Maximizing the Revenue of a Hydroelectric Dam with Reinforcement Learning
Group 3

Paola Feil[2732911] and Matilda Knierim[2700374]

Vrije Universiteit Amsterdam

1 Introduction

Hydroelectric energy is a valuable energy source and supplies more than 10% of the world's electricity. However, the operation of the storage levels and connected trade of the energy is yet a key challenge. In the project, we propose a solution for intelligently trading and storing water in a dam. For this we consider a dam that has a $30m$ difference from the upstream to downstream side. Moreover, the maximum storage capacity of the dam is 100 000 m^3. However, the hydro turbines can only transition $5m^3/s$. The main operations with the dam are to buy or sell energy to the current market price. To quantify the efficiency of the operation, the electricity market prices in Euros/MWh were used to find strategies to successfully maximise the revenue of the dam. These were given per hour over a course of three years (2007-2010).

The strategies proposed by stock trading literature range from relatively simple approaches based on market price ranges [1], to more sophisticated algorithms, such as deep reinforcement learning algorithms [5]. One algorithm that achieved good results in several stock trading experiments was Double Deep Q-Learning (DDQ) [5][8][3]. Thus, in our project we implemented three algorithms. We start with a simple "Buy low, sell high" baseline, then implement a Tabular Q-Learning (TQ) algorithm and finally train a DDQ agent. The aim is that the agent learns when to sell, buy, or hold the energy.

We assume that the baseline sets a stable ground for a good performance, which could bring about the same revenue as the simple TQ algorithm. Furthermore, it is proposed that the DDQ will outperform the baseline as well as the TQ.

Also, one problem the agent(s) can encounter is the positive reward that is returned when the agent sells energy in comparison when it buys water. Even though the agent receives value for buying the water, this is not reflected in monetary turns. We suggest that this unbalance in monetary rewards in comparison to the decreasing/increasing water value might impact the agents' learning process. Hence, two methods of reward shaping were introduced. We hypothesise that the performance of the algorithms increases with the implementation of reward shaping.

2 Data Exploration

2.1 Training Data Description

The training set consists of time series data providing the market's energy price (in $Euros/MWh$) at each hour in the date time range of 2010-01-01 00:00:00 to 2011-12-31 00:00:00, thus covering three years. Since no missing values nor missing hours in this date time range could be detected, a total of 26,304 prices are included in the training data. The maximum price included is $2500 Euros/MWh$ and the minimum is $0.01 Euros/MWh$. Their distribution is shown in Figure 1. As the histrogram shows, there is a strong positive skew in the prices, indicating unusually high energy prices.

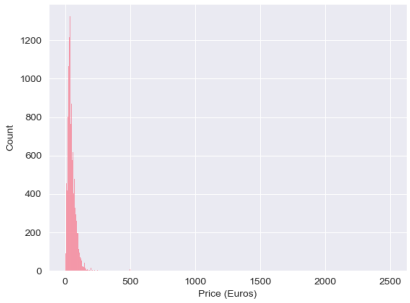

Fig. 1. Training Price Distribution Before Outlier Imputation

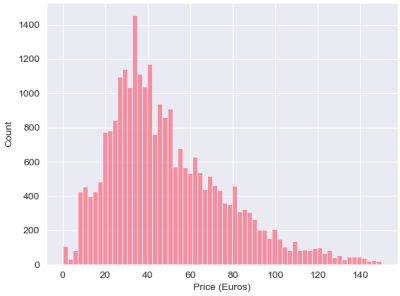

Fig. 2. Price Distribution After Outlier Imputation

2.2 Outlier Detection & Imputation

Since there is notably little data to train on, some of the outliers belonging to the positive skew were removed (i.e. the highest 2% of prices), and then imputed via interpolation between the previous and next measurement. The new data distribution can be seen in Figure 2. As the Figure shows, the data is not perfectly normal but the positive skew is reduced.

2.3 Validation Data Description

Similar to the training data, the validation set provides prices (in $Euros/MWh$) across two years, covering the full time range between 2010-01-01 00:00:00 and 2011-12-31 00:00:00. As no data is missing, this amounts to 17,519 prices.

