REINFORCEMENT LEARNING

Hallvard Høyland Lavik

hallvard.hoyland.lavik@nmbu.no

Motivation

Reinforcement learning is a powerful approach within machine learning where an artificial neural network (hereafter "agent") learns to interact with an environment. These interactions result in either an instantaneous or delayed "reward" which incentivize the agent to learn the optimal actions with respect to different "states" within the environment.

Initially, an agent starts with no knowledge of the environment and must learn through trial and error which actions lead to positive rewards. This learning process involves adjusting the weights of the agent over many iterations to improve performance. In cases where the rewards are significantly delayed, this learning process may be challenging – as the agent is unable to determine which of its actions led to the reward. An example of extremely delayed reward is OpenAIs work on mastering Minecraft. They had a goal for their agent to obtain diamond tools, which "usually takes proficient humans over 20 minutes (24,000 actions)". They achieved this through sequentially rewarding the agent based on the progress it made, thus providing it with a more immediate reward based on its actions. [**10**]

Methods

Reinforcement agents aim to execute actions within an environment that result in the highest rewards. The term "reinforcement" is used because the agent's actions are influenced by its past experiences. Optimal behavior can be achieved through various reinforcement learning techniques, which are typically categorized into two main types: model–free and model–based methods.

MODEL–FREE AND –BASED LEARNING

> Whether an agent is model–free or –based depend on its knowledge and understanding of the environment that is interacted with. Therefore, a model–free agent refers to an agent that has – and gains – no knowledge on the workings of the environment. In other words, the agent only learns to perform optimal actions given the environment, and nothing more. [**11**]
>
> For model–based algorithms, however, the agent either aims to learn the underlying dynamics (*e.g.*, physics) of the environment, or is given prior knowledge with regards to the environment. An example of a model–based agent is the chess–playing *AlphaZero* developed by Google DeepMind ([**12**]) which was given "perfect knowledge of the game rules" and "input planes (i.e. castling, repetition, no–progress) and output planes (how pieces move, promotions [...])".

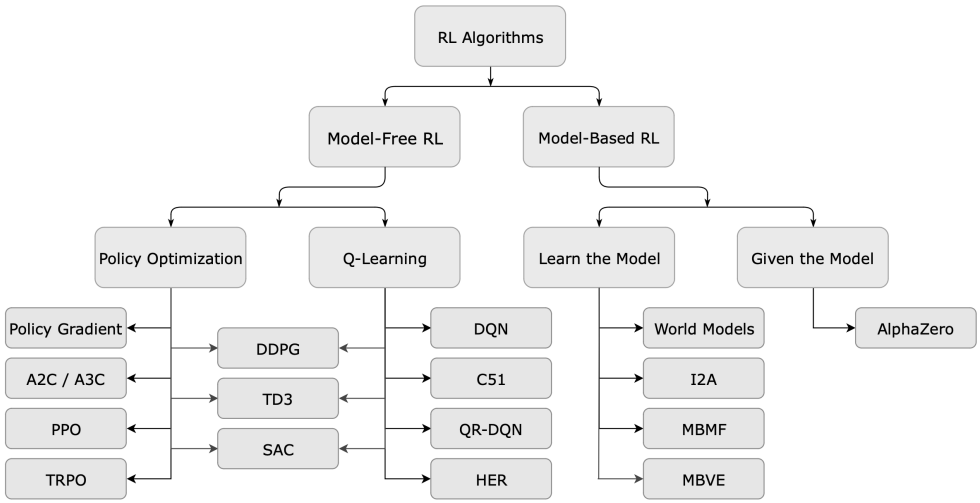

Figure 1: Reinforcement learning algorithms [**11**].

The various algoriths and their category can be seen in **Figure 1**, as presented by OpenAI ([**11**]). In this report, the algorithms **Policy Gradient** and **DQN** is studied.

Model–free and –based agents can be further categorized into either policy–based and value–based approaches, which explain how the agent predicts its actions.

A policy is a strategy that maps states to actions with the aim of maximizing future rewards. It is often represented as a probability distribution over the possible actions. On the other hand, the term "value" in value–based approaches refers to the mapping from states to the expected future rewards; it predicts the future reward for each action given an observed state.

Therefore, an agent can either learn to have confidence in its actions (policy–based) or try to estimate the expected value of being in a state (value–based). While the methods differ slightly, both aim to maximize future rewards.

