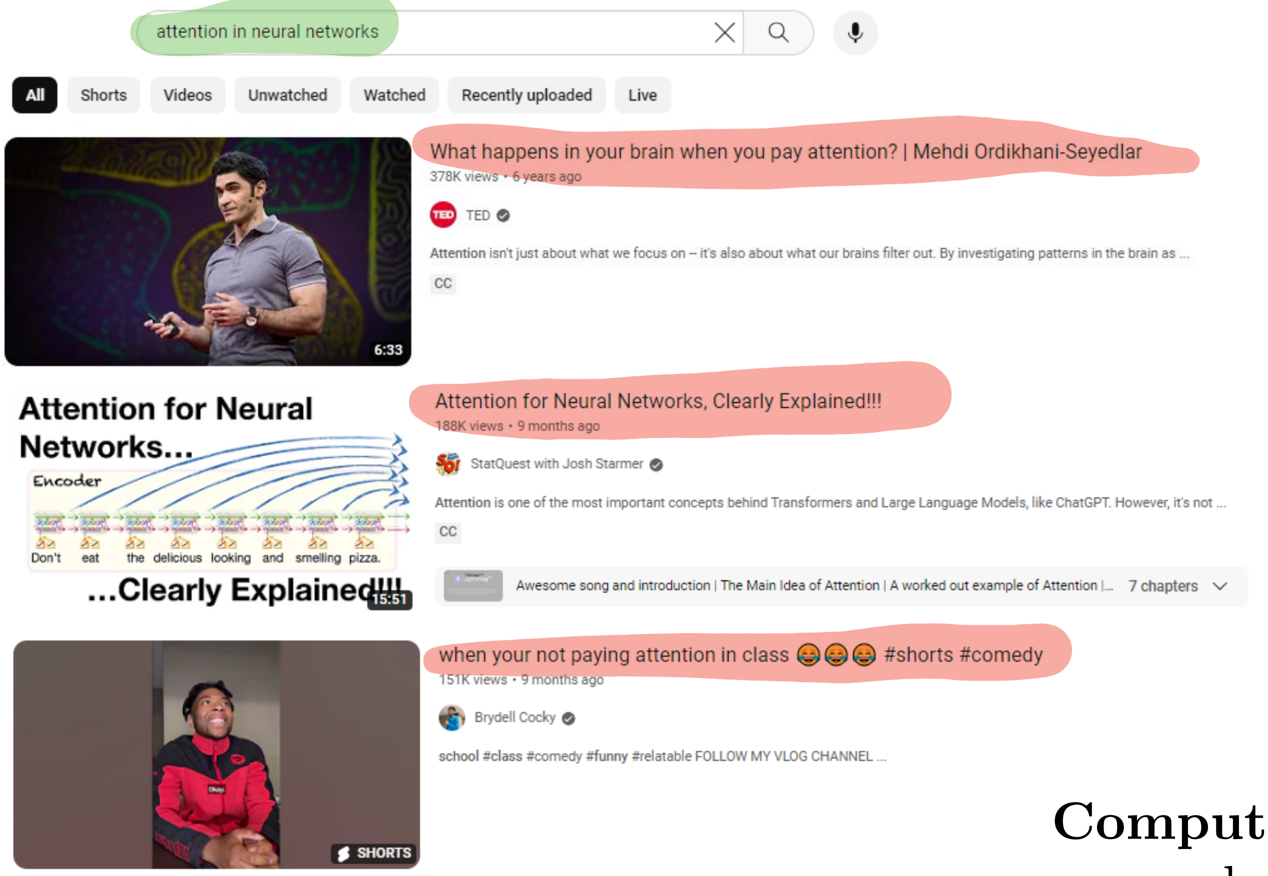
**New paradigm**
Transformers and self-attention

Instead of recurrence, transformers process all the inputs together using self-attention

