Using a Dueling Double Deep Q-Learning Approach with Prioritised Memory to Learn a 2D Racing Game - A Project Report

Marcel Stozir*
Dept. of Mathematical Sciences
University of Bath
Bath, United Kingdom
ms3174@bath.ac.uk

I. Problem Definition

Assuming a player (*agent*) controls in some sense an image of a car on an image of a track with the goal of reaching the finish line without hitting the track's borders. This, in essence, can be seen as an example of a *sequential decision problem* as the current position, speed, angular velocity etc. that make up the car's "state" s are inherently determined by previous decisions as, for instance, increasing the controlled acceleration in the previous frame will lead (*transition*) to a higher speed in the current frame. Hence, all previous decisions in frames $1, \dots, t-1$ will sequentially determine the current state s_t based on which an appropriate response (or action a_t) by the agent must be chosen in order to stay on track in an efficient manner. Now, the natural question arises of how the agent should act "optimally" in every possible state, or more rigorously: For a given initial state s_0 what is the best sequence of actions $a_0, a_1, \dots$, the agent should choose in order to reach the goal as quickly as possible?

Sequential problems of this kind involving the *Markov property* are naturally expressed in terms of a *Markov Decision Process* (MDP) for which we need to precisely define the following 5 properties that make up the *environment* with which the agent interacts:

(P1) A set of states S
(P2) A set of actions available in each state $A(s), s \in S$
(P3) A transition function $P(s' \mid s, a)$, $s, s' \in S$, $a \in A(s)$
(P4) A reward function $R(s, a, s')$, $s, s' \in S$, $a \in A(s)$
(P5) An initial state distribution $H(s)$, $s \in S$ about the probability of starting in a given state.

Starting from the ground up, it is firstly important to emphasise that as single frames make up a screen (and any moving object depicted thereon), we can automatically assume the time component in-between states to be finite and given by $t = 0, 1, \dots$. Next we focus on the action component of the MDP. For this matter, on a 2D plane, it suffices to allow the agent to control (i) the acceleration and deceleration component, as well as (ii) the general direction in which the wheels are pointed.

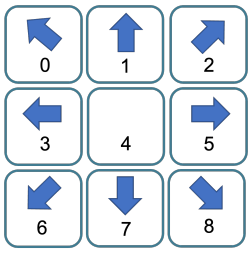


Fig. 1. Illustration of the action space.

The most straight forward way of achieving this is to give the agent access to 9 numerical action values $\mathcal{A} \coloneqq \{0, 1, \dots, 8\}$ as depicted in figure 1. There, any of the numbers $0, 1, 2$ will accelerate the car (until it reaches some specified maximal velocity) where additionally 0 and 2 turn the car (in a circular motion based on the current velocity) left or right, respectively. The same logic applies to indices $6-8$ but now for deceleration instead of the forward motion. In addition, the middle row in this grid allows for directional corrections of the trajectory $(3, 5)$ without changing it's motion or passing completely on choosing an effective control (4); resulting in a linear motion (with slow deceleration due to friction) of the car if this control was chosen while the car was travelling in a circular motion.

Next, in order to choose - in a sensible manner - specific features that define a single state we can already restrict our attention to only those that are affected by and will affect the choice of actions of the car. A very general approach would be to feed the entire screen image over a certain number of frames to the agent. Not only would this provide information about the absolute position of the car as well as the relative position of the car with respect to the track, but also provide

*The author is supported by a scholarship from the EPSRC Centre for Doctoral Training in Statistical Applied Mathematics at Bath (SAMBa), under the project EP/S022945/1.

all necessary information in order to describe quite accurately the exact motion of the controllable object.

Here, however, we simply use a sufficiently large number (20) scalar values $(d_1, \dots, d_D)$ of the distances of the car to the borders for specific, fixed angles around the car. This allows us to omit the necessity of using a *Convolutional Neural Network* (CNN) that receives the entire screen as input. We believe that the chosen approach would lead to a quicker learning of the problem at hand as two similar (or even identical) curves of the track appearing at different positions of the screen will be perceived by the agent as the same; whereas under CNNs (without preprocessing) these images would be treated as a new instance that must be learned.
In order to provide crucial information about the current motion of the car, we additionally give the agent knowledge about the exact velocity in either direction (v^+, v^-) as well as the current force perpendicular to the travel direction (called *(drift) momentum*) in either rotation, denoted by m^+, m^-. Splitting the positive and negative parts is assumed to allow for a more sophisticated weight choice in the networks later on as one edge the input and a neural does not need to simultaneously describe the positive and negative responses. Lastly, to significantly increase the speed of learning while giving the agent the ability to plan ahead, we us (relative) angles a_1, a_2 to the next two "reward gates" (described below) w.r.t. the car's direction. Our hope is that the agent will therefore quite quickly realise that it should follow the direction first directional vector while the second should inform the agent about upcoming curves; and may additionally smooth out the motion of the actions taken.
All in all, we therefore define the state space as

$$\mathcal{S} \coloneqq \left\{(d_1, \dots, d_D, v^+, v^-, m^+, m^-, a_1, a_2)\right\} \subset \mathbb{R}^p \,,$$

but note that all components are bounded (e.g. v^+ by a maximum velocity or d_i by the screen size). In addition, we separately standardise all components by their respective maximal possible value in order to avoid issues due to their different scales.

