

Neural Network and Deep Learning with Mathematica

Mohamed M. Hammad



2024

Neural Network and Deep Learning with Mathematica

M. M. Hammad

Department of Mathematics and Computer Science

Faculty of Science

Damanhour University, Egypt

 <https://orcid.org/0000-0003-0306-9719>

 m_hammad@sci.dmu.edu.eg

To
my
mother

Abstract

"Neural Network and Deep Learning with Mathematica" provides a comprehensive examination of the core challenges in neural network research and development, utilizing Mathematica's advanced computational capabilities. Mathematica's integration of symbolic and numeric computation allows for the precise formulation and analysis of neural network models, making it an invaluable tool for researchers and practitioners alike. Readers will benefit from Mathematica's extensive library of built-in functions, which simplifies the construction and training of neural networks, as well as its seamless integration with data visualization, statistical analysis, and image processing tools. The book addresses critical aspects of neural network methodologies, including model generalization, regularization, and optimization. It offers in-depth coverage of multilayer feed-forward neural networks, training processes, and optimization issues such as activation function saturation, vanishing and exploding gradients, and weight initialization. Advanced topics include complex-valued neural networks and sophisticated activation functions, with detailed exploration of their properties and applications. Key methodologies discussed include various learning rate schedules, adaptive algorithms, and techniques for hyperparameter tuning, such as Bayesian optimization and Gaussian processes. With detailed code examples, step-by-step explanations, and illustrative figures, this book is not only a resource for beginners but also a valuable reference for seasoned professionals in fields such as data science, machine learning, and engineering. We have created more than 200 manipulates cover different scenarios in neural networks, more than 500 light Mathematica codes (examples). The code will run as-is with no code from prior algorithms or third parties required beyond the installation of Mathematica. Whether you are a student, researcher, or practitioner, "Neural Network and Deep Learning with Mathematica" equips you with the knowledge and tools to overcome the complexities of modern artificial intelligence and contribute to the ongoing advancements in this exciting field.

Preface

In the rapidly evolving landscape of artificial intelligence and machine learning, neural networks and deep learning have emerged as pivotal technologies, revolutionizing fields ranging from image recognition to natural language processing. As these techniques continue to advance, the challenges associated with their implementation, optimization, and understanding have become increasingly complex. Addressing these challenges requires not only a deep theoretical understanding but also practical tools that can bridge the gap between abstract concepts and real-world applications. This book, *Neural Network and Deep Learning with Mathematica*, is designed to provide a comprehensive exploration of the key challenges faced in neural network and deep learning research and development, alongside practical solutions implemented using Mathematica.

There are several reasons why one might choose Mathematica for neural networks and deep learning:

- **Unified Symbolic and Numeric Computation:** Mathematica's strength lies in its ability to seamlessly integrate symbolic and numeric computations. This allows users to define neural network models using symbolic expressions and directly analyze their properties or gradients before moving to numerical implementation. This unique feature is particularly valuable for researchers who need to explore mathematical formulations or derive analytical results. The symbolic computation capabilities of Mathematica are indeed a significant advantage when working with neural networks and deep learning. This feature allows users to handle mathematical expressions symbolically, enabling them to perform tasks such as analytical derivations, evaluating complex integrals, and solving differential equations that involve their models. By integrating symbolic computation with neural network modeling, Mathematica provides a platform where users can conduct theoretical work with enhanced precision and rigor. This integration not only streamlines the workflow but also ensures that the analyses are more thorough and accurate, making Mathematica an excellent choice for those who require both computational power and mathematical insight in their neural network and deep learning projects.
- **Comprehensive Built-in Functions:** Mathematica provides a vast library of built-in functions for neural networks and deep learning, including functions for defining layers, loss functions, optimizers, and regularization techniques. These functions are highly optimized, ensuring efficient computation without the need for extensive manual coding. Additionally, Mathematica's high-level abstractions simplify the construction and training of neural networks, making it accessible even to those with limited programming experience.
- **Integration with Other Mathematica Features:** Neural networks in Mathematica can be easily integrated with its extensive suite of tools for data visualization, symbolic computation, image processing, and statistical analysis. This integration allows users to preprocess data, visualize model performance, and analyze results within a single environment, streamlining the workflow and enhancing productivity.
- **Automatic Differentiation and Optimization:** Mathematica supports automatic differentiation, which is crucial for training neural networks, as it simplifies the computation of gradients required for optimization. Combined with its powerful optimization algorithms, Mathematica enables efficient training of complex models, even when the underlying mathematical expressions are highly sophisticated.
- **Flexibility and Extensibility:** Mathematica offers a high degree of flexibility in model construction, allowing users to define custom layers, activation functions, and loss functions. This is particularly advantageous for researchers experimenting with novel architectures or those working on specialized applications. Furthermore, Mathematica's programming language, Wolfram Language, is highly extensible, allowing users to incorporate external libraries or frameworks if needed.
- **Interactive Notebooks:** Mathematica's notebook interface provides an interactive environment where users can write code, run simulations, visualize results, and document their work all in one place. This is especially useful for educational purposes, presentations, and collaborative research, as it allows for a clear and coherent presentation of ideas, code, and results.

- **Mathematica's AI Integration:** Mathematica leverages Wolfram's AI and machine learning capabilities, enabling users to perform tasks like data classification, clustering, and predictive modeling with minimal effort. This integration allows users to complement their neural network models with other machine learning techniques, providing a more comprehensive approach to solving complex problems.
- **Cross-platform Deployment:** Mathematica supports deployment across multiple platforms, including desktop, cloud, and mobile. This flexibility allows users to develop and train models locally and then deploy them on a cloud platform for scalability or share them as interactive applications accessible from any device.
- For those already familiar with the Mathematica language, conducting neural network and deep learning analyses can be both straightforward and highly efficient. Mathematica's design emphasizes expressiveness and readability, enabling users to perform even the most complex tasks with concise, elegant code. This streamlined approach not only accelerates the development process but also makes it easier to explore, modify, and refine models. Whether you're defining custom layers, experimenting with new architectures, or optimizing hyperparameters, Mathematica's intuitive syntax and powerful built-in functions simplify the entire workflow, making it an ideal tool for both beginners and experienced practitioners in the field of deep learning.
- **Educational and Research Support:** Mathematica has extensive documentation, tutorials, and a strong user community, which can be invaluable for learning and troubleshooting. Additionally, the Wolfram Demonstrations Project offers a wealth of interactive examples that can serve as inspiration or starting points for new projects.

By choosing Mathematica for neural networks and deep learning, users gain access to a powerful, flexible, and integrated environment that supports the entire lifecycle of model development—from theory and experimentation to deployment and analysis. This makes Mathematica an excellent choice for researchers, educators, and practitioners alike.

Throughout this book, we delve into a wide range of topics, from fundamental principles to advanced techniques, always with an emphasis on overcoming the specific hurdles that practitioners encounter. We explore essential concepts such as model generalization, regularization, and optimization, while also addressing cutting-edge topics like complex-valued neural networks, advanced activation functions, and automated hyperparameter tuning.

This book is designed not only to cater to beginners in the field but also to serve as a valuable reference for seasoned data scientists, machine learning practitioners, biostatisticians, finance professionals, and engineers. Whether they possess prior knowledge of deep learning or seek to fill gaps in their understanding, this book aims to address their needs. We assume that the reader has no prior experience in neural networks and optimization.

Each chapter is designed to build on the previous ones, creating a cohesive and comprehensive guide to artificial neural networks and deep learning. Whether you are a student, researcher, or practitioner, this book will equip you with the knowledge and skills to tackle the challenges of modern artificial intelligence and contribute to the ongoing advancements in this exciting field.

In the first chapter, we begin by laying the foundation. This chapter is an essential primer that introduces the basic concepts, functions, and tools of Mathematica. From arithmetic operations and algebraic manipulations to advanced graphing and functional programming, you will gain a solid understanding of how to utilize Mathematica's capabilities to perform a wide range of computations and visualizations. The chapter also covers the construction of modular and reusable code, which is crucial for developing efficient and scalable neural network models.

As you progress through the book, you will encounter increasingly sophisticated topics, each building on the knowledge gained in the previous chapters. You will explore descriptive statistics and probability theory, which are foundational to understanding data distribution and preparing data for machine learning tasks. The book will guide you through matrix calculus and gradient optimization, providing the mathematical tools needed for model training and fine-tuning.

One of the key strengths of Mathematica is its ability to seamlessly integrate symbolic computation with practical implementation. This is particularly evident in the chapters dedicated to constructing and optimizing multilayer feed-forward neural networks, where you will learn not only the theoretical aspects of neural networks but also how to implement them efficiently using Mathematica's powerful functions like NetChain, NetGraph, and NetTrain.

As neural networks grow in complexity, so do the challenges associated with their optimization. This book dedicates an entire chapter to addressing these challenges, offering strategies for monitoring training progress, mitigating issues such as vanishing gradients, and improving model generalization through advanced regularization techniques. You will also explore adaptive learning rate schedules and gradient descent variants, which are critical for fine-tuning your models and achieving optimal performance.

The book concludes with a deep dive into advanced activation functions and complex-valued neural networks, topics that are at the forefront of current research in deep learning. Here, Mathematica's symbolic and visualization capabilities truly shine, allowing you to explore and understand the behavior of these advanced functions in both two-dimensional and three-dimensional spaces.

Throughout "Neural Network and Deep Learning with Mathematica", you will find detailed explanations, practical examples, and Mathematica code snippets that ensure a hands-on learning experience. By the end of this book, you will be equipped with both the theoretical knowledge and practical skills to tackle complex neural network challenges, making full use of Mathematica's unique computational capabilities.

Finally, we extend our heartfelt thanks to Professor Mohamed Abdalla Darwish, Head of the Department of Mathematics and Computer Science, Faculty of Science, Damanhour University, Egypt, for his unwavering support. We are profoundly grateful to Professor Amr R. El Dhaba for his invaluable discussions and continued encouragement.

We also wish to express our sincere appreciation to our colleagues and friends for their invaluable feedback, thoughtful comments, and constructive suggestions. In particular, we would like to acknowledge Professor Hamed Awad, Dr. Fatma El-Safty, Dr. Hamdy El Shamy, Dr. Mohamed Elhaddad, Mohamed Yahia, Ayman A. Abdelaziz, Eman Farag, Hassan M. Shetawy, Walaa Mansour, Moaz El-Essawey, Aziza Salah, and Eman R. Hendawy for their contributions.

We hope that you find this book informative and inspiring, and that it serves as a valuable resource in your journey through the world of artificial neural networks and deep learning.

knowledge itself is power - Sir Francis Bacon 1597

Egypt 2024

M. M. Hammad

Bridging Theory and Practice: The Dual Approach of This Book

Many books on neural networks and deep learning tend to lean either heavily on theoretical aspects, with dense mathematical formulations, or focus predominantly on computational algorithms, often at the expense of a solid mathematical foundation. This book adopts a balanced approach that bridges the gap between these extremes. By seamlessly integrating theoretical principles with practical implementation, it empowers learners to fully harness the power of neural networks and deep learning in their pursuit of excellence across various domains.

However, attempting to cover both the theoretical concepts and computational algorithms exhaustively within a single volume would be impractical. To ensure a thorough exploration of both aspects, this book is divided into two complementary parts. The first part is titled "Artificial Neural Network and Deep Learning: Fundamentals and Theory." The second part is titled "Neural Network and Deep Learning with Mathematica" For each theoretical chapter in the first part, there is a corresponding chapter in the second part. We strongly recommend that after completing each theoretical chapter, you explore the corresponding practical implementation chapter in the complementary volume. This dual approach will provide you with a well-rounded understanding and the skills necessary to excel in this field.

The book "Neural Network and Deep Learning with Mathematica" adopts a refreshingly code-centric approach, enabling you to solidify your understanding through hands-on practice. Nearly all the concepts introduced are accompanied by illustrative code examples, making the learning experience both practical and tangible. Even the figures in the first part are generated using these code examples, emphasizing the code-first methodology. To ensure accessibility and ease of understanding, the code examples are deliberately crafted in a simple format, prioritizing readability over efficiency and generality. In line with our instructional philosophy, each code example serves a dual purpose: not only does it demonstrate a specific deep learning concept, but it also simultaneously introduces and reinforces Mathematica programming techniques. Readers will learn how to leverage Mathematica to perform complex neural network and deep learning calculations, simulate data, and create visual representations of their findings.

CONTENTS

CHAPTER 1: INTRODUCTION TO MATHEMTICA	1
Unit 1.1. Basic Concepts	2
Unit 1.2. Variables and Functions	12
Unit 1.3. Lists	17
Unit 1.4. 2D and 3D Graphing	28
Unit 1.5. Control Structure	39
Unit 1.6. Modules, Blocks, and Local Variables	50
Unit 1.7. Functional Programming	54
CHAPTER 2: DESCRIPTIVE STATISTICS AND PROBABILITY THEORY	63
Unit 2.1. Descriptive Statistics	66
Unit 2.2. Probability Distributions	86
CHAPTER 3: MATRIX CALCULUS AND GRADIENT OPTIMIZATION	98
CHAPTER 4: MULTILAYER FEED-FORWARD NEURAL NETWORK	133
Unit 4.1. Building Neural Network from Scratch with Mathematica and Universal Approximation Theorem	135
Unit 4.2. Layers: LinearLayer and ElementwiseLayer	160
Unit 4.3. Containers: NetChain and NetGraph	174
Unit 4.4. NetInitialize	190
Unit 4.5. Cost Functions	197
Unit 4.6. NetTrain	206
CHAPTER 5: CHALLENGES IN NEURAL NETWORK OPTIMIZATION	224
Unit 5.1. NetPort and NetExtract	226
Unit 5.2. Monitor Training Progress	233
Unit 5.3. Monitor Saturation and Vanishing Gradients During Training	244
Unit 5.4. NetInitialize (Xavier and Kaiming)	258
Unit 5.5. Feature Scaling: Standardize, Whitenend and Mahalanobis Distances	269
Unit 5.6. BatchNormalizationLayer	277
CHAPTER 6: LEARING RATE SCHEDULES AND GRADIENT DESCENT VARIANTS	298
Unit 6.1. Understanding Learning Rate Schedules and Adaptive Algorithms with Mathematica	300
Unit 6.2. Optimizing Neural Networks with Learning Rate Schedules and Adaptive Algorithms in Mathematica	339
CHAPTER 7: STRATEGIES FOR GENERALIZATION AND HYPER-PARAMETER TUNING	356
Unit 7.1. The Fundamentals of Overfitting: What It Is and Why It Happens	358
Unit 7.2. Performance Metrics	367
Unit 7.3. Gaussian Processes Implementation in Mathematica from Scratch	383
Unit 7.4. Setting up Bayesian Optimization in Mathematica	402
Unit 7.5. Automated Hyperparameter Tuning with Mathematica	420

CHAPTER 8: REGULARIZATION TECHNIQUES	435
Unit 8.1. L2 Regularization	437
Unit 8.2. TrainingStoppingCriterion	442
Unit 8.3. DropoutLayer	456
CHAPTER 9: ADVANCED ACTIVATION FUNCTIONS	465
Unit 9.1. Interactive Exploration of Activation Functions	467
Unit 9.2. Custom Layers in Neural Networks	518
Unit 9.3. Comparison of Some Activation Functions	539
CHAPTER 10: COMPLEX VALUED NEURAL NETWORKS	570
Unit 10.1. Complex Numbers and Functions	571
Unit 10.2. Complex Valued Activation Functions	585

CHAPTER 1

INTRODUCTION TO MATHEMATICA

Mathematica is a powerful computational software program widely used in various fields of science, engineering, and mathematics. It provides a comprehensive environment for performing symbolic and numeric computations, creating visualizations, and solving complex problems. In this introductory chapter, we have covered the basic concepts and functions of Mathematica, including the following topics.

- **Basic Concepts:** Mathematica is built on a foundation of mathematical and computational concepts. It allows you to perform arithmetic operations, manipulate mathematical expressions, and solve equations. The software provides a vast collection of built-in functions for numerical calculations, algebraic manipulations, calculus, linear algebra, and more. These functions serve as fundamental tools for performing various tasks in Mathematica.
- **Variables and Functions:** In Mathematica, variables are used to store values that can be accessed and manipulated throughout a computation. You can assign values to variables using the assignment operator (`:=`) or the equal sign (`=`). Functions, on the other hand, are defined using the syntax: `functionName[arguments]:=functionBody`. Functions encapsulate a series of instructions that can be reused and called with different arguments.
- **Lists:** Lists are an essential data structure in Mathematica. They allow you to store collections of values, such as numbers, strings, or even other lists. Lists can be created using curly braces (`{}`) and elements are separated by commas. Mathematica provides a rich set of built-in functions for manipulating and operating on lists, including appending, deleting, sorting, and extracting elements.
- **2D and 3D Graphing:** Mathematica offers powerful graphing capabilities for visualizing mathematical functions and data in two or three dimensions. You can plot functions using the `Plot` and `ParametricPlot` functions for 2D graphs, and `Plot3D` and `ParametricPlot3D` functions for 3D graphs. These functions allow you to customize various aspects of the plots, such as axes labels, plot ranges, colors, and styles.
- **Control Structures:** Control structures in Mathematica enable you to control the flow of execution in your programs. They include conditionals (`If`, `Switch`), loops (`For`, `While`), and functional programming constructs (`Map`, `Fold`, `Nest`). These control structures allow you to make decisions, iterate over lists or ranges of values, and perform operations on collections of data.
- **Modules, Blocks, and Local Variables:** Mathematica provides mechanisms for creating modular and reusable code through modules and blocks. Modules allow you to encapsulate a group of variables and functions, providing a local scope for their usage. Blocks are similar but are primarily used for scoping variables and expressions dynamically. Local variables defined within modules or blocks are not visible outside their scope, ensuring better code organization and reducing potential conflicts.
- **Functional Programming:** Functional programming leverages its symbolic foundation to provide a uniquely powerful approach to problem-solving. The integration of symbolic data and functions, combined with pattern matching and rule-based programming, enables concise and expressive code for applications.

Unit 1.1

Basic Concepts

Mathematica is a computer algebra system that performs numeric, symbolic, and graphical computations. Although Mathematica can be used as a programming language, its high-level structure is more appropriate for performing sophisticated operations through the use of built-in functions. For example, Mathematica can find limits, derivatives, integrals, and determinants, as well as plot the graph of functions and perform symbolic computations. The number of built-in functions in Mathematica is enormous. Our goals in this introductory chapter are modest. Namely, we introduce a small subset of Mathematica commands necessary to explore Mathematica discussed in this book.

Notebooks

A notebook is a document that allows us to interact with Mathematica. Each notebook is divided up into a sequence of individual units called cells, each containing a specific type of information such as text, graphics, input, or output. Text cells contain information to be read by the user but contain no executable Mathematica commands. The following cell, displaying `In[1] 220`, is an example of an input cell containing executable Mathematica commands. Mathematica computes the value of 2^{20} and the results of the calculation are displayed as `Out[1]:=1048576` in an output cell. When we create a new cell, the default cell type is an input cell. Suppose instead, we want to create a text cell. To do this, use the mouse to click on an area where we want to create a new cell and a horizontal line will appear. Then from the Format menu, select Style and then Text. A new text cell will then be created as soon as we begin typing. We can experiment with creating other types of cells by selecting a cell style of our choice, after first choosing Format and Style from the menu.

Plettes

A palette is similar to a set of calculator buttons, providing shortcuts to entering commands and symbols into a notebook. The name of a useful palette is “Basic Math Assistant Input” and it can be found by selecting the Palettes menu and then Basic Math Assistant. After opening Basic Math Assistant, drag it to the right side of the screen and resize the notebook, if necessary, so that both the notebook and palette are visible in non-overlapping windows. To demonstrate the usefulness of palettes, suppose we wish to calculate $\sqrt{804609}$. The Mathematica command for computing the square root of n is `Sqrt[n]`. The following input cell was created by typing in the information exclusively from the keyboard.

Input `Sqrt[804609]`
Output 897

A quicker and more natural way of entering $\sqrt{804609}$ can be accomplished by clicking on the square root button $\sqrt{\square}$ in the palette and then entering 804609.

Input $\sqrt{(804609)}$
Output 897

Packages

Note that, many of Mathematica functions are available at startup, but additional specialized functions are available from add-in packages. You can load a built-in or installed package in two ways, with the `Needs[]` function or with the symbols `<<`. The package name has quotation marks if you use the `Needs[]` function, but does not have a mark with `<<`. Package names are always indicated with a backward apostrophe at the end of the name, ```.

Input `Needs["PackageName`"]`
Input `<<PackageName``

Help with Mathematica

There are four important tricks to keep in mind to help with Mathematica:

- 1- If you want to know something about a Mathematica function or procedure, just type `? followed by a Mathematica command name, and then enter the cell to get information on that command.`

Input `?FactorInteger`

After you press Enter, Mathematica responds:

Output `FactorInteger[n] gives a list of the prime factors of the integer n, together with their exponents.`

- 2- Mathematica also can finish typing a command for you if you provide the first few letters. Here is how it works:
After typing a few letters choose Complete Selection from the Edit menu. If more than one completion is possible, you will be presented with a pop-up menu containing all of the options. Just click on the appropriate choice.
- 3- If you know the name of a command but have forgotten the syntax for its arguments, type the command name in an input cell, then choose Make Template from the Edit menu. Mathematica will paste a template into the input cell showing the syntax for the simplest form of the command. For example, if you typed `Plot`, and then choose Make Template, the input cell would look like this:

Input `Plot[f,{x,xmin, xmax}]`

- 4- The Wolfram Documentation is the most useful feature imaginable; learn to use it and use it often. Go to the Help menu and choose Wolfram Documentation. A window will appear displaying the documentation home page.

Document the Code

When you write programs in the Wolfram Language, there are various ways to document your code. As always, by far the best thing is to write clear code and to name the objects you define as explicitly as possible. Sometimes, however, you may want to add some "commentary text" to your code, to make it easier to understand. You can add such text at any point in your code simply by enclosing it in matching `(* *)`. Notice that in the Wolfram Language, "comments" enclosed in `(* *)` can be nested in any way.

`(* text *)` a comment that can be inserted anywhere in Wolfram Language code.

Mathematica Examples 1.1

Input	<code>If[a>b,(*then*)p,(*else*)q]</code>
Output	<code>If[a>b,p,q]</code>

Arithmetic Operations

Mathematica can be thought of as a sophisticated calculator, able to perform exact as well as approximate arithmetic computations. You can always control grouping the arithmetic computations by explicitly using parentheses. The following list summarizes the Mathematica symbols used for addition, subtraction, multiplication, division, and powers.

<code>x+y+z</code>	gives the sum of three numbers.
<code>x*y*z, xxy*xz, or x y z</code>	represents a product of terms.
<code>x-y</code>	is equivalent to $x + (-1 * y)$.
<code>x^y</code>	gives x to the power y .
<code>x/y</code>	is equivalent to $x y^{-1}$.

Mathematica Examples 1.2

Input	<code>2.3+5.63</code>
--------------	-----------------------

```

Output 7.93
Input 2.4/8.9^2
Output 0.0302992

Input 2*3*4
Output 24

Input (3+4)^2-2(3+1)
Output 41

```

The precedence of common operators is generally defined so that "higher-level" operations are performed first. For simple expressions, operations are typically ordered from highest to lowest in order: 1. Parenthesization, 2. Factorial, 3. Exponentiation, 4. Multiplication and division, 5. Addition and subtraction. Consider the expression $3 \times 7 + 2^2$. This expression has a value $(3 \times 7) + (2^2) = 25$.

Mathematica has several built-in constants. The three most commonly used constants are π , e , and i . You can find each of these constants on the Basic Math Assistant palette. Some built-in constants are listed below.

I	$(i = \sqrt{(-1)})$.
E	(2.71828) .
Pi	$(\pi = 3.14159)$.

Relational and Logical Operators

Relational and logical operators are instrumental in program flow control. They are used in Mathematica to test various conditions involving variables and expressions. The relational operators are listed below.

lhs==rhs	returns True if lhs and rhs are identical.
lhs!=rhs or lhs≠rhs	returns False if lhs and rhs are identical.
x>y	yields True if x is determined to be greater than y.
x≥y or x≥y	yields True if x is determined to be greater than or equal to y.
x<y	yields True if x is determined to be less than y.
x≤y or x≤y	yields True if x is determined to be less than or equal to y.

Logical operators are used to negate or combine relational expressions. The standard logical operators are listed below.

e₁&&e₂&&...	is the logical AND function. It evaluates its arguments in order, giving False immediately if any of them are False, and True if they are all True.
e₁ e₂ ...	is the logical OR function. It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.
!expr	is the logical NOT function. It gives False if expr is True, and True if it is False.

Mathematica Examples 1.3

```

Input 10<7
Output False

Input Pi^E<E^Pi
Output True

Input 2+2==4
Output True

Input (*Represent an equation:*)
x^2==1+x

```

```

Output  x^2==1+x

Input   (* Returns True if elements are guaranteed unequal, and otherwise stays unevaluated:
*)
a!=b
Output  a!=b

Input  1!=2
Output  True

Input  1>2||Pi>3
Output  True

Input  2>1&&Pi>3
Output  True

Input  (3<5)|| (4<5)
Output  True

Input  (3<5)&&! (4>5)
Output  True

```

Elementary Functions

In the following, we discuss some of the more commonly used functions Mathematica offers. The Wolfram Language has nearly 6000 built-in functions. All have names in which each word starts with a capital letter. Remember that the argument of a function must be contained within square brackets, []. Arguments to functions are always separated by commas.

Common Functions

<code>Log[z]</code>	gives the natural logarithm of z (logarithm to base e).
<code>Log[b,z]</code>	gives the logarithm to base b.
<code>Exp[z]</code>	gives the exponential of z.
<code>Sqrt[z]</code> or <code>✓z</code>	gives the square root of z.
<code>N[expr]</code>	gives the numerical value of expr.
<code>Abs[z]</code>	gives the absolute value of the real or complex number z.
<code>Floor[x]</code>	gives the greatest integer less than or equal to x.

Trigonometric Functions

<code>Sin[z]</code>	gives the sine of z.
<code>Cos[z]</code>	gives the cosine of z.
<code>Tan[z]</code>	gives the tangent of z.

Hyperbolic Functions

<code>Sinh[z]</code>	gives the hyperbolic sine of z.
<code>Cosh[z]</code>	gives the hyperbolic cosine of z.
<code>Tanh[z]</code>	gives the hyperbolic tangent of z.

Numerical Functions

<code>IntegerPart[x]</code>	integer part of x.
-----------------------------	--------------------

<code>FractionalPart[x]</code>	fractional part of x.
<code>Round[x]</code>	integer x closest to x.
<code>Max[x₁,x₂,...]</code>	the maximum of x ₁ ,x ₂ ,...
<code>Min[x₁,x₂,...]</code>	the minimum of x ₁ ,x ₂ ,...
<code>Re[z]</code>	the real part Re z.
<code>Im[z]</code>	the imaginary part Im z.
<code>Conjugate[z]</code>	the complex conjugate z*.

Combinatorial Functions

<code>n!</code>	factorial n(n - 1)(n - 2) ... × 2 × 1.
<code>n!!</code>	double factorial n(n - 2)(n - 4) ... × 3 × 1.
<code>Binomial[n,m]</code>	binomial coefficient $\binom{n}{m} = \frac{(n!)}{(m!(n-m)!)}$.
<code>Multinomial[n₁,n₂,...]</code>	multinomial coefficient (n ₁ + n ₂ +...)/(n ₁ ! n ₂ !).

Mathematica Examples 1.4

Input	<code>Log[10,1000]</code>
Output	3
Input	<code>Exp[I Pi/5]</code>
Output	$E^{(I \sqrt{\pi})/5}$
Input	<code>Sin[Pi/3]</code>
Output	$\frac{\sqrt{3}}{2}$
Input	<code>Sinh[1.4]</code>
Output	1.9043
Input	<code>N[1/7]</code>
Output	0.142857
Input	<code>Floor[2.4]</code>
Output	2
Input	<code>30!</code>
Output	265252859812191058636308480000000
Input	<code>Binomial[n,2]</code>
Output	$\frac{1}{2} (-1+n) n$
Input	<code>Multinomial[6,5]</code>
Output	462

Sum and Product Functions

Sums and products are of fundamental importance in mathematics, and Mathematica makes their computation simple. Unlike other computer languages, initialization is automatic and the syntax is easy to apply, particularly if the Basic Math Assistant Input palette is used. Any symbol may be used as the index of summation. Negative increments are permitted wherever an increment is used.

<code>Sum[f,{i,imax}]</code>	evaluates the $\sum_{i=1}^{i_{\max}} f$.
<code>Sum[f,{i,imin,imax}]</code>	starts with i = i _{min} .
<code>Sum[f,{i,imin,imax,di}]</code>	uses steps di.
<code>Sum[f,{i,{i₁,i₂,...}}]</code>	uses successive values i ₁ , i ₂ , ...
<code>Sum[f,{i,imin,imax},{j,jmin,jmax},...]</code>	evaluates the multiple sum $\sum_{i=1}^{i_{\max}} \sum_{j=j_{\min}}^{j_{\max}} f$.

<code>Product[f,{i,imax}]</code>	evaluates the $\prod_{i=1}^{i_{\max}} f$.
<code>Product[f,{i,imin,imax}]</code>	starts with $i = i_{\min}$.
<code>Product[f,{i,imin,imax,di}]</code>	uses steps di .
<code>Product[f,{i,{i1,i2,...}}]</code>	uses successive values i_1, i_2, \dots .
<code>Product[f,{i,imin,imax},{j,jmin,jmax},...]</code>	evaluates the multiple sum $\prod_{i=1}^{i_{\max}} \prod_{j=1}^{j_{\max}} f$.

Mathematica Examples 1.5

<code>Input</code>	<code>(* Numeric sum: *)</code>
	<code>Sum[i^2,{i,10}]</code>
<code>Output</code>	<code>385</code>
<code>Input</code>	<code>(* Symbolic sum: *)</code>
	<code>Sum[i^2,{i,1,n}]</code>
<code>Output</code>	<code>1/6 n (1+n) (1+2 n)</code>
<code>Input</code>	<code>Sum[1/i^6,{i,1,Infinity}]</code>
<code>Output</code>	<code>$\pi^6/945$</code>
<code>Input</code>	<code>(* Multiple sum with summation over j performed first: *)</code>
	<code>Sum[1/(j^2 (i+1)^2),{i,1,Infinity},{j,1,i}]</code>
<code>Output</code>	<code>$\pi^4/120$</code>
<code>Input</code>	<code>Product[i^2,{i,1,6}]</code>
<code>Output</code>	<code>518400</code>
<code>Input</code>	<code>Product[i^2,{i,1,n}]</code>
<code>Output</code>	<code>(n!)^2</code>
<code>Input</code>	<code>Product[2^(j+i),{i,1,p},{j,1,i}]</code>
<code>Output</code>	<code>$2^{(1/2 p (1 + p)^2)}$</code>

Limit and Series Functions

<code>Limit[expr,x->x0]</code>	finds the limiting value of $expr$ when x approaches x_0 .
<code>Series[f,{x,x0,n}]</code>	generates a power series expansion for f about the point $x=x_0$ to order $(x - x_0)^n$.

Mathematica Examples 1.6

<code>Input</code>	<code>Limit[(Sin[x])/x,x->0]</code>
<code>Output</code>	<code>1</code>
<code>Input</code>	<code>Limit[(1+x/n)^n,n->Infinity]</code>
<code>Output</code>	<code>E^x</code>
<code>Input</code>	<code>(* Power series for the exponential function around x=0: *)</code>
	<code>Series[Exp[x],{x,0,10}]</code>
<code>Output</code>	<code>1+x+x^2/2+x^3/6+x^4/24+x^5/120+x^6/720+x^7/5040+x^8/40320+x^9/362880+x^10/3628800+O[x]^11</code>
<code>Input</code>	<code>(* Power series of an arbitrary function around x=a: *)</code>
	<code>Series[f[x],{x,a,3}]</code>
<code>Output</code>	<code>f[a]+f'[a] (x-a)+1/2 f''[a] (x-a)^2+1/6 f^(3)[a] (x-a)^3+O[x-a]^4</code>
<code>Input</code>	<code>Series[x^x,{x,0,4}]</code>
<code>Output</code>	<code>1 + Log[x] x + 1/2 Log[x]^2 x^2 + 1/6 Log[x]^3 x^3 + 1/24 Log[x]^4 x^4 + O[x]^5</code>

Differentiation Function

$D[f, x]$	gives the partial derivative $\frac{\partial}{\partial x} f$.
$D[f, x, y, \dots]$	gives the derivative $\frac{\partial}{\partial x} \frac{\partial}{\partial y} \dots f$.
$D[f, \{x, n\}]$	gives the multiple derivative $\frac{\partial^n}{\partial x^n} f$.

Mathematica Examples 1.7

```

Input (* Derivative with respect to x: *)
D[x^n,x]
Output n x^(-1 + n)

Input (* Fourth derivative with respect to x: *)
D[Sin[x]^10,{x,4}]
Output 5040 Cos[x]^4 Sin[x]^6 - 4680 Cos[x]^2 Sin[x]^8 + 280 Sin[x]^10

Input (* Derivative with respect to x and y: *)
D[(Sin[x] y)/((x^2+y^2)),x,y]
Output -((2 x^2 Cos[x] y)/(x^2 + y^2)^2) - (2 y^2 Cos[x] y)/(x^2 + y^2)^2 + Cos[x] y/(x^2 + y^2) + (8 x y Sin[x] y)/(x^2 + y^2)^3 - (x y Sin[x] y)/(x^2 + y^2)

Input (* Derivative involving a symbolic function f: *)
D[x f[x] f'[x],x]
Output f[x] f'[x]+x f'[x]^2+x f[x] f''[x]

Input D[Sin[x] Cos[x+y],x,y]
Output -Cos[x+y] Sin[x]-Cos[x] Sin[x+y]

Input D[ArcCoth[x],{x,2}]
Output (2 x)/(1 - x^2)^2

```

Integration Functions

$\text{Integrate}[f, x]$	gives the indefinite integral $\int f dx$.
$\text{Integrate}[f, \{x, xmin, xmax\}]$	gives the definite integral $\int_{xmin}^{xmax} f dx$.
$\text{Integrate}[f, \{x, xmin, xmax\}, \{y, ymin, ymax\}, \dots]$	gives the multiple integral $\int_{xmin}^{xmax} dx \int_{ymin}^{ymax} dy \dots f$.

Mathematica Examples 1.8

```

Input (* Compute an indefinite integral: *)
Integrate[1/((x^3+1)),x]
Output ArcTan[(-1 + 2 x)/Sqrt[3]]/Sqrt[3] + 1/3 Log[1 + x] - 1/6 Log[1 - x + x^2]

Input \[Integral]Sqrt[x+Sqrt[x]]\[DifferentialD]x
Output 1/12 Sqrt[Sqrt[x] + x] (-3 + 2 Sqrt[x] + 8 x) + 1/4 ArcTanh[Sqrt[Sqrt[x] + x]/Sqrt[x]]

Input Integrate[1/((x^4+x^2+1)),{x,0,Infinity}]
Output \[Pi]/(2 Sqrt[3])

Input (* Compute an definite integral: *)
Integrate[x/(Sqrt[1-x]),{x,0,1}]
Output 4/3

Input Integrate[1/(((2+x^2) Sqrt[4+3 x^2])),{x,-Infinity,Infinity}]
Output ArcCosh[Sqrt[3/2]]

Input Integrate[x^2+y^2,{x,0,1},{y,0,x}]
Output 1/3

```

Algebraic Operations

Mathematica has many functions for transforming algebraic expressions. The following list summarizes them.

<code>Simplify[expr]</code>	performs a sequence of algebraic and other transformations on expr and returns the simplest form it finds.
<code>Expand[expr]</code>	expands out products and positive integer powers in expr.
<code>Factor[expr]</code>	factors a polynomial over the integers.
<code>Together[expr]</code>	puts terms in a sum over a common denominator, and cancels factors in the result.
<code>ExpandAll[expr]</code>	expands out all products and integer powers in any part of expr.
<code>FunctionExpand[expr]</code>	tries to expand out special and certain other functions in expr when possible reducing compound arguments to simpler ones.
<code>Reduce[expr, vars]</code>	reduces the statement expr by solving equations or inequalities for vars and eliminating quantifiers.

Mathematica Examples 1.9

<code>Input</code>	<code>Simplify[Sin[x]^2+Cos[x]^2]</code>
<code>Output</code>	<code>1</code>
<code>Input</code>	<code>Expand[(1+x)^10]</code>
<code>Output</code>	<code>1 + 10 x + 45 x^2 + 120 x^3 + 210 x^4 + 252 x^5 + 210 x^6 + 120 x^7 + 45 x^8 + 10 x^9 + x^10</code>
<code>Input</code>	<code>Factor[x^10-1]</code>
<code>Output</code>	<code>(-1 + x) (1 + x) (1 - x + x^2 - x^3 + x^4) (1 + x + x^2 + x^3 + x^4)</code>
<code>Input</code>	<code>Together[x^2/(x^2-1)+x/(x^2-1)]</code>
<code>Output</code>	<code>x/(-1+x)</code>
<code>Input</code>	(*Expand polynomials anywhere inside an expression:*)
<code>Output</code>	<code>ExpandAll[1/(1+x)^3+Sin[(1+x)^3]]</code>
<code>Output</code>	<code>1/(1 + 3 x + 3 x^2 + x^3) + Sin[1 + 3 x + 3 x^2 + x^3]</code>
<code>Input</code>	<code>FunctionExpand[Sin[24 Degree]]</code>
<code>Output</code>	<code>-(1/8) Sqrt[3] (-1 - Sqrt[5]) - 1/4 Sqrt[1/2 (5 - Sqrt[5])]</code>

Solving Equations

Solutions of general algebraic equations may be found using the `Solve` command. `Solve` always tries to give you explicit formulas for the solutions to equations. However, it is a basic mathematical result that, for sufficiently complicated equations, explicit algebraic formulas in terms of radicals cannot be given. If you have an algebraic equation in one variable, and the highest power of the variable is at most four, then the Wolfram Language can always give you formulas for the solutions. However, if the highest power is five or more, it may be mathematically impossible to give explicit algebraic formulas for all the solutions.

You can also use the Wolfram Language to solve sets of simultaneous equations. You simply give the list of equations and specify the list of variables to solve for. Not all algebraic equations are solvable by Mathematica, even if theoretical solutions exist. If Mathematica is unable to solve an equation, it will represent the solution in a symbolic form. For the most part, such solutions are useless, and a numerical approximation is more appropriate. Numerical approximations are obtained with the command `NSolve`.

<code>Solve[lhs==rhs,x]</code>	solve an equation for x.
<code>Solve[{lhs1==rhs1,lhs2==rhs2,...},{x,y,...}]</code>	solve a set of simultaneous equations for x,y,....
<code>Eliminate[{lhs1==rhs1,lhs2==rhs2,...},{x,...}]</code>	eliminate x, ... in a set of simultaneous equations.
<code>Reduce[{lhs1==rhs1,lhs2==rhs2,...},{x,y,...}]</code>	give a set of simplified equations, including all possible solutions.

<code>NSolve[expr, vars]</code>	attempts to find numerical approximations to the solutions of the system expr of equations or inequalities for the variables vars.
<code>FindRoot[f, {x, x₀}]</code>	searches for a numerical root of f, starting from the point x = x ₀ .

Mathematica Examples 1.10

```

Input  Solve[x^2+a x+1==0,x]
Output {{x -> 1/2 (-a - Sqrt[-4 + a^2])}, {x -> 1/2 (-a + Sqrt[-4 + a^2])} }

Input  Solve[a x+y==7&&b x-y==1,{x,y}]
Output {{x->8/(a+b),y->-(a-7 b)/(a+b))} }

Input  (* Eliminate the variable y between two equations: *)
      Eliminate[{x==2+y,y==z},y]
Output 2+z==x

Input  NSolve[x^5-2 x+3==0,x,Reals]
Output {{x->-1.42361} }

Input  Reduce[x^2-y^3==1,{x,y}]
Output y == (-1 + x^2)^(1/3) || y == -( -1)^(1/3) (-1 + x^2)^(1/3) || y == (-1)^(2/3)
(-1 + x^2)^(1/3)

Input  FindRoot[Sin[x]+Exp[x],{x,0}]
Output {x->-0.588533}

```

Some Notes

1- In doing calculations, you will often need to use previous results that you have got. In the Wolfram Language, % always stands for your last result.

<code>%</code>	the last result generated.
<code>%%</code>	the next-to-last result.
<code>% n</code>	the result on output line Out[n].
<code>Out[n]</code>	is a global object that is assigned to be the value produced on the n th output line.

Mathematica Examples 1.11

```

Input  77^2
Output 5929

Input  %+1
Output 5930

Input  3 %+%^2+%%
Output 35188619

Input  % 2+% 3
Output 175943095

```

2- Although Mathematica is a powerful calculating tool, it has its limits. Sometimes it will happen that the calculations you tell Mathematica to do are too complicated or may be the output produced is too long. In these cases, Mathematica could be calculating for too long to get an output so you might want to stop these calculations. To abort a calculation: go to "Kernel" and select "Abort evaluation". It can take long to abort a calculation. If the computer does not respond an alternative is to close down the Kernel. By doing this you do not lose the data displayed in your notebooks, but you do lose all the results obtained so far from the Kernel, so in case you are running a series of

calculations, you would have to start again. To close down the Kernel: go to "Kernel" and select "Quit Kernel" and then "Local". Closing down the Kernel is not a practice that is done only when you want to stop a calculation. Sometimes, when you have been using Mathematica for a long time you forget about the definitions and calculations that you have done before (you might have defined values for variables or functions, for example). Those definitions can clash with the calculations you are doing, so you might want to close down the Kernel and start your new calculations from scratch. In general, it is a good idea to close down the Kernel after you have finished with a series of calculations so that when you move to a different problem your new calculations do not interact with the previous ones.

Unit 1.2

Variables and Functions

When you perform long calculations, it is often convenient to give names to your intermediate results. Just as in standard mathematics, or other computer languages, you can do this by introducing named variables. It is very important to realize that the values you assign to variables are permanent. Once you have assigned a value to a particular variable, the value will be kept until you explicitly remove it. The value will, of course, disappear if you start a whole new Wolfram Language session.

<code>x=value</code>	assign a value to the variable x.
<code>x=y=value</code>	assign a value to both x and y.
<code>x=. or Clear[x]</code>	remove any value assigned to x.
<code>{x,y}={value₁,value₂}</code>	assign different values to x and y.
<code>{x,y}={y,x}</code>	interchange the values of x and y.

Mathematica Examples 1.12

<code>Input</code>	<code>x=5</code>
<code>Output</code>	5
<code>Input</code>	<code>x^2</code>
<code>Output</code>	25
<code>Input</code>	<code>x=7+4</code>
<code>Output</code>	11

In Mathematica, one can substitute an expression with another using rules. In particular one can substitute a variable with a value without assigning the value to the variable.

<code>lhs:=rhs</code>	assigns rhs to be the delayed value of lhs. rhs is maintained in an unevaluated form. When lhs appears, it is replaced by rhs, evaluated afresh each time.
<code>expr/.rules</code>	applies a rule or list of rules in an attempt to transform each subpart of an expression expr.
<code>lhs->rhs or lhs->rhs</code>	represents a rule that transforms lhs to rhs.

Mathematica Examples 1.13

<code>Input</code>	<code>x=5</code>
<code>Input</code>	<code>y:=x+2</code>
<code>Input</code>	<code>y</code>
<code>Output</code>	5
<code>Output</code>	7
<code>Input</code>	<code>x=10</code>
<code>Output</code>	10
<code>Input</code>	<code>y</code>
<code>Output</code>	12

Mathematica Examples 1.14

<code>Input</code>	<code>x+y/. x->2</code>
<code>Output</code>	2+y

```

Input  x+y/. {x->a,y->b}
Output a+b

Input  x^2+y/.x->y/.y->x
Output x + x^2

Input  x+2 y/.{x->y,y->a}
Output 2 a+y

```

The last example reveals that Mathematica goes through the expression only once and replaces the rules. If we need Mathematica to go through the expression again and replace any expression which is possible until no substitution is possible, one uses `//.`. In fact `/.` and `//.` are shorthand for `Replace` and `ReplaceRepeated`, respectively.

Mathematica Examples 1.15

```

Input  {x,x^2,a,b}/. x->3
Output {3,9,a,b}

Input  x+2 y//.{x->y,y->a}
Output 3 a

Input  x+2 y//.{x->b,y->a,b->c}
Output 2 a+c

Input  x+2 y/. {x->b,y->a,b->c}
Output 2 a+b

Input  Sin[x]/. Sin->Cos
Output Cos[x]

```

There are many functions that are built into the Wolfram Language. Here we discuss how you can add your own simple functions to the Wolfram Language. As a first example, consider adding a function called `f` which squares its argument. The Wolfram Language command to define this function is `f[x]:=x^2`. The names like `f` that you use for functions in the Wolfram Language are just symbols. Because of this, you should make sure to avoid using names that begin with capital letters, to prevent confusion with built-in Wolfram Language functions. You should also make sure that you have not used the names for anything else earlier in your session.

<code>f[x]=value</code>	definition for a specific expression <code>x</code> .
<code>f[x_]=value</code>	definition for any expression, referred to as <code>x</code> .
<code>Clear[f]</code>	clear all definitions for <code>f</code> .
<code>Function[x, body]</code>	is a pure function with a single formal parameter <code>x</code> .
<code>Function[{x_1, x_2, ...}, body]</code>	is a pure function with a list of formal parameters.
<code>Map[f, expr] or f/@expr</code>	applies <code>f</code> to each element on the first level in <code>expr</code> .
<code>Map[f, expr, levelspec]</code>	applies <code>f</code> to parts of <code>expr</code> specified by <code>levelspec</code> .

The character `_` (referred to as "blank") on the left-hand side is very important.

Mathematica Examples 1.16

```

Input  f[x_]:=x^2
Input  f[a+1]
Output (1+a)^2

Input  f[4]
Output 16

Input  f[3 x+x^2]
Output (3 x+x^2)^2

```

```

Input   Expand[f[(x+1+y)]]
Output  1+2 x+x^2+2 y+2 x y+y^2

Input   Function[u,3+u][x]
Output  3+x

Input   Function[{u,v},u^2+v^4][x,y]
Output  x^2+y^4

Input   (* Evaluate f on each element of a list: *)
        Map[f,{a,b,c,d,e}]
Output {f[a],f[b],f[c],f[d],f[e]}

Input   f/@{a,b,c,d,e}
Output {f[a],f[b],f[c],f[d],f[e]}

Input   (* Map at top level: *)
        Map[f,{{a,b},{c,d,e}}]
Output {f[{a,b}],f[{c,d,e}]}

Input   (* Map at level 2: *)
        Map[f,{{a,b},{c,d,e}},2]
Output {{f[a],f[b]},{f[c],f[d],f[e]}}
```

Input (*Map at levels 1 and 2:*)
 Map[f,{{a,b},{c,d,e}},2]
Output {f[{f[a],f[b]}],f[{f[c],f[d],f[e]}]}

One can define functions of several variables. Here is a simple example defining $f(x, y) = \sqrt{x^2 + y^2}$.

Mathematica Examples 1.17

```

Input   f[x_,y_]:=Sqrt[x^2+y^2]
Input   f[3,4]
Output  5
```

Some Notes

- There are four kinds of bracketing used in the Wolfram Language. Each kind of bracketing has a very different meaning.

(term)	parentheses for grouping.
f[x]	square brackets for functions.
{a,b,c}	curly braces for lists.
v[[i]]	double brackets for indexing (Part[v,i]).

- Compound expression

expr ₁ ;expr ₂ ;expr ₃	do several operations and give the result of the last one.
expr ₁ ;expr ₂ ;	do the operations but print no output.
expr;	do an operation but display no output.

Mathematica Examples 1.18

```

Input   x=4;y=6;z=y+6
Output  12

Input   a=2;b=3;a+b
Output  5
```

- 3- Particularly when you write procedural programs in the Wolfram Language, you will often need to modify the value of a particular variable repeatedly. You can always do this by constructing the new value and explicitly performing an assignment such as `x=value`. The Wolfram Language, however, provides special notations for incrementing the values of variables, and for some other common cases.

<code>i++</code>	increment the value of i, by 1 returning the old value of i.
<code>i--</code>	decrement the value of i, by 1 returning the old value of i.
<code>++i</code>	pre-increment i, returning the new value of i.
<code>--i</code>	pre-decrement i, returning the new value of i.
<code>i+=di</code>	add di to the value of i and returns the new value of i.
<code>i-=di</code>	subtract di from i and returns the new value of i.
<code>x*=c</code>	multiply x by c.
<code>x/=c</code>	divide x by c.

Mathematica Examples 1.19

```
Input  k=1;k++
Output 1
```

```
Input  k
Output 2
```

```
Input  k=x
Output x
```

```
Input  k++
Output x
```

```
Input  k
Output 1+x
```

```
Input  k=1;++;k
Output 2
```

```
Input  k
Output 2
```

```
Input  k=1;k--
Output 1
```

```
Input  k
Output 0
```

```
Input  k=1;k-=5
Output -4
Input  k
Output -4
```

- 4- Primarily there are three equalities in Mathematica, `=`, `:=`, `==`. There is a fundamental differences between `=` and `:=` explained in the following examples:

Mathematica Examples 1.20

Input	<code>x=5;y=x+2;</code>
Input	<code>y</code>
Output	7

Input	<code>x=10</code>
Output	10

Input	<code>x=5;y:=x+2;</code>
Input	<code>y</code>
Output	7

Input	<code>x=10</code>
Output	10

Input	y	Input	y
Output	7	Output	12
Input	x=15	Input	x=15
Output	15	Output	15
Input	y	Input	y
Output	7	Output	17

It is clear that when we defined `y=x+2` then y takes the value of $x+2$ and this will be assigned to y. No matter if x changes its value, the value of y remains the same. In other words, y is independent of x. But in `y:=x+2`, y is dependent on x, and when x changes, the value of y changes too. Namely using `:=` then y is a function with variable x. Finally, the equality `==` is used to compare:

Mathematica Examples 1.21

Input	5==5
Output	True
Input	3==5
Output	False

Unit 1.3

Lists

Lists are extremely important objects. In doing calculations, it is often convenient to collect together several objects and treat them as a single entity. Lists give you a way to make collections of objects. Lists are sequences of Mathematica objects separated by commas and enclosed by curly brackets. A list such as `{3,5,1}` is a collection of three objects. But in many ways, you can treat the whole list as a single object. You can, for example, do arithmetic on the whole list at once, or assign the whole list to be the value of a variable.

Defining your own lists is easy. You can, for example, type them in full, like this:

```
Input      oddList = {81, 3, 5, 7, 9, 11, 13, 15, 17}
Output     {81, 3, 5, 7, 9, 11, 13, 15, 17}
```

Alternatively, if (as here) the list elements correspond to a rule of some kind, the command `Table` can be used, like this:

```
Input      oddList = Table[2 n + 1, {n, 0, 8}]
Output     {1, 3, 5, 7, 9, 11, 13, 15, 17}
```

The functions for obtaining elements of lists are

<code>First[list]</code>	the first element.
<code>Last[list]</code>	the last element.
<code>Part[list,n] or list[[n]]</code>	the nth element.
<code>Part[list,-n] or list[[-n]]</code>	the nth element from the end.
<code>Part[list,{n1,n2,...}] or list[{{n1,n2,...}}]</code>	the list of the n1th, n2th, ... elements.
<code>Take[list,n]</code>	the list of the first n elements.
<code>Take[list,-n]</code>	the list of the last n elements.
<code>Take[list,{m,n}]</code>	the list of the mth through nth elements.
<code>Rest[list]</code>	list without the first element.
<code>Most[list]</code>	list without the last element.
<code>Drop[list,n]</code>	list without the first n elements.
<code>Drop[list,-n]</code>	list without the last n elements.
<code>Drop[list,{m,n}]</code>	list without the mth through nth elements.

Mathematica Examples 1.22

```
Input    First[{a,b,c}]
Output   a

Input    First[{{a,b},{c,d}}]
Output   {a,b}

Input    Last[{a,b,c}]
Output   c

Input    {a,b,c,d,e,f}[[3]]
Output   c

Input    {{a,b,c},{d,e,f},{g,h,i}}[[2,3]]
Output   f

Input    Take[{a,b,c,d,e,f},4]
Output   {a,b,c,d}
```

```

Input  Rest[{a,b,c,d}]
Output {b,c,d}

Input  Most[{a,b,c,d}]
Output {a,b,c}

Input  Drop[{a,b,c,d,e,f},2]
Output {c,d,e,f}

```

Some functions for inserting, deleting, and replacing list and sublist elements are

Prepend[list, elem]	insert elem at the beginning of list.
Append[list, elem]	insert elem at the end of list.
Insert[list, elem, i]	insert elem at position i in list.
Insert[list, elem, {i, j, ...}]	insert elem at position {i, j, ...} in list.
Insert[list, elem, {{i1, j1, ...}, {i2, ...}, ...}]	insert elem at positions {i1, j1, ...}, {i2, ...}, ... in list.
Delete[list, i]	delete the element at position i in list.
Delete[list, {i, j, ...}]	delete the element at position {i, j, ...} in list.
Delete[list, {{i1, j1, ...}, {i2, ...}, ...}]	delete elements at positions {i1, j1, ...}, {i2, ...}, ... in list.
ReplacePart[list, elem, i]	replace the element at position i in list with elem.
ReplacePart[list, elem, {i, j, ...}]	replace the element at position {i, j, ...} with elem.
ReplacePart[list, elem, {{i1, j1, ...}, {i2, ...}, ...}]	replace elements at positions {i1, j1, ...}, {i2, ...}, ... with elem.

Mathematica Examples 1.23

```

Input  Prepend[{a,b,c,d},x]
Output {x,a,b,c,d}

Input  Append[{a,b,c,d},x]
Output {a,b,c,d,x}

Input  Insert[{a,b,c,d,e},x,3]
Output {a,b,x,c,d,e}

Input  Insert[{a,b,c,d,e},x,-2]
Output {a,b,c,d,x,e}

Input  Delete[{a,b,c,d},3]
Output {a,b,d}

Input  Delete[{a,b,c,d},{{1},{3}}]
Output {b,d}

Input  ReplacePart[{a,b,c,d,e},3->xxx]
Output {a,b,xxx,d,e}

Input  ReplacePart[{a,b,c,d,e},{2->xx,5->yy}]
Output {a,xx,c,d,yy}

```

Some functions for rearranging lists are

Sort[list]	sort the elements of list into canonical order.
Union[list]	give a sorted version of list, in which all duplicated elements have been dropped.
Reverse[list]	reverse the order of the elements in list.
RotateLeft[list]	cycle the elements in list one position to the left.

<code>RotateLeft[list,n]</code>	cycle the elements in list n positions to the left.
<code>RotateRight[list]</code>	cycle the elements in list one position to the right.
<code>RotateRight[list,n]</code>	cycle the elements in list n positions to the right.
<code>Permutations[list]</code>	generate a list of all possible permutations of the elements in list.
<code>Partition[list,n]</code>	partition list into nonoverlapping sublists of length n.
<code>Partition[list,n,d]</code>	generate sublists with offset d.
<code>Split[list]</code>	split list into sublists consisting of runs of identical elements.
<code>Transpose[list]</code>	transpose the first two levels in list.
<code>Flatten[list]</code>	flatten out nested lists.
<code>Flatten[list,n]</code>	flatten out the top n levels.
<code>FlattenAt[list,i]</code>	flatten out a sublist that appears as the ith element of list.
<code>FlattenAt[list,{i,j,...}]</code>	flatten out the element of list at position {i,j, ...}.
<code>FlattenAt[list,{{i1,j1,...},{i2,j2,...}}]</code>	flatten out elements of list at several positions.
<code>Join[list1,list2,...]</code>	concatenate lists together.
<code>Union[list1,list2,...]</code>	give a sorted list of all the distinct elements that appear in any of the lists.

Mathematica Examples 1.24

Input	<code>Sort[{d,b,c,a}]</code>
Output	{a,b,c,d}
Input	<code>Sort[{4,1,3,2,2},Greater]</code>
Output	{4,3,2,2,1}
Input	<code>Union[{1,2,1,3,6,2,2}]</code>
Output	{1,2,3,6}
Input	<code>Union[{a,b,a,c},{d,a,e,b},{c,a}]</code>
Output	{a,b,c,d,e}
Input	<code>Reverse[{a,b,c,d}]</code>
Output	{d,c,b,a}
Input	<code>RotateLeft[{a,b,c,d,e},2]</code>
Output	{c,d,e,a,b}
Input	<code>RotateRight[{a,b,c,d,e},2]</code>
Output	{d,e,a,b,c}
Input	<code>Permutations[{a,b,c}]</code>
Output	{ {a,b,c}, {a,c,b}, {b,a,c}, {b,c,a}, {c,a,b}, {c,b,a} }
Input	<code>Partition[{a,b,c,d,e,f},2]</code>
Output	{ {a,b}, {c,d}, {e,f} }
Input	<code>Flatten[{{a,b},{c,{d},e},{f,{g,h}}}]</code>
Output	{a,b,c,d,e,f,g,h}
Input	<code>Transpose[{{a,b,c},{x,y,z}}]</code>
Output	{ {a,x}, {b,y}, {c,z} }
Input	<code>Join[{a,b,c},{x,y},{u,v,w}]</code>
Output	{a,b,c,x,y,u,v,w}

Vectors and matrices in the Wolfram Language are simply represented by lists and by lists of lists, respectively. Functions for generating lists are `Range[]`, `Table[]`, and `Array[]`.

Vectors

Mathematica has many functions for generating vectors. The following list summarizes them.

<code>{e₁, e₂, ...}</code>	is a list of elements.
<code>Range[n]</code>	create the list {1,2,3, ..., n}.
<code>Range[n₁,n₂]</code>	create the list {n ₁ , n ₁ + 1, ..., n ₂ }.
<code>Range[n₁,n₂,dn]</code>	create the list {n ₁ , n ₁ + dn, ..., n ₂ }.
<code>Table[f,{i,n}]</code>	build a length-n vector by evaluating f with i = 1,2, ..., n.
<code>Length[list]</code>	give the number of elements in list.
<code>List[[i]] or Part[list,i]</code>	give the i th element in the vector list.

Mathematica Examples 1.25

<code>Input</code>	<code>List[a,b,c,d]</code>
<code>Output</code>	<code>{a,b,c,d}</code>
<code>Input</code>	<code>v={x,y}</code>
<code>Output</code>	<code>{x,y}</code>
<code>Input</code>	<code>Range[4]</code>
<code>Output</code>	<code>{1,2,3,4}</code>
<code>Input</code>	<code>Range[x,x+4]</code>
<code>Output</code>	<code>{x,1+x,2+x,3+x,4+x}</code>
<code>Input</code>	<code>Table[i^2,{i,10}]</code>
<code>Output</code>	<code>{1,4,9,16,25,36,49,64,81,100}</code>
<code>Input</code>	<code>Length[{a,b,c,d}]</code>
<code>Output</code>	<code>4</code>
<code>Input</code>	<code>{5,8,6,9}[[2]]</code>
<code>Output</code>	<code>8</code>

<code>c v</code>	multiply a vector v by a scalar.
<code>a.b</code>	dot product of two vectors a.b.
<code>Cross[a,b]</code>	cross product of two vectors (also input as a × b).
<code>Norm[v]</code>	Euclidean norm of a vector v.
<code>Normalize[v]</code>	gives the normalized form of a vector v.
<code>Orthogonalize [{v₁,v₂,...}]</code>	gives an orthonormal basis found by orthogonalizing the vectors v _i .

Mathematica Examples 1.26

<code>Input</code>	<code>{a,b,c}.{x,y,z}</code>
<code>Output</code>	<code>a x+b y+c z</code>
<code>Input</code>	<code>Cross[{a,b,c},{x,y,z}]</code>
<code>Output</code>	<code>{-c y+b z,c x-a z,-b x+a y}</code>
<code>Input</code>	<code>Norm[{x,y,z}]</code>
<code>Output</code>	<code>Sqrt[Abs[x]^2 + Abs[y]^2 + Abs[z]^2]</code>
<code>Input</code>	<code>Normalize[{1,5,1}]</code>
<code>Output</code>	<code>{1/(3 Sqrt[3]), 5/(3 Sqrt[3]), 1/(3 Sqrt[3])}</code>

```
Input Orthogonalize[{{1,0,1},{1,1,1}}]
Output {1/Sqrt[2], 0, 1/Sqrt[2]}, {0, 1, 0}
```

Matrix

Mathematica has many functions for generating matrices. The following list summarizes them.

<code>{{{a,b},{c,d}}}</code>	matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.
<code>Table[f,{i,m},{j,n}]</code>	build an $m \times n$ matrix by evaluating f with i ranging from 1 to m and j ranging.
<code>List[[i,j]]</code> or <code>Part[list,i,j]</code>	give the i,j th element in the matrix list.
<code>DiagonalMatrix[list]</code>	generate a square matrix with the elements in list on the main.
<code>Dimensions[list]</code>	give the dimensions of a matrix represented by list.
<code>Column[list]</code>	display the elements of list in a column.
<code>c m</code>	multiply a matrix m by a scalar.
<code>a.b</code>	dot product of two matrices a.b.
<code>Inverse[m]</code>	matrix inverse m.
<code>MatrixPower[m,n]</code>	gives the n^{th} power of a matrix m.
<code>Det[m]</code>	Determinant m.
<code>Tr[m]</code>	Trace m.
<code>Transpose[m]</code>	Transpose m.

Mathematica Examples 1.27

```
Input m=={{a,b},{c,d}}
Output {{a,b},{c,d}}
```



```
Input m[[1]]
Output {a,b}
```



```
Input m[[1,2]]
Output b
```



```
Input v={x,y}
Output {x,y}
```



```
Input m.v
Output {a x+b y,c x+d y}
```



```
Input m.m
Output {{a^2+b c,a b+b d},{a c+c d,b c+d^2}}
```



```
Input s=Table[i+j,{i,3},{j,3}]
Output {{2,3,4},{3,4,5},{4,5,6}}
```



```
Input DiagonalMatrix[{a,b,c}]
Output {{a,0,0},{0,b,0},{0,0,c}}
```



```
Input Det[m]
Output -b c+a d
```



```
Input Transpose[m]
Output {{a,c},{b,d}}
```



```
Input h=Table[1/(i+j-1),{i,3},{j,3}]
Output {{1,1/2,1/3},{1/2,1/3,1/4},{1/3,1/4,1/5}}
```

Input	<code>Inverse[h]</code>
Output	<code>{ {9, -36, 30}, {-36, 192, -180}, {30, -180, 180} }</code>

Array

<code>Array[f, n]</code>	generates a list of length n, with elements $f[i]$.
<code>Array[f, n, r]</code>	generates a list using the index origin r.
<code>Array[f, n, {a, b}]</code>	generates a list using n values from a to b.
<code>Array[f, {n1, n2, ...}]</code>	generates an $n_1 \times n_2 \times \dots$ array of nested lists, with elements $f[i_1, i_2, \dots]$.

Mathematica Examples 1.28

Input	<code>Array[f, 10]</code>
Output	<code>{f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10]}</code>
Input	<code>Array[f, {3, 2}]</code>
Output	<code>{ {f[1, 1], f[1, 2]}, {f[2, 1], f[2, 2]}, {f[3, 1], f[3, 2]} }</code>

Layout & Tables

<code>Print[expr]</code>	prints expr as output.
<code>MatrixForm[list]</code>	prints with the elements of list arranged in a regular array.
<code>TableForm[list]</code>	prints with the elements of list arranged in an array of rectangular cells.
<code>Grid[{{expr11, expr12, ...}, {expr21, expr22, ...}, ...}]</code>	is an object that formats with the $expr_{ij}$ arranged in a two-dimensional grid.
<code>Row[{expr1, expr2, ...}]</code>	is an object that formats with the $expr_i$ arranged in a row, potentially extending over several lines.
<code>Row[list, s]</code>	inserts s as a separator between successive elements.
<code>Column[{expr1, expr2, ...}]</code>	is an object that formats with the $expr_i$ arranged in a column, with $expr_1$ above $expr_2$, etc.
<code>Multicolumn[list, cols]</code>	is an object that formats with the elements of list arranged in a grid with the indicated number of columns.
<code>Multicolumn[list, {rows, Automatic}]</code>	formats as a grid with the indicated number of rows.

