
CHALMERS
UNIVERSITY OF TECHNOLOGY

# Reinforcement learning for ecosystems

## Using explainable dynamic neural networks to train reinforcement learning agents in a simulated virtual animal ecosystem

Master's thesis in Algorithms, Languages and Logic

Felix Hulthén

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

# Reinforcement learning for ecosystems

Using explainable dynamic neural networks to train reinforcement learning agents in a simulated virtual animal ecosystem

Felix Hulthén


CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
*Division of Data Science and AI*
Chalmers University of Technology
Gothenburg, Sweden 2020

Reinforcement learning for ecosystems
Using explainable dynamic neural networks to train reinforcement learning agents in a simulated virtual animal ecosystem
Felix Hulthén



Supervisor: Claes Strannegård, Computer Science and Engineering
Examiner: Marina Axelson-Fisk, Mathematical Sciences

Master's Thesis 2020
Department of Computer Science and Engineering
Division of Data Science and AI
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A screenshot of the developed reinforcement learning environment based on a virtual animal ecosystem.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Reinforcement learning for ecosystems
Using explainable dynamic neural networks to train reinforcement learning agents in a simulated virtual animal ecosystem
Felix Hulthén
Department of Computer Science and Engineering
Chalmers University of Technology

# Abstract

This report covers the development of an inherently explainable and dynamic neural network for reinforcement learning in virtual animal ecosystems, based on the *life-long learning from zero* (LL0) neural network used in supervised learning. The developed network (RLL0) is a fuzzy neural network with specialised growing and pruning rules for an ever changing environment. Results from benchmarking against a reference network show that RLL0 has a comparable performance while using far fewer trainable parameters. This, combined with its adapted architecture for visualising a learnt behaviour, shows promising future extensions to and use of the explored algorithms for animal behavioural based biological simulations.

Keywords: Life-long learning, reinforcement learning, fuzzy neural networks, artificial intelligence, animals

# Acknowledgements

Having had the opportunity to work with artificial intelligence in animal behavioural simulations, a field I find truly fascinating, I would like to express my uttermost gratitude to my supervisor, examiner and the people from Dynamic Topologies Sweden AB for their help in making this project possible.

Felix Hulthén, Gothenburg, June 2020

# Contents

# List of Figures

# List of Tables

# List of Equations

# 1

# Introduction

Problems such as autonomous driving, object recognition and large scale agent based simulations have long presented challenging optimisation tasks for many reasons, with one being performance. In the past years solutions to these problems have become more attainable with the increase of computational power and function optimisation techniques. This has let computers perform complex tasks with methods that do not require explicit programming, such as learning and winning against a professional opponent in the game of Go [1] using machine learning. Incredible feats like this; however, come at the price of network modelling, energy consumption and memory, which themselves are challenging problems.

Networks with dynamic architectures partially combat these issues by adapting its structure during training [2], [3] and by managing their own learning capacity [4]. However, they do not avoid being initially slow to train and neither is their learned behaviour trivial to explain during or even after training. These issues were some of the the problems addressed by the *Life-long learning form zero* (LL0) dynamic networks used in supervised learning [5], [6] It was designed to use far less resources and learn very quickly in classification tasks. With the increasing benefits seen in adapting reinforcement learning for biology simulations [7], having a network like LL0 could make otherwise computationally expensive simulations more efficient.

To use these properties in biology simulations this report proposes the *reinforcement life-long learning from zero* (RLL0) neural network, an adaptation of LL0 for reinforcement learning. It is designed to work seamlessly with existing training algorithms and inherits most of the explainability and resource efficiency of the original. Its performance was tested by benchmarking it against a reference network in a selection of common and new environments, including a custom made environment based on a virtual animal ecosystem. Based on the results it can learn agent policies rivalling the reference network using only a fraction of trainable parameters.

## 1.1 Aim and limitations

The aim of this report is to explore the possibility of adapting and using LL0 in reinforcement learning problems related to animal behaviour in biological simulations. As such the underlying research question for this work was, "Can the dynamic network LL0 be adapted and used for reinforcement learning of virtual animal ecosystem problems"? The purpose behind this report was to test LL0's adaptability and suitability in reinforcement learning, more specifically how its unique properties could be transferred to and used in this more general domain of problems, and whether biological simulations could benefit from it. The actual practical difficulties associated with using the network, such as problem specific parameter tuning and hardware optimisation, were not focused on and therefore not covered by this report.

# 2

# Theory

The following chapter presents reinforcement learning, its most prominent training algorithms and techniques for those unfamiliar with the topic. This is followed by an introduction to fuzzy neural networks and finally the original LL0 neural network as used in supervised learning in section 2.11.

## 2.1 Reinforcement learning

The benefits of artificial intelligence and therein machine learning is its ability for computer programs to learn a difficult function optimisation without human intervention. To do this an algorithm is trained on data samples to (ideally) learn the global minimum or maximum. However, what data is available and which assumptions can be made depend on what type of training is used, with two common being *supervised learning* and its more general form *reinforcement learning* [8].

The goal of supervised learning is to minimise the prediction error of a mapping function between input and desired output. This function is learnt by training on mapping samples in a labelled dataset. However, in practice these labelled datasets are most often not available or simply not an option in terms of time and cost. This is what makes reinforcement learning so appealing. It does not need labels to learn. Instead, it learns form the response of an external reward signal after performing a mapping, much like how an animal learns to perform a task by observing the response of its environment.

A problem in reinforcement learning is represented as a Markov Decision Problem. This combines a set of states $\mathbb{S}$ and a set of actions $\mathbb{A}$ that can be performed by an agent at a state. During each step an action is selected by the agent based only on its current state $s \in \mathbb{S}$, causing it to transition to a new state according the matrix $\mathbb{P}$. This process repeats until the agent reaches a terminal state.

In reinforcement learning this interaction is represented by two entities, an *agent* subject to learning and its *environment*, as seen in figure 2.1. At each time step $t \geq 0$ the agent is given an observation vector $s_t$ by the environment and selects an action $a_t$ according to its policy function $\pi : \mathbb{S} \to \mathbb{A}$. After each interaction the environment transitions to a new state and produces a reward $r_t$ based on its reward function $\mathbb{R}$. From continued interaction the agent is tasked with learning how to maximise the cumulative reward ,called *return*, that is the sum of the agent's received reward throughout its environment trajectory. To control how an agent perceives long and short term rewards, an additional discount factor $\gamma \in [0, 1]$ is used to form the *discounted return*. Acting optimally based on this means learning an optimal policy $\pi^*$. Such a policy can be formed based on the optimal *expected discounted return* of each state, called the value function $V(s; \mathbb{R}, \mathbb{P})$ and always pick the action that maximises the expected return of the next step.

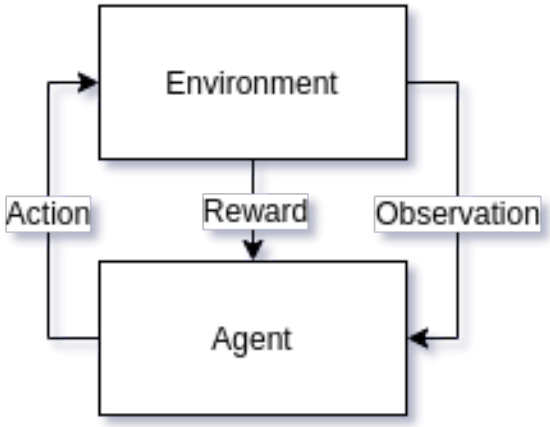


**Figure 2.1:** The general representation of the reinforcement learning process. At each time step the agent receives an observation vector from the environment and is task with maximising its return by selecting an appropriate action.

By learning from experience, a successful agent finds what actions are optimal given an observation. In the case that all environment parameters are known, an optimal policy can be found through a *model-based* method such as value iteration. However, in a practical setting it is almost never the case that $\mathbb{R}$ and $\mathbb{P}$ are known, forcing the agent to instead learn a *model-free* policy.

As it is not possible to directly infer $V(s; \mathbb{R}, \mathbb{P})$ in model-free learning, an alternative is to learn the Q-value function $Q(a, s|A_t = a, S_t = s)$, which does not depend on the environment's inner dynamics. Here the agent only needs to map action and state pairs to a discounted return estimate. Acting optimally then means fulfilling the Bellman optimality equation in equation 2.1 and always picks the action with the highest quality estimate.

$$Q_*(a, s) = \mathbb{E}[r + \gamma \max_{a'} Q_*(s', a')] \tag{2.1}$$

**Equation 2.1:** The goal of the agent is to arrive at a solution to the Bellman optimally equation. With it an optimal policy can be formed by acting greedy based on the action quality values.

However, in most cases the observations given to an agent do not reflect the entire environment state. As such it is not generally possible to find a truly optimal policy. Instead, reinforcement learning agents are more realistically able to find sub-optimal policies. The issue then becomes how fast the agent is able to achieve this during training.

## 2.2 Monte Carlo training

The *Monte Carlo* (MC) training algorithm is based on stochastic interaction with the environment. At the end of an episode the agent's experienced trajectory is used to form return estimates of states $\hat{g}_t$ along the path. The Q-value estimates for actions of these states on the trajectory are then slightly updated towards the new return estimates on a *fist-visit* or *every-visit* basis.

Because this training algorithm uses all states along an agent trajectory it requires the learnt problems to be episodic. In practice these MC update targets have a large variance due to the experienced trajectories generally only reflecting a fraction of the possible environment trajectories. However, if the estimated returns are accurate it leads to fast learning.

## 2.3 Q-learning and neural networks

A more stable algorithm is the *temporal difference* (TD) algorithm. In this report we consider only the TD(1), a *Q-learning* algorithm that looks one step ahead. This training algorithm updates estimates for $Q(a_t, s_t)$ based on the reward $r_t$ received after performing an action $a_t$ in state $s_t$ and transitioning to the next state $s_{t+1}$, see equation 2.2 which scales the update with $\alpha$ as a learning rate.

$$Q(a_t, s_t) \leftarrow \alpha[r_t + \gamma \max_{a'}(Q(a', s_{t+1})) - Q(a_t, s_t)] \tag{2.2}$$

**Equation 2.2:** Q-learning updates the estimate slightly towards the experienced reward $r_t$ at time step $t$ and the maximum estimate of the next state $s_{t+1}$.

To efficiently work with the often massive input and output dimensions of practical problems, neural networks are commonly used as a Q-value estimation function. In an neural network a single update generally affects the estimates of many states, making it possible to generalise to never before seen states after training. A popular implementation is the *Deep Q-learning Network* (DQN) [9]. It uses *Stochastic Gradient Decent* (SGD) to update network parameters, a method that alters their values by small aggregated steps in the reverse gradient direction of a loss function.

The presented DQN uses off-policy training that records transition experiences in a buffer collected during training according to a *behavioural* policy. This policy uses a set of network parameters $\theta$ that are updated towards a target produced by a *target* policy with parameters $\theta'$. The two policies are synced every so often, which gives the network more stable targets than if it had used the same policy for both behaviour and target, called on-policy training.

The network targets can; however, be further improved with *Double DQN* (DDQN) [10], which turns the DQN target in equation 2.3 into the DDQN target in equation 2.4. This change avoids the agent learning too optimistic estimates. The behavioural and target policies periodically switched their roles and therefore alternate between being the target and being trained.

$$Q(a_t, s_t; \theta) \leftarrow \alpha[r_t + \gamma Q(\max_a Q(a_t, s_{t+1}; \theta'), s_{t+1}; \theta') - Q(a_t, s_t; \theta)] \tag{2.3}$$

**Equation 2.3:** The original update function used by DQN that learnt too optimistic Q-values.

$$Q(a_t, s_t; \theta) \leftarrow \alpha[r_t + \gamma Q(\max_a Q(a_t, s_{t+1}; \theta), s_{t+1}; \theta') - Q(a_t, s_t; \theta)] \tag{2.4}$$

**Equation 2.4:** In the DDQN the target is changed to be based on the next state action of the behavioural network, rather than the target network as used in the DQN update.

## 2.4 Proximal Policy Optimisation

Instead of learning Q-values as in MC and TD to later use them in an agent policy, another approach is to directly learn and update a policy. This is what is done in a *policy-gradient* algorithm such as *Proximal Policy Optimisation* (PPO) [11]. This algorithm combines policy-gradient networks with *trust regions* to limit the destructive impact of network updates. It also uses an *actor-critic* to estimate the value of being in a state, which is used to update the action distribution produced by the network.

The PPO algorithm learns these estimates and distributions by recording the agent's environment trajectory in an experience buffer, which is later used for updating in an on-policy fashion. Once the buffer is full the critic value estimates form advantage values, the difference between the predicted and the buffered return estimate. The advantages are used to update the policy to favour the actions that were associated with a positive advantage and discourage those that were negative. The updates are; however, limited by a change ratio $\tau$ to prevent an update from devastating the network performance.

## 2.5 Adaptive moment estimation

When training neural networks the rate at which parameters are updated is important for efficient training. A single learning rate for all parameters could benefit some, yet be devastating for other parameters. If too small the agent will learn very slowly or even get stuck in a local optima and if instead a too large it could prevent the agent from making any fine tuned adjustments. *Adaptive moment estimate* (ADAM) assigns an individual adaptive learning rate to each network parameter based on the first and second moment estimates $m_i$ and $v_i$ [12]. These estimates speed up or slow down changes made to a parameter depending on the observed magnitude and direction of change, see equation 2.5. The rate also regard the passing of time using the decay factors $\beta_1$ and $\beta_2 \in [0, 1)$ to make smaller fine tuned changes at later time steps.

$$
\begin{aligned}
m_{t,i} &\leftarrow \beta_1 \cdot m_{t-1,i} + (1 - \beta_1) \cdot \nabla_\theta(\theta_{\mathbf{t-1}}) \\
v_{t,i} &\leftarrow \beta_2 \cdot v_{t-1,i} + (1 - \beta_2) \cdot \nabla_\theta(\theta_{\mathbf{t-1}})^2 \\
\hat{\alpha}_{t,i} &= \frac{m_{t,i}}{(1 - \beta_1^t)(\sqrt{\frac{v_{t,i}}{(1-\beta_2^t)}} + \epsilon)}
\end{aligned}
\tag{2.5}
$$

**Equation 2.5:** ADAM introduces an effective learning rate $\hat{\alpha}_{t,i}$ per parameter $p_i$ at time step $t$ with respect to the gradients from the estimation function $f_\theta$ based on the first and second moment estimates.

## 2.6 Prioritised experience replay

Data points collected during training are assumed to be independent and identically distributed; however, in practice this is not the case. When training an agent on sequentially collected data points it is beneficial to break the temporal bond between consecutive samples to avoid correlations. A technique for this is *experience replay*, which records an agent's environment trajectory as a vector of the state transition, action and received reward $(s_t, a_t, r_t, s_{t+1})$ in a buffer. This buffer is later sampled according to some distribution and a selected number of recorded experiences are replayed for the network to train on.

In its most basic form of experience replay data points are sampled using an uniform distribution. However, this undermines the importance of learning from experiences that deviate from the network's expectations and produce large estimation errors. By using *prioritised experience replay* (PER) a higher sampling probability is given to records with large prediction errors [13], see equation 2.6. This is combined with the *importance sampling* weighting scheme in equation 2.7 to scale parameter changes and avoid introducing a bias due to this sampling distribution.

$$I(i) = \frac{e_i^\alpha}{\sum_j e_j^\alpha} \tag{2.6}$$

**Equation 2.6:** The importance $I(e)$ for some experience sample $i$ is based on the prediction error $e_i$. The sampling probability can be adjusted with the $\alpha \geq 0$ parameter.

$$w_i = (\frac{1}{\mathrm{N}} \frac{1}{I(i)})^\beta \tag{2.7}$$

**Equation 2.7:** The importance sampling introduces a bias to the $\mathbb{N}$ samples and therefore a correction weight $w_i$ is used for sample $i$, adjusted with the $\beta \geq 0$ parameter.

## 2.7 Reward shaping

In many environments the number of time steps between a non-neutral reward can be long and make the agent training process slow. *Reward shaping* takes inspiration from animal training and addresses this issue by giving small supplemental rewards to encourage the agent when getting closer to the desired behaviour [14]. This supplemental reward $f_t$ is therefore problem specific and is a complement to the original reward $r_t$, as seen in equation 2.8. However, care must be taken to avoid the agent shifting focus from the ultimate goal, instead learning a behaviour optimised on the supplemental reward. This can be done by gradually decreasing the supplemental reward and eventually make it obsolete.

$$\mathbb{R}(a_t, s_t) = r_t + f_t \tag{2.8}$$

**Equation 2.8:** Reward shaping adds a supplemental term to the reward function to assist the agent in learning when the time between a non-neutral reward is long.

A generally suitable supplemental reward function is $f(s_t, s_{t+1})$, that takes the difference of a potential function $\Theta(s)$ between two time steps. Using this the agent is less likely to optimise on the supplemental reward, as it is directly derived from the potential of success of its main objective.

## 2.8 Dynamic networks

When constructing a static neural network its architecture has to be properly modelled to fit the problem it should solve. This is done prior to training and thereafter is a fixed structure. As such the network's capacity could turn out to be under or overdimensioned and require a new iteration of modelling and training. These issues are addressed by networks with a dynamic architecture that can adapt to the task it is learning, without the need for more manual modelling [15].

The way in which the network expands its capacity or makes other structural changes varies between implementations. Some networks trigger a network change when the performance goes below a certain threshold [2], [6], [16]. Other approaches include use a manual network expansion when used across multiple different tasks [3]. In either case the network architecture is not fixed after the modelling stage and can adapt to what is experienced during its training.

## 2.9 One-shot learning

During network training small parameter changes are used to gradually improve the produced estimates. Because of these small changes the overall training speed is slow even in the most basic tasks. An alternative is to have the agent improve directly from its mistakes by immediately compensating for it. This can be achieved with a memory to compare later observations and avoid making the same mistake twice. In supervised learning this error and compensation can be based directly on the data point labels to greatly reduce the time spent training.

## 2.10 Fuzzy neural networks

The *fuzzy neural network* (FNN) is a type of network based on *fuzzy sets* and membership functions to determine the degree of node activation [17]. These networks are inherently non-linear and often designed with few layers and a single trainable output parameter. This is unlike networks such as the fully connected *multi layer perceptron*, that is inherently linear unless non-linear activation functions are used.

In a fuzzy network the membership functions are formed either before or during training. In supervised learning these can be create beforehand by fitting the functions according to the training dataset [18]. If a dataset is not available the membership functions can instead be dynamically created with *growing* and *pruning* rules [19]. This helps strike a balance between the number of nodes and performance of the network.

The membership function are often based on *coarse coding* or some of its derivatives. Input coarse coding forms overlapping receptive regions to mark sets of points in the input space are associated with a neuron, as seen in figure 2.2 [8]. However, the hard edges of these regions do not allow for a point to be a partial member. To make the membership functions fuzzy it is common to instead use a *radial base function* (RBF) to map an input to the range $[0, 1]$. If this function is made differentiable it can be integrated in the gradient calculations for parameter updates. A common choice for a RBF is to use Normal-distributions [19]–[23], see figure 2.3.

