Deep Reinforcement Learning in Atari 2600 Games

Bachelor's Project Thesis

Daniel Bick, daniel.bick@live.de,
Jannik Lehmkuhl, j.lehmkuhl@student.rug.nl,
Supervisor: Dr M. A. Wiering

Abstract: Recent research in the domain of Reinforcement Learning (RL) has often focused on the popular deep RL algorithm Deep Q-learning (DQN). A different deep RL algorithm, similar to DQN, called Deep Quality-Value Learning (DQV), has received much less attention, albeit its great potential in outperforming DQN due to DQV's theoretical advantage of having an increased learning speed due to only approximating a state-value mapping instead of a state-action-value mapping as a target network. This thesis focuses on comparing these two aforementioned deep RL algorithms on their performances in learning to play different Atari 2600 games provided by the OpenAI gym. The impact of different exploration strategies on the learning performance of DQN and DQV will be examined, more specifically, a diversity-driven approach (Div-DQN and Div-DQV) and a noisy network approach (NoisyNet-DQN and NoisyNet-DQV) will be compared to traditional implementations of DQN and DQV. The results show that the standard DQV algorithm outperforms DQN and that DQV-based variants in general slightly outperform DQN-based variants. The NoisyNet approach shows the overall best training outcome, followed by DQV, the diversity-driven approach, and DQN.

1 Introduction

Reinforcement Learning (RL) is a Machine Learning approach in which an agent is trained by receiving either reward or punishment while interacting with the environment (Russell & Norvig, 2016). Due to the generality of this approach, it can be applied to a variety of real-life applications, including the control of aerial and autonomous vehicles, robotic arms, and humanoid robots (Kober et al., 2013). Other approaches, not dealing with physically embodied robots, use RL for tasks as diverse as stock market predictions (Lee, 2001) or to explore effective strategies for peer-to-peer teaching in a cooperative multiagent RL setting (Kim et al., 2019). Common practice often used in assessing and comparing the learning ability of different RL algorithms involves comparing these algorithms based on their performance in learning to play video games from raw pixel data, a termination signal, and a corresponding reward signal (Mnih et al., 2013). This is because video games offer a complex and challenging environment to be mastered even for human players (Mnih et al., 2015), while also maintaining an intuitive and easily comparable performance measure in the form of scores. Following this popular approach of training different RL algorithms on playing video games and assessing the quality of the algorithms' respective learning outcomes by comparing the evaluation scores obtained by the trained agents, in this thesis the two deep RL algorithms *Deep Q-learning* (Mnih et al., 2013) and *Deep Quality-Value Learning* (Sabatelli et al., 2018) will be contrasted with each other alone and in combination with the two exploration strategies *Div-DQN* (Hong et al., 2018) and *NoisyNet* (Fortunato et al., 2017) on their performances in learning to play four Atari 2600 games.

1.1 Background

The field of Reinforcement Learning (RL) is concerned with algorithms that learn how to optimally behave in a given environment by learning an optimal policy from a reward signal observed while interacting with the environment, where the optimal policy dictates the algorithm, or *agent*, which action to choose in a given state in order to maximize the agent's expected total reward (Russell & Norvig, 2016). Furthermore, an RL agent is not assumed to have any prior knowledge about the reward signal or environment it has to learn to behave in (Russell & Norvig, 2016).

This approach assumes a *Markov Decision Process* (MDP) underlying the environment the agent is situated in (Sutton & Barto, 2017). That is, the agent faces a sequential decision problem with accumulative rewards in a fully observable environment, in which actions have a stochastic outcome and where the transition model is Markovian. Markovian refers to the environment's property that, for each state s, the probability of transitioning into successor state *s'* depends on no state except s (Russell & Norvig, 2016).

Thus, by repeatedly observing rewards obtained by taking different actions in different states and progressively adjusting its policy, an agent gradually learns which actions it must take in each state in order to maximize its expected total reward (Sutton & Barto, 2017). The outcome of this process is called the optimal policy (Russell & Norvig, 2016).

Initially, RL often relied on tabular representations of the optimal policy (Watkins et al., 1992; Sutton & Barto, 2017). As Busoniu et al. (2017) point out, however, this approach is only tractable for problems with a limited state-action space since otherwise an employed table may grow too large.

In response to this limitation, the deep RL algorithm *Deep Q-learning* has been introduced (Mnih et al., 2013), which is an extension of Q-learning, where an associated agent implementing Deep Q-learning is called the *Deep Q-Network* (DQN).

In Q-learning, being one instance of RL, an agent learns the expected value, or *utility*, of taking a particular action in a particular state for all actions in each state (Russell & Norvig, 2016). This yields the so-called action-utility function, also known as *Q-function* (Russell & Norvig, 2016), where the learned utility of taking action a in state s is the sum of both the observed immediate reward for taking a in s plus the discounted maximal expected utility from executing the best possible action in the next state (Sutton & Barto, 2017).

When exploiting a learned policy, following the Bellman equation, in each state a Q-learning agent selects the action that yields the maximal expected utility given the current state (Mnih et al., 2013).

Mnih et al. (2013) extended this approach by substituting the Q-function of classical Q-learning with a Convolutional Neural Network (CNN) architecture, and hence a universal function approximator (Hill, 1994), being able to generalize the optimal policy to be learned from seen states to yet unseen but similar states (Mnih et al., 2015; Sutton & Barto, 2017). In this approach, instead of precisely learning the Q-function, the agent learns to approximate the Q-function. Also, by using a CNN, a deep RL agent becomes capable of working on raw imagery state representations for tasks to be learned instead of relying on a more compact state representation, possibly including the usage of hand-crafted features, as was often the case in the past (Mnih et al., 2013).

DQN makes use of the following techniques. The weights of the CNN are trained using stochastic gradient descent on minibatches of experiences. An experience is an observed state transition, containing the state from which the agent transitioned, the action chosen by the agent in the state, the reward observed in response to executing the action in the state, the information whether the state is a terminal state, and, if the state is not a terminal state, the consecutive state. As will be explained in section 3, *experience replay*, the *ϵ-greedy* exploration strategy, and the *target network* approach are used for sampling of minibatches, action selection, and computing future utilities, respectively (Mnih et al., 2013).

Since the introduction of DQN, many researchers have proposed modifications to the standard DQN algorithm aiming at improving various aspects about it of which an overview and a corresponding evaluation are presented in Mnih et al. (2018). Two proposed modifications are as follows.

One diversity-driven exploration strategy for DQN, called *Div-DQN*, has been proposed by Hong et al. (2018). In this approach, the standard loss function of a deep RL agent gets replaced by a variant that rewards the discovery of novel, insufficiently explored states. As training progresses, rewards for novel discoveries decrease in magnitude to promote convergence to the optimal policy.

In a second exploration strategy, called *NoisyNet-DQN* (Fortunato et al., 2017), the standard linear layers of DQN get replaced by modified linear layers being disturbed by parametrized noise. In this approach, the value of taking a certain action in a given state depends on both sets of parameters learning the agent's policy and sets of parameters being altered by Gaussian noise to an extent learned by the agent.

The importance of good exploration strategies in RL for achieving better training outcomes has been highlighted by Kaelbling et al. (1996).

A distinct deep RL algorithm, similar to DQN, is *Deep Quality-Value Learning* (DQV) proposed by Sabatelli et al. (2018), which is a modification of the QV(λ) Learning algorithm proposed by Wiering (2005). This approach differs from DQN in that during training both a Q-function and a *V-function* get approximated in parallel and the periodically updated CNN constituting the target network in DQV is based on the approximation of the V-function instead of the Q-function, as is the case in DQN. The V-function directly assigns an utility to a state, where a state's utility is the sum of the reward the agent receives for being in the current state and the expected discounted cumulative reward the agent will obtain when continuing to follow its policy from the next state onward (Sabatelli et al., 2018). The intuition motivating this difference is the idea that the V-function might converge faster onto a single utility per state than the Q-function would converge onto a set of utilities per action per state, possibly accelerating training (Sabatelli et al., 2018).

1.2 Contributions

Although the effects of introducing many modifications to DQN on its performance in learning optimal policies have been studied extensively already, little research has focused on the comparison of DQN and DQV. Also, the effects of introducing many of the extensions designed for DQN to DQV on DQV's learning performance are still understudied. Therefore, this thesis aims at comparing the two algorithms DQN and DQV based on their performance in learning to play the four Atari 2600 games *Breakout*, *Q*Bert*, *Centipede*, and *Enduro* from the open-source OpenAI gym. The thesis compares the two algorithms in their original forms and in combination with both the variations Div-DQN and NoisyNet, as mentioned earlier, separately, yielding 6 variants.

For the evaluation, the algorithmic variants will be ranked per game based on the test scores they obtain playing the learned games. The final comparison of the variants will be based on the average rank each algorithmic variant obtains across the four games.

2 The Games

The training and testing environments used in this research are the four *Atari 2600* games *Breakout*, *Q*Bert*, *Centipede*, and *Enduro* simulated through the *Arcade Learning Environment* (Bellemare et al., 2012) and provided by the open-source *OpenAI gym*, which provides a standardized interface across the four game environments.

Each game environment has a frame rate of 60 Hz (Mnih et al., 2013) and an agent can issue action commands by sending integers drawn from a set of valid actions to an environment. Each issued action is then executed k times, where k is uniformly sampled from the set of values $\{2,3,4\}$. Thus, the games chosen implement frame skipping. In response to an action command, in this research, an agent receives from an environment the updated imagery state representation of the environment, i.e. a frame, the reward the agent obtains for the execution of the action, and a boolean value indicating whether the game has terminated after execution of the commanded action or not. Throughout a given game, all actions remain valid continually.

A frame is an array of dimension $210 \times 160 \times 3$, containing the RGB color values of an environment's momentary imagery state representation.

Rewards are float-values, which may be positive, negative, or 0. Due to varying reward signals across the four games, in this research, all positive rewards are clipped to 1.0 and all negative rewards are clipped to -1.0 in order to be able to use the same agent hyperparameters across the different games.

Below, the individual games will be presented.

2.1 Breakout

The goal of Breakout (version *Breakout-v0*) is to destroy rows of bricks along the top of the screen by hitting a ball against the bricks, which lets a hit brick disappear and makes the ball bounce back toward the bottom of the screen. The player must prevent the ball from hitting the screen's bottom by commanding a paddle in such a way that it bounces the ball towards the bricks again. The ball may hit the ground maximally 5 times before the game terminates and must be reset. Each time that a ball hits the ground or that a game is over due to winning or losing it, either of these events is going to be encoded as a terminal game state. Hitting a brick yields a positive reward. In Breakout, an agent can choose from 4 valid actions in each state. Action 0 has no effect. Choosing action 1 for the first time starts the game, but has no effect anymore afterwards. Actions 2 and 3 move the paddle right and left, respectively.

2.2 Q*Bert

In Q*Bert (version *Qbert-v0*), a character has to recolor all cubes in a scene by jumping on top of them at least once. One level of the game is completed as soon as all cubes in the scene have been recolored. At the same time, the agent must avoid enemies and obstacles since making contact with either of them would cost the agent a life and would therefore indicate a terminal state to the agent. Also having won or lost an entire game or level yields a terminal state, as well as falling off cubes does. An agent has 3 lives before the game terminates and has to be reset. Recoloring a yet unvisited cube yields a positive reward. In Q*Bert, an agent can choose from 6 valid actions in each state. Action 0 has no effect. Choosing action 1 for the first time starts the game, but has no effect anymore afterwards. The remaining actions make the character jump in the directions right-back, right-front, left-back, and left-front, respectively.

2.3 Centipede

In Centipede (version *Centipede-v0*), a character has to destroy all parts of a centipede travelling downward a scene, where the centipede is moving from left to right or vice versa, changing its direction and descending one row in the scene whenever it encounters an obstacle or the left or right edge of the game's environment. When the character makes direct contact with a centipede or another enemy, it loses a life. To avoid this, the character can escape enemies by moving up, down, left, right or in a combination of two directions. An agent has 3 lives and besides losing a life, losing or winning a game or level also marks a terminal state. Positive rewards can be earned by destroying a static block in a scene shooting at it repeatedly and by destroying parts of a centipede, hitting it with a laser beam. If hit in the middle, a centipede splits into two independent parts. In Centipede, there are 18 valid actions in each state. Action 0 has no effect. The actions 1 through 5 are *shoot*, *go up*, *go right*, *go left*, and *go down*, respectively. The remaining actions are combinations of the aforementioned actions.

