A PARTY-BASED SPARSE GAUSSIAN PROCESS OPTIMIZER

Chandrajit Bajaj
Oden Institute
University of Texas at Austin
bajaj@cs.utexas.edu

Omathar Bharat Vaidya
Oden Institute
University of Texas at Austin
omatharv.vaidya@austin.utexas.edu

Yi Wang
Oden Institute
University of Texas at Austin
panzer.wy@utexas.edu

ABSTRACT

Task learning in neural networks typically requires finding a globally optimal minimizer to a loss function objective. Conventional designs of swarm based optimization methods apply a fixed update rule, with possibly an adaptive step-size for gradient descent based optimization. While these methods gain huge success in solving different optimization problems, there are some cases where these schemes are either inefficient or suffering from local-minimum. We present a new particle-swarm-based framework utilizing Gaussian Process Regression to learn the underlying dynamical process of descent. The biggest advantage of this approach is greater exploration around the current state before deciding a descent direction. Empirical results show our approach can escape from the local minima compare with the widely-used state-of-the-art optimizers when solving non-convex optimization problems. We also test our approach under high-dimensional parameter space case, namely, image classification task.

Keywords Particle Swarm-based techniques · Gaussian Process Regression · Dynamical system · Gradient Descent.

1 Introduction

Gradient Descent (GD), a discrete-time iterative scheme, was first introduced in [1] to solve unconstrained optimization problem. Since then, numerous schemes (using the core principle of gradient descent) have been invented, studied theoretically, and analyzed. For instance, Stochastic Gradient Descent (SGD), proposed by [13], quickly became one of the most essential optimization algorithms with the rise of machine learning. Some popular methods developed afterwards include momentum method [11], AdaGrad [4], AdaDelta [19], RMS-Prop [5], Adam [7], Nadam [3], AMSGrad [12], etc.

If $\theta$ represents the parameters of the deep network $\mathcal{M}(\theta)$, the goal of a general task-learning problem is to learn a task $T$ under the unconstrained optimization setting. The task-learning problem can be formulated as optimizing a loss function $\mathcal{L}(\theta)$ that best represents the task and finding the best parameter $\theta^{best}$ via the stochastic optimization problem:

$$\theta^{best} = \min\theta \mathcal{L}(\theta) = \min\theta E_{\theta} [\mathcal{L}(\theta, w)],$$

(1)

where, the variable $w$ represents a random variable whose samples can be observed or generated [9]. This is also called as empirical risk minimization. In this research, as an example, we will consider the case of supervised machine learning for solving a classification problem in images. The model $\mathcal{M}$ represented by the parameters $\theta$, is a mapping from features to labels. In this case, $w$ represents the sampling of features and labels from the data distribution. A simple way to solve optimization problem (1) is to use sample-average approximation, i.e. given $N$ i.i.d. samples $\{w^j\}_{j=1}^N$ of the features and labels, the optimization problem is framed as an empirical approximation of (1):

$$\theta^{best} = \min\theta \frac{1}{N} \sum_{i=1}^N \mathcal{L}(\theta, w^i).$$

(2)

An alternative way is stochastic approximation. SGD [13] utilizes point-wise stochastic estimates of gradients of the cost function via samples of the data distribution for solving (1):

$$\theta_{t+1} = \theta_t + \alpha_t \nabla_{\theta} \mathcal{L}(\theta_t, w_t).$$

(3)
where, \( \alpha_t \) is the step-size and the gradient \( \mathcal{L}(\theta_t, w_t) \) is evaluated at a single-sample, making it fairly cheap. A variant of SGD is batch gradient descent:

\[
\theta_{t+1} = \theta_t - \frac{\alpha_t}{|S_k|} \sum_{j \in S_k} \nabla_\theta \mathcal{L}(\theta_t, w_j)
\]  

(4)

where, \( S_k \) is a mini-batch sample from \( \{1, 2, 3, \ldots, N\} \) of size \( |S_k| \), which considers mini-batch samples rather than one sample in SGD [5, 14].

Our goal is to develop an optimization algorithm that better the current state-of-the-art algorithms for task-learning, and empirically showing these results for the classification task. We note that one of the biggest disadvantages of the widely used methods is that they compute the gradients at a single point, which is usually the state of the network parameters at that moment, and use this information for deciding the descent direction. One of the biggest disadvantages of this approach is that there isn’t sufficient exploration around this point, which a few particle-swarm optimization methods have shown. We take an inspiration from particle-swarm optimization to utilize particles for sampling more than one gradients and then use this information in a productive way to model the gradient descent dynamics. A naive approach would be to initialize particles every iteration using a normal distribution around the current parameters’ state, sample gradients at these particle locations and to take an average of these gradients as the approximate gradient for the network parameters. While this technique ensures exploration, it’s clear that it won’t be accurate in case of a tough parameter terrain. Hence, we propose a novel idea of using Gaussian Process Regression [17] to effectively model the pattern shown by these gradients. This approach ensures approximate gradients and sufficient exploration to show the direction of descent towards the global minima instead of being stuck in a local minima. If the optimizer is stuck in a narrow local minima, there is a chance of having few particles discovering a direction of getting out resulting in a likely overall prediction by the Gaussian Process model to follow that direction. We call our technique ParticleGP.

