Pareto-DQN: Approximating the Pareto front in complex multi-objective decision problems

Mathieu Reymond
Vrije Universiteit Brussel
Brussels, Belgium
mreymond@ai.vub.ac.be

Ann Nowé
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

In many real-world problems, one needs to care about multiple objectives. These objectives can be contradicting and, depending on the decision maker, the different compromises will be ranked differently. In this preliminary work, we propose a novel algorithm: Pareto-DQN, that will estimate the Pareto front of complex environment, with a high-dimensional state-space. As a proof-of-concept, we successfully apply our algorithm to the Deep-Sea-Treasure environment, a well known Multi-objective reinforcement learning benchmark.

KEYWORDS

Reinforcement Learning; Deep Reinforcement Learning; Multi-objective Reinforcement Learning

1 INTRODUCTION

Many real-world decision problems require to cope with multiple, possibly contradicting objectives. When investing in the stock market, for example, one typically wants to execute a strategy that will maximize profit, while at the same time incur no risk of losing the investment. This is often not possible and, depending on the decision maker, the relative importance of each objective varies. As opposed to single-objective optimization, e.g., maximizing one's score on Atari's Breakout game, no single best solution can be found for these multi-objective problems. Instead, we deal with a set of possible compromises. Each member of this set is called non-dominated when no single objective can be improved without harming the other ones. The set of all non-dominated solutions for a particular problem is called the Pareto front.

In this paper, we will describe how we will estimate the Pareto front for complex, high-dimensional multi-objective problems so that it can be used by the decision maker to execute an informed strategy.

A straightforward approach to multi-objective optimization is to scalarize the different criteria, effectively reducing the problem to a single-objective one. Most often, a weighted sum over the objectives is used [1, 7, 8] but, as this only allow for solutions on the convex parts of the Pareto front [2], non-linear scalarizations have been investigated too [18]. As a downside, the weights of the scalarization function need to be decided upon a priori to reflect the type of solution desired. First of all these weights might not reflect the true preference of the user. Moreover, this approach suffers from instability, as a small change in weights might lead to drastically different solutions [15].

Another method is to search for the Pareto front, and let the decision maker choose its preferred solution a posteriori. This is preferred, as the solutions on the Pareto front directly capture the trade-offs between the objectives. However, the downside of this approach is that it comes at a higher computational cost.

One algorithm that directly tries to find the Pareto front is called Pareto-Q-Learning (PQL) [19], and extends the classic Q-learning algorithm [22]. However, like Q-learning, PQL can only perform in an environment with a small-sized state-space. Currently, much of the work done in Multi-Objective Reinforcement Learning has been performed in a low-dimensional (although sometimes continuous) state-space setting [5, 10, 12, 16, 21]. In contrast, single-objective Reinforcement Learning is able to operate well in complex, high-dimensional problems, sometimes exceeding human-level performance [9, 13, 20]. The incorporation of Deep Learning techniques as a means of scalability towards high-dimensional state-spaces plays a major role to this success, with as one of the first notable examples Deep Q-Networks (DQN) [6].

Inspired by PQL, we will extend the DQN algorithm to cope with multiple objectives. For each state-action pair, our algorithm, denoted Pareto-DQN (PDQN), will output the corresponding Pareto front, instead of the scalar Q-values that DQN would predict. We will show preliminary results of our method on two different experiments: one proof-of-concept, standard multi-objective benchmark, and another one characterized by a high-dimensional state-space.

2 BACKGROUND

In order to explain PDQN, we will first elaborate the DQN and PQL algorithms.

2.1 Reinforcement Learning

Reinforcement learning (RL) [14] is a machine learning technique that allows an agent to learn by trial-and-error while interacting with an environment, given some numerical feedback known as a reward signal. The environment is modelled as a Markov decision process (MDP) $M = (S, A, T, \gamma, R)$ [11], where S, A are the state and action spaces, $T\colon S \times A \times S \to [0, 1]$ is a probabilistic transition function, γ is a discount factor determining the importance of future rewards and $R\colon S \times A \times S \to \mathbb{R}$ is the immediate reward function. The reinforcement learning agent needs to learn a policy π, i.e. a mapping between states and actions that maximizes the discounted sum of received rewards, i.e., returns.

