Project Report for Consensus-Based Optimization with Exponential Noise

Massimo Fornasier*[1,2], Philip Hierhager†[3], Konstantin Riedl‡[1,2] and Tim Roith§[4]

[1] *Technical University of Munich, School of Computation, Information and Technology, Department of Mathematics, Munich, Germany*
[2] *Munich Center for Machine Learning, Munich, Germany*
[3] *Technical University of Munich, School of Computation, Information and Technology, Department of Computer Science, Munich, Germany*
[4] *DESY, Hamburg, Germany*

Contents

1 Background and Derivation **2**
1.1 Machine Learning, Neural Networks and Training . . . 2
1.2 Particle Swarm Optimization (PSO) . . . 2
1.3 Quantum-Inspired PSO . . . 3
1.4 Consensus-Based Optimization (CBO) . . . 3
1.5 CBO with exponential noise . . . 4

2 Integration into CBXpy and runner implementation **6**

3 Numerical Experiments **7**
3.1 Mean-Field behaviour . . . 7
3.2 Hyperparameter Performance with different Dimensionality . . . 8
3.3 Hyperparameter Performance with different Alpha values . . . 8
3.4 Hyperparameter Performance with different number of particles . . . 8
3.5 Real-world Machine Learning Applications . . . 9

4 Interpretation and Discussion **11**

*Email: massimo.fornasier@ma.tum.de
†Email: philip.hierhager@tum.de
‡Email: konstantin.riedl@ma.tum.de
§Email: tim.roith@desy.de

1 Background and Derivation

1.1 Machine Learning, Neural Networks and Training

Machine Learning uses data observations to generalize in the presence of uncertainty. Instead of using fixed basis functions for the introduction of non-linearity, Neural Networks fix the number of basis functions in advance but allow them to be adaptive. To learn its parameters v, minimizing an appropriate loss function $\mathcal{E}_i(v)$ averaged over all data samples M is required. This leads to the optimization problem

$$x^* := \arg\min_{v \in \mathbb{R}^d} \mathcal{E}(v) \quad \text{with} \quad \mathcal{E}(v) = \frac{1}{M} \sum_{j=1}^{M} \mathcal{E}_j(v), \tag{1}$$

which is referred to as training in machine learning. Unlike algorithms like the support vector machine where the basis functions are fixed, optimization problems of Neural Networks are high-dimensional, non-convex and might lean to being NP-hard in general. Local optimization algorithms like Gradient Descent and in particular its stochastic variants work surprisingly well. They often reach very good (in the sense that they generalize well) local minima compared with the global optimum. There are however numerous other global optimization methods that are investigated for the training of neural networks.

1.2 Particle Swarm Optimization (PSO)

One of those algorithms is Particle Swarm Optimization (PSO) which was initially introduced by Kennedy and Eberhart [8]. It solves (1) by considering N particles described by triplets (X_k^i, V_k^i, P_k^i) with k denoting the iteration number and i the particle number. $X_k^i \in \mathbb{R}^d$ and $V_k^i \in \mathbb{R}^d$ describe the position and velocity, and $P_k^i \in \mathbb{R}^d$ the local best position of particle i at iteration k. $G_k \in \mathbb{R}^d$ with $G_k = \arg\min_i \mathcal{E}(P_k^i)$ denotes the global best up to iteration k. The particle velocity and position are updated according to

$$\begin{aligned} V_{k+1}^i &= V_k^i + c_1 r_k^i (P_k^i - X_k^i) + c_2 R_k^i (G_k - X_k^i) \qquad (2)\\ &= V_k^i + \gamma_k^i + \eta_k^i \\ X_{k+1}^i &= X_k^i + V_k^i, \qquad (3) \end{aligned}$$

where

$$\begin{aligned} \gamma_k^i &= \frac{c_1}{2}(P_k^i - X_k^i) + \frac{c_2}{2}(G_k - X_k^i) \\ \eta_k^i &= \frac{c_1}{2}\tilde{r}_k^i (P_k^i - X_k^i) + \frac{c_2}{2}\tilde{R}_k^i (G_k - X_k^i) \end{aligned}$$

with the deterministic contribution γ_k^i and the stochastic one η_k^i. c_1 and c_2 are known as acceleration coefficients, r_k^i and R_k^i are two different random variables distributed uniformly with $U(0,1)$, $\tilde{r}_k^i$ and $\tilde{R}_k^i$ are then distributed uniformly with $U(-\frac{1}{2}, \frac{1}{2})$, and the vector G_k describes the position of the best particle of the swarm. Every particle is dragged towards its own historical best position and the global best position communicated in the swarm - formalized as a local attractor p_k^i the particle converges to

$$p_k^i = \frac{c_1 r_k^i P_k^i + c_2 R_k^i G_k}{c_1 r_k^i + c_2 R_k^i}$$

or

$$p_k^i = \psi_k^i P_k^i + (1 - \psi_k^i) G_k, \quad \text{where} \quad \psi_k^i = \frac{c_1 r_k^i}{c_1 r_k^i + c_2 R_k^i}$$

Often in PSO, $c_1 = c_2$ and this equation simplifies to

$$p_k^i = \psi_k^i P_k^i + (1 - \psi_k^i) G_k, \quad \text{where} \quad \psi_k^i \sim U(0,1). \tag{4}$$

1.3 Quantum-Inspired PSO

The Quantum-Inspired PSO was first introduced by Sun et al. [10] and Yang et al. [11] and was improved in several works since then in e.g. [9] or [3]. The local attractor of particles in PSO can be thought of as a particles in a Newtonian attractor potential field where particle i moves toward the local attractor p_k^i with its potential energy declining to zero like a returning satellite orbiting the earth. We can now extend this imagery to the quantum mechanical case where the particles have the same attractor, but are described by quantum-mechanical bound-state equations. We use the Schrödinger equation at an infinitely high potential well — an often used simple approximation for quantum mechanics in physics

$$i\frac{\partial}{\partial t}\Psi(X,t) = \hat{H}\Psi(X,t)$$
$$\text{with} \quad \hat{H} = -\frac{\hbar}{2m}\nabla^2 + V(\hat{x})$$

with mass m, potential field $V(x)$ and Hamiltonian $\hat{H}$. We start by defining the one-dimensional case and denote the position of the particle as X, the attractor as p and the variable $Y = X - p$. With p being the center of an infinitely high potential wall and L its characteristic length, the position of the particle can be described with the quantum-mechanical one-dimensional wave function

$$\Psi(Y) = \frac{1}{\sqrt{L}} \exp^{-\frac{|Y|}{L}}. \tag{5}$$

In quantum mechanics, the probability density function of a particle is given by $Q(Y) = |\Psi(Y)|^2$. To sample from this in a quantum mechanical system, we just observe the particle which makes its wave function to collapse to one position. By using its cumulative distribution function $F(Y) = 1 - \exp(-|Y|/L)$ and the Markov Inverse sampling we can also simulate this classically with

$$Y = \pm\frac{L}{2}\ln\frac{1}{u} \quad \text{with} \quad u \sim U(0,1) \tag{6}$$

which can be straightforwardly extended with the local attractor p_k^i and with $X_k^i = Y_k^i + p_k^i$ to the d-dimensional update rule of particle i at iteration step $k+1$,

$$X_{k+1}^i = p_k^i \pm \frac{L_k^i}{2}\ln\frac{1}{u_k^i} \quad \text{with} \quad u_k^i \sim U(0,1) \tag{7}$$

