12 Synergies with Reinforcement Learning

Reinforcement learning and neuroevolution are two prominent approaches for optimizing the performance of neural networks, but they employ fundamentally different methodologies with distinct trade-offs. In the first part of this chapter we will look at their respective advantages and disadvantages, and ways they could be combined.

In the second part of the chapter, we review approaches that go a step further, allowing evolved networks to invent their own learning algorithm without relying on existing RL methods. By leveraging the principles of neuroevolution, these networks can evolve not only their architectures and weights but also the intrinsic rules that govern how they learn and adapt over time.

12.1 RL Vs. NE

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions by taking actions in an environment to maximize cumulative reward. This approach involves the agent interacting with the environment in a trial-and-error manner, receiving feedback in the form of rewards or punishments. RL algorithms, such as Q-learning, Deep Q-Networks (DQN), and Policy Gradient methods, focus on finding a policy that dictates the best action to take in each state of the environment. One of the main advantages of RL is its ability to handle a wide variety of tasks, especially those involving sequential decision-making and dynamic environments. It is particularly effective in domains where the environment's model is unknown or too complex to be explicitly defined, such as robotics, game playing, and autonomous driving.

However, RL also has several drawbacks. It often requires a significant amount of data and computational resources due to the extensive exploration needed to discover effective policies. The training process can be unstable and sensitive to the choice of hyperparameters. Moreover, RL algorithms can struggle with high-dimensional state and action spaces.

Neuroevolution, on the other hand, is particularly advantageous in its ability to optimize both the topology and parameters of neural networks simultaneously, making it suitable for tasks where the optimal network structure is not known a priori. Additionally, neuroevolution tends to be more robust to the pitfalls of local minima, as the population-based search can explore a broader solution space compared to gradient-based methods used in RL. For example, by repeatedly running the algorithm from scratch, policies discovered using evolution tend to be more diverse compared to those discovered by reinforcement learning

algorithms. Despite these strengths, neuroevolution also faces certain limitations. For example, neuroevolution might not perform well in environments requiring real-time learning and adaptation since evolutionary processes generally operate on a longer timescale compared to RL's incremental updates. Additionally, especially when the environment provides dense rewards each time step, RL methods often show a higher sample efficiency than NE approaches.

12.2 Synergistic Combinations

In practice, RL and neuroevolution can be synergistically combined to leverage the strengths of both approaches. In the next chapters, we will take a look at some of the ways this can be achieved.

12.2.1 Evolutionary Reinforcement Learning

One of the primary difficulties in deep reinforcement learning is to discover optimal policies while avoiding early convergence to suboptimal solutions. Various techniques, such as intrinsic motivation or curiosity, have been suggested to address this issue. However, these methods are often not universally applicable and necessitate careful tuning. Given their population-based nature, effective exploration is an area where evolutionary approaches shine. Additionally, because returns are consolidated across entire episodes, they can often better deal with sparse rewards.

Evolutionary Reinforcement Learning (ERL) (Khadka and Tumer 2018) is a hybrid algorithm that addresses some of these challenges. ERL utilizes an evolutionary population to generate diverse data for training an RL agent and periodically integrates the RL agent back into the EA population to infuse gradient information into the EA process. This approach harnesses EA's capability for temporal credit assignment using a fitness metric, effective exploration through a variety of policies, and the stability of a population-based strategy. Simultaneously, it leverages off-policy Deep Reinforcement Learning to enhance sample efficiency and accelerate learning through the use of gradients.

An overview of the approach is shown in Figure 12.1a. Similar to the standard neuroevolution approach, a population of deep neural networks is evolved through an evolutionary algorithm (mutations and crossover), where the fitness is calculated as the cumulative sum of the reward during a rollout. Additionally, a portion of the best-performing individuals (the elites) are not mutated. This part of the algorithm is shown in the left side of Figure 12.1a.

To allow the algorithm to also learn within an episode, instead of only between episodes as in the standard neuroevolution setup, during each interaction for each actor and each timsteps information such as the current state, action, next state, and reward is stored in a replay buffer. This replay buffer is then used to train agents with a deep RL approach. While the EA explores through noise in the parameter space (i.e. mutating the weights of the network directly), RL approaches often explore through noise in the action space by sampling from the outputs of the network. ERL leverages both, by generating additional experiences for the replay buffer through a noisy version of the RL actor network.

To provide information back to the EA and to take advantage of the information from the gradient descent learning, every once in a while during a synchronisation phase, the weight

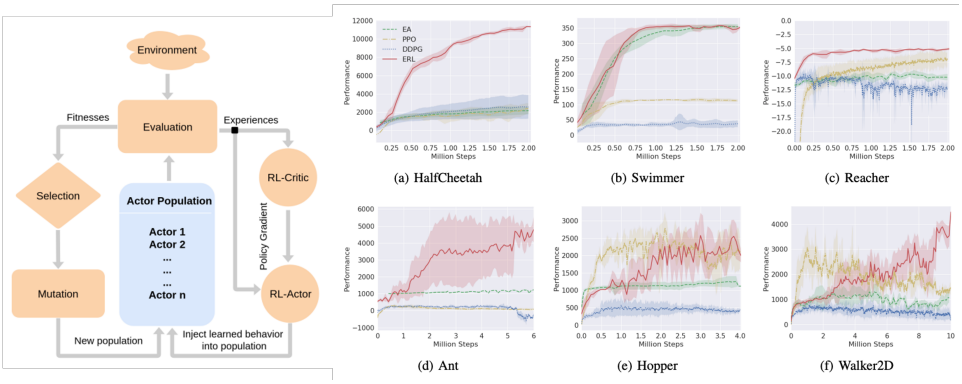

Figure 12.1: **Evolutionary Reinforcement Learning.** Left: In ERL, a population of neural networks is evolved through NE. Data collected during those rollouts is used to train a deep RL agent, which is periodically injected into the EA population. Right: In most domains, ERl significantly outperforms vanially EA and deep RL approaches. (Figure from Khadka and Tumer 2018)

of the RL actor network are copied back into the EA population. This network is then evaluated like any other network in the population, which allows good discovered policies to survive and extend their influence over subsequent populations while non-competitive policies will have less chances to reproduce. This transfer is shown to be particularly useful in domains with sparse rewards and deceptive fitness landscapes.

This method leverages EA's ability to explore the policy space and handle sparse rewards while enhancing sample efficiency and learning speed through DRL's gradient-based optimization. The algorithm is demonstrated on continuous control benchmarks, significantly outperforming state-of-the-art DRL methods like DDPG and PPO Figure 12.1b. ERL maintains effective exploration, stabilizes convergence, and enhances performance across various tasks by combining the episodic returns and population stability of EAs with the gradient efficiency of DRL.

12.2.2 Evolving Value Networks for RL

Many RL approaches rely on the concept of a value function. The value function estimates the expected cumulative reward that an agent can achieve from a given state or state-action pair and can thus guides the agent's actions. In deep RL, these value functions are implemented as neural networks, enabling agents to learn complex behaviors in environments with high-dimensional state and action spaces. However, decisions about the architecture of such a value neural network can crucially impact performance and not ideally chosen values can lead to poor agent performance.

A significant advantage of NE methods such as NEAT is that they can not only optimize the weights of a neural network but allow evolving the neural architecture at the same time. This approach is thus well suited to evolve the right initial parameters and architecture of RL agent value networks that are better at learning. This setup differs from the typical usage of NEAT to evolve a direct action selector network, where the network directly outputs the

action to be taken by the agent. Here the network only outputs the value of each state-action pair and the actual action to be taken is then derived from those values.

Before we detail how to integrate NEAT with the particular RL algorithm Q-learning, we first briefly describe how the Q-learning algorithm works by itself. Q-learning is a model-free reinforcement learning algorithm that aims to find the optimal policy for a given finite Markov Decision Process (MDP). The goal of Q-learning is to learn the action-value function, $Q(s, a)$, which represents the expected utility (cumulative reward) of taking action a in state s and then following the optimal policy thereafter.

The Q-learning algorithm involves initializing the Q-values arbitrarily for all state-action pairs, except for the terminal states where the Q-values are set to zero. At each time step t, the agent observes the current state s_t and selects an action a_t based on a policy derived from the current Q-values, such as the ε-greedy policy. This policy balances exploration and exploitation by choosing a random action with probability ε and the action with the highest Q-value with probability $1 - \varepsilon$.