The Q-learning algorithm [22] iteratively approximates the expected returns using the following update-rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a, s') + \gamma \max_{a' \in A} Q(s', a')) \tag{1}$$

For many real-world problems the state-space becomes too large to keep track of all the Q-values. To alleviate this issue, the DQN

algorithm approximates those high-dimensional states by feeding them to a neural network, which outputs Q-values for these approximations. The network is trained by performing gradient descent, using a Mean-Squared Error loss with $R(s,a,s') + \gamma \max_{a' \in A} Q(s',a')$ as target. However, the non-stationarity of the target Q-function introduces instability in training.

DQN tackles this problem by including a target network $\hat{Q}$ and an experience-replay buffer. Executed transitions (s_t, a_t, r_t, s_{t+1}) are collected by the experience-replay buffer D, which is then uniformly sampled to generate training batches. Additionally, $\hat{Q}$ is used to estimate the target Q-values, resulting in the following update-rule:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \max_{a' \in A} \hat{Q}(s',a') - Q(s,a))^2] \quad (2)$$

Where θ are the parameters of the Q-network. Finally, the weights of the Q-network are periodically copied to the target network.

2.2 Multi-objective Reinforcement Learning

In single-objective RL algorithms, the agent is given a scalar reward for each performed action. Instead, Multi-objective RL (MORL) possesses a vectorial reward function $R\colon S \times A \times S \rightarrow \mathbb{R}^d$, where d is the number of objectives. Thus, the expected return is also in vectorial form, which means we cannot straightforwardly apply the classic Q-update rule, as the maximum of a set of vectors is unclear, e.g., two distinct non-dominated returns might have a different priority depending on the user.

As such, Roijers et al. [12] define two main approaches towards MORL algorithms: the outer-loop and the inner-loop method. In the outer-loop approach, a single-objective RL algorithm is repeatedly applied on a different scalarization of the reward function. We then keep track of all the discovered policies that lead to non-dominated returns. As opposed to that, the inner-loop method modifies the workings of a single-objective algorithm to use set-operations and prune away dominated solutions.

This is the approach taken by PQL. For each state-action pair, a set of non-dominated solutions is kept (initially empty), which is updated using the non-dominated set of the next state over all possible actions. More formally, the set of expected returns for a state-action pair is defined as:

$$Q_{set}(s,a) \leftarrow \overline{R}(s,a) \oplus \gamma ND_t(s,a) \quad (3)$$

where $\overline{R}$ is the average immediate reward, ND_t is the set of non-dominated future returns and $\oplus$ is a vector-sum operation ($\overline{R}$ is thus added to each element of ND_t). As opposed to Q-learning, where a scalar Q-value is updated using another scalar ($\max_{a' \in A} Q(s',a')$), PQL updates a set of Q-vectors with another set. However, it is unclear which element of $Q_{set}(s,a)$ to update with which element in s'. To cope with this lack of correspondence between the elements of different Q_{set}'s, we keep track of both $\overline{R}$ and ND_t separately (for more details, we refer to the work of Van Moffaert and Nowé [19]). $Q_{set}(s,a)$ can then be reconstructed at any time, using equation 3. As ND_t represents the future returns, it is updated using the non-dominated set of all the Q_{set}'s of the next state:

$$ND_t(s,a) \leftarrow ND(\cup_{a' \in A} Q_{set}(s',a')) \quad (4)$$

Or, equivalently:

$$ND_t(s,a) \leftarrow ND(\cup_{a' \in A} \overline{R}(s',a') \oplus \gamma ND_t(s',a')) \quad (5)$$

Where ND is a function that removes all dominated vectors from the set. Additionally, $\overline{R}$ is updated using:

$$\overline{R}(s,a) \leftarrow \overline{R}(s,a) + \frac{R(s,a) - \overline{R}(s,a)}{n(s,a)} \quad (6)$$

Where $n(s,a)$ returns the number of times a was performed in s.