3 Environment

The environment to operate the hydroelectric dam with an RL agent was implemented using the *Gym* library [2]. Its main components comprise the state space and action space, and the functions regulating the deterministic dynamics, that is the `step`, `reward` and `reset` function.

3.1 State & Action Space

The state space of the environment was implemented with dimensions, that is *water level* and *price*, which represent the fill level of the water reservoir (in m^3) and the current energy price (in $Euros/MWh$) each hour, respectively. Their value ranges are 0 to 100,000 m^3 for the *water level* and 0 to infinity for the *price* since no upper bound is given by the market. While DDQN and the baselines can handle such a continuous state space, Tabular Q-Learning cannot and therefore the states were discretized in the agent (see Tabular Q-Learning). Regarding the action space, it may be sensible to work with a continuous action space to not always sell or buy the maximum amount of energy possible, however, neither Tabular Q-Learning nor DDQN are capable to work with continuous actions. Hence, the action space was discretized into three actions, namely *sell* (0), *hold* (1), and *buy* (2).

3.2 Reset & Step Function

The `reset` function places the agent in the starting state, that is a half full water reservoir (i.e. $waterlevel = 50,000m^3$) and the *price* of the first time point of the passed time series data. Once an action has been selected, the `step` function moves the agent to the next state, which is one hour further in time. The new state is then determined by retrieving the current *price* and calculating the new *water level* based on the chosen action. If the agent chose to hold, the *water level* remains unchanged. Regarding the other actions, the agent always buys or sells the maximum amount possible to be dealt in one hour given its *water level* and the maximum hourly dam processing capacity of $18,000m^3$. Hence, the agent sells 0 to 18,000m^3, or buys 0 to 18,000$m^3 \cdot 1.25$ of which only 80% reach the water reservoir due to the upstream efficiency, corresponding to maximal $18,000m^3$.

3.3 Reward Function & Reward Shaping

The `reward` function is then called in the `step` function to calculate the reward of the deterministic transition. The reward is based on the cost or revenue generated by the action. Therefore, if the agent held, tried to sell while the reservoir was empty or attempted to buy with a full tank, a reward of zero is returned. If the agent manages to actually buy an amount of energy x_t, the reward corresponds to $x_t \cdot (-price_t)$. However, as downstream efficiency is only 90%, the reward of selling an amount of energy x_t corresponds to $x_t \cdot price_t \cdot 90\%$.

Moreover, reward shaping was implemented to guide the agent to better generalization by providing it with more meaningful feedback and thus incentivizing it to explore better policies. For the problem at hand, the reward as described above may lead the agent to never buy as to avoid negative rewards. Two different reward shaping strategies were implemented to counteract this issue: The first reward shaping, referred to as Type 1 in this report, considers the water level in addition to the price. The second reward shaping, referred to as Type 2 in this report, only considers the price. For the former strategy, Type 1, the median price of the training data is taken and multiplied with the change in potential energy. This term is then added to or subtracted from the regular reward when the agent buys or sells, respectively, in order to quantify the value gained or lost in potential energy. The latter strategy, Type 2, is based on the quantiles of the prices contained in the training data. If the agent buys or sells when the price belongs to the highest quantile, its reward is multiplied by the factor 2. If the agent sells or buys in the lowest quantile, its reward is divided by the factor 2. This should motivate the agent to rather buy when the price is low and sell when the price is high.

3.4 Episode Definition

One episode in the environment is defined as a whole walk-through the passed time series. Yet, to speed up training for the DDQN network, warm starts were implemented as well. That is, the agent is being put at a random point in time of the series that still allows him to take 400 transitions and begins with a randomly selected *water level*.

4 Baselines

Two baseline algorithms were set-up for comparing the performance of the more complex algorithms. The first baseline that was implemented was the random baseline. Thus, the agent moves one time through the environment by randomly picking one of the actions sell, hold and buy. Based on this action the agent collects the reward and moves to the next state until the end of the episode is reached, in other words when the end of the validation data set is reached. Unsurprisingly, the random baseline had a low performance with an end return of -32973. Even though these results were not surprising, it was an interesting insight to obtain a threshold for completely random behavior for this problem statement.

