HAW HAMBURG

MASTER THESIS
Rico Helmboldt

Evaluating Saliency Map Methods in Reinforcement Learning

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Rico Helmboldt

Evaluating Saliency Map Methods in Reinforcement Learning

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr.-Ing. Marina Tropmann-Frick
Supervisor: Prof. Dr.-Ing. Stephan Pareigis

Submitted on: 28 Oktober 2022

Rico Helmboldt

Thema der Arbeit

Evaluating Saliency Map Methods in Reinforcement Learning

Stichworte

Erklärbare KI, XAI, Salienzkarten, Pixel Attribution, Deletion, Verstärkendes Lernen, Atari, DQN, Vanilla Gradients, SmoothGrad, GradCAM, LRP, LIME, RisE

Kurzzusammenfassung

Die Salienzkartenmethoden Vanilla Gradients, SmoothGrad, (guided-) GradCAM, LIME und RisE werden quatitativ in Bezug auf ihre Korretheit evaluiert. Dies wird in einem verstärkenden Lernen Setting mit DQN und Atari Breakout umgesetzt. Zur quantitativen Evaluierung wird Deletion verwendet, wobei der Performanceverlust des Agenten gemessen wird. Es wird gezeigt, dass LRP die Korrekteste Methode ist, aber zusammen mit RisE verwendet werden sollte, um eine vollständgere Erklärung zu erhalten.

Rico Helmboldt

Title of Thesis

Evaluating Saliency Map Methods in Reinforcement Learning

Keywords

Explainable AI, XAI, Saliency maps, Pixel attribution, Deletion, Reinforcement learning, Atari, DQN, Vanilla Gradients, SmoothGrad, GradCAM, LRP, LIME, RisE

Abstract

The saliency map methods Vanilla Gradients, SmoothGrad, (guided-) GradCAM, LIME and RisE are evaluated quantitatively with regards to their correctness. This is done in a reinforcement learning setting with DQN and Atari Breakout. As means to evaluate them quantitatively, Deletion is applied to measure how quickly the performance dwindles. It is shown that LRP is the most correct saliency map method, but should be used together with RisE for a more complete explanation.

Contents

1 Introduction

The introduction starts with a motivation given in 1.1. Then, the research objective and the research questions are given in 1.2. Related works to this research are presented in section 1.3.

1.1 Motivation

Machine learning models are deployed effectively in various tasks ranging from classification to natural language processing. Many machine learning models are implemented via neural networks. While their performance and results are satisfactory, they are not easily interpretable due to their nature and can be seen as black-boxes. Saliency maps are a way to gain insights into predictive models when images are used as input. They highlight important parts of the input image with regards to the chosen output of the model. This methodology is comparable to feature importance with non-image inputs ([1]), but instead of giving important features a high relevance value, the important pixels are highlighted. There are many saliency map methods which are based on many different ideas. Using different saliency map methods may result in different highlighted areas of the input image resulting in different explanations, but they all aim to highlight the relevant image areas. The goal of this research is to determine which of these methods produces the most correct saliency maps. Correctness describes how faithful the saliency map is with regards to the model ([2]): A highly correct saliency map accurately highlights the exact pixels which were important for the models decision.

1.2 Research Objective

The goal of this research is to determine quantitatively which of the considered saliency map methods works best in a reinforcement learning setting with regards to their correct-

ness. Correctness in the domain of explainable AI describes how faithful the explanation is with respect to the black-box ([2]), with the key idea "nothing but the truth". The reinforcement learning setting is set to be the game Atari Breakout, played by a double DQN agent, which are explained in chapter 2. The considered saliency map methods include vanilla gradients, SmoothGrad, GradCAM, Guided GradCAM, LRP, LIME and RisE, which are explained in section 3.2. To evaluate the saliency map methods quantitatively, deletion (explained in section 3.3) is applied to them. Furthermore, potential differences in the highlighted areas between the various saliency map methods are examined. One saliency map method might highlight different parts of the image than another, then it necessary to compare their correctness and potentially examine further in case both are correct. The research objective includes the following questions:

1. Which saliency map method has the highest degree of correctness in a reinforcement learning setting?
2. Are there differences in the highlighted areas between the saliency map methods?
3. Which saliency map method is recommended to be used?

To answer these questions, an introduction into reinforcement learning (chapter 2) is given first before the saliency map methods are explained in chapter 3.

1.3 Related Works

A broad introduction into reinforcement learning is given by [3]. The general architecture of reinforcement learning is explained, but a large emphasis is laid upon the approaches to tackle the reinforcement learning task. This ranges from temporal difference methods, which learn after every step with an estimated reward, to monte-carlo methods, which learn only when all true rewards are received.

Likewise, a broad introduction into explainable AI is given by [1]. Various approaches to explain machine learning models are described, ranging from interpretable models which are explainable themselves, to model-agnostic methods which try to give explanations from the outside. Among other topics, pixel attribution methods as used in this research are described.

An example of a interpretable model is Hierarchical Actor Critic (HAC) proposed by [4]. HAC works in a reinforcement learning setting, where multiple hierarchical layers divide

the policy into sub-policies. These policy layers have individual (sub-)goals, thus making it possible to track the strategy of the agent by its sub-goals. HAC is one, but not the only, approach to learn the policy next to the explanation.

Further methods which aim to have a higher degree of interpretability, but are not interpretable models per se, include State Representation Learning (SRL, [5]). SRL learns lower dimensionality state representations with high meaningfulness by processing high dimensionality observations. An example would be the to learn a (x, y) position from raw pixel values as observation. Now, variations in the environment based on the agents decision can be captured which allows the extrapolation of explanations.

Next to the saliency map methods used in this research, there are further methods like DeconvNet [6], Integrated Gradients [7], Feature Ablation [8] or SHAP [9]. DeconvNet is specialized for explaining convolutional neural networks (CNN) and works by building a deconvolution network (the DeconvNet), which reverses the convolution and pooling steps performed by the CNN. After a specialized training, the output of the DeconvNet highlights the pixels of the image which were striking during the convolution and pooling of the CNN. Integrated Gradients works by calculating interpolations between the original input image and an all-black image, resulting in dimmed images. They are subsequently fed through the neural network classifier and the gradients to each pixel are observed to calculate the relationship between the changes to a pixels and the predicted output: The gradient informs how relevant each pixel is with regards to the output, which can then be colorized to produce a saliency map. With Feature Ablation, the input features are split into several groups which are perturbed together to determine the importance of each group. In case of images, several pixels are bundled together to form a group (more groups lead to a higher computation time). The importance of each group can then be highlighted as saliency map. SHAP uses a game theoretic approach and is based on shapley values: Each feature is a player, the machine learning model prediction is the payout and shapley values are a method from coalitional game theory telling how to fairly distribute the payout among the players. Each feature gets assigned to a shapley value indicating how much the feature contributed to the specific outcome. Since SHAP forms all possible coalitions (permutation of features), it can be very computationally expensive.

[10] uses the deletion procedure, as used in this research, to evaluate RisE saliency maps on RGB image classification. The methodology works as follows: Investigate a saliency map method by iteratively removing the most relevant pixels and observe how

much worse the model performs. This research transfers this idea into the reinforcement learning setting, where the drop in performance of the agent is observed. Further, 7 more saliency map methods next to RisE are evaluated.

Other approaches to quantitatively evaluate explainable AI methods include [11], which focuses on hidden malicious functionalities inside the machine learning model. These so-called backdoor trigger patterns are the key reason why the machine learning model misclassifies specific instances, and an explainable AI methods should spot them during their explanation generation. For this, three metrics are introduced which quantify how well an explainable AI method covers these backdoor trigger patterns.

There are contributions which focus on the qualitative evaluation of explainable AI methods. For example, [12] compare rule-based and example-based explainable AI methods and evaluate them qualitatively on humans. They found that extracting rules give a small insight into the machine learning model workings, but both approaches persuade humans into following the advice of the explanation, even if incorrect. This is because both approaches only give explanation for single specific instances (local explanation, see section 3.1), but the underlying machine learning model workings.

2 Reinforcement Learning

This chapter gives an introduction into the relevant ares of reinforcement learning. For this, the task is described in section 2.1, followed by the explanation of the reinforcement learning structure in section 2.2 and the used algorithms in section 2.3. The architecture of reinforcement learning consists of an environment in which an agent navigates to accomplish some task. These tasks are modeled as Markov Decision Process (MDP).

Markov Decision Process

The Markov Decision Process (MDP) defines the reinforcement learning environment and is a 4-tuple with:

- $\boldsymbol{S}$ **:** The state space S contains all the possible states $s \in S$ the agent can be in. The state space is discrete if it is finite, where each state can be assigned to a number. Otherwise, it is continuous, where the state has to be modeled as a vector. The game chess has a discrete state space, as it has a finite set of combinations of the figures. The mountain car task has a continuous state space, as the car can be at any position (x, y), where x and y are floating point numbers.
- $\boldsymbol{A}$ **:** The state space A contains all the possible actions $a \in A$ the agent can perform. The action space is discrete if it is finite, where each action can be assigned to a number. Otherwise, it is continuous, where the action has to be modeled as a vector. The game chess has a discrete action space, as it has a finite set of available actions. The continuous mountain car task has a continuous action space, as the car can be accelerated by any non-discrete amount (floating point).
- $\boldsymbol{P_a(s, s')}$ **:** The probability distribution of reaching the next state s' from s with action a.
- $\boldsymbol{R_a(s, s')}$ **:** The reward when going from state s to the next state s' with action a.

This 4-tuple defines the environment of the reinforcement learning task. It is responsible

for giving out observations (current states) and rewards based on received actions. The probability distribution $P_a(s, s')$ is the logic of the environment dynamics, be it some physical system or a video game, and can be stochastic or deterministic.

Policy

The policy π defines the reinforcement learning agent and chooses an action $a_t \in A$ based on an observation $s_t \in S$ at time step t. This action a_t is then passed to the environment which return the next state $s_{t+1} \in S$. The optimal policy π^* chooses the actions so that it receives the highest summed reward over the course the MDP.

2.1 Environment: Atari Breakout

The game Breakout is an Atari game from 1976 where the goal is to achieve the highest possible score by hitting bricks with a ball. The player has to keep the ball on the game field, which he achieves by controlling a platform to bounce the ball back up. If the ball flies past the platform, a live is lost and a new ball spawns. The player has 5 lives per game, loosing all lives means game over.

Figure 2.1: The game Atari Breakout in its beginning phase. The player steers the platform at the bottom left or right to shoot away the bricks at the top with the bouncing ball.

Here, Breakout is the environment of the reinforcement learning task, with the goal for the agent to achieve the highest possible score. Since the observations are the frames

of the games with pixel values, the state space is continuous while the action space is discrete with 4 actions: NOOP (do nothing), FIRE (spawn new ball if none on the field), LEFT, RIGHT. To cope with a continuous state space and a discrete action space, a suitable algorithm has to be used for the agent.

2.2 Reinforcement Learning Structure

The agent tries to accomplish a task in the given environment with state s_t by choosing the actions a_t at every time step t it deems to be the best.

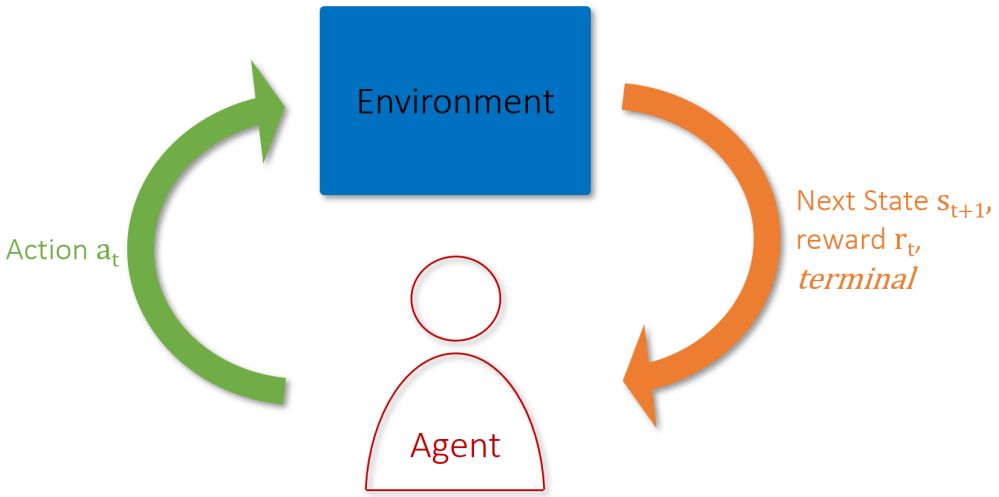


Figure 2.2: The agent chooses an action based on the current state, passes it to the environment and receives the next state next.

For this, the agent receives information from the environment (see figure 2.2), including the new game state s_{t+1} as frame-stack of the last few game frames, a reward r_t based on the performed action a_t in state s_t and a *terminal* flag indicating whether this step lead to a game over.

Frame-Stack

A frame-stack represents a game state s_t and includes the last few game frames in grayscale. More accurately, the frame-stack includes the game frames from time-step t to $t-(k-1)$, where k defines how many frames are included. k is chosen to be $k=4$, as the frame-stack then includes enough information to be able to estimate the flight path of the ball while not providing too much unnecessary information.

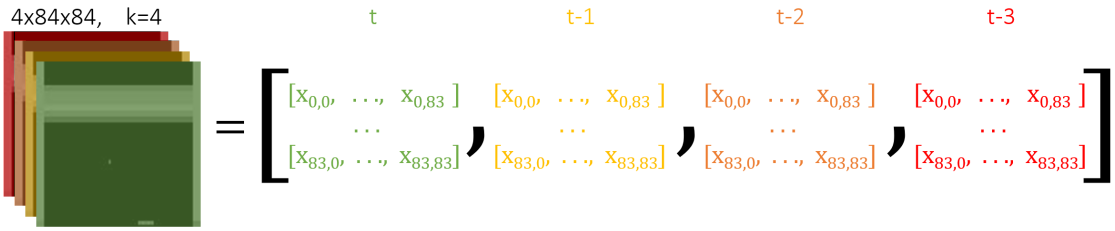


Figure 2.3: A frame-stack with $k = 4$ includes the newest frame of size 84x84 as well as the three previous frames. The resulting tensor has the shape 4x84x84.

The frames are converted to grayscale as shown in figure 2.3 so that convolutional layers are applicable to work on the frame-stack later (see section 5.1.1).

2.3 Agent Algorithms

The agent has to control the platform in Breakout so that it achieves the highest possible score by hitting the bricks, if possible without loosing lives. The agent receives continuous states and outputs one of the 4 discrete actions (do nothing, fire, left, right, see section 2.1). The algorithm used in this work is double DQN (see section 2.3.3), an extension of Deep Q-Network (DQN, section 2.3.2), which is based on Q-Learning (see next section 2.3.1).

On-policy vs. off-policy

Algorithms can be on-policy or off-policy. Off-policy algorithms use a different policy for collecting samples from the environment than the one that is actually used when evaluating or the training is finished. For example, Q-Learning has a greedy policy (always choose the action with the highest Q-value) but when collecting samples, an ϵ-greedy policy is used. The ϵ-greedy policy chooses a random action with a given chance defined by the current value of ϵ and a greedy action otherwise. On-policy algorithms like SARSA ([13]) use the same policy for both cases.

Online vs. offline

Online algorithms collect samples during training while offline algorithms have a fixed, static dataset. Online algorithms tend to have access to generally more and more diverse data as new samples can be collected and added to the dynamic dataset, while offline algorithms train faster since there is no communication needed with the environment.

There are many more categorizations like model-free vs. model-based, value-based vs. policy based and so on, which are explained in [3] but these categorizations do not matter for this work.

2.3.1 Q-Learning

Q-Learning is an online off-policy algorithm which aims to learn the optimal policy $\pi*$ by assigning values to state-action pairs, called Q-values, which describe how good an action a in state s is. For every possible state-action pair, such a Q-value is stored in the Q-table. The Q-table has all distinct states on one axis and all possible actions on the other axis. Thus, Q-Learning is only applicable with both a discrete state and action space. The Q-values in the Q-table are initialized with 0 and then iteratively updated during training. After the training is finished, the action with the highest Q-value is taken greedily at each time-step to follow the best learned policy. This policy is optimal if the Q-values are assigned with appropriately good repressiveness. Q-Learning is online and off-policy: For collecting samples from the environment, an ϵ-greedy policy is used and these samples are stored in a replay-buffer, of which mini-batches are sampled later for training.

Replay Buffer

The replay buffer stores samples as 5-tuple consisting of the state s_t, the chosen action a_t, the resulting next state s_{t+1}, the received reward r_t and the terminal flag $terminal$ informing about whether this was the last step terminating an episode in the environment. Random samples of these 5-tuples $s_t, a_t, s_{t+1}, r_t, terminal$ are samples from the replay buffer as mini-batch used for training.

Training Loop

The training starts by filling the replay buffer with samples which are generated using random actions. This is done to have a basis for sampling mini-batches. Then, the training loop is run for a given amount of episodes, consisting of 2 phases: Collection and training.

In the collection phase, an action a_t at time-step t is chosen according to the ϵ-greedy policy which is delegated to the environment. Then, the next state s_{t+1}, the reward r_t and the terminal flag (indicating whether the game is over) are observed. The information of this sample is stored in the replay buffer as 5-tuple $s_t, a_t, s_{t+1}, r_t, terminal$.

In the training phase, a mini-batch is sampled from the replay-buffer. The size if the mini-batch is given as hyper-parameter. Then, the Q-values are updated according to the Bellman equation for each sample of the mini-batch:

$$Q_{\text{new}}(s_t, a_t) = Q(s_t, a_t) + \lambda \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.1}$$

where $Q(s_t, a_t)$ is the old Q-value regarding state s_t and and action a_t, the term is the braces evaluates to the new target Q-value, λ is the learning rate given as hyper-parameter, r_t is the reward given going from state s_t to the next state s_{t+1}, γ is the discount factor given as hyper-parameter and $\max_a Q(s_{t+1}, a)$ is the highest Q-value of all possible actions within the next state s_{t+1}. The learning rate λ defines how much the Q-value is updated towards the target Q-value. A higher learning rate may lead to faster convergence but also to instability while a lower learning rate is more stable but lets Q-learning converge slower. The discount factor $0 \leq \gamma < 1$ defines how much the future states are weighted in the target Q-value estimation. A discount factor of 1 would lead to an infinite sum, as all future Q-values are included non-discounted. Setting the discount factor close to 1 means that the future plays a bigger role in the estimation of the Q-value, while a discount factor of 0 means that the Q-value is sorely based on the reward given in this sample r_t.

