



Machine Learning

Lesson #2 for Unicredit Services

Executive summary

Overview

2

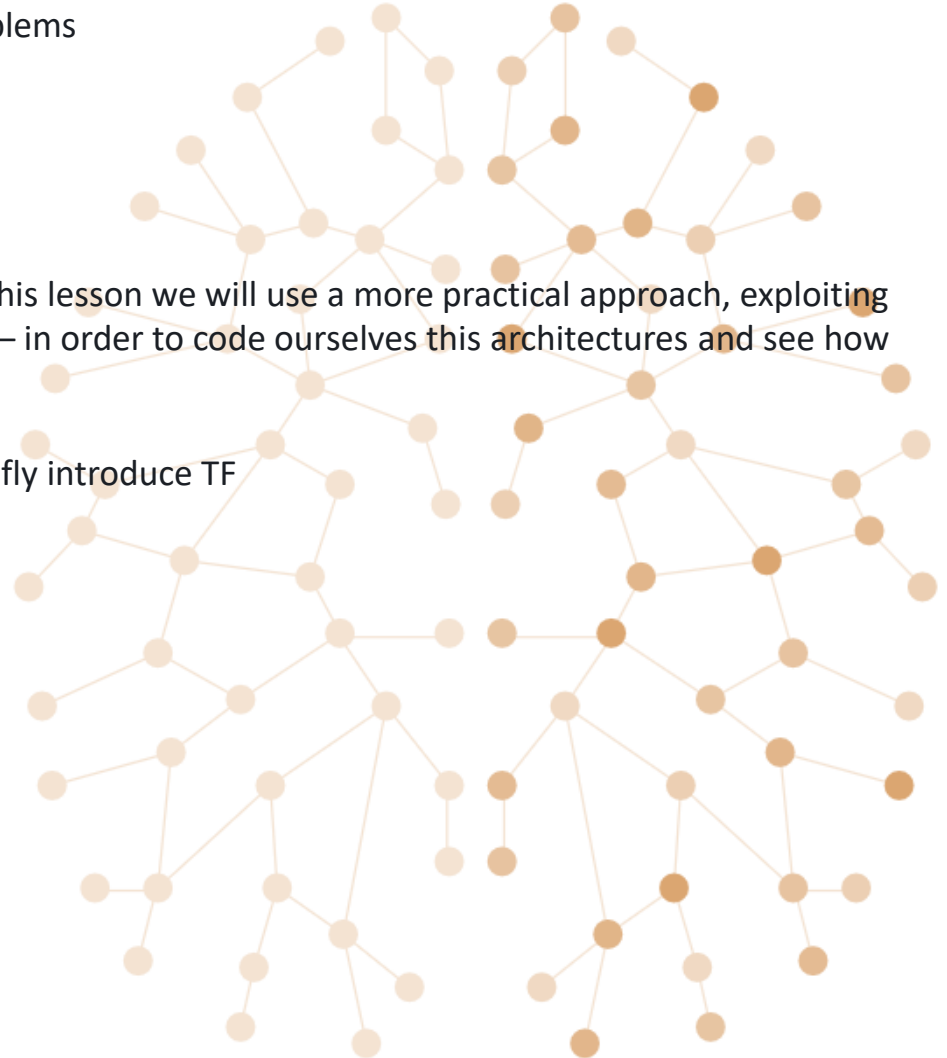
Having introduced the basics of feedforward Neural Networks (NN) in the first lesson, in this second one we will focus on two different architectures, widely used in practice for different problems

These two architectures are:

- **Convolutional Neural Networks (CNNs)**
- **Recurrent Neural Networks (RNNs)**

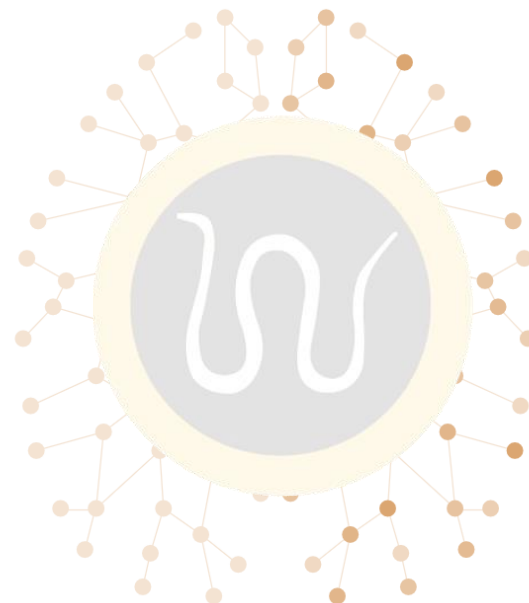
Said that we want to introduce and explain CNNs and RNNs, in this lesson we will use a more practical approach, exploiting one of the most used open source ML library – **TensorFlow (TF)** – in order to code ourselves this architectures and see how they can be used in a first use case

To do so, before diving directly into CNNs and RNNs, we will briefly introduce TF



