Deep reinforcement learning methods
Their advantages and shortcomings

Ashley Hill

CEA, LIST, LCSR

4th May 2020

Who am I?

Ashley Hill, PhD student at CEA Saclay, LIST, LCSR.

Currently working on reinforcement learning for predicting an optimal control gain, in dynamic, uncertain, and noisy environment.

Co-author of the Stable-Baselines reinforcement learning library (details later).

If you have any questions:

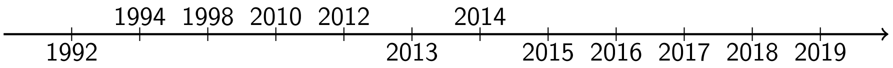
- github@hill-a.me
- ashley.hill@cea.fr

Before we begin...

If you have any questions during the presentations, or if I have not explained things correctly, don't hesitate to interrupt me to ask questions.

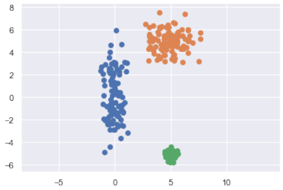
Contents

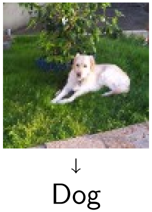
A timeline of deep supervised learning and deep reinforcement learning



- 1992: TD-gammon, one of the first NN RL methods
- 1994: LENET5, one of the first deep convolutional NN
- 1998: Start of AI winter
- 2010: End of AI winter, first GPU NN, DAN CIRESAN NET
- 2012: AlexNet, new high score on image net
- 2013: DQN, RL playing Atari
- 2014: Inception
- 2015: AlphaGo, first victory of an IA against an expert player at GO
- 2016: A2C & DDPG
- 2017: TRPO, PPO & HER
- 2018: TD3, SAC, & OpenAI five
- 2019: AlphaStar, solving a Rubik's cube with one hand, & Deep mimic.

Machine learning overview





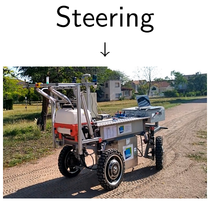


Figure 1: On the left self-supervised example. In the middle supervised example. On the right reinforcement learning example.

ML type	Signal size	Example Tasks
Self-Supervised	Input data	Clustering
Supervised	Output size	Classification, regression
Reinforcement Learning	Sparse scalar	Control, planning

Reinforcement learning: Imitating real world learning

How do children/pets learn in real life?

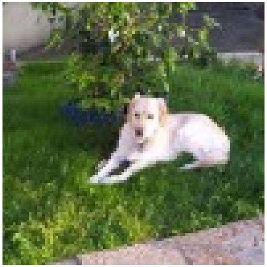


Figure 2: A dog.

For a given stimuli, they act. From said action, feedback is given.
Ex: Hot stove with pain, miss behaving pet with owner, ...

Furthermore, it is model free learning!

Reinforcement learning loop

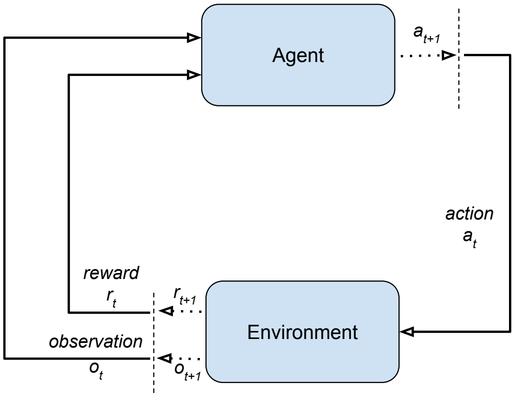


Figure 3: Reinforcement learning feedback loop, some visual similarities with control loops

Markov modeling of the problem

Many real world problems can be seen as a random process:

- Card games (Black jack)
- Random walk
- Yahtzee

Where the random processes has possible states, with a probability of transition from state to state.
A method to model these processes is the Markov models.

Markov property

The Markov property:

Definition

X_n being the state at time n
x_n being the value at time n

$$P(X_n = x_n | X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = P(X_n = x_n | X_{n-1} = x_{n-1})$$

Refers the memory less aspect of random processes.

Markov chain

Example of Markov modeling when the system is autonomous:

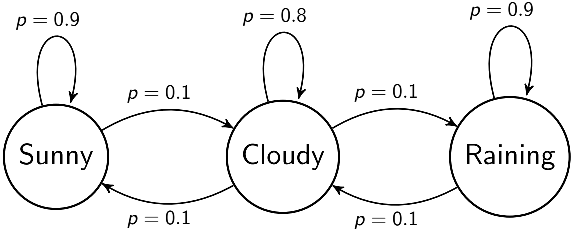


Figure 4: An example of a Markov chain for weather.

Higher chance to stay in a state, cannot change from Sunny to Raining.

Markov decision process

Extending the Markov chain for controlled systems, with actions and rewards:

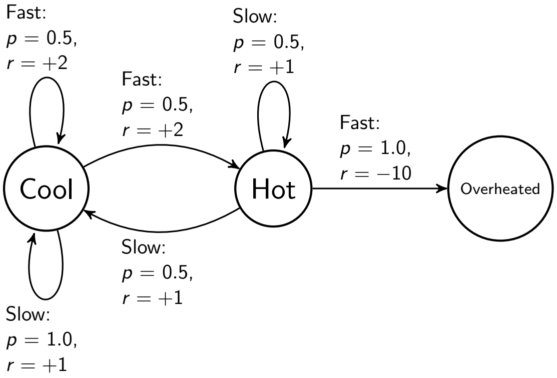

Fast: $p = 0.5$, $r = +2$
Fast: $p = 0.5$, $r = +2$
Slow: $p = 0.5$, $r = +1$
Fast: $p = 1.0$, $r = -10$
Cool
Hot
Overheated
Slow: $p = 0.5$, $r = +1$
Slow: $p = 1.0$, $r = +1$

Figure 5: An example of a Markov decision process for a racing car.

Reinforcement learning loop

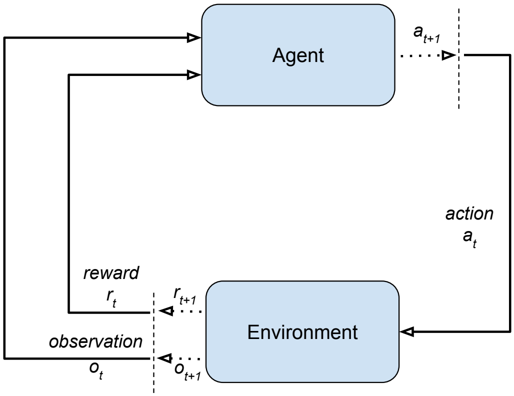
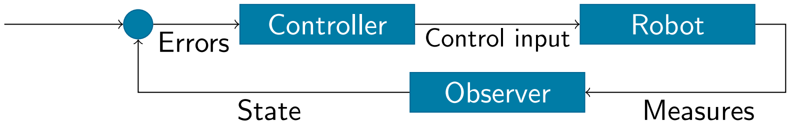


Figure 6: Reinforcement learning feedback loop, some visual similarities with control loops

Markov modeling from a control loop

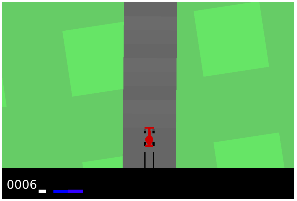


- The observation in the control loop, are the states s_t.
- The actions a_t, are the controller's output.

