Investigating the benefits of adding self-attention to a Deep Q-Network

Roberto Schiavone[1][0009−0001−9167−1496]
r.schiavone@vu.nl

Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands

Abstract. In recent years, the Transformer architecture has revolutionized how we process sequential data, but its applications in online RL settings are still very limited. In this paper a Transformer architecture is applied as the first layer inside a DQN in order to process in parallel a set of observations. The hypothesis is that the encoder-decoder architecture can extract more information from an observation than a linear layer. Furthermore, the multi-head attention can help the network focus on what experiences are deemed most important. In the end I provide detailed results, backed by a recently suggested statistical framework, to show that the proposed architecture surpasses the baseline on 1 of the 3 classic control environments used for evaluation, but on average it performs worse, providing a good starting point for future research.

Keywords: Reinforcement Learning · Transformers · Classic control

1 Introduction

Reinforcement Learning (RL) is the field of Artificial Intelligence that focuses on training an intelligent agent to make sequential decisions in an environment in order to maximize a specific goal or reward. It is inspired by how humans and animals learn through trial and error, by taking actions and receiving feedback. The agent does not know *a priori* which actions to take, but it must instead discover the ones that yield the highest reward by trying them. In some cases, actions affect not only the immediate reward but also the next state and, through that, all subsequent rewards [36,14].

Unfortunately, despite the progresses made in the field, pioneered by the seminal works on Deep Reinforcement Learning [26,25], RL is ridden with problems that still make it an unfeasible paradigm for most Machine Learning (ML) tasks. RL is sample inefficient, requiring hundreds of millions of steps for benchmarks such as the Atari Learning Environment (ALE) [6], corresponding to tens to hundreds of hours of human play experience [19,17]. Good reward functions that encourage the actual wanted behavior are difficult to design, and prone to fail in unexpected ways [19,2]; the final results are unstable and hard to reproduce due to random chance generating too much variance, and thus playing a major role during training [16,19]. Randomness can also lead the agents to get stuck in local optima from which it is nigh-on-impossible to escape for the rest of the training phase [19].

In recent years, the Transformer architecture has reshaped the landscape of AI, since it is extremely well-suited for applications that deal with sequential data, such as Natural Language Processing [29,10] and Computer Vision [24]. By being able to process sequences as a whole rather than token by token, Transformers do not suffer from long dependency issues and do not have to deal with past information, since there are no past states to account for.

In this paper I explore the impact of using a Transformer architecture as the first layer of a Deep Q-Network (DQN), and I train it on chosen toy environments in the context of online[1], off-policy[2] learning. The key idea behind it is that a set of observations can be seen as a sequence on which a Transformer architecture can be trained. Furthermore, the multi-head attention mechanism can help capture what experiences from the set are the most important.

In the end I show that the architecture I propose, called Transformer Q-Network (TQN), is not only able to learn and converge to the optimal policy over time, but it is also able to yield a higher score than the baseline on 1 of the 3 chosen environments. The results are further corroborated following the statistical framework proposed by `rliable` [1].

[1] In an online setting, instead of relying on previously collected data, the agent explores the environment and it collects experiences by actively interacting with it.

[2] The agent learns not only from the current interaction, but only from previous interactions that have been gathered with a different policy.

2 Related Work

2.1 Transformers in Offline RL

Trajectory Transformer The Trajectory Transformer trains on sequences of transitions $(state, action, reward)$ and it repurposes beam search as a reward-maximizing strategy [20]. The Decision Transformer, explained below, is an improved iteration of the Trajectory transformer.

Decision Transformer Decision Transformers are able to learn meaningful patterns from a dataset of previously gathered experiences. A linear layer embeds each token, which is then augmented with information about the current timestep. Lastly, a Generative Pretrained Transformer (GPT) model predicts the future actions. By leveraging a causally masked Transformer, the model exceeds the performance of previous state-of-the-art (SOTA) offline RL baselines for Atari, OpenAI Gym, and Key-to-Door tasks [12]. My approach draws inspiration from the Decision Transformer, but it differs in the implementation and how it is applied to online learning instead.

Q-Transformer Q-Transformer is the latest architecture for offline Q-learning with a Transformer model, only one month old at the moment of writing. The Q-Transformer architecture discretizes the action space and uses an autoregressive model update to avoid the curse of dimensionality on the discretized actions. The encoding of each observation is concatenated with the embeddings from the previous predicted action and it is then processed by Transformer layers. One-hot action vectors are used to predict the Q-values of the next actions. The architecture adopts Conservative Q-learning (CQL) to minimize the over-estimation of Q-values.[11]. Q-Transformer is the latest offline RL architecture that successfully uses a Transformer to improve the performance of Q-learning, and it beats the Decision Transformer on all the previously mentioned benchmarks.

2.2 State of the Art: MEME

Recurrent Replay Distributed DQN (R2D2) R2D2 incorporates a Long Short-Term Memory (LSTM) into an n-step, dueling DQN with a prioritized distributed replay in order to capture temporal dependencies in the data. A Long Short-Term Memory (LSTM) layer after the convolutional one and an aggressive approach in experience prioritization help achieve a better performance than Rainbow [22]. The *stored state* training strategy stores experiences from different agents in an experience replay that is subsequently fed to the agent, the *burn-in* strategy uses a portion of the experiences to bring the network to the starting training state [22]. R2D2 is one of the earliest and most successful examples to explicitly use a layer for sequential data, in this case a LSTM.

NeverGiveUp (NGU) NGU adds to the approach proposed by R2D2 an a universal value function approximator to approximate the optimal value function. The model introduces a Retraced Q-learning loss function, and it distributes training in order to collect a large amount of experiences from parallelized environments [4]. NGU encourages the agent to continue exploring the environment, hence the name, even if it encounters difficulties, so that the agent can learn better policies. NGU combines curiosity-driven exploration with distributed RL agents; the model is the cornerstone for Agent57 [3].

Agent57 Agent57 splits the state-action value function in two different components, with different parameterized sets of weights, to train two neural networks sharing the same architecture. As a result, the training stability significantly improves. Furthermore, Agent57 introduces the *meta-controller* as a mechanism to select the best policy during training [3].

MEME MEME builds upon the previous 3 architectures, R2D2, NGU and Agent57. The model introduces a trust region to determine which samples contribute to the loss, employs a variation of the NFNet architecture to stabilize the network without using layer normalization, and achieves more robust behavior through policy distillation[3] [21]. I find it worth mentioning MEME because, at the moment of writing, it is the state of the art for online RL on the Atari benchmark.

[3] Policy distillation is the transfer of knowledge from a more complex policy to a simpler one, allowing the simpler policy to learn and perform well while reducing at the same time the computational complexity of training it.

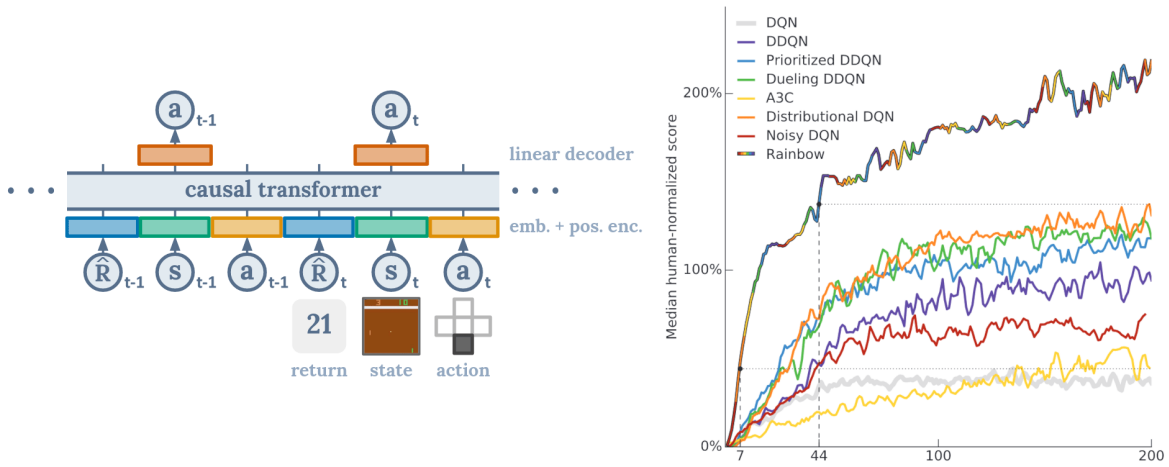

Fig. 1: On the left, the Decision Transformer architecture [12]. On the right, Rainbow (discussed later, see Chasing Rainbows) performance on the Atari benchmark compared to its single components [17].

3 Background

3.1 Reinforcement Learning

RL is the area of Machine Learning (ML) that deals with sequential decision-making. In a RL setting, an agent interacts with the environment through trial-and-error approach and, for each action taken, it receives either a positive or negative reward as unique feedback. The agent doesn't know *a priori* which actions to take, but it must instead discover the ones that yield the highest reward by trying them. In some cases, actions affect not only the immediate reward but also the next state and, through that, all subsequent rewards [36,14].

The sequence of actions and observations, $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$, can be modeled as a Markov decision process (MDP) in which each sequence is a distinct state; the algorithm converges to the optimal action-value function by using the Bellman equation in an iterative update [36,14,26].

3.2 Markov decision process

An MDP [8] is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}_a, \mathcal{R}_a, \gamma)$, where:

- $\mathcal{S}$ is a finite set of states called the *state space*,
- $\mathcal{A}$ is a finite set of actions called the *action space*,
- $\mathcal{P}_a = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t+1$,
- $\mathcal{R}_a(s, s')$ is the reward received after transitioning from state s to state s', due to action a,
- $\gamma \in [0, 1]$ is a discount factor.