Mathematica Examples 1.29

Input	<code>MatrixForm[{{1, 2}, {3, 4}}]</code>
Output	$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
Input	<code>MatrixForm[Table[1/(i+j), {i, 4}, {j, 4}]]</code>
Output	$\begin{pmatrix} 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \\ 1/5 & 1/6 & 1/7 & 1/8 \end{pmatrix}$
Input	<code>TableForm[Table[1/(i+j), {i, 4}, {j, 4}]]</code>
Output	$\begin{array}{cccc} 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \\ 1/5 & 1/6 & 1/7 & 1/8 \end{array}$
Input	<code>Grid[{{a, b, c}, {x, y, z}}]</code>
Output	$\begin{array}{ccc} a & b & c \\ x & y & z \end{array}$

Input `Grid[{{a,b,c},{x,y^2,z^3}},Frame->All]`
 Output

a	b	c
x	y^2	z^3

Input `Row[{aaa,b,cccc}]`
 Output aaabcccc

Input `Row[{aaa,b,cccc}, "----"]`
 Output aaa----b----cccc

Input `Column[{1,12,123,1234}]`
 Output 1
12
123
1234

Input `Column[{1,22,333,4444},Frame->True]`
 Output

1
22
333
4444

Input `Multicolumn[Range[50],{6,Automatic}]`
 Output 1 7 13 19 25 31 37 43 49
2 8 14 20 26 32 38 44 50
3 9 15 21 27 33 39 45
4 10 16 22 28 34 40 46
5 11 17 23 29 35 41 47
6 12 18 24 30 36 42 48

Notes:

1- When using `MatrixForm`, it is very important to note that the `MatrixForm` is used for display purposes only. If a matrix is defined with the `MatrixForm` in it, that matrix definition cannot be used in any subsequent calculation. For example, consider the following definition of matrix `m`:

Input: `m = MatrixForm[{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}]`

Output:

```
{  
  {1, 2, 3, 4},  
  {5, 6, 7, 8},  
  {9, 10, 11, 12}  
}
```

We cannot perform any operations on matrix `m` in this form. For example, using the `Transpose` function on `m` simply returns the initial matrix `m` wrapped in `Transpose`.

Input: `Transpose[m]`

Output:

```
Transpose[  
{  
  {1, 2, 3, 4},  
  {5, 6, 7, 8},  
  {9, 10, 11, 12}  
}]
```

The solution obviously is not to use the `MatrixForm` in the definition of matrices. After the definition, we can use the `MatrixForm` to get a nice-looking display.

```
Input: m={{1,2,3,4},{5,6,7,8},{9,10,11,12}}; MatrixForm[m]
Output:
({{
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
})
Input: Transpose[m]

Output: {{1,5,9},{2,6,10},{3,7,11},{4,8,12}}
```

2- To explicitly define a 3×1 column vector is to enter it as a two-dimensional list. Here, each entry defines a row of a matrix with one column.

```
Input: a={{1},{2},{3}};MatrixForm[a]
Output:
({{
  {1},
  {2},
  {3}
})
Input: b={4,5,6}
Output: {4,5,6}
```

In this case `Transpose[a]` . `b` makes sense. However, the result is a 1×1 matrix and not a scalar.

```
Input: Transpose[a].b
Output: {32}
```

Obviously, now `a` . `b` will produce an error because the dimensions do not match.

```
Input: a.b
Output:

Dot::dotsh: Tensors {{1},{2},{3}} and {4,5,6} have incompatible shapes.

{{1},{2},{3}}.{4,5,6}
```

To get a 3×3 matrix, we need to define `b` as a two-dimensional matrix with only one row, as follows:

```
Input: b={{4,5,6}}
Output: {{4,5,6}}
```

Now $a (3 \times 1) . (1 \times 3)$ can be evaluated directly, as one would expect.

```
Input: MatrixForm[a.b]
Output:

({{
  {4, 5, 6},
  {8, 10, 12},
  {12, 15, 18}
})
```

Mathematica Examples 1.30

```

Input      u={{x},{y}};
MatrixForm[u]
MatrixForm[Transpose[u]]
Output    ({{x}, {y}})
Output    ({{x, y}})

Input      v={{x,y}};
MatrixForm[v]
MatrixForm[Transpose[v]]
Output    ({{x, y}})
Output    ({{x}, {y}})

Input      (* Scalar product of vectors in three dimensions (row product column): *)
{{a, b, c}} . {{x}, {y}, {z}}
Output    {{a x + b y + c z}>

Input      (* Scalar product of vectors in three dimensions (column product row): *)
{{x},{y},{z}}.{{a,b,c}}//MatrixForm
Output    ({{a x, b x, c x}, {a y, b y, c y}, {a z, b z, c z}})

Input      (* Scalar product of vectors in two dimensions (row product column): *)
u1 = {{1, 1}};
v1 = {{-1}, {1}};
u1 . v1
Output    {{0}>

Input      (* The product of a matrix and a vector: *)
{{a, b}, {c, d}} . {{x}, {y}}
Output    {{a x + b y}, {c x + d y}>

Input      (* The product v^T M of a vector and a matrix: *)
{{x, y}} . {{a, b}, {c, d}}
Output    {{a x + c y, b x + d y}>

Input      (* The product v^T Mw of a matrix and two vectors: *)
{{x, y}} . {{a, b}, {c, d}} . {{r}, {s}}
Output    {{r (a x + c y) + s (b x + d y)}}

Input      (* Product of exact matrices: *)
m = {{1, 2}, {3, 4}, {5, 6}};
n = {{6, 5, 4}, {3, 2, 1}};
m . n // MatrixForm
n . m // MatrixForm
Output    ({{12, 9, 6}, {30, 23, 16}, {50, 37, 24}})

```

```

Output      {48, 37, 26}
           })
({{
   {41, 56},
   {14, 20}
  })

Input      (* Visualize the input and output matrices: *)
{ MatrixPlot[n], MatrixPlot[m . n] }

Output

,
```

3- Since Mathematica treats matrices as essentially lists, it does not distinguish between a column or a row vector. The actual form is determined from syntax in which it is used. For example, define two vectors **a** and **b** as follows:

```
Input : a={1, 2, 3}; b={4, 5, 6};
```

The inner product is evaluated simply as **a . b**, resulting in a scalar. Explicitly evaluating **Transpose[a]. b** will produce the same result.

```

Input : a . b
Output: 32
Input : Transpose[a] . b
Output: 32

```

If we want to treat **a** as a column vector (3×1) and **b** as a row vector (1×3) to get a 3×3 matrix from the product, we need to use the **Outer** function of Mathematica, as follows:

```

Input : ab = Outer[Times, a, b]; MatrixForm[ab]
Output :
({{
   {4, 5, 6},
   {8, 10, 12},
   {12, 15, 18}
  })

```

Mathematica Examples 1.31

```

Input      (* Dot effectively treats vectors multiplied from the right as column vectors:
           *)
          a = {{1, 2}, {3, 4}, {5, 6}};
          a . {1, 1}

Output     {3, 7, 11}

```

```

Input      a . {{1}, {1}}
Output    {{3}, {7}, {11} }

Input      (* Dot effectively treats vectors multiplied from the left as row vectors: *)
           {1, 1, 1} . a
Output    {9, 12}

Input      {{1, 1, 1}} . a
Output    {{9, 12}}

```

Gradient function

`Grad[f,{x1,...,xn}]` gives the gradient $(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$.

Mathematica Examples 1.32

```

Input      (* The gradient in three-dimensional Cartesian coordinates: *)
           Grad[f[x, y, z], {x, y, z}]
Output    {f^(1,0,0)[x,y,z], f^(0,1,0)[x,y,z], f^(0,0,1)[x,y,z]}

Input      (* The gradient in two dimensions: *)
           Grad[Sin[x^2 + y^2], {x, y}]
Output    {2 x Cos[x^2 + y^2], 2 y Cos[x^2 + y^2]}

Input      (* The gradient of a vector field in Cartesian coordinates, the Jacobian matrix: *)
           Grad[{f[x, y, z], g[x, y, z], h[x, y, z]}, {x, y, z}] // MatrixForm
Output    ({
           {f^(1,0,0)[x,y,z], f^(0,1,0)[x,y,z], f^(0,0,1)[x,y,z]},
           {g^(1,0,0)[x,y,z], g^(0,1,0)[x,y,z], g^(0,0,1)[x,y,z]},
           {h^(1,0,0)[x,y,z], h^(0,1,0)[x,y,z], h^(0,0,1)[x,y,z]}
         })

Input      (* Compute the Hessian of a scalar function: *)
           grad1 = Grad[x*y*z, {x, y, z}]
Output    {y z, x z, x y}

Input      Grad[grad1, {x, y, z}] // MatrixForm
Output    ({
           {0, z, y},
           {z, 0, x},
           {y, x, 0}
         })

```

Unit 1.4

2D and 3D Graphing

The graph of a function offers tremendous insight into the behavior of the function and can be of great value in the solution of problems in mathematics. One of the outstanding features of Mathematica is its graphing capabilities. Mathematica contains functions for 2D and 3D graphing of functions, lists, and arrays of data.

Basic Plotting

<code>Plot[f,{x,xmin,xmax}]</code>	plot f as a function of x from x_{\min} to x_{\max} .
<code>Plot[{f1,f2,...},{x,xmin,xmax}]</code>	plot several functions together.

When the Wolfram Language plots a graph for you, it has to make many choices. It has to work out what the scales should be, where the function should be sampled, how the axes should be drawn, and so on. Most of the time, the Wolfram Language will probably make pretty good choices. However, if you want to get the very best possible pictures for your particular purposes, you may have to help the Wolfram Language in making some of its choices.

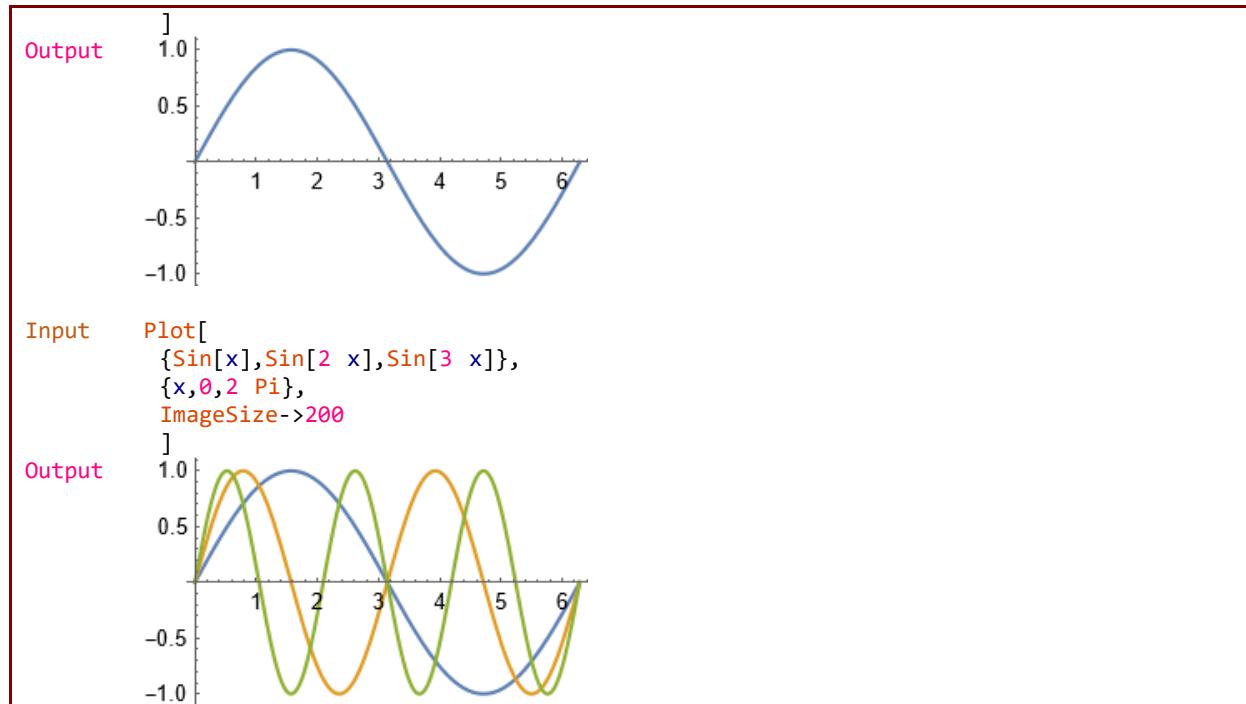
There is a general mechanism for specifying "options" in Wolfram Language functions. Each option has a definite name. As the last argument to a function like `Plot`, you can include a sequence of rules of the form `name->value`, to specify the values for various options. Any option for which you do not give an explicit rule is taken to have its "default" value.

Some options for `Plot` function are

<code>AspectRatio</code>	the height-to-width ratio for the plot; Automatic sets it from the absolute x and y coordinates
<code>Axes</code>	whether to include axes
<code>AxesLabel</code>	labels to be put on the axes; ylabel specifies a label for the y axis, { xlabel,ylabel } for both axes
<code>AxesOrigin</code>	the point at which axes cross
<code>BaseStyle</code>	the default style to use for the plot
<code>FormatType</code>	the default format type to use for text in the plot
<code>Frame</code>	whether to draw a frame around the plot
<code>FrameLabel</code>	labels to be put around the frame; give a list in clockwise order starting with the lower x axis
<code>FrameTicks</code>	what tick marks to draw if there is a frame; None gives no tick marks
<code>GridLines</code>	what grid lines to include; Automatic includes a grid line for every major tick mark
<code>PlotLabel</code>	an expression to be printed as a label for the plot
<code>PlotRange</code>	the range of coordinates to include in the plot; All includes all points
<code>Ticks</code>	what tick marks to draw if there are axes; None gives no tick marks
<code>PlotStyle</code>	a list of lists of graphics primitives to use for each curve (see "Graphics Directives and Options")
<code>ClippingStyle</code>	what to draw when curves are clipped
<code>Filling</code>	filling to insert under each curve
<code>FillingStyle</code>	style to use for filling
<code>PlotPoints</code>	the initial number of points at which to sample the function
<code>MaxRecursion</code>	the maximum number of recursive subdivisions allowed

Mathematica Examples 1.33

```
Input   Plot[
          Sin[x],
          {x,0,2 Pi},
          ImageSize->200]
```



3D Plot

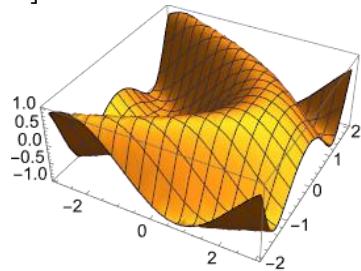
`Plot3D[f,{x,xmin,xmax},{y,ymin,ymax}]` make a three-dimensional plot of f as a function of the variables x and y .

Some options for `Plot3D` function are

<code>Axes</code>	whether to include axes
<code>AxesLabel</code>	labels to be put on the axes: <code>zlabel</code> specifies a label for the z axis, <code>{xlabel,label,zlabel}</code> for all axes
<code>BaseStyle</code>	the default style to use for the plot
<code>Boxed</code>	whether to draw a three-dimensional box around the surface
<code>FaceGrids</code>	how to draw grids on faces of the bounding box; <code>All</code> draws a grid on every face
<code>LabelStyle</code>	style specification for labels
<code>Lighting</code>	simulated light sources to use
<code>Mesh</code>	whether an xy mesh should be drawn on the surface
<code>PlotRange</code>	the range of z or other values to include
<code>SphericalRegion</code>	whether to make the circumscribing sphere fit in the final display area
<code>ViewAngle</code>	angle of the field of view
<code>ViewCenter</code>	point to display at the center
<code>ViewPoint</code>	the point in space from which to look at the surface
<code>ViewVector</code>	position and direction of a simulated camera
<code>ViewVertical</code>	direction to make vertical
<code>BoundaryStyle</code>	how to draw boundary lines for surfaces
<code>ClippingStyle</code>	how to draw clipped parts of surfaces
<code>ColorFunction</code>	how to determine the color of the surfaces
<code>Filling</code>	filling under each surface
<code>FillingStyle</code>	style to use for filling
<code>PlotPoints</code>	the number of points in each direction at which to sample the function; <code>{nx,ny}</code> specifies different numbers in the x and y directions
<code>PlotStyle</code>	graphics directives for the style of each surface

Mathematica Examples 1.34

Input `Plot3D[
 Sin[x+y^2],
 {x, -3, 3},
 {y, -2, 2},
 ImageSize->200
]`

Output**Plotting Lists of Data**`ListPlot[{y1, y2, ...}]`plot y_1, y_2, \dots at x values 1, 2,`ListPlot[{{x1, y1}, {x2, y2}, ...}]`plot points $(x_1, y_1), \dots$ `ListLinePlot[list]`

join the points with lines.

`ListPlot3D[array]`

generates a three-dimensional plot of a surface representing an array of height values.

`ListPlot3D[{{x1, y1, z1}, {x2, y2, z2}, ...}]`generates a plot of the surface with heights z_i at positions x_i, y_i .`ListPlot3D[{data1, data2, ...}]`plots the surfaces corresponding to each of the data_i .`ListPointPlot3D[array]`

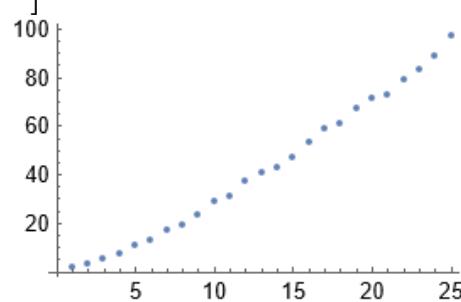
generates a 3D scatter plot of points with a 2D array of height values.

`ListPointPlot3D[{{x1, y1, z1}, {x2, y2, z2}, ...}]`generates a 3D scatter plot of points with coordinates x_i, y_i, z_i `ListPointPlot3D[{data1, data2, ...}]`

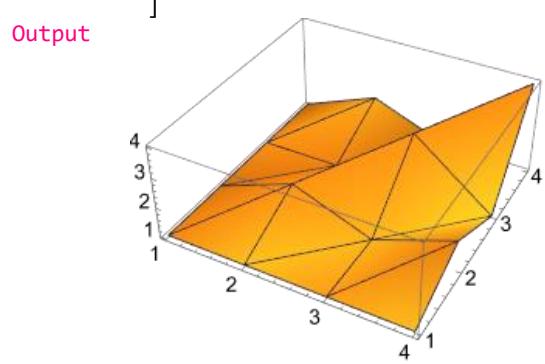
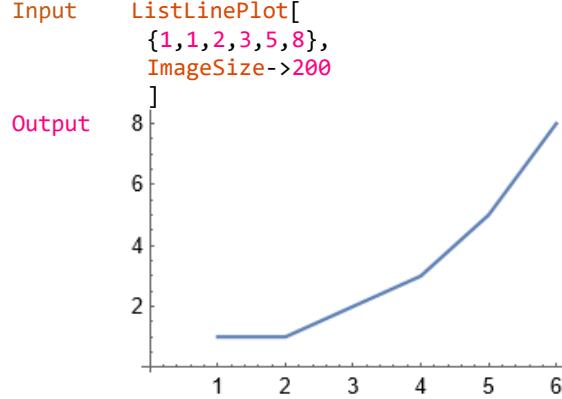
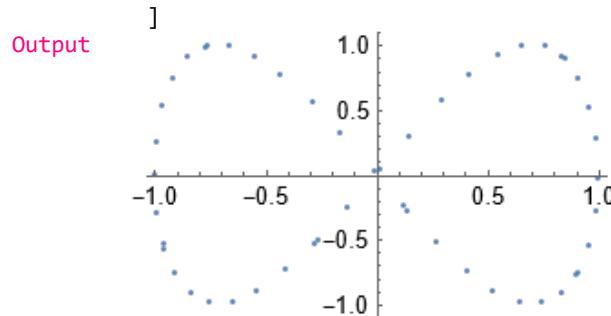
plots several collections of points, by default in different colors.

`DensityPlot[f, {x, xmin, xmax}], {y, ymin, ymax}]`makes a density plot of f as a function of x and y .`ContourPlot[f, {x, xmin, xmax}], {y, ymin, ymax}]`generates a contour plot of f as a function of x and y .**Mathematica Examples 1.35**

Input `ListPlot[
 Prime[Range[25]],
 ImageSize->200
]`

Output

Input `ListPlot[
 Table[
 {Sin[n], Sin[2 n]},
 {n, 50}
],
 ImageSize->200
]`

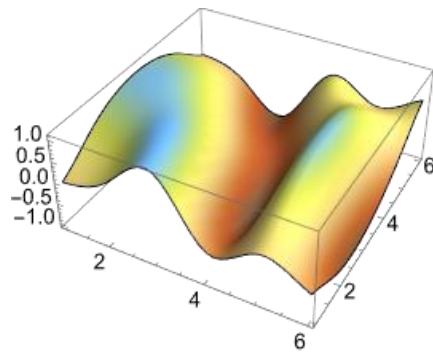


Input

```
data=Table[  
Sin[j^2+i],  
{i,0,Pi,Pi/5},  
{j,0,Pi,Pi/5}  
];
```

ListPlot3D[
data,
Mesh->None,
InterpolationOrder->3,
ColorFunction->"SouthwestColors",
ImageSize->200
]

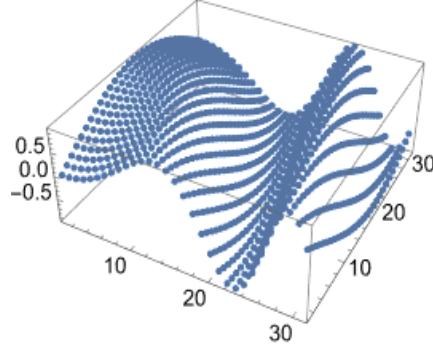
Output



Input

```
ListPointPlot3D[  
  Table[  
    Sin[j^2+i],  
    {i,0,3,0.1},  
    {j,0,3,0.1}  
  ],  
  ImageSize->200  
]
```

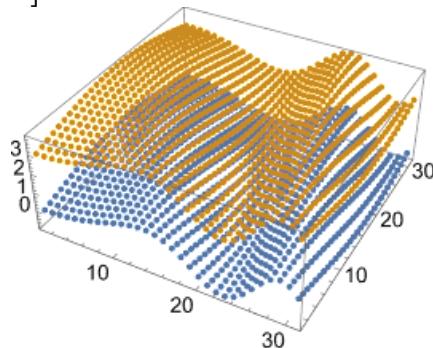
Output



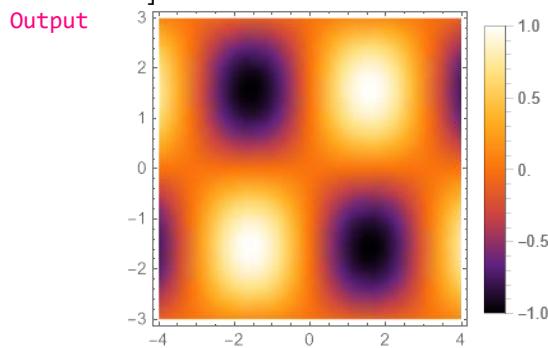
Input

```
ListPointPlot3D[  
  {  
    Table[  
      Sin[j^2+i],  
      {i,0,3,0.1},  
      {j,0,3,0.1}  
    ],  
    Table[  
      Sin[j^2+i]+3,  
      {i,0,3,0.1},  
      {j,0,3,0.1}  
    ]  
  },  
  ImageSize->200  
]
```

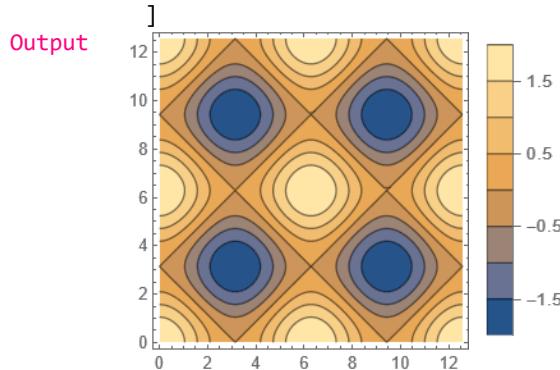
Output



Input `DensityPlot[
 Sin[x] Sin[y],
 {x, -4, 4},
 {y, -3, 3},
 ColorFunction -> "SunsetColors",
 PlotLegends -> Automatic,
 ImageSize -> 200
]`



Input `ContourPlot[
 Cos[x] + Cos[y],
 {x, 0, 4 Pi},
 {y, 0, 4 Pi},
 PlotLegends -> Automatic,
 ImageSize -> 200
]`



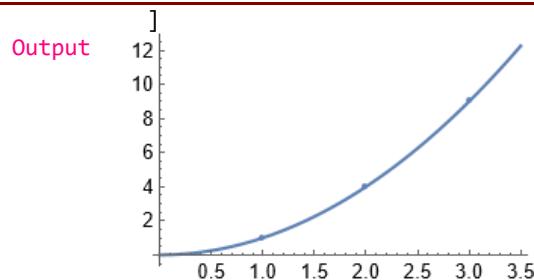
Combining Plots

`Show[plot1, plot2, ...]
GraphicsGrid[{plot1, plot2, ...}, ...]
GraphicsRow[{plot1, plot2, ...}]
GraphicsColumn[{plot1, plot2, ...}]`

combine several plots.
draw an array of plots.
draw several plots side by side.
draw a column of plots.

Mathematica Examples 1.36

Input `Show[
 Plot[
 x^2,
 {x, 0, 3.5},
 ImageSize -> 200
],
 ListPlot[
 {1, 4, 9},
 ImageSize -> 200]`



Vector Field Plots

```
VectorPlot[{vx,vy},{x,xmin,xmax},{y,ymin,ymax}]

VectorPlot3D[{vx,vy,vz},{x,xmin,xmax},
{y,ymin,ymax},{z,zmin,zmax}]

VectorDensityPlot[{{vx,vy},s},{x,xmin,xmax},
{y,ymin,ymax}]
```

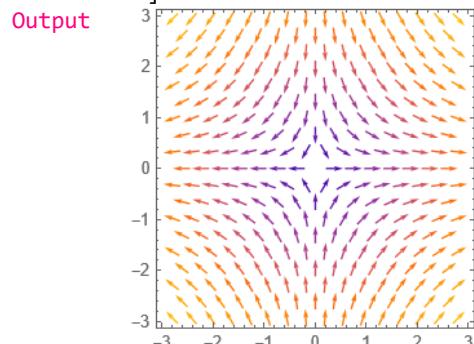
generates a vector plot of the vector field $\{v_x, v_y\}$ as a function of x and y .

generates a 3D vector plot of the vector field $\{v_x, v_y, v_z\}$ as a function of x , y , and z .

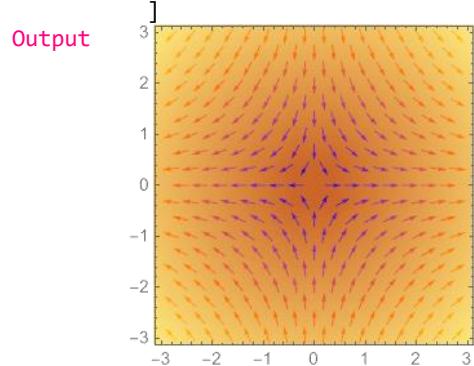
generates a vector plot of the vector field $\{v_x, v_y\}$ as a function of x and y , superimposed on a density plot of the scalar field s .

Mathematica Examples 1.37

Input `VectorPlot[
 {x,-y},
 {x,-3,3},
 {y,-3,3},
 ImageSize->200
]`

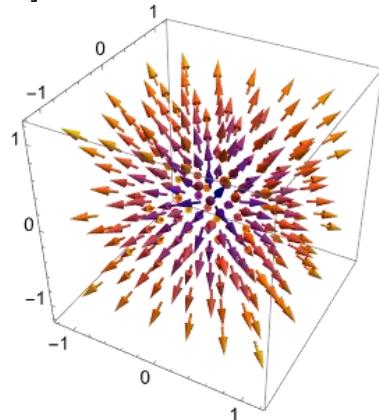


Input `VectorDensityPlot[
 {x,-y},
 {x,-3,3},
 {y,-3,3},
 ImageSize->200
]`



Input `VectorPlot3D[
 {x,y,z},
 {x,-1,1},
 {y,-1,1},
 {z,-1,1},
 ImageSize->200
]`

Output



Manipulate

The single command `Manipulate` lets you create an astonishing range of interactive applications with just a few lines of input. The output you get from evaluating a `Manipulate` command is an interactive object containing one or more controls (sliders, etc.) that you can use to vary the value of one or more parameters. The output is very much like a small applet or widget: it is not just a static result, it is a running program you can interact with.

`Manipulate[expr,{u,umin,umax}]`

generates a version of `expr` with controls added to allow interactive manipulation of the value of `u`.

`Manipulate[expr,{u,umin,umax,du}]`

allows the value of `u` to vary between `umin` and `umax` in steps `du`.

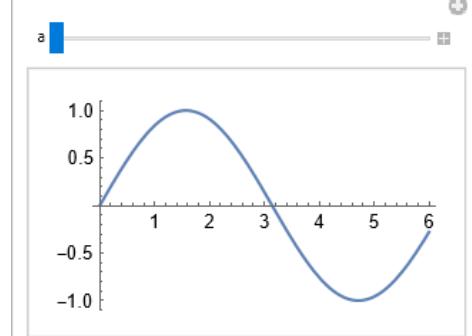
`Manipulate[expr,{{u,uinit},umin,umax,...}]`

takes the initial value of `u` to be `uinit`.

Mathematica Examples 1.38

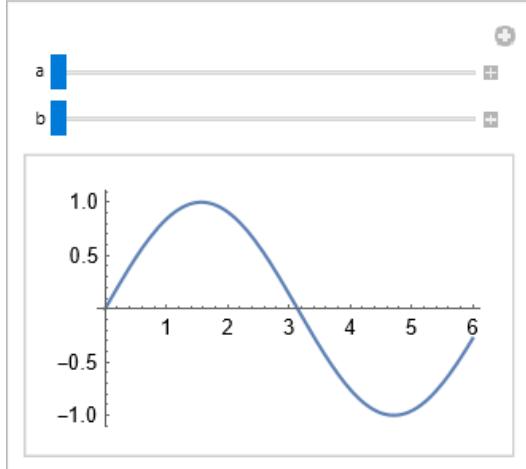
Input `Manipulate[
 Plot[
 Sin[x (1+a x)],
 {x,0,6},
 ImageSize->200
],
 {a,0,2}
]`

Output



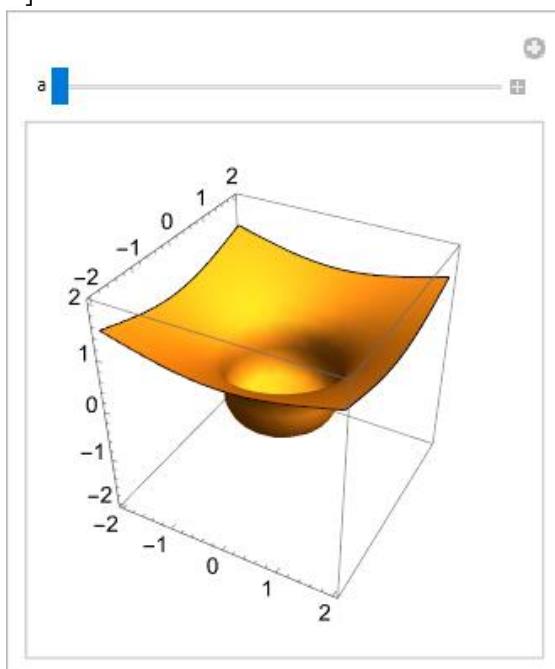
```
Input   Manipulate[  
  Plot[  
    Sin[a x+b],  
    {x,0,6},  
    ImageSize->200  
  ],  
  {a,1,4},  
  {b,0,10}  
]
```

Output



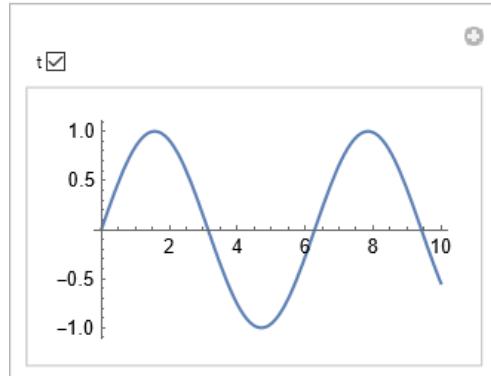
```
Input   Manipulate[  
  ContourPlot3D[  
    x^2+y^2+a z^3==1,  
    {x,-2,2},  
    {y,-2,2},  
    {z,-2,2},  
    Mesh->None,  
    ImageSize->200  
  ],  
  {a,-2,2}  
]
```

Output



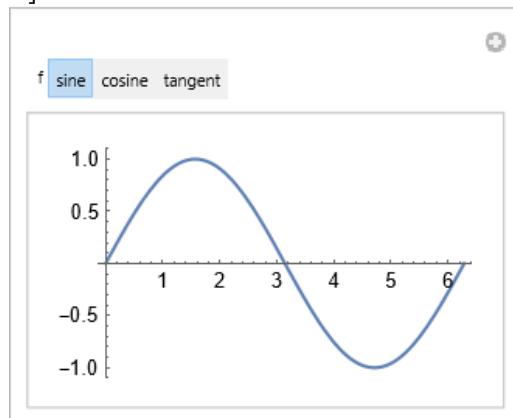
```
Input   Manipulate[
  Plot[
    If[t, Sin[x], Cos[x]],
    {x, 0, 10},
    ImageSize -> 200
  ],
  {t, {True, False}}
]
```

Output



```
Input   Manipulate[
  Plot[
    f[x],
    {x, 0, 2 Pi},
    ImageSize -> 200
  ],
  {f, {Sin -> "sine", Cos -> "cosine", Tan -> "tangent"}}
]
```

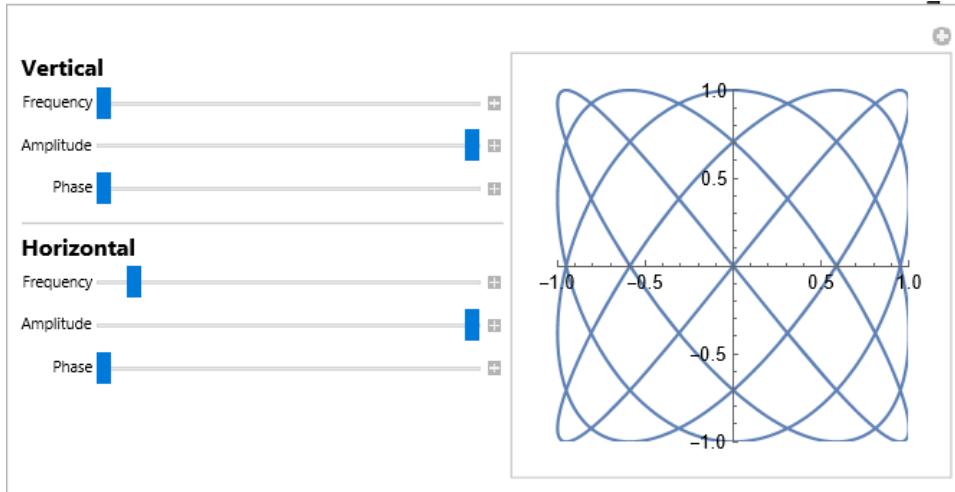
Output



```
Input   Manipulate[
  ParametricPlot[
    {a1 Sin[n1 (x+p1)], a2 Cos[n2 (x+p2)]},
    {x, 0, 20 Pi},
    PlotRange -> 1,
    PerformanceGoal -> "Quality",
    ImageSize -> 200
  ],
  Style["Vertical", Bold, Medium],
  {{n1, 1, "Frequency"}, 1, 4},
  {{a1, 1, "Amplitude"}, 0, 1},
  {{p1, 0, "Phase"}, 0, 2 Pi},
  Delimiter,
  Style["Horizontal", Bold, Medium]
]
```

```
{ {n2, 5/4, "Frequency"}, 1, 4},  
{ {a2, 1, "Amplitude"}, 0, 1},  
{ {p2, 0, "Phase"}, 0, 2 Pi},  
ControlPlacement -> Left  
]
```

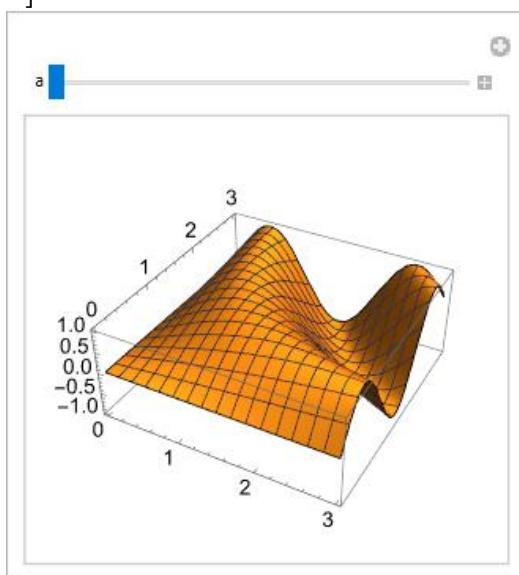
Output



Input

```
Manipulate[  
 Plot3D[  
 Sin[x y+a],  
 {x, 0, 3},  
 {y, 0, 3},  
 ImageSize -> 200  
 ],  
 {a, 0, 1}  
 ]
```

Output



Unit 1.5

Control Structure

Most programming languages use control structures to control the flow of a program. The control structures include decision-making and loops. Decision-making is done by applying different conditions in the program. If the conditions are true, the statements following the condition are executed. The values in a condition are compared by using the comparison operators. The loops are used to run a set of statements several times until a condition is met. If the condition is true, the loop is executed. If the condition becomes false, the loop is terminated, and the control passes to the next statement that follows the loop block.

Conditional Statements

Programmers often need to check the status of a computed intermediate result to branch the program to such or another block of instructions to pursue the computation. Several examples of the branching condition structures are next.

<code>lhs:=rhs;/test</code>	is a definition to be used only if test yields True.
<code>If[test,then,else]</code>	evaluate then if test is True, and else if it is False.
<code>Which[test₁,value₁,test₂,...]</code>	evaluate the test ₁ in turn, giving the value associated with the first one that is True.
<code>Switch[expr,form₁,value₁,form₂,...]</code>	compare expr with each of the form _i , giving the value associated with the first form it matches.
<code>Switch[expr,form₁,value₁,form₂,...,_,def]</code>	use def as a default value.
<code>Piecewise[{{value₁,test₁},{value₂,test₂},...}]</code>	represents a piecewise function with values value _i in the regions defined by the conditions test _i .
<code>Piecewise[{{value₁,test₁},...},def]</code>	give the value corresponding to the first test ₁ which yields True.

Note that,

- 1- `If[condition, t, f]` is left unevaluated if the condition evaluates to neither `True` nor `False`.
- 2- `If[condition, t]` gives `Null` if the condition evaluates to `False`.

Mathematica Examples 1.39

<code>Input</code>	<code>(*If can be used as a statement:*) x=-2; If[x<0, y=-x, y=x];</code>
<code>Output</code>	<code>y 2</code>
<code>Input</code>	<code>(*If can also be used as an expression returning a value:*)x=-2; y=If[x<0,-x,x]</code>
<code>Output</code>	<code>2</code>
<code>Input</code>	<code>If[7>8,x,y</code>

```

Output      ]
Output      y

Input      x=2;
If[
  x==0,Print["x is 0"],Print["x is different from 0"]
]
Output      x is different from 0

Input      x=3;
y=0;
If[
  x>1,y=Sqrt[x],y=x^2
];
Print[y]

m:=If[
  x>5,1,0
];
Print[m]
Output      Sqrt[3]
Output      0

Input      a=2;
Which[
  a==1,x,
  a==2,b
]
Output      b

Input      (* Use True for an else clause that always matches: *)
sign[x_]:=Which[
  x<0,-1,
  x>0,1,
  True,
  Indeterminate
]
{sign[-2],sign[0],sign[3]}
Output      {-1,Indeterminate,1}

Input      (* Use PiecewiseExpand to convert Which to Piecewise: *)
PiecewiseExpand[
  Which[
    c1,a1,
    c2,a2,
    True,a3
  ]
]
Output      {a1   c1
           a2   !c1&&c2
           a3   True

Input      expr=3;
Switch[
  expr,
  1,Print["expr is 1"],
  2,Print["expr is 2"],
  3,Print["expr is 3"],
  _,Print["expr has some other value"]
]
]

```

```

Output expr is 3

Input k=2;
n=0;
Switch[
  k,
  1,n=k+10,
  2,n=k^2+3,
  _,n=-1
];

Print[n]
k=5;
n:=Switch[
  k,
  1,k+10,
  2,k^2+3,
  _,-1
];
Print[n]
Output 7
Output -1

Input (* Find the derivative of a piecewise function: *)
D[
  Piecewise[
    {
      {x^2,x<0},
      {x,x>0}
    }
  ],
  x
]
Output 
$$\begin{cases} 2x & x < 0 \\ 1 & x > 0 \\ \text{Indeterminate} & \text{True} \end{cases}$$


Input (* Define a piecewise function: *)
pw=Piecewise[
  {
    {Sin[x]/x,x<0},
    {1,x==0}
  },
  -x^2/100+1
]
(* Evaluate it at specific points: *)
pw/. {{x->-5},{x->0},{x->5}}
Output 
$$\begin{cases} \frac{\sin(x)}{x} & x < 0 \\ 1 & x == 0 \\ 1 - \frac{x^2}{100} & \text{True} \end{cases}$$

Output {Sin[5]/5,1,3/4}

Input Piecewise[
  {
    {x^2,x<0},
    {x,x>0}
  }
]

```

```

Output  
$$\begin{cases} x^2 & x < 0 \\ x & x > 0 \\ 0 & \text{True} \end{cases}$$

Input   Piecewise[
        {
          {Sin[x]/x, x<0},
          {1, x==0}
        },
        -x^2/100+1
      ]
Output  
$$\begin{cases} \frac{\sin(x)}{x} & x < 0 \\ 1 & x == 0 \\ 1 - \frac{x^2}{100} & \text{True} \end{cases}$$

Input   (*Remove unreachable cases:*)
Piecewise[
  {
    {e1,d1},
    {e2,d2},
    {e3,True},
    {e4,d4},
    {e5,d5}
  },
  e
]
Output  
$$\begin{cases} e1 & d1 \\ e2 & d2 \\ e3 & \text{True} \end{cases}$$

Input   Piecewise[
        {
          {e1,d1},
          {e2,d2},
          {e3,d2&&d3},
          {e4,d4}
        },
        e
      ]
Output  
$$\begin{cases} e1 & d1 \\ e2 & d2 \\ e4 & d4 \\ e & \text{True} \end{cases}$$

Input   x=4;
If[
  x>0,
  y=_sqrt[x],
  y=0
]
Output  2
Input   (* PiecewiseExpand converts nested piecewise functions into a single piecewise
function: *)
pw=Piecewise[
  {
    {Piecewise[{{1,x>=0}},2],Piecewise[{{x,x<=1}},x/2]^2>=1/2}
  },
  3
]

```

```

        ]
PiecewiseExpand[pw]
Output 1
{1 x >= 0      ((x/x) x <= 1)^2 >= 1/2
2 True
3 True
Output 2
1 x >= Sqrt[2]|| 1/Sqrt[2] <= x <= 1
2 x >= Sqrt[2]|| 1/Sqrt[2] <= x <= 1|| x <= -1/Sqrt[2]
3 True
Input (* Min, Max, UnitStep, and Clip are piecewise functions of real arguments: *)
PiecewiseExpand@{
Min[x,y],
Max[x,y,z],
UnitStep[x],
Clip[x,{a,b}]
}
Output 3
{x x-y <= 0, {x x-y >= 0&&x-z >= 0, {1 x >= 0, {a a-x > 0
y True, {y x-y < 0&&y-z >= 0, {0 True, {b b-x < 0&&a-x <= 0
z True
Input (* Abs, Sign, and Arg are piecewise functions when their arguments are assumed to
be real: *)
Assuming[
Element[x,Reals],
PiecewiseExpand/@{Abs[x],Sign[x],Arg[x]}]
Output 4
{-x x < 0, {-1 x < 0, {π x < 0
x True, {1 x > 0, {0 True
Input (* KroneckerDelta and DiscreteDelta are piecewise functions of complex arguments: *)
PiecewiseExpand@{KroneckerDelta[x,y],DiscreteDelta[x,y]}
Output 5
{1 x-y == 0, {1 x == 0&&y == 0
0 True, {0 True
Input (* Derivatives are computed piece-by-piece, unless the function is univariate in a
real variable: *)
D[
Piecewise[
{
{((x^2-1)/(x-1),x!=1}
},
2],
x]
Output 6
2 x
----- - -1 + x^2
-1 + x (-1 + x)^2
0
x ≠ 1
True
Input x=-4;
y=Which[
x>0,1/x,
x<-3,x^2,
True,0
]
Output 16

```

```

Input   a=-4;
       b=4;
       y=Switch[
         a^2,
         a b,1.0/a,
         b^2,1.0/b,
         ,01
       ]
Output  0.25

```

Looping Statements

Mathematica has several looping functions, the most common of which is **Do[]**.

Do [expr,n]	evaluates expr n times.
Do [expr,{i,imax}]	evaluates expr with the variable i successively taking on the values 1 through i_{\max} (in steps of 1).
Do [expr,{i,imin,imax}]	starts with $i = i_{\min}$
Do [expr,{i,imin,imax},di]	uses steps di .
Do [expr,{i,{i₁,i₂,...}}]	uses the successive values i_1, i_2, \dots
Do [expr,{i,imin,imax},{j,jmin,jmax},...]	evaluates expr looping over different values of j etc. for each i .

Mathematica Examples 1.40

```

Input   Do[
         Print[5^k],
         {k,3,7}
       ]
Output  125
       625
       3125
       15625
       78125

Input   t=x;
Do[
         Print[t=1/(1+k t)],
         {k,2,6,2}
       ]
Output  1/(1+2 x)
       1/(1+4/(1+2 x))
       1/(1+6/(1+4/(1+2 x)))

Input   Do[
         Print[{i,j}],
         {i,4},
         {j,i}
       ]
Output  {1,1}
       {2,1}
       {2,2}
       {3,1}
       {3,2}
       {3,3}
       {4,1}
       {4,2}
       {4,3}
       {4,4}

```

```

Input   sum=0;
Do[
  Print[sum=sum+i],
  {i,1,4}
];
sum
Output 1
      3
      6
      10
      10

Input   fact=1;
Do[
  Print[fact=fact*i],
  {i,1,4}
];
fact
Output 1
      2
      6
      24
      24

Input   Do[
  Do[
    Do[
      Print["i= ",i," j= ",j," k= ",k],
      {i,1,2}
    ],
    {j,1,2}
  ],
  {k,1,2}
]
Output i= 1 j= 1 k= 1
      i= 2 j= 1 k= 1
      i= 1 j= 2 k= 1
      i= 2 j= 2 k= 1
      i= 1 j= 1 k= 2
      i= 2 j= 1 k= 2
      i= 1 j= 2 k= 2
      i= 2 j= 2 k= 2

Input   sum=0;
Do[
  Print[i,",",sum=sum+i^2],
  {i,1,6,2}
];
sum
Output 1 , 1
      3 , 10
      5 , 35
Output 35

Input   Do[
  Do[
    If[Sqrt[i^2+j^2]\[Element]Integers,Print[i," ",j]],
    {j,i,10}
  ],
  {i,1,10}
]

```

```
Output      ]
          3   4
          6   8

Input   Do[
        Print[k!],
        {k,3}
      ]
Output  1
        2
        6

Input   Do[
        Print[k, " ", k^2, " ", k^3],
        {k,3}
      ]
Output  1   1   1
        2   4   8
        3   9   27

Input   Do[
        Print[k, " squared is ", k^2],
        {k,5}
      ]
Output  1  squared is  1
        2  squared is  4
        3  squared is  9
        4  squared is 16
        5  squared is 25

Input   Print["k k^2"]
        Print["----"]
        Do[
          Print[k, " ", k^2],
          {k,5}
        ]
Output  k k^2
        -----
        1   1
        2   4
        3   9
        4   16
        5   25

Input   Do[
        Print[k],
        {k,1.6,5.7,1.2}
      ]
Output  1.6
        2.8
        4.
        5.2

Input   Do[
        Print[k],
        {k,3(a+b),8(a+b),2(a+b)}
      ]
Output  3 (a+b)
        5 (a+b)
        7 (a+b)
```

```

Input (* The step can be negative: *)
Do[
  Print[i],
  {i,10,8,-1}
]
Output 10
9
8

Input (* The values can be symbolic: *)
Do[
  Print[n],
  {n,x,x+3 y,y}
]

Output x
x+y
x+2 y
x+3 y

Input (* Loop over i and j,with j running up to i-1: *)
Do[
  Print[{i,j}],
  {i,4},
  {j,i-1}
]
Output {2,1}
{3,1}
{3,2}
{4,1}
{4,2}
{4,3}

Input (*The body can be a procedure:*)
t=67;
Do[
  Print[t];t=Floor[t/2],
  {3}
]
Output 67
33
16

```

<code>Nest[f,expr,n]</code>	apply f to expr n times.
<code>FixedPoint[f,expr]</code>	start with expr, and apply f repeatedly until the result no longer changes.
<code>NestList[f,expr,n]</code>	gives a list of the results of applying f to expr 0 through n times.
<code>While[test,body]</code>	evaluate body repetitively, so long as test is.
<code>For[start,test,incr,body]</code>	executes start, then repeatedly evaluates body and incr until test fails to give True.
<code>Break[]</code>	exits the nearest enclosing Do, For, or While.

Mathematica Examples 1.41

```

Input Nest[
  f,x,3
]
Output f[f[f[x]]]

Input Nest[

```

```
Function[t,1/(1+t)],x,3
]
Output 1/(1+1/(1+1/(1+x)))

Input NestList[
  f,x,4
]
Output {x,f[x],f[f[x]],f[f[f[x]]],f[f[f[f[x]]]]}

Input NestList[
  Cos,1.0,10
]
Output {1.,0.540302,0.857553,0.65429,0.79348,0.701369,0.76396,0.722102,0.750418,0.731404,0
.744237}

Input FixedPoint[
  Function[t,Print[t];Floor[t/2]],67
]
Output 67
33
16
8
4
2
1
0
Output 0

Input n=17;
While[
  n=Floor[n/2];n!=0,
  Print[n]
]
Output 8
4
2
1

Input n=1;
While[
  n<4,Print[n];
  n=n+1
]
Output 1
2
3

Input Do[
  Print[i];
  If[i>2,Break[],
  {i,10}
]
Output 1
2
3

Input For[
  i=1;t=x,
  i^2<10,
  i=i+1,
```

```
t=t^2+i;
Print[t]
]
Output 1+x^2
2+(1+x^2)^2
3+(2+(1+x^2)^2)^2

Input  For[
  sum=0.0;x=1.0,
  (1/x)>0.15,
  x=x+1,
  sum=sum+1/x;
  Print[sum]
]
Output 1.
1.5
1.83333
2.08333
2.28333
2.45
```

Unit 1.6

Modules, Blocks, and Local Variables

Global Variables are those variables declared in Main Program and can be used by Subprograms. Local Variables are those variables declared in Subprograms. The Wolfram Language normally assumes that all your variables are global. This means that every time you use a name like x , the Wolfram Language normally assumes that you are referring to the same object. Particularly when you write subprograms, however, you may not want all your variables to be global. You may, for example, want to use the name x to refer to two quite different variables in two different subprograms. In this case, you need the x in each subprogram to be treated as a local variable. You can set up local variables in the Wolfram Language using modules. Within each module, you can give a list of variables that are to be treated as local to the module.

<code>Module[{x,y,...},body]</code>	a module with local variables x , y ,
<code>Module[{x=x0,y=y0,...},body]</code>	a module with initial values for local variables.

Mathematica Examples 1.42

```

Input   k=25
Output  25

Input   Module[
{k},
Do[
Print[k," ",2^k],
{k,3}
]
]
Output  1  2
        2  4
        3  8

Input   k
Output  25

```

Thus, we can create programs as a series of modules, each performing a specific task. For subtasks, we can embed modules within other modules to form a hierarchy of operations. The most common method for setting up modules is through function definitions,

Mathematica Examples 1.43

```

Input   k=25;
integerPowers[x_Integer]:=Module[
{k},
Do[
Print[k," ",x^k],
{k,3}
]
]
integerPowers[k]
Output  1  25
        2  625
        3  15625

```

```

Input   k
Output  25

Input   t=17
Output  17

Input   Module[
  {t},
  t=8;Print[t]
]
Output  8

Input   t
Output  17

Input   g[u_]:=Module[
  {t=u},
  t+=t/(1+u)
]
Input   g[a]
Output  a + a/(1+a)

Input   h[x_]:=Module[
  {t},
  t^2-1/(t=x-4)>1
]
Input   h[10]
Output  35

```

The format of `Module` is `Module[{var1, var2, ...}, body]`, where `var1, var2, ...` are the variables we localize, and `body` is the body of the function. The value returned by `Module` is the value returned by the last operator in the body (unless an explicit `Return[]` statement is used within the body of `Module`. In this case, the argument of `Return[arg]` is returned). In particular, if one places the semicolon after this last operator, nothing (`Null`) is returned. As a variant, it is acceptable to initialize the local variables in the place of the declaration, with some global values: `Module[{var1 = value1, var2, ...}, body]`. However, one local variable (say, the one "just initialized" cannot be used in the initialization of another local variable inside the declaration list. The following would be a mistake: `Module[{var1 = value1, var2 = var1, ...}, body]`. Moreover, this will not result in an error, but just the global value for the symbol `var1` would be used in this example for the `var2` initialization (this is even more dangerous since no error message is generated and thus we don't see the problem.) In this case, it would be better to do initialization in steps: `Module[{var1=value1,var2,...}, var2=var1;body]`, that is, include the initialization of part of the variables in the body of `Module`. One can use `Return[value]` statement to return a value from anywhere within the `Module`. In this case, the rest of the code (if any) inside `Module` is skipped, and the result value is returned.

To show how this is done, the following code is an example of a module which will simulate a single gambler playing the game until the goal is achieved or the money is gone.

Mathematica Examples 1.44

```

Input   GamblersRuin[a_,c_,p_]:=Module[
  {ranval,var1,var2,var3},
  var1=a;
  var2=c;
  var3=p;
  While[
    0<var1<var2,
    ranval=Random[];
    If[
      ranval<var3,
      var1=var1+1,

```

```

    var1=var1-1
  ]
];
Return[
  var1==var2
]

```

There are several things to notice in this example. First, this is the same thing we have done in the past to define a function. That is, we have a function name `GamblersRuin` with three input variables, `a`, `c`, and `p`. The operator `:=` is used to start the definition. Secondly, the function involves the Mathematica command `Module`. This just tells Mathematica to perform all the commands in the module (like a subroutine in Fortran or a method in C++). There are some special features we need to understand in the `Module` command. The `Module` command has two arguments. The first argument is a list of all the local variables that will only be used inside the module. In the above example, the local variable list is `{ranval, var1, var2, var3}`. These variables are only used in the module and are cleared once the module has been executed. The second argument is all the commands that will be executed each time the module is called. There are some assignment commands at the beginning that are used to make things cleaner. The module uses temporary variables so that the values of the input variables are not overwritten when the module executes.

The last command is added to our list of input lines to return a result from the work done by the `Module`. Without this we would never get any results from our calculation. Any recognized variable type or structure within Mathematica can be returned by a `Module`. In this example, the returned value is the result of testing two variables in the code for equality. The code fragment `var1==var2` tests to determine if the variables, `var1` and `var2`, are equal. If the two variables are equal, then the line outputs `True` and if they are not equal, the line outputs `False`.

Again, all but the last command must be ended by a semicolon. This is to make sure that the commands are separated in the execution. Commands separated by blank spaces will be considered as terms to be multiplied together. Leaving out the semicolon will give rise to lots of error messages, wrong results or both.

Modules in Mathematica allow one to treat the names as local. When one uses `Block` then the names are global, but the values are local.

<code>Block[{x,y,...},body]</code>	evaluate expr using local values for x,y,
<code>Block[{x=x0,y=y0,...},body]</code>	assign initial values to x,y; and evaluate ... as above.

`Block[]` is automatically used to localize values of iterators in iteration constructs such as `Do`, `Sum`, and `Table`. `Block[]` may be used to pack several expressions into one unity.

Mathematica Examples 1.45

```

Input  Clear[x,t,a]
Input  x^2+3
Output 3+x^2

Input  Block[{x=a+1},%]
Output 3+(1+a)^2

Input  x
Output x

Input  t=17
Output 17

Input  Module[
  {t},
  Print[t]
]

```

```
Output t$6220
```

```
Input t
```

```
Output 17
```

```
Input Block[  
  {t},  
  Print[t]  
 ]
```

```
Output t
```

Unit 1.7

Functional Programming

In the Wolfram Language, functional programming stands as a highly evolved and seamlessly integrated core feature. This integration is significantly enhanced by the language's symbolic nature, which imparts richness and convenience. The approach of treating expressions such as `f[x]` as both symbolic data and the application of a function `f` establishes a uniquely potent method to harmonize structure and function. This synthesis yields an efficient, elegant representation of numerous common computations.

In the realm of functional programming, several key techniques contribute to the expressive power of the Wolfram Language:

1. Pure functions: The Wolfram Language embraces the concept of pure functions, enabling the creation of anonymous functions without the need for formal function definitions. This concise and powerful feature enhances the flexibility of functional programming.
2. Applying functions repeatedly: Operations like `Nest` facilitate the iterative application of functions, allowing for the creation of sequences or the exploration of dynamic processes through successive function applications.
3. Applying functions to lists and other expressions: Functions such as `Map`` (`/@``), `Apply`` (`@@``), and `MapThread`` facilitate the application of functions to lists and expressions, offering versatile tools for element-wise operations, function application, and coordination across multiple expressions.
4. Building lists from functions: The `Array`` function, among others, provides a convenient means to generate lists based on specified functions. This capability is crucial for efficient data generation and manipulation in a functional programming paradigm.

Pure Functions

Pure functions are a fundamental concept in functional programming and are used extensively in the Wolfram Language. When engaging in functional operations like `Nest` and `Map`, it is essential to explicitly specify a function for application. Throughout the examples, we consistently utilized the 'name' of a function to define and apply the operation. Pure functions provide a more flexible approach, enabling the definition and application of functions directly to arguments without the need for explicit function names. There are several equivalent ways to write pure functions in the Wolfram Language.

<code>Function[x, body]</code>	is a pure function with a single formal parameter <code>x</code> .
<code>or</code>	
<code>x ->body</code>	
<code>or</code>	
<code>x\[Function]body</code>	
<code>Function[{x1, x2, ...}, body]</code>	a pure function that takes several arguments
<code>body& or Function[body]</code>	is a pure (or "anonymous") function. The formal parameters are <code>#</code> (or <code>#1</code>), <code>#2</code> , etc.

Where the character \mapsto is entered as `|->`, `fn` or `\[Function]`.

Mathematica Examples 1.46

```

Input  (* This code defines a pure function taking one parameter `v` and returns `5 + v`. When applied with `y`, it results in `5 + y`: *)
Function[v, 5 + v][y]
Output 5+y

Input  (* This is an equivalent way of expressing the previous pure function, using the `#` shorthand for the parameter. It also results in `5 + y`: *)
Function[5 + #][y]
Output 5+y

Input  (* Another way to write the same expression, using the slot Operator (`#`) directly: *)
(5 + #) &[y]
Output 5+y

Input  (* A pure function taking two parameters `u` and `v` and returning  $u^4 + v^3$ : *)
Function[{u, v}, u^4 + v^3][x, y]
Output x^4+y^3

Input  (* An alternative way to express the two-parameter pure function using slots: *)
(#1^4 + #2^3) &[x, y]
Output x^4+y^3

Input  (* This code assigns the pure function `5 + #` to the symbol `f`: *)
f = (5 + #) &
Output 5+#1&

Input  (* The use of the previously assigned pure function `f`: *)
{f[a], f[b]}
Output {5+a,5+b}

Input  (* Uses `Select` to choose elements greater than 0 from the list: *)
Select[{1, -1, 2, -2, 3}, # > 0 &]
Output {1,2,3}

Input  (* Sorts a list of pairs based on the second part of each element: *)
Sort[{{a, 3}, {c, 1}, {d, 2}}, #1[[2]] < #2[[2]] &]
Output {{c,1},{d,2},{a,3}}

```

Association

In the Wolfram Language, associations stand as key-value pairs, playing a pivotal role alongside lists. These structures efficiently link keys to corresponding values, enabling rapid lookup and seamless updates, even when dealing with extensive datasets containing millions of elements. An Association acts like a symbolically indexed list. The value associated with a given key can be extracted by using the part specification `Key[key]`. If key is a string, `Key` can be omitted.

`Association[key1→val1, key2→val2, ...]` represents an association between keys and values.
or
`<|key1→val1, key2→val2, ...|>`

Mathematica Examples 1.47

```

Input  (* This code creates an association with keys `a`, `b`, and `c` associated with values `x`, `y`, and `z`, respectively: *)
<|a->x,b->y,c->z|>
Output <|a->x,b->y,c->z|>

```

```

Input  (* Extracting the value associated with the key `b`: *)
<|a->x,b->y,c->z|>;
%[b]
Output y

Input  (* Converts a list of rules to an association, resulting in the same association as
in the first example: *)
Association[{a->x,b->y,c->z}]
Output <|a->x,b->y,c->z|>

Input  (* Converts an association to a list of rules: *)
Normal[<|a->x,b->y,c->z|>]
Output {a->x,b->y,c->z}

Input  (* Uses `Position` to find the position of the value `2` in the association: *)
Position[<|a->4,b->2,c->1,d->5|>,2]
Output {{Key[b]}}
```

```

Input  (* Appends a new key-value pair `d->w` to the existing association: *)
Append[<|a->x,b->y,c->z|>,d->w]
Output <|a->x,b->y,c->z,d->w|>

Input  (* Extracts the value associated with key `b` as if it were a part: *)
<|a->x,b->y,c->z|>[[Key[b]]]
Output y

Input  (* When the key in an association is a string, no explicit `Key` is needed to
extract it as a part: *)
<|"a"->x,"b"->y,"c"->z|>[["b"]]
Output y

Input  (* Shows that numerical part specifications work structurally on associations,
extracting the second element (value associated with key `b`): *)
<|a->x,b->y,c->z|>[[2]]
Output y

Input  (* Demonstrates how lookups in associations interoperate with parts in lists and
other expressions: *)
{<|a->x,b->{y,z}|>}[[1,Key[b],2]]
Output z

Input  (* Uses `Keys` to extract the list of keys from the association: *)
Keys[<|a->x,b->y,c->z|>]
Output {a,b,c}

Input  (* This code uses named arguments from an association, extracting `u` and `v`: *)
g[#u, #v, #u] &[<|"u" -> x, "v" -> y|>]
Output g[x,y,x]

```

Applying functions to lists

When working with a list of elements in the Wolfram Language, it becomes crucial to apply operations independently to each element. The `Map` function comes into play as a versatile mechanism for achieving this. With `Map`, you can effortlessly apply a specified function to every individual element within a list. This functionality enhances the expressiveness and efficiency of your code, allowing for seamless transformations across diverse datasets. Whether you're manipulating numerical data, symbolic expressions, or custom-defined functions, `Map` provides a concise and powerful approach to element-wise operations in the Wolfram Language.

