Project 2

FYS-STK4155

Erik Joshua Røset & Oskar Idland
University of Oslo, Department of Physics
(Dated: October 31, 2024)

Add the abstact in the end

https://github.com/Oskar-Idland/FYS-STK4155-Projects

I. INTRODUCTION

The ever-growing complexity of modern data calls for robust machine learning techniques capable of handling non-linear patterns and high-dimensional spaces. While traditional methods, such as Ordinary Least Squares (OLS), Ridge, and Lasso regression, offer valuable insights into model behavior and regularization, their limitations become apparent when addressing more intricate datasets. In our previous project, we explored these regression techniques in-depth, assessing their effectiveness on synthetic and real-world datasets, with particular attention to the bias-variance tradeoff and resampling methods for model validation. However, as data complexity increases, more sophisticated methods, such as neural networks, become necessary to model non-linear relationships and provide superior generalization.

In this project, we extend the concepts introduced earlier by diving into two fundamental areas of machine learning: optimization techniques and neural networks. These topics form the backbone of many modern machine learning algorithms and offer powerful tools for tackling a wide range of problems, from regression and classification to complex tasks in pattern recognition and forecasting. We implement and analyze these methods using both synthetic data (a simple 2 dimensional terrain data generated through the Franke function) and real-world data (the Wisconsin Breast Cancer dataset for classification tasks).

Gradient descent plays a critical role in optimizing the parameters of complex models, especially when closed-form solutions are impractical or non-existent. In implementing Stochastic Gradient Descent (SGD), we explore various techniques for improving its convergence and stability. These include the incorporation of momentum and mini-batch processing, as well as adaptive learning rate methods like Adagrad, RMSprop, and Adam that automatically adjust the learning rate during training. Understanding how these adaptations affect the optimization process—from convergence speed to the ability to escape local minima—is crucial for developing efficient and robust machine learning models.

Furthermore, we implement feed-forward neural networks from scratch, providing a deep understanding of their inner workings. These models, inspired by biological neurons, consist of multiple layers of interconnected nodes that enable the learning of intricate data structures through non-linear activation functions. We investigate the impact of different activation functions (Sigmoid, RELU, and Leaky RELU), network architectures, and hyperparameters on model performance. By comparing our implementations with established libraries like Scikit-learn, we gain insights into both the theoretical foundations and practical considerations of neural network development.

The methods explored in this project have profound implications for modern machine learning applications. Efficient optimization techniques like SGD and its variants have made it possible to train large-scale neural networks on massive datasets, enabling breakthroughs in fields ranging from computer vision to natural language processing. Understanding these fundamental building blocks is crucial for developing more advanced machine learning systems and addressing increasingly complex real-world challenges. In section II, we present the theoretical framework underlying gradient descent optimization and neural networks,,with particular attention to the mathematical foundations of backpropagation and various activation functions. Section III details our methodology, including the implementation of both basic SGD and its advanced variants, as well as our neural network architecture design choices. In section IV, we present comprehensive results comparing the performance of different optimization methods and network configurations across both regression and classification tasks. Finally, section V summarizes our findings and discusses their implications for future applications in machine learning.

Through this comprehensive exploration of optimization methods and neural networks, we aim to bridge the gap between traditional regression techniques and modern deep learning approaches. Our analysis spans both regression and classification tasks, allowing us to evaluate the strengths and limitations of each method across different problem domains. The project not only builds upon our previous work with linear models but also establishes a foundation for understanding more advanced machine learning architectures and their optimization challenges.

II. THEORY

In this project, we extend our previous exploration of machine learning techniques into more advanced optimization algorithms and neural networks. Gradient descent methods, along with their various enhancements, are crucial in training neural networks, which are used to model complex, non-linear relationships. While the foundational principles of regression analysis and optimization were addressed in our previous report, here we focus specifically on gradient descent with and without momentum, and tuning methods such as decaying learning rates, AdaGrad, RMSprop, and ADAM. Additionally, the fundamentals of neural networks will be discussed, emphasizing their relevance to the current project.