3.3 Bellman equation

For a given policy π and a state s and action a in the MDP, the Bellman equation for state-action values (Q-values) is defined as follows [8]:

$$Q_\pi(s, a) = \sum [P(s' \mid s, a) * (R(s, a, s') + \gamma * \sum [\pi(a' \mid s') * Q_\pi(s', a')])]$$

- $Q_\pi(s, a)$: the expected cumulative reward (Q-value) of taking action a in state s and then following policy π.
- $P(s' \mid s, a)$: the probability of transitioning to state s' from state s when taking action a.
- $R(s, a, s')$: the immediate reward received when transitioning from state s to state s' by taking action a.
- γ: the discount factor.
- $\pi(a' \mid s')$: the probability of taking action a' in state s' according to policy π.

3.4 Exploration vs. Exploitation

In an offline RL setting, an agent merely learns from a dataset of experiences separately obtained. In online RL problems instead, an agent has to face the peculiar exploration-exploitation dilemma. During training, the agent must choose whether to explore more states in order to discover a better policy, or to exploit the information already available to maximize the reward in the short term, but risk being stuck in a local optimum. The exploration-exploitation problem is not fully solved and various partial solutions have been proposed throughout the years. Below, I present a non-exhaustive list of the most used approaches.

Epsilon-Greedy In an ϵ-greedy policy, a parameter ϵ determines the action taken by the agent. If a random number sampled from a continuous uniform distribution $\mathcal{U}(0, 1)$ is less than ϵ, then the agent takes a random action to explore the environment, otherwise it exploits its knowledge of the environment by taking the action that in the past yielded the best results.

ϵ-greedy is the most used policy because it is easy to understand, but in some cases it is not the most efficient. Furthermore, it is difficult to find the best value for ϵ depending on the task. A commonly adopted tactic is to linearly or exponentially decrease ϵ over time by an extra parameter ϵ_{decay}. Thus, the agent performs more explorations of the environment at the beginning of the training, and takes less random actions further down the road. For this paper, ϵ-greedy is the most relevant approach because it is used in Deep Q-learning.

Thompson sampling and Upper-Confidence-Bounds (UCB) Thompson sampling is a heuristic where the action is chosen by drawing it from a probability distribution over possible values of unknown parameters. It then chooses the action assuming that the sampled values are the true ones, and over time it adjusts the probability distribution according to the feedback received from the environment [37].

UCB is another strategy used in online RL with partial information feedback. Compared to Thompson sampling, UCB takes a deterministic approach instead. Taking into consideration an action's expected reward and a confidence interval representing its uncertainty, an upper confidence bound is computed for each action, then the one with the highest upper confidence bound is chosen. This way the agent is encouraged to choose an exploratory action by selecting the one with the highest uncertainty.

Boltzmann distributions Another approach is to model the probabilities of choosing the different actions with a Boltzmann distribution. A temperature parameter τ controls the level of randomness; a higher temperature means a more uniform distribution, making it easier to choose a random action, while a lower temperature pushes the agent to exploit the actions that are deemed better.

Adding noise The last approach worth mentioning that is used to balance exploration-exploitation is to add noise to the current observation (see also Chasing Rainbows). By adding noise to its inputs, the actions taken from the agent are less deterministic, thus encouraging exploration.

3.5 Q-learning

Q-learning is an off-policy TD control algorithm defined by [41]

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a, s') + \gamma \max_a Q(s', a) - Q(s, a) \right].$$

In this case, the learned action-value function Q directly approximates q_*, the optimal action-value function, independent of the policy being followed.

3.6 Deep Q-learning

Deep Q-learning is a variant of Q-learning where the model is a convolutional neural network. The input of the neural network is the state of the environment, and the output is a value function estimating future rewards [25].

The network is trained with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, an experience replay is introduced [26].

Algorithm 1 Deep Q-Learning with Experience Replay [26,25]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
  end for
end for

3.7 Chasing Rainbows

Double DQN Double DQN is an extension of DQN designed to address the overestimation problem present in the Q-learning algorithm and DQN. Using two separate networks, one for action estimation and one for evaluation, reduces overestimation and improves overall performance [15].

In a double DQN setting, given a minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory $\mathcal{D}$, the target is updated as follows [17]:

$$(R_{t+1} + \gamma_{t+1} q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2. \tag{1}$$

Prioritized Experience Replay (PER) A standard experience replay memory samples previous transitions uniformly. Prioritizing which transitions are replayed makes experience replay more efficient and effective. Transitions with a bigger temporal-difference (TD) error have a higher priority; stochastic prioritization and importance sampling alleviate the introduced bias and loss of diversity [32].

The probability p_t is the probability of sampling a transition relative to the the last encountered absolute *TD error*:

$$p_t \propto |R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t)|^\omega, \tag{2}$$

where ω is a hyperparameter that determines the shape of the distribution [17].

Dueling DQN A dueling DQN explicitly separates the representation of state values and action advantages in two separate functions. The two streams, representing the dueling architecture, share a common feature layer [40].

The new action values are computed as following:

$$q_\theta(s, a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}} \tag{3}$$

where ξ, η and ψ are, respectively, the parameters of the shared encoder f_ξ, of the value stream v_η, and of the advantage stream a_ψ; $\theta = \{\xi, \eta, \psi\}$ is their concatenation [17].

***n*-step learning** A multi-step variant of DQN is defined by minimizing the alternative loss [34]

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2, \quad \text{where } R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k=+1}. \tag{4}$$

Distributional DQN Instead of learning the expected return, the network can be trained to learn the distribution of returns [7]:

$$d'_t \equiv (R_{t+1} + \gamma_{t+1}\boldsymbol{z}, \boldsymbol{p}_{\bar{\theta}}(S_{t+1}, \bar{a}^*_{t+1})), D_{\mathrm{KL}}(\Phi_{\boldsymbol{z}} d'_t \| d_t), \tag{5}$$

where $\Phi_{\boldsymbol{z}}$ is a L2-projection of the target distribution onto the fixed support $\boldsymbol{z}$, and $\bar{a}^*_{t+1} = \arg\max_a q_{\bar{\theta}}(S_{t+1}, a)$ is the greedy action with respect to the mean action values $q_{\bar{\theta}}(S_{t+1}, a) = \boldsymbol{z}^\intercal \boldsymbol{p}_\theta(S_{t+1}, a)$ in state S_{t+1} [17].

NoisyNet NoisyNet enhances the DQN weights with parametric noise. The added stochasticity, over time, aids the network to ignore the noise over time, allowing the network to perform a more efficient exploration. The parameters of the noise are learned over time, and the added noise removes the need of the hyperparameter ϵ, that determines the ratio between exploration and exploitation [13].

The noisy stream is computed as following:

$$\boldsymbol{y} = (\boldsymbol{b} + \boldsymbol{W}x) + (\boldsymbol{b}_{\text{noisy}} \odot \epsilon^b + (\boldsymbol{W}_{\text{noisy}} \odot \epsilon^w)\boldsymbol{x}), \tag{6}$$

where ϵ^b and ϵ^w are random variables, and $\odot$ denotes the element-wise product [17].

Rainbow Rainbow is an integrated agent that combines approaches from **Eq.** 1 to **Eq.** 6 and adds them to **Alg.** 1. The result algorithm beats the previous SOTA performance on the Atari 2600 benchmark [17]. Some components of Rainbow, such as Double DQN and the Prioritized Experience Replay are reused and compared later on in the Method section.

3.8 Transformers

A Transformer model is composed of an encoding part connected to a decoding part. The encoding part is made of multiple encoder layers stacked on top of each other; each encoder processes its input through a self-attention layer, which enables it to consider other parts of the sequence as it encodes each input. The output of the self-attention layer becomes then the input of a feed-forward neural network [39]. The result of the encoding layers is added to the original sequence to create a more refined understanding of the input.

The decoding part shares the same structure of the encoding part, but each decoder adds, between the self-attention and the feed-forward layer, an extra attention layer that helps the decoder focus on the important parts of the input sequence [39].

The Transformer architecture has strong generalization capabilities, it is more efficient than sequential models since it is able to process sequential data in parallel, and it is easy to scale to large amounts of data.

Embedding In a Transformer architecture, each element is mapped to a unique embedding. The purpose of embedding is to provide the model with information about what each token of the input sequence represents. Through this process the model can find over time better and better representations for the input.

Since Transformers process sequential data in parallel, they do not have a representation of the order of the inputs in the sequence. Positional encoding adds information about where each element is in the sequence; some common approaches include adding a sinusoidal function to the input sequence or to learn the encoding from scratch during training.

The Attention Mechanism First introduced by [33], the attention mechanism allows the model to "pay attention" to the different parts of the input sequence in order to learn which parts are more relevant for the final prediction. For each element of the input sequence, the attention mechanism computes a score that indicates its importance for the current prediction. The score is computed by taking into account the context and other relevant information from the input sequence. Elements with higher scores contribute more to the final prediction.