Reward function

The reward function is defined by an expert.
It returns a quality assessment of a given transition.
For example:

- Racing car: $r_t = |y_t| - |y_{t-1}|$
- Robotic arm: $r_t = |d_t| - |d_{t-1}|$



Objective function

From Sutton's book[1] (one of the best references for RL):

Definition

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

The goal of reinforcement learning is to maximize the cumulative sum of the reward.

$$G_t = \sum_{k=0}^{\infty} r_{t+k+1}$$

[1] Sutton, Barto, et al., *Introduction to reinforcement learning*.

Return & discount

However, calculating the cumulative sum on a continuous task reveals a problem: a diverging sum.

As such we add a new notion, the *discount factor* γ. Which gives us the *return*, a exponential decay of the reward over time. Setting a γ less than one, favors immediate reward:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The intuitive idea:

1000€ now $>$ 1000€ in 1 year $>$ 1000€ in 100 years

Q-Value & Value function

How do we solve problems with this modeling.

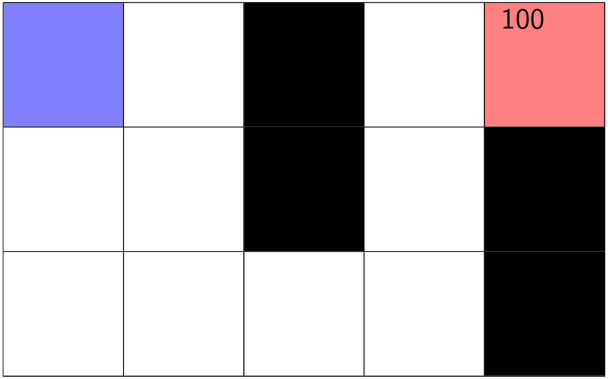


Table 1: Classic labyrinth problem: Getting from the blue area to the red area.

A method to converge to the highest cumulative reward is needed...

Q-Value & Value function

In the case of reinforcement learning, ideally we want to maximize the expected return.
The expected return for a given states is encoded as the *Value function*:

$$V(s) = \mathbb{E}[G_t | s_t = s]$$

The expected return for a given states and action is encoded as the *Q value*:

$$Q(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a]$$

Q-Value & Value function

Using a discount of 0.9, $V(s) = \mathbb{E}[\sum_{k=0}^{T-t-1} 0.9^k r_{t+k+1} | s_t = s]$

43	48	■	90	100
48	53	■	81	■
53	59	66	73	■

Table 2: Classic labyrinth problem: Getting from the blue area to the red area.

Rooms that are closer to the end, will have a higher $V(s)$. Actions that lead to the end for a given state, will have a higher $Q(s, a)$.

Time difference – Bellman equation

Bellman optimization for $V(s)$:

$$V(s) = \mathbb{E}[G_t | s_t = s]$$

$$V(s) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1}) | s_t = s]$$

For $Q(s, a)$ we get:

$$Q(s, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') | s_t = s, a_t = a]$$

Contents

Labyrinth example

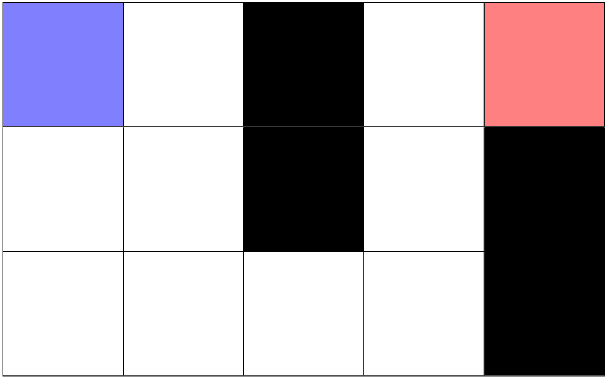


Table 3: Classic labyrinth problem: Getting from the blue area to the red area.

This is a discrete action, discrete state problem. It can be solved using dynamic programming, or tabular Q-learning.

Control example: mountain car

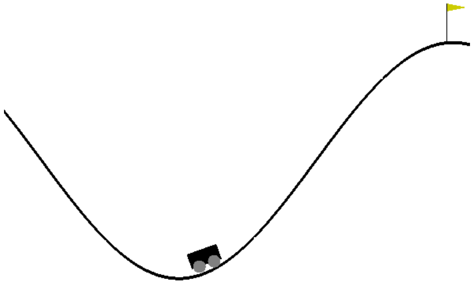


Figure 7: Mountain car: the reach the goal by building up inertia.

This is a discrete action, **continuous** state problem. It **cannot** be solved using dynamic programming, or tabular Q-learning. For this, we would need to either discretize the state space or use a approximate function for estimating the value functions.

Control example: mountain car

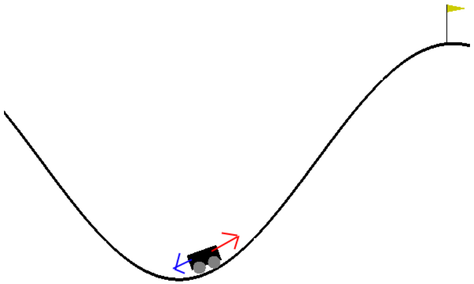


Figure 8: Mountain car: the reach the goal by building up inertia. the a_0 action in denoted in blue, the a_1 action is denoted in red.

For example, the action a_1 will lead to a higher $V(s)$, as the reward is higher when the car is near the goal.

Function approximation

There exist many function approximations methods:

- Linear
- Polynomial
- Neural Network
- ...

In the context of this class, we will focus on Neural Networks, as they are able to estimate achieve a non-linear estimation of the Q-value.

Neural network based Q-Value

Using a neural network as a universal function estimator [1] for the mapping from the state space to each Q value for every discrete action:
$(S \mapsto Q(s, a_n), \forall a_n \in A)$

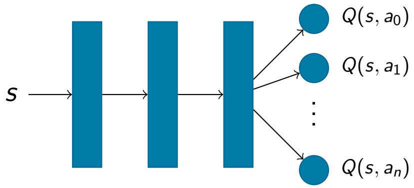


Figure 9: Neural network based Q-value, architecture for discrete actions.

[1] Hornik, Stinchcombe, and White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks".

The TD error

$$Q(s,a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') | s_t = s, a_t = a]$$

The *TD-error* is defined as such:

$$TD_{err} = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) | s_t = s, a_t = a]$$

It represent the error between the current Q-value, and the $t+1$ approximation. From this:

$$Q(s,a) = Q(s,a) + \alpha\, \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) | s_t = s, a_t = a]$$

Where α is the learning rate.

