

# DATA MINING 2

## Time Series - Classification

---

Riccardo Guidotti

a.a. 2019/2020

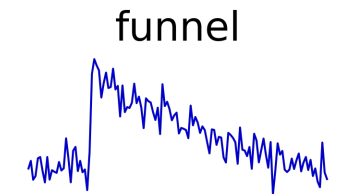
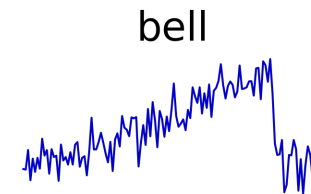
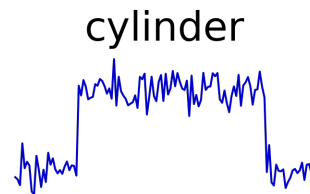


UNIVERSITÀ DI PISA

# Time Series Classification

---

- Main difference between classification and forecasting: forecasting is about predicting a future state/value, classification is about predicting the current label/class.
- Applications:
  - Automated detection of heart diseases
  - Discovery of presence in a room from temperature, humidity, light
  - Identification of the activity performed from smart devices (walking, sitting, laying)
  - Identification of stock market anomalies in pricing, sales volumes, stocks
  - Warning of Natural Disasters (flooding, hurricane, snowstorm),
- Techniques:
  - Motif Discovery
  - Machine Learning Classifiers
  - Deep Neural Networks



# Problem Formulation

---

- Given a set  $X$  of  $n$  time series,  $X = \{x_1, x_2, \dots, x_n\}$ , each time series has  $m$  ordered values  $x_i = \langle x_{t1}, x_{t2}, \dots, x_{tm} \rangle$  and a class value  $c_i$ .
- The objective is to find a function  $f$  that maps from the space of possible time series to the space of possible class values.
- Generally, it is assumed that all the TS have the same length  $m$ .

# Time Series Classification and Similarities

---

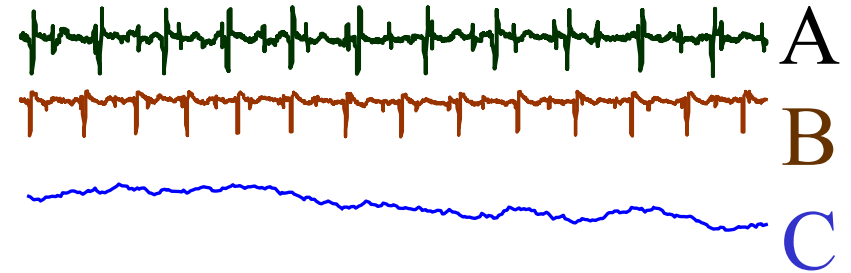
- To some extent, TS classification rely on a measure of similarity between data.
- What makes time series classification an interesting area of investigation is that similarity between series is often embedded within the autocorrelation structure of the data.
- General approaches to measuring similarity between time series:
  - similarity in time (i.e. correlation-based)
  - similarity in change (autocorrelation-based)
  - ***similarity in shape*** (shape-based)
  - ***similarity in structure*** (features-based)
  - ***similarity in representation*** (NN-based)

# Structural-based Classification

---

# Structural-based Classification

- The basic idea is to:
  1. Extract *global* features from the time series,
  2. Create a feature vector, and
  3. Use it to as input for machine learning classifiers
- Example of features:
  - mean, variance, skewness, kurtosis,
  - 1<sup>st</sup> derivative mean, 1<sup>st</sup> derivative variance, ...
  - parameters of regression, forecasting, Markov model



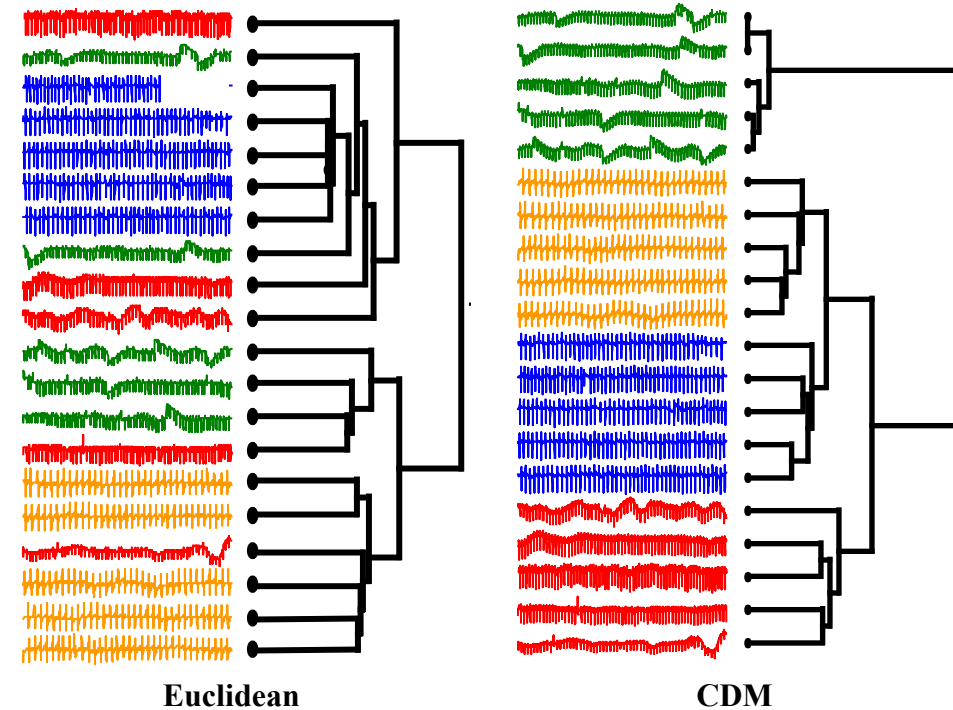
Feature\Time Series	A	B	C
Max Value	11	12	19
Mean	5.3	6.4	4.8
Min Value	3	2	5
Autocorrelation	0.2	0.3	0.5
...	...	...	...

# Shape-based Classification

---

# Shape-based Classification

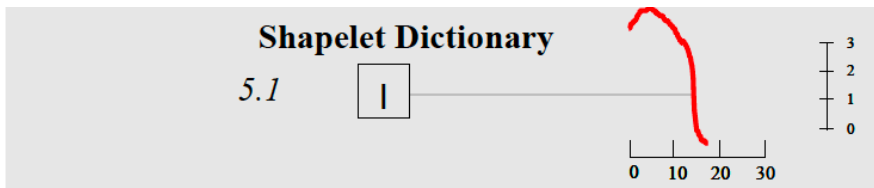
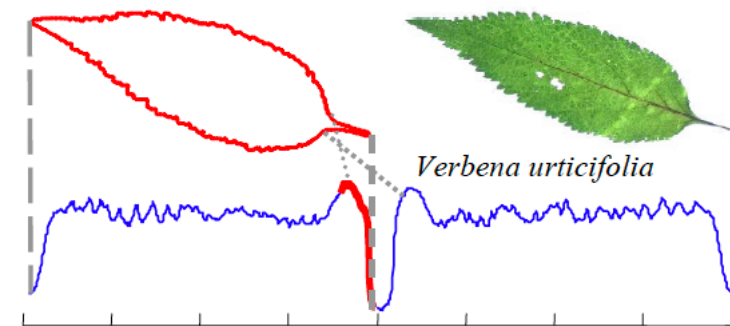
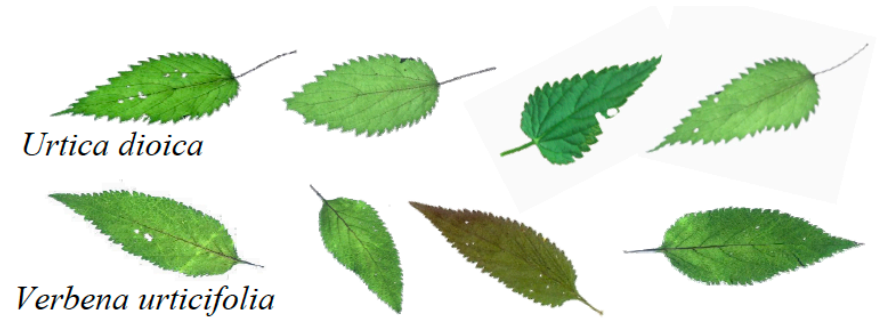
- Calculate the distance between TS using an appropriate distance function:
  - Euclidean/Manhattan
  - Dynamic Time Warping
  - Compression Based Dissimilarity
- Use an instance-based classifier (k-NN) to make the classification.



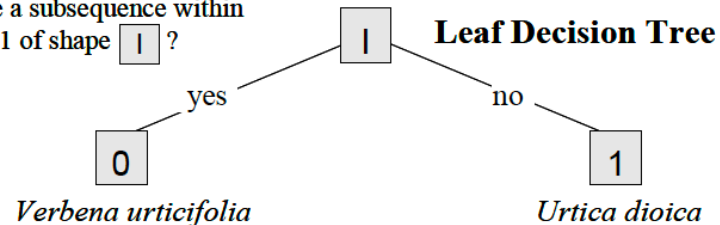


# Shape-based Classification

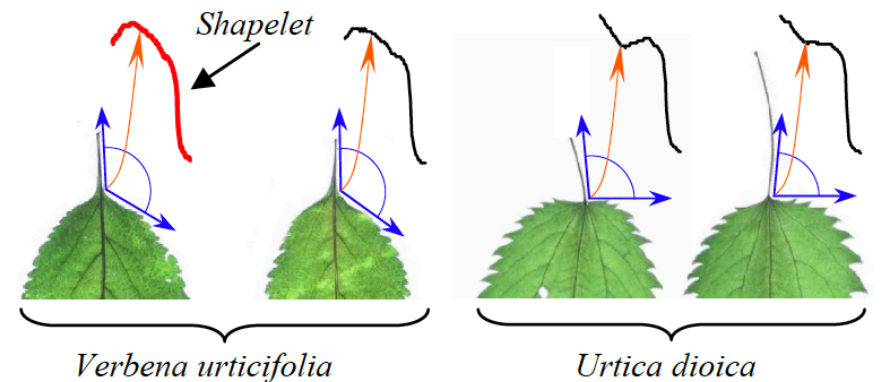
1. Represent a TS as a vector of distances with representative subsequences, namely shapelets.
2. Use it to as input for machine learning classifiers.



Does  $Q$  have a subsequence within a distance 5.1 of shape I?



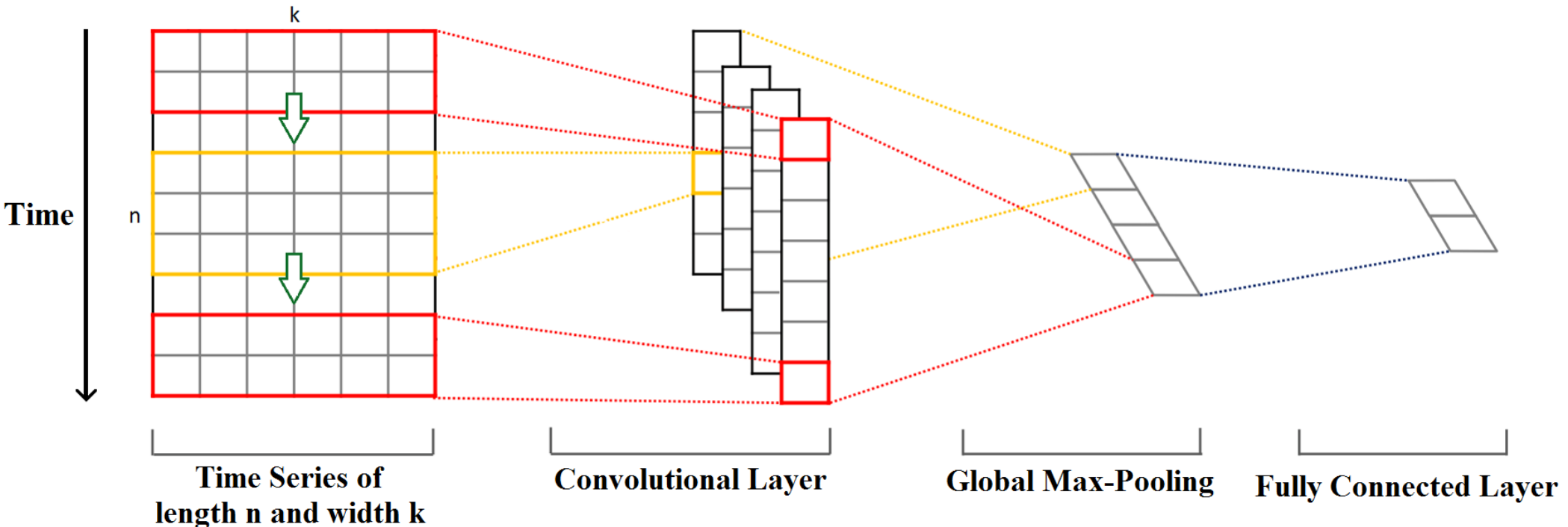
3.2	8.7
1.4	7.9
6.7	4.2
9.2	3.4

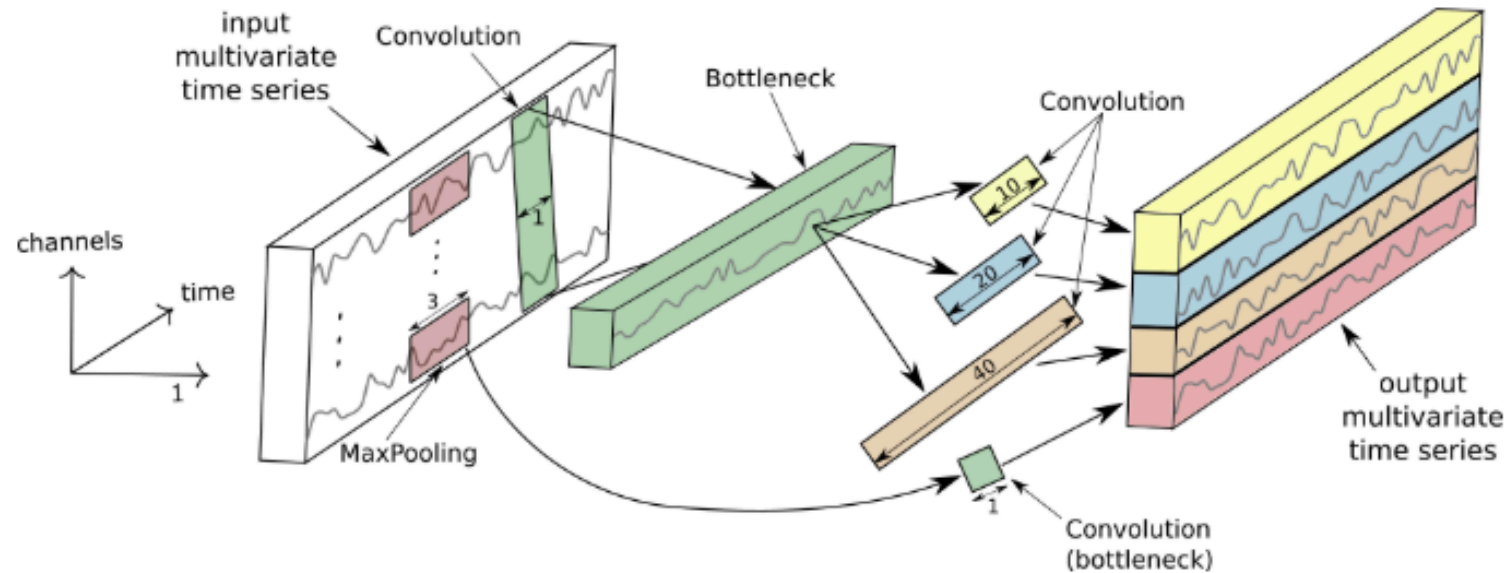


# Time Series Classification with DNN

---

# Time Series Classification with DNN





# Convolutional Neural Network

Slides edited from Stanford

[http://cs231n.stanford.edu/slides/2019/cs231n\\_2019\\_lecture09.pdf](http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture09.pdf)

# Convolutional Neural Network

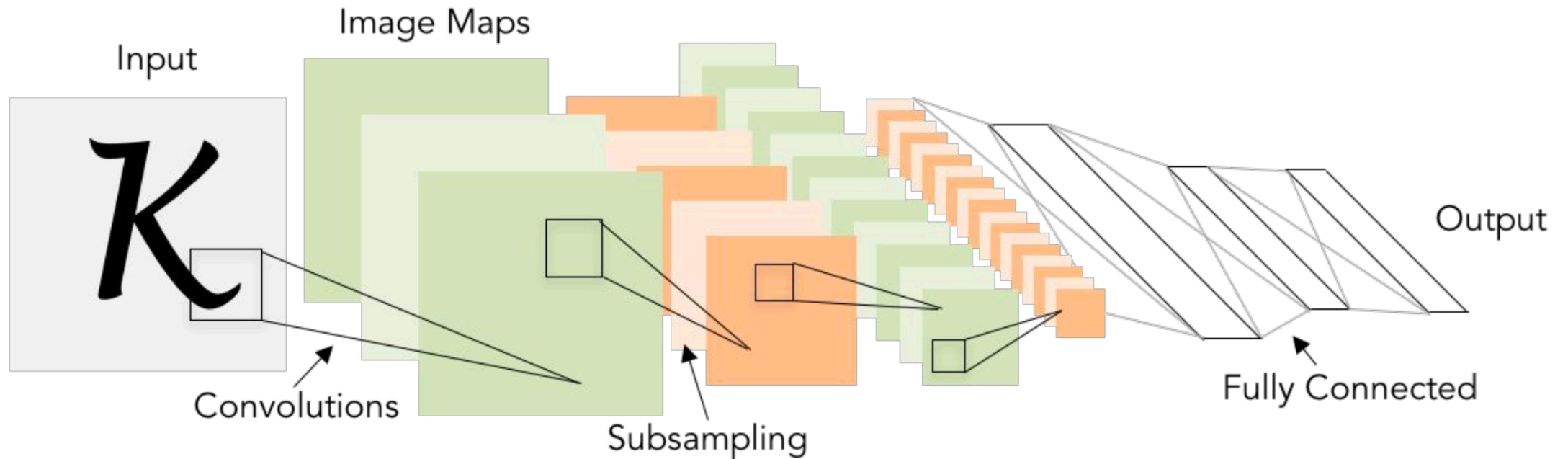
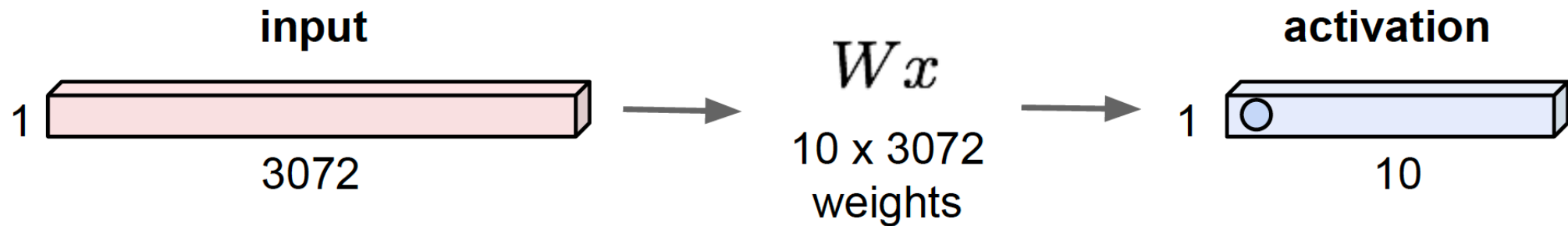


