Reusing 100% of the Brain: Neuro-reuse for Self-* Planning

anonymous authors

***Abstract*—Software systems are increasingly leveraging autonomy to manage uncertainty in their operating environments. Cloud-based service providers, autonomous vehicles, and electrical grids are examples of systems that need to adapt to changes in their environments without human intervention, and these systems are only growing larger, more interconnected, and more complex. To respond to this growing complexity, many autonomous systems use artificial neural networks to determine how to adapt. While ANNs allow for complex system behavior and performance improvements, they are often treated as black boxes with minimal ability to generalize across tasks. As a result, the planning component of any system can be rendered useless when unexpected changes occur at run-time, causing system failures, training delays, or the necessity of costly human input.**

In this paper we describe, implement, and evaluate *neuro-reuse* a new method for autonomously dealing with unexpected changes in self-* systems. Using NEAT, an existing genetic algorithm for evolving neural networks, we show that neuro-reuse improves the final utility of our exemplar self-* system compared to retraining from scratch in 65% of trials. Additionally, we show that neuro-reuse can outperform retraining from scratch even in complex change scenarios, enabling the large scale autonomous systems of the future to leverage prior knowledge to adapt in the face of unexpected changes.

***Index Terms*—plan reuse, neural networks, neuro-evolution, self-* systems, planning, uncertainty, transfer learning**

I. Introduction

In the week between March 12th and March 19th, Microsoft teams—a communication platform for workplaces—went from 32 to 44 million active daily users [1]. This anomaly in usage was the result of a mass transition to work-from-home at the start of the coronavirus pandemic, a degree of change few systems accounted for. In response to the influx of requests, on March 16th servers went down for hours all over Europe—wreaking havoc on the teams who rely on the platform.

The massive surge in traffic caused by increased work-from-home is just one example of how unexpected changes can disturb autonomous software systems, and as such systems become larger and more complex, their failures will only become more catastrophic. To respond to such changes effectively, and without human intervention, self-adaptive, self-healing, self-managing, or self-* systems [2] must be able to autonomously adapt to unexpected or unprecedented changes.

Typically, the response to these changes is handled by a *planning* component [2], which decides how the system should adapt. There are many approaches for determining a strategy, including *offline planning* approaches such as rule-based plans, which may be human specified [3] or generated by techniques like probabilistic model checking [4], and *online planning*, which autonomously generates plans at run-time. Some examples include model-based approaches [5], stochastic search [6], Bayesian optimization [7], and reinforcement learning [8].

Within the realm of online planning, autonomous planners are increasingly using neural network based controllers (neuro-controllers) to decide on an adaptation [9], [10]. Artificial Neural Networks (ANNs) are well-suited for this task since they can allow self-* systems to learn highly complex behaviors efficiently, and have been shown to outperform humans in difficult tasks like chess [11] and poker [12].

The cost of this sophistication, however, is often an inability to generalize across tasks. Classically, neural networks are trained for one specific task and then utilized on data from the same feature space. This means that unexpected changes in the system's operating environment can render their output ineffective [13]. Understanding why a network outputs one particular value is equally difficult, which leads many system designers to treat their underlying ANN as a black-box [14]. This is unfortunate, because knowledge reuse is a promising avenue for enabling systems to more effectively adapt following an unexpected change for self-* systems that use rule-based planning languages rather than ANNs [3], [6], [15].

The subset of machine learning known as *Transfer Learning* responds to this problem by exploiting information about one task to improve performance on another related task. While transfer learning has shown success in areas like image classification, game learning, and sentiment analysis [16], [17], it has not yet been investigated as a method for responding to unexpected changes in self-* systems.

In this context, we seek to transfer information across plans so that self-* systems can benefit from the functionality of ANNs, while mitigating their fragility to unexpected changes. Our proposed technique, *neuro-reuse*, makes this possible by re-purposing prior neuro-controllers to automatically adapt to new operating conditions, like a change in traffic patterns following increased work-from-home.

By exploiting information contained in the original controlling neural network, neuro-reuse evolves plans towards a new solution. Rather than starting from scratch after an unexpected change, or re-planning with human input, *neuro-reuse* utilizes the evolutionary algorithm NEAT [18] (NeuroEvolution of augmenting topologies) to adapt a pre-existing neuro-controller to a new scenario. Neuroevolution combines evolutionary computation with deep learning to incrementally evolve neural networks. This can enable self-* systems to benefit from the functionality and run-time improvements of planning with neural networks by reducing the time and cost

of re-training in the face of uncertainty.

In this paper, we present a novel approach for adapting to unexpected changes in neuro-controlled self-* systems by evolving neural networks that have been rendered ineffective as a result of an unexpected change. Existing neuro-controllers are reused by seeding the population of a neuroevolutionary algorithm, allowing more effective retraining following unexpected changes. We implement an approach for reusing neuro-controllers for self-* systems using the NEAT algorithm [18]. This contribution is novel in demonstrating (a) how to re-use neuro-controllers for the adaptive systems context, (b) how effective transfer is in comparison to simply re-planning from scratch, and (c) what factors influence its effectiveness. To address these items, we evaluate the approach on a simulated cloud-based web service provider inspired by Amazon Web Services. We then compare the utility during evolution for neuro-reuse against re-planning from scratch during training and testing. We show that neuro-reuse converges to a higher final utility than retraining from scratch 65% of the time, and has a higher utility during the first 350 generations of evolution 93% of the time. We further investigate how neuro-reuse performs when seed networks are overfit to training data, and we investigate the effectiveness of reuse in response to varying degrees of unexpected changes, showing that reuse can still be effective for even large changes. Our source code, data, and analysis scripts are available for review and replication [1].

We make the following contributions:

- An approach for reusing existing neuro-controllers, allowing self-* systems to more effectively re-train following an unexpected change.
- An empirical study based on a simulated cloud-based service provider inspired by AWS, showing that the proposed approach results in improved re-training following unexpected change scenarios in 65% of trials.
- An investigation into how training data quality and the degree of unexpected change influence the effectiveness of neuro-reuse, finding that reuse is not more susceptible to overfitting than training from scratch, and can also result in improvements even for large unexpected changes.

The remainder of the paper is organized as follows, Section II describes the necessary background, including self-* systems and neuroevolution. Section III describes the case study self-* system that we use as a running example and in evaluation. Section IV describes the proposed approach. Section V presents the experimental setup used in the evaluation of our approach. Section VI provides the results. Section VII discusses limitations and directions for future work. Section VIII compares this paper to related work. Lastly, Section IX provides a concludes.

II. Background

Here we provide the background on self-* systems, planning, Artificial Neural Networks, and NEAT necessary necessary for understanding our contributions.

