Universiteit Leiden

Scalable Evolutionary Strategies for Deep Reinforcement Learning via Integer-only Neural Networks using Low-rank Updates

Carlos Eduardo Moreira Mendes (s3647048)

Supervisors:
Felix Kleuker & Aske Plaat

Bachelor Thesis– Data Science and Artificial Intelligence

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 30, 2026

Abstract

We investigate whether a pure 8-bit integer policy network, trained without backpropagation via an evolutionary strategy, can match the performance of a state-of-the-art gradient-based approach, Proximal Policy Optimization (PPO), on reinforcement learning tasks. Building on the method EGGROLL [7], which generates low-rank perturbations over a population of candidates and computes parameter updates from their fitness signals, we implement *QEggRoll*, an integer-only variant in which every weight, activation, and update step is confined to the `int8` domain. We evaluate QEggRoll against Proximal Policy Optimisation (PPO) on four environments spanning discrete and continuous control, namely CartPole-v1, Pendulum-v1, MountainCar-v0 and MountainCarContinuous-v0, and complement the comparison with an 81-configuration ablation study on Pendulum-v1. QEggRoll is competitive on CartPole and MountainCar, demonstrates superior long-run stability on Pendulum where PPO suffered from learning collapse, but falls behind in the performance metrics on the continuous twin of MountainCar. The ablation reveals that population size and LoRA rank are the dominant hyperparameters, and that every configuration converges monotonically without divergence. Together, these results suggest that integer evolutionary strategies are a viable alternative to gradient-based reinforcement learning on moderate complexity tasks, with natural advantages in training stability and computational efficiency.

Contents

1 Introduction and Motivation

Evolutionary Strategies (ES) is a class of black-box optimization algorithms that stems from Evolutionary Algorithms [10], a research field that arose from early research in evolutionary computation during 1960s and 1970s, largely driven by work on biological inspired optimization [9]. These early studies established the idea of a population of candidate solutions, that could iteratively be improved by applying operations such as the mutation and recombination of its members, and then selecting the individuals that performed better, according to some given fitness metric, to populate the next generation of candidate solutions, eventually to achieve a desired fitness target.

In recent years, ES are better described as an attractive alternative to the abundant gradient based backpropagation methods that are used to train large language models and other types of neural networks. What makes this class of algorithms attractive is that they do not require the objective function to be differentiable, meaning that they are fit to optimize a broader class of models. Moreover, ES methods tend to perform better in noisy optimization landscapes, as the population-based search approach tolerates irregularities and discontinuities in the search space [7]. Lastly, and arguably one of its better traits, ES is highly scalable with parallel computing as demonstrated by Salimans et al. [6].

The practical barrier to ES at large scale is memory and compute since training a large neural network with standard ES would require storing a separate perturbation matrix for every population member, and applying those perturbations requires a full matrix multiply per member. In billion-parameter models, these costs quickly dominate and make large populations infeasible. Sarkar et al. addressed this with EGGROLL (Evolution Guided GeneRal Optimisation via Low-rank Learning) [7], an ES algorithm that generates low-rank perturbations, dramatically reducing both the storage and the per-member forward-pass cost. EGGROLL demonstrated that this low-rank approximation incurs no significant performance penalty because the aggregate update across a population of low-rank perturbations still remains high-rank. The authors also proved a fast $\mathcal{O}(1/r)$ convergence of the low-rank update to its full-rank counterpart.

Alongside their main results, they also released a 'nano' version of their model ('EGG' - Evolutionary Guided GRU), a pure integer non-linear RNN. They demonstrated this architecture in the context of language model pre-training, showing that a very large population (2^{20}) is able to stably train the language model [7]. However, the Reinforcement Learning (RL) experiments in the paper were performed with the float-point variant of the model. Whether a pure-integer network could also succeed in RL setting is a question left open.

This bachelor thesis directly investigates this question, namely, of whether a pure-integer network trained via evolutionary strategies could match the performance of PPO on RL tasks. We implement a pure 8-bit integer policy network be trained in a similar manner as the nano-egg. This model will be compared against a Proximal Policy Optimization (PPO) baseline. The model and perturbation scheme follows the methodology of Sarkar et al. [7] as closely as possible, including re-using some environments from the original test suite to draw conclusions about the algorithms performance.

This thesis is structured to guide the reader through the motivation, theoretical background needed, including preliminary concepts, full methodology overview, and evaluation of results: Section 2 provides the necessary background and related work; Section 3 explains core preliminary concepts for understanding of the model; Section 4 dives into the details of the implementation; Section 5 goes over the design and structure experiments; 6 presents the results of the said experiments and provides a discussion about them; Section 7 concludes the thesis.

2 Background

To frame our contribution, this section will prime the reader with some important background knowledge about the fields to which this work draws inspiration and expands upon. The object of the section is to survey a bit of the past of each of those field, leaving the important mathematical details to later sections, and highlighting a few points from which this paper may be of interest.

2.1 Deep Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning concerned with learning optimal behaviour through interaction with an environment. An agent observes a state s_t, selects an action a_t according to a policy π, receives a scalar reward r_t, and transitions to a new state s_{t+1}. The objective is to learn a policy that maximises the expected cumulative return, that is, the expected total sum of rewards $(r_1, r_2, r_3, ..., r_T)$ received along all states leading to a final state (s_T). This formulation, known as a Markov Decision Process, provides the theoretical foundation for most RL algorithms [8].

Early RL methods maintained explicit tables mapping every state-action pair to an estimated value, enabling exact computation but restricting applicability to small, discrete state spaces [9]. For problems with high-dimensional or continuous observations, such as pixel-based game environments or robotic joint configurations, this tabular approach became infeasible.

Deep RL emerges from the natural synergy that RL, with its need to handle large state spaces, has with Neural Networks (NNs), with their known representational capacity and ability to learn hierarchical structure. Therefore, instead of storing individual state-action values, a neural network parameterized by θ approximates the value function $Q_\theta(s, a)$ or the policy $\pi_\theta(a \mid s)$ directly from high-dimensional observations [9].

The transition introduced new challenges alongside its advantages. Experience collected during an episode is temporally correlated, which violates the independent-and-identically distributed assumption that stochastic gradient descent depends on. Moreover, the target values used to train the value function are themselves computed from the network being updated, creating non-stationarity landscape that can destabilize training. Although many of these issues have been addressed by DQN [5] and its variant architectures, they still face many challenges, such as problems with sparse rewards, long time horizons, and gradient calculation.

2.2 Evolution Strategies

Evolution Strategies are a family of black-box optimization algorithms that optimize an objective $F(\theta)$ without computing or requiring its gradient [10]. The core idea is to evaluate a population of perturbed candidate solutions (mutation), weight them by their relative performance (fitness), and move the parameter mean in the direction of better-performing members.

Besides the relaxed differentiability requirement, ES exhibits two properties that make it particularly attractive for large-scale RL: Firstly, all population evaluations are independent and can run in parallel with communication limited only to sharing scalar fitness values, enabling

near-linear scaling with the number of workers [6]; Secondly, because ES never backpropagates through the model, it is indifferent to the datatype chosen for the parameters, meaning the network can be stored and evaluated in any precision (int8, float32), and ES still produces a valid update signal. This second property is central to the pure-integer approach presented in this paper, as it would not easily be reconciled with other optimization approaches.

2.3 Fixed-Point Integer Arithmetic

Standard neural networks, near every modern digital systems, rely on floating-point arithmetic. In a floating-point representation, part of the stored value is devoted to an exponent that determines the scale of the number, allowing a more dynamic range at the cost of complex arithmetic for simple problems. With fixed-point numbers, contrary to the float-point dynamic range, every value shares the same global scale, and numbers are stored simply as integers. Hence, the final effect is that the location of the binary point is 'fixed' rather than exponent field.

Formally, let $p \geq 0$ denote the fractional width. A stored integer n (the internal hardware representation) is interpreted as the real quantity, $\tilde{n}$, as follows:

$$\tilde{n} = \frac{n}{2^p}. \tag{1}$$

Importantly, the factor 2^p is not stored alongside the number, otherwise both the internal representation and real quantity would be the same. Instead it is the governing parameter of the system that gives each represented value n its real counter-part $\tilde{n}$. For instance, in 4-fixed point representation (where $p = 4$), the integer 32 represents the real value 2 and the integer -16 represents the real value -1.

This interpretation makes addition and subtraction particularly simple. Since all quantities share the same scaling factor, ordinary integer addition corresponds directly to addition in the represented real domain:

$$\widetilde{a + b} = \tilde{a} + \tilde{b}. \tag{2}$$

Multiplication requires additional care. If two fixed-point values are multiplied,

$$\tilde{a} = \frac{a}{2^p}, \qquad \tilde{b} = \frac{b}{2^p}, \tag{3}$$

then

$$\tilde{a} \cdot \tilde{b} = \frac{a \cdot b}{2^{2p}}. \tag{4}$$

The integer product therefore contains an extra factor of 2^p relative to the original representation. To restore the intended scale, the result must be divided by 2^p, which is equivalent to a right shift of p bits:

$$\widetilde{ab} = \left\lfloor \frac{ab}{2^p} \right\rfloor = (a \cdot b) \gg p. \tag{5}$$

In practice, this division is always implemented by a right bit-shift by p positions. Conceptually, fixed-point multiplication can therefore be viewed as an ordinary integer multiplication followed by a rescaling step that restores the validity of the interpretation.

The same principle applies to matrix multiplication. Consider an input matrix x and weight matrix W, both represented in fixed-point form. Each entry of the product $xW^\top$ is a sum of fixed-point products and therefore accumulates the same excess scaling factor. The output is consequently computed as:

$$\mathrm{MatMul}(X, W) = \left\lfloor \frac{XW^\top}{2^p} \right\rfloor \tag{6}$$

The distinction between the internal integer value and its external real interpretation is critical when reading the code because a weight stored as 64 means 4. Keeping this dual reading in mind is essential for interpreting weight initializations, perturbation scales, and threshold values throughout the model.

3 Preliminaries

In this section, we will discuss some concepts that are central to understanding the model and its training structure. Each concept bellow is explained somewhat independently from each other and their connection to the bigger picture, although hinted at by each subsection, will be made clear by the next section 4.

