"Y'all got any more of those Rewards?" Investigating Transfer Learning in Modern Deep Reinforcement Learning Architectures

Gerrit Bartels
School of Human Sciences
Osnabrück University
Osnabrück, Germany
gbartels@uni-osnabrueck.de

Thorsten Krause
School of Business Administration and Economics
Osnabrück University
Osnabrück, Germany
thkrause@uni-osnabrueck.de

Jacob Dudek
School of Human Sciences
Osnabrück University
Osnabrück, Germany
jadudek@uni-osnabrueck.de

***Abstract*—Maximum-entropy based policies consider multiple paths to success contrary to traditional approaches. Our project explored the benefits of transfer learning regarding Soft Q-Networks and Double Deep Q-Networks, and an alleged relationship between overfitting and negative transfer. The popular NES game Super Mario Bros. served as a test bed for our research. We proposed and executed an experimental setup, provide a ready-to-use implementation and identified and put forth major challenges that future research can build upon. To the best of our knowledge, we are the first to directly investigate the transfer learning capabilities of Soft Q-Learning. With our contribution, we lay the ground work to answer fundamental questions in transfer learning that could potentially safe expensive resources.**

***Index Terms*—reinforcement learning, double deep Q-networks, soft Q-learning, prioritized replay buffer, transfer learning, #publishmoreunsuccessfulresearch, #wehadwaytoolittle-timeforthis**

I. Introduction

Through reinforcement learning, machine-based agents autonomously learn to perform one or multiple tasks at hand. However, traditional reinforcement learning approaches only consider a single deterministic optimal path to success. Concerned about resulting noise and imperfect behavior, Ziebart et al. [1] propose a maximum entropy-based, stochastic alternative. Haarnoja et al. [2] point out additional benefits of maximum entropy-based policies as robustness or exploration for multi-modal objectives. They explain

> *"Instead of learning the best way to perform the task, [maximum entropy-based] policies try to learn all of the ways of performing the task."* [2]

As existing solutions could only represent a limited range of distributions, Haarnoja et al. [2] further propose *Soft Q-Learning* as a more general approach capable of asymptotically approximating arbitrary distributions. The authors explicitly developed Soft Q-Learning for pre-training agents on general tasks and fine-tuning them for more specific tasks afterwards, i.e., for a transfer learning scenario. In reinforcement learning, transfer learning means training an agent on source tasks and leveraging the generated knowledge to perform or learn a target task [3]. For example, Haarnoja et al. [4] apply Soft Q-Learning to combine policies trained on different stacking tasks for a robotic agent. Nemcek & Parr [5] investigate performance bounds of Soft Q-Learning agents in transfer learning scenarios.

However, to the best of our knowledge, researchers have not yet explicitly investigated the general potential advantage of Soft Q-Learning for transfer learning over deterministic methods. To deepen our understanding on how Soft Q-Learning compares to deterministic approaches on transfer learning tasks, we pursue the following research question:

> *RQ 1. How do possible benefits gained through transfer learning compare between Soft Q-Learning and traditional approaches?*

During our research, we noticed that models which trained longer on the source task tended to learn the target task slower. Researchers have claimed a link between overfitting on the source task and low performance on the target task [6], which is known as *negative transfer* [7]. As we could not detect definite proof for this relationship, we seek to investigate its presence regarding both traditional approaches and Soft Q-Learning. Knowing that a policy should not train on the source task for too long could save resources for fine-tuning on the target task, increasing final performance. Hence, we define a second research question:

> *RQ 2. Does overfitting on the source task lead to negative transfer in Soft Q-Learning or in traditional approaches?*

We pursued both research questions experimentally on the popular NES game *Super Mario Bros.*. A Double Deep Q-Network (DDQN) and a Soft Q-Network (SQN) agent learned to play the first level and reused the weights for learning the second level. Afterwards, we compared their performance to that of agents who only trained on the second level. We are, to the best of our knowledge, the first to directly investigate the transfer learning capabilities of Soft Q-Learning. As both levels are similar, but not identical in both visuals and tasks, they resemble real world use-cases. Our research aimed to shed light on Soft Q-Learning's transfer learning capabilities, to help practitioners take more informed choices between deterministic and stochastic policies for transfer learning, and to give founded advice on how far a policy should train on the source tasks to allow practitioners save time and energy in times where both work power and resources are scarce.

II. Background

A. Deep Q-Networks

Motivated by the success of deep neural networks in computer vision at the time, Mnih et al. [8] combine them with reinforcement learning techniques and manage to drastically outperform previous Q-Learning approaches. Their *Deep Q-Network* (DQN) utilizes a convolutional neural network (CNN) architecture with parameters θ that takes in raw pixel input s and outputs corresponding Q-values for a finite action space $\mathcal{A}$. For training, the model minimizes the mean squared error (MSE) between the current estimate $Q_\theta(s,a)$ and the target $r+\gamma\text{max}_{a'}Q_\theta(s',a')$ as in

$$\nabla_\theta \text{MSE}\big(Q_\theta(s,a), r+\gamma\text{max}_{a'}Q_\theta(s',a')\big) \tag{1}$$

As the resulting reinforcement learning algorithm is off-policy the authors opted for using a replay memory, also known as *Experience Replay Buffer* (ERB) [9], to store the latest N generated experience samples. During training, the buffer samples batches uniformly, thereby assigning the same importance to all stored experiences. This increases data efficiency as samples may be used more than once, and stabilizes training due to lower correlation between samples and an averaged behaviour distribution. However, the DQN implementation from Mnih et al. [8] suffers from two problems: *Moving targets* and *overestimation bias*. The former arises as every change to the current Q-value estimate also affects the target. Consequently, convergence is difficult. The latter stems from the standard DQN setup where the same network determines $\text{max}_{a'}$ and calculates the target. In this case, the Q-values will on average be larger than what they should be, leading to said overestimation bias [10]. To counter the problem of moving targets, Mnih et al. [11] developed *delayed DQNs*. They introduce a second neural network θ', the *target network*, to compute the targets. By not training its parameters, but only periodically replacing them with those of the original network – the *model network* θ – they alleviate the problem of moving targets.

B. Double Deep Q-Networks

To combat overestimation bias, van Hasselt et al. [10] propose combining Double Q-Learning [12] with DQNs calling the resulting method Double Deep Q-Networks (DDQN). By decoupling action selection from action evaluation, they mitigate the risk of frequently picking overestimated Q-values as targets. For efficiency, the authors use the target network to evaluate the action chosen by the model network. They argue that the networks differ enough to make it unlikely that both overestimate the same values. This changes the target formulation from equation (1) to

$$\nabla_\theta \text{MSE}\big(Q_\theta(s,a), r+\gamma Q_{\theta'}\big(s', \text{argmax}_{a'}Q_\theta(s',a')\big)\big).$$

The authors were able to report significantly reduced overestimation and higher policy quality when comparing their approach to delayed DQNs on different Atari games [10].