The Q-update rule has thus been modified to cope with vectorial rewards. However, the action-selection mechanism (typically ϵ-greedy) also depends on the Q-values. To apply such a mechanism on PQL, an indicator-measure will be applied to assess the quality of a given Q_{set}. The hypervolume is one such an indicator: it computes the total volume under all the points in Q_{set} for a given reference point. The greedy policy then chooses the action with the highest hypervolume.

3 METHOD

Similarly to classic Q-learning, the main issue with PQL lies in the need to keep a set of non-dominated points for each possible state-action pair. As this is an unrealistic assumption, we will use a neural network to approximate the Pareto front. Moreover, another approximator will be used to estimate the average immediate reward. We will combine the additions proposed by DQN to scale towards high-dimensional state-spaces as well as the modifications on the update-rule and action-selection mechanisms used in PQL with a novel way estimate the Pareto front into a new algorithm: Pareto-DQN.

3.1 Estimating the immediate reward

As we need to keep track of the immediate average reward $\overline{R}$ separately, we will approximate the function with a neural network. Given a state and an action as inputs, it will output a vector $r \in \mathbb{R}^d$, corresponding to the estimated reward. Contrary to Q-values, the immediate reward is a stationary target. Therefore, we will thus omit a separate target network and directly update the current network with the perceived rewards. Due to the nature of backpropagation, the output for a given state-action pair will automatically be the average reward after convergence.

3.2 Estimating the non-dominated set

Next to $\overline{R}$, we will approximate ND_t. Due to this set being of a variable size, i.e., it contains a different number of non-dominated returns depending on the state, we cannot simply use state-actions as inputs and output a fixed number of points. Even if we bounded the set to the p best points, the ordering of these points would matter. Indeed, in order to appropriately update the network, each output neuron needs to be compared with the same target-value (provided the input is the same). A change of ordering would result in a change of target and thus hinder training. However, keeping track of this ordering is not a viable solution: each time a new non-dominated point is discovered (and this is state-dependent), an existing one should be removed, resulting in a change of ordering.

To cope with these issues, we use a network that not only takes a state and action as input, but $d-1$ additional values $o_1 \ldots o_{d-1}$, corresponding to all but the last objective. The output is then the

predicted value for the remaining objective o_d. Combining o_d with the inputs $o_1 \dots o_{d-1}$ thus yields a single point for (s, a) in $\mathbb{R}^d$. Adding $\overline{R}(s, a)$ to that point (equation 3) results in a single point on the Pareto front. Approximating the whole Pareto front is then performed by sampling p points from $\mathbb{R}^{d-1}$, predicting o_d for each sample using the $ND_t(s, a)$ predictor, and then appending those predictions to their corresponding $o_1 \dots o_{d-1}$ point.

However, there are some implications resulting from this architecture. First of all, the Pareto front of some states might only span over a subdomain of $\mathbb{R}^{d-1}$. This subdomain is, however, unknown, meaning we will always sample from the whole $\mathbb{R}^{d-1}$ domain and make predictions for points that cannot possibly exist on the Pareto front. This limitation is overcome by training our predictor to output values worse than the smallest possible reward for these points (see figure 1a). Moreover, we observed that only updating the network with newly discovered non-dominated points lead to unstable behavior, as these updates would affect the predictions for other points. As a result, the entire Pareto front is always updated, by using the predictions of those other points as their own targets, to enforce them to remain unchanged.

