University of Exeter
Department of Computer Science

Meta-Learning Through Learnt Self-Addition

Alexander Wild

January, 2017

Supervised by Professor Richard Everson

Submitted by Alexander Wild, to the University of Exeter as a thesis for the degree of Master of Science by Research in Computer Science, January, 2017.

I certify that all material in this thesis which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other University.

Abstract

This thesis presents a meta-learning architecture designed to form an agent able to operate in a classic reinforcement learning environment. Drawing on several existing meta-learning techniques, this agent learns its environment by subdividing it between multiple predictive components, each with their own machine learning capabilities.

These components are competitive and co-operative, competing for 'worth', which is used periodically to remove under-performing components and direct the creation of new ones. Components compete to make as many accurate predictions as possible, balancing number of predictions made against the accuracy they can achieve. Co-operation comes from trading information, either prediction values or memory data, to other components, in return for a portion of any worth the other component receives.

Components may vary in internal architecture. They can store different information, can use different machine learning algorithms, can share different information or can subdivide the environment in different ways. They are all measured by the same worth metric, so the agent's component set will consist of a highly heterogeneous pool, determined by which components work best in which roles.

This creates a complex set of multiple machine learners, arranged into several trees of information-suppliers and information-user, rather than depending on a single machine learner to handle the entire problem. Meta-learning comes from the agent's ability to learn how to create these structures. It learns to predict which component types will work best in which circumstances, as well as learning which parameters to provide these with when it creates them.

This thesis evaluates the basic properties of such an agent under different conditions. Tests cover its ability to correctly evaluate the worth of its components, its ability to learn to usefully select which types of new components to generate, and its ability to learn from one task to improve its performance on the next. Experiments on a variety of learning problems confirm the architecture is able to exhibit the required component creation, deletion and balancing with no internal restructuring between tasks.

Several possible future expansions are also explored, using the component-worth metric as a metric for the value of information. Testing suggests this allows the agent to be able to learn an information-seeking drive, which directs it to move to a source of information in

order to solve a later task, rather then depending exclusively on the information it receives passively.

Contents

1 Introduction **7**

1.1 Novel Contributions 8

1.2 Introduction to rest of work 8

2 Background **10**

2.1 Meta-Learning Approaches 10

2.1.1 Introduction 10

2.1.2 Portfolio 12

2.1.3 Parameter Learning 13

2.1.4 Ensemble methods: Combination of classifiers 13

2.1.5 Input features available to preceding approaches 16

2.1.6 Meta-Cognition 16

2.2 Proposed Agent in Above Context 17

3 SAMLA: Self-Additive Meta-Learning Agent **19**

3.1 Structure 20

3.2 Example of architecture as described thus far 21

3.3 Component prediction generation process 23

3.4 Component Error Comparison 24

3.5 Valuing and Removing Components 26

3.6 Component Addition and Meta-Learning 29

3.6.1 Pseudocode: Agent's main decision loop 30

3.7 Similar Architectures 32

3.7.1 Self-modifying AERA and EXPAI frameworks 32

3.7.2 The Hayek machine 36

4 Validation Experiments **38**

4.1 Introduction 38

4.2 Component Valuation Test 38

4.3 Meta-Learning Demonstration, using the Weka toolkit and the MNIST dataset 46

4.3.1 Structure of the test 47

4.3.2 Results 48

4.4 Component Stacking: Deduction of hidden variable from recent error rate . 49

4.4.1 Task 49

4.4.2 Component Architecture 50

4.4.3 Results . . . 51
4.5 Reinforcement Learning mechanism . . . 52
4.6 Reinforcement Meta-Learning: Useful feature discovery . . . 53
4.6.1 Task . . . 53
4.6.2 Results . . . 55
4.6.3 The Failed Agent . . . 56
4.7 Expanded MNIST testing . . . 57
4.8 Results . . . 58
4.9 Discussion of results . . . 62

5 Experiment into Self-Directed Information Seeking 63
5.0.2 Competitive Hidden Variable Predictions . . . 64
5.0.3 Hidden Variable Prediction Testing . . . 65
5.0.4 Hidden Variable Prediction vs Conditional Memory . . . 69
5.0.5 Recorded properties of objects . . . 70

6 Future Work 73
6.1 Evolving components . . . 73
6.2 Self-Structured Learning . . . 74
6.2.1 Curiosity: internal reward for creation of components . . . 74
6.2.2 Alternative signals for prediction . . . 74
6.2.3 Self-directed Reinforcement Learning . . . 75
6.2.4 Conditional Memory Components as Information Utility Measures . . . 76
6.2.5 Overview of agent with planned future features . . . 78

7 Conclusion 81

Nomenclature

s_t	The state presented to the agent at time-step t
V_t	The state vector from s_t
A_{ti}	The available action with index i from s_t
C_j	The component with index j
O_j	The output of component with index j
R_t	The reward received by agent at time t
$\hat{R}_{tij}$	Prediction of reward for action i by C_j at t
L_j	C_j's mean squared error over all predictions $\hat{R}_*$
D_j	C_j's baseline prediction, the average of all rewards $\hat{R}_*$
B_j	C_j's baseline error, equivalent to the error of a maximally simple predictor
G_j	C_j's adjusted error metric
Q_{tj}	C_j's total accumulated worth at time-step t
W_{tj}	C_j's total accumulated worth divided by its age at time-step t
$\mathscr{C}$	Set of all components within the agent
$\mathscr{P}$	Set of predictions gathered for the currently presented action
$\mathscr{R}$	Set of selected predictions for all actions available in s_t

1 Introduction

This thesis presents, we believe, a novel architecture for an agent operating within a classic Reinforcement Learning (RL) environment. The aim is to accelerate learning of new environments by carrying over information learnt from previously solved problems. To accomplish this the agent meta-learns the over-arching structural patterns in environments of the type it will be presented with by learning a set of similar but different environments before being tested against the "real" trial environment.

The architecture is termed SAMLA, short for Self-Additive Meta-Learning Agent.

The aim is to produce an agent which predicts the effects, in terms of reward, of its actions. This would allow it to operate in a reinforcement learning environment by continuously predicting the utility function's outcome and selecting actions to maximise it. To accomplish this prediction the agent has a set of what are termed 'components', each of which is able to predict the future reward within a subset of the agent's environment. The subset of the environment a given component makes predictions for is termed its 'scope'. Components exist competitively, each attempting to maximise the number of predictions it makes against the accuracy of these predictions.

Quantity of predictions must be balanced against quality, as only a single component is chosen as the 'expected most accurate predictor' for any given prediction, and components are judged based on the number of times they were chosen. Specialisation, making low numbers of predictions, allows a component to make highly accurate predictions, but limits the number of chances it has at being selected. Generalists can cover gaps which no other component is competing for, but suffer in terms of accuracy, so lose to specialists if those elect to make a prediction on a given time-step. A key aspect, however, is that each component may be using different problem-space sub-division strategies, leading to a competition amongst components not only to find the correct degree of specialisation, but also to find the correct dividing lines in the environment, along which to specialise.

The structure of components is arbitrary, from the agent's perspective. They all possess the same broad structure, with inputs and outputs taking standard forms, allowing them to link to each other and the agent in a consistent manner, but they may vary greatly in terms of internal structure. The competitive approach selects for useful components, while culling useless ones, as determined by their accumulated reward. This allows a variety of different machine learning algorithms to be made available to the agent, each implemented by a different component type. Despite differing internal structures, the agent is able to

treat them in the same manner, and they are able to be evaluated against each other with a common metric.

Meta-learning is accomplished by the agent itself learning which component structures are useful for the learning tasks it faces in its environment and which are not, and deliberately adding new useful component types to itself. As well as simply selecting the type, the agent is able to direct the new components to take data from existing components, as well as select parameters for the machine learning algorithms contained within the component. This component-placement behaviour is learnt by standard reinforcement learning mechanisms, and uses the same component-usefulness measure which is used to determine which components to preserve and which to remove from the set.

This allows a wide variety of separate well-known machine learning algorithms to be tested, used and combined by the high-level agent. As such, this system can harness the power of tried and tested algorithms, from k-means to tile-based RL approaches to recurrent memory neural nets, discarding approaches which are less useful, and attempting to combine those which prove successful. As such, the agent is able to learn which of these are best suited to the type of environment it is facing, and can predict these environments using the most efficient and effective of the tools it has been provided with, without need for designer knowledge.

The architecture is inspired by Baum's [3] work on "Intelligence as an economy of agents". It shares the same basic premise of a set of competitive and co-operative agents using an economic model as means of removing useless or unneeded agents from the set but diverges sufficiently from this to be considered a separate approach.

1.1 Novel Contributions

The novel contributions this work is believed to bring are as follows:

- The architecture is able to combine two major approaches to meta-learning, portfolio-of-algorithms and stacking, into a single agent in a way which allows the strengths of both to be exploited.
- The architecture is able to perform meta-learning in reinforcement learning environments, a relatively understudied area of meta-learning
- The architecture performs a degree of meta-cognition, and does so in a way currently unexplored in the literature. This is demonstrated in a test of the agent's ability to discover hidden variables based on its own error rate.

1.2 Introduction to rest of work

The rest of this work is structured as follows:

Chapter 3 We discuss other works in the field of meta-learning. We review the major branches and approaches to the field, and then attempt to place this work's architecture in the context of this existing literature.

Chapter 4 We present the structure of the proposed agent. We look at the fundamental flow from input to output, and how the agent performs self-addition. We also present a comparison between this agent and two other key works, sufficiently similar to require individual attention. These comparisons are presented later than the other background reading as they require an overview of the architecture for a thorough assessment.

Chapter 5 We assess the agent, by performing a series of tasks, each of which tests an aspect of the agent's desired capabilities. The first test evaluates the architecture's ability to value components, based on how much they contribute towards the agent's predictive abilities. The second test evaluates the agent's meta-learning capabilities, by using a standard machine learning library on a publicly available data set. The third test evaluates the agent's ability to perform in a Reinforcement Learning environment. The fourth test the evaluates the agent's ability to meta-learn within an RL environment. The fifth test then tests the agent's ability to subdivide the environment between components.

Chapter 6 We present a brief exploration of possible future work building on this architecture, focusing on the possibility for self-motivated learning and information acquisition. Although preliminary testing indicates a high potential for future results, additional work is needed for confirmation and full assessment.

Chapter 7 We present a brief conclusion and summary of the work as a whole.

2 Background

The proposed system fits into two major fields of machine learning research, Reinforcement Learning and Meta-learning.

Reinforcement Learning, RL defines the problems the system is designed to approach and solve, as well as providing the algorithms against which its performance can be measured. The field concerns attempting to generate a 'policy', a state-to-action mapping allowing the agent to know which action to perform in any given state in order to maximise a reward function. This reward function, as well as the actions the RL agent can perform and the nature of the information available to it, are provided by the agent's designer. A key difference between RL and other machine learning problems is that of 'credit assignment'. When an agent received a high reward, or a negative one, it must determine which recently taken actions are responsible for this outcome, and which were irrelevant.

Meta-Learning defines the approach the agent uses to solve these problems, as well defining which metrics it is intended to compete against other RL algorithms on. Meta-learning is the field of 'learning how to learn', approaching problems with knowledge gained from previously seen and learnt problems. A classic view of meta-learning is the two layered architecture, in which a high level algorithm learns to guide the learning of a lower one, in order to maximise the lower one's performance.

Of these two fields, it is fair to describe Reinforcement Learning as the older and broader field. This work is not attempting to compete against the current leading RL algorithms by most measurement metrics, but rather to bring approaches from the smaller field of Meta-Learning, in order to produce useful results in specific circumstances.

This crossover is thought to be interesting as a greater majority of Meta-Learning research has been into the machine learning field of direct supervised learning, with far fewer papers addressing ML in RL problems, as can be seen from the overview papers [1] [12] [29] [30]. The architecture of the proposed system is also, we believe, sufficiently novel to both fields to be worth studying.

2.1 Meta-Learning Approaches

2.1.1 Introduction

When presented with a supervised machine learning problem, either classification or regression, a vast number of possible solutions exist. Many algorithms exist for both these

tasks, and each algorithm requires a set of parameters to be supplied, to guide its performance. In many cases, the variables might also be selected from a larger pool of possible inputs for the learner, or the designer might have to choose which pieces of information will be useful when designing a physical sensor.

These decisions are collectively referred to as the bias [4]. The parameters and precise implementation details of a machine learning algorithm are its bias, while an agent's bias would be the set of all previously mentioned designer-decisions. While this bias is commonly introduced by the human user of the algorithm, meta-learning is the study of selecting this bias in an automatic fashion [4].

Bias is the portion of the space of all possible solutions which the machine learner will explore to find the solution to the problem it has been presented with and/or an ordering of the subspace, altering the priority placed on exploring certain solutions. If this bias encompasses the correct solution the learner is advantaged by being able to ignore incorrect solutions, or by being directed towards the correct one faster. This term 'bias' should not be confused with statistical bias, which is a systematic error in a predictor. It is instead a term for the restriction strategy a learner uses to exclude a portion of its solution space.

Such an exclusion is firstly useful in that it either reduces the number of solutions the learner must explore before it finds the correct one or moves it to explore the correct one earlier, but is also useful if the agent has insufficient training cases to decide between two or more possible solutions, only one of which is actually valid. If the bias space does not include the flawed ones, or marks them as lower priority, the learner will select the correct one.

By learning which biases were productive in previous tasks, and by correlating various sources of information about the data-set with patterns of success and failure in various biases, a machine learning algorithm attempts to select an appropriate bias for the task at hand. This bias is then passed to one or more lower-level algorithms, which attack the problem directly. Their success can then be used as another training case for the higher-level meta-learning algorithm.

As an aside, it is possibly worth noting that bias has not been escaped. Neither a human engineer or a meta-learning algorithm are free from biases or assumptions. They simply use knowledge gathered from previously faced problems to increase the amount of information available to solve the present problem.

A Bayesian analysis of data is a possible exception to this, in that it could be made to use all data points equally, and makes no pre-assumptions about the probabilities involved or their inter-dependencies. Selecting the variables to present to the Bayesian learner, however, would still represent a designer-introduced bias, as would electing to use a Bayesian approach as opposed to other machine learning strategies.

2.1.2 Portfolio

A simple, yet highly effective approach to meta-learning is to provide the agent with a library of common machine learning algorithms. A higher level, but not necessarily particularly complex, machine learner then learns which algorithms work best on which problems, based on a designer-provided fitness metric. Such metrics could be the usual measures of ML success, such as accuracy or speed, possibly further biased by a designer-added metric for understandability of the results produced. The higher level learner is termed the 'meta-learner', and the algorithm of which it is attempting to maximise the effectiveness is termed the 'base learner'.

Effectiveness, in this case, would be defined as the ability of the meta-learning layer to select the correct algorithm from the standard machine learning algorithm pool in a way which maximises this designer-chosen metric. To evaluate this effectiveness, all possible solutions would need to be evaluated, to compare the meta-learner's performance against the theoretical optimal selection. An example of this type of assessment is Santos et al's paper [20] which compares a meta-learner with a portfolio of 40 algorithms against a theoretical optimal.

Examples of algorithm portfolio use in general are Guo et al's [10]'s work on algorithm selection for NP hard problems, Salama et al's [19] work on the use of neural networks as meta-learners or Carvalho et al's [6] work on meta-learning decision tree construction biases for small-disjunct leaf generation.

What is done with the output of the top-level meta-learner varies. In some implementations the algorithm predicted to achieve the best results based on this metric is immediately run on the data. This approach produces a meta-learner which can be applied as if it were a normal machine learner, taking a training set and learning to return an output, be it a classification, a Reinforcement Learning policy or a regression curve. Examples are Gagliolo and Schmidhuber's work on an online portfolio learning algorithm for multi-armed bandit problems [9], or Xu's work on 'SATzilla' [32], a portfolio approach to the problem of satisfiability.

The only externally visible functional difference between a portfolio meta-learner setup and a non-meta-learning one is that it has used previous training sets to improve its capabilities. This functional similarity allows meta-learning algorithms to be used in situations which previously used standard algorithms without requiring major structural rewrites to the surrounding code.

Another, similar, approach is to run all the algorithms in the library, then combine their results with a weighting based on their expected usefulness for this particular task, as predicted by the higher-level algorithm. Alternatively, this ranking could be returned to the user, allowing them to make an informed decision about which algorithm to implement [5].

This allows user knowledge as well as meta-learnt knowledge to be combined into algorithm selection.

2.1.3 Parameter Learning

Another, somewhat similar approach is to have a single high-level meta-learning algorithm and a single low-level task-learning algorithm. The meta-learning layer here would bias the task learner by altering the parameters it takes initially. A good example of this would selecting the width of the kernel in a Support Vector Machine, as seen in Soares et al's work [24], although their approach does include aspects of ranking meta-learning, as described later in this section.

In an abstract sense this is very similar to the portfolio approach. The high-level learner has a bias space to select from, and chooses one which maximises the expected utility of a user-provided success metric. The principal difference being the granularity and range of the bias space. A portfolio approach may have a wide range of possible biases, if the lower-level ML algorithms are varied in their capabilities, while a parameter-setting meta-learner would have a smaller space, as it can only act within the constraints of the singular ML algorithm it has been provided with. However, the parameter setting has a far greater precision available to it, if the parameters are continuous values, and is more able to fine-tune its approach than a portfolio-selecting meta-learner can.

Portfolio selection therefore has a wider range across the space of all possible biases, but the parameter setting approach has an infinite number of points it may choose within this space. This implies that neither is necessarily superior, but should either be selected in response to the needs of the problem, or combined into a single architecture.

2.1.4 Ensemble methods: Combination of classifiers

A different approach to meta-learning is to combine multiple machine learners into a single learning algorithm. Various strategies exist to accomplish this goal, with their own strengths and weaknesses.

Boosting and Bagging Bagging and boosting are machine learning processes which revolve around splitting a single data-set into multiple subsets by subsampling. This is done either randomly in the case of bagging, or weighted to enhance learning, for boosting. Multiple instances of the same machine learning classifier are training on distinct subsets of the training data. These classifiers are then combined by a process of voting. Newly presented instances are voted on, with each classifier returning the class it believes the instance belongs to. The majority vote is then returned as the overall classification [21] [12].

The weighting used by boosting is to test the performance as new classifiers are generated, and preferentially select instances which the existing classifiers mis-classify. The

training set these new classifiers would be trained on are the parts of the overall training set which the existing ensemble of classifiers would otherwise fail to handle correctly.

This thesis does not consider bagging as a meta-learning algorithm for two major reasons. The first is that it only receives a single training set, so cannot learn to improve its learning on new problems based on experience of previously seen ones. The second is that no "learning about learning" occurs, the crude definition of meta-learning. There exists no higher level classifier which receives feedback from the lower level ones, and so the classification method cannot learn to select a bias its learning.

Boosting, on the other hand, features partial meta-learning. While it still only handles a single problem at a time, it does dynamically select a bias based on feedback from lower-level learning. It compares its existing classifier set against the training data, evaluates its performance, and adjusts how new classifiers are trained. As such, boosting could be considered a basic form of meta-learning, albeit one which cannot transfer this knowledge outside of the task at hand.

Bucket Brigade One of the earliest applications of ensemble methods is the so called Bucket Brigade classifier approach [11], in which each layer possesses only a single algorithm, with each layer hopefully gaining accuracy by using the results of the previous layer. Another architecture is to have a wide first layer of machine learning algorithms which run in parallel, with a second layer of a single learner which takes as inputs the concatenated outputs of this entire first layer. Applications of this can be seen in Chan and Stolfo's work [7].

The key difference between this approach and boosting and bagging is the use of a machine learner to read from the outputs of other learners. In boosting and bagging a simple voting procedure is used, while bucket brigades use the outputs from the first layer of machine learners, their attempts at classification, as inputs into another.

Stacking Stacking is named from its architecture consisting of multiple layers of algorithms, stacked on top of each other, with each layer running in serial, and using the outputs of the previous layer as feature inputs. Each layer is attempting to perform the same classification task, but is able to use the outputs of the previous layer's machine learners as well as the input features, thus improving its performance. The highest layer is then used as the overall classification.

While stacking is sometimes considered to be a meta-learning algorithm [29] it differs greatly from Baxter's [4] description of meta-learning as picking a bias for a learner. It has been argued [1] on these grounds that it should not be considered a part of meta-learning proper, but merely a related field of study. Those who consider it a part of meta-learning do so because it incorporates learning about learning, from which the term is derived. Ultimately, however, each machine learner is independent of any other, simply taking the

outputs of other learners as if they were environmentally-provided features in a standard input vector.

It is also worth noting that the bias for the machine learners in this approach, regardless of the layer they are a part of, must be user-provided. As a result, the solution space must be narrowed by the designer rather than a higher-level algorithm, requiring designer knowledge of the problems the algorithm will face.

Cascading As investigated by Michelson el at [14], cascading is a form of stacking which repeatedly subdivides the data based on sub-classes, passing only a given class to the next machine learning layer. To take an example, a cascading learner trained to recognise cats might subdivide between animals and non-animals, mammals and non-mammals, the finally cats and non-cats. The top layer learner attempts to distinguish between animals and non-animals, allowing the second layer to only focus on learning the distinction between mammals and non-mammals. This increased specialisation improves the accuracy by reducing the problem's scope at each stage. It requires a set of training data which is annotated with the full class taxonomy, however, as opposed to the simpler class structures of conventional classification algorithms.

An alternative approach to cascading, as used by the Viola Jones classifier [31], is to arrange a set of classifiers into an ordered list. Each element of the list attempts to solve the same classification test, in this example that of face-detection. Each classifier can return false, which would then be used as the overall response, or allow the next classifier in the list to run. Early classifiers are crude, very rapid, and highly prone to false positives, while avoiding high false-negatives. This architecture is designed for speed, to quickly return false on obvious instances, removing the need to run all later classifiers.

Meta-Decision Trees A more complex approach, used by [28] and [8], is the meta-decision tree, which employs a decision tree structure to combine a set of base-level classifiers into a single, higher performance one. A set of classifiers are trained on the training data, or each on subset of the training data. A second algorithm then learns to take their results, concatenated into a single feature vector, possibly with meta-outputs such as confidence, and attempts to learn which of its base level classifiers would be most likely to return the correct class.

This method is essentially simply a different means of combining multiple results into one, but relies on finding a single high-performance algorithm for each case, rather than performing a weighted average across all algorithms. While it has aspects of the portfolio approach, in that multiple different architectures of base level classifier could be chosen, from which the meta-learner would return only one, it differs in that all base algorithms have been trained on the given problem, as in ensemble techniques. The portfolio approach differs in that it selects an algorithm which will be applied to the entire current problem, rather than a single instance.

The main difference between the portfolio selection approach and the meta-decision tree, in terms of results, is the number of learners created. A meta-decision tree will have a number of algorithms available to it at run-time, when the learner faces a non-training task, allowing it a greater maximum theoretical performance. Conversely the portfolio approach is faster to train and requires lesser computational resources once trained, as once trained only a single algorithm, the one chosen by the meta-learner, needs to run.

