2 The Basics

This chapter will first cover the basics of evolutionary algorithms, including evolution strategy and genetic algorithms. Following, we will cover the basics of how neural networks work, including the most prominent architectures appearing in this book such as feedforward, convolutional, recurrent neural networks, Long Short-Term Memory Networks, and Transformers. Readers familiar with these techniques should feel free to skip this chapter.

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of powerful optimization techniques inspired by the principles of Darwinian evolution. These algorithms operate on a population of potential solutions, evolving them over successive generations to solve complex problems. EAs are widely used in various fields, including artificial intelligence, engineering, economics, and biology, due to their ability to find optimal or near-optimal solutions in large and complex search spaces.

As an illustrative example, imagine you're trying to train a robot to fetch a ball. In an evolutionary algorithm, you would begin with a population of random candidate solutions, like different fetching attempts by the robot. Each solution is evaluated based on a fitness

Figure 2.1: *Survival of the fittest. reddit 2017*

function, which acts like the reward system (how close did the robot get?). The most successful solutions are then used as a foundation for the next generation. These ”parents” undergo variations (mutations) and combine successful traits (recombination), just like how offspring inherit and adapt their parents' traits. Over time, the population evolves, with solutions becoming increasingly fit for the task. This approach makes EAs well-suited for problems where there's no single perfect solution, or where the solution itself is complex and defies easy definition with formulas. Unlike backpropagation, which requires a clearly defined error function, EAs only need a way to evaluate ”goodness,” not a step-by-step guide. This opens doors for applications in a vast number areas where traditional gradient-based optimization techniques cannot easily be applied.

There are many problems where the backpropagation algorithm cannot be used. For example, in reinforcement learning (RL) problems, we can also train a neural network to make decisions to perform a sequence of actions to accomplish some task in an environment. However, it is not trivial to estimate the gradient of reward signals given to the agent in the future to an action performed by the agent right now, especially if the reward is realized many timesteps in the future. Even if we are able to calculate accurate gradients, there is also the issue of being stuck in a local optimum (Figure 2.2), which exists many for RL tasks.

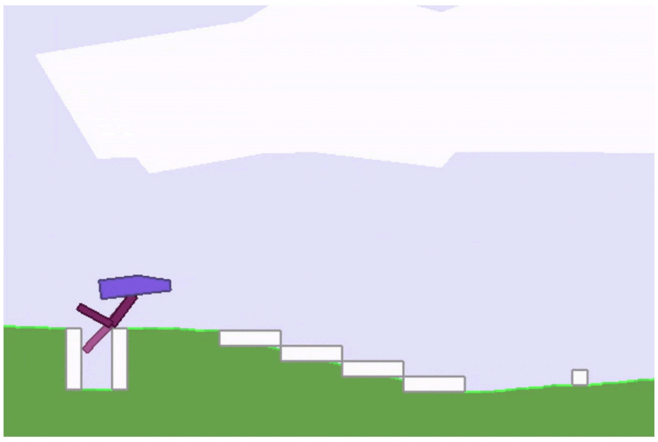

Figure 2.2: *Bipedal walker agent stuck in a local optimum.* For animations of both stuck and successful behaviors, see https://neuroevolutionbook.com/neuroevolution-demos.

A whole area within RL is devoted to studying this credit-assignment problem, and great progress has been made. However, credit assignment is still difficult when the reward signals are sparse. In the real world, rewards can be sparse and noisy. Sometimes we are given just a single reward, like a bonus check at the end of the year, and depending on our employer, it may be difficult to figure out exactly why it is so low. For these problems, rather than rely on a very noisy and possibly meaningless gradient estimate of the future to our policy, we might as well just ignore any gradient information, and attempt to use black-box optimization techniques like genetic algorithms (GA) or evolution strategy (ES).

In 2017, a group of RL researchers published a paper, *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*, which showed that evolution strategies can offer many benefits compared to RL. The ability to abandon gradient calculation allows such algorithms to be evaluated more efficiently. It is also easy to distribute the computation for an ES algorithm to thousands of machines for parallel computation. By running the algorithm from scratch many times, they also showed that policies discovered using ES tend to be more diverse compared to policies discovered by RL algorithms. Additionally, even for the problem of identifying a machine learning model, such as designing a neural net's architecture, we cannot directly compute gradients and EAs have been proven very useful. We'll explore the difference between RL and NE in more detail in Chapter 12 and dig deeper into neural architecture search in Chapter 10.

This chapter provides an overview of Evolutionary Algorithms, focusing on two of the most prominent types: Genetic Algorithms and Evolution Strategy. We will explore the underlying principles, key components, and applications of these algorithms, which share the following principles:

1. **Population-Based Search**: EAs maintain a population of candidate solutions. Each individual in the population represents a potential solution to the problem.
2. **Selection**: The selection process is inspired by the concept of "survival of the fittest." Individuals with better fitness have a higher probability of being selected for reproduction.
3. **Variation Operators**: EAs use variation operators, such as crossover (recombination) and mutation, to introduce diversity into the population by creating new individuals.
4. **Fitness Evaluation**: Each individual is evaluated using a fitness function that measures how well it solves the problem. The fitness function guides the selection process.
5. **Reproduction and Replacement**: Selected individuals reproduce to form a new generation, replacing some or all of the old population.
6. **Termination**: The algorithm iterates through multiple generations until a termination condition is met, such as reaching a maximum number of generations or achieving a satisfactory fitness level.