Lastly, we start the car deterministically in exactly the same position (with no motion), i.e. some state s_0, so that the distribution H in (P5) is trivial, while we also assume no randomness between the actions of the player and the response of the car. Hence, the transition function P in (P3) is determined deterministically by a physical model of a car moving in a (semi-)circular motions with centrifugal forces (drift). However, when the border is touched or the finish line attempted to be crossed from the wrong direction, the next state is automatically a "terminal state" after which the player gets reset to s_0.
Finally, each step (frame) of the agent in any state receives an immediate penalty of $-1/\text{fps}$, while contacting the border is additionally penalised by -10 and the goal of reaching the finish line (in the right direction) rewarded by $+50$. In order to ensure that the player does not kill itself as quickly as possible to avoid further penalty, we place a number (34 to be exact) of so-called "reward gates" along the track with approximately equidistant spacing in order for the agent to receive a reward of $+1$ when crossing any of these gates once.

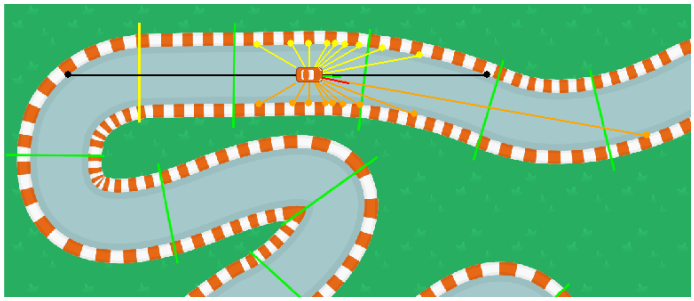


Fig. 2. Illustration of the state space (omitting scalar values for velocity and drift momentum), and reward structure (using green gates and yellow for the finish line). Black, orange and yellow lines and circles emitted from the car indicate the relative distances d_i while the green and red lines point toward the center of the next reward gate and the one after; therefore defining a_1, a_2.

All previous notions about the state-space and reward systems are compactly visualised in figure 2.

II. Background

The previously described problem can essentially be seen as a sequential decision making problem in which the agent (car) interacts with the environment (track) over discrete time steps (picture frames). Hence, we can apply the discrete Markov decision problem theory described by *Sutton & Barto* (2018; [SB18]) in order to describe the objective of the agent.

In particular (see [SB18], [FWXY19] for details), for an agent behaving according to some policy π we can assume the existence of some unknown function $Q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ (called *Q-function*) describing the *value* of any state action pair (s, a) under policy π, and which is given as the solution to the *Bellman optimality equation*

$$Q_\pi(s,a) = \mathbb{E}_{r,s'}\Big[r + \gamma\, \mathbb{E}_{a' \sim \pi(s')}\big[Q_\pi(s', a')\big]\Big] \quad \forall\, (s,a)\,, \tag{1}$$

where for any $(s, a) \in \mathcal{S} \times \mathcal{A}$ the next state s' is distributed according to transition dynamic $P(\cdot \mid s, a)$, and the immediate reward r distributed according to the environment's reward distribution $R(\cdot \mid s, a)$.
Loosely speaking, the objective of *Q-learning* is then to approximate the optimal Q-function Q^* given by $Q^*(s,a) = \max_\pi Q_\pi(s,a)$ for any (s, a), based on a series of interactions (or *trajectories*)

$$\Big\{(S_t, A_t, R_t, S_{t+1}) : t = 0, \dots, T\Big\}$$

of the agent with the environment where T denotes the (possibly ∞-valued) time at which a terminal state is reached.

Given the continuous and hence non-finite nature of the state space, so-called *tabular methods* are no longer applicable to estimate the desired Q-function. In addition, as there is a priori no clear answer on how all components of a given state interact with each other to form its value for the agent, we simply use a *Neural Network* (*NN*) to define the value or Q-function; whose parameters are, in turn, trained in each time-step following the classical *Deep Q-Learning* approach. This procedure is presented in the following section.