After executing the action a_t, the agent receives a reward r_t and observes the next state s_{t+1}. The Q-value update rule is then applied to update the Q-value for the state-action pair (s_t, a_t) based on the observed reward and the maximum Q-value of the next state. The Q-value update rule is given by:

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

where α is the learning rate, determining the extent to which new information overrides the old information, and γ is the discount factor, determining the importance of future rewards.

The algorithm repeats this process until convergence, meaning that the Q-values no longer change significantly. The optimal policy π^* can then be derived by selecting the action with the highest Q-value for each state:

$$\pi^*(s) = \arg\max_a Q(s, a).$$

In reinforcement learning, specifically in Q-learning, the traditional Q-table method of storing the action-value function $Q(s, a)$ for each state-action pair becomes impractical for large state or action spaces due to the exponential growth of the Q-table. To overcome this limitation, a neural network can be employed to approximate the Q-table. A neural network can represent the Q-table by approximating the Q-values for each state-action pair. The primary idea is to use the neural network as a function approximator to estimate the Q-value function $Q(s, a; \theta)$, where θ represents the parameters (weights and biases) of the neural network. The network receives the state representation s as input, and the output layer provides the estimated Q-values for all possible actions in that state. Given a state s, the neural network performs a forward pass to compute the Q-values for all actions:

$$Q(s, a; \theta) = \mathrm{NN}(s)$$

During training, the neural network parameters θ are updated to minimize the difference between the predicted Q-values and the target Q-values through gradient descent.

As mentioned at the start of this chapter, traditional temporal difference (TD) methods such as Q-learning, rely on manually designed function approximators to estimate the value

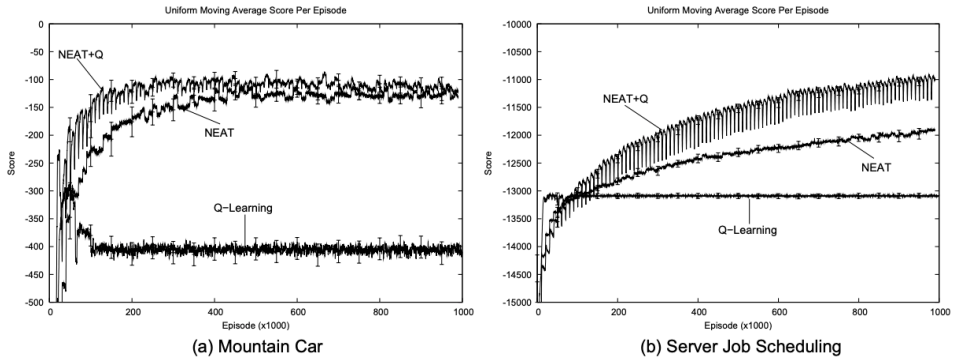

Figure 12.2: **Evolutionary function approximation.** Q-learning with a manually designed neural network is compared to both NEAT and NEAT+Q. Both NEAT method significantly outperform Q-learning in both the Mountain car (a) and server job scheduling domain (b). (Figure from Whiteson 2006)

function, which can be labor-intensive and suboptimal. An approach called evolutionary function approximation (Whiteson 2006), combines NEAT with Q-learning, resulting in the NEAT+Q algorithm. In a bilevel optimisation setup (see Chapter 11.2), NEAT evolves the structure and weights of neural networks in the outer level, while Q-learning updates these weights during the learning process in the lower-level optimization process. The aim in this combination is to allow the system to discover effective neural network configurations that are better suited for learning accurate value functions, thereby enhancing the performance of TD methods. Because Q-learning optimizes the weight of this network in the lower-level optimization algorithm, we have to make a choice about what to do with those modified weights in the outer-level. As we have seen previously (Chapter 4.2.3), we can either follow a Lamarckian approach, in which the weights updated by Q-learning are written back into the original NEAT genomes, or follow a Darwinian approach, where the weight changes are discarded and the original genomes are used to create the neural networks for the next generation. While the Darwinian approach is the more biologically plausible one, a Lamarckian approach could have potential benefits for RL tasks because the same learning doesn't have to be repeated for each generation. A Darwinian approach, on the other hand, could take advantage of the Baldwin effect, as we have seen previously in Section 4.2.3.

When comparing these methods in different domains such as the mountain car or server job scheduling, it becomes obvious that while Q-Learning learns a lot quicker in early epochs, performance soon plateaus (Figure 12.2). NEAT and Q-learning, on the other hand, continue improving with NEAT-Q significantly outperforming regular NEAT in both domains. Interestingly, if Q-Learning is starting out with one of the best networks evolved by NEAT, it is able to match the performance of NEAT+Q. Two example of such evolved networks are shown in Figure 12.3. The evolved networks are sparsely connected and irregular, suggesting that finding them through a manual process is unlikely to succeed.

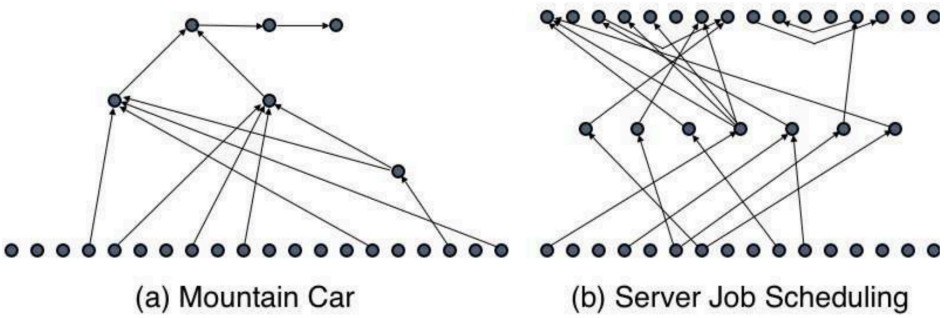

Figure 12.3: **NEAT+Q Evolved Networks Topologies.** Shown are the best neural network evolved by NEAT+Q for the mountain car (a) and server job scheduling (b). Inputs are shown at the bottom, while outputs are shown at the top. Each input is also directly connected to each output node (connections not shown). Output nodes can also be connected to other output nodes. The sparsity and irregularity of these networks suggests that they might be difficult to find through a manual process (Figure from Whiteson 2006)

12.2.3 Evolutionary Meta-Learning

The Q+NEAT approach from the previous section was an early example of what is now often called evolutionary meta-learning. In evolutionary meta-learning, an outer loop evolutionary optimization is typically tasked to find starting parameters for an inner loop optimization process with the goal of making a policy adaptable. This approach is closely related to bilevel optimisation (Chapter 11.2). However, the goal of the evolutionary meta-learning is often more specific: find starting parameters in the outer loop from which it is possible to adapt in a few optimisation steps to the tasks drawn from a task distribution in the inner loop.

This type of meta-learning was popularized by the influential work on model agnostic meta-learning (MAML) (Finn, Abbeel, and Levine 2017). While deep RL approaches have been shown to reach human or even superhuman performance in a variety of tasks, there is still a large gap to the learning efficiency of humans. Typical RL approaches require many trials to learn while humans can perform decently well on a variety of tasks with relatively little experience. The MAML approach tries to address this issue to enable more rapid adapatation to different tasks. However, the original MAML relies on second-order gradients, which makes it computationally intensive and sensitive to hyperparameters. Different versions of evolutionary meta-learning have since been developed to improve on the original MAML. For example, MAML-Balwin (Fernando et al. 2018) uses an evolutionary algorithm in the outer loop and RL in the inner loop, while ES-MAML (Song et al. 2019) uses an evolutionary optimizer in both the inner and outer loop. In this section, we will look at those variants in more detail.

What the evolutionary meta-learning methods have in common is that they try to exploit the Baldwin effect to evolve agents that can few-shot learn across a particular distribution of tasks. In this way, the objectives extends beyond helping to navigate difficult fitness landscapes, such as the ones encountered in the needle-in-the-haystack problem from earlier studies of the Baldwin effect (Figure 4.4). While it is theoretically possible to solve these

tasks without learning, here we are interested in tasks that would be impossible to solve with through evolution alone without phenotypic plasticity. Consider, for instance, the scenario where the robots depicted in Figure 14.6 experience a malfunction, such as the loss of a sensor or a limb. Similarly, envision the rockets illustrated in Figure 6.1 encountering an engine failure or a neural network evolved to control one race car is put into another different but similar race car. When the environment changes suddenly, there is often no time to re-evolve a controller and in these circumstances, a standard feedforward network will completely fail. Here the agent has to adapt online to maintain performance.

