Introduction to Deep Reinforcement Learning

IA Frameworks

Brendan Guillouet - 14/12/2020

Institut National des Sciences Appliquées de Toulouse

Tools

ML Python Libraries

scikit learn
Keras
gensim
surprise
Gym
TensorFlow
NLTK

Python Environment

PC
jupyter
ANACONDA

Viz' Python Libraries

matplotlib
Version 3.1.1
seaborn
plotly

Framework & Tool

docker
Google Cloud Platform
git

Table of contents

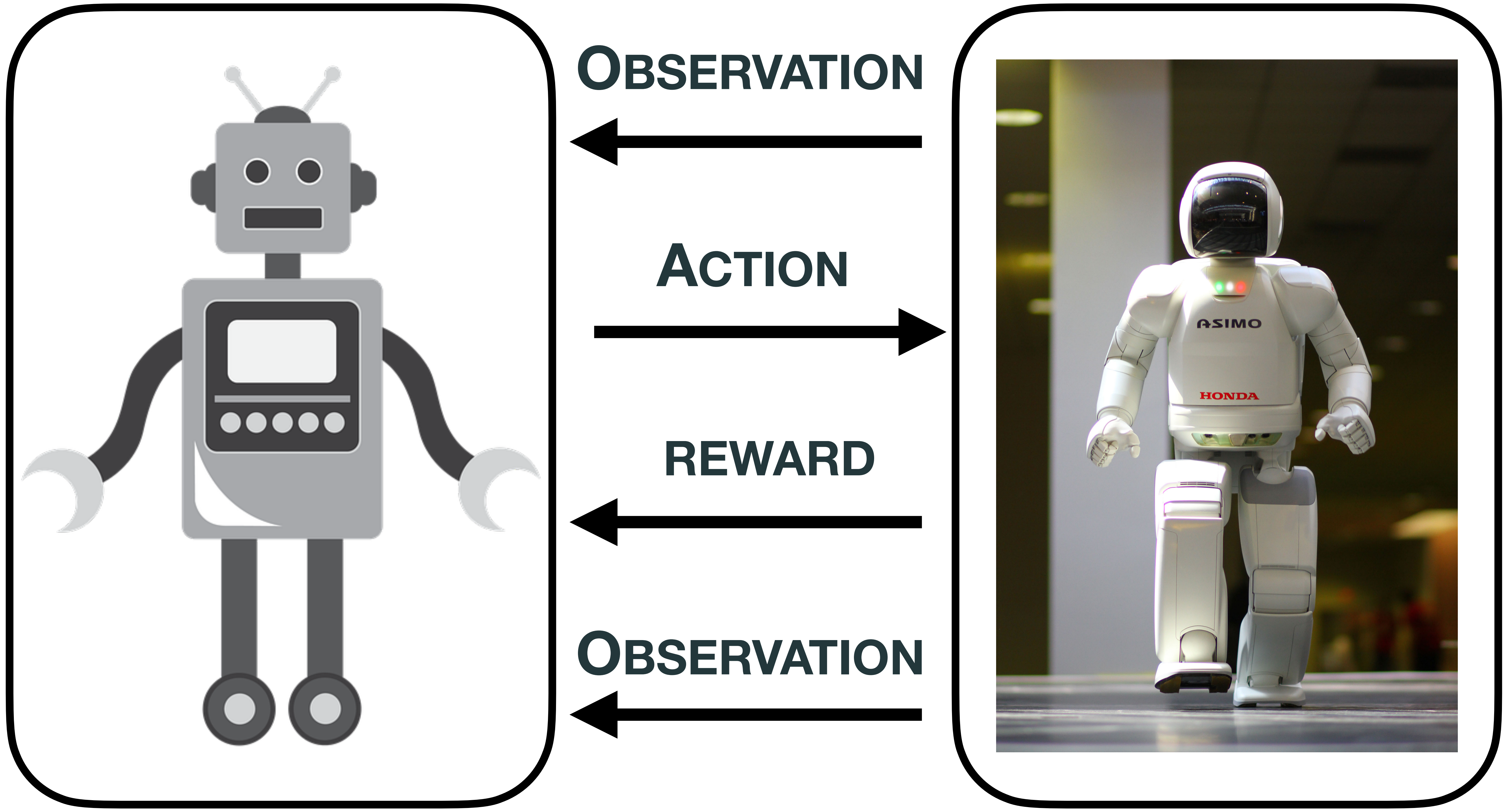
Introduction

Definitions

In reinforcement learning, an agent makes **OBSERVATIONS** of the **STATES** of an **ENVIRONMENT**.

It takes **ACTION** within the **ENVIRONMENT** and receives **REWARDS**.

Example: Walking Robot



OBSERVATION: Measurement from various sensor (time series).

ACTION: Move members (3D).

REWARD: Positive when it approach the destination. Negative otherwise.

Definitions

When using reinforcement learning you need to define:

An **Agent**. The one who will take action within the environment.

An **Environment** where the action will be taken.

The **State** of the environment defines it after each action.

The **Observation** is what we will used from the state to decide the next action.

A list of possible **Actions** actions that will affect the environment and produce a new state.

A **Reward**. A numerical value which reveal how positive or negative the action is.