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

# Fully Connected Layer

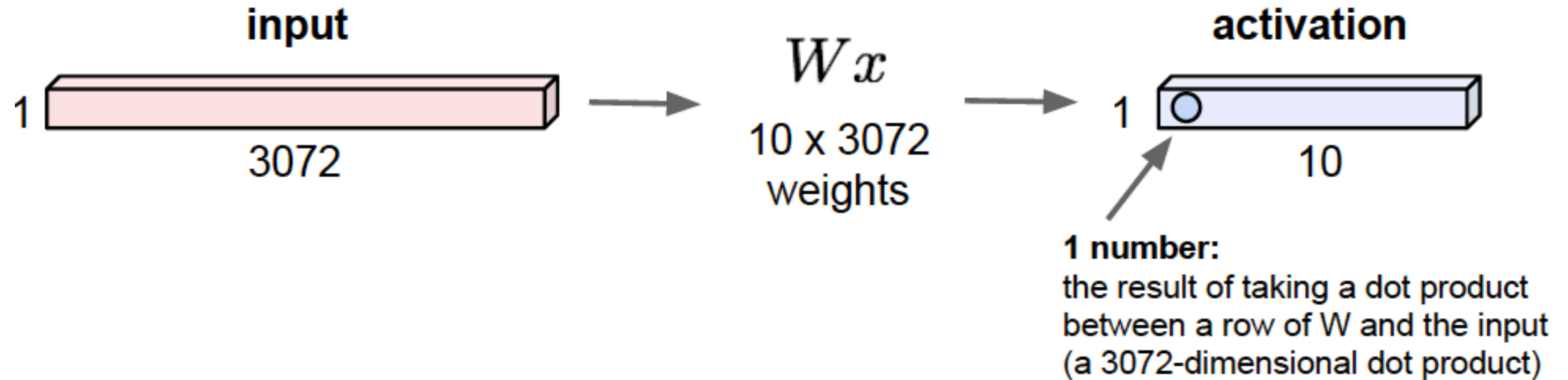
---

32x32x3 image -> stretch to 3072 x 1



# Fully Connected Layer

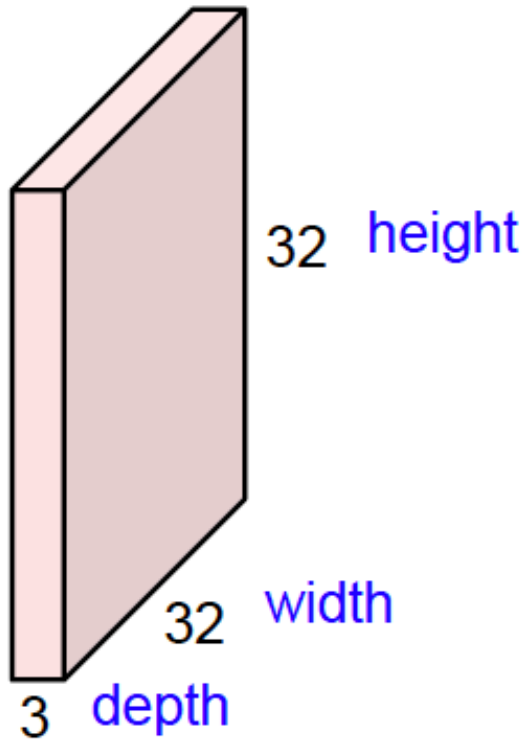
32x32x3 image -> stretch to 3072 x 1



# Convolution Layer

---

32x32x3 image -> preserve spatial structure

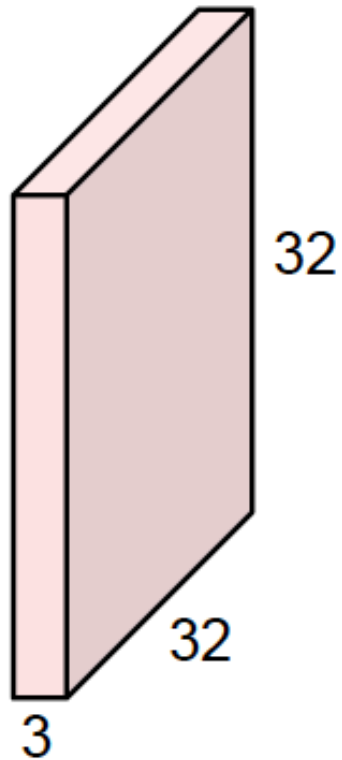




# Convolution Layer

---

32x32x3 image



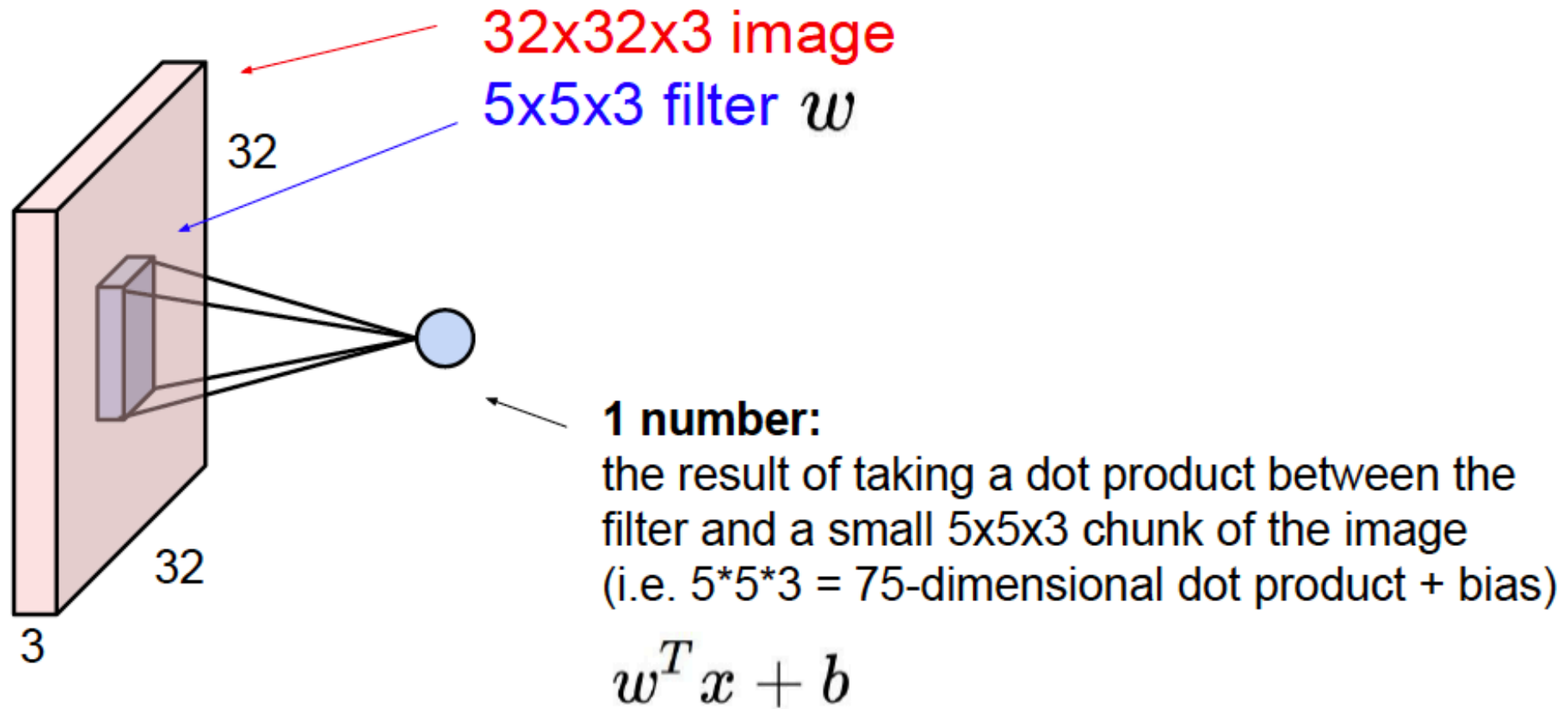
Filters always extend the full depth of the input volume