Canonical tasks in this vain are half Cheetah Goal Direction and Goal Velocity, two high-dimensional MuJoCo locomotion tasks. In Goal Direction, the agent has to rapidly learn to run in a particular direction. In Goal Velocity, the agent has to learn to adapt its locomotion to match a given velocity. In both tasks, the agents have to learn quickly during their lifetime. Here, the usual genetic algorithm approach for optimizing neural network weights without lifetime learning can be compared to an evolutionary MAML version (MAML-Baldwin), in which the initial weights are evolved through a simple GA in the outer loop and an RL method (policy gradient method A2C) updates them in the inner loop (Fernando et al. 2018). During meta-training, different tasks (e.g. goal directions or target velocities) are sampled in the inner loop and the network needs to adapt to them only through reward feedback alone. This task would be easy if the network would receive the desired velocity or direction as input. However, in these domains this information is only provided in form of a reward to the RL algorithm. For the Goal Velocity task this reward is the negative absolute value between the agent's current velocity and the target velocity and the magnitude of the velocity in either the forward or backward direction for the Goal Direction task.

While a typical genetic algorithm fails to solve these tasks, MAML-Baldwin evolves agents that can quickly adapt their behavior based on the task requirements. For example, in only 30 simulated seconds, the robot can learn to change its velocity to the target velocity. The comparison between the goal velocity and goal direction tasks reveals an interesting difference. The goal direction task demands a significant shift in strategy, as it requires the agent to move forward in some episodes and backward in others. In this scenario, Lamarckian evolution tends to get trapped in a local optimum, where it can only move backward effectively. Conversely, Baldwinian evolution adapts more successfully to these varying tasks. In the goal velocity task, however, Lamarckian evolution performs better because the final velocity achieved in the previous task often provides a suitable starting point for the target velocity in the next task because the target velocity is increased by 0.2 each episode.

The approaches we saw so far, including evolutionary meta-learner MAML-Baldwin, still relied on a policy gradient method in the inner loop. However, particularly when dealing with real robots, the noise present in the real world presents challenges to method relying on gradient estimates since even small differences due to initial conditions, noise in the sensors/actuators, etc. can lead to very different trajectories. It would thus be desirable to also being able to use the more robust evolutionary optimisation approach in the inner loop. However, one requirement is that the inner loop optimization should be data efficient because meta-learning is generally expensive.

ES-MAML (Song et al. 2019) provides such a mechanism. Compared to the original MAML, ES-MAML is conceptually simple, does not required estimating any second derivatives, and is easy to implement. An ES-MAML variant particularly suited for noisy

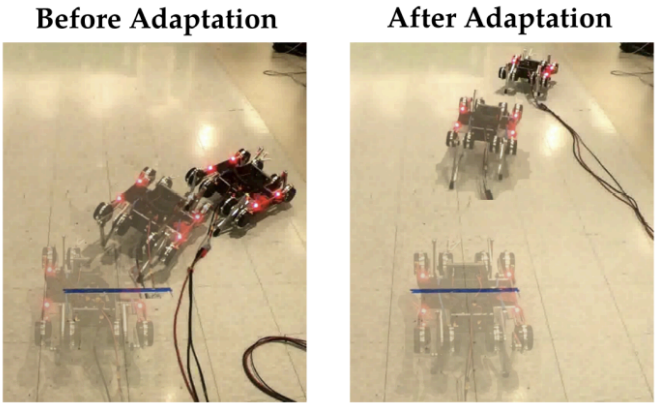

Figure 12.4: **Quick Adaptation through Evolutionary Meta-Learning.** The evolutionary meta-learning approach ES-MAML allows a robot only trained in a simulated environment to transfer to the real world and adapt to changes not seen during training, such as reduced motor power and an added payload of 500g placed to the robot's side. (Figure from Song et al. 2020)

domains performs evolution strategy on the initial network parameters in the outer loop and then a simple batch hill-climb algorithm in the inner loop (Song et al. 2020). Hill climbing in ES-MAML involves starting with an initial set of model parameters and then iteratively making small, random perturbations to these parameters. After each perturbation, the modified parameters are evaluated based on their performance on the current task. The algorithm then compares the performance of the modified parameters to that of the previous ones. If the performance improves, the algorithm accepts the new parameters; if not, it rejects them and reverts to the previous parameters.

This combination has shown to be particularly efficient, outperforming state-of-the-art MAML and allowing a quadrupedal robot only trained in a simulation to not only overcome the sim-to-real gap but to also adapt to changes in the real-world such as a (1) reduced motor power and added payload, and (2) a slippery surface. An example of the robot before and after adaptation is shown in Figure 12.4.

In sum, evolutionary meta-learning approach can exploit the Baldwin effect to produce powerful few-short learning agents, are often easier to optimise than their gradient-descent based alternatives, and can deal with noisy environments that methods based on gradient estimates can struggle with.

12.3 Evolving Neural Networks To Reinforcement Learn

In the previous sections we have reviewed a selection of hybrid approaches that combine RL and NE methods. While these synergistic combinations have proven very useful, they still mostly rely on domain agnostic learning approaches that can take many trials to learn. Additionally, the aforementioned meta-learning approaches are designed to quickly learn new tasks but struggle to continually learn; that is, learning new tasks without forgetting

what was previously learned. Finally, animals are born with innate priors that facilitate fast learning, which go well beyond the current MAML-like paradigms of only learning good starting weights. For example, a newly hatched chick orients itself towards moving objects right from birth, before any learning takes place (Versace et al. 2018). This evolved prior subsequently helps the animal to quickly and robustly learn to recognize complex objects under varying points of view, abilities our current AI systems still struggle with.

In this section we show that neural networks by themselves can be evolved to start with useful priors and the ability to perform reinforcement learning during their lifetime. This ability can enable them to deal with environments with non-stationary rewards and sudden environmental changes. While evolution is a relatively slow process that allows capturing gradual environmental changes, learning enables an individual to adapt to changes that happen during its lifetime. However, evolving these learning abilities is difficult not only because the neural network needs to learn which connections to change during the lifetime but also when to change.

One way that neuroevolution can allow agents to learn is to create recurrent connections in the network, which enables them to maintain information through feedback lopes. For example, in the T-maze navigation domain in Section 6.1.5, NEAT was able to evolve a recurrent network that was able to to keep information about the high reward location from one trial in the maze to the other. More complex recurrent networks, such as LSTMs, have been the main workhorse of machine learning methods that learn to reinforcement learn (J. X. Wang et al. 2016).

However, recurrent neural networks are not the only way that artificial agents can adapt quickly. Several different learning mechanisms are reviewed in this section, from more simple local Hebbian learning, to more advanced methods such neuromodulation that allow more precise control over plasticity. We also explore how to combine the ideas of plasticity with indirect encodings, reviewing the adaptive HyperNEAT approach. Finally, we look at approaches that extend neural networks with an external memory to further separate adaptation and control, which allows them more easily to evolve the ability to continually learn.

Later in this book, when we go into more detail on what neuroevolution can tell us about biological evolution (Section 14.4), we will return to the questions of how learning, development, and evolution interact and how much intelligent behavior is innate vs how much is learned.

12.3.1 Evolving Hebbian Learning Rules

A way to allow evolved neural networks to learn during their lifetime is to not only evolve the networks weights but also rules how those weights should change based on incoming and outgoing activations, inspired by the plasticity in biological nervous systems. The idea that all connection weights are genetically determined is unlikely to happen in nature, where information is compressed and thus initial weight values are likely not precisely encoded in the genome. The most well-known such rule, which we already encountered in Chapter 4.2, is Hebbian learning. This mechanisms is named after psychologist Donald Hebb and often summarised as: "Cells that fire together wire together." In mathematical terms, this can be written as: $\Delta w_{i \to j} = \eta x_i x_j$, where Δw is the change in weight from neuron i to neuron j is

based on the activation between them (x_i and x_j). The learning rate η for each connection can be evolved, allowing evolution to optimize the necessary degree of plasticity.