The characteristic length of the potential wall L_k^i can now either be modelled as

$$L_k^i = 2\alpha|X_k^i - p_k^i| \tag{8}$$

or using the mean best $C_k = \frac{1}{N}\sum_{i=1}^N P_k^i$ as

$$L_k^i = 2\alpha|X_k^i - C_k|.$$

By iteratively updating the positions of the particles after (7) with the local attractor (4), evaluating their fitness scores and updating personal, global and potentially mean best a competitive optimizer for non-convex loss landscapes can be designed.

1.4 Consensus-Based Optimization (CBO)

CBO uses a finite number of interacting agents $X^1, \ldots, X^N$ to explore the domain with a drift in the direction of the consensus and a stochastic diffusion term, and to form a global consensus about the minimizer x^* as time passes [2, 4, 6, 7]. In contrast to PSO, CBO is easy enough to be theoretically analyzable on the one hand but sophisticated enough to be competitive in applications

on the other [1, 5]. After initializing the system with independent initial data $(X_0^i \sim \rho_0)_{i=1,\dots,N}$ for a suitable measure ρ_0 (typically a multi-variate normal distribution with certain mean and covariance matrix), the dynamics of each individual particle X^i at time step k can be formally described as follows.

Given a time horizon $T > 0$ and a time discretization $t_0 = 0 < \Delta t < \cdots < K\Delta t = T$ of $[0, T]$, we denote the location of agent i at time $k\Delta t$ by $X^i_{k\Delta t}$, $k = 0, \dots, K$. For user-specified parameters $\alpha, \lambda, \sigma > 0$, the time-discrete evolution of the i-th agent is defined by the update rule

$$\begin{aligned} X^i_{(k+1)\Delta t} =& X^i_{k\Delta t} - \Delta t\lambda \left(X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right) H\big(\mathcal{E}(X^i_{k\Delta t}) - \mathcal{E}\big(x_\alpha(\widehat{\rho}^N_{k\Delta t})\big)\big) \\ &+ \sigma \left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2 B^i_{k\Delta t}, \end{aligned} \tag{9}$$

$$X^i_0 \sim \rho_0 \quad \text{for all } i = 1, \dots, N, \tag{10}$$

where $((B^i_{k\Delta t})_{k=0,\dots,K-1})_{i=1,\dots,N}$ are independent, identically distributed Gaussian random vectors in $\mathbb{R}^d$ with zero mean and covariance matrix $\Delta t \mathsf{Id}_d$ [5].

The dynamics (9) of each particle is governed by two competing terms. Firstly, a drift term

$$\Delta t\lambda \left(X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right) H\big(\mathcal{E}(X^i_{k\Delta t}) - \mathcal{E}\big(x_\alpha(\widehat{\rho}^N_{k\Delta t})\big)\big) \tag{11}$$

drags the respective agent towards a momentaneous consensus point $x_\alpha(\widehat{\rho}^N_{k\Delta t})$, which is computed as a weighted average of the agents' positions given by

$$x_\alpha(\widehat{\rho}^N_{k\Delta t}) := \sum_{i=1}^N X^i_{k\Delta t} \frac{\omega_\alpha(X^i_{k\Delta t})}{\sum_{j=1}^N \omega_\alpha(X^i_{k\Delta t})}, \quad \text{with} \quad \omega_\alpha(x) := \exp(-\alpha\mathcal{E}(x)). \tag{12}$$

It serves as a guess for the global minimizer and is motivated by the fact that

$$x_\alpha(\widehat{\rho}^N_{k\Delta t}) \approx X^g_{k\Delta t} \quad \text{with} \quad g = \operatorname*{arg\,min}_{i=1,\dots,N} \mathcal{E}(X^i_k) \quad \text{for} \quad \alpha \gg 1,$$

i.e., $x_\alpha(\widehat{\rho}^N_{k\Delta t})$ is close to the current best position among the particles. Thus it ensures that the particles are approximately moving into the correct direction, namely the direction of the minimizer x^*. The univariate function $H : \mathbb{R} \to [0, 1]$ is often taken to be $H \equiv 1$, but can be also used to deactivate the drift term for agents whose objective is better than the momentanous consensus, so $H(x) \approx \mathbb{1}_{x \geq 0}$ with $x := \mathcal{E}(X^i_t) < \mathcal{E}(x_\alpha(\widehat{\rho}^N_t))$. For the development of the CBO algorithm with exponential noise we will assume $H \equiv 1$. Secondly, a diffusion term, given by

$$\sigma \left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2 B^i_{k\Delta t} \tag{13}$$

randomly moves particles to feature the exploration of the energy landscape of the objective function $\mathcal{E}$. The further away the particles are from the common consensus $x_\alpha(\widehat{\rho}^N_{k\Delta t})$, the greater the noise scaling factor $\left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2$.

Ideally, through the drift-diffusion mechanism of the CBO the agents reach a near optimal global consensus, so that the associated empirical measure

$$\widehat{\rho}^N_t := \frac{1}{N} \sum_{i=1}^N \delta_{X^i_t} \tag{14}$$

converges to a Dirac delta $\delta_{\widetilde{v}}$ at some $\widetilde{v} \in \mathbb{R}^d$ close to x^*.

1.5 CBO with exponential noise

Like Particle Swarm Optimization is changed by using inspiration from quantum mechanics, this document describes how to extend consensus-based optimization being inspired by quantum mechanics. In equation (8), the local attraction point of the particle swarm algorithm is used to determine

the length of the imagined potential wall and therefore the strength of the noise for every particle as described in equations (6) and (7). The noise is then added to the local attraction point to create a quantum-inspired version of the Particle Swarm optimization. For the Consensus-based optimization we take the same approach. We use the attraction point that is given by equation (12) and employ a quantum-mechanical bound state as in equation (5) using this new point. The stochastic dependency of CBO to the distance can be integrated then via the modelling of the characteristic length of the well. Furthermore, as the attraction point is now identical for all particles, a convex combination of the particle position and the actual quantum-inspired term is taken.