Nevertheless, there are two cases where points need to be artificially added to cover the whole $\mathbb{R}^{d-1}$ domain. First, when reaching a terminal state, the Pareto front only consists of the received (terminal) reward. Second, after applying equation 3, the range covered by ND_t got shifted by $\overline{R}$. In both cases, points are sampled from the uncovered parts of the $\mathbb{R}^{d-1}$ domain. For each point, the chosen target value will be a value worse than the least possible reward for objective d (e.g. the value of the reference point in dimension d) if the point is non-dominated. On the contrary, if the sample is dominated by a point, the target value will be that point's value (see figures 1b-1c). This ensures our Pareto front remains unchanged, while still sampling from the whole domain.

3.3 Evaluation policy

During training, the actions the agent takes depend on the Pareto front's hypervolume. In contrast, once the Pareto front is known, the user will select his preferred point, and expect the agent to reach it. PQL does this by following that point during the episode. At each timestep, we compute, for every possible action, the corresponding Q_{set}. One of the sets contains the target point (computed using $\overline{R}$ and a point of ND_t). The agent then takes the matching action, and replaces the target value with the appropriate point from ND_t. This is performed until termination, at which point the target has been reached (and is equal to the discounted sum of all the computed $\overline{R}$ vectors).

4 EXPERIMENTS

We empirically evaluate our algorithm on two different environments. The first one, *Deep Sea Treasure* [15], is a well-known MORL benchmark that serves as a proof-of-concept for our method, as no deep-RL approach is needed. In the second environment, we create a traffic-intersection simulator, using the Simulation of Urban MObility (SUMO) simulator [4]. This environment has a high-dimensional state-space, and cannot be solved using the original PQL algorithm.

Algorithm 1: PDQN

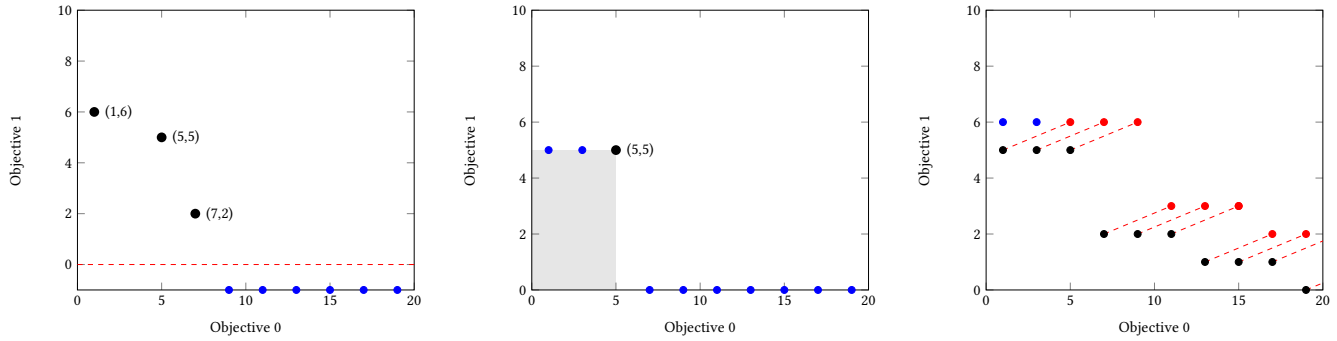
initialize replay-memory D
initialize reward estimator $\overline{R}$
initialize non-dominated estimator ND_t
initialize target non-dominated estimator $\hat{ND}_t = ND_t$
for *episode* = 1 **to** M **do**
  while *not terminal* **do**
    sample points p from $\mathbb{R}^{d-1}$
    $Q_{set}(s, ., p) \leftarrow \overline{R}(s, .) \oplus \gamma ND_t(s, ., p)$
    $hv \leftarrow \text{hypervolume}(Q_{set}(s, ., p))$
    $a \leftarrow \epsilon - \text{greedy}(hv)$
    execute a in environment, observe state s', reward r, terminal t
    add transition (s, a, r, s', t) to D
    sample minibatch $(s_i, a_i, r_i, s'_i, t_i)$ from D
    sample points p_i from $\mathbb{R}^{d-1}$
    $y_i = \begin{cases} ND(\cup_{a' \in A} \hat{Q}_{set}(s'_i, a', p_i)) & \text{if not } t_i \\ r_i & \text{otherwise} \end{cases}$
    update ND_t by performing gradient descent step on $(y_i - Q_{set}(s_i, a_i, p_i))^2$
    update $\overline{R}$ by performing gradient descent step on $(r_i - \overline{R}(s_i, a_i))^2$
    every C steps copy ND_t to $\hat{ND}_t$
  end
end