ON– AND OFF–POLICY LEARNING

Reinforcement learning algorithms can also be categorized into on– and off–policy. The distinction between these two lies in the way the agent's parameters are updated.

On–policy methods, such as policy–based approaches, involve the agent determining actions based on its current policy. The policy is then updated according to the reward received from the chosen action.

Off–policy methods, on the other hand, involve the use of two separate policies: one for exploration and another for learning. An example of this is Q–learning. In Q–learning, the agent's network (Q) and the target network (Q–hat) are independent when calculating the loss, making it an off–policy approach.

POLICY–BASED APPROACH

Denoted as $\pi(a|s)$, the policy of an agent is the certainty of an actions a will yield the highest possible future reward, given a state s. When the agent performs the most probable action, it observes the new state and is given a reward (assuming an immediate reward is given). Therefore, in a policy–based approach, finding the optimal policy (denoted as $\pi^{\star}(a|s)$) which maximizes the expected future rewards given the current state is what the agent leans to do. [**6**]

POLICY–BASED GRADIENT

In a policy–based gradient approach, the performance of the agent is optimized based on the probability distribution of the possible actions given a state with respect to its parameters ($\pi(a|s)_\theta$). This optimization is based on the agent's experience in an on–policy manner, in order to obtain a gradient which then directly influence the parameters. **[8]** While there are various approaches in determining the gradients, a modified version of the REINFORCE ("REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility") algorithm, as presented by Williams in "Simple Statistical Gradient–Following Algorithms for Connectionist Reinforcement Learning" **[16]**, is implemented as:

initialize agent

for game **do**

create empty memory

observe initial state

while alive **do**

forward propagate state through agent to obtain action probabilities

select action based on random weighted choice

execute chosen action

observe next state reward and mortality

store reward and logarithm of chosen action probability in memory

end while

calculate discounted rewards R' based on memory

calculate policy gradient G based on memory and R'

update agent parameters with respect to gradient

end for Modifed from Yoon **[17]**

Where the expected future reward R_i for each time step $i \in [0, N]$ is calculated as:

$$\begin{aligned} R_i &= r_i + \gamma R_{i+1} \qquad 0 < \gamma < 1 \\ R'_i &= (R_i - \mu_R)/\sigma_R \end{aligned} \quad . \tag{1}$$

Here, r_i is the reward at time step i, and γ is the discount factor. The expected future rewards are calculated backwards, as $R_{N+1} = 0 \Rightarrow R_N = r_N$. These expected rewards are discounted, such that the rewards far into the future are worth less than instantaneous rewards. The expected rewards are then standardized.

The policy gradient G_i at each time step i is calculated as:

$$G_i = -\log\left[\pi(a_i|s_i)\right] \times R'_i \qquad \Rightarrow \qquad G = \sum_i^N G_i \tag{2}$$

where $\pi(a_i|s_i)$ is the probability of taking the chosen action a_i given state s_i (*i.e.*, $\max_a \pi(a|s_i)$). The overall gradient, G, is the sum of these individual gradients G_i.

VALUE–BASED APPROACH

In contrast to the **policy–based gradient approach** which outputs a probability distribution across the possible actions, the value–based approach predicts a Q-value (representing the expected future rewards) related to each possible action. **[7]**

Q–LEARNING

In order to learn optimal policy in an environment, a Q-learning agent learns to associate each state with expected rewards for each possible actions. The expected values for the states can thus be calculated through the state-value function (*i.e.*, the Bellman equation):

$$\begin{aligned} V_\pi(s) &= E_\pi \left[R'_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s \right] \\ &= R'_t + \gamma \sum_{i=t+2}^{T} R'_i \end{aligned} \quad . \tag{3}$$

Here, $V_\pi(s)$ represents the value of state s under policy π, E_π the expected: immediate reward R'_{t+1} (see Equation (1)) plus the discount factor γ times the value of the next state $V_\pi(S_{t+1})$. This equals the expected future rewards from t, given that the agent is at state s. **[4] [2] [15]**

The Bellman Optimality Equation for the Q–value (action–value function) can thus be derived from the state–value function (3):

$$Q(S_t, A_t) = E\left[R_{t+1} + \gamma \times \max_a Q(S_{t+1}, a) \right] \tag{4}$$

which yields the optimal Q–value for the given state and action following the optimal policy thereafter. This equation is the sum of the immediate reward and the discounted maximum expected reward for the next state, and therefore represents the value (*i.e.*, reward) of taking the action. **[1] [9] [15]**

Equation (4) can thus be rewritten to encapsulate the Q-learning update rule:

$$Q(S_t, A_t) = Q(S_{t-1}, A_{t-1}) + \alpha \left[R_{t+1} + \gamma \times \max_a Q(S_t, a) - Q(S_{t-1}, A_{t-1}) \right], \tag{5}$$

where α is the learning rate of the agent **[5] [9]**. This equation updates the Q-value for the state-action pair (S_{t-1}, A_{t-1}) based on the immediate reward R_{t+1} and the maximum Q-value for the next state S_t, discounted by the factor γ.

