Understanding Policies of Reinforcement Learning Agents Playing the Atari Game Pong

Galip Ümit Yolcu, Dennis Weiss, Egemen Okur

July 2021

Abstract

With the growing complexity of Deep Learning algorithms, it has become difficult to comprehend the decisions of agents. There have been many successes of using deep representations in Reinforcement Learning. In this paper we investigate a Deep Reinforcement Learning agent with respect to how the agent takes actions and what part of the input it attends to. While using a Dueling Neural Network architecture [5] for the game Pong, we used saliency maps and game trees to visualize an agent's actions and decisions. Saliency information has been implemented with Gaussian blur perturbation and in conjunction with our results gathered from the game tree, we contributed to understand how Reinforcement Learning agents make decisions.[1]

1 Introduction

Reinforcement Learning (RL) agents, which learn autonomously without any human intervention, have achieved state-of-the-art results in Atari games and have proven to be effective at maximizing rewards. Despite their impressive performance, they have been a black box, as the learned weights of a complex model give no immediate insights of the agent's strategies. Understanding an agent's actions and policies have also been important for Machine Learning experts to interpret their models before using them to solve real-world problems. This transparency need can only be met when a model is able to provide justifications with regard to its predictions and decision-making.

In this paper, we investigate a Deep RL agent which uses raw visual input to make its decisions. Our main contribution is to conduct investigative explorations into explaining Atari agents. In particular, we focus on exploring the utility of visual saliency and on visualizing the game tree of our agent to gain insight into the decisions made by these agents.

The demonstration of the saliency approach allows non-experts to detect where the agent's region of interest is, whereas the game tree allows one to understand why the agent is taking an action by showing all possible actions with their corresponding Q-value.

Game trees are directed graphs consisting of nodes which are positions (or states in our case) in a game and edges are the possible actions. This provides the advantage of showing which choice of action can lead to which state and therefore increases the explainability of the agent. To our knowledge, there has not been any game tree visualization of an Atari game.

The Dueling Neural Network Architecture has been used, since an advantage of this architecture is that it separates the estimation of the state value and the relative advantages of each action which allows us to visualize both outputs and increase transparency with the

[1] A presentation of the results are available on `xrl-nip.herokuapp.com` Due to big dependencies and computational complexity, we couldn't run real time visualisations on the hosed website. The code is reachable from `https://github.com/DennisWeiss/pong-deep-q-network-explainability`.

use of saliency maps. In the following parts, we will explain our model and our methodologies.

2 Model

The Dueling Neural Network Architecture was trained with the DDQN algorithm with experience replay as explained in [5]. The key advantage of this architecture is that it does not estimate the value of all the action choices but keeps a separate estimation of the state value and the advantages for each action. This is helpful in states where the action taken does not have a relevant effect to the environment.

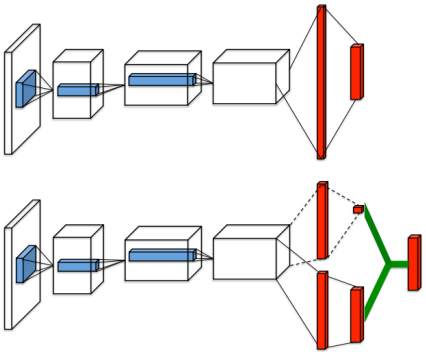

Figure 1: Adapted from [5]. On top a traditional Q-network is shown. On the bottom a dueling network with 2 streams (advantage and state-value) is shown.

The model takes 4 grayscale images as input. These are preprocessed versions of the last 4 consecutive frames from the Atari environment. These images, denoted by s for the remainder of this text, define the states and the network outputs Q-values for the 6 actions in the Atari Learning Environment (ALE).

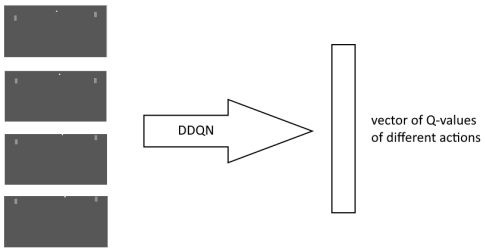

Figure 2: The input is made out of 4 grayscale images and the output is the vector of Q-values.

