Universiteit Utrecht

Master Thesis Proposal

Towards Self-Evolving Reinforcement Learning Architectures

Ruben Franquinet

Master Computing Science & Master Artificial Intelligence

Student	Ruben Franquinet, 1614959, r.franquinet@students.uu.nl
Academic Supervisor	dr. Roxana Radulescu, r.t.radulescu@uu.nl
Second Academic Supervisor	dr. Ioanna Lykourentzou, i.lykourentzou@uu.nl
Daily Supervisor	Dimitri Hooftman, Dimitri.Hooftman@infosupport.com
Host	Info Support B.V., Business Unit Mobility & Public (BUMP) at Kruisboog 42, 3905 TG Veenendaal
Date	2025-11-17

Abstract

Machine learning agents are becoming more and more advanced, achieving higher levels of performance across a wide range of tasks. However, despite this progress, there still is a wide variety of limitations. One of those limitations is the ability to adapt to changing environments. In the real world, rules and patterns often change over time, requiring continuous adaptation. Humans naturally cope with such changes, but most machine learning agents struggle to do so. They rely on fixed architectures and predefined hyperparameters, which are determined by the engineer at design time. This means that adapting to changes in their environment requires manual intervention by the engineer. To address this limitation, this thesis proposes a self-improvement loop that enables agents to monitor their performance autonomously and, if necessary, modify their architecture. The goal is to create reinforcement learning agents that can not only learn from experience but also evolve their internal structure to remain effective in dynamic and unpredictable environments. The system will be evaluated in an evolving environment where new observations and actions are introduced over time. A meta-controller determines when adaptation is needed and selects architectural modifications such as adding layers or neurons. The main idea behind this is that enabling online architectural adaptation for RL agents will allow the agents to recover more quickly from performance degradation and remain effective in non-stationary environments. By enabling agents to evolve their internal structure while learning, this research aims to take a step toward more robust, adaptable, and self-improving reinforcement learning systems.

Contents

Introduction

Reinforcement learning (RL) (Sutton et al., 1998) has recently emerged as one of the most promising and active research areas within machine learning (François-Lavet et al., 2018). RL is currently the preferred paradigm for sequential decision-making problems, where an agent must learn a control policy that selects actions over time to maximise long-term reward. This makes it widely used for complex control tasks such as game playing and robotics. Some of the most well-known examples in games are AlphaGo (Silver et al., 2016) and AlphaZero (Silver et al., 2017) while Sony's Gran Turismo agent (Fuchs et al., 2021) achieved racing performance comparable to world-class human drivers. It differs from supervised methods in a way that RL agents learn by interacting with their environment and receiving rewards based on those actions rather than relying on static labelled datasets. This makes them highly suited for continuous improvement, a property that has led many researchers to view RL as a potential step towards Artificial General Intelligence (AGI) (LeCun, 2022; Silver et al., 2021). By learning directly from experiences, RL agents can adapt their behaviour to the normal dynamics of an environment. However, what happens when the structure of the environment changes, new features appear, old features disappear and the agent's observation space evolves? Standard RL agents are unable to cope with these changes. They can update their policy parameters through backpropagation, allowing them to refine how they represent the world within a fixed architecture. However, they cannot autonomously modify their internal structure. As a result, performance degrades once the environment changes too much from the model's original assumptions. This gap motivates the main research question of this research:

How can reinforcement learning agents improve themselves online by dynamically reconfiguring their neural architecture in changing environments?

This will require agents that can not only adapt their policy and weights, but also be able to reconsider how they represent and process the world. This research explores whether RL agents can autonomously reconfigure their own neural architectures in an online environment in response to performance degradation or environmental shifts. We propose a self-improvement loop that detects when the model is not performing well, explores potential architectural changes, and evaluates them. The ultimate goal is to build agents that not only learn from what they know, but also from what they did not know before. To answer this research question, we first need to answer a few sub-questions.

How can a reinforcement learning agent determine when its current architecture becomes insufficient for the task or environment?

The first step for RL agents to adapt to a new environment is to detect when their current architecture or perceptual representation is no longer sufficient in that environment. We have to detect signals that indicate this insufficiency. For example, performance drops or a rise in prediction errors. The key challenge is to distinguish between normal fluctuations in performance and actual structural mismatches between the model and its environment. The latter indicates that the existing architecture can no longer capture the necessary features or relationships and that architectural adaptation may be required.

How can a reinforcement learning agent integrate new environmental features and actions that were not previously represented in its observation space?

Once the agent has identified that its current architecture is no longer sufficient, it may also need to introduce new features and actions that were previously not present in its observation or action space. But how can the agent introduce new features that were previously unobserved? In this study, the size of the input space is fixed, meaning that all possible input channels are

predefined, but some may remain inactive until new features become relevant. Secondly, we will manually introduce new actions to the agent when they become available. In this study, the agent's responsibility is to learn to interpret and utilise these newly activated inputs and outputs effectively, not to discover their existence.

