50.039 Theory and Practice of Deep Learning

W11-S3 More on Reinforcement Learning

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Introduction

1. What are **actor-critic** learning methods? And which problems do these approaches address?
2. What are **more advanced problems** in RL?
   - Markov states
   - Partially observable environment
   - SARSA
   - Non-stationary problems
3. What is **Reinforcement Learning with Human Feedback** (RLHF)?

Time for something a bit more advanced! (cont’d)

Definitely out-of-scope, but interesting nonetheless!

A representation problem

- **Problem:** in many RL problems, the states and/or actions sets are not necessarily finite.
- In that case, it is impossible to represent the Q and V functions as tables.
- **And, even worse, for these problems, coming up with a closed form expression of the V and Q functions might prove challenging.**

Initialized

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0

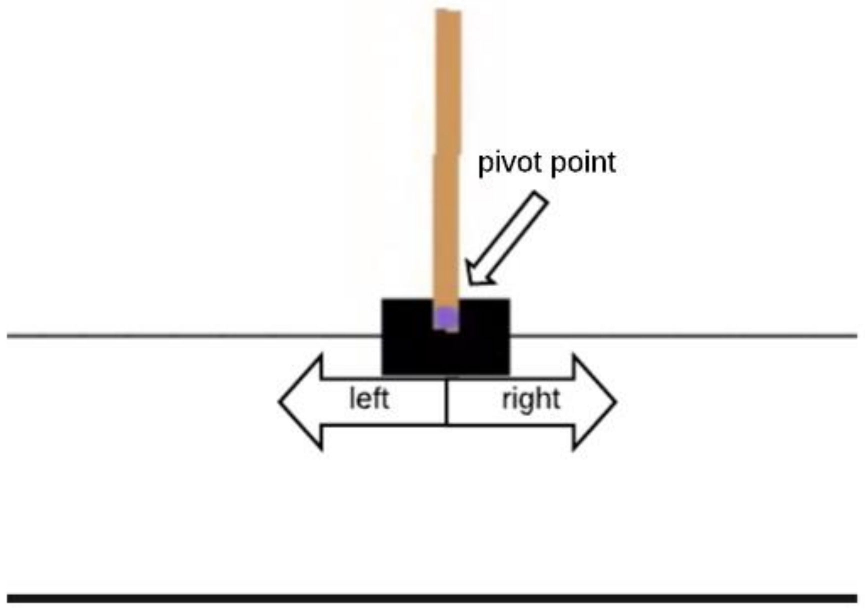
Training

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839
	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29

Toy example #3 (shown, but implementation out of the scope of this class)

Example: the cart-pole problem.

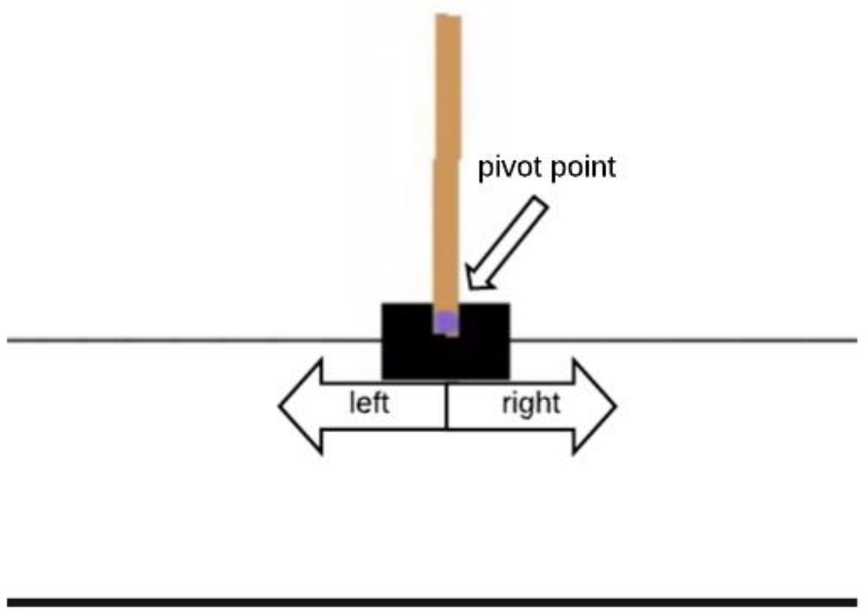
- **State:** our current visualization of the cart (i.e. an image).
- **Actions:** 2 of them, go left or go right at a fixed speed.
- **Reward:** +1 for each unit of time where the cart does not leave the screen and the pole does not fall below a certain angle.
- **Next state generation:** cart and pole both follow simple programmed rules of physics.



Toy example #3 (shown, but implementation out of the scope of this class)

- **Problem:** in many RL problems, the states and/or actions sets are not necessarily finite.
- In that case, it is impossible to represent the Q and V functions as tables.

- **How do we address this issue?**
- **Give it to an AI! (as usual)**



Toy example #3 (shown, but implementation out of the scope of this class)

- **Solution:** replace the Q-table with a Deep Neural Network, whose job is to estimate the value of each action (left/right) in the current state.
- The objective is then to train, just like before with our Q-table.
- **However, we are no longer changing the table values but the Neural Net parameters!**

```python
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

Toy example #3 (shown, but implementation out of the scope of this class)

- On each round of the game, use the Q network to compute the Q-value of both actions (left/right) in the current state.
- Use the one with the maximal value (**exploitation**) or a randomly chosen action (**exploration**).
- Use ***ϵ*-greedy policy** to decide how to explore/exploit.

```python
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # Here, t.max(1) will return largest column value of each row.
            # Second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)
```

Toy example #3

To train this DNN, we need a dataset of some sort.

- Do so by playing the game multiple times and keeping a history of the (state, action, rewards, next_state, done) tuples.
- Here, done indicates that the game has ended (out of screen or low angle on pole).
- Structure is roughly similar to our dataloaders?

```python
# Define namedtuples for transitions and history
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

```python
class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """
        Saves a transition to memory.
        """
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """
        Get sample from history.
        """
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """
        Get length of history (number of samples).
        """
        return len(self.memory)
```

Toy example #3 (shown, but implementation out of the scope of this class)

- **Core idea for memory replay:** we are trying to approximate a complex, nonlinear function Q, with a Neural Network.
- To do this, we must calculate targets using the **Bellman equation** and then consider that we have a **supervised learning** problem at hand.
- **Important:** However, one of the fundamental requirements for SGD optimization is that the training data is independent and identically distributed and when the Agent interacts with the game, the sequence of experience tuples can be highly correlated.
- The naive Q-learning algorithm that learns from each of these experiences tuples in sequential order runs the risk of getting swayed by the effects of this correlation.

