Comparison of Evolutionary Methods and Gradient-Based Method in Neural Network Training

Maksim Makaranka and Kamil Sulkowski

Faculty of Electronics and Information Technology, Warsaw University of Technology, Nowowiejska 15/19, Warsaw, 00-655, Poland.

Abstract

This paper presents a comparative study of evolutionary optimization techniques and a standard gradient-based method for training feed-forward neural networks. Specifically, we implement and evaluate a differential evolution algorithm and the (1+1) Cholesky Covariance Matrix Adaptation Evolution Strategy (CMA-ES) variant, along with the Adam optimizer, a widely adopted gradient-based method. Experiments are conducted on a regression task, peptide retention time prediction, using a fully connected neural network. We analyze key aspects such as training efficiency (measured by the number of fitness function evaluations), achieved prediction error, and convergence stability. All hyperparameters are empirically selected in accordance with established practices. The purpose of the study is to assess the practical utility of evolutionary strategies such as (1 + 1)-Cholsky-CMA-ES compared to backpropagation-based gradient descent, providing quantitative and qualitative insights for machine learning practitioners.

Keywords: Neural Networks, Evolutionary Algorithms, Differential Evolution, (1+1)-CMA-ES, (1+1)-Cholesky-CMA-ES, Gradient Descent, Adam Optimizer, Metaheuristic Optimization, Machine Learning

1 Introduction

Optimizing neural networks is a central challenge in machine learning, with most practical applications relying on gradient-based optimization methods such as Adam. These methods are efficient, can scale to large models, and achieve high predictive performance in a wide range of tasks. Nevertheless, they require differentiability of the

loss function and sometimes struggle with highly irregular or multimodal optimization landscapes.

Evolutionary algorithms offer an alternative: they operate without explicit gradient information and can explore the search space more globally. Among these, Differential Evolution (DE) and the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) are widely used for continuous optimization. In this study, we focus on the (1+1)-Cholesky-CMA-ES variant, an elitist evolutionary strategy that considers a single candidate solution at a time, adapting its search distribution based on the result of each trial. This approach can offer computational efficiency (utilizing the Cholesky update replacing the computationally costly matrix decomposition step) and robust performance for certain classes of optimization problems, particularly where the search space is continuous but not amenable to gradient-based methods.

In this work, we compare the performance of Adam optimizer, Differential Evolution, and (1+1)-Cholesky-CMA-ES in the context of a neural network regression task: predicting chromatographic peptide retention times from amino acid composition. Each method is evaluated using consistent data splits and model architectures. Our analysis examines training efficiency, final regression error, and convergence stability, aiming to provide practical guidance regarding the advantages and limitations of evolutionary optimization compared to classical gradient-based learning in neural network applications.

2 Methodology Overview

2.1 Datasets

All experiments are performed on benchmark datasets for peptide retention time prediction, including `serum`, `krokhin`, `mouse`, and `petritis`. Each dataset is a plain text file in which every line contains a peptide sequence and its normalized retention time (already preprocessed to the $[0, 1]$ range), separated by a tab character. For example:

```
ACDEFGHIKLMNPQRSTVWY    0.54
```

Here, "ACDEFGHIKLMNPQRSTVWY" is a peptide sequence composed of the standard twenty amino acids, and `0.54` is its normalized chromatographic retention time.

Before model training, each dataset is randomly divided into a training set (90%) and a validation set (10%). This split is held fixed for all optimization algorithms to ensure consistent and fair comparison.

2.2 Neural Network Model and Feature Encoding

The task is cast as a regression problem: given a peptide sequence, predict its chromatographic retention time. Each peptide is encoded as a simple feature vector of length 20, where each element corresponds to the count of a specific amino acid in the sequence. This encoding does not use any positional or local sequence information,

but suffices to capture global amino acid contributions and provides a common basis for optimizer comparison.

The predictive model is a fully connected feedforward neural network with two hidden layers. The architecture consists of:

- **Input:** A 20-dimensional count vector, where each element represents the number of times each standard amino acid appears in the sequence.
- **Hidden Layer 1:** Fully connected layer (ReLU activation), width selected as a hyperparameter.
- **Hidden Layer 2:** Fully connected layer (ReLU activation), width selected as a hyperparameter.
- **Output Layer:** Linear output node producing the predicted retention time.

This architecture is kept fixed for all optimization algorithms to ensure a fair comparison.

2.3 Optimization Algorithms

We compare three optimization methods for neural network parameter learning:

Adam

Adam is a widely used gradient-based optimizer that combines momentum and adaptive step sizes for each parameter. At each iteration, the optimizer computes the gradient of the loss function (SmoothL1loss[1]) with respect to the network weights using backpropagation, and updates the weights according to running estimates of the first and second moments of the gradients. Key hyperparameters include the learning rate and the number of training epochs.

Differential Evolution (DE)

Differential Evolution (DE) is a population-based, gradient-free evolutionary algorithm designed for continuous numerical optimization. In each generation, a set of candidate solutions (where each solution corresponds to a vector of neural network parameters) is maintained. To explore the search space, new candidate vectors are proposed by combining several randomly selected solutions from the current population. Mutation is performed by adding the scaled difference between two candidates to a third one, while recombination (crossover) allows the mixture of parameters from different candidates to create further diversity. Each new candidate replaces the current one only if it achieves a better fitness, i.e., a lower training loss on the target task. Key hyperparameters of DE include the size of the population, the differential weight (which controls the magnitude of the mutation), the crossover probability (which governs how often parameters are exchanged between solutions), and the total number of generations (iterations) over which the algorithm runs.

(1+1)-CMA-ES

The $(1 + 1)$ Covariance Matrix Adaptation Evolution Strategy ($(1 + 1)$-CMA-ES) is an evolutionary algorithm for black-box optimization of continuous functions [2, 3]. It is *elitist*: at each generation, only the best solution found so far is retained.

