FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Matyáš Lorenc

Evolution strategies for policy optimization in transformers

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Roman Neruda, CSc.
Study programme:	Computer Science - Artificial Intelligence (N0619A140003)
Study branch:	Computer Science - Artificial Intelligence

Prague 2024

To everyone who has shaped me over the years. To all who have shone like a beacon to guide me through the chaotic mess that life can sometimes be.

Title: Evolution strategies for policy optimization in transformers

Author: Matyáš Lorenc

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: We explore the capability of evolution strategies to train a transformer architecture in the reinforcement learning setting. We perform experiments using OpenAI's highly parallelizable evolution strategy and its derivatives utilizing novelty and quality-diversity searches to train Decision Transformer in Humanoid locomotion environment, testing the ability of these black-box optimization techniques to train even such relatively large (compared to the previously tested in the literature) and complicated (using a self-attention in addition to fully connected layers) models. The tested algorithms proved to be, in general, capable of achieving strong results and managed to obtain high-performing agents both from scratch (randomly initialized model) and from a pretrained model.

Keywords: Evolution strategies, Tranformers, Policy optimization, Novelty

Název práce: Evoluční strategie pro optimalizaci policy v transformerech

Autor: Matyáš Lorenc

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem práce je prozkoumat schopnost evolučních strategií trénovat architektury transformerů v prostředí zpětnovazebního učení. Provedeme experimenty s využitím vysoce paralelizovatelného algoritmu OpenAI-ES a dvou jeho variant využívajících konceptů novelty a quality-diversity prohledávání k trénování architektury Decision Transformeru v prostředí MuJoCo Humanoida a otestujeme tak schopnost těchto black-box optimalizačních technik trénovat i takto relativně velké (ve srovnání s dříve testovanými) a komplikované modely (využívajících self-attention vedle klasických plně propojených vrstev). Testované algoritmy se v našich experimentech ukázaly obecně jako schopné dosahovat silných výsledků a dokázaly vyvinout vysoce výkonné agenty — a to jak z náhodně inicializovaného modelu, tak z předtrénovaného modelu.

Klíčová slova: Evoluční strategie, Transformery, Optimalizace policy, Novelty

Contents

1. Introduction

The problem of reinforcement learning is regarded as one of the most difficult in the field of machine learning. There are many approaches to solving it. Some are based on computing a gradient to optimize the objective, some are derivative-free. One such class of derivative-free optimization algorithms are evolution algorithms. Their subclass of evolution strategies has been proved to be a viable alternative to gradient approaches for (deep) reinforcement learning. Albeit the gradient approaches generally have a better sample utilization, the evolution strategies are greatly parallelizable. Moreover, evolution strategies have better exploration of possible solutions and their trained agents are usually more diverse than those trained by gradient-based algorithms. They can even incorporate techniques that vastly improve the exploration, such as searching for novelty, instead of, or in addition to, just seeking a better performance. The pinnacle of such algorithms for reinforcement learning is nowadays OpenAI-ES as a purely objective-based algorithm, NS-ES as a purely novelty-based representative, and NSR-ES as an algorithm combining the searching for objective and for novelty, to balance quality and diversity of the trained agents.

On a different note, transformer architecture is lately the go-to solution in the field of neural networks and supervised learning for an ever-growing range of problems. And recently, there have been attempts to reformulate reinforcement learning as a sequence modeling problem and to leverage the capabilities of transformers in such tasks to obtain a new approach for solving this class of problems, yielding us models such as Decision Transformer or Trajectory Transformer. The Decision Transformer was originally introduced as a model for offline reinforcement learning using a supervised learning of sequence prediction, but the authors claim it would function well even in the classical reinforcement learning tasks.

We decided to subject the combination of the evolution strategy algorithms and the Decision Transformer to experiments, and test the ability of derivative-free algorithms to train this more complicated and bigger — compared to the models that had been experimented with in the literature so far, which were simple feed-forward models — transformer architecture. For large and complicated models it may be hard, or even almost impossible, to train them from scratch using evolutionary approach — it might be virtually impossible to randomly stumble on a model that does something interesting in the population. Hence, shifting toward better performing models of the population may not be a viable method of obtaining a well-trained agent for larger models. Therefore, we wanted to experiment with first pretraining the agent using a supervised learning of sequence prediction on data generated by a smaller model, which we can train using some arbitrary reinforcement learning method.

Due to the original implementation of the algorithms, which we found to be not so user-friendly and not really easily modifiable to incorporate custom models, environments, and behaviors, we decided to reimplement the algorithms in a much more user-friendly and versatile way and to provide even some replication experiments for the original papers. Our implementation can be found here: `https://github.com/Mafi412/es_contra_dt`

The goals of this thesis are then to test the viability of training a transformer architecture using evolution strategies; to evaluate the effectiveness of utilizing a novelty search on this model architecture; to investigate the possibility of seeding these algorithms with pretrained models; and to provide extensible, user-friendly, and versatile implementation for the examined algorithms.

Let us outline the structure of this work here. In Chapter 2, we introduce some basic topics and concepts this thesis works with, such as the problem of reinforcement learning, evolution strategies, novelty search, and transformer architecture. We follow up on this in Chapter 3, where we introduce the core algorithms — OpenAI-ES, NS-ES, and NSR-ES — and Decision Transformer architecture. In Chapter 4, we describe methods used and we go through interesting details of our implementation. We then proceed to state the setup of our experiments, after which we go through and discuss the results of the experiments, both in Chapter 5. We conclude in Chapter 6 by recapitulating and discussing the results and laying out the possibilities for future work.

2. Background

Let us begin by laying the foundations of fields used as building blocks for research in this thesis. First, we will formulate a problem of reinforcement learning; then we will take a look at evolutionary algorithms; and finally, we will conclude this chapter by introducing transformers.

2.1 Reinforcement learning

As mentioned previously, this first section will introduce the concept of *reinforcement learning* (RL). For a deeper understanding, we recommend our readers refer to the introductory book on this topic, ”Reinforcement Learning: An Introduction” [1]. But we lay out the basic concepts here as well.

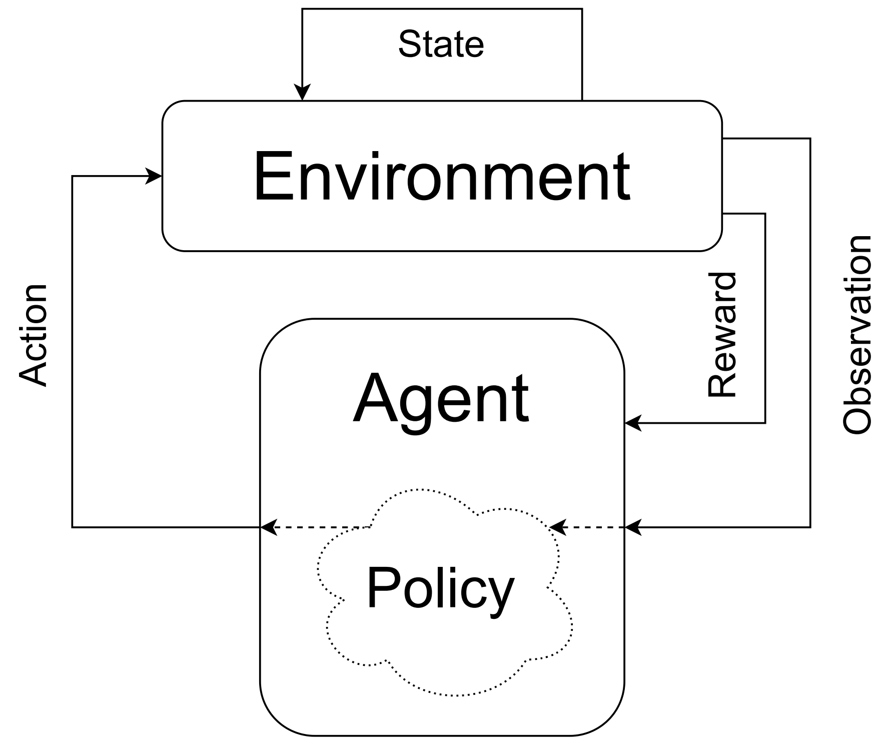


Figure 2.1: Basic reinforcement learning setting

The basic setting of RL, which can be seen in figure 2.1, consists of an *agent* perceiving *observations* and performing *actions* according to a *policy* in an *environment*, obtaining a *reward*. The agent then tries to maximize its *return*. Let us break down the concepts mentioned above: The environment is an entity without its own goal (otherwise it could be perceived as another agent) with an inner state. It changes the state based on the previous ones plus the agent’s actions and returns the observation together with the reward based on the new state. The agent is then another entity that processes the given observation to determine the action. The policy is realized by a function the agent uses to derive the action from the observation, possibly some model, recently mostly a neural network (NN). Reward is a numerical signal from the environment to the agent telling it whether it did something good or bad. One cycle of performing the agent's action in the environment and obtaining a new observation and a reward, we call a *timestep*. One *episode* is then a sequence of timesteps, beginning in an environment's state and with an initial observation given to the agent, and terminating by reaching some end conditions, like time running out, the agent doing something really bad or really good, or anything the particular use case requires. Or it might even not end at all. Lastly, the return is an aggregate of all the rewards obtained by the agent during one episode (so far). It might be a simple sum of the rewards for finite environments (where the episodes are always finite), or a discounted sum of the rewards for possibly infinite environments. (Sometimes the words "state" and "observation" are used interchangeably, especially for the deterministic environments, where what the agent gets from the environment as observation is the full information about the environment's state.)

Methods for solving this problem and training the agent to perform actions yielding high return might construct a model of the environment [2], they may just learn to predict a reward for a given action in a given state [3], or have no understanding of the environment whatsoever [4]. They can leverage the model of the environment to train without interacting with the real one [5]. They can update the agent after every timestep (temporal difference methods - TD) [6], or after each episode [4].

Recently, the model has been mostly implemented using a deep NN (DNN), as was pointed out above. Such a subclass of methods is called deep reinforcement learning, however, henceforth we will call it just RL in this work for simplicity, because it is the subclass we research here. Such DNN based agents are nowadays trained mostly using gradient approaches, such as, e.g., Trust Region Policy Optimization [7], and Proximal Policy Optimization [8].

2.2 Evolutionary algorithms

One large and quite a successful family of black-box optimization algorithms is the family of *evolutionary algorithms* (EA). They consist of various metaheuristics inspired by nature. Probably one of the most famous and the most simple representative of this algorithm class is *genetic algorithms* (GA) [9]. They take a *population* of *individuals* — those representing possible solutions to the given problem, mostly in a form of a binary, an integer or a real-valued vectors — evaluate each of them on how good they are based on the task at hand, yielding us *fitness* (*objective* value) for each individual, then somehow recombine the more successful of them via a *crossover*, and possibly *mutate* the results by slight changes to create *offsprings*, hence forming a new population, or we might call it a new *generation*. And then repeat this again, until a terminal condition is met. We can see that this process is inspired by genes and their competition and evolution in the real world.

For more information about this broad and diverse family of algorithms, which the EAs are, we recommend the readers refer to a summary book, "Evolutionary Computation" [10]. From now on, we will focus on one specific subclass of EAs, *evolution strategies* (ES).

2.2.1 Evolution strategies

Introduced as a tool to deal with high-dimensional continuous-valued domains [11], ES work again with a population of individuals — real-valued vectors. (However, in some cases, the population might consist of a single individual.) In each generation, they derive a new set of individuals by somehow mutating the original population; the new set is then evaluated, and a new generation is formed by a subset of individuals from the set based on their fitness. Ergo, we might say they are similar to GAs, except they do not use a crossover. Another difference may be that while GAs tend mostly to derive the next generation by creating the same number of new individuals as there were in the original population by crossover and mutations, ES often deploy (μ, λ)-scheme or $(\mu + \lambda)$-scheme, which means that the population size is μ, number of newly generated offsprings is λ in every iteration and we select μ best individuals from the offsprings in (μ, λ)-scheme, or from both the previous population and the offsprings in $(\mu + \lambda)$-scheme to construct a new generation.

The main difference compared to GAs, however, is the mutation. In ES, the mutation rate, or step size, is learned, or coevolved, with the population. The basic example may be enlarging the size of mutation when many successful offsprings are created (explore more) and shrinking it when few successful offsprings are being made (exploit more). Another step is to coevolve every individual with its own mutation rate. But we can take it step further and adapt mutation size for every element of the individual-vector independently. Or we can mutate every element with respect to the others, using their covariance defined by an underlying covariance matrix, just as the Covariance Matrix Adaptation ES (CMA-ES) algorithm does [12]. CMA-ES is an efficient method with strong results, yet it requires keeping the covariance matrix, which is infeasible for higher-dimensional problems because the size of the matrix grows quadratically with respect to the dimension.

Natural Evolution Strategies (NES) [13] personify another approach to the core idea of ES. The only individual in their population is a distribution whose mean value is the candidate solution to the given problem. The distribution is defined by some parameters, and those are the updatable entities the NES works on. Thus, for example, for Gaussian distribution, NES would work on its mean and covariance matrix. In each iteration, the algorithm then samples multiple instances from the distribution, evaluates them, and estimates how to change the parameters so that we obtained higher expected fitness for the samples using the *natural gradient*, which renormalizes the update with respect to uncertainty. An example of NES is, e.g., OpenAI-ES, which we will talk more about in Section 3.1.

2.2.2 Evolution strategies in reinforcement learning

As mentioned in Section 2.1, it has recently become quite common to implement RL agents (or more precisely, their policies) using DNNs. But what other are the DNNs than parameterized functions with high-dimensional and (usually) continuous-valued parameters? Therefore, we can represent the policy by its DNN's parameters and use some ES algorithm to optimize it. The fitness of an individual is then obtained as a mean return of the individual (representing the agent's policy) from several episodes. For this purpose, various ES algorithms were proposed [14].

This approach to RL has its disadvantages. For example, in order to compute individual's fitness, we have to run the whole episodes. But what if we have an infinite environment? Then we have to artificially make each episode finite by adding some timestep limit. But then we solve a problem which is close to, but not the same as the original one. Another disadvantage this approach has is that its sample utilization is low compared to the gradient-based methods. What is meant by that is, in essence, we are capable of extracting more information from a timestep or an episode (a sample) using gradients.

However, ES bring along a lot of advantages as well. Many of the ES algorithms are easily parallelizable, and much of the research in the area has been focused on this aspect, yielding us algorithms with almost linear improvement of performance when more computing power is used [15]. And because ES are derivative-free algorithms, we can optimize not just classical smooth NNs, but our models can even contain some discrete subfunctions or be otherwise non-differentiable.

2.2.3 Novelty search

EAs in general, for us here specifically ES, bring with them another advantage. They allow easy access to and incorporation of strong exploration tools, such as searching not for the highest fitness, but for *novelty*.

The concept of searching not for objective, but just for novelty — the so-called *novelty search* (NS) — was introduced [16, 17] as an instrument to deal with deceptive objective functions and their treacherous local optima, and was based on a reasoning that nature, more specifically evolution in the nature, does not follow any specific objective, but is more open-ended, exploring all the possibilities. Therefore, the authors reasoned, if we are willing to forgo the search for objective and instead search for some behavioral novelty, we can solve some of the problems that proved too difficult for objective-based algorithms, hence enlarging our toolbox for solving diverse problems.

For us to be capable of defining novelty, we first need to be able to define something which we will define the novelty on. Such a concept is a *behavior*, or a *behavior characteristic* in other words. Behavior describes what an agent does in an environment. It might be the last state when the episode ends. It might be a sequence of actions the agent performed during the whole episode. Or anything else that comes in mind and proves useful for the search, as long as it provides some metric that can be used to compute distance, or similarity, of two behaviors. Even though we do not really care for the objective in the NS, the behavior should still somehow align with it. So if we wanted to train some robot to walk, we should probably not choose ”how many times did the robot turn around” as a behavior but rather, for example, its final position in space.

Now that we have a behavior, we can run almost any EA just with the following alterations: The behaviors of agents from past populations are stored in an *archive*. The novelty of an individual can then be defined as the mean distance of its behavior characteristic from its k-nearest neighbors from the archive. And wherever we used a fitness in the algorithm, we now use the novelty instead.

Hence, we get simple, yet possibly effective algorithms, as was shown through experiments by the original authors [17]. However, as anyone could probably guess, although there are many environments solvable by search for novelty alone, there are many more environments where it is handy to search for both the fitness (to obtain high-performing agents) and the novelty (to escape local optima and diversify the agents). For such occasions, there exist *quality-diversity* algorithms (QD) [18] which do exactly that by computing both the novelty and the fitness of an individual and combining the two information together to decide how good the given individual is.

2.3 Transformers

The field of NNs has been around for quite some time. NNs are (mostly layered) oriented graphs with *artificial neurons* for nodes. The artificial neuron is then basically a function taking values on the incoming edges as input, aggregating them using a linear combination with a bias constant and passing the result through some non-linearity-introducing function (called activation), then sending it along the outgoing edges, as can be seen in figure 2.2.