ChatGPT



# Understanding attention as a search problem




Query $(Q)$

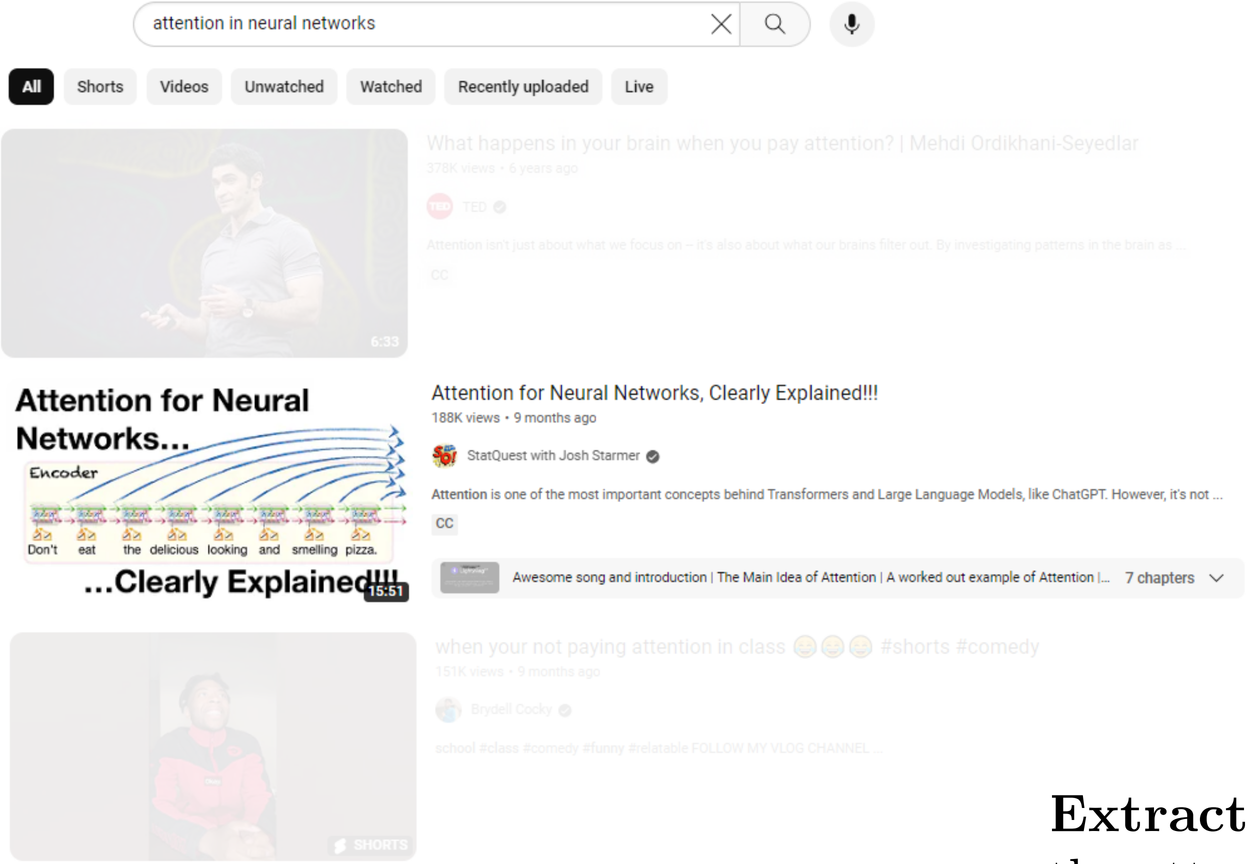
Key $(K_1)$

Key $(K_2)$

Key $(K_3)$

**Compute the attention score:** how similar is each key to the desired query?

# Understanding attention as a search problem




Query (Q)