Agenda

3

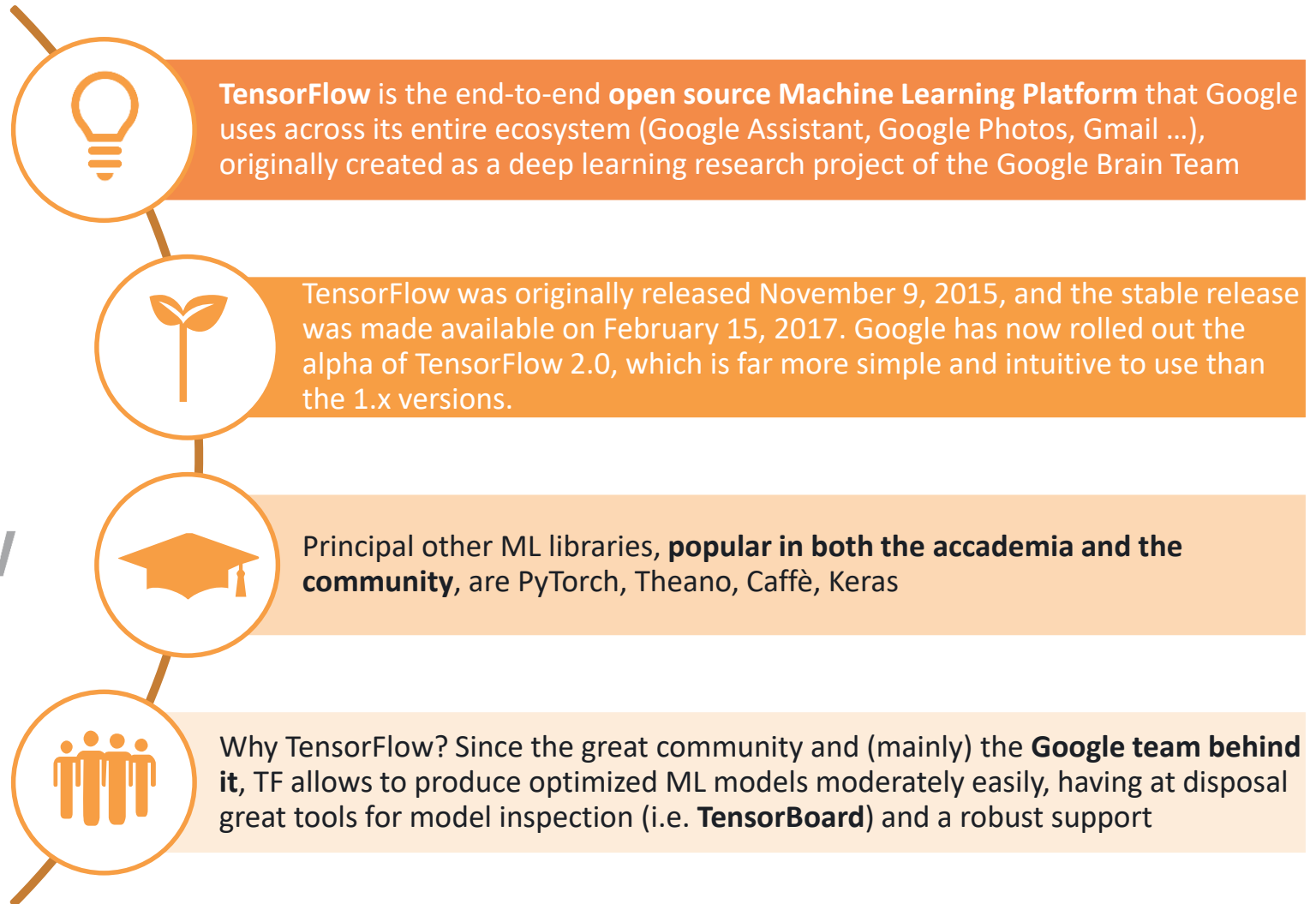


01	TensorFlow, a brief introduction	3
02	Convolutional Neural Networks	6
03	Recurrent Neural Networks	19

TensorFlow, a brief introduction

What is TF?

4



TensorFlow, a brief introduction

Main concepts to keep in mind

5

In the first release, back in 2015, TF was quite a complex library to use ... The user had to handle concepts like **placeholders** and **variables**, **graphs**, **sessions**, **eager execution**, ...

Even if it was a powerful tool for ML research and for production software, it **was quite an headache for the mid user**.

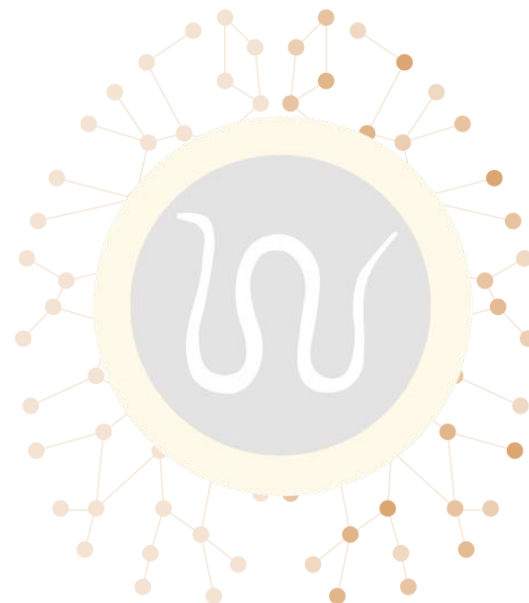
Since the second generation, **TF 2.x** released in the H2 of 2019, TF has now become **far more user-friendly**, starting to gain back users lost in favour of other simpler frameworks like, just to name one, PyTorch.

Just a few snapshots of the potentialities TF provides:

- The **Sequential API**, TF allows to **create models simply as plugging together different building blocks already prepared and highly customizable**. This has been made possible thanks to the integration with one of the most used high level ML interface, Keras
- The **Subclassing API**, more advanced users can **create a class for their model**, then write the forward pass imperatively, having easily the possibility to code custom layers, activations, and training loops.
- **TensorBoard**, a web-based platform that reads TF logs, produced during the training/ evaluation phase, and provide the user with different functionalities to **diagnose** and **dissect the model and its components**
- **TensorFlow Hub**, a repository for many **pre-trained models** that can be downloaded and integrated directly into the user workflow (i.e. with some Transfer Learning)

Agenda

6



01	TensorFlow, a brief introduction	3
02	Convolutional Neural Networks	6
03	Recurrent Neural Networks	19



Convolutional Neural Networks (CNNs) are a class of deep neural networks which put their foundations around a very specific mathematical, the **convolution** – from there arises their name



Provided the description of a general NN, CNNs are no different in their core components (i.e. neurons, layers, activations, ...) and mechanism of training and evaluation



CNNs were first introduced for solving **image recognition** problems as:

- **Classify** images (i.e. name what they see)
- **Cluster** images by similarity (photo search)
- Perform object **recognition** within scenes

It's also **due to their efficacy in such tasks that deep learning frameworks became that popular** in the recent years. The success of a deep convolutional architecture called **AlexNet** in the 2012 ImageNet competition was the shot heard round the world.

CNNs are **powering major advances in computer vision (CV)**, which has obvious applications for self-driving cars, robotics, drones, security, medical diagnoses, and treatments for the visually impaired.

Since the CNNs growing popularity in these tasks, researchers around the world started experimenting with these architectures also for problems other than CV such as:

- Text analysis
- Time series analysis
- ...

Convolutional Neural Networks

Why CNNs? – *Getting the general idea*

8

Biological roots

CNNs were invented first to address image recognition tasks, leveraging once more on a **biological example**: the **visual cortex**

As in the visual cortex different neurons respond differently and individually with respect to small regions of the visual field, the idea behind CNN is indeed to build an *architecture which is able to concentrate only to small regions of a picture*, one at a time, emulating the visual cortex **receptive field**

Common NN, why it is unfeasible

In a common NN framework, the net receives an input and it propagates the same layer by layer, where each neuron is fully connected with all the others. Finally the last fully connected layer, the output layer, predicts the scores for each class (always building on the image classification example)

In the case of a simple example as the CIFAR-10 dataset, we can already easily see some limitations of a fully connected approach ...

- CIFAR-10 dataset has **images of small sizes**, namely 32x32x3 (32 width, 32 height, 3 RGB channels)
- If we were to build a fully connected layer, each neuron in such a hidden layer will have $32 \times 32 \times 3 = 3072$ connections (i.e. weights)...
- Then, building a NN, we would most certainly have a layer with a least a bunch of these neurons; with 1 hidden layer of only 4 neuron we will **already have to fine-tune c.a. 1,2k parameters.. And we will have still an output layer to compute!**

A CNN approaches such a problem focusing only to small parts of the image at a time, fact that translates in neurons linked together only for fixed-size regions, **significantly shrinking the number of learnable parameters**