2.1.5 Input features available to preceding approaches

While it is possible for stacking approaches to meta-learning to simply use the output of machine learners in an earlier layer as the inputs for the meta-learning algorithm, designers of portfolio and parameter selection approaches must provide input features to the meta-learning layer in order for it to achieve useful results.

One approach, used by [5] is to provide the meta-learning with designer-selected statistical characteristics of the dataset. Such properties could be the size of the dataset, the standard deviation of the input features or various measures of information content. This allows very processed information to be fed into the meta-learner, allowing a far more structured understanding of the data-set in question than might otherwise be possible, but requires the designer to know which measures would be useful.

Another approach useful in portfolio algorithms is to run a subset of all available learners on a subset of the full data set, to form a first impression of their relative performance. Various output statistics, such as accuracy, time taken and standard deviation of error can be taken from each learning algorithm tested, and formed into a feature vector which can be fed into the meta-learning algorithm.

This approach is known as 'landmarking', and has been shown [17] to have an acceptable success rate, and reduces the needed designer-knowledge of the problem. It is interestingly similar to stacking, in that it relies on a machine learning algorithm taking the outputs of certain learners to produce a response better than any of their individual capabilities.

2.1.6 Meta-Cognition

Meta-cognition concerns agents able to learn about their own learning, planning and reasoning. It could be considered less a field of research and more a broad umbrella term able to describe a wide range of algorithms, due to the breadth of approaches the term can cover and the range of problems it can be applied to.

A good example of this is the Meta-Cognitive Loop agent [2], which is able to recognise situations in which previously learnt rules are no longer applicable and begin learning anew. This is not a "pure" meta-cognition agent as it relies on an already-studied Reinforcement Learning approach and acts within an RL domain, and instead uses meta-cognition to augment its capabilities.

Specifically, its meta-cognitive capabilities are its ability to partially predict the usefulness of its already-gained knowledge. It develops a behavioural policy, as is common to many RL agents, which it uses for action within the environment, but is able to discard this policy when it becomes a liability rather than an asset. In [2] this occurs through a designer-written function which wipes the policy when it fails a certain number of times, but it is conceivable that the subsystem could be expanded into a meta-cognitive learning function which attempts to predict future errors from recent errors, and either wipe the learnt policy or blend it with a newly learnt one.

"Pure" meta-cognitive agents exist in a theoretical state, defined here as those which use meta-cognition as their primary mechanism, rather than add a meta-cognitive layer to a conventional learning algorithm. An example would be the Goedel machine [22], a fully self-referential learner which is able to alter any part of its internal structure to improve its capabilities. Such a learner would be fully meta-cognitive, as it would be aware of every aspect of its internal functioning. However, at the time of writing this approach has yet to be implemented.

2.2 Proposed Agent in Above Context

The agent proposed by this paper aims to incorporate features of parameter selection, algorithm portfolio learning and algorithm stacking.

The agent operates by forming a set of base learners, termed components, each of which attempts to learn to predict the results of the actions the agent takes in a subset of the environmental states the agent encounters. This results in different components handling different parts of the environment, which may well represent different tasks, in terms of the information the components need to gather and how this information is processed.

What is meant by "different parts of the environment" can change based on agent architecture and which components are selected. Give different component architectures with different subselection strategies the agent can learn which subdivision strategy leads to optimal results, in terms of predictive accuracy. Examples might be geographic subdivision, subdivision based on objects present in visual field, subdivision based on recent reward or subdivision based on other component's predictions.

The agent creates new components over time, as well as removing those deemed less useful, based on the 'worth' metric. Meta-learning occurs by the agent attempting to maximise the worth of the components as it adds them, by taking learnt actions to select the new components' properties. It then receives feedback after the component has been evaluated, following a large number of time-steps (ex 100 000), and so can trial-and-error learn to create high-worth components.

When creating the individual components the agent is able to select from a range of options. These options are designer-specified parameters which affect how the component

will operate internally, which can include both the algorithm the component can use as well as the parameters the algorithm is passed. This represents aspects from both portfolio selection, choosing the algorithm the component will use, and parameter selection. The current architecture can only allow for discrete options, leading to a lower range of choices than a continuous parameter selection meta-learner, however.

Algorithm stacking occurs by linking the outputs of components into the inputs of others. Components are executed in serial, in the order in which they were created, so any newly created component may be linked to, and so take inputs from, existing ones. This new component can then use the processing of the earlier components to aid its own. This allows sequences of components to work together, each building on previous ones, possibly combining the outputs of multiple components into one.

It would therefore seem reasonable to say that the agent proposed in this work possesses a degree of capability in all three of these types of meta-learning. It is not specialised for parameter selection, so will under-perform in these compared to dedicated meta-learners, but should be able to match algorithm portfolio learners. The performance on algorithm stacking would depend on the exact nature of the components the agent is given to work with.

3 SAMLA: Self-Additive Meta-Learning Agent

In this chapter we discuss the architecture of a self-additive meta-learning agent, shortened to SAMLA, in greater depth. We discuss how the SAMLA architecture takes inputs from its environment, processes them, and then selects an action from those available to it.

The agent operates in a series of time-steps, structured in the same way as conventional reinforcement learning agents. A time-step consists of a state presentation, an action selection and a reward. The state is a feature vector representing the external information available to the agent at that time. If the agent were an embodied robot, for example, that vector could be the data from a visual sensor. The agent then selects an action from the set of actions possible at that time-step. This action then results in a reward, which is often zero, which is fed back into the agent.

Actions are defined by the agent's designer, and provided to it when it is created. This means that while the set of available actions can change from time-step to time-step this set must always be a subset of the set of all possible designer-defined actions. The agent can encounter new situations in terms of feature vector inputs, but never new actions.

The agent operates by making predictions as to the rewards which given actions will produce, both in the short and long terms. Initially we investigate the short term operations of the agent, which can then be expanded into the long term. Short term actions occur simply by predicting the rewards for a given action in the next time-step, without considering any further consequences that action may have. As a result, short term predictions cannot be used for learning multi-step tasks.

Predictions are made by a collection of processing elements termed 'components', each of which attempts to make predictions of the reward which will be received if the agent were to take a particular action. These predictions are then used by the agent to select the action with the highest expected future reward, by picking a single prediction for each action available at the current time step. This prediction is chosen from the set of all predictions made for that action at that time-step, and is the prediction with the most accurate component. Accuracy of a component is a function of its past errors, accumulated by feedback from all the predictions it has previously made.

Not all components will make predictions during all time-steps, so the chosen component will not always be the same. Components are referred to as having a 'scope' of prediction, this scope is defined as the subset of states they will attempt to predict the outcomes of.

3.1 Structure

At time-step t the meta-learning agent is presented with a state, s_t. This state consists of a feature vector V_t and a set of actions $\{A_{t1}, A_{t2}, ...A_{tn}\}$, which are the actions available to the agent at time t. From these, the agent must select a single action, A_{tc}, which is returned to the environment, where c is the index of the chosen action. The environment then provides the agent with a reward value, R_t.

The agent contains a set of components, $\mathscr{C}$, consisting of components $\{C_0, C_1, C_2, ...\}$. A component is processing element whose goal is to generate useful predictions which the agent may use. Its internal structure is unimportant from the agent's perspective, the component simply follows a defined set of interactions, receiving inputs and returning outputs. How it arrives at these outputs is irrelevant to the agent itself.

In general, a component will consist of two functions. The first defines the 'scope' of the component, the subset of the environment space which it will attempt to make predictions for. A simple example might be a component which only operates in a given geometric area. The first function takes the provided X/Y co-ordinate, and determines whether they fall within the area of the world the component has been allocated by the agent. If it is, the component makes predictions about the effects of the agent's actions, if not, it does not, to avoid reducing its overall accuracy. More complex functions can select which predictions to make based on other inputs, including possibly inputs taken from other components' outputs.

The second function is a machine learning algorithm which will make these predictions. Both of these will use the feature vector presented to the component at each time step, while the learner will also follow the rewards the agent receives and the actions it takes, so that it can learn which input patterns combined with which actions correspond to which received rewards.

When a state is presented at time t, the agent iterates through each component in turn, presenting each with the feature vector, V_t. Each component C_i has its own output vector O_i, which it may update at this point. Output vectors vary based on component type. They must have an output, for consistency's sake, but it may be zero-length. Their contents are designer-defined, and should ideally contain any information which is produced by the machine learner inside the component which may be of use for other components. It normally contains the last prediction the component has made, but can also contain the previous error the component has received through feedback, the contents of the component's own memory or any other information the designer deems appropriate.

Components are executed serially, based on the order they were added into the agent, so the oldest components are processed first. Any component may, when its turn in this iteration arrives, read from any outputs it has been connected to. These connections are created during creation process of the component, as part of the parameters which define the component's behaviour. As a result, the supplier components are necessarily older than the supplied component.

This linking of components allows for algorithm stacking, as the output of one component can be fed as input into another. It forms a tree graph, with information flowing from older components' output vectors into younger components' inputs.

Therefore, when presented with a feature vector a component has access to two sources of information external to itself. It has access to the feature vector V_t, but also may be connected to and reading from any number of other components' output vectors. This could allow it to take another component's prediction, perform its own processing using that prediction as a starting point, and produce a more accurate prediction, which it then submits as its own. In this way two components would work together to make a single prediction with a higher accuracy than either could achieve alone.

3.2 Example of architecture as described thus far

Figure 3.1 represents a basic agent, consisting of three components. Black circles are the outputs, O_i, of the components. Component 1 has defined an output array of length 2, Component 2 has an output of length 1, and Component 3 has an output of length 0. Line arrows represent sources of information used or produced by the components. All components are using the feature vector which the agent receives from the environment, while component 3 is also using an input defined by Component 1.

Arrows leading to the prediction set represent the predictions made, which occurs in a fashion described later. Both components 1 and 2 are making predictions as to the expected value of the currently investigated action for the given feature-vector. Component 3 is not, nor is it providing any information to any other components, and as such is useless to the agent during this particular action presentation on this particular time-step.

This pass only consists of a single action from a single time-step, however. Which components make predictions can vary from time-step to time-step, but connections between components remain fixed. This means that it is possible that component 3 is simply a specialist which only rarely makes predictions. However, it may be that it is entirely useless, and neither predicts nor assists other components. If so, component 3 would be a candidate for deletion, as explained below.

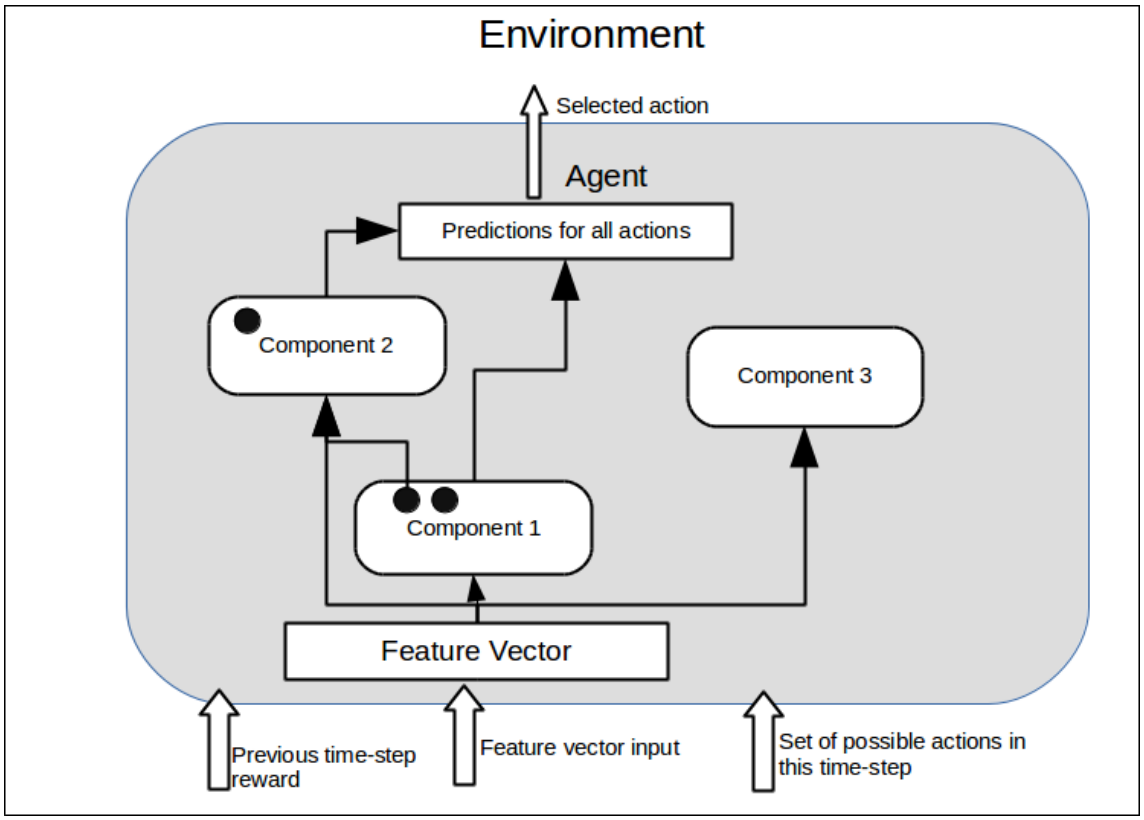

Figure 3.1: A sample agent's internal structure. The agent is receiving information from environmental sources (bottom), processing it using inter-connected components, assembling a prediction set, then returning an action to the environment based on this set.

3.3 Component prediction generation process

For each action available to the agent at time step t each component C_j may elect to make a prediction of the reward the agent would receive if it took action A_{ti}. This prediction is written $\hat{R}_{tij}$, where t is the time-step, i is the action's index and j is the index of the component making the prediction. Therefore it is the predicted value of the reward signal which the agent will receive from the environment between time steps t and $t+1$ if it selects action A_i in time step t.

Each action is iterated through separately, so each component is run once per action available at time-step t. This allows each component to produce a new output vector for each action. A good example of an output might be the prediction that component has made for that action. This would allow another component, run later in the component sequence, to base its prediction on that of the previous one, and attempt to improve its predictive accuracy by combining its own machine learning algorithm's results with that of its supplier component. Multiple passes, one for each possible action, easily enable this, with the supplier component outputting its prediction for the currently presented action, allowing other components to read directly from its output without need for internal memory.

After an iteration through all components for A_{ti} the parent agent will therefore have been presented with a set, possibly empty, of predictions for the expected value of A_{ti}, each in the form $\hat{R}_{tij}$. From these, the agent must form a single value prediction, $\hat{R}_{ti}$, which it associates with action A_{ti}. It goes through each action, assigning to each a single prediction taken from the set of all predictions made by the components for that particular action. The ultimate goal is to arrive at a set of such predictions, $\{\hat{R}_{t0}, \hat{R}_{t1}, \hat{R}_{t2}...\}$, for every action index 0 to $n-1$. t in this case would be the time-step's index, the second value would be the index of the action. n is the number of actions available at this time-step.

Once each action available at this time-step has been assigned an expected value the agent can select between them. While the valuations were produced by the components, their role in the decision making process is now complete. The agent must then select from a set of options, each with an expected reward. This is a classic RL problem, as the agent must attempt to balance reward maximisation with exploration. Many strategies could be implemented without altering the overall architecture. In this case, roulette-wheel selection is used, in which each action is given a probability equal to its proportion of the overall sum adjusted expected reward. The alteration is to add a constant k to the values such that the lowest expected reward is equal to zero. The maximum would therefore be adjusted to $maximum - k$. This allows the selection process to handle negative expected rewards.

The probability for a given action being selected, in the current implementation of exploration-vs-exploitation, is therefore:

$$p(select(A_{ti})) = (\hat{R}_{ti} - k) / \sum_{q=0}^{n-1} (\hat{R}_{tq} - k)$$

3.4 Component Error Comparison

Each component is assigned an 'error metric' by the agent, which the agent uses to compare it against other components when selecting a prediction to use for a given action. This error metric is a function of three values, the component's average error when predicting reward, denoted by L, a 'baseline error', denoted by B, and the component's index in the agent's set of agents, j. This baseline prediction is the error a naive predictor would make if it had the same scope as the component, and is described below.

A component C_j may make a single prediction, $\hat{R}_{ti}$, for every action i at every time-step t, following the presentation of state s_t. Of the actions available at t, only one action may be selected by the agent, and only predictions made for that selected action are retained by the agent, the rest are discarded. This prediction can be compared against the received reward, R_t, to produce the error for that individual prediction, L_t, by taking the square of the difference, $L_t = (\hat{R}_{ti} - R_t)^2$. The average error L of a component is the average of all the individual errors for each prediction the component has ever made. The number of these will be equal or less than the number of time-steps in which the component had the option to make a prediction, which is equal to the number of time-steps elapsed since the component was created.

Two sets of time-steps and their rewards need to be compared. The first is the component's lifetime, the set of all time-steps which have occurred since the component was created. The second is the subset of those in which the component made predictions. The first set, the lifetime time-steps, is used to establish a 'baseline error'.

The first step in computing the baseline error is to derive a naive prediction, denoted D_{tj}, where j is the index of the component and t is the time-step. This is the average reward received by the agent during each time-step in the component's lifetime. Each component therefore has a slightly different D_{tj}, unless they were created on the same time-step, in which case they will have identical D_{tj}.

Since the D value is adjusted at every time-step, as a new reward is received by the agent from the environment it must be written as D_{tj} rather than simply D_j, as D_{tj} may differ from $D_{(t+1)j}$.

The aim of a D value is to find the prediction and hence the error a 'maximally simple' predictive component would make. Such a component would do nothing but record all

incoming reward for each time-step, average them, and predict this value D for every action at every time-step. If a component cannot achieve a lower error than such a simplistic predictor it is deemed zero-worth. Each component must have its own complementary 'simplistic predictor' running in parallel so that the two of them are compared using the exact same data.

Every time a prediction is made by a component, and therefore every time its average predictive error, L, is updated, another complementary prediction is made, which is the D value of that component. The baseline error for that time-step is $B_{tj} = (D_{tj} - R_t)^2$. The component's baseline error B_t, is the average of all such complementary predictions. The error metric, denoted G_t, of a component is defined by $G_t = L_t/B_t$, or 1 if B_t is 0. This is the average predictive error of the component relative to its average naive predictive error, which it would have achieved were it to simply predict the average agent reward.

If $G_{tj} \geqslant 1$ the component can be ignored by the agent when selecting which components' predictions to use in decision making. Such a value would indicate that the component had an error equal to or higher than its naive comparison predictor, and therefore was making unhelpful predictions. If $G_{tj} < 1$ then the component is making predictions at a level which is better than chance, and thus can be used by the agent.

This error metric is used by the agent to determine which component's prediction to employ when attempting to predict the result of an action A_{ti} at time t. The agent is seeking to select a single prediction for the action's expected reward, and examines all predictions which the set of components have produced in order to select a single one.

Predictions are compared by taking the error metrics of the components which produced them, multiplying the younger (in terms of time-steps since creation) component's error metric by k, and selecting the component with the lower of the two values. k in this case is a bias value selected by the designer such that $2 > k > 1$. It causes the agent to discriminate between components of similar error-metrics by their age, preferentially selecting the older component. Older components are chosen in the case of ties.

This slight bias towards older components avoids the agent continually replacing components with new ones despite younger components not having usefully lower error rates, thus removing its ability to build on existing components. This bias was empirically determined to be necessary, as slight variations in error rates between components are inevitable, resulting in components continually being replaced by identically structured new additions. These new additions would then eventually also suffer fluctuations in their error rates, which would allow still new components to take their place. For the tests in this thesis a value of 1.1 for k is used.

Selecting between multiple components consists of a series of these one-to-one comparisons until a single component has been shown to out-perform all rivals. Since component

ages can be used to form an ordered list, the comparison function outlined above can sort the component set without inconsistency.

3.5 Valuing and Removing Components

Once a component's prediction has been selected as the predicted result for a given action, and used as the agent's prediction for the reward obtained if it were to take action A_{ti}, the component itself is paid. To avoid re-using the term 'reward', which is already used in a reinforcement learning context, the components receive 'worth', using a monetary metaphor. 'Payment', therefore, takes the place of 'rewarding', and 'worth' takes the place of 'accumulated reward'. For every time-step at most one component per possible action is paid, specifically the one making the prediction which was selected for that action, i.e. the most historically accurate one. However, since multiple actions are usually possible for any given time-step, multiple components may receive a payment.

A component's total accumulated worth, the sum total of all it has been paid, is denoted Q_j for component C_j. Its worth, however, is its rate of income, not its total accumulated payment. Its worth is denoted W_j and is Q_j/age_j

Multiple components are paid as each possible action needs to have a prediction assigned, in order to compare actions against one another. As a result, despite an untaken action not being chosen by the agent, the component's prediction was still useful to the agent, as it was used to know which actions to avoid. Therefore for each untaken action, as well as the one the agent did choose, a component must be paid. It is entirely possible, of course, that a particular component might have made several useful predictions, and so be paid multiple times in a single time-step. This results in the agent making a number of payments to components equal to the number of actions. These may be to one or more components, up to a number of components equal to the number of actions, if each action's prediction was made by a different component.

For each result predicted, the component pays out a fixed value to the components. This value is arbitrary, as all payment comparisons are relative, rather than absolute. The payment is provided by the agent to the component which was chosen as the best predictor for that result. The components then share out worth amongst themselves, by dividing it equally between themselves and any component they are receiving information from. This occurs hierarchically, with each component in turn distributing received worth down to any other components it was using. Once distributed, the worth is simply accumulated by the components, giving them a lifetime sum total monetary value Q_{tj} where t is the current time-step and j is the component's index. Its income over time is denoted by W_{tj} and is the Q_{tj} divided by the number of time-steps since the component was created.

Distribution down to a component's suppliers occurs by dividing the total worth received equally between the component which was paid and all other components it is using as

information-suppliers. These will possibly then distribute the worth they received from the higher-up component with others, again dividing the portion of the initial reward which was sent to them into equal portions and passing them down to their suppliers, while keeping one of the portions for themselves.

A component's total accumulated payment therefore, if payment is Z and it has m other components it is using information from, is described by:

$$Q_{(t+1)j} = Q_{tj} + (Z/(m+1))$$

Each of those m supplier components then receives the same amount of worth which the previous component added to its total worth, but these suppliers cannot directly add it to their own total accumulated worth. Instead they must divide it between themselves and their own suppliers equally, in the same process as above. Since no loops can exist in the supplier-supplied component-to-component links this process terminates with a set of components with no suppliers.

A component's value to the system as a whole, therefore, is determined by how useful it is in terms of predictions made, as well as how useful it was to other components, providing these in turn have a source of payment to distribute to it. It is a function of this accumulated monetary value Q_{tj} which is used to determine which components to remove, when a periodic garbage collection pass is performed on the component set.