In traditional Q–learning, these values are stored in a table which thus contains all optimal actions for each given state. However, for environments with many and/or continuous actions and states this approach is not as efficient.

DEEP Q–LEARNING

In deep Q–learning, the optimal Q–value is approximated through the agent instead of obtaining it from the Q-table – therefore the Q-table can be discarded, as all information is stored within the agent parameters.

The parameters of the agent are updated in an off–policy manner with respect to a loss which is the difference between the predicted Q–value and a target Q–value, performing gradient descent with respect to this. **[5]** Equation (5) can therefore be modified and rewritten:

$$\text{loss} = \underbrace{Q(R_t, a|\theta)}_{\text{Q–predicted}} - \underbrace{R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a|\hat{\theta})}_{\text{Q–target}}$$

Here, $Q(R_t, a|\theta)$ is the previously predicted Q-value for taking action a at state R_t given the parameters θ,

R_{t+1} the immediate reward after taking action a at time t, γ is the discount factor, $\hat{Q}(S_{t+1}, a|\hat{\theta})$ is the maximum estimated Q-value for the next state S_{t+1} given the parameters $\hat{\theta}$.

$\hat{Q}$ is known as the target network, and is a copy of Q that is updated every C steps. The reason for not using $\hat{Q} = Q$ is to provide a more slowly changing target, which stabilizes training, as explained by Mnih *et.al.*: "[...] makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ [...] using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_i, making divergence or oscillations much more unlikely." **[9]**

Finally, the parameters θ of the agent are updated through the optimizer with respect to the loss function (*e.g.*, mean squared error). This process of updating the parameters is done iteratively to improve the agent's policy over time.

ϵ–GREEDY EXPLORATION

In deep Q–learning, an epsilon–greedy exploration is a strategy used to force the agent to try out different actions than predicted. In this approach, a random action within the action space $a \in A$ is chosen with a probability of ϵ, and the predicted action chosen otherwise.

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \max_a Q & \text{otherwise} \end{cases} \tag{6}$$

The value of ϵ is typically high at the beginning of the learning–process, and reduced throughout – for instance exponentially decaying by some factor.

initialize agent with replay memory and $\hat{Q}$ as a copy of agent

for game **do**

observe initial state

while alive **do**

forward propagate state through agent to obtain Q–values

select action randomly with probability ϵ otherwise $\max_a Q$

execute chosen action

observe next state reward and mortality

store state, action, new state and reward in agent memory

end while

randomly sample minibatch from memory

calculate discounted rewards R' based on batch

calculate expected and actual Q–values based on batch

update agent parameters with respect to loss and update ϵ

every C steps update $\hat{Q}$ as copy of agent

end for

Slightly modified from Mnih *et.al.* **[9]**

DOUBLE DEEP Q–LEARNING

Due to the nature of the loss calculations, which incorporates maximizing the expected Q–value, the algorithm overestimates slightly. This, in turn, may lead to convergence issues, as the target values tend to have some variance (even though γ tries to counter this): "The max operator in standard Q–learning and

DQN [...] uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates", as written by Hasselt, Guez and Silver [**3**]. In this paper *ibid.*, they proposed using two networks – one representing the target value, and the other being the agent. These networks are then separately updated, and *ibid.* showed that the agent then converged better – opposed to using a **single network**. [**3**]

Architecture

During implementation of a reinforcement learning agent, one has to take into account the possible observation and action spaces the agent is hard-coded to represent. Therefore, this section is divided into multiple subsections representing the different possible input-dimensions.

ONE–DIMENSIONAL OBSERVATION SPACE

Representing a finite number of observable values – as seen in the **cart-pole environment**.

Thus, the agent architecture consists of fully connected layers in a feed–forward manner. Here, the number of layers, nodes and activation functions may be chosen freely.

The number of nodes in the final layer corresponds to the action space.

TWO–DIMENSIONAL OBSERVATION SPACE IMAGE INPUT

Representing a continuous observation space of set width and height – as seen in the **Tetris environment**.

