Deep Q-Learning and Explainable AI: Elucidating Training Behaviour, Transfer Learning and Modularity

Michael Dammann

April 2021
Self-optimizing Systems
Winter Term 2020/21
HAW Hamburg - M.Sc. Computer Science
`michael.dammann.compsci@gmail.com`
michaeldammann.github.io

Abstract. In this work, explainable artificial intelligence (XAI) visualization methods are applied in the following fields: (i) to examine the training process of a deep Q-learning Atari agent, (ii) for insights into transfer learning between Atari agents and (iii) to investigate how MATS, a modular approach proposed in this work, functions. The visualizations methods proof to be useful in all of these areas giving strong hints about the inner workings of a model and why an agent succeeds or does not at a given task.

Keywords: Explainable AI · Reinforcement Learning · Machine Learning · Deep Q-Learning · Saliency Maps · Atari · Modular Neural Network

1 Introduction

The field of (deep[1]) reinforcement learning (RL) - and machine learning (ML) in general - is continously growing. With ML applications gaining a foothold in more and more domains there is a demand for transparency and fairness in the used algorithms. This means e.g. that a model should not only give a prediction (like *"Will you get a loan?"*) but also a reason (*"Why you will/won't get a loan"*) for a prediction. The need for explainable artificial intelligence (XAI) stems from the fact that the applied ML models usually are neural networks (NNs) which by design represent a black box, which means they are not explainable. The goal of the field of XAI is to elucidate how these opaque models work for interpretability, ethics and debugging reasons.

In this work, the main goal is to understand how and what a deep Q-learning Atari agent learns during the training process. To accomplish this, *post-hoc* XAI visualization methods are being applied to the agent's NN. After presenting the related work (section 2) and giving background on the methods used and the

[1] There is a distinction between RL and deep RL, but in this work deep RL will simply be refered to as RL.

theory behind them (section 3), this work is then divided into three parts. In the first part (section 4), the training process of an Atari DQN agents will be examined using an analysis of saliency and feature maps. In the second part (section 5), transfer learning between Atari agents will be investigated. In the third part (section 6) an approach called MATS is proposed that separates the agent into multiple modules that each have their own task. This approach is also inspected using XAI methods. In the end, this work closes with a summary (section 7).

2 Related Work

In the following section an overview will be given on research done in the topics touched by this work. Also it will be described how this publication differentiates from research already done.

RL and XAI: There exist publications that make use of methods from the field of XAI to explain the decisions of a RL agent. For example Z. Wang et al. [39] and Atrey et al. [1] use *saliency maps* (see section 3.3) to visualize what an agent focuses on for a value estimate. Greydanus et al. [11] propose a perturbation-based approach to visualize an agent's reasoning. Attention [2], which has proven to improve performance and interpretability of NNs, has also been applied in multiple ways in RL [6, 24, 30]. J. Wang et al. [38] present DQN-Viz, which is an approach that is more in spirit of this work, since it provides a lot of insight not only of the final agent but also the training process, providing multiple visualizations using a *visual analytics* approach.

As described, this work also tries to elucidate the training process of an agent and not (just) the decision making of the final agent. This means for example answering questions like: *At what point in training is the agent able to handle certain situations and why*? To answer these questions, in addition to saliency maps, this work also uses feature maps, filter visualizations and statistics on the weight changes in the NN of an agent. These methods are then also applied in the fields of transfer learning (part II, section 5) and the MATS approach proposed in part III (section 6).

RL and Transfer Learning: Transfer learning is the practice of using parts of an already trained ML model on a new task. Zhu et al. [41] provide a survey on RL and transfer learning. They organize different approaches into the subcategories of (i) *learning from demonstrations*, for example from a human expert, (ii) *policy transfer* from one or multiple pretrained policies, (iii) *representation transfer*, which means transfering feature representations (iv) *inter-task mapping*, which uses mapping functions between a source and a target domain and (v) *reward shaping*, which adds additional rewards to guide the agent in the direction of good solutions.

In their *future perspectives* section, Zhu et al. bring up the point of interpretability in the context of transfer learning. This is the point this works comes in as it tries to evaluate the success of multiple *representation transfer* learning tasks based not only on metrics (i.e. return) but also on XAI visualizations. This is how it could be made easier to interpret why a representation transfer was a success or not. (*What parts of the input does the agent focus on using this representation transfer in comparison to a normally trained agent*?)

RL, Modular NNs and Autoencoders: For reasons of interchangeability, explainability and performance, the tasks of a NN can be split into several *modules*. In RL, this idea is closely related to *hierarchical* RL, where a task and actions are grouped up and/or divided, usually ranging from a top-level view to a detailed one. Calvo et al. [4] present a hierarchical, modular approach for autonomous robot navigation. Kulkarni et al. [20] propose a hierarchical method for temporal abstraction in deep Q-learning. The approach of Zhang et al. [40] consists of lower-layer workers that learn a policy and a higher-layer scheduler that selects a policy.

In this work, to create a modular approach, autoencoders (AEs) are used, which are a common way to learn features [15]. Multiple works have been published in the intersection of feature learning with AEs and RL. Kimura [16] uses features learned by an AE on images to teach a robot a game using deep Q-Learning (see section 3.2). Van Hoof et al. [37] present a RL-AE approach for tactile and visual data. Lange et al. [22] propose a framework that trains the AE not before but concurrently with the agent, which might be useful because some situations do not occur until later steps in training.

The modular MATS approach proposed in part III (section 6) is build on autoencoder representations, which are then processed in a specific way by a CNN.

3 Background

3.1 Neural Networks

NNs are highly parameterized, universal [7] function approximators that are used in the field of ML [19]. They are the tool of choice in deep RL for value approximation and policy gradient methods. They consist of input, hidden (often many of those: *deep* learning) and output layers. Every layer receives the input from the previous layer (apart from the first layer which gets the input) and performs a linear transformation using *learned* weights. Usually this is followed by a non-linear activation function. These vanilla versions of NNs are often called *multilayer perceptrons* or *dense* NNs.

For training, a network is given *labeled* inputs (supervised learning) and the task consists of predicting the correct label[2]. The network gets better at this task by iteratively minimizing a defined error that describes the distance between the network output and the target value (e.g. mean absolute error or binary cross-entropy). This is accomplished using *stochastic gradient descent* (SGD) (or a variant like *Adam*), which is realized in NNs using *backpropagation*.

Convolutional neural networks (CNNs) [23] are a type of NN that are able to process spatial data like images (3D-arrays: height, width, channels) in contrast to dense networks that work with vectors as input. The learnable weights

[2] *Label* can also refer to a regression variable and not necessarily a class.

form a window (*filter*) that is sliding over the input to create the next layer by element-wise multiplication (*convolution*). Local connectivity, shared weights for all window positions and keeping the multidimensional structure are the central concepts for CNNs. Along the lines of how multilayer perceptrons work, the output of a convolutional layer (so called *feature maps*) is given as input to the next convolutional layer. Often the final output of a NN is a vector like in RL value or policy approximation tasks. To get from a spatial 3D structure to a 1D vector the feature map of the last convolutional layer is simply *flattened* (i.e. all values are sequentially added to a long vector) and than given as input to a *dense* layer.

Such architectures have led to great success in many fields of ML like image classification [13] or object detection [31] and in relation to this work especially in RL tasks that learn straight from pixels and not from handcrafted features like in the considered Atari games. The work from Mnih et al. [26,27] presenting a deep Q-learning approach using CNNs for solving Atari games is considered a breakthrough and will be build onto in this paper.

3.2 Deep Q-Learning

Generally speaking, RL is the study of agents interacting with environments learning to maximize a reward signal. *Interacting* means that at every time step, the agent receives the current state s by the environment and is executing an action according to a given policy $\pi(s)$.[3] Afterwards the environment yields the next state s' for the agent and a numerical reward r. [34]

$Q_\pi(s,a)$ is called the action-value function or simply Q-value and describes the expected return from taking action a in state s and then following policy π:

$$Q_\pi(s,a) \equiv \mathbf{E}[R_1 + \gamma R_2 + \gamma^2 R_2 + ... | S_0 = s, A_0 = a, \pi] \tag{1}$$

Here, γ is the discount factor, which is a hyperparameter that is used to balance weighting between immediate and *delayed* rewards. The optimal action-value function is defined as $Q_*(s,a) = \max_\pi Q_\pi(s,a)$ and thus using $Q_*(s,a)$ an optimal policy π_* can be obtained by taking the action of highest value in every state. [34]

Q-learning is an approach for learning (estimates of) optimal action-values. Firstly, it is an off-policy algorithm, which means that the value function is not updated necessarily using the actual taken action (determined by a behaviour policy choosing actions *epsilon-greedily*) but using the highest valued action. Secondly, it is a temporal difference learning algorithm, which means that after *every* step (contrary to *Monte Carlo* methods) the value function - an estimate - is updated, based on another estimate (*bootstrapping*). Thirdly, this approach belongs to the category of model free methods, which means that the states and rewards are generated by the environment and no model of the environment is learned. [34]

[3] $\pi(s)$ describes a deterministic policy (output is an action a). The notation for a stochastic policy is $\pi(a|s)$ (output is probability for taking a). [34]

Originally starting with tabular methods, a lot of today's high-dimensional RL tasks make use of parameterized function approximators like NNs because the number of states is too big to be stored efficiently in tables. For example taking an Atari game frame (128 colours, 210x160 resolution) as a single state, the compelling size of the state space is $128^{210 \cdot 160}$. Applying NNs to Q-Learning, the parameters θ_i of a *Deep Q-Network* (DQN, see figure 1 (a) for the Atari DQN) [27] can be optimized via SGD using the following loss function at iteration i:

$$L_i(\theta_i) = \mathbf{E}_{s,a,r,s'} \left[\left(y_i^{DQN} - Q(s,a;\theta_i) \right)^2 \right] \tag{2}$$

with the target

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{3}$$

where θ_i^- stands for the parameters of a *target network*. Using the same network that is getting updated for also calculating the target leads to instabilities in training (*chasing it's own tail*). Freezing the parameters of a target network and only updating it every τ steps leads to better training performance [27].