The loss function

With this, loss function is relatively simple, it is the mean squared error of the *TD-error* over the states and actions:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\rho;s'\sim\text{Env}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right)^2\right]$$

Where ρ is defined as the behavior distribution, in our case generated from $\pi_\epsilon(s)$.

Exploration vs Exploitation

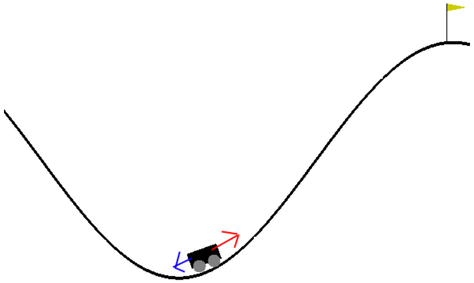


Figure 10: Mountain car: the reach the goal by building up inertia. the a_0 action in denoted in blue, the a_1 action is denoted in red.

Now in this environment, the motor is not strong enough to go up the hill. Using the highest Q-value (greedy Q value) will lead to a "burn-in", as no exploration occur es and the Q-value is updated with the over-explored regions.

Exploration vs Exploitation

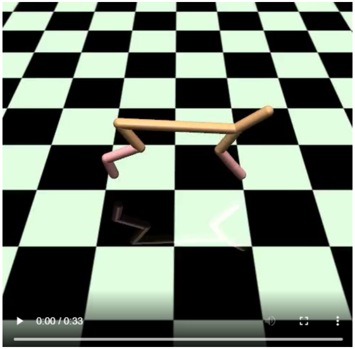


Figure 11: https://drive.google.com/file/d/1HIXdvY07VUSBFOANHRPJ-UeEt6DoOOJE/view?usp=sharing

Explore too much, learn nothing of value. Exploit too much, "burn-in" bad policies.

Exploration: Epsilon greedy

As with most reinforcement learning, the method needs to explore the environment in order to learn the estimated Q-value.
For this, an ϵ-*greedy* policy can be used:

$$\pi_\epsilon(s) = \begin{cases} \pi(s) & \epsilon < x \\ a_n,\ n \sim \mathcal{U}(0, |A|) & \epsilon \geq x \end{cases}, \text{ with } x \sim \mathcal{U}(0,1)$$

With often $\pi(s) = \text{argmax}_a\, Q(s,a)$ for Q-learning.

This allows for some random exploration, while still exploiting with the optimal policy. In practice, ϵ varies over time from 1.0 down to a lower value, to explore early, and exploit later.

Unstable however

Then putting this all together, and doesn't work...
We have encountered what Sutton [2] calls *The deadly triad*:

- **Function approximation**: Such as neural networks.
- **Bootstrapping**: Update targets that include existing estimates (TD methods do this)
- **Off-policy training**: Due to the $\max_{a'} Q(s, a'; \theta_i)$ update value, not being based on the target policy $\pi_\epsilon(s)$.

This is one of the reasons, the neural network over estimates the Q value, and losses stability.

[2]Sutton, Barto, et al., *Introduction to reinforcement learning*.

Unstable however: Deadly triad example

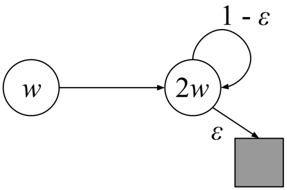


Figure 12: A Markov chain, demonstrating the Deadly triad. The reward is always 0. (image from the sutton's book[3])

for a linear value estimation $v_w(s) = w \times s$:

$$\Delta_w \propto (r_t + \gamma v_w(s_{t+1}) - v_w(s_t))\nabla_w v_w(s_t)$$

at s_0:

$$\Delta_w \propto (\gamma w 2 - w)$$

[3]Sutton, Barto, et al., *Introduction to reinforcement learning*.

Stabilizing

Isolation from Off-policy would be complicated, however the two other aspects of *The deadly triad* can be address using:

- **Experience replay**: Update targets with past observations, mitigating the **Bootstrapping**.
- **Fixed target Q network**: Have 2 Q networks, one for the Q value, the other for the target Q value, mitigating the **Function approximation**.

Experience replay

At each timestep t:

- store the transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$.
- sample a random mini-batch of transition (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$.
- calculate the gradient descent with $(r_j + \gamma \max_{a'} Q(s_j + 1, a'; \theta) - Q(s_j, a_j; \theta))^2$ for the entire mini-batch.
- back-propagate the mini-batch gradient.

This allows the method to "relive" past transition at every timestep, mitigating the issues related with **Bootstrapping**.

Fixed target Q network

2 Q networks, Q and Q_t.
For each timestep:

- update the Q network

For every n timesteps:

- Copy the weights from the Q network to the target Q_t network

This allows for a consistent target Q value, lowering the variance, and mitigating the issues related with **Function approximation**.

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a\sim\rho;s'\sim\mathrm{Env}}\left[\left(r + \gamma \max_{a'} Q_t(s',a';\theta) - Q(s,a;\theta)\right)^2\right]$$

Deep Q network (DQN) method

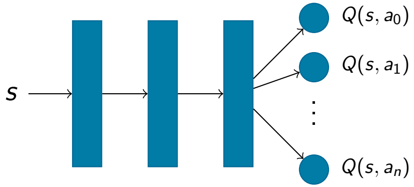


Figure 13: Neural network architecture of the DQN method.

- $\mathcal{L}(\theta_i) = \mathbb{E}_{s,a\sim\rho;s'\sim\text{Env}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2\right]$
- Experience replay (sample efficiency increase)
- Fixed target Q network
- ϵ-greedy exploration

DQN success: Playing Atari games

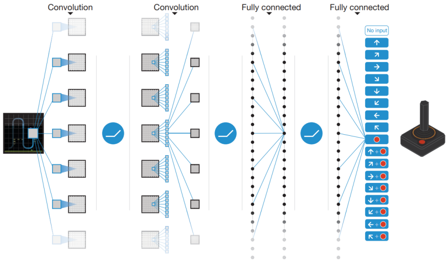


Figure 14: The DQN architecture used in the DQN nature paper [4] (image from said paper).

Video: https://youtu.be/V1eYniJ0Rnk?t=18

Added features to play Atari games:

- Frame stacking
- Frame skipping
- Preprocessing image
- Convolutional neural network

Contents

Control example: mountain car

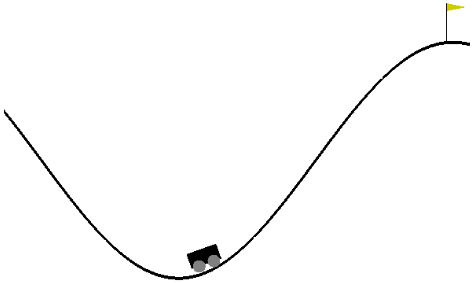


Figure 15: Mountain car: the reach the goal by building up inertia.

This is a discrete action, **continuous** state problem. It **cannot** be solved using dynamic programming, or tabular Q-learning. For this, we would need to either discretize the state space or use a approximate function for estimating the value functions.

