# Hooked on Phoenix: Deep Q-Learning on the Classic Atari Game

**Evan Darke** [1] **Jake Smola** [1] **Michael Mernagh** [1]

## Abstract

This paper investigates a variety of deep architectures that can be used to play Phoenix, a classic Atari game, using only raw pixel values as input. The deep networks are trained to estimate the value of each possible action for screenshots seen while playing Phoenix. The network estimates are used by a reinforcement learning agent that both chooses actions based on the estimate, and thus generates new inputs as it the learned estimates improve. We also examine using autoencoders to compress screenshots as input for deep Q-Learning. The agents trained on these estimates exceed human performance.

## 1. Introduction

Phoenix is a classic arcade game where the player controls a vehicle that moves left and right across the bottom of the screen. The player shoots upward at alien birds which return fire and dive-bomb the player. The player can also activate a shield for a short period every 5 seconds. The player fights four waves of enemies which employ different movement patterns and attack speeds. Finally, the player faces a boss battle against an alien mothership, after which the waves repeat with a higher difficulty. The player is rewarded for each enemy they kill, with the reward dependent on factors such as whether the bird was shot while dive-bombing the player.

The boss battle varies significantly from the other levels. A mostly-stationary alien ship fires down on the player, who must shoot up through a protective stratum, and then must carve a hole through a rotating shield before being able to kill the alien pilot. Doing so yields a substantial reward.


*Figure 1.* Player using shield while firing at enemies

We trained an agent to navigate through Phoenix's levels using only raw visual sensory data as input and learning through its own experience without expert derived reward functions. The reinforcement learning techniques we explore are widely applicable to many real world tasks, such as autonomous helicopter flight, which are much more expensive to simulate than Atari games.

To estimate reward, we modified DeepMind's Convolutional Neural Network as part of our Deep Q-learning model. We further revised this model to incorporate a dueling architecture as well as double Q-learning. To improve storage requirements and compute latency, we also developed an autoencoder. However, our best results were achieved by training on raw pixel data. The dueling and double Q-learning models were able to achieve superhuman performance. Our best model is able to excel through the initial swarms of alien birds, but struggles with the more complex boss battles.

## 2. Background / Related Work

Atari games have been used as benchmarks for reinforcement learning, both because they are easily simulatable, and because their complexity is a suitable test for complex policies.

A notable paper was the original Deep Q Network paper by [6], which showed that the Q-function can be reliably approximated by a deep network, and introduced experience replay as a method for training on mostly independently gathered samples from a highly correlated environment. It also introduced a mechanism for stabilizing the policy by maintaining two approximators, one for estimating the value of the current state-action pair, and the other for estimating the value of future state-action pairs, where the two approximators are synchronized periodically. While the DQN results showed improvements on a large number of Atari games, results for Phoenix were not included in this paper.

[8] showed that a naive Q-learner suffers from maximization bias. In other words, the model yields an overoptimistic estimate of the maximum Q-value which degrades performance. [2] presented an efficient adaptation to Deep Q Networks that incorporates the Double Q-learning algorithm which often leads to better policies.

DeepMind subsequently released a paper introducing the Dueling Architecture [9]. In this paper, they proposed creating separate networks for approximating the Q-value of a state, and for approximating the advantage for each possible action from that state. By combining these functions, the dueling architecture can quickly learn a more accurate state value function, especially when a number of actions are redundant for a given state. This architecture can be combined with other deep Q-learning models, since in practice it involves reusing parameters from the lower convolutional layers of a deep architecture, splitting their activation into separate streams, and then recombining

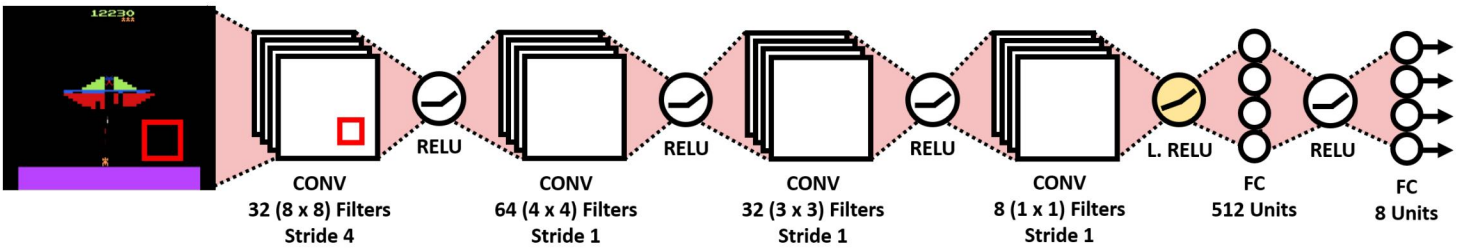


*Figure 2.* Double Q-Learning Architecture.