4.1 Deep Sea Treasure

In *Deep Sea Treasure*, the agent is a submarine looking for hidden treasures at the bottom of the ocean (see figure 2). On one hand, the agent seeks the highest possible treasure (objective 0). On the other hand, its fuel consumption is a concern: minimizing it is the second objective. The agent can perform one of four actions at each step (going up, down, left or right), each action giving -1 fuel. The episode stops when a treasure is reached. The optimal Pareto front is a known, concave function. The concave property means that some points will not be reachable using any linear scalarization of the reward function.

The state-space is of size 110, and is represented as a one-hot encoding of the agent's coordinates. Similarly, the performed action is represented as a one-hot vector of size 4. Both the non-dominated and the reward estimators have the same network architecture:

Where FC_s is a fully connected layer that takes the state concatenated with a value for objective 0 as input, and FC_a a fully connected layer with as input an action. The output o is of size 2 for the reward estimator, and 1 for the non-dominated point estimator. Both are optimized using Adam [3], with a learning rate of 10^{-3} and 10^{-4} respectively. We keep $\gamma = 1$ (no discount) and a batch size



(a) A Pareto front (in black) that only ranges over a subspace of the objective 0 domain. By incorporating domain knowledge about the objective-space (e.g., only positive rewards for objective 1, in red), we can train the estimator to output values outside this space (in blue), and discard them during evaluation.

(b) Sampling strategy when the Pareto front corresponds to the final terminal reward (in black). The newly added points (in blue) dominated by $(5, 5)$ in objective 0 receive a value of 5 for objective 1, while the ones dominating it receive a value of 0 (the reference point). The hypervolume (in gray) remains unchanged.

(c) Sampling strategy of the Pareto front (in black) after shifting due to reward $(4, 1)$ (in red). This causes parts of the objective 0 domain to be uncovered. Samples are added (in blue) in a similar fashion as figure 1b.

Figure 1: The Sampling strategies used to train the non-dominated set estimator.

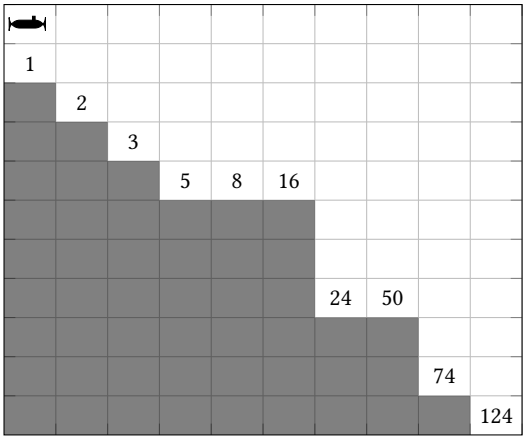


Figure 2: The *Deep Sea Treasure* environment. The agent starts on the top-left corner and tries to reach any of the treasures. Farther treasures are worth more.