5x5x3 filter

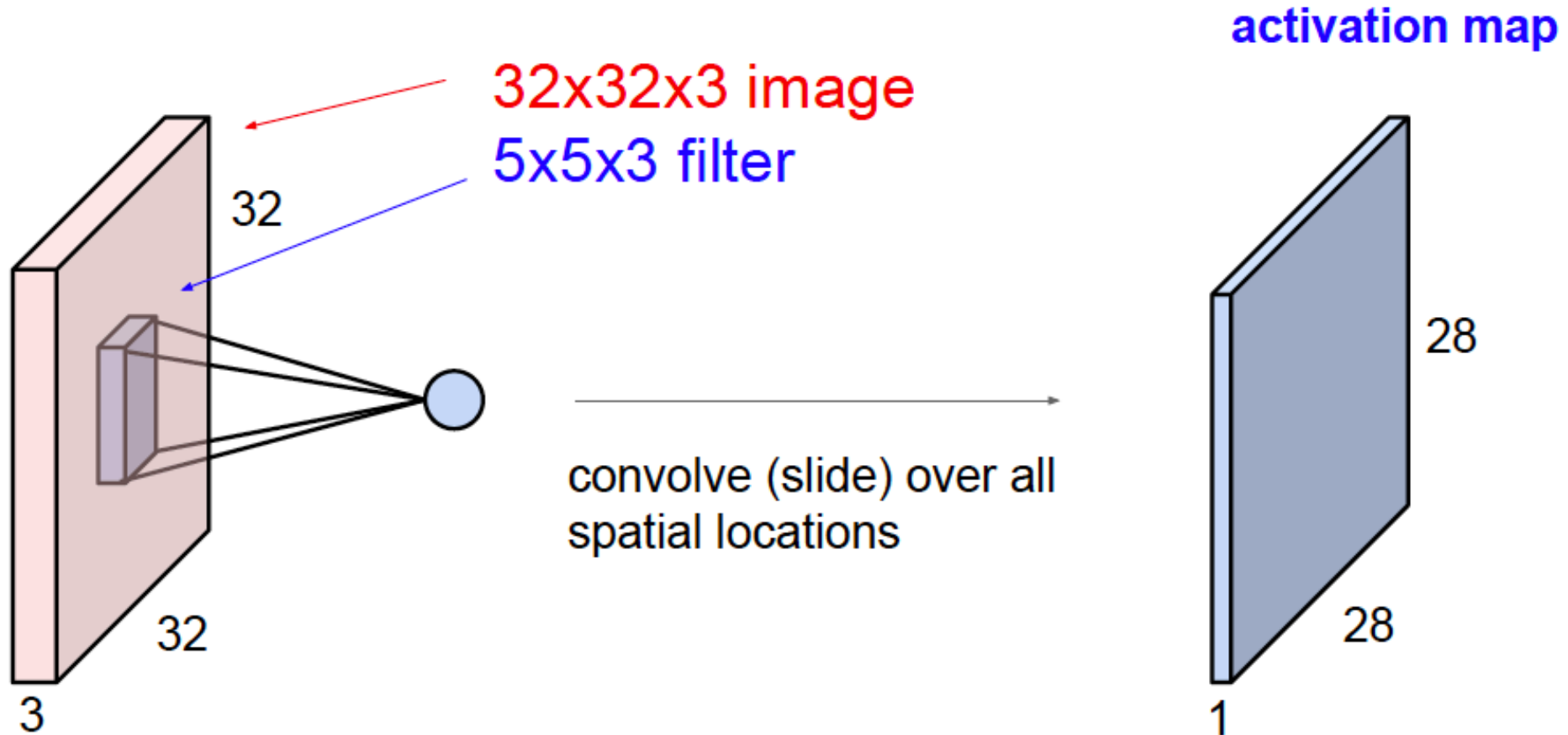


**Convolve** the filter with the image  
i.e. “slide over the image spatially,  
computing dot products”

# Convolution Layer



# Convolution Layer



# Convolution Layer

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

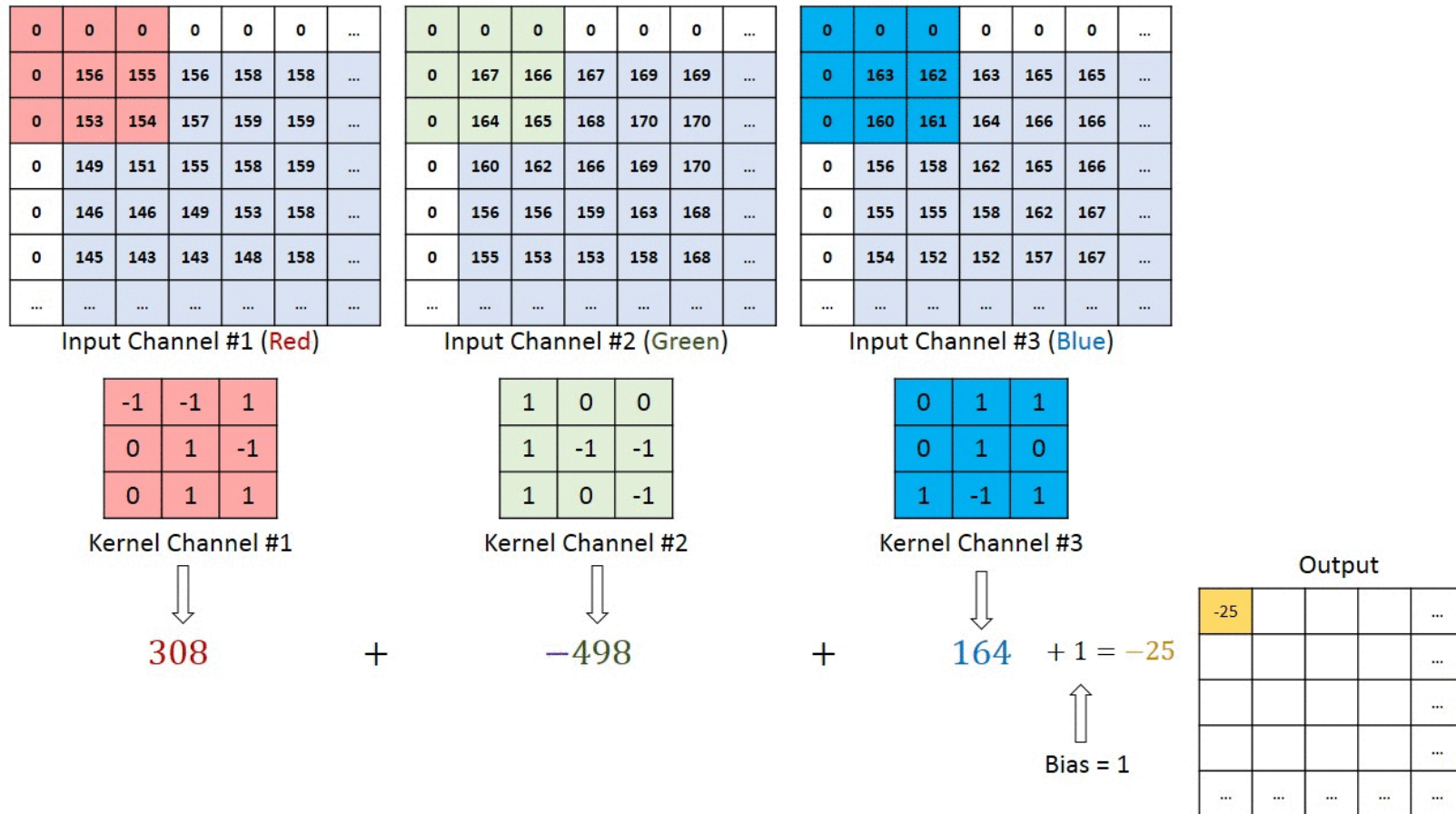
4		

Convolved  
Feature

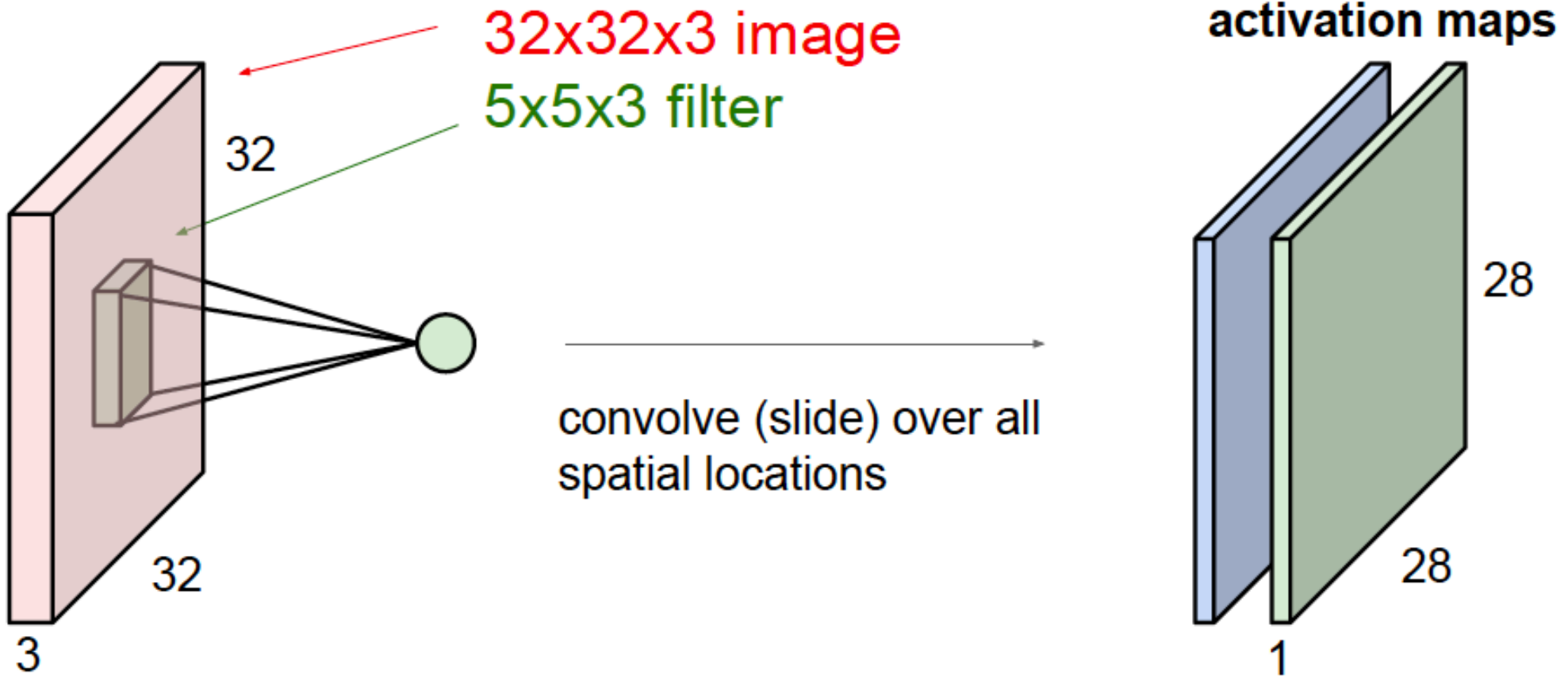
1	0	1
0	1	0
1	0	1

Convolution  
Kernel

# Convolution Layer

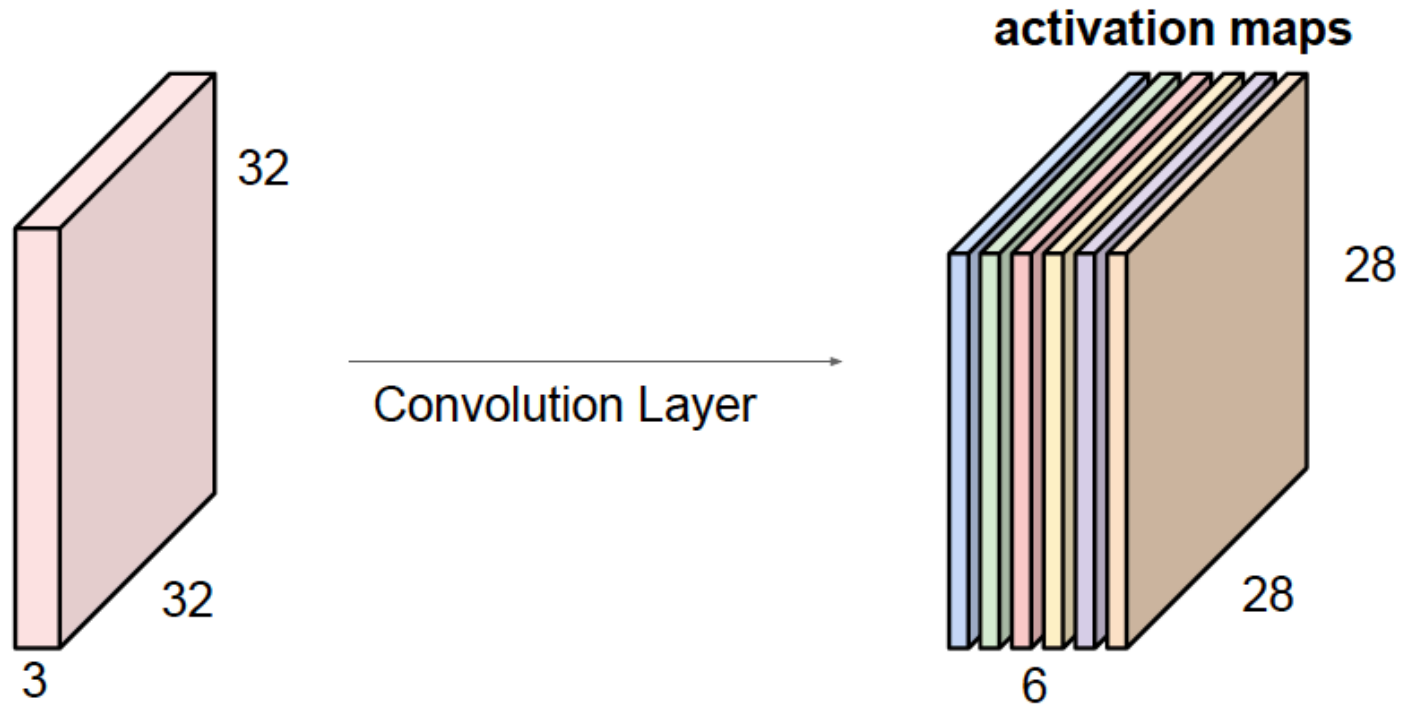


# Convolution Layer



# Convolution Layer

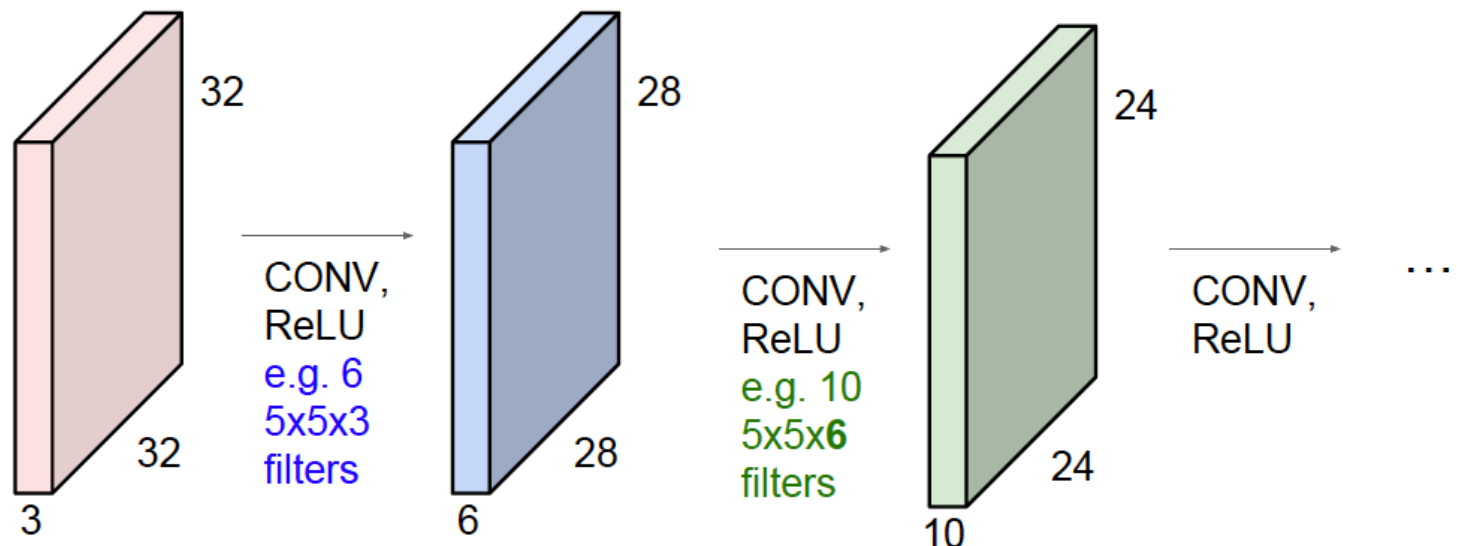
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a “new image” of size 28x28x6!

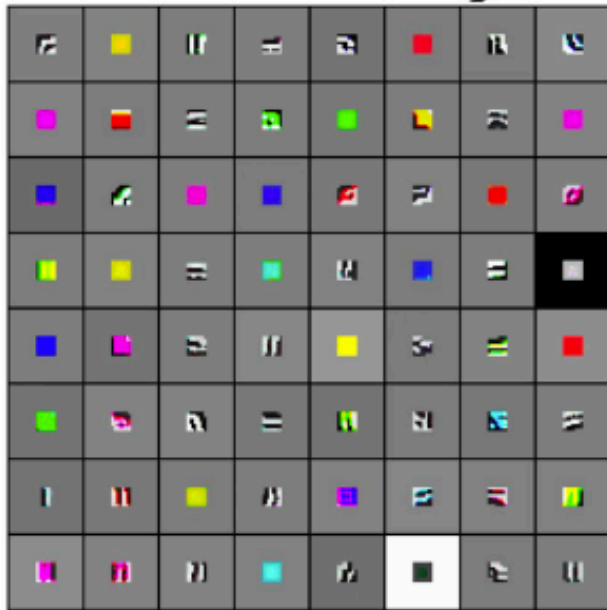
# Convolutional Neural Network

- CNN is a sequence of Conv Layers, interspersed with activation functions.
- CNN shrinks volumes spatially.
- E.g. 32x32 input convolved repeatedly with 5x5 filters! (32 -> 28 -> 24 ...).
- Shrinking too fast is not good, doesn't work well.

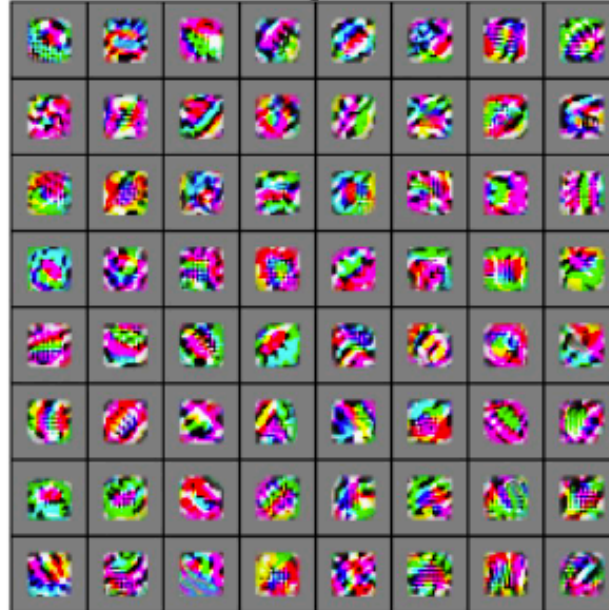




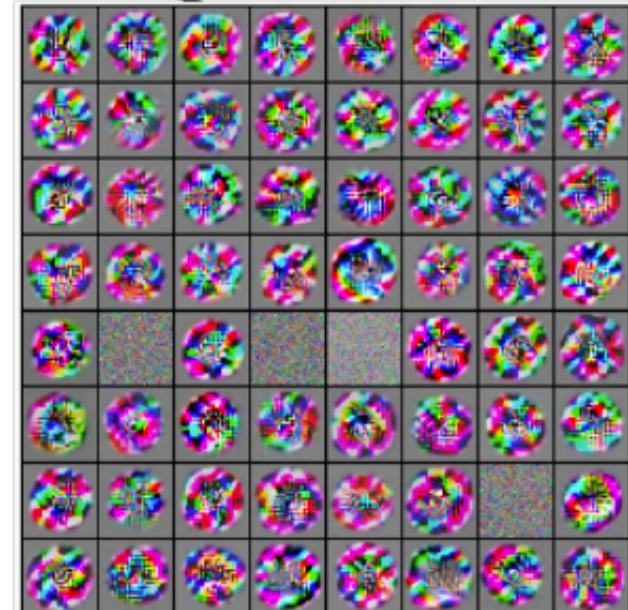
# CNN for Image Classification



VGG-16 Conv1\_1



VGG-16 Conv3\_2

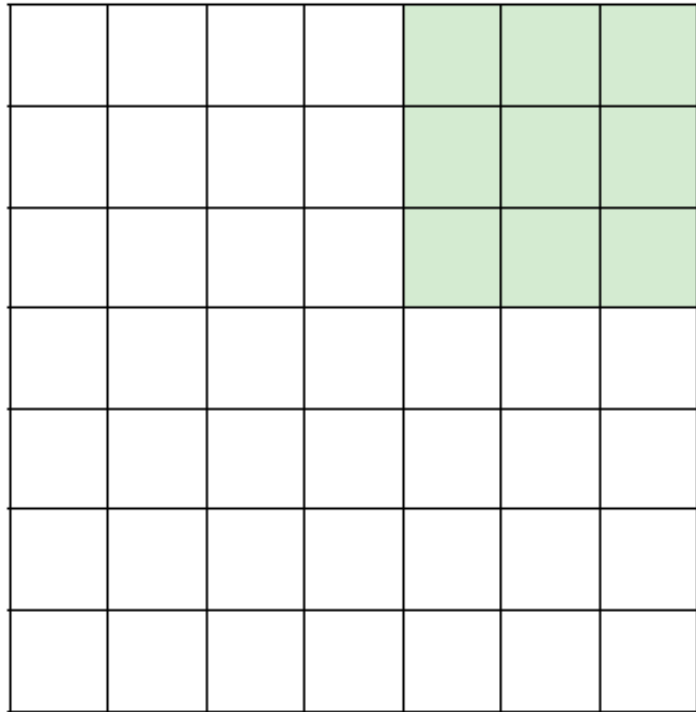


VGG-16 Conv5\_3

# Stride

---

7



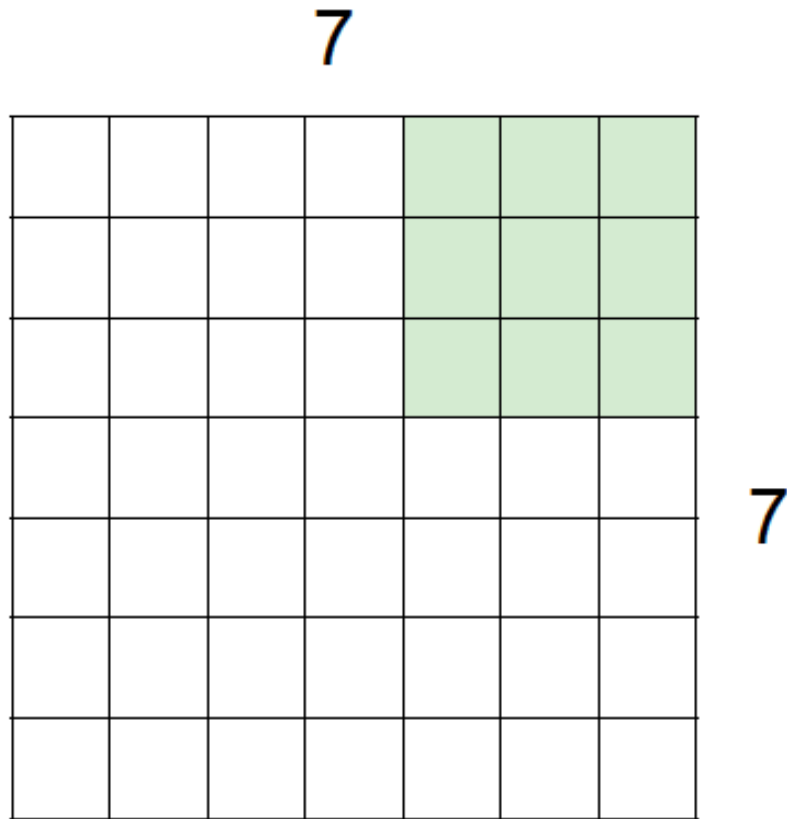
7x7 input (spatially)  
assume 3x3 filter

**=> 5x5 output**

7

# Stride

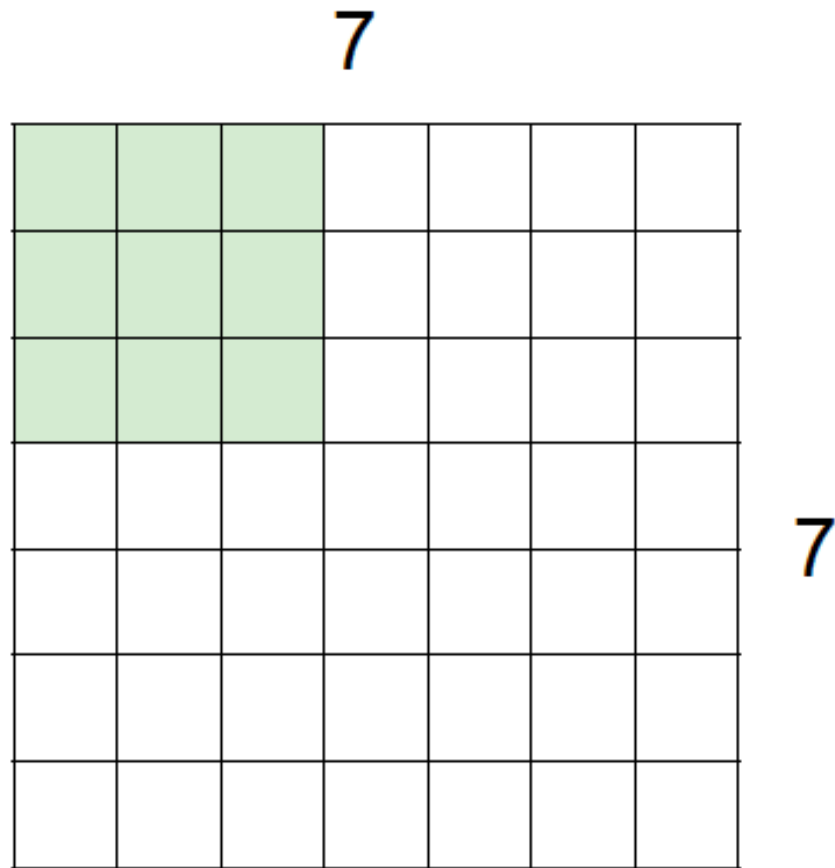
---



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**  
**=> 3x3 output!**

# Stride

---

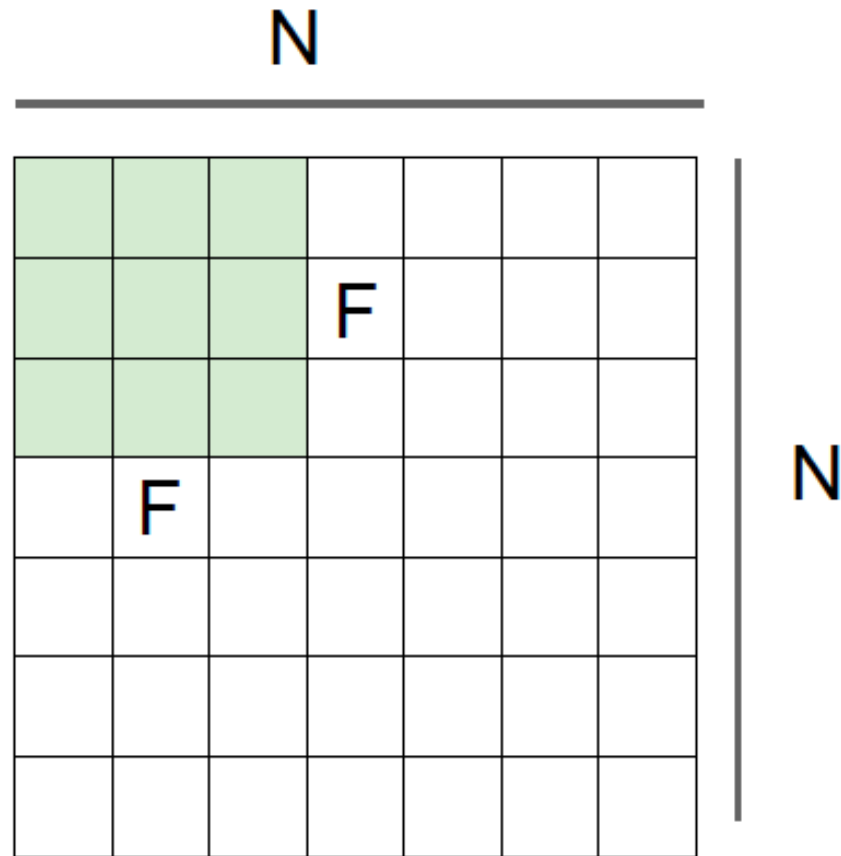


7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

**doesn't fit!**  
cannot apply 3x3 filter on  
7x7 input with stride 3.

# Stride

---



Output size:

$$(N - F) / \text{stride} + 1$$

e.g.  $N = 7, F = 3$ :

$$\text{stride } 1 \Rightarrow (7 - 3) / 1 + 1 = 5$$

$$\text{stride } 2 \Rightarrow (7 - 3) / 2 + 1 = 3$$

$$\text{stride } 3 \Rightarrow (7 - 3) / 3 + 1 = 2.33 \text{ :}\backslash$$

# Padding

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7** output!