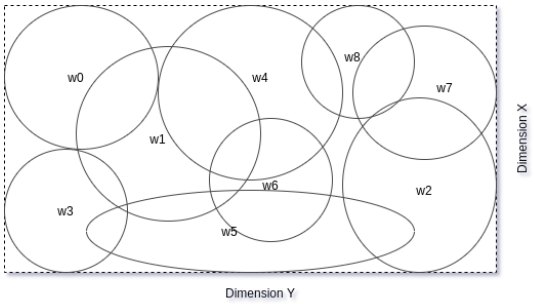


**Figure 2.2:** The 2-dimensional ellipses form a discrete coarse coded approximation of the continuous input space and maps it to a single trainable parameter ($w_0$ to $w_8$). This lets it approximate any function with an adjustable resolution.

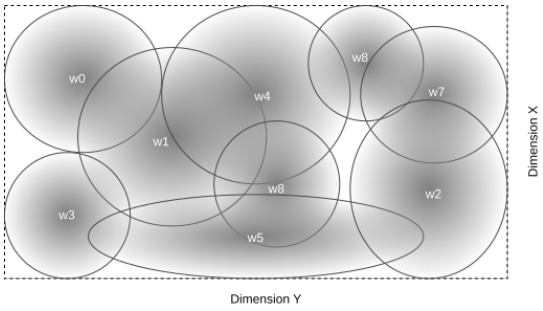


**Figure 2.3:** A radial base function maps the input space membership in the range $[0, 1]$, represented by the gradient to assign levels to the points in the 2d input space. Points closest to the feature centre have the highest level.

Due to fuzzy networks directly filtering the their input into fuzzy sets and mapping these to an output, they have a level of inherent explainable behaviour. This can be used both to visualise the network and manually alter nodes after training.

## 2.11 Life-long Learning from Zero

The *Life-long Learning from Zero* (LL0) is an implementation of a fuzzy neural network with one-shot learning and a dynamic architecture. It has previously been used for classification tasks and shown to have a high performance and resource efficiency [6]. When compared to other networks of the same complexity, LL0 produces results that quickly converged on a dataset and with a higher accuracy. LL0 achieves this by using a set of specialised rules for expanding the network, and in the process extract abstract concepts from the dataset. This extraction method is the core idea behind the network, which it uses to form concept nodes.

Like other fuzzy networks LL0 estimate functions by discretising its input space with concept nodes and trainable Normal-distribution membership functions, see figure 2.4 for a simple diagram of how they operate. Each concept node uses its attached value nodes, containing the membership functions, to calculate its own *activation level*. These activation levels are used by the network to identify and classify inputs in terms of concepts. By forming multiple concepts node layers the network can gradually learn more abstract patterns from the input, an example would be to form a concept node for berries after seeing the similarity between blueberries and raspberries.

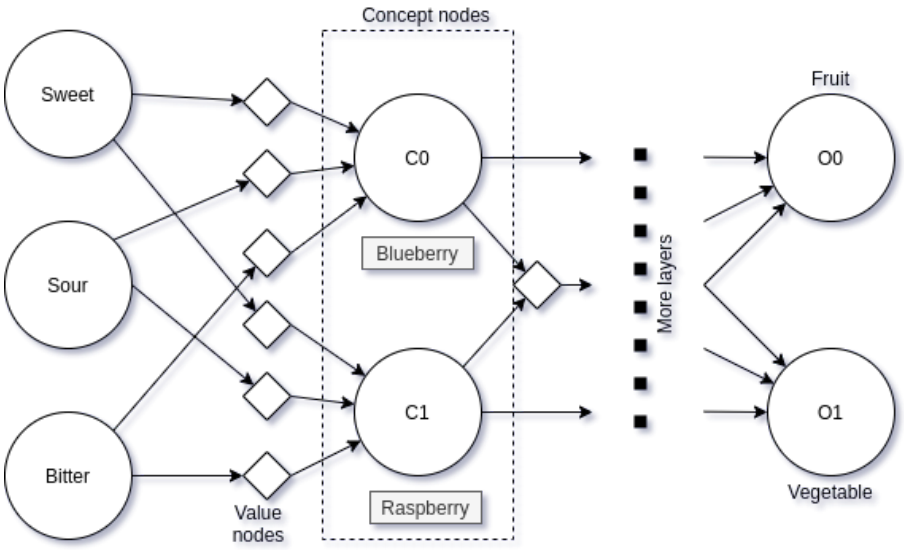


**Figure 2.4:** A rough illustration of the LL0 network using value nodes as membership functions and concept nodes to learn abstract patterns from the input, here exemplified by the addition of the labels 'blueberry' and 'raspberry'.

To allow for sharing membership information between concepts the membership functions are stored in the preceding value nodes, containing a single Normal-distribution *probability density function* (pdf) with the trainable parameters $\mathcal{N}_k(\mu_k, \sigma_k)$. Each of the input elements pass through a separate value node and are passed to a concept node for evaluating its activation according to equation 2.9.

$$a(\mathbf{x}) = \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{pdf_k(x_k)}{pdf_k(\mu_k)} \tag{2.9}$$

**Equation 2.9:** An example of a concept node activation function based on its $i$ inputs from connected value nodes with $\mathbf{x}$ as input.

To limit the noise from concept nodes with too low activation, LL0 uses a threshold between $(0, 1)$ as a minimum level. This limits the network to output only estimates from confident concept nodes. The estimates are learnt through normal network training, including the trainable parameters for the membership functions in the value nodes.

Even if the premise of LL0 is well suited for supervised learning problems, it was not designed for reinforcement learning. Among several other problems, the reinforcement learning training algorithms do not have access to labelled data points, meaning several of the core techniques in LL0 are not applicable. For LL0 to perform well in the larger set of reinforcement learning problems, it is necessary to generalise it to not depend on labelled data points.

# 3

# Related work

Of currently widely used methods for solving biological simulations, *stochastic dynamic programming* is very commonly used and involve searching over a large stochastic state space. However, this quickly becomes impractical in larger dimensions and requires better suited optimisation techniques. Frankenhuis et al's suggest reinforcement learning as an alternative [7]. However, even if reinforcement learning can work with large dimensions, it is not ideal if then instead it depends on a black box estimation function. The related field of medicine has had a lot of work in this domain due to the need of explainable artificial intelligence [24], in biological simulations this could also be used to avoid unintentional and unrealistic agent behaviour.

This does not necessary mean the networks themselves have to be explainable. There has been work on extracting explainable behaviour directly from existing network architectures with methods such as *sensitivity analysis* and *layer-wise relevance propagation* [25]. However, these methods are not inherently explainable and work from the output side of the network and back to the input, unlike what is possible with networks like LL0 that were designed for explainability.

Given the LL0 network's resource efficiency [6], it could be a promising candidate for biological simulations in more ways than just explainability. There has already been a previous attempt to adapt LL0 for reinforcement learning [16], which had issues when used in more complicated environments. They converted many of the rules used to manage the growth and concept generalisation, keeping the trigger to build new concept nodes and immediately compensate for prediction errors. This was also used by early dynamic networks like the *Cascade-correlation neural network* [2], one of the pioneering networks with this property. Both of these networks used threshold hyper parameters to control their growth and overall approximation quality. Another approach was instead used by the *progressive neural networks* to dynamically extend the network when transferd between similar tasks [3]. It keeps a full copy of its past experience and connects it to an extension network, meaning it does not lose any performance after learning more tasks.

These dynamic properties are also seen in work with fuzzy neural networks for supervised learning of classification and regression tasks [20], [21]. Several of these implementations extract their membership function parameters by optimising directly on datasets, such as what was done by Chen et al in their *orthogonal least squares* [18] to minimise the number of network nodes and balance its approximation resolution. Other placement strategies involve ranking multiple complete solutions in genetic algorithms [17].

While these methods work for classification tasks with labelled datasets, they assume a static environment that does not change over time and therefore are not suited for reinforcement learning. However, fuzzy networks can be constructed to regard temporal differences in time series, making them viable for life-long learning tasks like the ever changing conditions of a water treatment facility [22]. Adapting fuzzy networks for reinforcement learning can also be seen in implementations for controlling telecommunication antennas [23].

Most of these fuzzy networks manage their nodes with similar growing and pruning rules. Examples from static environments include a sensitivity function to grade nodes according to their activity over a dataset, favouring stable nodes with less noisy activation [20]. An alternative time dependent version is seen in the work by Han et al based on Fourier-series to achieve a similar effect [19]. In either case, these methods make trade-offs between network size, stability and approximation resolution.

# 4

# Method

In the following chapter the developed dynamic network adaption of LL0 for reinforcement learning, called RLL0, is presented along with its network growing and pruning rules. Following this is an overview of the environments, including an in-depth description of the virtual animal ecosystem, used for benchmarking RLL0's performance against a reference network.

## 4.1 Reinforcement life-long learning from zero

To use LL0 as an estimation network in a reinforcement learning problem it was necessary to change the behaviour based on assumptions only true in supervised learning. This adapted network was given the name *Reinforcement life-long learning from zero* (RLL0) and made significant changes to the dynamic property and removed the one-shot learning. This removal was due to the stochastic nature of reward functions and lack of labels for the one-shot learning.

The RLL0 neural network is in a sense a simplified version of LL0 that operates with a single hidden layer. It uses the same RBF Normal-distributions for its membership functions to form receptive regions in the input space; however, the method used to generate and maintain nodes has changed to the generalisation rules in section 4.1.4.

### 4.1.1 Trainable parameters

The concept nodes in RLL0 are similar to those found in LL0, see figure 4.1 for a decomposition of the node and its parameters. Each concept node passes its input vector $\mathbf{x} \in \mathcal{R}^k$ through a $k \times 1$ membership function matrix to determine how well it matches its receptive region. Just as in LL0 the membership functions use a probability density function from a $\mathcal{N}(\mu_k, \sigma_k)$ Normal distribution per input element, as seen in equation 4.1. The resulting levels are aggregated to an activation level according the activation function, with the two tested functions in equation 4.2 and with importance weights $w_k$ in equation 4.3. After calculating the activation level of all network concept nodes, the node with the highest activation level is said to be the *top active* concept and is considered the most reliable concept.

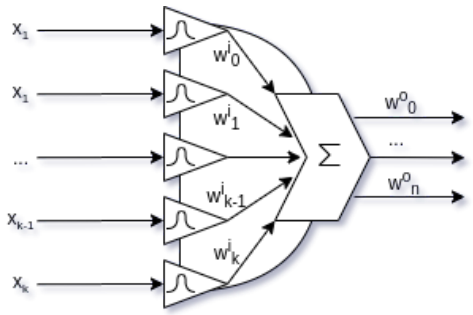


**Figure 4.1:** Each concept node uses a Normal distribution PDF to match an input vector against its receptive region.

$$pdf_k(\mathbf{x}) = \frac{1}{\sigma_k} e^{-\frac{1}{2}(\frac{\mu_k - x_k}{\sigma_k})^2} \tag{4.1}$$

**Equation 4.1:** The PDF used to calculate the concept node activation level given some input vector $\mathbf{x}$. Note that each concept nodes holds a set of $k$ separate distributions, one per input element.

$$a(\mathbf{x}) = \prod_k \frac{pdf_k(x_k)}{pdf_k(\mu_k)} = e^{-\sum_k (\frac{\mu_k - x_k}{\sqrt{2}\sigma_k})^2} \tag{4.2}$$

**Equation 4.2:** The function used to calculate how active a concept node given an input vector $\mathbf{x}$. The function operates in the $(0, 1)$ range, with a higher value meaning the input is considered more familiar.

$$a'(\mathbf{x}) = \frac{\sum_k \frac{pdf_k(x_k)}{pdf_k(\mu_k)}}{\sum_k w_k^i} \tag{4.3}$$

**Equation 4.3:** An alternative to $a(\mathbf{x})$ that uses the importance weights $w^i$ to adjust the contribution of individual inputs.

The way in which the concept nodes are constructed makes changes made to one node independent from changes to some other node. This can be seen in figure 4.2, which depicts a RLL0 network with 3 concept nodes in the hidden layer. The number of trainable parameters in a concept node depends on the dimensions of the input and output layers, as seen in table 4.1.

Each network input element is fully connected to the concept node layer, through their respective PDF membership functions. In addition to the two trainable distribution parameters, mean $\mu_k$ and standard deviation $\sigma_k > 0$, there is an importance weight $w_k^i \in [0, 1]$ for weighting inputs differently in the activation function. This activation is scaled by a single trainable parameters $w_j^o$ per output element. However, the concept node only produces any output if it is the top active, the remaining concept nodes are silenced and do not contribute.

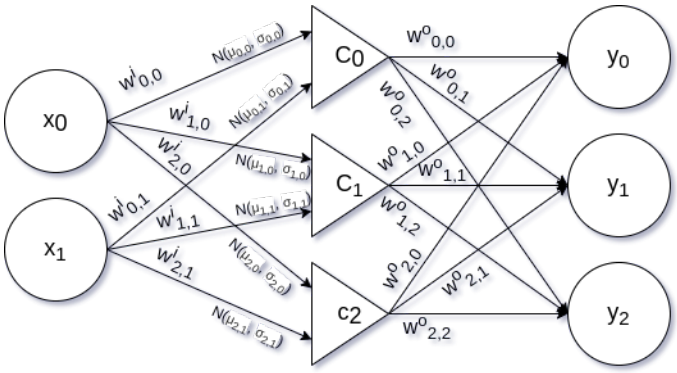


**Figure 4.2:** The RLL0 neural network has a single hidden layer with concept nodes. Each node learns to estimate the output for its receptive region by adjusting its trainable parameters.

**Table 4.1:** The trainable concept node parameters in the RLL0 neural network receiving input vectors of size $n$ and produces an output vector of size $m$.

| Name | Parameter type |
|---|---|
| $\boldsymbol{\mu} \in \mathbb{R}^n$ | Input distribution mean vector. |
| $\boldsymbol{\sigma} \in \mathbb{R}^n$, | Input distribution standard deviation vector. |
| $\boldsymbol{w}^i \in (0, 1)^n$ | The input importance weight vector. A higher value gives more weight to that input element, relative to the other inputs. |
| $\boldsymbol{w}^o \in \mathbb{R}^m$ | The output weights for scaling the concept node activation signal. Effectively this is the output estimate for the concept node's receptive region. |

### 4.1.2 Backpropagation

The trainable parameters in the RLL0 neural network are updated by *stochastic gradient decent* (SGD) and *backpropagation*. Stochastic gradient decent is an update method which moves the parameter values in the opposite direction of their gradient with respect to a loss function, the error between the predicted value $\hat{y}$ and the desired target value $y$. In equation 4.4 is an example of a parameter $p$ update with SGD at time step $t$ with input $\mathbf{x}$ and the loss function $L(\mathbf{x}, y)$. To control the magnitude of the change, the update is scaled by a scalar $\alpha > 0$, called the *learning rate*.

To calculate the gradients between layers RLL0 uses backpropagation to split the gradients according to the chain rule, as seen in equation 4.5. Thus the gradients can be calculated across layers and propagated backwards down the network.

$$p_{i+1} \leftarrow p_i - \alpha \cdot \frac{\partial L(\mathbf{x}, y)}{\partial p_i} \tag{4.4}$$

**Equation 4.4:** A parameter $p$ at time step $t$ is gradually moved in the opposite direction of its loss function $L(\mathbf{x}, t)$ based on the input $\mathbf{x}$ and target $y$. The change is scaled by a learning rate $\alpha$ to control avoid making too large changes.

$$\frac{\partial a(c)}{\partial b} = \frac{\partial a(c)}{\partial c} \cdot \frac{\partial c}{\partial b} \tag{4.5}$$

**Equation 4.5:** Using the chain rule the gradients in SGD can be split to simpler gradients and combined with those calculated in the next network layer.

The loss function chosen for RLL0 was the *mean square error*, see equation 4.6. It is based on the network predictions $\hat{y}_t \in \hat{\mathbb{Y}}$ and the target values $y_t \in \mathbb{Y}$. The two sets $\hat{\mathbb{Y}}$ and $\hat{\mathbb{Y}}$ are collected by the agent during environment interaction and used by the training algorithms to generate the update gradients.

$$\frac{1}{n}\sum(y_t - \hat{y}_t)^2 \tag{4.6}$$

**Equation 4.6:** The MSE loss function used by RLL0 to generate network update gradients from the input $y$ and target $\hat{y}_t$.

### 4.1.3 Network gradients

The following section describes RLL0's parameter gradient equations when using the $a'$ concept node activation function. This activation function utilise more of the network's trainable parameters than $a$; however, the overall difference between the two is small and the $a$ function's gradient calculations are therefore left out.

Calculation of the network gradients is done for each train on data point and follows the backpropagation presented in the earlier section. The loss function is denoted as $L(\hat{y}_t, y_t)$ for the RLL0 network's predicted value $\hat{y}_t$ and target $y_t$ at time step $t$ when using action $a_t \in 1, ..., m$. The only parameters in the network that are affected by the gradients is the concept node that was top active at $t$, the remaining node parameters are frozen. In equation 4.7 are the gradients for the network output weights $\mathbf{w^o}$ and in equation 4.8 the gradients for the importance weights $\mathbf{w^i}$.

$$\frac{\partial L(\hat{y}_t, y_t)}{\partial w_j^o} = \begin{cases} L(\hat{y}_t, y_t) \cdot a'(\mathbf{x}_i), & \text{if } a_t = j \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

**Equation 4.7:** The output weight gradient for the $j^{\text{th}}$ output element, for $j \in \{1, ..., , m\}$, is simply the entire loss value when the action $a_i$ was taken at time step $t$.

$$\frac{\partial L(\hat{y}_t, y_t)}{\partial w_j^i} = L(\hat{y}_t, y_t) \cdot \frac{(\frac{pdf(x_{i,j}, \mu_j, \sigma_j)}{pdf(\mu_j, \mu_j, \sigma_j)})}{\sum_{k=1}^{n} w_k^i} \tag{4.8}$$

**Equation 4.8:** The importance weight loss for the $j^{\text{th}}$ input element, for $j \in \{1, ..., n\}$, given the input vector $\mathbf{x}_i$ is calculated based on the activation in the concept node's membership function.