Which strategies can be used to perform architectural modifications during training?
To adapt the architecture of the RL agent, we need to define which structural actions the agent or its meta-controller can perform. These actions determine the space of possible architectural modifications and therefore influence the agent's ability to adapt. The choice of the modification strategy depends on the nature of the deficiency detected in the previous step. For example, adding new layers or neurons might be necessary when the agent requires increased representational capacity to model the newly introduced features, while pruning redundant neurons or layers can help prevent overfitting.

How can the agent dynamically update its architecture without losing previously acquired knowledge?
Once the agent has determined that its current architecture is insufficient to model the environment effectively, the next step is to apply one of the architectural actions identified in the previous sub-question. However, sudden architectural changes can disrupt previously learned representations and degrade performance due to catastrophic interference, where new learning overwrites existing knowledge. Therefore, new actions must build upon existing knowledge rather than erasing it. Several strategies can support this process, such as function-preserving transformations, transfer learning techniques, and continual learning methods. These methods help to ensure that architectural changes lead to learning rather than forgetting.

Background

Neural Network

An artificial neural network (ANN) is a computational model that calculates an output based on its input. The concept of artificial neurons was first introduced by McCulloch & Pitts (1943), who proposed a simplified mathematical representation of biological neurons, which are the core of the human brain. From now on, when referring to a neural network (NN), we will be talking about the artificial neural network, not the biological one.

Neurons and Network Structure

The basic building block of a NN is a neuron. The concept of an artificial neuron is inspired by biological neurons in the human brain, which receive electrical signals from other neurons and transmit an output signal. Similarly, an artificial neuron does compute an output based on weighted input signals. It takes one or multiple inputs, multiplies them by a value called the weight of the input, sums all the weighted inputs together, adds a bias term to the weighted sum, and passes the result through an activation function as can also be seen in Figure 1. This leads to the following formula:

$$y = \varphi\left(\left(\sum_{i=1}^{n} w_i x_i\right) + b\right)$$

Where

- x is the input vector
- φ is the activation function
- w_i is the weight for the ith input
- x_i is the ith input
- b is the bias term
- n is the number of input values

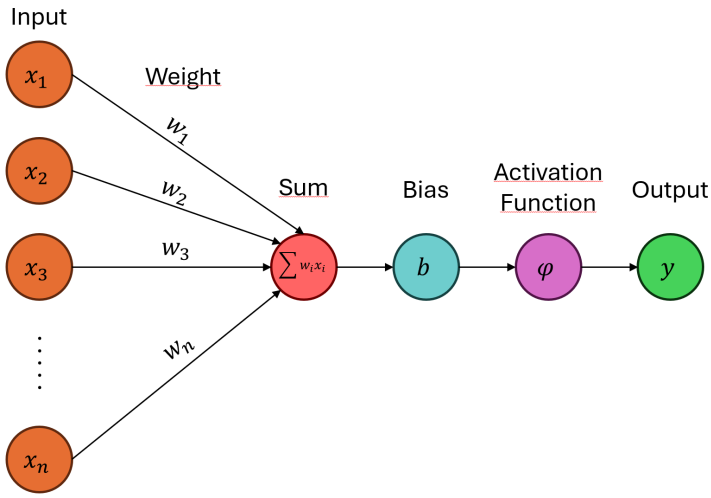

Figure 1: An artificial neuron

A single neuron can represent only simple, mostly linear relationships between input and output. By combining many neurons across multiple layers, a NN can approximate arbitrarily

complex nonlinear functions. This is important because it allows NNs to learn complex patterns. Moreover, there are infinitely many possible patterns that could occur in the data. This is why we combine the neurons into a NN. A NN consists of three types of layers, and each layer can have one or more neurons as can be seen in Figure 2. The input layer receives all the features. Each feature is a single value. The input layer passes all the values to the next layer without changing them. Then we have the hidden layer, which consists of neurons like we have described before. And finally, we have the output layer. This layer takes the output from the final hidden layer and computes an output value. This layer can also have multiple outputs, depending on the number of output values required. There is always only one input and one output layer, but there can be multiple hidden layers. In a fully connected network, each neuron receives input from all neurons in the previous layer.

