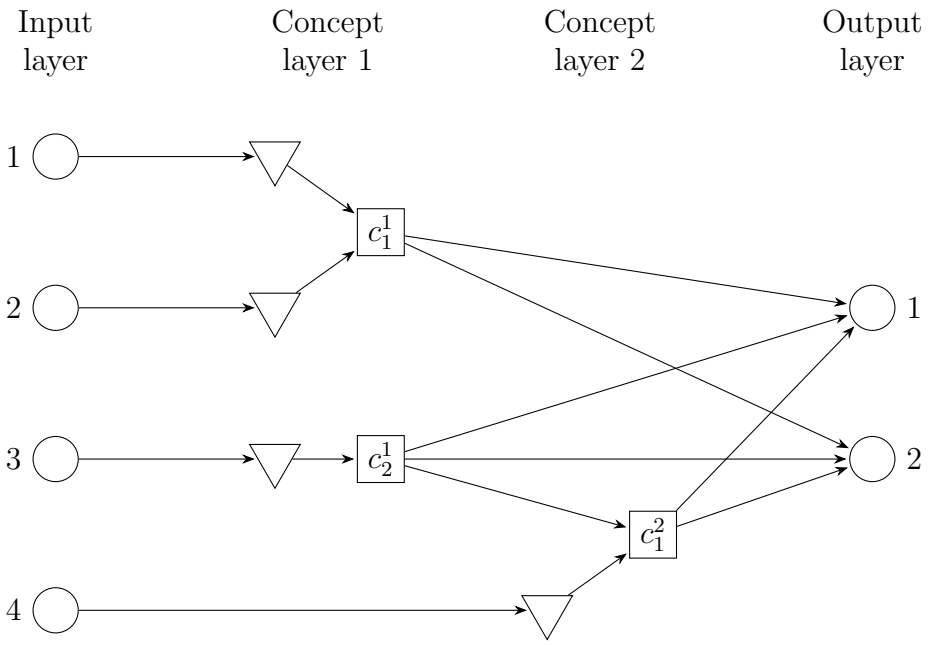
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Dynamic Network Architectures for Deep Q-Learning

Modelling Neurogenesis in Artificial Intelligence

Master's thesis in Computer science and engineering

Pontus Eriksson
Love Westlund Gotby

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Dynamic Network Architectures for Deep Q-Learning

Modelling Neurogenesis in Artificial Intelligence

Pontus Eriksson
Love Westlund Gotby

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Dynamic Network Architectures for Deep Q-Learning
Modelling Neurogenesis in Artificial Intelligence
Pontus Eriksson
Love Westlund Gotby

Supervisor: Claes Strannegård, Department of Computer Science and Engineering
Examiner: Morteza Haghir Chehreghani, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of a dynamic network architecture

Typeset in LaTeX
Gothenburg, Sweden 2019

Dynamic Network Architectures for Deep Q-Learning
Modelling Neurogenesis in Artificial Intelligence
Pontus Eriksson
Love Westlund Gotby
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Artificial neural networks have become popular within a range of machine learning fields for their ability to solve complex problems, with one of the uses as function approximators in Q-learning. These networks generally have static architectures, which is a problem in the regard of artificial *general* intelligence, since no single specific architecture is optimal for all problems. In this thesis, we implement and evaluate a proof of concept for a novel approach of a dynamic network architecture, resulting in a model that can be seen as a combination of compressed classical table-based Q-learning and artificial neural networks. The model presented performs true *tabula rasa* deep Q-learning, starting with an empty network that is gradually extended with nodes when experiencing "surprising" events, and is capable of generalization by abstracting important features from noisy input. Finally, we show that the model can learn from delayed rewards in simple environments and compare it with the well-established DQN algorithm.

Keywords: dynamic neural network, artificial neural network, ANN, Q-learning, DQN, deep learning, machine learning, reinforcement learning.

Acknowledgements

We would like to express our thanks to our supervisor Claes Strannegård who has provided the idea for the dynamic model, as well as support and interesting discussions during the project.

We would also like to thank Patrick Andersson and Anton Strandman for input regarding the report, and Herman Carlström and Filip Slottner Seholm for discussions regarding the implementation.

Finally, we would like to thank our opponent Florian Stellbrink for good question and valuable feedback.

Pontus Eriksson, Love Westlund Gotby, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1

Introduction

The field of artificial intelligence (AI) has become part of both science and everyday life, as its applications range from producing images of black holes [1] to finding media content similar to what you like. While successful, the algorithms are often tailored to a specific problem and used for that purpose only. The subbranch artificial *general* intelligence (AGI) aims at creating more generalized models that can understand and solve multiple problems without manual reconfiguration from an engineer.

Artificial neural networks (ANNs), often used for reinforcement learning, generally have static architectures decided by an engineer, while the weights are updated according to experiences. This static design of the networks is a problem in the regard of AGI since no single specific architecture is optimal for all problems, exposing a need for a dynamic design instead. Similarly, biological brains are continuously rewired for adaptation [2, 3], further motivating a dynamic architecture for ANNs.

In this thesis we implement and evaluate a proof of concept model from a novel approach for a dynamic network architecture in the form of a combination of compressed classical table-based Q-learning and artificial neural networks, based on the work of Strannegård et al. [4]. The model presented performs true *tabula rasa* deep Q-learning, starting with an empty network that is gradually extended with nodes when experiencing "surprising" events, and is capable of generalization by abstracting important features from noisy input. Finally, we show that the model can learn from delayed rewards in simple environments.

Although theorized, the dynamic network model by Strannegård et al. has not previously been implemented and compared to other related techniques. Therefore, the purpose of this thesis is to implement the dynamic model combined with Q-learning and compare it experimentally to a regular deep Q-learning network that is the standard reinforcement algorithm used today, and also to compare it on an idea-level to other methods for dynamic networks.

The structure of the thesis is as follows. Chapter 2 will present a background to reinforcement learning, the limitations of regular neural networks, and also describe the idea that the thesis builds upon, the novel dynamic model. Related work in the field of dynamic neural networks will also be given at the end of this chapter.

Chapter 3 will go through the implementation details and evaluation methods used. In Chapter 4 the evaluation results of the dynamic model are presented. The results, advantages and disadvantages of the dynamic model, similarities with previous dynamic architecture models, and future work are discussed in Chapter 5, and final conclusions about the model as a whole are presented in Chapter 6.

2

Background

This chapter aims at providing the reader with the necessary background to the subject, introducing the fundamental concepts of reinforcement learning, Q-learning, and limitations of traditional neural networks. The novel model for dynamic architectures is then presented, followed by some changes made to the baseline DQN networks that have been shown to improve its performance [5, 6, 7, 8] and that were used in experiments, and finally related work to dynamic models.

2.1 Reinforcement Learning

One field of machine learning called *reinforcement learning* (RL) deals with problems where a software agent, i.e. a decision-making program, is to maximize an arbitrary reward over time by choosing actions at discrete timesteps in an observable *environment*. The general idea is to use some policy for choosing actions, evaluate the chosen actions in some environmental state by interpreting a change in the environment as a positive or negative reward, and feeding the new state into the agent for the next decision, illustrated in Figure 2.1. By using a combination of the chosen action, which state the action was taken in, and the reward it led to, the agent can learn from the experience by updating its policy used for taking actions in that state accordingly. This is repeated until reaching a *terminal state*, e.g. game over or a win-condition. All states from the start up until the terminal state together form what is called an episode.

2.1.1 Markov Decision Process

Markov decision processes (MPDs) can be used to model discrete systems wherein each state s, a number of actions can be taken, and each action a has a stochastic outcome in the form of a new state s' and a corresponding real-valued reward $r = R(s, a, s')$. Although the state transitions are stochastic, the probability for a transition into state s' depends only on the current state s and the taken action a, satisfying the *Markov property*. This implies that the current state is the only information needed to make a well-informed decision. The typical problem formu-

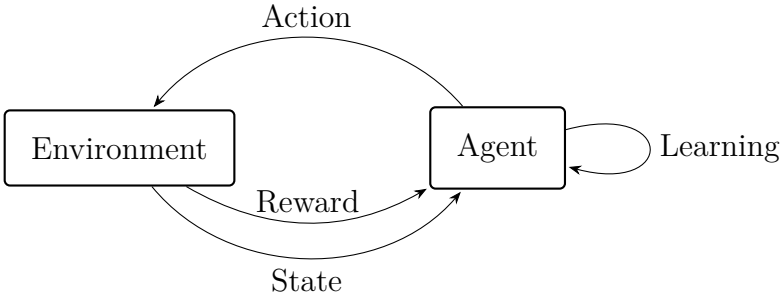


Figure 2.1: Illustration of a reinforcement learning model. The agent takes an action based on the current state, leading to a change in the environment. From this a reward is extracted and is together with the new state fed into the agent, which updates its policy and takes a new action given the new state.

lation when using MDPs is to find a policy that maximizes either the *accumulated*, *average*, or *discounted* reward over a long time, where a policy can be described as a set of rules to follow for which action to take given any state. For this reason, RL problems are commonly formulated as MDPs.

2.2 Q-Learning

Neural networks within AI have proven to be successful for deciding which action to take given the current state of a system. For most real-life situations such as mimicking an animal or playing games, however, the optimal action for a single timestep might not be the optimal one in the long run. A simplified slightly related example is the Stanford marshmallow experiment, where a subject can either choose to eat a treat straight away or wait for a short period of time in order to get two treats instead [9].

To solve this type of decision by finding optimal discounted reward-maximizing policies in Markov decision processes, Watkins introduced a new form of reinforcement learning which he termed *Q-learning* [10]. The basic idea is to estimate the *quality* Q of a legal action a in state s:

$$Q(s, a) = r + \max_{a'} Q(s', a'),$$

i.e. the *predicted* sum of the highest Q-value from the actions available in the next state s', plus the reward r of a in s.

$$a_{opt} = \arg\max_{a} Q(s, a) \tag{2.1}$$

Following the policy in Equation (2.1), therefore, maximizes the *total* discounted reward over all succeeding steps from the current one.