This leads to

$$\begin{aligned} X^i_{(k+1)\Delta t} &= (1-\beta)X^i_{k\Delta t} + \beta\left(x_\alpha(\widehat{\rho}^N_{k\Delta t}) \pm \frac{L^i_{k\Delta t}}{2}\ln\frac{1}{u^i_{k\Delta t}}\right) \\ &= X^i_{k\Delta t} - \beta(X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})) + \frac{\beta}{2}L^i_{k\Delta t}\left(\pm\ln\frac{1}{u^i_{k\Delta t}}\right) \\ &= X^i_{k\Delta t} - \lambda\Delta t\left(X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right) + \sigma_{\Delta t}\left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2 \hat{B}^i_{k\Delta t} \end{aligned} \tag{15}$$

with

$$B^i_{k\Delta t} = (\pm\ln\frac{1}{u^i_{k\Delta t}}) \quad \text{with} \quad u^i_{k\Delta t} \sim U(0,1) \tag{16}$$

$$x_\alpha(\widehat{\rho}^N_{k\Delta t}) := \sum_{i=1}^N X^i_{k\Delta t}\frac{\omega_\alpha(X^i_{k\Delta t})}{\sum_{j=1}^N \omega_\alpha(X^j_{k\Delta t})}, \quad \text{with} \quad \omega_\alpha(x) := \exp(-\alpha\mathcal{E}(x)) \tag{17}$$

$$\sigma_{\Delta t} = \sigma\sqrt{\Delta t} \tag{18}$$

$$L^i_{k\Delta t} = \left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2 \tag{19}$$

$$\alpha, \lambda, \sigma \geq 0 \quad \text{and} \quad \beta \in (0,1)$$

with which we hope to minimize $\left\|x_\alpha(\widehat{\rho}^N_{K\Delta t}) - x^*\right\|_2$. As in standard CBO, the dynamics (15) of each particle is governed by two competing terms. Firstly, a drift term

$$\lambda\Delta t(X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})) \tag{20}$$

drags the respective agent towards a momentaneous consensus point $x_\alpha(\widehat{\rho}^N_{k\Delta t})$. This is as in the original CBO method. Secondly, a diffusion term, given by

$$\sigma\left\|X^i_{k\Delta t} - x_\alpha(\widehat{\rho}^N_{k\Delta t})\right\|_2 \hat{B}^i_{k\Delta t} \tag{21}$$

again randomly moves the particles to explore the energy landscape of the objective function $\mathcal{E}$. Compared to the standard CBO and its diffusion term (13), the CBO variant proposed herewith employs not a Gaussian but an exponential noise distribution $\hat{B}^i_{k\Delta t}$ in equation (16) with more stochastic mass on its tails leading to greater exploration, but less exploitation capabilities as can be seen in Figure 1)

Through the specific modelling of the characteristic length $L^i_{k\Delta t}$ in equation (19) we achieve the same scaling as in the standard CBO.

The update rule (15) together with exponential noise leads us to the following algorithm:

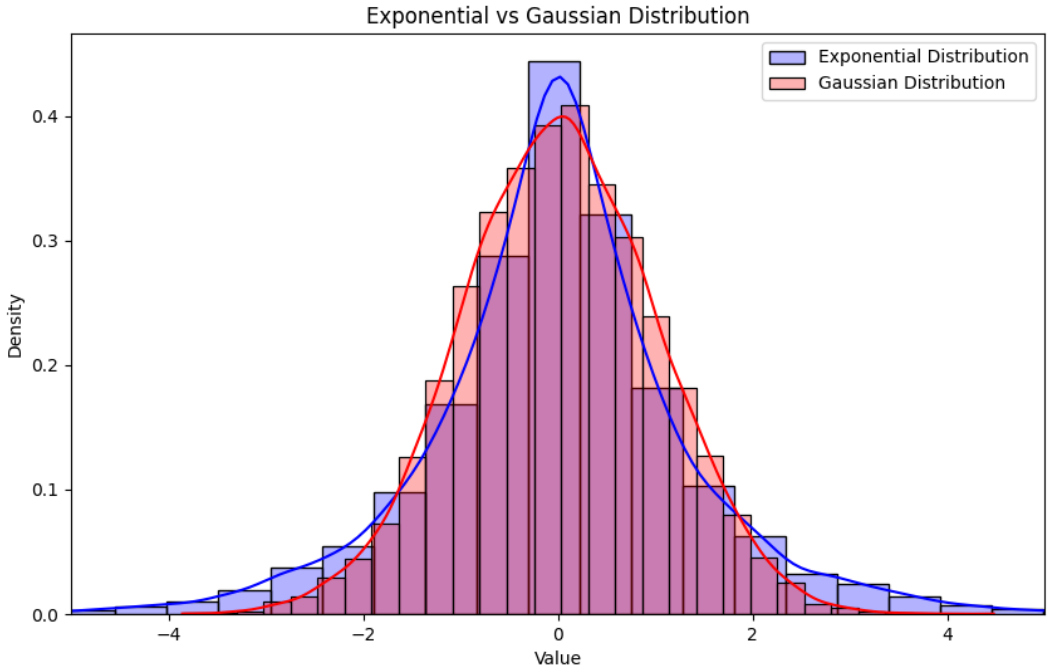


Figure 1: Comparison of the Gaussian and exponential distribution. The exponential function has a heavier tail and therefore greater exploration capability. The Gaussian function has greater exploitation capability as more probability mass is located around the mean.

Algorithm 1: Consensus-based Optimization Algorithm with Exponential Noise

Input: Population size N, time steps K, time step length Δt, consensus weight α, exploration parameter σ

Output: Near optimal solution

Initialize particles X_0^i randomly with a Gaussian distribution (10);
for $k \leftarrow 1$ **to** $K-1$ **do**
  for *particle* i **do**
    Evaluate fitness $\mathcal{E}(X_{k\Delta t}^i)$ of each particle;
    Calculate the attractor $x_\alpha(\widehat{\rho}_{k\Delta t}^N)$ with (17);
    Update position $X_{(k+1)\Delta t}^i$ with (15);
  end
end
Evaluate fitness $\mathcal{E}(X_{K\Delta t}^i)$ of each particle;
Output best solution;

2 Integration into CBXpy and runner implementation

The CBO algorithm with exponential noise is integrated into CBXpy by writing a custom noise sampler that can then be used in several Particle Dynamics tasks like CBO, but for example Consensus-based Sampling, PolarCBO, and others. The custom noise sampler is an exponential sampler with a random sign assigned to the output.