A. Deep Q-Network (DQN)

In general (see [FWXY19]), in its most simple form using only sequential neural networks, *Deep Q-learning* starts by assuming that the true value Q^* of taking an action a in a given state s can be approximated sufficiently well by some parameterised (multivariate) decomposition

$$Q(s,a;\,\theta) \coloneqq g_a \circ f_k \circ \cdots \circ f_1(s)\,,$$

where $k \in \mathbb{N}$ is the number of hidden layers in each of which

$$f_i(x) \coloneqq \sigma_i\big(w_i^T x + b_i\big)\,,$$

defines the output of the i-th layer for weights w_i and bias b_i under activation function σ_i, and

$$g_a(y) \coloneqq \sigma_{k+1}\big(w_a^T y + b_a\big)\,,$$

defines the action a specific output function. Then, the parameter θ is given by all trainable weights of the neural network; under the assumption that the number of layers and neurons is appropriately chosen and remain fixed.

In order to train said weights using standard methods of deep learning applied to the reinforcement learning task at hand we need to (i) define a reasonable loss function $L : \Theta \to \mathbb{R}_+$ on the space of parameters based on which sequential parameter updating can be performed, and (ii) describe a sensible procedure of gathering new experiences based on the agent's interactions while it can access learned parameter improvements at the same time.
Omitting the focus on reinforcement learning for now, a standard approach in supervised learning is to minimise the *expected prediction error* under the $\mathcal{L}_2$-norm of a model $f(\cdot;\,\theta)$ for some (assumed) random variables $(X,Y) \sim \nu$ having the relation $f^*(X) = Y$. In other words, some parameter θ is optimal if it minimises the true but unknown expected prediction loss

$$L(\theta) = \mathbb{E}_{(X,Y)\sim\nu}\Big[\big(f^*(X) - f(X;\,\theta)\big)^2\Big]\,.$$

This is accomplished on the basis of a large set $\mathcal{D} \coloneqq \{(x_i,y_i)\}_i$ of independent samples of (X,Y) so that the unknown expected value is assumed to be estimated by the *mean-squared error*

$$\hat{L}(\theta) \coloneqq \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f(x_i;\,\theta)\big)^2\,.$$

In our case for Markovian processes, we may similarly try to minimise the $\mathcal{L}_2$-prediction error of Q in the sense that for any initial state $s_0 \in \mathcal{S}$ the optimal parameter is defined as the one minimising (inspired by [WSH$^+$15])

$$L(\theta) \coloneqq \mathbb{E}_{\nu_\pi(s_0)}\Big[\big(Q^\pi(s,a) - Q(s,a;\,\theta)\big)^2\Big]\,,$$

where $\nu_\pi(s_0)$ should denote the true, unknown stationary distribution on the space of all state-action pairs (s,a) when the process is started in s_0 and follows the policy π, i.e. the probability of reaching s from s_0 when following policy π along the way.
Although this general description is not particularly useful at first glance - as it essentially requires solving the Bellman optimality equation to obtain ν_π - it provides a natural way of motivating the sampling of experience later on. For this, we assume the t-th step of the agent in the environment following policy π has produced (S_t, A_t, R_t, S_{t+1}), and that the current estimator of θ is given by some θ_t. Then, as R_t and S_{t+1} are real observations from the environment, at time t the agent's belief (or estimate) about the value of (S_t, A_t) is given by

$$\widehat{Q}_t(S_t,A_t) \coloneqq R_t + \gamma \max_{a'} Q(S_{t+1},a';\,\theta_t)\,,$$

so that - similarly to the approximation in the standard supervised learning case - we can estimate the expected prediction error using enough independent samples $\mathcal{D} \coloneqq \{(s_i,a_i,r_i,s_i')\}_{i=1,\dots,n}$ of the interactions with the environment - where index i is simply a labelling of the data set and not a time-step any longer - by

$$\hat{L}(\theta_t) \coloneqq \frac{1}{n}\sum_{i=1}^{n}\big(y_i^{DQN} - Q(s_i,a_i;\,\theta_t)\big)^2\,,$$

where $y^{DQN} \coloneqq r_i + \gamma \max_{a'} Q(s',a';\,\theta_t)$. However, in contrast to the standard deep learning methodology, the target value (y^{DQN}) in this current form is inherently dependent on the chosen parameter - leading to stability issues of the resulting learning behaviour [WSH$^+$15]. To combat this issue, *Mnih et al.* (2015; [MKS$^+$15]) rather propose to use a fixed parameter $\theta^- \coloneqq \theta_t$ for a fixed number τ of learning iterations in order to obtain $y_1^{DQN}, \dots, y_n^{DQN}$, before updating θ^- to the new estimator $\theta_{t+\tau}$ for the next learning phase. This is accomplished by defining a separate *target network* with the exact same layout as the trainable network (*primary network*), but whose parameters remain fixed at $\theta_{m\,\tau}$, $m = t - t \bmod \tau$, for all subsequent τ iteration steps.