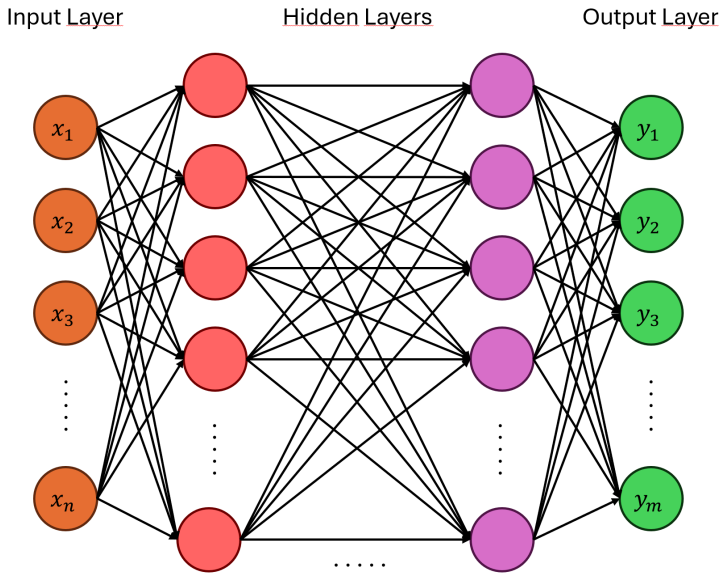

Figure 2: An artificial neural network

Activation Function

If we use no activation function (or an identity activation function), the neuron's output becomes a purely linear combination of its inputs. We would have the following formula:

$$y = \left(\sum_{i=1}^{n} w_i x_i \right) + b$$

The problem with this approach is that the output remains linear, as stacking multiple linear layers without a nonlinear activation function would still result in a single linear transformation. This means that we can only learn linear patterns in data. However, since there are many cases where this is not the case, we must introduce a mechanism to account for nonlinearity. Luckily, we have something for this, activation functions. An activation function applies a mathematical transformation to the neuron's input, introducing nonlinearity into the network. Since different datasets contain different patterns, sometimes even a single dataset contains multiple patterns, there are several activation functions. Each activation function has different characteristics that affect how a network learns. Popular ones are Sigmoid, ReLu or Softmax. Every neuron in a hidden layer has an activation function.

Backpropagation

To enable a NN to learn from data, its weights and biases must be adjusted so that the network's output becomes closer to the desired result. The method to do this adjustment is called backpropagation Rumelhart et al. (1986). It is the key algorithm that enables NNs to learn complex patterns in data efficiently. Backpropagation uses gradient descent, an optimisation technique that updates the parameters in the direction of the negative gradient of the loss function. The goal is to minimise the loss function and thus get the best set of parameters for the data. It does this with the following update rule:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

Where

- η is the learning rate
- L is the total loss
- $\frac{\partial L}{\partial w}$ is the gradient of the loss with respect to the weight

In a NN, the output of each neuron depends on the outputs of neurons in all preceding layers. To efficiently compute how each weight influences the overall loss, backpropagation uses the chain rule. This rule allows the gradient to be expressed as a product of derivatives across layers. With this rule, we can calculate the derivative of the loss with respect to one weight w_i as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Where

- $z = \sum_i w_i \cdot x_i + b$
- $y = h(z)$ is its output

During the training phase, this method is applied recursively, starting from the output layer and working backwards to the input layer. Each neuron contributes its local gradient, and the overall error flows backwards through the network to update all parameters. Backpropagation provides an efficient way to compute gradients in NNs with many layers, making it possible to train deep neural architectures.

Loss Function and Optimisation

A loss function quantifies how well the network performs. It measures the difference between the predicted output and the desired output. As we saw before, the gradients of the loss function are used to update the model's parameters during training. Different problems require different loss functions. The most common loss function for regression problems is the Mean Squared Error (MSE), which computes the average of the squared difference between the actual and predicted values of N samples:

$$L = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Where

- N is the number of samples
- y_i is the true value for sample i
- $\hat{y}$ is the predicted output for sample i

Once the loss is computed, optimisation algorithms such as Gradient Descent or Adam use the gradients of this value to update the weights and biases of the NN, moving them in the direction that reduces the loss. Choosing an appropriate loss function and optimiser is crucial for the training and final performance of the model. In supervised learning, the loss directly measures prediction accuracy against labelled data. However, in reinforcement learning, which we will discuss in the next section, the loss is often derived from reward-based objectives that indicate how well the agent's actions performed. Despite this difference, both rely on the same optimisation principles described here.

Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on learning through interaction with an environment rather than from labelled data or hidden structures in data. The foundation of RL was formalised by Sutton et al. (1998), who described it as the process of learning what to do in order to maximise a numerical reward.

Fundamental Concepts

While supervised learning optimises predictions using labelled data, and unsupervised learning seeks to discover patterns in unlabelled data, RL is driven by experience. An RL agent interacts with an environment by taking actions, observing the resulting state, and receiving a reward that indicates how good that action was. Over time, the agent learns a policy that maximises its expected return. A policy is a mapping from states to actions.