2.4 Enduro

The goal of the racing game Enduro (version *Enduro-v0*) is to pass a given number of competing racers per game level by driving faster than them and overtaking them. If a player does not pass 200 racers in the first level and 300 racers per consecutive level, the player loses the game. This yields a terminal state, as well as winning a game or level does. A player's character drives automatically and a player or agent controls the speed and steering (left vs right) of its racer to avoid crashing into competing racers since a collision throws the agent's racer back on the racing track. Meanwhile, other racers may overtake the agent's racer again, which then have to be overtaken by the agent again. The maximal number of competitors to be overtaken may not exceed a level's initial count. Overtaking a competitor yields a positive reward and being overtaken yields a negative reward. One level finishes as soon as a given number of competitors has been overtaken. In Enduro, an environment's visual appearance changes over time and there are 9 valid actions in each state. Action 0 has no effect. Action 1 speeds up the car and actions 2 and 3 make the car steer the right and left, respectively. The remaining actions are combinations of the aforementioned actions.

3 System Description

3.1 Reinforcement Learning

Reinforcement Learning (RL) has become one of the most popular and most widely used machine learning techniques (Sutton & Barto, 2017). One of the major benefits of the reinforcement learning approach is the fact that it acts largely independent from human supervision, that means no human intervention is required at any time, all the necessary information and feedback is taken directly from the problem environment. RL is usually, as in this thesis, applied in the form of an agent that is confronted with a problem or an environment and through interaction with this environment learns to optimize its behaviour with regards to being successful in the world it has to deal with. The agent has to respond to different states of the world making a decision at each step, in other words, in each state the agent has to choose an action which leads to a different, ideally more favourable, state. The only means by which the agent can evaluate its behaviour and learn are rewards it receives from the environment. Thus the agent's behaviour is driven towards obtaining higher rewards, which in most environments corresponds to solving the problems at hand.

Reinforcement Learning problems can in principle be described as Markov Decision Processes (MDP). MDPs consist of a finite set of states (in many RL environments this set might in fact be continuous and therefore infinite), a finite set of actions, a calculated probability of reaching a successor state through a specific action and a reward for reaching that state. Distant rewards are generally discounted and immediate rewards have consequently a stronger influence on the agent's behaviour.

The overall expected gain (G), that is the predicted total reward of a specific state-action chain is calculated through the following function:

$$G = \sum_{t=0}^{\infty} \gamma^t * r_t \tag{3.1}$$

where the discount factor is denoted by γ and the reward at time t is denoted by r_t. The discount factor is a number between 0 and 1, where a lower number leads to a stronger focus on immediate rewards while a discount factor of 1 would value distant rewards just as highly as immediate ones. In Atari 2600 games the reward is given in the form of the score achieved after performing an action, or more specifically the difference in two subsequently observed scores following a state transition. In order to be consistent between different games the implementations discussed in this paper make use of reward clipping, i.e. positive rewards received from the environment are clipped to 1 while negative rewards are clipped to -1.

There are, however, some particularities that distinguish RL problems, especially in combination with neural networks as will be the case in this paper, from plain MDPs. As already mentioned, the state space in applied RL problems is often not finite, which also entails that it is impossible for the agent to know the best action in a given state; usually there is no single optimal action to choose. For this reason function approximators are used that, instead of calculating the probability of each action leading to a specific state, predict an estimated future reward based on each action given the current state. From this the agent can ultimately choose the most promising action. Applying those approximators results in a very flexible learning process and makes RL feasible in environments with very large or infinite state spaces. It is also worth noting that in large environments, like Atari games, rewards are usually sparse as most state transitions bring no reward at all, so here again using function approximators to predict (potentially quite distant) future rewards is a very useful learning strategy.

3.2 Q-Learning

Q-learning is a type of reinforcement learning in which so called quality-values (Q-values) are used to indicate the expected future rewards of actions based on a current state (Watkins, 1989). The algorithm predicts the Q-values of all state-action pairs, that is of all possible actions in a state, and thereby determines which action is the most promising choice in the current situation. The action with the highest Q-value bares the highest prospective reward. Q-values are updated over time based on experience following the update rule in Eq. 3.2:

$$\begin{aligned} Q_{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \\ *(r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \end{aligned} \tag{3.2}$$

where s_t is the state at t, a_t the action taken at t, r_t the reward obtained at t (i.e. the reward obtained for performing a_t in s_t), α a learning rate, and γ a discount factor.

In standard Q-learning implementations the Q-values of each state-action pair are stored in a look-up table, however, in large and diverse environments, such as games, this is clearly not feasible for various reasons. Storing specific values for all individual state-action pairs leads to an unmanageable amount of information and very restricted and inflexible behaviour as states that have not been previously encountered do not have a Q-value assigned to them and a corresponding action can therefore not be chosen if the agent is not able to generalize from its existing knowledge. Aside from that, with a large or possibly infinite state space it would be very inefficient in terms of memory and processing speed to explicitly store Q-values and iterate through all of them every time a decision has to be made. A better solution is thus to use a deep neural network as a function approximator to estimate Q-values. This allows for efficient decision making based on the network's weight tuning regardless of whether the state at hand is known or not. The network's weights are updated through backpropagation of the experienced rewards and so over time it can be trained to respond to world states by generating Q-values for the possible actions. Instead of looking up explicit Q-values in a table, the system then feeds the current observation (the state of the world) into the neural network which in turn produces estimations of the possible actions' Q-values for this state, based on the current tuning of the network, from which the agent can then choose the most promising one. This empowers a flexible and efficient decision making process in the agent. When using a function approximator for Q-learning the update function becomes:

$$Q_{new}(s_t, a_t) = r_t + \gamma * \max_a Q(s_{t+1}, a) \tag{3.3}$$

for non-terminal states, and:

$$Q_{new}(s_t, a_t) = r_t \tag{3.4}$$

for terminal states.

3.3 Architecture

For the reasons provided above, all RL implementations discussed in this thesis make use of Convolutional Neural Networks (CNN) (LeCun et al., 1998) as their function approximators. The network's architecture was taken from DeepMind's original DQN paper (Mnih et al., 2013) and consists of two convolutional layers, followed by one fully-connected hidden layer and one output layer. The first convolutional layer contains 16 filters of size $8 * 8$ with a stride of 4, the second one contains 32 filters of size $4 * 4$ with a stride of 2, the fully-connected layer consists of 256 nodes, and finally the output layer has as many output nodes as there are possible actions for a given game (varying from 4 to 18). All layers except the output layer use the rectified linear unit (ReLU) activation function.

The states that are fed into the network consist of four frames, the current one and the three previous ones, that have been rescaled to 84x84 pixels and transformed into greyscale in order to reduce each frame's dimensionality from three (RGB) to only one, resulting in a network input of 84x84x4. The Atari learning environment as used in this project already implements a non-stochastic form of frame skipping (Bellemare et al., 2012) that automatically repeats every issued action for k consecutive frames, where k is uniformly sampled from $\{2,3,4\}$. Rewards experienced during these consecutive frames are summed up. This leads to a clear computational advantage as less decisions have to be made by the agent for any given number of frames, thus increasing the number of visited states in a given number of decisions, and additionally results in a greater difference between consecutive states which is beneficial when using a function approximator as states become easier to distinguish.

States in which the current game ends or the agent loses a life are marked as *terminal*, so that during training they can be excluded from reward projection and do not benefit from predicted future rewards which should not be attributed to states that lead to the end of the game.

States are not immediately used for network training as they occur but first stored in a replay memory (Lin, 1992) with a maximum capacity of 1,000,000, once the capacity limit is reached the oldest state in the memory gets replaced with the new one. In order to train the network and update its weights, 32 experiences (i.e. states with corresponding actions and rewards) are randomly and uniformly sampled from the replay memory and used as a batch. This form of batch training in combination with a replay memory has numerous advantages over online training in which the observed states are directly used for training and then discarded. Stored experiences can be used for training multiple times and usually states stacked together in a single batch are not consecutive, this breaks interdepence and correlation between different states which would otherwise negatively effect the training. It is, however, important that the batches are independent and identically distributed in order to avoid any bias towards individual states.

3.4 DQN

In 2013 the DeepMind team introduced a new approach to deep reinforcement learning which they called Deep Q-Learning (Mnih et al., 2013). The networks on which this algorithm is applied are called Deep Q-Networks, or DQNs in short. A DQN is essentially a CNN trained with Q-Learning. That means it takes as its input a world state (in this case four frames of an Atari 2600 game) and produces as its output Q-values for each possible action the agent can take. After every action the network is updated, using replay memory batch training, through back-propagation of the experienced reward and a projected discounted future reward. The discount factor used for predicted rewards was chosen to be 0.99 since rewards in the Atari environments are very sparse and often actions taken lead to a reward that is only experienced in the future, immediacy does not play an important role. RMSprop was selected as the optimizer while the mean-squared error was used as the loss function:

$$\frac{1}{n}\sum_{i=1}^{n}(Yp_i - Yt_i)^2 \tag{3.5}$$

where n is the batch size, i one element from the batch, Yp is the predicted Q-value and Yt is the target Q-value.

A technique that stabilizes training is the addition of a second network, a so called target network (Mnih et al., 2013). Such a network has exactly the same architecture as the main Q-network (the policy network) but is only updated periodically with a delay, thus the target network's weight tuning does most of the times not represent the current state of the Q-network but lags slightly behind in terms of training. The target network is used to calculate the predicted future rewards that are used for updating the Q-network. Since the target network is not updated with every new experience it helps avoid positive feedback loops which in single-network setups could prevent the network from learning properly and instead make it stick to suboptimal behaviour.

The standard exploration method used in DQN learning is diminishing ϵ-greedy (Sutton and Barto, 2017). In this simple technique the agent either chooses a completely random action or consults the Q-network for action selection. The decision for which method to follow is based on random number generation and a threshold ϵ which is decreased over time, so the probability of the agent using the network for action selection increases with every training step. In the implementation used for this thesis ϵ was linearly annealed by subtracting a fixed value after each network update. A fixed minimum value for ϵ ensures that exploration always occurs to a certain degree during training.

3.5 DQV

A variation of standard Q-learning is Quality-Value-Learning (QV-Learning) introduced by Wiering in 2005 in which a state-value function (V-function) additional to the Q-function is tracked and used for updating the Q-values (Wiering, 2005). This increases training speed as the V-function only considers states instead of state-action pairs and the policy is consequently updated more frequently than the Q-function and therefore converges faster to optimal values. QV-Learning can naturally be applied to the DQN model which is then called DQV (Sabatelli et al., 2018). The update function becomes then

$$V_{new}(s_t) = r_t + \gamma * V_{target}(s_{t+1}) \tag{3.6}$$

for the V-values and

$$Q_{new}(s_t, a_t) = r_t + \gamma * V_{target}(s_{t+1}) \tag{3.7}$$

for the Q-values.

DQV in principle follows the workings of DQN, the crucial difference is found in the network setup. In DQV three networks are used: a standard Q-network, a V-network and a target network based on the V-network. The two V-networks follow the same architecture as the Q-network but differ in that they only have a single output node instead of one for every action. As in DQN, the target network is used for the prediction of future rewards and is for stabilization purposes not updated continuously. It is important to note, though, that in DQV the target network follows the V-network instead of the Q-network. Both the Q-network and the V-network are updated based on the reward prediction made by the V-target network using identical loss functions and optimizers which were again chosen to be the mean-squared error and RMSprop respectively.

3.6 Diversity-Driven Exploration

One approach to improve the learning performance of the standard DQN algorithm is to replace the rather basic diminishing ϵ-greedy exploration strategy with a more sophisticated and efficient technique. One such advanced exploration strategy is a diversity driven exploration strategy introduced by Hong and colleagues in 2018 (Hong et al., 2018). This strategy can be applied to various off- and on-policy reinforcement learning algorithms, including DQN, which is then called Div-DQN. In this paper the diversity driven exploration is also applied to a DQV implementation which shall thus be called Div-DQV.

The crucial idea of the diversity driven approach is to make the agent explore a wide state space in a directed and efficient way. So instead of just introducing a degree of randomness into the algorithm, which may lead to unpredictable and unsatisfying learning outcomes as the to be explored state space grows larger, this strategy implements a distance measure which compares the network's current prediction to previous predictions and can reward decisions that lead to the discovery of previously unencountered situations.