3.1 Evolution Guided GeneRal Optimisation via Low-rank Learning

Despite its theoretical appeal, naive ES is impractical for large neural networks. Generating a perturbation $E \in \mathbb{R}^{m \times n}$ for each of N population members requires $O(Nmn)$ memory and $O(Nmn)$ additional compute per forward pass. At billion-parameter scale these costs quickly dominate, making large populations prohibitively expensive [7].

Sarkar et al. [7] addressed both bottlenecks with EGGROLL. By applying the Low-Rank Adaptation (LoRA) introduced by Hu et al. [2] for gradient-based large models fine-tuning, EGGROLL replaces the full-rank perturbation weight matrix with a rank-r approximation $E_i = \frac{1}{\sqrt{r}} B_i A_i$, where $B_i \in \mathbb{R}^{m \cdot r}$ and $A_i \in \mathbb{R}^{r \cdot n}$ with $r \ll \min(m, n)$. This reduces the memory from mn to $r(m + n)$ for each member in the population. Additionally, EGGROLL uses a counter-based deterministic random number generator to reconstruct perturbations on demand from a compact seed, so neither A_i nor B_i need to be stored between the forward pass and the parameter update, further lowering the memory overhead. This is efficient because the random noise matrices (B_i *and* A_i) generated for each member in the forward pass do not have to linger around in memory until the parameter update step. Instead, for large population sizes especially, memory can be freed between batches of members during the forward pass, allowing better use of working memory while maintaining access to their noise matrices via the compact seed.

The computational saving in the forward pass is similar in structure: Instead of adding the full perturbation matrix $\Delta W \in \mathbb{R}^{m \cdot n}$ to the base weight W_0 and performing $xW^\top$ for each member, EGGROLL instead computes the shared base activation $xW_0^\top$ once and saves it in memory, adding the low-rank correction xB_iA_i per member. This correction requires $O(r(m+n))$ operations instead of $O(mn)$ (the same reduction factor as in the memory). Importantly, despite each individual perturbation being rank r, the aggregate parameter update across all N members has rank up to $\min(Nr, m, n)$, so the expressiveness of the update is not inherently reduced for a sufficiently large choice of N and r. The authors proved formally [7] that the EGGROLL update converges to the full-rank ES update at an $\mathcal{O}(1/r)$ rate, justifying the use of small ranks even for exceedingly large models.

3.2 Evolutionary Strategies as Black-Box Gradient Estimation

As mentioned in 2.2, Evolutionary Strategies aim to optimize a scalar objective $F(\theta)$, such as the total reward accumulated over an episode, with respect to parameters θ, without requiring F to be differentiable with respect to θ. This is particularly attractive in reinforcement learning, where the reward signal often involves discrete events, long time horizons, or other structures that make

gradient propagation hard to reconcile.

In our case, F will be the expected cumulative return of an agent in a fixed reinforcement learning environment. Hence, F is given our agent (a matrix M of weights) and outputs its 'fitness', namely $F(M)$. The core idea here is to maximize the expected fitness under a perturbation distribution $p(\varepsilon)$:

$$J(M) = \mathbb{E}_{\varepsilon \sim p(\varepsilon)} \left[F(M + \sigma\varepsilon) \right] \tag{7}$$

Here, $J(M)$ represents the expected fitness of a matrix M which depends on the strength of the perturbation $\sigma > 0$ and our perturbation scheme, the distribution $p(\varepsilon)$.

Note that if $\varepsilon \sim \mathcal{N}(0, I)$, the perturbed matrix $Z = M + \sigma\varepsilon$ is itself normally distributed around M, i.e. $Z \sim \pi(Z \mid M) = \mathcal{N}(M, \sigma^2 I)$. This lets us rewrite the objective as an expectation with respect to this induced search distribution:

$$\begin{aligned} J(M) &= \mathbb{E}_{\varepsilon \sim p(\varepsilon)} \left[F(M + \sigma\varepsilon) \right] && (8) \\ &= \mathbb{E}_{Z \sim \pi(Z|M)} \left[F(Z) \right] && (9) \\ &= \int F(Z)\, \pi(Z \mid M)\, dZ. && (10) \end{aligned}$$

We can now differentiate with respect to M using the log-likelihood trick, exactly as in [11]:

$$\begin{aligned} \nabla_M J(M) &= \nabla_M \int F(Z)\ \pi(Z \mid M)\ dZ && (11) \\ &= \int F(Z)\ \nabla_M \pi(Z \mid M)\ dZ && (12) \\ &= \int F(Z)\ \frac{\nabla_M \pi(Z \mid M)}{\pi(Z \mid M)}\ \pi(Z \mid M)\ dZ && (13) \\ &= \int \left[F(Z)\ \nabla_M \log \pi(Z \mid M) \right]\ \pi(Z \mid M)\ dZ && (14) \\ &= \mathbb{E}_{Z \sim \pi(Z|M)} \left[F(Z)\ \nabla_M \log \pi(Z \mid M) \right] && (15) \end{aligned}$$

It remains to compute $\nabla_M \log \pi(Z \mid M)$ explicitly for our Gaussian search distribution. Since

$$\pi(Z \mid M) = \frac{1}{\left(\sigma\sqrt{2\pi}\right)^d}\ \exp\left(-\frac{\|Z - M\|^2}{2\sigma^2} \right), \tag{16}$$

its log-density is

$$\log \pi(Z \mid M) = -\frac{d}{2} \log(2\pi\sigma^2) - \frac{\|Z - M\|^2}{2\sigma^2}, \tag{17}$$

so that

$$\nabla_M \log \pi(Z \mid M) = \frac{Z - M}{\sigma^2}. \tag{18}$$

Substituting back $Z = M + \sigma\varepsilon$, we have $Z - M = \sigma\varepsilon$, hence

$$\nabla_M \log \pi(Z \mid M) = \frac{\sigma\varepsilon}{\sigma^2} = \frac{\varepsilon}{\sigma}. \tag{19}$$

Plugging this back into the expectation and returning to the ε-parametrization gives the following search gradient:

$$\nabla_M J(M) = \mathbb{E}_{\varepsilon \sim p(\varepsilon)} \left[F(M + \sigma\varepsilon) \cdot \frac{\varepsilon}{\sigma} \right] = \frac{1}{\sigma} \mathbb{E}_{\varepsilon \sim p(\varepsilon)} \left[F(M + \sigma\varepsilon) \cdot \varepsilon \right]. \tag{20}$$

We can see from the equation above that, in practical terms, the gradient direction is produced by a fitness-weighted average of perturbations. Meaning that perturbations that perform well, higher $F(M + \sigma\varepsilon)$, pull the gradient M toward their direction, ε.

In practice, the expectation is estimated by drawing N independent samples $\varepsilon_1, \ldots, \varepsilon_N \sim \mathcal{N}(0, I)$, evaluating the fitness $F(\theta + \sigma\varepsilon_i)$ for each sample, and calculating the Monte Carlo approximation:

$$\nabla_M J(M) \approx \frac{1}{N\sigma} \sum_{i=1}^{N} F(M + \sigma\varepsilon_i) \cdot \varepsilon_i. \tag{21}$$

The parameters are then updated by gradient ascent:

$$M \leftarrow M + \alpha \cdot \frac{1}{N\sigma} \sum_{i=1}^{N} F(M + \sigma\varepsilon_i) \cdot \varepsilon_i, \tag{22}$$

where α is the learning rate. This is similar to the formulation introduced by Salimans et al. [6] with deliberate adjustments to fit the relevant theory for the EGGROLL [7].

Lastly, it's important to highlight that although we are updating the weight matrix by gradient descent, it is still a black-box method, as the objective function F does not need to be differentiable. We only need F to be evaluable, that is, assign an output (deterministic or otherwise) to a given input. Moreover, this estimator in 21 is parallelizable, all N evaluations are independent from each other and can be distributed across workers, enabling it to be competitive in wall-clock time to other gradient based methods despite the lower sample efficiency [6].

The specific form of the update equation used in this work, which incorporates low-rank perturbations and integer arithmetic, is developed in Section 4.

3.3 LoRA (Low-Rank Adaptation)

In recent years, big technological corporations have been funded to train increasingly larger language models in the alleged search for general artificial intelligence, capable of solving a variety of tasks with human-level cognition. Needless to say, such endeavour requires endless resources (such as many data centers) and roughly every data point available on the internet or otherwise.

The dominant paradigm for deploying these large pre-trained neural networks is fine-tuning, which means starting from their 'out of the shelf' weights W_0 obtained through the pre-training on the broad dataset, and re-training/updating them on the task-specific objective. However, due to

the large scale of such models, full fine-tuning (updating weight matrices of the whole network) is both computationally and memory-intensive. With this in mind, Hu et al. (2022) proposed Low-Rank Adaptation (LoRA) [2] as a parameter-efficient alternative, motivated by the empirical observation that the weight changes required during task adaptation tend to have low intrinsic rank, meaning they are well-approximated by matrices of much lower rank than the original weight dimensions would suggest.

The LoRA proposition can be summarized as follows: Rather than learning a dense update $\Delta W \in \mathbb{R}^{m \cdot n}$ for each matrix $W_0 \in \mathbb{R}^{m \cdot n}$ in the model, a low-rank approximation can be made by decomposing each matrix as:

$$W_i = W_0 + \Delta W_i \approx W_0 + B_i A_i \tag{23}$$

Where W_0 a frozen pre-trained weight matrix and $B_i \in \mathbb{R}^{m \cdot r}$, $A_i \in \mathbb{R}^{r \cdot n}$ are thin column and row matrices ($r \ll min(m, n)$), reducing the number of trainable parameters of the full-rank update from $m \cdot n$ to $r(m + n)$ at a cost of the reduced degrees of freedom in the final update, since we are spanning a matrix in $\mathbb{R}^{m \cdot n}$ with a fraction of its original parameters.

We shall call this a 'LoRA factorization' for the remaining of this paper and its function will be central to the perturbation scheme of the model, when noise is applied to each population member.

3.4 Antithetic Sampling

Although the Monte Carlo gradient estimator presented in equation 21 is considered unbiased, meaning it will converges to the true gradient in expectation, it can still exhibit high variance, requiring large populations to produce stable update directions. Therefore, reducing the variance is important, specially when each fitness evaluation involves a full environment episode, which could be expensive depending on the environment.