Given an evolution algorithm `EvolutionAlgorithm`, we can use it in the following way:

```python
solver = EvolutionAlgorithm()
while True:
  # Ask the EA to give us a set of candidate solutions.
  solutions = solver.ask()
  # Create an array to hold the fitness results.
  fitness_list = np.zeros(solver.popsize)
  # Evaluate the fitness for each given solution.
  for i in range(solver.popsize):
    fitness_list[i] = evaluate(solutions[i])
  # Give list of fitness results back to EA.
  solver.tell(fitness_list)
  # Get best parameter, fitness from EA.
  best_solution, best_fitness = solver.result()
```

```
if best_fitness > MY_REQUIRED_FITNESS:
  break
```

Although the size of the population is usually held constant for each generation, they don't need to be. The EA can generate as many candidate solutions as we want, because the solutions produced by an EA are *sampled* from a distribution whose parameters are being updated by the EA at each generation. We will further explain this sampling process with an example of a simple evolution strategy in Section 2.1.2.

2.1.1 Simple Genetic Algorithm

Genetic Algorithms (GAs) are a popular type of Evolutionary Algorithm that mimics the process of natural selection. GAs were first introduced by John Holland in the 1970s and have since become one of the most widely used EAs.

In GAs, each individual in the population is typically represented as a chromosome, which is a string of genes. The genes can be binary (0s and 1s), real numbers, or any other representation suitable for the problem at hand. The initial population is generated randomly or using a heuristic to provide a diverse set of starting solutions.

The selection process determines which individuals will contribute their genetic material to the next generation. Common selection methods include:

- **Roulette Wheel Selection**: Individuals are selected probabilistically based on their fitness, with better individuals having a higher chance of being chosen.
- **Tournament Selection**: A small group of individuals is selected randomly, and the fittest individual in the group is chosen.
- **Rank-Based Selection**: Individuals are ranked based on their fitness, and selection probabilities are assigned according to their rank.
- **Truncation Selection**: This method involves selecting the top fraction of individuals based solely on their fitness. Only the highest-performing individuals above a certain fitness threshold contribute to the next generation, while the rest are excluded. Truncation selection often leads to rapid convergence but can reduce genetic diversity.

Crossover, or recombination, is a key operator in GAs that combines the genetic material of two parent individuals to create offspring. Common crossover techniques include:

- **Single-Point Crossover**: A random crossover point is chosen, and the genes from the two parents are exchanged at this point.
- **Two-Point Crossover**: Two crossover points are selected, and the segment between them is swapped between the parents.
- **Uniform Crossover**: Each gene is independently chosen from one of the two parents with equal probability.

Mutation introduces small random changes to an individual's genes to maintain diversity in the population. This helps prevent premature convergence to local optima. The mutation rate, which determines how often mutations occur, is typically kept low.

The fitness of each individual is evaluated using a fitness function specific to the problem being solved. The fitness function measures the quality of the solution represented by the individual.

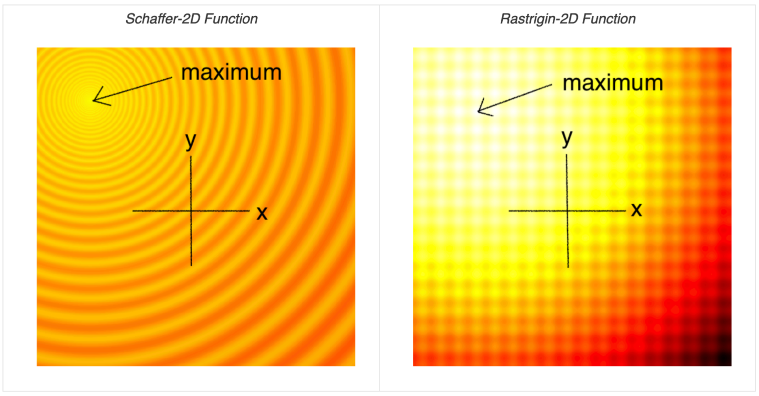

Figure 2.3: 2D Rastrigin and Schaffer functions.

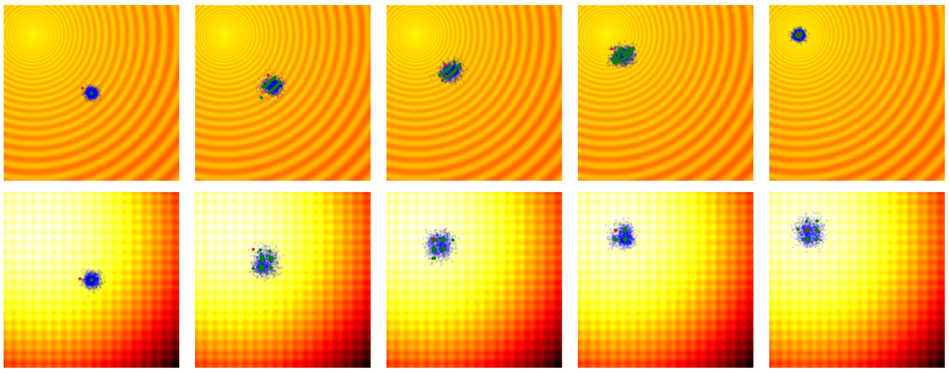

Figure 2.4: Simple GA progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos.