[1]https://bit.ly/2EAtXVK

A. Self-* Planners

Autonomic computing refers to systems that are able to make decisions without human intervention based on a set of high level objectives. Ideally, "autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures" [2]. At its core, this requires a system that can govern and maintain itself by tracking, analyzing, and responding to changing parameters in its environment.

IBM formalizes these ideals by defining four traits that self-*, or autonomic [2], systems must contain: self-configuration, self-optimization, self-healing, and self-protection. Self-healing refers broadly to automatically responding to bugs and/or failure conditions, and self-optimization to a system that can continually monitor its own performance and make optimizing adjustments.

Architecturally, self-* systems often respond to these demands by dividing their functions into *managed* and *managing* subsystems with the MAPE-K architecture. Within the managing component, there is a subsystem called a *planner* responsible for determining how that system should adapt based on changes in its operating environment. Strategies are composed of tactics, which consist of individual actions. In a cloud service provider, these architectural changes might be "start a server at location A" or "decrease traffic on server B".

Planning can be done offline, such as with adaptation strategies manually written in a planning language like Stitch [3], or generated automatically online. Many approaches for this problem have been proposed, including approaches based on Markov Decision Processes [5], Bayesian optimization [7], reinforcement learning [8], and evolutionary approaches [6], [15], [19], [20]. Knowledge reuse has been proposed as a way to allow planners to more effectively replan following unexpected changes [6], [15], these approaches seed the starting population of an evolutionary algorithm with outdated solutions to improve replanning.

One promising approach is the use of neuro-controllers, which allow a neural network to decide the system's adaptation actions based on the state of the system and its environment. Neural networks have the ability to solve complex problems, including outperforming the best human players at games such as Go [11] and Poker [12], and have been increasingly explored in the area of self-* systems [9], [10], [21]. However, the performance of neural net based approaches can degrade if the system's operating environment changes after training.

While self-* systems can adapt to the kinds of changes for which they were designed, they often struggle with unanticipated, *unexpected changes*. This is particularly true for systems built around neural networks, which can be rendered ineffective after changes to the system's operating environment. Various taxonomies of uncertainty have been proposed [22], in this work, we use unexpected changes to mean changes that were unanticipated during design and training time.

B. NEAT

Artificial Neural Networks are machine learning models with the ability to learn non-linear decision boundaries. A network is composed of input nodes, hidden nodes, and output nodes, which are connected by edges. Hidden nodes are organized into hierarchical layers. Based on the value of input nodes (or neurons) and their corresponding edge weights, each layer passes information to the next until the output layer produces a final set of values.

While the number of hidden layers and types of connections can be manually specified, the optimal arrangement is often unclear. Conventional neural networks keep this topology constant, and instead learn the value of weights through backpropogation. Alternately, NEAT (NeuroEvolution of Augmenting Topologies) trains by evolving the hidden topology, weights, and connections of artificial neural networks.

Neural network architecture is encoded in NEAT with a *genotype*. Each genome has a set of *node* and *connection* genes. The node gene set (analogous to neurons) contains the input, output, and hidden nodes. Connection genes (analogous to weights) are each associated with two nodes, an innovation number, and a boolean "enable" variable.

The analogy to backpropagation in neuroevolution is *mutation*: networks can mutate by either adding a connection between two existing nodes, or by adding a new node and disabling a pre-existing connection.

After mutation, *crossover* with gene alignment is performed to generate new networks. Once two parents are selected, whatever genetic information they share is automatically copied onto the child genome. If there are any remaining genes, those of whichever network performed better are passed on to the child.

Using *speciation*, NEAT ensures networks only compete with other networks of a similar structure. The fitness of a model is also penalized based on the size of its species. This explicitly supports innovation, since networks starting to develop complexity are protected long enough to evolve into strong competitors. Protecting innovation gives NEAT the advantage of starting with a minimal network, manually defined initialization. In contrast, many other approaches must be initialized randomly to ensure similar genetic diversity.

In our application of NEAT, we partially violate this principle of minimal initialization. Instead of starting minimally, we seed NEAT with prior networks that were trained on different problems. In our evaluation, we did not observe any issues with genetic diversity—but this may have been because the networks used as seeds started their evolution minimally.

III. Exemplar self-* system

To serve as a running example and a case study system, we present a self-* web-services adaptive planner (SWA) inspired by AWS and adapted from prior work [15]. SWA is a cloud-based web server that provides content in response to user requests, while weighing competing quality objectives. For instance, SWA can increase the number of ads served and increase revenue, but this will also decrease user satisfaction.

TABLE I
Regions in exemplar system with cost and number of availability zones. Cost data from Concurrency Labs [23].

Location	Name	Cost in $ per instance per month	Number of Availability Zones
N. Viriginia	us-east-1	69.12	6
Ohio	us-east-2	69.12	3
Oregon	us-west-2	69.12	4
Mumbai	ap-south-1	72.72	3
Stockholm	eu-north-1	73.44	3
Canada	ca-central-1	77.04	2
Ireland	eu-west-1	77.04	3
London	eu-west-2	79.92	3
Paris	eu-west-3	80.64	3
N. California	us-west-1	80.64	2
Frankfurt	eu-central-1	82.80	3
Seoul	ap-northeast-2	84.96	3
Singapore	ap-southeast-1	86.40	3
Sydney	ap-southeast-2	86.40	3
Tokyo	ap-northeast-1	89.28	3
Sao Paulo	sa-east-1	110.16	3

A. Architecture

The architecture of SWA mirrors that of AWS, with virtual server instances grouped in availability zones, which are further grouped into regions. SWA models 50 availability zones available over 16 regions. Table I shows these regions and the cost associated with running a server instance in each region. Prices are based on AWS's price list API [23].

Quality Objectives

The system must balance two competing quality objectives: profit, and user experienced latency. To increase profit, the system can send ads along with requested content, but these profits are mediated by the cost of running additional server instances. The system can also reduce the resolution of media content to handle requests faster—however this decreases user satisfaction and ad revenue.

B. Adaptation Tactics

While operating, the system deals with continually shifting conditions (i.e., the number of requests and the failure of server instances). To self manage these uncertainties, SWA has three adaptation tactics—(a) start or stop instance (b) raise or lower dimmer (which controls the percentage of requests with reduced media content) and (c) redistribute requests throughout availability zones. Each zone can have 0 to a maximum of 5 servers running at a time, each dimmer has five settings, and each traffic level also has five settings. With these variables considered at each of 50 availability zones, there are a total of 6×10^{108} possible configurations.

