Project 2

FYS-STK4155

Erik Joshua Røset & Oskar Idland
University of Oslo, Department of Physics
(Dated: November 5, 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 significant 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 individual 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:

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.

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.

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.

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

Having established the mathematical framework for optimizing model parameters through gradient descent and its variants, we now turn our attention to neural networks - a powerful class of models whose training relies heavily on these optimization techniques. Neural networks represent a significant advancement beyond the linear models discussed in our previous work, offering the capability to model complex, non-linear relationships in data. These networks can be trained for both regression tasks, where we predict continuous values similar to our previous project, and classification tasks where we predict discrete categories. While linear regression models are constrained to linear combinations of input features, neural networks can approximate arbitrary continuous functions through the composition of multiple non-linear transformations. This property, formally stated in the universal approximation theorem [**?**], means that neural networks can theoretically approximate any continuous function to arbitrary accuracy given sufficient size.

This approximation capability is achieved through a network of interconnected nodes organized into layers, where each node applies a non-linear transformation to its input. The network learns by adjusting parameters called weights and biases, which determine how information flows and is transformed between nodes. During training, these parameters are iteratively adjusted to minimize a cost function, allowing the network to learn patterns directly from the data without explicit feature engineering.

1. *Feed-Forward Neural Networks*

A feed-forward neural network (FFNN) is structured into three distinct types of layers: the input layer, hidden layers, and output layer. The input layer represents the input data, with each node corresponding to a feature in the dataset. The output layer produces the final prediction, with its structure depending on the task - a single node for regression problems or multiple nodes for classification tasks. Between these lie the hidden layers, where the network learns to identify and extract relevant patterns from the input data. The term ”hidden” stems from the fact that while we observe the inputs and final outputs during training, we do not directly observe what features these intermediate layers learn to represent.

The FFNN generalizes our previous linear models by defining the model output $\tilde{y}$ as a nested series of transformations of the input vector $x \in \mathbb{R}^n$. The fundamental computation at each node takes the form:

$$z = wx + b \tag{7}$$

$$a = f(z) \tag{8}$$

Where w are the weights, b is the bias, and $f(\cdot)$ is a non-linear activation function. This computation can be extended to a layer of neurons by organizing the weigths into a matrix $W \in \mathbb{R}^{m\times n}$ and the biases into a vector $\mathbf{b} \in \mathbb{R}^m$:

$$\mathbf{z} = W\mathbf{X} + \mathbf{b} \tag{9}$$

$$\mathbf{a} = f(\mathbf{z}) \tag{10}$$

Where $\mathbf{X} \in \mathbb{R}^{b\times n}$ represent a batch of input vectors and $\mathbf{a} \in \mathbb{R}^{b\times m}$ is the layer output and $\mathbf{b}$ is a vector of biases. For a network with L layers, we can express the complete forward propagation as:

$$\begin{aligned}
\mathbf{z}^1 &= W^1\mathbf{X} + \mathbf{b}^1 \\
\mathbf{a}^1 &= f(\mathbf{z}^1) \\
\mathbf{z}^2 &= W^2\mathbf{a}^1 + \mathbf{b}^2 \\
\mathbf{a}^2 &= f(\mathbf{z}^2) \\
&\cdots \\
\mathbf{z}^L &= W^L\mathbf{a}^{L-1} + \mathbf{b}^L \\
\mathbf{a}^L &= f(\mathbf{z}^L)
\end{aligned} \tag{11}$$

This iterative process of matrix multiplications and non-linear transformations enables the network to progressively build more complex representations of the input data at each layer. The final output $\mathbf{a}^L$ is then used to compute the cost function, which is minimized during training to learn the optimal weights and biases.

2. *Initialization and Regularization*

The weights and biases represent the parameters that the network learns during training. Each weight w^l_{ij} represents the strength of the connection between node i in layer $l-1$ and node j in layer l, determining how much influence the output from one node has on the next. The bias b^l_j for each node j in layer l allows the network to shift its activation function, providing an additional degree of freedom in fitting the data. Together, these parameters define how information is transformed as it moves through the network.

The choice of initial weights can significantly impact whether a network learns effectively or fails to train. Poor initialization can lead to either vanishing or exploding gradients, particularly in deep networks. If initial weights are too small, the signals shrink as they pass through each layer until they become too weak to drive meaningful learning. Conversely, if weights are too large, the signals grow exponentially, saturating activation functions and stalling learning. A common approach is to draw initial weights from a normal distribution with zero mean and a variance scaled by the number of input connections to each neuron.

$$w_{jk}^l \sim \mathcal{N}\left(0, \frac{1}{n_{in}}\right), \tag{12}$$

where n_{in} is the number of inputs to the layer. This scaling helps maintain a similar scale of activations and gradients across layers at the start of training. Biases are typically initialized from a normal distribution with a small scale factor to break symmetry while keeping activations in a reasonable range:

$$b_j^l = 0 \text{ or } b_j^l = 0.01. \tag{13}$$

This small random initialization for biases helps prevent all neurons in a layer from developing identical behavior during early training.

