Enhancing Data Resolution with Deep Neural Networks

Saurav Singh Chandel[†]

[†] *Department of Mathematics and Statistics, Memorial University of Newfoundland, St. John's (NL) A1C 5S7, Canada*

E-mail: sschandel@mun.ca

Abstract: This research explores the multifaceted world of neural networks and optimization algorithms. We delve into the design and analysis of optimization algorithms for single and two-parameter loss functions, investigating the impact of loss function modification and learning rates. Visualizations in 2D and 3D spaces illuminate the behavior of these optimizers, particularly when confronted with complex landscapes like the Ackley function. Furthermore, we analyze the influence of neural network architecture, activation functions, loss functions, and optimizers on upscaling a complex function. Our findings demonstrate the importance of selecting appropriate architectures and optimization strategies to achieve optimal results. This research contributes to the field of machine learning by providing insights that can guide practitioners in the effective design and training of neural networks, ultimately leading to the development of more powerful applications.

Keywords: Neural networks, optimization, gradient descent, adaptive learning rates, feed-forward networks, upscaling, visualization

1 Introduction

1.1 Understanding Deep Learning

Neural networks, inspired by the intricate workings of the biological brain, represent a powerful tool within the field of machine learning. Their ability to approximate complex functions and learn intricate patterns from data makes them remarkably versatile across a wide range of applications. This research investigates two interconnected elements crucial to the success of neural networks: the techniques employed for function upscaling and the optimization algorithms that drive their training process.

Function upscaling with neural networks involves enhancing the resolution or complexity of a mathematical function. We will explore how factors like network architecture (number of layers, neurons), the quality and characteristics of the training dataset, and the choice of activation functions significantly affect the neural network's ability to perform upscaling tasks effectively. Understanding these influences offers insights into optimizing neural network design for specific function-related challenges.

The core of neural network training lies in optimization algorithms. These algorithms iteratively refine the internal parameters (weights and biases) of the network to minimize errors and achieve superior performance. We will dissect a range of prominent optimization algorithms, including Stochastic Gradient Descent (SGD), Adam, and others, examining their unique mechanisms and behaviors. This analysis will focus on how different learning rates, adaptive capabilities, and convergence properties influence the network's training trajectory.

Our combined examination of function upscaling and optimization has a twofold purpose. Firstly, we aim to illuminate how optimization choices directly impact performance in function upscaling tasks. Secondly, we intend to create a foundation for further research into the intricate dynamics of neural networks. This research seeks to provide practitioners with a clearer understanding of how these elements interact, allowing them to select more effective strategies for designing, training, and deploying neural networks in diverse real-world applications. [4]

1.2 Applications of Deep Learning Neural Networks

Classification: Neural networks for identifying or categorizing data.

- **Neural Network Types:**
  - Convolutional Neural Networks (CNNs): Dominant in image classification tasks due to their ability to extract spatial features efficiently.
  - Recurrent Neural Networks (RNNs): Well-suited for sequential data like text or time series. Applications include natural language processing and sentiment analysis.
  - Transformers: Recent state-of-the-art architectures, achieving incredible results on language tasks and image classification.
- **Use Cases:**
  - Image Classification: Identifying objects in photos (e.g., medical diagnosis, self-driving cars, product categorization).
  - Text Classification: Categorizing documents (news articles, emails), detecting spam, and sentiment analysis of social media posts.

  - Audio Classification: Recognizing specific sounds or spoken words (e.g., virtual assistants, music categorization).

Estimation: Neural networks for predicting values or behaviors.

- **Neural Network Types:**
  - Feedforward Neural Networks (FNNs): Versatile for various estimation tasks. Structure can be tailored to the problem's complexity.
  - Recurrent Neural Networks (RNNs): Excel at tasks requiring understanding of sequences, such as time series prediction or forecasting.
  - Generative Adversarial Networks (GANs): Can generate new data that mimics a given training dataset, leading to innovative estimation applications.
- **Use Cases:**
  - Image Upscaling: Improving the resolution and detail of low-resolution images (e.g., photo enhancement, old video restoration).
  - Missing Data Prediction: Completing partial input data or restoring missing values (e.g., incomplete sensor data, damaged images).
  - Behavior Prediction: Forecasting future outcomes based on past patterns (e.g., stock market prediction, weather forecasting, customer behavior analysis).

Important Considerations

- Data Nature: The choice of neural network architecture heavily depends on the type of data (images, text, audio, time series).
- Task Complexity: More complex tasks might necessitate deeper or more specialized neural network architectures.
- Accuracy vs. Interpretability: Some applications prioritize accuracy (e.g., medical diagnosis), while others require greater model explainability (e.g., financial decision-making).

1.3 Components of Neural Networks

1.3.1 Neurons

We can think of each neuron as a single mathematical function defined as $\boldsymbol{\phi}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{B})$.

- $\boldsymbol{x}$ is the input to the neuron from the previous neuron. The output from the previous neuron is used as the input for the next neuron.
- $\boldsymbol{W}$ is the linear multiplier of the input and $\boldsymbol{B}$ is the offset of the expression before it is passed into the activation function.
- $\boldsymbol{\phi}$ is the activation function defined for that layer. Activation functions normalize the inside expression and send their output to the next neuron.

We will discuss more about different activation functions later.

A neuron is the smallest functional unit of a neural network; each neuron in a layer is connected to all the neurons in the previous and next layer. They work similar to long chains whose weights and biases can be altered and optimized to get the desired results.

1.3.2 Layers

Neural networks are hierarchical structures where information is processed layer-by-layer. Each layer comprises interconnected processing units called neurons. These layers act as building blocks, progressively transforming the input data. Early layers typically extract low-level features from the input data. As information travels through subsequent layers, these features are combined and become progressively more complex, culminating in the network's final output. The specific types of layers employed depend on the nature of the data and the task at hand.

Specialized Layers for Specialized Tasks: Convolutional and Pooling Layers

In the realm of Artificial Neural Networks (ANNs), fully connected layers are commonly used. These layers connect every neuron in one layer to all neurons in the next, allowing for complex feature extraction across the entire input. However, for specific tasks involving grid-like data, such as image recognition, Convolutional Neural Networks (CNNs) leverage specialized layer types. Convolutional layers are designed to efficiently process such data. They employ filters that slide across the input, analyzing small localized regions and extracting relevant features. These features might be edges, shapes, or textures in the case of image recognition. Following convolutional layers, pooling layers are often used for dimensionality reduction. Pooling layers summarize information from smaller regions of the data, reducing computational costs and promoting a degree of invariance to slight shifts or rotations in the input. We'll explore CNNs, featuring these specialized layers, further ahead in the paper.

1.3.3 Activation Functions

Activation functions wrap around a neuron, taking weights and biases as parameters and producing a normalized output, usually in the interval [0, 1]. They play a crucial role in deep learning by: [7]

- **Introducing Non-Linearity:** Enabling neural networks to model complex relationships in data.
- **Decision-Making:** Influencing the network's overall decision based on neuron activation levels.
- **Gradient Flow:** Impacting how gradients propagate during backpropagation, affecting the model's learning process.