The remaining gradients used for calculating the parameters in the membership functions can be seen in equation 4.9 for the distribution mean and in equation 4.10 for the standard deviation.

$$\begin{aligned} \frac{\partial L(\hat{y}_t, y_t)}{\partial \mu_j} &= \frac{\partial L(\hat{y}_t, y_t)}{\partial a_j(x_{i,j})} \cdot \frac{\partial a_j(x_{i,j})}{\partial \mu_j} \\ &= \frac{\partial L(\hat{y}_t, y_t)}{\partial a_j} \cdot w_j^i \cdot (-e^{-0.5 \cdot (\frac{\mu_j - x_{i,j}}{\sigma_j})^2} \cdot \frac{(\mu_j - x_{i,j})}{\sigma_j^3}) \end{aligned} \tag{4.9}$$

**Equation 4.9:** The gradient used to update the Normal distribution mean $\mu_j$ using backpropagation.

$$\begin{aligned} \frac{\partial L(\hat{y}_t, y_t)}{\partial \mu_j} &= \frac{\partial L(\hat{y}_t, y_t)}{\partial a_j(x_{i,j})} \cdot \frac{\partial a_j(x_{i,j})}{\partial \mu_j} \\ &= \frac{\partial L(\hat{y}_t, y_t)}{\partial a_j} \cdot w_j^i \cdot \frac{(1 - (\frac{\mu_j - x_{i,j}}{\sigma_j})^2) \cdot e^{-0.5 \cdot \frac{\mu_j - x_{i,j}}{\sigma_j})^2}}{\sigma_j} \end{aligned} \tag{4.10}$$

**Equation 4.10:** The gradient for the Normal distribution standard deviation $\sigma_j$ when updated using backpropagation.

### 4.1.4 Handling concept nodes

Unlike the generalisation rules in LL0 for classification problems, RLL0 does not have access to the ground truth labels. This means it is not possible to calculate any reliable prediction errors during training. Using a prediction error to form a new concept node to compensate is therefore unlikely to accurately resolve the estimation error, which has to be learnt over the course of several interactions to account for its variability. The chosen growth rule in RLL0 was to instead triggered at any point the input observation did not produce a sufficiently active existing concept node, and train the existing nodes when one is.

A concept node is considered sufficiently activated if its activation goes above a set threshold $\Omega \in (0, 1)$ and is the top active node. The growth rule can thus be formulated as the boolean function $\mathrm{G}(\mathbf{s_t})$ in equation 4.11. If it returns a value of 1 a new concept node is added, with its mean vector set to the current observation vector and its both standard deviation and importance weight vectors initialised to a vector of ones. Otherwise the existing top active node is trained.

$$G(s_t) = \begin{cases} 0, & \max\limits_{c \in \mathbb{C}} \; a'_c(s_t) \geq \Omega \\ 1, & \text{otherwise} \end{cases} \tag{4.11}$$

**Equation 4.11:** If no concept $c \in \mathbb{C}$ has its activation $a'_c(s_t)$ above the threshold $\Omega$ a new concept node is added.

As RLL0 does not have a one-shot property like LL0, it instead tries to speed up training by bootstrapping on nearby nodes. The growth rule assumes that concept nodes with a short Euclidean distance between their mean value vectors have similar estimations. Newly created concepts therefore copy their nearest neighbour's output vector weights. If there are no neighbours these instead assume the value 0. This method favours environments where the expected return from one state is similar to other states near by in the input space.

## 4.2 Generalisation rules

To improve learning RLL0 uses a set of growing and pruning rules to reducing the number of network nodes and avoid bad placement of receptive regions in the input space. These rules are adaptations of those used by Chan et al in supervised learning [20] to favour frequently used concept nodes.

### 4.2.1 Growing rule

Due to the design of the concept nodes used by RLL0 it is necessary to have nodes cover the input space with their receptive regions to make any estimations. In terms of the number of concept nodes used, it is wasteful to form concept nodes that cause large overlaps with existing nodes. While their placement is already guaranteed a minimum distance by the activation threshold and activation function, this still leads to significant overlap and unnecessary nodes. By adding an additional growing condition to enforce a minimum distance to the existing trigger, fewer nodes are used. This enforces a Euclidean distance $\delta$ between the mean vectors of concept nodes as in equation 4.12, moving them in the opposite direction their closest neighbour if too close.

$$\mu_{c,k} \leftarrow \begin{cases} \mu_{c,k} - (\delta - |\Delta_{c,c'}|) \cdot \frac{\Delta_{c,c'}}{|\Delta_{c,c'}|} & \text{, if} |\Delta_{c,c'}| < \delta \\ \mu_{c,k} & \text{, otherwise} \end{cases} \tag{4.12}$$

**Equation 4.12:** The minimum distance rule updates the mean value $\mu_{\mathbf{c}}$ of some newly created concept node $c$ and its closest neighbour $c'$, based on Euclidean distance $|\Delta_{c,c'}|$ between the two centres $\mu_{\mathbf{c}}$ and $\mu_{\mathbf{c}'}$.

### 4.2.2 Pruning rule

The pruning rule is used to rid the network of unnecessary concept nodes. They may have been created at some point and are no longer of use. This can happen when the agent learns a better strategy or that the observable input space has changed. To maintain performance and the ability to learn anew,if needed, the network uses a pruning rule that prioritising nodes that are currently useful for the network, measured by their average activation level over some span of time.

In RLL0 this pruning rule is run across all concept nodes in the network at an interval $\Gamma$, the number of performed network updates before pruning again. The pruning process consists of forming a neighbourhood set $N_T$ from the network's experience buffer. Each neighbourhood is a set of points from the experienced input vectors during the buffered time span $T$, formed by clustering them to their nearest neighbour. Pseudo code for this clustering can be seen in figure 4.3.

The neighbourhoods calculate their centre points and are used in the concept node sensitivity function $S(c, N_T) \in [0, 1]$ in equation 4.13 measures how active each node $c$ was throughout this time span. A value near 0 means infrequent use and 1 constant use. By setting a threshold $\omega \in [0, 1)$ and adjusting it to be $\omega << 1$, the less sensitive nodes can be pruned from the network if below this threshold.

$$S(c, N_T) = \frac{\sum_{p \in N_T} c(p)^2}{|N_T|} \tag{4.13}$$

**Equation 4.13:** The sensitivity function calculates how sensitive the concept $c$ given the neighbourhood centres $N_T$. A value close to 0 shows constant insignificant activation and a value close to 1 means frequent high activation over the time span $T$.

```
function FindNeighbourhoodCentres(P_T)
    Queue = MinPriorityQueue()
    foreach p0 in P_T:
        foreach p1 in P_T:
            Queue.push(distance(p0,p1),p0,p1)
    Seen = EmptySet()
    Neighbourhoods = NewNeighbourhoodList()
    while Queue not empty:
        (distance, p0, p1) = Queue.pop()
        Seen.add(p0 and p1)
        if p0 and p1 not in Seen:
            n = NewNeighbourhood()
            n.add(p0)
            n.add(p1)
            Neighbourhoods.add(n)
        elseif p0 not in Seen:
            Add p0 to p1's neighbourhood
        elseif p1 not in Seen:
            Add p1 to p0's neighbourhood
    N_T = EmptySet()
    for each Neighbourhood in Neighbourhoods:
        N_T.add(Neighbourhood.centre())
    return N_T
```

**Figure 4.3:** Pseudo code for finding the $N_T$ neighbourhood centre from the set of observations vectors $P_T$ in the experience buffer during the time span $T$.

## 4.3 Full network summary

On the next page figure 4.4 the full RLL0 network training algorithm is given as pseudo code. Since the training algorithms and growing and pruning rules are separate it is possible to enable and disable these in case the rules are not suitable for a specific learning environment.

```
function LearnRLL0(
    n: num inputs,
    m: num outputs,
    G: training algorithm,
    E: environment,
    V: training interval,
    T: activation threshold,
    I: spacing threshold,
    J: pruning threshold,
    K: pruning frequency):

    N = EmptyRLL0Net(n, m)
    B = EmptyExperienceBuffer()
    k = 0
    while t < maxTimeSteps:
        E.reset()
        s = E.observation()
        while not end of episode:
            t = t + 1
            if k > 0 and k mod K = 0:
                N.prune(J)
                k = 0
            if N.activation(s) < T:
                N.grow(s, I)
            action = N.sampleAction(s, G)
            reward = E.step(action)
            newS = E.observation()
            B.record(s, a, r, newS)
            s = newS
            if G = PPO and t mod V = 0:
                N.trainPPO(B)
                B.empty()
                k = k + 1
            if G = DDQN and t mod V = 0:
                N.trainDDQN(B)
                B.empty()
                k = k + 1
        if G = MC:
            N.trainMC(B)
            B.empty()
            k = k + 1
    return N
```

**Figure 4.4:** Pseudo code of RLL0's learning process.

## 4.4 Evaluation method and environments

To measure how well the developed RLL0 network performed in comparison to a well-known neural network, it was evaluated through a benchmark against a two layer fully connected reference neural network with tanh activation. This benchmarking test consisted of training the two networks in several different setups of the developed virtual animal ecosystem environment, in addition to common control systems from OpenAI Gym [26] and several of the environments previously developed by Eriksson et al.

For each environment the networks were trained over a fixed number of time steps and with several training algorithms. These runs had their results averaged over 10 separate training sessions for a more reliable performance estimate. The table 4.2 contains a summary of all the benchmarking setups. Due to the lack of multi agent and Monte Carlo training support for the reference network neither MC training nor the animat #2 environment could be used with the reference network.

The remainder of this chapter presents the animat environment in detail and the others very briefly, as they are already well documented in their original papers.

**Table 4.2:** A summary of the benchmarking setups used for the different networks and algorithms.

| Environment | Steps | RLL0 alg. | Refnet alg. | # Agents |
|---|---|---|---|---|
| Animat #0 | $10^6$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Animat #1 | $10^6$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Animat #2 | $10^6$ | MC, PPO, DDQN | — | 2 |
| Cartpole | $10^6$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Lunar lander | $10^6$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Berry | $10^5$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Noisy berry | $10^5$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Discrete catch | $10^5$ | MC, PPO, DDQN | PPO, DDQN | 1 |
| Grid | $10^5$ | MC, PPO, DDQN | PPO, DDQN | 1 |

### 4.4.1 The animat environment

The animat environment refers to a simulation environment containing animats, which are virtual animals tasked with surviving in a virtual ecosystem. To test RLL0's suitability in a biological simulation the benchmark included an animat environment, similar to the one presented by Strannegård et al used for combining reinforcement learning and genetic programming in virtual animals [27]. However, this new environment is more capable and uses both continuous space and a different method of animal representation. An image taken of one of the environment setups can be seen in figure 4.5. This image has several of the environment's optional visualisation capabilities enabled to demonstrate some of its most prominent features.

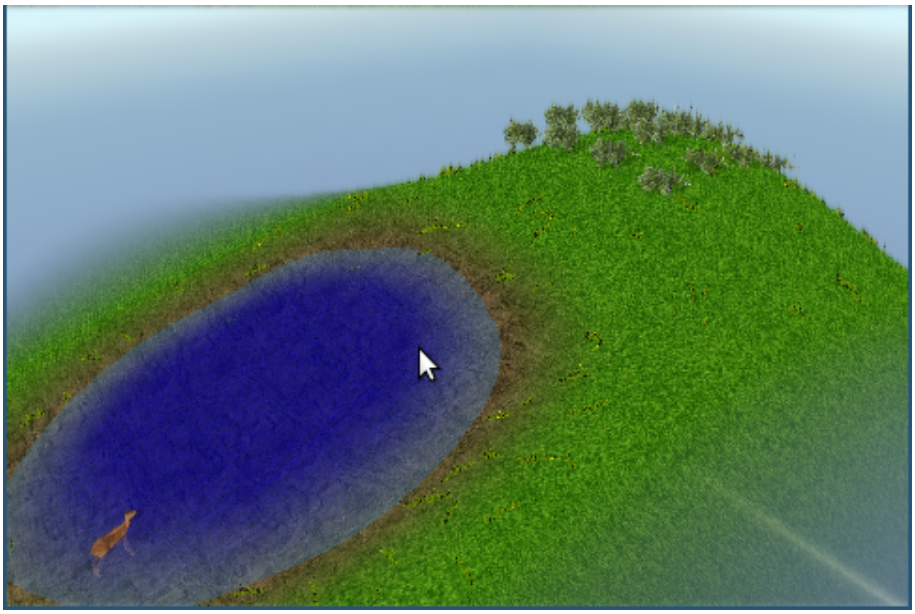

**Figure 4.5:** The animat environment simulates a virtual animal ecosystem, where agents learn to survive by foraging, avoiding predators or finding prey.

Each animat in the environment is controlled by a reinforcement learning agent. The agent is assigned a species to define what set of actions and a reward function that reflects its unique needs. For instance, the antelope animat has to feed on foliage and drink water whereas the tigers must prey on other animals. The antelope therefore has to avoid predators to survive, yet still expose itself when approaching a waterhole to drink. In this sense the environment is attempting to simulate the rise of animal behaviour and natural equilibrium with artificial intelligence.

For an animat to survive in this environment it has to maintain its own hunger, thirst and health. These are represented as a vector of real numbers (saturation, hydration, health) $\in [0, 1]^3$. At each time step the animat receives these values as part of its state observation input, in addition to a fixed size vector containing smell intensities as seen in equation 4.14. These smells are the only available means for to the animat to localise itself, as they have no taste, sight, feeling or hearing.

$$s_t = (\text{saturation}_t, \text{hydration}_t, \text{hunger}_t, d_{1,t}, d_{2,t}, ..., d_{n_s,t}) \qquad (4.14)$$

**Equation 4.14:** At each time step $t$ the animat agent receives an observation vector $s_t$ with its state vector and a collection smell intensities $d_{i,t}, 1 \leq i \leq n_s$ for localising itself.

The values for the smell intensities in the observation vector are used by the agent for foraging and to avoid dangers such as predators. Each sample $d_i \in [0, 1]$ is taken from receptor points around the animat that register how intensive a particular type of smell is at that location, see figure 4.6 for this placement. The actual number of receptors used is controlled by an environment parameter and can be adjusted as seen fit. All receptors are relative to the animat location and rotation, which means they act as an animal's nose. Using several receptors then representing turning its head to detect smells in different directions.

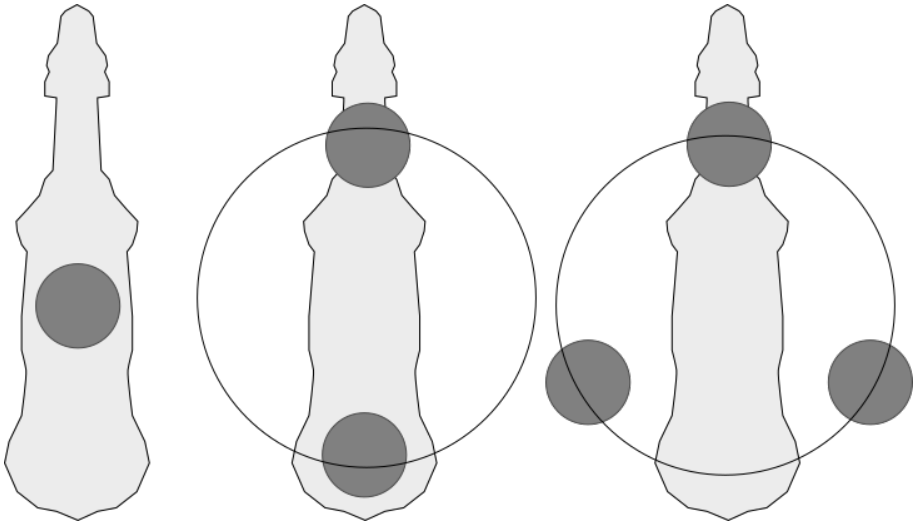

**Figure 4.6:** An illustration of three antelopes from above with a different number of smell receptors, the filled circles. With just one receptor the agent can only learn the intensity. However, if more samples are made it can theoretically learn the direction as well.

The types of smells available to the animat, and therein how many entries of different smells in their observation vectors, depends on the number of different type of entities in the simulated world. An entity is a either a type of natural resource, grass or water, an animal species or a type of (as far as this simulation is concerned) inanimate object such as a bush or tree.

Each type of entity in the environment emits an unique smell simulated by a simplified pressure propagation algorithm in an independent vector grid layer. The magnitude and direction of each vector in a grid controls the propagation of smell. When running the environment the smell intensities and directions are visualised in the by rendering the vector grids in different colours, as seen in figure 4.7.

Because the animats move in continuous space, their receptor points interpolate between the vectors in the vector grids. The same interpolation method is also used at each time step when an entity produces a small amount of smell in its type's corresponding entity layer. This way prey animals can detect nearby food and predators, whom in turn can hunt by tracking the smell of its prey.

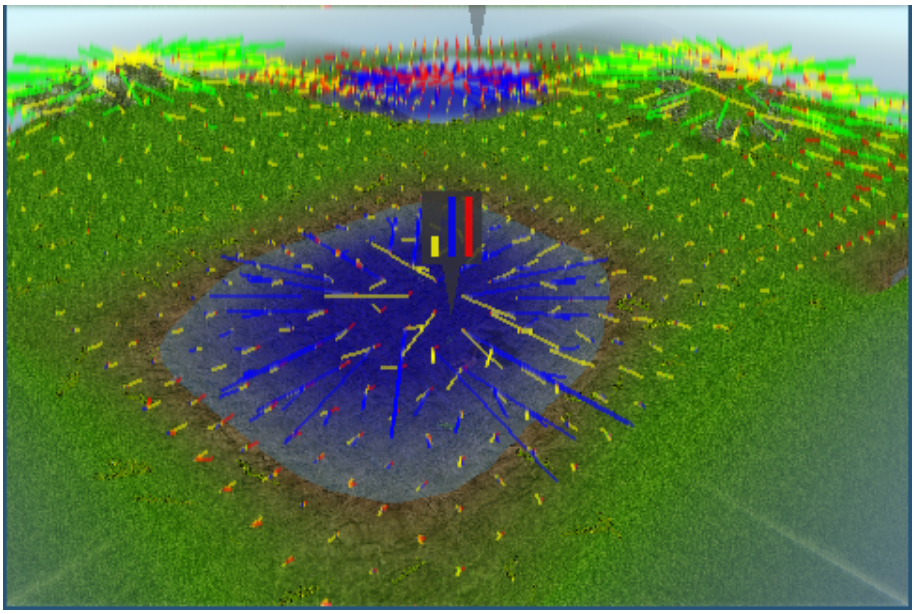

**Figure 4.7:** The coloured lines represent the spread of smell. Each colour represents a different entity smell layer and the intensity and direction by the magnitude of the vector.

Based on these observations the agent has to select an action from the its species' action set. These actions consists of two subsets, one set for following a particular smell and another with type specific actions. If the agent picks an action corresponding to a smell it will move in the opposite direction detected by its receptor, as seen in figure 4.8. In the other set there are type specific actions, such as some eating and attacking interaction. The type specific actions are described in detail in section 4.4.1.1.