In order to drive the learning progress to explore a broader horizon and escape local optima the agent receives imaginary rewards for visiting unknown states. This is realized through a modification of the loss function used for updating the Q-values. More specifically, the standard DQN loss function (traditionally the mean-squared error) is extended by a distance measure in the form of a Kullback-Leibler (KL) divergence. The KL-divergence is zero for identical values and grows larger as the difference between the two values increases. In order to compute this distance measure the Q-values encountered in each state are additionally stored in the replay memory. During training, a softmax function is applied to both the stored Q-values for the sampled state as well as the newly predicted Q-values for said state. The actual distance measure is now determined by calculating the KL-divergence of the two softmax results:

$$L = \alpha * D_{KL}(\pi(a|s)||\pi'(a|s)) \tag{3.8}$$

with D_{KL} indicating the KL-divergence, $\pi(a|s)$ being the softmax result of the newly predicted Q-values, $\pi'(a|s)$ being the softmax result of the Q-values retrieved from the replay memory and α being an exploration factor determining how much exploration should be emphasized. Hong et al. mention an adaptive scaling strategy for α in their paper, however, the implementation used in this thesis uses a linear annealing strategy for α for reasons of simplicity.

The new loss function is now arrived at by simply subtracting the loss term in Eq. (3.8) from the standard MSE loss term. All other elements and properties of the underlying DQN algorithm remain unchanged. Since the distance measure is subtracted from the standard loss term, the loss is lower for larger distances which encourages the agent to explore states that differ greatly from the ones already known.

In terms of computational resources the diversity driven exploration does not add too much overhead in comparison to standard DQN/DQV. The additional storage of Q-values in the replay memory requires some additional memory and the computation of the loss function involves a few more steps as two softmax and one KL-divergence calculation are added. Overall, the training cost is not notably increased through the use of the diversity driven approach.

3.7 NoisyNet

NoisyNet (Fortunato et al., 2017) is an exploration strategy aiming at overcoming limitations of ϵ-greedy and other state-independent exploration strategies in deep RL, as will be outlined below.

The intuition behind this approach is to promote exploration by constantly disturbing the weights and biases of a deep RL agent by noise during the agent's training process. By inducing noise into the agent's predictions, exploration is encouraged since the predictions, which determine which state the agent will visit next, do not exclusively depend on the agent's present training outcome any longer, but also on the induced noise. This works by replacing all linear layers of an artificial neural network used in a deep RL agent by so-called noisy layers.

Each noisy layer is composed of two internal linear layers, each containing both a set of weights and a set of biases. The input to a noisy layer is multiplied by both internal sets of weights separately and both respective outcomes are summed up to a single outcome. Then both sets of biases are added to the combined output of the previous step. The result marks the final output of the noisy layer. While one internal layer is a standard linear layer, the weights and biases of the second internal layer get scaled by random values, i.e. noise (as will be explained in detail below), whenever a forward pass through the network is executed. In the course of this, the agent trains one set of weights that learns the predictions the agent has to produce in the absence of noise once training has finished, and hence the policy, and a second set of weights which determines the extent to which the induced noise affects a prediction of the agent given the input data during training.

Thus, the NoisyNet approach is a state-dependent exploration strategy, whereas state-independent exploration strategies, like the traditional ϵ-greedy exploration strategy, often fail to lead to large-scale deviations from the agent's current policy which would be necessary during training for efficient exploration (Fortunato et al., 2017).

Assumptions about the distribution by which noise is injected into the agent's weights can be neglected since the extent to which the noise affects the predictions is learned automatically by the agent during training.

Considering computing time, the number of parameters to be trained in each linear layer of a network doubles due to having to train two sets of weights and biases instead of one per linear layer in a NoisyNet. Also, repeated computation of noise in a NoisyNet adds on top of the regular training time needed for training a standard deep RL algorithm.

The NoisyNet approach is implemented into DQN and DQV as follows. Since the NoisyNet approach is an exploration strategy itself, it replaces the ϵ-greedy exploration strategy. All linear layers with their respective input and output dimensions in a standard deep RL algorithm get replaced by noisy layers of the same input and output dimensions. The loss function of a given standard deep RL agent remains unchanged, except for the fact that both the policy network and the target network employed for computing the loss get replaced by the introduced NoisyNets. Backpropagation in a NoisyNet happens as usual with respect to the computed loss for all parameters, including both those that learn the policy and those that learn to incorporate the noise. To keep the different deep RL algorithms tested in this research comparable, the overall model architecture and all hyperparameter settings, including the backpropagation algorithm, remained unchanged from those in the standard deep RL algorithms tested here.

Implementing a noisy layer works as follows. The setup of a standard linear layer is described by:

$$y = wx + b \tag{3.9}$$

where y denotes the output of the linear layer, w the weights of the layer, x the input to the layer, and b the layer's biases. In a noisy layer, while in general equation 3.9 to compute y remains intact, w and b from equation 3.9 get replaced as follows:

$$w = \mu^w + \sigma^w \odot \epsilon^w \tag{3.10}$$

$$b = \mu^b + \sigma^b \odot \epsilon^b \tag{3.11}$$

Let p and q denote the input and output dimensions of a given noisy layer, respectively. μ^w and σ^w denote the sets of weights, each of dimension $q \times p$, that learn the network's policy and the weighting of the noise, respectively. μ^b and σ^b denote the sets of biases, each of dimension q, that get tuned to the network's policy and the weighting of the injected noise, respectively. ϵ^w and ϵ^b contain *factorised Gaussian noise*, as introduced by Fortunato et al. (2017), of dimensions $q \times p$ and q, respectively. The symbol $\odot$ denotes element-wise multiplication.

Sampling noise, i.e. factorised Gaussian noise, works as follows. Two sets of random values, ϵ_j and ϵ_i, of dimension p and q, respectively, are sampled from a Gaussian distribution with a mean value of 0.0 and a standard deviation of 1.0. Then, the noise values of both ϵ^w and ϵ^b are computed as follows:

$$\epsilon^w_{i,j} = f(\epsilon_i)f(\epsilon_j) \tag{3.12}$$

$$\epsilon^b_i = f(\epsilon_i) \tag{3.13}$$

where f is defined as $f(x) = sgn(x)\sqrt{|x|}$, as proposed by Fortunato et al. (2017).

Noise gets resampled each time before a forward pass through a NoisyNet is executed during training. In cases of a batch being passed to a NoisyNet, as happens during batch training, noise is resampled only once before the entire batch is passed through the network.

As suggested by Fortunato et al. (2017), each $\mu_{i,j}$ is initialized to a value randomly drawn from an independent uniform distribution $\mathcal{U}[-\sqrt{\frac{1}{p}}, +\sqrt{\frac{1}{p}}]$ and each $\sigma_{i,j}$ is initialized to $\frac{0.5}{\sqrt{p}}$, where p denotes a layer's input dimension.

For evaluation of the training progress of an agent, a partial copy of the agent's NoisyNet is created, which contains the unmodified convolutional layers and from each noisy layer the subset of parameters trained on the agent's policy, i.e. μ^w and μ^b. The resulting non-noisy policy network is then employed for playing evaluation games.

4 Experiments

4.1 Experimental Setup

In order to be able to compare the training performance of the six deep reinforcement learning algorithms discussed in this paper, a number of tests were conducted. Four Atari game environments from the OpenAI gym were chosen in which the algorithms' learning performances, that is the achieved scores, and more specifically the development of achieved scores over the course of the training, were measured: Breakout, Centipede, Q*Bert and Enduro. Each individual run consisted of exactly 10,000,000 network updates, with the exception of Enduro which, due to time issues, has only been trained for 7,000,000 network updates. When the agent loses a life or finishes the game an epoch ends and the environment is reset. In order to evaluate the agents' training after its conclusion, 20 evaluation games were played every 10,000 network updates and their mean scores recorded. Since these evaluation games exclusively used the network's output to choose an action, the scores obtained here are fully representative of the agent's training progress and therefore the algorithm's performance. For each reinforcement learning algorithm discussed in this thesis ten independent runs were performed for each of the chosen games. A complete training run consisted of 10 million network updates (7 million for Enduro), including 20 thousand evaluation games (14 thousand for Enduro).

4.2 Hyperparameters

The tuning and setup of the learning algorithms was taken from the 2015 DeepMind article (Mnih et al., 2015) and always kept consistent so that a proper comparison can be made. Actual training of the agents, that is updating of the networks, always commenced after 50,000 states had been visited and stored in the replay memory. This was supposed to ensure that there always was a large enough collection of previous states to sample from. Actions in the states preceding the 50,000 mark were chosen completely randomly. The replay memory had a capacity of 1,000,000 experiences where one state consisted of 4 consecutive frames, the batch size of experiences to be used for updating the network was kept at 32. An agent's target network used for reward prediction was updated to match the Q-network or V-network every 10,000 updates. The discount factor for rewards was 0.99, the learning rate 0.00025 and the linear ϵ decay rate 1e-06 with a minimum ϵ of 0.1. The RMSProp optimizer used a gradient momentum of 0.95 and a minimum gradient of 0.1.

Since the paper by Hong et al. (2018) does not state a decay rate for the factor α it was set to 1e-06 with a minimum α of 0.1 in order to match ϵ in the standard implementations.

4.3 Resources

The complete code of the implementations used in this thesis and the raw output data of the experiments can be found in the following GitHub repository: /Jannik0/RUG_ReinforcementLearning.

5 Results

5.1 Collection of Scores

In order to obtain scores representative of the training process and outcome of the six different algorithms while training on each of the four games, the scores of 20 evaluation games that are played every ten thousand frames (and therefore network updates) were recorded and averaged. Those scores were then again averaged over ten independent full training runs of ten million frames (seven million in the case of Enduro).

The plots of the individual results obtained that way can be found in Appendix C. In those graphs the thick line represents the average score the respective algorithm achieved at each evaluation session over 20 games and 10 runs. The graphs additionally contain error bars for every data point that are based on the lowest and highest average score achieved over the 20 evaluation games at this evaluation session by any of the 10 runs.

The key results and the comparison between the six algorithms will be dealt with in the following.

5.2 Key Results

The plots included in this section contain smoothened training curves of all six tested algorithms, the results have been bundled for each of the four games to make a comparison more intuitive. For detailed and unsmoothened individual results with error bars please refer to appendix C. The tables presented here contain measurements of the very last evaluation sessions (after ten million network updates - or after seven million in the case of Enduro) of the ten independent runs.

5.2.1 Breakout

All six algorithms clearly learned to play the game of Breakout as the learning curves in Figure 5.1 indicate. The three DQV based algorithms, namely the standard DQV, the diversity-driven DQV and the NoisyNet DQV, improved their performance significantly faster in the beginning compared to all three DQN algorithms. However, they also settle earlier than the standard DQN implementation which towards the end of the training process achieves the highest scores among all algorithms. Interesting to note is furthermore that the more sophisticated exploration strategies did not have the desired effect in the training of Breakout as they do not reach the performance of their respective underlying algorithm at any point, with NoisyNet DQV being the only exception as it manages to outperform the base DQV towards the very end of the training process. The training curves of the two DQN variants (Div-DQN and NoisyNet DQN) are almost identical and slightly favour the diversity-driven approach while the DQV variants' curves show great discrepancies with the NoisyNet clearly achieving better scores throughout most of the training process. When comparing the two diversity-driven implementations it is observable that the DQV variant has a better start than the DQN variant but quickly transitions into an almost linear training progress while the progress made by Div-DQN is slightly exponential and even surpasses Div-DQN by a marginal degree near the end of the training.

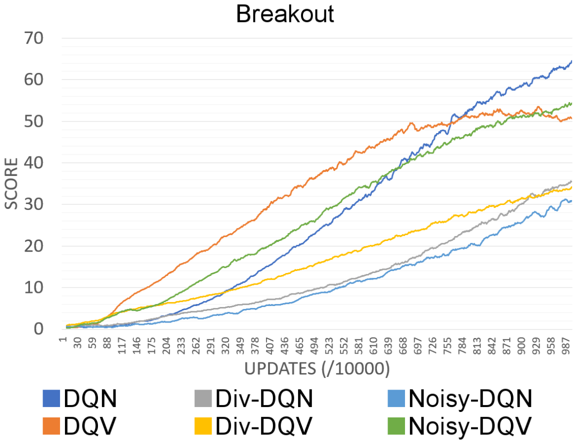

Figure 5.1: Average results of the six algorithms' evaluation games while learning to play Breakout (10 million updates with 20 evaluation games every 10 thousand updates; lines are smoothened)

Table 5.1 shows the results of the last evaluation sessions of the ten training runs for each algorithm after training on Breakout. Here it becomes apparent that the algorithms with the lowest average scores (namely Div-DQN, Div-DQV and NoisyNet-DQN) also have the largest spread between their lowest and highest achieved scores and especialy show a significant deterioration of their scores' lower bound when compared to the three better performing algorithms. The algorithms with the highest average score, DQN, also achieved the highest lower and upper bounds.