Common Activation Functions

- **ReLU (Rectified Linear Unit):**

  $\boldsymbol{f(x) = \max(0, x)}$

  Pros:

  – Computationally efficient due to its simple thresholding operation.

  – Helps avoid the vanishing gradient problem in deep networks, promoting faster training.

  Cons:

  – Can lead to the "dying ReLU" problem: Neurons with negative inputs may become permanently inactive, hindering learning.

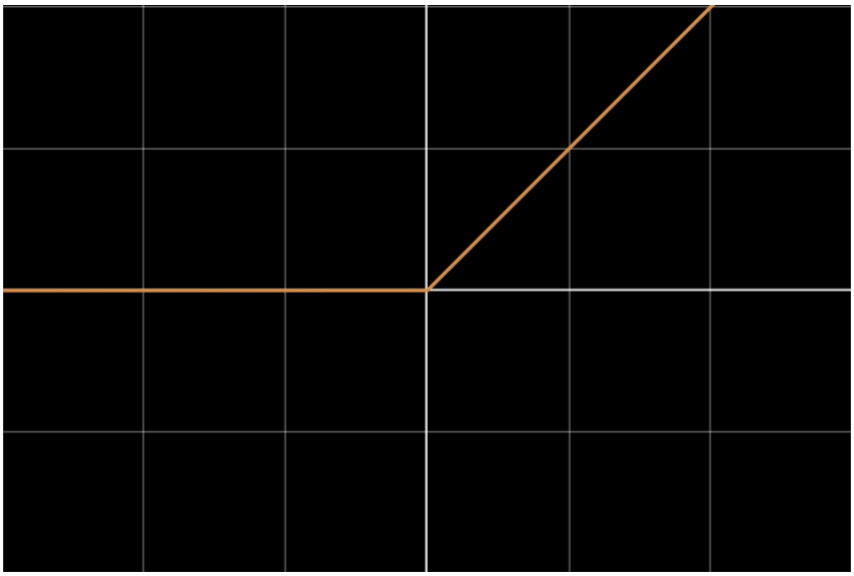


Figure 1. Graph of the ReLU activation function

- **Sigmoid:**

  $\boldsymbol{f(x) = 1/(1 + e^{-x})}$

  Pros:

  – Maps output smoothly between 0 and 1, making it historically useful for modeling probabilities or interpreting output as confidence scores.

  Cons:

  – Prone to vanishing gradients, especially when inputs fall far from zero, leading to slow or stalled learning.

  – Computationally a bit more expensive than ReLU due to the exponential calculation.

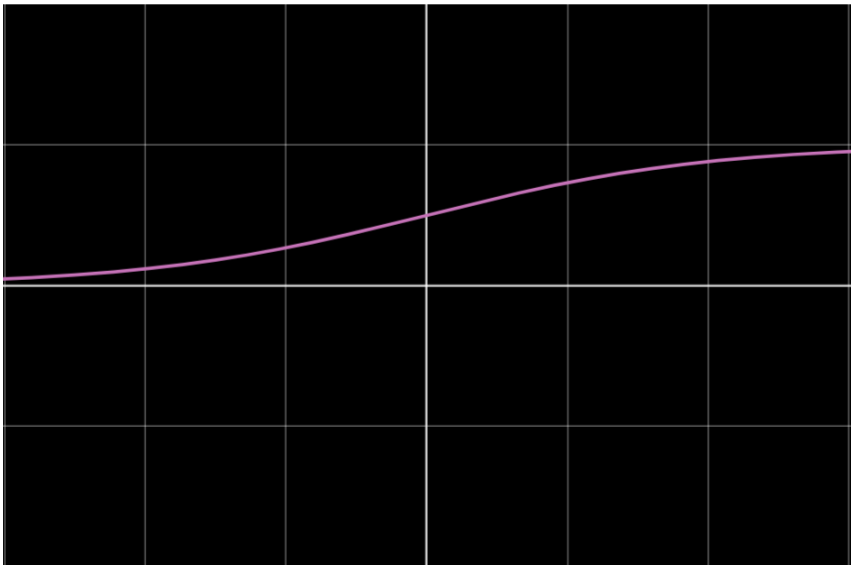


Figure 2. Graph of the Sigmoid activation function

- **Tanh (Hyperbolic Tangent):**

  $$f(x) = (e^x - e^{-x})/(e^x + e^{-x})$$

  Pros:
  – Zero-centered output, sometimes aiding in faster convergence.
  – Less extreme gradient saturation compared to Sigmoid, especially near zero.

  Cons:
  – Still susceptible to the vanishing gradient problem, although less severe than Sigmoid.
  – Slightly more computationally intensive than ReLU.

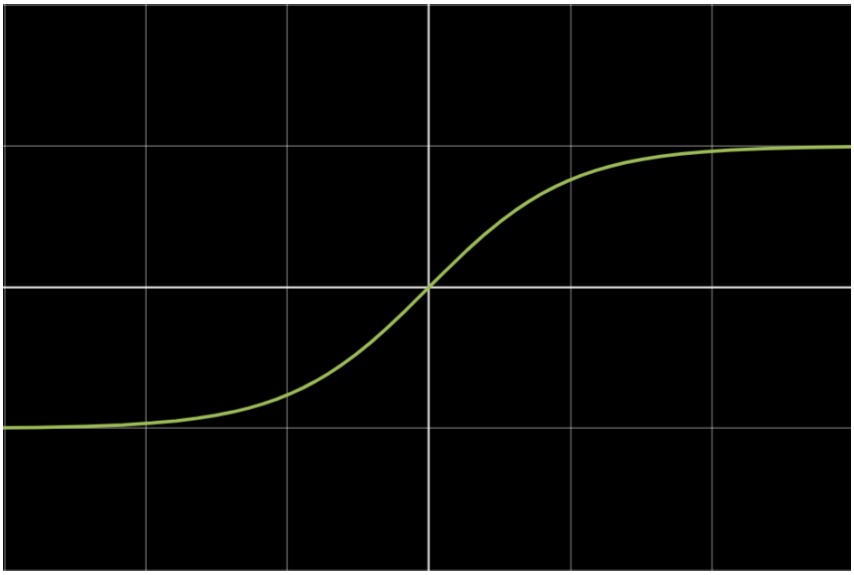


Figure 3. Graph of the Hyperbolic Tangent activation function

- **Leaky ReLU:**

  $\boldsymbol{f(x) = \max(0.01x, x)}$

  Pros:

  – Addresses the "dying ReLU" problem by allowing a small, non-zero gradient for negative inputs, enabling continued learning.

  – Shares the efficiency benefits of ReLU.

  Cons:

  – Slightly increased computation compared to ReLU due to the leaky negative part.

  – The effect of the leak might need tuning as the optimal value can vary.