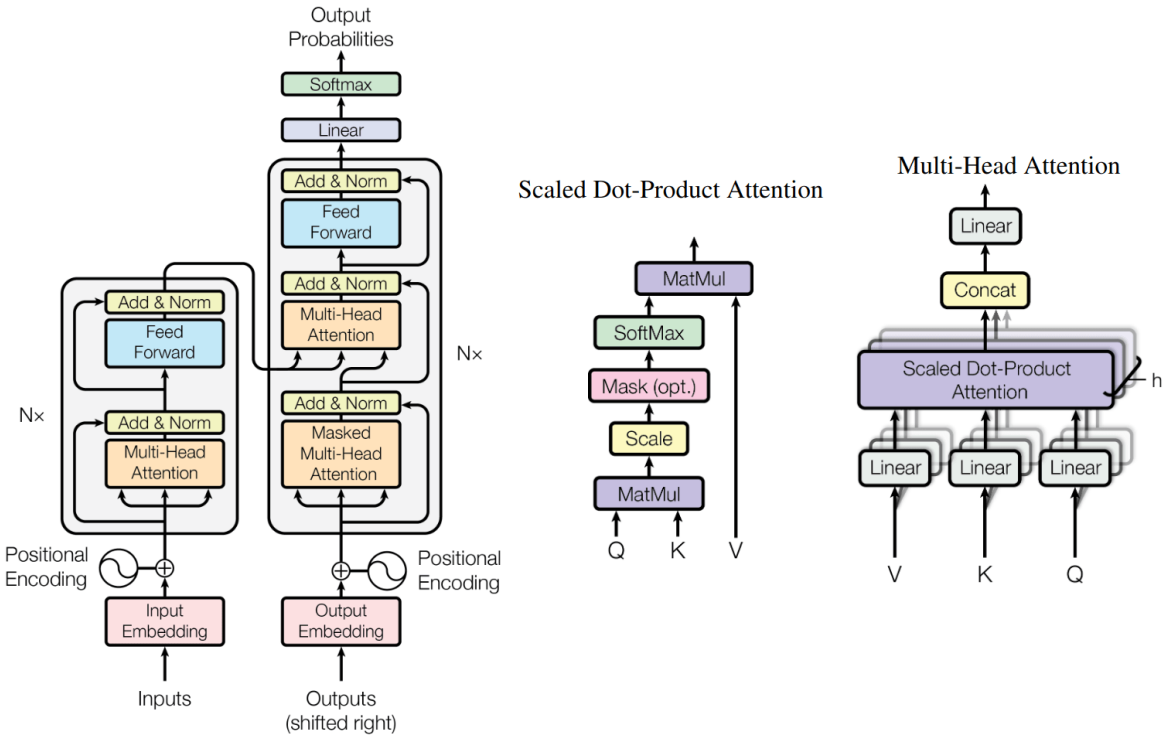

Fig. 2: From left to right: the Transformer architecture, the Scaled Dot-Product Attention and the Multi-Head Attention, consisting of several attention layers running in parallel [39].

4 Method

The intuition behind the TQN, shared by the related work on offline RL [12,11], is that a set of observations, on which the network is trained, can be seen as a sequence, for which the Transformer architecture is particularly suited.

By replacing the first layer of a DQN with a Transformer layer, the neural network can extract more relevant features, and the multi-head attention mechanism is trained to recognize the more relevant observations from the set, without the added complexity of a PER. However, this happens on a smaller scale, since PER samples the most important experiences from the entire experience replay, while the multi-head attention chooses the most relevant experiences from a smaller set of observations.

The expected result can become a viable way to increase stability during training and to improve performance, leading to a quicker convergence to the optimal policy.

4.1 Baseline

The input to the neural network consists of a tensor containing the state of the environment. The neural network processes the input through two hidden layers consisting of 256 units, both followed by a rectifier nonlinearity. The output layer is a linear layer whose values are the predicted Q-values for each individual action, given the input state [26,25]. Each layer of the neural network is a fully-connected layer (see **Fig.** 3. In order to reduce over-estimation, a copy of the network is used to evaluate the predictions (see Double DQN).

4.2 Transformer Q-Network

The Transformer Q-Network consists instead of an encoding component, a decoding component, followed by two hidden fully-connected layers and the output layer. Both the encoding and decoding components also stack two hidden layers each, all sharing the same parameters. Each encoder/decoder layer has an input and output size equal to the observation size, with a hidden layer of size 256. The dropout

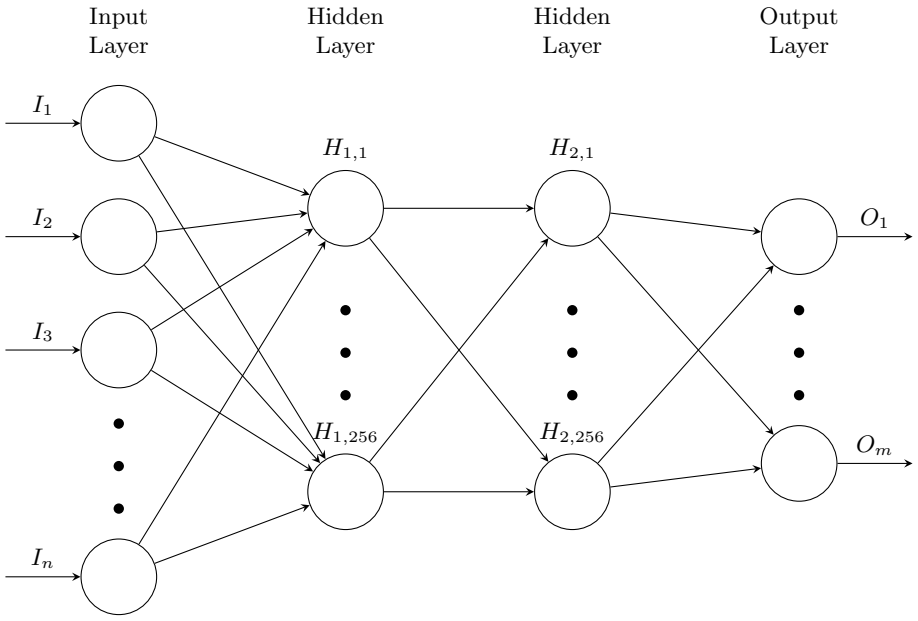

Fig. 3: The Deep Q-Network architecture.

value is set to 0.1 and the attention heads are set equal to the observation size, which can be 4, 6 or 8 depending on the environment.

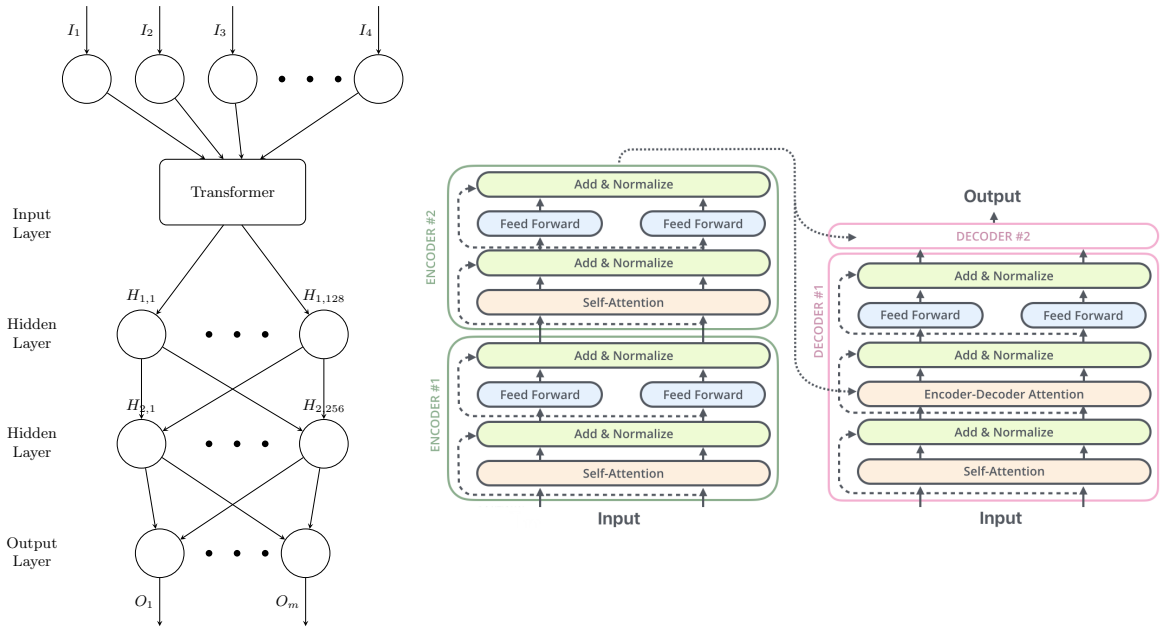

Fig. 4: On the left, a simplified view of the Transformer Q-Network. On the right, the inner components of the Transformer layer. Image on the right adapted from "The Illustrated Transformer"[4].

First, the batch of observations gets reshaped to a single batch, containing a sequence of observations, the previous batch size is now the length of the sequence. The encoder/decoder pair extracts the

[4] https://jalammar.github.io/illustrated-transformer/

relevant features from the observation, then the two fully-connected hidden layers of size 128 and 256, respectively, process it further. Each linear layer is always followed by a rectifier nonlinearity activation function. Finally, the output layer is again a linear layer whose values are the predicted Q-values for each possible action.

4.3 A Note on Embedding

Embedding is used with the attention model to transform a discrete token of the input sequence in a continuous vector representation. For the chosen environments, the observation is already in the form of a vector of floating point numbers, therefore preprocessing it through an embedding layer is not needed.

Depending on the task, before embedding a sequence, each token may be augmented with positional encoding. Again, this step was deemed not needed for two reasons: the sequence is not going through an embedding layer, and each observation does not temporally come after the previous one, but it is independently sampled from the experience replay. Preliminary iterations of the network that also employed an embedding step had the score over time collapse towards the worst possible value with no sign of recovery.

Algorithm 2 Baseline: Double DQN with Experience Replay [26,25,15]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_{j+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{j+1}, a; \theta); \theta_{\text{target}}) & \text{for non-terminal } s_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
  end for
end for

5 Experimental Setup

Both the baseline and the new TQN architecture are implemented from scratch in Python with PyTorch [28]. The flow of the code follows the previous work done by CleanRL [18], the logic is encapsulated in classes similar to "Rainbow Is All You Need" [27] and the hyperparameters share the same naming convention as Stable-Baselines 3 [31].

Despite being more complex, the TQN has fewer parameters than the DQN, $49k$ vs $68k$.

5.1 Environments

Although all three are considered toy environments, these environments are chosen with the following criteria in mind. First of all, all of them have a continuous observation space represented by a vector of `float`, and all of them expect, at each step, a single discrete action. Therefore, the neural network architecture does not need changes to adapt to the environments.