C. Soft Q-Learning

Inspired by previous maximum entropy-based policies [1], Haarnoja et al. [2] adapt Deep Q-Learning to prioritize high entropy, i.e. unfamiliar paths, during training. The authors show that under minor preconditions, the fixed point iterations

$$Q_{soft}(s_t,a_t) \leftarrow r_t + \gamma\mathbb{E}_{s_{t+1}\sim p_s}\big[V_{soft}(s_{t+1})\big]\ \forall s_t, a_t \tag{2}$$

$$V_{soft}(s_t) \leftarrow r_t + \alpha\int_{\mathcal{A}}\exp\left(\frac{Q_{soft}(s_t,\mathbf{a}')}{\alpha}\right)d\mathbf{a}'\ \forall s_t \tag{3}$$

converge towards Q^*_{soft} and V^*_{soft} respectively to yield the optimal policy

$$\pi^*_{\text{MaxEnt}}(a_t|s_t) = \exp\left(\frac{Q^*_{soft}(s_t,a_t)-V^*_{soft}(s_t)}{\alpha}\right) \quad \forall s_t, a_t \tag{4}$$

for a heat parameter $\alpha > 0$ and an action space $\mathcal{A}$. The policy therefore resembles a softmax function and no longer acts greedily compared to Q-Learning. They call their method *Soft Q-Learning* and since it allows incorporating universal function approximators like neural networks, researchers have called the corresponding models *Soft Q-Networks* (SQNs) [13]. Note that $\mathbf{a}'$ is a random variable, so its distribution affects the (Lebesgue) integral in mapping (3). The authors also elaborate how their approach can handle continuous action spaces through approximation of the integral in mapping (3). However, as our use-case contains a low-dimensional, discrete action space, we could directly perform mapping (2) and (3) without any approximation techniques. Moreover, later research [14] simplifies mapping (3) to

$$V_{soft}(s_t) \leftarrow \mathbb{E}_{a_t\sim\pi}\big[Q_{soft}(s_t,a_t) - \log\pi(a_t|s_t)\big]\ \forall s_t \tag{5}$$

As the entropy term $-\log\pi(a_t|s_t)$ is non-negative, mapping (5) implies that the policy prioritizes higher-entropy paths in that the policy is implicitly less confident. By scaling the entropy term with a fixed factor $\beta > 0$, we can further control how much the policy prioritizes high entropy paths. The heat parameter α is still important to control exploration. According to (4), for a given state s_t and actions a and a', the likelihood to chose a' over a equals

$$\frac{\pi^*_{\text{MaxEnt}}(a'|s_t)}{\pi^*_{\text{MaxEnt}}(a|s_t)} = \exp\left(\frac{Q^*_{soft}(s_t,a')-Q^*_{soft}(s_t,a)}{\alpha}\right).$$

A larger choice of the heat parameter α, therefore, implies a more uniform policy π.

Intuitively, SQNs do not only learn a single optimal, but multiple valid paths to achieve a task due to their stochastic policy and targeted exploration strategy. As the Q-values in both Q-Learning and Soft Q-Learning are subject to training noise, a Q-Learning agent may often find itself on sub-optimal paths that it cannot follow. A Soft Q-Learning agent may be capable to recognize previous paths and still succeed in the task.

Although research on Soft Q-Learning is scarce (searching "Soft Q-Learning" on Google Scholar returned only 739 results), researchers have successfully applied it to real-world use-cases. For example, the Japanese billion-dollar company DeNA Co., Ltd. applied Soft Q-Learning to distributed fleet control, reducing passenger waiting time by over 16.4% in a realistic simulation architecture [15]. Other applications of Soft Q-Learning encompass pedestrian simulations [16] and text generation [17].

D. Prioritized Experience Replay Buffer

Prioritized replay [18] enhances Experience Replay Buffers. It aims to more frequently select transitions from which the model can learn the most, instead of uniformly sampling from all experiences as in [8], [11]. Prioritized experience replay treats the ERB as a priority queue and draws training samples via Thompson sampling according to the respective TD-errors δ. Since the TD-error for new interactions with the environment is unknown, the authors suggest assigning a maximum priority to ensure that each observation affects training at least once. For efficiency, they further recommend only updating priorities for transitions after drawing them from the buffer and using them for training. The authors notice that focusing on a small subset of high-priority experiences results in reduced diversity in sample selection and overfitting. To address this issue, they introduce a scaling parameter α to interpolate between uniform sampling ($\alpha = 0$) and greedy prioritization ($\alpha = 1$). The probability $P(i)$ of sampling experience i follows

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

with $p_i = |\delta_i| + \epsilon$ to always ensure a minimum base priority. Since training aims to minimize the expected TD-error, sampling by priorities introduces a bias that the authors correct for through importance sampling. For each sampled experience, we obtain the corresponding importance sampling weight w_i according to

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$$

with N being the size of the buffer and β controlling the amount of correction. The authors report that gradually increasing β to 1 during training and setting α to 0.6 produces good results.

With their Prioritized Experience Replay Buffer (PERB), Schaul et al. [18] increased performance over ERBs in 41 out of 49 of the Atari games used in [11]. Their technique has been widely adopted by various deep reinforcement learning algorithms, such as the infamous Rainbow [19] or the work of Kapturowski et al. [20], and has successfully contributed to game-related reinforcement learning as in [21].

E. Transfer Learning in Reinforcement Learning

In the reinforcement learning domain, transfer learning allows to leverage knowledge gained in one context to improve training in another. For example, it can aim at obtaining better initial samples to get a head start in training (*jumpstart improvement*), requiring less samples for training overall (*learning speed improvement*) or final performance (*asymptotic improvement*) [22], [23].

Based on [23] and [24], we shorten the definition of transfer learning in the reinforcement learning context by [3] to

Given a set of source domains $\mathcal{M}_\mathbf{s}$ and a target domain $\mathcal{M}_t$, transfer learning aims to learn an optimal policy π^* for the target domain, by leveraging exterior information from $\mathcal{M}_s$ as well as interior information from $\mathcal{M}_t$.

Various taxonomies of transfer learning for reinforcement learning exist that differ both with respect to scenarios and methods. For example, Lazaric [22] characterizes scenarios by

- *Setting:* Similarity of state and action space as well as reward dynamics between source and target tasks.
- *Knowledge:* Type of transferred information, i.e., samples, representations or pre-trained policies.
- *Objectives:* Purpose behind employing transfer learning, usually related to enhanced training performance.

while Zhu et al. [3] employ a total of six dimensions. By accumulating the taxonomies in [22], [23], [25], we derive the following popular methods for transfer learning in reinforcement learning methods:

- *Starting-Point methods:* Pre-train policy on the source domains and continue training on the target domain.
- *Instance transfer:* Reuse samples from the source domains for training on the target domain.
- *Policy transfer:* Pre-train policies on the source domain. Incorporate their knowledge in training on the target domain, i.e., through weighting schemes.
- *Representation Transfer:* Exploit lower-dimensional representations of state, action or reward dynamics to generalize between domains.
- *Inter-Task Mapping:* Directly map between source state and action spaces and target state and action space to derive a target policy.
- *New reinforcement learning methods:* Some algorithms inherently tackle transfer learning by design. Examples are available in [25].

Moreover, some approaches address specific subdomains such as multi-agent reinforcement learning [26], [27].