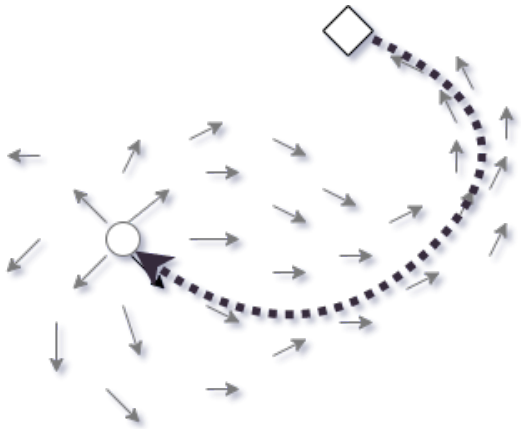

**Figure 4.8:** The agent (diamond) follows the smell vectors in the opposite direction along the dashed curve. It will eventually arrive at the source (circle) if followed.

Just like the action sets, the reward function for each animat is unique to its species. For all animats the reward received at a time steps is based on the change of its current homeostasis value $z_t$ at time step $t$, the lowest of its three status values. At each step the reward function returns an amplified version of this change, as seen in equation 4.15, to help the agent prioritise addressing the lowest value. In the reward function several additional functions are used, including $D(t)$ which returns 1 if the animat is considered dead at time step $t$ or 0 otherwise. Another is the agent specific reward shaping function $R_s^{\text{type}}$ to help the agent detect favourable situations early in training, such as approaching food when hungry. However, this supplemental reward quickly becomes redundant once the agent learns a more favourable strategy.

$$r_t = (1.0 - D(t)) \cdot \frac{(z_t - z_{t-1})}{\max(z_{t-1}, 0.1)} - 10 \cdot D(t) + \frac{R_s^{\text{type}}(t)}{100} \quad (4.15)$$

**Equation 4.15:** An animat's reward function at time $t$. It returns an amplified difference in its homeostasis to make the reward lower the longer the animat waits with addressing the lowest status value.

The used homeostasis representation means a saturation $\approx 1$, a hydration level $\approx 1$, and a health level $\approx 1$, represents a good state agent condition. However, the reward function encourages an agent to let these values drop before eating or drinking. This is to avoid the hard local learning minima of agents just maximising the fastest falling status value. With the used reward function good agent performance is instead achieved after letting these values lower and then perform an action to raise them. With this approach it becomes easier for the agent to learn to address the other status values as well.

The end of a training episode is marked by the animat's death. If any of its status values reach 0 or it lives for more than $10^4$ time steps, the animal is considered dead and a terminal state is returned to the training algorithm.

As a short summary, for an agent in one of the animat environments to perform well it must learn a sustainable behaviour according its type specific reward function. Whereas prey eat vegetation, the predators have to eat other animats. Due to their conflicting objectives and their codependency, the animals have to learn a collective behaviour. In the following section the two different animat types used in the developed environment are presented, as well as the 3 versions of the environment used for benchmarking, called *Animat #0*, *Animat #1* and *Animat #2*.

#### 4.4.1.1 Animat types

In the three animat environments used for benchmarking RLL0 there were two types of animats, antelopes and tigers as prey and predator respectively.

The antelopes can be configured to eat either grass or other specific entities, such as bushes or trees, to force it on a desired diet. This configuration affects the reward shaping function $R_s^{\text{antelope}}$ for antelopes, which can be seen in equation 4.16. This function uses the level of nearby foliage $g_{a,t} \in [0, 1]$ and water $w_{a,t} \in [0, 1]$ to give a small reward when near greenery and hungry or water when thirsty.

For interacting with the environment other than through movement, the antelope has two additional actions. The eating action makes the antelope drink when standing in water and eat when either on top of grass or near vegetation, whichever belongs to its diet. The final antelope action is an idle action, which does nothing and can be used by the antelope to wait.

$$R_s^{\text{antelope}}(t) = \begin{cases} g_t & , if \text{ saturation}_t < \text{hydration}_t \\ w_t & , \text{otherwise} \end{cases} \tag{4.16}$$

**Equation 4.16:** The reward shaping function for the antelopes endorse getting closer to vegetation when hungry and water when thirsty.

Unlike the foraging behaviour by the antelopes, the tigers need to prey on other animats to eat. For this reason they have a slightly different reward shaping function, as seen in equation 4.17. It uses the positive integer $n_{a,t} \in \mathbb{N}$ to represent how many animats are nearby. While a tiger can eat other animats for saturation, it still has to drink water when thirsty. As the antelopes and tigres share the same waterholes, it becomes an ideal hunting ground.

Just like the antelopes the tigers have two additional actions. When a tiger selects their attack action they will feed on nearby animats and inflicting damage to the other animat's by lowering its health status variable. An animat naturally regain their health when left alone over time, but the tiger will have to kill the prey in order to maximise its food. The final eating action is used by the predator to drink when standing in water. Just like the antelope the tiger has to alternate between finding water and hunting.

$$R_s^{\text{tiger}}(t) = \begin{cases} n_t & , if \text{ saturation}_t < \text{hydration}_t \\ w_t & , \text{otherwise} \end{cases} \tag{4.17}$$

**Equation 4.17:** The reward function for the tigers rewards the animat for getting closer to water when thirsty and to prey when hungry.

#### 4.4.1.2 Animat scenarios

The animat environment is fully configurable to setup different scenarios, such as a particular layout of the simulated virtual environment and what entities are its inhabitants. For the three scenarios used for the RLL0 benchmarking, two used only the antelope type and the third both the antelope and tiger types.

The *Animal #0* scenario uses one antelope that has access to both water and food in the form of grass. The animat has to learn how to survive by going between the two sources, as staying in one place will eventually cause either starvation or dehydration. The abundance of predators and plenty of grass means it is just a matter of alternating between eating or drinking and moving between the two sources.

A slightly more complex environment is the *Animat #1* scenario. This environment exchanges the antelope's diet from grass to bushes. This is slightly harder, as there are fewer food locations and they are further from the water banks. To get to the bushes the agent must follow the smell of grass and then switch to finding the bushes.

The final *Animat #2* scenario introduces a predator into the scenario used in Animat #1, making it a multi-agent environment. In this environment the tiger will hunt the antelope, and the antelope has to learn to avoid it and manage its own needs just as in the previous scenarios.

### 4.4.2 Cartpole environment

The cartpole environment is a part of the OpenAI gym and is a typical example environment for testing a network's learning capabilities in a control system. The agent's goal is to learn to balance an inverted pendulum, see figure 4.9. At each time step the agent receives a four element wide state vector, with the cart base's position, velocity, pole tip velocity and pole angle. Based on this the agent has to pick one of the two discrete action, corresponding to applying a pushing force on the cart form either the left or right side. In whichever case, the agent is rewarded with a constant of 1 at each step until the end of the episode. This termination is triggered once the agent has balanced the pole for more than 500 time steps, goes off the edge or the pole angle strays more than a set threshold from the vertical axis.

### 4.4.3 Lunar lander environment

The lunar lander environment, like cartpole, is a control system environment from the OpenAI gym in which the agent is tasked with controlling a lunar lander onto a landing platform, see figure 4.10. In each time step the agent receives an eight element wide observation vector and can choose one of four actions, to activate one of the three thusters or do nothing. If a side thruster is activated the agent receives a $-0.03$ reward or a $-0.3$ reward if the main thruster below the lander is active. Upon landing the agent can receive a reward up to 140, depending on how well it lands on top of the landing pad. The episode ends once the lander touches the ground or goes beyond the environment edges.

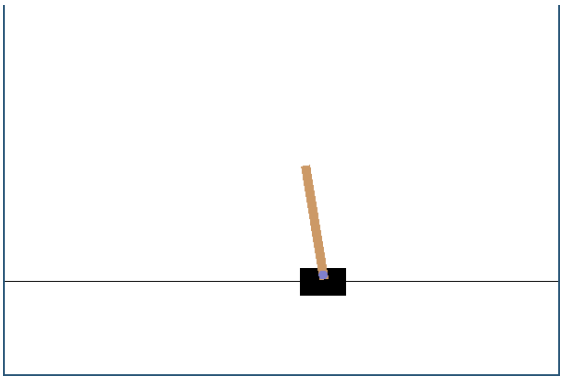

**Figure 4.9:** The goal of the cartpole environment is to learn a control system for balancing an inverted pendulum.

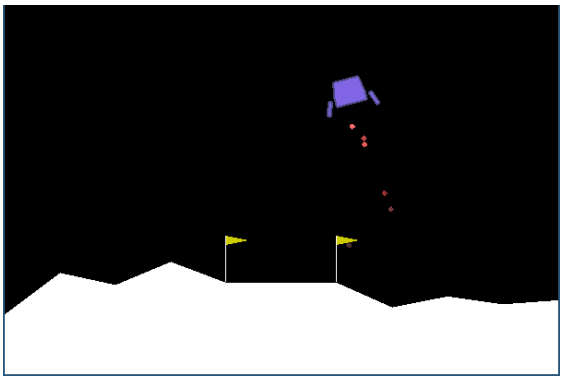

**Figure 4.10:** In the lunar lander problem the agent has to learn how to control the three thrusters and gently land the lander onto the marked platform.

### 4.4.4 Additional environments

For comparison with the previous work by Eriksson et al in their adaptation of LL0 to reinforcement learning some of their custom environments were also included in the benchmark. This includes the berry classification environments(*Berry* and *Noisy berry*), discrete ball and paddle game (*Discrete catch*) and the corridor maze (*Grid*). These environments are only briefly described. For more details on these environments, see the original paper [16].

**Berry and Noisy berry** are two classification problems encapsulated as reinforcement learning environments. The agent is tasked with classifying what berries are safe to eat based on observing the sweetness, sourness and bitterness. The agent can either choose to eat or not eat the berry.

**Discrete catch** is a ball and paddle game where the agent controls a paddle left and right in discrete steps to catch a falling ball. It does this based on observing the balls two-dimensional coordinate and the paddles horizontal position. If the agent catches the ball it receives a positive reward.

**Corridor maze** is a grid based maze game where the agent has to find its way through a small maze with corridors as quickly as possible. The agent occupies a square in this grid and uses the colour of its current floor tile to decide on an action.

# 5

# Results

In this chapter the benchmarking results from the RLL0 neural network and reference network are presented. To show the effect of the growing and pruning rules in RLL0 the results are shown as two versions, one with the rules enabled and the other with them disabled. In most environments the RLL0 hyper parameters that gave good performance were very similar, they are therefore presented in table table 5.1 and any environment specific deviations are listed in their respective section.

**Table 5.1:** The base configuration for the RLL0 neural network in the benchmarking tests. These parameters worked well across most environments, with some needing small changes.

| **RLL0 Parameter** | **Value** | **RLL0 Parameter** | **Value** |
|---|---|---|---|
| Threshold ($\Omega$) | 0.9 | Prune interval ($\Gamma$) | 64 |
| Trainable mean ($t_m$) | *no* | Forced distance ($d_f$) | 0.15 |
| Trainable std ($t_s$) | *no* | Neighbour threshold ($\Omega_n$) | 0.5 |
| Trainable weight ($t_w$) | *no* | Sensitivity threshold ($\Omega_g$) | 0.01 |
| Max patterns ($p_{max}$) | 1000 | ADAM $\beta_1$ | 0.9 |
| Initial learning rate ($\mathrm{lr}_{\mathrm{init}}$) | 0.01 | ADAM $\beta_2$ | 0.99 |
| Learning rate | $\frac{\mathrm{lr}_{\mathrm{init}}}{\frac{12 \cdot t}{t_{max}} + 10^{-6}}$ | PER $\alpha$ | 0.4 |
| Initial epsilon ($\epsilon_{\mathrm{init}}$) | 0.5 | PER $\beta$ | 0.7 |
| Epsilon ($\epsilon$) | $\frac{\epsilon_{\mathrm{init}}}{\frac{12 \cdot t}{t_{max}} + 10^{-6}}$ | DDQN Replay period | 500 |
| Gamma ($\gamma$) | 0.99 | DDQN Policy update | $4 \cdot 10^3$ |
| Batch size | 32 | DDQN Replay budget | $10^4$ |
| Uses ADAM | not MC | PPO buffer size | $10^3$ |
| Activation function | $a$ | PPO num mini batches | 4 |
| | | PPO clip | 0.2 |

The used reference network consisted of two fully connected hidden layers with 64 nodes each, no bias term and a *tanh* activation function. In all training sessions the reference network used SGD with an ADAM optimiser, with the same parameter values as RLL0. This same network structure was used for the actor-critic network when run with the PPO training algorithm. As the training library used to run this network did not support the MC training algorithm, only its PPO and DDQN were used in the comparison.

Due to the high data density in the graphs presented in this chapter they have been aggregated to make the graphs more readable, a change that does not affect the conclusions that can be drawn from these diagrams.

## 5.1 Cartpole environment

The result from running this setup both with the generalisation rules enabled and disabled, and the reference network is summarised in figure 5.1. For this figure it can can be seen that the reference network outperformed both the RLL0 networks in terms of performance. However, looking at the concept node development in figure 5.2, RLL0 used only a fraction of the trainable parameters in the reference network, with 4480 parameters compared to the around 100 in RLL0.

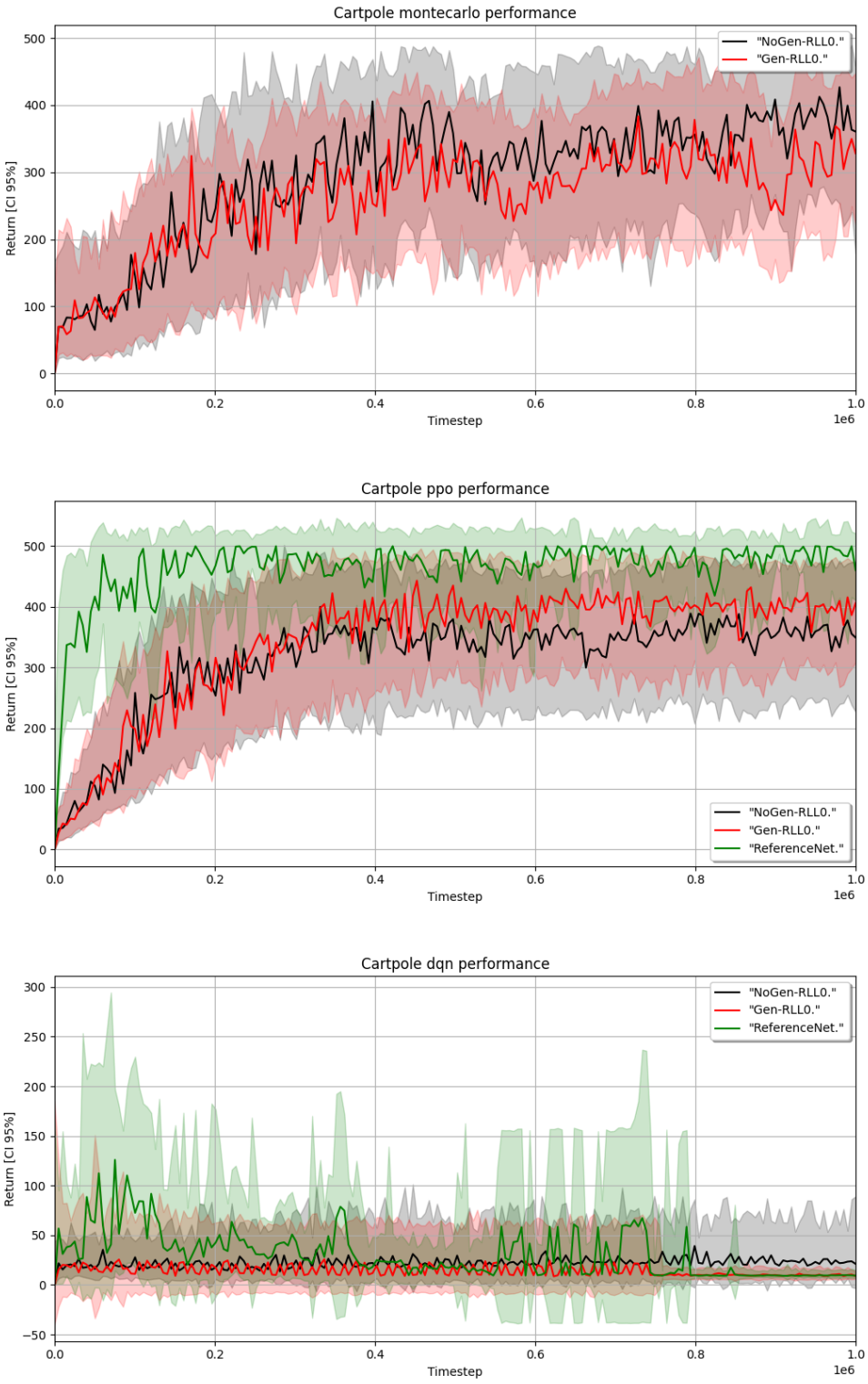


**Figure 5.1:** From top to bottom: The cartpole agent performance when run with the MC, PPO and DDQN network setups. As can be seen, in terms of sheer performance the reference network performed better than RLL0 in this environment.

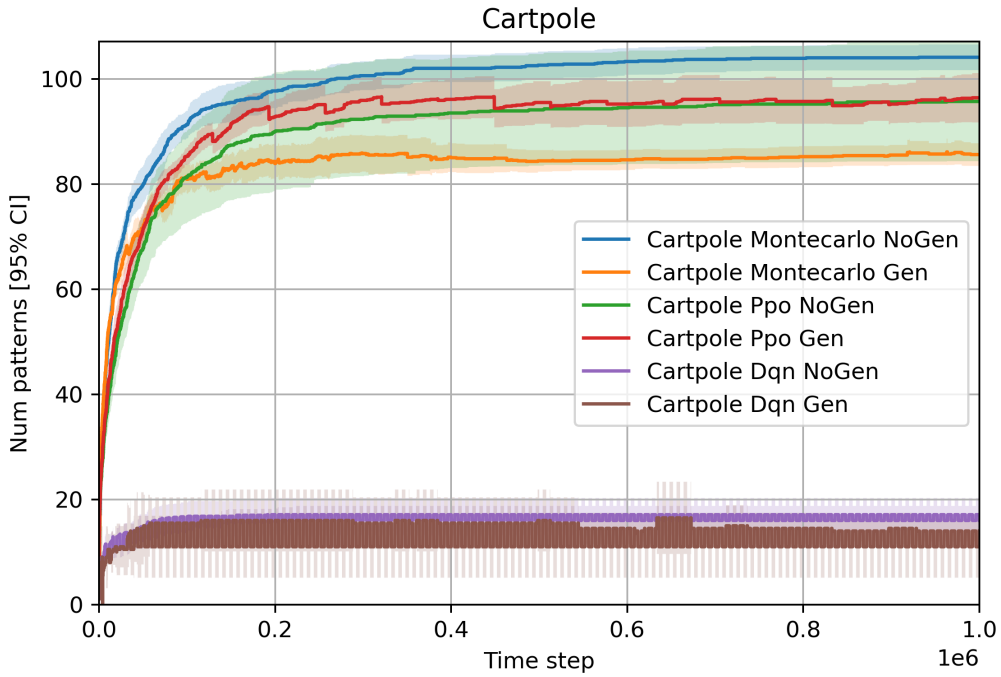


**Figure 5.2:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC, PPO and DDQN training algorithms in the cartpole environment. The flickering seen in the DDQN patterns is because of policy swapping.

