

# Interpreteerbare besluitvorming in diep versterkend leren met een prototype boomstructuur

Xander De Visch

Studentennummer: 01902584

Promotoren: prof. dr. ir. Pieter Simoens, dr. Sam Leroux

Begeleider: Mattijs Baert

Masterproef ingediend tot het behalen van de academische graad van  
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2022-2023

# Dankwoord

Nu het einde van de masterproef is aangebroken zou ik graag even willen stil staan om enkele mensen te bedanken. Jullie hulp heeft het mogelijk gemaakt om deze veeleisende maar leerrijke periode te doorlopen en de thesis tot een goed einde te brengen.

Om te beginnen zou ik graag mijn begeleider Mattijs Baert willen bedanken voor de raad en feedback. De antwoorden en inzichten verkregen in de wekelijkse gesprekken doorheen het verloop van de masterproef hebben enorm geholpen bij het verwerken van nieuwe concepten het afgelopen jaar. U stond altijd paraat om mijn vragen te beantwoorden en me bij te staan. De tips en verduidelijking hebben me in de goede richting gewezen waarvoor hartelijk dank.

Daarnaast wil ik mijn promotoren prof. dr. ir. Pieter Simoens en dr. Sam Leroux bedanken voor de kans tot uitwerken van deze thesis. Tot slot wil ik nog mijn familie en vrienden bedanken voor de onschabare steun en aanmoediging doorheen het verwerken van het proefstuk.

Dank aan allen!

Xander De Visc

# **Toelichting in verband met het masterproefwerk**

Deze masterproef vormt een onderdeel van een examen. Eventuele opmerkingen die door de beoordelingscommissie tijdens de mondelinge uiteenzetting van de masterproef werden geformuleerd, worden niet verwerkt in deze tekst.

# Abstract

Versterkend leren (eng: *reinforcement learning*) is een deeltak van machinaal leren waarbij agenten in omgevingen getraind worden op basis van signalen in de vorm van beloningen. De gedachtegang is zeer gelijkaardig aan het opvoeden van kinderen. Op basis van een vooraf opgesteld beloningsschema ontvangt een actor een beloning wanneer hij gewenst gedrag vertoont, of een straf in de vorm van een negatieve beloning wanneer hij ongewenste acties onderneemt. Door de omgeving te verkennen en te leren uit voorgaande acties gebonden met beloningen en straffen, leert de agent een bepaald gedrag aan. Het uiteindelijke doel is de agent een policy (*staat-actie-mapping*) aan te leren die de beloning maximaliseert. Neurale netwerken worden vaak gebruikt om de optimale policy te bepalen, gezien het grote aantal mogelijke acties en toestanden in complexe omgevingen.

Een populair model dat reeds succesvolle resultaten gekend heeft in het verleden, is het deep Q-network. Dit model is in staat om de waardefunctie direct te benaderen vanuit observaties in de pixelruimte met behulp van convolutionele neurale netwerken. De waardefunctie berekent de waarde in een bepaalde staat rekening houdend met de kans op toekomstige beloningen. Het probleem met deze aanpak is de *black box*-natuur van convolutionele neurale netwerken die het moeilijk maakt om de aangeleerde policy te interpreteren en de redenen waarom keuzes genomen worden te evalueren. In deze thesis is er gekeken hoe de interpreteerbaarheid, verklaarbaarheid en transparantheid van dit model verbeterd kan worden door middel van een combinatie met de Neural Prototype Tree. Deze structuur bevat een boomstructuur van prototypes. Een prototype is een tensor in de latente ruimte die kan worden voorgesteld als een deel van een trainingsafbeelding. Door de waardefunctie te benaderen aan de hand van een aangepaste versie van de Neural Prototype Tree die getraind kan worden aan de hand van Q-waarden (waarden die aangeven hoe goed acties zijn indien uitgevoerd in een bepaalde staat), wordt een globaal interpreteerbare visualisatie verkregen die de optimale policy in kaart brengt. Aan de hand van de besluitvorming in de vorm van een visuele beslissingsboom, krijgt de gebruiker van het model een duidelijk zicht op de redenen waarom de policy kiest voor bepaalde acties in verschillende toestanden.

# Interpretable deep reinforcement learning policies using prototype trees

Xander De Visch

Promotors: prof. dr. ir. Pieter Simoens, dr. Sam Leroux

Supervisor: Mattijs Baert

**Abstract**—Reinforcement learning is a subfield of machine learning where agents are trained in environments based on signals in the form of rewards. The thought process is very similar to raising children. Using a reward scheme, an actor receives a reward when desired behavior is followed or a punishment in the form of a negative reward when actions are chosen that are not desired. By exploring the environment and learning from previous actions associated with rewards and punishments, the agent learns a certain behavior. The ultimate goal is to teach the agent a policy (state-action mapping) that maximizes the reward. Neural networks are often used to determine the optimal policy, given the large number of possible actions and states in complex environments.

One popular model that has achieved successful results in the past is the Deep Q-network. This model is capable of directly approximating the value function from pixel space observations of the state using convolutional neural networks (CNNs). The value function calculates the value in a particular state, taking into account the probability of future rewards. The problem with this approach is the black box nature of CNNs, which makes it difficult to interpret the learned policy and evaluate the reasons behind the choices made. In this paper, there is explored on how the interpretability, explainability, and transparency of this algorithm can be improved by combining it with the Neural Prototype Tree. This structure contains a tree-like structure of prototypes. A prototype is a tensor in the latent space that can be represented as part of a training image. By approximating the value function using a modified version of the Neural Prototype Tree that can be trained based on Q-values (a value that represents how good an action is, taken at a state), a globally interpretable visualization is obtained that maps the optimal policy. By using a visual decision tree for decision-making, the user of the model gains a clear understanding of the reasons why the policy chooses certain actions in different states.

**Index Terms**—Reinforcement Learning, Deep Q-network, interpretability, prototype, Neural Prototype Tree

## I. INTRODUCTION

Following the groundbreaking revolution of deep learning in 2012, particularly in the field of computer vision with the introduction of AlexNet, which achieved a remarkable 10% reduction in top-5 error rate using deep neural networks, a multitude of questions arose in the domain of reinforcement learning. The traditional techniques like Q-learning and tabular settings could not handle problems with large state- and action space. Seeing the breakthrough in supervised learning raised the question if the traditional RL-methods could be com-

bined with deep learning and be applied on high-dimensional problems, such as controlling a car or a robot that consists of a much larger domain where traditional RL algorithms struggle to cope. Mnih et al. managed to combine the two in the paper "Playing Atari with Deep Reinforcement Learning" [1]. In this work the first model was presented that could train the policy of an agent directly through images using convolutional neural networks. The model managed to get impressive results on 7 Atari 2600 video games. In 3 out of 7 instances, it even outperformed human experts. Since this paper the field has seen an increase in popularity and number of publications. This resulting in even more incredible achievements like AlphaGo, a reinforcement learning agent that beat the world champion in the game Go, a very popular board game in China, OpenAI Five, an agent that defeated a team of experts in a very complex strategy video game, Dota 2 and numerous publications of autonomous robots. Despite the results, reinforcement learning is still underutilized in business and corporations. The fact that the agents are trained using neural networks creates policies with a typical black box nature. Deploying these agents in critical environments like self-driving cars or healthcare without having a good insight on the reasons why an agent executes a specific action can be catastrophic. This leads to a crucial demand in more explainable and interpretable models. In this paper there is looked at how DRL-techniques like DQN could be combined with more interpretable models like decision trees to create a model that delivers global intrinsic interpretability.

## II. RELATED WORK

Most of the models that try to interpret and explain policies of deep learning agents or deep learning models in general work in a post-hoc way. This means that the parameters of a model are estimated after training a model. By evaluating and testing the logic of a trained model a representation of the inner workings is being created. This however only creates an approximation of the mechanisms of a model. Ad-hoc methods like the Neural Prototype Tree [2] use methods that contain structures that are transparent by design like decision trees resulting in a more verifiable decision making process.

## III. SETUP ENVIRONMENT AND TRAIN DQN AGENT

As intermediary step for the combination of the DQN agent and the ProtoTree, the ProtoTree will first be trained on labeled observation images. To do this first an agent is trained in

an environment using a DQN with a shallow CNN. The DQN algorithm is created with the Pytorch framework. The environment where the agent will be deployed in exists of a grey path and is created with the Gym library. The agent in the form of a blue cube has to navigate around the path to reach the target in the form of a green cube. The agent can take four possible actions (up, down, left and right) every step. The reward scheme consists of a dense system where the agent receives a reward of the difference in y-coordinate between the position from the state after taking an action and the state before taking the action. This because a sparse reward scheme consisting of a reward of -1 at every step except for when the agent reaches the target did not provide good results. The agent cannot move beyond the boundaries. When an action is chosen that would move the agent over the edge, the agent gets obstructed and no reward is received. The agent also receives a high reward of 1000 when reaching the target.



Fig. 1. Left: example of extracted screen of area around agent, right: image of complete environment

The agent has to learn to move up to reach the target while only having access to observations with dimensions of 3x3 of an area centered around the agent that can be seen in figure 1. To train the DQN-agent the following procedure is followed:

1) First, a preprocessing step is performed on the game screen, followed by providing a partial version of this screen as a state to the DQN.

2) Next, an action is chosen using the  $\epsilon$ -greedy algorithm. With a probability of  $\epsilon$ , a random action is chosen, and with a probability of  $\epsilon - 1$ , the action with the maximum Q-value is selected.

3) After executing the chosen action in the current state, the state transitions to a next state, and the agent receives a reward. The transition consists of a 4-tuple of the reward, the original state, the taken action, and the next state, which is then stored in the replay buffer.

4) Subsequently, a random batch of transitions is sampled, and the loss function is computed. This loss function is the mean squared error (MSE) between the Q-target value and the predicted Q-value. Following this, a gradient descent step is performed on the prediction network to minimize the loss value.

This procedure is repeated a maximum of 300 times in an epoch, after that the environment is reset. An epoch ends when

the agent reaches the target or when the maximum number of 300 actions is reached. The number of epochs corresponds to the number of games the agent is trained on. The result of the training can be seen in figure 2. In the graph it is visible that the number of actions in an episode steadily drops until around the 400th episode, after that the number stagnates for a moment until the 600th episode. The number of actions in an episode increases to a point that the agent can't even reach the target anymore. This phenomenon is known in RL as policy collapse. This is a problem that occurs very often because the DQN uses a non-linear function approximation in combination with temporal difference which creates a very unstable learning process. To obtain a trained policy that is able to traverse the path without colliding with the border, early stopping is used. Instead of continuing training, the training process is stopped when the performance significantly decreases.

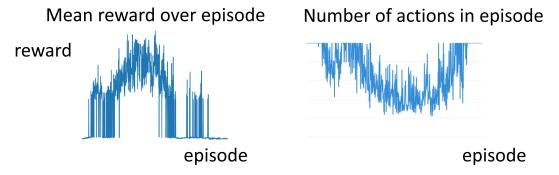


Fig. 2. Left: total number of actions in episode, right: mean reward over episode

#### IV. TRAINING THE PROTOTREE IN A SUPERVISED WAY USING LABELED OBSERVATION IMAGES

After obtaining a trained policy that uses a DQN to find its way through the path while only having access to the observation images of areas around the agent, the ProtoTree was trained using labeled observation images. To create the dataset, the images are automatically labeled by testing the policy on the environment after training, where at each step the optimal action is chosen ( $\epsilon = 0$ ) and the weight optimization process is disabled. This results in a dataset of observation images with labels of the suggested actions from the policy. The next step is to train the ProtoTree on this dataset. In Figure 3, reproduced from the paper "Neural Prototype Trees for Interpretable Fine-grained Image Recognition" [2], the forwarding process to obtain a class probability can be observed as a training sample traverses through the prototype tree. The samples in the form of observation images are passed through the CNN structure to obtain the feature maps, denoted as  $z$ . Afterward, the feature maps are divided into patches of shape  $H \times W \times D$ , ensuring that the patches have the same shape as the prototypes. At each level the patch that has the highest similarity with the prototype is determined based on a generalized form of convolution to determine the routing. The prediction is then determined by multiplying the arrival probability in a leaf with the class probabilities obtained by normalizing the logits using the softmax function. The weights of the CNN and parameters of the prototypes are optimized

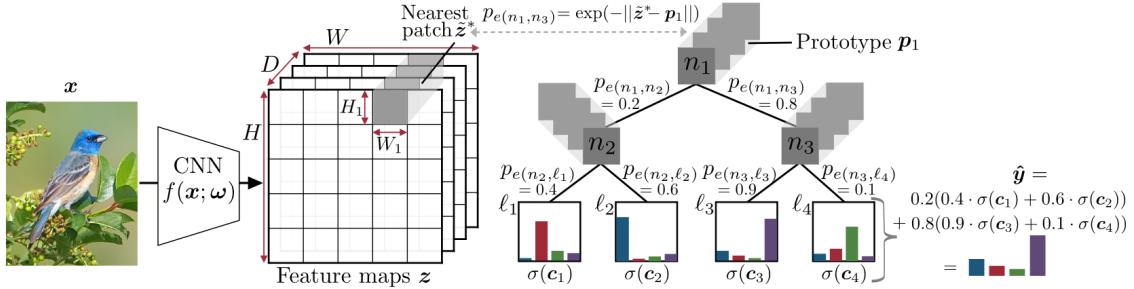


Fig. 3. forwarding process of the ProtoTree to obtain a class probability prediction [2]

with Stochastic Gradient Descent (SGD) by minimizing the cross-entropy loss between the predicted class distribution and the labels. Although it is possible to also train the class leaves  $c_l$  with SGD, Nauta et al. opted for a derivative free optimization process. In the paper "Deep Neural Decision Forests" [3], Kortschieder et al. observed that the optimization of parameters in the leaves of a decision tree is a convex optimization problem. Nauta et al. transferred this reasoning to the logits of the ProtoTree, resulting in the following formula:

$$c_{i,j}^{(t+1)} = \sum_{(x,y) \in T} (\sigma(c_l^{(t)}) \odot y \odot \pi_l) \odot \hat{y}. \quad (1)$$

The CNN that is used is a pretrained VGG-11. Instead of unfreezing the parameters of the VGG after a period of 30 epochs, all parameters were kept frozen except the last convolutional output layer with shape of 1x1. Training the ProtoTree with this adaptation resulted in an accuracy of 96 %.

In Figure 4 the visualization can be seen. The root node checks if a part of the path is located underneath the agent. If this is not the case, the tree decides to move downwards. This aligns with the choice of the trained policy, which had learned to approach the goal from above after reaching the end of the path. However, if a part of the path is visible directly below the agent, the traversal moves to the child node. In this node, the prototype represents a part of the road with a section of white background in the bottom right. In the decision-making process, this is generalized to examining whether there is a part of the road visible with white pixels to the right of the grey road. If this is not the case, the tree decides to move to the right. Finally, in the leaf node, it checks if there is a segment of the road visible with the background on the left side. If so, the tree decides to move upward; otherwise, it chooses to move to the left.

## V. COMBINATION OF DQN AND PROTOTREE

Now that it has been observed that training the ProtoTree in a supervised manner on the observation images is possible, the next step was to look at how the ProtoTree could be internally

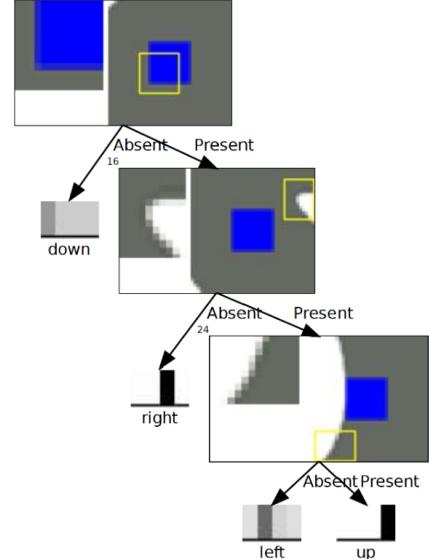


Fig. 4. visualization of ProtoTree trained on dataset labeled with actions of a trained policy

used in the DQN-algorithm. To do this some changes had to be made to the ProtoTree. The weights of the CNN and the prototypes are learned through SGD by minimizing the cross-entropy loss function between the predictions and the labels. However, in an RL environment, the ProtoTree does not have access to labels but only receives reward signals. To train based on these rewards, the optimization process has been adapted.

In Figure 5, a visualization of this adapted working method can be seen. Similar to the optimization of the simple CNN used in III, the first step involves retrieving a batch of transition tuples from the memory buffer. The samples in the form of observation images, are passed through the VGG-11 structure to obtain the feature maps, denoted as  $z$ . In chapter IV, it was found that leaving the weights of this CNN unchanged yielded better results, so they are not optimized here either.

Afterward, the feature maps are again divided into patches of shape H x W x D, ensuring that the patches have the same shape as the prototypes. At each level, the patch that has the highest similarity with the prototype is determined based on a generalized form of convolution to determine the routing. In the implementation of the ProtoTree proposed by Nauta et al., the prediction was then determined by multiplying the arrival probability in a leaf with the class probabilities obtained by normalizing the logits using the softmax function. In this adapted version, the normalization using softmax is omitted. The logits are directly multiplied by the arrival probabilities in the leaves. These values are considered as the Q-values. Similar to traditional RL algorithms, the  $Q_{estimate}$  is selected based on the chosen action. The formula for determining the Q-estimate is then:

$$Q_{estimate} = \sum_{l \in L} c_{l,a} \cdot \pi_l(f(x; \omega)). \quad (2)$$

To stabilize the training process, a target network is utilized. This target network is a cloned version of the ProtoTree. The weights of the actively optimized ProtoTree are copied to the target network at a fixed frequency. The observation images of the next state, denoted as  $s'$ , obtained after taking the chosen action, are passed through the target network. This batch of images undergoes the same steps until the leaf nodes as  $Q_{estimate}$ . After summing over all the leaves, the action is chosen based on the index of the highest value instead of the action from the tuple. This value is then multiplied by the discount factor  $\gamma$  and summed with the obtained reward, as shown in the formula below:

$$Q_{target} = r + \gamma \cdot (\max_a \sum_{l \in L} c_l \cdot \pi_l(f(x; \omega))). \quad (3)$$

Because equation 1 is geared towards classification problems and this RL-problem is more a regression problem, the class logits are not trained with a derivative free approach. Instead the class logits together with the prototypes and the add on layer are trained by minimizing the mean squared error (MSE) loss between  $Q_{estimate}$  and  $Q_{target}$ .

In figure 6 the results are seen of training the DQN agent by using the ProtoTree internally. Unlike the DQN agent with the simple CNN, this implementation manages to consistently achieve the goal. The policy of the agent with the ProtoTree successfully reaches the goal after about 30 episodes. Afterward, the average reward increases until the optimal path is found. This point is usually reached around 50 episodes. Once the optimal path is reached, the policy maintains its choice. The exploration factor sometimes leads to a slightly lower average reward but does not prevent the agent from reaching the goal or causing a decline in subsequent episodes. The reward remains stable. In general it can be assumed that the function approximation for the Q-values, when using the ProtoTree, leads to a much more stable learning process compared to approximating the Q-values with a simple CNN.

In figure 7 you can see on the left the visualization of a trained ProtoTree using the supervised way and on the right a visualization of a ProtoTree based on the Q-values.

The selected prototypes, after visualization in both ways, are usually very similar. The chosen prototypes contain parts that capture a portion of the agent and the road or road edges. However, the leaf nodes of the ProtoTree trained using the supervised method are more specific to a particular action. Nevertheless, the performance remains highly similar. In contrast to the ProtoTree trained with Q-values, the supervised tree only observes images that are part of the optimal path. On the other hand, the ProtoTree from the DQN integration receives a random batch of states visited in the past, which includes parts that are not part of the optimal path.

## FUTURE WORK

The current implementation of integrating DQN and ProtoTree uses a single environment image to predict an action. This allows solving problems where an agent needs to predict an action regardless of speed and motion. Many environments and problems require quick responsiveness and prompt adaptation to specific movements. To visualize these problems using the ProtoTree, the structure needs to be expanded so that actions are not chosen based on a single frame, but rather the difference between multiple images. In this adaptation, a way must be sought to handle this difference using prototypes.

This paper mainly focused on how the ProtoTree could be integrated into the DQN algorithm proposed by Mnih et al. In addition to the visual representation of a policy of a DQN or related agent, consideration could also be given to how the ProtoTree can be combined with other state-of-the-art RL models such as Proximal Policy Optimization (PPO) [4] or Asynchronous Advantage Actor-Critic (A3C) [5].

It should also be noted that after further research on the ProtoTree and, more specifically, the concept of prototypes, there has been substantial criticism on these concepts. In the paper "HIVE: Evaluating the Human Interpretability of Visual Explanations" [6], it was found that the prototypes trained in the latent space contribute to a better interpretation of the model's choices, even for incorrect predictions. However, they are not clear enough for users to distinguish between correct and incorrect predictions. In a subsequent work by Nauta et al., titled "PIP-Net: Patch-Based Intuitive Prototypes for Interpretable Image Classification" [7], the authors acknowledged these criticisms and responded as follows: "The previously introduced prototype models, including the ProtoTree from Chapter 4, can learn prototypes that do not align with human visual perception, meaning that the same prototype can refer to different concepts in the real world, making interpretation non-intuitive." [7].

In a continuation of the integration of DQN with the ProtoTree, it can be investigated how the structure can be

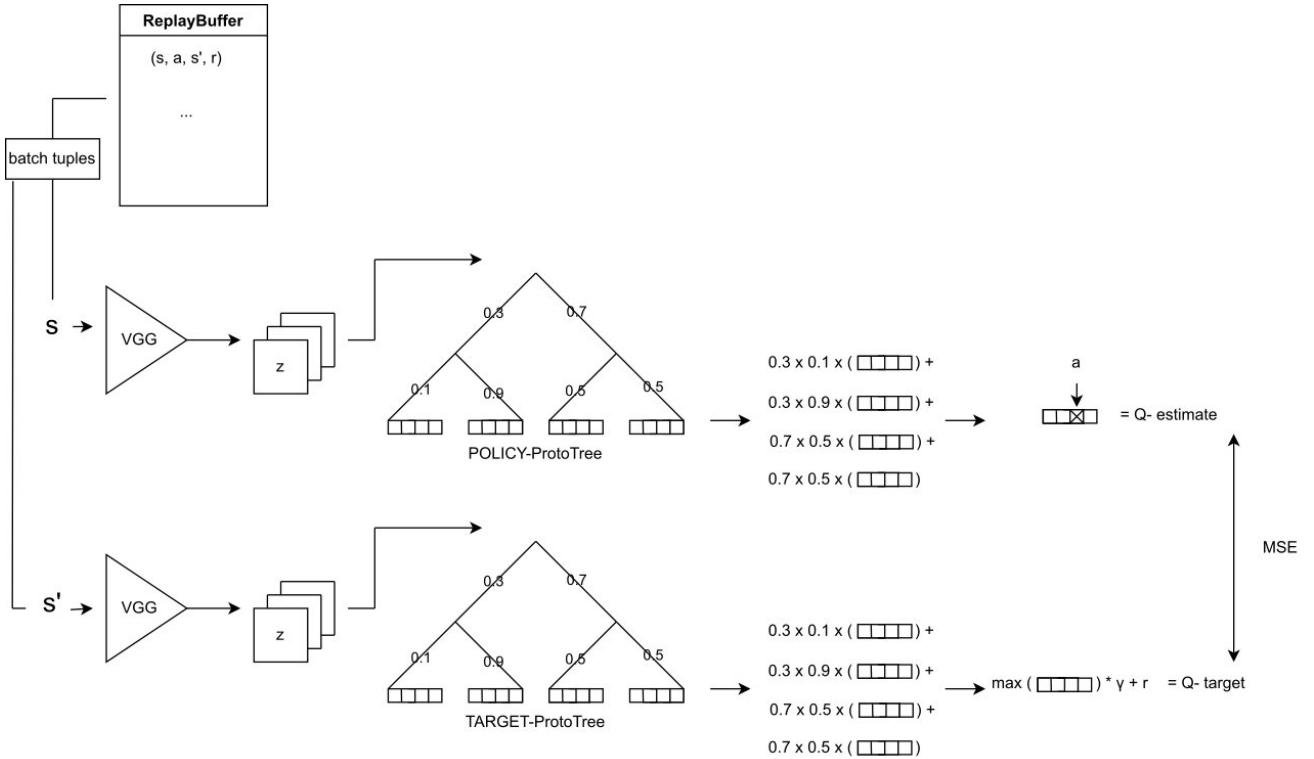


Fig. 5. Adapted optimization process of ProtoTree using mean squared error (MSE) loss between  $Q_{\text{estimate}}$  and  $Q_{\text{target}}$

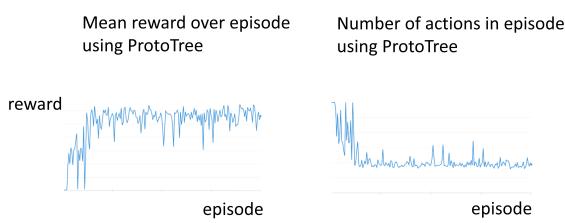


Fig. 6. Left: mean reward over episode using the ProtoTree, right: total number of actions in episode using the ProtoTree

modified so that the formation of prototypes aligns more closely with human visual perception.

## CONCLUSION

This paper investigated how the policy of a DRL algorithm can be represented using more explanatory structures such as decision trees. After studying the architectures of a DQN and the Neural Prototype Tree, abbreviated as ProtoTree, a successful integration has been achieved. The integration is able to train a policy that can guide an agent in the form of a blue square to reach a goal in the form of a green square in a minimum number of steps, without deviating from the path. The integration internally uses the ProtoTree to predict  $Q$ -values. By internally using the ProtoTree, a globally interpretable policy is obtained after the training process. Unlike post-hoc methods that visualize the choices of a policy after training, this approach provides immediate insight into the choices the policy will make when deployed. This global interpretability increases human confidence in the trained DQN instances. It provides direct understanding of the reasons why a particular agent chooses an action. This way, it becomes clear when the agent is making decisions based on correct assumptions or simply learning external factors and complex relationships. The integration can be applied to environments that do not require quick responsiveness and where actions can be inferred from a single image. Applying the integration to these environments provides clear insights and the ability to achieve the desired goal correctly.

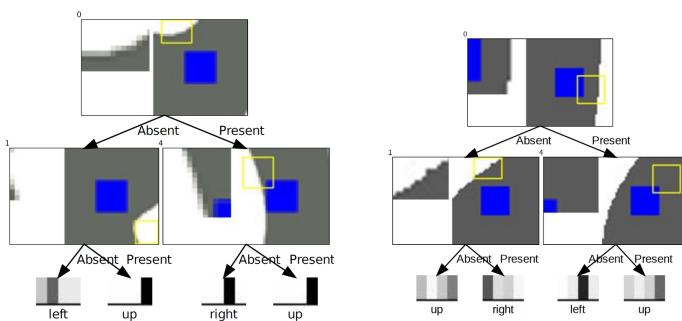


Fig. 7. Left: visualization of a trained ProtoTree using the supervised way, right: visualization of a ProtoTree based on the  $Q$ -values

Despite these advantages, the structure should still be taken with caution and handled carefully. Because the prototypes are constructed and optimized in the latent space and then converted to the pixel space based on environmental images, the selected prototypes do not align completely with human vision. This can result in prototypes being activated by different interpretations. These divergent interpretations pose a high risk in delicate environments such as self-driving cars and robotics in service and healthcare.

#### REFERENCES

#### BIBLIOGRAPHY

- [1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning.” [Online]. Available: <http://arxiv.org/abs/1312.5602>
- [2] M. Nauta, R. van Bree, and C. Seifert, “Neural prototype trees for interpretable fine-grained image recognition.” [Online]. Available: <http://arxiv.org/abs/2012.02046>
- [3] P. Kotschieder, M. Fiterau, A. Criminisi, and S. R. Bulò, “Deep neural decision forests,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1467–1475, ISSN: 2380-7504.
- [4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms.” [Online]. Available: <http://arxiv.org/abs/1707.06347>
- [5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning.” [Online]. Available: <http://arxiv.org/abs/1602.01783>
- [6] S. S. Y. Kim, N. Meister, V. V. Ramaswamy, R. Fong, and O. Russakovsky, “HIVE: Evaluating the human interpretability of visual explanations.” [Online]. Available: <http://arxiv.org/abs/2112.03184>
- [7] M. Nauta, J. Schlötterer, M. van Keulen, and C. Seifert, “Pip-net: Patch-based intuitive prototypes for interpretable image classification,” 2023.

