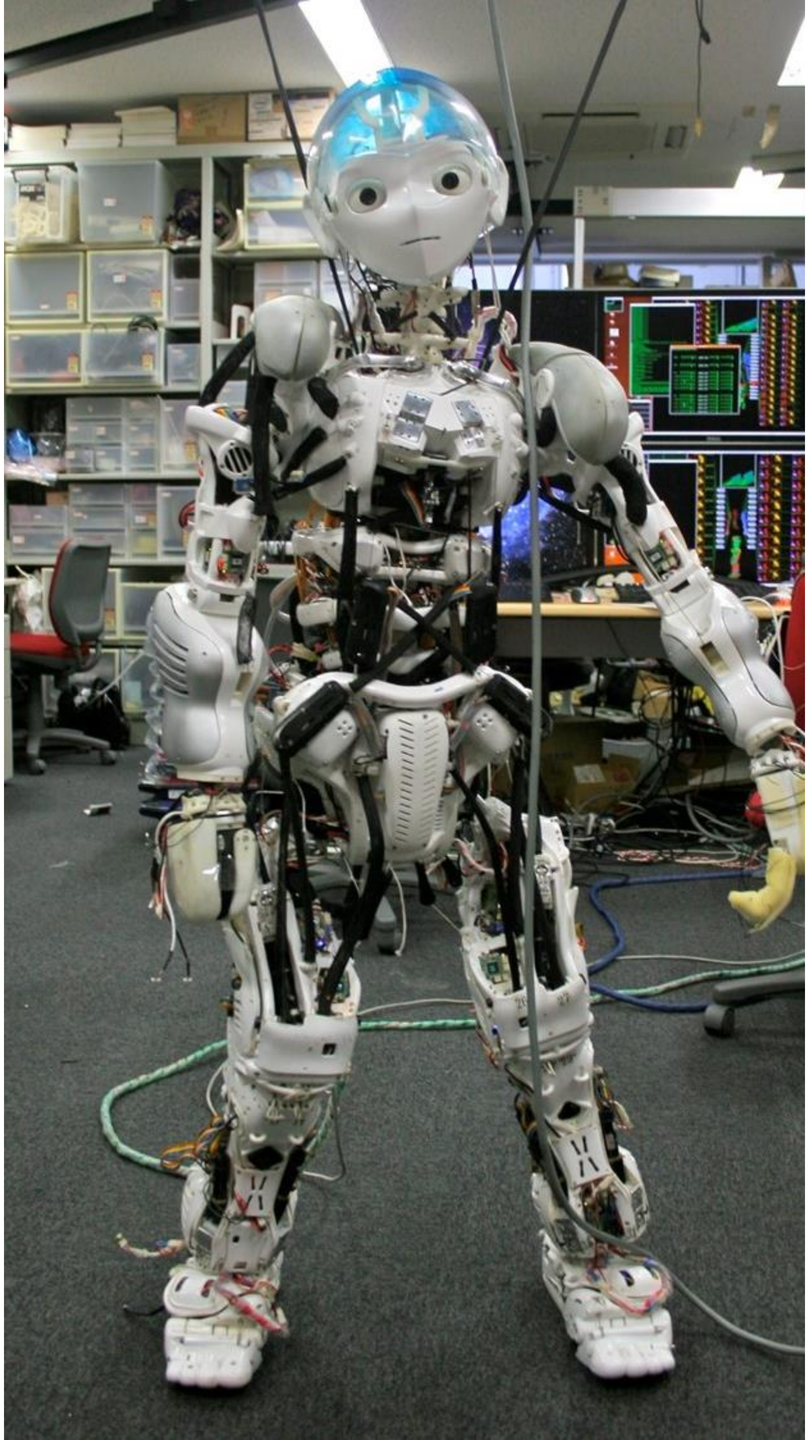
CS6482 Deep RL

1: Policy Gradient Approach

J.J. Collins

Dept. of CSIS

University of Limerick



Objectives

- Policy Gradient applied to Cartpole
- REINFORCE algorithm

Outline

Summary of

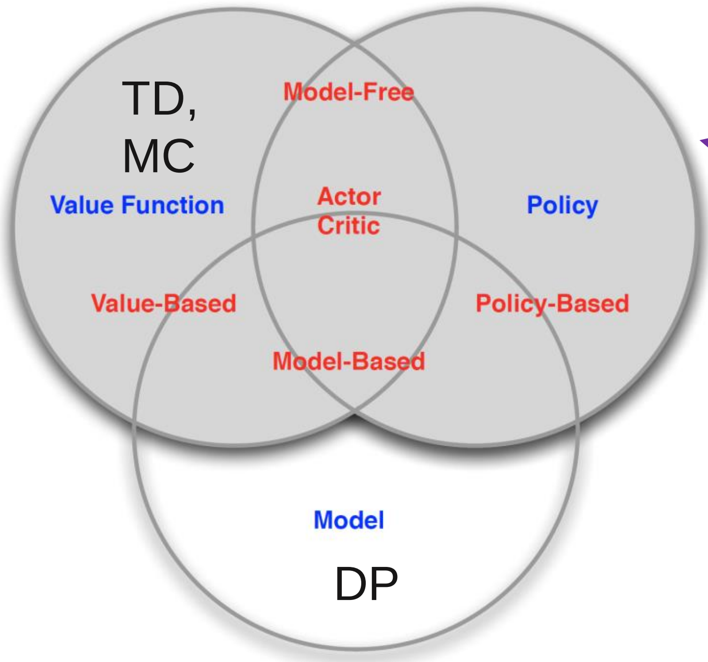
Chapter 18 in Gueron. Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2nd Edition, O'Reilly. 2019.

