3 The Fundamentals of Neuroevolution

In the previous chapter, we have described a few evolution algorithms (EA) that can optimize the parameters of a function without the need to explicitly calculate gradients and we reviewed the basics behind different types of neural networks. In this chapter, we will explore how these algorithms can be combined to evolve neural networks.

Neural network models are highly expressive and flexible, and if we are able to find a suitable set of model parameters, we can use neural nets to solve many challenging problems. Deep learning's success largely comes from the ability to use the backpropagation algorithm to efficiently calculate the gradient of an objective function over each model parameter. With these gradients, we can efficiently search over the parameter space to find a solution that is often good enough for our neural net to accomplish difficult tasks.

However, as discussed in the previous chapter, a vast number of important problems we encounter in the world do not have readily, well-behaved differentiable objective functions available. This also includes the space of problems where the expressiveness of neural networks can add a lot of value, once properly trained.

In this chapter, we will explore how evolutionary algorithms can be applied to help find a suitable set of model parameters for a neural network to enable it to solve challenging and interesting problems, such as those encountered in reinforcement learning environments, or in classification tasks. We first start by exploring the application of evolution algorithms (EA) to some of these RL problems, and also highlight methods we can use to find policies that are more stable and robust.

3.1 Evolution Strategies for Reinforcement Learning

While RL algorithms require a reward signal to be given to the agent at every timestep, EA algorithms only care about the final cumulative reward that an agent gets at the end of its rollout in an environment. In many problems, we only know the outcome at the end of the task, such as whether the agent wins or loses, whether the robot arm picks up the object or not, or whether the agent has survived, and these are problems where EA may have an advantage over traditional RL. Below is a pseudo-code that encapsulates a rollout of an agent in an OpenAI Gym environment, where we only care about the cumulative reward:

```
def rollout(agent, env):
  obs = env.reset()
  done = False
```

```
  total_reward = 0
  while not done:
    a = agent.get_action(obs)
    obs, reward, done = env.step(a)
    total_reward += reward
  return total_reward
```

We can define `rollout` to be the objective function that maps the model parameters of an agent into its fitness score, and use an EA solver to find a suitable set of model parameters as described earlier:

```
env = gym.make('worlddomination-v0')
solver = EvolutionStrategy()          # use our favourite EA
while True:
  solutions = solver.ask()            # ask the EA to give set of params
  fitlist = np.zeros(solver.popsize) # create array to hold the results
  for i in range(solver.popsize):     # evaluate for each given solution
    agent = Agent(solutions[i])       # init the agent with a solution
    fitlist[i] = rollout(agent, env) # rollout env with this agent
  solver.tell(fitness_list)           # give scores results back to EA
  bestsol, bestfit = solver.result() # get best param & fitness from EA
  if bestfit > MY_REQUIREMENT:        # see if our task is solved
    break
```

Our agent takes the observation given to it by the environment as an input, and outputs an action at each timestep during a rollout inside the environment. In the following example, we use a simple feed-forward network with 2 hidden layers to map from an agent's observation, a vector x, directly to the actions, a vector y:

$$h_1 = f_h(W_1 x + b_1), \tag{3.17}$$

$$h_2 = f_h(W_2 h_1 + b_2), \tag{3.18}$$

$$y = f_{out}(W_{out} h_2 + b_{out}) \tag{3.19}$$

The activation functions f_h, f_{out} can be *tanh*, *sigmoid*, *relu*, or whatever we want to use. For the output layer, sometimes we may want f_{out} to be a pass-through function without nonlinearities. If we concatenate all the weight and bias parameters into a single vector called W, we see that the above neural network is a deterministic function $y = F(x, W)$. We can then use EA to find a solution W using the search loop described earlier.

But what if we don't want our agent's policy to be deterministic? For certain tasks, even as simple as rock-paper-scissors, the optimal policy is a random action, so we want our agent to be able to learn a stochastic policy. One way to make a deterministic policy network into a stochastic policy network is by making the final layer is a set of μ and σ parameters and the action is sampled from $N(\mu, \sigma I)$. Adding such randomness to the output also helps to encourage the agent to explore the environment and escape from local optima.

