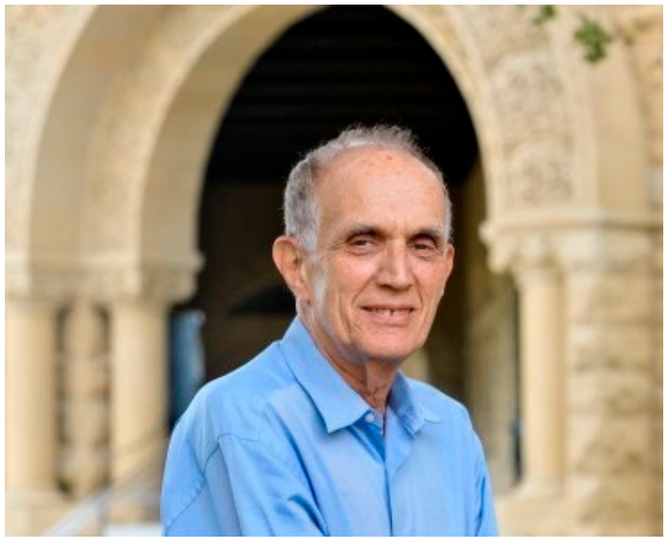
# Ensemble methods

# Bagging
# (Bootstrap Aggregating)



Bradley Efron (1979)

Leo Breiman (1994 )

# Bootstrap

## Why working with a single data set?

```
[[ 0.524]
 [ 0.4  ]
 [ 0.361]
 [ 0.078]
 [ 0.859]
 [ 0.654]
 [ 0.384]
 [ 0.738]
 [ 0.156]
 [ 0.572]]
```

# EE-411 Fundamentals of inference and learning, EPFL

## Exercice Session 1: looking at data with python and pandas

In this first set of exercices, we will start the course by solving some simple python problems that will help you warm up for the future. We advise you to run python notebooks on your browser, for instance using google colab (Watch Introduction to Colab to find out more) or our own serveur in epfl (on noto.epfl.ch). If you need to refresh your python skills, you may start by our introductory notebooks here and there.

**What you will learn today:** In this first session, we will discuss briefly how to use python, how to use pandas (a powerful, and easy to use, open source data analysis and manipulation tool), and discuss the idea of permutation test and bootstrap, that are amazingly useful concepts from statistics.

## The Brexit data: who wanted out?

We will introduce the concept of permutation test in the hypothesis testing problem exploiting a cool analysis on the Brexit referendum following this great resource https://matthew-brett.github.io/les-pilot/brexit_ages.html. This will give us the opportunity to review the basic functionalities of pandas, a pivotal package in data-handling which we will find often during this course. In fact, you may want to follow an introduction to pandas (for instance this one, or the very useful Most Important Pandas Functions for Data Science).

Here is our problem: The Hansard society made a poll in the 2016 interviewing 1771 people on the Brexit referendum (data available here https://beta.ukdataservice.ac.uk/datacatalogue/studies/study?id=8183).

If we dig into the data, we see that the average age of the "Brexiteers" is higher from the one of the "Remainers". We are interested in addressing the following question: how **confident** can we be that this rule is general and is not an artifact due to low sample size?

This problem falls in the broader context of hypothesis testing problem:

- Hypothesis (H1) - effective difference in the age of Brexiteers and Remainers

vs

- "Null hypothesis" (H0) - no difference in the age of the two groups

This seems like a hard problem but we will see that with a very simple idea we will be able to characterize th
steps and let's understand how to handle a dataset with pandas.

**set?**

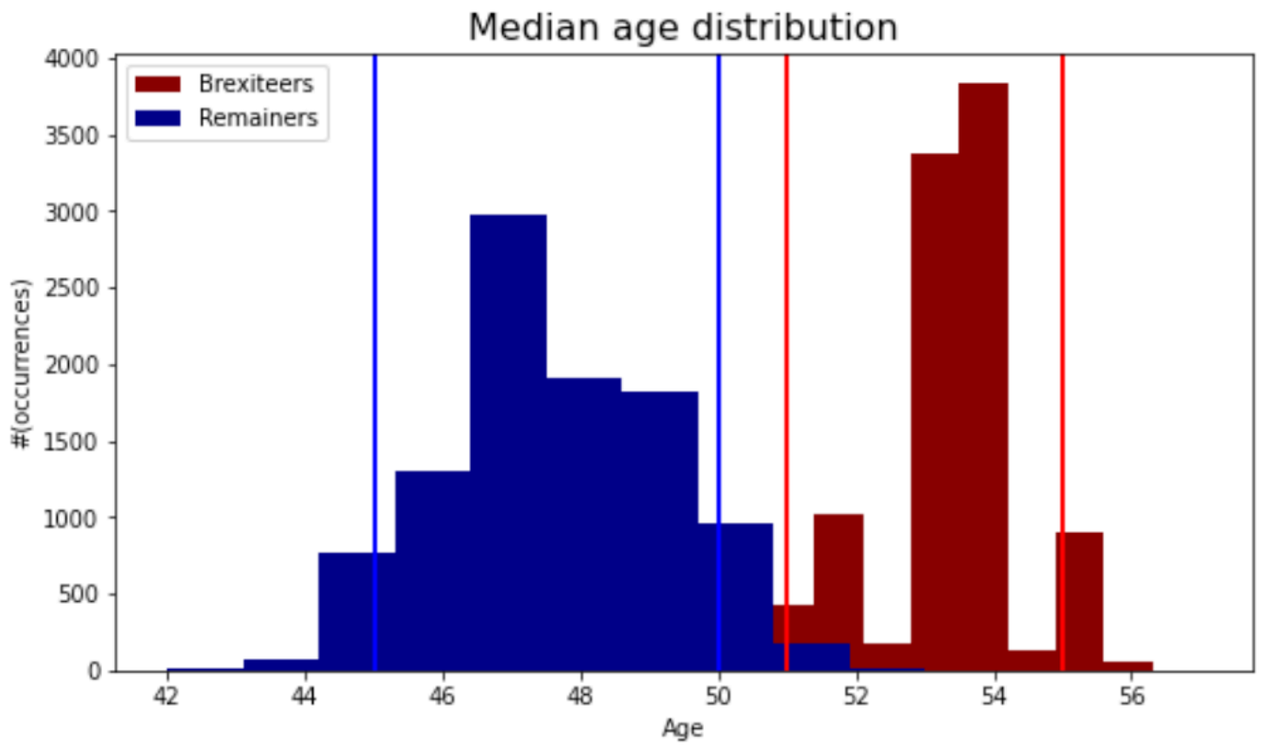
**replacement:**

```
56]
56]
24]
78]
24]
59]
61]
78]
54]
 ]]
```

```
[[ 0.078]
 [ 0.524]
 [ 0.859]
 [ 0.384]
 [ 0.859]
 [ 0.361]
 [ 0.361]
 [ 0.738]
 [ 0.572]
 [ 0.156]]
```

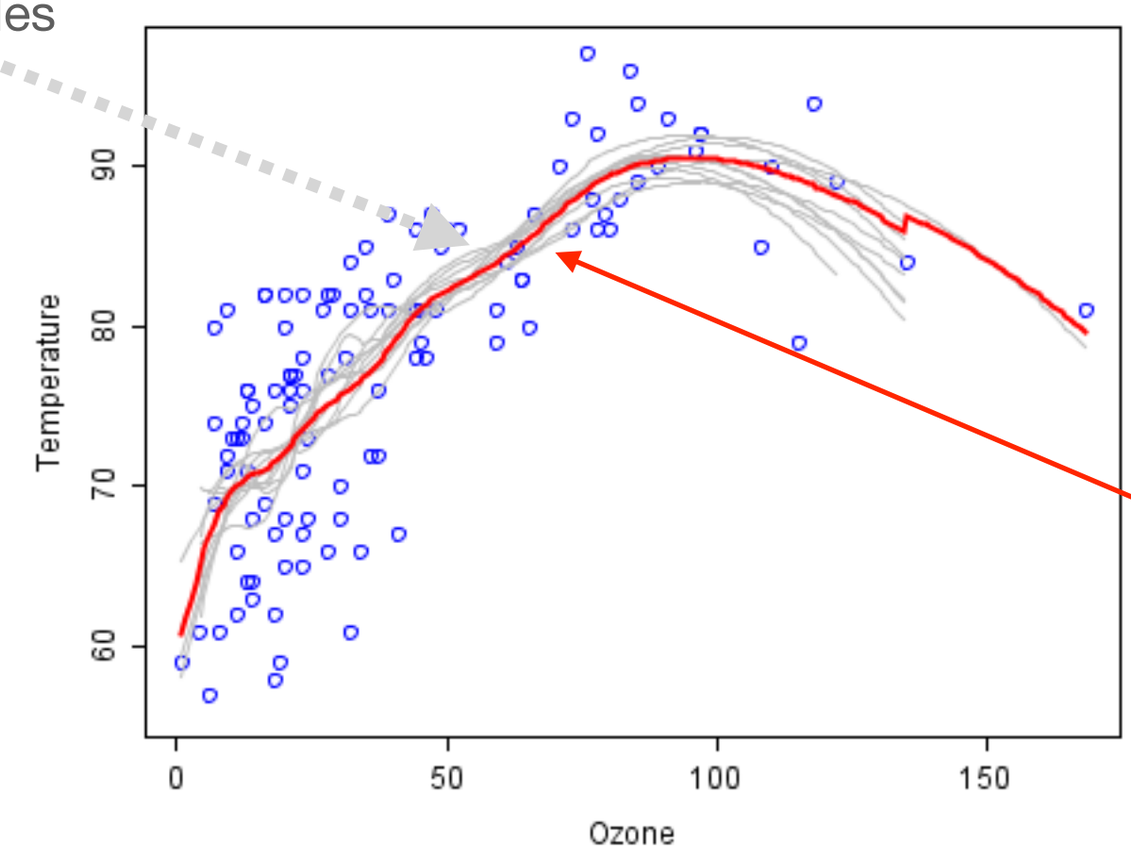
**Consider these**

**Many uses: error anal**



# Bagging
# (Bootstrap Aggregating)

Fits on bootstrap samples

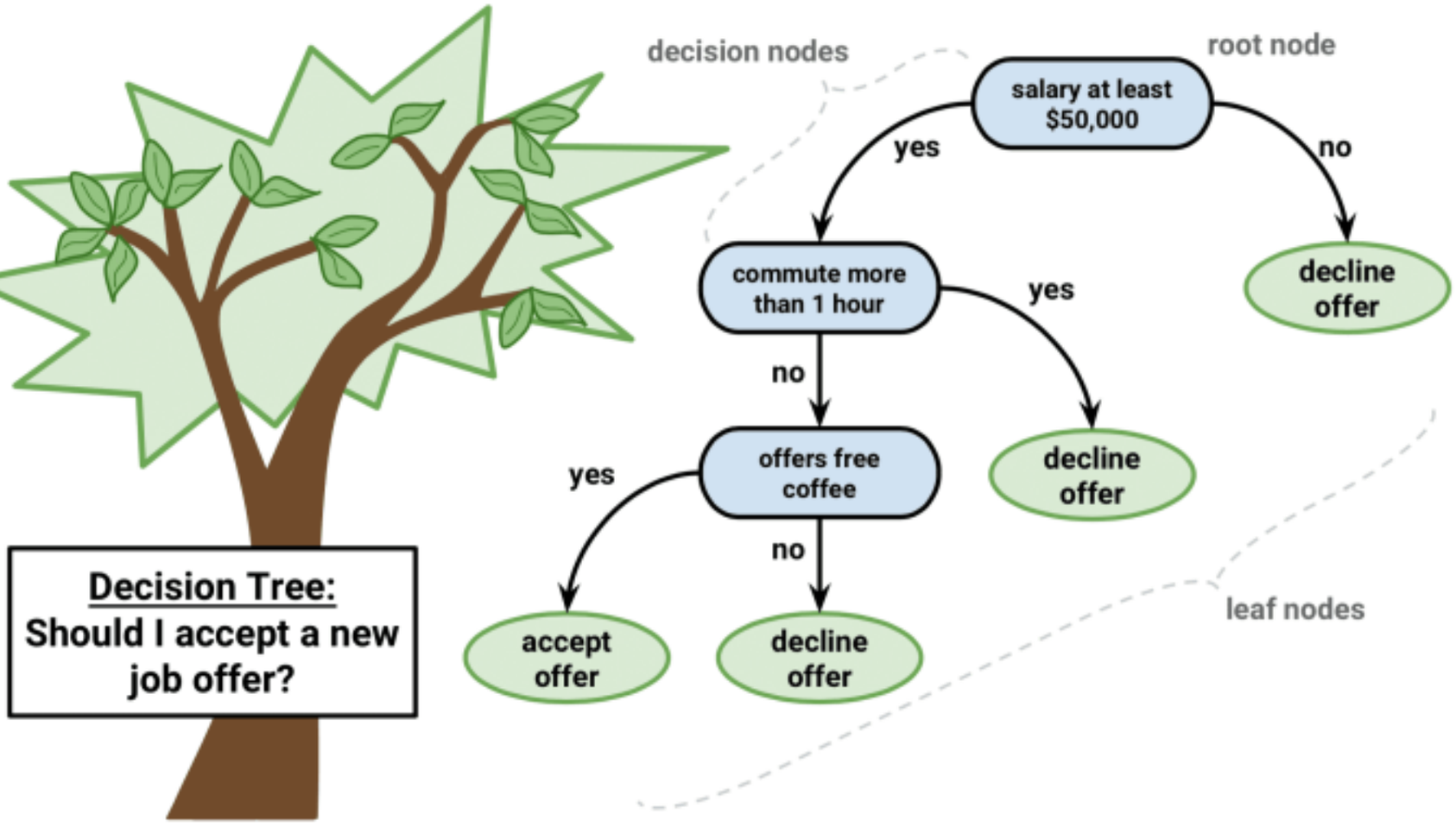


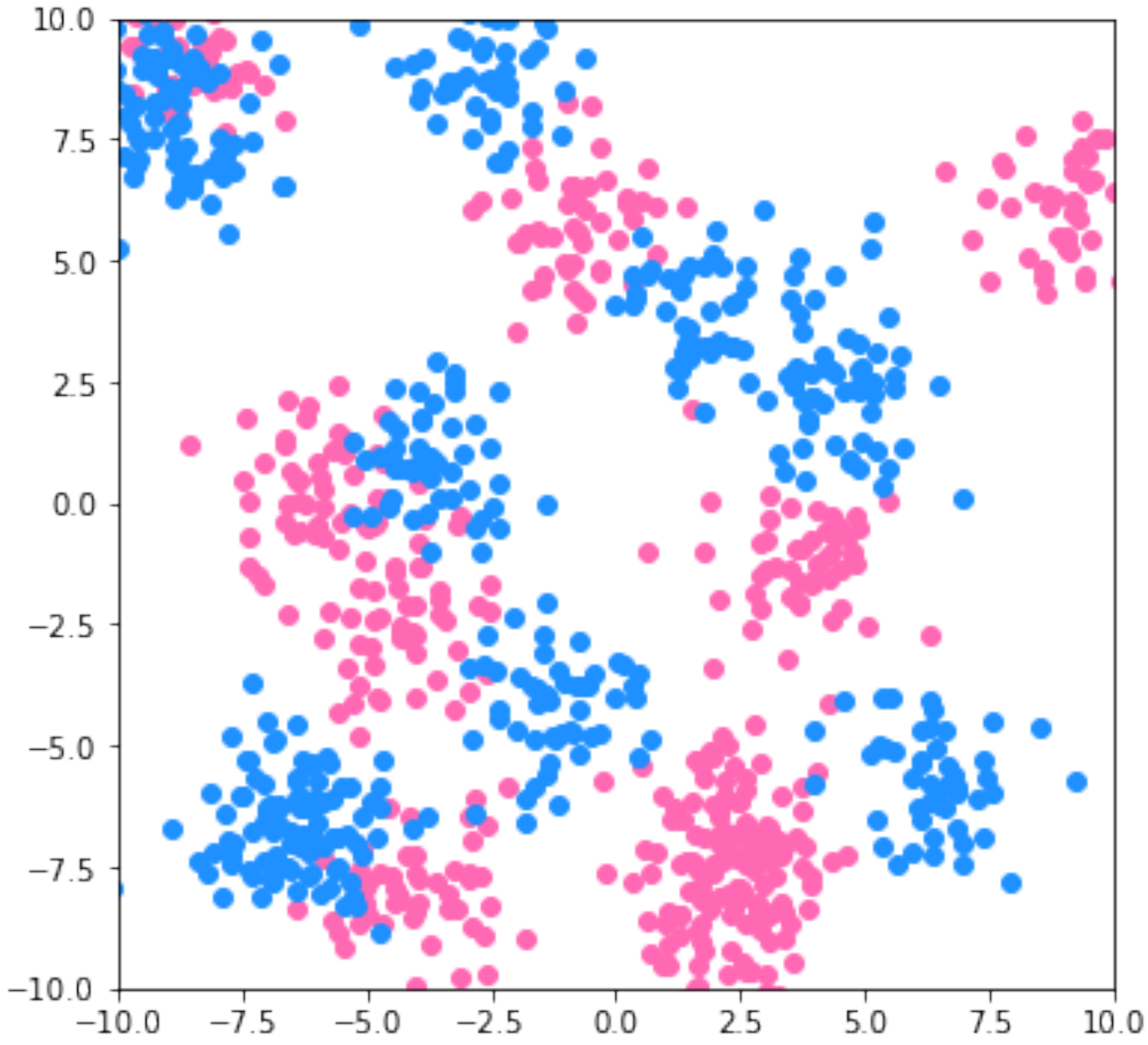

**Ozone data**

**Averaging models reduces overfitting!**

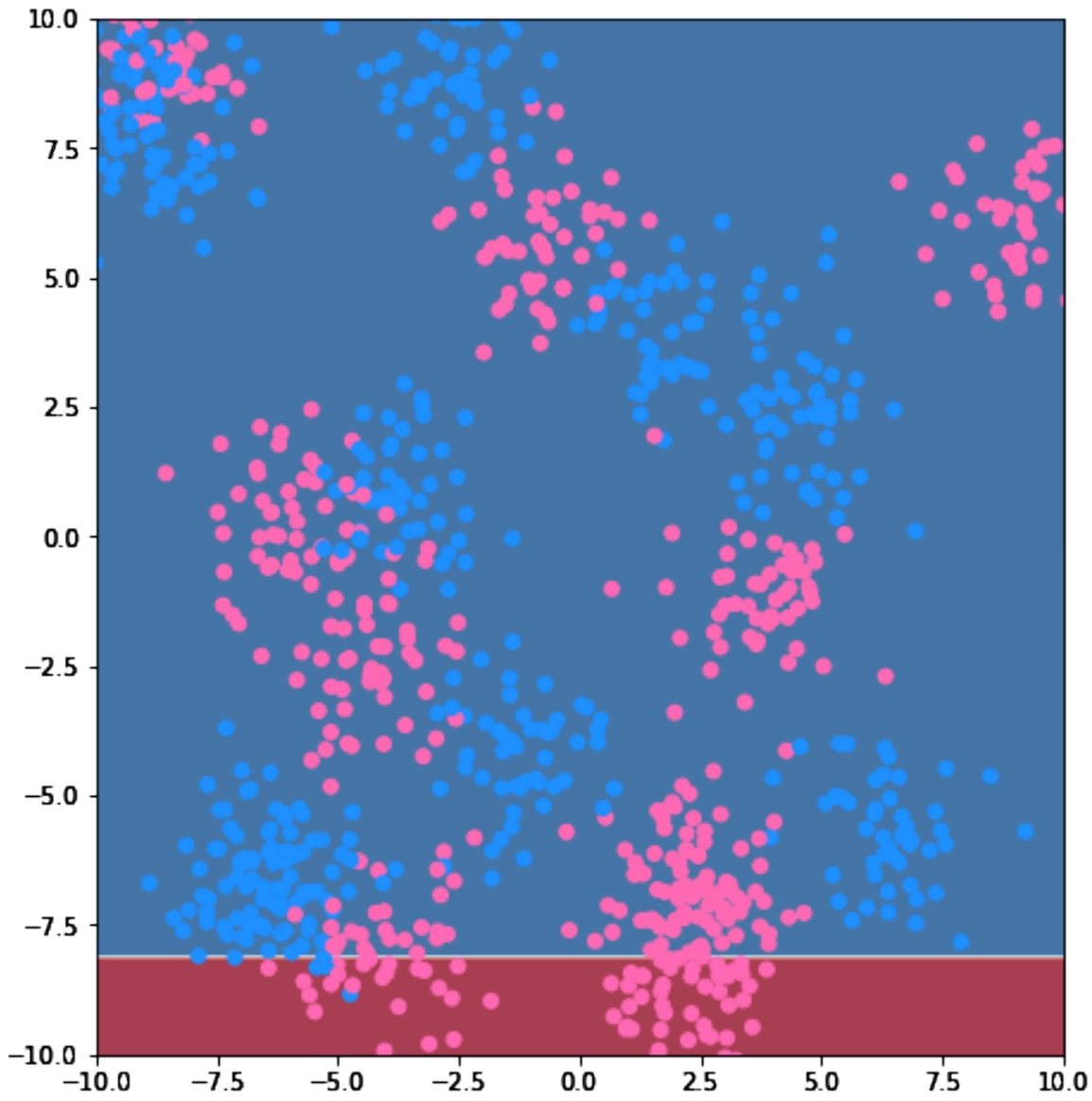
# And for classification?

# Decision trees!




10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

# Decision tree, depth=1



# How regions are decided

**Decision boundaries are decided by finding**

**a) one of the variable (i=1…d)**

**b) a particular value of this variable to cut between $x_i \geq v$ and $x_i \leq v$**

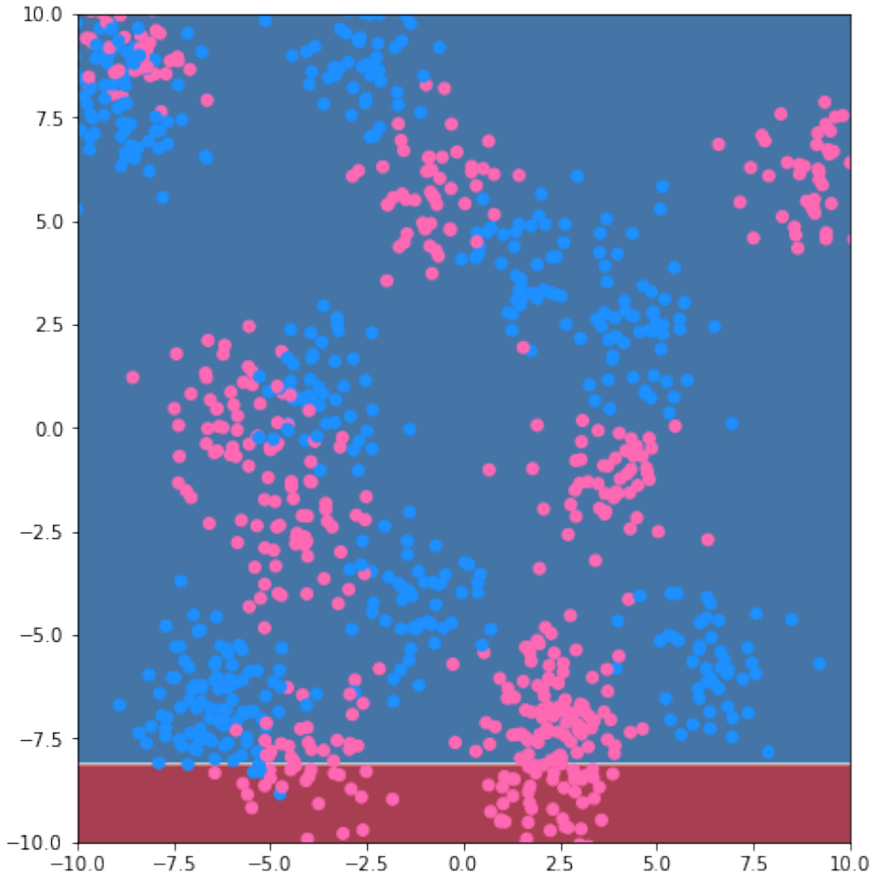
**<u>(a) & (b) are chosen to minimise the GINI criterion of all regions</u>**

### GINI Coefficient: Call

* $p_1(R)$ the fraction of points with label 1 in region R
* $p_2(R)$ the fraction of points with label 2 in region R

**The Gini impurity is given by** $Imp(R) = \sum_{i=1}^{2} p_i(R)(1 - p_i(R))$

*(Note: this is 0 if everyone is perfectly classified)*

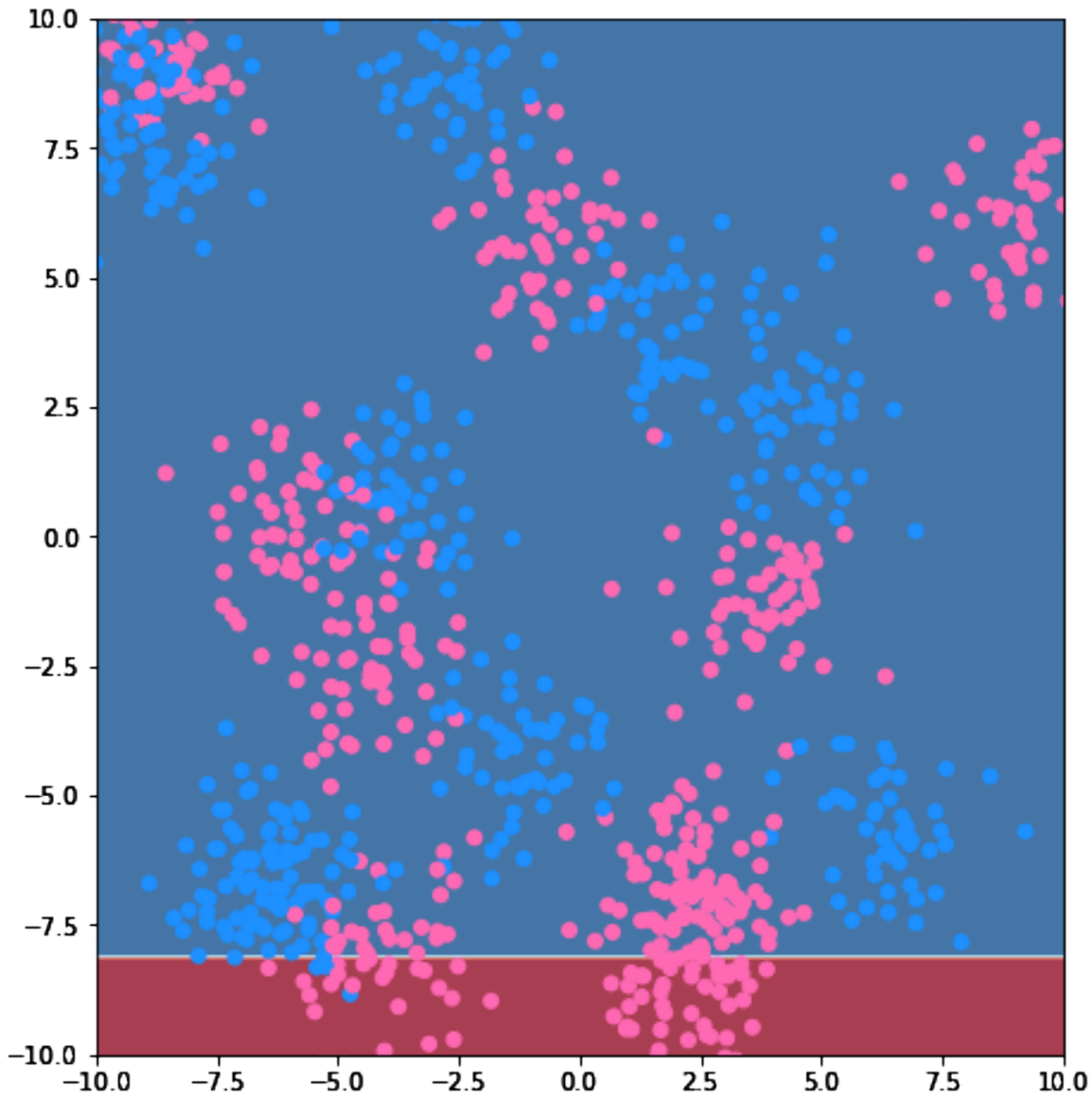



### Alternative possibilities

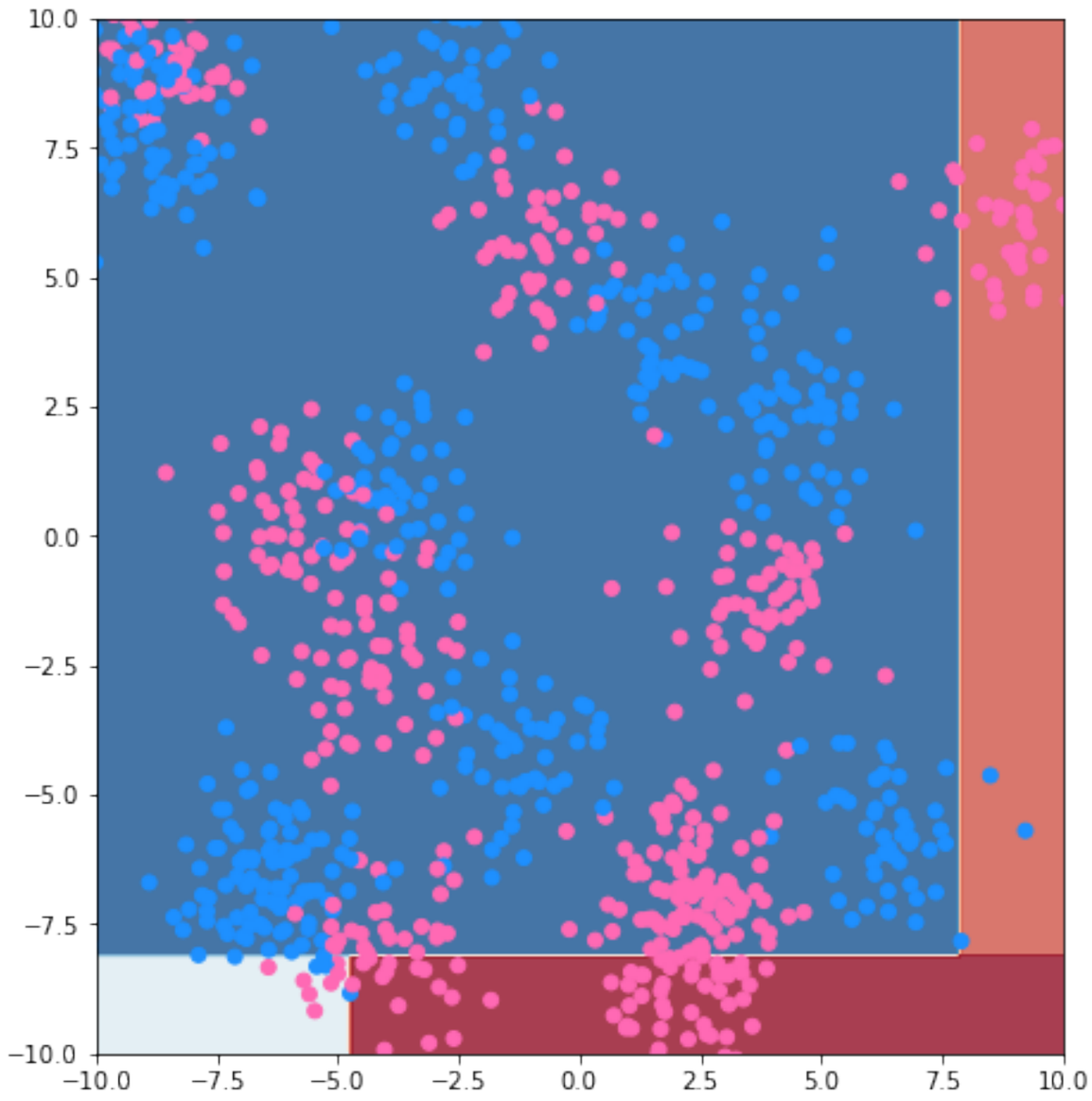
**Shannon entropy :** $Imp(R) = \sum_{i=1}^{2} - p_i(R)\log(p_i(R))$

