
Machine
Learning
for
Astrophysics

Reinforcement Learning and Deep Learning, The Revenge

Outline

Introduction to ML & Supervised ML

- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

Unsupervised ML

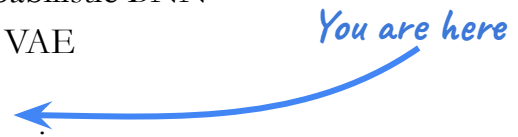
- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Anomaly Detection
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Self-Organizing Maps

Deep Learning

- Basics of NN: computation graphs
- Training a NN: *forth*- and *back*-propagation
- Optimization Algorithms
- Transfer Learning
- ResNets
- Bayesian NN, Probabilistic BNN
- Autoencoders and VAE

Deep Learning, The Revenge

- Reinforcement Learning
- Convolutional Neural Networks
- LeNet-5, AlexNet
- Data Augmentation, Semantic Segmentation
- Generative Adversarial Networks
- (hints on) Recurrent Neural Networks



The Holy Trinity of Machine Learning

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

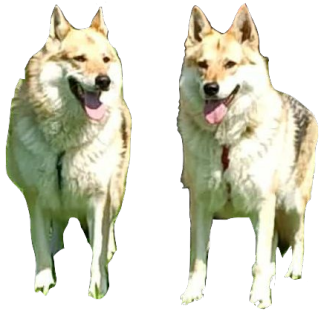
Goal:
learn a function that maps
$x \rightarrow y$

"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:
learn underlying structure of x



"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

Reinforcement Learning

Data: *state-action* pairs

Goal:

learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

The *agent* is the main actor in the reinforcement learning paradigm, and lives in an environment.

A *state* is a representation of the current environment that the *agent* is in.
The *agent* observes the *state*, and takes a decision on which *state* to move next.

A *state* can either be discrete (e.g., a chess board) or continuous (e.g., SuperMario)

An *action* is the mechanism by which the *agent* transitions between *states* of the environment

Actions are usually discrete, and you should imagine them as controlling the joystick of a console → △○×□

The *policy* π is the *agent's* probabilistic mapping from *states* to *action*.
The chosen *policy* is trained as to maximize future *rewards*.

A *reward* is a score given to certain states.
Positive scores are assigned to *states* in which the *agent* should go in the future
Negative scores are assigned to *states* in which the *agent* should **NOT** go in the future.

Reinforcement Learning

Data: *state-action* pairs

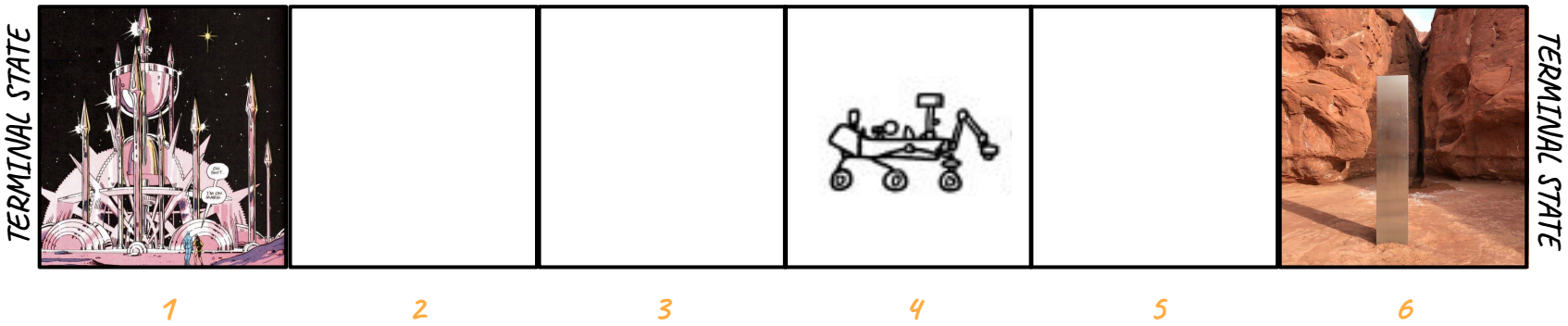
Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

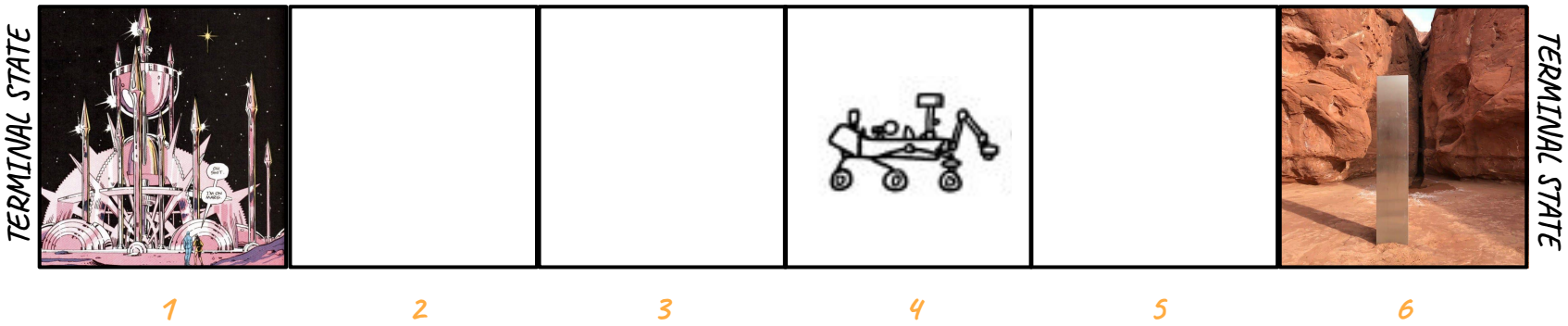
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.

completely taken from Andrew Ng Machine Learning Course, thanks Andrew!

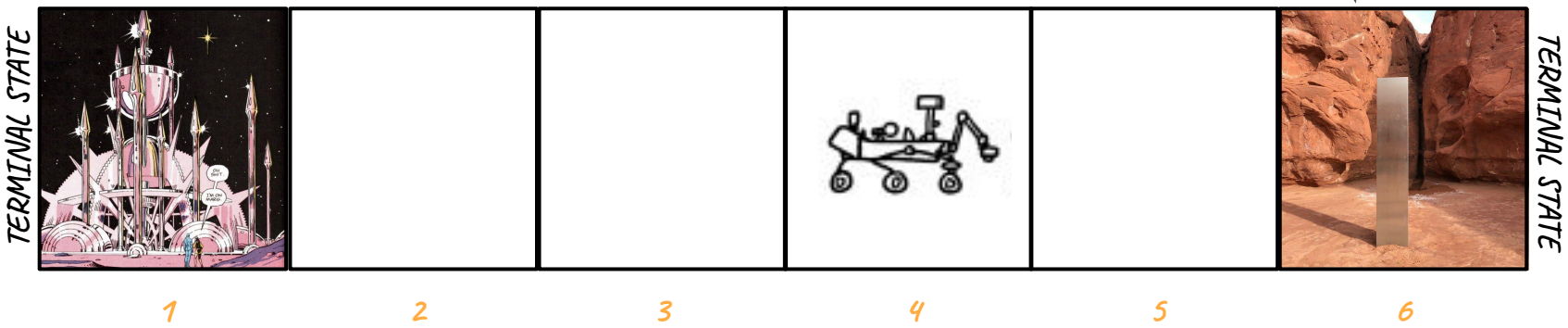
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.



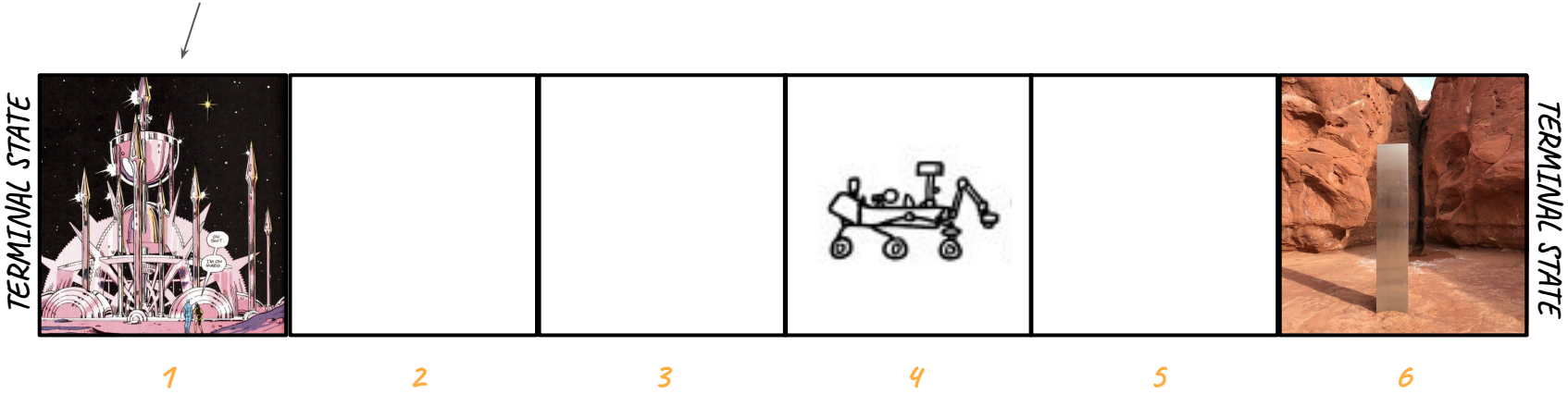
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.



To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space Odyssey, so it is not *that* interesting.

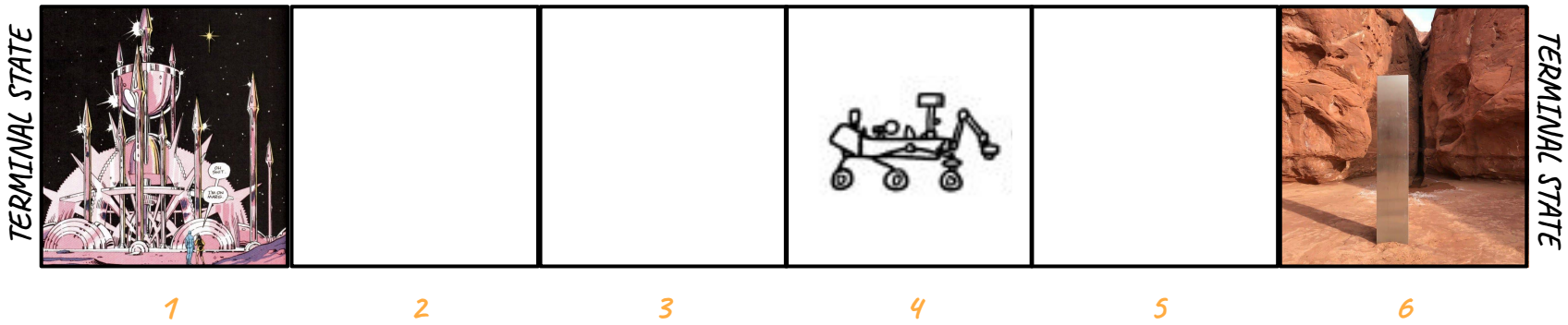


To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.

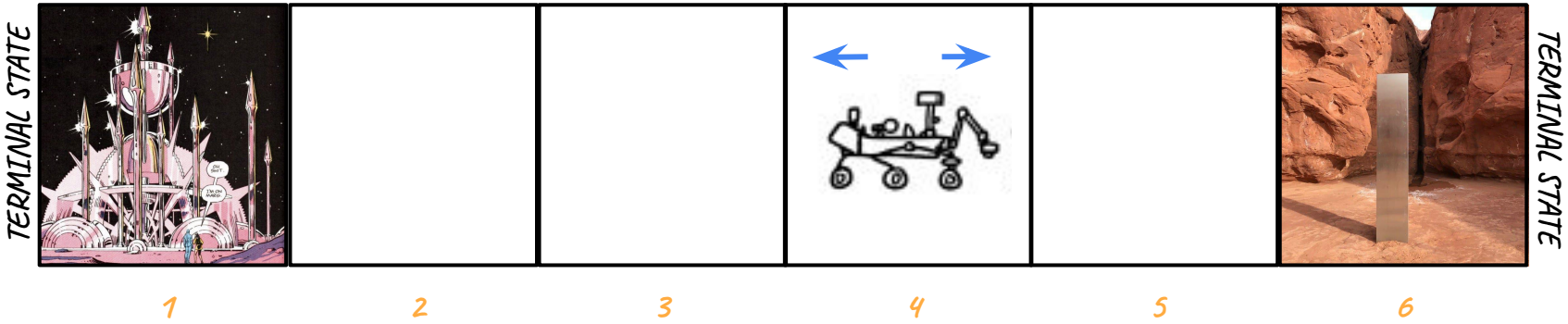


Mars Rover

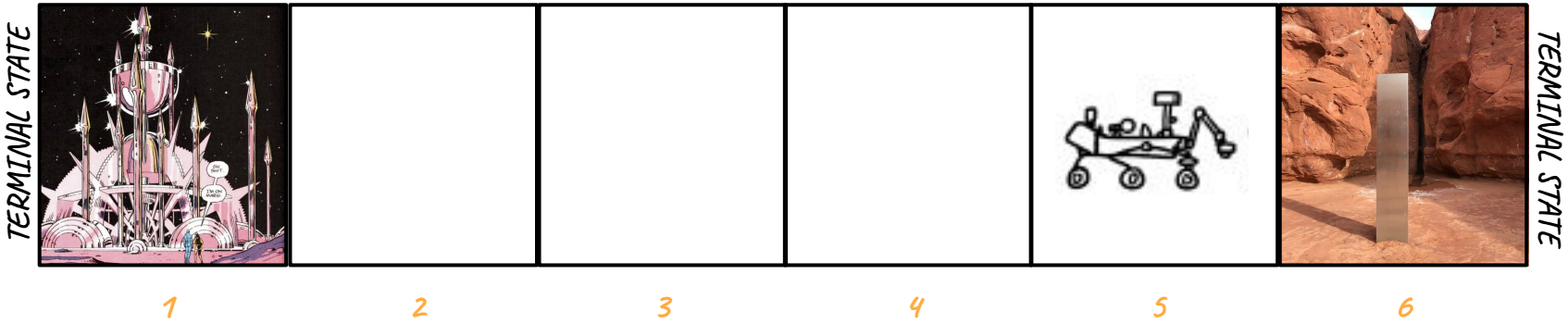
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.

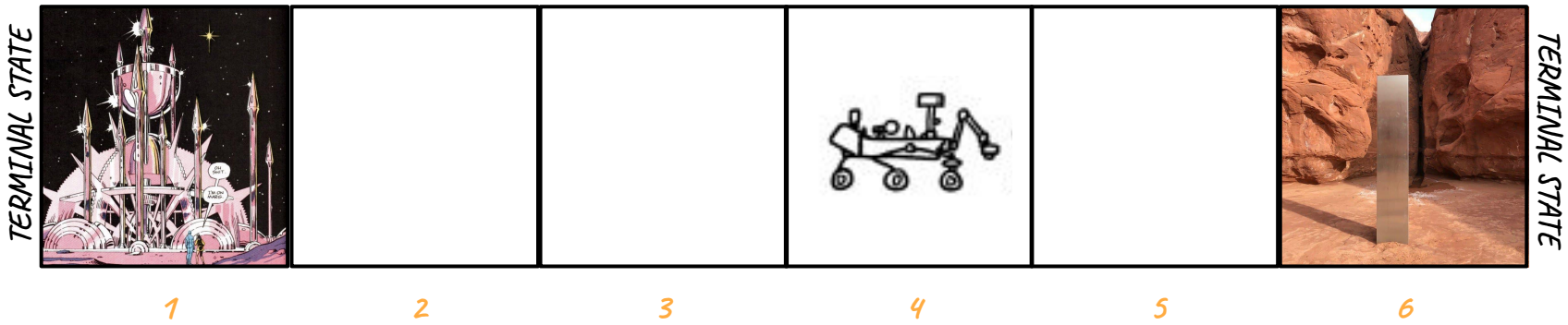


To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.

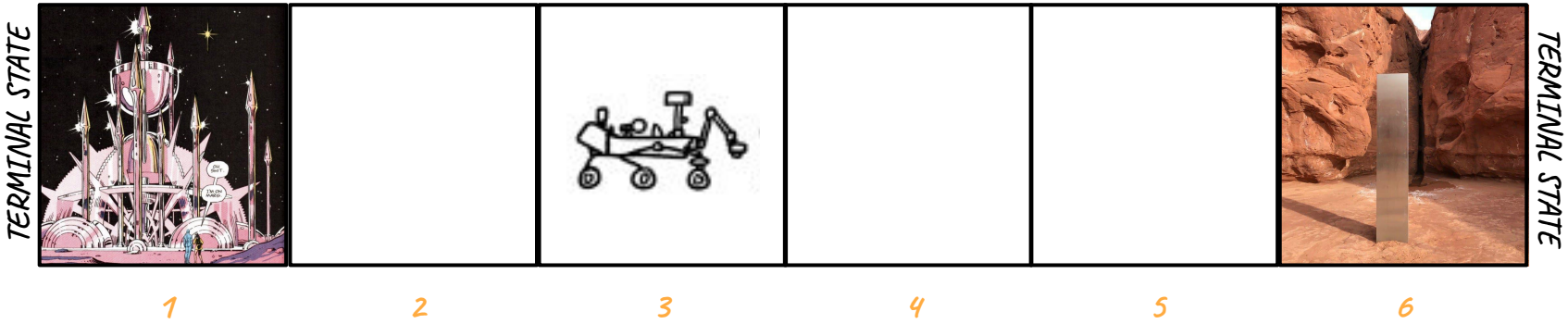


Mars Rover

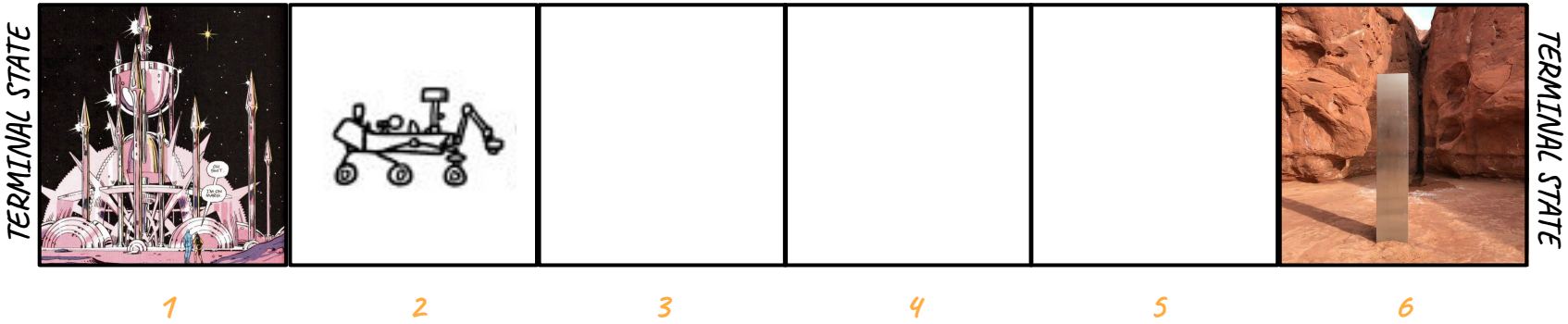
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



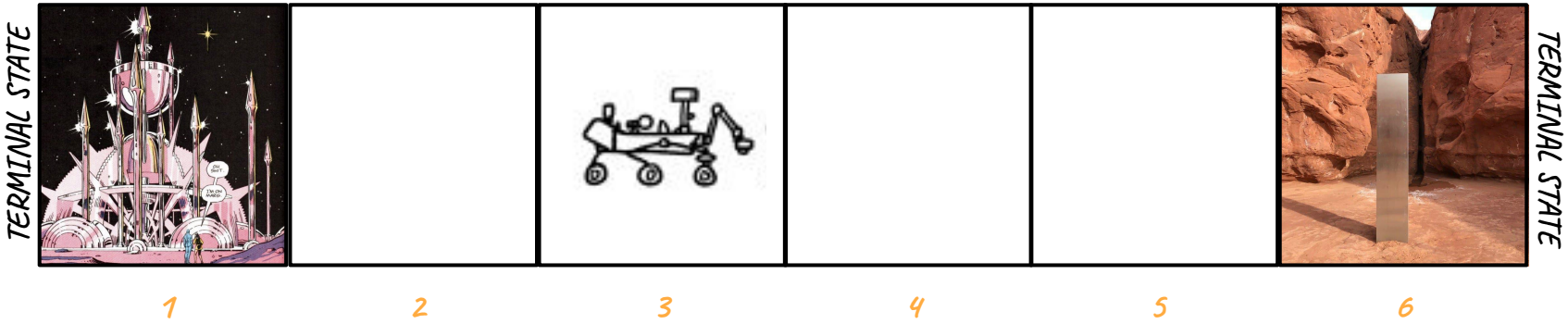
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.

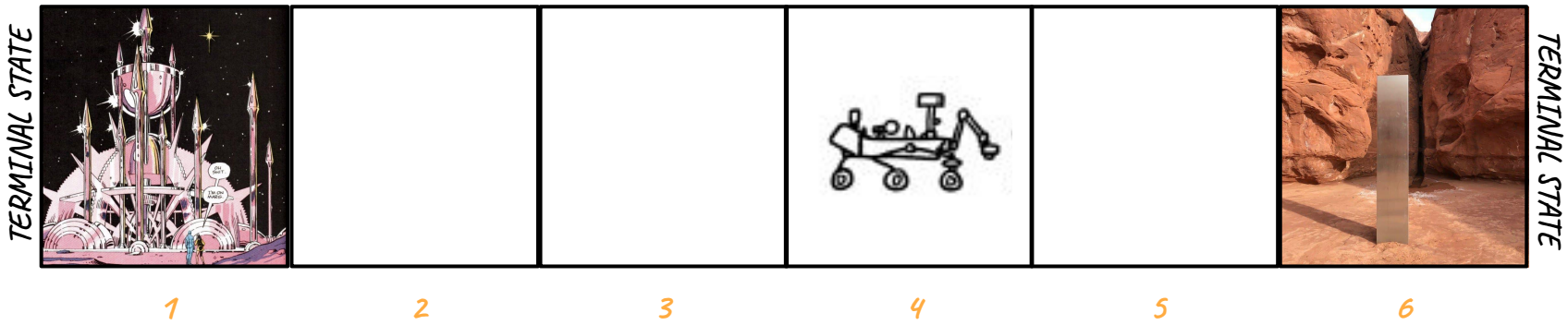


To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.

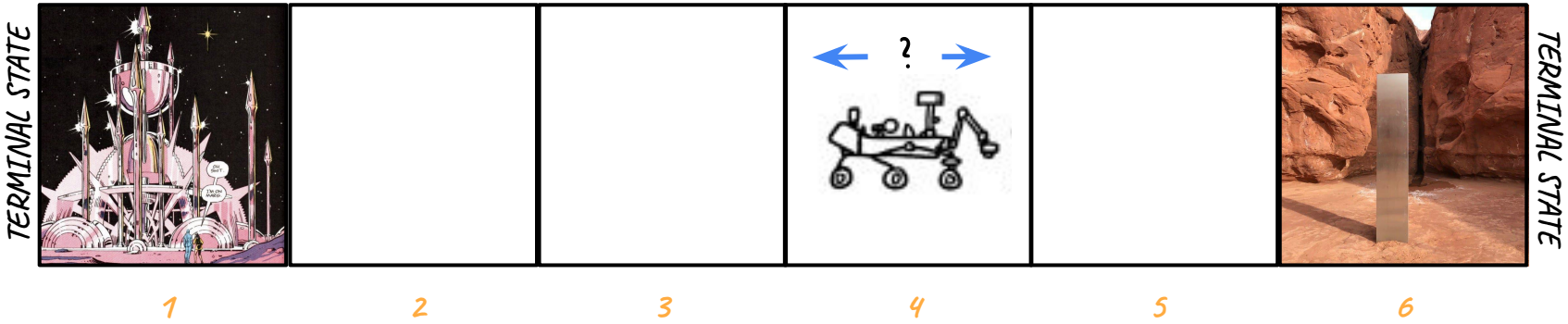


Mars Rover

To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.

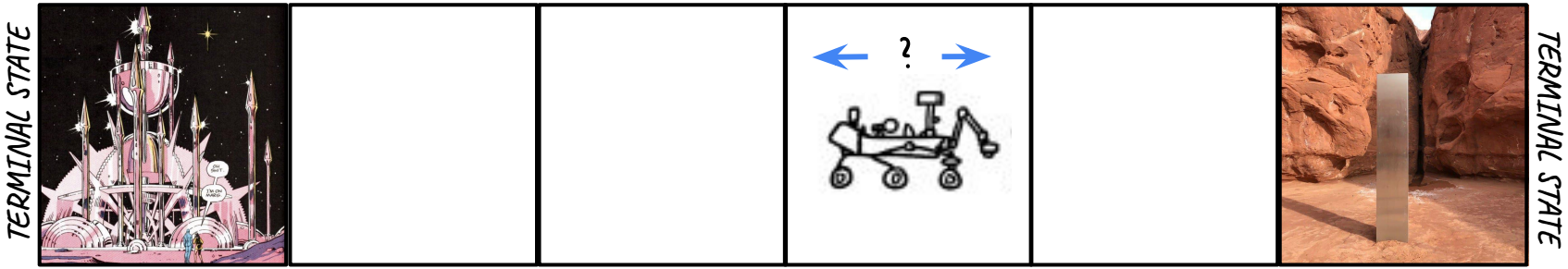


To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



Well, in order to decide what to do next, Curiosity should know that is the *reward* that each state brings with itself.

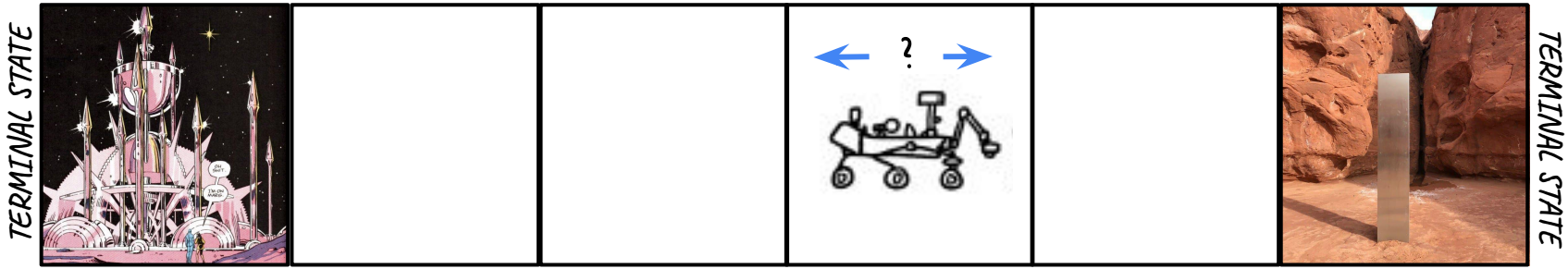
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

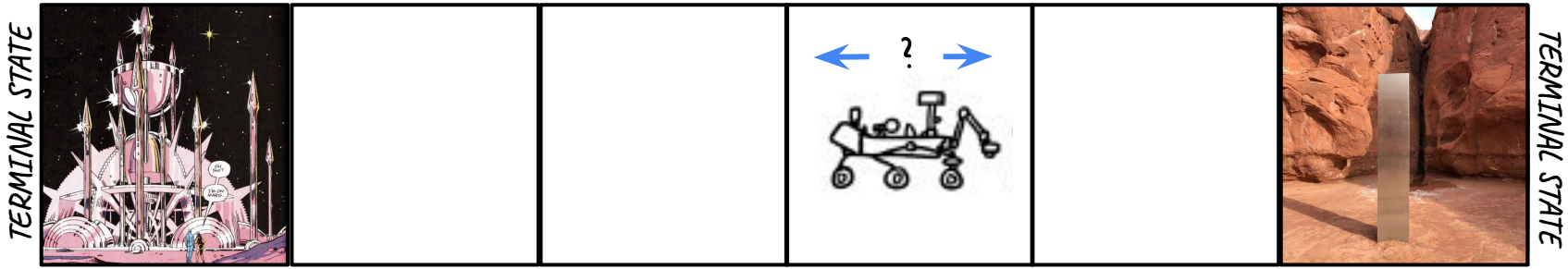
Well, in order to decide what to do next, Curiosity should know that is the *reward* that each state brings with itself.

To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space Odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from Watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



Well, in order to decide what to do next, Curiosity should know that is the ***reward*** that each state brings with itself.
If it decides to go left ←, then it will get rewards *0* (*s4*), *0* (*s3*), *0* (*s2*), and *100* (*s1*), and then arrives at a ***terminal state***, the day is over.
If it decides to go right →, then it will get rewards *0* (*s4*), *0* (*s5*) and *40* (*s6*), and then arrives at a ***terminal state***, the day is over.

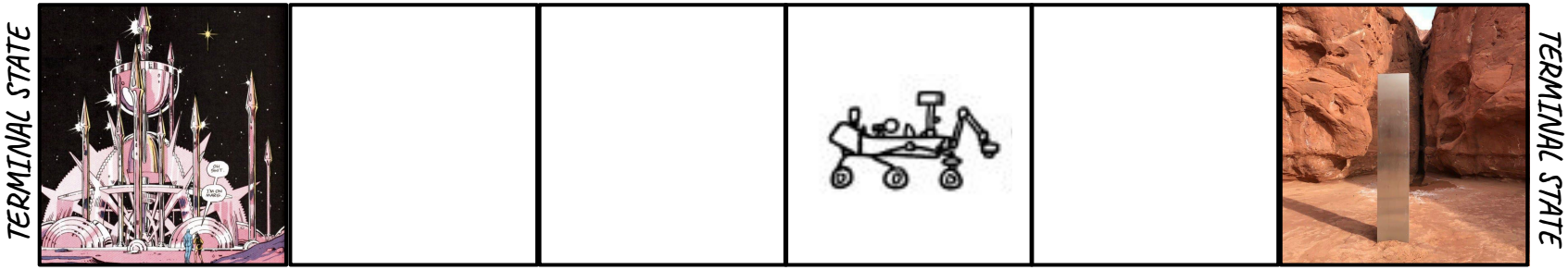
To understand Reinforcement Learning, let's take a simple six *discrete-states* example.
Here the *agent* is the Curiosity rover, in *state 4*.
In *state 6* we have an interesting area to map, but is just a monolite, we all watched 2001: A Space Odyssey, so it is not *that* interesting.
On the contrary, in *state 1* we have Dr. Manhattan's Mars glass castle, from Watchmen, which is extremely more interesting to study.
How can we make the Curiosity rover choose where to go autonomously? Here we have only two *actions*: go left and go right.



Well, in order to decide what to do next, Curiosity should know that is the *reward* that each state brings with itself.
If it decides to go left ←, then it will get rewards 0 (s4), 0 (s3), 0 (s2), and 100 (s1), and then arrives at a ***terminal state***, the day is over.
If it decides to go right →, then it will get rewards 0 (s4), 0 (s5) and 40 (s6), and then arrives at a ***terminal state***, the day is over.
At any time, the *agent* is in a *state s*, chooses some *actions a*, and enjoys some *reward R(s)*, and as a result of its action arrives a new *state s'*.

core elements of Reinforcement Learning

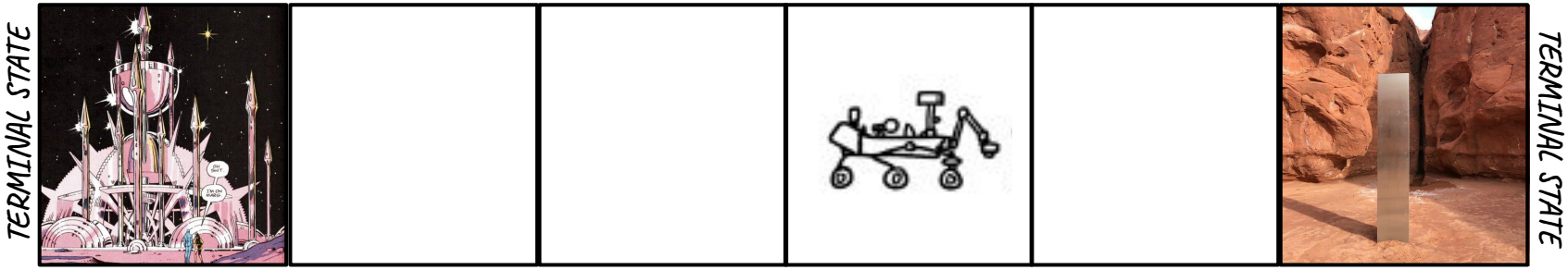
$(s, a, R(s), s') \xrightarrow{\text{e.g.}} (4, \leftarrow, 0, 3)$



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

Actually, the rover does not choose its next **action** based on **rewards** only, but on return, under the principle that **rewards** that you can get quickier are perhaps better than the ones you'll get later.

$$\text{Return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots \text{ until terminal state}$$ with γ being the **discount factor (e.g. 0.9)**



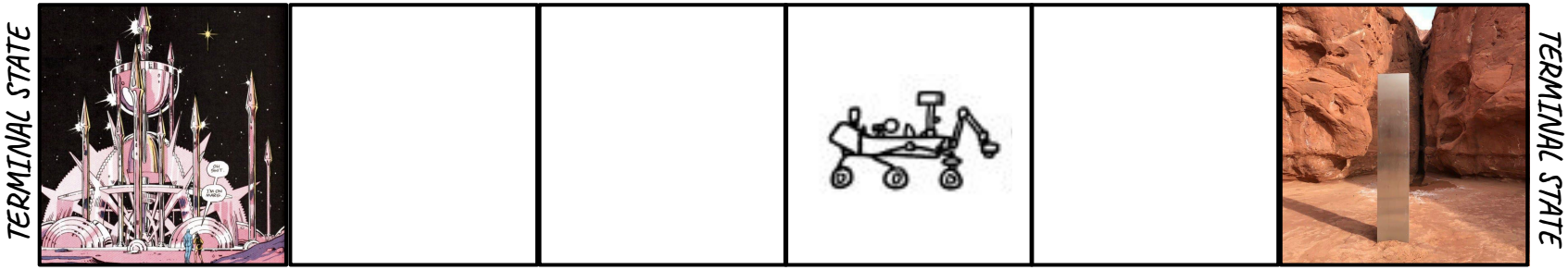
	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

Actually, the rover does not choose its next **action** based on **rewards** only, but on return, under the principle that **rewards** that you can get quickier are perhaps better than the ones you'll get later.

$$\text{Return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots \text{ until terminal state}$$ with γ being the **discount factor (e.g. 0.9)**

The return depends on the **action** the **agent** takes.
So, with a discount factor of 0.9:

- going full left ← will give a **return** of 72.9
- going full right → will give a **return** of 32.4



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

Actually, the rover does not choose its next **action** based on **rewards** only, but on return, under the principle that **rewards** that you can get quickier are perhaps better than the ones you'll get later.

$$\text{Return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots \text{ until terminal state}$$ with γ being the **discount factor (e.g. 0.9)**

The return depends on the **action** the **agent** takes.

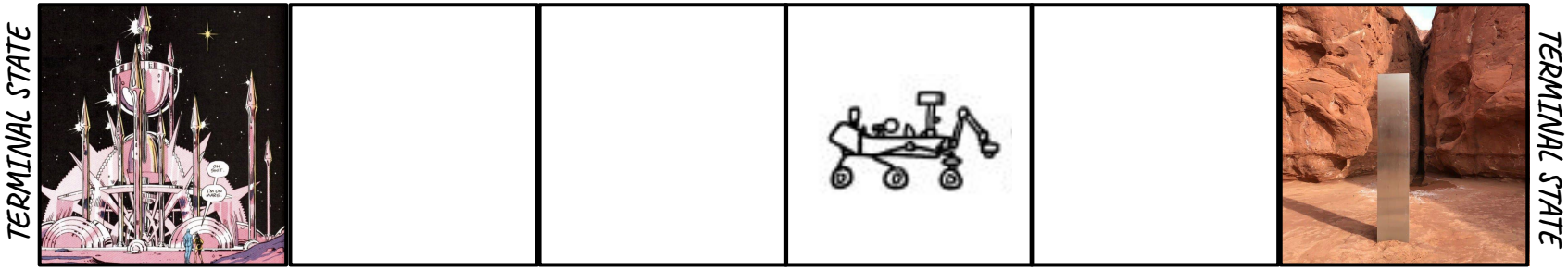
So, with a discount factor of 0.9:

- going full left ← will give a **return** of 72.9
- going full right → will give a **return** of 32.4

… and with a discount factor of 0.4:

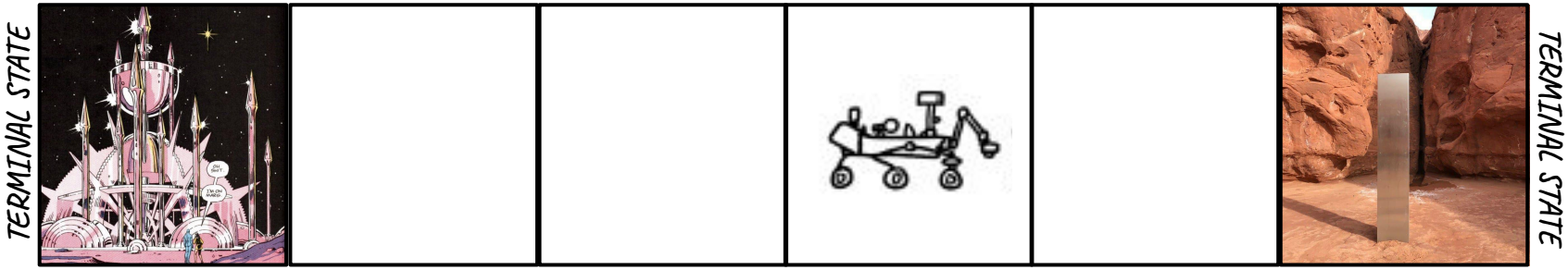
- going full left ← will give a **return** of 2.7
- going full right → will give a **return** of 3.6

So, the higher the **discount factor**, the more patient the **agent** will be



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

In Reinforcement Learning, the goal is to find a function called **policy π**, whose job is to take as input any **state s** and map it to some **action a** that we want the **agent** to take: **π(s) = a**.
We want to find the **policy π** that tells what **action** to take in every **state** so as to maximize the return.

