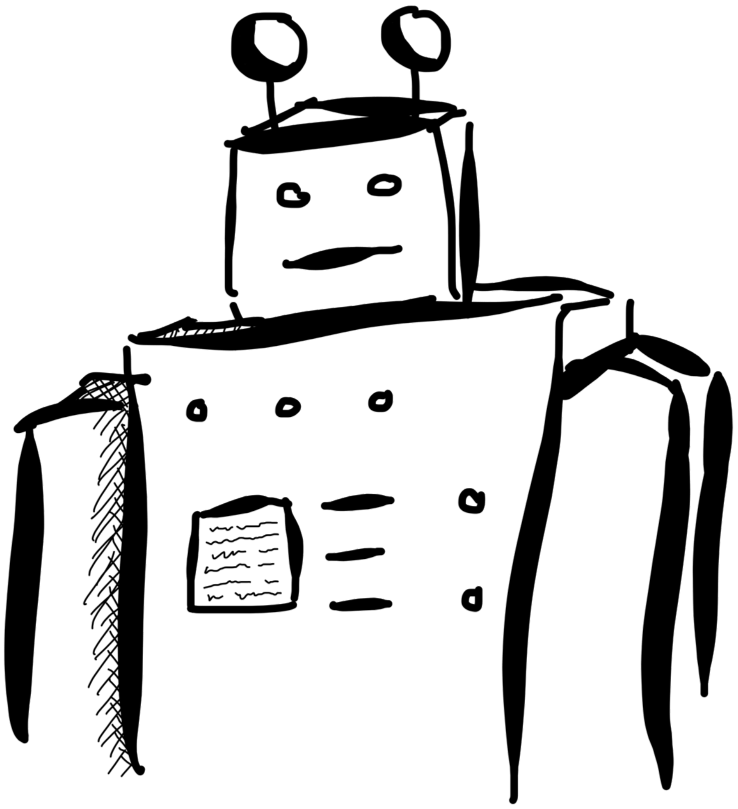
REINFORCEMENT LEARNING

Hallvard Høyland Lavik

hallvard.hoyland.lavik@nmbu.no



Contents

Figures

Listings

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 optimal actions with respect to the environment's states.

As the agent explores the environment, it starts with no knowledge and must learn through trial and error which actions lead to positive rewards. This learning process involves adjusting the parameters of the agent (*i.e.*, the weights of the network) based on the rewards it receives over many iterations. 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 said reward. For example, in OpenAI's work on mastering Minecraft [1], They had a goal for their agent to obtain diamond tools, which "usually takes proficient humans over 20 minutes (24,000 actions)". To overcome this challenge, OpenAI sequentially rewarded the agent based on its progress, providing a more immediate reward for its actions.

Methods

Reinforcement agents aim to execute actions within an environment that result in the highest rewards. The term "reinforcement" is used because the agent learns to adapt its actions based on past experiences. Optimal behavior can be achieved through various reinforcement learning techniques, which are broadly categorized into two main types: model–free and model–based methods, as seen in Figure 1.

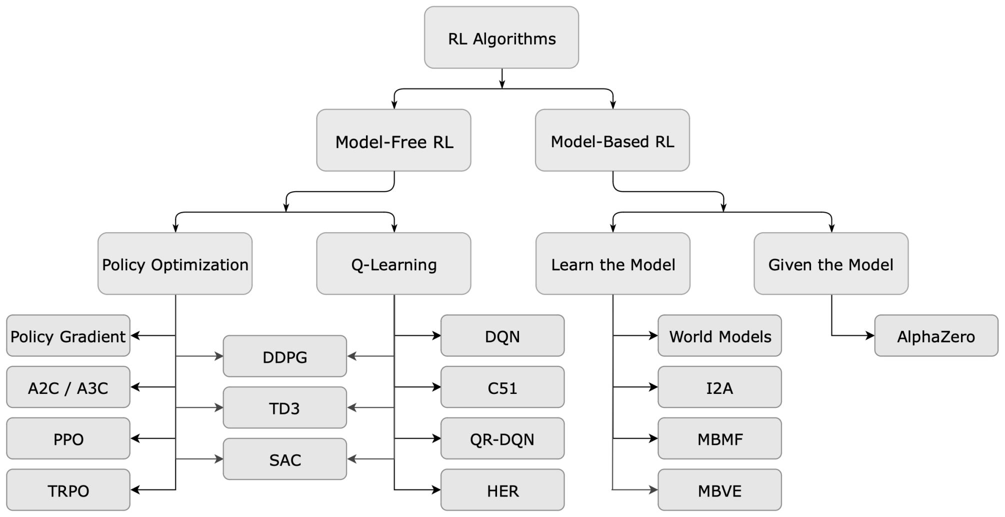


Figure 1: Reinforcement learning algorithms [2].