<code>Map[f,expr]</code> or <code>f/@expr</code>	applies <code>f</code> to each element on the first level in <code>expr</code> .
<code>Map[f,expr,levelspec]</code>	applies <code>f</code> to parts of <code>expr</code> specified by <code>levelspec</code> .
<code>Map[f]</code>	represents an operator form of <code>Map</code> that can be applied to an expression.

Mathematica Examples 1.48

Input	(* Applies the function `f` to each element of the list `{a, b, c, d, e}`: *) <code>Map[f, {a, b, c, d, e}]</code>
Output	{ <code>f[a],f[b],f[c],f[d],f[e]</code> }
Input	(* Another way to express the same operation using the short input form: *) <code>f /@ {a, b, c, d, e}</code>
Output	{ <code>f[a],f[b],f[c],f[d],f[e]</code> }
Input	(* Demonstrates using explicit pure functions with `Map`. The code applies the function `1 + g[#]` to each element of a list: *) <code>(1 + g[#]) & /@ {a, b, c, d, e}</code>
Output	{ <code>1+g[a],1+g[b],1+g[c],1+g[d],1+g[e]</code> }
Input	(* Applies the function `f` at the top level of the nested list: *) <code>Map[f, {{a, b}, {c, d, e}}]</code>
Output	{ <code>f[{a,b}],f[{c,d,e}]</code> }
Input	(* Applies the function `f` at level 2 of the nested list: *) <code>Map[f, {{a, b}, {c, d, e}}, {2}]</code>
Output	{ <code>{f[a],f[b]},{{f[c],f[d],f[e]}}</code> }
Input	(* Mapping at levels 1 and 2: *) <code>Map[f, {{a, b}, {c, d, e}}, 2]</code>
Output	{ <code>f[{f[a],f[b]}],f[{f[c],f[d],f[e]}]</code> }
Input	(* Uses a map operator to apply the function `f` to each element of the list: *) <code>Map[f][{a, b, c, d}]</code>
Output	{ <code>f[a],f[b],f[c],f[d]</code> }
Input	(* Applies the function `h` to the values in the association: *) <code>Map[h, < a -> b, c -> d >]</code>
Output	{ <code>< a->h[b],c->h[d] ></code> }

In the Wolfram Language, the `Map` function is a powerful tool for applying a function to each element of a list. However, there are situations where you may need to apply a function of several arguments to corresponding parts of different expressions. This is where the `MapThread` function comes into play. While `Map` operates on a single list, `MapThread` allows you to simultaneously traverse multiple lists, applying a specified function that takes arguments from corresponding positions in each list. This functionality is particularly useful for handling data in a pairwise or element-wise manner across several expressions.

<code>MapThread[f,{{a1,a2,...},{b1,b2,...},...}]</code>	gives { <code>f[a1,b1,...],f[a2,b2,...],...</code> }.
<code>MapThread[f,{expr1,expr2,...},n]</code>	applies <code>f</code> to the parts of the <code>expr</code> at level <code>n</code> .
<code>MapThread[f]</code>	represents an operator form of <code>MapThread</code> that can be applied to an expression.

Mathematica Examples 1.49

Input	(* Applies the function `f` to corresponding pairs of elements from the given lists:*) <code>MapThread[f,{{a,b,c},{x,y,z}}]</code>
Output	{ <code>f[a,x],f[b,y],f[c,z]</code> }

Input	(* Applies the function `f` to corresponding elements of a matrix (nested lists), considering elements at level 2: *) MapThread[f, {{{a,b},{c,d}},{{u,v},{s,t}}},2]
Output	{f[a,u],f[b,v]},{f[c,s],f[d,t]}}
Input	(* Applies the function `f` to corresponding values of associations: *) MapThread[f,{< a->1 >,< a->2 >}]
Output	< a->f[1,2] >
Input	(* Uses the operator form of `MapThread` to apply the function `f` to corresponding elements of the given lists: *) MapThread[f][{{a,b,c,d},{1,2,3,4}}]
Output	{f[a,1],f[b,2],f[c,3],f[d,4]}

When dealing with an expression like `f[{a, b, c}]`, you're providing a list as the argument to a function. However, there are situations where you may want to apply a function directly to the elements of the list, treating each element as a separate argument. This is where the **Apply** function, denoted by `@@`, comes into play. Instead of applying the function to the list as a whole, **Apply** allows you to transform the list into individual arguments. The **Apply** function is used to change the head of an expression or to "apply" a function to a list of arguments.

<code>f@@expr</code>	replaces the head of <code>expr</code> by <code>f</code> .
or	
<code>Apply[f,expr]</code>	
<code>Apply[f,expr,levelspec]</code>	replaces heads in parts of <code>expr</code> specified by <code>levelspec</code> . <code>Apply[f]</code> represents an operator form of <code>Apply</code> that can be applied to an expression.

Mathematica Examples 1.50

Input	(* Replaces the head of the list `{a,b,c,d}` with the function `f`: *) Apply[f,{a,b,c,d}]
Output	<code>f[a,b,c,d]</code>
Input	(* Provides an equivalent way to achieve the same result using the `@@` operator: *) f@@{a,b,c,d}
Output	<code>f[a,b,c,d]</code>
Input	(* Sums the elements of the list `{1,2,3,4}` by replacing the head with `Plus`: *) Plus@@{1,2,3,4}
Output	<code>10</code>
Input	(* Applies `f` to the list `{{a,b},{c},d}` , getting rid of one level of lists: *) f@@{{a,b},{c},d}
Output	<code>f[{a,b},{c},d]</code>
Input	(* The operator form of `Apply` to achieve the same result as in the previous example: *) Apply[f][{{a,b},{c},d}]
Output	<code>f[{a,b},{c},d]</code>
Input	(* Applies the function `f` to an association, keeping only the values: *) Apply[f,< 1->a,2->b,3->c >]
Output	<code>f[a,b,c]</code>
Input	(* Applying `List` to an association is equivalent to using `Values` on the association: *) Apply[List,< 1->a,2->b,3->c,4->{d} >]

```
Output Values[<|1->a,2->b,3->c,4->{d}|>]
          {a,b,c,{d}}
          {a,b,c,{d}}
```

`MapApply` is equivalent to using `Apply` on parts at level 1:

<code>f@@@expr</code>	replaces heads at level 1 of expr by f.
<code>or</code>	
<code>MapApply[f,expr]</code>	

<code>MapApply[f]</code>	represents an operator form of <code>MapApply</code> that can be applied to an expression.
--------------------------	--

Mathematica Examples 1.51

```
Input (* The `MapApply` function replaces the head of each sublist with the function `f`:*)
      MapApply[f, {{a, b}, {c, d}}]
Output {f[a,b],f[c,d]}

Input (* An equivalent operation using the `@@@` operator, applying `f` to each element of
       the list: *)
      f @@@ {{a, b}, {c, d}}
Output {f[a,b],f[c,d]}

Input (* Summing Sublists by Replacing the Head with Plus: *)
      Plus @@@ {{a, b, c, d}, {1, 2, 3, 4}, {e, f, 5, 6}}
Output {a+b+c+d,10,11+e+f}

Input (* This demonstrates the use of the operator form of the assumed `MapApply` function,
       applying `f` to each element of the nested list: *)
      MapApply[f][{{a, b}, {c}, {d, e}}]
Output {f[a,b],f[c],f[d,e]}

Input (* MapApply Affects Only the Values of an Association: *)
      MapApply[f, <|{1, 2} -> {a}, {3, 4} -> {c, d, e}|>]
Output <|{1,2}->f[a],{3,4}->f[c,d,e]|>
```

Applying Functions Repeatedly

Many programs you write will involve operations that need to be iterated several times. The `Nest` function in the Wolfram Language is a powerful tool for iterative computation. Operations like `Nest` take a function `f` that operates on a single argument and apply it successively. In each iteration, the result of the previous step becomes the new argument for `f`. This recursive application continues for the specified number of iterations, allowing for the creation of sequences.

<code>Nest[f,expr,n]</code>	gives an expression with f applied n times to expr.
-----------------------------	---

Mathematica Examples 1.52

```
Input (* This code nests the function `f` three times, starting with the initial value `x`:
      *)
      Nest[f,x,3]
Output f[f[f[x]]]

Input (* The function to nest can be a pure function. In this case, it squares and adds 1
      to the initial value `1` three times: *)
      Nest[(1+#)^2&,1,3]
```

```

Output 676
Input (* This example shows how nesting a function can build a formula. The function
`(1+#)^2&` is nested five times, starting with the variable `x`: *)
Nest[(1+#)^2&,x,5]
Output (1+(1+(1+(1+x)^2)^2)^2)^2

Input (* Nesting can return a single number. In this case, the square root of `100.0` is
taken four times: *)
Nest[Sqrt,100.0,4]
Output 1.33352

```

In the Wolfram Language, operations like `Nest` are designed to iteratively apply a function `f` of one argument, using the result of the previous step as the new argument for each subsequent iteration. This concept becomes even more powerful when generalized to functions of two arguments. In such cases, the iterative application continues, but each result obtained provides only one of the new arguments required. A practical approach is to obtain the other argument at each step from the successive elements of a list, creating a flexible and dynamic process for manipulating data and performing computations efficiently.

<code>FoldList[f,{a,b,c,...}]</code>	gives $\{a, f[a, b], f[f[a, b], c], \dots\}$.
<code>Fold[f,x,list]</code>	gives the last element of <code>FoldList[f,x,list]</code> .
<code>Fold[f,list]</code>	is equivalent to <code>Fold[f,First[list],Rest[list]]</code> .
<code>Fold[f]</code>	represents an operator form of <code>Fold</code> that can be applied to expressions.

Mathematica Examples 1.53

```

Input FoldList[f,x,{a,b,c,d}]
Output {x,f[x,a],f[f[x,a],b],f[f[f[x,a],b],c],f[f[f[f[x,a],b],c],d]}

Input (* This code successively applies the function `f` to the seed value `x` and the
elements of the list `{a, b, c, d}`: *)
Fold[f, x, {a, b, c, d}]
Output f[f[f[f[x,a],b],c],d]

Input (* Using `Fold` with the `List` function to create nested ordered pairs: *)
Fold[List, x, {a, b, c, d}]
Output {{{{x,a}},b},c},d

Input (* This code multiplies the elements of the list `{a, b, c, d}` one element at a
time, starting with the seed value `1`: *)
Fold[Times, 1, {a, b, c, d}]
Output a b c d

Input (* Starting the fold operation from the first element of the list: *)
Fold[f, {a, b, c, d}]
Output f[f[f[a,b],c],d]

```

Building Lists from Functions

<code>Array[f,n]</code>	generates a list of length n , with elements $f[i]$.
<code>Array[f,n,r]</code>	generates a list using the index origin r .
<code>Array[f,n,{a,b}]</code>	generates a list using n values from a to b .

Mathematica Examples 1.54

```
Input (* This code generates an array of length 10, where each element is obtained by
       applying the function `f` to its corresponding index: *)
       Array[f,10]
Output {f[1],f[2],f[3],f[4],f[5],f[6],f[7],f[8],f[9],f[10]}

Input (* Generating an array using a different function. In this case, the function is
       `1+#+2&` :*)
       Array[1+#+2&,10]
Output {2,5,10,17,26,37,50,65,82,101}

Input (* Generates a 3x2 array where each element is obtained by applying the function `f`
       to its indices: *)
       Array[f,{3,2}]
Output {{f[1,1],f[1,2]},{f[2,1],f[2,2]},{f[3,1],f[3,2]}}

Input (* Creates a 3x4 array using a custom function: *)
       Array[10 #1+#+2&,{3,4}]
Output {{11,12,13,14},{21,22,23,24},{31,32,33,34}]

Input (* Generates an array with index origin 0: *)
       Array[f,10,0]
Output {f[0],f[1],f[2],f[3],f[4],f[5],f[6],f[7],f[8],f[9]}

Input (* Creates a 2x3 array starting with indices 0 and 4: *)
       Array[f,{2,3},{0,4}]
Output {{f[0,4],f[0,5],f[0,6]},{f[1,4],f[1,5],f[1,6]}}

Input (* Generates an array where indices are sampled between 0 and 1: *)
       Array[f,10,{0,1}]
Output {f[0],f[1/9],f[2/9],f[1/3],f[4/9],f[5/9],f[2/3],f[7/9],f[8/9],f[1]}

Input (* Generates an array of length 10 using the pure function `3 + #^3 &` : *)
       Array[3 + #^3 &, 10]
Output {2,5,10,17,26,37,50,65,82,101}
```


CHAPTER 2

DESCRIPTIVE STATISTICS AND PROBABILITY THEORY

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 1 of the book [1] titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation. This chapter also serves as a summary of the book titled *Statistics for Machine Learning with Mathematica Applications*. For detailed proofs of theorems, additional examples, and comprehensive explanations, including Mathematica applications, please refer to Ref [2]."

Descriptive statistics form the foundational toolset for summarizing and interpreting data, offering insights into central tendencies and variability within datasets [3-10]. It serves as the initial lens through which raw data is scrutinized, paving the way for more advanced analyses. On the other hand, probability theory plays a crucial role in quantifying uncertainty and randomness inherent in data. It underpins the probabilistic nature of real-world phenomena, intertwining with statistical measures to create a comprehensive framework for data analysis.

In artificial intelligence (AI) applications, probability theory serves two fundamental purposes:

1. Guiding the rationale behind AI reasoning processes, shaping algorithmic design to compute or approximate expressions derived from probability.
2. Providing an analytical framework to dissect and evaluate AI system behavior, assessing performance under diverse conditions and the implications of algorithmic choices.

By integrating probability and statistics in AI development, researchers gain insights into theoretical foundations, contributing to the refinement and optimization of intelligent systems.

Mathematica provides powerful tools for analyzing and visualizing data, including built-in functions for random sampling, order and count statistics, frequency distributions, and distribution shapes. By using these functions, you can gain insights into your data, identify patterns and outliers, and make informed decisions based on statistical analysis. Additionally, Mathematica offers various functions to compute central tendency, dispersion and shape measures. These functions provide quick and accurate calculations for determining the typical or central value of a dataset and for visualizing location statistics.

- Random sampling is a method of selecting a subset of individuals or data points from a larger population in a way that each member of the population has an equal chance of being selected. In Mathematica, you can use built-in functions such as `RandomSample` and `RandomChoice` to generate random samples from a dataset or population. These functions can be useful for testing hypotheses, simulating experiments, and exploring datasets.
- Also, you can use the built-in function `Histogram` to generate a histogram of the frequency distribution of a dataset.
- Moreover, you can use the built-in functions `PDF`, and `CDF`, to visualize the shape of a distribution.
- The `Mean` function in Mathematica calculates the arithmetic mean of a list of numbers. It is a commonly used measure of central tendency and provides the average value of the dataset.
- The `Median` function computes the middle value of a sorted dataset. It is useful for finding a representative value that is not influenced by extreme values or outliers.

- The **Commonest** function determines the mode(s) of a dataset, which represents the most frequently occurring value(s). This can be useful when dealing with categorical or discrete data.
- Mathematica provides functions like **Quartiles** and **Quantile** to calculate specific quantiles of a dataset. These functions allow you to find values that divide the dataset into equal proportions, such as the first quartile (25th percentile) or the median (50th percentile).
- The **TrimmedMean** function in **Mathematica** calculates the mean of a dataset after excluding a specified percentage of extreme values from both ends. It can be useful in situations where outliers may significantly affect the overall mean.
- The **WinsorizedMean** function provides a robust measure of central tendency by reducing the impact of outliers or extreme values on the calculated mean. It achieves this by replacing extreme values with values from a specified percentile.
- Both the **HarmonicMean** and **GeometricMean** functions provide alternative measures of central tendency that are suitable for specific types of data. While the **HarmonicMean** is useful for rates and ratios, the **GeometricMean** is applicable for multiplicative relationships and positive values.
- Mathematica offers built-in functions for visualizing location statistics. For example, you can create box plots using **BoxWhiskerChart** to display the median, quartiles, and potential outliers in a dataset.
- The **InterquartileRange** function calculates the range between the upper quartile and the lower quartile in a dataset. It is useful for identifying the spread or dispersion of the middle 50% of the data.
- The **QuartileDeviation** function calculates the semi-interquartile range, which is half of the Interquartile Range. It provides a measure of dispersion around the median and is less affected by extreme values.
- The **MeanDeviation** function computes the average absolute deviation of each data point from the mean. It gives an indication of the average distance between individual data points and the mean. **MeanDeviation** is less influenced by extreme values and provides a robust measure of dispersion.
- The **StandardDeviation** function calculates the standard deviation, which is a widely used measure of dispersion. It quantifies the amount of variation or spread in a dataset by measuring the average distance between each data point and the mean. A higher standard deviation indicates greater variability.
- The **Variance** function computes the average squared deviation of each data point from the mean. It provides a measure of the overall variability in a dataset.
- The **TrimmedVariance** function calculates the variance after trimming a certain percentage of extreme values from both ends of the dataset. Trimming reduces the impact of outliers and extreme values on the variance calculation, providing a more robust measure of dispersion.
- The **WinsorizedVariance** function is similar to **TrimmedVariance**, but instead of removing extreme values, it replaces them with values closer to the mean.
- The **Moment** function computes the nth moment of a dataset.
- The **CentralMoment** function calculates the nth central moment of a dataset. It measures the dispersion of data around the mean.
- The **FactorialMoment** function computes the nth factorial moment of a dataset.
- The **Skewness** function measures the asymmetry of a dataset's distribution. It indicates whether the dataset is skewed to the left (negative skewness) or to the right (positive skewness) relative to the mean.
- The **QuartileSkewness** function is a measure of skewness based on quartiles.
- The **Kurtosis** function measures the peakedness or flatness of a dataset's distribution. It provides insights into the tail behavior and presence of outliers.

The chapter will include practical examples and exercises to reinforce the concepts learned. By the end of this chapter, you will be equipped with the knowledge and skills to perform descriptive statistical analysis using Mathematica, making it easier to summarize and interpret data effectively.

Moreover, we will delve into the world of discrete random variables, which play a crucial role in modeling various phenomena with countable outcomes. We explore their Probability Mass Functions (PMFs), Cumulative Density Functions (CDFs), and Moment Generating Functions (MGFs), while leveraging the power of Mathematica to perform computations and gain insights.

- `PDF` and `CDF` assist in determining the probability density and cumulative probability of discrete random variables, respectively.
- `Expectation` and `NExpectation` functions calculate the expected value or mean of a discrete random variable, providing insights into its central tendency.
- `MomentGeneratingFunction` and `CentralMomentGeneratingFunction` allow for the calculation of higher-order moments and central moments, offering a more comprehensive understanding of the distribution of the random variable.
- Mathematica offers a comprehensive set of built-in functions to handle various probability distributions effortlessly. In this chapter, we also explore two essential probability distributions: Binomial distribution, and discrete uniform distribution.

Additionally, we explore the world of continuous random variables, which are essential for modeling phenomena with uncountable outcomes. We study the continuous probability distributions, exploring their Probability Density Functions (PDFs), CDFs, and MGFs. We will demonstrate how to define, manipulate, and analyze continuous random variables and probability distributions using Mathematica's syntax and functionality. We will study two fundamental probability distributions: Normal distribution, and uniform distribution.

By engaging in hands-on exercises and experimenting with different scenarios, readers will enhance their understanding of the concepts and develop proficiency in utilizing Mathematica for probability analysis.

Unit 2.1

Descriptive Statistics

Mathematica provides a wide range of functions for generating different types of random data, such as integers, real numbers, vectors, matrices, and graphs which can be useful for several applications, such as simulations, modeling, and data analysis.

1. When generating random data, it is important to specify the range and distribution of the data. For example, you can use the `UniformDistribution` function to generate data uniformly distributed between a specified minimum and maximum value, or the `NormalDistribution` function to generate data following a normal distribution with a specified mean and standard deviation.
2. The random data generated by Mathematica's built-in functions is pseudo-random, meaning that the sequence of numbers generated is deterministic and depends on the seed value. It is important to set a specific seed value using the `SeedRandom` function if you want to reproduce the same sequence of random numbers.
3. One of the key advantages of Mathematica's random data generation functions is their integration with other mathematical and statistical functions in the program. This allows users to easily incorporate generated random data sets into larger simulations and models, and to analyze the results using a wide range of tools.

<code>RandomInteger[{imin,imax}]</code>	gives a pseudorandom integer in the range {imin,imax}.
<code>RandomInteger[imax]</code>	gives a pseudorandom integer in the range {0,...,imax}.
<code>RandomInteger[]</code>	pseudo randomly gives 0 or 1.
<code>RandomReal[]</code>	gives a pseudorandom real number in the range 0 to 1.
<code>RandomReal[{xmin,xmax}]</code>	gives a pseudorandom real number in the range xmin to xmax.
<code>RandomPoint[reg]</code>	gives a pseudorandom point uniformly distributed in the region reg.
<code>RandomPoint[reg,n]</code>	gives a list of n pseudorandom points uniformly distributed in the region reg.
<code>RandomChoice[{e1,e2,...}]</code>	gives a pseudorandom choice of one of the ei.
<code>RandomChoice[list,n]</code>	gives a list of n pseudorandom choices.
<code>RandomSample[{e1,e2,...},n]</code>	gives a pseudorandom sample of n of the ei.
<code>RandomSample[{w1,w2,...} \rightarrow {e1,e2,...},n]</code>	gives a pseudorandom sample of n of the ei chosen using weights wi.
<code>RandomSample[{e1,e2,...}]</code>	gives a pseudorandom permutation of the ei.
<code>RandomVariate[dist]</code>	gives a pseudorandom variate from the symbolic distribution dist.
<code>RandomVariate[dist,n]</code>	gives a list of n pseudorandom variates from the symbolic distribution dist.
<code>RandomVariate[dist,{n1,n2,...}]</code>	gives an $n_1 \times n_2 \times \dots$ array of pseudorandom variates from the symbolic distribution dist.
<code>SeedRandom[s]</code>	resets the pseudorandom generator, using s as a seed.
<code>SeedRandom[]</code>	resets the generator, using as a seed the time of day and certain attributes of the current Wolfram System session.

Mathematica code 2.1

RandomInteger

```
Input      (* This code generates random data for a scatter plot by using the RandomInteger
           function with the interval {1,20}. The data variable is a list of n pairs of random
           values generated using the Table function. The data is plotted using the ListPlot
           function: *)
n=100;
data=Table[
{RandomInteger[{1,20}],RandomInteger[{1,20}]},
```

```
{n}
];

ListPlot[
  data,
  PlotRange->{{0,21},{0,21}},
  PlotStyle->Directive[Purple,PointSize[Medium]],
  ImageSize->170
]

```

Output

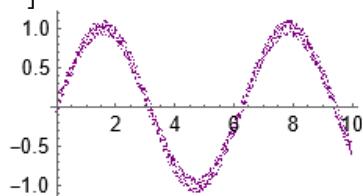
Mathematica code 2.2**RandomReal**

```
Input (* This code generates random data points for a smooth curve by adding random noise to the function Sin[x]. The RandomReal function with the interval {-0.1, 0.1} is used to generate random noise. The data variable is a list of x and y values generated using the Table function with a step size of 10/n. The data points are plotted using the ListPlot function with the PlotRange option set to All: *)

```

```
n=1000;
data=Table[
  {x,Sin[x]+RandomReal[{-0.1,0.1}]},
  {x,0,10,10/n}
];
ListPlot[
  data,
  PlotRange->All,
  PlotStyle->Purple,
  ImageSize->170
]
```

Output

**Mathematica code 2.3****RandomPoint**

```
Input (* Generate a list of points in a unit disk: *)

```

```
pts=RandomPoint[Disk[],500];
Graphics[
  {PointSize[0.02],Purple,Opacity[0.3],Point[pts]},
  ImageSize->150
]
```

Output



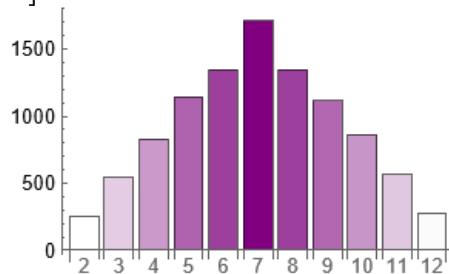
Mathematica code 2.4**RandomChoice**

Input (* This code uses the RandomChoice function to generate 10000 rolls of two six-sided dice, sums the values of the dice, and stores the results in the variable rolls. It then creates a table counts of the number of times each possible sum occurs, from 2 to 12. Finally, it displays the frequency of each sum using a BarChart with labeled bars: *)

```
rolls=Table[
  RandomChoice[Range[6]]+RandomChoice[Range[6]],
  {10000}
];

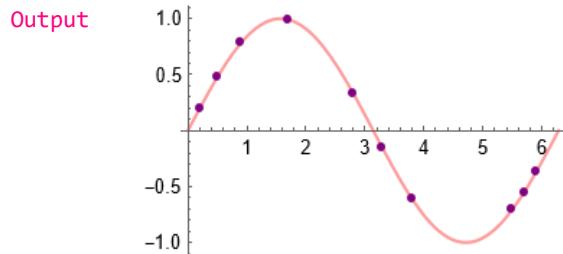
counts=Table[
  Count[rolls,i],
  {i,2,12}
];

BarChart[
  counts,
  ColorFunction->Function[{height},Opacity[height]],
  ChartStyle->Purple,
  ChartLabels->Range[2,12],
  ImageSize->220
]
```

Output**Mathematica code 2.5****RandomSample**

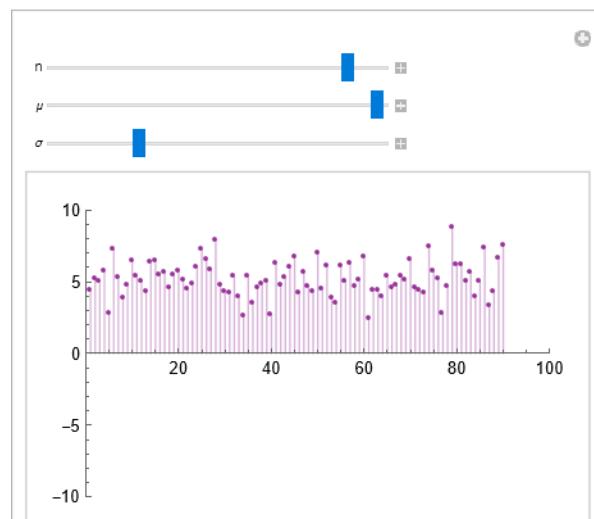
Input (* This code plots the sine function f[x] between 0 and 2 Pi, and then superimposes 10 Purple points on the plot. These points are randomly sampled from a list of equidistant points on the function. The RandomSample function is called with the list of points as the first argument, and 10 as the second argument to select 10 random points from the list: *)

```
f[x_]:=Sin[x];
Plot[
  f[x],
  {x,0,2 Pi},
  PlotStyle->{Red,Opacity[0.4]},
  Epilog->{
    PointSize[0.02],
    Purple,
    Point[
      RandomSample[
        Table[{x,f[x]},{x,0,2 Pi,0.1}],
        10
      ]
    ],
    ImageSize->220
}]
```

**Mathematica code 2.6****RandomVariate**

Input (* The code uses the Table function to generate a list of n random variables drawn from a normal distribution with mean mu and standard deviation sigma using the RandomVariate function. This list is then passed to ListPlot to create the plot. The Manipulate function allows for interactive exploration of the plot by adjusting the values of n, mu, and sigma. This code can be used as a starting point for more advanced statistical simulations and explorations. For example, one could modify the code to simulate data from a different distribution, or to add additional controls for exploring different aspects of the distribution (such as skewness or kurtosis): *)

```
Manipulate[
 ListPlot[
 Table[
 RandomVariate[NormalDistribution[μ, σ]],
 {n}
 ],
 Filling->Axis,
 PlotStyle->{Directive[Purple, Opacity[0.8]]},
 PlotRange->{{0, 100}, {-10, 10}},
 ImageSize->300
 ],
 {{n, 50}, 1, 100, 1},
 {{μ, 0}, -5, 5, 0.1},
 {{σ, 1}, 0.1, 5, 0.1}
 ]
```

Output

Some of the most commonly used Mathematica functions include `Histogram`, and `Histogram3D`. By using these functions, you can gain a deeper understanding of the underlying patterns in your data and make informed decisions based on your findings.

<code>Histogram[{x1,x2,...}]</code>	plots a histogram of the values xi.
<code>Histogram[{x1,x2,...},bspec]</code>	plots a histogram with bin width specification bspec.
<code>Histogram[{x1,x2,...},bspec,hspec]</code>	plots a histogram with bin heights computed according to the specification hspec.
<code>Histogram[{data1,data2,...},...]</code>	plots histograms for multiple datasets datai.
<code>Histogram3D[{{x1,y1},{x2,y2},...}]</code>	plots a 3D histogram of the values {xi,yi}.
<code>Histogram3D[{{x1,y1},{x2,y2},...},bspec]</code>	plots a 3D histogram with bins specified by bspec.
<code>Histogram3D[{{x1,y1},{x2,y2},...},bspec,hspec]</code>	plots a 3D histogram with bin heights computed according to the specification hspec.
<code>Histogram3D[{data1,data2,...}]</code>	plots 3D histograms for multiple datasets datai.

Remarks:

- The following bin specifications `bspec` can be given:

<code>n</code>	use n bins
<code>{dx}</code>	use bins of width dx
<code>{xmin,xmax,dx}</code>	use bins of width dx from xmin to xmax
<code>{b1,b2,...}</code>	use bins [b1,b2],[b2,b3],...
<code>Automatic</code>	determine bin widths automatically
<code>"name"</code>	use a named binning method
<code>{"Log",bspec}</code>	apply binning bspec on log-transformed data
<code>fb</code>	apply fb to get an explicit bin specification {b1,b2,...}

- Possible named binning methods include:

<code>"Sturges"</code>	compute the number of bins based on the length of data
<code>"Scott"</code>	asymptotically minimize the mean square error
<code>"FreedmanDiaconis"</code>	twice the interquartile range divided by the cube root of sample size
<code>"Knuth"</code>	balance likelihood and prior probability of a piecewise uniform model
<code>"Wand"</code>	one-level recursive approximate Wand binning

- Different forms of histogram can be obtained by giving different bin height specifications `hspec` in `Histogram[data,bspec,hspec]`. The following forms can be used:

<code>"Count"</code>	the number of values lying in each bin
<code>"CumulativeCount"</code>	cumulative counts
<code>"SurvivalCount"</code>	survival counts
<code>"Probability"</code>	fraction of values lying in each bin
<code>"Intensity"</code>	count divided by bin width
<code>"PDF"</code>	probability density function
<code>"CDF"</code>	cumulative distribution function
<code>"SF"</code>	survival function
<code>"HF"</code>	hazard function
<code>"CHF"</code>	cumulative hazard function
<code>{"Log",hspec}</code>	log-transformed height specification
<code>fh</code>	heights obtained by applying fh to bins and counts

Mathematica code 2.7

Histogram

```
Input      (* This code generates six histograms of 500 random samples drawn from a normal distribution with mean 0 and standard deviation 1, using different methods to set the vertical scale of the histograms. The different methods used are: "Count", "Probability", "PDF", "CumulativeCount", "CDF", and "SF". This allows for a comparison of different ways of visualizing the same data, which can help in understanding the data and selecting the most appropriate method for a particular analysis: *)
```

**Mathematica code 2.8****Histogram**

Input

```
(* This code generates a histogram of 1000 random samples drawn from a normal distribution with mean 0 and standard deviation 1. The resulting histogram has no outline or borders around the bars and is colored in purple. Without the borders around the bars, the shape of the distribution may be more apparent. This can be especially useful when comparing multiple histograms with different data sets or parameters: *)
```

```
Histogram[
  RandomVariate[
    NormalDistribution[0,1],
    1000
  ],
  ChartBaseStyle->EdgeForm[None],
  ChartStyle->Purple,
  ImageSize->170
]
```

Output

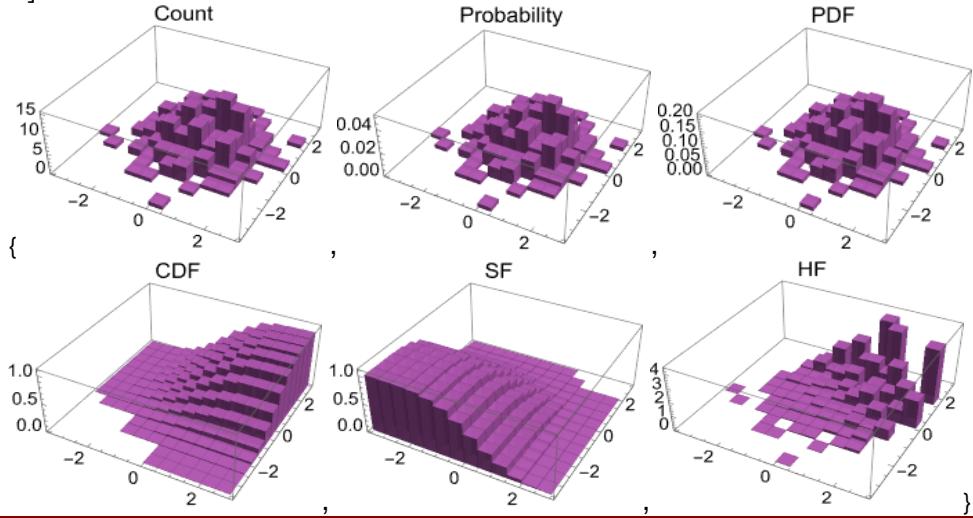
Mathematica code 2.9**Histogram3D**

Input

```
(* This code generates a sequence of 3D histograms for a random sample of size 300 from a bivariate normal distribution with mean 0 and standard deviation 1 in each dimension. The use of different height functions allows for a more comprehensive understanding of the sample's distribution and provides a range of useful visualization tools for data analysis: *)
sampledata=RandomVariate[
  NormalDistribution[0,1],
  {300,2}
];

Table[
  Histogram3D[
    sampledata,
    Automatic,
    height,
    PlotLabel->height,
    ColorFunction->Function[{height},Opacity[0.9]],
    ChartStyle->RGBColor[0.6,0.30,0.60],
    ImageSize->170
  ],
  {height,{"Count","Probability","PDF","CDF","SF","HF"}}
]
```

Output



The Wolfram Language's descriptive statistics functions operate both on explicit data and on symbolic representations of statistical distributions, making it a valuable tool for data analysis and exploratory data science tasks.

Mean[list]	gives the statistical mean of the elements in list.
Mean[dist]	gives the mean of the distribution dist.
Median[list]	gives the median of the elements in list.
Median[dist]	gives the median of the distribution dist.
Commonest[list]	gives a list of the elements that are the most common in list.
Commonest[list,n]	gives a list of the n most common elements in list.
TrimmedMean[list,f]	gives the mean of the elements in list after dropping a fraction f of the smallest and largest elements.
TrimmedMean[dist,...]	gives the trimmed mean of a univariate distribution dist.

<code>WinsorizedMean[list,f]</code>	gives the mean of the elements in list after replacing the fraction f of the smallest and largest elements by the remaining extreme values.
<code>WinsorizedMean[dist,...]</code>	gives the winsorized mean of a univariate distribution dist.
<code>HarmonicMean[list]</code>	gives the harmonic mean of the values in list.
<code>GeometricMean[list]</code>	gives the geometric mean of the values in list.
<code>RootMeanSquare[list]</code>	gives the root mean square of values in list.
<code>RootMeanSquare[dist]</code>	gives the root mean square of the distribution dist.
<code>Quartiles[list]</code>	gives a list of the 1/4, 1/2 and 3/4 quantiles of the elements in list.
<code>Quartiles[dist]</code>	gives a list of the 1/4, 1/2 and 3/4 quantiles of the distribution dist.

Mathematica code 2.10**Mean**

```
(* The code analyzes a dataset of heights by calculating the mean and generating two plots. The first plot displays the data set, while the second plot displays the data set along with horizontal line for the mean: *)
```

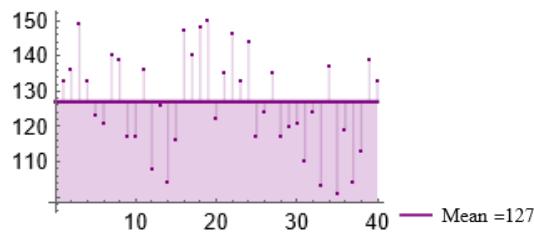
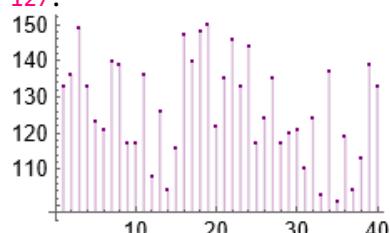
```
h={133,136,149,133,123,121,140,139,117,117,136,108,126,104,116,147,140,148,150,122,135,146,133,144,117,124,135,117,120,121,110,124,103,137,101,119,104,113,139,133};
```

```
m=N[Mean[h]]  
n=Length[h];
```

```
ListPlot[  
h,  
Filling->Axis,  
PlotStyle->Purple,  
ImageSize->170  
]
```

```
ListPlot[  
{h,{{0,m},{n,m}}},  
Joined->{False,True},  
Filling->{1->m,2->Axis},  
PlotStyle->Purple,  
ImageSize->170,  
PlotLegends->{None,"Mean =127"}  
]
```

Output



Mathematica code 2.11**Commonest**

Input (* The code reads in a dataset h of heights and calculates Commonest elements. It then produces two plots: a scatter plot of the dataset h, and a plot that shows both h and three horizontal lines at the commonest elements and mean. These plots are used for visualizing the distribution of the dataset and highlighting the location of the commonest elements and mean: *)

```
h={133,136,149,133,123,121,140,139,117,117,136,108,126,104,116,147,140,148,150,122,135,146,133,144,117,124,135,117,120,121,110,124,103,137,101,119,104,113,139,133};
```

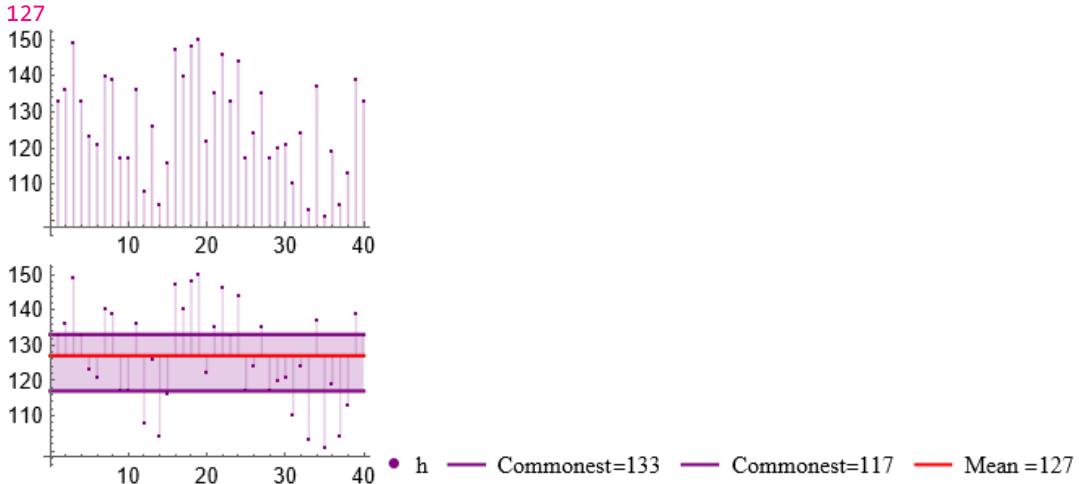
```
Tally[h]
c=N[Commonest[h]]
m=Mean[h]
n=Length[h];
```

```
ListPlot[
h,
Filling->Axis,
PlotStyle->Purple,
ImageSize->170
]
```

```
ListPlot[
{h,{{0,c[[1]]},{n,c[[1]]}}},{{0,c[[2]]},{n,c[[2]]}},{{0,m},{n,m}}},
Joined->{False,True,True,True},
Filling->{1->m,2->{3}},
PlotStyle->{Purple,Purple,Purple,Red},
ImageSize->170,
PlotLegends->{"h","Commonest=133","Commonest=117","Mean =127"}
]
```

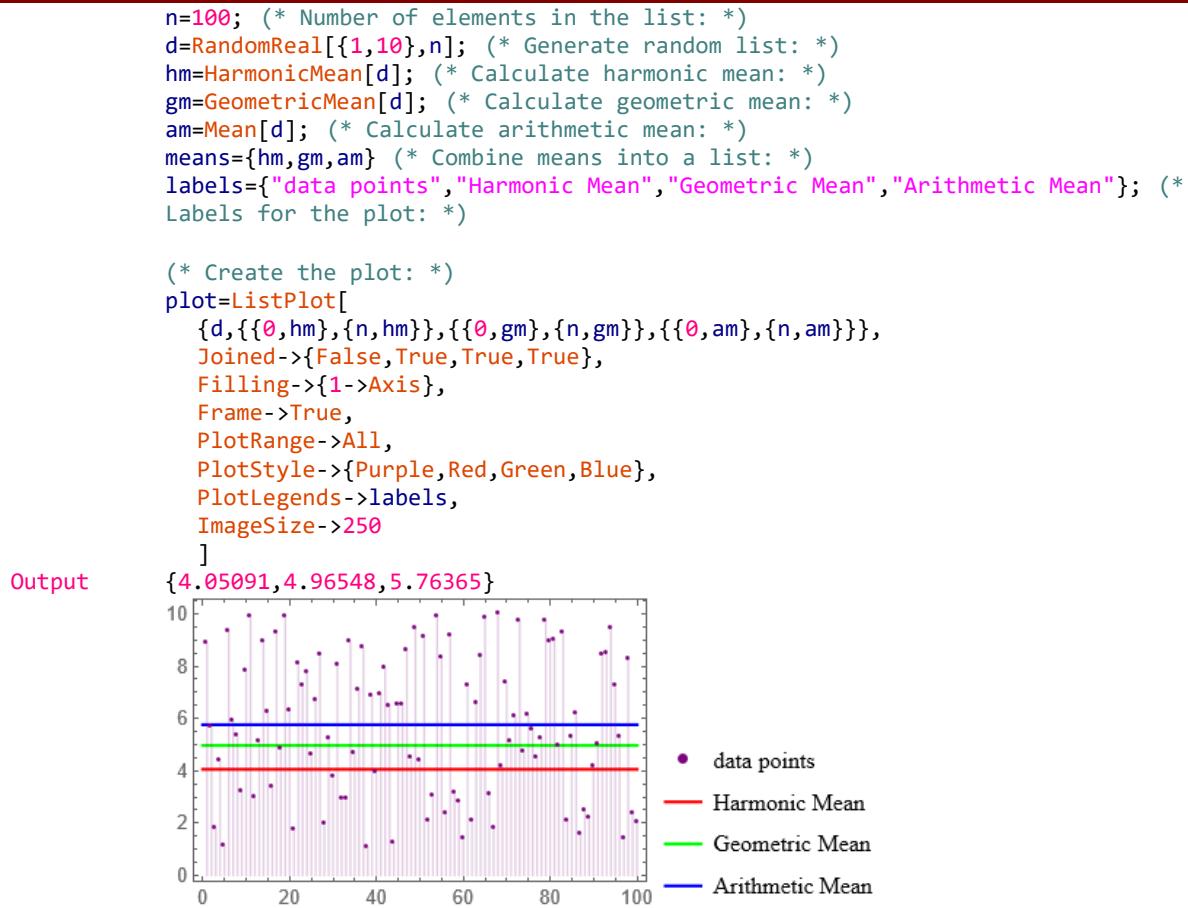
Output

```
{133,4},{136,2},{149,1},{123,1},{121,2},{140,2},{139,2},{117,4},{108,1},{126,1},{104,2},{116,1},{147,1},{148,1},{150,1},{122,1},{135,2},{146,1},{144,1},{124,2},{120,1},{110,1},{103,1},{137,1},{101,1},{119,1},{113,1}}
{133.,117.}
```

**Mathematica code 2.12****HarmonicMean, GeometricMean and Mean**

Input (* The code will generate a list of random numbers, calculate their Harmonic Mean, Geometric Mean, and Arithmetic Mean, and then plot the means against the number of elements in the list. For positive data, we can see that Harmonic Mean <= Geometric Mean <= Arithmetic Mean: *)

```
SeedRandom[1234];
```

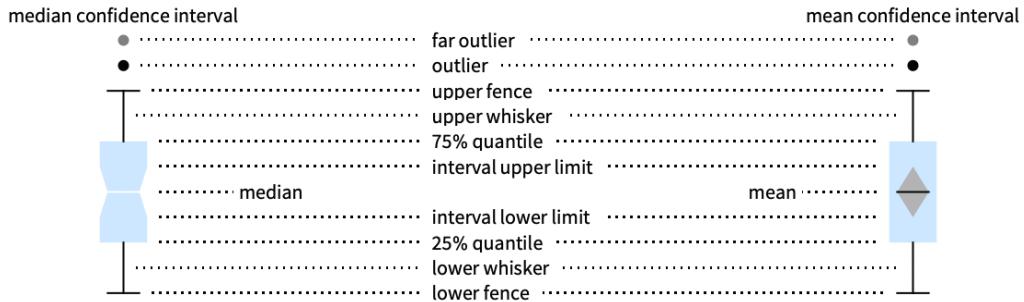


The Mathematica function `BoxWhiskerChart` is a powerful tool for visualizing and analyzing data distributions. Here are some features of this function:

- **Clear representation of data:** `BoxWhiskerChart` provides a clear and concise representation of the distribution of a dataset. It displays important statistical measures such as the median, quartiles, and outliers, making it easy to understand the central tendency and spread of the data.
- **Customizable appearance:** The function offers a wide range of options to customize the appearance of the box-and-whisker plot. You can adjust the colors, styles, and sizes of the boxes, whiskers, outliers, and other elements to suit your preferences or match your presentation or publication style.
- **Comparative analysis:** `BoxWhiskerChart` allows for easy comparison of multiple datasets. You can plot several box-and-whisker diagrams side by side or in a stacked manner, making it straightforward to identify differences or similarities in distributions.
- **Interaction and exploration:** The resulting chart is interactive, meaning you can hover over different elements to obtain more detailed information about specific data points or summary statistics. This interactivity enhances the exploratory data analysis process and allows for a deeper understanding of the underlying distribution.

<code>BoxWhiskerChart[{x1,x2,...}]</code>	makes a box-and-whisker chart for the values x_i .
<code>BoxWhiskerChart[{x1,x2,...},bwspec]</code>	makes a chart with box-and-whisker symbol specification $bwspec$.
<code>BoxWhiskerChart[{data1,data2,...},...]</code>	makes a chart with box-and-whisker symbol for each $data_i$.
<code>BoxWhiskerChart[{{data1,data2,...},...},...]</code>	makes a box-and-whisker chart from multiple groups of datasets $\{data_1, data_2, \dots\}$.

`BoxWhiskerChart` draws a box-and-whisker summary of the distribution of values in each `datai`. See the following figure



The following box-and-whisker specifications `bwspec` can be given:

"Notched"	median confidence interval notch
"Outliers"	outlier markers
"Median"	median marker
"Basic"	box-and-whisker only
"Mean"	mean marker
"Diamond"	mean confidence interval diamond
<code>{elem1, val11, ...}, ...</code>	box-and-whisker element specification
<code>{"name", {elem1, val11, ...}, ...}</code>	named bwspec with element modifica

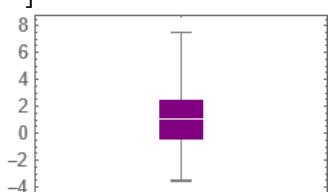
Mathematica code 2.13

BoxWhiskerChart

Input (* The code uses the `RandomVariate` function to generate a data vector with 200 values. It samples from a normal distribution with a mean of 1 and a standard deviation of 2. then, the code generates a basic box-and-whisker chart using randomly generated data: *)

```
BoxWhiskerChart[
  RandomVariate[
    NormalDistribution[1, 2],
    200
  ],
  ChartStyle -> Purple,
  ImageSize -> 170
]
```

Output

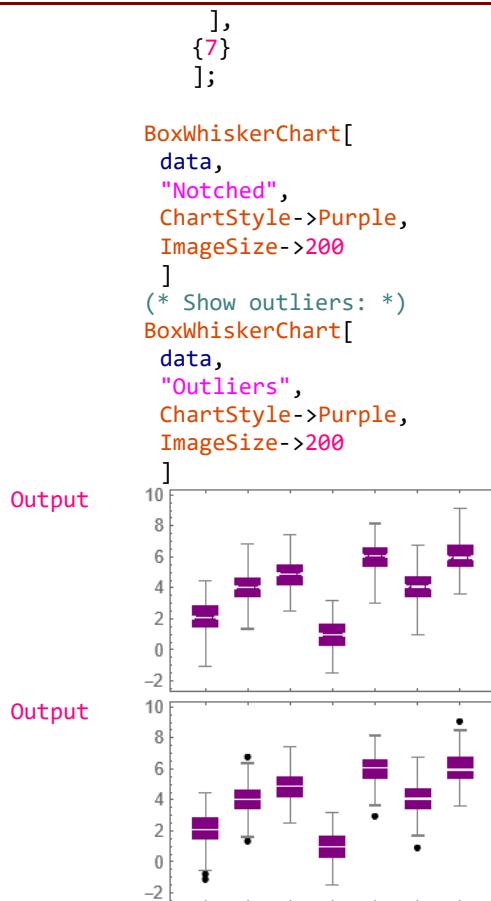


Mathematica code 2.14

BoxWhiskerChart

Input (* The code demonstrates customization options for a box-and-whisker chart. It generates a list of seven datasets, each containing 200 data points randomly sampled from normal distributions. The mean of each dataset is randomly chosen from the integers 0 to 6, while the standard deviation is fixed at 1. The first `BoxWhiskerChart` call creates a notched box-and-whisker chart, displaying confidence intervals around the medians of each dataset. The second `BoxWhiskerChart` call explicitly shows outliers, highlighting extreme or unusual data points: *)

```
data = Table[
  RandomVariate[
    NormalDistribution[RandomInteger[6], 1],
    200
  ],
  {7}
]
```

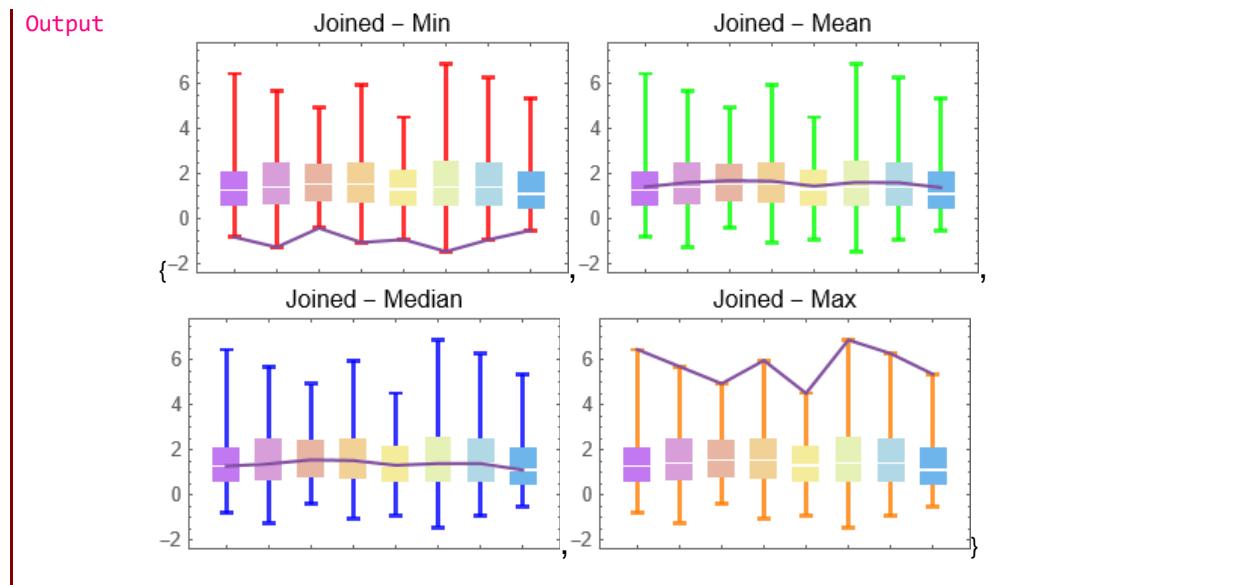
**Mathematica code 2.15****BoxWhiskerChart**

```

Input (* The code generates a series of BoxWhiskerCharts to visualize the distribution of
a dataset called 'data' using the SkewNormalDistribution. The dataset has dimensions
8 rows by 200 columns. The code uses the BoxWhiskerChart function to create a box
and whisker plot for each column in the 'data' dataset. Each chart represents
statistical measures such as the minimum, mean, median, and maximum values with
different line styles and colors: *)
data=RandomVariate[
  SkewNormalDistribution[0,2,4],
 {8,200}
];

Table[
  BoxWhiskerChart[
    data,
    {
      {"Whiskers",Directive[Thick,s[[2]],Opacity[0.8]]},
      {"Fences",Directive[Thick,s[[2]],Opacity[0.8]]}
    },
    Joined->s[[1]],
    PlotLabel->Style[Row[{"Joined - ",s[[1]]}]],
    ChartStyle->"Pastel",
    ImageSize->200
  ],
{s,{{"Min",Red}, {"Mean",Green}, {"Median",Blue}, {"Max",Orange}}}
]

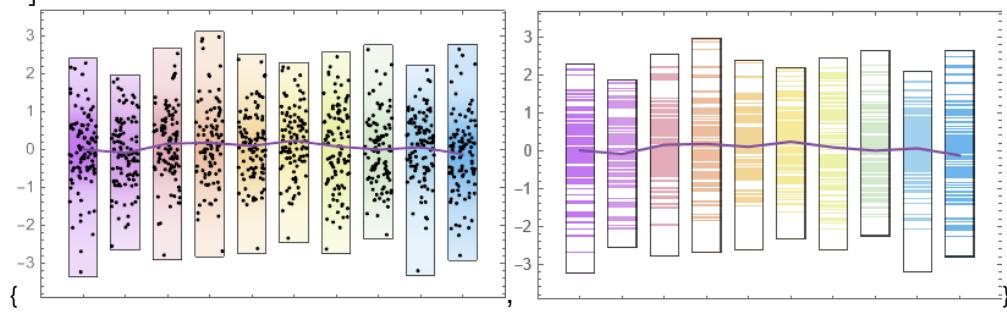
```

**Mathematica code 2.16****BoxWhiskerChart**

Input (* The code generates multiple density plots using the DistributionChart function. It generates a 2D array of random data called data. It consists of 10 rows and 100 columns, where each element is sampled from a standard normal distribution. The code then uses a combination of Table and DistributionChart to generate density plots for each element in the list of chart element functions. The Joined option is set to "Mean" to connect the mean values of the distributions with a line. The ChartElementFunction option is used to specify different chart element functions, such as "PointDensity" and "LineDensity". *)

```
data=RandomVariate[NormalDistribution[],{10,100}];
```

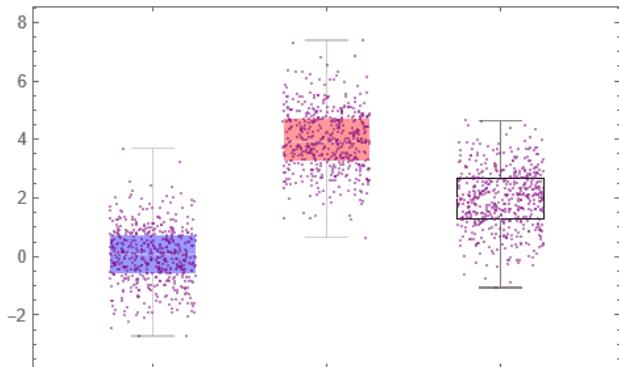
```
Table[
  DistributionChart[
    data,
    Joined->"Mean",
    ChartElementFunction->s,
    ChartStyle->"Pastel",
    ImageSize->300
  ],
  {s,{ "PointDensity", "LineDensity"}}
]
```

Output

Mathematica code 2.17**BoxWhiskerChart**

Input

```
(* The code generates three sets of random data (data1, data2, data3) by sampling
from a normal distribution with different means ( $\mu$ ) using RandomVariate. Each set
consists of 500 data points. The BoxWhiskerChart function is used to create a box-
and-whisker plot. The Epilog option is used to add additional elements to the chart.
In this code, points are overlaid on the chart to represent the individual data points
of each dataset. The points are created using Map and Transpose. The random
displacement in the x-axis is in a range of -0.25 to 0.25: *)
{data1,data2,data3}=Table[
  RandomVariate[NormalDistribution[ $\mu$ ,1],500],
  { $\mu$ ,{0,4,2}}
];
BoxWhiskerChart[
  {data1,data2,data3},
  ChartStyle->
  {Directive[Blue,Opacity[0.4]],Directive[Red,Opacity[0.4]],Directive[White,EdgeForm
[Thickness[0.001]]]},
  ImageSize->350,
  Epilog->{
    Directive[Purple,Opacity[0.5]],
    PointSize[0.0045],
    Map[
      Point,{
        Transpose[{ConstantArray[1,Length[data1]]+RandomReal[{-0.25,0.25}],Length[data1],data1}],
        Transpose[{ConstantArray[2,Length[data2]]+RandomReal[{-0.25,0.25}],Length[data2],data2}],
        Transpose[{ConstantArray[3,Length[data3]]+RandomReal[{-0.25,0.25}],Length[data3],data3}]}
      ]
    ]
]
```

Output

Dispersion statistics summarize the scatter or spread of the data. Most of these functions describe deviation from a particular location. For instance, variance is a measure of deviation from the mean. Mathematica provides a set of functions and tools for calculating, visualizing, and analyzing dispersion statistics, allowing users to gain deeper insights into the variability and distribution of their data. Let us go through them in detail.

<code>InterquartileRange[list]</code>	gives the difference between the upper and lower quartiles for the elements in list.
<code>InterquartileRange[dist]</code>	gives the difference between the upper and lower quartiles for the distribution dist.
<code>QuartileDeviation[list]</code>	gives the quartile deviation or semi-interquartile range of the elements in list.
<code>QuartileDeviation[dist]</code>	gives the quartile deviation or semi-interquartile range of the distribution dist.
<code>MeanDeviation[list]</code>	gives the mean absolute deviation from the mean of the elements in list.

<code>StandardDeviation[list]</code>	gives the sample standard deviation of the elements in list.
<code>StandardDeviation[dist]</code>	gives the standard deviation of the distribution dist.
<code>Variance[list]</code>	gives the sample variance of the elements in list.
<code>Variance[dist]</code>	gives the variance of the distribution dist.
<code>TrimmedVariance[list,f]</code>	gives the variance of the elements in list after dropping a fraction f of the smallest and largest elements.
<code>TrimmedVariance[dist,...]</code>	gives the trimmed variance of a univariate distribution dist.
<code>WinsorizedVariance[list,f]</code>	gives the variance of the elements in list after replacing the fraction f of the smallest and largest elements by the remaining extreme values.
<code>WinsorizedVariance[dist,...]</code>	gives the winsorized variance of a univariate distribution dist.

Mathematica code 2.18**QuartileDeviation and StandardDeviation**

```

Input      (* This code generates a 2D dataset from a standard normal distribution and
           calculates various statistical measures such as quartile deviation, standard
           deviation, and mean for each component of the dataset. It then displays the results
           and creates a scatter plot with highlighted points representing the statistical
           measures: *)
data=RandomVariate[
  NormalDistribution[0,1],
  {2000,2}
];
qdx=QuartileDeviation[data[[All,1]]];
qdy=QuartileDeviation[data[[All,2]]];

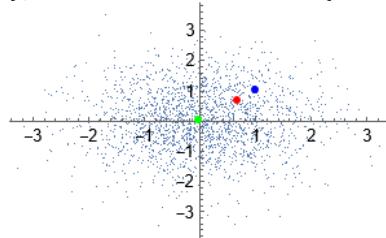
sdx=StandardDeviation[data[[All,1]]];
sdy=StandardDeviation[data[[All,2]]];

mx=Mean[data[[All,1]]];
my=Mean[data[[All,2]]];
Print[
 "Mean= {" ,mx,",",my,"}, ",
 "QuartileDeviation= {" ,qdx,",",qdy,"}, ",
 "StandardDeviation= {" , sdx,",",sdy,"} "
]

]
ListPlot[
 data,
 Epilog->{
 Red, PointSize[0.02],Point[{qdx,qdy}],
 Blue, PointSize[0.02],Point[{sdx,sdy}],
 Green, PointSize[0.02],Point[{mx,my}]
 },
 ImageSize->220
]

```

Output Mean= { -0.0169259 , -0.00549935 }, QuartileDeviation= { 0.676502 , 0.655909 }, StandardDeviation= { 0.998782 , 1.0005 }



Mathematica code 2.19**StandardDeviation**

```

Input   (* The code generates a list of 1000 random numbers from a standard normal
        distribution. It then calculates the standard deviation of the data and creates a
        histogram plot with the probability density function (PDF). The plot includes red
        points representing the mean and ± standard deviation of the data: *)
        
data=RandomVariate[
  NormalDistribution[0,1],
  1000
];

StandardDeviation[data]

Histogram[
  data,
  Automatic,
  "PDF",
  Epilog->{
    Red,
    PointSize[0.03],
    Point[{{StandardDeviation[data],0},{Mean[data],0},{-
StandardDeviation[data],0}}]
  },
  ColorFunction->Function[{height},Opacity[height]],
  ChartStyle->Purple,
  ImageSize->200
]

```

Output 0.96111

Output

Mathematica code 2.20**Variance**

```

Input   (* Variance of a list of numbers: *)
Variance[{3.2,1.21,3.4,2,4.66,1.5,5.61,7.22}]

Output 4.45003

Input   (* Variance of elements in each column: *)
Variance[{{4,1},{5,2,7},{5,3,8},{5,4,9}}]
Variance[{4,5,2,5,3,5,4}]
Variance[{1,7,8,9}]

Output {0.429167,155/12}
Output 0.429167
Output 155/12

Input   (* Find the variance of WeightedData: *)
data={8,3,5,4,9,0,4,2,2,3};
w={0.15,0.09,0.12,0.10,0.16,0.,0.11,0.08,0.08,0.09};
Variance[WeightedData[data,w]]

Output 7.26594

Input   (* This code creates a Manipulate function with slider and a dropdown menu. The n
        slider allows the user to adjust the sample size, while the dist dropdown menu allows

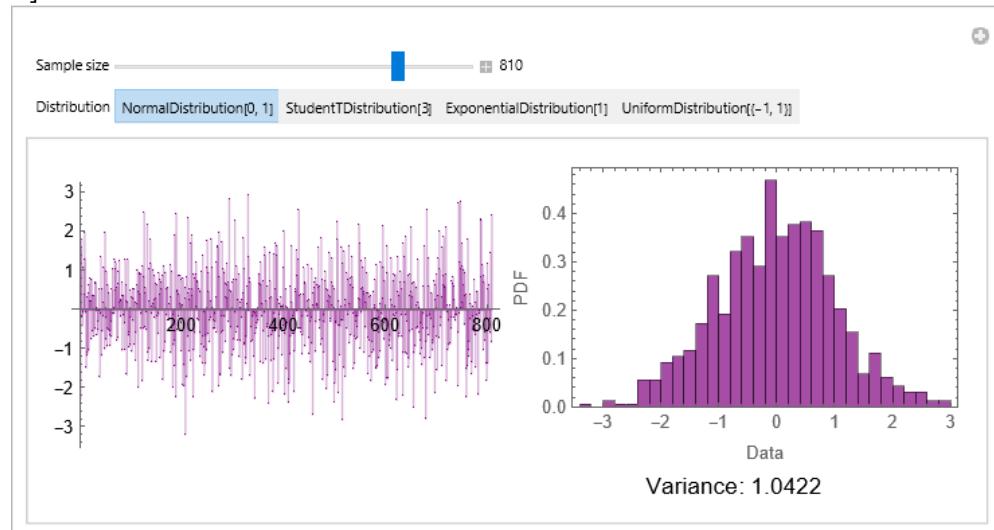
```

the user to choose the distribution that the sample is drawn from. The code then uses RandomVariate to generate a sample of size n from the selected distribution, calculates the variance using Variance, and displays both a ListPlot and a Histogram of the sample data along with the calculated variance. *)

```
Manipulate[
Module[
{data,var},
data=RandomVariate[dist,n];
var=Variance[data];

Grid[
{
{
ListPlot[
data,
PlotRange->All,
ImageSize->250,
Filling->Axis,
PlotStyle->Purple
],
Histogram[
data,
"FreedmanDiaconis",
"PDF",
Frame->True,
FrameLabel->{"Data", "PDF"},
ImageSize->250,
ColorFunction->Function[Opacity[0.7]],
ChartStyle->Purple
]
},
{Null,Text["Variance: "<>ToString[var]]}
}
],
{{n,300,"Sample size"},10,1000,10,Appearance-
>"Labeled"},{{dist,NormalDistribution[0,1],"Distribution"},{NormalDistribution[0,1],
StudentTDistribution[3],ExponentialDistribution[1],UniformDistribution[{-1,1}]}}},
Alignment->Center
]
]
```

Output



A variety of moments are used to summarize a distribution or data. Mean is used to indicate a center location, variance and standard deviation are used to indicate dispersion, etc. The Wolfram Language fully supports moments of any order, univariate or multivariate, for symbolic distributions and data. Moreover, you can get some information about the shape of a distribution using shape statistics functions. Skewness describes the amount of asymmetry. Kurtosis measures the concentration of data around the peak and in the tails versus the concentration in the flanks. let us start by considering moment functions.