# Inhoudsopgave

<b>Abstract</b>	iv
<b>Lijst van figuren</b>	xiv
<b>Lijst van tabellen</b>	xvi
<b>Lijst van afkortingen</b>	xvii
<b>Lijst van codefragmenten</b>	xviii
<b>1 Inleiding</b>	1
<b>2 Hoofdconcepten Reinforcement Learning</b>	2
2.1 Agent en omgeving . . . . .	2
2.2 Markov-beslissingsproces . . . . .	3
2.3 Policy $\pi$ . . . . .	4
2.4 Pad $\tau$ . . . . .	5
2.5 Winstwaarde R . . . . .	6
2.6 Staatwaarde V . . . . .	6
2.7 Staat-Actiewaarde Q . . . . .	6
2.8 Doel van RL . . . . .	7
2.9 Bellman-vergelijking . . . . .	8
2.10 Temporal Difference Learning . . . . .	9
2.11 Exploratie . . . . .	10
2.12 <i>Off-Policy learning</i> . . . . .	10
2.13 Q-learning . . . . .	11
<b>3 Deep Reinforcement Learning</b>	12
3.1 Atari Arcade Games . . . . .	13
3.2 Architectuur van een Deep Q-network (DQN) . . . . .	13
3.3 Correlatie . . . . .	15
3.4 <i>Experience Replay</i> en <i>infrequent weight updates</i> . . . . .	15
3.5 Overzicht van gehele architectuur . . . . .	16

3.6	Gym library . . . . .	17
3.7	Pytorch . . . . .	17
3.8	DQN toegepast op de Cart Pole-omgeving . . . . .	17
3.8.1	Replay Buffer . . . . .	18
3.8.2	Toestand . . . . .	18
3.8.3	Lagenstructuur van CNN . . . . .	19
3.8.4	Actiekeuze . . . . .	19
3.8.5	Optimalisatie van parameters van het CNN . . . . .	20
3.8.6	Resultaten . . . . .	21
3.9	Double Deep Q Network (DDQN) en <i>frame skipping</i> . . . . .	22
3.10	Overschakelen naar Frozen Lake-omgeving . . . . .	22
<b>4</b>	<b>Prototype Tree</b>	<b>27</b>
4.1	Interpreteerbaarheid aan de hand van prototypes . . . . .	27
4.2	Componenten . . . . .	28
4.2.1	Doelstelling . . . . .	28
4.2.2	CNN met voorgetrainde gewichten . . . . .	29
4.2.3	Feature maps z . . . . .	29
4.2.4	L2 Conv2D . . . . .	29
4.2.5	Probabilistisch gewicht . . . . .	29
4.2.6	Afgelopen pad . . . . .	30
4.2.7	Predictie . . . . .	30
4.3	Trainingsproces ProtoTree . . . . .	30
4.4	<i>Snoeien</i> en Visualisatie van eindresultaat . . . . .	31
4.4.1	Snoeien . . . . .	31
4.4.2	Visualisatie van eindresultaat met prototypes . . . . .	32
4.5	Resultaten na trainen van ProtoTree . . . . .	33
<b>5</b>	<b>Voorstellen van policy van DQN-agent</b>	<b>35</b>
5.1	Labels van afbeeldingen . . . . .	35
5.2	Opbouw van de dataset . . . . .	36
5.3	Eerste resultaat van voorstelling van policy . . . . .	36
5.4	<i>Inductieve bias</i> toevoegen . . . . .	38
5.5	Bevriezen van gewichten . . . . .	39
5.6	Eigen RL-omgeving om grotere dataset te verkrijgen . . . . .	42
5.7	DQN-agent trainen op eigen RL-omgeving . . . . .	43
5.8	Voorstellen van getrainde policy met ProtoTree . . . . .	45
<b>6</b>	<b>Combinatie DQN en ProtoTree</b>	<b>48</b>
6.1	ProtoTree trainen zonder labels . . . . .	48

6.2	Optimalisatie van bladknopen . . . . .	50
6.3	Visualisatie aan de hand van de geheugenbuffer . . . . .	50
6.4	Resultaten in verband met de policy . . . . .	50
6.5	Resultaten in verband met de visuele voorstelling . . . . .	52
6.6	Vergelijking DQN met ProtoTree en eenvoudig CNN . . . . .	54
6.7	Vergelijking supervised trainen van ProtoTree en trainen op basis van RL-methode . . . . .	55
	<b>toekomstig werk</b>	<b>57</b>
	<b>Conclusie</b>	<b>59</b>
	Ethische en maatschappelijke reflectie . . . . .	59
	<b>Referenties</b>	<b>61</b>
	<b>Bijlagen</b>	<b>63</b>
	Bijlage A . . . . .	64
	Bijlage A . . . . .	65
	Bijlage C . . . . .	66
	Bijlage C . . . . .	67
	Gebruik van artificiële intelligentie in masterproef . . . . .	68

# Lijst van figuren

2.1	Agent en omgeving [1] . . . . .	3
2.2	Voorbeeld van een eenvoudige MDP met drie toestanden: (groene cirkels), twee acties (oranje cirkels) en twee beloningen (oranje pijlen). [2] . . . . .	4
2.3	Voorbeeld van doolhofomgeving [3] . . . . .	5
2.4	Policy van robot [4] . . . . .	5
2.5	Transitiegraaf met mogelijke acties van policy $\pi$ [5] . . . . .	7
2.6	Exponential decay van $\epsilon$ [6] . . . . .	10
3.1	Een verzameling aan Atari 2600-omgevingen gesimuleerd via de Arcade Learning Environment. [7] . . . . .	13
3.2	Verschil Q-learning en Deep Q-learning [8] . . . . .	14
3.3	Schets van de infrequent weight updates van een DQN [9] . . . . .	16
3.4	Voorbeelden van verschillende omgevingen beschikbaar in de Gym library [7] . . . . .	17
3.5	Cart Pole-omgeving uit de gym library [7] . . . . .	18
3.6	Voorbeeld van een geëxtraheerd beeld uit de "Cart Pole"-omgeving . . . . .	20
3.7	Duration plot met op de y-as de duur dat Cart Pole zich kan rechthouden, uitgedrukt in actiestappen en op de x-as het aantal trainingsepisodes. . . . .	21
3.8	Trainingscurve met op de y-as de MSE-loss en op de x-as het aantal tijdsstappen verstreken in het algoritme. . . . .	21
3.9	Voorbeeld van de Frozen Lake-omgeving [7] . . . . .	24
3.10	Frame dat observatie van toestand voorstelt in het trainingsproces van agent in Frozen Lake . . . . .	25
3.11	Plot met beloningen van DQN-agent, met op de x-as de episodes . . . . .	25
3.12	Trainingscurve met MSE-loss . . . . .	26
4.1	Visuele voorstelling van een ProtoTree getraind op de CUB-200-2011 dataset bestaande uit 200 verschillende vogelsoorten [10] . . . . .	28
4.2	Componenten van de ProtoTree [10] . . . . .	28
4.3	Voorbeeld van visualisatieproces van een prototype [10] . . . . .	33
4.4	deelboom van ProtoTree getraind op CUB-200-2011 dataset . . . . .	34
5.1	Voorbeeld van afbeeldingen uit de dataset van omgevingsbeelden gelabeld met acties genomen door de policy . . . . .	36
5.2	Eerste resultaat van toepassen van ProtoTree op dataset met observatiebeelden gelabeld met acties van de policy . . . . .	37
5.3	Voorstelling van werkwijze van voorstelling van prototypes met inductive bias afhankelijk van de positie. . . . .	38

5.4	Plot die de loss-waarde weergeeft per epoch in het trainingsproces van de ProtoTree wanneer de gewichten van het CNN bevroren worden voor 30 epochs . . . . .	39
5.5	Resultaat van toepassen van ProtoTree op dataset met observatiebeelden gelabeld met acties van de policy, waarbij de gewichten van het CNN onaangepast blijven. . . . .	41
5.6	Beeld van RL-omgeving ontworpen met de Pygame-module . . . . .	42
5.7	Observatiebeeld gecentreerd rond agent . . . . .	42
5.8	Aantal acties per episode . . . . .	44
5.9	Gemiddelde beloning over alle acties tot op een bepaalde episode . . . . .	44
5.10	Gemiddelde beloning over episode genomen . . . . .	45
5.11	Visualisatie van getrainde ProtoTree . . . . .	46
5.12	Visualisatie van tweede getrainde ProtoTree . . . . .	47
6.1	Optimalisatieproces van integratie DQN met ProtoTree . . . . .	49
6.2	Aantal acties per episode in het trainingsproces aan de hand van de ProtoTree . . . . .	51
6.3	gemiddelde beloning over een episode genomen in het trainingsproces aan de hand van de ProtoTree . . . . .	51
6.4	Visuele voorstelling van policy-ProtoTree met maximale diepte 3 . . . . .	52
6.5	Visuele voorstelling van eerste policy-ProtoTree met maximale diepte 2 . . . . .	53
6.6	Visuele voorstelling van tweede policy-ProtoTree met maximale diepte 2 . . . . .	53
6.7	Visuele voorstelling van policy-ProtoTree waarvan de visualisatie van de prototypes groter zijn dan de afmeting van de agent. . . . .	54
6.8	Vergelijking van gemiddelde beloning over een episode genomen tussen DQN dat gebruik maakt van eenvoudig CNN en DQN dat gebruik maakt van ProtoTree . . . . .	55
6.9	Vergelijking van het aantal acties in een episode tussen DQN dat gebruik maakt van eenvoudig CNN en DQN dat gebruik maakt van ProtoTree . . . . .	55
6.10	Links: visuele voorstelling van supervised trainen van ProtoTree, rechts: visuele voorstelling van trainen van ProtoTree met RL-methode . . . . .	56
6.11	Verduidelijking van verschil in alignatie tussen de geleerde prototypes van bijvoorbeeld de ProtoTree en de prototypes geleerd in PIP-Net, die meer overeenkomen met keuzes die een mens zou maken. [11] . . . . .	58

# Lijst van tabellen

3.1	Overzicht van convolutionele lagen in DQN. . . . .	20
1	hyperparameters DQN-agent bij Frozen-Lake omgeving . . . . .	64
2	hyperparameters en argumenten voor het trainen van ProtoTree op omgevingsbeelden van eigen aangemaakte omgeving . . . . .	65
3	hyperparameters DQN-agent bij eigen aangemaakte omgeving . . . . .	66
4	hyperparameters DQN-agent gecombineerd met ProtoTree bij eigen aangemaakte omgeving . . . . .	67

# Lijst van afkortingen

**A3C** Asynchronous Advantage Actor-Critic.

**ALE** Arcade Learning Environment.

**CNN** Convolutional Neural Network.

**DDQN** Double Deep Q Network.

**DenseNet** Densely Connected Convolutional Networks.

**DNN** Deep neural networks.

**DQN** Deep Q Network.

**DRL** Deep Reinforcement Learning.

**GAN** Generative Adversarial Network.

**GPU** Graphical Processing Unit.

**MDP** Markov Decision Process.

**MSE** Mean Squared Error.

**PPO** Proximal Policy Optimization.

**ResNet** Residual Neural Network.

**VGG** Visual Geometry Group.

# **Lijst van codefragmenten**

# 1

## Inleiding

*Deep reinforcement learning* (DRL) is een vorm van machinaal leren en kunstmatige intelligentie waarbij machines kunnen leren van hun acties, zeer gelijkaardig aan hoe mensen leren van hun ervaringen. De agent observeert de huidige status van een omgeving, kiest een actie en ontvangt een beloningssignaal dat de kwaliteit van de gekozen actie weergeeft. Het doel is om een *policy* te vinden die de beloning maximaliseert.

DRL maakt gebruik van *deep neural networks* (DNNs). DNNs hebben de laatste jaren zeer sterke resultaten behaald en bleken specifiek op het gebied van computervisie superieur tegenover andere algoritmen in het domein van machinaal leren.

Het probleem is echter de *black box*-natuur van DNNs. Door de hoog dimensionele *feature*-ruimte en de complexe architectuur is het duidelijk dat het uitrollen van DRL-instanties in zeer kritieke omgevingen zoals zelfrijdende auto's, robotica in dienstsectoren, gezondheidszorg, etc. een gevoelige zaak kan zijn en delicate consequenties met zich mee kan brengen.

Een ander essentieel probleem is de verklaarbaarheid voor systemen die onderhouden worden door mensen. De onderhouders vertrouwen op deze systemen en moeten inzicht krijgen in faal gevallen. In het geval van *policy errors* is het cruciaal dat de oorsprong van het probleem a posteriori gekend is.

De doelstelling van de masterproef is *interpretable machine learning*-modellen onderzoeken en combineren met DRL. Interpretable machine learning-technieken kunnen een antwoord leveren op problemen zoals transparantheid, verklaarbaarheid en interpreerbaarheid van de DRL-systemen met een intrinsieke black box-structuur. Interpretable machine learning-technieken zoals bijvoorbeeld beslissingsbomen kunnen echter geen hoogdimensionele data zoals afbeeldingen en video's behandelen.

In deze thesis zal er een afweging gemaakt worden tussen de predictieve performantie van de DRL-technieken en de interpreerbaarheid van beslissingsbomen. De combinatie van interpretable machine learning-modellen en neurale netwerken tot een interpreerbaar DRL-algoritme ligt aan de basis van dit onderzoek. De Neural Prototype Tree, kortweg ProtoTree heeft het representatief vermogen en bevat een ingebouwde binaire boomstructuur. Doorheen de masterproef zal deze verder bestudeerd worden met als einddoel een werkende implementatie van dit model te interageren in een reinforcement learning-architectuur. Dit model wordt uiteindelijk getest op een eigen gecreëerde omgeving, aangemaakt met de Gym library (aangeboden door The Farama Foundation [12]).

# 2

## Hoofdconcepten Reinforcement Learning

In dit hoofdstuk worden de fundamenteen van reinforcement learning in kaart gebracht. Reinforcement learning is een subvak van machine learning die de laatste jaren veel aandacht heeft gekregen. Het doel van reinforcement learning-algoritmen is het leren van een optimale policy. In verschillende hedendaagse applicaties zoals zelfrijdende auto's, videospellen en robotica werden met deze algoritmes indrukwekkende resultaten behaald. In videospellen zijn ze reeds sterker geworden dan topspelers in bepaalde games. Hieronder volgt een overzicht van de belangrijkste concepten. De kennis over deze onderwerpen is hoofdzakelijk verworven aan de hand van het boek "Deep Reinforcement Learning"<sup>[1]</sup> geschreven door de heer Plaat.

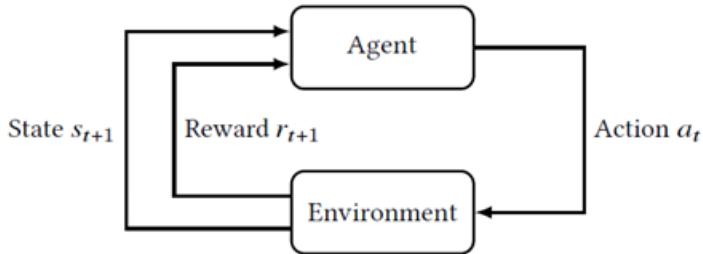
### 2.1 Agent en omgeving

De basisbouwstenen van een probleem in reinforcement learning (RL) bestaan uit de agent en de omgeving. De agent is een actor die verschillende keuzes kan maken in een specifieke omgeving. De omgeving bevindt zich in een zekere staat  $s_t$  op moment t zoals te zien in figuur 2.1. De agent heeft keuze tot verschillende acties  $a_t$  in de omgeving. Het resultaat van een actie zorgt voor een transitie in staat ( $s_{t+1}$ ) op een volgend tijdstip. Na de uitvoering van een welbepaalde actie zendt de omgeving een signaal terug in de vorm van een beloning r.

Het uiteindelijke doel is dat de agent een sequentie van acties leert die resulteert in maximale beloning. Dit wordt gedaan aan de hand van de policy-functie  $\pi$ . Deze functie geeft op basis van elke staat de best mogelijke actie terug voor die staat. Door verschillende acties uit te proberen, ondervindt de agent verschillende beloningen of straffen (negatieve beloningen). De agent leert door herhaaldelijk te interageren met de omgeving wat te doen in welke situatie.

In het geval van zelfrijdende auto's kan de auto met besturingssysteem worden voorgesteld als de actor, de omgeving als de openbare weg, de staten als verschillende verkeerssituaties en de acties in de vorm van manoeuvres die de auto kan doen. Veronderstel dat het doel van een auto is om punt B te bereiken vertrekende vanuit punt A waarbij de auto zich aan de verkeersregels moet houden. Een mogelijkheid is dat de auto een negatieve beloning ontvangt als hij een verkeersovertreding

## 2 Hoofdconcepten Reinforcement Learning



Figuur 2.1: Agent en omgeving [1]

maakt en beloningen ontvangt na het bereiken van bepaalde tussenstops.

### 2.2 Markov-beslissingsproces

Een Markov Decision Process (MDP), vernoemd naar Russisch wiskundige Andrey Markov (1856-1922) biedt een wiskundige omkadering om besluitvormingen voor te stellen die deels stochastisch en deels onder de controle van een besluitvormer vallen [2]. MDP's worden gebruikt in RL om sequentiële beslissingsproblemen te modelleren. Deze optimalisatieproblemen zijn oplosbaar met technieken van het dynamisch programmeren en kunnen worden geschat aan de hand van MDP's (zie 2.2).

MDP's beschikken over de Markoveigenschap die geldt als volgt: "de volgende staat hangt enkel af van de huidige staat en de gekozen actie en niet van historisch geheugen van voorgaande staten of van andere informatie". Dit is cruciaal voor RL-problemen omdat dit het beredeneren van toekomstige staten mogelijk maakt met enkel informatie van de huidige toestand.

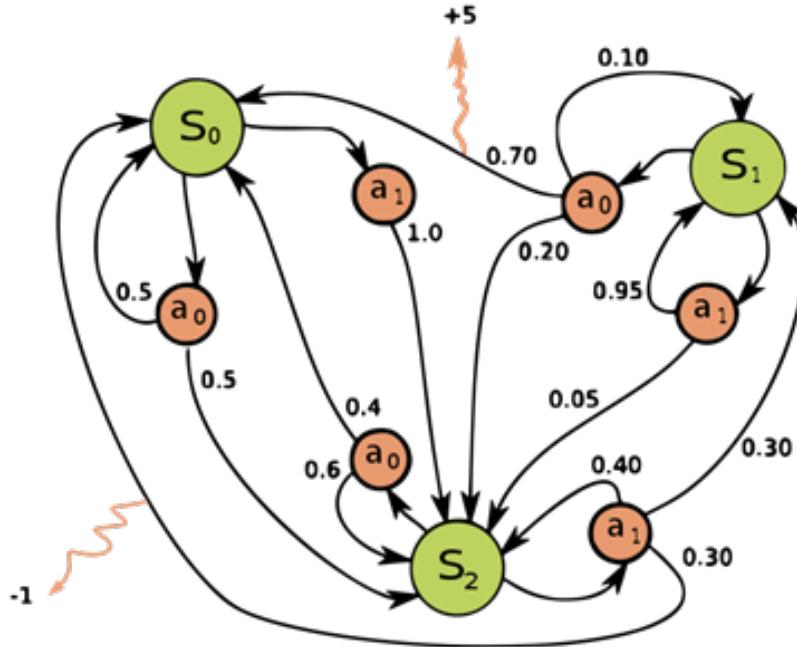
Formeel wordt een MDP in RL voorgesteld door middel van de 5-tuple  $(S, A, T_a, R_a, \gamma)$ :

- $S$  is een verzameling van toegelaten staten in de omgeving
- $A$  is een verzameling van toegelaten acties
- $T_a$  is de kans dat actie  $a$  in staat  $s$  op tijdstip  $t$  zal overgaan in de staat  $s'$  op tijdstip  $t+1$
- $R_a$  is de ontvangen beloning nadat actie  $a$  gezorgd heeft voor de overgang van staat  $s$  naar staat  $s'$
- $\gamma$  is de verminderingssfactor die het verschil in belang van huidige en toekomstige beloningen voorstelt

De verminderingssfactor  $\gamma$  geeft weer hoeveel belang er moet gehecht worden aan beloningen in de toekomst. Wanneer de opdracht continu is en voor een lang tot oneindig aantal seconden moet lopen, wordt de verminderingssfactor  $\gamma$  kleiner dan 1 gezet. Bij een omgeving waarbij een verkeerde actie kan resulteren in het einde van het spel, zal de lagere waarde van  $\gamma$  ervoor zorgen dat de beloningen op het huidig moment en in de nabije toekomst harder doorwegen dan in een ver vooruitzicht.

Bij episodische taken zoals in figuur 2.3, waarbij een agent het einde van een doolhof wil bereiken en enkel een beloning

## 2 Hoofdconcepten Reinforcement Learning



Figuur 2.2: Voorbeeld van een eenvoudige MDP met drie toestanden: (groene cirkels), twee acties (oranje cirkels) en twee beloningen (oranje pijlen). [2]

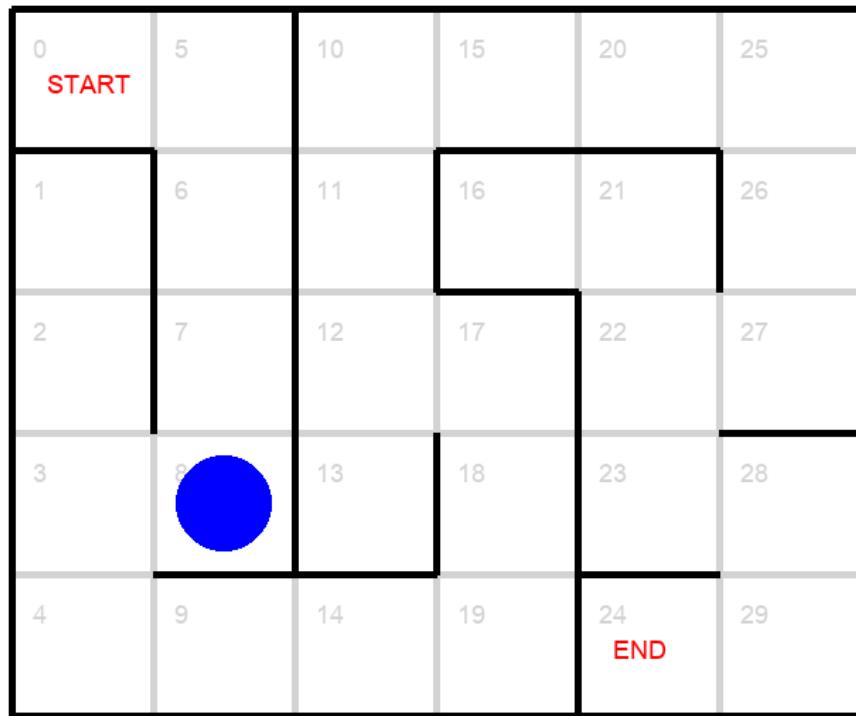
ontvangt wanneer de actor het doel bereikt heeft, zal de verminderingssfactor rond de waarde 1 geplaatst worden aangezien de taak eindig is en de finale staat waarbij de agent het einde van het doolhof bereikt heeft, het einddoel is.

### 2.3 Policy $\pi$

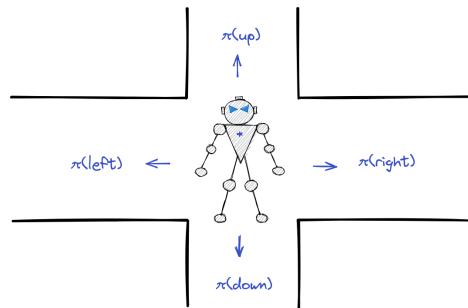
Op wat baseert de agent zich om een bepaalde actie te kiezen? Beloningen hebben invloed op acties van de agent in de toekomst, maar beslissen niet welke keuze er gemaakt wordt op het huidig tijdstip. De policy-functie karakteriseert hoe de agent reageert in de omgeving en welke acties het best genomen worden in een bepaalde staat.

De policy-functie is een functie die een gegeven toestand toewijst aan een probabilitätsdistributie met alle mogelijke acties. Als de agent de policy  $\pi$  volgt op tijdstip  $t$ , dan is  $\pi(a|s)$  de kans dat  $A_t = a$  indien  $S_t = s$ . Dit wil zeggen dat op tijdstip  $t$ , onder de policy van  $\pi$  de kans om actie  $a$  te kiezen in staat  $s$  gelijk is aan  $\pi(a|s)$ . In figuur 2.4 worden de mogelijke policy-waarden weergegeven van een robot. De kansdistributie zal verdeeld worden over de 4 mogelijke acties (links, rechts, omhoog en naar beneden). De distributie kan zowel continu als discreet zijn, afhankelijk van het probleem.

## 2 Hoofdconcepten Reinforcement Learning



Figuur 2.3: Voorbeeld van doolhofomgeving [3]



Figuur 2.4: Policy van robot [4]

### 2.4 Pad $\tau$

Een pad, ook wel een traject genoemd, is een opeenvolging van toestanden en acties. Op elk tijdstip wordt de staat geobserveerd, een actie genomen om dan terug een nieuwe staat te evalueren. Dit proces herhalen levert een pad op in de omgeving. Een pad is een volledig traject dat genomen kan worden in het sequentiële beslissingsprobleem. Dit wordt voorgesteld door

## 2 Hoofdconcepten Reinforcement Learning

middel van een lijst van opeenvolgende staten en gekozen acties:  $(s_0, a_0, s_1, a_1, \dots, s_T)$  met  $s_T$  de finale toestand.

### 2.5 Winstwaarde R

De winstwaarde  $R$ , ook wel opbrengst genoemd, is de accumulatieve beloning over een geheel pad. Ze wordt berekend aan de hand van de som van beloningen in het pad, gewogen met de verminderingssfactor  $\gamma$ :

$$R(\tau) = \sum_{t=0}^T r_t \gamma^t. \quad (2.1)$$

### 2.6 Staatwaarde V

Het doel is niet om de winstwaarde van 1 pad te ontdekken, maar de verwachte cumulatieve opbrengst van alle mogelijke paden afhankelijk van de policy-functie  $\pi$ . Er wordt begonnen bij een bepaalde startstaat  $s$  op tijdstip  $t$  en gekeken wat de verwachte waarde over alle paden is, behaald door een bepaalde policy. Deze verwachte waarde staat bekend als de staatwaarde  $V$ . De formule geldt als volgt:

$$V^\pi(s) = E_{\tau \sim \pi}[R(\tau) | s_0 = s]. \quad (2.2)$$

De uitdrukking  $\tau \sim \pi$  geeft weer dat het traject wordt bepaald door de actie-policy  $\pi$ . De staatwaarde geeft met andere woorden weer hoeveel de verwachte opbrengst bedraagt voor een agent wanneer hij zich in een bepaalde staat  $s$  bevindt en de policy  $\pi$  volgt.  $V^\pi(s)$  wordt ook wel de waardefunctie genoemd.

### 2.7 Staat-Actiewaarde Q

De staat-actiewaarde  $Q$  is gelijkaardig aan de staatwaarde  $V$ , maar geeft de verwachte waarde weer van een bepaalde startstaat en een specifieke actie. De opbrengst wordt bekeken van het volgen van de actie  $a$  in een bepaalde startstaat  $s$ . Na het uitvoeren van actiewaarde  $a$  wordt de policy van  $\pi$  gevolgd.

Gegeven een startstaat  $s$ , geeft de staatwaarde  $V$  de verwachte winstwaarde weer wanneer de eerste actie gebaseerd is op de policy  $(a \sim \pi(s))$ , terwijl de staat-actiewaarde  $Q$  de verwachte winstwaarde beschouwt wanneer de eerste actie gegeven is, zoals weergegeven in vergelijking 2.3:

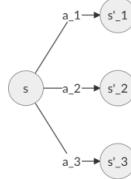
$$Q^\pi(s, a) = E_{\tau \sim \pi}[R(\tau) | s_0 = s, a_0 = a]. \quad (2.3)$$

Beschouw de situatie in figuur 2.5 waar de agent de keuze heeft tussen 3 acties,  $a_1, a_2, a_3$  om een overgang te maken van staat  $s$  naar  $s'$ . De staat-actiewaarde is gebonden aan 1 gekozen actie van de 3, de staatwaarde daar tegenover bepaalt de

## 2 Hoofdconcepten Reinforcement Learning

verwachte waarde van de 3 mogelijke staat-actiewaarden (alle acties die gesampled kunnen worden uit  $\pi$ ). De formule van de staatwaarde kan dus worden weergegeven als de verwachte waarden over 3 staat-actiewaarden:

$$V^\pi(s) = E_{\tau \sim \pi}[Q^\pi(s, a)]. \quad (2.4)$$



Figuur 2.5: Transitiegraaf met mogelijke acties van policy  $\pi$  [5]

De staat-actiewaarde  $Q$  is een essentiële bouwsteen in *Q-learning*- en DQN-algoritmen.  $Q^\pi(s, a)$  staat ook wel bekend als de actie-waardefunctie.

### 2.8 Doel van RL

De beloningen en winstwaarde motiveren de agent om bepaalde toestanden te bezoeken. Het doel van de actor is om bepaalde keuzes te maken die de hoogst mogelijke verwachte opbrengst opleveren. Deze keuzes worden weergegeven in de optimale actie-policy  $\pi^*$  en deze policy is wat het RL-algoritme probeert te achterhalen.

De optimale actie-policy  $\pi^*$  kan gevonden worden door de optimale staatwaardefunctie en de optimale staat-actiewaardefunctie zoals weergegeven in vergelijkingen 2.5 en 2.6

$$V^*(s) = \max_{\pi} E_{\tau \sim \pi}[R(\tau)|s_0 = s]. \quad (2.5)$$

$$Q^*(s, a) = \max_{\pi} E_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a]. \quad (2.6)$$

De operatie  $\max_{\pi}$  geeft weer dat de policy gekozen wordt die de optimale waarde oplevert. Een policy  $\pi$  presteert beter dan een policy  $\pi'$  als en slechts als de verwachte waarde van  $\pi$  groter is dan van  $\pi'$  over alle toestanden ( $\pi > \pi'$  als  $V^\pi(s) > V^{\pi'}(s), \forall s$ ) [13].

Merk op dat de formules voor de optimale staatwaarde en staat-actiewaarde onafhankelijk zijn van een bepaald policy. De optimale opbrengst bereiken vanuit een bepaalde staat (en actie) is het objectief, ongeacht de gebruikte policy.