MODEL–FREE AND –BASED LEARNING

Whether an agent is categorized as model–free or –based depend on its knowledge and understanding of the environment it interacts with. A model–free agent refers to an agent that has – and gains – no knowledge on the workings of the environment, without attempting to understand the environment's underlying dynamics. In other words, the agent only learns to perform optimal actions given the environment, and nothing more. [2]

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 [3] 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 [...])”.

The various algoriths and their category can be seen in Figure 1, as presented by OpenAI [2]. In this report, the algorithms Policy Gradient and DQN is studied, which both fall under the model-free category.

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

A policy can be understood as a strategy used for choosing actions that maximize future rewards. This strategy is typically represented as a probability distribution, describing the likelihood of each of the actions leading to the highest future rewards, given a state of the environment.

In contrast, a value-based method evaluates the relationship between states of the environment and their expected future rewards. Instead of providing a probability of achieving a future reward, it predicts the expected return for each action in a given state, thereby offering a measure of the desirability of each state-action pair.

In the context of the game of blackjack, the state corresponds to the current cards in hand, and the possible actions consist of either `hit` or `stand`. A policy represents the probability of each action leading to the highest future reward, given the current state; it assesses the likelihood of choosing `hit` versus `stand` resulting in a reward. On the other hand, a value-based approach associates each possible action with the expected future reward in the context of the current state. For each state, it predicts the anticipated return of selecting either `hit` or `stand`, as opposed to providing the probability of each action leading to a reward.

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), used for exploration, and the target network ($\hat{Q}$), used for learning, are independent when calculating the loss, making it an off–policy approach.

That is, on–policy learning directly optimize to choose actions leading to rewards, whereas off-policy learning indirectly optimizes for said reward by comparing its (Q) predictions to a ”ground truth” ($\hat{Q}$) prediction.

POLICY–BASED APPROACH

Denoted as $\pi(a|s)$, the policy of an agent is the probability that an action, a, will yield the highest possible future reward, given a state, s. When the agent performs the chosen 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. [4]

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. [5] 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" [6], is here presented in pseudocode. This approach is based on, and modified from, Yoon [7].

REINFORCE algorithm

```
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 [7]
```

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

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

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, using the mean, μ_R, and standard deviation, σ_R.

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

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

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. [8]

Q–learning

In order to learn optimal policy in an environment, a Q-learning agent learns to associate each state with the expected rewards for each possible action. 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. [9, 10, 11]

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. [12, 13, 11]

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 [14, 13]. 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's parameters.

The parameters of the agent are updated in an off–policy manner through gradient descent, with respect to a loss; the difference between the predicted and target Q–values [14]. Equation (5) can therefore be modified and rewritten:

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

Here, $Q(S_t, a|\theta)$ is the previously predicted Q-value for taking action a at state S_t given the parameters θ, and R_{t+1} the immediate reward after taking action a at time t. γ is the discount factor, and $\hat{Q}(S_{t+1}, a|\hat{\theta})$ 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 (where C is an arbitrary integer). 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." [13]

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 predictive ability 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 what is 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.

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

The value of ε is typically high at the beginning of the learning–process, and reduced thereafter (*e.g.*, linearly or exponentially decaying), until a final ε is reached.

Deep Q–learning algorithm

```
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 $\varepsilon$ 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 $\varepsilon$
    every $C$ steps update $\hat{Q}$ as copy of agent
end for
```

Slightly modified from Mnih *et al.* [13]

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 [15]. 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. [15]

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.

```
for i, (_in, _out) in enumerate(
        zip([network["inputs"]] + network["nodes"],
            network["nodes"] + [network["outputs"]])
):
    setattr(self, f"layer_{i}",
            torch.nn.Linear(_in, _out, dtype=torch.float32))
```

MULTI–DIMENSIONAL OBSERVATION SPACE

Representing a continuous observation space of set width and height – as seen in the Enduro 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 hyperparameters and activation functions.