If source and target domain are not sufficiently similar, the performance gain through transfer learning may generally decrease [7]. This problem is commonly known as negative transfer [7] and also occurs in the reinforcement learning context [25], [28]. Past research has commented on a suspected relation between negative transfer and overfitting [6].

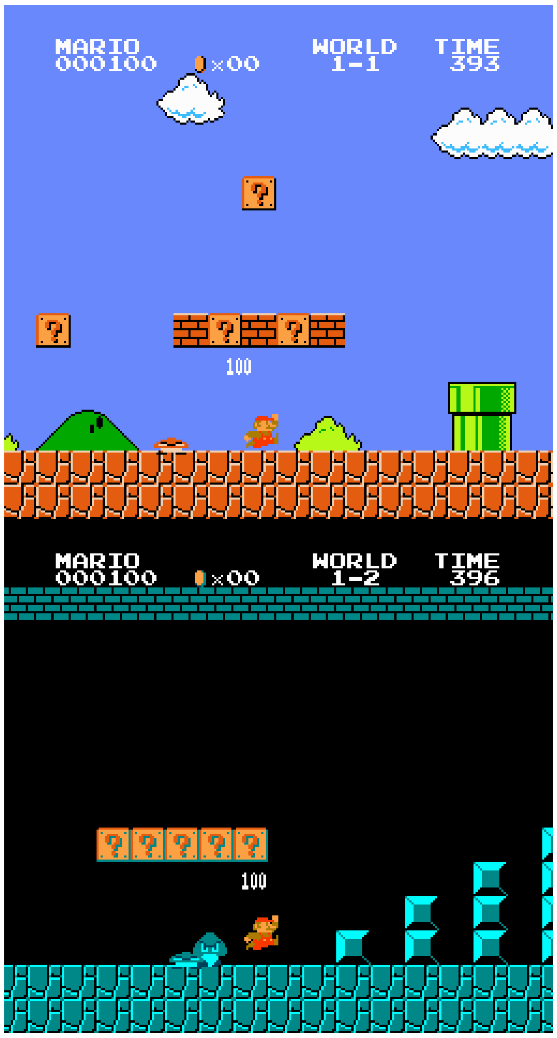

Fig. 1. Exemplary scenes from Super Mario Bros. levels 1-1 (top) and 1-2 (bottom).

Applications of transfer learning in reinforcement learning include training robots from simulated data [29], order dispatching in ride sharing platforms [30], deriving knowledge from sample-efficient 2D to cost-extensive 3D-environments [31] or gaming [32].

III. Methods

A. Environment

We measured and compared transfer learning capabilities on the popular NES game "Super Mario Bros.". The game consists of 32 levels in which the player has to control Super Mario through a parkour of obstacles (to reach a goal flag at the end). The player can choose from 256 distinct actions, all related to moving right, left, jumping, throwing fire balls (only once Mario picked up a fire flower) or idling.

We relied on a ready-to-use implementation of Super Mario Bros.[1] that, among other ready-to-use reinforcement learning environments of various NES games[2], has served for previous reinforcement learning research [33], [34]. We perceive Super Mario Bros. as particularly well suited for reinforcement learning studies as its implementation is computationally cheap and it is well known and therefore comprehensible to other researchers. It is also very convenient to work with. We build the environment by calling

```
from nes_py.wrappers import JoypadSpace
import gym_super_mario_bros
from gym_super_mario_bros.actions
    import SIMPLE_MOVEMENT
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, SIMPLE_MOVEMENT)
```

and perform a (random) action via

```
action = env.action_space.sample()
state, reward, done, info = env.step(action)
```

to receive a status, a reward, a confirmation whether the state is terminal and additional information about the game's state. Moreover, we render the game's current state by calling

```
env.render()
```

B. Reinforcement learning methods

To answer the first research question, we implemented SQN and chose DDQN as a comparative deterministic method. While delayed DQN would be more "vanilla" than DDQN, we found its performance unsuitable for the problem at hand. DDQN extends DQN just slightly and performs sufficiently well. The pseudo code in Alg. 1 and Alg. 2 summarize our SQN and DDQN implementations. To prevent overflows in the SQN's policy, we subtract the largest logit from all the logits so that the exponents in the softmax are always smaller or equal 1:

```
logits = (q_values)/self.heat_param

# Substract max for numeric stability
logits = logits - np.max(logits)

return logits
```

In [Appendix A], we display our implementations of the SQN's main characteristics – target calculation and action sampling.

C. Transfer learning approach

Our transfer learning task consisted of learning to finish one level and leveraging the gained knowledge to finish another, which is characteristic for a starting-point-method. In reference to the taxonomy by Lazaric [22], only the state spaces differ between levels and we only transfer the policy. Due to the exploratory nature of our research, we do not have a specific purpose for employing transfer learning.

As the various levels comprise similar visual assets and distinct challenges, they accurately resemble real world transfer learning use cases. We chose level 1-1 as the source domain

[1]https://pypi.org/project/gym-super-mario-bros/

[2]https://pypi.org/project/nes-py/

Algorithm 1 DDQN Training Loop

```
1:  Input: num_epochs, num_steps, decay_factor, batch_size, learning rate $\eta$, Polyak parameter $\tau$, discount factor $\gamma$, policy parameter $\epsilon$, PERB size $N$ and parameters $\alpha$ & $\beta$
2:  Initialize PERB with size $N$ and fill it to its capacity
3:  Initialize model network $\theta$
4:  Initialize delayed target network $\theta'$
5:  $\theta' \leftarrow \theta$                                  ▷ Copy weights
6:  for num_epochs do
7:      $\theta' \leftarrow (1-\tau)\theta' + \tau\theta$         ▷ Polyak Averaging
8:      for num_steps do
9:          Interact with the environment           ▷ $\epsilon$-greedy action
10:         Store ($s$, $a$, $r$, $s'$) in PERB with max prio
11:         if terminal then
12:             Reset environment
13:             $\epsilon \leftarrow \epsilon \cdot decay\_factor$             ▷ Anneal $\epsilon$
14:         end if
15:     end for
16:     Sample batch_size transitions from the PERB (controlled by $\alpha$) and obtain importance sampling weights $w$ (controlled by $\beta$)      ▷ see PERB implementation
17:     $y \leftarrow \begin{cases} r, & \text{if s'= terminal} \\ r + \gamma Q_{\theta'}(s', \text{argmax}_{a'}[Q_\theta(s', a')]), & \text{else} \end{cases}$
18:     $\delta \leftarrow \text{Huber}(y, Q_\theta(s, a))$
19:     Use $\delta$ to update prios in PERB
20:     Increase $\beta$ towards 1
21:     $\theta \leftarrow \theta + \eta[\nabla_\theta[w\delta]]$             ▷ SGD step with Adam
22: end for
```