Our implementation is based on the Cholesky-update version of (1 + 1)-CMA-ES, as described in [2, 3], with some simplifications for computational efficiency in high-dimensional settings. The algorithm operates as follows:

1. **Sampling:** At iteration t, generate a new candidate solution:

$$\mathbf{z}^{(t)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad \mathbf{x}_{\text{new}} = \mathbf{m}^{(t)} + \sigma^{(t)} \mathbf{A}^{(t)} \mathbf{z}^{(t)}$$

   where $\mathbf{m}^{(t)}$ is the current solution, $\sigma^{(t)}$ is the global step size, and $\mathbf{A}^{(t)}$ is the Cholesky factor of the current covariance matrix.
2. **Selection:** Evaluate $f(\mathbf{x}_{\text{new}})$. If $f(\mathbf{x}_{\text{new}}) < f(\mathbf{m}^{(t)})$, accept the candidate: set $\mathbf{m}^{(t+1)} = \mathbf{x}_{\text{new}}$; otherwise, keep $\mathbf{m}^{(t+1)} = \mathbf{m}^{(t)}$.
3. **Step-size adaptation:** Update the smoothed success rate and the step size using the 1/5th success rule:

$$p_{\text{succ}}^{(t+1)} = (1 - c_p) p_{\text{succ}}^{(t)} + c_p \cdot \mathbb{I}_{\text{success}}$$

$$\sigma^{(t+1)} = \sigma^{(t)} \cdot \exp\left(\frac{p_{\text{succ}}^{(t+1)} - p_{\text{target}}}{d_\sigma (1 - p_{\text{target}})} \right)$$

   where $\mathbb{I}_{\text{success}} = 1$ if the candidate was accepted, 0 otherwise; c_p is the smoothing parameter; p_{target} is the target success rate (typically 0.2); and d_σ is a damping factor.
4. **Covariance adaptation:** If the candidate is accepted and $p_{\text{succ}}^{(t+1)} < p_{\text{thresh}}$, we update the Cholesky factor $\mathbf{A}$ using a rank-one update in the direction of the most recent successful mutation:

$$\mathbf{A}^{(t+1)} = c_a \mathbf{A}^{(t)} + \text{coeff} \cdot (\mathbf{A}^{(t)} \mathbf{z}^{(t)})(\mathbf{z}^{(t)})^\top$$

   where $c_a = \sqrt{1 - c_{\text{cov}}}$, c_{cov} is the covariance learning rate, and

$$\text{coeff} = \frac{c_a}{\|\mathbf{z}^{(t)}\|^2} \left(\sqrt{1 + \frac{(1 - c_a^2)\|\mathbf{z}^{(t)}\|^2}{c_a^2}} - 1 \right)$$

   This efficient rank-one Cholesky update [3] adapts the search distribution to reinforce successful search directions. In contrast to the full algorithm, which uses an evolution path (an exponentially weighted sum of successful steps) and maintains the Cholesky inverse for theoretical robustness, our implementation omits these for computational efficiency, updating $\mathbf{A}$ in the direction of the most recent mutation only. This simplification is common and effective for large-scale continuous optimization.
5. **Repeat** for a fixed number of generations, or until stopping criteria are met (e.g., no improvement for a set number of generations, or σ falls below a minimum threshold).

Key principles:

- Only two solutions are maintained at any time: the current best ("parent") and a single candidate ("offspring").
- Step-size (σ) increases if improvements are frequent, and decreases if not, enabling automatic adaptation to the problem's landscape.
- The covariance (shape) matrix $\mathbf{A}$ adapts to exploit correlations in successful steps, making the search efficient for non-separable or ill-conditioned problems.
- The algorithm is simple, requires few hyperparameters, and is especially effective for unimodal or smooth problems.

Differences from the original formulation:

- Our implementation *omits the evolution path* $\mathbf{p}_c$ in the covariance update for computational efficiency, as is common in large-scale neural network optimization. The direction of adaptation is given directly by the successful mutation vector $\mathbf{z}^{(t)}$.
- We do *not* implement margin correction, as our problem is continuous; margin correction is only necessary for mixed-integer or discrete variables.
- We implemented early stopping mechanism which terminates the algorithm's run if σ value reaches value lower than configured `CMAES_SIGMA_MIN` parameter or training loss was not improved for `CMAES_EARLY_STOP_PATIENCE` consecutive generations.

All formulas and adaptation rules described above are based on the original $(1 + 1)$-CMA-ES and its analysis as presented in [2, 3].

2.4 Hyperparameter Selection

All main hyperparameters in the experiments were selected empirically and adjusted to ensure robust performance across optimizers and datasets. Hyperparameter values are managed centrally via an environment file (`.env`), which is loaded at runtime for reproducibility.

Adam: Key hyperparameters include the learning rate (`ADAM_LR`), number of training epochs (`ADAM_EPOCHS`), and batch size (`BATCH_SIZE`). These control the magnitude, stability, and frequency of parameter updates during gradient-based training.

Differential Evolution (DE): Main hyperparameters are the population size (`DE_POP_SIZE`), mutation factor (`DE_F`), crossover probability (`DE_CR`), and the number of generations (`DE_GENERATIONS`). These control the diversity, exploration, and exploitation in the evolutionary process.

(1+1)-CMA-ES: Most key hyperparameters, including the target success rate for step-size adaptation (`CMAES_P_TARGET`), smoothing parameter for the running success rate (`CMAES_C_P`), and the threshold for triggering covariance adaptation (`CMAES_P_THRESH`), are set according to values recommended in the literature. The number of generations (`CMAES_GENERATIONS`) and the initial mutation step size (`CMAES_INITIAL_SIGMA`) were chosen empirically based on preliminary experiments. For the covariance matrix adaptation rate (`CMAES_C_COV`) and the damping factor for step-size changes (`CMAES_D_SIGMA`), default formula-based values were found to be suboptimal for our network dimensionality ($n = 96$), so these parameters were selected through an additional empirical search, as described in Section 5.

General: All algorithms use the same model size, data splits, and loss function for fair comparison. The random seed (`RANDOM_SEED`) is fixed for reproducibility.

A complete listing of all hyperparameter settings and their exact values is available in the accompanying implementation and `.env` file.

3 Implementation

The experiment was implemented in Python using PyTorch for modularity and clarity. Each main file in the `src/` directory has its own distinct function in the workflow:

Code Overview

- **dataset.py**: Defines the `PeptideDataset` class for loading peptide sequence data and normalized retention times, and encoding sequences as amino acid count vectors.
- **model.py**: Implements the neural network architecture (`RetentionPredictor`), a feedforward regressor with two hidden layers.
- **train_adam.py**: Implements the training pipeline using the Adam optimizer from PyTorch, including data preparation, model instantiation, and epoch-wise training and validation.
- **train_de.py**: Implements neural network training using the Differential Evolution algorithm with population-based parameter search.
- **train_cmaes.py**: Implements neural network training using the (1+1)-CMA-ES algorithm, including candidate generation, update rules, and result reporting.
- **utils.py**: Provides shared utility functions for model evaluation, parameter flattening/restoration, and fitness calculation used across evolutionary algorithms.

How to Run

Dependencies: This project uses `poetry` for dependency management. To install all required dependencies, run `poetry install`.
Running Training Scripts: To reproduce results for a given optimizer, run the appropriate script from the `src` directory:

- `poetry run python src/train_adam.py`
- `poetry run python src/train_de.py`
- `poetry run python src/train_cmaes.py`

Relevant settings, such as dataset path, network architecture, and optimizer hyperparameters, are provided via environment variables set in the `.env` configuration file.
Interactive Experiments: For running and analyzing experiments across multiple datasets and algorithms, use the `Experiments.ipynb` notebook. This notebook allows you to monitor the training process, compare learning curves and scatter plots for each algorithm, and view main regression metrics for all runs.