The lower layers of the Dueling Neural Network are convolutional with ReLU activation function as in original DQNs. But instead of following the convolutional layers with a single sequence of fully connected layers, it uses two streams - value function stream and advantage function stream - to separately estimate the value of a state and the advantages of each action.

The outputs of the advantage stream $A(s)$ and the value stream $V(s)$ are then combined via an aggregating layer producing an estimate of the state-action value function $Q_a(s)$:

$$Q_a(s) = V(s) + A_a(s) - \frac{1}{|\mathcal{A}|} \sum_{a\prime \in \mathcal{A}} A_{a\prime}(s) \tag{1}$$

The set of possible actions is denoted by $\mathcal{A}$. We refer to the paper [5] for a detailed description of the model's hyperparameters.

We chose this model architecture for the following reasons:

- The model defines different choices of meaningful values which we may be interested in visualizing while investigating how the agent works: namely, $V(s)$, $A(s)$ and $Q(s)$. Looking at these values does indeed give different kinds of explanations of the agent according to the results in the original paper.
- The model was tested in the Atari environment and was shown to be effective.
- We used a repository including simple, ready-to-analyze code and trained agent on GitHub **original-repo**, which we based all our visualizations on.

3 Methods of explainability

3.1 Game tree visualization

In Q-Learning the Q-value is an estimate of the future discounted reward of the underlying Markov Decision Process (MDP). The future discounted reward $V_i(s)$ at time step i in state s can be stated recursively as follows:

$$V_i(s) = \max_{a \in \mathcal{A}} \underbrace{\mathbb{E}_{\mathbb{P}(s'|s,a)} \left[R(s'|s,a) + \gamma V_{i+1}(s') \right]}_{V_i(s,a)} \tag{2}$$

$\mathbb{P}(s'|s,a)$ denotes the probability of transitioning from state s to state s' when choosing action a. $R(s'|s,a)$ denotes the corresponding reward of the MDP. $\gamma \in (0,1)$ is the discount factor. For a deterministic environment, like the Pong environment we were dealing with, the expression can be simplified to:

$$V_i(s) = \max_{a \in \mathcal{A}} \underbrace{R(s'|s,a) + \gamma V_{i+1}(s')}_{V_i(s,a)} \tag{3}$$

$V_i(s,a)$ is a numeric measure for the goodness of taking action a and being in state s. One can observe the following structure for $V_i(s,a)$:

$$V_i(s,a) = R(s'|s,a) + \gamma \max_{a \in \mathcal{A}} V_{i+1}(s',a) \tag{4}$$

Q-Learning will let the values of $Q(s,a)$ converge to $V_i(s,a)$. However this will only provide a measure on the quality of state-action combinations, but will not give a direct insight into why an action is advantageous or why not and which actions will be chosen in the consecutive steps. By replacing $V_i(s,a)$ by the approximations $Q(s,a)$ we observe:

$$Q(s,a) = R(s'|s,a) + \gamma \max_{a \in \mathcal{A}} Q(s',a) \tag{5}$$

Given that the agent is in state s, we only need to know the Q-values of possible actions in the state of the next timestep s' (and the immediate reward of the MDP) to reconstruct the computation of the Q-value. Intuitively, a Q-value will be high if there is a high Q-value in the next state for at least one action and the Q-value will be low if all Q-values of the next state are low.

3.1.1 Branching on taken actions

For visualization purposes it is not feasible to display the entire game tree, even if the number of steps n we look ahead is rather low. The number of nodes in the last layer of that tree is $|\mathcal{A}|^n$ and thus grows exponentially with the number of steps with exponential basis equal to the number of possible actions. Furthermore, doing an exhaustive search is also not feasible from a computational standpoint, as for each node of the game tree the step of the game needs to be simulated and the model computing the Q-values needs to be run. Instead of doing an exhaustive search through the entire game tree, we only branch on the action that is taken in the given timestep and the computational cost is then linear with respect to the number of steps we look ahead.