<code>Moment[list,r]</code>	gives the r^(th) sample moment of the elements in list.
<code>Moment[dist,r]</code>	gives the r^(th) moment of the distribution dist.
<code>CentralMoment[list,r]</code>	gives the r^(th) central moment of the elements in list with respect to their mean.
<code>CentralMoment[dist,r]</code>	gives the r^(th) central moment of the distribution dist.
<code>Skewness[list]</code>	gives the coefficient of skewness for the elements in list.
<code>Skewness[dist]</code>	gives the coefficient of skewness for the distribution dist.
<code>QuartileSkewness[list]</code>	gives the coefficient of quartile skewness for the elements in list.
<code>QuartileSkewness[dist]</code>	gives the coefficient of quartile skewness for the distribution dist.
<code>Kurtosis[list]</code>	gives the coefficient of kurtosis for the elements in list.
<code>Kurtosis[dist]</code>	gives the coefficient of kurtosis for the distribution dist.

Mathematica code 2.21**Moment and Kurtosis**

```

Input (* First Moment: The first moment of a set of data is the mean: *)
      list={1,2,3,4,5};
      moment1=Mean[list]
      Moment[list,1]
Output 3
Output 3

Input (* Use symbolic data: *)
      Moment[{x,y,z},2]
Output 1/3 (x^2+y^2+z^2)

Input (* Kurtosis for a list of values: *)
      Kurtosis[{a,b,c}]
Output (3 ((a+1/3 (-a-b-c))^4+(b+1/3 (-a-b-c))^4+(1/3 (-a-b-c)+c)^4))/((a+1/3 (-a-b-c))^2+(b+1/3 (-a-b-c))^2+(1/3 (-a-b-c)+c)^2)^2

```

Mathematica code 2.22**CentralMoment**

```

Input (* In this code, the Manipulate function creates an interactive interface that allows
you to explore the effects of changing the distributions and the order of the central
moment. The code includes a histogram plot of the sample data using the Histogram
function. The plot is displayed as a probability density function, and the calculated
central moment is shown as a text label. The Manipulate parameters include the choice
of distribution, sample size, and order of the central moment. The user can select
different distributions, adjust the sample size using a slider, and modify the order
of the central moment: *)

Manipulate[
Module[
{data,mean,centralMoment},
(*Generate sample data from a distribution*)
data=RandomVariate[Distribution,sampleSize];
(*Calculate central moment*)
centralMoment=CentralMoment[data,n];
(*Plotting the histogram*)

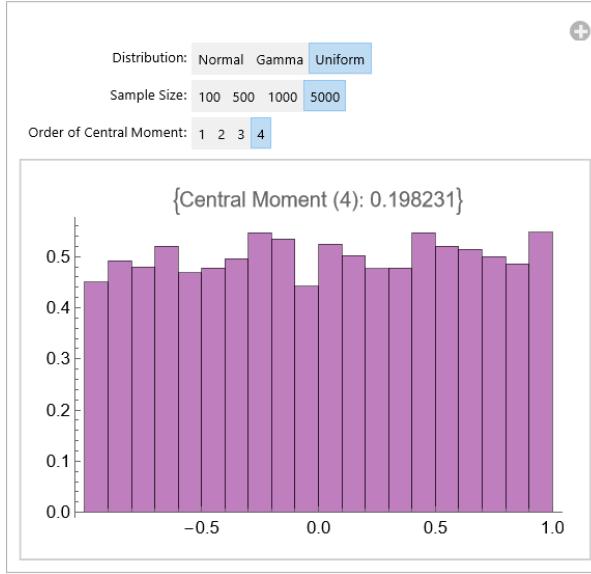
```

```

Histogram[
  data,
  Automatic,
  "PDF",
  PlotRange->All,
  PlotLabel->{Text["Central Moment ("<>ToString[n]<>"):"]
  "<>ToString[centralMoment]]},
  ImageSize->300,
  ColorFunction->Function[Opacity[0.5]],
  ChartStyle->Purple
  ]
],
(*Manipulate parameters*)
{{Distribution,NormalDistribution[0,1],"Distribution:"},{NormalDistribution[0,1]-
>"Normal",GammaDistribution[2,1]->"Gamma",UniformDistribution[{-1,1}]->"Uniform"},{{sampleSize,1000,"Sample Size:"},{100,500,1000,5000}},{{n,2,"Order of Central Moment:"},{1,2,3,4}}}
]

```

Output

**Mathematica code 2.23****Skewness**

Input (* The code creates three skew normal distributions with different shape parameter α and visualizes their probability density functions (PDFs) on a plot. It computes the skewness values for each distribution and displays them in the plot legend. The code demonstrates how the shape parameter α affects the shape of the distributions: *)

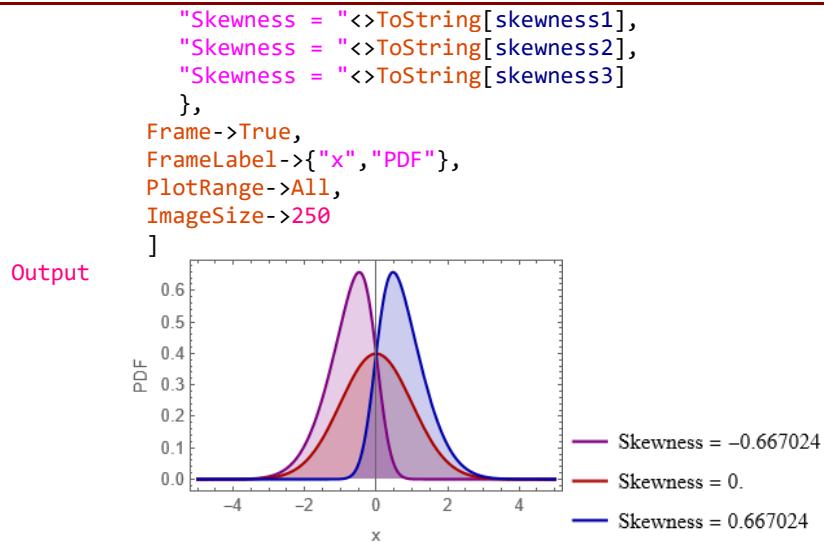
```

dist1=SkewNormalDistribution[0,1,-3]; (* alpha=-3*)
dist2=SkewNormalDistribution[0,1,0]; (* alpha=0*)
dist3=SkewNormalDistribution[0,1,3]; (* alpha=3*)

skewness1=N[Skewness[dist1]];
skewness2=N[Skewness[dist2]];
skewness3=N[Skewness[dist3]];

Plot[
 {PDF[dist1,x],PDF[dist2,x],PDF[dist3,x]},
 {x,-5,5},
 PlotStyle->{Purple,Darker[Red],Darker[Blue]},
 Filling->Axis,
 PlotLegends->

```



Unit 2.2

Probability Distributions

<code>PDF[dist,x]</code>	gives the probability density function for the distribution dist evaluated at x.
<code>CDF[dist,x]</code>	gives the cumulative distribution function for the distribution dist evaluated at x.
<code>MomentGeneratingFunction[dist,t]</code>	gives the moment-generating function for the distribution dist as a function of the variable t.

Mathematica code 2.24 PDF, CDF and MomentGeneratingFunction

```
Input (* The PDF and CDF of a univariate continuous distribution: *)

$$\text{PDF}[\text{NormalDistribution}[\mu, \sigma], x]$$


$$\text{CDF}[\text{NormalDistribution}[\mu, \sigma], x]$$


$$\text{MomentGeneratingFunction}[\text{NormalDistribution}[\mu, \sigma], x]$$


$$\text{Mean}[\text{NormalDistribution}[\mu, \sigma]]$$


$$\text{Variance}[\text{NormalDistribution}[\mu, \sigma]]$$

Output 
$$e^{\frac{(x-\mu)^2}{2\sigma^2}}$$


$$\frac{1}{\sqrt{2\pi}\sigma}$$

Output 
$$\frac{1}{2} \text{Erfc}\left[\frac{-x+\mu}{\sqrt{2}\sigma}\right]$$

Output 
$$e^{x\mu + \frac{x^2\sigma^2}{2}}$$

Output 
$$\mu$$

Output 
$$\sigma^2$$

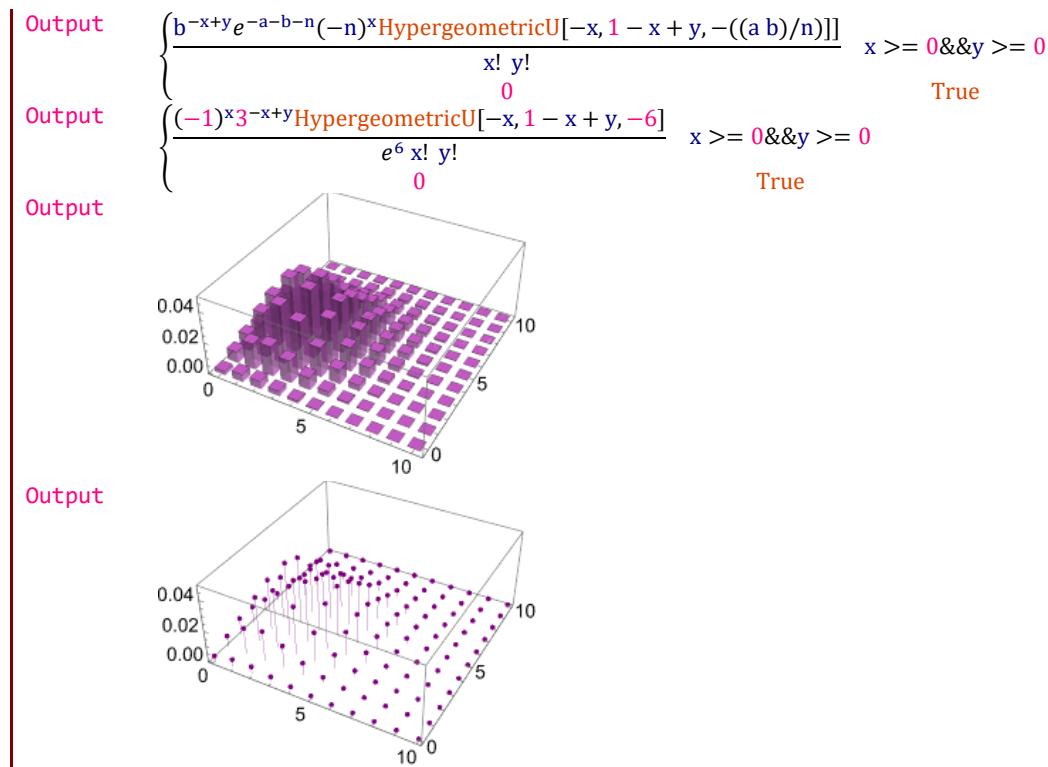
```

Mathematica code 2.25 PDF

```
Input (* The resulting 3D plot visualizes the PMF of the Multivariate Poisson distribution
with specific parameter values n=1, a=2, and b=3. The plot allows you to observe the
PMF values for different combinations of x and y within the specified ranges. Each
cell in the plot represents the PMF value for a specific x and y combination: *)
pdf1=PDF[MultivariatePoissonDistribution[n,{a,b}],{x,y}]
pdf=PDF[MultivariatePoissonDistribution[1,{2,3}],{x,y}]

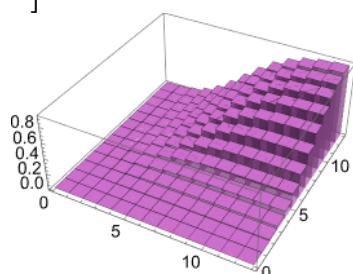
DiscretePlot3D[
 pdf,
 {x,0,10},
 {y,0,10},
 ExtentSize->0.6,
 PlotStyle->Purple,
 ImageSize->200
]

DiscretePlot3D[
 pdf,
 {x,0,10},
 {y,0,10},
 PlotRange->All,
 PlotStyle->Purple,
 ImageSize->200
]
```

**Mathematica code 2.26****CDF**

```
Input (* The CDF for a multivariate Poisson distribution:*)
DiscretePlot3D[
  CDF[MultivariatePoissonDistribution[5, {2, 3}], {x, y}],
  {x, 0, 12},
  {y, 0, 12},
  ExtentSize -> Right,
  PlotStyle -> Lighter[Purple, 0.1],
  ImageSize -> 200
]
```

Output

**BinomialDistribution[n,p]**

represents a binomial distribution with n trials and success probability p.

DiscreteUniformDistribution[{imin,imax}]

represents a discrete uniform distribution over the integers from imin to imax.

DiscreteUniformDistribution[{{imin,imax},{jmin,jmax},...}]

represents a multivariate discrete uniform distribution over integers within the box {{imin,imax},{jmin,jmax},...}.

NormalDistribution[μ,σ]

represents a normal (Gaussian) distribution with mean μ and standard deviation σ.

NormalDistribution[]

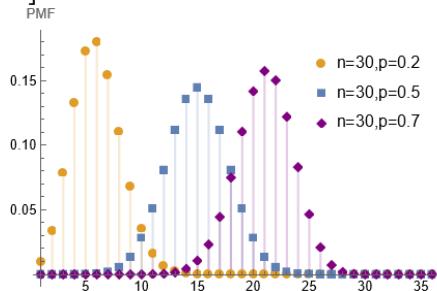
represents a normal distribution with zero mean and unit standard deviation.

<code>UniformDistribution[{min,max}]</code>	represents a continuous uniform statistical distribution giving values between min and max.
<code>UniformDistribution[]</code>	represents a uniform distribution giving values between 0 and 1.
<code>EstimatedDistribution[data,dist]</code>	estimates the parametric distribution dist from data.

Mathematica code 2.27**BinomialDistribution**

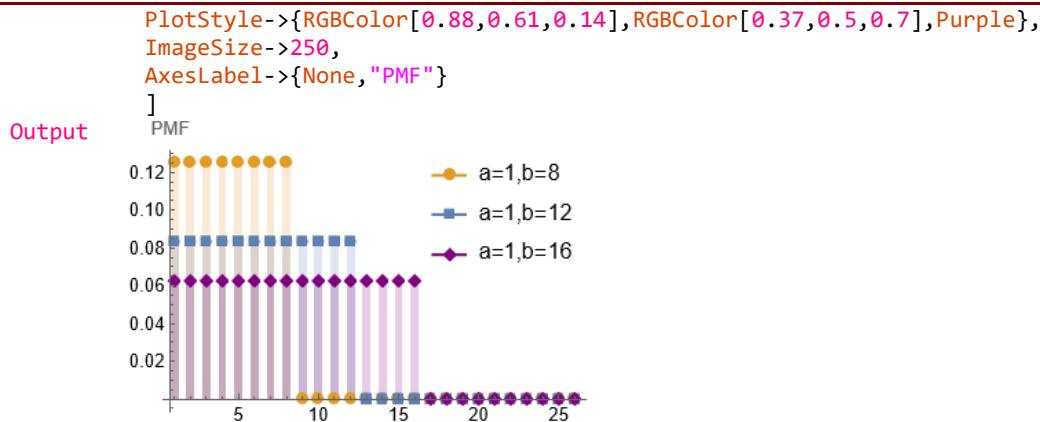
Input (* The code generates a discrete plot of the PMF for a binomial distribution with parameters n=30 and three different values of p (0.2, 0.5, and 0.7). The plot shows the values of the PMF for all possible values of k between 0 and 37: *)

```
DiscretePlot[
 Evaluate[
 Table[
 PDF[
 BinomialDistribution[30,p],k
 ],
 {p,{0.2,0.5,0.7}}
 ]
 ],
 {k,37},
 PlotRange->All,
 PlotMarkers->Automatic,
 PlotLegends->Placed[{"n=30,p=0.2","n=30,p=0.5","n=30,p=0.7"},{0.8,0.75}],
 PlotStyle->{RGBColor[0.88,0.61,0.14],RGBColor[0.37,0.5,0.7],Purple},
 ImageSize->320,
 AxesLabel->{None,"PMF"}
 ]
```

Output**Mathematica code 2.28****DiscreteUniformDistribution**

Input (* The code generates a discrete plot of the PMF for a discrete uniform distribution with parameters a=1 and three different values of b (8, 12, and 16). The plot shows the values of the PMF for all possible values of j between 0 and 26: *)

```
DiscretePlot[
 Evaluate[
 Table[
 PDF[
 DiscreteUniformDistribution[{1,b}],j
 ],
 {b,{8,12,16}}
 ]
 ],
 {j,26},
 ExtentSize->1/2,
 PlotRange->All,
 PlotMarkers->Automatic,
 PlotLegends->Placed[{"a=1,b=8","a=1,b=12","a=1,b=16"},{0.8,0.75}]]
```

**Mathematica code 2.29****UniformDistribution**

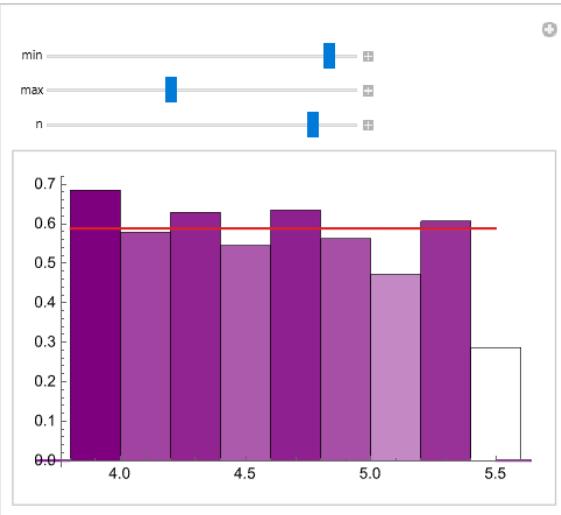
Input (* The code creates a dynamic histogram of data and a plot of the PDF generated from Uniform distribution using the Manipulate function. The Manipulate function creates interactive controls for the user to adjust the values of min and max, which are the parameters of the Uniform distribution and the sample size: *)

```

Manipulate[
Module[
{
  data=RandomVariate[
    UniformDistribution[{min,max}],n
  ]
},
Show[
Histogram[
  data,
  Automatic,
  "PDF",
  ColorFunction->Function[{height},Opacity[height]],
  ImageSize->320,
  ChartStyle->Purple
],
Plot[
  PDF[
    UniformDistribution[{min,max}],x
  ],
{x,0,7},
ColorFunction->"Rainbow"
]
],
{{min,1,"min"},1,4,0.1},
{{max,4.5,"max"},4.5,7,0.1},
{{n,300,"n"},100,1000,10}
]
]

```

Output

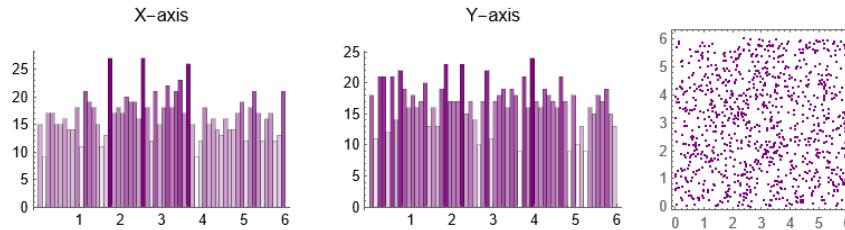
**Mathematica code 2.30****UniformDistribution**

```

Input (* The code generates a 2D dataset with 1000 random points that follow Uniform
       distribution with min=0 and max=6. The dataset is then used to create a row of three
       plots. The first plot is a histogram of the X-axis values of the dataset. The second
       plot is a histogram of the Y-axis values of the dataset. It is similar to the first
       plot, but shows the distribution of the Y-axis values instead. The third plot is a
       scatter plot of the dataset, with the X-axis values on the horizontal axis and the
       Y-axis values on the vertical axis. Each point in the plot represents a pair of X
       and Y values from the dataset: *)

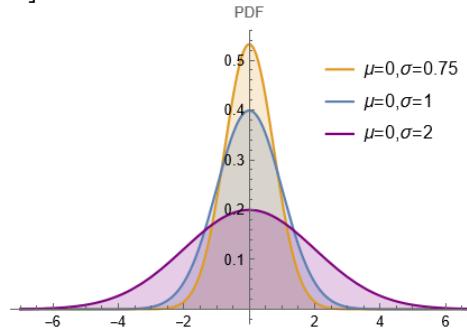
data=RandomVariate[
  UniformDistribution[{0,6}],
  {1000,2}
];
GraphicsRow[
{
  Histogram[
    data[[All,1]],
    {0.1},
    PlotLabel->"X-axis",
    ColorFunction->Function[{height},Opacity[height]],
    ChartStyle->Purple
  ],
  Histogram[
    data[[All,2]],
    {0.1},
    PlotLabel->"Y-axis",
    ColorFunction->Function[{height},Opacity[height]],
    ChartStyle->Purple
  ],
  ListPlot[
    data,
    PlotStyle->{Purple,PointSize[0.015]},
    AspectRatio->1,
    Frame->True,
    Axes->False
  ]
}
]

```

Output**Mathematica code 2.31****NormalDistribution**

Input (* The code generates a plot of the probability density function (PDF) for a normal distribution with different values of standard deviation $\sigma = (0.75, 1 \text{ and } 2)$ and a fixed mean ($\mu=0$). The plot shows the values of the PDF for all possible values of x between -7 and 7. The resulting plot shows the bell-shaped curve of the normal distribution with different shapes, reflecting the impact of varying the value of the standard deviation: *)

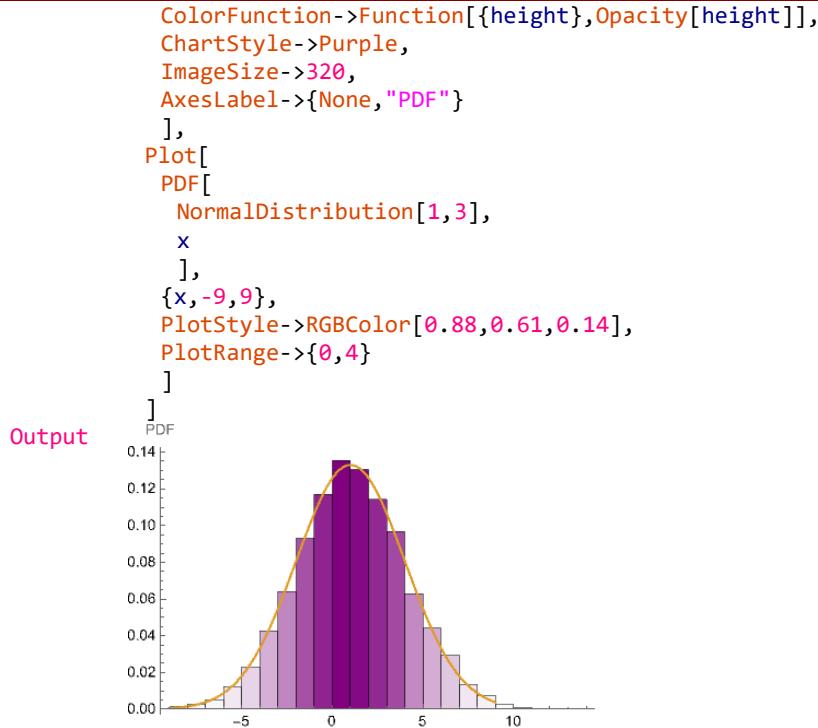
```
Plot[
 Evaluate[
 Table[
 PDF[
 NormalDistribution[0, \sigma],
 x
 ],
 {\sigma, {0.75, 1, 2}}
 ]
 ],
 {x, -7, 7},
 PlotRange -> All,
 Filling -> Axis,
 PlotLegends -> Placed[{{"\u03bc=0,\u03c3=0.75", "\u03bc=0,\u03c3=1", "\u03bc=0,\u03c3=2"}, {0.8, 0.75}}, {0.8, 0.75}],
 PlotStyle -> {RGBColor[0.88, 0.61, 0.14], RGBColor[0.37, 0.5, 0.7], Purple},
 ImageSize -> 320,
 AxesLabel -> {None, "PDF"}
 ]
```

Output**Mathematica code 2.32****NormalDistribution**

Input (* The code generates a histogram and a plot of the PDF for a Normal Distribution with parameters $\mu=1$ and $\sigma=3$ and sample size 10000: *)

```
data = RandomVariate[
 NormalDistribution[1, 3], 10^4
 ];
 Show[
 Histogram[
 data,
 20,
 "PDF",

```

**Mathematica code 2.33****NormalDistribution**

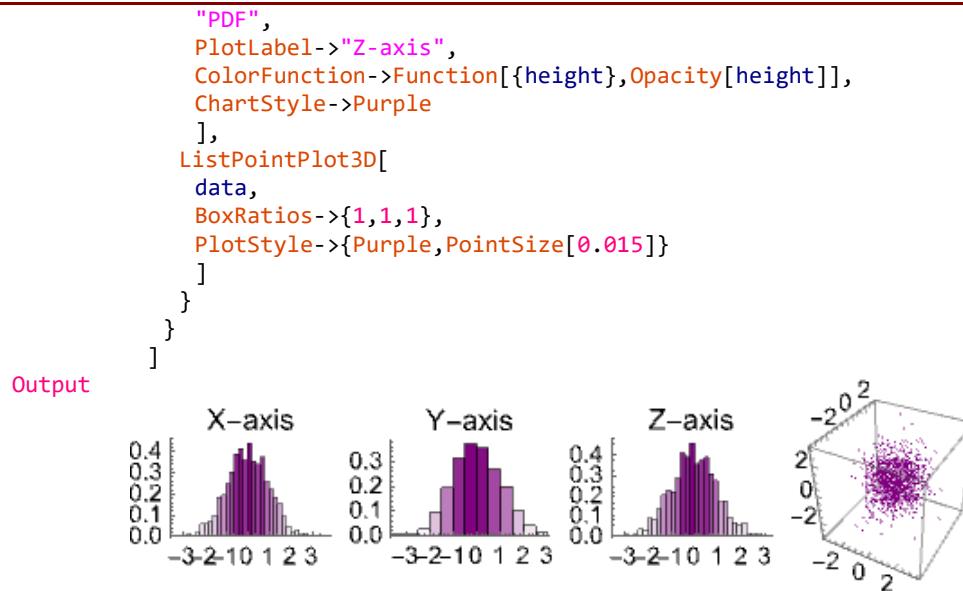
Input (* The code generates a set of random data points with a normal distribution in three dimensions, and then creates three histograms, one for each dimension, showing the distribution of the points along that axis. Additionally, it creates a 3D scatter plot of the data points: *)

```

data=RandomVariate[
NormalDistribution[0,1],
{1000,3}
];

GraphicsGrid[
{
{
Histogram[
data[[All,1]],
Automatic,
"PDF",
PlotLabel->"X-axis",
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple
],
Histogram[
data[[All,2]],
Automatic,
"PDF",
PlotLabel->"Y-axis",
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple
],
Histogram[
data[[All,3]],
Automatic,

```

**Mathematica Examples 3.34**

```

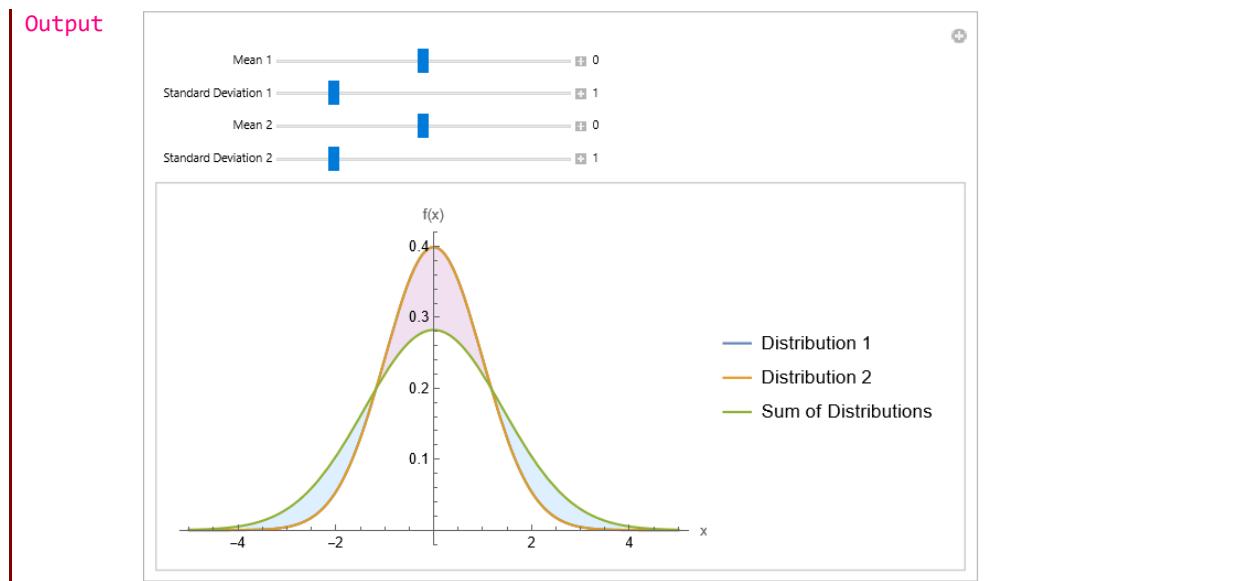
Input (* The code creates a Manipulate interface where the user can adjust the parameters
of two normal distributions (dist1 and dist2) and see the resulting sum of these
distributions (distSum) plotted on the same graph. The plot shows the PDFs of each
distribution as well as the PDF of the sum of the distributions. The
TransformedDistribution function is used to create the sum of two distributions by
defining the distribution of the sum of two random variables, x and y, where x follows
dist1 and y follows dist2. Hence, we prove that Normal distribution is closed under
addition: *)

Manipulate[
  dist1=NormalDistribution[mean1,sd1];
  dist2=NormalDistribution[mean2,sd2];

  distSum=TransformedDistribution[
    x+y,
    {Distributed[x,dist1],Distributed[y,dist2]}];

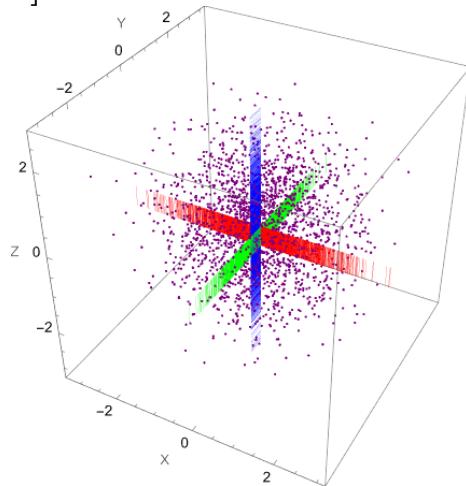
  Plot[
  {
    PDF[dist1,x],
    PDF[dist2,x],
    PDF[distSum,x]
  },
  {x,-5,5},
  PlotRange->All,
  AxesLabel->{"x","f(x)"},
  Filling->{1->{2},2->{3}},
  FillingStyle->{LightBlue,LightPurple},
  PlotLegends->{"Distribution 1","Distribution 2","Sum of Distributions"}
  ],
  {{mean1,0,"Mean 1"},-5,5,Appearance->"Labeled"},
  {{sd1,1,"Standard Deviation 1"},0.1,5,Appearance->"Labeled"},
  {{mean2,0,"Mean 2"},-5,5,Appearance->"Labeled"},
  {{sd2,1,"Standard Deviation 2"},0.1,5,Appearance->"Labeled"}
]

```

**Mathematica Examples 3.35**

Input

```
(* The code generates a 3D scatter plot of normally distributed points, where the x-axis is red, y-axis is green, and z-axis is blue: *)
data=RandomVariate[
  NormalDistribution[0,1],
  {2000,3}
];
Graphics3D[
{
{PointSize[0.006],Purple,Point[data]},
Thin,
{Red,Opacity[0.4],Line[{#,0,0},{#,0,-0.5}]}&/@data[[All,1]],
Thin,
{Green,Opacity[0.4],Line[{{0,#,0},{0,#,-0.5}}]}&/@data[[All,2]],
Thin,
{Blue,Opacity[0.4],Line[{{0,0,#},{0,-0.5,#}}]}&/@data[[All,3]]},
BoxRatios->{1,1,1},
Axes->True,
AxesLabel->{"X","Y","Z"},
ImageSize->320
]
```

Output

Mathematica code 2.36**NormalDistribution and EstimatedDistribution**

Input

```

(* The code demonstrates a common technique in statistics and data analysis, which
is the use of random sampling to estimate population parameters. The code generates
random samples from a normal distribution with mean 0 and standard deviation 1, and
then using these samples to estimate the parameters of another normal distribution
with unknown mean and standard deviation. This process is repeated 20 times, resulting
in 20 different estimated distributions. The code also visualizes the resulting
estimated distributions using the PDF function. The code plots the PDFs of these
estimated distributions using the PDF function and the estimated parameters. The plot
shows the PDFs in a range from -3.5 to 3.5. The code also generates a list plot of
2 sets of random samples from the normal distribution with mean 0 and standard
deviation 1. The plot shows the 100 random points generated from two random samples.
The code generates also a histogram of the PDF for Normal distribution of the two
samples: *)

estim0distributions=Table[
  dist=NormalDistribution[0,1];

  sampledata=RandomVariate[
    dist,
    100
  ];

  ed=EstimatedDistribution[
    sampledata,
    NormalDistribution[ $\alpha$ , $\beta$ ]
  ],
  {i,1,20}
]

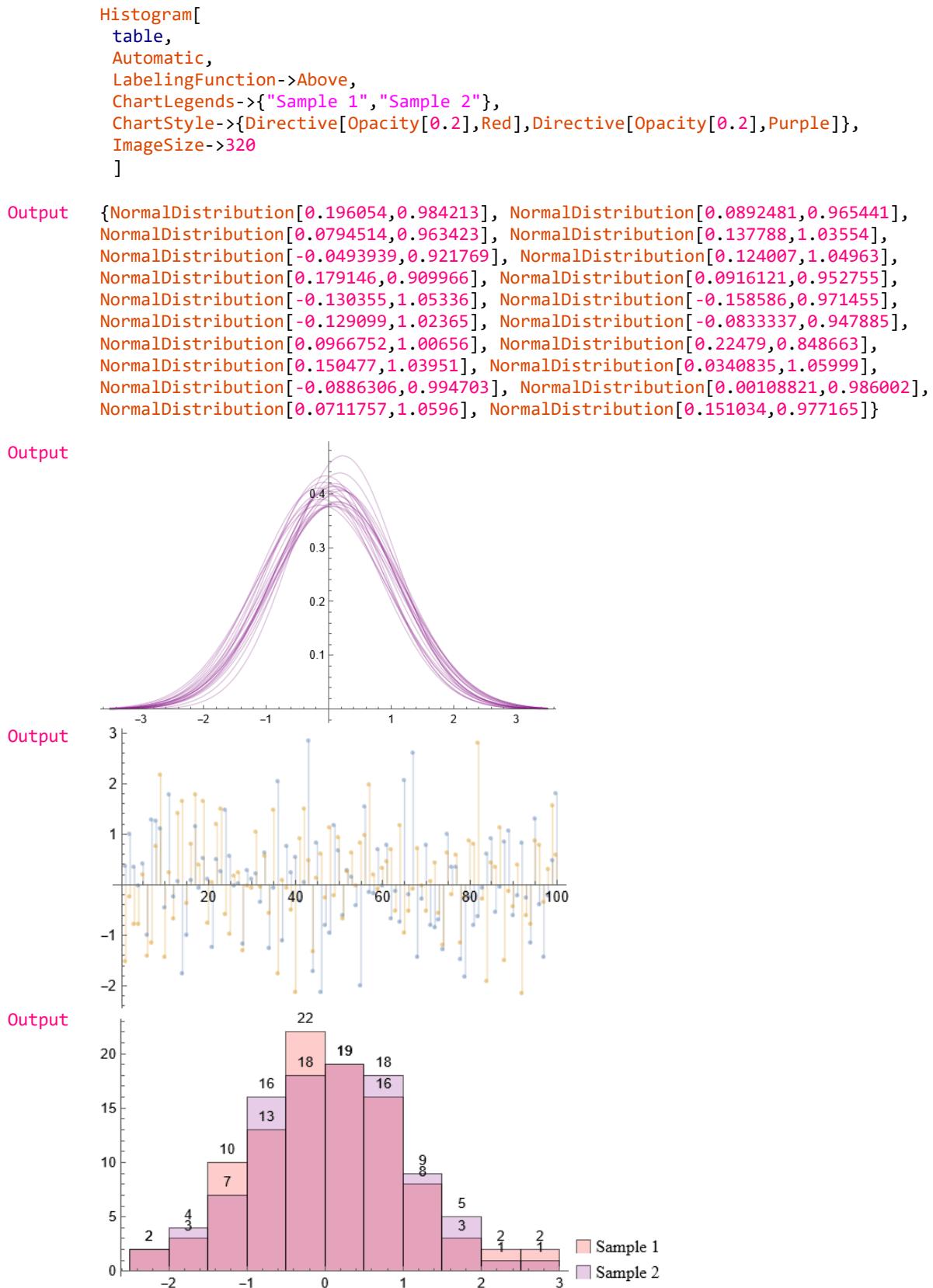
pdf0ed=Table[
  PDF[estim0distributions[[i]],x],
  {i,1,20}
];

(* Visualizes the resulting estimated distributions *)
Plot[
  pdf0ed,
  {x,-3.5,3.5},
  PlotRange->Full,
  ImageSize->400,
  PlotStyle->Directive[Purple,Opacity[0.3],Thickness[0.002]]
]

(* Visualizes 100 random points generated from two random samples *)
table=Table[
  dist=NormalDistribution[0,1];
  sampledata=RandomVariate[
    dist,
    100],
  {i,1,2}
];

ListPlot[
  table,
  ImageSize->320,
  Filling->Axis,
  PlotStyle->Directive[Opacity[0.5],Thickness[0.003]]
]

```



CHAPTER 3

MATRIX CALCULUS AND GRADIENT OPTIMIZATION

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 2, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation. This chapter also serves as a summary of the book titled *Mathematics for Machine Learning and Data Science: Optimization with Mathematica Applications*. For detailed proofs of theorems, additional examples, and comprehensive explanations, including Mathematica applications, please refer to Ref [11]."

In the realm of AI and machine learning, the modern development of NNs stands as a testament to the marriage of sophisticated mathematical frameworks and computational ingenuity. At the heart of this evolution lies the profound influence of matrix calculus [12-14] and gradient optimization techniques [15-23]. These foundational pillars have not only reshaped the landscape of NN design but have also propelled advancements in diverse domains, ranging from computer vision to natural language processing and beyond.

Matrix calculus, with its roots in linear algebra, provides a powerful toolset for analyzing and manipulating multidimensional data structures. It offers a systematic framework for computing derivatives and gradients of functions involving matrices and vectors, enabling efficient optimization in high-dimensional spaces. Matrix calculus is indeed essential for building and training NNs. NNs, especially deep learning models, heavily rely on matrix operations for their computations.

Here's why matrix calculus is crucial for NNs:

- NNs are typically represented and implemented using matrices and vectors. Each layer in a NN can be seen as a matrix operation, where inputs (vectors) are multiplied by weights (matrices) and passed through functions (activation functions).
- The training of NNs often involves optimization algorithms like Gradient Descent (GD). Matrix calculus provides the necessary tools to compute gradients efficiently, enabling the optimization process to update the network parameters (weights) in the direction that minimizes the objective (loss) function.
- Backpropagation is the primary algorithm used to compute gradients efficiently in NNs. It is essentially an application of the chain rule from calculus, which involves matrix multiplication and transposition operations.
- Matrix calculus allows for efficient computation of derivatives and gradients in NNs. This efficiency is crucial for training deep NNs, which may have millions of parameters.
- Most deep learning frameworks handle much of the matrix calculus under the hood. However, understanding the underlying principles of matrix calculus can help in debugging, optimizing, and customizing NN architectures.

Complementing matrix calculus is the arsenal of gradient optimization techniques, which are fundamental to training NNs. By iteratively adjusting model parameters in the direction of steepest descent, these methods seek to minimize a predefined objective function, such as the loss function in supervised learning tasks. From classic algorithms like GD to more advanced variants like Stochastic Gradient Descent (SGD) and Adaptive Moment (Adam) optimization, these techniques play a pivotal role in navigating the vast landscape of model parameter space efficiently and effectively.

This chapter aims to introduce the concepts of numerical differentiation, matrix calculus, gradient optimization, and demonstrate their implementation using Mathematica.

- Numerical differentiation is a crucial technique in computational mathematics, enabling the approximation of derivatives from discrete data points. This is particularly useful when analytical differentiation is challenging or impossible.
- Matrix calculus extends traditional calculus to higher dimensions, dealing with functions that map matrices to scalars, vectors, or other matrices.
- Optimization is at the heart of machine learning, where we aim to minimize a loss (or objective) function. Gradient-based methods, such as gradient descent, iteratively adjust the parameters to reduce the loss.

By the end of this chapter, you will be able to:

- Implement matrix calculus operations in Mathematica.
- Implement numerical differentiation techniques in Mathematica.
- Utilize Mathematica to perform gradient-based optimization.

This chapter will include detailed explanations, examples, and Mathematica code snippets to ensure a comprehensive understanding of these topics.

Mathematica Code 3.1

```

Input      (* The code demonstrates the numerical approximation of the first derivative of
           the function f(x)= sin(x) at x0=Pi/4 using finite difference methods. It calculates
           the forward, backward, and central difference approximations with a step size
           h=0.1, and compares these results to the exact derivative obtained analytically.
           The code prints the approximations and the exact derivative, and generates a plot
           of the function over the interval [x_0-h,x_0+h], highlighting the point at x0
           where the derivative is approximated: *)

(* Define the function to differentiate: *)
f[x_]:=Sin[x]

(* Define the point at which to approximate the derivative: *)
x0=Pi/4;

(* Define the step size: *)
h=0.1;

(* Forward difference approximation: *)
forwardDifference=(f[x0+h]-f[x0])/h;

(* Backward difference approximation: *)
backwardDifference=(f[x0]-f[x0-h])/h;

(* Central difference approximation: *)
centralDifference=(f[x0+h]-f[x0-h])/(2 h);

(* Exact derivative for comparison: *)
exactDerivative=D[f[x],x]/. x->x0;

(* Print results: *)
Print["Forward Difference Approximation: ",forwardDifference]
Print["Backward Difference Approximation: ",backwardDifference]
Print["Central Difference Approximation: ",centralDifference]
Print["Exact Derivative: ",N[exactDerivative]]

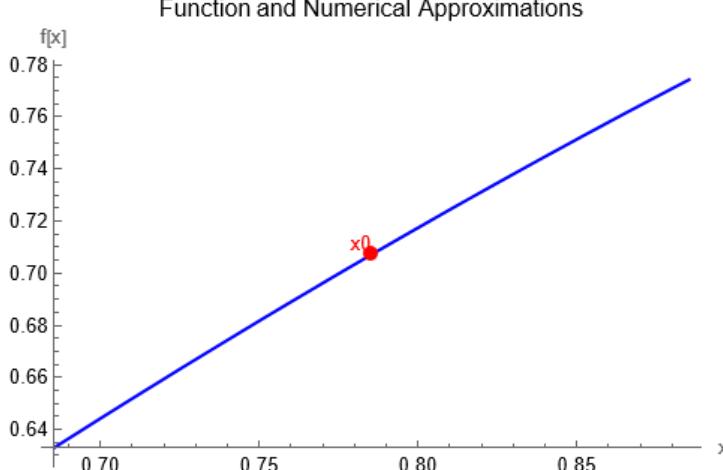
```

```
(* Plotting the function and the approximations: *)
plotFunction=Plot[
  f[x],
  {x,x0-h,x0+h},
  PlotStyle->Blue,
  PlotLabel->"Function and Numerical Approximations",
  Epilog->{
    Red,
    PointSize[Large],
    Point[{x0,f[x0]}],
    Text["x0",{x0,f[x0]},{1,-1}]
  },
  AxesLabel->{"x","f[x]"}
];

Show[plotFunction]
```

Output

Forward Difference Approximation: 0.670603
 Backward Difference Approximation: 0.741255
 Central Difference Approximation: 0.705929
 Exact Derivative: 0.707107

**Mathematica Code 3.2**

Input

(* The goal of the `Manipulate` code is to dynamically illustrate and compare the numerical approximations of the first derivative of the function $f(x) = \sin(x)$ at $x_0 = \pi/4$ using finite difference methods (forward, backward, and central differences) with a variable step size h . The code calculates these approximations and the exact derivative, displays the results, and plots the function along with the tangent lines corresponding to the numerical approximations. The interactive slider allows users to adjust the step size h , providing a visual and numerical understanding of how the step size affects the accuracy of the derivative approximations: *)

```
Manipulate[
 Module[
 {f,x0,forwardDifference,backwardDifference,centralDifference,exactDerivative,plo
 tFunction},
 (* Define the function to differentiate: *)
 f[x_]:=Sin[x];
 (* Define the point at which to approximate the derivative: *)
 x0=Pi/4;
```

```

(* Forward difference approximation: *)
forwardDifference=(f[x0+h]-f[x0])/h;

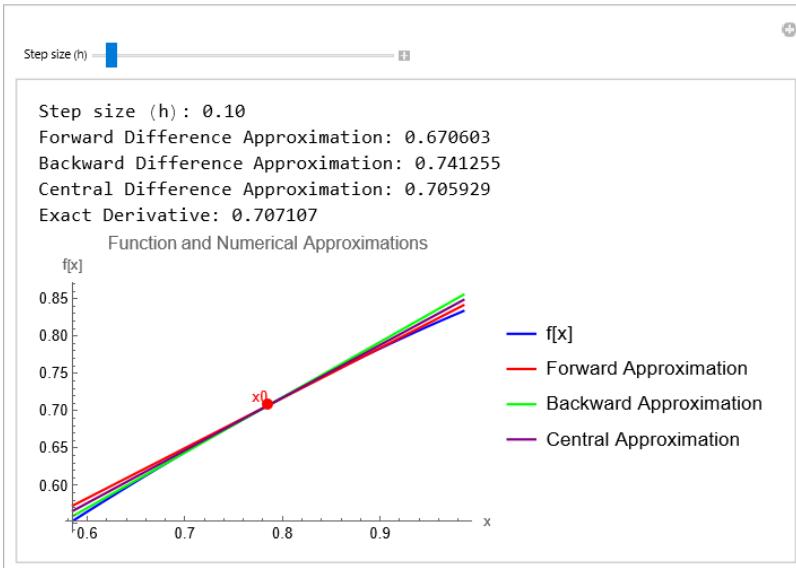
(* Backward difference approximation: *)
backwardDifference=(f[x0]-f[x0-h])/h;

(* Central difference approximation: *)
centralDifference=(f[x0+h]-f[x0-h])/(2 h);

(* Exact derivative for comparison: *)
exactDerivative=D[f[x],x]/. x->x0;

(* Display results in a column format: *)
Column[
{
  Row[{"Step size (h): ", NumberForm[h,{4,2}]}],
  Row[{"Forward Difference Approximation: ",
    NumberForm[forwardDifference,{8,6}]}],
  Row[{"Backward Difference Approximation: ",
    NumberForm[backwardDifference,{8,6}]}],
  Row[{"Central Difference Approximation: ",
    NumberForm[centralDifference,{8,6}]}],
  Row[{"Exact Derivative: ", NumberForm[N[exactDerivative],{8,6}]}],
  (* Plot the function and the approximations *)
  Plot[
  {
    f[x],
    (* Tangent line using forward difference approximation: *)
    f[x0]+(x-x0) forwardDifference,
    (* Tangent line using backward difference approximation: *)
    f[x0]+(x-x0) backwardDifference,
    (* Tangent line using central difference approximation: *)
    f[x0]+(x-x0) centralDifference},
    (*Plot range*)
    {x,x0-2 h,x0+2 h},
    PlotStyle->{Blue,Red,Green,Purple},
    PlotLegends->{"f[x]","Forward Approximation","Backward
    Approximation","Central Approximation"},
    Epilog->{
      Red,
      PointSize[Large],
      Point[{x0,f[x0]}],
      Text["x0",{x0,f[x0]},{1,-1}]
    },
    PlotLabel->"Function and Numerical Approximations",
    AxesLabel->{"x","f[x]"},
    ImageSize->300
  ]
}
]
],
(* Manipulate control for step size (h): *)
{{h,0.1,"Step size (h)"},0.001,2,0.01}
]

```

Output**Mathematica Code 3.3**

```
Input (* Define the scalar function f(x): *)
f[x_]:=x^2
(* Compute the derivative of f with respect to x: *)
D[f[x],x]
Output 2 x
```

Mathematica Code 3.4

```
Input (* The code defines a scalar function f(x)=a.x, where x is a vector {x1,x2,x3} and a is a constant vector {a1,a2,a3}, representing the dot product of a and x. It then computes the gradient of f with respect to the vector x using the `Grad` function, resulting in the vector {a1,a2,a3}, which consists of the partial derivatives of f with respect to each component of x: *)
(* Define the scalar function f(x) where x is a vector: *)
a={a1,a2,a3};
x={x1,x2,x3};
f[x_]:=a.x

(* Compute the gradient of f with respect to vector x: *)
Grad[f[x],x]
Output {a1,a2,a3}
```

Mathematica Code 3.5

```
Input (* The code defines a scalar function f(X)= Tr(Transpose(A). X), where X and A are (2*2) matrices. The function computes the trace of the product of the transpose of matrix A and matrix X. It then calculates the gradient of f with respect to each element of the matrix X using the `Table` and `D` functions, resulting in a (2*2) matrix where each element is the partial derivative of f with respect to the corresponding element in X: *)
(* Define the scalar function f(X) where X is a matrix: *)
X={{x11,x12},{x21,x22}};
A={{a11,a12},{a21,a22}};
```

```

Tr[Transpose[A].X]
f[X_]:=Tr[Transpose[A].X]
(* Compute the derivative with respect to each element of the matrix X: *)
grad=Table[D[f[X],X[[i,j]]],{i,1,2},{j,1,2}]

Output      a11 x11+a12 x12+a21 x21+a22 x22
Output      {{a11,a12},{a21,a22}}

```

Mathematica Code 3.6

```

Input      (* The code defines a vector function f(x)= {x^2,x^3} and computes its derivative
           with respect to x. The `D` function is used to differentiate each component of
           the vector function, resulting in the vector of derivatives {2x,3x^2}: *)
(* Define the vector function f(x): *)
f[x_]:={x^2,x^3}

(* Compute the derivative of f with respect to x: *)
D[f[x],x]
Output     {2 x,3 x2}

```

Mathematica Code 3.7

```

Input      (* The code defines a vector function f(x_1,x_2)= {x_1^2+x_2,x_1 x_2} and computes
           its Jacobian matrix with respect to the vector x= {x_1,x_2}. The `D` function is
           used to differentiate each component of the vector function with respect to x_1
           and x_2, resulting in the Jacobian matrix: *)
(* Define the vector function f(x): *)
f[x1_,x2_]:= {x1^2+x2,x1 x2}
(* Compute the Jacobian of f with respect to vector x: *)
JacobianMatrix=D[f[x1,x2],{{x1,x2}}]

Output    {{2 x1,1},{x2,x1}}

```

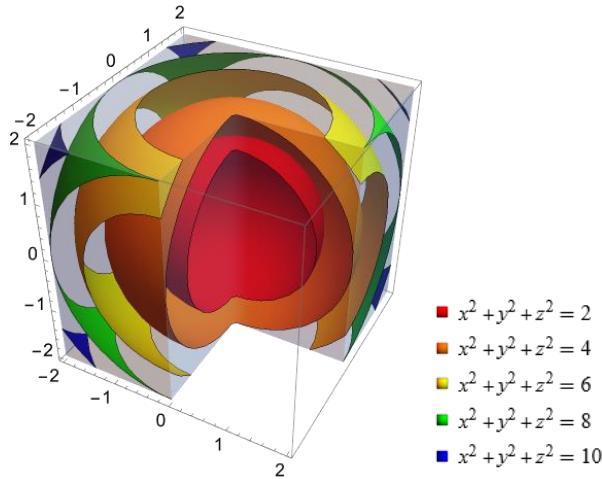
Mathematica Code 3.8

```

Input      (* The code creates a 3D contour plot of the function x^2+y^2+z^2 over the ranges
           [-2,2] for x, y, and z. The plot includes 5 contour levels and is restricted to
           the region where x<0 or y>0. The contour levels are colored in Red, Orange, Yellow,
           Green, and Blue: *)
ContourPlot3D[
  x^2+y^2+z^2,
  {x,-2,2},
  {y,-2,2},
  {z,-2,2},
  Contours->5,
  (* Restrict the plot to the region where x<0 or y>0: *)
  RegionFunction->Function[{x,y,z},x<0||y>0],
  ContourStyle->{Red,Orange,Yellow,Green,Blue},
  Mesh->None,
  LabelStyle->Directive[Black,13],
  PlotLegends->"Expressions",
  ImageSize->300
]

```

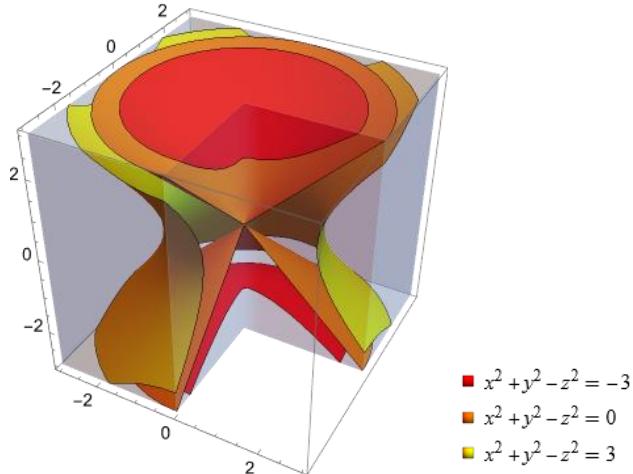
Output

**Mathematica Code 3.9**

Input

```
(* The code creates a 3D contour plot of the function x^2+y^2-z^2 over the ranges
[-3,3] for x, y, and z. The plot includes specific contour levels at -3, 0, and
3, with the contour levels colored in Red, Orange, and Yellow. The plot is
restricted to the region where x<0 or y>0 using the `RegionFunction`: *)
ContourPlot3D[
  x^2+y^2-z^2,
  {x, -3, 3},
  {y, -3, 3},
  {z, -3, 3},
  Contours -> {-3, 0, 3},
  ContourStyle -> {Red, Orange, Yellow},
  Mesh -> None,
  (* Restrict the plot to the region where x<0 or y>0: *)
  RegionFunction -> Function[{x, y, z}, x < 0 || y > 0],
  PlotLegends -> "Expressions",
  ImageSize -> 250
]
```

Output

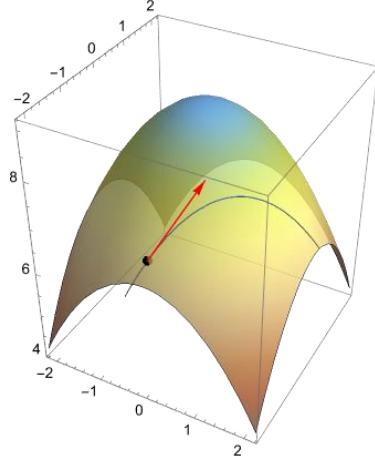
**Mathematica Code 3.10**

Input

```
(* ResourceFunction["DirectionalDerivative"] [f, vars, pt, v] computes the
directional derivative of the function f of the variables vars at the point pt in
the direction of the normalized vector v: *)
```

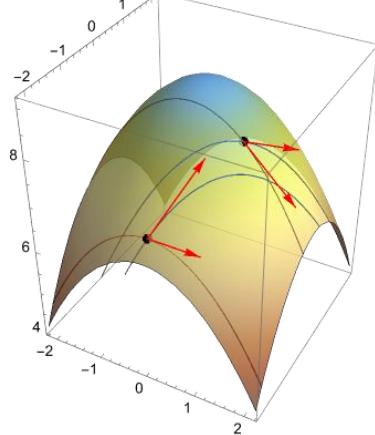
```
(* Visualize a directional derivative: *)
ResourceFunction["DirectionalDerivativePlot3D"][
 9-x^2/2-y^2/2,
 {x,-2.2,2.2},
 {y,-2.2,2.2},
 {θ,-2},
 {Cos[π/4],Sin[π/4]},
 ColorFunction->"SouthwestColors",
 ImageSize->250
]
```

Output

**Mathematica Code 3.11**

```
Input (* Plot using two direction vectors and two points:*)
ResourceFunction["DirectionalDerivativePlot3D"][
 9-x^2/2-y^2/2,
 {x,-2.2,2.2},
 {y,-2.2,2.2},
 {
  {0,-2},{1,-1/2}
 },
 {
  {Cos[π/4],Sin[π/4]},{1,0}
 },
 ColorFunction->"SouthwestColors",
 ImageSize->250
]
```

Output



Now, let's consider a simple example to illustrate how the gradient descent method works. Mathematica codes 3.12 and 3.13 create interactive visualization demonstrating the gradient descent optimization process for a one- and two-dimensional loss function, respectively. Here's an overview of how the Mathematica codes 3.12 achieves this goal:

- The code defines a quadratic loss function $J(\theta) = (\theta - 3)^2 + 5$, representing a curve in a one-dimensional space.
- The `GradientDescent` function is implemented to perform the gradient descent optimization algorithm. This function iteratively updates the parameter θ based on the gradient of the loss function until convergence or the maximum number of iterations is reached. It returns the history of parameter values during optimization.
- The `Manipulate` function from Mathematica is utilized to create an interactive interface. Users can adjust parameters such as the learning rate, initial guess for θ , tolerance for convergence, and maximum iterations. The interface dynamically updates the visualization based on user inputs.
- The loss function curve $J(\theta)$ is plotted against the parameter θ . Additionally, the optimization path taken by gradient descent is displayed as a sequence of points (red) and arrows (blue) on the plot. Users can observe and analyze how changes in the parameters influence the optimization process and its convergence towards the minimum of the loss function.

Let's discuss how users can observe the effects on the optimization path and convergence behavior using the provided interactive visualization.

1. The learning rate determines the step size of each iteration in the gradient descent algorithm. Users can adjust the learning rate using the slider labeled "Learning Rate." A higher learning rate may lead to faster convergence but can also cause overshooting or oscillations around the minimum. Conversely, a lower learning rate may result in slower convergence but with more stable behavior. Users can experiment with different learning rates to observe how they affect the optimization path and convergence behavior.
2. The initial guess for the parameter θ determines the starting point of the optimization process. Users can adjust the initial guess using the slider labeled "Initial Guess." Different initial guesses may lead to different optimization paths and convergence behaviors. Users can explore how changing the initial guess influences the optimization path and convergence behavior.
3. The tolerance parameter determines the convergence criterion for the optimization process. If the absolute gradient of the loss function falls below the tolerance value, the optimization process is considered converged. Users can adjust the tolerance using the slider labeled "Tolerance." A smaller tolerance value leads to stricter convergence criteria, potentially requiring more iterations for convergence. Users can observe how changing the tolerance affects the number of iterations and convergence behavior.
4. The maximum iterations parameter limits the number of iterations allowed for the optimization process. If the optimization process does not converge within the specified maximum iterations, it terminates. Users can adjust the maximum iterations using the slider labeled "Max Iterations." Setting a higher maximum iterations value allows for more iterations, potentially leading to convergence even with slower convergence rates. Users can experiment with different maximum iterations settings to observe their effects on convergence behavior.

