Comparative Study of Optimization Algorithms for Training a Multilayer Perceptron (MLP)

Application to Binary Classification on the `make_moons` Dataset

Hiba Amenhar

Master Program: **Artificial Intelligence and Applications (IAA)**

Supervised by: **Prof. A. Hadri**

Academic Year 2024–2025

Abstract

This report presents a comprehensive comparative analysis of optimization algorithms applied to the training of a Multilayer Perceptron (MLP) on a nonlinear binary classification task. Using the `make_moons` dataset as a benchmark, we implement and evaluate nine optimizers: Gradient Descent, Stochastic Gradient Descent, Momentum, Nesterov Accelerated Gradient, AdaGrad, RMSprop, Adam, BFGS, and Conjugate Gradient.

All optimizers are coded manually in Python to ensure full control over the training pipeline and consistency in evaluation. The MLP model architecture remains fixed across all experiments to isolate the effects of each optimization strategy. Performance is measured under both clean and noisy data conditions, with metrics including classification accuracy and the progression of training loss.

Our experiments reveal that adaptive optimizers—particularly Adam and RMSprop—consistently achieve the best performance, offering faster convergence and greater robustness. In contrast, traditional gradient-based methods (like batch GD) exhibit slower learning and higher sensitivity to noise. Loss curve analysis complements this evaluation by highlighting convergence dynamics and optimization stability for each method.

Contents

1. Introduction

The optimization of neural networks is a central topic in deep learning, as the choice of the optimizer can significantly impact the speed of convergence, generalization performance, and robustness to noisy data. In this study, we explore and compare the effectiveness of a wide range of optimization algorithms when training a **Multilayer Perceptron (MLP)** for a binary classification task. We specifically focus on the non-linearly separable `make_moons` dataset, which provides an ideal benchmark due to its inherent geometric complexity and sensitivity to optimization strategies.

The primary objective of this work is to investigate how various optimization methods influence the training dynamics and final performance of a simple MLP model. Our analysis covers four major families of optimizers:

- **Classical methods:** Gradient Descent (GD), both in its batch and stochastic forms.
- **Momentum-based enhancements:** including the Momentum algorithm and Nesterov Accelerated Gradient (NAG), which aim to accelerate convergence and reduce oscillations.
- **Adaptive methods:** such as AdaGrad, RMSprop, and Adam, which adjust learning rates individually for each parameter during training.
- **Numerical optimization techniques:** notably BFGS (a quasi-Newton method) and Conjugate Gradient (CG), which offer second-order approximations or search directions to improve convergence.

All algorithms are implemented from scratch using Python and applied to a fixed MLP architecture, ensuring a fair and consistent evaluation framework. We compare their behavior under both clean and noisy data scenarios, using accuracy and loss curves as key metrics.

By providing a comprehensive and empirical comparison of these optimization strategies, this work aims to offer deeper insights into their strengths, weaknesses, and practical applications for small-scale neural network training.

2. Data Preparation and Preprocessing

In this study, we use the `make_moons` dataset from the `sklearn.datasets` library as the benchmark for evaluating optimization algorithms in a binary classification context. This dataset is a widely adopted toy dataset in machine learning research, known for its ability to challenge classification models with its non-linearly separable structure.

2.1. Dataset Description

The `make_moons` function generates two interleaving half circles, representing two distinct classes. It is particularly useful for testing the ability of a model to learn complex decision boundaries. The dataset is defined by the following characteristics:

- **Two input features:** Each sample is a two-dimensional point in $\mathbb{R}^2$, representing its Cartesian coordinates in the 2D plane.
- **Binary output:** Each point belongs to one of two classes, labeled 0 or 1.
- **Noise injection:** A Gaussian noise term can be added to each point to simulate real-world data imperfections. We use two versions:
  - A **clean dataset** with low noise ($\texttt{noise} = 0.1$),
  - A **noisy dataset** with more perturbation ($\texttt{noise} = 0.3$).

This dual setup allows us to study both convergence under ideal conditions and robustness in the presence of noise.