Pioneering work in evolving such *plastic neural networks* was performed by the labs of Dario Floreano and Stefano Nolfi (Nolfi and Floreano 2000) who studied evolving controllers for simulated and real robots, a field called evolutionary robotics. In one of their seminal works Floreano and Mondada 1996b trained a real miniature mobile robot to navigate a simple maze. Instead of evolving the weights directly, which are initialized to small random values at the start of a robot's deployment, a genetic algorithm determines which of four possible learning rates η (0.0, 0.3, 0.7, 1.0) each synapse in the network should have. In addition, the genome also encoded which type of four Hebbian learning rule variations should be applied at each synapse. These rules included: (1) a simple Hebbian rule, (2) a postsynaptic rule, in which the weight is decreased if the postsynaptic unit is active and presynaptic is not, (3) a presynaptic rule, which decreases the weight when the presynaptic neuron is active and the postsynaptic not, and (4) a covariance rule in which the weight is decreased if the activate difference between pre and postsynaptic neuron is below a given threshold, and otherwise increased. The weights of the network were updated every 300 ms following the synapse-specific evolved rule.

Info Box: The journey to a PhD in Neuroevolution

I (Sebastian Risi) first encountered neural networks during my undergrad studies in Germany in 2002. There was no course on neuroevolution (or even evolutionary algorithms) at my university but my interest really got piqued when I got my hands on the Evolutionary Robotics book by Nolfi & Floreano. Back then I had to really convince my professor to let me write a Diploma thesis about this niche topic. During my research for the thesis, I encountered Ken Stanley's & Risto's work on NEAT and was blown away. Why not let evolution decide on everything, including the structure of the network! At this point I basically knew I wanted to pursue a PhD in this direction and below is an excerpt of the email I wrote Ken in November 2007:

"I recently graduated from the Philipps-University Marburg in Germany with a master's degree in Computer Science. I am wondering if you have any PhD positions available in the area of Neuroevolution for video games or a related field. Especially the NERO project and your publications about Neuroevolution of Augmenting Topologies have drawn my attention.

My research interests focus on Artificial Intelligence, Neural Networks, Genetic Algorithms and biologically inspired computational methods in general. My curriculum vitae can attest to my extensive experience in these areas.

I am highly interested in further investigating the nature of systems that allow phylogenetic and ontogenetic adaptation and that display neural development. I think that the evolution of adaptive Neural Networks that are able to learn online can be used to create totally new game experiences going beyond the nature of classical video games.

I am looking forward to hear from you. Thank you for your consideration."

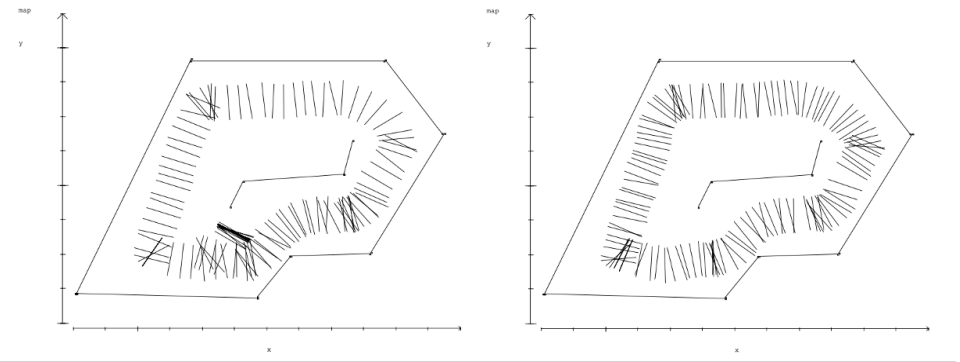

Figure 12.5: The navigation of the robot before (left) and after lifetime learning (right). The evolved learning rules allow the robot to quickly learn to navigate a maze without colliding with the walls (Figures from Floreano and Mondada 1996b)

Even though, in retrospect, the sentence *"My curriculum vitae can attest to my extensive experience in these Areas."* was probably stretching it a bit, Ken decided to hire me as a PhD student and we got to work together on methods for evolving more capable plastic neural networks and many more. We also started a company called Finchbeak together in which we worked on the Petalz video game (we discussed the technology behind it in Chapter...), which made around 200$ in profit (from which we had to subtract large accountant bills...) but was a great learning experience.

And somehow my interests didn't change so much in all these years! The same way I got inspired by Floreano's & Nolfi's Evolutionary Robotics book, I hope this book might inspire somebody else to join us in this exciting research field.

While the employed plastic networks are tiny compared to current networks (they have 27 connections in total, with eight infrared sensors, one hidden neuron, and two motor output neurons), the evolved rules enable the networks to quickly "learn" how to navigate during their lifetimes, even from completely random weights. In less than 10 sensor-motor loops the best-evolved individuals were able to move forward without getting stuck at walls (Figure 12.5). Analyzing the evolved solutions shows that there isn't one particular learning rule that appears more often in these networks. However, the basic Hebbian rule is not used frequently, which is likely due to the fact that it lacks the capability to decrease synaptic efficacy, potentially hindering future adaptability. It is also interesting to note that, while the behavior of the robot is stable and it can perform navigation without colliding with walls, the weights of these networks continuously change during navigation. This is in stark contrast to most other networks we encountered in this book, including networks trained through methods such as reinforcement learning. In these *fixed* networks the weights do not change during inference and only during a dedicated training period. Plastic neural networks thus take us a step closer to biological neural networks, which undergo continual changes throughout their whole lifetimes.

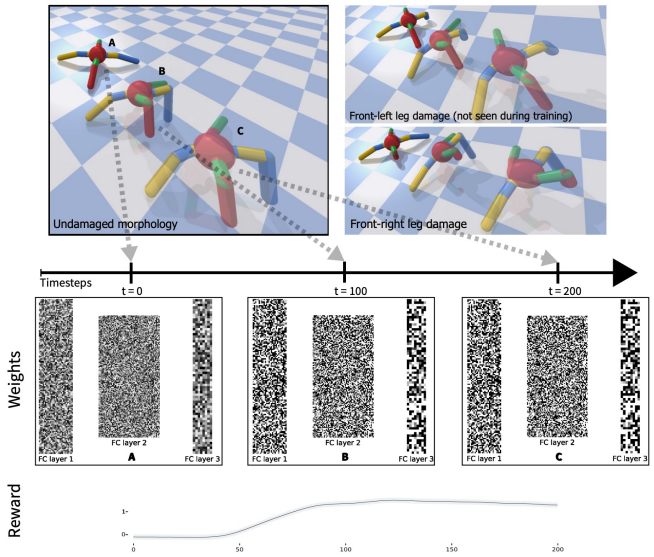

Figure 12.6: **Dynamics in Random Networks with Synapse-specific Hebbian Plasticity.** The evolved Hebbian rules allows the controller to quickly learn to control a quadrupedal robots, starting from randomly initialised starting weights. The figure shows the networks are three different timesteps (A, B, C) during the lifetime of a robot with the standard morphology. The change in the initially random weights quickly, which is drive purely by the learned Hebbian rules, is reflected in the increase in the reward performance (bottom). Even when the morphology of the robot changes through damage to one of the legs (top,right), the same Hebbian network is able to adapt in a few timesteps, allowing the robot to continue locomoting. (Figures from Najarro and Risi 2020b)

Najarro and Risi 2020b have presented an approach that allows these plastic neural networks to tackle more complex problems by building on the recent advances in scaling evolution strategies to evolve networks with millions of weights (Chapter 12.4). This combination allows us to navigate a larger genomic search space for plastic neural networks, which does not only include increased network sizes but also more general plasticity rules. While we were previously limited to only choosing from a set of four discrete Hebbian rules, the generalised Hebbian rule employed here enables each connection to implement its very specific weight update:

$$\Delta w_{ji} = \eta [A o_j o_i + B o_j + C o_i + D], \tag{12.34}$$

where w_{ji} is the weight between neuron i and j, η is the learning rates, correlation terms A, presynaptic terms B, postsynaptic terms C, constant D, with o_i and o_j being the presynaptic and postsynaptic activations respectively. We thus have a total of five parameters (η, A, B, C, D) per connection, which are optimized through a variation of OpenAI's ES algorithm (Chapter 2.1.5).