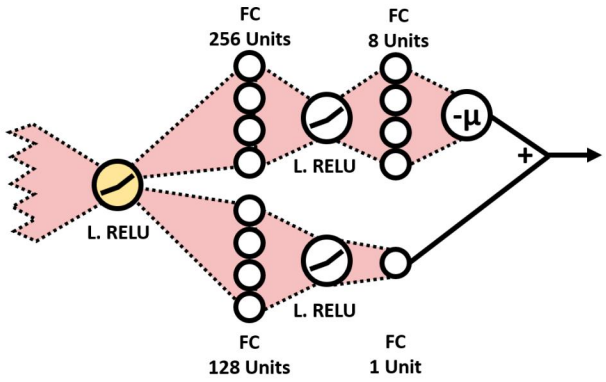


*Figure 3.* Dueling Architecture. The first leaky ReLU activation unit is shaded to signify where the architecture begins relative to the entire double q-learning architecture.

the streams for a unified output.

Additionally, autoencoding image representations using neural networks has been explored thoroughly [1, 3], and have been shown to be able to efficiently represent image data. Within the domain of reinforcement learning from images, the use of autoencoders has also been explored generically by [4].

## 3. Dataset

As part of the reinforcement learning of our model, data is generated by observations that the agent sees while exploring. Each observation is a (210 x 160) RGB image. Our network takes an observation as input, and outputs an estimate for the total future rewards that can be obtained from taking each of the eight possible agent actions from this state. Updates to the network incorporate scalar rewards that are obtained by taking a particular action from a state plus the estimated value of discounted future rewards from the subsequent state.

### 3.1. Baselines

We compare the performance of our learning algorithm to that of a random agent and amateur human player by assessing the mean final score each agent achieves in one episode. The random agent reward was computed by running 1,000 Monte Carlo rollouts. The DQN was evaluated as the mean reward over 50 episodes using a $\epsilon$-greedy policy with $\epsilon = .05$ after 10,000,000 training iterations. Both the random agent and the DQN were constrained to selecting actions once every four frames as was done in [6]. The amateur human reward was computed as the mean of three trials from two players for a total of six trials. The table below shows the average reward for all agents.

| Agent | Avg Reward |
|---|---|
| Random | 460 |
| DQN | 3250 |
| Human | 3875 |

*Table 1.* Agent performance comparison.

## 4. Approach

### 4.1. Preprocessing

We preprocess each frame by converting it to greyscale and downsampling the height and width by two. This reduces the size of the state representation by a factor of $\frac{1}{12}$, allowing for faster training and predictions at a potential accuracy cost.

### 4.2. Autoencoder

The size of state representation is very large to be visually pleasing to human players, but it is also exceptionally sparse. Therefore, we hypothesized that we could speed up learning by using an autoencoder to reduce the size of the state representation. We designed a deep convolutional autoencoder that compresses a frame from shape (210, 160, 3) to (13, 20, 1) for a total compression ratio of $\frac{1}{384}$. The autoencoder consists of alternating convolutional and pooling layers which are then upsampled and convolved to reconstruct the original input with logistic loss applied to each component of each output pixel. A sketch of the autoencoder architecture, inputs, and outputs is shown in Figure 4.

We replaced the preprocessing step in our original algorithm with invoking our autoencoder and storing the resulting low dimensional representation in the replay buffer. Storing low-dimensional state representations reduces the latency of the q-network computations, and more importantly, greatly multiplies the capacity of the replay buffer, which improves sample efficiency. The autoencoder embeddings are feed into a modification of our dueling architecture without the first 3 convolutional layers.