5.2.2 Centipede

In Figure 5.2 the training results of the six algorithms for the game Centipede can be found. Although none of the algorithms really managed to clearly improve their performance throughout the

Table 5.1: Summary of scores achieved in the last evaluation session after 10 million updates in the game Breakout

Algorithm	Average Score (±SE)	Min. Score	Max. Score
DQN	64.9 (±1.9)	58.2	76.4
DQV	49.3 (±2.7)	37.7	62.2
Div-DQN	35.5 (±5.6)	10.7	54.3
Div-DQV	35.2 (±3.8)	17.9	52.5
Noisy-DQN	31.5 (±6.6)	2.1	53.2
Noisy-DQV	55.0 (±2.4)	45.3	70.4

training, some progress can be observed in the beginning, especially for the DQN family of algorithms. The three DQN implementations also outperform all three DQV algorithms during the last two thirds of the training process. Additionally, a clear order of variants becomes apparent which applies to both the DQN as well as the DQV family: The NoisyNet architecture achieves the highest scores, followed by the diversity-driven exploration strategy. Both exploration extensions lead to improved performance over their base algorithms and progress fairly similarly over the course of the training duration, only the comparison between the DQN variants shows a larger difference towards the end of the training.

The two algorithms that on average performed best in the game of Centipede, NoisyNet DQN and Div-DQN, achieved the highest lower bound of scores (as can be seen in Table 5.2) which is an important contributing factor to their comparably high average scores. However, the highest overall score in a any single run, by a considerable margin, was obtained by the NoisyNet variant of DQV, but this algorithm also has a rather low lower bound of scores leading to an overall worse average performance compared to the two aforementioned algorithms.

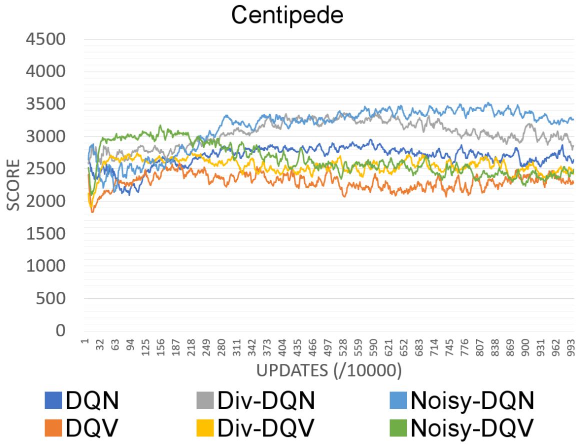

Figure 5.2: Average results of the six algorithms' evaluation games while learning to play Centipede (10 million updates with 20 evaluation games every 10 thousand updates; lines are smoothened)

Table 5.2: Summary of scores achieved in the last evaluation session after 10 million updates in the game Centipede

Algorithm	Average Score (±SE)	Min. Score	Max. Score
DQN	2784 (±234)	1260	4017
DQV	2603 (±272)	1380	4421
Div-DQN	2938 (±232)	2211	4607
Div-DQV	2505 (±203)	1720	3542
Noisy-DQN	3383 (±130)	2776	4172
Noisy-DQV	2709 (±337)	1484	5106

5.2.3 Q*Bert

Figure 5.3 summarizes the results obtained during the training process with the game Q*Bert. Now again, all six algorithms clearly managed to learn to play the game and greatly improve their performances over the course of the training. The DQV algorithms once again have a more successful initial learning period but this time they also manage to maintain this advantage throughout the entirety of the training process. Ultimately the three DQV implementations achieve higher scores than their respective DQN counterparts and for the most part even higher than all three DQN algorithms, only the base DQV implementation is eventually outperformed by the diversity-driven DQN variant. In Q*Bert the extended exploration strategies show their strengths and clearly perform better than their corresponding base algorithms in both the DQN and DQV families. In DQN the diversity-driven exploration strategy maintains a relatively stable advantage over the NoisyNet architecture, even though the difference is not too distinct. Both extensions display a faster and more stable training process than the base DQN implementation whose performance even decreases towards the end of the training. The curves of the three DQV implementations take a very similar course among each other, however, here again both the diversity-driven exploration as well as the NoisyNet architecture clearly outperform the base DQV algorithm throughout almost the entire training process. Here the performance difference between Div-DQV and NoisyNet DQV is even smaller than between Div-DQN and NoisyNet DQN and even though the diversity-driven approach achieves higher scores during the largest part of the training the NoisyNet implementation takes over in the last third and ultimately reaches a higher average score at the end of the tranining.

As can be seen in Table 5.3, DQN achieves considerably worse results in Q*Bert than any other implementation. The average score as well as the lowest and highest score are far below that of the other compared implementations. One interesting observation is that the Div-DQN and NoisyNet DQN implementations achieve maximum scores comparable to those of the three DQV algorithms, however, their lower bound of scores is far below that of the DQV family which explains the overall performance difference. Even though Div-DQV's average score is above that of the base DQV implementation both the lower and upper bounds are below that of DQV, this indicates that more Div-DQV runs came close to the algorithm's highest scores compared to DQV. By far the highest lower score bound belongs to the overall best performing algorithm, NoisyNet DQV.

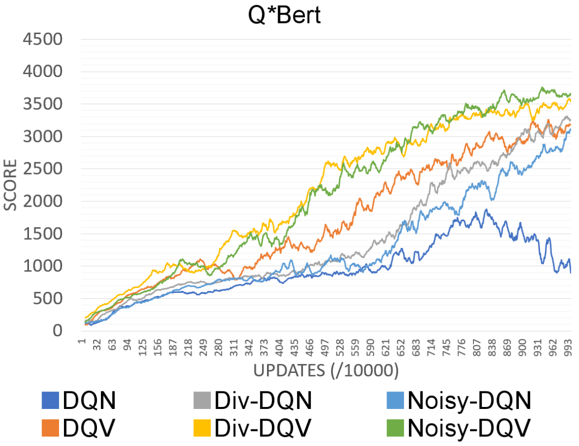

Figure 5.3: Average results of the six algorithms' evaluation games while learning to play Q*Bert (10 million updates with 20 evaluation games every 10 thousand updates; lines are smoothened)

Table 5.3: Summary of scores achieved in the last evaluation session after 10 million updates in the game Q*Bert

Algorithm	**Average Score (±SE)**	**Min. Score**	**Max. Score**
DQN	485 (±117)	144	1413
DQV	3187 (±255)	2039	4310
Div-DQN	3339 (±325)	700	4201
Div-DQV	3449 (±221)	2024	4000
Noisy-DQN	2883 (±357)	568	4111
Noisy-DQV	3726 (±87)	3393	4290

5.2.4 Enduro

The training progression of the six algorithms in the game Enduro can be seen in Figure 5.4. All implemented algorithms showed very consistent and rigorous improvements in their performance in playing the game. Overall, the maximum scores obtained by any of the implementations towards the end of the

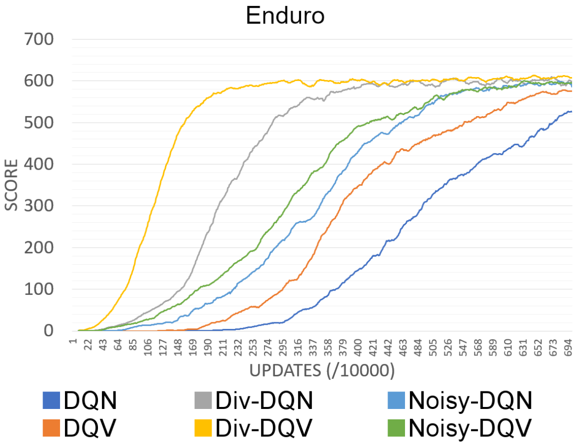

Figure 5.4: Average results of the six algorithms' evaluation games while learning to play Enduro (7 million updates with 20 evaluation games every 10 thousand updates; lines are smoothened)

training is very similar, however, the learning onset and speed of improvement differs greatly between the different algorithms. The three DQV-based algorithms outperform their respective DQN counterparts by a significant degree, even though the difference is not as prominent between the two NoisyNet implementations. Still, all three DQV implementations start improving earlier than their DQN counterparts and also maintain this advantage until the end of the training. When comparing the different exploration strategies a clear tendency becomes apparent as well. The more sophisticated exploration strategies led to a very distinct improvement in performance over their base algorithms with Div-DQN and NoisyNet-DQN clearly and consistently outperforming the standard DQN, and Div-DQV and NoisyNet-DQV clearly and consistently outperforming the standard DQV. Both NoisyNet extensions have a clear advantage over the standard DQN and DQV from the beginning of the training all the way through to the end. However, by far the best learning results were achieved by the diversity-driven exploration strategy with both Div-DQN as well as Div-DQV showing considerable advantages over the other four algorithms with an earlier improvement onset, especially for Div-DQV, and a much earlier performance peak, while also maintaining the smoothest and thus most stable learning curve. Ultimately, however, in the game of Enduro the differences between the tested algorithms lie more in the speed of performance increase rather than the final achieved scores.

Once more, the base DQN algorithm performs the worst among the six algorithms with the lowest average and maximum scores, as Table 5.4 shows. The ranking of the final scores is for the most part consistent between the average score and the lowest score with the only grave exception being the minimum final score of Noisy-DQN which is by far the lowest while the algorithm's average final score is only the second lowest. Similar observations can be made about the final maximum scores. Here Noisy-DQN, the algorithm with the overall second worst training performance and average score, has the third highest score, while the ranking of the other five algorithms remains faithful to the other two scoring categories. Div-DQV obtained the highest scores in all three categories and furthermore, the difference between Div-DQV's lowest and highest scores is also the smallest which also explains its overall high and stable results. Aside from that, the final scores reinforce the impression of the training curves with the DQV-based algorithms outperforming their DQN-based counterparts and the exploration extensions leading to an advantage over the standard implementations with the diversity-driven approach yielding the best results.

Table 5.4: Summary of scores achieved in the last evaluation session after 7 million updates in the game Enduro

Algorithm	**Average Score (±SE)**	**Min. Score**	**Max. Score**
DQN	524.9 (±26.6)	359.8	596.3
DQV	580.1 (±9.0)	504.5	601.1
Div-DQN	598.8 (±6.1)	560.3	617.3
Div-DQV	611.0 (±21.3)	591.2	635.4
Noisy-DQN	536.3 (±44.8)	152.2	621.6
Noisy-DQV	589.7 (±11.2)	502.9	622.6

6 Conclusion and Discussion

Throughout this thesis we compared the deep RL algorithms DQN and DQV in their original forms and in combination with both the diversity-driven and the NoisyNet exploration strategies separately.

As can be seen from the results, it is uncertain how informative the evaluation scores obtained for the game Centipede are since learning performance is low on this game across all algorithms. A likely reason for that may be the usage of a too small model architecture since previous research successfully training an agent on playing this game used a larger model architecture than used in this research (Mnih et al., 2015; Hessel et al., 2017). Still, test scores obtained on Centipede will be included in this conclusion and discussion.

A ranking based on the results of how the algorithms performed across the four games in terms of final evaluation scores is presented in table D.1 in appendix D. Ranking is done because of varying reward signals across the games, which prevents a direct comparison of the evaluation scores across games.

The ranking can be summarized as follows. While both NoisyNet variants, NoisyNet-DQN and NoisyNet-DQV, outperformed all other algorithms and share the first rank, the second rank is shared by DQV and Div-DQN. They are followed by Div-DQV and DQN on the third and fourth rank, respectively.

Supporting the results found by Sabatelli et al. (2018), DQV demonstrated better learning performance throughout the four games than DQN. Moreover, it can be concluded that also DQV profits from at least a subset of the exploration strategies originally introduced to DQN. This is supported by the observation that the standard DQV algorithm only scored rank 2 across the four games, while NoisyNet-DQV scored rank 1.

Only the ranks 3 and 4 are not shared by both a DQN and DQV variant. At this point in the ranking, a DQV variant, i.e. Div-DQV, outperforms a DQN variant, i.e. the standard DQN algorithm. From this, it can be concluded that DQV-based variants have an overall slight advantage over DQN-based variants with respect to the final learning outcome given a fixed amount of training, which entails a slightly increased learning speed of DQV variants compared to DQN variants.

Besides focusing on obtained evaluation scores, further differences of the algorithms can be observed over the course of training with respect to training speed and the spread of the evaluation scores obtained during training, as can be seen in figures C.1 through C.24. From these figures, it seems that the learning process of DQV variants is more gradual and steady than that of DQN variants. Also, the spread between the highest and lowest evaluation scores obtained in a single evaluation session is smaller in DQV variants than in DQN variants. This holds for Breakout and Q*Bert. In Enduro, particularly the high learning speed of the diversity-driven approaches until their convergence onto a certain evaluation score stands out.

Another important observation is that DQV did not only outperform DQN with respect to final evaluation scores, but also showed faster training speed than DQN over long initial periods of training across all games, as can be seen from the figures in the results section.

