

# Reinforcement Learning

Prof. Gheith Abandah

Reference: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by Aurélien Géron (O'Reilly), 2017, 978-1-491-96229-9.

# Introduction

- YouTube Video: *An introduction to Reinforcement Learning* from Arxiv Insights

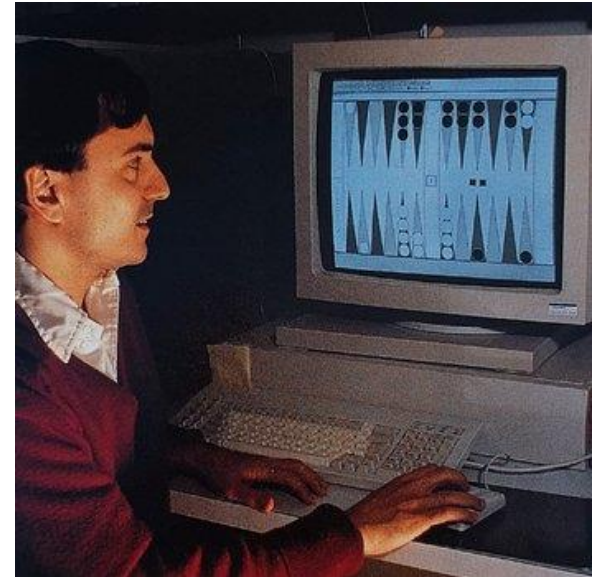
<https://youtu.be/JgvyzIkgxF0>

# Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policy
5. The Credit Assignment Problem
6. Policy Gradient Algorithm
7. Exercises

# 1. Introduction – History

- RL started in **1950s**
- **1992**: IBM's TD-Gammon, a Backgammon playing program.
- **2013**: DeepMind demonstrated a system that learns to play Atari games from scratch.
- Use **deep learning** with raw pixels as inputs and without any prior knowledge of the rules of the games.
- **2014**: Google bought DeepMind for \$500M.
- **2016**: AlphaGo beats Lee Sedol.



# 1. Introduction – Definition

- In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*.
- Its objective is to learn to act in a way that will maximize its expected long-term rewards.
- In short, the agent acts in the environment and learns by trial and error to maximize its *pleasure* and minimize its *pain*.