After selection, crossover, and mutation, a new generation of individuals is produced. This new population may replace the entire old population (generational replacement) or only a portion of it (steady-state replacement).

To get a better idea of how the GA operates, we can visualize it solving a simple toy problems. For example, Figure 2.3 shows top-down plots of shifted 2D Schaffer and Rastrigin functions , two of several simple problems used for testing continuous black-box optimisation algorithms. Lighter regions of the plots represent higher values of $F(x, y)$. As you can see, there are many local optimums in this function. Our job is to find a set of *model parameters* (x, y), such that F(x, y) is as close as possible to the global maximum. Figure 2.4 illustrates how the simple genetic algorithm proceeds over succeeding generations. The green dots represent members of the elite population from the previous generation, the blue dots are the offsprings to form the set of candidate solutions, and the red dot is the best solution.

Genetic algorithms help diversity by keeping track of a diverse set of candidate solutions to reproduce the next generation. However, in practice, most of the solutions in the elite surviving population tend to converge to a local optimum over time. There are more sophisticated variations of GA out there, such as CoSyNe, ESP, and NEAT (which we will discuss later in this book), where the idea is to cluster similar solutions in the population together into different species, to maintain better diversity over time.

2.1.2 Simple Evolution Strategy

One of the simplest evolution algorithm, which we will call *simple evolution strategy* (ES). The term *evolution strategy* is attributed to Ingo Rechenberg, from his PhD thesis where he coined the term (Rechenberg 1989). Unlike GAs, which use a binary or symbolic representation, ES typically operate on real-valued vectors and are more focused on optimizing continuous functions.

In ES, each individual is represented by a vector of real numbers, which correspond to the parameters of the solution. The initial population is usually generated randomly or based on some prior knowledge.

Selection in ES is deterministic, meaning that a fixed number of the best individuals (based on fitness) are selected to produce offspring for the next generation. Common selection strategies include:

- (μ, λ) **Selection**: A population of μ parents produces λ offspring. Only the best λ offspring are selected to form the new population.
- $(\mu + \lambda)$ **Selection**: Both the μ parents and λ offspring are considered for selection, with the best μ individuals forming the new population.

In ES, variation is introduced primarily through mutation, which perturbs the real-valued parameters. Mutation is usually applied by adding a normally distributed random vector to each individual. The mutation strength, often denoted by σ, controls the magnitude of these perturbations. Crossover is less commonly used in ES compared to GAs but can be applied by combining the parameter vectors of two or more parents.

We can imagine just sampling a set of solutions from a Normal distribution, with a mean μ and a fixed standard deviation σ. In our 2D problem, $\mu = (\mu_x, \mu_y)$ and $\sigma = (\sigma_x, \sigma_y)$. Initially, μ is set at the origin. After the fitness results are evaluated, we set μ to the best solution in the population, and sample the next generation of solutions around this new mean. Figure 2.5 shows how the algorithm behaves over 20 generations on the two problems mentioned earlier. The green dot indicates the mean of the distribution at each generation, the blue dots are the sampled solutions, and the red dot is the best solution found so far by our algorithm.

This simple algorithm will generally only work for simple problems. Given its greedy nature, it throws away all but the best solution, and can be prone to be stuck at a local optimum for more complicated problems. It would be beneficial to sample the next generation from a probability distribution that represents a more diverse set of ideas, rather than just from the best solution from the current generation.

2.1.3 Covariance-Matrix Adaptation Evolution Strategy (CMA-ES)

A shortcoming of both the Simple ES and Simple GA is that our standard deviation noise parameter is fixed. There are times when we want to explore more and increase the standard deviation of our search space, and there are times when we are confident we are close to a good optima and just want to fine-tune the solution. Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) does exactly that. CMA-ES an algorithm that can take the

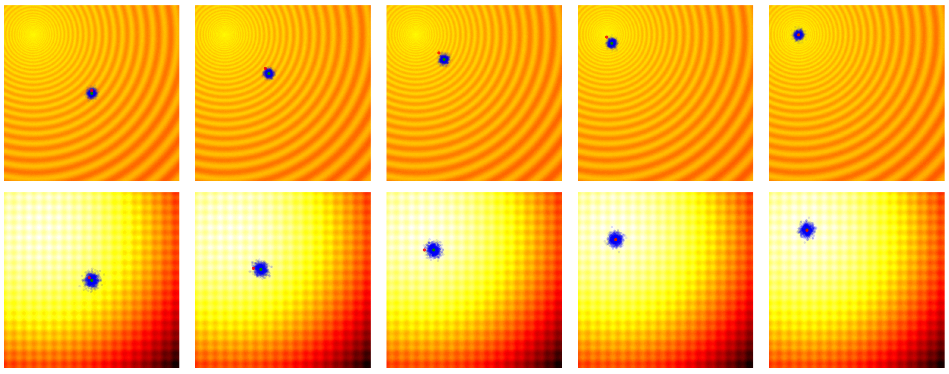

Figure 2.5: Simple ES progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos

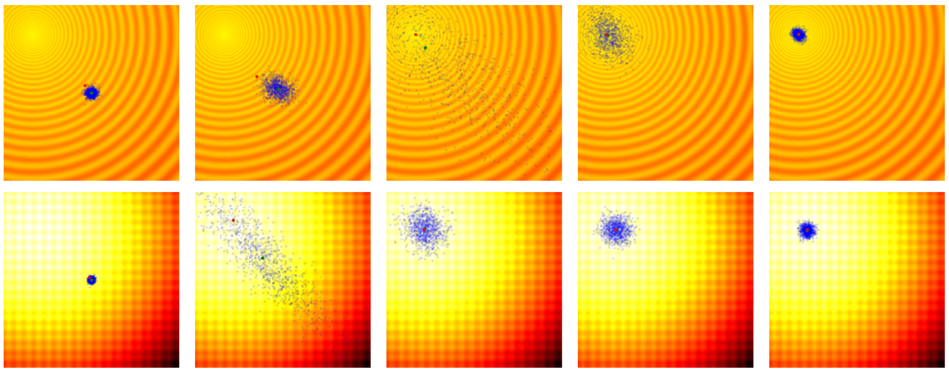

Figure 2.6: CMA-ES progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos

results of each generation, and adaptively increase or decrease the search space for the next generation. It will not only adapt for the mean μ and sigma σ parameters, but will calculate the entire covariance matrix of the parameter space. At each generation, CMA-ES provides the parameters of a multi-variate normal distribution to sample solutions from. An example of this process is shown in Figure 2.6. So how does it know how to increase or decrease the search space?

Before we discuss its methodology, let's review how to estimate a covariance matrix. This will be important to understand CMA-ES's methodology later on. If we want to estimate the covariance matrix of our entire sampled population of size of N, we can do so using the set of equations below to calculate the maximum likelihood estimate of a covariance matrix C. We first calculate the means of each of the x_i and y_i in our population:

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \tag{2.1}$$

$$\mu_y = \frac{1}{N}\sum_{i=1}^{N} y_i. \tag{2.2}$$

The terms of the 2x2 covariance matrix C will be:

$$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)^2, \tag{2.3}$$

$$\sigma_y^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu_y)^2,. \tag{2.4}$$

$$\sigma_{xy} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y). \tag{2.5}$$

Of course, these resulting mean estimates μ_x and μ_y, and covariance terms σ_x, σ_y, σ_{xy} will just be an estimate to the actual covariance matrix that we originally sampled from, and not particularly useful to us.

CMA-ES modifies the above covariance calculation formula in a clever way to make it adapt well to an optimization problem. We will go over how it does this step-by-step. Firstly, it focuses on the best N_{best} solutions in the current generation. For simplicity let's set N_{best} to be the best 25% of solutions. After sorting the solutions based on fitness, we calculate the mean $\mu^{(g+1)}$ of the next generation $(g+1)$ as the average of only the best 25% of the solutions in current population (g), i.e.:

$$\mu_x^{(g+1)} = \frac{1}{N_{best}}\sum_{i=1}^{N_{best}} x_i, \tag{2.6}$$

$$\mu_y^{(g+1)} = \frac{1}{N_{best}}\sum_{i=1}^{N_{best}} y_i. \tag{2.7}$$

Next, we use only the best 25% of the solutions to estimate the covariance matrix $C^{(g+1)}$ of the next generation, but the clever *trick* here is that it uses the *current* generation's $\mu^{(g)}$, rather than the updated $\mu^{(g+1)}$ parameters that we had just calculated, in the calculation:

$$\sigma_x^{2,(g+1)} = \frac{1}{N_{best}}\sum_{i=1}^{N_{best}}(x_i - \mu_x^{(g)})^2, \tag{2.8}$$

$$\sigma_y^{2,(g+1)} = \frac{1}{N_{best}}\sum_{i=1}^{N_{best}}(y_i - \mu_y^{(g)})^2, \tag{2.9}$$

$$\sigma_{xy}^{(g+1)} = \frac{1}{N_{best}}\sum_{i=1}^{N_{best}}(x_i - \mu_x^{(g)})(y_i - \mu_y^{(g)}). \tag{2.10}$$

Armed with a set of μ_x, μ_y, σ_x, σ_y, and σ_{xy} parameters for the next generation $(g+1)$, we can now sample the next generation of candidate solutions. Figure 2.7 visually illustrates how it uses the results from the current generation (g) to construct the solutions in the next generation $(g+1)$:

1. Calculate the fitness score of each candidate solution in generation (g).
2. Isolates the best 25% of the population in generation (g), in purple.