Overall, the NoisyNet approache seems to outperform both the diversity-driven approach and the standard algorithms DQN and DQV, while the diversity-driven approach seems to largely outperform only DQN.

Generally, whether enhanced exploration benefits training outcome seems to be dependent on the concrete learning objective at hand. While the standard DQN, using the standard ϵ-greedy exploration strategy, outperformed all other algorithms in Breakout, it showed unstable learning Q*Bert, where evaluation scores dropped again after some initial successful training. This observation emphasizes the importance of the search for a well-adjusted balance between exploration and exploitation.

The last observation to notice is that previous research using the same DQN model architecture as used in this thesis reported larger evaluation scores for DQN on the games Breakout and Q*Bert and lower scores on Enduro (Mnih et al., 2013). The question remains whether this difference is caused by the fact that Mnih et al. (2013) used a fixed frame skipping rate, while in this research a variable frame skipping rate was used. Investigating the influence of differing frame skipping techniques on an algorithm's learning performance may be subject to further research.

References

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *arXiv e-prints*, art. arXiv:1207.4708, 2012.

L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst. *Reinforcement learning and dynamic programming using function approximators.* CRC press, 2017.

M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017. URL `http://arxiv.org/abs/1706.10295`.

M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL `http://arxiv.org/abs/1710.02298`.

T. Hill, L. Marquez, M. O'Connor, and W. Remus. Artificial neural network models for forecasting and decision making. *International journal of forecasting*, 10(1):5–15, 1994.

Z. W. Hong, T. Y. Shann, S. Y. Su, Y. H. Chang, and C. Y. Lee. Diversity-driven exploration strategy for deep reinforcement learning. *arXiv e-prints*, art. arXiv:1802.04564, 2018.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

DK Kim, M. Liu, S. Omidshafiei, S. Lopez-Cot, M. Riemer, G. Tesauro, M. Campbell, S. Mourad, G. Habibi, and J. P. How. Heterogeneous knowledge transfer via hierarchical teaching in cooperative multiagent reinforcement learning. 2019.

J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. doi: 10.1177/0278364913495721. URL `https://doi.org/10.1177/0278364913495721`.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 1998.

J. W. Lee. Stock price prediction using reinforcement learning. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No.01TH8570)*, volume 1, pages 690–695 vol.1, June 2001. doi: 10.1109/ISIE.2001.931880.

L. J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

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*, 518:529 EP –, 2015. URL `https://doi.org/10.1038/nature14236`.

S. J. Russell and P. Norvig. *Artificial intelligence: A modern approach.* Pearson, Upper Saddle River, 2016. ISBN 9781292153964.

M. Sabatelli, G. Louppe, P. Geurts, and M. A. Wiering. Deep quality-value (dqv) learning. *ArXiv e-prints*, art. arXiv:1810.00368, 2018.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2017.

C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL `https://doi.org/10.1007/BF00992698`.

M. A. Wiering. Qv (lambda)-learning: A new onpolicy reinforcement learning algrithm. *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 17–18, 2005.

A Division of Work

Table A.1:
Implementations and Experiments

Algorithm	**Contributor**
DQN	Daniel & Jannik
DQV	Daniel & Jannik
Div-DQN	Jannik
Div-DQV	Jannik
NoisyNet DQN	Daniel
NoisyNet DQV	Daniel

Table A.2:
Thesis Writing

Section	**Contributor**
Abstract	Daniel & Jannik
Introduction	Daniel
The Games	Daniel
System Description (except NoisyNet)	Jannik
System Description (only NoisyNet)	Daniel
Experiments	Jannik
Results	Jannik
Conclusion and Discussion	Daniel
Quality assurance and improvements of all sections	Daniel & Jannik

B Hyperparameters

Table B.1:
Parameter Values

Parameter	**Value**
replay memory capacity	1000000
training start	50000
target network update	10000
minibatch size	32
discount factor	0.99
learning rate	0.00025
RMSProp gradient momentum	0.95
RMSProp gradient minimum	0.01
exploration start (ϵ/α)	1
exploration decay (ϵ/α)	1e-06
exploration minimum (ϵ/α)	0.1

C Individual Results

The graphs found on the following pages show the unsmoothened results of the evaluation games performed by the agents at multiple stages during training (see section *Experimental Setup* for more details). The values are the average scores achieved over 10 independent runs, each with 20 evaluation games played at each interval, and include error bars based on the best and worst performing run at each evaluation session.