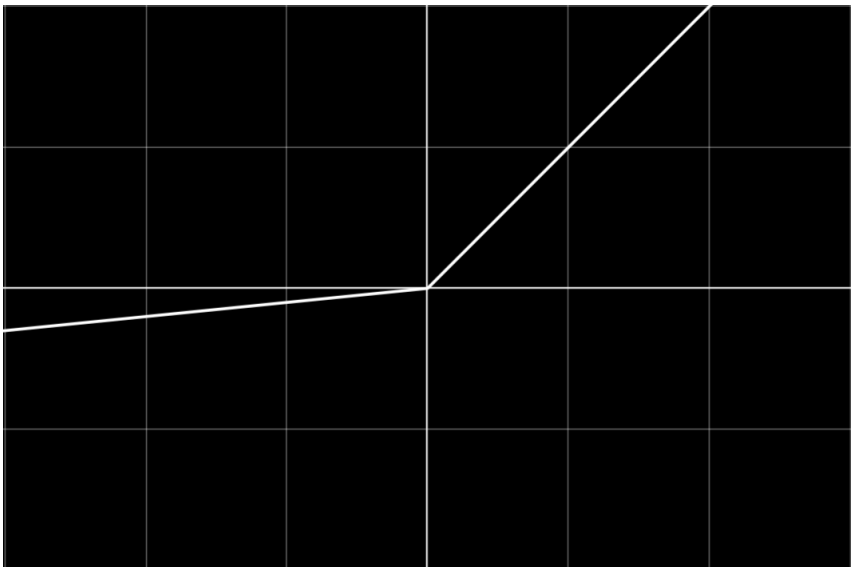


Figure 4. Graph of the Leaky ReLU activation function

Choosing an Activation Function: Consider the task type, layer position, and experiment to find the best fit for your neural network.

- **Task Type**

  Classification (Output Layer):

  – Binary Classification: Sigmoid (output between 0 and 1 represents probability of a class).

  – Multi-Class Classification: Softmax (generalizes Sigmoid for multiple classes, providing probabilities across them).

  Regression (Output Layer):

  – Linear: Can be used if outputs aren't bounded.

  – Sigmoid or Tanh: If you need to squash the output within a range (e.g., predicting values between 0 and 1).

  Image Processing (Hidden Layers):

  – ReLU: Common due to its efficiency and mitigation of vanishing gradients.

  – Leaky ReLU: To address the "dying ReLU" problem.

- **Layer Position**

  Hidden Layers:
  – ReLU and variants: Promote efficient learning and avoid vanishing gradients in deep networks.

  Output Layer:
  –Task-specific choices, as outlined above for classification and regression.

- **Experimentation**

  No Single "Best" Function:
  – The optimal activation function is often problem-dependent.

  Start with Common Choices:
  – Begin with ReLU (hidden layers), Sigmoid/Softmax (classification outputs), depending on your task.

  Try Alternatives:
  – Experiment with Leaky ReLU, Tanh, or others if you encounter issues like vanishing gradients or slow convergence.

  Empirical Evaluation:
  – Compare the performance of your neural network with different choices to find what works best for your data and problem.

- **Illustrative Use Cases**

  Image Classification (Cat vs. Dog):
  – Hidden Layers: ReLU or Leaky ReLU
  – Output Layer: Sigmoid (binary classification)

  Predicting Stock Prices:
  – Hidden Layers: ReLU or its variants.
  – Output Layer: Likely linear (unbounded output), or Tanh (if you want to constrain predictions within a range).

  Sentiment Analysis (Positive, Negative, Neutral):
  – Hidden Layers: ReLU or its variants.
  – Output Layer: Softmax (3 classes)

1.3.4 Loss Functions

Loss functions play a pivotal role in the training process of neural networks. They serve as a compass, guiding the network towards more accurate predictions by quantifying the discrepancy between the calculated output and the desired true output. The choice of loss function directly impacts the optimization process, making it crucial to select a function that aligns well with the nature of your problem.

[5]

Common Loss Functions

- **Mean Squared Error (MSE):** A popular choice for regression problems, calculated as the average of squared differences between true and predicted values.

$$MSE = (1/n) * \sum (y_{true} - y_{predicted})^2$$

- **Mean Absolute Error (MAE):** Another regression loss function, calculated as the average of absolute differences between true and predicted values.

$$MAE = (1/n) * \sum |y_{true} - y_{predicted}|$$

- **Binary Cross-Entropy:** Widely used for binary classification tasks. It effectively penalizes incorrect predictions, making it suitable for scenarios where predictions represent probabilities.

$$-(y_{true} * \log(y_{predicted}) + (1 - y_{true}) * \log(1 - y_{predicted}))$$

- **Categorical Cross-Entropy:** A generalization of binary cross-entropy, designed to handle multi-class classification problems where a sample can belong to only one of several classes.

$$CE = -\sum_{i=1}^{C} y_{true,i} \log(y_{predicted,i})$$

Where:
$*$ C is the number of classes
$*$ $y_{true,i}$ is a binary indicator (0 or 1) of whether class label i is the correct classification for the observation.
$*$ $y_{predicted,i}$ is the predicted probability that the observation belongs to class i.

Choosing a Loss Function

- **Problem Type:**
  − Regression: MSE, MAE − Classification: Cross-entropy variants

- **Robustness to Outliers**
  − MAE is less affected by outliers compared to MSE, as large errors are not amplified as drastically.

- **Desired Learning Behavior**
  Cross-entropy strongly penalizes incorrect predictions, potentially leading to faster convergence in some cases.

- **Data Distribution**
  − Imbalanced Classes: Consider weighted variants of cross-entropy or focal loss to give more importance to examples from underrepresented classes.
  − Outliers: If your data has many outliers, MAE might be more suitable due to its robustness.

- **Convergence Behavior**
  − Vanishing Gradients: In certain scenarios, cross-entropy loss can lead to slower convergence when predictions are far from the true labels. Consider scaling techniques or try

alternative loss functions.
– Stability: If you encounter erratic training behavior, explore loss functions known for smoother updates.

- **Interpretability**
  – Probabilistic Meaning: For tasks requiring confidence scores or well-calibrated probabilities, cross-entropy loss is often preferable.
  – Relative Errors: If the magnitude of the error itself is less important than the direction (over- or under-estimation), MSE or MAE might be suitable. [2]

Beyond Common Choices

- Custom Loss Functions: For highly specialized problems, you might need to design a custom loss function that directly optimizes the most important metrics for your use case.
- Huber Loss Combines the advantages of MSE (smoothness) and MAE (robustness to outliers) with a tunable parameter for flexibility.

Practical Tips

- Start with Defaults: Begin with established loss functions suitable for your problem type (MSE/MAE for regression, cross-entropy for classification).
- Experiment: If you face issues with optimization or performance, evaluate alternative loss functions.
- Evaluate on Your Data: The best loss function depends on how well it helps your model learn the specific patterns and nuances of your dataset.