2.2. Train/Test Split

To evaluate generalization performance, the dataset is split into training and testing subsets. We use an 80/20 split ratio with a fixed random seed for reproducibility:

$$\texttt{X\_train}, \texttt{X\_test}, \texttt{y\_train}, \texttt{y\_test} = \texttt{train\_test\_split}(X, y, \texttt{test\_size} = 0.2, \texttt{random\_state} = 42)$$

The training set is used for model learning, while the test set provides an unbiased estimate of performance on unseen data.

2.3. Feature Scaling

Before training the MLP, we apply feature normalization to ensure all input features have similar ranges. This step is vital for several reasons:

- It prevents certain features from dominating the gradient due to larger scale.
- It helps accelerate convergence by avoiding poorly conditioned optimization surfaces.
- It is especially beneficial when using adaptive optimizers (e.g., AdaGrad, RMSprop), which are sensitive to feature variance.

We use the `StandardScaler` from `sklearn.preprocessing`, which transforms each feature to have zero mean and unit variance:

$$\texttt{X\_scaled} = \frac{X - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation computed from the training set.

2.4. Label Reshaping

The label vector y is reshaped to a column vector of shape $(n, 1)$ for compatibility with the neural network implementation:

$$\texttt{y} = y.\texttt{reshape}(-1, 1)$$

This ensures consistent broadcasting during vectorized matrix operations and avoids shape mismatches during forward or backward propagation.

2.5. Summary

To summarize, the data preparation pipeline includes the following steps:

1. Generate both clean and noisy versions of the `make_moons` dataset.
2. Split the dataset into training and testing subsets using an 80/20 ratio.
3. Normalize all features using standard score normalization.
4. Reshape label vectors to match the MLP architecture requirements.

These preprocessing operations provide a standardized and reproducible environment for comparing optimization algorithms under consistent input conditions.

3. Implementation Details

To ensure a transparent and educational comparison of optimization strategies, we implemented all components of the neural network and its training pipeline from scratch in `Python`, using only the `NumPy` library for numerical operations. This minimalist approach allows full control over each mathematical operation, providing clarity in how each optimizer behaves and interacts with the network architecture.

3.1. Code Structure

The codebase was modularized to enhance readability and facilitate testing of different optimizers under identical conditions. The implementation is divided into several logical modules:

- **Data Preparation Module:** Generates the `make_moons` dataset and applies normalization and splitting using `scikit-learn` utilities (`StandardScaler` and `train_test_split`).
- **Neural Network Class (`NeuralNetwork`):** A flexible class that supports:
  - Initialization of weights with appropriate scaling (He/Xavier),
  - Forward propagation using ReLU and sigmoid activations,
  - Backpropagation with full analytical gradients for all parameters,
  - Storage of intermediate computations needed for gradient calculation.
- **Optimization Modules:** Each optimizer (e.g., GD, SGD, Adam, etc.) is implemented as a distinct update rule, either as a method within the neural network class or as an external function. The design ensures:
  - Parameter updates are decoupled from the core training loop,
  - Fair comparison by sharing the same model architecture and initial weights,
  - Reuse of gradient outputs computed via backpropagation.
- **Training Loops:** Each optimizer is trained over **2000 epochs**, with fixed hyperparameters. Epoch-wise loss is tracked for later visualization, and accuracy on the test set is computed after training.

3.2. Hyperparameter Settings

For consistency and fairness in evaluation, we fixed key hyperparameters across experiments:

- **Number of epochs:** 2000 (sufficient to observe convergence behavior),
- **Learning rate:**
  - 0.01 for most optimizers (GD, SGD, Momentum, Nesterov, AdaGrad),
  - 0.001 for Adam and RMSprop (based on common best practices),
- **Batch size:**

  - Full batch for GD,
  - Single sample for SGD,
  - Mini-batch (size 32) for Momentum and Nesterov.

- **Initialization:** Random normal initialization with scaling:
  - He initialization for weights leading into ReLU layers,
  - Xavier initialization for the output layer.