**Misclassification error:** $Imp(R) = 1 - \max_i p_i(R)$

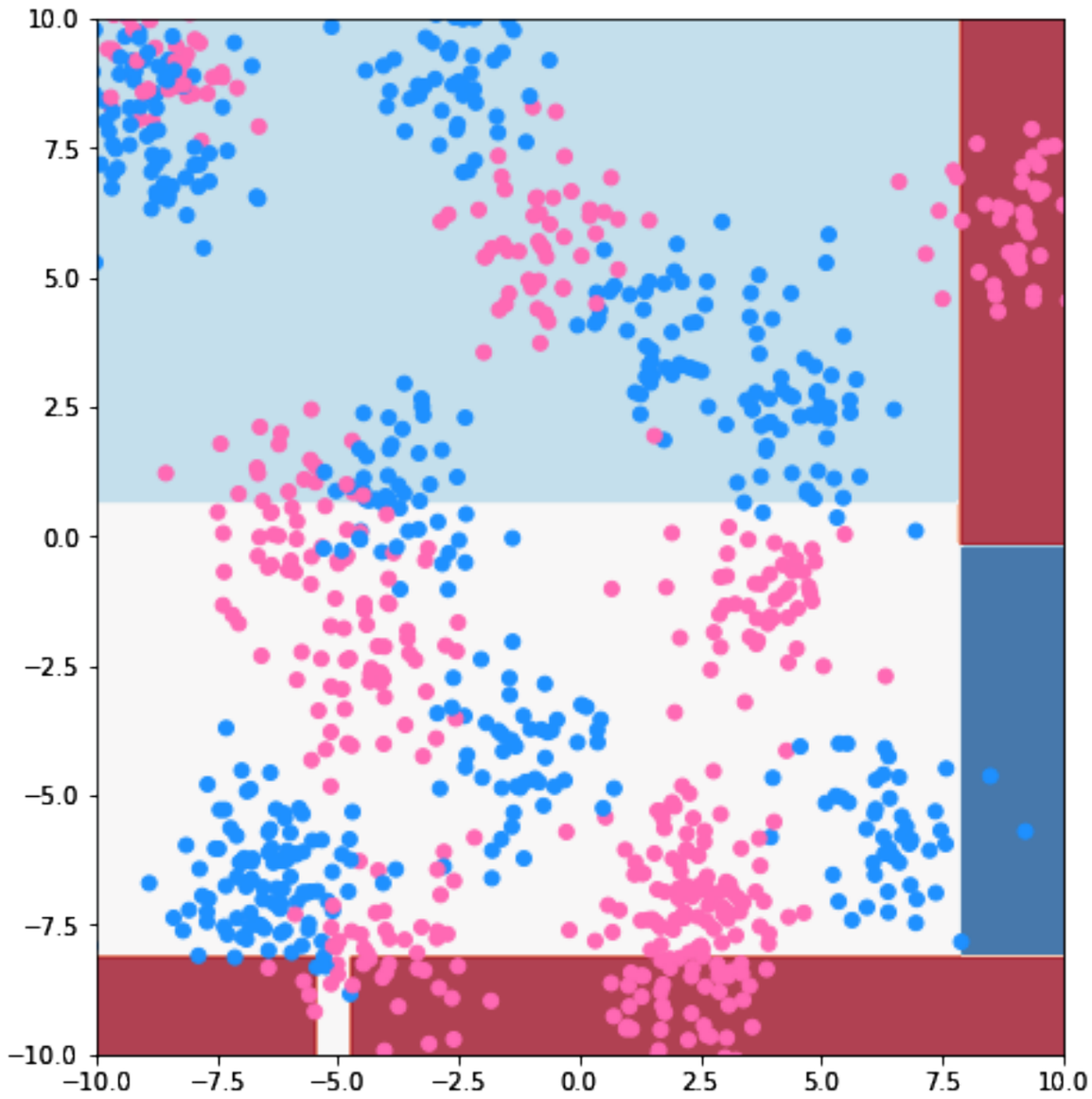
# Decision tree, depth=1



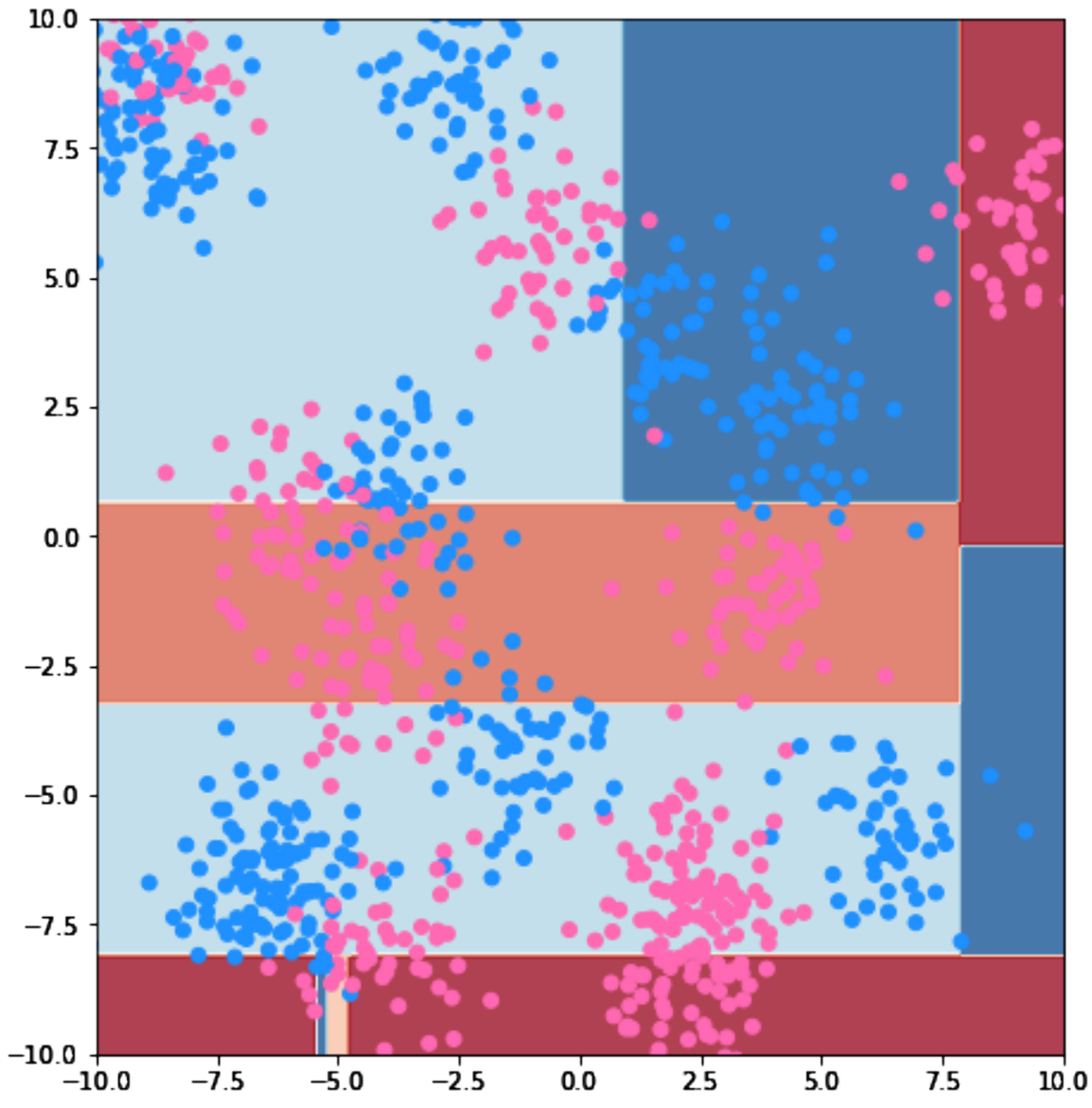
# Decision tree, depth=2



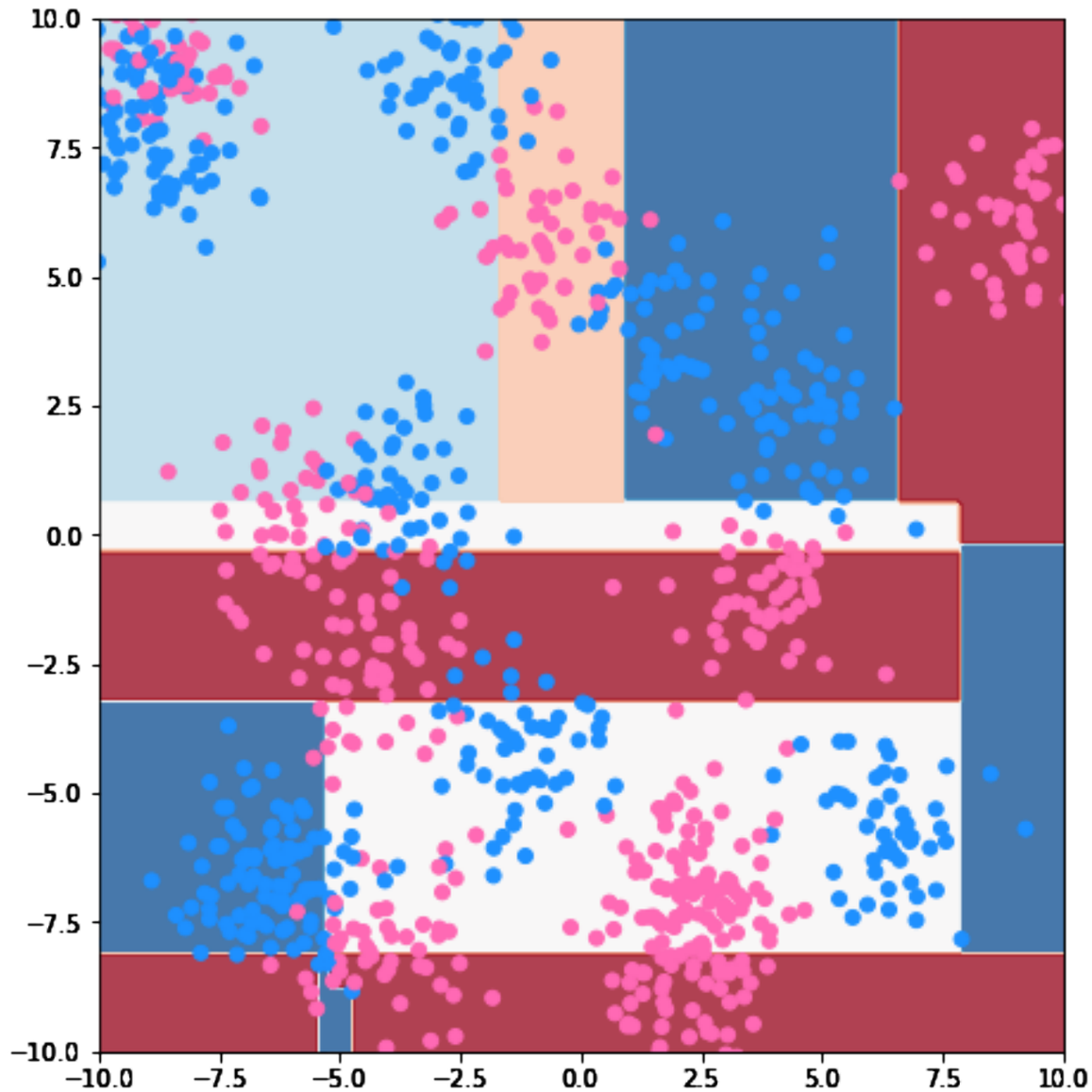
# Decision tree, depth=3



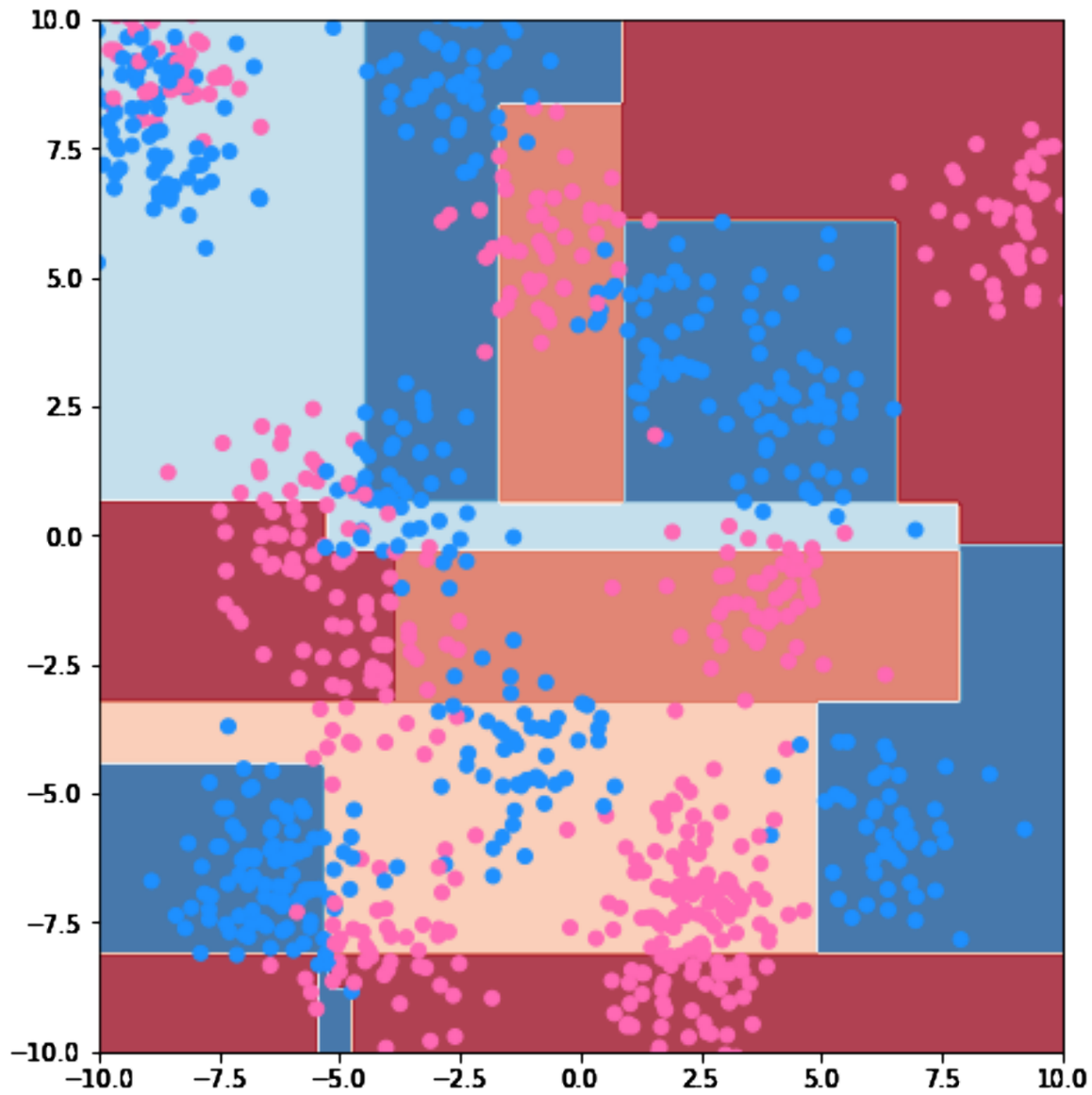
# Decision tree, depth=4



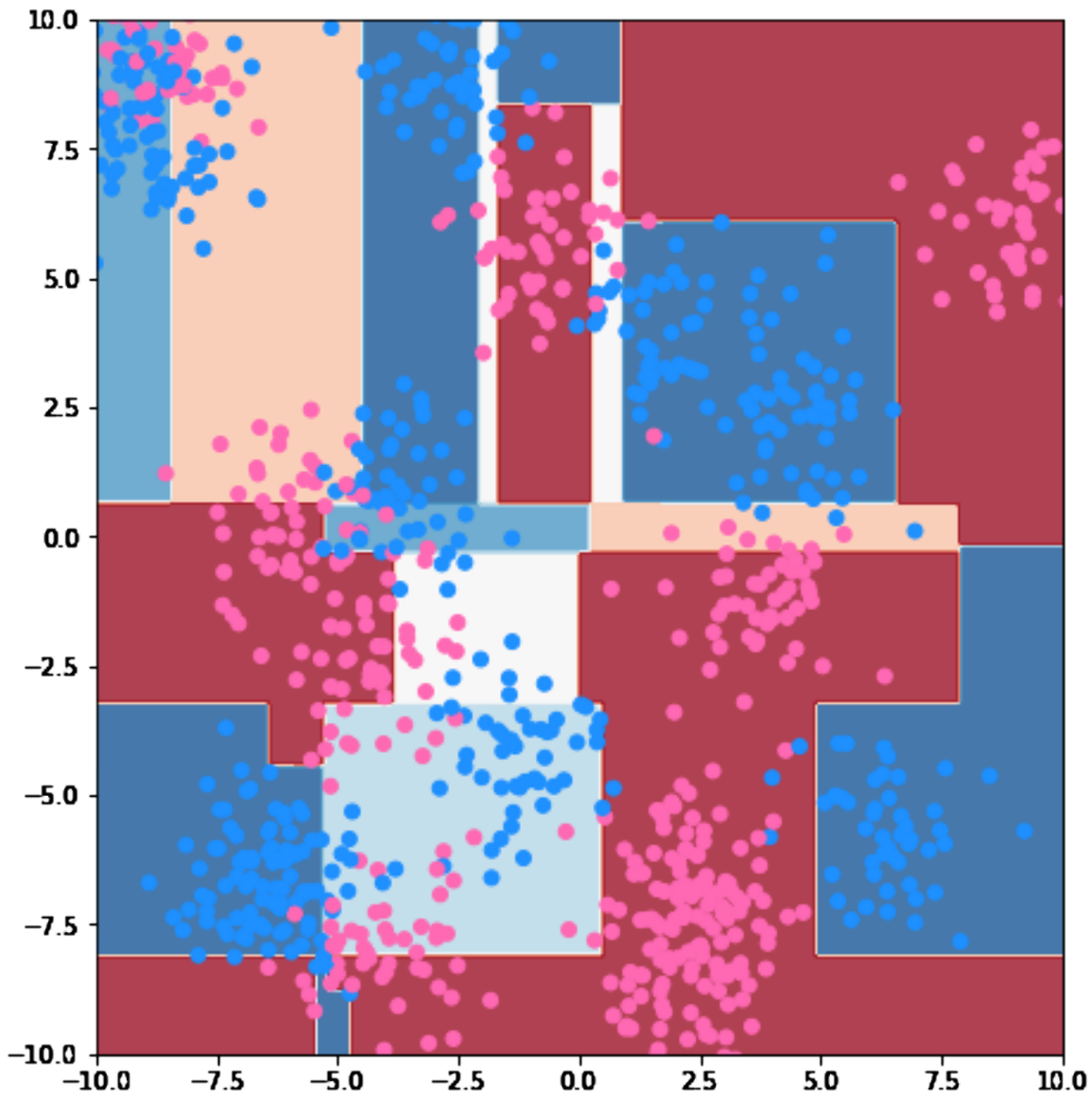
# Decision tree, depth=5



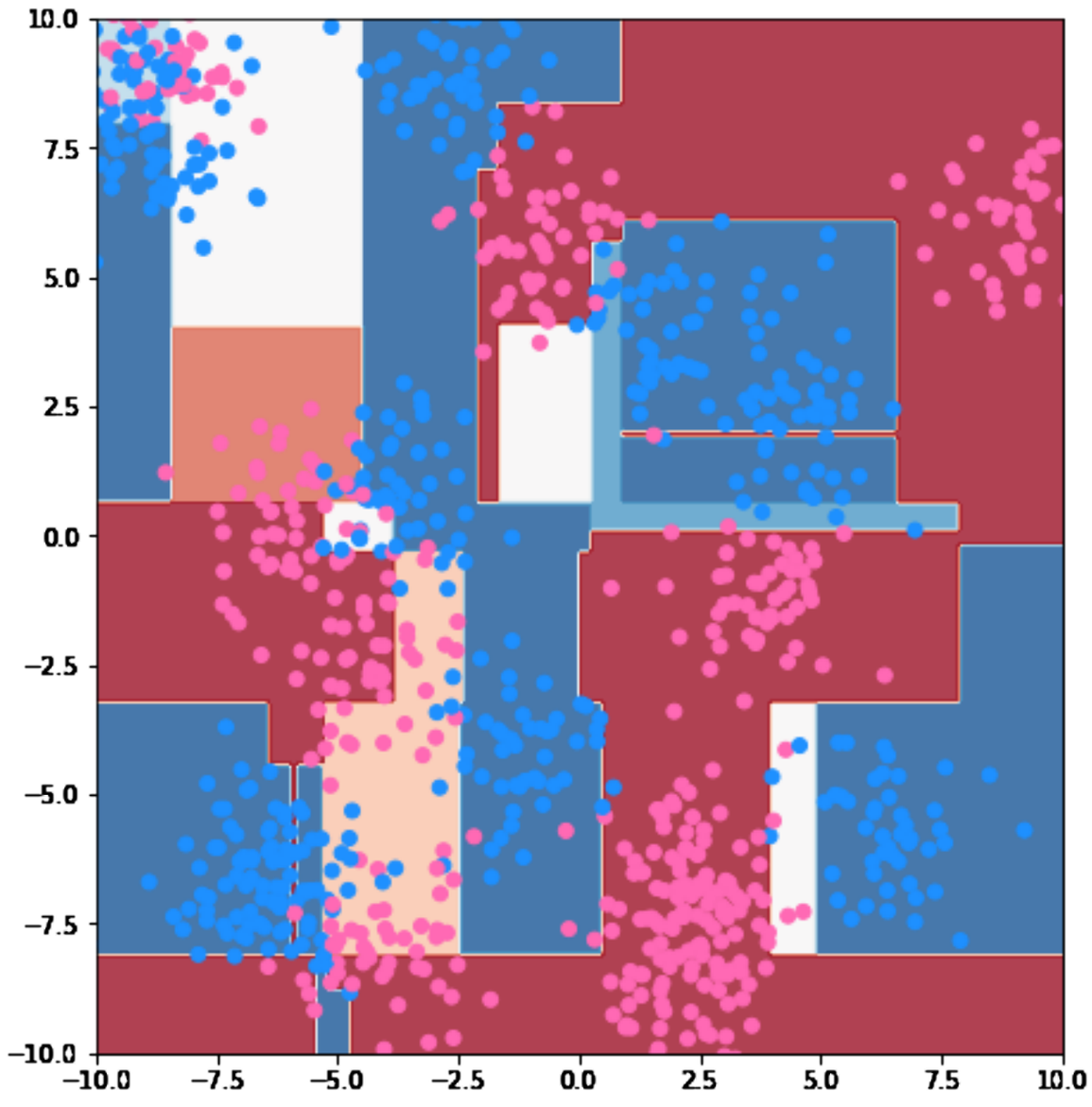
# Decision tree, depth=6



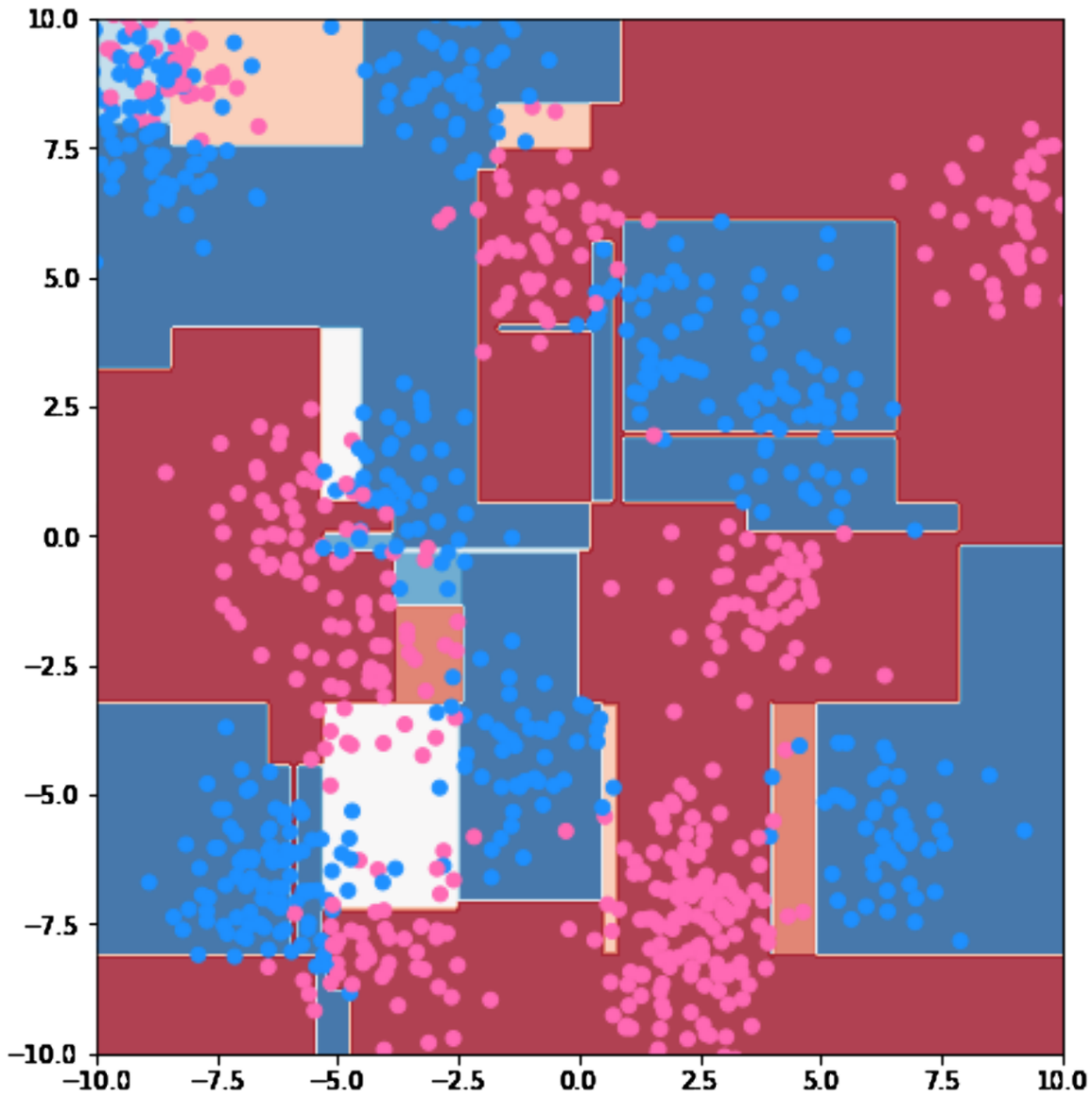
# Decision tree, depth=7



# Decision tree, depth=8



# Decision tree, depth=9



# Bagging

"Bootstrap Aggregation"