We demonstrate the effectiveness of our algorithm by empirical results on widely-known quadratic non-convex optimization problems and on imaging-based classification tasks. The paper is arranged as follows: Section 2 discusses related work in particle-based methods and gradient-descent; Section 3 specifies our approach; Section 4 demonstrates the results of comparison of ParticleGP with other state-of-the-art optimizers; Finally, Section 5 discusses the limitations and opportunities of improvement.

## 2 Related Work

### 2.1 GD-based Methods

GD-based methods adopt a calculation of parameter update \( \delta_t \) using gradient information at each time step, i.e., if \( \theta_t \) are the network parameters, they are updated as:

\[
\theta_t = \theta_{t-1} + \delta_t
\]  

(5)

Prior techniques in optimization provide a mechanism to compute the best \( \delta_t \) for fastest convergence to a global minima. GD simply follows the gradient and is scaled by learning rate. Two common tools to improve GD are the sum of gradients (called as the **first moment**) and the sum of the gradients squared (called as the **second moment**). Momentum [11] uses the first moment with a decay rate to gain speed, whereas AdaGrad [4] uses the second moment with no decay to deal with sparse features. RMSProp [5] uses the second moment with a decay rate to improve its rate of convergence over AdaGrad. The fairly popular Adam [7] uses both first and second moments, and is generally regarded as the best choice. Nadam [3] utilizes Nesterov-acceleration over the Adam scheme.

### 2.2 Particle-based Methods

Compared to GD-based methods, particle-swarm based methods rely on functional evaluations rather than gradient computations for deciding the best direction of descent. Several previous research papers have developed Particle Swarm techniques for optimization purposes. One of the original works was by [6], which introduced Particle Swarm Optimization (PSO). Each particle in the swarm has information about: the best location that it has visited, called \( \hat{\theta}_{best} \) and the best location that any particle has visited overall, called the \( \hat{\theta}_{best} \). Every particle follows the following dynamical system, wherein it’s position and velocity are updated:

\[
\begin{align*}
\delta_{t+1} &= w \delta_t + c_1 r_1 (\hat{\theta}_{best} - \theta_t) + c_2 r_2 (\hat{\theta}_{best} - \theta_t) \\
\theta_{t+1} &= \theta_t + \delta_{t+1}
\end{align*}
\]  

(6)

(7)

The position of the particles directly corresponds to the value of parameters i.e. dimension of parameter space is equal to dimension of swarm. Let position of the \( i^{th} \) particle at time step \( t \) be given by \( \theta^i_t \). We have: \( \theta^i_{best} = \arg \min \mathcal{L}(\theta^i_t) \)
and $\hat{\theta}_{best} = \arg\min_{\theta} \mathcal{L}(\theta^i)$, where $\mathcal{L}$ is the Lagrangian. The vectors $\delta$ and $\theta$ represent the velocity and position of the particle respectively, $w$ is the inertia term, $c_1$ and $c_2$ are the relative weights given to cognitive learning and social learning respectively, whereas $r_1$ and $r_2$ are random points drawn from the uniform probability distribution $U(0, 1)$ as the damping term. After several iterations, the swarm collectively moves towards the minima as required. Since then, a lot of papers have focused on swarm intelligence. [18] created a slight modification in the iteration scheme for the velocity by adding a momentum term:

$$\delta_{t+1} = (1 - \lambda)\delta_t + c_1 r_1 (\theta_{best} - \theta_t) + c_2 r_2 (\hat{\theta}_{best} - \theta_t)$$

(8)

where, $\lambda$ denotes the momentum factor. This update helped improve performance by giving a weight to past velocities as well, to relieve excessive oscillation. EM-PSO [10] uses an additional momentum term $M$ where, $M_i = \beta M_i + (1 - \beta)\delta_t$

(9)

$$\theta_{t+1} = \theta_t + \delta_{t+1}$$

(10)

This adds flexibility to the task of exploration better than M-PSO and ensures faster convergence. One of the key aspects that the above PSO techniques had lacked was convergence to global optimum. [16] uses a single Hamiltonian Monte Carlo (HMC) particle for effectively searching the optimization space and ensures convergence to global optimum while retaining the benefits of EM-PSO by using $N$ EM-PSO particles. The table 1 summarizes different single-agent prior works; whereas the table 2 summarizes different particle-swarm based optimizers for single-task learning.