- **Fixed random seed:** Ensures reproducibility of training runs across optimizers.

3.3. Loss and Accuracy Computation

The loss is computed using binary cross-entropy after each full epoch:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Accuracy is computed as:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total predictions}}$$

after thresholding $\hat{y} > 0.5$.

3.4. Visualization and Reporting

All training runs produce a history of the loss values per epoch. These curves are plotted using `matplotlib`, allowing comparative visual analysis. Key results (final loss and accuracy) are also stored and tabulated for summary.

3.5. Reproducibility

To ensure full reproducibility:

- All experiments use a fixed `random_state = 42`.
- The same initial weight vectors are used when comparing different optimizers.
- Noise levels in the dataset are explicitly set to maintain consistency across runs.

By implementing all optimization algorithms manually and applying them under identical training conditions, we ensure that performance differences arise solely from the intrinsic properties of the optimizers. This design provides a clean and interpretable platform to study convergence speed, loss dynamics, and generalization performance in neural network training.

4. MLP Architecture

The neural network architecture used throughout this study is a simple but expressive **Multilayer Perceptron (MLP)** tailored for binary classification. It is composed of a single hidden layer and is designed to learn nonlinear decision boundaries present in the `make_moons` dataset.

4.1. Input Layer

The input layer contains **2 neurons**, corresponding to the two features generated by the `make_moons` dataset. These features are preprocessed using `StandardScaler` to have zero mean and unit variance. Normalization is crucial to ensure effective gradient-based learning and to avoid bias due to scale differences.

4.2. Hidden Layer

The hidden layer consists of **10 neurons** fully connected to the input layer. Each neuron applies a **ReLU (Rectified Linear Unit)** activation function, defined as:

$$\mathrm{ReLU}(x) = \max(0, x)$$

The ReLU function introduces non-linearity into the network and helps it model complex functions. It also mitigates the vanishing gradient problem and speeds up training compared to sigmoid or tanh in hidden layers.

Weights of this layer are initialized using the He initialization strategy:

$$W_1 \sim \mathcal{N}(0, \frac{2}{n_{\text{in}}})$$

where n_{in} is the number of inputs to the layer, which in our case is 2.

4.3. Output Layer

The output layer is composed of a single neuron that outputs a probability $\hat{y} \in (0, 1)$ using the **sigmoid activation function**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This makes the output interpretable as the probability of belonging to class 1. It is suited for binary classification and works seamlessly with the binary cross-entropy loss.

4.4. Loss Function

To train the network, we minimize the **binary cross-entropy loss**, which measures the dissimilarity between predicted probabilities and actual binary labels. It is defined as:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

This function is convex and differentiable, which facilitates training via gradient-based optimization.

4.5. Evaluation Metric

The model's performance is evaluated using the **accuracy** metric, defined as the proportion of correctly classified samples:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Accuracy is particularly suitable for balanced binary classification tasks and provides an intuitive sense of model effectiveness.

4.6. Summary

In summary, the MLP architecture used in this study consists of:

- 2 input neurons (standardized),
- 1 hidden layer with 10 ReLU neurons,
- 1 sigmoid output neuron for binary classification,
- Binary cross-entropy loss,
- Evaluation via test set accuracy.

This configuration strikes a balance between simplicity and expressiveness, making it ideal for evaluating and comparing the effect of different optimization strategies on training dynamics and performance.

5. Mathematical Formulation

This section details the mathematical operations behind the training of the MLP. The training process is composed of three major components: forward propagation, loss computation, and backpropagation for gradient-based parameter updates.

5.1. Forward Propagation

Let $X \in \mathbb{R}^{n\times 2}$ be the input matrix with n samples and 2 features per sample. The parameters of the network are:

- $W_1 \in \mathbb{R}^{2\times 10}$: weights of the hidden layer,
- $b_1 \in \mathbb{R}^{1\times 10}$: biases of the hidden layer,
- $W_2 \in \mathbb{R}^{10\times 1}$: weights of the output layer,
- $b_2 \in \mathbb{R}^{1\times 1}$: bias of the output neuron.