Looking at the concept node space of the RLL0 networks it is possible to see the explainable structure of fuzzy neural networks. In figure 5.3 and figure 5.4 the nodes from one network have been projected from the input space onto two selected axes, forming a 2-dimensional plain in the 4-dimensional space. The projection is repeated in subplots for each of the two actions in the action set. These subplots contain a cloud of circles with a number, each representing a projected concept node's normal distribution $\mu$ projected onto the plain. The circle radius of these points represent their Q-value estimation relative the other nodes. A larger radius translates to a larger relative Q-value estimation. Around each point is also a transparent ellipse which represents the concept node's activation space given the used activation threshold $\Omega$. The more opaque ellipses represent the greedy actions with respect to the other subplots for inputs in that concept node's receptive region.

Looking at the two projections of a network from the RLL0 without generalisation rules in figure 5.3 for the (position, velocity space) projection, it can be see that the selected action quite well corresponds to what one would expect from this environment. In the top projections the greedy action is alternating between pushing the cart from the left to right as it moves across its rail. Here it is also possible to see why the agent did not perform too well. In to bottom left of the two subplots the dominating action is the 'Push right', which it really should not be as it would be better to move the cart back towards the right side to avoid going off the edge.

Instead looking at the two graphs in figure 5.4 of the (angle, rotation rate) projection one can see that the agent has learnt to balance the pole. By moving the cart base left when the pole angle is to large and going clockwise, and to the right when instead going counter clockwise. In the top right corner of the left subplot and bottom left corner in the right subplot are outliers, which the agent did not value as good, indicated by their small circle radii. As these concept nodes come from the non-pruning network they were kept by the network even though they do more harm than good.

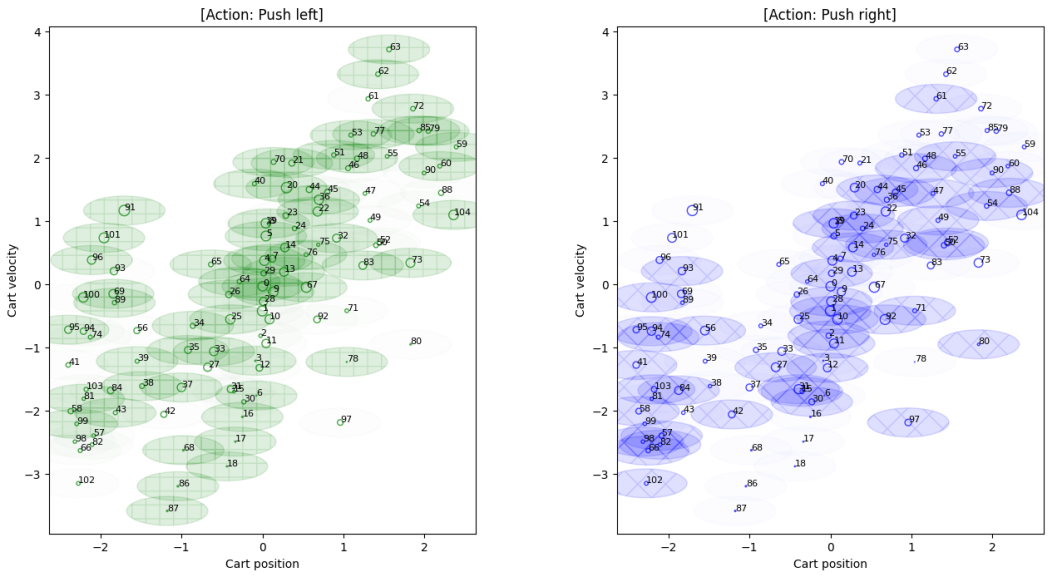


**Figure 5.3:** The projection of network concept nodes from a RLL0 trained on the cartpole environment without the generalisation rules.

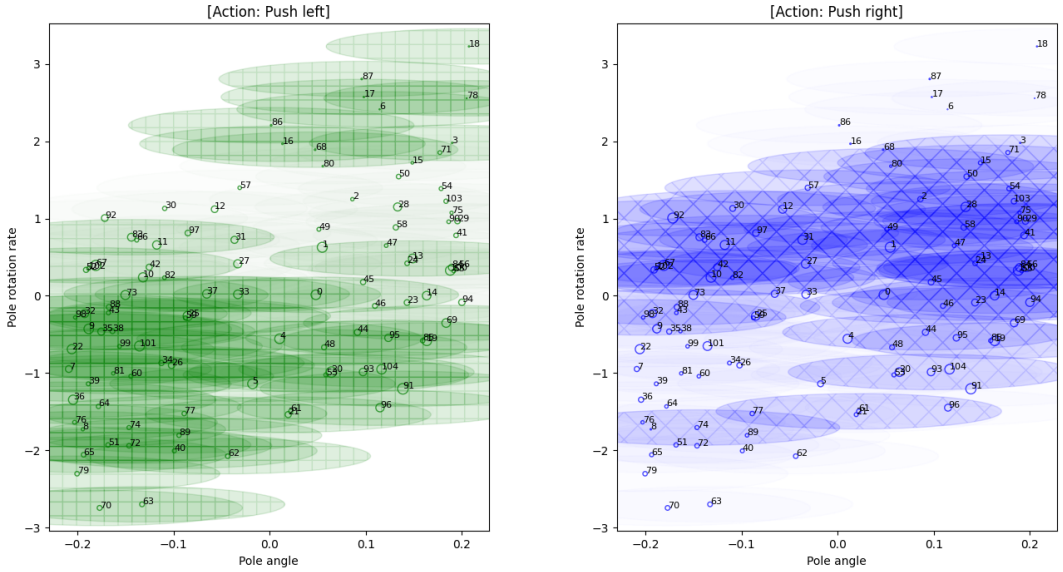


**Figure 5.4:** Another projection plain of network concept nodes from a RLL0 trained on the cartpole environment without the generalisation rules.

## 5.2 Lunar lander environment

The lunar lander environment used the base RLL0 configuration for its agent network, with the exception of setting the activation threshold $\Omega$ to 0.5. This threshold was set considerably lower value in this environment due to the number of generated patterns. This low threshold means that the network's estimation resolution is greatly reduced and in turn is reflected in its poor performance compared to the reference network in figure 5.5. As the results for the DDQN algorithm made no progress regardless of the used network, it has been omitted from the results. In this environment the reference network outperformed both RLL0 configurations, which were unable to learn any useful policy.

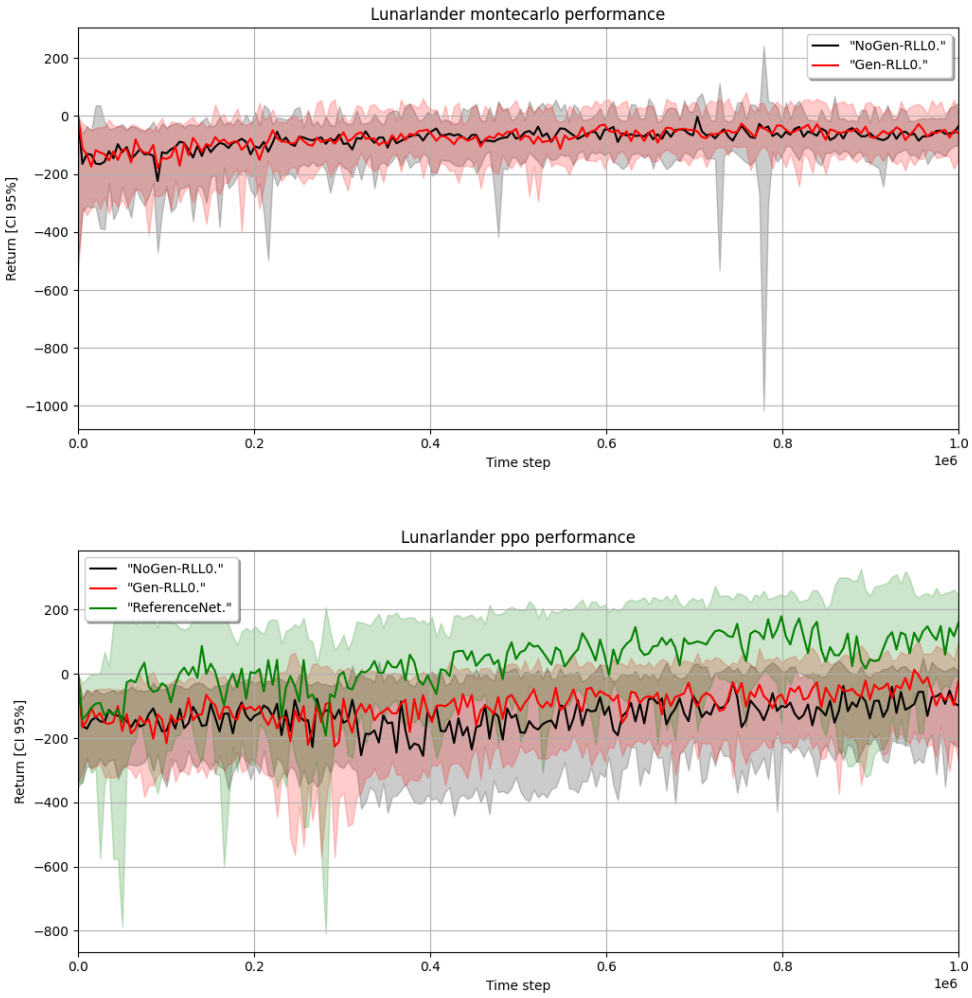


**Figure 5.5:** From top to bottom: The lunar lander agent performance when run with the MC and PPO network setups. In this environment the reference network performed far better than RLL0.

The concept node growth for these networks can be seen in figure 5.6, which shows the number of patterns rapidly rising as the agent explores the large input space. In this environment both with and without the generalisation rules performed about the same; however, here the pruning rule becomes very obvious. Every $64^{th}$ completed episode the network runs its generalisation rules and prunes a significant portion of the concept nodes that were not used.

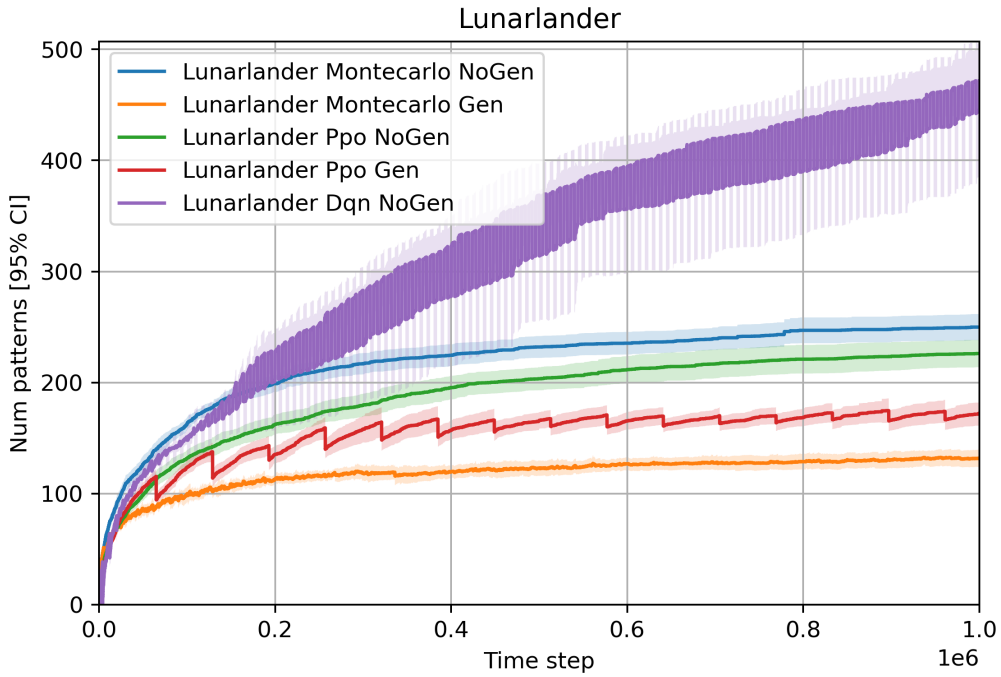


**Figure 5.6:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC and PPO training algorithms in the lunar lander environment. As seen, significantly fewer concept nodes are used when the generalisation rules are enabled.

## 5.3 Animat environments

All of the animat environments were run with the base RLL0 parameters and their result in the Animat #0 environment compared to the reference network can be seen in figure 5.7. And again, the DDQN results are omitted since none of the networks managed to learn the environment in the designated training time.

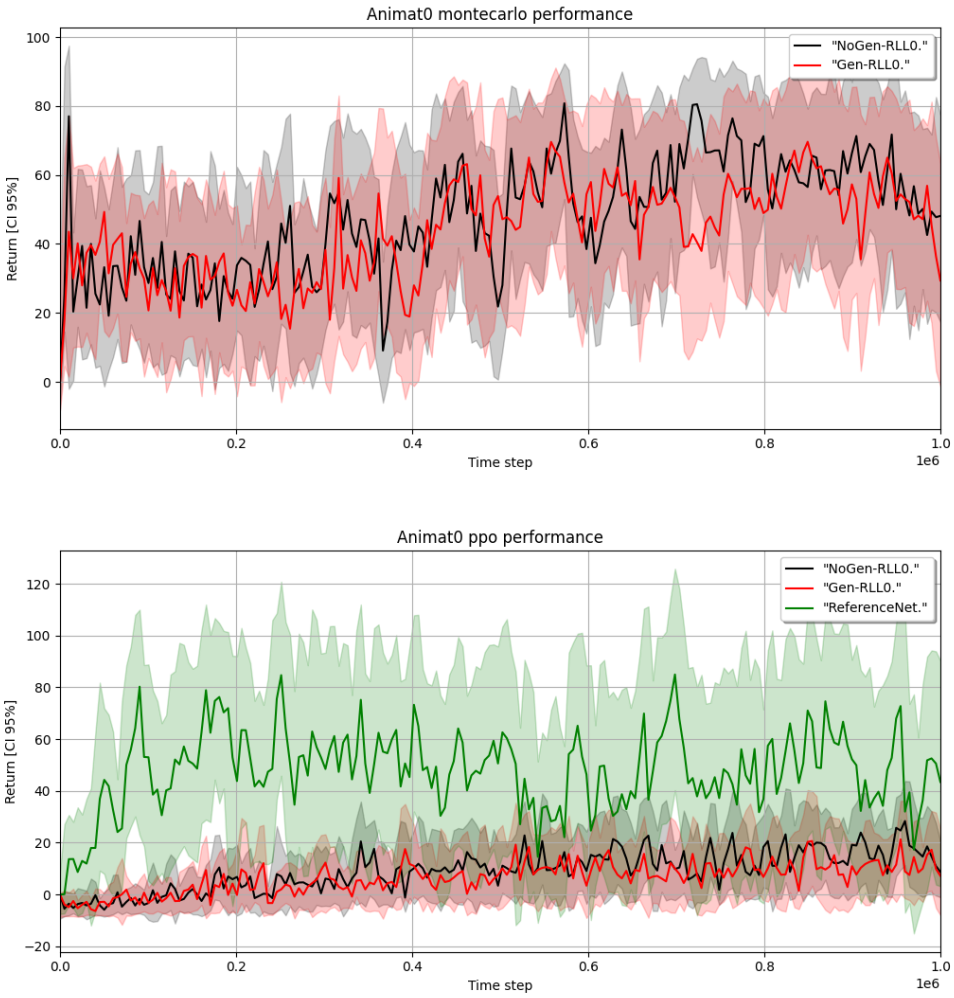


**Figure 5.7:** From top to bottom: The animat #0 agent performance when run with the MC and PPO network setups.

As seen in the diagrams for Animat #0, the RLL0 neural network had poor performance with the PPO algorithm, yet performed on the same level as the reference network when running the MC training algorithm. The same result can be seen in the diagrams for Animat #1 in figure 5.8; however, in this case the reference network could not learn any working agent policy.

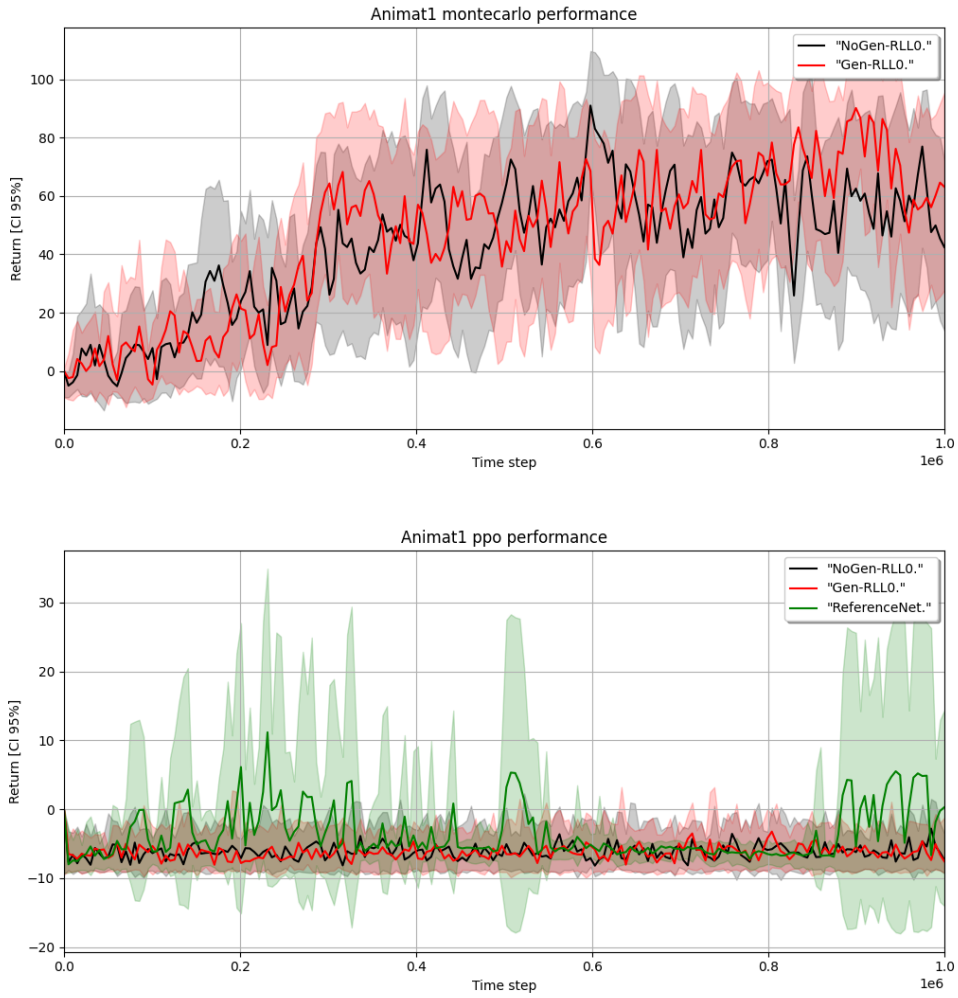


**Figure 5.8:** From top to bottom: The animat #1 agent performance when run with the MC and PPO network setups.

Just as in the previous environments the number of concept nodes used by the RLL0 networks is far fewer than the number of trainable parameters in the reference network, as seen in figure 5.9. Here the pattern development over the all training sessions show that an animat early finds the few most essential concepts and learns a policy for this environment. In this figure one can also note that the generalisation rules contribute to adding more concept nodes than when disabled, which is due to the forced node spacing threshold. The same development can be seen for the patterns in second animat environment in figure 5.10.