On this basis we can proceed under the assumption that $\widehat{Q}$ is independent of the parameter changes in the primary network. Then, the theory of gradient descent with learning rate $\alpha \in (0,1)$ suggests updating of θ_t according to the rule (assuming dominated convergence of $Q(\cdot;\,\theta_t)$ to apply)

$$\begin{aligned}\theta_{t+1} &\coloneqq \theta_t - \alpha\,\frac{1}{2}\,\nabla_{\theta_t} L(\theta_t)\\ &= \theta_t + \alpha\,\mathbb{E}_{\nu_\pi}\Big[\big(\widehat{Q}_t(s,a) - Q(s,a;\,\theta_t)\big)\,\nabla_\theta\, Q(s,a;\,\theta_t)\Big]\,,\end{aligned}$$

which - according to the theory of stochastic gradient descent - can iteratively be updated based on single observations (or batches thereof) of some data set $\mathcal{D}$, i.e. ([vHGS15])

$$\theta_{t+1} \coloneqq \theta_t + \alpha\big(y_i^{DQN} - Q(s_i, a_i;\, \theta_t)\big)\, \nabla_\theta\, Q(s_i, a_i;\, \theta_t)\,,$$

where, again, y^{DQN} is the *target value* defined as

$$y_i^{DQN} \coloneqq r_i + \gamma \max_{a'} Q(s_i', a';\, \theta^-)\,. \tag{2}$$

Prioritised Experience Replay (PER)

Up to this point we have omitted any discussion about the previously mentioned second necessity ("(ii)") for deep Q-learning, namely how the learner obtains the data set $\mathcal{D} \coloneqq \{(s_i, a_i, r_i, s_i')\}_{i=1,\dots,n}$ used to train the neural network; let alone addressed the inherent assumption about independence of the samples while interactions with the environment are clear dependent due to the Markovian nature of the process.
For this matter, we realise that in n' interaction episodes of the agent following some evolving policy will produce a data set $\mathcal{D}' \coloneqq \{e_1, \dots, e_n'\}$ of experience trajectories $e_i \coloneqq \{(s_t^i, a_t^i, r_t^i, s_t'^i)\}_{t=0,\dots,T_i}$ where T_i, again, indicates the terminal state time which is assumed to be finite. Hence, after relabelling we can assume WLOG that a data set $\mathcal{D}_n \coloneqq \{(s_i, a_i, r_i, s_i')\}_{i=1,\dots,n}$ of size $n = \sum_j T_j$ is available to train the network. Then, in order to negate effects of correlation of the samples, in each learning step a small batch size $k \ll n$ will be drawn independently at random from the data set (or *memory*). This method, called *experience replay* ([Lin93], [MKS+15]), then produces the theoretical loss function

$$L(\theta_t) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\Big(y^{DQN} - Q(s, a;\, \theta_t)\Big)^2\right],$$

which is approximated based on a batch size of $k \ll n$ by

$$\widehat{L}(\theta_t) = \frac{1}{k}\sum_{i\in\mathcal{I}} \big(y_i^{DQN} - Q(s_i, a_i;\, \theta_t)\big)^2\,,$$

where $\mathcal{I}$ is a uniformly random sample of size k without replacement from $[n] \coloneqq \{1, \dots, n\}$.

This last equation clearly suggests that all selected samples are treated equally in the updating step due to the uniform *weighting* of $1/k$. In reinforcement learning however it may be crucial to learn as quickly as possible certain (or rare) true state-action pair values whose current value estimate is far off. For instance, in our car driving game later stages of the track become increasingly difficult. Hence, in order to reliably navigate those parts the neural network needs to be trained on large amounts of samples from these areas, but as the time it takes the agent to reach these parts is very long, these experiences make up only a very small portion of the overall data set so that uniformly selecting them at random is a *rare event*.
In order to combat this issue, *Schaul et al.* (2015; [SQAS15]) propose to use *prioritised experience replay* (*PER*) for the sampling of past experiences. In essence, this method is an instance of *importance sampling*, and suggests that all experiences should (in a consistent manner) be weighted according to their discrepancy between the then assumed value $\widehat{Q}(s, a)$ and the network's predicted value $Q(s, a;\, \theta)$. Namely the probability $p(e)$ of choosing a particular experience $e = (s, a, r, s')$ obtained at time t should be proportional to the absolute error

$$\Big|\widehat{Q}(s, a;\, \theta^-) - Q(s, a;\, \theta_t)\Big|\,,$$

where θ^- denotes the target network's weight used at time t. Simply setting p_i as this absolute error when the i-th of n samples in the data set was obtained allows us to sample this exact experience in a consistent manner using the probability measure on $\mathcal{D}$ given by

$$p(e_i) \coloneqq \frac{p_i}{\sum_j p_j} \qquad \forall\, i = 1, \dots, n\,,$$

after which the weights are chosen in the weighted importance sampling manner using

$$w_i \coloneqq \big(n\, p(e_i)\big)^{-\beta}\,,$$

where $\beta \in (0, 1]$ is a parameter controlling the non-uniformity bias, where $\beta = 1$ corresponds to a purely uniform weighted choice (see [SQAS15] for details).