3.2 Evolving Robust Policies for Bipedal Walker

NE is particularly well suited to searching for robust policies. We want to control the tradeoff between data efficiency, and how robust the policy is over several random trials. To demonstrate this, we go over an example using the environment called `BipedalWalkerHardcore` available inside OpenAI gym, which uses the Box2D Physics Engine.

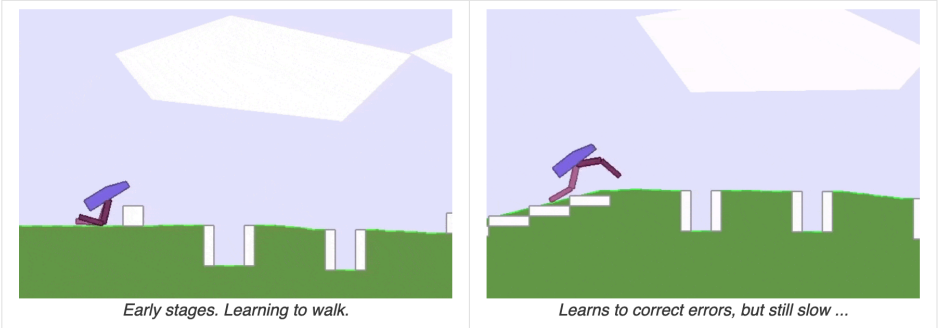


Figure 3.1: Various stages of progress in `BipedalWalkerHardcore`. For animations of these early learning behaviors and later successful ones, see https://neuroevolutionbook.com/neuroevolution-demos.

In this environment our agent has to learn a policy to walk across randomly generated terrain within the time limit without falling over. There are 24 inputs, consisting of 10 lidar sensors, angles and contacts. The agent is not given the absolute coordinates of where it is on the map. The action space is 4 continuous values controlling the torques of its 4 motors. The total reward calculation is based on the total distance achieved by the agent. Generally, if the agent completes a map, it will get score of 300+ points, although a small amount of points will be subtracted based on how much motor torque was applied, so energy usage is also a constraint.

`BipedalWalkerHardcore` defines solving the task as getting an average score of over 300 over 100 consecutive random trials. While it is relatively easy to train an agent to successfully walk across the map using an RL algorithm, it is difficult to get the agent to do so consistently and efficiently, making this task an interesting challenge. EA algorithms are able to produce policies that solve this task, while standard RL algorithms typically lead to capable policies that fall short of an average score of 300. For instance, PPO only achieves an average scores of around 240 to 250 over 100 random trials.

Because the terrain map is randomly generated for each trial, sometimes we may end up with an easy terrain, or sometimes a very difficult terrain. We don't want our natural selection process to allow agents with weak policies who had gotten lucky with an easy map to advance to the next generation. We also want to give agents with good policies a chance to redeem themselves. To achieve this, we can define an agent's episode as the average of 16 random rollouts, and use the average of the cumulative rewards over 16 rollouts as its fitness score.

Another way to look at this is to see that even though we are testing the agent over 100 trials, we usually train it on single trials, so the test-task is not the same as the training-task we are optimizing for. By averaging each agent in the population multiple times in a stochastic environment, we narrow the gap between our training set and the test set. If we can overfit to the training set, we might as well overfit to the test set, since that's the problem we ultimately want our solution to solve!

Of course, the data efficiency of our algorithm is now 16x worse, but the final policy is a lot more robust. When we tested the final policy over 100 consecutive random trials, we got an average score of over 300 points required to solve this environment. Without this averaging method, the best agent can only obtain an average score of around 220 to 230 over 100 trials, which demonstrates the benefit of averaging runs in the fitness function we ultimately optimize for.

The ability to control the tradeoff between data efficiency and policy robustness is quite powerful, and useful in the real world where we need safe policies. In theory, with enough compute, we could have even averaged over of the required 100 rollouts and optimised our Bipedal Walker directly to the requirements. Professional engineers are often required to have their designs satisfy specific Quality Assurance guarantees and meet certain safety factors. We need to be able to take into account such safety factors when we train agents to learn policies that may affect the real world.