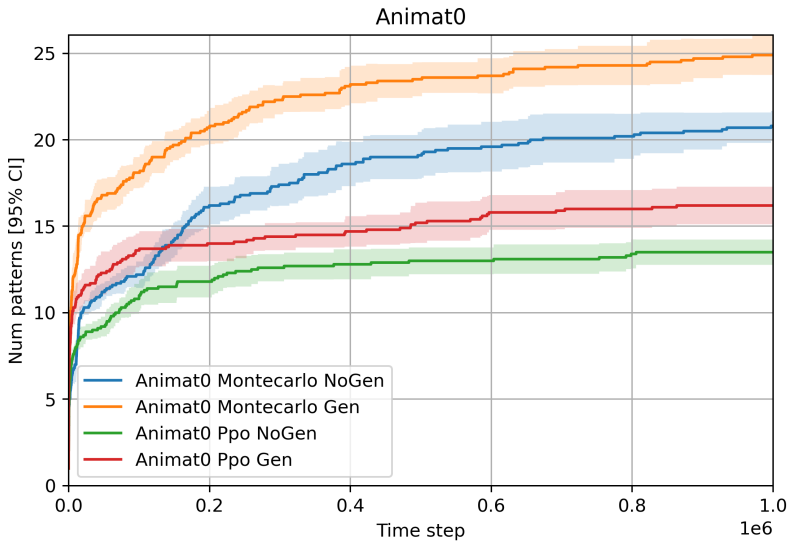


**Figure 5.9:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC and PPO training algorithms in the animat #0 environment.

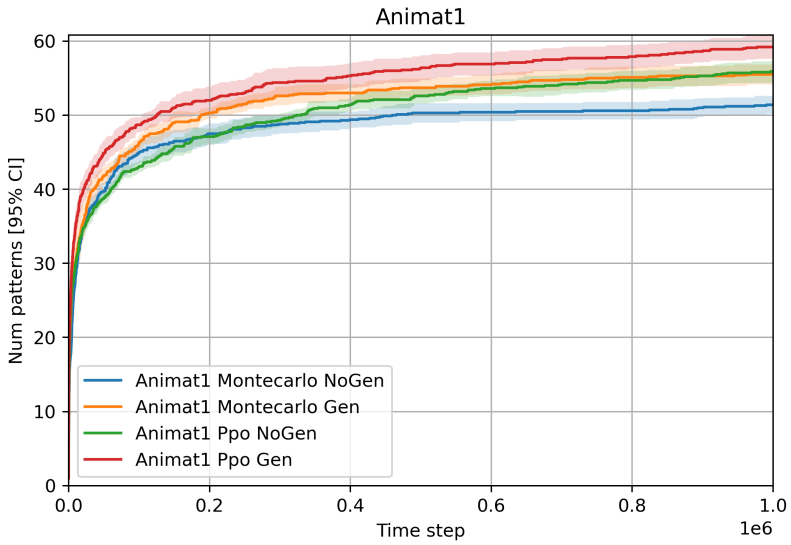


**Figure 5.10:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC and PPO training algorithms in the animat #1 environment.

Since only RLL0 had a training setup that allowed for multiple agents, there is no performance data for the reference network in the Animat #2 environment. However, the results for the tiger animat using RLL0 can be seen in figure 5.11 and the antelope in figure 5.12. Just as in the other animat environments the MC algorithm gave better results compared to its performance with the PPO algorithm. The patterns for these two agents can be found in figure 5.13 for the tiger networks and figure 5.14 for the antelope.

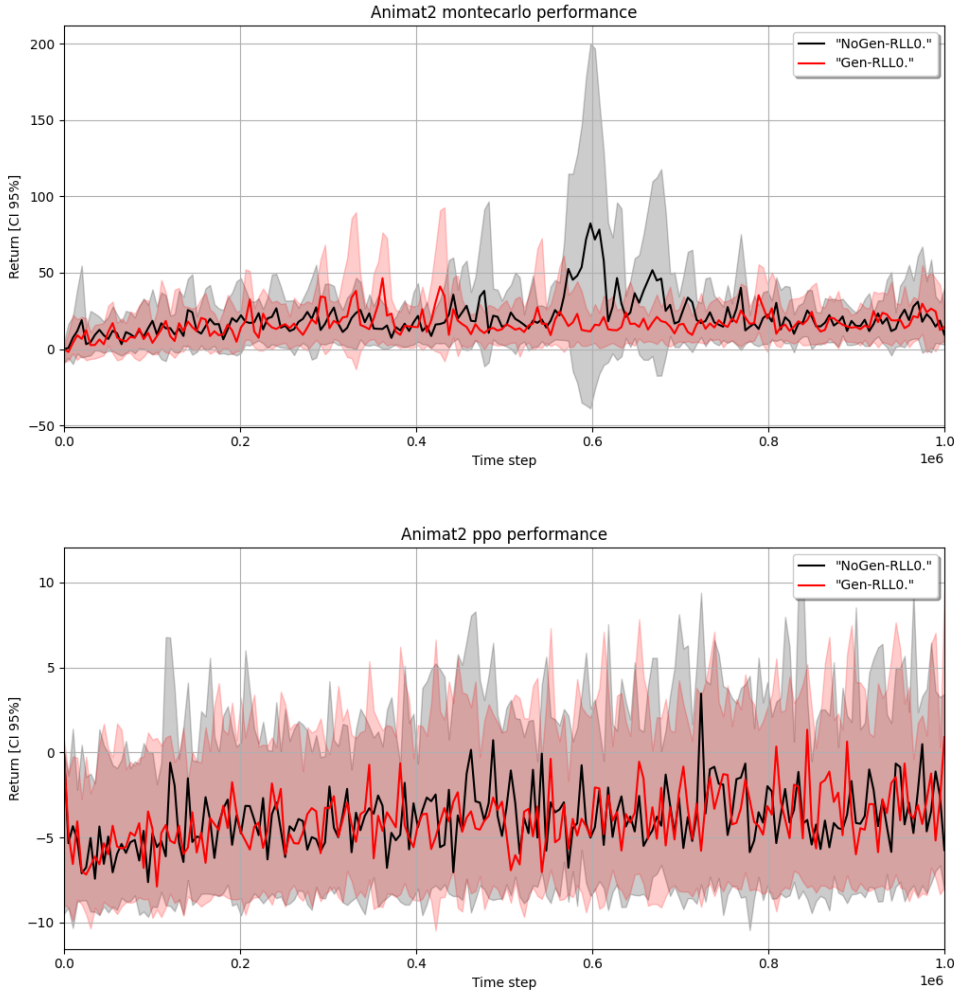


**Figure 5.11:** From top to bottom: The animat #2 agent performance when run with the MC and PPO network setups as the tiger animat.

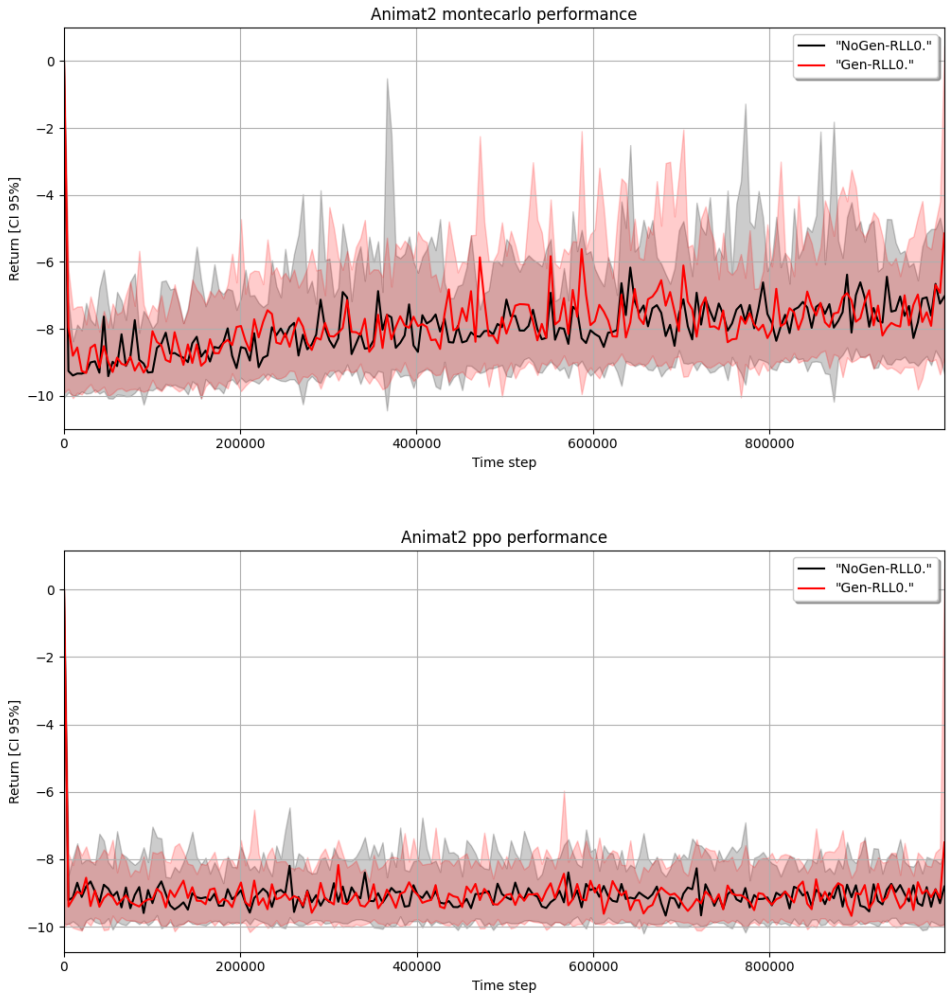


**Figure 5.12:** From top to bottom: The animat #2 agent performance when run with the MC and PPO network setups as the antelope animat.

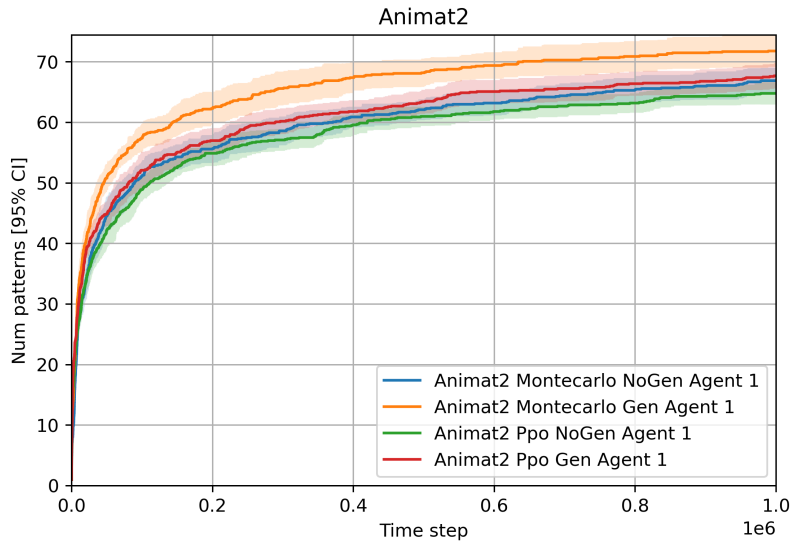


**Figure 5.13:** The number of patterns used by the tiger RLL0 agent network when not using the generalisation rules, when using them and with the MC and PPO training algorithms in the animat #2 environment.

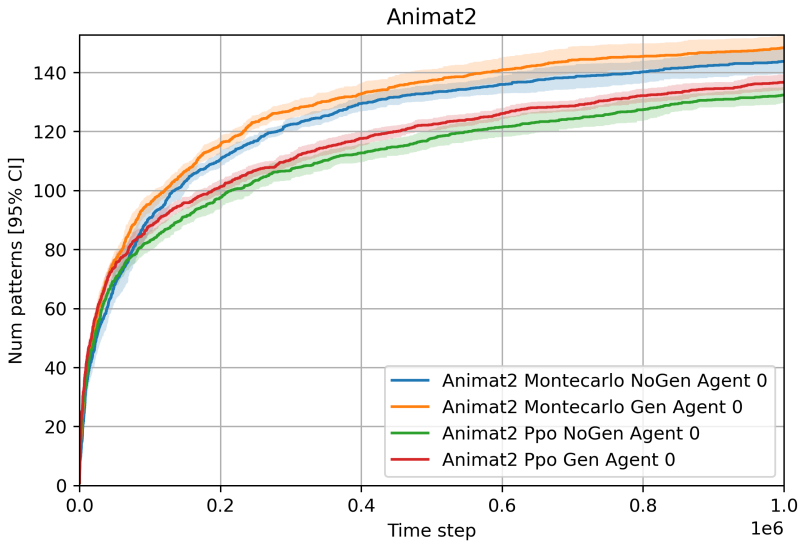


**Figure 5.14:** The number of patterns used by the antelope RLL0 agent network when not using the generalisation rules, when using them and with the MC and PPO training algorithms in the animat #2 environment.

By projecting the RLL0 concept nodes a RLL0 network trained on the Animat #0 environment into figure 5.15, it is possible to see how the agent reasons when balancing eating and drinking. The agent has learned to eat once it is about half way between being hungry and thirsty, which is a compromise between not dying and maximising the return. Looking at the actions for following the smell of water and grass, the agent has learned to stop eating when full and start looking for water instead The same happens when it has filled up on water and it will start with looking for grass again. In the Idle action we can even see that the agent waits a bit before eating, so as to maximise the reward from eating and drinking, which happens when the homeostasis value is low.

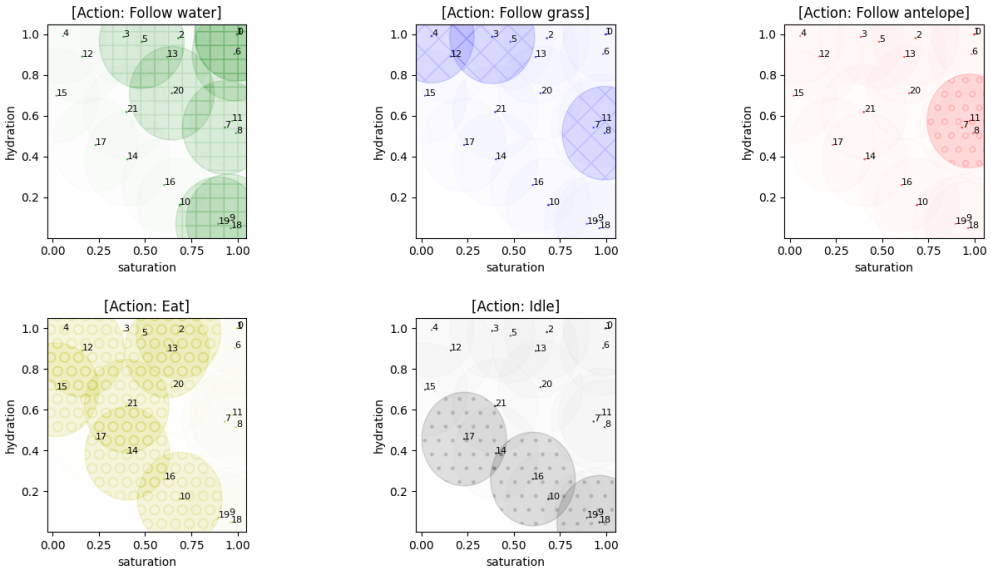


**Figure 5.15:** The projected concept nodes from an agent trained on the animat #0 environment. The agent has learned to alternate between eating and drinking to stay alive and maximise the objective function.

## 5.4 Additional environments

In the additional environment by Eriksson et al the the RLL0 network used the base configuration, with the change of setting the activation threshold $\Omega$ to 0.95. The results from training RLL0 compared to the reference network can be seen in the figures for berry 5.16, noisy berry 5.17, grid 5.18 and discrete catch 5.19. Apart from in the discrete catch environment, the RLL0 networks managed to be on par with the reference network on all of these environment and use significantly fewer parameters, as seen in figure berry 5.20, noisy berry 5.21, grid 5.22 and discrete catch 5.23.