Convolutional Neural Networks

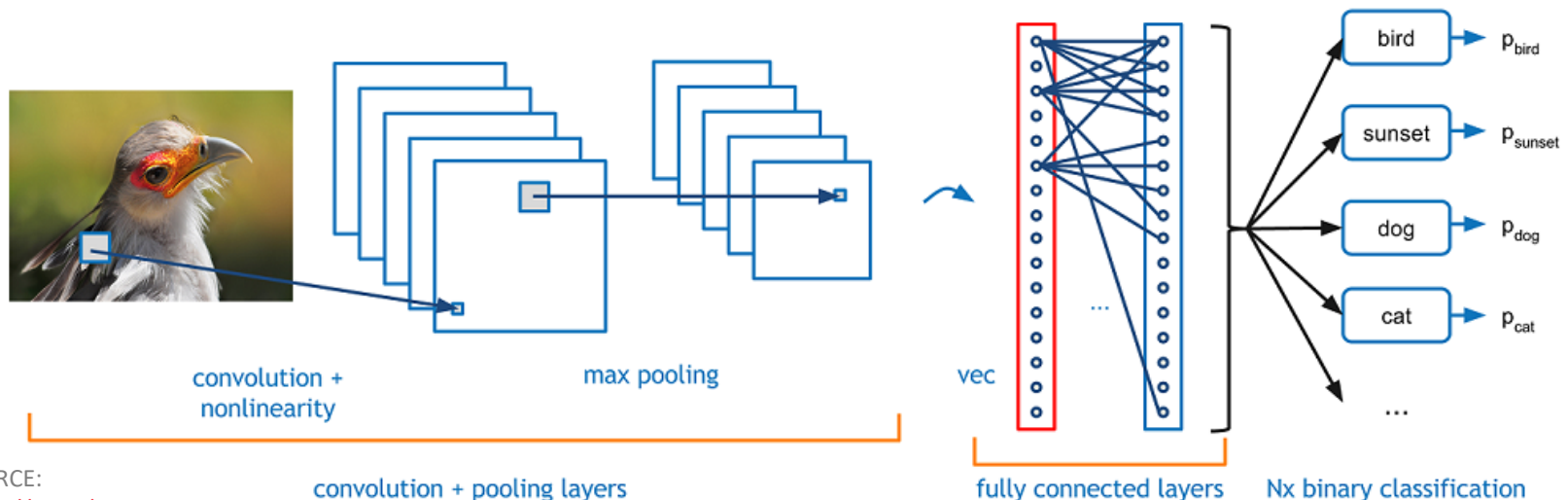
A dive into their architecture – Overview

9

As we just said, CNNs are a type of NN with their **specificity in the architectural design**. Let's see what it means ...

Generally, a CNN it is composed by **4 macro layers**:

1. **Input Layer**, which in the common case of image recognition is a 3d volume: width, height and the channels (R, G, B)
2. **Convolutional Layer**, which computes the output of neurons that are connected to local regions in the input
3. **Pooling Layer**, which performs a downsampling operation along the spatial dimensions (width, height)
4. **Fully Connected (FC) Layer**, which computes the prediction (i.e. in case of a image classification, it evaluates



SOURCE:
<https://bit.ly/2vwlegO>

Convolutional Neural Networks

A dive into their architecture – The Convolutional Layer

(1/6)

10



The **Convolutional Layer** is the architectural component aimed at **extract important and specific information from a particular region of the input data**

Such a layer **performs a convolutional** operation as it is **scanning the input** along its dimensions

Before going into more details, just recall what a convolution is:



A **convolution** between two functions f and g is a mathematical operation aimed at quantifying how much the two overlap

Mathematically, the convolution is defined as the integral of f and g , where the latter is reversed and shifted:

$$(f * g)(t) := \int_{-\infty}^{+\infty} f(\tau)g(t - \tau) d\tau$$

The concept of overlap is practically what the CNNs leverages on, **exploiting filters who in specific areas of the input look for similarities to learnt patterns**

This way a CNN learns, for example, how an eye looks like, without having to process all the pixels of the image all together in the computation of its detection

In the next slides, we will see better the specificity this architectural component

Convolutional Neural Networks

A dive into their architecture – The Convolutional Layer

(2/6)

11

Building once more on the image classification example, a ConvLayer **expects a volume input:**
(Width, Height, Channels)

As before anticipated, the layer approaches the analysis of such a volume thanks a set of **learnable filters**. Each is small spatially (along width and height), but extends through the full depth of the input volume.

This filters are the key actors in analyzing the input, building their behavior around the concept of **local connectivity**: *each neuron of the layer connects only to a local region of the input volume.*

Summarizing the concepts so far seen, it is important to keep in mind that:

- The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron; this is more commonly known as **filter size**
- The **extent of the connectivity along the depth axis is always equal to the depth of the input volume**

Such an **asymmetry** is crucial in how we treat the spatial dimensions (width and height) and the depth dimension: *the connections are local in space (along width and height), but always full along the entire depth of the input volume*

With these concepts, we have now all the info to see practically how the ConvLayer works



Convolutional Neural Networks

A dive into their architecture – *The Convolutional Layer*

(3/6)

12

Explained the concept of connectivity, we have now to introduce other 3 hyperparameters, other than the filter size, that have to be defined in order for the ConvLayer to work properly.

Here we summarize all the 4 hyperparameters used in a ConvLayer

Hyperparameter	Description	Tag
Filter size	<i>Receptive field of the CNN.</i> This translates on the number of inputs (along width and height) that the ConvLayer reads in for a single convolution	F
Depth	<i>Number of filters per ConvLayer.</i> This corresponds to the number of filters the CNN will have in the current ConvLayer, each looking for something different in the input	K
Stride	<i>Step taken by the filter.</i> This corresponds to the size of the step performed by the filter, moving along the input; i.e. a stride equal to 1 means that the filters moves by 1 pixel at a time, if instead the stride is set to 2, then filter jumps by 2 pixels	S
Zero-Padding	<i>0 added to the input.</i> Parameter that allows to control the spatial size of the output volumes and that helps to improve the convolution around the edges of the input	P

Convolutional Neural Networks

A dive into their architecture – The Convolutional Layer

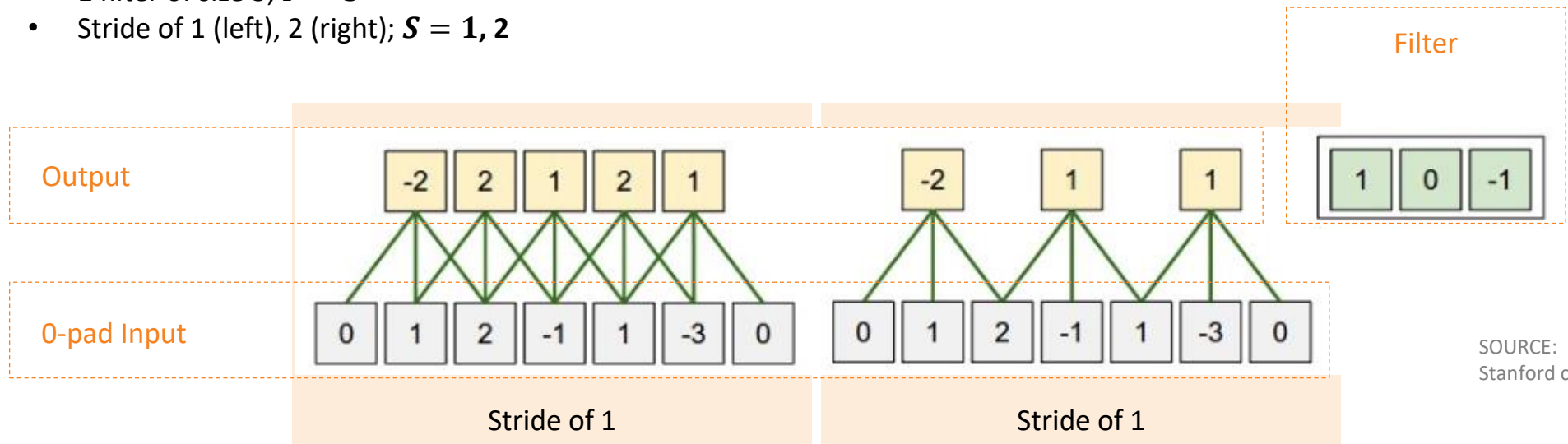
(4/6)