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

```
for i, (_in, _out, _kernel, _stride, _padding) in enumerate(
        zip([network["input_channels"]] + network["channels"][:-1],
            network["channels"],
            network["kernels"],
            network["strides"],
            network["padding"])
):
    setattr(self, f"layer_{i}",
            torch.nn.Conv2d(
                    _in, _out,
                    kernel_size=_kernel,
                    stride=_stride,
                    padding=_padding
                ))

# Here, "_output" is the number of output nodes of Conv2d;
setattr(self, f"layer_{len(network['channels'])}",
        torch.nn.Linear(_output, network["nodes"][0],
                        dtype=torch.float32))

for i, (_in, _out) in enumerate(zip(
            network["nodes"],
            network["nodes"][1:] + [network["outputs"]])
):
    setattr(self, f"layer_{len(network['channels']) + i + 1}",
            torch.nn.Linear(_in, _out, dtype=torch.float32))
```

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” [13], as well as presumably using a different agent architecture as to the following.

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, as this is the value used for calculating the policy gradient.

Obtaining the action. along with the logarithm of its probability, is achieved by adding a wrapper to the forward pass of the agent:

```
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, as the action is chosen through a randomly weighted sampling 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. [7]

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 (*i.e.*, a minibatch) are selected every time the agent's parameters are updated. See the code for the full implementation.

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.

The optimal Q-value for the final step in the sequence (contained in `steps` in the code below, as there are multiple sequences stacked) is set the actual observed reward, as Mnih *et. al.* [13] 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. [14]

Image input

When 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:

```python
def preprocess(self, state):
    state = torch.tensor(
        state, dtype=torch.float32
    ).view(self.shape["original"])

    state = state[:, :, self.shape["height"], self.shape["width"]]

    state /= 255.0

    state = torch.nn.functional.interpolate(
        state, size=self.shape["reshape"][2:4], mode='area'
    )

    return state
```

where the input first is cropped to a specified `shape`, before it is normalized and resized. 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["reshape"]` the wanted shape of the preprocessed state.

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:

For the agent to gather the useful information based on the observed states, a wrapper is put around the environments' step function. an arbitrary number of frames, `skip`, can be combined between each observation. When combining the frames, the maximum value per pixel is selected based on the skipped of frames. The reason for this is to both keep the process memory–efficient, and to give the agent some sense of movement; when combining the ensemble of `skip` frames by using the maximum values across them, a trailing effect of moving objects is obtained.

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

    done = False
    rewards = 0.0
    states = torch.zeros(self.shape["reshape"])

    for i in range(0, self.shape["reshape"][1]):
        new_states = torch.zeros((1, skip, *self.shape["reshape"][2:4]))

        for j in range(skip):
            new_state, reward, terminated, truncated, _ = environment.step
                (action.item())
            done = (terminated or truncated) if not done else done
            rewards += reward

            new_states[0, j] = self.preprocess(new_state)
        states[0, i] = torch.max(new_states, dim=1, keepdim=True).values

    return action, states, rewards, done
```

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.

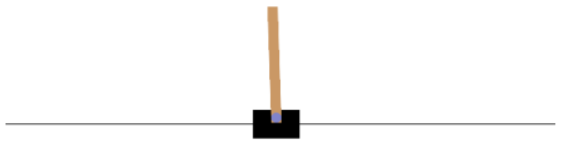


Figure 2: Random observed cart-pole environment state.

