# 50.039 Theory and Practice of Deep Learning

## W11-S3 More on Reinforcement Learning

Matthieu De Mari

# About this week

1. What is **Reinforcement Learning**?
2. What are the key ideas behind **reinforcement learning** and its **framework**?
3. What is the **exploration vs. exploitation tradeoff**?
4. How do we **train** an **RL agent** by exploring, then progressively exploiting?
5. What are some **advanced strategies** in **multi-arm bandit problems**?
6. What are the **Q** and **V functions** for a RL problem?
7. What is **Q-learning** and how can it be implemented in RL problems?

# About this week

8. What is **Deep Q Learning**? And which problem does it address?
9. What is **RL with Human Feedback**?
10. How to implement RLHF via the **REINFORCE** algorithm?
11. What are **actor-critic** learning methods? And which problems do these approaches address?
12. What are more advanced problems in RL?
    - Markov states
    - Partially observable environment
    - SARSA
    - Non-stationary problems

# A representation problem

**Problem:** in many RL problems, the number of states and/or the number of actions sets are not necessarily finite.

- It is then impossible to represent the $Q$ function as tables.
- **And, even worse, for these problems, coming up with a closed form expression of the $V$ and $Q$ functions might prove simply too 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 |

# Limitations of Q-learning (last lecture)

**Open question: What if the number of states and/or the number of actions is not finite?**

- We would not be able to use a Q-table to store values…

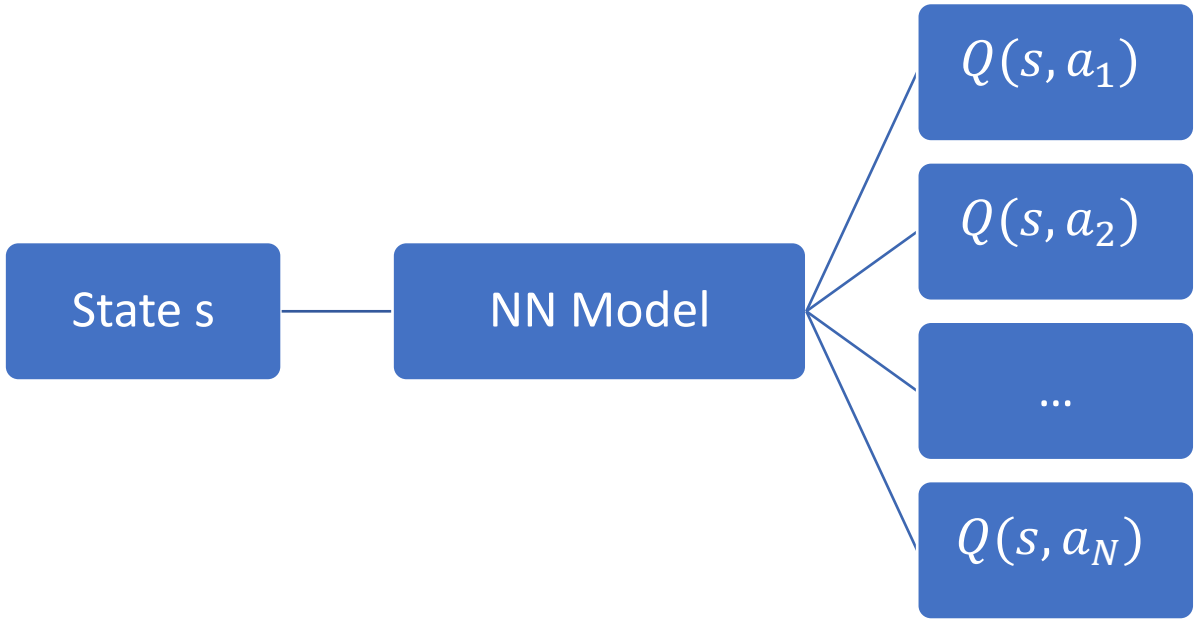
→ **Idea:** Replace the table with a Deep Learning model that predicts the value of Q for the given state and each possible action!

- Train that model to match the theoretical Q-value!
- Reuse all ideas from before regarding the updating of Q-values, exploration vs. exploitation, etc.

# Using a NN model as a Q-function

As we have mentioned, Q-Tables are impractical for large or continuous state spaces.

- Neural networks can, however, be used as **function approximators** for $Q(s, a)$
- Model input: state $s$
- Model output: approximate value of $Q(s, a)$ for each possible action $a$.



# Using a NN model as a Q-function

In Notebook 6, we simply represent this model as

- A sequence of Linear layers,
- Interleaved with some simple activations such as ReLU.

Later on, we will adjust the input_size and output_sizes to match the dimensionalities of our RL problem, that is the states dimensionality and number of actions k for $Q(s, a_k)$.

```python
# Neural network for approximating Q-values
class QNetwork(nn.Module):
    """
    Out Q-Neural Network for predictions
    """
    def __init__(self, input_size, hidden_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.fc3(x)
        return x
```

# Using a NN model as a Q-function

Next, we define our Deep Q-Learning agent, which interacts with the environment and learns an optimal policy using a neural network. Our DQLAgent class represents an AI agent that:

1. Uses a deep neural network (Q-Network) to approximate Q-values.
2. Explores and exploits the environment.
3. Trains using Deep Q-Learning (DQL) by adjusting NN parameters to learn the optimal Q-values, so that the RL agent policy is able to find the optimal actions to use in any given state.
4. Improves over time by balancing exploration and exploitation, with exploration rate decay.

# Using a NN model as a Q-function

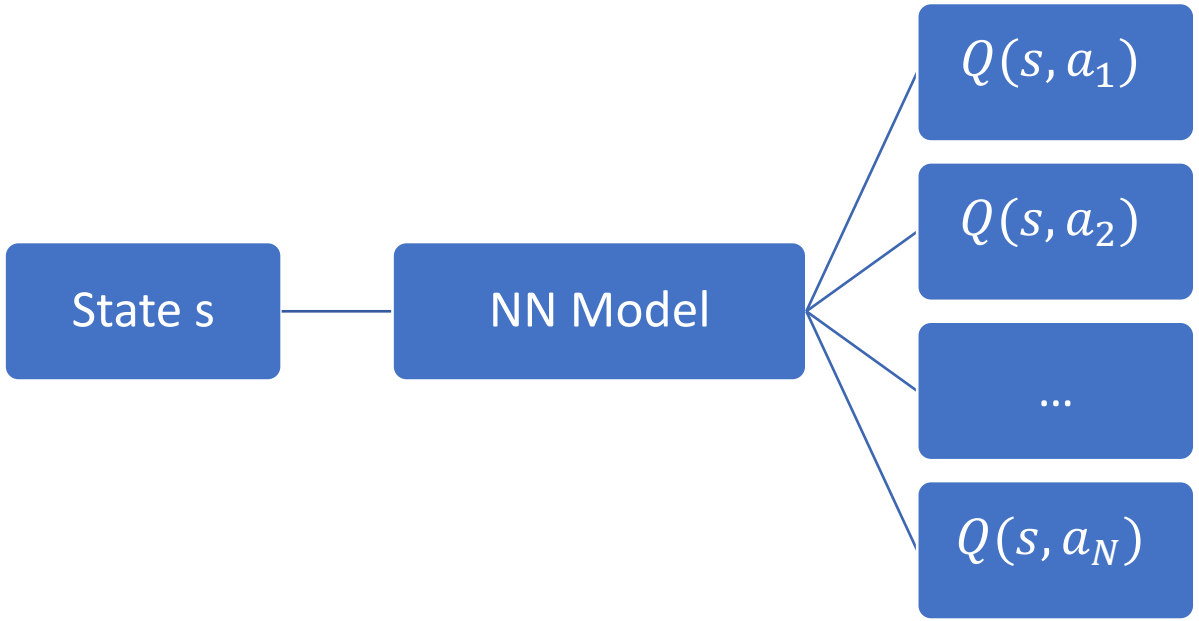
Next, we define our Deep Q-Learning agent, which interacts with the environment and learns an optimal policy using a neural network. Our DQLAgent class represents an AI agent that:

1. **Uses a deep neural network (Q-Network) to approximate Q-values.**
2. **Explores and exploits the environment.**
3. Trains using Deep Q-Learning (DQL) by adjusting NN parameters to learn the optimal Q-values, so that the RL agent policy is able to find the optimal actions to use in any given state.
4. Improves over time by balancing exploration and exploitation, with exploration rate decay.

# Using a NN model as a Q-function

The RL agent in our problem will implement two behaviors for exploration and exploitation.

- **Exploration:** Use the model to compute all $Q(s, a_k)$ values for a given state $s$ and all possible actions $a_k, \forall k$.
- Implement a random decision policy of some sort (could be uniform random or softmax based to encourage certain decisions for exploration).

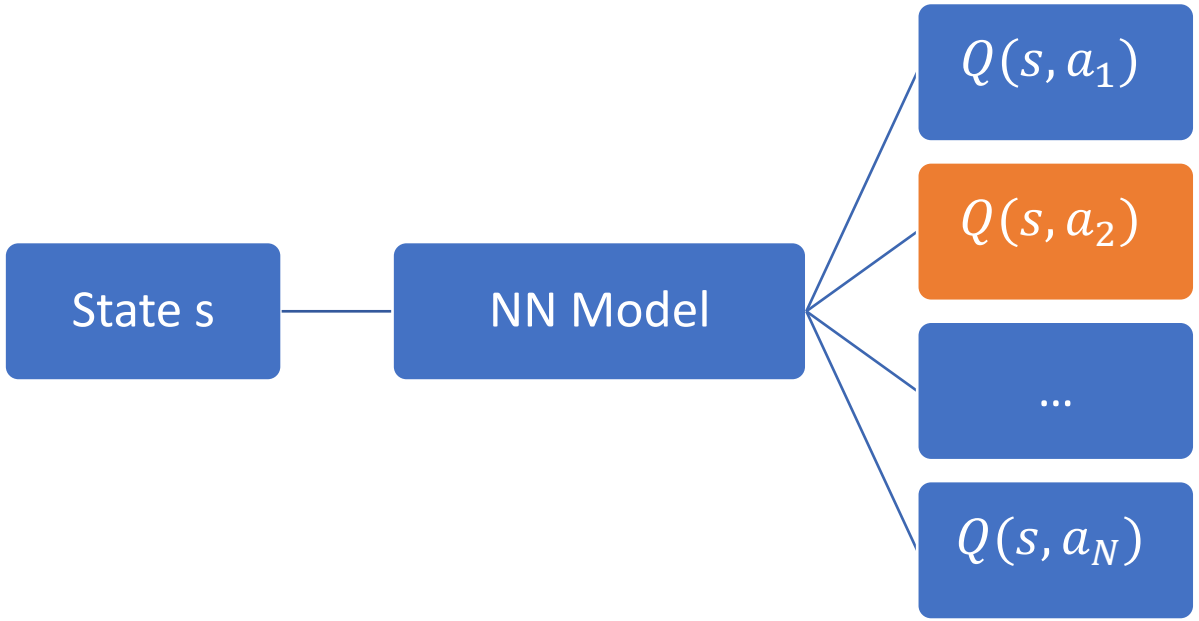


# Using a NN model as a Q-function

The RL agent in our problem will implement two behaviors for exploration and exploitation.

- **Exploitation:** Use the model to compute all $Q(s, a_k)$ values for a given state $s$ and all possible actions $a_k, \forall k$.
- Use the action $\boldsymbol{a}$ that has the **maximal** $Q(s, a)$ value.