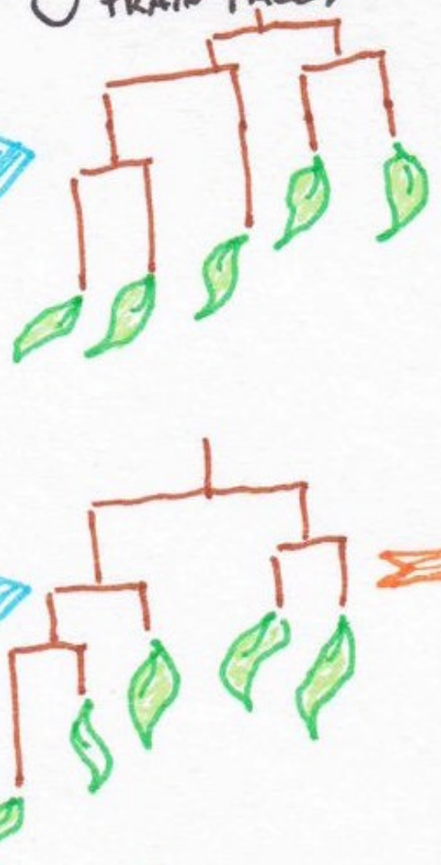
DATASET

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

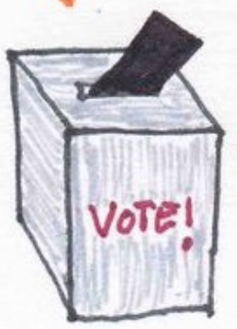
RESAMPLE W/ REPLACEMENT

| | | | | |
|---|---|---|---|---|
| 3 | 2 | 4 | 4 | 2 |
| 1 | 3 | 1 | 1 | 1 |
| 2 | 1 | 1 | 3 | 3 |
| 3 | 1 | 3 | 4 | 4 |

TRAIN TREES



ALL TREES VOTE

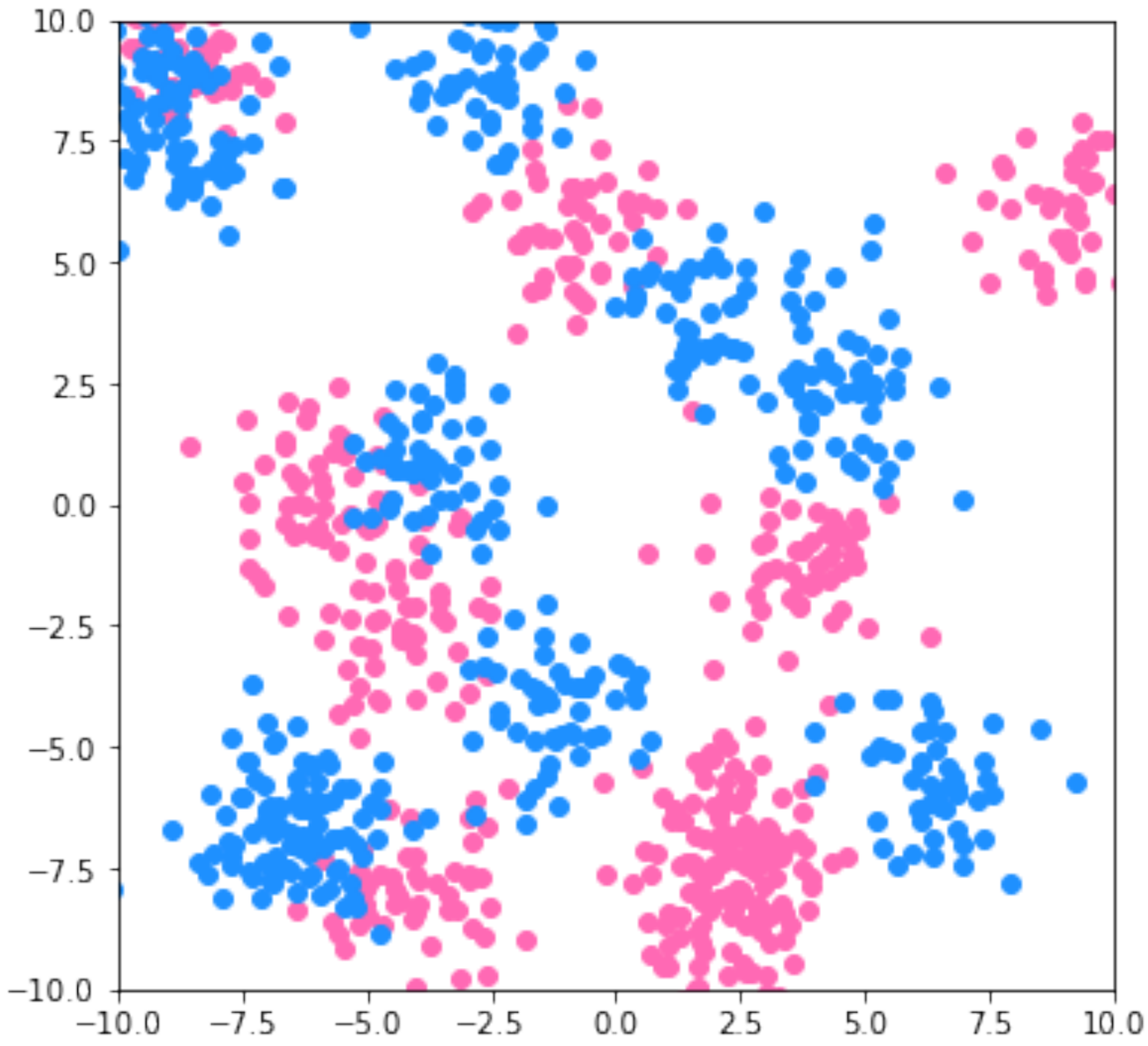



OUTCOME IS PREDICTION

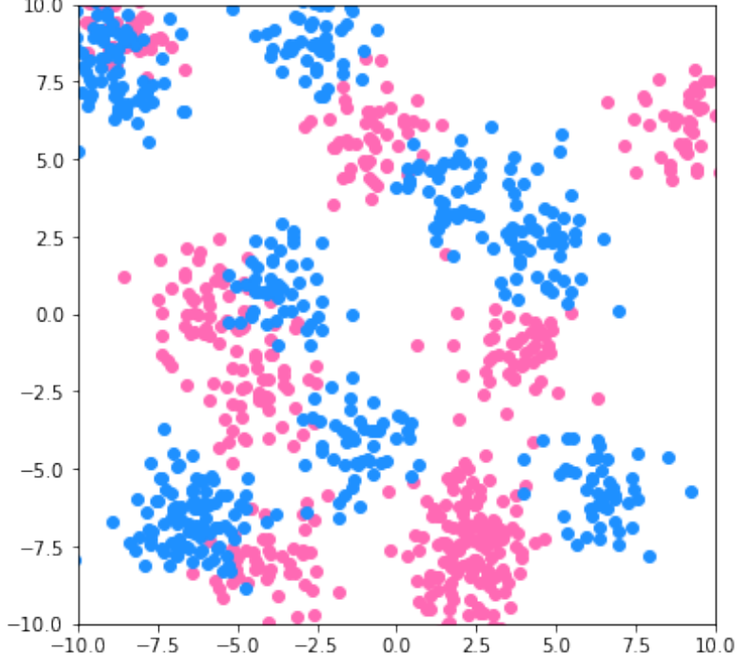
Prediction

Note: Random Forests is bagging but random sets of features in addition to bootstrapped samples

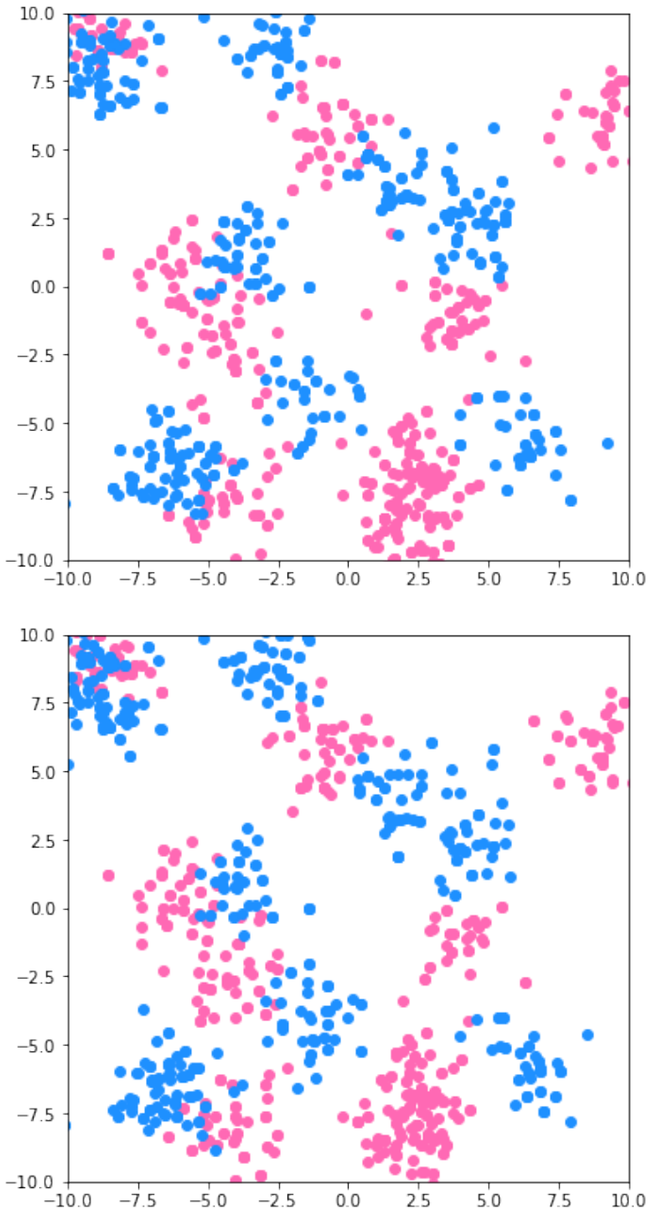
# The original set

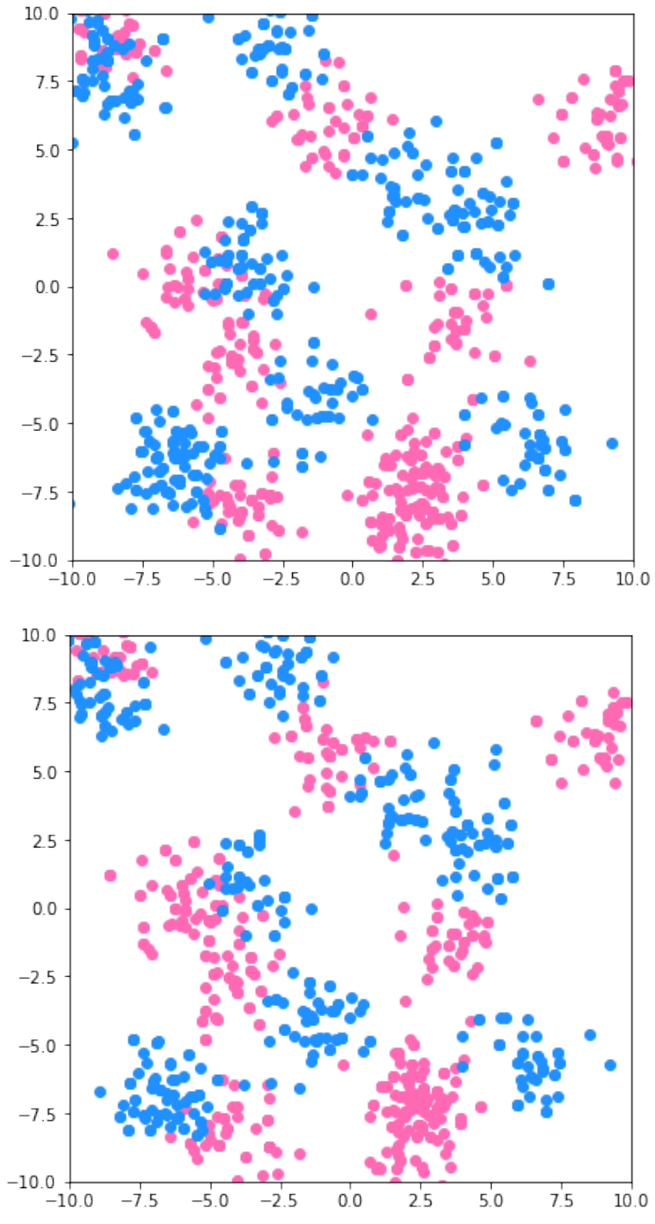


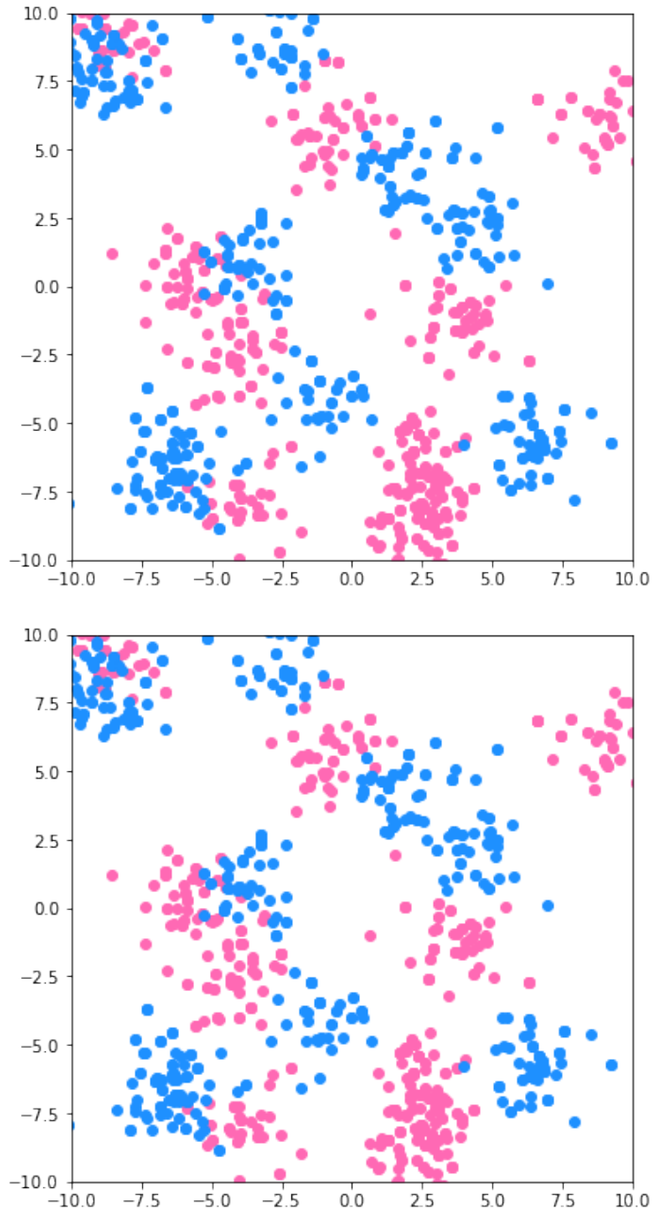
# BOOSTRAP!!



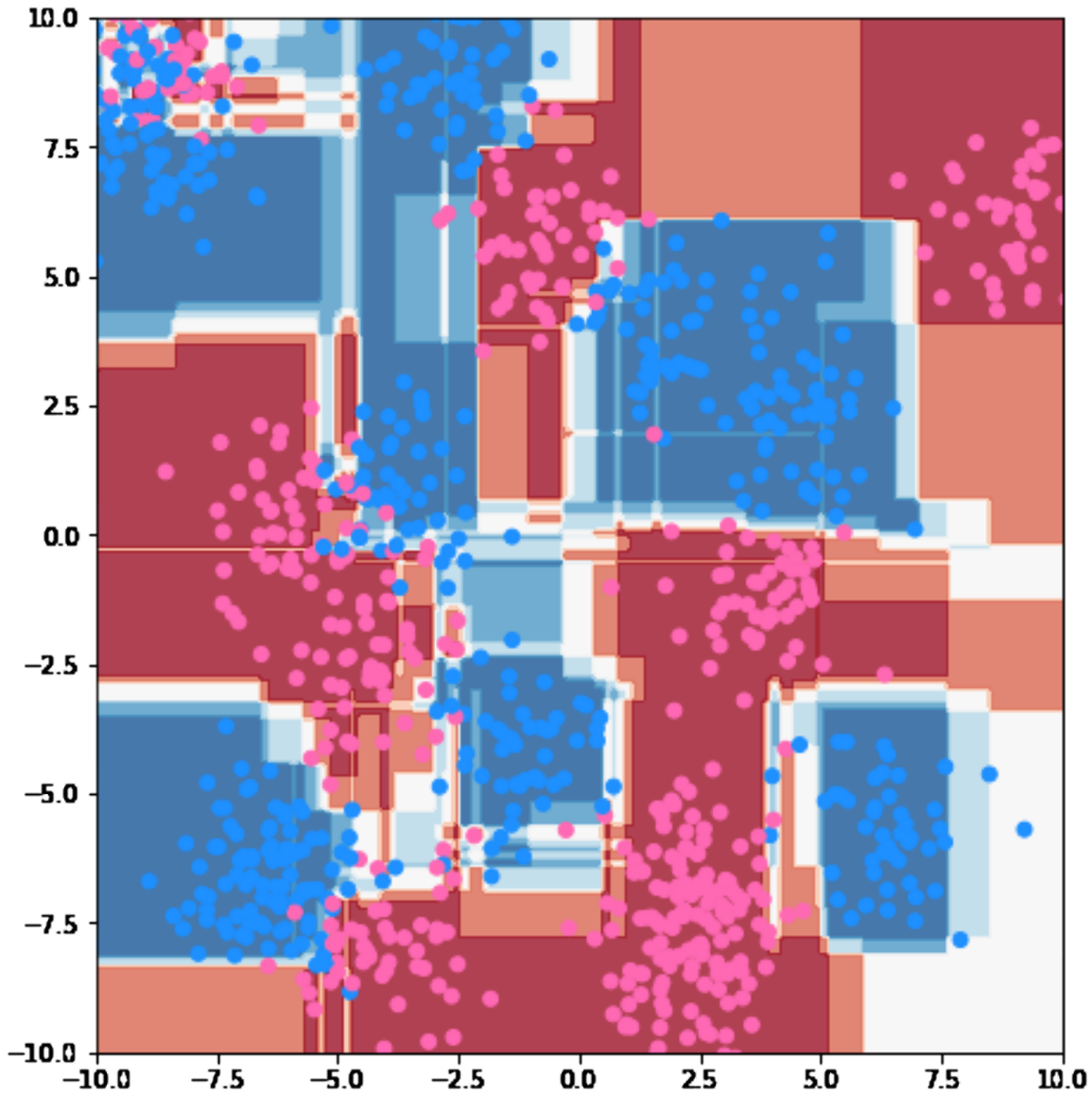
# BOOSTRAP!!



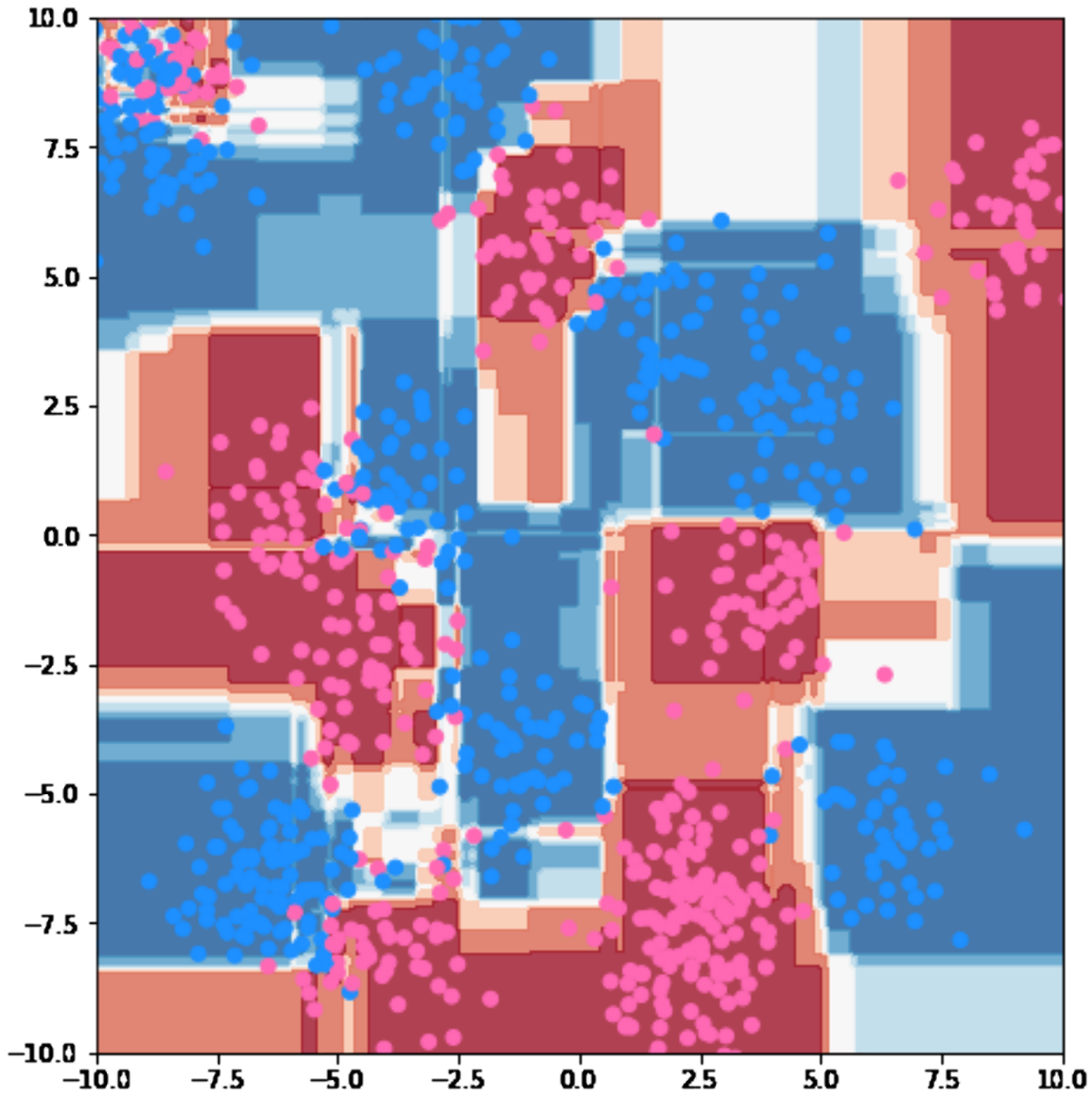




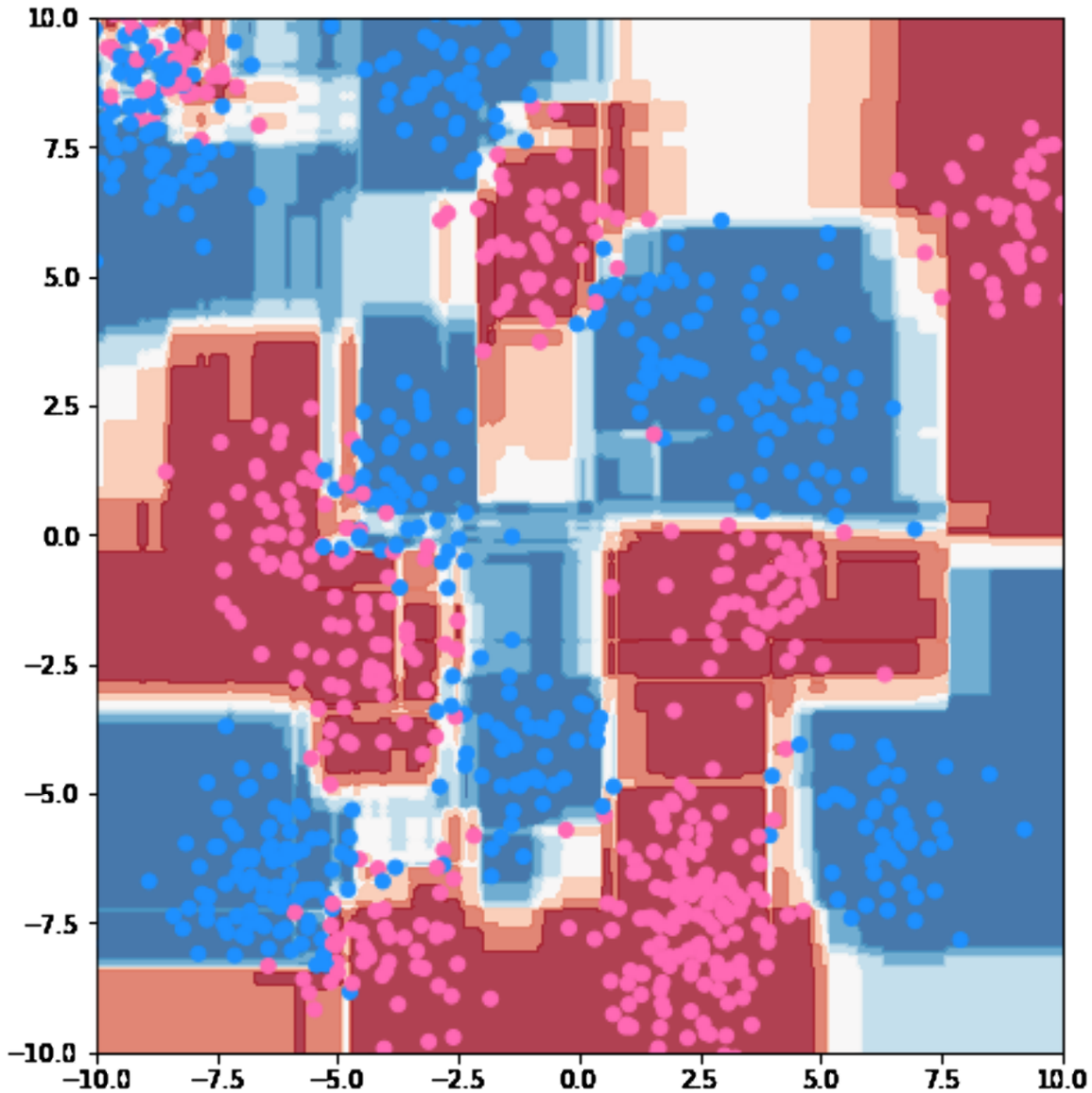
# Bagging 5 Decision tree!



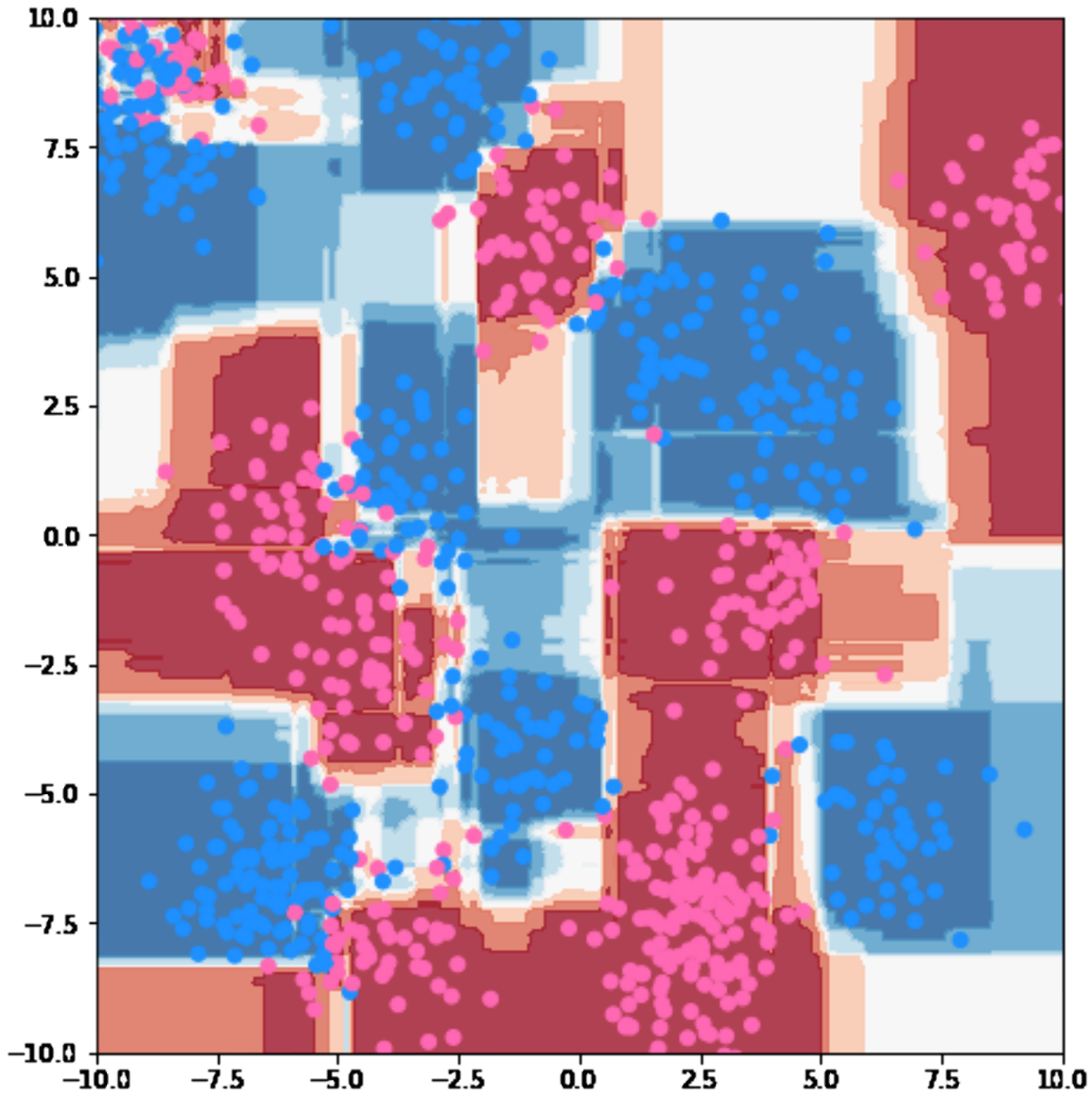
# Bagging 10 Decision tree!