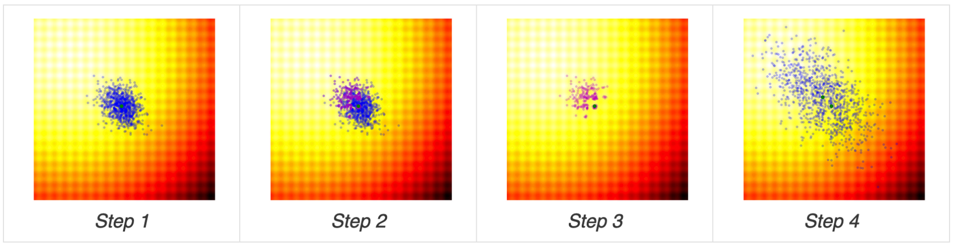

Figure 2.7: Illustration of a CMA-ES step.

3. Using only the best solutions, along with the mean $\mu^{(g)}$ of the current generation (the green dot), calculate the covariance matrix $C^{(g+1)}$ of the next generation.
4. Sample a new set of candidate solutions using the updated mean $\mu^{(g+1)}$ and covariance matrix $C^{(g+1)}$.

Let's visualize the scheme one more time, in more detail (Figure 2.8). Because CMA-ES

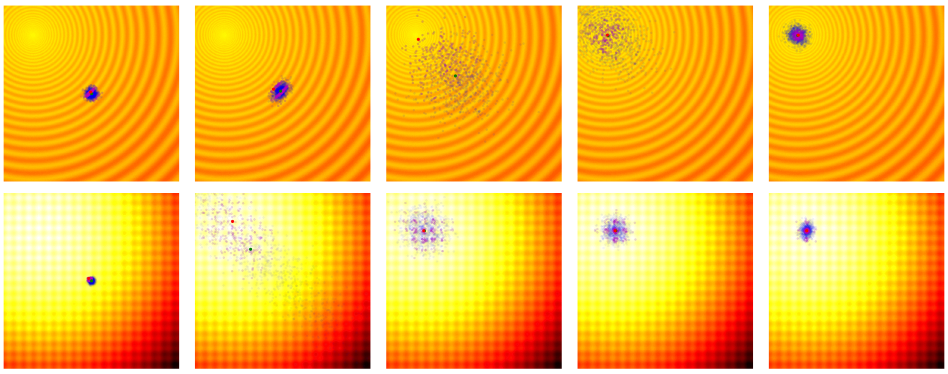

Figure 2.8: CMA-ES progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos

can adapt both its mean and covariance matrix using information from the best solutions, it can decide to cast a wider net when the best solutions are far away, or narrow the search space when the best solutions are close by. Our description of the CMA-ES algorithm for a 2D toy problem is highly simplified to get the idea across. For more details, we suggest reading the CMA-ES Tutorial (Hansen 2016) prepared by Nikolaus Hansen, the author of CMA-ES.

This algorithm is one of the most popular gradient-free optimization algorithms out there, and has been the algorithm of choice for many researchers and practitioners alike. The only real drawback is the performance if the number of model parameters we need to solve for is large, as the covariance calculation is $O(N^2)$, although recently there has been approximations to make it $O(N)$. CMA-ES is generally a good algorithm of choice when the search space is less than a thousand parameters. We find that it is still usable up to around 10K parameters if we're willing to be patient.

2.1.4 Natural Evolution Strategies

"Imagine if you had built an artificial life simulator, and you sample a different neural network to control the behavior of each ant inside an ant colony. Using the Simple Evolution Strategy for this task will optimize for traits and behaviors that benefit individual ants, and with each successive generation, our population will be full of alpha ants who only care about their own well-being.

Instead of using a rule that is based on the survival of the fittest ants, what if you take an alternative approach where you take the sum of all fitness values of the entire ant population, and optimize for this sum instead to maximize the well-being of the entire ant population over successive generations? Well, you would end up creating a Marxist utopia."

A perceived weakness of the algorithms mentioned so far is that they discard the majority of the solutions and only keep the best solutions. Weak solutions contain information about what *not* to do, and this is valuable information to calculate a better estimate for the next generation.

Many people who studied RL are familiar with the REINFORCE paper (Ronald J Williams 1992a). In this 1992 paper, Williams outlined an approach to estimate the gradient of the expected rewards with respect to the model parameters of a policy neural network. This paper also proposed using REINFORCE as an Evolution Strategy, in Section 6 of the paper. This special case of *REINFORCE-ES* was expanded later on in Parameter-Exploring Policy Gradients (PGPE) (Sehnke et al. 2010) and Natural Evolution Strategies (NES) (Wierstra et al. 2008).

In this approach, we want to use all of the information from each member of the population, good or bad, for estimating a gradient signal that can move the entire population to a better direction in the next generation. Since we are estimating a gradient, we can also use this gradient in a standard SGD update rule typically used for deep learning. We can even use this estimated gradient with more advanced SGD techniques known in the deep learning literature, such as Momentum, RMSProp, or Adam if we want to.

The idea is to maximize the *expected value* of the fitness score of a sampled solution. If the expected result is good enough, then the best performing member within a sampled population will be even better, so optimizing for the expectation might be a sensible approach. Maximizing the expected fitness score of a sampled solution is almost the same as maximizing the total fitness score of the entire population.