The Q-value estimations are stored in a table and updated iteratively according to

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{predicted Q-value}} + \eta \cdot \Big(\underbrace{r_t + \gamma \cdot \max_a Q(s_{t+1}, a)}_{\text{target Q-value}} - \underbrace{Q(s_t, a_t)}_{\text{predicted Q-value}} \Big) \tag{2.2}$$

where s_t and a_t is the observed state and action taken at time t, respectively, η is the learning rate, r_t is the reward received from taking action a_t in state s_t, and $\gamma \in (0, 1)$ is the discount factor that controls how future rewards are valued in comparison to instantaneous rewards. A value of $\gamma = 0$ lets the policy ignore future rewards and only choose actions based on immediate rewards, and $\gamma = 1$ values immediate rewards equally much to future rewards. As the Q-values represent the *predicted* discounted return for taking an action and then following an optimal policy, the *target* Q-value of the prediction is the sum of the reward of the action plus the discounted maximum Q-value of the next state. The update rule in Equation (2.2) adds a fraction of the difference between the prediction and the target, shifting the approximation towards the target Q-value.

2.2.1 Deep Q-learning

While Q-learning is built on solid principles, for large state-spaces it is often unfeasible to store a table of Q-values for each state. Based on the assumption that the same action in similar states will yield similar rewards and similar following states, a function approximator for the Q-values can be used instead.

Neural networks have proven successful for function approximation, but can run into issues with convergence [11]; the networks can go into a tail-chasing behavior, causing their weights to diverge which in turn leads to devastating results.

With a couple of improvements targeting the problem of convergence, a combination of deep neural networks and Q-learning, called deep Q-networks (DQNs), has been successfully used as Q-value estimators to let software AI play Atari arcade games [5].

2.3 Limitations of regular neural networks

In the context of AGI, there are some issues with regular neural networks, three of which will be motivated here as desired properties for more general intelligence: lifelong learning, transfer learning, and generality.

While there are existing methods targeted at improving neural networks using different approaches to dynamic architectures, e.g. Progressive Neural Networks [12] and the Cascade-Correlation Learning Architecture [13], they are not meant to be a complete solution towards a general model, but rather solutions specifically for transfer learning and generality in the sense of being able to solve multiple problems, respectively.

2.3.1 Lifelong learning

As the objective of AGI is a model that repeatedly can figure out and solve new and previously unseen problems, learning cannot be bound by a limited time frame, but must rather happen continuously during the entire life of the agent.

Catastrophic interference is the undesired inclination of neural networks to shift their focus towards whatever they are learning at the moment, causing them to quickly forget previously learned information if not experiencing an even distribution of the data. While information that has not been used for a longer period of time could arguably be forgotten, it is not desirable to forget for example how to do your job as a chef just because you took a weekend course in art.

2.3.2 Transfer learning

The ability to use information from past experiences to understand newly encountered problems is one of the reasons a single human being can excel at many different things. This ability equips us with a basic toolbox in new situations so that even if we have not yet mastered the new problem, we have a better chance at tackling it right off the bat than if we were to try any action at random. As the goal of AGI is for one and the same model to be able to handle in principle any problem without having to start training from scratch, transfer learning is essential.

2.3.3 Generality

The success of ANNs for various AI tasks has surpassed many expectations and continue to fuel the hype as further achievements are reached. But the majority of these networks are designed specifically for the problem at hand, and not for general problem-solving. Bigger networks can theoretically learn more complex behaviors, but due to the fully connected style often used this also increases the computational effort during training and the risk of overfitting [14]. For further advancements in the sense of *truly* general AI, the same model needs to be able to learn to solve a number of different problems with a training cost that is linear in the number of problems.

2.4 The dynamic network architecture

The new dynamic model consists of a few notions: concept nodes, a focus set, learning by adding new nodes, learning through backpropagation, and generalization [4]. The following sections will go through this list and describe each notion in detail,

and then explain how they are combined into a complete machine learning model as a dynamic network.

The following notation is required for this section on the dynamic model:

- The input to a network will be denoted by the vector $\mathbf{x}$.
- Concept nodes will be denoted c_j^i where i and j specifies the layer i and its index j in the layer.
- A subscripted vector $\mathbf{u}_k$ indicates component k of $\mathbf{u}$.
- The depth of a network is equal to the number of layers with concept nodes, and the width of a layer is the number of concept nodes in that layer. The notion of depth and width is further explained with the help of Figure 2.2.

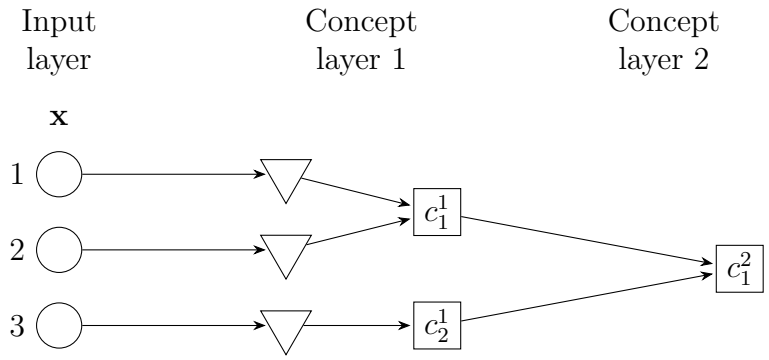


Figure 2.2: This network has a depth of two as it has two layers of concept nodes. The width of layer 1 is two as there are two concept nodes in that layer, and the width of layer 2 is one as it has only one concept node. A deeper network means that concept nodes are used as inputs to other concept nodes, and deeper layers therefore capture more detailed concepts. Concept node c_1^2 builds on both c_1^1 and c_2^1 which indicates that in this case it is relevant to know not only when c_1^1 and c_2^1 are active independently, but also when they are active simultaneously. A wider layer only adds further concepts at the same level of detail.

2.4.1 Concept nodes

Concept nodes are the equivalent of a neuron in regular neural networks. The similarities and differences between the two are:

1. Unlike a neuron in a regular neural network that is connected only to the previous layer of neurons, a concept node can be connected to any external input and any other concept node in previous layers. It cannot, though, be connected to an external input via more than one direct or indirect connection, i.e. each concept node may depend on one specific value, and one specific value only, of each external input. To clarify, a concept node cannot depend on a

single external input to be for example both 0.3 and 0.7, as this could never lead to a full activation of the concept node.

2. A concept node remembers specific values v_j for each of its inputs that it can compare with external inputs x_j in order to compute how well it recognizes a complete state $\mathbf{x}$. This is done via a function $h(x_j; v_j)$ with image $[0, 1]$, and all of this is illustrated with an example network in Figure 2.3. Outputs 0 and 1 are equal to no recognition and full recognition respectively. For inputs other than concept nodes no saved value is required since the activation of a concept node always is in the range $[0, 1]$, as explained below in Item 3. The input to a concept node is then a vector with a component $h\left(x_j; v_j\right)$ for each of its external inputs j, and a component with the activation of each input that is a concept node in a previous layer.

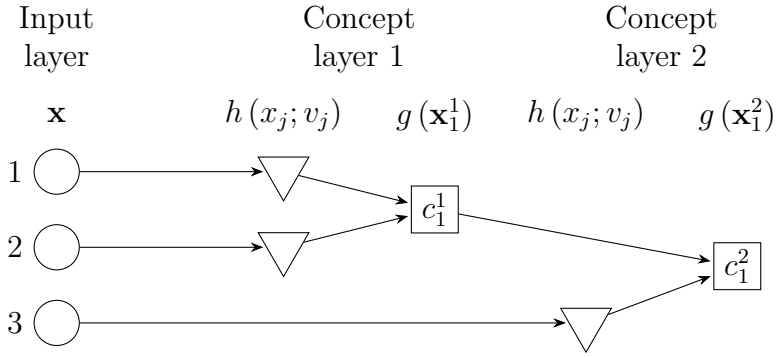


Figure 2.3: An illustration of how the inputs to the two concept nodes c_1^1 and c_1^2 are computed. The triangles are nodes that computes the similarity of an input to a concept node from the function $h(x_j; v_j)$ given an input value x_j and the remembered value v_j of input j. The input to c_1^1 is the vector $\mathbf{x}_1^1 = [h(x_1; v_1), h(x_2; v_2)]^\top$ given v_1 and v_2 remembered by c_1^1, and the input to c_1^2 is $\mathbf{x}_1^2 = [g(\mathbf{x}_1^1), h(x_3; v_3)]^\top$ where $g(\mathbf{x}_1^1)$ is the activation of c_1^1 and given v_3 remembered by c_1^2.

3. Similarly to a neuron, a concept node c_j^i has an activation a_j^i given a network input $\mathbf{x}$. This activation is computed from an activation function $g(\mathbf{x}_j^i)$, where $\mathbf{x}_j^i$ is the input to concept node c_j^i. Each concept node also has a corresponding weight vector $\mathbf{w}_j^i$ with trainable weights for each of its inputs. Initially, all weights of a concept node are set to 1. See Figure 2.4 for an example network with further explanation.

4. Unlike regular neurons, all concept nodes are connected directly to the nodes in the output layer of the network in addition to any concept nodes in deeper layers. This means that a concept node c_j^i has connections to all output nodes representing the different actions in the environment. Every concept node has a vector of weights for these connections, similar to the weights on its inputs, but these weights represent the concept nodes Q-value estimations of the corresponding actions. This vector is denoted as $\mathbf{Q}_j^i$, and the predicted Q-value of action k by concept node c_j^i is then $\left(\mathbf{Q}_j^i\right)_k$.

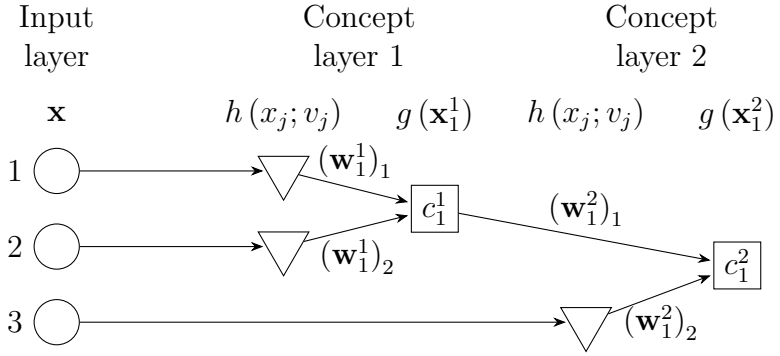


Figure 2.4: The activation of the concept node c_1^1 is computed from the activation function $g(\mathbf{x}_1^1)$ and similarly for c_1^2 the activation is $g(\mathbf{x}_1^2)$. Regarding the inputs to concept nodes, see Figure 2.3. Each concept node has a weight vector $\mathbf{w}_j^i$ with components $\left(\mathbf{w}_j^i\right)_k$ corresponding to input components $\left(\mathbf{x}_j^i\right)_k$ of an input vector $\mathbf{x}_j^i$.