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. [16, 17]

```python
import gymnasium as gym

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

The agent controls the cart movement by pushing it one way or another. The termination (or truncation) is determined by the observed values (see value ranges below), or if the game 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**
$-2.4 < x < 2.4$	cart position	0	push cart to the left	+1	every time–step
$-\infty < x < \infty$	cart velocity	1	push cart to the right		(until termination)
$-12^\circ < x < 12^\circ$	pole angle				
$-\infty < x < \infty$	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.

VALUE–BASED Q–LEARNING AGENT

Likewise, the value–based agent was initialised with the same hyperparameters 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 the full implementation with examples.

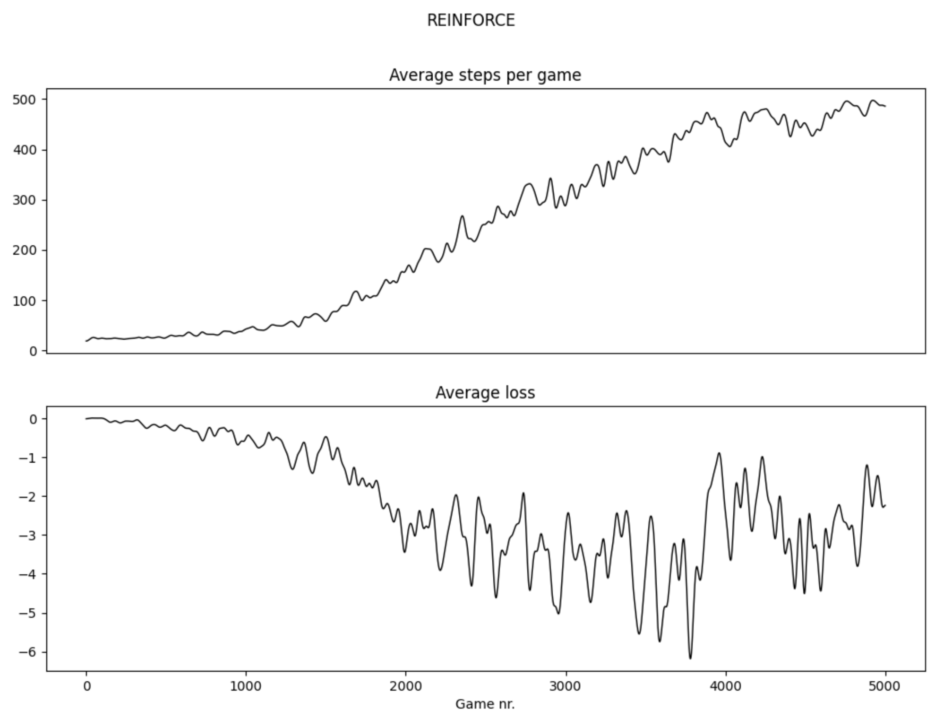


Figure 3: Training of the policy–based (REINFORCE) agent.

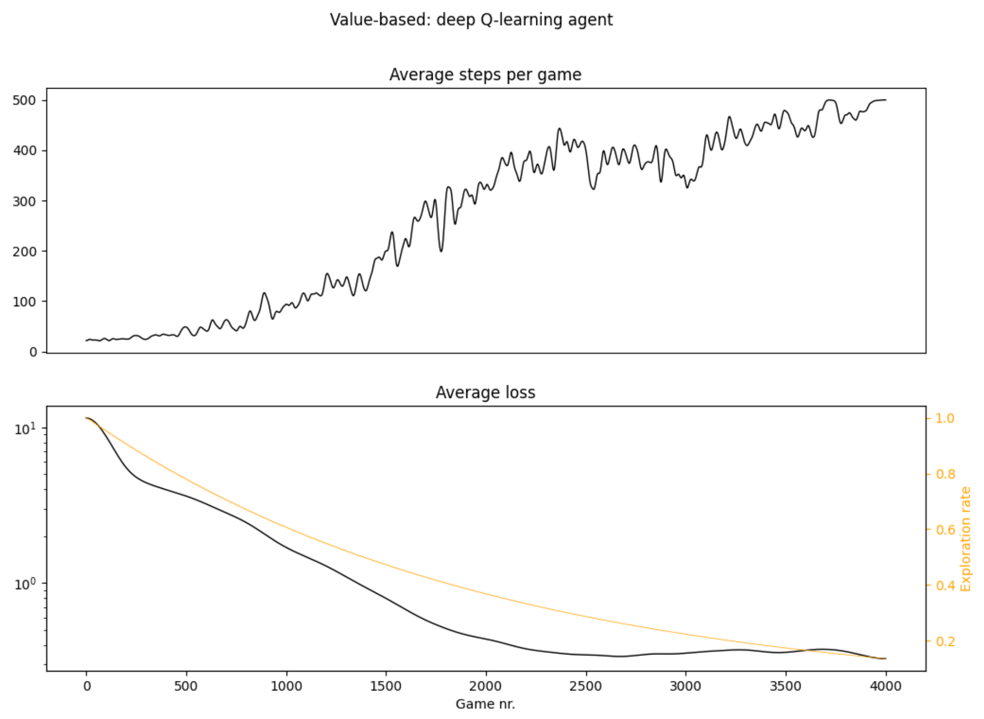


Figure 4: Training of the value–based (DQN) agent.

Enduro environment

The Enduro environment (seen in Figure 5) is a part of the `gymnasium` [16] package in Python. The agent controls the car movement by moving it left, right, down, down–left or down–right. And whether it fires a "shot" straight ahead, to the right or to the left. The agent can also choose to do nothing, and is given a reward when another vehicle is overtaken. The game ends when the number of cars passed each virtual "day" is below a required limit.
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.

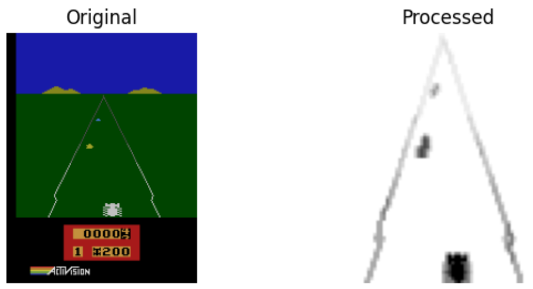