In general, common to see CONV layers with stride 1, filters of size  $F \times F$ , and zero-padding with  $(F-1)/2$ .  
(will preserve size spatially)

- $F = 3 \Rightarrow$  zero pad with 1 pixel
- $F = 5 \Rightarrow$  zero pad with 2 pixel
- $F = 7 \Rightarrow$  zero pad with 3 pixel

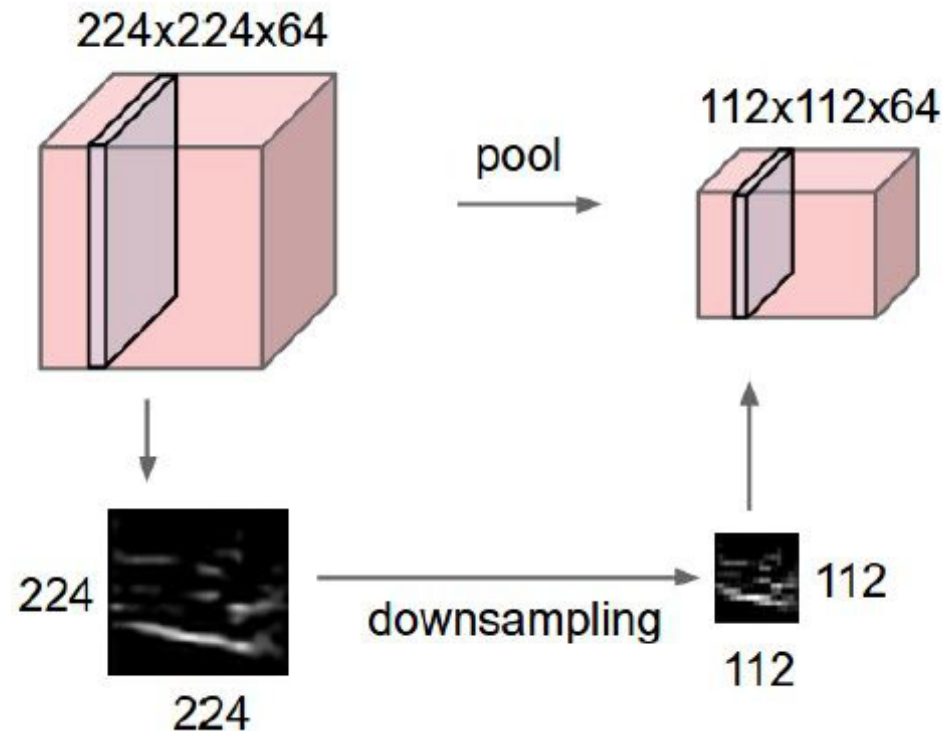
# Summary

---

- Accepts a volume of size  $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters  $K$ ,
  - their spatial extent  $F$ ,
  - the stride  $S$ ,
  - the amount of zero padding  $P$ .
- Produces a volume of size  $W_2 \times H_2 \times D_2$  where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$  (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces  $F \cdot F \cdot D_1$  weights per filter, for a total of  $(F \cdot F \cdot D_1) \cdot K$  weights and  $K$  biases.
- In the output volume, the  $d$ -th depth slice (of size  $W_2 \times H_2$ ) is the result of performing a valid convolution of the  $d$ -th filter over the input volume with a stride of  $S$ , and then offset by  $d$ -th bias.

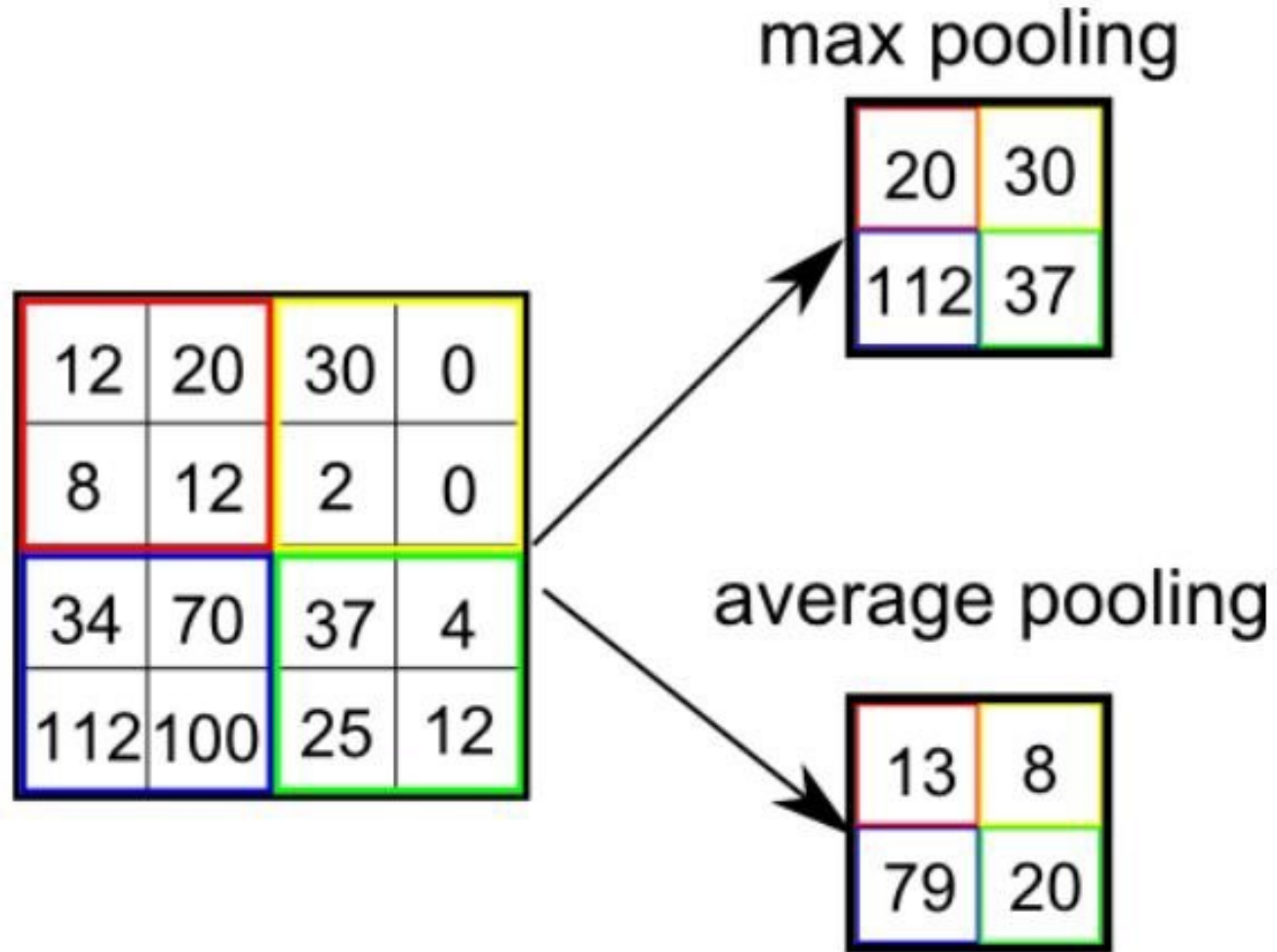
# Pooling Layer

- Makes the representations smaller and more manageable
- Operates over each activation map independently





# MaxPooling and AvgPoling

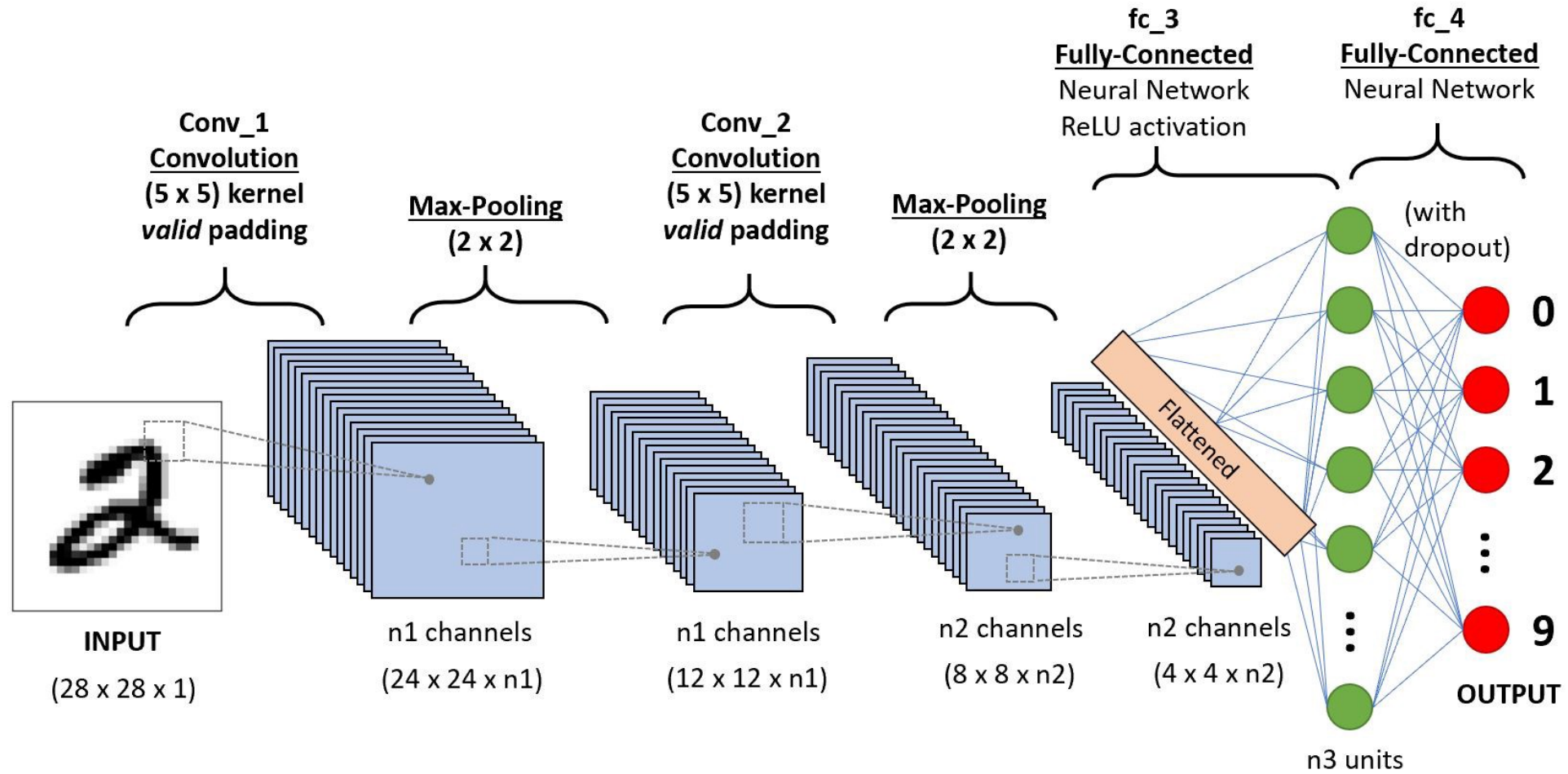


# Pooling

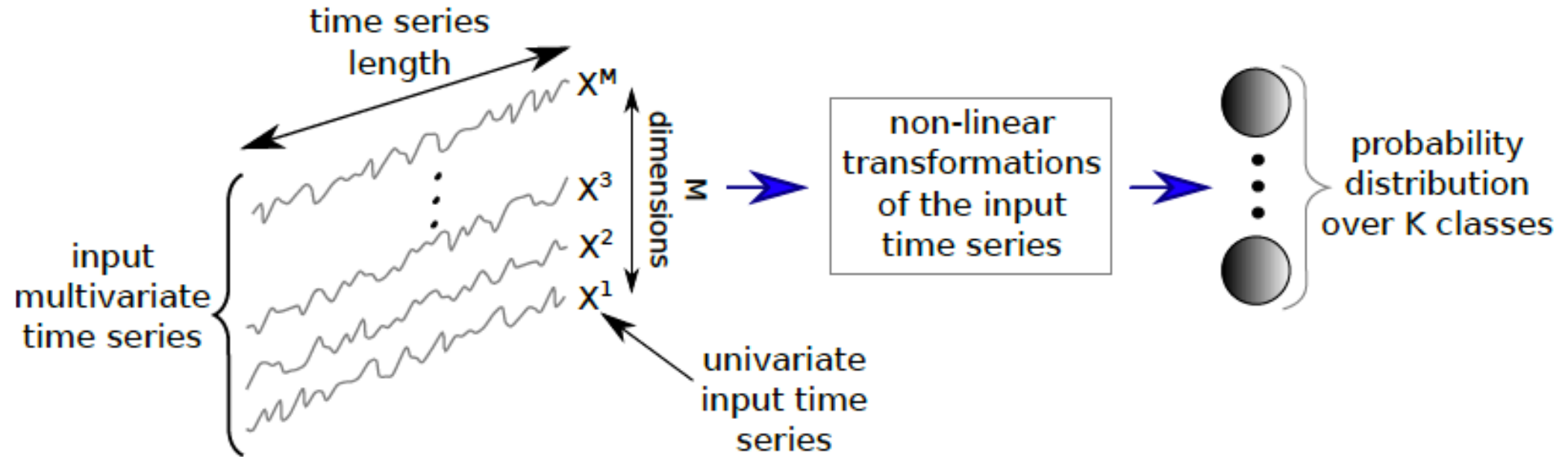
---

- Accepts a volume of size  $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent  $F$ ,
  - the stride  $S$ ,
- Produces a volume of size  $W_2 \times H_2 \times D_2$  where:
  - $W_2 = (W_1 - F) / S + 1$
  - $H_2 = (H_1 - F) / S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