13

Defined these hyperparameters, the ConvLayer computes the output through the convolution, producing a result which has a size that can be expressed as a function of the input's

To explain it a little, let's see what happens with a simple 1d convolution:

- 5-dimensional input array; $W = 5$
- 1-dim 0 padding; $P = 1$
- 1 filter of size 3; $F = 3$
- Stride of 1 (left), 2 (right); $S = 1, 2$



SOURCE:
Stanford cs231

Such a setting, leads to an **output of size:**

$$\frac{W - F + 2P}{S} + 1$$

Convolutional Neural Networks

A dive into their architecture – The Convolutional Layer

(5/6)

14

In the general setting, a ConvLayer performs the following set of calculations, which leads to an output volume which depends, of course, on the input volume and the hyperparameters settings

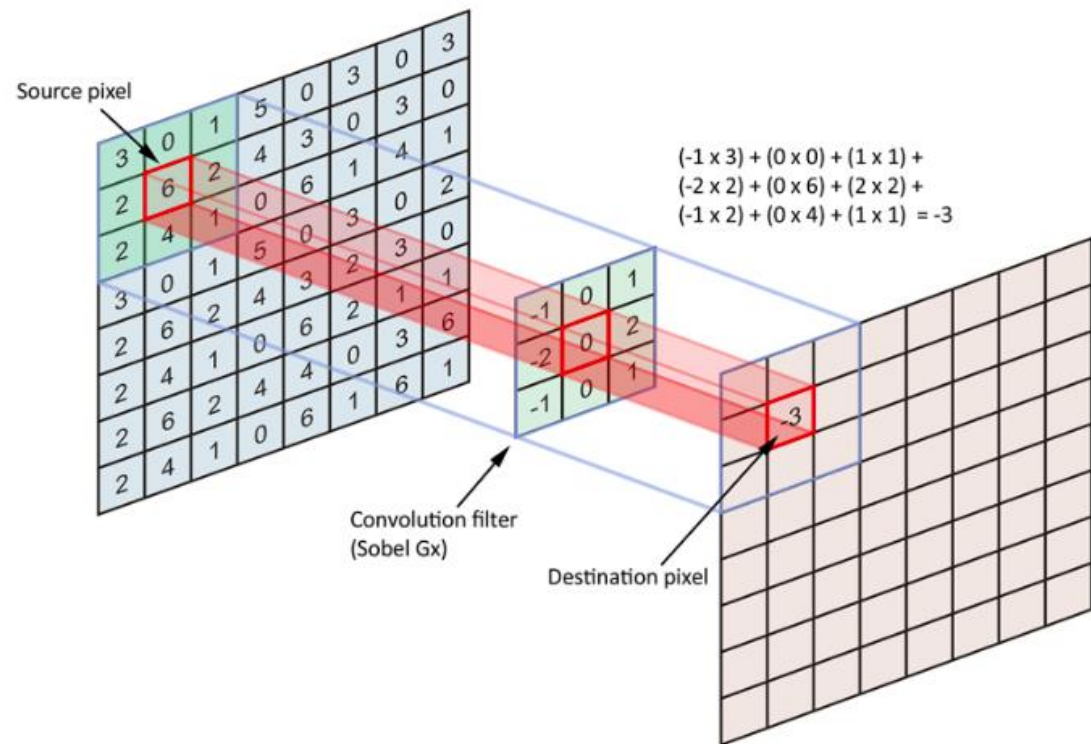
General steps:

1. Loading an input volume of size $W_1 \times H_1 \times D_1$
2. Convoluting every filter with the input, producing a volume of size $W_2 \times H_2 \times D_2$ where

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$
$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$
$$D_2 = K$$

To summarize and clarify a bit ...

The d -th depth slice (of size $W_2 \times H_2$) of the output volume is the result of a valid convolution of the d -th filter over the input volume, with a stride of S , after a 0-padding of P , and then offset by the d -th bias



Common settings for the hyperparameters are:

$$F = 3, S = 1, P = 1$$



Convolutional Neural Networks

A dive into their architecture – *The Convolutional Layer*

(6/6)

15

Animated convolution example, provided in the Stanford course cs231, in the CNNs section



Convolutional Neural Networks

A dive into their architecture – The Pooling Layer

(1/2)

16



The **Pooling Layer** is the architectural component aimed at **downsampling the inputs received**

Typically applied after a ConvLayer, it performs some a **dimensionality reduction** by mapping specific regions of the input into smaller ones, by means of **predefined criterion** (usually taking the max element – i.e. max pooling)

Here some examples of pooling operations *(credits to Stanford cs230 by Afshine and Shervine)*

	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	<ul style="list-style-type: none">- Preserves detected features- Most commonly used	<ul style="list-style-type: none">- Downsamples feature map- Used in LeNet

Convolutional Neural Networks

A dive into their architecture – The Pooling Layer

(2/2)

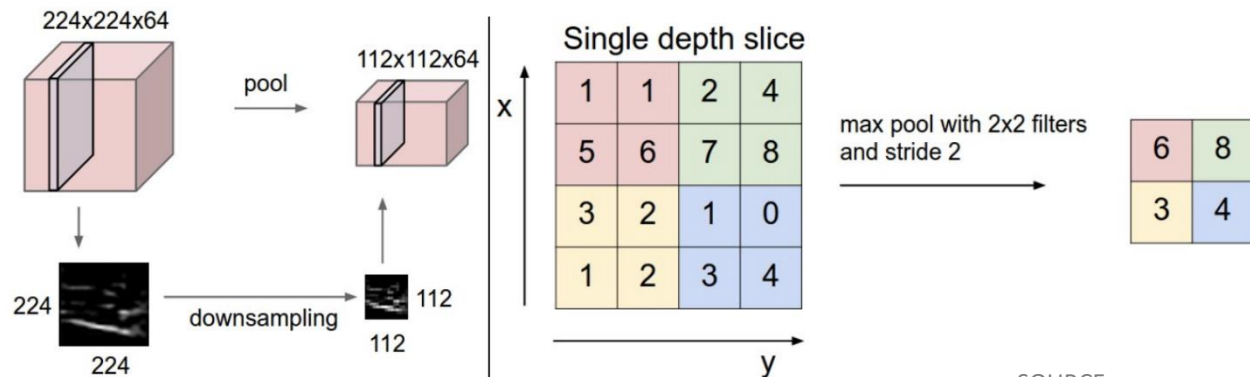
17

Pooling Layers, shrinking the input dimension, **reduce the amount of parameters** and **computation** in the network; doing so, it also **help controlling overfitting**.

The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using, for example, the MAX operation.

The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, **discarding 75% of the activations**.

Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.