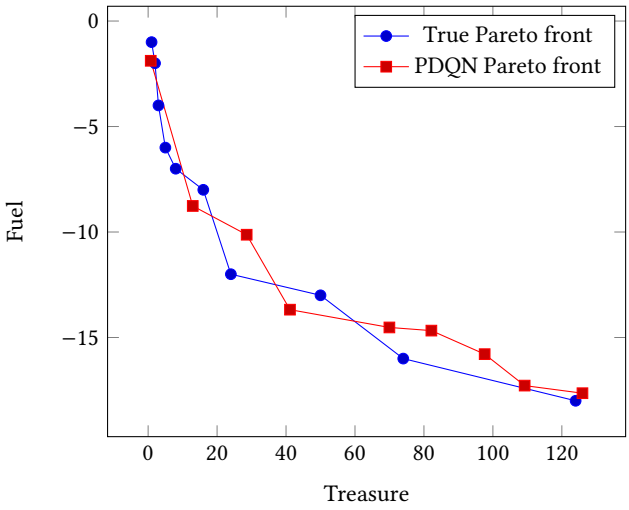


Figure 3: The true Pareto front (in blue) in the start-state compared with the estimated one (in red). The estimate is computed as the non-dominated set of $Q_{set}(s_0, .)$.

of 32. The output of ND_t is normalized by dividing $\overline{R}$ with $(124, 19)$ in order to improve stability. The reference point used to compute the hypervolume was chosen as $(-1, -2)$, to ensure that it will be below any normalized output of ND_t. Finally, we train the agent for 5×10^4 episodes.

4.1.1 Results. Figure 3 shows the estimated Pareto front (red) compared to the true one (blue) in the start-state. It was computed by keeping the non-dominated point of $Q_{set}(s_0, ., p)$, with p sampled using the same strategy as during training. First of all, we can observe that the network was able to approximate the general trend of the compromises imposed by the environment. Moreover, only 5000 episodes are needed to reach an estimated hypervolume of 3.11 (not shown), at which point the it oscillates around this value (the true hypervolume being 3.22).

While PQL is able to find the optimal Pareto front, it does so by keeping track of the future returns, for every possible state-action pair. In contrast, PDQN provides an informative approximation, even though most of the points used for training where never actually reached, since they were sampled over the whole treasure-objective range $[0, 124]$.

4.2 Traffic environment

In order to evaluate PDQN in a more complex state-space, we created a traffic intersection using the SUMO library (see figure 4).

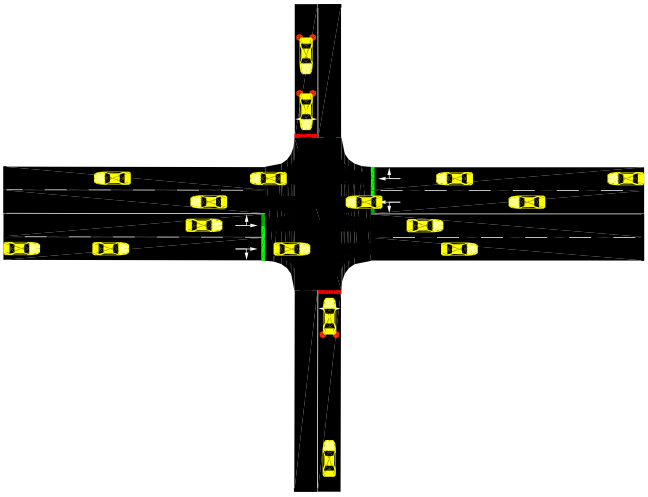


Figure 4: A visualization of the Traffic environment.

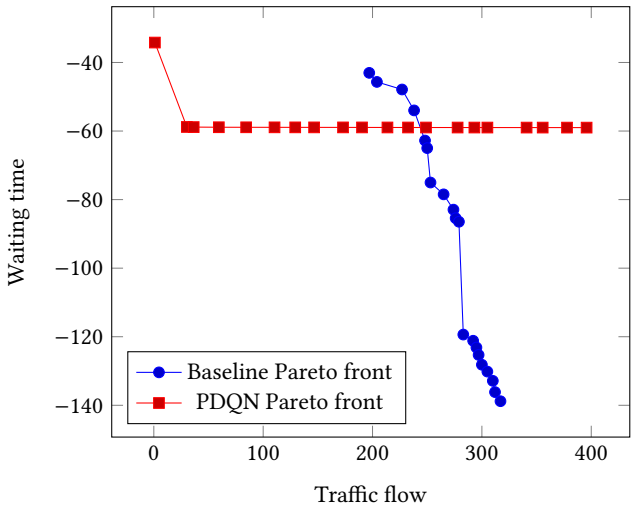
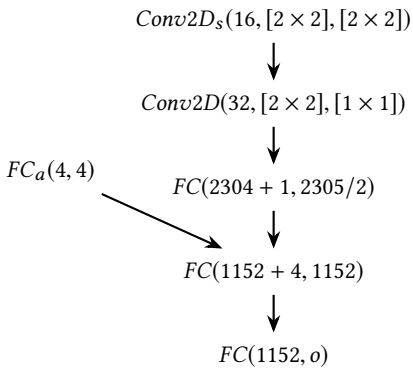