1.3.5 Optimizers

Optimizers are the engines driving the learning process of neural networks. They employ sophisticated strategies to update the weights and biases of neurons, with the ultimate goal of minimizing the loss function. Gradient descent lies at the heart of many optimizers: they use the gradient of the loss function to determine the direction in which to adjust the parameters.

Preparing for Deepr Analysis
The choice of an optimizer plays a crucial role in how quickly and effectively a neural network can learn. To gain a better understanding of their behavior, we will delve into the following:

- *Analyzing Different Optimizers:* We will set up experiments with custom loss functions to observe the performance of various optimizers under different conditions.
- *Practical Considerations:* We will explore factors influencing optimizer selection, such as convergence speed, robustness to different data characteristics, and their hyperparameters.

2 Analyzing Optimizer Behavior

In this section of the paper, we will delve deeper into the working of optimizers. Our focus will be on analyzing how different optimizers behave in the context of various loss functions. We will explore their convergence properties, speed, and sensitivity to parameters like learning rate. This analysis will provide valuable insights for selecting the most effective optimizers for specific neural network applications.

2.1 Phase 1: Development of Optimization Algorithms

The initial phase of our research was dedicated to the design and creation of optimization algorithms tailored for machine learning tasks. We implemented these algorithms using the Python programming language to ensure flexibility and reproducibility. This phase comprised the following key steps:

2.1.1 Algorithm Design

We began by designing optimization algorithms suitable for optimizing single-parameter loss functions. We mainly focused on six key algorithms *Stochastic Gradient Descent (SGD), Vanilla Momentum, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and Adam.* [8]

2.1.2 Implementation

Following the design phase, we proceeded to implement these algorithms in Python. Due to this research being mainly focused on one parameter loss function, these algorithms were specifically designed according to our needs.

2.2 Phase 2: Empirical Evaluation

The second phase of our research focused on empirically evaluating the performance of the developed optimization algorithms. This phase was vital in understanding how these algorithms would function in practical machine-learning scenarios.

2.2.1 Modification of Loss Function

Initially, we considered the following Loss function: $L(p) = sin(2p) + asin(4p)$
Where p is the parameter of the loss function and a is an arbitrary constant that we alter while experimenting.
We kept $p^{(0)} = 0.75$ and had two different values of $a = 0.499$ and $a = 0.501$

2.2.2 Learning Rate Experiments

To gain insights into the algorithms' robustness and convergence characteristics, we executed experiments using different learning rates. This investigation helped us determine the sensitivity of the algorithms to the learning rate parameter.
We chose Learning rate

$$\eta\varepsilon\{0.1, 0.01, 0.001\}$$

.

2.3 Visualization of 2D Functions

Previously, to better understand the workings of optimizers, we illustrated the optimizers with 1D datasets. Now that we have a basic understanding, we can start by going up in dimensions as that is a better representation of real-world scenarios where we have multiple parameters hence making it higher dimensional.
We start by plotting six functions

- Six-Hump Camel

$$f(x,y) = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + \left(-4 + 4y^2\right)y^2$$

- Michalewicz

$$f(\mathbf{x}) = -\sum_{i=1}^{D} \sin(x_i)\sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

- Sphere

$$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$$

- Ackley

$$f(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i)\right) + 20 + \exp(1)$$

- Shubert

$$f(x,y) = \sum_{i=1}^{5} i\cos\left((i+1)x + i\right)\sum_{i=1}^{5} i\cos\left((i+1)y + i\right)$$

- Branin

$$f(x,y) = a(y - bx^2 + cx - r)^2 + s(1-t)\cos(x) + s$$

on a 3D Grid to see what we are dealing with.

2.3.1 Finding the Minima using Optimizers

Then we modify the previous optimizers we wrote for 1D datasets to process 2D datasets so that we can visualize the trajectory taken by the optimizers. Each of the optimizers is individually run for the previously mentioned 2D functions to illustrate the workings and interpret the results so we can conclude the use-case scenarios and discuss some of their characteristics. For testing purposes, we are going to select a relatively low learning rate of *0.001*.

2.4 Results

2.4.1 1D Functions

This section unveils the findings from our research on optimization algorithms in machine learning. We present the outcomes from the two main phases of our study: algorithm development and empirical evaluation. These results contribute to the discussion on optimization in machine learning and offer practical insights for optimizing strategies.

The following table shows the data from running several experiments with $p^{(0)} = 0.75$, $a = 0.499$, $a = 0.501$:

	Name	Epochs	Learning Rate	Final Loss	Optimized Parameter
2	SGD	151	0.1	8.88178E-16	2.61780115
3	SGD	1574	0.01	8.43769E-14	2.617800892
4	SGD	15677	0.001	9.83658E-14	2.617800201
5	Vanilla momentum	272	0.1	5.81757E-14	2.617801424
6	Vanilla momentum	248	0.01	3.88578E-14	2.617798629
7	Vanilla momentum	1621	0.001	8.63754E-14	2.617801282
8	Nesterov Accelerated Gradient	16	0.1	7.77156E-15	2.617801158
9	Nesterov Accelerated Gradient	150	0.01	6.21725E-14	2.617801127
10	Nesterov Accelerated Gradient	1606	0.001	3.4861E-14	2.61780101
11	AdaGrad	1125	0.1	8.74856E-14	2.617800842
12	AdaGrad	92226	0.01	9.99201E-14	2.61779782
13	AdaGrad	100000	0.001	1.35738E-06	1.29186867
14	RMSProp	100000	0.1	0.001009424	2.666339446
15	RMSProp	262	0.01	1.75415E-14	2.617801129
16	RMSProp	1933	0.001	5.21805E-14	2.6178011
17	Adam	286	0.1	7.23865E-14	2.617801549
18	Adam	1150	0.01	8.21565E-14	2.617801286
19	Adam	6023	0.001	9.74776E-14	2.617800531