# 1. Introduction – Examples

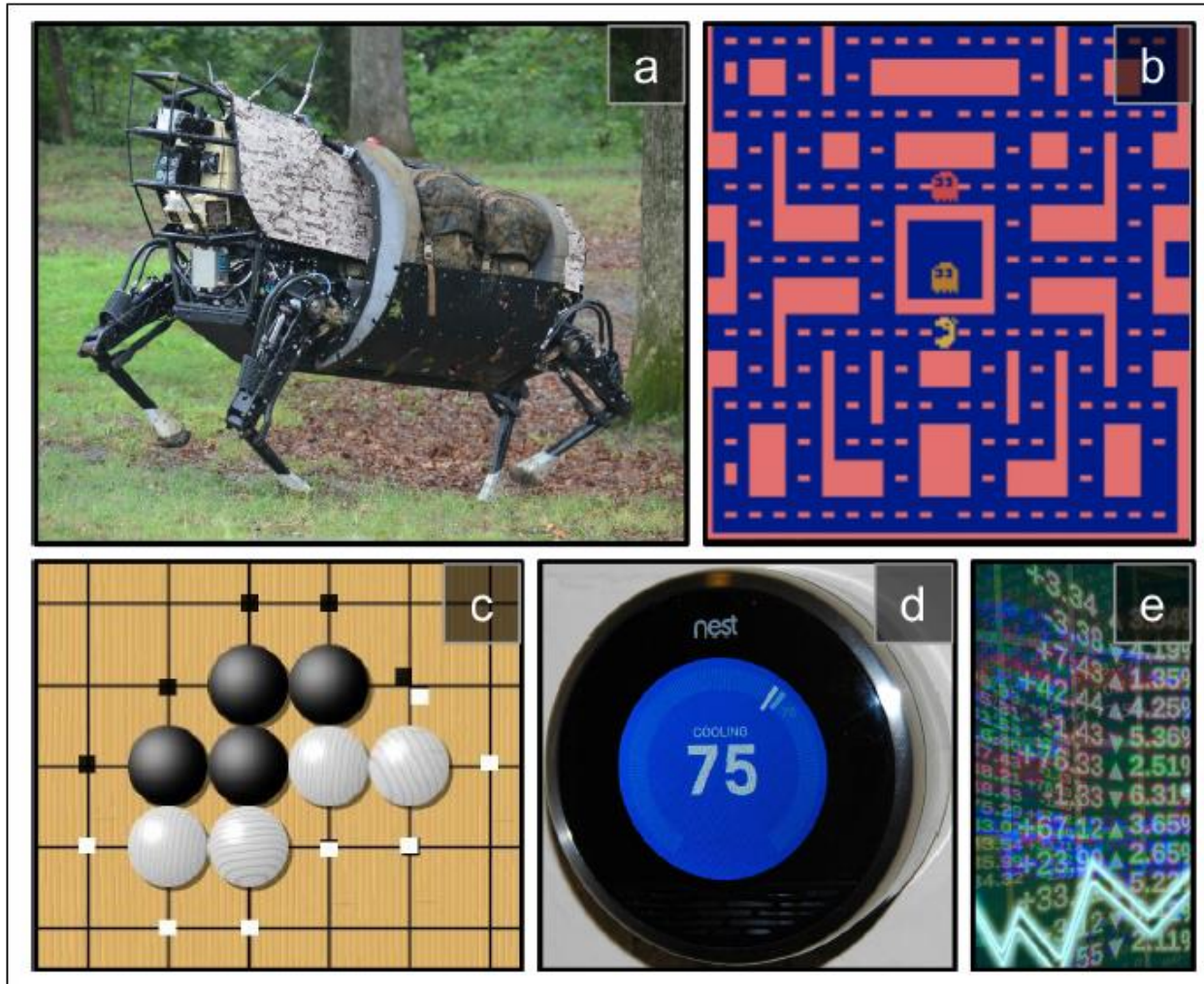
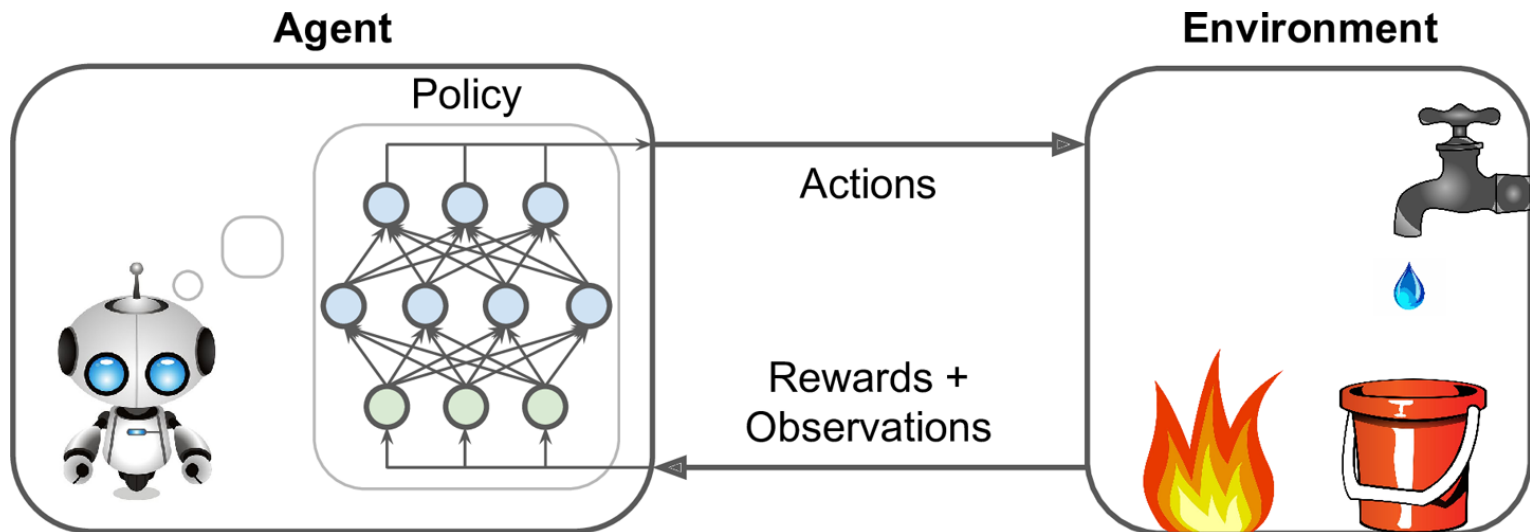