# Example of CNN

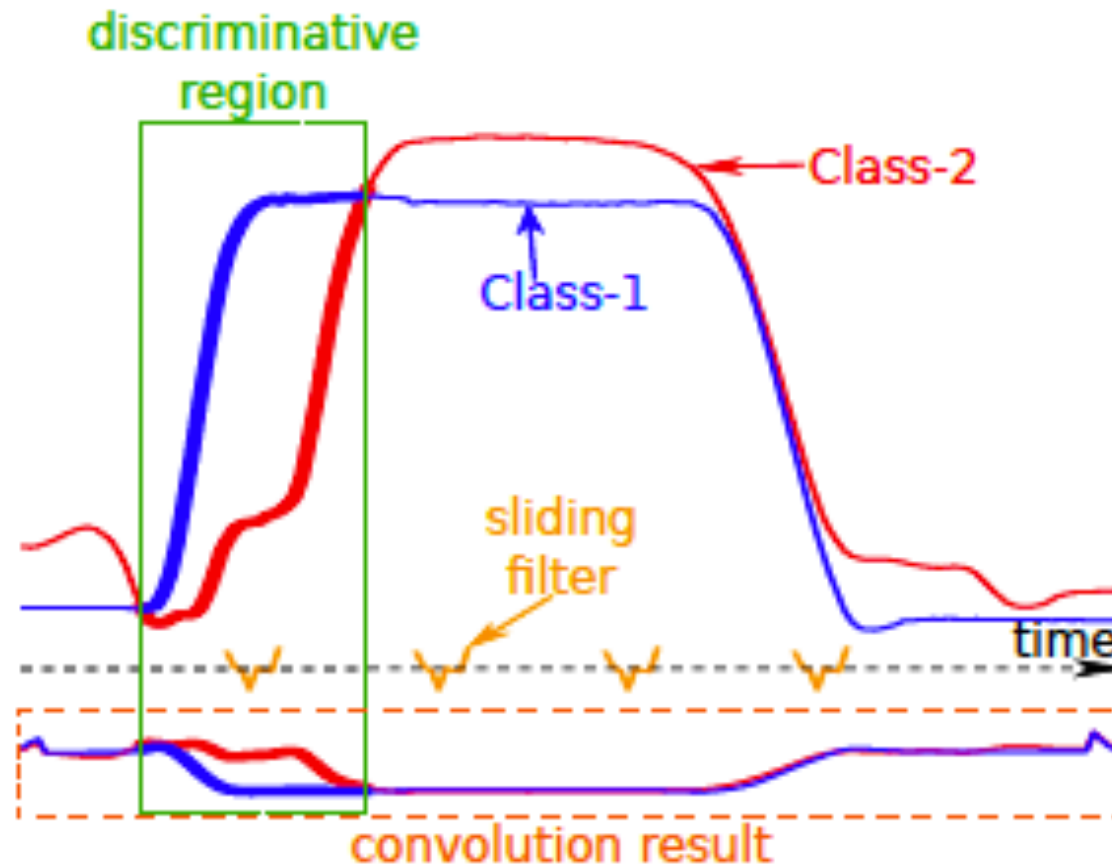


# CNN for Time Series Classification

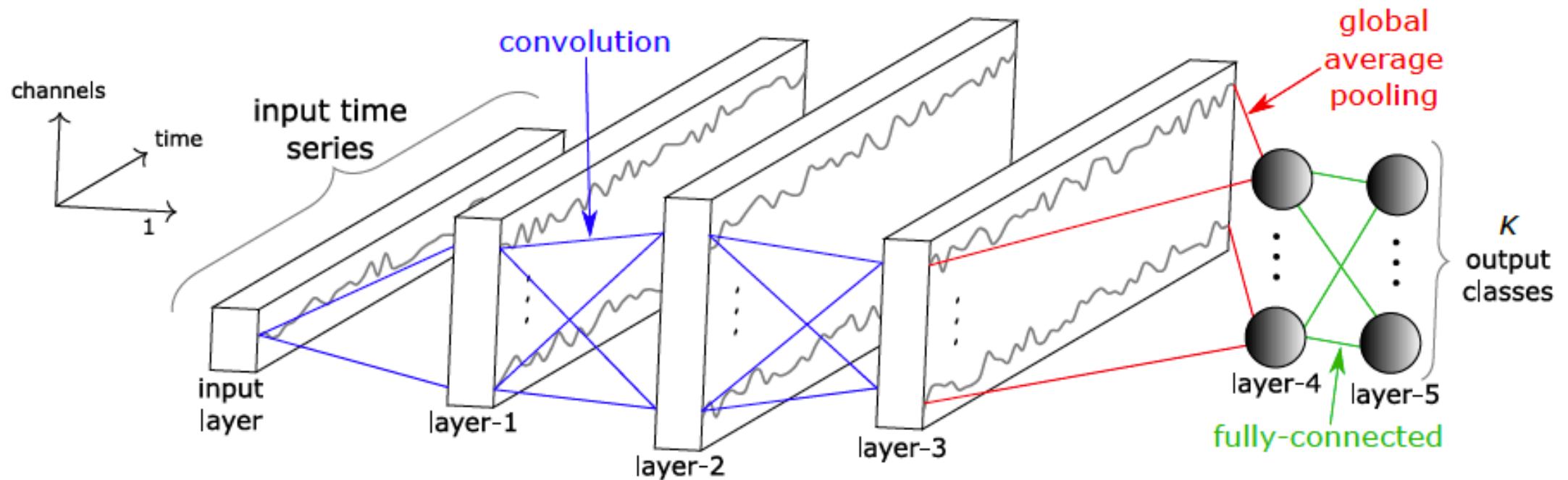


# CNN for Time Series Classification

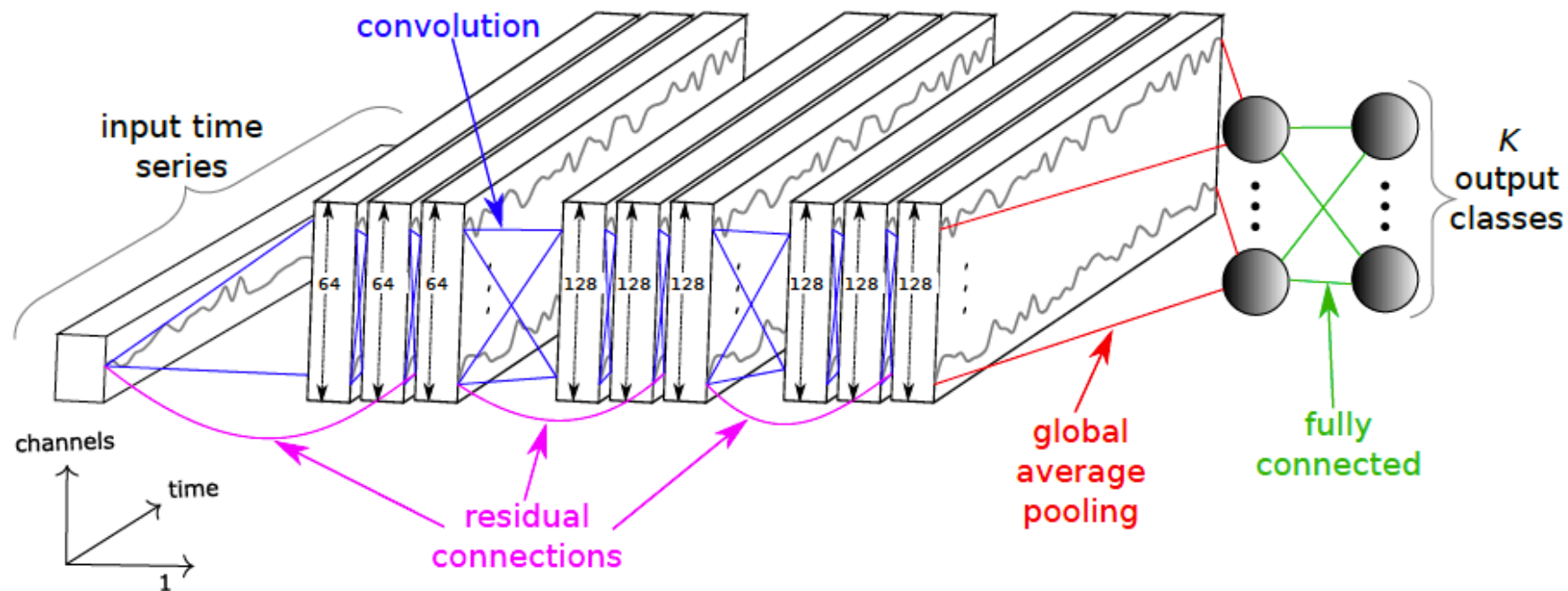
- Result of applying a learned discriminative convolution.



# CNN for Time Series Classification



# Residual Neural Network (ResNN/ResNet)



The main characteristic of ResNets is the shortcut residual connection between consecutive CONV layers. The difference with the usual CNN is that a linear shortcut is added to link the output of a residual block to its input thus enabling the flow of the gradient directly through these connections, which makes training a DNN much easier by reducing the vanishing gradient effect.

# CNN Summary

---

- ConvNets stack Convolutional, Pooling, Fully Connected Layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically CNN looked like
  - $[(\text{CONV-RELU})^*N\text{-POOL?}]^*M\text{-(FC-RELU)}^*K, \text{SOFTMAX}$
  - where N is usually up to  $\sim 5$ , M is large,  $0 \leq K \leq 2$ .
- Recent advances such as ResNet/GoogLeNet have challenged this paradigm



# Recurrent Neural Network

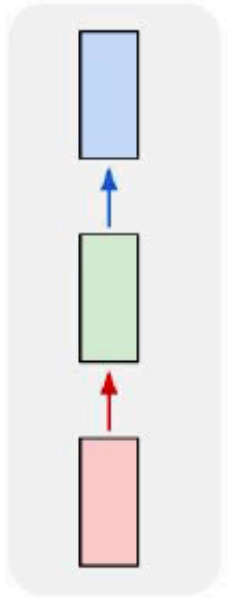
---

Slides edited from Stanford

[http://cs231n.stanford.edu/slides/2019/cs231n\\_2019\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture10.pdf)

# Types of Recurrent Neural Networks

one to one



Vanilla NN

one to many

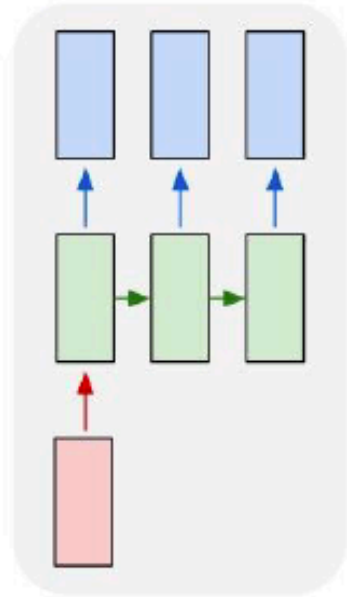
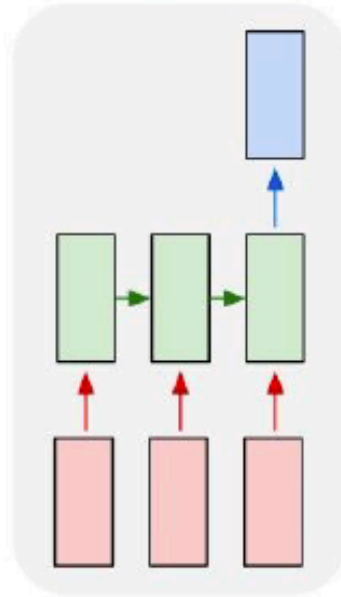


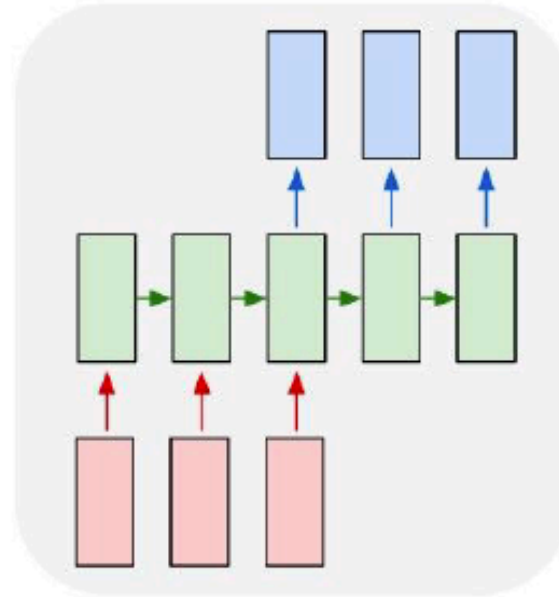
Image -->  
Sequence of Words  
Image Captioning

many to one



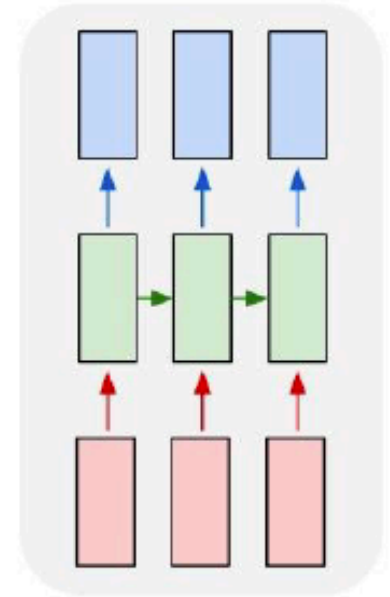
Sequence of Words -->  
Sentiment  
Sentiment Classification  
TS Classification

many to many



Sequence of Words -->  
Sequence of Words  
Machine Translation

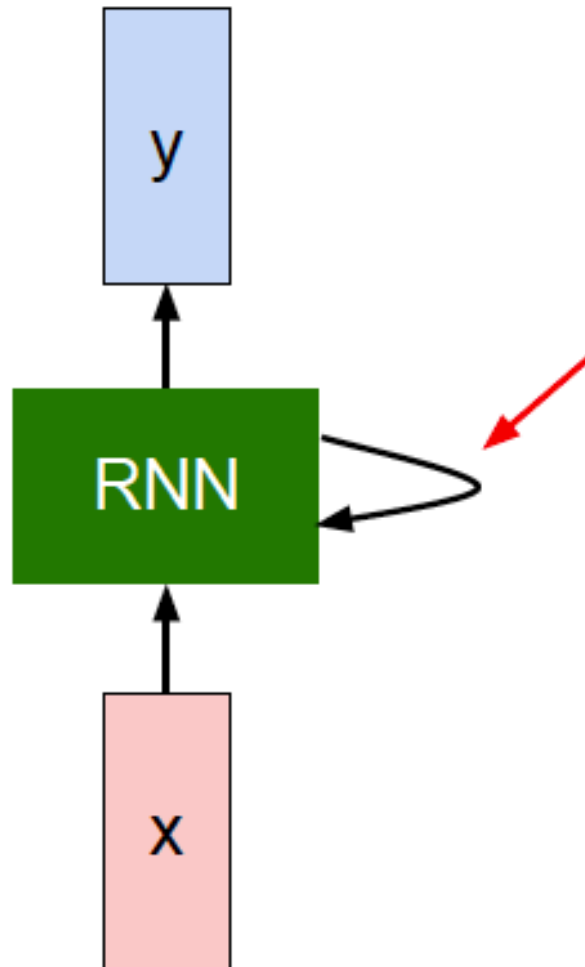
many to many



Video Classification

# Recurrent Neural Network - RNN

---



Key idea: RNNs have an “internal state” that is updated as a sequence is processed

# Recurrent Neural Network - RNN

- We can process a sequence of vectors  $\mathbf{x}$  by applying a *recurrence formula* at every time step:

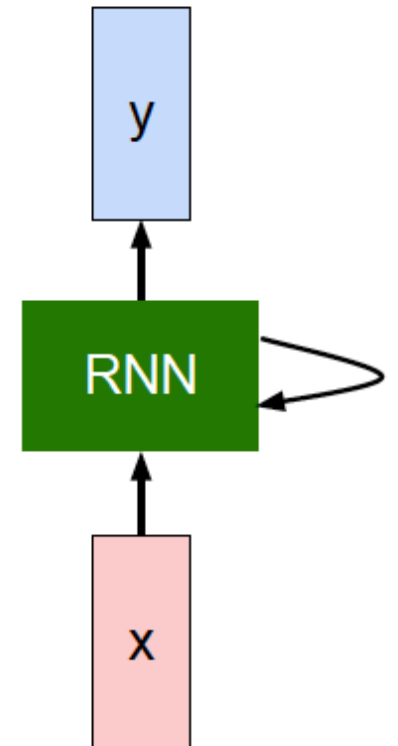
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

some function  
with parameters  $W$

old state

input vector at  
some time step



# (Simple) Recurrent Neural Network

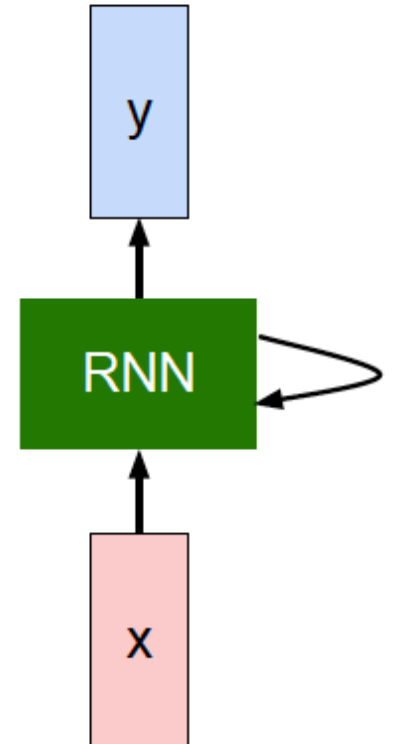
---

$$h_t = f_W(h_{t-1}, x_t)$$



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

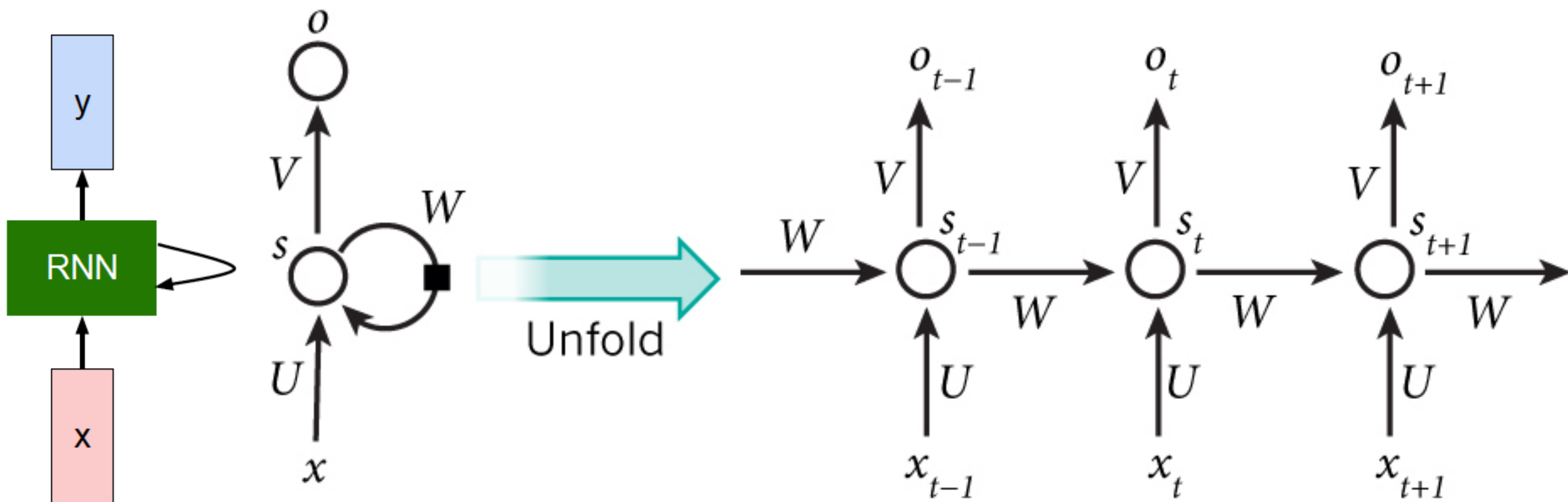


# RNN Idea

---

- The idea behind RNNs is to make use of sequential information.
- In a traditional NN we assume that all inputs (and outputs) are independent of each other.
- But for sequence dependent task this is a bad idea: if you want to predict the next word in a sentence you better know which words came before it.
- RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations.
- Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far.
- In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps

# Unfolded RNN



# Unfolded RNN

---

- $x_t$  is the input at time  $t$ . For example,  $x_1$  could be a one-hot vector corresponding to the second word of a sentence.
- $s_t$  is the hidden state at time  $t$ . It is the “memory” of the network.  $s_t$  is calculated based on the previous hidden state and the input at the current step:  $s_t = f(U x_t + W s_{t-1})$ .
- The function  $f$  is usually *tanh* or *ReLU*.  $s_{-1}$ , which is required to calculate the first hidden state, is typically initialized to all zeroes.
- $o_t$  is the output at time  $t$ . For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary.  $o_t = \text{softmax}(V s_t)$ .