Figure 5: The Pareto front found by PDQN (in red), compared to baseline static policies, that switch the light at fixed intervals (in blue).

The car flow at the intersection is controlled by traffic lights, which will be our agent. We devise 2 objectives. First, one should maximize the traffic flow. Every car that leaves the intersection provides a positive reward of 1. Secondly, we want to minimize the car's waiting time. At every time step, every car that has yet to pass the intersection increments its waiting time. We provide the longest waiting time as a negative reward to the agent. These objectives might seem aligned but, as one road has two lanes while the other only possesses one, the optimal policy to maximize traffic flow is to simply always keep the lights green for the larger road. However, if one focuses on the waiting time, one should regularly switch the lights. The episode length is fixed at 200 steps.

The agent can execute two actions: turn the lights green (switching the other to red) for either of the two roads. When a light switches, there is a delay of 3 time-steps during which the light turns yellow, allowing the cars to safely brake. Each car has a position (x, y), a driving speed and a current waiting time. We simplify the state by tiling the 2-dimensional space and putting the waiting time of each car in the appropriate slot, depending on its position. The tiling is sufficiently fine-grained as to guarantee that no two cars can fill the same slot. We use the four corner-slots to indicate the appropriate traffic-light's color (0, 1, 2 for red, yellow, green respectively). The image is partitioned in 20×18 tiles and, to alleviate the missing speed information, a history of 4 frames is kept.

Similarly as with the the previous experiments, the reward and non-dominated estimators share the same architecture:



Where $Conv2D_s$ is a convolutional layers with a concatenation of the state and objective 0 as input, 16 filters of size 2×2 and stride 2×2. Again, we optimize the networks with Adam, using learning rates 10^{-4}, 10^{-5} for respectively the reward, non-dominated estimator. The output of ND_t is normalized by dividing $\overline{R}$ with $(400, 120)$, and the reference point used is $(-2, -2)$.

4.2.1 Results and Future work. Due to the complexity of the environment, we were unable to calculate the optimal Pareto front. A baseline was created by executing policies that switch the lights at fixed, regular intervals. The length of these intervals ranges from 3 to 150 steps, with an increase of 5 steps per interval. Only the policies producing non-dominated returns are kept. Figure 5 depicts these baseline policies (in blue) compared with the estimated Pareto front produced by PDQN (in red).

First of all, we observe that, regardless of the chosen policy, there will always be a minimum amount of cars passing through the lights. The baseline performs better than expected in that regard, as we observed episodes with a traffic flow inferior to 100 during

training. This occurs when The light is red most of the time for the horizontal, larger road. The opposite also applies: the flow and waiting time can be greatly improved by keeping the light green most of the time for the horizontal road, and periodically switch them to allow for the waiting cars on the vertical road to leave.

Although both these factors would account for more flat Pareto front than the actual baseline, the estimate produced by PDQN outputs the same waiting-time value, regardless of the chosen traffic flow. The only exception is the leftmost sample (no traffic flow), which we devise is due to the lack of any observed return in that region, even using the worst possible policy.