After training, the game is played by always choosing the action with the highest Q-value regarding the current state s_t. Q-Learning is applicable with discrete state and action spaces, however, if either space is continuous, the algorithm has to be adapted. In case of Atari Breakout, the action space is discrete but the state space is continuous. Deep Q-Network can then be used as explained in the next section 2.3.2.

2.3.2 Deep Q-Network

Deep Q-Network is an online off-policy algorithm, like Q-learning, but adjusts it so that it can work with continuous state spaces

Q-Network

DQN adjusts Q-Learning by introducing a neural network to approximate the Q-table. The neural network receives the state as input and outputs the Q-values for every action.

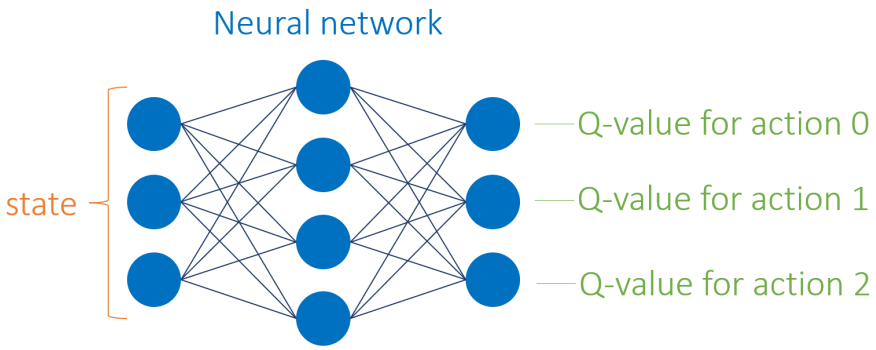


Figure 2.4: The neural network in DQN takes the continuous state as input and outputs the Q-values for each possible action.

This architecture allows for a continuous state space, as every feature of a continuous state is placed at one position of the input vector (see figure 2.4). The output layer of the Q-network consists of as many neurons as there are actions, where the activation value at each neuron corresponds to the Q-value of this action. In the Atari Breakout setting, there have to be 4 output neurons for the actions NOOP, FIRE, LEFT and RIGHT. Given one state, one forward pass through the Q-network thus estimates the Q-values for all actions with regard to the input state. However, this architecture also limits the action space to discrete, since a continuous action space would require an infinite amount of neurons in the output layer. Using such a Q-network is necessary when the state space is either continuous or so large that it becomes infeasible to store all the Q-values in a table like in Q-learning.

Target Network

Next tot he normal Q-network, DQN inducts a target network, which has the exact same architecture as the Q-network and is used to calculate the target Q-values in the Bellman equation. The target network itself isn't trained, instead, the weights are copied from the Q-network from time to time. This interval is specified as hyper-parameter. This methodology improves the stability of DQN by keeping the targets stable for a specified amount of time.

Replay Buffer

DQN uses a replay buffer, in which recorded transitions are stored, which are later sampled to train the Q-network. One transition at each time-step t is a 5-tuple consisting of the state s_t, the chosen action a_t, the resulting next state s_{t+1}, the received reward r_t and the terminal flag *terminal* informing about whether this was the last step terminating

an episode in the environment. The use of a replay buffer further improves the stability of DQN.

Whenever a task includes a continuous action space which cannot be discretized in a sensible way, DQN is no longer a suitable choice as reinforcement learning algorithm. Then, actor-critic architectures can be used, which adjust the algorithm and neural networks to be able to handle continuous state and action spaces. Since Atari Breakout has a continuous state space but a discrete action space, DQN is the algorithm of choice.

Algorithm

Algorithm 2: Deep Q-Network

Input: θ ▷ Initial network parameters

$\bar{\theta} \leftarrow \theta$ ▷ Initialize target network weights

$\mathcal{R} \leftarrow \emptyset$ ▷ Initialize replay buffer

for each iteration **do**

$a_t \leftarrow \begin{cases} \text{randInt}(|A|) \textbf{ if } \text{randFloat}(0,1) \leq \epsilon \\ \underset{a}{\text{argmax}}(Q_\theta(s_t, a)) \textbf{ otherwise} \end{cases}$ ▷ ϵ-greedy action

$\epsilon \leftarrow \max(\epsilon - \Delta\epsilon,\ \epsilon_{\min})$ ▷ Reduce ϵ

$s_{t+1} \sim \rho(s_{t+1}|s_t, a_t)$ ▷ Sample next state from environment

$\mathcal{R} \leftarrow \mathcal{R} \cup (s_t, a_t, s_{t+1}, r(s_t, a_t), terminal)$ ▷ Store transition in replay buffer

if iteration % train interval = 0 **do**

$(s_i, a_i, r_i, s_{i+1}, terminal_i) \sim \mathcal{R}$ ▷ Sample N transitions

$y_i = r_i + (1 - terminal) \cdot \gamma \max(Q_{\bar{\theta}}(s_{i+1}))$ ▷ Calculate targets

$\mathrm{L} = (y_i - Q_\theta(s_i)[a_i])^2$ ▷ Update Q-network with loss

end if

if iteration % target update interval = 0 **do**

$\bar{\theta} \leftarrow \theta$ ▷ Set target network weights

end if

end for

Output: θ ▷ Optimized parameters

Table 2.3.2 describes how the DQN algorithm works. As setup, the Q-network and target network are initialized, an empty replay buffer is set up, and the hyper-parameters are set (see figure 5.3). Then, the training loop begins. The training loop consists of 3 phases: collection, training and target network update.

In the collection phase, an ϵ-greedy action is chosen to perform a step in the environment. The chance of choosing a random action is defined by the current ϵ value, otherwise the action with the highest Q-value according the Q-network with respect to the current state s_t is chosen. The action is passed to the environment and the resulting next state s_{t+1}, the reward r_t and the terminal flag are observed. The complete transition is then added to the replay buffer.

The training phase is entered every few iterations, defined by the train interval hyper-parameter. In this phase, the Q-network weights are updated. For this, a mini-batch is sampled from the replay-buffer. The size of the mini-batch is defined by the batch-size hyper-parameter. For each sample in the mini-batch, the target Q-value is calculated:

$$y_i = r_i + (1 - terminal_i) \cdot \gamma \max(Q_{\bar{\theta}}(s_{t+1})) \tag{2.2}$$

where r_i denotes the reward for sample i, γ is the discount factor given as hyper-parameter and $\max(Q_{\bar{\theta}}(s_{t+1}))$ is the highest Q-value regarding the next state s_{t+1} according to the target network $Q_{\bar{\theta}}$. The term $(1 - \text{terminal}_i)$ evaluates to 0 if sample i is terminal, otherwise 1. This ensures that in case of a terminal transition, the reward r_i is directly taken as target, ignoring the discounted highest Q-value of the next state. The target network or Q-network are able to estimate Q-values based on the next state s_{t+1}, however their values are irrelevant since this next state s_{t+1} is the terminal one from where there is no need to look further into future states or actions. With the targets, the loss is calculated:

$$\mathrm{L} = \frac{1}{N} \sum_{i=0}^{N} (y_i - Q_\theta(s_i, a_i))^2 \tag{2.3}$$

where y_i is the target Q-value y of sample i (see equation 2.2) and $Q_\theta(s_i, a_i)$ is the actual current Q-value estimate regarding state s_i and action a_i. The Q-network weights can then be optimized with that loss.

The target update phase is entered every few iterations, defined by the target update interval hyper-parameter. This phase copies the current Q-network weights into the target network, thus overwriting the target network weights:

$$\bar{\theta} \leftarrow \theta \tag{2.4}$$

where θ are the network weights of the primary Q-network and $\bar{\theta}$ are the weights of the target network.

After the training loop, the Q-network with its optimized weights is returned as result. DQN includes features that are essential for its performance and stability like the replay buffer and the target network, but they can be further improved with later ideas like double DQN.

2.3.3 Double DQN

Algorithm 2: Double DQN

Input: θ ▷ Initial network parameters

$\bar{\theta} \leftarrow \theta$ ▷ Initialize target network weights

$\mathcal{R} \leftarrow \emptyset$ ▷ Initialize replay buffer

for each iteration **do**

$a_t \leftarrow \begin{cases} \text{randInt}(|A|) \textbf{ if } \text{randFloat}(0,1) \leq \epsilon \\ \underset{a}{\text{argmax}}(Q_\theta(s_t, a)) \textbf{ otherwise} \end{cases}$ ▷ ϵ-greedy action

$\epsilon \leftarrow \max(\epsilon - \Delta\epsilon,\ \epsilon_{\min})$ ▷ Reduce ϵ

$s_{t+1} \sim \rho(s_{t+1}|s_t, a_t)$ ▷ Sample next state from environment

$\mathcal{R} \leftarrow \mathcal{R} \cup (s_t, a_t, s_{t+1}, r(s_t, a_t), terminal)$ ▷ Store transition in replay buffer

if iteration % train interval = 0 **do**

$(s_i, a_i, r_i, s_{i+1}, terminal_i) \sim \mathcal{R}$ ▷ Sample N transitions

$y_i = r_i + (1 - terminal_i) \cdot \gamma \left(Q_{\bar{\theta}}(s_{t+1}, a)\right)$ $|_{a=\max_{a'} Q_\theta(s_{t+1}, a')}$ ▷ Calculate targets as in figure 2.5

$\mathrm{L} = (y_i - Q_\theta(s_i)[a_i])^2$ ▷ Update Q-network with loss

end if

if iteration % target update interval = 0 **do**

$\bar{\theta} \leftarrow \theta$ ▷ Set target network weights

end if

end for

Output: θ ▷ Optimized parameters

Vanilla DQN suffers from overestimation of Q-values, which leads to the execution of a wrongly judged-well strategy before realising that it was in fact not so good based on the received rewards. Double DQN improves the stability by changing the target Q-value calculation, the highlighted line in table 2.3.3. It differs from vanilla DQN in the last part

of equation 2.2. Vanilla DQN takes the highest Q-value of the next state s_{t+1} according to the target network $Q_{\bar{\theta}}$ (see equation 2.2), whereas double DQN uses both the primary Q-network and the target network together to calculate the target.

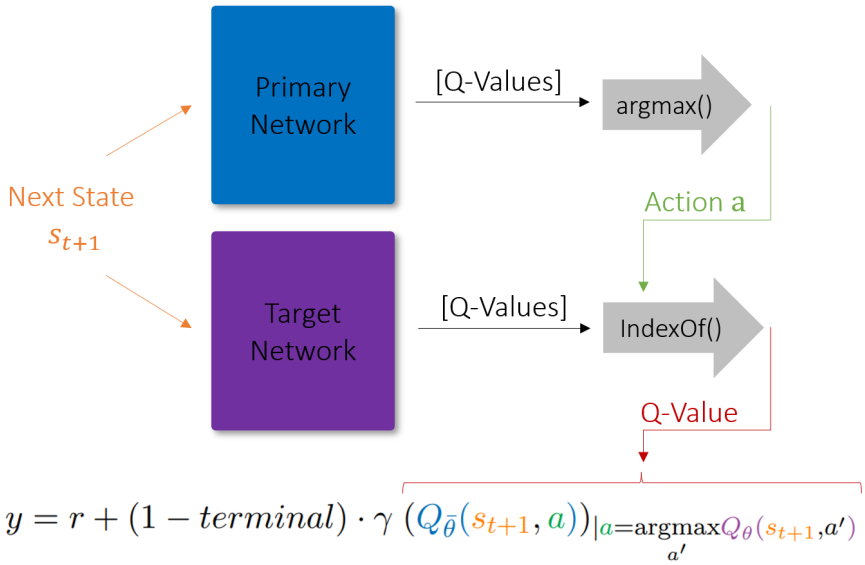


Figure 2.5: The calculation of target Q-values with double DQN. The Q-value of the next state is estimated by the target network with the action regarding that Q-value given by the highest Q-value of the primary network.

The Q-value of the next state in the last part of the equation seen in figure 2.5 is put together from the Q-value estimate from the target network regarding the next state s_{t+1}, like in vanilla DQN, but instead of taking the highest Q-value, the Q-value of action a is taken, where a is the action with the highest Q-value according to the primary Q-network $\underset{a}{\text{argmax}} Q_{\theta}(s_{t+1})$. The Q-value estimate still comes from the target network, but the action regarding the Q-value to take is chosen by the primary Q-network.

3 Explainable AI

With the relevant reinforcement learning descriptions given in chapter 2, an introduction into explainable AI is given: Various categories are described in section 3.1 before the actual saliency map methods are explained in section 3.2.

Black Boxes

Explainable AI (XAI) includes a set of methods and tools to explain the decision-making of machine learning models such as neural networks, which are by nature not directly interpretable. They can be considered as black-box, where only input and output can be observed, but the internal mechanisms of the black-box remain hidden. XAI tries to give explanations for the behaviour of the black-box model. There are a few machine learning methods that are interpretable by nature like regression trees, which circumvent the need for an additional XAI system by not being a black-box in the first place. Also, there exist machine learning algorithms which give explanations by themselves ([4], [14]), but they are developed with explanations in mind and fall in the category of intrinsic XAI methods.

Human Demands

Machine learning is used in many domains to support or replace human labour. While it might be sufficient to just apply some machine learning model in some domains, others require more than just a good performing machine learning model. A model that predicts upcoming electronic malfunctions of cloud server parts has to have a good performance as its only requirement. Meanwhile, a model that suggests medical treatments for patients also has to be trustworthy. A doctor is trustworthy by explaining his reasoning and being empathetic. A machine learning model is probably not empathetic and thus has to give excellent reasoning. The more severe a machine learning model influences peoples lives, the higher the need for explanations is [15]. Patients will hardly accept a life-dependent decision without good reasoning, especially if the decision is suggested by a machine learning model.

Political Demands

Not only does the human demand motivate the use of XAI systems, but also some legal authorities start to look at it. The European General Data Protection Regulation (GDPR; see https://gdpr.eu/) dictates that the use of black-box models (in relevant domains) must come with reasonable explanations.

XAI in the Future

The scientific interest in XAI went up in the past and will probably do so in the future [16]. One reason is that machine learning models are assigned to more and more tasks, which include more and more where human or political demands require XAI. Another reason is that the need for XAI persists even when the tools used for AI change: The X is independent from AI. The writing X-AI would make more sense if it weren't so clumsy. DeepMind introduced their new "Generalist Agent" [17], which solves a variety of tasks, from text completion over physics simulation to Atari games, based on a transformer (also known as Attention [18]). However, this powerful and flexible AI is still, or precisely for this reason, not interpretable. Regardless of the used technique to train an AI, the need for XAI persists.

XAI provides methods and tools to understand black-box models. The various methods fall into a few categories, which are described in section 3.1.

3.1 Explainable AI Categories

While all XAI methods aim to explain the behaviour of a black-box model, they can differ greatly in their philosophy. The main differentiation comes from whether they are model-specific or model-agnostic and whether they give local or global explanations. Some of the investigated saliency map methods are model-specific (gradient-based methods in section 3.2.1) while others are model-agnostic (perturbation-based in section 3.2.2), however all of them provide local explanations.

3.1.1 Model-Specific Approaches

Model-specific XAI approaches make use of knowledge of the internals of the black-box model. Thus, they peek inside the black-box and utilize information within it. In this case, the assumed black-box is actually a white-box from the perspective of the XAI

approach. Using this kind of XAI, the explainer has to suit the model, as not all model-specific methods work with every type of model. For example, a model-specific XAI method may only work with neural networks, and might further restrict the types of layers the neural networks contains. Some model-specific approaches are part of the model itself, known as intrinsic methods or interpretable models [1], thus they have to be incorporated during the deployment of the model and can't be added later on. Nevertheless, using the internals of the black-box model gives model-specific XAI approaches more information to work with than model-agnostic approaches.

3.1.2 Model-Agnostic Approaches

Model-agnostic XAI approaches make statements about the black-box model by carefully observing the outputs with regards to the inputs. They can thus be applied to all black-box models, since the functionality of the black-box is irrelevant. The black-box could even be a human classifying an input to some output classes, it wouldn't matter for model-agnostic approaches. The model-agnostic XAI method estimates the actual model workings as best as possible and then generates an explanation for this estimation. Making estimations of the actual model workings and not using some internal information leads to a discrepancy between the estimation and the actual model, which is also known as model soundness. The goal is to keep the model soundness as low as possible. Model-agnostic XAI approaches can be added post-hoc to any machine learning system: Since they only require input-output pairs of the black-box, they work with any kind of model and the model-agnostic XAI method can be added later.

3.1.3 Local and Global Explanations

A local explanation explains exactly one instance: A single input-output pair is investigated, the local explanation gives insights into why this output came about with regards to these specific inputs. Local explanations are favored whenever a specific decision should be explained. For every instance which should be explained, the XAI method has to construct a specific and only for this instance valid explanation.

A global explanation explains all instances in a given dataset. It captures the general relationship between inputs and outputs, which is favored whenever the general functionality of the model should be explained. However, a global explanation may be too

imprecise when a single specific instance should be explained. An adequate amount of local explanations can also be aggregated into a global explanation.

3.2 Saliency Maps

A saliency map has a slightly different meaning depending on the application domain. In computer vision, a saliency map is an image that highlights the regions on which people's eyes focus first. These regions can be highlighted in various ways, for example by masking them or overlaying a heatmap over the original image. These artificially engineered saliency maps are not the same as the ones constructed by biological or natural vision. The V1 Saliency Hypothesis [19] proposes, that the primary visual cortex (V1) in the brain of primates constructs a saliency map which helps to guide the attention towards interesting elements in their visual field. In deep learning, there exists no unified definition for saliency maps (also known as sensitivity map or pixel attribution map). However, the understanding of this concept is very consistent throughout the literature ([1], [20], [21], [22]), which makes it possible to condense a definition.