Mathematica Code 3.12

Input

```
(* The code creates an interactive visualization demonstrating the gradient descent
optimization process for a one-dimensional loss function. A quadratic loss function
J(\theta)=(\theta-3)^2+5 is defined, representing a curve in a one-dimensional space. The
code implements the gradient descent optimization algorithm in the GradientDescent
function. This function iteratively updates the parameter \theta based on the gradient
of the loss function until convergence or the maximum number of iterations is
reached. Users can adjust parameters like the learning rate, initial guess for \theta,
tolerance for convergence, and maximum iterations. The interface dynamically
updates the visualization based on user inputs. The loss function curve J(\theta) is
plotted against the parameter \theta. The optimization path taken by gradient descent
is displayed as a sequence of points (red) and arrows (blue) on the plot. Users
```

can observe and analyze how changes in the parameters influence the optimization process and its convergence towards the minimum of the loss function: *)

```

Manipulate[
Module[
{θHistory, θOpt, minLoss},

(* Define the loss function: *)
J[θ_]:= (θ-3)^2+5;

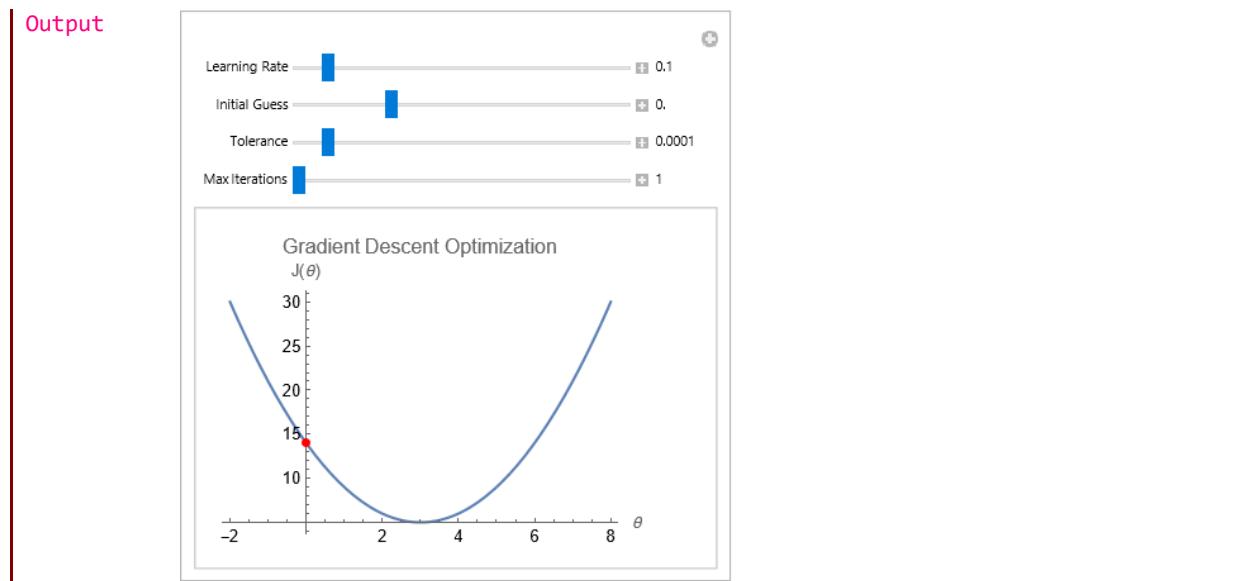
(* Gradient Descent Algorithm: *)
GradientDescent[α_, θ₀_, tol_, maxIter_]:=Module[
{θ=θ₀, history={θ₀}, iter=0},
While[
  iter<maxIter&&Abs[D[J[x],x]/. x->θ]>tol,
  θ=θ-α D[J[x],x]/. x->θ;
  AppendTo[history,θ];
  iter++;
];
history
];

(* Perform gradient descent: *)
θHistory=GradientDescent[α,θ₀,tol,maxIter];

(* Plot the loss function and optimization path with arrows: *)
plot=Plot[
  J[θ],
  {θ,-2,8},
  PlotRange->All,
  Epilog->{
    Red,
    PointSize[0.02],
    Point[{#,J[#]}&/@Most[θHistory]],
    Blue,
    Arrowheads[0.03],
    Arrow/@Partition[Thread[{Most[θHistory],J/@Most[θHistory]}],2,1]
  },
  PlotLabel->"Gradient Descent Optimization",
  AxesLabel->{"θ", "J(θ)" },
  ImageSize->250
];
plot
],


(* Controls: *)
{{α,0.1,"Learning Rate"},0.01,1,0.01,Appearance->"Labeled"}, 
{{θ₀,0.0,"Initial Guess"},-2,5,0.1,Appearance->"Labeled"}, 
{{tol,0.0001,"Tolerance"},0.00001,0.001,0.00001,Appearance->"Labeled"}, 
{{maxIter,1,"Max Iterations"},1,15,1,Appearance->"Labeled"}]
]

```

**Mathematica Code 3.13**

Input

```
(* The code offers an interactive visualization of the gradient descent optimization process for a quadratic loss function. A quadratic loss function, J(x,y), is defined to represent a surface in the 2D space. This function serves as the target for optimization. The code implements the gradient descent optimization algorithm in the GradientDescent function. This function iteratively updates the parameters x and y based on the gradient of the loss function, aiming to minimize it. Users can specify parameters like the learning rate, initial guesses for x and y, tolerance for convergence, and maximum number of iterations. As users modify these parameters, the optimization process is visually represented on a contour plot. The contour plot displays the loss function surface. The optimization path taken by gradient descent is depicted as a sequence of points (blue) and arrows (red) on this plot. Users can observe how changes in the parameters affect the optimization process and its convergence towards the minimum of the loss function: *)
```

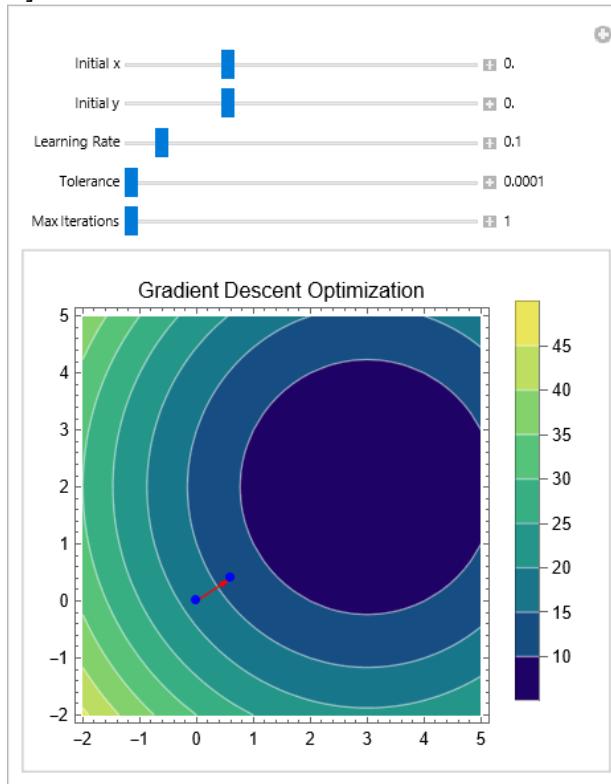
```
(* Define the loss function: *)
J[x_,y_]:= (x-3)^2+(y-2)^2+5
(* Gradient Descent Algorithm: *)
GradientDescent[a_,{x0_,y0_},tol_,maxIter_]:=Module[
{x=x0,y=y0,history={{x0,y0}},iter=0,grad},
While[
  iter<maxIter&&Norm[grad=D[J][xp,yp],{{xp,yp}}]/. {xp->x,yp-
>y}>tol,{x,y}={x,y}-a grad;
  AppendTo[history,{x,y}];
  iter++;
];
history
]
(* Manipulate interface: *)
Manipulate[
Module[
{history,contourPlot,arrows},
(* Perform gradient descent: *)
history=GradientDescent[a,{x0,y0},tol,maxIter];
(* Contour plot of the loss function: *)
contourPlot=ContourPlot[
J[x,y],
```

```

{x,-2,5},
{y,-2,5},
PlotLabel->"Gradient Descent Optimization",
AxesLabel->{"x","y"},
PlotRange->All,
Epilog->{Blue,PointSize[0.02],Point[history]},
Contours->Function[{min,max},Range[min,max,5]],
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
LabelStyle->Directive[Black,10],
ImageSize->250
];
(* Add arrows to the contour plot: *)
arrows=Graphics[
{
Red,Arrowheads[0.03],
Arrow/@Partition[history,2,1]
},
PlotRange->All
];
(* Combine the contour plot and arrows: *)
Show[contourPlot,arrows]],
(* Controls: *)
{{x0,0.0,"Initial x"},-2,5,Appearance->"Labeled"}, 
{{y0,0.0,"Initial y"},-2,5,Appearance->"Labeled"}, 
{{α,0.1,"Learning Rate"},0.01,1,Appearance->"Labeled"}, 
{{tol,0.0001,"Tolerance"},0.0001,0.1,Appearance->"Labeled"}, 
{{maxIter,1,"Max Iterations"},1,100,1,Appearance->"Labeled"}, 
TrackedSymbols:>{x0,y0,α,tol,maxIter}
]

```

Output



Moreover, for many real-world optimization problems involving multivariable functions, analytical solutions may not be feasible, and numerical methods like the conjugate gradient, principal axis, Levenberg Marquardt, Newton, quasi Newton, interior point, and linear programming, are indispensable for finding approximate solutions. Each method has its advantages and limitations, and the choice of method depends on the specific characteristics of the optimization problem at hand. In the following chapters of this book, we will explore numerical optimization methods for neural networks in depth. For now, let us utilize Mathematica to implement solutions for optimization problems. We focus on the practical utilization of Mathematica's optimization functions for optimization problems. From identifying optimal solutions to visualizing the optimization process, Mathematica equips users with powerful tools to tackle a wide array of optimization challenges efficiently and accurately.

The commands `FindMinimum`, `NMinimize`, `Minimize` and `FindMinimumPlot` can do optimization for both single-variable and multivariable functions.

FindMinimum

`FindMinimum[f, x]`

searches for a local minimum in f , starting from an automatically selected point.

`FindMinimum[f, {x, x0}]`

searches for a local minimum in f , starting from the point $x=x_0$.

`FindMinimum[f, {{x, x0}, {y, y0}, ...}]`

searches for a local minimum in a function of several variables.

`FindMinimum[{f, cons}, {{x, x0}, {y, y0}, ...}]`

searches for a local minimum subject to the constraints cons .

`FindMinimum[{f, cons}, {x, y, ...}]`

starts from a point within the region defined by the constraints.

The following options can be given:

<code>AccuracyGoal</code>	<code>Automatic</code>	the accuracy sought
<code>EvaluationMonitor</code>	<code>None</code>	expression to evaluate whenever f is evaluated
<code>Gradient</code>	<code>Automatic</code>	the list of gradient components for f
<code>MaxIterations</code>	<code>Automatic</code>	maximum number of iterations to use
<code>Method</code>	<code>Automatic</code>	method to use
<code>PrecisionGoal</code>	<code>Automatic</code>	the precision sought
<code>StepMonitor</code>	<code>None</code>	expression to evaluate whenever a step is taken
<code>WorkingPrecision</code>	<code>MachinePrecision</code>	the precision used in internal computations

Possible settings for `Method` include "`ConjugateGradient`", "`PrincipalAxis`", "`LevenbergMarquardt`", "`Newton`", "`QuasiNewton`", "`InteriorPoint`", and "`LinearProgramming`", with the default being `Automatic`.

NMinimize

`NMinimize[f, x]`

searches for a global minimum in f numerically with respect to x .

`NMinimize[f, {x, y, ...}]`

searches for a global minimum in f numerically with respect to x, y, \dots .

`NMinimize[{f, cons}, {x, y, ...}]`

searches for a global minimum in f numerically subject to the constraints cons .

`NMinimize[..., x ∈ rdom]`

constrains x to be in the region or domain $rdom$.

The following options can be given:

<code>AccuracyGoal</code>	<code>Automatic</code>	number of digits of final accuracy sought
<code>EvaluationMonitor</code>	<code>None</code>	expression to evaluate whenever f is evaluated
<code>MaxIterations</code>	<code>Automatic</code>	maximum number of iterations to use
<code>Method</code>	<code>Automatic</code>	method to use
<code>PrecisionGoal</code>	<code>Automatic</code>	number of digits of final precision sought
<code>StepMonitor</code>	<code>None</code>	expression to evaluate whenever a step is taken
<code>WorkingPrecision</code>	<code>MachinePrecision</code>	the precision used in internal computations

Heuristic methods include:

<code>"NelderMead"</code>	simplex method of Nelder and Mead
<code>"DifferentialEvolution"</code>	use differential evolution
<code>"SimulatedAnnealing"</code>	use simulated annealing
<code>"RandomSearch"</code>	use the best local minimum found from multiple random starting points
<code>"Couenne"</code>	use the Couenne library for non-convex mixed-integer nonlinear problems

Minimize

`Minimize[f, x]`
minimizes f symbolically with respect to x .

`Minimize[f, {x, y, ...}]`
minimizes f symbolically with respect to x, y, \dots

`Minimize[{f, cons}, {x, y, ...}]`
minimizes f symbolically subject to the constraints $cons$.

`Minimize[..., x ∈ rdom]`
constrains x to be in the region or domain $rdom$.

`Minimize[..., ..., dom]`
constrains variables to the domain dom , typically `Reals` or `Integers`.

Remarks:

- `Minimize` is also known as infimum, symbolic optimization and global optimization (GO).
- `Minimize` finds the global minimum of f subject to the constraints given.
- `N[Minimize[...]]` calls `NMinimize` for optimization problems that cannot be solved symbolically.

Moreover, the following Mathematica built-in functions are important.

<code>ArgMin[f, x]</code>	gives a position x_{\min} at which f is minimized.
<code>ArgMin[f, {x, y, ...}]</code>	gives a position $\{x_{\min}, y_{\min}, \dots\}$ at which f is minimized.
<code>NArgMin[f, x]</code>	gives a position x_{\min} at which f is numerically globally minimized.
<code>NArgMin[f, {x, y, ...}]</code>	gives a position $\{x_{\min}, y_{\min}, \dots\}$ at which f is numerically globally minimized.
<code>MinValue[f, x]</code>	gives the minimum value of f with respect to x .
<code>MinValue[f, {x, y, ...}]</code>	gives the exact minimum value of f with respect to x, y, \dots
<code>NMinValue[f, x]</code>	gives the minimum value of f with respect to x .
<code>NMinValue[f, {x, y, ...}]</code>	gives the minimum value of f with respect to x, y, \dots

Mathematica Code 3.14

```

Input    (* The code snippets demonstrate various applications of the `MinValue` function.
        The first snippet finds the minimum value of a univariate function  $4x^4-5x^2+7$  with
        respect to  $x$ . The second snippet extends this to a multivariate function  $x^2+y^2-4)^2+2$ , finding the minimum value with respect to both  $x$  and  $y$ . The third snippet
        introduces a constraint, determining the minimum value of  $x^2+3y$  subject to
         $x^2+y^2\leq 4$ . Finally, the fourth snippet finds the minimum value of a parameterized
        function  $bx^3+ax^2+cx+d$  with respect to  $x$ , returning the result as a function of
        the parameters  $a$ ,  $b$ ,  $c$ , and  $d$ : *)

(* Find the minimum value of a univariate function: *)
MinValue[4 x^4-5 x^2+7,x]
(* Find the minimum value of a multivariate function: *)
MinValue[(x^2+y^2-4)^2+2,{x,y}]
(* Find the minimum value of a function subject to constraints: *)
MinValue[{x^2+3 y,x^2+y^2\leq 4},{x,y}]
(* Find the minimum value as a function of parameters: *)
MinValue[b x^3+a x^2+c x+d,x]

Output  87/16
Output  2
Output  -6
Output  {
        {\!\!\! [Piecewise], {
            {d, c==0&&a>=0&&b==0},
            {(-c^2+4 a d)/(4 a), (c>0&&a>0&&b==0)|| (c<0&&a>0&&b==0)},
            {-\infty, True}
        } }
}

```

Mathematica Code 3.15

```

Input    (* The code snippets utilize the `NMinValue` function to find global minimum values
        in different scenarios. The first snippet computes the global minimum value of the
        univariate function  $4x^4-5x^2+7$  with respect to  $x$ . The second snippet extends this
        to a multivariate function  $x^2+y^2-4)^2+2$ , determining the global minimum value with
        respect to both  $x$  and  $y$ . The third snippet introduces a constraint, finding the
        global minimum value of  $x^2+3y$  subject to the constraint  $x^2+y^2\leq 4$  with respect to
         $x$  and  $y$ : *)

(* Find the global minimum value of a univariate function: *)
NMinValue[4 x^4-5 x^2+7,x]
(* Find the global minimum value of a multivariate function: *)
NMinValue[(x^2+y^2-4)^2+2,{x,y}]
(* Find the global minimum value of a function subject to constraints: *)
NMinValue[{x^2+3 y,x^2+y^2\leq 4},{x,y}]

Output  5.4375
Output  2.
Output  -6.

```

Mathematica Code 3.16

```

Input    (* The code snippets use `FindMinValue` to locate minimum values under different
        conditions: the first snippet finds a minimum value of the univariate function  $4x^4-5x^2+7$ 
        with respect to  $x$ ; the second snippet finds a minimum value of the
        multivariate function  $x^2+y^2-4)^2+2$  with respect to both  $x$  and  $y$ ; and the third
        snippet finds a minimum value of the function  $x^2+3y$  subject to the constraint
         $x^2+y^2\leq 4$  with respect to  $x$  and  $y$ : *)

```

```

(* Find a minimum value of the univariate function: *)
FindMinValue[4 x^4-5 x^2+7,x]
(* Find a minimum value of a multivariate function: *)
FindMinValue[(x^2+y^2-4)^2+2,{x,y}]
(* Find a minimum value of a function subject to constraints: *)
FindMinValue[{x^2+3 y,x^2+y^2<=4},{x,y}]

Output 5.4375
Output 2.
Output -6.

```

Mathematica Code 3.17

```

Input (* The code snippets use `ArgMin` to find minimizer points in various scenarios:
the first snippet identifies the minimizer point of the univariate function  $4x^4-5x^2+7$  with respect to  $x$ ; the second snippet determines the minimizer point of the multivariate function  $x^2+y^2-4)^2+2$  with respect to  $x$  and  $y$ ; the third snippet locates the minimizer point of the function  $x^2+3y$  subject to the constraint  $x^2+y^2 \leq 4$  with respect to  $x$  and  $y$ ; and the fourth snippet finds the minimizer point of the parameterized function  $ax^2+bx+c$  with respect to  $x$ , expressed as a function of the parameters  $a$ ,  $b$ , and  $c$ : *)

(* Find a minimizer point for a univariate function: *)
ArgMin[4 x^4-5 x^2+7,x]
(* Find a minimizer point for a multivariate function: *)
ArgMin[(x^2+y^2-4)^2+2,{x,y}]
(* Find a minimizer point for a function subject to constraints: *)
ArgMin[{x^2+3 y,x^2+y^2<=4},{x,y}]
(* Find a minimizer point as a function of parameters: *)
ArgMin[a x^2+b x+c,x]

Output -(Sqrt[5]/2)/2
Output {0,-2}
Output {0,-2}
Output {
  {\[Piecewise], {
    {-(b/(2 a)), (b>0&&a>0)|| (b<0&&a>0)},
    {0, (b==0&&a==0)|| (b==0&&a>0)},
    {Indeterminate, True}
  }}
}

```

Mathematica Code 3.18

```

Input (* Find a global minimizer point for a univariate function: *)
NArgMin[4 x^4-5 x^2+7,x]
(* Find a global minimizer point for a multivariate function: *)
NArgMin[(x^2+y^2-4)^2+2,{x,y}]
(* Find a global minimizer point for a function subject to constraints: *)
NArgMin[{x^2+3 y,x^2+y^2<=4},{x,y}]

Output 0.790545
Output {1.8151,0.839899}
Output {0.,-2.}

```

Mathematica Code 3.19

```

Input (* Find a point {x} at which the univariate function  $4x^4-5x^2+7$  has a minimum: *)
FindArgMin[4 x^4-5 x^2+7,x]

```

```
(* Find a point {x,y} at which the function  $x^2+y^2-4$ ) $^2+2$  has a minimum: *)
FindArgMin[(x^2+y^2-4)^2+2,{x,y}]
(* Find a point at which a function is a minimum subject to constraints: *)
FindArgMin[{x^2+3 y,x^2+y^2<=4},{x,y}]

Output {0.790569}
Output {1.41421,1.41421}
Output {0.,-2.}
```

Mathematica Code 3.20

```
Input (* The code defines the function  $f(x,y)=\sin(x)\cos(y)$  and achieves several goals: it calculates the minimum value of  $f(x,y)$  over the rectangle  $[0,0]$  to  $[1,1]$ , identifies the location (minimizer point) where this minimum occurs, and creates a 3D plot of the function with the minimum point highlighted. Additionally, it generates a contour plot to visualize the function in 2D, marking and labeling the minimum value location. These visualizations help in understanding the function's behavior and the position of its minimum value within the specified domain: *)

(* Define the function: *)
f[x_,y_]:=Sin[x]*Cos[y]

(* Find the minimum value of the function over the specified rectangle: *)
 minValue=MinValue[
  f[x,y],
  {x,y}\[Element]Rectangle[{0,0},{1,1}]
];

(* Find the minimizer point of the function over the specified rectangle: *)
 minPoint=ArgMin[
  f[x,y],
  {x,y}\[Element]Rectangle[{0,0},{1,1}]
];

(* Create a 3D point with the coordinates of the minimizer point and the corresponding function value: *)
 minPoint3D={minPoint[[1]],minPoint[[2]],f[minPoint[[1]],minPoint[[2]]]};

(* Plot the function over the specified rectangle and highlight the minimum point: *)
Show[
  (* 3D plot of the function: *)
  Plot3D[
    f[x,y],
    {x,0,1},
    {y,0,1},
    PlotRange->All,
    AxesLabel->{"x","y","f(x, y)"},
    PlotLabel->"Plot of f(x, y) = Sin[x] * Cos[y] over \n the rectangle [0, 0] to
    [1, 1]",
    ColorFunction->"BlueGreenYellow",
    Mesh->None,
    ImageSize->250
  ],
  Graphics3D[
  {
    Blue,
    PointSize[Large],
    (* Highlight the minimum point on the 3D plot:*)
    minPoint3D
  }
]
```

```

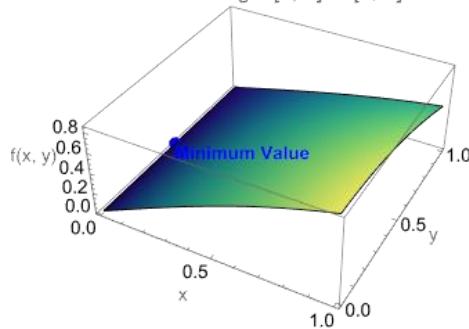
        Point[minPoint3D],
        Text[Style["Minimum Value",Blue,Bold],minPoint3D,{ -1,1}]
    }
]

(* Contour Plot to visualize the function in 2D: *)
ContourPlot[
f[x,y],
{x,0,1},
{y,0,1},
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
Epilog->{
    Blue,
    PointSize[Large],
    (* Plots a point at the location of the minimum value: *)
    Point[{minPoint[[1]],minPoint[[2]]}],
    (* Adds a label "Minimum Value" near the point: *)
    Text[Style["Minimum Value",Blue,Bold],{minPoint[[1]],minPoint[[2]]},{ -1,1}]
},
FrameLabel->{"x","y"},
PlotLabel->"Contour Plot of f(x, y) = Sin[x] * Cos[y] over \n the rectangle [0, 0] to [1, 1]",
ImageSize->250
]
]

```

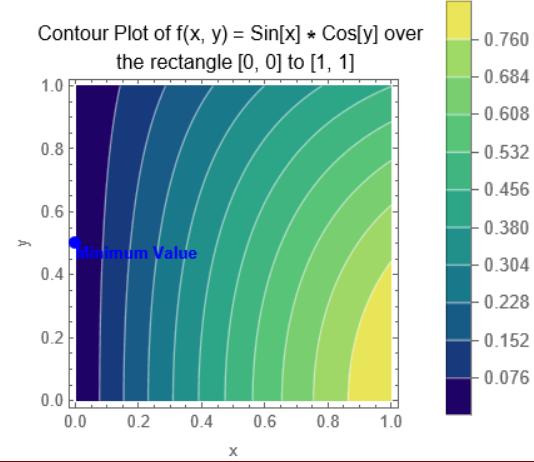
Output

Plot of $f(x, y) = \text{Sin}[x] * \text{Cos}[y]$ over
the rectangle [0, 0] to [1, 1]



Output

Contour Plot of $f(x, y) = \text{Sin}[x] * \text{Cos}[y]$ over
the rectangle [0, 0] to [1, 1]



Mathematica Code 3.21

```

Input    (* Minimize a univariate function: *)
Minimize[4 x^4-5 x^2+7,x]
(* Minimize a multivariate function: *)
Minimize[(x^2+y^2-4)^2+2,{x,y}]
(* Minimize a function subject to constraints: *)
Minimize[{x^2+3 y,x^2+y^2<=4},{x,y}]
(* A minimization problem containing parameters: *)
Minimize[b x^3+a x^2+c x+d,x]

Output   {87/16,{x-->-(Sqrt[5]/2)/2}}
Output   {2,{x->0,y->-2}}
Output   {-6,{x->0,y->-2}}
Output   {{}
          \[Piecewise], {
            {d, (c==0&&a==0&&b==0)||((c==0&&a>0&&b==0)),
             {(-c^2+4 a d)/(4 a), (c>0&&a>0&&b==0)||((c<0&&a>0&&b==0)),
              {-\infty, True}}
            }
          },{x->{
            \[Piecewise], {
              {-(c/(2 a)), (c>0&&a>0&&b==0)||((c<0&&a>0&&b==0)),
               {0, (c==0&&a==0&&b==0)||((c==0&&a>0&&b==0)),
                {Indeterminate, True}}
              }
            }}}
```

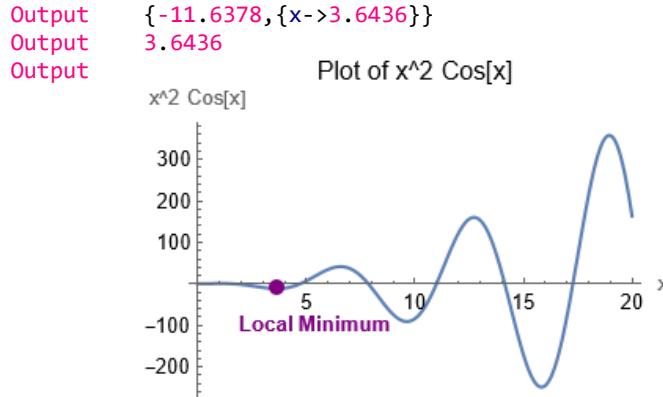
Mathematica Code 3.22

Input

```
(* The code achieves several goals: it finds the local minimum of the function x^2 cos(x) starting from an initial guess of x=2.5 and extracts the value of x at this local minimum. The code then plots the function x^2 cos(x) over the range x=0 to x=20, and highlights the local minimum point on the plot with a purple point and a label. This visualization helps in understanding the behavior of the function and identifying the location of the local minimum within the specified range: *)
(* Extract the value of x at the local minimum *)
localMin=FindMinimum[x^2 Cos[x],{x,2.5}]

(* The value of x at the local minimum: *)
xMin=localMin[[2,1,2]]

(* Plot the function and highlight the local minimum: *)
Plot[
  x^2 Cos[x],
  {x,0,20},
  PlotRange->All,
  AxesLabel->{"x","x^2 Cos[x]"},
  PlotLabel->"Plot of x^2 Cos[x]",
  Epilog->{
    Purple,
    PointSize[Large],
    (*Plot a point at the local minimum*)
    Point[{xMin,xMin^2 Cos[xMin]}],
    (*Add a text label near the local minimum point*)
    Text[Style["Local Minimum",Purple,Bold],{xMin,xMin^2 Cos[xMin]},{-0.5,3}]
  },
  ImageSize->250
]
```

**Mathematica Code 3.23**

```

Input (* This Mathematica code achieves several goals: it finds the minimum of the
       quadratic function  $x^2+y^2$  subject to linear and integer constraints, specifically
        $x+2y \geq 3$ ,  $x \geq 0$ ,  $y \geq 0$ , and  $y$  being an integer. It also determines the minimum of
       the same quadratic function over a geometric region defined by the rectangle  $[-1, -1]$  to  $[1,1]$ . Finally, the code creates a contour plot of the function  $x^2+y^2$  over
       this rectangle and highlights the minimum point found in the previous computation,
       providing a visual representation of the function and the location of its minimum
       value within the specified region: *)

(* Find the minimum of a quadratic function, subject to linear and integer
constraints: *)
FindMinimum[{x^2+y^2, x+2 y>=3&&x>=0&&y>=0&&y\Element Integers},{x,y}]

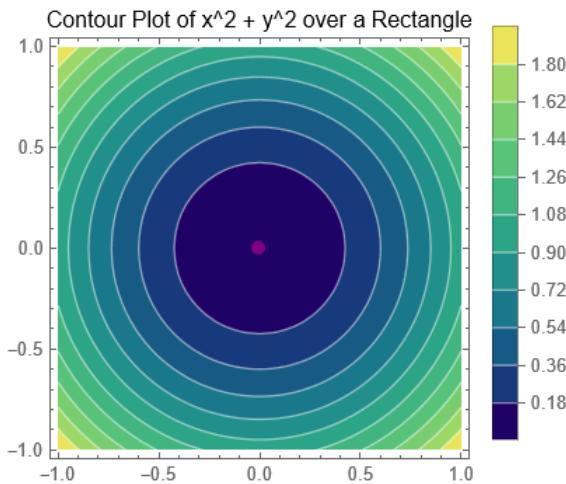
(* Find a minimum of a function over a geometric region: *)
FindMinimum[{x^2+y^2,{x,y}\Element Rectangle[{-1,-1},{1,1}]},{x,y}]

(* Plot it: *)
Show[
(* Contour plot of the function over the specified rectangle: *)
ContourPlot[
x^2+y^2,
{x, -1, 1},
{y, -1, 1},
Contours->10,
ContourStyle->\{White\},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
ImageSize->250,
PlotLabel->"Contour Plot of  $x^2 + y^2$  over a Rectangle"
],
Graphics[
{
Purple,
PointSize[Large],
(* Plot a point at the minimum found in the previous FindMinimum call: *)
Point[{x,y}/. Last[%]]
}
]
]

Output {2.,{x->1.,y->1.}}
Output {0.,{x->0.,y->0.}}

```

Output

**Mathematica Code 3.24**

Input

```
(* The code achieves the following goals: it uses `NMinimize` to find the minimum
value of the function  $x^3 \cos[2x] + y^2 \cos[2y] + 1$  within the geometric region defined
by the rectangle  $[-2, -2]$  to  $[2, 2]$ . It then creates a contour plot of this function
over the specified rectangle, highlighting the minimum point found by `NMinimize`
with a purple point: *)

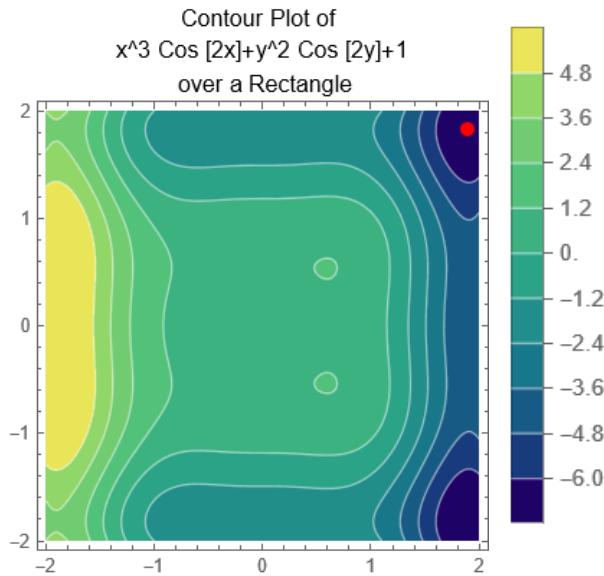
(* Minimize a function over a geometric region: *)
NMinimize[{x^3 Cos[2x] + y^2 Cos[2y] + 1, {x, y} \[Element] Rectangle[{-2, -2}, {2, 2}]}, {x, y}]

(* Plot the function and highlight the minimum point: *)
Show[
  (* Contour plot of the function over the specified rectangle: *)
  ContourPlot[
    x^3 Cos[2x] + y^2 Cos[2y] + 1,
    {x, -2, 2},
    {y, -2, 2},
    Contours -> 10,
    ContourStyle -> {White},
    ClippingStyle -> Automatic,
    ColorFunction -> "BlueGreenYellow",
    PlotLegends -> Automatic,
    ImageSize -> 250,
    PlotLabel -> "Contour Plot of \n x^3 Cos [2x]+y^2 Cos [2y]+1 \n over a Rectangle"
  ],
  Graphics[
    {
      Red,
      PointSize[Large],
      (* Plot a point at the minimum found in the previous NMinimize call: *)
      Point[{x, y} /. Last[%]]
    }
  ]
]
```

Output

```
{-7.33508, {x -> 1.90438, y -> 1.8218}}
```

Output

**Mathematica Code 3.25**

```
Input      (* Use Reap and Sow to track expressions during evaluation: *)
Reap[
  Sow[a];(* Sow the value'a' to be reaped later *)
  b;(* Evaluate'b', but do not sow it *)
  Sow[c];(* Sow the value'c' to be reaped later *)
  Sow[d];(* Sow the value'd' to be reaped later *)
  e        (* Evaluate'e', but do not sow it *)
]

Output    {e,{{a,c,d}}}
```

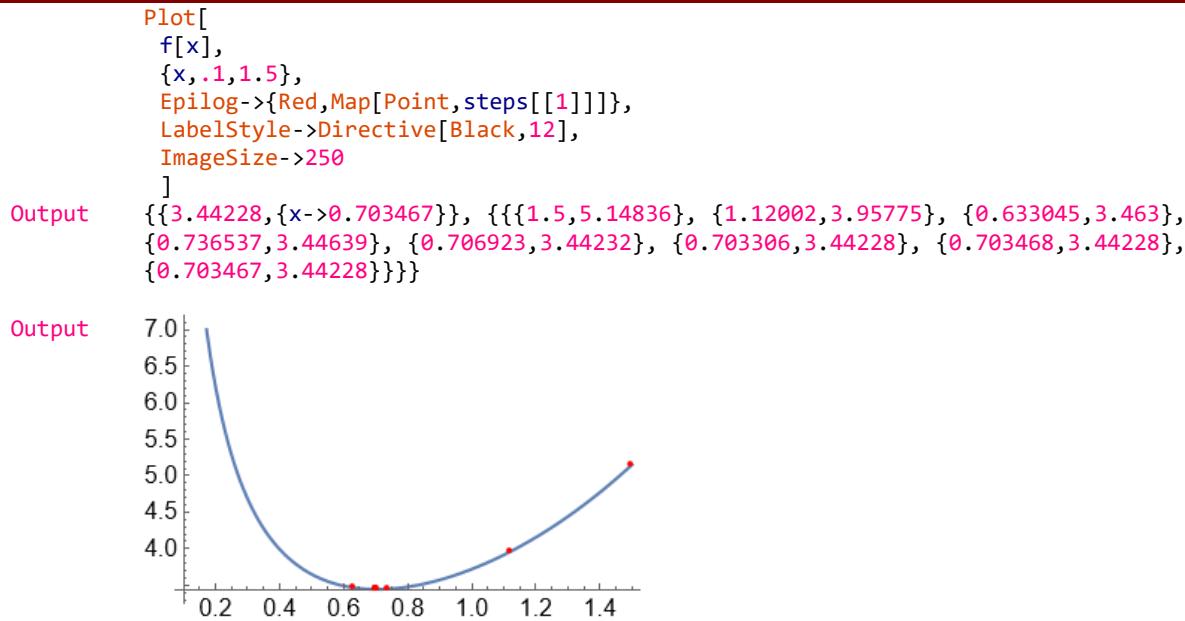
Mathematica Code 3.26

```
Input      (* Compute a sum, "sowing" i^2 at each step: *)
Reap[
  Sum[
    Sow[i^2]+1,
    {i,10}
  ]
]

Output   {395,{{1,4,9,16,25,36,49,64,81,100}}}
```

Mathematica Code 3.27

```
Input      (* Use Reap and Sow to collect step data: *)
f[x_]=Exp[x]+1/x;
{res,steps}=Reap[
  FindMinimum[
    f[x],
    {x,3},
    StepMonitor:>Sow[{x,f[x]}]
  ]
]
(* Show steps on a plot of the function: *)
```

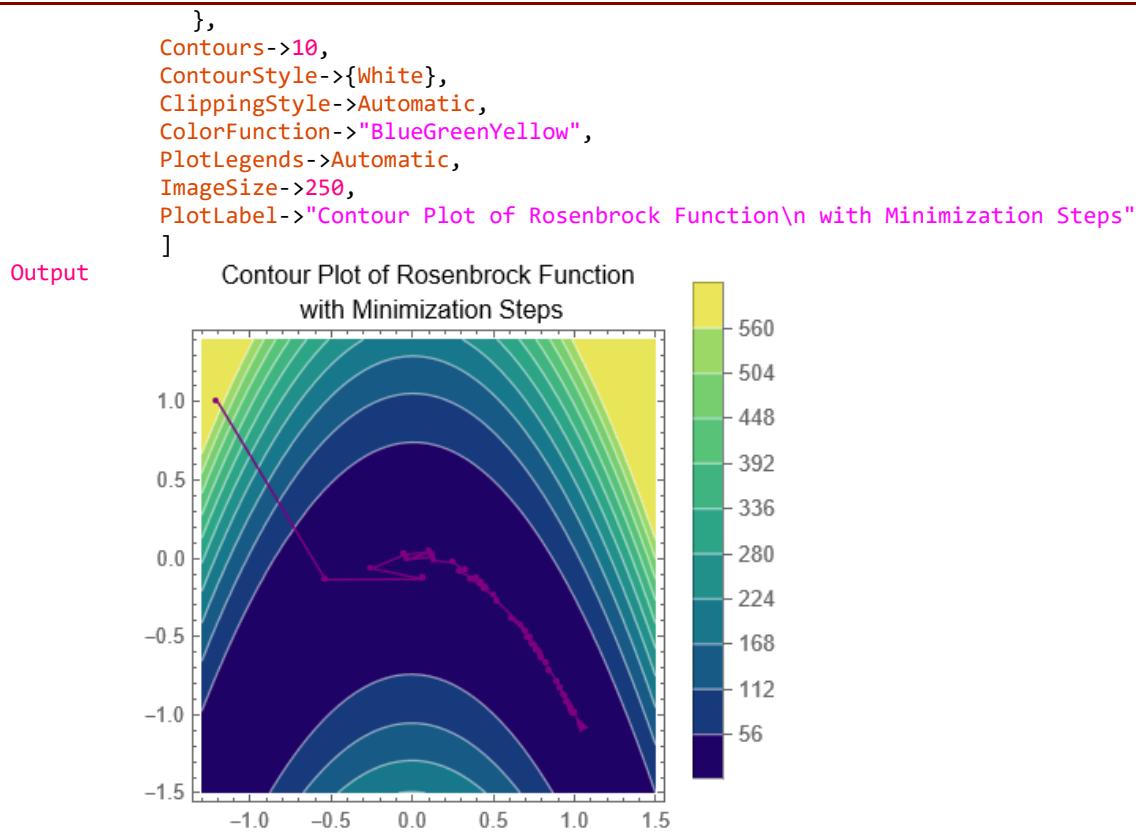
**Mathematica Code 3.28**

```

Input (* The code achieves the following goals: it defines the Rosenbrock function, a well-known test function for optimization algorithms. Using NMinimize with the "NelderMead" method, it finds the minimum of the Rosenbrock function while tracking the steps taken during the optimization process. It includes the initial point in the list of tracked steps. The code then creates a contour plot of the Rosenbrock function, visualizing both the function's landscape and the path taken by the optimization algorithm to reach the minimum, using arrows and points to illustrate each step: *)

(* Steps taken by NMinimize in finding the minimum of the classic Rosenbrock function: *)
(* Define the Rosenbrock function: *)
rosenbrockFunction[x_,y_]:=(1-x)^2+100 (-x^2-y)^2
(* Use NMinimize to find the minimum and track the steps: *)
minimizationSteps=Reap[
  NMinimize[
    rosenbrockFunction[x,y],
    {x,y},
    (*Optimization method*)
    Method->"NelderMead",
    (*Track and store the steps taken by the algorithm*)
    StepMonitor:>Sow[{x,y}]
  ]
][[2,1]];
(* Add the initial point to the list of steps:: *)
minimizationSteps=Join[{{-1.2,1}},minimizationSteps];
(* Create a contour plot of the Rosenbrock function with the steps: *)
ContourPlot[
  rosenbrockFunction[x,y],
  {x,-1.3,1.5},
  {y,-1.5,1.4},
  Epilog->{
    Purple,
    Arrow[minimizationSteps],
    Point[minimizationSteps]
}

```

**Mathematica Code 3.29**

Input

```

(* The code achieves the following goals: it defines a function with a ring of
minima,  $x^2+y^2-16)^2$ , and uses `NMinimize` with the "DifferentialEvolution" method
to find the minimum of this function. During the optimization process, it tracks
all points evaluated by the algorithm. The code then creates a contour plot of the
function over the specified range for x and y, highlighting the points evaluated by
the optimization algorithm that are close in value to the final solution: *)

(* Record all the points evaluated during the solution process of a function with
a ring of minima: *)

(* Define the function with a ring of minima: *)
ringMinimaFunction[x_,y_]:= (x^2+y^2-16)^2

(* Use NMinimize to find the minimum and track the evaluated points: *)
{solution,evaluatedPoints}=Reap[
  NMinimize[
    ringMinimaFunction[x,y],
    {{x,-5,5},{y,-5,5}},
    (*Optimization method*)
    Method->"DifferentialEvolution",
    (*Track and store the points evaluated by the algorithm*)
    EvaluationMonitor:>Sow[{x,y}]
  ]
];

(* Plot all the visited points that are close in objective function value to the
final solution: *)
ContourPlot[

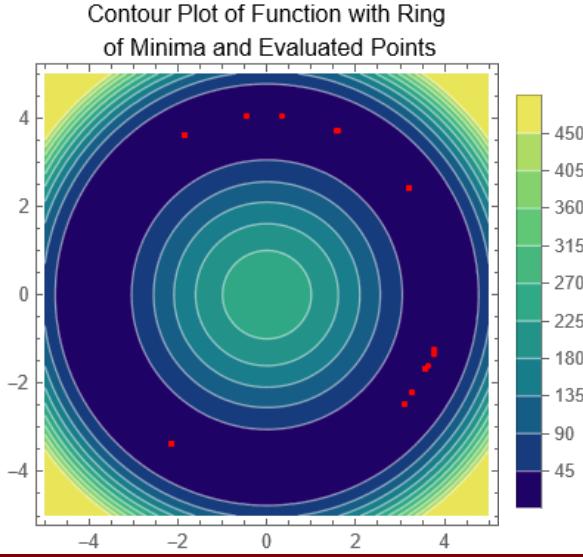
```

```

ringMinimaFunction[x,y],
{x,-5,5},
{y,-5,5},
Epilog->
{
  Red,
  Map[
    (*Plot points at each evaluated point*)
    Point,
    Cases[
      (*Extract the evaluated points*)
      First[evaluatedPoints],
      (*Filter points close to the solution*)
      pt_ /; Abs[ringMinimaFunction@@pt-First[solution]]<=0.05
    ],
    Contours->10,
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    PlotLegends->Automatic,
    ImageSize->250,
    PlotLabel->"Contour Plot of Function with Ring\n of Minima and Evaluated Points"
  ]
}

```

Output

**Mathematica Code 3.30**

```

Input      (* The code achieves the following goals: it defines a target function (1-x)^2+100
           (-x^2-y)^2+1 to minimize and uses `FindMinimum` to find the minimum, tracking each
           step of the optimization process. It includes the initial point in the list of
           tracked steps. The code then creates a contour plot of the logarithm of the target
           function, visualizing the steps taken by the optimization algorithm with lines and
           points: *)
```

```

(* Steps taken by FindMinimum in finding the minimum of a function: *)
```

```

(* Define the function to minimize: *)
targetFunction[x_,y_]:=(1-x)^2+100 (-x^2-y)^2+1
```

```

(* Use FindMinimum to find the minimum and track the steps: *)
```

```

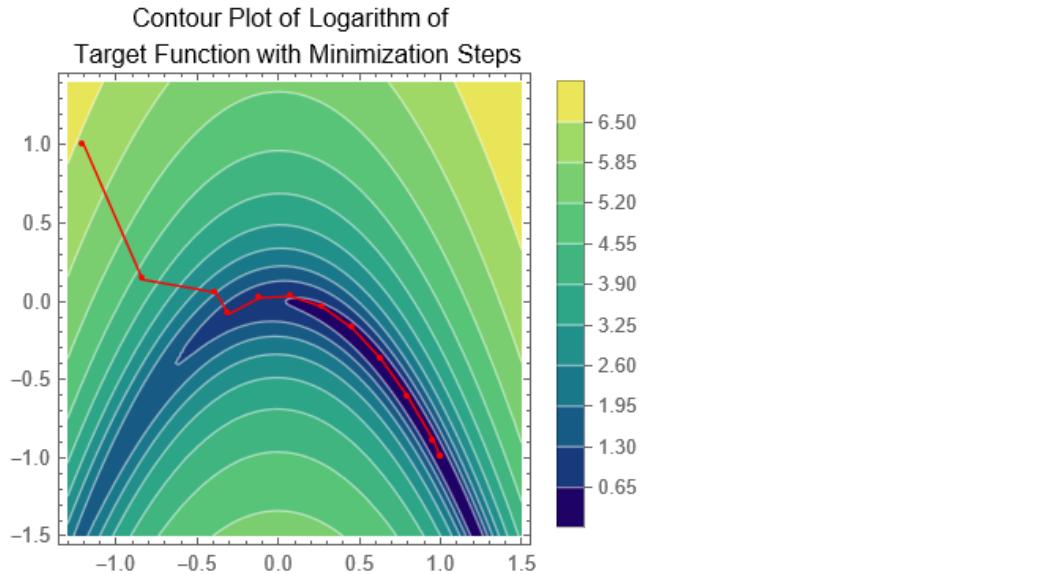
steps=Reap[
  FindMinimum[
    targetFunction[x,y],
    (* Initial guesses for x and y: *)
    {{x,-1.2},{y,1}}},
    (* Track and store the steps taken by the algorithm: *)
    StepMonitor:>Sow[{x,y}]
  ]
][[2,1]];

(* Add the initial point to the list of steps: *)
steps=Join[{{{-1.2,1}}},steps];

(* Create a contour plot of the logarithm of the function with the steps: *)
ContourPlot[
  Log[
    targetFunction[x,y]],
  {x,-1.3,1.5},
  {y,-1.5,1.4},
  Epilog->{
    Red,
    (* Draw a line through the steps taken: *)
    Line[steps],
    (* Plot points at each step: *)
    Point[steps]
  },
  Contours->10,
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  PlotLegends->Automatic,
  ImageSize->250,
  PlotLabel->"Contour Plot of Logarithm of \n Target Function with Minimization
  Steps"
]

```

Output



Mathematica Code 3.31

```

Input (* The code achieves the following goals: it defines a target function  $(1-x)^2+100(-x^2-y)^2+1$  to minimize. It then defines a function `minimizeAndPlot` that performs minimization using a specified optimization method, tracks the steps taken during the optimization process, and creates a contour plot of the logarithm of the target function with the steps highlighted. Finally, it calls the `minimizeAndPlot` function for each of the specified optimization methods ("Newton", "QuasiNewton", "ConjugateGradient", "PrincipalAxis", "LevenbergMarquardt", "InteriorPoint"), generating contour plots that visualize the optimization path for each method, providing insights into the behavior and efficiency of each optimization technique: *)

(* Define the function to minimize: *)
targetFunction[x_,y_]:= (1-x)^2+100 (-x^2-y)^2+1

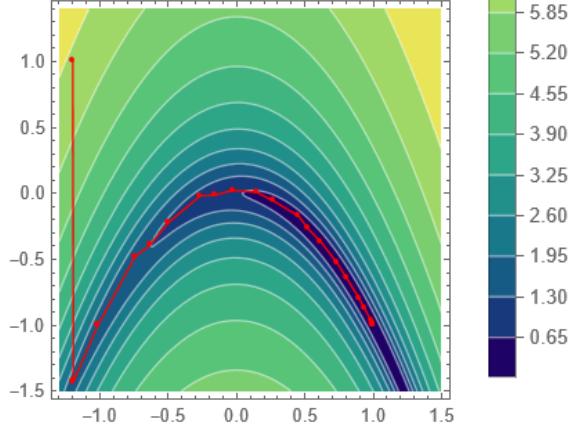
(* Define a function to perform minimization and plot the results: *)
minimizeAndPlot[method_]:=Module[
  {minimizationSteps,methodName},
  methodName=ToString[method];
  (* Use FindMinimum to find the minimum and track the steps: *)
  minimizationSteps=Reap[
    FindMinimum[
      targetFunction[x,y],
      (* Initial guesses for x and y: *)
      {{x,-1.2},{y,1}},
      (* Optimization method: *)
      Method->method,
      (* Track and store the steps taken by the algorithm: *)
      StepMonitor:>Sow[{x,y}]
    ]
  ][[2,1]];
  (* Add the initial point to the list of steps: *)
  minimizationSteps=Join[{{-1.2,1}},minimizationSteps];
  (* Create a contour plot of the logarithm of the function with the steps: *)
  ContourPlot[
    Log[targetFunction[x,y]],
    {x,-1.3,1.5},
    {y,-1.5,1.4},
    Epilog->{
      Red,
      (* Draw a line through the steps taken: *)
      Line[minimizationSteps],
      (* Plot points at each step: *)
      Point[minimizationSteps]
    },
    Contours->10,
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    PlotLegends->Automatic,
    ImageSize->250,
    PlotLabel->"Contour Plot of Logarithm of Target Function\n with Minimization
    Steps\nMethod: "<>methodName
  ]
]
]

(* Perform minimization and plotting for each method: *)
minimizeAndPlot["Newton"]
minimizeAndPlot["QuasiNewton"]
minimizeAndPlot["ConjugateGradient"]
minimizeAndPlot["PrincipalAxis"]
minimizeAndPlot["LevenbergMarquardt"]
minimizeAndPlot["InteriorPoint"]

```

Output

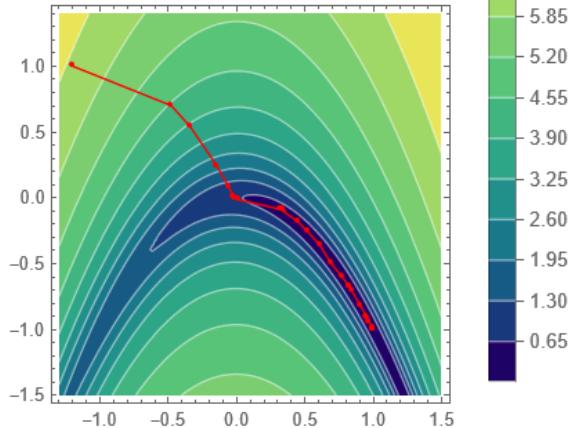
Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: Newton



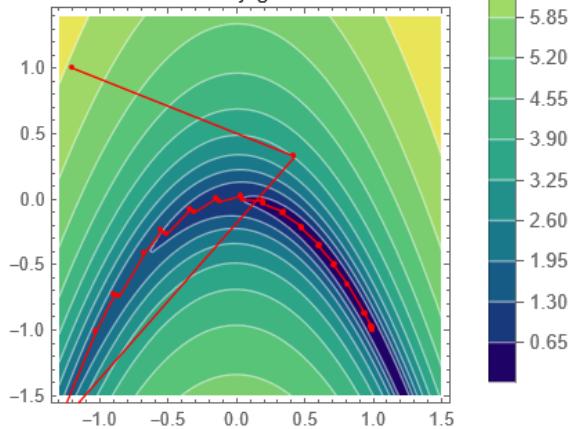
FindMinimum::lstol: The line search decreased the step size to within the tolerance specified by `AccuracyGoal` and `PrecisionGoal` but was unable to find a sufficient decrease in the function. You may need more than `MachinePrecision` digits of working precision to meet these tolerances.

Output

Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: QuasiNewton

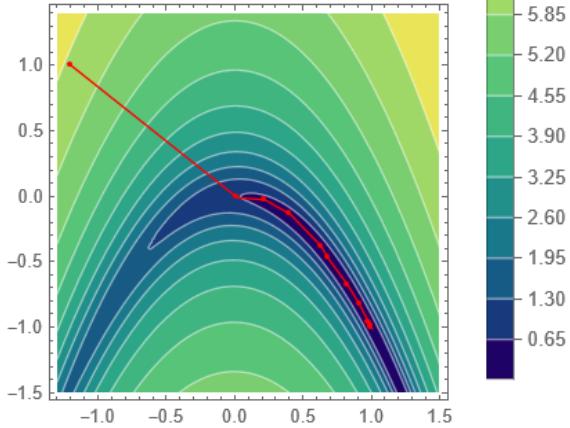
**Output**

Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: ConjugateGradient

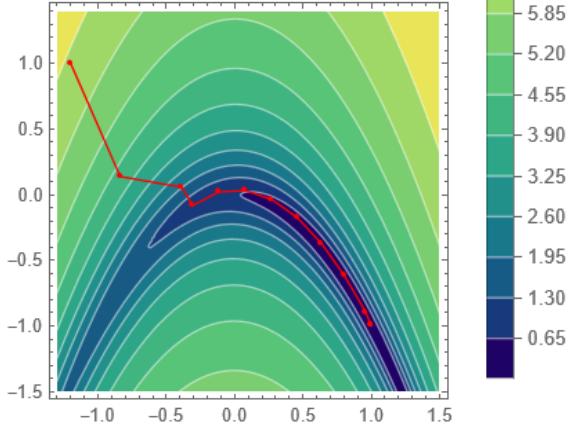


Output

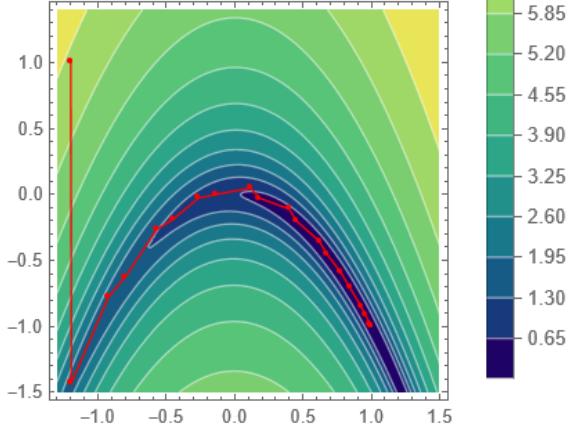
Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: PrincipalAxis

**Output**

Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: LevenbergMarquardt

**Output**

Contour Plot of Logarithm of Target Function
with Minimization Steps
Method: InteriorPoint

***Mathematica Code 3.32*****Input**

(* The code achieves the following goals: it defines the Rosenbrock function, specifies an initial point for the minimization process, and lists several

```

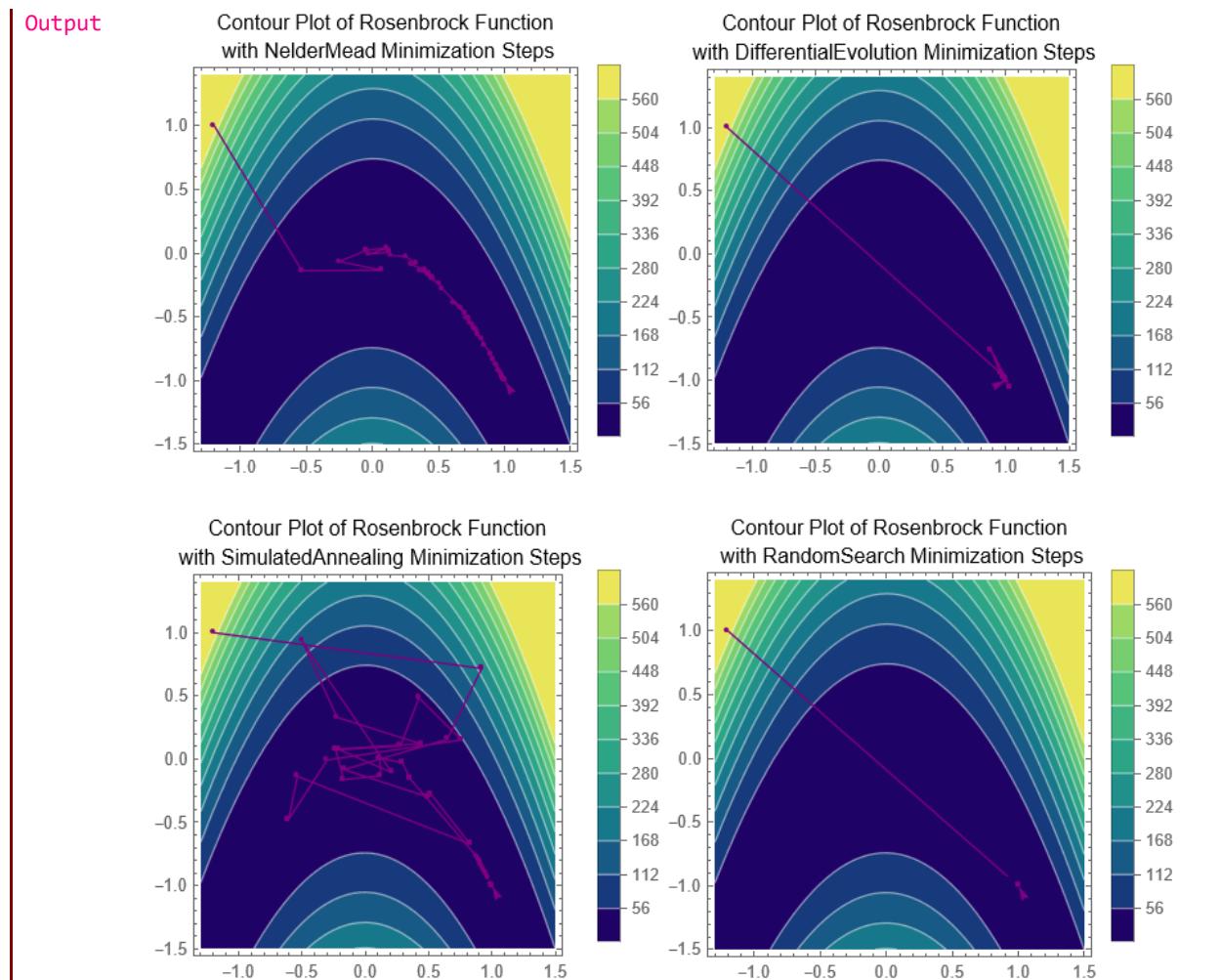
optimization methods ("NelderMead", "DifferentialEvolution", "SimulatedAnnealing",
"RandomSearch"). For each optimization method, it uses NMinimize to find the minimum
of the Rosenbrock function, tracking the steps taken during the optimization process.
It then creates a contour plot of the function with the steps highlighted using
arrows and points. Finally, it arranges the generated plots in a grid layout for
easy comparison of the optimization methods and their respective paths, providing
insights into the behavior and efficiency of each optimization technique: *)

(* Define the Rosenbrock function: *)
rosenbrockFunction[x_,y_]:= (1-x)^2+100 (-x^2-y)^2
(* Initial point for the minimization process: *)
initialPoint={-1.2,1};

(* List of optimization methods to be used: *)
methods={"NelderMead","DifferentialEvolution","SimulatedAnnealing","RandomSearch"} ;
(* Generate plots for each optimization method: *)
plots=Table[
  Module[
    {minimizationSteps},
    (* Use NMinimize to find the minimum and track the steps: *)
    minimizationSteps=Reap[
      NMinimize[
        rosenbrockFunction[x,y],
        {x,y},
        (* Optimization method: *)
        Method->method,
        (* Track and store the steps taken by the algorithm: *)
        StepMonitor:>Sow[{x,y}]
      ]
    ][[2,1]];
    (* Add the initial point to the list of steps: *)
    minimizationSteps=Join[{initialPoint},minimizationSteps];
    (* Create a contour plot of the function with the steps: *)
    ContourPlot[
      rosenbrockFunction[x,y],
      {x,-1.3,1.5},
      {y,-1.5,1.4},
      Epilog->{
        Purple,
        (* Draw arrows indicating the steps taken: *)
        Arrow[minimizationSteps],
        (* Plot points at each step: *)
        Point[minimizationSteps]
      },
      Contours->10,
      ContourStyle->{White},
      ClippingStyle->Automatic,
      ColorFunction->"BlueGreenYellow",
      PlotLegends->Automatic,
      ImageSize->250,
      PlotLabel->"Contour Plot of Rosenbrock Function\n with "<>method<>""
    Minimization Steps"
    ]
  ],
  {method,methods}
];

(* Arrange the plots in a grid: *)
Grid[Partition[plots,2]]

```

**Mathematica Code 3.33**

Input

```

(* The code achieves the following goals: it defines a function f(x,y)= cos(x^2-3y)+ sin(x^2+y^2) to be minimized. It then generates a 3D surface plot and a contour plot of this function, providing visual representations of the function's landscape and level curves. The code uses `FindMinimum` to find the minimum of the function employing four different optimization methods: Newton, Quasi-Newton, Conjugate Gradient, and Principal Axis. Finally, it plots the convergence paths for each of these methods, illustrating the steps taken by each optimization technique to reach the minimum: *)

<<Optimization`UnconstrainedProblems` 

(* Define the function f(x,y): *)
f[x_,y_]:=Cos[x^2-3 y]+Sin[x^2+y^2];

(* Plot the 3D surface of f(x,y): *)
Plot3D[
  f[x,y],
  {x,-3,3},
  {y,-3,3},
  ColorFunction->"Rainbow",
  PlotLegends->BarLegend[Automatic],
  AxesLabel->{"x","y","f(x, y)"},
```

```

PlotLabel->"3D Surface Plot of f(x, y)" ,
ImageSize->250
]

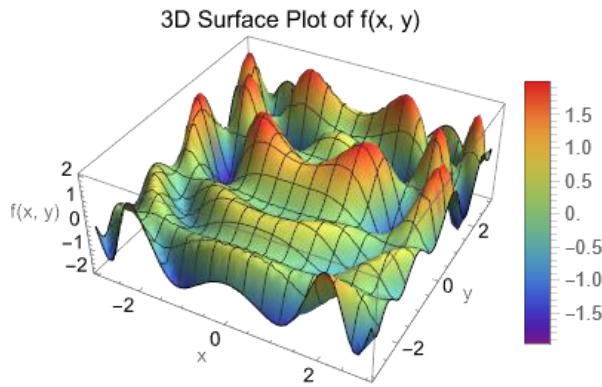
(* Plot the contour of f(x,y): *)
ContourPlot[
f[x,y],
{x,-3,3},
{y,-3,3},
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
ImageSize->250,
Contours->10,
AxesLabel->{"x","y"},
PlotLabel->"Contour Plot of f(x, y)"
]

(* Find the minimum of f(x,y) using Newton, QuasiNewton, ConjugateGradient, and
PrincipalAxis methods: *)
minNewton=Table[
FindMinimum[
f[x,y],(*Function to minimize*)
{{x,1},{y,1}},(*Initial guess for variables*)
Method->method (*Optimization method*)
],
{method,{"Newton","QuasiNewton","ConjugateGradient","PrincipalAxis"}}
]

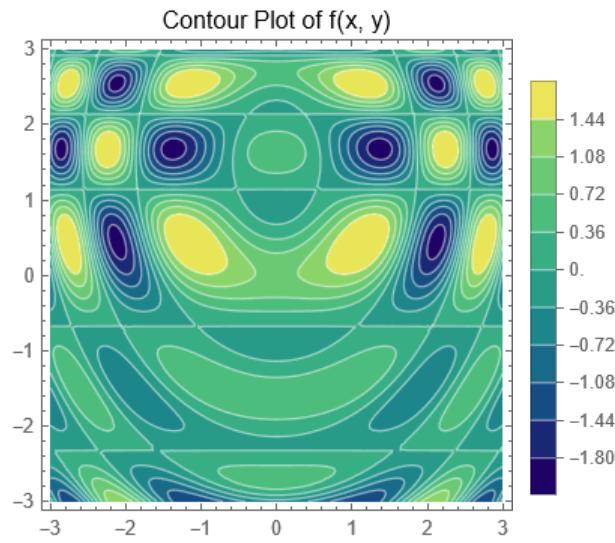
(* Plot the convergence of FindMinimum using Newton, QuasiNewton, ConjugateGradient,
and PrincipalAxis methods: *)
Table[
FindMinimumPlot[
f[x,y],(*Function to minimize*)
{{x,1},{y,1}},(*Initial guess for variables*)
Method->method[[2]],(*Optimization method*)
PlotLabel->Style[Row[{ "Convergence Plot\n",method[[1]]}]],
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
ImageSize->250,
Contours->10,
ImageSize->250
]
,
{method,{
{"(Newton's Method)","Newton"},
 {"(Quasi-Newton Method)","QuasiNewton"},
 {"(Conjugate Gradient Method)","ConjugateGradient"},
 {"(Principal Axis Method)","PrincipalAxis"}}
}
]
]

```

Output



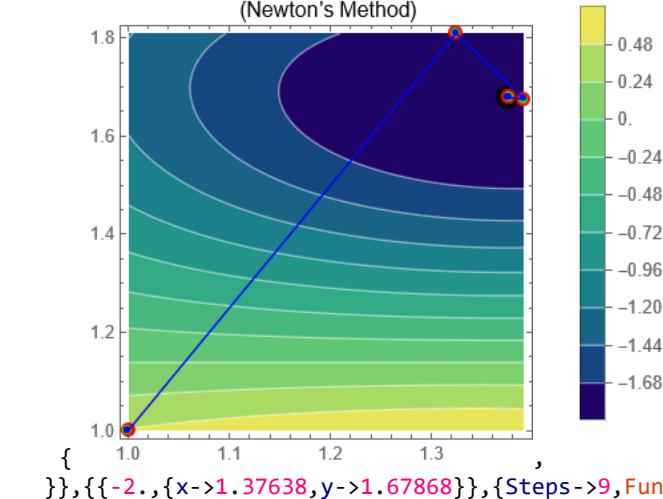
Output



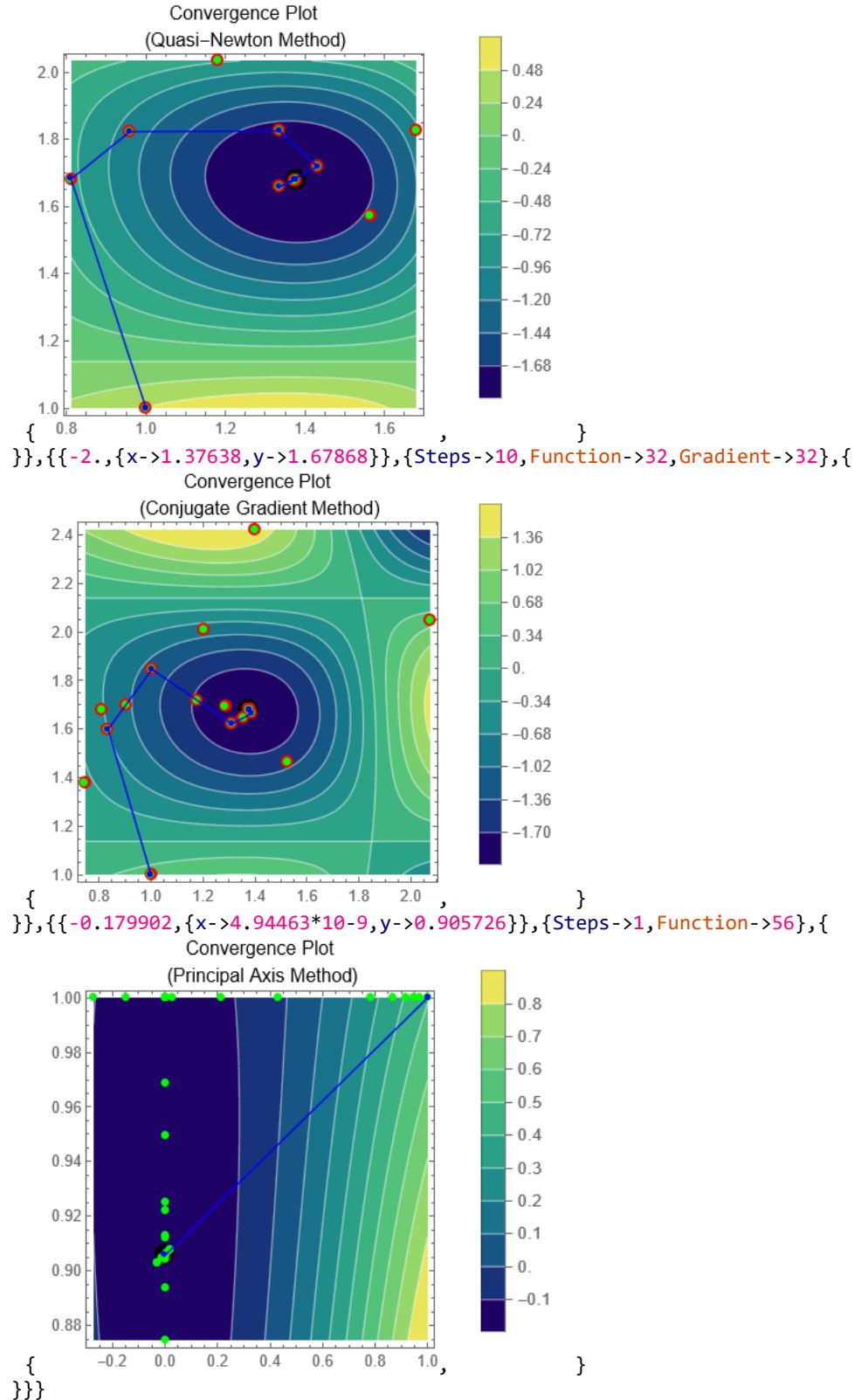
```
Output {{-2.,{x->1.37638,y->1.67868}},{-2.,{x->1.37638,y->1.67868}},{-2.,{x->1.37638,y->1.67868}},{-0.179902,{x->4.94463*10^-9,y->0.905726}}}}
```

Output

```
{{{ -2.,{x->1.37638,y->1.67868}},{Steps->5,Function->6,Gradient->6},{{Convergence Plot  
(Newton's Method)}}
```



```
},{{-2.,{x->1.37638,y->1.67868}},{Steps->9,Function->13,Gradient->13},{{}}},{{-2.,{x->1.37638,y->1.67868}},{Steps->9,Function->13,Gradient->13},{{}}}}
```



CHAPTER 4

MULTILAYER FEED-FORWARD NEURAL NETWORK

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 3, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation.