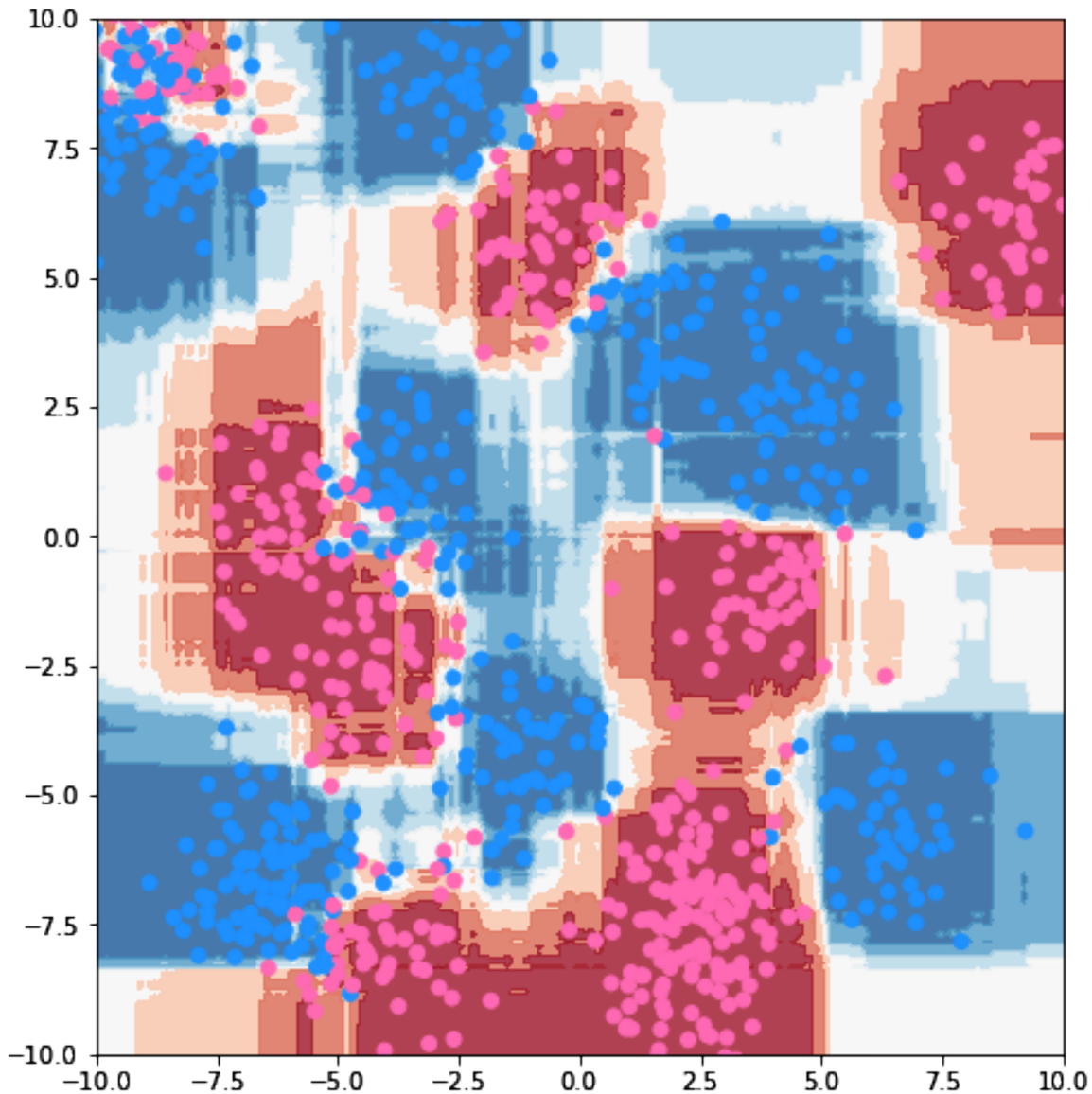
# Bagging 20 Decision tree!



# Bagging 20 Decision tree!



# In practice: Random Forrest




Further randomization: at each candidate split in the learning process, a random subset of the features. For a classification problem with $p$ features, $\sqrt{p}$ **(rounded down)** features are used in each split

# Sk-learn!

## sklearn.ensemble.RandomForestClassifier

*class* sklearn.ensemble.RandomForestClassifier(*n_estimators=100*, ***, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*, *ccp_alpha=0.0*, *max_samples=None*)

[source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Read more in the User Guide.

# Boosting

**Can we make many dumb learners smart?**

**Yes (but you need a smart leader!)**

# Can we make many dumb learners smart?

Kearns and Valiant ’88:
- Does weak learnability imply strong learnability?
In other words, can we boost from weak to strong?

Schapire ’89
Freund and Schapire ’95
- Yes, with adaBoost

# How to combine many weak classifiers?

## Ingredients:

(i) T classifiers $h_t(.)$, each of them slightly better than random
(ii) A training set with N labeled examples

**Adaptive linear combination of classifiers**

$$H(\vec{x}) = \sum_{t=1}^{T} \alpha_t h_t(\vec{x})$$

ADABOOST

**Exponential loss:**

$$\mathcal{R} = \sum_i e^{-Y_i H(\vec{x}_i)}$$

## Goal: start from t=0, add new classifiers and

(i) Adapt the weight alpha at each steps
(ii) Re-weight the instances in the training set : more weight to ill-classified instances
(each new classifier concentrate on badly classified examples)

# How does one set the weight and α at each time steps?

**Assume we have done the job until time τ !**

$$\mathcal{R} = \sum_i e^{-Y_i H(\vec{x}_i)} \qquad H(\vec{x}) = \sum_{t=1}^{T} \alpha_t h_t(\vec{x})$$

$$\mathcal{R} = \sum_i e^{-Y_i \overbrace{\sum_{t=1}^{\tau-1} \alpha_t h_t(\vec{x}_i)}^{\lambda_i^\tau} - Y_i \alpha_\tau h_\tau(\vec{x}_i)}$$

$$\mathcal{R} = \sum_i \underbrace{e^{-Y_i \lambda_i^\tau}}_{\omega_i^\tau} e^{-Y_i \alpha_\tau h_\tau(\vec{x}_i)}$$

$$\mathcal{R} = \sum_i \omega_i^\tau e^{-Y_i \alpha_\tau h_\tau(\vec{x}_i)}$$

$\omega_i^\tau$ **= weights for each instances at time τ**

# How does one set the weight and α at each time steps?

**Assume we have done the job until time τ !**

$$\mathcal{R} = \sum_i \omega_i^\tau e^{-Y_i \alpha_\tau h_\tau(\vec{x}_i)}$$

$$\mathcal{R} = e^{-\alpha_\tau} \sum_{i \in \mathrm{OK}} \omega_i^\tau + e^{\alpha_\tau} \sum_{i \in \mathrm{NOT\ OK}} \omega_i^\tau = e^{-\alpha_\tau} \sum_i \mathbf{1}(Y_i = h_i)\omega_i^\tau + e^{\alpha_\tau} \sum_i \omega_i^\tau \mathbf{1}(Y_i \neq h_i)$$

$$\mathscr{R} = e^{-\alpha_t} \underbrace{\left(1 - \sum_i \omega_i \mathbf{1}(Y_i \neq h_i)\right)}_{\epsilon_t} + e^{\alpha_t} \underbrace{\left(\sum_i \omega_i \mathbf{1}(Y_i \neq h_i)\right)}_{\epsilon_t}$$

$\epsilon_t$ **is what the classifier $h_t(.)$ is trying to minimise**

$$\mathcal{R} = e^{-\alpha_\tau} - e^{-\alpha_\tau}\epsilon_t + e^{\alpha_\tau}\epsilon_t = e^{-\alpha_\tau}(1 - \epsilon_t) + e^{\alpha_\tau}\epsilon_t$$

**Let us choose $\alpha_t$ in order to minimize the global risk**

$$\partial_{\alpha_t}\mathcal{R} = 0 \rightarrow \alpha_t = \frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}$$

# ADABOOST TRAINING

$$\forall\ i : \omega_i^{t=1} = \frac{1}{n}$$

$$\text{for t} = 1, \ldots, \text{T}_{\max} \qquad \textbf{Do}$$

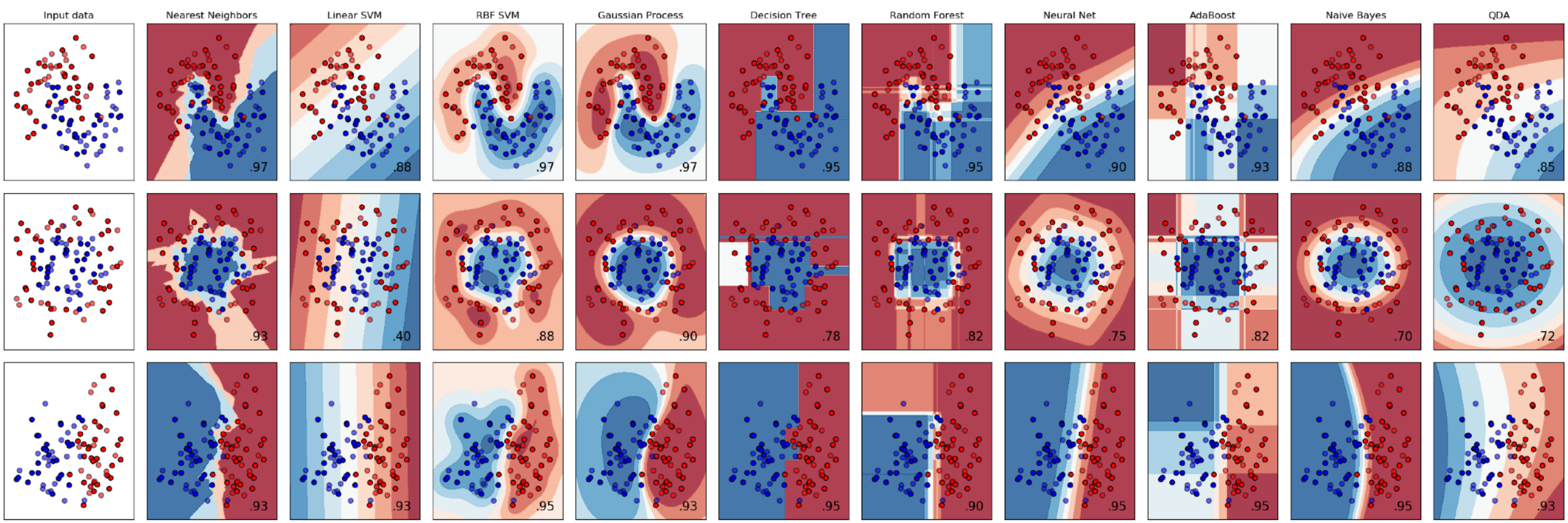
**Run a classifier for :** $\epsilon_t = \left[\sum_i \omega_i^{\tau} \mathbf{1}(Y_i \neq h_i))\right]$

**Set:** $\alpha_t = \frac{1}{2} \log \frac{1 - \epsilon_t}{\epsilon_t}$

**Agregate classifier:** $H(.) = H^{t-1}(.) + \alpha_t h_t(.)$

**Update weights:** $\omega_i^{t+1} = \frac{\omega_i^t e^{-Y_i \alpha_t h_t(\vec{x}_i)}}{\sum_i \omega_i^t e^{-Y_i \alpha_t h_t(\vec{x}_i)}}$

# A tour on standard machine learning classifiers



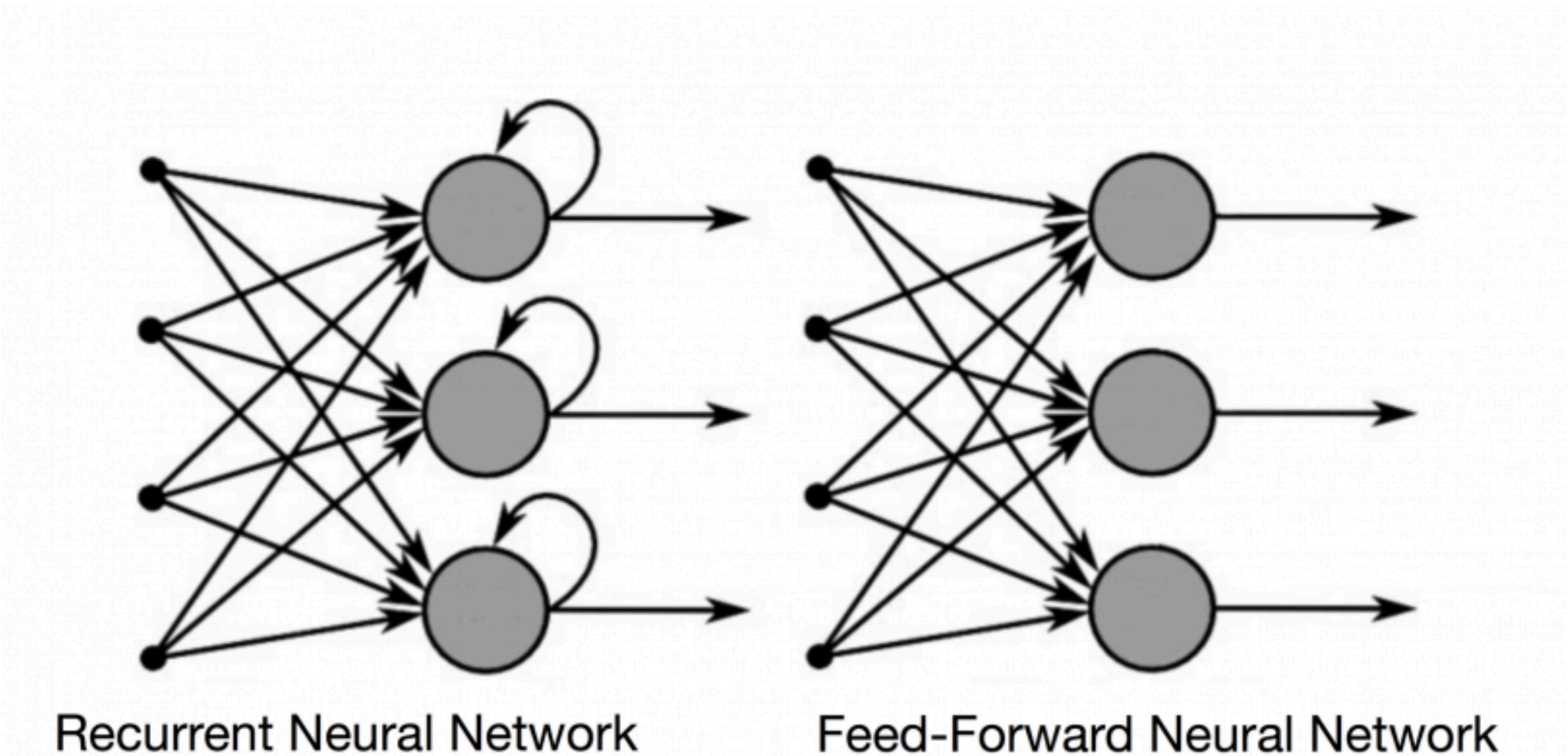

# From sk-learn

# Achitectures

## Recurrent Neural Networks

**Times series**

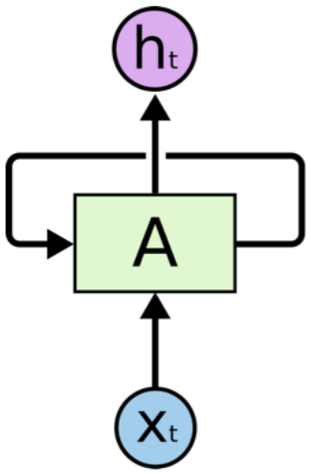
# RNN vs Feedfoward



Recurrent Neural Network

Feed-Forward Neural Network

# Recurent Neural Networks

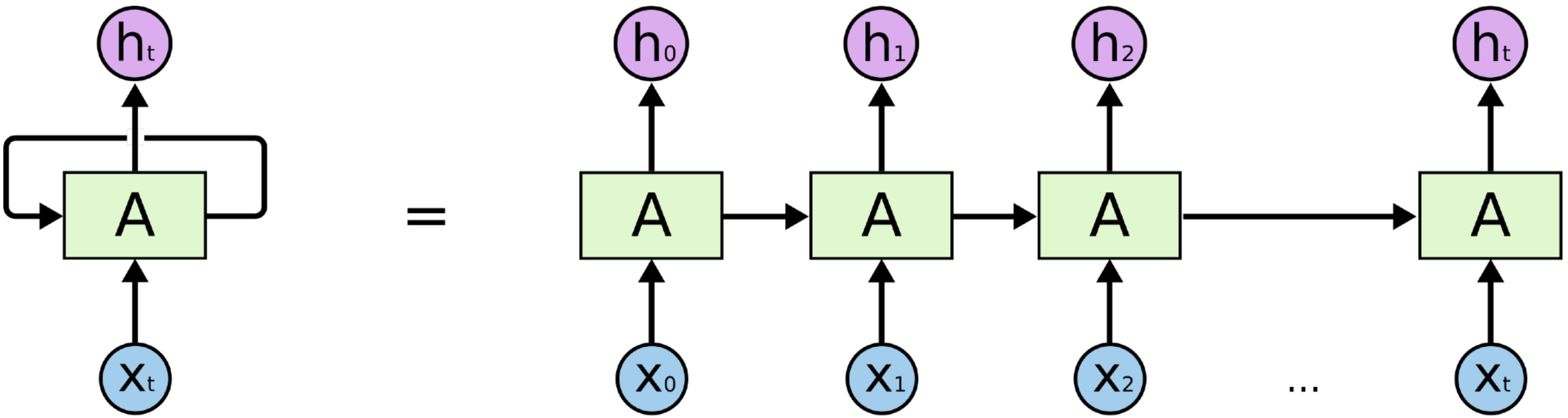



**A simple RNN:**

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

**Learn W,U,b such that $h_t$ gives what you want!**

# Recurent Neural Networks



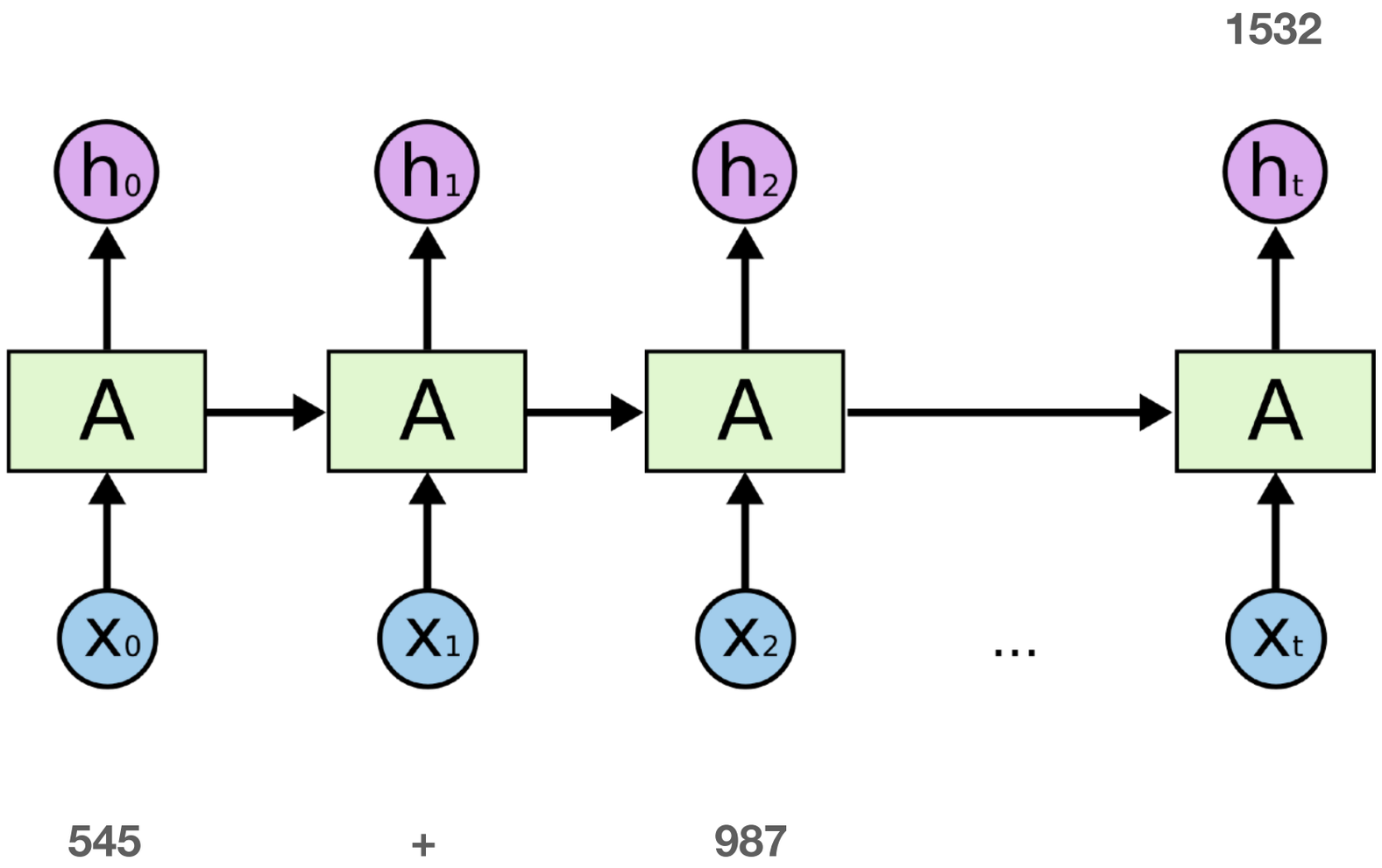

**A simple RNN:**

$$h_t = \tanh(W x_t + U h_{t-1} + b)$$

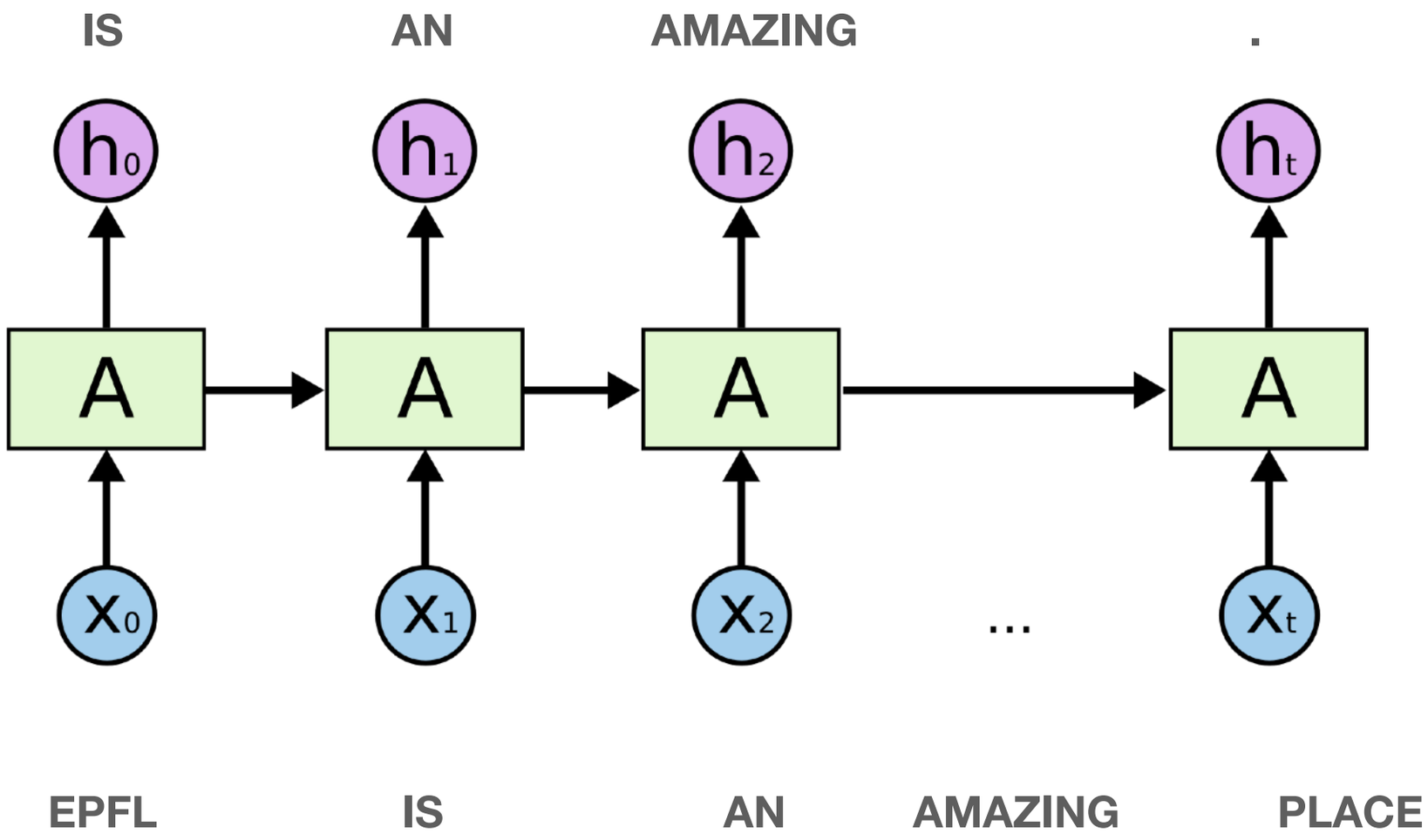
**Learn W,U,b such that $h_t$ gives what you want!**

*Slide credit: Christopher Olah*

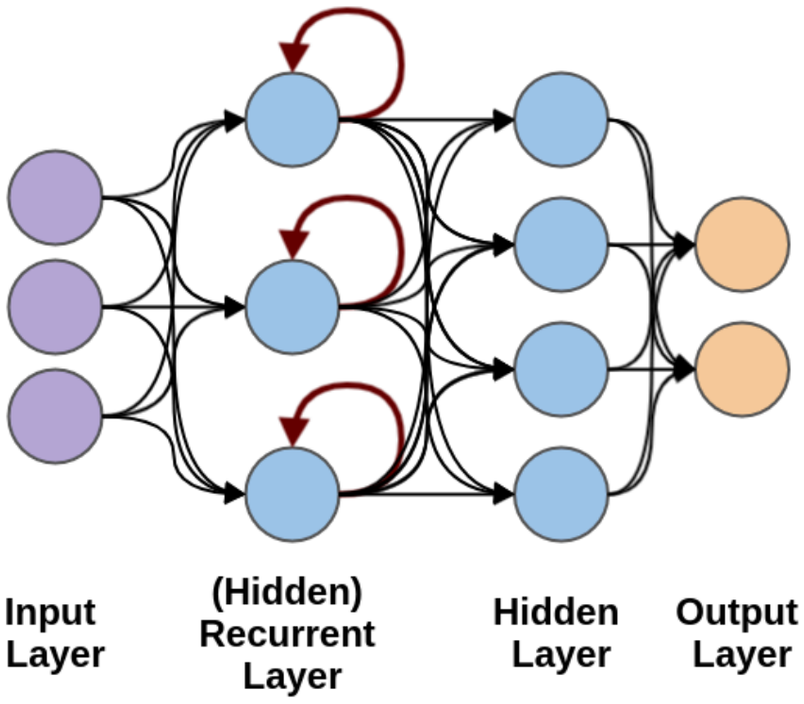
# Recurent Neural Networks



# Recurent Neural Networks

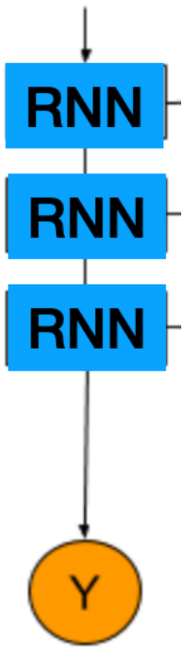


# Recurent Neural Networks



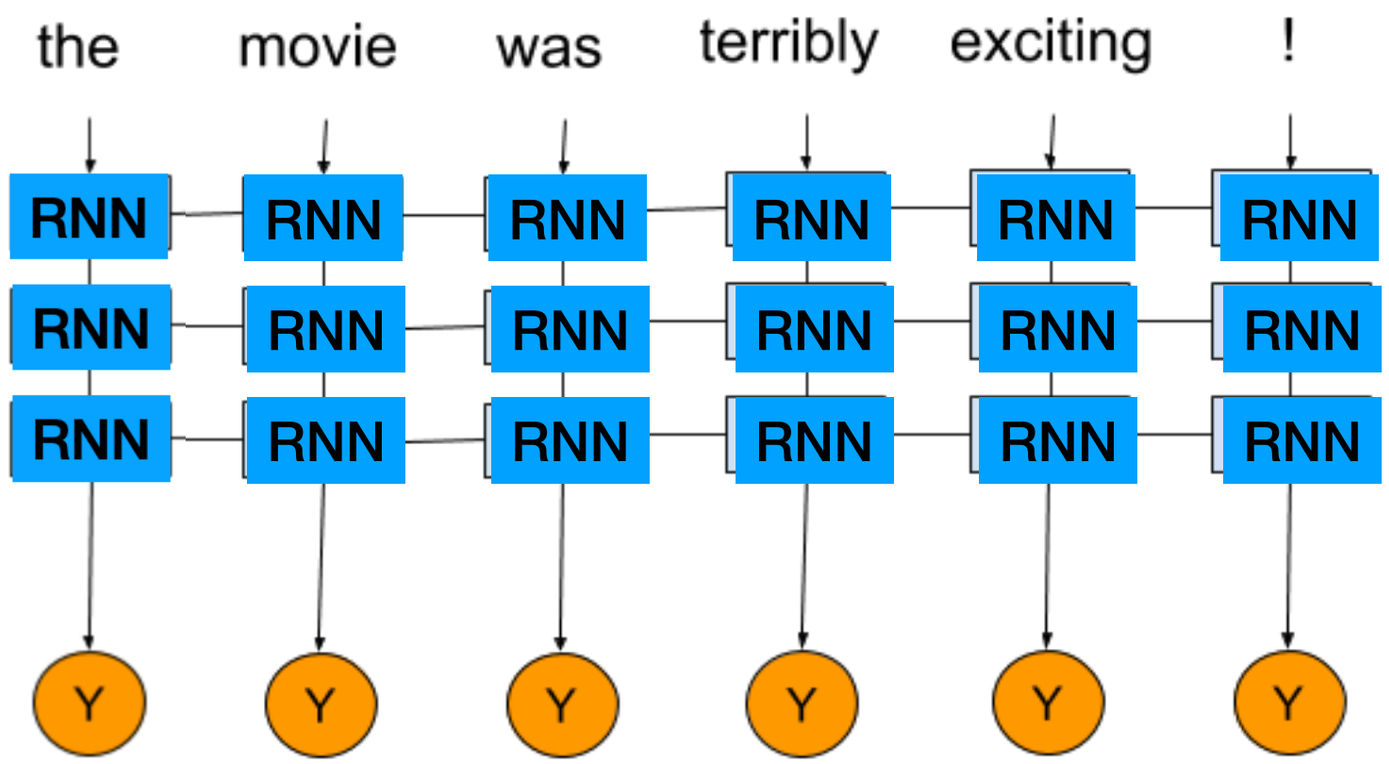

*"This is a stupid example." – Statement*
*"This is another statement, perhaps this will trick the network" – Statement*
*"I don't understand" – Statement*
*"What's up?" – Question*
*"open the app" – Command*
*"This is another example" – Statement*
*"Do what I tell you" – Command*
*"come over here and listen" – Command*
*"how do you know what to look for" – Question*
*"Remember how good the concert was?" – Question*
*"Who is the greatest basketball player of all time?" – Question*
*"Eat your cereal." – Command*
*"Usually the prior sentence is not classified properly." – Statement*
*"Don't forget about your homework!" – Command*