The total predicted Q-value Q_k of action k is computed with a function $f(k; F)$ where F is the *focus set* introduced in the next section. Figure 2.5 illustrates this with an example network.

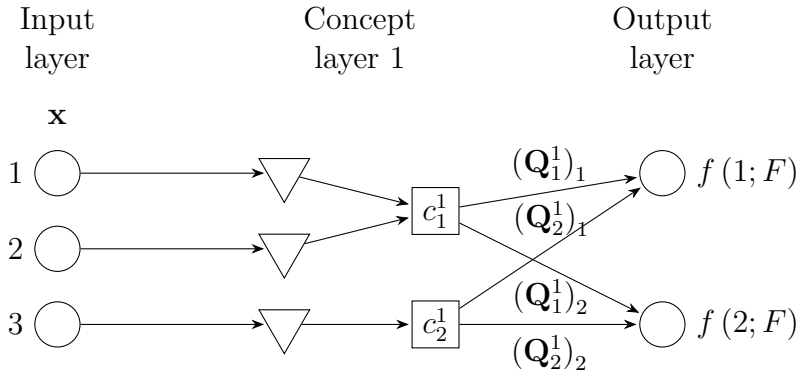


Figure 2.5: Predicted Q-values for every output by each concept node. A concept node c_j^i has a corresponding vector $\mathbf{Q}_j^i$ of predicted Q-values $\left(\mathbf{Q}_j^i\right)_k$ for each action k. The predicted Q-values by the network are computed from a function $f(k; F)$ where F is the *focus set* given a state $\mathbf{x}$.

2.4.2 Focus set

When computing the output from a state $\mathbf{x}$ in a given network, the individual activation of each node is computed layer by layer in the network. Then, for all concept nodes, a concept node c is selected to be part of a focus set $F(\mathbf{x})$ of maximum size $M \in \mathbb{N}$ if:

1. c has an activation A at least ϕ,

2. there are no concept nodes in deeper layers that depend on c that has an activation of at least ϕ, and

3. c is among the M nodes with the highest *importance* I that also satisfy Item 1 and 2,

where $\phi \in (0, 1)$ is a real-valued *activation threshold* parameter and the importance I_j^i of a concept node c_j^i is defined as

$$I_j^i = A_j^i \cdot \max_k \left| \left(\mathbf{Q}_j^i \right)_k \right|.$$

Importance is used because a concept node that does not have the highest activation, but still higher than ϕ, which predicts a Q-value with large magnitude for some action should have the possibility to take precedence over other concept nodes with higher activation but with Q-values of smaller magnitude; a concept node with a larger magnitude Q-value may indicate that its concepts might be of greater value in the given state.

The purpose of the focus set is to select one or more concept nodes that are used to compute the final output of the network, that is the Q-values of all possible actions. More specifically, the Q-value of action k is computed from a function $f(k; F)$ that depends on the focus set F.

If the focus set $F(\mathbf{x}) = \emptyset$ for some input $\mathbf{x}$, i.e. no concept node recognized its concept, a random action will be performed in the current state and the model will check whether it should add a new node for the state or not.

2.4.3 Adding new nodes

The absolute difference between target Q-value and the predicted Q-value of an action in a given state is called prediction error. An error that is larger than a set *surprise threshold* is interpreted as an unknown experience, i.e. the model is surprised that the predicted Q-value is so far off from the target Q-value, and that the current configuration is not yet equipped to handle the given state properly. To accommodate this, a new node that recognizes this and similar states, and associates it with the remembered action and corresponding Q-value, is added to the network at a depth directly after the deepest concept input. This gives the model a variant of one-shot learning, meaning that the error from a single experience sparks an attempt to learn to avoid that error the next time a similar state is seen. The reason for this not being true one-shot learning is because there is no guarantee that the target Q-value is the true Q-value for the given state and action, it is only a guess.

2.4.4 Reinforcement learning through backpropagation

In the opposite case of a large error and adding new nodes, when the prediction error is smaller than the set threshold the current state is treated as a variation of the state that the network remembers through its concept nodes. Learning is in this case done by regular reinforcement learning using backpropagation. In contrast to regular neural networks, where all nodes are trained by default, only the concept nodes that are chosen to be in the focus set and their ancestors, i.e. all nodes that the nodes in the focus set build from, are updated according to their partial derivative on the target-prediction error. In short, only the parts of the network that was allowed to affect the final Q-values are trained with backpropagation.

2.4.5 Generalization of concepts

The model should be able to handle noisy environments, meaning that if two distinct concept nodes actually represent the same specific situation, but with varying noise from the environment, the model should be able to recognize this and create a new and generalized concept node in place of the other two concept nodes. This can proposedly be done in two different ways:

1. The model should realize when two concept nodes have similar Q-value vectors and have an intersection of inputs that is not empty, they might actually represent the same concept but with a different noise. The model should then create a *generalized* concept node with the inputs of the new concept node as the intersection of the inputs to the old concept nodes, and using the average of the relevant saved values of external inputs, input weight vector components, and Q-value vectors.

2. The second idea for generalization is called *abstraction* and suggests that if an input to a concept is environmental noise, the model should eventually realize this and prune this input, leaving an abstraction of the previously too specific concept which was fitted to input noise. This can happen in two cases. First, a concept node's memory of an input may not be specific enough, meaning that a memory should be of some specific input value rather than all possible values. Second, one component of the weight vector of a concept node gets under a parameter threshold, meaning the activation of the input does not have a large enough impact on the activation of the concept node. After an input has been pruned, a merging step can be applied where the newly pruned concept node is compared to other concept nodes with the intention of merging the two, as the removal of an input could make the resulting node too similar to another one.

2.4.6 A complete model

Using the notions from Section 2.4, a machine learning model is constructed. This model is introduced as the Dynamic agent, cf. a Deep Q-Learning Network agent (DQN agent). To summarize, the network in a Dynamic agent is built similarly to the network of a DQN agent: there are concept nodes in different layers, computing the output consists of a layer-by-layer feed-forward pass and choosing a focus set to compute predicted Q-values for all actions, and fine-tuning the concept nodes is done by backpropagation. However, the things setting the architectures of the dynamic and the DQN agent aside from each other is what is done dynamically by the Dynamic agent:

1. Concept nodes are added to the layer only as the need arises by a large prediction error on some state.

2. During feed-forward, a layer is not dependent only on the immediately previous layer, as in most artificial neural networks, but may depend on any previous layer in the network.

3. The output, meaning the prediction of how good an action in a given state is, may depend only on a part of the network. It follows from this that only that part of the network is trained when fine-tuning parameters with backpropagation.

2.5 Reinforcement learning improvements

Additional relevant techniques that have been proven to improve reinforcement learning, deep learning, and machine learning in general, are presented in this section.

Due to the success of Mnih V. et al. [5] and that the ideas presented there became the de facto standard for deep Q-learning, the methods used there, namely the preprocessing, training details, evaluation procedure, and *experience replay*, were in this project considered part of traditional DQNs.

2.5.1 Experience replay

Some situations in e.g. games will produce a sequence of very similar states, actions, and rewards. In these cases, training can bias the model towards focusing on that specific situation, even though it is not more important than any other. To combat this, the DeepMind project used a technique called experience replay, inspired by the biological process with the same name [5]. The idea is to save *transitions* used for Q-learning, that is (state, action, next state, reward, terminal state)-tuples, in a data

set, usually of some fixed size. New transitions are added to the data set as they are received and old transitions are removed on a first-in-first-out basis when the array has been filled. Transitions are then uniformly sampled from this data set instead of always training on the current state of the environment. When successively sampling transitions in this manner they are unlikely to be closely connected to the current state, and also temporally to each other, resulting in more stable training. Another benefit is that it is possible to sample a transition from very long ago, counteracting the issue of catastrophic interference as old transitions can be used for training.

This technique can be further expanded by weighting the probability of replaying experiences based on how important that specific experience is for further learning. This improvement is called *prioritized experience replay* and has been shown to yield a more efficient learning [6].

The priority p of a transition is computed from the error $|Y_t - Y_p|$ between the *target* and *predicted* Q-values for a given state as

$$p = (\text{error} + \epsilon)^\alpha$$

with an α such that $0 \leq \alpha \leq 1$ and some small $\epsilon > 0$. The addition of ϵ takes care of the case when the error is zero and the transition therefore never sampled. How much the magnitude of the error should affect the probability of sampling a transition is governed by the parameter α. Setting $\alpha = 0$ means that every probability is equal, making it equivalent to the uniform case described above, while $\alpha = 1$ means that the memories are sampled with priority equal to their error.

2.5.2 Double DQN

Textbook case Q-learning tends to overestimate Q-values as the maximum Q-value from the next state is used to evaluate the chosen action. In order to manage this, a modification was introduced by [7] called *Double Q-learning* which involves using two Q-value estimators, choosing an action from the two estimators, and finally updating one of them from random choice by having them evaluate each other's actions.

As the same parameters that are updated from learning are also used for computing the Q-values during training, causing us to move closer to the target while also moving the target, resulting in runaway values. To combat this, a modification to the standard DQN algorithm was presented by Mnih et al. [5]. The resulting modified DQN algorithm had a target network with parameters θ^- and an online network with parameters θ. The target network is used to compute the target Q-values during training, and the online network chooses actions and is trained using the target Q-values. Thanks to a separate network being used to compute target values, the training of the online network becomes more stable. The parameters from the online network are periodically copied to the target network every τ timesteps.

The target Q-values for the modified DQN at timestep t is then

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \theta_t^-\right).$$

A further improvement called *Double DQN* was introduced by van Hasselt et al. [8], which also uses two networks similar to the DQN used in [5], but with a modification to how the target is calculated so that the action is chosen by the online network, but its value is evaluated by the target network. The target Q-values can the be written as

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \arg\max_a Q\left(S_{t+1}, a; \theta_t\right); \theta_t^-\right).$$

This helps to reduce the Q-value overestimation issue with the original Q-learning algorithm, leading to a more stable training with faster convergence, and has been shown to give better results [8].

2.5.3 Exploration

At the start of learning with no a priori experience, it is useful to first explore the environment, both to probe the state space and to try different actions in every state, with the purpose of avoiding quickly getting stuck in local optima.