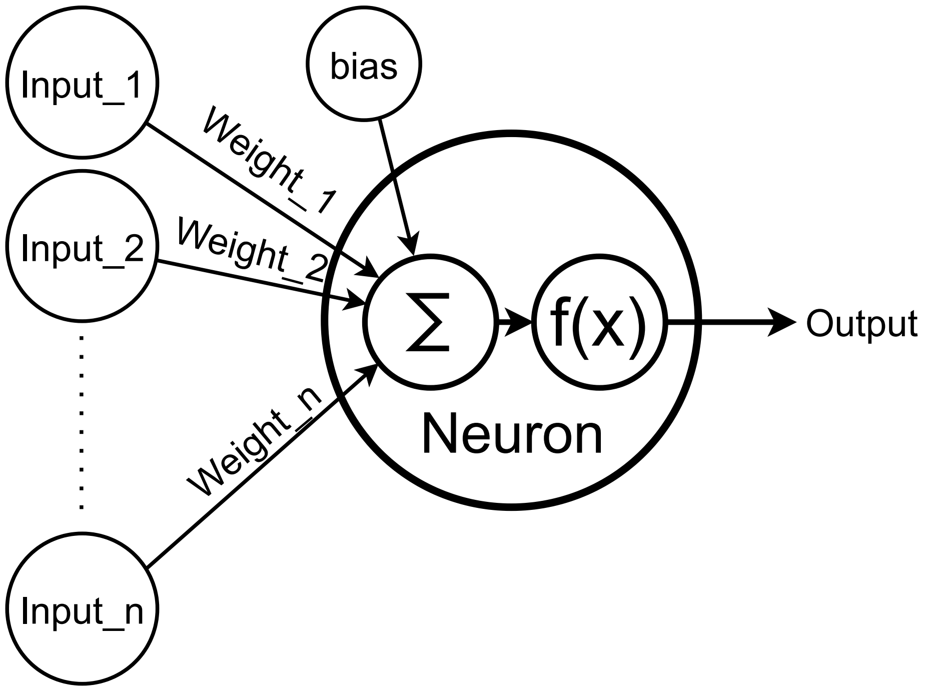


Figure 2.2: Artificial neuron

Of course, the neuron shown in figure 2.2 is the basic building block of NNs, but we can include others, like pooling layers [19] that with a little computational cost reduce the dimensionality of its input, or recurrent cells mentioned later in this work.

The DNNs got their attribute "deep" first by stacking a large amount of neuron layers, but then the term DNN was being used as a designation for all the modern NN architectures and nowadays there is little distinction being made between NNs and DNNs.

In recent years, *transformers* have surged as the state-of-the-art neural architecture for numerous tasks of supervised learning. It all started with their introduction as a new sequence-to-sequence architecture [20], as an alternative to recurrent NNs (RNN). Since then they have yielded great results in natural language processing (NLP) tasks, fueling even the current surge of chatbots, and they even got adapted, e.g., to image recognition [21]. As a rule of thumb, they seem to have a great generalization power; the greater the larger the model employed. However, they also require a lot of training data to achieve these great results.

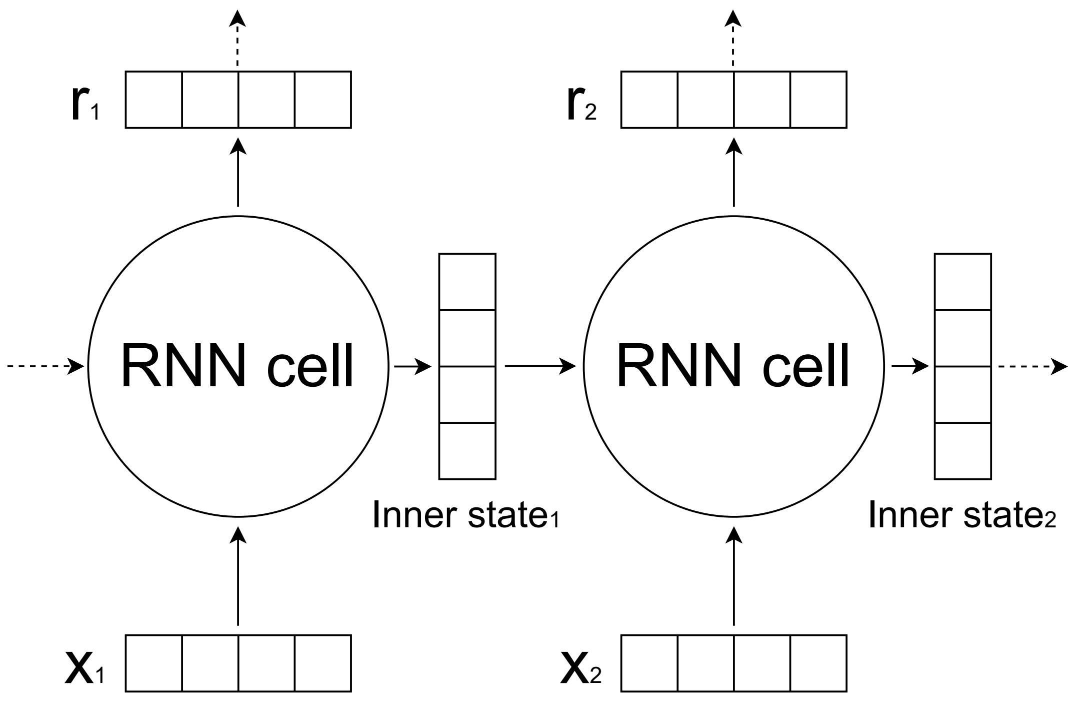


Figure 2.3: RNN cell usage for sequence processing

Before delving into the inner workings of transformers, let us take a brief look at their predecessors, the RNNs. Processing a sequence generally requires being able to interconnect multiple elements of the sequence, which can be arbitrarily distant from one another, to produce an answer. However, the classical feed-forward (FF) NNs cannot do anything like this. Hence, the RNNs were born. They consist of recurrent cells which take an input, output a result of similar operations like the classical FF layer does, but in addition to that they keep an inner state, or memory, and use that as well to compute the output, not just the inputs themselves. The inputs and the memory are used not only to produce the output, but also to update the memory itself. The intended use is then to feed the sequence elements to the RNN cell one by one, using the memory altered by the previous computations to have an effect on the current calculation and inject some knowledge based on the previous elements of the sequence, as can be seen in figure 2.3. It is good to realize that there is one and the same cell in the picture with the same parameters being used twice. For further explanation of RNNs, we highly recommend the article ”Fundamentals of RNNs” [22].

Yet still, from the given facts on RNNs we can see that, for example, in NLP, when we want to translate a sentence, when processing any given word, we have access only to the information on those words that came before in the sequence. This has been partially dealt with by feeding the sequence in both the original and the reversed order to the RNN, and combining the results. But there exists another huge problem. When we process the last element of the sequence, the complete information of the whole sequence has to be stored in the fixed size memory. Thus, the RNNs struggle with long-distance relations in the sequence and long sequences in general — even though it has, in theory, unlimited length of input sequence. So, we can see that the RNNs have a few afflictions. And it is those the authors of transformers set out to deal with.

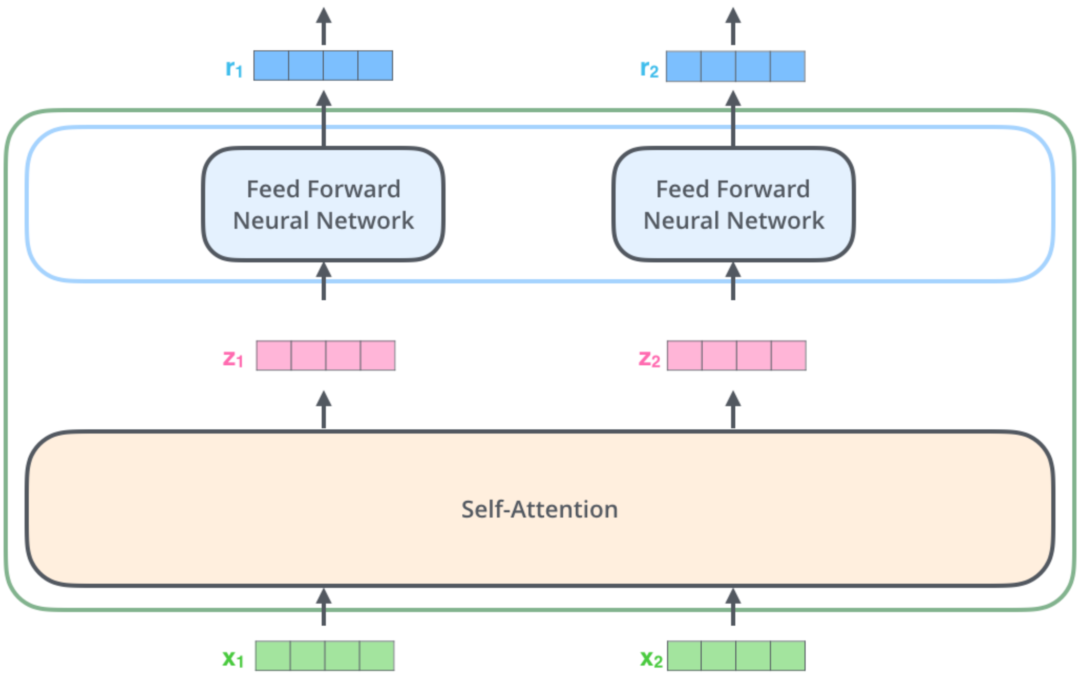


Figure 2.4: Basic transformer block [23]

The transformers interlace FF layers — which are fully connected layers of basic neurons shown in figure 2.2 — with *self-attention* layers, as can be seen in figure 2.4. (In practice, there are also residual connections and layer normalizations to stabilize the training, but we do not talk about them here for simplicity.) And it is the self-attention to which the transformers owe their success, so let us focus our attention on providing a high-level overview of its function. For each element of an input sequence, the self-attention constructs a ”key”, a ”query”, and a ”value” using fully connected layers of neurons. Then, to compute an element of an output sequence on the same position, it takes a combination of all the values, each weighted proportionally to the product of the query belonging to the given position and a key corresponding to the value; hence combining the information from the whole input sequence to produce every single element of the output sequence, as shown by the following equation.

$$output_i = \sum_{j=1}^{n} \text{softmax}\left(\left\{query_i^T \cdot key_k;\; k = 1, \ldots, n\right\}\right)_j \cdot value_j$$

We can, of course, use a mask and for every element hide the part of the sequence that is behind the element, and so using just information that came before in the sequence to derive the output element. This is called causal masking, and such a transformer that uses this masking we then call a *causal* transformer.

The part of an input sequence the transformer performs its computation on is called a context window, and the maximal size of the sequence it can process at once is called a (maximal) *context length.* The layers of transformer do not really care for the length of an input sequence, but transformers tend to perform worse on input longer than what they trained on, and so they are mostly artificially fixed to some maximal context length. This is done by cropping the input. Another aspect contributing to the fixation of context length is that for causal transformers, it is often computationally advantageous to precompute the attention mask, which we could not do if the context length was unrestricted. Moreover, the longer the context, the more expensive the (self-attention) computation.

It would be infeasible and unnecessary to train the key, query, and value encoders for every position of the input sequence, so it is important to note that just one encoder is trained for each of the three and then used on every position of the sequence, just like there is only one RNN cell used for every input in the RNNs, as shown in figure 2.3. The same is true even for the FF subnetworks that can be seen in figure 2.4, those share parameters as well and are basically one and the same subnetwork used multiple times.

However, this only worsens the problem that the self-attention layer in its computation does not have any way of finding out the positions of the individual input elements and of using this information in its computation. Therefore, we add a *positional encoding* to the input of the model. (Not every self-attention layer, just the whole model.) This can be realized by concatenating the input element with its positional encoding, but mostly the encoding and the element are sumed together, as shown in figure 2.5. The positional encoding can be either learned, or we may use some of the predefined ones, like, e.g., sinusoids of different frequencies, as shown in the original paper [20].

The last thing we should mention are *embeddings.* Not always are the elements of the input sequence in such a format that we could plug them into the model straight away without any alteration. As an example, we might use text data. We first have to transform textual data into numerical data. Other example might be pictures, like the ones outputed by Atari games environments as observations, where we need to transform the grid, multidimensional data to vectors. For that, there are embeddings. Those might be world embeddings or character-level embeddings for text, like, e.g., those produced by Word2vec [24], image embeddings computed by convolutional NN (more on convolutions, for example, in book "Convolutional Neural Networks in Visual Computing" [25]), or simply a result of passing the input vector through a FF layer. (Just a side note, we can either use a pretrained embedding (like the aforementioned Word2vec) or train our own from scratch.) The same applies for the output of the model, where a similar, yet inverse process may occur, where we can construct a word from a numerical vector, or embed the resulting vector into the right dimensions, into the right shape. We call such a process a *decoding*.

The usage of transformer for sequence processing would then look as shown in figure 2.5.

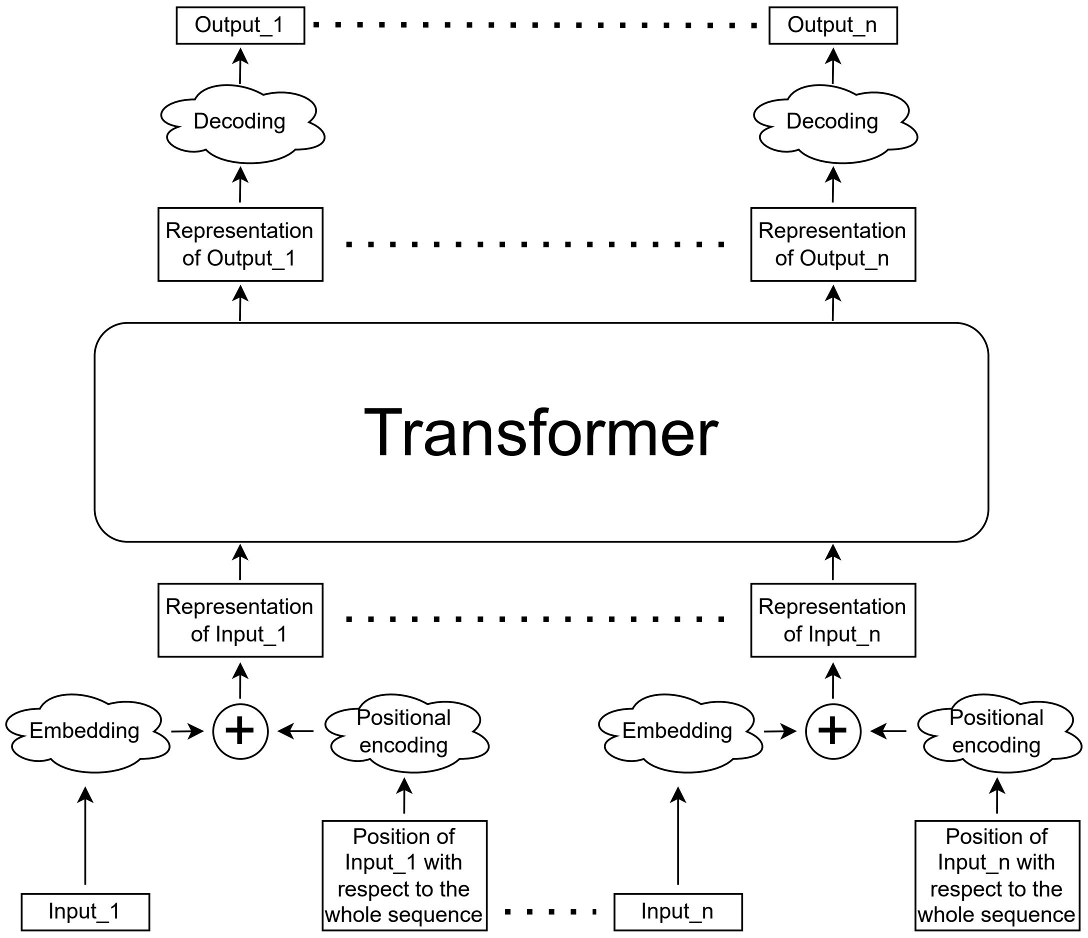

Output_1
Output_n
Decoding
Decoding
Representation of Output_1
Representation of Output_n
Transformer
Representation of Input_1
Representation of Input_n
Embedding
+
Positional encoding
Embedding
+
Positional encoding
Input_1
Position of Input_1 with respect to the whole sequence
Input_n
Position of Input_n with respect to the whole sequence

Figure 2.5: Transformer usage for sequence processing

3. Related work

In this chapter, we outline key concepts of the three algorithms and one model which are closely related to our work. The first one is a specific ES proposed by OpenAI [15] — since the release of the article, it has received the name *OpenAI-ES* in the literature. In the second section we introduce *NS-ES* and *NSR-ES*, being NS and QD versions of the aforementioned OpenAI-ES [26]. And in the last section we show a variant of a transformer architecture adjusted for RL known as a *Decision Transformer* (DT) [27].