Definition 3.2.1. A saliency map highlights the areas of an input image, which were decisive regarding the output/decision of the model. The saliency map layers highlights on top of the input image, so that relevant image regions can be identified.

Feature attribution gives explanations by attributing a relevance to each input feature regarding the neural network output. Features which were decisive for the output of the model get assigned a high relevance, while features which didn't have an impact get assigned a low relevance. The feature relevances are calculated with respect to one desired output class, which is also referred to as target class. Pixel attribution (=saliency map) translates this idea into the image setting with CNNs, where each input-pixel or image regions receive a relevance. Saliency maps either have the same size as the input image or can be meaningfully projected onto it, such as by scaling and overlaying the saliency map onto the input image. Figure 3.1 shows one example of such a saliency map.

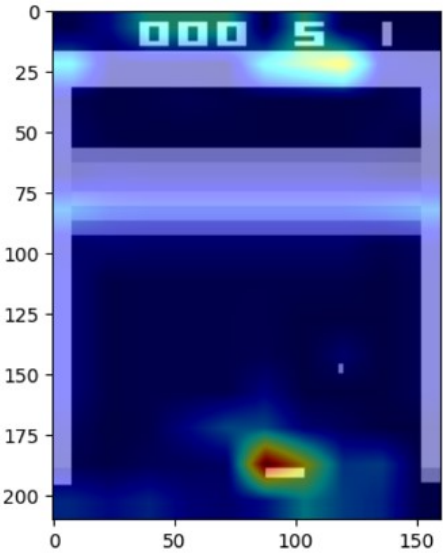


Figure 3.1: A saliency map in the Atari game Breakout in its beginning phase. The relevant image areas regarding the decision of the agent are red, while the uninteresting areas are blue.

The concept of saliency maps has its origins in neuroscience [23] in 1998, and were first witnessed in deep learning in [24] in 2013, where it was one of various visualization techniques to compute images. From there, the idea gained traction and many different methods to construct them were proposed. These methods can be categorized into two main approaches: Gradient based methods (chapter 3.2.1) and perturbation based methods (chapter 3.2.2). Some common techniques are used in both of these approaches.

Choosing a target class

The calculation of the relevance of pixels or image areas is typically done with respect to one class of interest. The class of interest can be chosen arbitrarily, which makes sense in many domains as it might be interesting to understand which pixels or image regions contribute to another class. For example when classifying MNIST[1] numbers, the class of interest could be the number "3" would make sense for an input image containing this actual number. However, it is also possible to pick "6" as class of interest to visualize the pixels or image areas which contribute towards the class "6" instead of "3". The chosen class "3" or "6" is also referred to as target class (target when constructing the saliency map). When working with probabilities in the output layer like in the MNIST domain, it is beneficial to modify the actual output of the neural network before constructing the

[1] http://yann.lecun.com/exdb/mnist/

saliency map. The output is transformed into a one-hot array so that the target class has a value ($\hat{=}$ probability) of 1 and all other classes a value of 0. This is done to explicitly isolate the pixel relevances or image areas towards the desired target class.

In the DQN setting with Breakout, the actual class/action is unknown, since there are no definitive labels as in MNIST. Here, the class with the highest score (meaning the action with the highest Q-value) is picked as target class.

3.2.1 Gradient Based Methods

Gradient based methods calculate the saliency map based on the gradient of the prediction with respect to the input image. The image of interest is fed to the neural network, and for some desired output (for example the output class of interest) the gradient gets calculated. The gradient is used to assign each pixel a value, which can be interpreted as the relevance of that pixel. Gradient based methods are thereby mandatory model-specific, since the internal neural network architecture has to be known to calculate the gradient. Gradient based methods mainly differ from one another in how the gradient is calculated.

Vanilla Gradients

Algorithm 3: Vanilla Gradients

Input: `network, image`	▷ Neural net and input frame-stack
`pred = network(image)`	▷ Forward pass
`pred = setActivations(pred)`	▷ Set class activations except target class to zero according to 3.1
`grads = gradient(pred, image)`	▷ Gradients w.r.t. input image using 3.2
`grads = abs(grads)`	▷ Take absolute from gradients
`salMap = reduceSum(grads)`	▷ Sum up frame gradients
`salMap = normalize(salMap)`	▷ Normalize gradients
`salMap = colorMap(salMap · 255)`	▷ Colorize gradients
Output: `salMap`	▷ Saliency map

The Vanilla Gradients technique [24] (2013) was the first saliency map method in deep learning (called "Image-Specific Class Saliency" in the original paper). The technique works by first forward-passing the input image through the neural network and then choosing a target class (=class of interest), which means setting all other output values to zero while keeping the original value of the target class (see chapter 3.2):

$$S'_c(x) = \begin{cases} x & \text{if } x = \arg\max_{c \in C} \left(S_c(x)\right) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where $S'_c(x)$ is the class activation function for class c given input x. This manipulates the original class activation function so that it only outputs the original value in case of the target class, and zero otherwise. Then, the gradient with respect to the input pixels is calculated:

$$M_c(I) = \frac{\Delta S'_c(I)}{\Delta I} \tag{3.2}$$

Where I is the input image and S'_c is the adjusted class activation function from equation 3.1. The result is a matrix with the same shape as the input image, where each (gradient-)value corresponds to the relevance of that pixel regarding the target class. Then, the absolutes of the gradients are taken, as a negative gradient of some value $-g$ indicates the same relevance as a positive gradient $g = |-g|$. The absolute gradients of the different channels (3 RGB channels) are then summed up, so that the summed gradient value at one pixel includes the relevance of all RGB channels at that pixel $g_{\text{RGB}} = g_{\text{R}} + g_{\text{G}} + g_{\text{B}}$. The last step to construct the saliency map is to normalize the positive-only values and colorize them with some color map.

Adjusted Vanilla Gradients

The only adjustment to use vanilla gradients in the presented Breakout task (2.1) lies in the interpretation of the variables. Instead of one input image with 3 RGB channels, a frame-stack of 4 grayscale images is supplied. Thus the gradient values now indicate the relevance of a pixel of one of these frames, instead of a RGB channel value. The gradient values of the frames (previously RGB channels) can still be summed up in the same way. The mathematical computations are the same as in the non-adjusted version: Instead of summing up the gradients for each channel at one pixel, the gradients for each frame at that pixel are summed. An example saliency map of the adjusted vanilla gradients method is shown in figure 3.2.

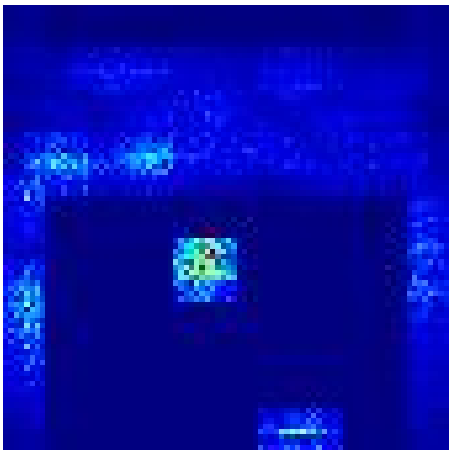


Figure 3.2: An example of a saliency map constructed by the adjusted vanilla gradients method.

The saliency maps produced by vanilla gradients can be very noisy. SmoothGrad (section 3.2.1) introduces an idea to combat this issue.

SmoothGrad

Algorithm 4: SmoothGrad

```
Input: network, image, n, σ                    ▷ Neural net, frame-stack, noise params
  pred = network(image)                        ▷ Forward pass
  expected = setActivations(pred)              ▷ Set class activations according to 3.1
  gradArr = []
  for i=1 to n do
    noisyImage = image + noise(σ)
    noisyPred = network(noisyImage)            ▷ Vanilla Gradients start
    loss = loss(expected, noisyPred)
    grad = gradient(loss, noisyImage)
    grad = abs(noisyGrad)                      ▷ Vanilla Gradients end
    gradArr.add(grad)
  end
  grads = reduceAvg(gradArr)                   ▷ Average gradient matrices
  salMap = reduceSum(grads)                    ▷ Sum up frame gradients
  salMap = normalize(salMap)                   ▷ Normalize gradients
  salMap = colorMap(salMap · 255)              ▷ Colorize gradients
Output: salMap                                 ▷ Saliency map
```

The problem with many gradient based saliency map methods is that, to a human eye, they seem very noisy, as some seemingly random individual pixels are highlighted. The motivation of SmoothGrad [25] is "removing noise by adding noise" (in the title of the paper). In particular, Gaussian noise is added to the input image before it is fed to the neural network. Of course, the result should be clipped to stay within the range of valid pixel values. This is done n times, and the resulting gradient maps are then averaged:

$$\hat{M}_c(I) = \frac{1}{n} \sum_{1}^{n} M_c(I + \mathcal{N}(0, \sigma^2)) \tag{3.3}$$

where $\hat{M}_c(I)$ is the smoothed gradient map with input image I regarding target class c, M_c is the gradient map method SmoothGrad is coupled with (or based on) and $\mathcal{N}$

Adjusted SmoothGrad

As with vanilla gradients, the only adjustment lies within the interpretation of the variables. Instead of a RGB image, a frame-stack of 4 grayscale frames is supplied. The shapes of the matrices differ in comparison to RGB images, but the methodology is the same. The same applies when summing up the gradients: Instead of summing up the gradients of the 3 RGB channels at each pixel, the gradients of the 4 frames at each pixel are summed up instead. An example saliency map with the adjusted SmoothGrad method is shown in figure 3.3.

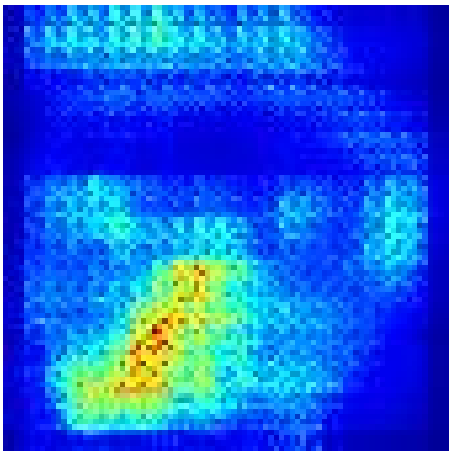


Figure 3.3: An example of a saliency map constructed by the adjusted SmoothGrad method.

The concept of SmoothGrad can be applied to all gradient based saliency map methods, as it does nothing more than observing n gradient maps produced by some method (e.g. vanilla gradients), and then averaging them.

GradCAM

Algorithm 5: GradCAM

Input: `network, image`	▷ Neural net, frame-stack
`pred = network(image)`	▷ Forward pass
`activation = network .lastConvActivation()`	▷ Get activations of last convolutional layer w.r.t. the input image
`expected = setActivations(pred)`	▷ Set class activations according to 3.1
`loss = loss(expected, pred)`	
`grads = gradient(loss, activation)`	▷ Gradients w.r.t. last convolutional layer
`avgGrads = avg(grads)`	▷ Average gradients of each feature map
`heatmap = avgGrads ∘ activation`	▷ Multiply averaged gradients with activations of last convolutional layer
`heatmap = avg(heatmap)`	▷ Average values of each feature map pixel
`heatmap = normalize(heatmap)`	▷ Normalize values
`salMap = resize( heatmap, image.size())`	▷ Scale up to input image
`salMap = colorMap(salMap · 255)`	▷ Colorize values
`salMap = salMap · 0.5 + image`	▷ Overlay with input image
Output: `salMap`	▷ Saliency map

GradCAM [CITEHERE] stands for Gradient-weighted Class Activation Map and as the name suggests provides visual explanations via saliency maps, based on the gradient of the neural network. While other methods like Vanilla Gradients (section 3.2.1) work for all neural network architectures, GradCAM is specialized for CNNs. Vanilla Gradients computes the gradients up to the input image and thus making the internal neural network architecture irrelevant. GradCAM on the other hand calculates the gradients up to the last convolutional layer. Thus, a CNN is required. It extends the class activation

mapping idea (CAM, [26]) to weigh the activations of the last convolutional layer by their gradients. The last convolutional layer is used for that because it captures high-level semantics being further back in the CNN while still containing spatial information which are lost in the subsequent fully connected layers. The idea behind this approach is to visualize which areas of the input image are focused by the convolutional layer. The first convolutional layer receives the image of size (x, y) as input and outputs feature maps that encode learned features (activations), where x and y are the width and height of the input image. The convolutional layers thereafter do the same but receive the feature maps of the previous convolutional layer as input, instead of the input image. The last convolutional layer outputs its activations of size x_L, y_L, n_L, where n_L are the number of filters and x_L and y_L are the width and height of the resulting feature maps. Afterwards, the forward pass through the remaining fully connected layers is completed to receive the prediction. With this, a target class is chosen according to equation 3.1 and the loss is calculated. Then, the gradients are calculated with respect to the last convolutional layer, thus having the size x_L, y_L, n_L. These gradients are then averaged for each feature map, resulting in the size n_L. The averaged gradients get multiplied with the activations of the last convolutional layer over the last axis, so that the resulting size is x_L, y_L, n_L again. Now, the values on the feature map axis are condensed (averaged), so the result is a value matrix with size x_L, y_L. This matrix is then upscaled to the size of the input image x, y and colorized. Upscaling the relatively small feature map to the larger input image makes the resulting saliency map rather coarse, as the resolution of the feature map is much lower.. The colorized matrix shows the areas whose are focused by the CNN. As final step, the matrix is overlayed with the input image to see these highlighted areas and the actual content of the image.

Adjusted GradCAM

In the experiment setting, the RGB image is replaced with a grayscale frame-stack. The convolutional layers are configured to slide over the last two axes, the x- and y-axis of the frame-stack, since the "channels" are now in the first dimension (for more details see section 5.1.1). This change runs through all steps: The last convolutional layer now outputs activations of size n_L, x_L, y_L, thus the gradients also have the size n_L, x_L, y_L and the multiplication of the averaged gradients with the activations of the last convolutional layer as well n_L, x_L, y_L. During the averaging and multiplications, it is important that the correct axes are used. Furthermore, the colorized matrix is overlayed with only the first frame of the frame-stack, as it is infeasible to overlay it with the complete input

frame-stack. An example saliency map with the adjusted GradCAM method is shown in figure 3.4.

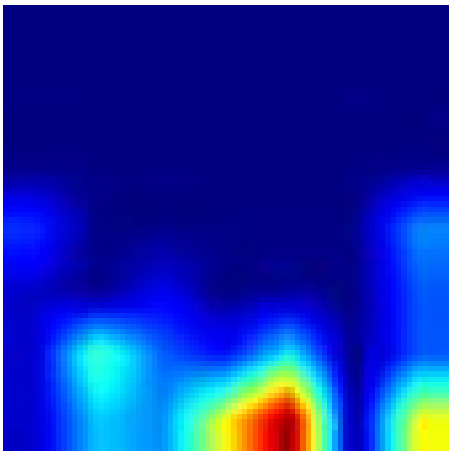


Figure 3.4: An example of a saliency map constructed by the adjusted GradCAM method.

GradCAM can be combined with another gradient-based method to focus the coarse saliency map to finer regions of the image, as is explained in section 3.2.1.

Guided GradCAM

Algorithm 6: Guided GradCAM

```
Input: network, image                          ▷ Neural net, frame-stack
  pred = network(image)                        ▷ Forward pass
  activation = network                         ▷ Get activations of last
     .lastConvActivation()                       convolutional layer w.r.t.
                                                 the input image
  expected = setActivations(pred)              ▷ Set class activations ac-
                                                 cording to 3.1
  loss = loss(expected, pred)
  grads = gradient(loss, activation)           ▷ Gradients w.r.t. last
                                                 convolutional layer
  avgGrads = avg(grads)                        ▷ Average gradients of
                                                 each feature map
  heatmap = avgGrads ∘ activation              ▷ Multiply averaged gra-
                                                 dients with activations of
                                                 last convolutional layer
  heatmap = avg(heatmap)                       ▷ Average values of each
                                                 feature map pixel
  heatmap = normalize(heatmap)                 ▷ Normalize values
  salMap = resize(                             ▷ Scale up to input image
     heatmap, image.size())
  vGrads = gradient(pred, image)               ▷ Vanilla gradients start
  vGrads = abs(vGrads)
  vMap = reduceSum(vGrads)
  vMap = normalize(vMap)                       ▷ Vanilla gradients end
  salMap = salMap ∘ vMap                       ▷ GradCAM fusion with
                                                 Vanilla Gradients
  salMap = colorMap(salMap · 255)              ▷ Colorize values
Output: salMap                                 ▷ Saliency map
```

Guided GradCAM [27] fuses GradCAM with a gradient-based method which calculates the gradients all the way up to the input image to produce finer saliency maps. This is done by multiplying the GradCAM heatmap with the heatmap of the other method

as shown in table 3.2.1 with Vanilla Gradients as example. GradCAM functions like a lense focusing Vanilla Gradients towards the coarse areas which GradCAM chooses as relevant. The fused heatmap is then colored and returned as saliency map. In contrast to GradCAM, an overlay with the original input image is unnecessary, as Vanilla Gradients already includes enough content of the input image.

Adjusted Guided GradCAM

Guided GradCAM consists of GradCAM and another method, for example Vanilla Gradients, which both have to be adjusted for the experiment setting as explained in 3.2.1 and 3.2.1 respectively. No further adjustments are necessary. An example saliency map with the adjusted Guided GradCAM method is shown in figure 3.5.

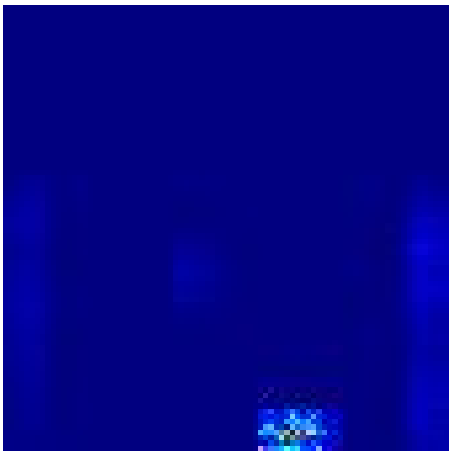


Figure 3.5: An example of a saliency map constructed by the adjusted Guided GradCAM method.