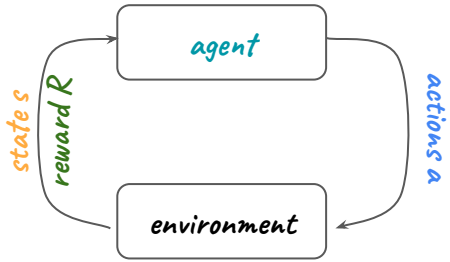


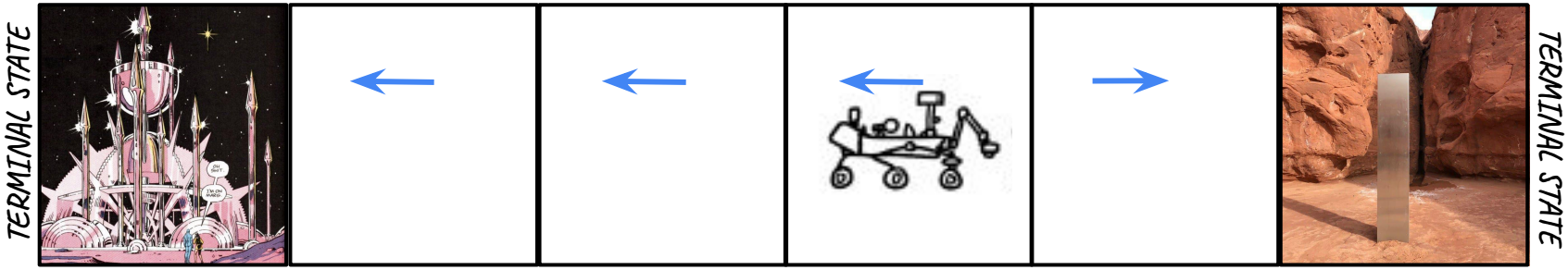
	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

In Reinforcement Learning, the goal is to find a function called **policy π**, whose job is to take as input any **state s** and map it to some **action a** that we want the **agent** to take: **π(s) = a**.
We want to find the **policy π** that tells what **action** to take in every **state** so as to maximize the return.

In this case, a **policy** might be go always left ←, or go always right →, or go always towards the maximum **reward**, or always to the maximum **reward** *unless* there is another one just one step away, there is an Universe of possible **policies** that applies to this example.

This is evaluated with a Markov Decision Process, meaning that the future depends only on where the **agent** is at the present, and not on past **actions**.



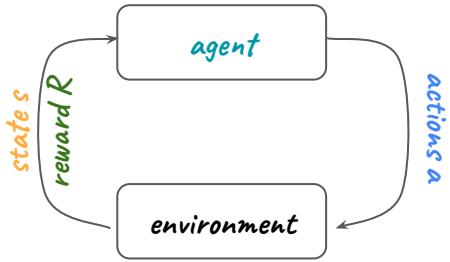


	1	2	3	4	5	6
Reward:	100	0	0	0	0	40

In Reinforcement Learning, the goal is to find a function called **policy π**, whose job is to take as input any **state s** and map it to some **action a** that we want the **agent** to take: **π(s) = a**.
We want to find the **policy π** that tells what **action** to take in every **state** so as to maximize the return.

In this case, a **policy** might be go always left ←, or go always right →, or go always towards the maximum **reward**, or always to the maximum **reward** *unless* there is another one just one step away, there is an Universe of possible **policies** that applies to this example.

This is evaluated with a Markov Decision Process, meaning that the future depends only on where the **agent** is at the present, and not on past **actions**.



State-Action value function

A key concept is the one of *state-action* value function, or the *Q-function*.
Q(s,a) captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*

State-Action value function

A key concept is the one of *state-action* value function, or the ***Q-function***.
Q(s,a) captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
Q(s,a) is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

State-Action value function

A key concept is the one of *state-action* value function, or the ***Q-function***.
$Q(s,a)$ captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
$Q(s,a)$ is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

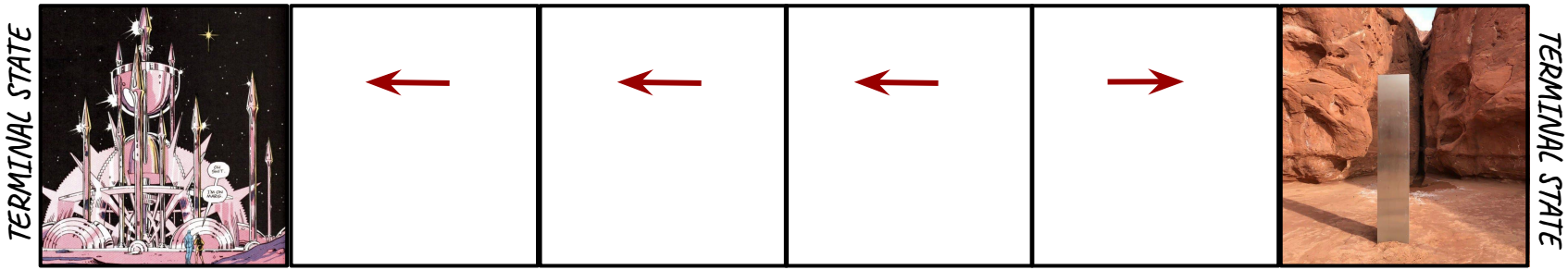
I know, it is a bit of a circular definition (if I already knew the optimal behaviour, why bother measuring Q?), but will become clear in few slides

State-Action value function

A key concept is the one of **state-action** value function, or the **Q-function**.
Q(s,a) captures the expected total future **return** an **agent** in a **state s** can receive by executing a certain **action a**
Q(s,a) is evaluated as the **return** obtained if the **agent**

- starts in **state s**
- takes **action a**, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever **state**.
Let's assume a **policy π** (red arrows) at each **state** so that in **s2**, **s3** and **s4** the **action** is ←, and in **s5** is → as in the previous slide.

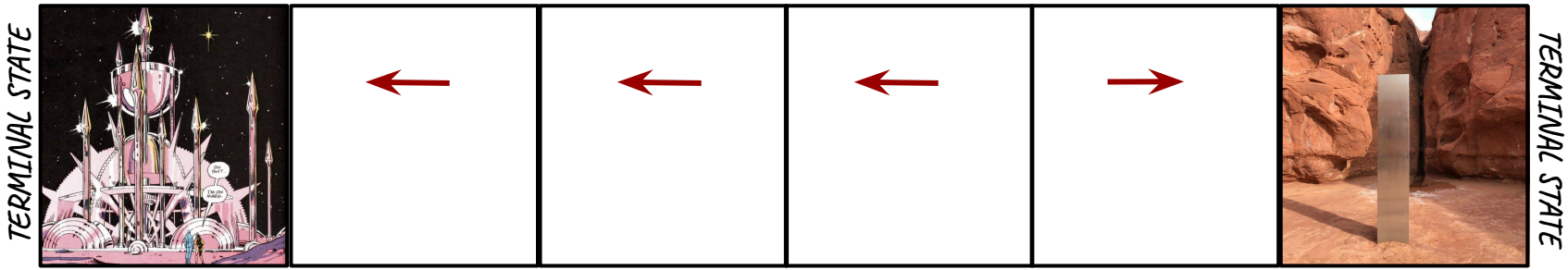


	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

A key concept is the one of *state-action* value function, or the ***Q-function***.
Q(s,a) captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
Q(s,a) is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a ***policy π*** (red arrows) at each *state* so that in *s2*, *s3* and *s4* the *action* is ←, and in *s5* is → as in the previous slide.

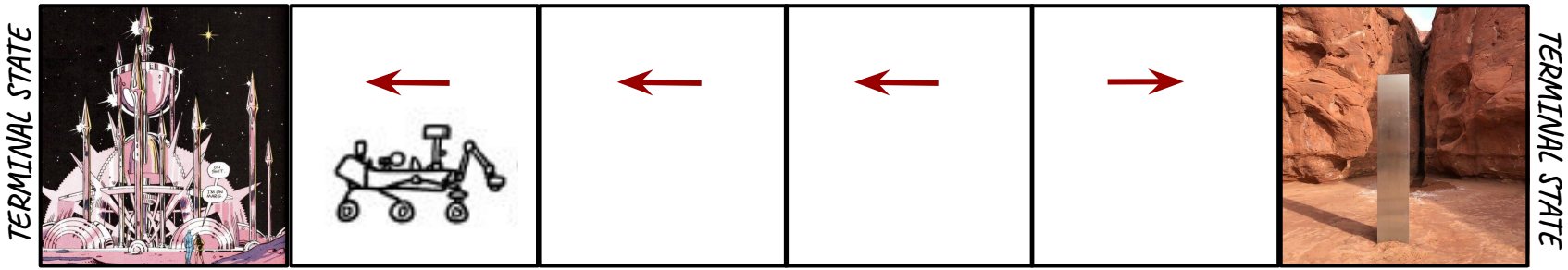


State-Action value function

A key concept is the one of **state-action** value function, or the **Q-function**.
Q(s,a) captures the expected total future **return** an **agent** in a **state s** can receive by executing a certain **action a**
Q(s,a) is evaluated as the **return** obtained if the **agent**

- starts in **state s**
- takes **action a**, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever **state**.
Let's assume a **policy π** (red arrows) at each **state** so that in **s2**, **s3** and **s4** the **action** is ←, and in **s5** is → as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

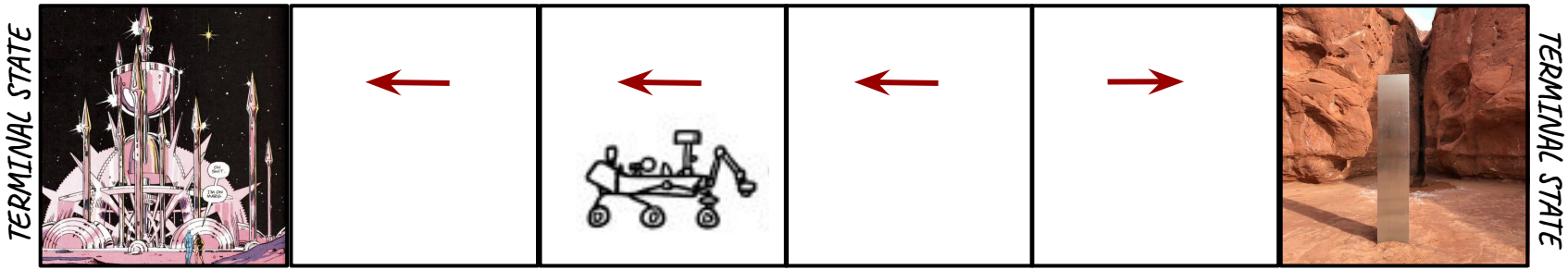
Q(2,→)

State-Action value function

A key concept is the one of *state-action* value function, or the ***Q-function***.
$Q(s,a)$ captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
$Q(s,a)$ is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a ***policy*** π (red arrows) at each *state* so that in *s2*, *s3* and *s4* the *action* is ←, and in *s5* is → as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

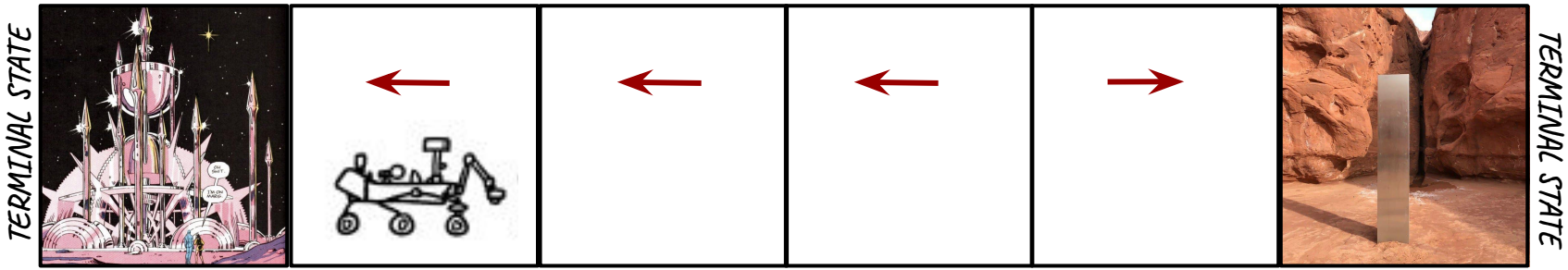
$Q(2,\rightarrow) = 0 + 0.5*0$

State-Action value function

A key concept is the one of *state-action* value function, or the ***Q-function***.
Q(s,a) captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
Q(s,a) is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a ***policy π*** (red arrows) at each *state* so that in *s2*, *s3* and *s4* the *action* is ←, and in *s5* is → as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

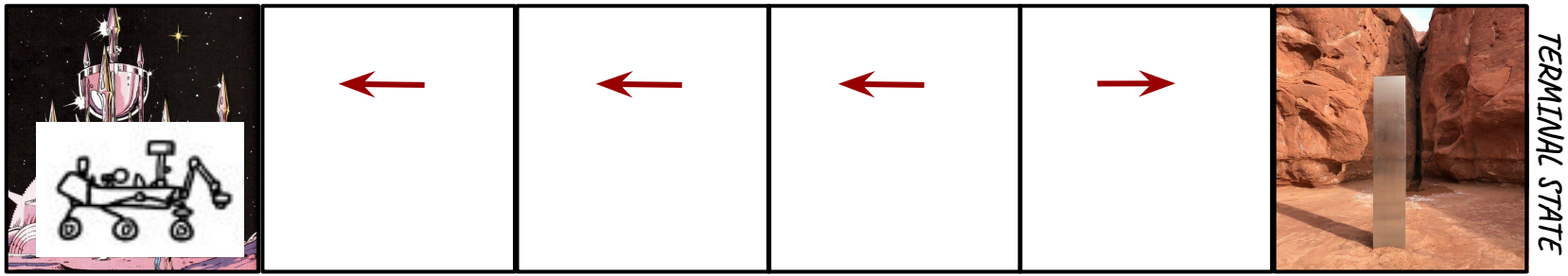
$Q(2,\rightarrow) = 0 + 0.5*0 + (0.5)^2*0$

State-Action value function

A key concept is the one of *state-action* value function, or the *Q-function*.
Q(s,a) captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
Q(s,a) is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a *policy π* (red arrows) at each *state* so that in *s2*, *s3* and *s4* the *action* is ←, and in *s5* is → as in the previous slide.



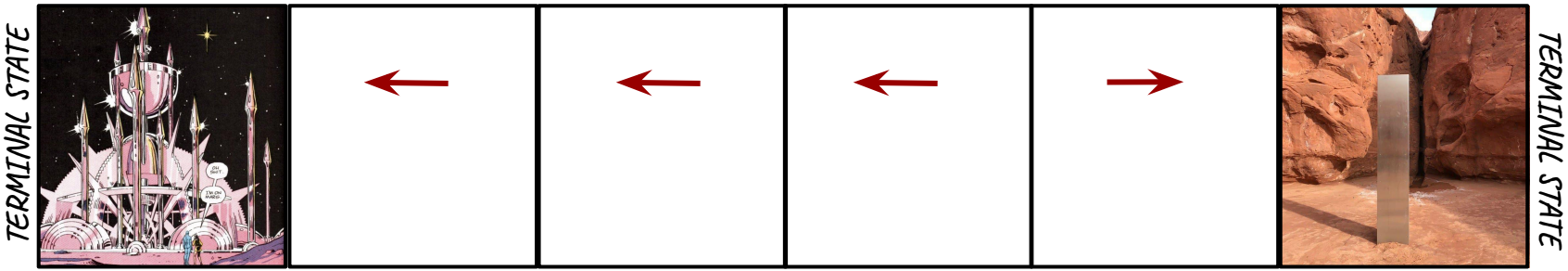
	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

$$Q(2,\rightarrow) = 0 + 0.5*0 + (0.5)^2*0 + (0.5)^3*100 = 12.5$$

A key concept is the one of *state-action* value function, or the ***Q-function***.
$Q(s,a)$ captures the expected total future *return* an *agent* in a *state* s can receive by executing a certain *action* a
$Q(s,a)$ is evaluated as the *return* obtained if the *agent*

- starts in *state* s
- takes *action* a, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a ***policy*** π (red arrows) at each *state* so that in $s2$, $s3$ and $s4$ the *action* is $\leftarrow$, and in $s5$ is $\rightarrow$ as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

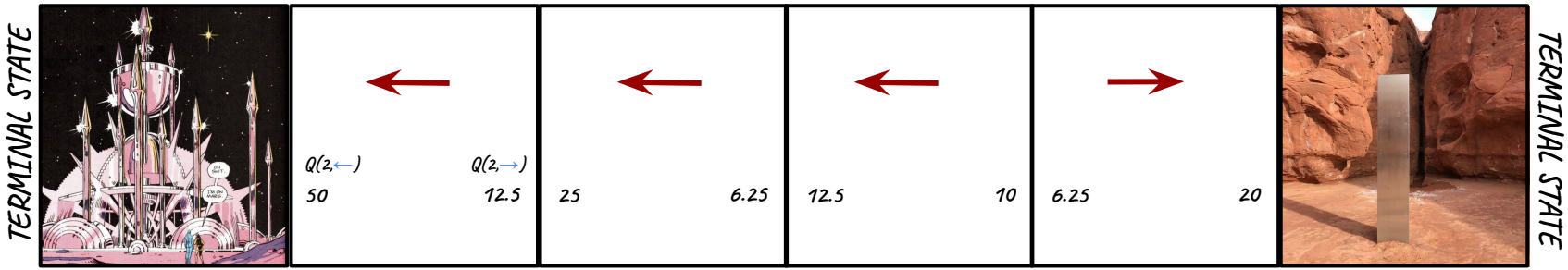
$Q(2,\rightarrow) = 12.5$ $Q(2,\leftarrow) = 50$ $Q(4,\leftarrow) = 12.5$

State-Action value function

A key concept is the one of *state-action* value function, or the ***Q-function***.
$Q(s,a)$ captures the expected total future *return* an *agent* in a *state s* can receive by executing a certain *action a*
$Q(s,a)$ is evaluated as the *return* obtained if the *agent*

- starts in *state s*
- takes *action a*, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever *state*.
Let's assume a ***policy*** π (red arrows) at each *state* so that in *s2*, *s3* and *s4* the *action* is ←, and in *s5* is → as in the previous slide.



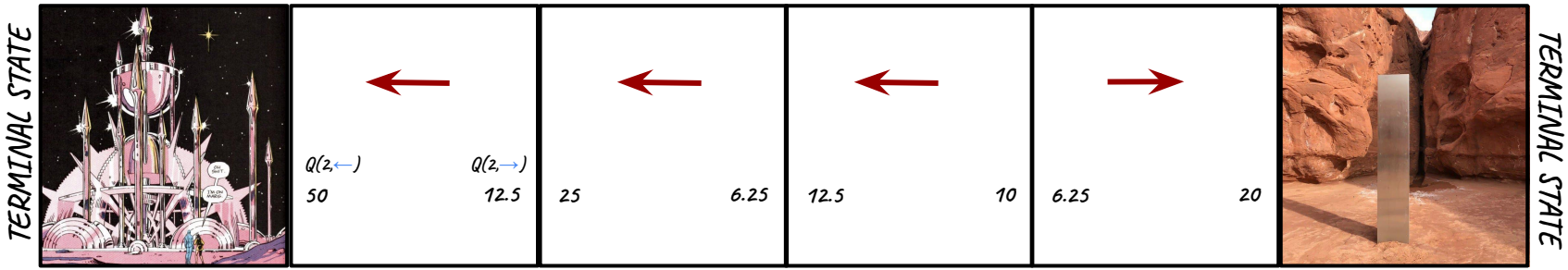
	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

State-Action value function

A key concept is the one of **state-action** value function, or the **Q-function**.
Q(s,a) captures the expected total future **return** an **agent** in a **state s** can receive by executing a certain **action a**
Q(s,a) is evaluated as the **return** obtained if the **agent**

- starts in **state s**
- takes **action a**, once
- then behave optimally after that

Let's get back to the Mars rover example, with $\gamma = 0.5$. The rover now could start in whatever **state**.
Let's assume a **policy π** (red arrows) at each **state** so that in **s2**, **s3** and **s4** the **action** is ←, and in **s5** is → as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

The best possible **return** from state **s** is **max Q(s, a)**. The best possible **action** in **s** is the one with **max Q(s,a)**. The **policy** are those **actions a**.

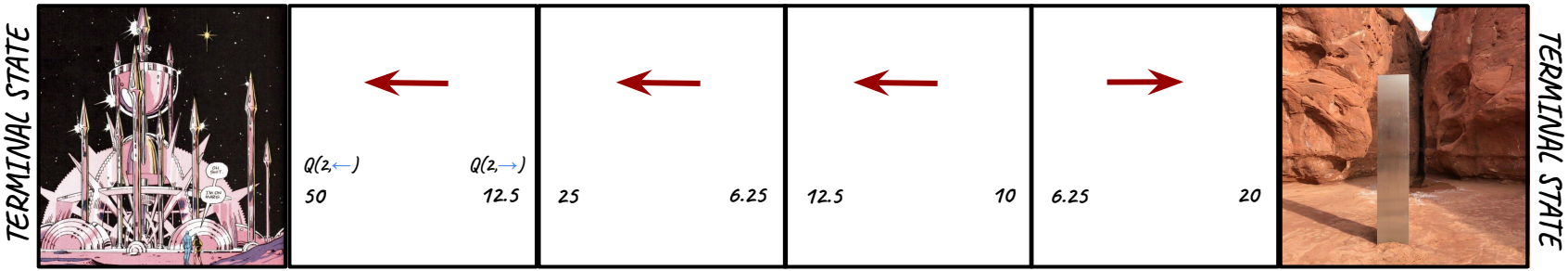
State-Action value function

A key concept is the one of **state-action** value function, or the **Q-function**.
Q(s,a) captures the expected total future **return** an **agent** in a **state s** can receive by executing a certain **action a**
Q(s,a) is evaluated as the **return** obtained if the **agent**

- starts in **state s**
- takes **action a**, once
- then behave optimally after that

Now the **Q-function** definition makes more sense.
The only thing left is: how do I *actually* measure the optimal **policy π(s)**?

Let's get back to the Mars rover example, with **γ = 0.5**. The rover now could start in whatever **state**.
Let's assume a **policy π** (red arrows) at each **state** so that in **s2**, **s3** and **s4** the **action** is ←, and in **s5** is → as in the previous slide.



	1	2	3	4	5	6
Reward:	100	0	0	0	0	40
Return:	100	50	25	12.5	20	40

The best possible **return** from state **s** is **max Q(s, a)**. The best possible **action** in **s** is the one with **max Q(s,a)**. The **policy** are those **actions a**.

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

Bellman equation

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation

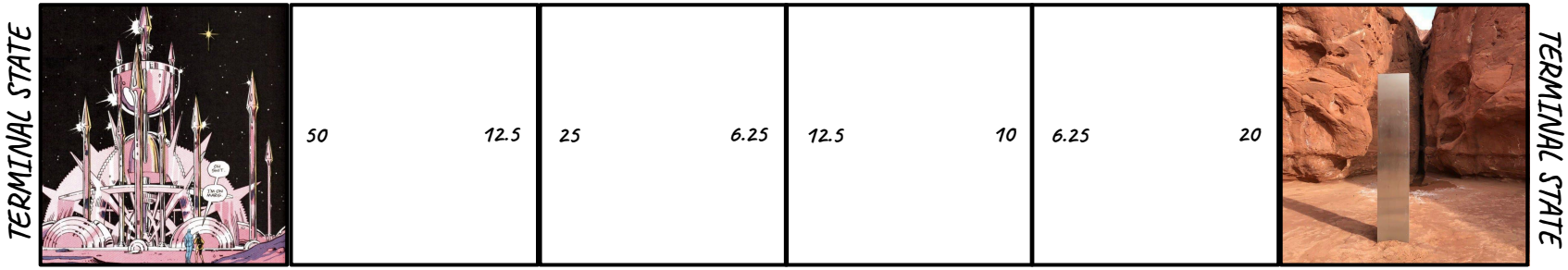
Bellman equation

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation

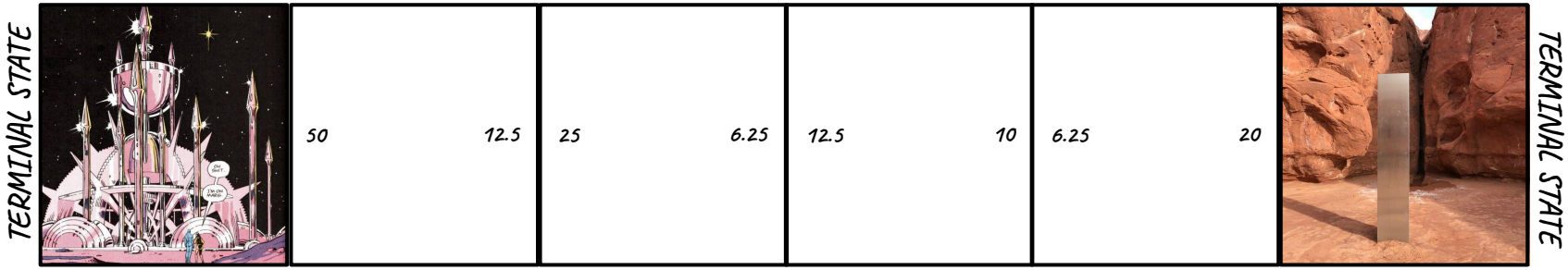


If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation



$$Q(2, \rightarrow) = R(2) + 0.5 \max_{a'} Q(3, a') = R(2) + 0.5 * 25 = 12.5$$

$$Q(4, \leftarrow) = R(4) + 0.5 \max_{a'} Q(3, a') = R(4) + 0.5 * 25 = 12.5$$

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

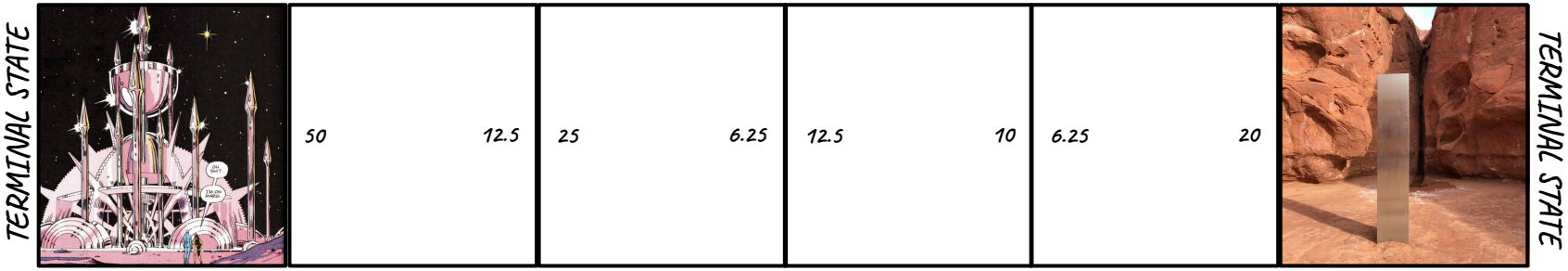
- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation

return you get right away

return from behaving optimally starting from s'



$$Q(2, \rightarrow) = R(2) + 0.5 \max_{a'} Q(3, a') = R(2) + 0.5 * 25 = 12.5$$

$$Q(4, \leftarrow) = R(4) + 0.5 \max_{a'} Q(3, a') = R(4) + 0.5 * 25 = 12.5$$

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

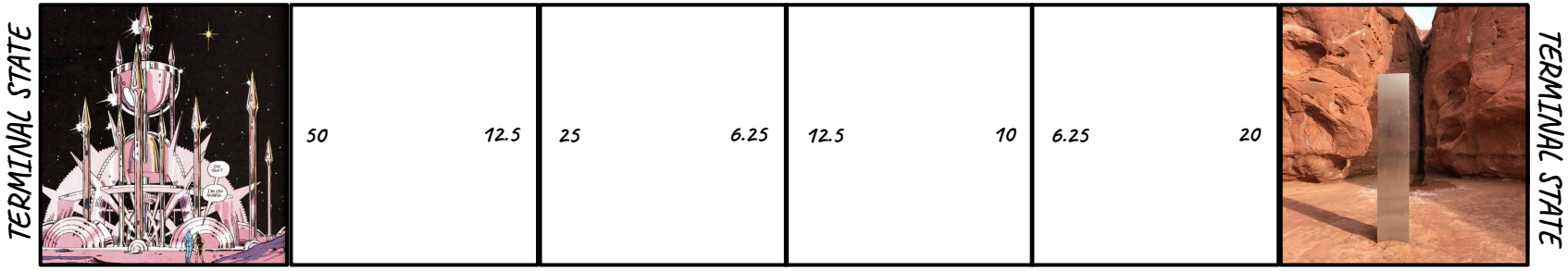
- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation

$R(s)$: return you get right away

$\max_{a'} Q(s', a')$: return from behaving optimally starting from s'



All of those is quite deterministic. In reality, you have to generalize to random (stochastic) environments.
The consequence is that there is a distribution of possible rewards you get, depending on stochastic phenomena. The return becomes:

$$\text{Expected Return} = \text{average}(R_1 + \gamma R_2 + \gamma^2 R_3 + \dots)$$

which is what you expect to get, *on average*

If you have a measure of $Q(s,a)$, then you have a way to pick an action a at each state s. What is the mathematical expression for $Q(s,a)$?
Let's call:

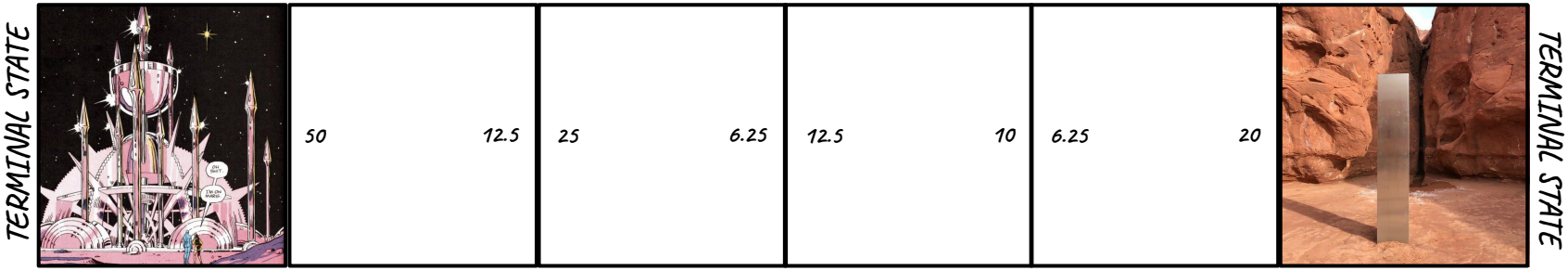
- s the current state
- a the current action
- $R(s)$ the reward of current state
- s' the state the agent gets to after taking action a
- a' the action the agent takes in state s'

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

This is called the Bellman Equation

return you get right away

return from behaving optimally starting from s'



All of those is quite deterministic. In reality, you have to generalize to random (stochastic) environments.
The consequence is that there is a distribution of possible rewards you get, depending on stochastic phenomena. The Bellman eq. becomes:

$$Q(s,a) = R(s) + \gamma E[\max_{a'} Q(s', a')]$$

The ultimate goal of Reinforcement Learning is:

to choose a **policy** $\pi(s) = a$ that will tell the agent what **action** a to take in **state** s so as to maximize the **expected return:**

$$Q(s,a) = R(s) + \gamma\, E[\max_{a'} Q(s', a')]$$

Don't worry, we are arriving to the *reinforcement* part, which is the way a Neural Network is able to approximate the right hand side of the **Bellman equation** by starting with a random **policy** and slowly arriving to one that maximizes the **expected return**.

From discrete states to continuous states

The ultimate goal of Reinforcement Learning is:

to choose a **policy** $\pi(s) = a$ that will tell the agent what **action** a to take in **state** s so as to maximize the **expected return:**

$$Q(s,a) = R(s) + \gamma\, E[\max_{a'} Q(s', a')]$$

Don't worry, we are arriving to the *reinforcement* part, which is the way a Neural Network is able to approximate the right hand side of the **Bellman equation** by starting with a random **policy** and slowly arriving to one that maximizes the **expected return.**

One last thing: the Mars rover example was a **discrete states** example. In reality you will have **continuous states** examples.
That means that the **state** is not just a number out of discrete values, but *a vector of numbers*, any of which can take any kind of values.

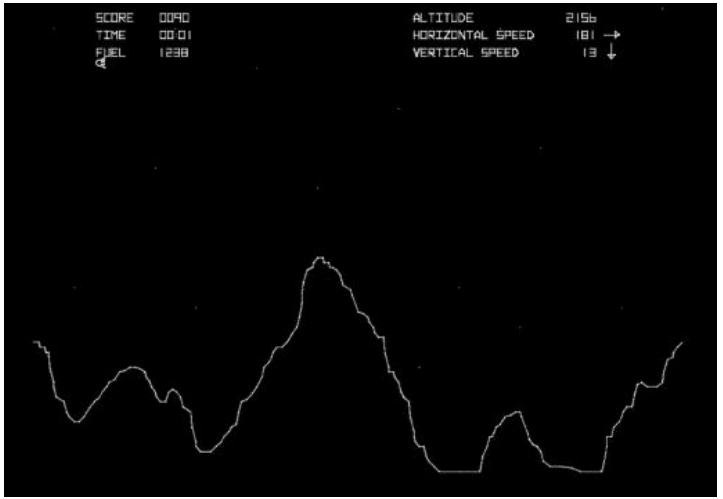
e.g. for the lunar lander game a **state** is a vector of:

$s = [x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, l, r]$

if the left/right leg is sitting on ground (0/1)

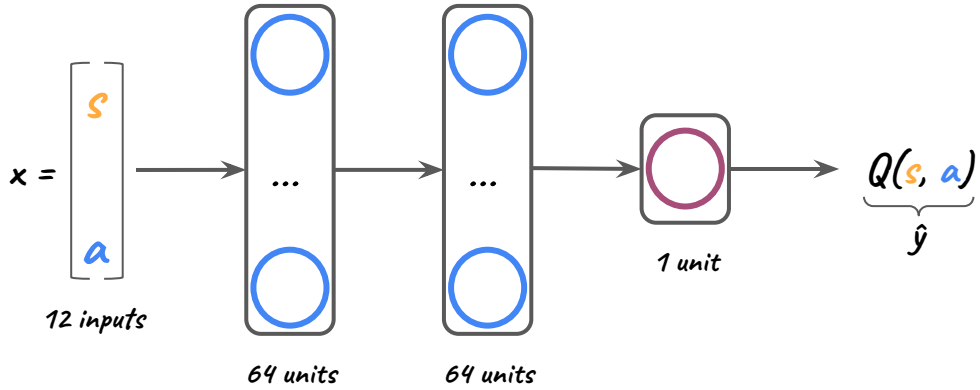
and the **actions**

- do nothing
- fire left thruster
- fire right thruster
- fire main thruster

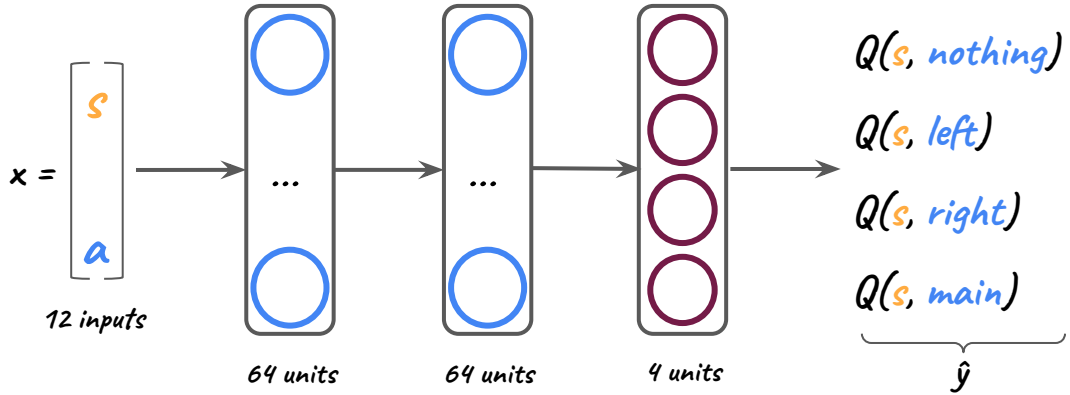


The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.