3.1 OpenAI-ES

OpenAI-ES is a representative of neuro-evolutionary strategies, which means it is an ES working on vectors of NN parameters as individuals. It also belongs to the class of NES mentioned in Section 2.2.1, so the sole individual in each generation is, in fact, a distribution over the NN parameters, the distribution being a Gaussian, from which the offsprings are sampled each timestep, which are then evaluated, and their evaluation is used to update the parameters of the distribution. However, the authors did not find benefits in evolving the covariance matrix of the distribution nor the individual deviations (or variances), so those are kept fixed and identical to simplify the program and the inter-process communication. Hence, the distribution can then be represented only by its mean, which is an individual in the space of searched NN parameters. The algorithm is designed in such a way that it is highly parallelizable.

Its core algorithm is rather simple. It is all the details that make it work so well — as one might say, the devil really is in the details. But at its core we have several workers; in each iteration, every worker samples a noise from a multivariate normal distribution with zero mean and the unit matrix as its covariance matrix; each worker then adds the sampled noise multiplied by the noise deviation hyperparameter to the parameters of the current solution, being the current mean of the search distribution, and evaluates the modified individual by its rollout in the environment. They then cache the returns as a fitness and send it to every other worker. When this evaluation phase is over, each worker then reconstructs all the perturbations (noises) tried this iteration — that is possible because of the known seeds for every worker — and uses the perturbations to estimate the natural gradient. This is done by taking average of all the perturbations, each multiplied by the fitness the individual modified by the respective noise obtained during evaluation, and dividing the whole average by the noise deviation to renormalize the update with respect to uncertainty. This natural gradient is then used to shift the mean of the search distributions in the parameter space by adding it (multiplied by a learning rate) to the parameters of the mean. This concludes the update phase, and a new iteration begins by the evaluation phase with the updated search distribution.

A pseudocode of the core algorithm, as introduced in the paper, can be found in Algorithm 1. We remind the reader that by computing returns using function F, we mean running a rollout of the agent in the environment and returning its mean return as a fitness.

Algorithm 1 OpenAI-ES core algorithm

```
    Input: Learning rate $\alpha$, noise std. deviation $\sigma$, initial policy parameters $\theta_0$
    Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$

1:  for $t = 0, 1, \dots$ do
2:      for each worker $i = 1, \dots, n$ do
3:          Sample $\epsilon_i \sim \mathcal{N}(0, I)$
4:          Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
5:      end for
6:      Send all scalar returns $F_i$ from each worker to every other worker
7:      for each worker $i = 1, \dots, n$ do
8:          Reconstruct all perturbations $\epsilon_j$ for $j = 1, \dots, n$ using known random seeds
9:          Set $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} F_j\epsilon_j$
10:     end for
11: end for
```

In a practical implementation of OpenAI-ES, each worker instantiates a huge block of a Gaussian noise at the beginning and in every iteration draws the noise from there, which impacts a theoretical independency of the noises, but proved not to be a problem in practice. Next important detail is that it uses a virtual batch normalization (VBN) [28] of input (observations) in its agents to improve the reliability of training. And to reduce variance, it employs antithetic, otherwise known as mirrored sampling [29]. Ergo, for every tested noise ϵ it also tests its opposite noise $-\epsilon$. The last really important detail is that it uses fitness shaping by applying a rank transformation [13] to the returns. For any more details or for discussion of the design choices, we refer our readers to the original paper [15].

3.2 NS-ES and NSR-ES

The previous method is a typical — and quite a powerful and a successful — representative of an objective-based neuro-ES. However, as we commented in Section 2.2.3, not every problem is solvable just by following the information contained in the fitness function. Could we modify this great algorithm so that it searches for novelty instead? And it is exactly the answer to this question that is given in the paper introducing our other two algorithms [26].

First of all, the article presents the NS variant of OpenAI-ES, which they call NS-ES in the paper. For novelty to actually work, we need some diverse behaviors kept in the archive. Therefore, it is better to have a real population, not just the one sole individual the OpenAI-ES operates on. But because in the OpenAI-ES paper, they call "population" all the individuals derived by mutation from the single base individual every generation (which is not an arbitrariness, it is grounded in a terminology of the NES algorithm class the OpenAI-ES is a member of, as mentioned in the previous section), the authors of NS-ES decided to go with the term *meta-population*. Which is usually not large, the authors declare to use a meta-population of size 5 in their experiments. In each iteration, they choose one individual of meta-population with probability proportional to the novelty of the individual computed in relation to the archive. The chosen individual is then worked with and updated just as in OpenAI-ES, the only difference being the usage of novelty instead of fitness during updates. The last detail that needs to be mentioned is how the archive is updated and what behaviors it contains. Every time an individual of the meta-population is updated, its behavior in the environment is recorded and added to the archive, so the archive contains the behaviors of all the past and present members of the meta-population. To conclude this algorithm, let us just quickly state the update rule as a variation on the one used in Algorithm 1. It is the same, just the fitness $F_j = F(\theta_t + \sigma\epsilon_j)$ is replaced by the novelty computed with respect to the archive A, written as $N(\theta_t + \sigma\epsilon_j, A)$. (We just have to keep in mind that in reality, we do not use the novelty itself for the update, but a rank derived from the novelty):

$$\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} N(\theta_t + \sigma\epsilon_j, A)\epsilon_j$$

This leads to the following pseudocode, Algorithm 2:

Algorithm 2 NS-ES core algorithm

```
Input: Learning rate $\alpha$, noise std. deviation $\sigma$, initial policy parameters $\theta_0^m$ for every $m \in \{1, \dots, p\}$
Initialize: $n$ workers with known random seeds, initial meta-population of size $p$ with parameters $\theta_0^m$, and behavioral archive $A$ populated by behaviors of initial meta-population

1: for $t = 0, 1, \dots$ do
2:     Choose a value for $m$ from set $\{1, \dots, p\}$ with respective probabilities $P(k) = \frac{N(\theta_t^k, A)}{\sum_{l=1}^{m} N(\theta_t^l, A)}$ for every $k \in \{1, \dots, p\}$
3:     for each worker $i = 1, \dots, n$ do
4:         Sample $\epsilon_i \sim \mathcal{N}(0, I)$
5:         Compute novelty scores $N_i = N(\theta_t^m + \sigma\epsilon_i, A)$
6:     end for
7:     Send all scalar novelty scores $N_i$ from each worker to every other worker
8:     for each worker $i = 1, \dots, n$ do
9:         Reconstruct all perturbations $\epsilon_j$ for $j = 1, \dots, n$ using known random seeds
10:        Set $\theta_{t+1}^m \leftarrow \theta_t^m + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} N_j\epsilon_j$
11:        for $k \in \{1, \dots, p\} \setminus \{m\}$ do
12:            Set $\theta_{t+1}^k \leftarrow \theta_t^k$
13:        end for
14:        Add behavior of $\theta_{t+1}^m$ to archive $A$
15:    end for
16: end for
```

As the next step, the authors wanted to derive a QD modification of OpenAI-ES. The first one they came up with was an algorithm they called NSR-ES with the ”R” standing for ”Reward” (fitness). It is almost identical to the NS-ES, except for the fact that for update it uses mean of fitness and novelty. So during data collection, or the rollouts in the environment to be more specific, they record the behavior and the return obtained, they compute novelty of the behavior, after the data collection is finished, they rank-normalize both novelties and fitnesses independently and then create the final weight for the update as mean of the novelty-rank and fitness-rank of the tested individual. The update rule - again, with us keeping in mind we use ranks in practice, not the raw fitness and novelty - looks as follows:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} \frac{F(\theta_t + \sigma\epsilon_j) + N(\theta_t + \sigma\epsilon_j, A)}{2} \epsilon_j$$

The NSR-ES is the QD algorithm we use in our work because of its implementational simplicity and because it is fully sufficient for our purposes. Nevertheless, the authors derived one more QD algorithm, NSRA-ES, where "A" stands for "Adapt". Again, this is almost identical to NSR-ES; just the update rule is different:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} \left(wF(\theta_t + \sigma\epsilon_j) + (1-w)N(\theta_t + \sigma\epsilon_j, A)\right) \epsilon_j,$$

where w stands for an adaptive weight, initially set to 1.0, but adaptively modified according to the following rules: It is decreased towards zero if the performance of the whole algorithm stagnates across a fixed number of generations (signalizing reaching a local optimum, and so even the need for increased exploration) and continues to be decreased until the performance increases again, at which point the w begins to get increased towards 1.0 again. And even though this is a much stronger algorithm than NSR-ES, it is also more complicated to implement, and mainly, in most of our experimental setting it would work just as OpenAI-ES, because there is not much deception in the environment we are working on in this thesis, hence the w would be most of the time set to 1.0. Therefore, we use NSR-ES as a representative of QD methods.

For both previously mentioned algorithms, NSR-ES and NSRA-ES, the pseudocode would be almost identical to the one in Algorithm 2. The only differences would be that in the data collection part, they would gather the information on fitness of the individuals in addition to the novelty scores, and in the update section the corresponding update rule described above would be used.

3.3 Decision Transformer

As mentioned in Section 2.1, in the RL we want the agent to choose actions at timesteps zero, one, and so on, such that it maximizes its return. This can be, however, viewed as a sequence modeling problem. And then, naturally, the best sequence processing architecture available, the transformer, comes into focus. Thus, the Decision Transformer (DT) was introduced [27].

Its main idea is that we want the agent's policy to produce an action based on not just the last observation, but the whole history (or the part which squeezes into its context window) of past observations and undertaken actions. This would help mainly in non-deterministic environments, where the current state-of-the-art architecture is RNN. RNNs, however, as mentioned in Section 2.3, have significant problems with clarity of information kept from distant past, and hence long-term relationships in the environment. Therefore, we want to employ transformers to deal with these problems, just as they have dealt with them in the NLP. And to have some way to affect the agent's performance, we add conditioning on *return-to-go*, which is a return we want to obtain from now until the rest of the episode.

So, let us take a look at the proposed architecture itself. DT consists of a causal transformer, embeddings for observations / states, actions, and returns-to-go, position encoder, and linear decoders to transform the output of the transformer to actions, as shown in figure 3.1.

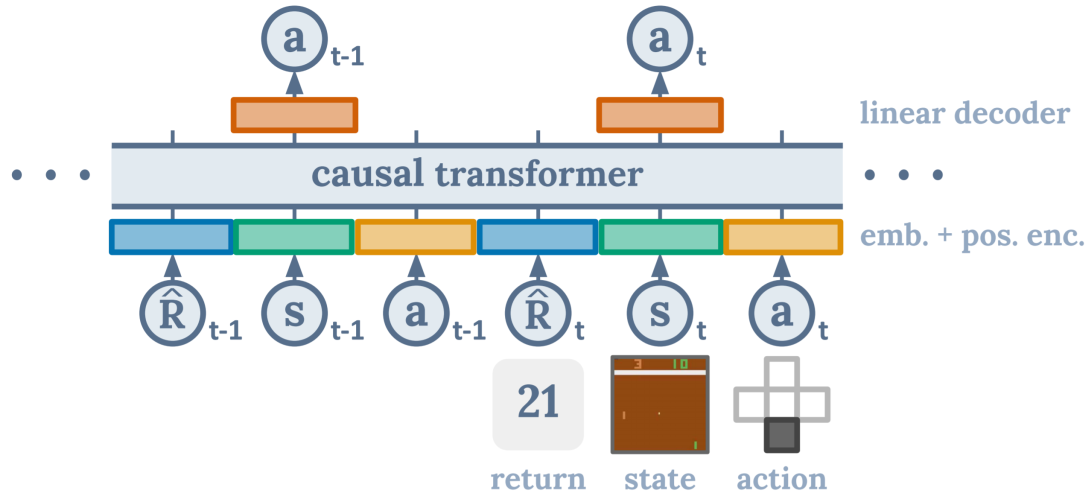


Figure 3.1: Decision Transformer architecture [27]

Every timestep we feed the model with a sequence of past triplets return-to-go, observation, and action performed, adding the current return-to-go and observation (and possibly a placeholder for the yet unperformed action). They get embedded by their respective embeddings, the positional encoding is added, and everything is then fed through the transformer. We then decode the last output state of the transformer to obtain the action to carry out. An important difference from the usual transformer is that every part of the timestep triplet, meaning return-to-go, observation, and action belonging to one particular timestep, shares one positional encoding, as opposed to the classical transformer, where every input sequence element gets its own. A pseudocode for the action inference by DT can be seen in Algorithm 3. Returns-to-go are constructed in a kind of recursive way. The user has to supply the original one for the first timestep (which has the meaning as the desired performance of the model, in other words target return), while for all the following timesteps the return-to-go is constructed by subtracting the last reward obtained from the return-to-go belonging to the previous timestep.

In the original paper, the authors analyze the influence of the target return on the DT's performance. They tried the architecture only in an offline RL setting, which means that the agent is shown some diverse dataset of trajectories (rollouts in the environment) and it should learn to interpolate the best possible behavior in an situation without additional learning, or data. In this setting, they observed that the model is quite capable of matching the target return up to the highest return seen in the dataset. (So, when prompted with some target return, it will

Algorithm 3 Action prediction using Decision Transformer

```
   # returns_to_go, observations, actions: input sequences
   # timesteps: sequence of timestep indices the values
   #    in the other input sequences correspond to
   # EncodeTimesteps: positional encoding function
   # Embed*: embedding layers / NNs
   # Stack: interleaves the sequences
   # Transformer: causal transformer
   # GetLastActionState: gets the hidden state in the output
   #    sequence from which the action should be inferenced
   # DecodeAction: linear decoder (FF layer)

 1: function DT($returns\_to\_go$, $observations$, $actions$, $timesteps$)
        # compute positional encodings and the embeddings
        # for the input sequences with the respective
        # positional encodings added to each element
 2:     $pos\_encodings = \textsc{EncodeTimesteps}(timesteps)$
 3:     $obs\_embeds = \textsc{EmbedObservations}(observations) + pos\_encodings$
 4:     $act\_embeds = \textsc{EmbedActions}(actions) + pos\_encodings$
 5:     $rtgs\_embeds = \textsc{EmbedRtgs}(returns\_to\_go) + pos\_encodings$

        # interleave the embedded sequences
 6:     $input\_embeds = \textsc{Stack}(rtgs\_embeds, obs\_embeds, act\_embeds)$

        # use transformer to get hidden states
 7:     $transformer\_output = \textsc{Transformer}(input\_embeds)$

        # select the correct hidden state for action prediction
 8:     $last\_action\_state = \textsc{GetLastActionState}(transformer\_output)$

        # predict and return the action to be taken
 9:     return $\textsc{DecodeAction}(last\_action\_state)$
10: end function
```

end the episode with a return close to the desired one. And what more, on some of the environments tested the DT was sometimes capable of extrapolation and was able to match the target return even for values higher than the highest seen in the training dataset. However, in other environments the performance deteriorated for target returns higher than the highest trained on.

The last mention in this section should be about existing DT's alternatives like Trajectory Transformer [30]. So, as we can see, there exist multiple different approaches to solving the sequence modeling version of RL using a transformer. Nevertheless, at the time of preparation of this work, Trajectory Transformers and DTs reported comparable results; thus we chose to work with DT in this thesis.

4. Approach

In this chapter, we first go through our goals, our approach to the given problem, and the methods used; then we take a look at interesting implementation highlights.

4.1 Methods

In Section 2.2.2, we described that the ES are being used more and more often as a full-fledged alternative to the gradient methods mainly due to their parallelizability on the RL problem, where the agent, or its policy, to be more specific, is often realized by a NN. In Section 2.3, we pointed out the qualities of transformer architecture, and we have seen how they can be taken advantage of for the RL in Section 3.3. Yet at the time the work on this thesis commenced, we had not found any paper that attempted to interconnect these two research topics. Therefore, we set out to do just that.

We chose a state-of-the-art ES for RL — OpenAI-ES — and chose DT from the fairly equivalent alternatives mentioned in Section 3.3 as a representative of transformers specifically adjusted for RL, and proceeded to try and train the latter using the former. We decided to include NS techniques as well, and thus we included training using NS-ES and NSR-ES algorithms.

Yet, because the transformers are a bit more complicated and rather larger models than just the ordinary FF models used in the papers introducing OpenAI-ES and NS-ES with NSR-ES, we thought it might be advantageous to first pretrain the DT before feeding it to the ES. (After all, the DT was introduced as an offline RL model, meaning it was trained in a supervised-like manner in the original paper.) The reason for this is as follows: When having a large and complicated model, where the probability is low that a random sampling around a random initial model will yield us models such that the information obtained from their fitness is well illuminating as to in which direction to best shift the weights of the model, then we may utilize another approach than ES to get a model that is no longer random and does something interesting, though perhaps basic in the environment, and improve this model further by the means of ES. We achieved this by having a smaller pretrained model (like a classical FF NN) — which can be trained using any arbitrary RL algorithm, be it a gradient one or an ES — and using this smaller model to generate trajectories, or rollouts, that are then cut to chunks the size of the context length of our DT (by sliding a window over the trajectories), shuffled, and stacked into batches, which are then fed to the DT's supervised training. This should help change the weights of DT from random to relatively reasonable and make the job of ES easier, since it does not start with a completely randomly behaving model. Just a quick side note, this is not equivalent with the offline RL from the original paper [27] — the data are not that diverse, because they are generated by a single model. It is more a behavior cloning than an offline RL. For our purposes, however, this is fully sufficient. The high-level overview of the algorithm for pretraining DT as described above can be seen in Algorithm 4.