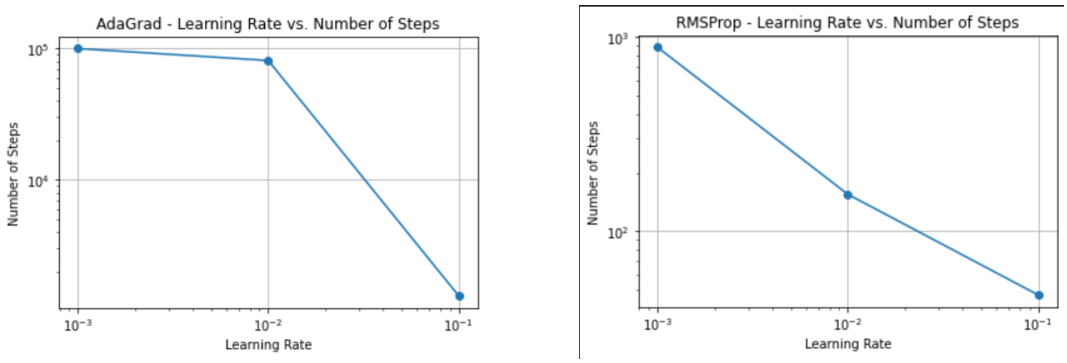
Figure 5. a = 0.499

1	Name	Epochs	Learning Rate	Final Loss	Optimized Parameter
2	SGD	215	0.1	9.79217E-14	1.552555968
3	SGD	1946	0.01	9.95384E-14	1.552550979
4	SGD	16885	0.001	9.9934E-14	1.552535369
5	Vanilla momentum	270	0.1	5.04041E-14	2.618185569
6	Vanilla momentum	272	0.01	3.21965E-14	2.618186157
7	Vanilla momentum	1781	0.001	9.98437E-14	1.552551119
8	Nesterov Accelerated Gradient	16	0.1	1.90958E-14	2.618186064
9	Nesterov Accelerated Gradient	150	0.01	2.90878E-14	2.618186007
10	Nesterov Accelerated Gradient	1808	0.001	9.93164E-14	1.552551122
11	AdaGrad	1312	0.1	9.89278E-14	1.552552339
12	AdaGrad	81134	0.01	9.99964E-14	1.552504844
13	AdaGrad	100000	0.001	1.35136E-06	1.291396571
14	RMSProp	47	0.1	7.53148E-14	1.552557842
15	RMSProp	155	0.01	8.30586E-14	1.552557792
16	RMSProp	886	0.001	9.15865E-14	1.552557737
17	Adam	286	0.1	7.30527E-14	2.618186451
18	Adam	1042	0.01	9.91013E-14	1.552554314
19	Adam	5730	0.001	9.99201E-14	1.552551256

Figure 6. a = 0.501

These tables consist of all the experimental data for the six algorithms used in this research for 3 distinct values of learning rates at the exact value of a.
We can interpret different patterns and behaviors within different algorithms and under other conditions. But before we make an inference, we should also see the graph for the particular set of experiments with $a = 0.501$

We can infer the convergence characteristics concerning the number of steps, as discerned from the graph, reveal crucial insights into the optimization algorithms' behavior. A steep decline in the loss function within the initial steps underscores the algorithms' rapid adaptability. Subsequently, a gradual leveling-off suggests that the optimization process approaches a stable state, indicating the algorithms' ability to converge efficiently. The specific rate of convergence becomes apparent from the slope of this plateau, offering valuable information for practitioners fine-tuning the algorithms for various applications. In analyzing this convergence trend, it becomes evident that optimization algorithms exhibit distinct dynamics in reaching the optimal solutions, facilitating informed decisions on their deployment and parameter configuration.



(a) AdaGrad (b) RMSProp

Figure 7. Learning Rate vs Steps

2.5 2D Functions

In the following section, we present the results of our optimization experiments on a set of diverse functions, including the *Six-Hump Camel, Michalewicz, Sphere, Ackley, Shubert, and Branin functions*. These functions have been carefully chosen for their distinct characteristics, such as multiple local minima, flat regions, and varying degrees of complexity. We utilize a range of optimization algorithms to explore and visualize the trajectory of each optimizer in search of the global minimum. The resulting plots provide valuable insights into the performance of different optimizers on these challenging functions.

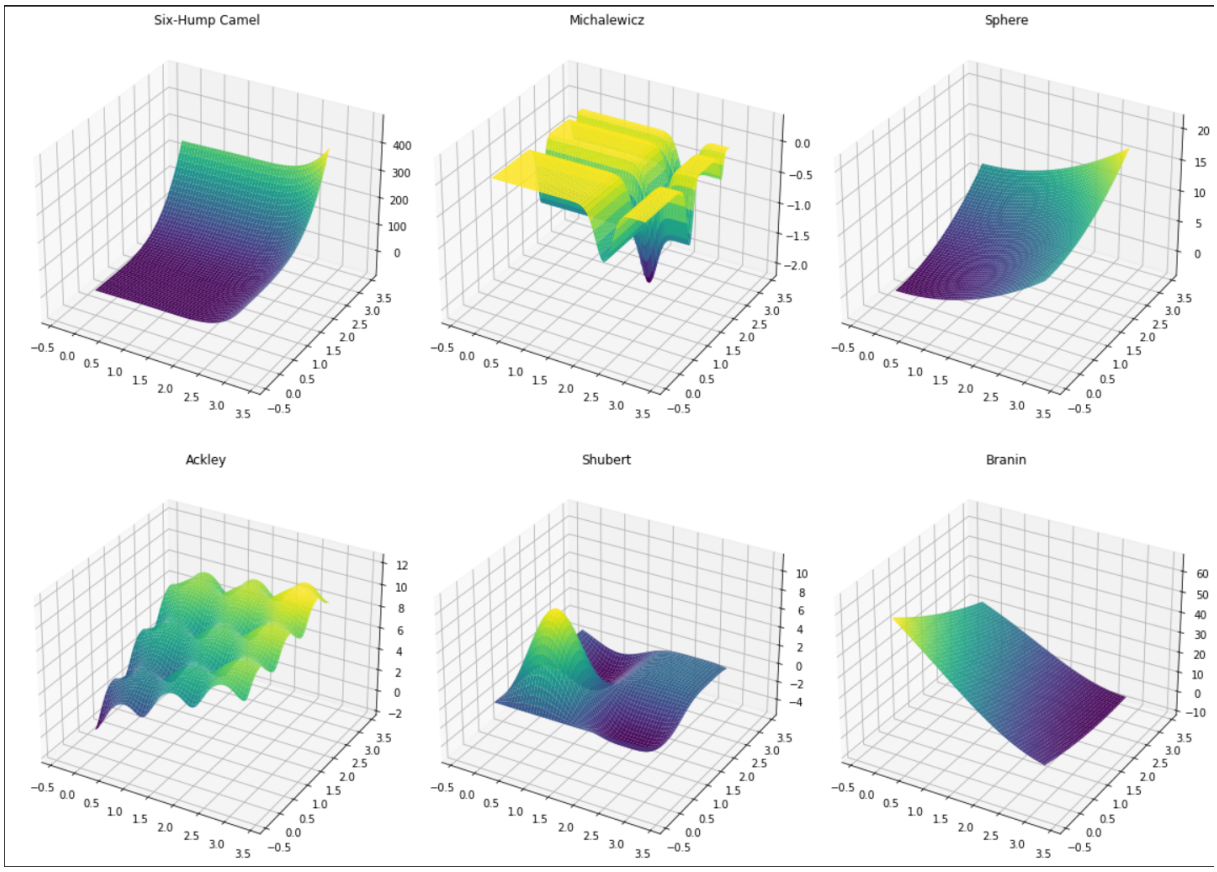


Figure 8. 2D Functions

In the following visualizations, we showcase the intricate landscapes of the *Six-Hump Camel, Michalewicz, Sphere, Ackley, Shubert, and Branin functions*, along with the trajectories mapped by various optimization algorithms. Each function represents a unique optimization challenge, with complex surfaces featuring multiple local minima. Our objective is to illustrate how different optimization techniques navigate these intricate landscapes, providing valuable insights into their ability to converge to the global minimum. These plots vividly display the journeys of optimizers as they search for optimal solutions, shedding light on the effectiveness of each algorithm.