Car example: continuous action

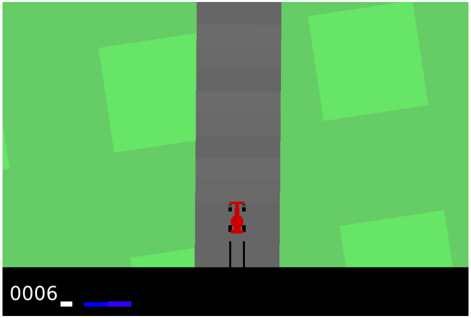


Figure 16: Car Racing: finish the lap in the fastest time, without leaving the track.

This is a **continuous** action, continuous state problem. It **cannot** be solved using the DQN method. For this, we would need to either discretize the action space or find the mapping from the state space to the action space.

Need function from state to action

Transitioning from discrete actions to continuous actions, means we can no longer use the trick we used with DQN, as there are a quasi-infinite number of actions possible.

Lets assume we have a neural network called the actor: $\mu(\theta_\mu) : S \mapsto A$. In this case, the Q value can be a neural network we will call the critic: $Q(\theta_Q) : s \in S, a \in A \mapsto Q(s, a)$.

The loss function for the critic is similar to the DQN method:
$\mathcal{L}(\theta_Q) = \mathbb{E}_{s,a\sim\rho;s'\sim\mathrm{Env}}\left[(r + \gamma Q(s', \mu(s', \theta_\mu); \theta_Q) - Q(s, a; \theta_Q))^2\right]$

A solution: Deterministic Policy Gradient

Deterministic policy gradient is depended on the Q-value from the critic using the action from the actor, and defines:

$$\nabla_{\theta_\mu} \mathcal{L}(\theta_\mu) = \mathbb{E}_s \left[\nabla_{\theta_\mu} Q(s, \mu(s; \theta_\mu); \theta_Q) \right]$$

Intuitively, this implies the direction of gradient for Q is the direction of gradient for θ_μ

The cause of this, is the critic must converge before the actor can converge correctly.

A solution: Deterministic Policy Gradient

Assuming the Q-value has converged:

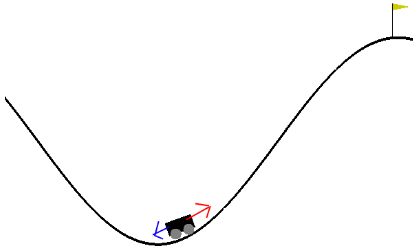


Figure 17: Mountain car: the reach the goal by building up inertia. the a_0 action in denoted in blue, the a_1 action is denoted in red.

$$\nabla_{\theta_\mu} \mathcal{L}(\theta_\mu) = \mathbb{E}_s \left[\nabla_{\theta_\mu} Q(s, \mu(s; \theta_\mu); \theta_Q) \right]$$

The gradient of the Q-value, is towards the goal. This means the equation has as gradient the for the action, the direction of higher Q-values.

Actor critic: sequential setup

$Actor : S \mapsto A \; Critic : s \in S, a \in A \mapsto Q(s, a)$

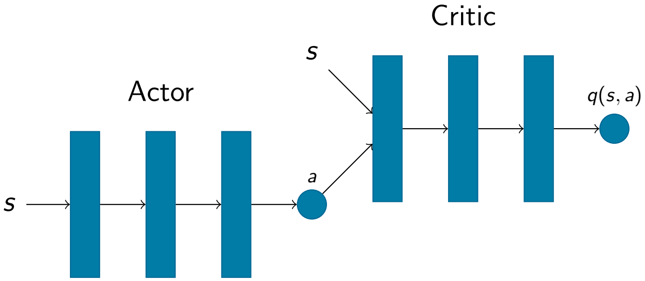


Figure 18: The sequential actor critic neural network architecture.

Exploration: random noise to action

Exploration is still needed due to the problem underlined with DQN. However we cannot use the ϵ-greedy method for exploration.

As such, we sample noise from a Gaussian, and add it to the action:

$$\pi(s) = \mu(s; \theta_\mu) + x, \text{ where } x \sim \mathcal{N}(0, a_\sigma)$$

Usually, a_σ is initialized depending on the environment, and is slowly reduced.

Exploration: random noise policy network

An alternative to generate exploration, is to add the noise directly to the weights and biases defined in θ_μ:

$$\pi(s) = \mu(s; \theta_\mu + x), \text{ where } x \sim \mathcal{N}(0, a_\sigma)$$

As with the action noise, a_σ is slowly reduced. This is environment independent, and allows for exploration of the policy space without prior knowledge of the action amplitudes.

Deep Deterministic Policy Gradient (DDPG) method

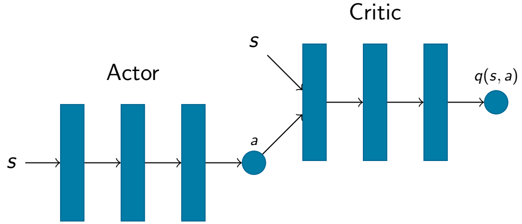


Figure 19: Neural network architecture of the DDPG method.

- $\mathcal{L}(\theta_Q) = \mathbb{E}_{s,a\sim\rho;s'\sim\text{Env}}\left[\left(r + \gamma Q_t(s', \mu_t(s', \theta_\mu); \theta_Q) - Q(s,a;\theta_Q)\right)^2\right]$
- $\mathcal{L}(\theta_\mu) = \mathbb{E}_s\left[Q_t(s, \mu(s;\theta_\mu);\theta_Q)\right]$
- Experience replay
- Fixed target networks (actor and citic)
- Action noise or Policy noise

DDPG success: MuJoCo

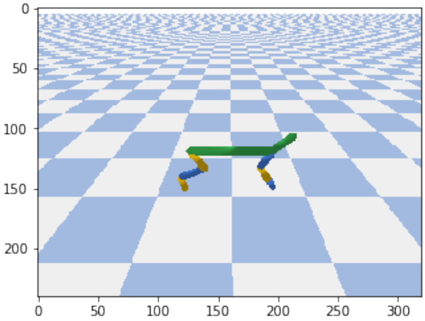


Figure 20: The HalfCheetah MuJoCo environment used in the DDPG paper[5]. The angle of each segment is controlled by the reinforcement learning method directly.

Video: https://youtu.be/iFg5IcUzSYU?t=14

[5] Lillicrap et al., "Continuous control with deep reinforcement learning".

Contents

Moving towards policy gradient methods

	On-policy	Off-policy
value function	SARSA	Q-learning, DQN
policy gradient	A2C, PPO	DDPG

Up until now, we have seen Off-policy methods, meaning our search policy is distinct from our target policy.

However, we will be moving towards On-policy policy gradient methods. Less sample efficient (no Experience replay), but are more stable (avoiding the Deadly triad) and optimize the policy directly.