The second baseline algorithm was based on the "sell high, buy low" strategy [1]. It suggests to analyse previous market prices to gain an understanding about the price distribution and developments. Then, an upper and a lower price thresholds should be set, one for when to buy and one for when to sell. This strategy enables the agent to directly interpret the market prices it encounters. On the other hand, it is vulnerable to biases in the training data. We defined our thresholds based on the training data set. The quantiles of the prices for each

year in the training dataset was calculated. Then, one final quantile categorisation was determined by taking the median over each quantile threshold over the three years. We aimed to reduce the price fluctuation between the years as much as possible with this approach and have generalisable thresholds. Based on these quantiles, the agent always buys if the price is in the lowest quantile and sells if the price is in the highest quantile. Otherwise the agent holds. With this strategy, the agent receives a return of 53108.

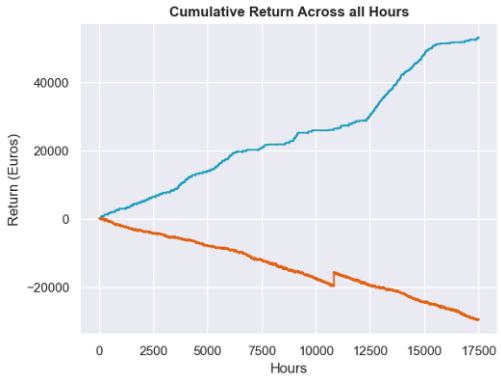

Fig. 3. Comparison between the random (orange) and quantiles (blue) baseline

5 Tabular Q-Learning

5.1 Algorithm Description

TQ was the first reinforcement learning algorithm that was implemented to tackle the problem statement. The basic idea of TQ is to gain experience about an environment in a "learning-by-doing" approach. If the agent has encountered a positive experience it is more likely to do it again, whereas negative consequences will be encountered less likely in the future. By taking steps in the environment, the agent is constantly updating its Q-function. The Q-function assigns a value to each state-action pair, in other words taking action a in state s, based on either positive or negative experiences that were made [7].

In TQ, the Q-function is represented by a look-up Q-table, where the Q-value for each possible state and action pair is stored. The Q-table gets updated in each step that the agent is doing with the formula $Q(s,a) = Q(s,a) + \alpha\,[reward + \gamma max_a Q(s_{t+1},a) - Q(s,a)]$, with s as the current state, a as the current action and s' as the next state. Then, the agent moves through the environment and gains experience until TQ converges and the Q-table represents are learned policy. The learned policy reflects which action the agent should take in which state to maximise the return.

TQ has several advantages and disadvantages. First, an advantage is that it is a model free algorithm and thus does not need to know the transition probabilities of an environment. This is especially useful if the environment is unknown, as it is in our case. Secondly, Q-Learning does not need to wait until the end of an episode but can update the Q-function "on the go". Last but not least, TQ is proven to converge and thus finds a policy. On the other hand, TQ needs to have a discretized state and action space and suffers from overestimation bias due to its update $max_a Q(s_{t+1}, a)$.

5.2 Algorithm Implementation

The implementation of our TQ agent can be found in the TabQAgent.py file. The agent uses the aforementioned environment. Since discretized states are needed for TQ, we decided categorize the price again into quantile bins, ranging from 0 as "very low price" to 3 as "very high price". The water level was discretized into 10 equally sized bins. Moreover, to update and evaluate the policy, a Q-table with the shape water level dimensions x price dimensions x actions was initialised. The state discretization was done by the agent after every step, to enable it to update the Q-table.

Moreover, the epsilon greedy policy was chosen to sample the actions. Epsilon greedy takes action using the greedy policy with a probability of 1-ϵ and a random action with a probability of ϵ. Thus, the policy keeps a certain balance between exploration and exploitation. We implemented an adaptive epsilon, which started at 1 (full exploration) and decreased with each agent step until it reached an end epsilon of 0.05. The learning rate was implemented in an adaptive manner as well. It started at 1 and was decreased after every episode with the formula $lr = lr/\sqrt{episode + 1}$, where lr represents the learning rate and episode the current iteration of the episodes. The discount factor was set to 0.99 and the number of episodes to 1000.