While Project 1 introduced regularization in the context of Ridge regression, neural networks benefit from similar techniques to prevent overfitting. The most common approach is L2 regularization (weight decay), which adds a penalty term to the cost function:

$$\begin{aligned} C(\theta) &= \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\theta) \\ &\to \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\theta) + \lambda ||w||_2^2 \\ &= \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\theta) + \lambda \sum_{ij} w_{ij}^2. \end{aligned} \tag{14}$$

Where λ is the regularization parameter, N is the number of training samples, and the sum runs over all weights in the network. This modification encourages the network to use smaller weights and distribute the learned patterns across multiple nodes rather than relying too heavily on any single connection.

3. Activation Functions

The non-linear activation functions are fundamental components that enable neural networks to approximate complex functions. By introducing non-linearity between layers, these functions allow the network to learn and represent patterns that would be impossible with purely linear transformations. The careful selection of activation functions significantly impacts both the network's expressive capability and its training dynamics.

An activation function $f(\cdot)$ takes the weighted sum z of a node's inputs and transforms it into an output signal $a = f(z)$. For our implementations we consider the following activation functions.

Identity: The identity function, simply defined as

$$f(z) = z,$$

passes its input unchanged. While this function is rarely used in hidden layers due to its linearity, it serves as the standard activation function for regression tasks in the output layer.

Sigmoid: The Sigmoid function, defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{15}$$

maps any real input the the interval $(0, 1)$. This smooth, continuously differentiable function has historically been popular due to its biological inspiration and interpretable output range that naturally corresponds to probability-like quantities. However, it suffers from several drawbacks:

- For inputs with large magnitudes, the function becomes nearly flat, which can complicate the training process
- Its output is not zero-centered, which can introduce systematic bias during training.
- The exponential computation is relative expensive.

Despite these limitations, the sigmoid remains useful in the output layer for binary classification tasks where probability interpretation is desired.

Rectified Linear Unit (ReLU): The ReLU function, defined as

$$f(z) = \max(0, z), \tag{16}$$

has become the default choice for many neural network architectures. As it is a very simple threshold function, it is computationally efficient. It enables sparse activation, as all negative inputs are mapped to exactly zero. However, it suffers from the ”dying ReLU” problem, where neurons can become inactive and stop learning if they receive consistently negative inputs.

Leaky Rectified Linear Unit (Leaky RELU): To address the ”dying ReLU” problem, the Leaky RELU function introduces a small slope for negative inputs:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}, \tag{17}$$

where α is a small constant (typically 0.01). This modification prevents complete deactivation of neurons while maintaining the computational efficiency of the standard ReLU. By allowing a small, non-zero output for negative inputs, Leaky ReLU ensures that neurons can continue to participate in the learning process even when receiving predominantly negative inputs.

Softmax: Used primarily in the output layer for multi-class classification, the softmax function generalizes the sigmoid to multiple classes:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{18}$$

Where K is the number of classes. The softmax function produces a probability distribution, I.e. the output values sum to 1, which is useful for interpreting the network's output as class probabilities.

The choice of activation function is a critical design decision that can significantly impact the network's performance. For the hidden layers, the ReLU and Leaky ReLU functions are often preferred due to their computational efficiency and generally better training performance than sigmoid. Cite While the activation function for the output layer depends on the task, the softmax function is commonly used for multi-class classification tasks, while the sigmoid function is suitable for binary classification problems and in regression problems one often uses the identity function.

4. Backpropagation

The backpropagation algorithm provides an efficient method for computing gradients in neural networks, enabling the network to learn from its errors by adjusting weights and biases. While the forward pass computes predictions, backpropagation determines how each parameter should be adjusted to reduce the prediction error. The algorithm derives its name from the way it propagates error gradients backward through the network, from the output layer to the input layer.

To understand backpropagation, we start with a cost function C that measures the network's prediction error. For regression tasks, this is typically the mean squared error (MSE). The goal is similar to that of the gradient descent algorithms: to minimize the cost function. To achieve this, we compute the gradient of the cost function with respect to the network's parameters, the weights $\frac{\partial C}{\partial w_{jk}^{l}}$ and $\frac{\partial C}{\partial b_j^l}$ for each weight and bias in the network.
The key insight behind backpropagation is the chain rule, which allows us to decompose these calculations into smaller, more manageable steps. We introduce an intermediate quantity, the error term δ_j^l, which represents the error in node j in layer l.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}. \tag{19}$$

Where z_j^l is the weighted input to the activation function for node j in layer l. This error term represents how a change in the node's input affects the overall cost.