Chapter 13 in Sutton and Barto. Reinforcement Learning: an Introduction, 2nd Edition. The MIT Press. 2018.

Slides from Guni Sharon https://people.engr.tamu.edu/guni/csce689/index.html

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

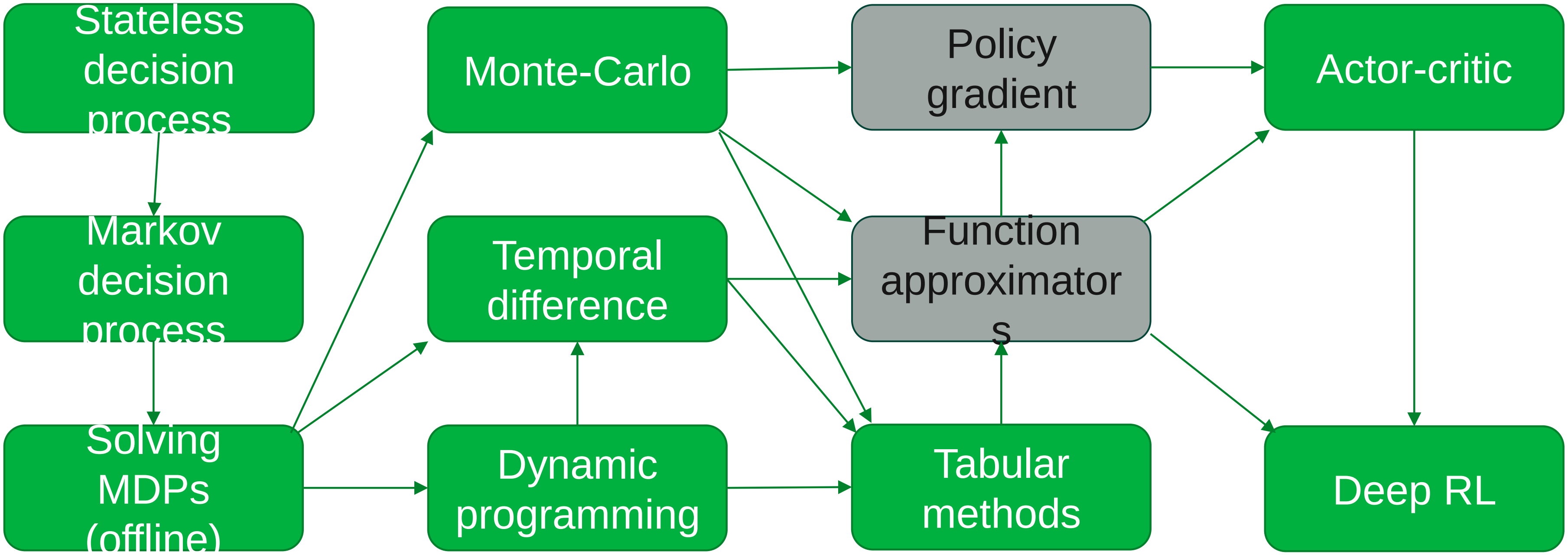
A taxonomy of RL solutions



Focus of today!