Another concept which has shown to improve deep Q-learning by a lot is *experience replay* [27]. The agent's experiences are stored in a memory. For training, mini-batches are randomly sampled from this memory. This decorrelates the samples and makes the data more like i.i.d. data which is generally assumed in NNs. It also leads to more efficient use of previous experience since it can be used multiple times.

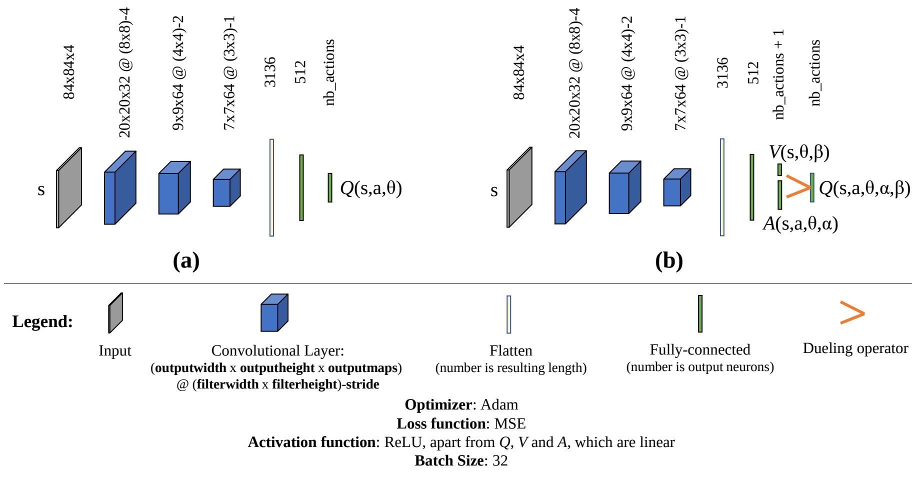


Fig. 1: Network architectures used for (a) Double DQN and (b) Dueling Double DQN with learnable parameters θ and (θ, α, β) respectively.

In eq. 3, using the same DQN to both select and evaluate an action can result in *overoptimistic* value estimates. To reduce this problem, van Hasselt et al. [12] propose to use the target network for evaluation and the online network for selection and call their approach *Double DQN* (DDQN), which uses the following target accordingly:

$$y_i^{DDQN} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta_i^-) \tag{4}$$

Wang et al. [39] state that for many states, it is not needed to estimate each action-value because the action has no effect on what happens. For example in the Atari game *Breakout* it does not matter in which direction the player moves if the ball is very far away. However it could be useful to estimate the state-value $V(s)$. Wang et al. [39] propose the *Dueling DQN* architecture which is using two *streams* to predict the state-value $V(s)$ and *advantages* $A(s,a)$ (see figure 1 (b)). With this, which states are valuable or not can be learned explicitly. Their relationship is given by

$$Q(s,a) = V(s) + A(s,a) \tag{5}$$

So intuitively, the advantage shows how much better choosing a specific action is compared to the other actions. Eq. 5 could be used to naively calculate $Q(s,a)$ for the network output. However this formulation is *unidentifiable*, which means that $V(s)$ and $A(s,a)$ cannot be recovered uniquely by $Q(s,a)$.[4] This results in poor practical performance. To address the issue of indentifiability, Wang et al. [39] propose the following formulation subtracting the average A (where θ, α and β represent learnable parameter vectors):

$$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + \left(A(s,a,\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a',\theta,\alpha) \right) \tag{6}$$

This changes the original semantics of A and V by a constant but has shown to improve performance greatly. In figure 1 (b), this step is illustrated as the orange *dueling operator*. It is implemented as part of the DQN so no extra algorithmic step is needed. This means that any deep Q-learning algorithm (like the ones described above) can be applied for training and that V and A will be learned automatically via SGD and backpropagation.

In this work, Double DQN (*DDQN*) and Dueling DQN using the Double DQN algorithm (*DDDQN* or *D^3QN*) will be applied.

3.3 Explainable AI

The described good performance of NNs does not come without drawbacks. As mentioned, NNs with their great number of learnable parameters are *black-box*

[4] To illustrate this, take some constant and add it to $V(s)$ and subtract it from $A(s,a)$. $Q(s,a)$ will stay the same.

models, i.e. they don't give insights into their reasoning process. The field of XAI has set itself the task of elucidating NN reasoning and to (at least partially) turn the opaque into white-box models. According to Carvalho et al. [5], XAI methods can essentially be categorized in the following two classes:

Ante-hoc/Intrinsic models: This refers to NNs that augment their architecture by some component that is added *before* training time and *learns* to explain the prediction of the network. A typical example is the attention mechanism which guides the model to explicitly (i.e. humans can interpret it) focus on certain parts of the input. For example in machine translation, the "result" of the attention mechanism for a single time step of translating is a attention vector that associates *learned attention weights* with the individual input parts [2]. The attention weights can be used to determine what parts of the input are important for the current part of the translation.

Post-hoc: In contrast to ante-hoc methods, post-hoc methods are applied to opaque models *after* the training is finished. *Saliency maps* [32] are an example for these methods and will be applied in this work. The basic idea behind saliency is pretty straightforward: Given a *trained* model and an input image, the gradient of a output category is computed with respect to the pixels of the input image[5]:

$$\frac{\partial output}{\partial input} \tag{7}$$

Intuitively, the gradient shows how much the output (like the Q-estimate) changes when small changes are made to certain parts the input. One can acquire saliency maps by reshaping the gradient vector to the input image size and visualizing it by applying some colormap (so high gradients are displayed in bright colors). This is how saliency maps can be used to examine which parts of the input determine the output the most.

Visualizing feature maps, which is also done in this work, can also be seen as a post-hoc methods, since it provides insights to how convolutional layers process inputs after the a model is trained.

3.4 Atari games

The considered environments in this work are the Atari games *Breakout*, *MsPacman* and *BeamRider*. In Breakout, the player has to destroy bricks in the wall at the top of the screen to gain points. In MsPacman, one has to collect food and dodge the enemies in the shown labyrinth. In Beamrider, the objective to win is shooting the oncoming, moving spaceships. Atari games have become a benchmark in the scientific RL community for agents that *learn from pixels*.

Following Mnih et al. [27], for Atari games a state s consists of **four subsequent gray scale frames** of the game. This is because environments are formally described as *Markov decision processes* (MDPs), i.e. (simply put) the current state completely represents the process that led up to this state and all

[5] So for an MxN input image the gradient vector is MxN long.

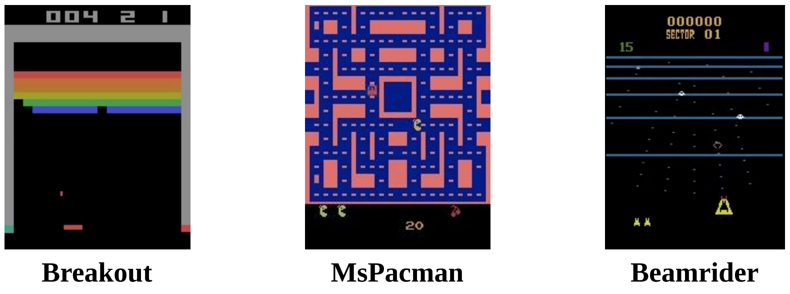


Fig. 2: Considered Atari environments for this work.

information that is necessary to decide for an action. So for example for Breakout, the four subsequent frames hold all this necessary information: where the objects (bricks, ball, player) are positioned and in what direction and how fast they are moving. In contrast a single frame would not be sufficient to determine direction and speed. Also for the next action and state it is not important e.g. where the ball was positioned a few seconds ago, which is why only the *current* four frames are taken into account.

An action a is a specific, *discrete* button press like “left” or “right” which moves the agent. The rewards (e.g. for hitting a brick in Breakout) are clipped between $+1$ and -1 for better comparability between environments. All Atari tasks are considered *episodic*. I.e. there exist terminal states and the tasks end. In Breakout for example, the player is given five balls (*lives*). After all balls are used, one gets the final score, the bricks at the top reset and the episode ends.

4 Part I: Elucidating RL Training Behaviour

4.1 Experimental setup

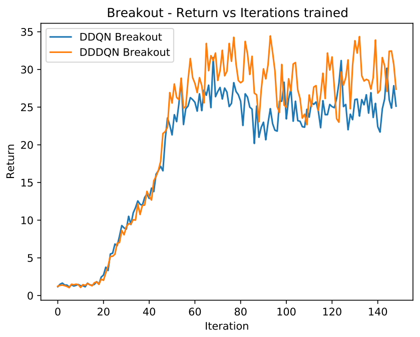
In this section, the training behaviour of deep Q-learning Atari agents is examined. For this, the games Breakout, MsPacman and Beamrider (see section 3.4) are used. They have been chosen because they represent different classes of games. This is meant in the sense of Breakout being comparatively simple with only the tasks of anticipating and hitting the ball in the right direction. MsPacman is a more complex game involving pathfinding and evading enemies. Breakout lies somewhere between these two with moving enemies that have to be hit.

The examined deep Q-learning algorithms are DDQN and D^{3}QN. The used NN architectures can be seen in figure 1, the architectures are based on [27] and [39] respectively. Each agent is trained for 3 million steps. This is a training time that has shown to result in agents that have a solid understanding of the game, albeit not being perfect at it. For the sake of limited time resources[6], 3

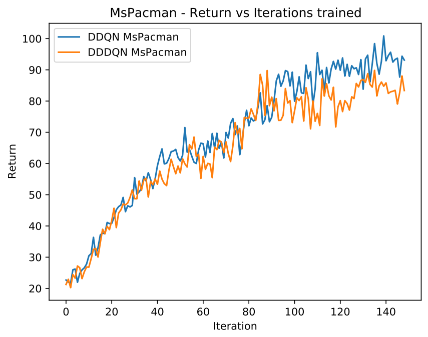
[6] Training for 3 million steps takes around 10 hours on the used hardware.

million steps seem like a good middle ground between having good agents and also being able to carry out enough experiments to have meaningful results. More technical details on used software, settings and hyperparameters for the experiments done in this work can be found in the appendix.