You can't apply reinforcement learning if you're not able to define these objects properly!

Objective

Maximise the long-term rewards.

Do not pull all the effort on capturing the queen if it means losing all you pieces.

How ? There are two mains approaches**:**

Value-Based. Look for the optimal reward.

- *Learn to estimate the expected rewards for each action in each state.*
- *Use this knowledge to choose the best action.*

Policy-Based. Look for the optimal **policy**.

- *Learn directly the best action to take for each observation.*

NB: Methods like Actor-Critic try to optimise both policy and rewards.

OBJECTIVE

Maximise the long-term rewards.

Do not pull all the effort on capturing the queen if it means losing all you pieces.

How ? There are two mains approaches**:**

VALUE-BASED. Look for the optimal reward.

- *Learn to estimate the expected rewards for each action in each state.*
- *Use this knowledge to choose the best action.*

POLICY-BASED. Look for the optimal policy.

- ***Learn directly the best action to take for each observation.***

NB: Methods like Actor-Critic try to optimise both policy and rewards.

Definitions

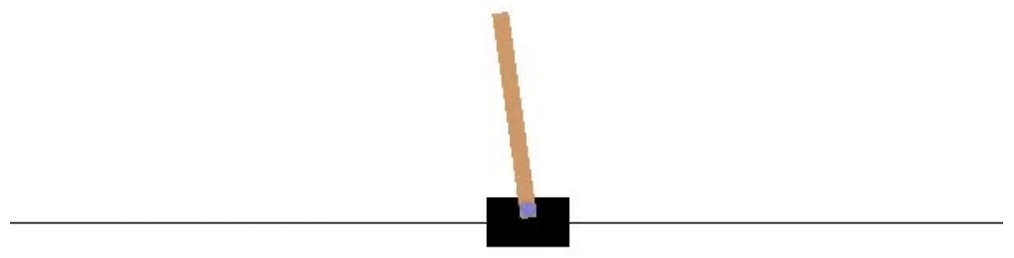
Policy

The algorithm used by the ages to determine its action.

$$\Pi(s) = argmax_a \ p(a/s)$$

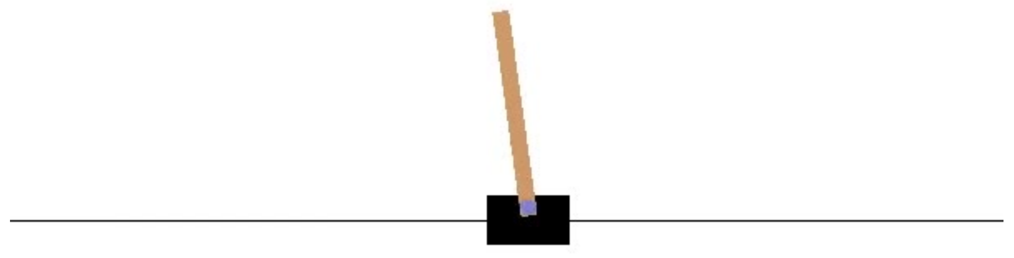
It can be whatever you want :