B. Double DQN (DDQN) and Dueling DDQN (D3QN)

At this point the standard deep Q-learning methodology was presented. The remainder of this section is therefore only concerned with presenting promising improvements upon this approach.
Firstly, it is a well-known problem of the standard Q-learning procedure that using the same approximated Q-function to select (via $a_{t+1} \coloneqq \arg\max_{a'} Q(s_{t+1}, a';\, \theta)$) and evaluate the choice (with $r_t + \gamma Q(s_{t+1}, a_{t+1};\, \theta)$) will lead to the so-called *maximisation bias* (see [SB18] §6.7). A common way to mitigate this problem is to use another learnable Q-function Q_2 to evaluate the choice of the primary one Q_1 using the value

$$\widehat{Q}(S_t, A_t) \coloneqq R_t + \gamma\, Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big)$$

across both functions to evaluate state-action pair (S_t, A_t); while both are chosen for either purpose equally likely and updated accordingly using *double Q-learning* (see [SB18]).

Now, this maximisation bias is also present in deep Q-learning leading to overly optimistic value estimates ([vHGS15]). In order to combat this issue, *van Hasselt et al.* (2015; [vHGS15]) therefore propose to utilise the target network also as the double Q-learning function in the sense that instead of using y_i^{DQN} from (2) as the estimated current value of the i-th state-action pair one should use

$$y_i^{DDQN} \coloneqq r_i + \gamma\, Q\big(s_i',\, \arg\max_{a'} Q(s_i', a';\, \theta_t);\, \theta^-\big)\,, \tag{3}$$

so that - similarly to the double Q-learning proposed in Sutton & Barto - the primary network (given by $Q(\cdot;\, \theta_t)$) is used to choose the next "best" action while the target

network (via $Q(\cdot;\,\theta^-)$) is used to produce an evaluation of the current value of the state. As the target value receives the updated parameters of the primary network after a certain number of iteration steps (or continuously in a soft-updating manner), both networks can therefore be seen as converging to the same true, unknown Q^* as long as convergence can be ensured.

As a last improvement of this currently described DDQN procedure, we turn to the most commonly used (former state-of-the-art) architecture called *Dueling DDQN* proposed by *Wang et al.* (2016; [WSH+15]). In general, the underlying idea is that with some additional but separate hidden layers to the basic structure of a DQN one can simultaneously estimate the value function $V(S)$ of a given input state S as well as the Q-function for any action in that state. Specifically, as visualised in figure 3, let the standard DQN's architecture similarly to (1) be given by

$$A(s,a;\,\theta,\alpha) \coloneqq g_a \circ f_{k,1} \circ \cdots \circ f_1(s)\,,$$

where θ now describes all weights and biases appearing in the first $k-1$ hidden layers while α holds all weights in the first (of currently 1) last hidden layer and the $|\mathcal{A}|$-sized output layer of the action values; now denoted as A-function. Then, separately from the first k-th hidden layer one creates another one which is connected to f_{k-1} and produces a scalar output, i.e. a secondary stream of the neural network is described by

$$V(s;\,\theta,\beta) \coloneqq h \circ f_{k,2} \circ f_{k-1} \circ \cdots \circ f_1(s)\,,$$

for some output function $h(y) \coloneqq \sigma^V_{k+1}(w_V^T y + b_V)$, and β describing all weights and biases between connected to $f_{k,2}$ and h.
Hence, we have just produced a neural network still with k hidden layers, with the same input dimensions as before but output dimension $|\mathcal{A}|+1$.

Now, from the theory of Q-learning, under a given policy the true value V^π of a state s - generally measuring the expected future rewards following the actions from this state - is given by the expected value of $Q(s,a)$ where the action is chosen according to π, i.e.