4.2 Saliency, feature maps and filters to understand learning



(a) Breakout



(b) MsPacman

Fig. 3: Return vs. iterations trained for (a) Breakout and (b) MsPacman. 1 iteration = 20k steps.

Figure 3 shows the received return during the stages of the training process. It can be seen that the D^3QN performs slightly better than the DDQN, which is consistent with the findings of Wang et al. [39]. Contrary to Breakout, looking at MsPacman, the D^3QN seems to perform slightly worse than DDQN. Note however, that after 3 million steps the agent does not perform at it's best possible and that this may change after more learning.

Figure 4 shows saliency maps and Q-values for the actions *right* and *left* and for a certain state throughout training. The state shows the player located on the right and the ball (zooming in on the picture might be necessary) moving in the bottom left direction. Quite dynamic saliency changes can be observed. The focus generally lies on the middle area, where the ball is moving. The saliency changes from the left to the right and vice versa over time. In the end, the saliency is relatively balanced with a slight focus on the side the player would move towards applying the action. For this state the saliency does not seem to focus on where the ball is moving throughout the training. A possible explanation might be that the player is already positioned in the right place to hit the ball and no correction towards the movement of the ball is needed.

In figure 5, it is displayed how two feature maps (first layer) and their filters change during the training process. Note that the input consists of four channels

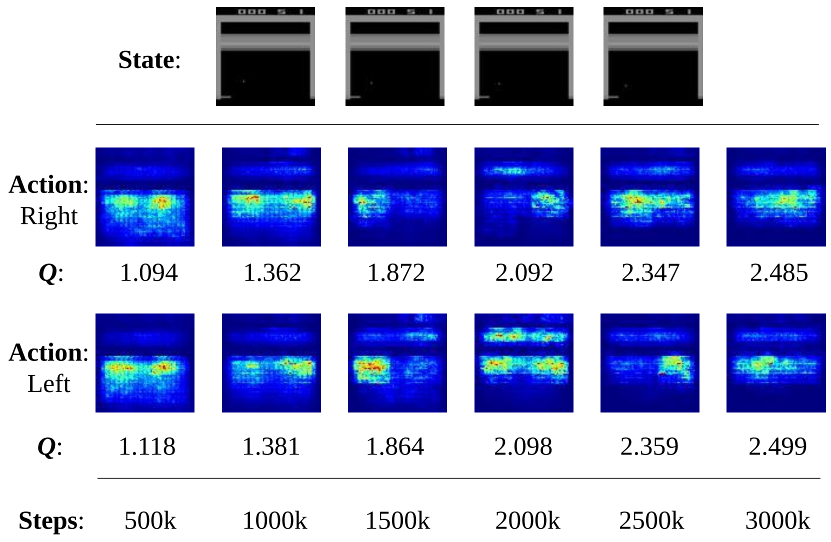


Fig. 4: Saliency maps throughout training for the DDQN Breakout agent.

corresponding to four grayscale frames, which leads to four filters. The filter on the left seems to detect objects moving in the bottom left direction. This can be derived by looking at the final filters after 3 million steps: the first one puts focus on the top right section of the input. Throughout the channels, the attention is then shifted to the bottom left. It can be seen that the agent needs around 1.5 million training steps to isolate the phenomenom of *object moving to the bottom left*. Before that, artefacts like the line at the top are also detected[7].

The filter on the right seems to detect horizontal parts of the screen like the player in the bottom left or the boundaries of the blocks in the top area. Judging by the feature map, the agent has learned to detect horizontal lines at 0.5 million steps, which is quicker than the 1.5 million steps for the detection of an *object moving to the bottom left*. A reason for this could simply be that the ball is a lot smaller than the horizontal lines on the screen and thus initially neglected in early training stages.

Similar to figure 4, figure 6 shows saliency maps and Q-values for the action *right*, this time for a more advanced state. The state shows the ball moving to the bottom right and the player being positioned in the bottom left. Starting from a diffuse saliency map marking the general area of ball movement, the final saliency map displays focus on already hit parts of the wall and also the direction the ball is moving to. Compared to figure 4, more attention might be paid to the ball because the player is not yet in the proper position the hit the ball and needs to move towards the right.

[7] Note however that filters are often ambiguous and don't necessarily only detect a single phenomenon.

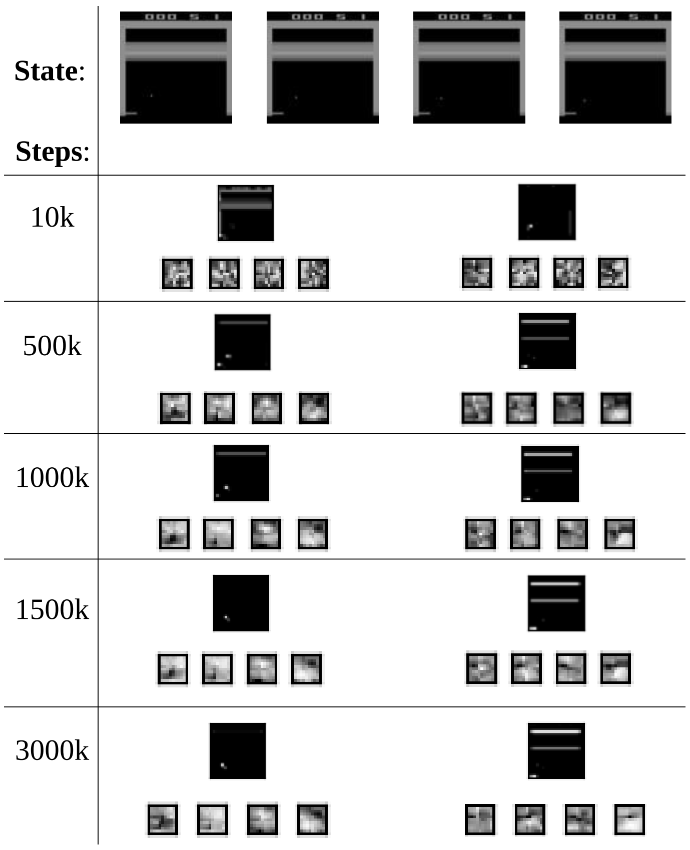


Fig. 5: Feature maps and their filters from the first convolutional layer throughout training for the DDQN Breakout agent.

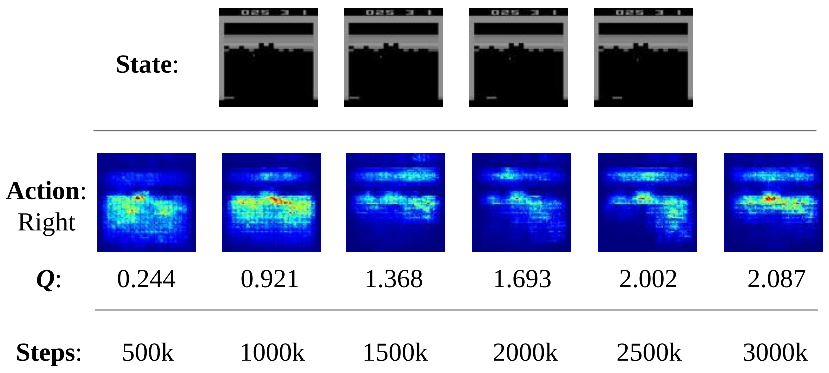


Fig. 6: Saliency maps throughout training for the DDQN Breakout agent for an advanved state.

In figure 7, the D^{3}QN agent for MsPacman is examined. Again, saliency maps are displayed. Three states are shown from top to bottom, the first two picture an advanced stage of the game, the last one displays the beginning of a game. For the top two states it can be seen how the saliency maps first outline the labyrinth as a whole. After 3 million steps, the focus lies on the part of input where the agent is positioned in the bottom left.

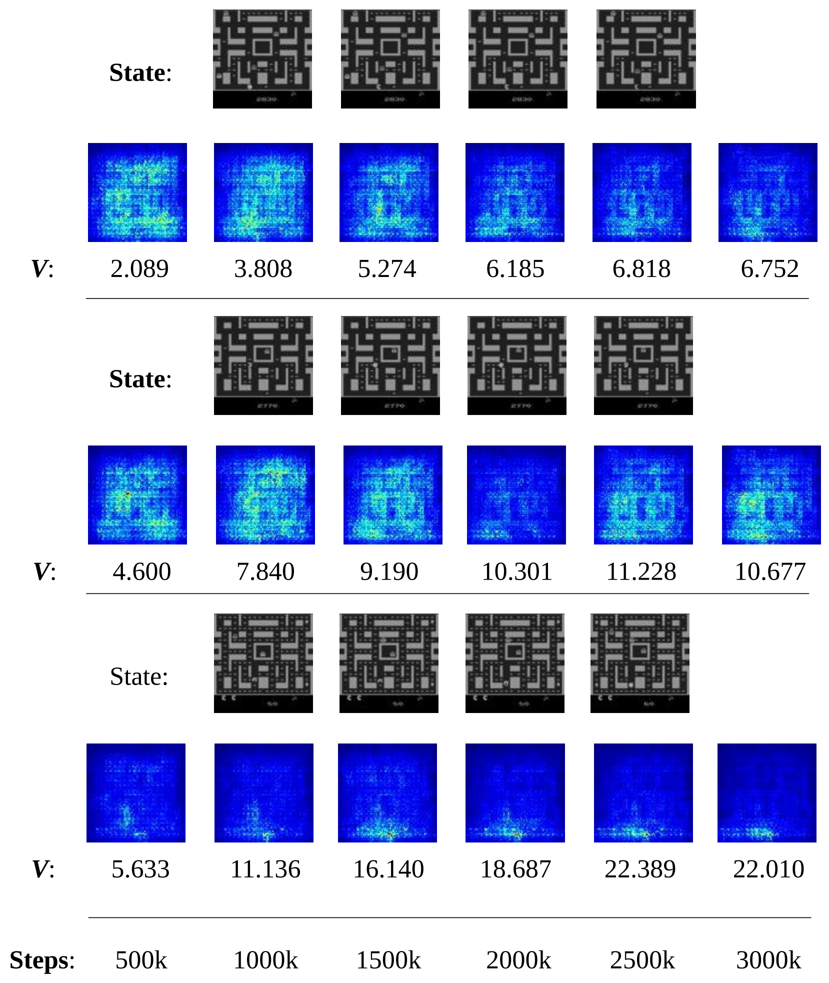
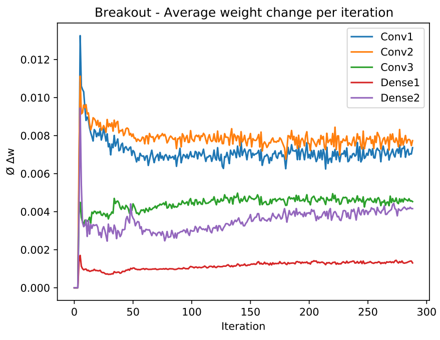


Fig. 7: Saliency maps throughout training for the D^{3}QN MsPacman agent for multiple states.