These more complex plastic neural networks allow us to tackle problems that are very difficult or even impossible to solve for standard feed-forward networks. For example, it is

interesting to evaluate how well a plastic controller works compared to a standard network when environmental circumstances change unexpectedly. To this end, a standard 3-layer feedforward network with [128, 64, 8] nodes per layer (totaling 12,288 trainable weight parameters) was compared to a plastic neural network with the same architecture in which only the plasticity parameters were evolved (totaling $12{,}288 \times 5 = 61{,}440$ Hebbian coefficients). Three different versions of a quadruped robot were devised to simulate the impact of partial damage to one of its limbs. Fitness was determined as the average distance covered by two versions of the robot, one in its standard form and the other with damage to its right front leg. The third version, which has damage to its left front leg, is excluded from the training process to later assess the networks' ability to generalize. It is important to note that the robot did not have access to a reward (i.e. distance traveled) during its lifetime.

While a feed-forward static neural network often works well on the morphologies it is trained on, it fails when confronted with the new robot morphology not seen during training. The evolved plastic network, on the other hand, quickly finds network weights that allow high performance in these more complex domains, even when starting from completely random weights in each episode. Additionally, the Hebbian approach is able to adapt to damages in the quadruped such as the truncation of the left leg, which it has not seen during training (Figure 12.6). Instead of needing many thousands learning steps as is common in standard reinforcement learning approaches that start from tabula rasa, the evolved Hebbian learning rules allow the neural network to reach high-performance after only 30 – 80 timesteps. Interestingly, the Hebbian network achieves this performance across the three different morphologies, all without the network receiving any reward-based feedback. The incoming activation patterns during the lifetime are sufficient for the network to self-adjust, even without explicit knowledge of the specific morphology it is simulating.

12.3.2 Learning when to learn through neuromodulation

Hebbian learning is far from the only adaptation mechanism in the brain. Another mechanisms is neuromodulation, which plays many different roles in biological nervous systems. Neuromodulation refers to the process by which neural activity is regulated or modified by neurotransmitters and other chemicals within the brain and nervous system. This process can influence various aspects of neuronal function, including the strength and efficacy of synaptic connections, the excitability of neurons, and overall neural network dynamics. Neuromodulation plays a crucial role in the brain's ability to adapt to new information, experiences, and environmental changes, affecting learning, memory, mood, and behavior.

Given the numerous functions of neuromodulation in biological nervous systems, it has also been incorporated in evolving plastic neural networks. In these instances, neuromodulation is typically set to modify the Hebbian plasticity of neurons in the neural network. This ability is useful because it allows switching plasticity "on" and "off", enabling reward-mediated learning. For example, plasticity of some weights might be switched off if they were responsible for obtaining a high reward in the environment while other connection should increase their plasticity when the reward is lower than what was expected. In a pioneering demonstration of this idea (Soltoggio et al. 2008) used an approach similar to NEAT, in which structural mutations during evolution could not only insert and delete standard hidden nodes but also neuromodulatory nodes. In contrast to standard neural networks,

in which each node has the same type of effect on all the nodes it is connected to, in a neuromodulated network each node i calculates both a standard activation a_i and a modulatory activation m_i as follows:

$$a_i = \sum_{j \in Std} w_{ij} o_j,$$

$$m_i = \sum_{j \in Mod} w_{ij} o_j,$$

where w_{ij} is the strength of the connection between node i and j, and o_j is the output of the postsynaptic neuron, which is calculated based on the standard activation $o_j(a_j) = tanh(\frac{a_j}{2})$. In contrast to how pure Hebbian plasticity was modeled as $\delta_{ji} = \eta[Ao_j o_i + Bo_j + Co_i + D]$ we are now making the weight change also dependent on the calculated modulatory activation m_i: $\Delta w_{ji} = tanh(\frac{m_i}{2})\delta_{ji}$. A graphical interpretation of this idea is shown in Figure 12.7a. The activation of modulatory neuron $Mod0$ affects the plasticity of the connections $w_{1,4}, w_{2,4}, w_{3,4}$ that are connected to the neuron being modulated. In contrast to the approach by Najarro and Risi 2020b presented in the previous section, here every connection shares the same evolved parameters, A, B, C, and D.

The approach was tested on a task often used for cognitive experiments with animals, the single and double T-Maze. This task involves correctly turning once or twice to reach a high or low reward, and then returning to the starting point. Each agent underwent 100 trials, during which the location of the high reward was switched to another arm of the maze, requiring the agents to adapt. This approach allowed testing if neural architectures that can include neuromodulation would be favored by evolution and provide computational advantages in tasks requiring adaptation. The results on a (double) T-Maze navigation domain support this (Figure 12.7)d. While the Hebbian networks perform on par with the neuromodulatory network on the single T-Maze task, the neuromodulatory approach shows clear advantages in the double T-Maze when the learning and memory requirements of the task increase (Figure 12.7)d.

Figure 12.7d shows an example of an evolved neuromodulatory network. While it is difficult to analyse exactly what is going on in these networks, further experiments revealed that switching neuromodulation off in networks that were evolved with it, resulted in networks that could still navigate without crashing but they didn't show any useful adaptive behavior anymore (e.g. remember the position of the high reward from one trial to the next). This observations seems to suggests that neuromodulation helps in separating the neural circuit responsible for control from the one responsible for adaptation. In Chapter 12.3.4 we look at an approach that models this separation more explicitly, by giving neural networks an external memory component.

12.3.3 Indirectly encoded plasticity

A challenge with the previously mentioned approaches to encode plasticity is that the local learning rules for every synapse in the network must be discovered separately by evolution. However, similar to how connectivity patterns in brain follow certain regularities, the distribution of plasticity rules across a neural network likely would also benefit from such regularities as well.

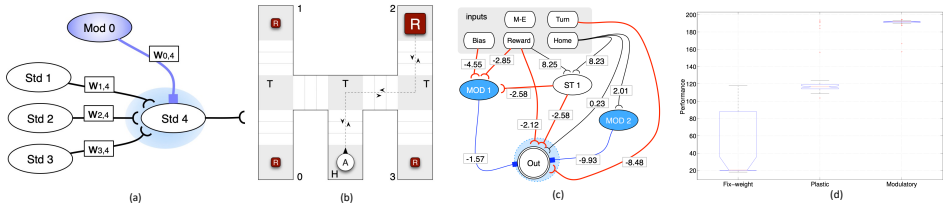

Figure 12.7: **Evolving Neuromodulated Plasticity** (a) In contrast to standard neural networks, in a neuromodulated network two different types of neurons (Mod=modulatory, Std=standard) interact with each other. Here, a modulatory neuron (Mod 0) is modulating the plasticity of synapses ($w_{1,4}, w_{2,4}, w_3, 4$) connecting to its target neuron (e.g. Std 4). (b) In the double T-maze the agent has to remember the position of a high reward, which can switch from trial to trial, and also learn to navigate back to the start after collecting it. The example of an evolved solution neural network is shown in (c). The networks that can incorporate neuromodulatory neurons perform significantly better than fixed-weight networks and purely Hebbian networks on the double T-Maze (d). (Figures from Soltoggio et al. 2008)

It turns out that the HyperNEAT approach we introduced in Chapter 4.3.3 to indirectly encode weight patterns can be generalised to also indirectly encode the plasticity of a network. As in the brain, different regions of the ANN should be more or less plastic and employ different learning rules, which HyperNEAT allows because it sees the geometry of the ANN. The main idea behind this approach, which is called adaptive HyperNEAT (Risi and Stanley 2010), is that CPPNs in HyperNEAT can not only encode connectivity patterns but also patterns of plasticity rules.