Let us now dive deeper into the individual steps of the algorithm. As for

Algorithm 4 Pretraining the DT

```
    Input: RL algorithm $A$, DT $m$, smaller model $m'$, environment $e$

1: Train $m'$ utilizing algorithm $A$ in environment $e$
2: Create a generator $g$ using $m'$ and $e$, which generates trajectories of $m'$ in $e$
3: Train $m$ on sequence prediction task using trajectories generated by $g$
```

the step 1, we utilized SAC algorithm [31] to train a simple FF agent with two hidden layers consisting of 256 neurons each for the Humanoid environment based experiments. For the Atari environment, we prepared to use an already pretrained agent [32], yet we did not utilize this in the end for reasons mentioned in Section 4.2.

For step 2, we collect trajectories of the rollouts of the model in our environment. We then slide a window of width of the context length of the DT, sliding it just by one step every time, hence collecting overlapping slices of the whole trajectory.

We then shuffle the slices and arrange them in batches. These batches are then used in step 3 to train the transformer to predict the next token of the subsequence. We interlace the data generation and the training of the DT, ergo there is an opportunity for parallelization in this approach.

The last design decision that had to be made for utilizing the pretrained agent is the following: For the NS-ES and NSR-ES, that have a meta-population from which they choose an agent to further train every iteration, we decided to initialize the whole meta-population by the pretrained agent, ergo we have a meta-population consisting of the same agents. We argue that this is not such a problem for two main reasons. First, the environment evaluation is usually a bit different every time due to different start conditions etc., thus the behaviors of the agents will be different. And second, the job of any NS algorithm is to diversify the population, therefore we should soon enough end up with a diverse meta-population. Of course, trying out different seeding strategies remains a possible topic for future work.

4.2 Implementation highlights

Here in this section, we want to highlight details of our implementation of OpenAI-ES and details of models used.

We found the official OpenAI implementation to be neither user-friendly nor well-commented (nor completely adherent to the original paper - they deviate, e.g., in the way the multiprocessing data sharing works), did not have any intention of running the algorithm on Amazon Elastic Compute Cloud (for which the original code is written), wanted to have more control of the model, and needed to be able to tweak it to incorporate NS and environments used, plus to accommodate the use of DT as a policy model. Therefore, we decided to reimplement the algorithm from scratch. Although it brought along some (in future fixable) disadvantages, like having to rely on a single multi-processor machine instead of a cluster, it yielded much more readable, versatile and easily expandable code (planned to be released as a public package in the future), gave us a deeper understanding of the details of the algorithm — ergo, we could better understand, anticipate, summarize, and comment on the results of our experiments — and allowed for seamless integration of DT and NS techniques. Our implementation can be found here: `https://github.com/Mafi412/es_contra_dt`

The current implementation is based on the Python multiprocessing package and shared memory, thus it cannot be used to compute on a cluster and requires a single machine with multiple processors. This was not such a huge problem with most of our experiments, yet we reached its limits when experimenting with larger model and computationally slower environment, where one iteration of the algorithm ran a full day on thirty processors (where our maximal capacity is currently circa sixty processors per machine), which was unfeasible and we had to remove similar experiments from our work to put them off for later. We intend to update the implementation in the future, in order for it to reach its full potential. Then, we believe, it will help further experiments and new improvements in the area, because of much better readability and versatility (compared to the original code by OpenAI).

As mentioned above, our implementation is quite versatile. It is easy to use with custom models and environments — and even behavior characteristics for novelty-based searches. We define an abstract wrapper class for both the environments and the models, where the custom entities are held inside as a field, and it suffices to override and implement a few methods, which are then used in rollouts. Thanks to the model wrapper class, one can even easily integrate the use of the VBN of input into their model — without which the OpenAI-ES has proved to be brittle in the original paper — because the wrapper contains the VBN parameters and keeps track of it during training. As for how to add the custom behavior to the training, we derived a new abstract class for environment from the previous one, where we add one another method returning the agent's behavior in the environment so far. Therefore, the responsibility for keeping track of the behavior lies on the environment, which makes sense, because the behavior is defined with respect to the environment. The behaviors then have their own abstract class from which they should inherit. It implements basic comparison method, and the custom behaviors should then override this method. Therefore, it is easy to implement a new behavior. We even prepared some general behavior characteristics to be readily available. We have, for example, CombinationBehaviorCharacteristic which takes multiple arbitrary behaviors with corresponding weights and then sums the values from each individual comparison of each individual subbehavior multiplied by the respective weight. We have UniformCombinationBehaviorCharacteristic which is derived from the previous one and just weights all the subbehavior values equally. These are composite behaviors, so they require some basal behavior characteristics that they can combine. Ergo, we prepared some quite universal basal behavior characteristics. We have multiple behaviors working with sequences of taken actions, we have behaviors comparing the final or the average states, and we can use the fitness as a behavior characteristic, comparing two fitnesses using their ratio.

The next thing we would like to highlight are some differences in our and OpenAI's implementation, so let us go through them here:

- First and foremost, in the paper [15], they mention the use of the weight decay as a form of keeping the effect of newly added noises still significant enough. So after every update, we decay the weights of the model; therefore, it would not happen that there could be weights grown to such sizes that the newly added noise does practically not change the function of the model. This reasoning for the use of weight decay makes sense and tells us why we want to use precisely weight decay. Nevertheless, in their implementation they use L2-regularization. This is fine for some optimizers, as for, e.g., SGD or SGD with momentum it is virtually the same as weight decay (when rescaled by the learning rate). However, as mentioned in paper "Decoupled Weight Decay Regularization" [33], it is not the same, for example, for ADAM optimizer, which is nonetheless exactly the one they are using in their experiments. Our implementation, on the other hand, remains true to the original paper and uses proper weight decay.

- As mentioned in Sections 3.1 and 3.2, during update the gradient estimate should be normalized with respect to the uncertainty, it should be a natural gradient estimate, thus the dividing the average of the weighted noises by the standard deviation should be used. Despite that, in the original OpenAI code this is not done, while we have it implemented theoretically correctly. They probably hid this into the value of learning rate (which is different from ours in our experiments); however, then the learning rate and the noise deviation are unnecessarily coupled hyperparameters. We did not find any justification for why they omitted this detail.

- In their code, the original authors wait for the workers to collect both at least some number of episodes and at least some number of timesteps before concluding the iteration, computing an update, and beginning a new iteration. Nevertheless, because of their hyperparameter setting, where they require just ten times more timesteps than episodes, we decided not to undergo the work required to implement the lower bound on collected timesteps. The reason is that in the environments we used in experiments, one episode was at least around fifteen timesteps long, even with a random model. Even in most other environments, there is no problem with this simplification. Nonetheless, it remains an open question for us whether to include this functionality when we remake the multiprocessing to be capable of operating on cluster and not just the single machine.

- During rollouts, they add noise to the outputs of the network. However, because we did not find sufficient reasoning for this in the paper, did not see the point in it, and believe it — contrary to their claims and reasoning for this in the paper — rather obscures the true performance of the new parameters with added noise, we decided to leave our implementation without it at first. And because it had shown performance comparable to their implementation, we decided not to include it even in the final version.

- In their policy, they clip the observation after the normalization by the VBN by an arbitrary values of $-5, 5$ before feeding it to the model. Ablation study of this bit had shown it has no effect on the result, so we removed it from the code, because we did not find any reasoning for the exact values used and wanted to show the good functionality of the algorithm for any arbitrary model in an arbitrary environment, where it would be non-trivial

— more so without the knowledge of a reason why there were the exact values for clipping used in the original paper — to derive come constants to be used as bounds on the observation.

- The last distinction to be mentioned is purely aesthetic. It is the ranking used to rank the fitnesses and/or novelty scores to use them during the update. In the OpenAI's code, they use a centered rank, the idea of which is to take the ranking from $[0, 1]$ and center it by subtracting 0.5, obtaining ranks from $[-0.5, 0.5]$, whereas even though we use a centered rank as well, we understand it a bit different and in the end arrive at the final interval $[-1, 1]$. Thus, every our rank is twice theirs. The difference is really purely aesthetic because using their ranking is equivalent to using ours with half the learning rate. We decided to use the ranks from $[-1, 1]$, because of the interpretation of the ranks. For the best individual, its rank is 1, which means we want to go "1 full step" (modified in reality by learning rate) in its direction from the current individual, whilst we want to get a full step away from the worst, hence the rank of -1.

To conclude this section, the last thing we want to discuss are our observations regarding DTs. We did use the original implementation in this case, but had to slightly modify some minor details, and thoroughly rewrite the training code to work with our data. Some other details we did not fancy, yet still we decided to keep them there to preserve the original architecture for the sake of reproducibility and comparability and just comment them here.

- The original authors provide two DT implementations, one for Atari games environments, the other for Gym MuJoCo (Multi-Joint dynamics with Contact) environments, together with an example of usage in the form of the code for the offline RL with their dataset. For the MuJoCo model, we can see that for sequences shorter than the context length, they add padding to the left and use masking in order for the self-attentions not to use this portion of the input. However, this is completely unnecessary, because we have a causal transformer, therefore we already have a mask in use. If only they padded the sequence from the right instead of from the left, they would not have to create and add another mask and it would make much more sense. The only reason we were able to come up with why they might want to do it their way is so that they would then always have the output action on the same spot of the output for every input sequence. Nonetheless, this is really meaningless, since we always know the length of our input sequence when feeding it to the model, and then it is trivial to extract the right token from the output.

- Another thing regarding passing an input sequence to the model is keeping data from the past to be used in the future input. In the original code, the authors do not deal with sequences already longer than the context window. They always just append the new data to the list, keeping the whole history and passing it whole to an inference function, which just crops the sequence to the required length (not modifying the data stored) and passes it to the transformer. This proved to be no problem in MuJoCo environment, but for some reason leads to a severe slowdown in Atari environment. Thus, we keep the stored sequence cropped to match the context length in the Atari version of DT. (We did not use this DT version in the experiments in the end because of the time concerns regarding the rollouts of DT agents in Atari environments, but the implementation is prepared for future experiments with the upgraded implementation of the ES or on a more powerful machine, and we regard this detail important for anyone who would decide to experiment with DT themselves.)

- In the original training loop for the MuJoCo version of DT, besides the fact they utilize their offline RL benchmark dataset, they also differ from our modification in what they do with the data before it is fed to the model. They take full advantage of having the entire dataset up front and compute the mean and standard deviation of the states encountered in the data. They then utilize this and normalize the data before those are passed to DT. They save the values of mean and deviation and use the normalization with these values even during inference. Compared to that, we do not use any normalization. This is for two main reasons. First, we generate the training data on fly, so we cannot precompute the values. We could have used the VBN, which is later used in ES training anyway, but we wanted the pretrained agent to be as generic and basic regarding the input modification as possible, so that the experiments could show the ability of OpenAI-ES to further train any pretrained agent. And this is the second reason why we chose not to perform any normalization during the pre-training.

- To conclude the remarks on DTs, we would like to discuss the functionality of return-to-go tokens. In the original experiments of the authors, they showed quite a good correlation between the desired return given to the model and the model's final performance, as discussed in Section 3.3. This is a result of the offline RL setting, for in offline RL datasets there are diverse trajectories generated by diverse policies. However, in our setting, the training data was generated by a single policy (the smaller model), as mentioned in Section 4.1. And as could be expected, this resulted in almost complete non-responsiveness of the model to the desired return. And just as a little foreshadowing, this is not something the ES would fix in the pretrained agents in the current form, nor is it capable of teaching the DT to take the desired return into account when training them from scratch. Hence, the DTs trained by the OpenAI-ES or ”fine-tuned” by it are completely desired return agnostic and do not respond in any way to return-to-go tokens; they just learn to ignore it, for they are not incentivized during the training to care about them, they just want to obtain highest return possible. Yet, we have an idea for how to deal with this, which is presented in Chapter 6 as a possibility for a future work.

5. Experiments

Here in this chapter, we first walk through the setting of our experiments and mention some details that are common throughout the experiments. We then dive straight into the individual experiments, starting with the plain ES, continuing with the NS variant, and concluding with a section concerning the QD algorithm. Each of these experimental sections consists of three subsections. First, we discuss experiments on a FF network same as in the original paper — which is partially a correctness check of our implementation, partially a replication experiment of the original study, and partially a way to familiarize ourselves with the algorithm behavior before proceeding to a more complex model. We then focus on the performance of the algorithms on DT without pretraining. And finally, we take a look at how much the given algorithm is capable of exploiting a pretrained model (again DT) for its purposes. The only exception to this scheme is the first experimental section, Section 5.2, which discusses the ES, for we have another subsection in this section regarding a few hyperparameter values search studies for those hyperparameters that differ between our and the original implementation.

5.1 Experiments setup

Let us begin by describing the experiments and their setting. The basic idea is, as stated in Section 4.1, to test the ability of the three algorithms mentioned in Chapter 3 — OpenAI-ES, NS-ES, and NSR-ES — to train DT architecture to perform well in a given environment. We chose MuJoCo [34] Humanoid locomotion environment available in OpenAI Gym [35] as a testbed of these algorithms, for it is one of the more complicated environments for a continuous control, and thus a good benchmark of the algorithm performance. Another reason is that Humanoid is the environment used in the original papers for both OpenAI-ES, and NS-ES and NSR-ES. We tried to perform experiments on Atari environments [36] as well, but, as mentioned in Section 4.2, this had to be postponed for future work due to time constraints and a lack of a suitable computing power.

As mentioned previously, we begin testing individual algorithms by reexamining their performance on a FF model, as used in the original papers. This model has two hidden layers consisting of 256 neurons, each with tanh as the activation function. This yields a model with 166,144 parameters. Compared to this, the DT used for the Humanoid has 825,098 parameters, whence we can see the rise in complexity originating from the growth of network size alone when deploying the DTs. The DT hyperparameters used are identical to those used by the authors in the DT paper and their experiments. (For Atari, the DT has 2,486,272 parameters.) For each experiment, we conducted three runs of the training and aggregated the results in our plots. For every run of each of the experiments presented, 30 workers were used utilizing 30 CPU cores (with one master handling synchronization, evaluation, and saving the agent) to give a context to the wall-clock time plots. As we already mentioned in Section 4.2, with the current algorithms the DT learns to ignore the return-to-go tokens. Still, we assume it might have had some minor influence on the training, and therefore we disclose the desired return all the DT models were fed at the beginning of each episode, whose value was 7000 for the Humanoid DT.

The last thing to be mentioned regarding all the experiments are the algorithm hyperparameters used. In most cases, they are identical to those used in the original implementations. Nevertheless, there are some differences. We will postpone talking about number of iterations to the individual experiment sections, but there are two other significant distinctions, both further explained in Section 5.2.2: Instead of ADAM optimizer, we use SGD with momentum (SGDM); and we deploy different learning rate (called stepsize in original implementation). All the hyperparameters used can be found in the source codes as default values of the input arguments in the training scripts, denoted as *train_*.py*, the only exceptions being the experiments with pretrained agents, which use a different learning rate and noise deviation, those both having a value 0.01 instead of the default one, and turn off the VBN by setting the update_vbn_stats_probability hyperparameter to -1. Without turning off the VBN, the pretrained agent suddenly starts getting completely different inputs and thus cannot function properly.

To conclude this setup section of this chapter, we will talk about the behavior characteristic used for the experiments using novelty (those testing NS-ES and NSR-ES). Even though we implemented multiple characteristics, as mentioned in Section 4.2, for Humanoid experiments we still used a simple characteristic utilizing the agent's final coordinates and comparing them using the Euclidean metric, because that is the one being used in the original paper. This works well in the NS setting, nonetheless it brings along some disadvantages for QD approaches. But this will be further discussed in the respective section.

The source code and experiment results in the form of logs, plots, trained agents, and video recordings of rollouts of the trained agents can be found here: `https://github.com/Mafi412/es_contra_dt`

5.2 Evolution strategy

This section is dedicated to experiments with OpenAI-ES. First, we try to replicate the results of the original paper with the FF model; then we talk through a few case studies explaining our different setting of some of the hyperparameters as compared to the original one; after that, we take a look at the performance of the algorithm on an agent with DT as its policy; and we conclude by the last subsection, where we experiment with pretraining this agent first, then training it further using the ES.