Figure credit: David Silver, "Introduction to RL"

Reinforcement Learning



Policy Gradient Approaches

https://www.trustedreviews.com/news/roborock-launches-saros-z70-the-robot-vacuum-cleaner-that-can-pick-up-socks-4580544

Robotic Vacuum Cleaner

Rewards is weight of dust collected in a 30 min run

Policy

Move forward with probability p

Randomly rotate left or right with probability 1-p

Rotation angle {-r,+r}

Two policy parameters p and r

How could you train the robot?

Policy Gradient Approaches

- Search the policy space
- Use brute search, Genetic Algorithms, Gradient Descent , etc.

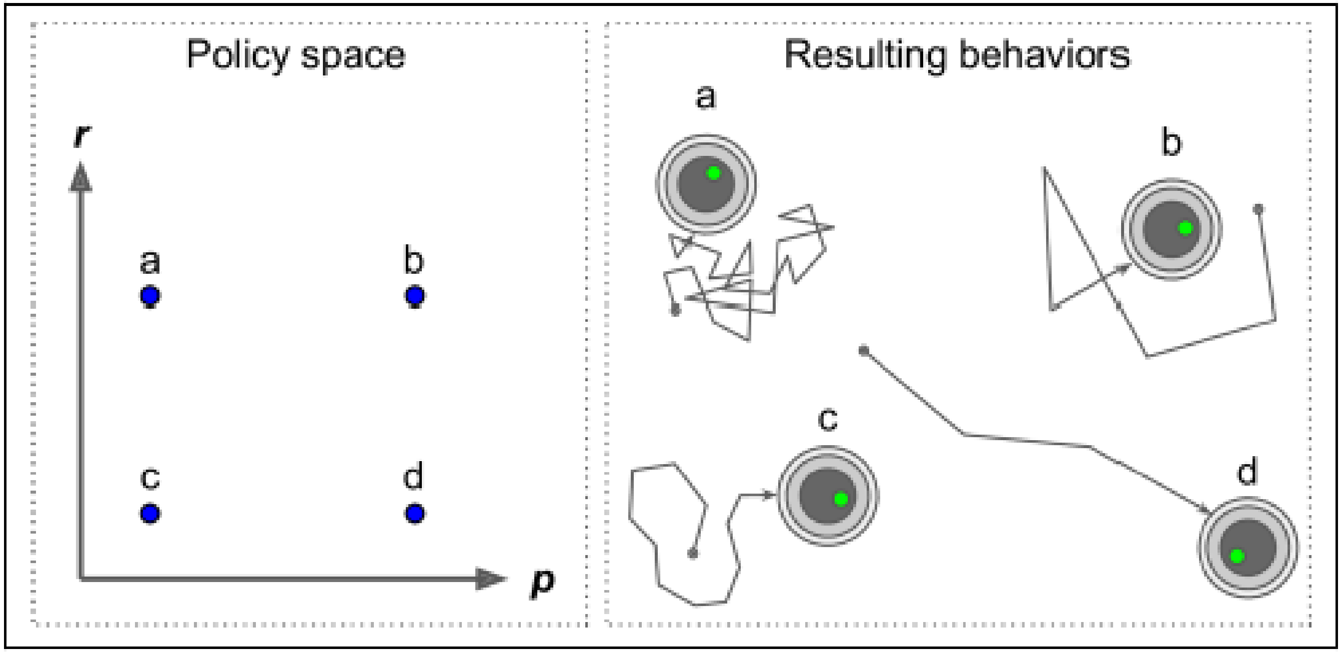


Figure. Geuron©

Reinforce Algorithm [Williams 1992]

1. **Play the game several times and compute the gradient at each step**
   - Gradient that would make the chosen action more likely
2. **Compute Action Advantage**
   - How does an action compare to other actions? → Action Advantage
   - Run many episodes, and normalise each action's returns
   - By subtracting from the mean and dividing by standard deviation
3. **Multiply each action's Action Advantage by its gradient**
   - If an action's Action Advantage is positive apply gradient
   - Else apply negative gradient
4. **Compute mean of all resulting gradient vectors and use this to perform a Gradient Descent step**

Reinforce Algorithm [Williams 1992]

- Call the model giving it a single observation (reshaped as a batch with a single instance), and outputs the prob of going left.
- If left_proba is high then action will most likely be False since a random number sampled between 0 and 1 will probably not be greater.
- And False means 0 when you cast it to a number so y_target would be 1-0=1, meaning that we predict that the probability of going left is 100%.