For removing useless or unnecessary components from the agent's component pool three similar possible approaches can be used. The first is to remove the component with the lowest average income, that is to say taking the component's monetary total Q_{tj} and dividing it by the number of time-steps which have elapsed since that particular component was created. Provided there is also a minimum age before which the components cannot be removed this technique works well, and allows the system to remove components which are either inaccurate, representative of an exceedingly rare environmental feature, or simply useless. This value is denoted W_{tj}, the component's income over time.

The two other methods involve dividing the component's worth not by its age but by its computational resource use, either by clock-time used to carry out its internal computation, or its clock time plus its maximum-ever memory use. Their relative worths would need to be defined by the designer, as well as the thresholds above which the agent starts removing components. While not employed in this thesis, such component-removal strategies could be useful for creating a maximum-performance agent, in terms of real-time decision making on finite hardware. It could certainly be interesting as a further study to see how an agent attempts to learn an environment if its most accurate and useful algorithms were computationally expensive and their use needed to be limited to only the most difficult parts of the problem.

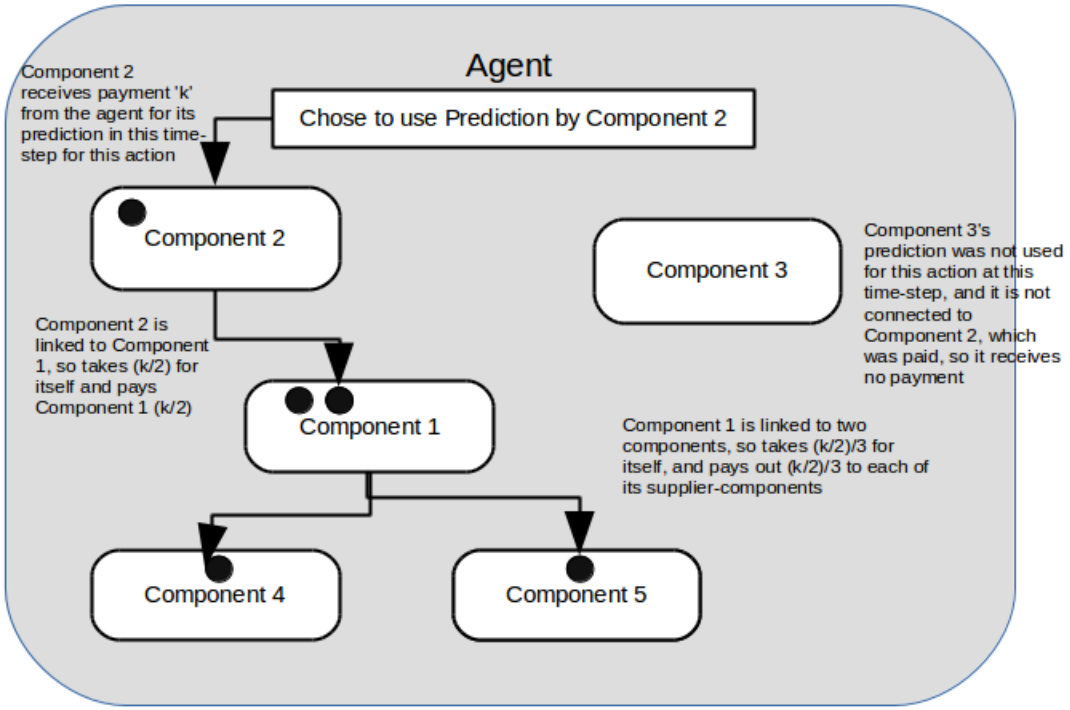

Figure 3.2: A sample agent paying a component for its prediction. This is for a single action in the given time-step, multiple actions can exist, so this sequence can occur multiple times during the same time-step. The component paid by the agent for other actions may be the same or may be different, but the internal links between components remain the same. In this example, component 2 has been paid a reward denoted k for its prediction. It has one supplier component, so must share it between the two of them, each therefore receiving $(k/2)$. Component 1, however, has two suppliers of its own, so takes this $k/2$ and divides it between the three of them, each therefore taking $(k/2)/3$ payment. They add this to their total worth, and their worth to the agent is that total divided by the number of time-steps they have been in existence, that is to say their income over time.

3.6 Component Addition and Meta-Learning

Meta-learning occurs by learning which components to add to the pool. This learning is, in turn, accomplished by components. At designer specified times (usually either following a high error or at fixed time periods) the agent is prompted to create a new component. The designer will have specified a series of archetypes from which it can choose, which may also have a set of parameters to set.

These are a set of actions, from the agent's perspective. It is presented with a set of options in the form of actions, and must take them, similar to how it takes other actions. Meta-learning was designed to be as similar to ordinary Reinforcement Learning as possible, to simplify architecture and reduce conflicting design goals.

When creating a component, the first action the agent must take is to select a configuration for the new component, followed by the parameters this configuration requires for implementation. A common configuration is for the component archetype to be the machine learning algorithm it contains, and its parameter to be its scope of prediction. The agent therefore must select a type of learner and specify how specialised it wishes it to be. The utility function in this case, which the agent attempts to maximise, is the future worth of this new component. This is a quantifiable, learnable, signal.

The reward function for any action taken during meta-learning is simply the total accumulated worth of the created component, Q_{tj}, after a fixed number of time-steps from that component's creation. The agent is therefore trying to predict the total reward the component will receive if it is created in a given configuration, so it can select the configuration which maximises total accumulated worth.

For every option available in the creation of the component, say every type of machine learning algorithm available, predictions are made as to how much worth a component using that setup would eventually accumulate. One of these options will be selected, and the relevant predictions stored alongside the newly created component. After a designer-specified number of time-steps (usually in the order of magnitude of a few million) the component's total worth is returned, and these predictions' accuracies can be evaluated.

Components specialising in making predictions concerning future worths of other components are the source of the meta-learning in the agent. They make predictions about how useful new components would be if given certain parameters, then wait for feedback from the agent. This is a slow process, but allows these components to determine which types of parameters (machine learner or scope for example) result in high-worth components, and which result in low-worth components which are removed.

These components are termed meta-learning components. Due to the slow nature of their prediction-feedback (they must wait for other components to be fully evaluated before

receiving feedback) they must be handled slightly differently from normal components. They still receive worth for making correct predictions, still compete mutually, and still take outputs from other components. However, they are shielded from removal for a long grace period, to allow their long prediction cycle to finish. As a result, they must only be added very rarely to the component pool, to prevent it from being flooded with meta-learning components.

This results in a somewhat distinct class of components, which are added and removed at a slower rate. In other respects, however, they operate normally. This means that they themselves are subject to replacement by superior competitors. Newer meta-learning components will be able to use the outputs of normal components to improve their accuracies, and will compete to find the best for each prediction type (one component may be best at predicting the worth of machine learning algorithms, while another might be best at selecting the scope).

Meta-learning, therefore, is in learning to predict which types of new component will be useful. This is learnt by the meta-learning components. This mechanism allows the agent to create better components for future tasks, improving its performance on those tasks.

Meta-meta-learning, however, is very slowly learnt by competition between the meta-learning components. This mechanism has no special attention given to it, it is simply hoped that these meta-learning components will be able to build on the information generated by non-meta-learning components, the 'normal' components. If they cannot, the agent remains entirely functional without meta-meta-learning capabilities. Infinite recursion of meta-learning may be interesting, but it is not the focus of this thesis.

3.6.1 Pseudocode: Agent's main decision loop

Algorithm 1 represents the basis for how the agent selects actions from those presented to it. The aim is to arrive at a prediction for the expected reward of each action available to it at the current time-step. It can then select a high-expected-value action, and as such maximise its lifetime total obtained reward. A degree of randomness is introduced in order to allow exploration, in a classic RL exploration-vs-exploitation design.

The fundamental process is to look at each available action in turn and present it to the agent's set of components (lines 2 to 8). A subset of the components will then make a prediction. The agent will gather these into a set (line 9), containing all predictions for this particular action, set $\mathscr{P}$. From this gathered set of predictions the agent then selects a single prediction (lines 13 to 31). It selects this by looking at the components' past errors, and selecting the one with the lowest previous error. This "best prediction" is then used as the agent's own prediction as to the reward it would achieve in the next time frame if it were to select the action A_{ti}. Once every A_{ti} in s_t has been assigned an expected value by the agent, it then selects an action based on these expectations.

Algorithm 1 Agent's main decision function, taking a state and returning an action.

Inputs

s_t State provided by environment, container for all other inputs as well as actions
V_{ti} Feature vector
$R_{(t-1)}$ Reward from previous time-step

Steps

$\mathscr{R} = \{\}$ ▷ New empty ordered array for predicted results of all actions in s_t
for $i = 0 \rightarrow number\_of\_actions$ **do**
    $\mathscr{P} = \{\}$ ▷ New empty list for predicted result for action A_{ti}
    for $j = 0 \rightarrow number\_of\_components$ **do**
        $C_j.takeData(V_{ti})$
        $C_j.takeDataFromOtherComponents()$
        if $C_j.currentFeatureIsInScope()$ **then**
            $P = C_j.generatePrediction()$
            $\mathscr{P} = \mathscr{P} \cup P$
        end if
    end for
    ▷ Predictions for action i have been made, now a single one must be placed into $\mathscr{R}$ at index i
    if $\mathscr{P}.isEmpty()$ **then**
        $\mathscr{R}[i] = 0$
        $continueLoop$
    else
        $bestPrediction = \mathscr{P}[0]$ ▷ Set default to be first in list
        for $j = 0 \rightarrow number\_of\_predictions\_made$ **do**
            ▷ //he1 is the error of the component which made the current-best prediction
            $he1 = bestPrediction.creatorComponent.historicError$
            ▷ //he2 is the error of the currently considered prediction's creator
            $he2 = \mathscr{P}[j].creatorComponent.historicError$
            $he2 = he2 \times k$
            ▷ //The second prediction must be from a younger component, because the component list is ordered by age. As such, he2 must be penalised, multiplying it by the bias k
            if $he2 < he1$ **then**
                $bestPrediction = \mathscr{P}[j]$
            end if
        end for
        $\mathscr{R}[i] = bestPrediction$
    end if
end for ▷ Each action has now been assigned a predicted result
$min = minimum\_value\_in\_list(\mathscr{R})$ ▷ Set minimum to lowest predicted value (can be <0)
$probabilities = \{\}$ ▷ Empty list for all the action-selection probabilities
for $i = 0 \rightarrow number\_of\_actions$ **do**
    $\mathscr{R}[i] = \mathscr{R}[i] - min$
    $sum = sum + \mathscr{R}[i]$
end for
for $i = 0 \rightarrow number\_of\_actions$ **do**
    $probabilities[i] = (\mathscr{R}[i])/sum$
end for
$action\_index = select\_by\_probability(\mathscr{R})$
return $action\_index$

It is worth noting that while every action has a set of predictions made of its expected value only one action will be selected by the agent. Therefore only the predictions made for the chosen action can be evaluated. All other predictions must be discarded, and their creator components' errors remain unchanged. The entire set of predictions, not just the bestPrediction, but the $\mathscr{P}$ set for the chosen action, is evaluated when the reward is obtained. This means that every component which made a prediction as to the result of the chosen action has its historic error updated in the next time-step.

3.7 Similar Architectures

Two other approaches exist in the literature which are sufficiently similar to the SAMLA architecture to require in-depth comparison. This comparison occurs after a discussion of the structure of the SAMLA architecture to allow technical details to be compared.

The two approaches are the AERA/EXPAI agent architecture and the Hayek machine. Both are approaches which revolve around an agent composed of multiple sub-agents, which are added and removed based on usefulness. The EXPAI architecture, of which AERA is an implementation, is more structurally similar to the SAMLA architecture of this thesis, but the Hayek machine served as an initial inspiration for SAMLA.

3.7.1 Self-modifying AERA and EXPAI frameworks

AERA [15,25,27], and the broader category of architectures EXPAI [26], of which AERA is an example, are self-modifying frameworks which use mechanisms similar to this thesis's SAMLA architecture.

In the AERA family of architectures an agent attempts to work towards a drive, roughly similar to a reinforcement learning agent's utility function, based on data received from the environment. It accomplishes this by subdividing the task between multiple **models**, termed **granules** in the EXPAI literature, similar to this SAMLA's **components**. To reduce ambiguity, the term 'granule' will be used for these elements.

These granules are similar to components in that they take inputs and return predictions and outputs. In AERA, the granules use pattern matching, establishing a set of preconditions which must be met for a pattern to be considered "present". They can then form a prediction of a future condition. This condition may be goal-related, therefore serve a role similar to that of prediction in SAMLA. It may also be a prediction of the fulfilment of a second granule's preconditions.

Planning can therefore be performed by taking a given set of truth values (environmental or granule outputs), evaluating expected future truth values given particular actions, then iteratively progressing in this fashion. Reverse is also possible, by taking desired future conditions, then running the process backwards, investigating which conditions would need to be true a time-step $t-1$ for the desired conditions to be true at time-step t.

Granules are evaluated based on their predictive accuracy and are able to be combined hierarchically, by referring to the truth values of granules created earlier. This is similar to SAMLA, which rewards components based on predictive accuracy, and allows components to stack by building on the outputs of others.

A key aspect of the EXPAI architectural approach, compared to the AERA approach, is the self-improvement drive. Based on a formal definition of curiosity by Schmidhuber [23], the agent is motivated to create new, effective, granules. This provides a self-motivated learning drive, which allows the agent to autonomously learn in periods where no designer-provided drive is applicable. This occurs in a learnt way, allowing experience to guide the creation of new granules.

Comparison Many aspects are therefore similar between the architectures outlined in this paper and the AERA/EXPAI model. Components/granules are added to cover gaps in the predictive abilities of the agents. They are removed if found to be low accuracy, kept if shown to be useful.

The first difference revolves around the primary goal of the SAMLA architecture. As an attempt at creating a meta-learning reinforcement learner, the components are machine learning algorithms in and of themselves. The agent selects not only what to use as their target (for some component types) but also which algorithm it expects to have the best results on the problem at hand. While expansions have been proposed for the EXPAI model which would provide learning capabilities to the granules these have yet to be implemented. This represents a major difference, given the focus on meta-learning in SAMLA.

Granules in the AERA model are created by a fixed granule-creation subsystem. No meta-learning occurs, as there is no feedback nor learning structure within this subsystem. A complex system exists to infer cause and effect, and create sets of new granules to investigate many possible cause-effect links at once, but no learnt targeting occurs.

Granules in the EXPAI (but not the AERA) architecture can guide the agent towards situations in which it creates new granules, but are not linked into the granule-creation mechanism. This means these architectures are learners, but not strictly meta-learners.

The second difference is in the nature of the outputs. SAMLA's components output expected utility following actions, as well as arbitrary data from their machine learning algorithm. Such data can be their prediction, some stored data, meta-data (recent error for example) or any other numeric value. This differs from the AERA model which allows only final outputs and relevant information such as priority and confidence. By allowing partially-processed information to be output in a component-defined way the SAMLA architecture allows a wider variety of types of information to be extracted from a component for use in later processing. It also allows for a slow selection for component-types which

provide useful information, without requiring designer knowledge of which information-sources would be useful.

This difference leads to certain differences between the capabilities of the two agents. This work explores the agent's ability to navigate grids and recognise images, while the AERA/EXPAI model revolves around linguistics and planning/simulation. These are two key fields in which one architecture has an advantage over the other.

SAMLA has the advantage in image classification and similarly structured environments, as its components are larger and have greater internal power. Existing algorithms can be leveraged, allowing the agent to represent a 'messy' concept such the difference between hand written digits within a single component, rather than requiring multiple granules to encompass the difference. This learning can be performed within the component itself, reducing the demand on the component-creation mechanism, as it can take a more abstract and high-level approach to component creation.

In the AREA/EXPAI architectures, conversely, the boolean representational structure of granules makes them ideal for planning. Granule predictions are for state transitions, so the entire agent is primarily geared towards plan-based action, as opposed to standard reinforcement learning, which simply deals in utility function expectations. Evidently there are many environments in which reasoned planning is a far more effective solution than simple utility function expectation maximisation.

Memory structures are another difference. In the SAMLA architecture, the components are able to store data internally, and share it with other components (an example is the recent-error sharing task). The component worth metric allows components to exist which serve simply as short term memory blocks. EXPAI architectures have a shared global memory, recording the results of granule pattern matching, and storing active predictions and goals. This allows the SAMLA architecture to use multiple different forms of memory, such as buffers, recursive neural networks, reservoir methods or other, and discard those which prove less useful.

Future work While mostly unimplemented, the future work section of this thesis lays out certain other key differences between the architectures, in terms of which future tasks the two approaches would be suited for. Both architectures are intended for further expansion, with works on EXPAI going to length to discuss a roadmap for the future of the architecture [26].

Both architectures aim to implement an artificial curiosity function which acts as an internal drive to create new useful components/granules. In this, the EXPAI structure is superior due to its mathematical underpinnings providing a strong plan of attack. This work's curiosity drive is primarily based around the work of Oudeyer et al [16], and has yet to be tested within the SAMLA architecture.

Conversely, this work outlines and tests a knowledge representation mechanism which EXPAI lacks. By dividing the learning task between the structure of the representation (the components) and the information which is contained within that representation (the information stored by those components), the agent has a very dynamic manner of storing information.

As outlined and tested in section 5.0.2, this mechanism allows a second type of information seeking behaviour. Shorter term then component-creation, it assumes that certain properties of the environment will be accessible to the agent, but not immediately. The agent must therefore take action to acquire this information. If this property is highly dynamic, the agent may have to repeatedly go check its value before taking an action which depends upon it.

An example might be the weather. The agent may need to take a set of actions to check the weather, rather than have access to it immediately. The representation here would be that there exists a property (the weather) which affects the outcome of the agents actions. A clear structural distinction exists within the agent between two key types of knowledge. Firstly that weather exists, and how weather affects its actions. Secondly what the weather is at this current time.

The mechanism allows this second type of knowledge to be handled in a separate fashion from the first. The first is slow, and requires new components to be created, it is a permanent part of the agent, and must only be learnt once. The second must be constantly rechecked. The agent needs to learn a series of actions to check the weather, so that it can do so repeatedly. It also needs to be able to evaluate the worth of that piece of knowledge, and the reliability of the knowledge once obtained (it will be useless after a week, but still valid after ten seconds, with a learnable trustworthiness curve in between).

This work outlines and tests a clear mechanism for hypothesising the existence of such information, learning the process of information acquisition, and attributing a quantifiable worth to that information. In SAMLA, learning about the nature and reliability of information occurs within components, and can be handled competitively to find the best algorithm for the job. The meta-learner need not concern itself with the minutiae, but must only find an algorithm to delegate to, a far easier job.

This delegation of tasks is expected to greatly improve the agent's performance on such jobs, which are expected to be common in environments where the agent is mobile but has senses which only cover a small portion of the environment. The ability to seek out information is key to autonomy, and simplifying the learning process allows otherwise intractable problems to be handled with ease.

In conclusion, the difference between the architectures comes primarily from scale of components/granules and the mechanism by which they are generated. This paper has

larger granules (components), containing machine learning algorithms, and uses the same competitive-prediction-by-granule mechanism for solving its primary task as it does for creating new granules. It is therefore a meta-learner, as it has one or more learning algorithm learning to create new learning algorithms to solve a task.

The AERA/EXPAI approach revolves around far smaller granules, individually containing far less information and representing far less of the total environment. Conversely, it has far more granules, and they can be evaluated at a faster rate, and created in greater numbers. This allows more fine-tuned removal and addition. They are also specialised in planning, and operate well in tasks which require these abilities.

Both architectures could be adapted to be more similar to the other, but as they stand are very different in terms tasks they are suited for and design goals.

3.7.2 The Hayek machine

This work's initial inspiration was the Hayek machine, described in Baum's paper [3], although the two architectures differ greatly. The key feature this thesis takes from the earlier paper is the concept of a set of agents forming a super-agent, by subdividing the problem between themselves.

In the Hayek machine the components bid for access to the world. One agent wins the bidding for each time-step, and may select which actions the agent as a whole takes. If it achieves reward from the environment it may keep it, allowing it to win further bids. Components also pay others for setting up situations which they benefit from, allowing chains of components to perform actions in sequence, generating reward for the last component, which pays it back down the line.

This differs from the agent presented in this work, as it allows components to directly take actions on the agent's behalf, rather than components to act as advisors to the agent by informing it of the expected outcomes of actions. One key reason the components as predictors approach was preferred over components as action-performers was the case of negative rewards. In the Hayek machine the components have a strong incentive to bid for control over the agent when the expected reward is positive, but have no incentive to bid for control when an negative reward is inevitable. This would prevent the agent from learning to select the lesser of two negative rewards, as no component would choose to take this portion of the environment.

Another key choice in using predictions rather than providing direct links between components and actions is that of bucket-brigade style component linking. In the Hayek architecture each component exists alone, and must take the agent's inputs and produce an output, and as such the agent overall cannot outperform any component's accuracy. In this work's architecture, however, components may use other components' predictions as starting points for their own. Therefore algorithms which would, in isolation, only have marginal success can be combined into more effective structures.

This use of predictions also adjusts how the agent selects and rewards components. In Baum's work, the components receive reward from the environment at the same time the agent does, then share it between themselves. In this work, however, the components are paid by the agent without any direct link between the reward, or worth, they receive and the reward the agent receives.

In terms of component generation the agents also differ. The components in the Hayek machine are generated at random, from the space of all possible 'scopes'. These scopes, as in the work, as the regions of the problem space which a given component is responsible for. This does not allow the agent to select the composition of its own components, which the agent architecture this work describes uses to enable meta-learning. It seems possible to construct a Hayek machine which allows learnt component creation, by allowing components to bid for the task of component creation. They would then receive reward for doing so after their 'child' component had been in existence for a given number of time-steps, allowing its contributions to the agent's performance to be evaluated.

In conclusion, it can be seen that while the two architectures share similarities they are functionally very dissimilar. A key driver of this is their differing goals. The Hayek machine is designed as a pure Reinforcement Learner, while the Self-Additive Meta-Learner is designed to bring meta-learning capabilities to the field. While this work is inspired by Baum's work, it is sufficiently different to be considered its own architecture, rather than be thought of as a variant of the Hayek machine.

4 Validation Experiments

4.1 Introduction

This section tests the essential foundations of the agent, verifying that it behaves as intended, with regard to component creation and evaluation.

The desired outcome is that the agent is able to form a pool of components which are, when combined, able to predict the rewards it will receive in its environment. Amongst these, the agent must be able to subselect the set of 'useful' components, those which produce accurate non-duplicate predictions. Components which have a lower error rate must be privileged over those with higher errors, and those which make the same predictions as other, older, components must be ignored as redundant. This must be accomplished by the worth metric, which should allow a sorting of the components from most useful to least, in terms of improvement of overall predictive accuracy.