### 4.3. Deep Q-Learning

In training our model, we implement a fairly complex convolutional neural network. Work done in [6] has demonstrated the difficulty of successfully training agents to play various two-dimensional games. Given the comparable complexity of Phoenix, we begin with a sim-

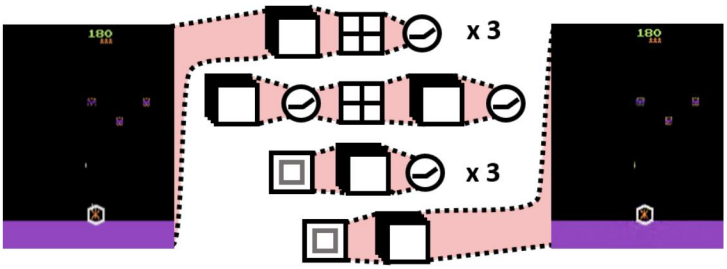


*Figure 4.* Autoencoder Architecture. Layers consist of: 3 × (Convolution, Average Pooling, Parameterized ReLU), (Convolution, Parameterized ReLU, Average Pooling, Convolution, Parameterized ReLU), 3 × (Up-sampling, Convolution, Parameterized ReLU), (Up-sampling, Convolution).

ilar implementation to improve our ability to iterate in successive experiments.

Learning target values are generated by bootstrapping our Q-value estimates using our network with a previous set of weights $\theta^-$ which has learned the function $\hat{Q}^-$, whereas $\hat{Q}$ will denote the Q function estimated by the current set of weights.

Give a tuple of the current state $s_t$, the action taken $a$, the next state $s_{t+1}$, and the immediate reward for the action $r$, we compute the target value $y$ as follows:

$$y^{(i)} = r^{(i)} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}^-(s_{t+1}^{(i)}, a'; \theta^-)$$

Our network will minimize the squared loss over a batch B

$$J(\theta) = \frac{1}{|B|} \sum_{i=1}^{|B|} (\hat{Q}(s^{(i)}, a^{(i)}; \theta) - y^{(i)})^2$$

The full architecture of our DQN is available in figure 2.

### 4.4. Double Q-Learning

As described in [2], we implemented a step toward double Q-learning to mitigate maximization bias which normal Q-learning suffers from. This requires a small change to our loss function where the action that maximizes future rewards is chosen by the target network and evaluated by primary network. Since the target network uses a stale version of the online network's weights, the choice of action and its evaluation is only partially decoupled.

Below is the objective function computed in double Q-learning:

$$y^{(i)} = r^{(i)} + \gamma \hat{Q}^-(s_{t+1}^{(i)}, \operatorname*{argmax}_{a' \in \mathcal{A}} \hat{Q}(s^{(i)}, a'; \theta); \theta^-)$$

In our implementation, $\hat{Q}(s, a, \theta)$ represents the predicted Q-value of online network while $\hat{Q}^-(s, a, \theta^-)$ represents that of the target network. Consequently, during each iteration, the online network is used to select the action which is evaluated on the target network. However, in traditional double Q-learning, these representations are not fixed; at every iteration, one of the two networks is chosen at random to select the best action before the other network is evaluated. Though this method is likely to ensure better generalization, it sacrifices computational time and demands overhead outside of the scope of the simpler deep Q-learning model.

Apart from the loss function, which is a function of the aforementioned target, our double Q-learning model closely relates to the standard deep Q-learning model. As before, the target network is still occasionally updated with weights from the online network.

### 4.5. Dueling Networks

We also created a dueling network, in order to speed up learning for states where learning all the advantages is unnecessary, as described by [9]. To do so, we adopted the split stream approach, and combined the estimate of the state value (the function $V$ parameterized by $\beta$) with the estimated advantages (the function $A$ parameterized by $\alpha$). To ensure that the functions $V$ and $A$ are uniquely identifiable for a given $Q$ value, we subtract the mean over the advantage function.

$$Q(s, a; \alpha, \beta) = V(s; \beta) + A(s, a; \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \alpha)$$

This ensures that the estimate is not only identifiable, but is also stable as the advantage function is updated during training.

A depiction of our dueling architecture is available in figure 3.