4 Experimental Setup

As stated previously, the hyperparameters of the algorithms were selected empirically and with guidance from relevant literature. However, for the first part of the experiments, the optimal hyperparameter values for (1+1)-CMA-ES algorithm were explored. Referenced works [2, 3] suggest default values for step-size control and covariance matrix adaptation:

$$\mathbf{d} = 1 + \frac{n}{2}, \qquad \mathbf{p}_{\text{target}} = \frac{2}{11}, \qquad \mathbf{c}_{\text{p}} = \frac{1}{12}, \qquad \mathbf{c}_{\text{cov}} = \frac{2}{n^2 + 6}, \qquad \mathbf{p}_{\text{thresh}} = 0.44.$$

Notably, the values of d and c_{cov} are directly dependent on the problem dimensionality n. Preliminary experiments indicated that using these formulas with our network's dimensionality ($n = 96$ parameters of the neural network) yielded suboptimal results. Thus, a basic hyperparameter search was conducted for these two parameters at the beginning of the experimental phase.

In the main part of the experiments, the neural network was trained using all three optimization algorithms - Adam, (1+1)-CMA-ES, and Differential Evolution - across all four datasets. To ensure statistical robustness and evaluate convergence stability, each experiment was repeated multiple times using different random seeds.

The quality of the trained models was evaluated using standard regression metrics:

- **Mean Squared Error (MSE)** - sensitive to large errors, highlights the magnitude of prediction deviations.
- **Mean Absolute Error (MAE)** - reflects the average magnitude of errors, more robust to outliers.
- **Coefficient of Determination (R^2)** - indicates how well predictions explain the variability of the target; values closer to 1 denote better fits.
- **Training time** - total duration of the training run.
- **Number of fitness evaluations** - used as a measure of computational cost, especially relevant for evolutionary algorithms.

To evaluate the convergence stability of the algorithms, the training loss was recorded after every generation or epoch. For the CMA-ES algorithm, the step size σ was monitored in each generation, while for Differential Evolution, the standard deviation of the parameter vectors in the population was computed. These statistics were used to analyze the dynamics of convergence and population spread over time.

To illustrate the results and facilitate comparison between algorithms, several types of plots were generated:

- **Learning curves**: displaying train and validation loss across generations/epochs for all three algorithms.
- **Scatter plots (Predicted vs True)**: visualizing the accuracy of regression predictions; ideally, points should align along the $x = y$ diagonal.
- **Convergence curves**: visualizing the evolution of σ (for CMA-ES) or population standard deviation (for DE) across generations, to reflect convergence behavior.
- **Learning dynamics plots**: simultaneously presenting train loss, validation loss, and per-generation training time to explore learning efficiency and cost.

The experiments were executed on a machine running Debian GNU/Linux 12 (Bookworm), equipped with an 8-core AMD Ryzen 7 7800X3D CPU, an NVIDIA GeForce RTX 4080 GPU, and 32 GB of RAM.

5 Results

5.1 (1+1)-CMA-ES Parameters Search

In the initial experiments with the (1+1)-CMA-ES algorithm, we used the default values for hyperparameters as suggested in the literature [2, 3]. In particular, the hyperparameters for step-size control and covariance matrix adaptation were calculated based on the number of model parameters $n = 96$ (the size of our neural network's parameter vector):

$$d = 1 + \frac{n}{2} = 49.0, \qquad c_{\text{cov}} = \frac{2}{n^2 + 6} = \frac{2}{9216 + 6} \approx 0.0002169$$

Although these values are theoretically well-founded, the training results obtained using them were unsatisfactory across multiple datasets. For illustration, below we present the results for the largest dataset `mouse`.

Final Metrics Summary (Default Hyperparameters, mouse dataset):

- R^2: **-6.590**
- Mean Squared Error (MSE): **0.3940**
- Mean Absolute Error (MAE): **0.5854**
- Evaluation Calls: **200**
- Training Time: **25.50 s**

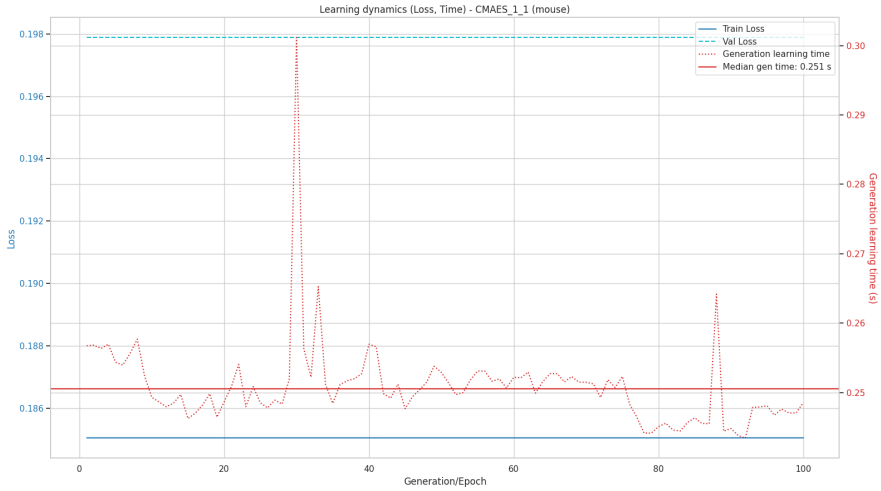

Fig. 1 Learning dynamics: mouse dataset, (1+1)-CMA-ES with default parameters

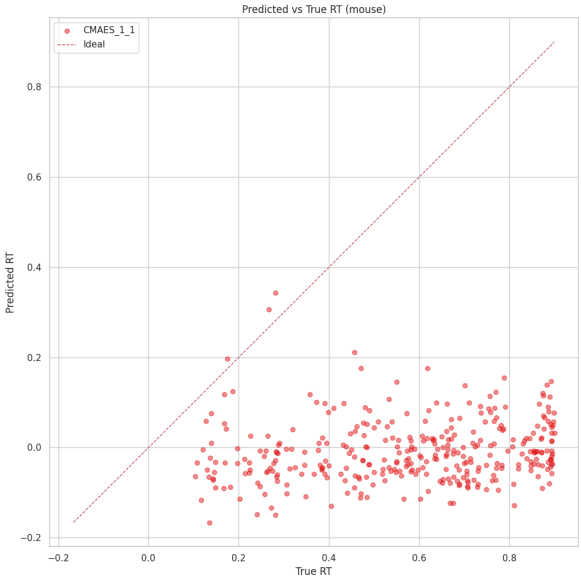

Fig. 2 Predicted vs. true retention times: mouse dataset, (1+1)-CMA-ES with default parameters