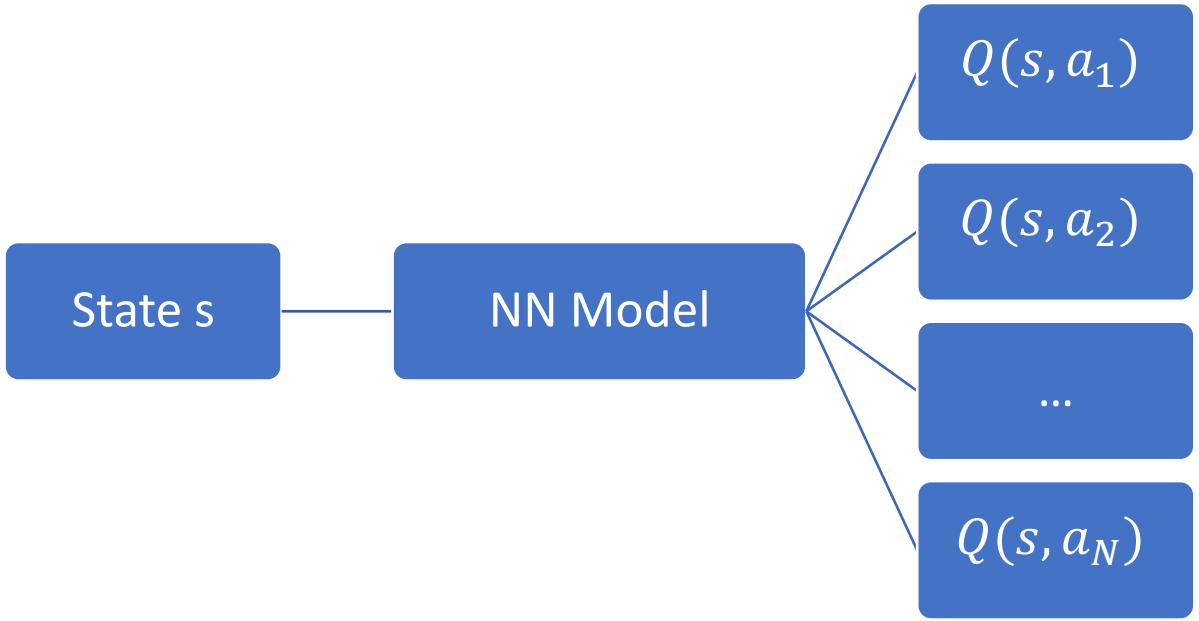
$$\boldsymbol{a} = \max_k [Q(s, a_k)]$$



# Using a NN model as a Q-function

As before, your policy $\pi$ will progressively transition from exploration to exploitation.

- Use a value $\epsilon \in [0, 1]$, which defines the probability of making an exploration move.
- The value of $\epsilon$ starts as 1 to encourage **exploration** and slowly decays towards 0 to move into **exploitation**.
- All ideas from W11S1 are open.



# Using a NN model as a Q-function

For instance, reuse the $\epsilon$-greedy strategy from W11S1.

The policy for our RL agent is then decided based on an exploration rate $\epsilon$, which will decay over time.

- Explore by choosing a random action among all available ones.
- Exploit by choosing the maximal value produced by the model.

```
# Epsilon-greedy action selection
if np.random.random() < exploration_rate:
    # Exploration move, randomly deciding
    action = random.choice(self.environment.actions)
else:
    # Exploitation move, using the Neural Network to decide on the best move
    with torch.no_grad():
        q_values = self.q_network(state)
        action = torch.argmax(q_values).item()
```

# Using a NN model as a Q-function

Next, we define our Deep Q-Learning agent, which interacts with the environment and learns an optimal policy using a neural network. Our DQLAgent class represents an AI agent that:

1. Uses a deep neural network (Q-Network) to approximate Q-values.
2. Explores and exploits the environment.
3. **Trains using Deep Q-Learning (DQL) by adjusting NN parameters to learn the optimal Q-values, so that the RL agent policy is able to find the optimal actions to use in any given state.**
4. Improves over time by balancing exploration and exploitation, with exploration rate decay.

# Using a NN model as a Q-function

**Question:** How would we train this model so that it learns to produce a good approximation for the Q values?

- Earlier, we had a Q-learning update formula, but it worked by adjusting the Q values in a Q-table directly.
- Here, we have to find a way to adjust the NN model parameters to improve the Q-values indirectly.

```python
# Neural network for approximating Q-values
class QNetwork(nn.Module):
    """
    Out Q-Neural Network for predictions
    """
    def __init__(self, input_size, hidden_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.fc3(x)
        return x
```

# Using a NN model as a Q-function

- **Previously:** When we had access to a Q-table and wanted to update its values, we would use the **Q-learning** formula, shown below.

## Updating the Q-table via Q-learning

- This sequence can then be used, to update the Q-table, according to

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

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- In a sense, this formula is the "**equivalent**" of our **gradient descent update**, but in the case of reinforcement learning, and is commonly referred to as **Q-learning**.

# Using a NN model as a Q-function

**Definition (a loss function for deep Q-learning?):**

Let us reuse the intuition behind this **temporal difference (TD) formula that was used in Q-learning**.

- The TD error is the difference $\delta$ between the current estimate given by the model at a given time $Q(s, a)$ and a better estimate $y$ (to be defined later), towards which we want to improve.

$$\delta = y - Q(s, a)$$

# Using a NN model as a Q-function

**Definition (a loss function for deep Q-learning?):**

- The TD target $y$ is defined in two possible ways.
- We define it as the **current reward $R$**, plus the best $Q$ value we would get if we picked the best action $a'$ to use in the next state $s'$ that results from using chosen action $a$ in current state $s$.

$$y = R(s, a) + \gamma \max_{a'}[Q(s', a')]$$

Here, $\gamma$ is the **discount factor**, as before in the Q-learning formula.

# Using a NN model as a Q-function

**Definition (a loss function for deep Q-learning?):**

- The TD target $y$ is defined in two possible ways.
- This can only be calculated if the game does not end immediately after using action $a$ in current state $s$.
- If the action $a$ is the last one produced in the game, we will then simply define $y$ as the current reward.

$$y = R(s, a)$$

# Using a NN model as a Q-function

**Definition (a loss function for deep Q-learning?):**

- The goal is then to reduce $\delta$ to zero by adjusting the parameters in the model that approximates the values of $Q(s, a)$.
- A **possible loss function** that encourages the value of $\delta$ going to 0, is then simply the squared TD error $L$:

$$L = \delta^2 = (y - Q(S, a))^2$$

This loss function is commonly referred to as the **deep Q-learning loss**.

# Implementing our loss function and training

Following the mathematical intuition about the TD loss, we can implement the loss function $L$ as shown below.

```
# Take action and observe the result, update reward
next_state, reward, status = self.environment.step(action)
next_state = torch.tensor(next_state.flatten(), dtype = torch.float32)
cumulative_reward += reward

# Compute target Q-value
with torch.no_grad():
    max_next_q = torch.max(self.q_network(next_state)).item()
    target = reward + discount*max_next_q if status == Status.PLAYING else reward
```

# Implementing our loss function and training

Following the mathematical intuition about the TD loss, we can implement the loss function $L$ as shown below.

Then, trust that the autograd and optimizer.step() will do the job!

```
# Update Q-value using the loss
q_values = self.q_network(state)
target_q_values = q_values.clone()
target_q_values[action] = target
loss = self.criterion(q_values, target_q_values)

# Backpropagation
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

# Using a NN model as a Q-function

Next, we define our Deep Q-Learning agent, which interacts with the environment and learns an optimal policy using a neural network. Our DQLAgent class represents an AI agent that:

1. Uses a deep neural network (Q-Network) to approximate Q-values.
2. Explores and exploits the environment.
3. Trains using Deep Q-Learning (DQL) by adjusting NN parameters to learn the optimal Q-values, so that the RL agent policy is able to find the optimal actions to use in any given state.
4. **Improves over time by balancing exploration and exploitation, with exploration rate decay (and we keep track of training performance!)**

# Training procedure

- Initialize hyperparameters for RL agent (discount coefficient $\gamma$, exploration rate $\epsilon$ initial value and decaying factor, number of episodes the agent will play the game, etc.)
- Initialize rewards and performance metrics to track training.

```
def train(self, discount = 0.90, exploration_rate = 1, exploration_decay = 0.995, episodes = 200, stop_at_convergence = True):
    """
    Train the model using Deep Q-Learning.
    """
    # Parameters for tracking training
    check_convergence_every = 5
    cumulative_reward = 0
    cumulative_reward_history = []
    win_history = []
```

# Training procedure

- Play the game a certain number of times, resetting the environment every time and choosing a random starting point (why not necessarily the same one every time?)

```
# Play the game for the specified number of times
for episode in range(1, episodes + 1):
    # Start from a randomly decided cell
    if not start_list:
        start_list = self.environment.empty.copy()
    start_cell = random.choice(start_list)
    start_list.remove(start_cell)
    state = self.environment.reset(start_cell)
    state = torch.tensor(state.flatten(), dtype = torch.float32)
    # Display
    if episode % 100 == 1 or episode == episodes:
        print(f"Episode: {episode} - Exploration rate: {exploration_rate}")
```

# Training procedure

- At the end of each episode, adjust the exploration rate (decay) and check if there is a convergence on the model learning (early stopping).

```
# Check convergence and stop early if agent has learnt everything already
if episode % check_convergence_every == 0:
    w_all, win_rate = self.environment.check_win_all(self)
    win_history.append((episode, win_rate))
    if w_all is True and stop_at_convergence is True:
        print("Won from all start cells. Stopped learning")
        break

# Decay exploration rate for exploration-exploitation trade-off
exploration_rate *= exploration_decay
```

# Training procedure

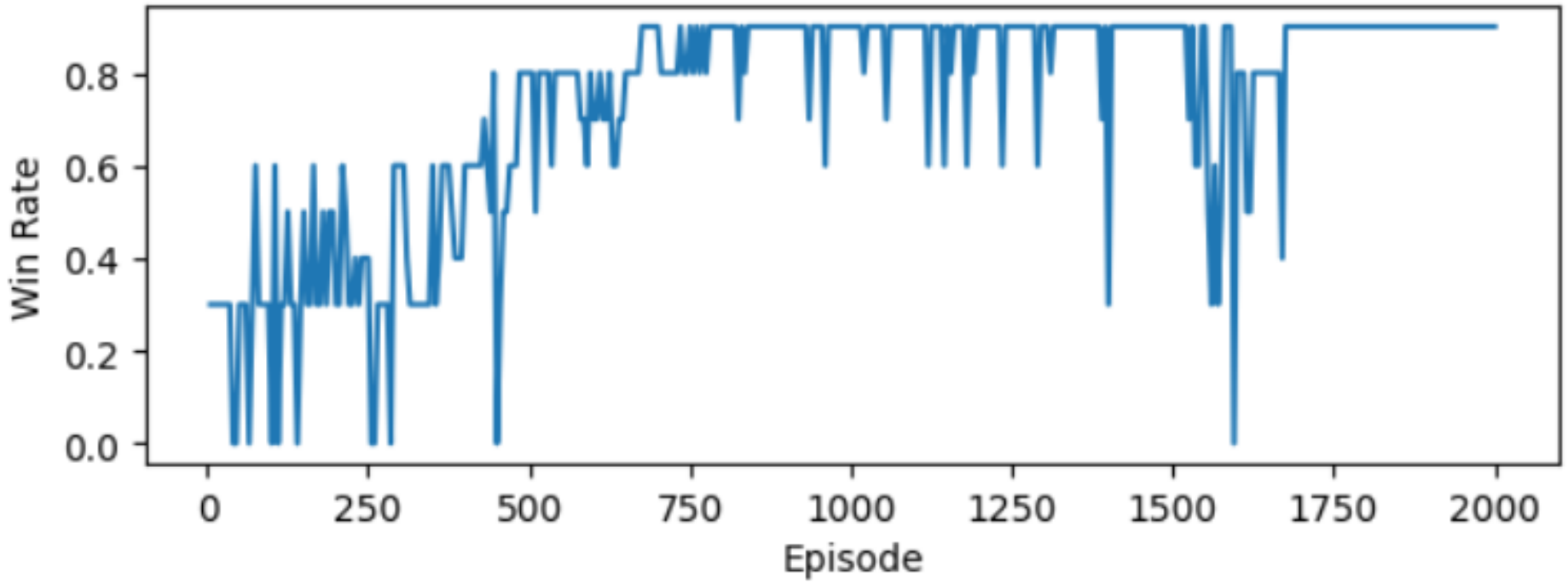
- Run the trainer function and see exploration rate decay to 0.