How good is the action

We would like to find a indication of how good an action is in a given state, to avoid penalizing an action in an overall bad state:

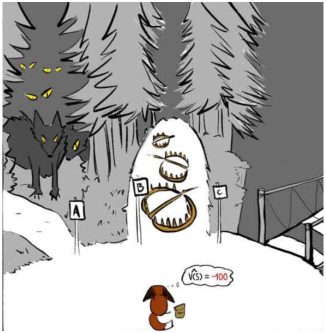


Figure 21: A bad state for the fox, but some actions are still better than others. (from https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752)

$$A(s, a) = Q(s, a) - V(s)$$

How good is the action

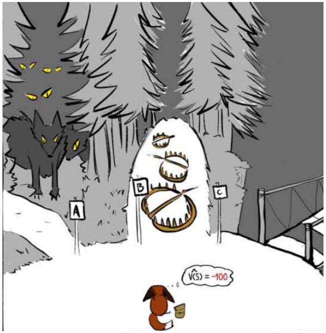


Figure 22: A bad state for the fox, but some actions are still better than others. (from https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752)

- Path A: $Q(s,a) = -100$, $A(s,a) = (-100) - (-100) = 0$
- Path B: $Q(s,a) = -150$, $A(s,a) = (-150) - (-100) = -50$
- Path C: $Q(s,a) = -20$, $A(s,a) = (-20) - (-100) = 80$

How good is the action: Advantage function

This notion of action quality for a given state is called the *Advantage*:

$$A(s, a) = Q(s, a) - V(s)$$

However, this means calculating the value function and the Q value, which is inefficient. Luckily we can do this:

$$Q(s, a) = \mathbb{E}\left[r_{t+1} + \gamma V(s_{t+1}) \middle| s_t = s, a_t = a\right]$$

$$A(s, a) = \mathbb{E}\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \middle| s_t = s, a_t = a\right]$$

Advantage has some strong intuitive sense to it.
We can now use the advantage function to optimize the policy, in order to more accurately target the wanted behavior (e.i take the action that maximize return).

Loss: Actor and Critic

Critic loss is identical to the previous methods (TD error):

$$\mathcal{L}_c(\theta) = \mathbb{E}_{s,r,s' \sim \text{ENV}} \left[(r + \gamma v(s') - v(s))^2 \right]$$

Actor loss is defined as the probability of taking the action, time the advantage of said action:

$$\mathcal{L}_a(\theta) = \mathbb{E}_{s,a} \left[\log(\pi_\theta(s,a)) A(s,a) \right]$$

Actor critic: parallel setup

$Critic : s \in S \mapsto V(s)\ Actor : S \mapsto A$

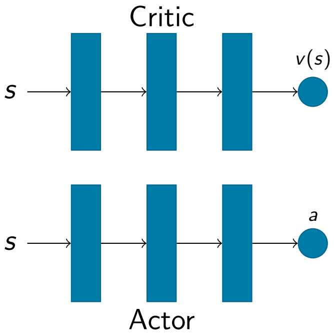
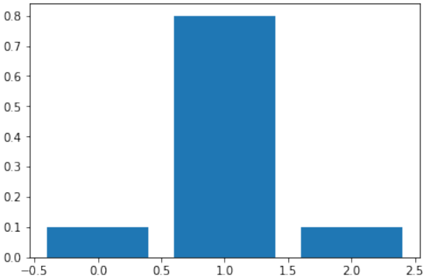
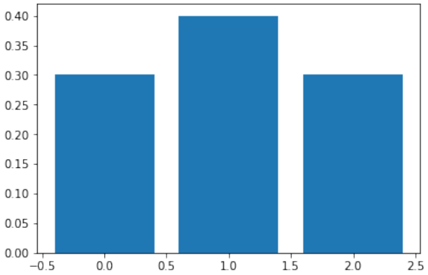


Figure 23: The parallel actor critic neural network architecture.

Exploration: entropy loss





Using Shannon's information entropy ($H(x) = -\sum_{i=0}^{n} P(x_i)\log_2(P(x_i))$):

- The first graph has an entropy of $H(x) = 0.922$ bits
- The second graph has an entropy of $H(x) = 1.571$ bits

Exploration: entropy loss

The actor will converge too quickly, we need a loss function to go against this. The entropy of the actor output will be used to keep the exploration high.

Entropy:

$$H(X) = -\sum_{i=0}^{n} P(x_i) \log_2(P(x_i))$$

Entropy loss:

$$\mathcal{L}_H(\theta) = \mathbb{E}_{s,a}\left[-\sum_{i} \pi_\theta(s,a) \log_2(\pi_\theta(s,a))\right]$$

Actor loss:

$$\mathcal{L}_a H(\theta) = \mathcal{L}_a(\theta) - c_H \mathcal{L}_H(\theta)$$

Exploration: mean and variance

A good way to explore, is to let the actor defined the noise:

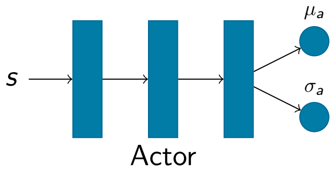


Figure 24: The actor that outputs mean and standard deviation of the actions.

This lets the method become On-policy, while having a stochastic policy:

$$\pi_\theta(s) \sim \mathcal{N}(\mu_a, \sigma_a)$$

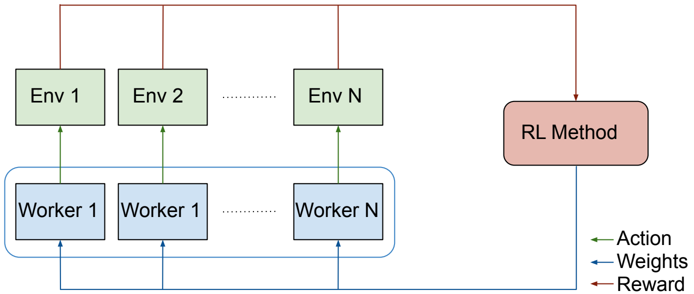
And, when the method is not training, we can directly output the mean:

$$\pi_\theta(s) = \mu_a$$

This is accomplished using a Softplus on the σ $y = \log(1 + \exp(x))$, and entropy loss avoid a collapse to zero on σ.

Workers

The gradient from the reward is noisy. As such, multiple agents can be used to average the reward signal, and improve the gradient:



This allows much faster convergence, as the cost of computational power.

Advantage Actor Critic (A2C) model

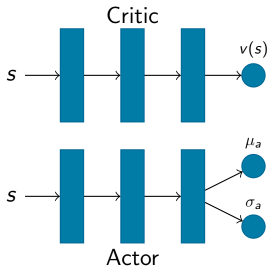


Figure 25: Neural network architecture of the A2C method[6].

- $\mathcal{L}_a H(\theta) = \mathcal{L}_a(\theta) - c_H \mathcal{L}_H(\theta)$
- $\mathcal{L}_c(\theta) = \mathbb{E}_{s,r,s' \sim \text{ENV}} \left[(r + \gamma v(s') - v(s))^2 \right]$
- In practice: $c_H = 0.01$, however this is tune able for a given task.
- Workers

[6] Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning".