3.1.2 Exhaustive search with best strategies displayed

In the second version of the game tree visualization we did a search through the entire game tree for the next n steps and compared the Q-value after n steps of all $|\mathcal{A}|^n$ steps. Then we only display the paths of the tree that lead to the best k strategies. A typical value for k to chose is 10. As there are k leaves of the tree, the width of the tree is at most k and thus suitable for visualization for a small enough value k. The runnning time grows exponentially w.r.t. n, but as long as n is chosen small, the game tree is still computable in short time.

3.1.3 Bounding the search tree

Instead of doing an exhaustive search and branching on every possible action, the computation can be sped up by only branching on actions with Q-values that are high enough. We define a Q-value to be high enough its softmax value is within a certain ratio of the softmax of the maximal Q-value among the possible actions. More formally, we branch on action a if the following condition is met:

$$\frac{Q(s,a)}{\sum_{a' \in \mathcal{A}} Q(s,a')} \geqslant \delta \cdot \frac{\max_{a' \in \mathcal{A}} Q(s,a')}{\sum_{a' \in \mathcal{A}} Q(s,a')}$$

The factor $\delta \in (0,1)$ determines how restrictive the branching is. A higher value of δ leads to less computation but higher risk of missing good strategies and a lower value of δ branches more often, but the chance of missing a strategy with a high Q-value shrinks. The reason why we can skip branching on actions with low Q-values can be seen in equation 5. If an action has a low Q-value, then with high probability all Q-values of the next timestep are going to have low Q-values. But as we are interested in finding paths with high Q-values in the final state, it is not worth to invest the computation time to further explore this part of the game tree.

3.2 Saliency maps

We have investigated different methods to visualize the importance of the input images in the outputs of the neural network. These methods aim to visualize the *salient* pixels in the input image, pixels which matter more, which have a significant effect in the agent's decision for the next action. Hence, they are named saliency maps. The basic idea in all these methods is to select a scalar value (an output neuron's activation) or a vector (the whole output of the model) and investigate how changing the input pixels effects this value. In our setting, since we will get a value for each pixel in the input images, we can visualize these values easily by putting them over the grayscale input images as a separate RGB channel.

Since our choice of model architecture offers different meaningful vectors to investigate, we will use $F(s)$ to denote whatever scalar or vector we are interested in using for the saliency map, while explaining the different methods.

We implemented 4 methods, in 2 groups. The first group are the *gradient based methods*. The most basic one is **vanilla backpropagation** and we also implemented a refinement of this: **guided backpropagation**. The other two methods are *perturbation based methods* which focus on perturbing the image in different ways to compute the importance of regions in the image.

3.2.1 Vanilla backpropagation

Gradient based methods use the gradient of a neuron's activation with respect to the input pixels evaluated at a given input, in order to measure the relevance/importance of each pixel in the activation of the neuron. The intuition is that the gradient tells us how much the output of the neuron changes for small changes in the pixel values, and thus gives a measure of importance with respect to the neuron's activation.

These methods were originally proposed in order to explain image classification models, computing saliency maps for each output class (for neurons at the last layer) [2], but have also been applied to the Atari setting [5]. We have computed maps by setting $F(s)$ to be the value estimate $V(s)$, an element of the advantage estimate vector $A(s)$ or an element of the output logits $Q(s)$. Notice that since we are computing gradients with respect to a scalar, we can not let $F(s) = Q(s)$ or $F(s) = A(s)$ as opposed to the perturbation based methods.

The most basic method, which we name **vanilla backpropagation**, computes $|\nabla_s F(s)|$ to obtain the saliency map [2]. These values can then be normalized to lie in the interval [0,1] and shown on top of the input images as a separate channel [5].

Note that we have 4 frames, thus 4 images as input. This means we get 4 maps and one value for each pixel of each input image.