If z is a solution vector sampled from a probability distribution function $\pi(z, \theta)$, we can define the expected value of the objective function F as:

$$J(\theta) = E_\theta[F(z)] = \int F(z)\,\pi(z, \theta)\,dz, \tag{2.11}$$

where θ are the parameters of the probability distribution function. For example, if π is a normal distribution, then θ would be μ and σ. For our simple 2D toy problems, each ensemble z is a 2D vector (x, y).

The NES paper (Wierstra et al. 2008) contains a nice derivation of the gradient of $J(\theta)$ with respect to θ. Using the same *log-likelihood trick* as in the REINFORCE algorithm allows us to calculate the gradient of $J(\theta)$:

$$\nabla_\theta J(\theta) = E_\theta[\, F(z)\, \nabla_\theta \log \pi(z, \theta)\,]. \tag{2.12}$$

In a population size of N, where we have solutions z^1, z^2, ... z^N, we can estimate this gradient as a summation:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} F(z^i)\, \nabla_\theta \log \pi(z^i, \theta). \tag{2.13}$$

With this gradient $\nabla_\theta J(\theta)$, we can use a learning rate parameter α (such as 0.01) and start optimizing the θ parameters of probability distribution function (pdf) π so that our sampled solutions will likely get higher fitness scores on the objective function F. Using SGD (or Adam), we can update θ for the next generation:

$$\theta \to \theta + \alpha \nabla_\theta J(\theta), \tag{2.14}$$

and sample a new set of candidate solutions z from this updated pdf, and continue until we arrive at a satisfactory solution.

In Section 6 of the REINFORCE paper (Ronald J Williams 1992a), Williams derived closed-form formulas of the gradient $\nabla_\theta \log \pi(z^i, \theta)$, for the special case where $\pi(z, \theta)$ is a factored multi-variate normal distribution (i.e., the correlation parameters are zero). In this special case, θ are the μ and σ vectors. Therefore, each element of a solution can be sampled from a univariate normal distribution $z_j \sim N(\mu_j, \sigma_j)$.

The closed-form formulas for $\nabla_\theta \log N(z^i, \theta)$, for each individual element of vector θ on each solution i in the population can be derived as:

$$\nabla_{\mu_j} \log N(z^i, \mu, \sigma) = \frac{z_j^i - \mu_j}{\sigma_j^2}, \nabla_{\sigma_j} \log N(z^i, \mu, \sigma) \qquad = \frac{(z_j^i - \mu_j)^2 - \sigma_j^2}{\sigma_j^3}. \tag{2.15}$$

For clarity, we use the index of j, to count across parameter space, and this is not to be confused with superscript i, used to count across each sampled member of the population. For our 2D problems, $z_1 = x$, $z_2 = y$, $\mu_1 = \mu_x$, $\mu_2 = \mu_y$, $\sigma_1 = \sigma_x$, $\sigma_2 = \sigma_y$ in this context.

These two formulas can be plugged back into the approximate gradient formula to derive explicit update rules for μ and σ. In the papers mentioned above, they derived more explicit update rules, incorporated a *baseline*, and introduced other tricks such as antithetic sampling in PGPE, which is what my implementation is based on. NES proposed incorporating the inverse of the Fisher Information Matrix into the gradient update rule. But the concept is basically the same as other ES algorithms, where we update the mean and standard deviation of a multi-variate normal distribution at each new generation, and sample a new set of solutions from the updated distribution. Figure 2.9 shows a visualization of this algorithm in action, following the formulas described above.

This algorithm is able to dynamically change the σ's to explore or fine tune the solution space as needed. Unlike CMA-ES, there is no correlation structure in our implementation, so we don't get the diagonal ellipse samples, only the vertical or horizontal ones, although in principle we can derive update rules to incorporate the entire covariance matrix if we needed to, at the expense of computational efficiency. Similarly to CMA-ES, the σ's can adapt so our search space can be expanded or narrowed over time. Because the correlation parameter is not used in this implementation, the efficiency of the algorithm is $O(N)$ so we

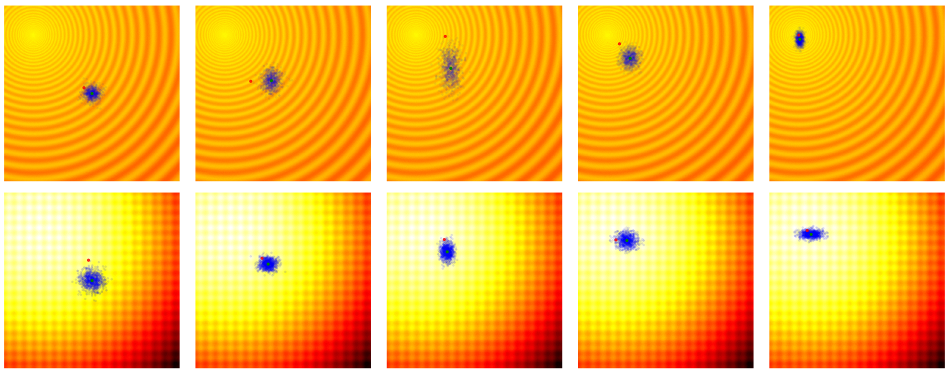
Figure 2.9: NES progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos

use PGPE if the performance of CMA-ES becomes an issue. It is preferable to use PGPE when the number of model parameters exceed several thousand.