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```


UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

Reinforce Algorithm [Williams 1992]

- Plays multiple episodes returning all rewards and gradients

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

Reinforce Algorithm [Williams 1992]

- The Algorithm plays the game several times and then discount and normalise rewards

- calling discount_rewards([10,0,-50]) will return [-22, -40, -50]

```
def discount_rewards(rewards, discount_rate):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_rate
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards, discount_rate)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Reinforce Algorithm [Williams 1992]

- Hyperprameters

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_rate = 0.95
```

```
optimizer = keras.optimizers.Adam(learning_rate=0.01)
loss_fn = keras.losses.binary_crossentropy
```

```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[4]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

Reinforce Algorithm [Williams 1992]

- Optimiser and Loss Function

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_rate = 0.95
```

```
optimizer = keras.optimizers.Adam(learning_rate=0.01)
loss_fn = keras.losses.binary_crossentropy
```

```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[4]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

Reinforce Algorithm [Williams 1992]

Training Loop

- Call play_multiple_episodes()
- Plays the game 10 times and returns rewards and gradients for each episode
- Calculate each action's Action Advantage – the final reward, using discount_and_normalise_rewards()
- Compute mean of the gradient for each weight multiplied by final reward
- Apply these man gradients

```
env = gym.make("CartPole-v1")
env.seed(42);

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    total_rewards = sum(map(sum, all_rewards))                     # Not shown in the book
    print("\rIteration: {}, mean rewards: {:.1f}".format(          # Not shown
        iteration, total_rewards / n_episodes_per_update), end="") # Not shown
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_rate)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Criticism of this PG Algorithm

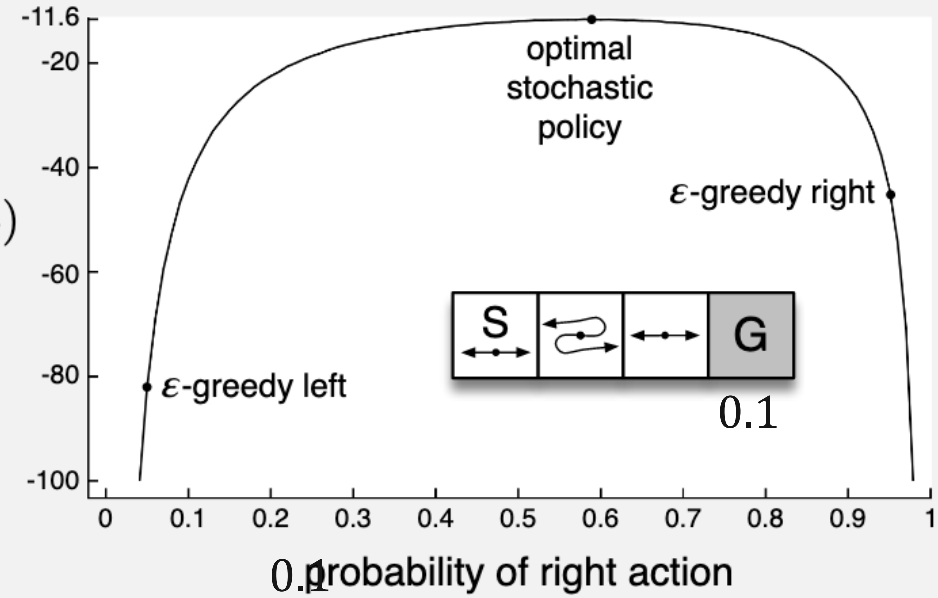
The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that it must run multiple episodes to estimate the advantage of each action, as we have seen. However, it is the foundation of more powerful algorithms, such as *actor-critic* algorithms (which we will discuss briefly at the end of this chapter).