In Guided GradCAM, GradCAM can be combined with any other gradient-based method which calculates the gradients all the way up to the input image.

LRP

Algorithm 7: LRP

```
Input: network, image, ϵ                    ▷ Neural net, frame-stack, fraction uplift
  activations = []                          ▷ Initialize activations
  layerWeights = []                         ▷ Initialize weights
  layerBiases = []                          ▷ Initialize biases
  output = image                            ▷ Input for first layer
  for each l in network.layers() do         ▷ Forward pass
    output = l.forward(output)              ▷ Input for next layer
    activations.add(output)                 ▷ Save activations
    weights.add(l.weights())                ▷ Save layer weights
    biases.add(l.biases())                  ▷ Save layer biases
  end
  R = [0, ..., 0]                          ▷ Initialize array with the same length as output
  R[argmax(output)] = max(output)           ▷ Target class value
  for i=types.length()-1 to 0 do            ▷ Relevance backprop.
    type = network.layers[i].type()         ▷ Layer type
    w = weights[i]                          ▷ Layer weights
    b = biases[i]                           ▷ Layer biases
    if i == 0 do                            ▷ At first layer...
      a = [1, ..., 1]                       ▷ set activations to 1
    else                                    ▷ At deep layers...
      a = activations[i-1]                  ▷ of previous layer
    end
    R = lrp-ϵ(R,a,w,b,type)                 ▷ Relevance calculation according to equation 3.4
  end
  salMap = normalize(R)                     ▷ Normalize relevances
  salMap = colorMap(salMap · 255)           ▷ Colorize values
Output: salMap                              ▷ Saliency map
```

Layer-wise relevance propagation (LRP, [28]) doesn't compute gradients but backpropa-

gates a relevance score from the output to the input image. It uses the weights, biases and activations at each layer during the forward pass to propagate the output back up until the input layer. For this, the activations at each layer are stored during the forward pass. Then, the relevance is initialized: The output of the neural network is overwritten with 0 except for the location of the highest value (target class). Thereby, the relevance of the target class is isolated. Now, the relevance is backpropagated through the layers according to the LRP-ϵ rule:

$$R_j = \sum_k \frac{a_j \; w_{jk}}{\sum_0^j [a_j \; w_{jk} + b_k] + \epsilon} R_k \tag{3.4}$$

where j and k are neurons of two consecutive layers, a_j is the activation of neuron j in the former layer, w_{jk} is the weight of the connection between neuron j and k, b_k is the bias of neuron k and R_k is the relevance of neuron k. The addition of a very small ϵ ensures that the bottom part of the fraction is not or close to 0. ϵ is typically set to $\epsilon = 1\text{e}^{-9}$. For the first layer, the activations are set to be 1, as otherwise, the pixel values would be used. This is nonsensical since the relevance should not be dependent on the pixel value: A white pixel (value 255, normalized 1.0) would then be naturally more relevant than a black one (value 0). The biases are ignored in convolutional layers. There are other rules than LRP-ϵ, like LRP-0 or LRP-γ, which can even be used in combination based on the depth of the layer. The highlights of saliency maps constructed by LRP are the result of these relevance backpropagation rules.

Adjusted LRP

The relevance backpropagation steps for the convolutional layers are adjusted to cope with the frame-stack input. The "channels" are at the first axis of the frame-stack, which has to be kept in mind when backpropagating the relevance. Handling the frame-stack like a RGB image would lead to shape mismatches. An example saliency map with the adjusted LRP method is shown in figure 3.6.

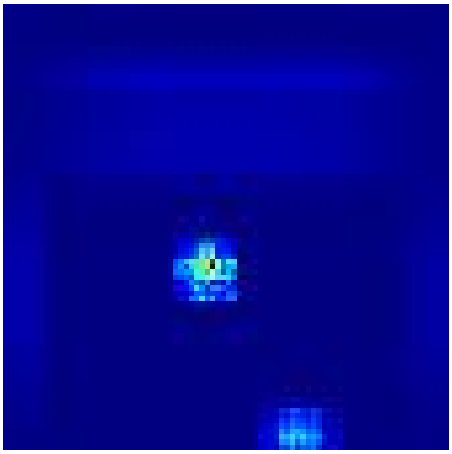


Figure 3.6: An example of a saliency map constructed by the LRP method.

Gradient-based methods require access to the interior of the neural network, as they all use the networks weights during backpropagation. Perturbation-based methods construct saliency maps in a model-agnostic style.

3.2.2 Perturbation Based Methods

Perturbation based methods alter the input in some form and carefully watch how the outputs of the model change with the perturbed input. Based on these changes, the XAI method then derives explanations. Perturbation based methods are usually model-agnostic, since only input-output tuples of the models are required to generate explanations. The actual workings of the model are irrelevant. On the one hand, this is very handy since these methods can be applied to any type of model, not just machine learning based neural networks like gradient based methods (section 3.2.1). On the other hand, model-agnostic methods only estimate the actual model as best as possible, but the estimation might still have some amount of distance to the actual model (known as model soundness, section 3.1.2).

LIME

Algorithm 8: LIME

```
Input: network, image, nSamples                ▷ Neural net, frame-stack, number of samples
  pred = network(image)                        ▷ Forward pass
  expected = setActivations(pred)              ▷ Set class activations according to 3.1
  segments = segmentation(image)               ▷ Superpixel segments
  segImgs = []                                 ▷ Segmented images
  for i=0 to nSamples do                       Draw samples
    segImg = random(segments)                  ▷ Random combination of superpixels
    segImgs.add(segImg)                        ▷ Add perturbed image
  end
  labels = network(segImgs)                    ▷ Predictions of perturbed images
  linear = LR(segImgs, labels)                 ▷ Linear regression model
  heatmap = linear.explain(image)              ▷ Calculate superpixel influence
  salMap = normalize(heatmap)                  ▷ Normalize influence values
  salMap = colorMap(salMap · 255)              ▷ Colorize values
Output: salMap                                 ▷ Saliency map
```

LIME [29] stands for Local Interpretable Model-Agnostic Explanations and does exactly what the name implies: As a model-agnostic method, it estimates a model locally (for one instance) where it is interpretable and gives an explanation there. It draws samples around the to-explain-instance (original instance) by slightly changing the input, feeds them to the model and observes the changes in the output. The number of samples is given as parameter (1000 in this work).

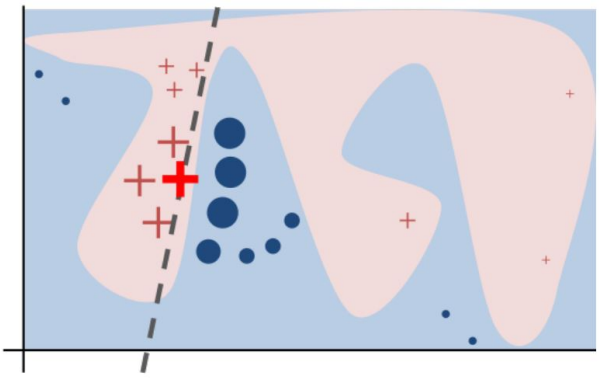


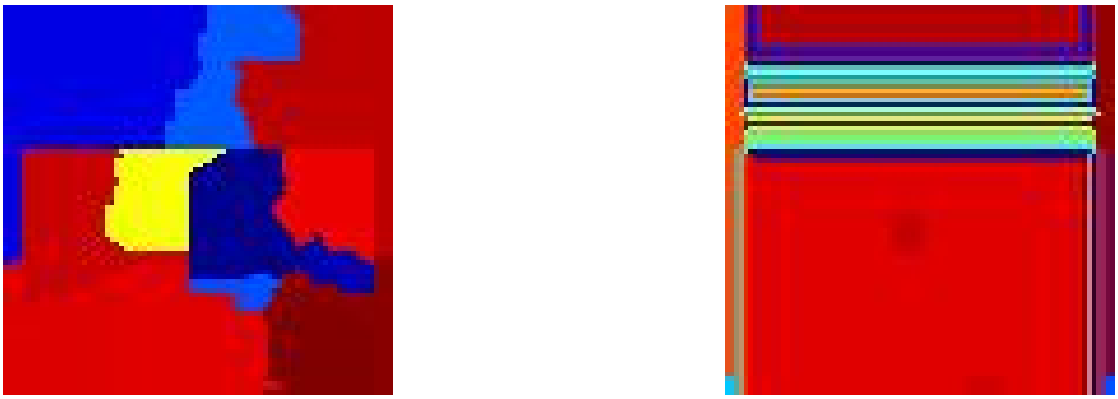
Figure 3.7: An illustration of LIME with 2 classes: The model classifies some samples as positive (red plus) and some as negative (blue circle). They are weighted by their proximity to the original sample (drawn bigger). LIME calculates a locally faithful linear model (dotted line) to the model function (red/blue background). Image source: [29].

The drawn samples produce different outcomes than the original instance. The difference of the sample output to the original depends on how relevant the changed input feature is for every given sample. The samples are then weighted by their proximity to the original sample as shown in figure 3.7. With that, LIME approximates the model locally with linear regression. Of course, the linear function doesn't capture the complete model prediction function, but it is locally faithful. LIME then estimates the importance of each feature: A feature that was only slightly changed but resulted in a very different outcome has a high influence. An explanation is given showing the magnitudes of positive and negative influences of each feature. Originally, LIME was intended to be used for feature vector inputs (not images), but can be applied to images with a few changes. [29] proposes to segment the images into superpixels, which subsequently can be handles like individual features. For example, a random combination of segments may be deleted by setting the pixel values to 0. The explanation then gives influence scores to each of these superpixels and can be visualized as saliency map.

Adjusted LIME

LIME has to be adjusted at the segmentation part to cope with the frame-stack input. Originally, the single RGB image is segmented into superpixels. However, the frame-stack consists of 4 grayscale images, therefore the segmentation is done on the first image and then applied to the rest, hence the superpixels capture the same pixel areas in all frames. Alternatively, the frames could be segmented separately, but this increases

the number of features by the number of frames, which in turn increases the number of samples needed to calculate a faithful linear model. Since the required computation time is already very high (see section 5.1.2) and the frames are only marginally different, it is assumed to be sufficient to apply the segments of the first frame to the remaining 3. Two different algorithms are used to segment the image, resulting in two different saliency maps: LIME with the Quickshift segmentation algorithm and LIME with the Felzenszwalb segmentation algorithm.



(a) LIME Quickshift

(b) LIME Felzenszwalb

Figure 3.8: An example of saliency maps constructed by LIME Quickshift (a) and LIME Felzenszwalb (b).

As shown in figure 3.8, the saliency maps constructed by LIME heavily depend on the used segmentation algorithm, as the superpixels (feature areas) are determined by them.

RisE

Algorithm 9: RisE

```
Input: network, image, nMasks, h, w, p        ▷ Neural net, frame-stack, RisE params
  pred = network(image)                       ▷ Forward pass
  targetClass = argmax(pred)                  ▷ Choose target class
  H = image.height()                          ▷ Image pixel height
  W = image.width()                           ▷ Image pixel width
  masks = []                                  ▷ Masks
  for n=0 to nMasks do                        ▷ Generate masks
    cell = [h][w]                             ▷ Matrix of size h x w
    for i=0 to h do
      for j=0 to w do                         ▷ For each cell pixel
        cell[i][j] = randFloat() < p          ▷ Mask pixel
      end
    end
    mask = resize(cell, H/h+1, W/w+1)         ▷ Upscale mask
    mask = crop(mask, H, W)                   ▷ Crop random area
    masks.add(mask)                           ▷ Store mask
    end
  end
  maskImgs = []                               ▷ Masked images
  for n=0 to nMasks do                        ▷ For every mask
    maskImg = image ∘ mask                    ▷ Mask image
    maskImgs.add(maskImg)                     ▷ Store masked image
  end
  labels = network(maskImgs)                  ▷ Masked images labels
  heatmaps = labels.T() · masks /             ▷ Weighted relevances
     nMasks / p
  heatmap = heatmap[targetClass]              ▷ Target class heatmap
  salMap = normalize(heatmap)                 ▷ Normalize values
  salMap = colorMap(salMap · 255)             ▷ Colorize values
Output: salMap                                ▷ Saliency map
```

Randomized input sampling for explanation of black-box models (RisE, [10]) works by altering the input and observing the changes in the prediction of the model. The image is perturbed by multiplying it with various masks (amount given as parameter, in this work: 8000). The mask creation process starts with initializing a cell matrix with width $w < W$ smaller than the image width W and height $h < H$ smaller than the image height H. In this work, the cell size is set to be $8x$ smaller than the image size. The values in this cell are then set to `1.0` with probability p (given as parameter, here: 0.5) and `0.0` otherwise. The cell is then upscaled $8x + 1 = 9x$ with bi-linear interpolation. The cell is now larger than the image, hence it is cropped to fit the size of the image. Upscaling to a size larger than the image and then cropping a random frame is done in order to ensure a high diversity of masks. Afterwards, the individual masks are multiplied element wise with the input image to produce the perturbed images. These perturbed images are fed through the neural network to receive the outputs. Then, the heatmaps are calculated as weighted sum of the random masks, with the weights representing the probability scores that the masks produce, which is then adjusted for the distribution of the random masks. The resulting tensor includes the heatmap matrices for each output class of the neural network, of which the one with regards to the target class is taken and further processed into a saliency map.

Adjusted RisE

Like with LIME, the random masks of RisE are applied in the same way to each frame of the frame-stack: With an RGB image, the masks can simply be multiplied element-wise with the image. With a frame-stack, a mask is multiplied element-wise with each of the four frames of the frame-stack. This implies that the exact same image region is observed for all frames, which is reasonable as the content only changes marginally between four subsequent frames. An example saliency map with the adjusted RisE method is shown in figure 3.9.

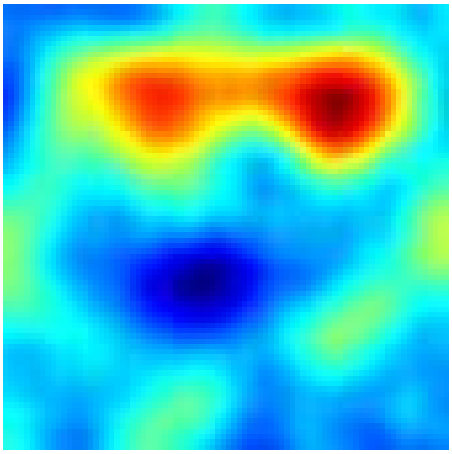


Figure 3.9: An example of a saliency map constructed by RisE.

RisE highlights the image areas which have a high influence on the decision of the model.

3.3 Deletion

The idea of deletion first appeared in [30] in 2015 and was first called so by [10], who also introduced RisE (see section 3.2.2), in 2019. Deletion works by removing the most relevant pixels/areas from the input image according to some saliency map and measuring how much the accuracy of the neural network suffers. If only a few of the most important pixels according to some saliency map have to be deleted in order to drastically reduce the accuracy of the neural network, then the saliency map has a high degree of correctness as it highlights the important pixels precisely.

Deletion in Reinforcement Learning

The idea of deletion can be transferred to the reinforcement learning setting as shown in [10]. Instead of measuring how much worse the neural network performs, the drop in performance of the reinforcement learning agent is measured. If important pixels of the input frame(-stack) are deleted, the agent should be less successful in accomplishing the given task. As an example in this case, 10% of the most relevant pixels according to SmoothGrad are deleted from the frame-stack (overwritten with background color black: pixel value 0) through which the agent should perform worse. This drop in performance in contrast to not deleting any pixels is recorded and can lead to statements about how

correct the saliency map is. If the agent performance drops off rapidly with a small amount of deletion, the degree of correctness is high as it correctly highlights pixels which were important for the agent.

4 Implementation

This chapter provides details to the implementation which is used for the experiments in chapter 5. As programming language, python 3.9 is used. The components of the implementation are described in section 4.1 and the used hardware in section 4.2. Each experiment has its own folder with its own JSON configuration file containing all hyperparameters and the description of the agent and environment, which are used to train the agent and save the model and logs in this folder. Afterwards, saliency maps and their evaluation are constructed whose results are also stored in this folder.

4.1 Components

The implementation is divided into components with high cohesiveness: The environment, described in section 4.1.1, comprises any tasks associated with the Atari Breakout reinforcement learning environment. The replay buffer 4.1.2 independently stores and samples transitions. The agent component (4.1.3) implements the double DQN agent. The policy is a small component which is responsible for choosing actions, hence it communicates with the agent component (4.1.4). The training loop integrates all other components to train the agent 4.1.5. Saliency maps and the deletion evaluation are done separately as described in section 4.1.6 and 4.1.7.

4.1.1 Environment

The library Gym (version 0.23.0) is used as basis for the Atari Breakout environment. Gym has to be installed with the Atari and its license extensions `gym[atari,accept-rom-license]==0.23.0`. The `Environment` class which is defined in `training/src/environment.py` wraps the actual Gym Breakout environment to return frame-stacks instead of just the current frame. Therefore, the frame-stack size $k = 4$ and its width and height of 84 are adjustable parameters in the initialization.

Performing Steps in the Environment

The `step` method receives an action (whose datatype is integer). This action is passed to the Gym environment, except for when a live was lost in the last step, which happens when the ball misses the platform and falls into the ground. In this case, whichever action was passed as argument is ignored and overwritten by the FIRE action, because this action is required to spawn a new ball. If this is not done, the agent might get stuck doing nothing after loosing one life to avoid further negative rewards. After passing the action to the Gym environment, the new frame, the reward, the terminal flag and further information are received. The new frame is preprocessed and the frame-stack is adjusted to contain the new preprocessed frame. The reward is clipped to $[-1; 1]$. The terminal flag is `True` if the game is over, however the game is over one step after the last live was lost. The frame-stack, the reward, the terminal flag and the extra information are returned. The `step` method is called every time a collection or evaluation step is performed, as well as when just playing the game. The time quantization between two steps in the game is 4 frames.