A. Gradient Descent Methods

Gradient descent is a fundamental optimization algorithm used to minimize a cost function $C(\beta)$, where β represents the model parameters. Unlike the closed-form solutions available for OLS and Ridge regression in Project 1, many modern machine learning models, particularly neural networks, require iterative optimization methods. The core principle of gradient descent is to iteratively update the parameters in the direction that minimizes the cost function. Given a cost function $C(\beta)$ and current parameters β_t at iteration t, the basic update rule is:

$$\beta_{t+1} = \beta_t - \eta \nabla_\beta C(\beta_t) \tag{1}$$

where $\eta > 0$ is the learning rate that controls the size of the update step, and $\nabla_\beta L(\beta_t)$ is the gradient of the cost function with respect to β . The gradient represents the direction of steepest ascent in the cost landscape, so moving in the negative gradient direction ensures we minimize the cost function.

1. Gradient Descent

In its simplest form, "plain" gradient descent (GD) computes the gradient using the entire dataset:

$$\nabla_\beta C(\beta_t) = \sum_{i=1}^{n} \nabla_\beta C_i(\beta_t) \tag{2}$$

Where n is the total number of training examples and $C_i(\beta_t)$ is the cost for the i-th example. While this approach would provide the most accurate estimate of the gradient, it has several significiant limitations:

- Computational Cost: For large datasets, computing the gradient over all n data points becomes extremely expensive, as each update step requires a complete pass through the dataset.
- Local Minima: Since the algorithm is deterministic, it will converge to a local minimum of the cost function if it converges at all. In machine learning applications, where we often deal with highly non-convex cost landscapes, this can lead to suboptimal solutions.
- Initial Conditions: The final solution strongly depends on the initialization of parameters. Different starting points may lead to different local minima, making the choice of initialization crucial.
- Uniform Step Size: The learning rate η is uniform across all parameter dimensions, which can be problematic when the cost surface has different curvatures in different directions. This often forces us to use a conservatively small learning rate determined by the steepest direction, significantly slowing down convergence in flatter regions.

2. Stochastic Gradient Descent

To address these limitations, Stochastic Gradient Descent (SGD) introduces randomness into the optimization process. Instead of using the entire dataset for each update, SGD uses randomly selected subsets of the data, called mini-batches:

The fundamental idea that SGD is built on is that the cost function can be written as the average of the cost functions for individual training examples. It then follows that the gradient can be computed as a sum over indivudual gradients. We can then approximate the gradient by only computing the gradient for a single minibatch:

$$\nabla_\beta C(\beta_t) \approx \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{x}_i, \beta_t) \tag{3}$$

The entire dataset can be split into n/M minibatches (B_k). The size M of the minibatch represents a key parameter choice in SGD. When $M = n$, we recover the "plain" gradient descent method, while $M = 1$ represents pure stochastic gradient descent where updates are made using a single randomly chosen data point. The choice of minibatch size M thus allows us to balance between the accurate but computationally expensive gradient estimates of batch gradient descent and the noisy but frequent updates of pure SGD. In practice, minibatch sizes are often chosen to be much smaller than n to maintain the computational efficiency and stochastic nature of SGD while reducing the variance in gradient estimates compared to when $M = 1$. This gives us the update rule for SGD:

$$\beta_{t+1} = \beta_t - \eta \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{x}_i, \beta_t) \tag{4}$$

Only processing a subset of the data at each iteration facilitates more frequent parameter updates, as the required computational power for each iteration is significantly reduced. The inherent noise in gradient estimates can also help the optimizer escape poor local minima and saddle points. However, the algorithm becomes highly sensitive to the choice of learning rate. Too large values can cause divergence, while too small values lead to slow convergence.

B. Advanced Optimization Methods

These limitations motivate two key enhancements to the basic SGD algorithm. The uniform step size problem can be addressed by introducing momentum, which helps the optimizer maintain velocity in consistent directions while damping oscillations in regions of varying curvature. The learning rate sensitivity issue can be tackled through adaptive learning rate methods, which automatically adjust the learning rate based on the observed geometry of the cost function during training.