Server instance quality depends on three variables: cost, brownout ratio, and power. *cost* measures the price charged to run an instance per unit of time, and *power* measures how many low-fidelity instances can be served within a fixed time frame. The ratio of low-fidelity instances to full instances the system can serve per time unit is the *brownout ratio*. The system is initialized with the costs shown in Table I, power

TABLE II
SCENARIO ATTRIBUTE AND SELECTION PROBABILITY DURING MUTATION.

Attribute Type	Selection Rate
Utility Coefficients	13.33%
Tactic Failure Rates	23.33%
Instance Cost	21.11%
Instance Power	21.11%
Instance Brownout	21.11%

TABLE III
AN EXAMPLE UNEXPECTED CHANGE SCENARIO.

	Server Cost us-east-1	Server Startup Time us-east 1	Ad revenue	...	Server Cost us-east-1	...
Start Scenario	10	30	5	...	15	...
Unexpected Change	20	17	5	...	15	...

is set to 1000, and brownout ratio is set to 2. Each of these variables is modeled on a per availability zone basis.

C. Utility

Like many search-based and machine learning approaches, evolutionary algorithms require a means of evaluating the fitness, or utility of candidate solutions. In this work, we assume a model of the system and environment is available that can provide an estimate of the expected utility through simulation. For our case study system, this utility is a weighted sum of profit and a latency penalty, aggregated over a time window. The incoming request rate is simulated based on randomly selected traces from a commonly used load dataset [24], and for the purposes of our evaluation the utility window consists of 200 minutes of request data. The aggregate utility is computed by adding the utility at each timestep, with timesteps discretized to one minute granularity.

D. Unexpected change scenarios

In this work, we specifically address a self-* system's ability to self heal and optimize in light of *unexpected changes*–those changes that were not considered at design time. To simulate *unexpected change*, we introduce a mechanism for mutating scenarios. A *scenario* is a vector of 158 attributes that define the behavior of the exemplar system. The first two attributes are profit and latency coefficients which weight these variables in the system's utility function. The next 6 are values denoting the success rate of each tactic. The last 150 are cost, power, and brownout ratio, which each have a setting in all 50 zones. A discussion about the kinds of changes that we can adapt to is provided in Section VII.

For each scenario mutation, the system selects one parameter of the scenario vector to alter. Since most of the parameters influence only individual availability zones, while others affect higher-level factors like tactic failure rates and objective weights, parameters are selected for mutation according to a distribution designed to promote scenario diversity, shown in Table II. Objective coefficients are chosen 13%, the time, tactic failure rates are chosen 23% of the time, and availability zone specific factors are changed 63% of the time. We selected these values to generate scenarios that exercised a variety of unexpected changes, and not to try and model how likely these changes are to occur in practice. Table III shows an example unexpected change scenario generated by performing two mutations to a scenario.

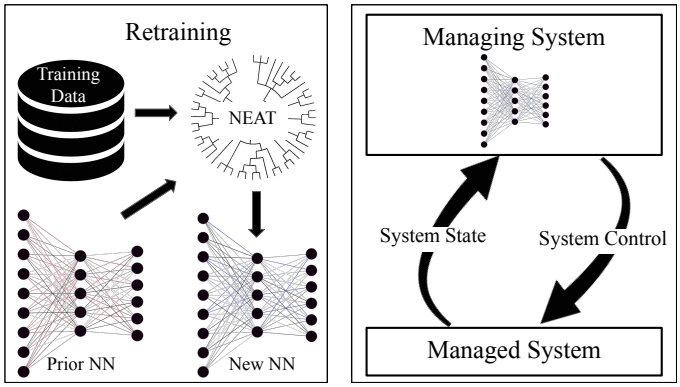

Fig. 1. A high level diagram of the proposed approach. On the left, the retraining process. On the right, the use of the neuro-controller in adaptation.

IV. APPROACH

We propose *neuro-reuse* as a strategy to enable self-* systems driven by neuro-controllers to adapt more effectively following an unexpected change. Neuro-reuse works by seeding the population of a neuroevolutionary algorithm with previously existing neural nets, allowing the search to reuse the information contained in these networks rather than starting over from scratch. Figure 1 shows the approach at a high level, with the left side showing the retraining process. After an unexpected change occurs, the existing neuro-controller becomes outdated and must be replaced for the system to continue satisfying its quality objectives. While the existing controller is ineffective, it may still contain useful knowledge that can increase the speed of retraining. This outdated ANN is provided as an input to the NEAT neuroevolutionary algorithm and is reused by seeding the starting population to generate an updated controller. The right side of Figure 1 shows the relationship between the neuro-controller and the managed system. To integrate the controller with the managed system, the system's state space and available control actions must be mapped to the controller. In this section, we first describe how inputs passed to the neuro-controller are represented, and how outputs are used to decide on plans for the motivating case study system described in Section III. We then discuss how neuro-controllers are reused in greater detail. Additionally, source code for our approach is available for review and replication [2].

[2]https://bit.ly/2EAtXVK

A. State Space Representation

To integrate a self-* system with NEAT ANNs, we must provide a minimal initial network to start the search, as well as a mapping from the system's state into the input space that an ANN will accept, and a mapping from the ANN's output to the control actions that the system can utilize to adapt. A minimal encoding is desirable because it allows evolution to begin with the smallest possible network, improving the effectiveness of NEAT.

*1) **Initial Network**:* Because NEAT optimizes and complicates networks as they evolve, it is designed to start with a minimal network structure. While NEAT is evolving an ANN, it modifies both network topology and weights, but it preserves the dimensionality of the input and output space (the monitors and affectors for a self-* system). Our initial network, shown in Figure 2, is a fully connected ANN with 10 input, 3 hidden, and 6 output neurons. The inputs correspond to information about the state of the system, and the outputs to actions that the system can take in response to these conditions. All of the links (weights) are initialized to zero and enabled.

*2) **Input Space**:* To minimally encode all possible system configurations, our input layer has 10 neurons, shown in Figure 2. The first neuron corresponds to the number of requests at that time-step. For traffic level, server instances, and dimmer level, we use three neurons each to encode the state of the system. Traffic and dimmer levels can be in one of 5 positions over 50 availability zones, represented as an array of length 50 where each element ranges from 0-4, and server instances can be in one of 6 positions (0-5). We convert these values to binary (allocating three bits per integer) and then concatenate them into one string. The string is then broken into three equal segments corresponding to the three neurons.