Still, the network is able to converge towards a plausible waiting time, even though it is the same regardless of the chosen traffic flow. As such, we do believe that those preliminary results can be improved upon, and that PDQN can be used in complex environments. First of all, we will investigate the effect of different sampling strategies: densely sampling the $\mathbb{R}^{d-1}$ space might lead to a more fine-grained solution. Secondly, as for this environment, the total return is composed of the sum of regular, small rewards (as opposed to Deep-Sea-Treasure, that provides a one-time treasure reward), we will incorporate more domain knowledge of the reward function in our approximators. Finally, we will investigate if some of the many improvements made on DQN, such as Double-DQN [17], can be applied in our setting.

5 CONCLUSION

We extended DQN to cope with multi-objective sequential decision problems, and were able to successfully apply it on a proof-of-concept benchmark. Our algorithm, PDQN, was able to approximate the Pareto front, using a novel network architecture and sampling strategy. This is also, to the best of our knowledge, the first time that an inner-loop method was devised for a Deep Reinforcement Learning setting. However, when applied on a high-dimensional traffic environment, PDQN fails to provide a believable estimate. Still, due to the convergence of the network, and the plausible waiting time, we do believe that the results can be further improved, and that PDQN can potentially be applied on high-dimensional state-space problems.

REFERENCES

[1] Andrea Castelletti, Giorgio Corani, A Rizzolli, R Soncinie-Sessa, and Enrico Weber. 2002. Reinforcement learning in the operational management of a water system. In *IFAC workshop on modeling and control in environmental issues*. 325–330.

[2] Indraneel Das and John E Dennis. 1997. A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems. *Structural optimization* 14, 1 (1997), 63–69.

[3] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[4] Daniel Krajzewicz, Georg Hertkorn, Christian Rössel, and Peter Wagner. 2002. SUMO (Simulation of Urban MObility)-an open-source traffic simulation. In *Proceedings of the 4th middle East Symposium on Simulation and Modelling (MESM20002)*. 183–187.

[5] Daniel J Lizotte, Michael H Bowling, and Susan A Murphy. 2010. Efficient reinforcement learning with multiple reward functions for randomized controlled trial analysis. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. Citeseer, 695–702.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[7] Sriraam Natarajan and Prasad Tadepalli. 2005. Dynamic preferences in multi-criteria reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 601–608.

[8] Daniel Neil, Marwin Segler, Laura Guasch, Mohamed Ahmed, Dean Plumbley, Matthew Sellwood, and Nathan Brown. 2018. Exploring deep recurrent models with reinforcement learning for molecule design. In *6th International Conference on Learning Representations (ICLR), Workshop Track*.

[9] OpenAI. 2018. OpenAI Five. https://blog.openai.com/openai-five/. (2018).

[10] Simone Parisi, Matteo Pirotta, and Marcello Restelli. 2016. Multi-objective reinforcement learning through continuous pareto manifold approximation. *Journal of Artificial Intelligence Research* 57 (2016), 187–227.

[11] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc.

[12] Diederik M Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. 2013. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research* 48 (2013), 67–113.

[13] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

[14] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press Cambridge.

[15] Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. 2011. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning* 84, 1-2 (2011), 51–80.

[16] Peter Vamplew, Rustam Issabekov, Richard Dazeley, Cameron Foale, Adam Berry, Tim Moore, and Douglas Creighton. 2017. Steering approaches to Pareto-optimal multiobjective reinforcement learning. *Neurocomputing* 263 (2017), 26–38.

[17] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[18] Kristof Van Moffaert, Madalina M Drugan, and Ann Nowé. 2013. Scalarized multi-objective reinforcement learning: Novel design techniques. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 191–199.

[19] Kristof Van Moffaert and Ann Nowé. 2014. Multi-objective reinforcement learning using sets of pareto dominating policies. *The Journal of Machine Learning Research* 15, 1 (2014), 3483–3512.

[20] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. 2019. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. (2019).

[21] Weijia Wang and Michèle Sebag. 2013. Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine learning* 92, 2-3 (2013), 403–429.

[22] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.