In the rapidly evolving landscape of AI and machine learning [24-39], Feed-Forward Neural Networks (FFNNs) stand out as foundational structures, powering a multitude of applications across various domains. This chapter serves as a comprehensive introduction to the inner workings of FFNNs, delving into essential concepts and mechanisms crucial for understanding their functionality and effectiveness. We will start by looking at the structure of a FFNN, followed by how they are trained and used for making predictions. We will also take a brief look at the loss functions that should be used in different settings, the Activation Functions (AFs) used within a neuron, and the different types of optimizers that could be used for training.

The training process is a crucial phase in the development of FFNNs, enabling them to learn from data and improve their performance over time. The training procedure can be broken down into two main components, each playing a distinct role in the network's learning process (forward propagation and back propagation).

We begin with an exploration of forward propagation in NNs, elucidating how input data traverse through the network's layers to produce output predictions. Through a step-by-step examination, readers will grasp the fundamental principles underlying the propagation of information within these intricate systems.

A pivotal aspect of NN training is automatic differentiation and its main modes [40-43]. Automatic differentiation illustrates the mechanisms through which gradients are computed efficiently, enabling the network to adapt and optimize its parameters during the learning process. We further explore the training process and loss/cost functions integral to optimizing NNs. From defining objectives through appropriate loss functions to navigating the landscape of optimization algorithms.

We shift our focus to the backward pass, often referred to as Back Propagation (BP). During this phase, the network evaluates the error or the disparity between its predictions (outputs) and the actual target values. This discrepancy serves as a guide for adjusting the network's weights to minimize the error and enhance its accuracy. BP involves traversing the network in reverse, updating weights based on the calculated error and its gradients.

Moreover, we explore the Universal Approximation Theorem (UAT) [44-46], which asserts that a FFNN with a single hidden layer can approximate any continuous function on a compact subset of \mathbb{R}^n . This theorem underscores the remarkable flexibility and potential of NNs in modeling complex, non-linear relationships.

In this chapter, we focus on the practical aspects of constructing NNs using Mathematica. We explore a variety of essential components and techniques for building NNs, ranging from fundamental layers to complex architectures.

- First, we will delve into the intricacies of constructing NNs from scratch using Mathematica. By exploring the foundational elements and step-by-step processes, you'll gain a comprehensive understanding of NN architecture, training methodologies, and performance optimization.
- Next, we begin by understanding the foundational building block of NNs, the **LinearLayer**. This layer performs linear transformations on the input data, playing a crucial role in connecting different layers of the network. The key components of a **LinearLayer** are the weights and biases. During the training process, these parameters are adjusted iteratively using optimization algorithms such as gradient descent, enabling the

network to learn from the data. The number of input features and the number of neurons in the `LinearLayer` determine the dimensions of the weight matrix.

- As we progress, we introduce the `ElementwiseLayer`, which enables element-wise operations such as ReLU AF or sigmoid activation across the network's neurons. This layer adds non-linearity to the network, allowing it to learn complex patterns in the data. Mathematica provides options for customizing `ElementwiseLayers`, such as choosing different AFs, or defining custom AFs, allowing practitioners to tailor the behavior of `ElementwiseLayers` according to the specific requirements of their tasks.
- We explore two essential constructs for organizing NN architectures: `NetChain` and `NetGraph`. `NetChain` enables the sequential composition of layers, where the output of one layer serves as the input to the next layer. This sequential arrangement facilitates the construction of straightforward feedforward NN architectures, where data flows from input to output through a series of transformations. Creating a `NetChain` in Mathematica involves specifying the layers and their configurations sequentially.
- `NetGraph` is a powerful construct in Mathematica for building NN architectures that involve non-sequential or more complex connectivity patterns. Unlike `NetChain`, which represents a linear sequence of layers, `NetGraph` allows for the creation of arbitrary computational graphs, enabling the design of intricate NN structures. `NetGraph` enables the creation of NNs with arbitrary connectivity patterns, where layers can be connected in any configuration, including branching, merging, looping, and skip connections. This flexibility allows for the construction of sophisticated architectures tailored to specific tasks. NN architectures built using `NetGraph` are represented as directed graphs, where nodes represent layers or operations, and edges represent the flow of data between them.
- A crucial step in NN construction is initializing the network's parameters. We discuss `NetInitialize`, a function in Mathematica that initializes these parameters, setting the stage for efficient training. Mathematica provides various initialization methods that can be specified through options in `NetInitialize`. These methods include `"Random"`, `"Xavier"`, `"He"`, and custom initialization functions.
- To quantify the performance of our NN models, we introduce loss functions. `MeanSquaredLossLayer` and `MeanAbsoluteLossLayer` are commonly used for regression tasks, measuring the difference between predicted and actual values. For classification problems, we explore the `CrossEntropyLossLayer`, a widely used loss function that measures the dissimilarity between predicted and actual class distributions.
- Finally, we conclude by discussing `NetTrain`, an essential function in Mathematica for training NNs. Under the hood, `NetTrain` employs gradient descent optimization algorithms, such as `"SGD"`, `"ADAM"` or `"RMSProp"` and `"SignSGD"`, to update the parameters of the NN iteratively.

Throughout this chapter, we provide insights and practical examples to empower readers in harnessing the capabilities of Mathematica for constructing, training, and fine-tuning NNs for various machine learning tasks.

Unit 4.1

Building Neural Network from Scratch with Mathematica and Universal Approximation Theorem

In this unit, let us leverage Mathematica's powerful features to implement a NN from scratch, train it on a regression task, perform forward and backward propagation, and visualize the training process and results. Implementing a NN from scratch is an invaluable exercise for deepening your understanding of how NNs work. By building a NN from the ground up, you gain insights into the inner workings of the algorithms involved, including forward and backward propagation, gradient descent optimization, weight initialization, AFs, and more. Building a NN from scratch helps you develop strong debugging skills. When you encounter errors or unexpected behavior, you'll learn to diagnose and fix issues by tracing through the code and understanding how each component interacts.

Mathematica code 4.1 serves as an educational tool to implement the theoretical equations involved in training a NN for regression tasks. It is implementing a NN to approximate the function $f(x, y) = x^2 + y^2$. By implementing these theoretical equations in a practical coding environment, the code enables you to gain hands-on experience with training NNs.

The code demonstrates how to perform forward propagation through the NN layers. It illustrates the calculation of pre-activation and post-activation values for each layer, showcasing how information flows through the network. Backpropagation, which involves computing gradients of the loss function with respect to the weights and biases of each layer, is a fundamental aspect of training NNs. The code meticulously calculates these gradients, enabling you to understand the mathematics behind backpropagation.

Below are the main steps of the code:

1. Generate a set of training data points by evaluating the function $f(x, y)$ over a range of x and y values.
2. Define the architecture of the NN with three layers: an input layer with two neurons (for x and y), two hidden layers with 10 neurons each, and an output layer with one neuron.
3. Initialize weights and biases randomly for each layer of the NN.
4. Define the hyperbolic tangent (\tanh) AF and its derivative for use in the NN.
5. Implement forward propagation to compute the outputs of each layer in the NN for a given set of input data.
6. Plot 3D plots for the training data and the output of the NN before training. Also, create histograms to visualize the distributions of outputs from each layer of the NN before training.
7. Define the mean squared error (MSE) loss function to quantify the difference between the predicted outputs of the NN and the actual outputs.
8. Set the learning rate and the number of epochs for training the NN.
9. Iterate through the training data for the specified number of epochs. In each epoch:
 - Perform forward propagation to compute the outputs of each layer.
 - Calculate the loss using the defined loss function.
 - Perform backpropagation to compute gradients of the loss with respect to the weights and biases of each layer.
 - After computing gradients, the weights and biases of each layer are updated using gradient descent to minimize the loss function.
 - The training loop continues until a stopping criterion is met. In this case, the loop stops either when the specified number of epochs is reached or when the loss falls below a certain threshold (0.0001 in this code).

10. The training progress is monitored by recording the loss at each epoch. This information is used to create a plot showing how the loss decreases over epochs, providing insight into the training performance.
11. Plot 3D plots for the training data and the output of the NN after training. Also, create histograms to visualize the distributions of outputs from each layer of the NN after training.
12. The code utilizes matrix-matrix multiplication for both forward and backward propagation. In forward propagation, it performs the matrix multiplication operation between the input matrices, typically representing the input features and weights of a neural network layer, to produce the output activations. These activations are then passed through AFs to compute the final output of the layer.
13. During backward propagation, the code computes the gradients of the loss function with respect to the input matrices using the chain rule of calculus. It propagates the gradients backwards, adjusting the weights of the network based on the computed gradients to minimize the loss.

While the code implements a basic FFNN and performs adequately for this simple function approximation task, there are several potential improvements that could be made, such as experimenting with different network architectures, AFs, learning rates, and optimization algorithms to improve training performance and convergence speed. Next chapters will provide an in-depth exploration of these subjects.

In Mathematica, Mathematica code 4.1 can be significantly condensed using Mathematica's framework, reducing it to just a few lines, see Mathematica code 4.2. [Units 4.2-4.6](#) will delve into the extensive capabilities of Mathematica's NN framework, which offers a rich array of functions and tools for the precise definition, efficient training, and seamless deployment of NNs. The Wolfram Language offers advanced capabilities for the representation, construction, training and deployment of NNs. A large variety of layer types is available for symbolic composition and manipulation. Thanks to dedicated encoders and decoders, diverse data types such as image, text and audio can be used as input and output, deepening the integration with the rest of the Wolfram Language. It combines ease of use with advanced capabilities, making it suitable for both beginners and experts in deep learning.

Mathematica Code 4.1

```

Input (* The code demonstrates the training of a neural network to approximate the function
f(x,y)=x^2+y^2. It begins by generating a dataset through the evaluation of this
function across a specified range of x and y values. Following this, a neural network
architecture is defined, comprising an input layer with 2 neurons, two hidden layers
each with 10 neurons, and an output layer with 1 neuron. Initialization of network
parameters involves randomly assigning weights and biases from a normal
distribution. Through forward propagation, the network computes outputs for each
layer based on the input data, while backpropagation facilitates gradient
computation for parameter updates. The training loop iterates through multiple
epochs, refining the network parameters via gradient descent to minimize the Mean
Squared Error loss between predicted and actual outputs. Visualization techniques
are employed to depict training progress and evaluate the network's performance pre-
and post-training through 3D plots and histograms: *)

(* Generate training data: *)

(* Define the function f(x,y): *)
f[x_,y_]:=x^2+y^2;

(* Generate x and y values for training data: *)
xValues=Table[x,{x,-1,1,0.01}];
yValues=Table[y,{y,-1,1,0.01}];

(* Create training data by evaluating the function for all combinations of x and
y:*)
(* Dim of trainingData={m,3} where m is the number of training examples: *)
trainingData=Flatten[

```

```

Table[
{x,y,f[x,y]},
{x,xValues},
{y,yValues}
],1];

(* The Length of trainingData: *)
m=Length[trainingData];

(* Separate input and output: *)
(* Row 1 and 2 are x and y values (Dim of inputdata={2,m}): *)
inputData=Transpose[trainingData[[All,1;;2]]];

(* f(x,y) values (Dim of outputdata={1,m}): *)
outputData={trainingData[[All,3]]};

(* Neural network architecture:*)
numL=3; (* Number of layers. *)
n0=2;(* Number of neurons in input layer. *)
n1=10;(* Number of neurons in hidden layer 1. *)
n2=10;(* Number of neurons in hidden layer 2. *)
n3=1;(* Number of neurons in output layer. *)

(* Define the neural network parameters:*)

(* Weights and biases for the first layer: *)

SeedRandom[123];
(* Dim of weights1={n1,n0}: *)
weights1=RandomVariate[NormalDistribution[0,1],{n1,n0}] ;
(* Dim of biases1={n1,1}: *)
biases1=Transpose[{RandomVariate[NormalDistribution[0,1],n1]}];
(* Dim of biasesmatrix1={n1,m}: *)
biasesmatrix1=Transpose[Table[Flatten[biases1],m]];

(* Weights and biases for the second layer: *)

(* Dim of weights2={n2,n1}: *)
weights2=RandomVariate[NormalDistribution[0,1],{n2,n1}] ;
(* Dim of biases2={n2,1}: *)
biases2=Transpose[{RandomVariate[NormalDistribution[0,1],n2]}];
(* Dim of biasesmatrix2={n2,m}: *)
biasesmatrix2=Transpose[Table[Flatten[biases2],m]];

(* Weights and biases for the output layer: *)

(* Dim of weights3={n3,n2}: *)
weights3=RandomVariate[NormalDistribution[0,1],{n3,n2}] ;
(* Dim of biases3={n3,1}: *)
biases3=Transpose[{RandomVariate[NormalDistribution[0,1],n3]}];
(* Dim of biasesmatrix3={n3,m}: *)
biasesmatrix3=Transpose[Table[Flatten[biases3],m]];

(* Define the Tanh activation function: *)
tanhActivation[x_]:=Tanh[x]

(* Define the derivative of the tanh activation function: *)
tanhDerivative[x_]:=1-Tanh[x]^2

```

```

(* Define the vectorized (matrix) forward propagation function for all training
examples: *)
forwardPropagation[inputs_]:=Module[
{preActivlayer1,postActivlayer1,preActivlayer2,postActivlayer2,preActivlayer3,post
Activlayer3},

(* First layer (Dim of preActivlayer1 and postActivlayer1 ={n1,m}): *)
preActivlayer1=(weights1.inputs+biasesmatrix1);
postActivlayer1=Map[tanhActivation,preActivlayer1];

(* Second layer (Dim of preActivlayer2 and postActivlayer2 ={n2,m}): *)
preActivlayer2=(weights2.postActivlayer1+biasesmatrix2);
postActivlayer2=Map[tanhActivation,preActivlayer2];

(* Output layer (Dim of preActivlayer3 and postActivlayer3 ={n3,m}): *)
preActivlayer3=(weights3.postActivlayer2+biasesmatrix3);
(* For regression model, we use identity activation function in output layer
 $\sigma(z_3)=z_3$ : *)
postActivlayer3=preActivlayer3;

(* Return the final output and intermediate layer outputs: *)

{preActivlayer1,postActivlayer1,preActivlayer2,postActivlayer2,preActivlayer3,post
Activlayer3}
];

(* Perform forward propagation for all training examples: *)
{preActivlayer1,layer1Output,preActivlayer2,layer2Output,preActivlayer3,layer3Output}=forwardPropagation[inputData];

(* Create the neural network output data without training (Dim of NNOutPutData
={m,3}): *)
NNOutPutData=Transpose[Join[inputData,layer3Output]];

(* Create 3Dplot for the trainingData and NN output data: *)
Table[
ListPlot3D[
data[[2]],
PlotLabel->Style[Row[{ "3D plot for ",data[[1]]}]],
ColorFunction->"Rainbow",
ImageSize->220
],
{data,{{"training Data \n before training",trainingData}, {"NN output data \n
before training",NNOutPutData}}}
]

(* Create histograms for first, second and output layers before training: *)
Table[
Histogram[
Flatten[layer[[2]]],
Automatic,
"Probability",
PlotLabel->Style[Row[{ "Histogram of ",layer[[1]]}]],
FrameLabel->{"Output","Probability"},
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250
],
{layer,

```

```

{
  {"the first Hidden Layer \n Output before training",layer1Output},
  {"the second Hidden Layer \n Output before training",layer2Output},
  {"Output Layer \n before training",layer3Output}
}
]

(* Define the Mean Squared Error (MSE) loss function: *)
mseLoss[yActual_,yPredicted_]:=Mean[Flatten[(yPredicted-yActual)^2]]

(* Training parameters: *)
learningRate=0.1;
numEpochs=1000;

(* Training loop: *)

Do[
  (* Forward propagation: *)

{preActivlayer1,layer1Output,preActivlayer2,layer2Output,preActivlayer3,layer3Output}=forwardPropagation[inputData];

  (* Calculate loss: *)
  currentLoss=mseLoss[outputData,layer3Output];

  (* Backpropagation: *)

  (* Calculate gradients for the output layer: *)

  (* delta3 for all training examples: *)
  delta3=(layer3Output-outputData);(* Dim of delta3 ={n3,m}: *)
  (* Dim of gradWeights3 ={n3,m}*{m,n2}={n3,n2}. gradWeights3 is average for all
  training examples: *)
  gradWeights3=(1/m)*delta3.Transpose[layer2Output];
  (* Dim of gradBiases3 ={1,1}. gradBiases3 is average for all training examples: *)
  gradBiases3=(1/m)*{{Total[Flatten[delta3]]}};

  (* Calculate gradients for the second hidden layer: *)

  (* delta2 for all training examples: *)
  delta2=(Transpose[weights3].delta3)*Map[tanhDerivative,preActivlayer2];(* Dim
  of delta2 ={n2,m}: *)
  (* Dim of gradWeights2 ={n2,m}*{m,n1}={n2,n1}. gradWeights3 is average for all
  training examples: *)
  gradWeights2=(1/m)*delta2.Transpose[layer1Output];
  (* Dim of gradBiases2 ={n2,1}. gradBiases3 is average for all training examples: *)
  gradBiases2=(1/m)*Transpose[{Total[Transpose[delta2]]}];

  (* Calculate gradients for the first hidden layer: *)
  delta1=Transpose[weights2].delta2*Map[tanhDerivative,preActivlayer1];(* Dim of
  delta1 ={n1,m}: *)
  (* Dim of gradWeights1 ={n1,m}*{m,n0}={n1,n0}. gradWeights3 is average for all
  training examples: *)
  gradWeights1=(1/m)*delta1.Transpose[inputData];
  (* Dim of gradBiases1 ={n1,1}. gradBiases3 is average for all training
  examples: *)
  gradBiases1=(1/m)*Transpose[{Total[Transpose[delta1]]}];
}

```

```

(* Update weights and biases for the output layer:*)
weights3=weights3-learningRate*gradWeights3;
biases3=biases3-learningRate*gradBiases3;
biasesmatrix3=Transpose[Table[Flatten[biases3],m]];
(* Update weights and biases for the second hidden layer: *)
weights2=weights2-learningRate*gradWeights2;
biases2=biases2-learningRate*gradBiases2;
biasesmatrix2=Transpose[Table[Flatten[biases2],m]];
(* Update weights and biases for the first hidden layer: *)
weights1=weights1-learningRate*gradWeights1;
biases1=biases1-learningRate*gradBiases1;
biasesmatrix1=Transpose[Table[Flatten[biases1],m]];

lastEpoch=epoch;
trainingprogress[epoch]={epoch,currentLoss};
If[currentLoss<0.0001,Break[],,
{epoch,numEpochs}
];

ListPlot[
Table[trainingprogress[i],{i,1,lastEpoch}],
PlotLabel->"Training Progress",
AxesLabel->{"Epoch","Loss"},
Joined->True,
ColorFunction->"Rainbow",
ImageSize->220
]

(* Perform forward propagation for all training examples after training: *)
{preActivlayer1n,layer1Outputn,preActivlayer2n,layer2Outputn,preActivlayer3n,layer3Outputn}=forwardPropagation[inputData];

(* Create the neural network output data after training: *)
NNOutPutDatan=Transpose[Join[inputData,layer3Outputn]];

(* Create 3Dplot for the trainingData and the output layer after training: *)
Table[
ListPlot3D[
data[[2]],
PlotLabel->Style[Row[{ "3D plot for the ",data[[1]]}]],
ColorFunction->"Rainbow",
ImageSize->220
],
{data,{ "training Data \n after training",trainingData},
 {"output layer after training \n after training",NNOutPutDatan}
 }
]

(* Create histograms of first, second and output layers after training: *)
Table[
Histogram[
Flatten[layer[[2]]],
Automatic,
"Probability",
PlotLabel->Style[Row[{ "Histogram of the ",layer[[1]]}]],
FrameLabel->{"Output","Probability"},
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
]
]

```

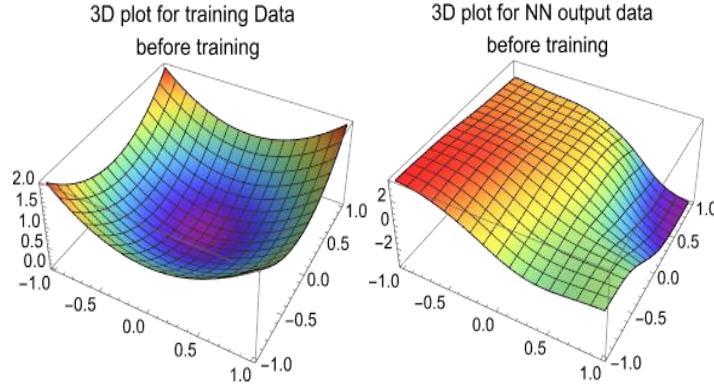
```

ImageSize->250
],
{layer,{

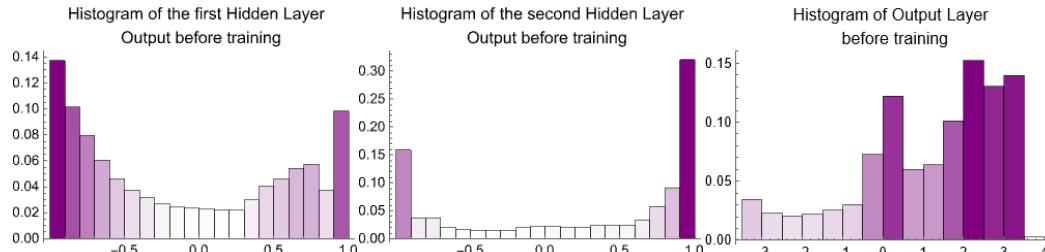
  {"First Hidden Layer \n Output after training",layer1Outputn},
  {"Second Hidden Layer \n Output after training",layer2Outputn},
  {"Output Layer \n after training",layer3Outputn}
}
}
]

```

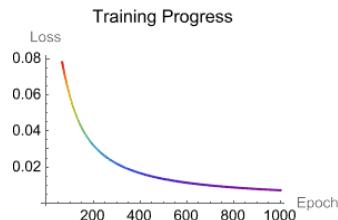
Output



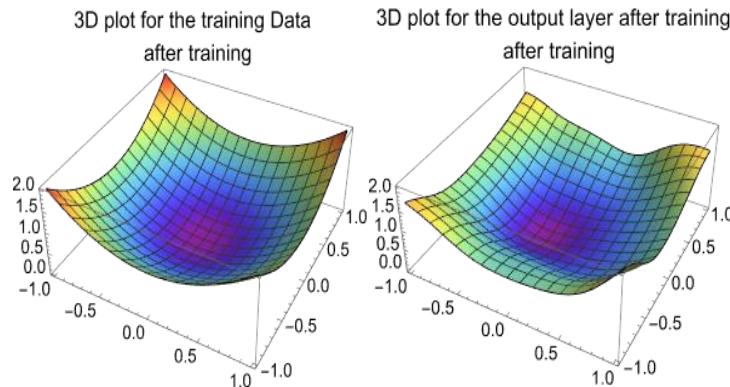
Output

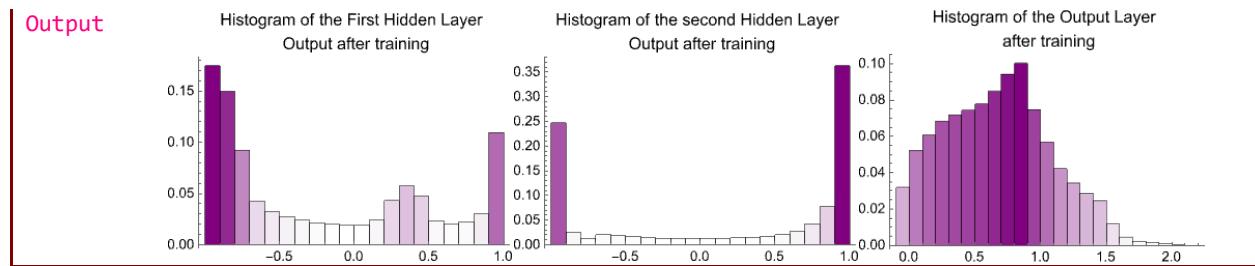


Output



Output



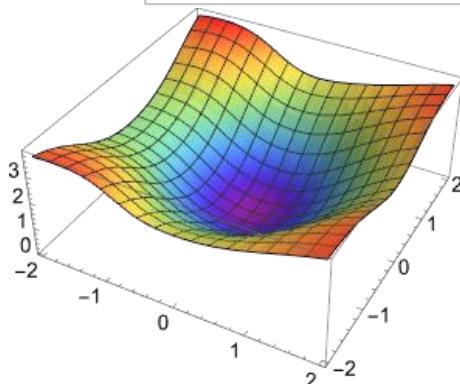
**Mathematica Code 4.2****Input**

```
(* The code demonstrates training a neural network to approximate a given function
and visualizing the learned function's surface. The code aims to achieve the
following: Generate a dataset consisting of input-output pairs where the inputs
are coordinates (x,y) and the outputs are the result of the function  $x^2+y^2$ .
Construct a neural network architecture with two hidden layers, each containing
10 neurons with hyperbolic tangent activation functions, and an output layer
with a single neuron. Train the neural network using the generated dataset to
learn the relationship between the input coordinates and the corresponding
function values. Visualize the learned function by plotting its 3D surface over
a specified range of x and y values: *)
```

```
(* Step 1:Generate the dataset: *)
data=Flatten@Table[
  {x,y}=>x^2+y^2,
  {x,-1,1,0.01},
  {y,-1,1,0.01}
];
(* Step 2:Define the neural network architecture: *)
neuralNet=NetChain[{10,Tanh,10,Tanh,1}];
(* Step 3:Train the neural network: *)
trainedNet=NetTrain[neuralNet,data,BatchSize->1024]
(* Step 4:Visualize the learned function: *)
Plot3D[
  trainedNet[{x,y}],
  {x,-2,2},
  {y,-2,2},
  NormalsFunction->None,
  ColorFunction->"Rainbow",
  ImageSize->220
]
```

Output

NetChain [Inputport: vector (size:2) Outputport: scalar]

Output

Comparing SGD and Full batch Gradient Descent (FBGD) provides insights into their respective strengths and weaknesses. The goal of the Mathematica Code 4.3 and Mathematica Code 4.4 are to implement FBDG and SGD, respectively, for linear regression using basic NN with a single neuron, one input, and one bias term.

Below are the main steps of Mathematica Code 4.3:

1. Define a cost function that measures the squared error between the predicted values and the actual values of the dataset.
2. Define the gradient of the cost function with respect to the model parameters (weights and bias) using the chain rule of differentiation.
3. Implement a FBDG algorithm that updates the model parameters using the gradients computed over the entire dataset at each iteration.
4. Visualize the linear regression model fitted to the dataset after running FBDG. This includes plotting the dataset and the regression line.
5. Plot the magnitude of the gradients over the iterations to observe the convergence behavior of the optimization algorithm.
6. Plot the gradients of the cost function with respect to the model parameters (weights and bias) over the iterations to understand how they change during optimization.
7. Plot the value of the cost function over the iterations to monitor the optimization progress and convergence.
8. Plot the values of the model parameters (weights and bias) over the iterations to observe how they change during optimization.

In comparison to SGD, Mathematica Code 4.4, which updates the model parameters using gradients computed from individual data points randomly sampled from the dataset, FBDG computes gradients using the entire dataset, Mathematica Code 4.3. This fundamental difference in sampling methodology leads to distinct convergence behaviors between the two optimization algorithms.

For example, the figure "Cost Function over 1000 Iterations", Mathematica Code 4.3, for FBDG typically shows smoother convergence compared to SGD, Mathematica Code 4.4, where the cost function tends to exhibit more fluctuations due to the randomness in selecting individual data points for computing gradients. While SGD can sometimes converge faster due to more frequent parameter updates, FBDG often provides more stable convergence and a smoother decrease in the cost function. However, FBDG may be computationally more expensive since it requires processing the entire dataset at each iteration.

In FBDG, the entire training dataset is used to compute the gradient of the cost function in each iteration. As a result, the cost function surface remains static throughout the training process since the gradient is calculated using all data points at once. Therefore, the plot displays a single static error surface. On the other hand, for SGD, only a single data point is used to compute the gradient in each iteration. This leads to more stochastic updates of the parameters and hence, a dynamic error surface (many error surfaces).

When visualizing various aspects such as magnitude of the gradient, evolution of weights and biases, cost function dynamics, and parameter values over iterations, FBDG consistently exhibits smoother curves compared to SGD. Conversely, SGD's reliance on stochastic updates often manifests in curves with more pronounced fluctuations and variability.

Mathematica Code 4.3

Input

(* The code implements a basic neural network with a single neuron, one input, and one bias term, primarily aimed at solving a linear regression problem. It utilizes the mean squared error (MSE) as the cost function to quantify the disparity between predicted and actual output values. Employing full batch gradient descent, the model iteratively adjusts its parameters, namely the weight connecting the input to the neuron and the bias, to minimize the cost function. Through visualization techniques such as plotting the gradient magnitude, gradients of the cost function with respect to the model parameters, cost function evolution, and

```

parameter values over iterations, the code offers insights into the training
process, enabling an understanding of how the model learns to make predictions
based on input data: *)

(* Define the cost function using one data point: *)
costFunction[x_,y_,w_,b_]=(y-(w*x+b))^2;

(* Define the gradient of the cost function using one data point: *)
costGradient[x_,y_,w_,b_]=D[costFunction[x,y,w,b],{{w,b}}];

(* Full Batch Gradient Descent function: *)
fullBatchGradientDescent[data_,initialGuess_,learningRate_,iterations_]:=Module[
{w=initialGuess[[1]],b=initialGuess[[2]],n=Length[data],gradientList={},costList
={{}},wValues={{}},bValues={{}},
 Do[
 (* Define the gradient of the cost function using full batch data: *)
 gradients=(1/Length[data])*Total[costGradient[#[[1]],#[[2]],w,b]&/@data];
 gradientList=Append[gradientList,gradients];

 (* Values of w and b: *)
 w=learningRate*gradients[[1]];
 b=learningRate*gradients[[2]];
 AppendTo[wValues,w];
 AppendTo[bValues,b];

 (* Define the cost function using full batch data: *)
 costs=(1/Length[data])*Total[costFunction[#[[1]],#[[2]],w,b]&/@data];
 costList=Append[costList,costs],
 {iterations}
 ];
 {w,b,gradientList,costList,wValues,bValues}
 ];

(* Generate some sample data: *)
data=Table[
 {x,2*x+3+RandomReal[{-1,1}]},
 {x,0,10}];

(* Run Full Batch Gradient Descent: *)
initialGuess={0.,0.};
learningRate=0.01;
iterations=1000;
{finalW,finalB,gradientList,costList,wValues,bValues}=fullBatchGradientDescent[d
ata,initialGuess,learningRate,iterations];

(* Mean squared errors for data points: *)
errors=Flatten[
 Table[
 {w,b,(1/Length[data])*Total[costFunction[#[[1]],#[[2]],w,b]&/@data]},
 {w,0,4,0.1},
 {b,0,4,0.1}
 ],
 1];

(* Print the final result: *)
Print["Final values: w = ",finalW,", b = ",finalB];

(* Visualize the Full Batch Gradient Descent for Linear Regression*)
Show[

```

```

ListPlot[
  data,
  PlotStyle->Red,
  PlotLegends->Placed[{"Data"}, {0.55, 0.1}],
  ImageSize->250
],
Plot[finalW*x+finalB,
{x, 0, 10},
PlotStyle->Blue,
PlotLegends->Placed[{"Regression Line"}, {0.55, 0.1}],
ImageSize->250
],
Frame->True,
FrameLabel->{"x", "y"},
PlotLabel->"Full Batch Gradient Descent Linear Regression"
]

(* Visualize the magnitude of the gradient over iterations: *)
ListLinePlot[
Norm/@gradientList,
PlotStyle->Blue,
Frame->True,
FrameLabel->{"Iterations", "Gradient Magnitude"},
PlotLabel->"Magnitude of Gradient over 1000 Iterations",
PlotRange->All,
ImageSize->250
]

(* Extract gradients of w and b: *)
gradientWList=gradientList[[All,1]];
gradientBList=gradientList[[All,2]];

(* Visualize the gradients of w and b over iterations: *)
ListLinePlot[
{gradientWList,gradientBList},
PlotStyle->{Blue,Red},
Frame->True,
FrameLabel->{"Iterations", "Gradient"},
PlotLegends-
>Placed[{\["\[\PartialD]L/\[\PartialD]w", "\[\PartialD]L/\[\PartialD]b"], {0.70, 0.4}],,
PlotLabel->" \[\PartialD]L/\[\PartialD]w and \[\PartialD]L/\[\PartialD]b over 1000 Iterations",
PlotRange->All,
ImageSize->250
]

(* Visualize the cost function over iterations: *)
ListLinePlot[
costList,
PlotStyle->Purple,
Frame->True,
FrameLabel->{"Iterations", "Cost"},
PlotLabel->"Cost Function over 1000 Iterations",
PlotRange->All,
ImageSize->250
]

(* Visualize the values of w and b over iterations: *)
ListLinePlot[
{wValues,bValues},
PlotStyle->{Blue,Red},

```

```

Frame->True,
FrameLabel->{"Iterations","Values"},  

PlotLegends->Placed[{"Value of w","Value of b"},{0.75,0.4}],  

PlotLabel->"Values of w and b over 1000 Iterations",
ImageSize->250
]

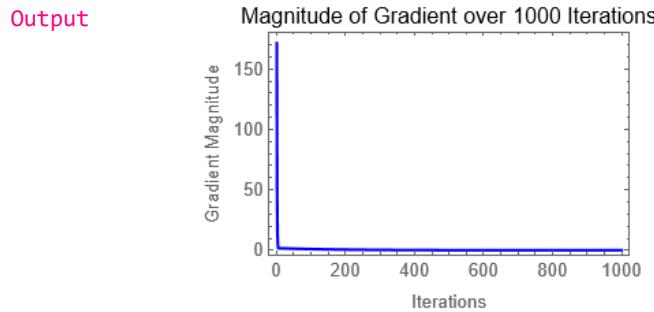
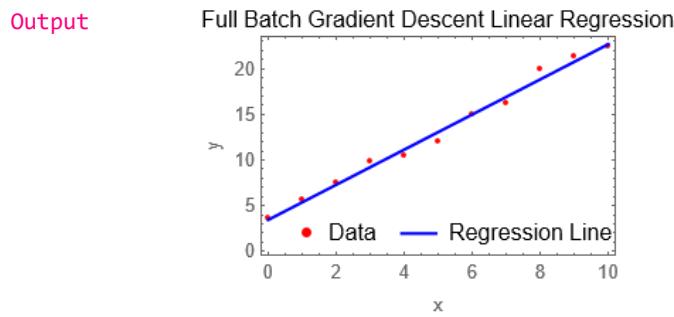
(* Visualize the cost function for values of w and b over iterations: *)
Show[
ListPlot3D[
errors,
PlotLegends->Automatic,
ColorFunction->"Rainbow",
PlotRange->Full,
PlotLabel->"Cost Function Surface and Updates \n of (w,b,cost) Over 1000
Epochs",
AxesLabel->{"w","b","Cost"},  

ImageSize->250
],
ListLinePlot3D[
Transpose[{wValues,bValues,costList}],
PlotStyle->Directive[Opacity[0.5],Red],
AxesLabel->{"w","b","Cost"},  

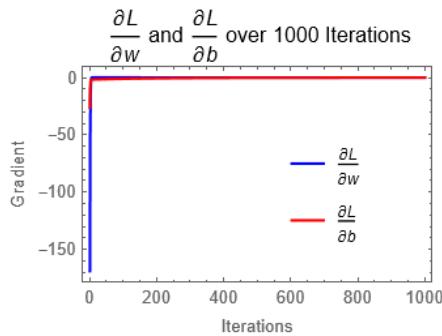
PlotRange->Full,
ImageSize->250
]
]

```

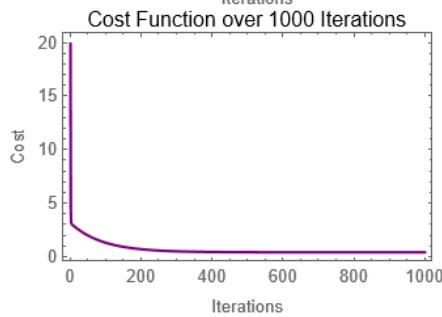
Output Final values: $w = 1.92445$, $b = 3.44856$



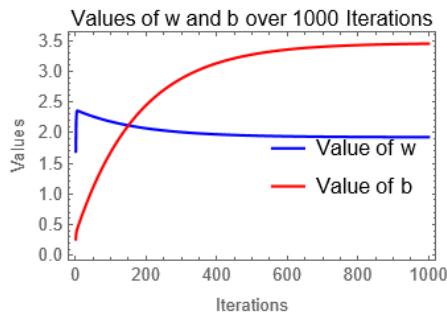
Output



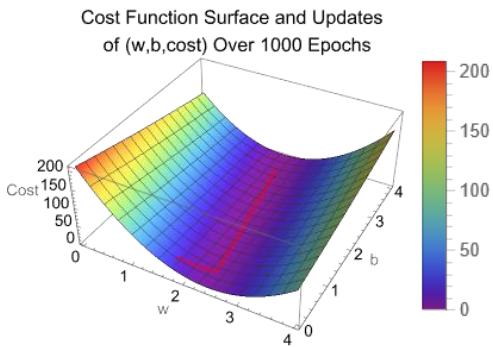
Output



Output



Output

**Mathematica Code 4.4**

Input

```
(* While both Mathematica codes 4.3 and 4.4 cover similar aspects, such as
visualizing the training process and optimization metrics, the differences lie
in the optimization algorithm. Mathematica code 4.4 uses SGD: *)
```

```
(* Define the cost function: *)
costFunction[x_,y_,w_,b_]=(y-(w*x+b))^2;
```

```
(* Define the gradient of the cost function: *)
costGradient[x_,y_,w_,b_]=D[costFunction[x,y,w,b],{{w,b}}];
```

```

(*Stochastic Gradient Descent function*)
stochasticGradientDescent[data_,initialGuess_,learningRate_,iterations_]:=Module[
{w=initialGuess[[1]],b=initialGuess[[2]],n=Length[data],gradient,x,y,gradientLi-
st,costList,wValues,bValues},
  gradientList={};
  costList={};
  wValues={};
  bValues={};
  Do[
    {x,y}=RandomChoice[data];
    gradient=costGradient[x,y,w,b];
    gradientList=Append[gradientList,gradient];
    w-=learningRate*gradient[[1]];
    b-=learningRate*gradient[[2]];
    AppendTo[wValues,w];
    AppendTo[bValues,b];
    cost=costFunction[x,y,w,b];
    costList=Append[costList,cost],
    {iterations}
  ];
  {w,b,gradientList,costList,wValues,bValues}
];

(* Generate some sample data: *)
data=Table[
  {x,2*x+3+RandomReal[{-1,1}]},
  {x,0,10}];

(* Run Stochastic Gradient Descent: *)
initialGuess={0.,0.};
learningRate=0.01;
iterations=1000;
{finalW,finalB,gradientList,costList,wValues,bValues}=stochasticGradientDescent
[data,initialGuess,learningRate,iterations];

(* Print the final result: *)
Print["Final values: w = ",finalW,", b = ",finalB];

(* Visualize the Stochastic Gradient Descent Linear Regression: *)
Show[
  ListPlot[
    data,
    PlotStyle->Red,
    PlotLegends->Placed[{"Data"},{0.55,0.1}],
    ImageSize->250
  ],
  Plot[finalW*x+finalB,
    {x,0,10},
    PlotStyle->Blue,
    PlotLegends->Placed[{"Regression Line"},{0.55,0.1}],
    ImageSize->250
  ],
  Frame->True,
  FrameLabel->{"x","y"},
  PlotLabel->"Stochastic Gradient Descent Linear Regression"
]

```

```

(* Mean squared error values for 11 data points: *)
errors=
Table[
  Flatten[
    Table[
      {w,b,costFunction[w,b,data[[i]][[1]],data[[i]][[2]]]},
      {w,-7,7,0.1},
      {b,-7,7,0.1}
    ],
    1],
{i,1,Length[data]}
];

(* 3D plot of mean squared error surfaces for 11 data points with respect to w
and b: *)
ListPlot3D[
  errors,
  ImageSize->250,
  PlotLabel->"Cost Function Surfaces",
  PlotStyle->Directive[Opacity[0.4]],
  PlotRange->Full
]

(* Visualize the magnitude of the gradient over iterations: *)
ListLinePlot[
  Norm/@gradientList,
  PlotStyle->Blue,
  Frame->True,
  FrameLabel->{"Iterations", "Gradient Magnitude"},
  PlotLabel->"Magnitude of Gradient over 1000 Iterations",
  PlotRange->All,
  ImageSize->250
]

(* Extract gradients of w and b: *)
gradientWList=gradientList[[All,1]];
gradientBList=gradientList[[All,2]];

(* Visualize the gradients of w and b over iterations: *)
ListLinePlot[
  {gradientWList,gradientBList},
  PlotStyle->{Blue,Red},
  Frame->True,
  FrameLabel->{"Iterations", "Gradient"},
  PlotLegends-
>Placed[{"\[PartialD]L/\[PartialD]w", "\[PartialD]L/\[PartialD]b"}, {0.70,0.25}],
  PlotLabel->" \[PartialD]L/\[PartialD]w and \[PartialD]L/\[PartialD]b over 1000
Iterations",
  PlotRange->All,
  ImageSize->250
]

(* Visualize the cost function over iterations: *)
ListLinePlot[
  costList,
  PlotStyle->Purple,
  Frame->True,
  FrameLabel->{"Iterations", "Cost"},
  PlotLabel->"Cost Function over 1000 Iterations",
  PlotRange->All,
  ImageSize->250
]

```

```

]

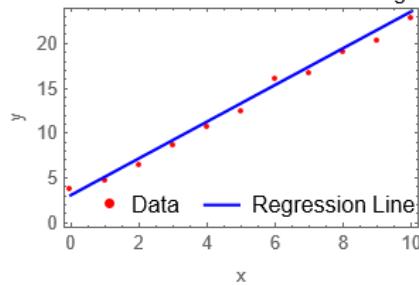
(* Visualize the values of w and b over iterations: *)
ListLinePlot[
 {wValues,bValues},
 PlotStyle->{Blue,Red},
 Frame->True,
 FrameLabel->{"Iterations","Values"},
 PlotLegends->Placed[{"Value of w","Value of b"},{0.75,0.2}],
 PlotLabel->"Values of w and b over 1000 Iterations",
 ImageSize->250
]

(* Visualize the cost function for values of w and b over iterations: *)
ListLinePlot3D[
 Transpose[{wValues,bValues,costList}],
 PlotStyle->Directive[Opacity[0.5],Purple],
 AxesLabel->{"w","b","Cost"},
 PlotLabel->"Updates of (w,b,Cost) \n Over 1000 Epochs",
 PlotRange->Full,
 ImageSize->250
]

```

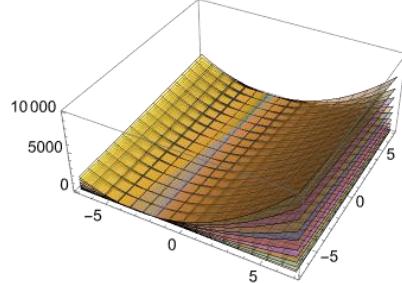
Output
Output

Final values: $w = 2.05194$, $b = 3.07732$
Stochastic Gradient Descent Linear Regression



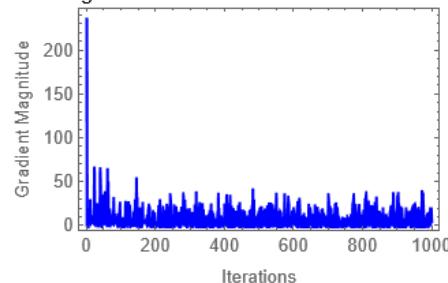
Output

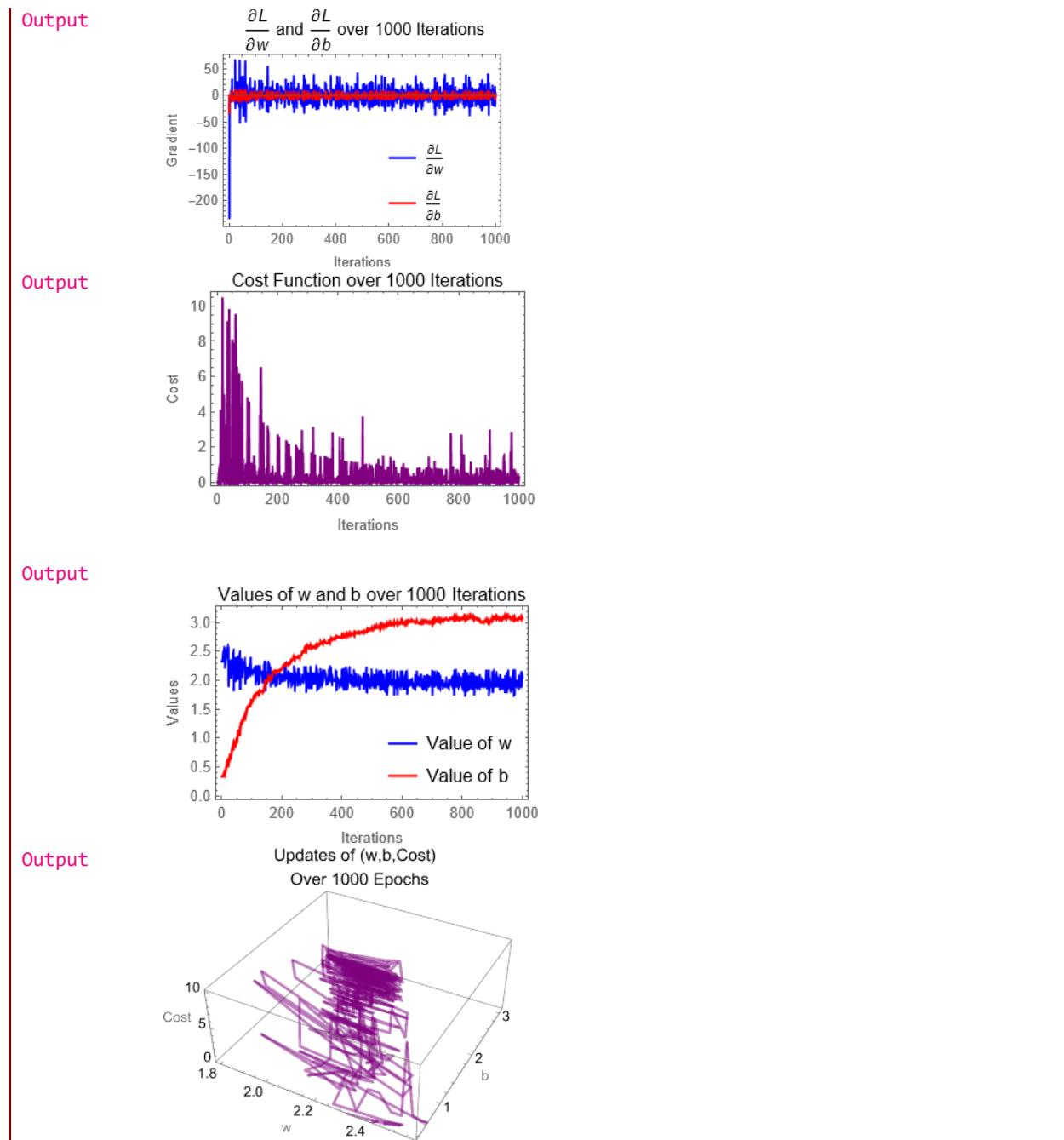
Cost Function Surfaces



Output

Magnitude of Gradient over 1000 Iterations



**Mathematica Code 4.5****Input**

```
(* case 1, without biases. Suppose we have a neural network with two layers, and
the activation function is the identity function ( $\Phi(x)=x$ ) for both layers. In
this example, the network effectively behaves as a single layer with the weight
matrix  $W'$ , demonstrating the reduction of a multi-layer network with identity
activation functions to a single-layer network: *)
```

```
(* Define weight matrices *)
W1={{2,1},{1,3}};
W2={{1,2},{3,4}};
(* Define input vector *)
x={1,2};
```

```
(* Compute output of the first layer *)
y1=W1.x;
(* Compute output of the second layer *)
y2=W2.y1;
(* Combine weight matrices *)
Wprime=W2.W1;
(* Compute the final output of the entire network *)
yFinal=Wprime.x;
(* Display results *)
{y1,y2,Wprime,yFinal}

Output {{4,7},{18,40},{{4,7},{10,15}}, {18,40}}
```

Mathematica Code 4.6

Input (* case 2, with biases. Suppose we have a neural network with two layers, and the activation function is the identity function ($\Phi(x)=x$) for both layers. In this example, the network effectively behaves as a single layer with the weight matrix W' , demonstrating the reduction of a multi-layer network with identity activation functions to a single-layer network: *)

```
(* Define weight matrices *)
W1={{2,1},{1,3}};
W2={{1,2},{3,4}};
(* Define biases *)
b1={1,2};
b2={3,4};
(* Define input vector *)
x={1,2};
(* Compute output of the first layer with biases *)
y1=W1.x+b1;
(* Compute output of the second layer with biases *)
y2=W2.y1+b2;
(* Combine weight matrices for the entire network *)
Wprime=W2.W1;
(* Compute the final output of the entire network with biases *)
yFinal=Wprime.x+(W2.b1+b2);
(* Display results *)
{y1,y2,Wprime,yFinal}

Output {{5,9},{26,55},{{4,7},{10,15}}, {26,55}}
```

Mathematica Code 4.7

Input (* The goal of the code is to define and visualize a mathematical function referred to as the "rectangular towers function." This function is constructed by taking the difference between two sigmoid functions with distinct peaks and shifts. The sigmoid functions are defined with a steep slope and a shift parameter, and their specific parameters are set to create a left sigmoid (sigmoidLeft) and a right sigmoid (sigmoidRight). The rectangularTowers function is then formed by subtracting the right sigmoid from the left sigmoid. The resulting function represents a rectangular towers-like structure. The code includes plotting commands to visually represent the rectangular towers function (rectangularTowers) as well as the individual left (sigmoidLeft) and right (sigmoidRight) sigmoid functions: *)

```
(* Define the sigmoid function with a steep slope and a shift parameter *)
sigmoid[x_,a_,b_]:=1/(1+Exp[a x+b])

(* Define two sigmoid functions with different peaks and shifts *)
sigmoidLeft[x_]:=sigmoid[x,-15,-10]
sigmoidRight[x_]:=sigmoid[x,-15,10]
```

```
(* Define the rectangular towers function by subtracting the two sigmoid functions *)
rectangularTowers[x_]:=sigmoidLeft[x]-sigmoidRight[x]

(* Plot the result: *)

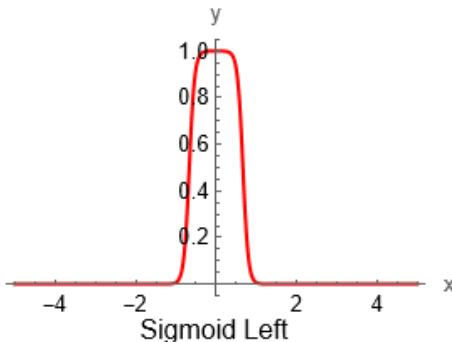
Plot[
  rectangularTowers[x],
  {x,-5,5},
  PlotRange->All,
  AxesLabel->{"x", "y"},
  PlotLabel->"Tower Function",
  ImageSize->200,
  PlotStyle->Red
]

Plot[
  sigmoidLeft[x],
  {x,-5,5},
  PlotRange->All,
  AxesLabel->{"x", "y"},
  PlotLabel->"Sigmoid Left",
  ImageSize->200,
  PlotStyle->Blue
]

Plot[
  sigmoidRight[x],
  {x,-5,5},
  PlotRange->All,
  AxesLabel->{"x", "y"},
  PlotLabel->"Sigmoid Right",
  ImageSize->200,
  PlotStyle->Blue
]
```

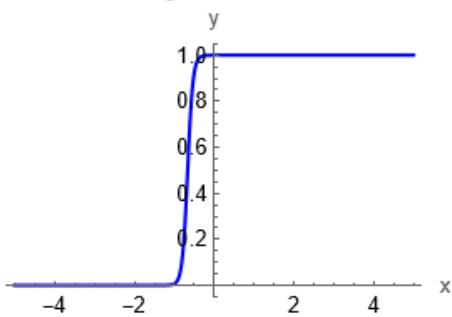
Output

Tower Function



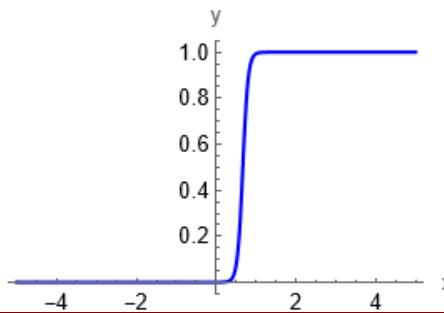
Output

Sigmoid Left



Output

Sigmoid Right

**Mathematica Code 4.8**

```

Input      (* The goal of the code is to define and visualize a function f(x) that is the
           difference between two sigmoid functions. The parameters a and b in the sigmoid
           function control the shape and position of the curve. The Manipulate function
           create an interactive plot where you can dynamically adjust the parameters a1,
           b1, a2, and b2 and observe the corresponding changes in the function f(x): *)

(* Define sigmoid functions: *)
sigmoid[x_,a_,b_]:=1/(1+Exp[a x+b])

(* Define the function f(x) as the difference of two sigmoid functions: *)
f[x_,a1_,b1_,a2_,b2_]:=sigmoid[x,a1,b1]-sigmoid[x,a2,b2]

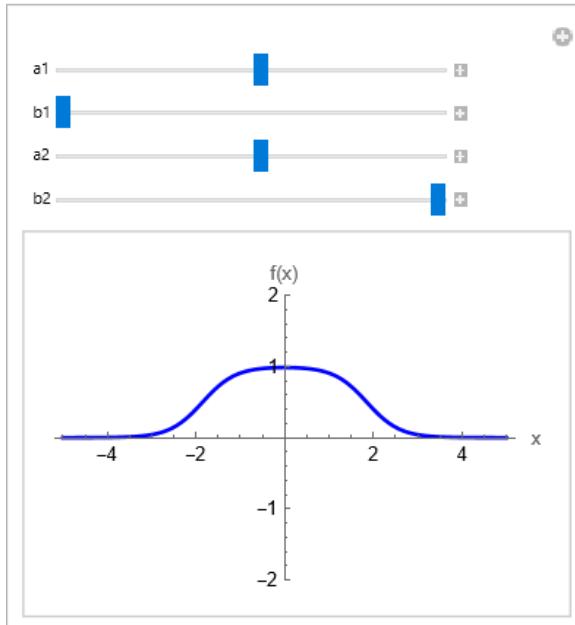
(* Set parameter values: *)

a1=2;b1=1; (* Parameters for the first sigmoid *)
a2=3;b2=-1; (* Parameters for the second sigmoid *)

(* Plot the function f(x): *)
Manipulate[
  Plot[
    f[x,a1,b1,a2,b2],
    {x,-5,5},
    PlotStyle->{Thick,Blue},
    AxesLabel->{"x","f(x)"},
    PlotRange->{-2,2},
    ImageSize->250
  ],
  {{a1,2.7,"a1"},0.1,5,0.1},
  {{b1,-5,"b1"},-5,5,0.1},
  {{a2,2.7,"a2"},0.1,5,0.1},
  {{b2,5,"b2"},-5,5,0.1}
]

```

Output



Mathematica Code 4.9

```

Input (* We sum two-dimensional sigmoid functions together and plot the result. The
       result is a ridged surface. Since our goal is to obtain localized bumps we select
       another pair of two-dimensional sigmoid functions, add them together to get a
       ridged surface perpendicular to the first ridged surface, and then add the two
       perpendicular ridged surfaces together. Take two-dimensional sigmoid function to
       the result which will depress the local minima and saddles to zero while
       simultaneously sending the central maximum towards 1: *)

(* Define the 2D sigmoid function: *)
sigmoid2D[x_,y_,a_,b_,c_]:=1/(1+Exp[(a x+b y)-c])

(* Define the function to create ridges by subtracting two 2D sigmoid functions:
*)
f2D[x_,y_,a1_,b1_,a2_,b2_,c1_,c2_]:=sigmoid2D[x,y,a1,b1,c1]-
sigmoid2D[x,y,a2,b2,c2]

(* Common plot options to be reused for all plots: *)
plotOptions={

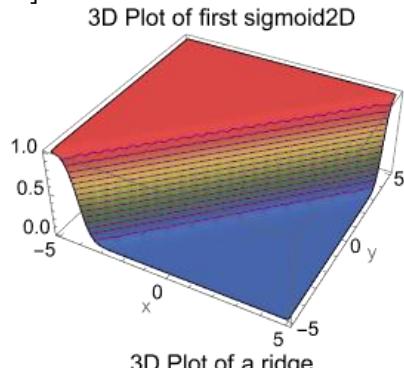
  ColorFunction->"DarkRainbow",
  PlotRange->All,
  MeshFunctions->{#3&},
  MeshStyle->{{Thin,Purple}},
  AxesLabel->{"x","y",""},
  ImageSize->200
};

(* Plot the first 2D sigmoid function: *)
Plot3D[
  sigmoid2D[x,y,5,-5,5],
  {x,-5,5},
  {y,-5,5},
  Evaluate@Append[
    plotOptions,
    PlotLabel->"3D Plot of first sigmoid2D"
  ]
]

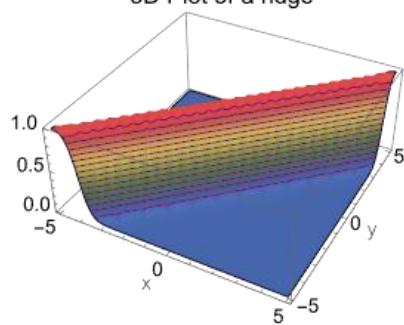
```

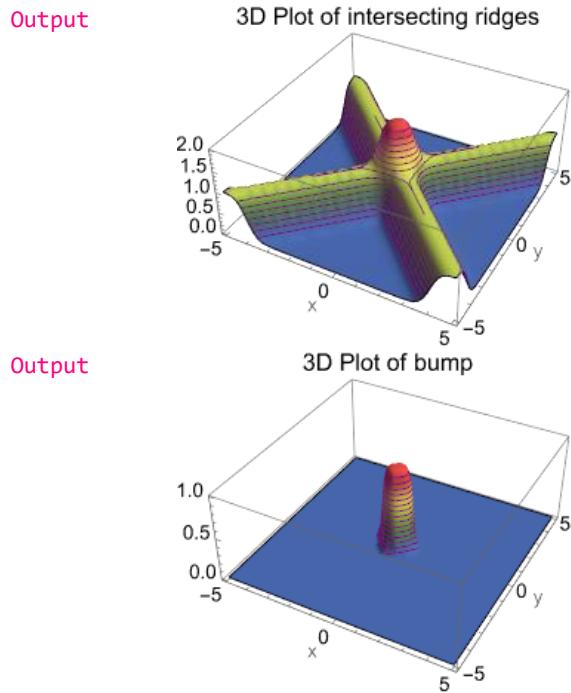
```
(* Plot the ridge created by the f2D function: *)
Plot3D[
  f2D[x,y,5,-5,5,-5,5,-5],
  {x,-5,5},
  {y,-5,5},
  Evaluate@Append[
    plotOptions,
    PlotLabel->"3D Plot of a ridge"
  ]
]
(* Plot intersecting ridges by summing two f2D functions: *)
Plot3D[
  f2D[x,y,5,-5,5,-5,5,-5]+f2D[x,y,-5,-5,-5,-5,5,-5],
  {x,-5,5},
  {y,-5,5},
  Evaluate@Append[
    plotOptions,
    PlotLabel->"3D Plot of intersecting ridges"
  ]
]
(* Plot the bump using a nested 2D sigmoid function: *)
Plot3D[
  Module[
    {x1,y1},
    x1=f2D[x,y,5,-5,5,-5,5,-5];
    y1=f2D[x,y,-5,-5,-5,-5,5,-5];
    sigmoid2D[x1,y1,-15,-15,-25]
  ],
  {x,-5,5},
  {y,-5,5},
  Evaluate@Append[
    plotOptions,
    PlotLabel->"3D Plot of bump"
  ]
]
```

Output



Output



**Mathematica Code 4.10**

```

Input (* The code aims to create an interactive environment for exploring the behavior
       of two-dimensional sigmoid functions and their differences, allowing users to
       visualize the impact of changing parameter values on the resulting plots. We sum
       two-dimensional sigmoid functions together and plot the result. The result is a
       ridged surface. Since our goal is to obtain localized bumps we select another
       pair of two-dimensional sigmoid functions, add them together to get a ridged
       surface perpendicular to the first ridged surface, and then add the two
       perpendicular ridged surfaces together. Take two-dimensional sigmoid function to
       the result which will depress the local minima and saddles to zero while
       simultaneously sending the central maximum towards 1: *)

(* Define the 2D sigmoid function: *)
sigmoid2D[x_,y_,a_,b_,c_]:=1/(1+Exp[(a x+b y)-c])

(* Define the function to create ridges by subtracting two 2D sigmoid
functions: *)
f2D[x_,y_,a1_,b1_,a2_,b2_,c1_,c2_]:=sigmoid2D[x,y,a1,b1,c1]-
sigmoid2D[x,y,a2,b2,c2]

(* Define common plot settings to be reused in all plots: *)
commonPlotSettings={

  ColorFunction->"DarkRainbow",
  PlotRange->All,
  MeshFunctions->{#3&},
  MeshStyle->{{Thin,Purple}},
  AxesLabel->{"x","y","f(x,y)"},
  ImageSize->250
};

(* Helper function to create a 3D plot with the given function and label: *)
createPlot3D[func_,{x_,xMin_,xMax_},{y_,yMin_,yMax_},plotLabel_]:=Plot3D[
  func,
  {x,xMin,xMax},
  {y,yMin,yMax},
  PlotLabel->plotLabel
];

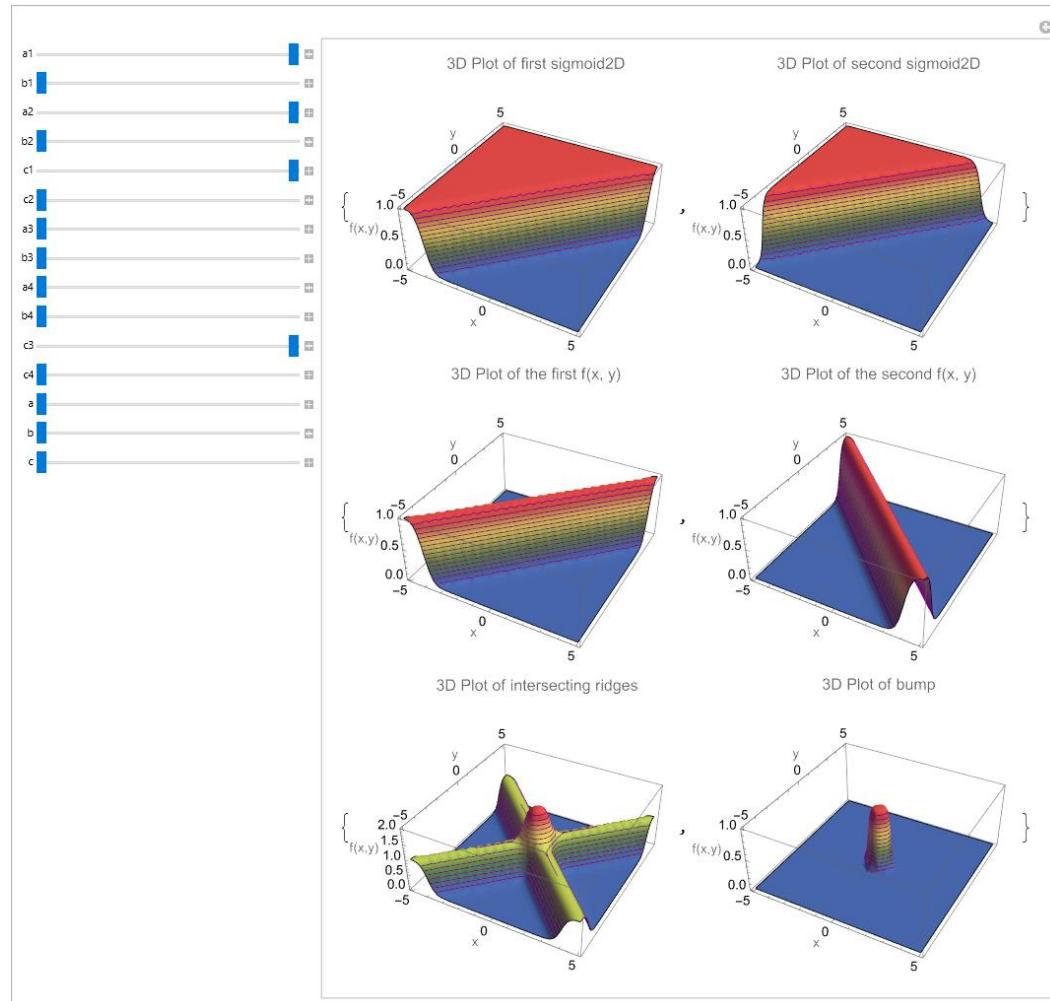
```

```
Evaluate@Append[
  commonPlotSettings,
  PlotLabel->plotLabel
]
]

(* Create an interactive environment using Manipulate: *)
Manipulate[
 Column[
 {
   (* Plot the first 2D sigmoid function: *)
   createPlot3D[
     sigmoid2D[x,y,a1,b1,c1],
     {x,-5,5},
     {y,-5,5},
     "3D Plot of first sigmoid2D"
   ],
   (* Plot the second 2D sigmoid function: *)
   createPlot3D[
     sigmoid2D[x,y,a2,b2,c2],
     {x,-5,5},
     {y,-5,5},
     "3D Plot of second sigmoid2D"
   ],
   {
     (* Plot the first ridge function: *)
     createPlot3D[
       f2D[x,y,a1,b1,a2,b2,c1,c2],
       {x,-5,5},
       {y,-5,5},
       "3D Plot of the first f(x, y)"
     ],
     (* Plot the second ridge function: *)
     createPlot3D[
       f2D[x,y,a3,b3,a4,b4,c3,c4],
       {x,-5,5},
       {y,-5,5},
       "3D Plot of the second f(x, y)"
     ],
     {
       (* Plot the sum of the two ridge functions: *)
       createPlot3D[
         f2D[x,y,a1,b1,a2,b2,c1,c2]+f2D[x,y,a3,b3,a4,b4,c3,c4],
         {x,-5,5},
         {y,-5,5},
         "3D Plot of intersecting ridges"
       ],
       (* Plot the result after applying the sigmoid function to the ridges: *)
       createPlot3D[
         sigmoid2D[f2D[x,y,a1,b1,a2,b2,c1,c2],f2D[x,y,a3,b3,a4,b4,c3,c4],a,b,c],
         {x,-5,5},
         {y,-5,5},
         "3D Plot of bump"
       ]
     }
   }
],
] ,
```

```
(* Parameter controls for interactive adjustment: *)
{{a1,5,"a1"},-5,5,0.1},
{{b1,-5,"b1"},-5,5,0.1},
{{a2,5,"a2"},-5,5,0.1},
{{b2,-5,"b2"},-5,5,0.1},
{{c1,5,"c1"},-5,5,0.1},
{{c2,-5,"c2"},-5,5,0.1},
{{a3,-5,"a3"},-5,5,0.1},
{{b3,-5,"b3"},-5,5,0.1},
{{a4,-5,"a4"},-5,5,0.1},
{{b4,-5,"b4"},-5,5,0.1},
{{c3,5,"c3"},-5,5,0.1},
{{c4,-5,"c4"},-5,5,0.1},
{{a,-15,"a"},-15,10,0.1},
{{b,-15,"b"},-15,10,0.1},
{{c,-25,"c"},-25,10,0.1}
]
```

Output



Unit 4.2

Layers: LinearLayer and ElementwiseLayer

LinearLayer

A `LinearLayer` represents a fully connected layer where each neuron in the layer is connected to every neuron in the preceding layer. When you use a `LinearLayer`, you're effectively introducing a set of neurons in your NN, each performing a linear transformation on the input data.