*3) **Output Space**:* The six outputs produced by the network are all doubles, which much be mapped onto the possible control actions. Each output neuron controls one of the six possible tactics: start server, shut down server, increase traffic, decrease traffic, increase dimmer, and decrease dimmer. The output layer produces this output as a six integer array. All of the doubles must then be converted to an integer from 0-50, corresponding to the 50 availability zones and one extra input for enacting no action. To map the output neurons to the possible control actions, we round each output to six decimal places and then multiply by $1,000,000$. This brings each double into the range $0 - 999,999$. We then mod by 51 which converts all the outputs to an integer from 0-50. For example if the output after the conversion is 32,0,0,0,0,12, then the system will start a server in zone 32 and decrease the dimmer in zone 12.

B. Reusing Networks

The high-level approach for reuse is shown on the left in Figure 3. The approach begins after an unexpected change occurs. In the example, the system is operating under scenario 1, and controlled by neural network *a*. After the operating environment changes unexpectedly, the system is now operating in scenario 2, and the neural network *a* is no longer suitable.

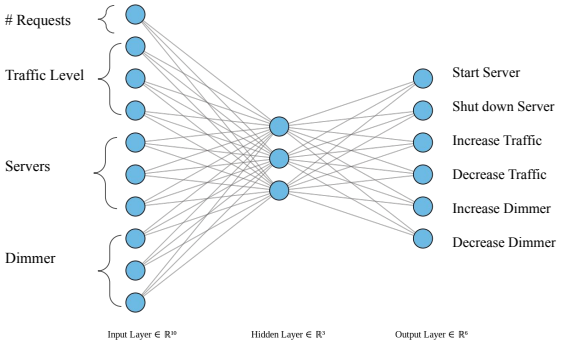

Fig. 2. Initial ANN that is manually defined for NEAT to evolve.

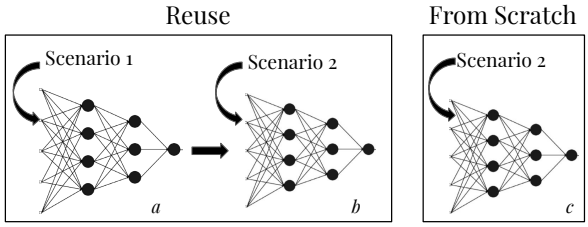

Fig. 3. The neural networks and unexpected change scenarios used in the evaluation.

As an example within the context of the case study system, the first scenario could have a large profit coefficient, making "high utility" plans those that send more advertisements. After the unexpected change, however, the profit coefficient could decrease dramatically, instead causing the system to prefer plans that lower latency and send fewer ads.

This event triggers retraining to occur. In this work, we do not address how the system detects the change and decides when retraining is necessary, instead we are interested in what the system should do when this occurs. Network *a* is seeded into the initial population of the search, and a search process starts to find a new controller, neural network *b*, that can cope with the new scenario 2.

While reuse by seeding the population of a neuroevolutionary algorithm has shown promise in related areas [25], [26]. Research in knowledge reuse for self-* systems has shown that reusing existing adaptation strategies is challenging and naive reuse can be counterproductive [6]. There are many factors that can influence the effectiveness of reuse, and it is unclear how reuse will perform in this context, motivating the need for our evaluation.

V. Evaluation

We evaluate the approach by presenting an empirical study using the exemplar system described in Section III. In this evaluation, we first investigate whether neuro-reuse can result in improved training effectiveness compared to retraining from scratch, either in terms of improved system utility or reduced training time. We then seek to understand the factors that contribute to the effectiveness of neuro-reuse, specifically investigating how the quality of the training data affects

its performance, as well as how neuro-reuse functions with different unexpected change sizes. We therefore investigate the following research questions:

- Does neuro-reuse result in increased effectiveness compared to generating a new neural network from scratch?
- How does reuse with a single trace compare to training with random traces?
- How does reuse effectiveness vary with the degree of change?

For purposes of evaluation, we split the request trace data into a training set and a testing set, with 80% of the trace data available for training and 20% held back for testing. During training, traces of length 200 are randomly selected from the train set to compute fitness. For testing, we randomly select 10 traces of length 200 from the test set at each generation. NEAT is configured to maintain a population of 150 individuals and run for 2000 generations. These values were chosen based on observing that the population converged to a solution within these settings in preliminary experiments.

To test reuse, we first train one neural network on a randomly selected scenario using NEAT. This serves as the network to be reused–call it network *a*. We then introduce an unexpected change to the system that necessitates re-planning. This is a distinct but also randomly selected scenario. Given an unexpected change, we compare two methods of re-planning:

1) **Planning from scratch.** Using NEAT, we train an entirely new neural network (network *c*) with the second scenario as input. In this case, the initial network architecture is identical to the one described in Section IV, but this network is entirely independent of information in network *a*. This results in a new neural network for the new scenario, with no information reused.
2) **Neuro-Reuse.** For reuse, our goal is to re-purpose a network trained for a previous scenario to respond to the new one. Instead of starting from the minimal network shown in Figure 2, we use the final network from the evolution process of network *a* as our NEAT initialization, breaking the assumption of minimality. Additionally, this network receives the second, mutated scenario as its input. We then reuse network *a*, which was trained to generate plans for the first scenario by evolving it into a new network *b*, that responds to the second scenario. That is, as opposed to network *c*, network *b* reuses information from the original network *a*. We generate the unexpected change scenario by applying mutations to the start scenario as described in Section III-D. The quality of each plan is evaluated with respect to the utility metric described in Section III-C. For both methods, we measure the utility over time during the re-planning stage, as well as test utility.

Our objective is to determine whether reusing the information encoded in the original network will either (1) reduce the time it takes for the system to converge to an acceptable plan or (2) improve the aggregate utility of the system within a given planning window.

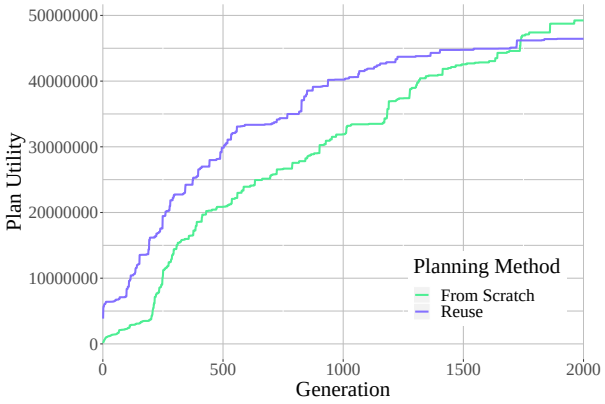

Fig. 4. The utility of neuro-reuse and planning from scratch plotted during training with a new randomized trace at every generation. Average is taken over 30 trials.