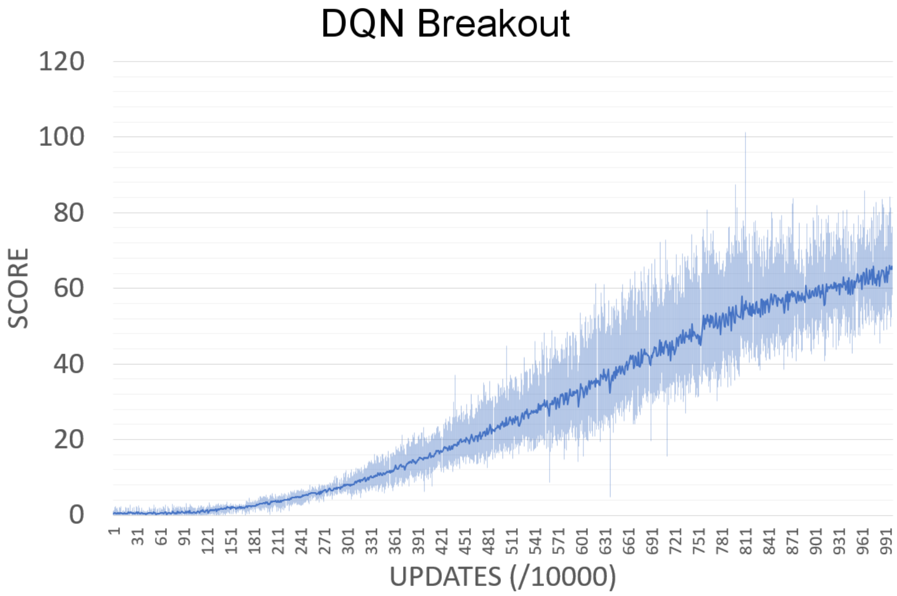

Figure C.1: Individual results for DQN in Breakout

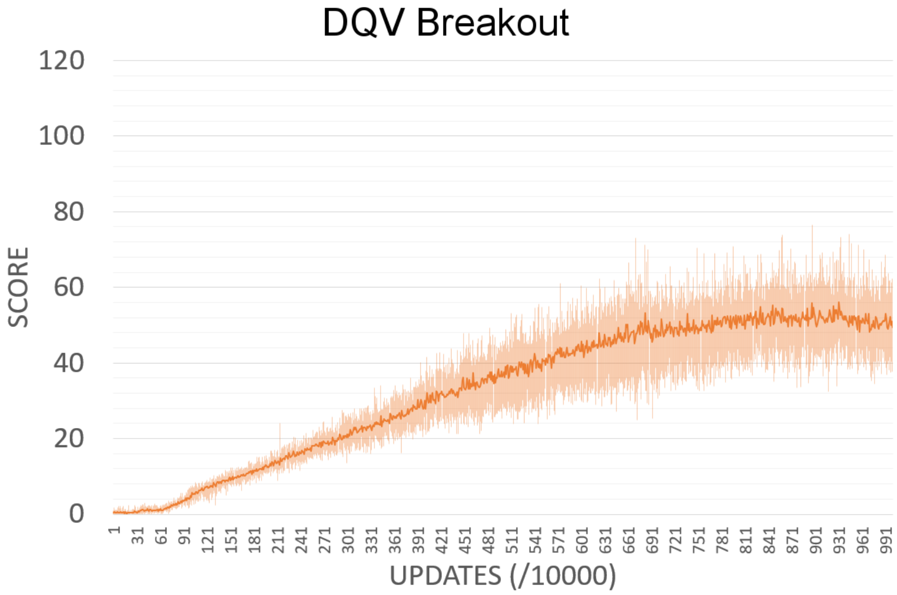

Figure C.2: Individual results for DQV in Breakout

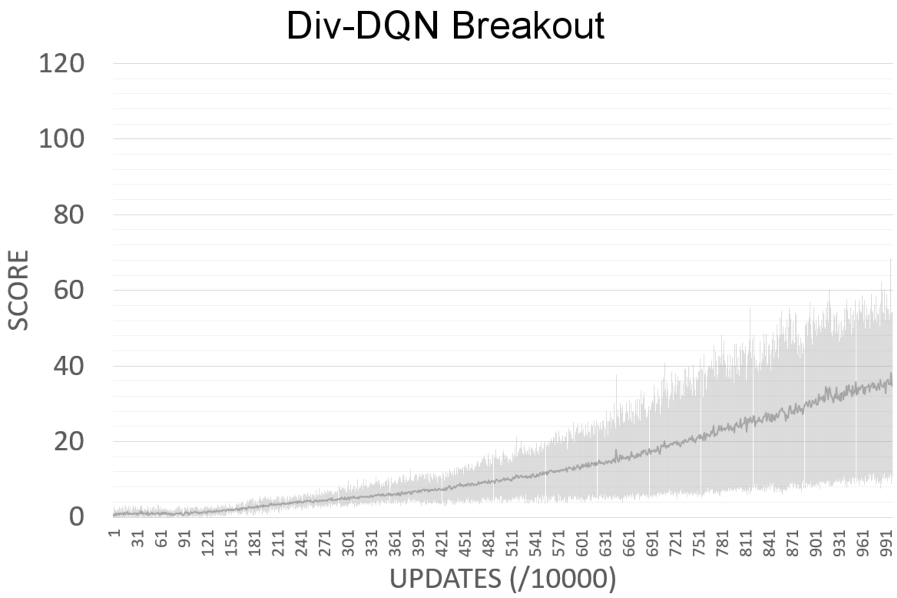

Figure C.3: Individual results for Div-DQN in Breakout

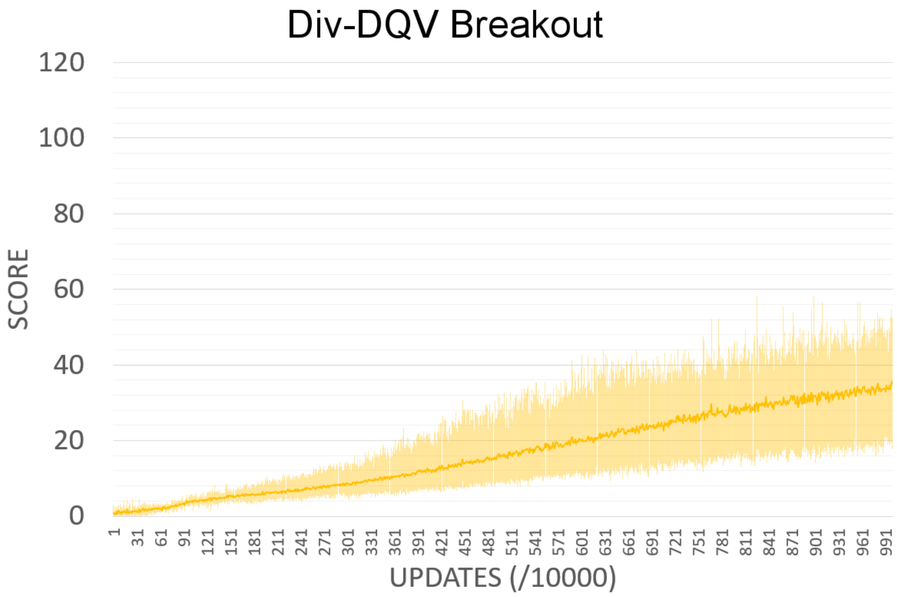

Figure C.4: Individual results for Div-DQV in Breakout

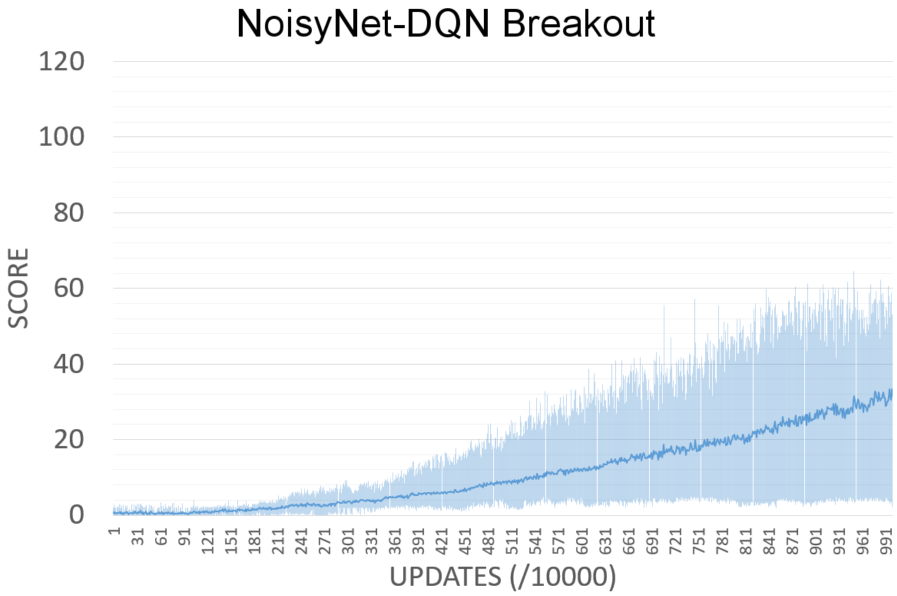

Figure C.5: Individual results for NoisyNet-DQN in Breakout

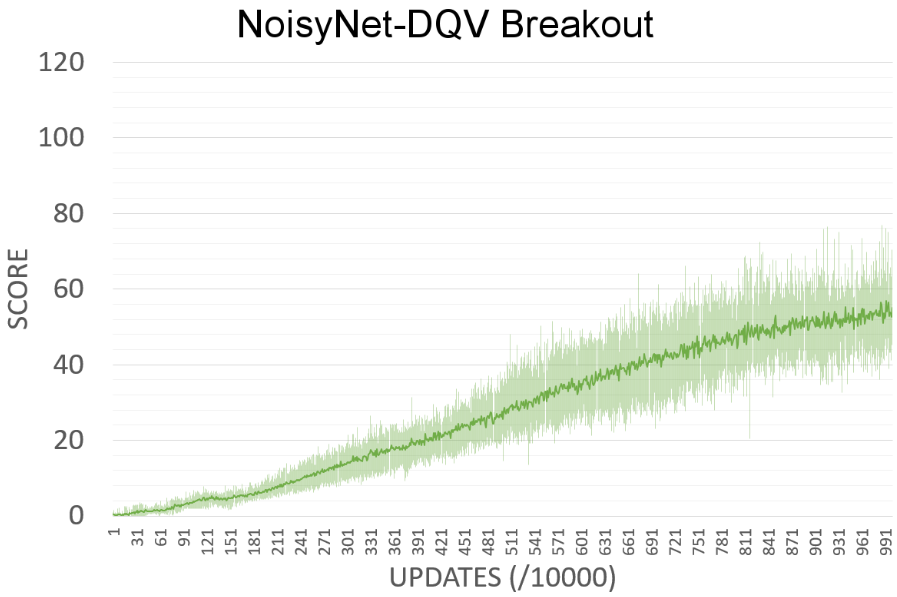

Figure C.6: Individual results for NoisyNet-DQV in Breakout

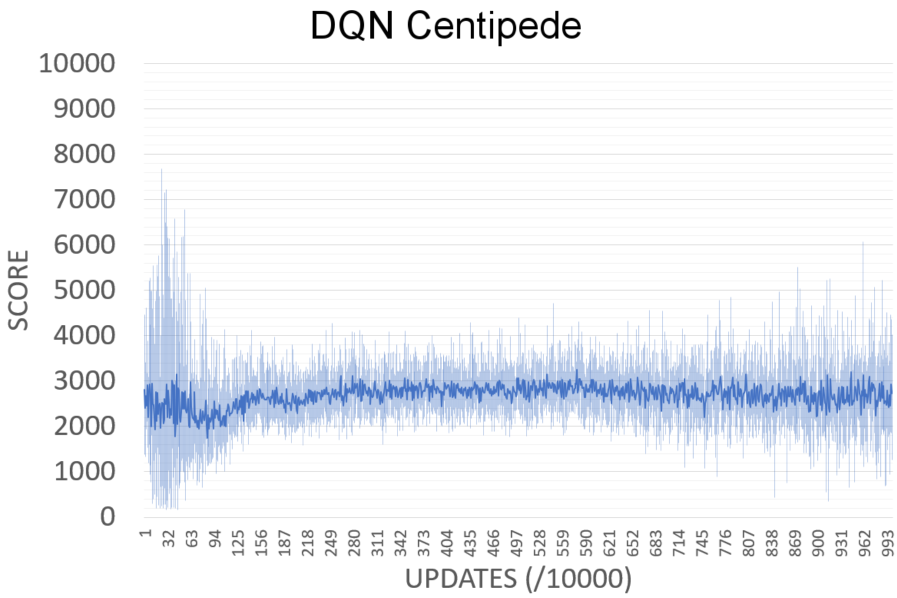

Figure C.7: Individual results for DQN in Centipede

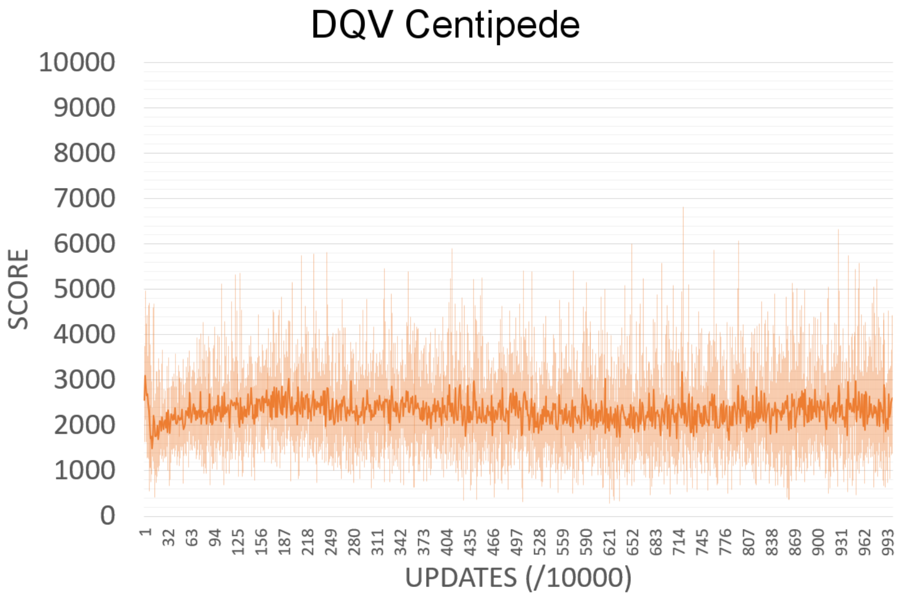

Figure C.8: Individual results for DQV in Centipede

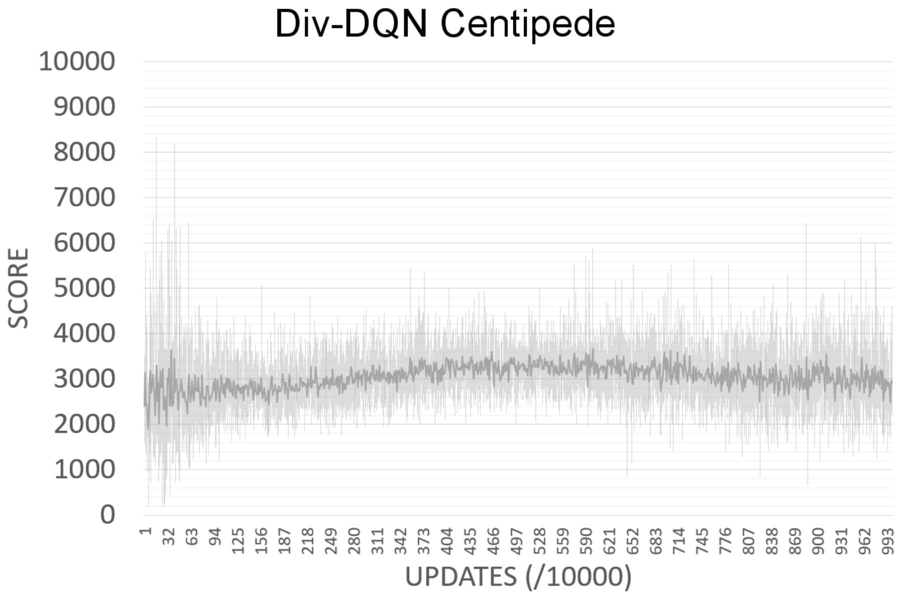

Figure C.9: Individual results for Div-DQN in Centipede

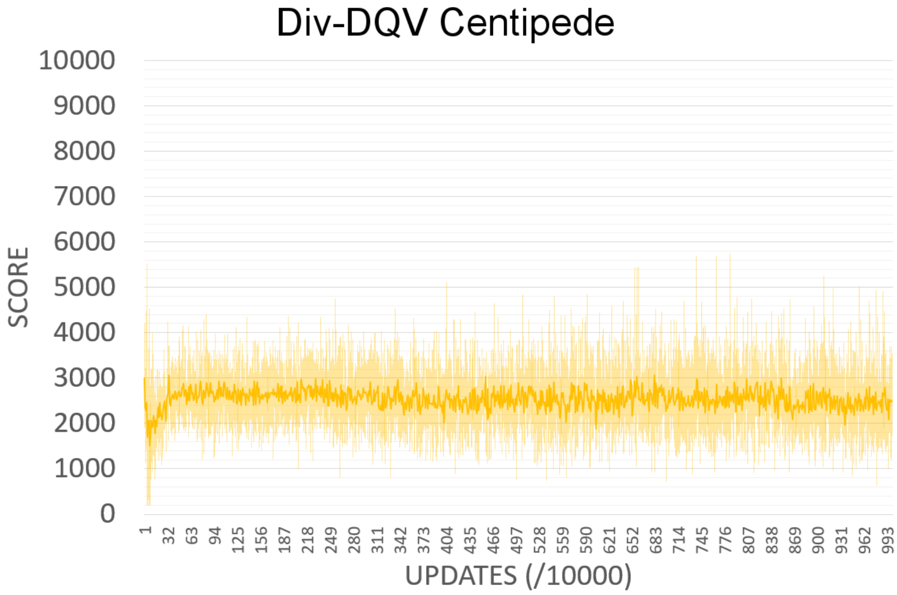

Figure C.10: Individual results for Div-DQV in Centipede

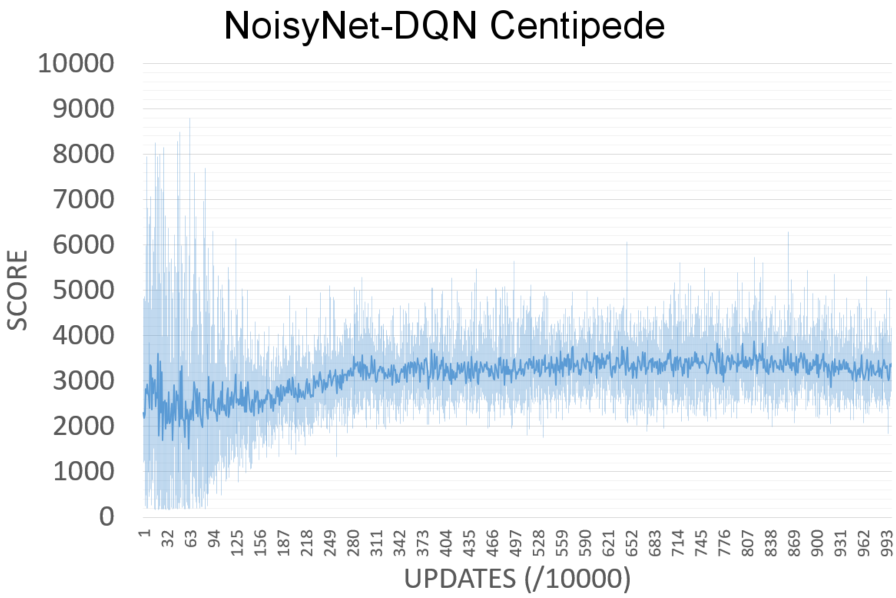

Figure C.11: Individual results for NoisyNet-DQN in Centipede

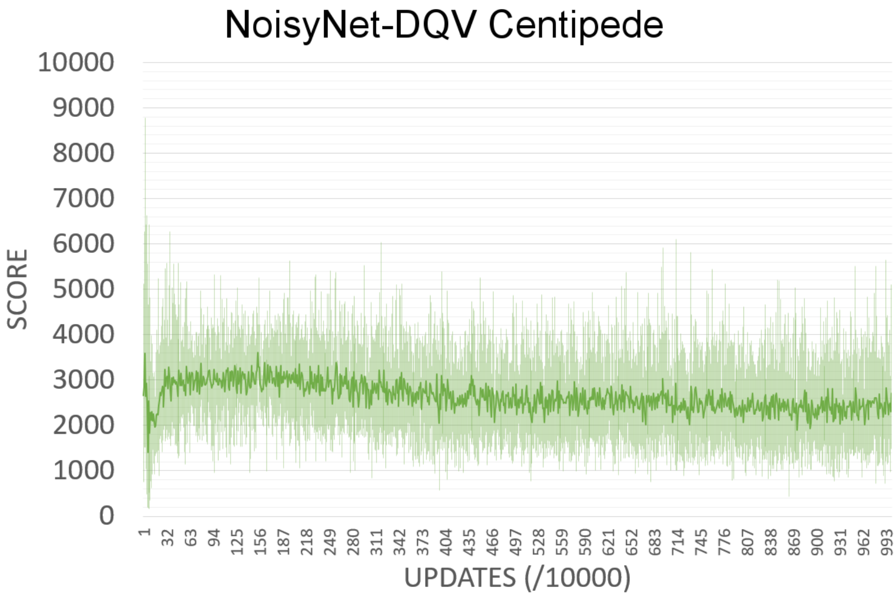

Figure C.12: Individual results for NoisyNet-DQV in Centipede

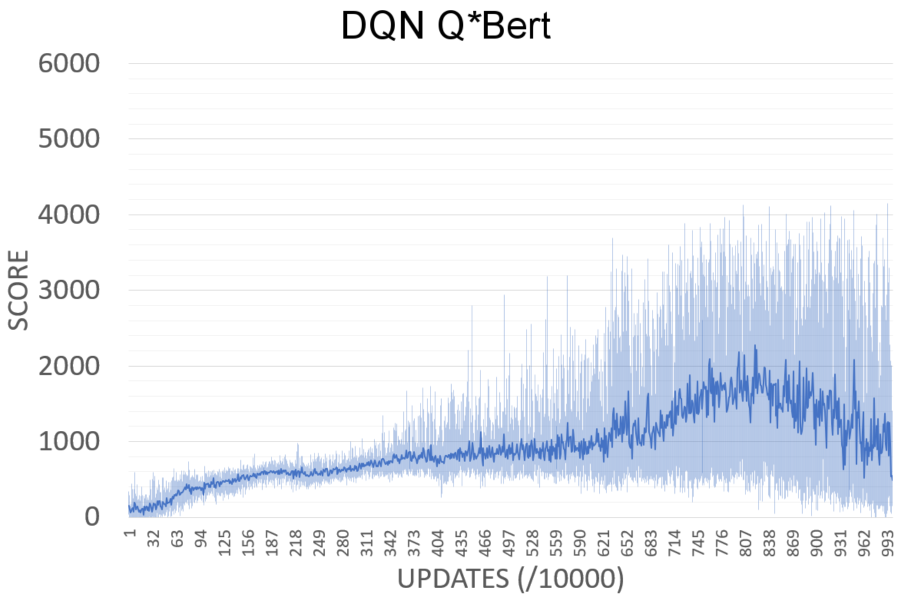

Figure C.13: Individual results for DQN in Q*Bert

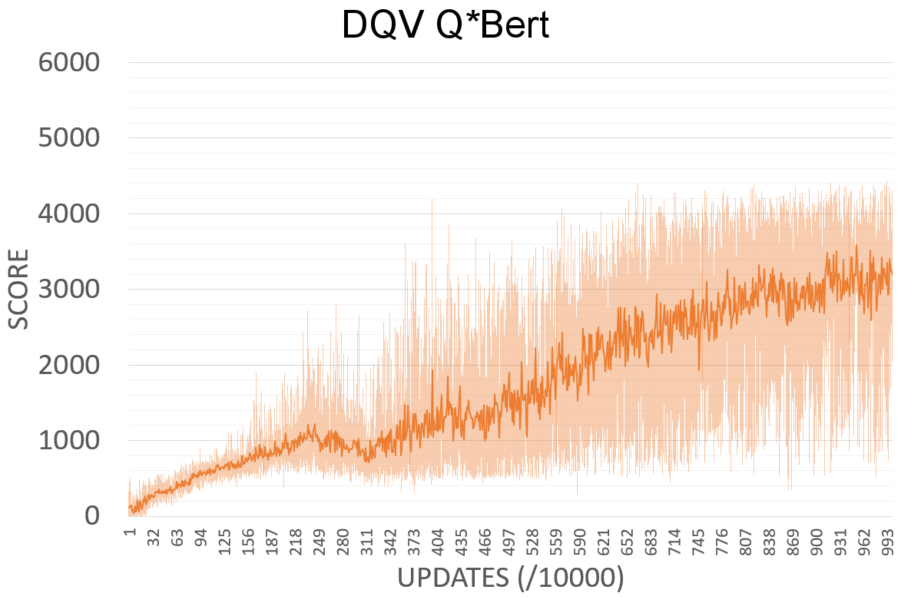

Figure C.14: Individual results for DQV in Q*Bert

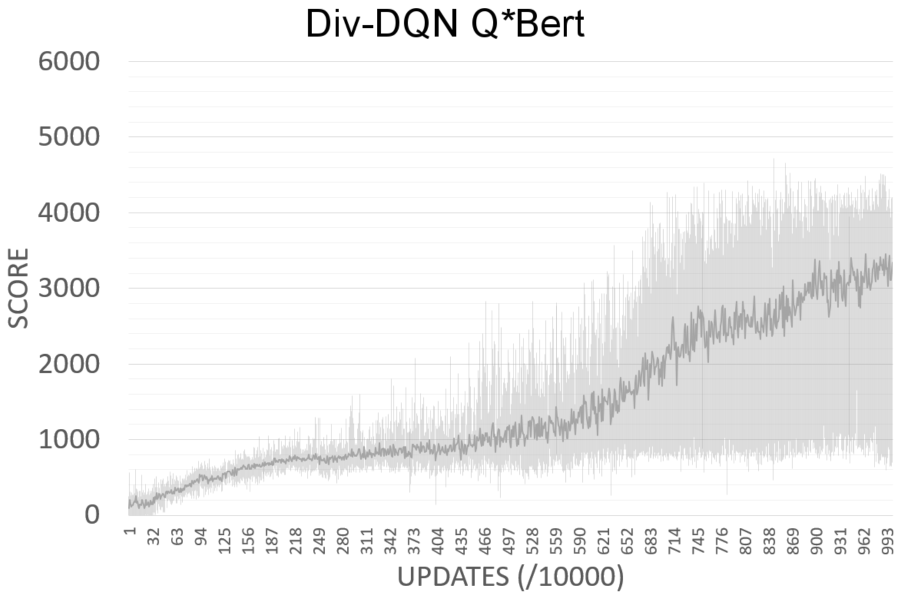

Figure C.15: Individual results for Div-DQN in Q*Bert

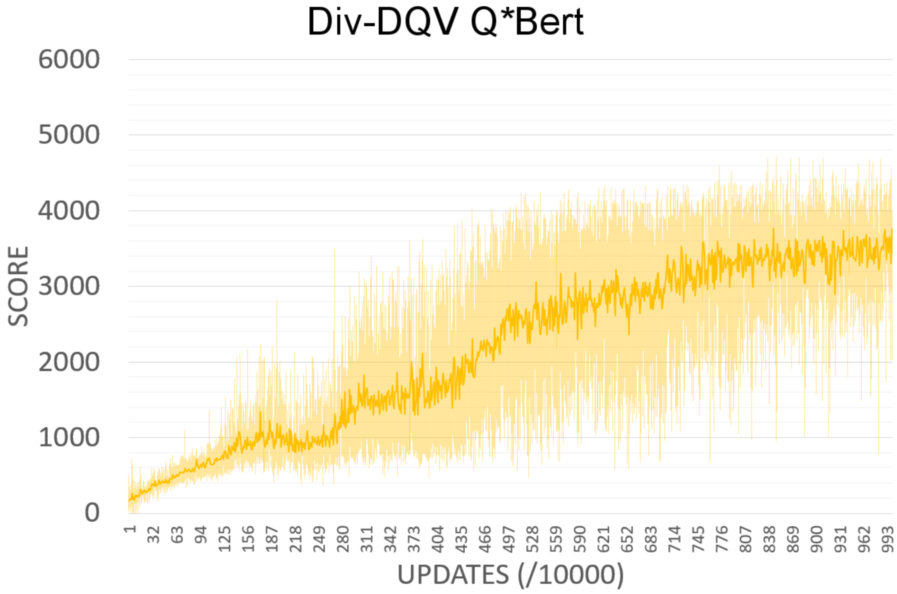

Figure C.16: Individual results for Div-DQV in Q*Bert

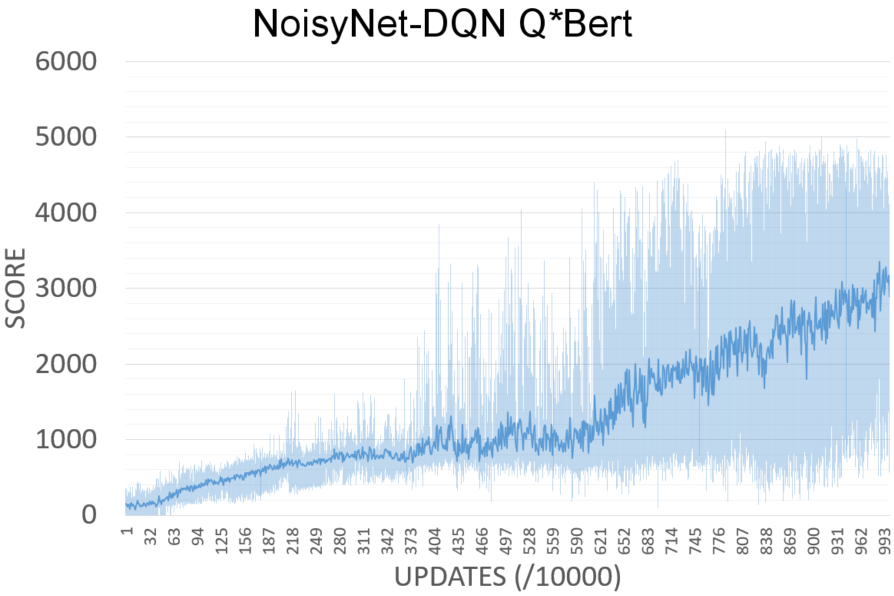