Preprocessing Frames

The frames received from the Gym environment are of size 160x210 and are firstly converted into grayscale. Then the grayscale image is cropped to 160x160 by cropping the top of the frame. The score and the game field above the blocks are lost but contain no useful information anyway. After that, the frame is down-scaled to 84x84 with nearest neighbor interpolation. Lastly, the grayscale values of $[0; 255]$ are normalized to the interval $[0; 1]$. For all these steps, TensorFlow operations are used, which is why the resulting Tensor is converted to a Numpy array before being returned. This preprocessing is done for every frame which ends up in the frame-stack.

Resetting the Environment

The `reset` method firstly resets the Gym environment and puts the resulting frame in the frame-stack. Then, to fill up the frame-stack, further environment steps with FIRE actions are done, depending on the size of the frame-stack. The game is started and the ball is spawned in this process due to the FIRE actions. The `reset` method is called every time the game is over due to all live lost or before evaluating the agent.

4.1.2 Replay Buffer

The replay buffer (located in `training/src/replay_buffer.py`) stores the trajectories and samples mini-batches. No additional library is used for that except Numpy. The replay buffer is initialized with a size (number of transitions to store), a (mini-)batch-size, the size of the frame-stack k and the frame width and height. Typically, the replay buffer stores complete transitions as tuples (s_t, a_t, s_{t+1}, r, $terminal$).

Optimizing for Memory Usage

While storing the complete transitions as tuples is the easiest strategy, it wastes a lot of memory since the frame-stack at time-step t contains the current game frames from t to $t - k$, thus the same frame would exist k times in the replay memory. And because not just the frame-stacks at every time step s_t would be stored but also the frame-stack of the next state s_{t+1}, the same frame actually exists $2 \cdot k$ times in the frame-stack. A replay buffer with 1 million transitions (which is quite common) with a frame size of 84x84 of floats (4bytes) and a frame-stack size of $k = 4$ would thus require $1000000 \cdot 4 \cdot 84 \cdot 84 \cdot 2 \cdot 4\text{bytes} = 225.792.000.000\text{bytes} \approx 226\text{GB}$ without the actions, rewards and terminal flags, which I and probably you don't have. This is the reason why instead, just the current game frames are stored, which reduces the memory usage to $1000000 \cdot 84 \cdot 84 \cdot 4\text{bytes} = 28.224.000.000\text{bytes} \approx 28\text{GB}$, a reduction of 8x. Of course, this requires to re-create the correct frame-stacks for s_t and s_{t+1} when sampling a mini-batch.

Adding Experience

The `add` method adds a sample (transition) consisting of an action, the current game frame, the reward and terminal flag to the replay buffer. The current game frames, actions, rewards and terminal flags are stored in separate Numpy arrays, which are connected via their indices, where the the same index i correlates to the same time step t. The replay buffer is implemented as rotating list, meaning that every new sample (transition) is stored at index $i + 1$, and after reaching the size of the replay buffer, the index to store this new sample at is reset to 0, from where the old samples are overwritten. The resulting Numpy arrays are chronologically ordered to the current index i and from the current index i.

Sampling a Mini-Batch

With the memory usage optimizations, it is required to re-create the correct frame-stacks when sampling a random mini-batch. A frame-stack for some index i can be

re-constructed by slicing the frames array from $i - k$ to i. The `sample` method chooses a random index i and validates it by checking whether a re-constructed frame-stack contains no terminal frames, since having a terminal frame at any position in the frame-stack would mean that the agent plays after the game is over. With enough valid indices found, the mini-batch of actions, rewards and terminal flags can simply be sliced out of the corresponding Numpy arrays. For the states and next states, the frame-stacks for each index i and $i + 1$ are re-created respectively. Of course, the frame-stack of the next state may have a terminal frame at the newest position, this transition would correlate to the last step before a game over. The mini-batch is then returned.

4.1.3 Agent

The library TensorFlow (version 2.9.0.) is used as basis for the DQN agent. The DQN agent is implemented as `DQN` class in `training/src/dqn.py`. It is initialized with all the required parameters: Two Q-networks (one of which is the target network), a loss function, an optimizer, the gamma value and an integer as interval to update the target network.

Q-Value Estimation

The `DQN` can be called with a frame-stack, which simply returns the Q-values for each possible action as Numpy array. The `DQN` is called every time the actual Q-value estimation is needed. This is needed when i) performing a collection step in the environment on the actual policy ii) evaluating the current DQN agent on the actual policy or playing Atari Breakout with the finished trained agent or iii) during training of the DQN agent.

Training

The `train` method receives a mini-batch and trains the Q-network as explained in 2.3.2. The gradient is calculated with the TensorFlow `GradientTape` and the weight updated are performed by the optimizer given in the initialization. Every now and then, the weights of the Q-network are copied to overwrite those of the target network. The `train` method is called after every collection step during training, and the mini-batch is extracted from the replay buffer.

4.1.4 Policy

The policy chooses an action based on the current state. In DQN, an ϵ-greedy policy is used. When evaluating the current performance of the agent, the ϵ-greedy policy is bypassed and the action with the highest Q-value is used instead. The policy is implemented as class in `training/src/e_greedy_policy.py` and is initialized with the Q-network, the number of possible actions, the initial ϵ probability, the final minimum ϵ probability and an annealing time which defines after how many iterations the final minimum ϵ probability is reached.

Choosing an Action

The `EGreedyPolicy` can be called to choose an ϵ-greedy based on the state (frame-stack). With a specific probability ϵ, a random action is chosen, otherwise the action with the highest Q-value determined by the Q-network is used. At the beginning, this probability is $\epsilon = 1.0$ and is decreased linearly over time defined by the annealing time, until the final ϵ probability is reached.

4.1.5 Training Loop

The training is started with `training/train_agent.py`. Here, all hyper-parameters are loaded from a configuration JSON file and the agent, environment, replay buffer and policy are instantiated. Before the actual training loop, a defined amount of collections is done with random actions. The training loop is run through defined by the iterations hyper-parameter and does the following:

1. Collect a sample from the environment according to the ϵ-greedy policy.
2. Sample a mini-batch from the replay-buffer and train the DQN agent.
3. If the evaluation interval is reached, reset the environment and record the performance of the DQN agent according to a greedy policy (always choose the action with the highest Q-value).

Hereby, the loss of the training steps and the episode rewards of the evaluation steps are recorded with TensorBoard. In the evaluation phase, the Q-network is saved if it achieved the highest episode reward so far.

4.1.6 Saliency Map Methods

The saliency map methods presented in chapter 3 are implemented as utility functions in `saliency_maps/heatmaps.py`. They expect the neural network model and the frame-stack, process the saliency map and return it with normalized values in the same size as the frame-size. All saliency map methods are specifically implemented to cope with frame-stacks and to be comparable with regards to their computation time. The linear regression model and explanation calculation of LIME is delegated to the `lime` library[1]. For starting the construction of the saliency maps for one episode, `saliency_maps/create_saliency.py` is invoked, which calls each saliency map method for each time-step of one episode, converts the normalized heatmaps into colored saliency maps and saves them into the corresponding experiment directory.

4.1.7 Deletion Procedure

The deletion procedure is implemented in `saliency_maps/deletion.py` and computes the deletion graphs for each saliency map method, which are saved into the corresponding experiment directory. For each method with each deletion percentage, one episode is played with the agent. Multiple instances with different saliency map methods may be invoked to parallelize the otherwise sequential workload.

4.2 Hardware

The experiments are performed on Windows 10. The machine is fitted with a AMD Ryzen 7 3700X paired with a NVIDIA GTX 1080 Ti and 32GB 3200Mhz memory. All subsequent computation time measurements and graphs are obtained using this machine. The relative relations within these measurements are universally valid and can be compared with other works, but the absolute magnitudes are tied to this specific machine and implementation.

[1] https://github.com/marcotcr/lime

5 Experiments

To answer research question 1 and 2 as accurate as possible, a wide variety of saliency map methods as described in section 3.2 are used, and are applied to two DQN agents in Atari Breakout. The experiment setup is explained in section 5.1, followed by the the training of the two agents in section 5.1.1. The construction of saliency maps is described in section 5.1.2 and the methodology of evaluating them in section 5.1.3. Then, the results are presented in section 5.2.

5.1 Experiment Setup

The saliency map methods are quantitatively evaluated with deletion (see section 3.3). For this, two DQN agents are trained for which the saliency maps are created. The details of the agent training are provided in section 5.1.1. Then, a complete episode is played with each deletion percentage per method to create a deletion graph (see section 5.1.2 and 5.1.3).

Figure 5.1: The number of episode rewards when playing 1000 episodes in Atari Breakout with random actions. The most common reward is 0 and declines for higher rewards. 99% of times the reward is 5 or lower. A reward of 8 was never achieved.

A threshold of the achieved game score (=average reward) is defined, at which so many important pixels are removed, that the agent isn't able to play the game adequately anymore. This performance threshold is defined to be at 5% of its original performance without deletion. The less pixels have to be deleted to fall under this 5% threshold, the higher the correctness of the saliency map. An agent which scores a episode reward of 5 is no better than a random acting agent 99% of the time, as can be seen in figure 5.1. The episode reward of 5 aligns with the threshold of 5% since the trained high performing agent scores a clipped episode reward of exactly 100 (see next section 5.1.1).

5.1.1 Training the Agent

Two DQN agents with different performance are trained, one with lower performance and one with higher performance regarding the achieved game score in Breakout. The difference in performance results from one agent receiving the *terminal* flag from the environment when a life is lost, which helps the agent in learning that loosing a life is bad. Without the *terminal* flag, the target Q-value y, when training with the equation as shown in figure 2.5, still depends on the Q-value estimate of the next state s_{t+1}, which

might raise the target Q-value y of state s to a value that is way too high for that a life was lost. With the *terminal* flag, the target Q-value y directly depends on the reward r, which is 0 in that case (see section 4.1.1).

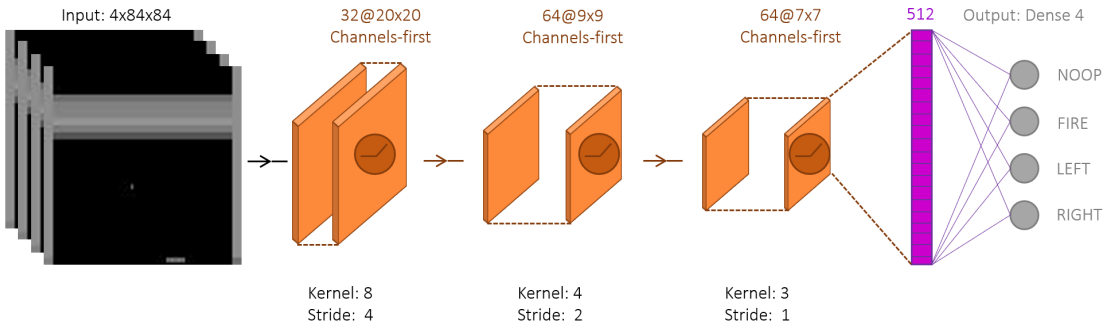


Figure 5.2: The architecture of the Q-network. The 4x84x84 frame-stack is fed through 3 convolutional layers followed by one fully connected layer before the four output neurons.

The Q-network architecture is the same for both agents and is based on the original from [31] and receives a stack of 4 grayscale images as input, each being 84x84 pixels, resulting in an input shape of 4x84x84. The top of the network consists of 3 convolutional layers with ReLU activations, they have increasingly more feature maps and decreasing kernel size and strides. They are connected directly without pooling layers in between, instead, the relatively high stride values ensure that the feature map size decreases. The filters glide over the last two dimensions (x- and y-axis of the grayscale images) instead of the first two dimensions, as the "channels" of the input frame-stack are in the first dimension.

Hyper-Parameter	Value	Description
Iterations	30.000.000	Number of passes through the training loop, thus the total number of environment steps (excluding evaluation and initial collect steps). Lower values are not sufficient to ensure convergence
Replay-buffer size	1.000.000	Maximum number of transitions which are stored in the replay buffer before overwriting older ones. More is better, but would not fit into RAM
Initial collects	50.000	Number of transitions with random actions which are filled into the replay buffer before starting the training loop
Batch-size	32	Number of transitions which are sampled from the replay buffer and used as one batch during training phase in the training loop. Higher values require more computation time, lower values lead to more noise
Learning rate	0.00001	Learning rate when applying gradients to the Q-network. Lower values converge slower, higher values lead to oscillations or no convergence at all
Gamma/Discount factor	0.99	Discount of the highest Q-value of the next state in the Bellman equation. Value should be close to 1 to avoid infinite sums while looking far into future states
Epsilon initial	1.0	Initial chance of choosing a random action during the collection phase in the training loop, ensure exploration
Epsilon final	0.1	Minimum chance of choosing a random action during the collection phase in the training loop
Annealing time	1.000.000	Number of iterations during which epsilon is linearly decreased from its initial value to its final value. Good balance between exploration and exploitation
Train interval	4	Interval between training steps in the training loop, measured against the iterations. Lower intervals lead to oscillations, higher lead to slower convergence
Target update interval	10.000	Interval between updating the target network weights, measured against the iterations. Lower intervals lead to oscillations, higher lead to slower convergence
Loss	Huber	The loss function which is used to calculate the loss for the Q-network between the actual and target Q-values. Huber handles outliers better than MSE
Optimizer	Adam	The optimizer which is used to apply the gradients to the Q-network
Evaluation interval	100.000	Interval between performing an evaluation episode, measured against the iterations
Maximum evaluation steps	10.000	Maximum number of enviornment steps during evaluation, since task might be endless

Figure 5.3: The hyper-parameters for the experiments. They are based on the original from [31] and are the same for both agents.

The hyper-parameters are the same for both agents and are based on the ones from [31]. Note that a higher replay-buffer size would be beneficial but is set to 1 million, as a larger value would surpass the memory of the machine (see section 4.1.2 for details of the replay-buffer and section 4.2 for details of the machine).

Agent 1

The low performing agent doesn't receive the *terminal* flag from the environment when

a life is lost. The agent achieves a score of 38 after 30 million iterations, to which it converged after 15 million iterations.

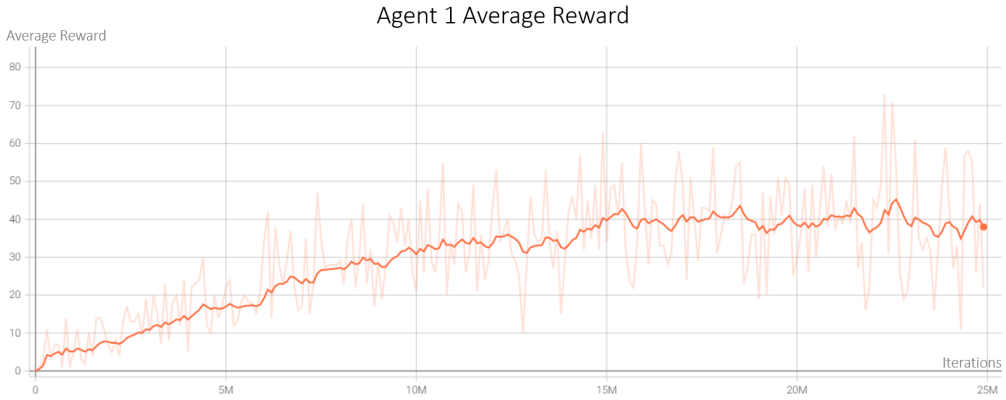


Figure 5.4: The average reward of agent 1 over 25 million iterations. The average reward converges to 38 after 15 million iterations.

The loss and the clipped episode reward during training can be seen in the appendix A.1 and A.2 respectively.

Agent 2

The high performing agent, which receives the *terminal* flag when a life is lost, achieves a score of 380 after 30 million iterations, to which it converged after 15 million iterations as well.

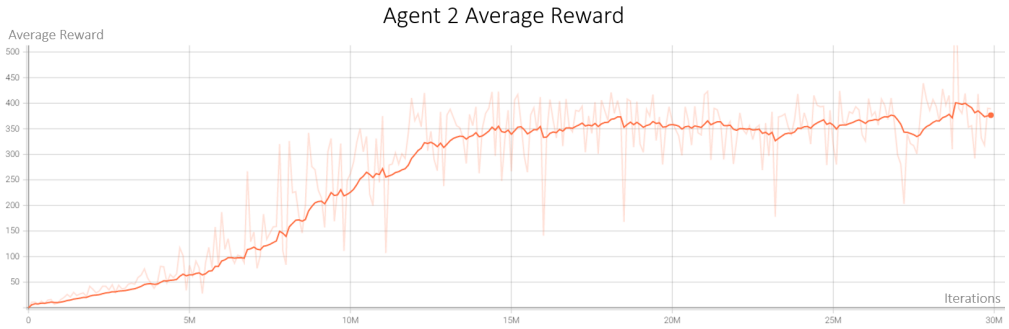


Figure 5.5: The average reward of agent 2 over 30 million iterations. The average reward converges to 380 after 15 million iterations.

The loss and the clipped episode reward of agent 2 during training can be seen in the appendix A.3 and A.4.

Passing the *terminal* flag when a live is lost doesn't change how fast the agent converges to its peak performance, but the performance increases from a game score of 38 to 380, a performance increase of 10x. The average reward curve until 15 million iterations of agent 2 is much steeper than the one from agent 1.

5.1.2 Saliency Map Construction

The saliency maps are created with the trained DQN agent from section 5.1.1.