Interestingly, they are all different, each in their own way, because of the way their reward function is defined. *Acrobot* always returns a fixed negative reward, *CartPole* always returns a fixed positive reward, and *LunarLander* provides a *shaped reward* function that returns every time a different positive or negative value, depending on the state of the lander.

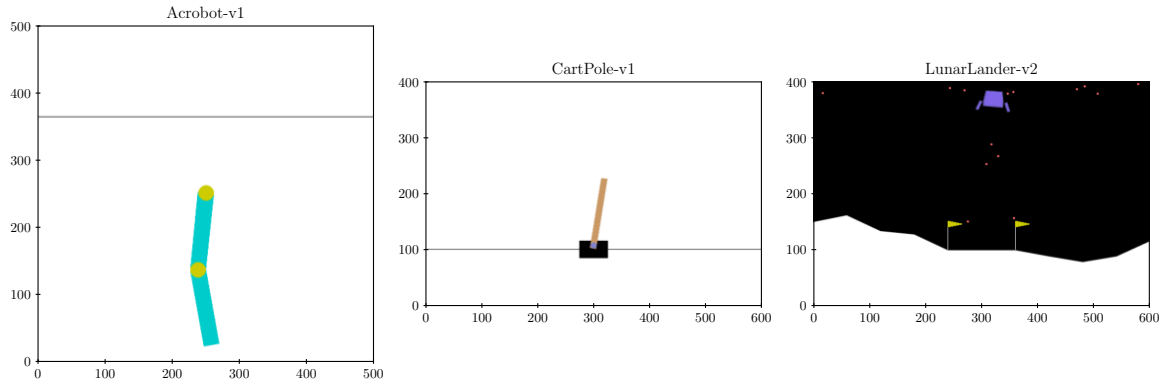

Fig. 5: Example screenshots for each environment. From left to right: Acrobot, CartPole, LunarLander.

Acrobot The Acrobot is a two-link under-actuated robot, an arm with the first joint fixed in place, and the joint between the two links actuated [35].

The observation contains the values of the sin and cos of the two joints, and their respective angular velocities. The three possible actions are applying positive torque, negative torque, and no torque to the actuated joint. The objective is to swing the other end above the bar by an amount equal to the length of one of the links. The reward is -1 for each step taken and an episode ends if the robot either reaches the bar or takes the maximum number of allowed steps. An episode is considered successful if it reaches a reward threshold of -100 [38].

A reward function that provides a fixed negative reward at each step, regardless of the state, is easy to design and pushes the agent to find its way to the target in the shortest possible amount of time. Conversely, the agent relies on the reward as a feedback on how good/bad the action taken is, and since the reward does not change, the agent may not have sufficient information to distinguish the good actions from the bad ones.

CartPole The CartPole is a cart to which a rigid pole is hinged on its center. The cart moves on a frictionless track and the pole starts from an upright position [5].

At each step, the available information about the environment is the cart position and linear velocity, and the pole angle and its angular velocity. The agent has no control on the pole, but it can push the cart left or right instead. Despite being discrete actions, the velocity is modeled depending on the angle of the pole. The reward is $+1$ for each step taken and the objective is to keep the pole upright for as long as possible. An episode terminates if the maximum score of 500 is reached, if the cart goes beyond the edges of the frame, or if the pole is too tilted [38].

Similar to *Acrobot*, the CartPole agent receives a constant reward at each step taken, but this time it is a positive $+1$. Again, the agent cannot rely on the reward to gauge the optimality of its actions, but instead the implicit target is to keep the episode running for as long as possible.

LunarLander The LunarLander environment is a rocket trajectory optimization problem. The lander has infinite fuel and it can either run the engines at full throttle or turn them off [9].

The observation provides information about the position of the lander in the space, the linear x and y velocities, its tilt, its angular velocity and whether or not each of the legs has touched the ground. The possible actions the lander can take are either be idle, or fire the engines (left, main or right respectively). The reward is modulated according to the distance from the landing pad, its speed and its tilt. An additional reward of ± 100 is assigned after either landing safely or crashing [38].

Since the reward function for the LunarLander provides a *shaped reward*, the agent is able to learn faster and make quick adjustments to its policy since it receives immediate feedback for each action taken. A shaped reward function better informs the agent on its closeness to the target but, at the same time, it is more difficult to design, and it may be exploited in unexpected ways. For example, instead of learning to land safely (in which case it would receive maximum reward), the LunarLander agent may learn to hover just above the landing pad because in that state it also receives a reward close to the maximum.

5.2 Loss and Parameter Optimization

To represent how well the predicted values align with the target values, I use the mean squared error (MSE), which is the average of the squared difference between two values:

$$\ell(x, y) = \bar{L}, \quad L = \{l_1, \dots, l_N\}^\intercal, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size, x and y are tensors of n elements each.

The Adam optimizer takes care of adapting the learning rate to each network parameter individually. The parameters with more significant gradients are updated according to a smaller effective learning rate, while parameters with smaller gradients get a larger effective learning rate. Furthermore, Adam optionally adds weight decay as a regularization technique that penalizes large weights during training [23], but in this case it is not used.

The optimizer updates the weights according to the following logic:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{b}_t} + \omega),$$

where θ_t is the updated weight, θ_{t-1} is the current weight, α is the learning rate, $\hat{m}_t$ is the biased first moment estimate, $\hat{b}_t$ is the biased second raw moment estimate, and ω is a small constant for numerical stability [23].

5.3 Algorithm Hyperparameters

Although different in the implementation from Stable-Baselines 3 [31], I choose to use the same hyperparameter names in most of the cases.

Optimal hyperparameters for DQN are taken from `rl-baselines3-zoo` [30]; the same set of hyperparameters is used for TQN. Every episode rollout is capped at a maximum of 500 steps.

Table 1: Hyperparameters used across all runs.

	Acrobot-v1	**CartPole-v1**	**LunarLander-v2**
max steps	100.000	50.000	100.000
gradient clip		10.0	
accumulate gradient batches		1	
batch size	128	64	128
learning rate	6.3e-4	2.3e-3	6.3e-4
buffer size	50.000	100.000	50.000
learning starts	0	1.000	0
tau		1.0	
gamma		0.99	
train frequency	4	256	4
gradient steps		1	
target update interval	250	10	250
exploration fraction	0.12	0.16	0.12
epsilon max		1.0	
epsilon min	0.1	0.04	0.1

- *max steps*: total number of environment steps to train on;
- *gradient clip*: maximum value for the gradient clipping of the network parameters;
- *accumulate gradient batches*: gradient steps to take before updating the policy;
- *batch size*: minibatch size for each policy update;
- *learning rate*: how much the policy weights should be updated w.r.t. the loss gradient;
- *tau*: soft update coefficient, if $\tau = 1$ a hard update is performed instead;
- *gamma*: discount factor;
- *train frequency*: steps to perform between each policy update;
- *target update interval*: environment steps to take between each target network update;

- *exploration fraction*: fraction of entire training period over which the exploration rate is reduced; e.g. if the max steps are 100.000 and the exploration fraction is 0.12, ϵ linearly decreases from $\epsilon_{\max}$ to $\epsilon_{\min}$ during the first 12.000 steps;
- *epsilon max*: exploration factor, initial probability of taking a random action;
- *epsilon min*: exploration factor, final probability of taking a random action.

5.4 Network Hyperparameters Tuning

For the TQN, I use a trial-and-error approach to find the best hyperparameters to make the network learn and converge. I only focus on tuning the parameters of the new network architecture; the default `nn.Transformer`[5] parameters (6 encoder layers, 6 decoder layers, and 2048 units for each hidden layer) provided by PyTorch are, computationally-wise, prohibitively expensive.

As a first step, I progressively decrease the number of layers until it is possible to train the network on a consumer GPU. On the GPU worker I used for training, the default Transformer parameters grind the iterations to a halt, the computation time explodes from mere minutes to hours, with sudden crashes of the system as a result of out-of-memory errors. I end up fixing the size of the encoder and decoder components to 2 layers each.

I then focus on the size of the hidden encoder/decoder layers. 2048 units are too much for such a simple input, and they destabilize the network. The model is too complex and it starts performing poorly on new observations. A very large hidden layer is also prone to the exploding or vanishing gradient problem. For this reason, I set the units of each hidden layer to 256. This proves to be sufficient for the network to converge.

Lastly, I investigate the impact of the number of heads in this multi-head attention model. Setting it to 1 brings the worst results during training; this is expected because a single attention head is not enough to capture the different relationships and the more complex dependencies within the data. Setting the number of attention heads equal to 2 already bring better results, but empirically the best results come by setting them equal to the size of the observation, 4, 6 and 8 depending on the environment.

6 Results and Discussion

For each run, I gather data such as loss, Q-values, episode reward and episode length; ϵ over time is also collected for debug purposes, but it is left out of the analysis because it is simply linearly annealed from $\epsilon_{\max}$ to $\epsilon_{\min}$ for each environment step for a total of *max steps* $\times$ *exploration fraction*, with no extra logic applied to it. Each model is trained on 20 different fixed seeds (see **Appendix C: Additional Experiments**). I then interpolate the results of each run so that I can compute the mean of each episode. In order to denoise the mean score per episode, I also compute a smoothed version of the obtained line, a rolling average with $n = \frac{\text{number of episodes}}{10}$.

Despite being the most simple environment, none of the networks are able to learn to balance the pole in the *CartPole* environment, and the TQN fails to find a successful policy for *Acrobot*. Supplementary analysis for these environments is relegated to the **Appendix C: Additional Experiments**.

Table 2: Performance of the models tested across all the environments.