Unlike other learning methods, RL does not predict classes or values. Instead, it estimates the expected future return associated with each possible action and state. To do this effectively, the agent must balance exploration, where it tries new actions to gather more information about the environment, and exploitation, where it chooses actions that it currently believes to yield the highest reward. This trade-off between exploration and exploitation is central to the field of RL. We prefer to always get the best reward, but we also want to know if other actions are better than we think. The mathematical framework used to formalise this decision-making process is known as a Markov Decision Process (MDP) (Puterman, 2014), which models the environment in terms of states, actions, transition probabilities and rewards. The interaction between the agent and environment in reinforcement learning is illustrated in Figure 3

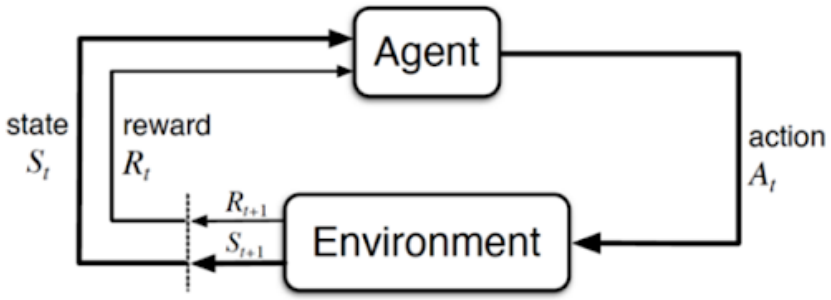

Figure 3: RL interaction loop between agent and environment (Sutton et al., 1998).

Learning from Rewards

The RL agent improves by updating the estimates of how good actions are based on the rewards it receives. This can be done in three main ways: value-based and policy-based and actor-critic methods. These three categories reflect different approaches to estimating or optimising the expected return and form the foundation of modern RL algorithms.

Value-based methods often rely on Temporal Difference (TD) learning (Sutton, 1988), which estimates a value function that predicts the expected return for each state-action pair. The update rule is based on the difference between successive predictions:

$$\delta_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Where
- δ_t is the temporal difference error
- r_t is the reward at time t
- γ is the discount factor
- $Q(s_{t+1}, a_{t+1})$ is the estimated action-value for the next state–action pair
- $Q(s_t, a_t)$ is the estimated action-value for the current state–action pair

This error reflects how much better or worse the actual return was compared to the expected return. It is used to update the value estimates.

Policy-based methods rely on policy gradient techniques (Sutton et al., 1999). Instead of learning which actions are good indirectly through value estimation, these methods adjust the parameters of a policy network to maximise the expected reward. The objective function is:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R]$$

Where
- $J(\theta)$ is the expected performance of the policy
- θ are the parameters of the policy
- R is the cumulative discounted reward
- $\mathbb{E}$ is the expectation under the policy
- π_θ is the policy with the parameters θ

In this method, the gradient with respect to θ indicates the direction in which the policy should change to improve performance. Policy gradient methods are particularly powerful when combined with deep NNs, leading to deep RL algorithms. These methods form the foundation for most modern RL systems and are closely related to the optimisation processes discussed in the next section.

The third category consists of actor–critic methods. These combine both value-based and policy-based learning. The actor is a policy network that selects actions, while the critic estimates the value function and evaluates how good those actions were. It uses the critic's feedback to adjust the actor. Actor–critic approaches form an important hybrid class that many modern RL algorithms build upon.

Deep Reinforcement learning

Deep Reinforcement Learning (DRL) combines the NN computational model with the decision-making framework of RL. Some of the most popular algorithms are Deep Q-Networks (DQN) (Mnih et al., 2015) and Proximal Policy Optimization (PPO) (Schulman et al., 2017). Traditional RL algorithms often struggle to handle environments with large or continuous state spaces (Lillicrap et al., 2015) since they rely on explicit tables or linear function approximations to represent value functions or policies. However, as we saw before, NNs are capable of approximating complex and nonlinear functions, making them ideal for representing these mappings. In DRL, a NN is used as a function approximator. As discussed in the previous section, there are three main categories of DRL methods. The simplest is value-based, here, the network estimates the value of each state-action pair. The second one is policy-based, here, the network

outputs a probability distribution over actions. Finally, we have the actor-critic methods, which combine two networks. One that estimates the policy (the actor) and one that estimates the value function (the critic).

During training, the agent collects experiences from the environment, which consist of states, actions, rewards, and next states. It uses this data to improve the NNs. The parameters of this NN are updated using the backpropagation algorithm. However, instead of minimising the loss based on ground truth labels, it minimises a loss derived from reward signals or temporal difference errors. This allows the network to learn behaviours that maximise long-term cumulative rewards rather than short-term accuracy.

Despite its success, DRL introduces several challenges. NNs can be unstable when trained with non-stationary data or delayed rewards, and are prone to issues such as catastrophic forgetting, where newly learned information overwrites earlier knowledge, and feedback instability, where errors in value estimates are recursively fed back into the learning loop. The optimisations are highly sensitive to architectural choices such as network depth and activation functions. These design decisions can have a significant impact on convergence speed and policy performance.