A widely used strategy for exploration, also used in this project, is the annealed ϵ-greedy strategy, which at every step explores the environment with probability ϵ, called the exploration rate, by choosing a random action, and exploits learned knowledge by choosing an action according to its policy with a probability $1 - \epsilon$. The exploration rate is set to linearly decay from a value ϵ close to or equal to 1, down to a small value $\epsilon \geq 0$. This is to explore the environment early in order to learn which actions lead to higher rewards in which states. When ϵ is small, most chosen actions will be based on what is known of the environment, but it is still important to keep ϵ non-zero as otherwise, it is possible to get stuck in local optima.

2.6 Related work

Machine learning is not a new field, and related works to this project are found mainly in the two preceding areas of reinforcement learning and growing neural networks. Two of the related works of interest are introduced below.

2.6.1 Progressive Neural Networks

To improve on the learning of *DeepMind*, a technique called *Progressive Neural Networks* [12] lets the network use information already learned from previously encountered similar games and situations. This is done by letting the nodes from

previously trained networks supply information to the nodes in the new network for the newly encountered specific game, enabling transfer learning.

2.6.2 Cascade-Correlation

In *The Cascade-Correlation Learning Architecture* [13] Fahlmanand and Lebiere introduce a new model for learning using growing deep neural networks. In a simplified explanation, the *input* weights of a number of *candidate* neurons fully connected to all previous neurons are trained separated from the active network, with the goal of maximizing the sum of the magnitude of the correlation between a candidate neuron's output and the residual output error of every output node of the network. When convergence is achieved, the candidate with the highest correlation is chosen as the winner and is added to a new layer at the deepest level of the network with its input weights frozen. Training resumes once again but now with the goal of finding an *output* weight resulting in the smallest error. In this way, a classification network can learn both in a smaller number of iterations and by using less computational power than regular training with backpropagation in a fixed architecture, while not compromising on the performance of the network.

3

Method

This chapter presents the implementation and evaluation methods used. First, the implemented agents are introduced, followed by some specifics about the dynamic architecture in connection to concept nodes and generalization. Finally, the environments in which experiments where done, and the hyperparameters used for evaluation in each environment, are given.

3.1 Agents

Two agents were implemented for comparisons. First, the textbook model and today's standard within reinforcement learning: a DQN agent that uses a deep Q-learning network for decision-making. The DQN uses a static architecture network with a fixed number of internal parameters. Second, the dynamic architecture network based on the novel idea by Strannegård et al. [4], hereafter called the Dynamic agent. The Dynamic agent has a fixed number of inputs and outputs but is otherwise not constrained in the number of internal parameters in the same way the DQN is.

Double DQN and prioritized experience replay were implemented for both agents in an attempt to increase performance and make a fair comparison. In case these improvements were not beneficial for the Dynamic agent, as it is rather different from regular DQNs, evaluations for comparisons were done both with and without these improvements.

Both agents were also trained in batch mode, meaning that at each learning iteration, instead of training using only one experience, a mini batch of experiences and an average error of these was used to update the network parameters. An initial learning delay equal to the batch size was also used, as the agents otherwise overfitted towards the early experiences. As with the rest of the hyperparameters, the batch size was changed to best fit each agent for each environment when evaluating.

3.2 Concept nodes

A concept node requires a function h to compute the similarity between an external input and a remembered state, and a function g to compute the activation of the node. The functions chosen for h was the Gaussian function

$$h(x) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

for some external input x, a remembered value μ, and a σ specifying how deviations from the remembered value affect the activation of the concept node. When adding a new node that has some external inputs, a μ equal to the value of each external input was remembered by the new concept node. The same initial σ was used for all external inputs.

The activation function g used, given a weight vector $\mathbf{w}_j^i$ and input $\mathbf{x}_j^i$ to the concept node c_j^i, was

$$g\left(\mathbf{x}_j^i\right) = \frac{{\mathbf{w}_j^i}^\top \mathbf{x}_j^i - b}{\sum_k \left(\mathbf{w}_j^i\right)_k}$$

where b is a trainable bias initially set to zero.

To compute the estimated Q-value of action k, i.e. $f\left(k; F\right)$ where F is the focus set, the function f used was

$$f\left(k; F\right) = \frac{\sum_{c_j^i \in F} \left(\mathbf{Q}_j^i\right)_k \cdot A_j^i}{\sum_{c_j^i \in F} A_j^i}.$$

This function computes the sum of Q-values from all concept nodes in the focus set, weighted with the activation of the nodes.

When the network was surprised by a large prediction error, it checked whether a new concept node should be created and added to the network. When creating a new concept node, a parameter for restricting the maximum number of inputs allowed was used. A low value of the maximum number of inputs has the potential to force the network into building by depth instead of simply placing all of its nodes in a single layer. If there is no limitation on the number of inputs, there will be no new concept nodes in deeper layers until some concept nodes have had their inputs pruned, as concept nodes are not allowed to build on the same external input multiple times and new concept nodes always use as many inputs, first chosen from the focus set and then from external inputs, as possible.

Further, the creation of a new node was only allowed if no node that recognizes the current state was already present in the network. To check this, the proposed concept node was tested if it had the same inputs as any other node, and if it did, the

already remembered value μ_r of each external input was compared to the proposed value μ_n of the new concept node. If the absolute difference $\mid \mu_r - \mu_n \mid$ was smaller than a threshold μ_{th} for all remembered values, the current state was deemed to already be represented in the network and backpropagation was applied instead of adding a new node, even though the network was surprised by a large prediction error.

3.3 Generalization through abstraction

After backpropagation learning, each concept node that took part in taking the decision was subject for abstraction. If either an input weight was under a set weight threshold, or the σ in the Gaussian function in connection with an external input was larger than a set threshold, that input was considered to be unimportant compared to other inputs or resulting in a high similarity for a too wide range of values in order to be one specific concept, and would therefore be pruned.

If the input to be pruned was the only input to the concept node, the concept itself was considered unimportant and was removed.

If pruning an input left only a concept input left, the node was removed and replaced with an edge between the remaining input and the node's children, as a concept node needs either at least one state feature input or at least two concept inputs.

If and only if an input was removed from a node, the node was subject for generalization, i.e. merging. The reasoning behind this is that there should be no other way for two nodes to get similar enough, as nodes are created to be unique. When comparing a pruned node with other nodes to see if they are similar, the same comparison as described in Section 3.2 for creating new concept nodes was used.

3.4 Environments

Varying levels of environment difficulty were used for training and evaluating the model, allowing an easy start and incremental increase of the required complexity of the model.

3.4.1 Berry

The Berry environment is a simple classification problem. The state of the environment consists of three boolean values, called *sweet*, *sour*, and *bitter*. These inputs, or rather flavors, together form a berry. Six berries were created, each with a label corresponding to whether the berry should be eaten or not. If the action chosen by

an agent in a given state, i.e. a certain berry, is the correct one according to the berry's label, the agent will receive a reward of $+1$. If the action is not the correct one, the agent will instead receive punishment in the form of a negative reward of -1. The environment was made sure to contain berries that shared two out of three feature values but had different rewards in order for the agents to have to differentiate berries on a single feature. One episode consists of one berry because the berries are not correlated in any way. The maximum return of an episode is $+1$ as an episode is only a single berry.

3.4.2 Noisy Berry

To extend on the binary tastes of the Berry environment, *Noisy Berry* introduces four kinds of berries, each with a unique set of tastes but in 100 varieties. The variety of each taste is drawn from a Gaussian distribution with different expectation values in the range $[0, 1]$ but with the same standard deviation. The taste values are also clipped in the range $[0, 1]$. One of the four berries gets one of its tastes drawn completely random at uniform in $[0, 1]$. The rest of the task remains the same as in Section 3.4.1.

The Noisy Berry environment tests if the agent can handle input noise, i.e. recognize variations of the same state and not create a new node for each variation of the same concept, and also its ability to abstract the random taste from one of the berries.

3.4.3 Grid

A simple environment that is able to test the temporal learning capabilities of an agent was implemented, illustrated in Figure 3.1. An episode in this environment terminates when reaching the reward-tile or if failing to reach it in 100 steps. The rewards given are a small negative reward -0.1 when the agent walks into a wall, and when reaching the reward-tile it receives the reward

$$r = 10 - 0.1t$$

where t is the current timestep equal to the number of steps taken by the agent. This is to encourage the agent to not waste actions by walking into walls, and instead reach the reward in as few steps as possible each episode. The return from one episode is also equal to this reward, and the maximum return is 9.6.

Running the Dynamic agent in this environment gives the possibility to confirm that the temporal learning works as expected by inspecting the estimated return, i.e. the estimated Q-values, of all actions in a given state. The idea here is that the agent should learn that the Q-value of moving down from the green tile is equal to the reward received. Then, moving down from the red tile should result in that the agent comes into a state, the green tile, where it knows it can get a reward.

This leads to that the Q-value of moving down from the red tile should be the one time discounted Q-value of action down on the green tile. Last, the agent should learn that from the rightmost blue tile, moving right to the red tile should yield Q-value equal to the one time discounted reward from action down on the red tile, that is equal to the two times discounted reward from action down on the green tile. Furthermore, as the reward for reaching the reward-tile can be different from time to time, depending on the number of steps taken, an additional difficulty for estimating Q-values is introduced.

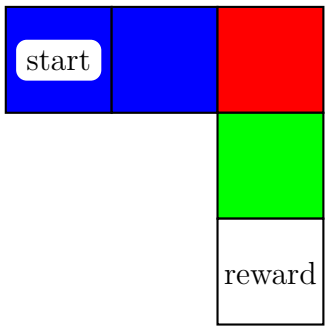


Figure 3.1: Grid environment. The agent starts in the leftmost blue tile and the goal is to walk to the reward-tile from only seeing the color of the tile that the agent is currently standing on. An action that would move the agent outside the grid results in that the agent is not moved but stays in the same tile.

3.4.4 Discrete Catch

This is an environment of size 6×6 in discrete steps where the agent is tasked with catching a ball that is falling straight downwards, illustrated in Figure 3.2. This tests the abilities of an agent in a finite state space of size $6 \cdot 6 \cdot 6 = 216$ and with three actions: move left, move right, and stand still. An episode consists of the ball starting at a random position along the ceiling and the paddle starting at a random position along the floor. The ball then falls towards the floor and the agent tries to position the paddle so that it will catch the ball. The input to an agent is the horizontal and vertical position of the ball and the horizontal position of the paddle as it is always positioned somewhere on the floor. There are sparse rewards of $+1$ when a ball is caught and zero otherwise. The maximum return of an episode is $+1$.