The tests in this section are designed to test the architecture's capabilities in a way which covers as much of its desired range of capabilities as possible. The goal was to produce a reinforcement learning agent out of the SAMLA architecture, which focuses on meta-learning. The first test verifies that the agent's core structure, the component addition, valuation and removal system is a valid and possible approach to self-addition. Most subsequent tests then show meta-learning capabilities, each attempting to cover a different type of meta-learning.

4.2 Component Valuation Test

This test is designed to demonstrate the ability of the system to attribute numeric worth to components in order to reach an optimal set of components. Such a set would be the one which, between them, best predicts the agent's environment. The agent must attempt to maximise its predictive accuracy using a finite number of components of limited computational capabilities. As such, worth must be assigned to order the components in terms of usefulness. Since lowest worth components are removed first, the worth must correspond to how much the agent's performance would be damaged by the component's removal, that is to say how 'useful' it is.

The task chosen to test this is a simplified version of a reinforcement learning task, adapted to remove the need for long-term credit assignment, in order to more easily explain the nature of the components the system is producing.

The environment for this test is a simple 25×25 grid, with (x,y) co-ordinates ranging from (0,0) to (24,24). Each cell has a particular value assigned to it, defined by $17 - \sqrt{(x-12)^2 + (y-12)^2}$ where x and y are the cell's (x,y) co-ordinates. This creates a gradient from the furthest corner, with a value of just above 0, to the central cell (12,12) which has a value of 17. The agent has a set of four actions, representing movement in the four cardinal directions. It must predict the value of the cell it will arrive in following any particular action. Once arriving at the (12,12) cell it is moved to a different random position on the grid. The reward received is the value it must predict, the value of the cell it *arrives* in. Hence each of the four actions available in any square have differing rewards. The goal of the agent is to maximise total reward received, which is best accomplished by repeatedly travelling to the (12,12) cell by the fastest route.

Component architecture for this task The agent's only feature input in this circumstance is its (x,y) position on the grid, thus this is the only information available to the components. This information is entirely sufficient to provide perfect predictions of future reward.

Only one component structure has been provided to the agent for this task. This component architecture is an average-value predictor which takes the (x,y) position the agent was at when the component was created and a parameter determining the scope of the component's predictive field. The initial (x,y) position, the feature vector the agent was presented with on the time-step when it created this particular component, is referred to as that component's 'formative vector'. It defines the centre of the portion of the environment the component is responsible for. The predictive field draws a circle centred on the formative position vector so as to encompass a proportion of the seen inputs equal to that demanded by the provided scope-parameter.

This is accomplished by having the component record the agent's input vector on the time-step when it was created, as well as the feature vectors from its first 1024 time-steps. It orders these by Euclidean distance from its formative feature-vector, and selects a distance which separates the $p1024$ closest samples from the rest, where p is the desired proportion. This has a risk of bias due to the sampling being only from a limited amount of the component's total lifetime (only the first 1024 time-steps) but is computationally inexpensive and produces acceptable results for this test's purposes.

When the agent's input indicates that it is within a component's chosen circle, the component then averages all seen grid-values (which are returned as reward signals to the agent) for each of the four cardinal movement actions. It then predicts these averaged values whenever it determines the agent is within 'its' part of the environment. As such, the probability that this component makes a prediction at any given time-step is p. In other cases the component simply skips that time-step, and the agent must use another component as a source of predictions.

Components can only store a single expected value for each action, internally, regardless of how many grid-squares their predictive scope covers. Since a component which encompasses more than one square on the grid will have to average its predictions across multiple squares, each with differing rewards, components lose accuracy as their predictive scope increases.

This sets up a balancing issue for the components. Their predictions will gain accuracy as their scope decreases, as a large scope will encompass too many differing values from too many parts of the grid for their prediction to be competitive against those of other components. Conversely, too small as scope will provide far too few chances at making any predictions, and components only receive worth for predictions made. This balance is controlled by the scope parameter, passed when the component is made.

The scope parameter is provided by the agent, and can therefore be meta-learnt to be optimise component creation. Other than this, no meta-learning can occur for this test. Components are created periodically, once every 1667 time-steps, rather than in response to learnt responses by the agent or to environmental cues.

When a component is created, the agent is able to assign it a proportion of the environment to cover, its 'scope'. This is selected by the agent from a set of 16 options, with the options being $1/2^{k/2}$ for k ranging from 1 to 16. This provides the agent with, if it can meta-learn this parameter, a way of creating components which will best balance quantity of prediction with quality of prediction.

Five separate runs were performed, each for 250,000 time-steps, with a component created every 1667 time-steps. This lead to the creation of 149 components, of which only 97 were kept. This is due to the presence of 3 meta-learning components, not included in this count, which counted against the 100 maximum-component cap. This cap was set unnecessarily high to allow a large number of components to be preserved for study.

Meta-learning components are, by necessity, handled differently to non-meta-learning components. While normal components make predictions about the reward received by the agent if it takes particular actions, the meta-learning components make predictions about the worth a component would have if it were created with a particular parameter set. The normal components receive information about their predictive accuracy in the next time-step, but the meta-learning components must wait till the component whose worth they were predicting has aged enough for this to be known. As such, the agent does not remove a component marked as a meta-learner until it has gone through a sufficient number of time-steps to have received enough feedback on its predictions to evaluate its predictive effectiveness. This 'sufficient number' is a designer set parameter, which for these tests was set to 16,000,000 time-steps.

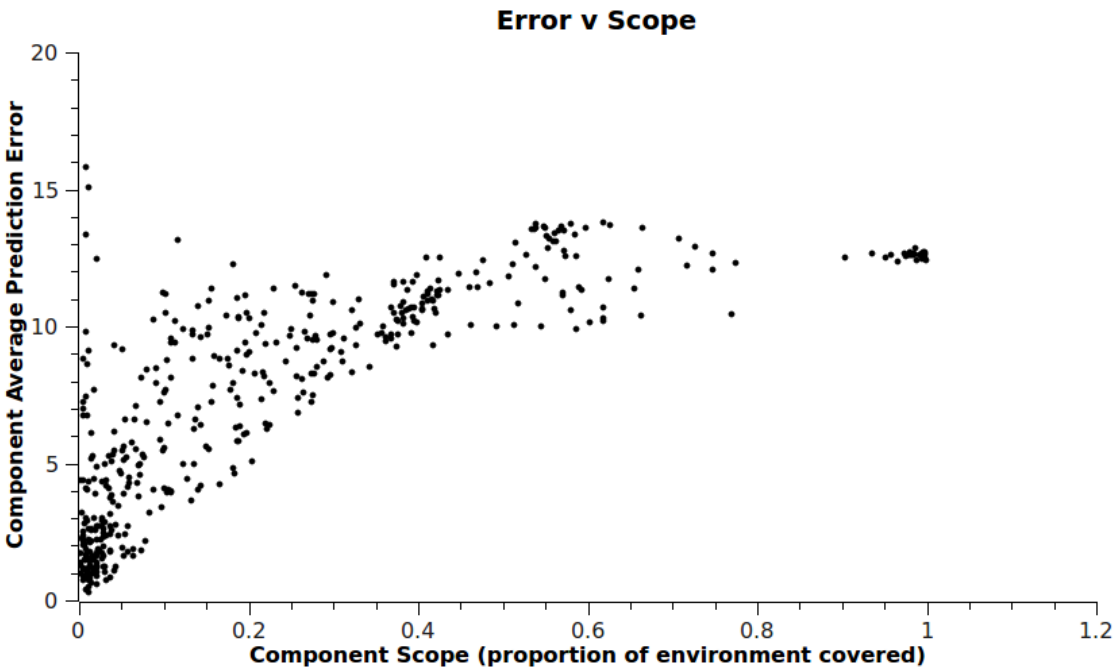

Figure 4.1: 5 agents' component error against component scope

The results from this test showed that the agent is able to correctly assess components, and is able to use generalists when they are the best choice, but replace them with specialists where available.

Component Error against Component Scope Figure 4.1 shows an analysis of 5 agents' components, comparing error and scope. It confirms what was expected, that the size of a component's chosen environmental scope, the part of the environment which it will make predictions within, is linked to its error. The larger the range of predictions a component attempts to make, the worse its accuracy at those predictions.

It is worth noting that some components failed to correctly match their actual scope, the proportion of the time-steps they made predictions in, to their scope-parameter. This resulted in them taking too much of the environment as their scope (those which are at 1.0 or near it in Figure 4.1). This is entirely expected, considering the limited information available to these components. Since components must attempt to learn for themselves the range of environmental values, then learn to subselect from these to match their provided scope-parameter, they are vulnerable to various ordinary issues relating to sampling and data sizes. The agent's architecture does not depend on the success of any individual component, so if these flaws cause the component to perform worse than it otherwise would it can simply be replaced by another.

Component Worth by Age The agent only has need of a single component per prediction, and will select older components preferentially. If two components make predictions as to the results of the same action at the same time frame, and have almost identical

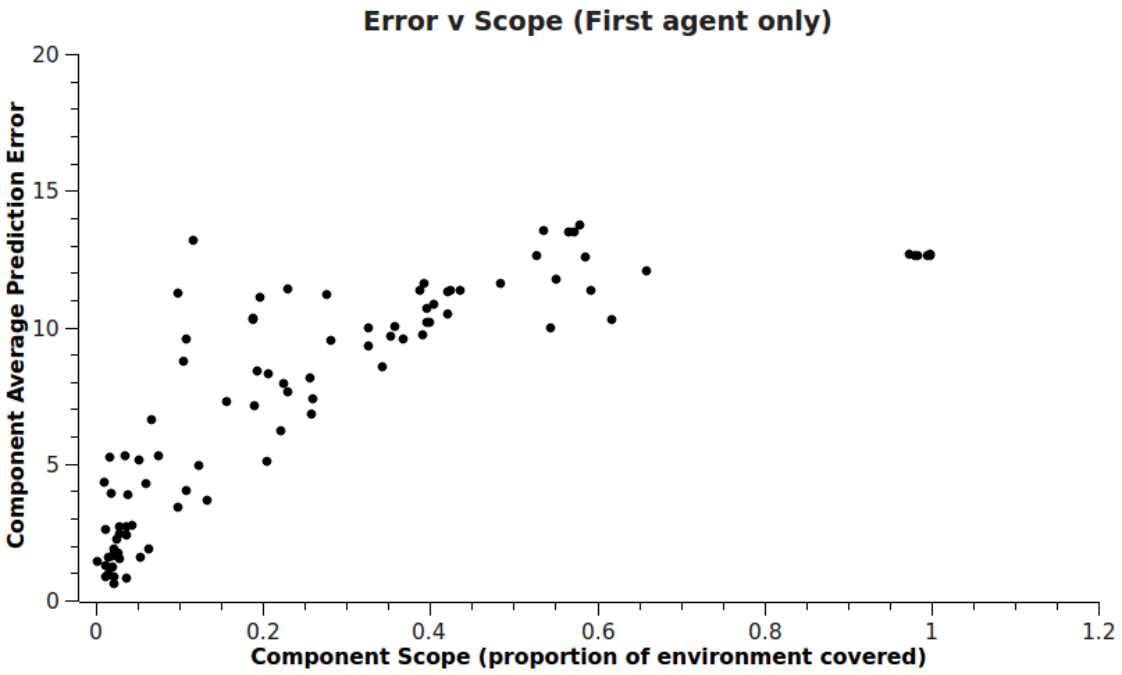

Figure 4.2: Component error against component scope, using data only from the first agent to run, demonstrating that the shape of the distribution for individual agents is similar to the combined graph 4.1

historical errors, the agent will use the older of the two. As such, components made early on in the agent's lifetime have a significantly higher chance of having high worth, as they will have 'claimed' parts of the environment, preventing any new components from gaining worth by making predictions while the agent is in those parts of the environment.

Components are considered to have an 'almost identical historical error' if their errors are such that the error of the elder component is less than a factor of k greater than the younger component, where k is a small value greater than 1. For this trial, this value has been set at 1.1. Higher values increase the improvement a new component needs to make on an older component's error to be accepted as superior, while lower values, down to 1, make it easier.

Values of 1, in which errors were compared against one another fairly, were shown to lead to excessive stabilisation times, with large numbers of components being replaced without improvement to the agent's overall predictive effectiveness. This is due to very small differences in historic errors having too large an impact on the agent's architecture. Conversely, values of 1.5 and above lead to inferior component maintaining their positions to the detriment of the agent's performance.

Figure 4.3 shows that the agent is valuing older components more, therefore preserving those it has at the expense of the new, all other variables being ignored. This is necessary for

Figure 4.3: Component age against component worth. Component ages are plotted as age/5000, for ease of reading

more complex component-component interactions, as it allows the output from older components to be used by others without risk of these older components being pointlessly removed.
It also demonstrates an ability to value components in relation to the whole set. New components which perform identical tasks to existing ones are useless to the agent, and are valued as such. Since low-worth components are removed first should the agent run out of space for new components this mechanism results in the agent not losing predictive capacity, as it is only removing redundancy.

Worth against Scope As previously discussed, a component must balance its scope of prediction, the amount of the environment which it makes predictions in, against the error it incurs from attempting to learn such a large area. These errors are competitive against the other components which have predictive scopes which overlap with its own. Only the most accurate (as determined from past predictions) component will be chosen for any particular action at any particular time-step, so to gain worth a component must be the most accurate at least some of the time.

Figure 4.4 shows the worths the components of the five agents achieved, plotted against their scope. As seen from this figure, several strategies exist for a component to gain sufficient worth to be kept. These correspond to the parameters the agent provides when the component is created, with the agent attempting to maximise the worth of its new

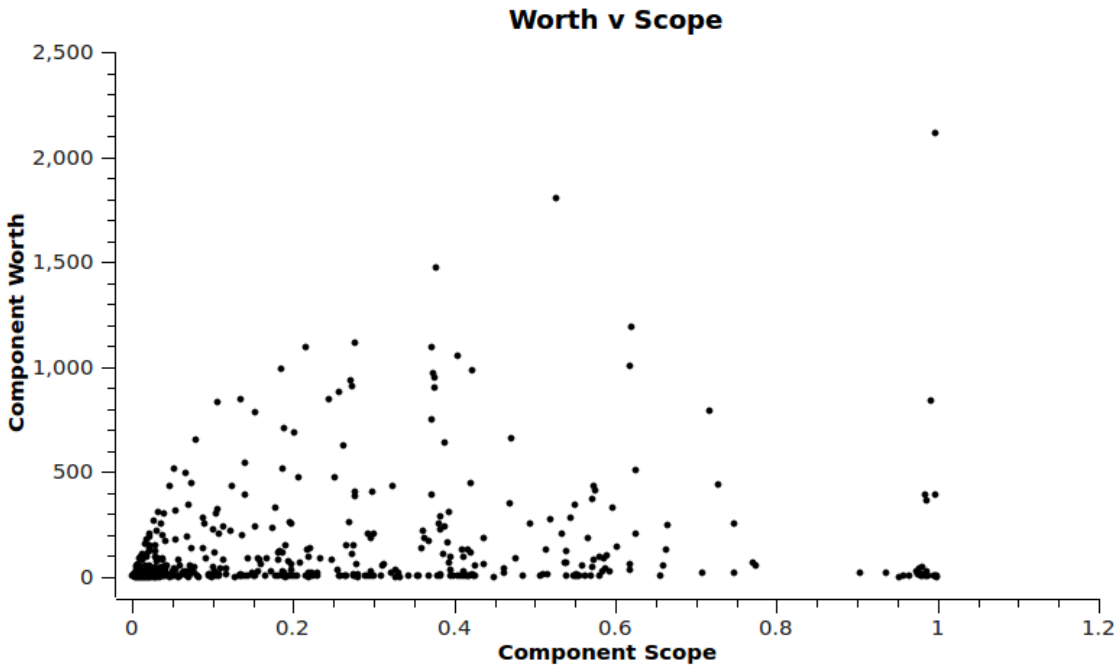

Figure 4.4: Component worth against component scope.

component.

The most reliably high-worth form of component is the moderate scope option. Those ranging between 0.1 and 0.4 in terms of proportion of the environment predicted exist in the highest density of high-worth components, and so have the highest chance of being a high-worth component. These moderates balance accuracy against scope, and so while an extreme specialist component might eclipse them occasionally, this component would necessarily only make predictions rarely. The moderate would therefore still have an acceptably large amount of the environment in which it was the most accurate, and so would accrue an ample amount of worth to be preserved. These components divide the environment into patches, with the number of patches determined by the number of components the agent is permitted to build.

The specialists (those below 0.1), on the other hand, employ a strategy of guaranteed highest-accuracy at the cost of having limited options to use this accuracy. While their predictions will always out-perform other components' predictions, the agent is rarely in the part of the environment they are able to handle, so they rarely receive worth. It can be seen that as scope decreases, so does the maximum possible reward. If a component is so specialised the agent almost never uses it it is a prime candidate for removal, so should have low worth. The agent would clearly not lose much predictive accuracy without it. These components only persist if they serve the role of hyper-specialist in a part of the environment the agent visits regularly.

On the other range of the scale are the extremely high scope components (> 0.4). These compete with one-another almost all the time, as their scopes overlap. As such, only very

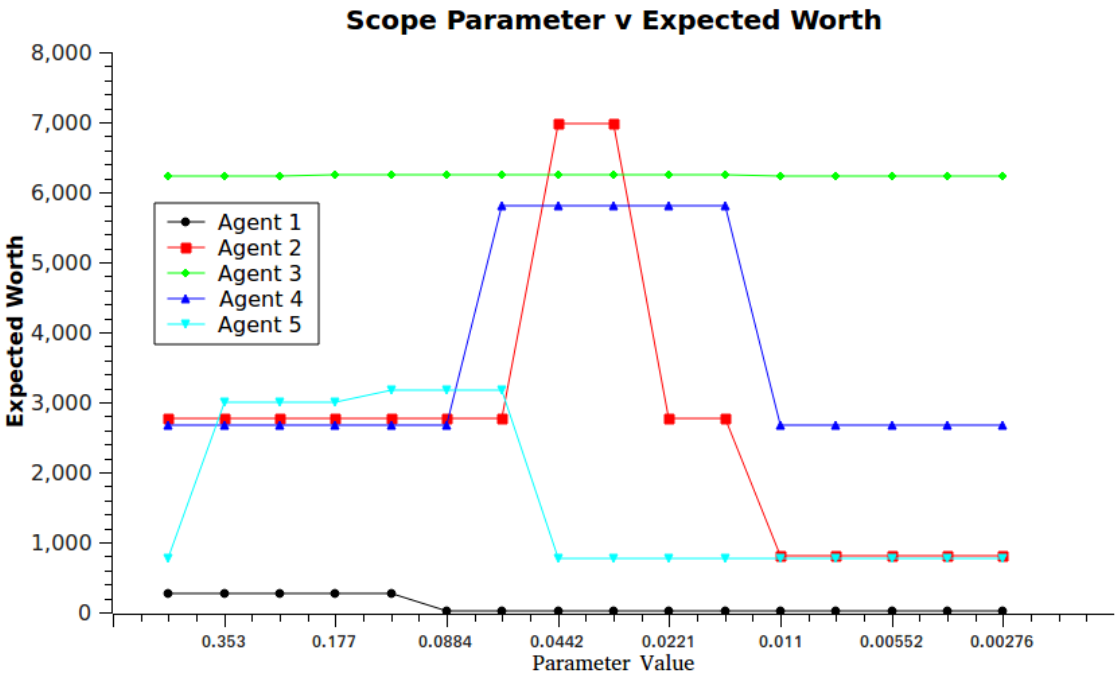

Figure 4.5: Expected worth of parameters.

few can gain worth, even without competition from more specialised components. Against these most specialised components they will also fail to gain worth, as their errors are high due to the need to average across so many different domains. Their key advantage, though, is that they win any competition in which they are the only competitor. With such a large scale they cover many areas which no other component does. While the agent will eventually fill in these regions will more accurate components, during the initial learning phase these components can make huge numbers of predictions. These components serve the role of fallback predictors, used if the agent has nothing better to use.

This may be a large proportion of the time, especially during the early learning period, so a generalist component may accumulate large amounts of worth before competition arrives. Even afterwards, if competing components are not evenly spread it may still have a high proportion of its predictions be accepted without competition. This results in the highest-worth components being generalists.

Meta-Learning of component scope parameter

While this test only had a single component type, the agent was able to assign the scope to the new components it created. As such, it was able to learn which values produced the highest worth components. The parameter was a value between 0.5 to 0.00276, chosen from 16 options, each $1/(2^{0.5})$ smaller than the previous.

The expected worth of a component, for each of these parameter options, is shown in Figure 4.5. It is after the full 250,000 time-steps of training, allowing sufficient time for a degree of meta-learning. Figure 4.5 indicates that, on average, the agents expected high-

scope components to be worth a moderate amount relative to other components. The highest expected worth occurred between 0.09 and 0.02. Anything past that, those which would lead to components responsible for less than 0.02 of the environment, was expected to produce very low worth components.

The values on predicted in Figure 4.5 are not directly equal to the component worths seen in previous figures. Those are total worth received by component divided by age, while this figure displays expected total worth received. Components are ranked against each other by *worth/age* in order to allow newer components to be added into the agent's component pool, and compete fairly against existing components. If a newer component is now achieving a better *worth/time_step* than an older component the older of the two should be removed, regardless of its total lifetime worth accumulation, since it has now been superseded.

When the agent is selecting which components to build and which parameters to use, however, it is simply concerned with the total worth the component will accumulate before the evaluation of that component occurs. Since this evaluation occurs after the same number of time-steps for each component there is no need to divide by age.

In terms of meta-learning, Figure 4.5 indicates a degree of success. Certainly the extremely low scope values (<0.02) were seen to result in extremely low worth components. This would translate to far fewer components being made with these parameters, therefore improving the agent's overall performance, as such components contribute little to the overall predictive effectiveness of the agent.

A greater number of components would need to be created, in a variety of separate environments, for actual meta-learning to occur. As with any machine learning, meta-learning depends on sufficiently large data sets to function. This will be investigated in later tests.

4.3 Meta-Learning Demonstration, using the Weka toolkit and the MNIST dataset

This test examines how the agent would select between a set of machine learning algorithms to best maximise its predictive accuracy. It uses five algorithms from the Weka toolkit [13].

The task is a visual recognition task between two different characters from the MNIST dataset [33], specifically classifying between the digits 4 and 5. The MNIST dataset was chosen for its common usage in machine learning fields. Rather than simply randomly subdivide the data to form training and testing subsets, it uses the human generated handwriting from separate individuals between training and testing, leading to far greater confidence that success on the MNIST data would translate into success on newly acquired real-world data.