While we demonstrate that EA can solve an existing task such as Bipedal Walker by training its neural network, the power of evolution doesn't stop there. In the natural world, evolution of our bodies happen at the same time as the evolution of our brains. We will discuss this further in Chapter 9.2>

3.3 Evolving Convolutional Neural Networks

Here we go over an example where we various EA algorithms are used to find weights for a small, simple 2-layer convolutional network (convnet) used to classify MNIST digits. In contrast to RL tasks, in supervised learning tasks gradient descent-based methods often outperform NE method. However, they can still provide good test beds for different NE algorithms.

Below are the results for various EA methods, using a population size of 101, over 300 epochs. We keep track of the model parameters that performed best on the entire training set at the end of each epoch, and evaluate this model once on the test set after 300 epochs. It is interesting how sometimes the test set's accuracy is higher than the training set for the models that have lower scores.

The results based on a single-run seem to indicate that CMA-ES is the best at the MNIST task, but the PGPE algorithm is not that far off. Both of these algorithms achieved 98% test accuracy, 1% lower than the SGD/ADAM baseline. Perhaps the ability to dynamically alter its covariance matrix, and standard deviation parameters over each generation allowed it to fine-tune its weights better than OpenAI's simpler variation.

Method	Train Set	Test Set
Adam (BackProp) Baseline	99.8	98.9
Simple GA	82.1	82.4
CMA-ES	98.4	98.1
OpenAI-ES	96.0	96.2
PEPG	98.5	98.0

Table 3.1
MNIST accuracy using several evolutionary methods (in %).

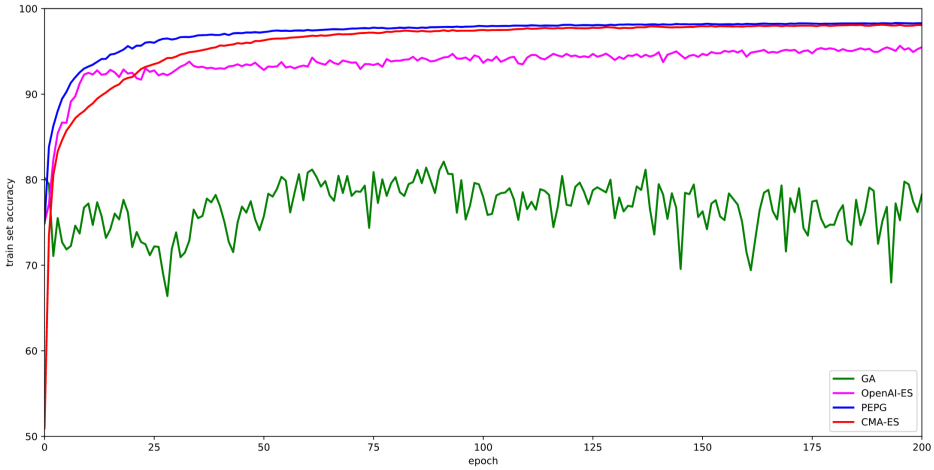


Figure 3.2: MNIST progress for several evolutionary methods.

3.4 Topology and Weight Evolving Networks: The NEAT Method

So far, we have concerned ourselves with applying EAs to searching for a suitable set of weight parameters for a given neural network. But what if we want to *it* evolve the neural network morphology (architecture) too?

Historically, many powerful neural network building blocks have been discovered through a process of trial-end-error to address certain existing neural network limitations. For example, Long Short Term Memory (LSTM) gates were hand-crafted to avoid the vanishing gradient problem for Recurrent Neural Networks (RNNs). Convolutional Neural Networks (CNNs) were created to minimize the number of connections required for computer vision problems. Residual Networks (ResNets) were designed to be able to stack up many layers of neural nets effectively. Recently, the Transformer architecture uses an attention mechanism to increase the expressivity and trainability of large neural networks. Usually, after such novel architectures are discovered and adapted in mainstream academic research, other researchers would look back and realize how simple the discovery actually was and regret that they were not the first to come up with the idea.