	Acrobot-v1		CartPole-v1		LunarLander-v2	
	R_{mean}	R_{best}	R_{mean}	R_{best}	R_{mean}	R_{best}
DQN	**-91.99**	**-61**	**77.54**	281.00	**151.88**	317.45
TQN	-490.26	-72	38.03	**500**	46.3	**322.74**

For the analysis, I take as a reference the *LunarLander* environment, since it is the only one where both architectures converge. For the LunarLander environment, an episode is considered a valid solution if it scores 200 points or more.

[5] https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

For the most relevant metrics I use **bootstrapping with stratified sampling** to compute the 95% **confidence intervals**; the bootstrapping uses 50.000 replications [6]; the sample efficiency curve normalizes the scores between 0 and the target score $\tau = 200$.

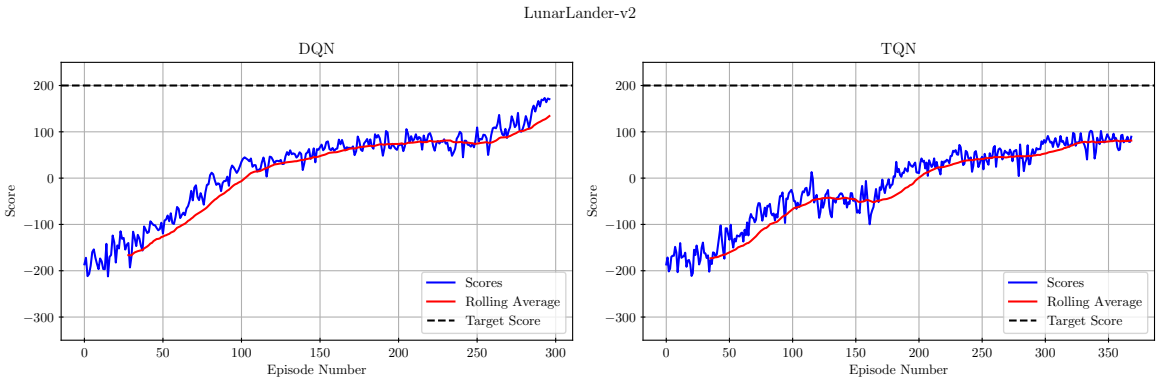

Fig. 6: Each run has the same number of environment steps but, since the episode length is not fixed, the total number of episodes may differ.

In **Fig.** 6, both sample efficiency curves have the shape of a logarithmic curve showing peak efficiency towards the end of the training and approaching the target score of 200, but TQN performs worse and has a higher variance. The new architecture shows convergence to the optimal policy over time, but from the plot it may seem that it is less efficient than the baseline. In reality, when I consider the the confidence intervals of the **interquartile mean (IQM)**, the performance on the middle 50% of the combined runs computed by removing the top and bottom 25% of the runs [1], the probability of improvement shows that the TQN performs better than the baseline (see **Fig.** 7). The **probability of improvement** is a metric that shows how likely it is for X to outperform Y on a randomly selected task, where

$$P(X > Y) = \frac{1}{M} \sum_{m=1}^{M} P(X_m > Y_m), \tag{7}$$

with $P(X_m > Y_m)$ the probability of X being better than Y at task m [1].

In this case, $P(DQN > TQN) = 0.48$, which is equivalent to say $P(TQN > DQN) = 0.52$, meaning that, given a random task, the Transformer Q-Network gets a better reward than the baseline 52% of the time.

To check whether each model has over-confident Q-values, I average the results of 400 test runs for each architecture (20 models, 1 for each seed, play environments tested with all the seeds). For a consistent range of comparison, both the rewards and Q-values gathered this way are normalized between -1 and 1.

Despite the Q-values represent the overall confidence of the network while the step rewards are only an immediate feedback, it is still useful to visualize their relationship to better understand the behavior of the two architectures. As the rewards increase, the Q-values should also increase. Ideally, in **Fig.** 8, the density should be aligned with the major diagonal. This would mean that Q-values estimates are well-aligned with the rewards each agent receives.

I can observe that, while both models present fluctuation, DQN is more robust and it takes a more conservative approach; it is the most "pessimistic" when it receives low rewards. The TQN architecture instead is over-confident in its estimates and, although the model is most confident when it receives the highest rewards, as expected, it also shows a high degree of confidence when it receives the lowest rewards.

[6] The process of repeatedly sampling with replacement from the underlying data in order to create a simulated dataset.

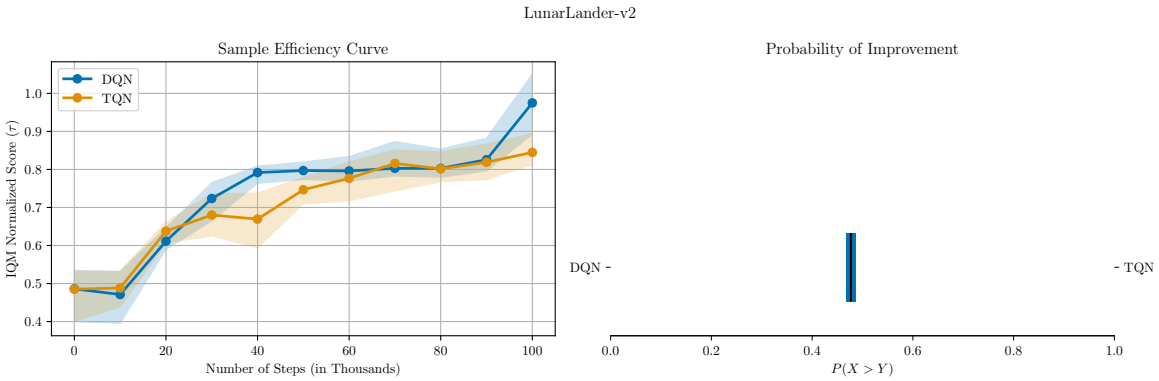

Fig. 7: On the left, sample efficiency of agents as a function of number of steps measured via IQM scores. Shaded regions show pointwise 95% percentile stratified bootstrap CIs, as suggested by [1]. On the right, the probability of improvement.

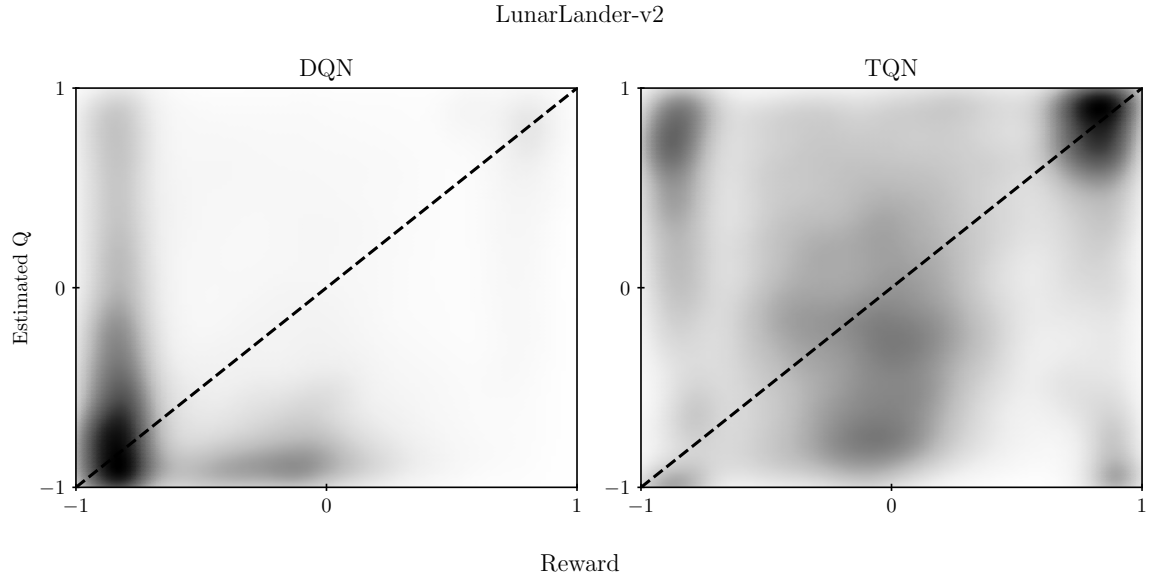

Fig. 8: Density plot of the estimated Q-values versus environment step rewards sampled from 400 episodes.

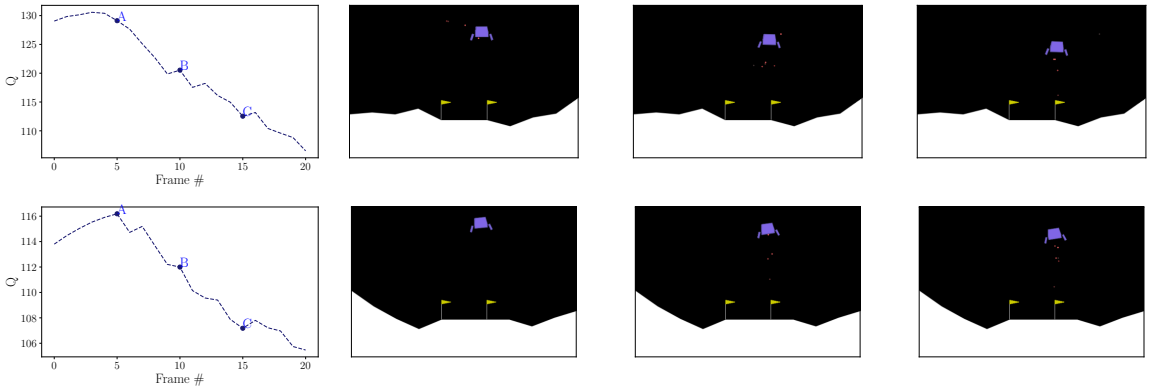

Fig. 9: On the left, the predicted Q-values for a portion of a LunarLander episode. The three screenshots on the right correspond to the frames labeled by A, B, and C. The first row is from the Deep Q-Network and the second one from the Transformer Q-Network.