```
# Train agent
h, w = model.train(discount = 0.95, exploration_rate = 1.0, exploration_decay = 0.995, episodes = 2000, stop_at_convergence = True)
```

```
Episode: 1 - Exploration rate: 1.0
Episode: 101 - Exploration rate: 0.6057704364907278
Episode: 201 - Exploration rate: 0.3669578217261671
Episode: 301 - Exploration rate: 0.22229219984074702
Episode: 401 - Exploration rate: 0.1346580429260134
Episode: 501 - Exploration rate: 0.08157186144027828
Episode: 601 - Exploration rate: 0.0494138221100385
Episode: 701 - Exploration rate: 0.029933432588273214
Episode: 801 - Exploration rate: 0.018132788524664028
Episode: 901 - Exploration rate: 0.01098430721937979
Episode: 1001 - Exploration rate: 0.006653968578831948
Episode: 1101 - Exploration rate: 0.004030777450394616
Episode: 1201 - Exploration rate: 0.002441725815522529
Episode: 1301 - Exploration rate: 0.0014791253130597608
Episode: 1401 - Exploration rate: 0.0008960103865166974
Episode: 1501 - Exploration rate: 0.0005427766029404454
Episode: 1601 - Exploration rate: 0.0003287980196801882
```

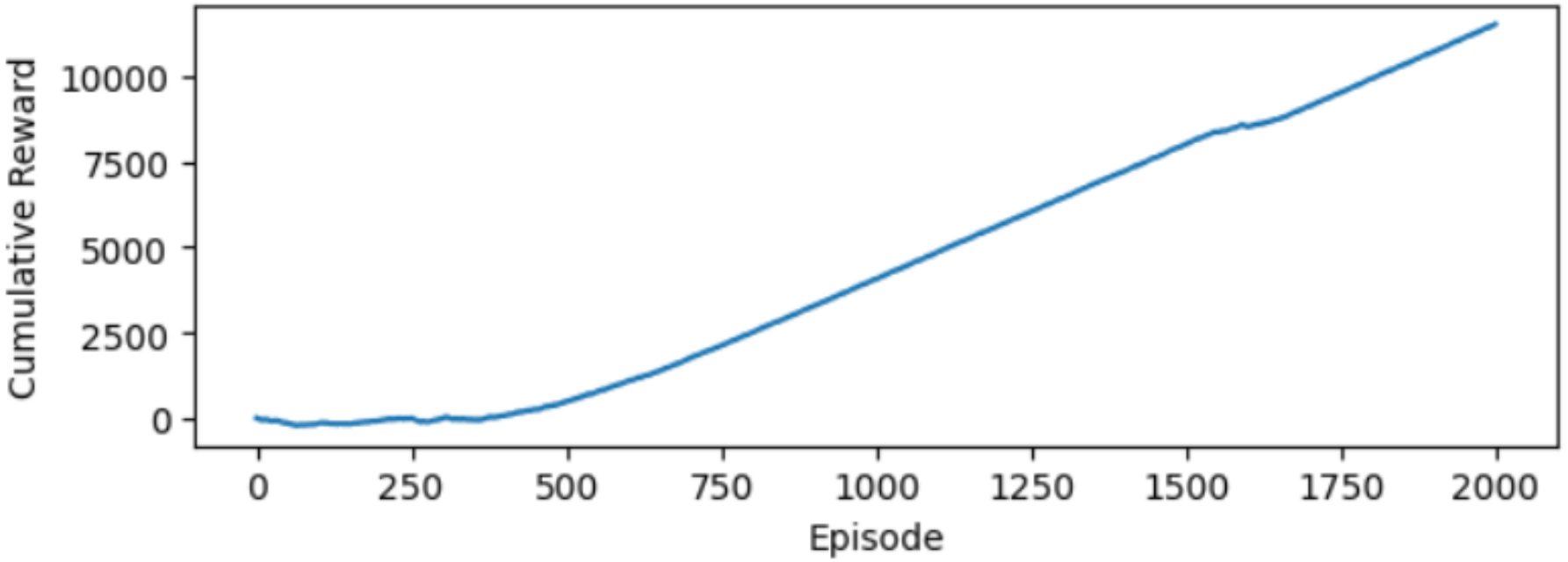
# Post-training Evaluation

- Display some training curves, showing for instance the evolution of the win rate (ability to reach the exit every time) over training episodes.



# Post-training Evaluation

- Display some training curves, showing an increase of cumulative rewards over training episodes (hence indicating that the agent is attempting/learning to maximize rewards).



# Post-training Evaluation

Test the model

- For instance, you can ask the model to make an exploitation move in specific locations of the maze and confirm that the agent has indeed learnt to find the shortest path to the exit.

```
# Test predictions
# Expected: Best move for cell (0, 0) is down, which is index 3
pos1 = np.array((0, 0))
print(model.predict(pos1))
# Expected: Best move for cell (1, 2) is right, which is index 1
pos2 = np.array((1, 2))
print(model.predict(pos2))
# Expected: Best move for cell (3, 2) is down, which is index 3
pos3 = np.array((3, 2))
print(model.predict(pos3))
```

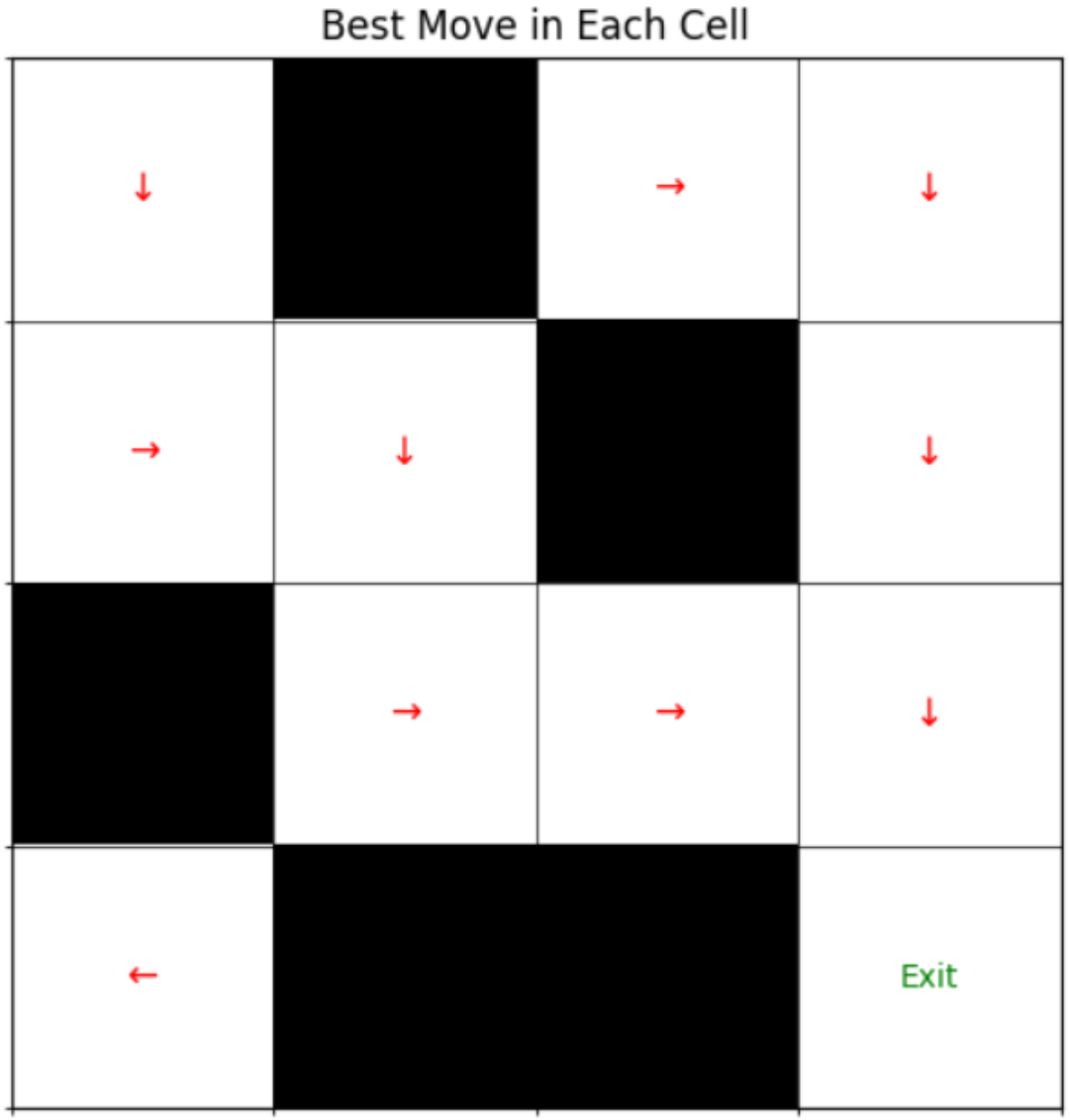
```
3
1
3
```

# Post-training Evaluation

Test the model

- For instance, you can ask the model to make an exploitation move in specific locations of the maze and confirm that the agent has indeed learnt to find the shortest path to the exit.