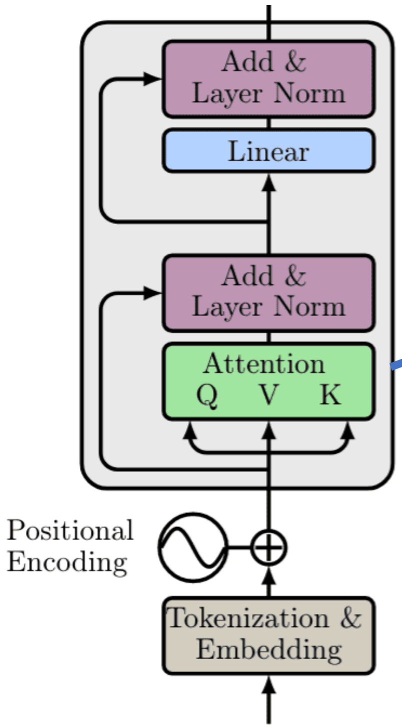
Key $(K_2)$

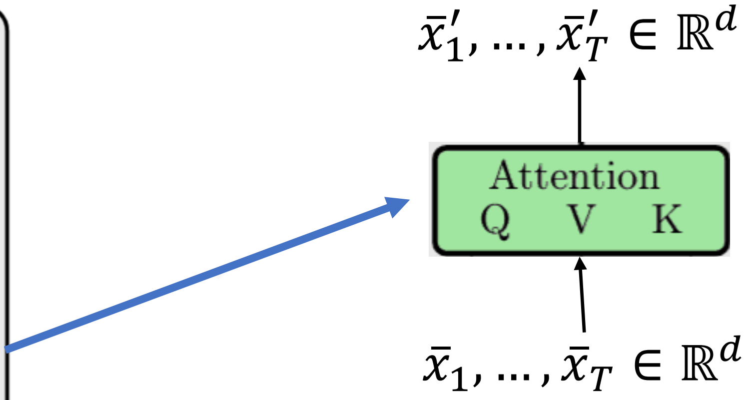
Value $(V_2)$

**Extract the value** based on the attention score

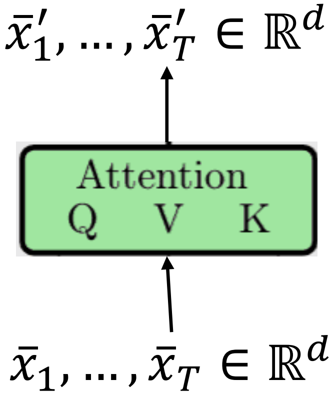
# Transformers and Self-Attention

Core idea: Instead of recurrence, transformers process all the inputs together using self-attention



# Self-Attention

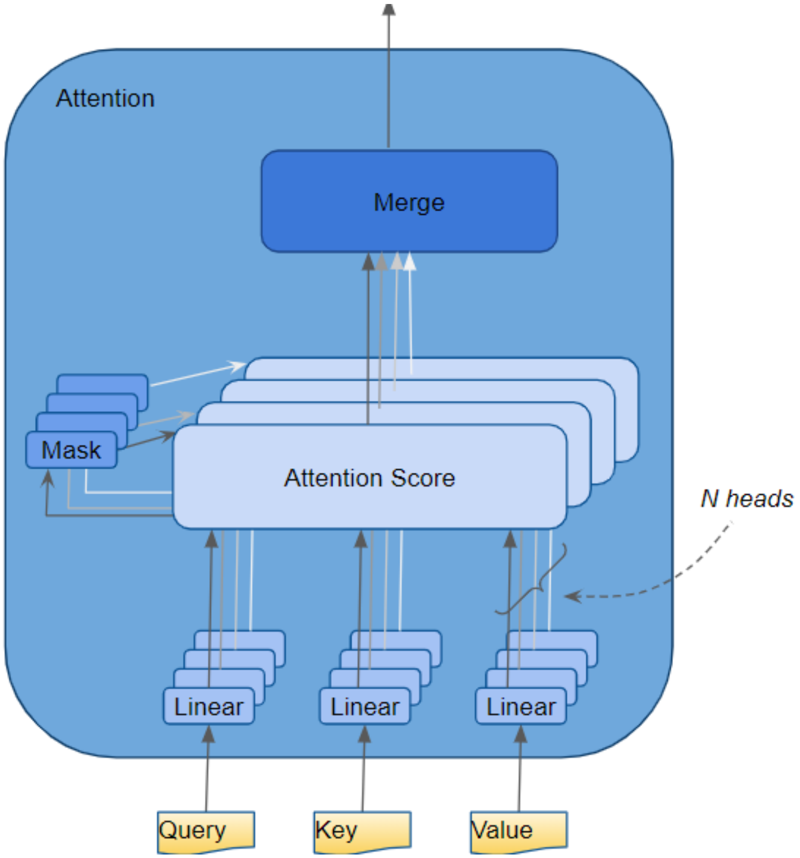



- 3 weight matrices: $W_K, W_Q, W_V$
- For each input token $\bar{x}_i$, compute key, query, and value vectors
  - $\bar{k}_i = W_K \bar{x}_i$
  - $\bar{q}_i = W_Q \bar{x}_i$
  - $\bar{v}_i = W_V \bar{x}_i$
- Compute the similarity scores between $\bar{x}_i$ and all tokens $\bar{x}_j$'s
  - $s_{i,j} = \dfrac{\bar{q}_i \cdot \bar{k}_j}{scaling\ factor}$
- Use softmax to compute the attention scores
  - $a_{i,j} = \exp(s_{i,j}) / Z_i$, where $Z_i = \sum_{j=1}^{T} \exp(s_{i,j})$