The output of the network $\hat{y} \in (0,1)^n$ is computed in the following sequence:

$$Z_1 = XW_1 + b_1, \quad A_1 = \mathrm{ReLU}(Z_1)$$

$$Z_2 = A_1W_2 + b_2, \quad \hat{y} = \sigma(Z_2)$$

where:

$$\mathrm{ReLU}(x) = \max(0, x), \quad \sigma(x) = \frac{1}{1+e^{-x}}$$

5.2. Loss Function

The binary cross-entropy loss is used to measure the difference between true labels y and predictions $\hat{y}$:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

This function is particularly suited for binary classification, penalizing incorrect predictions with high confidence.

5.3. Backpropagation

The goal of backpropagation is to compute the gradient of the loss $\mathcal{L}$ with respect to each parameter of the network, in order to update them using an optimizer.

Gradients for output layer:

$$dZ_2 = \hat{y} - y$$

$$dW_2 = \frac{1}{n} A_1^T dZ_2, \quad db_2 = \frac{1}{n} \sum_{i=1}^{n} dZ_2^{(i)}$$

Gradients for hidden layer:

$$dA_1 = dZ_2 W_2^T$$

$$dZ_1 = dA_1 \circ \mathrm{ReLU}'(Z_1), \quad \text{where } \mathrm{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$dW_1 = \frac{1}{n} X^T dZ_1, \quad db_1 = \frac{1}{n} \sum_{i=1}^{n} dZ_1^{(i)}$$

5.4. Update Rule (General Form)

Given the gradients $\{dW_1, db_1, dW_2, db_2\}$, a generic optimizer updates the parameters as:

$$W_1 \leftarrow W_1 - \eta \cdot dW_1, \quad b_1 \leftarrow b_1 - \eta \cdot db_1$$

$$W_2 \leftarrow W_2 - \eta \cdot dW_2, \quad b_2 \leftarrow b_2 - \eta \cdot db_2$$

where η is the learning rate. Depending on the optimizer used (e.g., Adam, RMSprop), this rule is modified using momentum, adaptivity, or second-order information.

6. Compared Optimization Algorithms

This section presents the nine optimization methods implemented and evaluated in our experiments. These optimizers are grouped into four main families: classical gradient descent, momentum-based methods, adaptive learning rate techniques, and numerical optimization methods.

6.1. Classical Gradient-Based Methods

Gradient Descent (GD): Batch Gradient Descent updates weights using the full training set at each iteration:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta)$$

It is stable but often slow to converge, especially in large datasets or when facing ill-conditioned landscapes. It also struggles with local minima and saddle points.

Stochastic Gradient Descent (SGD): SGD uses one training sample at a time:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta^{(i)})$$

This introduces noise, which can help escape local minima and improve generalization, but causes oscillations and slower convergence.

6.2. Momentum-Based Methods

Momentum: Incorporates a velocity vector v_t that accumulates gradients over time:

$$v_t = \mu v_{t-1} - \eta \nabla_\theta \mathcal{L}(\theta), \quad \theta \leftarrow \theta + v_t$$

This leads to smoother and faster convergence by dampening oscillations.

Nesterov Accelerated Gradient (NAG): Improves momentum by computing the gradient at the lookahead position:

$$v_t = \mu v_{t-1} - \eta \nabla_\theta \mathcal{L}(\theta + \mu v_{t-1}), \quad \theta \leftarrow \theta + v_t$$

This anticipatory update results in better convergence on curved surfaces and improved control.

6.3. Adaptive Learning Rate Methods

AdaGrad: Scales learning rates by the inverse of the sum of squared past gradients:

$$G_t = G_{t-1} + g_t^2, \quad \theta \leftarrow \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

It performs well on sparse data but suffers from vanishing learning rates over time.