```
# Visualize best moves in each cell after training
maze.visualize_best_moves(model)
```

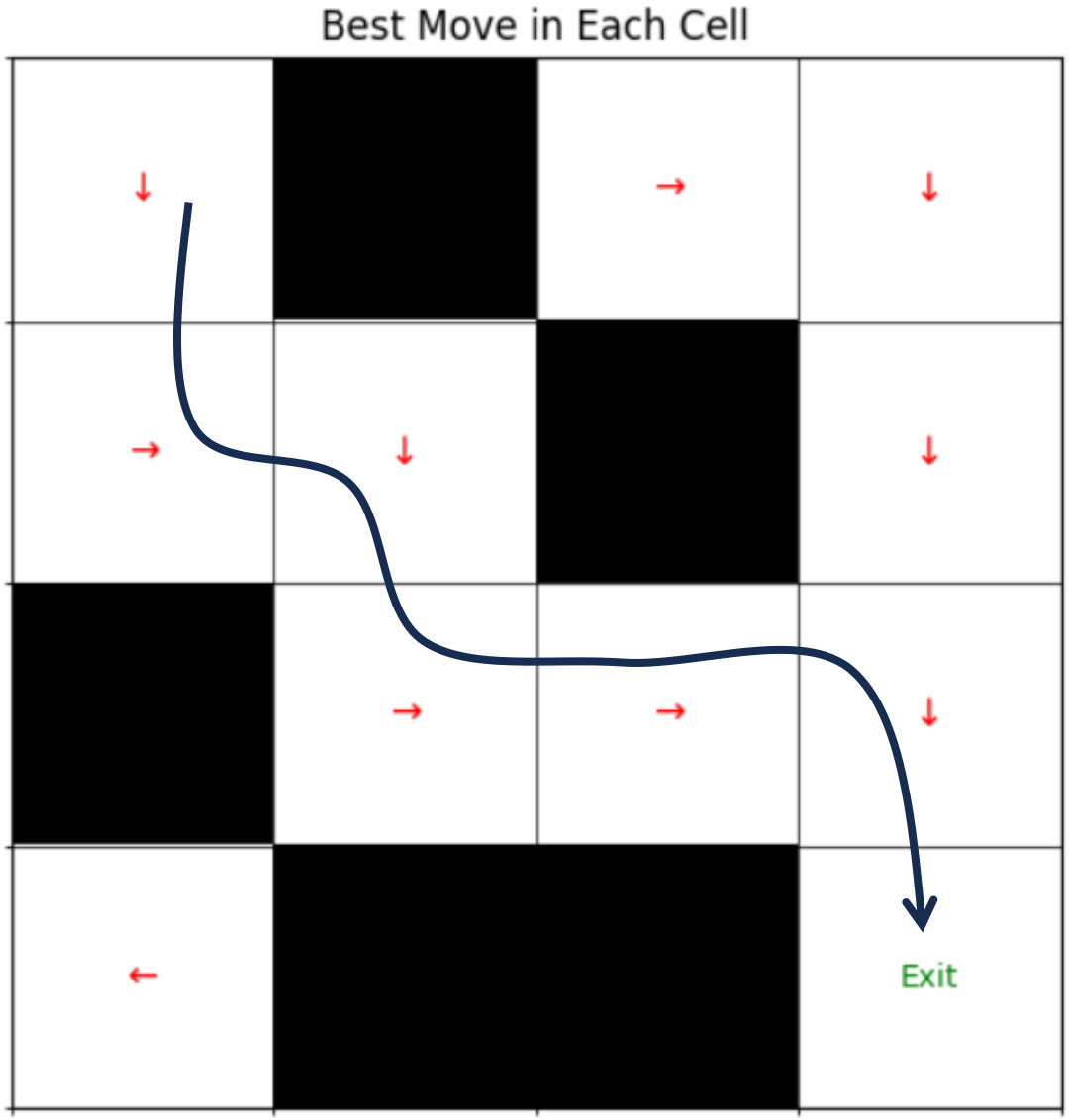


# Post-training Evaluation

- There are however cells that are still problematic, e.g. cell (0, 3).
- Not the end of the world given that this cell is impossible to reach from start cell (0, 0).
- This suggests that the Deep-Q-Learning agent has more to learn about the maze game.

**Yet, it was able to learn the shortest path from (0, 0) to (3, 3)!**

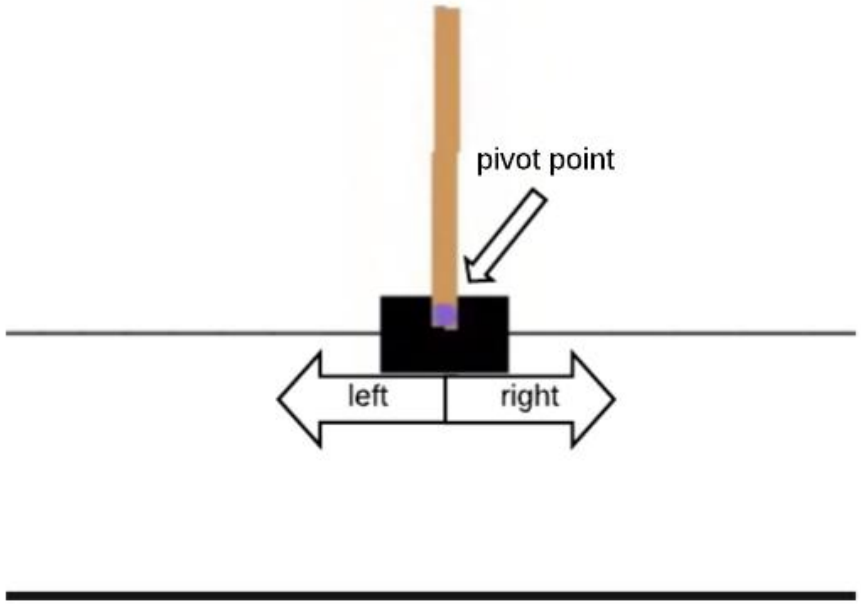
```
# Visualize best moves in each cell after training
maze.visualize_best_moves(model)
```



# A more difficult example (shown, but implementation out of the scope of this class)

**Example:** the cart-pole problem, a.k.a. learning to balance objects.

- **State:** our current visualization of the cart (**image** that the RL model has to process).
- **Actions:** 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



# A more difficult example (shown, but implementation out of the scope of this class)

- **Here:** 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.
- It takes an image (representing the current cart situation) as an input, instead of the coordinates of the maze earlier!
- Learn to observe and play the game correctly! (coordinate visual and motor skills).

```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))
```

# A more difficult example (shown, but implementation out of the scope of this class)

Same principle as before with the **Deep Q-learning** procedure as before, but more complex due to:

- Image processing
- Physics-based environment
- Need for experience replay (need to see the full episode for one instance of training)

```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))
```

# Training results

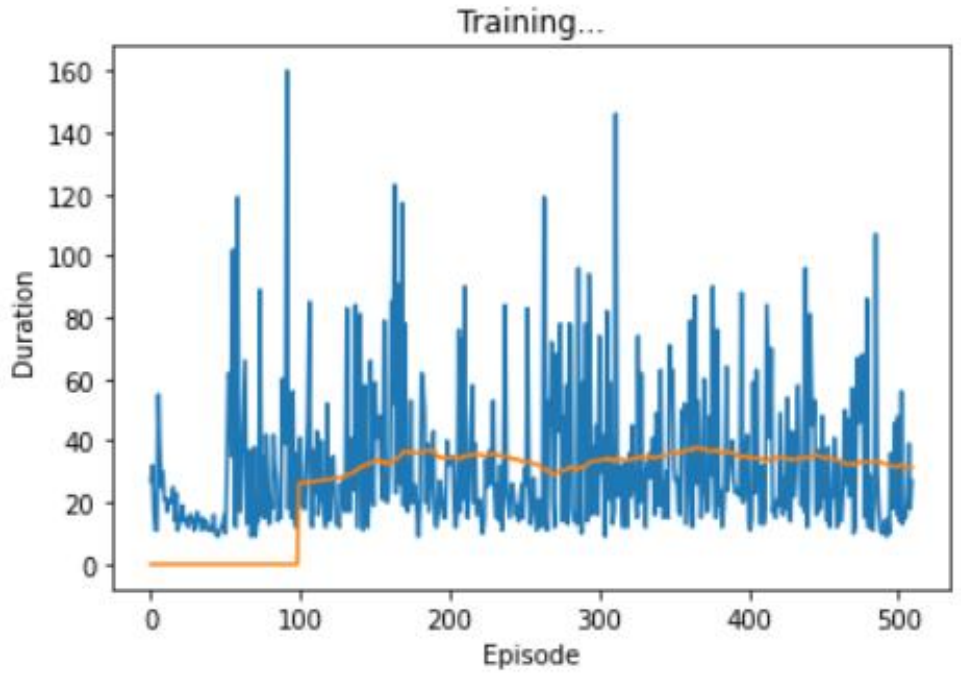
Same principle as before with the **Deep Q-learning** procedure as before, but more complex due to:

- Image processing
- Physics-based environment
- Need for experience replay

**But agent will eventually learn to keep the stick balanced for a solid 30 seconds on average!**