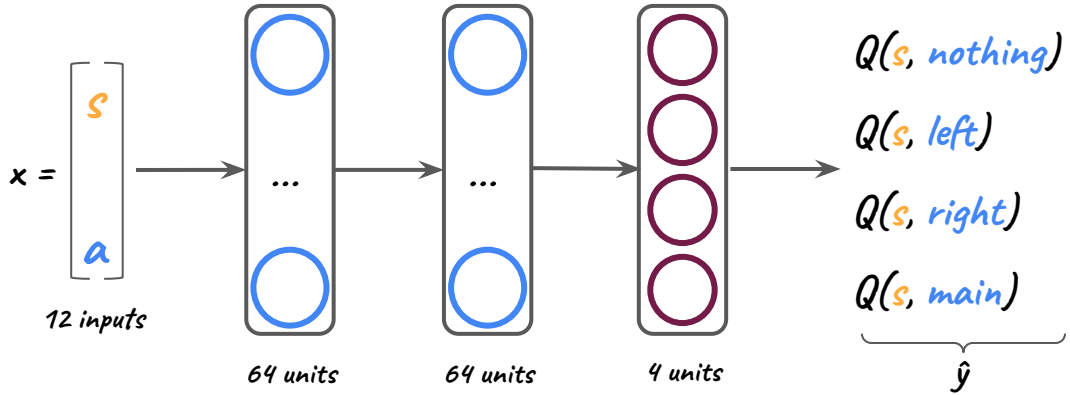
The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



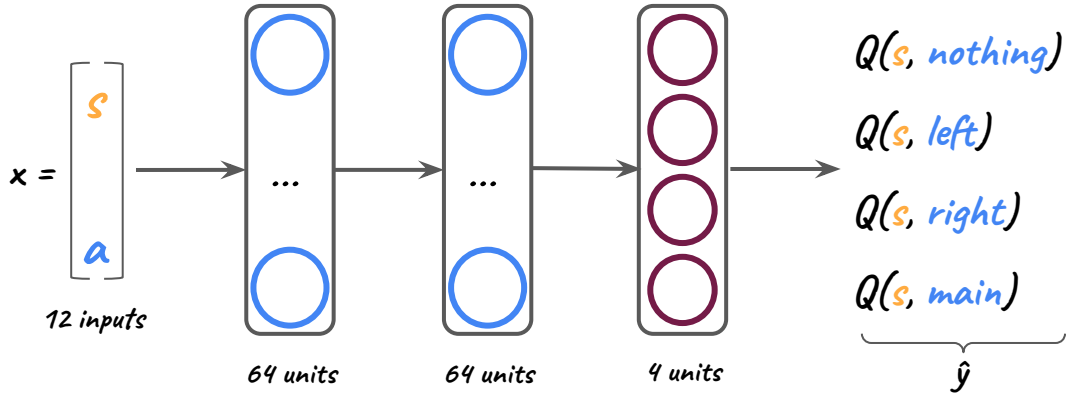
The approach is to use the *Bellman equation* to create a training set (x, y), and train the Neural Network to learn a mapping $x \rightarrow y$ from *state-action* pairs to the target value $Q(s, a)$:

$$\underbrace{Q(s,a)}_{x} = \underbrace{R(s) + \gamma \, E[\max_{a'} Q(s', a')]}_{y}$$

At first the *agent* is just going to take an *action* $a^{(1)}$ randomly, it will have some rewards $R(s^{(1)})$ for being in that *state* $s^{(1)}$, and as a result it will arrive to *state* $s'^{(1)}$. Repeat this multiple times, take the *state-action* pairs as x and the *reward - state* s' pairs as y:

Learning the state-value function

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The approach is to use the *Bellman equation* to create a training set (x, y), and train the Neural Network to learn a mapping $x \rightarrow y$ from *state-action* pairs to the target value $Q(s, a)$:

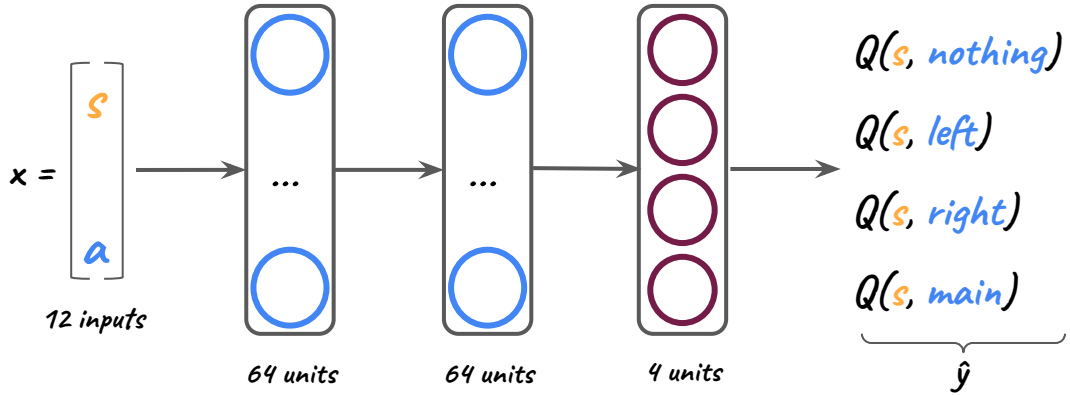
$$\underbrace{Q(s,a)}_{x} = \underbrace{R(s) + \gamma\, E[\max_{a'} Q(s', a')]}_{\hat{y}}$$

At first the *agent* is just going to take an *action* $a^{(1)}$ randomly, it will have some rewards $R(s^{(1)})$ for being in that *state* $s^{(1)}$, and as a result it will arrive to *state* $s'^{(1)}$. Repeat this multiple times, take the *state-action* pairs as x and the *reward* - *state* s' pairs as y:

$$(\underbrace{s^{(i)}, a^{(i)}}_{x}, \overbrace{R(s^{(i)}), s'^{(i)}}^{y})$$

and in this way you build a first training set (x, y)

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The approach is to use the *Bellman equation* to create a training set (x, y), and train the Neural Network to learn a mapping $x \rightarrow y$ from *state-action* pairs to the target value $Q(s, a)$:

$$Q(s,a) = R(s) + \gamma \, E[\max_{a'} Q(s', a')]$$

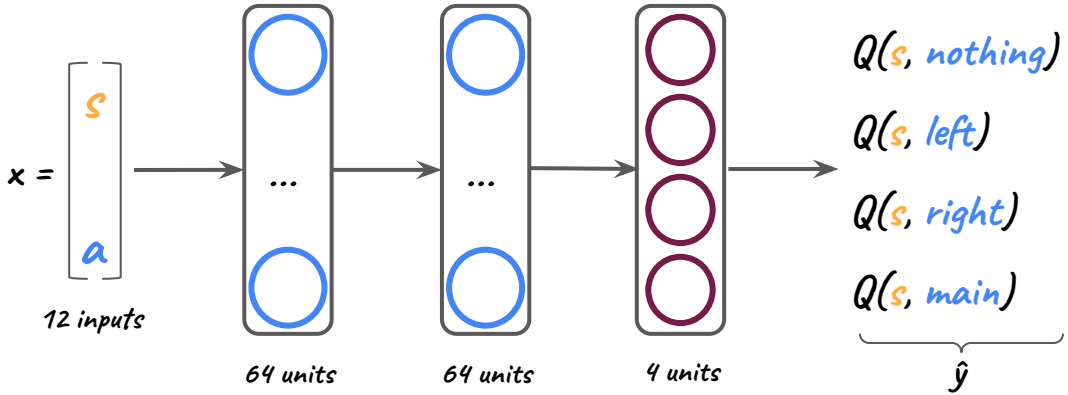
initially we have no idea about $Q(s', a')$, so we will take random values

At first the *agent* is just going to take an *action* $a^{(1)}$ randomly, it will have some rewards $R(s^{(1)})$ for being in that *state* $s^{(1)}$, and as a result it will arrive to *state* $s'^{(1)}$. Repeat this multiple times, take the *state-action* pairs as x and the *reward - state* s' pairs as y:

$$(s^{(i)}, a^{(i)}, R(s^{(i)}), s'^{(i)})$$

and in this way you build a first training set (x, y)

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The *Deep-Q Network (DQN)* algorithm works this way:

Initialize Neural Network randomly as guess of $Q(s, a)$

Repeat:

Take *actions* in the lunar lander, and store the 10k most recent tuples of $(s, a, R(s), s')$ to save memory (*replay buffer*)

Train the Neural Network:

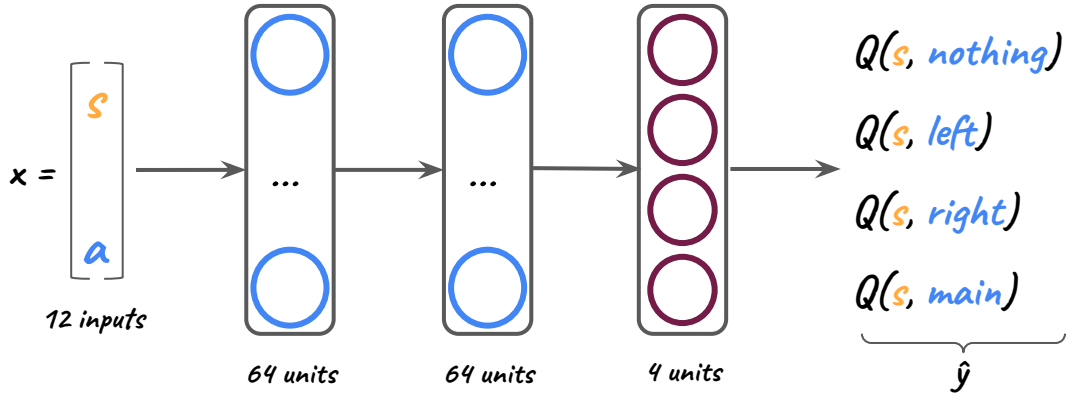
Create training set of 10k examples using

$$x = (s, a) \text{ and } y = R(s) + \gamma \max_{a'} Q(s', a')$$

Train Q_{new} such that $Q_{new}(s, a) = y$

Set $Q = Q_{new}$

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The *Deep-Q Network (DQN)* algorithm works this way:

Initialize Neural Network randomly as guess of $Q(s, a)$

Repeat:

Take *actions* in the lunar lander, and store the 10k most recent tuples of $(s, a, R(s), s')$ to save memory (*replay buffer*)

Train the Neural Network:

Create training set of 10k examples using

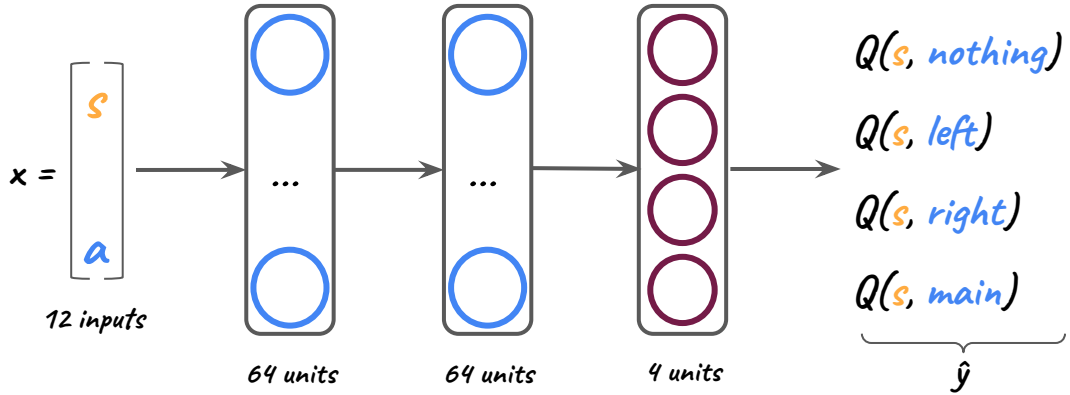
$$x = (s, a) \text{ and } y = R(s) + \gamma \max_{a'} Q(s', a')$$

Train Q_{new} such that $Q_{new}(s, a) = y$

Set $Q = Q_{new}$

Play the lunar lander game and save the results. In the first iterations the *agent* will just do random things; the more the playing, the more intelligent things the *agent* will do.

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The *Deep-Q Network (DQN)* algorithm works this way:

Initialize Neural Network randomly as guess of $Q(s, a)$

Repeat:

Take *actions* in the lunar lander, and store the 10k most recent tuples of $(s, a, R(s), s')$ to save memory (*replay buffer*)

Train the Neural Network:

Create training set of 10k examples using

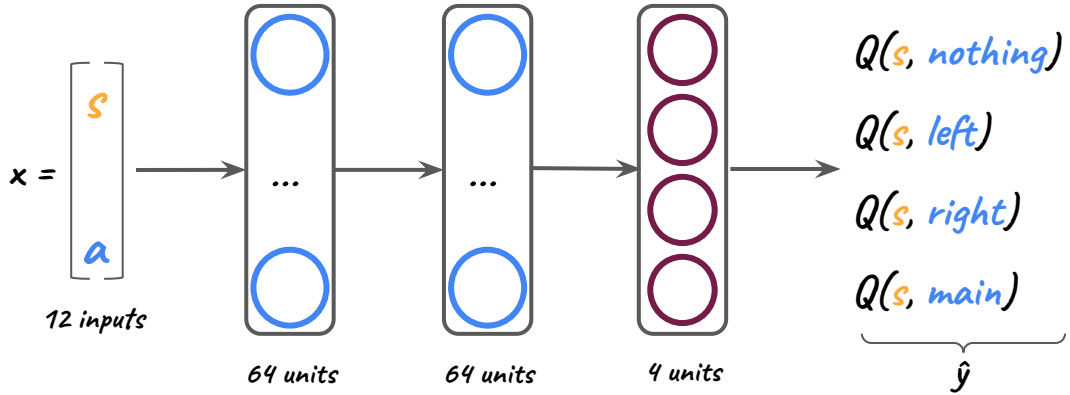
$x = (s, a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$

Train Q_{new} such that $Q_{new}(s, a) = y$

Set $Q = Q_{new}$

From all the games you've played, you know the set of chosen *actions* and the consequences they led to: this is exactly the right-hand side of the *Bellman equation*.

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal *policy* π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



The *Deep-Q Network (DQN)* algorithm works this way:

Initialize Neural Network randomly as guess of $Q(s, a)$

Repeat:

Take *actions* in the lunar lander, and store the 10k most recent tuples of $(s, a, R(s), s')$ to save memory (*replay buffer*)

Train the Neural Network:

Create training set of 10k examples using

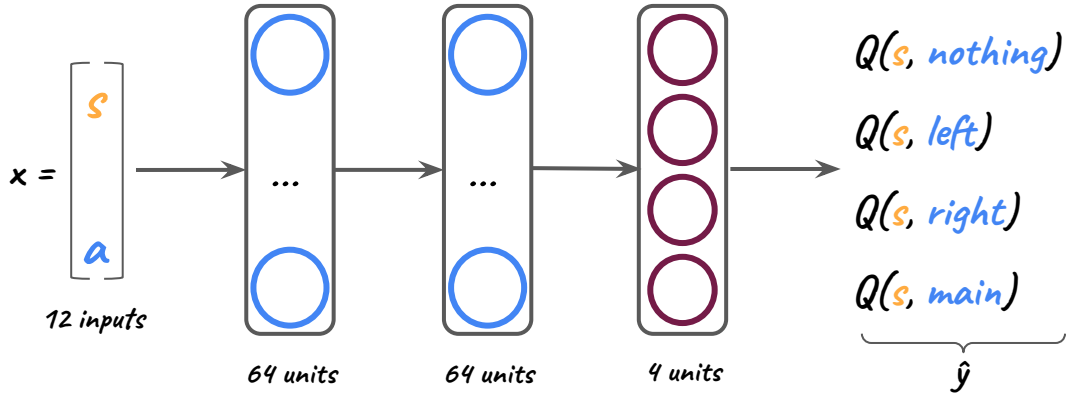
$x = (s, a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$

Train Q_{new} such that $Q_{new}(s, a) = y$

Set $Q = Q_{new}$

Assume that what you've learned is the optimal *state-value function* Q, and play the game again.
Repeat until the *agent* will always land without crashing.

The key idea is to train a Neural Network to compute (or approximate) the state-value function $Q(s, a)$ and how that intel could lead the agent to pick good *actions* and thus the optimal ***policy*** π.
Let's take the lunar lander example: an eight dimensional array for *states*, and four possible discrete *action*.



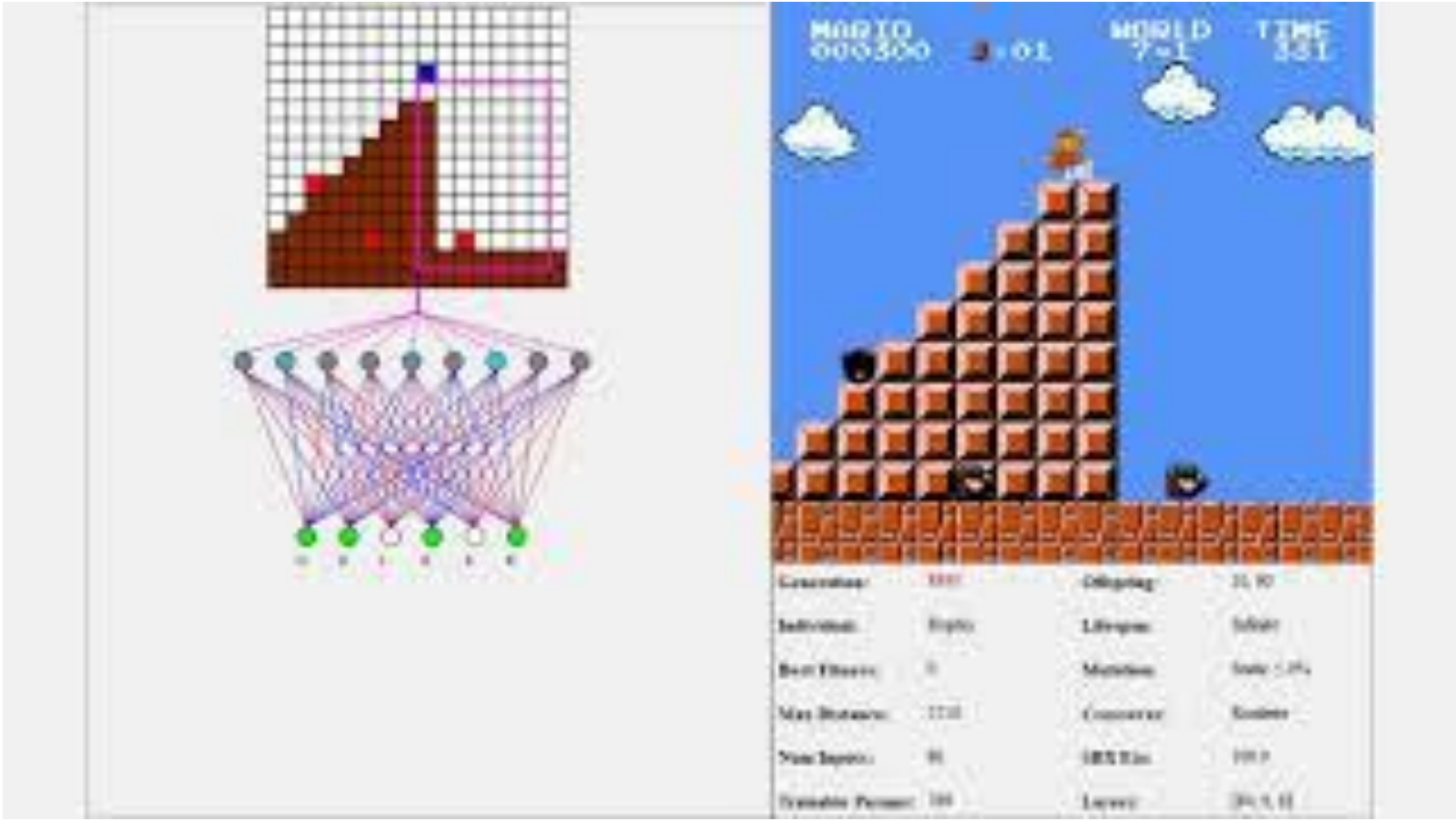
How do you pick the actions while the agent is still learning? One option is the ***ε-greedy policy.***

- with a certain probability, e.g. 0.95, pick the *action a* that actually maximizes $Q(s, a)$ → ***greedy action***
- with the remaining probability, e.g. 0.05, pick a random *action* → ***exploration step***

This will allow the agent to explore random things and see if they actually make sense in some certain *states*.
An option is to start with a certain high value of ε, and gradually decrease it (e.g. from 1.0 to 0.01).

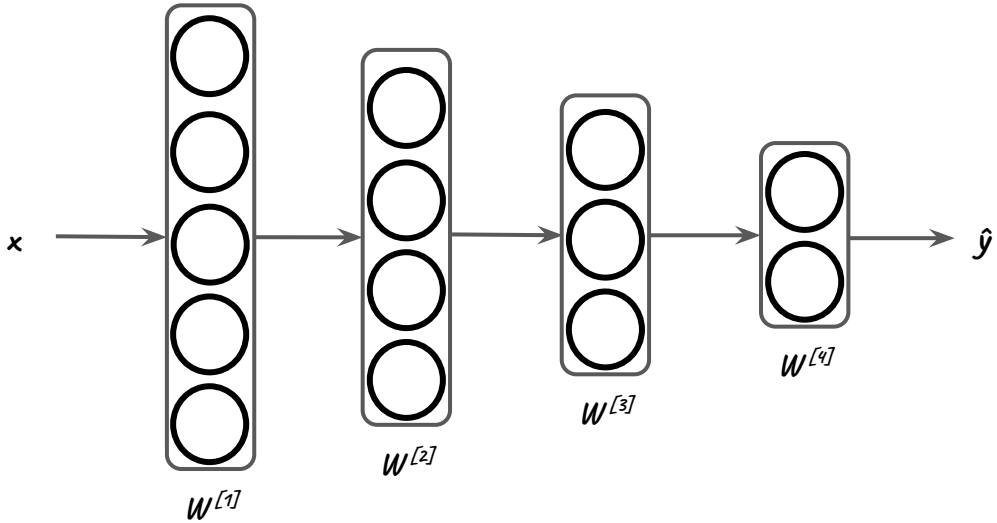
One last refinement: ***soft-update***. Changing the weights from Q to Q_{new} every time could be too abrupt of a change. With ***soft-update*** you can control how aggressively the agent changes behaviour, e.g. $W := 0.01\, W_{new} + 0.99\, W$

Reinforcement Learning to play SuperMario



Where were we with DL?

Last time we learned the basics about Neural Networks and Deep Learning.
A Neural Network is actually a set of layers composed of hidden units, taking some features x in input and predicting a label $\hat{y}$ based on the learned weights W, which are trained on a training set with some optimized version of *gradient descent*, e.g. Adam.
The simple example here above is a 4-layers Network, and those layers are called **"fully connected"** layers or **"dense"** layers.



This thing is actually doing this operation for each layer:

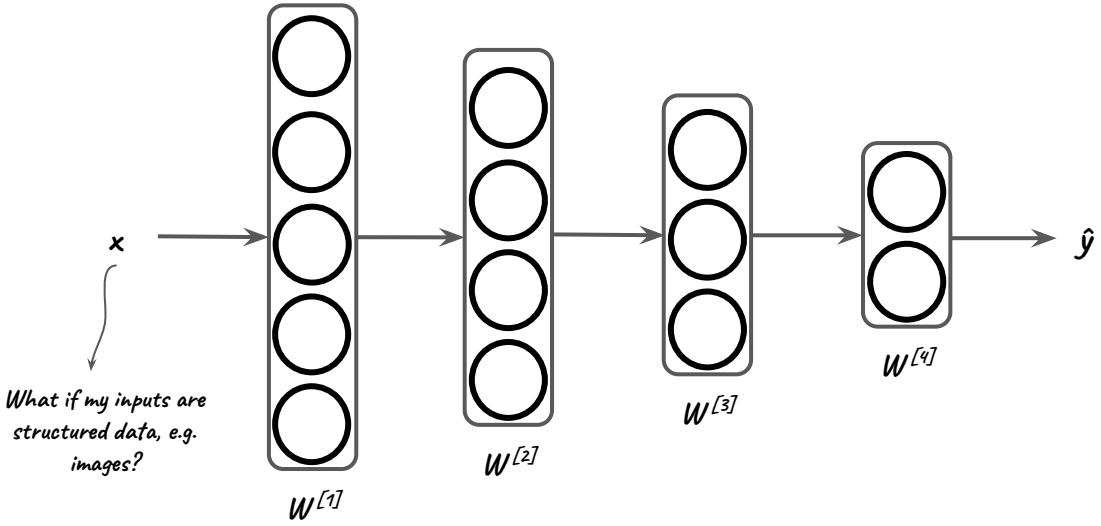
$$a_j^{[l]} = g(w_j^{[l]} a^{[l-1]} + b_j^{[l]})$$

DEEP LEARNING

This is a deep learning algorithm, in the sense that once you start to stack layers on layers you'll notice how the first layers *learn simple structures* from your data, and the deeper you go with the model, the more *complex things* it is able to learn and reproduce.

Where were we with DL?

Last time we learned the basics about Neural Networks and Deep Learning.
A Neural Network is actually a set of layers composed of hidden units, taking some features x in input and predicting a label $\hat{y}$ based on the learned weights W, which are trained on a training set with some optimized version of *gradient descent*, e.g. Adam.
The simple example here above is a 4-layers Network, and those layers are called **"fully connected"** layers or **"dense"** layers.



This thing is actually doing this operation for each layer:

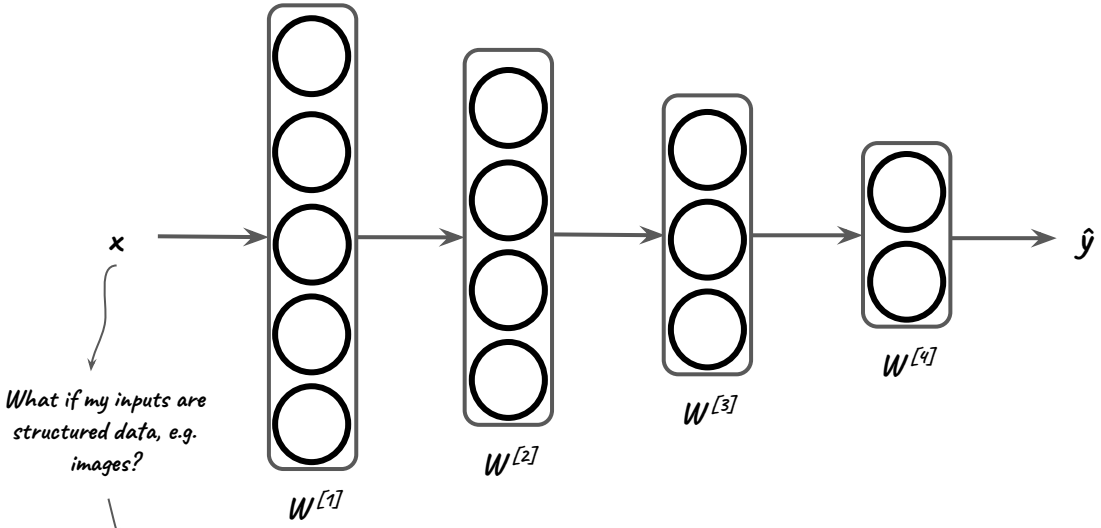
$$a_j^{[l]} = g(w_j^{[l]} a^{[l-1]} + b_j^{[l]})$$

DEEP LEARNING

This is a deep learning algorithm, in the sense that once you start to stack layers on layers you'll notice how the first layers *learn simple structures* from your data, and the deeper you go with the model, the more *complex things* it is able to learn and reproduce.

Where were we with DL?

Last time we learned the basics about Neural Networks and Deep Learning.
A Neural Network is actually a set of layers composed of hidden units, taking some features x in input and predicting a label $\hat{y}$ based on the learned weights W, which are trained on a training set with some optimized version of *gradient descent*, e.g. Adam.
The simple example here above is a 4-layers Network, and those layers are called **"fully connected"** layers or **"dense"** layers.



This thing is actually doing this operation for each layer:

$$a_j^{[l]} = g(w_j^{[l]} a^{[l-1]} + b_j^{[l]})$$

DEEP LEARNING

This is a deep learning algorithm, in the sense that once you start to stack layers on layers you'll notice how the first layers *learn simple structures* from your data, and the deeper you go with the model, the more *complex things* it is able to learn and reproduce.

The core concept of computer vision is the *convolutional layer*, whose key idea is the use of the *convolutions*.

Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.

Input matrix

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

*

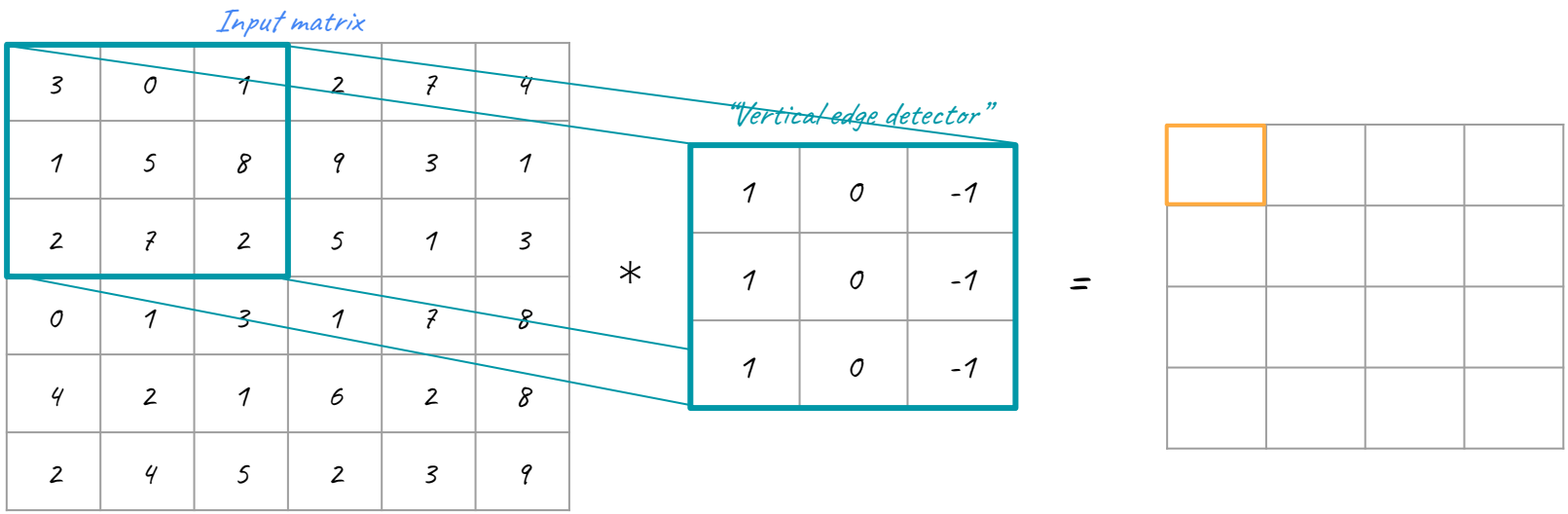
"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

=

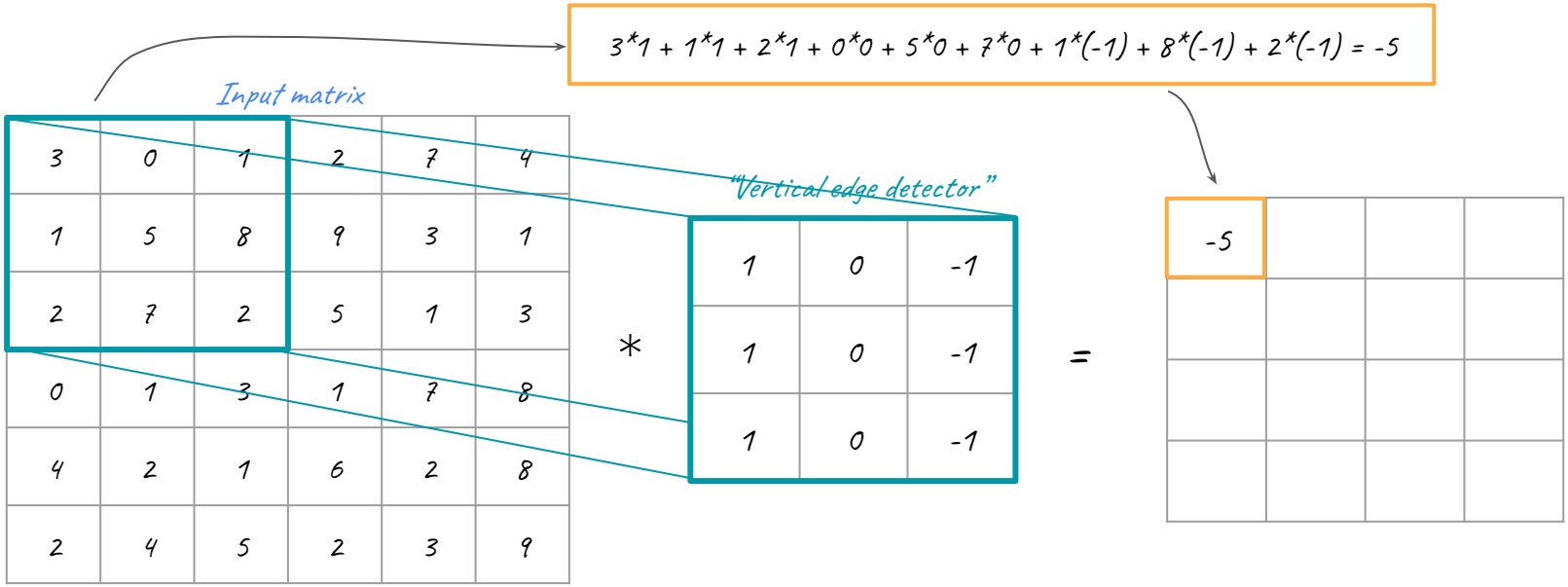
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



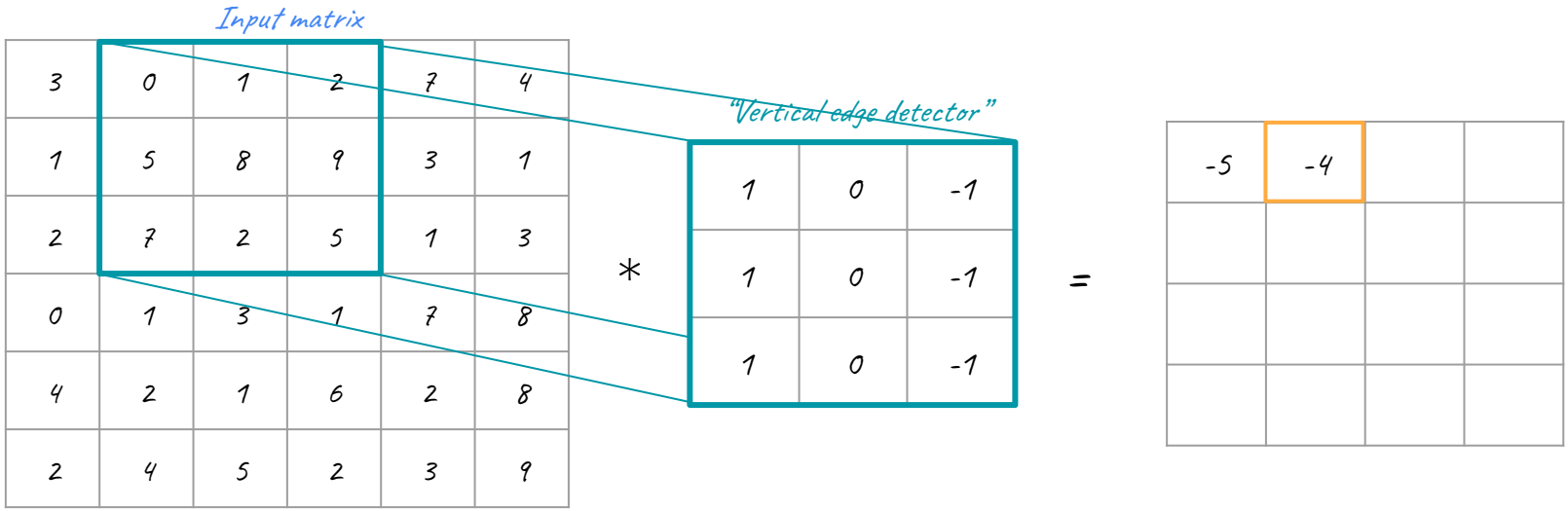
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