Algorithm 2 SQN Training Loop

```
1:  Input: num_epochs, num_steps, batch_size, heat_param, entropy_factor, learning rate $\eta$, Polyak parameter $\tau$, discount factor $\gamma$, PERB size $N$ and parameters $\alpha$ & $\beta$
2:  Initialize PERB with size $N$ and fill it to its capacity
3:  Initialize model network $\theta$
4:  Initialize delayed target network $\theta'$
5:  $\theta' \leftarrow \theta$                                  ▷ Copy weights
6:  for num_epochs do
7:      $\theta' \leftarrow (1-\tau)\theta' + \tau\theta$         ▷ Polyak Averaging
8:      for num_steps do
9:          Interact with the environment           ▷ Entropy based
10:         Store ($s$, $a$, $r$, $s'$) in PERB with max prio
11:         if terminal then
12:             Reset environment
13:         end if
14:     end for
15:     Sample batch_size transitions from the PERB (controlled by $\alpha$) and obtain importance sampling weights $w$ (controlled by $\beta$)      ▷ see PERB implementation
16:     $\pi \leftarrow \text{softmax}(Q_{\theta'}(s')/heat\_param)$
17:     $entropy \leftarrow entropy\_factor \cdot \sum[\pi \log \pi]$
18:     $V_{soft} \leftarrow \sum[\pi Q_{\theta'}(s')] - entropy$
19:     $y \leftarrow \begin{cases} r, & \text{if s'= terminal} \\ r + \gamma V_{soft}, & \text{else} \end{cases}$
20:     $\delta \leftarrow \text{Huber}(y, Q_\theta(s, a))$
21:     Use $\delta$ to update prios in PERB
22:     Increase $\beta$ towards 1
23:     $\theta \leftarrow \theta + \eta[\nabla_\theta[w\delta]]$             ▷ SGD step with Adam
24: end for
```

and level 1-2 as the target domain. Because the two are among the most visually similar levels and the obstacles of level 1-1 also occur in level 1-2, we presumed that the models could transfer most information. Fig. 1 illustrates the visual similarities and differences between both levels.

For each DDQN and SQN, we trained two agents. The first agent learned on the source domain until the cumulative reward per episode converged. Subsequently, we trained the agent further on the target domain. As a baseline, the other agent learned only on the target domain (hereafter referred to as "untrained"). We tracked the agents' win rates and cumulative rewards per episode. To pursue our second research question, we also trained versions of the first agent on the target domain that had less training time on the source domain. We essentially achieved this by loading the final weights from training on the first level:

```
model_network.load_weights("pathToWeights")
```

Before we compared SQN and DDQN with regard to transfer learning, we validated our SQN and DDQN implementations by monitoring their performance on the first level.

D. Preprocessing

The agents received the game state as a rescaled 84x84 grey-scale picture and drew from a restricted action space of five actions: (1) *idle*, (2) *move right*, (3) *jump right*, (4) *move right and throw a fire ball*, (5) *jump right and throw a fireball*. Because consecutive frames are highly correlated, we accelerated training by repeating each action over four frames and passing the corresponding states as a stacked 4x84x84 image. This way, we effectively skipped three out of four frames and informed the agent about moving objects. We also returned the cumulative reward over each four stacked frames after normalizing it into the range of $[-\frac{60}{600}, \frac{60}{600}]$. The implementation returns a positive reward equal to distance (in pixels) covered to the right, a penalty equal to time passed and a penalty of -15 for dying but clips the total reward per frame into [-15,15]. With a discount factor of $\gamma = 0.9$, our absolute total discounted cumulative reward is therefore always less or equal $\frac{4 \cdot 15}{600} \frac{1}{1-\gamma} = \frac{60}{600} \cdot 10 = 1$, to stabilize training. However, our expected discounted cumulative reward is around 0.6 and other reward scalings may be more suitable. We employed a wrapper for skipping and stacking images and normalizing the rewards, as well as a separate function for resizing and greyscaling the image to exploit TensorFlow's enhanced efficiency in graph mode [Appendix B].

E. Hyperparameters

Starting from other practitioners' successful hyperparametrizations[3], we manually and iteratively adapted our hyperparameters.

Each epoch consisted of three environment steps and a single gradient step. To accelerate training, the models drew training samples through Prioritized Experience Replay. For sampling observations, we used a sum tree approach [Appendix C]. Initially, we filled the buffer to its full size of 32,000 samples on a uniform policy (warm start). We further used the standard parameters $\alpha = 0.6$ and $\beta = 0.4$, and increased β towards 1 over the course of training.

Fig. 2 displays our network architecture. All convolutional layers used a kernel size of 3 and a stride of 1 without padding. For pooling, we used a size of 2 and the dense layer contained 64 neurons. All models employed a learning rate of 0.00025 and a batch size of 64. As our PERB implementation's runtime scaled linearly in batch size, our firm submission deadline rendered a larger batch size infeasible.

Tab. I lists the respective hyperparameter choices for the DDQN and SQN. Both algorithms employed the same reward discount factor γ and Polyak averaging factor τ to prevent the problem of moving targets [11]. The initial exploration rate ϵ_{init}, the corresponding decay rate and the minimal exploration rate ϵ_{min} only affect the DDQN. For transfer learning, we reset ϵ on the second level. The heat parameter α and entropy scaling factor β only relate to Soft Q-Learning. After varying both parameters in size, we chose them proportional to the reward scaling factor. We found the entropy term for purely random samples of

$$-5 \cdot \frac{1}{600} \cdot \frac{1}{5} \cdot \log \frac{1}{5} = \frac{1}{375}$$

equals about 20% of the average reward.

TABLE I
DDQN AND SQN HYPERPARAMETER VALUES

Hyperparameters	Algorithm	
	DDQN	SQN
γ	0.9	0.9
τ	$\frac{1}{3750}$	$\frac{1}{3750}$
ϵ_{init}	0.3	-
ϵ decay rate	0.999	-
ϵ_{min}	0.1	-
α	-	$\frac{1}{600}$
β	-	$\frac{1}{600}$

[3]https://www.analyticsvidhya.com/blog/2021/06/playing-super-mario-bros-with-deep-reinforcement-learning/

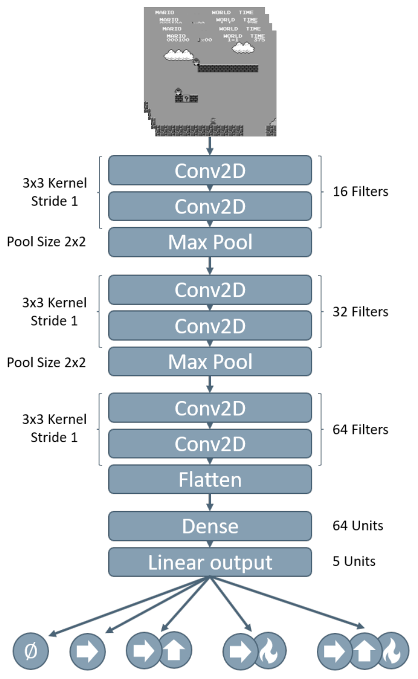

Fig. 2. Employed model network architecture

F. Hardware configuration

We implemented all models in Python v3.7.10 (v3.10.2) using the TensorFlow framework v2.3.0 (v2.10.0) with GPU support. For training, we used two local PCs running Windows 10 (11) with a Ryzen 9 5900X 12x 3.700GHz (Intel Core i9-10900K 10x 3.700Ghz) (CPU), 8GB RTX 3070 Ti FE (GPU) and 32GB DDR4-3200 (RAM).