REINFORCE success: AlphaGo Zero (with MCTS)

REINFORCE is used in AlphaGo Zero, the key difference with A2C is the actor loss $\mathcal{L}_a(\theta) = \mathbb{E}_{s,a}\left[-\log(\pi_\theta(s,a))v(s)\right]$

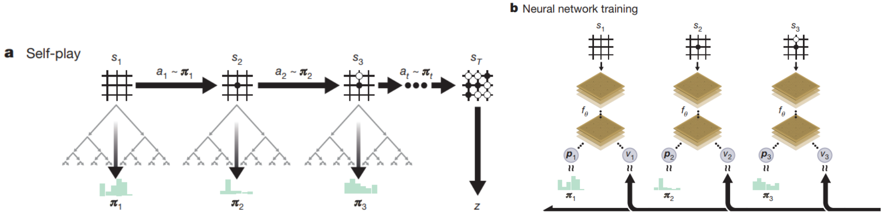


Figure 26: The training method for AlphaGo Zero from the paper[7] (images from said paper)

They also added self play and a Monte Carlo tree search to look ahead for the best moves.

[7] Silver et al., "Mastering the game of go without human knowledge".

Contents

Sample efficiency

Sample efficiency is defined as the amount of samples required to reach a decent policy. This is very context dependent, if the policy required is complex enough, then Off-policy might struggle converging, making On-policy methods better.

As an example, here is the a donkey car being trained in 7 minutes using Soft Actor-Critic (SAC):

Figure 27: https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb35f4, Video: https://youtu.be/iiuKh0yDyKE

Classification of the models, pros and cons

	tabular Q-learning	DQN	DDPG	A2C
Off-policy	✓	✓	✓	✗
On-policy	✗	✗	✗	✓
Experience replay	✗	✓	✓	✗
Continuous states	✗	✓	✓	✓
Discrete actions	✓	✓	✗	✓
Continuous actions	✗	✗	✓	✓
Multi-process	✗	✗	✗	✓

Table 4: The characteristics of each method

Which one to use?

As with most things, it depends.

For robotics: Off-policy with Experience replay are a good idea, as they are sample efficient.

For simulation, games, trading: On-policy for the stability and the fast convergence to local optima.

You will need to test multiple reinforcement learning algorithms to verify which one has the best performance.

Reward shaping

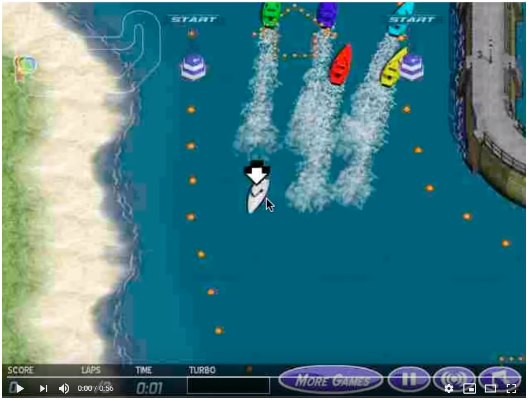


Figure 28: https://youtu.be/tlOIHko8ySg

Hyperparameter sensitivity

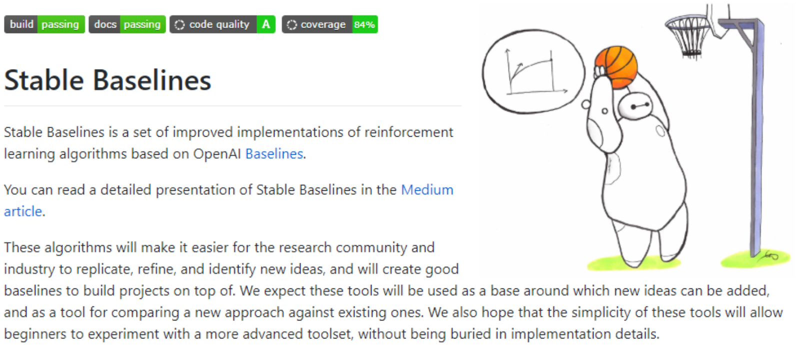
Running some reinforcement learning algorithms "out of the box" might not work and be unstable in the given environment. Reinforcement learning is particularly sensitive to hyperparameters.

As such, it is best to setup your environment and model, with a grid search or a hyperparameter search algorithm, such as Optuna (https://optuna.org/).

Contents

Stable baselines

Stable-baselines, is a reinforcement learning library that is designed to be user friendly with an sklearn-like interface.
Initially a fork of Open AI Baselines, it is has been cleaned up to the pep8 standards, fully commented, fully documented, includes tests, includes CI, includes major fixes, includes new algorithms (TD3 and SAC), and has been battle hardened.



Stable Baselines

Stable Baselines is a set of improved implementations of reinforcement learning algorithms based on OpenAI Baselines.

You can read a detailed presentation of Stable Baselines in the Medium article.

These algorithms will make it easier for the research community and industry to replicate, refine, and identify new ideas, and will create good baselines to build projects on top of. We expect these tools will be used as a base around which new ideas can be added, and as a tool for comparing a new approach against existing ones. We also hope that the simplicity of these tools will allow beginners to experiment with a more advanced toolset, without being buried in implementation details.

Stable baselines: fast prototyping

As with most python, just import it!

```
from stable_baselines import A2C

model = A2C('MlpPolicy', 'CartPole-v1').learn(10000)
```

The A2C model will train for 10000 timesteps, on the CartPole environment, using a multi-layer perceptron policy.

You want to try images with DQN?

```
from stable_baselines.common.atari_wrappers import
    make_atari
from stable_baselines import DQN

env = make_atari('BreakoutNoFrameskip-v4')
model = DQN('CnnPolicy', env).learn(10000)
```

Done, training for an Atari game with frame stacking.

Stable baselines: Compare quickly

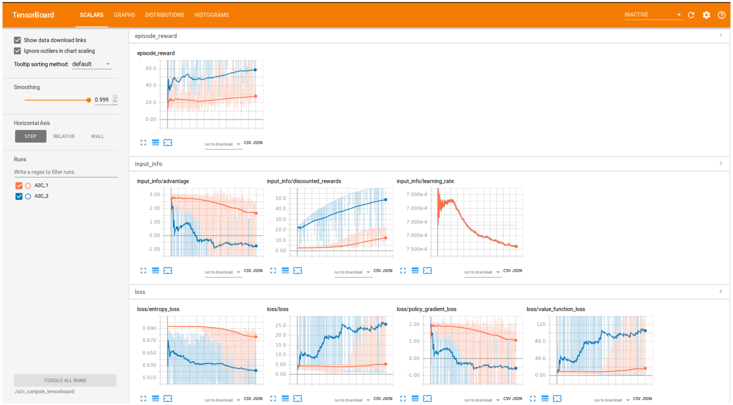


Figure 29: Tensorboard screenshot of 2 A2Cs training with different hyper parameters, on cartpole.

Stable baselines: Understand the architecture

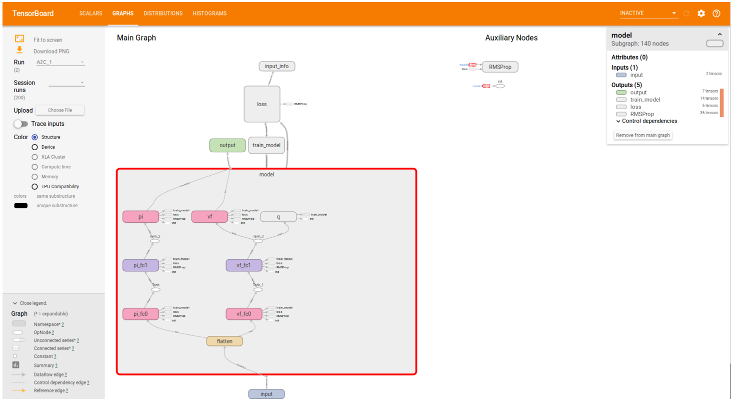


Figure 30: Tensorboard screenshot of the A2Cs tensor graph.