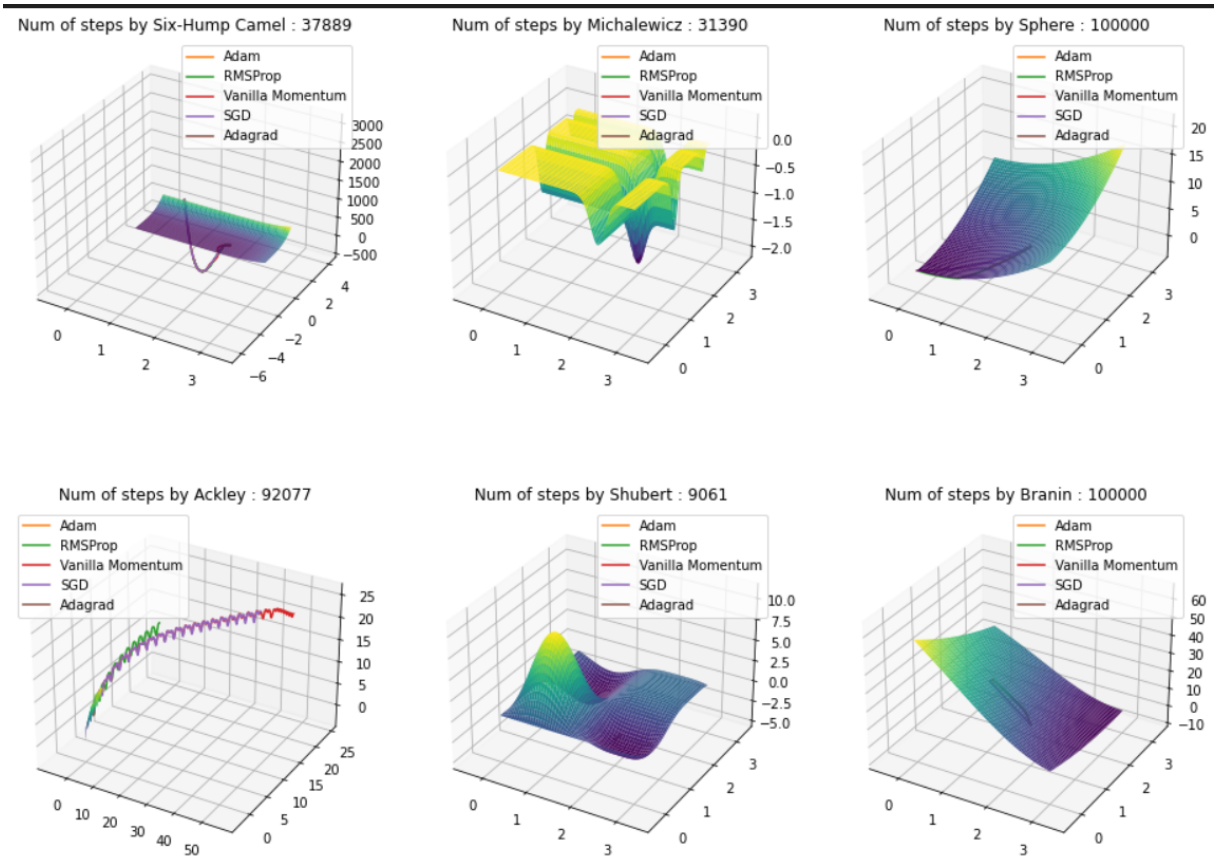


Figure 9. 2D Functions

To offer a more intuitive perspective, we present 2D projections of the *Six-Hump Camel, Michalewicz, Sphere, Ackley, Shubert, and Branin functions*, alongside the paths taken by diverse optimization algorithms. These projections onto the X-Y plane distill the complexity of the 3D landscapes, enabling a clearer visualization of how optimizers traverse these intricate terrains. By examining the trajectory of each optimizer on these 2D representations, we gain a deeper understanding of their ability to navigate and pinpoint global minima within the original 3D functions.