6 Double Deep Q-Learning

6.1 Algorithm Description

DDQN is a semi-gradient RL function approximator that combines the concepts of Deep Q-Networks (DQN) and Q-Learning. In DDQN, a neural network is used to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is predicted as the output. DDQN separates the target and training Q-networks, which helps to mitigate the instability caused by the correlation between the target and estimated Q-values, thus leading to more accurate and stable training. The target network has the same architecture as the online network but with frozen parameters. It gets updated in a set frequency by setting its parameters equal to the ones from the online network.

The online network is trained with the help of a replay memory, where past experiences of the agent are stored. A batch of past experiences are sampled

from the replay memory and the Q-values for each experience are predicted. To compute the loss, the target network predicts the maximum Q-value of each next state and the Smooth L1 Loss is applied. Thus, the algorithm alternates between sampling transitions from memory, computing Q-values using the online network, and updating the target network with the Q-values.

DDQN yields many advantages. First, using two neural networks addresses the problem of the overestimation of action values by decoupling the selection and evaluation of actions, leading to improved stability and performance. Secondly,DQN has the possibility to use continuous and highly dimensional state spaces, since it uses function approximation. Given that our problem statement has continuous variables, e.g. water level and price, it is likely that DDQN can learn a better policy than TQ. On the other hand, DDQN is not proven to converge since it only approximates the Q-function. It is also quite sensitive to the parameter settings in comparison to the traditional TQ and it takes more computational resources to train.

Q-Network Architecture & Gradient Descent The implementation of the Q-network can be found in the file DeepQAgent.py. The implemented neural net takes as input the two-dimensional state, and subsequently predicts the corresponding Q-values. This is done by passing the input through three linear layers. The first layer outputs 64 values which are activated using ReLU. This activation function was selected to avoid vanishing or exploding gradients. The next layer outputs 32 values which are again activated with ReLU before being passed to the last layer which outputs three values, that is the predicted Q-values for each state-action pair. Experimenting with more complex architectures did not improve the performance as the network seemed to overfit more strongly.

Considering the learning step, batch gradient descent was implemented with a batch size of 32. Furthermore, *PyTorch's* implementation of the Adam optimizer was used using its default parameters and and a learning rate of 0.00005. Adam is computationally efficient and suitable for non-stationary objectives as caused by the target network [4].

Double Q-Agent The implementation of the DDQN-Agent can be found in the file DeepQAgent.py. The library PyTorch [6] was used. The agent combines the aforementioned elements Dam environment, the Q-network, replay memory and epsilon greedy policy to approximate the Q-value function. The replay memory is implemented with a capacity of 104000 and filled with random transitions from the environment. Then, the online and target network are initialised with the same parameters.

The agent starts in the initial state of the environment and samples the action based on the epsilon greedy policy. An adaptive epsilon is used that starts with an epsilon of 1 and gradually decreases until it reaches 0.05. After each step of the agent, the current transition is added to the replay memory and the online network is trained on a batch of replay experiences. The states, which are fed into the network, as well as the rewards, which are used for the target update,

are normalised by subtracting the mean and dividing by the standard deviation for a stable training. After the loss is computed, a backwards propagation step is conducted using the Adam optimizer. After each 5000 training steps, the target network is updated by setting its parameters equal to the ones from the online network. One episode is again terminated after the agent went through all market prices. Furthermore, the current policy of the agent is evaluated in a greedy manner after every episode.

The agent is trained on 10 episodes (without the warm start implementation) with a learning rate of 0.00005, a discount factor of 0.98. These parameters as well as the adaptive epsilon and the update frequency of the target network were tuned by using a grid search.

7 Experiments & Results

7.1 Experiment Setup

TQ, and DDQN were trained and validated over five runs due to the random elements in these algorithms. For TQ and DDQN, these five runs were conducted three times: once without reward shaping and one time with each reward shaping strategy. While TQ was run for 1000 episodes, DDQN was run for 10 episodes, as initials trials showed that these numbers were sufficient to reach convergence.