SOURCE:
Stanford cs231

Generally speaking, a Pooling Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters
 - The **spatial extend**, F
 - The **stride**, S
- Produces an output volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = \frac{W_1 - F}{S} + 1$
 - $H_2 = \frac{H_1 - F}{S} + 1$
 - $D_2 = D_1$

Convolutional Neural Networks

A dive into their architecture – Fully Connected Layer

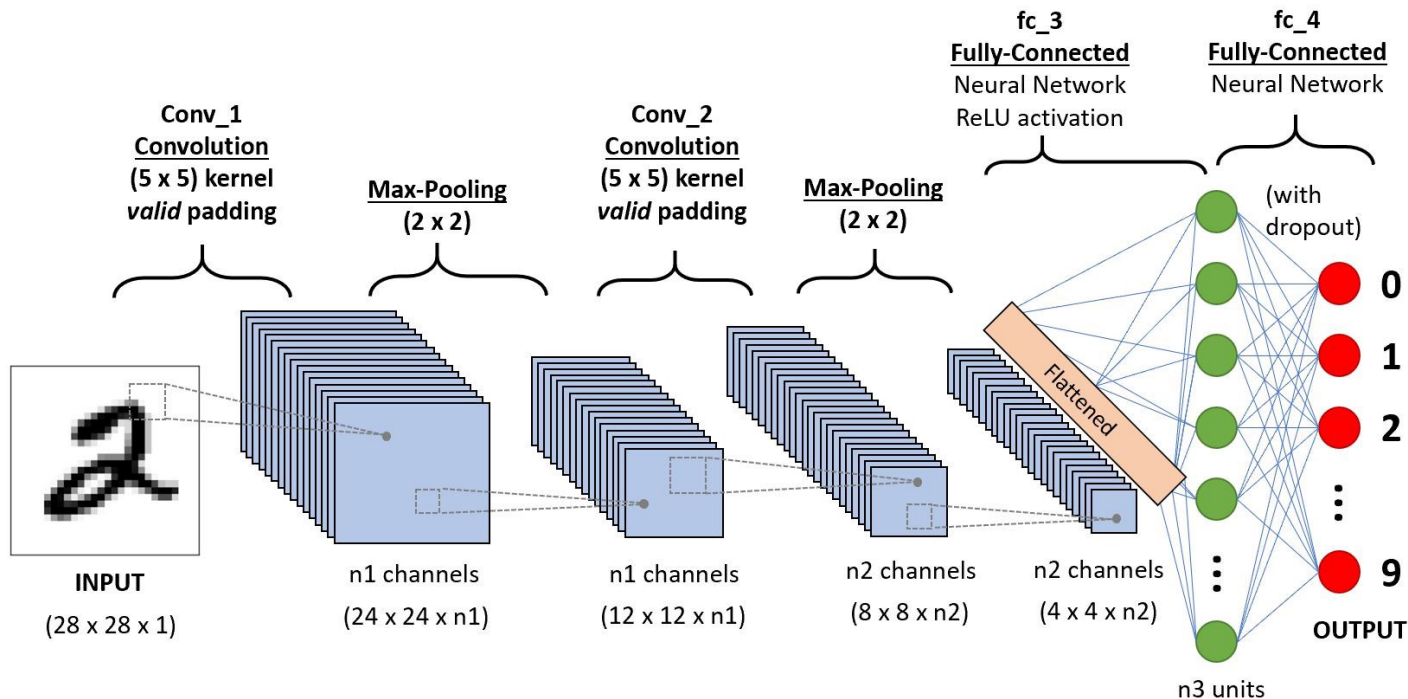
18

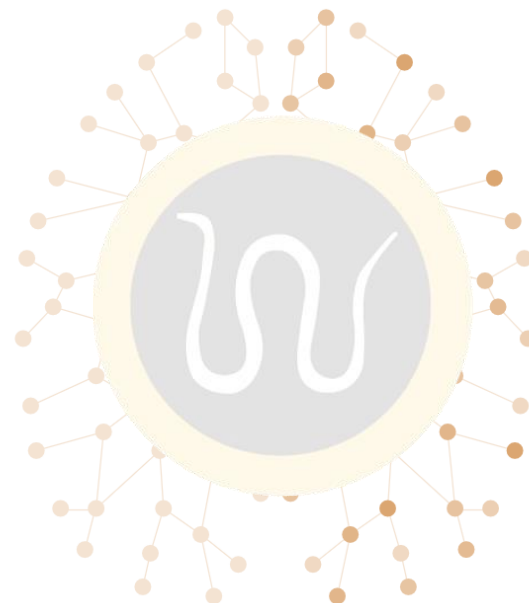


The **Fully Connected Layer** is the architectural component aimed at **predicting the output** of the CNN

The FC operates on flattened input, where **each input is connected to the every neuron in the layer**, just like in the settings seen in the first lesson

Fully Connected layers are usually present at the very end of a CNN architecture, as output layer, to produce the distribution of probability with respect to the classes provided





01	TensorFlow, a brief introduction	3
02	Convolutional Neural Networks	6
03	Recurrent Neural Networks	19



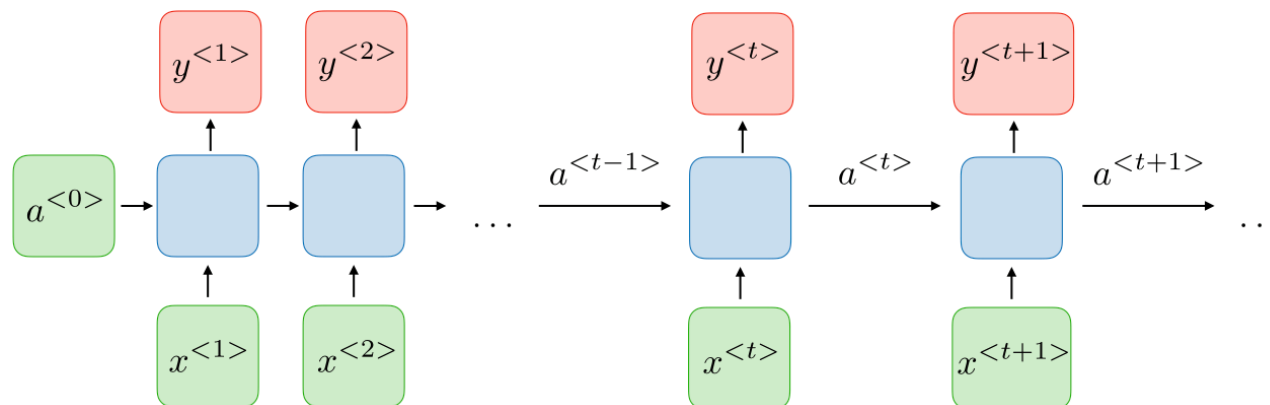
Recurrent Neural Networks (RNNs) are a class of deep neural networks which **allow previous outputs to be used as inputs** while having hidden states.