One goal of NE is to automate this process. While we have seen that EAs can find parameters for fixed-designs, NE does not need to stop at merely being able to discover neural network connection weights. It can also be extended to discover entire neural networks.

The ability automatically discover novel neural network architectures is a fundamentally important concept in Neuroevolution, and will play a central theme in this book.

A popular neural network morphology method is called Neuroevolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen 2002). NEAT is a method that can evolve new types of neural networks based on genetic algorithms. It was published more than two decades ago by Ken Stanley and Risto Miikkulainen and it has served as a foundation for over two hundred further algorithms and methods in the field (Papavasileiou, Cornelis, and Jansen 2021).

The way NEAT works is to represent a neural network as a list of connections. At the beginning, the initial population of networks will have a very simple architecture, such as having each input signal and bias simply connect directly to the outputs with no hidden layers. In the mutation operation, there is some probability that a new neuron will get created. A new neuron will be placed in between an existing connection, hence a new connection will be created after the introduction of a new neuron, like the below:

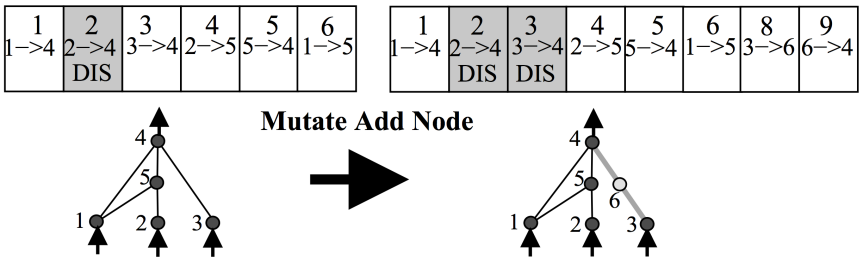


Figure 3.3: Adding a new neuron.

There is also some probability that a new connection will get created in the same mutation operation:

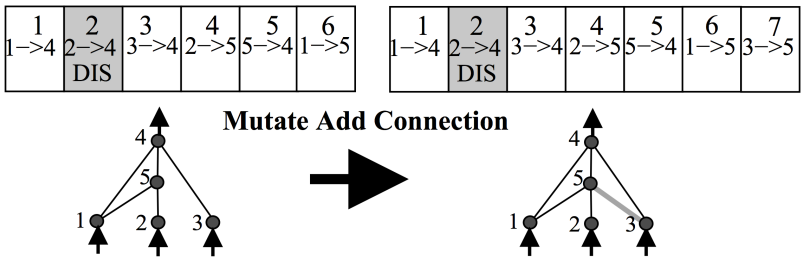


Figure 3.4: Adding a new connection.

Each neuron and connection in the entire population is unique, and assigned a unique integer label, which is called the historical marking. Thus, each network is simply a list of connections, along with the weight for those connections. Note that two different networks can have a similar connection, but the weights for each network will generally be different.

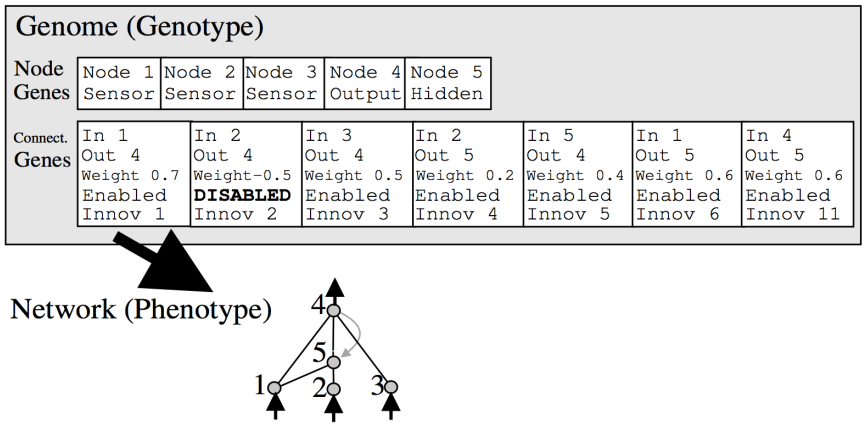


Figure 3.5: NEAT Genotype. Node genes define the types of nodes in the network: sensors (input nodes), outputs, and hidden nodes. Connection genes represent the connections between nodes, with each gene specifying the input and output nodes, connection weight, whether the connection is enabled or disabled, and an innovation number indicating the historical origin of the gene. The bottom section illustrates the neural network (phenotype) constructed based on the genome.