7.2 Tabular Q-Learning Results

The results for the TQ without reward shaping, reward shaping Type 1 and reward shaping Type 2 can be found in Figure 4, 5, and 6 in the Appendix. The shape of the validation and training curves look very similar in all three plots. The training curves show a gradual upward, almost linear learning trend, which converges abruptly after 1000 episodes. The validation curve shows an unusual trend which is very steep for the first 10 episodes and then stagnates. This trend is further discussed in the Discussion section. The significant difference comes down to the return amount. Both reward shaping method seemed to increase the return for the training and validation. While the return without reward shaping converges to approximately -3k and 1k for training and validation, respectively, reward shaping Type 1 increases these returns to around 68k and 31k, and reward shaping Type 2 converges to returns of around 60k and 37k for training and validation. Hence, the model with Type 2 reward shaping which does not take the water level into account seems to generally perform the best amongst these three models.

Next, the maximum Q-Values of the converged Q-tables were investigated for all three variations of TQ. These were retrieved from the each best run. 3D plots of these values across the discretized state space can be found in the Appendix in Figures 10, 11, and 6.

Regarding TQ with no reward shaping (see Figure 10), one can see that the maximum Q-values increase strongly with rising water levels, and also slightly

with rising prices. This is sensible as without reward shaping, selling is by far the most profitable action and can generate a lot of reward in when the water level and price are high. Moreover, the Q-values seem to jump up and down between the classes of the water level. This may be explained by the mismatch between the bins of the discretized water level and the rigid amounts of water influx and outflow from the dam which are mostly 18,000 m^3.

Considering the TQ model with reward shaping Type 1 (see Figure 11), one can observe that the lowest Q-values are located in the state space where the price is low and the water level is high. This may be explained by reward shaping, as when the reservoir is full and energy is cheap, the agent cannot buy and thus receives only zero rewards for buying or holding, or negative rewards for selling. Further, the highest Q-values are achieved when the the water tank is rather empty and the price is low. In this case, the implemented reward shaping may indeed lead to a positive reward for buying more energy. Moreover, the same issue regarding the binning of the water level can be observed in this plot, thus supporting the provided reasoning that this is due to the discretization of the state space rather than to the specific implementation of the rewards. Finally, 6, which corresponds to the TQ model with reward shaping of Type 2, shows that the highest Q-values are located in the state space where the water level as well as the price are high. This may be due to the circumstance that despite the fact that selling is punished it still always leads to a positive reward, and thus is the most chosen action in this situation as buying would return a negative reward or zero reward, and holding would also not increase the return. In general, a diagonal upward trend through the space can be observed so that the lowest values are located where the price is low and the water level low. This is sensible, as a low price and low water level would likely either yield zero rewards by holding or negative rewards by buying as selling is likely not an option in this situation. Finally, the same issue regarding the binning of water level is present in this plot.

7.3 DDQN Results

The results of the DDQN can be found in the Appendix in section 9.2. The training and validation curves all follow the same trend. The training curve gradually learns until the 4th episode and then stagnates. The validation curve, on the other hand, starts at a much higher return that the training and stagnates reasonably quickly, at the second episode. However, the amount of the obtained return are different for the three DDQN versions. The lowest validation return has the DDQN without reward shaping with an end return of around 25000. The training return for this the DDQN with no reward shaping is at around 45000. The validation return of the DDQN with a Type 1 reward shaping is around 35000, and thus slightly higher than in the version without reward shaping. The training return is located similarly as the validation return around 35000. The highest return of the DDQN algorithm was obtained with the Type 2 reward shaping. The agent received a return of around 40000 on the validation data set and 60000 on the training data stet.