| Name       | Scheme to compute $\delta_t$                                           |
|------------|------------------------------------------------------------------------|
| GD [1]     | $\delta_t = -\alpha_t \cdot \nabla_\theta \mathcal{L}(\theta_{t-1})$ |
| Momentum [11] | $\delta_t = -r_t \cdot \nabla_\theta \mathcal{L}(\theta_{t-1}) + \delta_{t-1} \cdot \beta_1$ |
| AdaGrad [4] | $v_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1})^2 + v_{t-2}$   |
|            | $\delta_t = -r_t \cdot \frac{\nabla_\theta \mathcal{L}(\theta_{t-1})}{\sqrt{v_{t-1}}}$ |
| RMSProp [5] | $v_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1})^2 \cdot (1 - \beta_2) + v_{t-2} \cdot \beta_2$ |
|            | $\delta_t = -r_t \cdot \nabla_\theta \mathcal{L}(\theta_{t-1})$ |
| Adam [7]   | $m_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1}) \cdot (1 - \beta_1) + m_{t-2} \cdot \beta_1$ |
|            | $v_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1})^2 \cdot (1 - \beta_2) + v_{t-2} \cdot \beta_2$ |
|            | $\bar{m}_{t-1} = \frac{m_{t-1}}{1 - \beta_1}$ |
|            | $\bar{v}_{t-1} = \frac{v_{t-1}}{1 - \beta_2}$ |
|            | $\delta_t = -r_t \cdot \frac{\bar{m}_{t-1}}{\sqrt{\bar{v}_{t-1}}}$ |
| Nadam [3]  | $m_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1}) \cdot (1 - \mu_{t-1}) + m_{t-2} \cdot \mu_{t-1}$ |
|            | $v_{t-1} = \nabla_\theta \mathcal{L}(\theta_{t-1})^2 \cdot (1 - \nu) + v_{t-2} \cdot \nu$ |
|            | $\bar{m}_{t-1} = \mu_t \cdot \frac{m_{t-1}}{1 - \prod_{i=1}^t \mu_i} + (1 - \mu_t) \cdot \frac{\nabla_\theta \mathcal{L}(\theta_{t-1})}{1 - \prod_{i=1}^t \mu_i}$ |
|            | $\bar{v}_{t-1} = \nu \cdot \frac{v_{t-1}}{1 - \rho_{t-1}}$ |
|            | $\delta_t = -r_t \cdot \frac{\bar{m}_{t-1}}{\sqrt{\bar{v}_{t-1}}}$ |

Table 1: List of single-agent prior optimizers

We note that optimizer methods in table 1 evaluate the direction of descent by doing gradient evaluations of the cost function $\mathcal{L}$ at different points. Meanwhile, for the optimizers in table 2, the best direction to explore state space and exploit towards the minima is captured stochastically via an estimate of the distribution of functional evaluations. For
instance, the terms \((\theta_{best} - \theta_{t-1})\) and \((\theta_{best} - \theta_{t-1})\) in PSO-based optimizers act as an equivalent to gradients in standard optimizers since they estimate the direction of convergence. Whereas, the exponential momentum in EM-PSO stores and utilizes prior-descent directions similar to Adam and RMS-Prop’s iteration schemes. Thus, what a single-agent does in standard optimizers is built intrinsically in swarm intelligence.

While it’s clear that there is a deeper connection between swarm-based methods and single-agent optimizers, there is a crucial aspect lacking in both. As discussed in section 1, one of the primary advantages of using such a technique is that the optimizer can process essential information about the geometry of the parameter space from its local neighbourhood and make an accurate prediction on the objectively best direction to take a descent step. The current state-of-the-art optimizers utilize direct or stochastic gradients at the given point, which while it helps taking the locally best decision, does not paint the entire picture. Our proposed ParticleGP is the most effective method in optimization problems with several local minima and non-convex behaviour: a scenario in which most state-of-the-art optimizers fail to reach the global optima. The setup for our problem is as follows: we consider a particle swarm \(\{\boldsymbol{\theta}_i^1\}_{i=1}^N\) with swarm size \(N\) and an agent \(\boldsymbol{\theta}_i^0\), where \(i\) represents the specific particle and \(t\) is the time-step index. These particles are sampled from a normal distribution, \(\boldsymbol{\theta}_{t+1}^i \sim \mathcal{N}(\boldsymbol{\theta}_t^0, \Sigma_t)\). The agent follows a gradient-descent based dynamical system whereas the surrounding particles assist the agent by proving crucial approximated information about the underlying geometry of the space. Let \(h(\boldsymbol{\theta}_i)\) define a random variable depicting a stochastic process, which is Gaussian in nature. It represents the fit of the dynamical process of gradient descent by predicting the gradients. After every iteration \(t\), a multi-output Gaussian process model uses information about the optimization manifold from the current particles’ positions \(\{\boldsymbol{\theta}_i^t\}\) and the gradients at these locations \(\{\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i^t)\}\) as training data to fit the change \(\delta_t\) for the main agent. We know that \(\theta_i \in \mathbb{R}^d\). Let \(\Theta_t\) denote the collection of location of all particles as: \(\Theta_t = [\theta_1^0 \theta_2^0 \ldots \theta_N^0]^T\), and the location of the agent is the testing data: \(\theta_0^0 = [\theta_0^1 \theta_0^2 \ldots \theta_0^N]^T\). Hence, \(\Theta_t \in \mathbb{R}^{N \times d}\) and \(\theta_0^0 \in \mathbb{R}^d\). The complete training data collection is given by: \((X_t, Y_t) = (\Theta_t, \nabla_{\boldsymbol{\theta}} \mathcal{L}(\Theta_t))\), whereas the point on which the model would be tested to calculate approximated gradient is: \(\theta_0^0\).