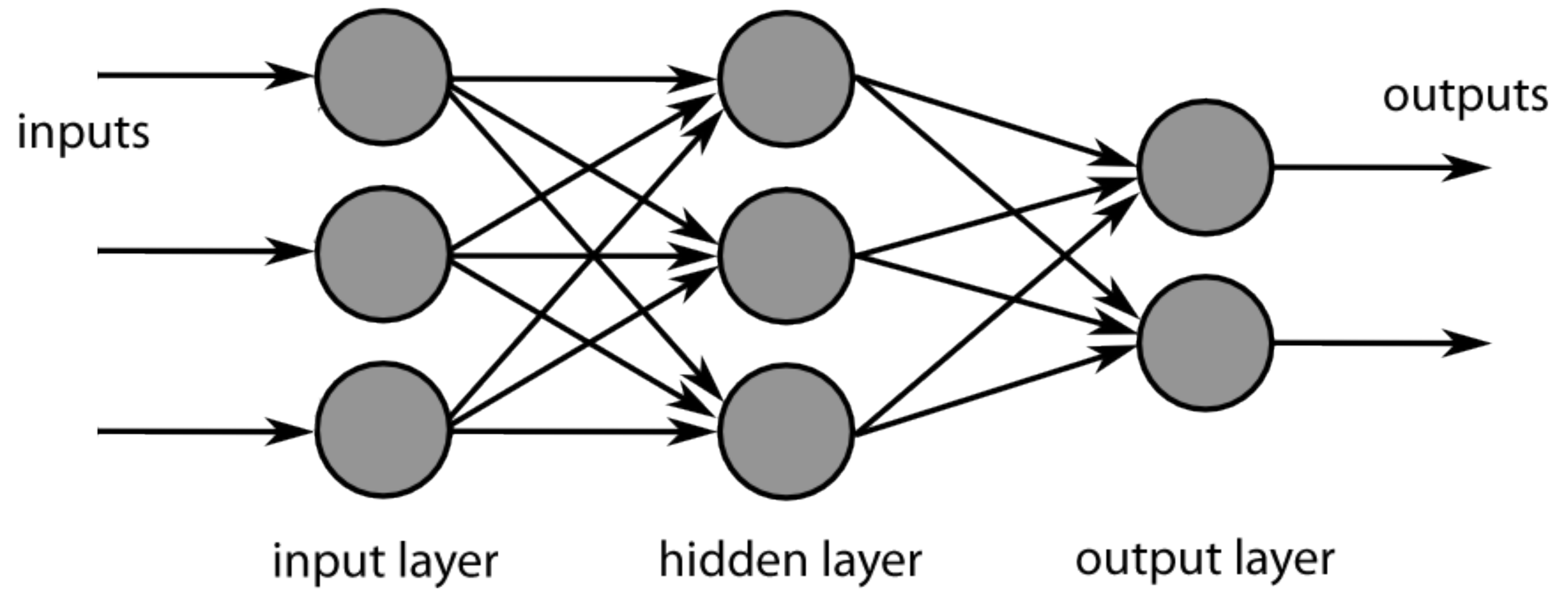
Rules. *Example: Apply force to the right if the pole it tilting to left else apply force to the left.*



Angle < 0° → *Push to the right.*

PARAMETRIC FUNCTION





Push to the right.

Policy Search

How do you train the policy ?

Random search.

Does not work when the space is too big, which is often the case.