```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())

```

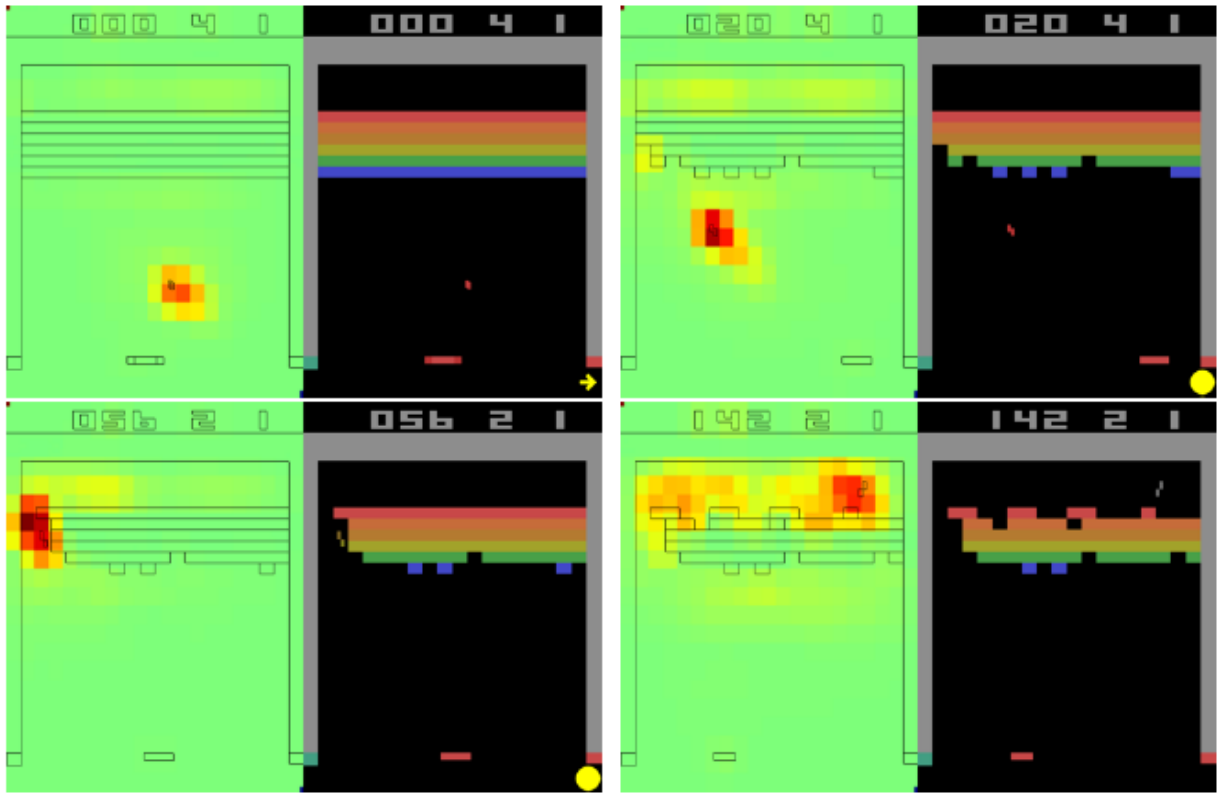


# Applications following this cart-pole example

- 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/

# Applications following this cart-pole example

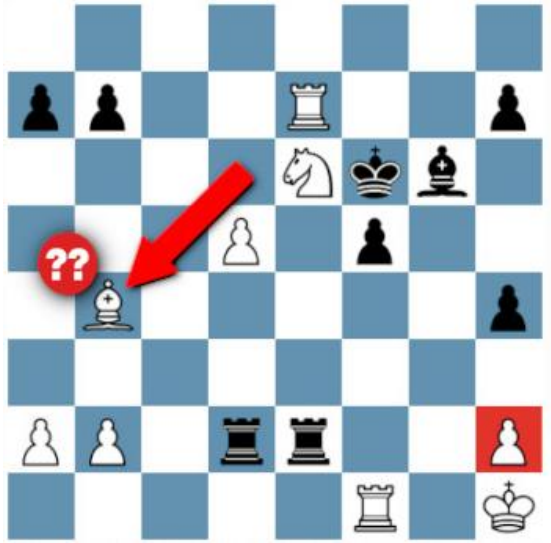
- Train an AI to play video games with Deep Reinforcement Learning (Mnih, 2013)!
- Paper: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
- Video: https://www.youtube.com/watch?v=TmPfTpjtdgg



# Let us start with a problem

**Problem:** Traditional RL relies on hand-crafted rewards… But in many real-world tasks:

- Reward function is often hard to define (chess, language, etc.)
- Human preferences matter more (e.g., dialogue, summarization).

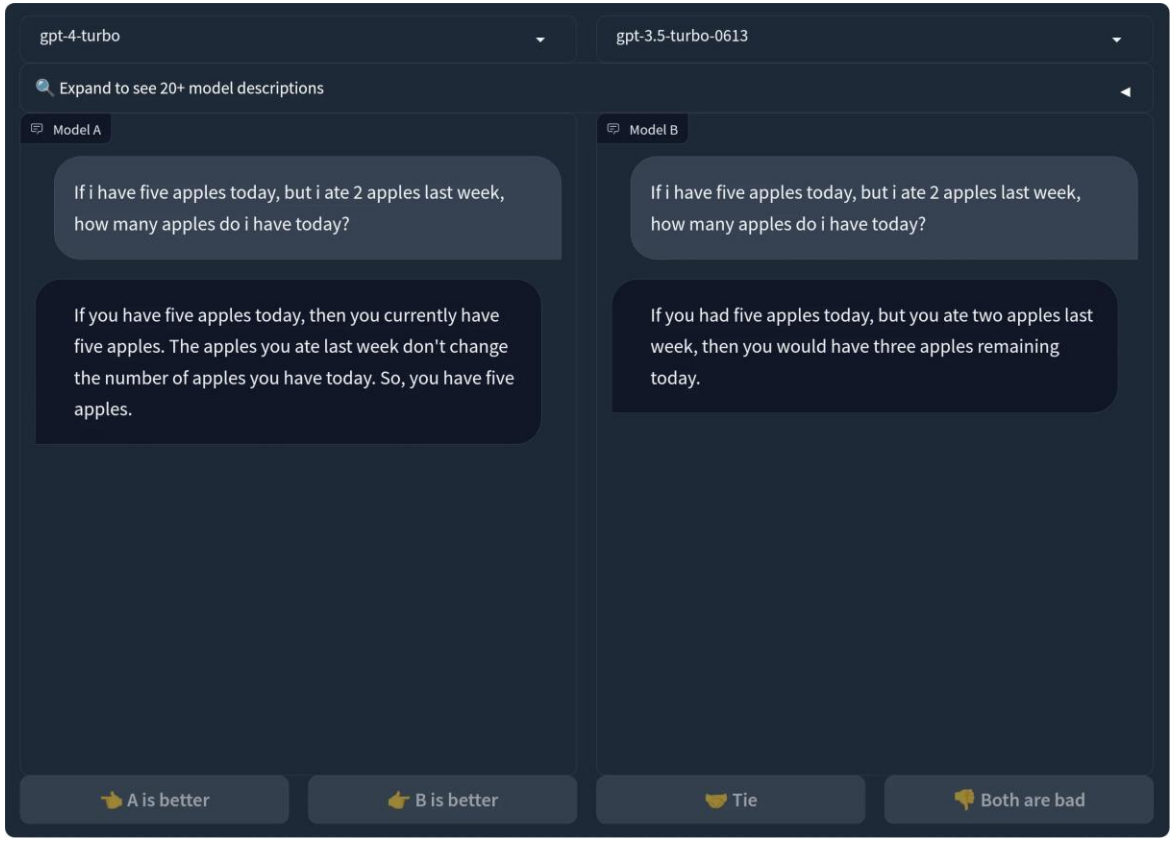



For instance, how would you estimate that a given chess move is brilliant?
Or a blunder?
Is there a reliable close-form expression for this?

# Let us start with a problem

**Problem:** Traditional RL relies on hand-crafted rewards… But in many real-world tasks:

- Reward function is often hard to define (chess, language, etc.)
- Human preferences matter more (e.g., dialogue, summarization).



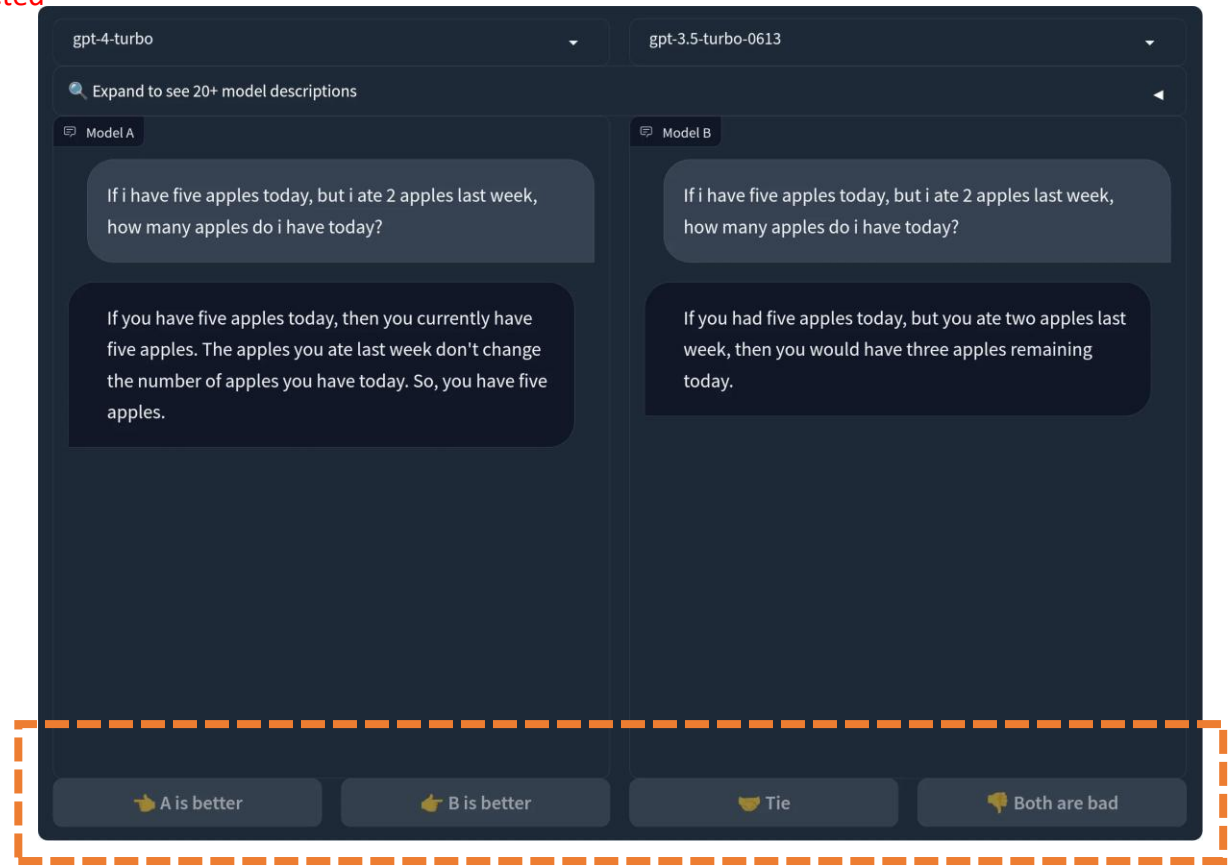

What closed-form metric can you use to evaluate the quality of a chatbot answer?

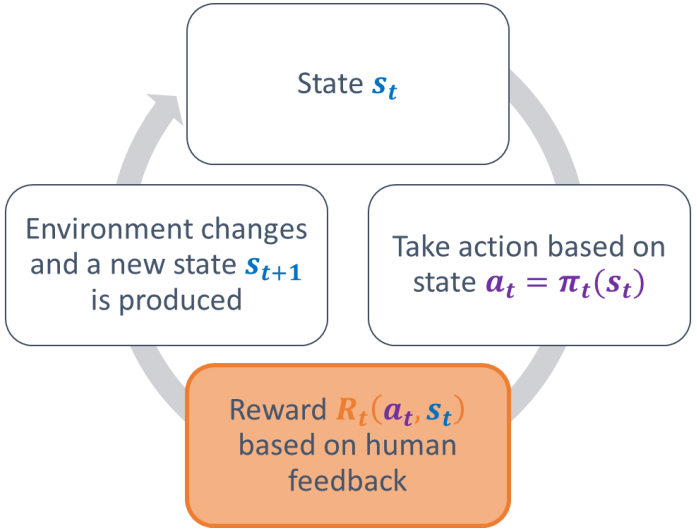
# RLHF: a definition

**Definition (Reinforcement Learning with Human Feedback):**

**Reinforcement Learning with Human Feedback (RLHF)** enables learning from human judgment instead of static reward functions.

In layman terms, it replaces the traditional closed-form reward function with **human-in-the-loop evaluation**, which the agent will then use as a reward/feedback during training.



# Our RLHF toy example

**Our toy example is implemented in Notebook 7.**

- **Objective:** Let us imagine that we are building a chatbot that needs to respond to user queries intelligently.
- **State:** Think of this as a chat simulation where a user greets the chatbot first before asking a question. The embedding of the user query play the role of a state that the agent model receives as input.
- *(Here this will not matter as we will simply replace the user query by a random embedding, but let us pretend it represents the user query, but that could be changed as well. Leaving it to you as extra practice.)*

# Our RLHF toy example

- **Objective:** Let us imagine that we are building a chatbot that needs to respond to user queries intelligently.
- **Actions:** The chatbot then perceives the question as an embedding, which is a vector representation of the query. The chatbot then has four possible responses to choose from to greet the user.
  - “Hello, how can I help you?“
  - “What do you need?“
  - “Please clarify your question.“
  - “Hello lovely.“
- *(Note that in practice we would allow the model to generate answers, in the form of a sequence of some sort, but we keep it simple here.)*

# Our RLHF toy example

- **Reward:** However, instead of training it on massive datasets, we decided to simulate an environment where the chatbot learns by trial and error, relying on human feedback to understand the quality of its answers, just like an agent in a reinforcement learning game.
- **Simulated human feedback:** it would be cumbersome to have humans (you) rate the quality of the answers over and over again, so we will implement a simulated feedback engine that gives assigned score to each possible action.
  - "Hello, how can I help you?" (Score: 10)
  - "What do you need?" (Score: 5)
  - "Please clarify your question." (Score: 2)
  - "Hello lovely." (Score: 1)

# Our RLHF toy example

- Each response has a predefined feedback score, representing how good the answer is. The higher the score, the better the response. Let us pretend that this score is given by a user, after this user has been prompted to evaluate the quality of the chatbot answer, using a satisfaction score between 0 and 10.
- This environment allows an AI agent to practice choosing responses, receiving feedback from humans in the form of a score, and learning from its mistakes.
- Over time, the chatbot could use reinforcement learning techniques (like deep Q-learning) to improve its ability to pick the best response based on the query embeddings.

# A simple chatbot environment

We start by defining a way to:

- **Generate random embeddings for user queries:** we will simply pretend that these represent the queries of a user.
- **Simulate human feedback to each possible answer:** This simply draws values from a dictionary, but would have to be replaced by actual human feedback to make this RLHF.

```python
# Simulate environment with query embeddings and predefined responses
class SimpleEnvWithFixedResponses:
    def __init__(self, embedding_dim):
        self.embedding_dim = embedding_dim
        # Predefined responses and their feedback scores
        self.responses = {"Hello, how can I help you?": 10,
                          "What do you need?": 5,
                          "Please clarify your question.": 2,
                          "Hello lovely.": 1}
        # Response options
        self.response_list = list(self.responses.keys())
        # Corresponding feedback scores
        self.feedback_scores = list(self.responses.values())

    def get_query_embedding(self):
        # Simulate a random query embedding vector
        return torch.randn(self.embedding_dim)

    def simulate_human_feedback(self, action):
        # Return the feedback score for the chosen response
        return self.feedback_scores[action]
```

# A simple chatbot environment

Next, we define our RL agent.

- It will be relying on a simple model that receives the embedding for the user query,
- And then chooses the best answer to produce out of the four possible answers.
- Simple linear model, as we are treating this as a classification task of some sort.