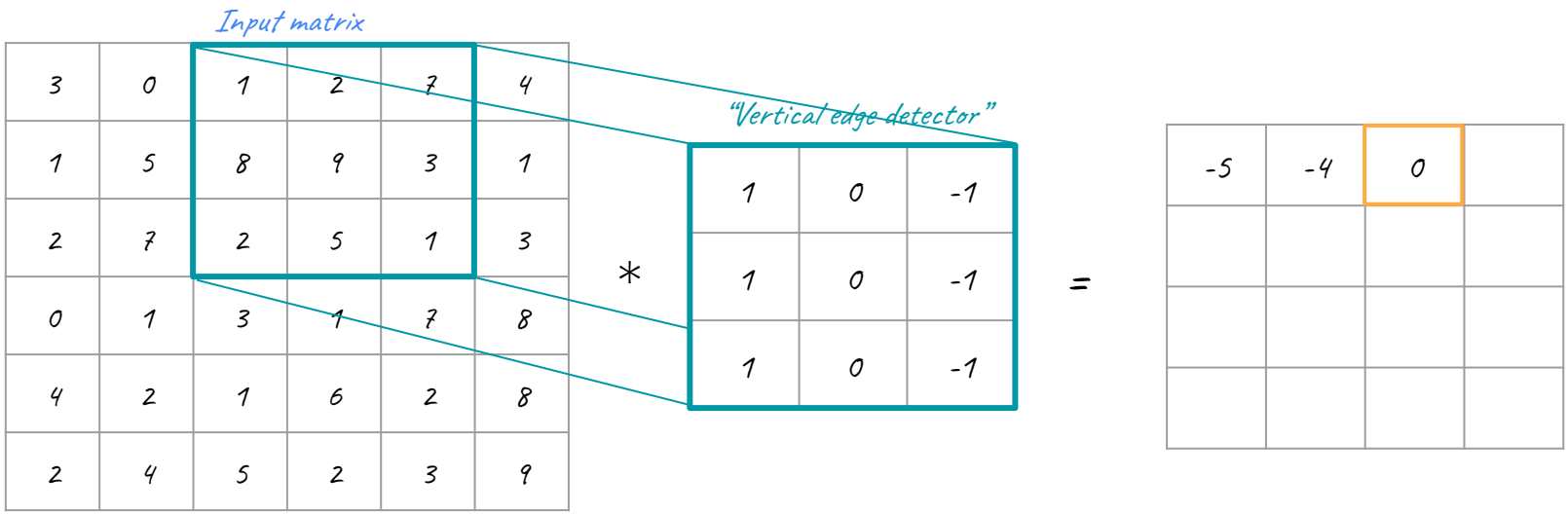
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



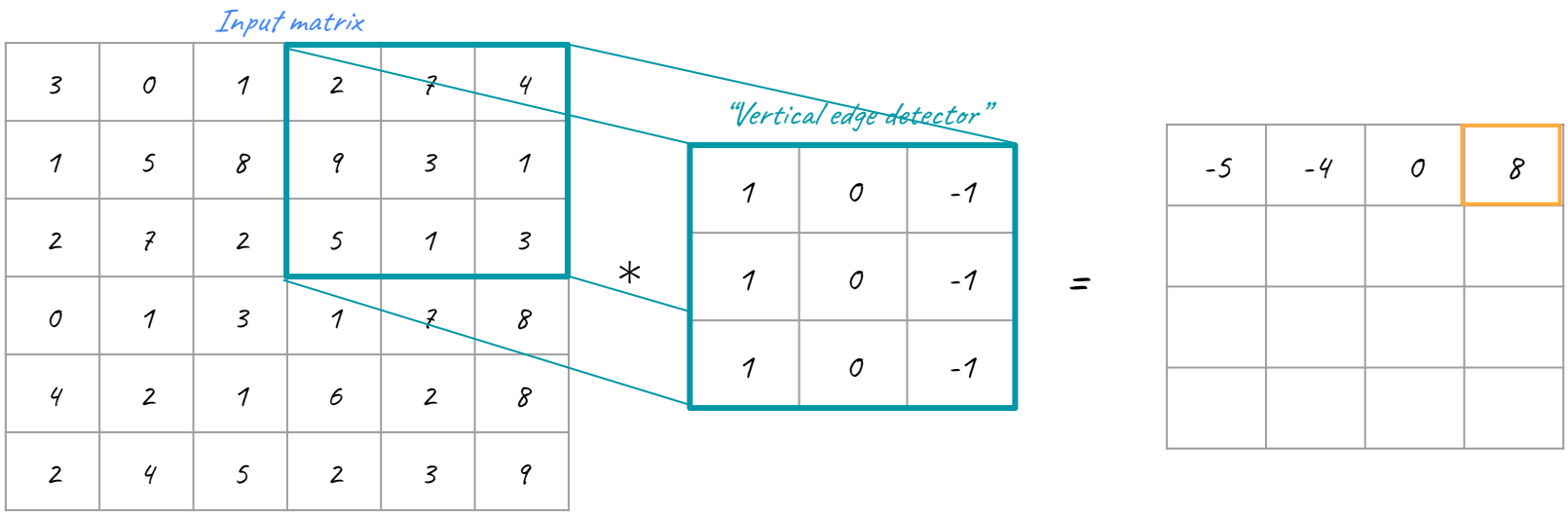
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



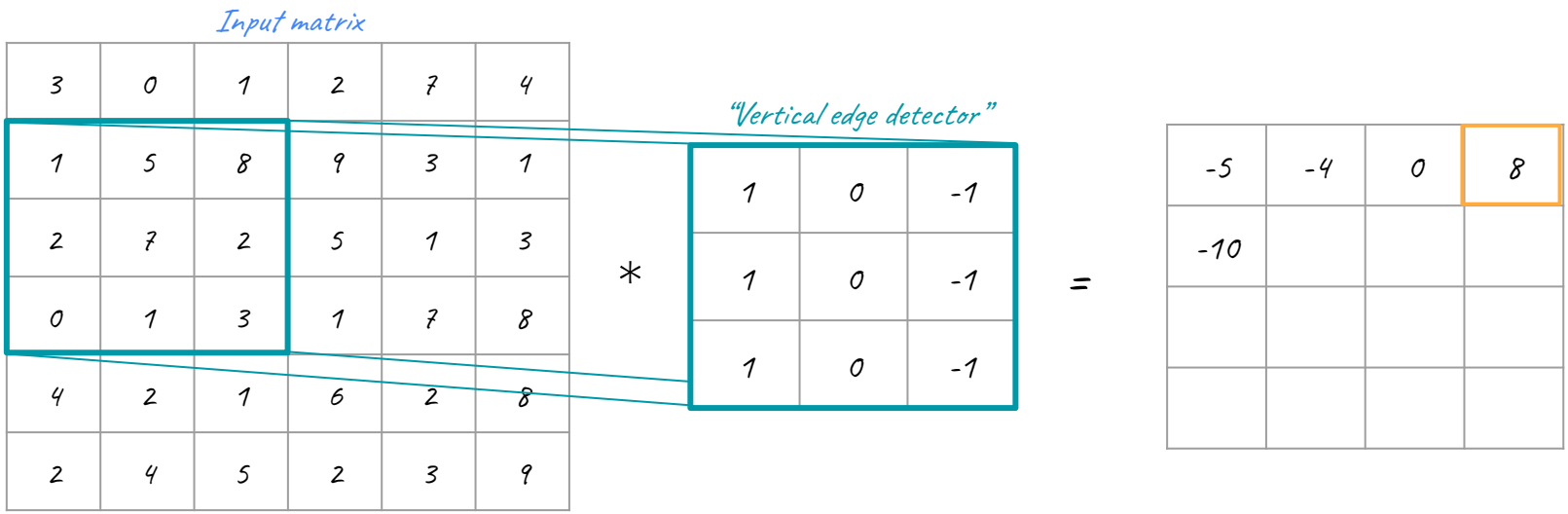
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.



Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.

Input matrix

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

$*$

"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

$=$

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.

Input matrix

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

$*$

"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

$=$

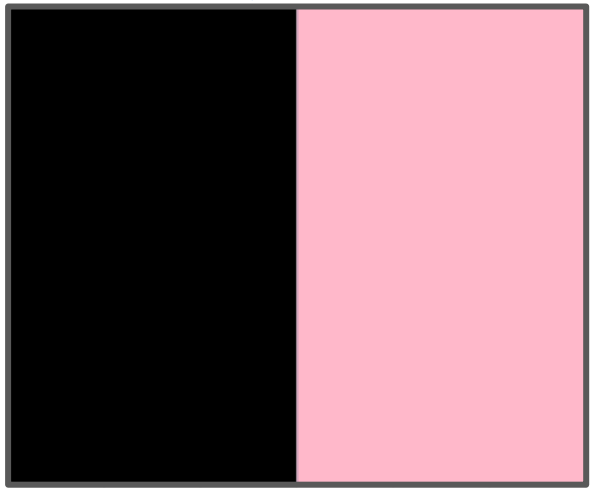
-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

That's **convolution** in a nutshell. Apply the *fxf filter* to the *nxn input matrix*, summing everything within the *fxf* windows applied to the *nxn* matrix, and you'll come out with a $(n-f+1)\times(n-f+1)$ matrix.

Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?

Input matrix



*

"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

=

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?

Input matrix

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

=

Yes, I know, technically 10 and 0 is some kind of grey and white and you need a 3 matrix array to do rosa, but I don't care, these are my slides and I wanted to do the Palermo FC flag.

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?

Input matrix

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

"Vertical edge detector"

1	0	-1
1	0	-1
1	0	-1

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

Yes, I know, technically 10 and 0 is some kind of grey and white and you need a 3 matrix array to do rosa, but I don't care, these are my slides and I wanted to do the Palermo FC flag.

The vertical edge detector filter actually detected, well, vertical edges!

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?

Input matrix

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

$*$

Generic Filter

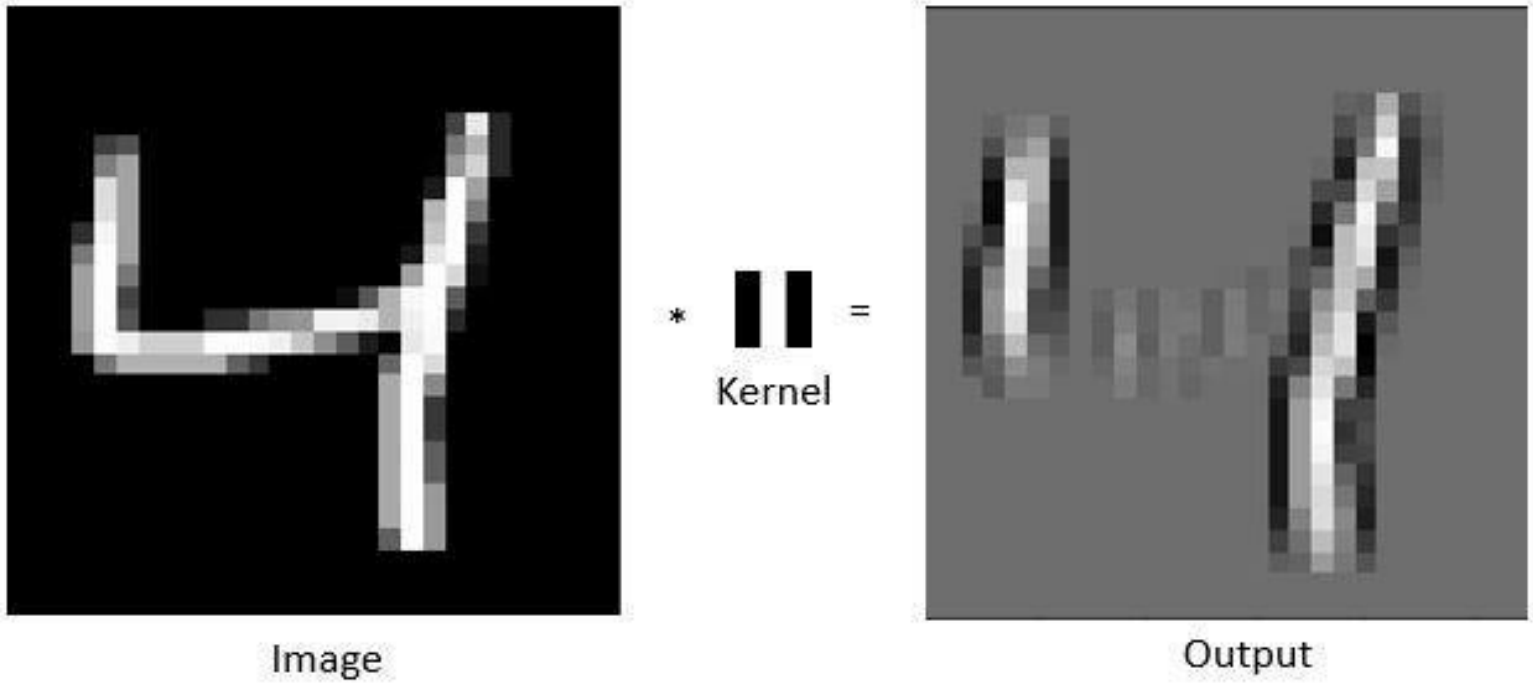
w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

There are numerous available filters, and in principle you could just treat the number as model parameters w_i and let backpropagation learn the best values.

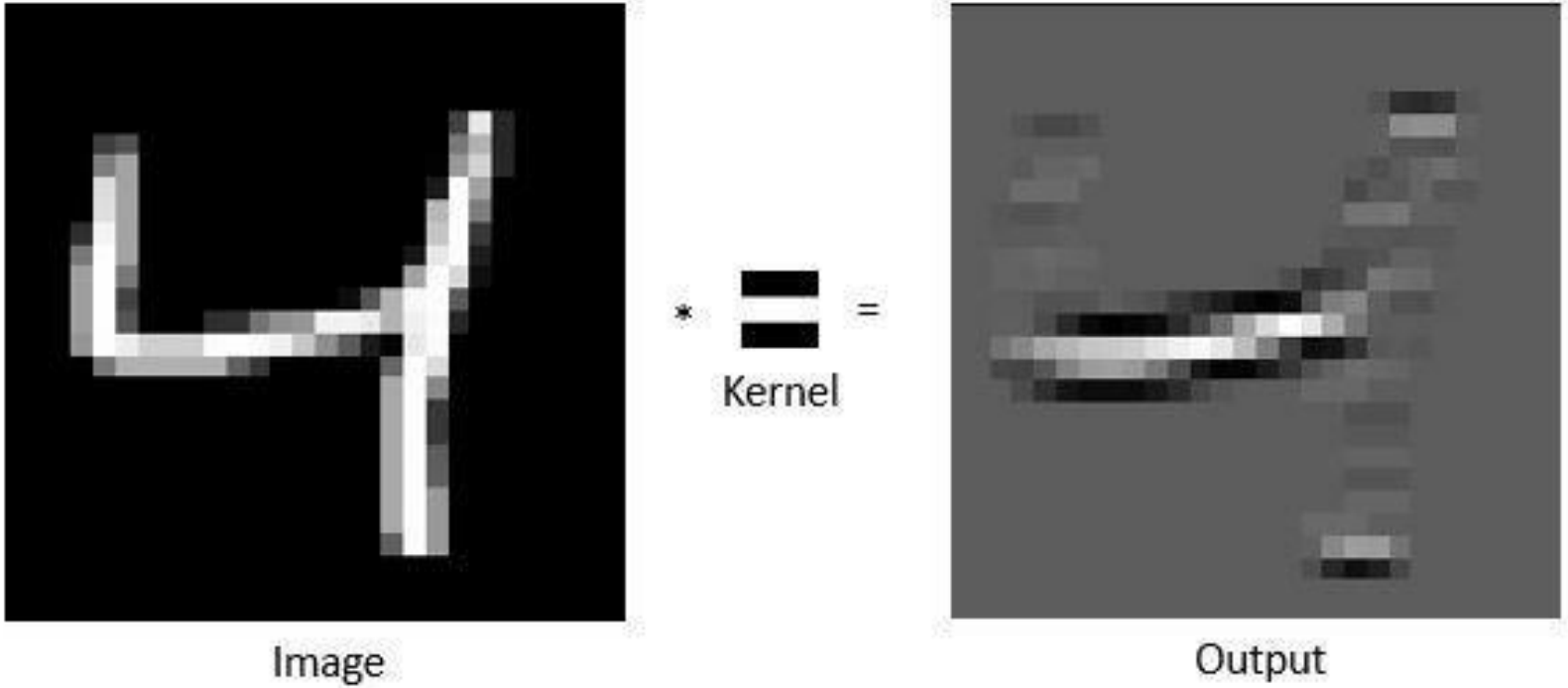
Yes, I know, technically 10 and 0 is some kind of grey and white and you need a 3 matrix array to do rosa, but I don't care, these are my slides and I wanted to do the Palermo FC flag.

Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?

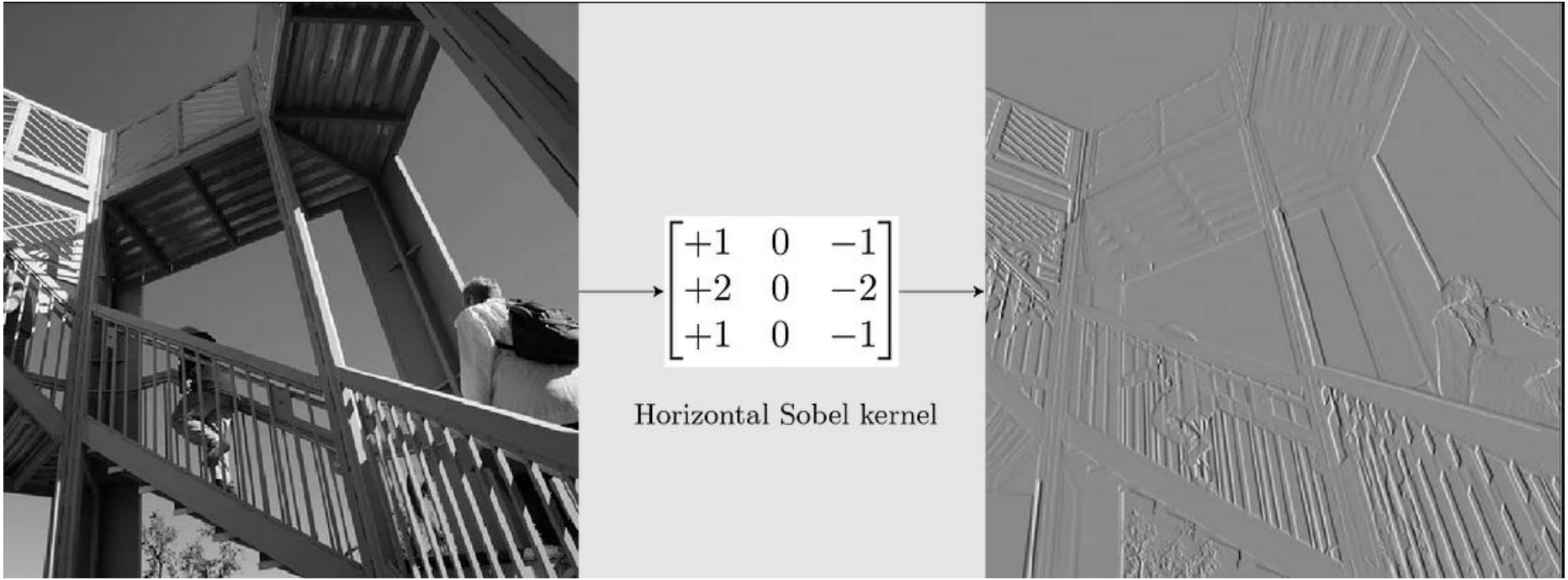


The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?



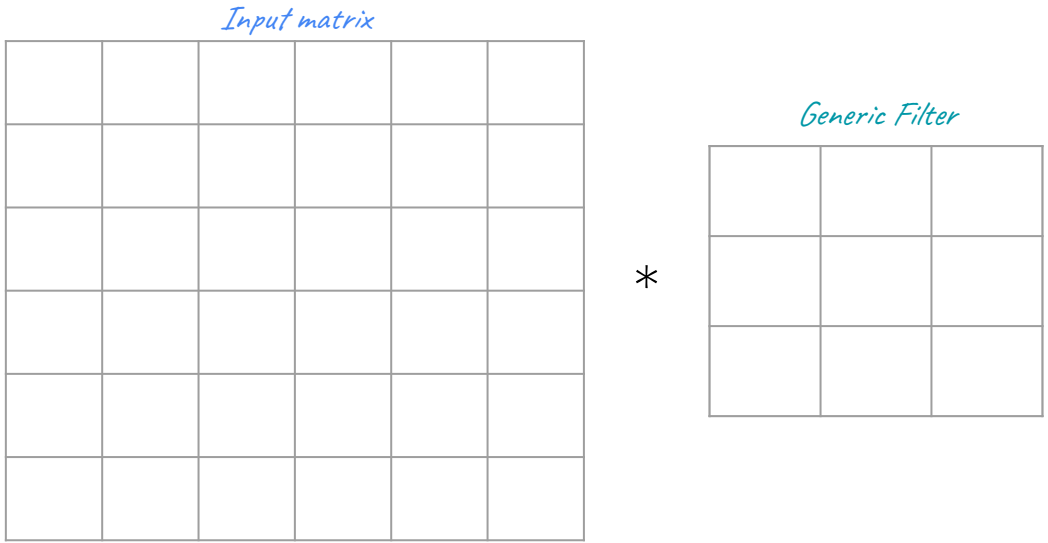
Why convolutions?

The core concept of computer vision is the **convolutional layer**, whose key idea is the use of the **convolutions**.
But, why are **convolutions** useful for structured data such as images? Let's see what convolving a matrix with a *filter* (sometimes also called a *kernel*) actually means.
Why it is called *"vertical edge detector"*?



Padding and strides

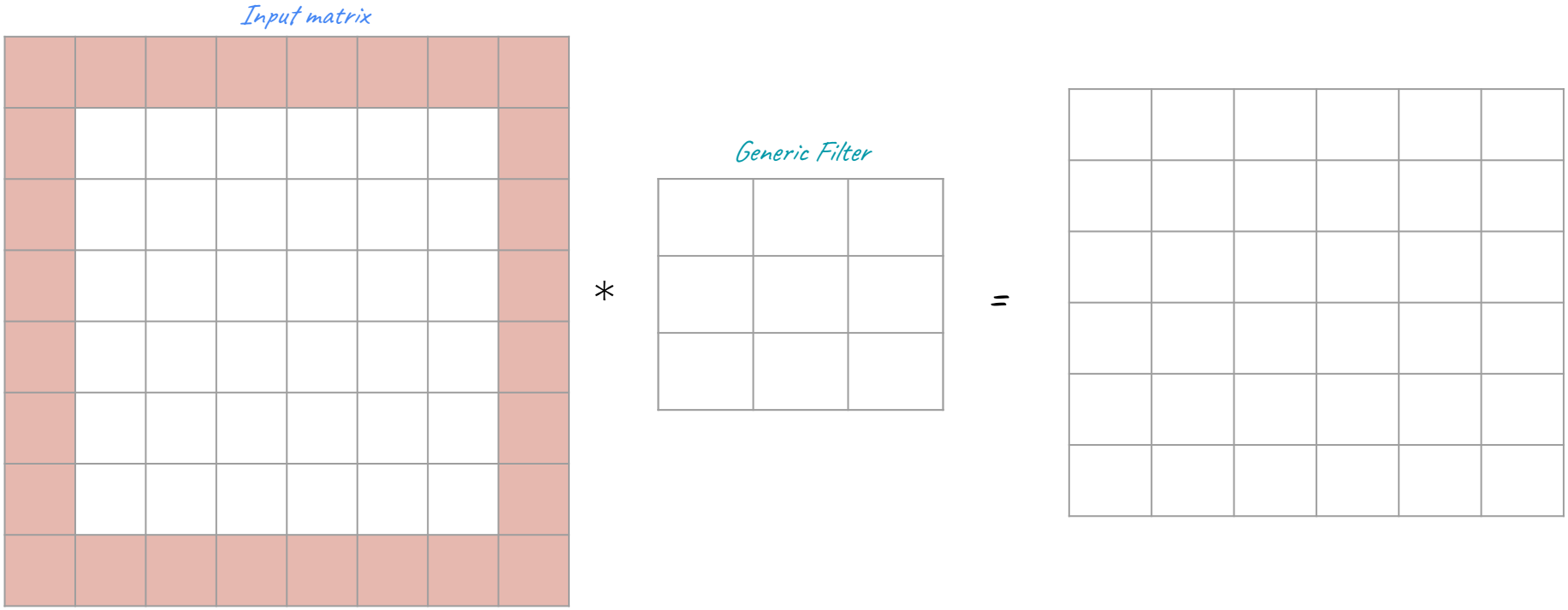
As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.



Padding and strides

As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.

Padding : before applying the convolutional operator, you add an additional border (*p*) around the edges.



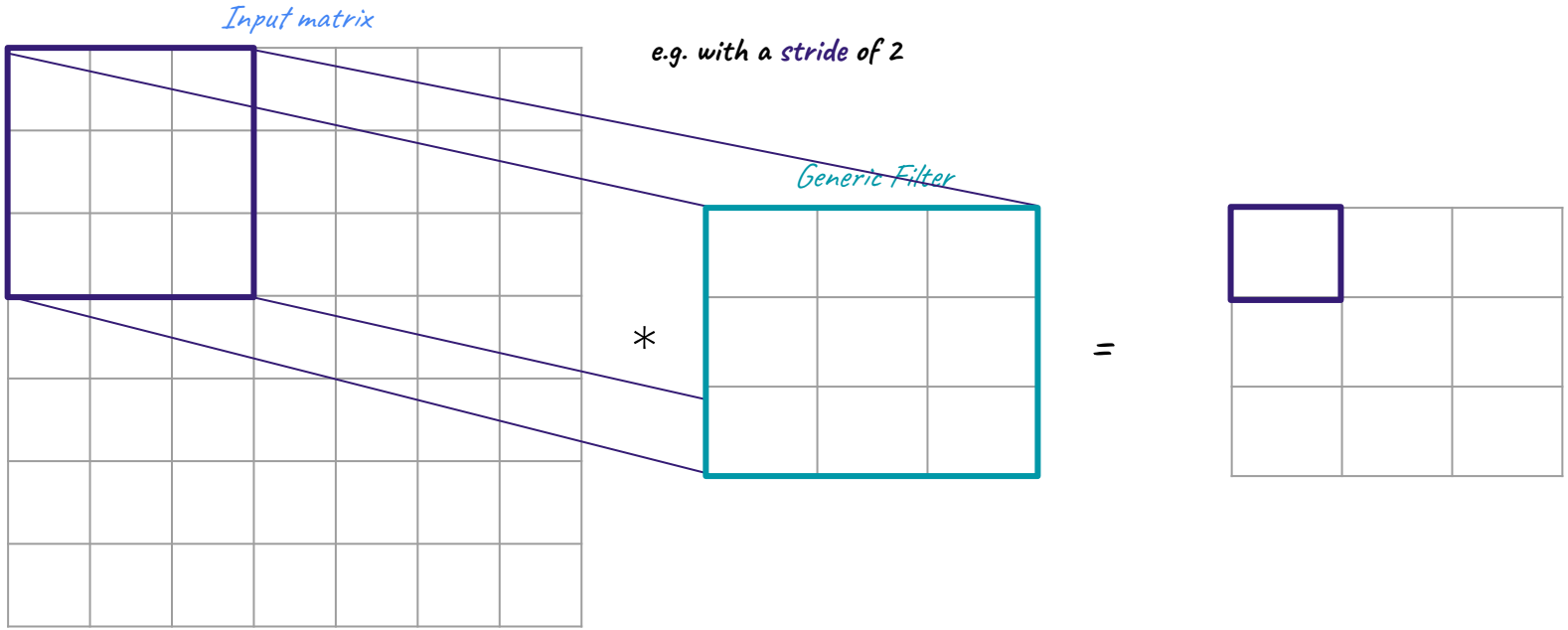
nxn input matrix padded with *p* and convolved with *fxf filter* gives back an *(n+2p-f+1)x(n+2p-f+1)* matrix.

Padding and strides

As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.

Padding : before applying the convolutional operator, you add an additional border (p) around the edges.

Striding : instead of jumping of one row/column per time, you jump *(stride)* a number s of rows and columns per time.



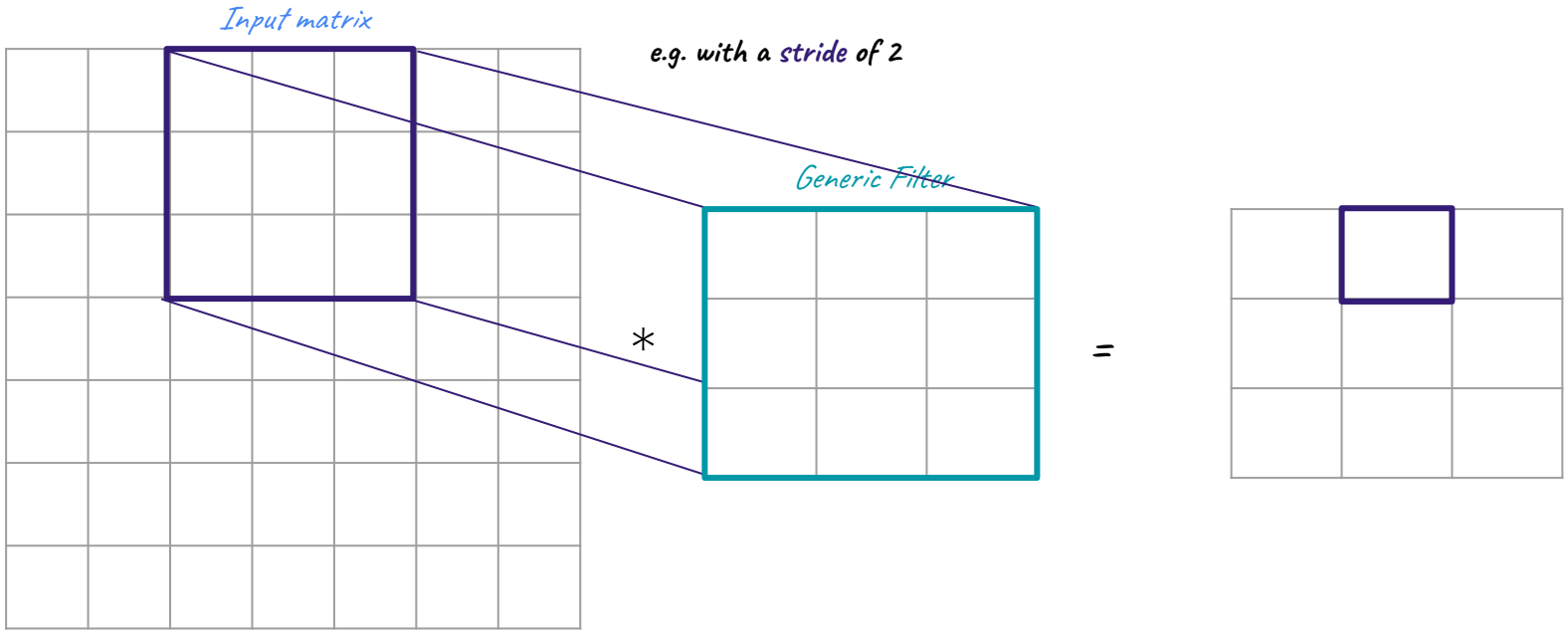
nxn input matrix padded with p and convolved with *fxf filter* with ***stride s*** gives back an $(n+2p-f)/s+1 \times (n+2p-f)/s+1$ matrix.

Padding and strides

As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.

Padding : before applying the convolutional operator, you add an additional border (*p*) around the edges.

Striding : instead of jumping of one row/column per time, you jump *(stride)* a number ***s*** of rows and columns per time.



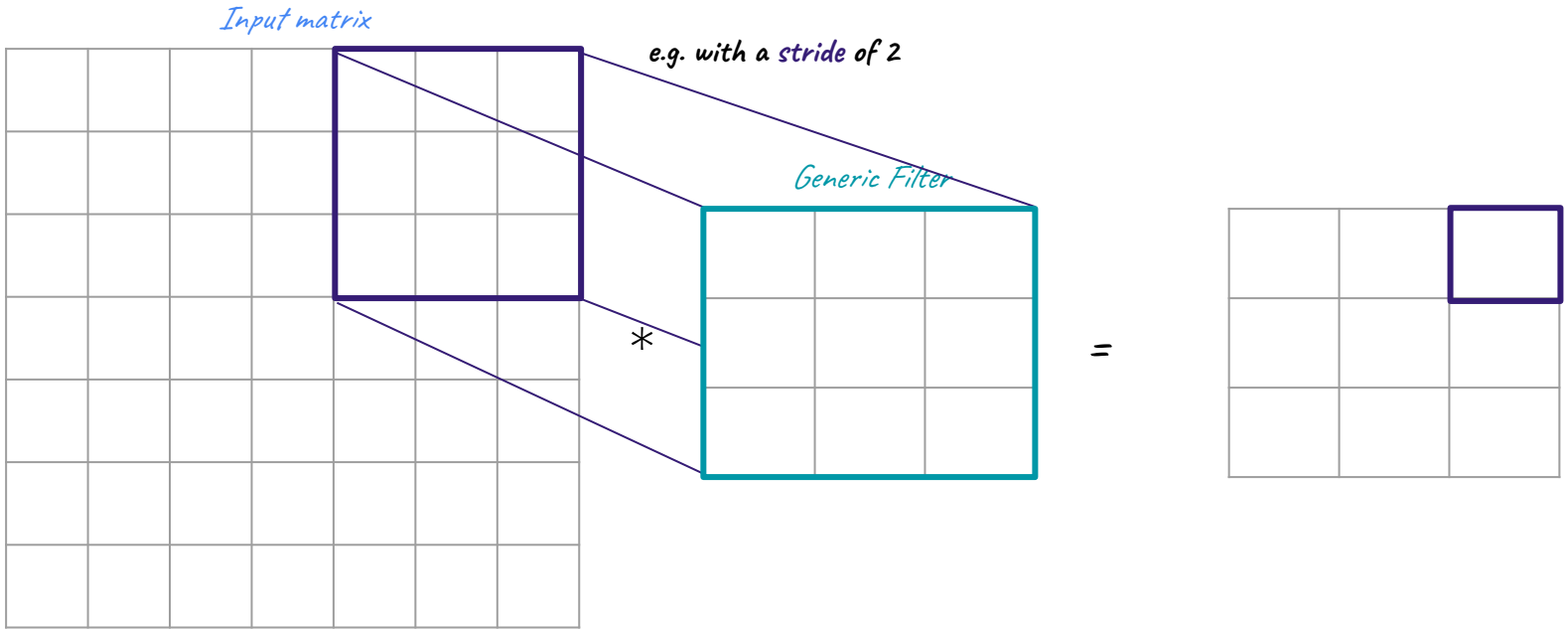
nxn input matrix padded with *p* and convolved with *fxf filter* with ***stride s*** gives back an *(n+2p-f)/s+1 x (n+2p-f)/s+1* matrix.

Padding and strides

As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.

Padding : before applying the convolutional operator, you add an additional border (*p*) around the edges.

Striding : instead of jumping of one row/column per time, you jump *(stride)* a number ***s*** of rows and columns per time.