IV. RESULTS

Fig. 3 and Fig. 4 show both models' cumulative reward and win rate per episode. The SQN's cumulative reward per episode increases continuously with minor noise. The DDQN's cumulative reward per episode increases for the first 2,000 episodes and stagnates thereafter. Overall, the SQN reached a 33% higher average cumulative reward than the DDQN. Win rate increased almost steadily for both models after 2,000 episodes. However, the SQN performed better than the DDQN with double the maximum win rate.

Fig. 5 and Fig. 6 display performance on level 1-2. For the SQN, no approach continuously dominates in cumulative reward, although the untrained model reaches the highest final performance. Regarding win rate however, the untrained model outperforms the pre-trained ones. Vice versa, for the DDQN, the untrained model outperforms regarding cumulative reward, but no model dominates in win rate. The SQN reached a strictly higher average cumulative reward and win rate in all training runs than the DDQN.

Google Colab usage restrictions and software issues cancelled many of our training runs. Time constrains prevented us to re-run all of the experiments so that for some we can only provide incomplete data.

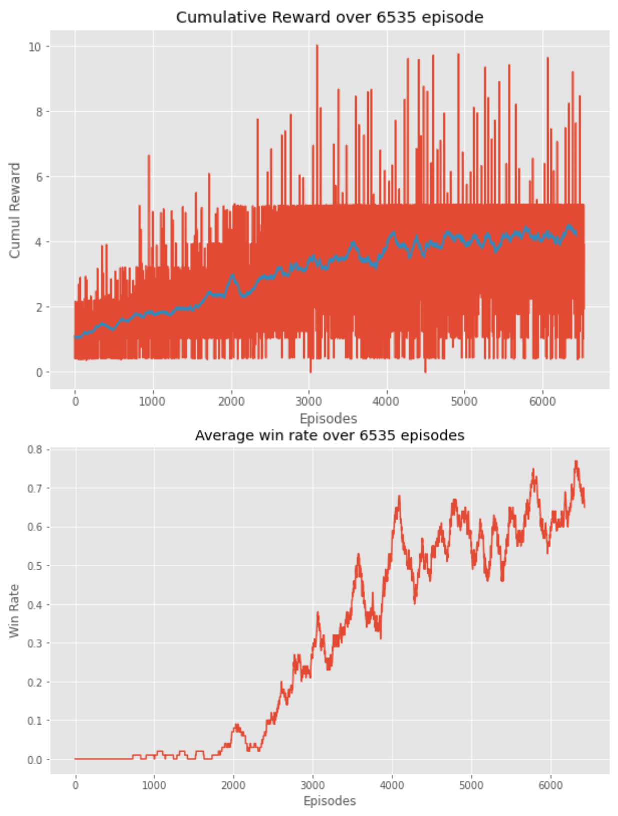

Fig. 3. SQN cumulative rewards (top) and wins (bottom) on level 1-1. The blue line and the wins are given as running averages over 100 episodes.

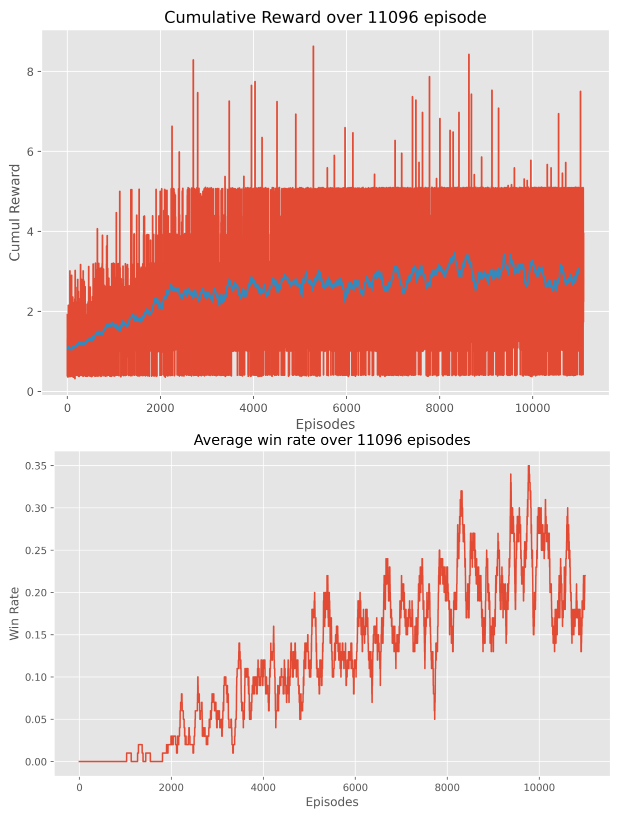

Fig. 4. DDQN cumulative rewards and (top) and wins (bottom) on level 1-1. The blue line and the wins are given as running averages over 100 episodes.

V. Discussion

We confirm that both models successfully learned to finish the level based on their significant win rate. Therefore, we consider our implementations correct and working.

As the cumulative reward of the untrained policy was the highest, neither the SQN nor the DDQN seemed to benefit from transfer learning at all. Instead, the untrained versions seem to learn even faster than the pre-trained ones. Also, those versions of the policy with less training time do not appear to have learned any faster than the fully trained ones. Hence, we could not observe any advantage or disadvantage of Soft Q-Learning over a traditional approach for transfer learning. Moreover, we could also not detect any effect of training time in the source domain on performance in the target domain.

Since our data do not show any (positive) effect through transfer learning, we assume that key mechanisms for our experiment were not present. For example, the source and target domains may not have been compatible for transfer learning or we needed to apply a more sophisticated transfer learning approach. Level 1-2, for instance, contains a very narrow passage, with an obstacle right behind it where the agent has a limited time window for performing the correct action (jumping) (Fig. 7). Passing it requires extensive over-fitting, which, hypothetically, may be easier for the untrained models than the pre-trained ones. Unfortunately, we were out of time to repeat our experiment on another target level. Also, the heat parameter may have caused the low learning rate of the pre-trained SQNs. For the final Q-estimators in level 1-1, the policy became almost deterministic, so that it initially explored very little on level 1-2. A higher heat parameter may have increased initial exploration allowing the policy to become more flexible. However, our deadline prevented us from trying additional hyperparameter settings. Another reason may be that we only employed a single level as the source domain. Here, it may have been beneficial to train the models on multiple levels instead, to achieve better generalization and possibly better knowledge transfer. Just as well, we could have been in bad luck and required a larger number of observations to receive significant and reliable results. Overall, we consider our research questions unanswered and call for future research to adapt our approach and take them on. Apart from other source and target domains, future research may also consider additional entropy-based and deterministic approaches to better generalize their findings. We hope our work can inspire other researchers to investigate how to approach transfer learning both regarding methods and training time, as answering either of our research questions may spare practitioners expensive resources such as time and energy. Yet, we do not consider our research unsuccessful as we proposed an experimental setup, provide a ready-to-use implementation and identified and put forth major challenges that future research can build upon.