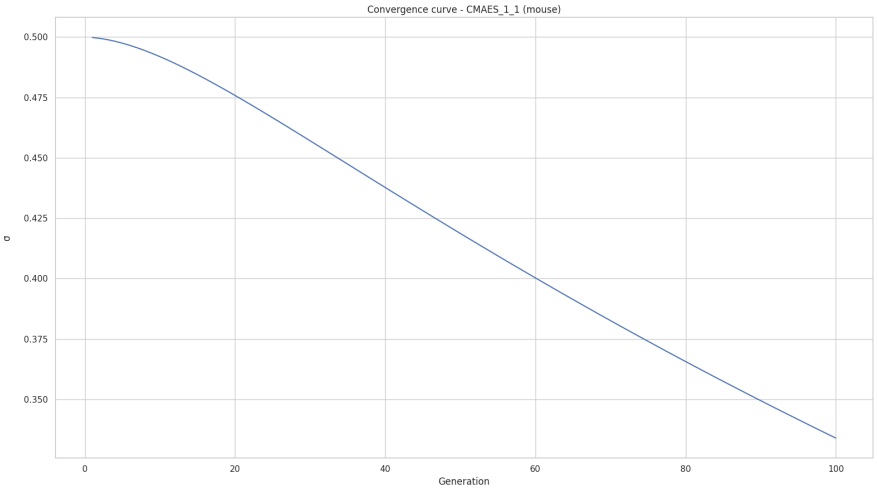

Fig. 3 Convergence curve (σ per generation) for (1+1)-CMA-ES with default parameters

As seen from both the performance metrics and the plots, the model performs poorly. In particular, the R^2 score indicates a severe mismatch between predictions and ground truth. This is further supported by the scatter plot, where predictions are largely spread and not concentrated around the identity line.

Moreover, the learning dynamics reveal that the training and validation loss remain nearly constant throughout the entire training process, showing no meaningful improvement. This indicates that the algorithm fails to make progress during optimization. Consequently, the training was terminated early due to the lack of improvement in the loss value for 100 consecutive generations, as dictated by the early stopping mechanism.

The convergence curve supports this observation: the step-size parameter σ steadily decreases generation by generation, which suggests a lack of successful updates. Such behaviour may point to poor adaptation of the algorithm's search dynamics to the problem landscape.

As a result, we conducted a targeted search over the d and c_{cov} parameters in subsequent experiments in order to improve training performance.

To guide the search for better values, we used not only the final regression metrics, but also monitored the learning dynamics - specifically the evolution of training and validation loss, and the shape of the convergence curves (i.e., behavior of σ over generations).

Several configurations were tested across multiple datasets. Below, we present the final validation metrics for different values of `CMAES_C_COV` and `CMAES_D_SIGMA` for the `mouse` dataset:

Table 1 Validation metrics for various CMA-ES hyperparameter settings on the `mouse` dataset.

C_COV	D_SIGMA	R^2	MSE	MAE	Eval Calls
0.002	10.0	-0.828	0.0949	0.2378	1000
0.02	2.0	0.288	0.0370	0.1505	1000
0.2	2.5	0.088	0.0473	0.1714	1000
0.02	2.5	-0.280	0.0664	0.2001	1000
0.06	2.5	0.173	0.0429	0.1672	1000
0.06	5.0	0.027	0.0505	0.1815	1000
0.06	**2.8**	**0.357**	**0.0334**	**0.1452**	**1000**
0.06	3.0	0.106	0.0464	0.1699	1000
0.12	2.8	0.209	0.0410	0.1629	1000

Among all tested configurations, the pair `CMAES_C_COV` $= 0.06$ and `CMAES_D_SIGMA` $= 2.8$ provided the best overall result, achieving the lowest MSE and MAE, as well as the highest R^2 score. The learning dynamics (Figure 4) show a consistent decrease in training and validation loss, indicating proper convergence. The predicted vs. true retention times shown in the scatter plot (Figure 5) further confirm good model accuracy as points tend to be placed alongside the diagonal line. Finally, the convergence curve in Figure 6 reveals a steady decay in σ over generations and convergence at the lower value, indicating stable convergence of the algorithm.

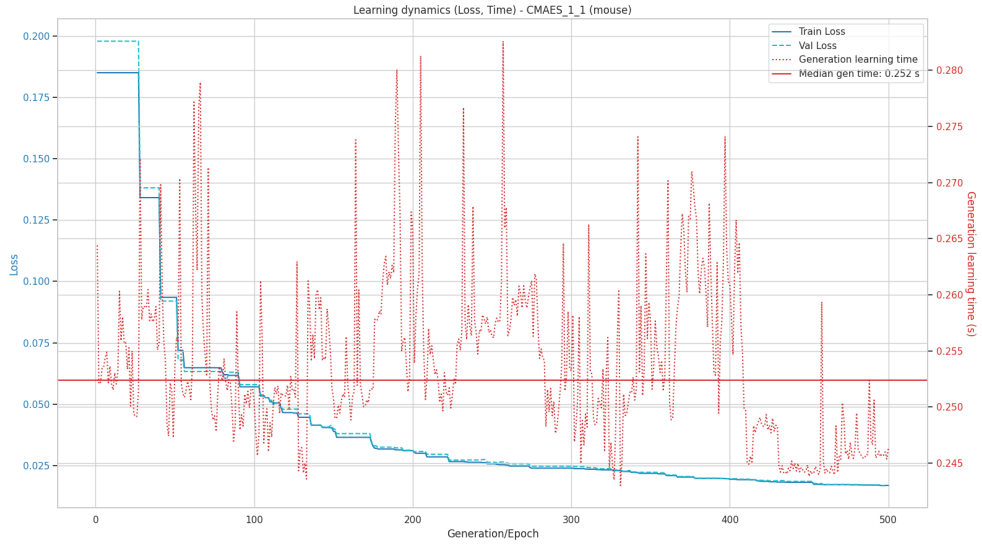

Fig. 4 Learning dynamics for best CMA-ES configuration: $c_{\text{cov}} = 0.06$, $d = 2.8$.

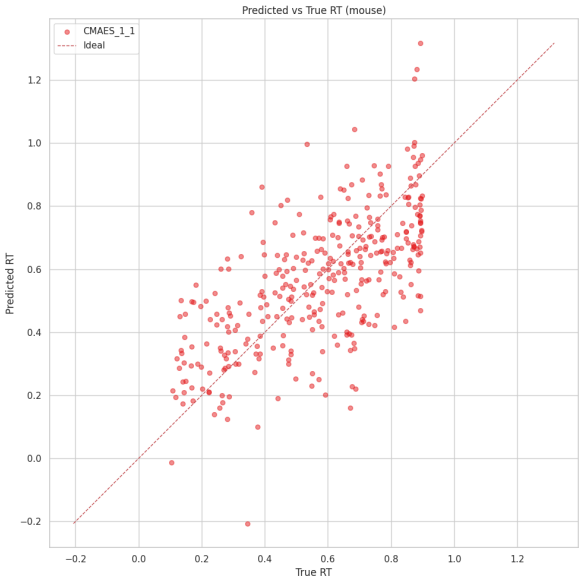

Fig. 5 Scatter plot of predicted vs. true retention times (best CMA-ES config).