# Multi-Layer RNN



# Multi-Layer RNN

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Embedding(vocab_size, 16))
model.add(tf.keras.layers.LSTM(32,return_sequences=Tru
e))
model.add(tf.keras.layers.LSTM(32,return_sequences=Tru
e))
model.add(tf.keras.layers.LSTM(32))
model.add(tf.keras.layers.Dense(1,
activation=tf.nn.sigmoid))

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['acc'])
```

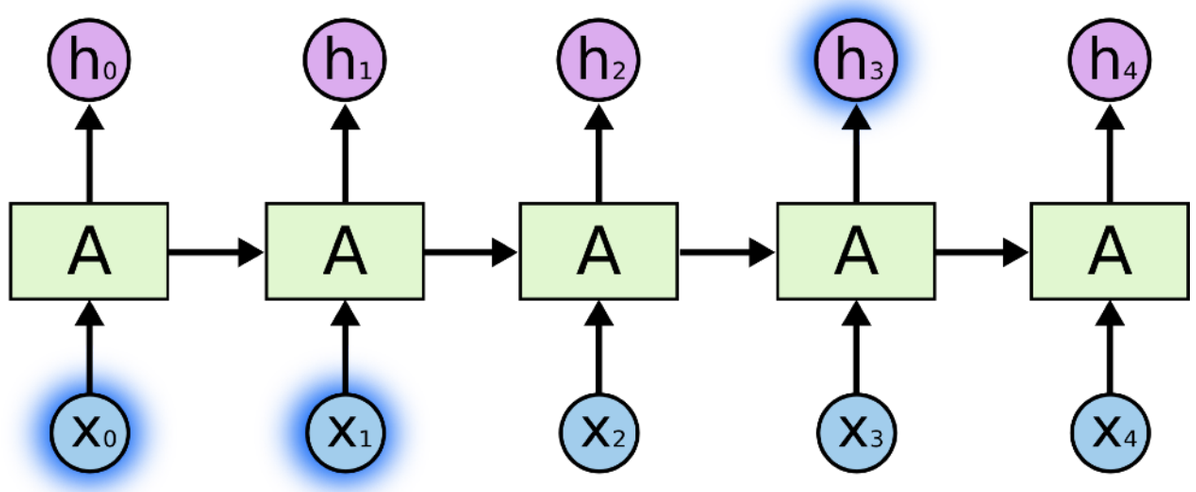



$$h_t^{(i)} = \tanh(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)})$$

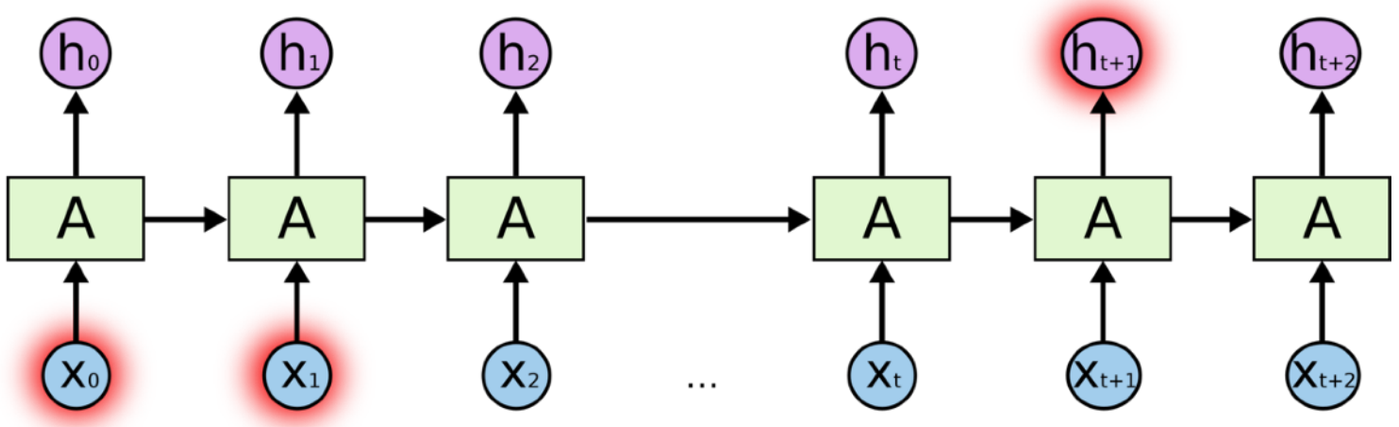
Learn $W^{(i)}, U^{(i)}, b^{(i)}$

# Recurent Neural Networks

## The Problem of Long-Term Dependencies



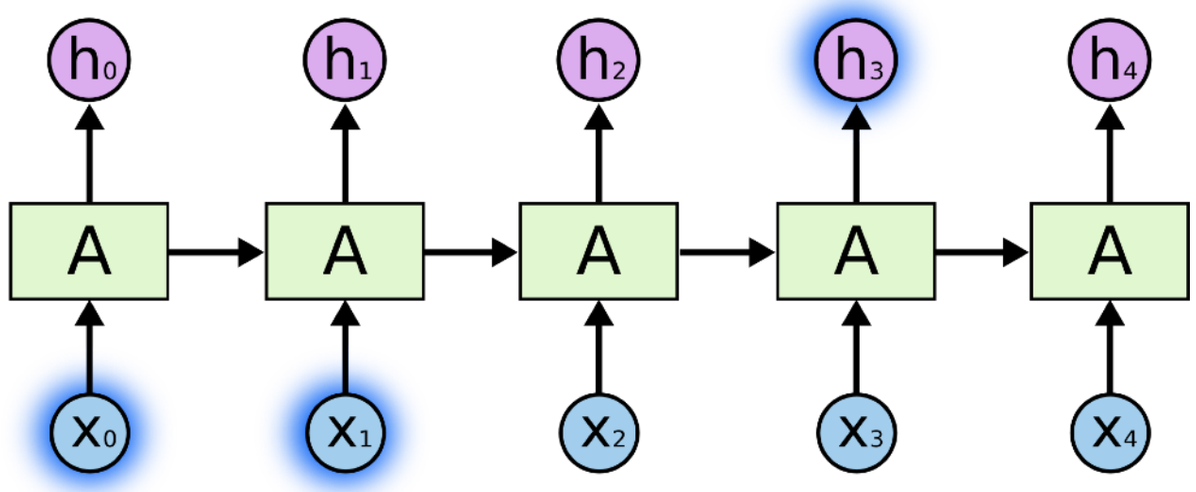

**the clouds are in the sky**



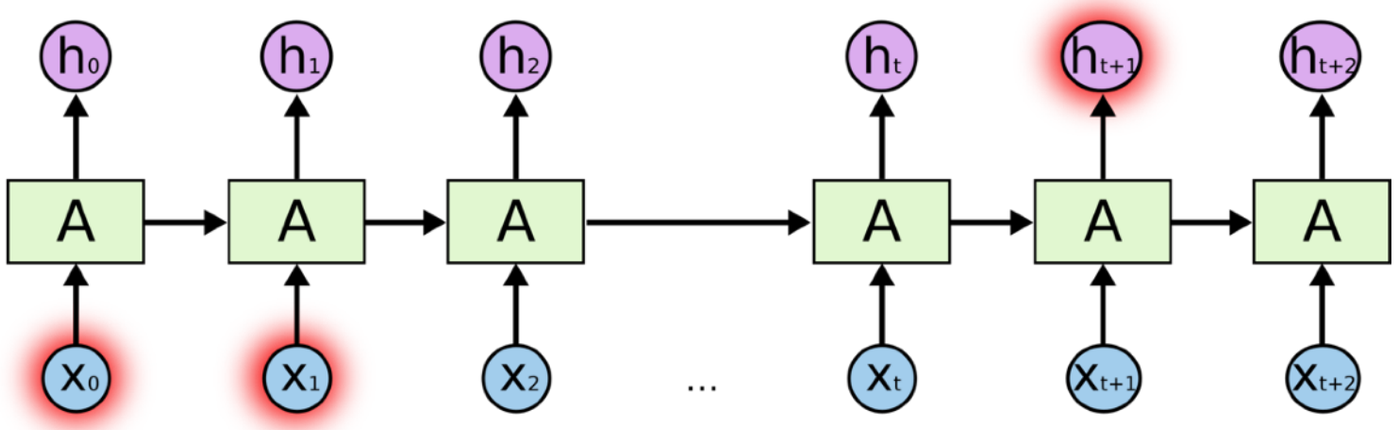

*Slide credit: Christopher Olah*

# Recurent Neural Networks

## The Problem of Long-Term Dependencies



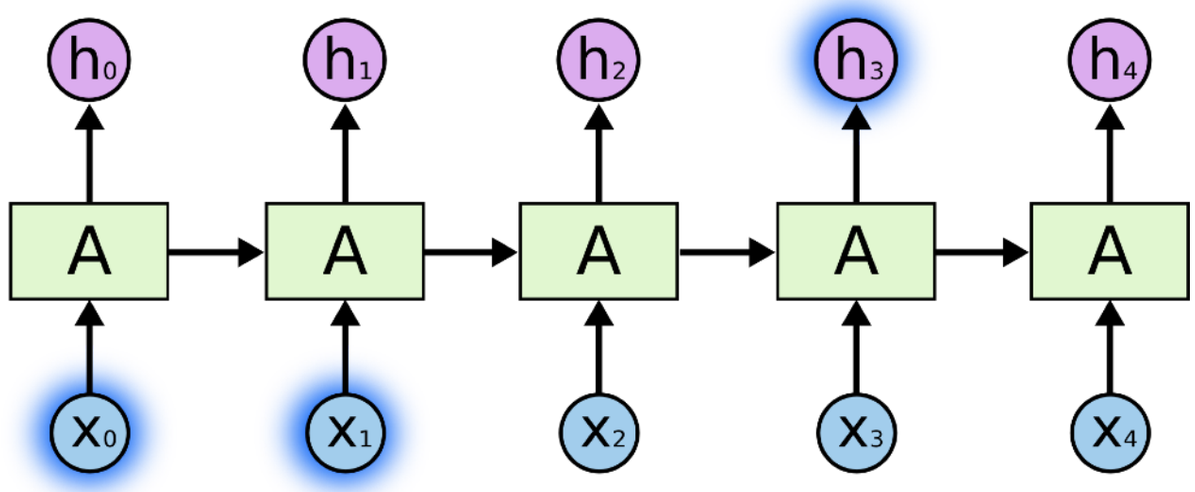

the clouds are in the sky



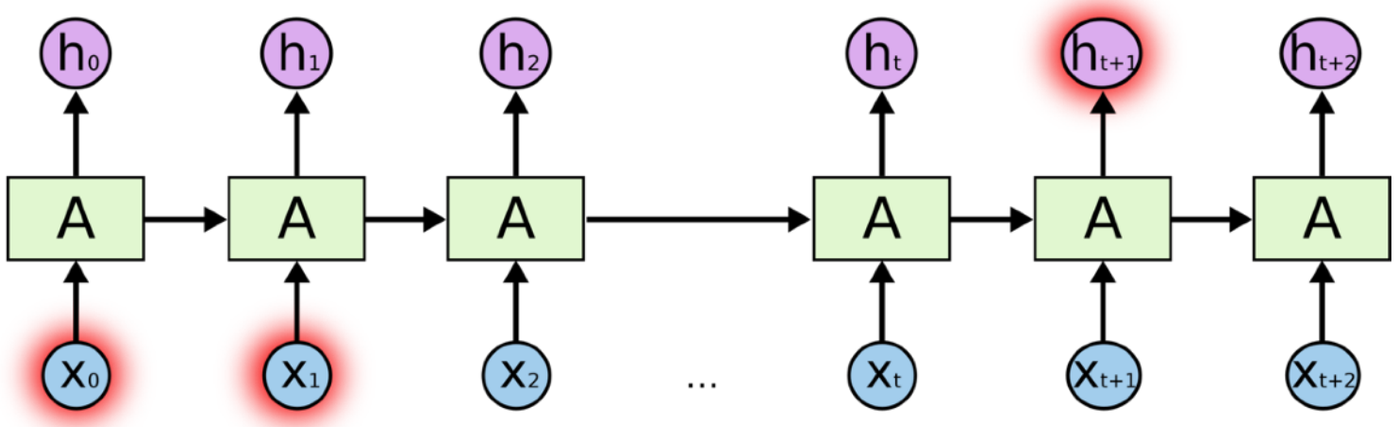

*Slide credit: Christopher Olah*

# Recurent Neural Networks

## The Problem of Long-Term Dependencies

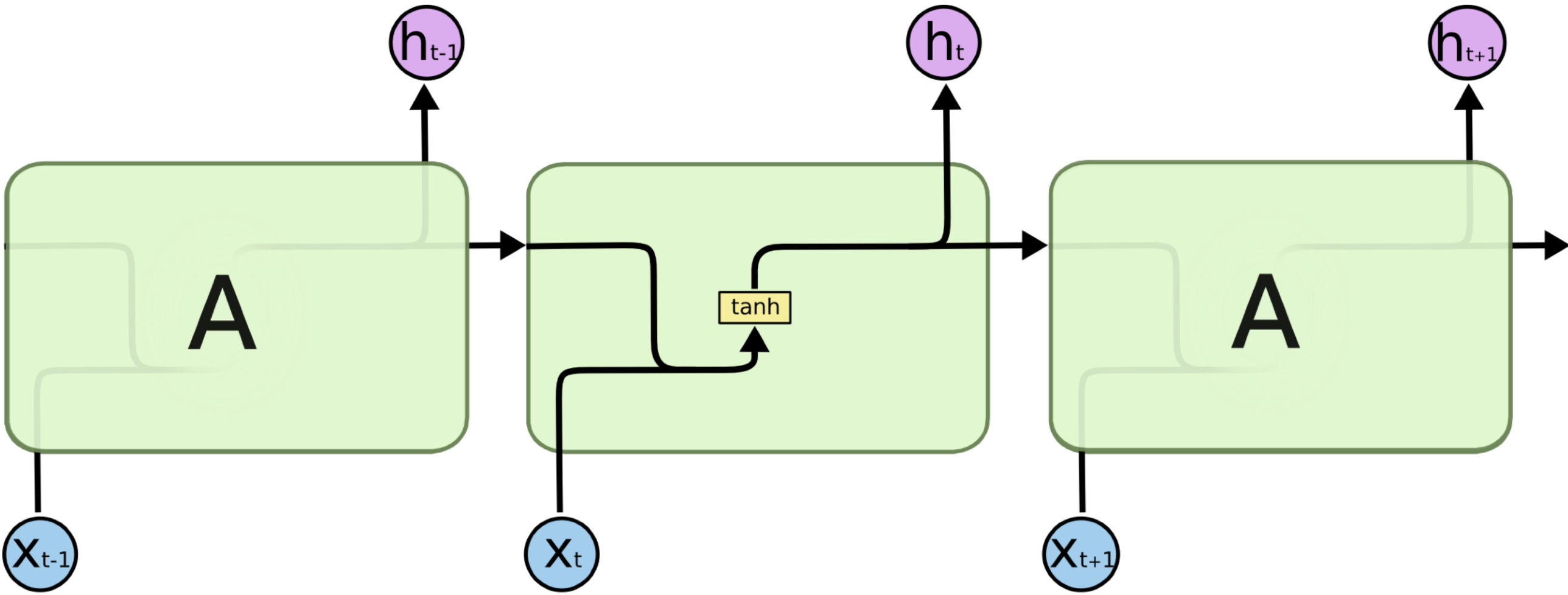



I grew up in France, so since I learn as a kid, I speak fluent French.




*Slide credit: Christopher Olah*

# Recurent Neural Networks



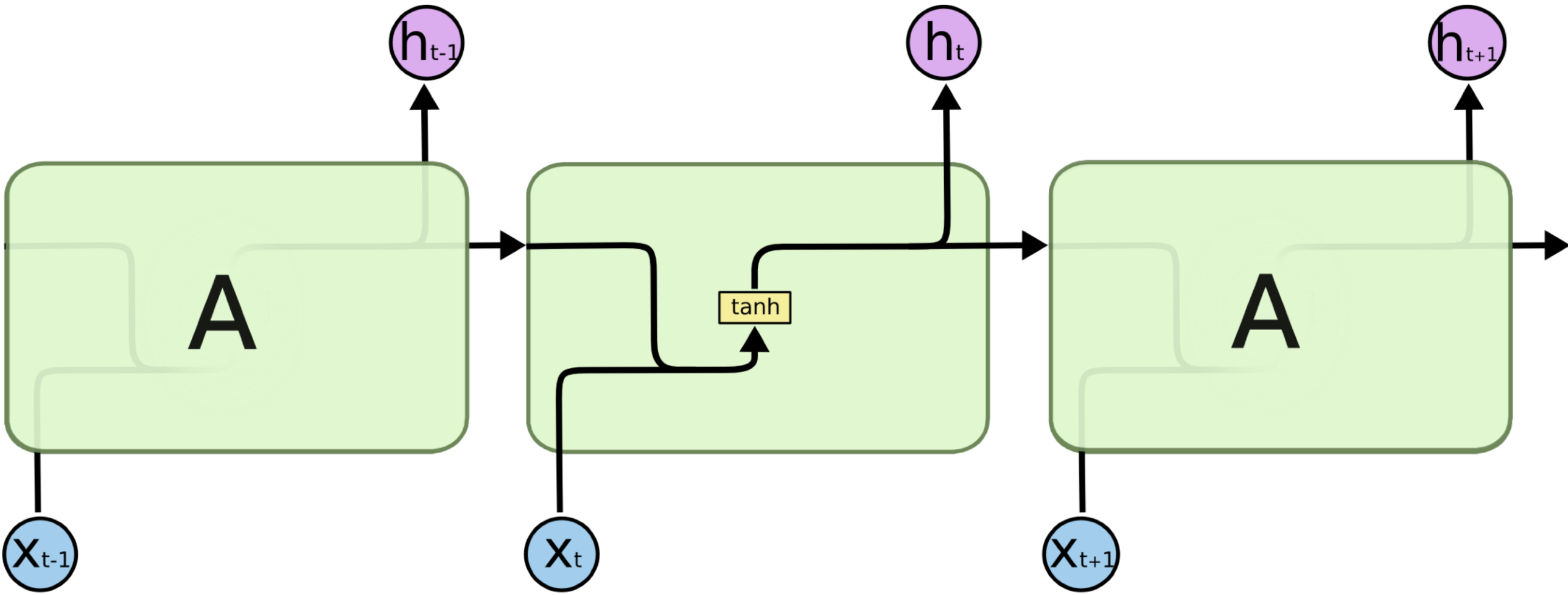

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

*Slide credit: Christopher Olah*

# LSTMs!

## **Long Short Term Memory** networks

**“learn what to remember and to forget”**

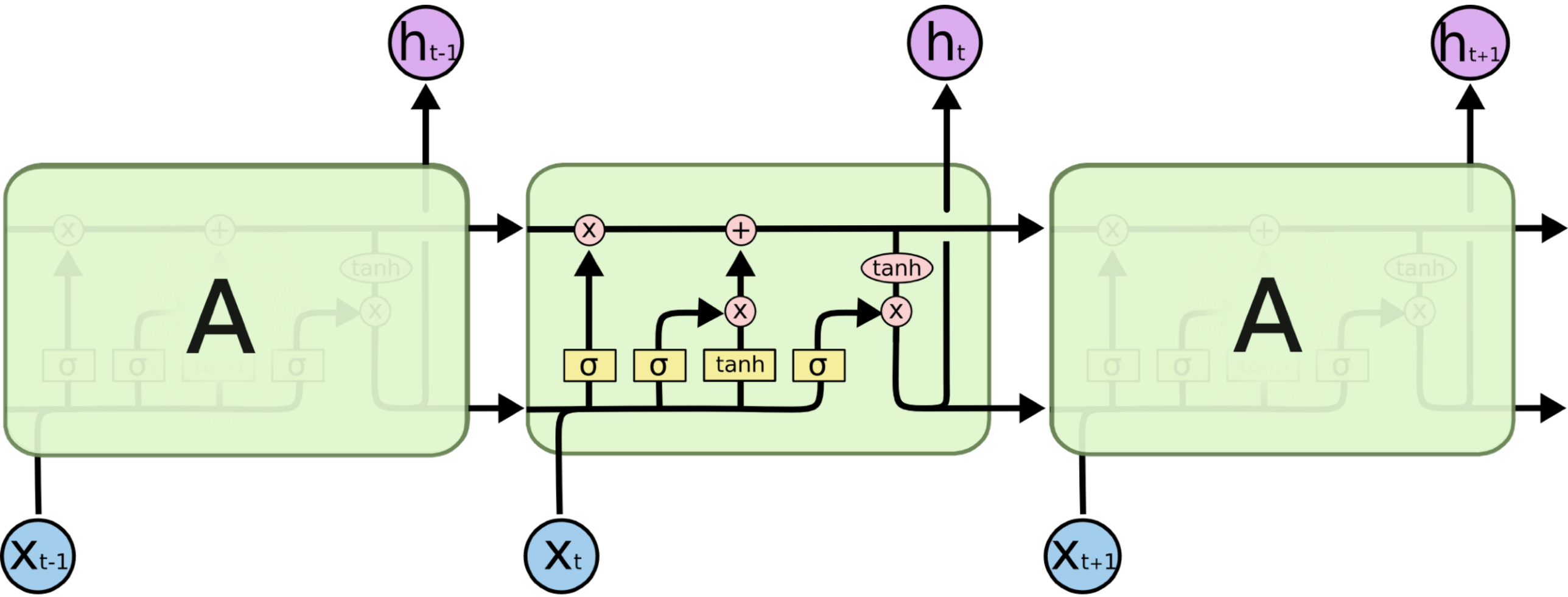



*Slide credit: Christopher Olah*

# LSTMs!

## **Long Short Term Memory** networks

**“learn what to remember and to forget”**

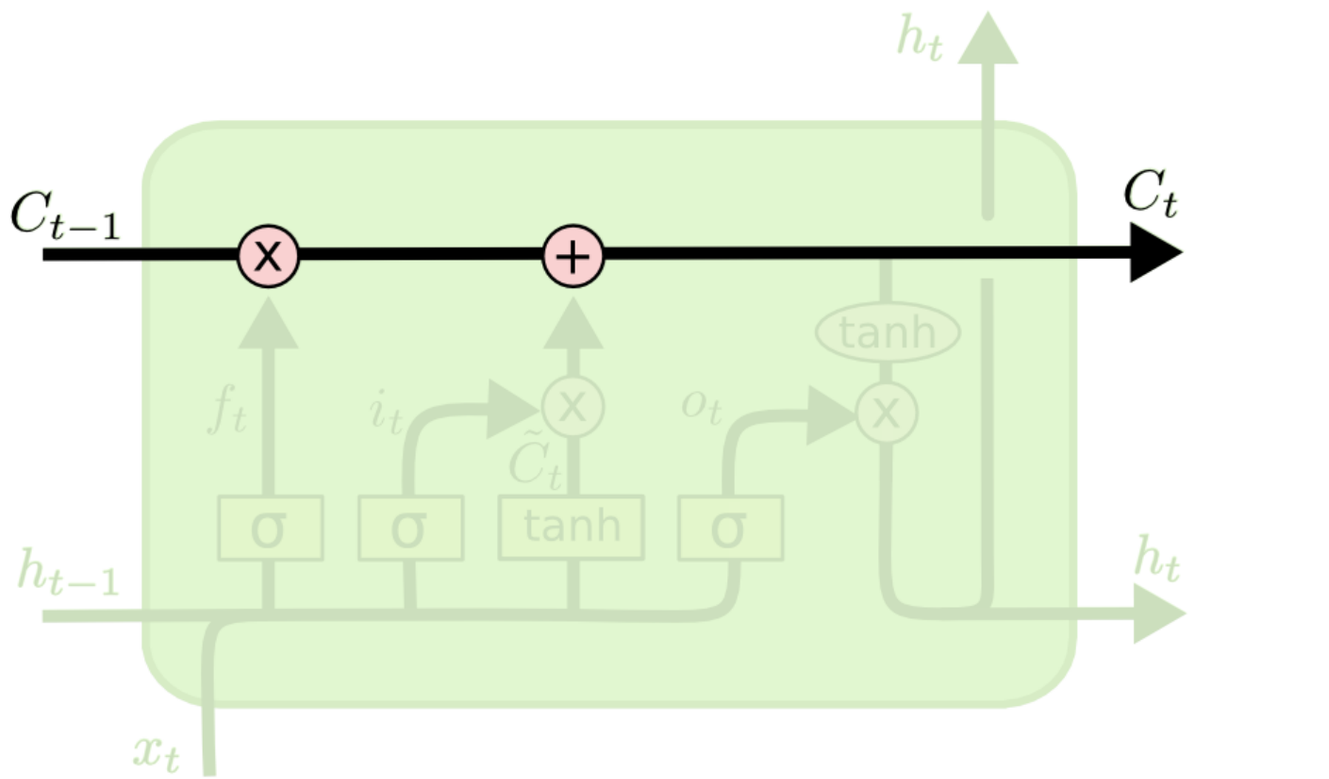


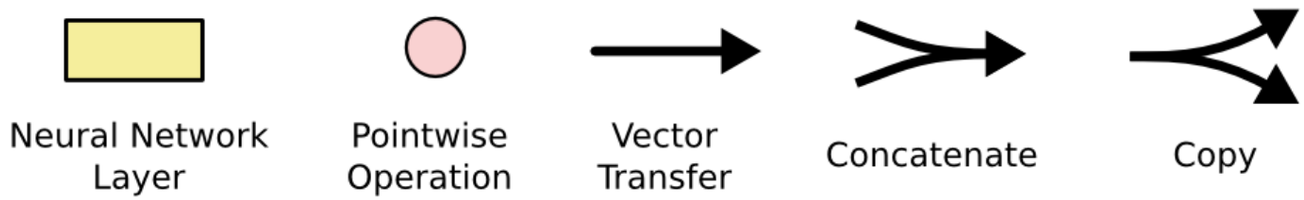

*Slide credit: Christopher Olah*

# LSTMs!

## **Long Short Term Memory** networks

### The CELL GATE! The Core Idea Behind LSTMs



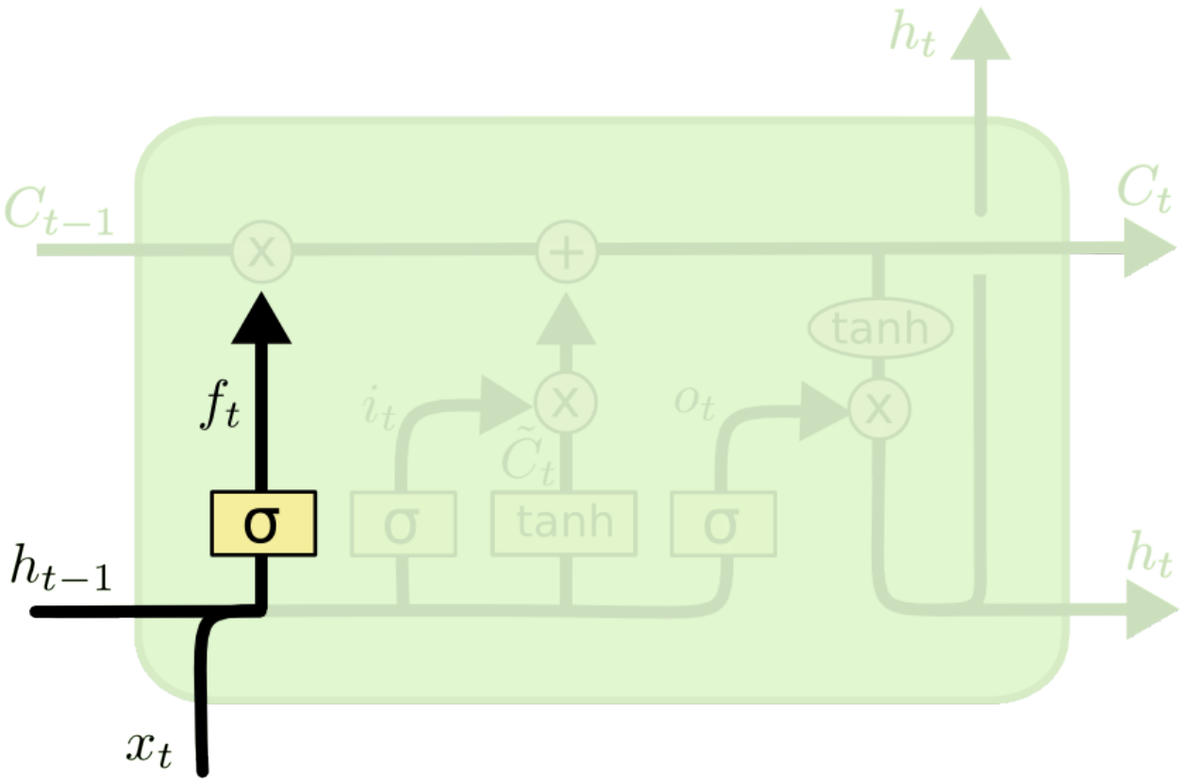




*Slide credit: Christopher Olah*

# LSTMs!

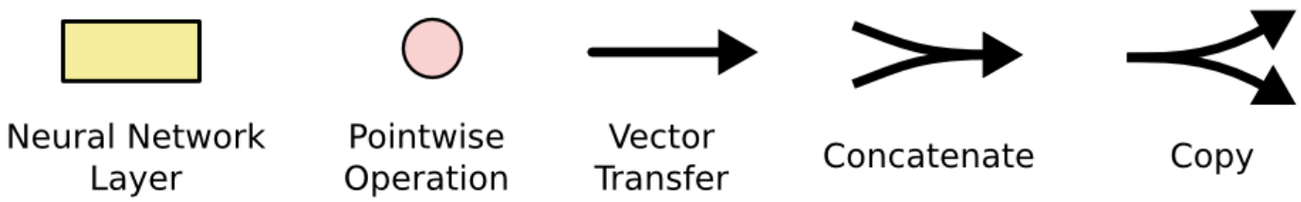
## **Long Short Term Memory** networks

First step: decide what information we're going to throw away from the cell state.




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

**Sigmoid!**

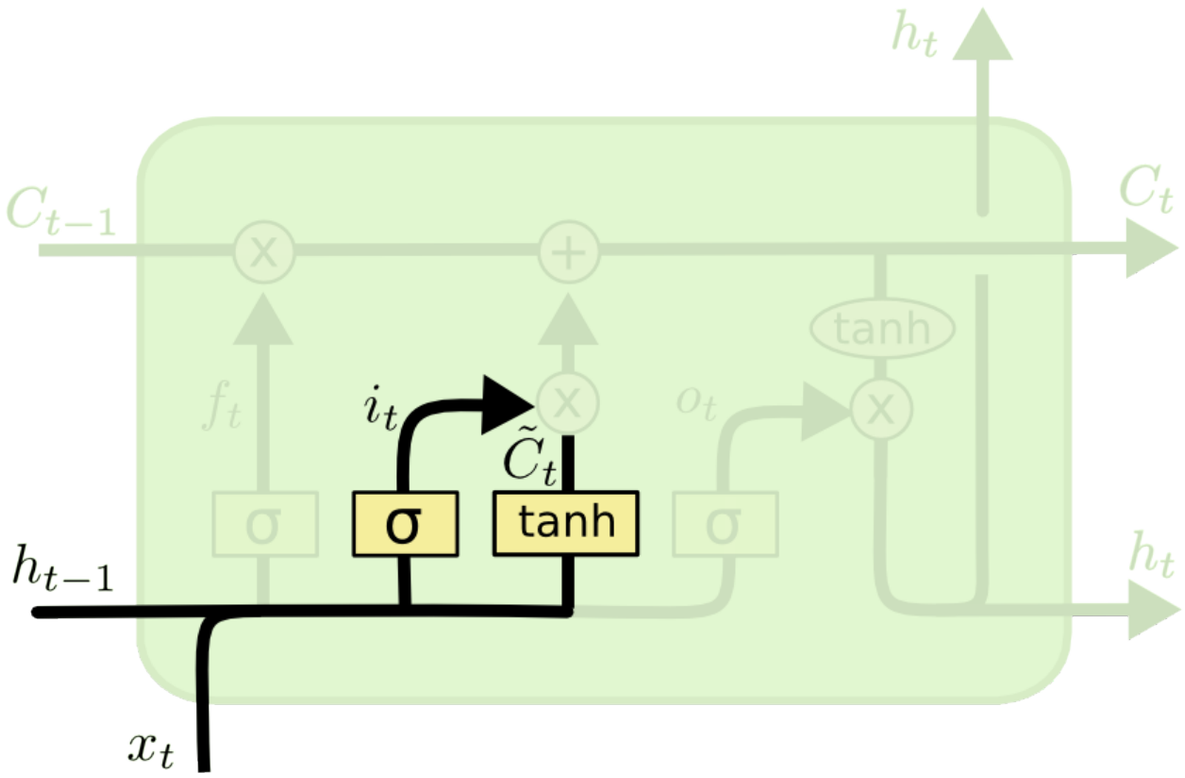



*Slide credit: Christopher Olah*

# LSTMs!

## **Long Short Term Memory** networks

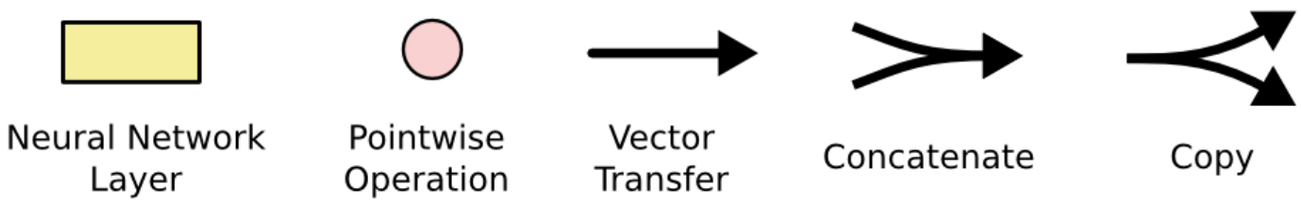
Second step: decide what information we're going to store in the cell state.




**Sigmoid!**

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C$$

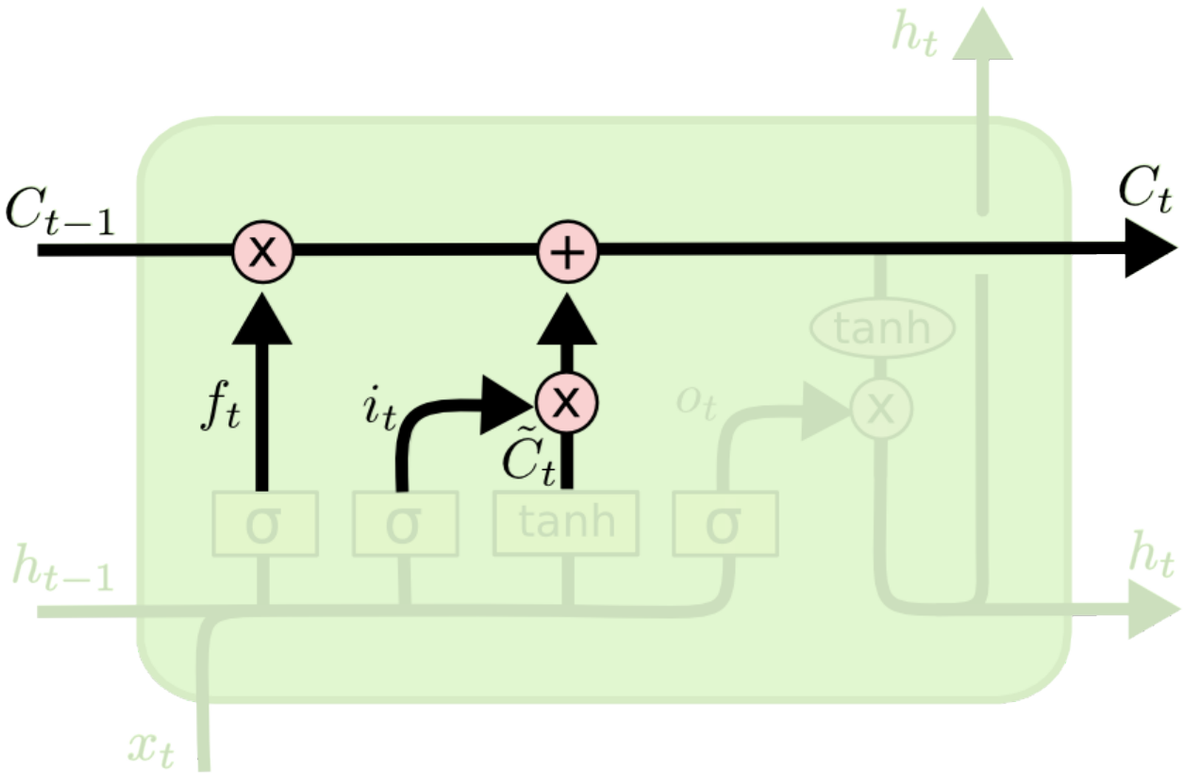



*Slide credit: Christopher Olah*

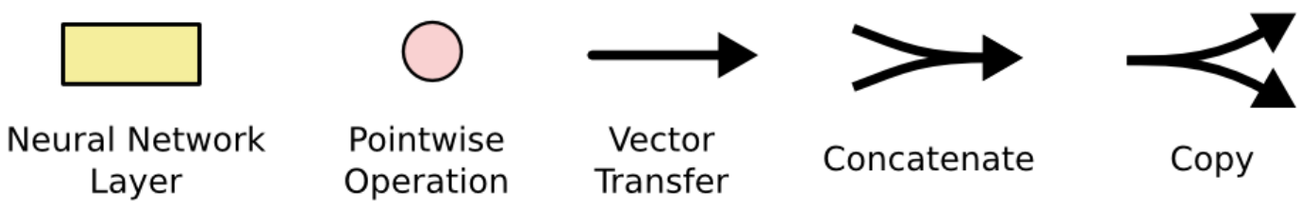
# LSTMs!

## **Long Short Term Memory** networks

Merge the last two steps in the cell gate




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

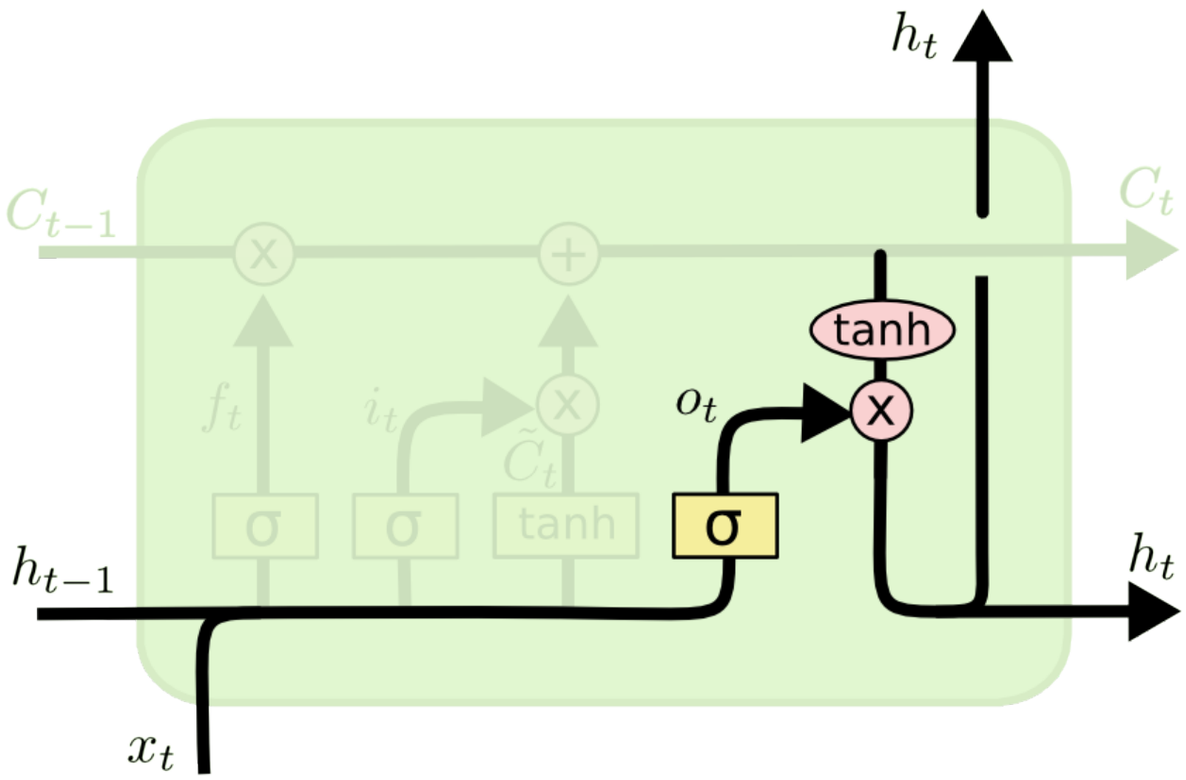



*Slide credit: Christopher Olah*

# LSTMs!

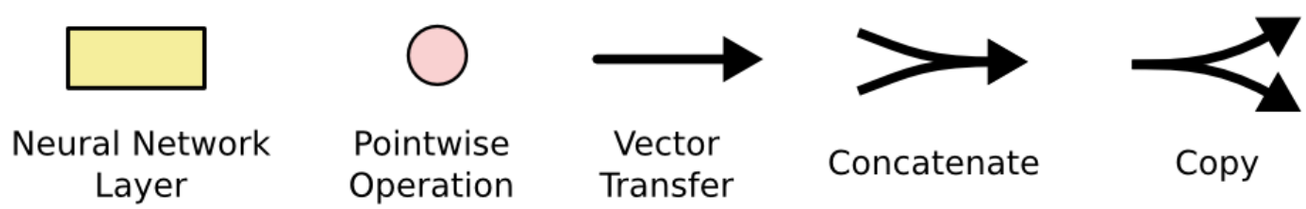
## **Long Short Term Memory** networks

Finally, a more standard output (weighted by the cell gate!)




$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$




*Slide credit: Christopher Olah*

# LSTMs!

## **Long Short Term Memory** networks

**RNN Bible**
@RNN_Bible

Suivre

3:11 The LORD is my strength and glory,
strifes of corrupt things: and thou wilt be thy
supplications.

# LSTMs!

## **Long Short Term Memory** networks

**SEED: Jobs**

*Good afternoon. God bless you.*

*The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done. The promise of the men and women who were still going to take out the fact that the American people have fought to make sure that they have to be able to protect our part. It was a chance to stand together to completely look for the commitment to borrow from the American people. And the fact is the men and women in uniform and the millions of our country with the law system that we should be a strong stretcks of the forces that we can afford to increase our spirit of the American people and the leadership of our country who are on the Internet of American lives.*

*Thank you very much. God bless you, and God bless the United States of America.*

# Adding numbers

## Learning to add number with a recurent neural network

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) that are capable of learning the relationships between elements in an input sequence. A good demonstration of LSTMs is to learn how to combine multiple terms together using a mathematical operation such as a sum and outputting the result of the calculation. This is called "sequence to sequence learning for performing addition" as in http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf and effectivly, we indeed teach the machine to add numbers.

Let us see how this works. First, we create a python class (an object) that allows to link characters (0,1,2,...) and one-hot-encoded categories, as well as to perform the reverse operation. This will save us a lot of time.

---

