Teaching Skiing to the Computer

Thomas Wimmer, Jules Soria, François Wissocq, Maximilien Chau

***Abstract*—This paper explores the application of Reinforcement Learning (RL) to solve the Atari game 'Skiing' using the Gymnasium toolkit. The main objective of the game is to navigate a skier down a slope while avoiding obstacles such as trees and flags, and make the skier go through gates. The game presents a challenging task due to the complex and dynamic environment, which makes it an ideal testbed for RL algorithms. In the paper, we evaluate different methods, from the classic SARSA and Q-learning to more advanced and modern techniques such as Double Deep Q-learning. We also evaluate different methods to interpret and extract information from the game visual display to take actions. We show the various problems faced during development, and show how the more complex algorithms do not necessarily perform better. Overall, this study demonstrates the potential of RL algorithms to learn effective policies for complex and dynamic environments such as the Skiing Atari game.** [1]

I. Introduction

Reinforcement Learning applied to video games is interesting as it can be applied to a wide variety of environments. Agents can experiment very different situations and have to adapt efficiently to them given their past experience. This is a similar behaviour to how humans play video games, thus leaving space for comparison between the two. Through video games, we can simulate risk-free environments and use algorithms to find well-optimised solutions for complex problems. The agents of the system should get rewards accordingly to their actions but in a mathematical manner, different from our intuition when playing a video game. Those rewards should translate the goal of the game but not necessarily how humans would try to reach it. Defining the reward of the game proves itself to be a difficult challenge as it requires a good understanding of how agents will take decisions given them. Facing the complexity of the environment defined by the game, agents should be able to recognise patterns and generalise rewarded decisions made from a particular observation to the similar scenarios. Therefore, an important part of this project was to define the right information to give to the algorithms from the observations of the game.

Other works such as [2] have already been done on reinforcement learning applied to Atari games showing impressive performance on those.

Our work aims at making a survey of different agents on the Atari 2600 game "Let's Play Skiing". We used different data pre-processing ideas to fit the implemented algorithms. Then, we tried to optimise the agents to compare methods and show when they are performing the best and how agents react to different reward policies.

[1] A Jupyter notebook that can be executed using Google Colab can be found here.

II. Background

A. Mathematical Formulae and Notations

Markov Decision Processes (MDPs) are mathematical models that are commonly used to represent sequential decision-making problems in Reinforcement Learning (RL)[4]. MDPs consist of a set of states, a set of actions, a reward function, and a transition function. The transition function specifies the probability of moving from one state to another when an action is taken, and the reward function specifies the immediate reward received upon transition from one state to another.

In an MDP, an agent interacts with an environment in discrete time steps. At each time step t, the agent observes a state s_t from a set of possible states $\mathcal{S}$, takes an action a_t from a set of possible actions $\mathcal{A}$, and receives a scalar reward r_t from a set of possible rewards $\mathcal{R}$. The agent then transitions to a new state s_{t+1} with probability $P(s_{t+1}|s_t, a_t)$, where P is the transition function.

The agent's goal in an MDP is to learn an optimal policy $\pi^*(s)$ that maps each state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$ that maximizes the expected cumulative reward. The expected cumulative reward is calculated as the sum of rewards obtained from time step t to the end of the episode, with a discount factor $\gamma \in [0, 1]$ applied to future rewards to account for the time delay. (As such, a value of $\gamma = 0$ will be forcing the agent to focus on immediate rewards while a value of $\gamma = 1$ will instead put the focus on the total rewards).

Q-Learning is a model-free RL algorithm used to learn the optimal action-value function (Q-function) of an agent in an MDP with a discrete action space. The Q-function estimates the expected cumulative reward of taking a specific action in a given state and following an optimal policy thereafter. Q-learning works by iteratively updating the Q-values of each state-action pair using the Bellman equation, which relates the Q-value of a state-action pair to the Q-values of its neighboring state-action pairs.

The Bellman equation for the Q-function in an MDP is given by:

$$Q(s_t, a_t) = \mathbb{E}\left[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \middle| s_t, a_t\right]$$

where r_t is the reward received after taking action a_t in state s_t, and a_{t+1} is the action taken in the next state s_{t+1} under the policy $\pi^*(s_{t+1})$.