```
# Policy network to map query embeddings to response probabilities
class PolicyNetwork(nn.Module):
    def __init__(self, embedding_dim, num_responses):
        super(PolicyNetwork, self).__init__()
        self.fc = nn.Sequential(nn.Linear(embedding_dim, 64),
                                nn.ReLU(),
                                nn.Linear(64, num_responses),
                                nn.Softmax(dim = -1))

    def forward(self, query_embedding):
        return self.fc(query_embedding)
```

# A simple chatbot environment

Next, we define our RL agent.

- As before, our agent will go through steps of exploration and exploitation, following a policy similar to $\epsilon$-greedy.
- It uses value for the exploration rate $\epsilon$ slowly decaying from 1 to 0 over time.
- Define its hyperparameters, just like before.

```python
# Policy network to map query embeddings to response probabilities
class PolicyNetwork(nn.Module):
    def __init__(self, embedding_dim, num_responses):
        super(PolicyNetwork, self).__init__()
        self.fc = nn.Sequential(nn.Linear(embedding_dim, 64),
                                nn.ReLU(),
                                nn.Linear(64, num_responses),
                                nn.Softmax(dim = -1))

    def forward(self, query_embedding):
        return self.fc(query_embedding)
```

```python
# Proximal Policy Optimization (PPO) agent
class RlhfAgent:
    def __init__(self, policy_net, lr = 1e-3, gamma = 0.99, clip_epsilon = 0.2):
        self.policy_net = policy_net
        self.optimizer = optim.Adam(policy_net.parameters(), lr = lr)
        self.gamma = gamma
        self.clip_epsilon = clip_epsilon
```

# Training the RL agent model

**How to define a loss function for training this model?**

- Instead of Q-learning, we want to optimize the policy directly.
- We want a loss function that increases the likelihood of our RL agent model generating high-reward answers.
- The reward $R$ for each answer comes from the simulated human feedback function.

Here, we will use the **REINFORCE algorithm.**

# Training the RL agent model

**Definition (REINFORCE algorithm):**

**REINFORCE** is a **policy gradient method** used in reinforcement learning to directly learn the policy function that maps states to action probabilities.

Q-learning algorithms are **value-based** methods, meaning that they would learn how good an action is in terms of Q-values and attempts to choose actions based on these.

**Problem:** Unfortunately, these Q-values cannot be easily calculated here, as we are relying on human feedback instead of a closed-form reward.

**REINFORCE**, on the other hand, is a different approach, and a **policy-based** method. This means that it directly learns the policy $\pi(s \mid a)$, i.e. the probability of taking actions, using neural networks.

# REINFORCE Algorithm Breakdown

**REINFORCE goes through the following steps**

- **Step 1 - Collect Rollouts:** Run the current policy in the environment and collect history (states, actions taken, rewards, and log probabilities of actions) on each played episode.

# REINFORCE Algorithm Breakdown

- **Step 2 - Use rewards to estimate advantages:** How much better was this action compared to what we have seen until now?

The **advantage** checks how using action $a$ in state $s$ during this episode performed compared to our previous episodes.

It is defined as $A(s,a) = R(s,a) - M$, where $R(s,a)$ is the reward obtained when using action $a$ in state $s$ during this episode and $M$ is the average rewards we have obtained on past episodes.

**Intuition:** If $A > 0$ (resp. $A < 0$), then action performed better than expected, retrain model to increase (resp. decrease) the probability of this action to be used in this state in the future.

# REINFORCE Algorithm Breakdown

- **Step 3 – Compute a surrogate loss $L$, that helps the RL model learn a better policy based on the advantages computed.**

$$L = -1 \times \mathbb{E}\left[\log\left(\pi(a \mid s)\right)\ A(s,a)\right]$$

Note that $\pi(a \mid s)$ corresponds to a certain output from our RL model.

If $A(s,a) > 0$, then using gradient descent to minimize this loss $L$ will increase the probability of that action $a$ to be used in state $s$.

Conversely, if $A(s,a) < 0$, we decrease its probability.

*(Can you see why?)*

```python
# REINFORCE agent
class RlhfAgent:
    def __init__(self, policy_net, lr = 1e-3):
        self.policy_net = policy_net
        self.optimizer = optim.Adam(policy_net.parameters(), lr = lr)

    def update_policy(self, query_embeddings, actions, rewards):
        """
        Simple REINFORCE update.
        """
        rewards = torch.tensor(rewards, dtype=torch.float32)
        actions = torch.tensor(actions, dtype=torch.int64)
        query_embeddings = torch.stack(query_embeddings)

        # Get action probabilities and compute log-probs
        action_probs = self.policy_net(query_embeddings)
        dist = torch.distributions.Categorical(probs=action_probs)
        log_probs = dist.log_prob(actions)

        # Estimate advantages (just normalized rewards for now)
        advantages = rewards - rewards.mean()

        # Policy loss: negative log-prob times advantage
        loss = -(log_probs * advantages).mean()

        # Update
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Define agent, by providing the model used to calculate policies.

One update requires to know the query embeddings (state), action and reward obtained on this episode.

Compute the log probabilities, advantages and the loss function.

Backpropagate with Adam just like before.

# Training the RL agent

With everything set up (the chat environment, the policy network, and the REINFORCE agent), we can now train our chatbot using Reinforcement Learning with Human Feedback (RLHF).

- Our trainer function uses 500 training episodes, where the chatbot repeatedly receives query embeddings, selects responses based on its policy, and gets human-like feedback. Each action probability is stored to compare past and new policies.
- Every 10 interactions, the REINFORCE algorithm updates the policy model, refining how it maps queries to responses. Over time, the chatbot improves its response selection, in an attempt of maximizing the rewards it obtains.

```python
# Training RLHF with query embeddings and fixed responses
def train_rlhf_with_fixed_responses():
    # Dimensionality of query embeddings
    embedding_dim = 8
    # Number of possible responses
    num_responses = 4
    env = SimpleEnvWithFixedResponses(embedding_dim)
    policy_net = PolicyNetwork(embedding_dim, num_responses)
    agent = RlhfAgent(policy_net)
    # Preparing training parameters
    num_epochs = 500
    rewards_over_time = []

    # Training loop
    for epoch in range(num_epochs):
        query_embeddings, actions, rewards = [], [], []
        # Generate training data for one epoch
        for _ in range(10):
            # Simulate 10 queries interactions per epoch
            query_embedding = env.get_query_embedding()
            probs = policy_net(query_embedding)
            action = torch.multinomial(probs, num_samples = 1).item()
            feedback = env.simulate_human_feedback(action)
            query_embeddings.append(query_embedding)
            actions.append(action)
            rewards.append(feedback)
        # Update the policy using REINFORCE formula
        agent.update_policy(query_embeddings, actions, rewards)
        # Record average reward
        avg_reward = np.mean(rewards)
        rewards_over_time.append(avg_reward)
        print(f"Epoch {epoch + 1}/{num_epochs}: Avg Reward = {avg_reward:.2f}")
    return rewards_over_time
```

Define environment, agent, models and training parameters

Play the game 10 times over 500 iterations (10 episodes per iteration, one update every 10 episodes)

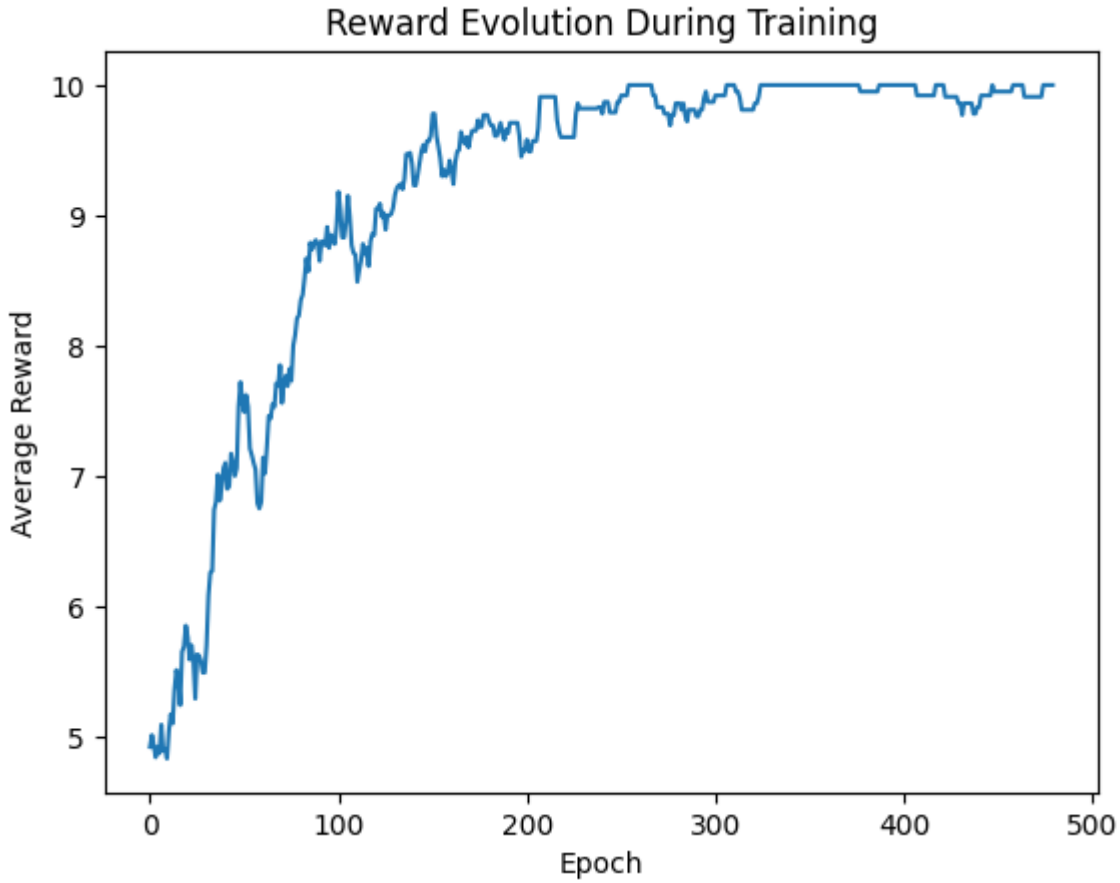
Log the outcomes of each episode (states, actions, rewards)

Update model using REINFORCE process

Display

# It trains!

- The model is able to train!
- It learns to choose the correct action (i.e. the one answer) that maximizes the possible reward (which is 10).
- In practice, however, we would replace the automated reward system, with a human grading one, but it is costly in terms of human resources!



# Applications of RLHF

- **Conversational AI and Chatbots:** Fine-tunes language models like ChatGPT to align with human preferences in terms of answers.
- **Game AI and Virtual Assistants:** Builds non-player characters (NPCs) or agents that adapt to human behaviour and feedback.
- **Ethical Decision-Making in AI:** Applies human values to scenarios with no clear reward functions (e.g., moral dilemmas).

# RLHF for ChatGPT fine-tuning? (Step 2)

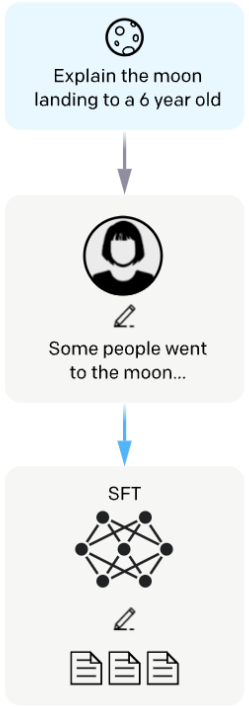
Step 1

**Collect demonstration data, and train a supervised policy.**

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



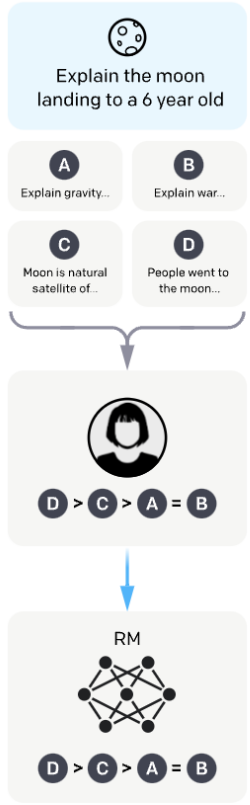

Step 2

**Collect comparison data, and train a reward model.**

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.