Geron, 2023

Stochastic Policy

- Reward = -1 per step
- $\mathcal{A} = \{left,\ right\}$
- Left goes left and right goes right except in the middle state where they are reversed
- States are identical from the policy's perspective
- $\pi^* = [0.41 \quad 0.59]$

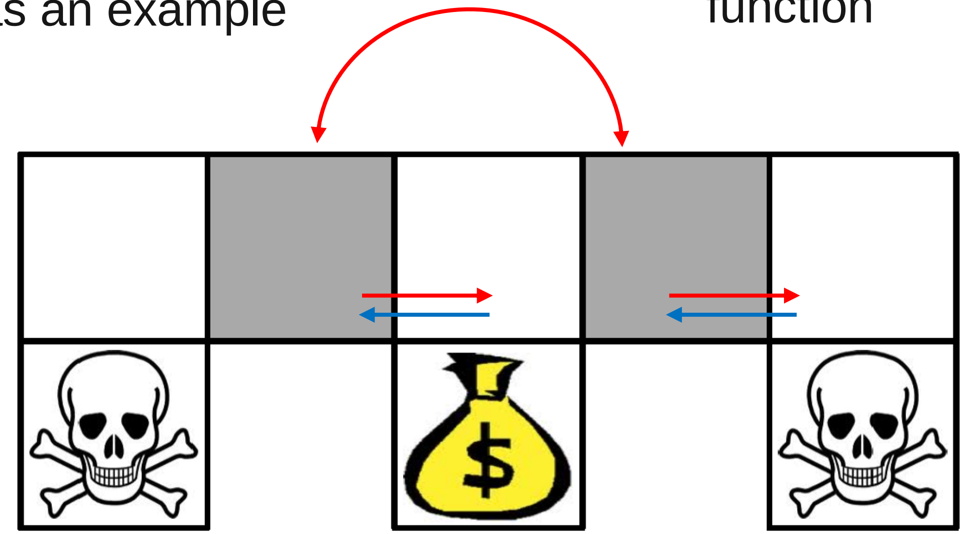
$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\mathsf{S})$$



Advantage of direct policy learning in a toy example

Two identical states in terms of feature representation $\phi(s, a)$; hence, they would have the same value function

ϵ-greedy as an example

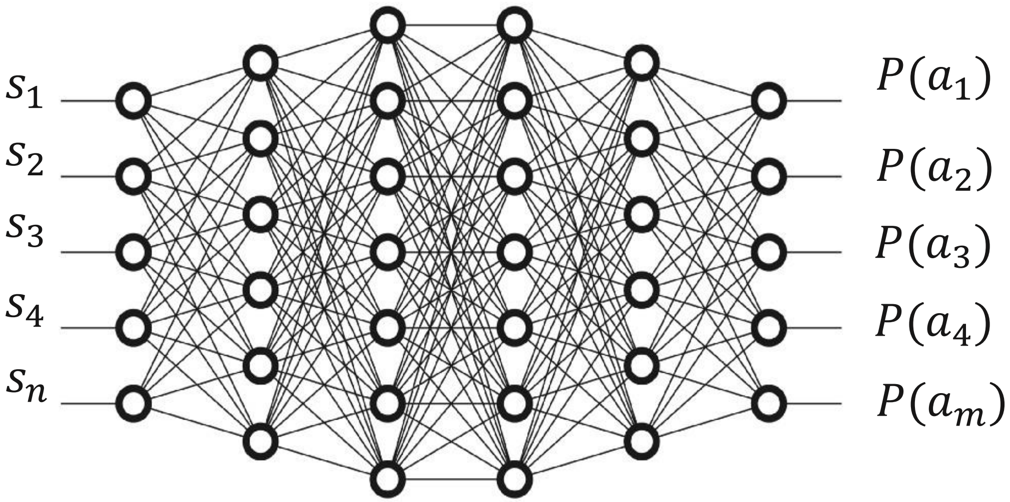


An optimal stochastic policy should randomize the actions at these two states

$\pi(a|s) \propto \exp(\theta^\top \phi(s, a))$ will work!