3.4.5 Continuous Catch

The Continuous Catch environment is considered a difficult problem but is similar to the discrete catch environment in that the agent should catch a ball falling from the ceiling before it hits the floor. The rewards are also the same: $+1$ if a ball is

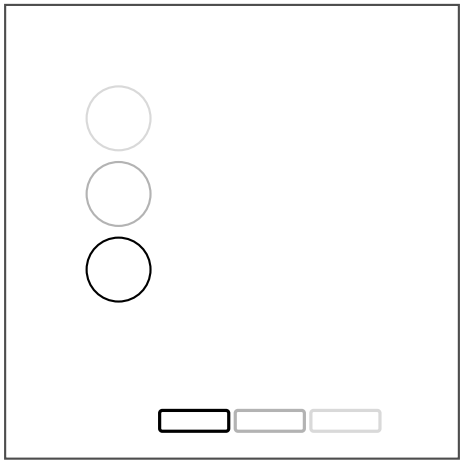


Figure 3.2: Discrete Catch environment. The agent's task is to move the paddle (rectangle) under the ball (circle) in order to catch it before it falls onto the floor. The agent and the ball starts at random positions along the floor and ceiling, and the agent is only allowed to move the paddle horizontally.

caught and zero otherwise, and the actions are also the same. The big difference from Discrete Catch, though, is that the ball falls in a random direction downwards within a set angle $\pm 30°$ from the vertical and with speed 1. An agent will now additionally receive the horizontal and vertical velocity of the ball as inputs. Also in contrast to Discrete Catch, the paddle always starts at the center of the floor and moves a distance of 0.5 per move action. If the ball happens to hit a wall, it bounces back in the reflected direction. The environment, along with the sideways movement of the ball, is illustrated in Figure 3.3. The environment has height 3, width 2, and the paddle has a width of 1. Initially, the size was similar to the one of Discrete Catch but had to be reduced for the Dynamic agent to have any chance of learning what do to.

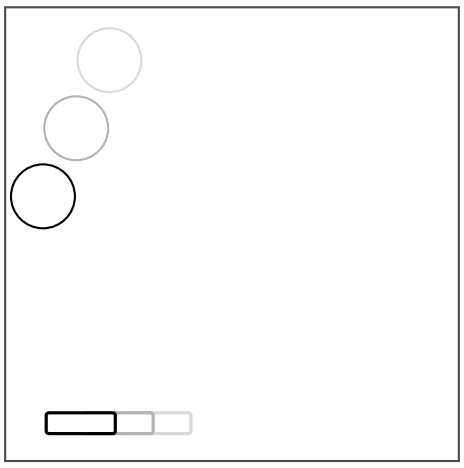


Figure 3.3: Continuous Catch environment. The agent is tasked with placing the paddle (rectangle) under the ball (circle), which has a horizontal as well as a vertical velocity, in order to catch it before it falls onto the floor.

3.4.6 CartPole balancing

Another difficult problem is the task of balancing an inverted pendulum attached to a cart which is allowed to move in a single dimension. This is the CartPole environment, illustrated in Figure 3.4, a classical problem from automatic control, implemented by OpenAI Gym [15]. The state consists of four real numbers describing the angle of the pole from the vertical, the angular velocity of the pole, the position of the cart, and the velocity of the cart. An episode starts at the equilibrium with a small random noise added to it. The performance in an episode is evaluated by the number of timesteps the cart is able to balance the pole before either the cart hits a wall or the pole falls too much to the side, terminating the episode. The maximum return of an episode is 200.

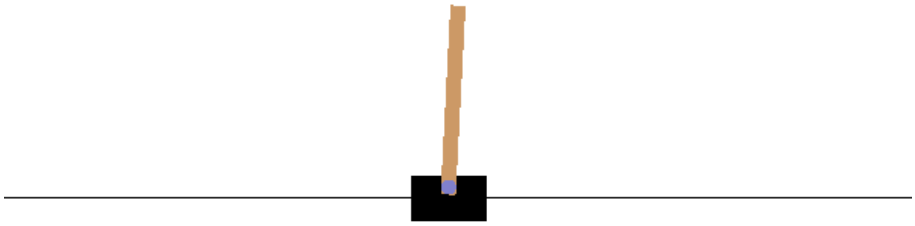


Figure 3.4: The CartPole environment. The task of the agent is to balance the pole upwards around a freely rotating joint, with a restriction to the magnitude of the cart acceleration while staying on the black line.

3.5 Evaluation hyperparameters

A limited grid search was conducted to find good hyperparameters for each agent and environment so that both agents performed as well as possible in each environment. This makes for a more fair comparison as it is not expected for the two agents to have the same optimal hyperparameters in a given problem.

For each environment and agent, the hyperparameters used are given in Tables 3.1 to 3.3

Table 3.1: Parameters used by the Dynamic and DQN agent in the Noisy Berry environment. The Dynamic agent did not converge in number of network parameters in the case of activation threshold $\theta = 0.0$, marked with *, but did converge with $\theta = 0.5$.

Parameter	Dynamic agent	DQN agent
Batch size	1	8
Memory capacity	1	20000
Memory α	0.0	0.7
Discount γ	0.95	0.95
Learning rate η	0.0005	0.05
Target update interval τ	5	25
ϵ decay steps	5	5
Focus set size M	1000	
Activation threshold θ	0.5*	
Surprise threshold	2.2	
μ_{th}	0.2	
σ threshold	0.3	
Weight threshold	0.5	
Max number of inputs	3	
Hidden neurons		[16]

Table 3.2: Parameters used by the Dynamic and DQN agent in the Grid environment.

Parameter	Dynamic agent	DQN agent
Batch size	8	8
Memory capacity	8000	8000
Memory α	0.7	0.7
Discount γ	0.95	0.95
Learning rate η	0.01	0.01
Target update interval τ	50	50
ϵ decay steps	5	200
Focus set size M	1000	
Activation threshold θ	0.0	
Surprise threshold	2.2	
μ_{th}	0.2	
σ threshold	0.5	
Weight threshold	0.2	
Max number of inputs	3	
Hidden neurons		[32]

Table 3.3: Parameters used by the Dynamic and DQN agent in the Discrete Catch environment.

Parameter	Dynamic agent	DQN agent
Batch size	48	48
Memory capacity	8000	8000
Memory α	0.7	0.7
Discount γ	0.95	0.95
Learning rate η	0.005	0.005
Target update interval τ	50	50
ϵ decay steps	300	300
Focus set size M	1000	
Activation threshold θ	0.0	
Surprise threshold	2.2	
μ_{th}	0.2	
σ threshold	0.5	
Weight threshold	0.2	
Max number of inputs	3	
Hidden neurons		[16, 32, 32]

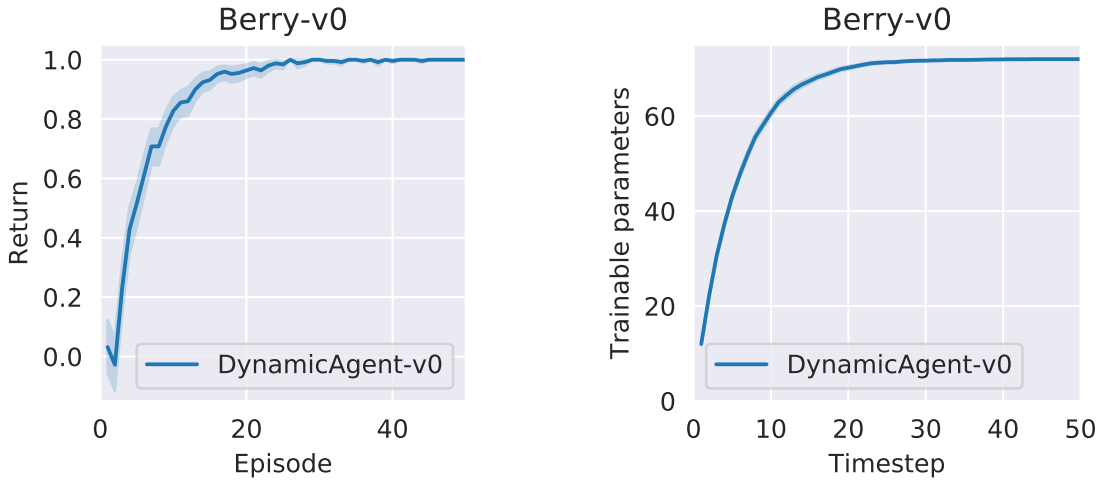
4

Results

In this chapter the results of the evaluations in the environments from Section 3.4 using the parameters from Section 3.5 are presented and highlighted. All results are averaged from many evaluations.

4.1 Berry

The Dynamic agent successfully classifies all berries in this environment, see Figure 4.1. As soon as it encounters a new berry, it creates a concept node, so that the next time it sees the same berry it will know which action to choose, resulting in receiving the maximum return per episode after about 25 episodes, see Figure 4.1a. The Dynamic agent requires 72 parameters for the concept nodes of all six berries. This is also what the network converges to, see Figure 4.1b. The DQN agent was not evaluated in this environment as the purpose was to verify the Dynamic agent.



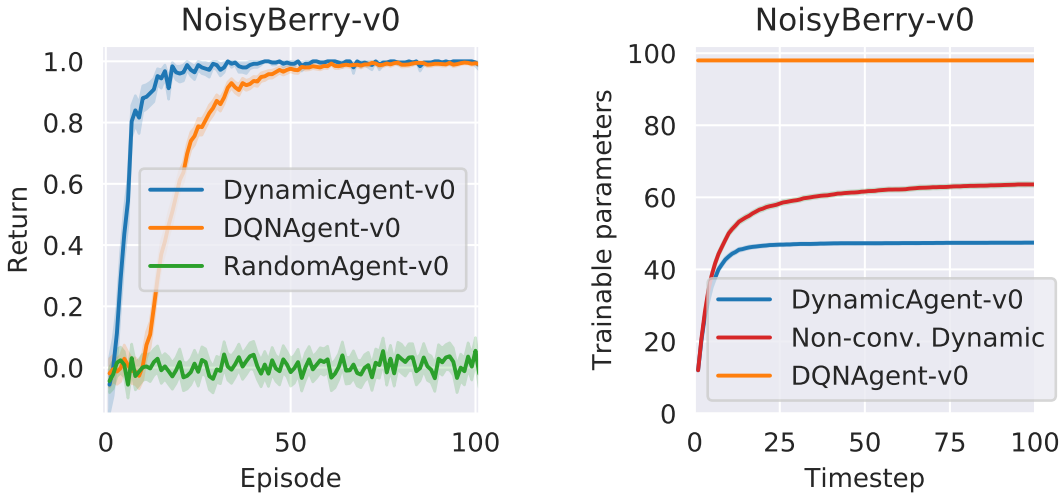
(a) The Dynamic agent learns how to receive a high return by differentiating the different berries.