```
--
Iteration 93
Train on 4500 samples, validate on 500 samples
Epoch 1/1
4500/4500 [==============================] - 1s - loss: 0.0831 - acc: 0.9857 - val_loss: 0.1084 -
val_acc: 0.9713
Q 34+14
T 48
☑ 48
---
Q 61+22
T 83
☑ 83
---
Q 21+97
T 118
☑ 118
---
Q 77+24
T 101
☑ 101
---
Q 1+91
T 92
```

# Wanna know more?

# EE-559 Deep Learning

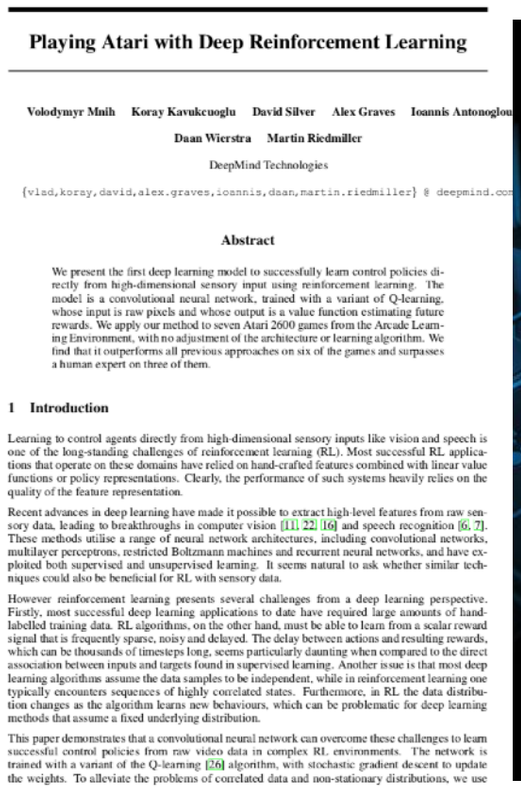
# EE-608 Deep Learning for NLP

# REINFORCEMENT LEARNING

# Take EE-618

# Reinforcment learning

# This tutorial

**Playing Atari with Deep Reinforcement Learning**

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

**Abstract**

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

**1 Introduction**

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand-labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

This paper demonstrates that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the Q-learning [26] algorithm, with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, we use

1


nature
International journal of science

Article | Published: 18 October 2017

## Mastering the game of Go without human knowledge

David Silver ✉, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis

*Nature* **550**, 354–359 (19 October 2017) | Download Citation ⬇

### Abstract

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by

DeepMind

# RL 101

## States & Actions

## Rewards, Policy & Value

# States & Actions

**Curent state of the system**

**Actions of the player**

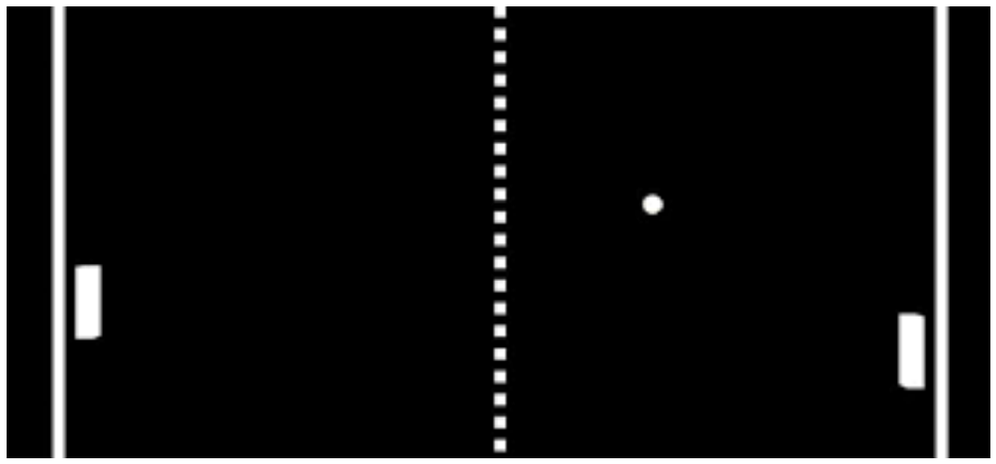
# States & Actions

**Curent state of the system**

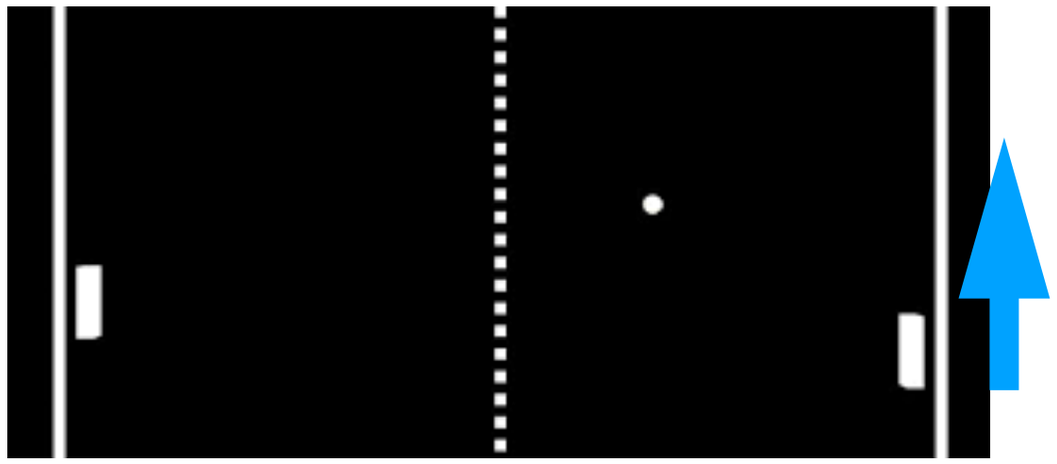
**Actions of the player**

# States & Actions

**Curent state of the system**



**Actions of the player**



# States & Actions

**Curent state of the system**

**Actions of the player**

# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

**the "Value" of the position includes future rewards**

$$V(\{a_t, s_t\}) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$$

# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

**the "Value" of the position includes future rewards**

$$V(\{a_t, s_t\}) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$$

**More precisely, it includes future rewards under perfect play**

$$V(\{a_t, s_t\}) = r(t) + \sum_{\tau=t+1}^{\infty} \gamma^{\tau} r^{\mathrm{BEST}}(\tau)$$

# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

**the "Value" of the position includes future rewards**

$$V(\{a_t, s_t\}) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$$

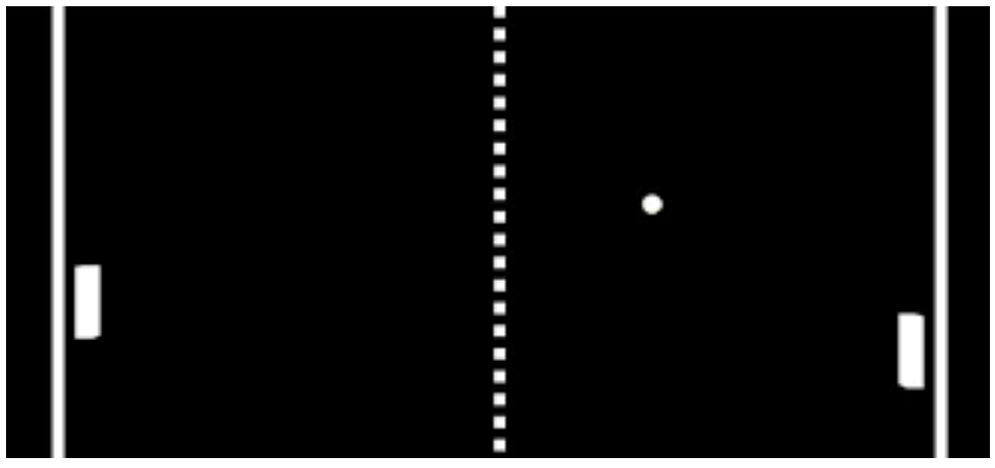
# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

**the "Value" of the position includes future rewards**

$$V(\{a_t, s_t\}) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$$



**Score = 1 if ball still in the game**

# Rewards & value

**Immediate reward** **total Value (including the future)**

**Reward $r^t$ = score at time t as a result of the action**

**the "Value" of the position includes future rewards**

$$V(\{a_t, s_t\}) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$$


**Score = ???**

# Policy

**What action are you playing given the state ?**

$$\pi(a \mid s) \equiv \mathbb{P}(a \mid s)$$

**Policy** $\pi(a \mid s) \equiv \mathbb{P}(a \mid s)$

**Reward $r^t$** = score at time t as a result of the action

**Value** $V(\{a_t, s_t\}) = r(t) + \sum_{\tau=t+1}^{\infty} \gamma^{\tau} r^{\mathrm{BEST}}(\tau) = \sum_{\tau=t}^{\infty} \gamma^{\tau} r(\tau)$

# Q-Learning

**Imagine one has access to a "cheat" table Q*(s,a) that gives,**
***<u>for every state of the system s,</u>***
**the <u>values</u> *for each action a under perfect play***

$$Q^*(s,a) = r(a) + \sum_{t=1}^{\infty} \gamma^t r^{\text{best}|\text{a}}(t)$$

| | **Situation 1** | **Situation 2** | **Situation 3** |
|---|---|---|---|
| **Q(1,Up)** | 7.8 | 10.6 | 1.1 |
| **Q(1,Down)** | 7.6 | 1.6 | 1.2 |
| **Q(1,Stay)** | 1.2 | 7.8 | 0.1 |

# Q-Learning

**Imagine one has access to a "cheat" table Q*(s,a) that gives,**
***<u>for every state of the system s,</u>***
**the <u>values</u> *for each action a under perfect play***

$$Q^*(s,a) = r(a) + \sum_{t=1}^{\infty} \gamma^t r^{\text{best}|\text{a}}(t)$$

OPTIMAL POLICY

| | Situation 1 | Situation 2 | Situation 3 |
|---|---|---|---|
| **Q(1,Up)** | 7.8 | 10.6 | 1.1 |
| **Q(1,Down)** | 7.6 | 1.6 | 1.2 |
| **Q(1,Stay)** | 1.2 | 7.8 | 0.1 |

# Q-Learning

**Imagine one has access to a "cheat" table Q*(s,a) that gives,**
***<u>for every state of the system s,</u>***
**the <u>values</u> *for each action a under perfect play***

$$Q^*(s,a) = r(a) + \sum_{t=1}^{\infty} \gamma^t r^{\text{best}|\text{a}}(t)$$

OPTIMAL POLICY

$$\pi^*(a \mid s) = \delta(a, \text{argmax}_{\text{a}}(Q^*(s,a)))$$

VALUE OF S

$$V(s) = \max_a Q^*(s,a)$$

# How to find the Q-table?

**(i) Start with an estimated one, possibility random** $Q^{t=0}(s, a)$

**(ii) Iterate the following equation:**

$$Q^{t+1}(s, a) = r_a(t) + \gamma \max_{a^{t+1}} \left[ Q^t(s^{t+1}, a^{t+1}) \right]$$

**(iii) Claim: the table will eventually converge to Q***

# Proof!

$$Q^{t+1}(s,a) = r_a(t) + \gamma \max_{a^{t+1}} \left[ Q^t(s^{t+1}, a^{t+1}) \right]$$

**(a) Q* is a fixed point!**

$$Q^*(s,a) = r_a + \sum_{t=1}^{\infty} \gamma^t r^{\text{best}|a}(t) = r(a) + \gamma V(s')$$

$$Q^*(s,a) = r_a + \gamma \max_{a'} \left[ Q^*(s' \mid a, a') \right]$$

**State s , action a -> State s’**

# Proof!

$$Q^{t+1}(s,a) = r_a(t) + \gamma \max_{a^{t+1}} \left[ Q^t(s^{t+1}, a^{t+1}) \right]$$

**(b) the iteration is contractive**

$$Q_1^{t+1}(s,a) - Q_2^{t+1}(s,a) = r(s,a) + \gamma \max_{a^{t+1}} \left[ Q_1^t(s^{t+1}, a^{t+1}) \right] - r(s,a) - \gamma \max_{a^{t+1}} \left[ Q_2^t(s^{t+1}, a^{t+1}) \right]$$

$$Q_1^{t+1}(s,a) - Q_2^{t+1}(s,a) = \gamma \left( \max_{a^{t+1}} \left[ Q_1^t(s^{t+1}, a^{t+1}) \right] - \max_{a^{t+1}} \left[ Q_2^t(s^{t+1}, a^{t+1}) \right] \right)$$

$$Q_1^{t+1}(s,a) - Q_2^{t+1}(s,a) \leq \gamma \max_{a',s'} \left( \left[ Q_1^t(s',a') \right] - \left[ Q_2^t(s',a') \right] \right)$$

$$\| Q_1^{t+1}(s,a) - Q_2^{t+1}(s,a) \|_\infty \leq \gamma \| \left[ Q_1^t(s',a') \right] - \left[ Q_2^t(s',a') \right] \|_\infty$$

# Proof!

$$Q^{t+1}(s,a) = r_a(t) + \gamma \max_{a^{t+1}} \left[ Q^t(s^{t+1}, a^{t+1}) \right]$$

**(a) Q* is a fixed point!**

**(b) the iteration is contractive**

**Convergence to Q***

## Banach fixed-point theorem

From Wikipedia, the free encyclopedia

In mathematics, the **Banach–Caccioppoli fixed-point theorem** (also known as the **contraction mapping theorem** or **contraction mapping principle**) is an important tool in the theory of metric spaces; it guarantees the existence and uniqueness of fixed points of certain self-maps of metric spaces, and provides a constructive method to find those fixed points. It can be understood as an abstract formulation of Picard's method of successive approximations.[1] The theorem is named after Stefan Banach (1892–1945) and Renato Caccioppoli (1904–1959), and was first stated by Banach in 1922. Caccioppoli independently proved the theorem in 1931.[2]

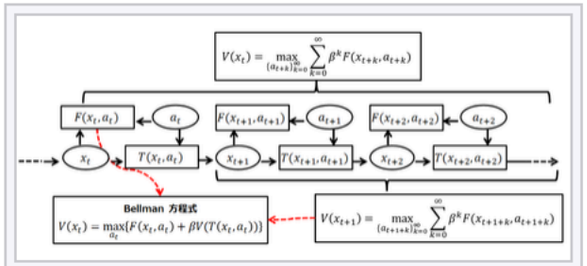
# Bellman equation (’57)

# Bellman equation

From Wikipedia, the free encyclopedia

This article includes a list of general references, but it remains largely unverified because **it lacks sufficient corresponding inline citations**. Please help to improve this article by introducing more precise citations. *(April 2018) (Learn how and when to remove this template message)*

A **Bellman equation**, named after Richard E. Bellman, is a necessary condition for optimality associated with the mathematical optimization method known as dynamic programming.[1] It writes the "value" of a decision problem at a certain point in time in terms of the payoff from some initial choices and the "value" of the remaining decision problem that results from those initial choices.[*citation needed*] This breaks a dynamic optimization problem into a sequence of simpler subproblems, as Bellman's “principle of optimality" prescribes.[2]




Bellman flow chart

The Bellman equation was first applied to engineering control theory and to other topics in applied mathematics, and subsequently became an important tool in economic theory; though the basic concepts of dynamic programming are prefigured in John von Neumann and Oskar Morgenstern's *Theory of Games and Economic Behavior* and Abraham Wald's *sequential analysis*.[*citation needed*] The term 'Bellman equation' usually refers to the dynamic programming equation associated with discrete-time optimization problems.[3] In continuous-time optimization problems, the analogous equation is a partial differential equation that is called the Hamilton–Jacobi–Bellman equation.[4][5]

In discrete time any multi-stage optimization problem can be solved by analyzing the appropriate Bellman equation. The appropriate Bellman equation can be found by introducing new state variables (state augmentation).[6] However, the resulting augmented-state multi-stage optimization problem has a higher dimensional state space than the original multi-stage optimization problem - an issue that can potentially render the augmented problem intractable due to the “curse of dimensionality”. Alternatively, it has been shown that if the cost function of the multi-stage optimization problem satisfies a "backward separable" structure then the appropriate Bellman equation can be found without state augmentation.[7]

# Bellman equation (’57)

$$Q^{t+1}(s,a) = r(t) + \gamma \max_{a^{t+1}} \left[ Q^{t}(s^{t+1}, a^{t+1}) \right]$$

Play according to your current Q-table, try all state, and all actions

Often **useless** in practice: we can never apply the Bellman operator **exactly**, except for very small problems where we can easily iterate over all (s,a)

**Instead try out randomly some actions & iterate a damped version of the algorithm**

$$Q^{t+1}(s,a) = (1-\delta) * Q^{t}(s,a) + \delta\left(R(a) * \gamma \max_{a'} Q^{*}(s' | a, a')\right)$$

**How to try “randomly ?”**

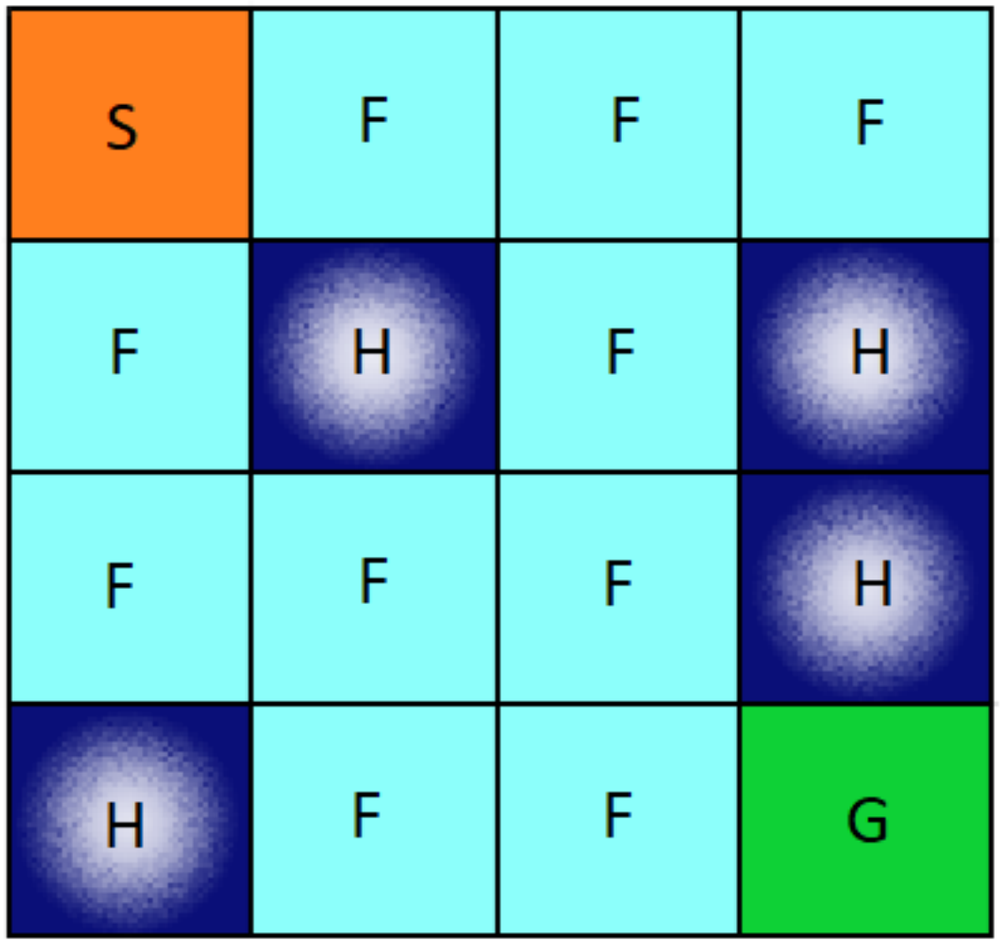
**Follow current policy (exploitation “in-policy”) p=1-ε**

**Try random moves (exploration “out-policy”) p=ε**

# A notebook for frozen lake!