RNNs have been introduced with the **specific purpose to model, handle and predict sequences**

To do so, RNNs leverage on particular building blocks, the so-called **cells**, which process the input signal (*a sequence itself*) in a chain of operations that can be briefly summarized as follows:

1. Every component of the input is passed to a specific cell of the RNN
2. Each cell process the current input, jointly with the output of the preceding and, in the case of the bidirectional-RNN, following cell
3. If the RNN is composed of many layers, the outputs of the current layer are passed to the next, and the activations are processed as in steps #1 and #2

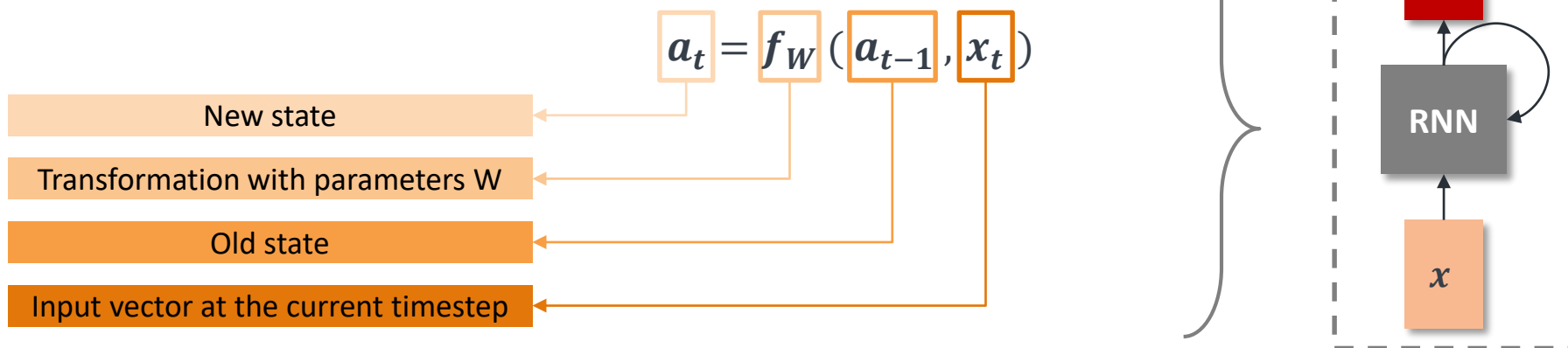


Recurrent Neural Networks

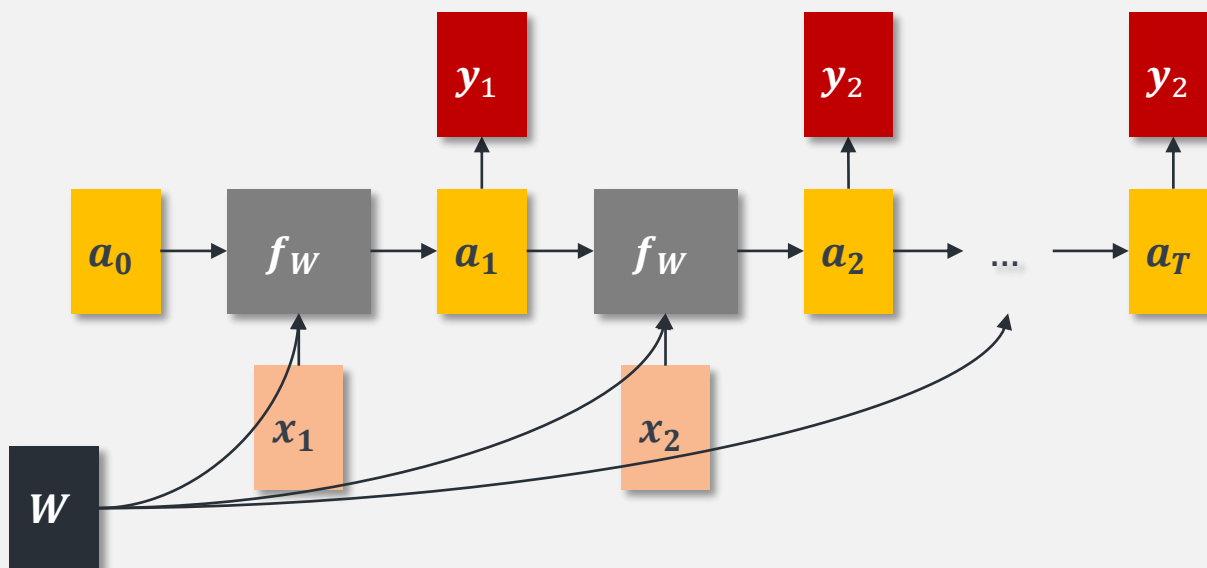
Introduction – *The main idea*

21

Given the previous high-level premise, a RNN aims at processing sequences x in a recurrent fashion, at every time step t :



The same transformations (i.e. f_W) and all the learnable parameters (W) are shared across the RNN ...

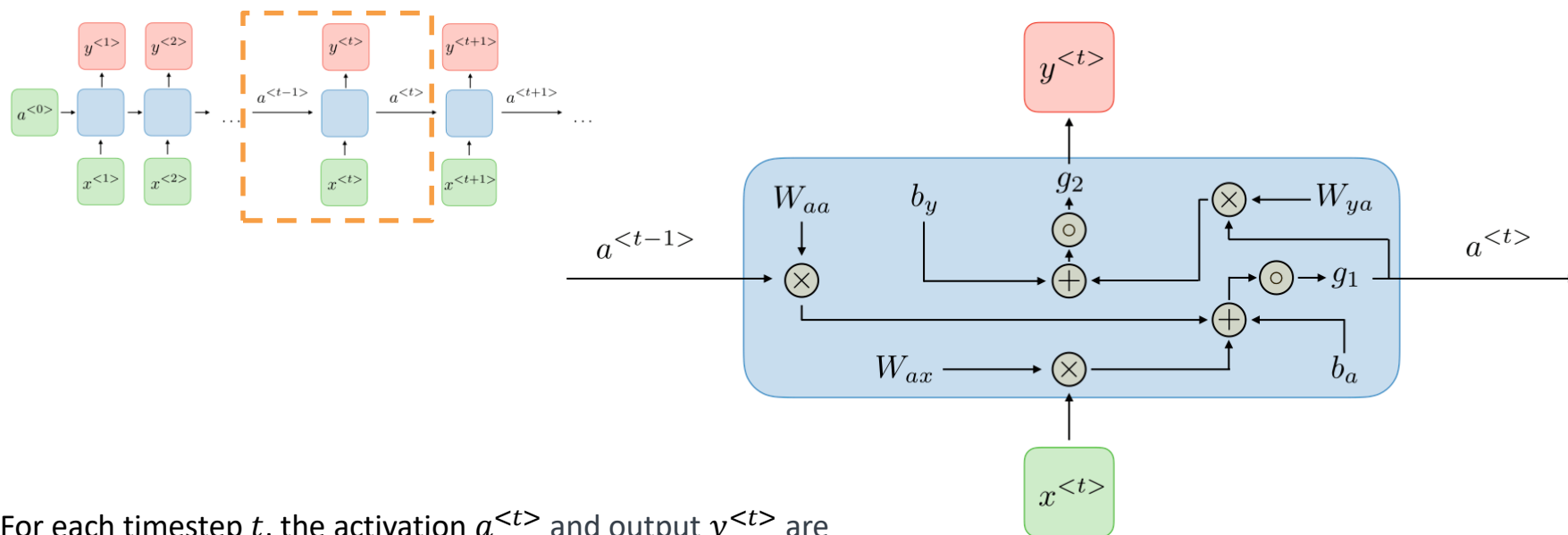


Recurrent Neural Networks

A dive into the cell architecture

22

Let's now dive into the specific operations which occur in a RNN (*credits to Shervine Amidi and Afshine Amidi*)



For each timestep t , the activation $a^{<t>}$ and output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