A straightforward way to enable HyperNEAT to indirectly encode plastic network is to augment the CPPN to not only produce each connections' weight, but additional connection-specific parameters such as learning rate η, correlation term A, presynaptic factor B, and postsynaptic factor C. When a policy network is initially decoded, it stores these parameters and the connection weights for each synapses and then updates the weight during its lifetime following this simplified version of the generalised Hebbian learning rules:

$$\Delta w_{ij} = \eta \cdot \left[A o_i o_j + B o_i + C o_j \right] .$$

This approach is able to solve a single T-Maze task, demonstrating that HyperNEAT is, in fact, able to distribute plasticity coefficients in a geometric manner. However, Adaptive HyperNEAT is clearly overkill for such simple domains and we have seen simpler approaches such as directly-encoded Hebbian learning or LSTMS (Chapter 6.1.5) being able to do the same. However. things become a bit more interesting, if we not only allow Adaptive HyperNEAT to encode these learning rule coefficients but enable it to evolve completely new learning rules itself. This more general Adaptive HyperNEAT model augments the four-dimensional CPPN that normally encodes connectivity patterns with three additional inputs: presynaptic activity o_i, postsynaptic activity o_j, and the current connection weight w_{ij}. That way, the synaptic plasticity of a connection between two two-dimensional

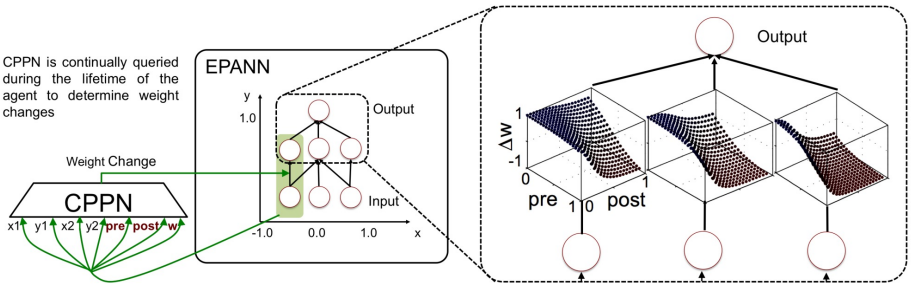

Figure 12.8: **Adaptive HyperNEAT.** In Adaptive HyperNEAT, the CPPN is queried each time step, given the location of nodes but also the current weight of the connection and the activity of the pre and postsynaptic neurons. This way, each connection in the network can learn arbitrary learning rules that can be geometrically encoded by the CPPN. (Figures from Soltoggio, Stanley, and Risi 2018)

points (x_1, y_1) and (x_2, y_2) can be described by:

$$\Delta w_{ij} = CPPN(x_1, y_1, x_2, y_2, o_i, o_j, w_{ij}).$$

Instead of only being queried at the beginning of an episode, here the CPPN is queried at every timestep to update the weights of the neural network. The same CPPN that thus decides on the initial weights and network connectivity is now also responsible for how to change the network, taking into account both the location and activity of the network's neurons.

A simple, yet effective domain to test the effectiveness of this method is a variation of the T-Maze domain with a non-linear reward encoding. That is, in this domain the agent receives a high reward for rewards with "color" input values 0.3 and 1.0 but a low reward for 0.1 and 0.8. Because the agent was given a network with no hidden nodes (which is not able to learn this nonlinearity), evolution needed to discover a CPPN that instead encodes the appropriate non-linear learning rules. And indeed, this more general Adaptive HyperNEAT version was able to solve the task while a normal Hebbian network and the simpler Adaptive HyperNEAT (which outputs the Hebbian learning co-efficient) failed. Interestingly, in this domain the discovered learning rules smoothly change with the location of the presynaptic node as shown in Figure 12.8.

Adaptive HyperNEAT can also be combined with the evolvable substrate approach (Chapter 4.3.4) to alleviate the experimenter from deciding on the number of nodes. For the first time, this unified approach called adaptive evolvable-substrate HyperNEAT (Risi and Stanley 2012a), was able to fully determine the geometry, density, and plasticity of an evolving neuromodulated ANN. While the aforementioned tasks these methods have been applied to so far are rather simple, they did demonstrate the CPPNs ability to learn arbitrary learning rules that allow and agents to quickly adapt to changes in its environment. The idea of learning to learn has since become a larger focus of the wider machine learning community but the groundwork was laid by NE methods. Scaling this approach up to work with larger networks and for more complex tasks is an exciting future research direction.

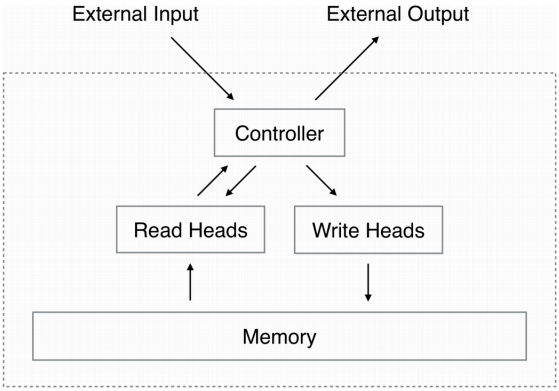

Figure 12.9: In a Neural Turing Machine (NTM), a neural network (the controller) is augmented with an external memory component that it can learn to read from and write to through dedicated read and write heads. The external memory allows the network to store information over many time steps and use it to learn algorithms such as copy, sort, or associative recall. (Figures from Graves, Wayne, and Danihelka 2014)

12.3.4 Learning to continually learn through networks with external memory

A major challenge in AI in general, and in evolving plastic neural networks in particular is continual learning. That is, learning new tasks or knowledge without forgetting what was previously learned. Most current neural networks struggle with this and suffer from a symptom called *catastrophic forgetting*, where they can learn a new tasks but forget the tasks they learned previously. While the mechanism of how our brains perform continually learning so well are not all yet understood,

[FIGURE OF SEASON TASK]

A memory-augmented Neural Network Graves, Wayne, and Danihelka 2014 is a neural architecture in which the circuit for control and the mechanism for adaptation are separated by design. In addition to learning through changes in connection strength or activations (such as in LSTMs), modelling memory directly offers another way for agents to adapt and remember. One realization of this type of memory-augmented neural network is the Neural Turing Machine (NTM) Graves, Wayne, and Danihelka 2014. The NTM combines traditional neural networks with the concept of a Turing machine, aiming to enhance the capability of neural networks by giving them the ability to read from and write to an external memory module. This fusion allows the NTM to not only process data through its neural network structure but also store and retrieve data, enabling it to perform tasks that require memory and are difficult for conventional neural networks. Just like LSTMs, NTMs are designed to handle long-range dependencies in data. LSTMs achieve this through their gating mechanisms (input, output, and forget gates) that regulate the flow of information, allowing the network to maintain or forget information over long intervals. Similarly, NTMs can maintain data over long periods using their external memory bank, albeit in a more explicit and controllable manner.

An overview of the NTM architecture is shown in Figure 12.9. At the heart of an NTM is a neural network that acts as the controller. This controller operates like any other neural

network, processing inputs and generating outputs. However, unlike standard neural networks, it interacts with an external memory bank through read and write heads, directing the read and write operations. The primary advantage of NTMs is their ability to perform tasks that require complex manipulation of data sequences or the execution of algorithms that conventional neural networks struggle with. This includes problems like sorting lists, simple arithmetic, or even executing simple programs.

The original NTM is designed to be completely differentiable, including the read and write mechanisms. This means the NTM can be trained end-to-end using gradient descent and backpropagation, similar to conventional neural networks. However, the differentible architecture of the NTM comes at the cost of access the entire memory content at each step, and has a fixed memory size, makes this approach inefficient for larger memory banks. Additionally, because of the attention is "soft", small errors can accumulate and the approach did not always generalise perfectly to e.g. copying long sequences.

An exciting direction is to train the NTM instead through NE, which does not only allow hard attention and potential better generalization, but the approach can also be directly applied to reinforcement learning-like problems that do not require input-output examples. In a demonstration of this idea, Greve, Jacobsen, and Risi 2016 presented an evolutionary NTM, in which both the structure and the weights of the network were evolved by NEAT.

This model features a theoretically unlimited memory capacity, capable of storing vectors of size M at each memory slot. It operates with a single, unified head for both reading and writing. In more detail, the ENTM executes four primary operations:

1. **Write:** A write interpolation parameter dictates the blending of the current memory vector at the head's location with a new write vector. This is calculated as follows:

$$M_{t+1}(h) \leftarrow M_t(h) \cdot (1 - w_t) + a_t \cdot w_t,$$

   where $M_t(h)$ represents the memory vector at the head's location at time t, w_t is the write interpolation weight, and a_t is the write vector.
2. **Content Jump:** If the output for content jump exceeds a certain threshold (0.5 in this case), the head jumps to a position on the memory tape most akin to the write vector, determined by an Euclidean distance metric in this implementation.
3. **Shift:** The controller can shift the memory head either to the left or right from its current position or maintain the position based on the highest activated shift output among the three provided.
4. **Read:** Following any content jumps and shifts, the content of the memory vector at the final location of the head is automatically fed into the neural network at the start of the next cycle.