For any Gaussian Process Regression problem, we require a prior and the likelihood function. Since the optimization is over a \(d\)-dimensional space, the multi-output Gaussian process model utilizes \(d\) different Gaussian process models for representing and prediction information at each dimension [8]. The random variable \(h(X_i)\) can hence be represented as: \(h(X_i) = [h_1(X_{i,1}), h_2(X_{i,2}), \ldots, h_d(X_{i,d})]\), where \(X_{i,t}\) represents location of all particles in the \(i^{th}\) dimension at the \(t^{th}\) iteration. Hence, to summarize, the relationship between observations, i.e. the gradients \((Y_t)\), and the output

| Name | Scheme to compute \(\delta_t\) |
|------|---------------------------------|
| PSO [6] | \(\delta_t = u\delta_{t-1} + c_1 r_1(\theta_{best} - \theta_{t-1}) + c_2 r_2(\theta_{best} - \theta_{t-1})\) |
| M-PSO [18] | \(\delta_t = (1 - \lambda)\delta_{t-1} + c_1 r_1(\theta_{best} - \theta_{t-1}) + c_2 r_2(\theta_{best} - \theta_{t-1}) + \lambda \delta_{t-1}\) |
| EM-PSO [10] | \(\delta_t = \beta \delta_{t-1} + (1 - \beta)\delta_{t-1}\) |
| REM-PSO [10] | \(\delta_t = M_t + A^T \phi_1 A(\theta_{best} - \theta_{t-1}) + A^T \phi_2 A(\theta_{best} - \theta_{t-1})\) |
| HMC-PSO [16] | \(\delta_t = \beta M_t + (1 - \beta)\delta_{t-1}\) |

Table 2: List of particle-swarm based optimizers

3 Our contribution: ParticleGP

We are motivated from searching the parameter space under PSO framework and propose a particle-based optimization algorithm using multi-output Gaussian process (called ParticleGP) for simulating the dynamical system of parameter updates. As discussed in section 1, one of the primary advantages of using such a technique is that the optimizer can process essential information about the geometry of the parameter space from its local neighbourhood and make an accurate prediction on the objectively best direction to take a descent step. The current state-of-the-art optimizers utilize direct or stochastic gradients at the given point, which while it helps taking the locally best decision, does not paint the entire picture. Our proposed ParticleGP is the most effective method in optimization problems with several local minima and non-convex behaviour: a scenario in which most state-of-the-art optimizers fail to reach the global optima. The setup for our problem is as follows: we consider a particle swarm \(\{\boldsymbol{\theta}_i^1\}_{i=1}^N\) with swarm size \(N\) and an agent \(\boldsymbol{\theta}_i^0\), where \(i\) represents the specific particle and \(t\) is the time-step index. These particles are sampled from a normal distribution, \(\boldsymbol{\theta}_{t+1}^i \sim \mathcal{N}(\boldsymbol{\theta}_t^0, \Sigma_t)\). The agent follows a gradient-descent based dynamical system whereas the surrounding particles assist the agent by proving crucial approximated information about the underlying geometry of the space. Let \(h(\boldsymbol{\theta}_i)\) define a random variable depicting a stochastic process, which is Gaussian in nature. It represents the fit of the dynamical process of gradient descent by predicting the gradients. After every iteration \(t\), a multi-output Gaussian process model uses information about the optimization manifold from the current particles’ positions \(\{\boldsymbol{\theta}_i^t\}\) and the gradients at these locations \(\{\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i^t)\}\) as training data to fit the change \(\delta_t\) for the main agent. We know that \(\theta_i \in \mathbb{R}^d\). Let \(\Theta_t\) denote the collection of location of all particles as: \(\Theta_t = [\theta_1^0 \theta_2^0 \ldots \theta_N^0]^T\), and the location of the agent is the testing data: \(\theta_0^0 = [\theta_0^1 \theta_0^2 \ldots \theta_0^N]^T\). Hence, \(\Theta_t \in \mathbb{R}^{N \times d}\) and \(\theta_0^0 \in \mathbb{R}^d\). The complete training data collection is given by: \((X_t, Y_t) = (\Theta_t, \nabla_{\boldsymbol{\theta}} \mathcal{L}(\Theta_t))\), whereas the point on which the model would be tested to calculate approximated gradient is: \(\theta_0^0\).

For any Gaussian Process Regression problem, we require a prior and the likelihood function. Since the optimization is over a \(d\)-dimensional space, the multi-output Gaussian process model utilizes \(d\) different Gaussian process models for representing and prediction information at each dimension [8]. The random variable \(h(X_i)\) can hence be represented as: \(h(X_i) = [h_1(X_{i,1}), h_2(X_{i,2}), \ldots, h_d(X_{i,d})]\), where \(X_{i,t}\) represents location of all particles in the \(i^{th}\) dimension at the \(t^{th}\) iteration. Hence, to summarize, the relationship between observations, i.e. the gradients \((Y_t)\), and the output
of the Gaussian Process Regression model \( h_i(X_{i,t}) \) is:

\[
Y_{i,t}(X_{i,t}) = I_N h_i(X_{i,t}) + \epsilon_i, \quad \text{where} \ \epsilon_i \sim \mathcal{N}(0, \eta^2 I_N); \ \forall i \in \{1, 2, 3, \ldots, d\}
\]  

(12)

The marginalized Gaussian process prior distribution is given by:

\[
\begin{bmatrix} h_i(X_{i,t}) \\ h_i(\theta_{i,t}^0) \end{bmatrix} \sim \mathcal{N}( \begin{bmatrix} M_{i,t} \\ k(\theta_{i,t}^0, X_{i,t}) \end{bmatrix}, \begin{bmatrix} k(X_{i,t}, X_{i,t}) & k(X_{i,t}, \theta_{i,t}^0) \\ k(\theta_{i,t}^0, X_{i,t}) & k(\theta_{i,t}^0, \theta_{i,t}^0) \end{bmatrix} ), \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(13)

where, \( M_{i,t} = [m_{i,t}^1, m_{i,t}^2, \ldots m_{i,t}^d]^{T} \) represents the generalized mean function for predicting gradients for each of the particles on the \( i^{th} \) dimension, \( m_{i,t} \), similarly, is the mean function for the main agent, \( k \) is the kernel function defined for two points \( x, x' \in \mathbb{R} \) as:

\[
k(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right)
\]