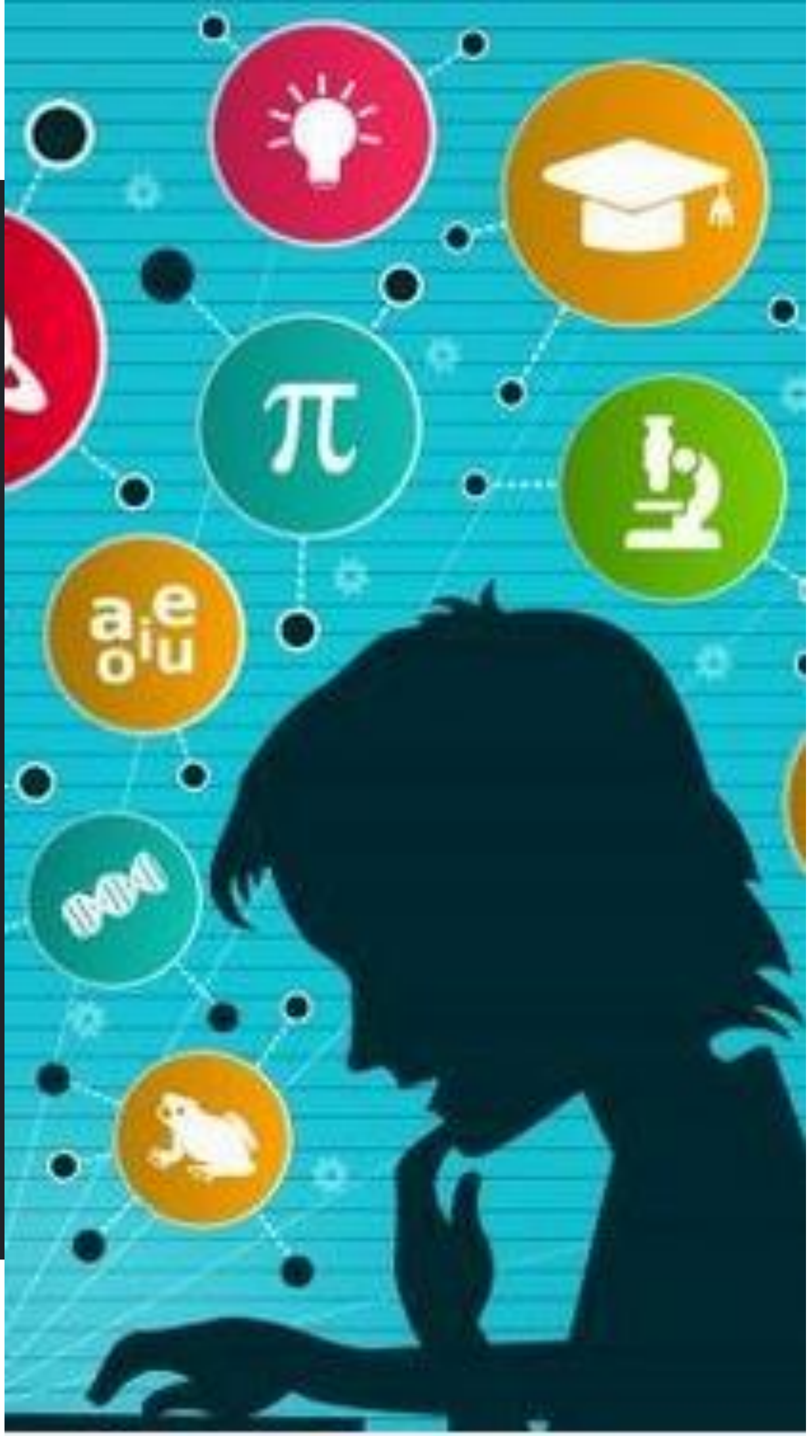
Notation

- The policy is a parametrized function: $\pi_\theta(a|s)$
  - For policy gradient we need a continuous, differentiable (soft) policy… (Softmax activation can be useful)
  - π is assumed to be differentiable with respect to θ
  - E.g., a DNN were θ is the set of weights and biases
- $J(\theta)$ is a scalar policy performance measure (sum of discounted rewards) with respect to the policy params



Improving the Policy

- SGD: $\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$
- Where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate, whose expectation approximates the gradient of the performance measure
- All methods that follow this general schema we call policy gradient methods
  - Might also learn an approximate value function
- Methods that use approximations to both policy and value functions for computing the policy's gradient are called actor–critic methods (more on this later)