### 4.6. Experiments

We train and test our agent using the Phoenix-v0 environment for OpenAI gym. A screenshot of gameplay is available in Figure 1. Every four timesteps, the environment sends a snapshot of the latest pixel values to the agent. The agent feeds the last four frames into a convolutional neural network to estimate Q values of each of the eight valid actions that the agent can perform. During training, the agent chooses the action with the highest Q value with probability $1 - \epsilon$ where $\epsilon = max(1 - \frac{iterations}{max\_iterations}, .10)$, and a random action otherwise. The $\epsilon$ hyperparameter decays during training to transition the model from more stochastic exploration in the early stages of training (when the Q-values are poorly approximated) to exploiting learned dynamics in later training stages (when the Q-values are closer to the ground truth).

Mini-batch inputs to the network are randomly sampled from of buffer of observations that the agent has seen. The size of the buffer is a hyperparameter. Using a buffer reduces the variance between mini-batches, as is shown in [5].

Updates to the network are made by bootstrapping using sample observations and rewards obtained by the learning agent. Specifically, we set the loss to be the squared loss comparing $\hat{y}$ (the output from the network) and sampled rewards (where the sampled rewards also incorporate a stable estimate of the discounted value of the states that will be visited in the future). This stable estimate is updated every so often as a copy of the current network, according to the `target_update_freq` hyperparameter.

*Table 2.* Results

| Agent | Avg Score | Uncertainty |
|---|---|---|
| DQN | 3404 | ± 211 |
| Double DQN | 4051 | ± 229 |
| Dueling DQN | 4111 | ± 205 |
| Double Dueling DQN | 3579 | ± 227 |

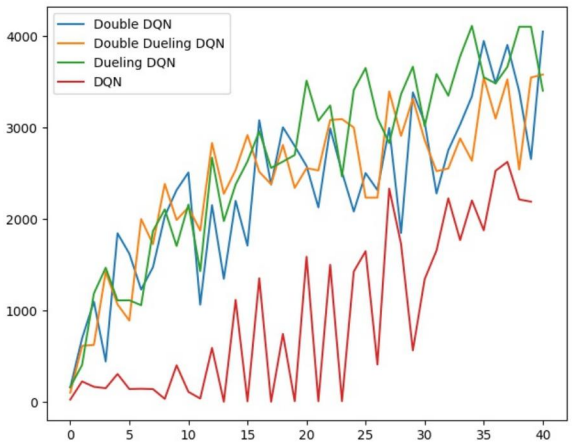


*Figure 5.* DQN Phoenix scores observed during training.

Table 2 shows the maximum average reward obtain by our four networks during evaluation. Notably, although both double DQN and dueling DQN significantly improved performance, combining the two improvements showed a performance regression. However, our double dueling network did show faster and more stable convergence than our ordinary DQN, shown in figure 5.

*Table 3.* Comparison of Loss Functions

| Agent | Avg Score |
|---|---|
| Huber | 1580 |
| Huber w/ Clipped Rewards | 2754 |
| **Squared** | **4201** |
| Squared w/ Clipped Rewards | 2725 |

$$L_{Huber}(y, \hat{y}) = \begin{cases} \frac{1}{2}[y - \hat{y}]^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta\left(|y - \hat{y}| - \delta/2\right) & \text{otherwise.} \end{cases}$$

In [6], the authors found that using the Huber loss, defined above, and clipping rewards to be in the range $[-1, 1]$ improved convergence in five Atari games by limiting the magnitude of gradients. However, clipping rewards has the disadvantage that the network is unable to distinguish between some actions which yield positive rewards of different magnitudes. Instead, we chose to clip our gradients by the global norm, which maintains the original direction of the gradient, and compared Huber and squared loss with and without reward clipping on Phoenix. Table 3 shows that, in our setting, squared loss without reward clipping is superior to other loss functions.

#### 4.6.1. Final Agent

After finalizing our architecture and identifying dueling DQN as our most promising model, we trained a final agent longer and with more exploration. With a buffer size of 2M, and reducing epsilon decay by a factor of 5, we trained another agent for 21M iterations.