Figure 5: Random observed Enduro environment state before (left) and after (right) preprocessing.

OBSERVATION

$0 \leq x \leq 255$	game screen 210×160

ACTIONS

0	no action	5	down–right
1	fire	6	down–left
2	right	7	right–fire
3	left	8	left–fire
4	down		

REWARDS

$+1$	vehicle overtake
-1	overtaken by another

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`* was used when training the agent.

The agent architecture (inspired by Cutajar [18]) used convolutions to handle the image input. The preprocessed game–screen can be seen in Figure 5, where six frames were skipped and combined as input to the agent. When combining the frames, the maximum value per pixel was selected based on the skipped of frames. The reason for this is to both keep the process memory–efficient, and to give the agent some sense of movement; when combining the ensemble of `n` frames by using the maximum values across them, a trailing effect of moving objects is obtained.

During training the parameters were updated every game with respect to a mini–batch of size `32`, while playing `4 000` total games. The results achieved were surprisingly good, as seen in Figure 6, where the agent was sufficiently able to learn optimal strategies. As the number of total steps (and rewards) continues to increase, one would expect results to continue improving if trained further. The checkpointed agent can be seen in action here.

* 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.

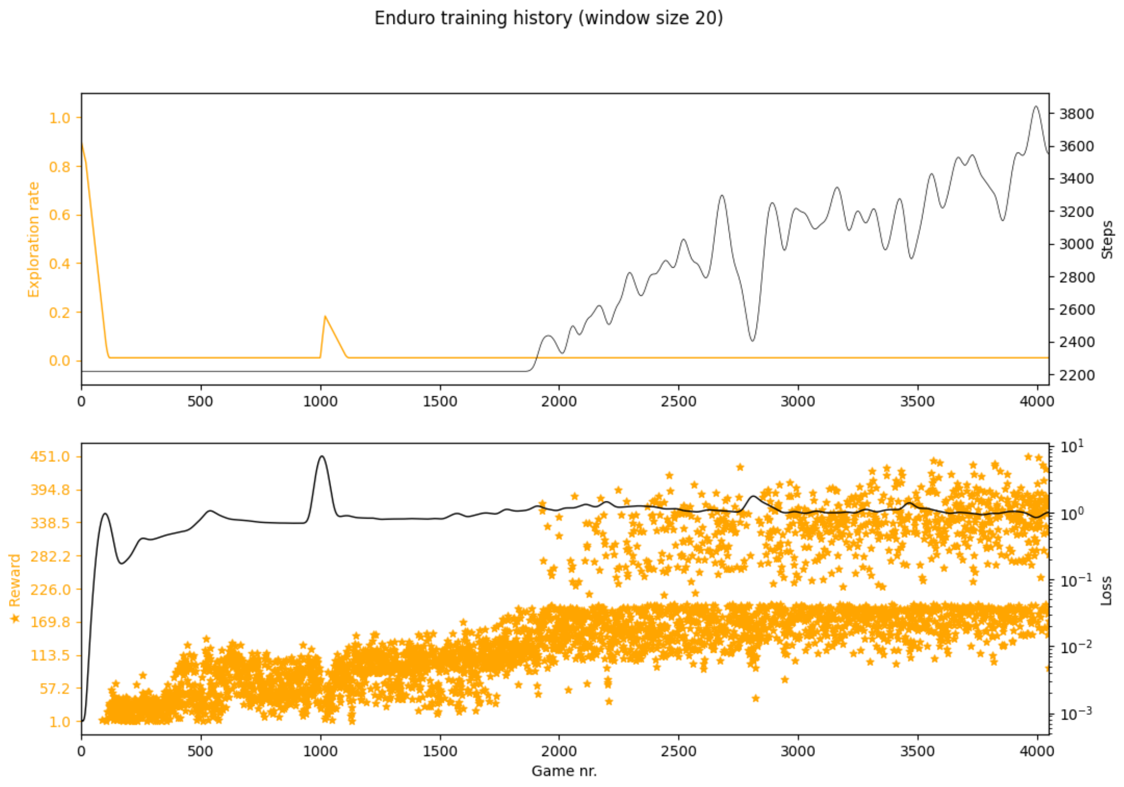


Figure 6: Enduro training results.

Tetris environment