Genetic algorithm

1. *Try a set of N policies.*
2. *Keep the n best policies*
3. *Generate N new policies that are random deviation of these n policies.*
4. *Iterate*

Policy gradient

1. *Play the game with regards to the policy parameters.*
2. *Update these parameters with gradient descent.*

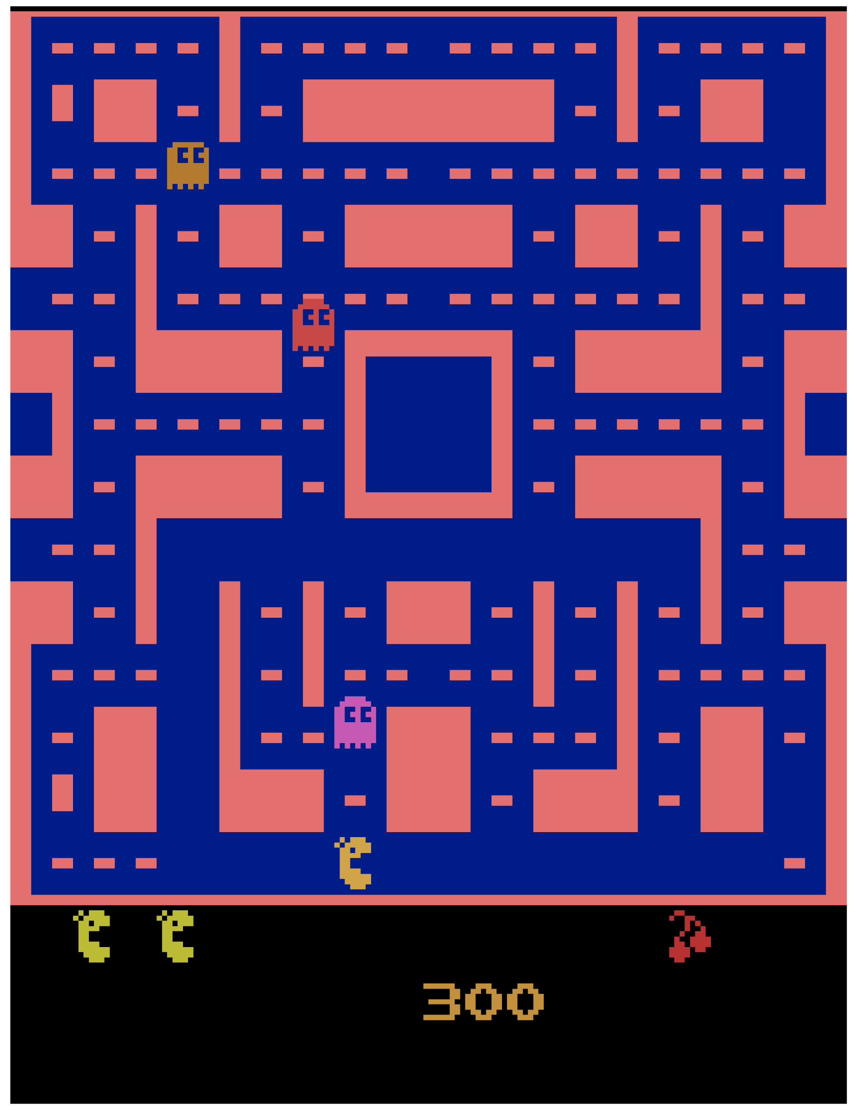
The credit Assignment Problem

What is the difference with **(Deep) Q-Learning** algorithm?

Do not evaluate the Q-value.

Predict the policy and the action to take **directly**.

How to choose the loss on which to train the policy ?



Proposition: the immediate reward ?

Problem: we do not know the influence on **the long-term reward** of an action.

Example: Going up is obviously not the best action.

Solution: DISCOUNTED CUMULATIVE EXPECTED REWARD

The Discounted Cumulative Expected Reward

Evaluate an action based on the sum of all the rewards that come after it.

$$R_t = \sum_{i=t}^{T} \gamma^i r_i$$

where γ is the discounted rate and r_t is the reward as step t.