3.2.2 Guided backpropagation

The previous approach is related to another one, which we will call the deconvolutional method. This method tries to create a saliency map by reversing the computation of the model, via a deconvolutional neural network [1]. In short, after making a forward pass (noting which indices were selected at pooling layers, if there are any) we start the computation from the end of the computation graph, with a one-hot vector corresponding to the output neuron we are interested in. We traverse the graph backwards normally, but we transpose the convolution filters. In the case of max pooling layers, we use the indices that we noted in the forward pass as a mask and set other indices to zero in the computed tensors. This method is specifically designed for networks with ReLU activations after convolutional layers. We refer the reader to [1] for more details.

The vanilla backpropagation method and the deconvolutional method mainly differ in the way they handle ReLU functions. Guided backpropagation combines both approaches. Simply, while computing the gradient for an output, we apply the ReLU function to the gradients being computed whenever a ReLU function is found in the graph, effectively blocking the backwards flow of negative gradients. The main idea is that this prevents the flow of gradients for pixels which decrease the activation of the neuron we are interested in. We refer the reader to the paper for details of guided backpropagation and the relation between the three methods. We chose to implement this method because it produces sharper, significantly more interpretable images in image classification tasks compared to both vanilla backpropagation and deconvolutional methods [3].

$F(s)$ can have the same values as for the vanilla backpropagation method.

3.2.3 Occlusion perturbation

Another way to measure the importance of a pixel in the activation of a neuron or a layer, is to perturb the image by changing the pixel value and to use the difference in the activation to compute the pixel's importance. In fact, if the pixel values are changed by a very small

amount, the normalized intensity differences will give an approximation of the gradient map we would get by applying the vanilla backpropagation method. However, the gradient only contains information about local changes. With perturbation methods, we can perturb a group of pixels to measure the importance of an area in the image.

In practice, we perturbed all 4 frames at the same time, once for each pixel with a perturbation centered at it, because these methods already need a lot of computation time to compute the results. What we are measuring can be thought of as measuring the effect of an area of the game screen through time, instead of the effect of an area of the screen at a particular instant in time.

Let $F(s)$ again denote the value we are interested in. Let $s_{p_{ij}}$ denote the perturbed version of the state where the perturbation is centered at the pixel (i, j) of all frames, then the saliency value is given by $\left\|F(s) - F(s_{p_{ij}})\right\|_2$ [6]. Notice that $F(s)$ does not have to be a scalar anymore, it can be the whole vectors $A(s)$ or $Q(s)$.

Figure 3: Original game screen

Figure 4: Box-occluded game screen, where a square of the image has been replaced by the average pixel value of the screen

The most basic form of these methods, which we call **occlusion perturbation**, is the simple idea of setting a square region on the image to a fixed value and replacing that region by a gray square. The color of the square is set to be the average value of the input frames.

3.2.4 Gaussian blur perturbation

In [6] a different approach for Reinforcement Learning agents playing games in the ALE is proposed. It is argued that replacing regions in the image by an average value introduces new semantic information (like new edges or changes in the game area) and are not suitable for assessing the relative importances of the pixels. Instead, they suggest measuring sensitivity to noise. In this method, perturbed images are an interpolation between the original image and a blurred version of the image. For each pixel (i, j), the occluded version is obtained using a Gaussian mask M_{ij}, centered at that pixel. This mask has the same dimensionality as one of the input images and contains a value in $[0, 1]$ for each pixel. Then for each pixel, we do a linear interpolation between the original value and the blurred value using the mask for all 4 frames.

Let I denote the original image. I_{blur} denotes the image after applying a Gaussian blur to the image. The perturbed image $I_{p_{ij}}$ is computed by:

$$I_{p_{ij}} = M_{ij} \odot I_{blur} + (1 - M_{ij}) \odot I \tag{6}$$

where $\odot$ denotes the Hadamard (elementwise) product. The saliency is computed as in the previous occlusion perturbation method.

Figure 5: Original game screen I

Figure 6: Gaussian blurred game screen I_{blur}

Figure 7: Mask M

4 Results

4.1 Game tree visualization

The visualization through the game tree helped to investigate the model's behaviour and validate if the agent's intended actions make sense. The first thing we noticed is, that in a scenario, where the ball is moving away or far away from the player the Q-values are quite similar for the next few steps. This makes perfect sense, as the action taken now does basically not matter, as the ball is not about to hit the agent's side and thus the immediate action has no effect on the game score. But once the ball is reaching close to the agent's side, there tend to be certain actions that have a much higher Q-values than other actions. This can be seen quickly by the drawn widths of the edges of the game tree.