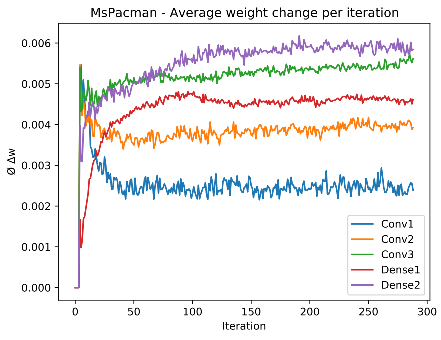
Another interesting observation that can be made from the top two states is that V roughly equates to the remaining food in the focused bottom left area.

For the top state, 7 points still remain in that area and $V = 6,752$. For the second there are still 12 point left and $V = 10.677$.

The state at the bottom paints a different picture. No noteworthy saliency is developed during the training, not even of the general labyrinth outline. The reason for this could be that in the beginning it does not matter which direction one chooses: the player in MsPacman moves automatically, every direction provides food (reward, note the high V) and the enemies are far away or have not spawned yet.



(a) Breakout



(b) MsPacman

Fig. 8: DDQN weight change vs. iterations trained for (a) Breakout and (b) MsPacman. 1 iteration = 10k steps.

Looking at figure 5 one could come to the conclusion that the filters don't change much anymore after 0.5 to 1.5 million steps. To examine which layers change most at what stage of the training, the *weight changes* per iterations of 10,000 steps have been calculated and visualized. This means that for every iteration, every single weight is being compared to the corresponding weight from the previous iteration's model by taking the absolute deviation. Per layer the mean absolute deviation is then calculated. The weight changes are visualized in figure 8. It can be seen that the weight change stays constant throughout training. At first glance this stands in contrast to figure 5 where the filters don't seem to change much anymore soon after training initiation. Though this does not mean that the numerical weights don't change anymore at all. For example a filter could be *strengthened* by increasing the already high weights and decreasing the low ones. This would not make for a big change in filter structure and it's visualization but still for a change in weights. Bringing figure 5 and 8 together, the possible takeaway of this observation is the following: the CNNs learn their final filter structures quite early in training. However they are not *dead* (which could happen with ReLU [9]), refine their structures later on and - in theory - retain their ability to change their filters.

4.3 Forgetting in Beamrider

The experiments with Beamrider differ from the Breakout and MsPacman ones. In figure 9 it can be seen that the Beamrider returns decrease after about half of the training is done. Interestingly, this happens in both DDQN and D^{3}QN. This phenomenon is called *forgetting*, which is a rather common occurency in RL or sequential learning problems in general like lifelong learning [18, 29]. The term *catastrophic forgetting* or *inference* refers to the phenomenom of a NN forgetting everything or a lot of what it has learned so far. The forgetting in Beamrider is less drastic, however still notable (around 30 %) and will be investigated in the following section.

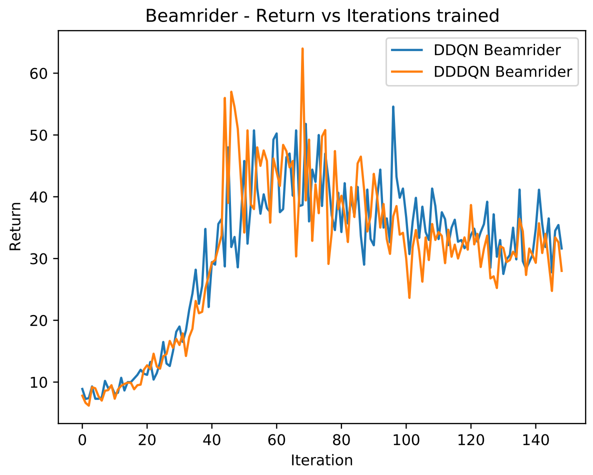


Fig. 9: Return vs. iterations trained for Beamrider. 1 iteration = 20k steps.

Two saliency maps for Beamrider are shown in figure 10. What can be seen is that during peak performance after about 1,000,000 steps the saliency maps semm to focus on the important parts of the state: The horizon and lane with a slight emphasis on the player position. After 3,000,000 steps, this has changed. The attention is drawn mainly to the top of the screen. This is a possible explanation for the degrading performance. Saliency maps for other states show a similar shift in attention.

Figure 11 displays two feature maps for Beamrider. It can be seen, like in figure 5, that the filters reach their general final shape quite early in training. They also don't change when forgetting takes place in Beamrider (after around 1,500,000 steps), so the agent seems to not generally unlearn the detection of concepts it has already learned in the first convolutional layer. This could mean that forgetting takes place more in the dense layers or deeper convolutional layers. Also that both the DDQN and D^{3}QN are *forgetting* after the same amount of steps could be a sign that the problem arises with exploring new regions in the state space by advancing in the game, which is a common reason for oscillations in received returns.

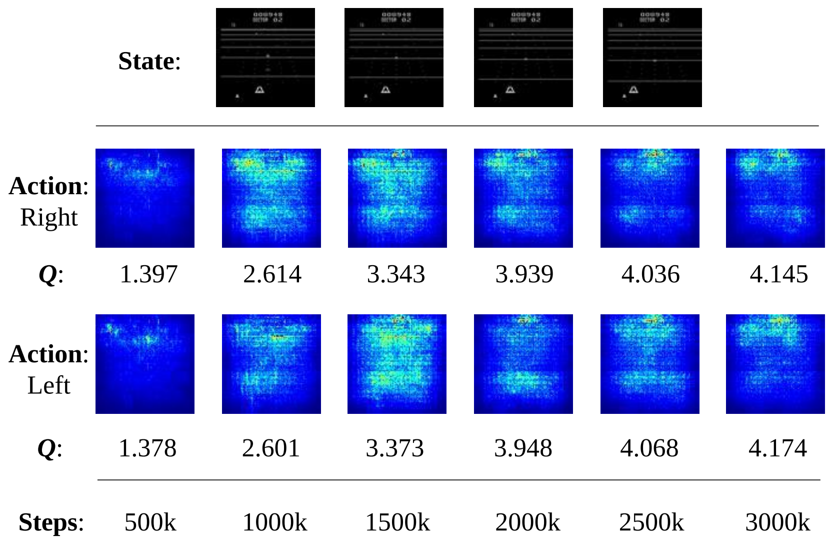


Fig. 10: Saliency maps for Beamrider throughout the training process.

4.4 Conclusion of Part I

It could be shown that saliency and feature map visualization help to understand how an agent learns and makes it's decision. It's applicable for both DDQN and D^{3}QN. Using saliency maps, it is possible to show how the attention of a model shifts during training. Feature maps can be used to investigate at what point in training an agent has learned certain concepts. The filters also seem finalized in their general structure quite early in training, but looking at the weight changes they seem to remain *plastic*. Forgetting in Beamrider was investigated and there are hints that the features detected in the first layer remain stable and that forgetting takes place in deeper layers.

In the future, of course more environments and visualization methods like Grad-CAM could be tried out. Also other algorithms like DDPG or PPO for continous action spaces could be examined. In general, but specifically for the *forgetting* problem, it could be helpful to train longer to see if the agents recover from their decline. Additionally, a comparison with DQNViz [38] results could be made to see to what extend the visualizations done in this work are comparable and provide additional insights.

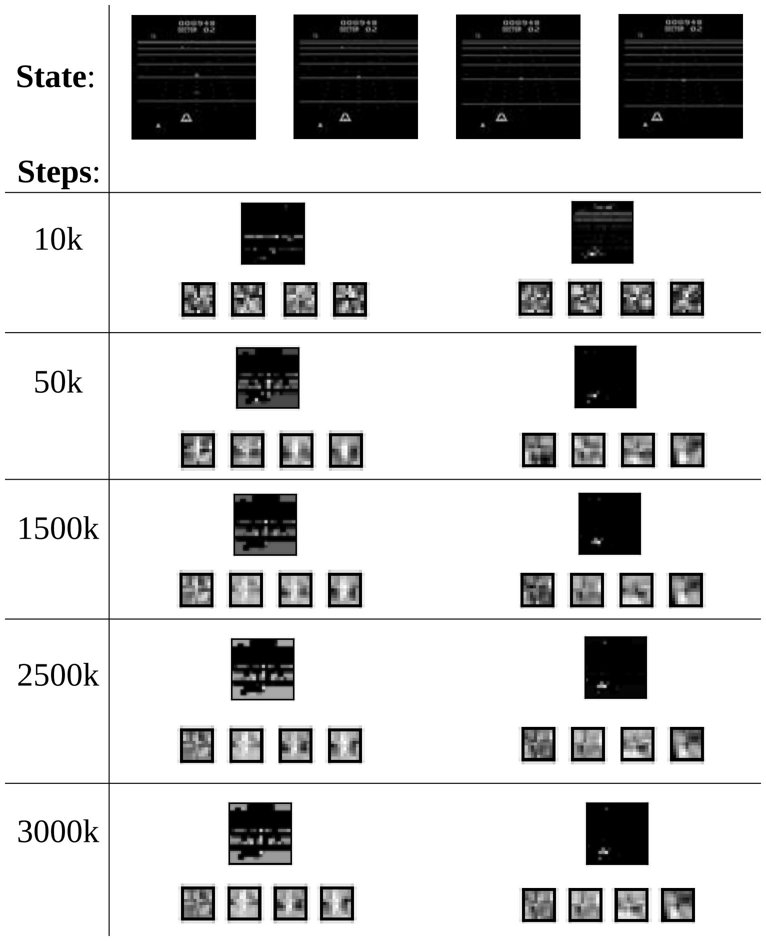


Fig. 11: Feature maps from first convolutional layer for Beamrider throughout the training process.