For image–input, convolutional layers are needed. Therefore, the architecture contains; firstly convolutional layers and thereafter fully connected layer(s). The number of convolutional– and fully connected layers depends on the environment and generalizability of the agent, and may be chosen freely – along with the respective hyper–parameters and activation functions.

The number of nodes in the final layer corresponds to the action space.

Implementation

When training the agent, one has to determine how to calculate the gradient/loss. This can either be a generalized or tailored approach, depending on the use–case of the agent. In the letter by Mnih *et.al.* they chose an approach that generalized well across tasks; "a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks", as well as presumably using a different agent architecture [**9**].

POLICY–BASED GRADIENT IMPLEMENTATION

The agent's parameters are initialized as random values by default. These parameters are then updated once for every game the agent plays (*i.e.*, one full interaction with the environment). Therefore, it is necessary to store the behaviour and corresponding rewards of the agent. Instead of saving the predicted actions, the logarithm of the probability of the chosen action is saved. The reason for this is that the logarithm of the agent's confidence is needed for calculating the policy gradient.

Obtaining the action and the logarithm of its probability is achieved by adding a wrapper to the forward pass given a state:

```
def action(self, state):
    actions = torch.softmax(self(state), dim=-1)

    action = np.random.choice(range(actions.shape[0]), 1,
                              p=actions.detach().numpy())[0]
    logarithm = torch.log(actions[action])

    return action, logarithm
```

In addition, this wrapper introduces some sense of exploration, by a randomly weighted sample of actions based on the agent's policy.

After every game the agent plays (during the training–loop), its parameters are updated with respect to the policy gradient. In order for the agent to best learn the optimal actions, it is common to evaluate the expected future rewards. Then, the agent can adjust its predicted action probabilities (policy) so that this expected reward is maximized. See the **approach** for mathematical equivalent formulas and psuedocode, or the **code** for the full implementation.

The expected reward given an action is the sum of all future (discounted) rewards. This is achieved by reversely adding the observed reward and the discounted cumulative future rewards. The rewards are then standardized.

```
rewards = torch.tensor(self.memory["reward"], dtype=torch.float32)

_reward = 0
for i in reversed(range(len(rewards))):
    _reward = _reward * self.discount + rewards[i]
    rewards[i] = _reward
rewards = (rewards - rewards.mean()) / (rewards.std() + 1e-7)
```

The policy gradient is the gradient of the expected reward with respect to the action taken. This is computed by multiplying the logarithm of the selected action probability with the standardized expected reward — previously calculated. The overall gradient is then the sum of all these products.

```
gradient = torch.zeros_like(rewards)
for i, (logarithm, reward) in enumerate(zip(self.logarithms, rewards)):
    gradient[i] = -logarithm * reward
gradient = gradient.sum()
```

A chosen optimizer is then used to back–propagate and update the agent's parameters using the given gradient. [**17**]

VALUE–BASED Q–LEARNING IMPLEMENTATION

Similarly to the policy–based approach, the value–based agent also needs to store its experiences. A given number of random experience sequences are selected every time the agent's parameters are updated respect to the **Q–learning algorithm**. See the **code** for the full implementation.

IMAGE INPUT

If the observation space of the environment is represented by an image, some methods are applied in order to smooth the learning curve.

Firstly, the observed image is preprocessed. This is done through the wrapper:

```
def preprocess(self, state):
    state = state / 255.0
    state = state[:, :, self.shape["height"], self.shape["width"]]
    state = max_pool2d(state, self.shape["max_pool"])

    return state
```

where the input first is normalized, then cropped, and lastly max–pooled. Here, `self.shape["width"]` and `self.shape["height"]` represent `slice`-objects, which crop out the valuable square of information within the `state`, and `self.shape["max_pool"]` the size of the max–pooling.