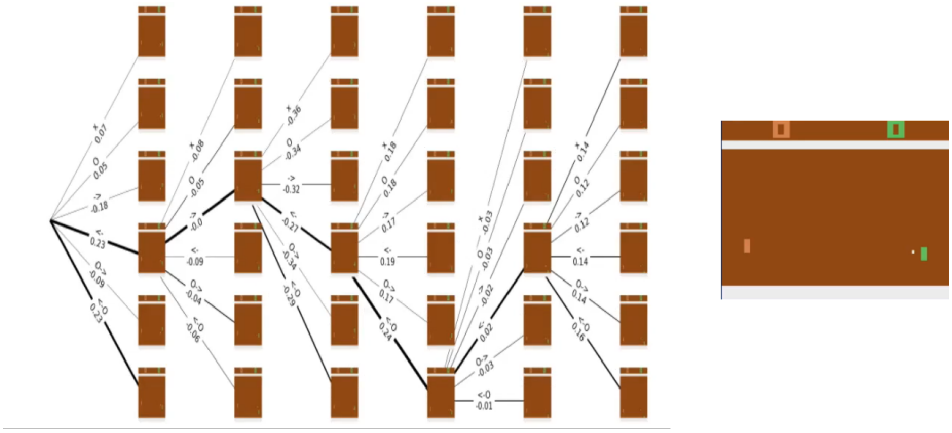

Figure 8: scenario, where the ball is approaching the player and there are certain actions that will succeed and thus yield a higher Q-value

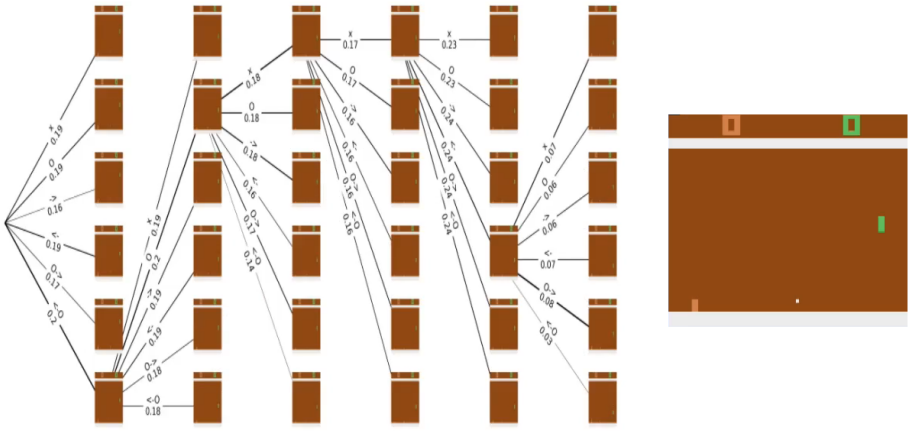

Figure 9: scenario, where the taken action does not influence the game in a way that will affect the score and thus the Q-values are similar throughout the different actions

Another observation - specific to this game - is that, vaguely speaking, the agent has learned that the FIRE action has no effect on the game. We conclude that by observing that LEFT and LEFTFIRE receive very similar Q-values by the model. The same applies for RIGHT, RIGHTFIRE and NOOP and FIRE, respectively.

One further notable observation can be made with the second version of the game tree, which shows the best n strategies among the next few steps. It can be seen that there are different combinations of actions that basically represent the same move in the game. One example would be, if the objective is to move 2 steps lower within the next 3 moves. LEFT

- NOOP - LEFT and NOOP - LEFT - LEFT are equivalent strategies and also yield very similar Q-values at the end, which validates the model.