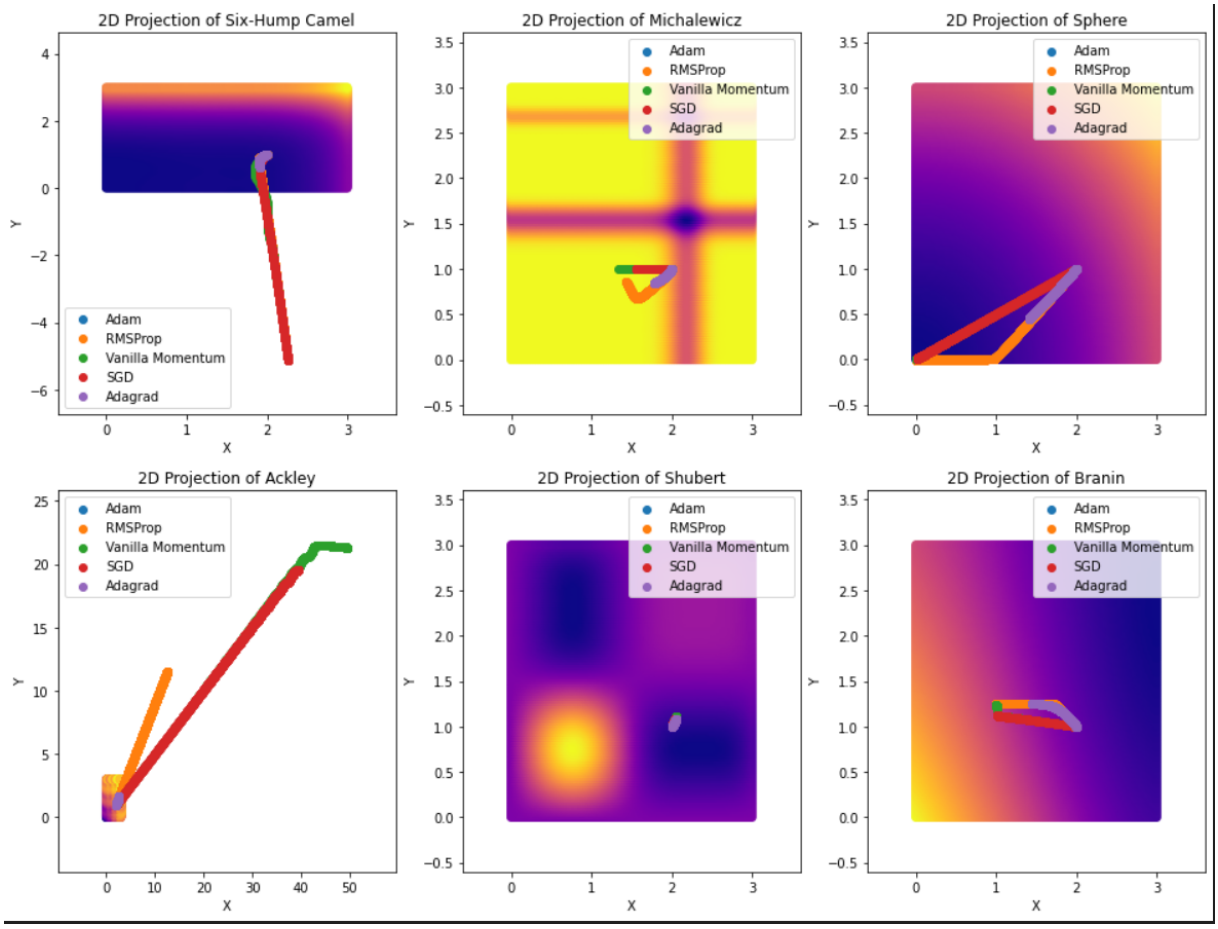


Figure 10. 2D Functions

2.5.1 Challenges with Ackley Function

The Ackley function is a bit tricky for optimizers. It has many small low points scattered around and one deep low point that we want to find. The problem is, that optimizers might get stuck in the small low points instead of reaching the deep one. Also, the Ackley function doesn't provide clear directions on which way to go, making it even more challenging for some optimizers. It's like searching for hidden treasure in a confusing maze.

Because of these difficulties, the Ackley function is often used to test how good optimizers are at finding the best solution in tough situations. It's like a tough obstacle course for optimization algorithms. [1]

2.5.2 Challenges with Sphere Function

A low learning rate has a significant impact on optimization algorithms, particularly in functions with minimal gradient information like the Sphere function. It leads to slower convergence, as algorithms take cautious, small steps during each iteration to avoid overshooting the minimum. However, this cautious approach also makes algorithms more susceptible to getting stuck in local minima, especially if those minima are shallow or wide. Moreover, low learning rates increase sensitivity to noise, raising computational costs and potentially causing premature convergence to suboptimal solutions. Striking the right balance between stability and convergence speed is crucial, as the ideal learning rate varies depending on the specific optimization problem. [3]

2.6 Conclusion

This research has embraced a comprehensive approach to design and empirically evaluate these algorithms, offering valuable insights for both practitioners and researchers.

The systematic development of optimization algorithms tailored for single-parameter loss functions, along with rigorous empirical assessments, has illuminated their adaptability and robustness. Through controlled modifications to loss functions and learning rates, we have gained a nuanced understanding of algorithm performance.

Our research has systematically designed and evaluated optimization algorithms for one-variable loss functions. By carefully adjusting loss functions and learning rates, we've gained valuable insights into algorithm adaptability and performance. We've also visualized how these algorithms behave in two-dimensional spaces, highlighting the challenges posed by complex functions like the Ackley problem. Moreover, our study of the sphere optimization problem revealed that low learning rates can lead to more steps for convergence.

In conclusion, this research contributes to the ongoing discourse surrounding optimization in machine learning. It equips the community with practical knowledge for selecting strategies that enhance model performance. As machine learning continues its rapid evolution, our findings provide a foundation for further advancements and refinements in the optimization field, offering a pathway toward more effective model training and ultimately, more powerful machine learning applications.

3 Analyzing Neural Networks

There are several different types of methods of training neural networks but we are going to discuss *Artifical Neural Networks (ANNs)* and *Convolution Neural Networks (CNNs)* which are one of the most commonly used methods to train neural networks.

3.1 Artifical Neural Network (ANN)

Artificial Neural Networks or also known as *Feed-Forward* neural networks because the inputs are only processed in the forward direction. They are specifically know for their remarkable strength in approximating functions hence they popularly also known as *Universal Function Approximators*. They are capable of learning any non-linear function because of the activation functions. They work best with 1-dimensional vectors. We use many feed forward networks to predict and upscale low resolution data set of a complex function to high resolution. They are not suited for *Image recognition* as that requires the 2-dimensional image to be converted into a 1-dimensional vector which significantly incrases the procesing. Also, if we have a deep neural network with a large number of hidden layers, during backpropogation the Gradient vanishes or explodes therefore giving inconsistent results. [9]

3.2 Convolution Neural Network (CNN)

Convolutional Nerual Networks are specially designed to process grid-based data such as images. They are quite similar to ANNs but one noteable difference being they are not always fully connected to the adjacent layers. They have a special *Convolutional layer* which determines the output of neurons connected to local regions of the input. They also have a *Pooling Layer* which further performs downsampling, reducing the number of parameters. Then they are connected to fully connected layers which works the same as ANNs.
CNNs are made up of neurons in 3-dimensions, height and width which is also called *Spatial*

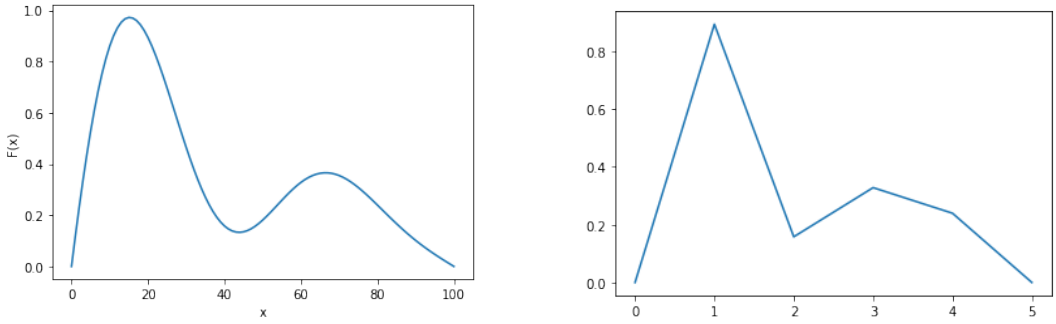
dimension and depth. They use special *Kernels* which are usually small in spatial dimension but deep in depth dimension. [6]

3.3 Results

In this section, we train different *Feed-Forward Neural Networks* which are trained on a low resolution range of a function and have to predict the higher resolution original range.
The function we want to upscale is

$$f(x) = (1 - x)\sin(2x) + (1 - x)^2 \sin(10x)$$

First we will see the orignal graph which is the output of this function on 100 uniformly distributed points in the interval $[0, 1]$

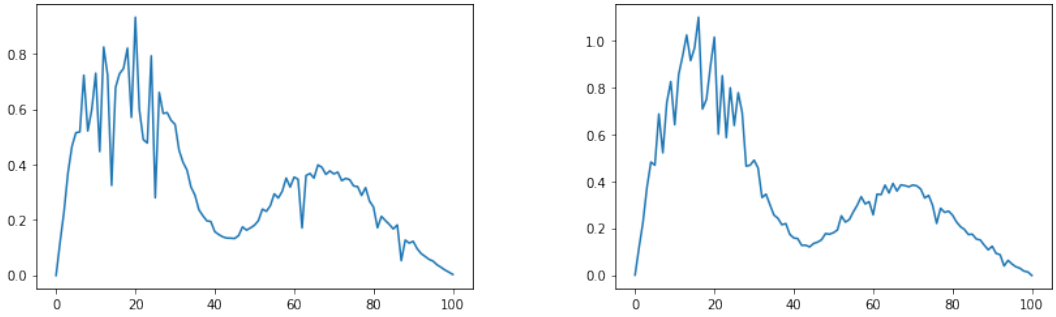


(a) Orignal output of $f(x)$ (b) $f(x)$ at every 20th point

Figure 11. Different resolution datasets

Now we will start by training a basic *Feed Forward Neural Network* with this low resolution data set with no hidden layer and also showing the comparison of adding a hidden layer.