Na het vinden van de optimale staat-actiewaarde  $Q^*$ , kan de optimale actie  $a^*$  in een bepaalde staat  $s$  achterhaald worden. De argmax operatie wordt gebruikt om de actie te selecteren die resulteert in de optimale staat-actiewaarde  $Q^*$  zoals weergegeven in vergelijking 2.7:

## 2 Hoofdconcepten Reinforcement Learning

$$a^*(s) = \operatorname{argmax}_a Q^*(s, a). \quad (2.7)$$

### 2.9 Bellman-vergelijking

Het bepalen van de waardefunctie/actie-waardefunctie kan gezien worden als het vinden van de wortelwaarde  $s$  van een boom met transities zoals in 2.5, maar dan vele malen groter en met deelbomen die de volledige toestandsruimte bedekken. In 1957 schetste Richard Bellman een oplossing voor dit probleem met de Bellman vergelijking (Engels: "Bellman equation").

Door de volledige toestandsruimte te overlopen en de waarde van een ouderknoop te berekenen aan de hand van de som van beloningswaarden van de kinderen, rekening houdend met de verminderingsfactor  $\gamma$  en aan de hand van bladknopen en de transitiefunctie  $T_a$  kan de waarde  $V(s)$  berekend worden.

Richard Bellman introduceerde de principes van dynamisch programmeren om dit probleem op te lossen. Het overlopen van de toestanden en acties zoals weergegeven in uitdrukkingen 2.8 en 2.9 verloopt op recursieve wijze.

Om de staatwaarde van een bepaalde staat  $s$  te berekenen moet de som genomen worden van de beloning van de huidige stap en de staatwaarde van de volgende staat  $s'$  met verminderingsfactor  $\gamma$ . De beloning van de huidige stap wordt opgewekt door de genomen actie bepaald door de policy, weergegeven met  $a \sim \pi(s)$ :

$$V^\pi(s) = E_{a \sim \pi(s), s' \sim T_a(s)}[r(s, a) + \gamma V^\pi(s')]. \quad (2.8)$$

De term  $s' \sim T_a(s)$  geeft weer dat de volgende toestand  $s'$  bepaald wordt door de transitiefunctie inherent aan de omgeving en op basis van huidige staat en actie.

$$Q^\pi(s, a) = r(s, a) + \gamma E_{s' \sim T_a(s), a' \sim \pi(s')}[Q^\pi(s', a')] \quad (2.9)$$

De letter  $a'$  is de volgende actie en wordt gekozen aan de hand van  $\pi(s')$ . Net zoals bij de staatwaarde wordt de staat-actiewaarde  $Q^\pi(s, a)$  bepaald door de som van de beloning van de huidige staat  $r(s, a)$  en de staat-actiewaarde van de volgende staat  $Q^\pi(s', a')$ . Aangezien de actie voor de huidige stap gegeven is door  $a$ , moet  $a$  niet gesampled worden op basis van policy  $\pi$ . Echter doordat zowel de staat en de actie van de volgende stap gebruikt worden, moet eerst de volgende staat bepaald worden aan de hand van de transitiefunctie ( $s' \sim T_a(s)$ ) om dan de volgende actie te kunnen samplen ( $a' \sim \pi(s')$ ).

De optimale Bellman-vergelijking voor de staatwaarde en staat-actiewaarde neemt respectievelijk de beste actie  $a$  en  $a'$  ( $\max a$ ) in plaats van te samplen uit  $\pi(s)$  en  $\pi(s')$ :

$$V^*(s) = \max_a E_{s' \sim T_a(s)}[r(s, a) + \gamma V^*(s')] \quad (2.10)$$

en

## 2 Hoofdconcepten Reinforcement Learning

$$Q^*(s, a) = r(s, a) + \gamma \max_{a'} E_{s' \sim T_a(s')} [Q^*(s', a')]. \quad (2.11)$$

Het probleem bij deze manier van werken is dat de recursieve betrekking vereist dat de transitiefunctie en de beloningsfunctie voor alle toestanden gekend zijn bij de agent. De transitiefunctie en beloningsfunctie samen worden ook wel het dynamisch model genoemd. Het dynamisch model is echter meestal niet gekend door de agent.

Dit leidt tot de nood aan modelvrije algoritmen die in staat zijn de waardefuncties en de policy af te leiden zonder het dynamisch model. Ondanks dit vormt de Bellman-vergelijking de basis voor algoritmen die de waardefuncties berekenen en voor vele andere methoden die RL-problemen proberen op te lossen.

### 2.10 Temporal Difference Learning

Temporal Difference (TD) Learning is een klasse van modelvrije RL-algoritmen. Deze term is geïntroduceerd door Sutton in 1988 [14]. Modelvrije algoritmen vervangen de transitiefunctie door een iteratieve sequentie van samples uit de omgeving. De naam "Temporal Difference" komt van het feit dat het verschil in waarde tussen twee tijdstippen gebruikt wordt om de waarde te berekenen op een nieuw tijdstip.

TD is een vorm van *bootstrapping* waarbij verwerkte samples gebruikt worden om de finale waarde te benaderen. Bootstrapping is een proces van opeenvolgende verfijning waarbij oude schattingen van een waarde worden verfijnd met nieuwe updates. Deze methoden nemen monsters uit de omgeving, zoals Monte Carlo-methoden en voeren updates uit op basis van huidige schattingen, zoals dynamische programmeermethodes. Vergelijking 2.12 geeft het principe weer:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s)). \quad (2.12)$$

De term  $V^\pi(s)$  wordt de *TD Estimate* genoemd. Dit is de geschatte waardefunctie van de huidige staat.

De uitdrukking  $r + \gamma V^\pi(s')$  staat bekend als de *TD Target*. Dit is de doelwaarde van de waardefunctie waar de TD Estimate naar geoptimaliseerd wordt. Door de additionele informatie  $r$  toe te voegen in de TD Target is deze waarde nauwkeuriger dan de TD Estimate. vergeleken met supervised learning zou TD Target beschouwd kunnen worden als het label en de TD Estimate als de predictie.

De term  $\alpha$  geeft de learning rate weer. De parameter  $\gamma$  slaat op de verminderingsfactor zoals eerder vermeld in sectie 2.2.

TD learning kan ook worden toegepast op de actie-waardefunctie. De formule ziet er als volgt uit:

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \alpha(r + \gamma Q^\pi(s', a') - Q^\pi(s, a)). \quad (2.13)$$

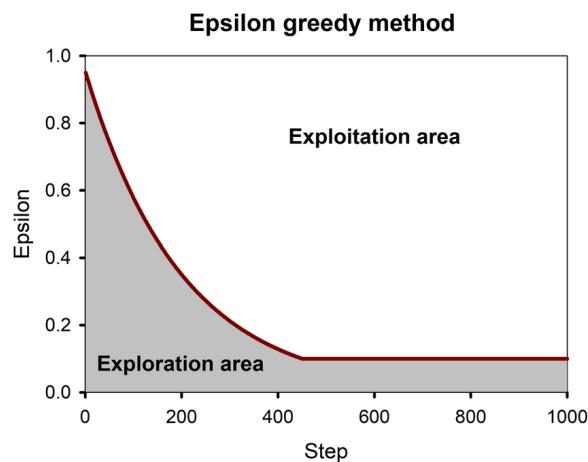
### 2.11 Exploratie

De policy  $\pi$  zorgt ervoor dat in elke toestand de actie gekozen wordt die de hoogste staatwaarde oplevert. Indien op elk moment de optimale actie gekozen wordt van de policy, wordt er een *greedy approach* gevolgd. Deze aanpak is echter kortzichtig en kan ervoor zorgen dat de agent vast komt te zitten in een lokaal maximum.

De agent moet de kans krijgen om andere acties uit te proberen ondanks de lagere waarde op korte termijn van deze acties. Er moet een afweging gemaakt worden tussen het nemen van acties met de hoogste waarde (exploitatie) en het kiezen van willekeurige acties om nieuwe acties te vinden die potentieel nog beter scoren (exploratie).

De *tradeoff*tussen deze twee termen is zeer belangrijk. Indien de agent zich alleen zou concentreren op exploitatie zoals bij een greedy approach, kan het geen nieuwe oplossingen ontdekken en leert de agent niet meer. Aan de andere kant indien de actor enkel willekeurige acties neemt, is het trainen nutteloos en wordt er geen gebruik gemaakt van de ervaring van de actor.

Een van de meest gebruikte oplossingen voor deze afweging is het  $\epsilon$ -greedy-algoritme. De waarde van epsilon is een *hyper-parameter*. Wanneer de agent een actie moet nemen heeft hij een kans van  $\epsilon$  om een willekeurige actie te kiezen en een kans van  $1 - \epsilon$  om te kiezen voor de optimale actie afhankelijk van de policy. Doorheen de tijd wordt de waarde van  $\epsilon$  verminderd om meer gebruik te maken van de ervaring van de agent. Dit wordt meestal gedaan aan de hand van een *exponential decay* zoals weergegeven in figuur 2.6.



Figuur 2.6: Exponential decay van  $\epsilon$  [6]

### 2.12 Off-Policy learning

Off-Policy learning is een belangrijk concept naast de afweging van exploratie en exploitatie. Normaal gezien wordt de policy strikt aangepast met resultaten van recente acties. De waarde van de geselecteerde actie wordt opgeslagen in dezelfde policy die gebruikt werd om de actie te selecteren. Deze vorm van leren staat bekend als on-policy learning.

Off-Policy learning is een alternatief dat waarden kan opslaan van een andere actie en niet de actie recent gekozen door de

## 2 Hoofdconcepten Reinforcement Learning

policy die instaat voor het gedrag van de agent in de omgeving. Het voordeel hiervan is dat wanneer de policy exploreert en een niet optimale actie kiest, deze ondermaatse waarde niet zal worden opgeslagen.

De architectuur van Off-Policy learning bestaat uit twee policies. De eerste policy staat bekend als de gedrags-policy en staat in voor het selecteren van acties, inclusief exploratie. De tweede policy, namelijk de doel-policy, wordt geüpdatet door de waarden op te slaan.

On-Policy learning zal de optimale policy benaderen en convergentie is meestal iets stabiel (lage variantie). Off-policy learning is in staat om de optimale policy te vinden (lage bias), maar kan onstabiel zijn door de max- operatie in combinatie met functie benadering. Dit wordt verder toegelicht in 3.9.

De meer stabiele convergentie zorgt voor conservatiever gedrag. In de praktijk kan dit een groot verschil maken als fouten kostbaar zijn. Bijvoorbeeld bij het trainen van een robot in het echte leven en niet in simulatie. Een meer conservatief leeralgoritme dat hoge risico's vermijdt is interessant als er geld op het spel staat.

### 2.13 Q-learning

Q-learning is een Off-policy algoritme gevormd door combinatie van TD learning en de Bellman-vergelijking. Q-learning verbetert de schatting van de huidige staat-actiewaarde bij elke aanpassingsstap. Dit wordt gedaan aan de hand van de beloningsinformatie en optimale schatting van de volgende staat-actiewaarde:

$$Q^*(s, a) \leftarrow Q^*(s, a) + \alpha(r + \gamma \max_{a'} Q^*(s', a') - Q^*(s, a)). \quad (2.14)$$

De term  $r + \gamma \max_{a'} Q^*(s', a')$  komt overeen met de TD target zoals besproken in 2.10.

De TD target geeft een betere schatting van de huidige staat-actiewaarde, TD estimate  $Q^*(s, a)$  wordt aangepast naar de waarde van TD target. Dit kan voorgesteld worden als het minimaliseren van het verschil tussen TD estimate  $Q^*(s, a)$  en de TD target  $r + \gamma \max_{a'} Q^*(s', a')$ .

De Q-functie wordt vaak voorgesteld aan de hand van een neurale netwerk. De loss-functie om dit netwerk te trainen is dan:

$$L(\theta) = (r + \gamma \max_{a'} Q_\theta^*(s', a') - Q_\theta^*(s, a))^2. \quad (2.15)$$

Deze uitdrukking stelt de loss-functie voor. Het doel is om deze functie te minimaliseren rekening houdend met de grootte van  $\theta$ . De term  $\theta$  stelt de parameters van het neurale netwerk voor. Meer hierover volgt in sectie 3.2 in volgend hoofdstuk.

# 3

## Deep Reinforcement Learning

De traditionele technieken zoals besproken in vorig hoofdstuk worden meestal gebruikt voor relatief kleine problemen zoals puzzels, doolhoven en andere problemen waarvan de toestandsruimte klein genoeg is om in het geheugen van de computer te passen. De toestandsruimte van hoog-dimensionele problemen zoals het besturen van een auto of een robot bestaat echter uit een veel groter domein waar de traditionele RL-algoritmen niet mee omkunnen. In de afdeling van supervised learning zorgde de ImageNet-classificatiewedstrijd elk jaar voor lichte verbetering in nauwkeurigheid. ImageNet is een database met afbeeldingen die georganiseerd is volgens de WordNet-hiërarchie, waarin elk knooppunt van de hiërarchie wordt weergegeven door honderden en duizenden afbeeldingen. Het project heeft een belangrijke rol gespeeld bij het bevorderen van computervisie en onderzoek naar deep learning [15].

In het jaar 2012 veranderde dit volledig. De doorbraak van AlexNet gedreven door deep neural networks behaalde een verbetering in top-5 error van 10%. De top-5 error rate is het percentage van aantal keren dat de classificator er niet in slaagde de juiste klasse op te nemen in zijn top vijf predicties. Vanaf dan overtroffen DNN's alle andere aanpakken op taken rond computervisie. Het jaar 2012 wordt aanzien als het doorbreekjaar van deep learning. Deze doorbraak in het gebied van supervised learning bracht allerlei vragen met zich mee in de sector van RL. Verschillende onderzoekers vroegen zich af of het mogelijk was om de concepten van deep learning te combineren met de traditionele RL-technieken en gelijkaardige verbeteringsresultaten te behalen als bij supervised learning.

In 2013 slaagden Mnih et al. er in om de combinatie tot een goed einde te brengen. In de paper "Playing Atari with Deep Reinforcement Learning"[16] presenteerde Mnih het eerste deep learning-model waarbij een RL-agent rechtstreeks leert van hoog-dimensionale sensorische input. Het algoritme dat gebruikt werd bestaat uit een combinatie van Q-learning en deep learning en kreeg de naam "Deep Q-Network"(DQN). Het werd toegepast op 7 Atari 2600 videospellen. Op 3 van de 7 overtrof het menselijke experts. Echte oog-handcoördinatie van deze complexiteit was nog niet eerder door een computer bereikt. Het leren van pixel tot joystick-acties vertoonde voor het eerst gedrag dat dicht in de buurt kwam van hoe mensen videospellen spelen.

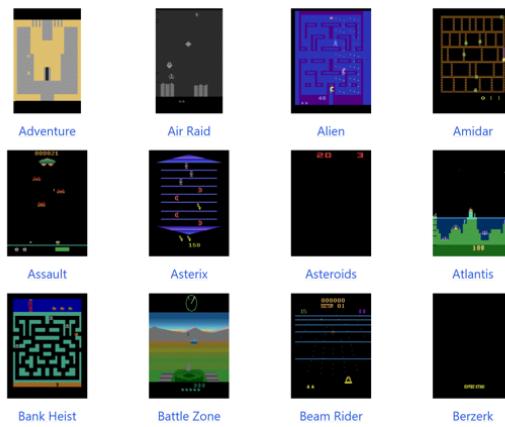
In dit hoofdstuk zal de werking van DQN's geschatst worden, alsook de problemen die ze met zich meebrengen en de mogelijke afwegingen. De architectuur wordt uitgewerkt op verschillende problemen aan de hand van het Pytorch framework

### 3 Deep Reinforcement Learning

en de Gym library waarover meer in sectie 3.6

#### 3.1 Atari Arcade Games

Als overgangsstap van eenvoudige doolhoven en puzzels naar complexe hedendaagse problemen zoals zelfrijdende auto's en automatische robots werd de Arcade Learning Environment (ALE) opgestart. Het is gebaseerd op een simulator van Atari 2600 video games uit de jaren 80. Het is gemaakt om verschillende uitdagende taken voor RL-onderzoek te bieden. ALE levert de agents een hoog dimensionele visuele input ( $260 \times 160$  RGB video's aan een frequentie van 60Hz). De toestandsruimte van 1 enkel beeld van pixels met 256 RGB kleurwaarden bedraagt dan  $256^{33600}$  ( $256^{260 \times 160}$ ). Voorbeelden van omgevingen uit ALE worden weergegeven in figuur 3.1.



Figuur 3.1: Een verzameling aan Atari 2600-omgevingen gesimuleerd via de Arcade Learning Environment. [7]

#### 3.2 Architectuur van een Deep Q-network (DQN)

De combinatie van Q-learning met een DNN wordt deep Q-learning genoemd. Een DNN dat een Q-functie benadert krijgt de naam "deep Q-network"(DQN) [13]. In deep Q-learning wordt met andere woorden een neurale netwerk gebruikt om de Q-waardefunctie te benaderen. Zoals te zien in figuur 3.2 bestaat de input voor het neurale netwerk uit de staat en de output uit de Q-waarden van alle mogelijke acties.

Bij de traditionele Q-learning algoritmen werd de beste actie bepaald aan de hand van een Q-tabel die staat-actieparen afbeeldt op Q-waarden. DQN's gebruiken een max-operatie op alle outputs van het neurale netwerk om te bepalen welke actie genomen moet worden.

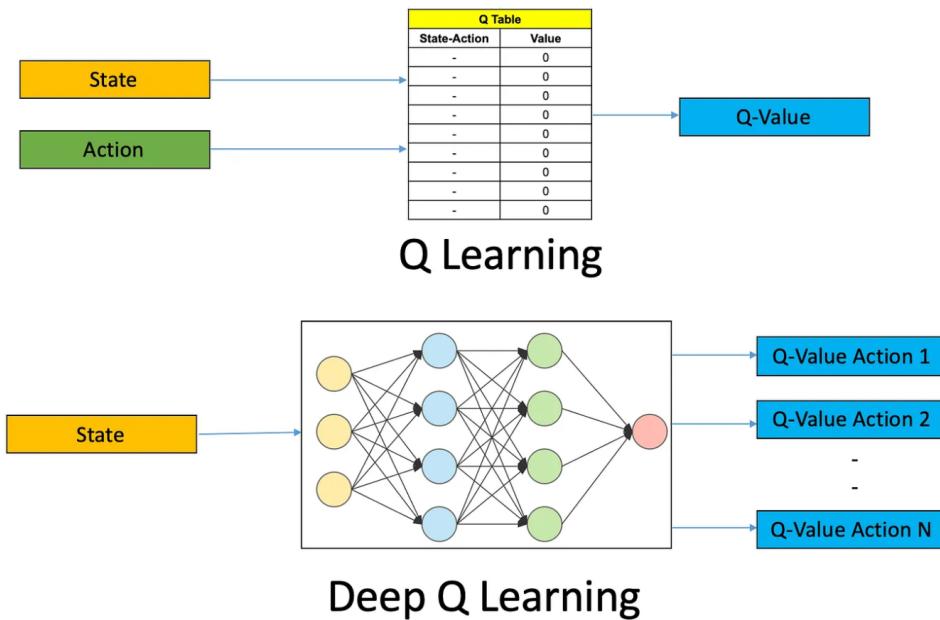
De loss-functie van het neurale netwerk bestaat uit de *mean squared error* (MSE) van de voorspelde Q-waarde en de Q-waarde van het doel afgetrokken met  $Q^*$  ( 2.15) zoals reeds besproken in de sectie rond Q-learning in het vorige hoofdstuk 2.13. De formule wordt hier ter herhaling nog eens weergegeven met verduidelijking van de parameter  $\theta$  in het tijdsdomein:

$$L(\theta) = (r + \gamma \max_{a'} Q_{\theta_{t-1}}^*(s', a') - Q_{\theta_t}^*(s, a))^2. \quad (3.1)$$

### 3 Deep Reinforcement Learning

Het doel van het netwerk is om de loss-functie te minimaliseren. Nadat de *loss-waarde* berekend is in een iteratiestap, worden de gewichten van het netwerk aangepast via Stochastic Gradient Descent (SGD) en *backpropagation* zoals bij een deep supervised learning-architectuur.

De doelwaarde hangt steeds af van de parameters van het netwerk in een vorige iteratiestap ( $\theta_{t-1}$ ). De doelwaarden veranderen doorheen het algoritme, dit in tegenstelling tot deep supervised learning-algoritmen waar de labels vaststaan en gekend zijn voordat het algoritme start. Aangezien zowel de voorspelling als de doelwaarde afhankelijk zijn van het optimalisatieproces van de parameters  $\theta$ , is er een risico dat het doel voorbijgeschoten zal worden en het proces zeer onstabiel wordt. Dit convergentieprobleem vormt de eerste uitdaging bij de architectuur van DQN's.



Figuur 3.2: Verschil Q-learning en Deep Q-learning [8]

## 3 Deep Reinforcement Learning

### 3.3 Correlatie

Tijdens het verloop van het trainingsproces van een DQN wekt de agent een sequentie aan acties en nieuwe toestanden op door te interageren met de omgeving. Naburige toestanden liggen slechts 1 actie van elkaar verwijderd wat ervoor zorgt dat Q-waarden van opeenvolgende *samples* gecorreleerd zijn. Dit vormt het tweede probleem in de besproken architectuur. De correlatie tussen opeenvolgende waarden kan voor convergentieproblemen zorgen met verhoogde kans om vast te komen zitten in lokale minima en het optreden van feedback loops. Een klassiek voorbeeld hiervan is een programma voor een schaakspel waarbij de tegenstander steeds begint met dezelfde opening.

### 3.4 *Experience Replay* en *infrequent weight updates*

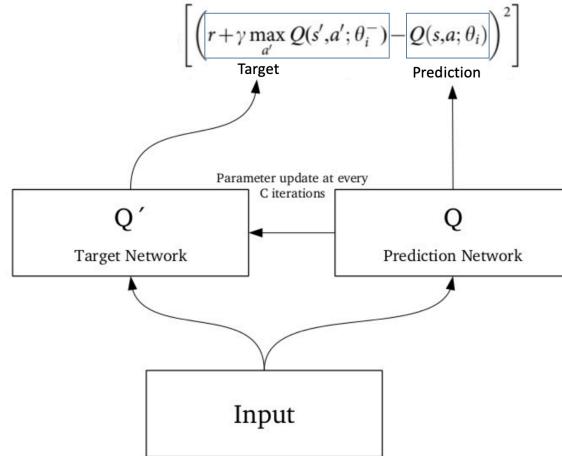
Experience Replay en infrequent weight updates zijn twee technieken geïntroduceerd door Mnih et al. om respectievelijk het convergentie- en correlatieprobleem op te lossen. Het gebruik van experience replay in DQN's breekt de correlaties tussen staten en levert een meer uiteenlopende verzameling aan samples. Experience replay maakt gebruik van een geheugen-buffer. Dit is een buffer van voorgaand verkende toestanden. Het algoritme zal willekeurige trainingsinstanties samplen uit deze opslagplaats met een uniforme verdeling. De laatste N samples worden opgeslagen in de buffer, waarbij N een hyperparameter (typische waarde  $10^6$ ) is.

Stel dat een robot voor een videogame gebouwd wordt, waarbij elk frame van de game een andere staat vertegenwoordigt. Tijdens de training kan een willekeurige batch van 64 frames uit de laatste 100.000 frames genomen worden om het netwerk te trainen. Dit levert een subset op waarbinnen de correlatie tussen de samples laag is en ook zal zorgen voor een betere sampling-efficiëntie. Het DQN wordt nu getraind op een meer diverse verzameling en niet meer op een verzameling van recente staten. De volgende staat waarmee het netwerk getraind wordt, is nu niet meer de directe opvolger, maar een toestand uit de buffer van verkende toestanden. Deze techniek is zeer gelijkaardig aan de methode om *mode collapse* in het trainingsproces van een *Generative Adversarial Network* (GAN) op te lossen.

De tweede techniek, infrequent weight updates, zorgt voor vermindering in divergentie die wordt veroorzaakt door de frequente updates van de gewichten van de doelwaarde. De intentie is om stabiliteit in de loss-functie te brengen en de convergentie van het netwerk te verbeteren. Het netwerk  $Q$  wordt gekloond naar het doelnetwerk  $\hat{Q}$  om de C updates. Het doelnetwerk  $\hat{Q}$  wordt gebruikt om het doel in te stellen voor de komende C aanpassingen aan het netwerk  $Q$ . De hyperparameter C stelt het aantal updates voor tot wanneer het netwerk  $Q$  gekloond wordt.

De gewichten van het doelnetwerk veranderen aan een veel lagere frequentie dan de gewichten van het netwerk dat instaat voor het gedrag (zie 3.3). Het opstellen van Q-waardedoelen aan de hand van een verzameling van oudere parameters zorgt voor een *delay* tussen het moment dat een update aan  $Q_\theta$  aangebracht is en het moment dat deze update de doelen aanpast. Op deze manier verbetert de stabiliteit van het Q-learning-proces [9].

### 3 Deep Reinforcement Learning



Figuur 3.3: Schets van de infrequent weight updates van een DQN [9]

### 3.5 Overzicht van gehele architectuur

Alvorens over te gaan naar de implementatie wordt nog eens kort de totale structuur weergegeven en het verloop van het algoritme.

- 1) Allereerst vindt er een *preprocess*-stap plaats op het gamescherm, om daarna een gedeeltelijke versie van dit scherm als toestand aan het DQN te bieden.
- 2) Daarna wordt een actie gekozen aan de hand van het  $\epsilon$ -greedy-algoritme zoals uitgelegd in sectie 2.11. Met de kans van  $\epsilon$  wordt er een willekeurige actie gekozen en met de kans  $\epsilon - 1$  de actie met maximale Q-waarde.
- 3) Na het uitvoeren van de gekozen actie in de huidige staat gaat de toestand over naar een volgende staat en ontvangt de agent een beloning. De transitie bestaande uit een *4-tuple* van de beloning, de originele staat, de genomen actie en de volgende staat en wordt weggeschreven in de geheugenbuffer.
- 4) Vervolgens wordt een willekeurige batch van transities *gesampled* en de loss-functie berekend. Dit is de MSE van de Q-doolwaarde en de voorspelde Q-waarde:

$$L(\theta) = (r + \gamma \max_{a'} Q_{\theta_{t-1}}^*(s', a') - Q_{\theta_t}^*(s, a))^2. \quad (3.2)$$

- 5) Aansluitend hierop wordt een gradient descent-stap uitgevoerd op het predicitie-netwerk om de loss-waarde te minimiseren.
- 6) Tenslotte wordt na  $C$  iteraties het netwerk  $Q$  gecloned naar het doelnetwerk  $\hat{Q}$ .

### 3 Deep Reinforcement Learning

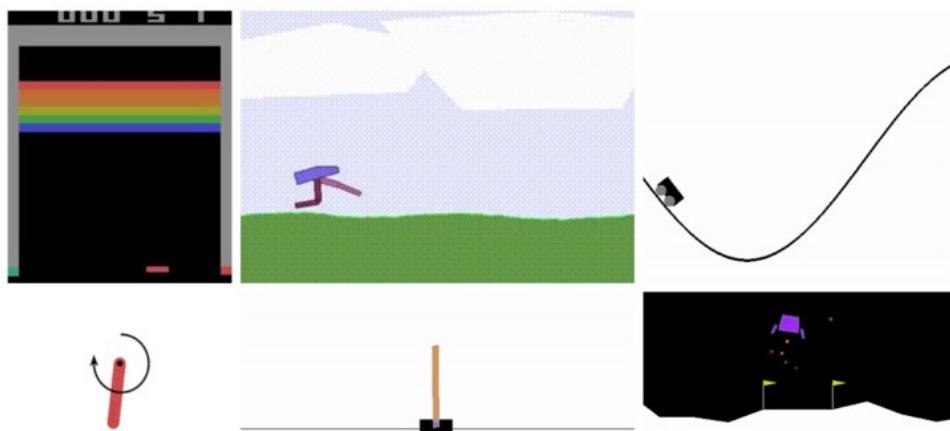
Deze stappen worden  $E$  keer herhaald.  $E$  is de hyperparameter die het aantal episodes weergeeft.

#### 3.6 Gym library

De Gym library van The Farama Foundation levert een *toolkit* aan verschillende gesimuleerde omgevingen. Het bestaat uit een API in Python met verschillende methodes zoals herstarten van een omgeving, een stap uitvoeren en de omgeving renderen. Dit zijn kenmerkende methoden in RL[7].

Gym biedt een eenvoudig platform aan voor het testen en ontwikkelen van agents in een divers aanbod van RL-omgevingen. Voorbeelden van omgevingen zijn de klassieke Atari Arcade Games, klassieke controleproblemen uit de mechanica zoals "Cart Pole" en de zogenaamde "toy games" zoals "Lunar Lander" 3.4 .

De library laat ook toe om zelf omgevingen aan te maken en het gebruik van omgevingen van derde partijen.



Figuur 3.4: Voorbeelden van verschillende omgevingen beschikbaar in de Gym library [7]

#### 3.7 Pytorch

Voor de implementatie van de DQN-architectuur is er gekozen voor Pytorch. Het is een *open-source framework* voor machinaal leren dat origineel ontwikkeld is bij Meta Ai. Het vormt nu een deel van de Linux Foundation Umbrella [17].

Pytorch is een high-level machine learning framework dat eenvoudig toegang biedt tot parallelle verwerking van data met behulp van GPU's en een interface voorziet voor de aanmaak van DNN's.