Figure 16-1. Reinforcement Learning examples: (a) walking robot, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader<sup>5</sup>

## 2. Policy Search

- The algorithm used by the software agent to determine its actions is called its *policy*.
- The policy can be *deterministic* or *stochastic*.
- **Policy search techniques:** Brute force, Genetic algorithm, Policy Gradient (PG), Temporal Difference (TD) Learning, Q-Learning.



# 3. OpenAI Gym

- OpenAI gym is a toolkit that provides *simulated environments* (Atari games, board games, 2D and 3D physical simulations, ...).
- OpenAI is a nonprofit AI research company funded in part by Elon Musk.

```
$ pip3 install --upgrade gym
```

```
>>> import gym
```

```
>>> env = gym.make("CartPole-v0")
```

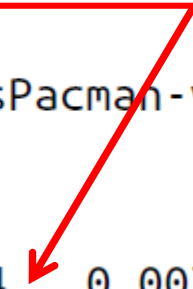
```
[2016-10-14 16:03:23,199] Making new env: MsPacman-v0
```

```
>>> obs = env.reset()
```

```
>>> obs
```

```
array([-0.03799846, -0.03288115, 0.02337094, 0.00720711])
```

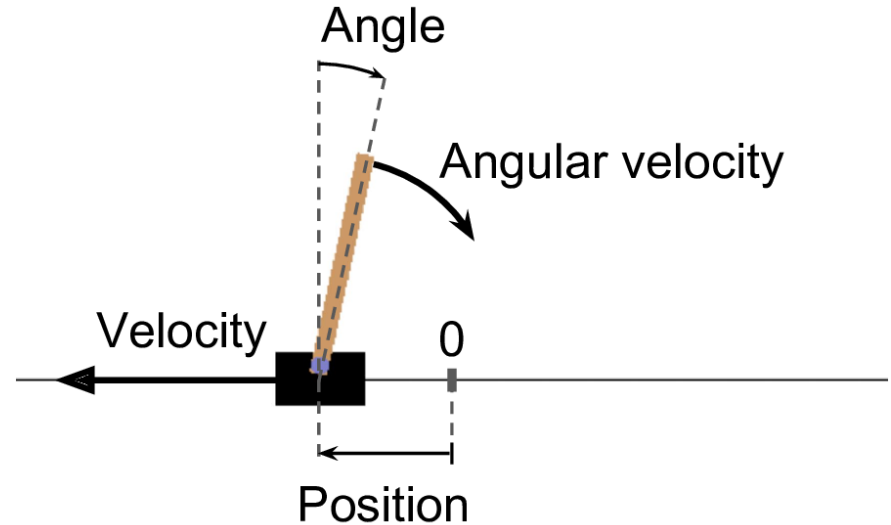
Cart position, cart speed, pole angle,  
pole velocity





# 3. OpenAI Gym

```
>>> env.render()
```



- *render()* can also return the rendered image as a NumPy array.

```
>>> img = env.render(mode="rgb_array")  
>>> img.shape # height, width, channels (3=RGB)  
(400, 600, 3)
```