To extend the functionality of NTMs to include reward-based learning tasks by adopting a neuroevolutionary strategy, where both the network's topology and weights are evolved. Initially replicating the copy tasks described by Graves et al. the evolutionary NTM is significantly less complex than the original, excels at generalizing to longer sequences in the copy task, and is not constrained by fixed-size memory banks.

We can compare the evolutionary NTM vs. the original NTM version trained through gradient descent on the copy task. In this task, the neural network must memorize and retrieve

a lengthy sequence of random binary vectors. The network receives an initial bit indicating the start of the task, followed by a sequence of random binary vectors, and then a delimiter bit that marks the beginning of the recall phase. Since NEAT begins with basic networks and progressively introduces nodes and connections, it can find a sparsely connected champion network that utilizes just a single hidden neuron. This evolved network is significantly smaller in size compared to the original NTM, which features full connectivity, 100 hidden neurons, and a total of 17,162 parameters. Additionally, and in contrast to the original NTM, the evolved networks generalize perfectly to long sequences.

Another benefit of having an external memory is that it can help in tasks requiring continual learning. While it can be difficult to learn new information in an LSTM or Hebbian network during the lifetime of the agent without catastrophic forgetting of previous information, it is straightforward to tackle this challenge with an expanding external memory (where new information can be put in an unused location in memory). A task to test the ENTM for continual learning is the season task introduced by Ellefsen, Mouret, and Clune 2015, in which the agent must learn to identify and remember which food items are nutritious and which are poisonous across different seasons, with the challenge increasing as the food items and their properties change from one season to another. The task tests the agent's ability to withstand catastrophic forgetting and learn new associations while retaining old ones, leveraging the external memory capabilities of the ENTM.

The champion network can learn new associations in a single trial without forgetting previously learned ones (Figure 12.10b) and can generalize almost perfectly to sequences it has never encountered before (achieving a test score of 0.988). The network stores information about the food items in four memory locations—two for each season (Figure 12.10c). Initially, the agent ignores all food items. However, after being penalized for neglecting nutritious items, it begins to remember the ones it missed and must consume in the future. Each nutritious item is stored in a separate memory location, resulting in the use of all four locations. This memorization process is achieved by linking the punishment input to the write interpolation output.

12.3.5 Exercises

Evolve the Hebbian parameters for a maze-navigating agent. Compare this agent to one controlled by a static neural network when the environment changes gradually during evolutionary time, versus a sudden change during the lifetime of the agent. Compare Lamarckian vs. Darwinian setups for different environments.

12.4 Scaling Up

Many deep learning algorithms, such as deep reinforcement learning, have benefited greatly from rapid training of neural networks on hardware accelerators such as GPUs, and thus shorter iteration times. Previously, these advances have been tailored to algorithms based on gradient descent but the NE community has been developing their own frameworks, constantly narrowing this gap. Here we will first have a look at how NE approaches such as ES and GAs have benefited from being parallelized across many CPUs and how their scaling compares to RL methods. Following, we briefly have a look at the emerging area of

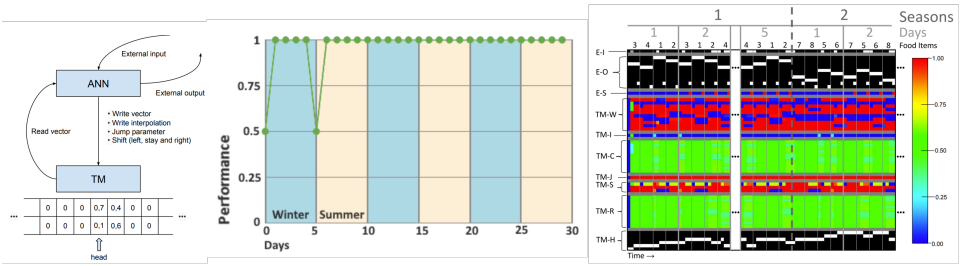

Figure 12.10: Evolvable Neural Turing Machine. (a) The evovable NTM is characterized by a hard attention mechanism and a theoretically infinite memory tape. (b) The NTM is able to learn new associates in one-shot without forgetting previously learned ones. (c) Days 3 and 4 of Season 1, as well as all days beyond Day 2 in Season 2, are not displayed but are completed flawlessly. Legend: E-I: ANN output indicating whether the food item should be consumed. E-O: ANN inputs from the environment: summer item (1–4), winter item (5–8), reward (9), punishment (10). E-S: Score indicator. TM-W: Write vector. TM-I: Write interpolation. TM-C: Content of the tape at the current head position after writing. E-J: Content jump input. TM-S: The three shift values in descending order: left, none, right. TM-R: Read vector. TM-H: Current head position after control operations. (Figures from Lüders, Schläger, and Risi 2016)

hardware-accelerated NE, which is poised to change the scale of neural networks NE can operate on.

Similarly to many classical machine learning algorithms such as reinforcement learning, neuroevolution approaches have shown to work far better when scaled to vastly larger computing resources. In a recent demonstration of this fact, Salimans et al. 2017 showed that an ES for evolving neural network parameters can be made to scale effectively with the number of available CPUs, by employing a communication strategy based on common random numbers that minimizes the amount of data needing to be exchanged between workers. ES is particularly amendable to parallelization because of its low bandwidth requirements, which is in stark contrast to methods based on backpropagation that require communicating complete gradients. By scaling ES to thousands of parallel workers, neural networks for complex tasks like 3D humanoid walking can be found in just 10 minutes and competitive results on Atari games can be achieved within an hour of training. This work further highlights some of the advantages of ES over traditional RL methods, such as being invariant to the frequency of actions, able to handle tasks with long time horizons without the need for temporal discounting, not requiring value function approximation or backpropagation, and performing well even with sparse or delayed rewards. Additionally, the experiments demonstrated that the slightly lower data efficiency of ES versus RL can often be mitigated in practice because of the lower compute requirements that are a result of not having to perform backpropagation and not needing a value function.

Because ES estimates a gradient in parameter space and then takes a step in that direction (Chapter 2.1.4), it can be seen as a type of gradient-descent method. For this reason, the larger ML community was still skeptical at the time that pure gradient-free evolutionary

algorithms (e.g. genetic algorithms) can operate at the scale of deep neural networks. However, this changed when Such et al. (2017) – in a surprising demonstration of the power of artificial evolution – showed that it is possible to optimize a deep convolutional neural network with over four million free parameters to play Atari games from pixels through an extremely simple GA alone. This approach uses truncation selection, where the top T individuals become the parents for the next generation, and elitism (copying the best individual unmutated to the next generation). Because the Atari environments are noisy, each of the top 10 individuals in their setup was evaluated on 30 additional episodes to get a better estimate of their true performance. To produce offspring, a parent was selected uniformly at random and its parameter vector θ was mutated by applying additive Gaussian noise:

$$\theta^t = \theta + \sigma\varepsilon \quad \text{where} \quad \varepsilon \sim \mathcal{N}(0, I).$$

That's it! No crossover, no evolving topologies, and no indirect encoding. This simple approach showed competitive performance to RL algorithms across a variety of different Atari games. Interestingly, no clear winner between RL and NE emerged but instead, performance differences are very much dependent on the game at hand. Among the 13 games tested, DQN, ES, and the GA each achieved the highest score on three games, while the RL method A3C achieved the top score on four games. Notably, in the game Skiing, the GA achieved a score higher than any previously reported at the time, surpassing a variety of different DQN variants. In some games, the GA's performance significantly exceeds that of DQN, A3C, and ES, particularly in Frostbite, Venture, and Skiing. When allowed to run six times longer (6B frames), scores improved across all games. With these post-6B-frame scores, the GA outperforms A3C, ES, and DQN in head-to-head comparisons on 7, 8, and 7 out of the 13 games, respectively. A summary of the results across many Atari games can be seen in Table 12.11. However, while a GA can efficiently find policies for many Atari games, it can struggle in other domains. For example, a GA took around 15 times longer than ES and still performed slightly worse when optimizing a neural network for humanoid locomotion. The reason for this difference might be that an ES algorithm has an easier time making precise weight updates than a GA, which could be critical for the intricate movements necessary for humanoid locomotion. Further research is needed to elucidate this issue in more depth.