(14)

whereas for the vector \( x = (x_1, x_2, x_3, \ldots, x_d) \in \mathbb{R}^d; \)

\[
k(x, x) = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \ldots & k(x_1, x_d) \\ k(x_2, x_1) & k(x_2, x_2) & \ldots & k(x_2, x_d) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_d, x_1) & k(x_d, x_2) & \ldots & k(x_d, x_d) \end{bmatrix} \in \mathbb{R}^{d \times d}
\]

(15)

According to the marginalization rule, we can establish the prior distribution as a multi-variate Gaussian distribution:

\[
h_i(X_{i,t}) \sim \mathcal{N}(M_{i,t}, k(X_{i,t}, X_{i,t}))
\]

(16)

\[
h_i(\theta_{i,t}^0) \sim \mathcal{N}(m_{i,t}, k(\theta_{i,t}^0, \theta_{i,t}^0)); \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(17)

[2] explores more about the importance of choosing an informative prior, and describes the techniques and assumptions for doing so. Let likelihood at training data points be given by: \( p_i(X_{i,t}) \) and at test data points be given by: \( p_i(\theta_{i,t}^0) \). We can now define the posterior distribution for modelling the dynamics using the Bayes rule as:

\[
p_i(h_i(X_{i,t})|Y_{i,t}) = \frac{p_i(Y_{i,t}|h_i(X_{i,t}))}{\int p_i(Y_{i,t}|h_i(X_{i,t})) p_i(h_i(X_{i,t})) \, dh_i(X_{i,t})} \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(18)

We note that the output (in the form of a random variable) \( Y_{i,t}(X_{i,t}) \) can be represented as a linear transformation of \( h_i(X_{i,t}) \), i.e.:

\[
Y_{i,t}(X_{i,t}) = I_N h_i(X_{i,t}) + \epsilon_i, \quad \text{where} \ \epsilon_i \sim \mathcal{N}(0, \eta^2 I_N); \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(19)

Using linear transformation property for Normal distributions, the marginal distribution for the output is:

\[
Y_{i,t}(X_{i,t}) \sim \mathcal{N}(M_{i,t} + \eta^2 I_N) \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(20)

where, \( K = \begin{bmatrix} k(X_{i,t}, X_{i,t}) & k(X_{i,t}, \theta_{i,t}^0) \\ k(\theta_{i,t}^0, X_{i,t}) & k(\theta_{i,t}^0, \theta_{i,t}^0) \end{bmatrix} \). Using this information, Bayes rule can be applied to compute the joint distributions:

\[
\begin{bmatrix} h_i(\theta_{i,t}^0) \\ Y_{i,t}(X_{i,t}) \end{bmatrix} \sim \mathcal{N}( \begin{bmatrix} M_{i,t} \\ k(\theta_{i,t}^0, X_{i,t}) \end{bmatrix}, \begin{bmatrix} k(X_{i,t}, X_{i,t}) & k(X_{i,t}, \theta_{i,t}^0) \\ k(\theta_{i,t}^0, X_{i,t}) & k(X_{i,t}, \theta_{i,t}^0) + \eta^2 I_N \end{bmatrix} ), \ \forall i \in \{1, 2, 3, \ldots, d\}
\]

(21)

The posterior distribution is \( h_i(\theta_{i,t}^0) | Y_{i,t}(X_{i,t}) \sim \mathcal{N}(\mu_{i,t}^*, \sigma_{i,t}^*) \), where:

\[
\mu_{i,t}^* = m_{i,t} + k(\theta_{i,t}^0, X_{i,t})(k(X_{i,t}, X_{i,t}) + \eta^2 I_N)^{-1}(Y_{i,t}(X_{i,t}) - M_{i,t})
\]

(22)

\[
\sigma_{i,t}^* = k(\theta_{i,t}^0, \theta_{i,t}^0) - k(\theta_{i,t}^0, X_{i,t})(k(X_{i,t}, X_{i,t}) + \eta^2 I_N)^{-1}k(\theta_{i,t}^0, X_{i,t})^{T}
\]

(23)

While utilizing the Gaussian Regression process, we have the following model parameters:

- length-scale \( l \), from the kernel function, is a single scalar.
- signal variance \( \sigma^2 \), also from the kernel function, is a single scalar.