Toy example #3 (shown, but implementation out of the scope of this class)

Definition (experience buffer in RL):

- We can prevent action values from oscillating or diverging catastrophically using a large buffer of our past experience and sample training data from it, instead of using our latest experience.
- This is called an **experience buffer**.
- The experience buffer contains a collection of experience tuples (state, action, rewards, next_state).
- The tuples are gradually added to the buffer as the agents keep on interacting with the game.

Toy example #3 (shown, but implementation out of the scope of this class)

Definition (experience replay):

- The simplest implementation is a buffer of fixed size, with new data added to the end of the experience buffer, so that it pushes the oldest experience out of it.
- The act of sampling a small batch of tuples from the experience buffer in order to learn is known as **experience replay**.
- In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a dataset of some sort.

- Do so by playing the game multiple times and keeping a history of the (state, action, rewards, next_state, done) tuples.
- Here, done indicates that the game has ended (out of screen or low angle on pole).
- Structure is roughly similar to our dataloaders?

```python
# Define namedtuples for transitions and history
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

```python
class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """
        Saves a transition to memory.
        """
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """
        Get sample from history.
        """
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """
        Get length of history (number of samples).
        """
        return len(self.memory)
```

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort, as well.

Our previous Q-learning was using this iterative update formula.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort, as well.

Our previous Q-learning was using this iterative update formula.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

- **Problem:** update $Q(s_t, a)$ via $Q(s_{t+1}, a)$. However, both states have only one step between them. This makes them very similar, and it is very hard for a Neural Network to distinguish between them.

Toy example #3 (shown, but implementation out of the scope of this class)

Solution: use **two Neural Networks**, one for **training** $Q(s_t, a)$ and one for producing **targets** $Q(s_{t+1}, a)$, or **evaluating** the other.

- That is, the predicted Q values of this second Q-network called the target network, are used to backpropagate through and train the main Q-network.
- **Note:** the target network's parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network.
- The idea is that using the target network's Q values to train the main Q-network will improve the stability of the training.

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To create a loss function, let us first recall that

$$Q_t^{\pi}(s_t, a_t) = R_t(s_t, a_t) + \gamma Q_{t+1}^{\pi}(s_{t+1}, \pi(s_{t+1}))$$

Let us denote the error δ as

$$\delta = Q_t^{\pi}(s_t, a_t) - \left(R_t(s_t, a_t) + \gamma \max_a Q_{t+1}^{\pi}(s_{t+1}, a)\right)$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To train our DNN, we want to minimize this error δ.

We will use the L1 norm on delta to do so.

$$L(\delta) = |\delta|$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To train our DNN, we want to minimize this error δ.

We will use the L1 norm on delta to do so.

Note: we can also use a slightly different loss function known as the **Huber loss**, which is slightly more robust to outliers.

$$L_d(\delta) = \begin{cases} \frac{1}{2}\delta^2 & if\ |\delta| \leq d \\ d\left(|\delta| - \frac{1}{2}d\right) & else \end{cases}$$

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

Trainer function

- Our trainer function will play the game 500 times.
- Keep track of different histories over the 500 games.
- Sample from history to train our main Q-Network.
- Backpropagate with mixed main and target Q-networks values.
- Occasionally update the target network.

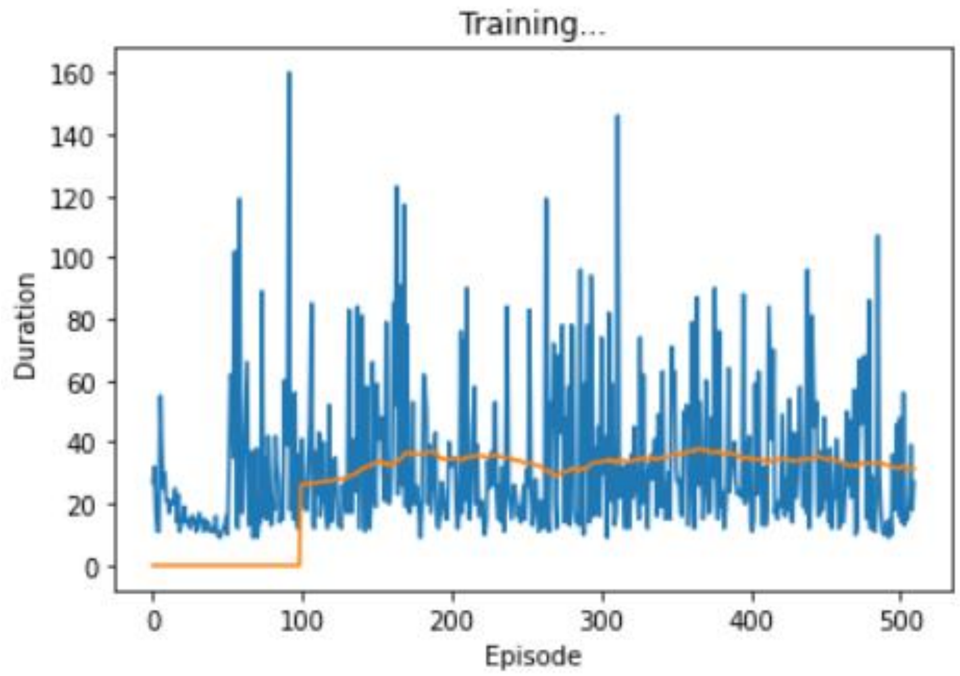
```python
"""
Full trainer on 500 iteration (for meaningful improvements)
"""
num_episodes = 500
for i_episode in range(num_episodes):
    print("Episode:", i_episode)
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

Training results

- Our RL agent will learn to balance the pole on the cart, by playing the game.
- Can display the length of each game/episode to see the progression!