PacMan Example *The agent decides to go up three times in a row. It gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step (by being killed).*

With $\gamma = 0.8$

Step	Immediate reward	Discounted cumulated expected reward
0	10	$R_0 = 10 + 0 \times 0.8 + (-50) \times 0.8^2 = -22$
1	0	$R_1 = 10 + (-50) \times 0.8 = -40$
2	-50	$R_2 = -50$

Policy Gradient Algorithm

Policy Gradient

Objective: Find a **policy** Π_θ, with parameters θ that maximises the expected values of the sum of the discounted rewards.

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

- The environment are usually non-deterministic.
- θ are the parameters of the neural network.
- Let's perform a gradient ascent search to learn the optimal θ.

$$\theta \leftarrow \theta + \alpha \nabla J(\theta)$$

- α is the learning rate.

Gradient search

How to find $\nabla J(\theta)$? with:

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right] \\
&= \mathbb{E}_{\pi_\theta}[R(\tau)] \\
&= \sum_{\tau} P(\tau) R(\tau)
\end{aligned}
$$

Where is the trajectory of the agent moving through the environment

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$

Log derivative trick

Using the *log-derivative* trick:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_\tau \nabla_\theta P(\tau) R(\tau) \\
&= \sum_\tau P(\tau) \frac{\nabla_\theta P(\tau)}{P(\tau)} R(\tau) \\
&= \sum_\tau P(\tau) \nabla_\theta log P(\tau) R(\tau) \\
&= \mathbb{E}[\nabla_\theta log P(\tau) R(\tau)]
\end{aligned}$$

During training, trajectories are randomly sampling.
To maximise the expectation above, we need to maximise it with respect to its argument i.e. we maximise:

$$\nabla_\theta J(\theta) \sim R(\tau) \nabla_\theta log P(\tau)$$

Gradient search

$$P(\tau) = \prod_{t=0}^{T-1} P_{\pi_\theta}(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t)$$

Transition probabilities

Probability to choose an action at a given state according to the policy (Output of softmax)

$$\begin{aligned}
\nabla_\theta \log P(\tau) &= \nabla \log \left(\prod_{t=0}^{T-1} P_{\pi_\theta}(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t) \right) \\
&= \nabla_\theta \left[\sum_{t=0}^{T-1} \left(\log P_{\pi_\theta}(a_t \mid s_t) + \log P(s_{t+1} \mid s_t, a_t) \right) \right] \\
&= \nabla_\theta \sum_{t=0}^{T-1} \log P_{\pi_\theta}(a_t \mid s_t)
\end{aligned}$$

Gradient search

$$\nabla_\theta J(\theta) \sim R(\tau)\, \nabla_\theta \sum_{t=0}^{T-1} logP(\tau)$$

$$\sim R(\tau)\, \nabla_\theta \sum_{t=0}^{T-1} \log P_{\pi_\theta}(a_t \,|\, s_t)$$

This is a weighted cross entropy where the weight are the discounted reward !

$$CE = -\sum p(x) log\big(q(x)\big)$$

Hence by training a neural network using weighted cross entropy with discounted reward as the weights

You are training a PG algorithm.

Keras - Discounted Loss

```
import tensorflow.keras.losses as klo
class discountedLoss(klo.Loss):
    """
    Args:
      pos_weight: Scalar to affect the positive labels of the loss function.
      weight: Scalar to affect the entirety of the loss function.
      from_logits: Whether to compute loss from logits or the probability.
      reduction: Type of tf.keras.losses.Reduction to apply to loss.
      name: Name of the loss function.
    """

    def __init__(self,
                 reduction=klo.Reduction.AUTO,
                 name='discountedLoss'):
        super().__init__(reduction=reduction, name=name)

    def call(self, y_true, y_pred, adv):
        log_lik = - (y_true * K.log(y_pred) + (1 - y_true) * K.log(1 - y_pred))
        loss = K.mean(log_lik * adv, keepdims=True)
        return loss
```