Code snippets of the integration can be found in Figure 2 showing the implemented code.
To run large-scale experiments, we also implemented an experiment runner suite and visualizations. Several code snippets of this runner can be found in Figures 9, 10, 11. In Figure 12 a key code snippet for the loading of the results is depicted and in Figure 13 one of the many visualization functions is depicted - here for heatmaps in subplots.

```python
def exponential_sampler():
    """The function returns the exponential sampler."""
    def _exponential_sampler(size=None):
        x = np.random.exponential(1.0, size)
        sign = np.random.choice([-1, 1], size)
        x *= sign
        return x
    return _exponential_sampler
```

(a) The implementation of the exponential noise sampler using numpy.

```python
@requires_torch
def exponential_torch(device):
    def _exponential_torch(size=None):
        x = torch.distributions.Exponential(1.0).sample(
            size
        )
        sign = torch.randint(0, 2, size) * 2 - 1
        x: torch.Tensor = x* sign
        return x.to(device)
    return _exponential_torch
```

(b) The implementation of the exponential noise sampler using torch.

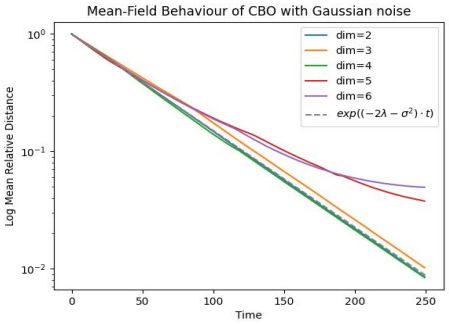
Figure 2: The sampler can be used to create a isotropic, anisotropic or covariance noise that behave differently in each dimension. Those noises can then be used to created different versions of all the particle dynamics algorithms like CBO, CBS, PolarCBO.

3 Numerical Experiments

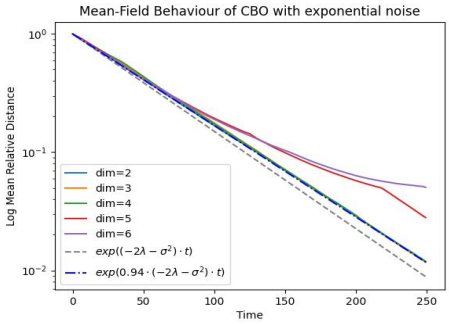
In this section, we present numerical experiments to compare the performance of the CBO algorithm with exponential noise with the standard CBO algorithm (with Gaussian noise). To run large-scale experiments, an experiment runner was written which hyperparameter settings could be set by using .yaml files. It automatically saves the results into an .csv file, that can then be used by various visualization functions to draw information from the results. In the experiments, mainly the optimization of the Rastrigin function is considered. Its parameter space is thoroughly tested to compare the CBO with Gaussian and exponential noise. The Rastrigin function is a non-convex function with many local minima and a global minimum at the origin. We compare the performance of the CBO algorithm with exponential noise and Gaussian noise in terms of the convergence rate and the quality of the solutions found. The success factor is defined as the percentage of runs in which the algorithm finds a function value below the second best minima.

3.1 Mean-Field behaviour

The first question that arises is how the algorithm behaves in the mean-field limit. Fornasier, Klock and Riedl explored in their paper "Convergence of Anisotropic Consensus-Based Optimization in Mean-Field Law" the convergence of the CBO method and found an exponential convergence in the mean-field. This rate is independent of the dimensionality of the problem. The result for the CBO with Gaussian noise could be confirmed with own experiments and can be seen in Figure 3. For CBO with exponential noise we take the same parameters and try to find a similar correlation. As it turns out, even with exponential noise, the mean-field convergence stays on its exponential trend with almost the same convergence rate of $\exp(-0.94(2\lambda - \sigma^2)t)$.



(a) The mean-field behaviour of CBO with Gaussian noise.



(b) The mean-field behaviour of CBO with exponential noise.

Figure 3: A depiction of the mean-field behaviours of CBO with Gaussian and exponential anisotropic noise. For the Rastrigin function $\mathcal{E}(v) = \sum_{k=1}^{d} v_k^2 + \frac{5}{2}(1 - \cos(2\pi v_k))$ with $x^* = 0$ and several local minima, we evolve the discretized system of isotropic and anisotropic CBO using $N = 10000$ particles, discrete time step size $\Delta t = 0.01$ and $\alpha = 10^{15}$, $\lambda = 1$, and $\sigma = 0.32$ for different dimensions $d \in \{2, 3, 4, 5, 6\}$. We observe in both cases that the convergence rate of the energy functional $\mathcal{V}(\widehat{\rho}_t^N)$ is independent from d and lies at around $(2\lambda - \sigma^2)$. However, the guess her sth is missing

3.2 Hyperparameter Performance with different Dimensionality

To thoroughly study the hyperparameter settings and differences for CBO with Gaussian and exponential noise several experiments with different lambda and sigma values are created. This subsection should give an insight into how success probability depends on their relationship of the method with different dimensions of the objective function. As you can see in the Figure 4, both CBO with Gaussian and exponential noise become very sensitive to the hyperparameter setting. It can be seen that CBO with exponential noise needs a not as high sigma as the CBO with Gaussian noise. That was as expected, as the exponential noise has more probability mass on its tails leading to greater exploration capabilities of the CBO algorithm.