- This RL approach of training some DNNs to replace the Q functions is commonly referred to as **Deep Q-learning**.

```python
def plot_durations():
    """
    Show episode durations for each episode.
    """
    plt.figure(2)
    plt.clf()
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())
```

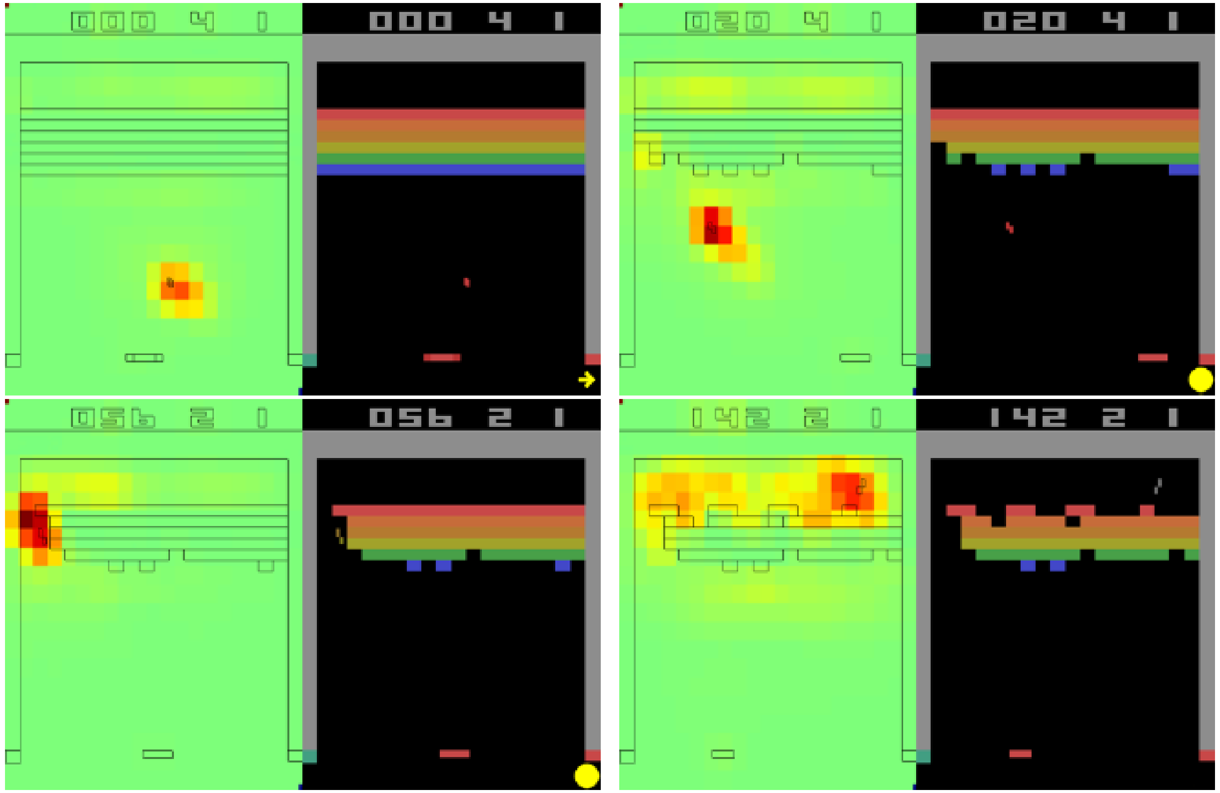


Following this cart-pole balance idea…

- Train an AI to keep a robot on its feet, despite some *"minor environment perturbations"* *(a polite way of saying you kick the hell out of the robot for fun).*
- Video: https://www.youtube.com/watch?v=NR32ULxbjYc
- **BostonDynamics** blog: https://blog.bostondynamics.com/

Following this idea of using computer vision to identify state and act…

- Train an AI to play video games with Deep Reinforcement Learning (Mnih, 2013)!
- Paper: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
- Video: https://www.youtube.com/watch?v=TmPfTpjtdgg



A quick word on actor-critic methods (more advanced stuff, out of scope)

In Deep Q-learning, we realized that most RL problems cannot have their Q functions computed easily.

- We then replaced the Q function with a Deep Neural Network to approximate this function.

A quick word on actor-critic methods (more advanced stuff, out of scope)

In Deep Q-learning, we realized that most RL problems cannot have their Q functions computed easily.

- We then replaced the Q function with a Deep Neural Network to approximate this function.
- **Additional suggestion:** Can the reward function always be computed?

A quick word on actor-critic methods (more advanced stuff, out of scope)

In Deep Q-learning, we realized that most RL problems cannot have their Q functions computed easily.

- We then replaced the Q function with a Deep Neural Network to approximate this function.
- **Additional suggestion:** Can the reward function always be computed?

→ **Replace more elements of the RL system with Deep Neural Networks.**

A quick word on actor-critic methods (more advanced stuff, out of scope)

Definition (actor-critic):

Actor-critic algorithms consist of two components.

- **Actor:** a DNN, whose purpose is to produce actions in response to given states, i.e. a policy. Can be trained as in Deep Q-learning.

A quick word on actor-critic methods (more advanced stuff, out of scope)

Definition (actor-critic):

Actor-critic algorithms consist of two components.

- **Actor:** a DNN, whose purpose is to produce actions in response to given states, i.e. a policy. Can be trained as in Deep Q-learning.
- **Critic:** a DNN, whose purpose is to evaluate the quality of the selected actions and suggesting directions for improvement, by defining a reward function or a Q function.

A quick word on actor-critic methods (more advanced stuff, out of scope)

Definition (actor-critic):