TP

TP de prise en main Stable-baselines:
https://github.com/araffin/rl-tutorial-jnrr19

Dans le Readme section Content, ouvrez les liens "Colab Notebook" pour chaque notebook.

Bibliography I

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: *Neural Networks* 3.5 (1990), pp. 551 –560. ISSN: 0893-6080.

Lillicrap, Timothy P. et al. "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971 (2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

Mnih, Volodymyr et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

Mnih, Volodymyr et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/.

Bibliography II

Schulman, John et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

Silver, David et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

Sutton, Richard S, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 2. 4. MIT press Cambridge, 1998.

Contents

OpenAI Gym interface

The stable baselines library, uses the OpenAI Gym interface. This interface must have 4 functions:

- __init__(*args, **kwargs): as with all python classes. Initializes an instance of the environment.
- reset(): resets the environment, returns the first observation
- step(action): takes an action as an array, does one step in the environment, and returns the observation, reward, is_done, and info (a dict)
- render(type): takes a string of the type of rendering, and renders an image of the environment.

and 2 variable:

- observation_space: The shape, size, and type of observation space.
- action_space: The shape, size, and type of action space.

Stable baselines: why was a fork needed?

- fixed tf.session().__enter__() being used, rather than sess = tf.session() and passing the session to the objects
- fixed uneven scoping of TensorFlow Sessions throughout the code
- fixed rolling vecwrapper to handle observations that are not only grayscale images
- fixed deepq saving the environment when trying to save itself
- fixed ValueError: Cannot take the length of Shape with unknown rank. in acktr, when running run_atari.py script.
- fixed calling baselines sequentially no longer creates graph conflicts
- fixed mean on empty array warning with deepq
- fixed kfac eigen decomposition not cast to float64, when the parameter use_float64 is set to True
- fixed Dataset data loader, not correctly resetting id position if shuffling is disabled
- fixed EOFError when reading from connection in the worker in subproc_vec_env.py
- fixed behavior_clone weight loading and saving for GAIL
- avoid taking root square of negative number in trpo_mpi.py
- fixed render function ignoring parameters when using wrapped environments
- fixed numpy warning when using DDPG Memory
- fixed DummyVecEnv not copying the observation array when stepping and resetting
- fixed graphs issues, so models wont collide in names
- fixed behavior_clone weight loading for GAIL
- fixed Tensorflow using all the GPU VRAM
- fixed models so that they are all compatible with vectorized environments
- fixed ‘set_global_seed‘ to update ‘gym.spaces‘’s random seed
- fixed PPO1 and TRPO performance issues when learning identity function
- fixed DQN wrapping for atari
- fixed ACER buffer with constant values assuming n_stack=4
- fixed some RL algorithms not clipping the action to be in the action_space, when using ‘gym.spaces.Box‘
- removed unused, undocumented and crashing function reset_task in subproc_vec_env.py
- ...

Partially observable Markov decision process

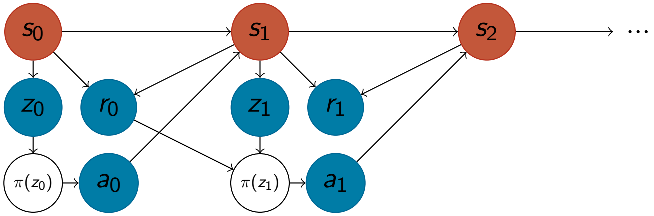


Figure 31: A POMDP graph. In blue the observable information, in red the hidden information, and in white the policy.

When a policy is applied to a POMDP, it is reduced to a hidden Markov model.

Partially observable Markov decision process

There are multiple levels of POMDPs:

- Temporal: The state can be reconstituted from the observation using temporal information (eg: speed, acceleration, ...).
- Hidden information: The observation lack variables needed to rebuild the state (eg: terrain quality, temperature, ...).
- Unknown external agent: (eg: poker, dota, ...)

POMDPs are not classified strictly, but are more a spectrum from most observable to least observable.

Some of these can be mitigated with recurrent neural networks, frame stacking, and a lot of training time. (OpenAI Five for example)

Contents

Number recognition - MNIST

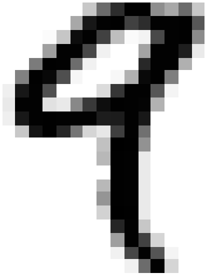


Figure 32: MNIST handwritten 9.

Neural network

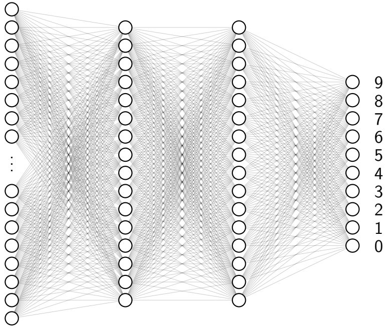


Figure 33: An example of a neural network for number recognition.

- Map from input data to a desired output
- $O(N)$ complexity
- Higher level representation for each layer.

Perceptron - Linear classifier

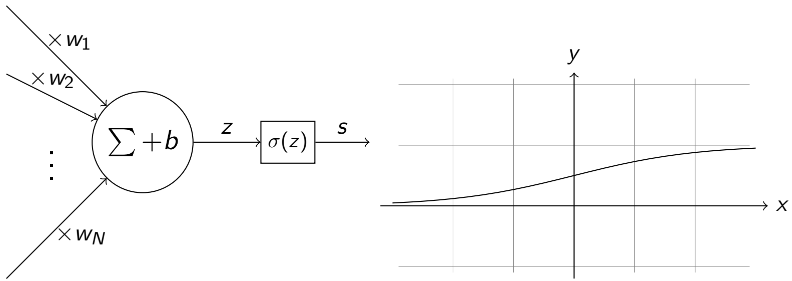
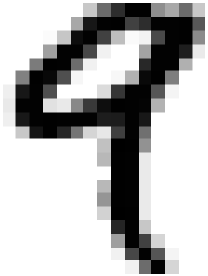
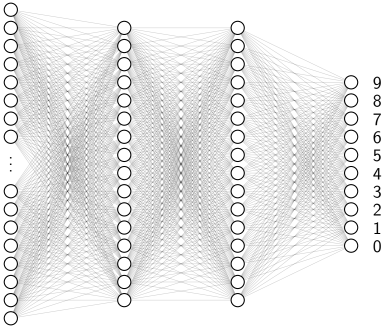


Figure 34: An example of a perceptron and the sigmoid activation function.