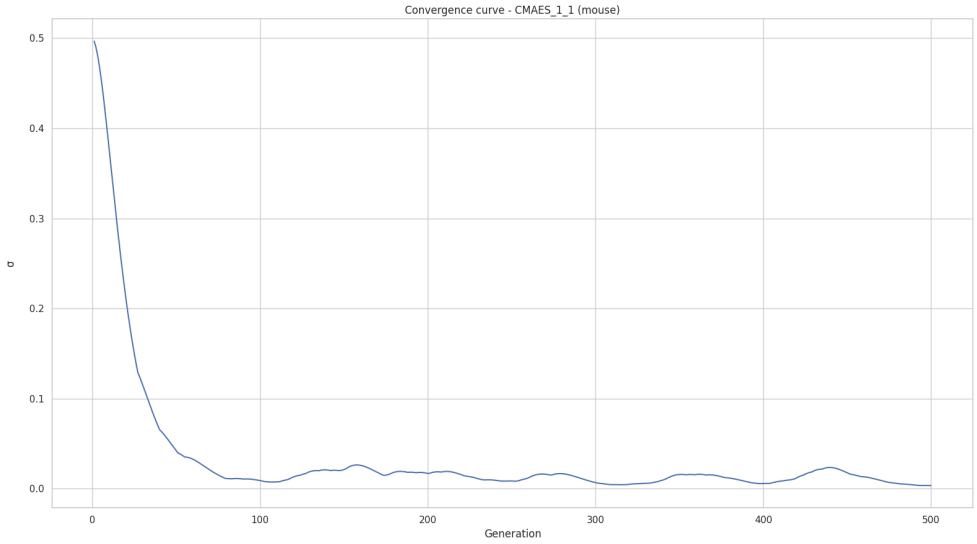

Fig. 6 Convergence curve: σ evolution for best configuration.

5.2 Algorithms performance comparison

Having established satisfactory hyperparameter values for the (1+1)-CMA-ES algorithm, we now proceed to a broader comparison of all three optimization strategies: gradient-based Adam, evolutionary Differential Evolution (DE), and the (1+1)-CMA-ES. The comparison was carried out across four different datasets, each representing a variant of the peptide retention time prediction problem, with training repeated using several random seeds to assess the robustness and stability of each method.

For each algorithm-dataset combination, we recorded standard regression performance and convergence metrics on the validation set as stated before. Additionally, we measured the number of objective function evaluations and the total training time as indicators of computational efficiency. Each experiment was repeated three times using different random seeds. The final reported metric values represent the average across these three independent runs.

This section presents a comparative analysis of the final validation metrics, as well as graphical visualizations including learning curves, predicted versus true value scatter plots, and convergence curves.

5.2.1 Regression metrics evaluation and analysis

Table 2 Validation metrics averaged over 3 runs for each algorithm on each dataset.

Algorithm	R^2	MSE	MAE	Eval Calls	Time (s)
mouse					
Adam	**0.895**	**0.00523**	**0.0562**	**200**	**61.46**
CMA-ES	0.276	0.0362	0.1506	1000	126.46
DE	0.793	0.0104	0.0813	10520	2284.69
krokhin					
Adam	0.858	0.00386	**0.0456**	**200**	**3.51**
CMA-ES	-1.331	0.0640	0.1893	591	4.17
DE	**0.864**	**0.00377**	0.0475	10520	124.17
petritis					
Adam	**0.781**	**0.00539**	**0.0537**	**200**	**26.57**
CMA-ES	0.156	0.0202	0.1124	871	48.84
DE	0.750	0.00607	0.0590	10520	1014.53
serum					
Adam	**0.870**	**0.00359**	**0.0460**	**200**	**11.11**
CMA-ES	0.125	0.0246	0.1189	1000	22.49
DE	0.756	0.00673	0.0622	10520	404.98

Preliminary analysis of the experimental results indicates that the metrics obtained are both quantitatively and qualitatively consistent. The regression metrics (R^2, MSE, MAE) clearly differentiate the performance of the tested algorithms, and the number of objective function evaluations and total training time correlate well with the expected computational complexity of each method. A summary of the aggregated results is presented in Table 2.

5.2.2 Visualization of training behavior on the `mouse` and `krokhin` datasets

We present a series of graphs to compare and illustrate the training behaviour of each algorithm. For clarity, the visualizations correspond to a single representative run with one selected random seed and for the biggest and the smallest datasets (`mouse` and `krokhin`). All plots for each run across different seeds and datasets are available in the `results` directory.

Learning Curves Analysis

The following figures present *learning curves*, which illustrate the training and validation loss values recorded at each generation or epoch during the learning process for each algorithm.

Figure 7 and Figure 8 show the results for the `krokhin` and `mouse` datasets, respectively. The training and validation loss are plotted over time for Adam, Differential Evolution (DE), and (1+1)-CMA-ES.

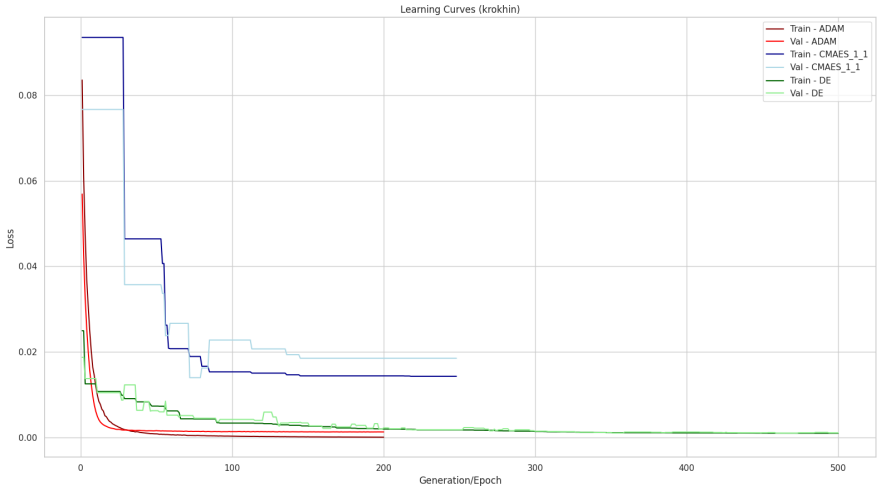

Fig. 7 Learning curves for the `krokhin` dataset.