Evaluate the Gradient in Performance

“

- $\nabla_\theta \widehat{J(\theta)} = ?$

- J depends on both the action selections and the distribution of states in which those selections are made
  - Both of these are affected by the policy parameter θ

- Seems impossible to solve without knowing the transition function (or the distribution of visited states)
  - $p(s'|s,a)$ is unknown in model free RL

- The PG theorem allows us to evaluate $\nabla_\theta \widehat{J(\theta)}$ without the need for $p(s'|s,a)$

Advantages of PG

1. The policy convergence over time as opposed to an epsilon greedy, value-based approach

2. Naturally applies to continuous action space as apposed to a Q learning approach

3. In many domains the policy is a simpler function to approximate

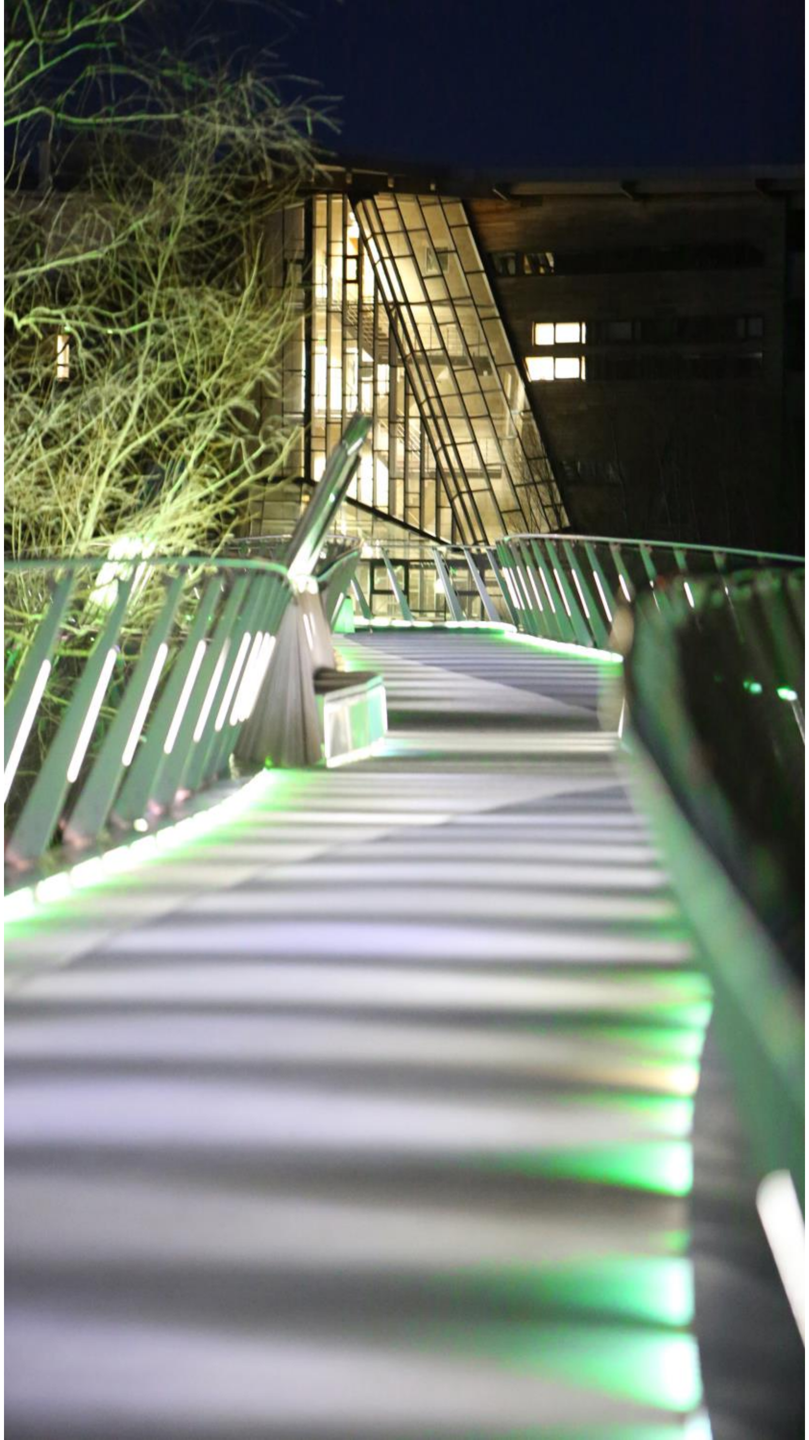
   Though this is not always the case

4. Choice of policy parameterization is sometimes a good way of injecting prior knowledge

   E.g., in phase assignment by a traffic controller

5. Can converge on stochastic optimal policies, as apposed to value-based approaches

   Useful in games with imperfect information where the optimal play is often to do two different things with specific probabilities, e.g., bluffing in Poker



Difference with Q

No explicit exploration is needed.