KERAS - CUSTOM MODEL CLASS

```python
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl
import tensorflow.keras.initializers as ki
import tensorflow.keras.metrics as kme

class kerasModel(km.Model):
    def __init__(self):
        super(kerasModel, self).__init__()
        self.layersList = []
        self.layersList.append(kl.Dense(9, activation="relu",
                    input_shape=(4,),
                    use_bias=False,
                    kernel_initializer=ki.VarianceScaling(),
                    name="dense_1"))
        self.layersList.append(kl.Dense(1,
                      activation="sigmoid",
                      kernel_initializer=ki.VarianceScaling(),
                      use_bias=False,
                      name="out"))

        self.loss = discountedLoss()
        self.optimizer = ko.Adam(lr=1e-2)
        self.train_loss = kme.Mean(name='train_loss')
        self.validation_loss = kme.Mean(name='val_loss')
        self.metric = kme.Accuracy(name="accuracy")

        @tf.function()
        def predict(x):
            """
            This is where we run
            through our whole dataset and return it, when training and testing.
            """
            for l in self.layersList:
                x = l(x)
            return x
        self.predict = predict

        @tf.function()
        def train_step(x, labels, adv):
            """
                This is a TensorFlow function, run once for each epoch for the
                whole input. We move forward first, then calculate gradients with
                Gradient Tape to move backwards.
            """
            with tf.GradientTape() as tape:
                predictions = self.predict(x)
                loss = self.loss.call(
                    y_true=labels,
                    y_pred = predictions,
                    adv = adv)
            gradients = tape.gradient(loss, self.trainable_variables)
            self.optimizer.apply_gradients(zip(gradients, self.trainable_variables))
            self.train_loss(loss)
            return loss

        self.train_step = train_step
```

Define a **KERAS MODEL** class.

optimiser, *train_loss*, *validation_loss*, *metric* are native keras function.

loss is the custom loss we defined before.

predict and *train_step* are redefine according to our needs

REINFORCE Algorithm (R.williams-1992)

1. Initiate the policy randomly.

2. Let the policy play the game several times. and at each step compute the gradients that would make the chosen action even, don’t apply these gradients yet.

3. Compute each action’s discounted cumulative expected reward

4. Multiply each gradient vector by the corresponding action’s score.

5. Compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Difference with Q-learning

In (deep) Q-learning, the parameters we are trying to find are those that minimise the difference between the actual Q values (drawn from experiences) and the target.

No explicit exploration.

Converge faster, but need more and complete episode.

Still better with memory replay.

No target network needed

Exploration vs Exploitation

Exploitation mode : Pick the best action according to the policy.

Problem : Being stuck in a non-optimal solution.

Exploration mode : Takes random action to explore the space.

Problem : Can take a lot of time.

E-greedy strategy

Take the best action (1-ε)% of the time

Take a random action ε% of the time.

Stochastic strategy : Use the probability to take an action to choose to act randomly or not.

action = 0 if random.uniform(0, 1) < predict_proba.

action = 1 otherwise.

Pseudo code

- Initiate the model to train *P*.
- While *num_episode* < *max_number_episode* OR *test_score* < goal
  - Play episode(s) **entirely** :
    - Choose action according to a strategy.
    - Save experiences.
  - If *len(*experiences*)* > *batch_size*
    - *Normalise* all discounted rewards
    - Train the model *P* with
      - X = States
      - y = actions
      - Loss = discounted_loss

Experiences = [state, action, discounted_reward]

Discounted reward are computed from complete episode (Monte-Carlo method).