Where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 are activation functions

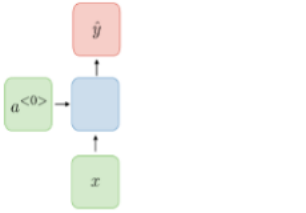
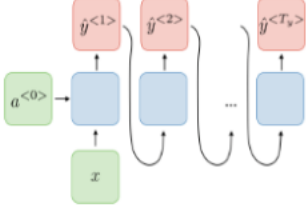
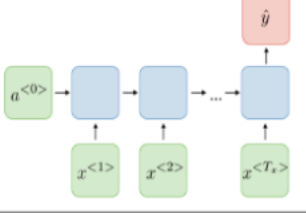
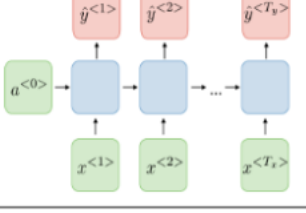
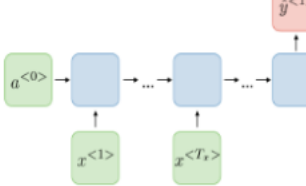
Recurrent Neural Networks

Examples of high level architectures

RNNs can be modelled for many different tasks.

Among the most interesting and exploited we can mention:

- *One-to-many*, as the case in which, giving the first note, the algorithm is asked to generate music
- *Many-to-one*, as for
 1. Text classification problems, which **sentiment analysis** is a classic example quite common in the financial industry
 2. **Forecasting** (i.e. regression problem) of the next **future price of a stock**, given a n -sized window of previous data
- *Many-to-many*, as for **text translation** problems, where a sentence of n words of the language A has to be translated in a sentence of m words in the language B

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

Given the generic structure of a RNN, depending on the specificity of the components (i.e. **gates**) of the cells and based on the latter do interconnect with each other, different kind of RNNs can be created

Among the most common types of RNNs, we can mention:

1. **Long Short Term Memory RNNs (LSTMs)**
2. **Gated Recurrent Unit RNNs (GRUs)**
3. **Bidirectional RNNs**

Before introducing the former, an important concept has to be discussed: the **Vanishing/Exploding Gradient** issue.

The **Vanishing/Exploding Gradient problem** is the issue, commonly encountered with RNNs, that consists (respectively) in:



- The **gradients decreasing up to almost, if not, 0**. This makes **impossible for the NN to keep on learning**, since every update of the learnable parameters will be null (recall the $\theta_i = \theta_i - \lambda \cdot \partial C / \partial \theta_i$ relation, where θ_i is any learnable parameter of the net)
- The **gradients massively increasing**. This makes **impossible for the NN to keep on learning**, since the updates to the net parameters will be way too big for the learning algorithm to steadily and correctly move towards the minimum loss.

Recurrent Neural Networks

The Vanishing/Exploding Gradient problem

(2/2)

25

As just said, this issue is commonly encountered in RNNs architectures mainly due to the fact that, even when building a “small” RNN, **we are actually dealing with a deep-architecture** ...

Every cell of a RNN, the alter ego of the nodes of a common NN, is actually a NN itself, with many different gates

Capturing long term dependencies means indeed **compute gradients for many cells and gates**, which **cumulates in a multiplicative way*** (BBTT). This multiplicative compounding leads to an easy vanishing or exploding value

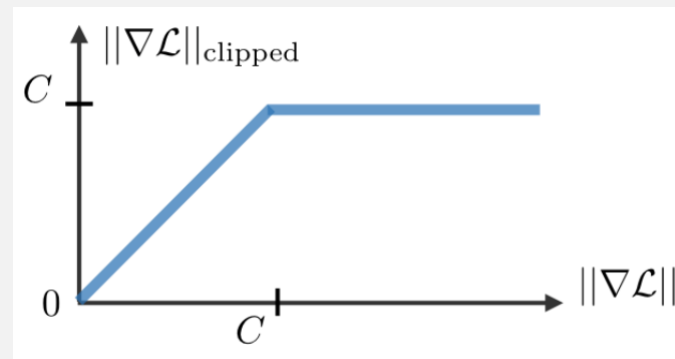
Such a problem is the main reason why training a RNN is not an easy task: it requires an accurate understanding of chosen RNN architecture and a deep control over the training process

The research community has proposed practical solutions in order to deal with this issue; an example of trick to **prevent the exploding gradient** is, for example, the following:



The **Gradient Clipping** is a technique used to cope with the exploding gradient problem, sometimes encountered when performing backpropagation (actually *Back Propagation Through Time*).

It consists in **capping the maximum value for the gradient** to a predefined value. This, in practice, has revealed to control the aforementioned issue, allowing the RNN to learn.



Recurrent Neural Networks

Long Short Term Memory (LSTMs)

(1/2)

26

The **Long Short Term Memory** (**LSTM**) is a RNN which formulation is aimed at avoiding the Vanishing/Exploding Gradient issue

A LSTM follows the basic functioning of a Vanilla RNN, but it differentiates by how the cells are built. A cell of a LSTM is composed by **4 gates**

Input or Update gate (i_t)

Gate which process the new input at the current time step

Gate or Relevance gate (g_t)

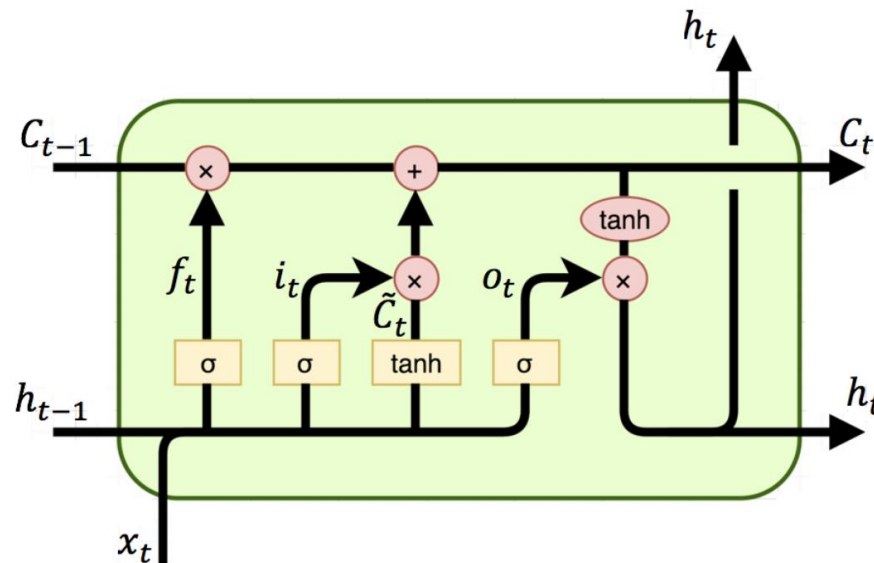
Gate that decides how much new information has to be used

Forget gate (f_t)

Gate that decides whether an information should be used or not

Output gate (o_t)

Gate that decides how much of the new info produced should be passed by



Recurrent Neural Networks

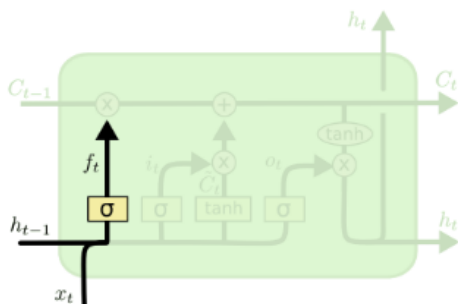
Long Short Term Memory (LSTMs)