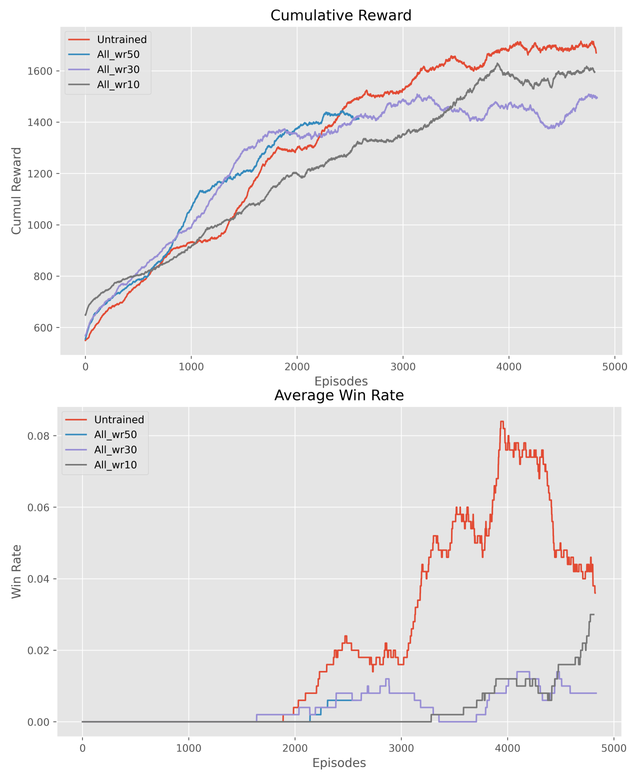

Fig. 5. SQN cumulative reward (top) and wins (bottom) on level 1-2 as running averages over 500 episodes. The lines correspond to the untrained model (red), and the pre-trained model from where it had achieved 10% (grey), 30% (purple) and 50% (blue) average win rate on level 1-1. Cumulative reward is multiplied by 600 to match the original reward distribution.

VI. Conclusion

In this project, we investigated the transfer learning capabilities of Soft Q-Learning and Double Deep Q-Learning. Particularly, we explored the benefits of transfer learning regarding both methods and an alleged relationship between overfitting and negative transfer. We pursued our research questions experimentally on the first two levels of the NES game Super Mario Bros. Although our experiments did not meet our expectations, we proposed and executed an experimental setup, provide a ready-to-use implementation and identified and put forth major challenges that future research can build upon. To the best of our knowledge, we are the first to directly investigate the transfer learning capabilities of Soft Q-Learning and conclude that by building on our ground work, researchers could generate insights into transfer learning that would save resources as time and energy in times where both work power and resources are scarce.

Acknowledgments

This research was funded by spilled lentils, Rohrbrüche, surging energy prices and two cursed Google Colab Pro subscriptions cancelling our trainings. We would like to thank our supervisor Leon for his thorough support (seriously).

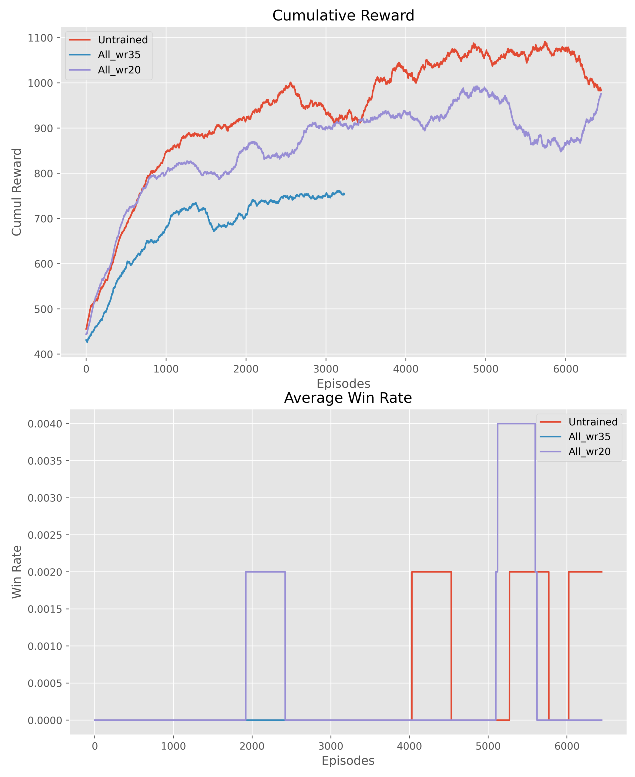

Fig. 6. DDQN cumulative reward (top) and wins (bottom) on level 1-2 as running averages over 500 episodes. The lines correspond to the untrained model (red), and the pre-trained model from where it had achieved 20% (purple) and 35% (blue) average win rate on level 1-1. Cumulative reward is multiplied by 600 to match the original reward distribution.

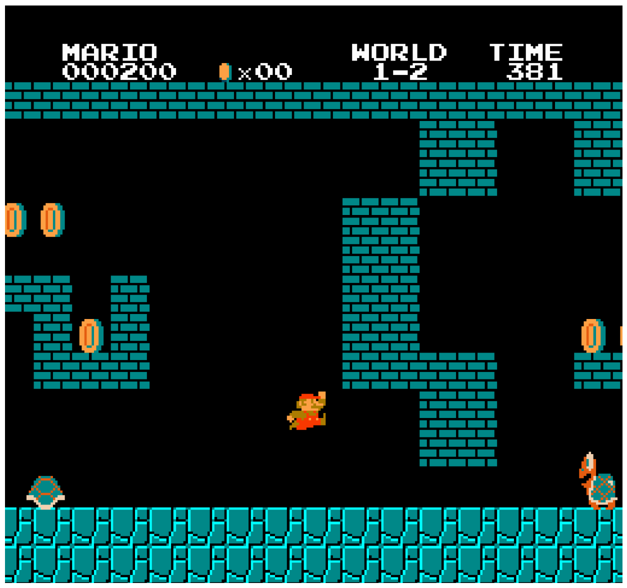

Fig. 7. Narrow passage in level 1-2, which is particularly difficult to pass due to the turtle patrolling the exit.

References

[1] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum Entropy Inverse Reinforcement Learning," 2008, p. 6. [Online]. Available: https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf?source=post_page

[2] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, "Reinforcement Learning with Deep Energy-Based Policies," Jul. 2017. [Online]. Available: http://arxiv.org/abs/1702.08165

[3] Z. Zhu, K. Lin, A. K. Jain, and J. Zhou, "Transfer Learning in Deep Reinforcement Learning: A Survey," 2022, arXiv:2009.07888 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2009.07888

[4] T. Haarnoja, V. Pong, A. Zhou, M. Dalal, P. Abbeel, and S. Levine, "Composable Deep Reinforcement Learning for Robotic Manipulation," 2018, arXiv:1803.06773 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1803.06773

[5] M. Nemecek and R. Parr, "Policy Caches with Successor Features," in *Proceedings of the 38th International Conference on Machine Learning*. PMLR, 2021, pp. 8025–8033. [Online]. Available: https://proceedings.mlr.press/v139/nemecek21a.html

[6] S. Gamrian and Y. Goldberg, "Transfer Learning for Related Reinforcement Learning Tasks via Image-to-Image Translation," in *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 2019, pp. 2063–2072, iSSN: 2640-3498. [Online]. Available: https://proceedings.mlr.press/v97/gamrian19a.html