```
qtable = np.random.uniform(0,1e-4,(state_size, action_size))
print(qtable)
```

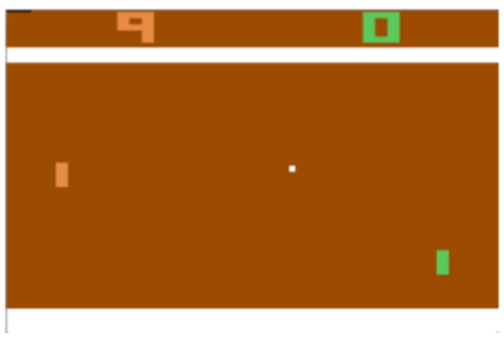
```
[[2.23373536e-05 9.53030158e-05 8.95416017e-05 6.67212136e-05]
 [6.37936388e-05 5.36921572e-05 5.81251633e-05 1.14374951e-05]
 [6.35668159e-05 4.66022420e-05 1.59235050e-05 7.21301648e-05]
 [3.91223360e-05 2.69637943e-05 5.47249711e-06 6.32151651e-05]
 [7.11248386e-05 4.48782366e-05 9.37446329e-05 6.05084050e-05]
 [6.80195147e-05 6.98881962e-06 8.28213306e-05 7.79682486e-06]
 [7.96609453e-05 1.50792366e-05 2.90486844e-05 3.53425389e-05]
 [2.53335290e-05 8.53179739e-05 8.39578114e-06 4.90978506e-06]
 [2.93531871e-05 3.03221281e-05 3.66774591e-05 6.14993506e-05]
 [5.99103996e-05 4.91679620e-05 6.30646577e-05 6.03383140e-05]
 [1.89577306e-05 7.13617019e-05 5.22561255e-05 4.81692791e-05]
 [1.65522189e-05 2.77649000e-06 9.70695996e-05 7.15228114e-05]
 [7.50908402e-05 7.08843014e-05 9.41159714e-05 2.91452041e-06]
 [2.51453352e-05 3.12565910e-05 3.99746151e-05 5.72117333e-05]
 [2.51211832e-05 9.69122843e-06 5.74488511e-05 2.32887949e-05]
 [1.39831530e-06 3.16944752e-05 4.25885418e-05 6.78702678e-05]]
```

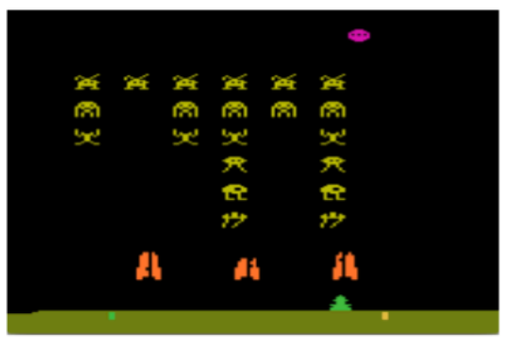



```
#Now, we play until dead or until it became tooooooooo long
for step in range(max_steps):
    # First we decide if we play in or out of policy:
    exp_exp_tradeoff = random.uniform(0, 1)
    # If this number > greater than epsilon --> exploitation (taking the biggest Q value for this state)
    if exp_exp_tradeoff > epsilon:
        action = np.argmax(qtable[state,:])
    # Else doing a random choice --> exploration
    else:
        action = env.action_space.sample()

    # Now we take the action (a) and observe the outcome state(s') and reward (r)
    new_state, reward, done, info = env.step(action)

    # Finally we perform the Bellman update...
    qtable[state, action] = qtable[state, action] + learning_rate * (reward + gamma * np.max(qtable[new_state, :]) - qta
    #... and update the reward for this game.
    # Note that here, we only get a reward 1 if we eventually reach the goal!
    total_rewards += reward