One classical technique for reducing Monte Carlo variance is antithetic sampling, or mirrored sampling [1]. The key idea, as explained in [6], is to evaluate perturbation in opposite pairs, that is, for each noise matrix ε we produce a fitness for both ε and $-\varepsilon$. This introduces a negative correlation between these two pairs so that their errors partially cancel each other out. Hence, in our setting, instead of drawing N independent perturbations $\varepsilon_1, \ldots, \varepsilon_N$ as 3.2 would lead you to believe, we draw $N/2$ perturbations and pair them with the opposite direction:

$$(\varepsilon_1, -\varepsilon_1), (\varepsilon_2, -\varepsilon_2), \ldots, (\varepsilon_{N/2}, -\varepsilon_{N/2}) \tag{24}$$

The resulting gradient estimate for each pair i is:

$$g_i = \frac{F(M + \sigma\varepsilon_i) - F(M - \sigma\varepsilon_i)}{2} \cdot \varepsilon_i \tag{25}$$

and as such, the full estimate in equation 21 becomes:

$$\nabla_M J(M) = \frac{1}{N/2 \cdot \sigma} \sum_{i=1}^{N/2} g_i. \tag{26}$$

The variance reduction follows from the negative covariance between $F(M + \sigma\varepsilon_i)$ and $F(M - \sigma\varepsilon_i)$ because when ε_i points in a direction that increases fitness, $-\varepsilon_i$ tends to decrease it (or vice-versa), and their difference, $F(\theta + \sigma\varepsilon_i) - F(\theta - \sigma\varepsilon_i)$, better isolates the contribution of the direction ε_i than either evaluation alone. As shown by [6], this can substantially reduce the number of population members required for a reliable gradient estimate, effectively doubling the information content of each perturbation pair.

An additional practical benefit is that the antithetic pairing makes the gradient estimate invariant to additive constants in the fitness function, since any offset c added to both $F(\theta + \sigma\varepsilon_i)$ and $F(\theta - \sigma\varepsilon_i)$ would cancel in the difference, eliminating the need for a separate baseline term.

Another thing to point out, is that the $N/2$ pairs also provide a natural fitness signal in the form of the sign of $F(M + \sigma\varepsilon_i) - F(M - \sigma\varepsilon_i)$, which indicates which of the two directions was better, yielding a binary $-1, 0, +1$ fitness value per pair. This is central to the pure integer update rule which we will described in the following section.

4 Model

In this section we will assemble the preliminary concepts presented in section 3 into a complete explanation of the model. For that, we have divided the explanation into 3 subsections: The pure-integer policy network structure and how the forward pass takes observations of a given environment to produce logits in the correct action space; The low-rank integer perturbation scheme that produces each population of candidate solutions; and finally, the integer update rule that works as the gradient descent towards better fitness values. Throughout the section, we use $p = 4$ for the fixed-point width, so that q encodes the real value $\tilde{q} = q/2^p$, and every weight and activation is an `int8` bounded to the interval $[-127, 127]$ (as in 2.3). We define $\text{clip}(z) = \min(\max(z, -127), 127)$ and denote the full network parameters by θ.

4.1 Integer Policy Network

The policy $\pi_\theta(a \mid s)$ is a feed-forward network of L blocks operating entirely in `int8`. Since we cannot expect every environment to have observations that neatly fit $[-127, 127]$ (such as angle reading and other continuous observations spaces), we must first quantize an observation $o \in \mathbb{R}^d$ to a fixed-point representation $x^{(0)}$:

$$x^{(0)} = \text{clip}\left(\text{round}(c_{\text{obs}}\, o)\right) \in \mathbb{Z}^d \tag{27}$$

where c_{obs} is a per-environment input scale chosen so that typical observations occupy the representable range.

Linear layer: This part is similar to linear layers in other float-point networks. However, due to the `int8` fixed-point representation we need to take a few measures the avoid saturating the activations and keeping it to a proper scale. For a weight matrix $W \in \mathbb{Z}^{m\times n}$ and input $x \in \mathbb{Z}^n$, the layer accumulates the matrix product in a wider `int32` type and rescales once, following the fixed-point multiplication rule of section 2.3:

$$\text{Linear}_W(x) = \text{clip}\left(\left\lfloor \frac{xW^\top}{2^p\, \lfloor\sqrt{n}\rfloor} \right\rfloor\right). \tag{28}$$

Firstly, it's important to point out that the accumulation into `int32` is necessary because multiplying and adding multiple `int8` would exceed the representational range of `int8`. Performing the xW^T in a wider integer datatype avoids immediate overflows. As explained in 2.3, due to the multiplication of two fixed-point values, we need to divide them by 2^p in order to keep the representation. Additionally, the factor $\sqrt{n}$ acts as a fan-in normalization, analogous to the scaling used in Xavier initialization, preventing the variance of the activations from growing with the input dimensionality. Finally, we use clip() to cast the output back into `int8` territory.

It's important to note that, although we write $\lfloor\sqrt{n}\rfloor$, because we always choose our hidden dimensions to be powers of 4 $(4, 16, 64...)$, the result of this operation is always an integer and the compilation shortcuts the square-root call.

The weights are initialized by sampling from a standard normal distribution, scaling by 2^p, and rounding to the nearest integer:

$$W_{ij} = \text{round}(2^p \mathcal{N}(0, 1)) \tag{29}$$

This produces integer-valued weights whose corresponding real-valued representations approximately follows a standard normal distribution.

Layer normalization: To prevent activations from either saturating the limited `int8` range or collapsing towards zero as they propagate through successive layers, the network applies an integer-only normalization step after each linear transformation. Differently than the conventional layer normalization, which uses the mean and variance of the activations, this implementation uses only integer arithmetic and therefore avoids computing variances, square roots, or floating-point operations which would likely fall outside pure-integer restriction.

Let $\gamma \in \mathbb{Z}^h$ denote a learnable parameter for each hidden dimension, initialized as $\gamma_i = 2^p$ (corresponding to a real-valued gain $\tilde{\gamma}_i = 1$). Given an activation vector $x \in \mathbb{Z}^h$, the average absolute activation is first computed as:

$$\mu = \max\left(1, \left\lfloor \frac{1}{h} \sum_{j=1}^{h} |x_j| \right\rfloor\right) \tag{30}$$

Each channel is then rescaled according to the following layer norm:

$$\text{LayerNorm}(x)_i = \text{clip}\left(\left\lfloor \frac{\gamma_i x_i}{\mu} \right\rfloor\right) \tag{31}$$

This operation can be interpreted as an ℓ_1-based normalization, as the activation vector is divided by its mean absolute magnitude rather than its standard deviation. This is used to ensure that the activations remains approximately constant across layers, reducing the risk of numerical saturation between layers. The constraint $\mu \geq 1$ prevents division-by-zero when all activations are zero.

Although introduced primarily for numerical stability, this normalization also contributes to a degree of nonlinearity to the network, since the scaling factor μ depends on the current activation vector.

Forward pass: The network consists of an input projection layer, followed by L repeated normalization and linear blocks, and a final output layer. Given an input vector $x^{(0)}$, the hidden representation is first projected into the network's hidden space and then successively transformed by alternating layer normalization and linear operations:

$$h \leftarrow \text{Linear}_{\text{proj}}(x^{(0)}) \tag{32}$$

$$h \leftarrow \text{Linear}_{\ell}\Big(\text{ReLU}\big(\text{LayerNorm}_{\ell}(h)\big)\Big) \qquad \ell = 1, \ldots, L, \tag{33}$$

$$z \leftarrow \text{Linear}_{\text{head}}(h) \tag{34}$$

The output are integer logits $z \in \mathbb{Z}^{|\mathcal{A}|}$. After each normalization step the activations are passed through an integer ReLU, which zeros all negative values. This is the integer analogue of the pqn (post-quantization nonlinearity) activation used in the float EggRoll variant, which applies ReLU after layer normalization. Unlike smooth activations such as sigmoid or GELU, integer ReLU is a simple comparison-and-clamp with no floating-point cost. The non-linearity of the network thus comes entirely from the integer ReLU and the data-dependent normalization scaling μ, both fully integer operations.

Action selection: The final network output is a vector of integer logits $z \in \mathbb{Z}^{|\mathcal{A}|}$, with one component per action (for discrete control) or per action dimension (for continuous control environments). How these logits are turned into an action depends on the action space, and in both cases, if the policy is being evaluated deterministically or stochastically.

For discrete control tasks, the logits define a distribution over the available actions. A deterministic policy acts greedily, selecting the highest-scoring action by:

$$a = \arg\max_i z_i \tag{35}$$

while a stochastic policy, instead, samples $a \sim z$, treating the logits as un-normalized log-probabilities.

For continuous control tasks, the logits are first dequantized from their fixed-point representation and squashed element-wise via tanh, then optionally scaled by a per-environment action scale c to match the environment's action range:

$$\mu = c \cdot \tanh\left(z\, 2^{-p}\right) \in (-c,\, c)^{|\mathcal{A}|}, \tag{36}$$

here, the factor 2^{-p} reverses the fixed-point scaling, the tanh bounds each action dimension, and c maps the output to the environment's action range. The action is the mean itself when the policy is evaluated deterministically, $a = \mu$, or a sample from a Gaussian with that mean and a fixed standard deviation σ if stochastic:

$$a = \mu + \sigma\, \xi, \qquad \xi \sim \mathcal{N}(0, I) \tag{37}$$

Importantly, these conversions occur only between the policy and the environment and the dequantization, the tanh, and the optional Gaussian sample are the only floating-point operations involved. All other internal computation of the network remains entirely within the pure integer bounds.

4.2 Low-Rank Integer Perturbation

The population is generated by perturbing W with the low-rank scheme of EGGROLL 3.1. Each pair of population members $k = 1, \ldots, N/2$ is assigned a perturbation, and the two members of the pair receive it with opposite sign, as per the antithetic sampling of 3.4.