Before diving into the individual experiments of this section, we briefly talk about the exploration capabilities of ES, which are often mentioned as one of their advantages over the gradient optimization techniques. These can be further enhanced by utilizing the NS, but even when using plain ES, when running the algorithm several times (with different seeds), the resulting agents may differ in what approach they adopt to obtain high fitness, because the best direction in which to move in the parameter space is estimated using random samples. In the Humanoid environment, this would mean that the agents learn distinct styles of moving forward. And really, when experimenting with OpenAI-ES on both the FF model and the Decision Transformer, we can see that the agents learn distinctive gaits. We can see this in the videos disclosed in the data section of our GitHub repository[1] or when we simulate the trained agents stored ibidem. The only exception is the training of the pretrained agent, but here it could have been expected, as the experiment does not run for a long time (in a manner of iterations) and to completely change the gait, the agent would probably first have to substantially decrease its return, whence we see that the ES would rarely allow for relearning the manner of moving while still utilizing the pretrained model. Thus, we can observe only the refinement of the pretrained gait in all three runs of this experiment.

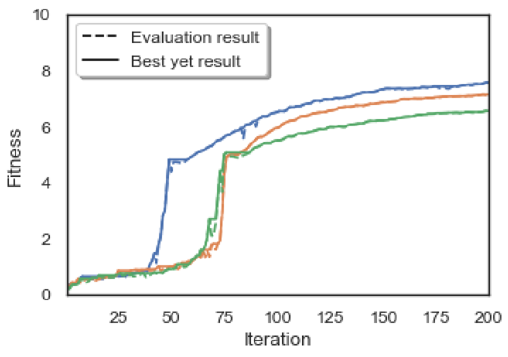
5.2.1 Feed forward network

In this subsection, we show the results of our replication experiment for the OpenAI-ES algorithm applied to the FF model, as done in the original paper. From Figure 5.1, we can see that the results are overall very good and the trained agents are quite high-performing. And, as mentioned previously, we can observe quite distinct behaviors in the resulting agents from the three runs as well. The recordings of rollouts of the three agents can be found on the GitHub repository referenced in this work. Hence, we can confirm the result of the original paper and state that OpenAI-ES is fully capable of standing toe-to-toe with gradient methods as an approach to solving RL problem.

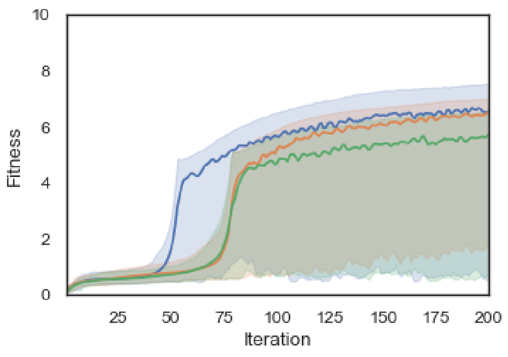
Here, we can see one great strength of the ES being used for RL. If the environment has short episodes for random or not-so-well-performing agent, then before the agent learns to do something interesting and the episodes start to get longer, one of the biggest benefits of gradient TD methods — being able to learn timestep-to-timestep instead of just after the whole episode is concluded — is not so profoundly emphasized. So the ES have virtually no disadvantages in this part of the training, while bringing along the advantages of the parallelization and the aforementioned ability to obtain diverse agents. We can see that most of the training is spent — in terms of both the runtime and the wall-clock time — in the second, the "fine-tuning" part of the training. We hypothesize that an approach combining ES in the beginning of the training — providing initial exploration and parallelization boost to the training — and gradient based TD in the later stages of the training — fine-tuning the agent pretrained by the ES part — can provide the exploration benefits of ES and sample efficiency of gradient methods, which proves useful in case only a limited number of CPUs are available.

The last point we would like to raise in this subsection is the correlation between the fitness of the mean of the search distribution and the fitness of the population sampled from the distribution. What can be observed in figures 5.1a and 5.1b is the fact that the mean of the distribution, representing the current solution of the algorithm, is better than 97.5 % of the individuals sampled, whence we can conclude that the search seems to be more about getting away from the bad solutions than going towards the best, at least in this instance. Ergo, we get this interesting lag in the plots between the fitness of the mean of the search distribution and the mean fitness of the population sampled from the distribution.

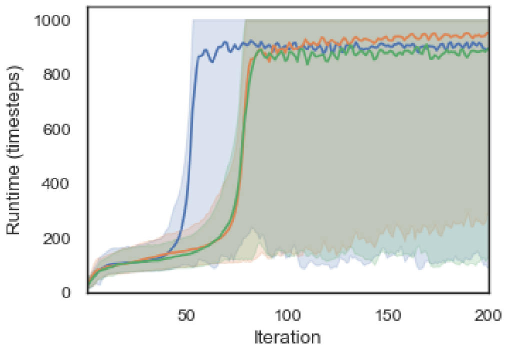
[1]Recordings of the agents trained during a specific experiment and during a specific run of the experiment can be found in the respective data directory with logs and checkpoints of the given experiment and run. This can then be found using a guide contained in the repository's ReadMe file.



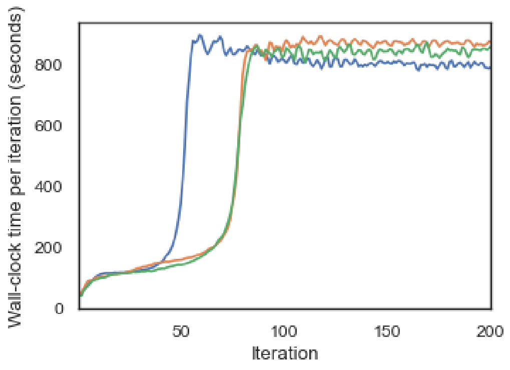
(a) Evaluation results



(b) Population fitnesses



(c) Runtimes



(d) Wall-clock times

Figure 5.1: OpenAI-ES on a FF model
In Figure 5.1a, we can see the results of evaluations after every iteration of the ES. Figure 5.1b shows the fitness values of the whole populations throughout the iterations (mean and percentile interval with standard width, ranging from 2.5 to 97.5 percentiles). Data plotted in Figure 5.1c are then the runtimes of the populations (again mean and 95% interval). And finally Figure 5.1d represents the wall-clock time it took the individual iterations to finish.

5.2.2 Feed forward network — Case studies

In this subsection, we show the results of experiments explaining our choice of values for those hyperparameters, whose values differ from those used in the original paper. These hyperparameters are mainly the learning rate, where we generally use the value 0.05 instead of 0.01, the optimizer used, instead of ADAM we use SGDM, and the size of population. All these experiments are performed with the FF model and Humanoid environment just as in the previous section, Section 5.2.1. All the hyperparameters — except those being examined — are set to their default values.

Let us start by talking about the learning rate. In the original implementation, they use the value 0.01 for the learning rate, however they also have a different update step. They use different optimizer — which we will get to later; they use L2-regularization while we utilize proper weight decay; and they do not normalize the gradient by the uncertainty while we do, dividing the weighted sum of the noises by the noise deviation. Therefore, we decided to search for our own best value for the learning rate. We ran several runs of the program, each with different

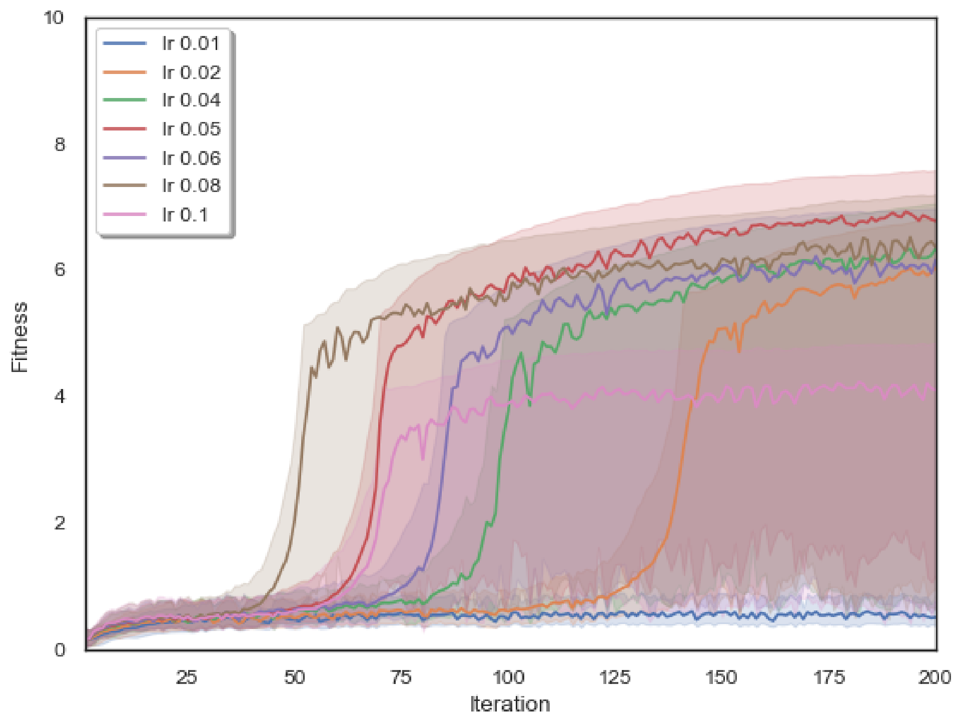


Figure 5.2: Various learning rates
This figure depicts a mean and 95% interval of the fitness of the whole population derived from the solution model using noise for various learning rates.

value of the learning rate. Every run was given the same seed, and the optimizer used was SGDM, which is the default optimizer for our experiments.

In Figure 5.2 — which shows the development of the population fitness — we can observe that the value for the learning rate used in the original paper, that being 0.01, is completely unusable because it is unable to train the agent and improve its performance with our implementation. We can see that initially the best performing value is 0.08, nonetheless it is outclassed in the later training by the value 0.05, which is slower (but still the second fastest) to begin improving the agent, but then quickly takes over and outperforms the aforementioned value. Hence, the value of 0.08 of the learning rate is not suitable for the fine-tuning part of the training, where the overall strategy is discovered and it is only further refined. Moreover, the swift rise of the agent trained with the learning rate of 0.08 towards higher fitness might be explained simply by a favorable seed. This conclusion supports the fact that even though 0.08 initially performs well and 0.05 is overall the best, the run with learning rate of 0.06 — this value being in between the former two — is worse than the runs with both the values 0.05 and 0.08. Thus, we decided to use the value 0.05 for the learning rate hyperparameter in all our experiments.

Another matter we discuss here is our choice of the optimizer used for the training. In the original work, they use ADAM optimizer, however when we tried to run the training with this optimizer using our implementation, it turned out

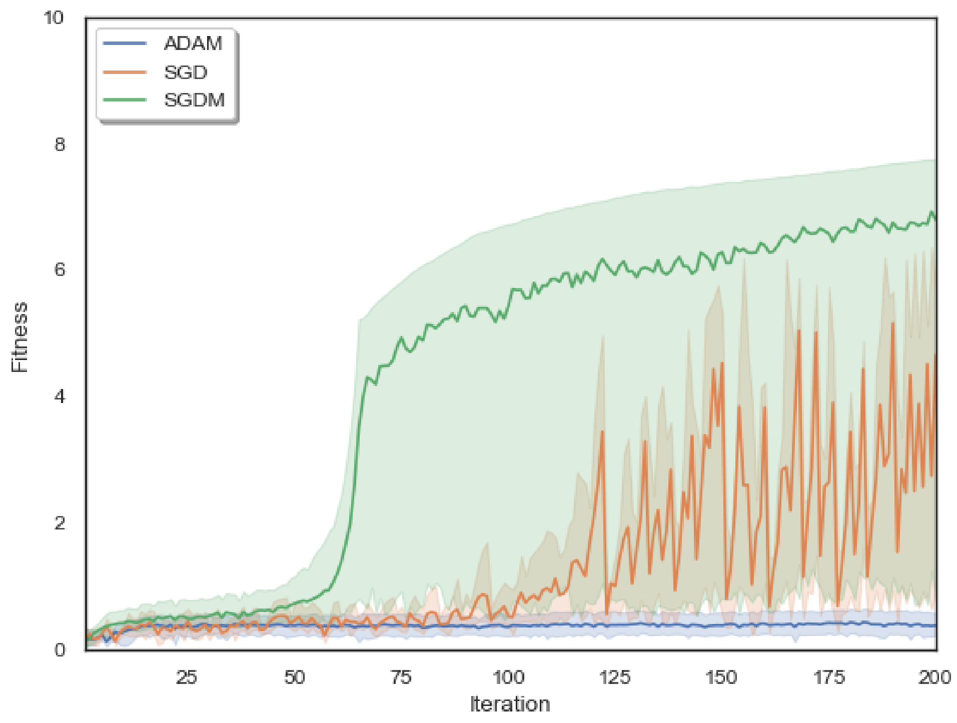


Figure 5.3: Various optimizers
This figure depicts a development of the fitness of the whole population derived from the solution model using noise, more specifically its mean and 95% interval, throughout the iterations for runs with various optimizers.

it does not work quite well. We examined our implementation of the optimizer, went through its computation step-by-step, but we still did not find anything wrong with it. So we do not really know the reason our implementation of the algorithm does not work with our implementation of the ADAM optimizer. The only idea we have regarding this issue is that it may be possible that this problem is caused by us using the proper weight decay, while they use the L2-regularization instead. And although, by the logic of why we should be using it, the weight decay is more correct, it might theoretically happen that the L2-regularization helps the optimizer. Still, this seems improbable, since then not even the other optimizers would most likely function well. Or maybe, this could be caused by us using the natural gradient, as opposed to their use of the classical one. Regardless of the reason behind the non-functionality of the ADAM, this led to experimenting with various optimizers. We ran experiments with ADAM, SGD, and SGDM optimizers. Results in the form of population fitness of these runs can be found in Figure 5.3.

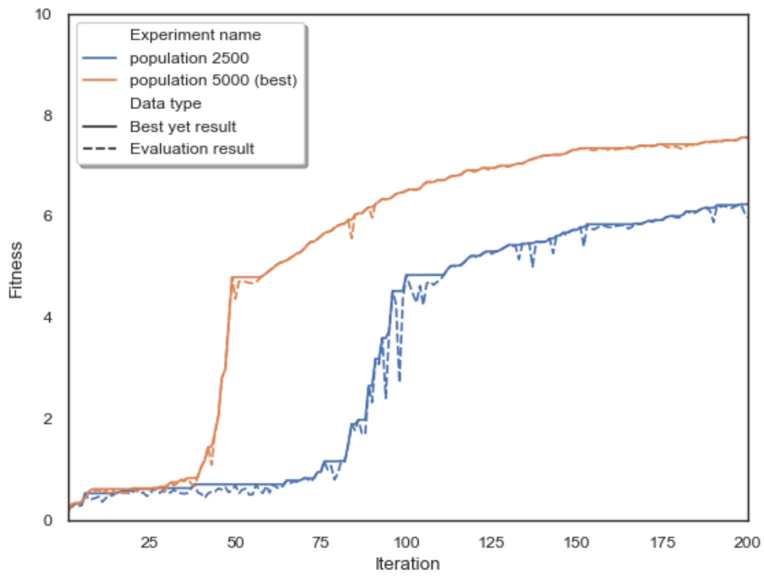
We can see that while ADAM is completely useless with our implementation, the other optimizers perform better. SGD shows better results than ADAM, but it is quite unstable. However, if we add a momentum to the SGD, thus obtaining SGDM, we can see that the training is much more stable and even much faster. Therefore, we chose SGDM as the default for all our experiments.

The last thing to be discussed in this subsection is the size of the population.

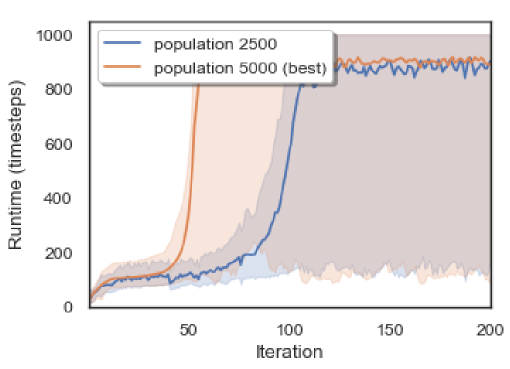
When we use a larger population, the iteration becomes longer as more episodes must be processed. But we also get a better estimation of the best direction in which to move in the parameter space, and the training algorithm requires fewer iterations to reach the same result. Nonetheless, this does not always have to be true. We do not have hard data to support this hypothesis, yet from some short experiments we have done it appears there might be such a thing as too large a population because from some population size, the performance begins to decrease again. We hypothesise this may be caused by having too many bad-performing agents and too few high-performing ones, therefore it is not clear enough what to try to move from and what to move towards in the parameter space. Another possible cause of this worsening of the performance for bigger populations is the manner in which the individuals of the population are created. They are sampled from the search distribution, yet in the practical implementation they are created by adding some noise to the mean of the population. The noises, however, are predefined in a shared table of noises, whence they are drawn. Thus, when we have too many individuals, their noises become highly dependent, which may, in theory, cause the observed problems.