(b) The Dynamic agent adds more parameters with each new concept node as it is presented with an unseen berry.

Figure 4.1: The Dynamic agent, using Double DQN and training on the current state, in the Berry environment.

4.2 Noisy Berry

In the Noisy Berry environment, see Figure 4.2, the Dynamic and the DQN agent both learn how to solve the task quickly. The DQN agent is behind from the start as it requires a delay since it uses a larger batch size in return for faster learning later. However, the Dynamic agent manages to learn faster than the DQN agent, in the sense of how much increase in return per episode the agents receive, see Figure 4.2a, despite the DQN agent using a larger batch size and higher learning rate. Figure 4.2b shows that the Dynamic agent quickly creates the concept nodes required for the environment. It does not always converge in this environment though; an activation threshold $\theta = 0.5$ leads to a converging network in the Dynamic agent, while $\theta = 0.0$ leads to a non-converging, or at least larger and slower converging network.



(a) The Dynamic agent learns slightly faster than the DQN that has a delayed training because of its larger batch size. Both agents use Double DQN and prioritized experience replay. Random agent as a reference.

(b) The DQN agent is static in size while the Dynamic agent creates new concept nodes when encountering an unrecognized state. The convergence of the Dynamic agent depends on the activation threshold θ.

Figure 4.2: DQN and Dynamic agent in the Noisy Berry environment. Note that in the Noisy Berry environment an episode consists of a single timestep, so Figures 4.2a and 4.2b are directly comparable.

4.3 Grid

The Dynamic agent outperforms the DQN agent in average episode return, see Figures 4.3a and 4.3b, while requiring only a fraction of the number of parameters

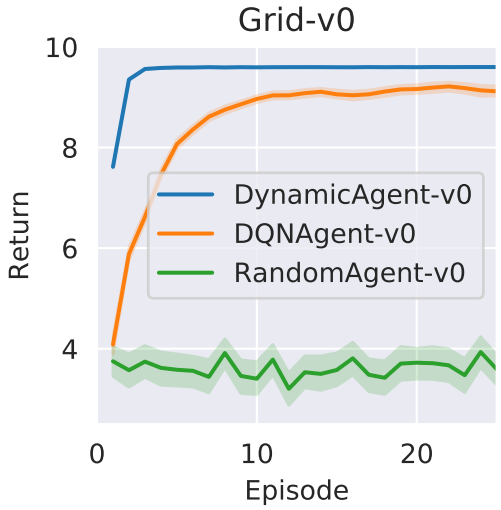
used by the DQN agent, see Figure 4.3c. The Dynamic agent learns how to reach the reward-tile and so does the DQN agent, although not quite as fast. Unlike the Dynamic agent, the DQN agent also does not learn to walk there in the fewest number of steps possible.

It is clear from Figures 4.3a and 4.3b how important Double Q-learning and prioritized experience replay is for the DQN agent, and that the Dynamic agent is insensitive to the lack of these improvements in this environment.

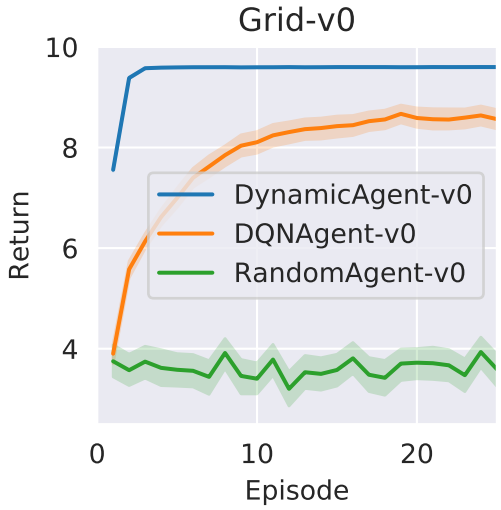
Note that the DQN agent is using a tuned network architecture and hyperparameters for optimal performance. The Random agent sets the baseline in this environment, showing that a random walk where some actions in certain states do not lead to any movement, it requires some 65 steps on average before reaching the reward-tile.

4.4 Discrete Catch

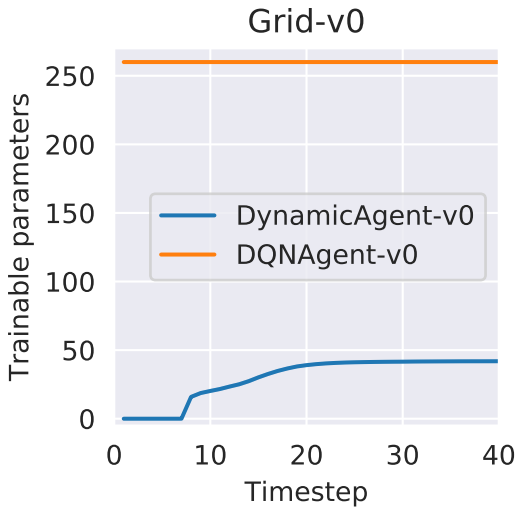
The DQN agent quickly learns how to solve the Discrete Catch environment, while the Dynamic agent falls behind, see Figure 4.4. The hyperparameter search for the Dynamic agent did not affect the results noticeably, with the exception of the number of decay steps used, indicating an otherwise insensitivity to the change of parameters in this environment. With more and more forced exploration for the Dynamic agent, regulated by the decay steps, it learns the environment better at the cost of more concept nodes. Note that as the Dynamic agent approaches the DQN in performance, the parameters required also approaches the number of parameters in the DQN agent.



(a) The agents use Double DQN and prioritized replay. The DQN agent learns how to reach the reward-tile quite fast, although not as fast as the Dynamic agent that learns the environment after just a couple of episodes.

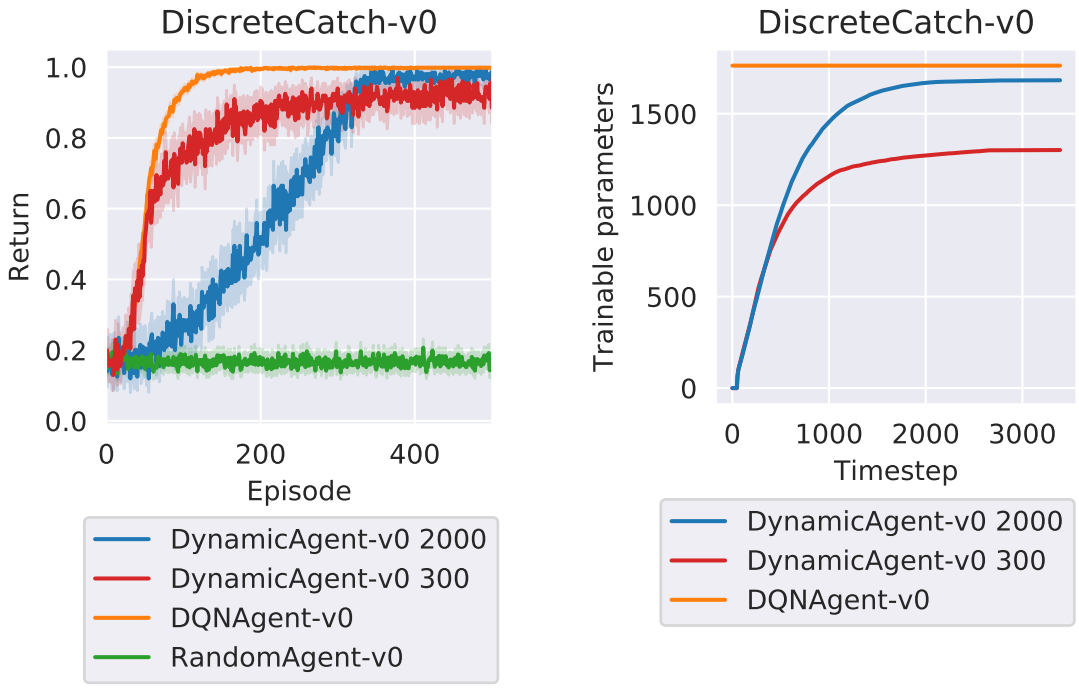


(b) The agents use standard DQN algorithms and uniform replay. The Dynamic agent performs similarly as with improvements, however the DQN agent learns slower, not as well, and not as stable as with learning improvements enabled.



(c) Number of trainable network parameters in the two agents. The delay of learning and creation of nodes, equal to the batch size, can be seen. After this, the dynamic network creates nodes until it converges after about 20 timesteps.

Figure 4.3: DQN and Dynamic agent in the Grid environment with Random agent as a reference. The same Random agent was used in Figures 4.3a and 4.3b as it is unaffected by learning improvements.



(a) Depending on the decay steps used by the Dynamic agent, it either quickly learns to get decent, or slowly learns to get good. The DQN agent shows better performance regardless.

(b) With more exploration the Dynamic agent creates more nodes, allowing for a better understanding of the environment.

Figure 4.4: DQN and Dynamic agent in the Discrete Catch environment with Random agent as a reference. A number after the Dynamic agent indicates the number of decay steps used for exploration.

5

Discussion

In this chapter the results from experiments, the dynamic model itself, related works, and future work, will be discussed.

5.1 Results

The results from evaluating the two agents, see Chapter 4, are discussed individually for each environment in Section 3.4. We try to give an explanation of the results based on how the dynamic model works in contrast to the DQN used for comparison.

5.1.1 Berry

The Dynamic agent performed well in the Berry environment, as it is a very simple classification problem where the true Q-values are given as rewards for all berries. One node for each berry is generated after an encounter, which is exactly as expected as the environment was designed to not have any large enough overlap in the state features for a concept node that actually remembers a different berry to be activated. This in turn leads to an empty focus set for every not previously encountered berry, which in combination with the rules for one-shot learning lets the agent create a concept node for each new berry, and never picks the wrong action with that berry if completely greedy. In conclusion, this environment shows both that the concept nodes are working as expected and that the dynamic model works for at least some classification tasks.