Matrix parameters: For a weight $W \in \mathbb{Z}^{m\times n}$, the perturbation is a rank-r integer matrix $E_k = B_k A_k$ with $B_k \in \mathbb{Z}^{m\times r}$, $A_k \in \mathbb{Z}^{r\times n}$ and $r \ll \min(m,n)$ (reusing the LoRA factorization from 3.3). The factor entries are drawn as `int8` samples from a single shared, counter-based pseudo-random source indexed by the pair k and the current generation, so that A_k and B_k are reconstructed on demand and never stored in memory, same as the implementation found in [7]. Following EGGROLL, the perturbed pre-activation never fully materializes $W + E_k$, the shared base $xW^\top$ is computed at the start stored and the rank-r perturbation is added per member:

$$\mathrm{Linear}_W^{(k)}(x) = \mathrm{clip}\left(\left\lfloor \frac{xW^\top + \left(xE_k^\top\right) \gg (p+s)}{2^p \lfloor\sqrt{n}\rfloor} \right\rfloor\right) \tag{38}$$

This reduces the cost from $O(mn)$ to $O\big(r(m+n)\big)$ per member. The arithmetic right-shift by $p+s$ bits scales the perturbation down to the desired magnitude, acting as the σ from Section 3.2. The parameter p is given by the fixed-point encoding chosen, while s is a hyperparameter that sets how large the noise is, with larger s giving smaller perturbations (throughout the experiments we use $s = 2$, similar as the value found in the nano-Egg). The antithetic pair of member k uses the negated correction $-E_k$.

Vector parameters: Parameters that are not matrices, namely the layer-norm gains γ, are perturbed element-wise with integer noise ε_k from the same shared source:

$$\theta^{(k)} = \mathrm{clip}\big(\theta + (\varepsilon_k \gg (p+s))\big). \tag{39}$$

4.3 Integer Update Rule

The continuous estimator of Section 3.2 (Eq. 22) are problematic in our pure-integer approach, because they assume it is possible to make small steps in the parameter space. Recall Eq. 22

$$\theta \leftarrow \theta + \alpha \cdot \frac{1}{N\sigma} \sum_{i=1}^{N} F(\theta + \sigma\varepsilon_i) \cdot \varepsilon_i,$$

Because every weight in the network is an `int8` value, we can only increase it or decrease it by integer values. However, the second term in the above equation can exhibit any value in the real numbers. Although in some cases it would be possible to round or adjust towards the nearest integer, depending on the learning rates it is possible that the weighted sum never exhibits large enough values. Hence, our methodology must adapt to deal with this impediment.

Antithetic fitness: As mentioned in Section 3.4, we can use the sign of the antithetic pairing to define a winner perturbation between the random ε and $-\varepsilon$:

$$s_k = \mathrm{sign}\big(F(\theta + \sigma E_k) - F(\theta - \sigma E_k)\big) \in \{-1, 0, +1\} \tag{40}$$

where F is the episodic return. The value 0 arises only on an exact tie, which in certain environments with sparse rewards would be somewhat frequent, if neither perturbation reaches a reward for example.

Accumulated direction: The fitness-weighted perturbations are summed in `int32` into a per-parameter direction estimate. For a matrix weight,

$$Z = \sum_{k=1}^{N/2} s_k \, B_k A_k \;\in\; \mathbb{Z}^{m\times n}, \tag{41}$$

which is the discrete analogue of the Monte Carlo estimator of equation (21). Here, each perturbation $E_k = B_k A_k$ is weighted by its pair's fitness s_k and summed. (For vector parameters the accumulation is element-wise, $Z_j = \sum_k s_k \, \varepsilon_{k,j}$.)

Thresholded unit step: Rather than scaling Z by a learning rate, each parameter is moved by a single least-significant bit in the direction of Z (± 1), but only where the accumulated evidence exceeds a threshold τ:

$$\theta \;\leftarrow\; \mathrm{clip}\big(\theta + \Delta\big), \qquad \Delta_{ij} = \begin{cases} \mathrm{sign}(Z_{ij}) & \text{if } |Z_{ij}| \geq \tau, \\ 0 & \text{otherwise.} \end{cases} \tag{42}$$

The threshold is not a fixed constant but scales with the size of the population,

$$\tau = c\,\sqrt{\tfrac{N}{2}} \tag{43}$$

where c is a small integer hyperparameter (we use $c = 2$ throughout). The $\sqrt{N/2}$ choice is to avoid that the parameters would fluctuate by just pure chance. Since there are $N/2$ antithetic pairs, by random chance we expect the signal to fluctuate on the order of $\sqrt{N/2}$.

This choice for the update bring two considerable effects: The first one and more readily noticeable, is that because the $\Delta \in \{-1, 0, 1\}$ the parameters may take a while to reach their optimal. Even if plenty mutations report a large fitness increase in a given direction, the update will only walk a single step towards it. Secondly, a parameter will only ever be updated if changing it brings significative evidence in the fitness-weighted sum, if a parameter is only marginally better than its alternatives this update rule will not walk towards it. This suppresses random-walking of the weights in noisy or near-tied fitness signals, and serves the same variance reducing role that averaging over the population plays in the continuous estimator 21.

5 Study Design

We evaluate QEggRoll against a PPO baseline across four reinforcement learning environments drawn from the EGGROLL benchmark suite [7]: the discrete control task `CartPole-v1`, the continuous control task `Pendulum-v1` (both from the gymnax library [3]), and the pair of discrete and continuous formulations of MountainCar-v0. These environments provide a good variety across action-space type (discrete and continuous), observation dimensionality, and task structure, and span a range of difficulty comparable to those typically encountered in compact RL benchmarks.

Additionally, we perform ablation studies on the hyperparameters of EGGROLL under the `Pendulum-v1` environment. Namely, the population sizes, LoRA rank, perturbation scale and number of fitness evaluations were tested in all 81 combinations of 3 chosen values for each parameter, with the goal of identifying which hyperparameters most affect performance and whether the method degrades harshly under non-optimal choices.

The question this thesis aims to address is of whether this pure-integer policy and evolutionary training scheme can match the performance of a state-of-the-art gradient-based approach baseline. Given the diverse choice for learning tasks, we aim to compare the techniques in primarily their final task performance (the returns in each of the environments) while taking into account the necessary computational resources (measured in sample efficiency and wall-clock time). Beyond the comparison with PPO, the ablation studies aim to give some insight on how the performance of QEggRoll will be affected under a different choice of model parameters and allow us to broaden the discussion about the method beyond its comparison with alternative approaches.

5.1 Experimental Protocol

QEggRoll and PPO: Both methods use the same network size, namely a 3 layer MLP with 256 hidden units per layer. The QEggRoll approach, as describe throughout section 4 works in `int8` in 4-fixed point convention, while the PPO approach uses the standard float32 network with ReLU activations and layer normalization via Rejax, a JAX native library [4]. The shared architecture size was deliberately chosen to ensure differences reflect the training algorithms and arithmetic choices rather than pure representation capacity.

All environments, CartPole Pendulum and the two MountainCar variants were each evaluated over 10 random seeds. Performance is measured as the mean episodic return over a fixed set of evaluation episodes, collected after the first training step and every other s training epochs (QEggRoll) or every s environment steps (PPO), with no gradient updates during evaluation (s is usually set to 20 hence we have data points for epochs 1, 20, 40, etc...). Due to how the two methods consume very different numbers of raw environment interactions, because QEggRoll's population evaluations far exceed PPO's on-policy rollouts on an absolute step count, the training curves are presented in three forms: total environment steps, measuring sample efficiency; wall-clock time, measuring computational cost; and normalised training progress scaled to $[0\%, 100\%]$, which aligns both methods from start to finish allowing direct comparison of their training curves.

Lastly, the original EGGROLL paper performed a hyperparameter optimization routine for

each environment before running the experiments on the best performing parameters. We shortcut this step here and re-use the reported best hyperparameter for each of the environments.

5.2 CartPole-v1

CartPole-v1 is a classical discrete-control task in which a cart must be driven left or right to keep a pole balanced upright. The observation is a four-dimensional vector of cart position, cart velocity, pole angle and pole angular velocity. The agents selects from two discrete actions (left or right). A reward of +1 is issued for every timestep the pole remains within $\pm$ 12° of the vertical direction and the cart stays within bounds (not too far off to each side). The episode ends when either constraint is violated or 500 steps are reached, giving a maximum return of 500.

Table 1: Hyperparameters for CartPole-v1.

Parameter	**QEggRoll**	**PPO**
Training budget	500 epochs	200,000,000 steps
Population / parallel envs	2048	256 environments
LoRA rank	4	—
Perturbation scale σ	0.25 ($\sigma_{\text{shift}} = 2$)	—
Evaluations per member/batch	1 episode	128 steps
Discount γ	—	0.995
GAE λ	—	0.9
Learning rate	—	3×10^{-4}
Gradient epochs per batch	—	4
PPO clip ε	—	0.2
Observation normalisation	32	enabled
Number of seeds	10	10

The perturbation scale $\sigma = 0.25$ is the value recommended in the original paper (Table 3 in appendix of [7]), and the rank $r = 4$ provides enough expressiveness to cover the four-dimensional action space efficiently.

5.3 Pendulum-v1

Pendulum-v1 is a continuous-control task in which the agent applies a torque to swing a pendulum to the upright position and keep it there. The observation is a three-dimensional vector $(\cos\theta, \sin\theta, \dot{\theta})$, and the agent action is a single real value in $[-2, 2]$. The reward penalizes the angle deviation from vertical, angular velocity, and torque magnitude simultaneously, so the optimal policy achieves a return close to 0. Note that all returns are negative and values closer to zero indicate better performance.

The smaller perturbation scale ($\sigma = 0.05$, $\sigma_{\text{shift}} = 4$) compared to CartPole reflects the finer-grained control required for the continuous action space: large perturbations would overshoot the narrow region of torques that successfully balance the pendulum.

Table 2: Hyperparameters for Pendulum-v1.

Parameter	**QEggRoll**	**PPO**
Training budget	500 epochs	400,000,000 steps
Population / parallel envs	4096	256 environments
LoRA rank	4	—
Perturbation scale σ	0.05 ($\sigma_{\text{shift}} = 4$)	—
Evaluations per member/batch	1 episode	256 steps
Discount γ	—	0.999
GAE λ	—	0.95
Learning rate	(integer steps, no LR)	3×10^{-4}
Gradient epochs per batch	—	16
PPO clip ε	—	0.1
Observation normalisation	16	enabled
Number of seeds	10	10

5.4 MountainCar-v0

MountainCar-v0 is a discrete control task that presents a hard challenge on the exploration side of the agent. Initially, the car rests in the bottom of a valley and accelerate left or right (or not at all) in order to reach a flag at the top of the hill. However, the car's engine is too weak to overcome the steepness of the valley, hence the car needs to learn to build momentum by swinging back and forth until it has enough speed to fully climb the hill. The observation is a two-dimensional vector of position $x \in [-1.2, 0.6]$ and velocity $v \in [-0.07, 0.07]$. The agent selects from three discrete actions (push left, no push, push right). A penalty of -1 is incurred each timestep, and the episode terminates when the goal position $x \geq 0.45$ is reached or 200 steps elapse, giving a theoretical return range of $[-200, -1]$, however, as described previously, it is not possible to reach the flag immediately.