III. Methodology/Approach

A. Our environment

We used in our project the Atari Skiing game (version 5) which is readily available in the Gymnasium library (previously called Gym, and developed by the famous OpenAI company).

Due to the nature of the game, the observation is an image, with three channels, of size 160x210 and with 8bit pixels, taking values in the range $[0 - 255]$ The entire observation space is $256^{(210*160*3)} \simeq 3.88\text{e}{+242750} \simeq \infty$ which is completely impossible to work with reasonably. We are faced with the *curse of dimensionality*.

To solve this curse, we have some tools at our disposal and we can use clever techniques to reduce the dimensions. First of all, what interests us, and the algorithm, is not the image itself, but the information contained.

B. Preprocessing

Indeed, using 2D filters, we can identify and extract key information from the observed image, such as the ski positions, the skier location on the image, and the position of the next gate. With this approach, we greatly reduce the quantity of information, but keep the quality. This permits us to use tabular algorithms which would have simply not worked at all otherwise.

Since Atari's games were some of the first video games, the design of the scenes within the game are fairly simple. The skier can position himself and his skis in eight different positions (four leaning to the left and four to the right, as shown in Fig. 1). Since the visuals for these positions always stay the same, it is possible to design eight different filters that we can pass over the input observation to find the position and ski positioning of the skier. This enables us to greatly reduce the input space to simply three variables: The x and y position of the skier, as well as his ski positioning. While the x-coordinate can be in the range between 0 and 140 (the image width of the cropped observation), the y-coordinate more or less always stays the same and the ski positioning is a variable in the range $[0, 7]$.

Fig. 1. There are eight different positions for the skier that we can manually detect and extract from the scenes.

Because the flags always keep the same shape, we can design a filter matching the gates that can be used in a convolution over the input image. It is possible that multiple gates are shown on the screen at the same time if the skier moves. To simplify and abstract the most important information, we are only interested in the gate that is the uppermost, as this will be the next gate the skier has to pass. We thus are able to reduce the input observation to a set of only 5 variables; The skier's position, his ski positioning, and the position of the next gate[2]

[2]It is important to note at this point that we are thus keeping most of the important information, but we e.g. don't take into account the trees in the scene. However, as the Atari Skiing Environment provided, is played in the easy gamemode, the trees are only to the left and right of the slope and thus are not obstacles in the way of the skier.

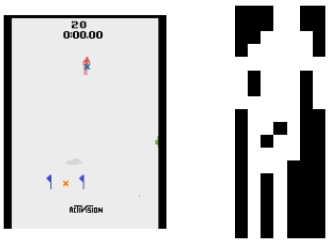

Fig. 2. The detection of the positions of the next gate and the skier (detected positions marked with an x) and his ski positioning gives accurate information about the scene in a 5-dimensional state.

C. The Reward Function

The environment itself has a native reward function, which is the elapsed time. As the goal of the game is to ski down the course as fast as possible, taking more time is worse. Therefore, the reward is always negative. At the end of the run, missed gates are expressed as time penalty (5 seconds for each gate missed). Bumping into a tree or a flag pole induces a penalty in time itself, by stopping the skier, but there is no other penalty.

This reward function is problematic, because it gives constantly negative rewards, no matter how the skier plays, and gives true feedback only at the end. As such, some agents learn to optimize their gameplay by staying immobile and wait for the timer to run out.

Having this in mind, there is a need for a new custom reward function. In the further course of this section, we will discuss the design of possible reward functions used in the experiments.

We can use the detectors presented in III-B to design our custom reward functions. We first propose a simple reward function that simply rewards every pass through a gate with a certain award, which is left to define as a hyperparameter. Further, if we want to stop the agent from steering towards the borders of the slope, where no gates can be found and trees stop the skier from moving, we can create a penalty for coming too close to the border and a reward for actively moving out of these regions again.

As explained before, the agents might learn to just break and stay in the same position until the time runs out. To prevent this behavior, we designed a reward and penalty that can be adapted for the different ski positioning. By that, we can penalize the horizontal positioning of the skis which is used to break.