5 Part II: RL and Transfer Learning

5.1 Experimental setup

In this part, it will be investigated how well *transfer learning* can be applied in the domain of RL. Transfer learning in general is described as the reuse of (parts of) an already trained NNs for a new task. It can be distinguished between *finetuning* weights and using *frozen* weights. *Finetuning* means using the weights from another NN as the starting point for training on a new task [21]. Doing so, the network can possibly access already learned concepts and does not have to start at zero. *Freezing* weights means that usually the convolutional layers including the flattened output directly after the last convolutional layer are reused (but not included in training/finetuned) on a new task [3]. The idea is that some already trained network has generalized enough that it's convolutional layers can be used as a universal *feature extractor*. By leaving out the convolutional layers on the new task, training can be accelerated by simply reducing the computational costs. In many cases, using transfer learning can improve result [25,28,36]. Examples for commonly used feature extractors are VGG16 [33], VGG19 [33], InceptionV3 [35] or ResNet50 [13], each being NNs that are trained on large data sets like ImageNet [8].

In the following part, the goals are (i) to determine how good RL agents trained on one task can be applied on a new task and (ii) using saliency and feature maps to find out *why* transfer learning works or does not. The following setup is chosen: An agent is trained on Breakout. Using the weights of this Breakout agent, the following experiments are carried out:

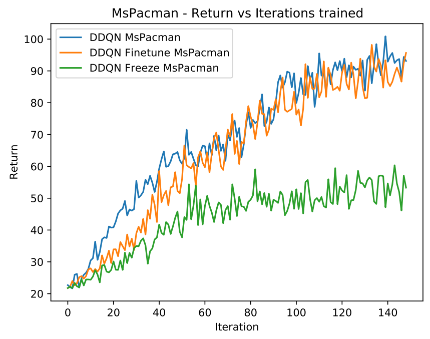
(i) Using Breakout weights to learn **Beamrider** by **finetuning**
(ii) Using Breakout weights to learn **MsPacman** by **finetuning**
(iii) Using Breakout **feature extraction** to learn **Beamrider**
(iv) Using Breakout **feature extraction** to learn **MsPacman**

As described in section 4.1, Beamrider is - at least for humans - closer related to Breakout than MsPacman. The experiments shall also elucidate whether this intuition translates to transfer learning, i.e. show if transfer learning works better for Beamrider than for MsPacman. The same network architectures and hyperparameters as in part I are applied.

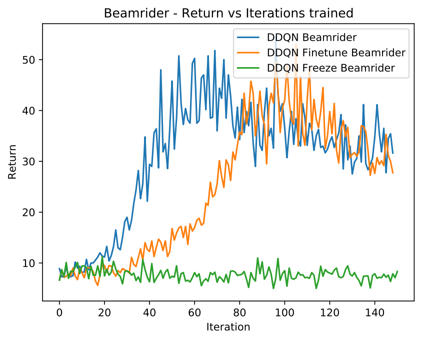
5.2 Results

Figure 12 shows the return received by the agents throughout the training process. For finetuning, it can be seen that in the end the finetuned agents perform on the same level as the vanilla agents. The Beamrider finetuned agent even replicates the forgetting taking place in the vanilla agent. In the first half of training, in both MsPacman and Beamrider there seems to be a lag in performance between finetuned and vanilla agents before the finetuned ones catch up.

Comparing the vanilla agents with the ones using the extracted features by the Breakout agent (called *freeze* in figure 12), the picture is very different. The MsPacman performance is notably worse than that of the vanilla agents and the Beamrider agent does not seem to learn anything. In the following it shall be examined why transfer learning succeeds or fails using multiple visualizations.



(a) MsPacman



(b) Beamrider

Fig. 12: Return vs. iterations trained for (a) MsPacman and (b) Beamrider. 1 iteration = 20k steps.

Figure 13 shows saliency maps for the three MsPacman agents. For the fine-tuned one (middle), it is visualized how during the training process the saliency maps gets closer to the vanilla one (top), which has kind of a pyramid shape. Looking at the agent using the features extracted by the Breakout agent (bottom), it can also be seen that the *pyramidic* saliency from the vanilla agent is approximated. Other saliency map comparisons showed similar findings. This means that at least partly the Breakout feature extraction can be reused for MsPacman and for reconstructring the saliency maps by the vanilla agent.

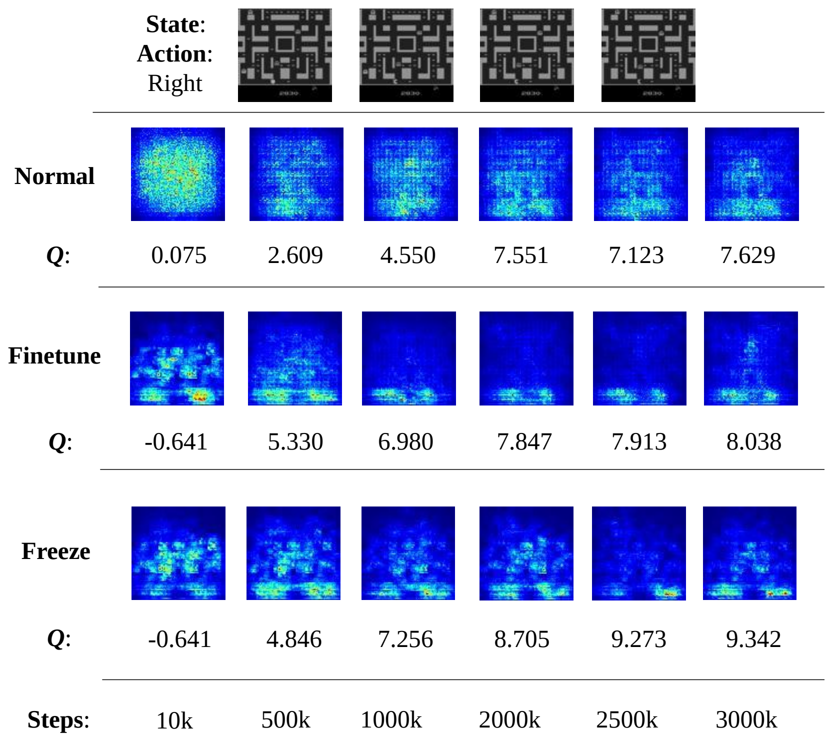


Fig. 13: Saliency maps throughout training for the transfer learning MsPacman agents for an exemplary state.

The saliency maps for the Beamrider agents are displayed in figure 14. For the finetuned agent it too can be seen that after 3,000,000 steps the vanilla saliency map is approximated. The *freeze* agent however shows a very different saliency map with a strong focus on only where the player is positioned but not on the areas where enemies spawn which is at the top of the screen. Other saliency maps even show no focus at all, they are just dark. Concluding, the Beamrider agent does not seem to be able to pay attention to the right parts of the input using the features extracted by the Breakout agent.

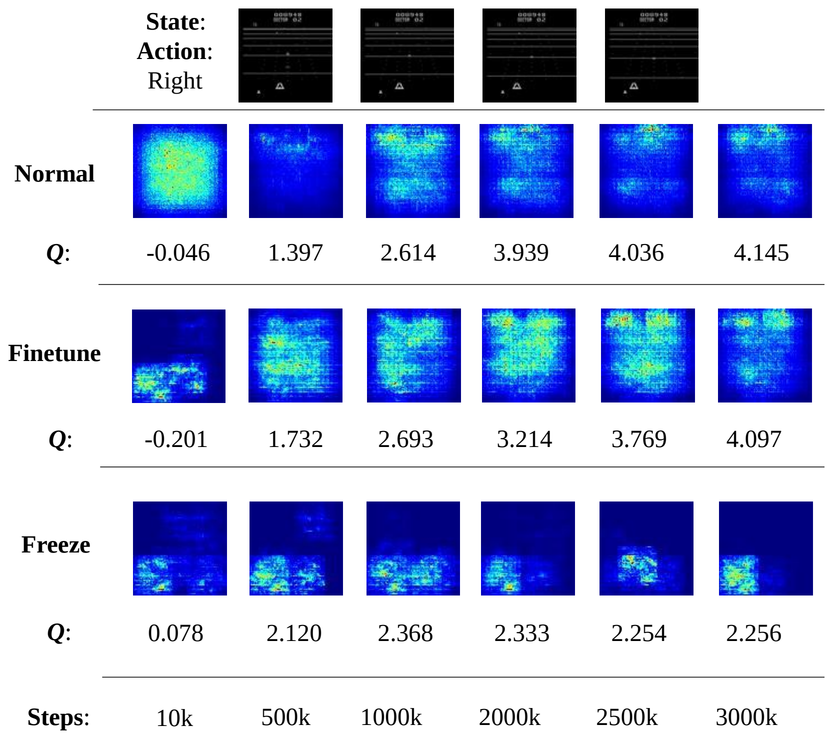


Fig. 14: Saliency maps throughout training for the transfer learning Beamrider agents for an exemplary state.

For further insight, the 32 feature maps of the first convolutional layer using the Breakout agent are visualized in figure 15. Looking at the Beamrider input (top) it can be seen that the focus mainly lies on the player position and horizontal lines. This is consistent with the findings of figure 5, where it is shown that the Breakot agent learns to detect (i.a.) the ball and horizontal lines. However, no feature map can be made out that pays attention to the enemies in front of the player. This is probably the reason why the saliency maps only show highlights in the player area and the agent does not learn. This is surprising since the ball in Breakout is about the same size as the enemies in Beamrider and moves in similar ways.