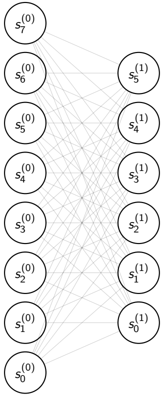
$$s = \sigma\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

Perceptrons in a network





Multi Perceptron Network (MLP)

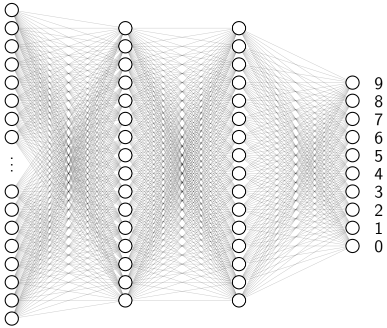


$$s_5^{(1)} = \sigma\left(w_{5,0}^{(0,1)}\, s_0^{(0)} + w_{5,1}^{(0,1)}\, s_1^{(0)} + \dots + w_{5,7}^{(0,1)}\, s_7^{(0)} + b_5^{(1)}\right)$$

$$\begin{bmatrix} s_0^{(1)} \\ s_1^{(1)} \\ \vdots \\ s_5^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{0,0}^{(0,1)} & w_{0,1}^{(0,1)} & \dots & w_{0,7}^{(0,1)} \\ w_{1,0}^{(0,1)} & w_{1,1}^{(0,1)} & \dots & w_{1,7}^{(0,1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{5,0}^{(0,1)} & w_{5,1}^{(0,1)} & \dots & w_{5,7}^{(0,1)} \end{bmatrix} \begin{bmatrix} s_0^{(0)} \\ s_1^{(0)} \\ \vdots \\ s_7^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ \vdots \\ b_5^{(1)} \end{bmatrix}\right)$$

$$s^{(1)} = \sigma\left(w^{(0,1)}\, s^{(0)} + b^{(1)}\right)$$

Multi Perceptron Network (MLP)



$$s^{(3)} = \sigma\left(b^{(3)} + w^{(2,3)}\sigma\left(b^{(2)} + w^{(1,2)}\sigma\left(b^{(1)} + w^{(0,1)}x\right)\right)\right)$$

MLP - Universal function approximator

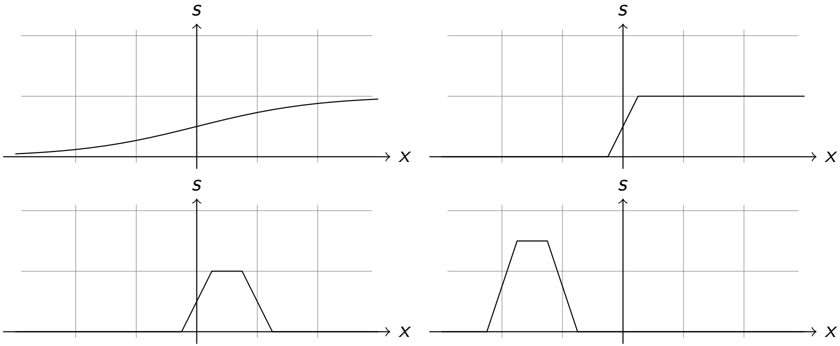


Figure 35:

$$s = \sigma\left(wx + b\right)$$

MLP - Gradient descent optimization

$$s^{(3)} = \sigma\left(b^{(3)} + w^{(2,3)}\sigma\left(b^{(2)} + w^{(1,2)}\sigma\left(b^{(1)} + w^{(0,1)}x\right)\right)\right)$$

Find the change in weights that minimize a cost function:

$$w^{(1,2)} = w^{(1,2)} + \alpha\frac{\partial C}{\partial w^{(1,2)}}$$

Calculable with the chain rule (dérivation des fonctions composées):

$$\frac{\partial C}{\partial w^{(1,2)}} = \frac{\partial C}{\partial s^{(3)}}\frac{\partial s^{(3)}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial s^{(2)}}\frac{\partial s^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial w^{(1,2)}}$$

$$\frac{\partial C}{\partial w^{(1,2)}} = C'\sigma'(z^{(3)})w^{(2,3)}\sigma'(z^{(2)})s^{(1)}$$

A2C shortcomings

A2C is a good method, it does however have a few shortcomings:

- **Policy gradient can be large**: Calculating the gradient for the policy, can induce very large changes in the actor, causing the policy behavior to change drastically.
- **Noisy Advantage function**: Unfortunately, the advantage can be a noisy signal, limiting the capacity of methods such as A2C

PPO addresses most of these short comings, though Policy gradient clipping, and a better estimation of the Advantage function.

Generalized Advantage Estimation

With δ_t^V as the TD-error:

$$\hat{A}_t^{(1)} = \delta_t^V = -V(s_t) + r_{t+1} + \gamma V(s_{t+1})$$

This can be unrolled:

$$\hat{A}_t^{(2)} = \delta_t^V + \gamma \delta_{t+1}^V = -V(s_t) + r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+1})$$

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) \sum_{l=0}^{k-1} \gamma^l r_{t+1+l} + \gamma^k V(s_{t+k+1})$$

Generalized Advantage Estimation

Taking a exponential weighted average of $\hat{A}_t^{(k)}$:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + ...\right)$$

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + ...\right)$$

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

$$\hat{A}_t^{\text{GAE}(\gamma,0)} = \delta_t^V = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{A}_t^{\text{GAE}(\gamma,1)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V$$

This allows for a more accurate estimation of the Advantage over time.

Clipping policy gradient

Actor loss:

$$\mathcal{L}_a(\theta) = \mathbb{E}_{s,a}\left[-\log(\pi_\theta(s,a))A(s,a)\right]$$

Clipped actor loss:

$$\mathcal{L}_{\text{clip}}(\theta) = \mathbb{E}_{s,a}\left[-\log(\pi_\theta(s,a))\min(\tau_t(\theta)\hat{A}_t, \text{clip}(\tau_t(\theta), 1-\epsilon, 1+\epsilon))\right]$$

With $\tau_t(\theta)$ denoting the probability ratio:

$$\tau_t(\theta) = \frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{\text{old}}}(a_t, s_t)}$$

Proximal Policy Optimization (PPO) model

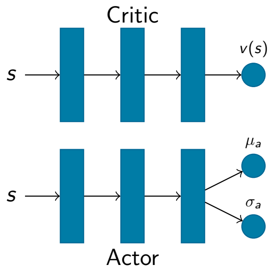


Figure 36: Neural network architecture of the PPO method[8].

- $\mathcal{L}(\theta) = \mathcal{L}_{\text{clip}}(\theta) + c_c\mathcal{L}_c(\theta) - c_H\mathcal{L}_H(\theta)$
- Generalized Advantage estimator GAE(λ)

[8] Schulman et al., “Proximal Policy Optimization Algorithms”.

PPO success: Dota

Figure 37: AI players from the OpenAI Five[9] in Dota against humans. (image from the reference)

- Uses 128 000 CPUs and 256 GPUs, doing 900 years of gameplay/day for 10 months.
- Uses LSTM based recurrent neural networks.
- Given a feature map of the visible area, and not an image.
- simplified gameplay (no wards or summons).