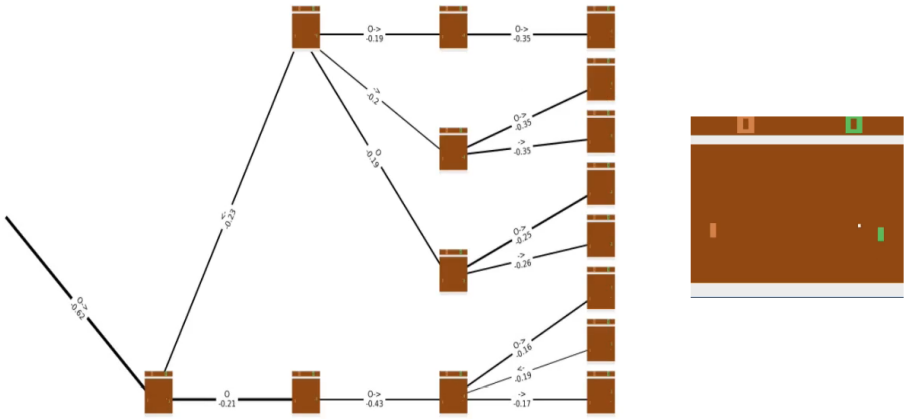

Figure 10: Different combination of actions exist that form the same high-level strategy with approximately the same Q-value

4.2 Saliency maps

Saliency maps for different choices of output values ($F(s)$ having different values such as $Q(s)$ or $V(s)$) all are qualitatively same for all methods. Thus, we will present results using $F(s) = V(s)$, arbitrarily.

In all the saliency maps that we have computed, we have seen that the model attends mostly to the most recent frame. The saliency values for other frames are significantly lower with each frame back in time. This is in line with intuition and shows that the agent has learned to attend mostly to the recent frames. Figure 11 shows the results of different methods for an examplary state.

4.2.1 Vanilla/guided backpropagation

Both gradient based methods give unintuitive results. It can be said that the agent seems to attend to small areas resembling lines, maybe as a way of effective computation. But we could not find out a reason for this behavior and another possible interpretation that the agent is very sensitive to adversarial attacks and has not learned robustly.

In order to test our implementation and results, we have computed an approximation of $\nabla_s F(s)$, by changing each pixel value by a small amount and measuring the differences in $F(s)$. The resulting maps also have the same structure, where the model attends more to horizontal areas of the image.

4.2.2 Occlusion perturbation

In [6] it is suggested that the occlusion perturbation method is not suitable for Atari agents, because adding a square to the grayscale input image changes the game arena and may have semantic information which results in meaningless of unintuitive maps. This is indeed the case with our model in the Pong game. Figure 11 shows an example state. The model seems to change behaviour drastically, as it tries to assign meaning to this change in the context of the game, which an image classification network may not have to do.

4.2.3 Gaussian blur perturbation

In practice, computing the Gaussian blur saliency map is computationally demanding. As a consequence, we clip the Gaussian mask after a certain radius around the pixels. This does