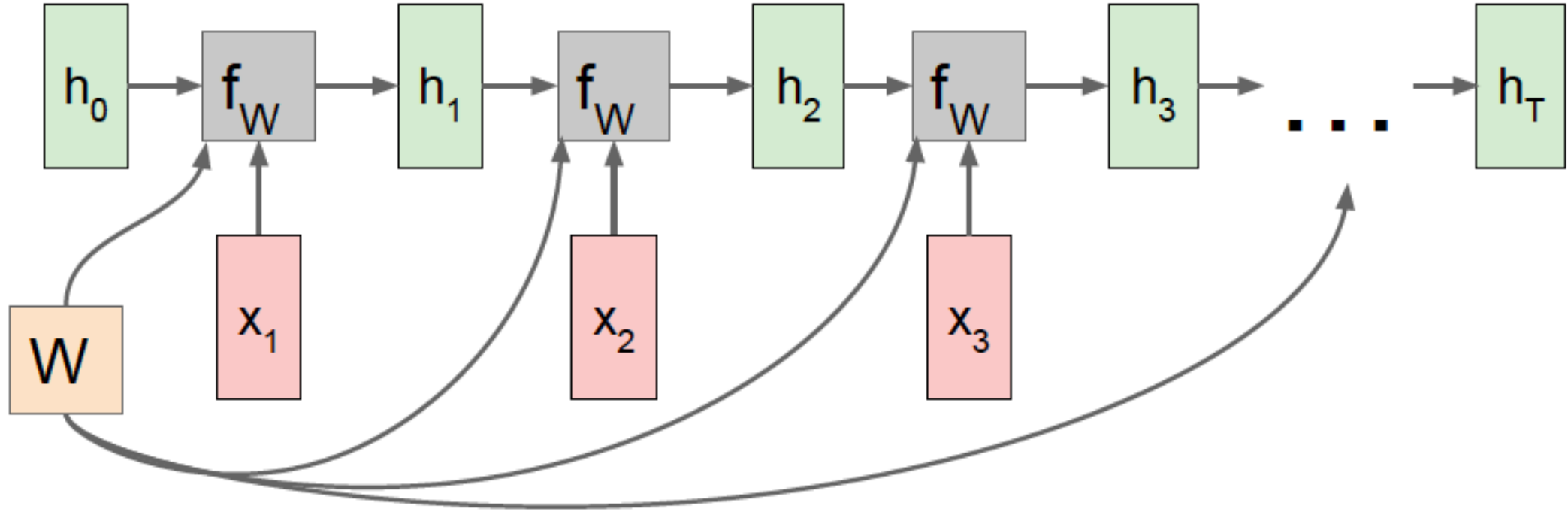


# Unfolded RNN

---

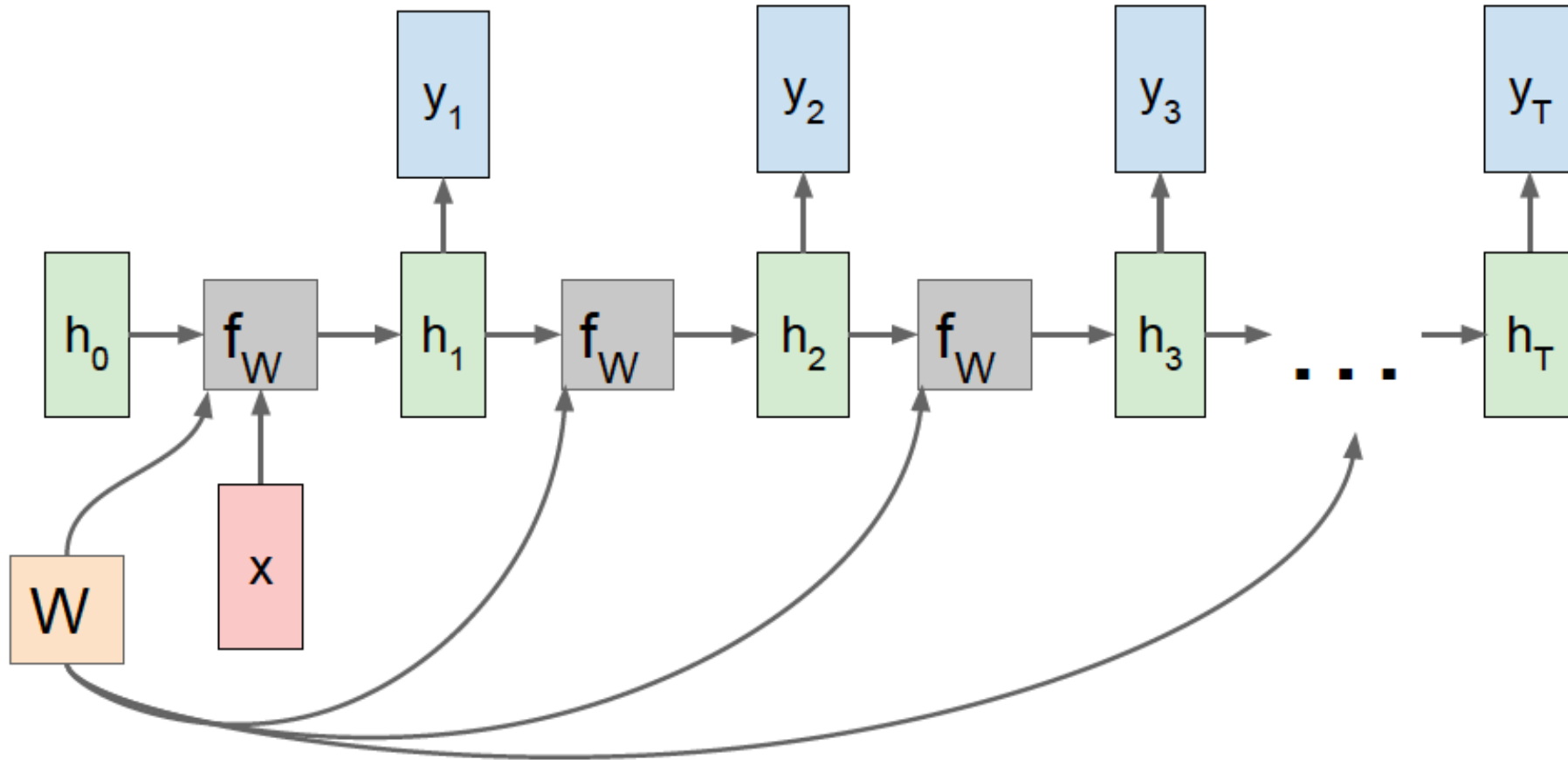
- The hidden state  $s_t$  is the memory of the network.  $s_t$  captures information about what happened in all the previous time steps.
- The output at step  $o_t$  is calculated solely based on the memory at time  $t$ .
- $s_t$  typically can not capture information from too many time steps ago.
- Unlike a DNN, which uses different parameters at each layer, a RNN shares the same parameters ( $U, V, W$ ) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs.
- The previous diagram has outputs at each time step, but depending on the task this may not be necessary.

# RNN: Computational Graph

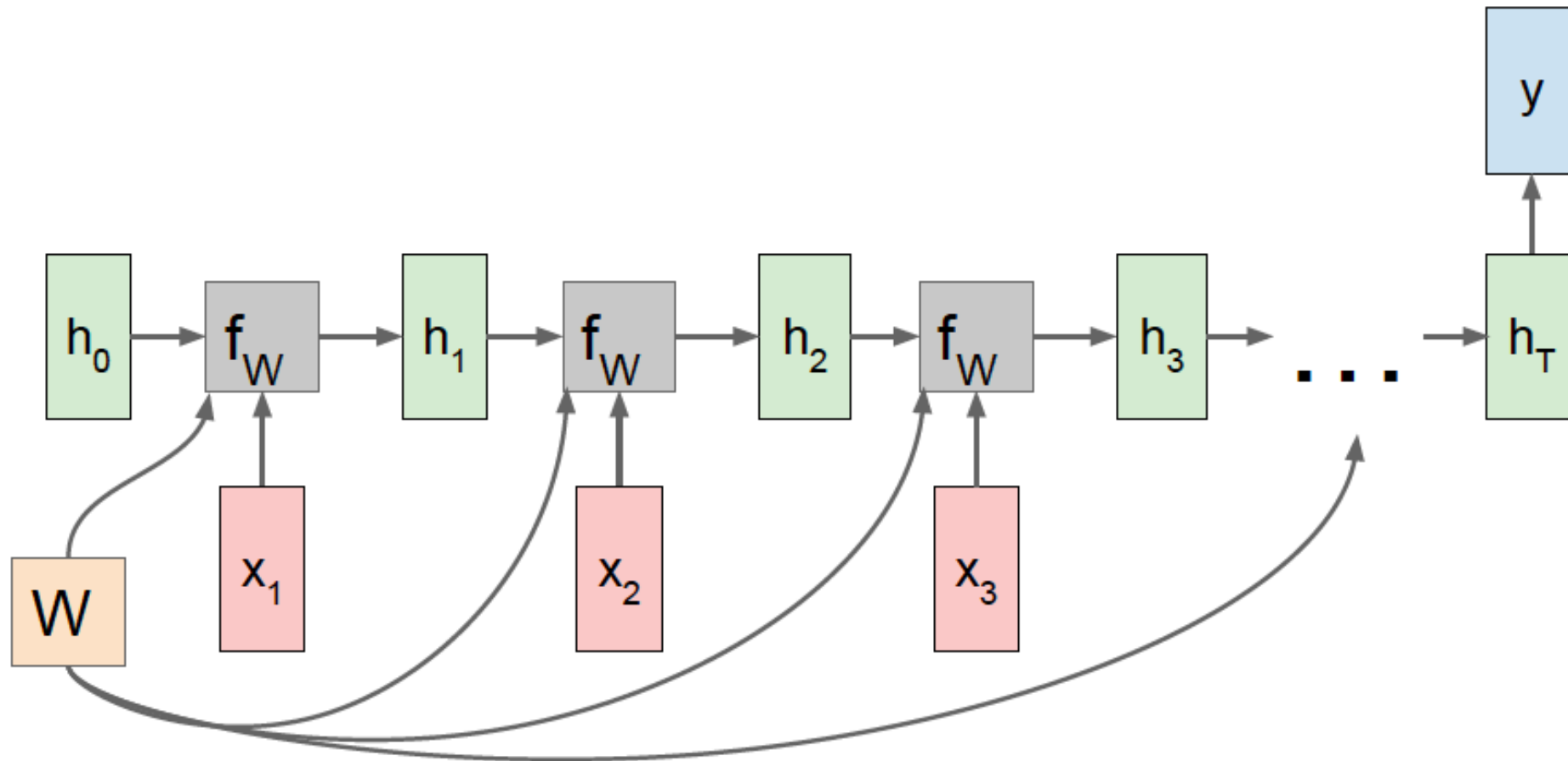


Reminder: Re-use the same weight matrix at every time-step

# RNN: Computational Graph: Many to Many



# RNN: Computational Graph: Many to One

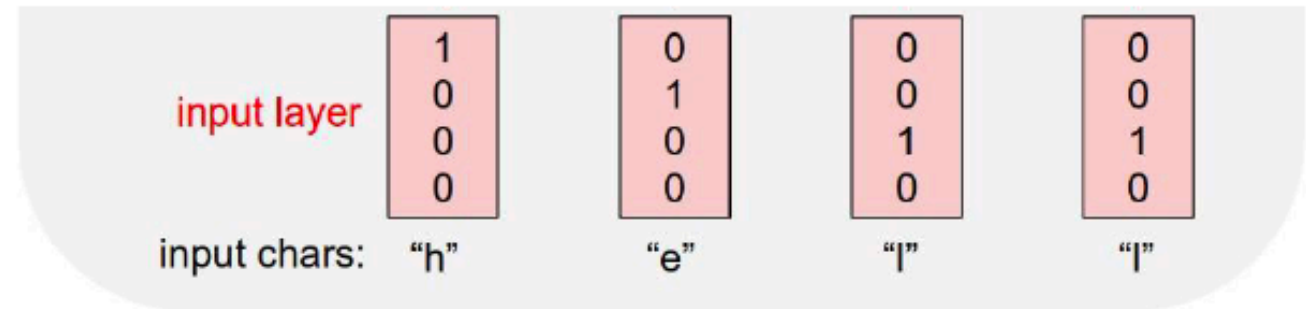


# RNN: Example Training

---

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”



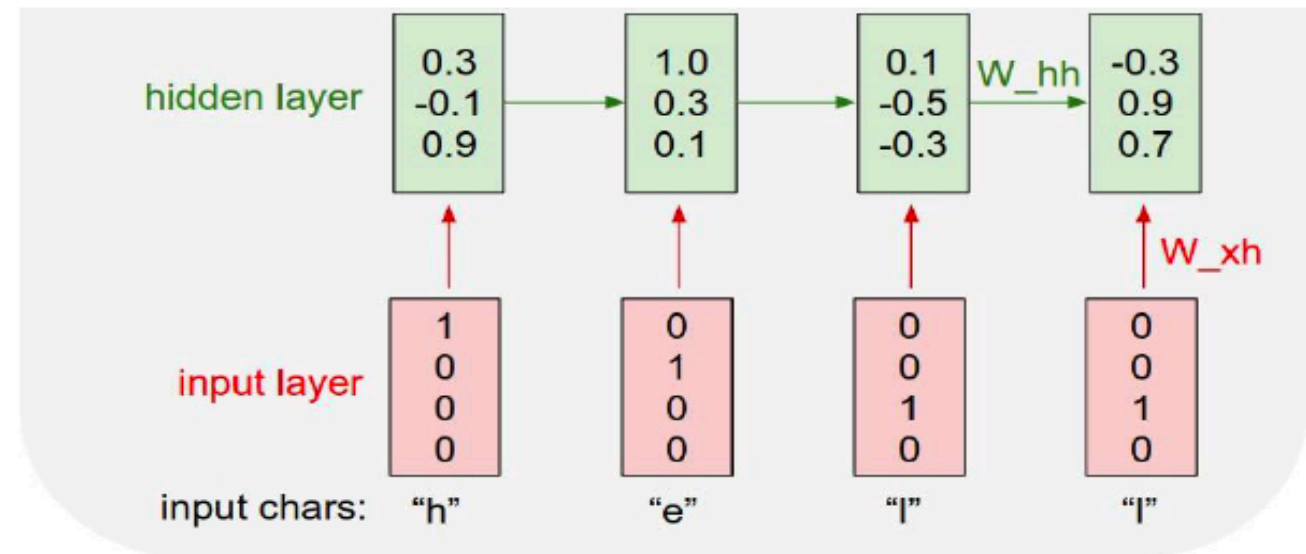
# RNN: Example Training

## Example: Character-level Language Model

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

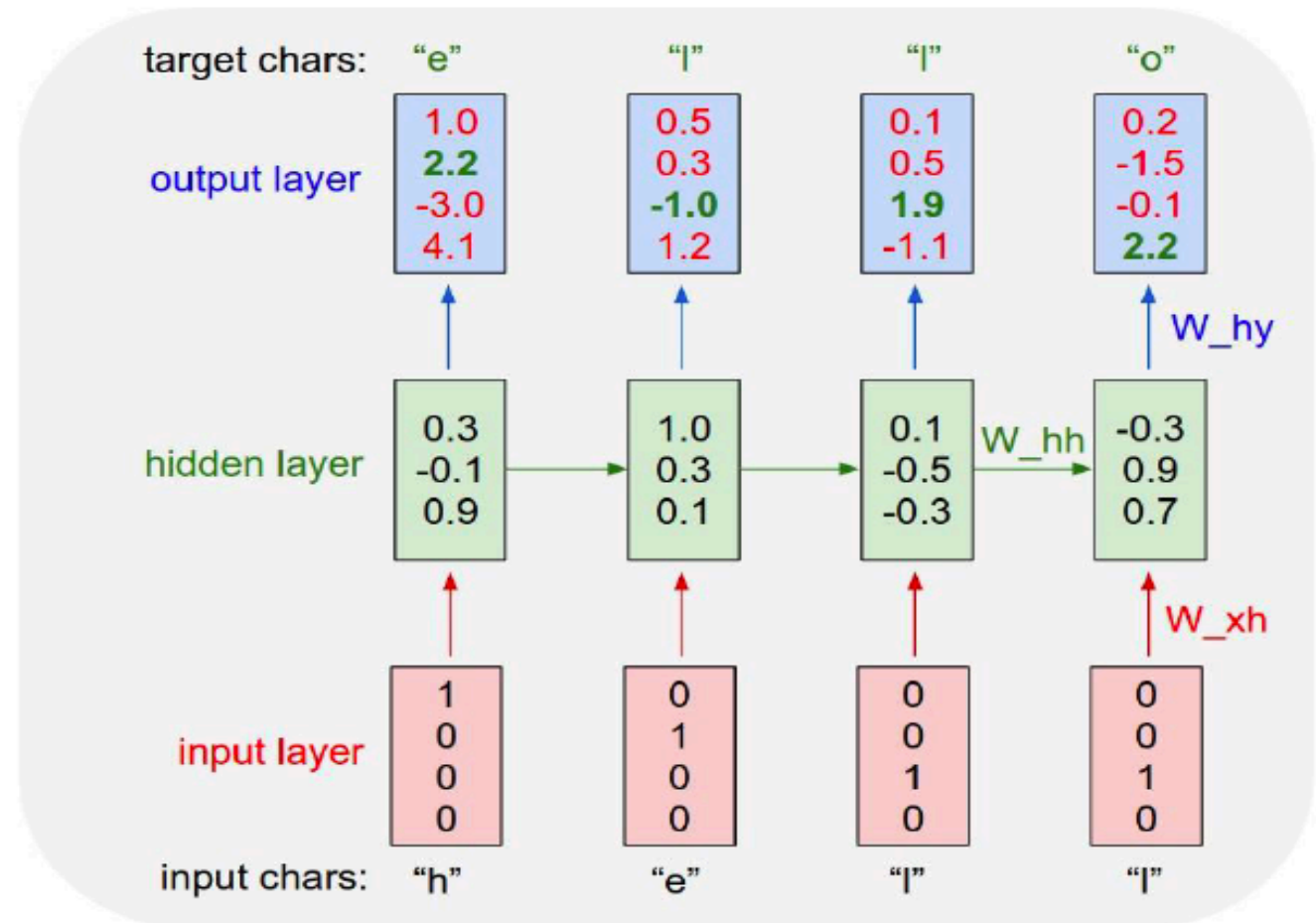


# RNN: Example Training

## Example: Character-level Language Model

Vocabulary:  
[h,e,l,o]

Example training  
sequence:  
“hello”

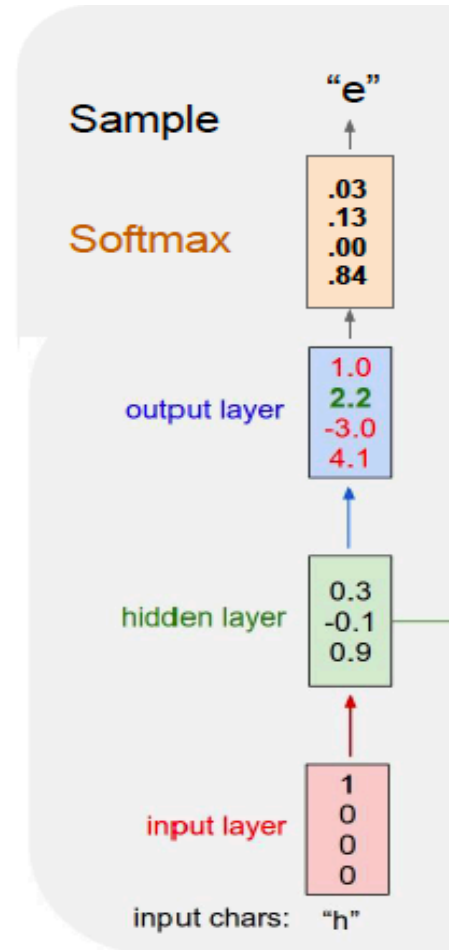


# RNN: Example Test

## Example: Character-level Language Model Sampling

Vocabulary:  
[h,e,l,o]

At test-time sample  
characters one at a time,  
feed back to model



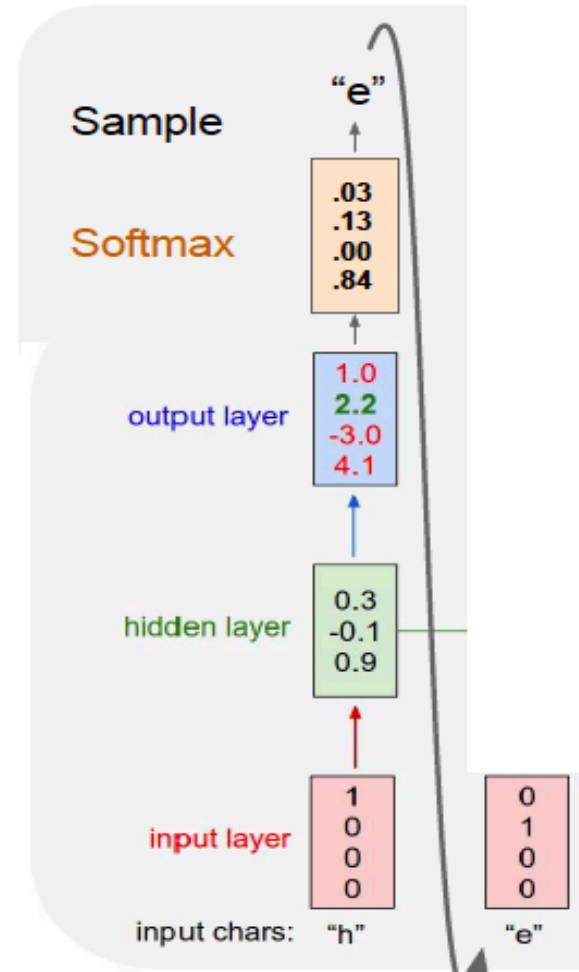


# RNN: Example Test

## Example: Character-level Language Model Sampling

Vocabulary:  
[h,e,l,o]