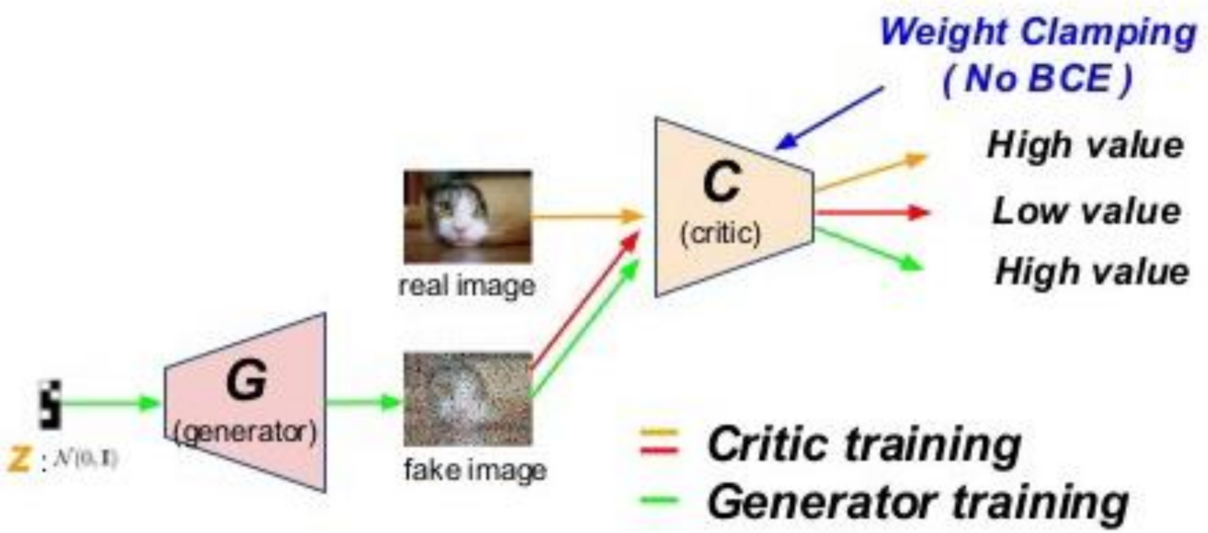
Actor-critic algorithms consist of two components.

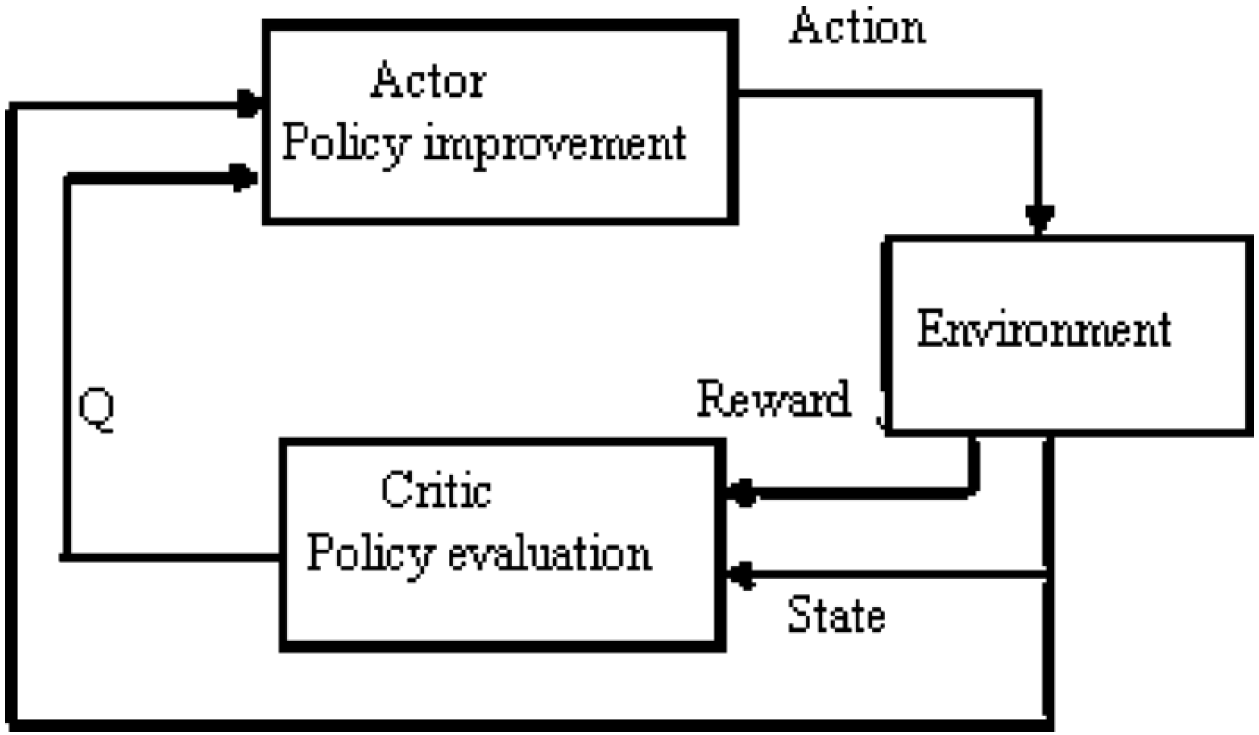
- **Actor:** a DNN, whose purpose is to produce actions in response to given states, i.e. a policy. Can be trained as in Deep Q-learning.
- **Critic:** a DNN, whose purpose is to evaluate the quality of the selected actions and suggesting directions for improvement, by defining a reward function or a Q function.

In a sense, **similar to the Generator-Critic pair** of the Wasserstein GANs!

- **Generator:** produce fake images.
- **Critic:** evaluate said images.