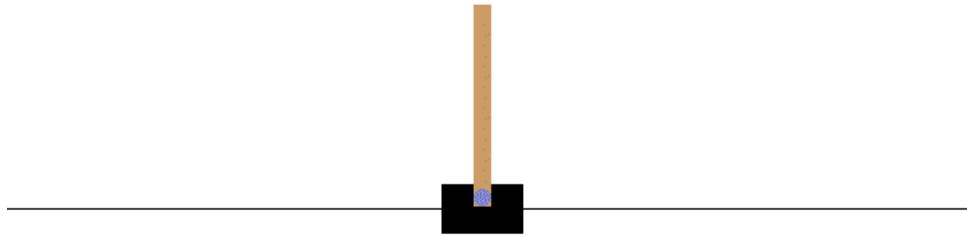
#### 3.8 DQN toegepast op de Cart Pole-omgeving

De volgende stap was om zelf een DQN op te bouwen aan de hand van de gym library en Pytorch. De omgeving van Cart Pole leek een ideaal punt om van te starten en werd gekozen om de theoretische concepten op toe te passen.

### 3 Deep Reinforcement Learning

Het doel van het spel is om een paal zo lang mogelijk te balanceren op een kar. De kar kan zich verplaatsen naar links of naar rechts, maar de hoek tussen de y-as en de paal mag niet groter zijn dan 12 graden en de kar mag niet uit het scherm verdwijnen. Een voorbeeld van de Cart pole-omgeving is te zien in figuur 3.5.

Aangezien het de bedoeling is dat de stok zolang mogelijk gebalanceerd blijft, krijgt de agent een beloning van +1 bij elke stap die hij neemt. Wanneer de hoek tussen de y-as groter is dan 12 graden, de kar van het beeld verschenen is of er 500 stappen genomen zijn, wordt het spel beëindigd.



Figuur 3.5: Cart Pole-omgeving uit de gym library [7]

In algoritme 25 is de pseudocode van het trainingsproces van het DQN te zien. De code bouwt verder op een RL-tutorial te vinden op de officiële pagina van Pytorch.

#### 3.8.1 Replay Buffer

Het geheugen van de agent bestaat uit een geheugenbuffer. De geheugenbuffer bevat een *double ended queue (dequeue)* en houdt transitie-tuples bij bestaande uit de huidige staat, gekozen actie, volgende staat en beloning. Wanneer aan een volle dequeue wordt toegevoegd, wordt het eerste item in de lijst verwijderd en het nieuwe item aan het einde van de lijst toegevoegd. Dit zorgt ervoor dat enkel de meest recente ervaringen worden opgeslagen. Bij elke optimalisatiestap zal het algoritme uit de opslagplaats een willekeurige batch samplen waarvan de grootte afhankelijk is van de hyperparameter "batchsize\_replaybuffer".

#### 3.8.2 Toestand

Zoals te zien op lijn 10 van het algoritme, bestaat de staat uit het verschil van twee frames van de omgeving. Dit is noodzakelijk om de snelheid van beweging en rotatie bij te houden. Een voorbeeld van een geëxtraheerd beeld dat aan het DQN

### 3 Deep Reinforcement Learning

gevoerd wordt, is te zien in figuur 3.6. Het beeld is 40 pixels hoog, 90 breed en bevat 3 kleurkanalen.

#### 3.8.3 Lagenstructuur van CNN

In tabel 3.1 is een overzicht van de gebruikte convolutionele lagen in het DQN te zien. Het overzicht is opgesteld met de standaard-batch size van 128. De architectuur bevat 7 lagen. De input voor het netwerk heeft de vorm [128, 3, 40, 90].

In de eerste 6 lagen worden 3 convolutie-operaties uitgevoerd. De operaties worden uitgevoerd met een *kernel size* van 5 en een *stride* van 2 waarbij de operaties telkens gevolgd worden door een batchnormalisatielaag. De laatste lineaire laag geeft 2 outputs weer voor elke staat. Dit zijn de staat-actiewaarden Q(staat, links) en Q(staat, rechts). Het doel van dit netwerk is om de verwachte waarden te voorspellen voor het verplaatsen van de kar naar links en naar rechts.

In totaal bevat het netwerk 40 866 parameters. Indien verondersteld wordt dat er met *float32*-datatype gewerkt wordt zoals bij grafische kaarten met cuda, neemt elke parameter 4 bytes in beslag. De totale grootte die de parameters in beslag nemen is gelijk aan 0.16 MB ( $40\ 866 \text{ parameters} * 4 \text{ (bytes/parameter)} / (2^{20} \text{ bytes / MB})$ ).

Om te kijken hoeveel geheugen het model in totaal in beslag neemt, moet er ook rekening gehouden worden met het geheugen dat de inputwaarde in beslag neemt. De inputwaarde bestaat uit een batch van 128 frames met afmetingen [3, 40, 90]. De inputwaarde heeft dus  $128 * 3 * 40 * 90 * 4 \text{ (bytes/parameter)} / (2^{20} \text{ bytes / MB}) = 5.27 \text{ MB}$  nodig.

Ten slotte heeft de voorwaartse / achterwaartse stap een bepaalde hoeveelheid geheugen nodig. De totale geheugenruimte voor de output tensoren wordt gevonden door voor elk laagtype de getallen van de output-vorm uit 3.1 met elkaar te vermenigvuldigen en daarna de producten van al deze lagen op te tellen. Deze som bedraagt 4 448 512. Dit wordt vermenigvuldigd met het aantal MB / parameter zoals bij de voorgaande berekeningen.

Tijdens de achterwaartse stap wordt de gradiënt van elke output tensor berekend. Het totaal aan geheugen voor de gradiënt bedraagt hetzelfde als die voor de output-vorm. Het geheugen nodig voor de voorwaartse / achterwaartse stap bedraagt:

$$2 * 4\ 448\ 512 * 4 \text{ (bytes/parameter)} / (2^{20} \text{ bytes / MB}) = 33.94 \text{ MB}$$

Het totaal aan geheugen nodig om het model te kunnen gebruiken bedraagt:  $5.27 \text{ MB} + 33.94 \text{ MB} + 0.16 \text{ MB} = 39.37 \text{ MB}$ .

#### 3.8.4 Actiekeuze

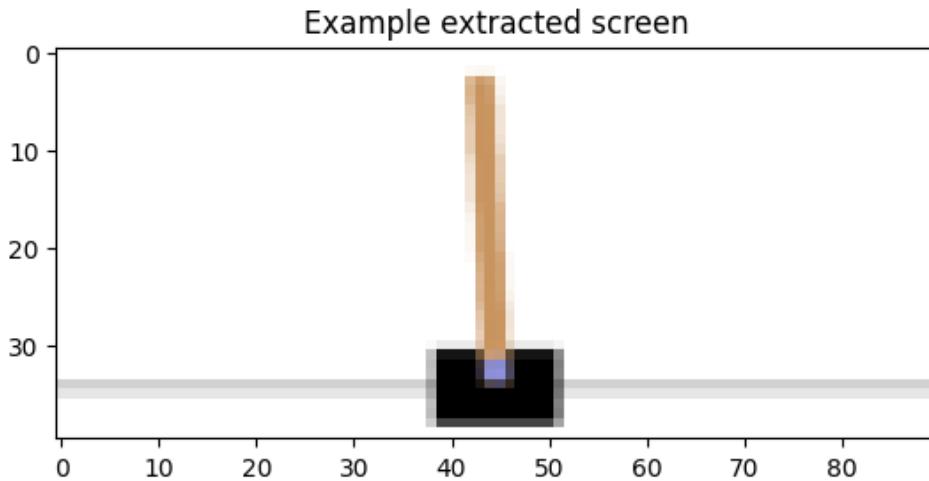
Op lijn 12 in het algoritme 25 is te zien dat het policy-netwerk een actie kiest. Wanneer een actie nodig is, wordt er eerst een willekeurig reëel getal gesampled tussen 0 en 1. Er wordt dan gekeken of het gekozen getal hoger is dan de  $\epsilon - threshold$ . Deze threshold wordt bepaald aan de hand van volgende formule:

$$threshold = startEpsilon + (startEpsilon - endEpsilon) * e^{(-1 \cdot aantalStappen / decay)} \quad (3.3)$$

De waarden startEpsilon en endEpsilon geven respectievelijk de startwaarde en eindwaarde weer in het  $\epsilon$ -greedy algoritme. Samen met de decay, die de exponentiële afzwakking weergeeft, zijn het drie hyperparameters die ingesteld kunnen worden. De variabele "aantalStappen" geeft het aantal stappen weer dat de agent reeds genomen heeft.

### 3 Deep Reinforcement Learning

Indien het willekeurig getal de threshold overschrijdt, wordt het toestandsbeeld doorgegeven aan het neuraal netwerk zonder het optimaliseren van de gewichten van het network. Het policy-netwerk geeft de verschillende Q-waarden terug en aan de hand van de max-operatie wordt de grootste waarde gekozen. Wanneer het monster de threshold niet overschrijdt, kiest de agent een willekeurige actie uit het actiedomein.



Figuur 3.6: Voorbeeld van een geëxtraheerd beeld uit de "Cart Pole"omgeving

Laagtype	outputvorm	aantal parameters
Conv2d	[128, 16, 18, 43]	1,216
BatchNorm2d	[128, 16, 18, 43]	32
Conv2d	[128, 32, 7, 20]	12,832
BatchNorm2d	[128, 32, 7, 20]	64
Conv2d	[128, 32, 2, 8]	25,632
BatchNorm2d	[128, 32, 2, 8]	64
Linear	[128, 2]	1,026

Tabel 3.1: Overzicht van convolutionele lagen in DQN.

#### 3.8.5 Optimalisatie van parameters van het CNN

Op het einde van een iteratie in de while-lus wordt het policy-netwerk geoptimaliseerd (te zien op lijn 19 in het algoritme). Eerst wordt er gekeken of de geheugenbuffer van de agent reeds genoeg transities bevat om 1 batch aan transitie-tuples op te halen. Indien dit niet het geval is, slaat het algoritme de optimalisatie over. In het andere geval haalt het algoritme 1 batch aan willekeurige transities op uit de buffer. Na het ophalen van een batch aan tuples bestaande uit (*huidigeStaat*, *actie*, *volgendeStaat*, *beloning*) worden de verwachte Q-waarden berekend aan de hand van formule 2.14 en het doelnetwerk voor stabilisatie. Daarna worden de gradiënten bepaald aan de hand van de loss-functie 2.15 en worden de parameters van policy-netwerk gewijzigd rekening houdend met de optimizer. Ten slotte worden na een bepaald aantal episodes de parameters van het policy-netwerk geladen in het doelnetwerk.

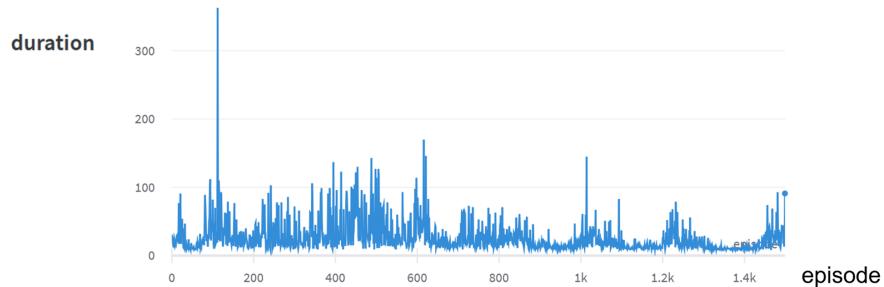
### 3 Deep Reinforcement Learning

#### 3.8.6 Resultaten

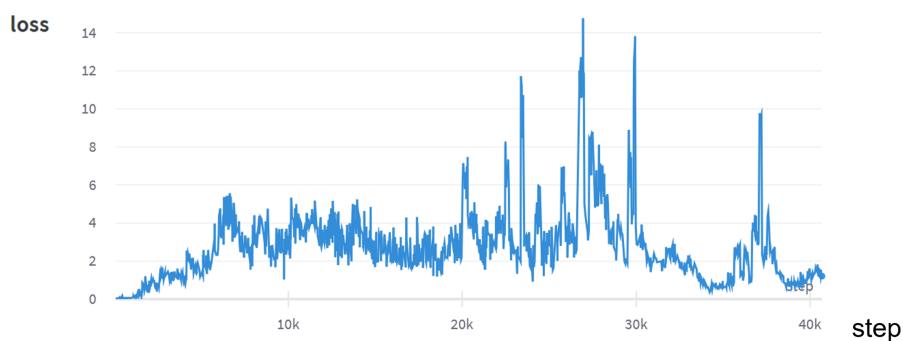
Na het bestuderen van de nodige theorie en het opstellen van het netwerk, was het tijd om de agent te gaan trainen. Een belangrijk deel in dit trainingsproces is de keuze van hyperparameters. RL-algoritmen bevatten meestal meer hyperparameters dan een supervised learning-algoritme. De hyperparameters spelen een zeer cruciale rol bij het trainen van een RL-agent en kunnen voor een totaal andere uitkomst zorgen.

De hyperparameters voorgesteld door PyTorch leverden matige resultaten op. Het verfijnen van de hyperparameters en andere lossfuncties en optimizers uitproberen, resulteerde in een lichte verbetering, maar presteerde nog steeds ondermaats. De agent slaagde er meestal in om in een bepaalde poging in één van de eerste 100 episodes zoals te zien in 3.7, een hoge score te halen. Het doel om consistent 500 stappen recht te blijven werd nooit behaald en bij het testen na het trainingsproces, krijgt de agent de stok meestal slechts enkele seconden hoog.

In de trainingscurve op figuur 3.8 is ook te zien dat het algoritme niet tot convergentie leidt en dat de loss-term zeer onstabiel blijft.



Figuur 3.7: Duration plot met op de y-as de duur dat Cart Pole zich kan rechthouden, uitgedrukt in actiestappen en op de x-as het aantal trainingsepisodes.



Figuur 3.8: Trainingscurve met op de y-as de MSE-loss en op de x-as het aantal tijdsstappen verstrekken in het algoritme.

### 3 Deep Reinforcement Learning

#### 3.9 Double Deep Q Network (DDQN) en *frame skipping*

Na verder onderzoek werd het duidelijk dat 2 cruciale problemen ervoor zorgden dat het trainingsproces niet tot een goed einde verliep. Ten eerste is de snelheid en rotatie van de paal moeilijk te beheren op basis van afbeeldingen. De eenvoudige toestandsrepresentatie als het verschil tussen de huidige en vorige frame is niet voldoende en er is een stapel van 4 of meer afbeeldingen van opeenvolgende toestanden nodig, samen met een techniek van frame skipping zoals voorgesteld door Mnih et al. [16].

Het tweede probleem is dat het gebruik van de max-operatie gevoelig is aan overschatting. Dit is een probleem van Q-learning in het algemeen en niet specifiek voor DQN.

Beschouw de volgende analogie om het probleem te verduidelijken. Stel dat er een test opgenomen wordt met 100 mensen om te kijken wie de grootste persoon is en hoeveel de lengte bedraagt. De gekozen testpersonen zijn allemaal 1,75m groot, maar dit is niet geweten. Het gebruikte meettoestel bevat een standaardfout van +/- 1 meter. Er zal dus ruis zitten op de metingen. De Variabele  $X = \{x^1, x^2, \dots, x^{100}\}$  bevat de opgemeten lengten. Wanneer de maximale lengte  $Y$  bepaald wordt aan de hand van de max-operatie:

$$Y = \max_i X^i \quad (3.4)$$

zal de gevonden waarde voor  $Y$  bijna altijd groter zijn dan 1,75m. Een manier om dit probleem op te lossen is door elke persoon twee keer te meten.  $X_1^i$  geeft de eerste meting van de  $i^{de}$  persoon weer en  $X_2^i$  de tweede. Door nu index  $i$  te zoeken op basis van de eerste meting:

$$n = \operatorname{argmax}_i (X_1^i) \quad (3.5)$$

en de effectieve maximale waarde dan te halen uit de tweede meting:

$$Y = X_2^n \quad (3.6)$$

zal het verkregen maximum minder afhankelijk zijn van ruis en is de kans op overschatting veel kleiner.

Dit principe werd door Van Hasselt et al. gebruikt in de paper "Deep Reinforcement Learning with Double Q-learning" [18]. Van Hasselt et al. gebruiken 2 onafhankelijke DQN's  $Q_{\phi 1}$  en  $Q_{\phi 2}$ . De TD target van de Q-learning formule (2.14) is nu niet meer,  $r + \gamma \max_{a'} Q^*(s', a')$ , maar gelijk aan  $r + \gamma Q_{\phi 2}(s', \operatorname{argmax}_{a'} Q_{\phi 1}^*(s', a'))$ .

Merk op dat deze twee netwerken los staan van de twee netwerken gebruikt om de weight updates stabieler te maken 3.4. Indien deze techniek ook gebruikt wordt, bevat het algoritme 4 netwerken.

Er wordt ook gebruik gemaakt van Adam [19] in de optimalisatie. Adam is een adaptief algoritme voor het optimaliseren van de learning rate dat speciaal is ontworpen voor het trainen van diepe neurale netwerken.

#### 3.10 Overschakelen naar Frozen Lake-omgeving

Omdat het onderzoek van deze masterproef vooral gericht is op het voorstellen van een policy van een DQN en niet op het verder onderzoeken van DDQN, is er besloten om over te stappen van omgeving. Het overslaan van sommige frames en

### 3 Deep Reinforcement Learning

---

**Algorithm 1** Pseudocode van trainingsproces van DQN-agent in Cart Pole omgeving

---

```
1: procedure trainAgent(epochs, doelFrequentie)           ▷ epochs voor aantal pogingen tot terminatie
2:   policyNetwerk  $\leftarrow DQN()$ 
3:   doelnetwerk  $\leftarrow DQN()$ 
4:   buffer  $\leftarrow ReplayBuffer()$ 
5:   optimizer  $\leftarrow Adam(l, w)$                                 ▷ learning rate en weight decay
6:   for i to epochs do
7:     Reset de omgeving
8:     voorgaandScherm  $\leftarrow geefOmgevingsBeeld()$ 
9:     huidigScherm  $\leftarrow geefOmgevingsBeeld()$ 
10:    huidigeStaat  $\leftarrow huidigScherm - voorgaandScherm$ 
11:    while not isBeindigd do
12:      Kies actie bepaald door policyNetwerk op basis van huidigeStaat
13:      beloning, isBeindigd  $\leftarrow geefResultaatVanActie(actie)$ 
14:      voorgaandScherm  $\leftarrow huidigScherm()$ 
15:      huidigScherm  $\leftarrow geefOmgevingsBeeld()$ 
16:      volgendeStaat  $\leftarrow huidigScherm - voorgaandScherm$     ▷ Observatie van nieuwe staat
17:      Sla (huidigeStaat, actie, volgendeStaat, beloning) op in buffer
18:      huidigeStaat  $\leftarrow volgendeStaat$                       ▷ overgang naar volgende staat
19:      optimaliseerPolicyNetwerk(policyNetwerk, optimizer, buffer)
20:    end while
21:    if i mod doelFrequentie == 0 then
22:      Laad de netwerkparameters van het policyNetwerk in het doelnetwerk
23:    end if
24:  end for
25: end procedure
```

---

### 3 Deep Reinforcement Learning

gebruik van een stapel van 4 of meer afbeeldingen zorgt er ook voor dat de ProtoTree geen beslissingen zou kunnen maken op basis van een prototype uit een enkele afbeelding (zie 4.1).

De nieuwe omgeving, Frozen Lake, bestaat uit een 4 bij 4 rooster waarbij de agent in de vorm van een elf, het pakket moet bereiken door op het ijs te glijden en zonder in plassen te vallen. De agent kan op elke tijdstip kiezen om naar links, onder, boven of rechts te bewegen. Wanneer de actor het pakket bereikt, ontvangt hij een beloning van +1. Indien de agent in een plas belandt, wordt de omgeving gereset en begint de agent terug vanaf de startpositie links bovenaan.

De omgeving in Gym biedt ook een mogelijkheid om de tegels slipperig te maken. In dit geval zal de agent in de gekozen richting bewegen met een waarschijnlijkheid van 1/3 en heeft hij een kans van 2/3 om zich te verplaatsen volgens een van de loodrechte richtingen op de gekozen richting (elke richting 1/3 kans).



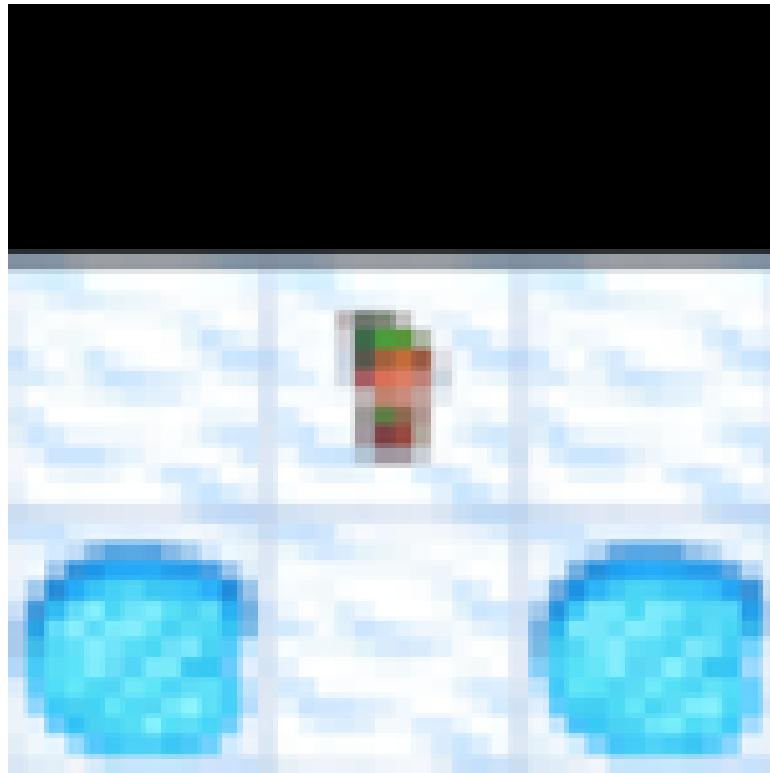
Figuur 3.9: Voorbeeld van de Frozen Lake-omgeving [7]

Om dit probleem op te lossen aan de hand van een DQN, is er geen toestand meer nodig die rekening houdt met verschillende frames om de snelheid of rotatie bij te houden. Als toestand is er gekozen voor een 3 bij 3 rooster dat gecentreerd is rond de agent. Het frame wordt eventueel met zwarte *padding* aan de rand van de omgeving aangevuld zoals te zien in figuur 3.10. Het frame wordt ook *downscaled* omwille van performantie.

Na licht *tunen* van de hyperparameters, werd het DQN toegepast op de nieuwe omgeving. De uitkomst van het algoritme leverde significant betere resultaten op dan bij de Cart Pole-omgeving. De gebruikte hyperparameters zijn te vinden in bijlage 6.7. Zoals te zien op de trainingscurve 3.12, begeeft de loss zich veel stabieler en is hij in staat om te convergeren.

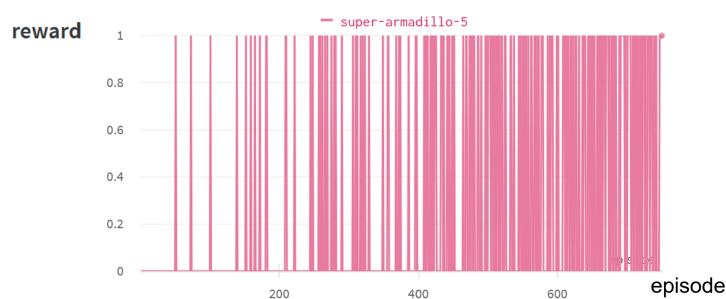
In figuur 3.11 is een plot met beloningen te zien, het geeft weer of de agent in staat is om een beloning te verkrijgen in een

### 3 Deep Reinforcement Learning



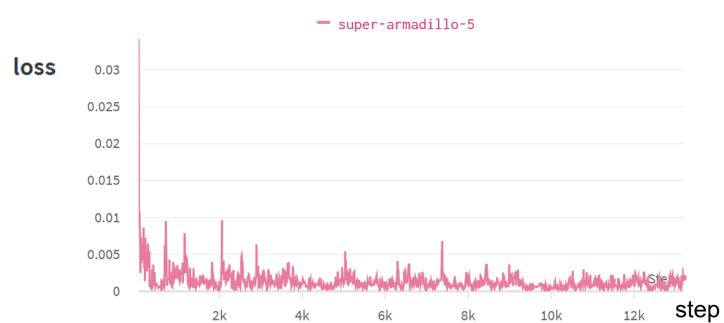
Figuur 3.10: Frame dat observatie van toestand voorstelt in het trainingsproces van agent in Frozen Lake

bepaalde episode. Na ongeveer 420 episodes is de agent in staat om consistent het einddoel te behalen zonder in plassen te vallen. De reden dat het plot nog af en toe aangeeft dat er geen beloning verkregen is, komt doordat de eindwaarde van het  $\epsilon$ -greedy algoritme op 0.05 geplaatst is, wat inhoudt dat de agent een kans van 1/20 heeft om telkens een willekeurige actie te kiezen in plaats van een actie uit zijn policy.



Figuur 3.11: Plot met beloningen van DQN-agent, met op de x-as de episodes

### 3 Deep Reinforcement Learning



Figuur 3.12: Trainingscurve met MSE-loss

# 4

## Prototype Tree

Diepe neurale netwerken hebben het afgelopen decennium ongelooflijke resultaten behaald en hebben hun superioriteit bewezen, vooral op het gebied van computervisie. Echter creëren de complexe architecturen en de hoogdimensionale feature space een typische black box-natuur. Het uitrollen van deze black box-instanties in kritieke omgevingen zoals zelf rijdende auto's, robotica en in de gezondheidszorg kan ernstige gevolgen met zich meebrengen.

Dit probleem leidt tot meer onderzoek naar interpreteerbare en verklaarbare modellen. Interpreteerbaarheid slaat op het vermogen om oorzaak en gevolg te bepalen op basis van een machine learning-model. Verklaarbaarheid houdt in dat uit het model zowel bepaald kan worden wat een parameter voorstelt als wat deze parameter bijdraagt tot de totale performantie van het model [20]. De Neural Prototype Tree, ook wel ProtoTree genoemd, kan een mogelijke oplossing leveren door een afweging te maken tussen performantie en interpreteerbaarheid [10].

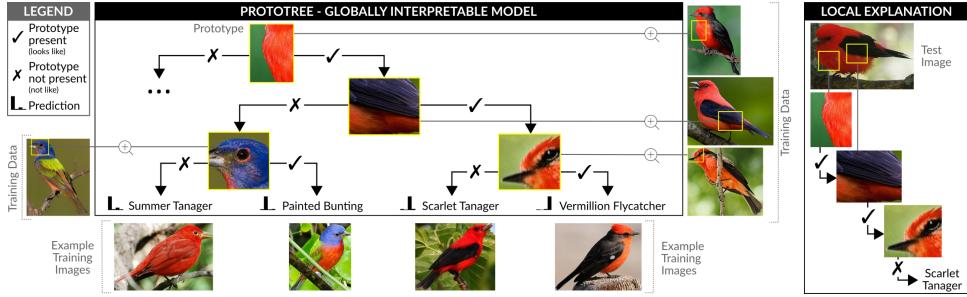
In dit hoofdstuk zal de structuur van de ProtoTree onderzocht en besproken worden. Daarna zal gekeken worden of gelijkaardige eindresultaten behaald kunnen worden zoals voorgesteld door Nauta et al. in de paper "Neural Prototype Trees for Interpretable Fine-grained Image Recognition" na het volgen van het trainingsproces[10]. In volgend hoofdstuk zal er gekeken worden hoe deze structuur gebruikt kan worden om de policy van een getraind DQN voor te stellen.

### 4.1 Interpreteerbaarheid aan de hand van prototypes

Het beslissingsproces wordt opgesplitst aan de hand van prototypes. Prototypes zijn africhtbare tensoren die worden geleerd met backpropagation. Tensoren zijn wiskundige objecten uit de lineaire algebra en de differentiaalmeetkunde die beschouwd kunnen worden als generalisatie van vectoren en matrices [21]. Een prototype kan worden gevisualiseerd als een patch van een trainingssample en werd voor het eerst geïntroduceerd in de paper "Prototypical Part Network"(ProtoPNet) geschreven door Chaofan Chen et al. [22].