# 3. OpenAI Gym – Balancing the pole

```
>>> env.action_space  
Discrete(2)
```

The possible actions are integers 0 and 1, which represent accelerating left (0) or right (1).

```
>>> action = 1 # accelerate right  
>>> obs, reward, done, info = env.step(action)  
>>> obs  
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])  
>>> reward  
1.0  
>>> done  
False  
>>> info  
{}
```

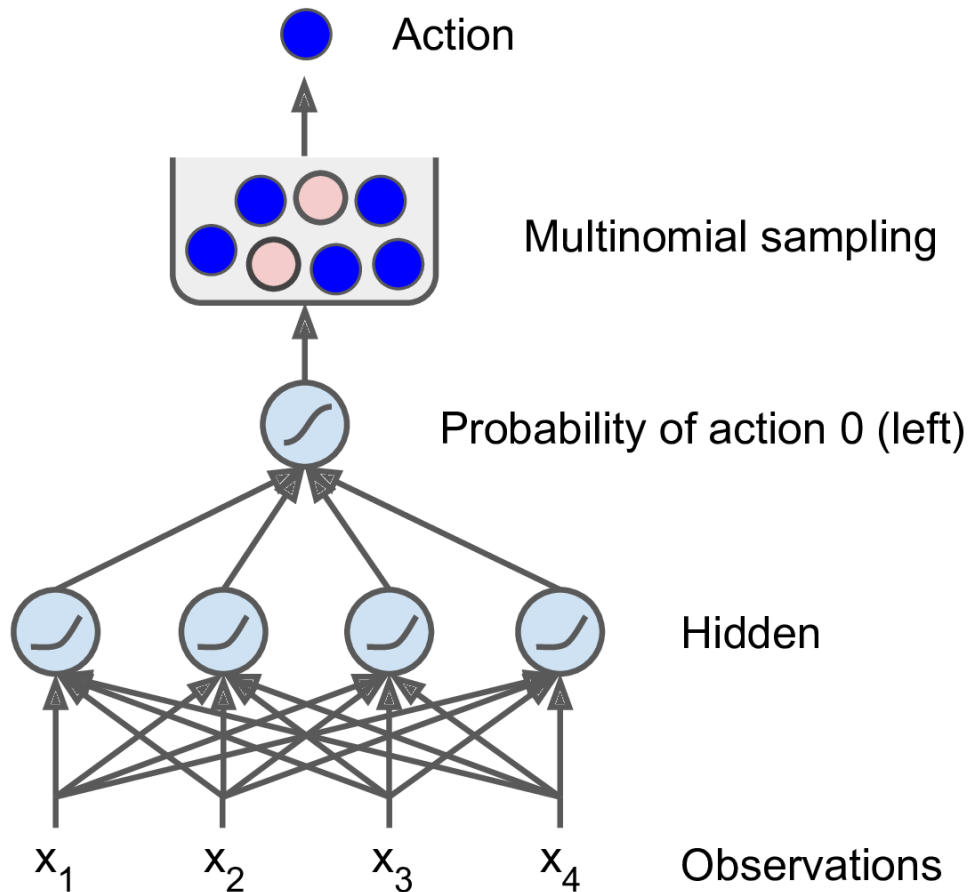
# 3. OpenAI Gym – Balancing the pole

```
def basic_policy(obs):  
    angle = obs[2]  
    return 0 if angle < 0 else 1  
  
totals = []  
for episode in range(500):  
    episode_rewards = 0  
    obs = env.reset()  
    for step in range(1000): # 1000 steps max, we don't want to run forever  
        action = basic_policy(obs)  
        obs, reward, done, info = env.step(action)  
        episode_rewards += reward  
        if done:  
            break  
    totals.append(episode_rewards)
```

Accelerates left when the pole is leaning left and accelerates right when the pole is leaning right.

```
>>> import numpy as np  
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)  
(42.125999999999998, 9.1237121830974033, 24.0, 68.0)
```

# 4. Neural Network Policy



Picking a random action based on the probability given by the neural network lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well.