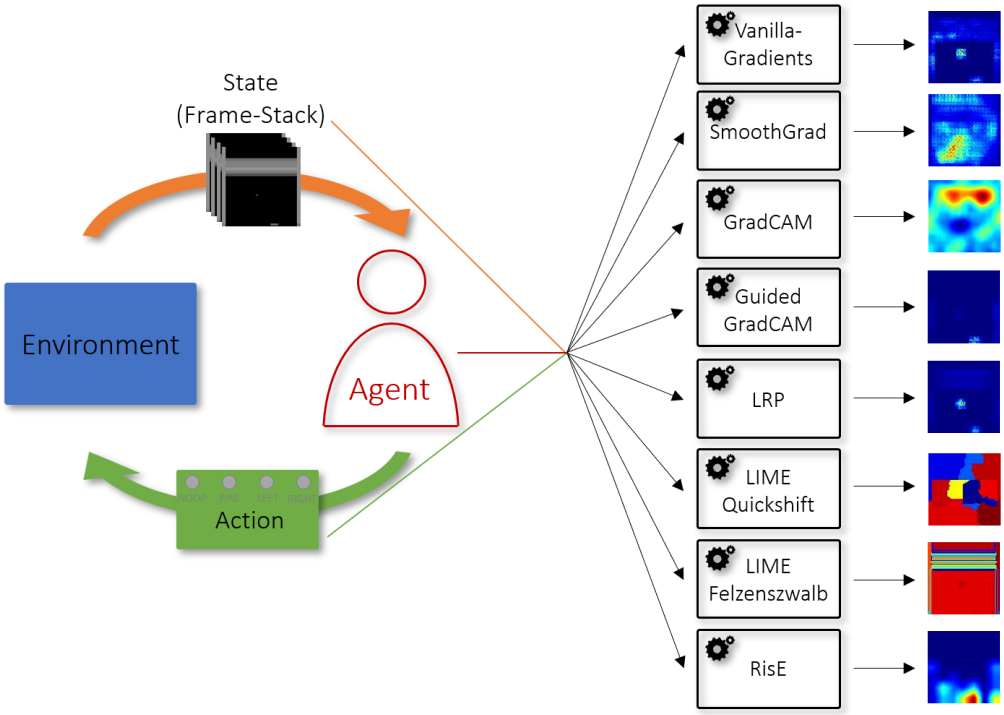


Figure 5.6: At each time-step, the state and agent action are intercepted to construct saliency maps on them. The state and action are identical for all saliency map methods.

The agent plays an episode and the saliency maps for each method described in 3.2 are calculated at each time-step as shown in figure 5.6. Thus, the saliency maps can be compared very well, because the basis is the same for all the saliency map methods: They have the same DQN agent in the same environment with the same gameplay, because all saliency map methods receive the exact same game states/frame-stacks.

The computation time for the gradient-based methods (see section 3.2.1) is much lower than of the perturbation-based methods (see section 3.2.2) as can be seen in figure 5.7.

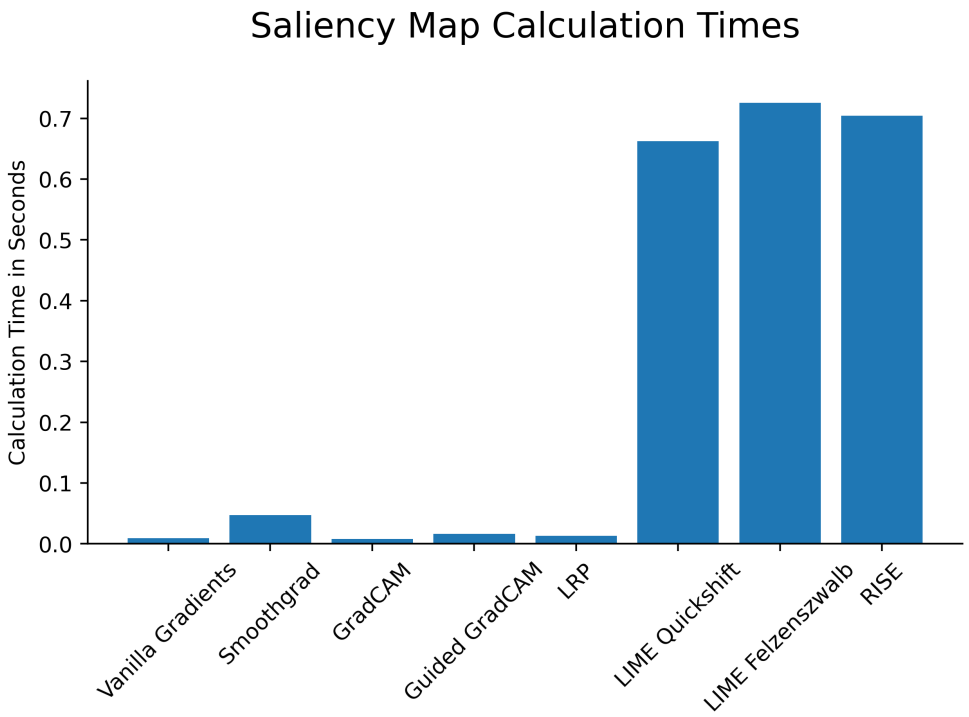


Figure 5.7: The calculation times for one saliency map for each method. The gradient-based methods are much faster with SmoothGrad taking the longest due to passing multiple frame-stacks through the Q-network. The perturbation-based methods LIME and RisE are much slower.

The computation time depends on the amount of Q-value estimations for input frame-stacks (passes through the Q-network), which is why SmoothGrad takes longer than for example vanilla gradients. The perturbation-based methods take so much longer because they pass significantly more (perturbed) frame-stacks through the Q-network, as the information to create a saliency map are pulled from the input-output tuples of the Q-network only, while the gradient-based methods make use of the model weights. Vanilla gradients, GradCAM, guided GradCAM and LRP run through the Q-network only one time, SmoothGrad 50 times (adjustable as parameter), but RisE passes 8000 (adjustable as parameter) perturbed frame-stacks through the Q-network, and LIME includes an image segmentation and linear regressions. The ∼30x faster computation of the gradient-based methods makes them more attractive and, furthermore, real-time

capable, as the time to create a saliency map with those methods is lower than $\frac{1}{24}$seconds, which is the framerate of the game.

The saliency map methods can be evaluated quantitatively by investigating them with the deletion procedure (see next section 5.1.3).

5.1.3 Saliency Map Evaluation with Deletion

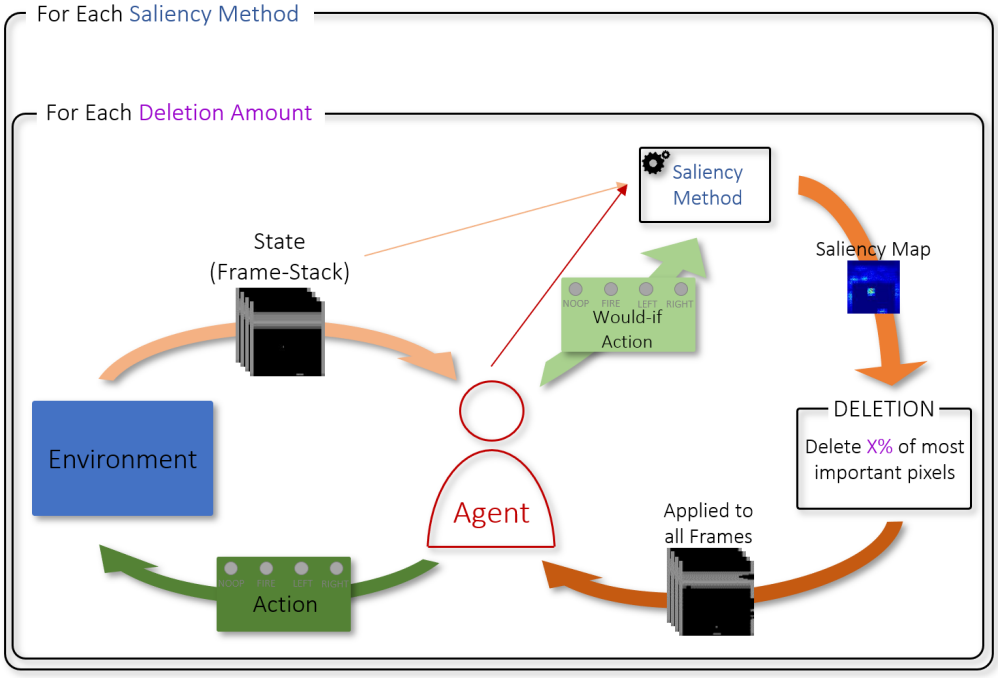


Figure 5.8: The deletion process. The most important pixels according to some saliency map method are deleted. The agent chooses one action based on the actual frame-stack for the saliency map creation only and one action based on the deleted frame-stack for the environment. The process is done at each time-step per episode for each deletion percentage for each method.

The deletion procedure (explained in section 3.3) is used to quantitatively evaluate the saliency maps as produced in section 5.1.2 and to answer research question 1. This is done by removing the most relevant information of the input frame-stack according to the saliency map and measuring how much worse the DQN agent plays. The agent needs to play one episode for each saliency map method for each deletion amount as shown in figure 5.8. Thus, the computation time to create the deletion graphs scales linearly with

the computation time to create the saliency maps (figure 5.7) and the interval between the deletion amounts. Especially for the perturbation-based methods LIME and RisE, this can become a lengthy matter.

Deletion percentage (to reach performance threshold)	Correctness
≤ 1%	Very high
1% – 2%	High
2% – 3%	Medium
3% – 4%	Low
> 4%	Very low

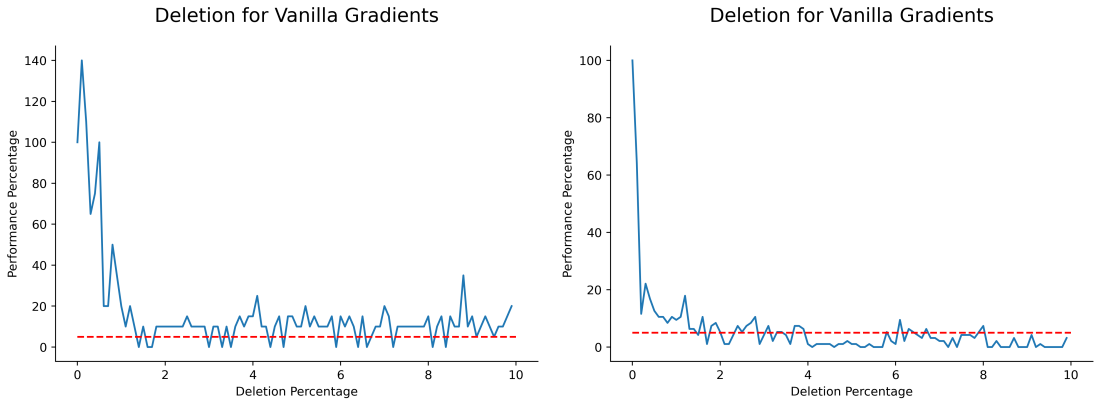
Figure 5.9: The degree of correctness of a saliency map depends on how much has to be deleted to reach the performance threshold, percentage-wise.

The degree of correctness for a saliency map method can then be quantitatively evaluated by how many important pixels (according to the saliency map) have to be deleted in order to reach the performance threshold. The relationship of deleted pixels versus the degree of correctness is shown in figure 5.9.

5.2 Results

In this section, the results of the saliency map method evaluation with deletion are presented. For each method, two agents are used. Agent 1 has a lower performance with an average reward of 38, while agent 2 has a higher performance with an average reward of 380. The performance threshold when enough important pixels are successfully removed lies at 5% of the initial agent performance, as explained in section 5.1.

5.2.1 Evaluation: Vanilla Gradients



(a) Agent 1

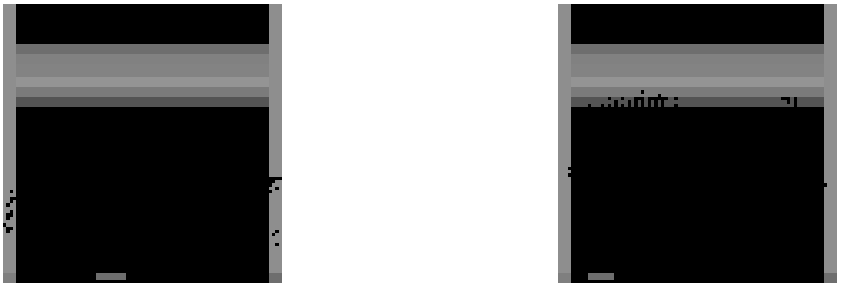
(b) Agent 2

Figure 5.10: The deletion graphs for vanilla gradients with agent 1 (a) and agent 2 (b). The threshold performance is reached at 1.4% deletion with agent 1 and 1.5% deletion with agent 2.

Observation

With vanilla gradients, agent 1 dropped under the 5% performance threshold with a deletion percentage of 1.4% and then oscillates around this threshold.

The performance of agent 2 first dropped under the performance threshold with 1.5% deletion. The performance then oscillates around the performance threshold before staying under it at 4% deletion. The oscillations are not as significant in comparison with agent 1. Between 6% and 8% deletion, the performance peaked above 5% a few times. As can be seen in figure 5.10, vanilla gradients deletes a few pixels of the lower bricks and the ball. Some frames, the ball is visible, which results in the ball "flickering".



(a) Agent 1, deletion: 1.4%

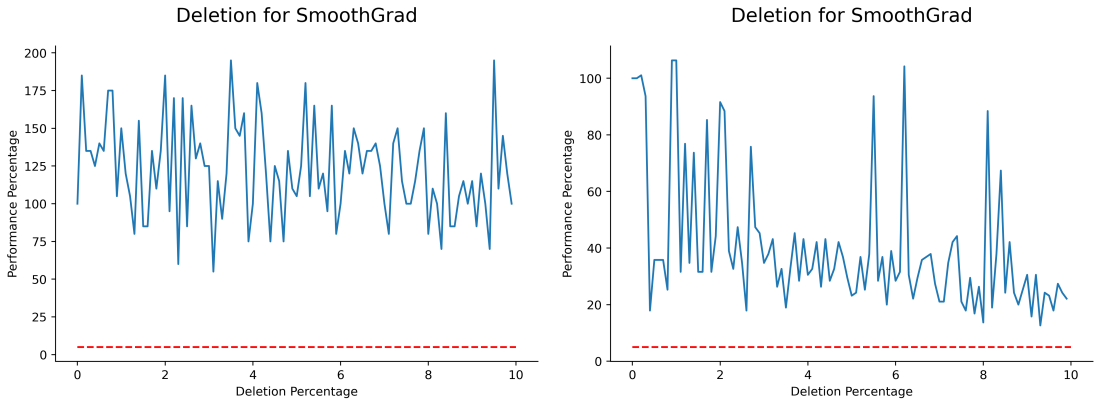
(b) Agent 2, deletion: 1.5%

Figure 5.11: Examples of deleted frames for vanilla gradients at performance threshold with agent 1 (a) and agent 2 (b). In both cases, the ball and a few other pixels are deleted.

Assessment

Vanilla gradients has a high degree of correctness for agent 1 and 2, only deleting 1.4% and 1.5% of the most important pixels to let the agent drop under the 5% performance threshold. The deletion curve is very smooth with only marginal oscillations, indicating a very high consistency. However, the curve with agent 2 is smoother than with agent 1, probably because vanilla gradients finds the relevant pixels more consistently when the neural network weights are more fine-tuned. Vanilla gradients intuitively deletes the ball, however not in every frame, which leads to the conclusion that the agent does track the ball, however not continuously at every frame, but only at specific ones. The deleted pixels at the lower bricks lead to believe that the agent has specific points of reference there.

5.2.2 Evaluation: SmoothGrad



(a) Agent 1

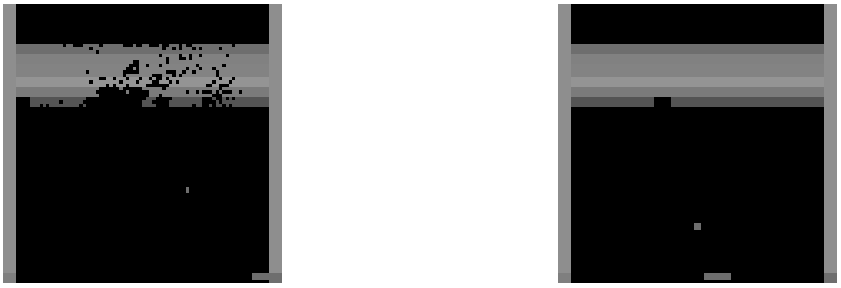
(b) Agent 2

Figure 5.12: The deletion graphs for SmoothGrad with agent 1 (a) and agent 2 (b). The threshold performance is not reached with both agents.

Observation

Agent 1 never touched the performance threshold within the tested deletion range between 0% and 10%, the lowest performance of 55% was reached with 3.2% deletion. There are major oscillations in the deletion curve, within those there are peaks at 200% of the original performance. SmoothGrad deletes pixels from anywhere in the frame, which sometimes includes the ball or the platform (see figure 5.13).

Agent 2 also never touched the performance threshold, the lowest performance (still 13%) was reached with 9.3% deletion. The deletion graph does have a decreasing tendency with major oscillations. SmoothGrad deletes areas which were background in the first place and deletes the ball occasionally (see figure 5.13).



(a) Agent 1, deletion: 9.9%

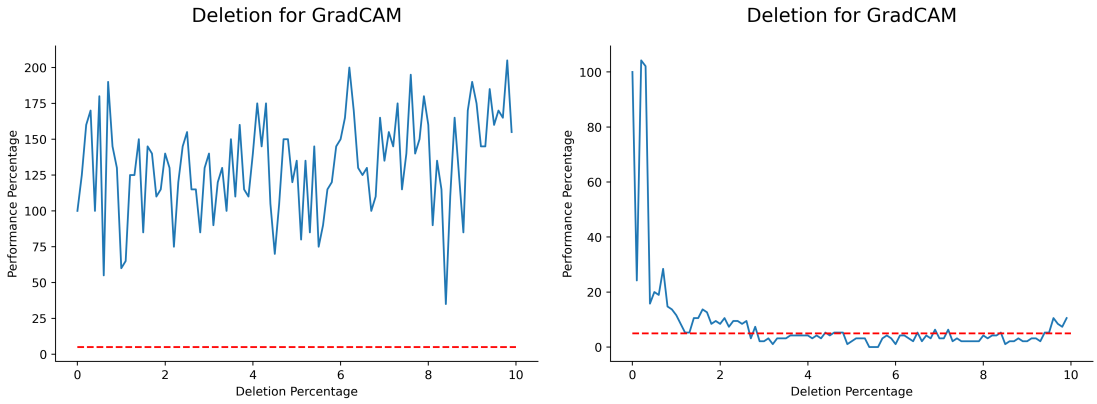
(b) Agent 2, deletion: 9.9%

Figure 5.13: Examples of deleted frames for SmoothGrad with agent 1 (a) and agent 2 (b). The area below the bricks is deleted, so that the resulting frame is not changed.