For the agent to gather the useful information based on the observed states, a wrapper is put around the environments' step function. This is done to disregard excess information, as new information generally takes a few frames in order to become apparent. Therefore, an arbitrary number of frames, `skip`, can be skipped between each observation:

```
def observe(self, environment, states, skip=1):
    act = self.action(states).item()

    done = False
    rewards = 0
    states = torch.zeros(self.shape["reshape"])

    for i in range(0, self.shape["reshape"][1]):
        for _ in range(skip-1):
            _, reward, terminated, truncated, _ = environment.step(0)
            done = (terminated or truncated) if not done else done
            rewards += reward

        _state, reward, terminated, truncated, _ = environment.step(act)
        done = (terminated or truncated) if not done else done
        rewards += reward

        states[0, i] = self.preprocess(_state)

    return action, states, rewards, done
```

As the value–based agent does not output a probability distribution, the maximum output represents the chosen action. In addition, the **epsilon-greedy** approach is used when selecting the action, to force exploration.

```
def action(self, state):
    if np.random.rand() < self.parameter["rate"]:
        action = torch.tensor([np.random.choice(
            next(reversed(self._modules.values()))).out_features
        )], dtype=torch.long, device=self.device)
    else:
        action = self(state).argmax(1)

    return action
```

Like for the policy–based approach, the **discounted rewards** are incorporated – thus nudging the agent to predict the expected future reward for any given state. As the value–based implementation uses a number of experienced sequences (stacked), the optimal Q-value is set to the reward for those steps where the one sequence ends and another begins ("steps" in the code below), as Mnih *et. al.* [**9**] suggested.

```
actual = self(states).gather(1, actions)

optimal = (self.parameter["gamma"]
           * network(new_states).max(1)[0].unsqueeze(1))
optimal = rewards + optimal

for step in steps:
    optimal[step] = rewards[step]

loss = torch.nn.functional.mse_loss(actual, optimal)
```

A chosen optimizer is then used to back–propagate and update the agent's parameters using the given loss, which is calculated as the mean squared error between the actual and optimal Q–values. **[5]**

Cart–pole environment

To gain a basic understanding of how reinforcement learning works one can experiment with a "simple" problem. The cart–pole problem being such, is an environment where the agent has to balance a pole on top of a cart by moving it either to the left or the right — based on the current state of the environment. In this environment, the agent gets immediate reward from its actions.

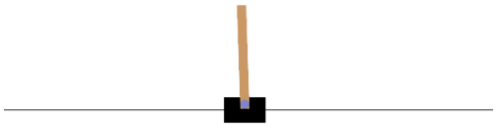

Figure 2: Random observed environment state.

The cart–pole environment (seen in **Figure 2**) is a part of the `gymnasium` package in Python. This package contains a number of different virtual environments which can be imported and used to train and validate ones own reinforcement agents. **[14]** **[13]**

```
import gymnasium as gym

environment = gym.make('CartPole-v1', render_mode="rgb_array")
state, _ = environment.reset()
```

The agent controls the cart movement, *i.e.*, by pushing it one way or another. The termination (or truncation) is determined by the observed values, or if the episode length is greater than 500 time–steps. For every time–step until termination or truncation, the agent is given a reward of +1.

	OBSERVATIONS		**ACTIONS**		**REWARD**
$0 \rightarrow \pm 2.4$	cart position	0	push cart to the left	+1	every time–step
$0 \rightarrow \pm \infty$	cart velocity	1	push cart to the right		(until termination)
$0 \rightarrow \pm 12°$	pole angle				
$0 \rightarrow \pm \infty$	pole angular velocity				

POLICY–BASED GRADIENT AGENT

Initializing the policy–based agent with four inputs and two outputs, with `15` and `30` nodes in the hidden layers, and training it during `4000` games of play and self–improvement led to surprisingly good results. The agent was trained with the `RMSprop` optimizer with a learning–rate of `0.00025` and a reward discount of `0.99`. See **Figure 3** for results.

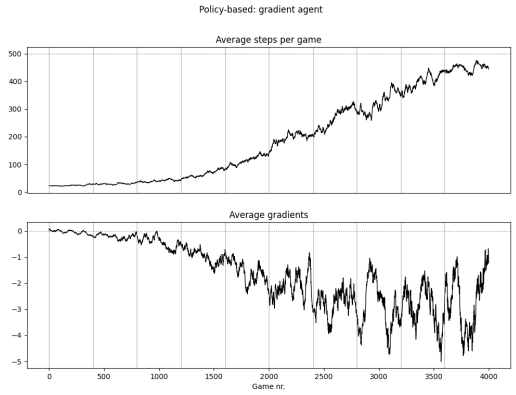

Figure 3: Training of the policy–based agent.