We first explore whether using neuro-reuse for a relatively small change scenario can result in improved effectiveness, in terms of search time and system utility. For this experiment we generate two neuro-controllers for each unexpected change scenario, one generated from scratch with no seed material, the second generated by seeding the search with the neuro-controller generated for the start scenario. For the first experiment, unexpected change scenarios were generated by applying five mutations, and we performed 30 trials.

Next, we study how overfitting impacts the network's ability to reuse information. Whereas the first experiments pass the neuro-controller a new randomly selected set of 200 traces at every generation, in these tests, we continually pass the neuro-controller the same trace for all 2,000 generations. Here, unexpected change scenarios were also generated with five mutations, and we again perform 30 trials.

Finally, we examine whether neuro-reuse can still be an effective means of re-planning when there is a large change between scenarios, and we study the effect of scenario change on performance. Specifically, we measure the test and training utility of planning from scratch when there are 1,5,10,15, and 20 changes to the scenario, and we performed 5 trials for each number of changes.

VI. Results

A. RQ1: Does neuro-reuse result in increased effectiveness compared to generating a new neural net from scratch?

To answer this question, we evaluated both the test and train results of *neuro-reuse* as compared to planning *from scratch*. Figure 4 shows the utility per generation averaged over 30 trials for both methods. Neuro-reuse outperforms re-planning from scratch on average for 86% of training generations, making it a more effective method for re-planning.

Observe, however, that the final average utility is slightly higher for planning from scratch. Different systems have different needs. Some may be focused on achieving a good plan quickly, whereas others may have more time to spend

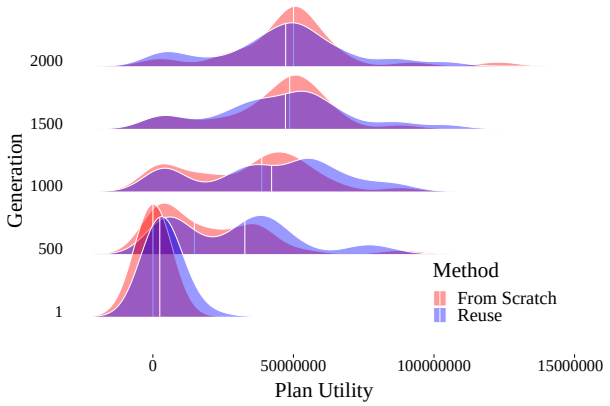

Fig. 5. Probability distribution of utility during evolution when traces are randomized at every generation. Snapshots are shown every 500 generations for both methods, and the white line represents the mean.

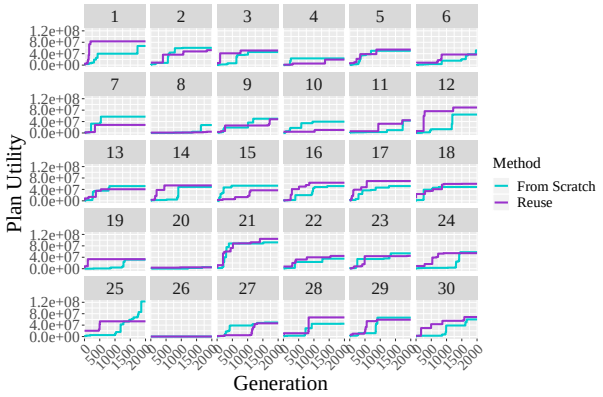

Fig. 6. Utility during training where both reuse and from scratch networks were passed a new randomized trace at every generation. Plots are shown for each of 30 independent trials.

developing a high utility plan. Thus, it is important to understand how the likelihood of finding a good plan changes at different stages in evolution.

To understand how these probabilities shift over the course of re-planning, we recorded the probability distributions as both networks evolve. Figure 5 shows the probability of both methods reaching a given utility at four snapshots in their evolution. During the first half of evolution (generations 1-1000), the average utility for neuro-reuse is substantially higher than replanning from scratch. During the second half of evolution, however, the planning strategies converge to a similar probability distribution. Thus, while neuro-reuse only reached a higher final train utility than planning from scratch on 50% of trials, it had the advantage of finding a higher utility plan within the first 1,000 generations on 73% of trials.

Additionally, when neuro-reuse does perform better, it usually does so by a wider margin. The trend is most apparent when observing the results on a trial by trial basis. Figure 6 shows how plan utility changed throughout evolution during each of 30 trials. There are generally three types of trials: those where neuro-reuse dominates such as 1,12, and 14, those where both methods appear nearly equivalent like 5, 22, and 29, and trials like 7, 10 and 15 where planning from scratch does better. In the first case–when neuro-reuse ends training with a better plan—the average utility is 32% higher than when planning from scratch wins. In other words, when neuro-reuse "wins" it wins big, and when it "loses", it loses small.

Another question that impacts our quality evaluation is how well neuro-reuse performs on new held-out test traces. At each generation of evolution, we performed 10 tests on held-out test traces to see how the performance on the test traces changes during its training. Figure 7 shows the average of these tests. At the end of training, neuro-reuse has a better test utility 65% of the time. Within the first 350 generations, reuse has a better test utility 93% of the time. Notice, however, there is a point at which neuro-reuse's test utility dips (∼350 generations), and where planning from scratch's utility starts to decline (∼1,500 generations). One explanation for this pattern is that

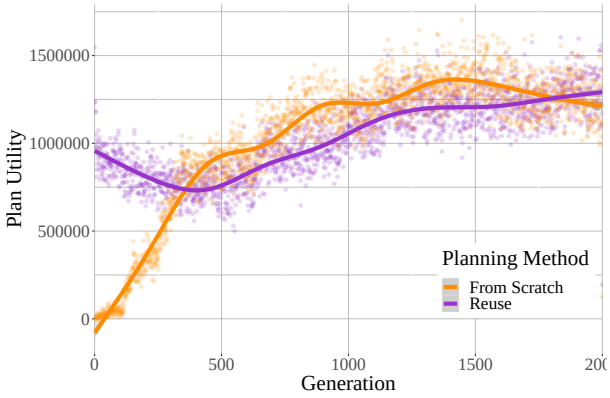

Fig. 7. Result of tests performed during network evolution. At each generation, we performed 10 tests. The average of these tests as well as the smoothed local regression line are plotted above.

test utility declinines as networks become more specialized for their training task. Since the reused network is evolved from a network that was already trained, this may explain why its test utility dips earlier.

Key Insight: Neuro-reuse is better than planning from scratch in general, as it maintains a higher utility during 86% of training and has a higher final test utility 65% of the time. In particular, the reused network is especially likely to reach a higher utility plan within the first half of training, indicating it is a useful technique for time or resource constrained re-planning.