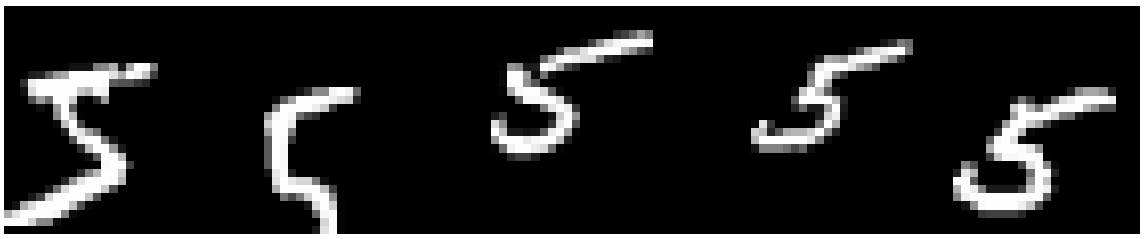

Figure 4.6: Example of MNIST data. The character partially out of frame is intentional, and is how a portion of the data was presented to the agent. For ease of viewing, all five images have been rescaled from 28×28 to 56×56 pixels, with no interpolation.

To reduce the accuracy of the algorithms, to improve test clarity, the characters are randomly translated by a few pixels. The size of the transform is uniformly distributed, independently on the x and y axis, to a maximum magnitude of half an image size. Images are 28×28 pixels, therefore maximum absolute transform distance per axis is 14 pixels. An example of five such images is available in Figure 4.6.

4.3.1 Structure of the test

1024 training images were generated for the two classes in this classification task, for a total of 2048. The two classes were the characters '4' and '5'. All images were taken from the MNIST training set then transformed by slight tranformation. This transformation consisted of two translations, one on the x and one on y axis. They translated between 0 and 14 pixels, randomly chosen uniformly. Images were allowed to lose parts of the glyph in this process, if it was partially translated out of the image. This data would simply be lost to the agent.

The images were then presented to the agent as a 768 element input vector, with a numeric value representing the class provided in the next time-step as reward. This allows the agent to view the problem as a prediction task, as it is predicting the next value of its reward signal based on the input feature vector. Both image sets were merged and shuffled into a single training set which was presented to the agent twice. During this time, the agent was permitted to create 6 separate components, which were trained on the presented data, attempting to learn then relationship between the data and the reward signal.

The components available were simply adapted classifiers, designed to either predict 1 or 0, depending on the class. They adapted the reward signal into a boolean class signal for use by the weka classifiers. Every component had a full scope, requiring it to make a prediction in every case. This resulted in only one component being 'chosen' by the agent. It received reward, and was used for future predictions, while all others would receive near zero worth, as they had their predictions be ignored. As such, the only way to maximise component worth was to reduce component error, by picking the component type with the best chance of correctly predicting the class from the input features.

After the two passes through the training data, a new testing set was derived from the MNIST data, again with translations. This testing set was taken from the MNIST testing corpus, and consisted of, as with the training set, 1024 instances of the character '4' and 1024 instances of the character '5'. The reward signal was disabled, preventing the agent's components from learning any further from this data.

Meta-learning was made possible by repeatedly presenting new training and testing sets to the agent. After each presentation all first-level learners, those predicting the classes of the presented glyphs, were purged. Meta-learning components, however, remained.

All that remained in each cycle was the components responsible for predicting the worths of new components, those which enable meta-learning. A new test could then be run, generating a new set of components on a freshly generated training and testing set. These image sets would be randomly generated by subselecting from the MNIST corpus and translating the characters position, leading to a structurally similar but still distinct problem from the previous. 50 tests were performed, after which the agent's expectations for the worths of each component type was recorded.

Six separate component types were available to the agent in this test, corresponding to six separate Weka algorithms. These were:

1. A Multilayer Perceptron
2. A C4.5 Tree
3. A Naive Bayes Tree
4. A Logistic Linear Model
5. A Voted Perceptron
6. An Instance Based Learner (closest single instance)

Where possible, they used the default parameters supplied by Weka. Only the multilayer perceptron had its parameters changed, to alter the number of hidden neurons to a single layer of 8.

4.3.2 Results

The testing process was run five times, so five separate agents were trained, and their meta-learnt valuations of the algorithms, corresponding to their expectations of the various classifiers' accuracies on this problem, were recorded. They are compared against the accuracies (proportion of correct classifications) of the algorithms when run independently on newly generated training and testing sets. This accuracy is referred to as the **comparative accuracy**, and can be seen as a measure of the 'true' usefulness of the algorithm for this particular task. Table 4.1 displays these results, and has been sorted based on the agent's expected worth for each component type.

	Expected Worth	Comparative Accuracy
InstanceBased	3511	0.95
NaiveBayes Tree	2045	0.80
C4.5	1151	0.74
Logistic Linear	475	0.68
Multilayer Perceptron	333	0.50
Voted Perceptron	168	0.66

Table 4.1: Agent's expected worth for a given algorithm against algorithm's performance in separate testing (accuracy in terms of proportion classified correctly)

As can be seen, the agent's expected worth correspond well to the accuracies of the various algorithms for this task. It deviates from the ordering which would be produced by sorting according to the comparative accuracy in its evaluation of the worth of the lower three, but these are algorithms which both the agent and the external evaluation agree are less than ideal for this task.

This indicates a degree of meta-learning on the agent's part, as it was able to select between algorithms to improve the predictive accuracy of its components. This is an example of the 'portfolio of algorithms' approach to meta-learning.

4.4 Component Stacking: Deduction of hidden variable from recent error rate

This test shows the ability of the architecture to use the outputs of existing components to improve the performance of newly created components. The task requires the agent to be able to deduce a hidden boolean variable from the results of its actions.

4.4.1 Task

The agent has two actions at every time step, termed A and B. It is also presented with a single feature input, an integer value between 0 and 7, inclusive. This value is the external state of the world and is termed the $VISIBLE\_INDEX$. It indexes into a reward table, which the agent must learn, which as been populated with values uniformly randomly distributed between -10.0 and 10.0. This reward table is generated when the environment is created and remains constant for as long as the agent is running, which is the full duration of any given test.

For the first 64 time-steps of the agent's existence in the environment, the task is a simple learning task. When it selects an action the reward is initially looked up in the reward table, termed $REWARD\_TABLE$, using the visible variable $VISIBLE\_INDEX$ as a key. This reward is then modified based on the agent's action. If action A is taken, the agent receives $REWARD\_TABLE[VISIBLE\_INDEX]$ as reward. If it selects action B, however, it receives the negative, $(-REWARD\_TABLE[VISIBLE\_INDEX])$. This allows the agent to learn when to perform action A, and when to perform action B. It is

able to learn to maximise reward by inverting the value in the table when that value would be negative, and as such can consistently gain a positive reward.

After 64 time-steps, however, the hidden variable is randomly altered. The hidden variable is a boolean value in the environment which randomly changes every 64 time-steps. It acts as another inverter. If the variable is true the reward will be as previously received, but if it is false the reward will be multiplied by -1. This hidden variable multiplier, termed $HIDDEN$, is therefore either 1 or -1. If the agent's action is a multiplier termed $ACTION$, which is 1 if the agent selects action A, and -1 if the agent selects action B, the reward received for a given action is $REWARD\_TABLE[VISIBLE\_INDEX] \times ACTION \times HIDDEN$.

This means that regardless of how well the agent can infer the reward table, its expected reward is zero if it cannot infer the hidden variable. If the hidden variable were to alter every time-step, the agent would have no way of devising a policy with better results than random action selection.

4.4.2 Component Architecture

The agent in this test has only a single component, a non-linear regressor able to learn to map inputs to rewards, and thus make predictions as to the results of actions. For as long as the hidden variable remains constant, the agent is able to use this component to predict the rewards of its actions, and thus achieves above chance results. However, this component alone cannot solve the task if the hidden variable is altered.

When the hidden variable is altered, components which were previously successful are now predicting the inverse of the rewards received, and as such are producing large errors. The agent which depends on these predictions therefore takes incorrect actions, and receives negative reward on average, as the hidden variable is inverting the results of its previously successful decision policy.

The solution comes from **component stacking**. The existing components have a single output, their previous time-step error rate. They also have an additional input, which the original components were unable to any make use of. This input connects to another component's output, and serves to limit the scope of prediction of a component. The original components are unable to use this input simply because there is no existing component to link to.

The components (other than the very first) define their scope based on their inputs, which is the error function of another already-existing component, their supplier. After each prediction is made, this supplier component sets its output variable to the error that prediction incurred. The user component then takes this value, and averages the values it sees. This builds up a picture of the range of errors its supplier component produces. It then selects as its scope either the time-steps where the previous error incurred by its

supplier component was higher than lifetime average or those in which its error was lower than the average. This creates a scope of roughly 50% of all time-steps, since the errors correspond to the hidden variable's state, and that is distributed equally between both boolean values.

This process relies on the first component, the one whose error function serves as an information stream to the second component, having a bias.

The first component has a full scope, and predicts at all time-steps. It learns to predict based on the first 1024 time-steps it sees. Due to the random nature of the hidden variable, these will not be uniformly distributed between states in which the hidden variable is true and those in which it is false. This creates a bias, in which its error will be correlated to the state of the hidden variable. If averaged over time, the state of the hidden variable can be inferred from the first component's error. The second component uses this information, and learns to predict only a subset of the environment. This subset is correlated to the state of the hidden variable, and therefore the second component's predictions are far closer to the actual result than chance.

This results in the agent relying on pairs of components. The first attempts, but fails, to learn to predict its environment based on visible input, and has its limiting input connected to a non-informative input stream. The second learns to predict based on the visible input, as the first, but limits the time-steps in which it will predict based on the error of the first component's previous prediction. It will either predict when its donor component has a recently high error or a low error, depending on its randomly chosen parameter.

The agent uses these predictions to select its actions. Each time-step, a number of second-level components will make predictions, depending on the error of their first-level donor component. The agent selects the second-level component with the lowest historic error, and uses that components predictions as its expected results for taking either action.

Importantly, no second-level component has a full scope, and each only predicts a subset of the environment. The agents predictions rely on at least two second-level components existing, each handling a portion of the conditions the agent finds itself in.

4.4.3 Results

The scores over time two batches of agents are shown in Figure 4.7. The agents with component stacking enabled clearly out-perform the agents without, and demonstrate progressive learning from a low start to a flat 'mature' state, after which no further learning was achieved.

The results show the agent behaved as expected. There were 40 runs of a control test, in which the agent was unable to usefully link components together. This was achieved by forcing component error outputs to 0, leading to them providing no useful information to

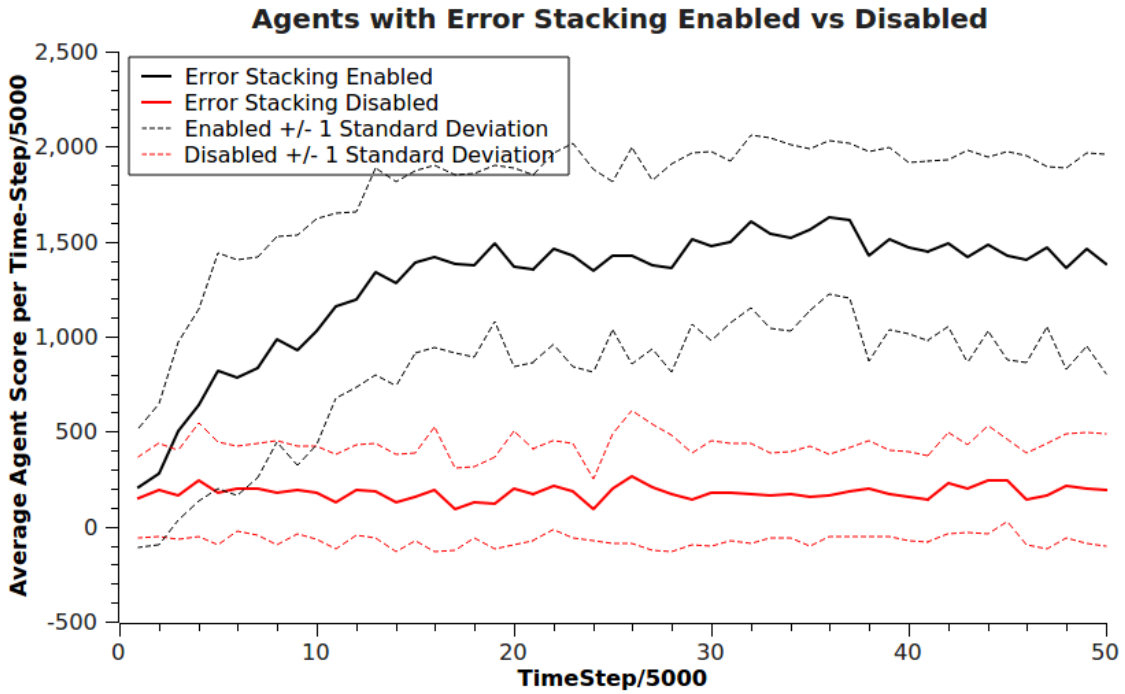

Figure 4.7: Average performance of 40 runs for agents with stacking enabled and agents with stacking disabled

their users. In this configuration agents showed no major learning, with the agents' scores remaining close to the baseline which would be achieved by chance. 40 runs were then performed of the agent without these limitations, in which the components were able to use other components' outputs. These showed consistent learning, achieving results well above chance.

No meta-learning occurred, nor was any possible, as no parameters were available for the agent to set.

It is worth noting that the random result would average to 0 reward/time-step, but the non-stacking agent still performed at a rate greater than this level. This is as of yet not completely understood, but appears to be related to structurally similar components with slight biases and extremely similar historical errors alternating based on the recent errors seen due to the hidden variable.

4.5 Reinforcement Learning mechanism

The agent presented in this paper, till this point, used a simple action-reward framework, in which each action receives a reward immediately, and no action depends on any other for success. For these tasks credit assignment, determining which action produced which reward, was therefore straightforward. In this task, however, reward depends on multiple actions occurring in sequence, a more difficult learning task. To expand the agent for longer term learning work task, a simple Temporal Differencing framework is employed.

When operating in a Reinforcement Learning environment, the agent's actions are stored in a queue, and are rewarded based on the rewards of not only that action's time-step, but the following N time-steps, where N is designer-specified. These are discounted by a designer specified parameter, λ. Total reward for an action at time-step t is therefore defined as:

$$\sum_{n=0}^{N-1} R_{(t+n)} \lambda^n$$

This is a standard Reinforcement Learning technique, which allows for action-chains to be credited for their end result, even if many of the actions within the chain produce no noticeable effect in and of themselves.

The components therefore, under this framework, predict this discounted sum of future rewards, for the next n actions, rather than simply the reward for the next action. Other than this, the agent behaves the same, and components competitively attempt to predict these rewards, by reducing their scope to improve their accuracy.

4.6 Reinforcement Meta-Learning: Useful feature discovery

This task tests the agent's ability to discover which sources of information provide useful information, as opposed to those which are simply noise. It is also a test using the agent's reinforcement learning capabilities.

4.6.1 Task

In this task, the agent must navigate a grid environment in order to achieve reward. To do so, it is presented with a series 16 of feature vectors. 15 of these are noise and 1 is its (x,y) cartesian position on the grid. While the non-noise input is marked, the agent has no a-priori knowledge of this. In this test, the marking is simple. Each vector has 3 elements. The 15 noise feature vectors have their values first two elements set to a random (x,y) co-ordinate within the grid borders, and their third value set to -1. That is to say these vectors are $\{random, random, -1\}$. The useful information vector is set to the agent's (x,y) co-ordinate, with their third element set to 1, so $\{x, y, 1\}$. This creates an easily recognised difference between the vectors, to test the agent's ability to learn to use it.

The component available to the agent in this case, other than the meta-learners, was a simple instance based learner. The learner was designed to replicate a tile-based RL strategy, attempting to assign a single averaged record to each (x,y) co-ordinate on the grid. The learner simply averaged the results for each action (the TD-discounted sum of the rewards for the next n time-steps), and used those as its predictions whenever the agent returned to that particular part of the grid.

The information it used to infer the agent's current position was derived from 1 of the 16 vectors presented each time-step. The agent was able to target its components when

they were created, picking a particular "prototype" vector, Z_j chosen from 1 of the 16 presented during the time-step the component was created on. The component recorded this representative vector, and selected as its input the vector most similar to it from the 16 possibilities.

Similarity was a weighted Euclidean distance from the prototype vector. The component took all 16 input vectors, and select the one with the lowest distance to its prototype vector. This was the Euclidean distance, weighted by a random vector k. This vector was generated for each component, with its values randomly distributed between 0 and 1. The distance was computed by:

$$distance(Z_j, input) = \sum_{i=0}^{2} k[i](Z_j[i] - input[i])^2$$

This lead to each component valuing a different portion of the input vectors to a different degree, when comparing them against their feature vectors. A component with $k = (0, 0, 1)$ for example, would order the input vectors to select those closest to its prototype vector's 0th element, corresponding to the x co-ordinate. In this test, such a component would be useless to the agent, as it is the last element which indicates whether an input vector is a useful information source.

If the component was created with the useful vector as its representative vector and a weighting vector which selected for inputs with a similar last element, it would be able to deduce the agent's (x,y) position, and thus its predictions would be useful to the agent. If it had not been given a useful representative vector, it would only read in random noise, and thus its predictions would not be above those achievable by chance.

The environment was a grid of 25 by 25, with a reward of 100 given to the agent whenever it reached (12,12). When it reached the reward it was randomly moved to another position on the grid. The agent could move in all four cardinal directions, and simply failed to move if it selected an action which would move it off the grid, wasting a time-step. The agent created a new component every 1500 time-frames.

To test the meta-learning capabilities, all non-meta-learning components were purged every 135,000 time-steps. This completely removed all ability of the agent to predict its environment, as all meta-learning components in this test were prevented from making any predictions as to the results of environment-altering actions. This allowed the agent to retain its learning as to the expected value of component configurations, but not the expected value of its movement actions.

The expectation was that an agent with meta-learning components would be able to more swiftly relearn its environment, as it knew which 'landmarks' to look for, which vectors in its input set were useful information and which were not. The components were also limited to only being allowed to record results for 125 input values, out of the environment's

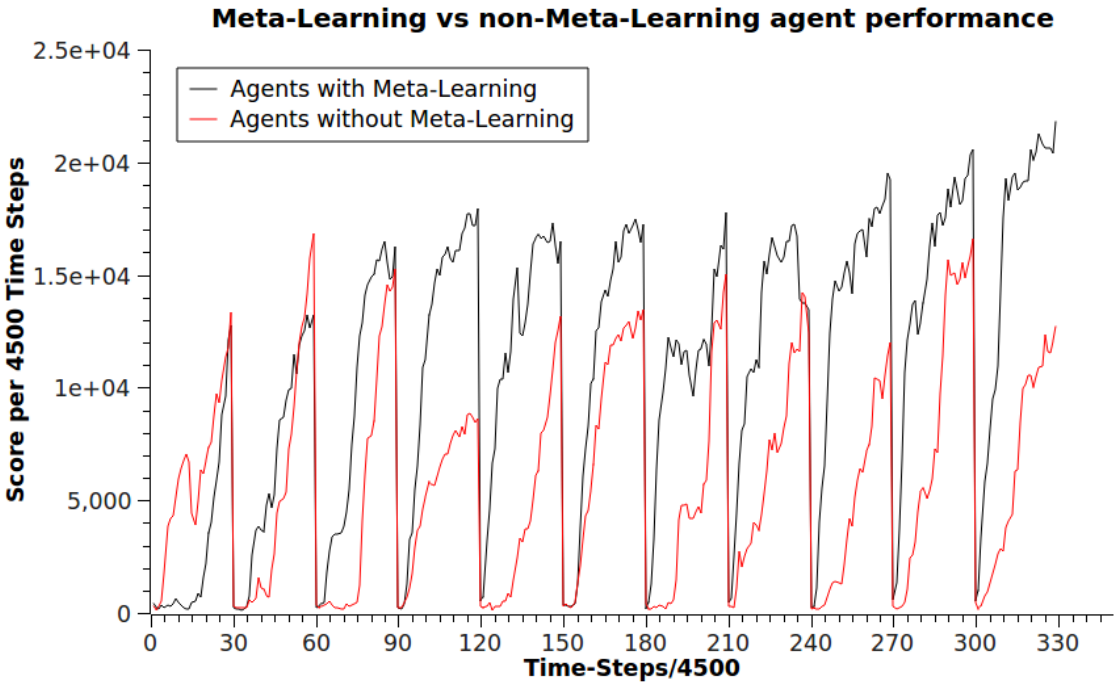

Figure 4.8: Average of 5 agents' scores over time, resetting learning every 135,000 time-steps.

total 625. Once they had taken 125 distinct input-vectors they would take these as their scope, and not make predictions for other input-vectors. This was done to require multiple components to cover the entire grid, so the agents would have to repeatedly build useful components in order to achieve a full optimal policy for the entire environment.

4.6.2 Results

Five agents were run with meta-learning capabilities, five were run without, and their performances recorded. Each was run for 1,485,000 time-steps. The agents' non-meta-learning components were purged every 135,000 time-steps, providing 11 separate learning episodes of 135,000 time-steps, 10 of which could involve meta-learning from the previous episodes. As expected, the meta-learning agents out-performed the non-meta-learners over the course of test.

As can be seen in Figure 4.8, both sets of agents performed roughly equally in the first episode (graph increments 0 to 30). The non-meta-learners, in red, out-performed their competitors, due to pure chance, but both reached a reasonable behavioural policy by the episode's end. The average reward then immediately drops, as the agents' components are purged, and they revert to a blind wandering strategy.

As shown in Figure4.8, in episode 2, from 30 to 60, the meta-learning is slightly visible, but not heavily pronounced. Other than a single agent which underperformed, the meta-learning agents were able to formulate an above-chance policy faster than their non-meta-learning competitors, due to their recognition of useful environmental features. This is

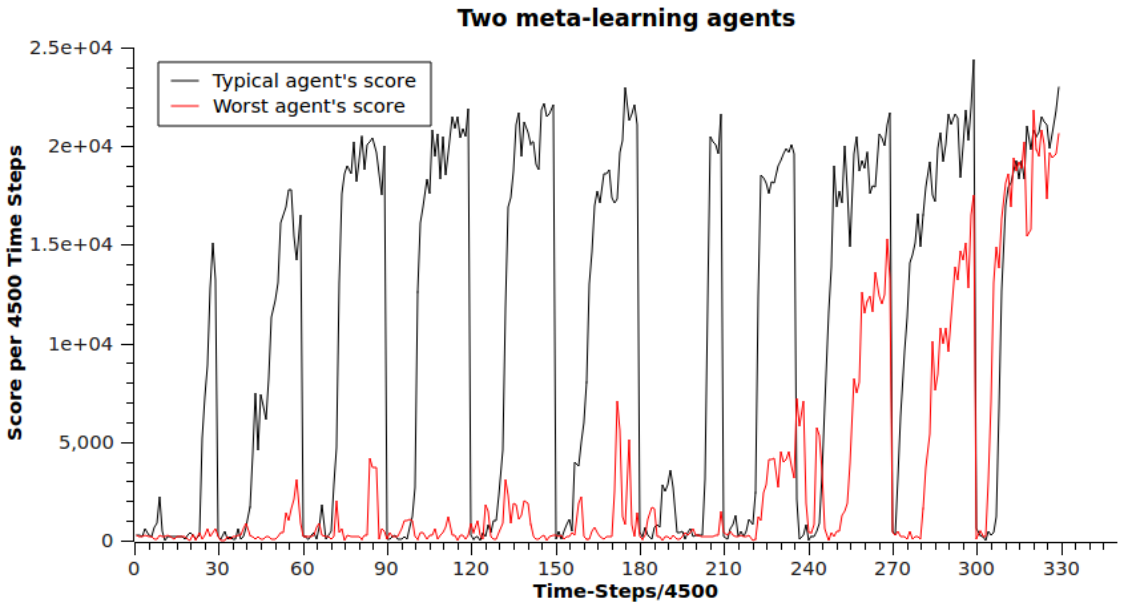

Figure 4.9: Comparison of the scores of two agents. The first (a typical result), and the second (which failed to perform to the same degree as its siblings).

reflected in the slightly earlier rise in score, as their components were better targeted to the useful environmental cues. It is believed to be by chance that the non-meta-learners out-performed the meta-learners towards the very end of episode 2.