Because each neuron and each connection is globally unique, it becomes possible to merge two different networks, to produce a new network. This is how the crossover operation is done in NEAT. The crossover operation is quite powerful. If we have a network that is good at some sub task, and another network that is good at some other sub task, we may be able to breed an offspring network that can potentially be good at combining both skills and becoming much better than both parent networks at performing a bigger task. Another important componentn of NEAT is speciation, which we will look at next.

Another important ingredient of NEAT and many other algorithms in evolutionary computing is speciation. Speciation is the idea to group the population of genes into different species consisting of similar members of the population. The idea is to give certain members of the population, who may not be the best at performing the task, but look and behave very different than those who are currently the best, more time to evolve to their full potential, rather than to kill them off at each generation. Imagine an isolated island populated by wolves and penguins only. If we let things be, the penguins will be dead meat after the first generation and all we would be left with are wolves. But if we create a special no kill zone on the island where wolves are not allowed to kill penguins once they step inside that area, a certain number of penguins will always exist, and have time to evolve into flying penguins that will make their way back to the mainland where there's plenty of vegetation to live on, while the wolves would be stuck forever on the island.

For a more concrete example, consider the earlier example about the 100 set of weights, and imagine if we modify the algorithm from only keeping the best 20 and getting rid of the rest, to first grouping the 100 weights into 5 groups according to the similarity by, say, using euclidean distance between their weights. Now that we have 5 groups, or species, of 20 networks, for each group we would keep only the top 20% again (so we keep 4 set of weights for each species), and get rid of the remaining 16. From there, we can either

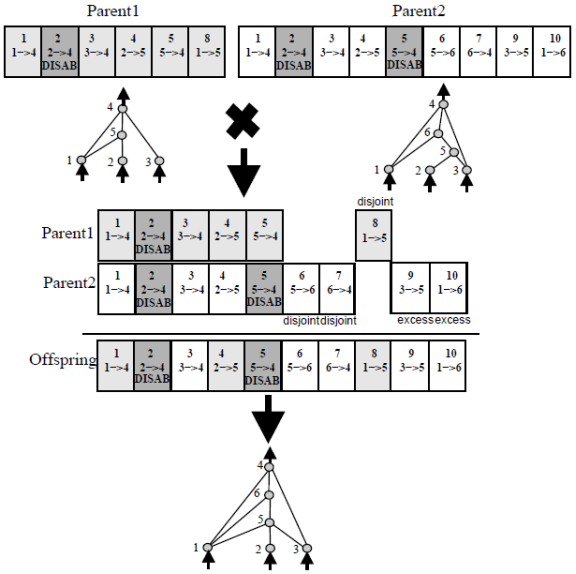


Figure 3.6: NEAT Crossover. The example shows the merging of two different networks, to produce a new network. The top row shows two parent genomes, Parent1 and Parent2, each represented by a series of genes (connections between nodes) and their corresponding neural network structures. The crossover begins by aligning the genes of the two parents. Matching genes (those present in both parents) are inherited randomly from either parent, while disjoint genes (genes that are present in one parent but not the other) and excess genes (genes that appear after the last gene of the other parent) are also considered. The resulting offspring genome combines these inherited genes, reflecting both the inherited traits from the parents and potentially new neural connections. The final offspring neural network structure, shown at the bottom, integrates the selected connections and nodes from both parents.

decide to populate the remaining 16 by crossover-mutating the 4 remaining members of each species, or from the entire set of surviving members in the larger population. By modifying the genetic algorithm this way to allow speciation, certain types of genes have the time to develop to their full potential, and also the diversity will lead to better genes that incorporate the best of different distinct types of species. Without speciation, it is easy for the population to get stuck at a local optimum.