8 Discussion

After looking at the results it can be said that reward shaping had a positive effect on the learning performance of the agent. The return was significantly higher for both Type 1 and Type 2 reward shaping in comparison with the baseline, for TQ as well as DDQN. Yet, reward shaping Type 2 yieled the best performance. This shaping strategy only took the current price into account and thus pushes the agent to behave a little more like the quantiles baseline. This underlines our hypothesis that the agent benefits from counteracting the initial reward function which only rewards selling energy with a positive reward, thus rarely buying. Moreover, the validation return of the DDQN was as expected to be higher than the validation return of TQ. Especially, DDQN with Type 2 reward shaping outperforms the others significantly. This could be traced back on the ability of DDQN to take continuous states as input, while TQ loses information about the environment by discretizing the states. Especially other implemented water binning seemed to decrease the performance of TQ (as stated in the results), which gives DDQN an extra performance advantage over the tabular version. Interestingly, the best performing algorithm was the quantiles baseline. We suggest that a simple algorithm worked better than DDQN due to the limited amount of available training data. Last but not least, the validation curve in all plotted results follow an unusual trend. The validation curve starts quite high in the first episodes and stagnates rather quickly. We have two possible explanations for this behavior. First, the validation set might have higher market prices in general, which gives also an almost untrained agent a head start. Secondly, wrongly tuned hyper-parameters could be responsible for the impacted learning and surprising behavior of the validation curves. For future research, we propose to discretize the state space more meaningfully to boost TQ results.

References

1. Buy Low, Sell High Strategy: An Investor's Guide, https://www.sofi.com/learn/content/buy-low-sell-high-investment-strategy/
2. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
3. Dang, Q.V.: Reinforcement learning in stock trading. In: Le Thi, H.A., Le, H.M., Pham Dinh, T., Nguyen, N.T. (eds.) Advanced Computational Methods for Knowledge Engineering. pp. 311–322. Springer International Publishing, Cham (2020)
4. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. International Conference on Learning Representations (12 2014)
5. Li, Y., Ni, P., Chang, V.: Application of deep reinforcement learning in stock trading strategies and stock forecasting. Computing **102**(6), 1305–1322 (12 2019). https://doi.org/10.1007/s00607-019-00773-w, http://dx.doi.org/10.1007/s00607-019-00773-w
6. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
7. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA (2018)
8. Wu, X., Chen, H., Wang, J., Troiano, L., Loia, V., Fujita, H.: Adaptive stock trading strategies with deep reinforcement learning methods. Information Sciences **538**, 142–158 (2020). https://doi.org/https://doi.org/10.1016/j.ins.2020.05.066, https://www.sciencedirect.com/science/article/pii/S0020025520304692