From Deep Q learning to actor-critic





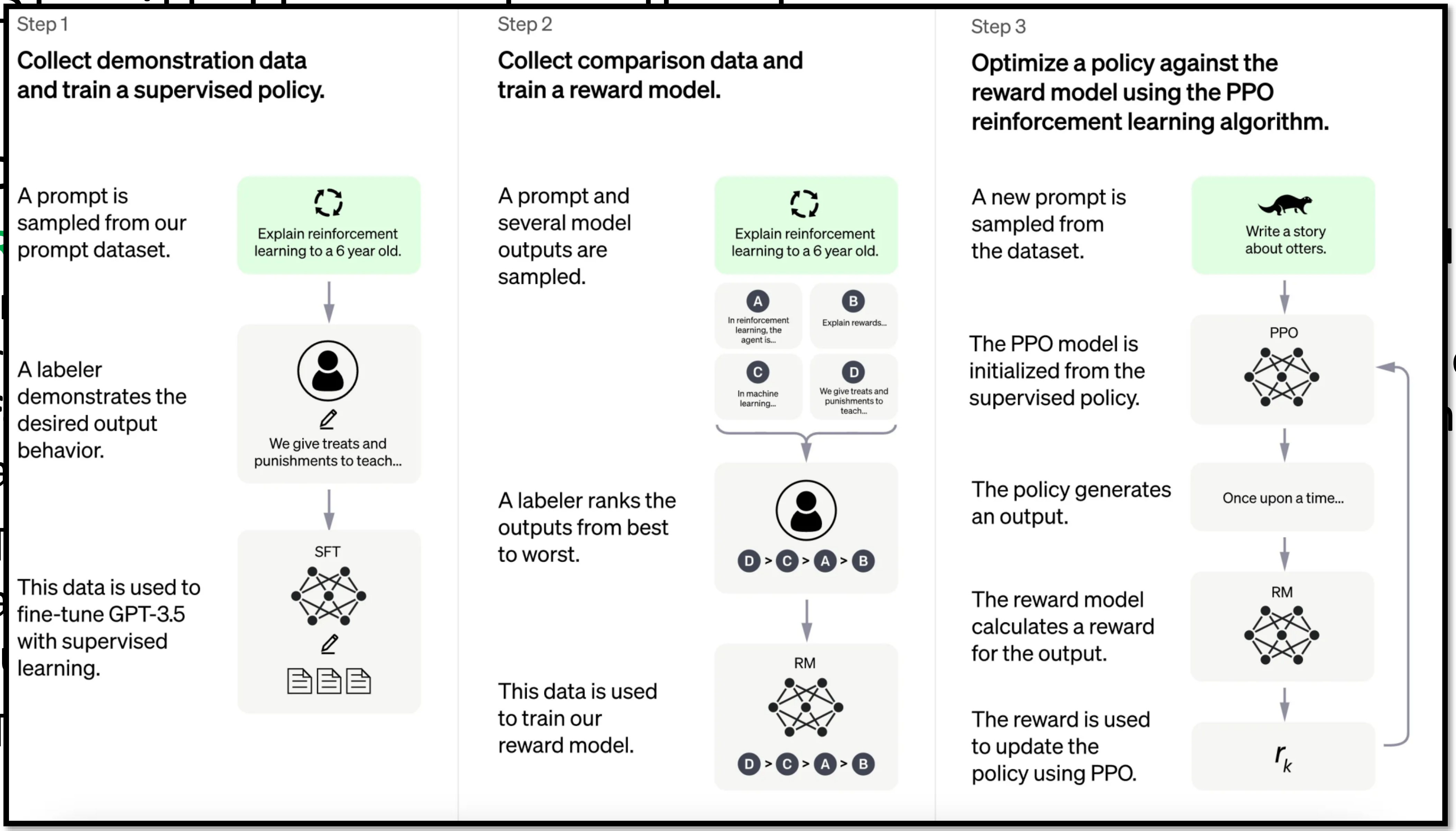
RL with Human Feedback

Definition (Reinforcement Learning with Human Feedback):

Reinforcement Learning with Human Feedback (or RLHF) is a method in which an AI model (pre-trained or not), is typically represented as a reinforcement learning agent, and is fine-tuned/trained based on some **feedback/reward provided by humans** to improve its performance on a specific task.

This approach combines the strengths of reinforcement learning algorithms, which learn through trial and error, with the expertise and judgment of human evaluators.

The way ChatGPT was trained in fact!


Step 1
Collect demonstration data and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain reinforcement learning to a 6 year old.
A labeler demonstrates the desired output behavior.
We give treats and punishments to teach...
This data is used to fine-tune GPT-3.5 with supervised learning.
SFT
Step 2
Collect comparison data and train a reward model.
A prompt and several model outputs are sampled.
Explain reinforcement learning to a 6 year old.
A
In reinforcement learning, the agent is...
B
Explain rewards...
C
In machine learning...
D
We give treats and punishments to teach...
A labeler ranks the outputs from best to worst.
D > C > A > B
This data is used to train our reward model.
RM
D > C > A > B
Step 3
Optimize a policy against the reward model using the PPO reinforcement learning algorithm.
A new prompt is sampled from the dataset.
Write a story about otters.
The PPO model is initialized from the supervised policy.
PPO
The policy generates an output.
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Markov Decision Processes (more advanced stuff, out of scope)

Sometimes, the transition from state s_t to next state s_{t+1}, following from action a_t, will not always be **deterministic**.

- In that case, we have to define some system dynamics, as below.

$$p(s',r|s,a) = P(s_{t+1} = s', r_t = r \mid s_t = s, a_t = a)$$

- Similar to our previous problem, with a stochastic twist.
- It is called a Markov Decision Process (MDP) problem.

Markov Decision Processes (more advanced stuff, out of scope)

- In MDPs, all the previous formulas have to be reworked to account for the stochastic aspect of the problem.

$$V_t^{\pi}(s) = E[G_t \mid s_t = s]$$

$$Q_t^{\pi}(s, a) = E[G_t \mid s_t = s, a_t = a]$$

- In MDPs, the Q and V functions can still be learned from experience, but their Bellman equations change slightly, to account for the stochasticity.

Markov Decision Processes (more advanced stuff, out of scope)

- In MDPs, the Q and V functions can still be learned from experience, but their Bellman equations change slightly, to account for the stochasticity.

$$V_t^\pi(s) = E[G_t \mid s_t = s]$$
$$V_t^\pi(s) = E[R_t + \gamma G_{t+1} \mid s_t = s]$$
$$V_t^\pi(s) = \sum_a \pi(a \mid s) \sum_{s\prime} \sum_r p(s', r \mid s, a) \left[r + \gamma E[G_{t+1} \mid s_{t+1=s\prime}]\right]$$

$$\boldsymbol{V_t^\pi(s) = \sum_a \pi(a \mid s) \sum_{s\prime} \sum_r p(s', r \mid s, a) \left[r + \gamma V_{t+1}^\pi(s')\right]}$$