It is crucial to note the importance of the role of these model parameters for effectively learning the dynamics. Sub-par values of these hyper-parameters can lead to completely incorrect dynamics (i.e. over-fitting and under-fitting the descent process). Fortunately, we are able to optimize them by solving a separate optimization problem i.e. the log of the marginal likelihood:
log(\(p(Y_{i,t}(X_{i,t}))\)) = log\left(\frac{1}{(2\pi)^{N/2}\det(K + \eta^2 I_N)}\right)^{1/2} \exp\left(-\frac{1}{2}(Y_{i,t}(X_{i,t}) - M_{i,t})^T(K + \eta^2 I_N)^{-1}(Y_{i,t}(X_{i,t}) - M_{i,t})\right) \\
= \frac{1}{2} \log(\det(K + \eta^2 I_N))^{1/2} - \frac{1}{2} Y_{i,t}(X_{i,t}) - M_{i,t})^T(K + \eta^2 I_N)^{-1}(Y_{i,t}(X_{i,t}) - M_{i,t}) - \frac{N}{2} \log(2\pi) 

Maximizing the above objective function would help in obtaining the dynamical process that best fits the parameters’ behaviour. The approximate gradients for the main agent are then sampled from the updated posterior distribution and coupled together from each of the \(d\) Gaussian-processes, i.e. \(\nabla_\theta \mathcal{L}(\theta_{i_t}^0) \sim \mathcal{N}(\mu_{i_t}^*, \sigma_{i_t}^*)\) and \(\nabla_\theta \mathcal{L}(\theta_t^0) = (\nabla_\theta \mathcal{L}(\theta_{1,t}^0), \nabla_\theta \mathcal{L}(\theta_{2,t}^0), \ldots, \nabla_\theta \mathcal{L}(\theta_{d,t}^0))\). Finally, the update to the main-agent’s/network’s parameters is carried out as follows:

\[
\delta_t^0 = -\alpha_t \cdot \nabla_\theta \mathcal{L}(\theta_t^0) \\
\theta_{t+1}^0 = \theta_t^0 + \delta_t^0
\]

This procedure can be completed for each iteration until the parameters converge to the optimal value. The algorithm 1 summarizes our approach.

**Algorithm 1** Particle-based Gaussian-process Regression for gradient descent

**Initialise:** (i) particle swarm \(\{\theta_{i_0}^0\}_{i=1}^{N_p} \in \mathbb{R}^d\), (ii) main agent representing weights of the neural network \(\theta_t^0 \in \mathbb{R}^d\), (iii) step-size for dynamical system update \(\eta_t\), (iv) co-variance matrix for sampling particles \(\Sigma_t\), (v) model parameters for multi-output Gaussian process \(l\) and \(\sigma^2\).

**for** \(t = 1, 2, 3\ldots T\) **do**

Select \(N_p\) particles from the multi-variate normal distribution: \(\{\theta_{i+1}^0\}_{i=1}^{N_p} \sim \mathcal{N}(\theta_{t}^0, \Sigma_t)\)

Obtain gradients at particle locations: \(\{\nabla_{\theta} \mathcal{L}(\theta_{i_0}^0)\}\)

**for** \(i = 1, 2, 3, \ldots, d\) **do**

Initialize the \(i^{th}\) Gaussian-process Regression prior distribution using equation (22)

Get the posterior distribution via updated values of \(\mu_{i,t}^*\) and \(\sigma_{i,t}^*\) computed using equation (22)

Optimize the log-marginal likelihood function in equation (24) to get the best model parameters \(l\) and \(\sigma^2\)

Get the updated posterior for final values of \(\mu_{i,t}^*\) and \(\sigma_{i,t}^*\) computed using equation (22)

Sample from posterior: \(\nabla_{\theta} \mathcal{L}(\theta_{i_0}^0) \sim \mathcal{N}(\mu_{i,t}^*, \sigma_{i,t}^*)\)

Concatenate values to get approximate gradients: \(\nabla_{\theta} \mathcal{L}(\theta_t^0) = (\nabla_{\theta} \mathcal{L}(\theta_{1,t}^0), \nabla_{\theta} \mathcal{L}(\theta_{2,t}^0), \ldots, \nabla_{\theta} \mathcal{L}(\theta_{d,t}^0))\)

Use GD/Momentum/Adagrad/RMS-Prop/Adam/NAdam for the dynamical system according to the table 2 as required. For a simple gradients update (i.e. GD), take: \(\delta_t^0 = -\eta_t \cdot \nabla_{\theta} \mathcal{L}(\theta_t^0)\)

Update the parameters of the main-agent network using: \(\theta_{t+1}^0 = \theta_t^0 + \delta_t^0\)

**end for**

4 Results

4.1 Non-Convex Quadratic optimization problems

We tested our algorithm, Particle-GP, on several difficult non-convex problems in \(\mathbb{R}^2\). These problems, due to their incredibly complex optimization manifold, were expected to prevent conventional optimizers converge to the global minima. We considered a set-up in which swarm size is \(N = 100\) particles. Hence, the total number of gradient computations for one step of Particle-GP is 100 times larger than one step in SGD. We sampled particles from a normal distribution with an epsilon ball of radius \(\min(0.1, \eta_t)\) centered at the current parameters’ state. We ran ParticleGP for 200 iterations, whereas the rest of the state-of-the-art optimizers for 20000 iterations on a logistic regression model with parameters as the position of the points in the optimization space. This was done to ensure the **total number of gradient computations are equal**. The time taken to run state-of-the-art optimizers is **155** seconds, whereas Particle-GP takes **33** seconds for the given number of iterations. The table 3 displays the step size (\(\eta_t\)) setup for the optimizers for each of the experiments. Each optimizer has its own mechanism to update gradients and due to tough terrains shown by these functions, it becomes necessary to run these optimizers on the settings that favour them the most.
Table 3: The step-size ($\eta_t$) for experiments on non-convex optimization problems (Experiments for **bold** functions required a StepLR action with weight decay rate 0.0001 and step size equal to 1 for a single step to shift the learning rate to normal. This was done after the optimizer reached a loss of lower than 10.)