Assessment

SmoothGrad has a very low degree of correctness, as the performance of the agent never dropped under 5% within the tested deletion range between 0% and 10%. SmoothGrad furthermore has a very low consistency, as can bee seen by the many major oscillations in the deletion curve. This is not surprising considering SmoothGrad mainly deletes patches where there is background to begin with, so there is no change compared to the non-deleted frame. Occasionally, the ball is deleted too, so the ball disappears from time to time for one frame. Interestingly, the performance of agent 1 even went up drastically multiple times within the oscillations. This is probably because agent 1 is not trained well and the low correctness of SmoothGrad might lead to deleting pixels that change the behaviour of the agent so that it plays better by accident. The results are surprising as SmoothGrad should be less noisy and more consistent compared to vanilla gradients, which is not the case in these measurements. That might be because the methodology of adding noise to the input image and then averaging the gradients later isn't applicable to the Breakout reinforcement learning setting: The noise is applied to the whole image, including the empty area below the bricks. This empty area, where normally only the ball exists, may suddenly be filled with color so that the agent focuses on these new colored pixels. Now there may be considerable gradients to these pixels according to the DQN gradient calculation, which are then highlighted in the saliency map. Deletion then deletes these supposedly important pixels, leading to the described phenomenon.

5.2.3 Evaluation: GradCAM



(a) Agent 1

(b) Agent 2

Figure 5.14: The deletion graphs for GradCAM with agent 1 (a) and agent 2 (b). The performance threshold is reached with agent 2 but not with agent 1.

Observation

Agent 1 never touched the performance threshold, the lowest performance is 32% at 8.3% deletion. The performance fluctuates strongly between 32% and 200% and no downward trend is recognizable. GradCAM only deletes a horizontal stripe at the very top of the frame as can be seen in figure 5.15.

Agent 2 fell under the performance threshold at 2.7% deletion and stayed there. The oscillations are minor in this case, except for a well identifiable peak at 0.2% deletion. Within these oscillations, the performance peaks a little above the performance threshold from time to time. GradCAM deletes large patches anywhere in the frame, which sometimes includes the ball or the controlled platform (see figure 5.15).

(a) Agent 1, deletion: 9.9%

(b) Agent 2, deletion: 2.7%

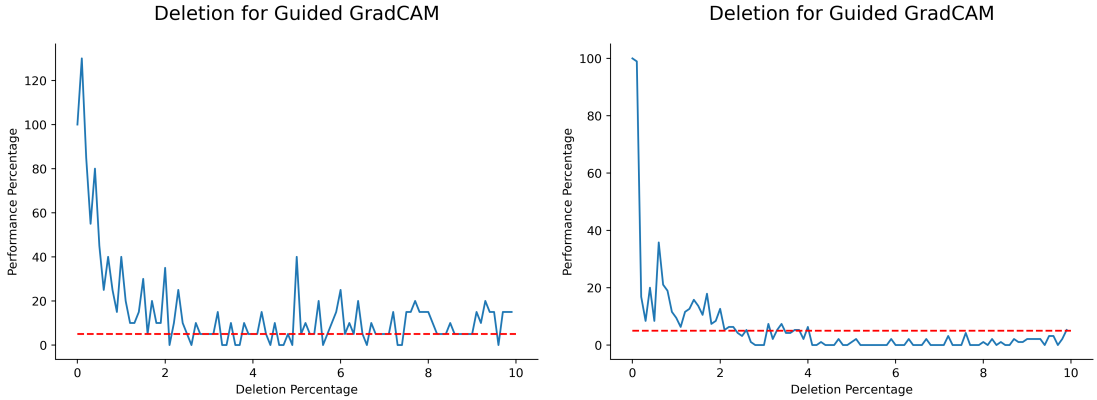
Figure 5.15: Examples of deleted frames for GradCAM at performance threshold with agent 1 (a) and agent 2 (b). With agent 1, the top pixel rows are deleted. With agent 2, large patches are deleted, which might sometimes include the ball or the platform.

Assessment

With agent 1, GradCAM always deletes the same top few pixel rows of the frame, which are not relevant for the game. Thus, the degree of correctness is very low and the deletion curve indicates no downward trend regarding the performance. Since agent 1 plays very imprecise, small changes in (even seemingly irrelevant parts of) the input frame-stack can lead to the agent suddenly performing a lot better or worse, which probably causes the strong oscillations in the deletion curve.

The deletion curve with agent 2 is a lot smoother, ignoring the spike at 0.2% deletion. The performance threshold is reached at 2.7% deletion, thus GradCAM has a medium degree of correctness in this case. The large patches which are deleted are in the neighborhood of the ball and the platform, sometimes including them and making them completely invisible from time to time. GradCAM seems to roughly target the correct areas of the frame, but isn't very precise in its doing.

5.2.4 Evaluation: Guided GradCAM



(a) Agent 1

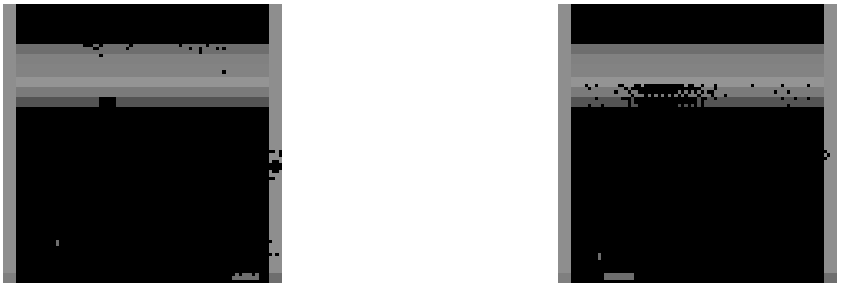
(b) Agent 2

Figure 5.16: The deletion graphs for Guided GradCAM with agent 1 (a) and agent 2 (b). The performance threshold is reached at 1.6% deletion with agent 1 and at 2.4% deletion with agent 2.

Observation

The performance threshold with agent 1 is reached at 1.6% deletion, however, the deletion curve is afflicted with oscillations and most data points are above the performance threshold from there on. The downward peaks of the oscillations come under the performance threshold. Adjacent pixels in various areas are deleted, sometimes including parts of the ball or the platform.

Agent 2 reaches the threshold performance at 2.4% deletion, but the deletion curve is a lot smoother and the performance stays below the threshold from there on. There is one exception between 3% and 4% deletion, where 3 data points reach a tiny bit above the performance threshold. Afterwards, the graph stays very flat at nearly 0% performance. Like with agent 1, adjacent pixels in various areas are deleted, however more focused.



(a) Agent 1, deletion: 1.6%

(b) Agent 2, deletion: 2.4%

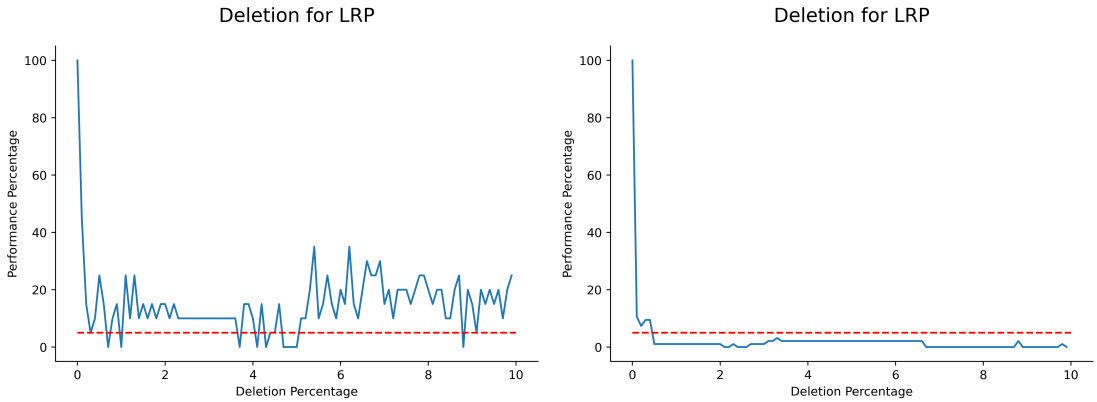
Figure 5.17: Examples of deleted frames for Guided GradCAM at performance threshold with agent 1 (a) and agent 2 (b). Pixels in a coherent area are deleted, sometimes including the ball or the platform.

Assessment

Guided GradCAM (with vanilla gradients as guide) lenses vanilla gradients towards the GradCAM focus, which helps agent 1 to now reach the threshold performance in contrast to GradCAM with agent 1 (see section 5.2.3). Reaching the performance threshold at 1.6% deletion indicates a high degree of correctness, however this result is only achieved due to a downward peak in the oscillations and even with that is not as good as just using vanilla gradients (see section 5.2.1). If the deletion curve would be smoothed, the threshold performance would not be reached at all. As indicated by the oscillations, Guided GradCAM is not very consistent too.

With agent 2, the degree of correctness is rated as medium, but the deletion curve is much smoother and the performance stays below the performance threshold, thus indicating a higher consistency. Nevertheless, just using vanilla gradients is superior in both the correctness and the consistency. The saliency maps of Guided GradCAM look nicer to the human eye because there are seemingly less randomly highlighted pixels, but vanilla gradients is in fact more accurate.

5.2.5 Evaluation: LRP



(a) Agent 1

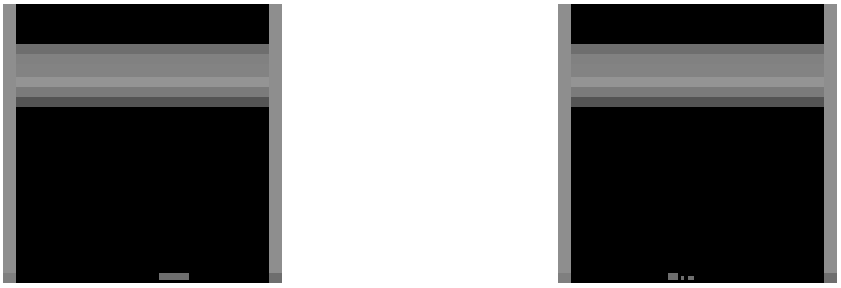
(b) Agent 2

Figure 5.18: The deletion graphs for LRP with agent 1 (a) and agent 2 (b). The deletion threshold is reached at 0.3% deletion with agent 1 and at 0.5% deletion with agent 2.

Observation

Agent 1 reaches the performance threshold at 0.3% deletion, but this is during a downward spike of oscillations. In general, the deletion curve is oscillating between 0% performance and 35% performance. LRP deletes the ball, though sometimes a few pixels of the ball are leftover.

With agent 2, the threshold performance is reached at 0.5% deletion. The deletion curve is extremely smooth and very flat after falling below the performance threshold, never touching the threshold again. LRP deletes the ball completely and furthermore a few pixels of the platform.



(a) Agent 1, deletion: 0.3%

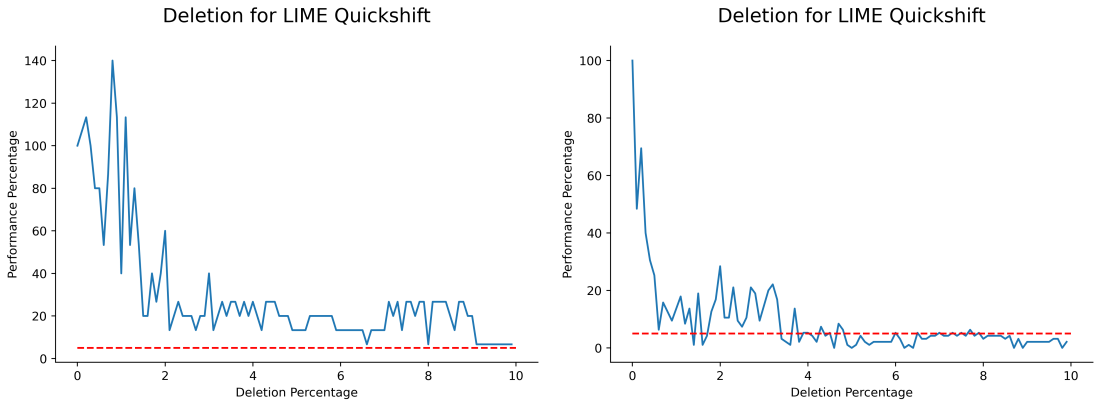
(b) Agent 2, deletion: 0.5%

Figure 5.19: Examples of deleted frames for LRP at performance threshold with agent 1 (a) and agent 2 (b). The ball is always deleted, together with some pixels of the platform.

Assessment

LRP has a very high degree of correctness in both cases. With agent 1, LRP is not precise enough to always delete the ball completely, resulting in the measured oscillations. With agent 2, LRP shows an almost perfect consistency as the deletion curve falls below the performance threshold almost immediately and stays there without any oscillations. LRP is able to delete the ball completely at every frame. The higher accuracy to delete the ball might be higher with agent 2 because the agent itself is more fine-tuned and focuses on the ball more precisely.

5.2.6 Evaluation: LIME Quickshift



(a) Agent 1

(b) Agent 2

Figure 5.20: The deletion graphs for LIME Quickshift with agent 1 (a) and agent 2 (b). With agent 1, the performance threshold is never reached while with agent 2, it is reached at 1.4% deletion.

Observation

LIME Quickshift with agent 1 never reaches the performance threshold but comes close to it multiple times after 6% deletion with the lowest performance being 7%. The deletion curve has major oscillations from 0% deletion to 2% deletion, from where on the oscillations get weaker. The performance drops until 2% deletion, from where on it stays at around 20% performance. LIME Quickshift deletes a large chuck of bricks on the upper right corner, as well as a few bricks below that.

With agent 2, the performance threshold is reached at 1.4% deletion. The deletion curve then oscillates strongly until 4% deletion, from where on it stays fairly flat at just under the performance threshold. LIME Quickshift deletes a horizontal stripe of middle bricks.

(a) Agent 1, deletion: 9.9%

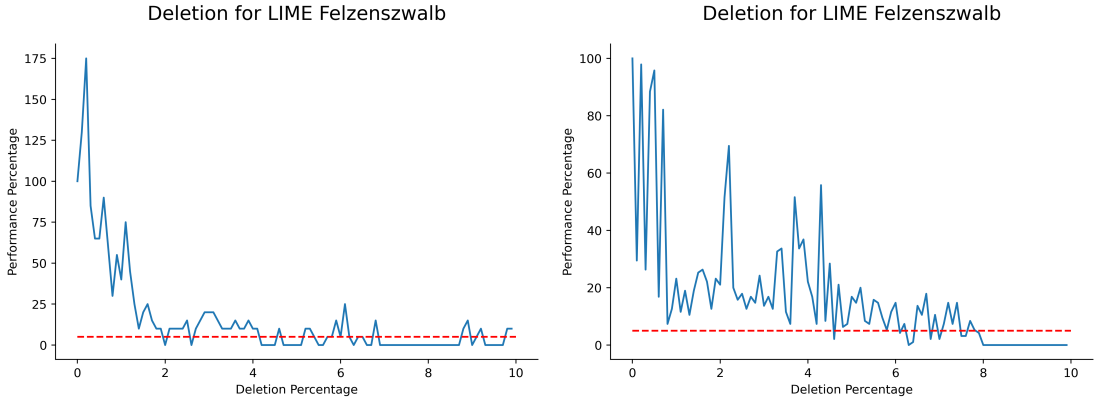
(b) Agent 2, deletion: 1.4%

Figure 5.21: Examples of deleted frames for LIME Quickshift at performance threshold with agent 1 (a) and agent 2 (b). With agent 1, most of the upper right bricks are deleted, with agent 2 just a row of the upper bricks.

Assessment

The degree of correctness is very low with agent 1, as the performance threshold is never reached. With agent 2, the degree of correctness is high according to the presented categorization (see figure 5.9). Both deletion curves indicate a medium to high degree of consistency. Much more interestingly, LIME Quickshift deletes chunks of bricks which are intuitively not important to keep playing the game at the same performance, but in fact the agents do play a lot worse. This is especially visible with agent 2 as shown in figure 5.21: A horizontal stripe of middle bricks is deleted while the ball and the platform are completely visible, still the agent only reaches 5% of its original performance. The expectation was that LIME Quickshift is highly incorrect due to the seemingly randomly segmented superpixels as shown in section 3.2.2. It seems that the agents have important reference points in these frame areas.

5.2.7 Evaluation: LIME Felzenszwalb



(a) Agent 1

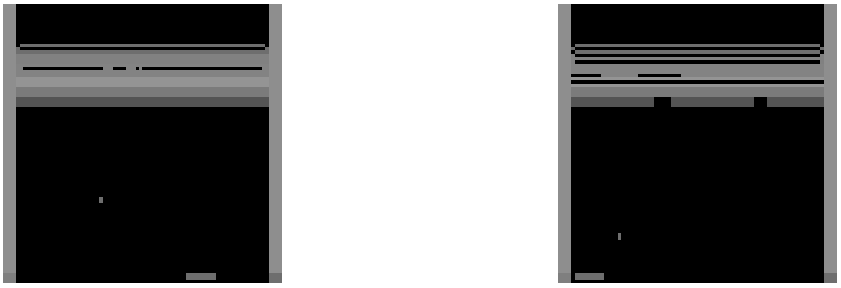
(b) Agent 2

Figure 5.22: The deletion graphs for LIME Felzenszwalb with agent 1 (a) and agent 2 (b). With agent 1, the performance threshold is reached at 2.0% deletion, with agent 2 at 4.6% deletion.