# 4. Neural Network Policy

1. Define the neural network architecture. The number of inputs is the size of the observation space, four hidden units, and one output probability (the probability of going left).

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

# 1. Specify the neural network architecture
n_inputs = 4 # == env.observation_space.shape[0]
n_hidden = 4 # it's a simple task, we don't need more hidden neurons
n_outputs = 1 # only outputs the probability of accelerating left
initializer = tf.contrib.layers.variance_scaling_initializer()
```

# 4. Neural Network Policy

2. Build a Multi-Layer Perceptron. The output layer uses the logistic (*sigmoid*) activation function in order to output a probability from 0.0 to 1.0. If there were more than two possible actions, there would be one output neuron per action, and use the *softmax* activation function instead.

*# 2. Build the neural network*

```
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
                          weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,
                          weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
```

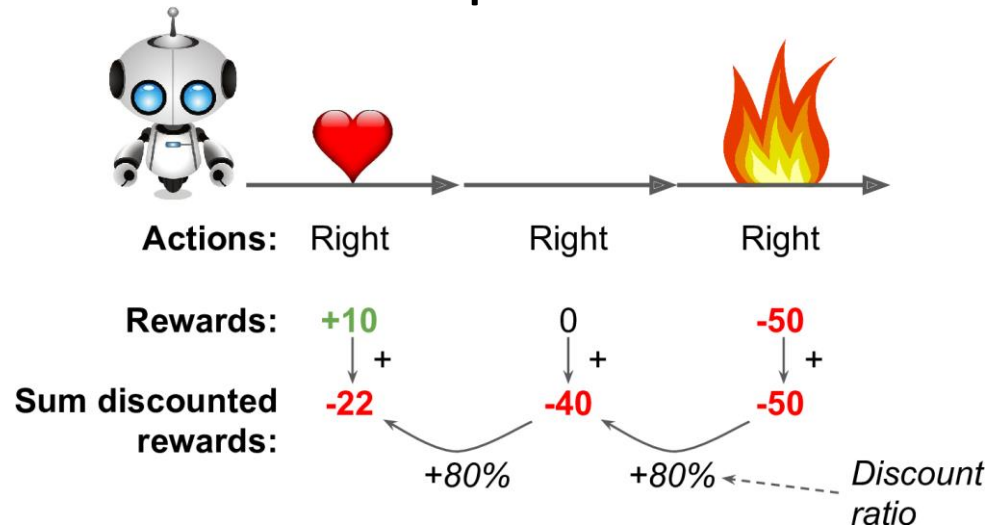
## 4. Neural Network Policy

3. The `multinomial()` function picks one random action: `outputs` probability of being 0, `1 - outputs` of being 1.

```
# 3. Select a random action based on the estimated probabilities  
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])  
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)  
  
init = tf.global_variables_initializer()
```

# 5. The Credit Assignment Problem

- Rewards are typically *sparse* and *delayed*.
- *Credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it.
- Evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount rate  $r$*  at each step.





# 6. Policy Gradient Algorithm

- Optimize the parameters of a policy by following the gradients toward higher rewards.
  1. Let the neural network policy play the game several times and at each step compute the gradients that would make the chosen action even more likely.
  2. Every several episodes, compute each action's score (using the credit assignment method).
  3. If an action's score is positive, it means that the action was good and you want to apply the gradients to make the action more likely in the future. However, if the score is negative, it means the action was bad and you want to apply the opposite gradients. The solution is simply to multiply each gradient vector by the corresponding action's score.
  4. Compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

# Summary

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policy
5. The Credit Assignment Problem
6. Policy Gradient Algorithm
7. Exercises

# Exercises

From Chapter 16, solve exercises:

- 1
- 2
- 5