*Table 4.* Final Results

| Avg Score | Uncertainty | Max Score |
|---|---|---|
| 4423 | $\pm$ 186 | 6560 |

#### 4.6.2. Autoencoder

The encoder network consists of 4 convolutional layers, using filters of size 4 for the first layer, and 3 for the subsequent layers. Each layer is immediately followed by an average pooling layer with a $2 \times 2$ filter. Parametric ReLU is applied after every convolutional layer. We collection 200k examples of game frames randomly sampled from episodes and created an 80/10/10 train/dev/test split.

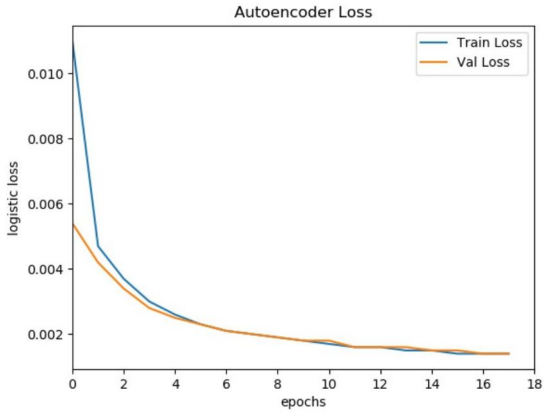


*Figure 6.* Autoencoder train and dev loss

Figure 6 shows the loss on the training and development set for our autoencoder. Notably, our autoencoder shows no overfitting because we were able to sample an arbitrarily large training set from our environment. We achieved a final test loss .0839, which matches our development loss.

Figure 4 shows the architecutre of our autoencoder along with an example of an input and a reconstructed output. This demonstrates that the autoencoder correctly encodes the player's position, state of the player's shield, enemy positions, and laser fire. Notably, it also encodes information which is useless for our task, such as the current score.

Using our autoencoder embeddings to train a DQN achieved an average score of 3143 when trained with the same hyperparameters as our networks. Training on raw pixels gives superior performance, but the autoencoder does have the advantage of using 97% less RAM. We suspect the autoencoder underperforms training directly on raw pixels because the autoencoder is trained to weigh all information equally when learning an encoding, whereas a convolutional network learns to extract just the information required for its task. Additionally, the autoencoder was only trained on a dataset with 200k frames, compared to the DQN which was trained on 10M.

#### 4.6.3. DQN

We slightly modified the network proposed by [5], principally by halving the number of filters in the last convolutional layer, and also introducing a subsequent 1x1 convolutional layer to reduce the dimensionality of the input to the fully connected layer. We

used a total of 4 convolutional layers, with filter sizes of 8, 4, 3, 1 respectively. The number of output channels for each layer was 32, 64, 32, 8 respectively, and we used strides of 1 for each layer, except for the first, which had a stride of 4. Note that the final convolutional layer uses a small number of channels. Since the state image contains a small number of features distinguishable to a human, it is reasonable to reduce the size of the output. The convolutional layers are followed by two fully connected layers, of size 512 and 8. We used relu as our non-linear activation.

#### 4.6.4. Dueling Networks

The streams of the dueling network diverged after the final convolutional layer of the base DQN. The state value function used 2 fully connected layers of size 128 and 1, while the advantage function used 2 fully connected layers of 258 and 8 respectively.

#### 4.6.5. Double Q-Learning

The double Q-learning model duplicated either the standard deep Q-learning architecture or the dueling architecture in defining . While the double Q-learning model was able to improve significantly upon the average scores of the simple DQN architecture, the combination of double Q-learning and dueling networks did not yield expected results; the double dueling model underperformed the dueling model. We discuss this phenomenon in section 4.8.

### 4.7. Hyperparameters

To begin our experimental cycle, we initiated our configuration with the hyperparameters from [5]. We increased the number of training steps from 1,000,000 to 10,000,000, but kept the remaining hyperparameters, including batch size, replay buffer size, target update frequency, learning rate, epsilon and others identical.

### 4.8. Performance Analysis

#### 4.8.1. Agent Challenges

All of our final networks perform significantly better than the random agent, and most achieve superhuman performance. Reviewing the recordings of random evaluations, we identified two key challenges of the agents: shield usage and the boss battle. As the player is only able to activate their shield every five seconds and there is no visual indicator of when the shield is available, the agent is unable to learn to use the shield effectively. As a result, the agent occasionally sits still as it is hit by a single bullet, which should be easily avoidable.