Despite using a target network to dampen its confidence, TQN is still prone to over-estimating Q-values. The over-confidence in the predictions matches what I show in **Fig.** 8; the step of the loss function indicates that TQN has learned steadily during training, but it has not fully converged yet.

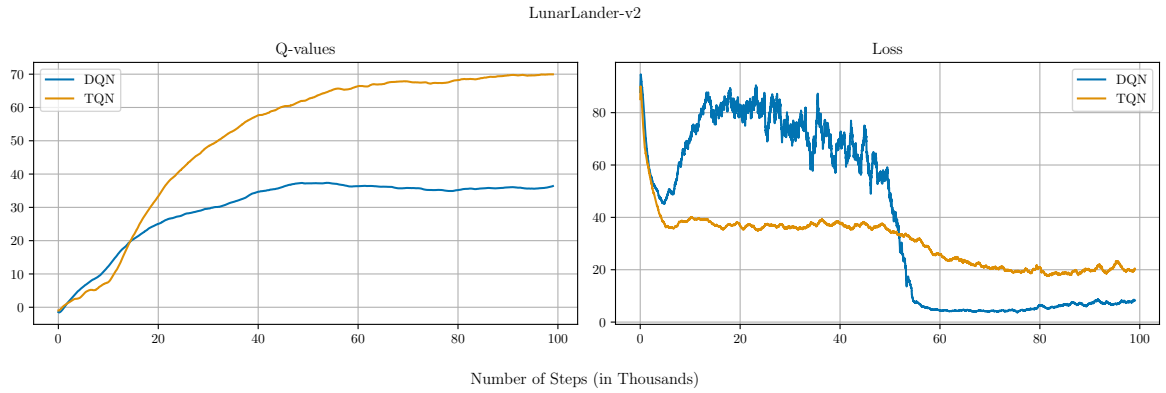

Fig. 10: Q-values over time and loss over time. TQN is over-confident and converges more slowly.

The last metrics (Fig. 11) are again computed by normalizing the scores between 0 and $\tau = 200$, the target score for which an episode of the LunarLander environment is considered successful (see LunarLander). Albeit the magnitude is very small (10^{-2}), the Transformer Q-Network is able to beat the baseline DQN baseline. Large confidence intervals for the **median** mean that there are a few high performing tasks skewing the results and, to compensate for it, I choose the IQM, explained above, as a better indicator of the performance [1]. The IQM is more robust than the mean to statistical outliers and, thanks to the smaller confidence intervals, it is able to detect improvements with fewer runs [1]. As opposed to the probability of improvement, both the IQM and the optimality gap that follows take into account the size of improvement.

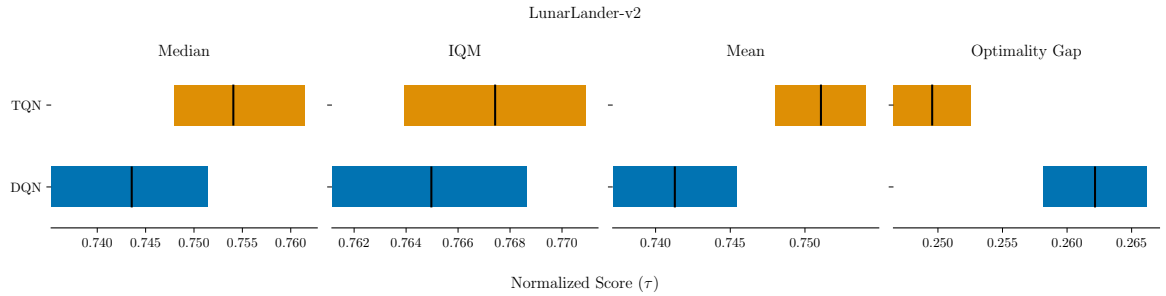

Fig. 11: Aggregate metrics with 95% confidence intervals.

Finally, a more robust alternative to the mean is the **optimality gap**, which is the amount by which the models fail to meet the target score τ. The optimality gap assumes that scores above the threshold are less important than the rest. In this case a higher value corresponds to worse performance, and the results in **Fig.** 11 once again confirm that the novel architecture performs indeed better than the baseline.

7 Conclusion

I present an online RL agent that can effectively learn from the environment by processing in parallel a sequence of transitions. I add a Transformer layer as a feature layer to a baseline double DQN implementation and I show that the architecture is able to exploit information from the environment and to converge over time to the optimal policy.

To evaluate the TQN against the baseline, I follow the approach first proposed by [1]. Despite reusing the same set of hyperparameters with a different architecture, the new architecture is able to learn and beat the baseline on 1 out of the 3 environments.

As a first step for extra future work, performing a hyperparameter sweep is necessary. Hyperparameters that are optimal for DQN may not be optimal for TQN and vice versa; new ones are also introduced, such as the number of encoder and decoder layers, their size and the number of self-attention heads. It would also be interesting to study how the new architecture can be applied to physics environments that operate over continuous action spaces. In this case, adding a Transformer layer to a Deep Deterministic Policy Gradient (DDPG) may be trivial.

8 Acknowledgements

First of all, my thanks go to professor Vincent François-Lavet for his feedback and inputs during the trajectory of this paper and for having given me the opportunity to have a glimpse of what RL research looks like; I look forward to contribute much, much more to the field. Then, in no particular order, I want to thank my whole family, my girlfriend Salomé for her unwavering love and support, and for her invaluable proof-reading skills, our cat Giulietta, who actually wrote all the code for this paper during feverish pair-programming sessions, my mother Maddalena, my father Girolamo and my sister Deborah for always being there.

References

1. Agarwal, R., Schwarzer, M., Castro, P.S., Courville, A.C., Bellemare, M.: Deep reinforcement learning at the edge of the statistical precipice. Advances in Neural Information Processing Systems **34** (2021) 1, 13, 14, 15, 16
2. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in ai safety (2016) 1
3. Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z.D., Blundell, C.: Agent57: Outperforming the atari human benchmark. In: III, H.D., Singh, A. (eds.) Proceedings of the 37th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 119, pp. 507–517. PMLR (13–18 Jul 2020), https://proceedings.mlr.press/v119/badia20a.html 2
4. Badia, A.P., Sprechmann, P., Vitvitskyi, A., Guo, D., Piot, B., Kapturowski, S., Tieleman, O., Arjovsky, M., Pritzel, A., Bolt, A., Blundell, C.: Never give up: Learning directed exploration strategies (2020) 2
5. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics **SMC-13**(5), 834–846 (1983). https://doi.org/10.1109/TSMC.1983.6313077 10
6. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research **47**, 253–279 (jun 2013). https://doi.org/10.1613/jair.3912, https://doi.org/10.1613/jair.3912 1
7. Bellemare, M.G., Dabney, W., Munos, R.: A distributional perspective on reinforcement learning. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. p. 449–458. ICML'17, JMLR.org (2017) 6
8. Bellman, R.: A markovian decision process. Journal of Mathematics and Mechanics **6**(5), 679–684 (1957), http://www.jstor.org/stable/24900506 3
9. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016) 10
10. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf 1
11. Chebotar, Y., Vuong, Q., Irpan, A., Hausman, K., Xia, F., Lu, Y., Kumar, A., Yu, T., Herzog, A., Pertsch, K., Gopalakrishnan, K., Ibarz, J., Nachum, O., Sontakke, S., Salazar, G., Tran, H.T., Peralta, J., Tan, C., Manjunath, D., Singht, J., Zitkovich, B., Jackson, T., Rao, K., Finn, C., Levine, S.: Q-transformer: Scalable offline reinforcement learning via autoregressive q-functions. In: 7th Annual Conference on Robot Learning (2023) 2, 7

12. Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., Mordatch, I.: Decision transformer: Reinforcement learning via sequence modeling. In: Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems. vol. 34, pp. 15084–15097. Curran Associates, Inc. (2021), https://proceedings.neurips.cc/paper_files/paper/2021/file/7f489f642a0ddb10272b5c31057f0663-Paper.pdf 2, 3, 7
13. Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., Legg, S.: Noisy networks for exploration (2019) 6
14. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G., Pineau, J.: An introduction to deep reinforcement learning (2018) 1, 3
15. van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. Proceedings of the AAAI Conference on Artificial Intelligence **30**(1) (Mar 2016). https://doi.org/10.1609/aaai.v30i1.10295, https://ojs.aaai.org/index.php/AAAI/article/view/10295 5, 9
16. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep reinforcement learning that matters. Proceedings of the AAAI Conference on Artificial Intelligence **32**(1) (Apr 2018). https://doi.org/10.1609/aaai.v32i1.11694, https://ojs.aaai.org/index.php/AAAI/article/view/11694 1
17. Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining improvements in deep reinforcement learning. Proceedings of the AAAI Conference on Artificial Intelligence **32**(1) (Apr 2018). https://doi.org/10.1609/aaai.v32i1.11796, https://ojs.aaai.org/index.php/AAAI/article/view/11796 1, 3, 5, 6
18. Huang, S., Dossa, R.F.J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., Araújo, J.G.: Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. Journal of Machine Learning Research **23**(274), 1–18 (2022), http://jmlr.org/papers/v23/21-1342.html 9
19. Irpan, A.: Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html (2018) 1
20. Janner, M., Li, Q., Levine, S.: Offline reinforcement learning as one big sequence modeling problem. In: Advances in Neural Information Processing Systems (2021) 2
21. Kapturowski, S., Campos, V., Jiang, R., Rakićević, N., van Hasselt, H., Blundell, C., Badia, A.P.: Human-level atari 200x faster (2022) 2
22. Kapturowski, S., Ostrovski, G., Dabney, W., Quan, J., Munos, R.: Recurrent experience replay in distributed reinforcement learning. In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=r1lyTjAqYX 2
23. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization (2017) 11
24. Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). pp. 10012–10022 (October 2021) 1
25. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning (2013) 1, 4, 5, 7, 9
26. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (Feb 2015). https://doi.org/10.1038/nature14236, https://doi.org/10.1038/nature14236 1, 3, 4, 5, 7, 9
27. Park, C.: Rainbow is all you need. https://github.com/Curt-Park/rainbow-is-all-you-need (2019) 9
28. 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: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., Garnett, R. (eds.) 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 9
29. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019) 1
30. Raffin, A.: Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo (2020) 11, 21
31. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research **22**(268), 1–8 (2021), http://jmlr.org/papers/v22/20-1364.html 9, 11
32. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay (2016) 5
33. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 27. Curran Associates, Inc. (2014), https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf 6

34. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine Learning **3**(1), 9–44 (Aug 1988). https://doi.org/10.1007/BF00115009, https://doi.org/10.1007/BF00115009 5
35. Sutton, R.S.: Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: Touretzky, D., Mozer, M., Hasselmo, M. (eds.) Advances in Neural Information Processing Systems. vol. 8. MIT Press (1995), https://proceedings.neurips.cc/paper_files/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf 10
36. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA (2018) 1, 3
37. Thompson, W.R.: On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika **25**(3-4), 285–294 (1933) 4
38. Towers, M., Terry, J.K., Kwiatkowski, A., Balis, J.U., Cola, G.d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Shen, A.T.J., Younis, O.G.: Gymnasium (mar 2023). https://doi.org/10.5281/zenodo.8127026, https://zenodo.org/record/8127025 10
39. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf 6, 7
40. Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., De Freitas, N.: Dueling network architectures for deep reinforcement learning. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. p. 1995–2003. ICML'16, JMLR.org (2016) 5
41. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning **8**(3), 279–292 (May 1992). https://doi.org/10.1007/BF00992698, https://doi.org/10.1007/BF00992698 4

Appendix A: Reproducibility

Source Code

Source code, trained models, logs, plots and other results are available at https://github.com/robertoschiavone/transformer-q-network.

Seeds

Every data point is collected using 20 different seeds, from 1693526400 to 1695168000, with increments of 86400 between them. Each seeds represents the Unix time from September 1, 2023 00:00:00 GMT to September 20, 2023 00:00:00 GMT. The seeds are chosen for no specific reason other than ensuring the reproducibility of the results. The final list of seeds is

```
seeds = [ 1693526400, 1693612800, 1693699200, 1693785600, 1693872000,
          1693958400, 1694044800, 1694131200, 1694217600, 1694304000,
          1694390400, 1694476800, 1694563200, 1694649600, 1694736000,
          1694822400, 1694908800, 1694995200, 1695081600, 1695168000 ].
```

Appendix B: Self Reflection

I started the project with a high degree of confidence about my knowledge of Reinforcement Learning, and the original starting point was the grandiose idea of training a single agent (not even a single architecture!) capable of playing all Atari games. I have learned that good research takes time, and it is not feasible to test all variants of plausible solutions on every environment ever published. Time acts as a strainer that filters out the irrelevant details and lets you focus on the important parts of the work.

During my research, I have come to realize that yes, this paper will show how much I know about AI and RL, but also that my knowledge is extremely narrow and limited, and **this is fine**. The field of RL is vast and it is thriving thanks to a fresh stream of interesting concepts and ideas, and I would need more than a lifetime to master and keep up with everything that has been published so far. I have found my niche though, with its engaging set of problems, growing at each discovered solution, puzzles whose original pictures are missing and all their pieces are scrambled.

Appendix C: Additional Experiments

Acrobot

While the baseline works as expected, TQN is not able to converge. Despite the score starts to weakly increase, the loss is still growing over time. All metrics further prove that the Transformer Q-Network is far from reaching an optimal policy.

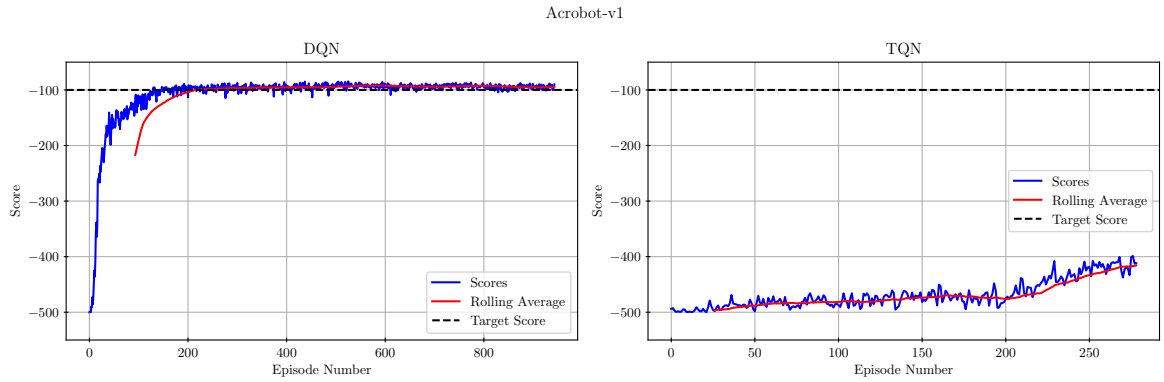

Fig. 12: Score trend over time during training for Acrobot.

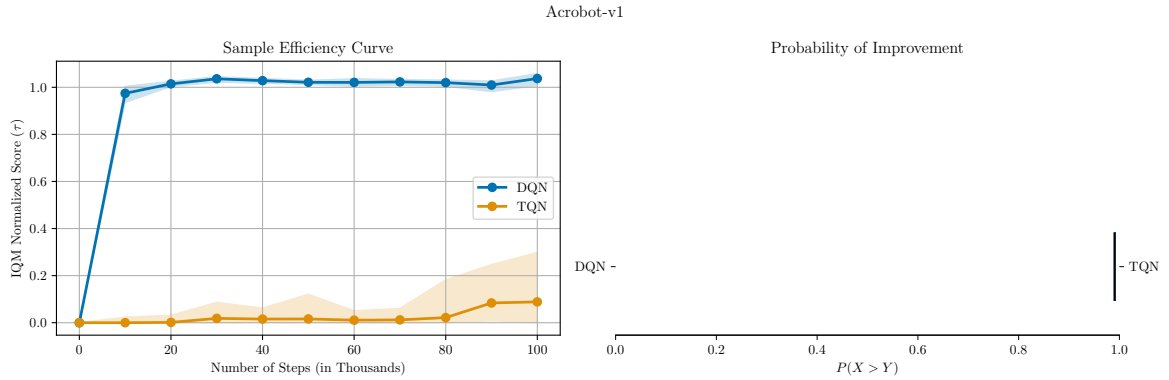

Fig. 13: Sample efficiency and probability of improvement for Acrobot.

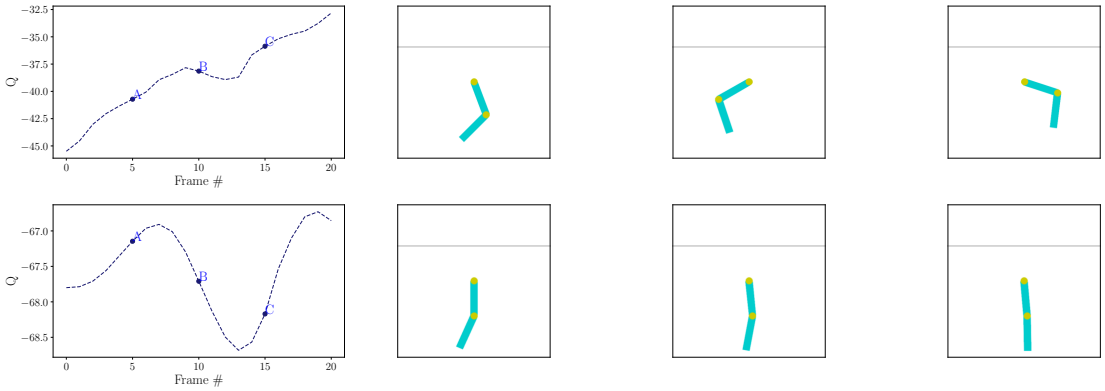

Fig. 14: The predicted Q-values for a portion of a Acrobot episode and the three screenshots corresponding to frames A, B, and C. From top to bottom: DQN and TQN.

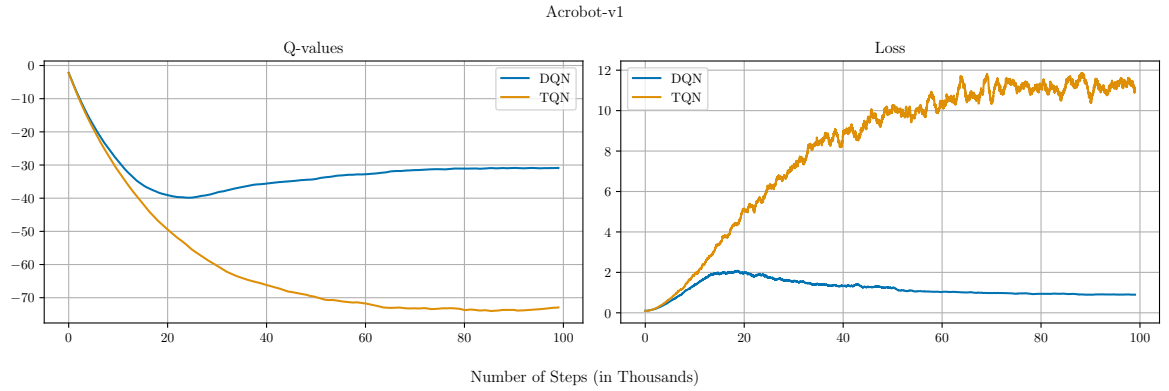

Fig. 15: Q-values and loss over time for Acrobot.