Looking at MsPacman (bottom), the feature maps are very hard to interpret, but there seem to be some representations that highlight certain parts of the maps, horizontal lines in the labyrinth or enemies (top left for example). This probably led to the better performance of the MsPacman agent using features extracted by the Breakout agent.

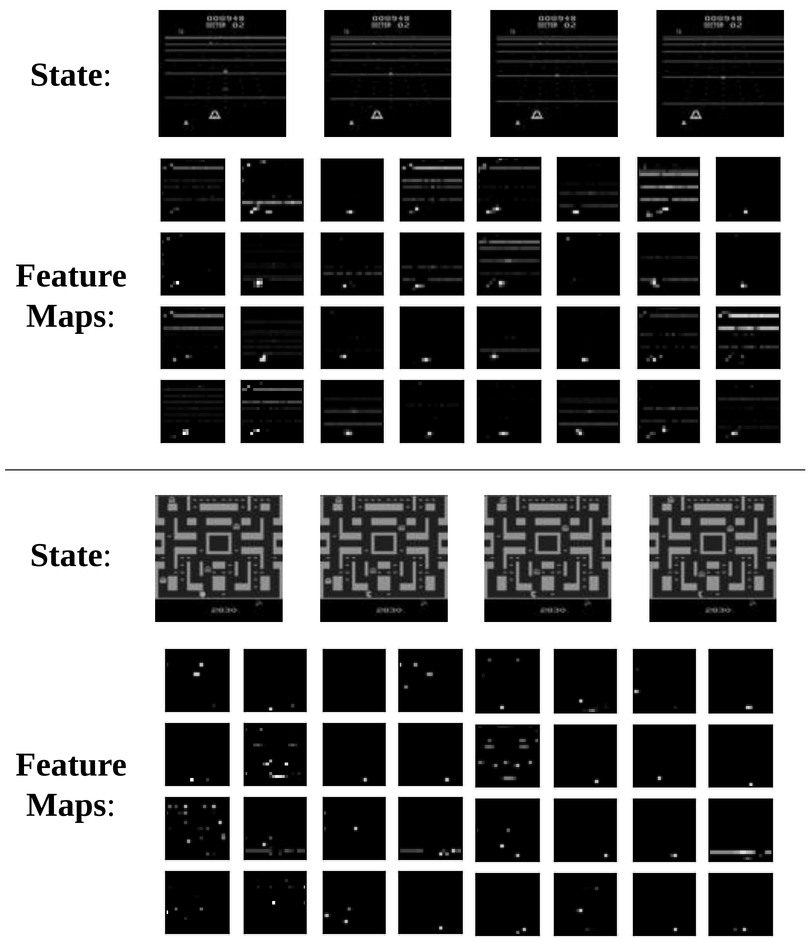


Fig. 15: Feature maps of first convolutional layer Breakout feature extraction for Beamrider (top) and MsPacman (bottom).

5.3 Conclusion and Future Work of Part II

For finetuning, both the MsPacman and Beamrider agent reach the level of their vanilla counterpart. However there is a lag in performance in the first half of training. This means the finetuned agents can't benefit from the weights learned in Breakout. Instead, they hinder them from learning. The agents seem to learn their new task from zero. That there is a lag in performance could be explained by wrong weight initialization. Usually, the Glorot Initializer is used, which initializes the weights depending on number of input and output neurons. This improves training performance [10]. Initializing with the Breakout weights, the Glorot conditions are not met anymore and training slows down.

It could be shown that transfer learning using features extracted by the Breakout agent works partly for MsPacman but not for Beamrider. Furthermore it was demonstrated that looking at saliency and feature maps, very good hints can be found regarding *why* transfer learning works or does not. The introductory assumption of the *freeze* Beamrider agent functioning better than the MsPacman one was not met. This shows (once again) that human intuition and how NNs learn are often not in line and that it is necessary to take a deeper look into how a model works.

In the future, experiments could be carried out extracting features using VGG19 or a comparable NN trained on ImageNet. Another approach could be transfer learning between discrete and continuous tasks. Also transfer learning in non-image tasks could be investigated.

6 Part III: MATS

6.1 Main Idea

In this part, an architecure is proposed that portions the original network into several parts with the idea in mind that each part serves it's own purpose. This approach will be named *Modular Autoencoder-based Task Splitting* (MATS) and described in the following section.

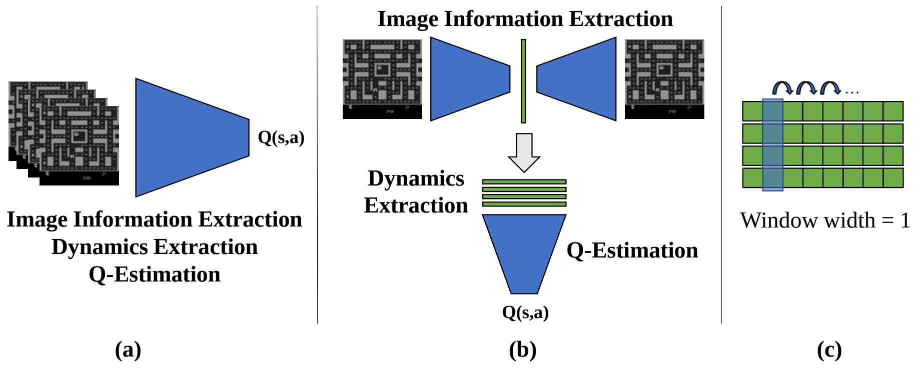
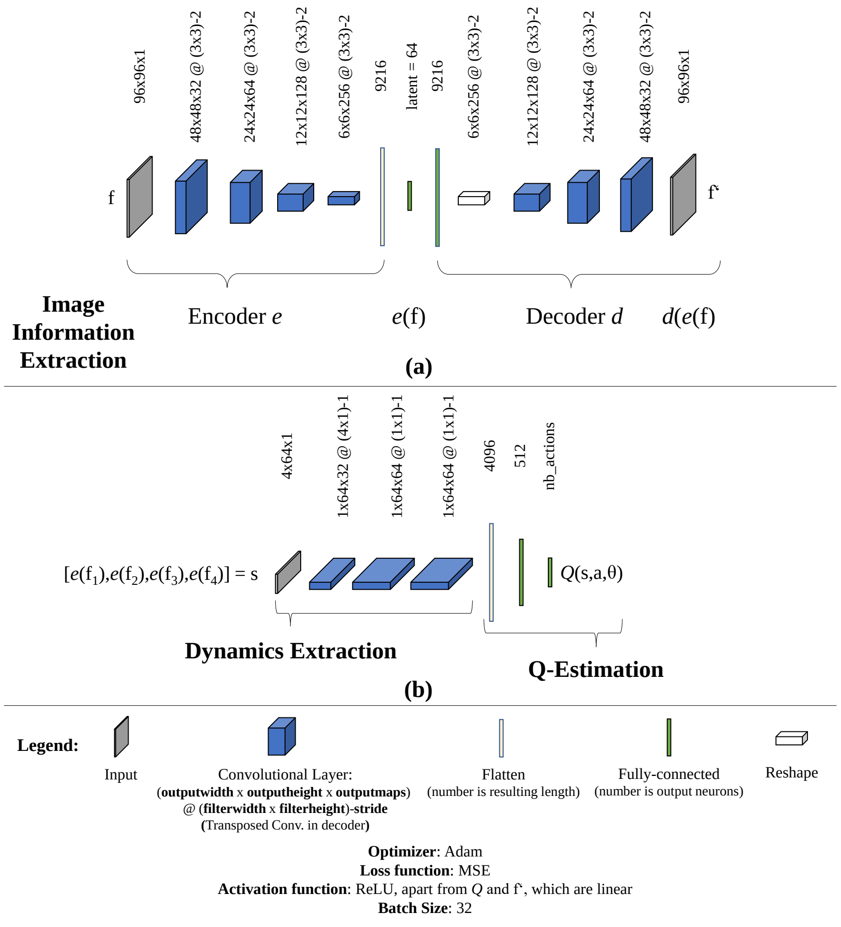


Fig. 16: General design of MATS approach: (a) vanilla DQN, (b) MATS networks, (c) sliding window in MATS dynamics extraction.

In figure 16 (a), the *vanilla* DQN as it is used in parts I and II is displayed. There are three requirements the DQN has to satisfy: (i) information has to be extracted from the input image, (ii) dynamics between the single input images need to be detected (like in what direction the player or enemy is moving) and finally (iii) use this extracted information to estimate Q.

The MATS approach is shown in figure 16 (b). The requirements are explicitly divided between several parts of the architecture. An autoencoder is used to compress image information of *single* frames into a vector. A state is then represented by a stack of four compressed frame vectors. This state is then given to the *Dynamics Extraction* CNN as input. The structure of this CNN is built on the following hypothesis: Each index of the compressed frame vectors has it's own meaning and parts of the image that it compresses. This is called *disentanglement*. For example (simplified) the first entry could represent the top left of the image while the second entry could stand for what can be seen in the bottom right. For the extraction of dynamics, it should then be possible to find out what changes in these parts of the image over four successive frames by comparing the four vectors index-wise. To accomplish this, the window width of the CNN filters is set to 1 (see figure 16 (c)). Using this constraint, each filter position only compares a certain part of the image throughout the time sequence. Like in DQN, after some convolutional layers the output is flattened to predict Q. In this setup, the autoencoder is pretrained on frames acquired by a trained agent. On the other hand, the dynamics extraction CNN and Q-estimation head form one single network that can be trained using any Q-learning algorithm.

Such a formulation of a RL agent could be useful for several reasons: For one, the different parts could be reused or fine-tuned for other tasks (transfer learning). Secondly, MATS could provide more clear representations of the data during the described multiple task steps, which could help interpreting the decisions an agent makes. Also, hierarchical architectures have shown to be able to improve RL agents [20]. Finally, using low-dimensional representation can also be beneficial when running an agent on slow hardware like for example some embedded system.