9 Appendix

9.1 TQ & DDQN Average Return During Training and Validation

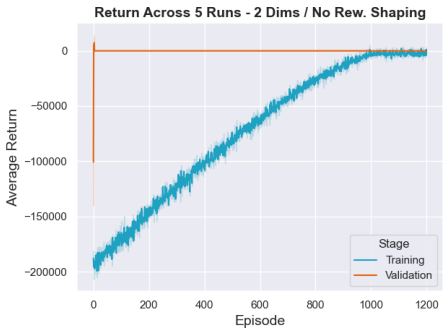

Fig. 4. TQ: Return; no R. Shaping

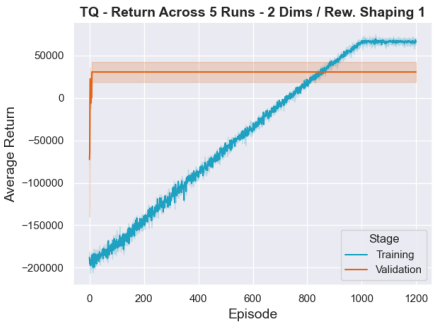

Fig. 5. TQ: Return w/ R. Shaping 1

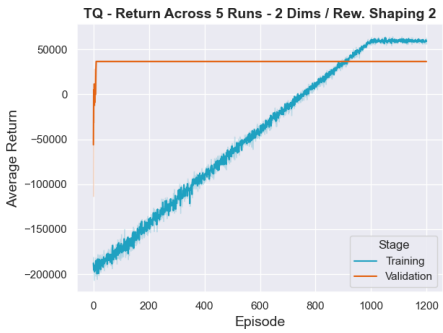

Fig. 6. TQ: Return w/ R. Shaping 2

Fig. 7. DDQN: Return; no R. Shaping

9.2 TQ - 3D Plots of the Maximum Q-Values

Fig. 8. DDQN: Return w/ R. Shaping 1

Fig. 9. DDQN: Return w/ R. Shaping 2

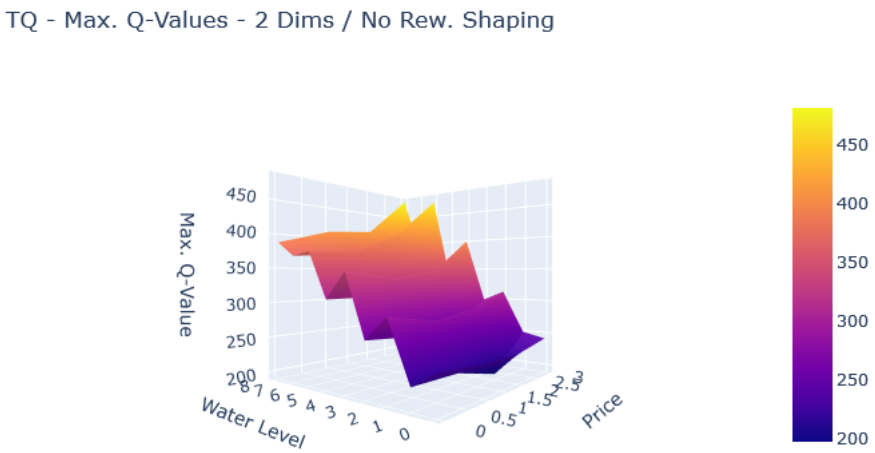

Fig. 10. TQ: Max. Q-Values without Reward Shaping

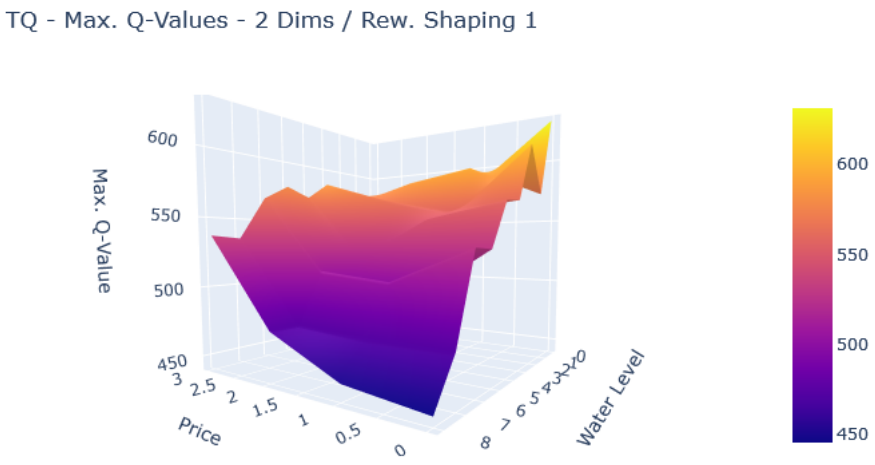

Fig. 11. TQ: Max. Q-Values with Reward Shaping 1

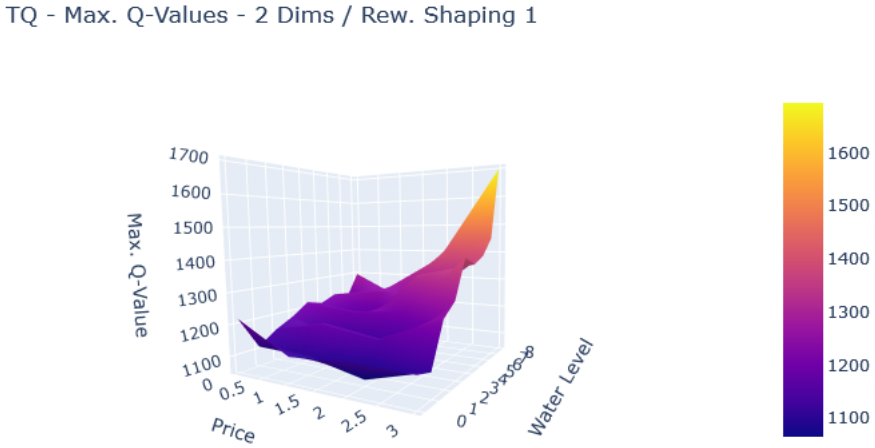

Fig. 12. TQ: Max. Q-Values with Reward Shaping 2