| Functions [15]       | ParticleGP | Adam | RMSProp | AdaGrad | NAdam |
|----------------------|------------|------|---------|---------|-------|
| Himmelblau           | 0.0005     | 0.0005 | 0.0005 | 0.1     | 0.0005 |
| Ackley               | 0.095      | 0.5   | 0.5     | 10      | 1     |
| Beale                | 0.01       | 0.001 | 0.5     | 0.001   | 0.001 |
| Goldstein Price      | 0.00001    | 0.001 | 0.001   | 0.1     | 0.001 |
| Three Hump Camel     | 0.25       | 2.5   | 0.1     | 1       | 2.5   |
| Easom                | 0.003      | 0.001 | 0.001   | 0.01    | 0.001 |
| Bukin                | 0.001      | 0.001 | 0.001   | 0.01    | 0.001 |
| Matyas               | 0.01       | 0.001 | 0.001   | 1       | 0.001 |
| **Dropwave**         | 1          | 1     | 0.25    | 0.1     | 0.01  |
| **Levy**             | 0.08       | 1     | 1       | 10      | 0.5   |

Table 4: Numerical results for non-convex optimization problems (**bold** indicates that the algorithm converges to local minima instead of global or didn’t show signs of converging to the global minima): These numbers represent the euclidean distance between the actual global minimum of the function and what the model trained using these optimizers predicted after the end of it’s training. (* -> none of the optimizers converged when initialized on the plane).  

| L-2 Norm distance from the global optimum |
|-----------------------------------------|
| Functions [15]       | ParticleGP | Adam | RMSProp | AdaGrad | NAdam |
|----------------------|------------|------|---------|---------|-------|
| Himmelblau           | 1.3968     | 22.4857 | 1.1985 | 23.3218 | 0.1148 |
| Ackley               | 0.1194     | 0     | 0.0007  | 0.0001  | 0     |
| Beale                | 0.0362     | 0     | 0       | 0       | 0     |
| Goldstein Price      | 0.0681     | 0.0004 | 0.0707  | 0       | 0.1138 |
| Three Hump Camel     | 0.6403     | 2.5012 | 2.5712  | 2.5033  | 2.2515 |
| Easom*               | 0.0028     | 0     | 0.0017  | 0.0002  | 0.0001 |
| Bukin                | 0.0035     | 0     | 0       | 0.5101  | 0     |
| Matyas               | 1.5624     | 2.051 | 1.9728  | 7.8506  | 7.3193 |
| **Dropwave**         | 0.6005     | 9.2909 | 1.3523  | 0       | 12.3008 |

Additionally, step-LR is required to ensure that the optimizers don’t escape the global minima. Table 4 showcases the numerical results of convergence of the model trained using different optimizers. The functions on which experiments were conducted are described in detail on [15]. Figure 1 present a few plots of trajectories of optimizers over different functions. For particleGP, only agent 0 is displayed.

We observe that for almost all optimization functions, ParticleGP was able to converge to the global minimum. It did better than other state-of-the-art optimizers in terms of the number of functions it found the global optima since the number of times it was successful in being close to the global optimum and not being stuck in a local minimum was one of the highest out of all optimizers, but poorly in terms of finding the exact point. This can be inferred due to ParticleGP using approximated gradients and not the exact ones. The biggest advantage of utilizing ParticleGP is that it uses useful information from particles in the neighbourhood, to find the direction of descent to the global minima; and hence is mostly successful in solving problems with a complex manifold.

### 4.2 Classification task

To justify that the ParticleGP algorithm can handle high dimensional parameter space, we test it using a simple neural network architecture involving CNNs in the Computer-Vision based classification task and compare it’s performance to state-of-the-art optimizers in the field. The datasets explored are: CIFAR-10, MNIST, Fashion-MNIST. The number of epochs all optimizers were run was 20, batch size of images used for training was 4096, the learning rate for all optimizers was fixed to be 0.001 and the loss function used was CrossEntropyLoss. For particleGP, we used same settings as above, except the number of particles ($N$) are 20. Additionally, after gradient approximation, we applied the Adam updates to compute values of parameters for the next iteration. Table 5 showcases the comparison. The figures 2, 3, and 4 illustrate plots of the running loss and training accuracy. We observe that ParticleGP had the highest
Figure 1: Sample Quadratic Optimization function mesh-plots: We present the topographic map of the function’s space along with the individual trajectory of progress of agents trained on each of the optimizers as they are trained from first epoch. The trajectory is shown in terms of color-coded points representing the model’s prediction of the global minima of the function.

Table 5: Comparison of the optimizers on classification datasets based test-accuracy and Cross-Entropy loss

| Dataset        | Metric        | ParticleGP | Adam  | RMSProp | NAdam | AdaGrad |
|----------------|---------------|------------|-------|---------|-------|---------|
| MNIST          | Testing Accuracy | 92.43%     | 91.73%| 89.98%  | 91.37%| 79.11%  |
|                | Loss          | 0.7694     | 0.8280| 0.9766  | 0.8751| 2.9356  |
| Fashion MNIST  | Testing Accuracy | 83.80%     | 83.78%| 83.85%  | 82.31%| 74.12%  |
|                | Loss          | 1.3626     | 1.3558| 1.3326  | 1.4435| 2.5767  |
| CIFAR-10       | Testing Accuracy | 50.01%     | 49.17%| 50.47%  | 45.74%| 36.39%  |
|                | Loss          | 4.1296     | 4.2224| 4.2047  | 4.5503| 5.2040  |

or the second-highest accuracy and running loss out of all optimizers. The plots showcase that despite ParticleGP being slower in the beginning, it catches up significantly and outperforms others at the end.