At test-time sample  
characters one at a time,  
feed back to model

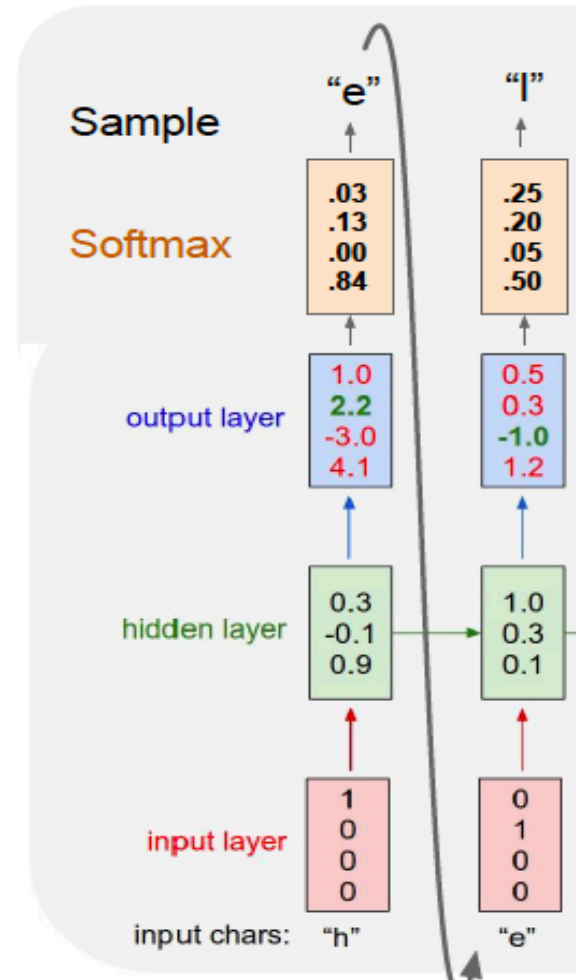


# RNN: Example Test

## Example: Character-level Language Model Sampling

Vocabulary:  
[h,e,l,o]

At test-time sample  
characters one at a time,  
feed back to model

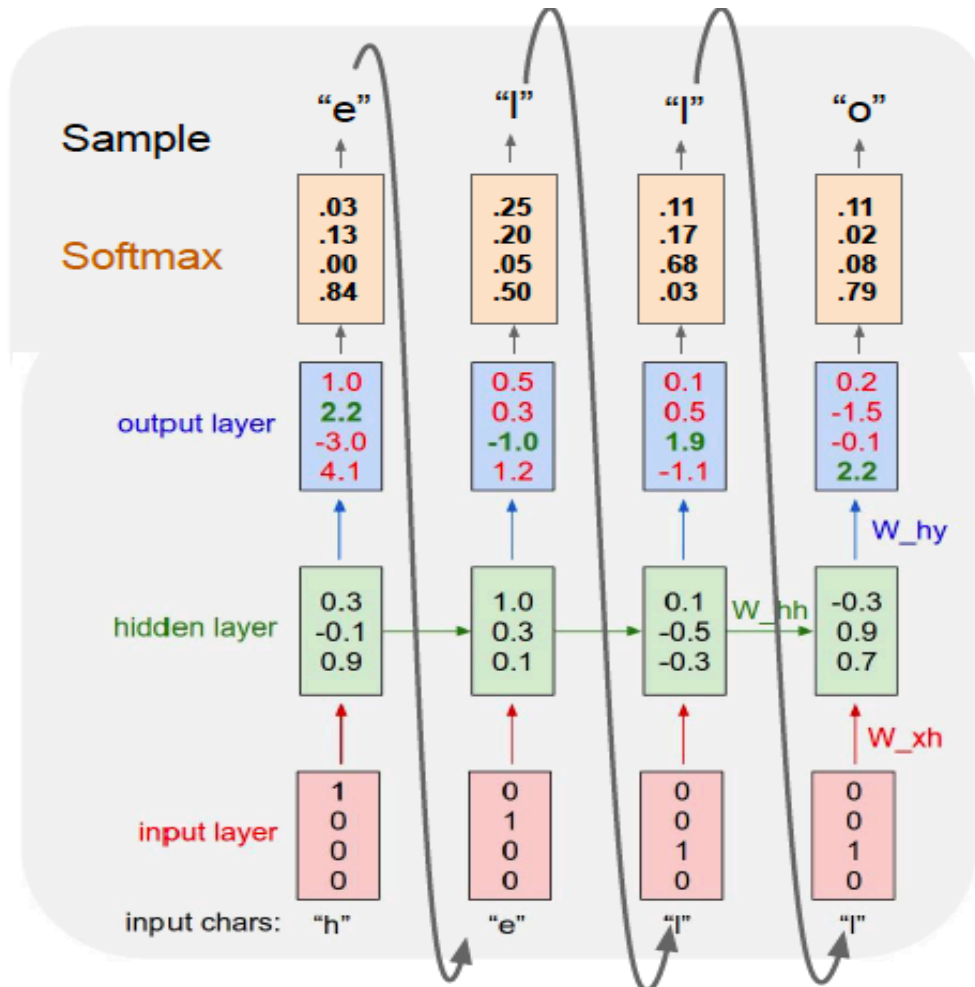


# RNN: Example Test

**Example:  
Character-level  
Language Model  
Sampling**

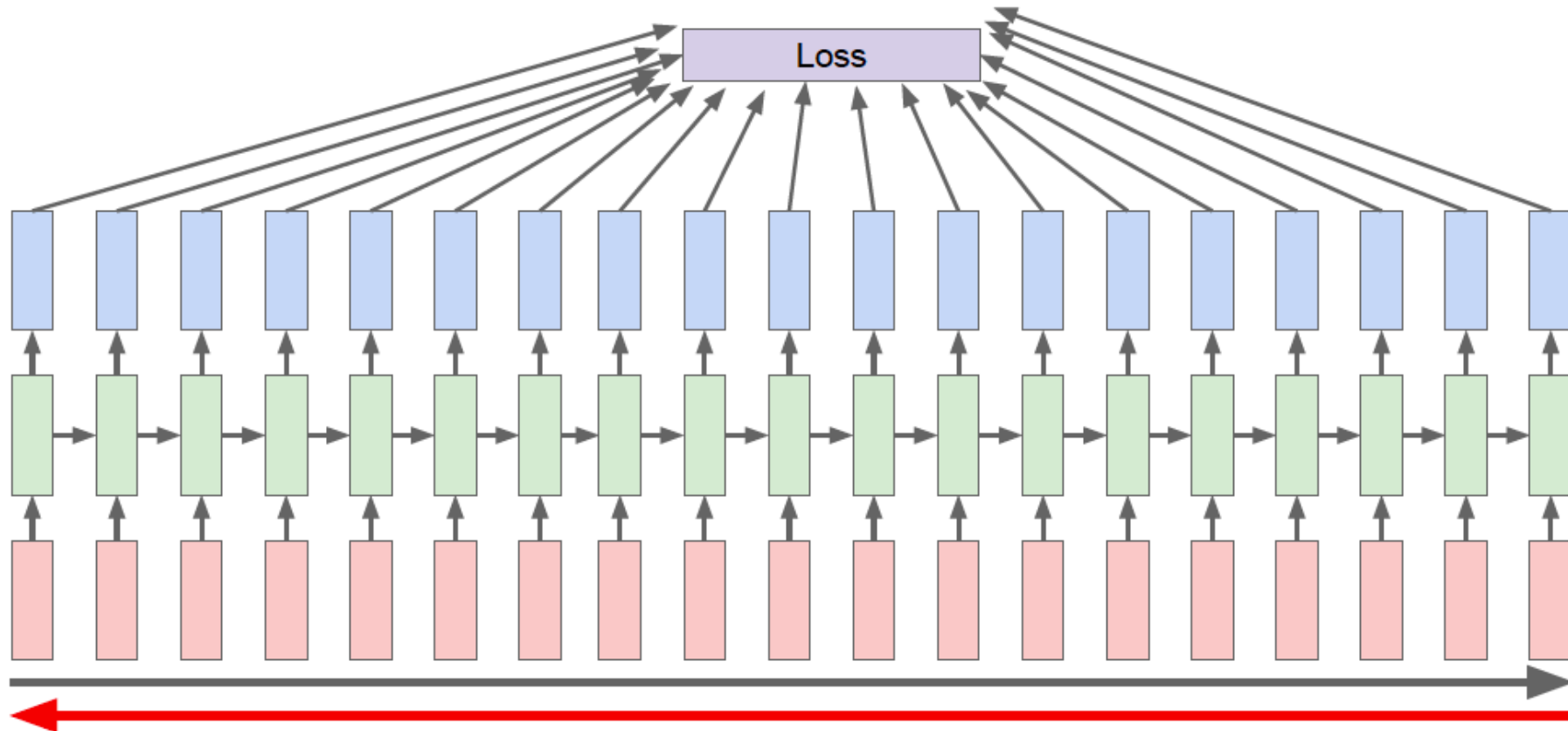
Vocabulary:  
[h,e,l,o]

At test-time sample  
characters one at a time,  
feed back to model



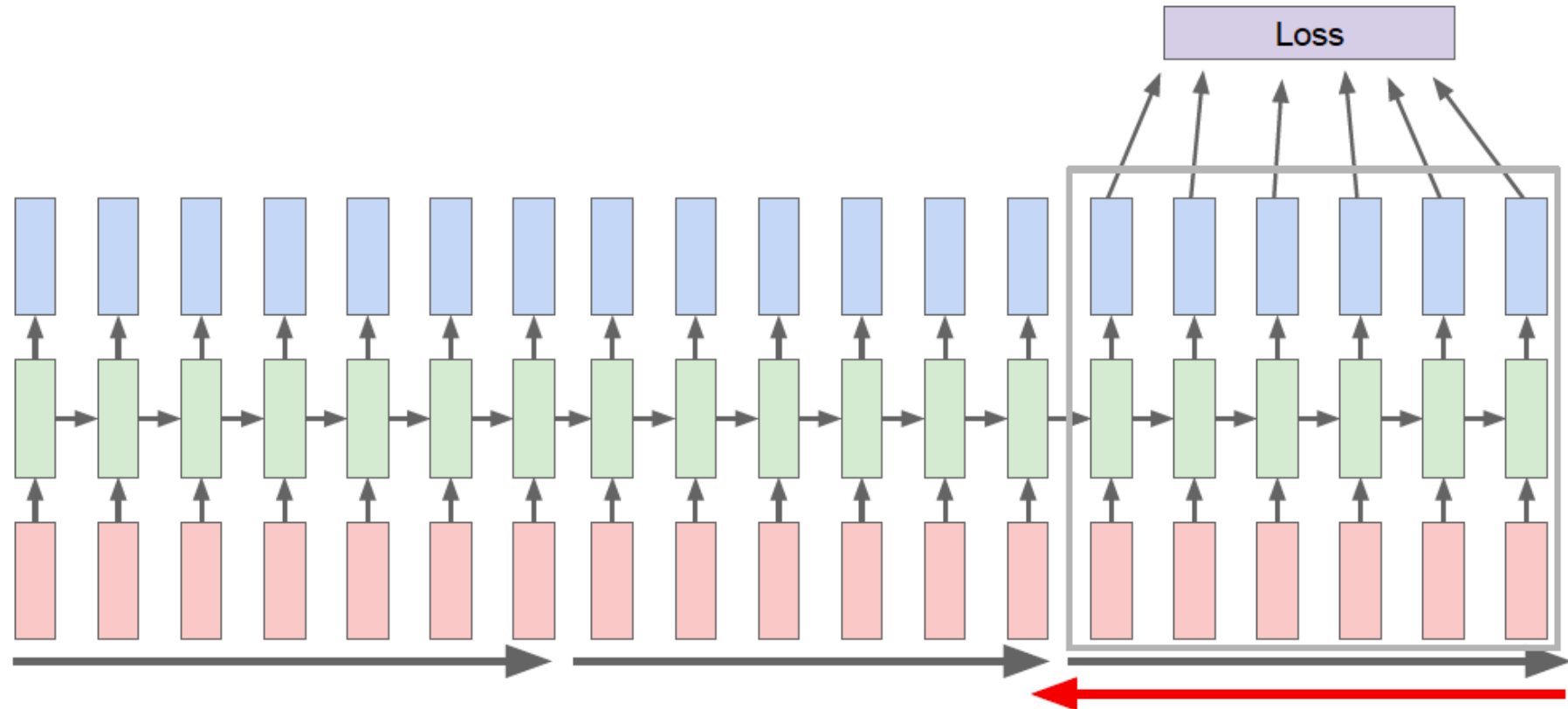
# Backpropagation Through Time

- Forward through entire sequence to compute loss
- Then backward through entire sequence to compute gradient



# Truncated BPTT

- It is an approximation of full BPTT that is preferred for long sequences since full BPTT's forward/backward cost per parameter update becomes very high over many time steps.
- The downside is that the gradient can only flow back so far due to that truncation, so the network can not learn dependencies that are as long as in full BPTT.



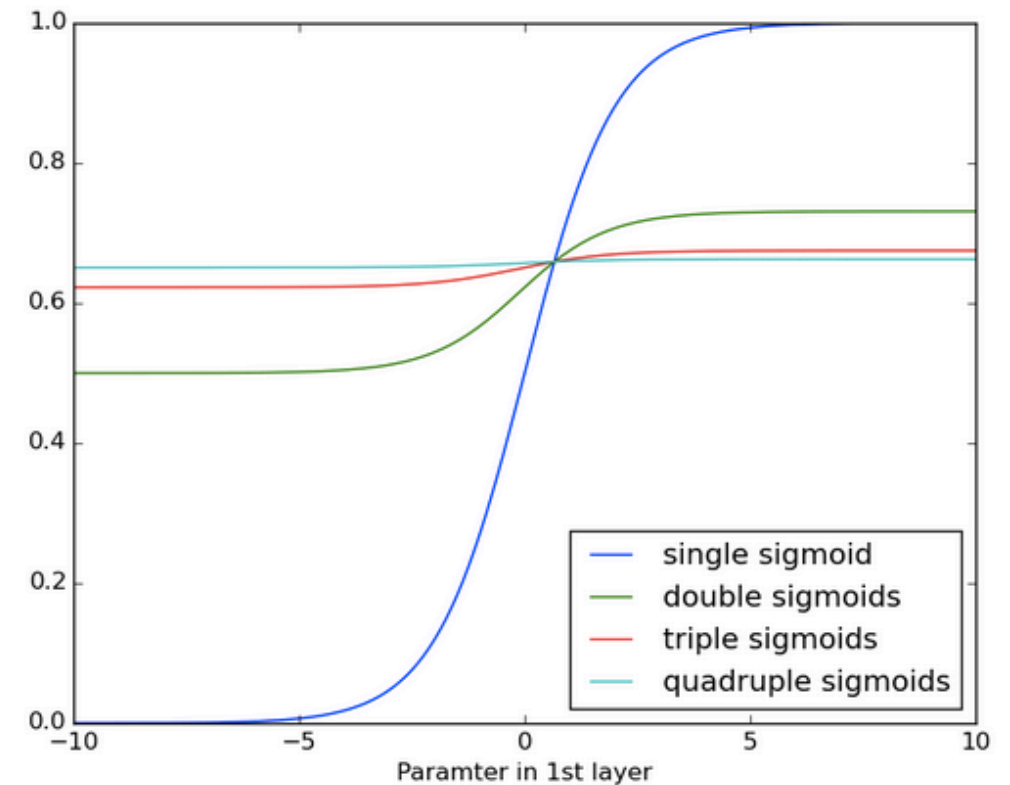
# Limitations of RNNs

---

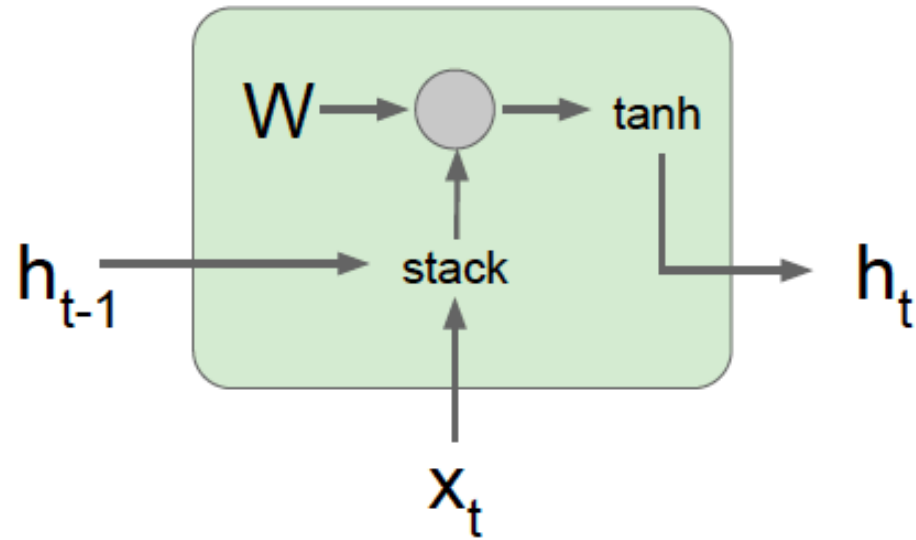
- RNN work fine when we are dealing with *short-term* dependencies.
- However, RNNs fail to understand the context behind an input.
- For instance, something that was said long before, cannot be recalled when making predictions in the present.
- The reason behind this is the problem of *Vanishing Gradient*.
- For a DNN, the weight updating that is applied on a particular layer is a multiple of the learning rate, the error term from the previous layer and the input to that layer. The error for a particular layer is a product of all previous layers' errors.
- When dealing with functions like *sigmoid/tanh*, the small values of its derivatives (occurring in the error function) gets multiplied multiple times as we move towards the starting layers. As a result of this, the gradient almost vanishes as we move towards the starting layers, and it becomes difficult to train these layers.

# Vanishing (and Exploding) Gradients

- The gradient expresses the change in all weights with regard to the change in error.
- If we can not know the gradient, we can not adjust the weights in a direction that will decrease error, and our network ceases to learn.
- Effects of applying a sigmoid function over and over again.