Understanding and improving the neural architecture used in DRL is therefore a critical research direction. The following section explores Neural Architecture Search, a technique that aims to automatically design network structures that lead to better performance and more stable learning when applied to DRL environments.

Neural Architecture Search

Neural Architecture Search (NAS) is a research field that focuses on automatically finding the optimal design for a NN Zoph & Le (2016). Traditionally, the structure of a NN, including the number of layers and activation functions, is determined manually by engineers through a process of experimentation and heuristic tuning. Since there are infinitely many possible network architectures, this manual process is extremely time-consuming and often leads to suboptimal designs. NAS aims to overcome these limitations by enabling algorithms that can automatically and efficiently discover network architectures that achieve high performance for a given task.

The main idea behind NAS is to treat the architecture design problem itself as an optimisation problem. In general, NAS consists of three key components: a search space, a search strategy, and a performance estimation strategy Elsken et al. (2019). The search space defines all possible architectures that the NAS algorithm will explore. A well-defined search space is important, as a too small search space might lead to a design that is not optimal, while a too large search space will be computationally heavy. The search strategy determines how new architectures are explored within the search space. The goal is to find the most optimal design without having to check every single one of them. To reduce computational cost during evaluation, many NAS approaches use techniques such as early stopping, proxy metrics, or weight sharing to estimate performance efficiently.

These components enable NAS to automatically discover architectures that outperform manually designed ones in both accuracy and efficiency. In the context of DRL, this approach is particularly valuable because the architecture of the NN has a strong influence on learning stability, convergence speed, and policy performance. Automatically finding architectures that perform well across tasks leads to more robust and generalisable DRL agents.

Meta Controller

A meta controller is a higher-level control mechanism that operates on top of another learning process (Dayan & Hinton, 1992). Instead of deciding which actions an agent should take in an environment, it decides when and how the learning process itself should adapt or update. In machine learning, such control mechanisms can be used to monitor, adapt, and optimise a learning system over time.

In RL, a meta-controller can be seen as an agent that controls other agents. It observes their performance and decides when to adjust the model. The goal of such a controller is to improve the learning efficiency or stability of the underlying agents. This creates a hierarchical setup where the meta-controller learns from the behaviour of the agent it manages, making adjustments based on observed performance trends. Unlike backpropagation, which only updates the internal parameters of a fixed model, a meta-controller can modify higher-level aspects of the learning system itself. For example, it may adjust the agent's network architecture, modify how inputs are processed, or even influence the agent's available actions.

Continual Learning

Continual learning refers to a model's ability to adapt to new data over time without forgetting previously acquired knowledge (Parisi et al., 2019). Unlike traditional training setups, where a model is trained once on a fixed dataset, continual learning focuses on adapting to new information as it becomes available. This makes it particularly important for agents that operate in dynamic or non-stationary environments, such as those with changing reward functions or evolving state spaces.

The main challenge in continual learning is catastrophic forgetting, as first identified by McCloskey & Cohen (1989), where a NN learns new information and updates its parameters using backpropagation, potentially overwriting previously acquired knowledge. As a result, the model may perform well on the newly learned data but forget how to handle tasks it learned earlier. To prevent this, different strategies are known. A common approach is to preserve important parameters using regularisation techniques such as Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017), or to replay past experiences so that older knowledge remains known (Shin et al., 2017). Continual learning is especially relevant in RL because of the environments that can change over time. An agent must recognise these changes, keep using what it has learned, and at the same time adapt to new conditions.

Related Work

The idea of creating DRL agents that improve over time is a new but active field of research. Therefore, recent research should be taken into consideration. A literature review on RL-based NAS was presented by Jaâfra et al. (2019). It provides an overview of RL-driven NAS, highlighting the use of controller networks to guide the architecture search. It showed that most NAS methods operate offline or in batch mode, rather than online during interaction with the environment. Another approach to offline improvement of the neural architecture is discussed by Zhang et al. (2021), who introduce Adaptive Scalable Neural Architecture Search (AS-NAS), which utilises a reinforcement evolutionary algorithm to adapt neural architectures efficiently. However, both these researches focus on adaptation outside of the agent's interaction loop, while the proposed research focuses on online adaptation.

In contrast to these offline approaches, Zhang et al. (2022) addresses continual adaptation in non-stationary environments using an adaptive layer on top of standard RL training. This can be seen as a meta-controller that determines when and how to update the model. However, Zhang et al. (2022) focuses on adjusting the policy representation and contextual encoding rather than the network architecture itself.

A survey on dynamic NN adaptation is presented by Han et al. (2021). This survey reviews a wide range of methods and highlights the breadth of dynamic adaptation strategies. The most interesting approaches here are those that utilise RL for architectural decisions, aligning with the meta-controller approach. However, this survey mainly focuses on supervised learning networks, where adaptation occurs at inference time or on a per-sample basis rather than continuously during training.