5.1.2 Noisy Berry

Similarly to the binary-taste berries, the model performs well in classifying noisy berries. The Dynamic agent learns faster than the DQN agent even when the DQN uses an aggressive learning rate of 0.05 that still allows it to converge. This is possible for the Dynamic agent solely from being able to learn required parameters from a

single sample by one-shot learning and, unlike the DQNs learning, not needing to change existing parameters by backpropagation.

What was interesting with this environment was introducing one taste to be purely random for one of the berries. This caused the creation of a few "duplicate" nodes as the second experience with the same berry could have a taste very different from the first. Here we saw that as training went on, either the σ of the Gaussian function grew, leading to wider variations of the random taste being accepted, or even more to our satisfaction, the weight of that input to the concept node decreased, meaning that the input was treated as less important compared to the others. This was precisely what we hoped to see, as this is what is required for abstraction to work, as described in Section 2.4.5.

For values of the activation threshold θ too far from 0.5, the dynamic network no longer converges in the number of parameters, see Figure 4.2b. This was seen both when increasing and decreasing the activation threshold. This could be because there is a certain range where the nodes created by the berry with one purely random taste are included in the focus set, but nodes for other berries are not. As the nodes in the focus set and their predecessors are trained by backpropagation, a focus set containing the "wrong" nodes could explain the non-convergence.

5.1.3 Grid

Grid is the simplest environment used that introduces delayed rewards. This means that this is the first environment where the discounted reward of Q-learning truly comes into place. The Dynamic agent starts to explore completely at random until it either runs into a wall, upon which it receives a negative reward, or it reaches the reward-tile that provides a reward based on the number of steps the agent has taken. Either reward opens up for the creation of a concept node that will either discourage or encourage an action, based on the reward, in the future. When encountering the same state again, the agent will always choose an action that either makes it move towards the reward, or at least not make the same mistake as before.

This environment confirmed that the temporal learning of the Dynamic agent worked as intended by inspecting the estimated Q-values associated with each state and action. Although the environment being simple, this is a proof that the Dynamic agent has the possibility of learning from delayed rewards. It also shows that the Dynamic agent has the potential to outperform a regular neural network in some types of problems.

5.1.4 Discrete catch

The Discrete Catch environment is the first environment tested in which the DQN agent outperforms the Dynamic agent. Given the same amount of decay steps, they

both learn at the same rate initially, but the Dynamic agent soon falls behind the DQN agent that has completely solved the environment after about 200 episodes. The Dynamic agent does not seem to be able to learn the environment perfectly, although it comes close as the number of decay steps is increased from 300 to 2000. Interestingly, the number of parameters used by the Dynamic agent also comes close to the number of parameters in the DQN agent as the decay steps are increased. This is probably a coincidence, as we saw that the Dynamic agent solved it decently with a smaller number of decay steps, that lead to the creation of fewer concept nodes.

Since this environment has a much larger state space than any of the previous environments, it is almost expected that the Dynamic agent has some difficulties, considering that it makes its decisions based on memories of specific states. For example, there are 90 different states in which the ball is to the left of the paddle, meaning that a lot of concept nodes are required for something that should not be that hard to realize. There is still a possibility that the σ used for computing similarity of states is trained by backpropagation to be large, this is however unlikely, as it would require that states with the only difference being the vertical position of the ball is trained on many times so that backprogagation leads to a large σ that can be abstracted. Even if this managed to happen, the agent would still need 15 concept nodes for a concept that one could argue should be able to be recognized with a single node, testing if the horizontal position of the ball is smaller than that of the paddle or not.

5.1.5 Continuous catch and CartPole

The Dynamic agent performed worse in the Continuous Catch and CartPole environments. The runtime performance is also poor because of the large number of concept nodes created following the, in practice, infinite state spaces of these two environments with multiple real valued and continuous inputs. This issue made a search for hyperparameters practically infeasible, and no results are available in these environments beyond the fact that the Dynamic agent has issues in large state spaces because it is trying to remember everything in its current stage, while the DQN agent still manages to solve the environments.

After a more successful solution to an environment similar to Discrete Catch, environments like Continuous Catch and CartPole could be tested if they are appropriately sized so that the Dynamic agent does not have to create too many concept nodes for the variation of a single input, and that it instead could be taken care of by the σ in the similarity function, or with a different kind of function used to compute similarity of states.

5.2 The dynamic model

Here we present what we found to be the most prominent advantages and limitations of the model, as well as discuss the use of the focus set.

5.2.1 Advantages

- The most obvious advantage of the dynamic model is that one does not have to choose a network architecture. Because of the endless possibilities in this design, which in a truly exhaustive search should all be evaluated with another exhaustive search of the other hyperparameters, getting optimal results with a traditional DQN is time consuming and inefficient. The dynamic model adds some new hyperparameters, but removes the need to choose the number of layers and the size of each layer.

- As the dynamic model supports immediate learning via creation of new nodes for surprising events in "new" states, it can quickly learn simple behaviours.

- The model seems to generalize well in the sense that reconfiguration of hyperparameters between environments does not considerably affect its decision-making performance.

5.2.2 Limitations

- As the idea behind the nodes of the network in the dynamic model is rather different than in traditional neural networks, the model itself has different properties. In the implemented dynamic model there is no notion of differences between state values, just memories of specific situations, e.g. "the ball was at $x = 0.5$, and is now at $x = 0.7$", instead of "the ball is at $x = 0.7$ and traveling with velocity $v_x = 0.2$". This limits the level of complexity in input that the model is able to handle as it in its current state is not as good at function approximation as it is at classifying inputs.

- To avoid redundant memories, the difference between stored values in concept nodes is compared to a hyperparameter threshold value. If different environments have different magnitudes of the state features, the threshold might need to be reconfigured for each environment.

- Transfer learning is likely to be an important part in successfully creating a more general model that can learn multiple problems after each other. The current dynamic model is not really capable of transfer learning between problems unless the state space of the problems overlap at least somewhat. In regions where there is no overlap it is impossible to transfer any knowledge

since the model cannot have any memories from inputs it has previously never seen. For instance, learned knowledge from the Discrete Catch environment is only transferable to the same environment with for example two more units of height. If the problem is translated enough along the horizontal axis, the remembered positions will lie outside of the input space, and nothing can be transferred.

- One of the major drawbacks of the dynamic model is performance. Because of the complexity of the model, the programs logical flow takes much more time than the mathematical computations. As the majority of this logic is sequential in contrast with the computationally heavy matrix multiplications in DQNs, it cannot benefit from parallelization in the same way. What can be done in parallel is the forward pass of each individual layer, as concept nodes in a single layer are guaranteed to be independent of each other.

5.2.3 Focus set

The focus set used by the Dynamic agent is necessary when adding new nodes, as this tells the model on which concepts it should build a new node from when the agent made a bad estimation of Q-values. For decision-making however, we have seen that setting the focus set size to 1000, a number larger than the number of concept nodes created in any of the environments the Dynamic solved, we observed no significant change in its decision-making performance. The reason for this is probably because any unwanted interference in the estimation of Q-values will be minimal as the "wrong" concept nodes does not get a high enough activation to have a notable impact on the combined output of all concept nodes.

However, an activation threshold for nodes to be included in the focus set still seems to be necessary for the network to converge and to avoid catastrophic interference during training. The activation threshold was the difference between a converging and a non-converging network in the Noisy Berry environment, as this allowed the model to train only the concept nodes that were actually relevant to the current state.

5.2.4 Lifelong learning, transfer learning, and generality

Regarding the three desired features lifelong learning, transfer learning, and generality, of a general intelligence, the dynamic model has the potential to better fulfill these desires than regular neural networks if given more work.

Lifelong learning is achieved because there is no limit in how many parameters there are in the network as a result of a static architecture. If the model encounters some input upon which it acts and receives negative feedback, it will try to learn how to make a better decision the next time it faces a similar input. It will do this without

changing its current parameters that are needed to perform well given other inputs, in contrast with how a regular neural network would and in the process "forget" what to do given those other inputs. This is however also a limitation of the model, as it may try to remember every input it has received and the size of its network will grow beyond being practically usable.

Transfer learning is as previously mentioned considered to be achievable by the model as well, but not with how it currently tries to recognize states. This recognition limits the transfer of previous knowledge to the exact same input space in the new problem, which is an issue that could be solved by remembering relations between the inputs instead of exact values.

In the context of generality a similar issue as with transfer learning is encountered; although a dynamic architecture allows for a more general model than a static, the parameter μ_{th}, used to set the minimum difference between state input values for them to be considered unique memories, does need to be adjusted according to the magnitude of the state features of each problem. For a truly general model, this would need to be addressed.

5.2.5 A parable with table-based Q-learning

As the dynamic model builds a network of nodes for recognizing concepts and the estimated Q-values associated with each concept, the network can be seen as a compressed Q-table used in traditional Q-learning, combined with learning à la backpropagation of neural networks. This is an interesting approach, as Q-tables are the optimal solution, but are unfeasible when the state-space is large, leading to neural networks being used as Q-function approximators. The implemented proof-of-concept only uses value memories of states, but could be extended with more types of nodes for better Q-table compression e.g. relation memory nodes.

The view of the dynamic model as a combination of traditional table-based Q-learning and deep Q-learning networks could serve both as an intuitive explanation of why it works, and a point of view during further development.

5.3 Previous models with dynamic architectures

The ideas behind the dynamic model and the previous models Progressive Neural Networks and The Cascade-Correlation Learning Architecture are compared and discussed in this section. The focus is to emphasize similarities and differences, and what they do better or worse compared to each other.

5.3.1 Progressive Neural Networks

While the progressive network approach introduced by [12] is one way of building an agent that is capable of lifelong and transfer learning, the approach is not sustainable as the number of problems an agent is trained on is increased, as pointed out by the original authors Rusu et al. The idea behind the dynamic model suffers somewhat from the same issues. Just like a progressive network, the dynamic model also requires evaluation of all its concept nodes, and therefore the complexity will also grow for each new problem. To handle this, the concept nodes that are used could be changed so that they instead of remembering a single state and receiving a high activation when that state is recognized, they should more remember a certain relation between its inputs. Such a change would make a concept node be able to have a high activation for many different inputs instead of a single one, reducing the number of nodes needed.