Optimizer: Adam
Loss function: MSE
Activation function: ReLU, apart from *Q* and f`, which are linear
Batch Size: 32

Fig. 17: (a) MATS autoencoder, (b) MATS dynamics extractor and Q-estimator.

6.2 Experimental Setup

As a proof-of-concept for MATS, MsPacman as the environment and DDQN for learning are considered. In figure 17 (a) the used autoencoder is shown. The image input is 96x96 which differs from the 84x84 used in parts I and II. This is because a 84x84 input is difficult to reconstruct[8]. As described the autoencoder is trained beforehand using around 100,000 frames collected during runs of a trained agent. Reconstructing Atari frames has proven to be more difficult than expected. This is mainly because a lot of important features are represented by very small parts of the image like the ball in Breakout or the food in MsPacman, each only measuring a handful of pixels. Since autoencoders are lossy compressors, these small parts are often noisy or poorly reconstructed. For Atari game frames, using a linear output and *mean squared error* as loss function has also shown to be more successful than a sigmoid output (gray values are between 0 and 1) combined with *binary cross-entropy.* In the end, the depicted architecture in figure 17 (a), using a *latent* vector size of **64**, has shown to reconstruct the input good enough to be considered for the MATS approach. Some reconstructions are displayed in figure 18.

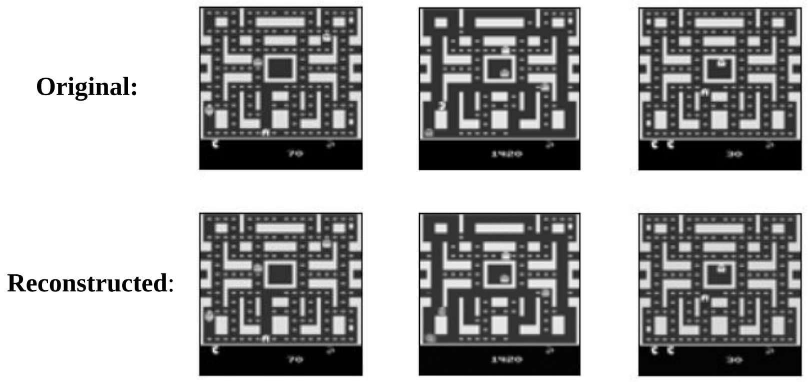


Fig. 18: Reconstructions using the MATS autoencoder.

The convolutional layers of the network shown in figure 17 (b) represent the *dynamics extractor*. Note the input with a dimension of 4x64 for the representation of four consecutive frames as vectors of size 64. Also note the filters using a width of 1 as described in figure 16 (c). Then a *Flatten* layer follows, leading to the Q-estimation part. As described, Dynamics extractor and Q-estimator form a single network that is treated as a regular DDQN and trained as such with the difference of using the encoded, stacked input instead of pixels.

[8] 84x84, using a stride of 2: Encoder dimensions: 84, 42, 21, 10; Decoder dimensions: 10, 20, 40, 80≠84.

6.3 Results

In figure 19 the return throughout the training of the MATS agent is displayed. It can be seen that the agent is able to learn, but that it's return is significantly lower (about 50 %) than that of the vanilla DQN (see figure 3 (b) for comparison).

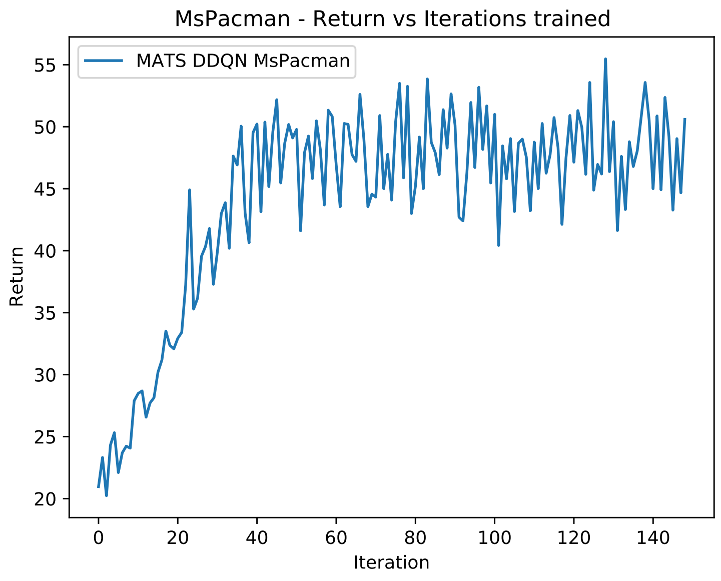


Fig. 19: Return vs. iterations trained for MsPacman using MATS with DDQN. 1 iteration = 20k steps.

To check the inner workings of this architecture, saliency maps are applied to the encoder and DDQN, see figure 20. Like explained, ideally, each latent vector index would pay attention to different parts of the frames (disentanglement). To examine whether this assumption is true, the saliency maps for multiple indices are visualized. Index 5 shows no clear focus with attention spanning all over the map. The labyrinth outline is visible though. Saliency for index 12 suggests that this index is responsible for parts in the middle of the frame. Unfortunately though, most of the encoder saliency maps look like that of index 5, which means they are very diffuse and can hardly be interpreted.

Looking at the DDQN saliency map for the action *up*, one could expect a highlighted index 12 since this part of the latent vector seems to correspond to parts of the state that are above the current player position - but this is not the case. Instead, in general the saliency maps seem to be very similar for the action *up* and *left* with only nuanced differences. Also for other states, the DDQN saliency looks similar to this one. What seems to be the case here is that the image information is not stored disentangled but rather very distributed among the latent vector.

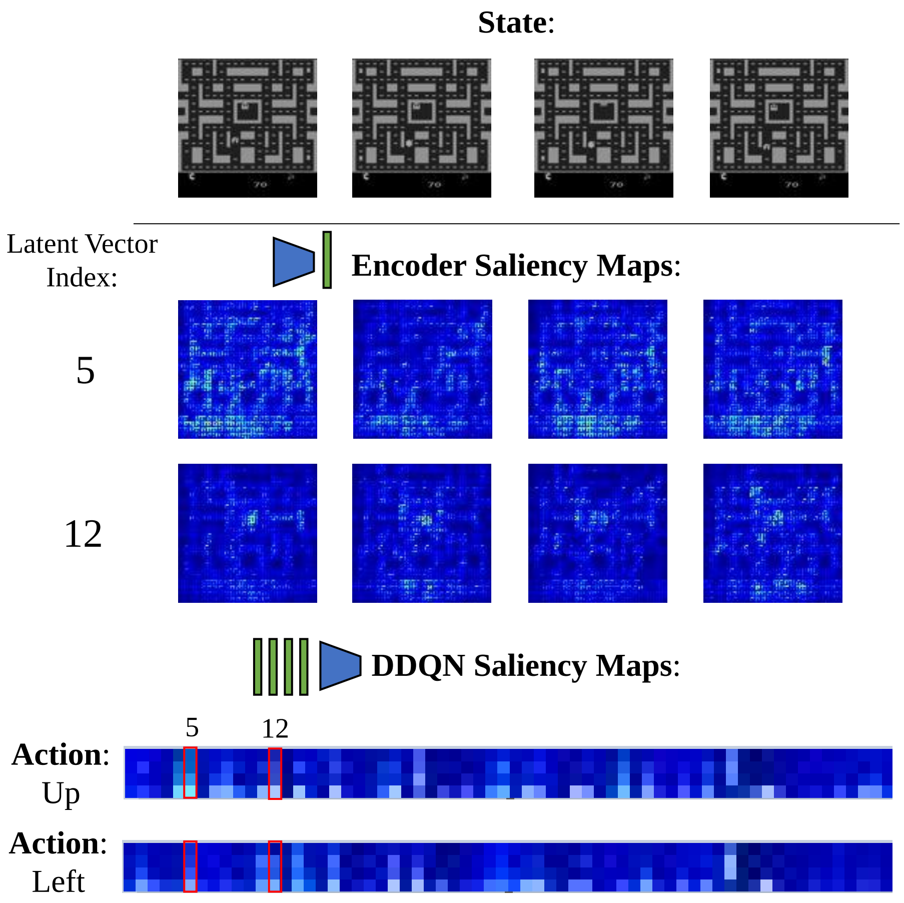


Fig. 20: Saliency maps for MATS for a certain state.

So deriving from the saliency maps, the assumption of disentangled representations in the latent vector was false. This could be a reason for the mediocre performance since the dynamics part is built on this not met assumption. However there exist approaches to disentangled representations and other possible augmentations to the NN that could help, which will be discussed in section 6.4. So making this approach work could still be possible by attempting to meet the assumptions made in the design of the MATS architecture.

6.4 Conclusion and Future Work of Part III

From a proof of concept perspective, it can be summarized that the MATS agent is able to learn in the MsPacman environment using Double Q-Learning, albeit on a lower level than the vanilla DDQN. It was shown that the vectorized frame representations are not disentangled. This could be a factor that degrades the agents performance since the dynamic extraction part counts on disentangled representations. A possible improvement could be made by using Variational [17] or β-Variational Autoencoder [14], which have shown to provide disentangled latent spaces.

Deep learning also has a tendency to show better results when a task is trained *end-to-end*, which training the autoencoder concurrently with the DDQN. This could be realized by simply updating the autoencoder every n steps with frames sampled from the replay memory. Lange et al. [22] provide a framework for this. Another option would be to directly connect the autoencoder to the backpropagation of the target error. This provides the Q-learning context to the autoencoder, which could also help the autoencoder to put more emphasis in his representations on the actually important parts of the screen (that often are small) and not just raw pixel values.

In general, it needs to be said that due to time constraints only one set of hyperparameters could be tried out. Many more variations in latent space size, number of layers, activation functions, discount factor etc. could be tried out and could lead to improvement.

After making the MATS approach work better, it could be tested how useful the three different parts are for transfer learning. For example the dynamics part could be reused for another Atari task to see how well it has generalized.