For the output layer, the error term is directly computed:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = (a_j^L - y_j) f'(z_j^L). \tag{20}$$

Where $f'(\cdot)$ is the derivative of the activation function. For the hidden layers, the error term is recursively computed based on the error terms of the subsequent layer:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l). \tag{21}$$

This equation shows how errors propagate backward: each node's error is a weighted sum of the errors in the next layer, scaled by the local derivative of its activation function.

Once we have computed these error terms, the gradient for any weight or bias follows a simple pattern:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l, \tag{22}$$

and

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{23}$$

In these equation the simplicity of the gradient expressions is evident. The gradient for a weight is the product of the error at its target node and the activation from its source node, while the gradient for a bias is simply at the error at its node.

During backpropagation, regularization terms are also added to the weight gradients to prevent overfitting, while the bias gradients remain unchanged. This selective regularization of weights but not biases reflects the different roles these parameters play in the network: weights determine the importance of connections, while biases set activation thresholds.

The complete backpropagation algorithm can then be summarized in four steps:

- Forward Pass: Compute activations for all layers
- Output Error: Calculate error terms δ^L for the output layer
- Backpropagate: Compute error terms δ^l for each hidden layer
- Gradient Computation: Calculate the gradients using the error terms

This process provides the gradients needed for the various optimization methods discussed earlier to update the network's parameters. The efficiency of backpropagation comes from its clever reuse of intermediate calculations through the chain rule, avoiding redundant computations that would occur in a more naive approach to gradient calculation.

5. *Learning Rate Tuning*

The learning rate tuning methods discussed in section II B 2 for gradient descent optimization are equally applicable to neural network training. The same challenges of balancing convergence speed with stability apply, and methods like AdaGrad, RMSprop, and Adam are commonly used to automatically adjust learning rates during neural network training.

D. Training Considerations for Optimization Methods

While gradient descent and neural networks may appear quite different at first glance, they share many fundamental training elements and challenges. Understanding these commonalities helps illuminate the underlying principles of machine learning optimization

1. *Cost Functions and Optimization Goals*

Both methods aim to minimize a cost function that measures prediction error. For regression tasks, we prefer the MSE discussed in Project 1 because its quadratic form makes it differentiable and provides stronger penalties for large errors. However, binary classification tasks often employ the logistic regression cost function:

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)], \tag{24}$$

where p_i represents the predicted probability of class 1 for observation i. This function is particularly well-suited for classification as it naturally handles probabilistic predictions and automatically enforces outputs between 0 and 1.

2. *Batch Processing and Stochasticity*

Both gradient descent and neural networks benefit from stochastic updates using mini-batches. This approach offers several advantages:

- Reduced memory requirements compared to full-batch methods
- More frequent parameter updates
- Introduction of beneficial noise that can help escape local minima
- Potential for better generalization

The choice of batch size involves similar tradeoffs in both cases: smaller batches provide more frequent updates but noisier gradient estimates, while larger batches give more stable gradients but slower convergence. The optimal batch size often depends on both the problem structure and computational constraints.

3. *Training Progress*

Both methods typically measure progress in terms of epochs - complete passes through the training data. The number of epochs needed depends on factors like dataset size, model complexity, and the difficulty of the task. Training continues until either a maximum number of epochs is reached or some convergence criterion is met. However, the interpretation of ”convergence” can differ between methods - neural networks may require more epochs due to their non-linear nature and larger parameter space.

4. *Hyperparameter Selection*

Both approaches require careful tuning of hyperparameters that control the learning process:

- Learning rates and their scheduling
- Regularization strength λ
- Optimization algorithm parameters (momentum, decay rates, etc.)
- Batch size and number of epochs

These hyperparameters often interact in complex ways. For example, the optimal learning rate typically decreases with batch size to compensate for more accurate gradient estimates. Similarly, stronger regularization (higher λ) may require more epochs or higher learning rates to reach convergence. Understanding these interactions is crucial for both gradient descent and neural network optimization.

The common thread through all these considerations is the fundamental challenge of optimization: balancing computational efficiency, model accuracy, and generalization ability. Whether using simple gradient descent or complex neural networks, success depends on carefully managing these tradeoffs through proper choice and tuning of training parameters. This understanding provides a unified framework for approaching machine learning optimization, regardless of the specific method employed.

III. METHODS & IMPLEMENTATION

IV. RESULTS & DISCUSSION

V. CONCLUSION

Appendix A: Code

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