Table 3: Hyperparameters for MountainCar-v0.

Parameter	**QEggRoll**	**PPO**
Training budget	500 epochs	200,000,000 steps
Population / parallel envs	2048	256 environments
LoRA rank	4	—
Perturbation scale σ	0.5 ($\sigma_{\text{shift}} = 1$)	—
Evaluations per member/batch	1 episode	200 steps
Discount γ	—	0.999
GAE λ	—	0.95
Learning rate	(integer steps, no LR)	1×10^{-3}
Gradient epochs per batch	—	4
PPO clip ε	—	0.2
Observation normalisation	[200, 500]	enabled (normalised)
Number of seeds	10	10

The large perturbation scale ($\sigma = 0.5$, $\sigma_{\text{shift}} = 1$) promotes broad exploration in this sparse-reward

setting.

5.5 MountainCarContinuous-v0

MountainCarContinuous-v0 shares the same physical setting as the discrete variant, namely, a car in a valley that must reach the flag at the top of the hill. The observation remains two-dimensional with position and velocity vectors. The action is now a single continuous force $a \in [-1, 1]$ rather than a discrete push. However, differently than the discrete versions, the reward function by a penalty to the used force of $-0.1\,a^2$ at each step and a bonus of $+100$ upon reaching the flag, which ends the episode, giving a theoretical return range of $[-100, +100]$ in the 1000 steps limit.

Table 4: Hyperparameters for MountainCarContinuous-v0.

Parameter	**QEggRoll**	**PPO**
Training budget	200 epochs	400,000,000 steps
Population / parallel envs	2048	256 environments
LoRA rank	4	—
Perturbation scale σ	0.25 ($\sigma_{\text{shift}} = 2$)	—
Evaluations per member/batch	1 episode	256 steps
Discount γ	—	0.999
GAE λ	—	0.95
Learning rate	(integer steps, no LR)	3×10^{-4}
Gradient epochs per batch	—	4
PPO clip ε	—	0.2
Observation scale	[200, 500] per dim	enabled (normalised)
Number of seeds	10	10

A smaller perturbation scale ($\sigma = 0.25$, $\sigma_{\text{shift}} = 2$) is used compared to the discrete variant, reflecting that the shaped reward already provides a continuous gradient signal that distinguishes members that reach the flag from members that don't, without requiring large perturbations.

5.6 Ablation Studies

The ablations studies where performed on the Pendulum environment due to its higher difficulty as a reinforcement learning task. Originally CartPole was attempted but quickly it was noticed that all configurations achieved perfect or near perfect scores, learning very little for a satisfactory discussion to be drawn out of it.

Each configuration is evaluated for 2000 epochs with a single seed. The four varied hyperparameters and their iterated values are:

- Population size: $N \in \{256, 512, 1024\}$,
- LoRA rank: $r \in \{1, 2, 4\}$,
- Perturbation scale: $\sigma_{\text{shift}} \in \{1, 2, 4\}$, corresponding to effective perturbation magnitudes $\sigma \approx 0.5, 0.25, 0.05$,

- Fitness evaluations per member: $k \in \{1, 3, 4\}$.

These combine for a total of $3^4 = 81$ configurations. To reduce other sources of variation that could favor a configuration over another, every configuration re-uses the same random seed for their runs. All experiments were executed on a single NVIDIA RTX 4060 Ti (8 GB VRAM). The original paper's recommended population sizes which far exceed the memory budget of this GPU for most environments, so population sizes were reduced accordingly.

6 Results and Discussion

The QEggRoll step count was estimated conservatively as *population size* $\times$ *steps per epoch*, which potentially overshoots in environments where an early exit is possible (such as letting the Pole fall in Cartpole-v1), meaning not all steps for that epoch were used, but still accounted for. For PPO, the step count is computed exactly from the training configuration as `num_envs` $\times$ total rollout steps.

6.1 Results for CartPole-v1

CartPole-v1 is a discrete-control task in which an agent is rewarded for keeping a cart within the track boundaries while balancing an attached pole upright for as long as 500 steps, this way achieving maximum reward. Recall the hyperparameters and broader environment explanations in 5.2.

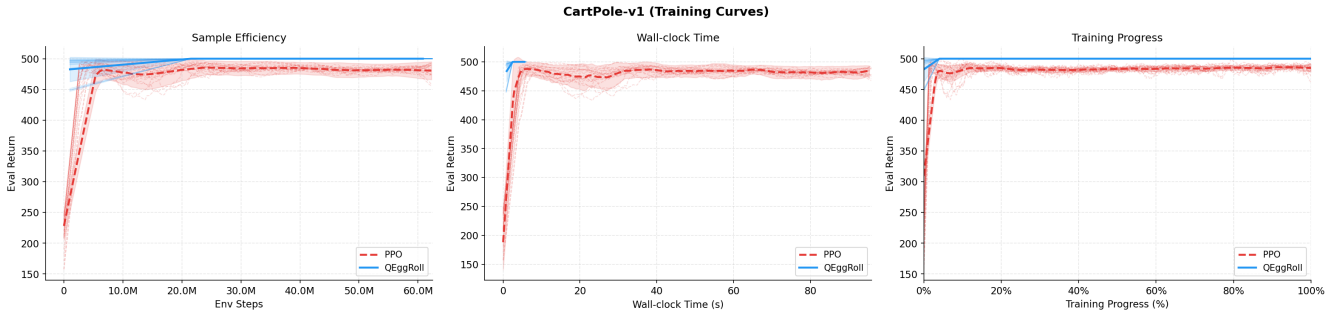

Figure 1: Training curves QEggRoll and PPO under CartPole environment. Each panel shows the mean evaluation return (thick line) and one standard deviation (shaded band) over 10 seeds for QEggRoll (in blue, population size $N = 2048$, 80 epochs) and PPO (in red, 256 parallel environments, 62M steps). PPO's curve was uniformly smoothed by a moving windows of 15 log checkpoints. Individual seed runs appear as thin lines of the same color. The episode return ranges from 0 (pole falls immediately) to 500 (maximum episode length). In all 3 plots the data points are the same, varying only on the x-axis variable: Left plot shows return over estimated environment steps; Center plot shows return over wall-clock time in seconds; Right plot shows return versus normalised training progress (0% = start, 100% = end of respective training budget) for ease of comparison between the curves.

The first thing that becomes very evident after looking at the graphs shortly is that there is a shocking asymmetry in the variance between seeds on both methods: QEggRoll achieves a mean return of around 475 in the first evaluation and afterwards sustains the maximum return of 500 throughout the remaining epochs, with no individual trace performing bellow that result; PPO on the other hand, begins with near zero returns (which we can see from its rise close to the origin) and improves to returns around 500. Still we can see from the variance bands and trace lines that some seeds lag behind with non-optimal returns.

Secondly, the wall-clock time view shows the computational efficiency of the method: QEggRoll finishes 80 epochs with a population size of 2048 consistently in around 5 seconds, whereas PPO

takes around 100 seconds for the same amount of environmental steps, an difference of about 20x in speed.

Regarding sample efficiency, PPO reaches high scores faster than QEggRoll but it seems that the EggRoll convergence is cleaner. Still, this task is clearly too simple to fully evaluate both methods, given that QEggRoll achieves perfect scores after the second evaluation at 20M steps.

6.2 Results for Pendulum-v1

Pendulum-v1 is a continuous-control reinforcement learning environment in which an agent is rewarded to swing up and balance a pendulum in its upright position by applying continuous torque. The reward function is maximized by keeping the pendulum upright with minimal magnitude and angular velocity of the applied torque. Recall the hyperparameters and explanations in 5.3.

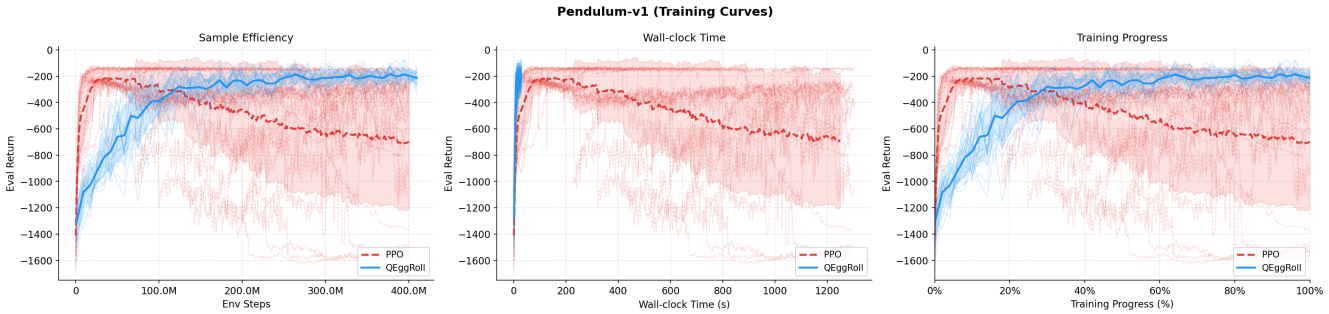

Figure 2: Training curves QEggRoll and PPO under Pendulum environment. Each panel shows the mean evaluation return (thick line) and one standard deviation (shaded band) over 10 seeds for QEggRoll (in blue, population size $N = 4096$, 500 epochs) and PPO (in red, 256 parallel environments, 400M steps). Individual seed runs appear as thin lines of the same color. The episode return ranges from approximately -1600 (near-random policy) to 0 (perfect upright balance); all returns are negative and values closer to zero indicate better performance. In all 3 plots the data points are the same, varying only on the x-axis variable: Left plot shows return over estimated environment steps; Center plot shows return over wall-clock time in seconds; Right plot shows return versus normalised training progress (0% = start, 100% = end of respective training budget) for ease of comparison between the curves.