Figure C.17: Individual results for NoisyNet-DQN in Q*Bert

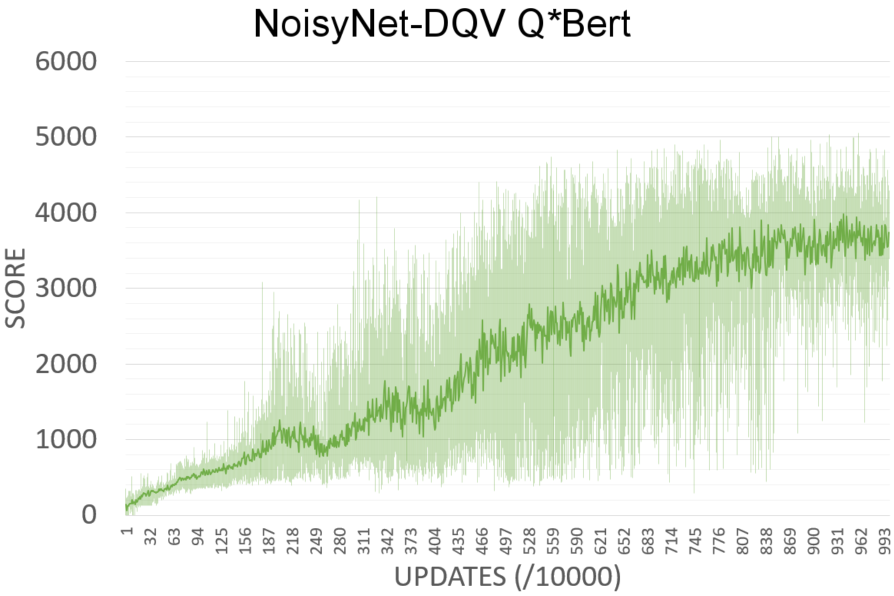

Figure C.18: Individual results for NoisyNet-DQV in Q*Bert

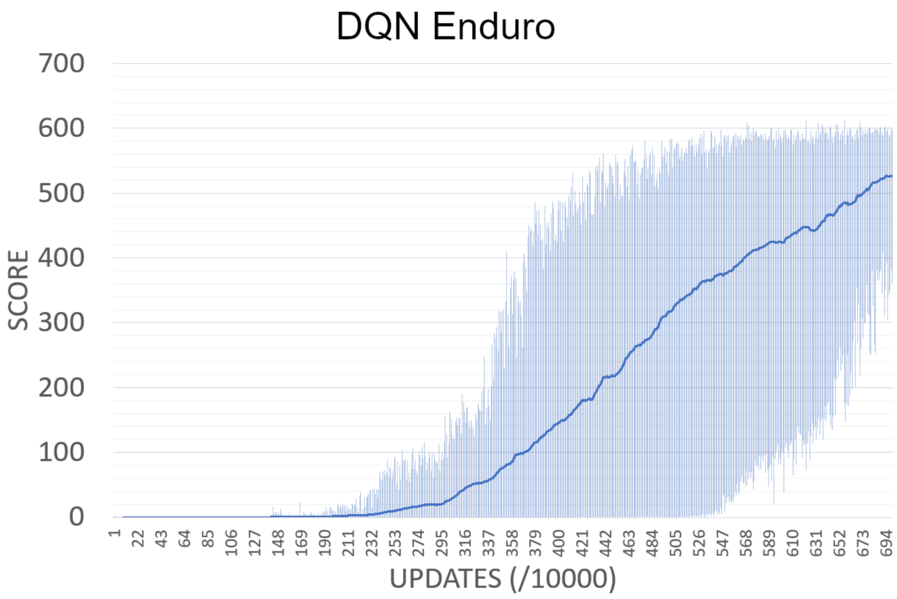

Figure C.19: Individual results for DQN in Enduro

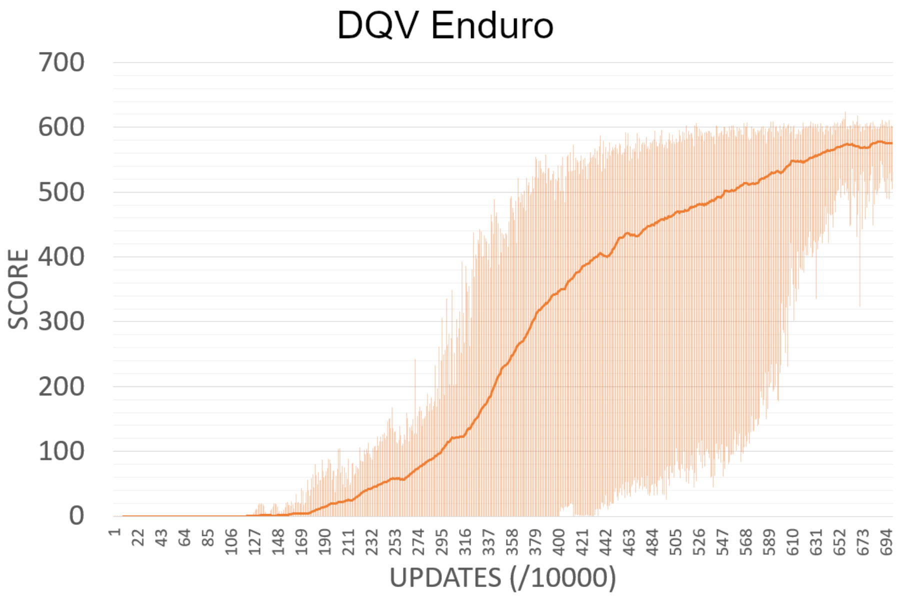

Figure C.20: Individual results for DQV in Enduro

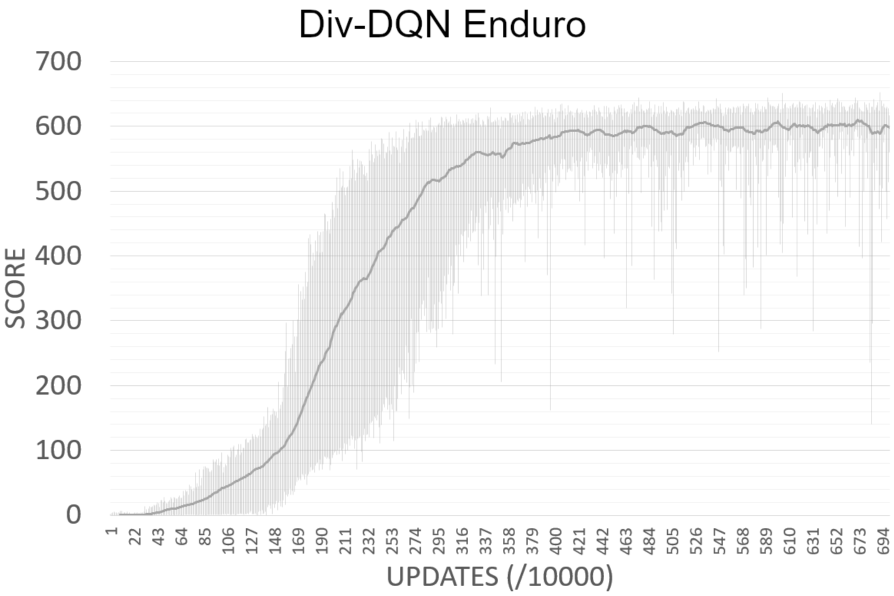

Figure C.21: Individual results for Div-DQN in Enduro

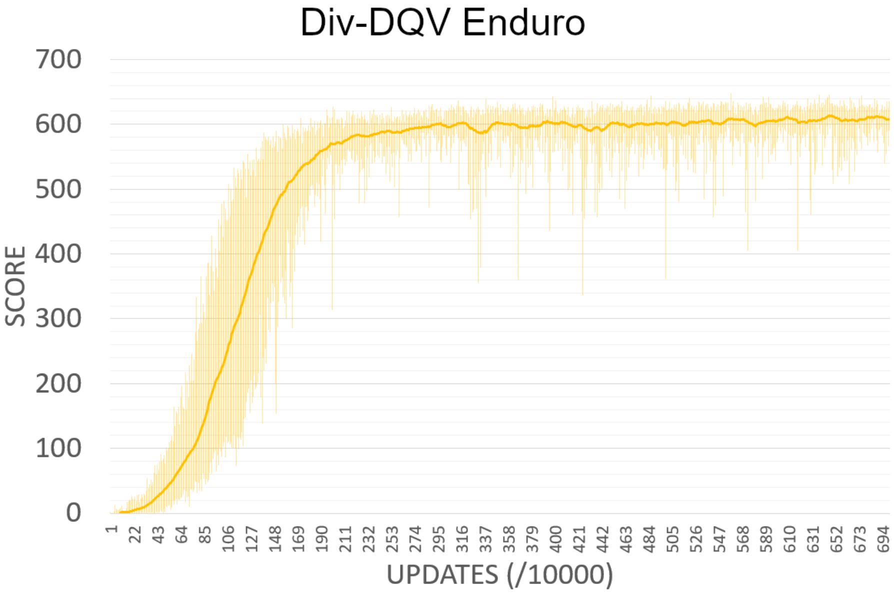

Figure C.22: Individual results for Div-DQV in Enduro

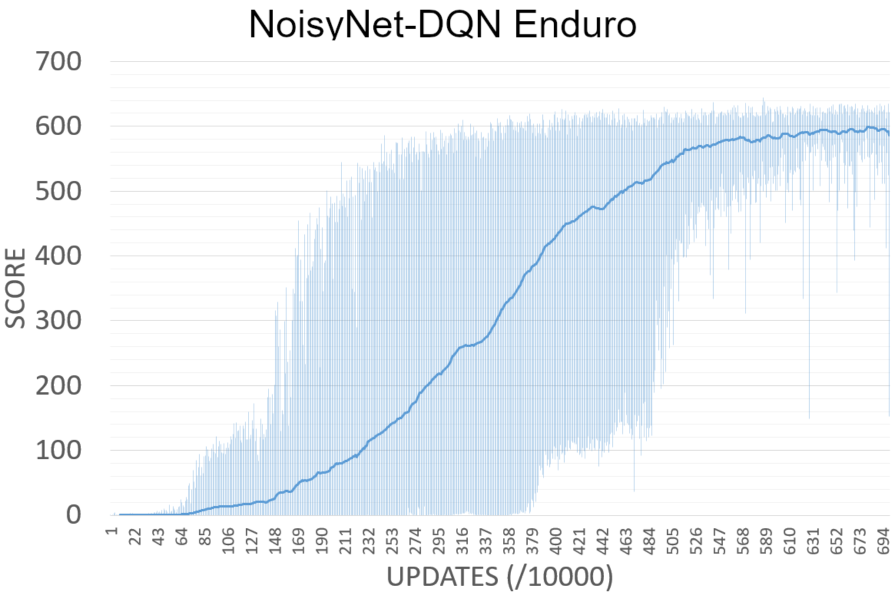

Figure C.23: Individual results for NoisyNet-DQN in Enduro

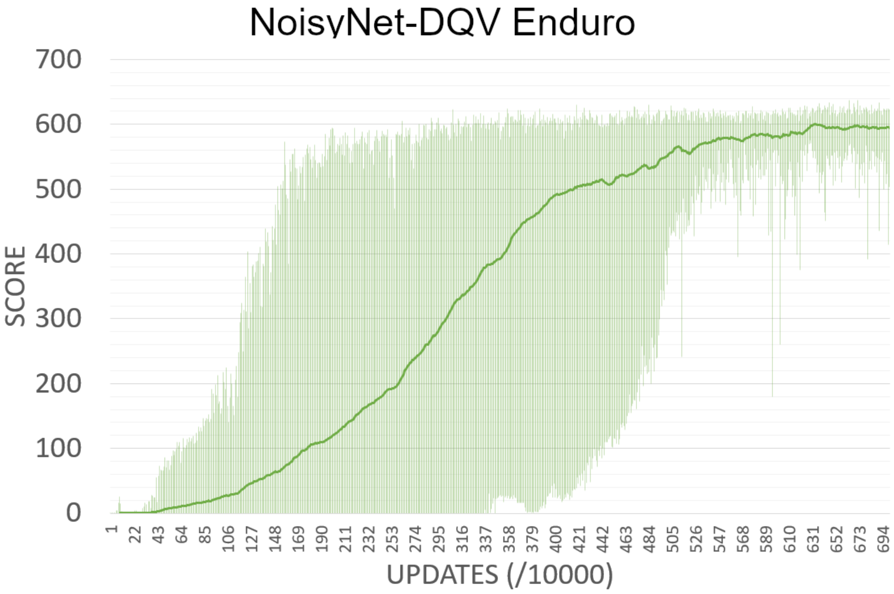

Figure C.24: Individual results for NoisyNet-DQV in Enduro

D Ranking of Algorithms

This appendix contains the final ranking of how the algorithms performed across all four games compared to each other.

The ranking happens as follows. Firstly, it is assumed that all algorithms performed equally well on each game. Given this null hypothesis, initially each algorithm gets the ranking interval 1 – 6 assigned for all games. In the next step, a non-parametric significance test, the Wilcoxon rank-sum test, is applied to test whether the final test scores obtained by any two algorithms on their ten training runs per game differ significantly or not. A non-parametric test was chosen since not all obtained test scores were normally distributed. Whenever the significance test indicates that the final test scores obtained by the two currently considered algorithms differ significantly given the same game, with p = 0.05, the top rank position of the algorithm associated