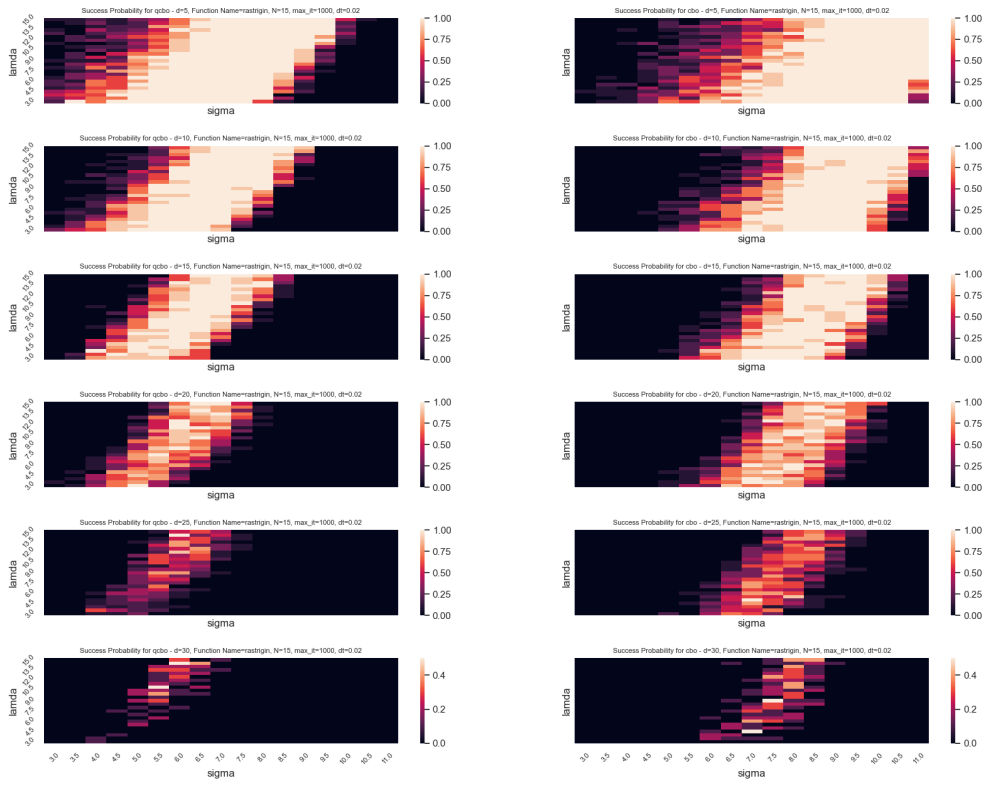


Figure 4: A demonstration of the increased sensitivity of hyperparameter settings with increased dimensionality. On the left side the CBO is depicted, on the right side the QCBO. Both are evaluated with $d = 5, 10, 15, 20, 25, 30$.

3.3 Hyperparameter Performance with different Alpha values

In this subsection the influence of the alpha values is investigated. The alpha value controls the weight of the consensus point in the CBO algorithm with exponential noise. It does not impact the hyperparameter sensitivity of neither the CBO with Gaussian noise nor the CBO with exponential noise as the Figure 5 shows.

3.4 Hyperparameter Performance with different number of particles

Another experiment that has been conducted is to map the relationship between the the number of particles, sigma and lambda. Plotting sigma on the x-axis and the number of particles on the y-axis, we recognize a probability distribution with quadratic contour arises when using the CBO with exponential noise. This probability distribution is then linearly shifted along the x-axis with changing

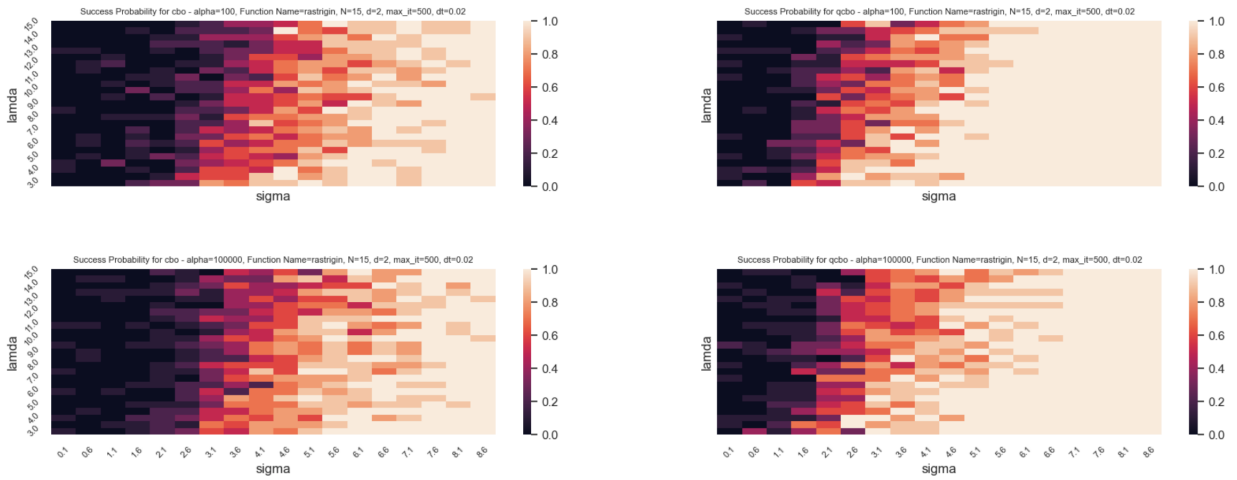


Figure 5: The Figure captures the differences in the success probability with different hyperparameter settings of λ and σ observed at different values of α. Both dynamics are evaluated at $alpha = 100, 10000$.

lambda values. The same happens with the CBO with Gaussian noise - even if the distribution contours seem far broader than the CBO exponential one. This can be seen in Figure 6 for CBO with Gaussian noise and in Figure 7 for CBO with exponential noise. The more narrow probability distribution than the CBO with Gaussian noise indicates a higher sensitivity to hyperparameters.

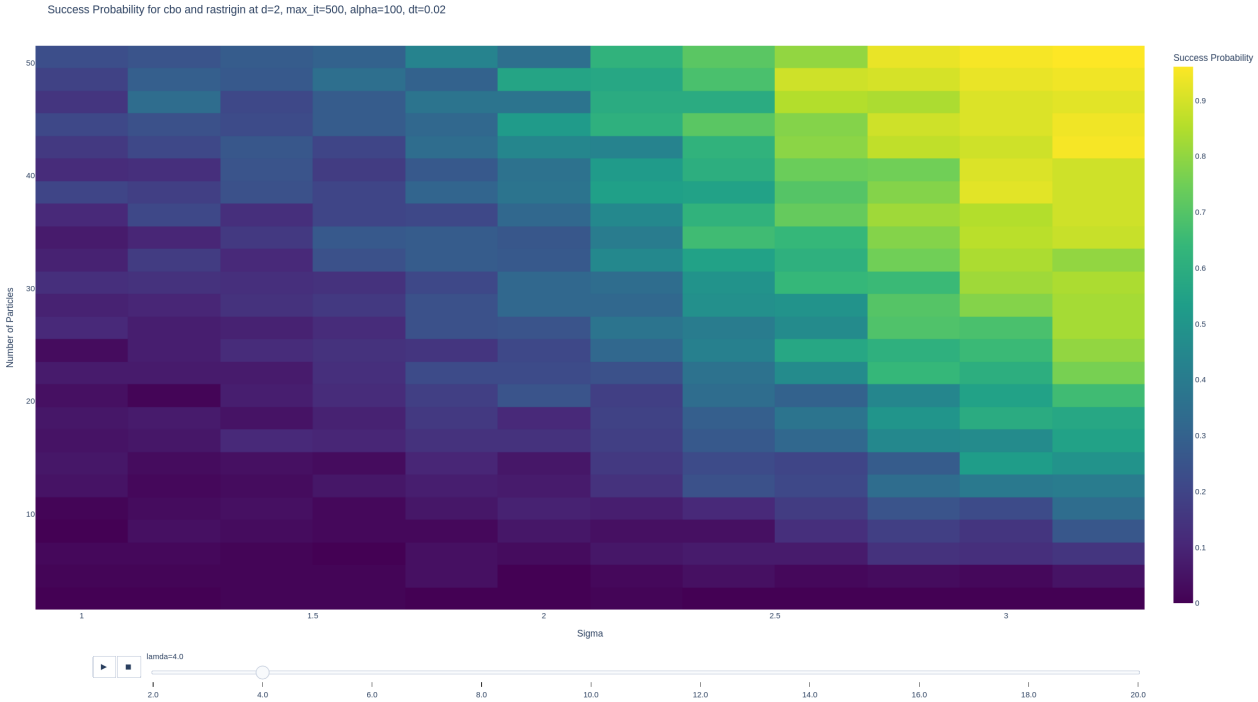


Figure 6: CBO with Gaussian noise: It is apparent that the higher the sigma and the higher the lambda, the higher the success probability