[7] Z. Wang, Z. Dai, B. Poczos, and J. Carbonell, "Characterizing and Avoiding Negative Transfer," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, 2019, pp. 11 285–11 294. [Online]. Available: https://doi.org/10.1109/CVPR.2019.01155

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[9] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3, pp. 293–321, 1992. [Online]. Available: https://doi.org/10.1007/BF00992699

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016. [Online]. Available: https://doi.org/10.1609/aaai.v30i1.10295

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[12] H. van Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems*, vol. 23. Curran Associates, Inc., 2010. [Online]. Available: https://papers.nips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html

[13] C. Yan, Q. Zhang, Z. Liu, X. Wang, and B. Liang, "Control of Free-Floating Space Robots to Capture Targets Using Soft Q-Learning," in *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Kuala Lumpur, Malaysia: IEEE, 2018, pp. 654–660. [Online]. Available: https://doi.org/10.1109/ROBIO.2018.8665049

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," 2018, arXiv:1801.01290 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1801.01290

[15] T. Oda and Y. Tachibana, "Distributed Fleet Control with Maximum Entropy Deep Reinforcement Learning," in *Machine Learning for Intelligent Transportation Systems, Poster Submission*, 2018, p. 7.

[16] F. Martinez-Gil, M. Lozano, I. García-Fernández, P. Romero, D. Serra, and R. Sebastián, "Using Inverse Reinforcement Learning with Real Trajectories to Get More Trustworthy Pedestrian Simulations," *Mathematics*, vol. 8, no. 9, p. 1479, 2020. [Online]. Available: https://doi.org/10.3390/math8091479

[17] H. Guo, B. Tan, Z. Liu, E. P. Xing, and Z. Hu, "Text Generation with Efficient (Soft) Q-Learning," 2021. [Online]. Available: http://arxiv.org/abs/2106.07704

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *4th International Conference on Learning Representations, ICLR 2016, May 2-4, 2016, Conference Track Proceedings.*, San Juan, Puerto Rico, 2016. [Online]. Available: http://arxiv.org/abs/1511.05952

[19] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018. [Online]. Available: https://doi.org/10.1609/aaai.v32i1.11796

[20] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney, "Recurrent Experience Replay in Distributed Reinforcement Learning," in *7th International Conference on Learning Representations, ICLR 2019, May 6-9, 2019.*, New Orleans, LA, USA, 2019.

[21] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-03051-4

[22] A. Lazaric, "Transfer in Reinforcement Learning: A Framework and a Survey," in *Reinforcement Learning*, M. Wiering and M. van Otterlo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 12, pp. 143–173. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27645-3_5

[23] M. E. Taylor and P. Stone, "Transfer Learning for Reinforcement Learning Domains: A Survey," *The Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.

[24] Y. Zhan and M. E. Taylor, "Online Transfer Learning in Reinforcement Learning Domain," 2015, p. 8. [Online]. Available: https://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11646

[25] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264. [Online]. Available: https://doi.org/10.4018/978-1-60566-766-9.ch011

[26] G. Boutsioukis, I. Partalas, and I. Vlahavas, "Transfer Learning in Multi-Agent Reinforcement Learning Domains," in *Recent Advances in Reinforcement Learning*, ser. Lecture Notes in Computer Science, S. Sanner and M. Hutter, Eds. Berlin, Heidelberg: Springer, 2012, pp. 249–260. [Online]. Available: https://doi.org/10.1007/978-3-642-29946-9_25

[27] F. L. D. Silva and A. H. R. Costa, "A Survey on Transfer Learning for Multiagent Reinforcement Learning Systems," *Journal of Artificial Intelligence Research*, vol. 64, pp. 645–703, 2019. [Online]. Available: https://doi.org/10.1613/jair.1.11396

[28] S. A. H. Minoofam, A. Bastanfard, and M. R. Keyvanpour, "TRCLA: A Transfer Learning Approach to Reduce Negative Transfer for Cellular Learning Automata," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–10, 2021. [Online]. Available: https://doi.org/10.1109/TNNLS.2021.3106705

[29] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. Canberra, ACT, Australia: IEEE, 2020, pp. 737–744. [Online]. Available: https://doi.org/10.1109/SSCI47803.2020.9308468

[30] R. Wan, S. Zhang, C. Shi, S. Luo, and R. Song, "Pattern Transfer Learning for Reinforcement Learning in Order Dispatching," 2021, arXiv:2105.13218 [cs]. [Online]. Available: http://arxiv.org/abs/2105.13218

[31] L. A. Celiberto, J. P. Matsuura, R. L. de Mantaras, and R. A. Bianchi, "Using Transfer Learning to Speed-Up Reinforcement Learning: A Cased-Based Approach," in *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*. Sao Bernardo do Campo: IEEE, 2010, pp. 55–60. [Online]. Available: https://doi.org/10.1109/LARS.2010.24

[32] K. Shao, Y. Zhu, and D. Zhao, "StarCraft Micromanagement With Reinforcement Learning and Curriculum Transfer Learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 3, no. 1, pp. 73–84, 2019. [Online]. Available: https://doi.org/10.1109/TETCI.2018.2823329

[33] D. Chen, Y. Wang, and W. Gao, "Combining a gradient-based method and an evolution strategy for multi-objective reinforcement learning," *Applied Intelligence*, vol. 50, no. 10, pp. 3301–3317, 2020. [Online]. Available: https://doi.org/10.1007/s10489-020-01702-7

[34] D. G. LeBlanc and G. Lee, "General Deep Reinforcement Learning in NES Games," *Proceedings of the Canadian Conference on Artificial Intelligence*, 2021. [Online]. Available: https://doi.org/10.21428/594757db.8472938b

APPENDIX

A. Soft Q-Learning

1) Target Calculation:

```
for s, a, r, s_prime, terminal in dataset:
    # Get q-values from target network
    q_prime = target_network(s_prime)


    logits = q_prime/policy.heat_param
    logits = logits - tf.math.reduce_max(logits, axis=1, keepdims=True)
    pi = tf.math.softmax(logits, axis=1)

    if np.sum(np.isnan(pi)) > 0:
        pi = tf.where(tf.equal(tf.reduce_max(q_prime, axis=1, keep_dims=True), q_prime),
                      tf.constant(1, shape=pi.shape),
                      tf.constant(0, shape=pi.shape))

    # Calculate the entropy of the policy pi in state s
    entropy = entropy_factor * tf.reduce_sum(pi*tf.math.log(pi + 0.001), axis=1)

    # Calculate v soft' according to Equation (3) in Haarnoja et al. 2018
    # https://arxiv.org/abs/1801.01290
    V_soft_prime = tf.reduce_sum(pi*(q_prime), axis=1) - entropy
```