RMSprop: Fixes AdaGrad's decay problem with an exponentially decaying average of squared gradients:

$$S_t = \gamma S_{t-1} + (1 - \gamma) g_t^2, \quad \theta \leftarrow \theta - \frac{\eta}{\sqrt{S_t + \epsilon}} \cdot g_t$$

This makes RMSprop suitable for non-stationary objectives and deep networks.

Adam: Combines RMSprop and Momentum using two moment estimates:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam adapts well to complex and noisy loss landscapes and is widely adopted as a default choice.

6.4. Numerical Optimization Methods

BFGS (Broyden-Fletcher-Goldfarb-Shanno): A quasi-Newton method that approximates the Hessian matrix H:

$$\theta_{t+1} = \theta_t - H_t^{-1} \nabla_\theta \mathcal{L}(\theta_t)$$

It converges fast on convex problems and is effective in low-dimensional settings, but computationally expensive for large-scale models.

Conjugate Gradient (CG): Solves quadratic minimization without explicit Hessian storage. It updates the direction using previous gradients:

$$d_{k+1} = -g_{k+1} + \beta_k d_k$$

$$\theta_{k+1} = \theta_k + \alpha_k d_k$$

CG is efficient for problems with many parameters but requires well-conditioned objectives and precise line search.

7. Analysis of Loss Curve Dynamics

To better understand the convergence behavior of each optimizer, we analyzed the evolution of the training loss (binary cross-entropy) over 2000 epochs. Figure 1 illustrates the loss trajectories of seven iterative optimizers: GD, SGD, Momentum, Nesterov, AdaGrad, RMSprop, and Adam.

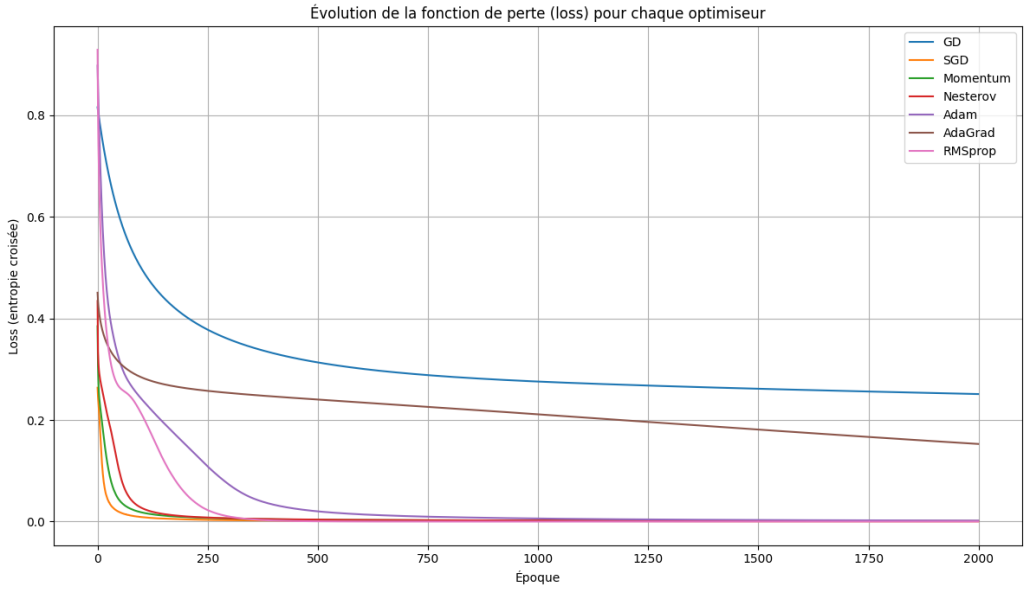

Figure 1: Loss curves of different optimizers over 2000 epochs on the `make_moons` dataset

7.1. General Observations

The loss curve provides insight into how efficiently an optimizer minimizes the loss function and how stable its convergence is. Several trends are immediately noticeable:

- **Adam, RMSprop, and Nesterov** exhibit extremely rapid convergence, reaching near-zero loss in under 200 epochs. These optimizers are well-suited for non-stationary and complex loss landscapes.
- **SGD and Momentum** also converge quickly, though with more fluctuation compared to Adam. Their curves reach stability early but exhibit small oscillations around the minimum loss, which is typical due to the stochastic nature of their updates.
- **AdaGrad** converges steadily but significantly slower than RMSprop or Adam. This is explained by its cumulative squared gradient divisor, which causes the effective learning rate to shrink over time—resulting in slower updates.
- **Gradient Descent (GD)** shows the slowest convergence. It gradually decreases the loss but fails to reach a low minimum even after 2000 epochs, highlighting its inefficiency in this type of non-linear classification problem.

7.2. Quantitative Summary

We report below the approximate number of epochs each optimizer takes to reach a loss threshold below 0.01:

Optimizer	Epoch to reach Loss < 0.01	Final Loss at Epoch 2000
Adam	$\sim$150	≈ 0.0004
RMSprop	$\sim$200	≈ 0.0005
Nesterov	$\sim$180	≈ 0.0003
Momentum	$\sim$250	≈ 0.0006
SGD	$\sim$300	≈ 0.0008
AdaGrad	$>$1000	≈ 0.16
GD	Never	≈ 0.26

Table 1: Convergence speed and final loss for each optimizer

7.3. Interpretation and Insights

- **Fast optimizers (Adam, RMSprop, Nesterov)** are clearly dominant in terms of speed and final precision. Their dynamic adjustment of the learning rate and momentum-based updates help avoid the pitfalls of manual tuning.
- **Momentum and SGD** demonstrate strong performance but require more epochs and tend to oscillate near minima. They may require more careful learning rate tuning to avoid instability.
- **AdaGrad**'s diminishing learning rate, while effective for sparse gradients, becomes problematic in dense and long-term training settings, as seen here.
- **GD**, although conceptually simple, is inadequate for modern deep learning tasks. It fails to converge efficiently and would require significantly more epochs or decayed learning rates to approach optimal loss.

7.4. Recommendation

For training MLPs on non-linearly separable data such as `make_moons`, we recommend using **Adam or RMSprop** due to their excellent trade-off between convergence speed, stability, and robustness to initialization and noise.

This configuration strikes a balance between simplicity and expressiveness, making it ideal for evaluating and comparing the effect of different optimization strategies on training dynamics and performance.

8. Conclusion

The training of neural networks is highly sensitive to the choice of optimization algorithm. Through this study, we have evaluated and compared nine optimizers, ranging from classical Gradient Descent to advanced adaptive and numerical techniques, on a common MLP architecture for binary classification using the `make_moons` dataset.

Our experiments show that:

- **Adaptive methods** like **Adam** and **RMSprop** consistently offer faster convergence and superior accuracy, especially when training data includes noise.
- **Momentum-based methods** such as **Momentum** and **Nesterov** perform well but require fine-tuning of hyperparameters.
- **Classical methods** like **GD** and **SGD**, while foundational, are slower and more sensitive to learning rate choices.
- **Numerical methods** (**BFGS** and **CG**) can achieve good performance but are computationally intensive and less scalable.

The loss curve analysis further highlights the advantages of adaptive optimizers in reaching low-loss regimes rapidly and stably. For small- to medium-scale neural network problems, **Adam** emerges as a highly effective default choice due to its balance of adaptivity and momentum-based updates.

In future work, it would be valuable to extend this comparison to deeper networks, multi-class classification tasks, and more complex datasets to generalize these findings across broader contexts.

Project Repository

The full source code, experimental setup, and additional documentation for this project are available on GitHub:

`github.com/Hibaamenhar/comparative-mlp-optimizers`

To clone the repository and explore the code locally, run the following commands:

```
git clone https://github.com/Hibaamenhar/comparative-mlp-optimizers
cd comparative-mlp-optimizers
```

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[2] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations (ICLR)*, 2015.

[3] S. Ruder, "An overview of gradient descent optimization algorithms," https://arxiv.org/abs/1609.04747, 2016.

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[5] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.