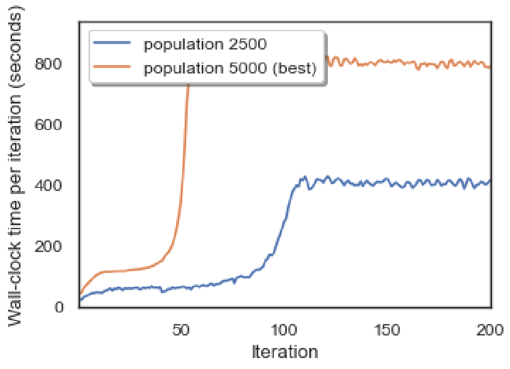
What we wanted to know is how the algorithm behaves with a smaller population than the one used in the original implementation, because when moving from FF model to DT, we get roughly five times more parameters. Hence, we need roughly five times larger population to get a good enough estimation of the best direction to move the parameters in. But this would mean a substantial increase in the time the algorithm would run. As a result, we wanted to gain an understanding of what happens when we decrease the population size. Therefore, we ran an experiment with half the population size compared to the default for the FF model and compared the result with the best of the experiments shown in Section 5.2.1. The results of the evaluation of the means of the search distributions of the runs and the time-related data about the individual iterations of the runs can be found in Figure 5.4.

In figures 5.4a and 5.4b we can observe that the run with the smaller population eventually reaches the point where the coarse gait is learned as well (with half the population size in twice as many iterations - roughly around the 100th iteration instead of around the 50th one), and then even the fine-tuning phase seems similar. So, with a smaller population, it will take longer to reach the same result. Still, as can be seen in Figure 5.4c, every iteration lasts only half as long with respect to the wall-clock time. This has little consequence for the initial phase of the training, yet it has a profound effect since the coarse gait is learned and the episodes become longer. This may lead us to propose a learning schedule, where initially the population is large, so the initial improvement is reached swiftly; yet as time passes, or when large enough initial improvement is reached, the population decreases in size, so that one iteration takes less time. Again, this applies only to those environments, where the episodes are shorter for random or not-so-well-performing agents and longer for better agents. Nevertheless, taking into account our observation that there can be such a thing as too large a population, and because with too little a population we do not have enough information to effectively estimate the gradient, this would probably require a lot of domain knowledge and experimentation with the hyperparameters setting before the training itself, and ergo it is not a viable training approach in real-life



(a) Evaluation results





(b) Runtimes

(c) Wall-clock times

Figure 5.4: Various population sizes
Figure 5.4a shows us the evaluation fitness after each iteration; Figure 5.4b contains a mean and 95% interval of the runtimes of the whole population each iteration; and Figure 5.4c shows us the wall-clock time of each iteration — all for runs with standard and half population sizes. For the standard population size, the best run observed in Section 5.2.1 has been chosen.

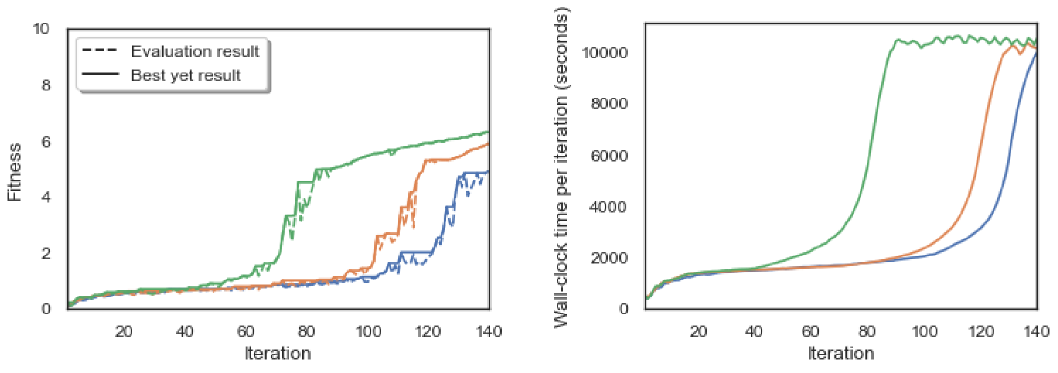
scenarios.

Still, we managed to show that smaller population might be enough for a successful training and so for the experiments with DT we use population size only twice the size used in experiments with the FF model.

5.2.3 Decision Transformer — Without a pretrained agent

Here, we will walk through the results of the experiments concerning the application of OpenAI-ES to training DT from scratch.

As can be observed in Figure 5.5a, the OpenAI-ES easily copes with even a more complex model such as DT, even though its population size is only double



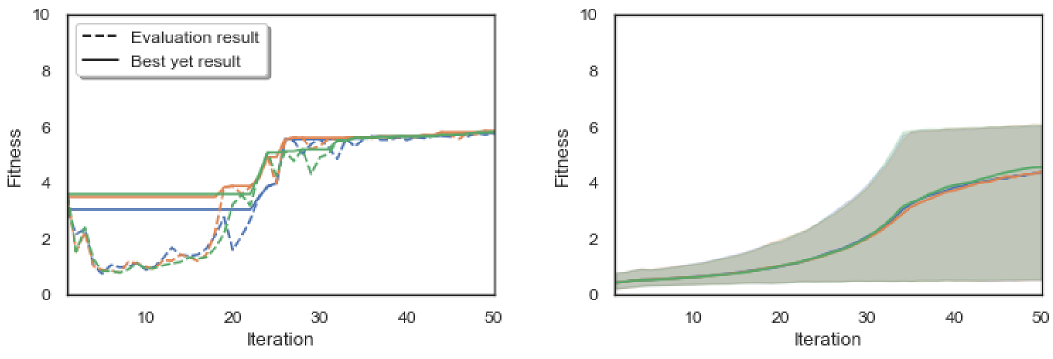
(a) Evaluation results (b) Wall-clock times

Figure 5.5: OpenAI-ES on a DT
Figure 5.5a shows us the evaluation fitness of the solution model after each iteration. Figure 5.5b shows us the wall-clock time each iteration needed to run.

compared to the population size in the case we trained the FF model, while it has approximately five times more parameters to set and a more complicated architecture overall. As we have seen in Section 5.2.2, if we had used larger population, we would have reached the leap in the evaluation fitness sooner than roughly around iteration 75 (for the best run) or 120 (for the others), which values can be seen in Figure 5.5a, nonetheless this would be at the cost of the higher wall-clock time duration of the training. And as we can see in Figure 5.5b, for the DT the duration of one iteration is significantly longer than for the FF model, because every model inference, every decision, which action to perform, takes longer. More profoundly so, when the coarse gait is learned and the episodes themselves become longer. Still, we have to keep in mind that utilizing twice as many CPUs would result in about half the wall-clock time required. And by every other increase in the computing power, the wall-clock time required would decrease accordingly. Yet, if there is a limit on the computing power available, as in our case, this shows us one limitation of using ES for the RL: When increasing the population size, we might reach a point where the episode duration combined with too large a population size makes learning from each timestep (by utilizing gradient TD methods) more favorable, compared to learning from the whole episodes (utilizing a population-based search, although parallelized).

5.2.4 Decision Transformer — With a pretrained agent

In this last subsection of the section dealing with OpenAI-ES, we focus on the ability of the aforementioned algorithm to further train or fine-tune a pretrained model. Generally, we would probably want to use the ES to train the model from scratch, so that the previously mentioned exploration, or let us call it the diversification, capability of the model is fully utilized, and because — as we commented previously — the ES is at its strongest when the episodes are short. Even so, as mentioned in Section 4.1, there might be problems when training large and complicated models by ES from scratch. Therefore, we want to try and utilize a pretrained agent for further training by the ES. The pretrained agent



(a) Evaluation results (b) Population fitnesses

Figure 5.6: OpenAI-ES on a pretrained DT
Figure 5.6a shows us the evaluation fitness of the solution model after each iteration. Figure 5.6b depicts a development of the fitness of the whole population derived from the solution model using noise, more specifically its mean and 95% interval, throughout the iterations.

used in the experiments has mean fitness around the value 4.

Figure 5.6a shows us that even though initially the performance of the trained agent decreases almost to the level of a random agent, it is later improved above the performance of the original agent. The initial worsening of the agent may be explained if we take a closer look at the early iterations in Figure 5.6b showing the fitnesses of the whole populations sampled from the search distribution. We can see that the fitness of the population is quite poor in the early stages. This shows us that although the agent was well-performing, it was not robust in the sense that adding a small noise to its weights would almost always completely break its performance. If we extend this idea, we can see that not only does the OpenAI-ES give us a high-performing agent, but the agent is quite robust as well, which might be a desirable property, e.g., when transferring the model to a system with lower precision where rounding of the weights might be expected, like when an agent is trained on a powerful computer but is deployed in a field on some mobile device.

If we come back to the fact that the agent had to be worsened in order to be improved again, an obvious question arises. Is there a benefit in seeding the algorithm by a pretreined agent compared to not doing so? This can be answered by comparing figures 5.6a and 5.5a. We can observe that the improvement comes much sooner when the pretrained agent is provided, around the 25th iteration, instead of roughly the 75th in the best observed case without the pretrained agent. So, this approach seems to be quite successful. Moreover, Figure 5.6b, nicely shows us that with the same pretrained model being passed to the algorithm as a starting point for further improvement, all three runs are almost the same despite not having the same seed, whence we can see that the training itself is quite robust.

Be as it may, this approach has its disadvantages as well. First of all, we need to have the pretrained agent. The next problem with this approach is that it does not allow the ES to produce diverse agents with different seeds. Yet, the biggest disadvantage is that it forces us to use different hyperparameter values than we would otherwise use. We have to decrease the value of the learning rate and the noise deviation so that the pretrained model is not completely randomized and broken in the first few steps, thus the further learning and fine-tuning will be slower. We also have to turn off the use of VBN, which can further harm future training efforts. Ergo, this approach is possibly viable only in the case mentioned in Section 4.1, when we have a model too big and too complicated for it to be easily capable to arrange its weights such that it does something interesting in the environment. Yet still, experimenting with this approach provides us with valuable insights.

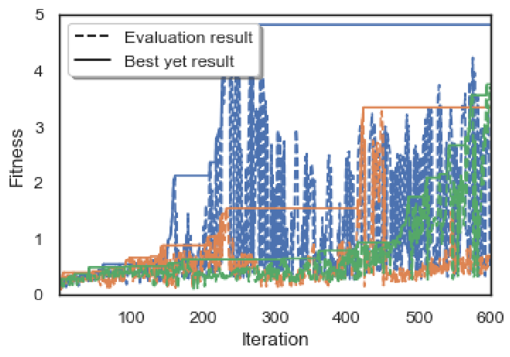
5.3 Novelty search

In this section, we experiment with our implementation of NS-ES. We begin by training the FF model, hence replicating the experiments from the original study; we then move on to experimenting with DT; and we conclude this section by taking a look at how well is the NS-ES able to exploit a pretrained model.

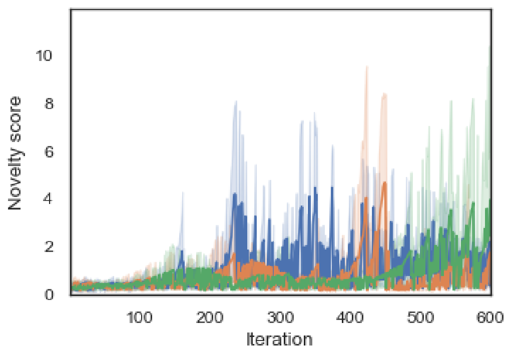
Before we dive into the experiments themselves, let us note one thing. The Humanoid environment is not a deceptive one. That means, in short, that the optimization of the fitness function is directly correlated with the optimization of the desired objective, that the fitness function does not lead us to a suboptimal local optimum, from which we cannot get by following the fitness alone. In the case of Humanoid environment, this means that fitness leads us to run further and further along the x-coordinate, the objective is to get as far in this direction as possible, and there is nothing stopping us from going in that direction. However, in the paper introducing among others NS-ES, the authors introduced a new deceptive environment, where they used Humanoid environment as a basis but added a small three-sided enclosure near ahead of agent's starting position. This then acted as a trap for the agents that tried to learn only how to go further along the x-coordinate. In this setting, the ES was unable to get around the trap, while the NS and QD algorithms were able to eventually circumvent the enclosure. Nevertheless, we are not attempting to show the capabilities of the given algorithms to solve a deceptive problem, but the ability to train the DT towards the goal of the NS, so whether the NS-ES will yield us agents moving in *some* direction, not just along the x-coordinate. Still, we have to keep in mind that when deploying an optimization algorithm, even a NS algorithm which does not use an objective in its process, we ultimately want to solve the original problem. Therefore, unlike in the aforementioned paper, we kept the reward to care only about the advancement made in the positive direction along the x-axis, while their reward is indifferent to the direction traveled. Thus, in the sense of the environment objective, fitness is what we are interested in, but in the sense of moving in any direction, the best observable and plottable indicator is probably the time the agent stayed on its feet (runtime), at least in the initial phase of the training.

5.3.1 Feed forward network

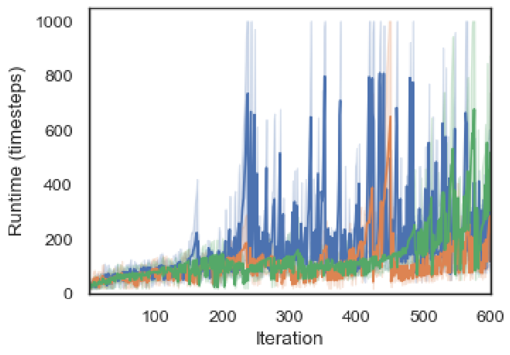
We begin by replicating the experiments by the original authors on the FF model with our implementation of NS-ES.



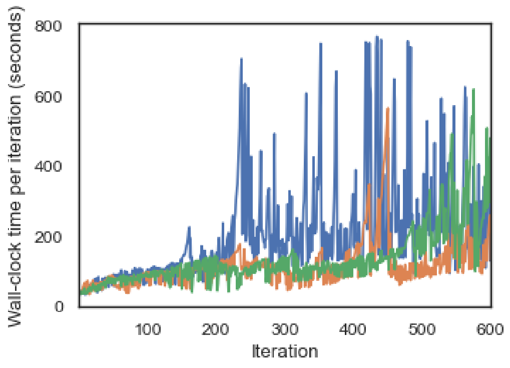
(a) Evaluation results



(b) Population novelty scores



(c) Runtimes



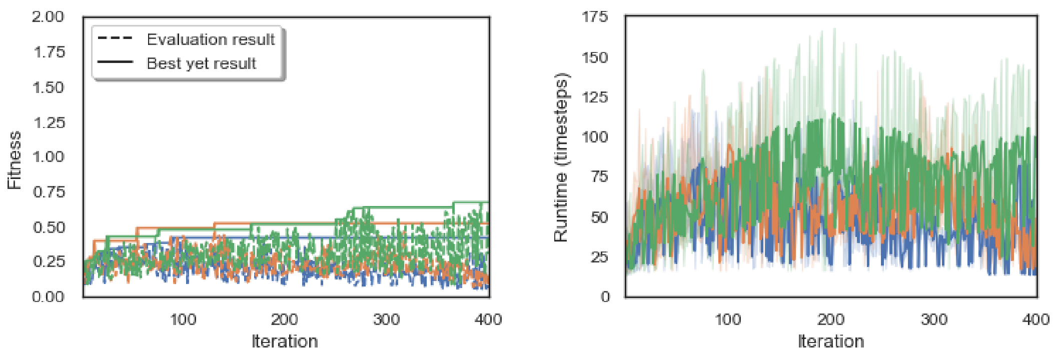
(d) Wall-clock times

Figure 5.7: NS-ES on a FF model
Figure 5.7a shows us the evaluation fitness of the solution model after each iteration. Figure 5.7b depicts a development of the novelty scores of the whole population derived from the solution model using noise, more specifically its mean and 95% interval, throughout the iterations. Figure 5.7c contains a mean and 95% interval of the runtimes of episodes performed by the agents of the population each iteration. And finally, Figure 5.7d shows us the wall-clock time each iteration needed to run.

As we can see in Figure 5.7a, we again obtained agents capable of moving in the environment, even though it took more iterations than when using OpenAI-ES, because of the meta-population, when we suddenly have not only one, but in our case five agents to train. By taking a look at the video recordings of the rollouts of the best agents in each run[2], we will find out that the best does not move straight forward along the x-axis, but walk sideways. Still, they are able to obtain relatively high returns. The jaggedness, or non-smoothness of the plot showing the evaluation fitness, as well as all the others from Figure 5.7 is caused by switching between the agents from the meta-population between iterations.

From Figure 5.7b, which shows us the mean novelty score of the whole population together with its percentile interval, we can observe an interesting thing: The surges in the novelty scores appear to correspond to the surges in the fitness of the evaluated agents. This is true, nevertheless we should realize there are novelty spikes even at iterations, that do not yield great results in the sense of

[2]Recordings can be found using a guide contained in the repository's ReadMe file.