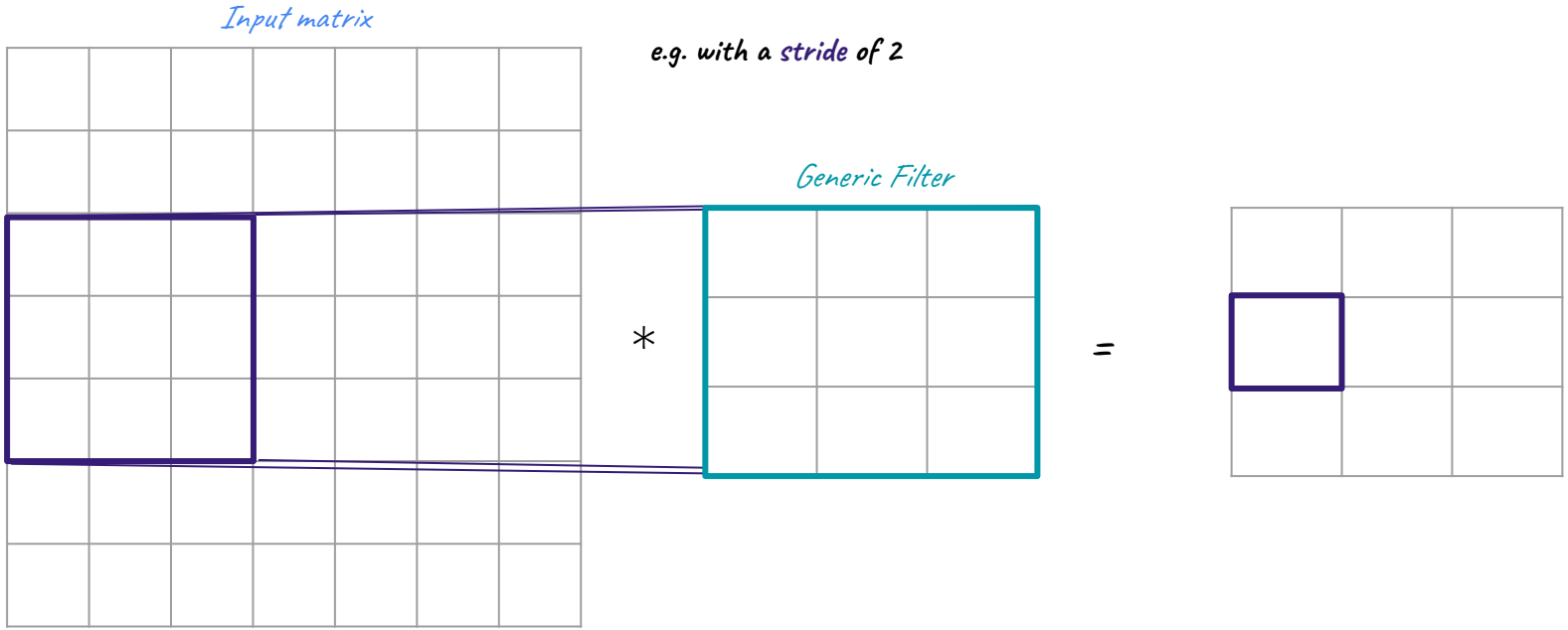
nxn input matrix padded with *p* and convolved with *fxf filter* with ***stride s*** gives back an *(n+2p-f)/s+1 x (n+2p-f)/s+1* matrix.

Padding and strides

As you saw, convolution gives back a smaller version of the original array. In order to build a CNN, a couple of small modifications to the convolution operation should be made, which are *padding* and ***striding***.

Padding : before applying the convolutional operator, you add an additional border (*p*) around the edges.

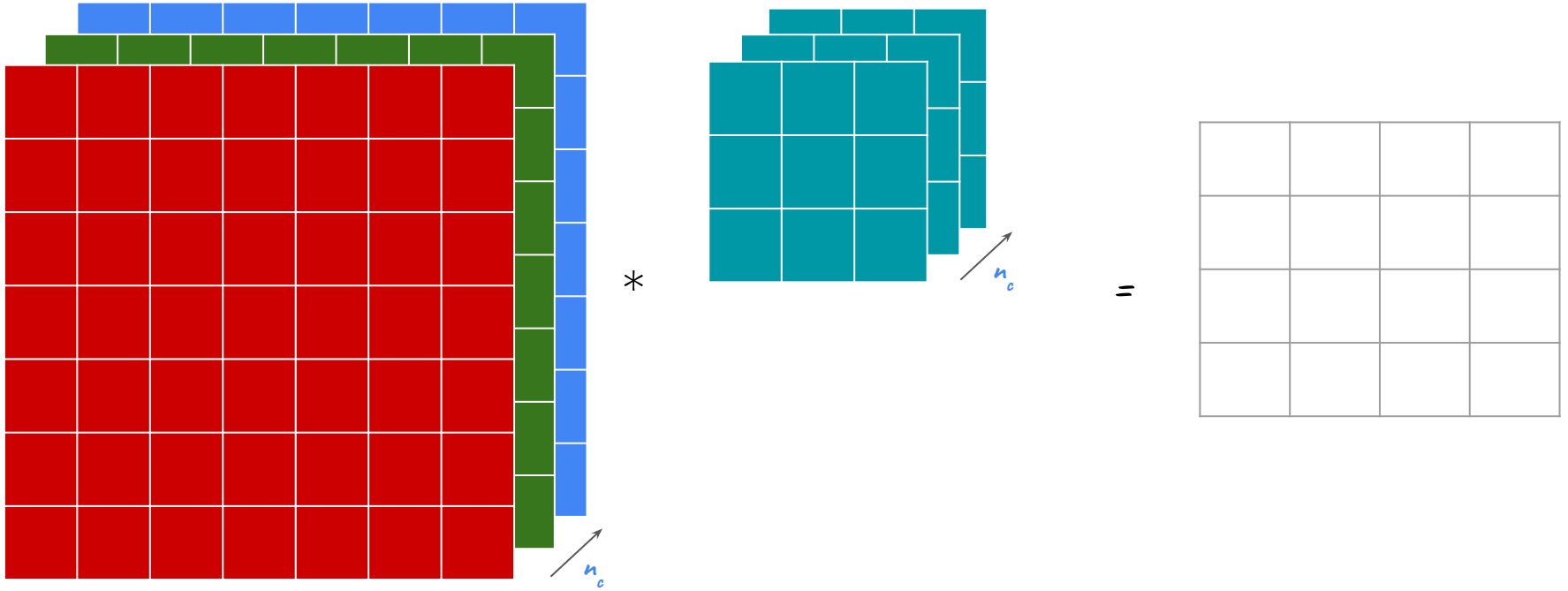
Striding : instead of jumping of one row/column per time, you jump *(stride)* a number ***s*** of rows and columns per time.



nxn input matrix padded with *p* and convolved with *fxf filter* with ***stride s*** gives back an *(n+2p-f)/s+1 x (n+2p-f)/s+1* matrix.

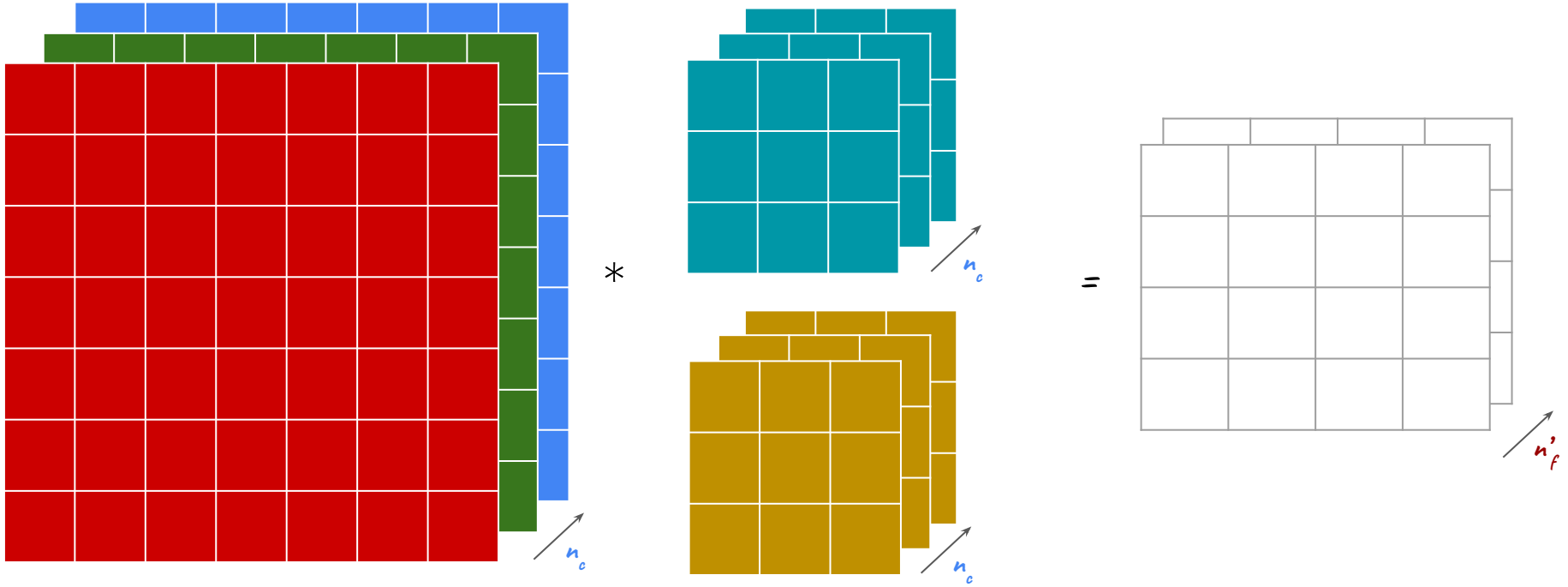
Convolution on actual RGB images

Of course, these concepts can be extended to not just a matrix nxn of numbers, but to a generic n_c-channels cube, such as RGB images.



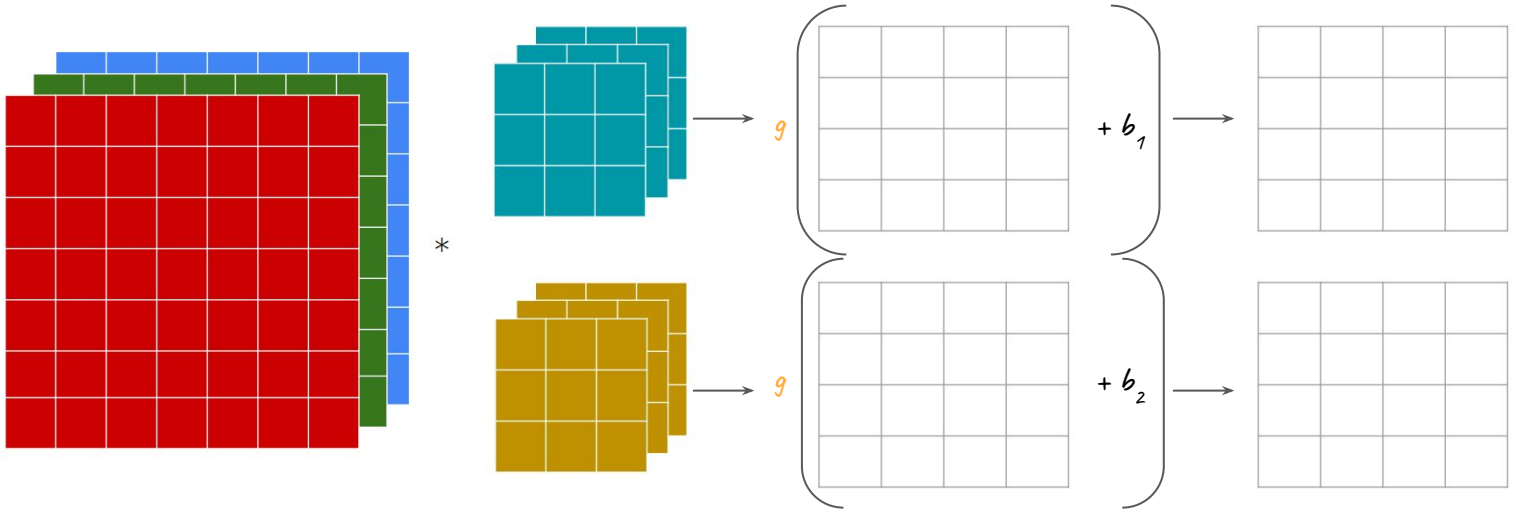
Convolution on actual RGB images

Of course, these concepts can be extended to not just a matrix $n \times n$ of numbers, but to a generic n_c-channels cube, such as RGB images. And you can use multiple filters at the same time (i.e. *vertical edge detector*, *horizontal edge detector*, in general whatever n'_f you want)!

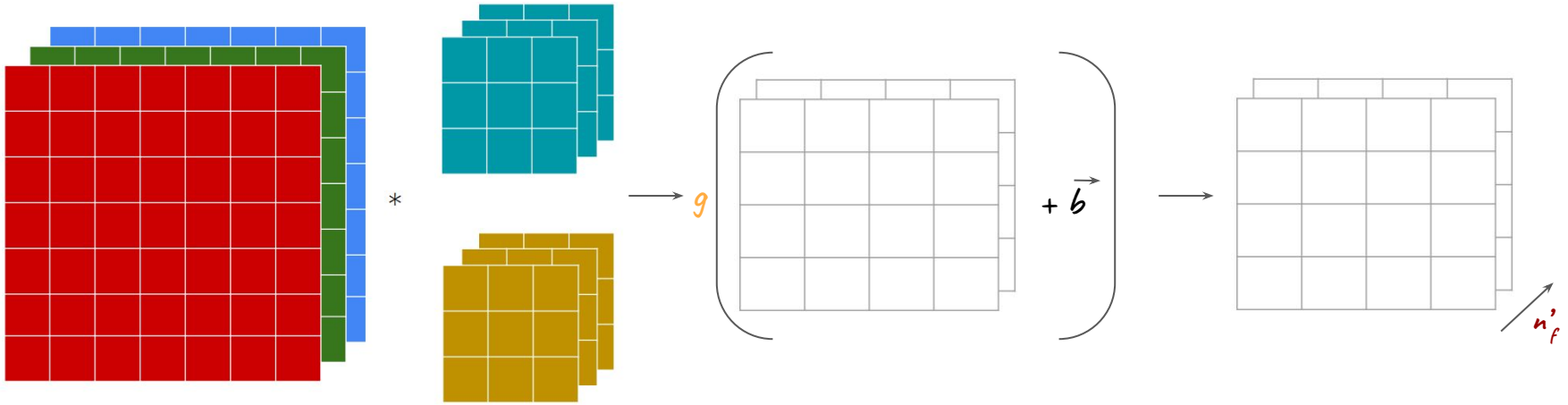


$n \times n \times n_c$ *input matrix* convolved with n'_f *filters* of size $f \times f \times n_c$ gives back an $(n-f+1) \times (n-f+1) \times n'_f$ matrix.

Now we can implement one layer of a *Convolutional Neural Network*.
The only thing to do to turn that convolution into a *convolutional layer* is to add a bias, and apply an *activation function*.



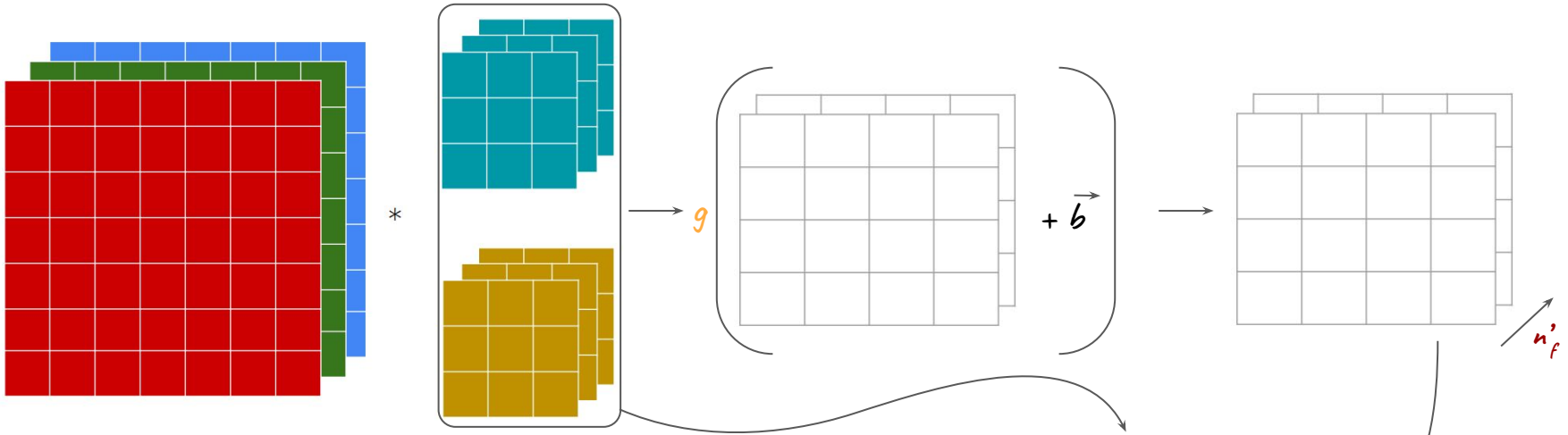
Now we can implement one layer of a *Convolutional Neural Network*.
The only thing to do to turn that convolution into a *convolutional layer* is to add a bias, and apply an *activation function*.



If you remember from the basics of Deep Learning, a typical layer does these two things:

$$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g(Z^{[l]})$$

Now we can implement one layer of a *Convolutional Neural Network*.
The only thing to do to turn that convolution into a *convolutional layer* is to add a bias, and apply an *activation function*.



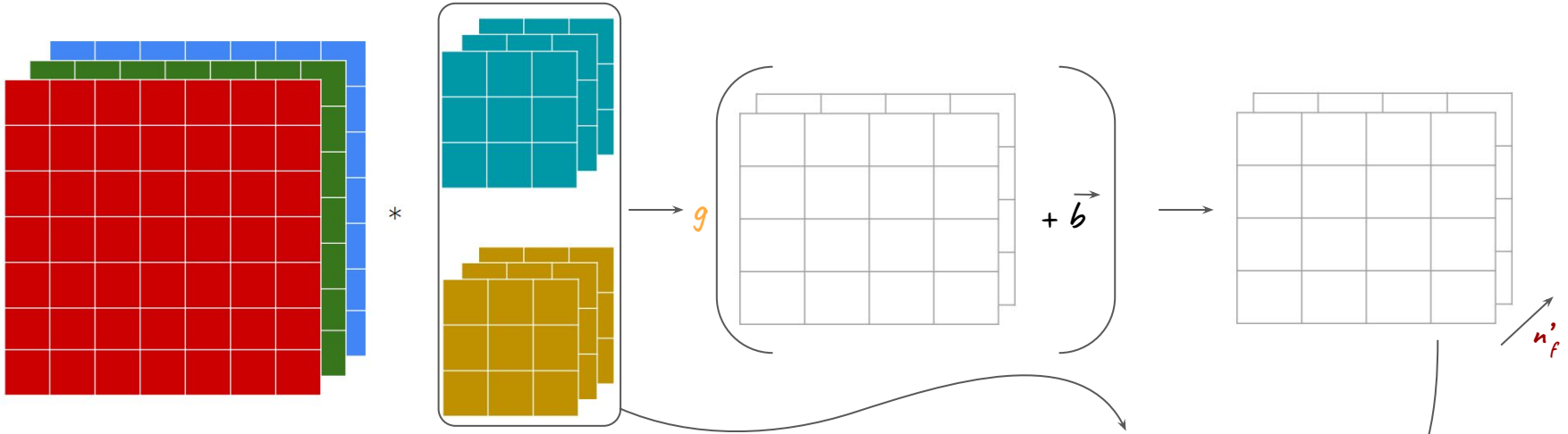
If you remember from the basics of Deep Learning, a typical layer does these two things:

$$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g(Z^{[l]})$$

That's one layer of a *Convolutional Network*. In this example we had two filters, but it is common in ConvNets to have more n'_f than n_c.
Those quickly explode to a great number of parameters, e.g. a single *convolutional layer* with ten **3x3x3** filters has **280** parameters.

However, notice the nice thing that the number of parameters does not depend on the size of the input image.

Now we can implement one layer of a *Convolutional Neural Network*.
The only thing to do to turn that convolution into a *convolutional layer* is to add a bias, and apply an *activation function*.



If layer l is a *convolutional layer*:

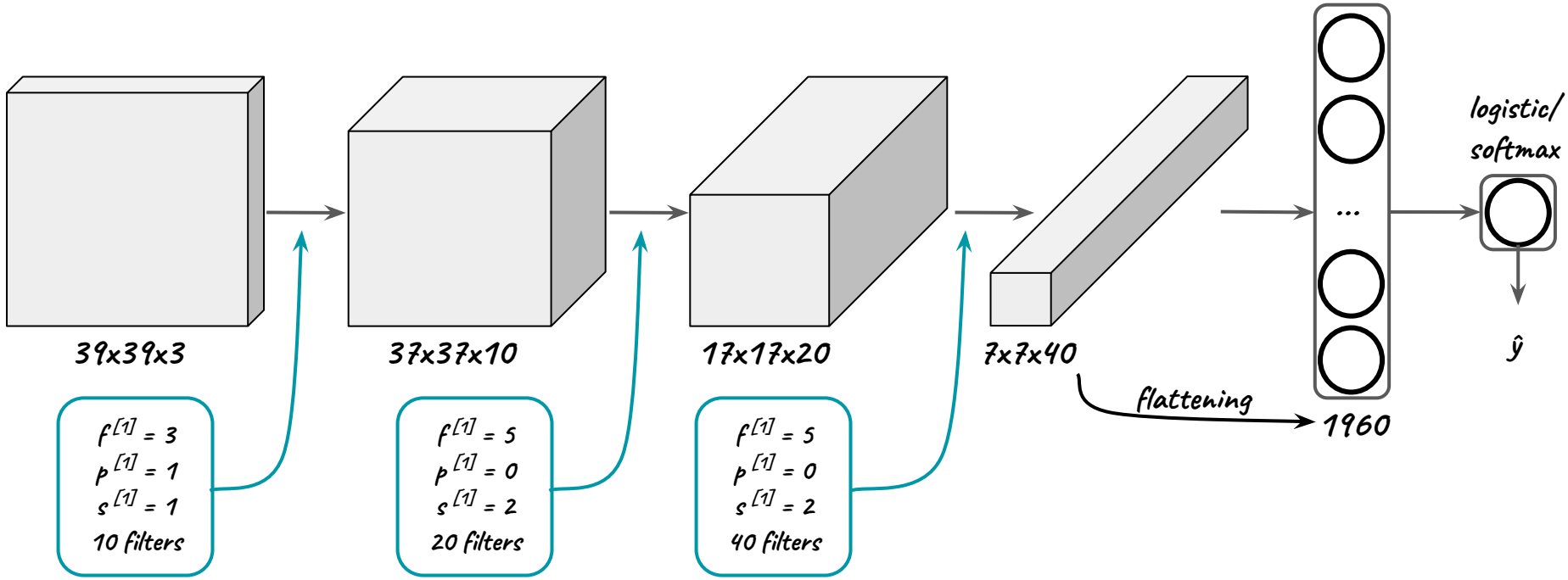
- $f^{[l]}$ is the *filter size*
- $p^{[l]}$ is the *padding*
- $s^{[l]}$ is the *stride*
- $n_c^{[l]}$ the *number of filters*

A *convolutional layer*:

- gets in input an activation: $n_h^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$
- gives back as output: $n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

with $n_h^{[l]} = [(n_h^{[l-1]} + 2p^{[l]} - f^{[l]})/s^{[l]} + 1]$

Let's see an example of a typical *Convolutional Neural Network*.

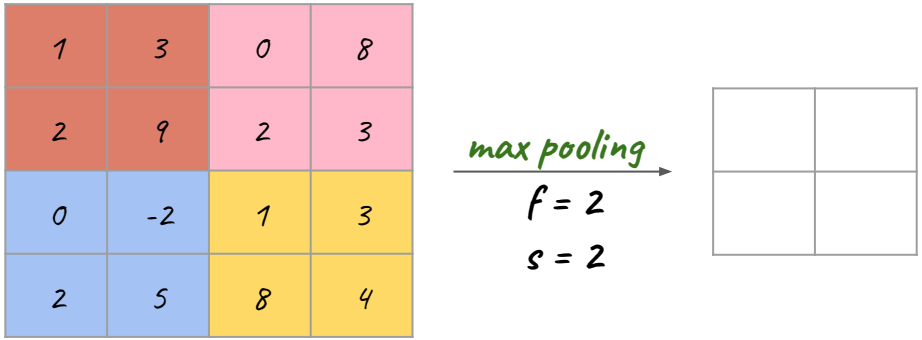


Notice that you start off with larger images, then the deeper you go a *ConvNet*, the height and width will gradually trend down as you go deeper in the network, whereas the number of channels will generally increase. In the end you pass from *convolutional layers* to *dense layers* unraveling what's in the final *convolutional layer*, and get the result from usual output layers as logistic/softmax.

In turns out that there are another typical kind of layer in a *ConvNet*, the *pooling layer*.
Pooling layers simply group together parts of the matrices exiting a *convolutional layer*, and reduce their dimensions by, well, pooling.
Let's see a simple example of *max pooling:*

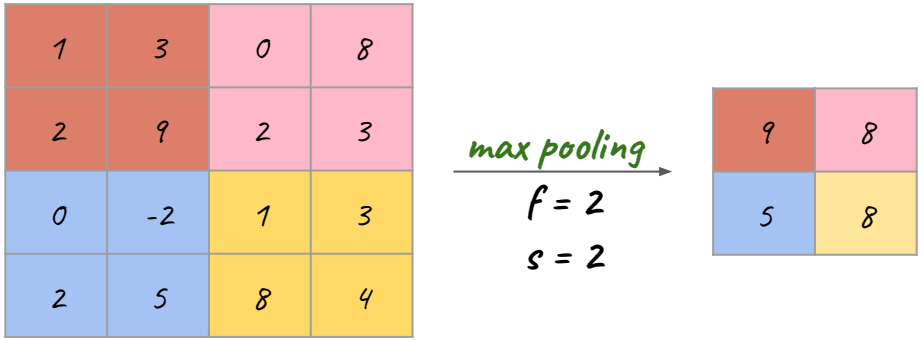
1	3	0	8
2	9	2	3
0	-2	1	3
2	5	8	4

In turns out that there are another typical kind of layer in a *ConvNet*, the *pooling layer*.
Pooling layers simply group together parts of the matrices exiting a *convolutional layer*, and reduce their dimensions by, well, pooling.
Let's see a simple example of *max pooling:*



Pooling layers

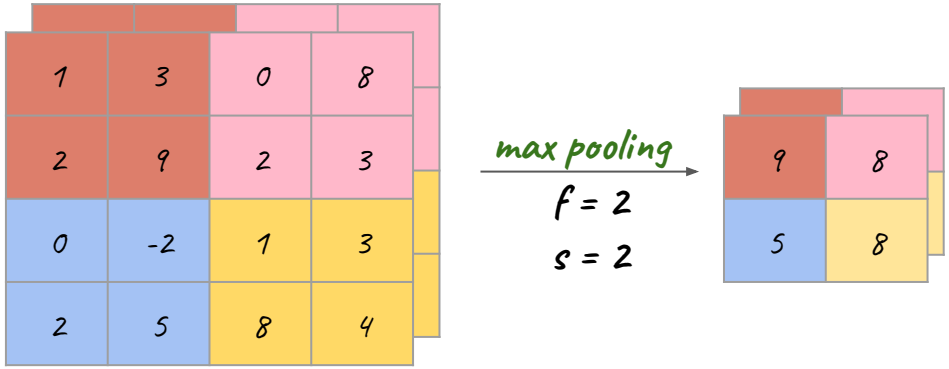
In turns out that there are another typical kind of layer in a *ConvNet*, the *pooling layer*.
Pooling layers simply group together parts of the matrices exiting a *convolutional layer*, and reduce their dimensions by, well, pooling.
Let's see a simple example of *max pooling:*



One interesting thing of *max pooling* is that it has a set of hyperparameters (*size f* and *stride s*), but no parameters to learn. There's actually nothing for gradient descent to learn here, once you fix *f* and *s*, it's just a fixed computation and gradient descent doesn't change anything.

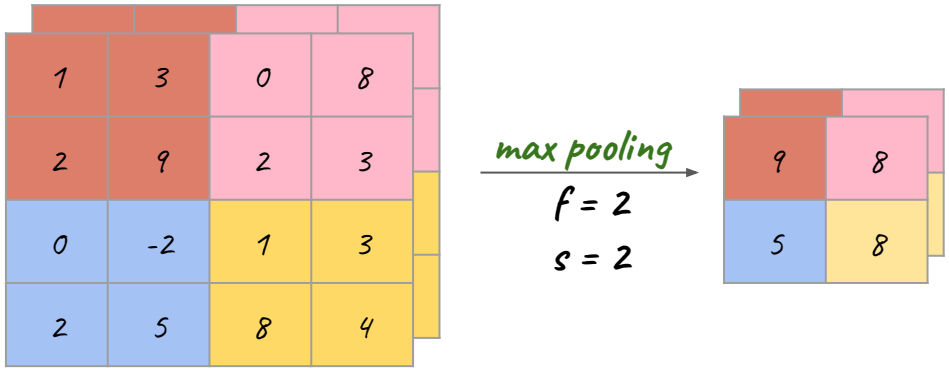
Pooling layers

In turns out that there are another typical kind of layer in a *ConvNet*, the *pooling layer*.
Pooling layers simply group together parts of the matrices exiting a *convolutional layer*, and reduce their dimensions by, well, pooling.
Let's see a simple example of *max pooling:*



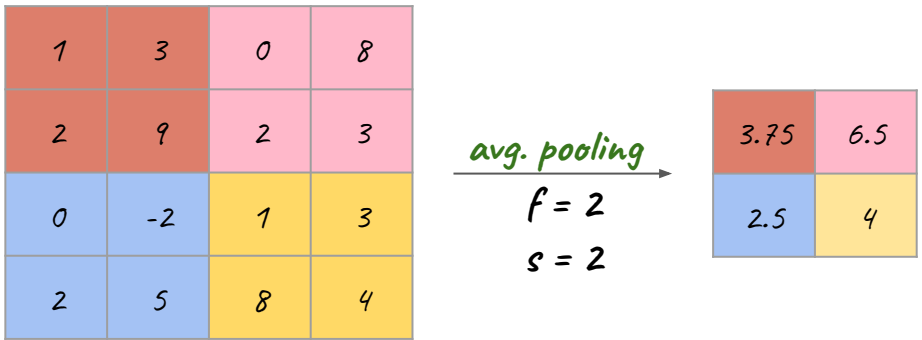
One interesting thing of *max pooling* is that it has a set of hyperparameters (*size f* and *stride s*), but no parameters to learn. There's actually nothing for gradient descent to learn here, once you fix *f* and *s*, it's just a fixed computation and gradient descent doesn't change anything.

In turns out that there are another typical kind of layer in a *ConvNet*, the *pooling layer*.
Pooling layers simply group together parts of the matrices exiting a *convolutional layer*, and reduce their dimensions by, well, pooling.
Let's see a simple example of *max pooling:*

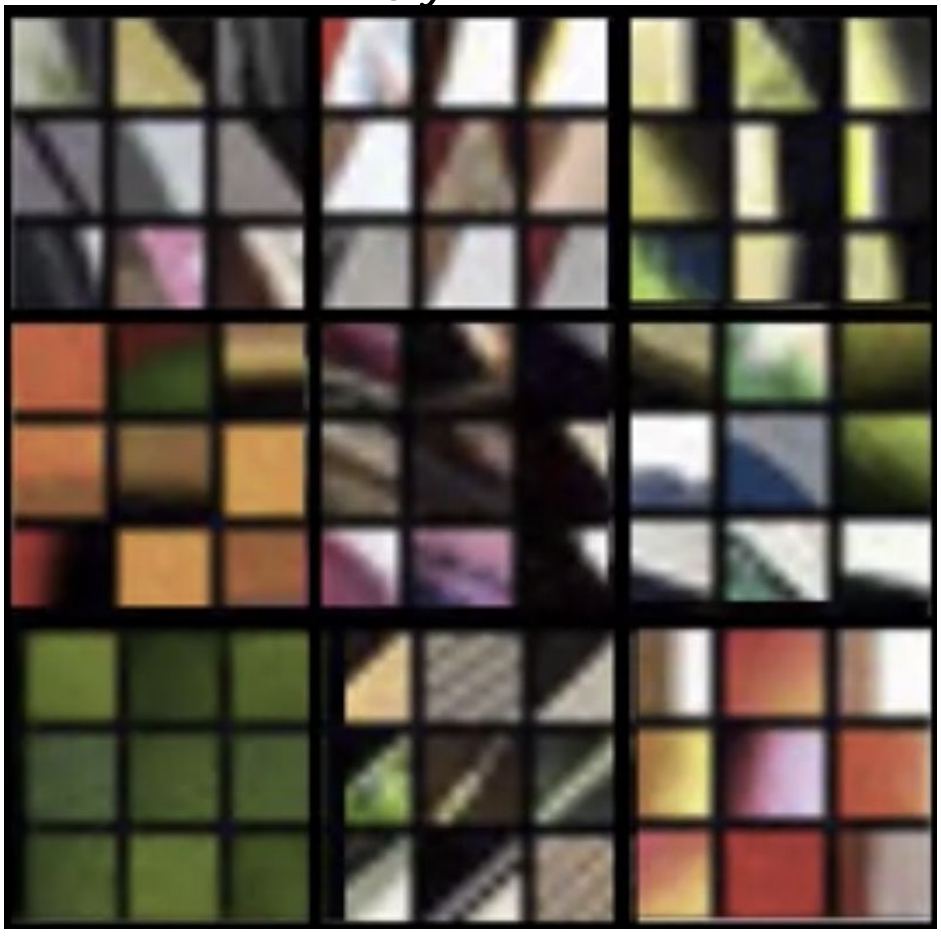


One interesting thing of *max pooling* is that it has a set of hyperparameters (*size f* and *stride s*), but no parameters to learn. There's actually nothing for gradient descent to learn here, once you fix *f* and *s*, it's just a fixed computation and gradient descent doesn't change anything.