Based on these results, we can infer that ParticleGP performs as good as or sometimes slightly better than the current state-of-the-art optimizers in classification tasks.
Figure 2: Performance on the MNIST dataset

Figure 3: Performance on the Fashion MNIST dataset

Figure 4: Performance on the CIFAR10 dataset
5 Conclusion

In this research, we proposed the Particle-based Gaussian process Optimizer, a novel technique utilizing Gaussian process (GP) regression, called as ParticleGP, for representing the dynamical process of gradient descent. In particular, the GP regressed over computed gradients at the neighbourhood points to predict the best direction of descent for the network parameters in every iteration of the scheme. This approach, motivated from utilizing gradients in standard optimizers and using evaluations from a swarm of particles from particle swarm optimization (PSO) methods, attempts to incorporate the best of both techniques. The usage of gradients proved essential to ensure decent along the fastest direction, while the swarm of particles increased the exploration for detecting and converging to the global minima. We described the details of the multi-output Gaussian process regression for predicting gradients and summarized our approach in a single algorithm. After testing ParticleGP on 10 distinct non-convex quadratic optimization problems, we observed it had a slightly better performance than other optimizers. We also tested ParticleGP on a computer vision task and found equivalent to slightly better results than conventional optimizers. The biggest limitation of this work is scalability since introducing Gaussian process regression in the optimizer considerably increases required computational resources and makes it challenging to use in heavier neural network architectures.

Limitations and Future Work

1. ParticleGP, since it uses a Gaussian process Regression model for predicting gradients after each iteration, it requires heavy computational resources for preparing the model and computing the matrix inverses for the posterior distribution update; and whether it showcases a performance tantamount to the same remains to be seen. In the appendix, we provide an approach to utilize sparse-Gaussian process.

2. The hyper-parameters learning_rate, swarm_size, variance for initializing particles directly impact the approximation of gradients for the main agent, and thus are extremely sensitive. A slight change affects the overall results by a significant margin. Hence, a mechanism to optimally control these hyper-parameters; which we are currently working on, is essential. The first sub-section in appendix provides motivation for a control-based gradient-descent technique.

3. The experiments used a relatively light model for computations. It is unclear how ParticleGP would perform in domain-specific models for involving heavy data-sets like VGG, Resnet, Transformer, ViT, etc.

References

[1] A Cauchy. Méthode générale pour la résolution des systemes déquations simultanées. Comptes Rendus de l’Academie des Sciences, 1847.

[2] Guido Consonni, Dimitris Fouskakis, Brunero Liseo, and Ioannis Ntzoufras. Prior distributions for objective bayesian analysis. Bayesian Analysis, 13(2):627–679, 2018.

[3] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research, 12(7), 2011.

[5] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. Cited on, 14(8):2, 2012.

[6] James Kennedy and Russell Eberhart. Particle swarm optimization. In Proceedings of ICNN’95-international conference on neural networks, volume 4, pages 1942–1948. IEEE, 1995.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[8] Haitao Liu, Jianfei Cai, and Yew-Soon Ong. Remarks on multi-output gaussian process regression. Knowledge-Based Systems, 144:102–121, 2018.

[9] Suyun Liu and Luis Nunes Vicente. The stochastic multi-gradient algorithm for multi-objective optimization and its application to supervised machine learning. Annals of Operations Research, pages 1–30, 2021.

[10] Rohan Mohapatra, Snehashu Saha, Carlos A Coello Coello, Anwesh Bhattacharya, Soma S Dhavala, and Sriparna Saha. Adaswarm: Augmenting gradient-based optimizers in deep learning with swarm intelligence. IEEE Transactions on Emerging Topics in Computational Intelligence, 2021.

[11] Ning Qian. On the momentum term in gradient descent learning algorithms. Neural Networks, 12(1):145–151, 1999.

[12] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In International Conference on Learning Representations, 2018.

[13] Herbert Robbins and Sutton Monroe. A Stochastic Approximation Method. The Annals of Mathematical Statistics, 22(3):400 – 407, 1951.
[14] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[15] S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved November 20, 2022, from [http://www.sfu.ca/~ssurjano](http://www.sfu.ca/~ssurjano).

[16] Omathav Bharat Vaidya, Rithvik Terence DSouza, Soma Dhavala, Snehanshu Saha, and Swagatam Das. Hmc-pso: A hamiltonian monte carlo and particle swarm optimization-based optimizer. *ResearchGate*, 2022.

[17] Christopher Williams and Carl Rasmussen. Gaussian processes for regression. *Advances in neural information processing systems*, 8, 1995.

[18] Tao Xiang, Jun Wang, and Xiaofeng Liao. An improved particle swarm optimizer with momentum. In *2007 IEEE Congress on Evolutionary Computation*, pages 3341–3345. IEEE, 2007.

[19] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.