B. RQ2: How does neuro-reuse with a single trace compare to training with random traces.

At its core, neuro-reuse relies on a network's ability to generalize across scenarios. Overfitting—the practice of giving a model access to biased data during training—often causes ANNs to generalize poorly. Extrapolating this concept, we

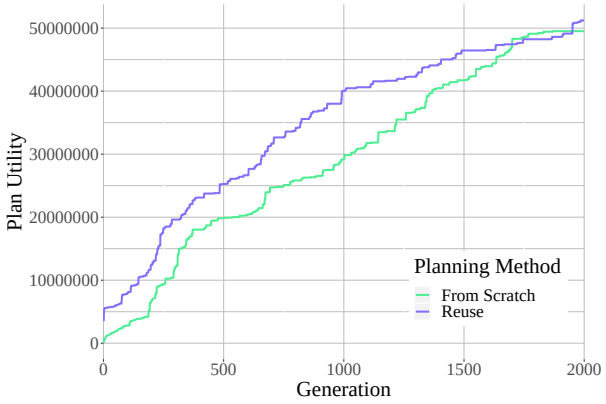

Fig. 8. The utility of neuro-reuse and planning from scratch plotted during training when the network is overfit on the same trace at every generation. Average is taken over 30 trials.

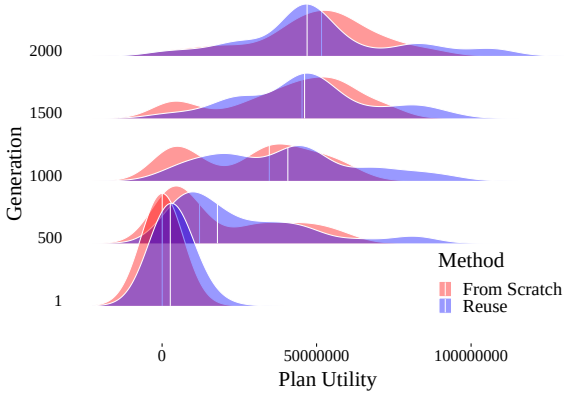

Fig. 9. Probability distribution of utility during evolution when the network is continually overfit on the same trace. Snapshots are shown every 500 generations for both methods, and the white line represents the mean.

examined the consequences of overfitting reused networks on the networks we evolve from them.

In this RQ, we repeat the experiments from RQ1, this time purposely over-fitting the training data with only a single trace of length 200. The results averaged over 30 trials are shown in Figure 8. Interestingly, reuse still has better performance for the majority of trials even when its source network was severely overfit—it outperforms planning from scratch on 87% of generations. The final average utility during training is also slightly higher on average when the traces are not randomized, however both of these changes were marginal.

Overall, training exhibited a similar pattern with and without random traces. Still, it is surprising that the utility of planning from scratch is largely unaffected, as training evaluation is often arbitrarily inflated by over-fitting. Additionally, learning from a network only exposed to one trace would seemingly decrease neuro-reuse's utility.

Comparing Figure 9 and Figure 5, however, some differences between the overfit and randomized trials begin to emerge. During the first 1000 generations, there is a decrease in the mean utility for reuse—particularly pronounced at generation 500. The line graphs also show this pattern, whereas reuse's utility curve appears exponential in Figure 4, it becomes linear in Figure 8. This suggests that reuse's early advantage in training was aided by learning from a network that had been exposed to randomness during its own evolution.

The full effects of over-fitting appear in Figure 10, which shows how the test evaluation shifted over the course of evolution. Notice that the highest utility reached during testing when overfit ($\sim 7.5 \times 10^5$) is roughly an order of magnitude lower than when traces are randomized ($\sim 1.5 \times 10^6$).

Although the utility of both methods decreased massively as a result of over-fitting, it is still interesting to observe that an average utility of approximately 600,000 was obtained by both methods. In situations with extremely limited amounts of data, this lower utility may still be relatively advantageous.

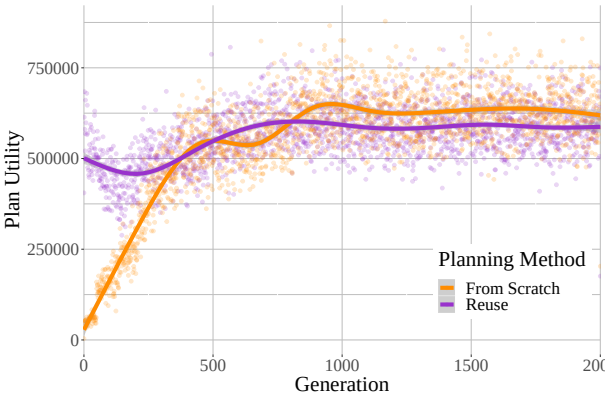

Fig. 10. Result of tests performed during network evolution with overfitting. At each generation, we performed 10 tests. The average of these tests as well as the smoothed local regression line are plotted above.

Key Insight: Overfitting does not severely impact the training evaluation of neuro-reuse or planning from scratch, however it dramatically decreases the test performance of both methods.

C. RQ3: How does the effectiveness of neuro-reuse vary with the degree of change?

Not all changes are created equal; some events radically alter the system by changing many parameters, whereas others may only make small adjustments. As we are interested in understanding neuro-reuse as a method for responding to unexpected events, we compare the two methods across varying degrees of change, allowing us to evaluate the relationship between system change and re-planning performance.

As described in Section III-D, each scenario mutation randomly alters one aspect of the system. By changing how many mutations the system undergoes, we can affect the degree of change between the scenarios the source and target networks are trained on. In all prior experiments, there were five random parameter mutations between the first and second

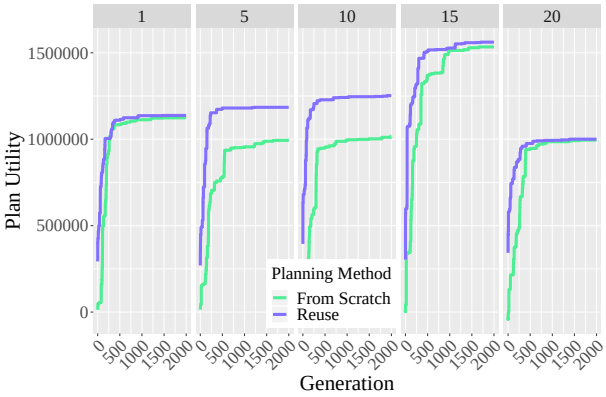

Fig. 11. The relationship between scenario change and plan utility during training. The utility shown was averaged over 5 trials for 1, 5, 10, 15, and 20 scenario mutations (respectively).