Restricted

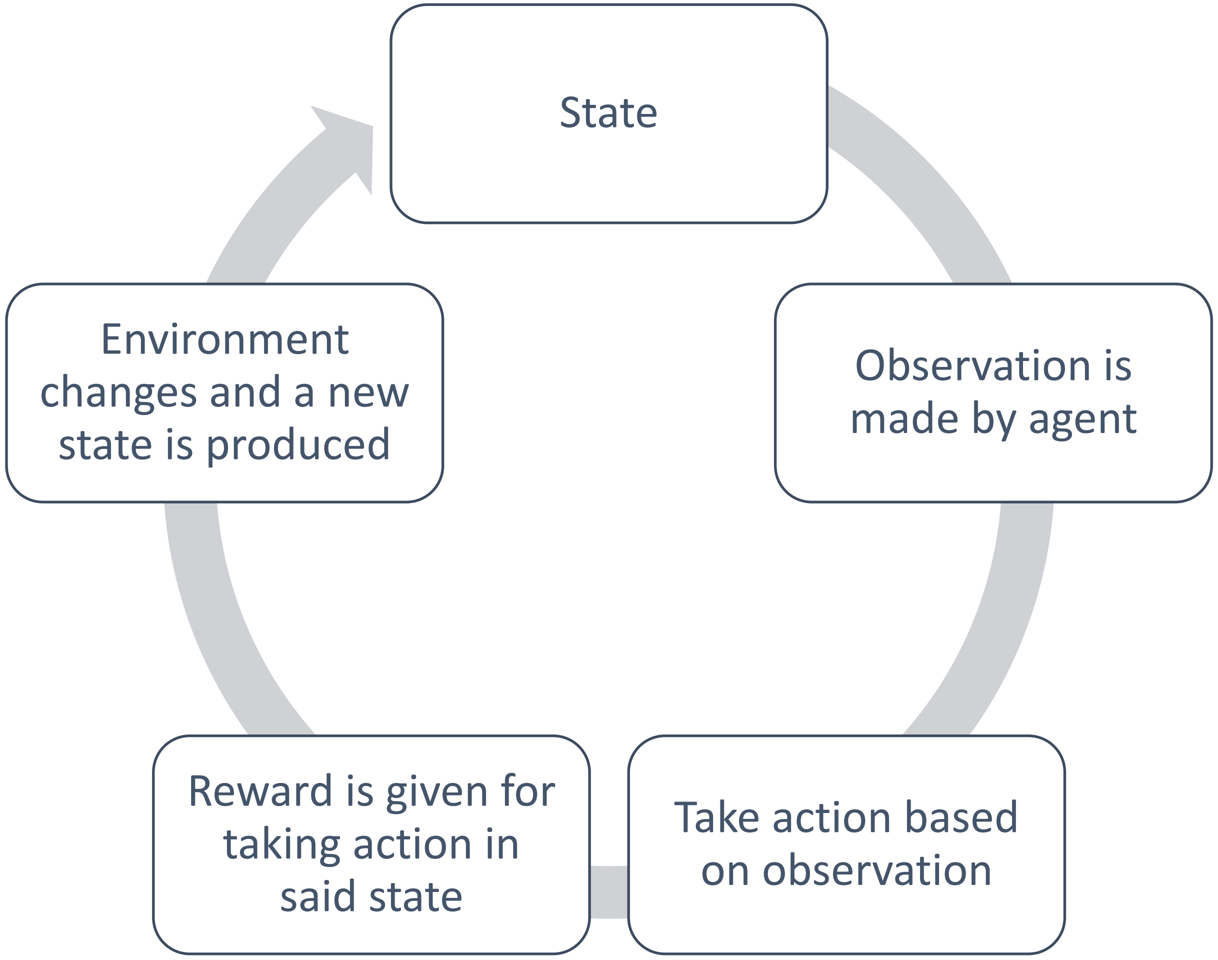
Partially observable MDP
(more advanced stuff, out of scope)

Definition (partially observable Markov Decision Process):

In our original RL framework, we assumed that the agent was seeing the exact state of the game at each time t.

This is also an assumption, which can be challenged.

State s_t is ground truth, agent received observation o_t and uses to decide on action.



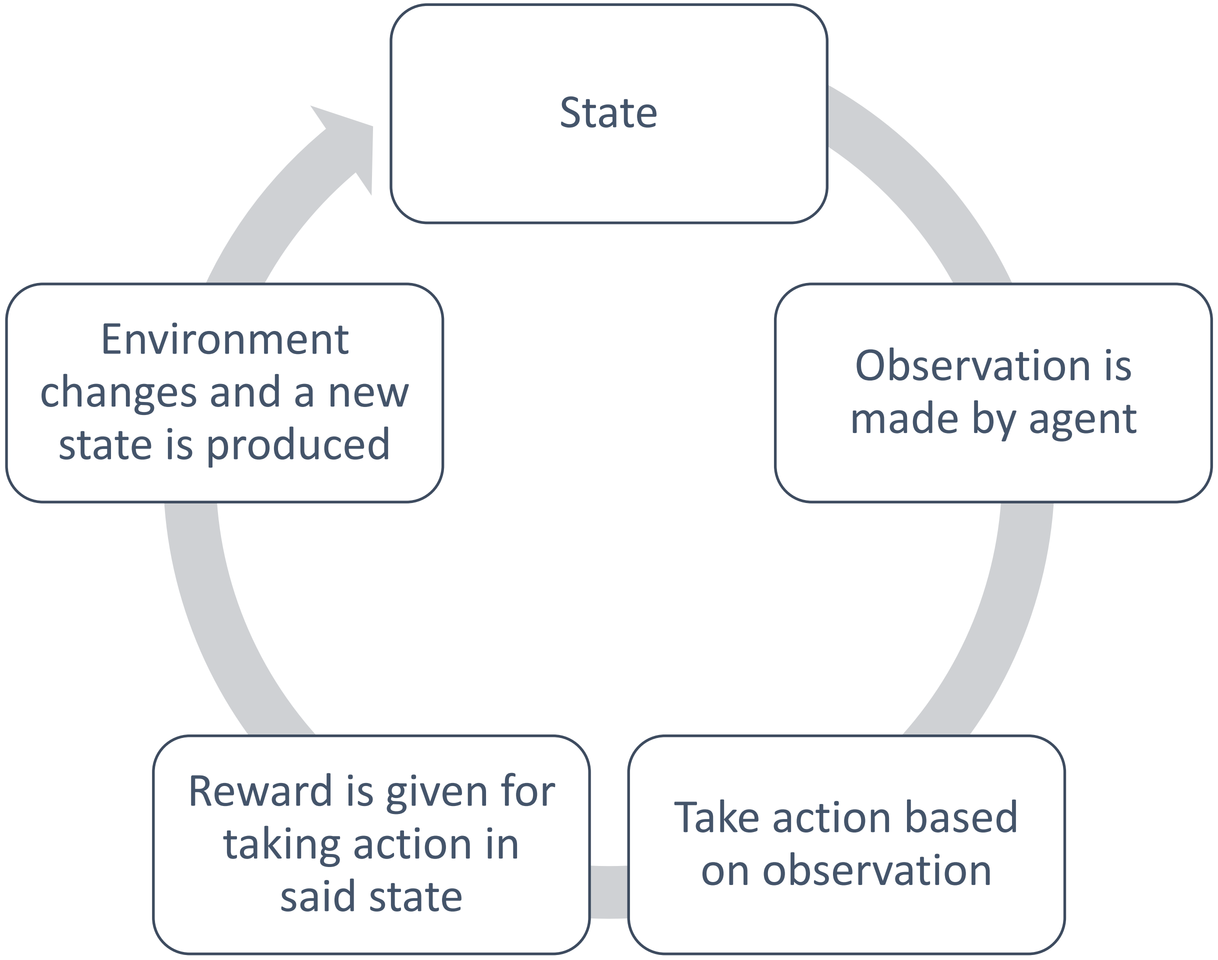
Partially observable MDP (more advanced stuff, out of scope)