The pendulum results are the most insightful from all the environments, as both methods draw clear movements in their performance over the epochs.

PPO can be seen to improve very rapidly, taking roughly 20-30M steps to reach returns above -200, which is a high performance for the task. Although the mean performance trends downwards, we can see that this is mostly due to 5 seeds whose traces fall bellow the variance performance by the end of the training, and generally it can be seen that plenty seeds manage to keep their performance above -200, surpassing the mean peaks for QEggRoll. Contrary to this, QEggRoll takes a long amount of environmental steps to reliably get to the -200 return mark, with a mean performance close to that performance only after about 200M steps. To its benefit, the method shows no seed with performance bellow of -400 around the same mark and the convergence is

cleaner with a smaller variance bound over the epochs.

Although PPO presents higher sample efficiency, the Wall-clock time view shows that all the 400M steps by QEggRoll are finished in less than 50 seconds, whereas PPO would only reach 50M environment steps (where its mean performance peaks) after more than 150 seconds.

In summary, despite PPO having better sample efficiency, under the same implementation and hardware used here, it would be a lot faster to employ QEggRoll to solve the environment.

6.3 Results for MountainCar-v0

MountainCar-v0 is an exploration discrete control task in which the agent must oscillate the car to build momentum before reaching the flag goal on the top of its valley, receiving -1 every step with no intermediate reward signal. Recall the hyperparameters and broader environment description on section 5.4.

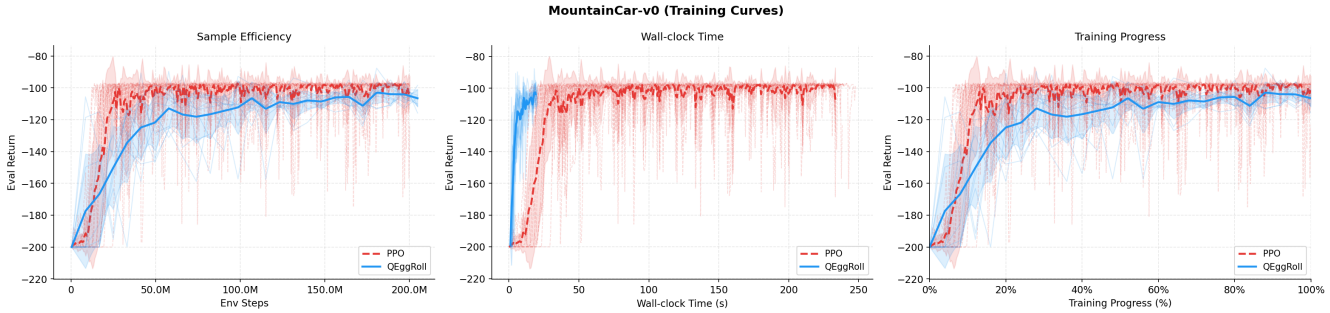

Figure 3: Learning curves for QEggRoll and PPO on MountainCar-v0. Each panel shows the mean evaluation return (thick line) and one standard deviation (shaded band) over 10 seeds for QEggRoll (in blue, population size $N = 2048$, 500 epochs) and PPO (in red, 256 parallel environments, 200M steps). Individual seed traces appear as thin translucent lines. The episode return ranges from -200 (goal never reached within 200 steps) to -1 (goal reached on the first step, physically impossible from the start position). In all three panels the underlying data are the same, varying only the x-axis: left shows estimated environment steps; centre shows wall-clock time in seconds; right shows normalised training progress (0% = start, 100% = end of respective budget).

Here, we observe similar behaviour as analysed in the Pendulum environment, where QEggRoll lags behind in sample efficiency, needing to take more steps before reaching equal performance while strongly outperforming in wall-clock time, to the extent that it finishes the allotted training steps in 30 seconds, compared to the near 4 minutes needed for PPO.

Differently from the previous task, we do not observe a collapse from the PPO agent, which is likely due to the structure of the tasks, where the inverted pendulum tends to chaotic behaviour while MountainCar reserves more structure and a less noisy gradient for the actions.

6.4 Results for MountainCarContinuous-v0

MountainCarContinuous-v0 is the continuous-action variant of the same physical task, with a reward function ($-0.1\,a^2$ per step and $+100$ on goal) that provides a denser gradient signal than the discrete variant. Recall the hyperparameters and broader environment description in Section 5.5.

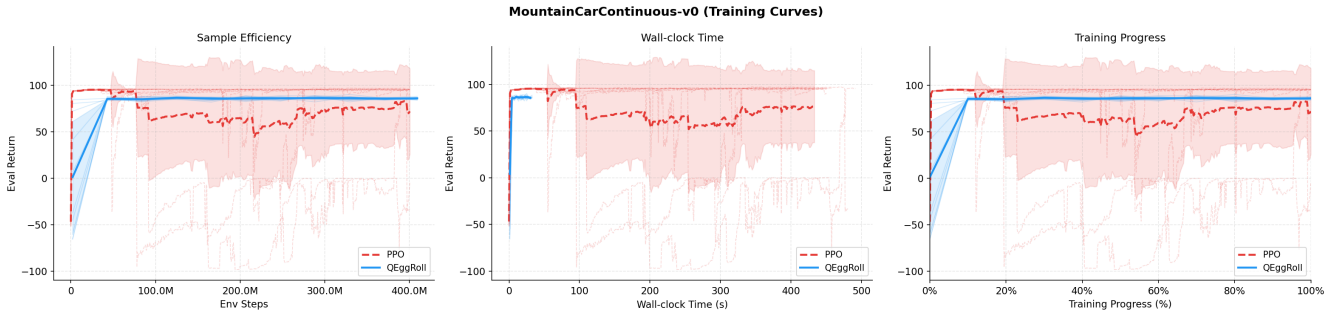

Figure 4: Training curves for QEggRoll and PPO on MountainCarContinuous-v0. Each panel shows the mean evaluation return (thick line) and one standard deviation (shaded band) over 10 seeds for QEggRoll (in blue, population size $N = 2048$, 200 epochs) and PPO (in red, 256 parallel environments, 400M steps). Individual seed traces appear as thin translucent lines. The episode return ranges from approximately -100 (full force applied throughout, goal never reached) to $+95$ (goal reached quickly with minimal effort). In all three panels the underlying data are the same, varying only the x-axis: left shows estimated environment steps; centre shows wall-clock time in seconds; right shows normalised training progress (0% = start, 100% = end of respective budget).

In comparison to previous environments, the behaviour of QEggRoll seems much closer in comparison to that displayed in CartPole than to its training curves in the discrete twin environment MountainCar. Although we could expect to observe similar training curves as in section 5.4 due to the identical setting of the task (the same valley/flag on the hill setup), the truth is that the change in the reward signal is of extreme importance to the model. In the discrete task, if no car manages to reach the flag in the population from their random mutations, there is no fitness signal whatsoever to be followed, as all members accrued the exact same -200 (-1 per step over 200 maximum steps) reward, and as such no learning happens. In the continuous scenario, because the agents are penalized for the force they exert on the car, a member that aimlessly flails the car around will have a lower fitness than that of a car that barely uses its actions. On top of that, the reward for reaching the goal is extremly rewarding compared to the previous environment. In the discrete case a car finishing before the last step would get -199 return (skipping the last -1 penalty), which is not a high improvement from -200. In this continuous scenario, members are awarded a +100 return for reaching the flag, meaning that even if they misused the action with maximal penalty for force used, the minimum achievable reward would only be 0.

These two factors come together to give a more granular gradient signal for the agent. It becomes not only about learning from members that reached the goal, but also from other agents that did not missued their actions. This way, the environment points agents to start exploring with the hills and building momentum from those even before the goal is reached by any member.

Lastly, although previous examples always painted the picture that PPO has higher sample efficiency but would struggle to reach the answer in the same wall-clock time, here QEggRoll lags behind in both departments as PPO near instantly solves the environment.

6.5 Results for the ablation studies

The ablation results are presented as one figure per LoRA rank value ($r \in \{1, 2, 4\}$), each containing a 3×3 grid of subplots. Rows correspond to population sizes ($N \in \{256, 512, 1024\}$, top to bottom) and columns correspond to perturbation scales ($\sigma_{\text{shift}} \in \{1, 2, 4\}$, left to right, with $\sigma_{\text{shift}} = 1$ giving the largest perturbations). Within each subplot, three lines overlay the learning curves for $k \in \{1, 3, 4\}$ number of fitness evaluations per population member. All runs re-use the same seeds and span 2000 training epochs on Pendulum-v1. The y-axis displays the mean evaluation return (in the cases where more than 1 fitness evaluation is performed) and higher values indicate better performance with a theoretical upper-bound of 0.