De prototype tree combineert de kracht van DNN's met de verklarende mogelijkheden van binaire beslissingsbomen. Het bevat een ingebouwde boomstructuur waarvan de interne knopen bestaan uit prototypes zoals te zien in figuur 4.1. Wanneer een afbeelding geklassificeerd wordt aan de hand van de ProtoTree wordt op elk niveau gekeken in welke mate het prototype van een knoop een overeenkomst vormt met een deel van de afbeelding. Net zoals het Classification And Regression Tree (CART)-algoritme kan dit vergeleken worden met het spel "wie is het?". De splitsing gebeurt nu niet op basis van 1 feature, maar op prototypical part. De bladknopen geven distributies van de verschillende klassen weer. De ProtoTree is inherent

## 4 Prototype Tree

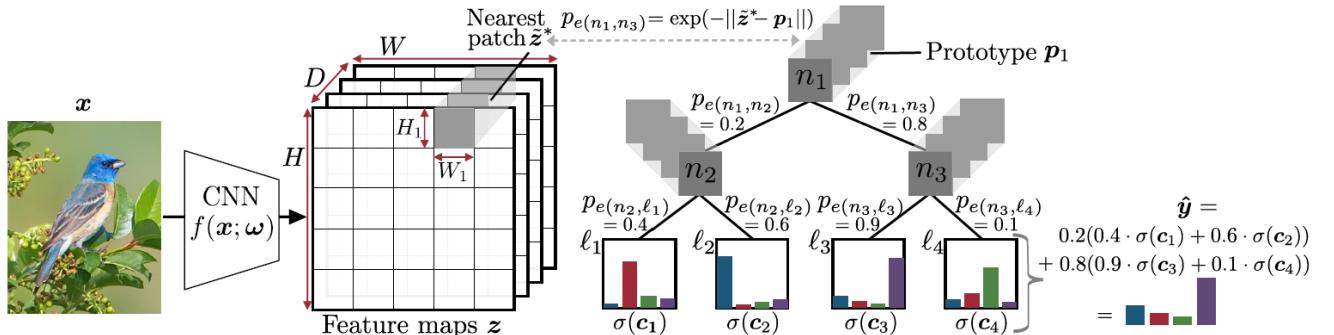


Figuur 4.1: Visuele voorstelling van een ProtoTree getraind op de CUB-200-2011 dataset bestaande uit 200 verschillende vogelsoorten [10]

interpreteerbaar. De interpreteerbare voorstelling wordt opgenomen in de structuur van het model. Dit in tegenstelling met post-hoc methoden die de gedachtegang van een DNN schetsen na het trainingsproces [10].

Na het trainingsproces van de ProtoTree kan heel het model samen met alle keuzes bekijken en beoordeeld worden. De evaluatie van deze globale weergave zorgt voor een duidelijk visueel overzicht van de gevaren en mogelijkheden van een model wat het inzetten van deze instanties een stuk veiliger maakt.

## 4.2 Componenten



Figuur 4.2: Componenten van de ProtoTree [10]

### 4.2.1 Doelstelling

Gegeven een afbeelding  $x$ , voorspelt een ProtoTree de kansverdeling over  $K$  klassen, aangeduid als  $\hat{y}$ . De letter  $y$  stelt het *one-hot* geëncodeerde label voor, zodat een ProtoTree kan getraind worden door de cross-entropy loss-functie tussen  $y$  en  $\hat{y}$  te minimaliseren [10].

## 4 Prototype Tree

### 4.2.2 CNN met voorgetrainde gewichten

Als eerste stap, zoals aangegeven in figuur 4.2, wordt de afbeelding door een Convolutional Neural Network (CNN) gestuurd. De ProtoTree gebruikt standaard een Residual Neural Network-50 (ResNet-50) waarvan de gewichten zijn voorgetraind op de INaturalist dataset. Deze dataset bevat 675 170 trainings- en validatiebeelden uit 5.089 natuurlijke fijnmazige categorieën. De dataset is onderverdeeld in 13 categorieën waaronder Plantae (planten), Insecta (insecten), Aves (vogels) en Mammalia (zoogdieren) [23].

De ProtoTree biedt ook de mogelijkheid tot andere convolutionele neurale netwerken zoals verschillende Visual Geometry Group (VGG)- en Densely Connected Convolutional Networks (DenseNet)-architecturen. De gewichten van deze architecturen zijn voorgetraind op ImageNet. Het neurale netwerk wordt ook wel weergegeven door  $f(x, \omega)$ , waarbij  $x$  de afbeelding voorstelt en  $\omega$  de gewichten van het neurale netwerk.

### 4.2.3 Feature maps z

De convolutionele output  $z$  bevat de afmetingen  $H \times W \times D$ . Het bestaat uit een aantal  $D$  feature maps die elk afmeting  $H \times W$  bevatten en vormt de input voor de boom. De boom bevat interne knopen en bladknopen. Elke interne knoop heeft exact twee kinderen en bevat een trainbare prototype.

Elke prototype bestaat uit een tensor met afmeting  $H_1 \times W_1 \times D$ . Waarbij  $H_1 < H$ ,  $W_1 < W$  en  $H_1 = W_1 = 1$ . De diepte van een prototype komt overeen met de diepte van de convolutionele output  $z$ . Hoe deze prototypes gekozen worden voor elke interne knoop wordt uitgelegd in 4.3. Het aantal prototypes dat de boom zal bevatten hangt af van de vooraf ingestelde diepte van de boom.

### 4.2.4 L2 Conv2D

Om te kijken welke patch van  $z$  het meest overeenkomt met een bepaald prototype  $p$ , wordt er gebruik gemaakt van een gegeneraliseerde vorm van convolutie, ook wel L2 Conv2D genoemd door Nauta et al. Elke prototype  $p_n$  uit de verzameling van prototypes  $P$  dient als kernel, wordt over elke patch  $\tilde{z}$  geschoven en berekent de euclidische afstand tussen  $p_n$  en de patch  $\tilde{z}$ . Het resultaat geeft voor elke prototype  $p_n$  een 2-dimensionale map met afstanden corresponderend met de verschillende patches van  $z$ . Aan de hand van een *Min pool*-operatie wordt de kleinste afstand geselecteerde en de bijhorende patch  $\tilde{z}$  gekozen. Deze patch wordt aangeduid met  $\tilde{z}^*$ , en is de patch die het meest overeenkomt met prototype  $p_n$ . Formeel kan deze bewerking uitgedrukt worden met de volgende formule:

$$\tilde{z}^* = \underset{\tilde{z} \in \text{patches}(z)}{\operatorname{argmin}} \|\tilde{z} - p_n\|. \quad (4.1)$$

### 4.2.5 Probabilistisch gewicht

De grootte van de afstand tussen de patch met grootste overeenkomst  $\tilde{z}$  en prototype  $p_n$  bepaalt de routering in de interne knoop. De tree werkt met soft routering. De patch  $\tilde{z}$  wordt gerouteerd naar beiden kinderen, elk met een gewicht in het interval [0-1]. Dit gewicht wordt bepaald aan de hand van volgende formule:

## 4 Prototype Tree

$$p_e(n, n.\text{rechts})(z) = \exp(-||\tilde{z} - p_n||). \quad (4.2)$$

De term  $p_e(n, n.\text{rechts})(z)$  geeft weer dat het probabilistisch gewicht bepaald wordt voor de tak (eng: *edge*)  $e$  die het prototype op het huidig niveau verbindt met het rechterkind. Het gewicht voor het routeren van  $z$  naar het linkerkind kan dan bepaald worden aan de hand van formule:

$$p_e(n, n.\text{links})(z) = 1 - p_e(n, n.\text{rechts}). \quad (4.3)$$

### 4.2.6 Afgelopen pad

Aangezien  $z$  bij elke interne knoop zowel naar links als naar rechts gestuurd wordt, elk met een gewicht dat afhangt van de euclidische afstand tussen de patch met grootste overeenkomst  $\tilde{z}$  en de prototype in de interne knoop, zal  $z$  elke tak doorlopen en in elke bladknoop  $l$  toekomen, met een bepaalde probabilistisch gewicht afhankelijk van de gewichten van het afgelopen pad  $P_l$ . Het pad  $P_l$  geeft de gevuldte sequentie aan taken weer vanuit de wortelknop tot de bladknoop  $l$ . Het gewicht van het routeren van  $z$  door het afgelopen pad  $P_l$  kan bepaald worden door middel van het product te nemen van de individuele gewichten van de takken die gevuld worden zoals weergegeven in onderstaande vergelijking:

$$\pi_l(z) = \prod_{e \in P_l} p_e(z). \quad (4.4)$$

Alle bladknopen bevatten een parameter  $c_l$  die tijdens het trainen van de boom geoptimaliseerd worden. Deze parameter geeft na het normaliseren met de softmax functie  $\sigma$  een probabiliteitsdistributie over de verschillende klassen  $K$ .

### 4.2.7 Predictie

De uiteindelijke voorspelling voor afbeelding  $x$ , komt tot stand door de convolutionele output  $z$  door te sijpelen naar alle bladknopen. Alle bladknopen dragen bij tot de finale predictie. De predictie bestaat uit de som over alle bladeren van het gewicht  $\pi_l(z)$  van het gevuldte pad  $P_l$  tot de bladknoop verminigvuldigd met de probabiliteitsdistributie van de bladknoop  $\sigma(c_l)$  (zie vergelijking 6.2):

$$\hat{y} = \sum_{l \in L} \sigma(c_l) \cdot \pi_l(f(x; \omega)). \quad (4.5)$$

Het resultaat is een probabiliteitsdistributie over alle de klassen  $K$ , waarbij de klasse met de grootste kans zal worden opgegeven in geval van een lokale voorspelling.

## 4.3 Trainingsproces ProtoTree

Het doel van het trainingsproces van de ProtoTree is om de parameters  $\omega$  van het CNN, de verschillende prototypes van de interne knopen en de probabiliteitsparameter  $c$  voor elke bladknoop te leren. In algoritme 2 is pseudocode te zien van het

## 4 Prototype Tree

proces zoal voorgesteld door Nauta et al. in "Neural Prototype Trees for Interpretable Fine-grained Image Recognition"[10].

De parameters  $\omega$  en de verzameling prototypes  $P$  worden aangeleerd via *mini-batch gradient descent* en *back-propagation* door middel van de *cross-entropy loss*-functie tussen  $y$  en  $\hat{y}$ . De probabiliteitsparameter  $c$  kan ook aangeleerd worden met back-propagation en cross-entropy, maar Nauta et al. opteerden om dit niet te doen en kozen voor een methode zonder afgeleiden. In de paper "Deep Neural Decision Forests"[24] merkten Kotschieder et al. op dat de optimalisatie van de parameters in de bladeren van een beslissingsboom een convex optimalisatieprobleem is en bedachten hiervoor onderstaande strategie:

$$\pi_{ly}^{(t+1)} = \frac{1}{Z_l^{(t)}} \sum_{(x,y') \in T} \frac{1_{y=y'} \pi_{ly}^{(t)} \mu(x|\Theta)}{P_T[y|x, \Theta, \pi^{(t)}]}. \quad (4.6)$$

Waarbij  $T$  de trainingsset is waarvan elke instantie  $(x, y')$  bestaat uit input  $x$  en een overeenkomstig label  $y'$ . Hierbij staat  $\pi_{ly}$  voor de kans dat een voorbeeld een blad bereikt en de klasse  $y$  aanneemt, terwijl  $\mu(x|\Theta)$  wordt beschouwd als de routeringsfunctie die de kans geeft dat sample  $x$  blad  $l$  bereikt. De term  $1_{y=y'}$  is een indicatorfunctie die gelijk is aan 1 als  $y = y'$  en 0 anders. De uitdrukking  $Z_l^{(t)}$  is een normalisatiefactor die ervoor zorgt dat  $\sum_y \pi_{ly}^{(t+1)} = 1$ .  $P_T$  staat voor de predictie.

Kotschieder et al. toonden aan dat in deze uitdrukking elke update een strikte vermindering in de verwachte waarde van de loss-functie oplevert (wat inhoud dat het model beter presteert op ongeziene data), totdat er een vast punt is bereikt. Nauta et al. brachten deze redenering over naar de *logits* van de ProtoTree wat resulteert in onderstaande formule:

$$c_{i,j}^{(t+1)} = \sum_{(x,y) \in T} (\sigma(c_l^{(t)}) \odot y \odot \pi_l) \oslash \hat{y}. \quad (4.7)$$

Symbol  $\odot$  staat voor elementsgewijze multiplicatie en  $\oslash$  voor elementsgewijze deling.

## 4.4 Snoeien en Visualisatie van eindresultaat

### 4.4.1 Snoeien

Om de interpreteerbaarheid te verhogen, kunnen nietszeggende bladknopen die geen duidelijke voorspelling maken, weggesnoeid worden. Op deze manier kunnen outputs vermeden worden waarbij de ProtoTree geen besluit kan vormen en een uniforme distributie teruggegeven wordt over  $K$  klassen. De ProtoTree bevat een hyperparameter  $\tau$  die de threshold voor het snoeien weergeeft.

Alle bladeren waarvan  $\max(\sigma(c_l)) \leq \tau$  (wat inhoud dat de kans op voorkomen van de meest waarschijnlijke klasse volgens de voorspelling kleiner is dan de threshold) worden weggehaald. Het is aangeraden om de waarde van  $\tau$  licht groter te zetten dan  $1/K$ . Indien alle bladeren in een deelboom  $T' \subset T$  weggesnoeid zijn, wordt  $T'$  samen met de interne prototypes weggeknipt.

---

**Algorithm 2:** Training a ProtoTree

---

**Input:** Training set  $\mathcal{T}$ , max height  $h$ ,  $nEpochs$

- 1 initialize ProtoTree  $T$  with height  $h$  and  $\omega, \mathbf{P}, \mathbf{c}^{(1)}$ ;
- 2 **for**  $t \in \{1, \dots, nEpochs\}$  **do**
- 3     randomly split  $\mathcal{T}$  into  $B$  mini-batches;
- 4     **for**  $(\mathbf{x}_b, \mathbf{y}_b) \in \{\mathcal{T}_1, \dots, \mathcal{T}_b, \dots, \mathcal{T}_B\}$  **do**
- 5          $\hat{\mathbf{y}}_b = T(\mathbf{x}_b)$ ;
- 6         compute loss  $(\hat{\mathbf{y}}_b, \mathbf{y}_b)$ ;
- 7         update  $\omega$  and  $\mathbf{P}$  with gradient descent;
- 8         **for**  $\ell \in \mathcal{L}$  **do**
- 9              $\mathbf{c}_\ell^{(t+1)} = \frac{1}{B} \cdot \mathbf{c}_\ell^{(t)}$ ;
- 10              $\mathbf{c}_\ell^{(t+1)} +=$  Eq.4.7 for  $\mathbf{x}_b, \mathbf{y}_b$ ;
- 11     prune  $T$  (optional);
- 12     replace each prototype  $p_n \in \mathbf{P}$  with its nearest latent patch  $\tilde{z}_n^*$  and visualize;

---

#### 4.4.2 Visualisatie van eindresultaat met prototypes

Zoals eerder vermeld, bestaan de verschillende prototypes uit tensoren met de afmeting  $H \times W \times D$ . Om een duidelijke weergave te creëren en te weten welk deel een bepaalde prototype voorstelt, worden de prototypes terug naar de pixelruimte omgezet. Dit gebeurt na het trainen van het prototype zoals weergegeven op lijn 12 in het algoritme.

Om dit te doen, wordt de trainingsdataset opnieuw doorlopen en wordt voor elke prototype de dichtstbijzijnde latente patch bijgehouden. Ook informatie over de afbeelding die werd gebruikt voor de projectie, wordt opgeslagen. Hierna wordt de afbeelding  $x_n^*$  die de meest gelijkende patch  $\tilde{z}_n^*$  bevat, omgezet naar een *similarity map*.

Deze map bestaat uit een 2-dimensionale lijst met scores die de gelijkenis tussen prototype  $p_n$  en alle patches weergeeft. Deze werkwijze kan worden weergegeven met onderstaande formule:

$$S_n^{(i,j)} = \exp(-\|\tilde{z}_n^{(i,j)} - p_n\|). \quad (4.8)$$

In deze uitdrukking geven  $i$  en  $j$  respectievelijk de rij en kolom weer waarop een bepaalde patch zich bevindt. Vervolgens wordt de similarity map geëupscaled door middel van bicubische interpolatie tot de afmetingen van  $x_n^*$ . Bicubische interpolatie is een wiskundige methode die wordt gebruikt om digitale afbeeldingen te vergroten of verkleinen door de kleur- of intensiteitswaarden van nieuwe pixels te schatten op basis van de waarden van naburige pixels in de originele afbeelding. De naburige pixels die een rol bijdragen bevinden zich links, rechts boven en onder een bepaalde pixel.

Hierna wordt het prototype in de visualisatie getekend in de vorm van een rechthoek in afbeelding  $x_n^*$  op de positie van de

## 4 Prototype Tree

dichtsbijzijnde patch  $\tilde{z}_n^*$ . Een voorbeeld van het visualisatieproces uit "Neural Prototype Trees for Interpretable Fine-grained Image Recognition" [10] is te zien in afbeelding 4.3.



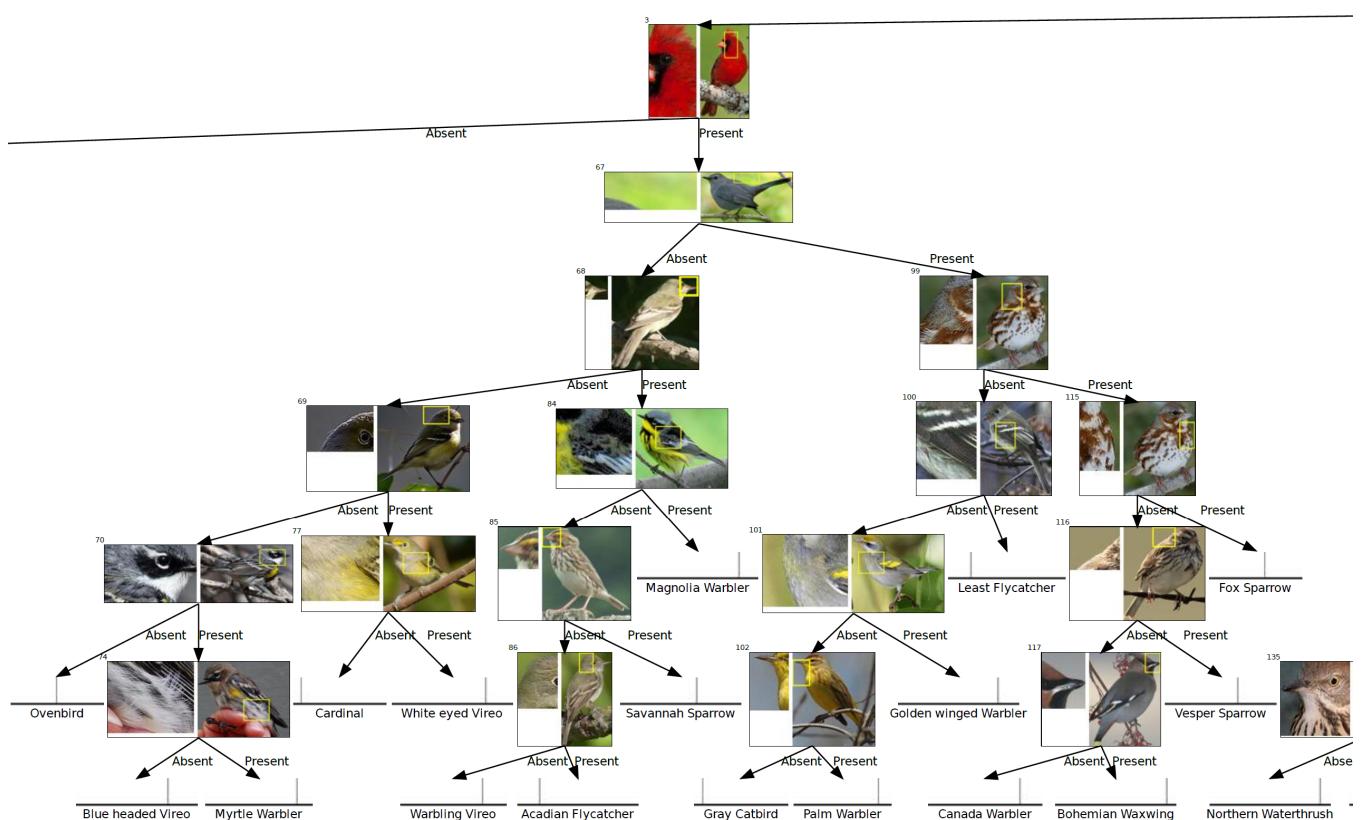
Figuur 4.3: Voorbeeld van visualisatieproces van een prototype [10]

## 4.5 Resultaten na trainen van ProtoTree

Na het bekijken van de nodige theorie en het bestuderen van de volledige ProtoTree was het tijd om het project geschreven door Nauta et al. werkende te krijgen en te zien of gelijkaardige resultaten behaald konden worden. Na het installeren van de benodigde bibliotheken en het bestuderen van de *codebase* werd het trainingsproces in actie gezet. Zoals eerder vermeld, wordt de ProtoTree getest op de CUB-200-2011 dataset bestaande uit 200 verschillende vogelsoorten [10].

Na een periode van 8 uur trainen op de Nvidia GeForce GTX 1080 Ti Graphical Processing Unit (GPU) op GPULab was het trainingsproces afgerond. De behaalde nauwkeurigheid na een trainingsperiode van 100 epochs met de hyperparameters voorgesteld in de paper bedroeg 82.28%. Dit resultaat is zeer gelijkaardig aan het resultaat voorgesteld door Nauta et al (82.50%). Een deelboom van de eindvisualisatie van de ProtoTree na het trainen, is te zien in figuur 4.4.

## 4 Prototype Tree



Figuur 4.4: deelboom van ProtoTree getraind op CUB-200-2011 dataset

# 5

## Voorstellen van policy van DQN-agent

In hoofdstuk 3 is er besproken hoe de structuur van een DQN in elkaar zit en hoe de agents getraind worden. Het verkregen resultaat leverde een getrainde policy op in de vorm van gewichten van een CNN. In dit hoofdstuk wordt er gekeken hoe deze policy zou kunnen voorgesteld worden aan de hand van de ProtoTree besproken in hoofdstuk 4.

De transparantie van de voorstelling kan een duidelijk beeld schetsen van hoe de agent tot zijn beslissingen gekomen is, zodat het DNN met typische black box-natuur toch in kaart gebracht kan worden. Dit kan bijdragen aan het vergroten van het vertrouwen in neurale netwerken en het stimuleren van het gebruik in gebieden waar een hoog niveau van verantwoording en transparantie noodzakelijk is. Dit gebrek aan vertrouwen en transparantie is een van de belangrijkste redenen waarom RL vandaag de dag minder aanwezig is in het bedrijfsleven dan supervised learning-technieken of unsupervised learning-technieken.

Dit hoofdstuk vormt de initiële stap naar een interpreteerbaar DQN. Na het trainingsproces van het klassieke DQN kunnen de afbeeldingen gelabeld worden met de optimale acties uit de policy om de ProtoTree op een supervised-manier te trainen. Deze methodologie zal onderzocht worden in dit hoofdstuk. Eerst wordt besproken hoe deze supervised-manier van werken in elkaar steekt. Vervolgens wordt er gekeken naar de ondervonden problemen en of het resultaat overeenkomt met de initiële gedachtegang.

### 5.1 Labelen van afbeeldingen

Het labelen van afbeeldingen is een repetitieve, tijdrovende bezigheid. Het labelen van gegevens vereist vaak menselijke expertise en tijd. Het is een dure operatie om mensen in te huren om deze gegevens handmatig te labelen. Manueel elke toestand afgaan in een omgeving en dan de optimale actie opvragen is niet haalbaar in situaties met grote toestandsruimtes. De afbeeldingen, die een toestand in de omgeving voorstellen, kunnen automatisch gelabeld worden door na het trainen de policy uit te testen op de omgeving, waarbij elke stap de optimale actie gekozen wordt ( $\epsilon = 0$ ) en waarbij het optimalisatieproces van de gewichten wordt uitgeschakeld. Een probleem hierbij is dat wanneer de agent de optimale policy volgt, het niet in alle verschillende toestanden terecht zal komen. Alleen de toestanden die deel uitmaken van het optimale pad zullen gelabeld worden, wat ervoor zorgt dat de dataset kleiner zal zijn.

Een andere manier zou kunnen zijn om niet telkens de optimale actie te kiezen, maar de agent op een aantal stappen een willekeurige actie te laten nemen door de waarde van  $\epsilon$  te vergroten. Wanneer een willekeurige actie genomen wordt, zal

## 5 Voorstellen van policy van DQN-agent

de toestand niet gelabeld worden met deze actie aangezien dit niet de optimale actie is. Dit zorgt er enkel voor dat andere toestanden verkend worden. Tijdens het trainen van het DQN worden echter niet gegarandeerd alle mogelijke toestanden overlopen. De agent verkent de omgeving door acties te ondernemen gekoppeld aan beloningen en gebruikt deze ervaring om zijn actie-waardeschattingen bij te werken. Er is geen garantie dat de agent alle mogelijke toestanden afloopt op deze manier.

De mate waarin de agent de omgeving verkent, hangt af van verschillende factoren, zoals de gebruikte exploratiestrategie, de complexiteit van de omgeving en het aantal episodes dat wordt gebruikt voor training. In de praktijk is het vaak moeilijk of onmogelijk om elke mogelijke toestand te onderzoeken, vooral voor grote en complexe omgevingen. Om een goed beeld te krijgen van de keuzes die genomen zijn aan de hand van de policy, is er gekozen om te labelen door op elke stap de actie voorgesteld door de policy te kiezen, zodat elke toestand in de dataset gezien is door de policy en zo een accurater beeld verkregen wordt.

### 5.2 Opbouw van de dataset

Nadat de verschillende afbeeldingen gelabeld zijn, bevat de dataset afbeeldingen met 4 verschillende labels (*left*, *right*, *up*, *down*). De agent in de Frozen Lake-omgeving start steeds linksboven en moet een weg zoeken doorheen het ijs tot het doel rechtsonder bereikt is. Door de opbouw van dit probleem is het aantal keer dat de acties up of left genomen worden klein tot nihil. Dit creëert een zeer ongebalanceerde dataset. De afbeeldingen met deze klassen zijn weggenomen. De opdracht voor de ProtoTree bestaat dan uit een binair classificatieprobleem tussen de klassen down en right, waarvan twee gelabelde afbeeldingen te zien zijn in figuur 5.1.



Figuur 5.1: Voorbeeld van afbeeldingen uit de dataset van omgevingsbeelden gelabeld met acties genomen door de policy

### 5.3 Eerste resultaat van voorstelling van policy

Het eerste resultaat na trainen van de ProtoTree met de gelabelde data van de policy was zeer ondermaats en praktisch onbruikbaar. Het doel van de prototypes gelinkt aan klasse k is om de meest relevante onderdelen voor het identificeren van afbeeldingen van klasse k te bevatten [22].

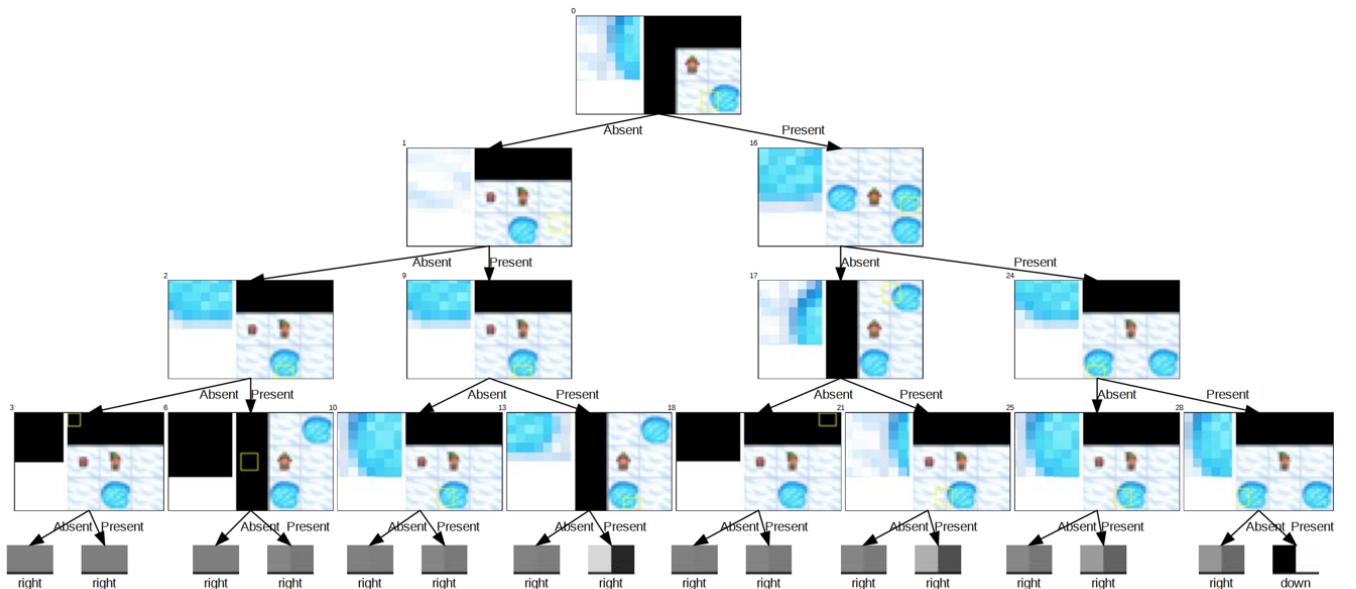
Goed gekozen prototypes resulteren na het trainingsproces van de ProtoTree in een kansverdeling waarbij de klasse van

## 5 Voorstellen van policy van DQN-agent

de afbeelding waarvan de prototype tree afstamt, een hoge waarschijnlijkheid heeft. De ProtoTree kijkt door middel van een gegeneraliseerde vorm van convolutie in combinatie met een *minpool*-operatie of het prototype aanwezig is in een afbeelding.

Deze operatie houdt echter geen rekening met de positie van voorkomen van een prototype in de afbeelding. Dit zorgt voor een robuuste herkenning die invariant is tegen rotaties, translaties en andere affiene transformaties. De afbeeldingen zijn telkens gecentreerd rond de agent en de initiële gedachtengang was dat de ProtoTree prototypes zou kiezen die een deel van de agent en een deel van de zwarte rand, een plas of het doel bevatten zodat hieruit de positie van de agent ten opzichte van de zwarte rand, plas of doel kan worden afgeleid.