In NEAT, speciation works as follows. It defines a distance operator to measure how different one network is to another network. Essentially, it counts the number of connections that are not shared by the two networks, and of the connections that is shares, it measures the difference in the weights inside the common connections, and the distance between two networks is a linear combination of these factors. Once we can calculate the distance between each network in our population, NEAT prescribes a formula to group networks that are within a certain distance together to form a species, or a sub population.

But as we have discussed earlier in the MNIST example, the backpropagation algorithm is quite efficient compared to EAs at solving for the weight parameters of a neural network if the objective is clearly defined and differentiable. Why not combine NEAT for finding the

architecture, and backpropagation, for solving the weights, using the best tool for the right job? Later on, in Chapter 10.1, we will discuss this when we explore Neural Architecture Search!

3.5 Neuroevolution Vs. deep Learning

Note that the networks that result from NEAT, and neuroevolution in general, are very different from those commonly used in deep learning. They are aimed at AI-based decision making, rather than prediction based on big data. The computational requirements are different, and therefore also the networks are different.

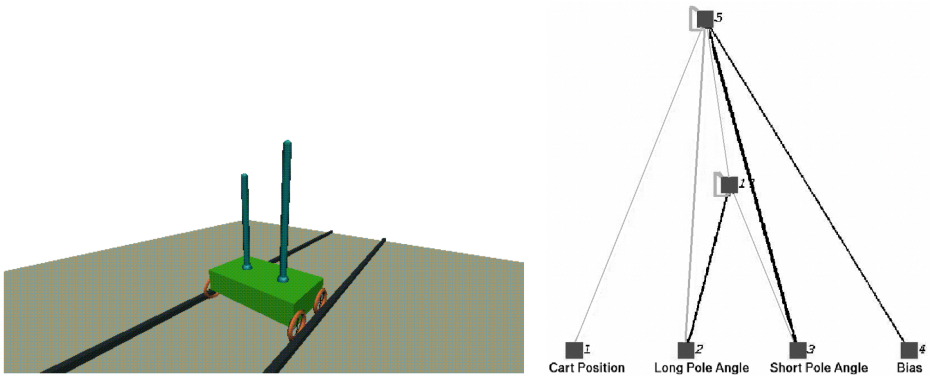
However, even in domains where deep learning can be applied, neuroevolution provides a potentially useful alternative. Performance with deep learning networks is based on overparameterization where individual components perform only minimal operations: for instance, the residual module in ResNet architectures combines bypassing the module with the transformation that the module itself computes (He et al. 2016b). In contrast, in NEAT every complexification is there for a purpose that can in principle be identified in the evolutionary history. It thus offers an alternative solution, one that is based on principled neural network design.

This kind of compact evolved neural networks can be useful in four ways: First, they can provide an explainable neural network solution. That is, the neural network still performs based on recurrency and embeddings, but its elements are constructed to provide a particular functionality, and therefore its behavior is transparent. One example is a solution NEAT discovered for the pole-balancing problem (with two poles, no velocity input): The network computes the derivative of the difference of the pole angles, which makes it possible to control the system with a very small network (Figure 3.7. Several other examples of such insights are reviewed in Sections 7.2.1 and 14.1.

Second, they can provide regularized neural network solutions, instead of overfitting to the dataset. The networks are compact, which generally leads to better regularization (Reed 1993; Oymak 2018; Ganon, Keinan, and Ruppin 2003), and they are chosen based on their overall performance instead of fine-tuned to fit individual examples. This property should be particularly useful when the datasets are relatively small, which is the case in many practical applications. Thus they can extend the scope of machine learning.

Third, they can utilize minimal hardware resources well. The advantages of deep-learning networks do not emerge until a very large number of parameters. If the hardware does not allow that scale (e.g. in edge devices), evolved networks provide an alternative principle that can be optimized to the given resources.