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}\big[Q^\pi(s,a)\big] \qquad \forall\, s \in \mathcal{S}\,.$$

Hence, the idea of Wang et al. is to define the so-called *advantage function* by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \qquad \forall\,(s,a) \in \mathcal{S}\times A\,,$$

in order to define - under a given policy π - the "advantage" of choosing a particular action a in a given state; where the π-implied choice makes up the overall value of this state, i.e. $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. This can be seen therefore be seen as a centered representation of the Q-function if treated as random variable under stochastic policy π.
As neither $V(\cdot;\,\theta,\beta)$ nor $A(\cdot;\,\theta,\alpha)$ necessarily represent the true state-value or advantage function we cannot ensure that

$$V(\cdot;\,\theta,\beta) + A(\cdot;\,\theta,\alpha) =: Q(\cdot;\,\theta,\alpha,\beta)\,, \tag{4}$$

describes a reasonable approximation of the Q-function even if the parameters are optimally chosen. For one, as Wang et al. describe, for a given Q-function the individual parts V and A making up (4) cannot be recovered uniquely as $V' \coloneqq V + c$ and $A' \coloneqq A - c$ for any constant c will produce the same Q-function.

Thus, it is required to have an additional equation in order to ensure uniqueness. One such choice is implied by the previously mentioned theoretical result that the expected (or mean) advantage of any particular action should remain 0 in any state, i.e. (at least under a purely exploratory policy) that A should satisfy

$$A^\pi(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A^\pi(s,a') = 0 \qquad \forall\,(s,a) \in \mathcal{S}\times\mathcal{A}\,.$$

Although to the best of our knowledge there does currently not exist a rigorous theory stating that this equation also holds under the learned off-policy strategy, we still apply this action-independent constant in the aggregation layer of the dueling DQN using

$$Q(s,a;\,\theta,\alpha,\beta) \coloneqq V(s;\,\theta,\beta) + \left(A(s,a;\,\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\,\theta,\alpha)\right), \tag{5}$$

as it does not affect the choice of the (ε-)greedy strategy based on the obtained Q-function estimate, as

$$\arg\max_a Q(s,a;\,\cdot) = \arg\max_a A(s,a;\,\cdot)\,, \tag{6}$$

holds; while simultaneously the application in (5) ensures consistency when a purely exploratory strategy is followed initially.

III. Method - Experimental Implementation

In this section we present a comprehensive but high-level overview of the experimental implementation; the complete Python code and further materials are freely accessible on GitHub.

As previously described, we assume a basic sequential (dueling) DDQN architecture as depicted in figure 3, where (based on significant tests) the choice of 2 fully connected hidden layers consisting of 32 and 16 neurons, respectively, appear to produce the most promising results for the problem at hand. In contrast to [WSH+15], we intentionally omit a separate, additional hidden layer for each stream of the dueling architecture in order to compare the dueling and vanilla DDQN approach solely based on the implications of using an additional scalar output; which is used to estimate the value function in the dueling version while the vanilla DDQN does not attempt this. This general architecture is simply implemented using *tensorflow 2.0*'s *keras* library, where the hidden activation functions are chosen to be ReLu while the output layers use linear relations.

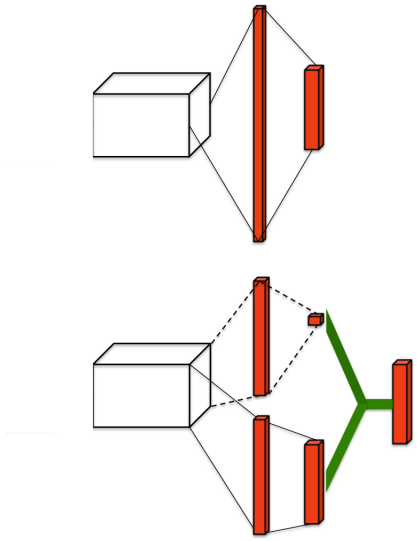


Fig. 3. Top: Standard DQN layout where the box represents hidden layers $1, \dots, k-1$ and the right-most layer the output layer of size $|\mathcal{A}|$. Bottom: General architecture of a dueling network where the last hidden layer of the standard DQN is separated into two streams. The top one is then utilised to estimate a single scalar output (small rectangle) while the lower of the two produces the same $|\mathcal{A}|$-dimensional output vector as in the standard DQN which is now the A-function estimate. The green lines connecting the two finally produce the desired Q-function. (Source: [WSH+15])