`LinearLayer[n]`

represents a trainable, fully connected net layer that computes $\mathbf{W} \cdot \mathbf{x} + \mathbf{b}$ with output vector of size n.

`LinearLayer[{n1, n2, ...}]`

represents a layer that outputs an array of dimensions $n_1 \times n_2 \times \dots$.

`LinearLayer[]`

leaves the dimensions of the output array to be inferred from context.

`LinearLayer[n, opts]`

includes options for initial weights and other parameters.

- The following optional parameters can be included:

<code>"Biases"</code>	<code>Automatic</code>	initial vector of biases (\mathbf{b} in $\mathbf{W} \cdot \mathbf{x} + \mathbf{b}$)
<code>"Weights"</code>	<code>Automatic</code>	initial matrix of weights (\mathbf{W} in $\mathbf{W} \cdot \mathbf{x} + \mathbf{b}$)

- When you create a NN layer in Mathematica using `LinearLayer` and don't specify weights and biases explicitly, they are automatically generated when you initialize the network using `NetInitialize` or train it using `NetTrain`.
- `"Biases" -> None`: This setting is used to explicitly specify that biases should not be used in the layer.
- When you apply a `LinearLayer` to input data, `LinearLayer[...][input]`, it computes the output by applying the learned weights and biases (if any) to the input data.
- If you provide a list of inputs `{input1, input2, ...}` to a `LinearLayer`, it computes the outputs for each of these inputs separately.
- The `NetExtract` function in Mathematica can be used to extract specific parts (such as weights and biases) from a trained NN. For a `LinearLayer` object, you can use `NetExtract` to retrieve its weights and biases.
- If you specify `LinearLayer[{}]`, it indicates that the `LinearLayer` should produce a single real number as output. This is useful in scenarios where you want the output of the layer to be a scalar value.
- `LinearLayer[n, "Input" -> m]`: This is indeed a common usage pattern for `LinearLayer`. It represents a linear layer that takes a vector of length `m` as input and produces a vector of length `n` as output. This is a fundamental building block in many NN architectures, where fully connected layers transform input vectors into output vectors.
- In larger NN architectures, sometimes the input shape of a specific layer cannot be inferred from previous layers. In such cases, you can use the `"Input" -> shape` option with `LinearLayer` to fix the input shape. The

shape parameter can take various forms to specify the input shape, allowing you to define the structure of the network more explicitly.

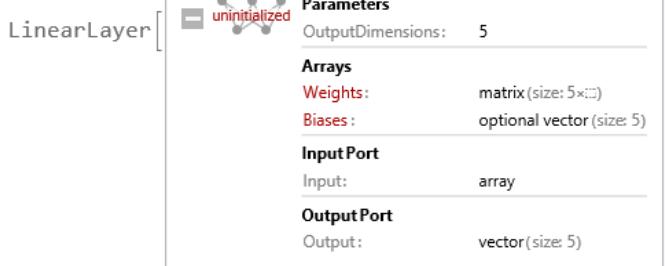
- Note that the layers containing learnable parameters appear in red, indicating that they require initial values before the net can be applied to an input.

"Real"	a single real number
m	a vector of length m
$\{m_1, m_2, \dots\}$	an array of dimensions $m_1 \times m_2 \times \dots$

Mathematica Code 4.11

Input (* Create a LinearLayer whose output is a length-5 vector: *)
LinearLayer[5]

Output



Mathematica Code 4.12

Input (* In this example: $\{\{1,2,3\}, \{4,5,6\}\}$ is a custom 2×3 weight matrix. $\{0.5, -1\}$ is a custom bias vector with length equal to the number of outputs (2 in this case). `LinearLayer[2, "Input" -> 3]` creates a linear layer with 3 inputs and 2 outputs. `RandomReal[1, {3, 3}]` generates a random matrix with 3 samples and 3 features as input data. These custom weights and biases are then used to create a `LinearLayer` with the specified parameters. The `customLinearLayer[inputData]` applies this custom linear layer to the input data: *)

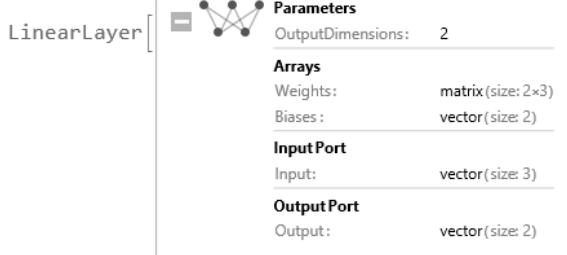
(* Specify a custom weight matrix and bias vector: *)
`customWeights = {{1, 2, 3}, {4, 5, 6}}; (*2x3 matrix*)`
`customBiases = {0.5, -1}; (* 1D vector with length equal to the number of outputs*)`

(* Create a LinearLayer with custom weights and biases: *)
`customLinearLayer = LinearLayer[2, "Input" -> 3, "Weights" -> customWeights, "Biases" -> customBiases]`

(* Generate some example input data: *)
`inputData = RandomReal[1, {3, 3}]`

(* Apply the custom linear layer to the input data: *)
`outputData = customLinearLayer[inputData]`

Output



```

Output {  

  {0.220028,0.802785,0.734378},  

  {0.0580037,0.538777,0.423361},  

  {0.525993,0.0475469,0.498489}  

}  

Output {  

  {4.52873,8.3003},  

  {2.90564,4.46607},  

  {2.61655,4.33264}  

}

```

Mathematica Code 4.13

```

Input (* In this example: Biases->None indicates that no biases should be used. The custom weight matrix is specified as {{1,2,3},{4,5,6}}. The resulting customLinearLayer is then applied to the example input data: *)  

(* Specify a custom weight matrix: *)
customWeights={{1,2,3},{4,5,6}}; (*2x3 matrix. *)  

(* Create a LinearLayer with custom weights and no biases: *)
customLinearLayer=LinearLayer[2,"Input"->3,"Weights"->customWeights,"Biases"->None]  

(* Generate some example input data: *)
inputData=RandomReal[1,{2,3}]  

(* Apply the custom linear layer to the input data: *)
outputData=customLinearLayer[inputData]  

(* Extract the weight matrix from the custom linear layer: *)
weights=NetExtract[customLinearLayer,"Weights"]  

(* Display the custom weight matrix: *)
Normal[weights]

```

Output

LinearLayer []

Parameters	
OutputDimensions:	2
Arrays	
Weights:	matrix(size: 2×3)
Biases:	None
InputPort	
Input:	vector(size: 3)
OutputPort	
Output:	vector(size: 2)

```

Output {  

  {0.590603,0.384391,0.635673},  

  {0.974991,0.950379,0.485747}  

}

```

```

Output {  

  {3.2664,8.0984},  

  {4.33299,11.5663}  

}

```

Output

NumericArray []

Type: Real32
Dimensions: {2, 3}
Data: 1., 2., 3., ...

```
Output {  
  {1.,2.,3.},  
  {4.,5.,6.}  
}
```

Mathematica Code 4.14

```
Input (* The code creates a LinearLayer with specified input and output dimensions. It  
       then uses NetInitialize to randomly initialize the layer's weights and biases. The  
       code generates example input data and applies the initialized linear layer to  
       compute the corresponding output: *)  
  
(* Specify the dimensions for the LinearLayer: *)  
inputSize=2;  
outputSize=3;  
  
(* Create a LinearLayer without specifying weights and biases: *)  
uninitializedLinearLayer=LinearLayer[outputSize,"Input"→inputSize]  
  
(* Initialize the LinearLayer with random weights and biases using NetInitialize:  
*)  
randomlyInitializedLinearLayer=NetInitialize[uninitializedLinearLayer]  
  
(* Generate some example input data: *)  
inputData=RandomReal[1,{2,inputSize}]  
  
(*Apply the randomly initialized linear layer to the input data: *)  
outputData=randomlyInitializedLinearLayer[inputData]
```

Output

LinearLayer[ **uninitialized** **Parameters**
OutputDimensions: 3
Arrays
Weights: matrix(size: 3×2)
Biases: optional vector (size: 3)
InputPort
Input: vector(size: 2)
OutputPort
Output: vector(size: 3)]

Output

LinearLayer[ **Parameters**
OutputDimensions: 3
Arrays
Weights: matrix(size: 3×2)
Biases: vector (size: 3)
InputPort
Input: vector(size: 2)
OutputPort
Output: vector(size: 3)]

Output

```
{  
  {0.9754,0.276369},  
  {0.0588502,0.681333}  
}
```

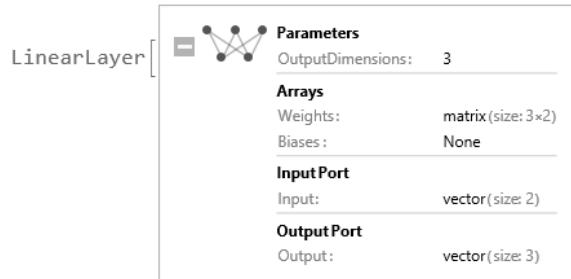
Output

```
{  
  {-0.364402,-0.799045,1.58972},  
  {-0.0551678,0.673965,-0.522975}  
}
```

Mathematica Code 4.15

```
Input (* Define and initialize a LinearLayer without biases: *)
linear=NetInitialize@LinearLayer[3,"Biases"->None,"Input"->2]
```

Output

**Mathematica Code 4.16**

```
Input (* The code demonstrates the manual computation of a linear transformation and
compares it with the result obtained using the built-in LinearLayer. The linear
function is defined to perform the transformation, taking input data, a weight
matrix, and a bias vector. The code initializes a LinearLayer with specified input
and output dimensions, applies it to example data, and then manually computes the
transformation using extracted weights and biases. The primary goal is to illustrate
the equivalence of results between the manual computation and the LinearLayer,
providing insights into the underlying linear transformation process within neural
networks: *)
```

```
(* Define a function to compute the linear transformation manually: *)
linear[data_,weight_,bias_]:=Dot[weight,data]+bias
```

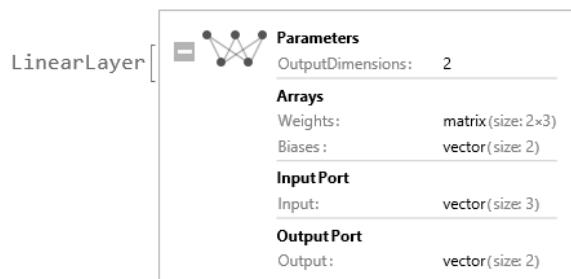
```
(*Example data vector*)
data={2,10,3};
```

```
(*Create and initialize a LinearLayer with 3 inputs and 2 outputs: *)
layer=NetInitialize@LinearLayer[2,"Input"->3]
```

```
(* Evaluate the LinearLayer on the example data: *)
layer[data]
```

```
(* Manually compute the same result using the defined linear function: *)
linearResult=linear[
  data,
  NetExtract[layer,"Weights"]//Normal,
  NetExtract[layer,"Biases"]//Normal
]
```

Output



```
Output {-3.75531,14.955}
```

```
Output {-3.75531,14.955}
```

ElementwiseLayer

The `ElementwiseLayer` is used to introduce element-wise non-linearities or transformations to the output of a NN layer.

`ElementwiseLayer[f]`

represents a net layer that applies a unary function `f` to every element of the input array.

`ElementwiseLayer["name"]`

applies the function specified by "name".

- The function `f` can be any one of the following: `Ramp`, `LogisticSigmoid`, `Tan`, `Tanh`, `ArcTan`, `ArcTanh`, `Sin`, `Sinh`, `ArcSin`, `ArcSinh`, `Cos`, `Cosh`, `ArcCos`, `ArcCosh`, `Cot`, `Coth`, `ArcCot`, `ArcCoth`, `Csc`, `Csch`, `ArcCsc`, `ArcCsch`, `Sec`, `Sech`, `ArcSec`, `ArcSech`, `Haversine`, `InverseHaversine`, `Gudermannian`, `InverseGudermannian`, `Log`, `Exp`, `Sqrt`, `CubeRoot`, `Abs`, `Gamma`, `LogGamma`, `Erf`, `InverseErf`, `Erfc`, `InverseErfc`, `Round`, `Floor`, `Ceiling`, `Sign`, `FractionalPart`, `IntegerPart`, `Unitize`, `KroneckerDelta`.
- In general, `f` can be any object that when applied to a single argument gives any combination of `Ramp`, `LogisticSigmoid`, etc., together with numbers, `Plus`, `Subtract`, `Times`, `Divide`, `Power`, `Surd`, `Min`, `Max`, `Clip`, `Mod`, `Threshold`, `Chop` and some logical operations using `If`, `And`, `Or`, `Which`, `Piecewise`, `Equal`, `Greater`, `GreaterEqual`, `Less`, `LessEqual`, `Unequal`, `Negative`, `NonNegative`, `Positive`, `NonPositive`, `PossibleZeroQ`.
- `ElementwiseLayer` supports the following values for "name":

"RectifiedLinearUnit" or "ReLU"	$\text{Ramp}[x]$
"ExponentialLinearUnit" or "ELU"	$x \text{ when } x \geq 0$ $\text{Exp}[x]-1 \text{ when } x < 0$
"ScaledExponentialLinearUnit" or "SELU"	$1.0507 x \text{ when } x \geq 0$ $1.758099340847161 \cdot (\text{Exp}[x]-1) \text{ when } x < 0$
"GaussianErrorLinearUnit" or "GELU"	$0.5 x(1+\text{Erf}[x/\sqrt{2}])$
"Swish"	$x \text{ LogisticSigmoid}[x]$
"HardSwish"	$x \text{ Min}[\text{Max}[x+3,0],6]/6$
"Mish"	$x \text{ Tanh}[\text{Log}[1+\text{Exp}[x]]]$
"SoftSign"	$x/(1+\text{Abs}[x])$
"SoftPlus"	$\text{Log}[\text{Exp}[x]+1]$
"HardTanh"	$\text{Clip}[x,\{-1,1\}]$
"HardSigmoid"	$\text{Clip}[(x+1)/2,\{0,1\}]$
"Sigmoid"	$\text{LogisticSigmoid}[x]$

- When you apply an `ElementwiseLayer` to a single input `ElementwiseLayer[...][input]`, it explicitly computes the output by applying the layer's operation to the input. This operation could be any element-wise function or operation specified when creating the `ElementwiseLayer`.

- If you provide a list of inputs `{input1, input2, ...}` to an `ElementwiseLayer`, `ElementwiseLayer[...][{input1, input2, ...}]`, it explicitly computes the outputs for each of these inputs separately. Each input will undergo the same element-wise operation independently.
- In a larger NN architecture, sometimes it's necessary to explicitly specify the dimensions of the input to an `ElementwiseLayer`. This is particularly important when the input dimensions cannot be inferred from previous layers. In such cases, you can use the "`Input`" $\rightarrow\{n_1, n_2, \dots\}$ option to fix the input dimensions of the `ElementwiseLayer`.

Mathematica Code 4.17

Input (* This Mathematica code demonstrates the creation of an `ElementwiseLayer` using the `Tanh` function and showcases its application to a single vector. Additionally, it provides a plot to visualize the `Tanh` function's behavior over a specified input range: *)

```
(* Create an ElementwiseLayer that computes Tanh of each input element: *)
tanh=ElementwiseLayer[Tanh]
```

```
(* Apply the layer to a single vector: *)
tanh[{-3,-2,-1,0,1,2,3}]
```

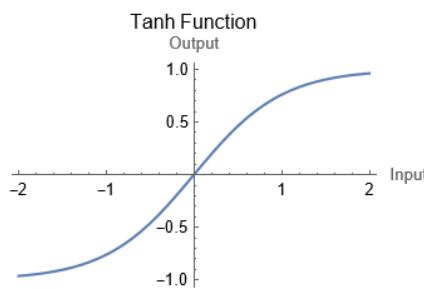
```
(* Apply the ElementwiseLayer to a range of values: *)
range=Range[-2,2,0.1];
resultVector=tanh/@range;
```

```
(* Plot the results: *)
```

```
ListLinePlot[
Transpose[{range,resultVector}],
AxesLabel->{"Input","Output"},
PlotLabel->"Tanh Function",
ImageSize->250
]
```

Output

Output {-0.995055, -0.964028, -0.761594, 0., 0.761594, 0.964028, 0.995055}

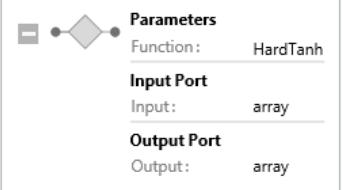
Output**Mathematica Code 4.18**

Input (* Create an ElementwiseLayer that computes hardTanh of each input element: *)
hardtanh=ElementwiseLayer["HardTanh"]

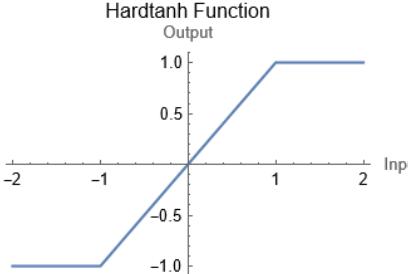
```
(* Apply the layer to a single vector: *)
hardtanh[{-3,-2,-1,0,1,2,3}]

(* Apply the ElementwiseLayer to a range of values: *)
range=Range[-2,2,0.1];
resultVector=hardtanh/@range;

(* Plot the results: *)
ListLinePlot[
Transpose[{range,resultVector}],
AxesLabel->{"Input","Output"},
PlotLabel->"Hardtanh Function",
ImageSize->250
]

Output
ElementwiseLayer[]
Parameters
Function: HardTanh
Input Port
Input: array
Output Port
Output: array

Output {-1.,-1.,-1.,0.,1.,1.,1.}

Output Hardtanh Function
Output


```

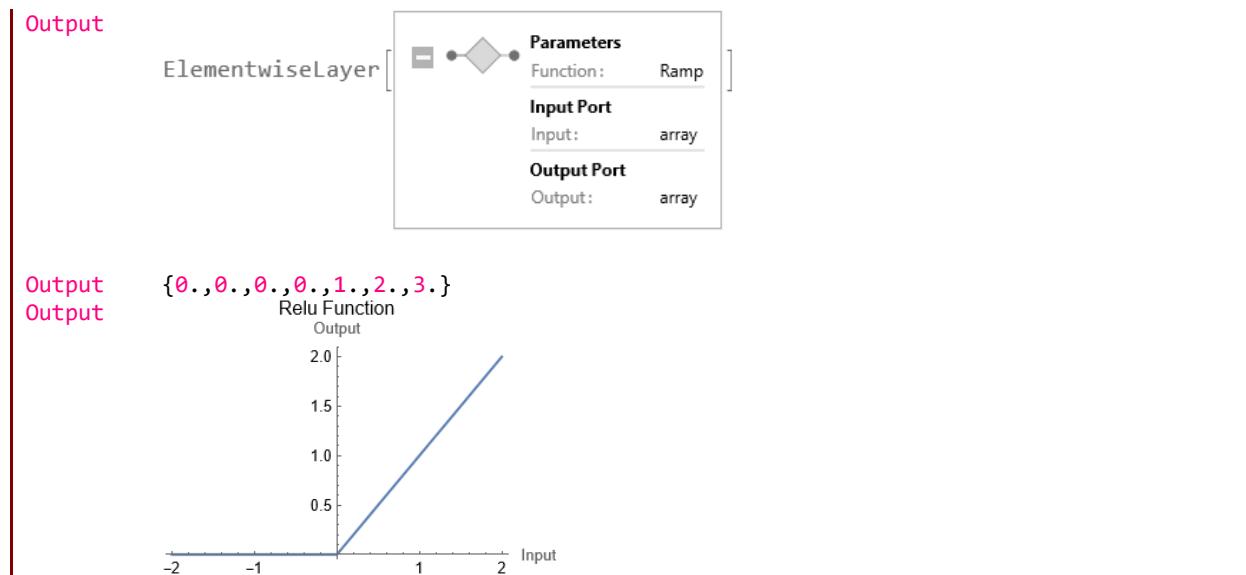
Mathematica Code 4.19

```
Input (* Create an ElementwiseLayer that computes Relu of each input element: *)
ramp=ElementwiseLayer[Ramp]

(* Apply the layer to a single vector: *)
ramp[{-3,-2,-1,0,1,2,3}]

(* Apply the ElementwiseLayer to a range of values: *)
range=Range[-2,2,0.1];
resultVector=ramp/@range;

(* Plot the results: *)
ListLinePlot[
Transpose[{range,resultVector}],
AxesLabel->{"Input","Output"},
PlotLabel->"Relu Function",
ImageSize->250
]
```

**Mathematica Code 4.20**

Input (* This code illustrates the creation of an ElementwiseLayer designed for vectors of size 3, and it demonstrates how the layer operates on both a single vector and a batch of vectors, applying the specified activation function (Ramp): *)

```
(* Create an ElementwiseLayer that takes vectors of size 3: *)
elem=ElementwiseLayer[Ramp,"Input"]->3
(* Apply the layer to a single vector: *)
elem[{1,2,3}]
(* When applied, the layer will automatically thread over a batch of vectors: *)
elem[{{1,2,3},{-1,-2,-3}}]
```

Output

ElementwiseLayer [Parameters
Function: Ramp
Input Port
Input: vector(size: 3)
Output Port
Output: vector(size: 3)]

Output {1.,2.,3.}
Output {
{1.,2.,3.},
{0.,0.,0.}
}

Mathematica Code 4.21

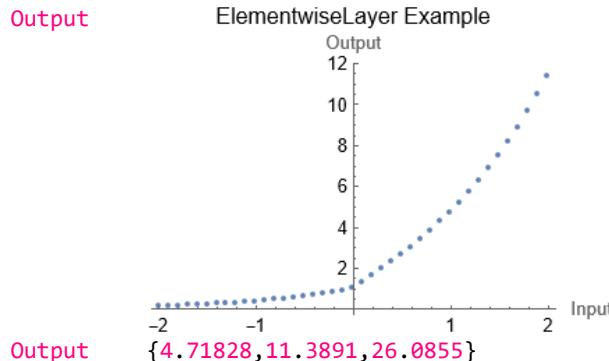
Input (* The code demonstrates the application of the ElementwiseLayer to specific vector and a range of values, and it visualizes the input-output relationship using a ListPlot: *)

```
(* Define an ElementwiseLayer with a custom function: *)
elem=ElementwiseLayer[2*Ramp[#]+Exp[#]&];
(* Apply the ElementwiseLayer to a specific vector: *)
resultVector=elem[{1,2,3}]
(* Create a range of values from -2 to 2 with a step size of 0.1: *)
range=Range[-2,2,0.1];
(* Generate data by applying the ElementwiseLayer to each value in the range: *)
data=Table[
  {range[[i]],elem[range[[i]]]},
```

```

    {i,1,Length[range]}
];
(* Plot the results using ListPlot: *)
ListPlot[
  data,
  ImageSize->250,
  PlotRange->All,
  AxesLabel->{"Input", "Output"},
  PlotLabel->"ElementwiseLayer Example"
]

```

**Mathematica Code 4.22**

Input (* In this example, the custom function includes operations like Ramp, subtraction, and logarithmic function. The ElementwiseLayer is then created using this custom function and applied to a specific input vector and the results are visualized using ListLinePlot: *)

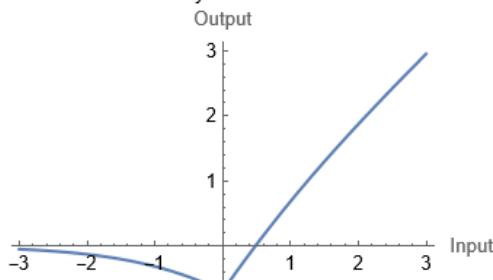
```
(* Define a custom elementwise function: *)
customFunction[x_]:=2*Ramp[x]-Log[1+Exp[x]]
```

```
(* Create an ElementwiseLayer using the custom function: *)
customElementwiseLayer=ElementwiseLayer[customFunction];
```

```
(* Apply the ElementwiseLayer to a range of values: *)
range=Range[-3,3,0.1];
resultVector=customElementwiseLayer/@range;
```

```
(* Plot the results: *)
ListLinePlot[
  Transpose[{range,resultVector}],
  AxesLabel->{"Input", "Output"},
  PlotLabel->"ElementwiseLayer Custom Function",
  ImageSize->250
]
```

Output ElementwiseLayer Custom Function



Mathematica Code 4.23

```

Input      (* This example demonstrates how you can use ElementwiseLayer with a combination
          of elementary mathematical functions (Ramp and Tanh) and a logical operation (If
          statement) to create a custom elementwise transformation for a given input array.
          The results are visualized using ListLinePlot: *)

(* Define a simple array: *)
inputArray={-2,-1,0,1,2};

(* Apply ElementwiseLayer with a combination of mathematical functions and if
statement: *)
elementwiseLayer=ElementwiseLayer[If[#>0,Ramp[#],Tanh[#]]]&;

(* Apply the layer to the input array: *)
resultArray=elementwiseLayer[inputArray];

(* Display the results: *)
Column[
{
  "Input Array:",MatrixForm[inputArray],
  "Result Array:",MatrixForm[resultArray]
}
]

(* Apply the ElementwiseLayer to a range of values: *)
range=Range[-3,3,0.1];
resultVector=elementwiseLayer/@range;

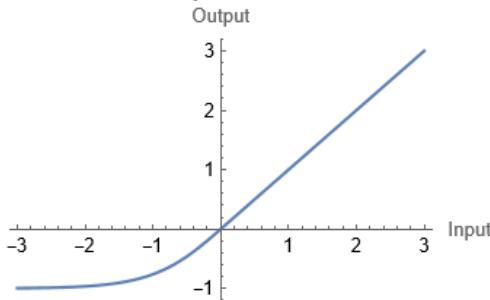
(* Plot the results: *)
ListLinePlot[
  Transpose[{range,resultVector}],
  AxesLabel->{"Input","Output"},
  PlotLabel->"ElementwiseLayer Custom Function",
  ImageSize->250
]

Output   {
  {Input Array:},
  {{{
    {-2},
    {-1},
    {0},
    {1},
    {2}
  }}},
  {Result Array:},
  {{{
    {-0.964028},
    {-0.761594},
    {0.},
    {1.},
    {2.}
  }}}
}

```

Output

ElementwiseLayer Custom Function

**Mathematica Code 4.24**

Input

```
(* The code defines a custom linear layer in a neural network with specified weights
and biases, then applies it to randomly generated input data. Subsequently, an
elementwise activation layer with the Rectified Linear Unit (ReLU) activation
function is created and applied to the output of the custom linear layer. This
process effectively demonstrates the construction of a custom layer in a neural
network architecture, showcasing how to set custom parameters and apply activation
functions to manipulate the output of the network: *)

(* Specify a custom weight matrix and bias vector: *)
customWeights={{1,2,3},{4,5,6},{4,3,2}}; (* 2x3 matrix. *)
customBiases={0.5,-1,0.1}; (* 1D vector with length equal to the number of
outputs. *)

(* Create a LinearLayer with custom weights and biases: *)
customLinearLayer=LinearLayer[3,"Input"]->3,"Weights"->customWeights,"Biases"->customBiases]

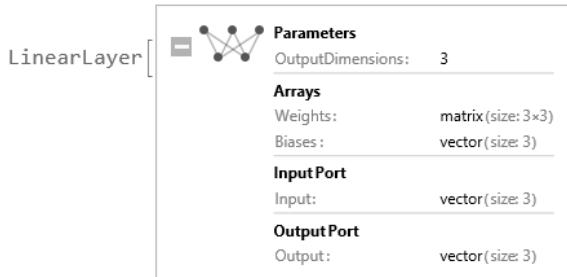
(* Generate some example input data: *)
inputData=RandomReal[1,{3,3}]

(* Apply the custom linear layer to the input data: *)
preactivation=customLinearLayer[inputData]

(* Create an ElementwiseLayer that takes vectors of size 3: *)
activation=ElementwiseLayer[Ramp,"Input"]->3

(* When applied, the layer will automatically thread over a batch of vectors: *)
activation[preactivation]
```

Output



Output

```
{  
{0.271247,0.129702,0.564545},  
{0.234738,0.846121,0.799014},  
{0.280505,0.54826,0.145928}  
}
```

Output

{

```

{2.72429,4.12077,2.70318},
{4.82402,8.96364,5.17534},
{2.31481,3.73888,3.15865}
}

Output
ElementwiseLayer[ Parameters
  Function: Ramp
  Input Port
  Input: vector (size: 3)
  Output Port
  Output: vector (size: 3)
]

Output {{2.72429,4.12077,2.70318},{4.82402,8.96364,5.17534},{2.31481,3.73888,3.15865}}

```

Mathematica Code 4.25

```

Input (* The code aims to demonstrate the initialization and application of a custom
linear layer within a neural network architecture. Initially, a linear layer is
created with custom weights and biases, followed by the generation of random input
data to which the layer is applied, resulting in pre-activation values. An element-
wise activation function, specifically the hyperbolic tangent function (Tanh), is
then applied to these pre-activation values to obtain post-activation values.
Additionally, histograms are generated to visually represent the distributions of
the initialized weights, pre-activation data, and post-activation data. This code
not only illustrates the process of initializing and utilizing custom layers but
also provides insights into the distributions of values at different stages of the
neural network's computation, aiding in understanding its behavior and performance:
*)

(* Specify the dimensions for the LinearLayer: *)
inputSize=200;
outputSize=3;

(* Specify a custom weight matrix and bias vector: *)
customWeights=RandomVariate[
  NormalDistribution[0,1],
  {outputSize,inputSize}
];

customBiases=RandomReal[0,outputSize];

(* Create a LinearLayer with custom weights and biases: *)
randomlyInitializedLinearLayer=LinearLayer[
  outputSize,
  "Input"->inputSize,
  "Weights"->customWeights,
  "Biases"->customBiases
]

(* Generate some example input data: *)
inputData=RandomReal[1,{1000,inputSize}];

(* Apply the randomly initialized linear layer to the input data: *)
preactivation=randomlyInitializedLinearLayer[inputData];
preActivationData=Flatten[preactivation];

(* Create an ElementwiseLayer that takes vectors of size 3: *)
activationFunction=ElementwiseLayer[Tanh,"Input"->outputSize]
postactivation=activationFunction[preactivation];
postActivationData=Flatten[postactivation];

extractWeights=NetExtract[randomlyInitializedLinearLayer,"Weights"];

```

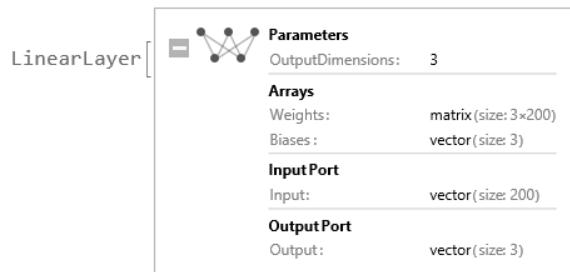
```

extractWeightsData=Flatten[extractWeights];

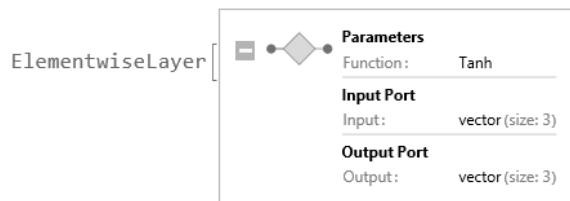
(* Create a histogram for the Initialized Weights, Preactivation Data and
Postactivation Data: *)
Table[
Histogram[
index[[2]],
Automatic,
"Probability",
PlotLabel->Style[Row[{ "Histogram of ",index[[1]]}]],
AxesLabel->{index[[1]],"Probability"},
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250
],
{index,
{
 {"Weights",extractWeightsData},
 {"Preactivations",preActivationData},
 {"Postactivations",postActivationData}
 }
]
]

```

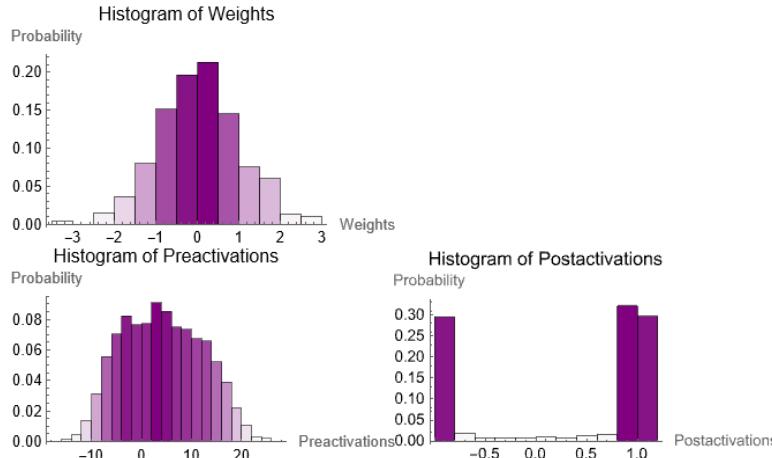
Output



Output



Output



Unit 4.3

Containers: NetChain and NetGraph

In Mathematica, NNs are typically built by combining multiple layers together to form a network architecture. While a single NN layer can perform basic transformations on its input data, it's usually not sufficient for more complex tasks. By combining multiple layers together, you can create deeper and more powerful NNs capable of learning intricate patterns and making more accurate predictions.

- The **NetChain** container is commonly used to chain layers together sequentially, where the output of one layer serves as the input to the next layer. This sequential structure is suitable for many standard NN architectures.
- However, there are cases where you may need more complex connectivity patterns between layers. For example, you may want to create skip connections, recurrent connections, or other non-sequential connections within the network. In such cases, the **NetGraph** container is more appropriate. **NetGraph** allows you to define a graph-like structure where nodes represent different layers or operations, and edges represent the flow of data between them. This flexibility enables you to construct a wide variety of NN architectures tailored to your specific needs.

NetChain

NetChain[{layer1,layer2,...}]

specifies a neural net in which the output of layer i is connected to the input of layer $i+1$.

NetChain[<"name1" -> layer1, "name2" -> layer2, ... |>]

specifies a net consisting of a chain of explicitly named layers.

- The input data is provided to the **NetChain**, and it is passed into the first layer specified in the chain. The input data then undergoes sequential processing through each layer in the chain, with the output of one layer serving as the input for the next layer. The final output of the **NetChain** is taken from the output of the last layer in the chain, providing the result of the network's computation.
- If the first layer has multiple input ports or the last layer has multiple output ports, the **NetChain** will have the same input or output ports respectively, allowing for more complex network architectures.
- **NetChain[...][data]** gives the result of applying the net to data.
- **Normal[NetChain[...]]** returns a list or association of the layers used to construct the chain.
- **NetChain[...][[spec]]** extracts the layer specified by **spec** from the net. This enables users to access and manipulate individual layers within the network.
- The **StandardForm** of **NetChain** provides a summary of the layers in the chain, along with the array dimensions of the output of each layer. Clicking on a layer in the chain reveals more detailed information about that specific layer, aiding in network inspection and debugging.
- Note that the layers containing learnable parameters appear in red, indicating that they require initial values before the net can be applied to an input.
- The overall input and output array shapes for the chain can be explicitly specified using the **"Input" -> shape** and **"Output" -> shape** options for **NetChain**, providing more control over the network's input and output dimensions.

Possible forms for shape include:

"Real"
"Integer"
Restricted["Integer",n]

a single real number

a single integer

an integer between 1 and n

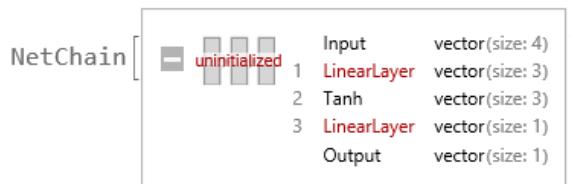
Restricted["Integer",{m,n}]	an integer between m and n
n	a vector of length n
{n1,n2,...}	an array of dimensions $n_1 \times n_2 \times \dots$
"Varying"	a vector whose length is variable.
{"Varying",n2,n3,...}	an array whose first dimension is variable and remaining dimensions are $n_2 \times n_3 \times \dots$

- Any lengths `ni` given as Automatic are inferred from the structure of the chain, simplifying network construction by automatically determining certain dimensions based on the network's architecture.
- `NetChain[...][data,...,opts]` specifies that options should be used when applying the network to data, allowing for additional customization during network evaluation. Possible options include: `BatchSize`, `NetEvaluationMode`, `RandomSeeding`, `TargetDevice` and `WorkingPrecision`.
- `EdgeList[NetChain[...]]` returns the list of connections in the network, providing insights into the network's connectivity and structure.
- `net[data,NetPort[oport]]` can be used to obtain the value of the net at `oport` when it is applied to the specified data. `oport` must refer to an output port, a subnet of the net, or a layer.

Mathematica Code 4.26

```
Input (* The code creates a neural network with three layers, where the first and third
       layers are linear transformations and the second layer applies a Tanh activation
       function. The input size is specified as 4, providing a complete description of the
       network architecture: *)

net=NetChain[
  {
    LinearLayer[3],(* Linear layer with 3 output nodes.*)
    ElementwiseLayer[Tanh],(* Tanh activation function.*)
    LinearLayer[1](* Linear layer with 1 output node.*)
  },
  "Input"->4
]
```

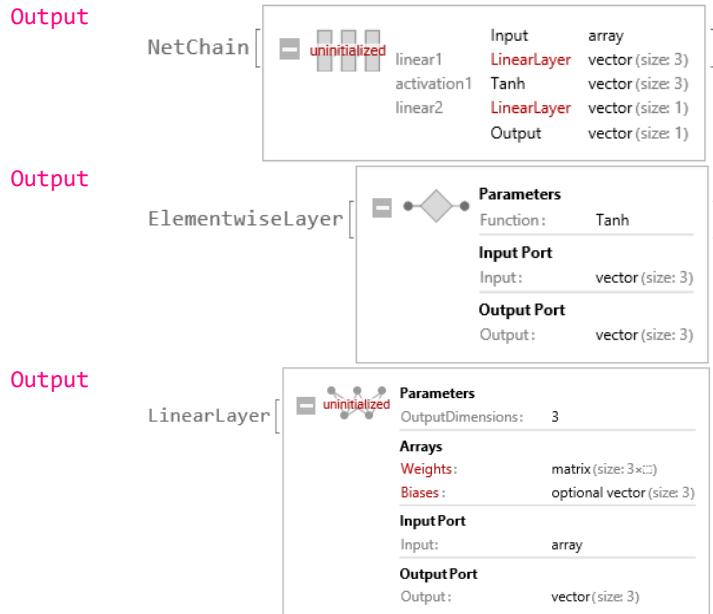
Output**Mathematica Code 4.27**

```
Input (* The code constructs a neural network with explicitly named layers, demonstrates
       how to extract a specific layer by name using NetExtract, and shows another method
       using Part syntax for extracting a layer. Explicitly naming layers in a neural
       network is a good practice, as it enhances code readability and makes it easier to
       reference and manipulate specific layers later in the code or during analysis. The
       use of an association (<|...|>) in the NetChain construction is a nice touch. It
       allows you to associate each layer with a meaningful name, making it clear which
       layer is being referred to in subsequent operations: *)

(* Construct a neural network with explicitly named layers: *)
net=NetChain[<|
  "linear1"->LinearLayer[3],(* Linear layer with 3 output nodes. *)
  "activation1"->ElementwiseLayer[Tanh],(* Tanh activation function. *)
  "linear2"->LinearLayer[1] (* Linear layer with 1 output node. *)
|>]
```

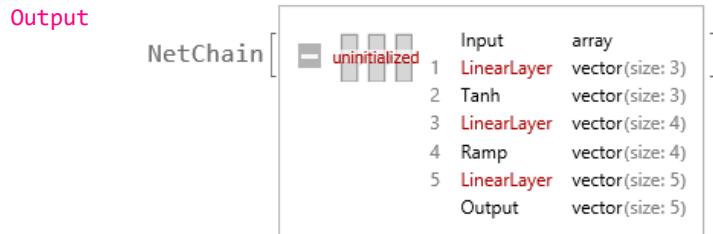
```
(* Extract the activation function by name: *)
secondLayer=NetExtract[net,"activation1"]

(* Use Part syntax to extract the first layer: *)
firstLayer=net[["linear1"]]
```

**Mathematica Code 4.28**

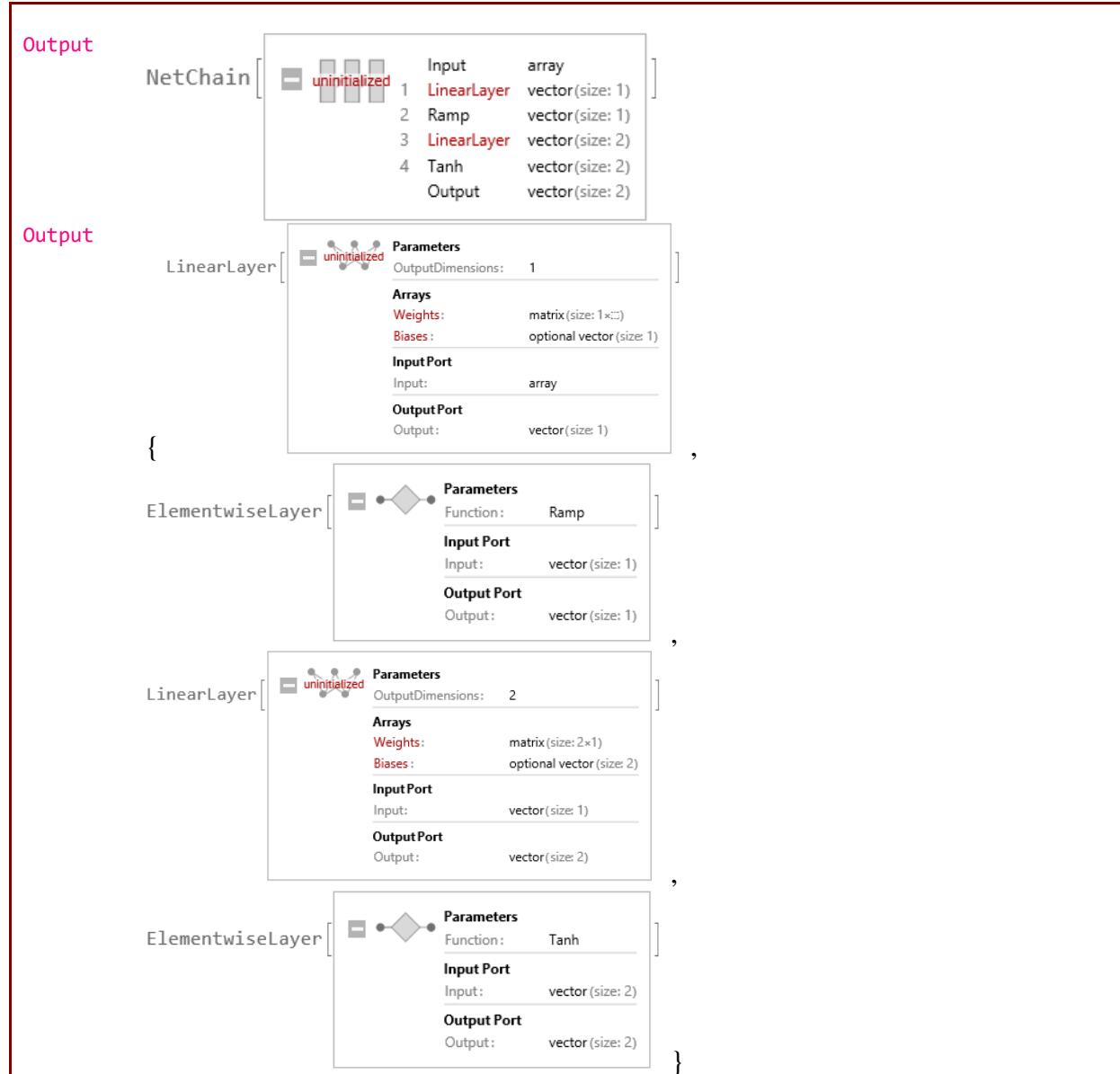
Input (* The code uses a concise syntax to construct a neural network. The network consists of layers specified in a list. Each element in the list represents a layer or a transformation applied to the data. The numeric value "3" indicates the number of nodes in the first layer. The "Tanh" symbol represents the ElementwiseLayer with the hyperbolic tangent (Tanh) activation function applied to its output. The numeric value "4" indicates the number of nodes in the second layer. The "Ramp" symbol represents the activation function applied by the ElementwiseLayer in the second position. The numeric value "5" indicates the number of nodes in the final layer: *)

```
NetChain[{3,Tanh,4,Ramp,5}]
```

**Mathematica Code 4.29**

Input (* The code constructs a neural network using the NetChain function with a list of layers. In this case, the network has a structure of 1 node, followed by Ramp activation, 2 nodes, and Tanh activation. The code showcases how to extract the layers of a neural network using the Normal function: *)

```
net=NetChain[{1,Ramp,2,Tanh}]
Normal[net]
```

**Mathematica Code 4.30**

```

Input (* The code defines a neural network in Wolfram Mathematica, net, comprising three
       element-wise layers: hyperbolic tangent (Tanh), cosine (Cos), and sine (Sin).
       Through various evaluations, it showcases the extraction of specific layer outputs,
       including the first layer (Tanh), the first two layers, and all layers, when given
       input data: *)

(* Create a NetChain: *)
net=NetChain[
  {
    ElementwiseLayer[Tanh],
    ElementwiseLayer[Cos],
    ElementwiseLayer[Sin]
  }
]

(* Obtain the output of the first layer of the net when applied to data: *)

```

```

net[{-1.2,3.1},NetPort[1,"Output"]]

(* This is equivalent to: *)
Tanh[{-1.2,3.1}]

(* Obtain the output of the first two layers of the net: *)
net[{-1.2,3.1},{NetPort[1,"Output"],NetPort[2,"Output"]}]

(* Obtain the outputs of all layers in the net: *)
net[{-1.2,3.1},NetPort[All,"Output"]]

(* Values extracts the numerical values of all outputs obtained in the previous
step: *)
Values[net[{-1.2,3.1},NetPort[All,"Output"]]]

```

Output



Output {-0.833655, 0.995949}

Output {-0.833655, 0.995949}

Output <|NetPort[{1,Output}]->{-0.833655, 0.995949},NetPort[{2,Output}]->{0.672174, 0.543706}|>

Output <|NetPort[{1,Output}]->{-0.833655, 0.995949},NetPort[{2,Output}]->{0.672174, 0.543706},NetPort[{3,Output}]->{0.622689, 0.517311}|>

Output {{-0.833655, 0.995949}, {0.672174, 0.543706}, {0.622689, 0.517311}}

Mathematica Code 4.31

Input (* The code constructs, initializes, and analyzes a neural network comprising three layers: a linear layer with 3 output nodes, a Tanh activation layer, and another linear layer with 1 output node. The network is initialized with random weights using the NetInitialize function, making it ready for training or evaluation. A graphical summary of the network's architecture is displayed using Information[net,"SummaryGraphic"]. Through visualization techniques, such as histograms, the code aims to offer insights into the network's architecture and behavior. It extracts individual layers for in-depth analysis and examines the distribution of data output from each layer, both sequentially and utilizing NetPort, providing a comprehensive understanding of the network's functioning and the effects of its activation functions. A 3D plot is generated to visualize the results of the initialized network on a range of input values: *)

```
(* Specify the dimension for Input: *)
inputDimension=2;
```

```
(* Construct a chain consisting of three layers and specify that the input is a
length "inputSize" vector: *)
neuralNetwork=NetChain[
{
  LinearLayer[3,"Biases"->None],
  ElementwiseLayer[Tanh],
```

```

    LinearLayer[1,"Biases"->None]
  },
 "Input"->inputDimension
]

(* Initialize the net with random weights, making it ready for training or
evaluation: *)
neuralNetwork=NetInitialize[neuralNetwork]

(* Display a graphical summary of the network's architecture, showing the
structure of layers and their connections: *)
Information[neuralNetwork,"SummaryGraphic"]

(*Extract individual layers for further analysis*)
firstLayer=NetExtract[neuralNetwork,1]
secondLayer=NetExtract[neuralNetwork,2]
thirdLayer=NetExtract[neuralNetwork,3]

(* Extract and display the weights of layer 1 and layer 3: *)
NetExtract[neuralNetwork,{1,"Weights"}]//Normal
NetExtract[neuralNetwork,{3,"Weights"}]//Normal

(* Apply the net to an input vector: *)
inputData=RandomReal[1,{1000,inputDimension}];

outputFirstLayer=firstLayer[inputData];
outputSecondLayer=secondLayer[outputFirstLayer];
outputThirdLayer=thirdLayer[outputSecondLayer];

(* Generate histograms to visualize the distribution of data output from first,
second , and third Layers, applying the layers sequentially: *)
Table[
 Histogram[
  Flatten[layer[[2]]],
  Automatic,
  "Probability",
  PlotLabel->Style[Row[{ "Histogram of Output Data (sequential):",layer[[1]]}],],
  AxesLabel->{layer[[1]],"Probability"}, 
  ColorFunction->Function[{height},Opacity[height]],
  ChartStyle->Purple,
  ImageSize->300
 ],
 {layer,{
  {"FirstLayer",outputFirstLayer},
  {"SecondLayer",outputSecondLayer},
  {"ThirdLayer",outputThirdLayer}
 }
]
]

(* Output of all layers using NetPort: *)
outputOfAllLayers=Values[
 Table[
  neuralNetwork[inputData[[i]],NetPort[All,"Output"]],
  {i,1,1000}
 ]
];
];

(* Extract output of each layer using NetPort: *)
outputFirstLayerNetPort=outputOfAllLayers[[All,1]];

```

```

outputSecondLayerNetPort=outputOfAllLayers[[All,2]];
outputThirdLayerNetPort=outputOfAllLayers[[All,3]];

(* Create histograms for the output data of each layer using NetPort: *)
Table[
Histogram[
Flatten[layer[[2]]],
Automatic,
"Probability",
PlotLabel->Style[Row[{ "Histogram of Output Data (NetPort):",layer[[1]]}]],
AxesLabel->{layer[[1]],"Probability"},
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->300
],
{layer,{ "FirstLayer ",outputFirstLayerNetPort},
 {"SecondLayer",outputSecondLayerNetPort},
 {"ThirdLayer",outputThirdLayerNetPort}
 }
]
]

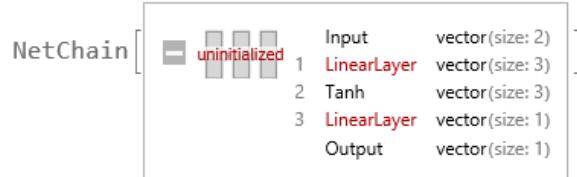
(* Generate x and y values: *)
xValues=Table[x,{x,-2,2,0.1}];
yValues=Table[y,{y,-2,2,0.1}];

(* Create data by evaluating the initialized neuralNetwork function for all
combinations of x and y:*)
trainingData =Flatten[
Table[
{x,y,neuralNetwork[{x,y}][[1]]},
{x,xValues},
{y,yValues}
],1];

(* Plot the results in a 3D plot for the initialized net: *)
ListPlot3D[
trainingData,
PlotLabel->"3D plot for the Initialize Net",
ColorFunction->"Rainbow",
ImageSize->250
]

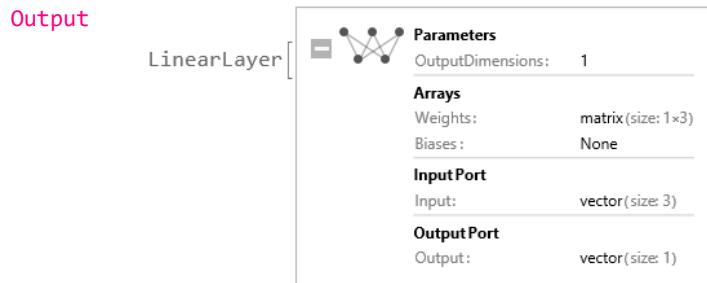
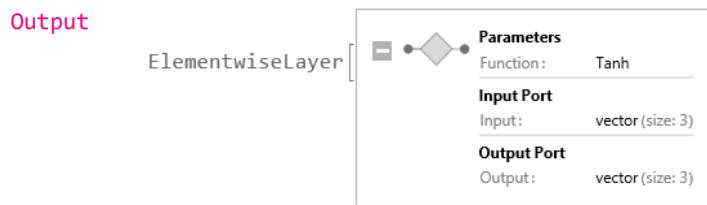
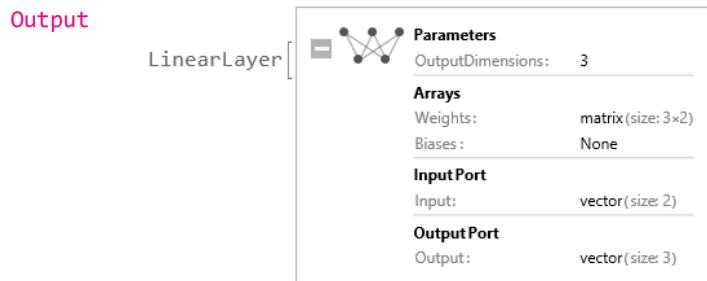
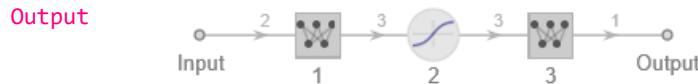
```

Output



Output

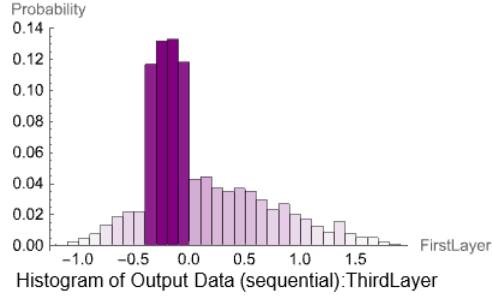




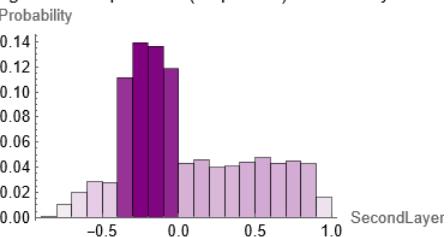
Output `{{-0.359447, -0.0499231}, {-1.12706, 1.08654}, {1.89365, -0.93114}}`

Output `{{-0.631947, 0.082531, 0.06014}}`

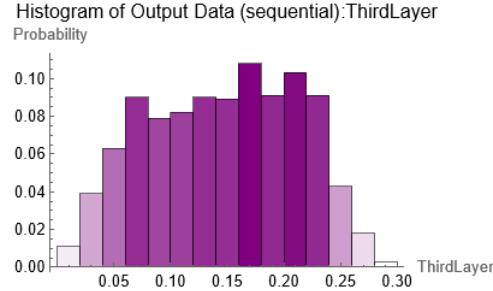
Output Histogram of Output Data (sequential):FirstLayer

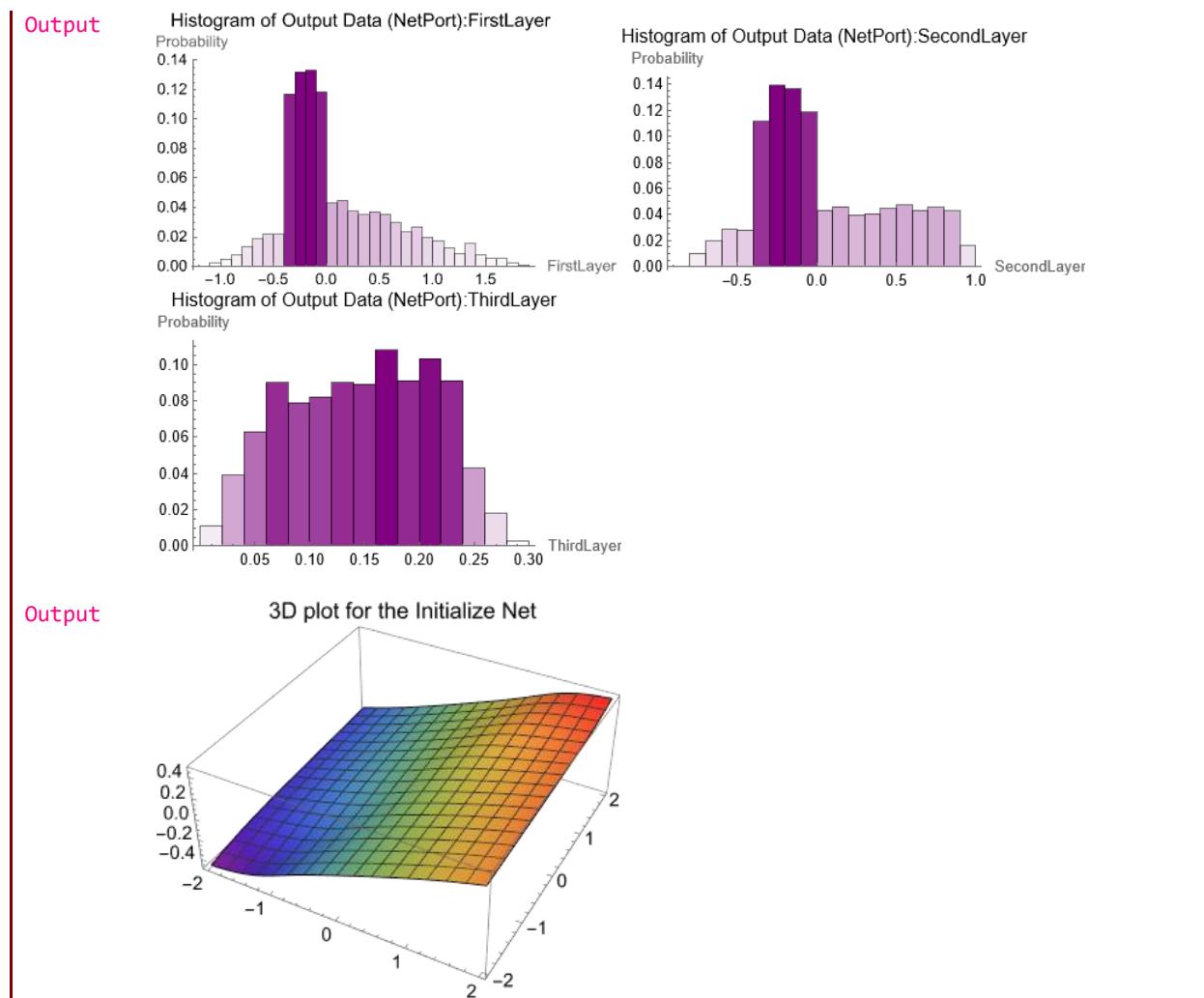


Histogram of Output Data (sequential):SecondLayer



Histogram of Output Data (sequential):ThirdLayer





NetGraph

`NetGraph[{layer1,layer2,...},{m1->n1,m2->n2,...}]`

specifies a neural net defined by a graph in which the output of layer m_i is given as input to layer n_i .