1. Momentum

A key limitation of basic gradient descent methods is their uniform step size across all directions, which can lead to slow convergence, especially in regions where the cost surface has different curvatures in different directions. Momentum addresses this by accumulating a velocity vector that helps accelerate convergence and dampen oscillations:

$$v_{t+1} = \gamma v_t + \eta \nabla_\beta C(\beta_t) \tag{5}$$

$$\beta_{t+1} = \beta_t - v_{t+1} \tag{6}$$

where γ (typically 0.9) is the momentum coefficient that determines how much of the previous velocity is retained. This modification provides several advantages:

- Faster convergence in regions where the gradient is consistent
- Reduced oscillations in directions of high curvature
- Ability to escape shallow local minima

2. Learning Rate Tuning Methods

Choosing the right learning rate η is critical to the success of gradient descent algorithms. If the learning rate is too large, the optimization may overshoot the minimum, while if it is too small, convergence will be very slow. Several adaptive learning rate methods have been proposed to dynamically adjust the learning rate during training:

a. Decaying Learning Rate One simple approach is to gradually decrease the learning rate as training progresses, using a schedule such as:

$$\eta_t = \frac{\eta_0}{1 + \lambda t}$$

where η_0 is the initial learning rate, t is the iteration, and λ is the decay rate. This helps ensure that larger updates are made at the start of training when far from the optimum, and smaller, more precise updates are made later.

b. AdaGrad (Adaptive Gradient Algorithm): AdaGrad adapts the learning rate for each parameter based on the historical gradients. It assigns a smaller learning rate to frequently updated parameters and a larger rate to less frequently updated ones:

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\beta L(\beta_t)$$

where G_t is the sum of the squares of the past gradients, and ϵ is a small constant to avoid division by zero. AdaGrad is well-suited for sparse data but may become overly conservative in later iterations due to the cumulative sum of gradients.

c. RMSprop To address AdaGrad's diminishing learning rates, RMSprop uses a moving average of the squared gradients to scale the learning rate. This helps maintain a balance between fast convergence and smooth parameter updates:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2$$

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\beta L(\beta_t)$$

where β (typically 0.9) controls the decay rate of the moving average, and $E[g^2]_t$ is the moving average of the squared gradients.

d. Adaptive Moment Estimation (ADAM) ADAM combines the advantages of both AdaGrad and RMSprop by keeping track of both the first moment (mean) and the second moment (uncentered variance) of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\beta L(\beta_t)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_\beta L(\beta_t))^2$$

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

$$\beta_{t+1} = \beta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

ADAM is one of the most widely used optimization algorithms today due to its ability to dynamically adjust learning rates and its robustness in practice.

C. Neural Networks

Neural networks are powerful machine learning models designed to capture complex, non-linear relationships in data. At their core, neural networks consist of multiple layers of interconnected neurons (or nodes). Each neuron computes a weighted sum of its inputs, applies a non-linear activation function, and passes the result to the next layer.

In a feedforward neural network, information flows from the input layer through one or more hidden layers to the output layer. Each layer transforms the input data in a hierarchical manner, allowing the network to learn representations at multiple levels of abstraction. The output of the network can be used for tasks such as classification, regression, or even generating new data.

The training of neural networks involves backpropagation, a process in which the gradient of the cost function is computed with respect to the network's parameters. Backpropagation allows for the efficient calculation of these gradients through the chain rule of calculus, which are then used to update the weights via gradient descent.

Relevant to this project, neural networks can benefit significantly from the advanced gradient descent methods described above. Techniques like momentum, ADAM, and learning rate decay help overcome challenges such as slow convergence and the vanishing gradient problem, especially in deep networks with many layers.

Activation Functions and Relevance to the Project

In this project, the choice of activation function plays a crucial role. The commonly used activation functions include: - ReLU (Rectified Linear Unit): ReLU is defined as $\text{ReLU}(x) = \max(0, x)$, which introduces non-linearity and mitigates the vanishing gradient problem. - Sigmoid: Sigmoid squashes the input to a value between 0 and 1, making it useful for binary classification tasks but prone to vanishing gradients. - Tanh: Similar to sigmoid but maps inputs to the range (-1, 1), which can center data and mitigate some issues caused by sigmoid.

Given the nature of the tasks in this project (such as handling non-linear data structures or optimizing complex cost functions), selecting appropriate activation functions and tuning the network's architecture will be key to achieving robust results.

III. METHODS & IMPLEMENTATION

IV. RESULTS & DISCUSSION

V. CONCLUSION

REFERENCES

Appendix A: Code

Link to our GitHub repository: https://github.com/Oskar-Idland/FYS-STK4155-Projects