Franke & others (2019) explores to combine neural architecture evolution with DRL for continuous tasks. In this work, evolutionary algorithms are used to modify the NN's architecture based on the agent's performance. It makes adaptations such as the number of layers or neurons. This approach demonstrates that structural adaptation can improve policy learning and overall performance for RL agents. However, this research updates the network offline and between training episodes, meaning that the agent cannot adapt its structure while interacting with the environment. In contrast, the meta-controller proposed in this research aims to enable such adaptations online.

Xu et al. (2021) and Rosenfeld & Tsotsos (2017) both address the challenge of maintaining good performance when new tasks are added in continual learning. Rosenfeld & Tsotsos (2017) introduces Deep Adaptation Modules (DAM) for incremental learning. Where previously learned knowledge is preserved by reusing and freezing parts of the existing network when adding new layers for new tasks. Xu et al. (2021) build further on this idea and proposed an adaptive progressive continual learning framework that dynamically expands network capacity when new tasks arrive. They introduced two implementations, Regularised Continual Learning (RCL) and Bayesian Online Continual Learning (BOCL). Both are created to determine when to add new layers or neurons to mitigate catastrophic forgetting and capacity saturation. However, both researches rely on task-based learning and predefined expansion rules. At the same time, the meta-controller proposed in this research aims to perform continuous and autonomous architectural adaptation during online learning, without the need for explicit task introduction.

Gao et al. (2020) introduces Continual Learning with Efficient Architecture Search (CLEAS), which uses NAS for continual learning to determine efficient network structures for each new task. This method also uses an RL-based controller to select architectural changes while reusing

previously learned parameters. CLEAS demonstrates that architecture search controlled by a controller can effectively support continual learning, which aligns closely with the proposed research. However, its adaptation process is still task-based and occurs between tasks rather than continuously. The meta-controller in this proposed research aims to perform architectural adaptations in an online environment while being guided by performance feedback rather than task introductions.

In a recent study Dohare et al. (2024) show that standard deep learning methods lose plasticity in continual learning settings. They demonstrate that maintaining diversity in the network is crucial for preserving adaptability. They propose several mechanisms to address this problem, such as random reinitialisation of parts of the network or structural changes that introduce new neurons or connections. The latter one is particularly relevant since the proposed research focuses on a meta-controller that could structurally change the network. This study, therefore, highlights the importance of ensuring that such a controller remains aware of potential plasticity loss and can act to prevent it.

Methodology

This thesis follows an experimental approach to investigate how RL agents can automatically adapt their neural architecture when performance degrades. The goal is to create a higher-level meta-controller that monitors the agent's performance and, when this drops to a certain level, adapts the architecture by executing the appropriate action. With this, there will be an agent that can continuously improve itself online without retraining or manual intervention.

This requires an experimental setup consisting of two components: the RL agent and the meta-controller. The RL agent is responsible for interacting with a dynamic environment and learning the policy that maps states to the right actions within that environment. This RL agent continues to learn using standard and widely adopted algorithms such as Deep Q-Networks (DQN) (Mnih et al., 2015) for value-based learning and REINFORCE (Sutton et al., 1999) or Proximal Policy Optimisation (PPO) (Schulman et al., 2017) for policy-based learning. However, once its performance declines beyond a predefined threshold, the meta-controller becomes active. The meta-controller operates as a higher-level decision-maker that observes the RL agent's learning progress. It receives input signals such as cumulative reward, gradient variance, or other metrics that reflect the agent's stability and performance. Based on these inputs, the meta-controller determines whether the RL agent's architecture should be updated or not. Moreover, if so, the meta-controller also decides which exact actions should be performed to update the architecture. Each action is executed on the RL agent's network. To make sure the agent can always efficiently respond we will duplicate the agent and update only the duplicate. After updating the duplicate of the agent we will swap the two agents. After every adaptation, the RL agent continues its learning process, allowing the meta-controller to evaluate if the change has led to improved performance or not. This forms the self-improvement loop as shown in Figure 4 in which the meta-controller continuously monitors, adapts, and validates the architecture.

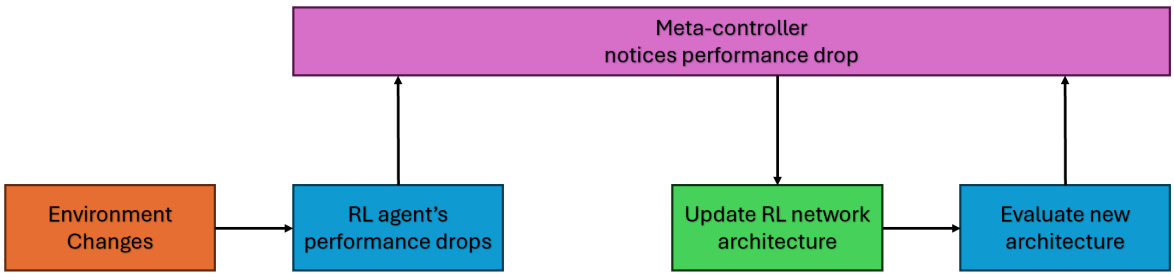

Figure 4: The proposed self improvement loop

To avoid the challenge of catastrophic forgetting, architectural changes are implemented using techniques such as function-preserving initialisation or partial weight transfer. This makes sure that existing knowledge is retained while new knowledge is introduced. This process results in a system that not only learns new behaviours through reinforcement but also evolves structurally to remain effective in non-stationary environments.