The Tetris environment (seen in Figure 7) is a part of the gymnasium [16] package in Python. The agent controls the piece movement 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), and the current built height (see below). 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 representation, 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 representation, and how to act based on this interpretation.

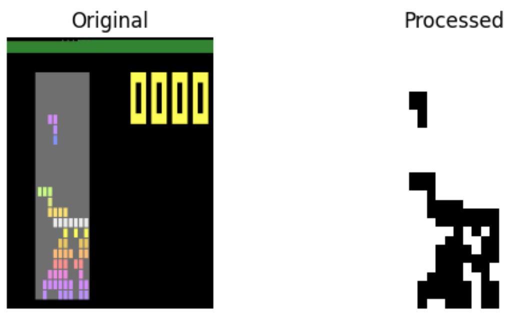


Figure 7: Random observed Tetris environment state before (left) and after (right) preprocessing.

Because of the high–dimensional input space which can take a high number of variations, the Tetris environment is slightly different from and more complex than the cart–pole environment, and therefore requires a more intricate agent network.

	OBSERVATION		**ACTIONS**	**REWARDS**
$0 \leq x \leq 255$	game screen 210×160	0	no action	see section below
		1	rotate	
		2	right	
		3	left	
		4	down	

CUSTOMIZED REWARD

For the agent to be nudged in the right direction when it comes to understanding the game, a customized reward-function is implemented. The reason for this is that countless architectures– and hyperparameter–combinations were priorly tested, without any skill improvement whatsoever. Thus, the build–up of tetriminos is combined with the (if any) reward given for rows removed. The wrapper is thus added to account for this additional rewarding. The intuition for introducing the rewarding with respect to build height is for the agent to gain some insight into how good a given position is. That is, high towers yield negative rewarding, whereas lower buildup is positive.