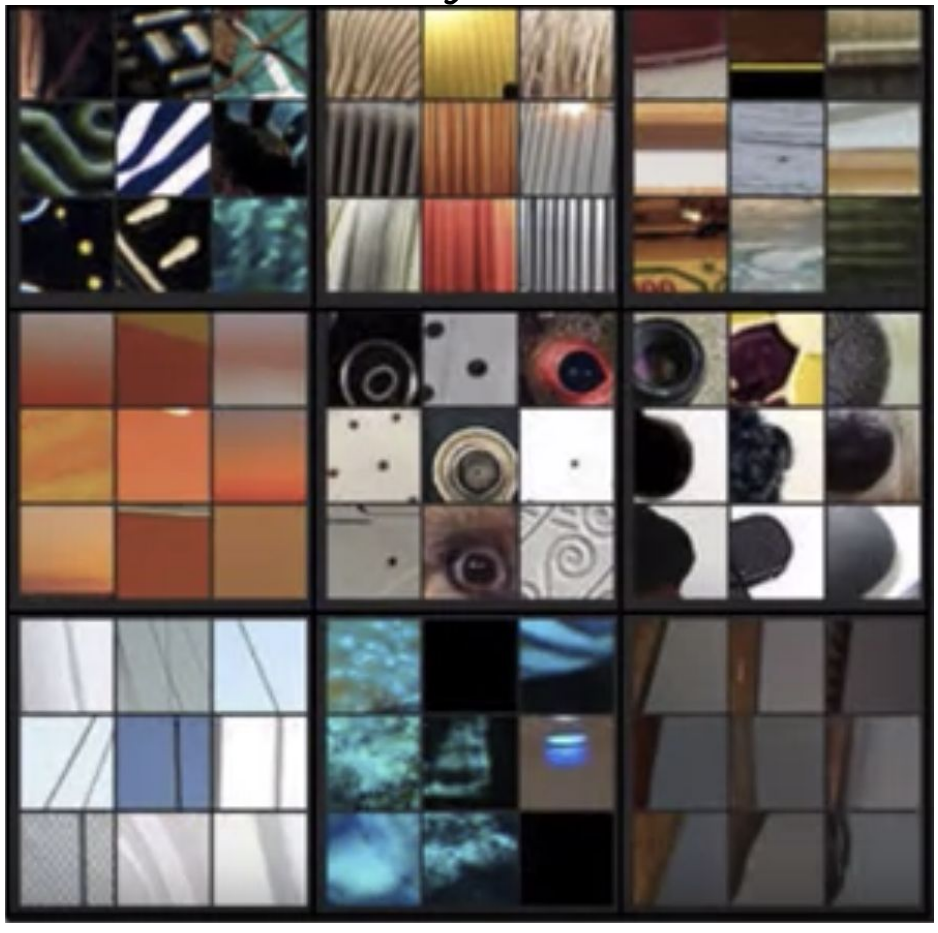
Another kind of pooling is *average pooling*:



Layer 1

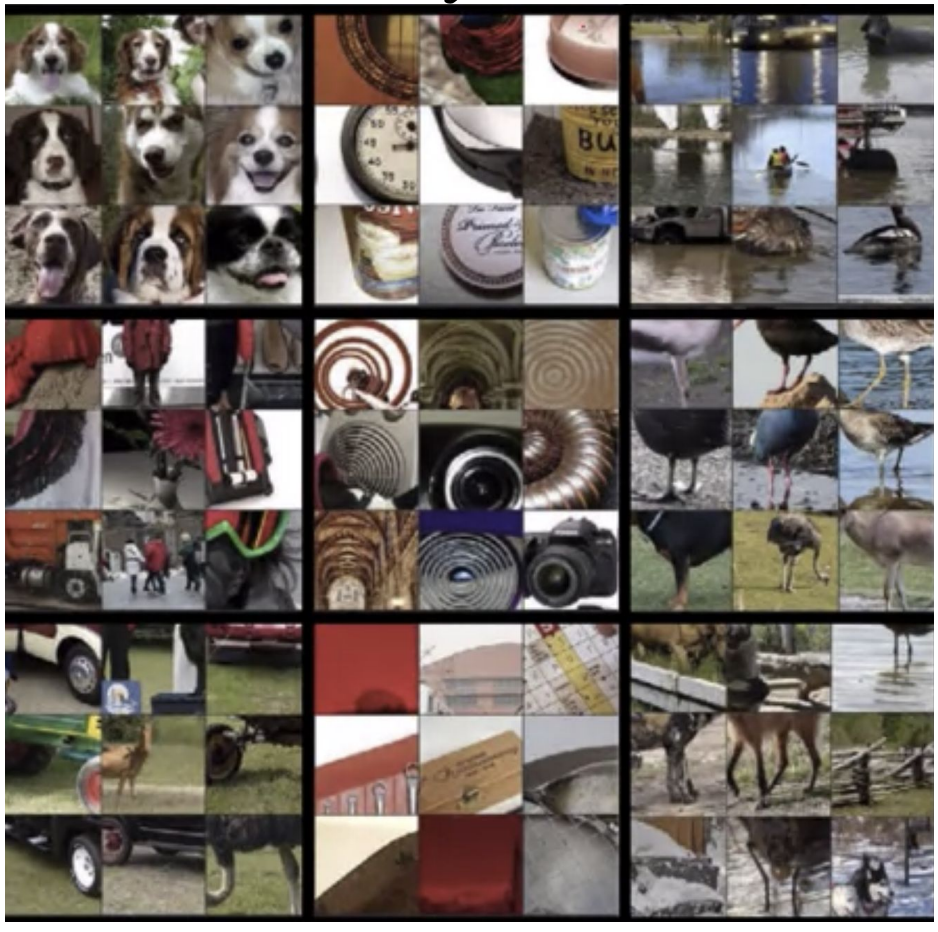


Layer 2

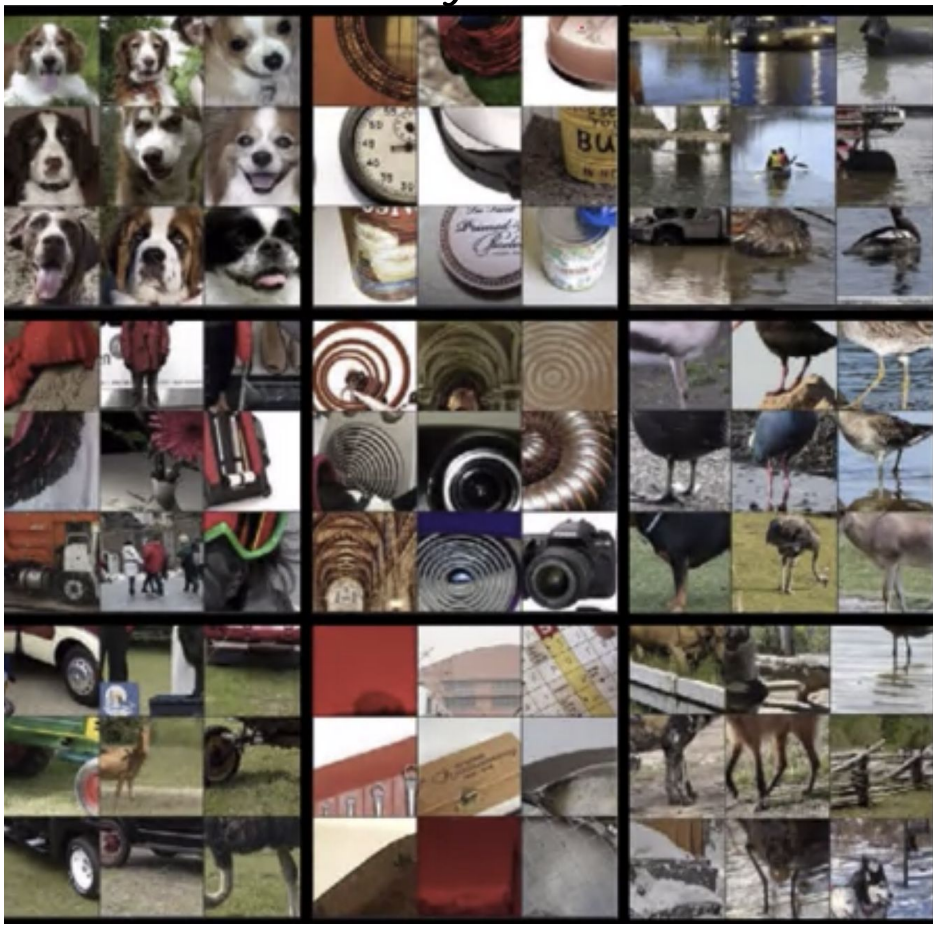


Layer 3

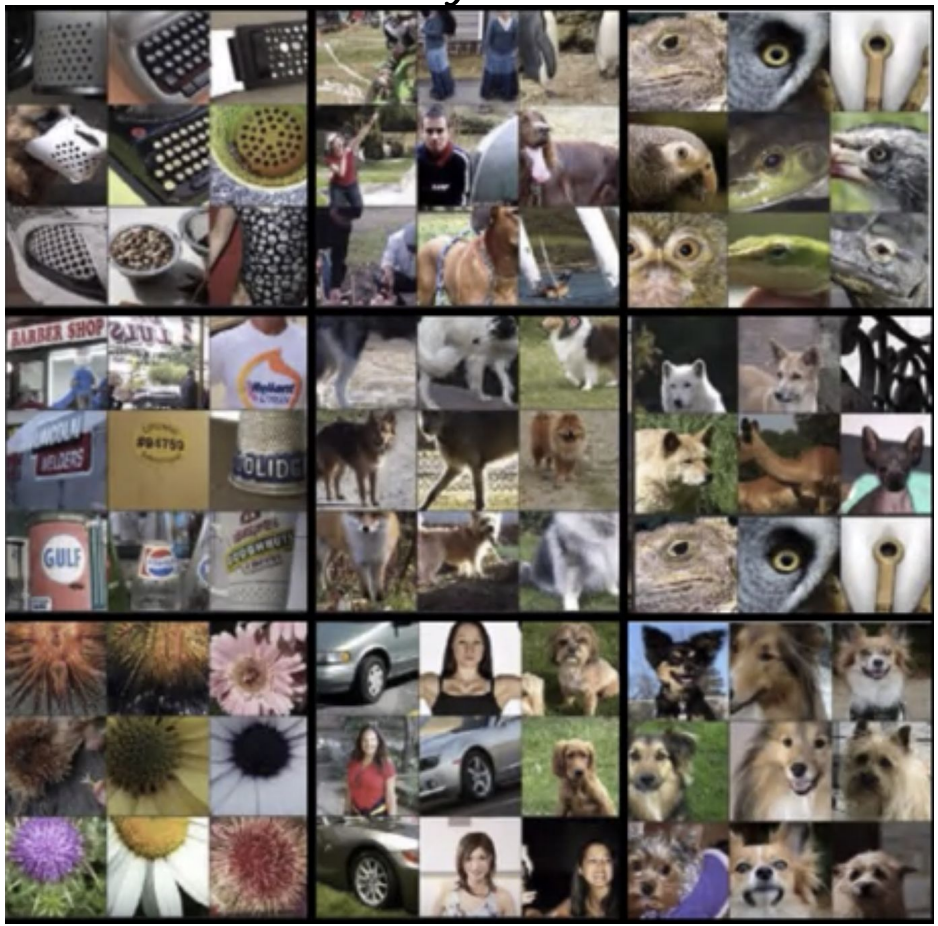
Layer 4



Layer 4



Layer 5

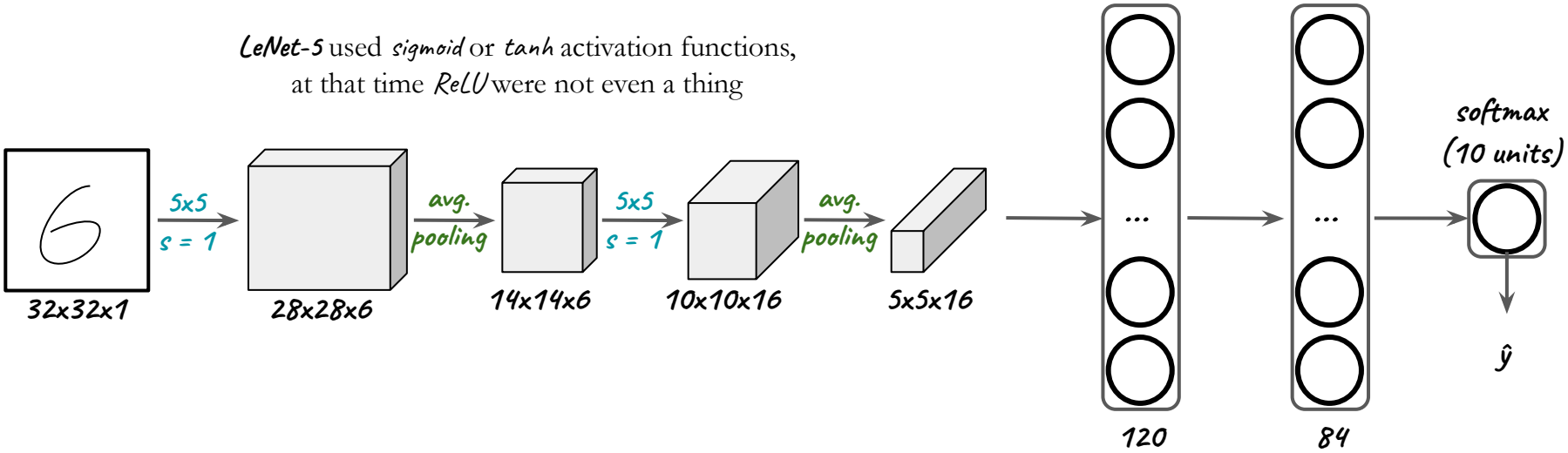


Now we're going to see a **real** *ConvNet*, one with a great historical importance: the ***LeNet-5*** architecture (1998, but the first ***LeNet*** architecture dates back to 1989). ***LeNet-5*** was trained to recognize simple digit images in the MNIST database.

LeCun et al., 1998, "Gradient-based learning applied to document recognition"

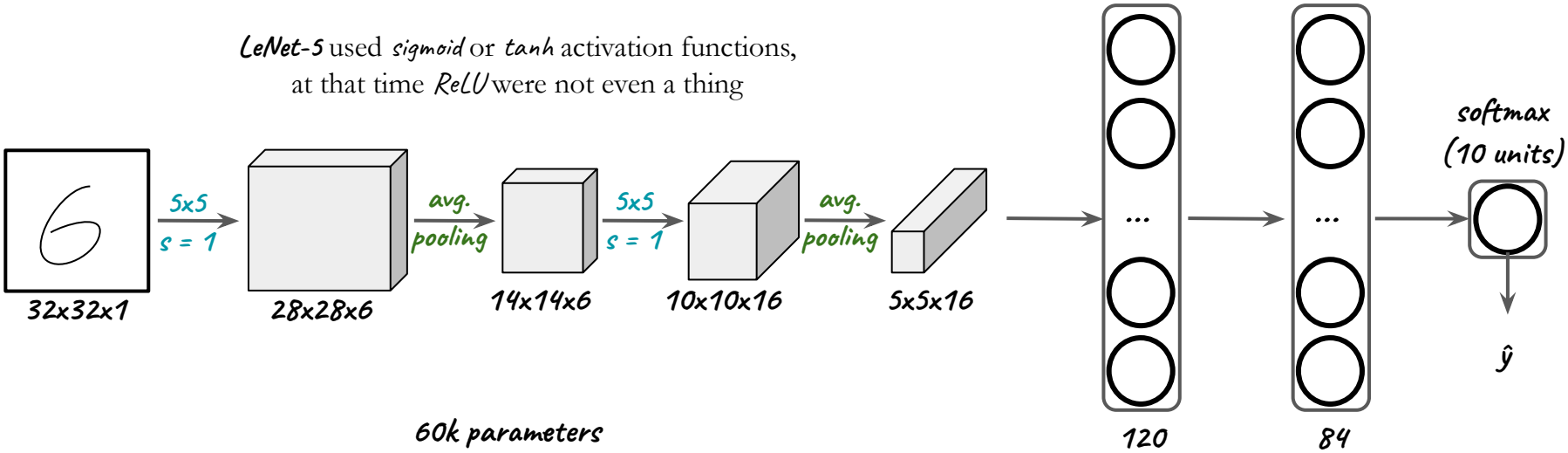
Now we're going to see a **real** *ConvNet*, one with a great historical importance: the *LeNet-5* architecture (1998, but the first *LeNet* architecture dates back to 1989). *LeNet-5* was trained to recognize simple digit images in the MNIST database.

LeNet-5 used *sigmoid* or *tanh* activation functions,
at that time *ReLU* were not even a thing



LeCun et al., 1998, "Gradient-based learning applied to document recognition"

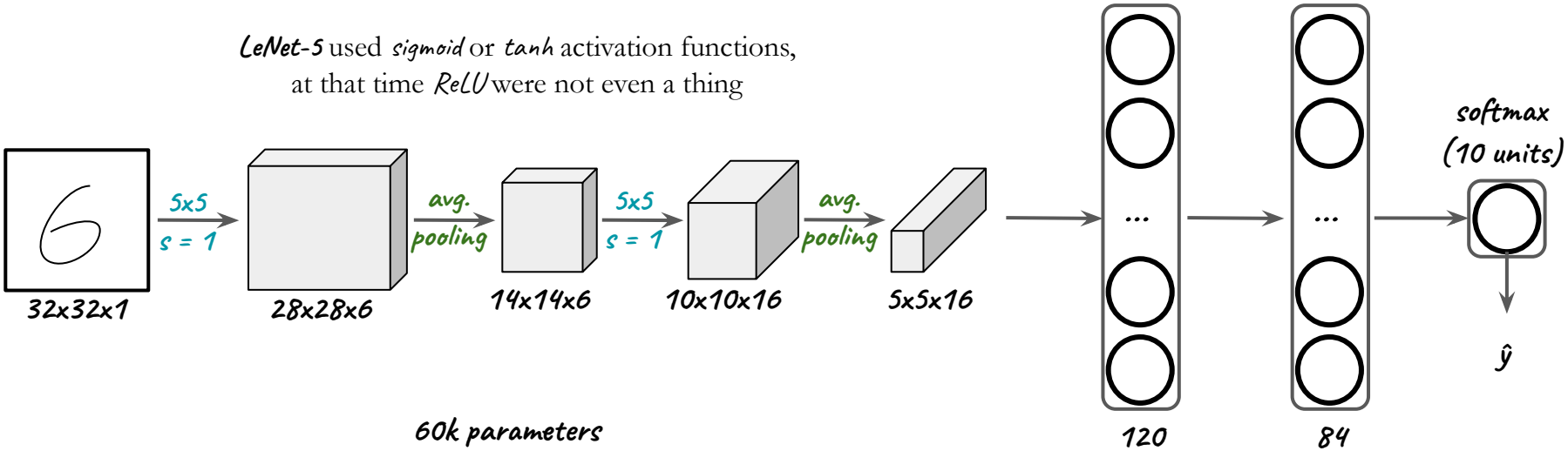
Now we're going to see a **real** *ConvNet*, one with a great historical importance: the *LeNet-5* architecture (1998, but the first *LeNet* architecture dates back to 1989). *LeNet-5* was trained to recognize simple digit images in the MNIST database.



LeCun et al., 1998, "Gradient-based learning applied to document recognition"

Now we're going to see a **real** *ConvNet*, one with a great historical importance: the *LeNet-5* architecture (1998, but the first *LeNet* architecture dates back to 1989). *LeNet-5* was trained to recognize simple digit images in the MNIST database.

LeNet-5 used *sigmoid* or *tanh* activation functions,
at that time *ReLU* were not even a thing



60k parameters

At that time, no GPUs were available, and training this small network required huge computational efforts, and lots of tricks to actually do the convolutions.

… and all for something that - at that time - was easily outperformed by a simple Support Vector Machine ¯_(ツ)_/¯

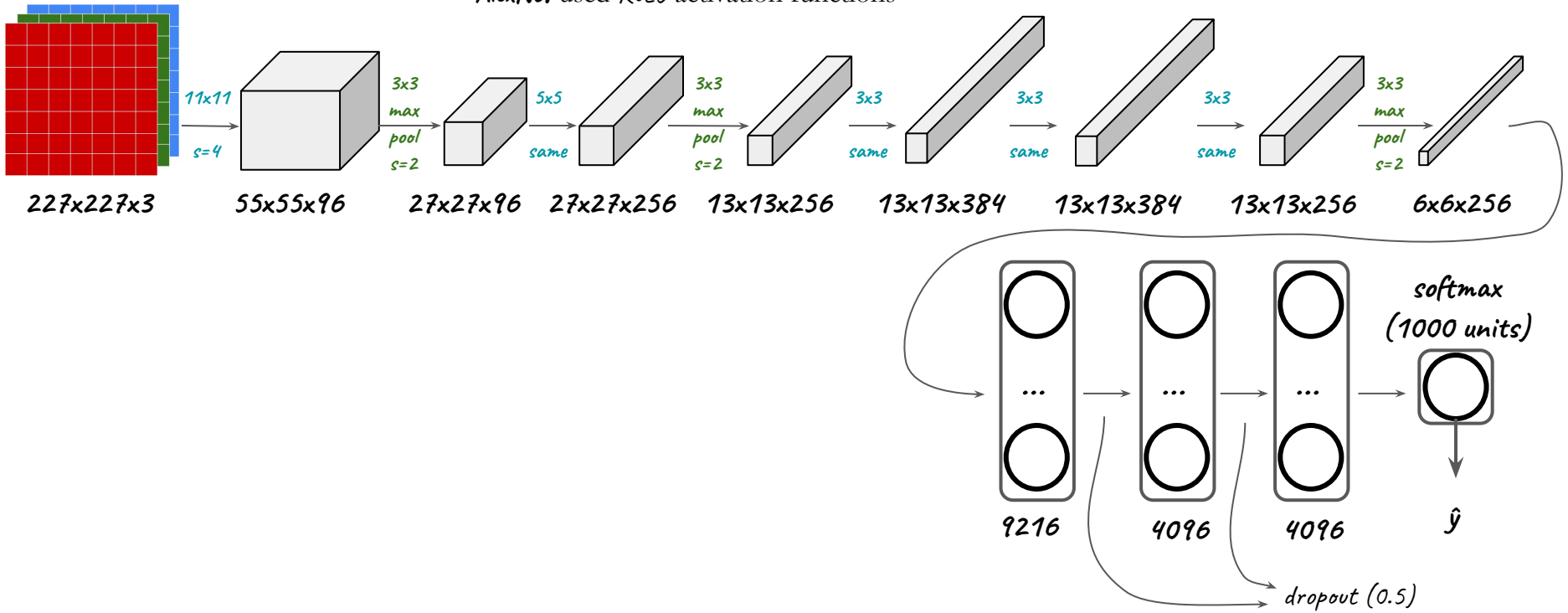
LeCun et al., 1998, "Gradient-based learning applied to document recognition"

The other fundamental *ConvNet* is *AlexNet* (2012), that proved how GPUs-trained deep-CNNs could easily outperform whatever other kind of image classification algorithm was available in the market. *AlexNet* was trained on the ImageNet dataset, with 1000 possible classes.

Krizhevsky et al., 2012, "ImageNet classification with deep convolutional Neural Networks"

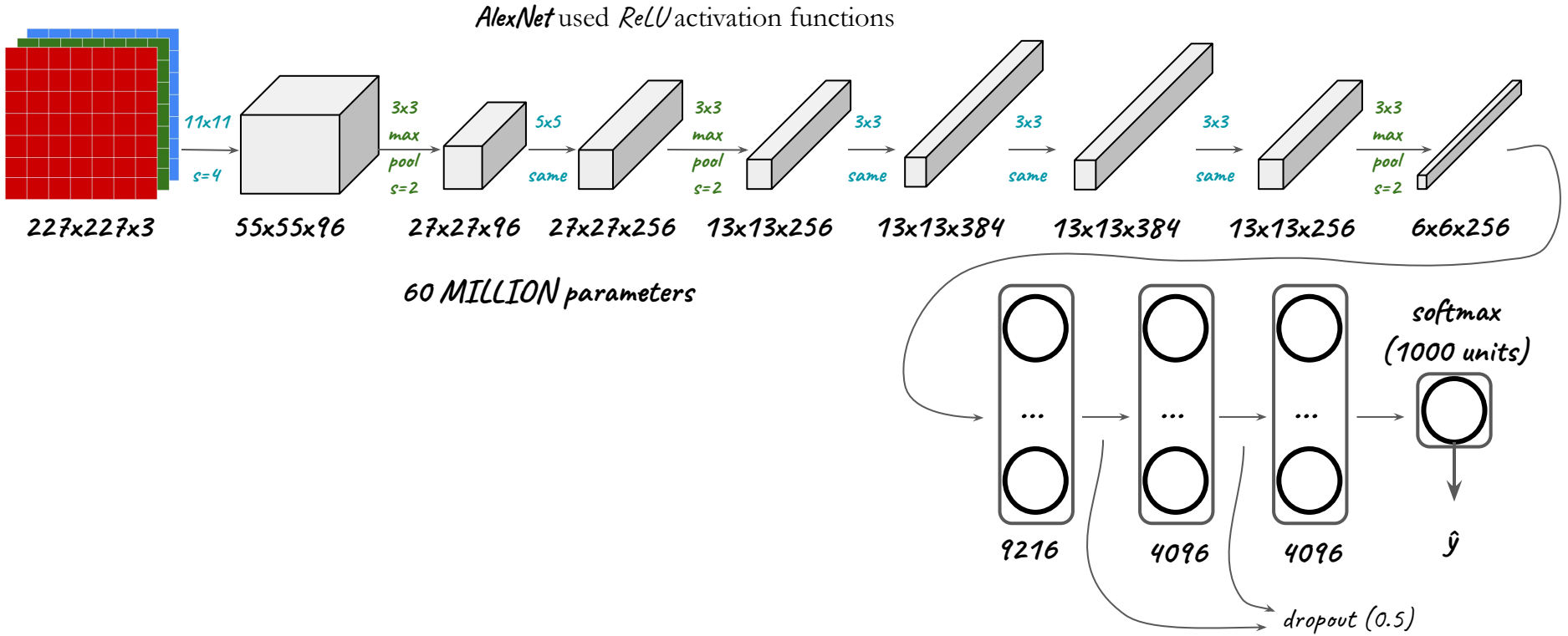
The other fundamental ConvNet is AlexNet (2012), that proved how GPUs-trained deep-CNNs could easily outperform whatever other kind of image classification algorithm was available in the market. AlexNet was trained on the ImageNet dataset, with 1000 possible classes.

AlexNet used ReLU activation functions



Krizhevsky et al., 2012, "ImageNet classification with deep convolutional Neural Networks"

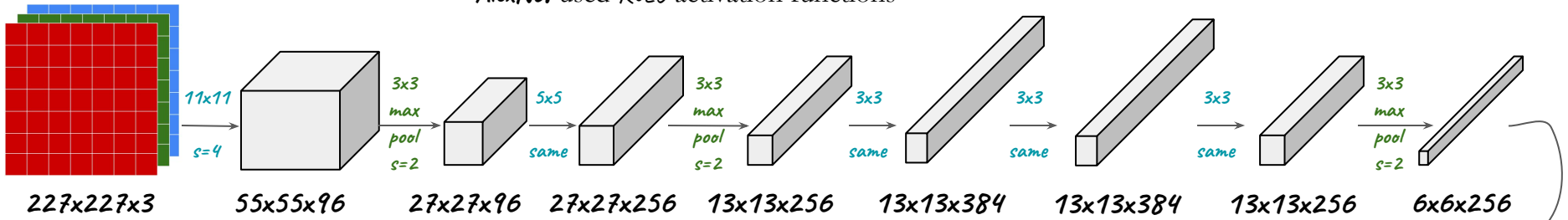
The other fundamental ConvNet is AlexNet (2012), that proved how GPUs-trained deep-CNNs could easily outperform whatever other kind of image classification algorithm was available in the market. AlexNet was trained on the ImageNet dataset, with 1000 possible classes.



Krizhevsky et al., 2012, "ImageNet classification with deep convolutional Neural Networks"

The other fundamental *ConvNet* is ***AlexNet*** (2012), that proved how GPUs-trained deep-CNNs could easily outperform whatever other kind of image classification algorithm was available in the market. ***AlexNet*** was trained on the ImageNet dataset, with 1000 possible classes.

AlexNet used *ReLU* activation functions

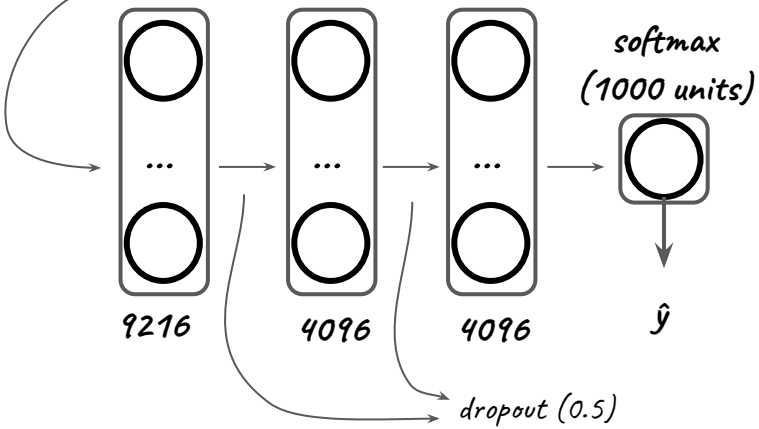


60 MILLION parameters

In 2012 GPUs were a reality, but a single GPU was still a bit slower than a typical CPU node, so it trained on two parallel GPUs with lots of technicalities on the messaging interface between the two that today we might consider obvious but 10 years ago were still in the making.

It required **SIX DAYS** of training.

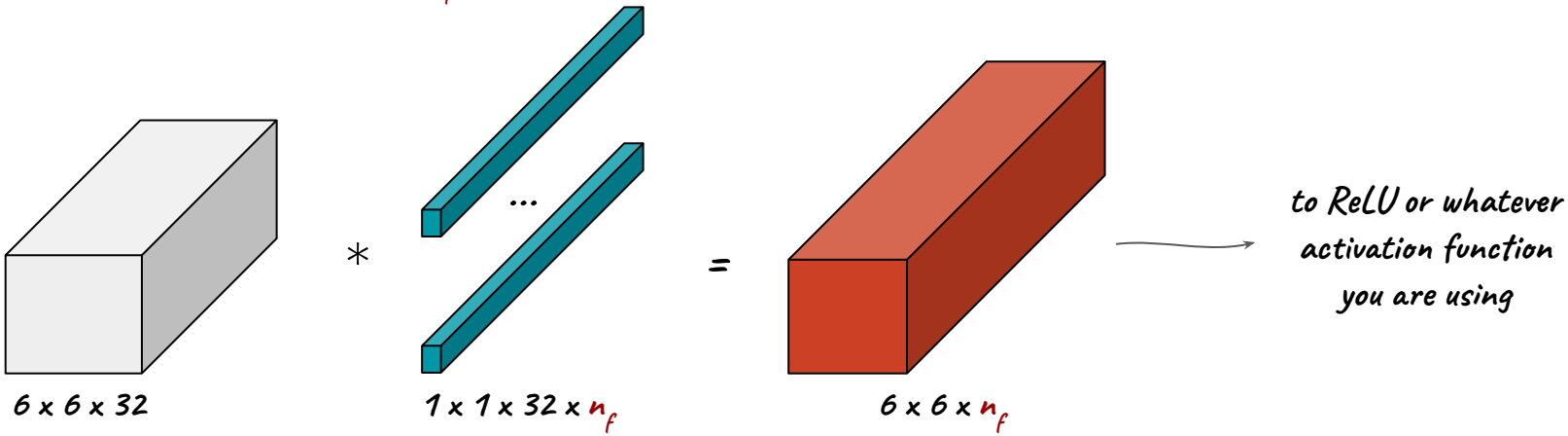
AlexNet also had another kind of layer, the ***local response normalization***, which is not used much since in the end it does not impact that much performances.



Krizhevsky et al., 2012, "ImageNet classification with deep convolutional Neural Networks"

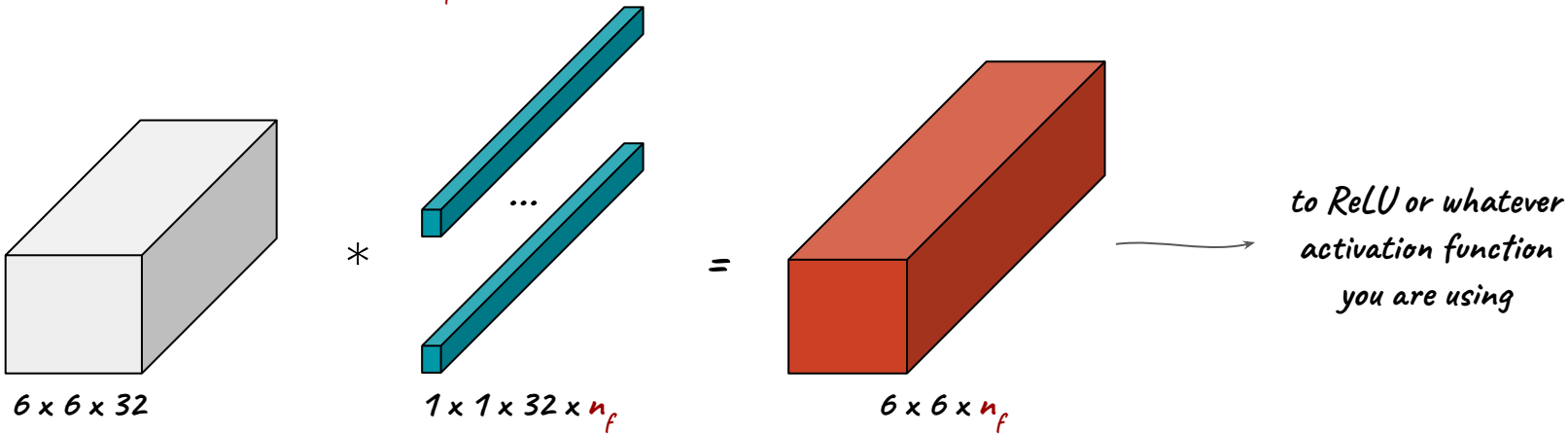
The idea behind the *inception module* is to *simultaneously* do convolutions with different filters **and** pooling.

The idea behind the **inception module** is to *simultaneously* do convolutions with different filters **and** pooling.
The core concept is the one of **1x1 convolutions** (or **Network in Network**), which in a 2D case is a simple multiplication, but in 3D on multiple filters n_f returns a convolved datacube of $n \times n \times n_f$ size.

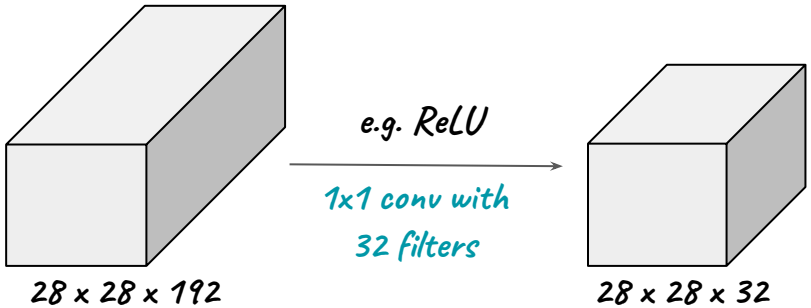


Lin et al., 2013, "Network in Network"

The idea behind the **inception module** is to *simultaneously* do convolutions with different filters **and** pooling.
The core concept is the one of **1x1 convolutions** (or **Network in Network**), which in a 2D case is a simple multiplication, but in 3D on multiple filters n_f returns a convolved datacube of $n \times n \times n_f$ size.



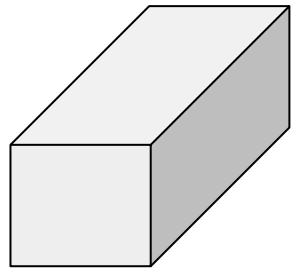
While **pooling** is useful to reduce the width and height of an image, **1x1 convolutions** are used to shrink the number of channels, thus saving ***lots*** of computational time.



Lin et al., 2013, "Network in Network"

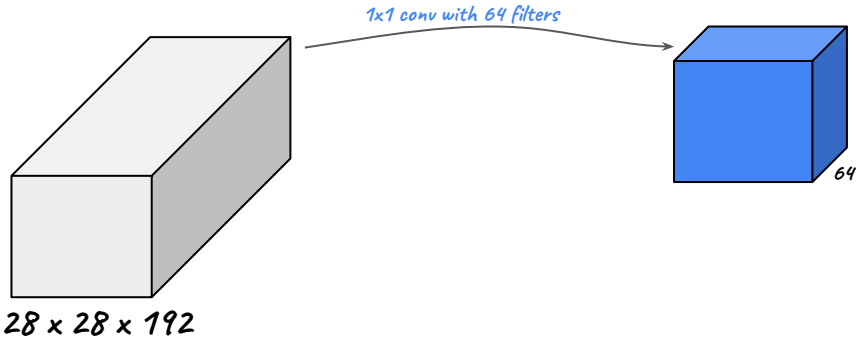
The Inception Module

The idea behind the *inception module* is to *simultaneously* do convolutions with different filters **and** pooling.



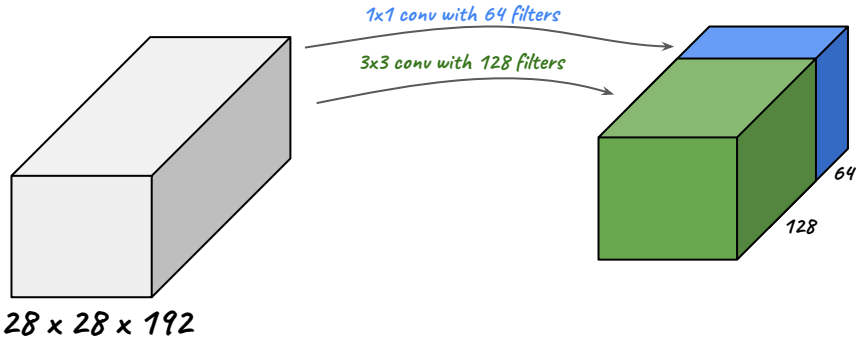
Szegedi et al., 2014,
"Going deeper with convolutions"

The idea behind the *inception module* is to *simultaneously* do convolutions with different filters **and** pooling.



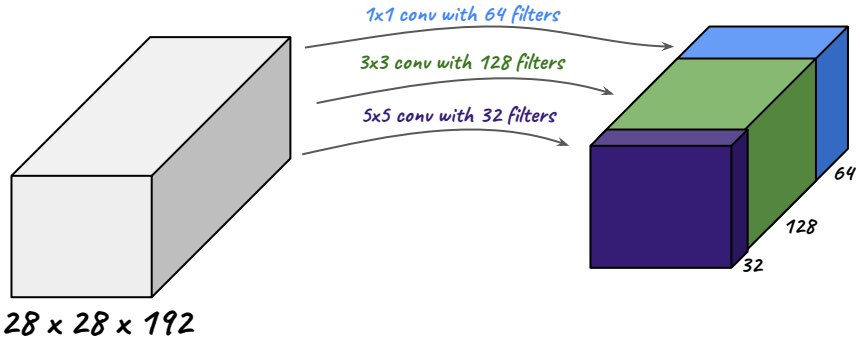
Szegedi et al., 2014,
"Going deeper with convolutions"

The idea behind the **inception module** is to *simultaneously* do convolutions with different filters **and** pooling.



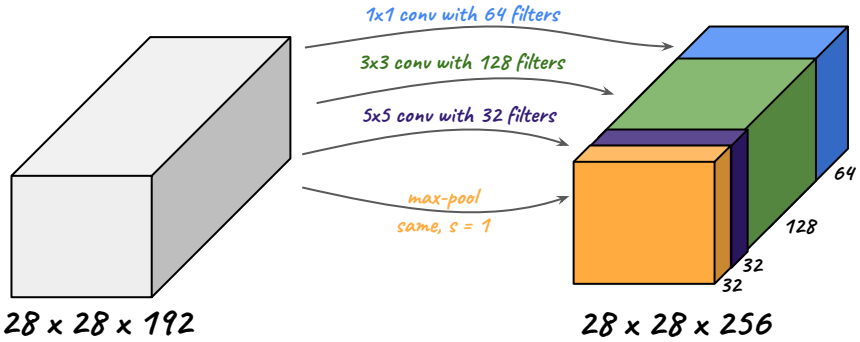
Szegedi et al., 2014,
"Going deeper with convolutions"

The idea behind the *inception module* is to *simultaneously* do convolutions with different filters **and** pooling.