3.5 Real-world Machine Learning Applications

We see that CBO and QCBO behave fairly similar when optimizing the machine learning landscape of a neural network with one hidden layer for the MNIST dataset. The model used is a multi-layer perceptron with a hidden layer of 100 neurons, 1D batch normalization, and cross entropy loss. The experiment shows that the heavy-tailed noise does not have much influence on the exploitation capability needed in neural network optimization. The diagram showing the accuracy over epochs can be seen in Figure 8.

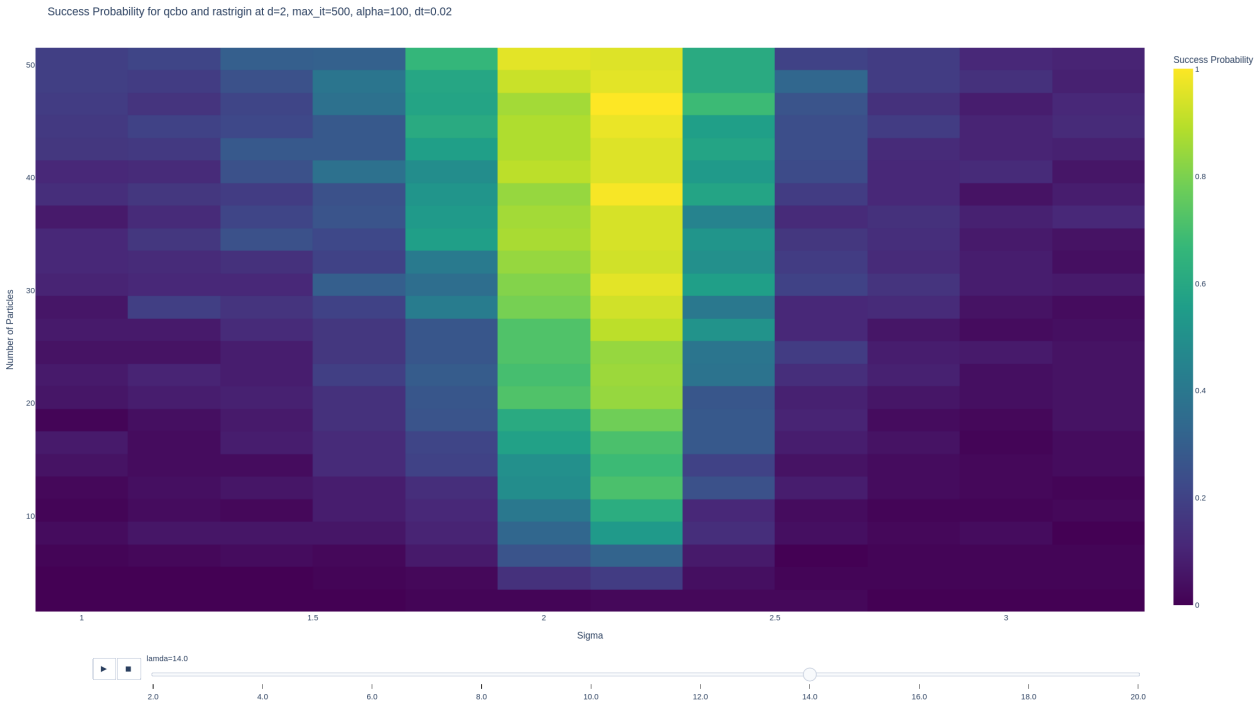


Figure 7: For CBO with exponential noise, we see a higher sensitivity to the hyperparameter settings than in the CBO with Gaussian noise.

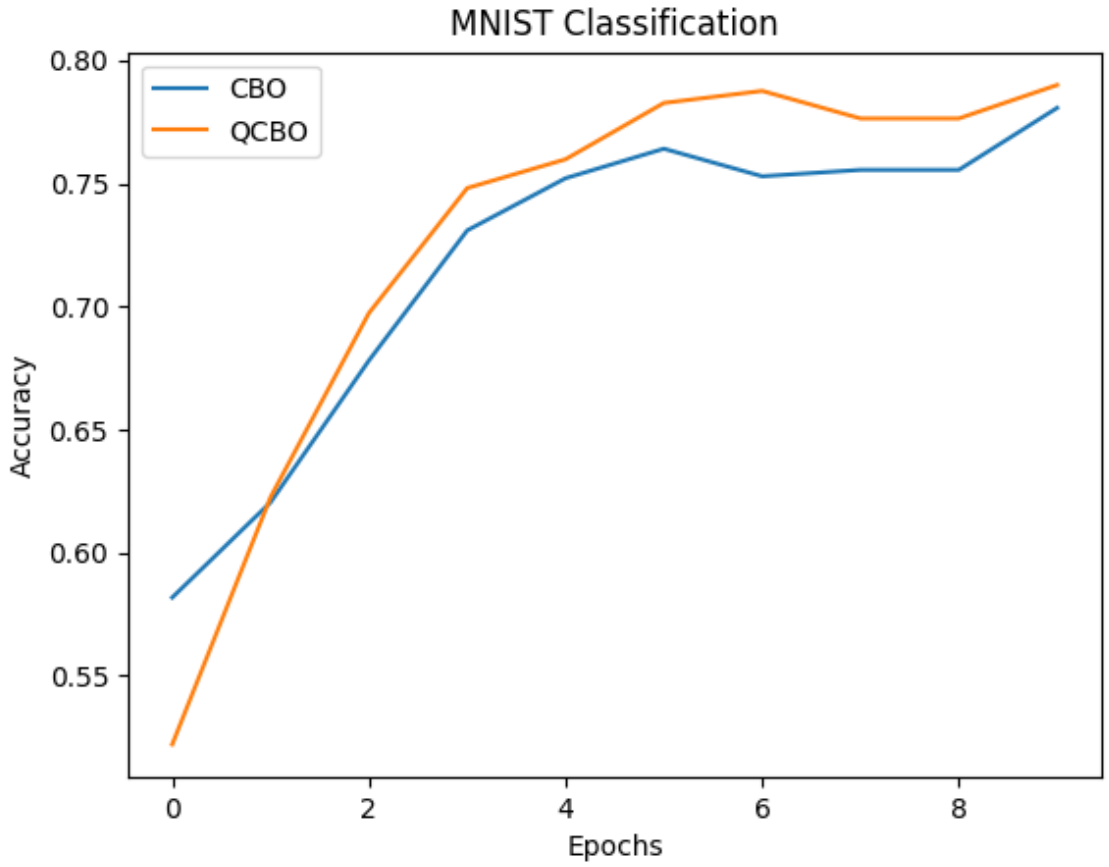


Figure 8: It can be observed that CBO with exponential noise works similar on a neural network optimization as the one with Gaussian noise. For both, the same parameters were used: $\alpha = 50.0$, $\Delta t = 0.1$, $\sigma = 0.1$, $\lambda = 1.0$, $max\_it = 200$