The agent also struggles with the boss battle described in section 1. The agent must learn to fire at a specific point on the mothership, break through a rotating shield, and finally shoot the alien pilot. All the while, the agent must continually dodge enemy fire. There are no intermediate rewards for damaging the mothership. At no point in training did the agent successfully defeat the boss, and therefore had no knowledge any eventual rewards. As a result, when confronted with the mothership boss, the agent simply avoids fire by hiding in the corner.

On a positive note, it is evident that the agent recognizes there is short cool-down period associated with firing and thus the importance of timing its shots to maximize impact and minimize game time. Neither of our amateur human players recognized that there was a cool-down period associated with firing during their initial playthroughs.

#### 4.8.2. Model Performance

[9] demonstrated results fairly consistent with our own, in which DQN underperforms Double DQN which underperforms dueling networks in the game of Phoenix. Perhaps the greatest surprise we have encountered concerns the performance of the double Q-learning model with dueling network architecture. As seen in table 2 and 5, this particular model performs only slightly better than standard DQN after sufficient training. This is unexpected, since the double Q-learning and dueling models perform very well independently and all models we deployed the same weight update subroutines between the online and target networks.

As mentioned in section 4.4, the only change our double Q-learning model incorporates over the standard DQN model concerns the objective passed as input to the loss function. Furthermore, the only difference between the standard DQN and dueling architectures concerns the last few layers. Because the Dueling and Double DQN models performed very well, we hypothesize that the relatively small size of the fully connected layers in the dueling architecture biases the model enough that it doesn't suffer as much from the maximization issues that Double Q learning addresses. Furthermore, we feel that more thorough hyperparameter tuning for the double dueling DQN model could perhaps yield more fruitful scores.

## 5. Conclusion

The networks we developed are capable of estimating the value function well enough for the agents to play considerably well. We showed that squared loss with gradient clipping outperforms Huber loss and reward clipping. Finally, we showed that autoencoders can be used to implement DQNs using much less RAM at a small cost to performance. Nonetheless, we believe our models are not performing to their maximum potential. We think that even more comprehensive hyperparameter search could potentially yield even higher results. Additionally, we suspect that prioritized experience replay, as proposed in [7], should improve the stability and speed of learning. To improve double dueling DQN performance, we believe a narrower analysis of the interaction between loss and the dueling architecture weights (and potentially early layer weights) may also uncover insights useful in improving the model or uncovering any causes of its weaker performance.

We also identified dynamics of Phoenix that pose difficulty for deep RL: the agents ability to use its shield is not observable in the state space and that the boss battle differs significantly from previous levels with highly delayed returns that make it difficult for a decaying epsilon policy to find a winning strategy. We think that an exploration strategy where $\epsilon$ is a function that decreases with iterations but increases with score (as an approximation for level progress) may enable the agent to explore again when it moves to the boss battle.

## 6. Contributions

Evan: Implemented Autoencoder, dueling network, random agent, novel exploration strategy, hyperparameter tuning.
Jake: Implemented double Q-learning, composed architecture graphics, tested/modified exploration strategy, hyperparameter tuning.
Michael: Implemented dueling network, brainstormed exploration strategies, hyperparameter tuning.

## 7. Course-related Components

### 7.1. CS234: Reinforcement Learning

- Deep Q-Learning and Double Q-Learning models
- Exploration Strategies
- Reinforcement learning framework and agent
- Selection of RL learning function

### 7.2. CS230: Deep Learning

- Autoencoder architecture and experiments
- Hidden layers of DQN
- Hidden layers of Dueling network
- Comparing a variety of loss functions

## 8. Source Code

Available: https://github.com/Edarke/Phoenix-DQN

## References

[1] Yoshua Bengio et al. "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1 (2009), pp. 1–127.

[2] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[3] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507.

[4] Sascha Lange and Martin Riedmiller. "Deep auto-encoder neural networks in reinforcement learning". In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE. 2010, pp. 1–8.

[5] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[6] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[7] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2nd ed. Vol. 1. 2. MIT press Cambridge, 2018.

[9] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.