For fair comparison, we are training all the models with Tensorflow.keras.Sequential, for 100 epochs using MeanSquaredError loss function and keras Adam optimizer

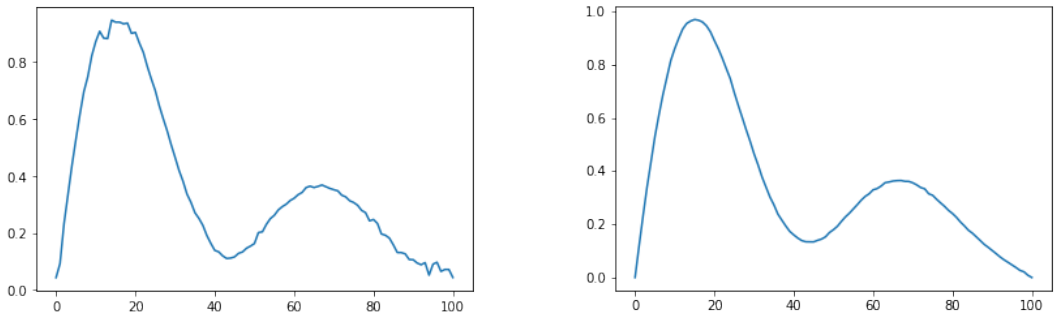


(a) Output: 0 hidden layers (b) Output: 1 hidden layer

Figure 12. Upscaled from every 20th point

As we can observe from the figures, the distortion in the model with 1 hidden layer around the maxima is significantly lower than the model with 0 hidden layers.

Now we can see what happens when we train a model with similar architecture but with a comparitively higher resolution datasets.
Using every 10th and 5th point to train the next models.



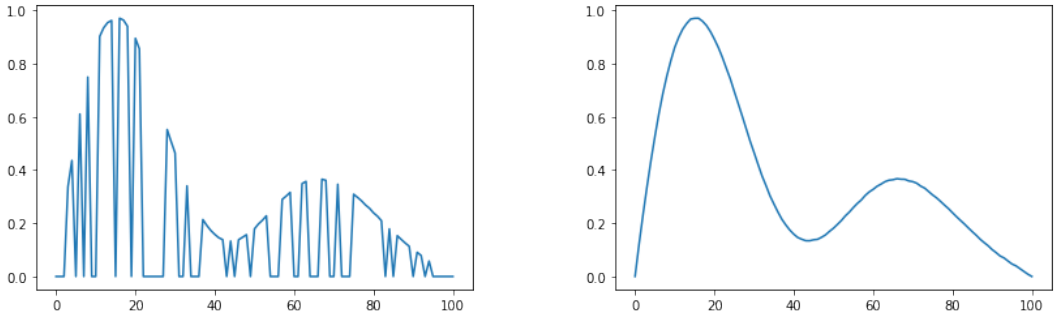
(a) Upscaled from every 10th

(b) Upscaled from every 5th point

Figure 13. Models using different training datasets

As we can observe from the results above, the quality of datasets is directly propotional to the quality of the predicted output.

Now to see the difference between activation functions for output layers.
We are going to use every 2nd point for training dataset, which is almost similar to the original dataset. We might think that since it is so close to the original dataset, activation functions or other parameters will not make a significant difference in the final output.
Moving further, we can examine the difference in *ReLU* and *Linear* activation functions for the output layer.



(a) Model using ReLU (b) Model using Linear

Figure 14. Models using different output layer Activation functions

As we can see, the Model using *ReLU* activation function for the output layer gives us output with significant error.

Why did that happen?

The inconsistency in the results is because *ReLU* activation function gives out 0 as output until the inner expression is positive, which is really helpful for classificatin type deep learning models where the results are discrete. But in our use case, we are predicting a continous function, which means result with minimal error is more desirable for us rather than having a discrete result. Therefore, using *Linear or Sigmoid* activation function works best with our neural network.

We successfully upscaled our dataset with loss of 1.3332×10^{-6} for our best model consisting of 3 layers, 256 nodes in the hidden layer, using a combination of *ReLU* and *Linear* activation function, using *MeanSquaredError* and *Adam* optimizer.

4 Conclusion

This research has delved into the fundamental components of neural networks, from their basic structure and function to in-depth analysis of feed-forward architectures. We systematically designed and evaluated optimization algorithms tailored for single and two-parameter loss functions, carefully modifying loss functions and learning rates to gain a nuanced understanding of algorithm behavior. Visualizations in 2D and 3D spaces further clarified the workings of these optimizers, revealing the challenges posed by complex landscapes. Through experimentation with neural network architectures, we demonstrated the significant impact of factors like dataset choice, hidden layers, activation functions, loss functions, and optimizers on model performance.

Ultimately, this work offers valuable insights into the intricate interplay of neural network design and optimization strategies. Our findings contribute to the ongoing discourse in machine learning, providing a foundation for selecting effective techniques to enhance model performance. As the field continues to evolve, this research can serve as a springboard for further refinement

and innovation in optimization approaches, leading to increasingly powerful machine learning applications.

References

[1] Ackley D., *A connectionist machine for genetic hillclimbing*, vol. 28, Springer science & business media, 2012.

[2] Buda M., Maki A. and Mazurowski M.A., A systematic study of the class imbalance problem in convolutional neural networks, *Neural Networks* **106** (2018), 249â€“259, doi:10.1016/j.neunet.2018.07.011.
URL http://dx.doi.org/10.1016/j.neunet.2018.07.011

[3] Ferreira O., Iusem A. and NÃ©meth S., Sphere optimization, *Top* **accepted** (2014).

[4] LeCun Y., Bengio Y. and Hinton G., Deep learning, *nature* **521** (2015), 436–444.

[5] Lin T.Y., Goyal P., Girshick R., He K. and DollÃ¡r P., Focal loss for dense object detection, 2017, 1708.02002.

[6] O'Shea K. and Nash R., An introduction to convolutional neural networks, *arXiv preprint arXiv:1511.08458* (2015).

[7] Ramachandran P., Zoph B. and Le Q.V., Searching for activation functions, 2017, 1710.05941.

[8] Ruder S., An overview of gradient descent optimization algorithms, *arXiv preprint arXiv:1609.04747* (2016).

[9] Sazli M.H., A brief review of feed-forward neural networks, *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering* **50** (2006).