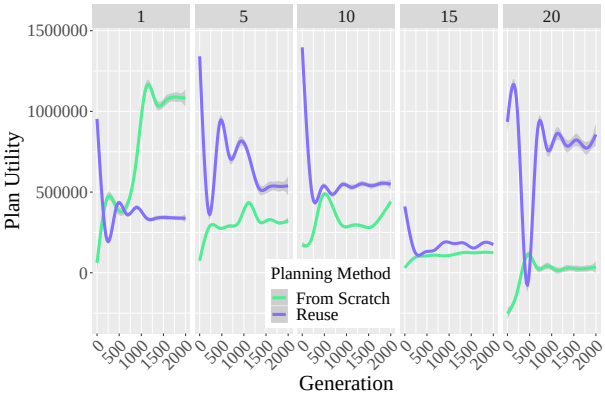

Fig. 12. The relationship between scenario change and plan utility during testing. Utility was averaged over 5 trials and smoothed with local regression for 1, 5, 10, 15 and 20 scenario mutations.

scenario. For this RQ, we test the system's performance when it is exposed to one, five, ten, fifteen, and twenty parameter mutations (respectively). The results are shown in Figure 11 averaged over 5 trials.

The results show that neuro-reuse generally performed better as the degree of change between scenarios increased. Throughout training, reuse and planning from scratch have similar utilities on one, fifteen, and twenty changes. When there were five and ten changes, however, reuse retained a higher utility throughout the entirety of training. With the exception of twenty mutations, the average training utility increased for both methods as the degree of change increased.

Although both methods appear similar during training, neuro-reuse performs substantially better on random tests when exposed to more change. The test results in Figure 12 show this pattern. When there is a relatively minor degree of change (1 mutation), planning from scratch does better on random tests. But as the system is exposed to a greater degree of change in the remaining trials, reuse distinctively outperforms planning from scratch. This result indicates that neuro-reuse results in a more durable network—one that can more effectively respond to scenarios it was not trained for.

Key Insight: Even with 4x as much change between the scenarios that the source and target network trained on, neuro-reuse out performs planning from scratch in both test and train evaluations, demonstrating that it is an especially useful method for dealing with environments of uncertainty.

VII. Discussion

In this section, we provide a discussion of the results, including threats to validity and ideas for future work.

A. Threats

There are several threats to the validity of the results that we attempted to minimize. Since we evaluated the approach on a single case study system, it is possible the results may not generalize to other systems. To mitigate this concern, we selected the case study system to be representative of existing real-world self-* systems. Similarly, by their definition, we cannot anticipate apriori all of the possible unexpected changes that a self-* system may encounter, and the system may face very different kinds of changes than the ones that we evaluated. Our approach for generating unexpected change scenarios through random mutations partially mitigates this concern, since it can generate scenarios that the human designers never considered, however the change space remains constrained by the supported mutation operators. The chosen mutation operators were selected to allow the system to change in ways that are likely to occur, and it is unrealistic to prepare for all possible unexpected changes.

B. Future Work

This work opens a number of promising research directions to seek further improvements in adaptation ability in response to unexpected changes. This work only examined unexpected changes that did not result in changes to the neural network's input or output dimensionality. It remains to be seen how applicable neuro-reuse is to these types of changes, for example the addition or removal of available adaptation tactics or sensors. Other work in reusing prior knowledge in self-* systems found that proactively generating planning knowledge to reuse resulted in increased adaptation effectiveness to novel scenarios [15]. The application of this idea to neuro-controlled systems could result in further improvements to retraining effectiveness, and perhaps mitigate the potential for overfitting. The application of novelty search was shown to be effective for enabling transfer learning in neuroevolution [25], [26] and could also be applied in the self-* systems context.

VIII. Related Work

Many approaches have been explored to enable self-* systems to decide how to adapt. These approaches include planning languages for offline planning such as Stitch [3] that enable humans to specify adaptation strategies. Other

approaches enable automated planning, such as using Markov Decision Processes [5], Bayesian optimization [7], and reinforcement learning [8]. Despite the research in this area, many approaches struggle to handle unexpected changes not considered at design time [22], the focus of this work.

Evolutionary approaches have also been explored as a way of deciding how to adapt, including using genetic algorithms [19], [20]. In particular, seeding the population of a genetic programming population has been investigated as a way to reuse existing knowledge in response to unexpected changes [6], [15]. Like these works, we seed the population of an evolutionary algorithm with prior knowledge to enable self-* systems to respond more effectively to unexpected changes. The approaches differ in that we reuse neuro-controllers rather than self-* strategies represented as ASTs, aiming to mitigate a key limitation of deep learning approaches, that they can stop working after changes to the system's operating environment.

A related area is case-based reasoning, which has investigated the idea of reusing information from from prior decision making contexts based on selected solutions from previously encountered cases [27], [28]. Seeding evolutionary algorithms with existing solutions has been explored in this context [29]. These works do not however address the challenges of adapting neuro-controllers to respond to unexpected changes.

In this work we focus on self-* systems driven by neuro-controllers. Using deep learning to facilitate self-adaptation is a growing area of research, and has been used for many planning and decision making tasks. Neural network approaches shown increasing promise in solving complex decision making tasks, which has been demonstrated by defeating top human players in the games of Go [11] and Poker [12]. Some examples include using neural networks for general planning [30] and route planning [31], the management of iot nodes [21], and distributed controllers for flying in formation [9]. In particular our work uses neuro evolution [18], [32] which has also been used in control tasks, including solving non-Markovian control tasks [33]. While neural networks are a promising solution idea for complex decision making tasks, existing work does not investigate how to enable these approaches to evolve in response to unexpected changes.

One approach aiming to improve the ability of neuro-controllers to adapt more effectively to change is the use of neuromodulation [10]. This idea is inspired by biology where chemicals in the brain can change or modulate the behavior of neurons, with the idea of increasing the plasticity of neural nets. Barnes et al. explore applying this idea to improve the ability to learn multi-stage tasks, as well as allow agents to adapt to changes in the number of agents in the environment. This focus on adaptation in response to additional agents is similar in motivation to our work where we seek to allow neuro-controllers to respond to unanticipated changes, although the approach of neuromodulation rather than network seeding is different from our approach.

The subset of machine learning known as *Transfer Learning* responds to this problem. Transfer learning seeks to exploit information about one task to improve performance on another related task. Approaches within this domain are divided into two categories: those that reuse models across probability distributions to categorize data [34], and those that deal with agents learning from limited environmental feedback (reinforcement learning tasks) [35]. Our approach effectively combines these domains by transferring information between artificial neural networks in a reinforcement learning problem domain. Additionally, transfer learning generally has not yet been investigated as a method for responding to unexpected changes in neuro-controlled self-* systems. A related example is the use of transfer learning to allow a deep learning robot to transfer knowledge from a simulator to the real world [36], which like our work reuses neural networks from one operating context to another, but does not focus on unexpected changes in self-* systems. Transfer learning has also been applied to neuroevolution in the context of keep-away in robotic soccer [25], [26]. Like our work, these approaches seed a neuroevolutionary algorithm with existing neural networks to improve training. The approaches differ in that we reuse information in order to enable a self-* system to respond to an unexpected change, while the related work uses transfer learning to help learn more complicated tasks.