Finally, we observed in our experiments that a valid strategy of the agent can be to simply go straight down the mountain, as it usually passes nine out of the twenty gates with this strategy (more on that in the next sections). To alleviate this problem, several ideas come to mind. We designed another function which gives a reward at every time step depending on the horizontal distance between the skier and the center of the next gate, with the intention to make the agent learn to steer more towards the gates. Another reward / penalty is based on the recently played moves of the agent. It thus promotes changes of direction and penalizes staying in the same position for too long.

The design of the right reward function includes an extensive hyperparameter search over finding the best combination of the designed rewards, up to setting the values for specific rewards or penalties.

D. Q-Learning

Q-Learning is one of the first solutions that comes to mind when it comes to implementing a reinforcement learning algorithm, as it has proven to give good results on a wide range of problems. It relies on the Bellman equation given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \big[R_{t+1} + \gamma \max_a (Q(s_{t+1}, a) - Q(s_t, a_t)) \big]$$

The Q-Learning algorithm chooses the action according to the state the agent is in and the best action to be taken in this state derived from the Q-table. In a video game like "Let's Play Skiing", the environment space is quite large. This causes our algorithm to have to go through too many state-action pairs to be efficient, and without training for a very long time, our agent's behavior would be very similar to a random behavior. For such algorithms using the Q-table, we have to further reduce the observation space.

Since Q-Learning relies on the Q-table and the state-action pairs contained, the reward attribution should be carefully implemented. When the skier passes a gate, he gets a reward but it is obviously not only for the last action he did. For this reason, we came up with the idea of redistributing the reward to the previous state-action pairs of the frames preceding the attribution of the reward. Those state-action frames are stored in a buffer and receive a weighted reward of the original one. The size of the buffer should be big enough to give weight to useful state-actions but not too big which would result in rewarding sub-optimal actions occurring in certain states.

E. A solution: SARSA

SARSA (State-Action-Reward-State-Action) is a reinforcement learning algorithm that is used to learn the optimal policy for a Markov decision process (MDP)[4]. It works by using an iterative process to update the Q-values of each state-action pair based on the observed reward and the next state and action taken. Unlike other reinforcement learning algorithms like Q-learning, SARSA takes into account the current policy being followed when selecting the next action to take. This makes SARSA more suitable for problems where the exploration-exploitation trade-off is important.

The formula for updating the Q-value for a state-action pair in SARSA is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \big[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \big]$$

In this formula, s_t and a_t represent the current state and action taken, respectively. R_{t+1} is the reward obtained after taking the action a_t in state s_t and transitioning to the next state s_{t+1} by taking action a_{t+1}, which is chosen using the current policy. α is the learning rate, which controls the step size of the Q-value update, and γ is the discount factor, which controls the importance of future rewards. The update rule essentially adjusts the Q-value for the current state-action pair based on the expected future rewards that can be obtained by taking the action determined by the current policy.

F. Another approach: REINFORCE

REINFORCE (REward Increment = Nonnegative Factor x Offset Reinforcement x Characteristic Eligibility) is a policy gradient method used for learning in reinforcement learning[6]. It works by directly optimizing the policy by maximizing the expected return, which is the sum of the rewards obtained over a trajectory. To do this, it computes the gradient of the expected return with respect to the policy parameters, and updates the policy in the direction of the gradient using stochastic gradient ascent. Unlike value-based methods that estimate the value function, REINFORCE can be used for problems with continuous action spaces.

The formula for updating the policy parameters using the REINFORCE algorithm is:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t$$

In this formula, θ represents the policy parameters, α is the learning rate, and G_t is the total return obtained after taking the action a_t in state s_t. $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under the policy defined by the current parameters θ. The term $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is the policy gradient, which tells us how the probability of taking the action a_t changes with respect to the policy parameters θ. The update rule essentially increases the probability of actions that lead to higher returns and decreases the probability of actions that lead to lower returns.

G. Deep Q-Learning

Deep Q-Learning [2] builds upon the ideas of TD- and Q-Learning but now replaces $Q(s, \cdot)$ with a neural network. This substitution makes the system more flexible and enables us to use a broad range of methods known from Deep Learning such as CNNs that can help in dealing with image as inputs.