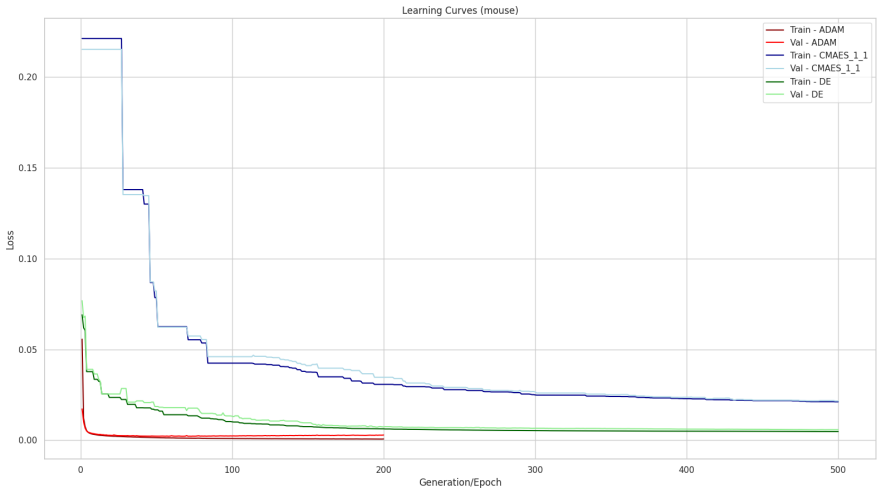

Fig. 8 Learning curves for the `mouse` dataset.

On the `krokhin` dataset, the Adam optimizer exhibits the fastest and most stable convergence. It achieves low training and validation loss values within just a few epochs, with minimal overfitting. The Differential Evolution (DE) algorithm also performs reasonably well, although it converges more slowly. In contrast, the (1+1)-CMA-ES algorithm demonstrates significantly slower convergence and reaches much higher loss values, suggesting limited effectiveness on this dataset.

For the `mouse` dataset, a similar pattern is observed. Adam achieves low loss values quickly and maintains a small gap between training and validation loss, indicating good generalization. DE improves gradually over time, with stable progress. Meanwhile, CMA-ES once again converges slowly and exhibits the highest overall loss, confirming its relatively lower performance in this regression task.

These observations align with the quantitative metrics presented earlier in Table 2, and they provide further insights into the learning behaviour of each optimization strategy.

Predicted vs True Retention Time Scatter Plots

Figures 9 and 10 present the scatter plots comparing predicted and true retention time (RT) values for the `krokhin` and `mouse` datasets, respectively. These plots help to visualize the regression accuracy of the trained neural networks by showing how closely the predictions align with the ideal diagonal line $y = x$.

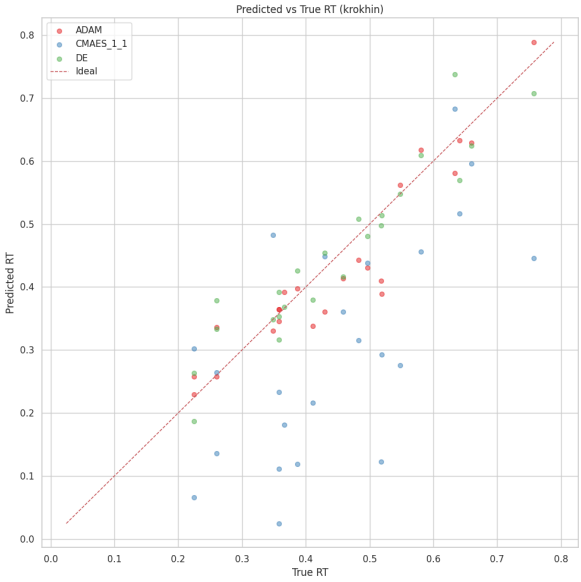

Fig. 9 Predicted vs True RT values on the `krokhin` dataset.

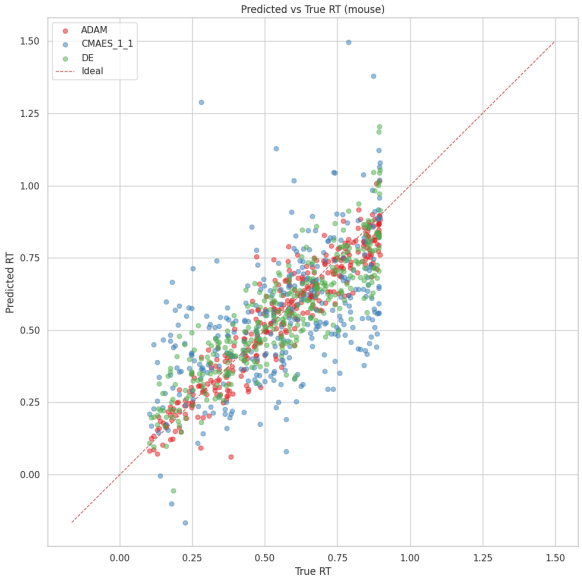

Fig. 10 Predicted vs True RT values on the `mouse` dataset.

For the `krokhin` dataset, ADAM and DE exhibit relatively accurate predictions close to the ideal diagonal, whereas CMA-ES displays noticeably more dispersed and biased predictions. In the larger `mouse` dataset, all methods perform better overall, with ADAM again providing the tightest clustering around the ideal line, followed by DE and CMA-ES.

Convergence Curves

To assess the convergence behavior of the evolutionary algorithms, we monitored the value of the σ parameter in (1+1)-CMA-ES and the population standard deviation in Differential Evolution (DE) across generations. These metrics reflect how the algorithms explore and exploit the search space during training.