4 Interpretation and Discussion

Integrating exponential noise into the CBO algorithm has shown similar results as the CBO algorithm with Gaussian noise. Our tests show that CBO with exponential noise maintains almost the same strong convergence rate as with Gaussian noise in the mean-field limit. This suggests that using exponential noise does not affect the ability of the algorithm to converge. One of the key takeaways is that CBO with exponential noise has similar behaviour during the optimization process. Even though the hyperparameter settings are naturally different, Gaussian and exponential noise sigma and lambda interactions take the same forms across different number of dimensions and alpha values.

In practical applications, such as optimizing neural networks, both noise models performed similarly. It shows that exponential noise does not hinder the algorithm's ability to exploit promising solutions but improves exploration without sacrificing accuracy.

In the future, it would be interesting to explore how exponential noise performs in more complex optimization problems and across different machine learning models.

References

[1] J. A. Carrillo, Y.-P. Choi, C. Totzeck, and O. Tse. An analytical framework for consensus-based global optimization method. *Math. Models Methods Appl. Sci.*, 28(6):1037–1066, 2018.

[2] J. A. Carrillo, S. Jin, L. Li, and Y. Zhu. A consensus-based global optimization method for high dimensional machine learning problems. *ESAIM Control Optim. Calc. Var.*, 27(suppl.):Paper No. S5, 22, 2021.

[3] A. Flori, H. Oulhadj, and P. Siarry. Quantum particle swarm optimization: An auto-adaptive pso for local and global optimization. *Computational Optimization and Applications*, 82(2):525–559, 2022.

[4] M. Fornasier, H. Huang, L. Pareschi, and P. Sünnen. Anisotropic diffusion in consensus-based optimization on the sphere. *arXiv preprint arXiv:2104.00420*, 2021.

[5] M. Fornasier, T. Klock, and K. Riedl. Consensus-based optimization methods converge globally in mean-field law. *arXiv preprint arXiv:2103.15130*, 2021.

[6] M. Fornasier, T. Klock, and K. Riedl. Convergence of anisotropic consensus-based optimization in mean-field law. *arXiv preprint arXiv:2111.08136*, 2021.

[7] M. Fornasier, L. Pareschi, H. Huang, and P. Sünnen. Consensus-based optimization on the sphere: Convergence to global minimizers and machine learning. *Journal of Machine Learning Research*, 22(237):1–55, 2021.

[8] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[9] J. Sun, W. Fang, X. Wu, V. Palade, and W. Xu. Quantum-behaved particle swarm optimization: analysis of individual particle behavior and parameter selection. *Evolutionary computation*, 20(3):349–393, 2012.

[10] J. Sun, B. Feng, and W. Xu. Particle swarm optimization with particles having quantum behavior. In *Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 325–331. IEEE, 2004.

[11] S. Yang, M. Wang, et al. A quantum particle swarm optimization. In *Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 320–324. IEEE, 2004.

Additional Figures

```python
class Runner:
    def __init__(
        self,
        experiment_config: ExperimentConfig,
        result_dir: str = "results",
    ) -> None:
        """Initialize the Runner.

        Args:
            experiment_config (ExperimentConfig): The experiment configuration.
            result_dir (str, optional): The directory to save the results. Defaults to "results".
        """
        self.experiment_config: ExperimentConfig = experiment_config
        self.experiment_result: ExperimentResult = ExperimentResult(
            experiment_config.experiment_name,
            experiment_config.config_opt,
            results_dynamic=[],
        )
        self.result_dir = result_dir
        self.local_minimum_functions = (
            {}
        )

    def run_experiment(self) -> ExperimentResult:
        """Run the experiment.

        Returns:
            ExperimentResult: The experiment result.
        """
        results_dynamic = self.run_dynamic_configs()
        self.experiment_result.results_dynamic = results_dynamic
        if self.result_dir is not None:
            if not os.path.exists(self.result_dir):
                os.makedirs(self.result_dir)
            self.experiment_result.to_csv(
                self.result_dir
                + "/"
                + "_".join(str(self.experiment_result.experiment_name).lower().split())
                + ".csv"
            )
        return self.experiment_result
```

Figure 9: The runner class initializes large-scale experiments for by loading an ExperimentConfig that was obtained from .yaml files. The function *run_experiment* is responsible for running all the different configurations and saving it as a ExperimentResult.

```python
def run_dynamic_configs(self) -> list[ResultDynamicRun]:
    """Run the dynamic configurations.

    Returns:
        list[ResultDynamicRun]: The results of the dynamic configurations.
    """
    results_dynamic = []
    opt_dict = self.get_opt_dict()
    for config_container_dynamic in tqdm(
        self.experiment_config.config_container_dynamic_gen
    ):
        if config_container_dynamic.f not in self.local_minimum_functions.keys():
            if str(config_container_dynamic.f) == "Ackley's function":
                self.local_minimum_functions["Ackley's function"] = 2.181
            elif str(config_container_dynamic.f) == "Rastrigin's function":
                self.local_minimum_functions["Rastrigin's function"] = 1
            elif str(config_container_dynamic.f) == "W function":
                self.local_minimum_functions["W function"] = (
                    0.1
                )
            else:
                raise NotImplementedError(
                    "The nearest local minimum for this function and dimension is not yet implemented."
                )
        result = self.run_dynamic_config(config_container_dynamic, opt_dict)
        results_dynamic.append(result)
    return results_dynamic
```