2.1.5 OpenAI ES

In a paper published by OpenAI in 2017 (Salimans et al. 2017), they implement an evolution strategy that is a special case of the REINFORCE-ES and PGPE algorithms outlined earlier. In particular, σ is fixed to a constant number, and only the μ parameter is updated at each generation. Figure 2.10 shows how this strategy looks like, with a constant σ parameter.

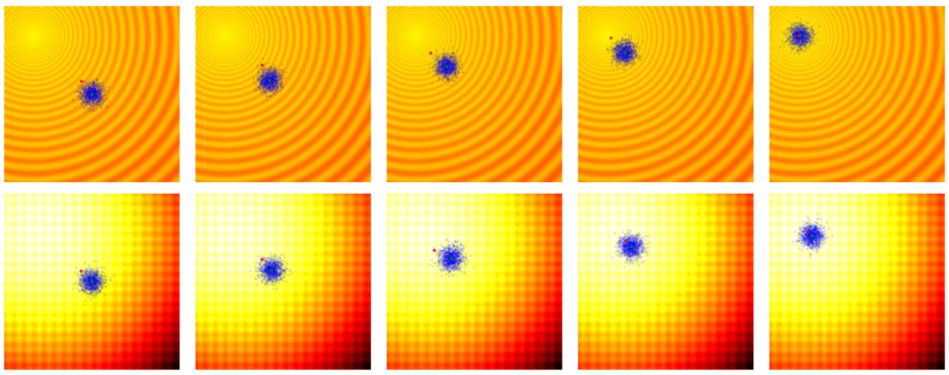
Figure 2.10: NES progress over 20 steps. For animations, see https://neuroevolutionbook.com/neuroevolution-demos

In addition to the simplification, this paper also proposed a modification of the update rule that is suitable for parallel computation across different worker machines. In their update rule, a large grid of random numbers have been pre-computed using a fixed seed. By doing this, each worker can reproduce the parameters of every other worker over time, and each worker needs only to communicate a single number, the final fitness result, to all of the other workers. This is important if we want to scale evolution strategies to thousands or even a million workers located on different machines, since while it may not be feasible to transmit an entire solution vector a million times at each generation update, it may be feasible to transmit only the final fitness results. In the paper, they showed that by using 1440 workers on Amazon's cloud computing platform, they were able to solve difficult robotic manipulation tasks in a matter of minutes.

In principle, this parallel update rule should work with the original algorithm where they can also adapt σ, but perhaps in practice, they wanted to keep the number of moving parts to a minimum for large-scale parallel computing experiments. This inspiring paper also discussed many other practical aspects of deploying ES for RL-style tasks, and we highly recommend going through it to learn more.

2.1.6 Fitness Shaping

Most of the algorithms above are usually combined with a *fitness shaping* method, such as the rank-based fitness shaping method We will discuss here. Fitness shaping allows us to avoid outliers in the population from dominating the approximate gradient calculation mentioned earlier:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} F(z^i)\,\nabla_\theta \log \pi(z^i, \theta). \tag{2.16}$$

If a particular $F(z^m)$ is much larger than other $F(z^i)$ in the population, then the gradient might become dominated by this outliers and increase the chance of the algorithm being stuck in a local optimum. To mitigate this, one can apply a rank transformation of the fitness. Rather than use the actual fitness function, we would rank the results and use an augmented fitness function which is proportional to the solution's rank in the population. Below is a comparison of what the original set of fitness may look like, and what the ranked fitness looks like:

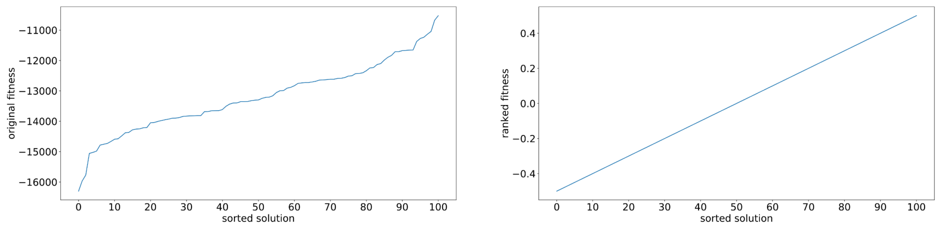

Figure 2.11: A comparison of what the original set of fitness may look like (left), and what the ranked fitness looks like (right).

What this means is supposed we have a population size of 101. We would evaluate each population to the actual fitness function, and then sort the solutions based by their fitness. We will assign an augmented fitness value of -0.50 to the worse performer, -0.49 to the second worse solution, ..., 0.49 to the second best solution, and finally a fitness value of 0.50 to the best solution. This augmented set of fitness values will be used to calculate the gradient update, instead of the actual fitness values. In a way, it is a similar to just applying

Batch Normalization to the results, but more direct. There are alternative methods for fitness shaping but they all basically give similar results in the end.

We find fitness shaping to be very useful for RL tasks if the objective function is non-deterministic for a given policy network, which is often the cases on RL environments where maps are randomly generated and various opponents have random policies. It is less useful for optimizing for well-behaved functions that are deterministic, and the use of fitness shaping can sometimes slow down the time it takes to find a good solution.