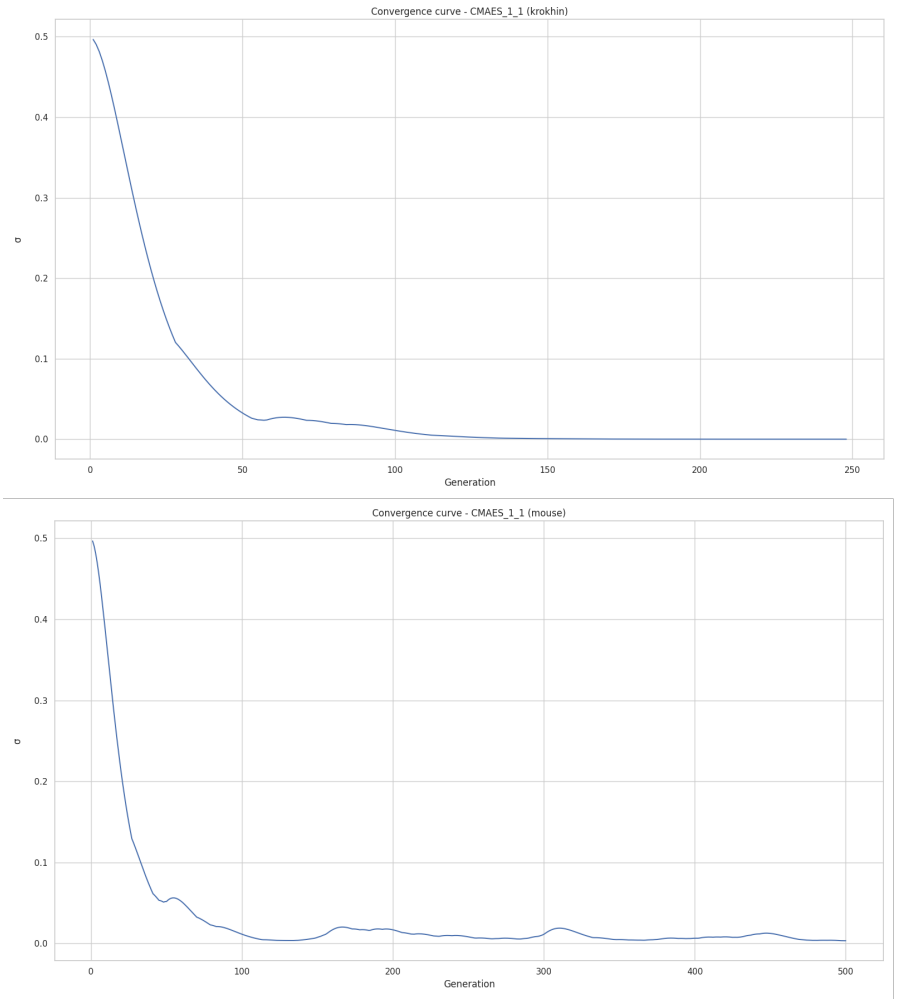

Fig. 11 Convergence curves for CMA-ES (σ value per generation) for the `krokhin` and `mouse` datasets.

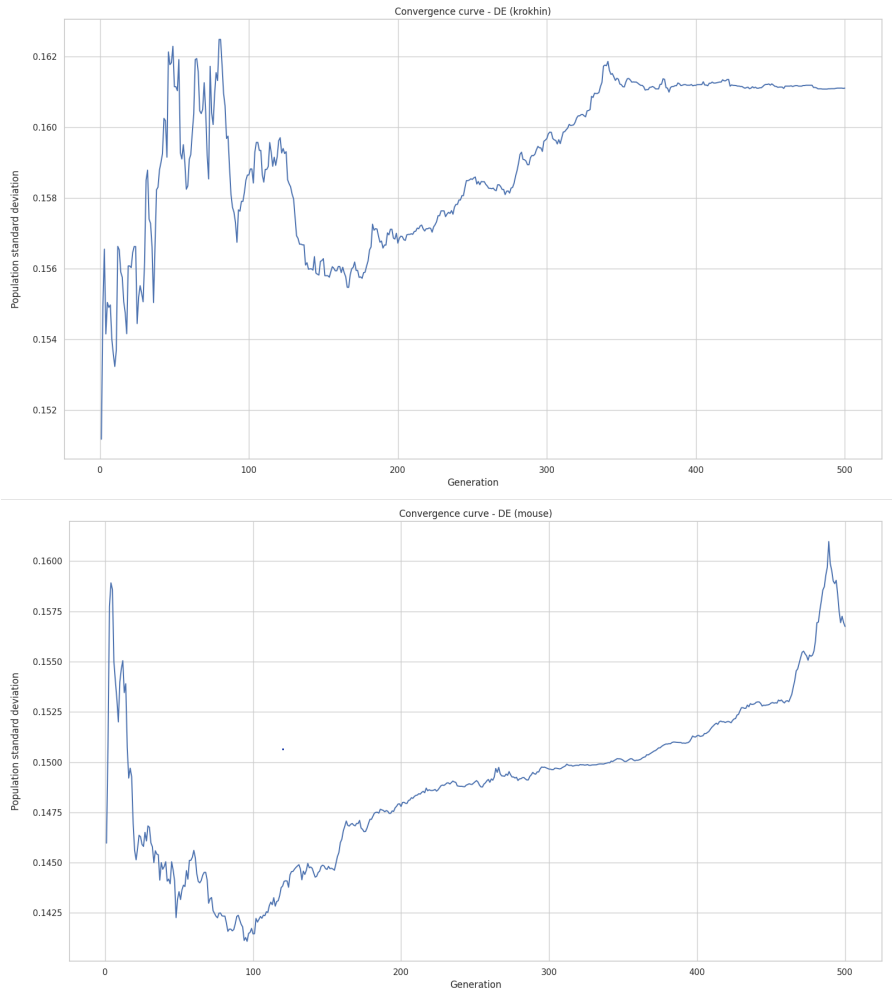

Fig. 12 Convergence curves for DE (population standard deviation per generation) for the `krokhin` and `mouse` datasets.

For CMA-ES, the σ parameter exhibits a rapid decline and then plateaus, indicating early convergence and a narrowing of the search space. In contrast, the population standard deviation in DE fluctuates or even increases slightly over time, which suggests ongoing exploration and slower convergence.

Learning dynamics

The following figures illustrate the learning dynamics of all three optimization algorithms by showing the evolution of training and validation loss across generations or epochs, along with the computation time per generation. Each plot includes two y-axes: one representing the loss (left axis) and the other representing the generation

time in seconds (right axis), with a horizontal line indicating the median generation time for reference.

Fig. 13 Learning dynamics for ADAM on `krokhin` and `mouse` datasets.

Fig. 14 Learning dynamics for (1+1)-CMA-ES on `krokhin` and `mouse` datasets.