Zoals te zien in figuur 5.2 zijn de geselecteerde prototypes totaal niet representatief voor een bepaalde klasse. De gekozen prototypes bevatten zwarte delen van de rand, plassen of witte pixels van het ijs. Geen enkel prototype bevattet een deel van de agent en elke prototype zou aanwezig kunnen zijn in elk van de klassen. Het resultaat is dan ook dat de voorspellingen bij de meeste paden geen besluit nemen.



Figuur 5.2: Eerste resultaat van toepassen van ProtoTree op dataset met observatiebeelden gelabeld met acties van de policy

Na dit eerste teleurstellende resultaat werd er gekeken naar verschillende manieren om dit resultaat te verbeteren. Langer trainen dan 100 episodes, de snoefactor verhogen en verschillende dieptes van de boom leverden geen beter resultaat.

Zoals vermeld in sectie 4.2.3 bestaan de prototypes uit tensoren van dimensie  $1 \times 1 \times D$ . De voor de hand liggende manier om de visualisatie van de prototypes groter te krijgen zodat deze de afmeting hebben van een volledige *grid cell*, is de afmetingen van de tensoren vergroten. Het doorlopen van de gegeneraliseerde convolutie operatie op tensoren met een grotere dimensie bleek echter zeer computationeel intensief. Het trainingsproces resulterde telkens in *RuntimeError: CUDA error: out of memory*. Om dit op te lossen zou de codebase van het project aangepast moeten worden zodat parallelle verwerking over meerdere GPU's ondersteund wordt.

Een veel computationeel zuinigere manier die gekozen is, is de originele afbeeldingen downscalen zodat de tensor met

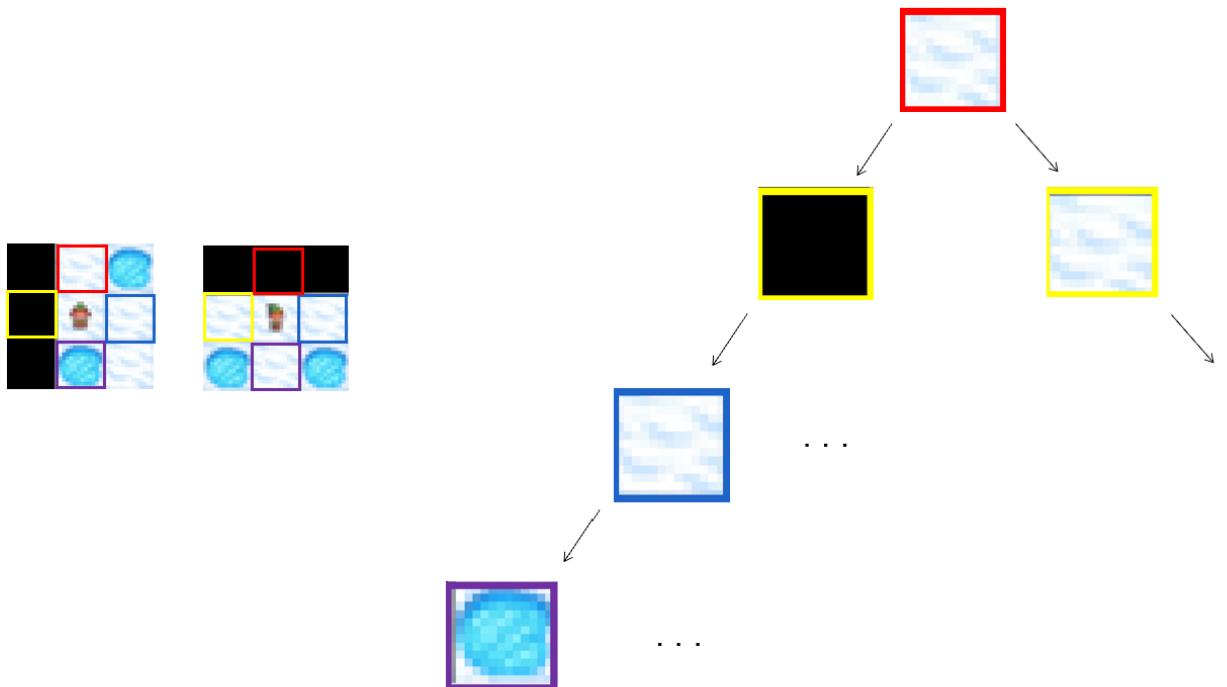
## 5 Voorstellen van policy van DQN-agent

afmeting 1x1xD een veel grotere ruimte inspant op de visualisatie. Het resultaat na het vergroten van de prototypes leverde echter ook geen verbeteringen op.

### 5.4 Inductieve bias toevoegen

Na het behalen van zwakke resultaten werd er gekeken naar een manier om inductieve bias te introduceren, zodat er rekening gehouden wordt met positie in de representatie van een prototype. In plaats van representatieve prototypes over de hele afbeelding te leren, worden specifieke delen van een afbeelding geselecteerd op basis van de diepte waar het prototype zich op bevindt. De ProtoTree zal prototypes in de boom selecteren op basis van vaste posities rond de agent.

De manier van werken wordt weergegeven in figuur 5.3. In het eerste knooppunt kijkt de boom naar de pixelwaarden van de rastercel boven de agent en selecteert een prototype op basis van de waarden. De tweede laag bestaat uit rastercellen die links van de agent te vinden zijn, derde rechts, vierde beneden en misschien op het laatste niveau zelfs weer omhoog omdat anders alleen het prototype in de wortel als "boven" zou worden geselecteerd. De verwijzing van de rastercellen die diagonaal van de agent zijn gepositioneerd zijn geen probleem omdat de agent alleen verticaal en horizontaal kan bewegen.

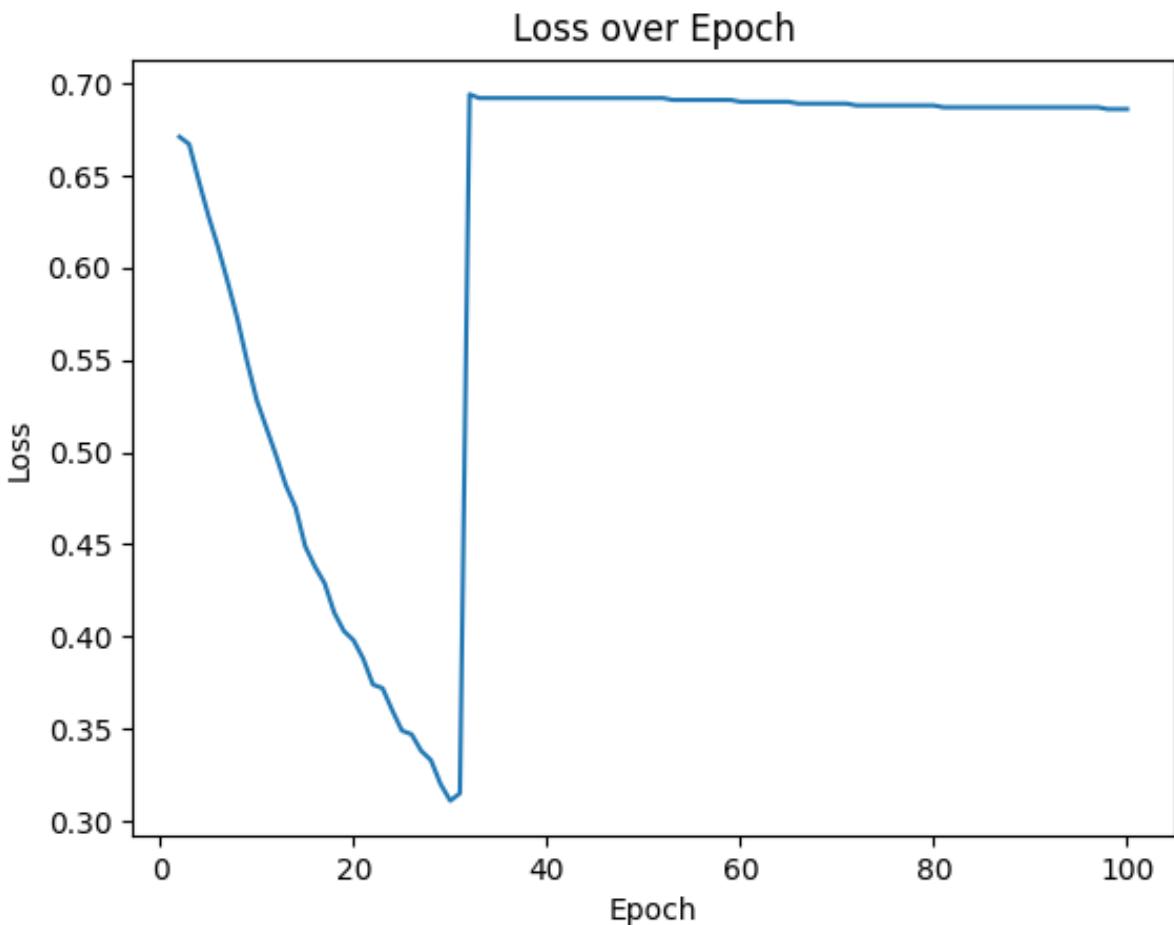


Figuur 5.3: Voorstelling van werkwijze van voorstelling van prototypes met inductive bias afhankelijk van de positie.

## 5 Voorstellen van policy van DQN-agent

### 5.5 Bevriezen van gewichten

Tijdens het ontwerpen van de inductieve bias en het testen van verschillende onderdelen van de ProtoTree werd een vaststelling gedaan in de *logfile* die de nauwkeurigheid en loss-waarde weergeeft per epoch. De bevinding die gevonden werd uit figuur 5.4, was dat er rond epoch 30 telkens een daling in de nauwkeurigheid plaatsvond en een enorme stijging in loss-waarde. Na verdere analyse werd opgemerkt dat deze trend enkele epochs standhield en dan geleidelijk aan de nauwkeurigheid zeer traag toenam.



Figuur 5.4: Plot die de loss-waarde weergeeft per epoch in het trainingsproces van de ProtoTree wanneer de gewichten van het CNN bevrieten worden voor 30 epochs

Na een periode van onderzoek naar wat de oorzaak zou kunnen zijn van dit verschijnsel, werd één van de meer dan twintig hyperparamters opgemerkt. De hyperparameter om het voorgetrainde VGG-netwerk x-aantal epochs te bevriezen, stond ingesteld op 30. Na het wijzigen van deze parameter naar het aantal trainingsepoths, volgde de loss-waarde en de nauwkeurigheid respectievelijk een dalende en stijgende trend.

De ProtoTree optimaliseert nu niet meer de paramters  $\omega$  van het CNN, maar blijft de voorgetrainde gewichten gebruiken

## 5 Voorstellen van policy van DQN-agent

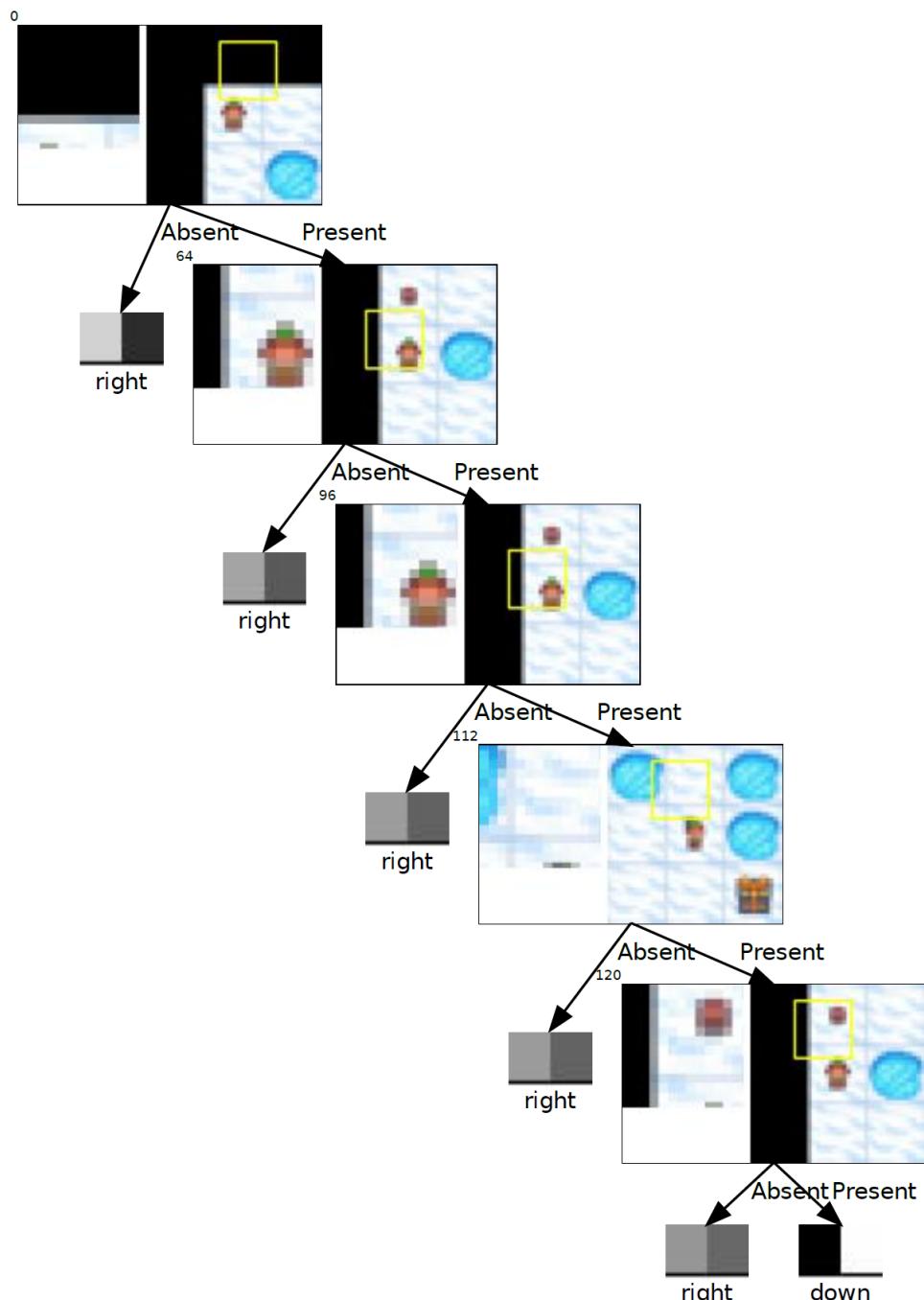
zonder wijziging. Enkel de laatste toegevoegde  $1 \times 1$  convolutie-laag wordt nog geoptimaliseerd. Het enige doel van de structuur is nu om de verschillende kenmerkende prototypes te leren, de probabiliteitsparameter  $c$  voor elke bladknoop en de toegevoegde convolutie-laag.

In figuur 5.5 is het resultaat te zien van de boom wanneer de voorgetrainde gewichten van het CNN onaangepast blijven. In de visualisatie is te zien dat de gekozen prototypes telkens een deel van de agent bevatten, samen met een deel van de rand, het ijs of een plas zoals initieel verwacht werd.

In tegenstelling tot de gekozen prototypes in figuur 5.2, zijn de prototypes nu wel degelijk typerend voor de bijhorende klassen. De voorspelde eindpredictie is soms wel kort door de bocht genomen. Dit komt doordat de dataset redelijk beknopt is. De omgeving bestaat uit een  $4 \times 4$  raster en de gelabelde afbeeldingen bestaan uit een  $3 \times 3$  vierkant gecentreerd rond de agent wat resulteert in een kleine dataset. Doordat de prototypes nu wel typerend zijn voor de bijhorende klasse, is er afgestapt van de positionele inductieve bias. Om de accuraatheid van de eindpredicties te verbeteren werd er gekeken naar een manier om de dataset te vergroten.

De Frozen Lake-omgeving bevat een mogelijkheid om het raster uit te breiden. Doordat de observatiebeelden die geleverd worden aan het DQN bestaan uit een  $3 \times 3$  omgevingsbeeld rond de agent zal het uitbreiden van de omgeving echter niet echt veel opbrengen. De gelabelde observatiebeelden zullen er hetzelfde uitzien, ook al is de intrinsieke positie verschillend in de totale toestand. De gehele toestand is echter niet te zien is op het observatiebeeld.

## 5 Voorstellen van policy van DQN-agent



Figuur 5.5: Resultaat van toepassen van ProtoTree op dataset met observatiebeelden gelabeld met acties van de policy, waarbij de gewichten van het CNN onaangepast blijven.

## 5 Voorstellen van policy van DQN-agent

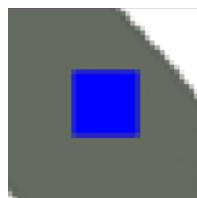
### 5.6 Eigen RL-omgeving om grotere dataset te verkrijgen

Om een grotere dataset te verkrijgen is er overgestapt naar een eigen gecreëerde omgeving. De omgeving is ontworpen door middel van Gym en de Pygame library. Pygame is een verzameling van Python-modules die ontworpen zijn voor het schrijven van videogames. Pygame voegt functionaliteit toe bovenop de uitstekende SDL-bibliotheek. Hiermee kunnen volledige functionele games en multimediacode's in Python gemaakt worden [25].

De omgeving is te zien in figuur 5.6. Het bestaat uit een grijs pad, de agent in de vorm van een blauw vierkant en het doel weergegeven door een groen vierkant. Het doel van de agent is om in zo min mogelijk stappen het einddoel te bereiken, zonder het pad te verlaten. De mogelijke actieruimte van de actor blijft discreet, met als mogelijke stappen om links, rechts, naar onder en naar boven te bewegen. Het DQN krijgt terug zoals bij de Frozen Lake-omgeving observatiebeelden te zien die gecentreerd zijn rond de agent. De beelden bevatten een breedte en lengte gelijk aan 3 keer de afmeting van de agent. Een voorbeeld hiervan is te zien in figuur 5.7. Het pad bevat verschillende bochten en kronkels wat ervoor zorgt dat de opgebouwde dataset van observatiebeelden met voorgestelde acties van de policy, een grotere variëteit zal bevatten.



Figuur 5.6: Beeld van RL-omgeving ontworpen met de Pygame-module



Figuur 5.7: Observatiebeeld gecentreerd rond agent

## 5 Voorstellen van policy van DQN-agent

### 5.7 DQN-agent trainen op eigen RL-omgeving

De DQN-agent trainen op de nieuwe omgeving verliep stroef. Als eerste poging werd het beloningssysteem van de Frozen Lake-omgeving overgenomen. De agent krijgt hierbij enkel een beloning met waarde gelijk aan één indien hij het doel bereikt. Voor alle overige stappen ontvangt hij geen beloning. Het resultaat na trainen met dit *sparse*-beloningssysteem was dat de agent nauwelijks het doel bereikte en constant vast kwam te zitten aan de randen. Om dit op te lossen werd het beloningsschema aangepast. Na deze aanpassing ontving de agent een straf van -1 elke keer dat hij de kant raakte. Ook ontvangt de agent nu een negatieve waarde voor elke stap op het pad, dit om de agent te stimuleren om de snelste weg richting het doel te nemen. Ook dit schema leverde geen betere resultaten op.

Na het uitproberen van verschillende sparse-beloningssystemen, werd het beloningsschema omgevormd naar een *dense*-systeem. In dit systeem ontvangt de agent een beloning die omgekeerd evenredig is met de afstand van de positie met het doel. Deze beloning is geschaald tussen 0 en 0.6. De agent ontvangt nog steeds een beloning van 1 als hij het doel bereikt en een straf van -1 wanneer hij botst met de omgeving.

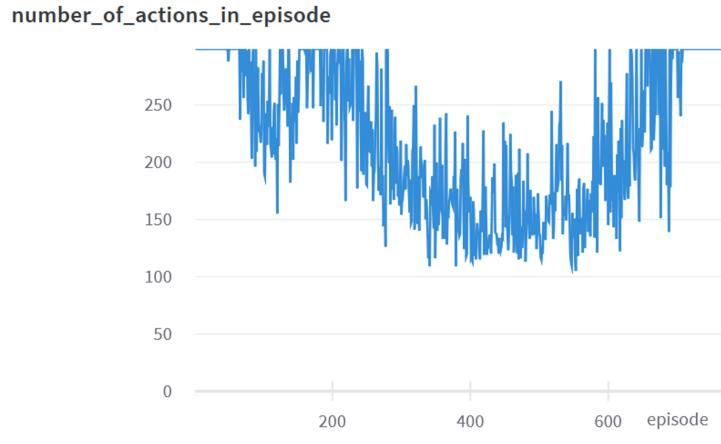
Wanneer na training deze policy werd uitgetest, was echter te zien dat de agent nu steeds een deel van het pad nam tot op een bepaald punt en dan besloot om de resterende stappen op en neer te bewegen. In RL staat dit fenomeen ook wel bekend als *reward hacking*. Het impliceert een out-of-the-box oplossing die meer beloning geeft dan het bedoelde antwoord van de ontwerper van de beloningsfunctie [26]. De opstelling van het beloningsschema zorgde ervoor dat de agent bij elke stap dat hij heen en weer bewoog, een positieve beloning kreeg. Om dit te vermijden werd de beloningsfunctie aangepast door telkens wanneer de agent een toestand herbezoekt, de waarde 1 af te trekken van de waarde die hij zou krijgen wanneer hij de toestand voor de eerste keer bezoekt. Deze aanpak was echter ook niet echt een goed idee. De agent leert op basis van 3 bij 3 beelden die bijgehouden worden in een geheugenbuffer. Beelden van verschillende toestanden kunnen zeer gelijkaardig zijn. De agent heeft geen notie van de onderliggende staat van de omgeving en weet niet welke toestanden hij allemaal al bezocht heeft. Een andere manier om reward hacking te vermijden, is om in plaats van een genormaliseerde beloning die omgekeerd evenredig is met de euclidische afstand te geven, het verschil in afstand tussen de huidige staat en de volgende staat als beloning geven. Wanneer nu de agent onmiddellijk een stap achteruit zet na het nemen van een stap vooruit, krijgt hij een straf met dezelfde absolute waarde, maar verschillend teken ten opzichte van de stap ervoor. Dit zorgt ervoor dat de agent niet meer gestimuleerd wordt om heen en weer te bewegen, aangezien de netto som van de beloningen van deze twee acties resulteert in 0.

In figuur 5.8 is een plot te zien dat het aantal acties per episode weergeeft. Indien de agent na 300 episodes het doel nog niet bereikt heeft, wordt de omgeving herstart. Dit om eindeloos ronddwalen tegen te gaan. In figuur 5.9 is de gemiddelde beloning over alle genomen acties tot op een bepaalde episode te zien en in figuur 5.10 de gemiddelde beloning over een episode genomen.

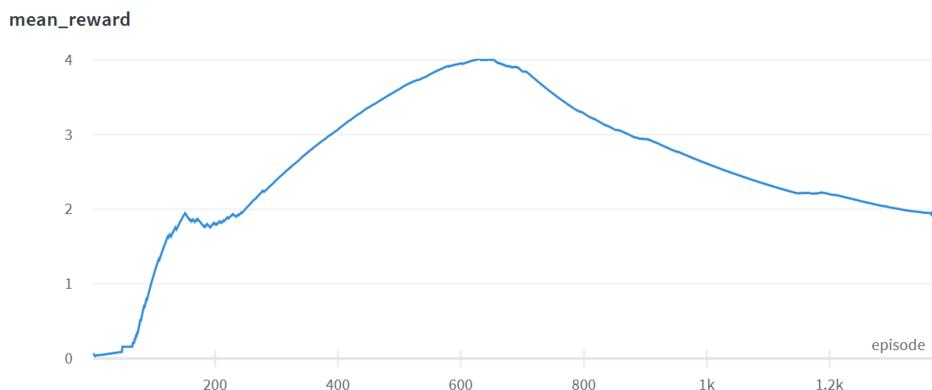
Het opgestelde pad is ongeveer te bereiken in een 120-tal stappen. Op de plots is te zien dat rond episode 300 de agent voor het eerst het doel bereikt. Vanaf dan slaagt hij een periode van ongeveer 300 episodes erin om consistent het doel te bereiken en stijgt de gemiddelde waarde volop. De schommelingen in het aantal acties zijn te wijten aan de exploratiefactor. Rond episode 600 is te zien dat de prestaties fel afnemen. De gemiddelde beloning crasht volledig en het aantal acties in een episode stijgt tot op het punt dat de agent zelfs niet meer in staat is om het doel te bereiken voordat de omgeving wordt

## 5 Voorstellen van policy van DQN-agent

herstart.



Figuur 5.8: Aantal acties per episode



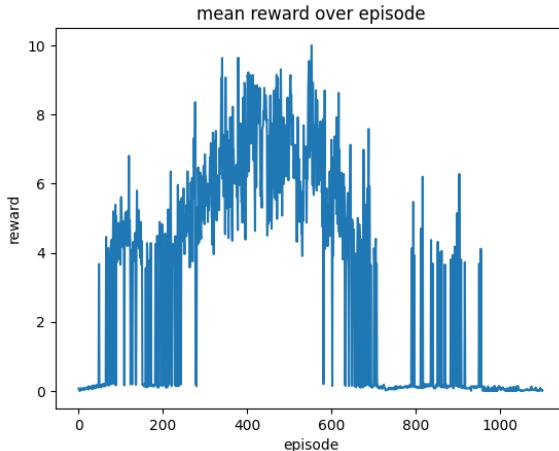
Figuur 5.9: Gemiddelde beloning over alle acties tot op een bepaalde episode

Dit verschijnsel staat bekend in RL als *policy collapse*. In plaats van dat de Q-waardefunctie convergeert naar de optimale policy naarmate de agent ervaring opdoet, divergeert ze (en de parameters van de approximator divergeren ook). Dit kan gebeuren bij het gebruik van niet-lineaire functiebenaderingen om actiewaarden te schatten. Meer in het algemeen gebeurt dit meestal wanneer een van de volgende kenmerken van het probleem aanwezig zijn:

- 1) Een functiebenadering, vooral een niet-lineaire (hoewel zelfs lineaire functiebenaderingen kunnen divergeren).
- 2) Een bootstrap-methode, b.v. Temporal Difference (TD) Learning (inclusief SARSA en Q-learning), waarbij waarden worden bijgewerkt vanuit dezelfde waardeschatter die wordt toegepast op opeenvolgende stappen.
- 3) Off-Policy trainen. [27, 28].

De gebruikte DQN-architectuur zoals beschreven in hoofdstuk 3, maakt gebruik van een CNN om de Q-waarden te benaderen.

## 5 Voorstellen van policy van DQN-agent



Figuur 5.10: Gemiddelde beloning over episode genomen

Ook is het een Off-policy algoritme en maakt het gebruik van TD Learning, waarbij de Q-target opgewekt wordt door het doelnetwerk. Aangezien DQN gebruikt maakt van al deze kenmerken, is het extreem gevoelig voor dit probleem. Deze kwestie oplossen is zeer moeilijk. Verschillende suggesties die tot convergentie kunnen leiden, zoals de implementatie van een DDQN om de overschatting van de argmax-operator tegen te gaan, kleinere learning rate voor stabilisatie, het policy-netwerk aan een lagere frequentie overkopiëren naar het doelnetwerk, het verlagen van de batch size en andere exploratieverhoudingen werden uitgeprobeerd. Ook werd de kleur van de achtergrond naar zwart geplaatst en de kleur van de agent gewijzigd om het probleem iets eenvoudiger te maken en hoger contrast in te voeren. Na uitvoerig testen van deze verschillende voorstellen, divergeerde de policy nog steeds. Een andere manier om toch een getrainde policy over te houden bij dit probleem is *early stopping*. Bij supervised learning is het gebruikelijk om tijdens de training de validatie-loss bij te houden. Zodra deze validatie-loss begint toe te nemen, wordt de training gestopt. De gebruikte hyperparameters zijn weergegeven in bijlage 6.7.

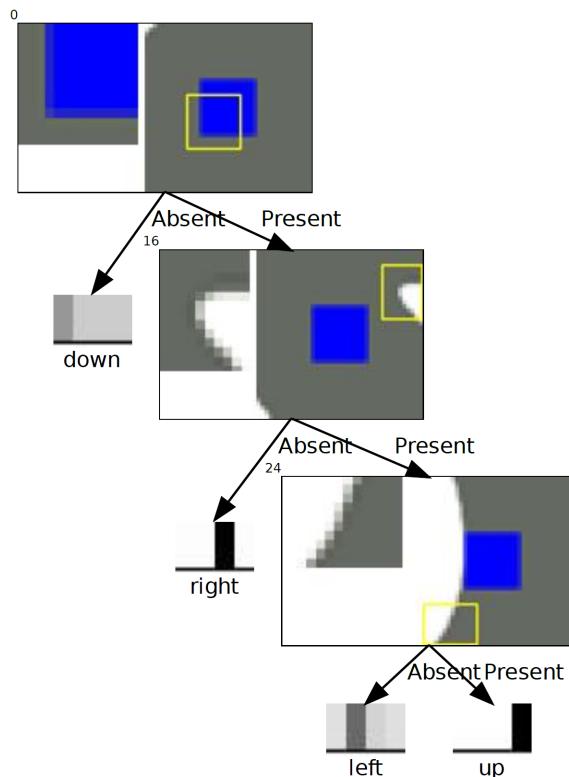
Dit is een mogelijke oplossing voor het stabiliteitsprobleem. In plaats van te blijven trainen, wordt er gestopt wanneer de prestaties drastisch verminderen. Dit in combinatie met het opslaan van het model met een vaste frequentie levert een policy op die in staat is het pad te bewandelen en het doel te bereiken zonder de randen te raken.