From episode 3 onwards, the meta-learners clearly outperform the non-meta-learners. Learning occurs earlier, as the agents require fewer training examples to successfully learn a useful policy. Since components are finite in number and multiple successfully targeted ones are needed for a full policy the meta-learners also out-perform on final score, by the end of all episodes after the 3rd, as well as training speed. While the non-meta-learners achieve a degree of success by randomly targeting their components, they never produce sufficient components to reach optimal performance.

4.6.3 The Failed Agent

As can be seen in Figure 4.9, one of the five meta-learners failed to achieve a useful policy until late into the task's total runtime. The typical agent, in black, performs as the others did, reaching a high score in all episodes after the 5th, showing far faster learning than on its initial episodes. While its performance was not perfect, it was strong and consistent.

The failed agent, however, achieves almost no progress from the random walker until episode 8, starting a graph increment 210 (time-step 1,080,000). It is not until episode 9 that it achieves the performance the first agent achieved in its 2nd episode.

It is believed this agent simply repeatedly failed to correctly target its components due to unlucky exploration. By repeatedly selecting noise inputs, rather than useful ones, its components had no ability to predicts the results of its actions, so it was unable to form a

useful policy. Without examples of successful components, its meta-learners were unable to infer which environmental features were information and which were noise, so it continued to randomly assign its components.

This continued till it eventually converged to the correct solution, and can be seen to follow the previous agent's learning patterns after than point. In the last episode, episode 11, it can be seen to have achieved a result roughly comparable to the typical agent's, both in terms of final score-per-time-frame and speed of learning.

4.7 Expanded MNIST testing

This test was designed to assess the agent's ability to handle task subdivision amongst multiple components, using real data and Weka algorithms. It was similar to the meta-learning test studied previously, but with 20 glyphs to recognise, spread amongst 5 classes. This makes it a far more complex task than the previous MNIST test, which involved only 2 glyphs (5 and 4), and 2 classes. The classifiers from the Weka toolkit remain the same as described in Section 4.3, forcing the agent to use multiple binary classifiers only able to select between 2 classes to solve a 5 class classification task.

The task consists of choosing which digit, between 0 and 4 inclusive, a given image represents. The dimensions are as previously, 28×28 black and white images, forming a 784 long feature vector for each time-step. 4096 images for each digit are loaded from the training corpus of the MNIST data set, then rotated by 0, 90, 180 and 270 degrees, with each rotation saved as a separate feature vector. All four are assigned the same class, the digit they represent, but can have very different visual appearances. No other transformations are made to the images.

This creates a more complex classification task, as a generalised shape for each digit can no longer be assigned. While a 0 may appear similar under all rotations, a 1 has two distinct shapes it will assume, and the other digits, 2, 3 and 4, will appear very different under all rotations.

This task is similar to the previous task subdivision test, but differs sufficiently to be worth investigating. In the previous test, all components used subdivision to maximise their worth by balancing predictive scope against predictive accuracy. In this test, however, those pressures must also be balanced against the inability of the components to predict more than one class.

Every component was provided with a scope-setting unsupervised learner. When the component was formed, it recorded the inputs presented at that time-step. It used this feature vector to sub-select a portion of all the feature vectors presented during the component's lifetime. As before, this occurred by forming a sphere within the space of all possible feature vectors, attempting to encompass an agent-selected proportion of all future inputs.

This unsupervised learner therefore worked based off the Euclidean distance between its 'prototype' image and the one currently presented to the agent.

Using this scope, the component then collects 1024 feature vectors from those which are sufficiently close to its prototype and uses this as a training set for its WEKA classifier. This classifier attempts to predict if a given feature vector belongs to the same class as the component's prototype initial formative training image. If so, the component makes its prediction, that the current feature vector belongs to the class that the component was built to predict. If it does not the component makes no prediction, as it cannot determine which of the 4 other classes this image might belong to.

The component, therefore, only makes a prediction if both the unsupervised Euclidean learner, broadly restricting the component's predictive scope, and the Weka learner, return true. The unsupervised learner attempts to subselect a set of images which appear superficially similar to the first one the component was presented with, its prototype. This is expected to have some degree of correlation to that prototype's class, but be too simple to achieve a high accuracy. It will, however, avoid the Weka algorithm needing to waste capacity on learning the 'easy' problems, leaving it to specialise on the specifics.

It is entirely possible that the unsupervised learner would fail to recognise many of the rotated digits as similar. The digit '1' rotated through 90 degrees is superficially very dissimilar to how it appears before rotation. For this reason, the agent is expected to require multiple components for each digit, or class.

4.8 Results

The results for this test were fairly disappointing. A single agent was trained, in order to examine the internal structure of its components. It was then compared to a basic multi-class classifier, made from one binary classifier per class, designed to represent a standard solution to the problem. The classifier used was the basic J-48 decision tree from the Weka toolkit, running with default parameters.

In the agent's case 35 training passes were performed, which from previous runs had been shown to be sufficient to allow the agent to reach its maximum performance. Each of these training passes randomly selected 24,000 training samples from the 81,920 generated initially (4096 images of each digit, 5 digits, 4 rotations per image).

Following these 24,000 steps of training a testing pass was performed. This occurred using a newly generated image set from the MNIST testing corpus, as opposed to the MNIST training corpus. Again, 4096 images were loaded for each digit then all four rotations added to the testing set. A random subset of 12,000 images were then selected from this testing set. The results can be seen in figure 4.10.

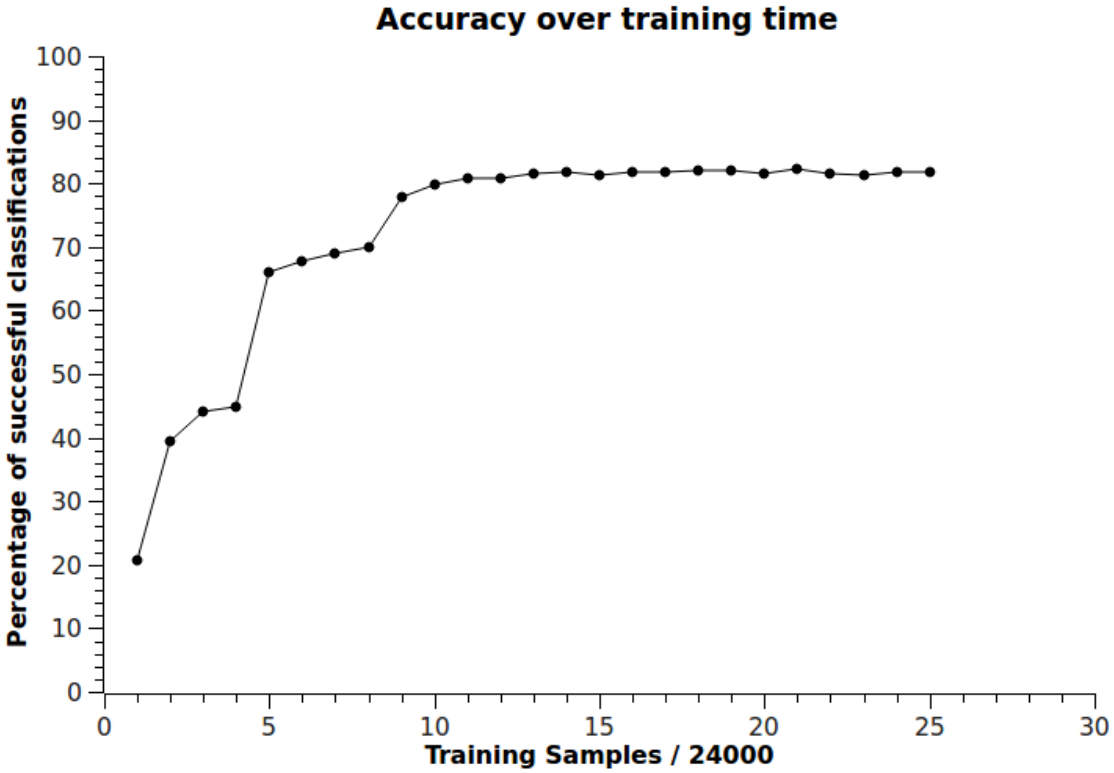

Figure 4.10: Percentage accuracy of the agent as training progresses.

As can be seen from this figure, the agent reaches maximum accuracy at 480,000 training samples. The maximum accuracy is 9,882 correct classifications out of a possible maximum of 12,000, or 82.35%. In comparison, the rival multi-classifier achieved 10,059 correct responses out of 12,000 tests, or 83.825%. The agent can therefore be seen to have comparable accuracy to this comparison classifier. The rival classifier was presented with only 40,960 training cases, and performed only a single training pass per sub-classifier (with one sub-classifier for each class). This means the rival used fewer training cases to achieve better performance in lower clock-time.

This is disappointing as it was expected SAMLA would exceed this rival. This seemed possible due to the agent's ability to subdivide the problem based on both feature inputs and classes, rather than purely on classes. The rival takes each training case and uses it either as a true or a false case for each classifier, with one classifier being assigned each of the five classes. This agent, however, subdivides along these lines, but then further subdivides based on the feature vector it is presented with, so a component might only handle upside down versions of the digit '2' for instance, while the rival would have a single classifier for all cases of the digit.

To investigate why it was unsuccessful, despite having the opportunity to use more classifiers than its rival, closer inspection of the agent produced is required. Figure 4.11 displays the worths the agent has placed on its components sorted by the age of these components. In a functional agent on an statistically stationary problem the agent should

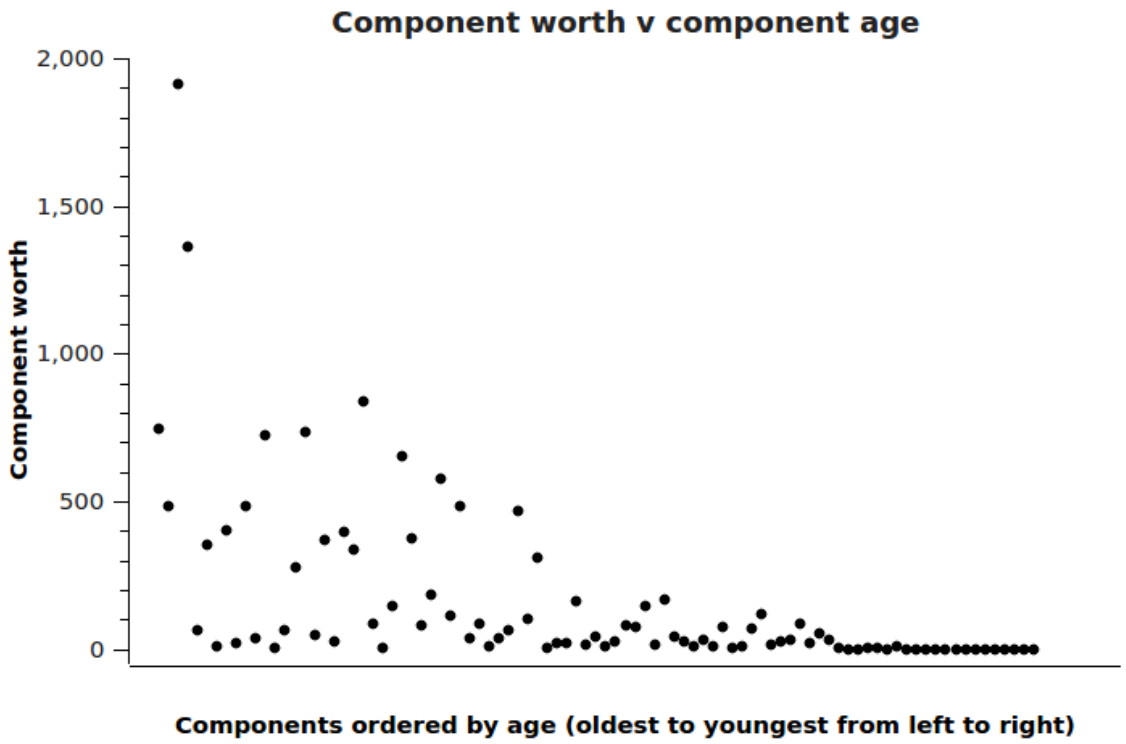

Figure 4.11: Graph of the worths the agent attributes to its predictive components, sorted by age.

assemble a set of components which solve the problem as accurately as possible, then reject further additions, while preserving this existing core. As can be seen, older components have a higher worth than younger ones. This means their predictions are being used more per time-step on average than the new additions. Since these new additions are not improvements over the older ones, this indicates the agent is functioning normally in this regard. This matches up the results seen in the earlier tests.

Figure 4.12 shows component error against component scope, illustrating how their accuracies were affected by their degree of specialisation. It indicates that the source of the problem may be in the subdivision function the components use. In figure 4.1, from the first test in this section, it was seen that the component's scope of prediction was connected to its error rate. Specifically, as the scope approached 0, meaning the component in question made almost no predictions at all, its error also approached 0, indicating it was almost perfectly accurate. This is expected, as it indicates specialisation. On the other hand, increasing scopes to, say, 0.5 greatly increased the error. Such a component would make predictions 50% of the time, and would be a generalist, which the agent would use if it did not have access to a specialist for that part of the environment. In the previous test (Figure 4.1) this was seen to occur.

In this test, however, specialisation showed a negative effect on the error rate. As scope increased average error rate decreased. While some highly specialised components were able to match the rates of those which were given a scope of 0.5, none did any better. This

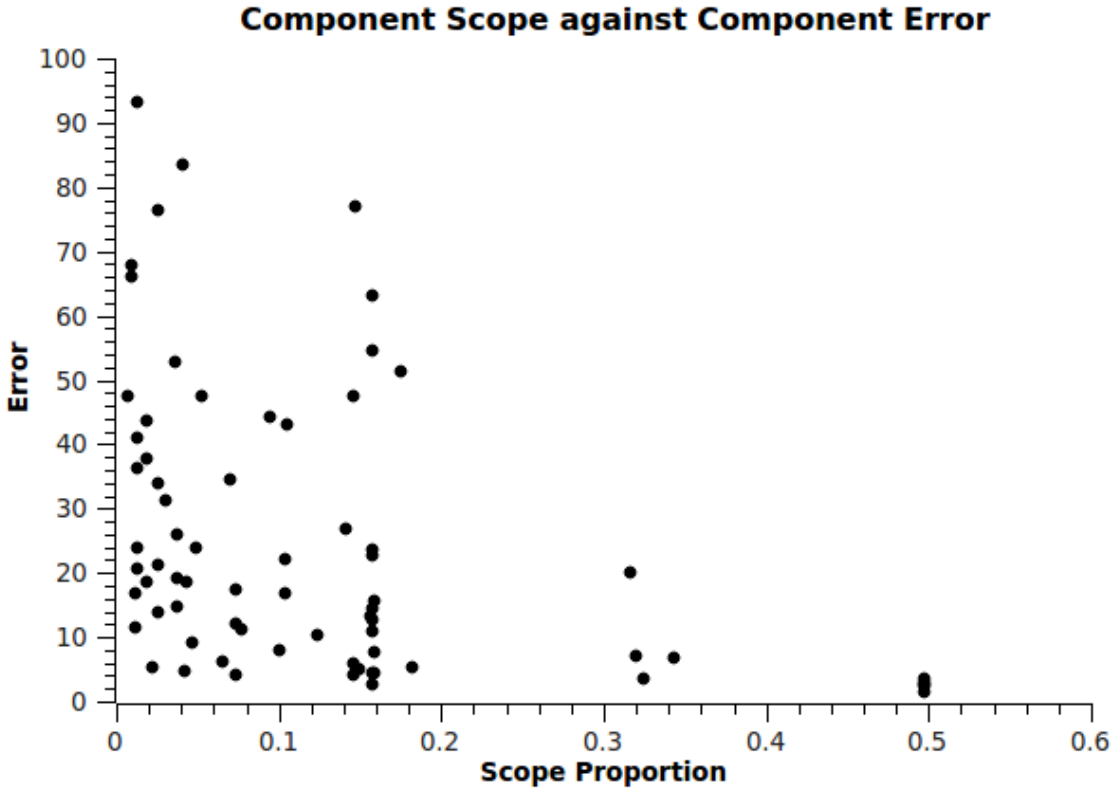

Figure 4.12: Graph of errors of each component in the agent's pool against the proportion of the environment it takes as scope. Error is for relative comparison only, units and absolute values are irrelevant.

indicates the component specialisation strategy was ineffective and possibly detrimental. Possibly a different strategy would have worked, but the one used did not make the task easier for the machine learning algorithms the components were using.

Specialisation of components is a key feature this architecture relies on to out-perform more basic machine learning algorithms. Interconnection of components is another, but was not available during this task. Meta-learning which algorithms are successful is another, but this test was too short for any major meta-learning to occur, so the agent had no advantage there.

The agent equipped with this particular component design did slightly worse than the rival classifier. While this architecture is not specifically designed to be the optimal classifier, it would seem reasonable to assume at least some performance could be achieved with a different component architecture.

It is also worth noting that the components were only able to train their internal machine learning algorithms with 1024 training cases. This was due to memory constraints, as many components would be training simultaneously. Conversely, the rival algorithm was able to use its full 40,960 training cases on each of its machine learners. A rival was tested using only 1024 training cases for each digit, and only achieved an accuracy of 7,448 correct

classifications for 12,000 tests, or 62%. This is below the agent's achieved results.

4.9 Discussion of results

The preceding tests may not have established the agent as the pinnacle of either reinforcement learning or classification, but have demonstrated a wide range of capabilities possessed by the agent. They have validated the agent as a meta-learner, confirming that its architecture functions as expected, able to expand usefully, without constantly replacing existing internal structures. These structures have been shown to be kept based on usefulness, and able to build upon each other.

In terms of meta-learning, the agent has been shown to have success in portfolio algorithm selection, algorithm stacking and parameter selection. This fulfils the thesis' primary aim of producing a meta-learning reinforcement learning agent. It has also been shown to be able to learn to target its learning, learning to recognise useful sources of information from useless ones. This meta-learning capability allows for faster learning on future tasks, assuming similarity between the environments these tasks occur in.

5 Experiment into Self-Directed Information Seeking

This section covers a set of tests of a tentative future direction for the architecture. It is separated into its own section due to being less rigorously investigated than other areas, and being outside the original goals of the architecture. It was added because it seemed a powerful mechanism which naturally arose from the abilities of the component-valuation mechanism which already existed, and so could provide a unique capability to agents constructed using the SAMLA architecture.

5.0.1

Learnt Information Seeking A key issue with the agent as it stands is its passive approach towards its environmental inputs. While it acts to maximise its utility by taking the action with the highest expected return at each time step, it does so purely on the information directly in front of it, in the form the designer has provided it.

It would be far more useful to have an agent able to seek out information, based on the expected usefulness of that information, in terms of improvements to the agent's predictive accuracy. The simplest of these actions might be simply to re-orient itself to look in a different direction, or focus on a different object. A single action, taken to gain a different perspective. The agent could face a task, look around to gain a better understanding, then turn back to the task and solve it with the aid of this new information.

Longer term actions might involve interacting with objects, say by picking them up to determine their weight or tapping them to determine their composition; it might also involve travelling to other parts of the environment, a learnt sequence of movements. In general, it requires normal Reinforcement Learning approaches with goals determined based on information-gain, rather than designer-provided utility functions.

It is possible that the component-based architecture of this agent allows some useful insights into which pieces of information would be useful to the agent, and at what times. The core principle of this is that the components should have the ability to sway the agent's behaviour based on their worth. Worth is a measure of how much the component contributes towards the agent's predictive ability, so it would seem reasonable to assume the information which a high worth component uses also contributes towards the agent's predictive ability.

5.0.2 Competitive Hidden Variable Predictions

An alternative approach to learnt information seeking is to deliberately attempt to find information resolving an unpredictable element in the environment. This creates a focus, allowing pieces of information to be evaluated. In the "component is assigned a condition under which it records" approach the agent relies on its meta-learning setting this condition usefully, or in a scattershot approach in which it makes large numbers of components and discards most. The hidden variable prediction approach, however, allows the agent to learn information sources through competitive predictions, similar to those routinely used.

The first step is to select a phenomenon in the environment to be predicted. This should presumably be one it has no current ability to predict. A new component is created, termed the DEFINER. The DEFINER is assigned a description of this phenomenon, for example "is at location A, took action B, received reward > 0". The DEFINER can then determined whether or not the given action in the given circumstance lead to the given result or not. This means that if another source within the agent makes a prediction as to whether or not the defined phenomenon will occur or not the DEFINER can feed back an error for this prediction.

The WRITER, therefore, is a type of component which is assigned a given DEFINER to make predictions with respect to. The DEFINER is, in effect, a hidden variable the WRITERs are attempting to predict. At any point any WRITER may make a prediction. The DEFINER then stores all predictions made by all WRITERs which are assigned to it until it next encounters its assigned condition+action. Either the phenomenon, in this case the result of an action, occurs or does not. All the stored predictions can now be evaluated. Since the result is a boolean it can be described as either a 1 or 0. This allows the WRITERs to be assigned errors, based on how accurately their predictions match the results. Predictions are purged at this point, now that they are evaluated.

The final component is the USER. It is also assigned a DEFINER, and uses the information the WRITERs of that DEFINER produce. It takes as information the best prediction made so far. That is to say (in this example) it takes the most recent prediction by the most accurate WRITER as to whether the agent will receive a positive reward if it takes action B in location A. It then makes predictions as normal, presenting them to the agent. The agent can then use these predictions, if they prove to be accurate.

In summary, the DEFINER is a component designed to record whether a given condition-action pair result in a given result. WRITERs competitively attempt to predict this outcome, and receive feedback as to their error from the DEFINER. The USERs then take the best prediction made by a WRITER and act as normal components, making predictions about the external reward function based on this prediction.