Step 3

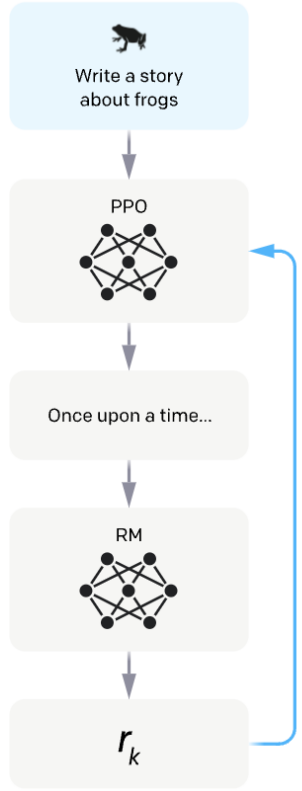
**Optimize a policy against the reward model using reinforcement learning.**

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



# 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 or rewards computed easily.

- **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):**

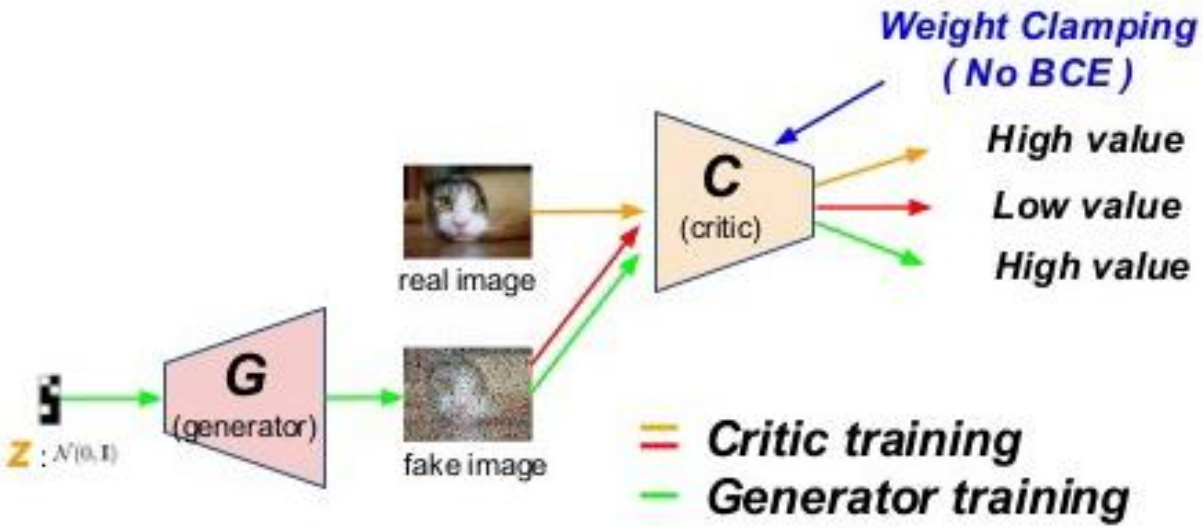
Actor-critic algorithms consist of two components.

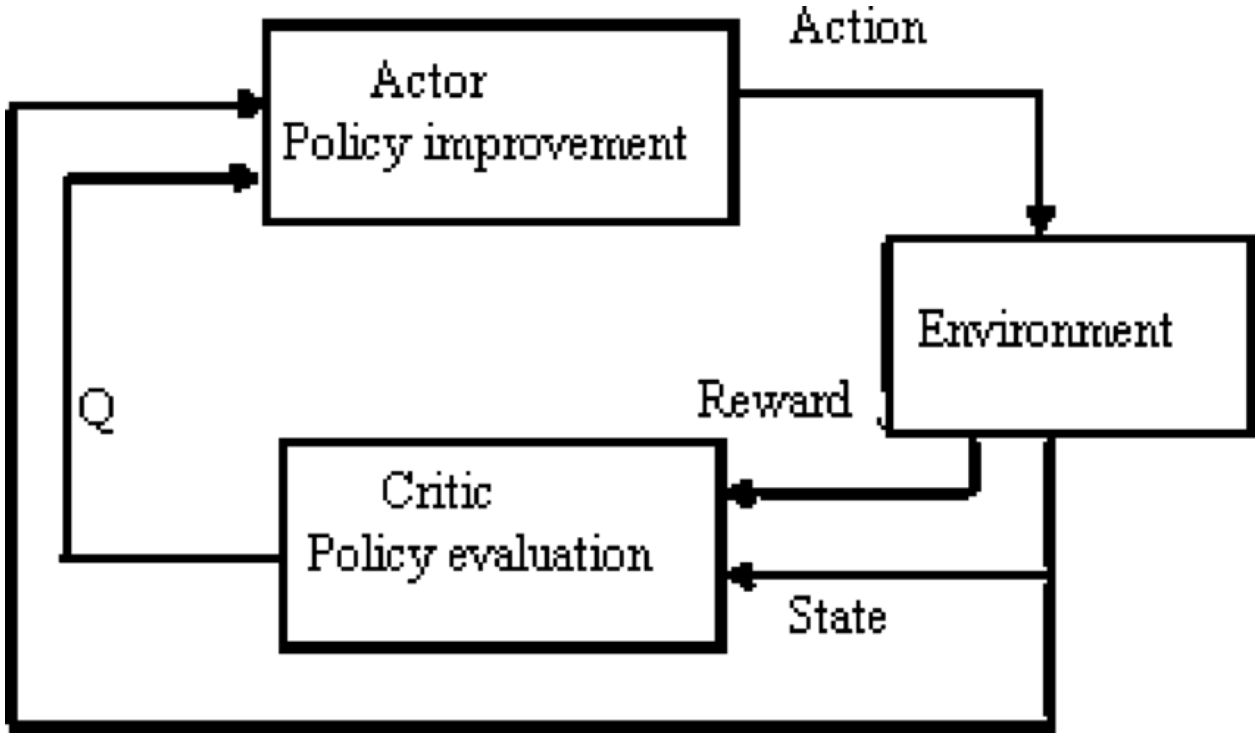
- **Actor:** a NN, whose purpose is to produce actions in response to given states, i.e. a policy. Can be trained as in Deep $Q$-learning or REINFORCE.
- **Critic:** a NN, whose purpose is to evaluate the quality of the selected actions and suggesting directions for improvement, by defining a reward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



# Actor-critic for ChatGPT fine-tuning? (Step 3)

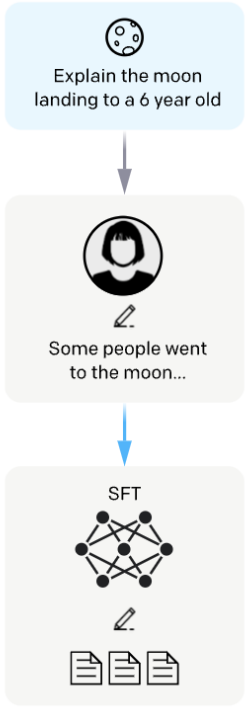
Step 1

**Collect demonstration data, and train a supervised policy.**

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



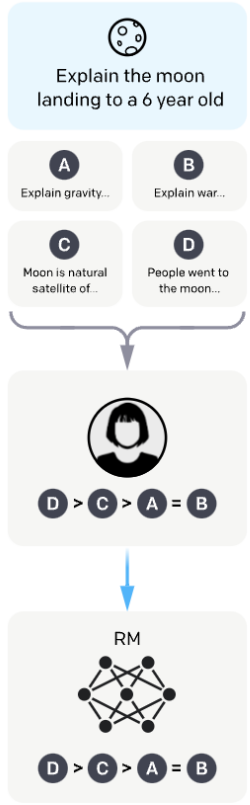

Step 2

**Collect comparison data, and train a reward model.**

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.




Step 3

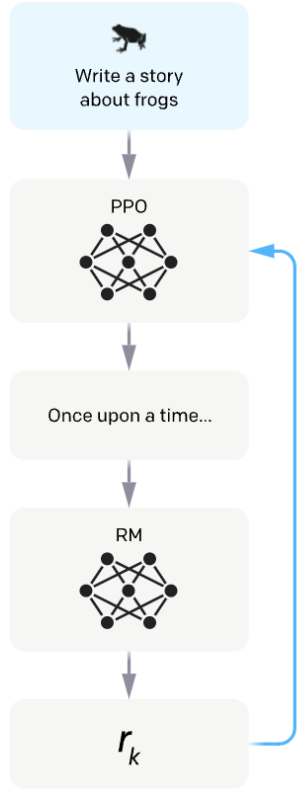
**Optimize a policy against the reward model using reinforcement learning.**

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



# 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.

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

$$V_t^{\pi}(s) = E[R_t + \gamma G_{t+1} \mid s_t = s]$$

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

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

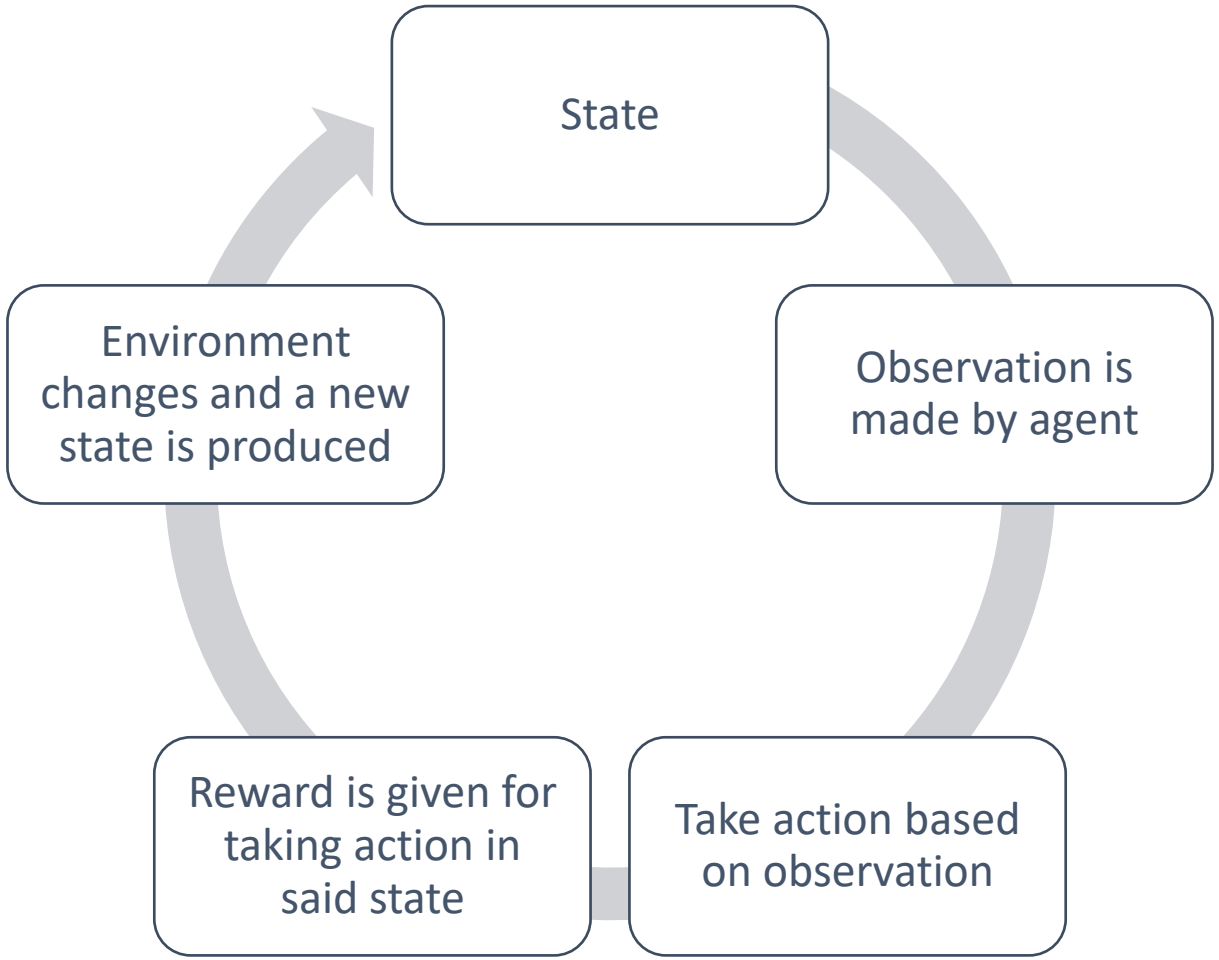
# Partially observable MDP (more advanced stuff, out of scope)