### 5.8 Voorstellen van getrainde policy met ProtoTree

Na het verkrijgen van een getrainde agent in de nieuwe omgeving, werd opnieuw een gelabelde dataset aangemaakt op dezelfde werkwijze als toegelicht in sectie 5.1. In figuur 5.11 en figuur 5.12 zijn visualisaties te zien van twee ProtoTrees getraind op omgevingsbeelden gelabeld met acties voorgesteld door de policy. De hyperparameters waarmee de twee bomen zijn getraind, zijn te vinden in bijlage 6.7. De behaalde resultaten zijn behoorlijk goed. In figuur 5.11 wordt er in de wortel gekeken of er zich een deel van het pad onder de agent bevindt. Indien dit niet het geval is, geeft de boom als besluit weer om naar beneden te gaan. Dit komt overeen met de keuze van de getrainde policy, die door het kiezen van het dense-beloningsschema het doel van bovenuit benadert na aflopen van het pad. Wanneer er toch een deel van het pad recht onder de agent te zien is, wordt er overgegaan naar de kindknoop. In deze knoop geeft het prototype een stuk van de weg weer met rechtsonder

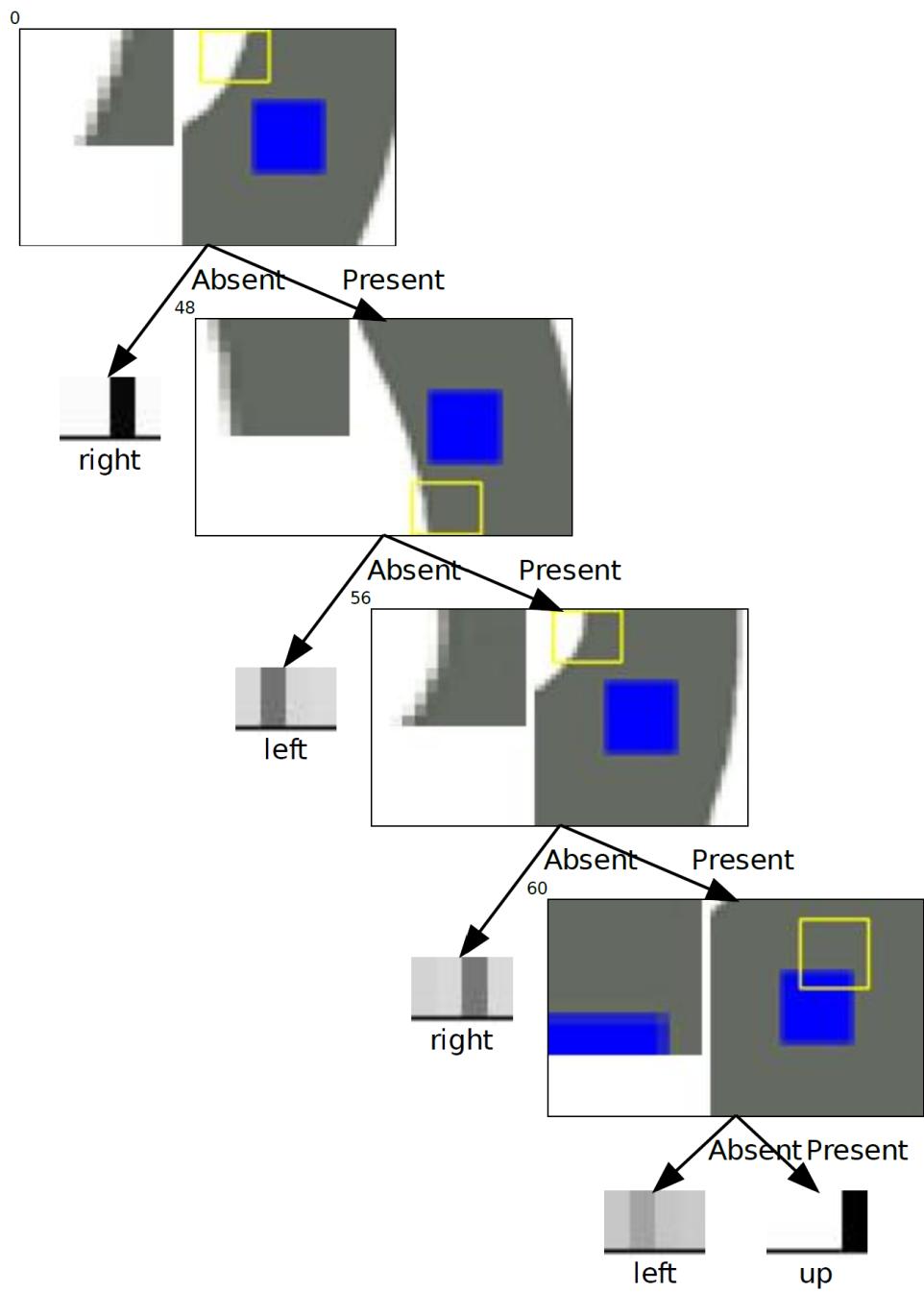
## 5 Voorstellen van policy van DQN-agent

een deel van de witte achtergrond. In de besluitvorming wordt dit veralgemeend naar het kijken of er een deel van de weg te zien is met witte pixels rechts van de grijze weg. Indien dit niet het geval is, besluit de boom om naar rechts te gaan. Tenslotte wordt er in de bladknoop gekeken of er een deel van de weg te zien is met aan de linkerkant de achtergrond, zoja besluit de boom om omhoog te gaan en zoniet, om naar links te gaan. Het enige wat in deze visualisatie ontbreekt, is het kijken naar omhoog. Indien de weg boven de agent vrij is, zou de boom moeten voorstellen om naar boven te gaan. Doordat de boom een maximale diepte heeft meegekregen van 3, heeft hij besloten om dit niet op te nemen ten koste van een van de andere prototypes voorgesteld in de figuur. In figuur 5.12 is de hyperparameter die de diepte bepaalt, op 4 geplaatst en is te zien dat 1 van de prototypes rekening houdt met de pixels boven de agent om te beslissen of de agent naar boven zou moeten bewegen.



Figuur 5.11: Visualisatie van getrainde ProtoTree

## 5 Voorstellen van policy van DQN-agent



Figuur 5.12: Visualisatie van tweede getrainde ProtoTree

# 6

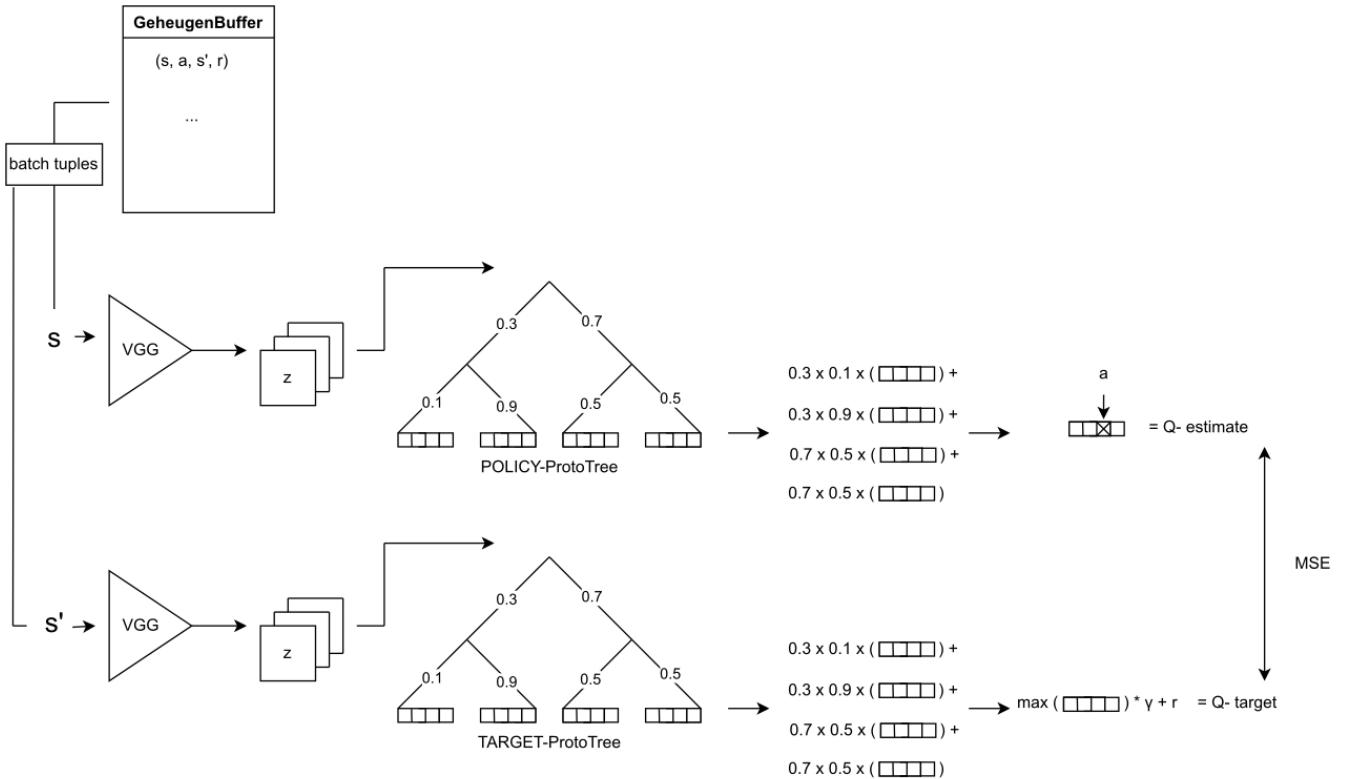
## Combinatie DQN en ProtoTree

Hoofdstuk 5 geeft weer hoe de getrainde policy van een DQN-agent kan voorgesteld worden aan de hand van de ProtoTree. De besproken methode doet dit post-hoc. Omgevingsbeelden worden gelabeld met de actie aangeraden door de getrainde policy van de agent. Na de veelbelovende resultaten van deze supervised-manier van werken, wordt er gekeken wat voor resultaten er verwacht kunnen worden op een volledig RL-georiënteerde wijze. In dit hoofdstuk wordt er besproken hoe de ProtoTree gecombineerd kan worden met het DQN-algoritme. Deze combinatie levert een intrinsiek interpreteerbare voorstelling van de policy voor in de vorm van een beslissingsboom. Eerst wordt er gekeken welke aanpassingen er moeten gebeuren aan de implementatie van de ProtoTree en het DQN-algoritme om deze twee samen te voegen. Daarna wordt er besproken welke implementatiekeuzes en afwegingen er gemaakt moeten worden. Tot slot wordt het behaalde eindresultaat gepresenteerd en besproken.

### 6.1 ProtoTree trainen zonder labels

In hoofdstuk 4 werd het trainingsproces van de ProtoTree besproken. De gewichten van het CNN en de prototypes worden aangeleerd via SGD en het minimaliseren van de cross-entropy loss-functie tussen de voorspellingen en de labels. In een RL-omgeving heeft de ProtoTree echter geen labels tot zijn beschikking maar enkel beloningssignalen. Om te trainen aan de hand van deze beloningen is het optimalisatieproces aangepast. In figuur 6.1 is een visualisatie te zien van deze aangepaste manier van werken. Net zoals bij de optimalisatie van het eenvoudig CNN dat gebruikt werd in hoofdstuk 3, wordt als eerste stap een batch aan transitie-tuples uit de geheugenbuffer opgehaald. De toestanden in de vorm van observatiebeelden worden door de VGG-11-structuur gestuurd om de feature maps  $z$  te bekomen. In hoofdstuk 5 is er onderdervonden dat het onaangepast laten van de gewichten van dit CNN betere resultaten opleverde en dus worden deze hier ook niet geoptimaliseerd. Hierna worden de feature maps opgesplitst in patches van de vorm  $H \times W \times D$  zodat de patches dezelfde vorm hebben als de prototypes. Op elk niveau wordt er gekeken welke patch de grootste overeenkomst heeft met het prototype op basis van een gegeneraliseerde vorm van convolutie om de routering te bepalen. In de implementatie van de ProtoTree voorgesteld door Nauta et al. [10] werd de voorspelling dan bepaald door de kans van aankomst in een blad te vermenigvuldigen met de klasprobabiliteiten, bekomen door de logits te normaliseren aan de hand van de softmax-functie. In deze aangepaste versie wordt het normaliseren aan de hand van softmax weggelaten. De logits worden direct vermenigvuldigd met de kansen van aankomst in de bladeren. Deze waarden worden dan beschouwd als de Q-waarden. Net zoals in de traditionele RL-algoritmen wordt de  $Q_{estimate}$  geselecteerd op basis van de gekozen actie. De formule voor de bepaling van

## 6 Combinatie DQN en ProtoTree



Figuur 6.1: Optimalisatieproces van integratie DQN met ProtoTree

de Q-estimate wordt dan:

$$Q_{estimate} = \sum_{l \in L} c_{l,a} \cdot \pi_l(f(x; \omega)). \quad (6.1)$$

Om het doel te stabiliseren wordt er gebruik gemaakt van een doelnetwerk. Dit is een gekloonde versie van de ProtoTree. De gewichten van de actief geoptimaliseerde ProtoTree worden met een vaste frequentie overgekopiëerd naar het doelnetwerk. De observatiebeelden van de toestand  $s'$ , verkregen na het nemen van de gekozen actie, worden doorheen het doelnetwerk gestuurd. Deze batch van beelden volgt dezelfde stappen tot de bladknopen als  $Q_{estimate}$ . Na de som over alle bladeren wordt de actie gekozen op de index van de grootste waarde in plaats van de actie uit de tuple. Deze waarde wordt dan vermenigvuldigd met de verminderingsfactor  $\gamma$  en gesommeerd met de verkregen beloning zoals weergegeven in onderstaande formule:

$$Q_{target} = r + \gamma \cdot (\max_a \sum_{l \in L} c_l \cdot \pi_l(f(x; \omega))). \quad (6.2)$$

## 6 Combinatie DQN en ProtoTree

De gewichten van de prototypes worden geoptimaliseerd door middel van de MSE loss tussen  $Q_{estimate}$  en  $Q_{target}$ .

### 6.2 Optimalisatie van bladknopen

Zoals besproken in sectie 4.3 gebruikt de ProtoTree een afgeleid vrije manier om de bladknopen te optimaliseren. Omdat deze uitdrukking specifiek is ontworpen voor het optimaliseren van classificatieproblemen, wordt deze in de aanpassing niet gebruikt. Net zoals de prototypes worden de bladknopen geoptimaliseerd via SGD.

### 6.3 Visualisatie aan de hand van de geheugenbuffer

Nadat de ProtoTree getraind is in een supervised-omgeving, wordt er voor een tweede keer over de dataset gelopen om de prototypes te visualiseren. De afbeeldingen doorlopen de volledige structuur, dit keer zonder dat de gewichten van de bladknopen, prototypes en van het CNN geoptimaliseerd worden. De afbeeldingen worden verdeeld in patches en voor elk prototype wordt er bijgehouden met welke patch het prototype de grootste overeenkomst heeft. Dit wordt zoals besproken in hoofdstuk 4, gebruikt om de prototypes weer te geven in de eindvisualisatie.

Het probleem is dat in de RL-omgeving er geen dataset beschikbaar is om over te lopen na het trainingsproces. De afbeelden die gebruikt werden tijdens de modeloptimalisatie komen uit de geheugenbuffer, die beelden van voorgaande toestanden bevat samen met genomen actie, volgende toestand en beloning in de vorm van transitie-tuples. De toestandsruimte van de eigen gecreëerde omgeving is niet extreem groot en bevat ongeveer 150 toestanden. De hyperparameter voor buffergrootte werd tussen de 1000 en 5000 gezet voor het opleidingsproces van de besproken DQN-architectuur uit hoofdstuk 3. Om de prototypes voor te stellen in de aangepaste versie die gebruik maakt van de ProtoTree is het mogelijk om over alle transities in de geheugenbuffer te lopen na het trainingsproces en de toestanden s te gebruiken om de prototypes voor te stellen.

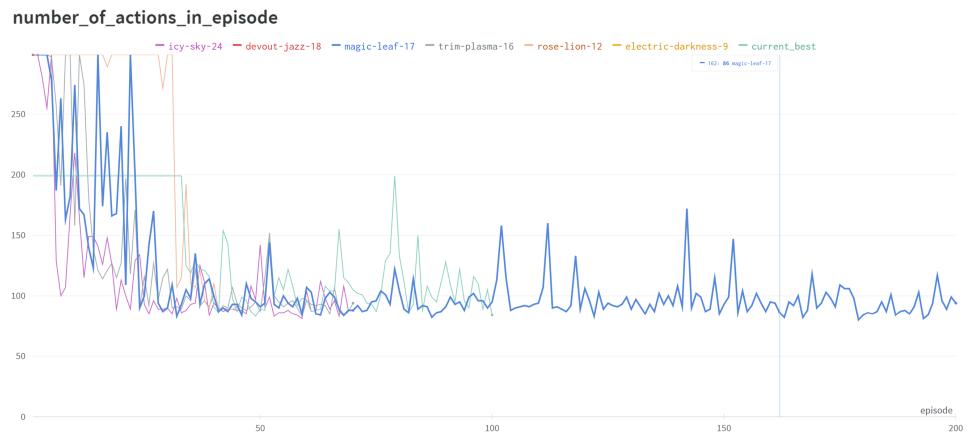
In omgevingen waarbij de toestandsruimte vele malen groter is, is het niet haalbaar om alle transities in de geheugenbuffer te overlopen. Een andere manier om bij deze omgevingen de prototypes voor te stellen, is om na het trainen de policy in de vorm van de gewichten in de ProtoTree te gebruiken om de agent doorheen het pad te laten lopen. Deze omgevingsbeelden kunnen dan gebruikt worden om nadien de prototypes te visualiseren. Om de exploitatie te verbeteren tijdens het trainen van RL-systeem wordt soms ook wel gebruik gemaakt van een geprioriteerde geheugenbuffer. Dit werd voor het eerst voorgesteld in de paper "Prioritized Experience Replay"[29]. In deze paper wordt een manier besproken om ervaringen te prioriteren, zodat belangrijke overgangen vaker worden herhaald en er efficiënter kan worden geleerd. Deze aanpak zou ook een oplossing kunnen zijn voor het voorstellen van de prototypes. Aangezien de prototypes een latente voorstelling vormen van de belangrijkste punten in de omgevingsbeelden is de kans zeer groot dat deze overeenkomen met patches van beelden uit transities met hoge prioriteit.

### 6.4 Resultaten in verband met de policy

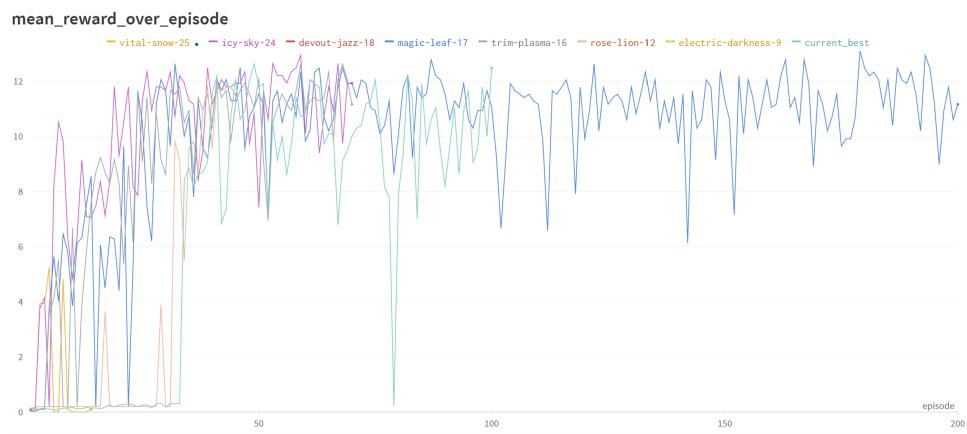
Na de aanpassingen aan de ProtoTree in verband met de dataset en optimalisatie van de parameters, besproken in de voorgaande secties, werd het trainingsproces in actie gezet. De agent maakt gebruik van hetzelfde algoritme 25 als besproken

## 6 Combinatie DQN en ProtoTree

in hoofdstuk 1. Het enige verschil is dat het policy-netwerk en het doelnetwerk vervangen zijn door ProtoTrees. In figuur 6.2 is een plot te zien van het aantal acties dat de agent neemt elke episode van getrainde agenten met verschillende hyperparameters van de ProtoTree. Op het diagram is te zien dat de agent er telkens in slaagt om op basis van de ProtoTree het doel te bereiken en een policy te leren die een vast aantal episodes het doel kan bereiken in minder dan 160 acties. In tegenstelling tot de resultaten van het trainen van het DQN met het eenvoudige CNN, vindt er geen policy collapse plaats en convergeert het trainingsproces rond de 95 acties per episode. In afbeelding 6.3 is een grafiek te zien die de gemiddelde beloning over een episode weergeeft in het trainingsproces. Ook deze grafiek is veel stabieler dan de grafiek geproduceerd met het DQN-algoritme dat gebruik maakt van het eenvoudige CNN.



Figuur 6.2: Aantal acties per episode in het trainingsproces aan de hand van de ProtoTree



Figuur 6.3: gemiddelde beloning over een episode genomen in het trainingsproces aan de hand van de ProtoTree

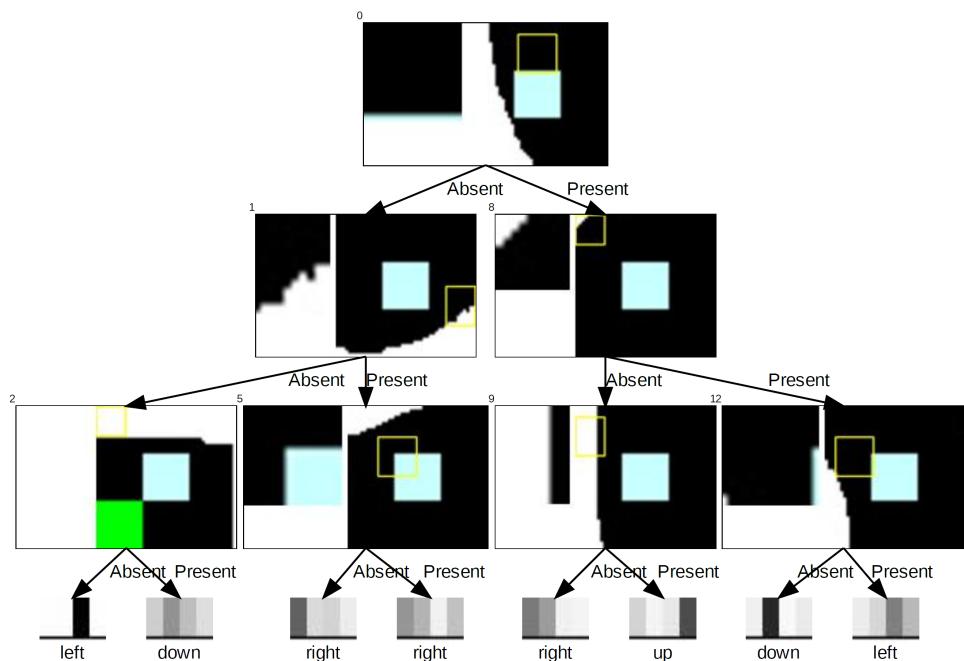
## 6 Combinatie DQN en ProtoTree

De diagrammen geven duidelijk weer dat de agent in staat is om de optimale policy te vinden aan de hand van de ProtoTree.

### 6.5 Resultaten in verband met de visuele voorstelling

Na het met succes opleiden van de agent, werd er gekeken naar de visuele voorstelling van de policy in de vorm van de policy-ProtoTree. In figuur 6.4 is een voorstelling van een getrainde policy te zien. In de paper van de ProtoTree werd voor betere prestaties aangeraden om ervoor te zorgen dat het aantal bladknopen licht groter is dan het aantal klassen. In de visuele voorstelling is duidelijk te zien dat de agent typerende prototypes selecteert. Het prototype in knoop 0 geeft de bovenste zijde van de agent weer met daarboven zwart deel van het pad. Prototypes 1, 8, 9 en 12 kijken of er randen van het pad te zien zijn langs bepaalde kanten. De besluitvorming in de bladknopen is ook hier echter soms kort door de bocht, incorrect of onbesluitloos.

Aangezien de agent in deze omgeving het pad van onder naar boven leert bewandelen zonder de muren te raken, wordt er zelden tot nooit gekozen om naar beneden te gaan en is de gekozen diepte van maximum 3 knopen te diep. Omwille van dit feit is de omgeving aangepast zodat de agent niet meer het doel kan behalen van bovenaf en werd de ProtoTree getraind met een maximale diepte van 2. Dit resulteert in 4 bladknopen. Dit aantal bevat 1 waarde meer dan het aantal klassen, namelijk de verschillende gekozen acties zonder "down".

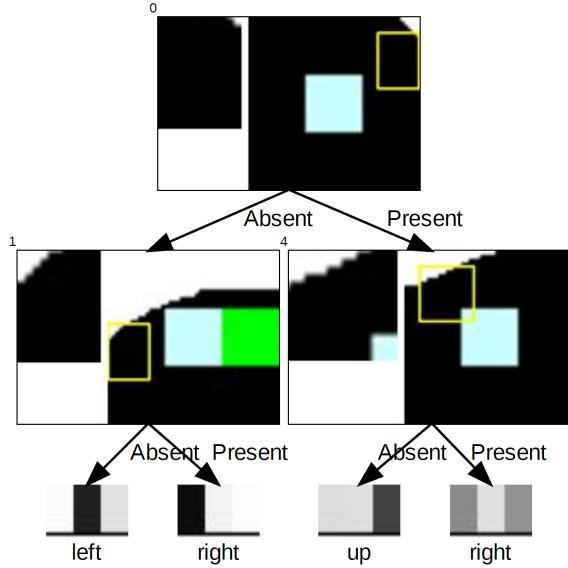


Figuur 6.4: Visuele voorstelling van policy-ProtoTree met maximale diepte 3

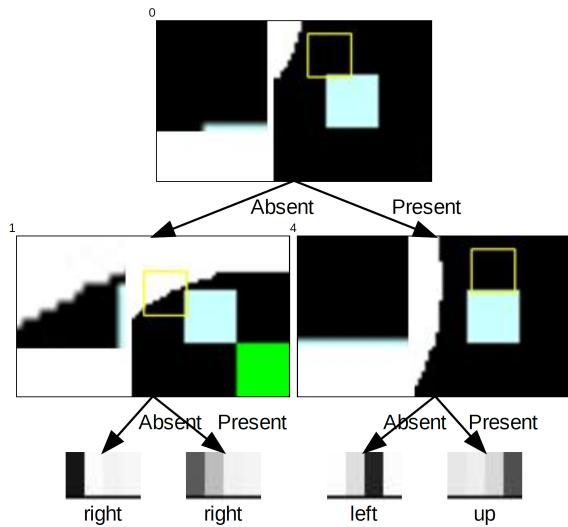
In figuur 6.5 en 6.6 zijn 2 visuele voorstellingen te zien van getrainde DQN-agents met ProtoTrees met een maximale diepte van 2 knopen. De gebruikte hyperparameters zijn te vinden in bijlage 6.7. De wortelknoop in afbeelding 6.5 kijkt of er een deel van de rechterraan van de weg te zien is. Prototype twee kijkt of er een linkerdeel van de weg te zien is in de vorm van een witte driehoek. Dit geeft een scherpe bocht naar rechts weer. Indien er geen overeenkomst is met dit prototype besluit de boom dat de actie "left" gekozen moet worden. In het andere geval opteert de boom voor de actie naar rechts. In

## 6 Combinatie DQN en ProtoTree

de rechterbladknoop kijkt de boom of er een deel van de rand boven zich te zien is alvorens te beslissen om naar boven of naar rechts te gaan.



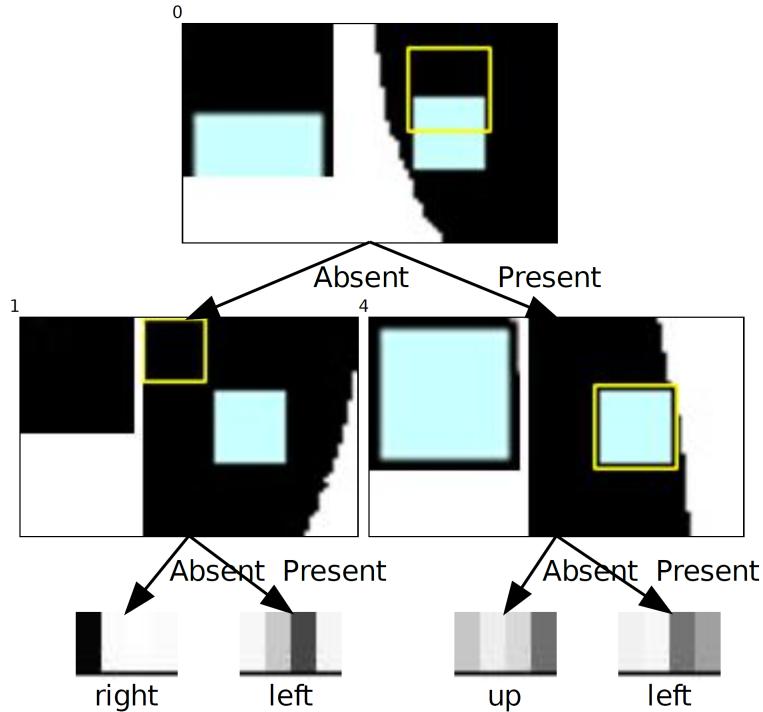
Figuur 6.5: Visuele voorstelling van eerste policy-ProtoTree met maximale diepte 2



Figuur 6.6: Visuele voorstelling van tweede policy-ProtoTree met maximale diepte 2

De visualisatie van de policy met een maximale diepte van 2 komt meer overeen met de besluitvorming die een mens zou maken op basis van zijn/haar visuele perceptie. In verband met de optimale grootte van de prototypes is ondervonden dat de beste resultaten verkregen worden indien de *bounding box* na het visualiseren dezelfde afmeting heeft als de agent of iets kleiner. Wanneer deze *bounding box* groter wordt genomen dan de afmeting van de agent, neemt de kans enorm toe dat de geselecteerde prototypes nietszeggende informatie bevatten, zoals een beeld met alleen de agent of volledig zwarte beelden (zie figuur 6.7).