Definition (partially observable Markov Decision Process):

The agent then decides on an observation made of the actual (partially hidden) state.

This is called a **partially observable Markov Decision Process**.

Agent will have to learn to observe on top of acting properly (e.g. card game, with opponent hiding his/her hand).



Partially observable MDP

(n

De
Ma

Th
ob
(pa

Th
ob
Pr

Ag
ob
(e.
hic

ation is
agent

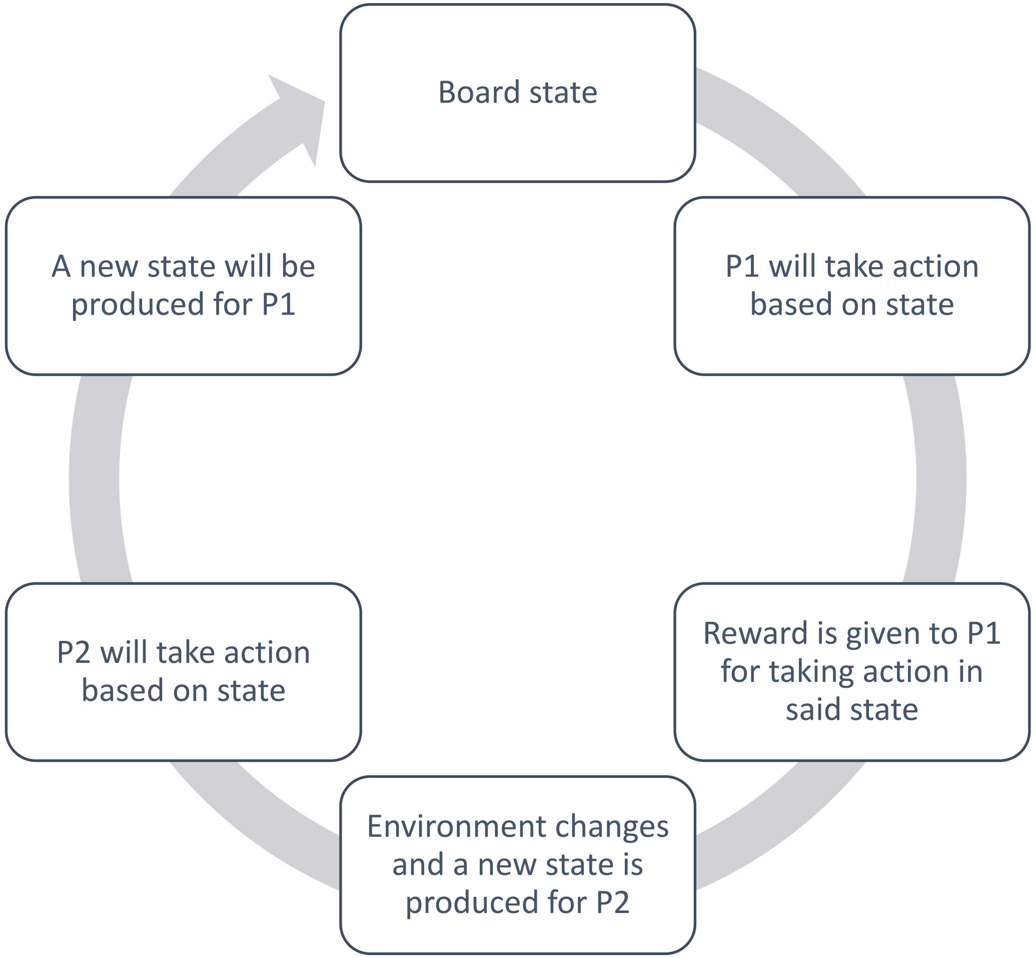
d

State-Action-Reward-State-Action or SARSA (more advanced stuff, out of scope)

- In many problems, e.g. Go, the new state seen by a given player is not the immediate result of the action of the said player.
- Instead, another player has to act first, before a new state is produced.