2) Policy:

```
# Policy Class
# Doesn't use any tf functions as numpy turned out to be faster, even compared to graph mode
class Policy():
    """Policy used to sample an action given the q-values of the current state
    """

    def __init__(self, heat_param: float=1):
        """Initialize the policy

        Arguments:
            heat_param (float): Heat parameter that controls the exploration of the policy
        """

        self.heat_param = heat_param


    def __call__(self, q_values):
        """Sample action according to the q-values

        Arguments:
            q_values (tf.Tensor): q-values of the current state given by the model network

        Returns:
            action (int): To-be-performed action
        """

        p = self.calc_probability(q_values)
        action = np.random.choice(len(q_values), p=p)

        return action


    def calc_probability(self, q_values):
        """Calculate the entropy-based probabilities for the given q-values

        Arguments:
            q_values (tf.Tensor): q-values of the current state given by the model network

        Returns:
            p (nd.array): Array of probabilities
        """

        # Calculate logits
        logits = self.calc_logits(q_values)

        p = np.exp(logits)
        p /= np.sum(p)

        # If p contains nan (occurs if one value is much larger than the others)
        # then replace the max with 1 and all others with zero
        # --> deterministic sampling for numeric stability
        if np.sum(np.isnan(p)) > 0:
            p = np.where(q_values == np.max(q_values), 1, 0)

        return p


    def calc_logits(self, q_values):
        """Calculate logits for the given q-values scaled by the heat parameter

        Arguments:
            q_values (tf.Tensor): q-values of the current state given by the model network

        Returns:
            logits (nd.array): Logits of the q values
        """

        logits = (q_values)/self.heat_param

        # Substract max for numeric stability
        logits = logits - np.max(logits)

        return logits
```

B. Preprocessing

1) Frame Skipping:

```
class SkipObs(gym.Wrapper):
    """Wrapper class that repeats the chosen action for a given amount of frames
    """

    def __init__(self, env=None, skip: int=4, reward_scale_factor: int=1):
        """Initialize the wrapper

        Arguments:
            env (gym): Environment the wrapper should be applied to
            skip (int): Number of frames to skip
            reward_scale_factor (int): Factor to scale the rewards with
        """

        super(SkipObs, self).__init__(env)

        self.skip = skip
        self.reward_scale_factor = reward_scale_factor


    def step(self, action: int):
        """ Function to overwrite the environments step function

        Arguments:
            action (int): The chosen action

        Returns:
            obs (nd.array): Last observation from the environment
            total_reward (float): Total reward from all steps (including skipped ones)
            done (boolean): Whether the episode is finished or not
            info (dict): Dictionary containing information about the environments state
        """

        total_reward = 0.0
        done = None

        # Apply the same action to "skip" many frames and calculate the total reward
        for _ in range(self.skip):

            obs, reward, done, info = self.env.step(action)

            # Reward scaling
            total_reward += reward / self.reward_scale_factor
            if done:
                break

        return obs, total_reward, done, info
```

2) Wrapper Combination:

```
# For environment creation
def make_env(level: str, movement_type: list, num_skip: int=4, num_stack: int=4, reward_scale_factor: int=1
                                                                ):
    """Function to apply all wrappers to the environment

    Arguments:
        level (str): Level to create
        movement_type (list): Defines the joypad space e.g. allow movement to the right only
        num_skip (int): Number of frames to skip
        num_stack (int): Number of frames stack
        reward_scale_factor (int): Factor to scale the rewards with

    Returns:
        env (gym): Wrapped environment
    """

    # Create Env from given level
    env = gym_super_mario_bros.make(level)

    # Select joypad space
    env = JoypadSpace(env=env, actions=movement_type)

    # Skip "num_skip" frames
    env = SkipObs(env=env, skip=num_skip, reward_scale_factor=reward_scale_factor)

    # Wrapper that stacks "num_stack" frames resulting in observations being of size (num_stack,240,256,3)
    env = FrameStack(env=env, num_stack=num_stack)

    return env
```

3) Image Resizing and Greyscaling:

```
@tf.function
def preprocess_obs(obs, resize_env):
    """ Function to preprocess the observations before they are fed into the network
    1. Rescale the pixel values between [0,1]
    2. Convert from RGB to greyscale
    3. Resize to given resize_env value
    4. Transpose stacked images such that consecutive ones are within the channel dimension

    Arguments:
        obs (gym.wrappers.LazyFrames): Stacked observations from the environment
        resize_env (tuple): Tuple containing the new size to which the observation should be adjusted

    Returns:
        preprocessed_obs (tf.tensor): Preprocessed observation
    """

    preprocessed_obs = tf.convert_to_tensor(obs/255)
    preprocessed_obs = tf.image.rgb_to_grayscale(preprocessed_obs)
    preprocessed_obs = tf.image.resize(preprocessed_obs, size=resize_env)
    preprocessed_obs = tf.transpose(tf.squeeze(preprocessed_obs), perm=[1,2,0])

    return preprocessed_obs
```

C. Prioritized Experience Replay Buffer

1) Weight Adjustment:

```
def adjust_priority(self, priority: float):
    """Function to adjust a priority

    Arguments:
        priority (float): To-be-adjusted priority

    Returns:
        adjusted_priority (float): Adjusted priority
    """

    # Add min prio to current prio and raise sum to the power of alpha
    adjusted_priority = np.power(priority + self.min_priority, self.alpha)

    return adjusted_priority
```

2) *Sampling:*

```
def sample(self, num_samples: int):
    """Function for sampling

    Arguments:
        num_samples (int): Number of samples to draw from the buffer

    Returns:
        dataset (tf.data.Dataset): Dataset containing the sampled experiences
        sampled_idxs (list): List of the sampled experience indices
        importance_sampling_weights (list): List of the corresponding importance sampling weights
    """


    sampled_idxs = []
    importance_sampling_weights = []
    sample_no = 0

    # Sample "num_samples" indices from the sum tree
    while sample_no < num_samples:

        # Uniformly sample between 0 and the summed priority (stored in the root node)
        # used for retrieving an experience sample index (i.e., sampling according to priorities)
        sample_val = np.random.uniform(0, self.base_node.value)
        sampled_node = retrieve(sample_val, self.base_node)

        if sampled_node.idx < self.available_samples - 1:

            sampled_idxs.append(sampled_node.idx)

            # Divide the sampled nodes' priority by the summed priority and caluclate the importance
                                                            sampling weight
            p = sampled_node.value / self.base_node.value
            importance_sampling_weights.append((self.available_samples + 1) * p)

            sample_no += 1


    # Apply the beta factor and normalize so that the maximum importance sampling weight < 1
    importance_sampling_weights = np.array(importance_sampling_weights)
    importance_sampling_weights = np.power(importance_sampling_weights, -self.beta)
    importance_sampling_weights = importance_sampling_weights / np.max(importance_sampling_weights)


    obs, actions, rewards, obs_prime, terminal = [], [], [], [], []

    for idx in sampled_idxs:
        # Append the sampled experience components from the buffer to the respective lists
        obs.append(self.buffer[idx][self.obs_idx])
        actions.append(self.buffer[idx][self.action_idx])
        rewards.append(self.buffer[idx][self.reward_idx])
        obs_prime.append(self.buffer[idx][self.obs_prime_idx])
        terminal.append(self.buffer[idx][self.terminal_idx])

    dataset = tf.data.Dataset.from_tensor_slices((obs, actions, rewards, obs_prime, terminal)).batch(
                                                            num_samples)

    return dataset, sampled_idxs, importance_sampling_weights
```