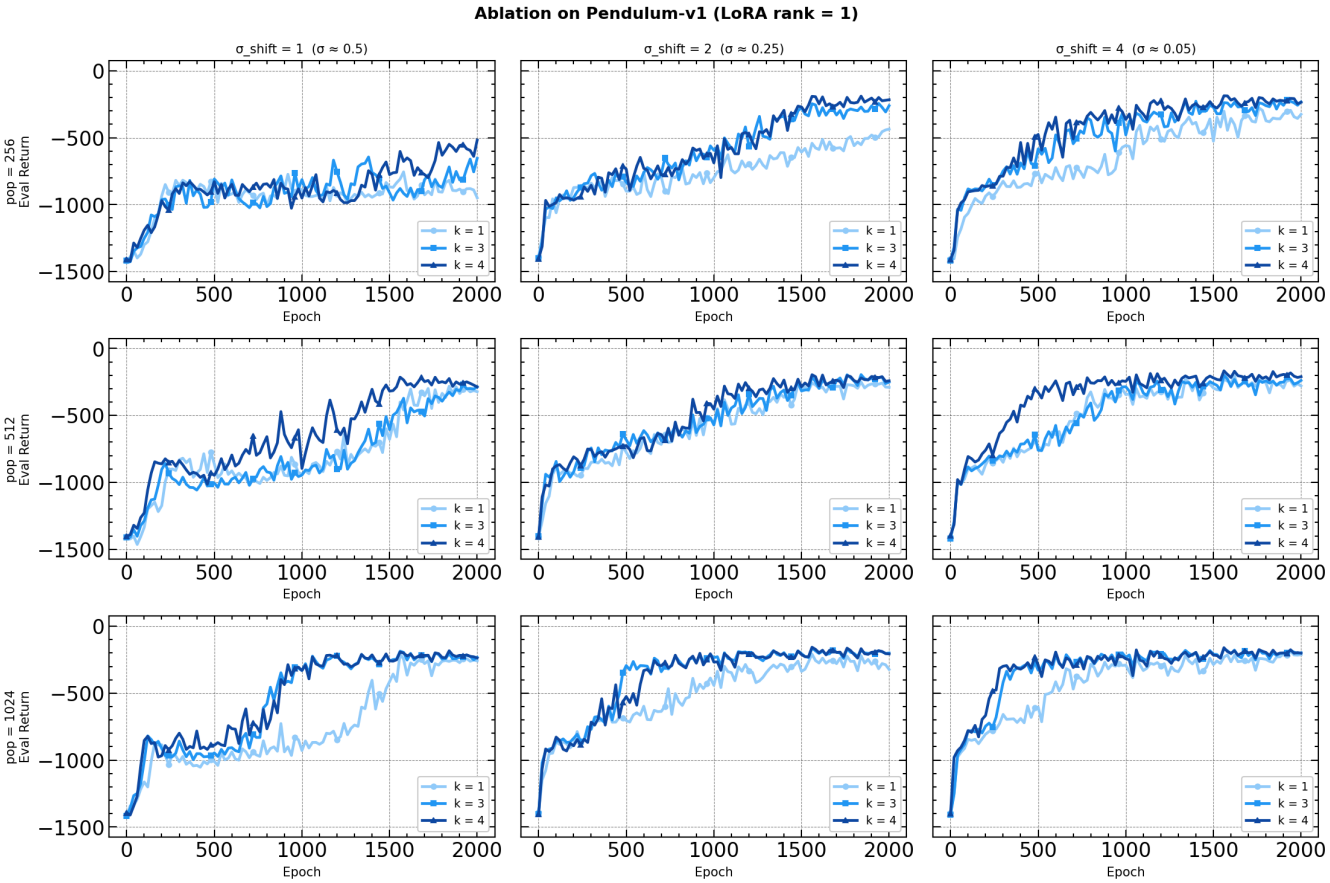

Figure 5: Ablation results for QEggRoll on Pendulum-v1 with LoRA rank $r = 1$. Each subplot shows evaluation return versus training epoch for one hyperparameter configuration. Rows vary population size: $N = 256$ (top), $N = 512$ (middle), $N = 1024$ (bottom). Columns vary perturbation scale: $\sigma_{\text{shift}} = 1$ ($\sigma \approx 0.5$, left), $\sigma_{\text{shift}} = 2$ ($\sigma \approx 0.25$, centre), $\sigma_{\text{shift}} = 4$ ($\sigma \approx 0.05$, right). Within each subplot, three lines show the effect of $k \in \{1, 3, 4\}$ fitness evaluations per member. Each run uses the same random seed for its 2000 epochs. Higher values (closer to 0) indicate better pendulum control.

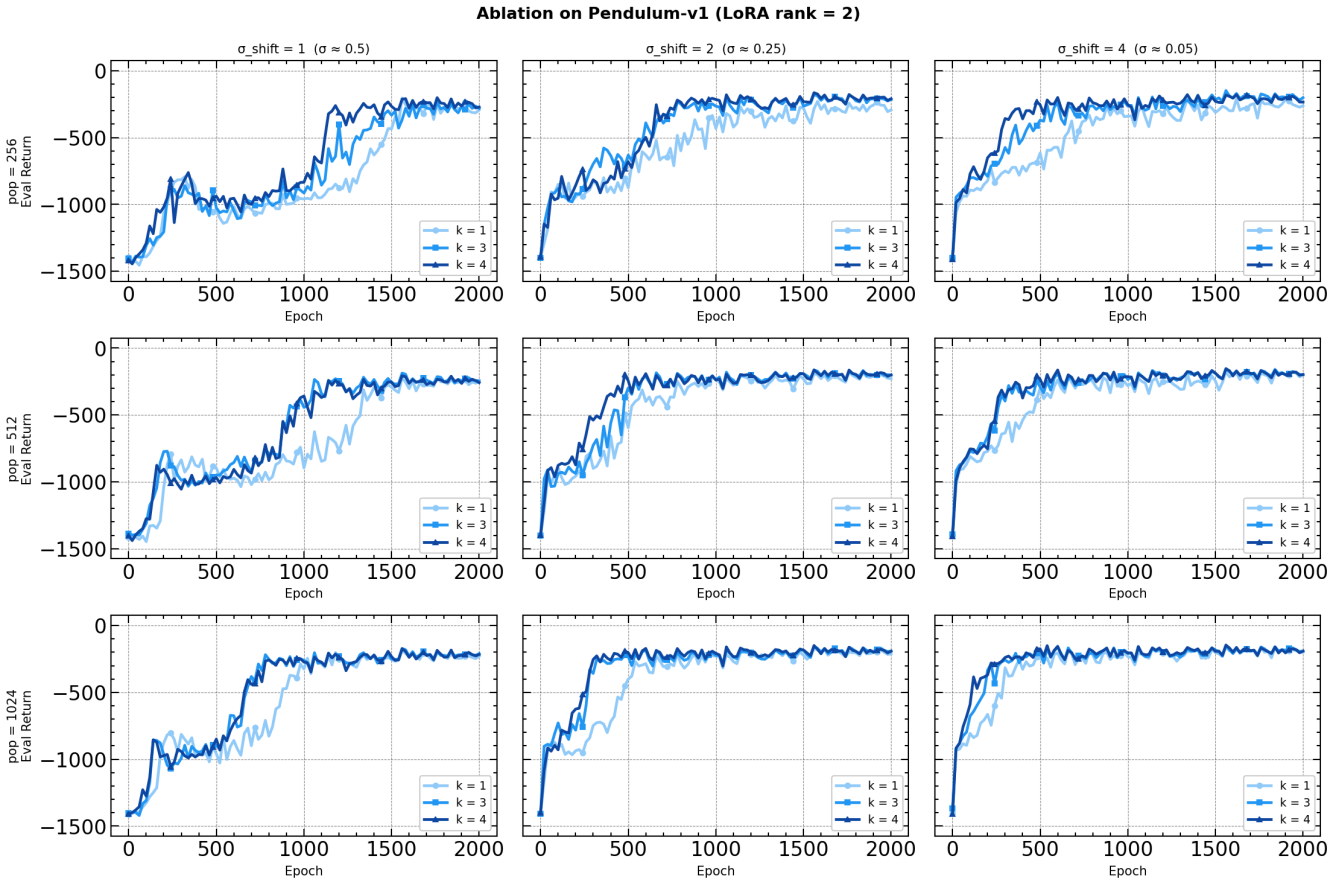

Figure 6: Ablation results for QEggRoll on Pendulum-v1 with LoRA rank $r = 2$. Each subplot shows evaluation return versus training epoch for one hyperparameter configuration. Rows vary population size: $N = 256$ (top), $N = 512$ (middle), $N = 1024$ (bottom). Columns vary perturbation scale: $\sigma_{\text{shift}} = 1$ ($\sigma \approx 0.5$, left), $\sigma_{\text{shift}} = 2$ ($\sigma \approx 0.25$, centre), $\sigma_{\text{shift}} = 4$ ($\sigma \approx 0.05$, right). Within each subplot, three lines show the effect of $k \in \{1, 3, 4\}$ fitness evaluations per member. Each run uses the same random seed for its 2000 epochs. Higher values (closer to 0) indicate better pendulum control.