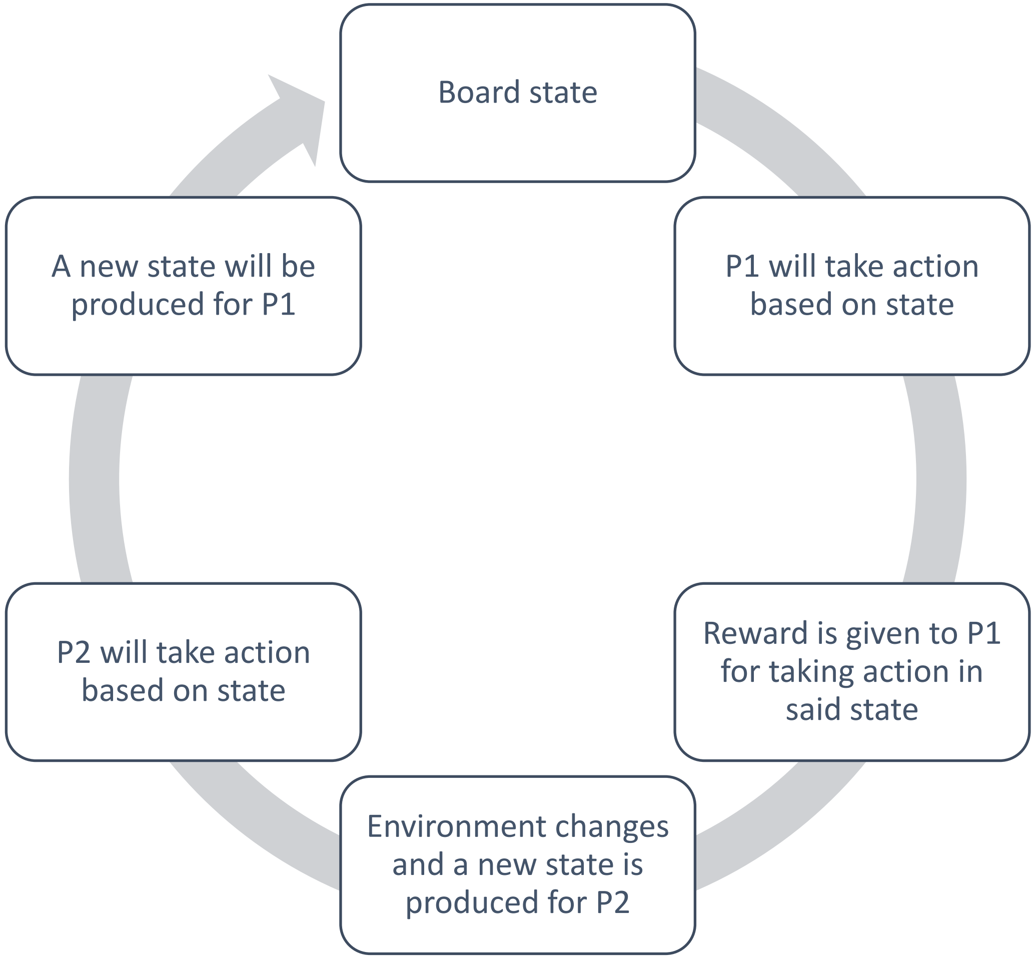
State-Action-Reward-State-Action or SARSA (more advanced stuff, out of scope)

- In many problems, e.g. Go, the new state seen by a given player is not the immediate result of the action of the said player.
- Instead, another player has to act first, before a new state is produced.
- This adds steps to the cycle, which becomes (state, action, reward, state_P2, action_P2).



State-Action-Reward-State-Action or SARSA (more advanced stuff, out of scope)

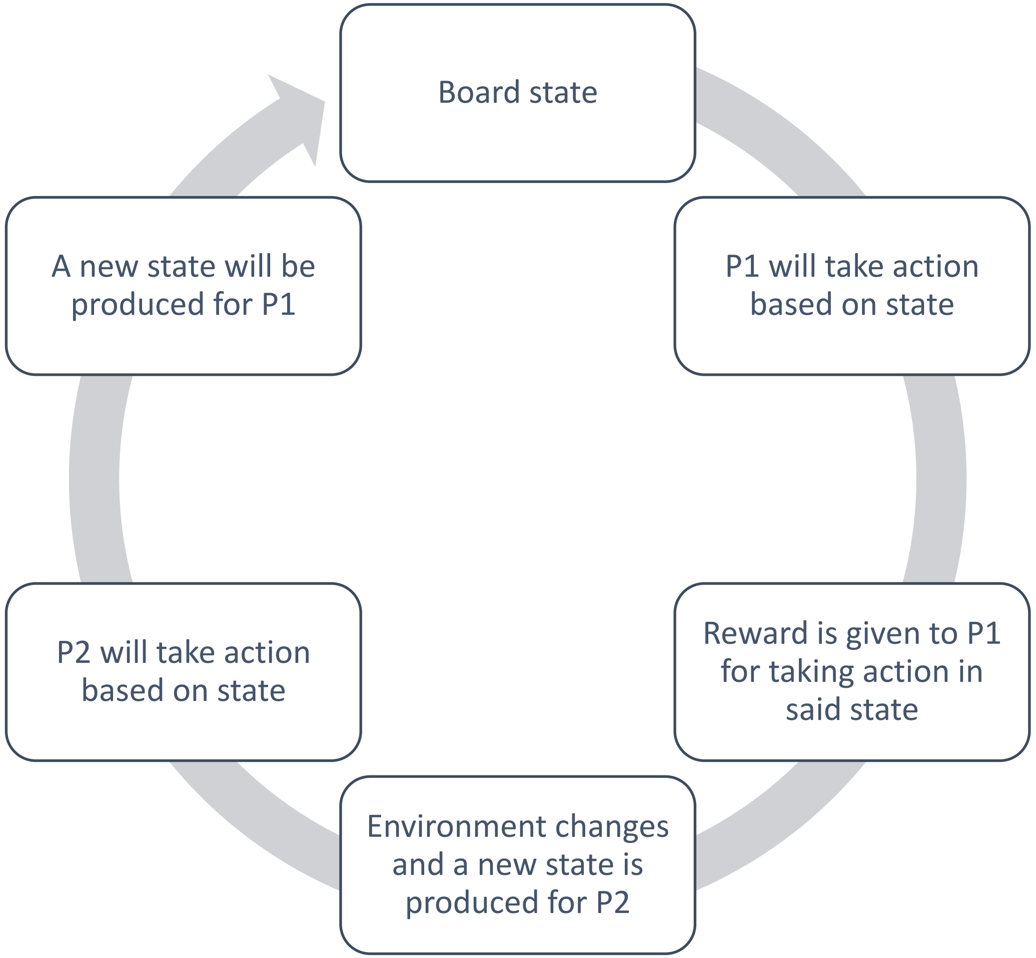
- This is called a **State-Action-Reward-State-Action (SARSA)** type of problem.
- In that case the agent has to learn how to play, but also has to learn how another player might respond to its actions.
- **RL meets game theory!**



State-Action-Reward-State-Action or SARSA (more advanced stuff, out of scope)

- This is called a **State-Action-Reward-State-Action (SARSA)** type of problem.
- In that case the agent has to learn how to play, but also has to learn how another player might respond to its actions.

→ Train two AIs at the same time (one for black, one for whites)? Make them play against each other and train together like generator-critic in WGANs?



Conclusion

1. What are **actor-critic** learning methods? And which problems do these approaches address?
2. What are **more advanced problems** in RL?
   - Markov states
   - Partially observable environment
   - SARSA
   - Non-stationary problems
3. What is **Reinforcement Learning with Human Feedback** (RLHF)?