Szegedi et al., 2014,
"Going deeper with convolutions"

The idea behind the **inception module** is to *simultaneously* do convolutions with different filters **and** pooling.

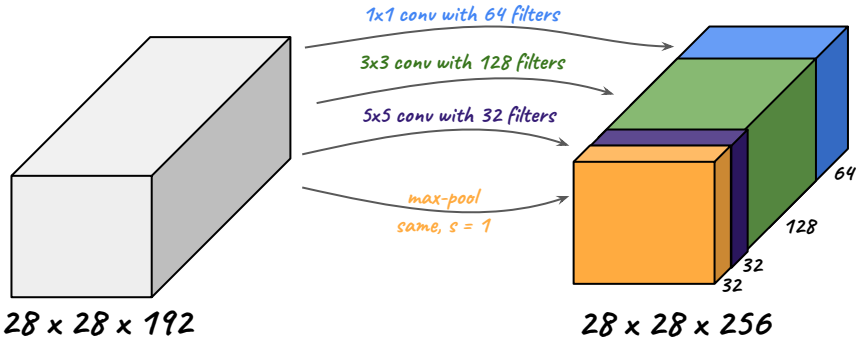


That's an example of **inception module**, at the base of the **inception networks**.

Instead of hand-picking filters and pooling layers, do them all at the same time and concatenate the outputs, and let the network learn whatever parameters it wants to use whatever combination of those filters sizes at once. Doing this without **1x1 convolutions** would be extremely costly (in this case, 120 *million* multiplications).

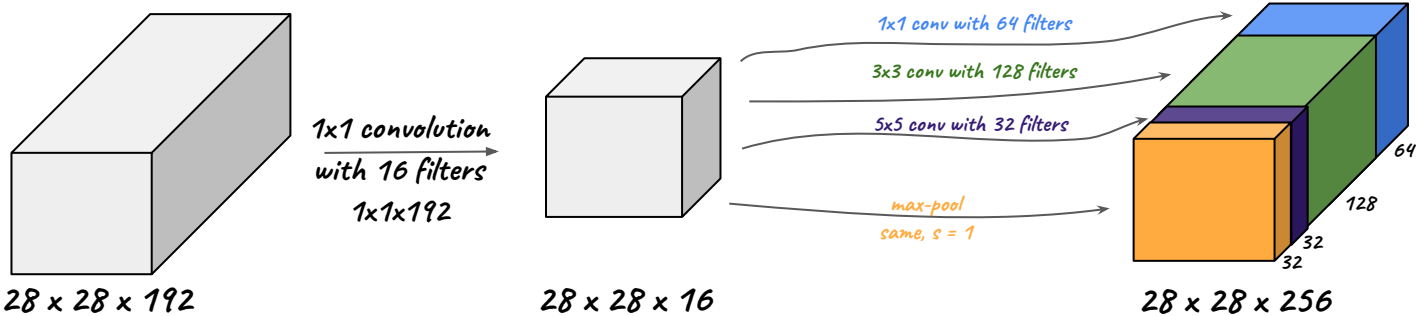
Szegedi et al., 2014,
"Going deeper with convolutions"

The idea behind the **inception module** is to *simultaneously* do convolutions with different filters **and** pooling.



That's an example of **inception module**, at the base of the **inception networks**.

Instead of hand-picking filters and pooling layers, do them all at the same time and concatenate the outputs, and let the network learn whatever parameters it wants to use whatever combination of those filters sizes at once. Doing this without **1x1 convolutions** would be extremely costly (in this case, 120 *million* multiplications).



12.4 million multiplications
a factor 10 reduction

Szegedi et al., 2014,
"Going deeper with convolutions"

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.

That's the picture of a spiral galaxy

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.

That's the picture of a spiral galaxy

That's the picture of a spiral galaxy too

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.

That's the picture of a spiral galaxy

(mirroring)

That's the picture of a spiral galaxy too

Yep, this is a spiral galaxy too

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.





You get it, those are two pictures of a spiral galaxy (random cropping)

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.





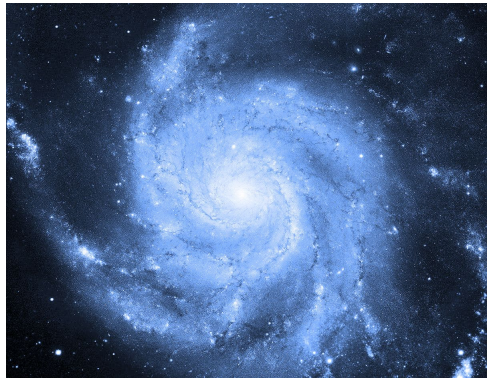

You know, this picture reminds me of a spiral galaxy, somehow

As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.





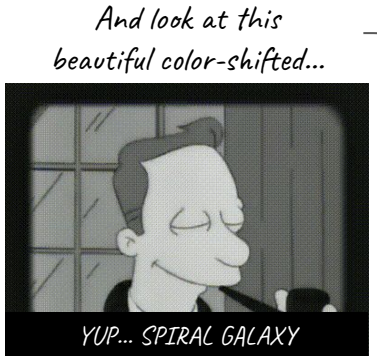
And look at this beautiful color-shifted...

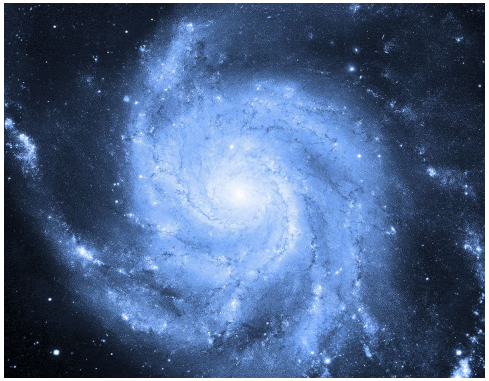


As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.









As you see, there are **lots** of parameters to fit to obtain the best possible model of *CNN*.
If you have to train on million of parameters, your training set cannot contain only tens of thousands of examples.
That's where *data augmentation* (and *synthesis*) comes to the rescue.



With a trivial set of rotations, mirroring, cropping, color-shifting (e.g. PCA-color augmentation) & co. you could easily increase your training set by a factor of 10



Now, I won't get that much into details of object detection and similar, but I'll give you a couple of hints on *semantic segmentation*, since it might be useful for Astrophysical applications.

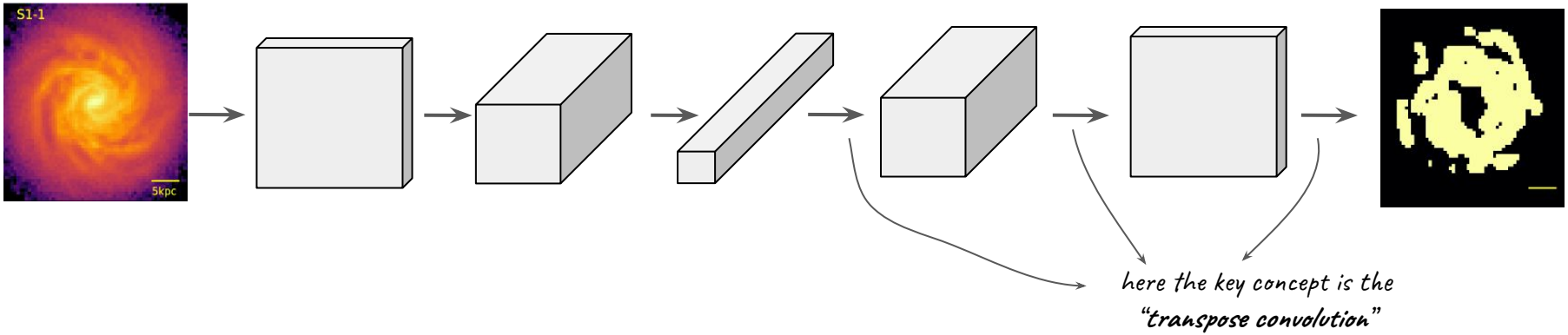
Ronneberget et al., 2015, "U-Net: convolutional Networks for Biomedical image segmentation"

Semantic Segmentation

Now, I won't get that much into details of object detection and similar, but I'll give you a couple of hints on *semantic segmentation*, since it might be useful for Astrophysical applications.
Semantic segmentations means labelling ***EVERY*** pixel of a picture. Automatically labelling. So, in the case of a self-driving car, each pixel on the camera has a label attached to it saying *pedestrian / stop light / pedestrian crossing / stop signal / other car etc... etc...*
In an Astrophysical application, you could e.g. train a *U-Net* to automatically recognize signal pixels from noise pixel, or spiral galaxy pixels from elliptical galaxy pixels, or radio jets pixels from compact radio emission, you name it.
What is the typical architecture of a U-Net?

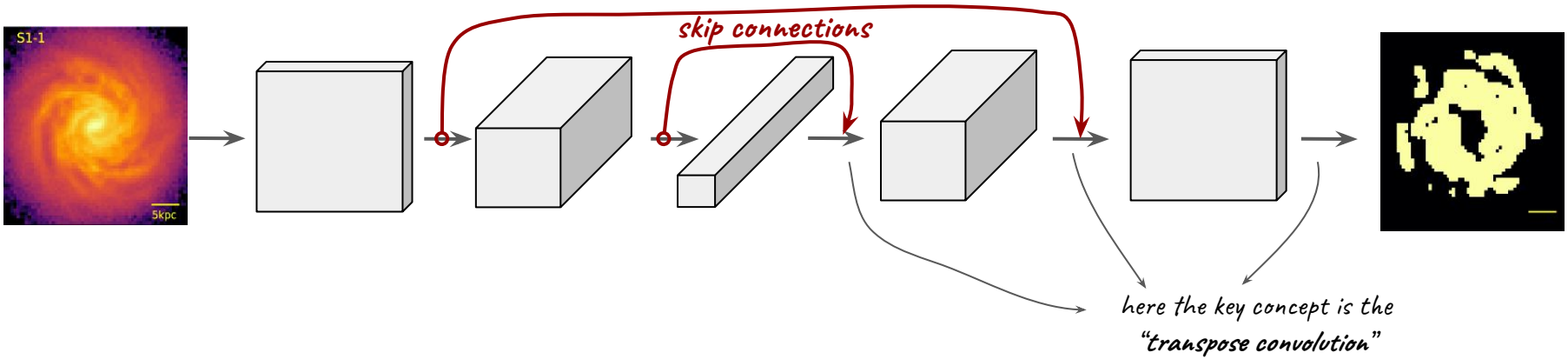
Ronneberget et al., 2015, "U-Net: convolutional Networks for Biomedical image segmentation"

Now, I won't get that much into details of object detection and similar, but I'll give you a couple of hints on *semantic segmentation*, since it might be useful for Astrophysical applications.
Semantic segmentations means labelling ***EVERY*** pixel of a picture. Automatically labelling. So, in the case of a self-driving car, each pixel on the camera has a label attached to it saying *pedestrian / stop light / pedestrian crossing / stop signal / other car etc... etc...*
In an Astrophysical application, you could e.g. train a *U-Net* to automatically recognize signal pixels from noise pixel, or spiral galaxy pixels from elliptical galaxy pixels, or radio jets pixels from compact radio emission, you name it.
What is the typical architecture of a U-Net?



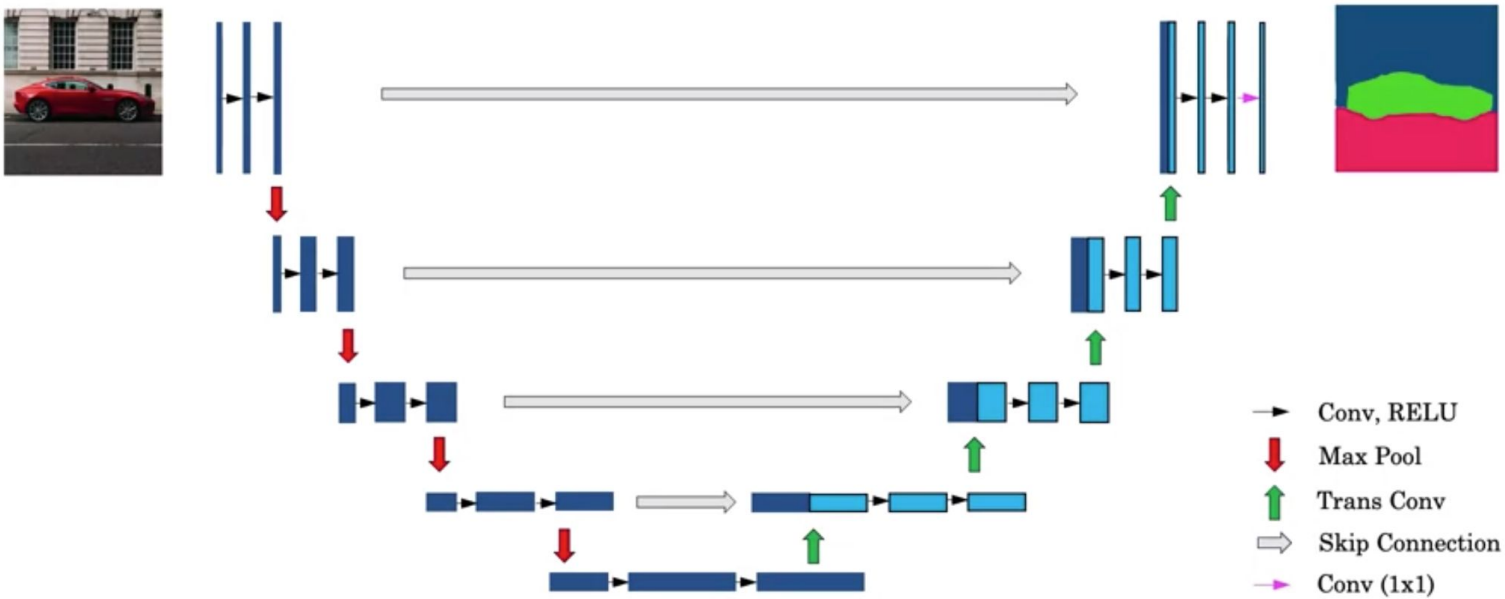
Ronneberger et al., 2015, "U-Net: convolutional Networks for Biomedical image segmentation"
Bekki et al., 2021, "Quantifying the fine structures of disk galaxies with deep learning: Segmentation of spiral arms in different Hubble types"

Now, I won't get that much into details of object detection and similar, but I'll give you a couple of hints on *semantic segmentation*, since it might be useful for Astrophysical applications.
Semantic segmentations means labelling ***EVERY*** pixel of a picture. Automatically labelling. So, in the case of a self-driving car, each pixel on the camera has a label attached to it saying *pedestrian / stop light / pedestrian crossing / stop signal / other car etc... etc...*
In an Astrophysical application, you could e.g. train a *U-Net* to automatically recognize signal pixels from noise pixel, or spiral galaxy pixels from elliptical galaxy pixels, or radio jets pixels from compact radio emission, you name it.
What is the typical architecture of a U-Net?



Ronneberger et al., 2015, "U-Net: convolutional Networks for Biomedical image segmentation"
Bekki et al., 2021, "Quantifying the fine structures of disk galaxies with deep learning: Segmentation of spiral arms in different Hubble types"

Now, I won't get that much into details of object detection and similar, but I'll give you a couple of hints on *semantic segmentation*, since it might be useful for Astrophysical applications.
Semantic segmentations means labelling ***EVERY*** pixel of a picture. Automatically labelling. So, in the case of a self-driving car, each pixel on the camera has a label attached to it saying *pedestrian / stop light / pedestrian crossing / stop signal / other car etc... etc...*
In an Astrophysical application, you could e.g. train a *U-Net* to automatically recognize signal pixels from noise pixel, or spiral galaxy pixels from elliptical galaxy pixels, or radio jets pixels from compact radio emission, you name it.
What is the typical architecture of a U-Net?



Ronneberget et al., 2015, "U-Net: convolutional Networks for Biomedical image segmentation"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

Which face belongs to a real human being, and which one is a GAN generated one?

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

Which face belongs to a real human being, and which one is a GAN generated one?

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

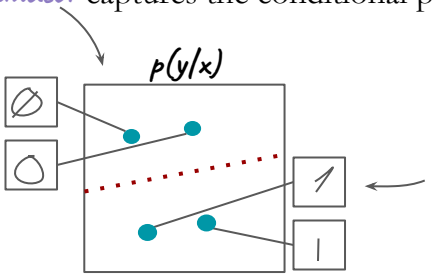
- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$



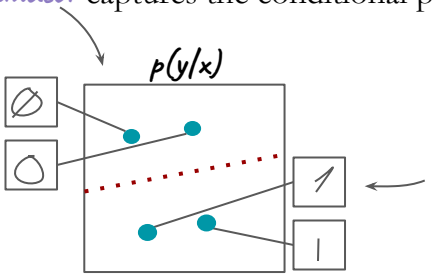
The discriminator tries to tell the difference between 0 and 1 by placing a decision boundary between the classes, without caring about where the data are actually positioned in the feature space

Goodfellow et al., 2014, "Generative Adversarial Networks"

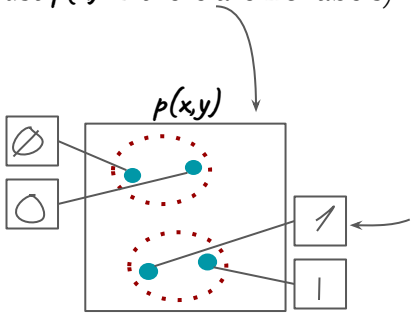
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***



The discriminator tries to tell the difference between 0 and 1 by placing a decision boundary between the classes, without caring about where the data are actually positioned in the feature space



*The generator on the other hand tries to produce convincing 0's and 1's by generating digits that fall close to their real counterparts in feature space, therefore it **HAS** to model the distribution throughout the feature space.*

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

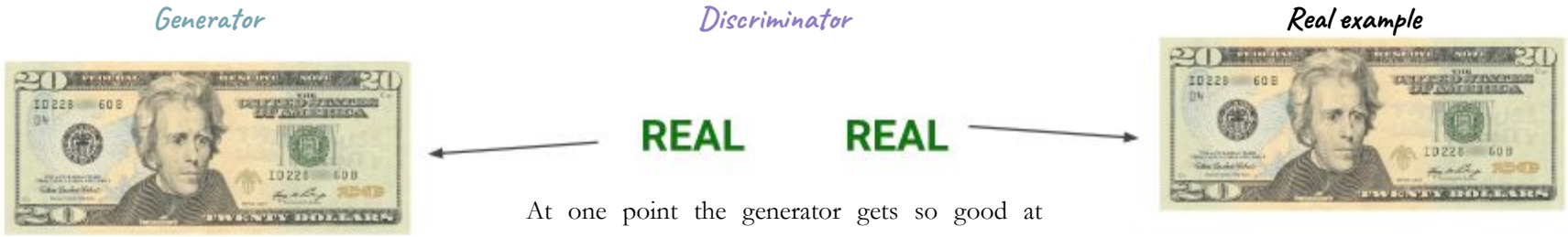
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

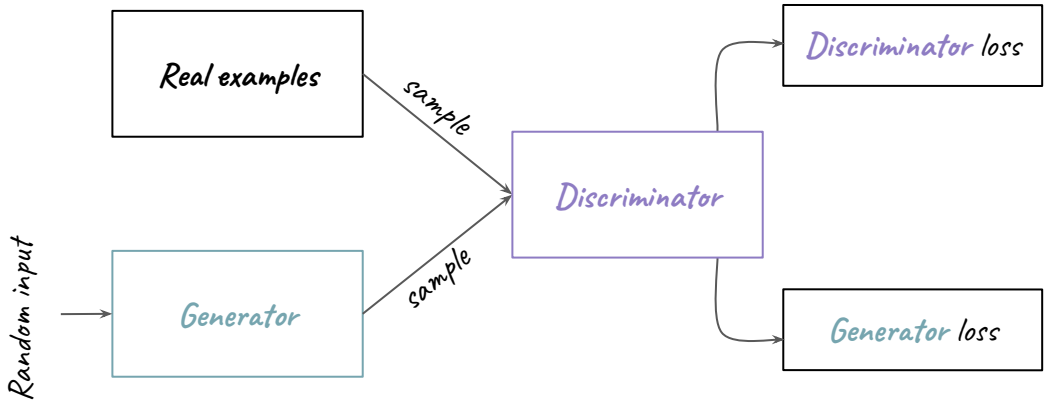
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

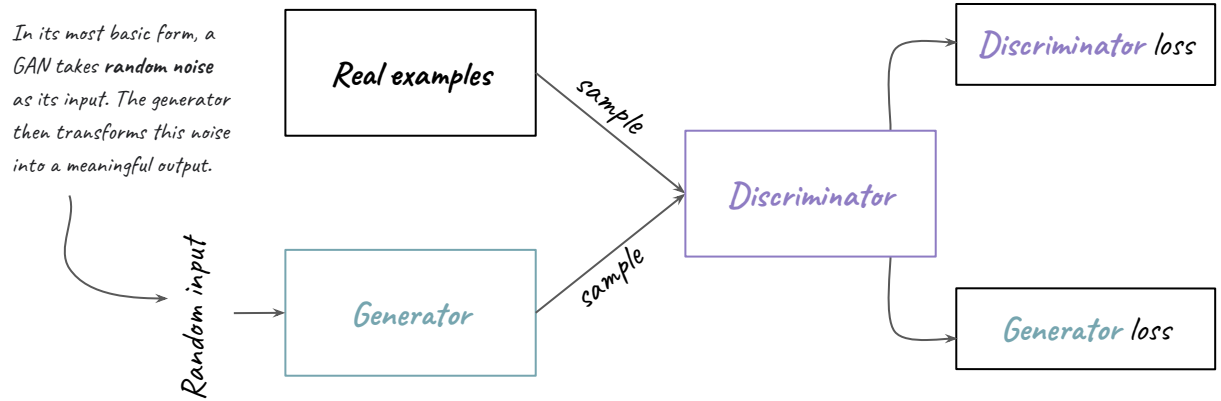
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

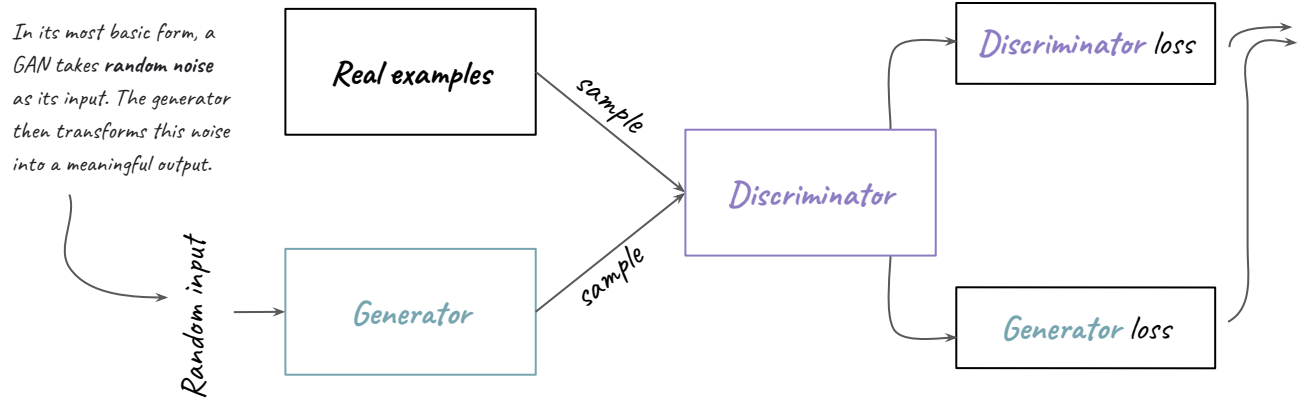
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



In the original 2014 paper, both the *generator* and *discriminator* losses are a *minimax losses*, that the *generator* tries to minimize while the *discriminator* tries to maximize.

Nowadays, most *GANs* are trained on the *Wasserstein loss*, but I do not have 150 hours to talk about these things so I will just leave this information here and refuse to elaborate further.

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

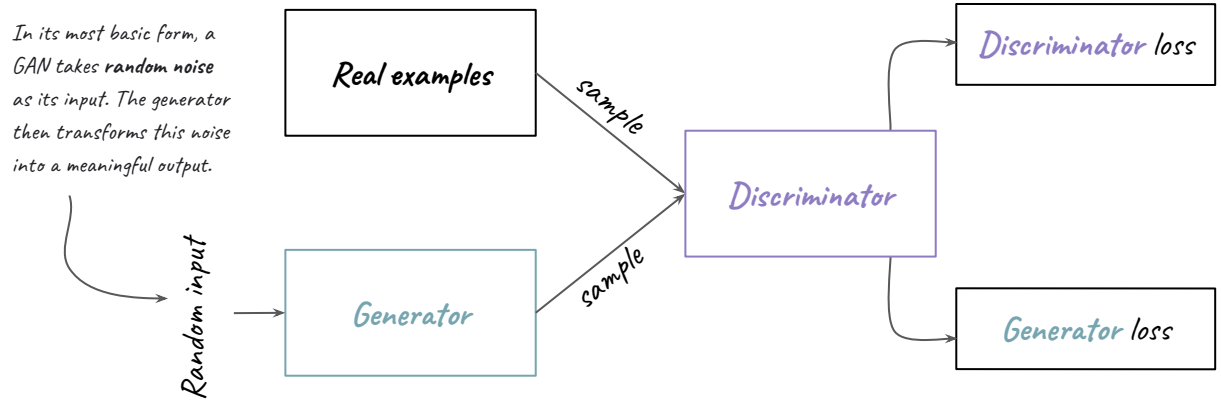
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Both the *generator* and the *discriminator* are neural networks.

The *generator* output is connected directly to the *discriminator* input.

Through backpropagation, the *discriminator's* classification provides a signal that the *generator* uses to update its weights

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

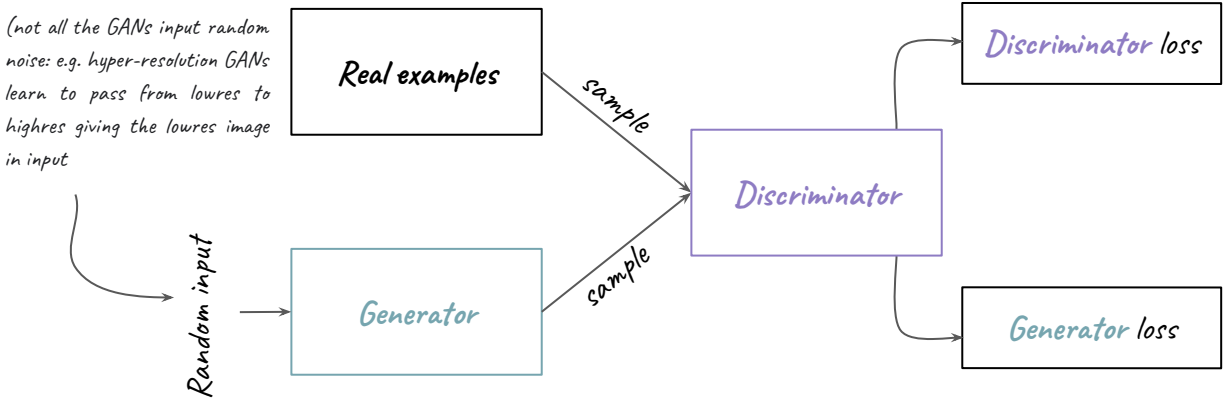
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability $p(x,y)$ of the training data (or just $p(x)$ if there are no labels)
- the *discriminator* captures the conditional probability $p(y|x)$

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Both the *generator* and the *discriminator* are neural networks.

The *generator* output is connected directly to the *discriminator* input.

Through backpropagation, the *discriminator's* classification provides a signal that the *generator* uses to update its weights

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

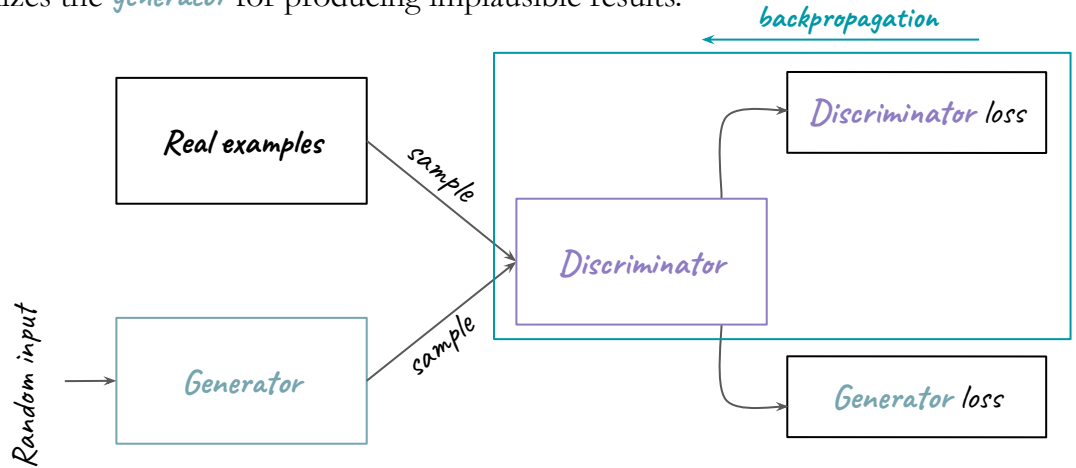
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



The *discriminator* is a simple classifier. Does not even need to have a softmax layer, hell, it could even be a simple logistic regression.

It gets trained on its ability to distinguish fake examples (0) from real examples (1).

During its training, it ignores the *generator* loss.

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

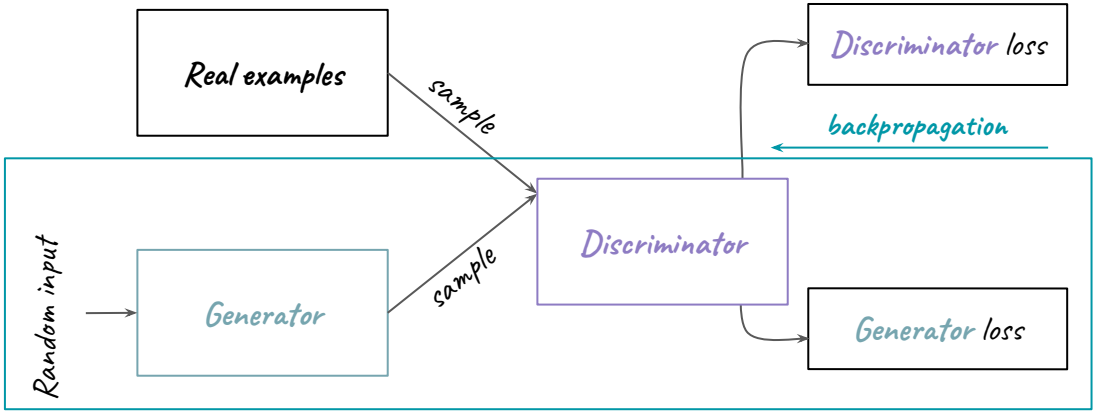
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



The *generator* learns to generate fake data based on the feedback it receives from the *discriminator*.

It gets trained with ***backpropagation***, whose loss is calculated by the *discriminator*, but pay attention, in this ***backpropagation*** phase **ONLY** the *generator* weights gets changed with the information coming from the loss.

Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

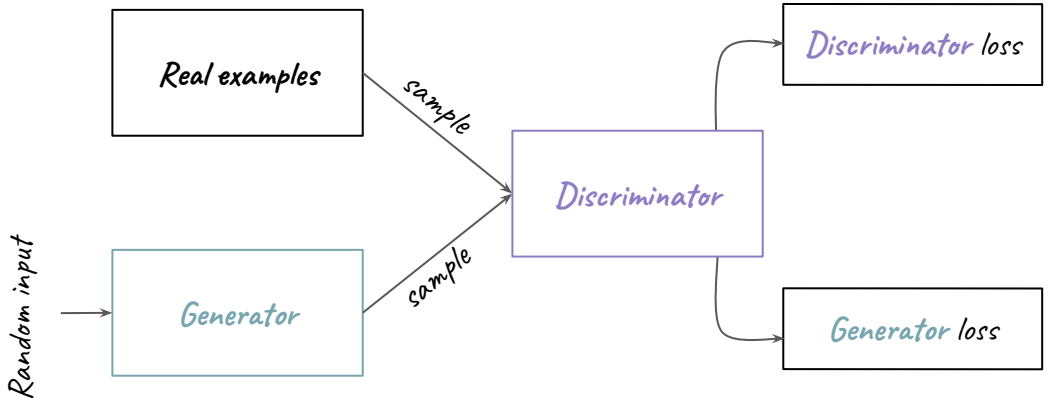
Generative Adversarial Networks (GANs) are *generative models*, meaning that they create new data that resemble the training data.
A typical *GAN* architecture pairs a *generator*, which learns to produce the target output, and a *discriminator*, which learns to distinguish the true data from the generated one.

To be more formal:

- the *generator* captures the joint probability ***p(x,y)*** of the training data (or just ***p(x)*** if there are no labels)
- the *discriminator* captures the conditional probability ***p(y|x)***

In practice:

- the *generator* learns to generate plausible data. The generated examples become negative examples (i.e., 0) for the discriminator.
- the *discriminator* learns to distinguish between the generator's fake examples and real positive (i.e., 1) examples in the training set, and penalizes the *generator* for producing implausible results.



Generator and *discriminator* get trained alternatively: one first for some epochs, then the other for some epochs, and repeat.

During the training of one, the other stays unchanged.

In principle, convergence is when the *discriminator* has a 50% chance of getting the right answer = flipping a coin.

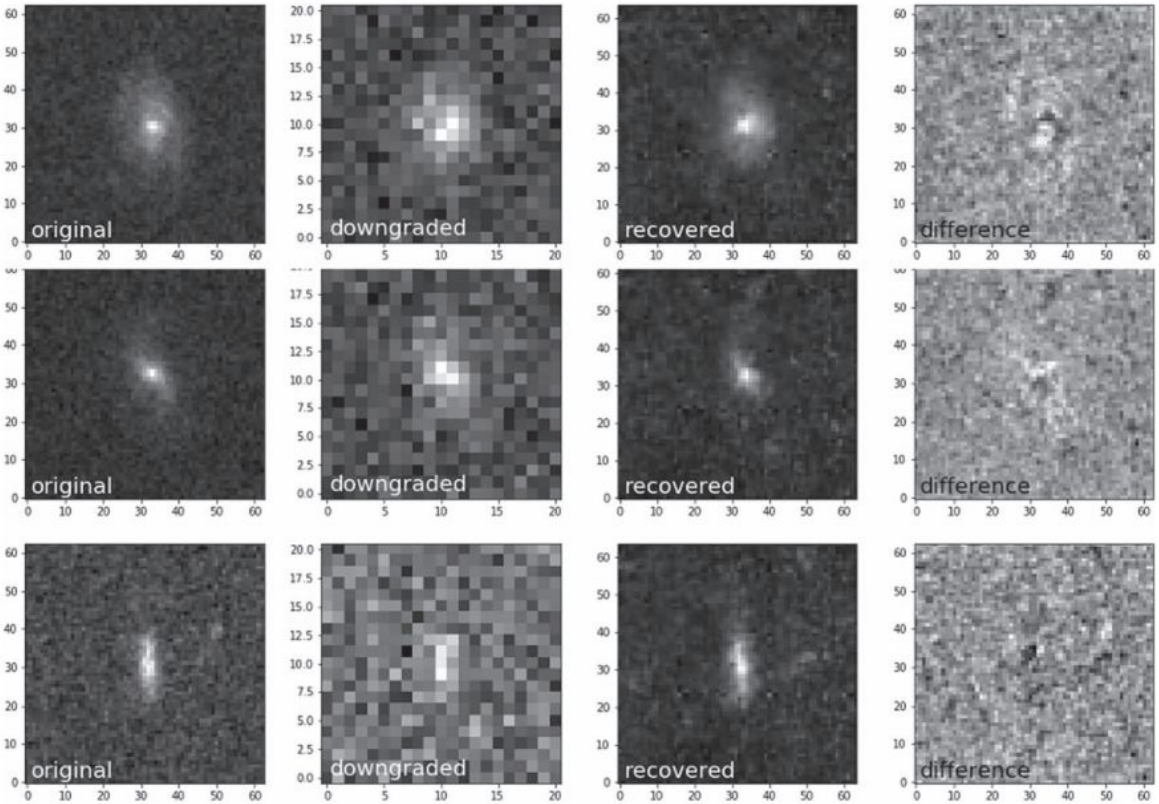
Google Advanced Course on GANs, here.

Goodfellow et al., 2014, "Generative Adversarial Networks"

GANs for source deblending and hyper-resolution

The practical applications of *GANs* (and generative models in general) are numerous, not just to produce fake pictures for a Tinder profile.