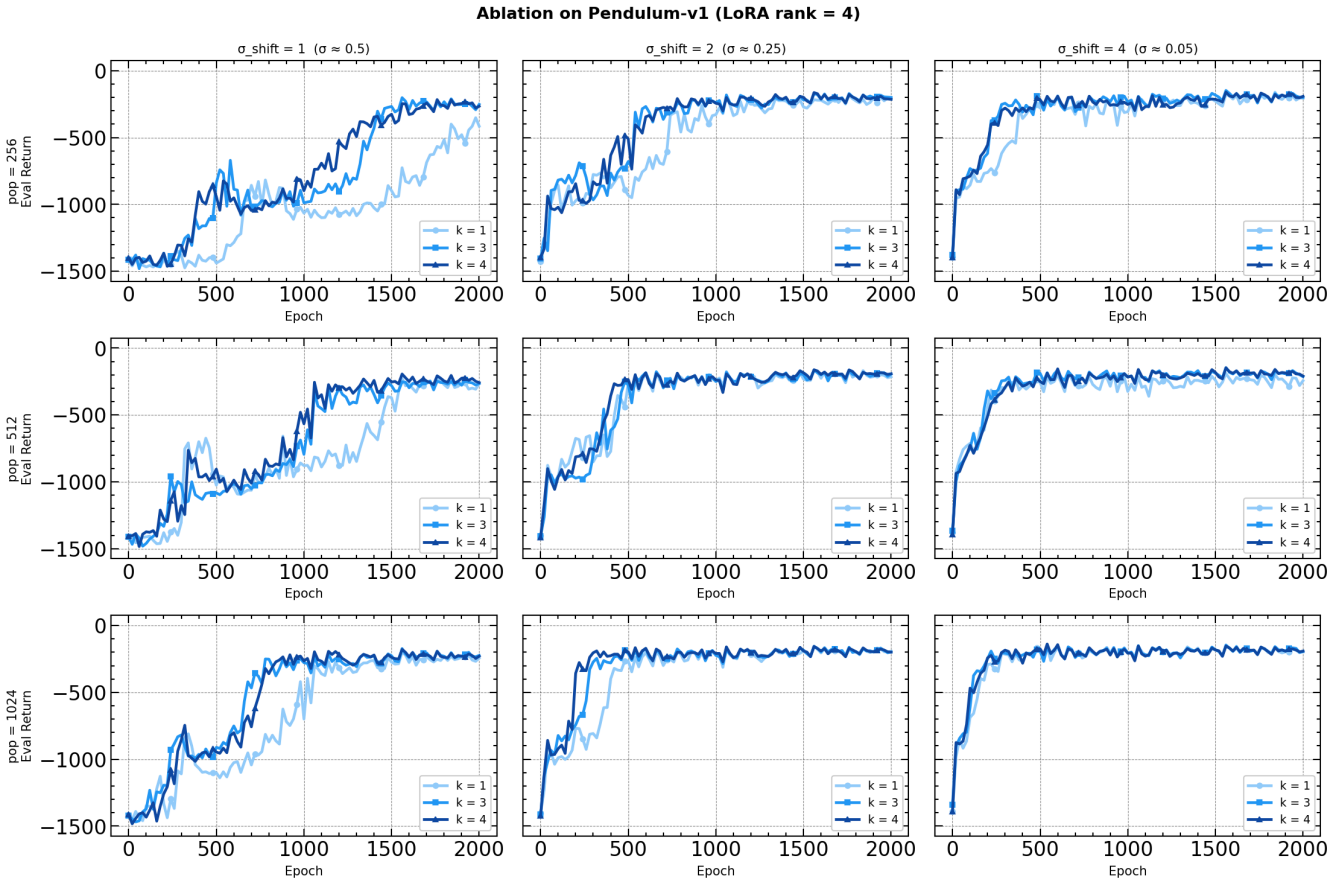

Figure 7: Ablation results for QEggRoll on Pendulum-v1 with LoRA rank $r = 4$. Each subplot shows evaluation return versus training epoch for one hyperparameter configuration. Rows vary population size: $N = 256$ (top), $N = 512$ (middle), $N = 1024$ (bottom). Columns vary perturbation scale: $\sigma_{\text{shift}} = 1$ ($\sigma \approx 0.5$, left), $\sigma_{\text{shift}} = 2$ ($\sigma \approx 0.25$, centre), $\sigma_{\text{shift}} = 4$ ($\sigma \approx 0.05$, right). Within each subplot, three lines show the effect of $k \in \{1, 3, 4\}$ fitness evaluations per member. Each run uses the same random seed for its 2000 epochs. Higher values (closer to 0) indicate better pendulum control.

Across all three rank figures there is a consistent pattern over all configurations, which is a monotonic increase from an initial return performance of about -1400 to a final performance of around -200 over the 2000 epochs (with the solo exception of the first configuration shown, with the lowers population, lowest rank, and highest effective noise). This shows that the integer evolutionary update is stable regarding hyperparameter choice.

Effect of perturbation scale: Firstly, it's important to remind that the σ_{shift} is the bit shift applied to the noise added to the candidates, meaning a lower shift produces a higher noise added. With this in mind, it is clear that the highest shift had the best performance, with less instability from epoch to epoch and more importantly, not presenting the jagged effect show by the shift of 1, which had a tendency to peak and fall as the returns reach about -1000. Moreover, we can see that another effect of reducing the noise is that the evaluation curves (k = 1, 3, 4) seem to fall

closer together, which is expected since reducing the noise brings the variance of performance down, meaning multiple evaluations would not be able to strongly distinguish better candidates apart from their pair.

Effect of population size: Population size, which is doubled row by row, is the second most visually dominant change that can be observed. It is clear that a higher population count would perform as observed but yet another important effect is its role in the gradient estimate. Because doubling the population double the antithetic pairs, the estimate is clearer and as such we can see smoother curves as population increases, especially on the first column where noise is the highest.

Effects on LoRA rank: Comparing across the 3 figures, LoRA rank defines the degrees of freedom that a candidate can vary from the mean population, with a higher rank meaning more weights can be changed independently of each other. At rank 1 we observe slower convergence to the maximum performance, with a linear-like increase, instead of the log-like look that training curves tend to resemble. This is likely due to this low degree of freedom effect, which won't allow the model to descent to the optimal direction cleanly, instead needing to do lower dimensional steps in that space. Overall, there is low differences to be analyzed between LoRA ranks 2 and 4. This is likely due to the aggregate final rank effect discussed in 3.1 which lays out that although each individual member only follows a perturbation scheme in rank r, because the final update averages all members (taking into account their performance), the final update tends to have rank '$\min(Nr, m, n)$', where m, n refer to the weight matrix dimensions (256×256). Hence, analyzing the first row of each figure, while taking into account the antithetic sampling which halves the noise direction evaluated in the aggregate formula, we have that for rank 1: $\min(\frac{N}{2}r, m, n) = \min(\frac{256}{2} \cdot 1, 256, 256) = 128$, in comparison it should be clear that for ranks 2 and 4 the minimum rank update for weight matrices will always be, in theory, 256.

Effect of fitness evaluations per member: Lastly, evaluating the k lines within each subplot, which differentiate how many evaluations are averaged together for each member gets per epoch, show a consistent but modest advantage for $k = 3$ and $k = 4$ over $k = 1$, most clearly visible during the middle phase of training (between epochs 500 and 1000). This difference is more noticeable for lower population sizes, which makes sense as with less members each epoch it becomes more important to properly evaluate them. Still, all three typically reach similar final levels by epoch 2000. This is consistent with the hypothesis that averaging multiple episode returns per member reduces the evaluation noise in the antithetic fitness signal, accelerating convergence but without substantially changing the final optimum.

7 Conclusions and Further Research

This thesis investigated whether a pure integer policy network, trained without any floating-point arithmetic under evolutionary strategies, can match the performance of state of the art algorithms in Reinforcement Learning tasks, more specifically, if the investigated approach can compete with Proximal Policy Optimization. The final model adapts the EGGROLL framework [7] to a pure integer domain, similar to EGG model (Evolutionary Guided GRU) for language tasks, but contained to RL tasks for benchmarking.

Across the tested environments, CartPole, Pendulum, and the MountainCar pair (discrete and continuous), and the ablation studies (81 hyper-parameter configurations on Pendulum), a few patterns could be observed in the comparison between QEggRoll and PPO. Namely, despite QEggRoll's lower sample efficiency on all tasks (meaning PPO achieves a higher performance with a lower amount of environmental steps), its massive ability to parallelize allows more throughput on the margins of an estimated 20× wall-clock time compared to the PPO implementation, meaning QEggRoll can achieve its peak performance faster. On the ablation experiments we saw that regardless of hyperparameter choice all but one configuration managed to reach high performance on the task in able time (the 2000 allotted epochs). Conversely, we were able to discuss about the perceived effects of each hyperparameter, making analysis on the effects that perturbation scale, population size and number of evaluations could have on the gradient signal and how a low LoRA rank can stunt learning speed by limiting the models capacity to represent the policy.

Regarding the central question of "*Whether a pure-integer network trained via evolutionary strategies could match the performance of PPO on RL tasks*" through our experiments and careful analysis of the results we are able to conclude that yes, a pure integer evolutionary approach to learning networks is competitive in both final performance and speed compared to the established RL method. Importantly, we not only found that the peak performance on the tested environments is comparable, but the higher efficiency in output speed of QEggRoll, mainly due to absence of the backpropagation step, greatly surpasses PPO's in the given regime posed for both approaches.

Limitations: In unison to the presented theory, experiment and results, it is important to keep in mind some of their limitations when interpreting the results and analysis posed. Firstly, our experiments are bounded to only 4 environments, and although we make the effort to choose the environments spanning different features (discrete versus continuous control, different observation dimensionalities, task complexity and etc...), the lack of more robust suite, with a greater number of environments, limits the reach of the generalisability of the analysis and conclusions; Secondly, due to hardware constraints and possible missed optimizations on the software, affirmations on the relative speed of algorithms and the effectiveness of larger populations for QEggRoll (the original EGGROLL paper uses population sizes up to 2^{20}) could still be subject to debate; Lastly, although the proposed research question puts and emphasis on the pure integer aspect of our proposed model, the comparisons made mixes arithmetic regimes (integer versus float) with algorithmic approaches (EGGROLL and PPO), meaning the differences in the final performance cannot be attributed solely to one or the other. Hence, no analysis where attempted in that front and the pure integer approach was pursued mostly to make the methodology attractive to those eager to use specialized hardware able to work faster instructions due to the integer operations constraint.

Future Research: The results in this thesis, along with its limitations, open several directions for further investigation:

(1) Increasing the test suite and populations sizes: Evaluating QEggRoll across a higher number of environments with sparser characteristics, along with testing higher population sizes, would allow a more generalizable analysis and conclusions regarding the effectiveness of the methodology. The higher population sizes, and increased model size for more complex tasks, would also require multi-GPU parallelism and other memory-efficient techniques which could further the perceived differences in performance comparing EGGROLL and other backpropagation/gradient methods.

(2) Deployment on integer-only hardware: Since the forward pass operates entirely in `int8`, the trained policy is directly deployable on micro-controllers and embedded processors without floating-point units, whereas gradient-trained policies would require post-training quantization to be applicable. Investigating inference on such hardware, and whether the integer training could help avoid the accuracy loss commonly observed in the post-quantization of float-trained networks, could be an interesting application of this work with practical results for on-device deep reinforcement learning.

(3) Isolating arithmetic from algorithm: As discussed in the limitations, a direct comparison between QEggRoll and a fully quantised `int8` PPO baseline would isolate the effect of the training algorithm from the effect of the arithmetic regime, allowing us to disregard the gained advantage from using a more expressive arithmetic range.

(4) Theoretical convergence of the unit-step update: The thresholded unit-step rule proposed in section 4.3 is heuristically motivated by the variance of the antithetic fitness signal, but a formal convergence analysis, that is, being able to bound the number of steps required to reach a theoretical optimal policy, would place the update rule on a firmer footing with better principled choices for the threshold and population sizes.

References

[1] Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V. Arnold, and Tim Hohm. Mirrored sampling and sequential selection for evolution strategies. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, pages 11–21, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[3] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

[4] Jarek Liesen, Chris Lu, and Robert Lange. rejax, 2024.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[7] Bidipta Sarkar, Mattie Fellows, Juan Agustin Duque, Alistair Letcher, Antonio León Villares, Anya Sims, Clarisse Wibault, Dmitry Samsonov, Dylan Cope, Jarek Liesen, Kang Li, Lukas Seier, Theo Wolf, Uljad Berdica, Valentin Mohl, Alexander David Goldie, Aaron Courville, Karin Sevegnani, Shimon Whiteson, and Jakob Nicolaus Foerster. Evolution strategies at the hyperscale, 2026.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Juan Terven. Deep reinforcement learning: A chronological overview and methods. *AI*, 6(3), 2025.

[10] Lilian Weng. Evolutionary strategies, 2019.

[11] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. Natural evolution strategies, 2011.