Environments

An important part of doing research in the field of RL is selecting the environment that is suited for the problem being studied. Since this research focuses on continuously improving RL agents in evolving environments, it requires an environment that can be extended with

new features and actions. This study will therefore use modular and extensible grid-based environments such as those provided by MiniGrid (Chevalier-Boisvert et al., 2023). MiniGrid environments are lightweight, customisable and designed to be easily extended with additional objects, colours, behaviours or action types. This makes them well-suited for simulating dynamic or non-stationary environments in which new features or interactions appear over time. To best simulate environmental changes, certain inputs associated with specific output actions are introduced over time after the initial training phase. To scope this research, we will first focus on the addition of new input patterns that lead to one of the existing outputs and only after this is correctly integrated we will introduce new output actions. Once the agent is deployed, these previously unseen actions and related input patterns are gradually introduced. This will cause a temporary drop in performance, alerting the meta-controller that adaptation of the model is needed.

Evaluation

To evaluate the effectiveness of the proposed study, the experiments will measure and compare the adaptive RL agent against a non-adaptive RL agent with a fixed network architecture. Both agents are trained in identical dynamic environments where the observation and action spaces evolve over time. For the performance comparison, this study will compare the cumulative reward, recovery speed after environmental shifts and the relative performance drop during changes in the environment. Besides pure performance, we should also consider architectural efficiency, as simply adding more neurons and layers may not be the best solution. Since increasing the size of the model both in width and depth means that the model will become computationally heavier. This will result in a model that requires more time and has higher costs. To ensure this, we will compare the network structure of the adaptive RL agent against RL agents that are not dynamically changed, but contain all the environmental features thus far in their initial training phase. A third and final comparison that should be done against a baseline RL agent is the one stated by Dohare et al. (2024) on plasticity. To compare the plasticity of the agents, we will use metrics such as catastrophic forgetting rate and gradient variance. Together, these three metrics provide a clear view of how well the meta-controller improves adaptability, manages architectural complexity and maintains stability in non-stationary RL environments.

Timeline

This thesis will follow an iterative process of development, experimentation, evaluation and documentation. The process is divided into five phases, each building upon the previous one. This step-by-step approach makes sure that each component of the self-improvement loop is validated before moving on to the next phase.

Phase 1: Environment and baseline setup (November - December)
In the first phase, the environment will be set up with a dynamic extension to introduce new actions and patterns over time. For this phase, a basic RL agent will be created to determine if the assumption is correct that the agent's performance will decline once new patterns or actions appear. This phase establishes the baseline for future experiments.

Phase 2: Meta-controller integration (Januari)
For the second phase, we will extend this basic RL agent by connecting the meta-controller that monitors the agent's performance and determines when to adapt the neural network architecture. Since in this phase the actual adaptation mechanisms are not yet implemented , we will evaluate the meta-controller by verifying that its adaptation triggers correspond to genuine performance degradation rather than random fluctuations.

Phase 3: Neural network adaptation (Februari - March)
The third phase will be the most exciting and challenging one. In this phase, we will add the actual neural network adaptations. To evaluate this, we will examine whether performance will start to recover after a decline. And equally important, whether the agent avoids catastrophic forgetting of previously learned behaviours.

Phase 4: Self-improvement loop evaluation (April)
In the fourth phase, the whole self-improvement loop will be evaluated by comparing it against other static RL agents that do not have this self-improvement loop. Performance will be measured in terms of cumulative reward, recovery speed, and stability across environmental changes.

Phase 5: Writing the Thesis (May - June)
In the final phase, the focus shifts from experimentation to documenting the research. This includes writing the full thesis, integrating the sections developed during earlier phases, and providing a critical reflection on the results and limitations. Additionally, implementation developed in this project will be prepared as an open-source repository with clear documentation to ensure reproducibility of the experiments. This phase ensures that the entire research process is transparent, well-structured, and accessible for future work.

Throughout all phases, progress and experimental results will be documented in the final thesis as each component is developed and tested. This ensures that both the implementation and findings are clearly documented per phase.

Conclusion

This study will focus on extending RL agents to adapt their neural architectures in response to changing environments autonomously. The ultimate goal is to design a self-improvement loop that, using a meta-controller, enables online self-improvement of the RL agent by monitoring its performance and making architectural adjustments as needed.