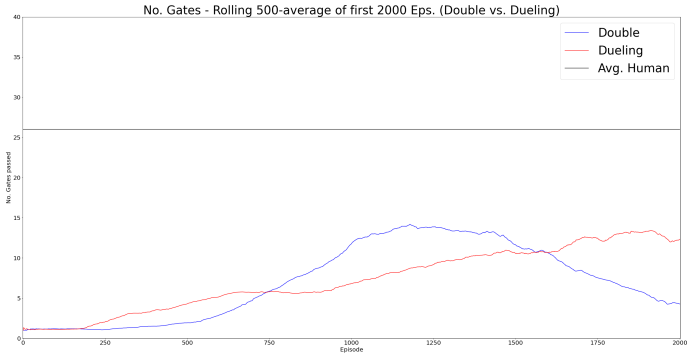
Then, in order to efficiently sample from the memory holding a large number of past experiences in a prioritised manner, a *SumTree* method proposed by Jaromír Janisch is implemented. A detailed description of it's advantages in given here but, in short, a binary tree with N_r initially empty leafs is created where N_r is the total size of the memory (or data set). Assuming a new experience is stored - from left to right in the leaf indices - it receives an a priori maximal initial priority value (previously described as weight w_i) to be sampled; as no absolute error of it's learning improvement exists before it was sampled at least once. Then, all parent nodes of the leafs hold the probability of sampling either of it's child leafs, and this procedure is sequentially repeated until the root node holds the total sum of priority values of all leafs. Now, in order to sample from the data set in a prioritised manner, a uniformly random value x is drawn from $[0, p_{max}]$ where p_{max} is the total priority in the tree. Starting at the root, and assuming WLOG the left child to have a smaller priority value $p_L < p_R$ than the right, then the left child is chosen if $x \leq p_L$, and right is chosen otherwise. Assuming $x \leq p_L$, we update $x \leftarrow p_{max} - x$ and $p_{max} \leftarrow p_L$, and iteratively follow the exact same binary left or right path down the tree until we reach a leaf. This leaf is then returned, and - based on this method - it can be shown that this has a probability of exactly it's current relative priority value to be chosen amongst all other leafs.

Now that the basic methods are well-defined, the general approach is as follow (see fig. 5 in Appendix A for specifics): After an initial period in which the agent interacts completely at random with the environment designed to fill the memory with enough samples (3 times the batch size), the agent starts at the fixed initial position s_0, chooses an action according to an ε_1-greedy strategy based on the current $Q(s_0, \cdot; \theta_t)$-value for this state (according to (5), or directly using (6)), and receives the reward r_0 and new (deterministic) state s_1. This experience $e_1 := (s_0, a_0, r_0, s_1)$ is stored in the memory, and a batch of k (here 64) old experiences is chosen using the previously described prioritised memory.
After calculating the target values y_i^{DDQN} for each of the experiences that did not produce a terminal state according to (3), and setting $y_i^{DDQN} = r_i$ otherwise, a batch-wise updating step of the parameters is produced in the usual deep learning manner. There, each of the samples of the batch is weighted by its current priority value, which is subsequentially updated according to the corresponding learning improvement (via absolute error) it produced.
Then, the current state is defined as s_1 and the entire procedure is repeated until a terminal state has been reached. This triggers the resetting of the environment so that the agent can repeat the game again, while having access to all previous experiences and the current DQN parameter estimates. Driven by the number of interactions with the environment, the exploration probability ε_n continuously decays (exponentially) to some limit value (here 0.01); while - as previously noted - the target network's parameters receives the primary network's ones after a fixed number of learning iterations (here every $10,000$ interactions).

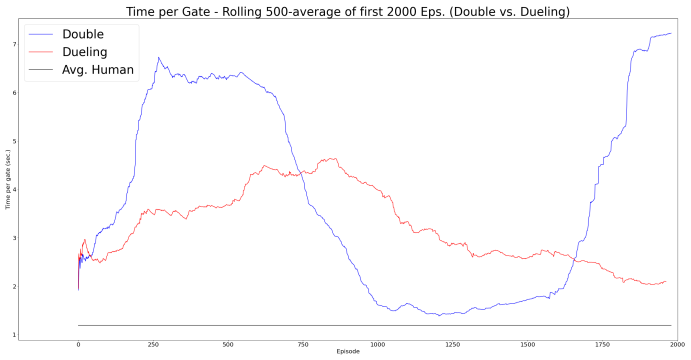
IV. Results

Based on the previously presented methodology we choose the specific parameters chosen as presented in Appendix A, and separately train the double and dueling-double networks for a number of $2,000$ repetitions of the game while continuously saving, amongst other performance indicators, (i) the number of reward gates the agent has crossed, and (ii) the number of steps it has used; each until the termination state. In our opinion, as the reward system is primarily used as an immediate feedback-loop to the agent, these two numerical values indicate most clearly how well the agent has learned to navigate the track in a forward motion. Thus, we can simply calculate the "avg. time per gate" using the number of reward gates crossed divided by the total steps taken and scaling it by the known (mean) frames per second. In addition to the number of gates crossed, this value indicates how "decisive" the agent has acted in between two reward gates across the entirety of a single repetition of a game.
As batch-wise deep learning suffers from significant noise in the learning trajectory, in order to visualise how well the agents have learned, their performance indicators are given in the form of a rolling 500-episode average in figure 4 besides an "average human" baseline that was calculated based on the performance of a real player averaged across 5 games.