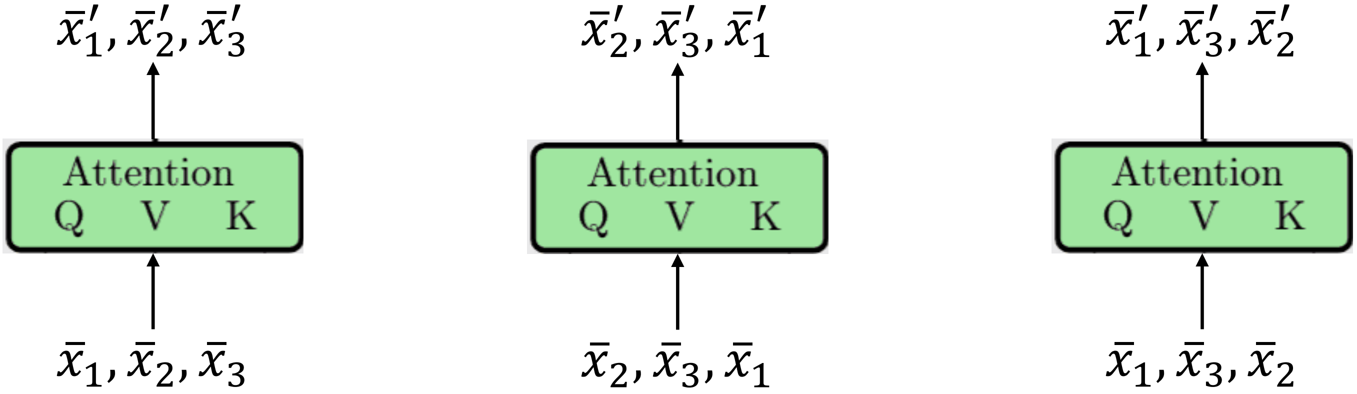
$\sum_j a_{ij} = 1$

- Output the weighted sum of value vectors
  - $\bar{x}_i' = \sum_{j=1}^{T} a_{i,j} \bar{v}_j$

# Multi-Head Self-Attention

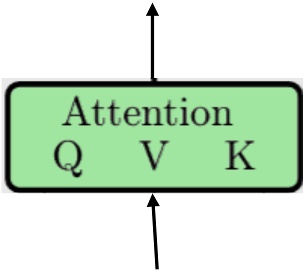


# Order of Inputs



# Positional Encoding

$\bar{x}_1', \ldots, \bar{x}_T' \in \mathbb{R}^d$




$\bar{x}_1, \ldots, \bar{x}_T \in \mathbb{R}^d$

Positional encoding $\bar{p}_1, \ldots, \bar{p}_T \in \mathbb{R}^d$

-

-

# Transformers: Pros and Cons

+ Good at long-term memory: each attention calculation looks at all inputs
+ Parallel computation: processes the entire sequence at once

- High computation cost: quadratic in sequence length $T$
    - Many approaches have been proposed to reduce this cost via approximations

# Transformers in Practice

Large language models

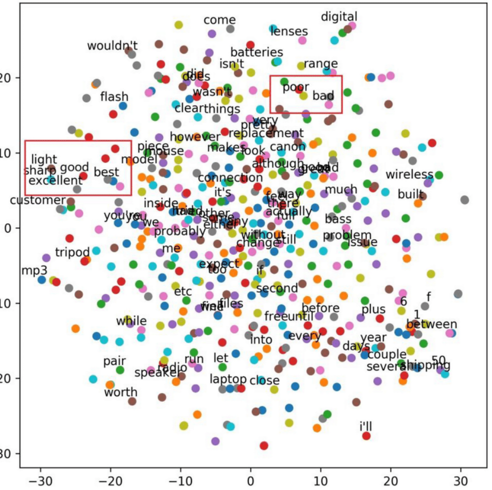
# Large Language Models (LLMs)

**Pre-training task: predicting the next word**

"I went to a café and ordered a ____"          "Latte"

Representing language to a neural network: word embeddings

- Vocabulary size $N$ (e.g. $N \approx 30k$)
- Encode all words as vectors $\bar{e}_1, \ldots, \bar{e}_N \in \mathbb{R}^d$ (learnable)




Treat the next-word prediction problem as multi-class classification and minimize the cross-entropy loss

**TL;DPA:**

1. Transformers (backbone of LLMs) deal with large sequences without relying on recurrence

2. Core component in transformers: self-attention, which tries to identify which parts of the text are relevant with respect to each token in the input

3. LLMs also rely on pre-training *without* labels by predicting the next word

# Thank you!

- Professor Kutty will resume lectures next class
- Check out CSE 598 – 006 next semester: Causality and ML
  https://bit.ly/um-causalml-class