Fourth, they can be constructed to fit hardware constraints. Gradient descent in principle requires high precision weights and differentiable activation functions that are expensive to implement in hardware. In contrast, evolution can be used to optimize the performance of networks with e.g. quantized weights, linear threshold units, or FPGA-compatible components that are easier to implement (Gaier and Ha 2019; Z. Liu et al. 2021; Shayani, Bentley, and Tyrrell 2008). Optimization of neural networks for neuromorphic hardware is a powerful emerging direction discussed in more detail in Section 11.4.



(*a*) A challenging pole-balancing task (*b*) A compact solution by NEAT

Figure 3.7: **A compact, explainable solution NEAT discovered for the pole-balancing problem.** (*a*) In this version, there are two poles on a moving cart that needs to be pushed left or right with a constant force at regular intervals to keep the poles from falling and the cart within the left and right boundaries of the 1-D track. The poles have different lengths and thus respond differently to the push, which makes the task possible, even though it is very difficult because the of nonlinear pole interactions. Moreover, the pole and cart velocities are not given as input to the controller, that is, the task is a partial observable Markov decision process (POMDP), which makes it even more challenging. Generally, reinforment learning methods cannot solve it and only the most powerful neuroevolution methods can Gomez, Schmidhuber, and Miikkulainen 2008. (*b*) This clever solution found by NEAT works by taking the derivative of the difference in pole angles. Using the recurrent connection to itself, the single hidden node determines whether the poles are falling away or towards each other. This solution allows controlling the system without computing the velocities of each pole separately. It would be difficult to design such a subtle and compact solution by hand, but the neuroevolution that complexifies makes its discovery more likely. (Figure *a* from Gomez, Schmidhuber, and Miikkulainen 2008) (Figure *b* from Stanley and Miikkulainen 2002).

Even though neuroevolved architectures are naturally compact, it is also possible to use neuroevolution to optimize large networks. One such approach is to take advantage of indirect encoding, as will be discussed in the next chapter. It is also possible to combine deep learning synergetically with evolution, which is a topic of Chapters 10 and 11. An interesting synergy is also emerging with generative AI models such as large language models, as we will see in Section 13. These are all recent and emerging extensions of neuroevolution. The unique core of it, however, is still on evolving intelligent behavior and decision making, as discussed in Chapters 6 through 9.

3.6 Chapter Review Questions

1. **Evolutionary Algorithms:** What advantages do evolutionary algorithms (EAs) offer over traditional reinforcement learning (RL) when solving tasks where only the final outcome is known, rather than intermediate rewards?

2. **Key Mechanism:** Describe how an EA can be applied to train a neural network to solve a reinforcement learning task. Include the role of the fitness function and population-based search.
3. **Deterministic vs. Stochastic Policies:** What is the difference between deterministic and stochastic policies in neuroevolution? Why might a stochastic policy be beneficial for certain tasks?
4. **Robust Policies:** In the context of the `BipedalWalkerHardcore` example, how does evaluating an agent over multiple trials improve the robustness of the policy? What tradeoffs does this introduce?
5. **Evolutionary Optimization:** Explain how neuroevolution can evolve both the weights and the architecture of a neural network. Why is evolving the architecture a significant step beyond evolving weights alone?
6. **NEAT:** What are the main components of the NEAT (Neuroevolution of Augmenting Topologies) algorithm? Describe how mutation, crossover, and speciation contribute to its effectiveness.
7. **Speciation:** How does speciation help maintain diversity in a population during neuroevolution? Provide an example illustrating its importance.
8. **Neuroevolution vs. Deep Learning:** In what scenarios might neuroevolution outperform deep learning? Highlight at least two scenarios where neuroevolution offers unique benefits.
9. **Explainability and Compactness:** Why might solutions discovered through neuroevolution, such as NEAT's compact pole-balancing solution, be more explainable than those generated by deep learning?
10. **Emerging Synergies:** How can neuroevolution complement other AI approaches, such as large neural networks, neuromorphic hardware, or generative AI models? Provide an example of one such synergy.