While existing work has already demonstrated that NAS and continual learning through architectural adaptation can improve learning efficiency. These studies are primarily performed in an offline or task-based setting. This study addresses that limitation by executing it in an online setting, where architectural adaptations occur continuously, thereby addressing a key limitation in current research. This will make RL agents more robust to real-world environmental changes, which are often dynamic and unpredictable.

The biggest challenge lies in preserving existing knowledge during architectural updates. Even a slight change in the architecture could disrupt the learned representations and lead to catastrophic forgetting. To prevent this and make sure that adaptations enhance rather than erase research, this research will evaluate methods such as function-preserving transformations, transfer learning techniques, and continual learning methods. By creating RL agents that can not only learn from experience, but also evolve their own structure, this research takes a step toward creating adaptive and self-improving intelligent systems.

Bibliography

Chevalier-Boisvert, M., Dai, B., Towers, M., Lazcano, R. de, Willems, L., Lahlou, S., Pal, S., Castro, P. S., & Terry, J. (2023). Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. *Corr*.

Dayan, P., & Hinton, G. E. (1992). Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, *5*.

Dohare, S., Hernandez-Garcia, J. F., Lan, Q., Rahman, P., Mahmood, A., & Sutton, R. (2024). Loss of Plasticity in Deep Continual Learning. *Nature*, *632*, 768–774. https://doi.org/10.1038/s41586-024-07711-7

Elsken, T., Metzen, J. H., & Hutter, F. (2019). Neural architecture search: A survey. *Journal of Machine Learning Research*, *20*(55), 1–21.

Franke, J. K., & others. (2019). Neural Architecture Evolution in Deep Reinforcement Learning for Continuous Control. *Arxiv Preprint Arxiv:1910.12824*.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., & others. (2018). An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, *11*(3–4), 219–354.

Fuchs, F., Song, Y., Kaufmann, E., Scaramuzza, D., & Dürr, P. (2021). Super-human performance in gran turismo sport using deep reinforcement learning. *IEEE Robotics and Automation Letters*, *6*(3), 4257–4264.

Gao, Q., Luo, Z., & Klabjan, D. (2020). Efficient Architecture Search for Continual Learning. *IEEE Transactions on Neural Networks and Learning Systems*, *34*, 8555–8565. https://doi.org/10.1109/tnnls.2022.3151511

Han, Y., Huang, G., Song, S., Yang, L., Wang, H., & Wang, Y. (2021). Dynamic Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *44*, 7436–7456. https://doi.org/10.1109/tpami.2021.3117837

Jaâfra, Y., Laurent, J., Deruyver, A., & Naceur, M. (2019). Reinforcement Learning for Neural Architecture Search: A Review. *Image Vis. Comput.*, *89*, 57–66. https://doi.org/10.1016/j.imavis.2019.06.005

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., & others. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, *114*(13), 3521–3526.

LeCun, Y. (2022). A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, *62*(1), 1–62.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *Arxiv Preprint Arxiv:1509.02971*.

McCloskey, M., & Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation: Vol. 24. Psychology of learning and motivation* (pp. 109–165). Elsevier.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., & others. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., & Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, *113*, 54–71.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Rosenfeld, A., & Tsotsos, J. K. (2017). Incremental Learning Through Deep Adaptation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*, 651–663. https://doi.org/10.1109/tpami.2018.2884462

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *Arxiv Preprint Arxiv:1707.06347*.

Shin, H., Lee, J. K., Kim, J., & Kim, J. (2017). Continual learning with deep generative replay. *Advances in Neural Information Processing Systems*, *30*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., & others. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., & others. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359.

Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, *299*, 103535.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44.

Sutton, R. S., Barto, A. G., & others. (1998). *Reinforcement learning: An introduction* (Vol. 1, Issue 1). MIT press Cambridge.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, *12*.

Xu, J., Ma, J., Gao, X., & Zhu, Z. (2021). Adaptive Progressive Continual Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *44*, 6715–6728. https://doi.org/10.1109/tpami.2021.3095064

Zhang, T., Lei, C., Zhang, Z., Meng, X.-B., & Chen, C. L. P. (2021). AS-NAS: Adaptive Scalable Neural Architecture Search With Reinforced Evolutionary Algorithm for Deep Learning. *IEEE Transactions on Evolutionary Computation*, *25*, 830–841. https://doi.org/10.1109/tevc.2021.3061466

Zhang, T., Lin, Z., Wang, Y., Ye, D., Fu, Q., Yang, W., Wang, X., Liang, B., Yuan, B., & Li, X. (2022). Dynamics-Adaptive Continual Reinforcement Learning via Progressive Contextualization. *IEEE Transactions on Neural Networks and Learning Systems*, *35*, 14588–14602. https://doi.org/10.1109/tnnls.2023.3280085

Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *Arxiv Preprint Arxiv:1611.01578*.