Figure 10: The *run_dynamic_configs* function runs all configurations and before includes the second-best minima objective energy values.

```python
def run_dynamic_config(
    self,
    config_container_dynamic: ConfigContainerDynamic,
    opt_dict: dict,
) -> ResultDynamicRun:
    """Run a dynamic configuration.

    Args:
        config_container_dynamic (ConfigContainerDynamic): The dynamic configuration.
        opt_dict (dict): The optimization dictionary.

    Returns:
        ResultDynamicRun: The result of the dynamic configuration.
    """
    f = config_container_dynamic.f
    config_dynamic = config_container_dynamic.config_dynamic
    dynamic = config_container_dynamic.dynamic(
        f, **config_dynamic
    )
    time, best_x = optimize_wrapper(dynamic, **opt_dict)
    best_f = f(best_x)
    success = np.where(
        best_f < self.local_minimum_functions[str(config_container_dynamic.f)], 1, 0
    )
    return ResultDynamicRun(
        name_dynamic=config_container_dynamic.name_dynamic,
        name_f=config_container_dynamic.name_f,
        index_config=config_container_dynamic.index_config,
        config_dynamic=config_dynamic,
        time=time,
        best_f=best_f,
        best_x=best_x,
        success=success,
    )
```

Figure 11: The *run_dynamic_config* runs one configuration and gives out the result.

```python
class ExperimentResult:

    @classmethod
    def from_csv(cls, filename: str) -> "ExperimentResult":
        """Create an ExperimentResult object from a CSV file.

        Args:
            filename (str): The filename of the CSV file.

        Returns:
            ExperimentResult: An ExperimentResult object.
        """
        csv.field_size_limit(1000000) # Increase field size limit to read large config_dynamic
        with open(filename, "r", newline="") as csvfile:
            reader = csv.reader(csvfile)
            next(reader)  # Skip header
            experiment_name = None
            config_opt = None
            results_dynamic = []
            for row in reader:
                if experiment_name is None:
                    experiment_name = row[0]
                    config_opt = {}  # Assuming config_opt is not stored in CSV
                result_dynamic = ResultDynamicRun(
                    name_dynamic=row[1],
                    name_f=row[2],
                    index_config=int(row[3]),
                    time=float(row[4]),
                    best_f=np.array(
                        eval(row[5])
                    ),  # Convert string representation of list to numpy array
                    best_x=np.array(
                        eval(row[6])
                    ),  # Convert string representation of list to numpy array
                    success=np.array(
                        eval(row[7])
                    ),  # Convert string representation of list to numpy array
                    config_dynamic=eval(
                        row[8]
                    ),  # Convert string representation of dict to dict
                )
                results_dynamic.append(result_dynamic)

            return cls(experiment_name, config_opt, results_dynamic)
```

Figure 12: This class is responsible for saving and loading results.

```python
def plot_heatmap_subplots(df: pd.DataFrame, row_name, col_name, x_name, y_name):
    # Calculate success probability
    df["Success Probability"] = df["Success"].apply(np.mean)
    melt_vars = [row_name, x_name, "Function Name", y_name]
    pivot_vars = [y_name, x_name]
    col_names = set(df[col_name])
    row_values = set(df[row_name])
    dfs = {}
    for c_name in col_names:
        row_df = {}
        for r_value in row_values:
            df_ = df[(df[col_name] == c_name) & (df[row_name] == r_value)]
            row_dfm = df_.melt(
                id_vars=melt_vars,
                value_vars=["Success Probability"],
                var_name="variable",
                value_name="success_prob",
            ).pivot(index=pivot_vars[0], columns=pivot_vars[1], values="success_prob")
            row_df[f"{r_value}"] = row_dfm
        dfs[c_name] = row_df

    # Plot the data side by side
    sns.set_theme(style="whitegrid")
    row_values = list(row_values)
    num_rows = len(row_values)
    num_cols = len(col_names)
    _, axes = plt.subplots(num_rows, num_cols, figsize=(
        20, 15), sharey=True, sharex=True)
    if num_rows == 1:
        axes = np.expand_dims(axes, axis=0)
    if num_cols == 1:
        axes = np.expand_dims(axes, axis=1)
    configs = df.iloc[0].to_dict()
    title_str = ", ".join([f"{k}={v}" for k, v in configs.items() if k not in [
        row_name, col_name, x_name, y_name] and k in ["d", "max_it", "dt", "Function Name", "lamda", "N"]])

    for i, col in enumerate(col_names):
        for j, row in enumerate(row_values):
            sns.heatmap(dfs[col][f"{row}"], ax=axes[j][i], cbar=True)
            axes[j][i].set_title(f"Success Probability for {
                col} - {row_name}={row}, " + title_str, fontsize=8)
            axes[j][i].invert_yaxis()

    plt.subplots_adjust(hspace=0.5)
    for ax in axes.flat:
        ax.tick_params(axis="y", labelsize=8, labelrotation=45)
        ax.tick_params(axis="x", labelsize=8, labelrotation=45)

    plt.savefig(
        f"results/images/{df.iloc[0].loc["Experiment Name"]}__plot_heatmap_subplots.png")
    plt.show()
```

Figure 13: This function generates heatmaps in a subplot.

```python
def _generate_configs_dynamics(config_dynamics: dict) -> dict:
    """Generate configurations for dynamics.

    Args:
        config_dynamics (dict): A dictionary containing the dynamics configurations.

    Returns:
        dict: A dictionary containing the configurations.
    """
    config = {}
    for cfg_name, value in config_dynamics.items():
        if isinstance(value, dict) and "range" in value:
            config[cfg_name] = range_generator(value)
        elif cfg_name == "name_f":  # dispatch and parse keyword all for function
            if value == "all":
                config[cfg_name] = [obj_name for obj_name in get_available_objectives()]
            elif isinstance(value, list):
                config[cfg_name] = value
            else:
                config[cfg_name] = value
        else:
            config[cfg_name] = value
    config_dynamic_generator = _dict_product_generator(config)
    return config_dynamic_generator

def generate_dynamics_product(
    experiment_config: dict[str, dict]
) -> Generator[ConfigContainerDynamic, Any, Any]:
    """Generate the product of dynamics configurations.

    Args:
        experiment_config (dict[str, dict]): A dictionary containing the experiment configurations.

    Yields:
        ConfigContainerDynamic: A container containing the dynamics configurations.
    """
    dynamics_name_and_cfg = _get_name_and_config_dynamics(experiment_config)
    for name_dynamic, _tmp_config_dynamic in dynamics_name_and_cfg:
        for i, configuration in enumerate(
            _generate_configs_dynamics(_tmp_config_dynamic)
        ):
            name_f = configuration["name_f"]
            f = dispatch_objective(name_f)
            configuration.pop("name_f", None)
            dynamic = dispatch_dynamics(name_dynamic)
            yield ConfigContainerDynamic(
                name_dynamic=name_dynamic,
                name_f=name_f,
                dynamic=dynamic,
                f=f,
                index_config=i,
                config_dynamic=configuration,
            )
```

Figure 14: Those two functions are the main responsible ones for converting a .yaml file into a generator of experiment configurations.