with the lower of the two mean final test scores worsens by one position, i.e. the top rank position value gets incremented by one, while the lowest possible rank for the currently considered algorithm with the higher average score decreases by one. This happens for each of the four games individually. The result of that step is shown in the table below in the columns *Breakout* through *Enduro*. Afterwards, ranks obtained on the four games get summed to a total score per algorithm, where each rank interval is represented by its mid-value. These results are shown in the column *Total Score* in the table below. Finally, the algorithms are ranked with respect to their total scores computed in the previous step. The result is shown in the column *Total Rank* in the table below. The lower an algorithm's absolute value of its total rank, i.e. the higher the rank, the better the performance of the algorithm with respect to the remaining algorithms. Below, the table performing this ranking is shown.

Table D.1: Ranking of algorithms across the four games Breakout, Centipede, Q*Bert, and Enduro. *Total Score* indicates the sum of ranks obtained by the algorithm. *Total Rank* indicates the overall rank obtained by an algorithm across the four games. Rank 1 indicates the top rank among all competing algorithms.

Algorithm	**Breakout**	**Q*Bert**	**Centipede**	**Enduro**	**Total Score**	**Total Rank**
DQN	1	6	2 – 6	5 – 6	16.5	4
DQV	2 – 5	1 – 5	2 – 6	2 – 5	14	2
Div-DQN	3 – 6	1 – 5	1 – 6	1 – 5	14	2
Div-DQV	4 – 6	1 – 5	2 – 6	1 – 4	14.5	3
NoisyNet-DQN	3 – 6	1 – 5	1 – 2	1 – 6	12.5	1
NoisyNet-DQV	2 – 3	1 – 5	2 – 6	1 – 5	12.5	1