Regarding transfer learning the dynamic model does not build a new node from all previous nodes, as a progressive network does. Instead, by using the focus set as input candidates it only build connections to previous nodes that are guaranteed to be capable of some contribution. This is also commented on by Rusu et al. as they suggest that the growth of parameters with new problems can be held back by adding fewer layers or by pruning connections, with the latter being more similar to how concept nodes are created. Additionally, progressive networks need information on which column of the network should be used when evaluating a specific problem. This is not the case for the dynamic model as concept nodes are not created and trained for a single problem. An advantage of this is that it is not required to specify what the current task is. A major drawback, though, is that the entire network is always used. Consequently, parts of the network that may be completely irrelevant to the problem at hand are still considered, leading to a lot of unnecessary work being done. A two step approach, where the problem is first identified in order for the correct sub-network to be applied to the problem, could also improve on unnecessary work being done while still using only the relevant knowledge. A final remark on how the progressive networks handle transfer learning is that if something is learned in a successive problem, this can by design never be used in a previously learned problem, even though the knowledge could be of great value. In contrast with how a progressive net has multiple columns, one for each problem it has trained on, the dynamic model does not have the same issue as nothing is learned for a specific purpose.

5.3.2 Cascade-Correlation Learning Architecture

The idea in Cascade-Correlation to train candidate neurons to maximize the covariance of the candidate neuron activation and the output residual error is quite similar to how new concept nodes are created in the dynamic model. Maximizing the covariance means that for all inputs which generate a large residual error, we

want the activation of the candidate neuron to be high. When the dynamic model experiences a large enough error to create a new concept node, the inputs to this node are chosen so that a similar input state will result in a high activation. As the new concept node will be more detailed than its inputs, any inputs that are concept nodes will be excluded from the focus set and will not affect the output. As a result, the attempt at one-shot learning by the new node will ensure that a high activation will be achieved from an input that previously generated a large error. Because Cascade-Correlation has previously showed to be suitable not only for supervised learning, but also for reinforcement learning problems [16, 17], this similarity of how new nodes are added to the two models suggests that the dynamic model could also have the potential to outperform static neural networks at both supervised and reinforcement learning. We have seen indications of this being true from our experiments as well, which motivates further work with the dynamic model.

What Cascade-Correlation does not offer when compared to the dynamic model is the ability to reduce the network when nodes that have been added are no longer needed, along with the properties of lifelong and continuous learning; there is a distinct training phase for adding new nodes when the existing nodes are not trained, after which the network architecture is fixated. At this point, the network can be used but nodes are neither added nor removed, and only the last layer is trained further. For a general-purpose AI that should ultimately be able to encounter any number of situations during its lifetime, this is not optimal as there could be later times when the higher-order features detected in the now fixed layers are never encountered again, or more importantly when there are other features present that cannot be detected by the fixed layers.

5.4 Future work

Here we provide our thoughts on how the model could be improved to better handle more complex inputs and what should be considered if continuing to work with the model in the future.

5.4.1 Limitless focus set

In our evaluations we found that there seemed to be no benefit in limiting the number of nodes to be included in the focus set. On the contrary, letting all nodes with sufficiently high activation be included in the focus set allowed the model to extract the important parts of concepts, ultimately leading to better generalization and convergence of the network. With this knowledge it would be possible to skip creating the focus set and instead only use the threshold currently required for being included in the focus set.

5.4.2 Different types of nodes

To let the dynamic model handle more complex situations than a momentary picture of the environment, one could introduce other types of nodes, which instead of simply remembering values can remember relations, e.g. differences, between values. This would allow the agent to handle temporal differences without having to create a node for each specific state. This could be especially useful in environments such as the Discrete and Continuous Catch environments, where it is the relations between state inputs that are interesting, e.g. a single concept node for "the ball is to the left of the paddle, move left" instead of a concept node for each state in which the ball is to the left of the paddle. This could give rise to a ball-following behaviour using much fewer concept nodes. Although this kind of behaviour is not generally preferred when the paddle is unable to keep up with the ball in speed, as we would like the agent controlling the paddle to come up with a deeper understanding of how it does not need to chase after a ball that is just about to bounce back from hitting a wall, it would not force the behaviour on the agent. It does however provide a more interesting way of building networks that potentially could solve more complex environments.

5.4.3 Runtime performance

The implemented dynamic model compared to a regular DQN is not as fast with regards to runtime on a CPU. There are three main reasons for this:

1. DQN implementations are normally done with matrices and matrix operations that are very efficient on GPUs because of how well they parallelize. This is evident even when running on a CPU when the networks are small enough as to not being able to utilize as many computational cores, as in our experiments. Although some matrices and matrix operations were used in the implementation of the Dynamic agent, the matrices were all very small, e.g. a matrix as input to every concept node and vectors with weights, and the gain in performance from this is therefore also small. This area should be further investigated however. We believe that an implementation with larger regular matrices or perhaps a sparse matrix representation would perform better.

2. There is a lot of logic required for the idea to be applied to reinforcement learning that is not required in for example supervised learning, where we have seen the model to perform well in terms of inference from the berry environments.

3. The prime focus of the thesis was to implement and evaluate a proof of concept of the new idea for a dynamic network architecture within reinforcement learning, in order to see if the idea has any potential. The focus was not to implement the Dynamic agent in the most computationally efficient way, as there was no telling if it would work at all in the first place. Emphasis has

therefore been on getting further understanding of the model and implementing it in a way that makes it easy to follow instead of trying to optimize all of the previously mentioned logical operations required.

If the model should be useful in practice the runtime performance of the implementation needs to be improved upon. An example of such improvements, although small, could be to apply batch-learning and generalization at set intervals, instead of at every iteration.

6

Conclusion

The dynamic model works well in the case of classification tasks and the simple Grid environment, both in which it outperforms the DQN. As the state space grows larger in the Discrete Catch environment, and even more so in Continuous Catch and Cart-Pole, the dynamic model will require further improvements in order to keep up with the well-established DQN. Noteworthy similarities between the dynamic model and Progressive Neural Networks and the Cascade-Correlation Learning Architecture, which both have proven successful in reinforcement learning applications, indicate that the dynamic model has potential to do the same given more time and effort. Additional comparisons with the purpose of drawing inspiration from said techniques could also be useful for future development.

The dynamic model can be seen as a combination of classical table-based Q-learning and the more recent DQNs, as it in a sense is a compressed table of Q-values in the form of a neural network. This parable is interesting, as both Q-learning and DQNs have been important milestones in reinforcement learning. With this reasoning in mind, it would be compelling to develop the model with better ways to create a mix of a compressed Q-table and a function approximator, for example with the addition of relations between values rather than just combinations of specific values.

As DQNs can utilize optimized parallel matrix multiplications they are hard to beat performance-wise, and might not even need improvements in this regard. Instead, the approach presented in this thesis could be interesting for the research of more general techniques, and useful for cases where for example the agent has to learn quickly from just a few experiences.

As the implemented dynamic model is merely a proof of concept of a brand new reinforcement learning algorithm, it would be naive to expect the first implementation to outperform such a successful algorithm as the DQN. We consider it an interesting idea with a first round of results that merit further development and research.

To continue working with the dynamic model, the Discrete Catch environment or one of similar difficulty with regards to state space and complexity in the input is probably a good starting point to test whether new ideas solve the issues of memorizing states and runtime performance.

Bibliography

[1] L. Hardesty, "A method to image black holes," https://news.mit.edu/2016/method-image-black-holes-0606, MIT News, 6 June 2016.

[2] H. van Praag, A. F. Schinder, B. R. Christie, N. Toni, T. D. Palmer, and F. H. Gage, "Functional neurogenesis in the adult hippocampus," *Nature*, vol. 415, no. 6875, p. 1030, 2002.

[3] P. S. Eriksson, E. Perfilieva, T. Björk-Eriksson, A.-M. Alborn, C. Nordborg, D. A. Peterson, and F. H. Gage, "Neurogenesis in the adult human hippocampus," *Nature medicine*, vol. 4, no. 11, p. 1313, 1998.

[4] C. Strannegård, "Dynamic networks," Manuscript, 2019.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[6] 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*, 2016. [Online]. Available: http://arxiv.org/abs/1511.05952

[7] H. van Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, 2010, pp. 2613–2621. [Online]. Available: http://papers.nips.cc/paper/3964-double-q-learning.pdf

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

[9] W. Mischel, E. B. Ebbesen, and A. Raskoff Zeiss, "Cognitive and attentional mechanisms in delay of gratification," *Journal of Personality and Social Psy-*

chology, vol. 21, pp. 204–218, 02 1972.

[10] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[11] J. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation (technical report lids-p-2322)," *Laboratory for Information and Decision Systems*, 1996.

[12] 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. [Online]. Available: http://arxiv.org/abs/1606.04671

[13] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," *Advances in Neural Information Processing Systems*, vol. 2, 10 1997. [Online]. Available: http://web.cs.iastate.edu/~honavar/fahlman.pdf

[14] R. Eldan and O. Shamir, "The power of depth for feedforward neural networks," in *29th Annual Conference on Learning Theory*, ser. Proceedings of Machine Learning Research, V. Feldman, A. Rakhlin, and O. Shamir, Eds., vol. 49. Columbia University, New York, New York, USA: PMLR, 23–26 Jun 2016, pp. 907–940. [Online]. Available: http://proceedings.mlr.press/v49/eldan16.html

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[16] F. Rivest and D. Precup, "Combining td-learning with cascade-correlation networks," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 632–639. [Online]. Available: https://www.aaai.org/Papers/ICML/2003/ICML03-083.pdf

[17] P. Vamplew and R. Ollington, "On-line reinforcement learning using cascade constructive neural networks," in *Knowledge-Based Intelligent Information and Engineering Systems*, R. Khosla, R. J. Howlett, and L. C. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 562–568.

[18] A. Géron, *Hands-On Machine Learning with SciKit-Learn and TensorFlow*, 1:st ed. O'Reilly Media, Inc., 2017.

A

Appendix

A.1 PyTorch

PyTorch is a computing package for Python, well suited for machine learning tasks, that can run computations on either a CPU or a GPU. One important difference between PyTorch and one of its main alternatives TensorFlow is that PyTorch provides a define-by-run framework for automatic differentiation of the computations done with tensors. This is optimal for a network such as [4], as the output, and therefore the loss, will depend on different nodes each iteration. PyTorch makes it possible to ignore this, as it builds a new graph each run, depending on which operations were used to compute the final output. Therefore, it is able to compute only the necessary gradients for backpropagation depending on the path of computations in the forward pass of a network.