2.1.7 Try these algorithms yourself

There is no better way to learn and to gain intuition than by trying out all of these EAs algorithms out yourself. There are probably open source implementations of all of the algorithms described in this book. The author of CMA-ES, Nikolaus Hansen, has been maintaining a numpy-based implementation of CMA-ES (https://github.com/CMA-ES/pycma) with lots of bells and whistles. His python implementation introduced me to the training loop interface described earlier.

Since this interface is quite easy to use, we also implemented the other algorithms such as Simple Genetic Algorithm, PGPE, and OpenAI's ES using the same interface, and put it in a small python file called `es.py`, and also wrapped the original CMA-ES library in this small library. This way, I can quickly compare different ES algorithms by just changing one line:

```python
import es

# solver = es.SimpleGA(...)
# solver = es.PGPE(...)
# solver = es.OpenES(...)
solver = es.CMAES(...)

while True:
  solutions = solver.ask()
  fitness_list = np.zeros(solver.popsize)

  for i in range(solver.popsize):
    fitness_list[i] = evaluate(solutions[i])

  solver.tell(fitness_list)
  result = solver.result()

  if result[1] > MY_REQUIRED_FITNESS:
    break
```

You can look at `es.py` on the GitHub repo https://github.com/hardmaru/estool and the accompanying notebook in the repo using the various ES algorithms. In the accompanying notebook, we show how to use the ES solvers in `es.py` to solve a 100-Dimensional version of the Rastrigin function with even more local optimum points. The 100-D version is somewhat more challenging than the trivial 2D version used to produce the visualizations in this article. Below is a comparison of the performance for various algorithms discussed:

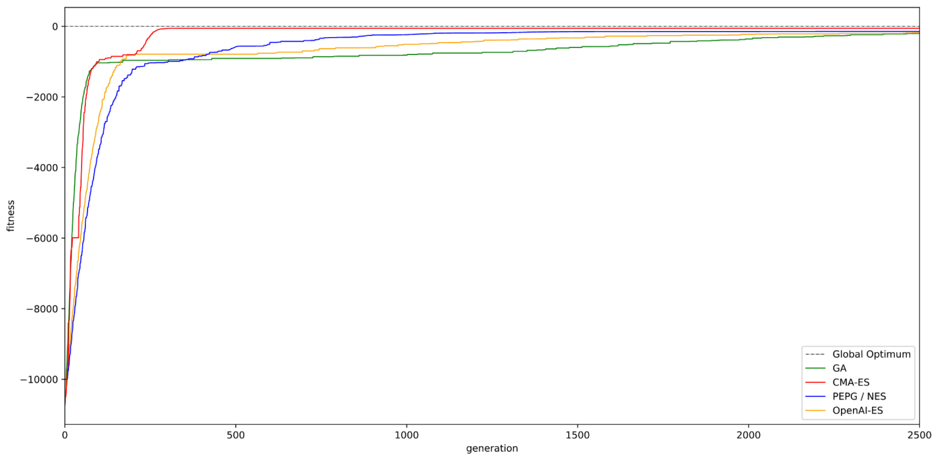

Figure 2.12: A comparison of the performance for various algorithms discussed in this Chapter for the 100-Dimensional Rastrigin function.

On this 100-D Rastrigin problem, none of the optimizers got to the global optimum solution, although CMA-ES comes close. CMA-ES is clearly the best performer. PGPE is in 2nd place, and OpenAI-ES / Genetic Algorithm falls behind. We had to use an annealing schedule to gradually lower σ for OpenAI-ES to make it perform better for this task.

```
[ 10.          10.          10.          10.          10.
  10.99495864  10.99495864   8.01008777  10.99495864   9.00504136
  10.          10.          10.           9.00504136  10.
   9.00504136  10.          10.99495864   8.01008776  10.99495864
  10.99495864  10.           9.00504137  10.99495863  10.99495864
   9.00504136  10.           9.00504136  10.99495863  10.
  10.99495863   8.01008776  10.99495863  10.          10.
  10.99495863   9.00504137  10.99495863   9.99999999  10.
  10.          10.          10.          10.          10.
  10.99495864  10.          10.           9.00504137   9.00504137
  10.           9.00504136  10.99495863  10.99495864  10.00000001
   9.00504136  10.           9.00504136  10.          10.
  10.           9.99999999  10.          10.           9.99999999
  10.           9.00504137  10.          10.           9.00504136
   9.00504137  10.           9.00504137   9.99999999  10.
   9.00504136   9.00504136  10.00000001  10.          10.
   9.00504136  10.99495864  10.          10.99495864  10.
  10.00000001  10.00000001  10.           9.00504136  10.99495863
   9.00504136  10.99495865  10.          10.          10.00000001
   9.99999999  10.          10.99495864   9.99999999  10.        ]
```

Figure 2.13: The final solution that CMA-ES discovered for 100-D Rastrigin function. The global optimal solution is a 100-dimensional vector of exactly 10.

2.2 Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning models loosely inspired by the structure and function of the human brain. They consist of layers of interconnected