Surprisingly, even a random search variation, which only evaluates randomly generated policies, can perform surprisingly well. While it does not outperform the GA on any of the games tested, which suggests that the GA is effectively optimizing over generations, it outperforms DQN on three games, ES on three, A3C on six. These results suggest that sometimes following the gradient (as is done in gradient-based optimization algorithms) can actually be detrimental to performance and it can be more efficient to do a dense search in some local neighborhood of parameters. It is also important to note that the policies found by this random search can be quite sophisticated. Consider an example from the game Frostbite, where an agent must execute a lengthy sequence of jumps across rows of icebergs moving in different directions (Figure 12.12). The agent must avoid enemies and can optionally collect food to build an igloo, brick by brick. Only after the igloo is completed can the agent enter it to receive a large reward. During its first two lives, a policy discovered through random search performs a sequence of 17 actions. This involves jumping down four rows of icebergs moving in various directions and then jumping back up three times to

	DQN	ES	A3C	RS	GA	GA
Frames	200M	1B	1B	1B	1B	6B
Time	~7-10d	~ 1h	~ 4d	~ 1h or 4h	~ 1h or 4h	~ 6h or 24h
Forward Passes	450M	250M	250M	250M	250M	1.5B
Backward Passes	400M	0	250M	0	0	0
Operations	1.25B U	250M U	1B U	250M U	250M U	1.5B U
amidar	**978**	112	264	143	263	377
assault	4,280	1,674	**5,475**	649	714	814
asterix	4,359	1,440	**22,140**	1,197	1,850	2,255
asteroids	1,365	1,562	**4,475**	1,307	1,661	2,700
atlantis	279,987	**1,267,410**	911,091	26,371	76,273	129,167
enduro	**729**	95	-82	36	60	80
frostbite	797	370	191	1,164	**4,536**	**6,220**
gravitar	473	**805**	304	431	476	764
kangaroo	7,259	**11,200**	94	1,099	3,790	**11,254**
seaquest	**5,861**	1,390	2,355	503	798	850
skiing	-13,062	-15,443	-10,911	-7,679	†**-6,502**	†**-5,541**
venture	163	760	23	488	**969**	†**1,422**
zaxxon	5,363	6,380	**24,622**	2,538	6,180	7,864

Figure 12.11: **Scores of ES and GA neuroevolution approaches on the Atari benchmark compared to RL.** Shown are game scores, with higher values showing better performance. Different methods perform best in different games. Even a random search variant (RS) can find policies that outperform policies found by DQN, A3C, and ES for some games. (Figure from Such et al. 2017)

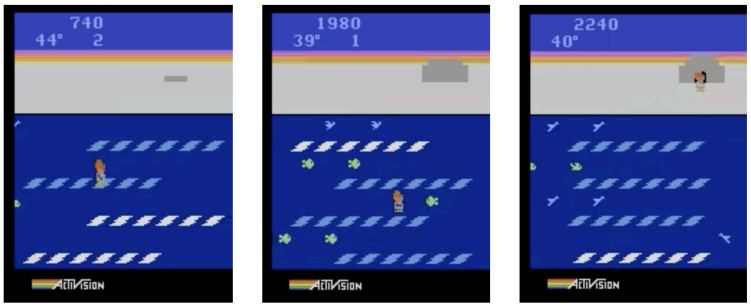

Figure 12.12: **Random Search Result on Frostbite.** A neural network found through random search performs a sophisticated strategy in the Atari game Frostbite and reaches a higher score than those produced by DQN, A3C, and ES. (Figure from Such et al. 2017).

construct the igloo. Once the igloo is built, the agent immediately moves toward it, enters it, and receives a large reward. This policy achieved a remarkably high score of 3,620 in less than an hour of random search, compared to an average score of 797 produced by DQN after 7-10 days of optimization.

That a simple GA performs so well in this domain demonstrates that methods that are easier to scale and can take advantage of inductive biases, such as convolutional architectures, can often perform surprisingly well. It is an interesting open question how well more complex indirect representations (Chapter 4) could perform when scaled up. Early small-scale experiments on evolving hypernetworks to playing Atari games have shown some mixed

results (Carvelli, Grbic, and Risi 2020), with hypernetworks performing worse in some and better in other games than directly evolving parameters with a GA. It is clear that the potential of indirect encodings is not yet fully realized and reaching state-of-the-art performance will likely depend on making these methods more scalable and potentially also using them in combination with more open-ended search methods (Chapter 9), which are likely better at navigating the more complex search space imposed by these indirect representations.

While NE algorithms have mostly relied on CPU parallelism in the past, the aforementioned work was also an early demonstration of the power of an NE approach that capitalizes on GPU acceleration. Even using only a single GPU, training can be significantly sped up. Since then more work has been done to further take advantage of distributed hardware-accelerated setups and the massive throughput provided by GPUs/TPUs. While distributing training across multiple CPUs can already give a substantial speedup, another level of training speed and network size can be reached by taking advantage of hardware acceleration. Deep learning methods in general, and RL methods in particular, have long been able to take advantage of training across a large number of sTPU and GPUs. In recent years, the advent of high-performance computing frameworks like JAX has also finally enabled such efficient hardware-acceleration for evolutionary algorithms. Two notable libraries that leverage JAX for evolutionary computation are EvoJAX (Tang, Tian, and Ha 2022) and EvoSAX (Lange 2023). For example, one of the important features of EvoJAX is its use of JIT compilation to optimize the evaluation of the fitness function. This ensures that the computationally intensive parts of the algorithm are executed as efficiently as possible. Additionally, EvoJAX supports vectorized operations, allowing simultaneous evaluation of multiple individuals, further enhancing performance.

It is interesting to imagine an alternate history where DeepMind used a GA instead of RL for their now seminal work on playing Atari from pixels (Mnih et al. 2015) and how it might have changed the AI research trajectory. Overall, the fact that NE and RL methods perform so differently in many games and domains points to the potential of hybridizing these methods to address their respective limitations, which we will further explore in the next Section.

12.4.1 Exercise on Scaling up NE

For this exercise, we will focus on scaling up NE, including getting familiar with some of hardware-accelerated NE frameworks.

12.5 Chapter Review Questions

1. **Reinforcement Learning vs. Neuroevolution:** What are the key strengths and weaknesses of reinforcement learning (RL) and neuroevolution (NE) when applied to optimization tasks? How do their approaches differ in handling sparse rewards and high-dimensional spaces?
2. **Evolutionary Reinforcement Learning (ERL):** How does Evolutionary Reinforcement Learning (ERL) combine evolutionary algorithms and deep reinforcement learning? What are the specific advantages of integrating these methods in tasks with sparse rewards?

3. **Replay Buffer in ERL:** What is the role of the replay buffer in ERL? How does it enable the algorithm to learn within episodes, unlike standard neuroevolution?
4. **NEAT+Q Approach:** How does the NEAT+Q algorithm integrate neuroevolution (via NEAT) with Q-learning? What are the advantages of this approach for evolving neural architectures in reinforcement learning tasks?
5. **Meta-Learning with Evolutionary Methods:** How does evolutionary meta-learning differ from traditional reinforcement learning? How does it exploit the Baldwin effect to enable few-shot learning across diverse task distributions?
6. **ES-MAML:** What makes ES-MAML particularly well-suited for meta-learning in noisy environments? How does it differ conceptually and computationally from gradient-based meta-learning methods like MAML?
7. **Evolving Networks to Reinforcement Learn:** What are the advantages of evolving neural networks capable of intrinsic reinforcement learning? How does this approach address the challenges of non-stationary rewards and environmental changes?
8. **Hebbian Learning Rules:** How does the evolution of Hebbian learning rules enable neural networks to adapt during their lifetimes? What are some limitations of using simple Hebbian mechanisms for complex tasks?
9. **Neuromodulation in Evolved Networks:** How does incorporating neuromodulation into evolved networks enhance their ability to learn and adapt? Why is neuromodulation particularly effective in tasks requiring memory and adaptation, such as the double T-Maze?
10. **Scaling Neuroevolution:** How have advancements in parallelization and hardware acceleration, such as GPU-based frameworks like EvoJAX, transformed the scalability of neuroevolution methods? What are the implications for optimizing large-scale neural networks compared to traditional reinforcement learning?