```python
def _reward(self, state, reward):
    state = self.preprocess(state)
    built = state.shape[0] - (state == 0.0).all(1).nonzero()[-1].item()

    if reward > 0:
        reward *= self.parameter["incentive"]
    elif self.parameter["new"]:
        self.parameter["new"] = False

        if built <= state.shape[0] / 2.5:
            reward = self.parameter["incentive"]
        else:
            reward = self.parameter["punishment"]

    reward *= state.shape[0] / (built * self._holes(state))
    return state, reward
```

Here, the algorithm finds the current build height by extracting the index of the first empty row from the bottom (`state == 0.0`). If the current action returned a reward (here representing the environments returned reward, *i.e.*, whether any rows have been filled) it is scaled by the `incentive`. Otherwise, the reward is set to either `incentive` or `punishment`, based on the current built height, given that the previous tetrimino has been placed and a new one starts falling (defined by the flag `self.parameter["new"]`, which checks for this).

```
def _holes(state):
    above = torch.roll(state, 1, 0)[1:, :].clone().detach()
    below = state[1:, :].clone().detach()
    difference = below * 2 - above

    holes = sum(difference.flatten() == -1) - 2
    holes = max(holes.item(), 1)
    return holes
```

In addition, a simple check for "holes" (*i.e.*, spaces in the game area with tetriminos above and on the sides) are used to scale the agents' feedback. An example of this logic can be seen here.

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`* was used when training the agent.

The agent architecture used convolutions to handle the image input, and during training, multiple frames were stacked to keep the process memory–efficient. The preprocessed game–screen can be seen in Figure 7, where three frames were skipped and combined as input to the agent. When combining the frames, the maximum value per pixel was selected based on the skipped of frames. The reason for this is to both keep the process memory–efficient, and to give the agent some sense of movement; when combining the ensemble of `n` frames by using the maximum values across them, a trailing effect of moving objects is obtained.

Through Figure 8 one sees that the loss consistently increases and flattens out, without resulting in respective increase in rewards and game lengths. This implies a poor choice of agent architecture and/or hyperparameters and room for improvement. The rewards seem to slightly increase as the agent learns, but without really getting any better. Again, this suggests either a poor parameter choice, or faults in the algorithm.

During training the parameters were updated every fifth game with respect to a mini–batch of size `64`, while playing `22 500` total games. The checkpointed agent can be seen in action here.

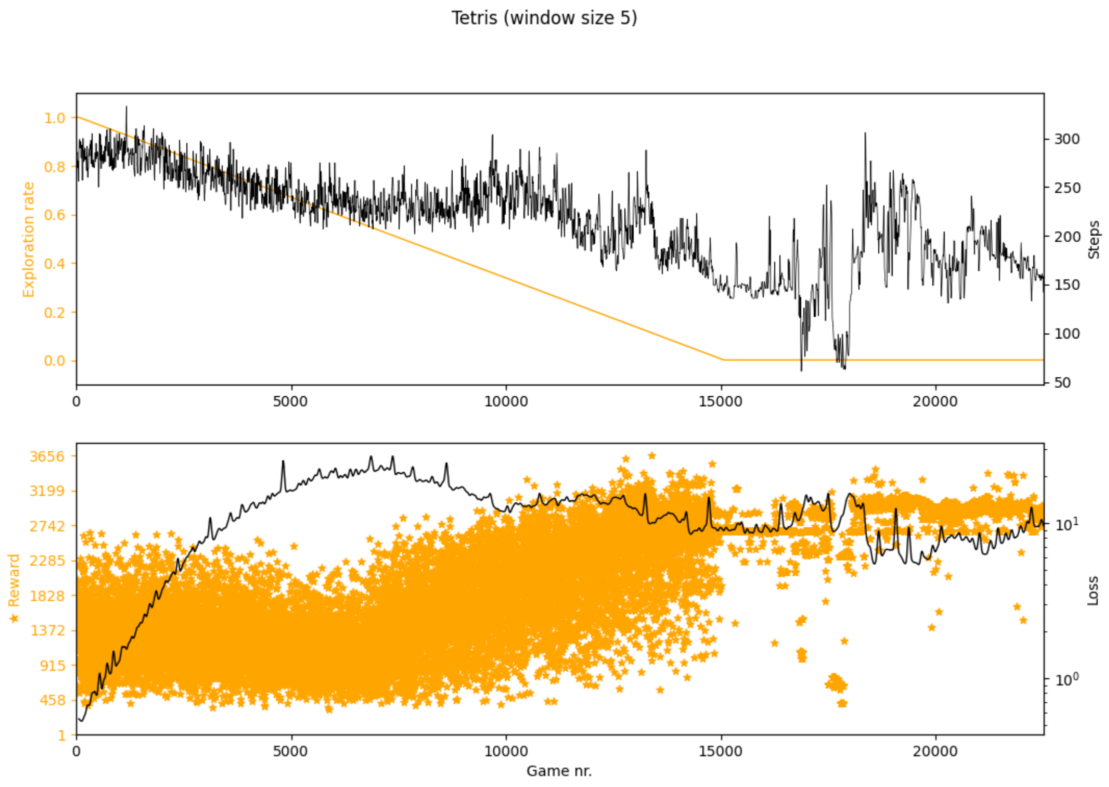


Figure 8: Tetris training results.

Breakout environment

The Breakout environment (seen in Figure 9) is a part of the `gymnasium` [16] package in Python. The agent controls the paddle movement by firing or moving the paddle left, right or nowhere, and is given a reward corresponding to the number of squares hit, their colour representing their reward (which is hard to see based on the normalized representation in Figure 9). The game ends if all "bullets" are used and makes it past the paddle.