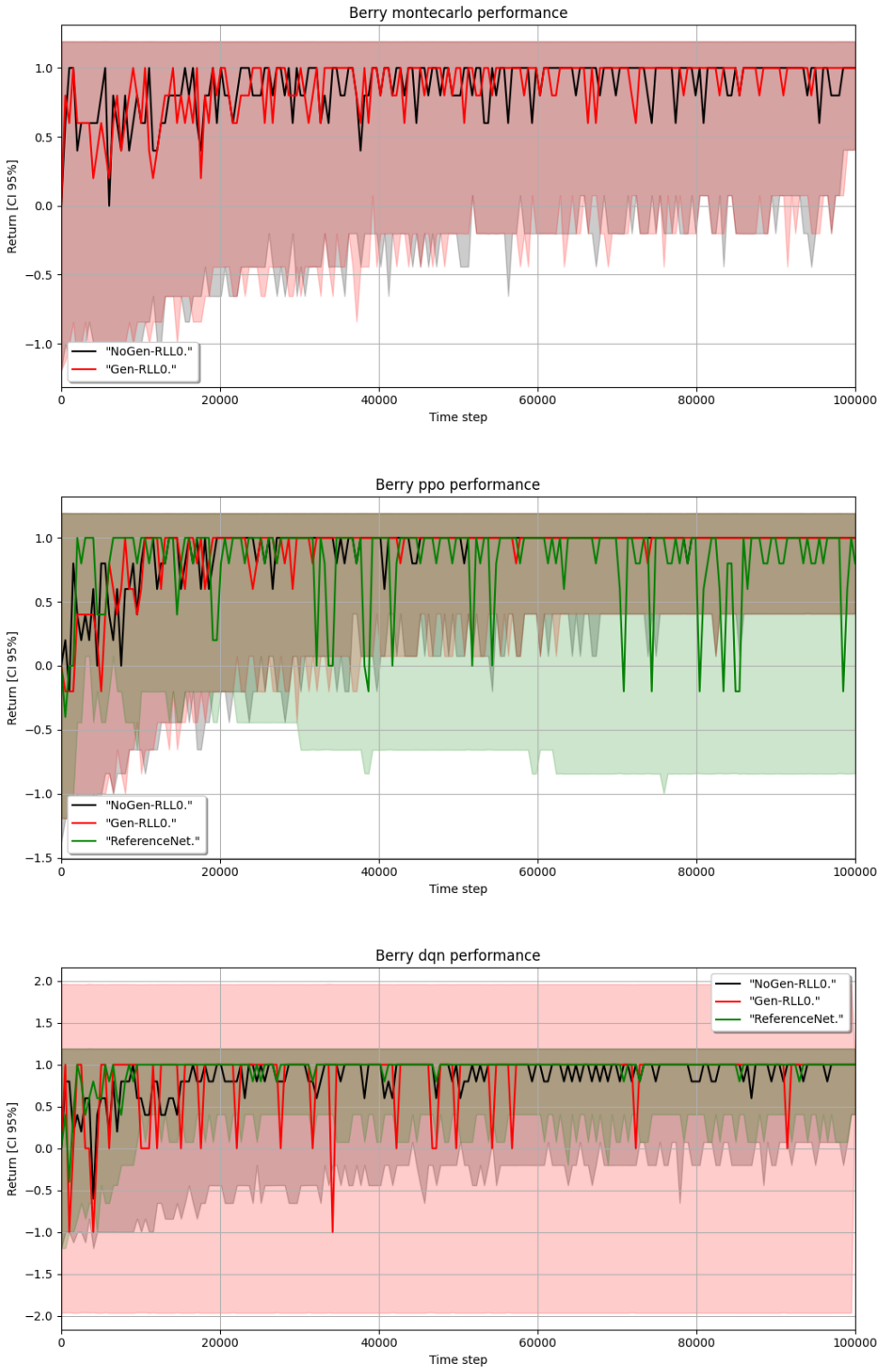


**Figure 5.16:** From top to bottom: The berry agent performance when run with the MC, PPO and DDQN network setups.

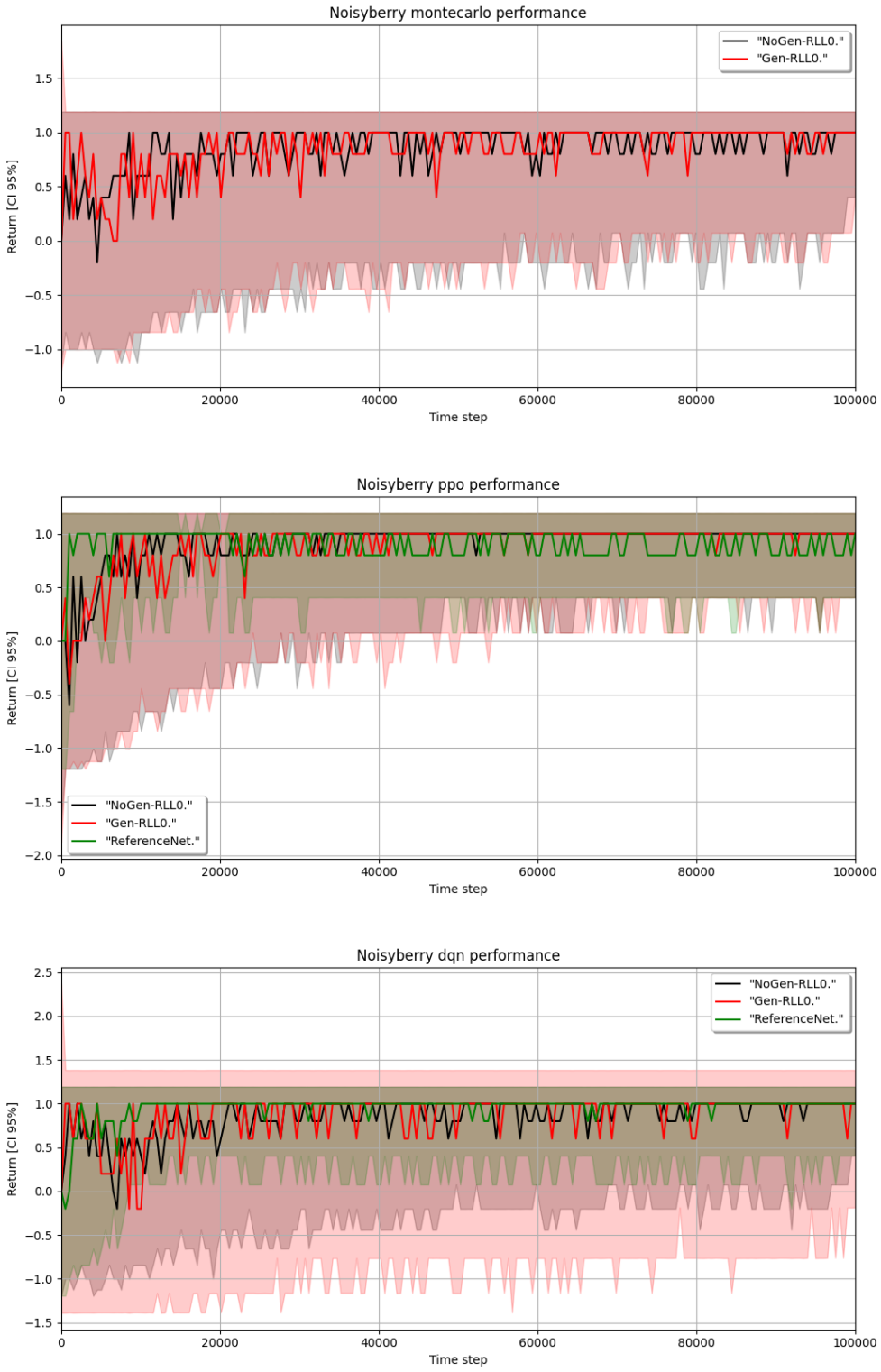


**Figure 5.17:** From top to bottom: The noisy berry agent performance when run with the MC, PPO and DDQN network setups.

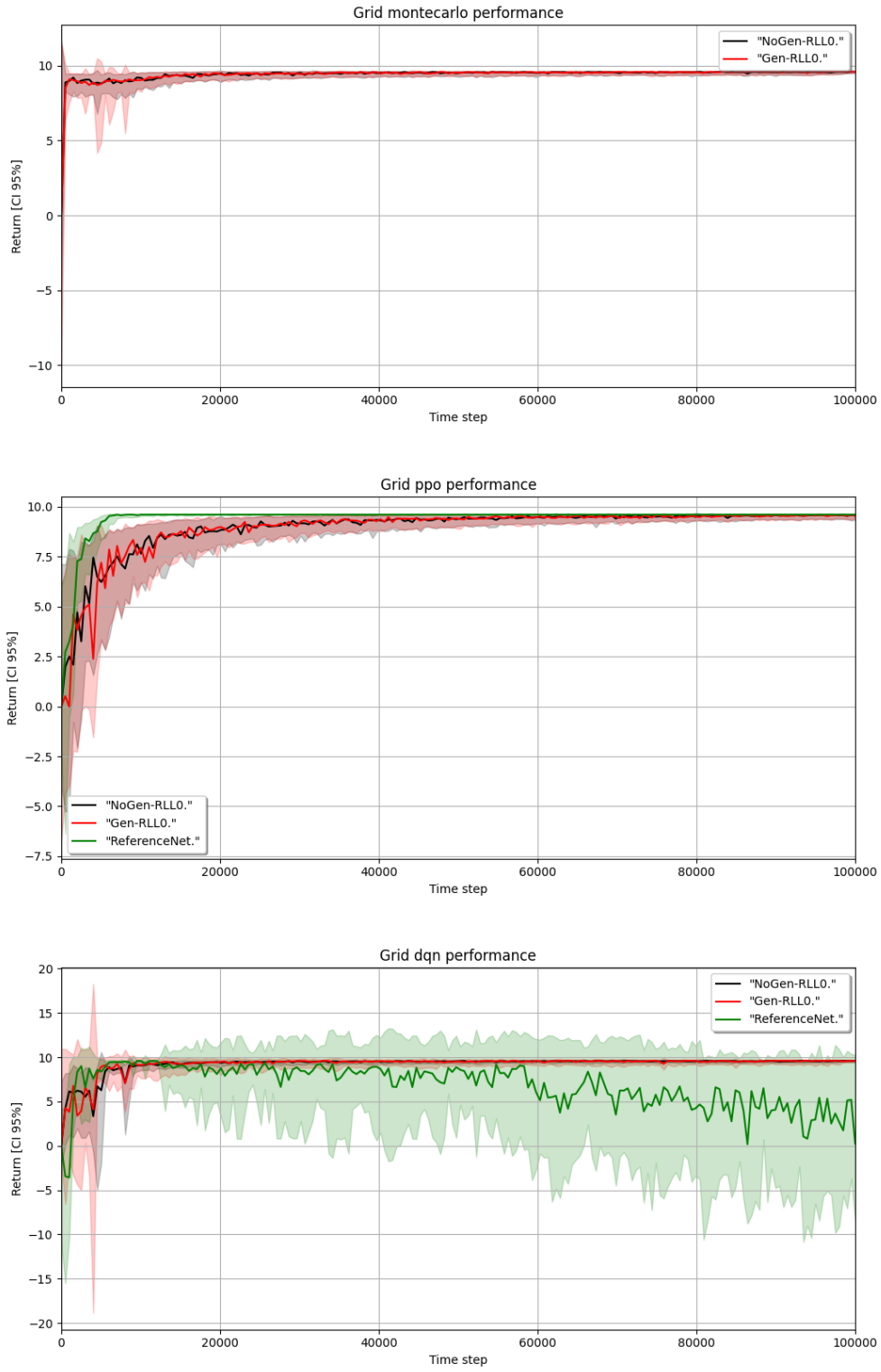


**Figure 5.18:** From top to bottom: The grid agent performance when run with the MC, PPO and DDQN network setups.

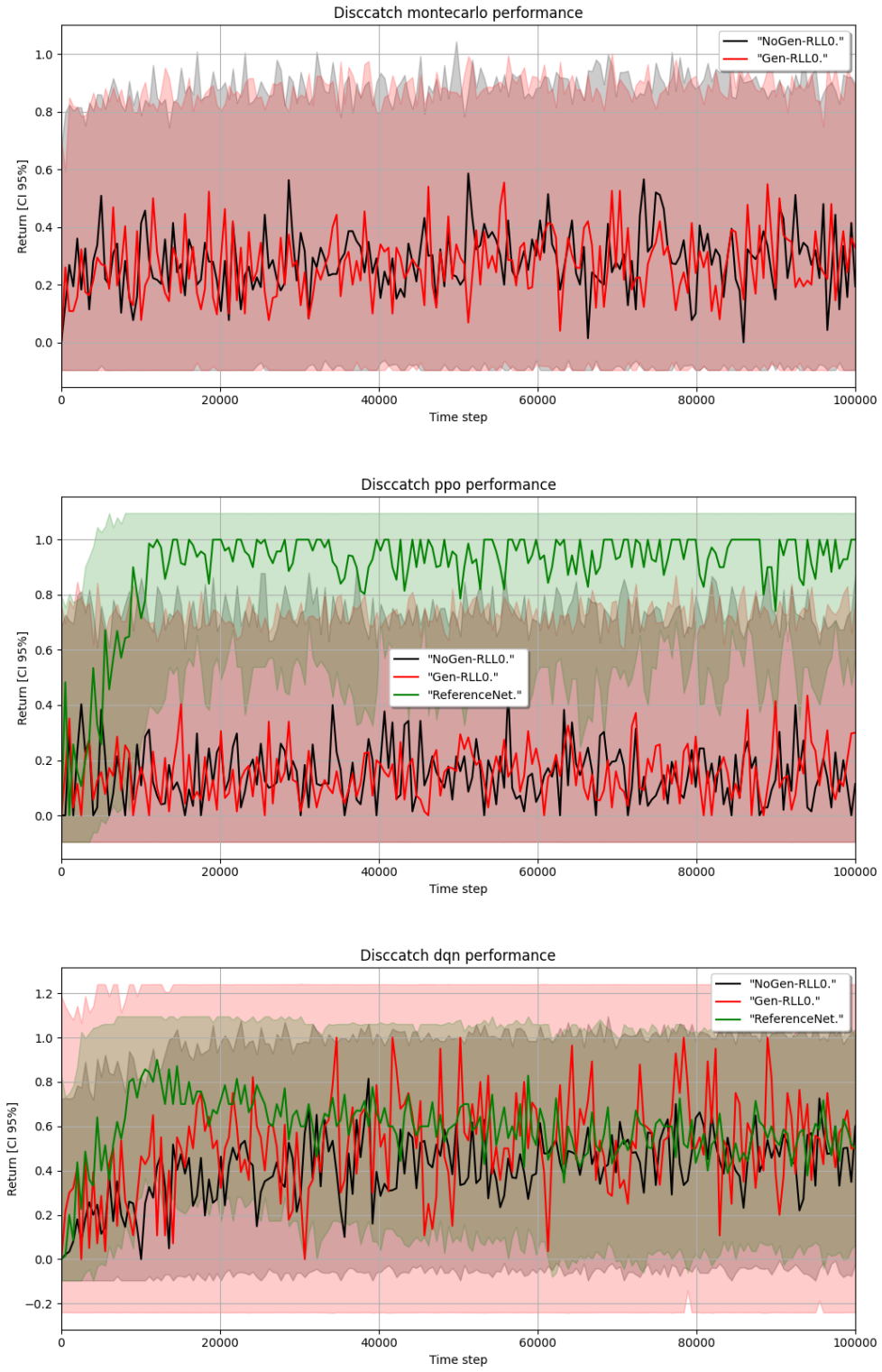


**Figure 5.19:** From top to bottom: The discrete catch agent performance when run with the MC, PPO and DDQN network setups.

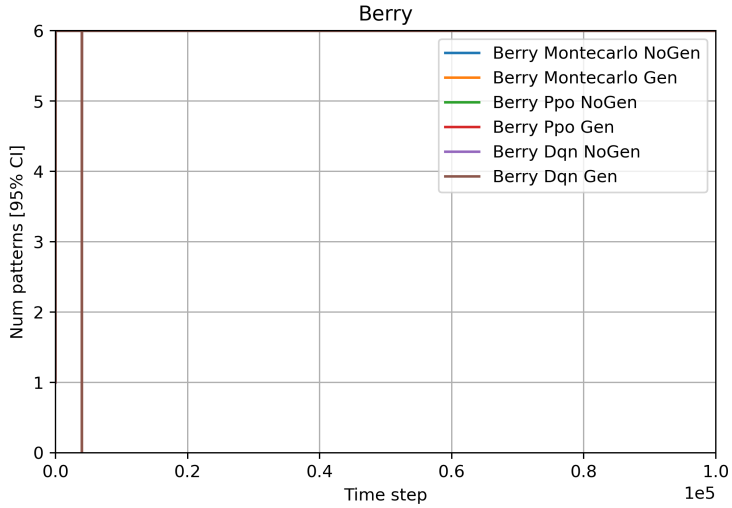


**Figure 5.20:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC, PPO and DDQN training algorithms in the berry environment.

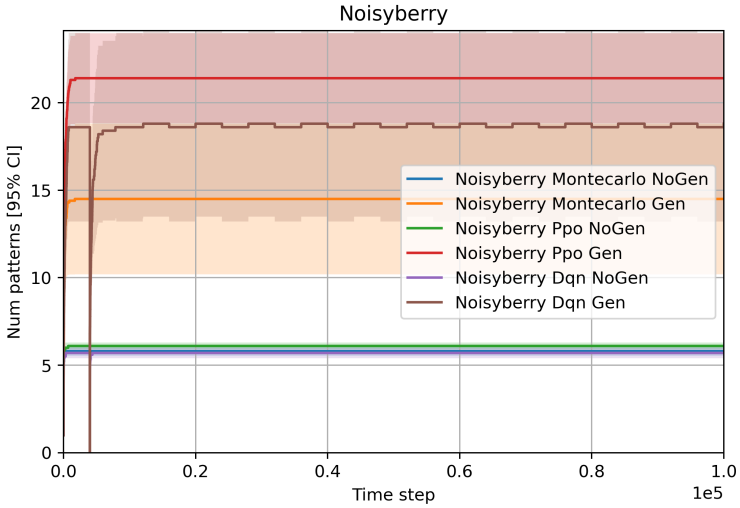


**Figure 5.21:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC, PPO and DDQN training algorithms in the noisy berry environment.

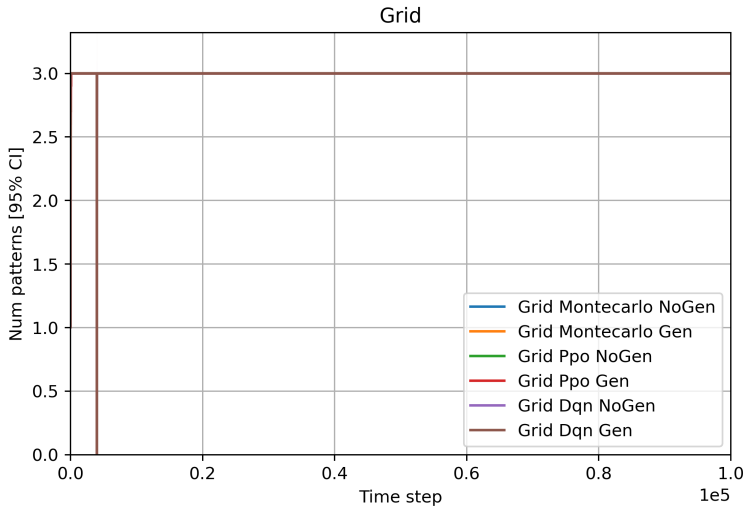


**Figure 5.22:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC, PPO and DDQN training algorithms in the grid environment.

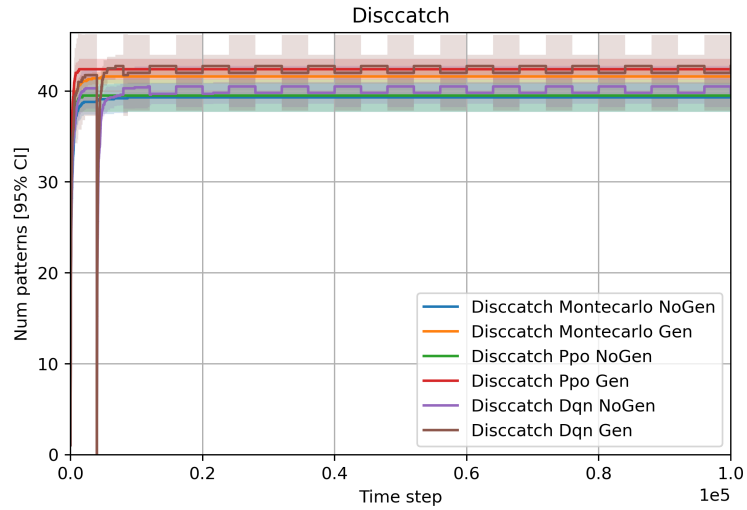


**Figure 5.23:** The number of patterns used by the RLL0 network when not using the generalisation rules, when using them and with the MC, PPO and DDQN training algorithms in the discrete catch environment.