**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.



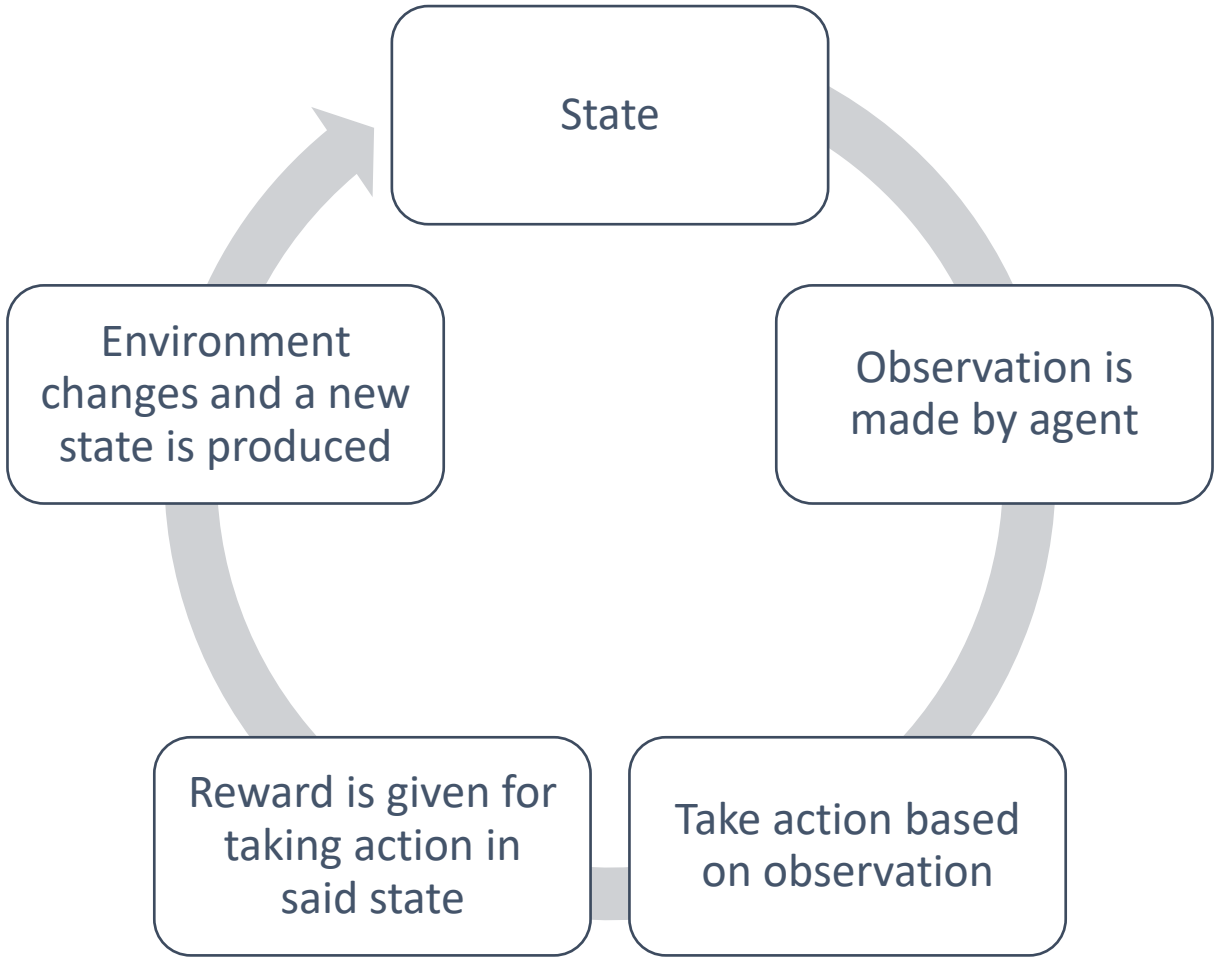
# Partially observable MDP (more advanced stuff, out of scope)

**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.
hid

ation is
y agent

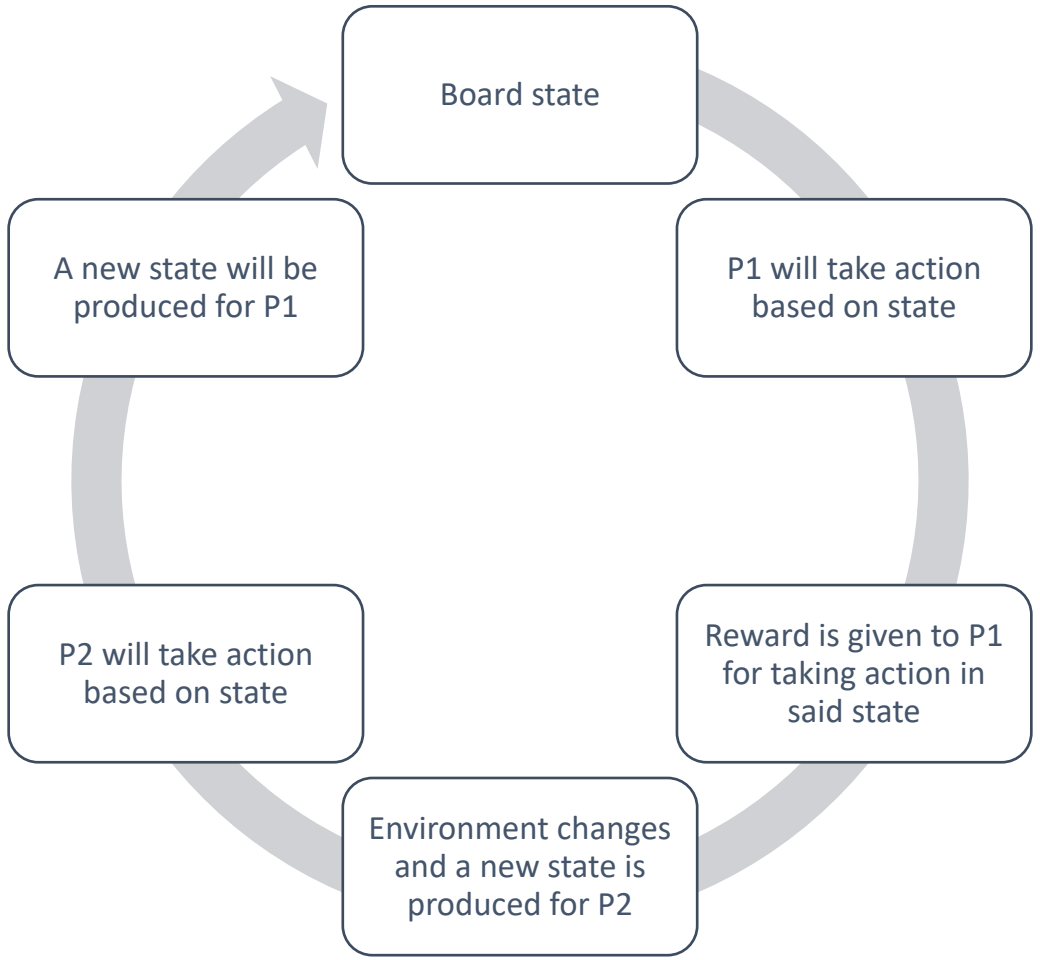
d

# 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.

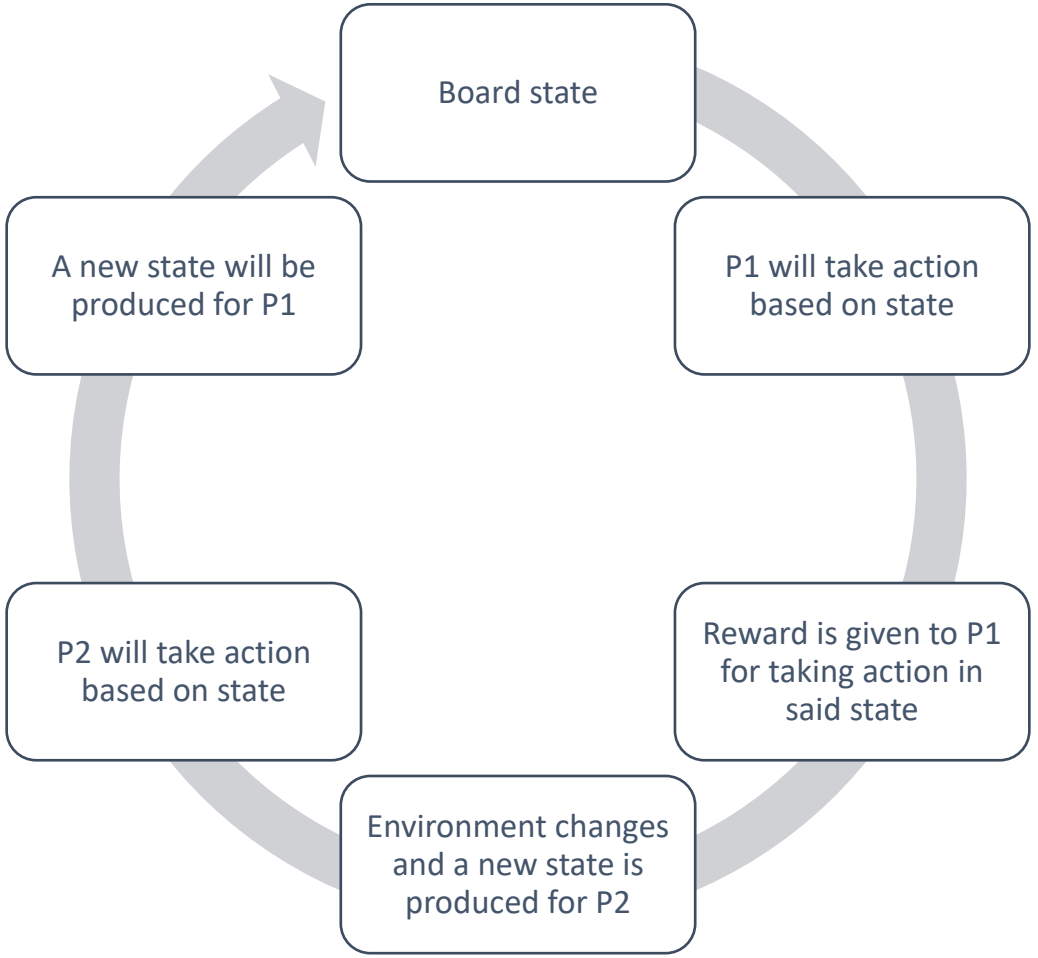
# State-Action-Reward-State-Action or SARSA (more advanced stuff, out of scope)

- 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 both together?**