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.

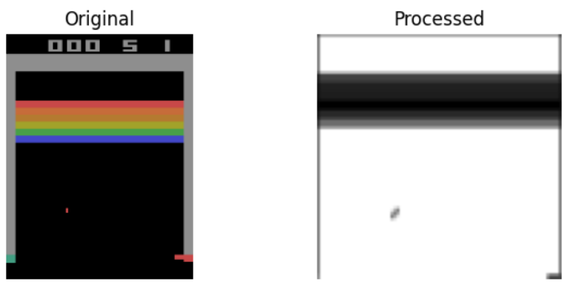


Figure 9: Random observed environment state before (left) and after (right) preprocessing.

	OBSERVATION		**ACTIONS**		**REWARDS**
$0 \leq x \leq 255$	game screen 210×160	0	no action	+1	blue or aqua brick
		1	fire	+4	yellow or green brick
		2	right	+7	red or orange brick
		3	left		

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`* was used when training the agent.

The agent architecture used convolutions to handle the image input, and during training, multiple frames were stacked to keep the process memory–efficient. The preprocessed game–screen can be seen in Figure 9, where four frames were skipped and combined as input to the agent. When combining the frames, the maximum value per pixel was selected based on the skipped of frames. The reason for this is to both keep the process memory–efficient, and to give the agent some sense of movement; when combining the ensemble of `n` frames by using the maximum values across them, a trailing effect of moving objects is obtained.

During training the parameters were updated every game with respect to a mini–batch of size `32`, while playing `59 000` total games (in two parts, with exploration restarted as seen by the spike in Figure 9). The results achieved and presented here were surprisingly bad, suggesting either poor hyperparameters and/or architecture. The checkpointed agent can be seen in action here.

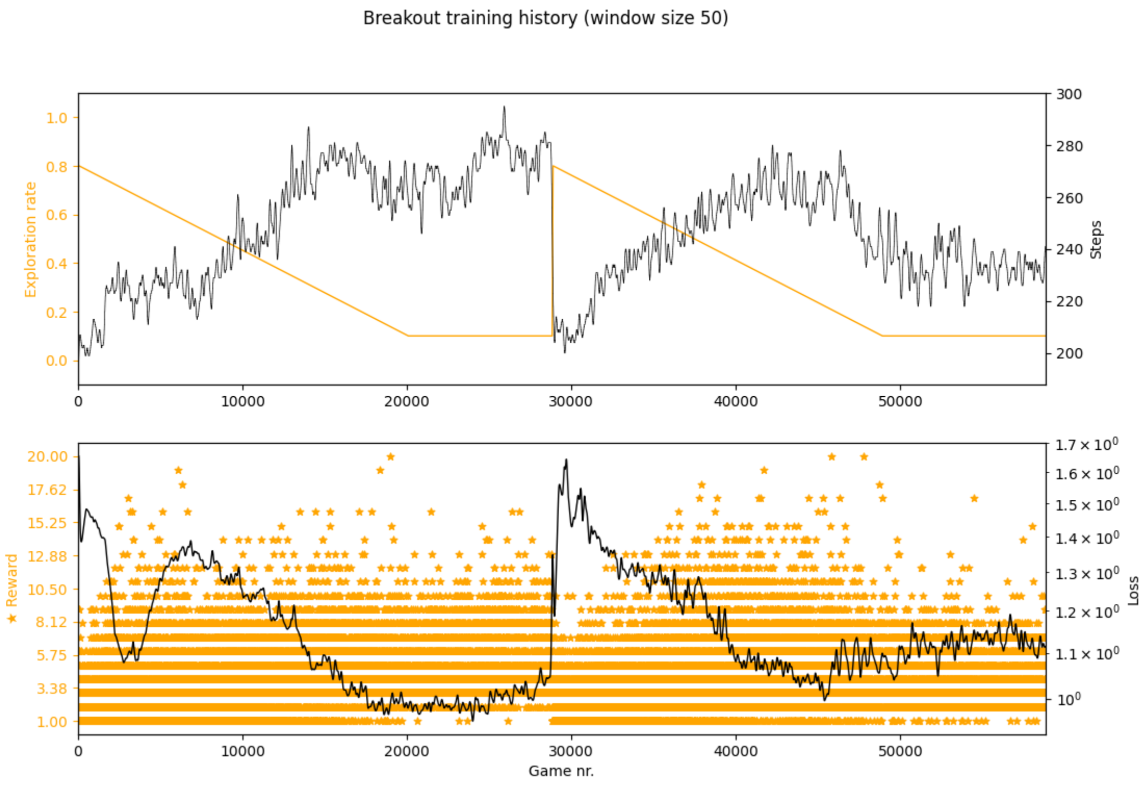


Figure 10: Breakout training results.

References

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

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

[3] 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.

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

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

[18] Mark Cutajar. *Playing Enduro using a DQN Agent*. 2021. URL: https://markcutajar.com/blog/atari-enduro-dqn/.

Source code

The source code behind this report can be found here.

Deep learning libraries

During implementation, PyTorch was mostly used.

I also experimented slightly with the new ml-explore (MLX) for Apple's M-chips. For the high-dimensional (*i.e.*, image input), Orion HPC was used, and there was therefore no point in converting this code to MLX.

Assistance

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.

During creation of the codebase, GitHub Copilot along with Mistral was used –mostly helping with tensor dimension issues.

Prompts equivalent to

```
How do I reshape the tensor [...]  to correspond with this [...]
Modify this code such that shapes match [...]
Smooth and plot the contents of a csv-file using pandas.
```

were used. *I.e.*, sparring when I was stuck on something.

During writing of this report, **no** artificial intelligence tools were used to generate text. Mistral was however used when either shortening or correcting/rephrasing certain paragraphs/sentences.

Prompts equivalent to

```
Shorten the paragraph [...]
Highlight errors in this paragraph [...]
```

were used.