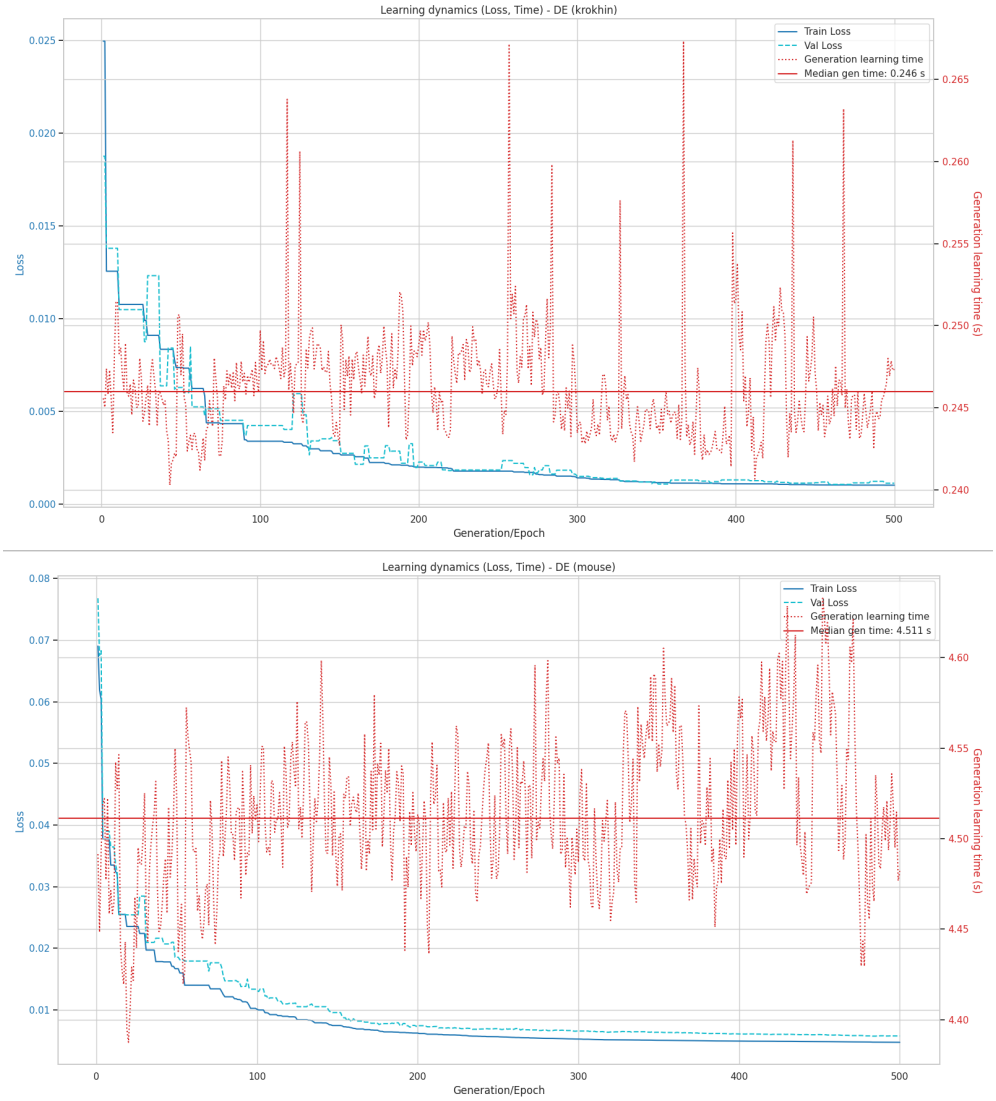

Fig. 15 Learning dynamics for Differential Evolution (DE) on `krokhin` and `mouse` datasets.

For ADAM, the learning process is both fast and smooth, with consistently low per-epoch times and steadily decreasing loss values. The (1+1)-CMA-ES shows moderately longer generation times, with noisy but generally improving loss. In contrast, the DE algorithm exhibits significant computational cost, especially on the larger `mouse` dataset, and a more erratic learning curve with delayed convergence, as evidenced by the relatively high and fluctuating generation times.

6 Discussion

The gradient-based Adam optimizer demonstrated the best overall performance across all datasets. It consistently achieved the highest regression accuracy, as evidenced by the highest R^2 scores and the lowest MSE and MAE values. Furthermore, it required the fewest objective function evaluations and the shortest total training time. These results are expected due to Adam's ability to exploit gradient information and apply adaptive learning rates, enabling efficient and stable weight updates even in high-dimensional neural network training.

Differential Evolution (DE) achieved competitive accuracy, in some cases comparable to or slightly lower than Adam (and even slightly outperforming Adam on the smallest dataset). However, it incurred significantly higher computational costs. The number of objective function evaluations reached over 10,000 for each experiment, and the training time was often an order of magnitude longer than for the other two methods. This can be attributed to the algorithm's population-based nature and lack of gradient information, resulting in slower convergence and higher computational burden.

The (1+1)-CMA-ES algorithm, despite hyperparameter tuning, performed the worst in terms of prediction accuracy. It consistently yielded the lowest R^2 and highest error metrics across datasets. Nonetheless, its computational efficiency was noticeably better than DE: the number of function calls was tightly bounded and training time remained modest – also thanks to the implementation of the Cholesky update. This suggests that while CMA-ES may converge quickly, it does so toward suboptimal regions of the parameter space in this problem, possibly due to the complexity of the loss landscape or insufficient exploration in high dimensions.

Insights from visualizations. The learning curves revealed distinct convergence behaviors. Adam consistently reduced both training and validation loss quickly and smoothly, indicating stable learning. DE also converged but more gradually and with more variance, while CMA-ES exhibited noisy and plateaued convergence.

Scatter plots of predicted versus true retention times showed that Adam's predictions clustered tightly around the ideal $x = y$ line, reflecting strong generalization. DE's predictions were more dispersed, while CMA-ES clearly underperformed, especially on the larger datasets, as many outliers can be seen on the graph.

Convergence curves further illustrated algorithmic differences: CMA-ES showed rapidly shrinking step-size σ across generations, indicating convergence (or premature convergence), whereas DE's population standard deviation remained high or even increased, reflecting persistent exploration, which may explain its high computational cost and slow convergence.

Learning dynamics plots provided an additional layer of insight. Adam demonstrated very low and stable generation times with smooth loss descent. DE, on the other hand, exhibited very long generation times (especially on the `mouse` dataset), consistent with the computational intensity of its population-based evaluations. CMA-ES was more efficient time-wise, but its learning trajectory was more irregular and showed limited progress after early generations.

Final conclusions. The experiments indicate that Adam remains the most suitable optimizer among the three for neural network regression in this domain, particularly in scenarios where gradient information is available. DE may be considered as a viable alternative when gradients are unavailable or unreliable, but its efficiency must be improved to be practical. The (1+1)-CMA-ES algorithm showed potential in terms of runtime and resource efficiency, but its regression performance was insufficient in the current configuration. Further improvement might involve hybrid approaches, more advanced adaptation techniques, or algorithmic modifications tailored to the high-dimensional optimization setting present here.

References

[1] PyThorch documentation - SmoothL1Loss. https://docs.pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html

[2] Yohei Watanabe, Kento Uchida, Ryoki Hamano, Shota Saito, Masahiro Nomura, Shinichi Shirakawa: (1+1)-CMA-ES with Margin for Discrete and Mixed-Integer Problems, (2023). https://arxiv.org/abs/2305.00849

[3] Suttorp, T., Hansen, N., Igel, C.: Efficient covariance matrix update for variable metric evolution strategies (2009) https://doi.org/10.1007/s10994-009-5102-1