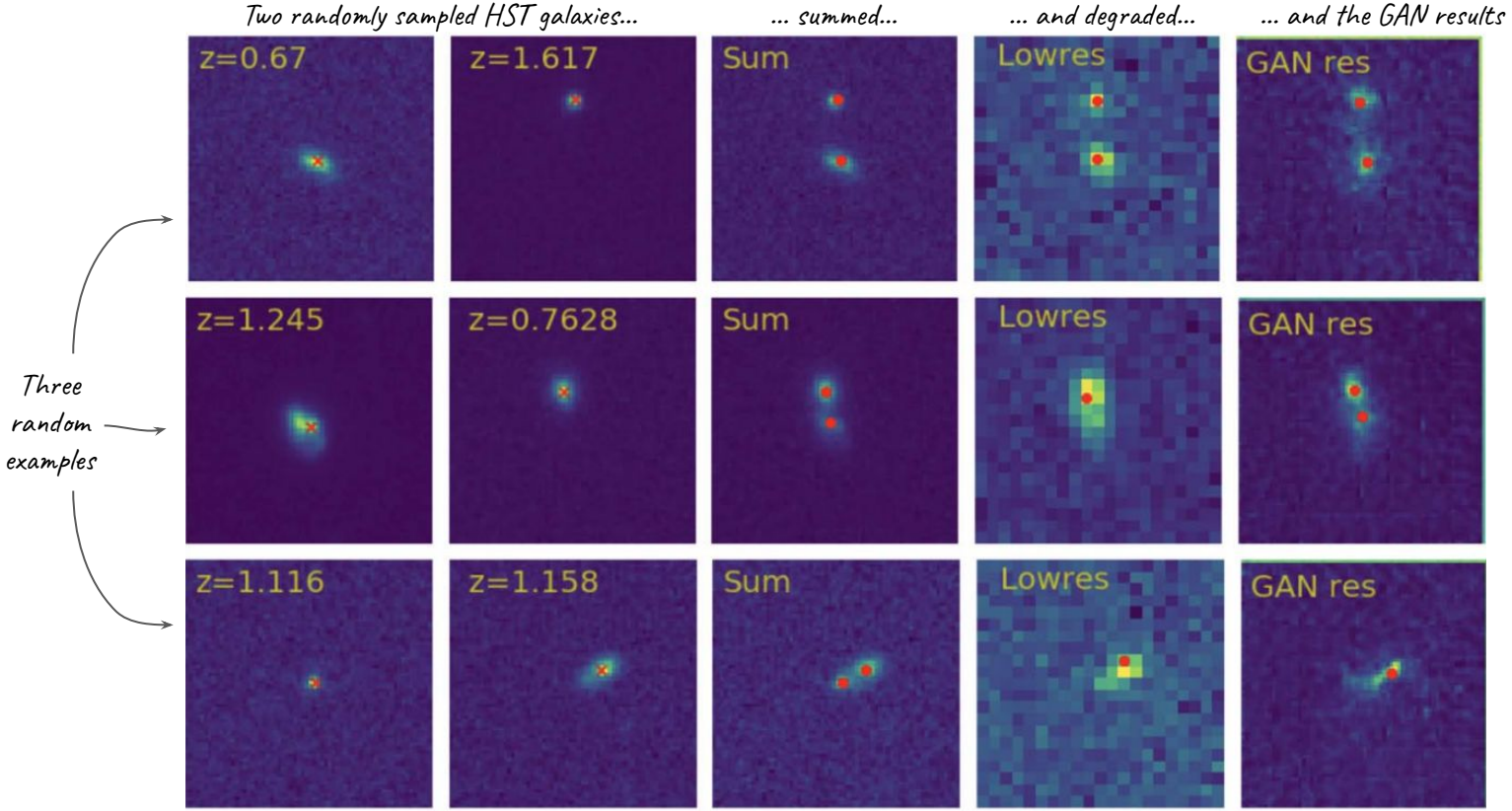
In this recent work, Hemmati and collaborators trained a *GAN* to deblend galaxies, and in the meanwhile they also produced a *low-res* to *high-res* machine



Hemmati et al., 2022, "Deblending Galaxies with Generative Adversarial Networks"

GANs for source deblending and hyper-resolution

The practical applications of *GANs* (and generative models in general) are numerous, not just to produce fake pictures for a Tinder profile.

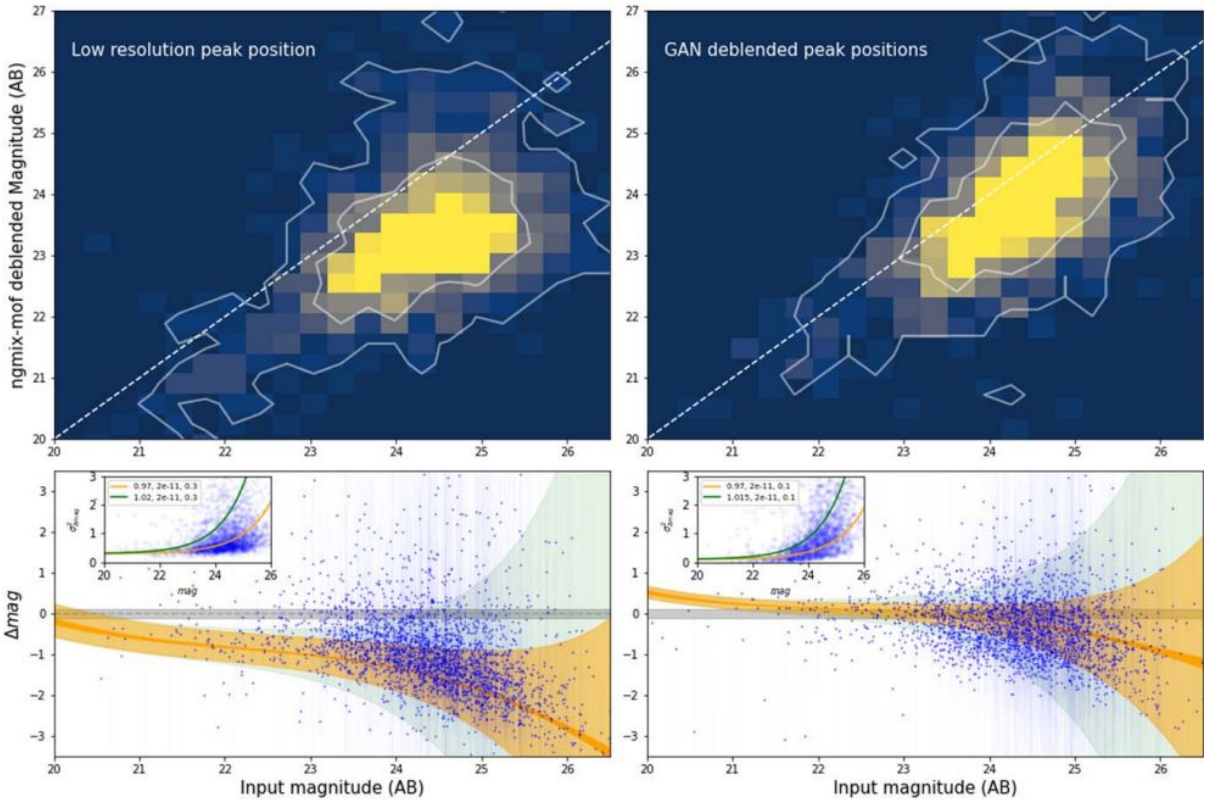


Hemmati et al., 2022, "Deblending Galaxies with Generative Adversarial Networks"

GANs for source deblending and hyper-resolution

The practical applications of *GANs* (and generative models in general) are numerous, not just to produce fake pictures for a Tinder profile.

The results without applying GAN deblending

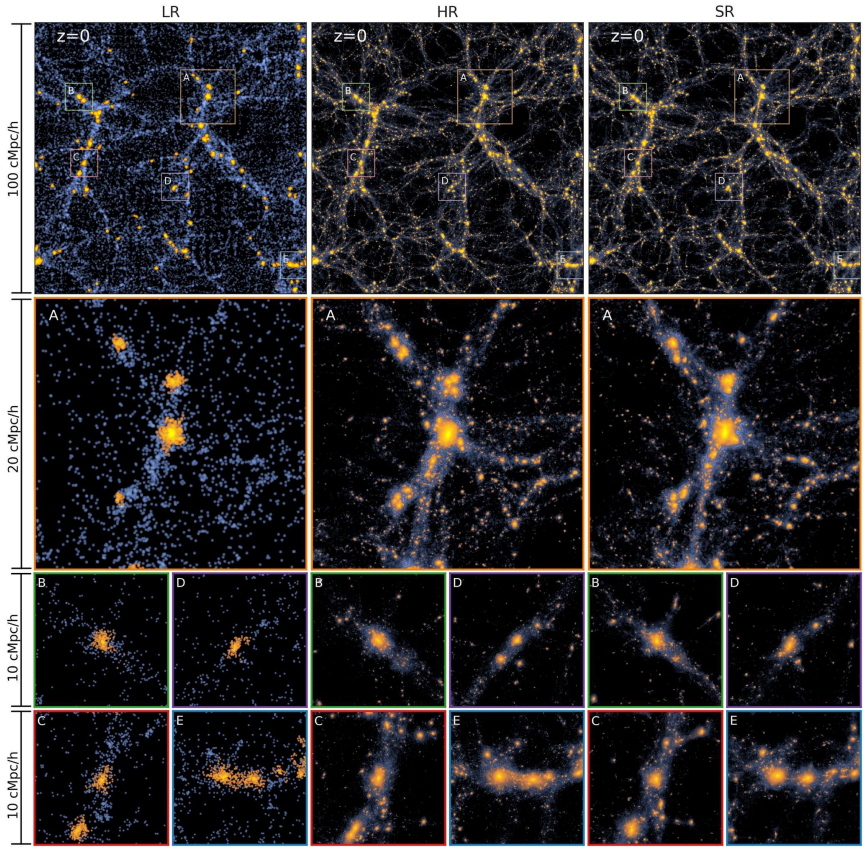


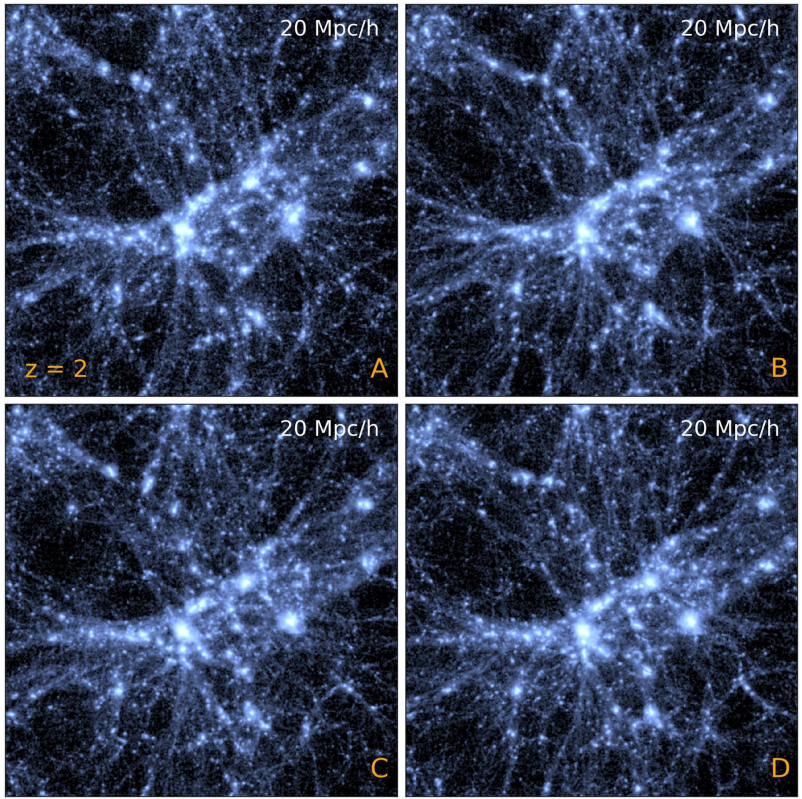
The results applying GAN deblending

Hemmati et al., 2022, "Deblending Galaxies with Generative Adversarial Networks"

GANs for source deblending and hyper-resolution

The practical applications of *GANs* (and generative models in general) are numerous, not just to produce fake pictures for a Tinder profile.





Four different hi-res GAN application on the same volume: if you notice the fine grain, yes, there are some differences, but the statistical details of all four images are exactly the same

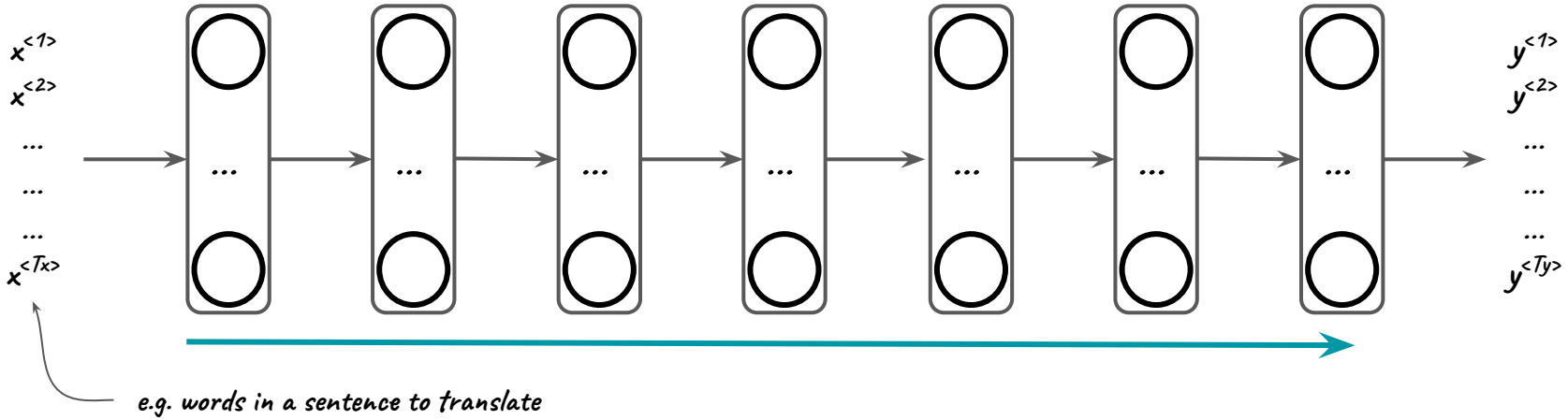
Li et al, 2021, "AI-assisted superresolution cosmological simulations"

Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing …

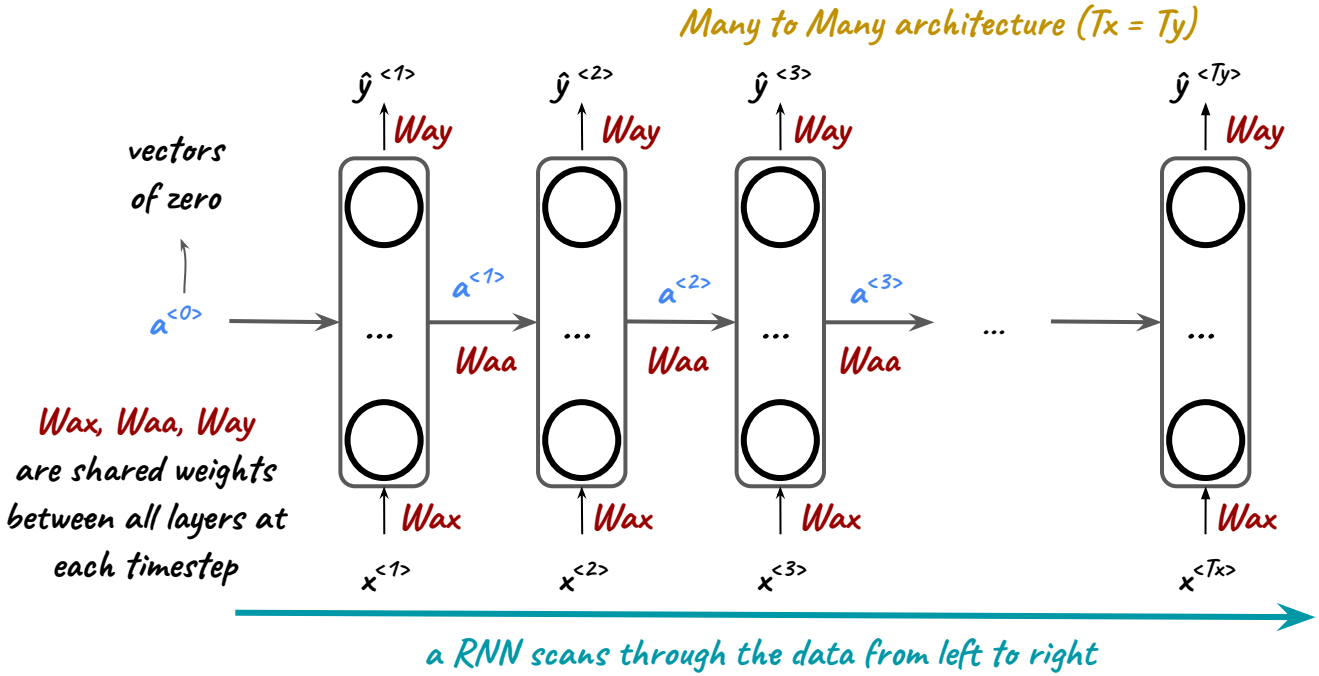
One of the problems of a typical Neural Network (a *feed-forward* Neural Network) is that it has no memory. A *FFNN* would have lots of trouble in processing a written text: the information in the previous sentences would be completely lost, the I/O might have different sizes in different examples, the number of parameters explodes fast even for nowadays computational efforts.



Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing …

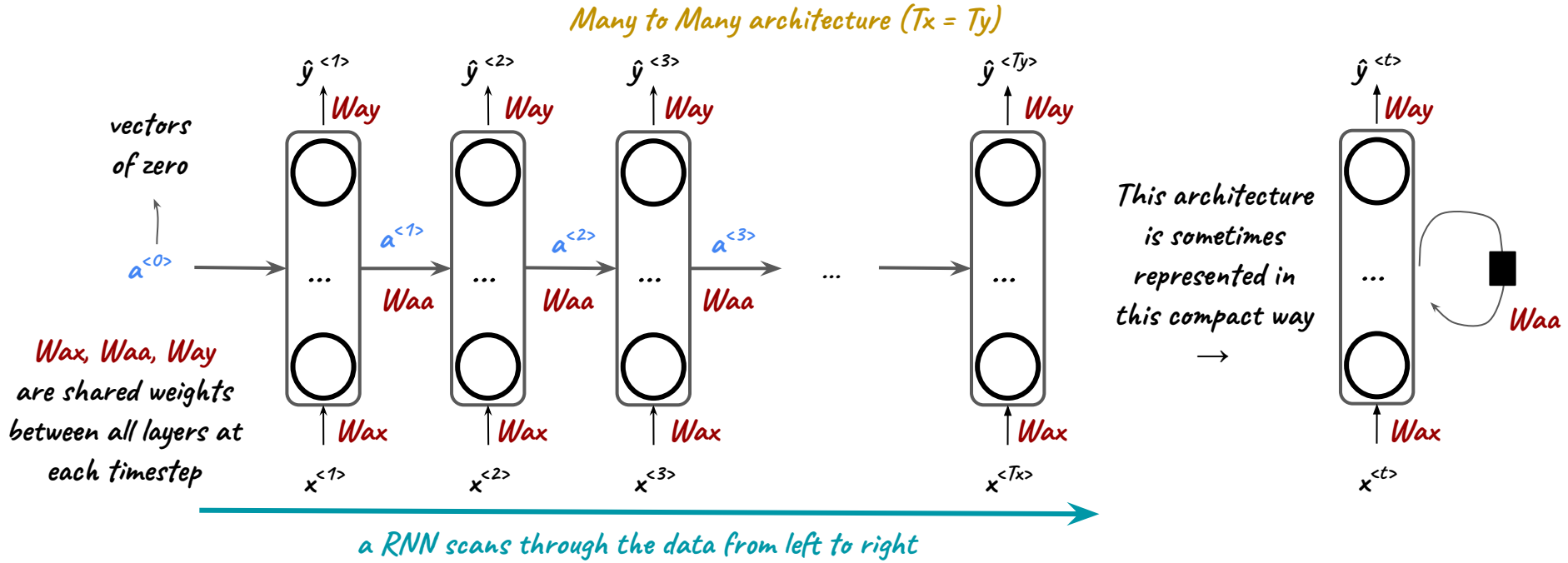
These networks scan the data from left to right (or the opposite, e.g. arabic or satanic sentences in rock music), and each example $x^{<t>}$ enters a layer and exits the layer as a $\hat{y}^{<t>}$ label. In this way previously scanned examples influences later examples.



Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing …

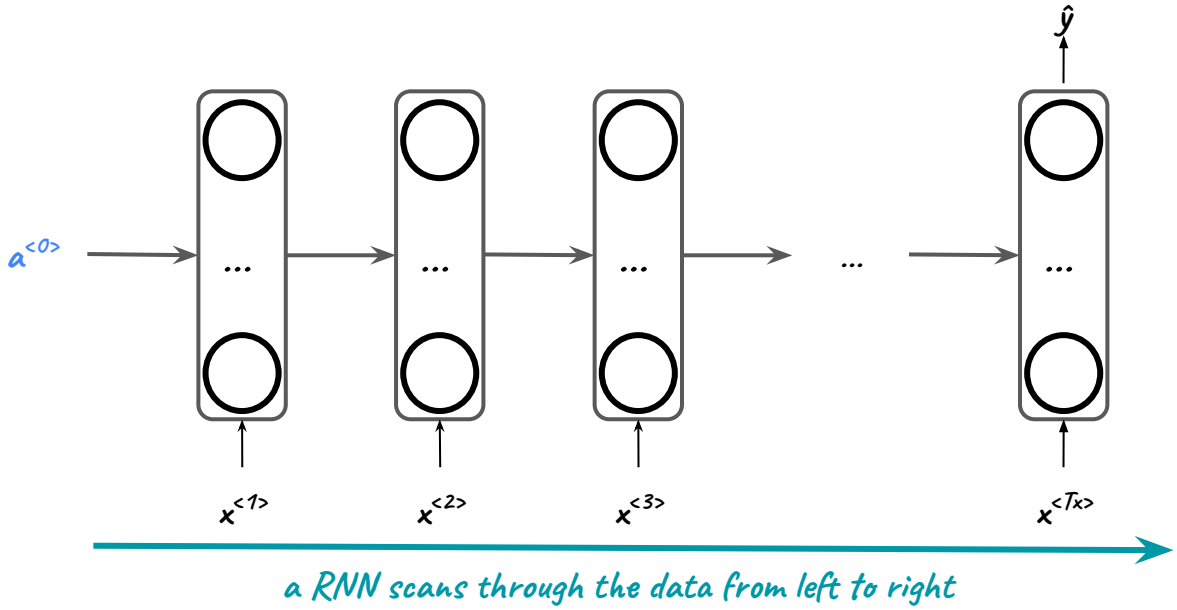
These networks scan the data from left to right (or the opposite, e.g. arabic or satanic sentences in rock music), and each example $x^{<t>}$ enters a layer and exits the layer as a $\hat{y}^{<t>}$ label. In this way previously scanned examples influences later examples.



Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing … These networks scan the data from left to right (or the opposite, e.g. arabic or satanic sentences in rock music), and each example $x^{<t>}$ enters a layer and exits the layer as a $\hat{y}^{<t>}$ label. In this way previously scanned examples influences later examples.

Many to One architecture (Ty = 1)



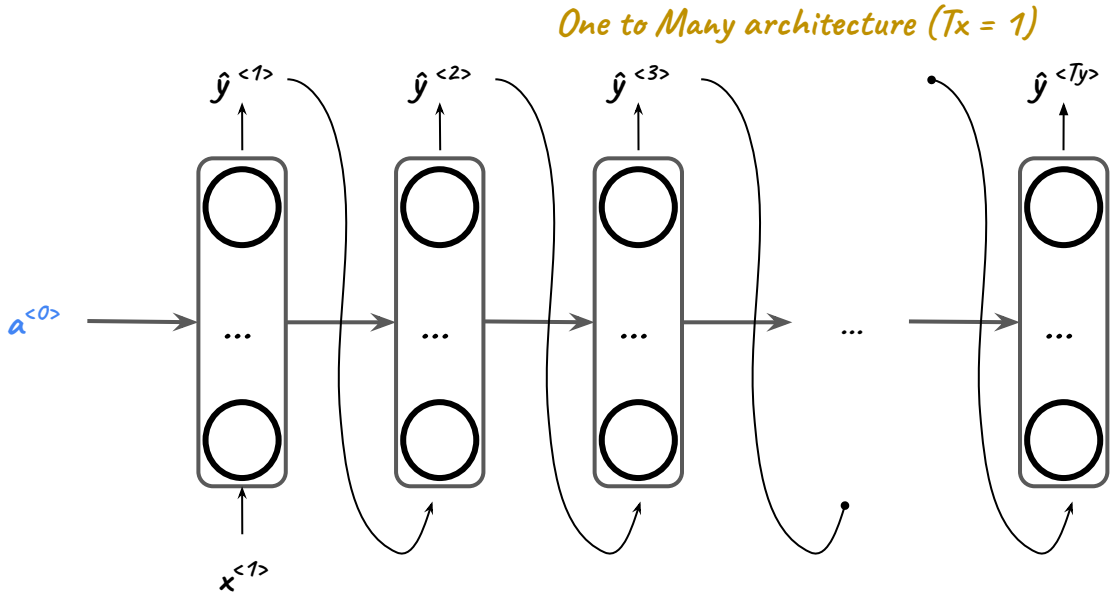
e.g., Sentiment Analysis

"This machine learning course sucked, the teacher was a disaster, it kept quoting obscure boomer-pop-culture references, I did not learn anything useful, please avoid it"

Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing …

These networks scan the data from left to right (or the opposite, e.g. arabic or satanic sentences in rock music), and each example $x^{<t>}$ enters a layer and exits the layer as a $\hat{y}^{<t>}$ label. In this way previously scanned examples influences later examples.



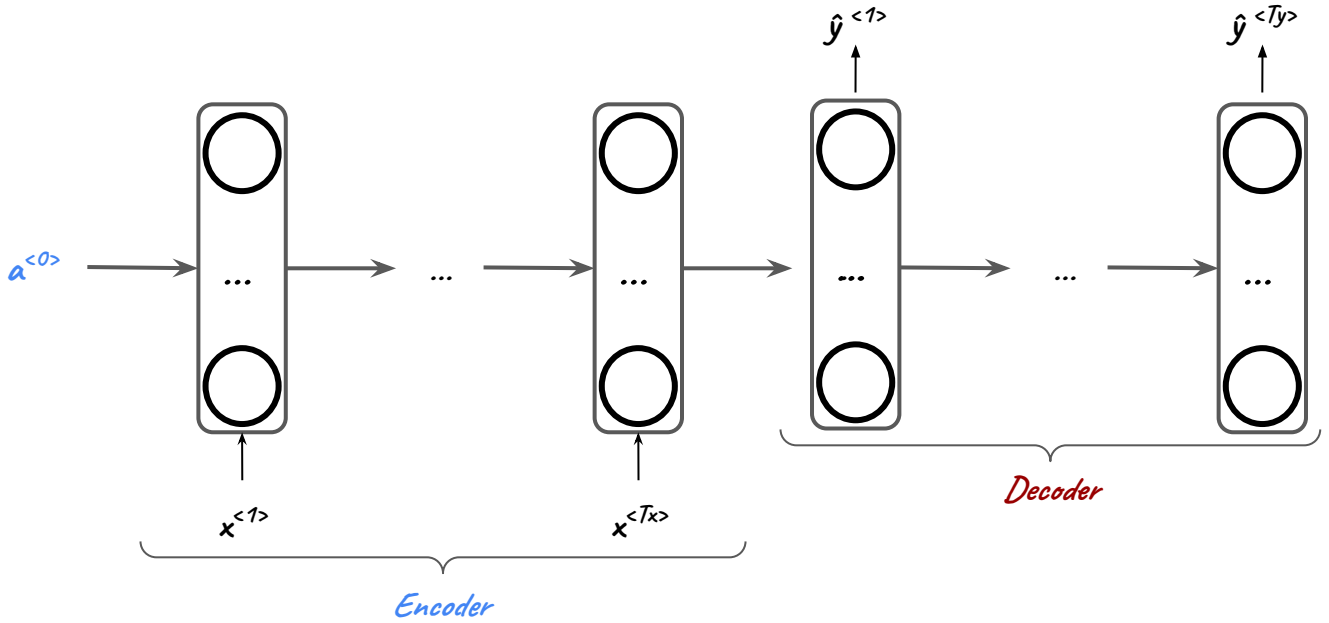
e.g., music generation where the input prompts only a genre, and the RNN outputs notes, rhythm, harmonies, contrapunctus etc...

Recurrent Neural Networks are not that straightforward, and it would take me other 6 hours to talk thoroughly about them. Here I'm just showing you a couple of slides to give an intuition about them, and if you're interested there are **lots** of available online resources on the subject to learn.

RNNs are the centerpiece of ***sequence models***: speech recognition, music generation, automatic translation, natural language processing …

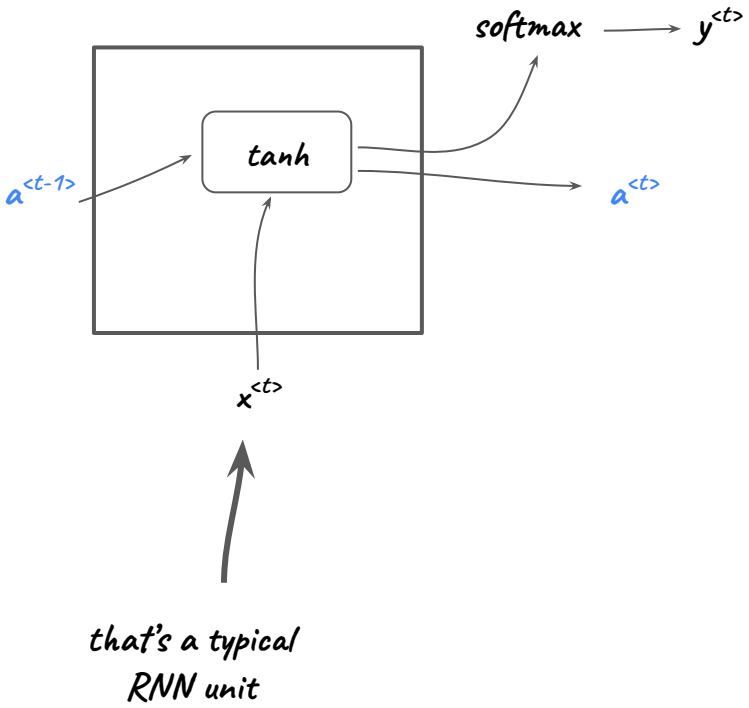
These networks scan the data from left to right (or the opposite, e.g. arabic or satanic sentences in rock music), and each example $x^{<t>}$ enters a layer and exits the layer as a $\hat{y}^{<t>}$ label. In this way previously scanned examples influences later examples.

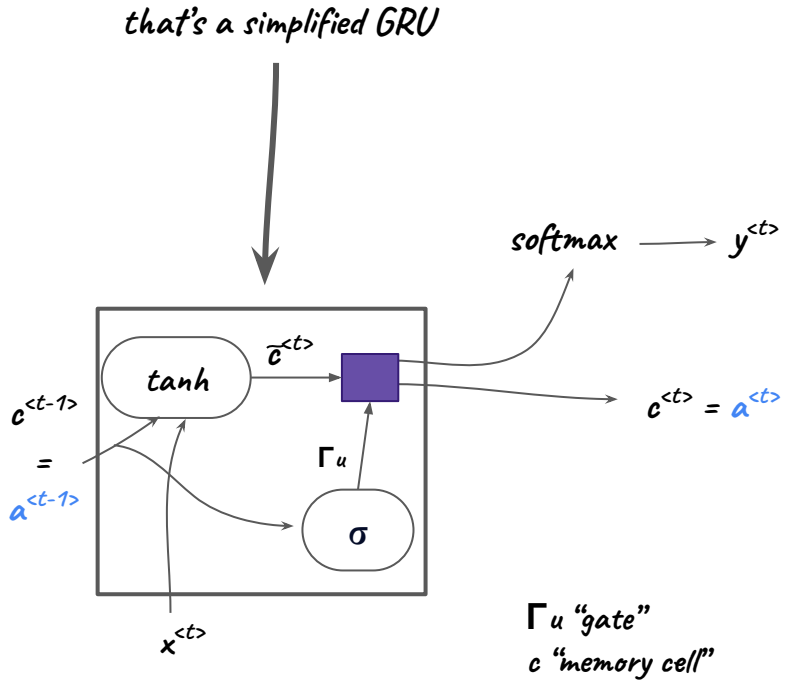
Many to Many architecture (Tx ≠ Ty)



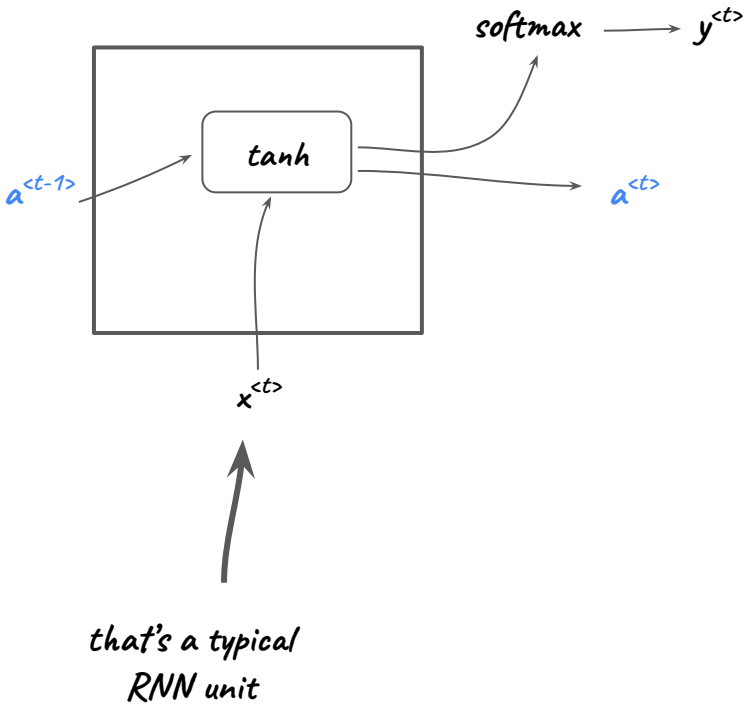
e.g. automatic translation (DeepL, Google Translate...) or sentences rephrasing (quillbot)

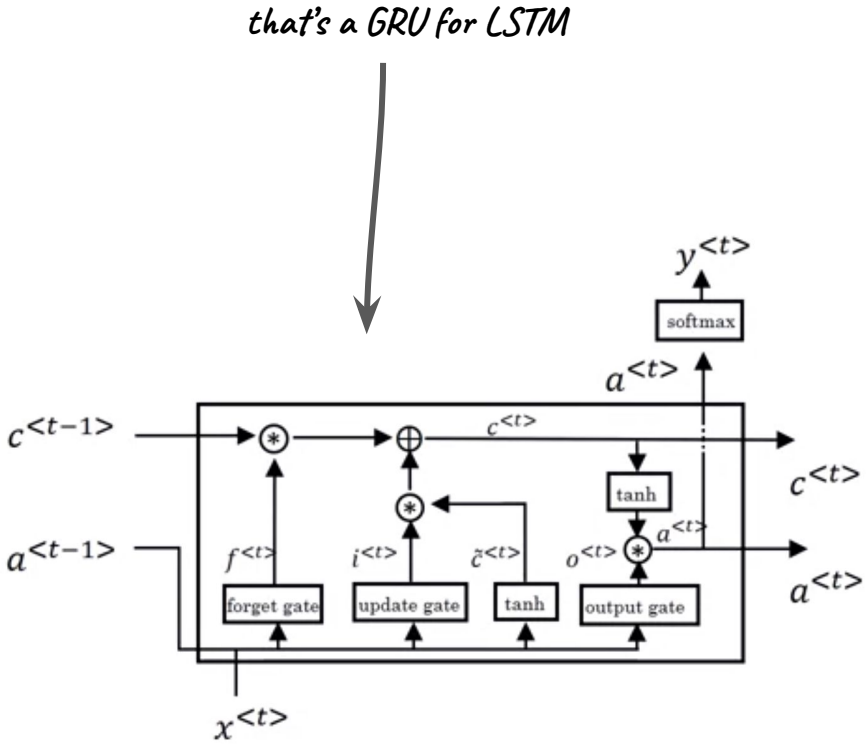
Those *RNNs* have short-term memory. *Gated Recurrent Units (GRUs)* are what make them *Long-Short Term Memory (LSTM)* models.





Those *RNNs* have short-term memory. *Gated Recurrent Units (GRUs)* are what make them *Long-Short Term Memory (LSTM)* models.





... and that's all, folks

Thank you slide for a machine learning course. Generate image

... and that's all, folks

Thank you, in the style of the Looney Tunes "that's all folks" credits

Generate image


we won't get rid of this
neurons madness
anytime soon