As a first observation we note that figure 4a indicates that the dueling double agent overall learns more consistently



(a)



(b)

Fig. 4. Top: Number of gates crossed. Bottom: Average time between two reward gates. Each is calculated using a rolling 500-average across the value obtained in each episode by the dueling double (red) and vanilla double (blue) learning agent across $2,000$ repetitions of the game. In black the average human performance is presented as the desired baseline.

compared to the vanilla double approach. The latter, although learning more rapidly after an initial burn-in period[1], suffers from stability issues past $1,000$ episodes. One explanation for this issue may lie in the known problem of deep learning called the *catastrophic forgetting* ([KPR+17]) where, in our case, the agent has prioritised training for later stages of the game so much that it forgets how to navigate the more simple initial stages; leading to a collapse of the overall performance. This instability issue may very well appear in the D3QN training but, as it holds more trainable weights, it is expected to appear later which was not captured in this (arguably short) learning horizon.
Still, both agents fall very short (max. average of 15) from the average human baseline which reached an average of 26; although a significantly longer training horizon may result in a much better average performance.

Regarding figure 4b - now neglecting the initially significant discrepancy and later collapse of the DDQN agent past $1,000$ episodes - we see that it again learns much more rapidly between episode 500 and $1,000$ to quickly navigate the first half[2] of the course while the D3QN agent improves at a more consistent, but overall slower speed and does not reach the DDQN agent's best time performance in the depicted learning horizon; which appears to be very close to the average human speed.

V. Discussion and Future Work

We have defined the MDP process for a standard 2D racing game, and presented in a comprehensive way how a (dueling) double deep Q-learning procedure may be used to obtain an "optimal" policy for an artificial agent interacting with the environment.
In particular, we have separately trained both a double DQN-agent as well as a dueling double DQN-agent that only differed by a single output component (the state value estimate). Performance graphs revealed that a number of $2,000$ repetitions of the game is too few to obtain a near-human level behaviour. Still, this small sample size provided evidence that suggest that the D3QN algorithm may have a more stable learning curve; although a counter argument that this model might also suffer from catastrophic forgetting was presented.

Besides these statistical error computations, we turn to an analysis of the real-time performance under the obtained greedy-strategy (see GitHub/video_performance/) for two representative agents[3]. There, both show an inconsistent trajectory behaviour in the sense that they both rapidly change their previous directional choice (specifically when traversing along a curve). This implies that both procedures overvalue the given directional state to the next gate, and that they lack the ability to plan ahead for more than one target[4] in a "natural" way. Although the D3QN-agent positions itself on the opposite side before an upcoming curve (i.e. plans ahead), it does so for any current velocity (even near 0). This suggests that the obtained neural network overvalues the positional arguments in the state value estimate while not capturing the necessary velocity dependency.
This lack of "natural behaviour" may be improved using *Watkins's* $DQN(\lambda)$ approach (see [SB18] §12.10) when saving the obtained experiences along a trajectory. In essence, similarly to $TD(\lambda)$ for tabular methods, so-called *eligibility traces* are used to retroactively reward (or penalise) all previous decisions in a certain trajectory that lead to obtaining a particular reward (or loss). Hence, the agent is artificially trained (or rather biased) to follow a clear path to obtain a certain reward. Applying this procedure to the problem at hand remains a task for a future project as - to the best of our knowledge - currently no efficient way exists to merge "Tree-Backup(λ)" memories with a prioritised sampling method in an efficient manner.

[1]This discrepancy in the initial learning behaviour may be caused by different random initial weights, and should not be seen as clear evidence that the dueling DDQN learns more quickly.

[2]Overall, the track is made up of 34 reward gates so that an average of 15 can be seen as an approximate halfway point.

[3]Specifically we focus on the Double DQN-agent in eps. 679 (before instability) and the Dueling Double DQN-agent in eps. 1684.

[4]For instance by accessing the angle to gate after next.

APPENDIX

A. Pseudocodes

The basic structure of a double DQN is given in fig. 5. As a clear overview was presented in section III, a comprehensive discussion is omitted. The interested reader can find further details in [vHGS15].

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples (s, a, r, s') from $\mathcal{D}$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Fig. 5. Standard pseudocode for double DQN-learning. (Source: [WSH+15])

In combination with this standard DDQN approach the additional sampling (and memorising) procedure under prioritised experience replay is presented in figure 6. A rigorous discussion about the specifics that were not presented in section II is beyond the scope of this work and can be found in [SQAS15]. As previously mentioned, the dueling approach differs only by a subsequent aggregation of the advantage function and the value function estimates, as well as how the best action is chosen. Hence, the basic code structure remains the same while specific changes are presented in [WSH+15].

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Fig. 6. Double DQN-learning with prioritised memory buffer. (Source: [SQAS15])

REFERENCES

[FWXY19] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning, 2019.

[KPR+17] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[Lin93] L. J. Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[WSH+15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.