IX. Conclusion

Software systems increasingly operate in uncertian environments, and need to autonomously respond to change. Research in self-adaptation has made progress in allowing systems to manage this uncertainty by autonomously reconfiguring at runtime. One promising approach to facilitate adaptation is the use of ANNs as controllers, since they have the ability to handle complex problems with large state spaces. While ANNs are powerful, they can struggle when the system's operating environment changes. In this work, we propose neuro-reuse as an approach to allow ANN driven self-* systems to adapt more effectively following unexpected changes. This approach combines neuroevolution and transfer learning to allow the system to leverage knowledge from prior operating contexts to retrain following an unexpected change. We evaluate this approach on a simulated cloud web service provider inspired by AWS, finding that neuro-reuse results in increased retraining performance and can outperform retraining from scratch even for large change scenarios.

Acknowledgment

Acknowledgments will be added to support a camera ready.

References

[1] Statista, *Microsoft Teams, daily and monthly users*, April 2020 (accessed August, 2020). [Online]. Available: https://www.statista.com/statistics/1033742/worldwide-microsoft-teams-daily-and-monthly-users/

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, pp. 41–50, 2003.

[3] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[4] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: a probabilistic model checking approach," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 1–12.

[5] A. Pandey, G. A. Moreno, J. Cámara, and D. Garlan, "Hybrid planning for decision making in self-adaptive systems," in *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2016, pp. 130–139.

[6] C. Kinneer, Z. Coker, J. Wang, D. Garlan, and C. Le Goues, "Managing uncertainty in self-adaptive systems with plan reuse and stochastic search," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2018, pp. 40–50.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 469–482.

[8] D. Kim and S. Park, "Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software," in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2009, pp. 76–85.

[9] S. Roy, U. Mehmood, R. Grosu, S. A. Smolka, S. D. Stoller, and A. Tiwari, "Learning distributed controllers for v-formation," 2020.

[10] C. Barnes, A. Ekart, K. O. Ellefsen, K. Glette, P. Lewis, and J. Torresen, "Coevolutionary learning of neuromodulated controllers for multi-stage and gamified tasks," 08 2020.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[12] N. Brown and T. Sandholm, "Superhuman ai for heads-up no-limit poker: Libratus beats top professionals," *Science*, vol. 359, no. 6374, pp. 418–424, 2018.

[13] C. Z. Vitaly Feldman, "What neural networks memorize and why: Discovering the long tail via influence estimation," 2020.

[14] F. . Tzeng and K. . Ma, "Opening the black box - data driven visualization of neural networks," in *VIS 05. IEEE Visualization, 2005.*, 2005, pp. 383–390.

[15] C. Kinneer, R. Van Tonder, D. Garlan, and C. Le Goues, "Building reusable repertoires for stochastic self-* planners," in *Proceedings of the 2020 IEEE Conference on Autonomic Computing and Self-organizing Systems (ACSOS)*, Washington, D.C., USA, 17-21 August 2020.

[16] P. H. Calais Guerra, A. Veloso, W. Meira, and V. Almeida, "From bias to opinion: A transfer-learning approach to real-time sentiment analysis," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 150158. [Online]. Available: https://doi.org/10.1145/2020408.2020438

[17] P. Jamshidi, M. Velez, C. Kästner, N. Siegmund, and P. Kawthekar, "Transfer learning for improving model predictions in highly configurable software," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 31–41.

[18] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, no. 2, p. 99127, Jun. 2002. [Online]. Available: https://doi.org/10.1162/106365602320169811

[19] A. J. Ramirez, B. H. Cheng, P. K. McKinley, and B. E. Beckmann, "Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809080

[20] S. Gerasimou, R. Calinescu, and G. Tamburrelli, "Synthesis of probabilistic models for quality-of-service software engineering," *Automated Software Engineering*, vol. 25, no. 4, pp. 785–831, 2018.

[21] A. Murad, F. A. Kraemer, K. Bach, and G. Taylor, "Autonomous management of energy-harvesting iot nodes using deep reinforcement learning," in *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2019, pp. 43–51.

[22] R. Calinescu, R. Mirandola, D. Perez, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *Proceedings of the 2020 IEEE Conference on Autonomic Computing and Self-organizing Systems (ACSOS)*, Washington, D.C., USA, 17-21 August 2020.

[23] C. Labs, *Choose your AWS Region Wisely*, accessed: 2020-18-17. [Online]. Available: https://www.concurrencylabs.com/blog/choose-your-aws-region-wisely/

[24] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE network*, vol. 14, no. 3, pp. 30–37, 2000.

[25] S. Didi and G. Nitschke, "Multi-agent behavior-based policy transfer," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2016, pp. 181–197.

[26] G. Nitschke and S. Didi, "Evolutionary policy transfer and search methods for boosting behavior quality: Robocup keep-away case study," *Frontiers in Robotics and AI*, vol. 4, p. 62, 2017.

[27] H. Muoz-Avila and M. T. Cox, "Case-based plan adaptation: An analysis and review," *IEEE Intelligent Syst.*, vol. 23, no. 4, pp. 75–81, 2008.

[28] S. J. Louis and J. McDonnell, "Learning with case-injected genetic algorithms," *Trans. Evol. Comp*, vol. 8, no. 4, pp. 316–328, 2004.

[29] A. Grech and J. Main, *Case-Base Injection Schemes to Case Adaptation Using Genetic Algorithms*, ser. ECCBR, Berlin, Heidelberg, 2004, pp. 198–210.

[30] R. Wyatt, "Using neural networks for generic strategic planning," in *Artificial Neural Nets and Genetic Algorithms*. Vienna: Springer Vienna, 1995, pp. 440–443.

[31] D. R. Bruno, N. Marranghello, F. S. Osrio, and A. S. Pereira, "Neurogenetic algorithm applied to route planning for autonomous mobile robots," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8.

[32] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[33] F. J. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuroevolution," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 13561361.

[34] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *ArXiv*, vol. abs/1911.02685, 2019.

[35] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *J. Mach. Learn. Res.*, vol. 10, p. 16331685, Dec. 2009.

[36] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," in *Conference on Robot Learning*, 2017, pp. 262–270.