As the input for the network is highly correlated, we must find a way to alleviate the problem of learning just a replay. A common solution for this problem is to use a replay buffer from which we sample during training.

In our initial experiments, we used a simple DQN, but we then also carried out experiments with the Double DQN method which uses a separate predictor network besides the q-network, as this method usually outperforms the simple DQN [5].

H. Representation Learning

Self-supervised learning has been shown to be effective in a variety of domains, including computer vision, natural language processing, and has become a popular technique for unsupervised pretraining of deep neural networks. We propose to use an autoencoder as an alternative (or complement) to handcrafted features. The goal being to find a procedure generalizable to other environments, reducing the time spent to design specific reward functions.

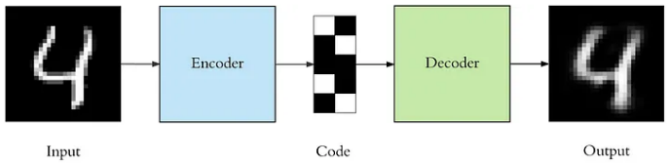

Fig. 3. autoencoder : lossy compression of an image into a latent space by the encoder, then the decoder reconstructs the image from the compressed data. (Picture by A. Dertat)

Convolutional networks are great to reduce information of grid-like structures. Some DRL architectures already include those components. Training an autoencoder and using it to extract features which are then given to a DRL agent has the advantage of decoupling the feature extraction process from the agent training. To optimize the reconstruction of an image, the autoencoder has to compress redundancy and use the embedding space to store salient features. Since Atari environments have a lot of redundancy, especially in the background, the resulting embeddings should contain useful information for downstream tasks.

For a simple autoencoder composed of a single hidden layer, the encoder with weights $\mathbf{W}_e$ and bias vector $\mathbf{b}_e$ take as input $\mathbf{x}$ followed by an activation function σ_e. The resulting vector $\mathbf{h}$ is then fed to the decoder, resulting in output $\hat{\mathbf{x}}$.

$$\begin{aligned} \mathbf{h} &= \sigma_e(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) \\ \hat{\mathbf{x}} &= \sigma_d(\mathbf{W}_d\mathbf{h} + \mathbf{b}_d) \end{aligned} \tag{1}$$

While training, minimizing the reconstruction error can be viewed as minimizing a euclidean distance like the L2 norm between the encoder input and the decoder output.

$$\mathbb{E}\,||\mathbf{x} - \hat{\mathbf{x}}||_2^2 \tag{2}$$

IV. Results and Discussion

A. Experiments

Tuning the hyperparameters of a neural network based method as DQN often is a tedious work, as not only the reward function and the input representation are to be tuned, but also network design, and classic neural network hyperparameters such as the learning rate.

In our experiments, we focused on two network architectures; a fully-connected neural network that takes the simplified input (5-dimensional vector) as input and a convolutional network that takes a downsampled version of the relevant scene as an input (i.e. as in [2]).

We soon realized that the most important element to tune is the reward function. As discussed in Section III-C, there exist multiple rewards we can think of and finding the right combination and parameterization of the different proposed rewards turns out as the major challenge for tuning this method.

As our initial experiments showed that standing still and waiting for the timer to run out is a often-observed solution with the environment's original reward function, as the negative reward is constant for each time step, we decided to use our own reward functions and discard the environment's reward.

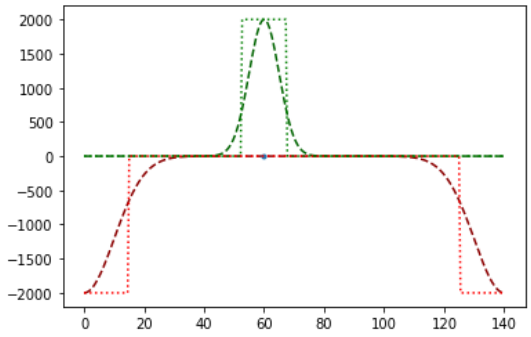

Fig. 4. The dotted lines show a discrete reward and penalty for going through gates and coming too close to the border, respectively. The dashed lines improve on these reward functions by making them continuous with a higher reward closer to the center of the gates and a higher penalty for coming closer to the edges of the screen.