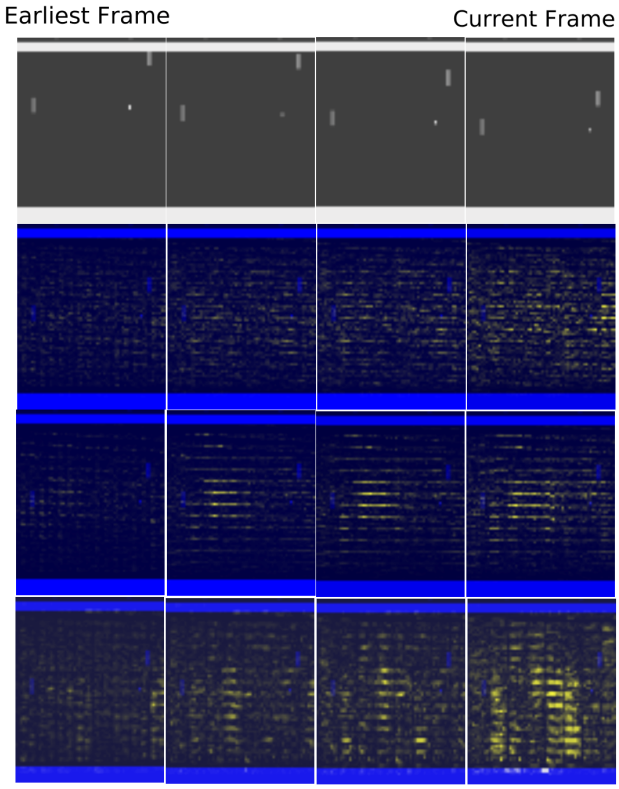

Figure 11: Results of gradient based and box occlusion saliency map methods for an example state. The first row shows the state frames, from the earliest to the most recent. The first row below the state images shows the results of the vanilla backpropagation approach. The next row shows guided backpropagation and the last row is the results for the box occlusion method. All saliency maps have the property that the saliency decreases for less recent frames.

not change the outcome too much, because the mask value decays exponentially. Furthermore, we have computed saliency maps only on the final frame as we have consistently seen that the final frame is the most important one for the agent.

The resulting maps are very intuitive. It shows that the agents focuses on the ball and both paddles with differing intensities in different situations. For the state given in Figure 11, Figure 12 shows that the agent is focusing more on its paddle than the ball or the opponent's paddle. We see higher than average saliency values for some parts of the edges of the game area and this suggests that the agent has not learned robustly and is vulnerable for adversarial perturbations which don't change the game state drastically.

5 Conclusion

The complexity of Machine Learning models increases with the development of new algorithms. With that, it has become an interest in recent years to be able to explain how these new algorithms work and how deep agents behave or act in certain situations.

In this paper, we have demonstrated techniques to understand how Reinforcement Learning agents make decisions. We analyzed methods such as gradient based and perturbation based saliency maps and game tree visualization. All saliency maps show clearly that the agent has learned to attend more to the recent frames, since they are more important in finding the Q-values for the next action. Whilst gradient based methods, specifically the vanilla backpropagation and guided backpropagation, have given unintuitive results that are not easily interpretable, the Gaussian blur perturbation method has given better results that

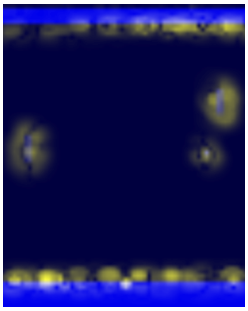

Figure 12: Example frame of the Gaussian blur method

clearly show where the agent pays attention to. From the perturbation based methods, the occlusion perturbation was not suitable for Atari games and has resulted in meaningless and unintuitive maps, whilst the Gaussian blur perturbation was more successful. With its output, we can see that the agent is increasingly paying attention to the ball and the paddles of the pong game.

Despite the saliency maps, the game tree visualization has also been used to understand how the agent gives importance to its actions. We clearly have seen how the Q-values are changing for each continuing frame. With that, we understand why the agent is moving up or down for given states. The total game tree was not visualized, instead, it was only demonstrated for the next n steps such that we can understand the next e.g. 10 possible states.

Further improvements for understanding policies of Reinforcement Learning agents can be achieved by optimizing gradient based methods for Atari games. The DDQN architecture gives us the advantage function of the game. Unfortunately, we were not able to use this feature fully to visualize and understand states of our advantage, because the saliency methods produced the same qualitative results. Another idea would be to analyze how the visualizations differ when rewards are changed, i.e. to cooperative or competitive mode as in [4].

References

[1] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, 2013. arXiv: 1311.2901 [cs.CV].

[2] K. Simonyan, A. Vedaldi, and A. Zisserman, *Deep inside convolutional networks: Visualising image classification models and saliency maps*, 2014. arXiv: 1312.6034 [cs.CV].

[3] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, *Striving for simplicity: The all convolutional net*, 2015. arXiv: 1412.6806 [cs.LG].

[4] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *CoRR*, vol. abs/1511.08779, 2015. arXiv: 1511.08779. [Online]. Available: http://arxiv.org/abs/1511.08779.

[5] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, *Dueling network architectures for deep reinforcement learning*, 2016. arXiv: 1511.06581 [cs.LG].

[6] S. Greydanus, A. Koul, J. Dodge, and A. Fern, *Visualizing and understanding atari agents*, 2018. arXiv: 1711.00138 [cs.AI].