# Vanilla RNN Gradient Flow



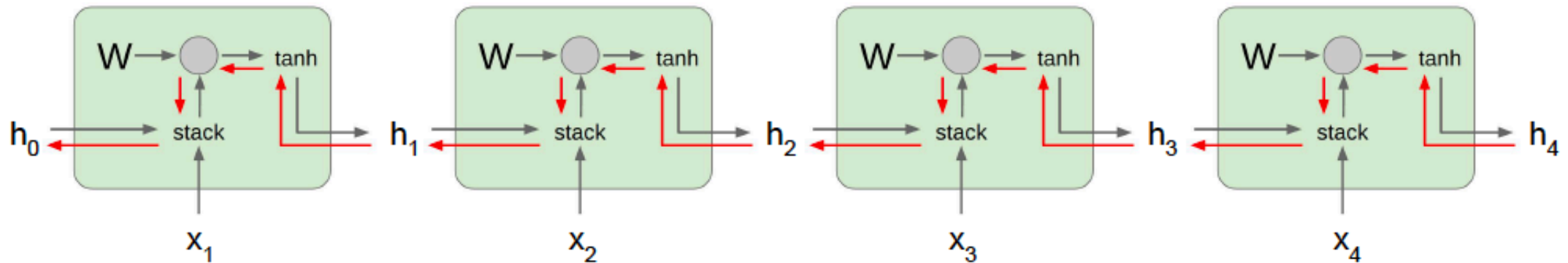
$$\begin{aligned}h_t &= \tanh(W_{hh}h_{t-1} + W_{hx}x_t) \\ &= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\ &= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)\end{aligned}$$

Backpropagation from  $h_t$   
to  $h_{t-1}$  multiplies by  $W$



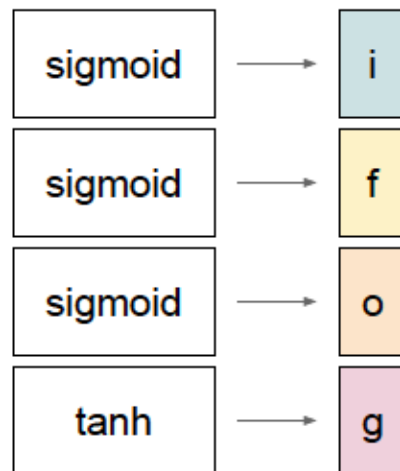
# Vanilla RNN Gradient Flow

- Computing gradient of  $h_0$  involves many factors of  $W$  (and repeated  $\tanh$ )
- Largest singular value  $> 1 \rightarrow$  **Exploding Gradients**
  - Gradient clipping: Scale Computing gradient if its norm is too big
- Largest singular value  $< 1 \rightarrow$  **Vanishing Gradients**
  - Change RNN architecture



# Long Short Term Memory (LSTM)

- LSTM contains in a gated cell information outside the normal flow of the recurrent network.
- Information can be stored in, written to, or read from a cell.



## Vanilla RNN

$$h_t = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

## LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

# Long Short Term Memory (LSTM)

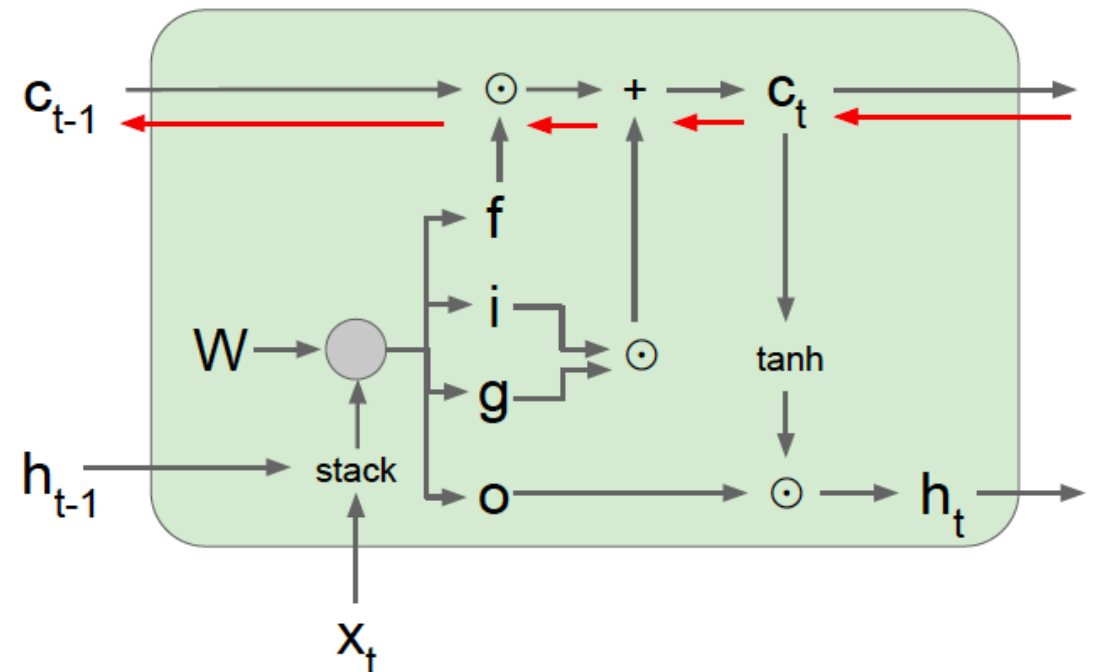
- The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close.
- These gates are implemented with element-wise multiplication by sigmoids, which are all in the range of 0-1, thus are differentiable and suitable for backpropagation

Backpropagation from  $c_t$  to  $c_{t-1}$  only elementwise multiplication by  $f$ , no matrix multiply by  $W$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

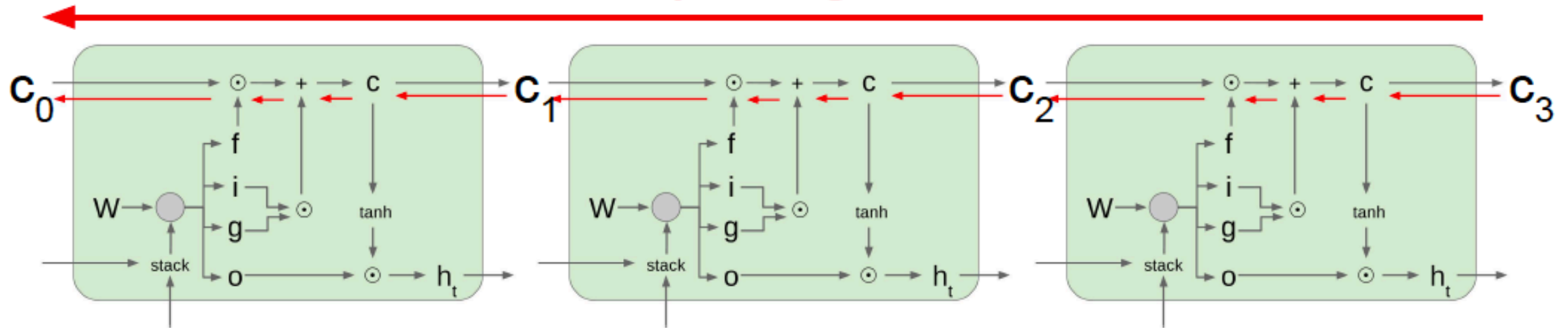
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

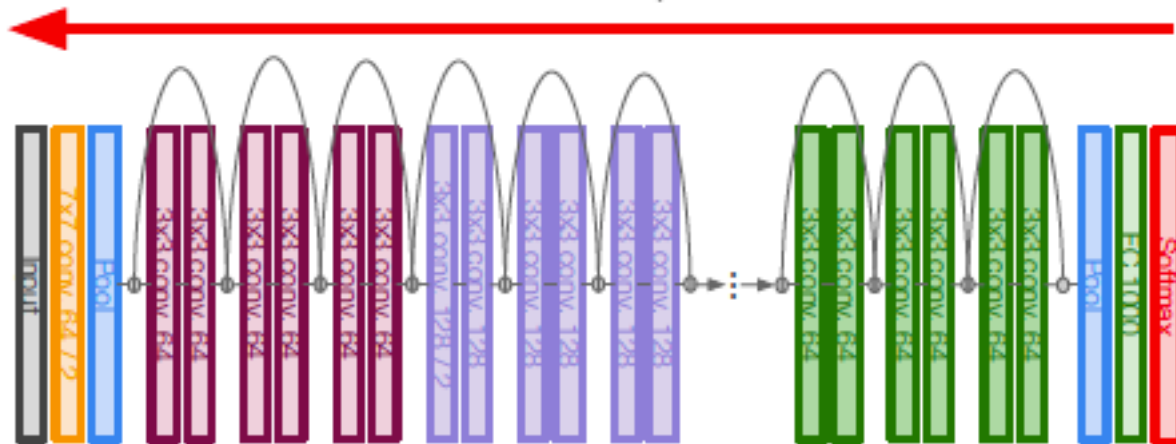


# Long Short Term Memory (LSTM): Gradient Flow

Uninterrupted gradient flow!



Similar to ResNet!



# RNN Summary

---

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research, as well as new paradigms for reasoning over sequences
- Better understanding (both theoretical and empirical) is needed.

# References

- Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. Chin-Chia Michael Yeh et al. 1997
- Time Series Shapelets: A New Primitive for Data Mining. Lexiang Ye and Eamonn Keogh. 2016.
- Josif Grabocka, Nicolas Schilling, Martin Wistuba, Lars Schmidt-Thieme (2014): Learning Time-Series Shapelets, in Proceedings of the 20th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2014
- Deep learning for time series classification: a review. Hassan Ismail Fawaz et al. 2019.

arXiv:1809.0436v4 [cs.LG] 14 May 2019

## Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets

Chin-Chia Michael Yeh, Yan Zhu, Lindmila Ulanova, Nurjahan Begam, Yifei Ding, Hong Anh Dao, Diego Furtado Silva, Abdallah Mounir, and Eamonn Keogh  
University of California, Riverside, Universidade de São Paulo, University of New Mexico  
{mrb003, yzhu001, abeg001, yzhu001, hdao001}@ucr.edu, dergofab@ucsb.edu.br, amoun@cs.ucr.edu

**Abstract**—The all-pairs-similarity-search (or similarity join) problem has been extensively studied for text and a handful of other domains. However, surprisingly little progress has been made on similarity joins for time series subsequences. The lack of progress probably stems from the daunting nature of the problem. For even modest sized datasets the obvious nested-loop algorithm can take months, and typical speed-up techniques in this domain (e.g., indexing, lower-bounding, irregularly-incrementally pruning and early-abandonment) at best produce one or two orders of magnitude speedup. In this work we introduce a novel scalable algorithm for time series subsequence all-pairs-similarity-search. For exceptionally large datasets, the algorithm can be trivially cast as an anytime algorithm and produce high-quality approximate solutions in reasonable time. The exact similarity join algorithm computes the answer to the time series motif and time series discord problem as a side-effect, and our algorithm incidentally provides the fastest known algorithm for both these extensively-studied problems. We demonstrate the utility of our ideas for many time series data mining problems, including motif discovery, anomaly discovery, shapelet discovery, semantic segmentation, density estimation, and contrast set mining.

**Keywords**—Time Series; Similarity Joins; Motif Discovery

**I. INTRODUCTION**  
The all-pairs-similarity-search (also known as similarity join) problem comes in several variants. The basic task is: Given a collection of data objects, retrieve the nearest neighbor for each object in the text domain; the algorithm has applications in a host of problems, including community discovery, duplicate detection, collaborative filtering, clustering, and query refinement [1]. While virtually all text processing algorithms have advantages in time series data mining, there has been surprisingly little progress on Time Series subsequence All-Pairs-Similarity-Search (TSAPSS).

We believe that this lack of progress stems not from a lack of interest in this useful primitive, but from the daunting nature of the problem. Consider the following example that reflects the needs of an industrial collaborator. A boiler at a chemical refinery reports pressure once a minute. After a year, we have a time series of length 524,608. A plant manager may wish to do a similarity self-join on this data with week-long subsequences (10,800) to discover repeating regimes (summer vs. winter or light distillate vs. heavy distillate, etc). The obvious nested-loop algorithm requires 112,800,602,560 Euclidean distance computations. If we assume each one takes 0.0001 seconds, then the join will take 11.8 days. The core combination of this work is to show that we can reduce this time to 4.3 hours, using an off-the-shelf desktop computer. Moreover, we show that this join can be computed and/or updated incrementally. This we could maintain this join essentially forever on a standard

This is the author's version of an article published in Data Mining and authenticated version is available online at: [https://doi.org/10.1007/978-1-4939-9909-6\\_15](https://doi.org/10.1007/978-1-4939-9909-6_15)

### Deep learning for time series classification

Hassan Ismail Fawaz<sup>1</sup> · Germain Forestier<sup>1,2</sup> · Jonathan W. Lussane Iduonghar<sup>1</sup> · Pierre-Alain Muller<sup>1</sup>

**Abstract** Time Series Classification (TSC) is an important and challenging task. With the increase of time series data availability, hundreds of TSC algorithms have been proposed since 2015 (Bagnall et al., 2017). Among these methods, only a few have considered Deep Neural Net task. This is surprising as deep learning has been very successful apply have indeed revolutionized the field of computer vision especially architectures such as Residual and Convolutional Neural Networks. In this work we present a novel time series data mining primitive called time series shapelets. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. While we believe there can be many uses in data mining, one obvious implication of them is to mitigate the two well-known algorithmic bottlenecks of TSC: the computational cost of distance computation on multi-epoch or long spans of all or part of the data for persons in classrooms (as is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear the notice and full citation on the first page. To copy copies in any form, or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, requires prior specific permission and/or a fee. Copyright 2009 ACM 978-1-4939-9909-6...\$5.00.

**Keywords** Deep learning · Time series · Classification · Review

### 1 Introduction

During the last two decades, Time Series Classification (TSC) has been considered as one of the most challenging problems in data mining (Yang and Wu, 2006; Ealing and Agon, 2012). With the increase of temporal data availability (Silva et al., 2018), hundreds of TSC algorithms have been proposed since 2015 (Bagnall et al., 2017). Due to their natural temporal ordering, time series data are present in almost every task that requires some sort of human cognitive process (Langkvist et al., 2014). In fact, any classification problem, using data that is registered taking into account some notion of ordering, can be cast as a TSC problem (Christian Borges Goncalves, 2017). Time series are encountered in many real-world applications ranging from electronic health records (Rajiljman et al., 2018) and human activity recognition (Nweke et al., 2018; Wang et al., 2018) to acoustic scene classification (Nwe et al., 2017) and cyber-security (Susto et al., 2018). In addition, the diversity of the datasets’ types in the UCR/UEA archive (Chen et al., 2015b; Bagnall et al., 2017) (the largest repository of time series datasets) shows the different applications of the TSC problem.

© H. Ismail Fawaz  
E-mail: hassan.ismail.fawaz@uniba.fr  
<sup>1</sup>IRIMAS, Université Haute Alsace, Mulhouse, France  
<sup>2</sup>Faculty of IT, Monash University, Melbourne, Australia

desktop, even if the data arrival frequency was much faster than one a minute.

Our algorithm uses an ultra-fast similarity-search algorithm under a constant time Euclidean distance as a subroutine, exploiting the overlap between subsequences using the classic Fast Fourier Transform (FFT) algorithm.

Our method has the following advantages/features:

- It is exact, providing no false positives or false dismissals.
- It is simple and parameter-free. In contrast, the more general metric space APSS algorithms require building and tuning spatial access methods and/or hash functions.
- Our algorithm requires an inconsequential space overhead, just  $O(n)$  with a small constant factor.
- While our exact algorithm is extremely scalable, for extremely large datasets we can compute the results in an anytime fashion, allowing ultra-fast approximate solutions.
- Having computed the similarity join for a dataset, we can incrementally update it very efficiently. In many domains this means we can effectively maintain exact joins on streaming data forever.
- Our method provides full joins, eliminating the need to specify a similarity threshold, which as we will show, is a near impossible task in this domain.
- Our algorithm is embarrassingly parallelizable, both on multicore processors and on distributed systems.

## Time Series Shapelets: A New Primitive for Data Mining

Lexiang Ye Dept. of Computer Science & Engineering, University of California, Riverside, CA 92521, lexiang@cs.ucr.edu  
Eamonn Keogh Dept. of Computer Science & Engineering, University of California, Riverside, CA 92521, eamonn@cs.ucr.edu

**ABSTRACT**  
Classification of time series has been attracting great interest over the past decade. Recent empirical evidence has strongly suggested that the simple nearest neighbor algorithm is very difficult to beat for most time series problems. While this may be considered good news, given the simplicity of implementing the nearest neighbor algorithm, there are some negative consequences of this. First, the nearest neighbor algorithm requires storing and searching the entire dataset, resulting in a time and space complexity that limits its applicability especially on large datasets. Second, beyond near classification accuracy, we often wish to gain some insight into the data.

In this work we introduce a new time series primitive, time series shapelets, which addresses these limitations. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. As we shall show with extensive empirical evaluation on diverse datasets, algorithms based on the time series shapelet primitives can be interpreted, more accurate and significantly faster than state-of-the-art classifiers.

**Categories and Subject Descriptors** H.2.8 (Database Management): Database Applications · Data Mining

**General Terms** Algorithms; Experimentation

### 1. INTRODUCTION

While the last decade has seen a huge interest in time series classification, to date the most accurate and robust method is the simple nearest neighbor algorithm [4][12][14]. While the nearest neighbor algorithm has the advantages of simplicity and not requiring extensive parameter tuning, it does have several important disadvantages. Chief among these are its space and time requirements, and the fact that it does not tell us anything about why a particular object was assigned to a particular class.

In this work we present a novel time series data mining primitive called time series shapelets. Informally, shapelets are time series subsequences which are in some sense maximally representative of a class. While we believe there can be many uses in data mining, one obvious implication of them is to mitigate the two well-known algorithmic bottlenecks of TSC: the computational cost of distance computation on multi-epoch or long spans of all or part of the data for persons in classrooms (as is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear the notice and full citation on the first page. To copy copies in any form, or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, requires prior specific permission and/or a fee. Copyright 2009 ACM 978-1-4939-9909-6...\$5.00.

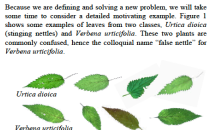


Figure 1: Sample of leaves from two species. Note that several leaves have the insect-bite damage.

Suppose we wish to build a classifier to distinguish these two plants, what features should we use? Since the area-variability of color and size within each class completely deters the inter-variability between classes, our best hope is based on the shape of the leaves. However, as we can see in Figure 1, the difference in the global shape are very subtle. Furthermore, it is very common for leaves to have distinctive or “eccentric” due to insect damage, and these are likely to confound any global measures of shape. Instead we attempt the following. We first convert each leaf into a one-dimensional representation as shown in Figure 2.

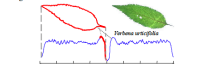


Figure 2: A shape can be converted into a one-dimensional “time series” representation. The curves for the highlighted section of the time series will be much simpler than this.

Such representations have been successfully used for the classification, clustering and outlier detection of shapes in recent years [8]. However, here we find that using a nearest neighbor classifier with either the (position-invariant) Euclidean distance or Dynamic Time Warping (DTW) distance does not significantly outperform random guessing. The reason for the poor performance of these otherwise very competitive classifiers seems to be due to the fact that the data is somewhat noisy (i.e. insect bite, and different insect bites), and this noise is enough to swamp the subtle differences in the shapes.