Following our observations, we first designed a discrete reward function that penalizes coming to close to the border and rewards going through gates. To improve on that, we further created a continuous reward function based on Gaussians as shown in Figure 4. Penalties were introduced to penalize the horizontal ski positioning of the skier and possibly reward straight skis, as this position is faster.

Unfortunately, we weren't able with these fixes to find a model that converges to a strategy that performs better than going straight down the mountain. The problem is that this tactic is actually a valid and easy to learn practice to achieve a reasonable success, as the skier will always be close to the gates and actually pass nine gates on his way, while not losing any time to do turns. Of course, the game in the end penalizes missing gates but it still is a problem that needs to be solved to include this knowledge into the learning and actually make the skier turn more often. Penalties for not changing the position over longer sequences didn't help.

An additional difficulty, that we so far didn't cover, is that steering is no direct operation, but one needs to play the same action three times to actually change ski positioning. This is a challenge for reinforcement learning algorithms, as the action does not have a direct impact on the next observed state. To account for this difficulty, we tested the several methods with a modified strategy in which the action that is selected by the agent is performed three times instead of just once.
Using this strategy, we successfully trained an agent that does not miss a single gate and doesn't crash, thus **reaching a near human performance**[3].

The agent completes the course within approximately 40 seconds. We were able to achieve times of around 34 seconds when playing it ourselves but after hours and days of training and finetuning, completing the whole course without missing a gate or even crashing with a gate is a great success. The configuration that led to the successful agent uses 100 episodes, with a memory size of 10000, $\gamma = 0.9999$ and

[3]We saved the weights and share them together with our implementation.

the dimensionality reduction of the states to 5 variables as explained in Section III-B.

B. Result table

Below is the table that summarizes the performances (measured by the average number of poles of the different approaches we took. Of course, these results depend on hyperparameters, such as the number of episodes.

SARSA	Q-learning	DQN
12.2	14.6	20

We observe that tabular methods do not perform as well as the neural network, and we visually see during the runs that they encounter some important issues which do not always get solved by curating a special reward function.

C. Notes on self-supervised feature extraction

The autoencoder pipeline implemented does not allow to extract meaningful features on which to base downstream predictions. For every frame given as input, a very similar noisy reconstruction is generated. The model overfits the data. The frames sampled from the environment are very similar and the pipeline should be further modified to accommodate this, by using different regularization and data augmentation techniques (like dropout, adding noise to input frames, random crops, etc).

The quest for self-supervised feature extraction has been a trend in recent years. The first important benefit would be to have a generalizable approach, avoiding the feature handcrafting time bottleneck, plus having a better resiliency to distributional shifts in the case of online learning. The second advantage, already mentioned previously, would be to decouple the feature extraction from the agent training. This would let us train much smaller controllers (agents), which would imply much shorter training times and open the doors for other kinds of automated learning strategies like neuroevolution [1]. The latter requiring a population of networks for individual training and crossovers. If controllers have lots of parameters, we cannot fit a descent population in memory. An other advantage of having small controllers decoupled from feature extraction is explainability, since the role of each modules would be better defined. Also, composability could be a good quality for reinforcement learning research and industry, in the same way as it plays a key role today in the design of large language models.

D. Notes on other approaches

The task for this report was to use reinforcement learning to teach the computer how to ski. In fact, after having found a way to greatly reduce the input space it would have been possible to create a simple heuristic algorithm to steer the skier. Using this algorithm or some human playthroughs expert examples, it is possible to create an imitation learning based approach, e.g. using DAgger [3]. Our experiments and (only limited positive) outcomes thus also show the difficulty and limits of reinforcement learning based approaches and where other algorithmic solutions might be better suited.

V. Conclusions

The project provided valuable insights into the different methods used in the context of Reinforcement Learning. Applying these methods in practice and exploring their strengths and weaknesses in a challenging environment gave us the chance to dive deeper into the contents of the Advanced Machine Learning course. We are glad that we were able to come up with a DDQN-based agent that successfully masters this difficult environment, using our hand-crafted features for dimensionality reduction and custom reward functions.

References

[1] Samuel Alvernaz and Julian Togelius. Autoencoder-augmented neuroevolution for visual doom playing. 2017.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[6] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.