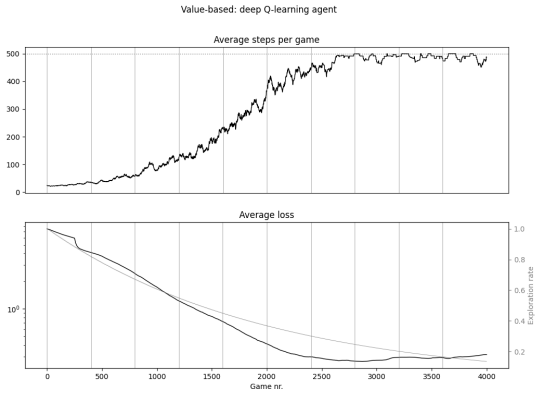

Figure 4: Training of the value–based (DQN) agent.

VALUE–BASED Q–LEARNING AGENT

> Likewise, the value–based agent was initialised with the same hyper-parameters as the policy–based agent, only scaling the learning–rate by a factor of ten due to using mini–batches.
>
> In addition, the Q–learning agent had a γ–value of `0.99` and respectively exploration rate, decay and minimum values of `1`, `0.995` and `0.01`.
>
> The agent was trained during `4000` games using the mentioned **algorithm**. The agent had a memory of 200 games and was updated with `64` game sequences every ten games it played. The target network, $\hat{Q}$ was updated every `250` games. See **Figure 4** for results.

See the **code** for full implementation with examples.

Tetris environment

The Tetris environment (seen in **Figure 5**) is a part of the `gymnasium` [**14**] package in Python. The agent controls the piece movement, *i.e.*, by rotating it or moving it left, right or down, and is given a reward corresponding to the number of rows that is filled (if any). The game ends when the pieces stack up to the top of the playing field.
The agent has to base its actions on the game–screen, and is given no initial knowledge about the game. Thus, the agent has to learn from scratch both how to extract meaningful features based on the game screen, and how to act based on this interpretation.
Therefore, the Tetris environment is significantly different from the **cart–pole** environment, and requires a more intricate agent network.

Figure 5: Random observed (cropped and normalized) environment state.

OBSERVATION		ACTIONS		REWARDS	
$0 \rightarrow 255$	game screen	0	No action	+1	Row filled.
		1	Rotate	+3	Two rows filled.
		2	Right	+8	Three rows filled.
		3	Left	+18	Four rows filled.
		4	Down		

VALUE–BASED Q–LEARNING AGENT

As the architecture for the Q-learning agent with respect to image input is relatively complex and large, `Orion HPC`[1] was used when training the agent.

See the **code** for full implementation with examples.

[1]The author acknowledge the Orion High Performance Computing Center (**OHPCC**) at the Norwegian University of Life Sciences (NMBU) for providing computational resources that have contributed to the research results reported within this paper.

References

[1] Amrani Amine. *A gentle introduction to Reinforcement Learning*. 2020. URL: https://aamrani1999.medium.com/a-gentle-introduction-to-reinforcement-learning-d26cba6455f7.

[2] Amrani Amine. *Q-Learning Algorithm: From Explanation to Implementation*. 2020. URL: https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187.

[3] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[4] Huggingface. *The Bellman Equation: simplify our value estimation*. URL: https://huggingface.co/learn/deep-rl-course/unit2/bellman-equation.

[5] Huggingface. *The Deep Q-Learning Algorithm*. URL: https://huggingface.co/learn/deep-rl-course/unit3/deep-q-algorithm.

[6] Huggingface. *Two main approaches for solving RL problems*. URL: https://huggingface.co/learn/deep-rl-course/unit1/two-methods.

[7] Huggingface. *Two types of value-based methods*. URL: https://huggingface.co/learn/deep-rl-course/unit2/two-types-value-based-methods.

[8] Huggingface. *What are the policy-based methods?* URL: https://huggingface.co/learn/deep-rl-course/unit4/what-are-policy-based-methods.

[9] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*. 2015. URL: https://doi.org/10.1038/nature14236.

[10] OpenAI. *Learning to play Minecraft with Video PreTraining*. 2022. URL: https://openai.com/research/vpt.

[11] OpenAI. *Part 2: Kinds of RL Algorithms*. 2018. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[12] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[13] Mark Towers et al. *Cart Pole*. URL: https://gymnasium.farama.org/environments/classic_control/cart_pole/.

[14] Mark Towers et al. *Gymnasium*. URL: https://github.com/Farama-Foundation/Gymnasium.

[15] Christopher J. C. H. Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. DOI: 10.1023/A:1022676722315. URL: https://doi.org/10.1023/A:1022676722315.

[16] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256.

[17] Chris Yoon. *Deriving Policy Gradients and Implementing REINFORCE*. 2018. URL: https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63.