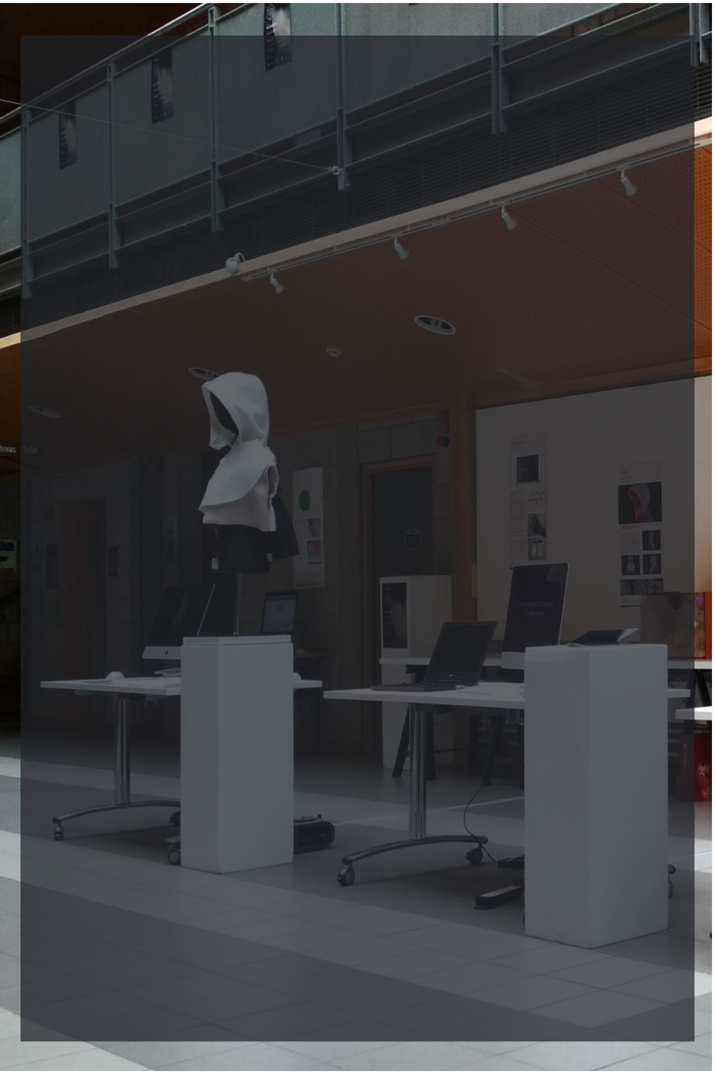
In Q-learning, we used an epsilon-greedy strategy to explore the environment and prevent our agent from getting stuck with non-optimal policy.

Now, with probabilities returned by the network, the exploration is performed automatically.

In the beginning, the network is initialized with random weights and the network returns uniform probability distribution.

This distribution corresponds to random agent behaviour.

Difference with Q



No replay buffer is used.

PG methods belong to the on-policy methods class, which means that we can’t train on data obtained from the old policy.

This is both good and bad.

The good part is that such methods usually converge faster.

The bad side is they usually require much more interaction with the environment than off-policy methods such as DQN.

Difference with Q

No target network is needed.

Here we use Q-values, but they're obtained from our experience in the environment.

In DQN, we used the target network to break the correlation in Q-values approximation, but we're not approximating it anymore.

Later, we'll see that the target network trick still can be useful in PG methods.

Homework

1. **Work through first half of chapter 18 in Geron**

2. **Implement PG for CartPole**

Thank you

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

University of Limerick,
Limerick, V94 T9PX,
Ireland.
Ollscoil Luimnigh,
Luimneach,
V94 T9PX, Éire.
+353 (0) 61 202020

ul.ie