7 Summary

In the first part of this work, XAI visualization were used to investigate the training process of an Atari agent. Several findings were reported like possible reasons for *forgetting* or at what stage in training an agent is able to detect certain concepts. In the second part, it was shown how visualizations can help to understand why in some cases, transfer learning in RL is a success and why in some cases it is not. Lastly, the third part proposed a modular approach that divides the task into several subtasks assigned to different parts of the architecture. This architecture did not work as expected and visualization methods were able to give hints were improvements could be made. Every part ended with a conclusion and a perspective on possible future work.

Summarizing, XAI visualization methods have shown to be helpful in many aspects of RL. The field of RL and ML in general will continue to grow. Used systems will become larger, the environments will be more complex and they will play a growing role in everyday life. Against this background, further efforts and research in the fields of XAI are necessary to manage such complexity.

References

1. Atrey, A., Clary, K., Jensen, D.D.: Exploratory Not Explanatory: Counterfactual Analysis of Saliency Maps for Deep Reinforcement Learning. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020 (2020)
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural Machine Translation by Jointly Learning to Align and Translate. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1409.0473
3. Banerjee, B., Stone, P.: General Game Learning Using Knowledge Transfer. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence. p. 672–677. IJCAI'07, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
4. Calvo, R., Figueiredo, M.: Reinforcement learning for hierarchical and modular neural network in autonomous robot navigation. In: Proceedings of the International Joint Conference on Neural Networks, 2003. vol. 2, pp. 1340–1345 vol.2 (2003). https://doi.org/10.1109/IJCNN.2003.1223890
5. Carvalho, D.V., Pereira, E.M., Cardoso, J.S.: Machine Learning Interpretability: A Survey on Methods and Metrics. Electronics **8**(8) (2019). https://doi.org/10.3390/electronics8080832, https://www.mdpi.com/2079-9292/8/8/832
6. Choi, J., Lee, B., Zhang, B.: Multi-focus Attention Network for Efficient Deep Reinforcement Learning. CoRR **abs/1712.04603** (2017), http://arxiv.org/abs/1712.04603
7. Csáji, B.C.: Approximation with Artificial Neural Networks. Master's thesis, Eötvös Loránd University (2001)
8. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
9. Douglas, S.C., Yu, J.: Why relu units sometimes die: Analysis of single-unit error backpropagation in neural networks. In: 2018 52nd Asilomar Conference on Signals, Systems, and Computers. pp. 864–868 (2018). https://doi.org/10.1109/ACSSC.2018.8645556
10. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Teh, Y.W., Titterington, M. (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 9, pp. 249–256. PMLR, Chia Laguna Resort, Sardinia, Italy (13–15 May 2010), http://proceedings.mlr.press/v9/glorot10a.html
11. Greydanus, S., Koul, A., Dodge, J., Fern, A.: Visualizing and Understanding Atari Agents. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 1792–1801. PMLR, Stockholmsmässan, Stockholm Sweden (10–15 Jul 2018), http://proceedings.mlr.press/v80/greydanus18a.html
12. Hasselt, H.v., Guez, A., Silver, D.: Deep Reinforcement Learning with Double Q-Learning. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. p. 2094–2100. AAAI'16, AAAI Press (2016)
13. He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90

14. Higgins, I., Matthey, L., Pal, A., Burgess, C.P., Glorot, X., Botvinick, M., Mohamed, S., Lerchner, A.: beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In: ICLR (2017)
15. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786), 504–507 (Jul 2006). https://doi.org/10.1126/science.1127647
16. Kimura, D.: DAQN: Deep Auto-encoder and Q-Network. ArXiv **1806.00630** (2018)
17. Kingma, D.P., Welling, M.: Auto-Encoding Variational Bayes. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014), http://arxiv.org/abs/1312.6114
18. Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A.A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., Hadsell, R.: Overcoming catastrophic forgetting in neural networks. Proceedings of the National Academy of Sciences **114**(13), 3521–3526 (2017). https://doi.org/10.1073/pnas.1611835114, https://www.pnas.org/content/114/13/3521
19. Kruse, R., Borgelt, C., Klawonn, F., Moewes, C., Steinbrecher, M., Held, P.: Computational Intelligence: A Methodological Introduction. Texts in Computer Science, Springer, London (2013). https://doi.org/10.1007/978-1-4471-5013-8
20. Kulkarni, T.D., Narasimhan, K., Saeedi, A., Tenenbaum, J.: In: Advances in Neural Information Processing Systems
21. Käding, C., Rodner, E., Freytag, A., Denzler, J.: Fine-tuning Deep Neural Networks in Continuous Learning Scenarios. In: ACCV Workshop on Interpretation and Visualization of Deep Neural Nets (ACCV-WS) (2016)
22. Lange, S., Riedmiller, M.A.: Deep auto-encoder neural networks in reinforcement learning. In: IJCNN. pp. 1–8. IEEE (2010), http://dblp.uni-trier.de/db/conf/ijcnn/ijcnn2010.htmlLangeR10
23. LeCun, Y., Haffner, P., Bottou, L., Bengio, Y.: Object Recognition with Gradient-Based Learning. In: Shape, Contour and Grouping in Computer Vision. p. 319. Springer-Verlag, Berlin, Heidelberg (1999)
24. Manchin, A., Abbasnejad, E., van den Hengel, A.: Reinforcement learning with attention that works: A self-supervised approach. In: Gedeon, T., Wong, K.W., Lee, M. (eds.) Neural Information Processing - 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12-15, 2019, Proceedings, Part V. Communications in Computer and Information Science, vol. 1143, pp. 223–230. Springer (2019). https://doi.org/10.1007/978-3-030-36802-9_25
25. Mathew, A., Mathew, J., Govind, M., Mooppan, A.: An improved transfer learning approach for intrusion detection. Procedia Computer Science **115**, 251–257 (2017). https://doi.org/https://doi.org/10.1016/j.procs.2017.09.132, 7th International Conference on Advances in Computing Communications, ICACC-2017, 22-24 August 2017, Cochin, India
26. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari With Deep Reinforcement Learning. In: NIPS Deep Learning Workshop (2013)
27. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (02 2015), http://dx.doi.org/10.1038/nature14236

28. Opbroek, A., Ikram, M., Vernooij, M., de Bruijne, M.: Transfer Learning Improves Supervised Image Segmentation Across Imaging Protocols. IEEE transactions on medical imaging **34** (11 2014). https://doi.org/10.1109/TMI.2014.2366792
29. Parisi, G.I., Kemker, R., Part, J.L., Kanan, C., Wermter, S.: Continual lifelong learning with neural networks: A review. Neural Networks **113**, 54–71 (2019). https://doi.org/https://doi.org/10.1016/j.neunet.2019.01.012, https://www.sciencedirect.com/science/article/pii/S0893608019300231
30. Rao, Y., Lu, J., Zhou, J.: Attention-Aware Deep Reinforcement Learning for Video Face Recognition. In: 2017 IEEE International Conference on Computer Vision (ICCV). pp. 3951–3960 (2017). https://doi.org/10.1109/ICCV.2017.424
31. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In: Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 28. Curran Associates, Inc. (2015)
32. Simonyan, K., Vedaldi, A., Zisserman, A.: Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings (2014), http://arxiv.org/abs/1312.6034
33. Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1409.1556
34. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018), http://incompleteideas.net/book/the-book-2nd.html
35. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the Inception Architecture for Computer Vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 2818–2826 (2016). https://doi.org/10.1109/CVPR.2016.308
36. Vakli, P., Deák-Meszlényi, R.J., Hermann, P., Vidnyánszky, Z.: Transfer learning improves resting-state functional connectivity pattern analysis using convolutional neural networks. GigaScience **7**(12) (11 2018). https://doi.org/10.1093/gigascience/giy130
37. van Hoof, H., Chen, N., Karl, M., van der Smagt, P., Peters, J.: Stable reinforcement learning with autoencoders for tactile and visual data. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3928–3934 (2016). https://doi.org/10.1109/IROS.2016.7759578
38. Wang, J., Gou, L., Shen, H.W., Yang, H.: DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks. IEEE Transactions on Visualization and Computer Graphics **25**(1), 288–298 (2019). https://doi.org/10.1109/TVCG.2018.2864504
39. Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N.: Dueling Network Architectures for Deep Reinforcement Learning. In: Proceedings of The 33rd International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 48, pp. 1995–2003. PMLR, New York, New York, USA (20–22 Jun 2016), http://proceedings.mlr.press/v48/wangf16.html
40. Zhang, J., Yu, H., Xu, W.: Hierarchical Reinforcement Learning by Discovering Intrinsic Options. In: International Conference on Learning Representations (2021)
41. Zhu, Z., Lin, K., Zhou, J.: Transfer Learning in Deep Reinforcement Learning: A Survey. CoRR **abs/2009.07888** (2020), https://arxiv.org/abs/2009.07888

A Used Frameworks

The project was realized using Python 3.8.

```
requirements.txt:

matplotlib==3.3.4
keras_rl2==1.0.4
tf_keras_vis==0.5.5
tensorflow==2.2.0
gym==0.18.0
opencv_python==4.5.1.48
numpy==1.20.1
Pillow==8.2.0
```

B Hyperparameters and Settings

The following hyperparameters and settings in `Keras-RL2` were used for every experiment.

```
memory= SequentialMemory(limit=1000000,
                         window_length=WINDOW_LENGTH)

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(),
                              attr='eps',
                              value_max=1.,
                              value_min=.1,
                              value_test=.05,
                              nb_steps=1000000)

dqn = DQNAgent(model=model,
               nb_actions=nb_actions,
               policy=policy,
               memory=memory,
               processor=processor,
               nb_steps_warmup=50000,
               gamma=.99,
               train_interval=4,
               delta_clip=1.,
               batch_size=32)

dqn.compile(optimizer=Adam(lr=.00025), metrics=['mae'])
```

Apart from that, `Tensorflow`, `Keras` und `Keras-RL2` default options were used.

For saliency map visualizations, `tf-keras-vis` was used with the following settings:

```
saliency = Saliency(model, clone=False)
saliency_map = saliency(loss,
                        states,
                        smooth_samples=20,
                        smooth_noise=0.20)
```