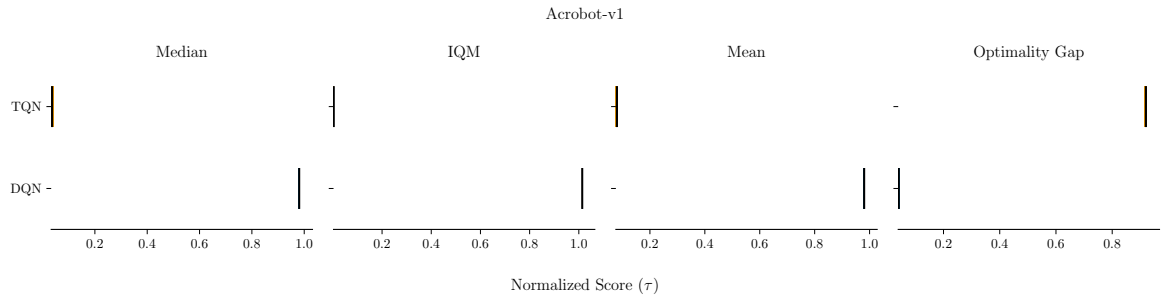

Fig. 16: Median, IQM, mean and optimality gap for Acrobot.

CartPole

CartPole is widely regarded as one of the simplest environments, and yet no architecture was able to learn in the given amount of steps. There may be some differences with the implementation from which I borrow the hyperparameters [30] that do not allow the agent to learn, or the hyperparameters are not really optimal and need further tuning. Time was limited though, and I was not able to further investigate the problem.

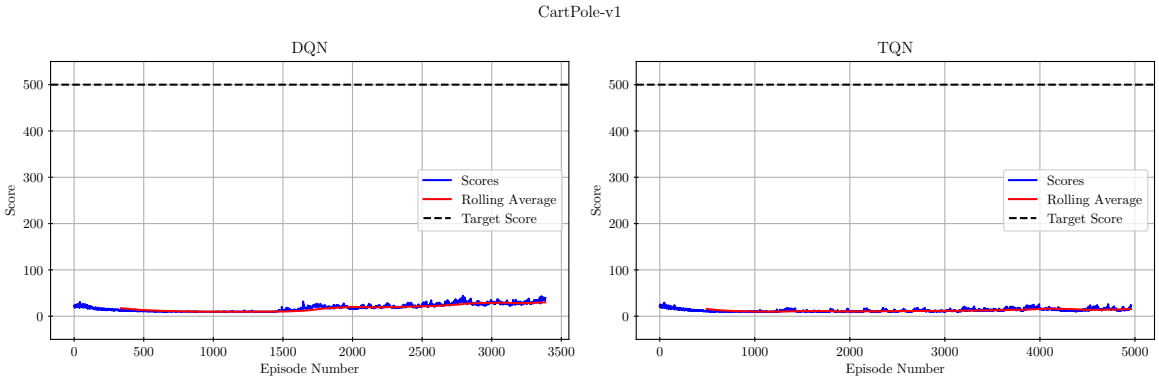

Fig. 17: Score trend over time during training for CartPole.

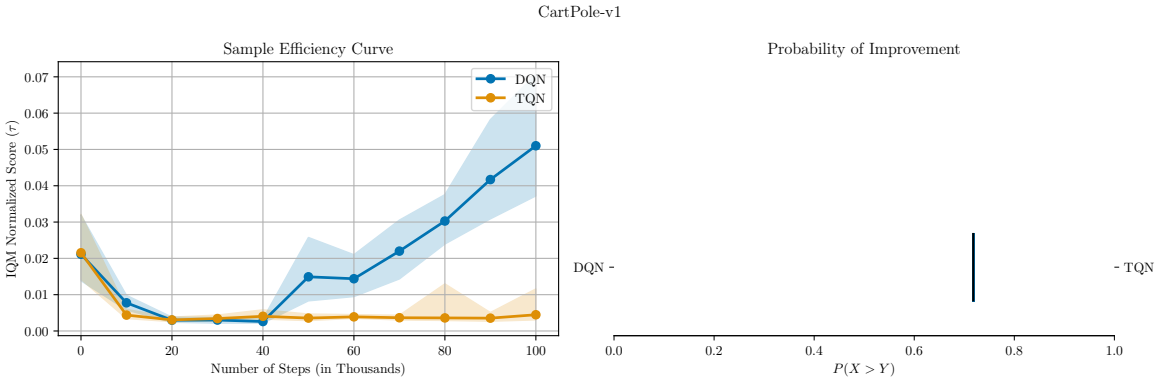

Fig. 18: Sample efficiency and probability of improvement for CartPole.

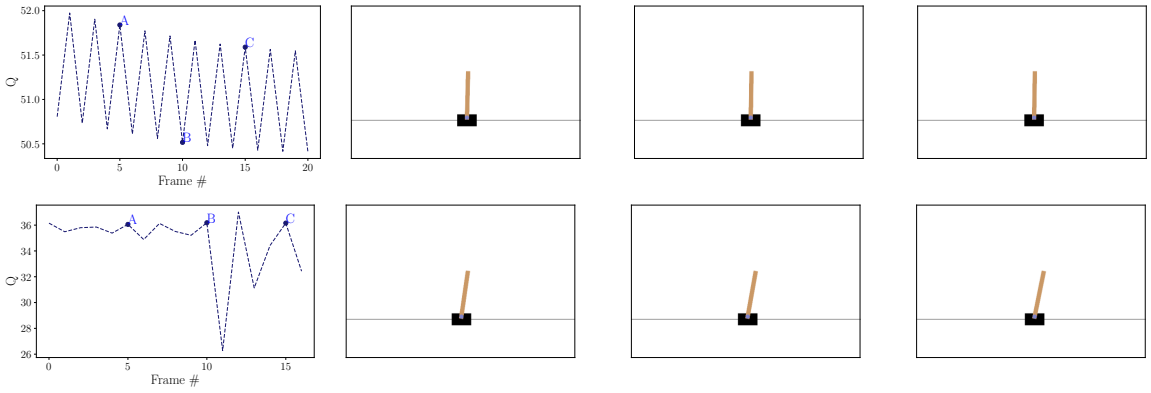

Fig. 19: The predicted Q-values for a portion of a CartPole episode and the three screenshots corresponding to frames A, B, and C. From top to bottom: DQN and TQN.

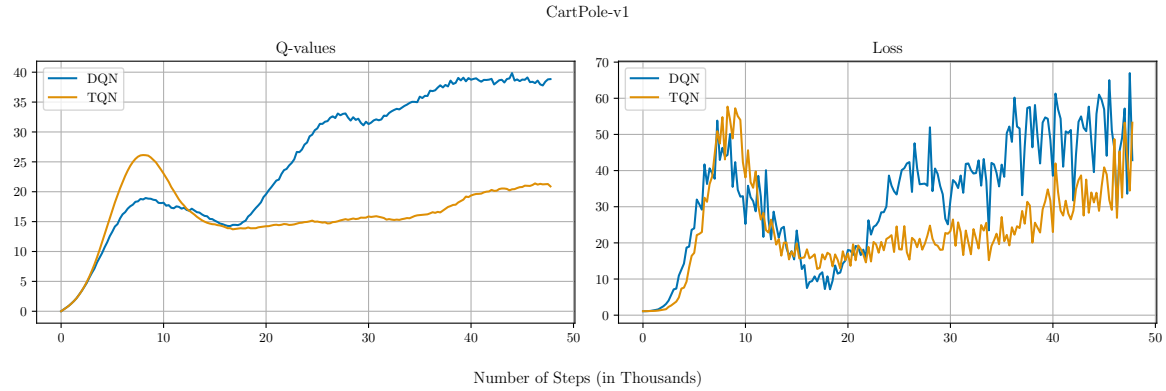

Fig. 20: Q-values and loss over time for CartPole.

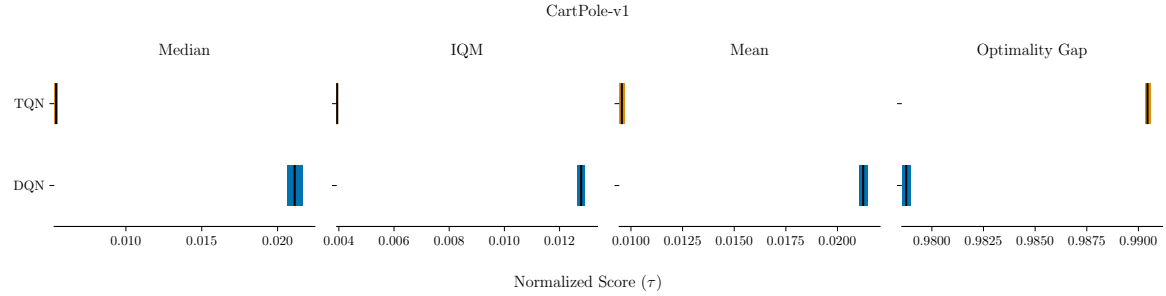

Fig. 21: Median, IQM, mean and optimality gap for CartPole.

Acronyms

AI Artificial Intelligence. 1, 19

DQN Deep Q-Network. 1, 2, 5–9, 11, 13–16, 20, 22

IQM interquartile mean. 13–15, 20, 22

LSTM Long Short-Term Memory. 2

MDP Markov decision process. 3
ML Machine Learning. 1, 3
MSE mean squared error. 11

NGU NeverGiveUp. 2

PER Prioritized Experience Replay. 5–7

R2D2 Recurrent Replay Distributed DQN. 2
RL Reinforcement Learning. 1–4, 7, 15, 16, 19

SOTA state-of-the-art. 2, 6

TD temporal-difference. 4, 5
TQN Transformer Q-Network. 1, 7–9, 11–16, 19, 20, 22

UCB Upper-Confidence-Bounds. 4