(a) Evaluation results (b) Runtimes

Figure 5.8: NS-ES on a DT
Figure 5.8a shows us the evaluation fitness after each iteration and Figure 5.8b contains a mean and 95% interval of the runtimes of episodes of the whole population each iteration.

the fitness of the evaluated agent. They do, however, correspond to the spikes in runtimes in Figure 5.7c, which is the best indicator of how well does the corresponding agent walk for the initial phase of the training, as mentioned in the introduction of Section 5.3. This initial correspondence between higher novelty and longer-lasting agents can be explained by the fact that in this environment, once the agent starts walking instead of falling immediately, its fitness will substantially increase and simultaneously its novelty will increase. This is because all the previous agents ended near around the origin, so wherever the walking agent ends up, it is far away from the previous end positions.

As can be deduced from the jagged nature of the graphs even towards the end of the training, and as can be confirmed by the recordings of the rollouts of the final agents, the search does not train all the members of the meta-population equally to do something interesting. In fact, some remain quite clunky and capable of only short episodes up until the end of the training. For all the agents to become proficient in the environment, we would need to let the algorithm run longer. The reason why not all the agents from the meta-population are trained equally well is that initially once an agent learns to walk in any direction, its novelty raises, and because of the novelty-based probabilistic selection of agent to train further next iteration, it has higher chance of being chosen again and further improved, while the rest of the agents, which are still clumped around the origin, has smaller chance of being chosen for training, to learn to walk, and hence to escape from the origin.

5.3.2 Decision Transformer — Without a pretrained agent

In this subsection, we show and discuss the experiments with training DT using NS-ES algorithm.

We chose to use a smaller number of iterations (around $\frac{2}{3}$ of the number used for the FF model — 400 instead of 600), because of the time complexity of the training the bigger, more complex model utilizing larger population. In Section 5.2.3, we can see this proved to be no major problem for the OpenAI-ES (using the same fraction of the iterations provided to the FF model, $\frac{2}{3}$); however, for the NS-ES we see that much more training is needed to improve the more complicated model like DT. Nonetheless, because of the time constraints, we had to give up letting the algorithm run longer. Yet, in Figure 5.8a we can see gradual improving of the returns, and in 5.8b it is possible to observe that the mean runtime slowly grows as well compared to the initial runtime. Nevertheless, the improvement is negligible, and further work would be required to indisputably confirm that it is possible to train DT utilizing only a novelty signal. Yet, if we take a closer look at the rollout recordings of the final agents[3], we find that even though the agents mostly did not learn to walk in any direction, they still mostly learned to each fall in a different direction than the rest. And in a few rare instances, there even are some hints of walking, or let us rather call it directed movement. Ergo, the NS-ES seems to be at least partially successful.

5.3.3 Decision Transformer — With a pretrained agent

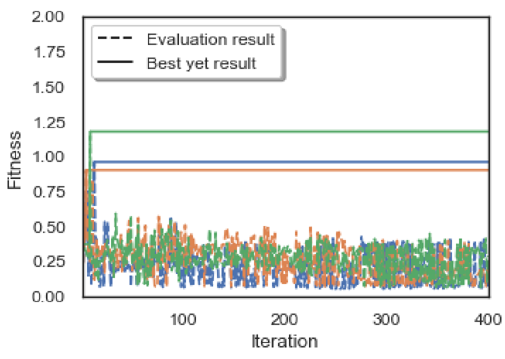
Here, we go through the results of experimenting with the use of a pretrained agent using a NS-ES algorithm for further training.

From Figure 5.9 we can observe that providing the NS-ES with a pretrained agent does not help the algorithm to make any progress. On the contrary, when compared to the runs without the pretrained agent discussed in Section 5.3.2, it shows complete lack of any progression being made. Even when visually inspecting the rollouts of the trained agents[4], we can see that most are completely random. Yes, in every run there is that one rare case of agent that seems like it might start doing something interesting just after the next training iteration, but it does not do so at the moment.

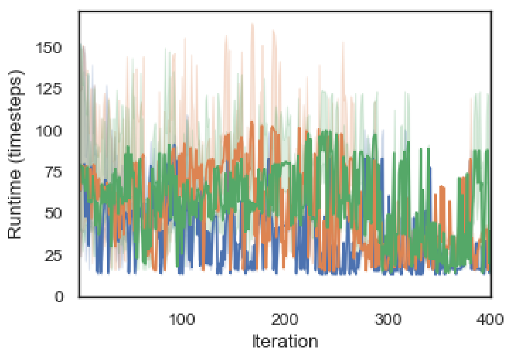
We can only guess what the reason for this non-functionality is. Our main hypothesis is that by seeding the pretrained agent to the whole population, as we did, we filled the behavior archive by five (meta-population size) behaviors of "going forward", essentially. Ergo, moving forward is not the preferred way and much more novel than that would be even just falling on the spot (which is the behavior of a random agent in this environment). Initially, random agent's behaviors are thus more desirable and trained towards, which breaks the pretrained agent. This phase can be seen in the initial part of Figure 5.9c. This would explain why the seeding by a pretrained agent does not help the search. Still, it does not explain the complete lack of progress throughout the runs of the algorithm. To explain this, we have to hypothesize even more: Even after the initial phase is over, after we end up with a random agent, moving forward is not as desirable as walking in any other direction. However, as we have seen in Section 5.2.4, where the pretrained agent was being used with OpenAI-ES, the pretrained agent is not robust and by adding a small noise we completely disrupt its function. Even so, in the aforementioned section we ended up with similar final agents with the same, only improved gait as the pretrained agent. That hints on the fact that although the pretrained agent is not robust, its gait might be something as a "attractor point" the agents in the vicinity of the pretrained

[3]Again, the recordings can be found in the respective data directory with logs and checkpoints. This can then be found using a guide contained in the repository's ReadMe file.

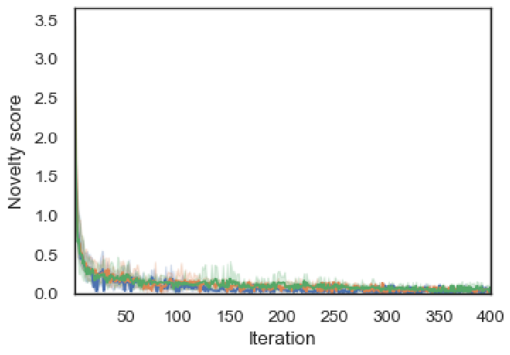
[4]This can again be found using a guide contained in the repository's ReadMe file.



(a) Evaluation results



(b) Runtimes



(c) Population novelty scores

Figure 5.9: NS-ES on a pretrained DT
Figure 5.9a shows us the evaluation fitness after each iteration. Figure 5.9b contains a mean and 95% interval of the runtimes of episodes performed by the agents of the population each iteration. Figure 5.9c depicts a development of the novelty scores of the whole population, more specifically its mean and 95% interval, throughout the iterations.

agent (in the parameter space) will incline towards. This leads us to believe that the algorithm must first really break away from the pretrained model by making it completely random in order to be able to begin to learn to reliably walk in other direction.

Still, these are just theories, and even though we are quite confident in our explanation of why the seeding by the pretrained agent does not improve the training, and could propose some solutions to this, like adding many dummy behaviors of random agents before the training, so that we prevent the quick degradation of the pretrained agent, we have much less confidence in the second explanation of why the training does seem to not work at all.

5.4 Quality-diversity

Here in the last section of this chapter, we will take a look at the QD algorithm NSR-ES and its ability to train DTs. As before, we begin by reevaluating its performance on FF model; then we proceed to the DT model without any pretraining; and we conclude by examining the performance when seeded with a pretrained agent.

Again, it is important to mention that we do not have a deceptive environment, so instead of the ability to overcome local optima, we investigate whether we can get a population of high-performing, yet diverse agents. Still, we should say that a different behavior characteristic would be suitable for this, we believe, because by utilizing the fitness, which is basically the distance traveled along the x-axis, we have a strong pressure on the direction of agent's movement, and it is our end objective. However, by utilizing the novelty based on final coordinates, we do not provide a means of diversifying this objective. It would be good for the deceptive problem, but for this it is not really that much suitable. Nonetheless, we kept this behavior characteristic to remain consistent with the previous experiments and the original paper. Despite that, in the initial phase of the training, when the progress made in any direction is not that substantial, we can see the diversification of the direction of the agent's movement, but still containing a positive component in the x-axis enforced by the fitness, as we will see in Section 5.4.2.

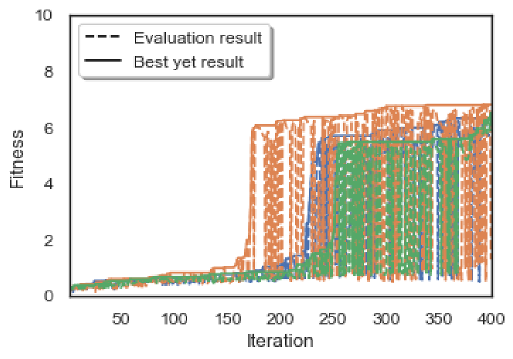
5.4.1 Feed forward network

This first subsection is again dedicated to the replication experiments of those done by the original authors of NSR-ES algorithm. We test the algorithm by once more training the FF model. As we can see in Figure 5.10a, we again obtain high-performing agents, even though it takes longer (with respect to the number of iterations needed) compared to OpenAI-ES algorithm and Figure 5.1a. This is the price to be paid for having a whole meta-population of agents to train instead of just one sole agent, as is the case in OpenAI-ES. It is also caused by the novelty being involved in assigning weights to the individuals of the population in each iteration, not just the fitness. Still, because in this case the behavior and objective are at least partially aligned (running further means both higher fitness and higher novelty score), the penalty for the added objective is not so grave.

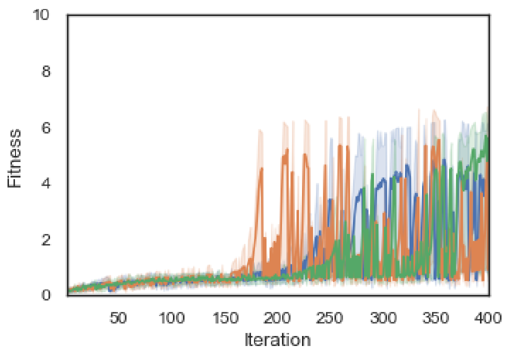
The confirmation of the alignment of the fitness and the novelty in this case can be observed in a correspondence that can be seen between figures 5.10b and 5.10c. There, we can see a partial correlation between the peaks of the population fitness and of the population novelty.

Just as in Section 5.3, where the NS-ES was tested, we can see the plots are quite jagged and non-smooth. Again, this is caused by having the meta-population and not every agent of the meta-population being trained equally. Even in the final meta-population, there are agents that fail to walk in any direction. Yet, even in these cases we can see the influence of the novelty in that they fall orderly each in a different direction to reach different final coordinates.

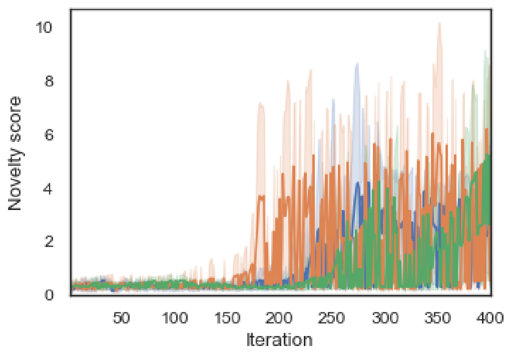
Still, in each of the three runs there are two agents from the meta-population, that managed to learn to walk. (In run number 3, the second walking agent still falls after a while, but it walks.) And even in each of these pairs, we can see the influence of including the novelty score for update, for the two agents of the pair almost never walk along the x-axis, not straight forward, but rather a bit diagonally, so their final position is as distinct from the other one as possible, but they still cover a lot of distance even in the x-axis direction. Even so, it might be quite interesting to try another behavior characteristic in the future work,



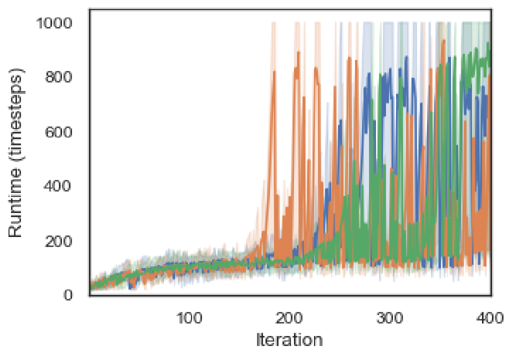
(a) Evaluation results



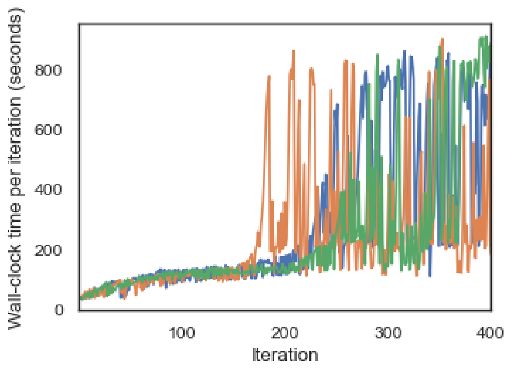
(b) Population fitness



(c) Population novelty scores

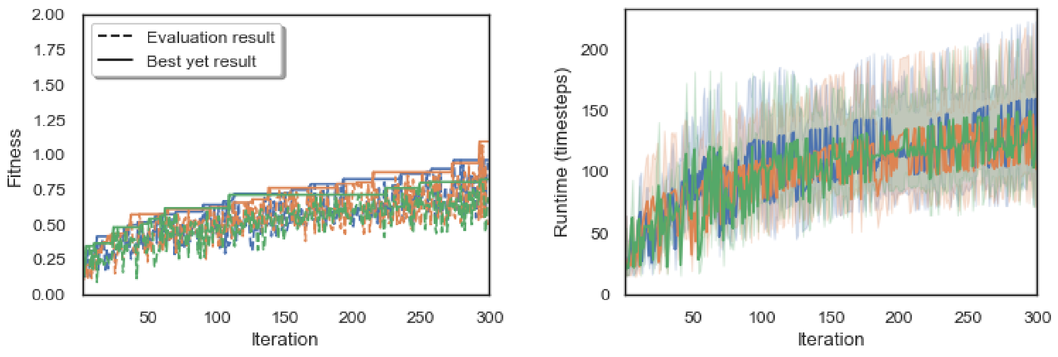


(d) Runtimes



(e) Wall-clock times

Figure 5.10: NSR-ES on a FF model
Figure 5.10a shows us the evaluation fitness of the solution model after each iteration. Figures 5.10b and 5.10c depict a development of the fitness and the novelty scores, respectively, of the whole population, more specifically their mean and 95% interval, throughout the iterations. Figure 5.10d contains a mean and 95% interval of the runtimes of episodes performed by the agents of the population each iteration. And finally, Figure 5.10e shows us the wall-clock time each iteration needed to run.



(a) Evaluation results (b) Runtimes

Figure 5.11: NSR-ES on a DT
Figure 5.11a shows us the evaluation fitness of the solution model after each iteration and Figure 5.11b contains a mean and 95% interval of the runtimes of the population each iteration.

for example, a characteristic on the sequences of actions performed — which is already prepared in the implementation — to see whether we will get agents going straight forward, but each with clearly distinctive gait. We hypothesize that this would only strengthen the innate ability of ES to produce diverse behaviors and thus may prove to be highly successful in achieving its goal of training meta-population of high-performing, yet diverse agents.

5.4.2 Decision Transformer — Without a pretrained agent

In this subsection, we cover the experiments concerning the effectivity of applying NSR-ES to train a DT.

We again, just as in Section 5.3.2 with NS-ES, reduced the number of timesteps compared to the case with the FF model for the same reasons as mentioned in the aforementioned section (300 as compared to 400). And just as in Section 5.3.2, this number of iterations proved to be lacking, and experiments lasting more iterations would be needed to fully and with no doubt confirm the effectiveness of utilizing NSR-ES to train the DT. Yet, unlike in Section 5.3.2, we can see in Figure 5.11a and Figure 5.11b that there are clear improvements in the meta-population in case of both the evaluation fitness and the population runtime, respectively. And when we take a look at the rollouts of the members of the final meta-population, we can see that, even though they all fall in the end, they fall each in a distinct direction, and we can even observe a slight movement of torso forward along an x-axis almost in every case which gives them some reward, even though they fall back after that. So we have even a visual confirmation that the algorithm relatively successfully trains them towards both the goals, to be distinct and to obtain a reward.

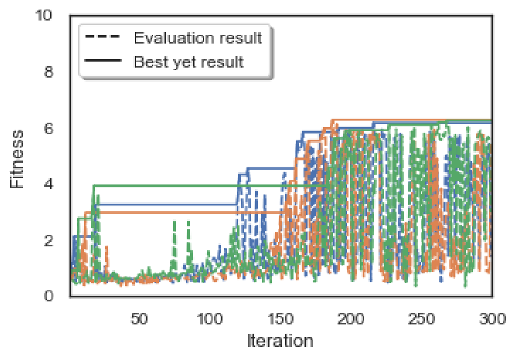
5.4.3 Decision Transformer — With a pretrained agent

In this last subsection of the final section of this chapter concerning experiments carried out, we will take a closer look at the ability of NSR-ES to exploit the pretrained DT to enhance the training.