Worth is passed from USER to DEFINER to WRITER, with only the most accurate WRITER currently submitting a prediction receiving worth from the DEFINER. This

means that if a USER cannot find a way to make use of a DEFINER's information it will not receive worth, and it, the DEFINER and all WRITERs assigned to that DESIGNER will be low worth and subject to deletion. It also means that inaccurate WRITERs will not receive worth. Since WRITERs still have scopes of prediction, like normal components, this again sets up the tradeoff between accuracy of prediction and number of prediction.

The WRITERs can then use the worth they receive to generate internal reward for the agent. This internal reward would direct the agent towards conditions under which the high-worth WRITERs can make useful predictions. It is in this manner that the agent learns to find information to solve problems, by slowly attributing more and more worth to WRITER-DEFINER-USER trios based on how much they help its overall predictive capabilities.

These components do not, in practice, have to be separate. A component could fulfil any of these roles at the same time, or indeed all three, if its internal structure were sufficiently complex. Since multiple WRITERs and USERs can be linked to one DEFINER this three-in-one component could simply be an initial component which is expanded on by later components assuming its roles, other than that of DEFINER, and thus improving the overall accuracy. In the following example, however, all three roles are served by separate components.

5.0.3 Hidden Variable Prediction Testing

In this test the agent was presented with a task requiring seeking out information to solve. The agent exists in a geometric environment topologically forming a tree. At the central node the agent is presented with 5 buttons. 4 of these will return a reward of -3, the remaining one will return 1. For the purposes of this first test, the correct button is only either the first or the second. The remaining three buttons will always result in -3. Alternatively, the agent can travel down the tree, by either choosing left or right.

The tree is 4 splits deep, each a left/right decision, so the final layer has 16 nodes. The environment therefore has 31 locations for the agent to visit. Each location is assigned a colour, red, green or blue, that is to say a 3 element vector. The agent's feature vector is its location in the environment, in terms of a 4 element vector, and a 3 element colour vector, for a 7 element feature vector. The agent can continue travelling down the tree from the start node by selecting either left or right, or can elect to return to the starting location. At the final level, 4 deep, it can only return.

Only one location in this environment is relevant to the agent. A single location at the penultimate level is chosen, that is to say accessible only if the agent performs the correct 3-action decision sequence. If the colour at that location is green the first button will return 1 and the second will yield -3, if it is blue the first button is -3 and the second is 1. The three remaining buttons are -3 regardless. Colours are randomly reassigned each time a button is pressed, so the agent must find the information again after each attempt.

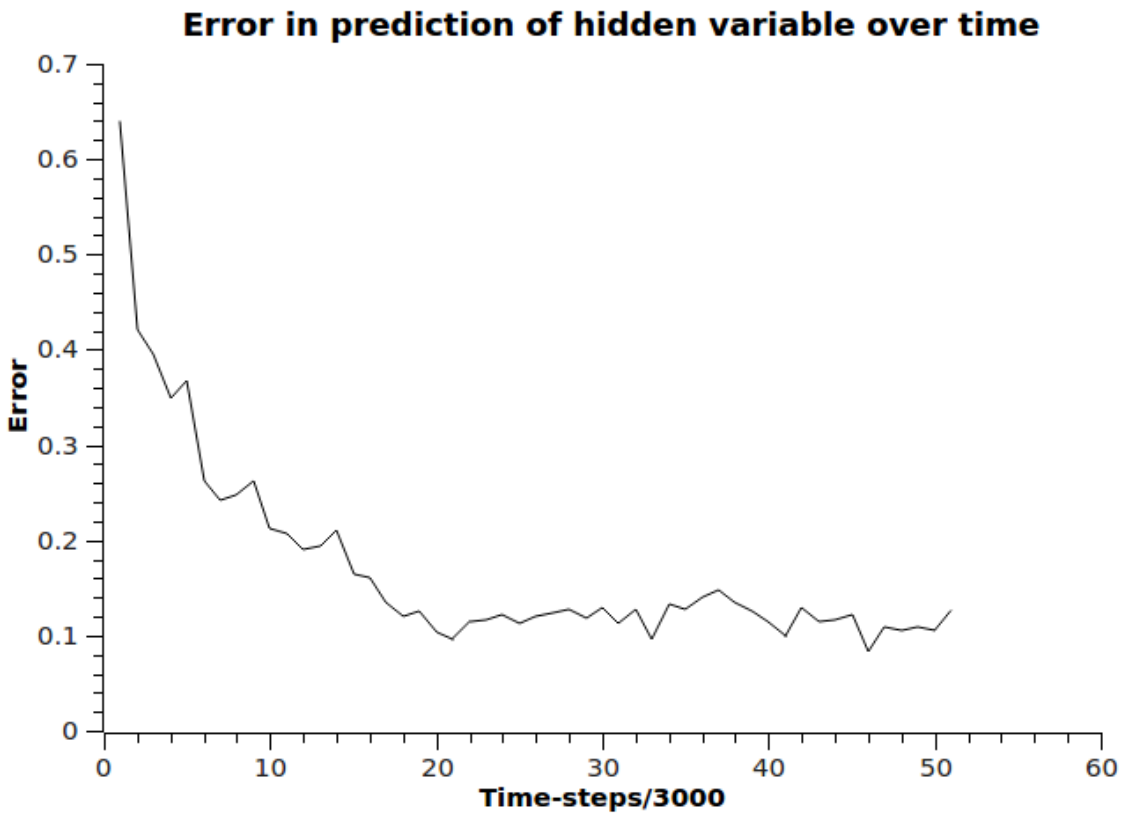

Figure 5.1: Graph of difference in predicted value of the hidden variable and its actual value.

This means that the agent has an expected reward of -2.2 per button press if it acts randomly. After a small number of trials, however, it should infer that the three last buttons are always negative, and only pick from the first two. This gives it an expected of -1. To achieve a positive reward it must learn to find the information.

The agent is given an automatic DEFINER, and cannot make any of its own. This DEFINER defines its phenomenon as "in the button location, press of button 1 will return a positive reward". WRITER components must then predict whether this assertion is true or not. Each of these WRITERs is given a scope of a single location, randomly chosen, from the tree. That is to say it will only attempt to predict the result of the button press if the agent is receiving information from its selected tree-location. Almost all of these tree locations are, of course, entirely useless noise.

If successful, only the WRITER at the correct location would receive any worth. Even that WRITER, however, would have to learn to connect the feature vector it receives, the colour of the location, to the prediction it needs to make. For this purpose, they were equipped with a simple regression tree learner which would output the expected probability of the DEFINER's phenomenon occurring.

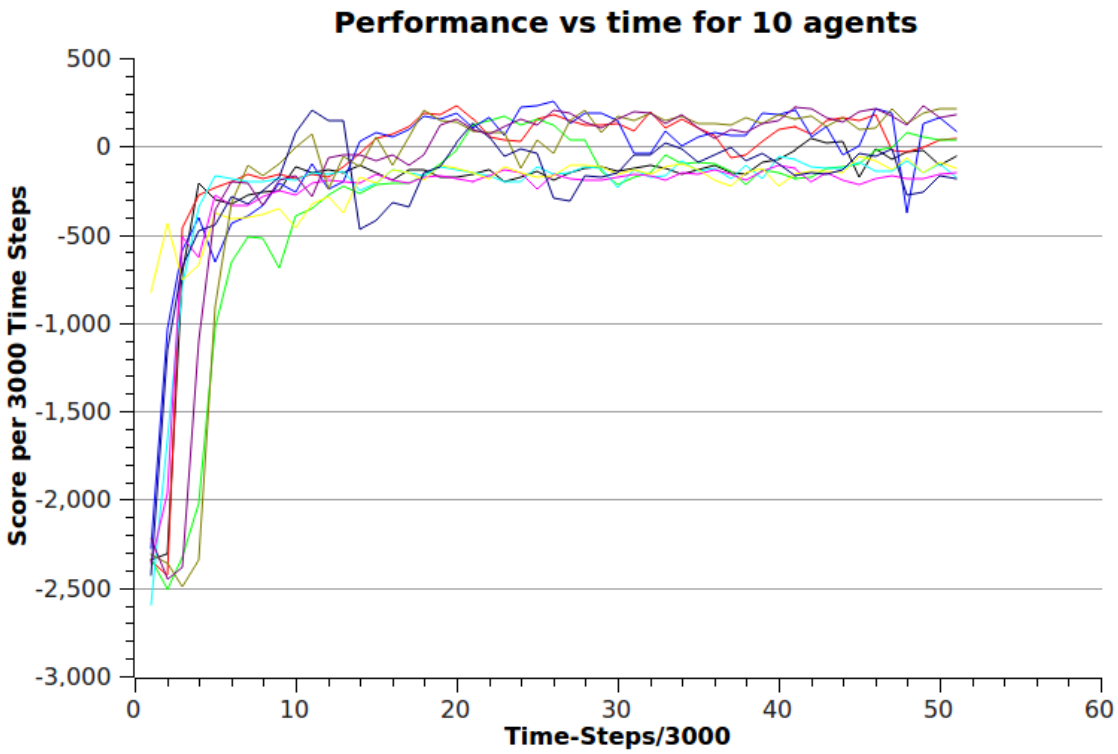

Figure 5.2: Graph of 10 agents' scores over time

The successful WRITER can then guide the agent to the information by emitting internal reward, which normal components will learn to predict. These normal will also take the role of USERs. They are also regression tree learners, which take the DEFINER's output, as well as the agent's feature vector, to attempt to predict either the external or internal reward the agent will receive from an action.

As can be seen in figures 5.1, 5.2 and 5.3 the agent proved successful in its task. 10 agents were run, each for 153,000 time-steps. For recording purposes, this was broken down into 51 episodes of 3000 time-steps, but this has no bearing on the agent's behaviour.

Figure 5.1 shows the agent was slowly able to learn to predict the value of the hidden variable. Since the phenomenon this hidden variable connects to is an evenly distributed boolean, the correct button is either the first or second, the baseline error is 0.5. This error is simply the average absolute difference between the true value of the hidden variable and the predicted value, not a mean squared error.

The error can be seen to quickly drop, then slowly stabilise at 20 episodes, or 60,000 time-steps. This is far above chance, and indicates the agents have indeed found the correct location in their environment. The non-immediate learning is due to the need to, after it has produced a WRITER able to generate the correct prediction as to the value of the hidden variable, learn the path to that component's chosen location. The continuing error is presumed to be due to the high exploration factor the agent was given, to promote

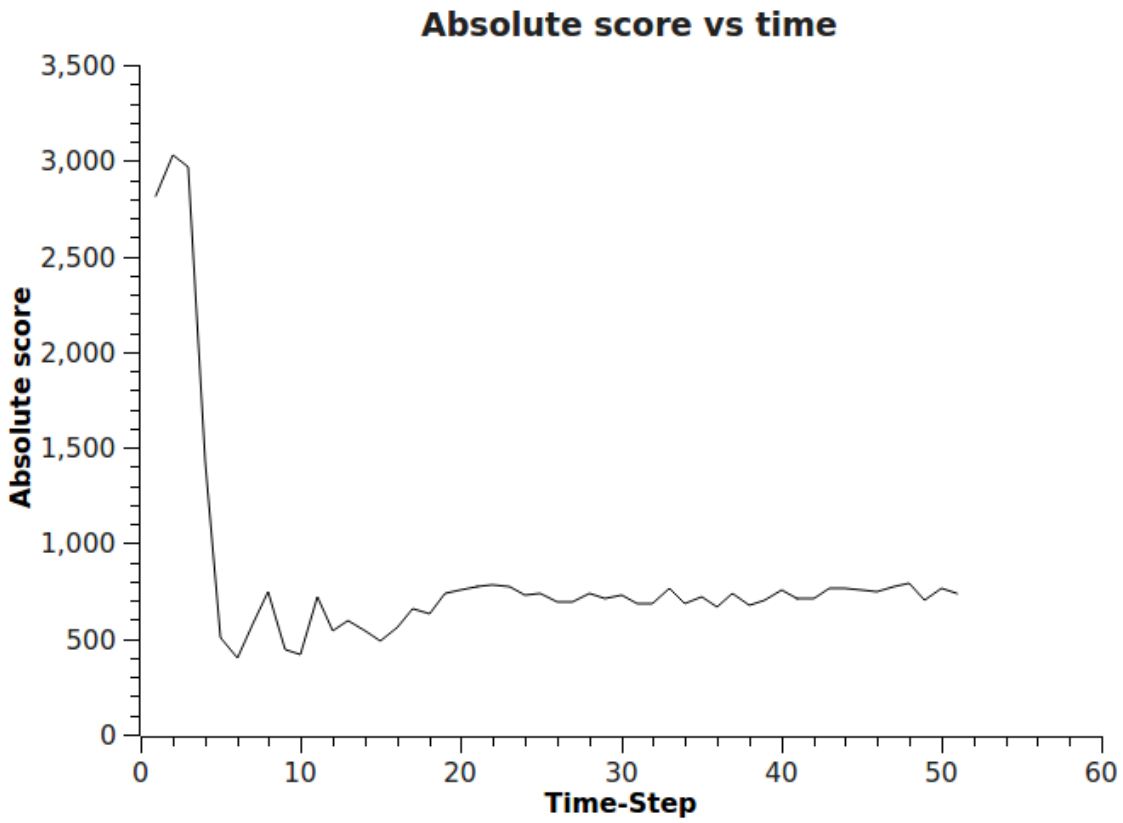

Figure 5.3: Graph of average of the 10 agents' accumulated absolute scores over time, indicating that near-zero scores were achieved by mixtures of negative and positive results, rather than simply by taking fewer actions

faster, but less optimal, reinforcement learning. This error is from sequences where the agent simply never reached the information-providing location, and a different WRITER had to make the prediction. This other WRITER would have no useful information, so would make a prediction of 0.5.

In terms of reward, the agents can be seen to achieve a positive or near positive average reward. Since the negative reward is -3 and the positive is 1, a positive score requires the agent to select the correct button more than 75% of the time. Some agents can be seen to have a positive reward, while some are below, presumably due to imprecisions in their learning.

A 0 score could be achieved by simply pressing no buttons ever, so Figure 5.3 plots the absolute score received, to check whether the agents are pressing buttons at all. As can be seen, the agents adopted a policy of far greater button pressing than they started with, since their initial policy was purely random and involved pressing the constantly-negative buttons routinely. However, it can also be seen that the agents slightly increased the frequency of button pressing following learning to find the correct solution.

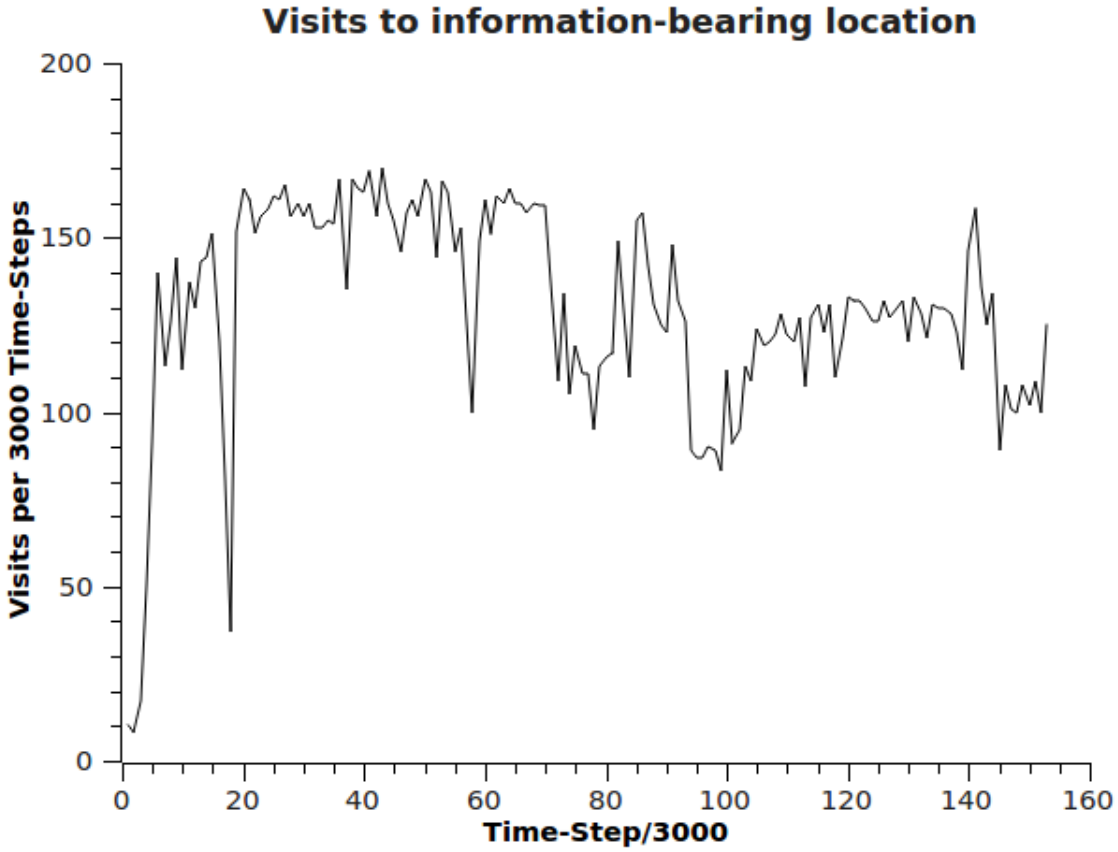

Figure 5.4: A single agent's visits to the information-bearing location, over time.

In terms of visits to the information-bearing location, analysis of a single agent run subsequently, simply to record this piece of information, revealed the rate of agent visits increased greatly. Average visits at the start of training were approximately 10 per 3000 time-steps, but rose to approximately 150. This is seen in Figure 5.4. While this agent was more unstable that the usual case in the previous 10 it seems sufficiently similar to assume the visit rates of the previous 10 increased in a roughly similar fashion.

In conclusion, it can be seen that this method has a degree of viability. While this test was simple, it demonstrated an agent architecture able to learn to find a piece of information amongst environmental noise, then follow behavioural sequences to acquire that information.

5.0.4 Hidden Variable Prediction vs Conditional Memory

Two approaches to information-sources learning have been outlined. On the one hand, the process could be entirely self-contained within a single component, which records if the state matches a given condition, then uses this information later. On the other hand, multiple components could interconnect, competitively attempting to find the best source of information to solve a given prediction problem.

Both use internal reward from components to guide the agent towards the information source, once these components have had slight success from random exploration, and gathered some worth. Both of them are able to discard useless components, which either record incorrectly or fail to improve the agent's predictive capabilities.

The first approach is condition based. It allows the agent to hypothesise that a given piece of information in the environment will be useful at some later time. It can be linked to another event, with a learner which attempts to directly use the information it holds, and/or it can supply this information to any other component which can make use of it. What it cannot do is learn this condition directly. The condition under which information is recorded must be meta-learnt by the agent.

The second approach is more active. The components are learning when to predict the value of this hidden variable, and what to predict, based on their feature inputs. It is also competitive, allowing slow improvements, as more specialist components take over from generalists.

An essential feature of the second approach, however, which has yet to be explored is that of retracting predictions. A component might decide, based on previous experience, that its prediction as to the value of the hidden variable is no longer likely to be accurate. If it wishes to preserve its low error, it could possibly learn to withdraw its prediction. It could either do this in response to errors the agent is making, implying it has faulty information; in response to some environmental cue, indicating that the agent's situation has changed; in response to some other hidden variable prediction being set, indicating contradiction; or simply in response to time passing, if the hidden variable is highly dynamic.

While the conditional memory components can be engineered to forget, the second architecture provides a natural framework for this additional capability to be added into the agent's overall feature set. It also allows learnt forgetting, in which components are able to connect their inputs to the probability of their stored data being incorrect.

5.0.5 Recorded properties of objects

The existing structure could be described as "Define a condition and a phenomenon, allow components to predict whether when that condition is next met that phenomenon will occur, distribute the most accurate prediction made to other components to make use of". It is a system attempting to store the value of a variable in the environment which the agent only has partial access to, allowing it to fetch the information from one location and use it in another. Currently, it can only represent information requiring no point of reference, zero-order logical predicates, rather than first or higher orders.

One extension which could be considered is to expand the WRITER $\rightarrow$ DEFINER $\rightarrow$ USER structure to include a NAMER. This NAMER would assign unique identifiers

to objects, allowing the agent to record properties of these objects discovered through investigation, then recall them at a later time.

This extension is considered less for its own sake, and more to demonstrate how the existing knowledge representation structure could be extended, by adding new roles which enable new types of representations, all basing themselves on the DEFINER core.

In essence, the previous solution can be though of as simply storing zero-order predicates about the environment. The agent can take actions to determine the truth value of P or Q, having hypothesised the existence of such a variable, and that it is responsible for causing phenomenon R under condition C. This augment is intended to allow the agent to store information in the form $P(x)$, and recall it at a later time. The difficulty is not from having an indexed data store, which is commonplace, the difficulty comes from needing to recognise said x repeatedly and consistently, and distinguish it from y and z.

A separate component is required for this storage/recall process due to the difficulty in connecting sensory data to physical objects, especially when those objects are lost from view for a time. In many environments the objects the agent will encounter will change in appearance over time. This therefore requires the agent to have some ability to learn which properties can be used to distinguish and recognise objects, and which change too rapidly or too inconsistently to be used.

The NAMER therefore would act as a mechanism to assign identities to sensed objects, based on some component-specific approach. WRITERs would each adopt a NAMER (initially chosen at random), and this WRITER-NAMER pair would then attempt to make predictions as to the results of the DEFINER's phenomenon. Each prediction would only be tested if the NAMER indicated it was the same object. This means that different NAMERs would recognise different objects as "being the same", and assign them the same unique ID. This divergence would occur both at write and a read time for that given variable.

Since the DEFINER is attempting to find the most accurate WRITER, and any WRITER has chosen a single NAMER, the NAMER's accuracy is implicitly determined by the DEFINER, but the DEFINER does not need to measure this accuracy directly. WRITERs which have been assigned a useful NAMER will succeed, and produce useful and low-error predictions, those which have not will not.

NAMERs are distinct from WRITERs due to the assumption that they would fail at a much higher rate. They cannot learn which properties of an object to look for, under the current architecture, so must be randomly constructed. Eventually the agent might meta-learn to produce more useful NAMERs, and it is possible that a learning structure could be developed to maximise prediction quantity and accuracy, but this is beyond the scope of this work.

If useful NAMERs are rare, it makes sense to separate them, so that multiple WRITERs can be linked to a single successful NAMER. They would pay worth to this NAMER, in return for its outputs, and so it would be easy to determine which NAMERs were successful and which were not after these NAMERs had existed from a number of time-steps.

6 Future Work

While the SAMLA architecture has been shown to produce interesting results, and meets the initial aims of being a novel reinforcement learner capable of meta-learning, various possibilities for future investigation present themselves. Many of these lines of investigation revolve around exploiting the component-worth metric as an internal drive.

6.1 Evolving components

In this work all component internal structures have been provided to the agent by the designer. This includes but is not limited to which machine learning algorithms to use, how many information sources to draw from and which pieces of information to output. While sufficient for these tests, for a much larger environment, requiring a far broader range of capabilities which may be unknown to the designer, greater flexibility is required.