While these results are not as good as those presented by Eriksson et al with their reinforcement learning LL0 implementation, as seen in the previous results, RLL0 gave better performance in other general reinforcement learning environments that the implementation by Eriksson et al could not learn.

# 6

# Discussion

As seen in the results chapter, the RLL0 performance compared to the reference network varied between the tested environments and chosen training algorithm. In the following chapter the RLL0 performance and some additional network rules that were omitted from the results chapter are discussed and related to possible future work.

## 6.1 Overall performance

The benchmarking results indicated that RLL0 was able to learn to some degree in all the tested environments. In most of them its performance was comparable with that of the reference network, though lower and using far fewer trainable parameters. This performance dip differed significantly between the tested training algorithms, to the extent that some environments could not be learnt with a specific algorithm. In general MC and PPO gave the best training performance for RLL0 performance across the control and animat environments, whereas DDQN without the generalisation rules did better in the other environments by Eriksson et al.

From the results in the control system environments, *lunar lander* and *cart pole* environments, we see that RLL0 did not perform as well as the reference network. This performance can be slightly increased by raising the activation threshold; however, at the cost of more concept nodes. The results presented here are therefore a trade off between concept nodes and performance. Instead looking at the animation environments, the RLL0 network performance was at least as good or better and using as few as 20 concept nodes in the Animat #0 environemnt. In the animat #1 environment the RLL0 based agents were even the only to learn a working policy. Though, in this environment more concept nodes were needed due to the larger network input and output dimensions and environment complexity.

## 6.2 Concept nodes

A major issue with RLL0 in its current form is its generalisation to handling new inputs, other than those seen during training. In its current form the network can report any unknown inputs, which could be beneficial, and continues with the top active concept node. However, it is unlikely that this gives a good estimate, as the purpose of the concept nodes is to estimate based on the inputs in its receptive region. This tight dependence between a concept node's receptive region and estimate accuracy is not ideal for a network that is trained once and then used to generalise over unseen input. Instead, the RLL0 neural network is more suited for a life-long learning setting. This would allow for continuously generating and pruning based on the its current environment. RLL0 can even enhance this further by setting a node budget to limit its size. While the RLL0 implementation in this report did not alter its internal behaviour when reaching the node limit, it would be of interest as future work to introduce some automatic network compression or new concept node handling rules for a similar purpose.

As seen in the results chapter, this fine grained node control also gives additional benefits in terms of inherent network explainability. While the concept node projection method shown in the results chapter can extract this information in an intuitive way, it becomes less useful for visualising abstract data such as image pixels or other less comprehensible input values. In the tested environments this was not an issue. The high level planning problems in the animat environments and their easy to understand input parameters makes these biological simulations a far more ideal setting for this method.

In the animat environments it is also easier to define a subset of the input space of interest. This subset of concept nodes can be directly extracted from the RLL0 network and visualised through projection to limit the shown information. Nodes can even be select based on their performance or greedy action to find a subset in the input space leading to a specific behaviour. If extended to live plotting during training it can show changes in an agent's behaviour as it learns. A feature like this could be of particular interest for life-long learning, as the agent is constantly adapting and improving its policy based changes in its environment.

## 6.3 Network design and trainable parameters

The choice to design RLL0 with a single hidden layer was due to the issues encountered by Eriksson et al when training their multi-layer reinforcement learning adaptation of LL0. That network formed new nodes when the prediction errors went above a fixed threshold. Using that as network growth trigger may have worked in their developed environments; however, are not very practical in most other environments. With a single hidden layer effort was instead spent on getting the network to learn these more complex environments using a concept node based network. Generalising into additional layers is, after all, an optimisation technique to reduce the number of concept nodes, increase network explainability and training efficiency. As the gathered results indicate that RLL0 is able to learn with this single layer, the next step would be to reintroduce both multiple levels of concept abstraction.

However, even though RLL0 is able to learn reinforcement learning problems, there are still issues regarding the trainable parameters in the matching distributions. For the same reason as estimation errors are unreliable growth triggers in reinforcement learning, so is updating the membership distribution parameters base on the same error. This causes both the output estimation and the concept node's receptive region to change. Because of the receptive region's local relevance in the input space this placement change cause further estimation errors in later time steps. This leads to a sort of *wandering* effect on the concept nodes, see figure 6.1, that quickly moves the nodes outside the practical input space and a surge of created concept nodes to fill their absence.

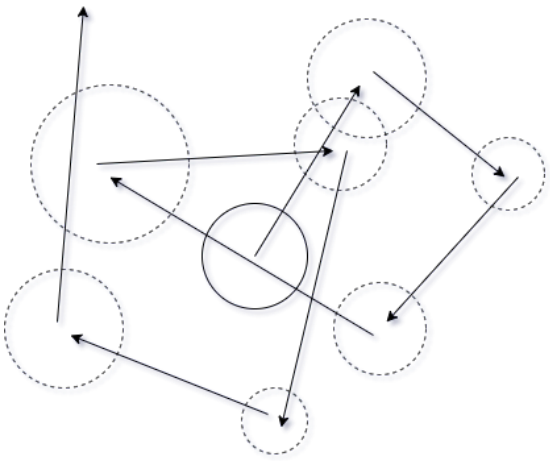

**Figure 6.1:** An example of a concept node wandering in a 2-dimensional input space. The matching distribution is represented by the circle, based on its mean and standard deviation. In each update the estimation error moves the trainable parameters in the direction of the arrow.

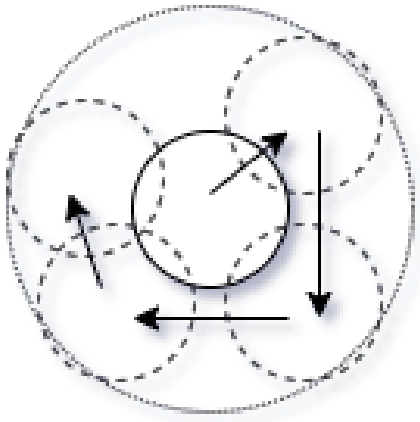

**Figure 6.2:** By anchoring the node within a radius from its original placement in the input space its wandering can be restricted. It does not; however, bring any performance improvements.

Throughout RLL0's development several attempts were made to suppress this issue. One of the attempts imposed a restriction on the parameter change relative to its initial value. This limitation functions as an anchor point for the matching distributions as shown in figure 6.2. While it does mitigate the node centres from wandering, it does not increase the network performance compared to just freezing the parameters. Even if the update are limited to only be used when the estimation errors are small, these changes add up and greatly reduce the performance. Compared to updating the membership functions with SGD, using more structured growing and pruning rules from fuzzy networks to control parameter gave far better results.

## 6.4 Growing and pruning rules

From the concept node growth diagrams in the results chapter we see that RLL0 is able to dynamically adapt the number of nodes in the network. In some small input spaces it can be covered with a single digit number of nodes whereas in the larger lunar lander environment it used hundreds. This difference in node count shows both the benefit and issue that arise from using a fixed threshold for concept creation. A single threshold value uniformly across the entire input space eventually leads to the estimation resolution becoming either to low or high in certain regions. Ideally this could be improved with adapting the resolution according to node performance, lowering it to compress poor performing parts and raising in high performing parts to learn an even better behaviour. This adaptive growing rule would likely be problem specific and could possibly use a heuristic function to deduce the needed resolution, as it otherwise would be hard to apply a general rule across all reinforcement learning problems with varying requirements.

To control the network resolution both Eriksson et al and Carlström et al used, among others, an intersection rule used to find more generalised concepts from several overlapping nodes. This same rule was tested with RLL0 and its inclusion does not necessarily negatively affect the network performance; however, it leads to far more concept nodes due to the smaller receptive regions of intersections. A similar rule with better performance in the cartpole environment was to take the union of two neighbouring concept nodes once their estimations differed more than some high percentage. By merging the better performing node over the less performing concept does not negatively affect the performance in cartpole and thus works as an efficient compression rule. Because this is a problem specific compression rule and does not work in the other environments it was not included in the results chapter; however, compression rules like this could be adapted to problem specific implementations.

Another explored rule for the sum based activation function $a'$ was to update the concept node importance weights by estimating what inputs correlated with high return predictions. This was done by clustering the concept nodes into neighbourhoods and measuring the internal overlap of the membership functions. By comparing the overlap of the top performing nodes with the worst performing nodes in a neighbourhood, the goal was to find what inputs were associated with a high predicted output. From this comparison the nodes had their importance weights adjusted to be less sensitive to least contributing inputs. In turn this meant concept nodes got a larger receptive region and thus fewer nodes were needed in the network. While this method was successful in identifying the important inputs in some environments, translating it to adjustments within that neighbourhood to improve the performance proved more challenging. Changing the already learned concepts meant the previously learnt estimations got less accurate. This in turn cascaded into new neighbourhoods and effectively forced the network to relearn from scratch.

The eventually selected rules were adapted from fuzzy classification and regression tasks. These rules could be better generalised across the environments compared to the other problem specific rules; however, they add additional non-trivial and problem specific training hyper parameters. Especially the pruning rule can cause problems in certain environments if essential concept nodes are not frequently used throughout all episodes. While the pruning could be limited to nodes used during that time span, it would still eventually lead to stray concepts nodes that need to be pruned. More work with these generalisation rules over a broader set of environments is needed and is also the currently considered the most important for improving RLL0 as a whole.

## 6.5 Introducing one-shot learning

The one-shot property in LL0 was absent in RLL0 as in reinforcement learning it is not possible to learn a definitive estimation from a single experience. Hence, what is experienced in one time step does not necessarily accurately reflect what can be expected from the same state later on. A method more suited for this type of learning is *few-shot learning*, which learns quickly from very few experiences. In published work there are multiple approaches to achieving few-shot learning, with one being *meta-learning* that makes changes directly to an agent's training process using a supervising agent. An example of such a network is the *meta stochastic gradient decent*. It learns how to control training parameters for SGD by training how to train networks in a domain of tasks [28]. As this technique can be added in a stand alone fashion to some existing network, it could potentially be used with RLL0 for speeding up animat training. However, it or some alternative would have to be adapted for RLL0's dynamic architecture.

## 6.6 Handling larger dimensions

The current implementation of RLL0 does not scale well once the dimensions increase. The number of needed concept nodes in a problem is directly connected to the size of the input space. To combat this issue in domains such as text it could be of interest to look into using techniques such as embedded vectors to lower the dimensional input and generate a more suitable input representation. Using embedded vector functions to map input space to an abstract vector space and the squared euclidean distance as a form of membership function to classify points was used in the *prototypical networks* with success [29], and could be of interest for RLL0 to adapt to larger state spaces in similar environments.

## 6.7 Virtual animal simulation

The animat agent representation presented in this report proved to be a working prototype model for mimicking the most fundamental animal behaviour. Representing the agent state by its status values, relative sensory input and an action set to follow smell gradient introduced more realistic and observable agent movement than what can be achieved by simple grid based actions. As all these observation elements are relative to the agent, the simulated world and its animal population can be expanded without any retraining, as long as the original conditions for foraging and population diversity remain the same. This makes the environment suited for exploring learning across generations with methods such as *genetic programming*, as was done by Strannegård et al [5].

While the scope of the tested animat environments in the benchmark was limited to basic needs and animal species, the simulation could be expanded to a more realistic representations and diverse ecosystem. The presented approach showed that the agents could learn to mimic the behaviour of different species through a multi agent environment and give a higher resolution to the simulations by representing each animal as its own trainable agent. The action set based on following smells also gave promising results, as focus is still kept on learning a policy for abstract planing rather than motor based skills. In future work this could be futher expanded on to include other senses to make the simulations even more realistic.

## 6.8 Simulation ethics

While far from being applicable to the developed simulation environment in its current state, if this type of animal environment is adapted to a more general purpose it could also be used to model human behaviour. This includes simulating behaviour in economics, infrastructure, traffic and human/animal gathering in cities and buildings. Such simulations could test and collect data from the behaviour of an agent within some set objective when exposed to a scenario of interest. As such it is not without risk of it being used for unwanted purposes and simulate an ill-intended scenario. Still, the same simulations could also be used for building evacuations based on learned behaviour on an individual level.

# 7

# Conclusion

Based on the results from the benchmark the performance of the developed RLL0 neural network, compared to the reference network, did not meet the same performance level in all environments. RLL0 was; however, able to learn in all environments using only a fraction of the trainable parameters in the reference network. It was also flexible and generic enough to be trained with state of the art reinforcement learning network training algorithms and using fuzzy network growing and pruning rules that can be fine tuned to exploit problem specific information.

RLL0's performance in the developed virtual animal ecosystem environment showed that it could learn a working agent policy to simulate fundamental animal behaviour, performing far better than the reference network in one of the tested settings. Using RLL0 in these environments also allowed for visualising the formed concept nodes and learnt agent policy, providing a form of explainable agent behaviour.

Based on these points the report concludes that the RLL0 dynamic network is a suitable agent estimation function for animal simulations for a compact, fast learning and well performing neural network with an explainable behaviour.

# Bibliography


[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 1, 2016, ISSN: 1476-4687. DOI: 10.1038/nature16961. [Online]. Available: https://doi.org/10.1038/nature16961.

[2] S. E. Fahlman and C. Lebiere, "Advances in neural information processing systems 2", in, D. S. Touretzky, Ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. The Cascade-correlation Learning Architecture, pp. 524–532, ISBN: 1-55860-100-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.107380.

[3] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks", *CoRR*, vol. abs/1606.04671, 2016. arXiv: 1606.04671. [Online]. Available: http://arxiv.org/abs/1606.04671.

[4] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks", *CoRR*, vol. abs/1612.00796, 2016. arXiv: 1612.00796. [Online]. Available: http://arxiv.org/abs/1612.00796.

[5] C. Strannegård, H. Carlström, N. Engsner, F. Mäkeläinen, F. S. Seholm, and M. H. Chehreghani, "Lifelong learning starting from zero", in *International Conference on Artificial General Intelligence*, Springer, 2019, pp. 188–197.

[6] H. Carlström and F. S. Seholm, "Supervised learning with dynamic network architectures", https://hdl.handle.net/20.500.12380/300060, Master's thesis, Chalmers university of technology, Gothenburg, Jul. 2019.

[7] W. E. Frankenhuis, K. Panchanathan, and A. G. Barto, "Enriching behavioral ecology with reinforcement learning methods", *Behavioural Processes*, vol. 161, pp. 94 –100, 2019, Behavioral Evolution, ISSN: 0376-6357. DOI: https://doi.org/10.1016/j.beproc.2018.01.008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0376635717303637.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning", *CoRR*, vol. abs/1312.5602, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning", *CoRR*, vol. abs/1509.06461, 2015. arXiv: 1509.06461. [Online]. Available: http://arxiv.org/abs/1509.06461.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[12] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: 1412.6980 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.05952.

[14] E. Wiewiora, "Reward shaping", in *Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 863–865, ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_731. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_731.

[15] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, M. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin, "Dynet: The dynamic neural network toolkit", *ArXiv*, vol. abs/1701.03980, 2017.

[16] P. Eriksson and L. W. Gotby, "Dynamic network architectures for deep q-learning: Modelling neurogenesis in artificial intelligence", https://hdl.handle.net/20.500.12380/300056, Master's thesis, Chalmers university of technology, Gothenburg, Jul. 2019.

[17] J. J. Buckley and Y. Hayashi, "Fuzzy neural networks: A survey", *Fuzzy Sets and Systems*, vol. 66, no. 1, pp. 1 –13, 1994, ISSN: 0165-0114. DOI: https://doi.org/10.1016/0165-0114(94)90297-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0165011494902976.

[18] S. Chen, C. F. Cowan, and P. M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks", *IEEE Transactions on neural networks*, vol. 2, no. 2, pp. 302–309, 1991.

[19] H. Han and J. Qiao, "A self-organizing fuzzy neural network based on a growing-and-pruning algorithm", *IEEE Transactions on Fuzzy Systems*, vol. 18, no. 6, pp. 1129–1143, 2010.

[20] P. P. K. Chan, X. Wu, W. W. Y. Ng, and D. S. Yeung, "Sensitivity based growing and pruning method for rbf network in online learning environments", in *2011 International Conference on Machine Learning and Cybernetics*, vol. 3, 2011, pp. 1107–1112.

[21] S. Y. Leow, K. S. Yap, H. J. Yap, and S. Y. Wong, "A performance evaluation of pruning effects on hybrid neural network", in *Journal of Fundamental and Applied Sciences*, vol. 9, 2017.

[22] H.-G. Han, Q. li Chen, and J.-F. Qiao, "An efficient self-organizing rbf neural network for water quality prediction", *Neural Networks*, vol. 24, no. 7, pp. 717 –725, 2011, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2011.04.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608011001390.

[23] S. Fan, H. Tian, and C. Sengul, "Self-optimization of coverage and capacity based on a fuzzy neural network with cooperative reinforcement learning", *EURASIP Journal on Wireless Communications and Networking*, vol. 2014, no. 1, p. 57, Apr. 12, 2014, ISSN: 1687-1499. DOI: 10.1186/1687-1499-2014-57. [Online]. Available: https://doi.org/10.1186/1687-1499-2014-57.

[24] A. Holzinger, C. Biemann, C. S. Pattichis, and D. B. Kell, "What do we need to build explainable AI systems for the medical domain?", *CoRR*, vol. abs/1712.09923, 2017. arXiv: 1712.09923. [Online]. Available: http://arxiv.org/abs/1712.09923.

[25] W. Samek, T. Wiegand, and K. Müller, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models", *CoRR*, vol. abs/1708.08296, 2017. arXiv: 1708.08296. [Online]. Available: http://arxiv.org/abs/1708.08296.

[26] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym", *CoRR*, vol. abs/1606.01540, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540.

[27] C. Strannegård, W. Xu, N. Engsner, and J. A. Endler, "Combining evolution and learning in computational ecosystems", *Journal of Artificial General Intelligence*, vol. 11, no. 1, pp. 1 –37, 2020. [Online]. Available: https://content.sciendo.com/view/journals/jagi/11/1/article-p1.xml.

[28] Z. Li, F. Zhou, F. Chen, and H. Li, "Meta-SGD: Learning to Learn Quickly for Few-Shot Learning", *arXiv e-prints*, arXiv:1707.09835, arXiv:1707.09835, Jul. 2017. arXiv: 1707.09835 [cs.LG].

[29] J. Snell, K. Swersky, and R. S. Zemel, "Prototypical networks for few-shot learning", *CoRR*, vol. abs/1703.05175, 2017. arXiv: 1703.05175. [Online]. Available: http://arxiv.org/abs/1703.05175.