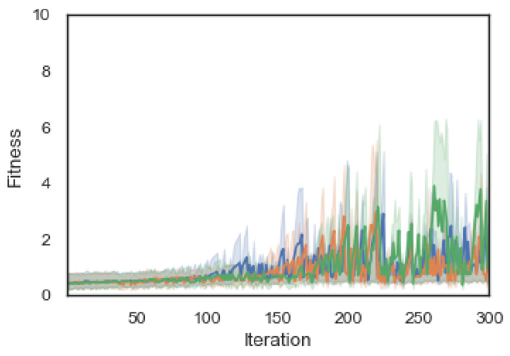
From Figure 5.12a, we can observe that — unlike with NS-ES in Section 5.3.3 — the NSR-ES can benefit from having a pretrained agent. And even though without the pretrained agent the number of iterations used was not enough as seen in Section 5.4.2, when seeded with the pretrained agent it proved to be capable of obtaining high-performing agents in the same number of iterations. Yet, the novelty-based part of the algorithm might seem not to work as perfectly, as the objective-based one, for the best agents are very similar in all three runs, as can be seen by taking a quick glance at their rollouts, and they are again, just like in Section 5.2.4, very similar to the pretrained model as well. They did not manage to alter the gait of the pretrained agent, they again just perfected it. However, if we take a closer look at the whole final meta-population, we can see that there really are different gaits in development; they only did not get the chance to get perfected more than the ”dominant” gait of the pretrained model. In data of run 1, there can be even observed a completely distinct gait, where the agent learned to walk sideways. In the other two runs, 2 and 3, on the other hand, there are multiple agents with gait similar to the ”dominant” one, but again, just as in the experiments with the FF model in Section 5.4.1, they walk not straight forward, but at least a bit diagonally, to end in another final position and hence gain higher novelty.

In Figure 5.12b, we can once more see — and hence confirm what we observed in Section 5.2.4 — that the pretrained agent was not robust and almost the whole population obtained by adding noise to the pretrained agent did not perform very well. In Figure 5.12c, in initial phase we can see similar dynamics to those observed in Section 5.3.3 in Figure 5.9c, which is a manifestation of the initial archive being filled by behaviors ending further away from the origin, but behaviors of later models ending near the origin. Nevertheless, unlike in Figure 5.9c, we have a completely different later phase, where we again learn to do something interesting and, therefore, start to obtain higher novelty score once more.

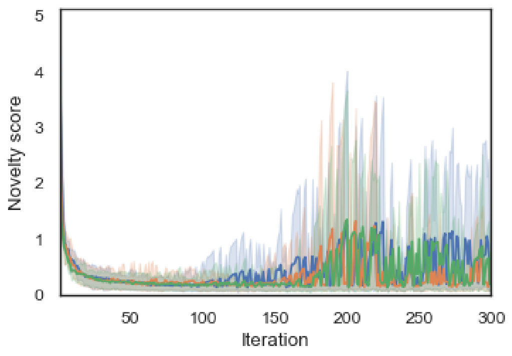
Ergo, we believe, we can confidently say that NSR-ES is fully capable of exploiting a pretrained model for further training. And once again just as in Section 5.4.1, it might be interesting to try to utilize other, e.g., action sequence-based behavior characteristic to obtain a meta-population of diverse high-performing agent obtained from a single pretrained model.



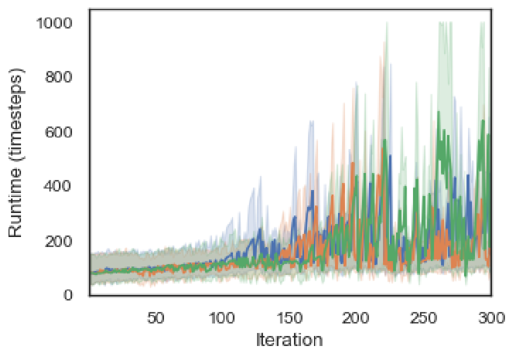
(a) Evaluation results



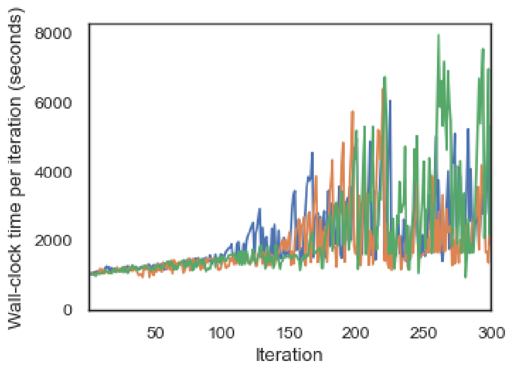
(b) Population fitness



(c) Population novelty scores



(d) Runtimes



(e) Wall-clock times

Figure 5.12: NSR-ES on a pretrained DT
Figure 5.12a shows us the evaluation fitness of the solution model after each iteration. Figures 5.12b and 5.12c depict a development of the fitness and the novelty scores, respectively, of the whole population, more specifically their mean and 95% interval, throughout the iterations. Figure 5.12d contains a mean and 95% interval of the runtimes of episodes performed by the agents of the population each iteration. And finally, Figure 5.12e shows us the wall-clock time each iteration needed to run.

6. Discussion

In this last chapter, we would like to conclude by recapitulation and further discussion of the results of this work and by providing possible directions and ideas for future work.

We managed to reimplement the algorithms and replicate the results of both the original papers about OpenAI-ES [15], and NS-ES and NSR-ES [26], respectively, as shown in Sections 5.2.1, 5.3.1, and 5.4.1. In our implementation, we focused on user-friendliness and versatility, so that the code could be easily used for experimenting with different models, environments, and behavior characteristics for the novelty-based searches, as described in more detail in Section 4.2. In the future, we plan to extend and upgrade our implementation, so that it is fully capable of working on a cluster, instead of just a single multi-processor machine. Still, with the current implementation we were unable to perform training in Atari games environment, for it took too long, almost one whole day for a single iteration, the reason being too large a population required, quite a big model, and a slower environment. And even though we were limited by the maximum number of cores on a single machine due to our implementation, and it would surely be better with more workers, there is always some upper limit on the computational power available, so there will always be this problem of utilizing ES to train large models. When one evaluation is too long and when many evaluations are needed to make one update step, the whole training will take too long.

We verified that the OpenAI-ES algorithm, as well as its variants NS-ES and NSR-ES performing NS, and QD search, respectively, are overall successful even in training larger and more complicated model, like the DT architecture, in the RL setting. In Section 5.2, we found out that OpenAI-ES had no problem training the DT, be it with or without a pretrained agent being seeded to the algorithm. Only a few problems arose for this ES. Once the agent learns to last longer in the environment and the episodes get longer, so does even the wall-clock time needed for one iteration. In case of a small FF model, this increase is not so perceptible, yet for the DT it is very significant. This, however, would not be such a big problem if we were able to utilize a whole cluster for the training, so it is not something insurmountable. The second problem is that when providing a pretrained agent to the algorithm, the training algorithm loses its innate ability to create distinct agents in multiple runs, it always just perfects the pretrained agent in the same way. Thus, seeding by a pretrained agent greatly reduces the iterations needed to obtain a high-performing agent, but we lose one of the biggest advantages of ES, its exploration, or rather diversification. And of course, we have to pretrain the model, but that may be much easier than training it from scratch using ES in some cases.

As for the NS-ES, as shown in Section 5.3, we found out that in order to successfully train DT it would need much more iterations than what we provided to train any of the agents from the meta-population to successfully walk in any direction. Still, we can observe traces of future gaits that are being learned, and we can see that the training for novelty has its effect on the final meta-population. However, in the case of providing the training with a pretrained agent, we observe remarkably bad performance. The fact that the NS is not capable of exploiting the pretrained agent to improve its search is explainable by many things, yet why no progress is visible at all even after many iterations remains largely a mystery, even though we have tried to propose an explanation for this as well.

Finally, we found the NSR-ES to be highly capable of successfully training DT, as documented in Section 5.4. Although the training from scratch would again require some more iterations, we were able to observe a clear progress. And when provided with the pretrained agent, the algorithm proved to be able to fully exploit it and train high-performing agents, even showing the capability to diversify the agents based on the behavior characteristic used.

Still, there are many possibilities for future research in this area. Aside from the ideas and suggestions proposed throughout Chapter 5, we lay out some others here. We could start by pointing out that our environment, just as environments used in the original paper for DT, was fully observable. It would be interesting to test whether having the transformer's context window, in other words, memory, helps us in partially observable tasks.

Another possible direction of the future research might be testing other EAs, to see whether they are able to deal with DTs and train this more complex architecture. For example, we could experiment with the GAs, mentioned at the beginning of Section 2.2, just as they were applied to training a FF model in the literature, be it as an objective-based search [37], or a novelty-based search [38].

We would like to conclude by pointing out the following. We managed to train the DT to perform well in the environment overall. Still, we did not manage to train the DT in its intended form, where it would respond to the desired return and the return-to-go tokens, in general. We could have removed the return-to-go tokens, thus saving a third of the space, and the result would probably not change much. Yet, the return-to-go tokens help DT to learn well in an offline manner, as stated in the original paper. And in particular, we believe it to be interesting to explore agents that we could tell how well we want them to perform and the agent would behave exactly as well or as badly as we want it to — all this using a single set of weights, a single agent. Therefore, we propose an approach that might achieve training and obtaining exactly such an agent which would manage to respond to the desired return and the return-to-go tokens. We would only modify the current algorithms, like OpenAI-ES, in the following manner. Each evaluation now consists of several subevaluations, each subevaluation gets a desired return sampled from a $\mathcal{N}(\mu, \sigma)$, a normal distribution centered at a value μ with variation σ. The value of μ would be computed from the returns obtained last iteration. The exact computation remains to be decided in a future work, but the resulting value should be somewhere between the best return obtained and the mean return obtained during the last iteration, so an improvement is gradually made. The variation σ would be a hyperparameter, or it could be a variation of the returns obtained last iteration. The fitness of an individual would not be the mean of its returns obtained, as is the case in OpenAI-ES, but rather a decreasing function of absolute values of the differences between the return obtained and the desired return in each subevaluation; therefore, the larger the differences, the smaller the fitness. This might be, e.g., minus a weighted sum of the differences, but again, the exact function used remains for a future investigation. Another approach could be to use a multi-objective ES, where one objective would be to maximize return obtained (based on subevaluations with the same desired returns for all the individuals, the desired return probably being a mean of the previous iteration returns plus some constant, or the best return observed during the last iteration), and the other objective would then be to maximize the fitness described above.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 2 edition, 2018. URL: `https://mitpress.mit.edu/9780262039246/reinforcement-learning/`.

[2] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. *Model-based Reinforcement Learning: A Survey.* Now Foundations and Trends, 2023. URL: `http://ieeexplore.ieee.org/document/10007800`.

[3] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7:133653–133667, 2019. `doi:10.1109/ACCESS.2019.2941229`.

[4] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992. `doi:10.1007/BF00992696`.

[5] Danijar Hafner, Timothy P. Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *ArXiv*, abs/1912.01603, 2019. URL: `https://api.semanticscholar.org/CorpusID:208547755`.

[6] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988. `doi:10.1007/BF00115009`.

[7] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL: `https://proceedings.mlr.press/v37/schulman15.html`.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, 07 2017. `doi:10.48550/arXiv.1707.06347`.

[9] K.F. Man, K.S. Tang, and S. Kwong. Genetic algorithms: concepts and applications. *IEEE Transactions on Industrial Electronics*, 43(5):519–534, 1996. `doi:10.1109/41.538609`.

[10] Kenneth A. De Jong. *Evolutionary Computation.* The MIT Press, 2016. URL: `https://mitpress.mit.edu/9780262529600/evolutionary-computation/`.

[11] Ingo Rechenberg. *Evolutionsstrategie — Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Friedrich Frommann Verlag, Stuttgart-Bad Cannstatt, Germany, 1973. URL: `https://api.semanticscholar.org/CorpusID:60975248`.

[12] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *arXiv*, 2016. `arXiv:1604.00772`.

[13] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(27):949–980, 2014. URL: `http://jmlr.org/papers/v15/wierstra14a.html`.

[14] Paolo Pagliuca, Nicola Milano, and Stefano Nolfi. Efficacy of modern neuro-evolutionary strategies for continuous control optimization. *Frontiers in Robotics and AI*, 7, 2020. URL: `https://www.frontiersin.org/articles/10.3389/frobt.2020.00098`, `doi:10.3389/frobt.2020.00098`.

[15] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv*, abs/1703.03864, 2017. URL: `https://api.semanticscholar.org/CorpusID:11410889`.

[16] Joel Lehman and Kenneth O. Stanley. *Novelty Search and the Problem with Objectives*, pages 37–56. Springer New York, New York, NY, 2011. `doi:10.1007/978-1-4614-1770-5_3`.

[17] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evol. Comput.*, 19(2):189–223, jun 2011. `doi:10.1162/EVCO_a_00025`.

[18] Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3, 2016. URL: `https://www.frontiersin.org/articles/10.3389/frobt.2016.00040`, `doi:10.3389/frobt.2016.00040`.

[19] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *ArXiv*, abs/2009.07485, 2020. URL: `https://arxiv.org/abs/2009.07485`, `doi:10.48550/arXiv.2009.07485`.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL: `https://openreview.net/forum?id=YicbFdNTTy`.

[22] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020. URL: `https://www.sciencedirect.com/science/article/pii/S0167278919305974`, `doi:10.1016/j.physd.2019.132306`.

[23] J Alammar. Transformer block (from web article The Illustrated Transformer). Digital Image, 2018. Slightly modified compared to the source image. URL: `http://jalammar.github.io/illustrated-transformer/`.

[24] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations*, 2013. URL: `https://api.semanticscholar.org/CorpusID:5959482`.

[25] R. Venkatesan and B. Li. *Convolutional Neural Networks in Visual Computing: A Concise Guide*. CRC Press, 1st edition, 2017. `doi:10.4324/9781315154282`.

[26] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 5032–5043, Red Hook, NY, USA, 2018. Curran Associates Inc. URL: `https://dl.acm.org/doi/10.5555/3327345.3327410`.

[27] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *arXiv preprint arXiv:2106.01345*, 2021. `doi:10.48550/arXiv.2106.01345`.

[28] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2234–2242, Red Hook, NY, USA, 2016. Curran Associates Inc. URL: `https://dl.acm.org/doi/10.5555/3157096.3157346`.

[29] John Geweke. Antithetic acceleration of monte carlo integration in bayesian inference. *Journal of Econometrics*, 38(1):73–89, 1988. URL: `https://www.sciencedirect.com/science/article/pii/0304407688900279`, `doi:10.1016/0304-4076(88)90027-9`.

[30] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. In *Advances in Neural Information Processing Systems*, volume 34, pages 1273–1286. Curran Associates, Inc., 2021. URL: `https://proceedings.neurips.cc/paper_files/paper/2021/file/099fe6b0b444c23836c4a5d07346082b-Paper.pdf`.

[31] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *ArXiv*, 2018. URL: `https://arxiv.org/abs/1801.01290`, `doi:10.48550/arXiv.1801.01290`.

[32] Florin Gogianu, Tudor Berariu, Lucian Bușoniu, and Elena Burceanu. Atari agents, 2022. URL: `https://github.com/floringogianu/atari-agents`.

[33] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017. URL: https://api.semanticscholar.org/CorpusID:53592270.

[34] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi:10.1109/IROS.2012.6386109.

[35] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. 06 2016. doi:10.48550/arXiv.1606.01540.

[36] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013. URL: http://dx.doi.org/10.1613/jair.3912, doi:10.1613/jair.3912.

[37] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *ArXiv*, abs/1712.06567, 2017. URL: https://api.semanticscholar.org/CorpusID:5044808.

[38] Ethan C. Jackson and Mark Daley. Novelty search for deep reinforcement learning policy network weights by action sequence edit metric distance. GECCO '19, page 173–174, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319619.3321956.

List of Figures

List of Abbreviations

- ADAM - Adaptive Moment Estimation optimizer
- CMA-ES - Covariance Matrix Adaptation ES
- DNN - Deep NN
- DT - Decision Transformer
- EA - Evolutionary Algorithm
- ES - Evolution Strategy
- FF - Feed Forward
- GA - Genetic Algorithm
- NES - Natural Evolution Strategies
- NLP - Natural Language Processing
- NN - Neural Network
- NS - Novelty Search
- QD - Quality-Diversity
- RL - Reinforcement Learning
- RNN - Recurrent NN
- SGD - Stochastic Gradient Descent
- SGDM - SGD with Momentum
- TD - Temporal Differences
- VBN - Virtual Batch Normalization