## 6 Combinatie DQN en ProtoTree



Figuur 6.7: Visuele voorstelling van policy-ProtoTree waarvan de visualisatie van de prototypes groter zijn dan de afmeting van de agent.

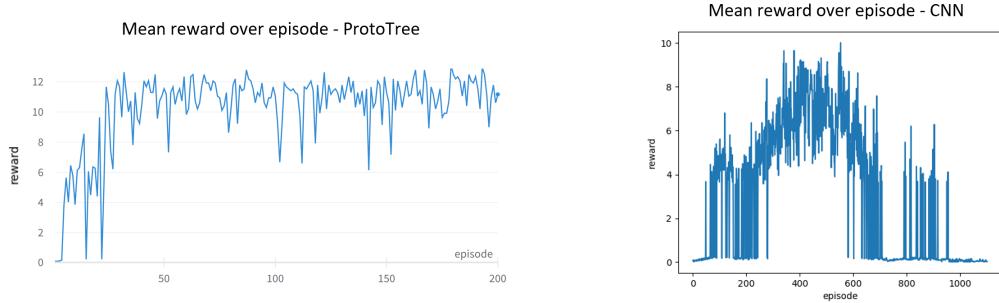
### 6.6 Vergelijking DQN met ProtoTree en eenvoudig CNN

In figuur 6.8 is het verschil te zien tussen de gemiddelde beloning over een episode behaald door de agent die getraind wordt met het DQN dat gebruik maakt van de ProtoTree en de agent met het DQN dat gebruik maakt van het eenvoudig CNN. De policy van de agent met de ProtoTree slaagt er in om na een 30 tal episodes het doel vast te bereiken (zie figuur 6.9). Hierna stijgt de gemiddelde beloning tot het optimale pad gevonden is. Dit punt ligt meestal rond de 50 episodes. Nadat het optimale pad bereikt is, houdt de policy de keuze vast. De exploratiefactor zorgt er soms voor dat de gemiddelde beloning iets lager ligt, maar zorgt er niet voor dat de agent het doel niet meer kan bereiken of voor een daling over de volgende episodes. De beloning blijft stabiel.

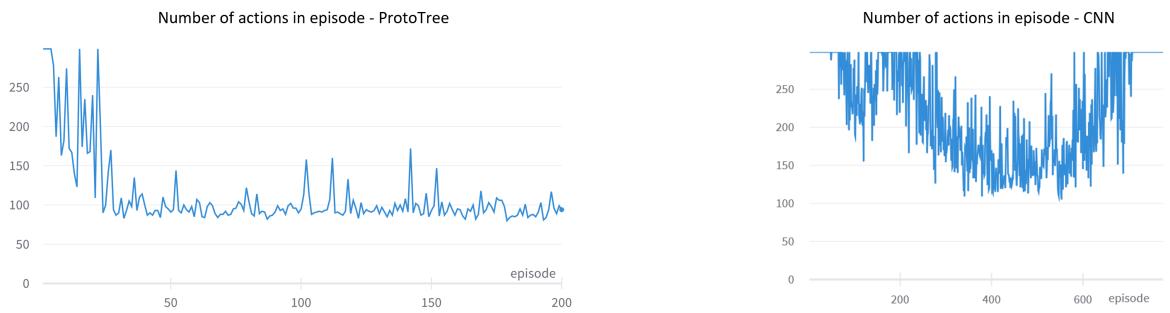
Zoals reeds vermeld in hoofdstuk 5, slaagt de policy van de agent met het eenvoudige CNN er in om na een aantal episodes het doel een periode continu te bereiken. Na dit moment verbetert de policy en stijgt de gemiddelde beloning. Dit gebeurt tot op een bepaald moment, waarna de policy helemaal afzwakt. Deze policy collapse zorgt ervoor dat de agent zelfs niet meer het doel bereikt en sommige episodes een gemiddelde beloning van 0 behaald, wat betekent dat de agent de hele episodes vastgezeten heeft aan de rand van het pad.

Over het algemeen kan verondersteld worden dat de functiebenadering voor de Q-waarden bij gebruik van de ProtoTree zorgt voor een veel stabieler leerproces dan wanneer de Q-waarden worden benaderd met het eenvoudig CNN. Het DQN gebruik makende van het CNN is ook meer gevoelig aan de keuze van de initiële gewichtverdeling. Het DQN met de ProtoTree heeft geen last van deze keuze en convergeert steeds, ongeacht de gekozen gewichtverdeling.

## 6 Combinatie DQN en ProtoTree



Figuur 6.8: Vergelijking van gemiddelde beloning over een episode genomen tussen DQN dat gebruik maakt van eenvoudig CNN en DQN dat gebruik maakt van ProtoTree



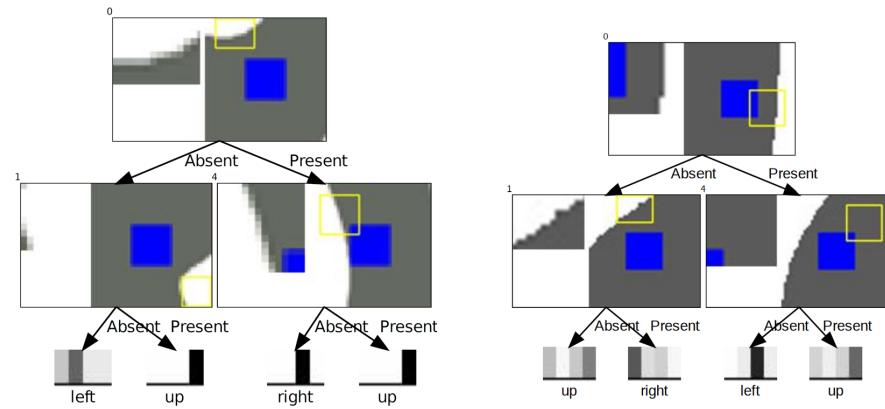
Figuur 6.9: Vergelijking van het aantal acties in een episode tussen DQN dat gebruik maakt van eenvoudig CNN en DQN dat gebruik maakt van ProtoTree

## 6.7 Vergelijking supervised trainen van ProtoTree en trainen op basis van RL-methode

Figuur 6.10 links is een visuele voorstelling van een ProtoTree die supervised getraind is en rechts een visuele voorstelling van een ProtoTree die getraind is aan de hand van de aangepaste RL-methode. De gekozen prototypes na visualisatie op beide manieren zijn meestal zeer gelijkaardig. De geselecteerde prototypes bevatten delen die een stuk van de agent bevatten en de weg of randen van de weg. De bladknopen van de ProtoTree getraind aan de hand van de supervised-methode zijn echter wel meer eenduidig naar een specifieke actie. Desondanks dit zijn de prestaties zeer gelijkaardig. In tegenstelling tot de ProtoTree die getraind wordt met de Q-waarden, ziet de supervised-boom enkel afbeeldingen die deel uitmaken van

## 6 Combinatie DQN en ProtoTree

het optimale pad. De ProtoTree uit de DQN-integratie krijgt telkens een willekeurige batch van staten die bezocht zijn in het verleden en krijgt dus ook delen te zien die niet deel uitmaken van het optimale pad.



Figuur 6.10: Links: visuele voorstelling van supervised trainen van ProtoTree, rechts: visuele voorstelling van trainen van ProtoTree met RL-methode

# Toekomstig werk

De huidige implementatie van de integratie van de DQN en de ProtoTree gebruikt telkens een enkel omgevingsbeeld om een actie te voorspellen. Dit laat toe om problemen op te lossen waar een agent los van snelheid en beweging een actie moet voorspellen. De agent is in staat problemen op te lossen waarvan de moeilijkheidsgraad zit in het beredeneren van een situatie onafhankelijk van tijd. Veel omgevingen en problemen hebben nood aan kort reactievermogen en snel inspringen bij specifieke bewegingen. Om deze problemen te visualiseren aan de hand van de ProtoTree moet de structuur worden uitgebreid zodat er geen actie gekozen wordt op basis van een enkele frame, maar het verschil tussen meerdere beelden. In deze aanpassing moet er gezocht worden naar een manier om dit verschil te behandelen aan de hand van prototypes.

Zoals vermeld in hoofdstuk 4, behaalt de ProtoTree de beste resultaten indien de blad-logits getraind worden aan de hand van een afgeleid-vrije optimalisatie. De manier voorgesteld door Nauta et al. voor deze optimalisatie geldt als volgt:

$$c_{i,j}^{(t+1)} = \sum_{(x,y) \in T} (\sigma(c_l^{(t)}) \odot y \odot \pi_l) \oslash \hat{y}. \quad (6.3)$$

Deze uitdrukking is specifiek gericht naar classificatieproblemen. In de integratie tussen het DQN en de ProtoTree is er geen gebruik gemaakt van deze uitdrukking. De blad-logits worden net zoals de prototypes geoptimaliseerd aan de hand van SGD. In een toekomstig werk kan er onderzocht worden of deze afgeleid-vrije manier toch kan worden omgezet naar een eerder regressieprobleem op basis van de Q-waarden.

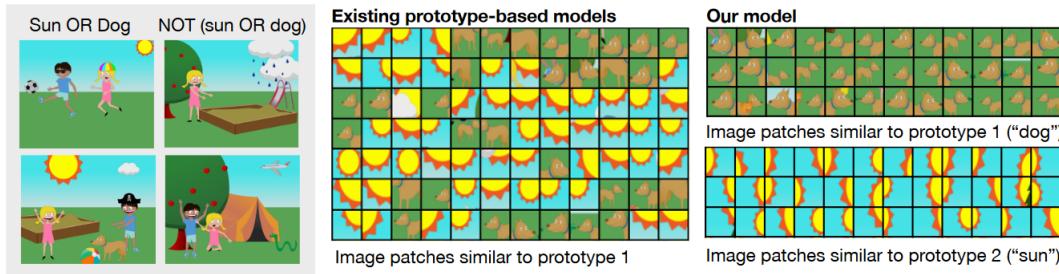
Wat ook opgemerkt moet worden, is dat na verder onderzoek op de ProtoTree en meer specifiek het concept van prototypes, er ook stevige kritiek op de concepten verschenen is. In de papers "Leveraging explanations in interactive machine learning: An overview"[30] en "HIVE: Evaluating the Human Interpretability of Visual Explanations"[31] werd vastgesteld dat de prototypes die getraind worden in de latente ruimte zorgen voor betere interpretatie van de genomen keuzes, zelfs voor onjuiste voorspellingen, maar niet duidelijk genoeg zijn voor gebruikers om onderscheid te maken tussen juiste en onjuiste voorspellingen. In een volgend werk van Nauta et al. "PIP-Net: Patch-Based Intuitive Prototypes for Interpretable Image Classification"[11] stemden Nauta et al. toe met deze kritieken en gaven hier de volgende reactie op: "De eerder geïntroduceerd prototype- modellen, waaronder ProtoTree uit Hoofdstuk 4, kunnen prototypes leren die niet in overeenstemming zijn met menselijke visuele perceptie. Dat wil zeggen dat hetzelfde prototype kan verwijzen naar verschillende concepten in de echte wereld waardoor interpretatie niet intuïtief is." [11]

Dit citaat werd verder verduidelijkt aan de hand van afbeelding 6.11 [11]. Links in de afbeelding is de probleemstelling van het eenvoudige voorbeeld te zien. Het model leert 2 situaties classificeren. De eerste klasse bestaat uit afbeeldingen waarbij de zon of een hond te zien is. Bij de tweede klasse is noch een hond, noch de zon te zien. Het optimalisatieprocess zoals besproken in hoofdstuk 4 zorgt ervoor dat delen van afbeeldingen van de zon en delen van afbeeldingen met een hond prototype 1 activeren zoals te zien in het midden van figuur 6.11. Dit is duidelijk zeer verschillend met de besluitvorming die een mens zou maken. Op basis van het visueel beeld in pixel-ruimte beslist een mens bijvoorbeeld gebaseerd op het feit dat een deel van een hond gezien wordt. Dit kan voorgesteld worden als het activeren van als het ware een prototype in het brein wanneer beelden van een hond waargenomen worden.

In een vervolg op de integratie van het DQN met de ProtoTree kan onderzocht worden hoe de structuur kan aangepast worden

## 6 toekomstig werk

zodat de vorming van de prototypes meer in lijn liggen met het besluit van de visuele perceptie van een mens.



Figuur 6.11: Verduidelijking van verschil in alignatie tussen de geleerde prototypes van bijvoorbeeld de ProtoTree en de prototypes geleerd in PIP-Net, die meer overeenkomen met keuzes die een mens zou maken. [11]

In deze thesis is vooral gekeken hoe de ProtoTree geïntegreerd kon worden in het DQN-algoritme voorgesteld door Mnih et al. [16]. Sinds de publicatie van dit werk heeft het onderzoek niet stil gestaan. In de paper "Rainbow: Combining Improvements in Deep Reinforcement Learning" [32] werden verschillende verbeteringen van het traditionele DQN-algoritme vergeleken en geëvalueerd. Door de beperkte tijdspanne van de thesis zijn deze verschillende verbeterde algoritmen niet uitgetest en aangepast met de ProtoTree. In een vervolgstuk zou kunnen gekeken worden wat het gebruik van deze algoritmen met inwendige ProtoTree te weeg brengt. Buiten de visuele voorstelling van een policy van een DQN of verwante agent zou ook gekeken kunnen worden hoe de ProtoTree kan gecombineerd worden met andere *state-of-the-art* RL-modellen zoals Proximal Policy Optimization (PPO) [33] of Asynchronous Advantage Actor-Critic (A3C) [34].

Ook is de structuur getest op de eigen ontworpen omgeving waarbij de agent in de vorm van een blauw vierkant doorheen een pad moet navigeren om het doel te bereiken om de besluitvorming in kaart te brengen. Dit liet toe om de beslissingen van de policy te visualiseren zonder dat er extreem hoge trainingstijden of GPU vereisten nodig waren. In de paper "Playing Atari with Deep Reinforcement Learning" [16] werden de agenten in de verschillende omgevingen telkens getraind met 10 miljoen beelden en een geheugenbuffer die een miljoen aan de meest recente beelden bevat wat niet echt haalbaar was voor deze thesis. In toekomstig werk zou de structuur getest kunnen worden op meer complexe omgevingen.

# Conclusie

In deze thesis is onderzocht hoe de policy van een DRL-algoritme kan voorgesteld worden aan de hand van meer verklarende structuren zoals beslissingsbomen. Na het bestuderen van de architecturen van een DQN en de Neural Prototype Tree, kortweg ProtoTree, is een succesvolle integratie verwezenlijkt. De integratie slaagt erin om een policy te trainen die in staat is een agent in de vorm van een groen vierkant, in een minimum aantal stappen het doel in de vorm van een groen vierkant te doen bereiken, zonder van het pad te gaan. De integratie maakt intern gebruik van de ProtoTree om Q-waarden te voorspellen. Door inwendig gebruik te maken van de ProtoTree, wordt er een globaal interpreteerbare policy verkregen na het trainingsproces. In tegenstelling tot post-hoc methoden waarbij de keuzes van een policy na het trainen worden gevisualiseerd, levert dit onmiddellijk inzicht in de keuzes die de policy zal nemen wanneer ze ingezet wordt. Deze globale interpreteerbaarheid verhoogt het menselijk vertrouwen in de getrainde DQN-instanties. Het geeft direct inzicht in de redenen waarom een actie gekozen wordt door een bepaalde agent. Op deze manier kan er duidelijk gezien worden wanneer de agent effectief beslissingen maakt op basis van correcte veronderstellingen of simpelweg zaken van buiten aan het leren is en complexe verbanden aan het leggen is. De integratie kan ingezet worden op omgevingen die geen snel reactievermogen nodig hebben en waarvan de actie kan worden afgeleid op basis van een enkel beeld. De integratie toepassen op deze omgevingen zorgt voor een duidelijk inzicht en bevat het vermogen om het vereiste doel tot een correct eind te laten verlopen.

## Ethische en maatschappelijke reflectie

De verhoogde interpreteerbaarheid, transparantheid en verklaarbaarheid die de ProtoTree met zich mee brengt in de integratie met DQN-agenten en andere RL-agenten kan voor een vooruitgang zorgen op het gebied van industrie, innovatie en infrastructuur (SDG 9). Vandaag de dag wordt reinforcement learning nog steeds zeer weinig gebruikt in het bedrijfsleven. Ondanks de vele voordelen en de progressie in het domein blijft verklaarbaarheid één van de hoofdredenen waarom er toch wordt weggebleven van RL. De globale verklaarbaarheid die de ProtoTree met zich meebrengt in de integratie zorgt ervoor dat beheerders van machines en applicaties in de industrie onmiddellijk inzicht hebben op de keuzes die de getrainde policy neemt. De visuele voorstelling aan de hand van prototypes geselecteerd uit omgevingsbeelden geven de gebruikers meer vertrouwen in de getrainde agenten. Ook zorgt het ervoor dat het uitrollen van deze instanties veel sneller kan gebeuren en dat er bij systeemfouten of policy-vergissingen veel sneller kan ingegrepen worden.

Desondanks deze voordelen moet de structuur toch met een korrel zout genomen worden en moet er voorzichtig mee omgesprongen worden. Doordat de prototypes opgesteld en geoptimaliseerd worden in latente ruimte en daarna worden omgezet aan de hand van omgevingsbeelden naar pixel-ruimte, liggen de geselecteerde prototypes niet volledig in lijn met het gezichtsvermogen van mensen. Dit kan ervoor zorgen dat prototypes geactiveerd worden door verschillende opvattingen. Deze verschillende opvattingen leiden tot een hoog risico in delicate omgevingen zoals zelfrijdende auto's en robotica in dienstverlening en gezondheidszorg. De visuele voorstelling aan de hand van een bounding box rond het deel van een observatiebeeld levert een beeld van het concept dat het meest overeenkomt met het prototype. De mogelijkheid tot activatie van concepten die voor het menselijk oog zeer verschillend zijn, brengt echter een potentieel gevaar met zich mee waar zeker rekening mee gehouden moet worden. Naast sommige onbedoelde voorvalen kunnen personen met malafide intenties hier misbruik van maken door deze onderliggende concepten te achterhalen of gelijkaardige concepten na te bootsen en zo de

## 6 Conclusie

bedoelde werking van systemen te dwarsbomen.

Deze opmerkingen nemen niet weg dat de integratie voor een grote verbetering zorgt in interpreteerbaarheid, verklaarbaarheid en transparantie van RL-systemen. De verkregen visuele voorstelling van de policy levert een inzicht in de keuze van een bepaalde actie zonder een grote afweging in performantie van de getrainde agent en is zeker een verbetering ten opzichte van de typische black box-natuur gebruik makende van eenvoudige CNN's.

# Referenties

- [1] A. Plaat, *Deep Reinforcement Learning, a textbook*. [Online]. Available: <http://arxiv.org/abs/2201.02135>
- [2] 40 resources to completely master markov decision processes. [Online]. Available: <https://dev.to/rodolfomendes/40-resources-to-completely-master-markov-decision-processes-49o3>
- [3] Reinforcement learning maze solver | the JetBrains academy blog. [Online]. Available: <https://blog.jetbrains.com/education/2023/05/09/reinforcement-learning-maze-solver/>
- [4] Deep learning playlist overview & machine learning intro. [Online]. Available: <https://deeplizard.com/learn/video/gZmobeGLOYg>
- [5] Google colaboratory. [Online]. Available: [https://colab.research.google.com/drive/1eN33dPVtdPViiS1njTW\\_-r-IYCDTFU7N](https://colab.research.google.com/drive/1eN33dPVtdPViiS1njTW_-r-IYCDTFU7N)
- [6] I. Sajedian, H. Lee, and J. Rho, "Double-deep q-learning to increase the efficiency of metasurface holograms," vol. 9, p. 10899.
- [7] Gym documentation. [Online]. Available: <https://www.gymlibrary.dev/>
- [8] H. Nguyen. Playing mountain car with deep q-learning. [Online]. Available: <https://ha-nguyen-39691.medium.com/playing-mountain-car-with-deep-q-learning-9bdce3715159>
- [9] A. Choudhary. Deep q-learning | an introduction to deep reinforcement learning. [Online]. Available: <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>
- [10] M. Nauta, R. van Bree, and C. Seifert, "Neural prototype trees for interpretable fine-grained image recognition." [Online]. Available: <http://arxiv.org/abs/2012.02046>
- [11] M. Nauta, J. Schlötterer, M. van Keulen, and C. Seifert, "Pip-net: Patch-based intuitive prototypes for interpretable image classification," 2023.
- [12] Announcing the farama foundation - the future of open source reinforcement learning. [Online]. Available: <https://farama.org/Announcing-The-Farama-Foundation>
- [13] Reinforcement learning series intro - syllabus overview. [Online]. Available: <https://deeplizard.com/learn/video/nyjbcRQ-uQ8>
- [14] R. S. Sutton, "Learning to predict by the methods of temporal differences," vol. 3, no. 1, pp. 9–44. [Online]. Available: <http://link.springer.com/10.1007/BF00115009>
- [15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning." [Online]. Available: <http://arxiv.org/abs/1312.5602>
- [17] PyTorch. [Online]. Available: <https://www.pytorch.org>

## 6 Referenties

- [18] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," version: 3. [Online]. Available: <http://arxiv.org/abs/1509.06461>
- [19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." [Online]. Available: <http://arxiv.org/abs/1412.6980>
- [20] J. Johnson. Interpretability vs explainability: The black box of machine learning. [Online]. Available: <https://www.bmc.com/blogs/machine-learning-interpretability-vs-explainability/>
- [21] E. W. Weisstein. Tensor. Publisher: Wolfram Research, Inc. [Online]. Available: <https://mathworld.wolfram.com/>
- [22] C. Chen, O. Li, C. Tao, A. J. Barnett, J. Su, and C. Rudin, "This looks like that: Deep learning for interpretable image recognition." [Online]. Available: <http://arxiv.org/abs/1806.10574>
- [23] H. et al. Papers with code - iNaturalist dataset. [Online]. Available: <https://paperswithcode.com/dataset/inaturalist>
- [24] P. Kotschieder, M. Fiterau, A. Criminisi, and S. R. Bulò, "Deep neural decision forests," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1467–1475, ISSN: 2380-7504.
- [25] About - pygame wiki. [Online]. Available: <https://www.pygame.org/wiki/about>
- [26] A. Irpan, "Deep reinforcement learning doesn't work yet," <https://www.alexirpan.com/2018/02/14/rl-hard.html>, 2018.
- [27] N. Slater. Answer to "what is "policy collapse"and what are the causes?". [Online]. Available: <https://datascience.stackexchange.com/a/22221>
- [28] Sutton & barto book: Reinforcement learning: An introduction. [Online]. Available: <http://incompleteideas.net/sutton/book/the-book-2nd.html>
- [29] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay." [Online]. Available: <http://arxiv.org/abs/1511.05952>
- [30] S. Teso, A. Alkan, W. Stammer, and E. Daly, "Leveraging explanations in interactive machine learning: An overview," vol. 6. [Online]. Available: <https://www.frontiersin.org/articles/10.3389/frai.2023.1066049>
- [31] S. S. Y. Kim, N. Meister, V. V. Ramaswamy, R. Fong, and O. Russakovsky, "HIVE: Evaluating the human interpretability of visual explanations." [Online]. Available: <http://arxiv.org/abs/2112.03184>
- [32] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning." [Online]. Available: <http://arxiv.org/abs/1710.02298>
- [33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms." [Online]. Available: <http://arxiv.org/abs/1707.06347>
- [34] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning." [Online]. Available: <http://arxiv.org/abs/1602.01783>

## **Bijlagen**

## Bijlage A

Hyperparameters gebruikt bij het trainingsproces van de DQN-agent op de Frozen Lake-omgeving.

hyperparameter	waarde
batch size	32
verminderingssfactor ( $\gamma$ )	0.9
$\epsilon$ -startwaarde	1
$\epsilon$ -eindwaarde	0.05
$\epsilon$ -decay	5000
target update	10
learning rate	0.001
weight decay	1e-5
grootte van geheugenbuffer	5000
episodes	4000

Tabel 1: hyperparameters DQN-agent bij Frozen-Lake omgeving

## Bijlage B

Hyperparameters en argumenten gebruikt bij het supervised trainen van de ProtoTree op omgevingsbeelden van eigen omgeving.

hyperparameter/argument	waarde
dataset	'gridpath'
learning rate voor de 1x1 conv-laag en laatste conv-laag van het onderliggende neurale netwerk	0.01
learning rate prototypes	0.01
pretrained CNN	'vgg11'
batch size	64
diepte van ProtoTree	4
epochs	100
optimizer	'AdamW'
W1	1
H1	1
Diepte van de prototype en van convolutionele output	3
milestones	60, 70, 80, 90, 100
gamma voor de MultiStepLR learning rate scheduler	0.5
freeze-epochs	100
upsample-threshold	0.98
disable-pretrained	False
disable-derivative-free-leaf-optim	False
pruning-threshold-leaves	0.4

Tabel 2: hyperparameters en argumenten voor het trainen van ProtoTree op omgevingsbeelden van eigen aangemaakte omgeving

## Bijlage C

Hyperparameters gebruikt bij het trainingsproces van de DQN-agent op de eigen aangemaakte omgeving.

hyperparameter	waarde
batch size	64
verminderingssfactor ( $\gamma$ )	0.95
$\epsilon$ -startwaarde	1
$\epsilon$ -eindwaarde	0.1
$\epsilon$ -decay	50000
target update	30
learning rate	0.0001
weight decay	0
grootte van geheugenbuffer	10000
episodes	1000
reset omgeving	300

Tabel 3: hyperparameters DQN-agent bij eigen aangemaakte omgeving

## Bijlage D

Hyperparameters gebruikt bij het trainingsprocess van de DQN en ProtoTree integratie op de eigen aangemaakte omgeving.

hyperparameter	waarde
batch size	64
verminderingfactor ( $\gamma$ )	0.95
$\epsilon$ -waarde	0.1
target update	25
learning rate voor de 1x1 conv-laag en laatste conv-laag van het onderliggende neurale netwerk	0.01
learning rate prototypes	0.01
learning rate blad-logits	0.001
pruning threshold	0.4
milestones	30,50,60,70
maximale diepte van ProtoTree	2
Diepte van de prototype en van convolutionele output	3
disable-derivative-free-leaf-optim	True
grootte van geheugenbuffer	4800
episodes	60
reset omgeving	300

Tabel 4: hyperparameters DQN-agent gecombineerd met ProtoTree bij eigen aangemaakte omgeving

## Bijlage E

Deze bijlage bevat uitleg over het gebruik van artificiële intelligentie in de masterproef. De code van de ProtoTree voorzien door Nauta et al. is geschreven aan de hand van Pytorch zoals vermeld in de scriptie. Voorheen de masterproef was er nog geen ervaring met dit framework verworven. Na het volgen van verschillende zelfstudie-opdrachten op de officiële pagina van Pytorch werden sommige stukken van de codebase niet helemaal begrepen. Bij deze stukken werd er eerst gekeken naar de documentatie-pagina van Pytorch, maar deze is niet altijd zo uitgebreid en bevat soms weinig voorbeelden. Om de code toch te begrijpen is gebruik gemaakt van ChatGPT. Na het meegeven van een codefragment wordt er in de *prompt* gevraagd om stap voor stap uit te leggen wat het fragment precies doet. Een ander voorbeeld is dat aan ChatGPT een bepaalde functie uit Pytorch werd gegeven en gevraagd werd om met een eenvoudig voorbeeld de functie te illustreren.

In verband met de thesis is ChatGPT nauwelijks gebruikt. Een situatie waarvoor het toch gebruikt werd, is wanneer er werd ondervonden dat een woord repetitief terugkwam. In deze situatie kreeg ChatGPT de vraag naar enkele synoniemen van een bepaald woord. Ook werd er soms gevraagd of een zin grammaticaal correct was of niet.

In verband met programmeerwerk werd er gebruik gemaakt van Visual Studio Code met ondermeer de GitHub CoPilot-extensie. Deze extensie laat vooral toe om herhaalde taken of invulwerk sneller te laten verlopen. Een voorbeeld hiervan is wanneer een functie wordt opgeroepen en de extensie suggesties geeft voor de benodigde parameters. Wanneer er vervolgens op de tab-toets gedrukt wordt, worden deze parameters automatisch ingevuld.