(2/2)

27

Forget gate

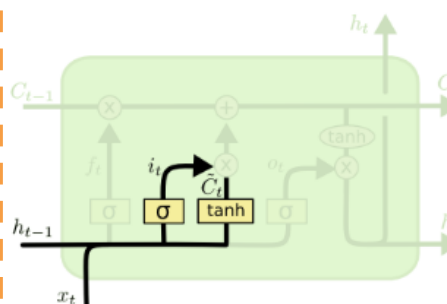
1



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Input and Gate gate

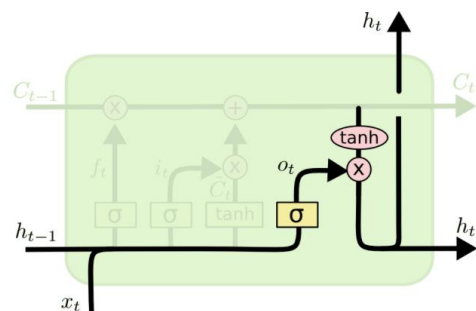
2



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Output gate

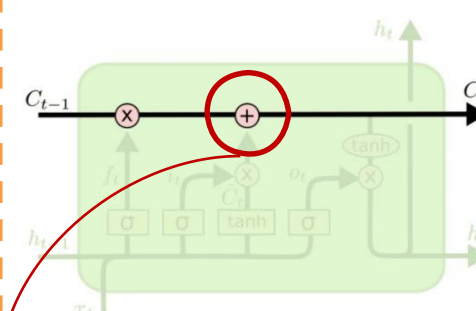
3



$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Computation of the new Cell State

4



$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$



This operation is **key** in order for the LSTM to not interrupt the so-called “**Gradient Flow**”
With such an additive relation, the **gradient does not vanishes**

Recurrent Neural Networks

Gated Recurrent Unit (GRUs)

28

The **Gated Recurrent Unit (GRU)** is a RNN which formulation is aimed at avoiding the Vanishing/Exploding Gradient issue, but in a simpler way with respect to LSTMs

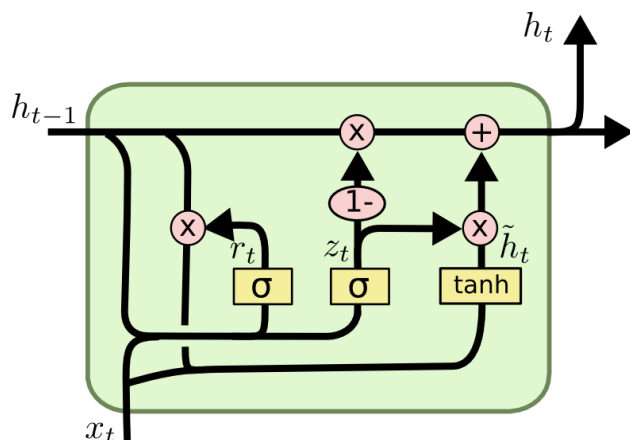
Differently from a LSTM, a GRU has cells composed by only **2 gates**

Input or Update gate (z_t)

Gate which process the new input at the current time step

Gate or Reset gate (r_t)

Gate that decides how much new information has to be used



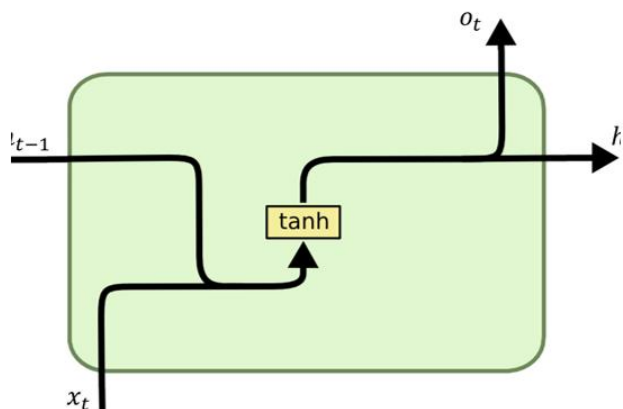
$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

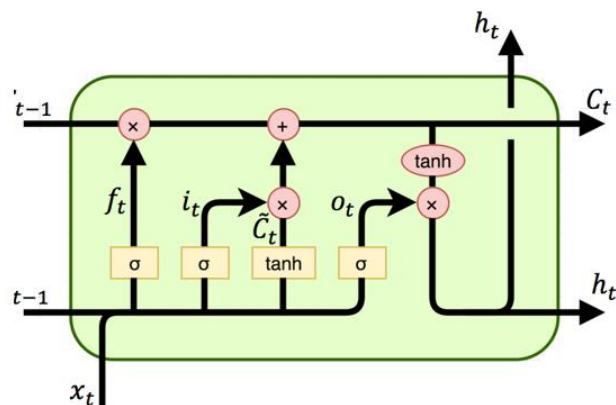
$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

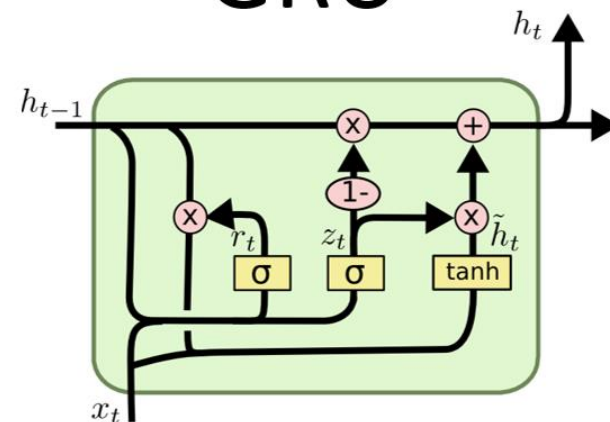
RNN



LSTM



GRU



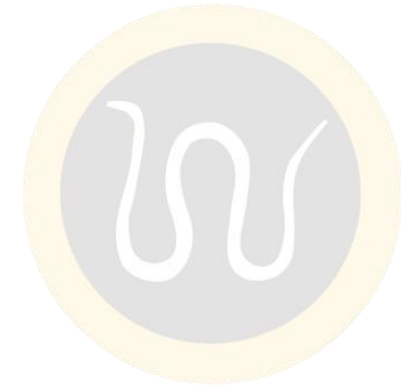
Contacts

Antonio Menegon

Manager and FinTech Stream Leader

Mobile: +39 366 9534672

E-mail: antonio.menegon@iasonltd.com



Iason is an international firm that consults Financial Institutions on Risk Management.

Iason integrates deep industry knowledge with specialised expertise in Market, Liquidity, Funding, Credit and Counterparty Risk, in Organisational Set-Up and in Strategic Planning.

To get in touch with us, please send an email to: info@iasonltd.com

This is a Iason's creation.

The ideas and the model frameworks described in this presentation are the fruit of the intellectual efforts and of the skills of the people working in Iason. You may not reproduce or transmit any part of this document in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of **Iason Consulting Ltd.**

www.iasonltd.com