Baseline

The discounted rewards are normalised before each batch.

$$\nabla_\theta J(\theta) \sim \mathbb{E}[(R(\tau) - N)\,\nabla_\theta log P(\tau)]$$

We subtract a baseline. *It reduce the variance of gradient estimation while keeping the bias unchanged.*

A common baseline is to subtract state-value.

$$\nabla_\theta J(\theta) \sim \mathbb{E}[(R(\tau) - V(s))\,\nabla_\theta log P(\tau)]$$

And because $Q^\pi(s,a) = \mathbb{E}\left(\lim_{H\to\infty}\sum_{t=0}^{H}\gamma^t r_t \,\middle|\, s_0 = s, a_0 = a, \pi\right)$

$$\nabla_\theta J(\theta) \sim \mathbb{E}[(Q(s,a) - V(s))\,\nabla_\theta log P(\tau)]$$

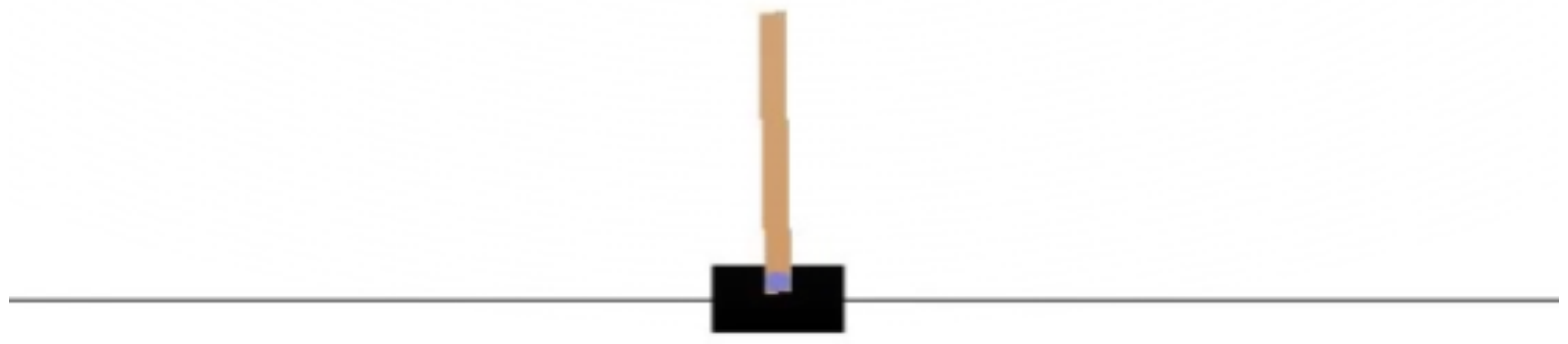
Actor-Critic

- Initiate the policy model P_θ and the Q-value model Q_ω .
- While *num_episode* < *max_number_episode* OR *test_score* < goal
  - Play a step of an episode :
    - Choose action according to a strategy (use P in exploit mode).
    - Save experiences
  - Train the policy model
    - $\theta \leftarrow \theta + \alpha \nabla Q(s,a) \log P_{\pi_\theta}(a_t \mid s_t)$
  - Train the Q-value model
    - Compute target from experiences
    - $\omega \leftarrow \omega - \alpha \nabla_\omega \mathbb{E}_{s \sim' P(s'|s,a)}[(Q_\omega(s,a) - target(s'))^2]$

TP

One Notebook *: Policy Gradient.ipynb*

- Implement hard coded policy
- Train a neural network model to learn a given policy.
- Train a neural model with a Policy Gradient algorithm.
    - Implement discounted reward function.
    - Implement loss using **keras.**
    - Write PG algorithm.



Blog & Code

https://adventuresinmachinelearning.com/policy-gradient-tensorflow-2/

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63#:~:text=REINFORCE%20is%20a%20Monte%2DCarlo,to%20update%20the%20policy%20parameter.&text=Store%20log%20probabilities%20(of%20policy)%20and%20reward%20values%20at%20each%20step

https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

References

Géron, A. (2017). Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts. *Tools, and Techniques to build intelligent systems*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3-4), 229-256.