Observation

With agent 1, the performance threshold is reached at 2.0% deletion. Until there, the deletion curve fluctuates strongly. After 2.0% deletion, the deletion curve oscillates around the performance threshold, but not by much. LIME Felzenszwalb deletes horizontal stripes of the upper brick rows.

With agent 2, the performance threshold is reached at 4.6% deletion. The deletion curve oscillates strongly until 8% deletion, after which the performance stays well below the performance threshold. LIME Felzenszwalb deletes horizontal stripes of the middle and upper brick rows.



(a) Agent 1, deletion: 2.0%

(b) Agent 2, deletion: 4.6%

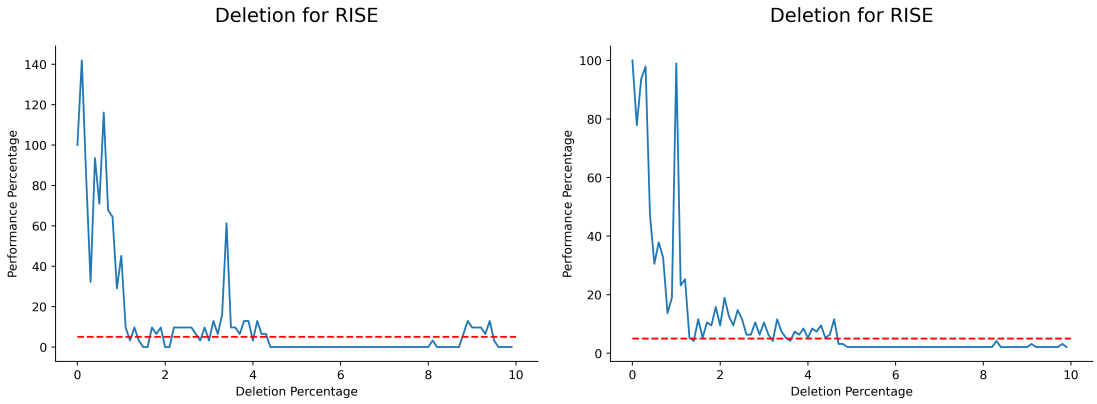
Figure 5.23: Examples of deleted frames for LIME Felzenszwalb at performance threshold with agent 1 (a) and agent 2 (b). Stripes in the upper bricks are always deleted.

Assessment

With agent 1, LIME Felzenszwalb has a high degree of correctness and seems fairly consistent according to the smoothness of the deletion curve. LIME Felzenszwalb deletes horizontal stripes of pixels in the upper bricks and nothing else, which is interesting considering the performance of a human would not suffer at all but those of the agent does. One would assume that LIME Felzenszwalb is highly inaccurate due to LIME weighting the upper bricks of the Felzenszwalb segmentation as most important and deleting them, but it seems the agent has important points of reference there.

The same applies to agent 2, where the LIME Felzenszwalb has a very low degree of correctness and is inconsistent according to the strong oscillations, but the deletion curve shows a downward trend and performance threshold is still reached. The deletion of horizontal pixel rows in the middle and upper bricks does greatly harm the agent performance.

5.2.8 Evaluation: RisE



(a) Agent 1

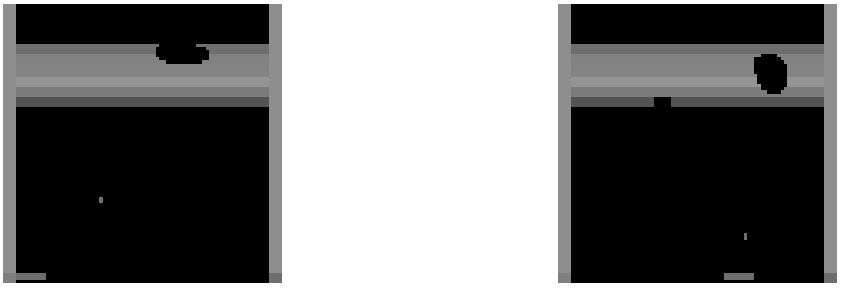
(b) Agent 2

Figure 5.24: The deletion graphs for RisE with agent 1 (a) and agent 2 (b). With agent 1, the performance threshold is reached at 1.2% deletion and with agent 2 at 1.4% deletion.

Observation

The deletion graph with agent 1 strongly oscillates until 1.2% deletion where the performance threshold is reached. It then hovers around the performance threshold before peaking upward once at 3.4% deletion. The performance then falls below the performance threshold again where it stays very flat until 8.7% deletion, where it starts oscillating a bit again. RisE deletes a contiguous patch of pixels of the upper right bricks.

The deletion graph with agent 2 also strongly oscillates until the performance threshold is reached at 1.4% deletion. From there it oscillates lightly above the performance threshold before falling below it at 4.6% deletion. From there on, the performance stays evenly below the threshold. RisE deletes a contiguous patch of pixels of the middle right bricks.



(a) Agent 1, deletion: 1.2%

(b) Agent 2, deletion: 1.4%

Figure 5.25: Examples of deleted frames for RisE at performance threshold with agent 1 (a) and agent 2 (b). A large patch of the upper right bricks is always deleted.

Assessment

RisE has a high degree of correctness with both agents. In both cases, the deletion curve indicates a low consistency until the performance threshold is reached, from where on the graphs are quite smooth, signifying a high consistency. RisE deletes a chunk of pixels in the bricks only, leaving the ball and the platform untouched, whereby the performance of a human player would not change but those of the agent does. This shows that the pixels in these patches are important for the agent.

5.3 Verdict

Saliency Map Method	Correctness	
	Agent 1	Agent 2
Vanilla Gradients	High	High
SmoothGrad	Very low	Very low
GradCAM	Very low	Medium
Guided GradCAM	High	Medium
LRP	Very high	Very high
LIME Quickshift	Very low	High
LIME Felzenszwalb	Medium	Very low
RisE	High	High

Figure 5.26: The degree of correctness for each saliency map method with agent 1 and agent 2. The verdict is based on the results of section 5.2.

With every saliency map methods except SmoothGrad, the performance threshold (5% of the original performance) is reached when applying deletion to them. All saliency map methods thus identify important pixels in principle (except SmoothGrad), be it more or less accurate. The amount of deletion required to reduce the agent performance to 5% varies and the degree of correctness may vary whether considering agent 1 or agent 2. SmoothGrad and Guided GradCAM are built upon vanilla gradients, but vanilla gradients is superior to both. The best perturbation-based saliency map method is RisE with a high correctness rating with both agents. The best gradient-based and overall saliency map method is LRP as it has a very high degree of correctness with both agents, no other method has a very high correctness rating in either case.

5.4 Discussion

Discussion Regarding Research Question 1

The idea of SmoothGrad to produce cleaner saliency maps by adding noise seems to be infeasible in the domain of reinforcement learning with Atari Breakout, probably

because the noise leads to higher gradients towards the noised pixels in the DQN gradient calculation. The gradient-based saliency map is thus based on a false assumption.

GradCAM or guided GradCAM are less correct than other gradient-based methods (except SmoothGrad). This may be because GradCAM and guided GradCAM look at the gradients with respect to the last convolutional layer in contrast to the input layer, making the saliency map coarser. The other saliency map methods are superior maybe due to the additional fineness gained by looking at the gradients with respect to the input layer.

LRP may be more accurate than the other gradient-based methods because it is not based on the actual gradients but rather backpropagates a relevance, which is more meaningful and valid: A high gradient at a pixel indicates a high relevance, yet the gradient might also be high at locations which were not important for the models decision, rather bigger weight adjustments just had to be made for this instance. LRP refines the gradient-based approach by backpropagating a relevance score.

RisE has a higher degree of correctness than the other perturbation-based methods LIME Quickshift and LIME Felzenszwalb. This is likely RisE looks at different areas in many different combinations, while LIME is dependent on the image segmentation calculated by Quickshift or Felzenszwalb. Quickshift and Felzenszwalb segment the image into superpixels, but those occupy a fixed area which LIME has to work with. LIME is thus reliant on the sensible-ness of the image segmentation algorithm. RisE on the other hand considers various areas determined by a huge number of random masks and therefore investigates the image more diversely.

Discussion Regarding Research Question 2

All gradient-based methods focus the ball, platform and lower rows of bricks, however they do it with varying precision. Since all of these methods construct their saliency map based on the gradient, it is traceable that they all focus on the same aspects of the frames.

All perturbation-based methods identify the upper right bricks as most important aspect of the frames. They all construct their saliency maps according to the same principle: Alter areas of the input-frames and observe which influence the model the most. It is thus traceable that they identify the same areas as most important, be it with small deviations.

The perturbation-based methods identify other relevant frame areas than the gradient-based methods, but both are correct. The focus of the gradient-based methods on the ball, platform and lower rows of bricks follows human intuition, while the focus on the upper right bricks of the perturbation-based methods gives incentives for further investigations. A human player would not be irritated by the removal of the upper right bricks, but the agents do in fact play a lot worse. The agents "think" in a different way than human intuition would suggest, even a high performing one, and humans are well advised not to blindly trust them. This insight is especially important for critical domains like healthcare.

Discussion Regarding Research Question 3

The highlighted areas between the gradient-based and perturbation-based methods differ, while both approaches can produce correct saliency maps (see section 5.3). Thus, only using the most correct saliency map methods doesn't capture all relevant areas of the image. Therefore, it is beneficial to use a gradient-based method and a perturbation-based method in order to retrieve as many relevant image areas as possible. Since in each category (gradient-/perturbation-based), one saliency map method is sufficient to capture all relevant image areas as possible by this category, it is sensible to use the most correct saliency map method of each category. The concluding recommendation is thus to use LRP and RisE.

6 Conclusion

In this chapter, a summary is given in section 6.1, which concludes the findings of this research and answers the research questions. Afterwards, an outlook for future work is given in section 6.2.

6.1 Summary

Various saliency map methods were evaluated quantitatively with regards to their correctness. The considered methods include vanilla gradients, SmoothGrad, GradCAM, guided GradCAM, LRP, LIME Quickshift, LIME Felzenszwalb and RisE. It is shown that perturbation-based methods take ∼30 times longer to calculate than gradient-based methods due to the amount of passes which were needed through the model. They were applied in a reinforcement learning setting, where a DQN agent plays Atari Breakout with the game frames as input. The quantitative evaluation was done by applying deletion to the observations with each saliency map method and measuring the drop in performance of the agent. This was done with two different agents to compare the results between a low performing a high performing agent.

Research Question 1: Highest Degree of Correctness

Research question 1 is answered as follows:

- The correctness of a saliency map not only depends on the method but also on the model and setting.
- The most correct saliency map method is LRP.

The results show that the degree of correctness does depend on the saliency map method and also on the model (agent) and the setting where it is used. In general, LRP has the highest degree of correctness.

Research Question 2: Differences in Highlights

Research question 2 includes the following findings:

- The gradient-based methods all find the same features as most important.
- The perturbation-based methods all find the same features as most important, but others than those oft he gradients-based methods.
- Investigating saliency maps from multiple methods gives more insights than just using one.

All the gradient-based methods find the same features as most important: The ball, platform and lower rows of bricks, however the precision in highlighting them varies. Likewise, all the perturbation-based methods find the same feature as most important: The upper right bricks. While these bricks are of little relevance for a human player, both agents rely on them to perform. This insight is only gained by investigating multiple saliency map methods with different approaches instead of just one. Machine learning models may not work by the system of rules that a human would expect, which makes XAI important and necessary.

Research Question 3: Recommendation

Research question 3 is condensed as follows:

- The recommendation is to use LRP and RisE.

With the findings related to research question 2, it becomes clear that it is insufficient to only use the most correct saliency map method: One saliency map method of one approach, gradient-based and perturbation-based, indeed makes another method from this approach obsolete, but should be combined with another method from another approach. Therefore, it is recommended to use LRP (as per the findings of research question 1) together with RisE, the most correct perturbation-based saliency map method.

6.2 Outlook

Further saliency map methods can be included to enlarge the scope of evaluated methods, such as DeconvNet [6] or Integrated Gradients [7]. In addition, SmoothGrad and guided GradCAM can be combined with other methods than Vanilla Gradients. Also, other environments than Atari Breakout can be used to confirm the findings or discover new

insights. Likewise, it can be investigated how the findings stack up against other agents than double DQN, for example SAC [32], DDPG [33] or TRPO [34]. Another idea is to try and use a model-based reinforcement learning algorithm by directly implementing the Atari Breakout logic into the algorithm, like with AlphaZero ([35]). Apart from that, the trained weights and biases can be examined to match these insights with the gradient-based methods. Another point to expand is to further investigate the different focus of gradient-based methods versus perturbation based methods. Different saliency map methods produce different results, of which multiple might be correct, therefore it might be possible to fuse all correct explanations together or and form an explanation which captures the complete workings of the machine learning model.

Bibliography

[1] Christoph Molnar. *Interpretable Machine Learning.* 2 edition, 2022.

[2] Meike Nauta, Jan Trienes, Shreyasi Pathak, Elisa Nguyen, Michelle Peters, Yasmin Schmitt, Jörg Schlötterer, Maurice van Keulen, and Christin Seifert. From anecdotal evidence to quantitative evaluation methods: A systematic review on evaluating explainable ai. *arXiv preprint arXiv:2201.08164*, 2022.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[4] Andrew Levy, Robert Platt, George Konidaris, and Kate Saenko. Learning multi-level hierarchies with hindsight. *7th International Conference on Learning Representations, ICLR 2019*, pages 1–16, 2019.

[5] Timothée Lesort, Natalia Díaz-Rodríguez, Jean-Franois Goudou, and David Filliat. State representation learning for control: An overview. *Neural Networks*, 108:379–392, 2018.

[6] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[7] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[8] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks. *arXiv preprint arXiv:1901.08644*, 2019.

[9] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[10] Vitali Petsiuk, Abir Das, and Kate Saenko. Rise: Randomized input sampling for explanation of black-box models. *British Machine Vision Conference 2018, BMVC 2018*, 1, 2019.

[11] Yi-Shan Lin, Wen-Chuan Lee, and Z Berkay Celik. What do you see? evaluation of explainable artificial intelligence (xai) interpretability through neural backdoors. *arXiv preprint arXiv:2009.10639*, 2020.

[12] Jasper van der Waa, Elisabeth Nieuwburg, Anita Cremers, and Mark Neerincx. Evaluating xai: A comparison of rule-based and example-based explanations. *Artificial Intelligence*, 291:103404, 2021.

[13] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. Citeseer, 1994.

[14] Tianmin Shu, Caiming Xiong, and Richard Socher. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 3:1–14, 2018.

[15] Muhammad Aurangzeb Ahmad, Carly Eckert, and Ankur Teredesai. Interpretable machine learning in healthcare. pages 559–560, 2018.

[16] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[17] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent. pages 1–40, 2022.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 6 2017.

[19] Zhaoping Li. A saliency map in primary visual cortex. *Trends in cognitive sciences*, 6(1):9–16, 2002.

[20] Lindsay Wells and Tomasz Bednarz. Explainable ai and reinforcement learning—a systematic review of current approaches and trends. *Frontiers in Artificial Intelligence*, 4:1–15, 2021.

[21] Linus Nilsson and Linus Nilsson. Explainable artificial intelligence for reinforcement learning agents explainable artificial intelligence for reinforcement learning agents. 2021.

[22] Arun Das and Paul Rad. Opportunities and challenges in explainable artificial intelligence (xai): A survey. pages 1–24, 2020.

[23] Laurent Itti, Christof Koch, and Ernst Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 20(11):1254–1259, 1998.

[24] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. 12 2013.

[25] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. 2017.

[26] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[27] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[28] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[29] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 13-17-Augu:1135–1144, 2016.

[30] Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Bach, and Klaus-Robert Müller. Evaluating the visualization of what a deep neural network has learned. 9 2015.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[32] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[33] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[34] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

A Appendix

Figure A.1: The loss of agent 1 during training.

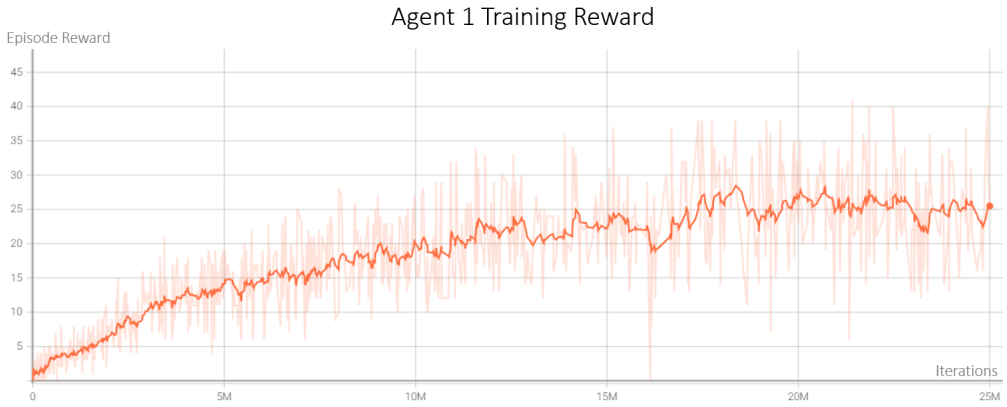


Figure A.2: The summed episode reward during training of agent 1. After each episode, the environment is reset.

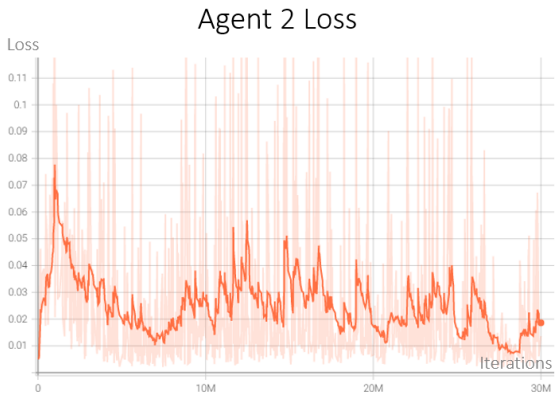


Figure A.3: The loss of agent 2 during training.

Figure A.4: The summed episode reward during training of agent 2. After each episode, the environment is reset.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original