Components are currently selected from a set of archetypes by the agent, then given specific parameters and targets, such as, in some tests, being primed with a given feature vector to use as the centre of their scope. These archetypes are defined by the designer, the specific implementations, which are the components themselves, are defined by the agent. As such, each component has an archetype it belongs to, a blueprint of its internal architecture.

The components' worth to the agent is already available, since it is the worth used to decide whether or not to remove a given component from the agent's component pool. This worth can be fed back to the archetype used to create the component, which can then have its own worth computed. This archetype's worth would be a function of the worths of all components which are created based on the architecture it defines.

After a designer-defined number of components have been created, all archetypes available to the agent will have been attributed a worth, based on which the agent has learnt to be most useful, in terms of the use of the components they can create. This can be used as a fitness function, and standard genetic evolution occur. Archetypes could be crossbred and mutated, forming a subtly new set of archetypes which the agent can use. These would, in principle, lead to the creation of better components, more able to predict the agent's environment. Such a process could repeat indefinitely, allowing the agent to slowly adapt the internal structures of the components it creates.

6.2 Self-Structured Learning

An interesting possibility for the agent would be to allow it to learn to predict which types of capability would be useful in the future, then act to maximise its capabilities beforehand, rather than reactively. Since components form the capabilities of the agent, and these are given a numeric worth, and the agent already predicts the worth of the component at the time the component is created, it is already predicting the expected worth of new capabilities. This prediction could be expanded to increase the agent's capabilities.

6.2.1 Curiosity: internal reward for creation of components

In Oudeyer et al's work [16] an agent consisting of a set of 'experts', each able to predict a portion of the environment was shown to be able to learn to move towards learnable parts of the environment. Unpredictable areas, and areas which had already been learnt were avoided, in favour of areas which allowed the generation of new experts. While not identical to the agent presented in this work the similarities are sufficient to suggest such an approach might work with this work's agent.

Since new components are only given worth if they improve the agent's predictive capabilities, with duplicates and useless components being given low or zero worth, the addition of new high worth components can be assumed to be linked to improvements in predictive ability. An internal reward signal could be created, corresponding to the expected worth of any component made during that time-step. In almost all time-steps this would be zero, as components are only created occasionally, but when non-zero it would be a function of the agent's expected improvement in its own predictive capabilities.

If the agent were to learn to maximise both the internal and external reward signals it could learn to seek out areas which would lead to the creation of new high-worth components. Such areas would not be those it had already explored and generated components to predict, nor those which it could not successfully form components for. The agent would therefore have a preference for tasks which would lead to it gaining new predictive abilities, and could use reinforcement learning techniques to maximise its own learning.

6.2.2 Alternative signals for prediction

A more complex agent, in terms of component structures and component interconnections, might benefit from having components attempting to predict more than simply the reward signal. In this work so far the only learning which is performed by component is attempting to predict the external reward signal for the part of the environment which their scope covers.

If a component were generated which attempted to predict an environmental cue, rather than the external reward signal, it could provide useful information to other components. It would serve as an information supplier, and receive worth from other components using

its outputs to improve their own predictions. An example might be a next-time-step predictor, which attempts to predict the sensory feature vector the agent will receive in the next time-step, based on the current one.

This could be useful in two ways. The first is that this approach may be a way to break down tasks too complex for a single component's learning algorithm to handle. It could either serve as a scope limiter, reducing the task size, or another information source. In either case, another component could benefit from being faced with an easier to solve task.

The second use for this type of structure might be in times of low external reward, or times when the agent is already predicting the environment as well as it can. In such circumstances these components which predict other environmental effects could serve as a way for the agent to continue learning, despite there being no way to improve its prediction of the external reward signal in the current state of the environment. When the environment changes, these components could have already performed useful learning, and serve as a ready-made foundation for the next generation of components.

The immense number of possible predictions the agent could make, considering it could attempt to predict any part of the environment to any number of time-steps in the future, makes meta-learning required to maximise the utility of these components. The agent must build, through trial-and-error, a rough picture of which environmental cues will be useful to predict, and on which time-scales. It would be attempting to learn which predictive capabilities would serve it best in the future, as suppliers to future components.

Such components could possibly benefit from self-motivated learning, if the agent were to seek environment cause-effect links to learn at times when the environment contained no obtainable external reward. This would allow an agent which could autonomously continue learning while it had no designer set tasks.

It is important to note that this learning is entirely grounded in the external reward signal provided by the designer. While these components do not themselves interact with it they only gain worth if they supply useful information to components which do. Without this there would be no preference from the agent's perspective to learn one environmental cue over another, which would be problematic in complex and noisy environments.

6.2.3 Self-directed Reinforcement Learning

Another meta-learning dependent autonomous learning would be self-directed reinforcement learning. This would be a process in which the agent selects a goal state, defined by a component, then attempts to reach it from various parts of the environment. The component would need to return a boolean, which can re-use the normal scope setting boolean functions which previously investigated components use. This would then serve as co-ordinator for the agent's learning, by registering itself into a list of possible non-immediately-rewarding RL goals.

Other components could then make predictions defined by one of these RL goals. Rather than predict external reward following actions, it would predict the probability of reaching its chosen goal following certain actions. They would develop in a similar way to already-seen RL prediction structures, as demonstrated in this work's testing.

This gives the agent a probability for every action of reach a given RL goal. It can then multiply this probability by the expected value of the goal, and add this into its weighting for each action. Initially, the value of a goal is simply some user-defined 'curiosity' value, which directs the agent towards such RL goals if no other external reward is available. As such, the agent could train itself to perform multi-action sequences.

These RL goals would serve two purposes. The first is in dynamic environments where some objectives could begin to deliver external reward where previously they did not. If the agent has already learnt a path to these objectives it can simply alter its expected value for the RL goal and multiply that with its existing probability mapping. This would enable it to quickly access the goal, without needing to learn it again.

The second is if this goal could serve as a useful starting point for a second RL action sequence. If the state the agent arrives in can be found to lead to some external reward, this could be extended to the goal itself. When at the self-defined goal the agent is provided by its components as to the expected future reward of actions, by normal RL mechanisms. These predictions can be seen as the value of the goal, since it provides access to these future rewards. By already learning the sub-goal the agent has a shorter action sequence to learn to reach its ultimate goal, reducing learning times.

Again, this self-directed learning is highly dependent on meta-learning to be successful. Any part of the agent's environment could be set as a self-directed RL goal, but most of these locations have no use, so learning to reach them would be pointless. The self-set RL goals would need to be attributed worth based on their future usefulness, either as sub-goals or if the environment alters to give them value. The agent would then need to learn to predict this worth, and to only set sub-goals when it had a high enough degree of expected future worth.

6.2.4 Conditional Memory Components as Information Utility Measures

One approach to self-directed reinforcement learning would be to have a conditional memory structure in some of the components. They would record from the environment if and only if their condition was met. What this condition would be depends on the particular components, but they could, for example, only record in a given location, or in the presence of a given object, or even if another component makes a prediction above a given threshold. Once information has been recorded, they would either wait for a second condition, at which point they would make a prediction, or supply the information to other components to use. The conditions these components record data under would constitute

the "goal states" the agent could learn to act towards. Since the components are given numeric worth based on usefulness, this could be translated into a reinforcement learning drive the agent could act to maximise, thus maximising the effectiveness of its most useful components.

This approach is somewhat similar to the neural Long-Short-Term Memory [18] architecture, in that both record at given points, based on a condition, then allow others to read from their memory store. Unlike the LSTM networks, these conditions would be set by the agent, when it creates the memory components, rather than being learnt. Clearly this strongly advantages the LSTM architecture initially, until this work's agent is able to meta-learn which conditions to look for. Once it has completed this meta-learning, however, assuming it is able to find patterns in the environment allowing it to do so, it would be able to place memory components specifically targeted towards features in the environment which represent non-immediately useful information.

An easy example would be to have a location in the environment, say the top of the area the agent finds itself in, which has a useful feature input. The bottom of the area has a set of buttons the agent can press, one of which yields a positive reward, the rest of which yield a negative. The information from the top of the grid is the only way to deduce which button will return positively, so the agent must travel upwards, to acquire the information, then back down to use it.

A component could be generated which records only if the location matches the top of the grid, but only makes predictions when the location matches the bottom. Many components could be generated, all with different locations for both recording and predicting, but only one which is set to record from the top of the grid will have any useful information regarding the decision the information faces. These locations are set by the agent, initially randomly, although in principle it should be able to meta-learn, given experience.

The result would be only the useful component gaining worth. If the recording location is incorrect the component has no way of predicting better than chance, while if the prediction location, the scope of the component, is set incorrectly the component will have nothing useful to predict. As it stands, however, the agent wanders blindly if there is no clear path to reward. Since the agent would have no incentive to press a button, since only one yields positively and the rest negatively, and no other rewards exist on the grid, it would simply take random movement actions.

A solution would be to have the components generate an internal reward signal the agent could seek. The user-defined external reward, received from the environment, would be the primary driver of the agent, but a lower-strength signal from the components themselves would be added in. This signal would be a function of the worth of the component, which would pay out a portion of its worth to affect the agent's behaviour.

In this example the component would generate an internal reward signal when it records information. In a sense it would be 'thanking' the agent for taking actions which advantaged this component. The component would gain from being in a location with an information source, as it can later use this information to make predictions, and thus gain worth. The agent had no other path to external reward, so followed this source of internal reward. The agent then gains, as its components are now better able to predict which button to use, and allow it to gain a positive reward after moving back to the bottom of the environment.

The goal of this addition to the agent is to allow short action sequences to be performed by the agent to gain information useful to the problem at hand. Since the condition which the component uses to determine whether to record or not is simple it cannot be used for highly complex or subtle problems. The condition is necessarily simple, as it must be created by the agent through parameter selection, in contrast to the LSTM model which learns its own condition. However, it could prove a fairly useful tool, in environments where information is close at hand but not directly present.

Also of note is that the information recorded by the component does not need to be limited to a single numeric value, of course. It can store the entire input vector that particular component was receiving during that time frame. This could include both the feature vector the agent was receiving as well as the outputs of the components it was connected to. It could then use machine learning to link this information to the results it is trying to predict. The conditions under which it records and makes predictions are fixed, but what it does with that information can be learnt.

Anecdotal success This structure has been tested very briefly, and indicates possibly promising results. Randomly generated components were created, and only one which had its recording condition and prediction conditions set usefully was given worth (subsequent usefully configured components were redundant, so not given worth). Internal reward was then generated by this component, and the agent was able to cycle between information and decisions, providing much better performance than expected by pure chance.

It is important to note, however, that this test was far from rigorous, and has not been repeated or studied sufficiently to draw any reliable conclusions from.

6.2.5 Overview of agent with planned future features

With both the currently implemented and proposed future features, the agent represents what this work is aiming for, an autonomous, self expanding learner.

The agent's learning consists of two main parts. Firstly is its representation of the world. This is its knowledge, the data it has acquired from the environment and stored in

either a component's own memory system or the shared memory created by the DEFINER-WRITER structures. The second part is its capabilities, the abilities it possesses to transform this representation into predictions about the effects of actions, in terms of their total future effect on the utility function.

The agent is self-expanding in that both its capabilities and representations can continually be improved, by iterative addition and testing of new components. The agent is autonomous as it is able to self-motivate to improve both its representation and its capabilities.

Capabilities Predictive capabilities are component based. They exist within the agent as the set of prediction-making components. Improvements to its capabilities come from improvements to this set. The first mechanism for this is improving coverage, creating sufficient components to make predictions for all actions at all times. The second mechanism is replacing these components which new ones with better suited algorithms, allowing more accurate learning using the same data. The third is stacking components, allowing them to combine their capabilities. The fourth and last mechanism is to specialise, to reduce the scope which newly created components take on in order to improve their predictive accuracy.

This incremental improvement has been partially demonstrated by previous tests, and evolved specialisation is hoped to increased it further still. By allowing components to randomise their specialisation strategies, those used to reduce their predictive scope, components in general will slowly subdivide the environment in an increasingly optimal fashion.

Self directed capability learning takes the form of a reward signal for creating high expected worth components, as seen in the section 6.2.1 on component worth as curiosity. This is a slow process in which the agent takes actions on its environment to place itself in situations where new, useful, components can be created. It is learnt, in that the agent learns through experience to predict the future worth of components, and will not be motivated to create components it does not expect to be useful.

Representations Representations are stored in two ways. The first is in the components themselves. Any component might have a short memory system, say recalling the last few input time-steps, or conditionally recording individual feature vectors. This mechanism is highly heterogeneous, with different component architectures storing different pieces of information in different ways.

This internal-to-components structure is an unstructured way of storing large amounts of information for short time periods, based on what proves useful. Components which store useful data, and possibly then share it with others, are kept, while those which do not are not. In this way the representation slowly grows, as new components are accepted,

and as the agent meta-learns which types of memory component will be accepted into the overall component set.

This representation structure's self motivation comes from components producing internal reward as seen in the section 6.2.4, on conditional memory components. This allows the agent to autonomously take actions to update the information stored within components, based on learnt behaviours combined with discovered component-worth.

The other representation structure is the external-to-components mechanism described in section 5.0.2, created by a DEFINER-WRITER setup, and used by an arbitrarily large number of USERs. The self-motivation mechanism for this representation is as described and tested above, and again is learnt based on the usefulness of the information being gathered. This usefulness is measured based on the components using the information, and ultimately requires the information to have some part in making useful predictions about the agent's user-defined utility function.

Evidently this structure would need expansion, with one of the many forms of expansion required being explored in section 5.0.5. The agent's full world-representational capabilities depend on the flexibility of the external-to-components representational structure. Since all major long term information storage is expected to take place within this structure it must be capable of representing as much of the world as possible in as simple a form as possible. Presumably this would include expansion into sets, ordered lists, maps and attribute-bearing links between objects to expand on the simplistic zero-order and first-order predicates already described.

Component generation motivation Since both of these representational structures are defined by components the agent benefits from the already-discussed component-creation motivation. The agent requires motivation both to seek and update the information stored within its knowledge representation structures, as well as the motivation to expand these structures themselves. This is, however, covered by the drive to create high-worth components, so does not need to be handled separately.

The complete agent As such, with these proposed improvements the agent would be capable of slow, progressive expansion of both its knowledge representation and its predictive capabilities. It is also capable of autonomously seeking to improve these, both by expanding its predictive and representational capabilities and actively fetching information from its environment in a first location to solve a problem in a second.

With these features the agent could be considered complete, as it would have self-improvement autonomy across all necessary functions. The agent could then be relied on to slowly self-expand its component set in a way which would maximise its predictive capabilities, and therefore its ability to maximise its utility function.

7 Conclusion

This thesis has investigated a number of aspects of the SAMLA architecture.

The first aim of this agent was to be an investigation into meta-learning in the field of reinforcement learning. To this end, a series of tests were performed, assessing the agent's meta-learning capabilities across a range of tasks, each task showing a degree of success.

The second aim was to provide a framework for later work. A large section on the future direction of the agent describe a realistic framework for expansion, all aimed at creating a powerful autonomous agent. Various features were discussed, including some initial testing, all aimed at covering a distinct required area for autonomous self-motivated learning and information discovery.

The first aim therefore forms a foundation for the second. The agent had to be shown to be able to perform self-addition in a useful, reliable way. These additions needed to be evaluated correctly, operate together without conflict, subdivide the environment to balance predictive accuracy and coverage, and removed when necessary without degrading the performance of the remaining components.

These capabilities were shown to be functional. The agent was shown to be able to learn to select the highest accuracy algorithm from those available to it, allowing it to increase its performance over sequential tasks, by meta-learning from the successes and failures in the previous. This in turn indicates that the component-worth evaluation provides useful feedback to the agent, allowing better components to be produced.

This component-worth metric was also shown to be useful in subdividing the environment, splitting the predictive task between multiple components, replacing generalists with specialists as these specialists become available, without losing access to the generalists in situations where no specialist can be found.

The component architecture was also shown to be able to form structures from multiple components, and their information sharing was shown to be able to grant them capabilities beyond those possible for a single component in isolation.

While some alterations were made to the component types in use in the various tests, the overall architecture of the agent remained unchanged for all these tests. This indicates the architecture itself has a broad range of capabilities, and seems a reliable foundation for

future testing and expansion, either in terms of new component structures or new features built upon the existing architecture.

The second aim uses the component-worth metric, which has been shown to be a functional metric for evaluating components, as a metric for evaluating information and capabilities. An autonomous agent must learn to take actions to maximise its capabilities in much the same way as it must learn to take actions to maximise its reward function. The proposed additions split the task between representational structure, the model the agent uses to describe its environment; the contents of this representational structure; and the ability to use this represented information in a way which leads to better actions being taken, in the context of the designer-supplied utility function.

Each of these provides an internal drive for the agent, leading it to learn how to improve itself, without requiring specific teaching. If this can be implemented successfully, the agent should have 'complete' self-improvement autonomy, that is to say a drive to improve all required aspects of the information-gathering task at hand.

The ability of the architecture to perform a wide range of tasks, including multiple types of meta-learning, with minimal restructuring indicates that it could be possible to create a singular agent able to succeed at all these tasks. Such an agent was not implemented in this thesis due to time constraints, and lack of an environment able to train and test all of these capabilities simultaneously. Such an agent should be possible, however, simply by broadening the range of components available to the agent at any one time, as well as incorporating all desired features into these components, such as information-source selection, parameter selection and algorithm portfolio selection.

In general this work can be considered a successful foundation for future work, as well as a broad overall summary of the architecture's known current abilities.

Bibliography

[1] Michael L. Anderson and Tim Oates. A review of recent research in metareasoning and metalearning. *AI Magazine*, 28(1):12, 2007.

[2] Michael L. Anderson, Matthew D. Schmill, Tim Oates, Donald Perlis, Darsana P. Josyula, Dean Wright, and Shomir Wilson. Toward domain-neutral human-level metacognition. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, pages 1–6, 2007.

[3] Eric B. Baum. Toward a model of intelligence as an economy of agents. *Machine Learning*, 35(2):155–185, 1999.

[4] Jonathan Baxter. A model of inductive bias learning. *J. Artif. Intell. Res.(JAIR)*, 12:149–198, 2000.

[5] Pavel B. Brazdil, Carlos Soares, and Joaquim Pinto Da Costa. Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[6] Deborah R. Carvalho and Alex A. Freitas. Evaluating six candidate solutions for the small-disjunct problem and choosing the best solution via meta-learning. *Artificial Intelligence Review*, 24(1):61–98, 2005.

[7] Philip K. Chan and Salvatore J. Stolfo. A comparative evaluation of voting and meta-learning on partitioned data. In *ICML*, pages 90–98, 1995.

[8] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54(3):255–273, 2004.

[9] Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3):295–328, 2007.

[10] Haipeng Guo and William H. Hsu. A machine learning approach to algorithm selection for np-hard optimization problems: a case study on the MPE problem. *Annals of Operations Research*, 156(1):61–82, 2007.

[11] J. H Holland. Properties of the bucket brigade. *Proceedings of an International Conference on Genetic Algorithms. Hillsdale, NJ*, 1985.

[12] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas. Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review*, 26(3):159–190, 2006.

[13] Geoffrey Holmes Bernhard Pfahringer Peter Reutemann Ian H. Witten Mark Hall, Eibe Frank. The weka data mining software: An update. *SIGKDD Explorations*, pages Volume 11, Issue 1, 2009.

[14] Matthew Michelson and Sofus A. Macskassy. Judging the performance of cascading models: A first look. *Fetch Technologies*, Volume 841, 2009.

[15] Eric Nivel, Kristinn R. Thórisson, Bas R. Steunebrink, Haris Dindo, Giovanni Pezzulo, M. Rodriguez, C. Hernandez, Dimitri Ognibene, Jürgen Schmidhuber, Ricardo Sanz, and others. Bounded recursive self-improvement. *arXiv preprint arXiv:1312.6764*, 2013.

[16] Pierre-Yves Oudeyer, Frédéric Kaplan, Verena V. Hafner, and Andrew Whyte. The playground experiment: Task-independent development of a curious robot. In *Proceedings of the AAAI Spring Symposium on Developmental Robotics*, pages 42–47. Stanford, California, 2005.

[17] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. Tell me who can learn you and i can tell you who you are: Landmarking various learning algorithms. In *Proceedings of the 17th International Conference on Machine Learning*, pages 743–750, 2000.

[18] J Schmidhuber S Hochreiter. Long short-term memory. *Neural Computation*, pages Volume 9, Issue 8, 1997.

[19] Mostafa A. Salama, Aboul Ella Hassanien, and Kenneth Revett. Employment of neural network and rough set in meta-learning. *Memetic Computing*, 5(3):165–177, 2013.

[20] Eugene Santos Jr., Alex Kilpatrick, Hien Nguyen, Qi Gu, Andy Grooms, and Chris Poulin. Flexible algorithm selection framework for large scale metalearning. pages 496–503. IEEE, 2012.

[21] Robert E. Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[22] Jürgen Schmidhuber. Ultimate cognition à la gödel. *Cognitive Computation*, 1(2):177–193, 2009.

[23] Jürgen Schmidhuber. A formal theory of creativity to model the creation of art. In *Computers and Creativity*, pages 323–337. Springer, 2012.

[24] Carlos Soares, Pavel B. Brazdil, and Petr Kuba. A meta-learning method to select the kernel width in support vector regression. *Machine Learning*, 54(3):195–209, 2004.

[25] Bas R. Steunebrink, Jan Koutník, Kristinn R. Thórisson, Eric Nivel, and Jürgen Schmidhuber. Resource-bounded machines are motivated to be effective, efficient, and curious. In *International Conference on Artificial General Intelligence*, pages 119–129. Springer, 2013.

[26] Bas R. Steunebrink, Kristinn R. Thórisson, and Jürgen Schmidhuber. Growing recursive self-improvers. In *International Conference on Artificial General Intelligence*, pages 129–139. Springer, 2016.

[27] Kristinn R. Thórisson, Nivel Nivel, Bas R. Steunebrink, Helgi P. Helgason, Giovanni Pezzulo, Ricardo Sanz Bravo, Jürgen Schmidhuber, Haris Dindo, Manuel Rodriguez Hernandez, Antonio Chella, and others. Autonomous acquisition of natural situated communication. *IADIS International Journal on Computer Science And Information Systems*, 9(2):115–131, 2014.

[28] Ljupčo Todorovski and Sašo Džeroski. Combining classifiers with meta decision trees. *Machine Learning*, 50(3):223–249, 2003.

[29] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[30] Ricardo Vilalta, Christophe Giraud-Carrier, and Pavel Brazdil. Meta-learning. *Data Mining and Knowledge Discovery Handbook*, pages 731–748, 2005.

[31] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[32] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.

[33] Y Bengio Y Lecun, L Bottou. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume Volume 86 Issue 11, pages 2278–2324. IEEE, 1998.