    # Our new state is state
    state = new_state

    # If done (if we're dead) : finish episode
    if done == True:
        break
    # otherwise we continue to play
```

# Atari games

## Reinforcement leanring

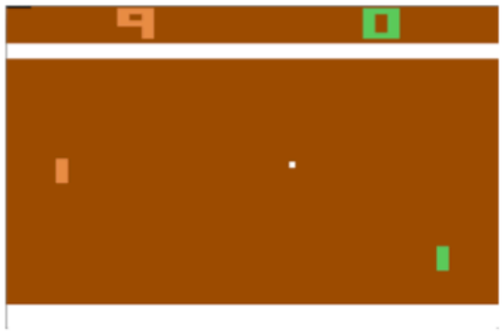


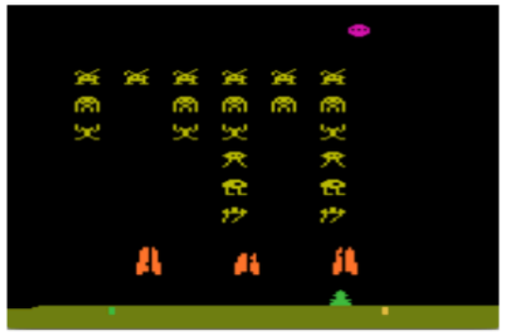





# Atari games

## Reinforcement leanring







**Ask a computer to learn how to play breakout such that the <u>score is good at the end of the game</u>**

**Rules: The computer “see” the image & control the joystick.**
**Asked to maximise the end score**

# Reinforcement learning

**Action $a^t$ = action of the joystick**

**Reward $r^t$ = score at time t as a result of the action**

**Q-learning ??**

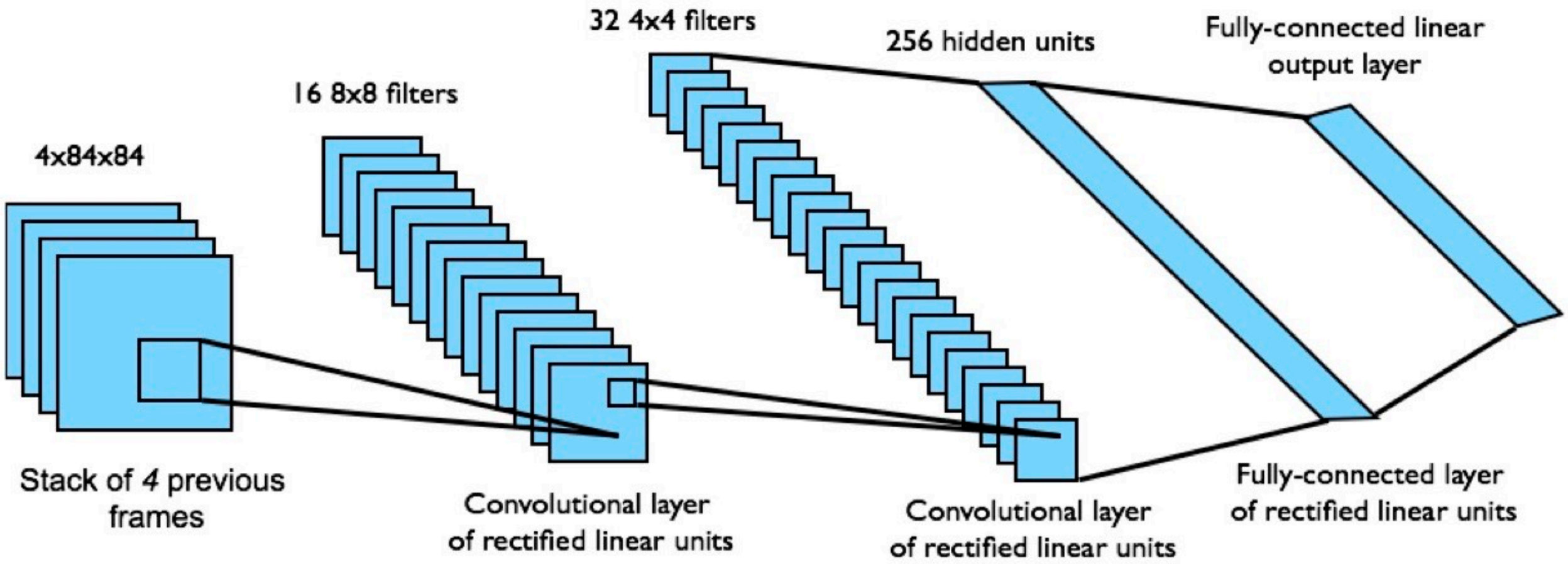
$$Q^{t+1}(s,a) = r(t) + \gamma \max_{a^{t+1}} \left[ Q^t(s^{t+1}, a^{t+1}) \right]$$

**PROBLEM: THE TABLE IS TOO BIG!!!**

**TOO MANY POSSIBLE “STATES” FOR THESE GAMES**

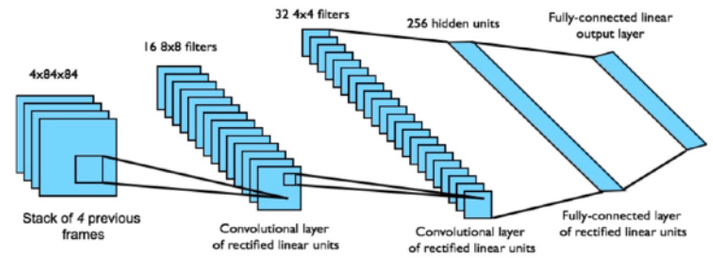
# Represent Q(s,a) as a conv-net

*Approximate the table by a neural network*




# Deep Q-learning

## (1) Represent the Q-table as a convent




## (2) Define the cost according to Bellman fix-point

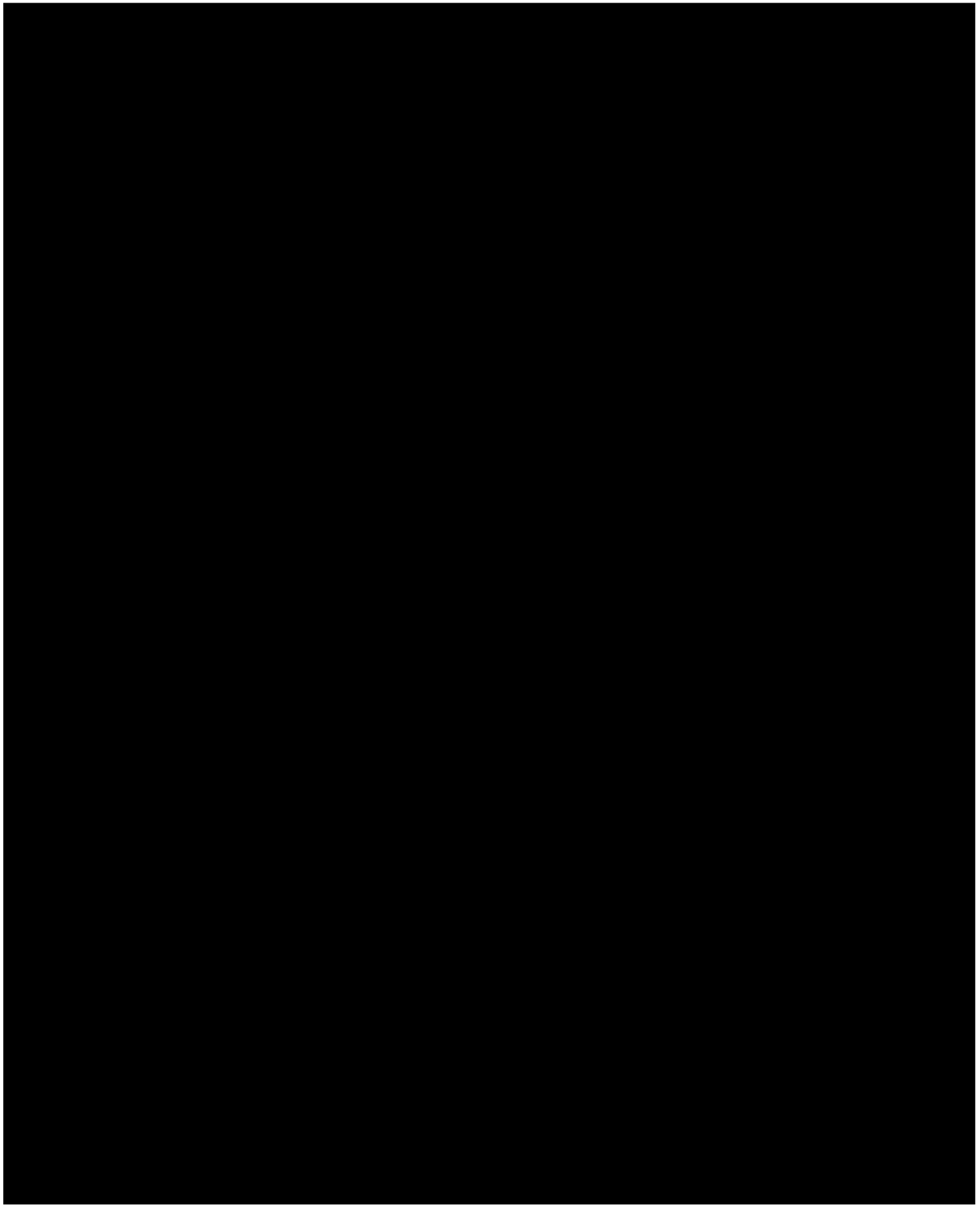
$$\text{Cost} = \sum_{s} \left( Q(s,a) - r(t) - \gamma \max_{a'} \left[ Q(s^{t+1}, a^{t+1}) \right] \right)^2$$

**Learn the function Q(s,a) by gradient descent**

## (3) Play according to Q, but explore other possibilities

**Follow current policy (exploitation “in-policy”)** **With P=1-ε**

**Try random moves (exploration “out-policy”)** **With P=ε**




Before training
peaceful swimming

# Policy Gradients

**Optimize directly the policy π(a|s) *<u>without</u>* using the Q-table**
**Advantage : can be use for continuous variable, direct optimization**

$$J(\theta) = \mathbb{E}_{\pi_\theta, \vec{s}}\left[V(\vec{s})\right]$$

**J is the expected value averaged over the policy**

**Compute the gradient of J !**

# REINFORCE

$$J(\theta) = \mathbb{E}[f(X)] = \int_x f(x) p_{\boldsymbol{\theta}}(x)\, dx$$

$$\nabla_\theta J(\theta) = \int_x f(x) \nabla_\theta p_{\boldsymbol{\theta}}(x)\, dx$$

**Now, assume that we are dealing with trajectories generated by a Markov chain:**

$$p_{\boldsymbol{\theta}}(\tau) = p(s_0) \prod_{i=0}^{\infty} \pi_{\boldsymbol{\theta}}(a_i|s_i) p(s_{i+1}|s_i)$$

$$\log p_{\boldsymbol{\theta}}(\tau) = \log p(s_0) + \sum_i \left(\log \pi_{\boldsymbol{\theta}}(a_i|s_i) + \log p(s_{i+1}|s_i)\right)$$

**Finally, gradients are given by:**

$$g = \nabla_\theta J(\theta) = \mathbb{E}\left[ f(\tau) \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \right]$$

# REINFORCE

$$g = \nabla_\theta J(\theta) = \mathbb{E}\left[ f(\tau) \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \right]$$

**Approximate the Expectation by an empirical sum over many “plays” with the current policy**

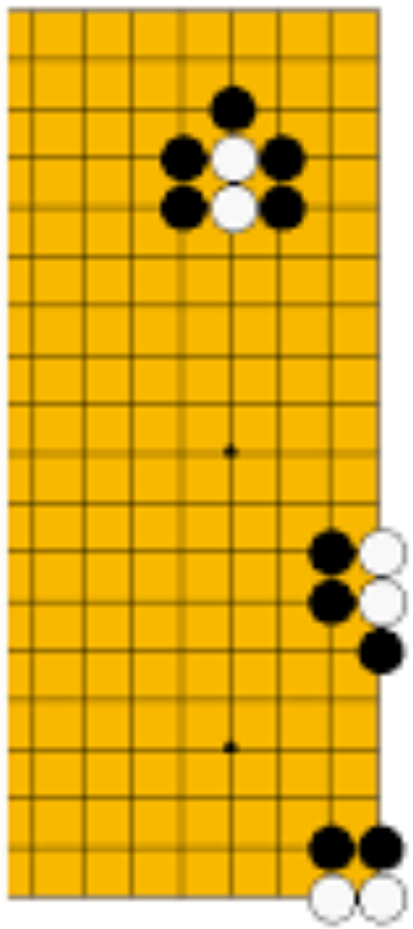
$$g = \nabla_\theta J(\theta) \approx \frac{1}{P} \sum_{p=1}^{P} f(\tau) \sum_i \nabla_\theta \log \pi_\theta(a_i \,|\, s_i)$$

# Go

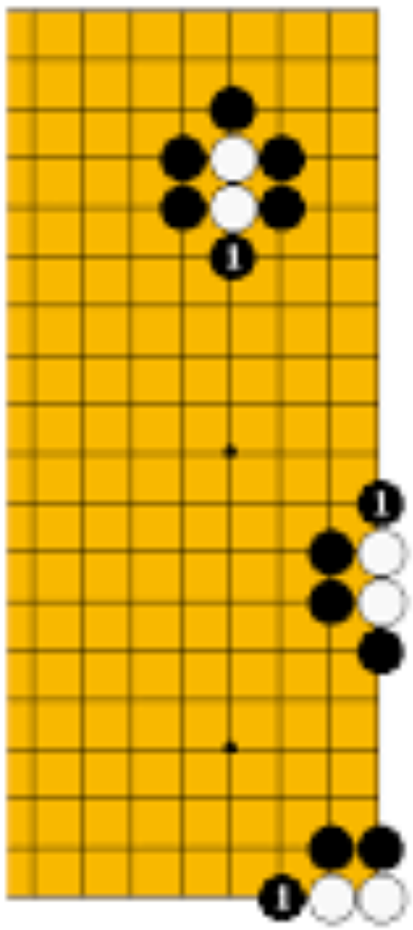
# 2,500 years old: Oldest game still played today

One type of piece, one type of move

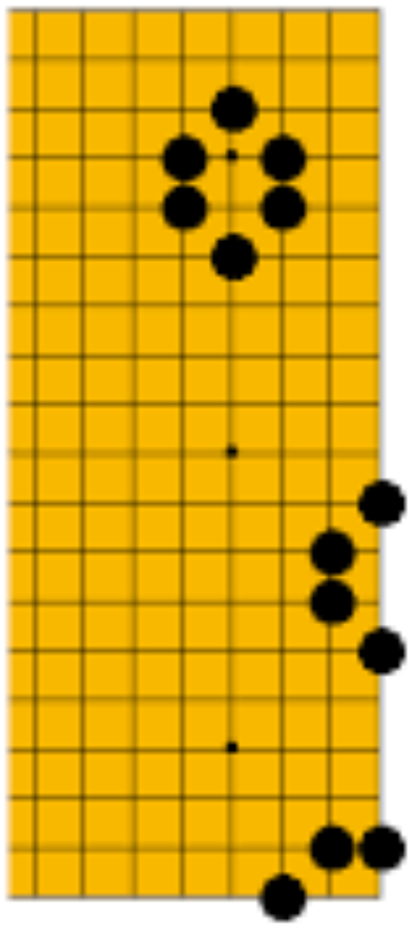
Once placed on the board, stones may not be moved, but stones are removed from the board if "captured". Capture happens when a stone or group of stones is surrounded by opposing stones on all orthogonally-adjacent points



Dia. 18

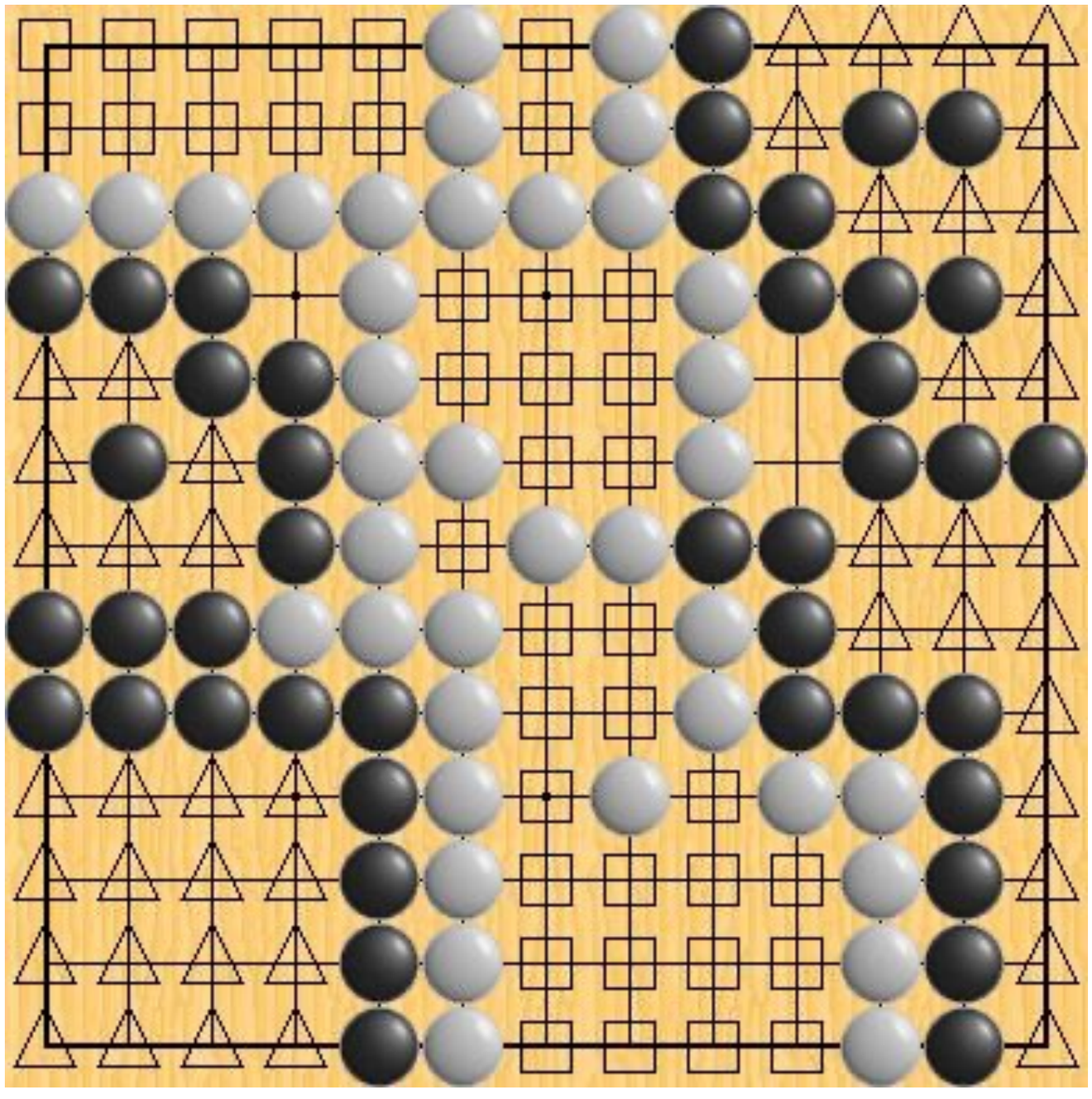


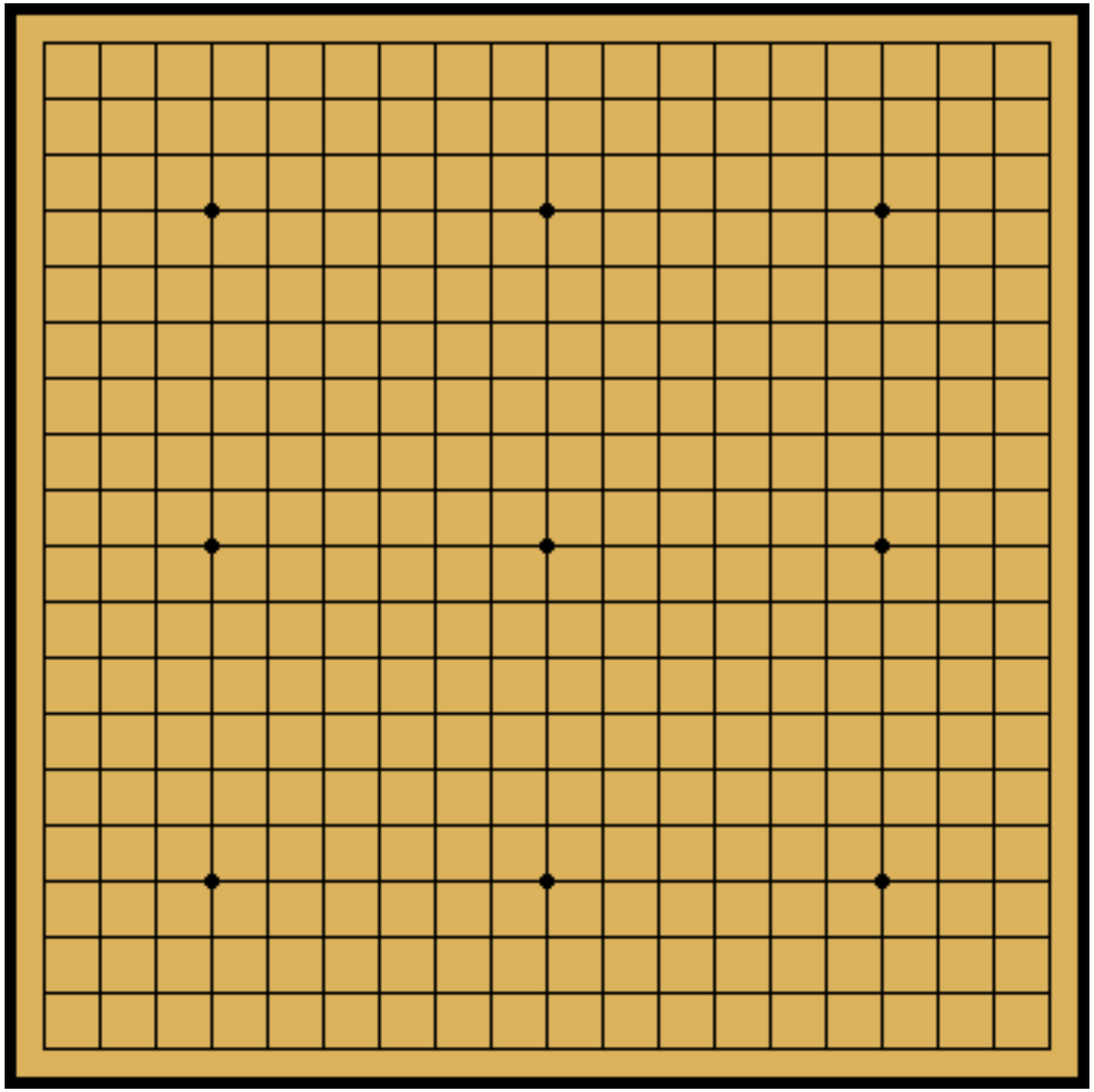

Dia. 19

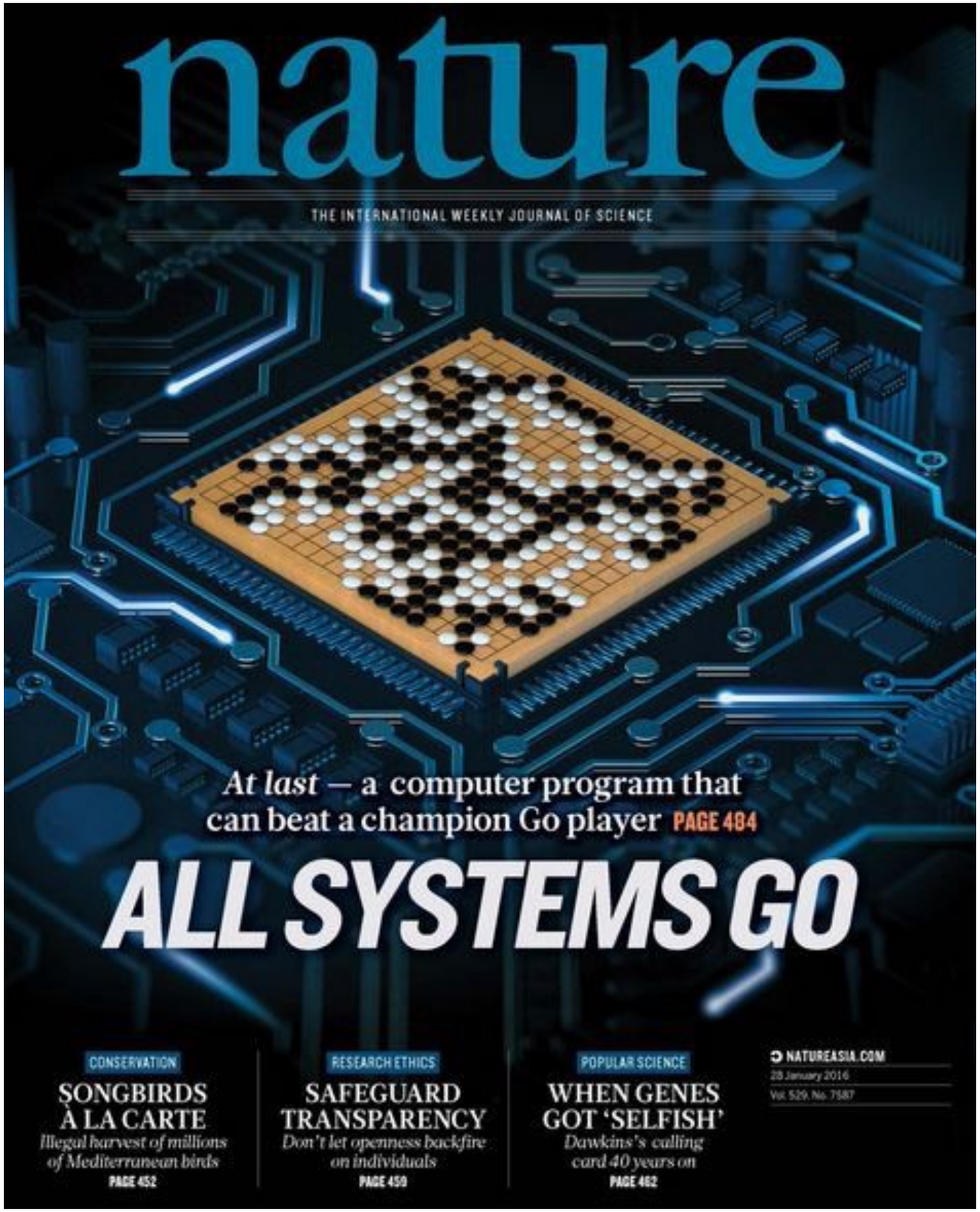


Dia. 20

A player's score is determined by the number of stones that player has on the board plus the empty area surrounded by that player's stones.






nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last — a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO
CONSERVATION
SONGBIRDS À LA CARTE
Illegal harvest of millions of Mediterranean birds
PAGE 452
RESEARCH ETHICS
SAFEGUARD TRANSPARENCY
Don't let openness backfire on individuals
PAGE 459
POPULAR SCIENCE
WHEN GENES GOT 'SELFISH'
Dawkins's calling card 40 years on
PAGE 462
NATUREASIA.COM
28 January 2016
Vol. 529, No. 7587

# STEP 1

## Supervised learning of policy networks

We trained a 13-layer policy network, which we call the SL policy network, from 30 million positions from the KGS Go Server. The net- work predicted expert moves on a held out test set with an accuracy of 57.0% using all input features, and 55.7% using only raw board position and move history as inputs,

The SL policy network $p_\sigma(a \mid s)$ alternates between convolutional layers with weights $\sigma$, and rectifier nonlinearities. A final softmax layer outputs a probability distribution over all legal moves $a$.

The policy network is trained on randomly sampled state-action pairs ($s$, $a$), using stochastic gradient ascent to maximize the likelihood of the human move $a$ selected in state $s$

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}$$

# STEP 2

## Reinforcement learning of policy networks

Play many games

Compute policy gradient, use REINFORCE

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

The outcome $z_t = \pm r(s_T)$ is the terminal reward at the end of the game from the perspective of the current player at time step $t$: +1 for winning and −1 for losing.

We evaluated the final RL algorithm by sampling each move $a_t \sim p_\varrho(\cdot|s_t)$ from its output probability distribution over actions. When played head-to-head, the RL policy network won more than 80% of games against the SL policy network. We also tested against the strongest open-source Go program, Pachi, a sophisticated Monte Carlo search program, ranked at 2 amateur *dan* on KGS, that executes 100,000 simulations per move. Using no search at all, the RL policy network won 85% of games against Pachi.

# STEP 3

## Reinforcement learning of value networks

$$v^p(s) = \mathbb{E}[z_t | s_t = s,\ a_{t \ldots T} \sim p]$$

Complicated function, approximated by a neural net (again same architecture)

Play many games with the policy RL, train the weights of the value network by regression on state-outcome pairs $(s, z)$, using stochastic gradient descent to minimize the mean squared error (MSE) between the predicted value $v_\theta(s)$, and the corresponding outcome $z$

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$$

# At this pont we have

**Policy (supervised)** $p_\sigma(a \mid s)$

**Policy (RL)** $p_\rho(a \mid s)$

**value networks** $v^p(s) = \mathbb{E}[z_t | s_t = s,\ a_{t \ldots T} \sim p]$

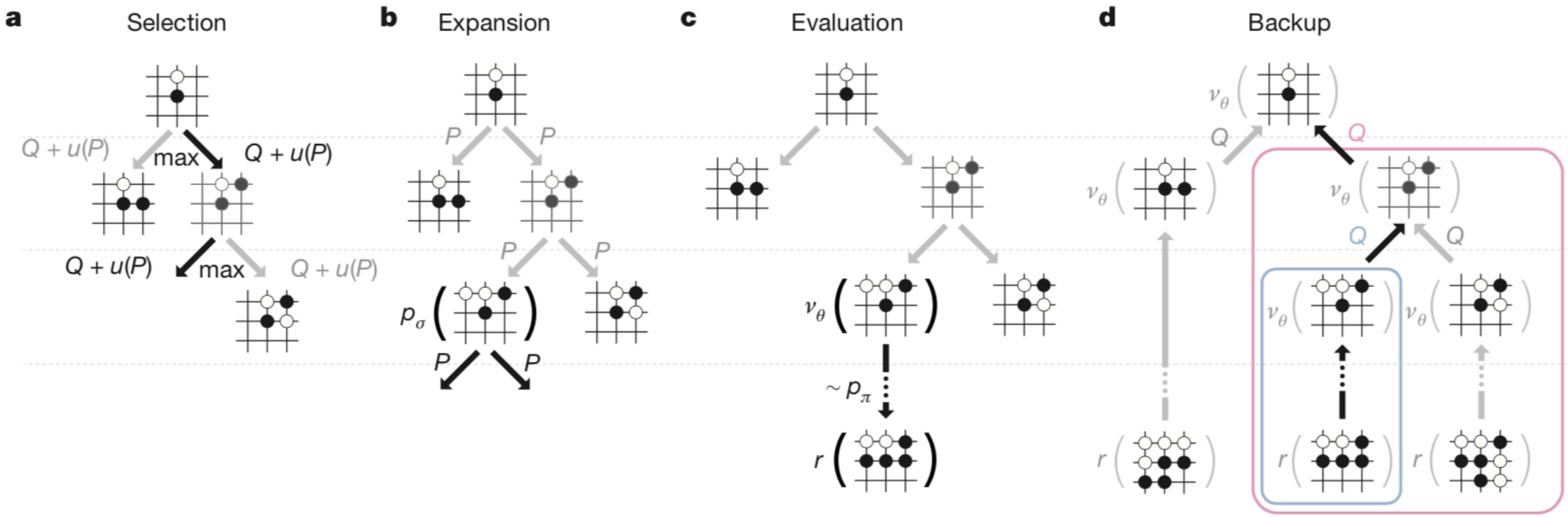
# How to use this? Monte-Carlo Tree-Search

**Guiding the search in the tree of possibilities!**

**Searching with policy and value networks**

# How to use this?

**Monte-Carlo Tree-Search**




At each time step $t$ of each simulation, an action $a_t$ is selected from state $s_t$ :

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i) \qquad Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s)$$

$$V(s) = (1-\lambda)\nu_\theta(s) + \lambda z_{\text{fast}}$$

$$a_t = \text{argmax}_a \left[ Q(s_t,a) + \frac{p_\sigma(s,a)}{1+N(s,a)} \right]$$

# Fan Hui

October 2015

# Lee Sedol

March 2016

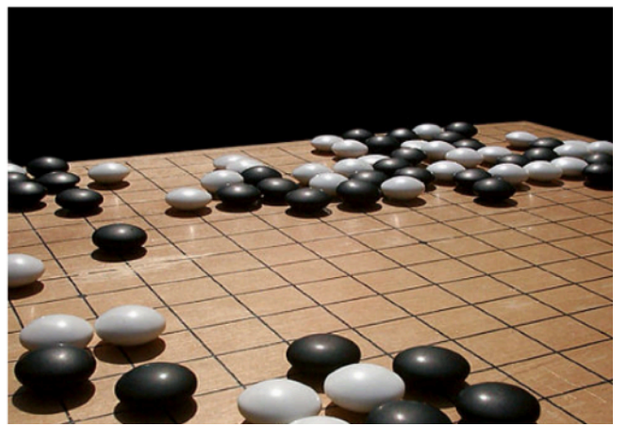
# Move 37

## See the exact moment the world champion of Go realises DeepMind is vastly superior [GIF]

**Jim Edwards**
Mar. 12, 2016, 2:30 PM 9,626



Go fans have noticed a couple of key moments in the match between world champion Lee Sedol and Google's DeepMind AlphaGo



# HOW GOOGLE'S AI VIEWED THE MOVE NO HUMAN COULD UNDERSTAND


### Game 2 [ edit ]

AlphaGo (black) won the second game. Lee stated afterwards that "AlphaGo played a nearly perfect game",[49] "from very beginning of the game I did not feel like there was a point that I was leading".[50] One of the creators of AlphaGo, Demis Hassabis, said that the system was confident of victory from the midway point of the game, even though the professional commentators could not tell which player was ahead.[50]

Michael Redmond (9p) noted that AlphaGo's 19th stone (move 37) was "creative" and "unique".[30] Lee took an unusually long time to respond to the move.[30] An Younggil (8p) called AlphaGo's move 37 "a rare and intriguing shoulder hit" but said Lee's counter was "exquisite". He stated that control passed between the players several times before the endgame, and especially praised AlphaGo's moves 151, 157, and 159, calling them "brilliant".[51]

# Alphago Zero (2017)


**Artificial intelligence research has made rapid progress in a wide variety of domains from speech recognition and image classification to genomics and drug discovery. In many cases, these are specialist systems that leverage enormous amounts of human expertise and data.**

However, for some problems this human knowledge may be too expensive, too unreliable or simply unavailable. As a result, a long-standing ambition of AI research is to bypass this step, creating algorithms that achieve superhuman performance in the most challenging domains with no human input. In our most recent **paper**, published in the **journal Nature**, we demonstrate a significant step towards this goal.

nature
International journal of science

Article | Published: 18 October 2017

## Mastering the game of Go without human knowledge

David Silver ✉, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis

*Nature* **550**, 354–359 (19 October 2017) | Download Citation ⬇

## Abstract

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by

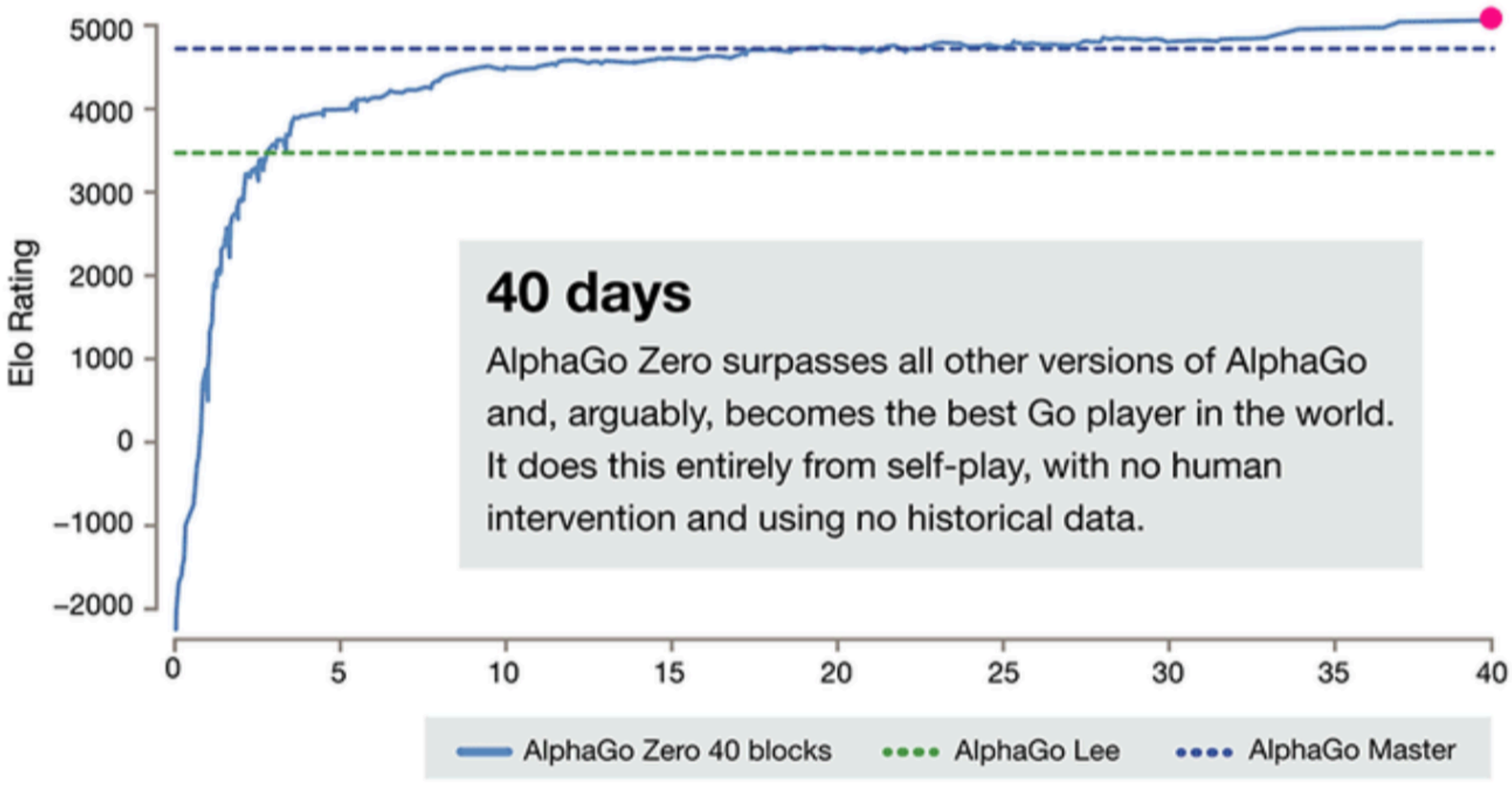
# A single network, Learning with RL

## 1 Reinforcement Learning in *AlphaGo Zero*

Our new method uses a deep neural network $f_\theta$ with parameters $\theta$. This neural network takes as an input the raw board representation $s$ of the position and its history, and outputs both move probabilities and a value, $(\mathbf{p}, v) = f_\theta(s)$. The vector of move probabilities $\mathbf{p}$ represents the probability of selecting each move (including pass), $p_a = Pr(a|s)$. The value $v$ is a scalar evaluation, estimating the probability of the current player winning from position $s$. This neural network combines the roles of both policy network and value network [12] into a single architecture. The neural network consists of many residual blocks [4] of convolutional layers [16,17] with batch normalisation [18] and rectifier non-linearities [19] (see Methods).

# AlphaGo Zero

- Beats AlphaGo by 100:0



# AlphaGo Zero

- Beats AlphaGo by 100:0

Computer Science > Artificial Intelligence

# Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis

*(Submitted on 5 Dec 2017)*

The game of chess is the most widely-studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by human experts over several decades. In contrast, the AlphaGo Zero program recently achieved superhuman performance in the game of Go, by tabula rasa reinforcement learning from games of self-play. In this paper, we generalise this


How Magnus Carlsen Learned From AlphaZero

845K views • 7 months ago

GothamChess

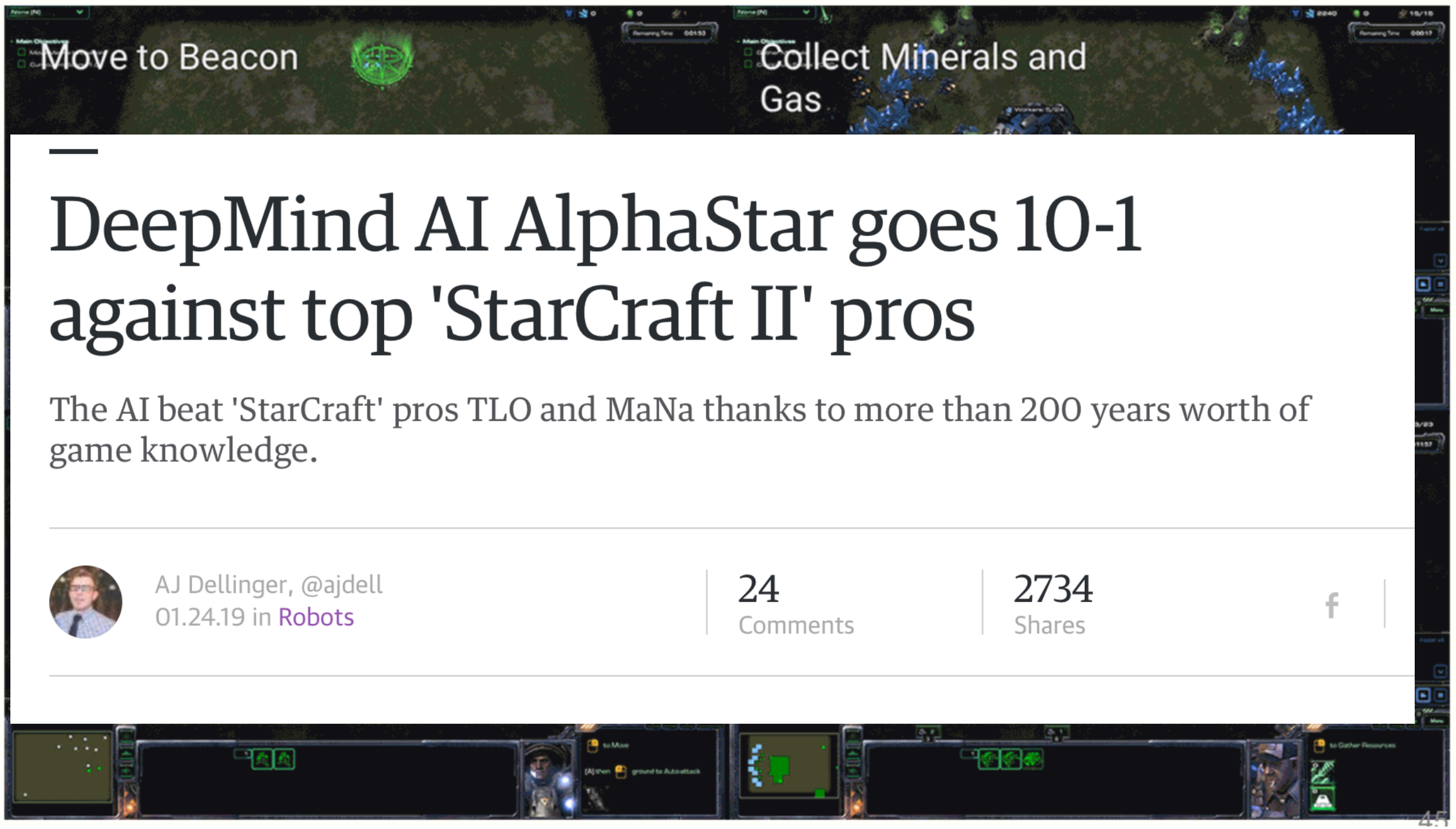
**Magnus Carlsen**. AlphaZero. Talking about his chess style changing and adapting against Grandmasters. Stockfish. Leela

# What's next for AI?

DeepMind's AI is Struggling to Beat Starcraft II - Bloomberg
https://www.bloomberg.com/.../deepmind-master-of-go-struggles-to-crack-its-next-mi...

# What's next for AI?

DeepMind's AI is Struggling to Beat Starcraft II - Bloomberg
https://www.bloomberg.com/.../deepmind-master-of-go-struggles-to-crack-its-next-mi...




# DeepMind AI AlphaStar goes 10-1 against top 'StarCraft II' pros

The AI beat 'StarCraft' pros TLO and MaNa thanks to more than 200 years worth of game knowledge.



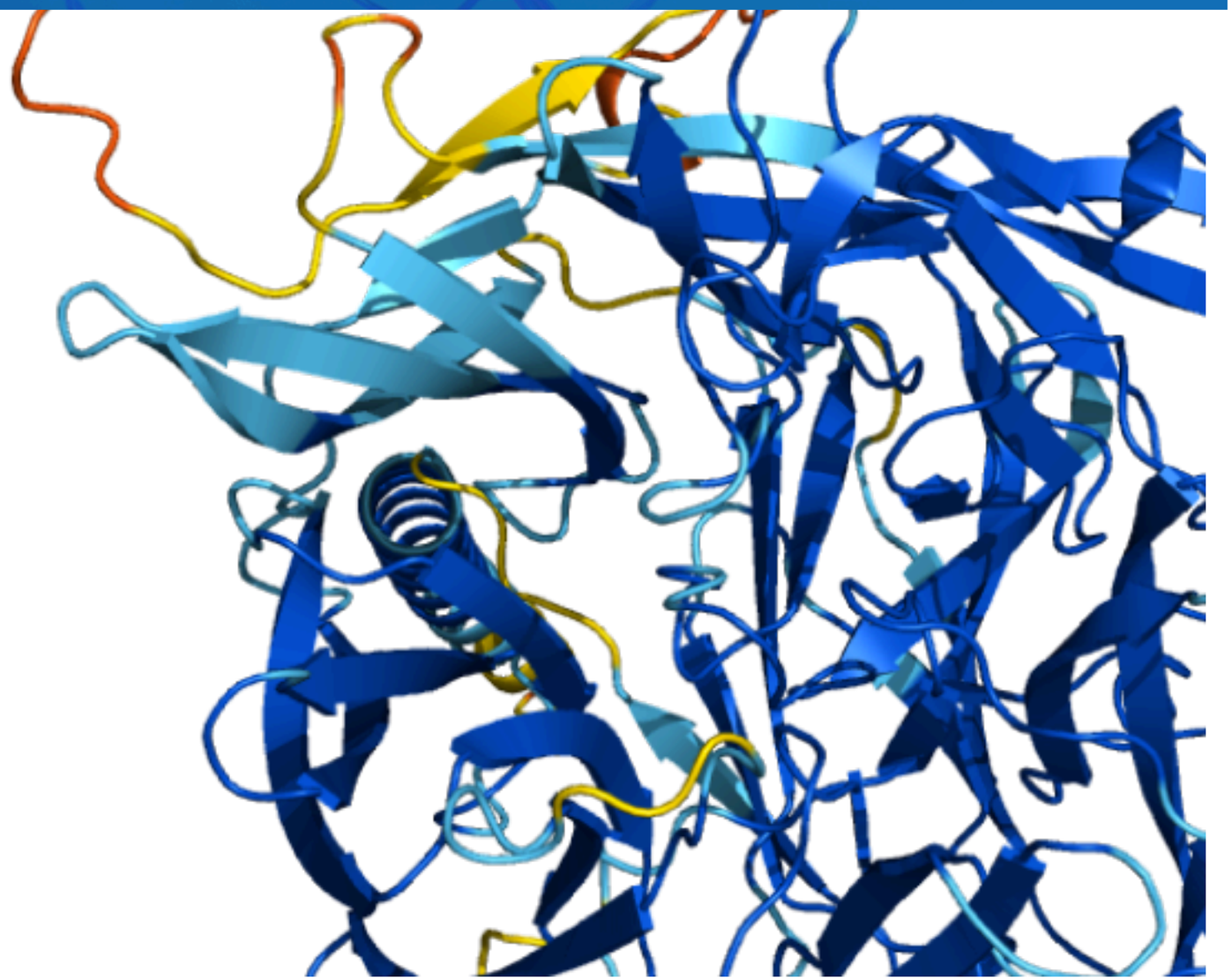
AJ Dellinger, @ajdell
01.24.19 in Robots

24
Comments

2734
Shares

# AlphaFold Protein Structure Database

Developed by DeepMind and EMBL-EBI



# WHAT NEXT???????

## MANY POSSIBLE LECTURES IN EPFL!

EE-559 Deep Learning

EE-618 Reinforcment learning

EE-608 Deep Learning for NLP