`NetGraph[<|"name1"->layer1,"name2"->layer2,... ->,{ "namem1"->"namen1",...}]`

specifies a net with explicitly named layers.

`NetGraph[layer]`

converts a layer or a `NetChain` into an equivalent minimal `NetGraph`.

- In Mathematica, `NetGraph` is a powerful function used for building and representing complex NN architectures as directed acyclic graphs (DAGs). It allows for combining various NN layers and operations
- When constructing a `NetGraph` in Mathematica, you can create input or output ports for the entire graph by specifying `NetPort["input"] ->` or `-> NetPort["output"]` respectively in the list of connections.
- In Mathematica's `NetGraph`, you can specify a linear chain of connections within the graph using the syntax `layer1 -> layer2 -> ... -> layerN`. This notation establishes a sequential connection between each layer, where layer1 is connected to layer2, layer2 is connected to layer3, and so on, until layerN. This linear

chain notation is useful for defining simple feedforward architectures or for building blocks of more complex networks, such as CNNs or RNNs.

- If the nth layer, or a layer named "layer", has more than one input or output port, you can disambiguate them by using the syntax `NetPort[n, "port"]` or `NetPort["layer", "port"]`. By using `NetPort` with the appropriate layer index or name, you can explicitly specify which input or output port of a layer you want to connect in the `NetGraph`, allowing for greater control and flexibility in designing complex NN architectures.
- If you leave one or more input or output ports of any layers unconnected within a `NetGraph`, they will automatically become ports of the entire `NetGraph`. This behavior allows for more flexibility in connecting multiple `NetGraph` instances or integrating them into larger network architectures. This behavior facilitates the construction of more modular and reusable network architectures in Mathematica, allowing you to easily combine different components or subnetworks into larger and more complex models.
- In Mathematica's `NetGraph`, you can mute some output ports of layers by setting them to `None`. This is particularly useful when you want to disable certain outputs of a layer within the graph.
- If the NN (`NetGraph`) has a single input port, you can directly apply the network to input data using the syntax `NetGraph[...][data]`. This applies the network to the input data and produces the output.
- If the network has multiple input ports, you provide data to each port using an association syntax like `NetGraph[...] [<|port1 -> data1, ... |>]`. This allows you to specify input data for each input port individually.
- If the network has a single output port, you can still use the syntax `NetGraph[...][data]` to obtain the output. The output will be the result for that single output port.
- For a net with multiple output ports, `NetGraph[...][data]` gives an association of the outputs for all ports.
- The `StandardForm` of `NetGraph` shows the connectivity of layers in the graph and annotates edges with the dimensions of the array that the edge represents. Clicking a layer or a port in the graph shows more information about that layer or port.
- `Normal[NetGraph[...]]` returns a list or association of the layers used to construct the graph. `EdgeList[NetGraph[...]]` returns the list of connections in the graph.
- `NetGraph[...][[spec]]` extracts the layer specified by spec from the net.

NetPort

`NetPort["port"]`

represents the specified input or output port for a complete net.

`NetPort[{n, "port"}]`

represents the specified port for layer number n in a NetGraph or similar construct.

`NetPort[{name, "port"}]`

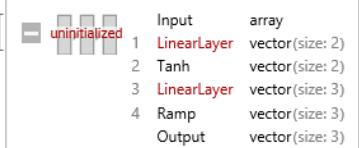
represents the specified port for the layer with the specified name.

Mathematica Code 4.32

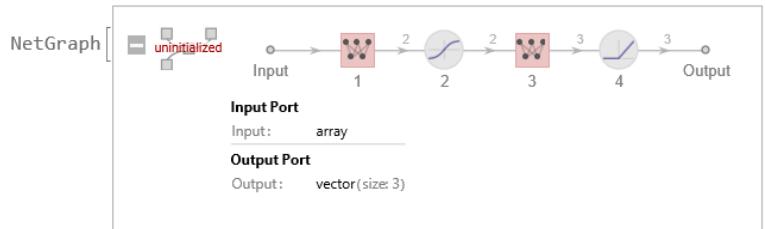
Input (* The code demonstrates a straightforward way to convert a `NetChain` into a `NetGraph` in Mathematica, providing flexibility in working with different neural network structures: *)

```
chain=NetChain[{2,Tanh,3,Ramp}]
NetGraph[chain]
```

Output

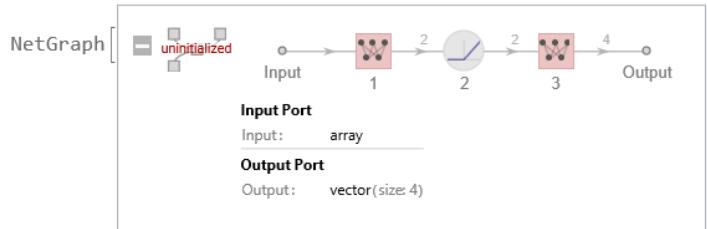
```
NetChain[
  Input      array
  1 LinearLayer vector(size: 2)
  2 Tanh       vector(size: 2)
  3 LinearLayer vector(size: 3)
  4 Ramp       vector(size: 3)
  Output     vector(size: 3)]
```

Output

**Mathematica Code 4.33**

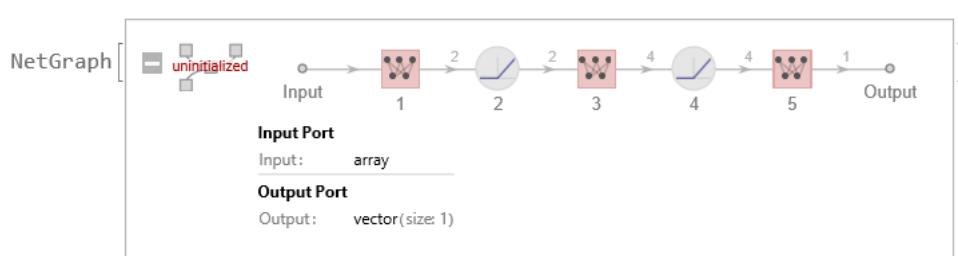
```
Input (* The code defines a simple neural network with a linear chain structure, consisting
      of two linear layers and one ReLU activation layer. The connections between the
      layers are specified in a sequential manner: *)
NetGraph[
{
  (* A linear layer with 2 neurons: *)
  LinearLayer[2],
  (* An element-wise layer with a ReLU activation function: *)
  ElementwiseLayer[Ramp],
  (* Another linear layer with 4 neurons: *)
  LinearLayer[4]
},
(* This list specifies the connections between the layers: *)
{1->2,2->3}
]
```

Output

**Mathematica Code 4.34**

```
Input (* By omitting explicit layer names and using consecutive edge rules, this code
      provides a concise representation of a neural network with a linear chain structure.
      The size of the linear layers is specified directly in the layer list. This style
      can be convenient for simple network structures where explicit names are not
      necessary: *)
NetGraph[{2,Ramp,4,Ramp,1},{1->2->3->4->5}]
```

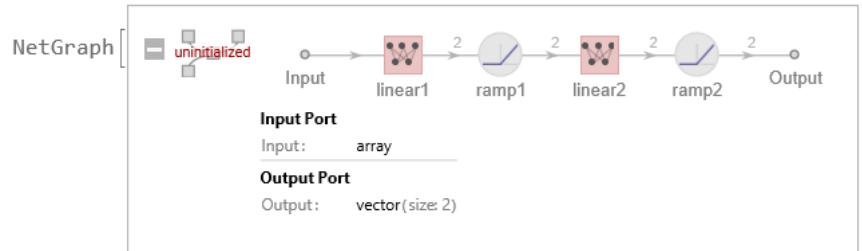
Output



Mathematica Code 4.35

Input (* The code showcases how to construct a NetGraph with explicitly named layers, providing a more human-readable and expressive way to define neural network architectures in Mathematica. The layer names can be used for better documentation and referencing within the code: *)

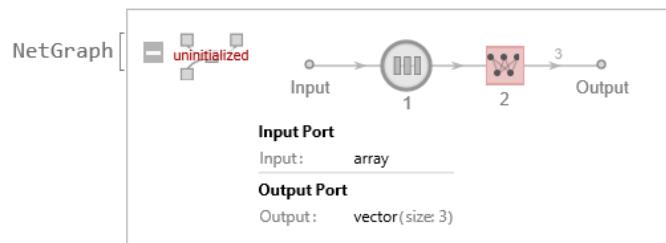
```
NetGraph[<|
  "linear1" -> 2,
  "ramp1" -> Ramp,
  "linear2" -> 2,
  "ramp2" -> Ramp
 |>,
 {"linear1" ->"ramp1" ->"linear2" ->"ramp2"}]
```

Output**Mathematica Code 4.36**

Input (* The code shows that a NetChain object, representing a sequence of layers, can be used as a single layer within a larger NetGraph structure. This allows for flexible and modular construction of neural network architectures: *)

```
(* Define a NetChain with Ramp and LogisticSigmoid layers: *)
chain=NetChain[{Ramp,LogisticSigmoid}];

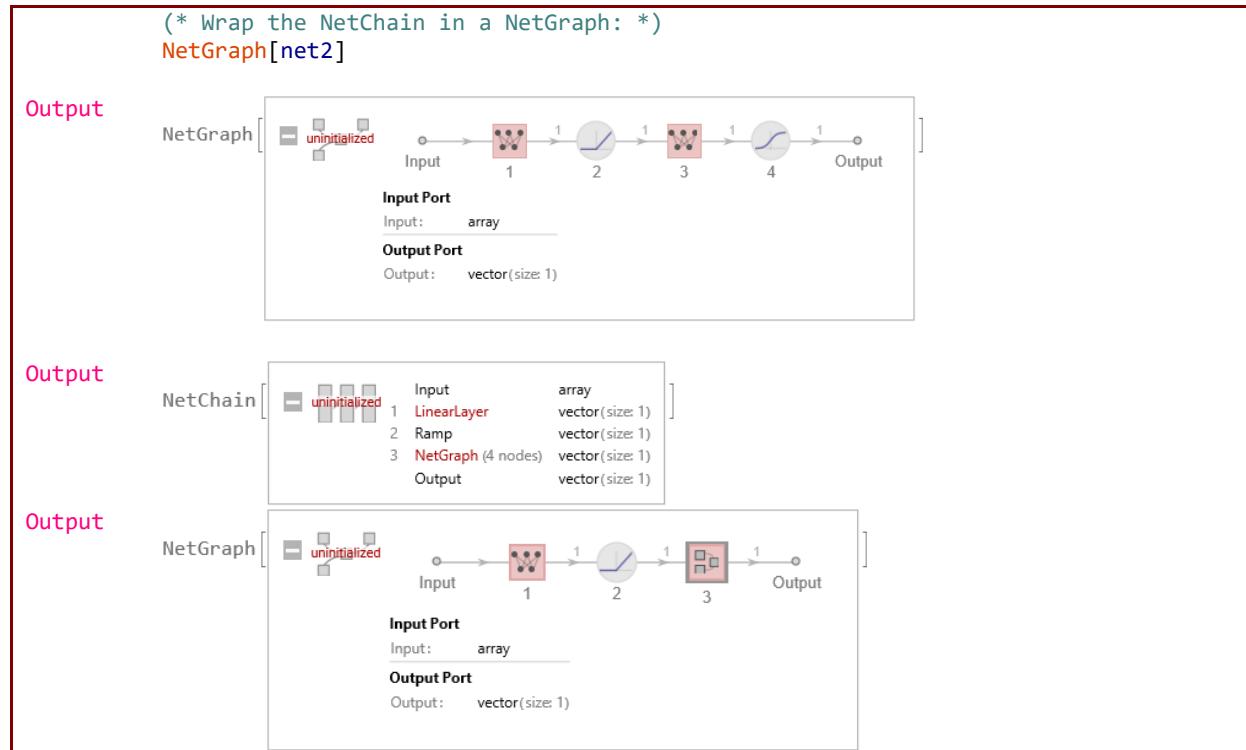
(* Create a NetGraph with the previously defined NetChain and a LinearLayer: *)
NetGraph[{chain,LinearLayer[3]},{1->2}]
```

Output**Mathematica Code 4.37**

Input (* The code showcases the combination of NetGraph and NetChain structures, demonstrating the versatility and composability of these constructs in defining neural network architectures in Mathematica. NetGraph objects with one input and one output can be used as layers inside NetChain objects: *)

```
(* Define a NetGraph with one input and one output: *)
net=NetGraph[{1,Ramp,1,LogisticSigmoid},{1->2->3->4}]

(* Create a NetChain with ElementwiseLayer[Ramp] and the previously defined NetGraph: *)
net2=NetChain[{1,ElementwiseLayer[Ramp],net}]
```

**Mathematica Code 4.38**

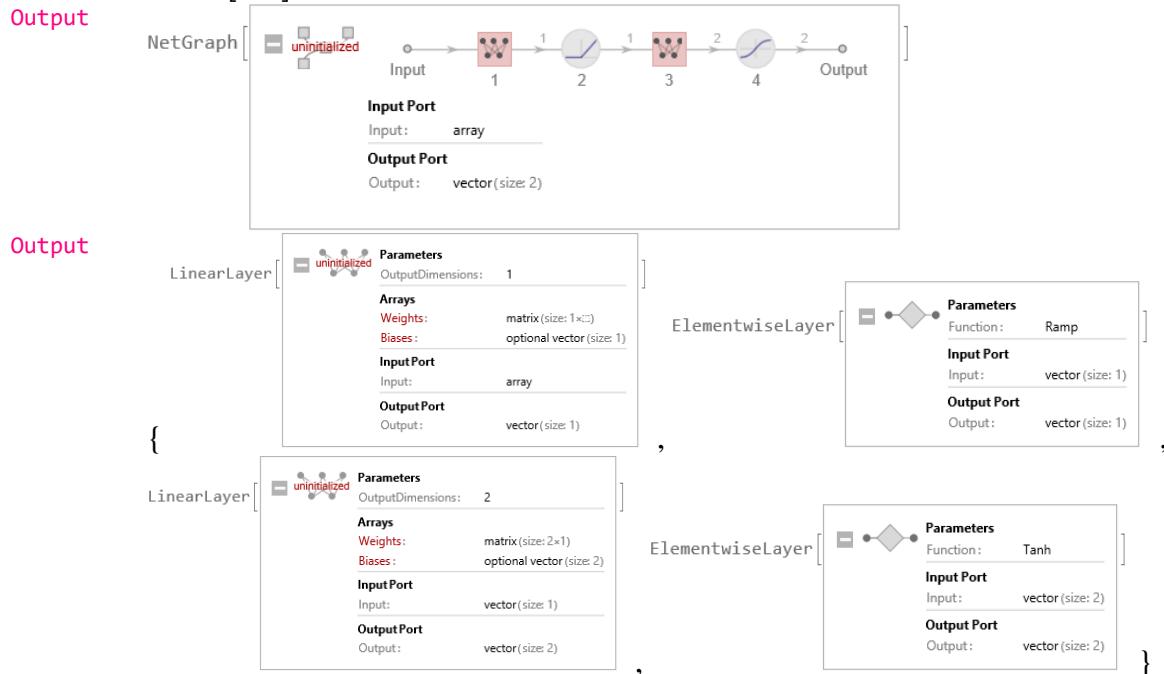
Input

```
(* In this example, the NetGraph function is used to create a neural network. It takes two arguments:the list of layers {1,Ramp,2,Tanh} and the connections between layers {1->2->3->4}: *)
```

```
net=NetGraph[{1,Ramp,2,Tanh},{1->2->3->4}]
```

(* The layers used to construct a NetGraph can be extracted using Normal: *)

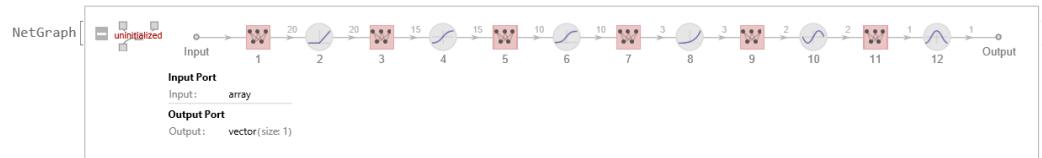
```
Normal[net]
```



Mathematica Code 4.39

Input (* In this example, each LinearLayer is followed by a different activation function. The activation functions include ReLU, hyperbolic tangent (Tanh), logistic sigmoid, exponential, sine, and cosine: *)

```
net=NetGraph[
 {
   (* ReLU activation: *)
   LinearLayer[20],Ramp,
 
   (* Hyperbolic tangent activation: *)
   LinearLayer[15],Tanh,
 
   (* Logistic sigmoid activation: *)
   LinearLayer[10],LogisticSigmoid,
 
   (* Exponential activation: *)
   LinearLayer[3],Exp,
 
   (* Sine activation: *)
   LinearLayer[2],Sin,
 
   (* Cosine activation: *)
   LinearLayer[1],Cos
 },
 {1->2->3->4->5->6->7->8->9->10->11->12}]
```

Output**Mathematica Code 4.40**

Input (*Construct a net graph with an operation requiring three inputs:*)

(* The code constructs a neural network graph using Mathematica's NetGraph function. It consists of five layers, where the first and third layers each have three neurons and the second and fourth layers apply the Ramp activation function. The fifth layer performs addition. Input data is supplied through a port named "Input". Connections between layers are established such that the output of the first layer is connected to the input of the second layer, the output of the third layer is connected to the input of the fourth layer, and the input port "Input", along with the outputs of the second and fourth layers, is connected to the fifth layer. This architecture enables the network to perform an operation requiring three inputs and produce an output based on their combination: *)

```
NetGraph[
 (*Define the layers of the net graph: *)
 {3,Ramp,3,Ramp,Plus},
 
 (*Define the connections between layers: *)
 {
   (* The output of the first layer is connected to the input of the second layer: *)
   1->2,
 
   (* The output of the third layer is connected to the input of the fourth layer: *)
   3->4
 }]
```

3->4,

```
(* The input port named "Input", along with the outputs of the second and
fourth layers, are all connected to the fifth layer using curly braces {}: *)
{NetPort["Input"],2,4}->5
}
]
```

Output

NetGraph[
 Input Port
 Input: vector(size: 3)
 Output Port
 Output: vector(size: 3)

Mathematica Code 4.41

```
Input (* Construct a net graph with multiple inputs: *)

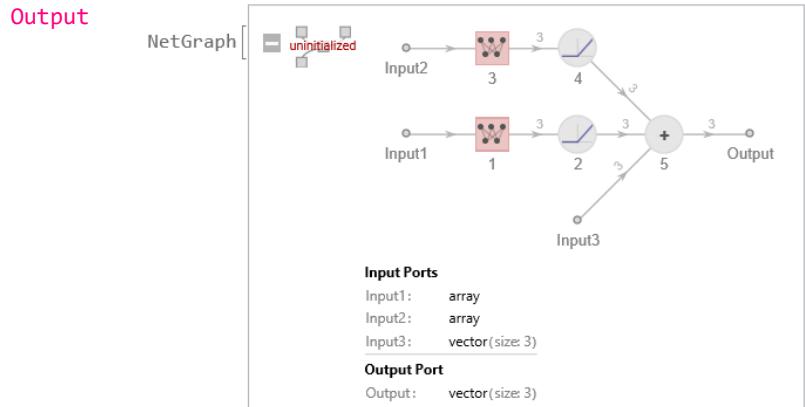
(* The NetGraph constructs a neural network with multiple input ports, featuring
five layers. The first and third layers consist of three neurons each, while the
second and fourth layers apply the Ramp activation function. The fifth layer
performs addition. Input data is supplied through three different input ports
labeled "Input1", "Input2", and "Input3". Connections between layers are established
such that "Input1" is connected to the first layer, "Input2" to the third layer,
and "Input3" is jointly connected with the output of the second and fourth layers
to the fifth layer. This architecture enables the network to process multiple inputs
simultaneously, incorporating them into the network's operations before producing
an output: *)

NetGraph[
  (* Define the layers of the net graph: *)
  {3,Ramp,3,Ramp,Plus},

  (* Define the connections between layers: *)
  {
    (* Connects "Input1" port to the first layer (index 1), and output of the first
    layer to the second layer (index 2): *)
    NetPort["Input1"]->1->2,

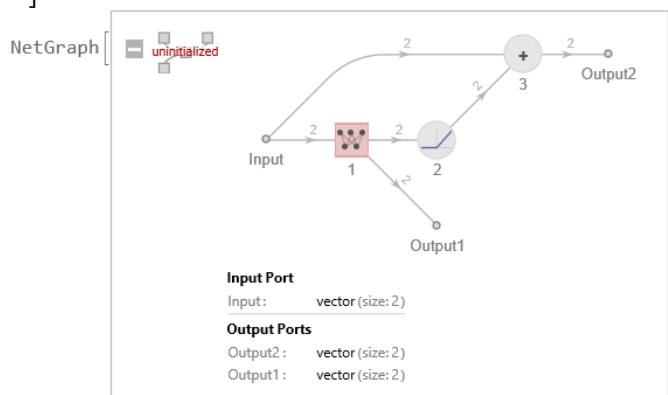
    (* Connects "Input2" port to the third layer (index 3), and output of the third
    layer to the fourth layer (index 4): *)
    NetPort["Input2"]->3->4,

    (*Connects "Input3" port and output of the second layer (index 2), and output
    of the fourth layer (index 4) to the fifth layer (index 5): *)
    {NetPort["Input3"],2,4}->5
  }
]
```

**Mathematica Code 4.42**

Input

```
(* Construct a net graph with multiple outputs: *)
(* The NetGraph constructs a neural network with multiple output ports, consisting
of three layers. The first layer comprises 2 neurons, the second layer applies the
Ramp activation function, and the third layer performs addition. Input data is
supplied through a port named "Input", and connections between layers are
established such that the output of the first layer is connected to the input of
the second layer. Furthermore, input data, along with the output of the second
layer, is routed to the third layer, where the resulting output is connected to a
port named "Output2". Additionally, the output of the first layer is directly
connected to another port named "Output1". This architecture enables the network
to produce multiple outputs simultaneously, offering greater flexibility in network
output handling: *)
NetGraph[
  (* Define the layers of the net graph: *)
  {2,Ramp,Plus},
  {
    (* Connects output of the first layer to input of the second layer: *)
    1->2,
    (* Connects "Input" port and output of the second layer to the third layer, and
then connects the output of the third layer to "Output2" port: *)
    {NetPort["Input"],2}->3->NetPort["Output2"],
    (* Connects output of the first layer to "Output1" port: *)
    1->NetPort["Output1"]
  }
]
```

Output

Unit 4.4

NetInitialize

`NetInitialize[net]`

gives a net in which all uninitialized learnable parameters in net have been given initial values.

`NetInitialize[net, All]`

gives a net in which all learnable parameters have been given initial values.

- In Mathematica, the `NetInitialize` function initializes the parameters of a NN (net) before it's used for computation. When `NetInitialize[net, All]` is used, it initializes all parameters, including those that are learnable or trainable.
- However, it's essential to note that when you initialize the network using `NetInitialize[net, All]`, any existing training or preset learnable parameters in net will indeed be overwritten by the newly initialized parameters. This means that if the network had been previously trained or had preset learnable parameters defined, they will be discarded and replaced with newly initialized parameters.
- In NN initialization, especially when using `NetInitialize` in Mathematica, weights are typically initialized with random values, and biases are usually initialized to zero.
- This approach is commonly used because it helps introduce some level of randomness into the network's initial weights, which can aid in breaking symmetry and preventing the network from getting stuck in local minima during training. Random initialization also helps in exploring the solution space more effectively.
- On the other hand, biases are often initialized to zero because they represent the additive constants in the linear transformation of the network's inputs. Initializing biases to zero ensures that the network starts with a neutral bias before it learns the optimal bias values during training.
- However, it's worth noting that there are variations and alternatives to these initialization strategies, and different initialization schemes may be more appropriate for different types of networks or tasks. For example, in certain cases, weights may be initialized using specific distributions or scaling techniques to better suit the network architecture or the nature of the data being processed.

The following optional parameters can be included:

<code>Method</code>	<code>"Kaiming"</code>	which initialization method to use
<code>RandomSeeding</code>	<code>1234</code>	seeding of pseudorandom number generator

Possible settings for Method include:

<code>"Kaiming"</code>	choose weights to preserve variance of arrays when propagated through layers, with the method introduced by Kaiming He et al. (2015)
<code>"Xavier"</code>	choose weights to preserve variance of arrays when propagated through layers, with the method introduced by Xavier Glorot et al. (2014)
<code>"Orthogonal"</code>	choose weights to be orthogonal matrices
<code>"Random"</code>	choose weights from a given univariate distribution
<code>"Identity"</code>	choose weights so as to preserve components of arrays when propagated through affine layers

For the method `"Random"`, the following suboptions are supported:

<code>"Weights"</code>	<code>NormalDistribution[0,1]</code>	random distribution to use to initialize weight matrices
<code>"Biases"</code>	<code>None</code>	random distribution to use to initialize bias vectors

For the methods `"Kaiming"` and `"Xavier"`, the following suboption is supported:

<code>"Distribution"</code>	<code>"Normal"</code>	either "Normal" or "Uniform"
-----------------------------	-----------------------	------------------------------

Mathematica Code 4.43

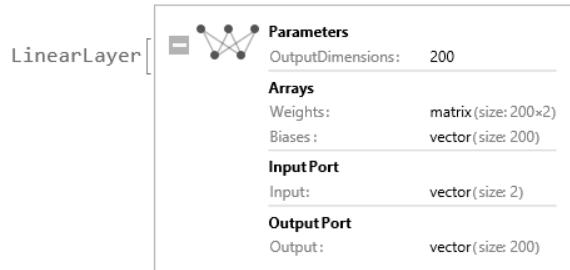
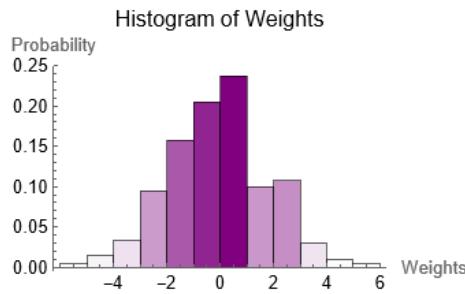
Input

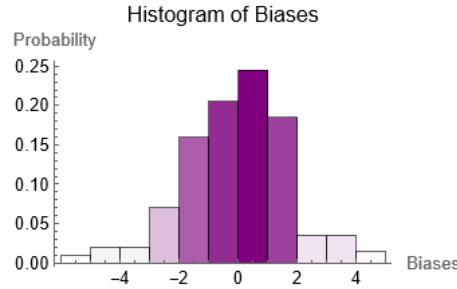
```
(* The code initializes a neural network with a linear layer of 200 neurons and 2 input dimensions using the "Random" initialization method. "Random" initialization is used, employing normal distributions with a standard deviation of 2 for both weights and biases. The code then extracts and visualizes the initialized weights and biases through histograms: *)

(* Specify "Random" initialization, using normal distributions with a standard deviation of 2 for both weights and biases: *)
net=NetInitialize[
  LinearLayer[200,"Input"→2],
  Method→{"Random","Weights"→2,"Biases"→2}
]

(* Extract and plot the initialized weights and biases:*)
Histogram[
  Flatten@NetExtract[net,"Weights"],
  Automatic,
  "Probability",
  PlotLabel→"Histogram of Weights",
  AxesLabel→{"Weights","Probability"},
  ColorFunction→Function[{height},Opacity[height]],
  ChartStyle→Purple,
  ImageSize→250
]

Histogram[
  Flatten@NetExtract[net,"Biases"],
  Automatic,
  "Probability",
  PlotLabel→"Histogram of Biases",
  AxesLabel→{"Biases","Probability"},
  ColorFunction→Function[{height},Opacity[height]],
  ChartStyle→Purple,
  ImageSize→250
]
```

Output**Output**

Output**Mathematica Code 4.44****Input**

```
(* The code illustrates how setting the random seed to Automatic in the NetInitialize
function ensures diverse initializations for repeated calls, while the default
behavior results in consistent initialization due to the fixed random seed. This
is important for exploring different network initializations during experimentation
and training: *)
```

```
(* The following two lines demonstrate the default behavior of NetInitialize, where
the same random seed (1234) is used for repeated calls to initialize a linear layer
with one neuron and one input dimension: *)
```

```
Normal@NetExtract[NetInitialize[LinearLayer[1, "Input" -> 1]], "Weights"]
Normal@NetExtract[NetInitialize[LinearLayer[1, "Input" -> 1]], "Weights"]
```

```
(* The next two lines show the use of RandomSeeding -> Automatic when initializing
the network. This ensures that repeated calls to initialize the network produce
different initializations by using different random seeds: *)
```

```
Normal@NetExtract[NetInitialize[LinearLayer[1, "Input" -> 1], RandomSeeding-
->Automatic], "Weights"]
Normal@NetExtract[NetInitialize[LinearLayer[1, "Input" -> 1], RandomSeeding-
->Automatic], "Weights"]
```

Output

```
{{{ -0.508336}}}
```

Output

```
{{{ -0.508336}}}
```

Output

```
{{{ -1.22817}}}
```

Output

```
{{{ -0.590982}}}
```

Mathematica Code 4.45**Input**

```
(* The code creates a base neural network that takes 2-dimensional vector inputs
and produces 1-dimensional vector outputs. It then generates and visualizes eight
randomly initialized copies of this network using 3D plots. Each 3D plot represents
the output of one initialized network for varying input values. Due to the randomness
in the initialization process, each network has different starting points (weights
and biases). Consequently, the responses of these networks to input vectors exhibit
diversity, visually depicted in the distinct shapes and patterns of the 3D plots.
By observing how the initialized networks respond to inputs, you gain insights into
how initialization influences the learning process. Networks that start with
different initial parameters may converge to different local minima during training,
resulting in varied learned representations: *)
```

```
(* Construct a base network that takes vector inputs of size 2 and produces
vector outputs of size 1:*)
net=NetChain[
```

```
{30,Sin,3,Tanh,3,LogisticSigmoid,1},
 "Input" ->2
];
```

```
(* Make a table of 8 randomly initialized copies of the base network: *)
nets=Table[
```

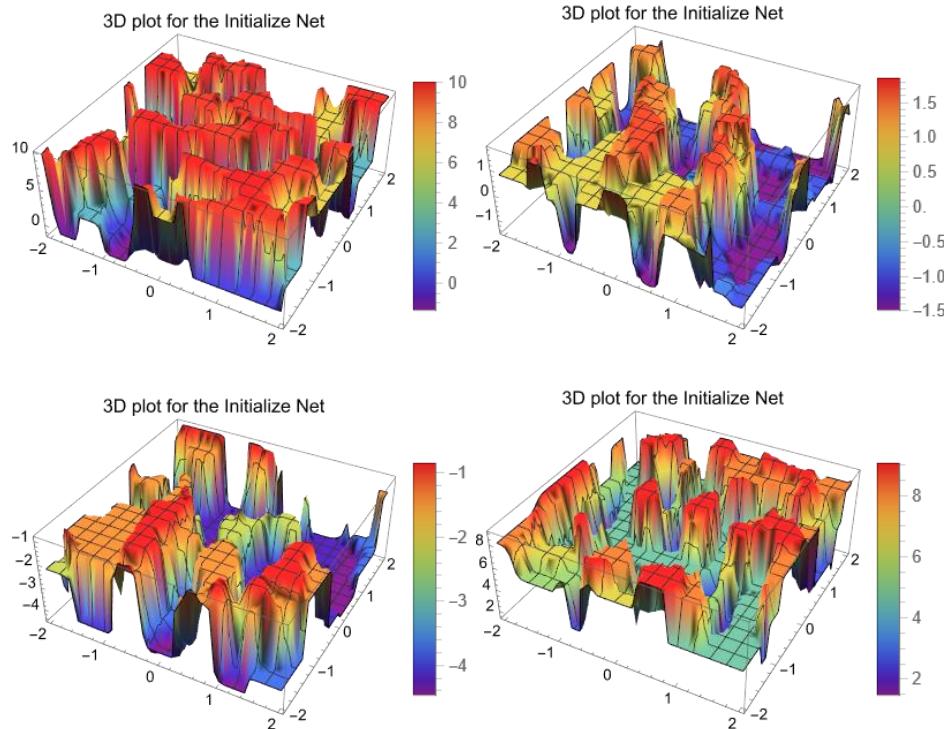
```

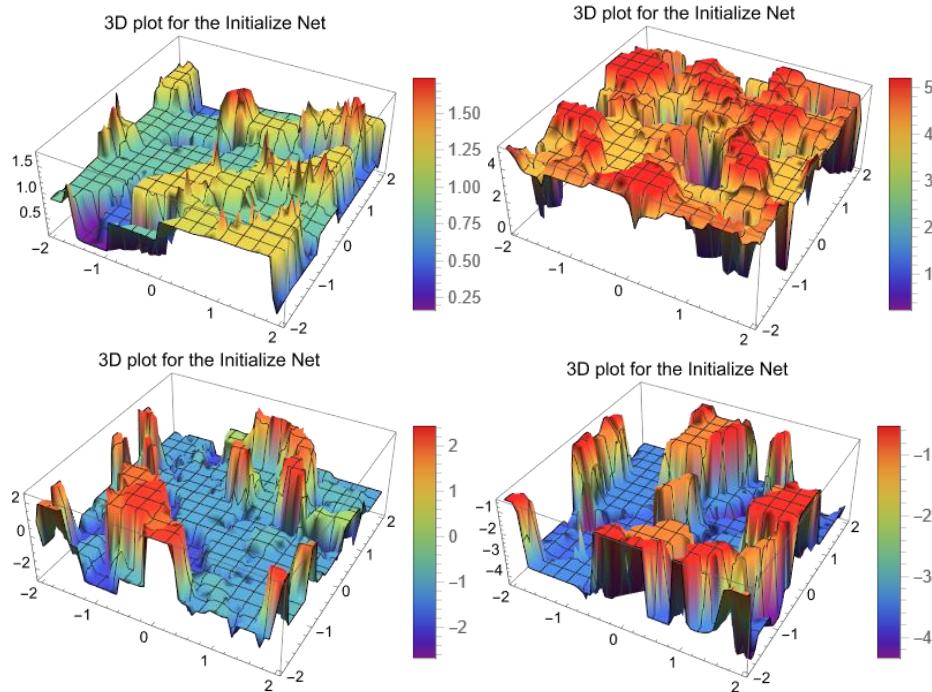
NetInitialize[
  net,
  (* Initialization method with specified weights and biases: *)
  Method -> {"Random", "Weights" -> 3, "Biases" -> 2},
  (* Automatic random seeding for reproducibility: *)
  RandomSeeding -> Automatic
],
8
];

(* 3D Plot 8 randomly initialized copies: *)
Table[
  Plot3D[
    (* Evaluate each initialized network on the 3D grid defined by x and y: *)
    nets[[i]][{x, y}],
    {x, -2, 2},
    {y, -2, 2},
    PlotLabel -> "3D plot for the Initialize Net",
    ColorFunction -> "Rainbow",
    PlotLegends -> Automatic,
    ImageSize -> 250
  ],
  {i, 1, 8}
]

```

Output



**Mathematica Code 4.46**

Input (* The code encapsulates an interactive exploration of a simple neural network architecture. It constructs a network comprising five layers, consisting of linear layers followed by Tanh activation functions, and initializes it with random weights. Through the manipulation interface, users can adjust the random seed and weight initialization to observe their effects on the network's behavior. The code computes and visualizes the distributions of weights within specific layers and the distributions of outputs from selected layers, offering insights into the internal dynamics of the network: *)

```

Manipulate[
Module[
{initializedNet,inputData,layer1,layer2,layer3,layer4,layer5,outputlayer1,output
layer2,outputlayer3,outputlayer4,outputlayer5,simpleNet,histogramWeights,histogr
amOutputs},

(* Define a simple neural network architecture: *)
simpleNet=NetChain[
{
  LinearLayer[10],ElementwiseLayer[Tanh],
  LinearLayer[10],ElementwiseLayer[Tanh],
  LinearLayer[1]
},
"Input"->200
];

(* Initialize the network with random weights: *)
initializedNet=NetInitialize[
  simpleNet,
  RandomSeeding->seed,
  Method->{"Random","Weights"->weights}
];

```

```
(* Generate random input data: *)
inputData=RandomReal[1,{100,200}];

(* Extract individual layers for further analysis: *)

{layer1,layer2,layer3,layer4,layer5}=Table[NetExtract[initializedNet,j],{j,1,5}]
;

(* Compute output of each layer: *)

outputlayer1=layer1[inputData];
outputlayer2=layer2[outputlayer1];
outputlayer3=layer3[outputlayer2];
outputlayer4=layer4[outputlayer3];
outputlayer5=layer5[outputlayer4];

(* Visualize the distributions of weights within specific layers: *)

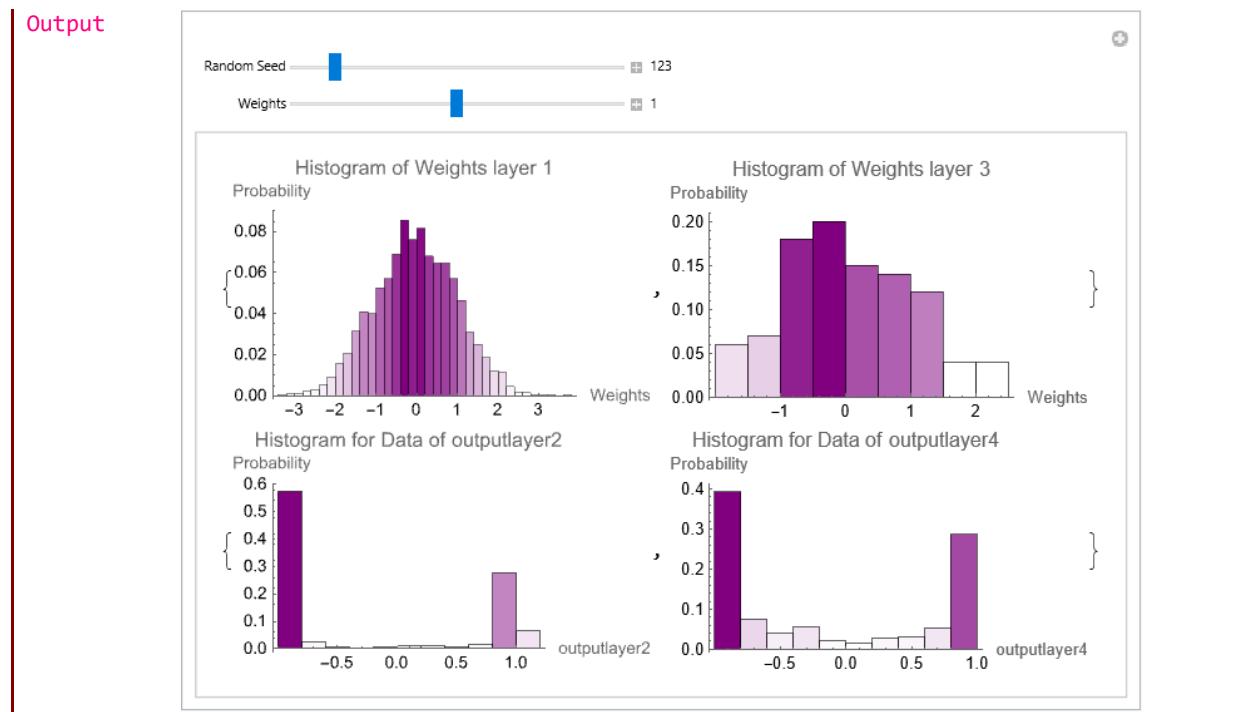
histogramWeights=Table[
  Histogram[
    Flatten@NetExtract[initializedNet,{i,"Weights"}],
    Automatic,
    "Probability",
    PlotLabel->StringForm["Histogram of Weights layer ``",i],
    AxesLabel->{"Weights","Probability"},
    ColorFunction->Function[{height},Opacity[height]],
    ChartStyle->Purple,
    ImageSize->250
  ],
  {i,{1,3}}
];

(* Visualize the distributions of outputs from selected layers: *)

histogramOutputs=Table[
  Histogram[
    Flatten[i[[2]]],
    Automatic,
    "Probability",
    PlotLabel->Style[Row[{"Histogram for Data of ",i[[1]]}],],
    AxesLabel->{i[[1]],"Probability"},
    ColorFunction->Function[{height},Opacity[height]],
    ChartStyle->Purple,
    ImageSize->250
  ],
  {i,{{"outputlayer2",outputlayer2}, {"outputlayer4",outputlayer4}}}
];

(* Display histograms of weights and outputs: *)

Column[{histogramWeights,histogramOutputs}]
],
{{seed,123,"Random Seed"},1,1000,1,Appearance->"Labeled"},
{{weights,1,"Weights"},0,2,0.01,Appearance->"Labeled"}
]
```



Unit 4.5

Cost Functions

`MeanSquaredLossLayer[]`

represents a loss layer that computes the mean squared loss between its "Input" port and "Target" port.

`MeanAbsoluteLossLayer[]`

represents a loss layer that computes the mean absolute loss between the "Input" port and "Target" port.

`CrossEntropyLossLayer["Index"]`

represents a net layer that computes the cross-entropy loss by comparing input class probability vectors with indices representing the target class.

`CrossEntropyLossLayer["Probabilities"]`

represents a net layer that computes the cross-entropy loss by comparing input class probability vectors with target class probability vectors.

`CrossEntropyLossLayer["Binary"]`

represents a net layer that computes the binary cross-entropy loss by comparing input probability scalars with target probability scalars, where each probability represents a binary choice.

`MeanSquaredLossLayer` exposes the following ports for use in NetGraph etc.:

<code>"Input"</code>	an array of arbitrary rank
<code>"Target"</code>	an array of the same rank as "Input"
<code>"Loss"</code>	a real number

- `MeanSquaredLossLayer[...][<|"Input"->in, "Target"->target|>]` explicitly computes the output from applying the layer.
- `MeanSquaredLossLayer[...][<|"Input"->{in1,in2,...}, "Target"->{target1,target2,...}|>]` explicitly computes outputs for each of the ini and targeti.
- When appropriate, `MeanSquaredLossLayer` is automatically used by `NetTrain` if an explicit loss specification is not provided.
- `MeanAbsoluteLossLayer[...][<|"Input"-> in, "Target"->target|>]` explicitly computes the output from applying the layer.
- `MeanAbsoluteLossLayer[...][<|"Input"->{in1,in2,...}, "Target"->{target1,target2,...}|>]` explicitly computes outputs for each of the ini and targeti.
- For `CrossEntropyLossLayer["Binary"]`, the input and target should be scalar values between 0 and 1, or arrays of these.
- For `CrossEntropyLossLayer["Index"]`, the input should be a vector of probabilities $\{p_1, \dots, p_c\}$ that sums to 1, or an array of such vectors. The target should be an integer between 1 and c, or an array of such integers.
- For `CrossEntropyLossLayer["Probabilities"]`, the input and target should be a vector of probabilities that sums to 1, or an array of such vectors.
- `CrossEntropyLossLayer[...][<|"Input"->in, "Target"->target|>]` explicitly computes the output from applying the layer.
- `CrossEntropyLossLayer[...][<|"Input"->{in1,in2,...}, "Target"->{target1,target2,...}|>]` explicitly computes outputs for each of the ini and targeti.
- Possible forms for shape include:

"Real"	a single real number
"Integer"	a single integer
Restricted["Integer",n]	an integer between 1 and n
Restricted["Integer",{m,n}]	an integer between m and n
n	a vector of length n
{n1,n2,...}	an array of dimensions $n_1 \times n_2 \times \dots$
"Varying"	a vector whose length is variable.
{"Varying",n2,n3,...}	an array whose first dimension is variable and remaining dimensions are $n_2 \times n_3 \times \dots$

Mathematica Code 4.47

Input

```
(* The code initializes example input and target data, creates Mean Squared Loss layer and Mean Absolute Loss Layer, and then applies the layers to calculate the mean squared loss and Mean Absolute Loss Layer on the provided data: *)
(* Define some example data: *)
inputData = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
targetData = {{2, 4, 6}, {8, 10, 12}, {14, 16, 18}};

(* Create a MeanSquaredLossLayer and MeanAbsoluteLossLayer: *)
meanSquaredLossLayer = MeanSquaredLossLayer[];
meanAbsoluteLossLayer = MeanAbsoluteLossLayer[];

(* Apply the MeanSquaredLossLayer and MeanAbsoluteLossLayer to compute the mean squared loss and Mean absolute loss: *)
meanSquaredLoss = meanSquaredLossLayer[<|"Input" -> inputData, "Target" -> targetData|>]
meanAbsoluteLoss = meanAbsoluteLossLayer[<|"Input" -> inputData, "Target" -> targetData|>]

Output 31.6667
Output 5.
```

Mathematica Code 4.48

Input

```
(* When using MeanSquaredLossLayer (or MeanAbsoluteLossLayer), you can specify the shape of the input port using the "port"->shape syntax. The shape parameter allows you to define the expected dimensions or structure of the input. The shape can take different forms to capture various aspects of the input structure. When you use "Input"->{n}, you are treating each element of the input vectors separately, calculating the mean squared loss (or mean absolute loss) individually for each element. On the other hand, without specifying the input size, the layer calculates the mean squared loss (or mean absolute loss) globally, considering the entire input tensor. The choice between these two options depends on the specific requirements of your model and how you want to interpret the loss: *)
(* Define example data with vectors of length 5: *)
inputData = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
targetData = {{2, 4, 6, 8, 10}, {12, 14, 16, 18, 20}, {22, 24, 26, 28, 30}};
(* Ensure that the sizes of the input and target vectors are the same (each vector has three elements), which is a typical requirement for training a neural network. *)

(* Create a MeanSquaredLossLayer and MeanAbsoluteLossLayer that takes length-5 vectors: *)
loss1 = MeanSquaredLossLayer["Input" -> {5}];
loss2 = MeanAbsoluteLossLayer["Input" -> {5}];
```

```
(* Create a MeanSquaredLossLayer and MeanAbsoluteLossLayer without the input port:
*)
loss3=MeanSquaredLossLayer[];
loss4=MeanAbsoluteLossLayer[];

(* Apply the layers to data: *)
loss1[<|"Input"->inputData,"Target"->targetData|>]
loss2[<|"Input"->inputData,"Target"->targetData|>]

loss3[<|"Input"->inputData,"Target"->targetData|>]
loss4[<|"Input"->inputData,"Target"->targetData|>]

Output {11.,66.,171.}
Output {3.,8.,13.}
Output 82.6667
Output 8.
```

Mathematica Code 4.49

Input (* The MeanSquaredLossLayer (or MeanAbsoluteLossLayer) without specifying the input size calculates the mean squared loss (or mean absolute loss) globally, considering the entire input and output tensors. The MeanSquaredLossLayer (or MeanAbsoluteLossLayer) without specifying the "Input" port is equivalent to the manual computation using the Mean and Flatten functions. The use of Flatten is important to handle multidimensional inputs: *)

```
(* Example data with vectors of length 5: *)
inputData={{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
targetData={{2,4,6,8,10},{12,14,16,18,20},{22,24,26,28,30}};

(* Manual definition for loss calculation: *)
manualMeanSquaredLoss[input_,target_]:=N[Mean[Flatten[(input-target)^2]]]
manualMeanAbsoluteLoss[input_,target_]:=N[Mean[Flatten[Abs[input-target]]]]

(* Compare the output of the MeanSquaredLossLayer (and MeanAbsoluteLossLayer) without "Input" port and the manual definition on an example: *)

(* Data container with "Input" and "Target" keys: *)
data=<|"Input"->inputData,"Target"->targetData|>

(* Apply MeanSquaredLossLayer (and MeanAbsoluteLossLayer) without "Input" port: *)
resultFromLayerS=MeanSquaredLossLayer[][[data]]
resultFromLayerA=MeanAbsoluteLossLayer[][[data]]

(* Apply the manual definition to the same data: *)
resultManualS=manualMeanSquaredLoss[inputData,targetData]
resultManualA=manualMeanAbsoluteLoss[inputData,targetData]

Output 82.6667
Output 8.
Output 82.6667
Output 8.
```

Mathematica Code 4.50

Input (* MeanSquaredLossLayer effectively computes a normalized version of SquaredEuclideanDistance. However, MeanAbsoluteLossLayer effectively computes a normalized version of ManhattanDistance: *)

```
(* Example data with vectors of length 5: *)
```

```

inputdata=RandomReal[5,20];
outputdata=RandomReal[5,20];

(* Compute Squared Euclidean Distance and normalize: *)
squaredEuclideanDistance=SquaredEuclideanDistance[inputdata,outputdata]/Length[inputdata]

(* Use MeanSquaredLossLayer to achieve the same result in a neural network context: *)
MeanSquaredLossLayer[][{<|"Input" -> inputdata, "Target" -> outputdata|>}]

(* Compute Manhattan Distance and normalize: *)
manhattanDistance=ManhattanDistance[inputdata,outputdata]/Length[inputdata]

(* Use MeanAbsoluteLossLayer to achieve the same result in a neural network context: *)
MeanAbsoluteLossLayer[][{<|"Input" -> inputdata, "Target" -> outputdata|>}]

Output 6.18515
Output 6.18515
Output 2.15588
Output 2.15588

```

Mathematica Code 4.51

Input

```

(* The code demonstrates two alternative manual definitions for calculating the
mean squared loss element-wise for corresponding input and target vectors. It then
compares the output of the MeanSquaredLossLayer with the "Input" port and the manual
definition on a specific example, showcasing the equivalence of the results: *)

(* Example data with vectors of length 5: *)
inputData={{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
targetData={{2,4,6,8,10},{12,14,16,18,20},{22,24,26,28,30}};

(* The code provides two alternative definitions for a manual mean squared loss
calculation. The first definition uses a Table construct to calculate the element-
wise mean squared loss for each pair of input and target vectors. The second
definition employs Transpose to achieve the same result in a concise manner: *)

(* Manual definition for a mean squared loss calculation: *)
manualMeanSquaredLoss[input_,target_]:=Table[N[Mean[(input[[i]]-
target[[i]])^2]],{i,1,Length[inputData]}]
(* or *)
(* manualMeanSquaredLoss[input_,target_]:=N[Mean[Transpose[(input-target)^2]]] *)

(* Compare the output of the MeanSquaredLossLayer with "Input" port and the manual
definition on an example: *)
(* Data container with "Input" and "Target" keys: *)
data=<|"Input" -> inputData, "Target" -> targetData|>

(* Apply the layer to data: *)
resultFromLayer=MeanSquaredLossLayer["Input" -> {5}][data]

(* Apply the manual definition to the same data: *)
resultManual=manualMeanSquaredLoss[inputData,targetData]

Output {11.,66.,171.}
Output {11.,66.,171.}

```

Mathematica Code 4.52

```

Input      (* The code demonstrates two alternative manual definitions for calculating the
           mean absolute loss element-wise for corresponding input and target vectors. It then
           compares the output of the MeanAbsoluteLossLayer with the "Input" port and the
           manual definition on a specific example, showcasing the equivalence of the results:
           *)

(* Example data with vectors of length 5: *)
inputData={{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
targetData={{2,4,6,8,10},{12,14,16,18,20},{22,24,26,28,30}};

(* Manual definition for a mean squared loss calculation: *)
manualMeanAbsoluteLoss[input_,target_]:=Table[
  N[
    Mean[Abs[input[[i]]-target[[i]]]
    ],
    {i,1,Length[inputData]}
  ]
(* or *)
(*   manualMeanAbsoluteLoss[input_,target_]:=N[Mean[Transpose[Abs[input-target]]]]
*)

(* Compare the output of the MeanAbsoluteLossLayer with "Input" port and the manual
definition on an example: *)
(* Data container with "Input" and "Target" keys: *)
data=<|"Input"→inputData,"Target"→targetData|>

(* Apply the layer to data: *)
resultFromLayer=MeanAbsoluteLossLayer["Input"→{5}][data]

(* Apply the manual definition to the same data: *)
resultManual=manualMeanAbsoluteLoss[inputData,targetData]
Output    {3.,8.,13.}
Output    {3.,8.,13.}

```

Mathematica Code 4.53

```

Input      (* Create MeanSquaredLossLayer and MeanAbsoluteLossLayer: *)
lossS=MeanSquaredLossLayer[];
lossA=MeanAbsoluteLossLayer[];

(* Apply the Layers to a pair of matrices: *)
lossS[<|"Input"→{{1,2,3,5},{4,5,6,4}),"Target"→{{1,1,0,1},{1,0,1,1}}|>]
lossA[<|"Input"→{{1,2,3,5},{4,5,6,4}),"Target"→{{1,1,0,1},{1,0,1,1}}|>]

(* Apply the Layers to a pair of vectors: *)
lossS[<|"Input"→{1,2,3,5),"Target"→{2,1,1,0}|>]
lossA[<|"Input"→{1,2,3,5),"Target"→{2,1,1,0}|>]

(* Apply the Layers to a pair of numbers: *)
lossS[<|"Input"→5,"Target"→1|>]
lossA[<|"Input"→5,"Target"→1|>]
Output    11.75
Output    3.
Output    7.75
Output    2.25
Output    16.
Output    4.

```

Mathematica Code 4.54

```

Input   (* Create a MeanSquaredLossLayer and MeanAbsoluteLossLayer that take two variable-
        length vectors: *)
lossS=MeanSquaredLossLayer["Input"->"Varying"];
lossA=MeanAbsoluteLossLayer["Input"->"Varying"];

(* Apply the layers to an input and target vector: *)
lossS[<|"Input"->{1,3,5}, "Target"->{2,-2,4}|>
lossA[<|"Input"->{1,3,5}, "Target"->{2,-2,4}|>

(* Thread the layers over a batch of input and target vectors: *)
lossS[<|"Input"->{{1,3,5},{2,2},{1,2,3,4}}, "Target"->{{2,3,4},{2,4},{5,7,8,1}}|>
lossA[<|"Input"->{{1,3,5},{2,2},{1,2,3,4}}, "Target"->{{2,3,4},{2,4},{5,7,8,1}}|>

Output  9.
Output  2.33333
Output  {0.666667,2.,18.75}
Output  {0.666667,1.,4.25}

```

Mathematica Code 4.55

```

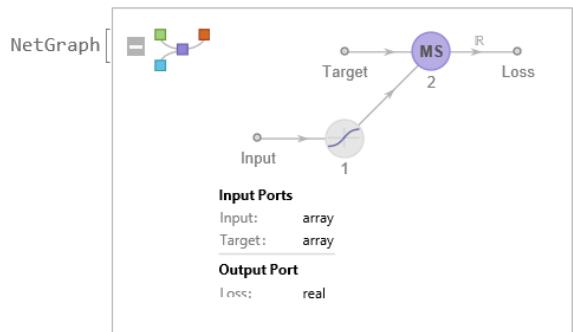
Input   (* The code creates two neural networks with a common structure: an ElementwiseLayer
        with Tanh activation followed by a MeanSquaredLossLayer. The difference is that the
        second network (net2) is configured to handle variable-length inputs using the
        option "Input"->"Varying" in MeanSquaredLossLayer. The networks are then applied
        to input data, producing results (resultNet1 and resultNet2): *)

(* Define the first network (net1) with a MeanSquaredLossLayer: *)
net1=NetGraph[
  {ElementwiseLayer[Tanh],MeanSquaredLossLayer[]},
  {1->NetPort[2,"Input"]}
]

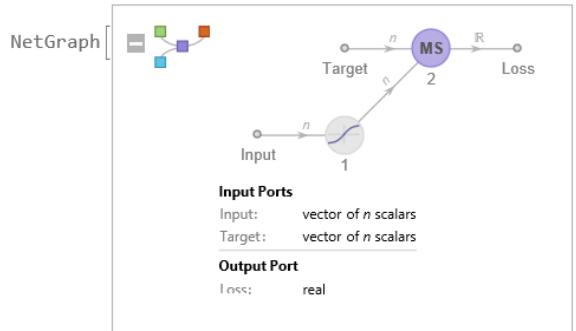
(* Define the second network (net2) with a MeanSquaredLossLayer that supports
variable-length inputs: *)
net2=NetGraph[
  {ElementwiseLayer[Tanh],MeanSquaredLossLayer["Input"->"Varying"]},
  {1->NetPort[2,"Input"]}
]

(* Applying the networks to input data containing "Input" and "Target" keys: *)
resultNet1=net1[<|"Input"->{{1,2,3},{1,3,5},{5,2,2}}, "Target"->{{2,1,3},{1,5,6},{3,4,2}}|>]
resultNet2=net2[<|"Input"->{{1,2,3},{1,3,5},{5,2,2}}, "Target"->{{2,1,3},{1,5,6},{3,4,2}}|>]

```

Output

Output



Output 6.77142

Output {1.85158, 13.6991, 4.76358}

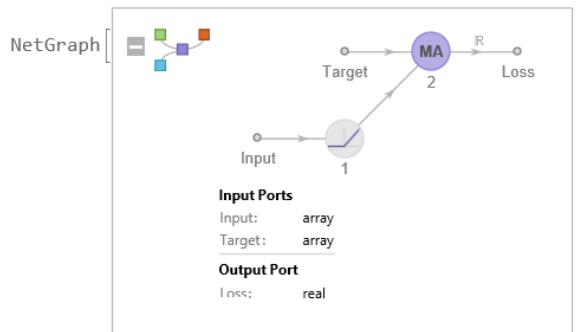
Mathematica Code 4.56

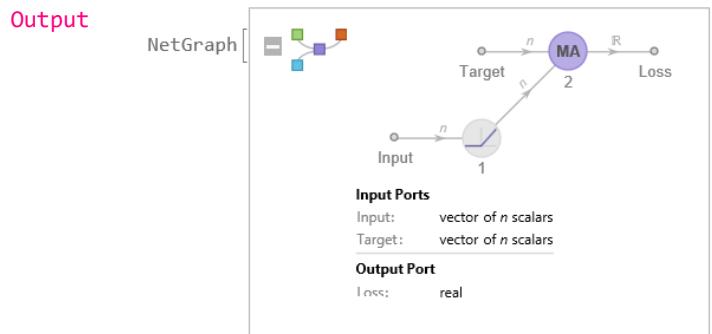
```
Input (* The code creates two neural networks with a common structure: an ElementwiseLayer
       with Relu activation followed by a MeanAbsoluteLossLayer. The difference is that
       the second network (net2) is configured to handle variable-length inputs using the
       option "Input"->"Varying" in MeanAbsoluteLossLayer. The networks are then applied
       to input data, producing results (resultNet1 and resultNet2): *)
(* Define the first network (net1) with a MeanAbsoluteLossLayer: *)
net1=NetGraph[
  {ElementwiseLayer[Ramp],MeanAbsoluteLossLayer[]},
  {1->NetPort[2,"Input"]}
]

(* Define the second network (net2) with a MeanAbsoluteLossLayer that supports
   variable-length inputs: *)
net2=NetGraph[
  {ElementwiseLayer[Ramp],MeanAbsoluteLossLayer["Input"->"Varying"]},
  {1->NetPort[2,"Input"]}
]

(* Applying the networks to input data containing "Input" and "Target" keys: *)
resultNet1=net1[<|"Input"->{{1,2,3},{1,3,5},{5,2,2}}, "Target"->{{2,1,3},{1,5,6},{3,4,2}}|>]
resultNet2=net2[<|"Input"->{{-1,-2,-3},{-1,-3,-5},{-5,-2,-2}}, "Target"->{{2,1,3},{1,5,6},{3,4,2}}|>]
```

Output





Output 1.
Output {2., 4., 3.}

Mathematica Code 4.57

Input (* The code serves to define, evaluate, and visualize the Binary Cross-Entropy Loss function. It begins by defining the binaryCrossEntropyLoss function to compute the loss based on input and target values, followed by an evaluation of this function on specific data to verify its correctness. Equivalence with a built-in function is then confirmed to ensure consistency. The code proceeds to plot the behavior of the loss as the input approaches 1 while the target remains fixed at 1, providing insight into its characteristics under specific conditions. Lastly, a contour plot is generated to illustrate the variation of the loss concerning different input-target combinations, highlighting that the loss is minimized when the target approaches the input: *)

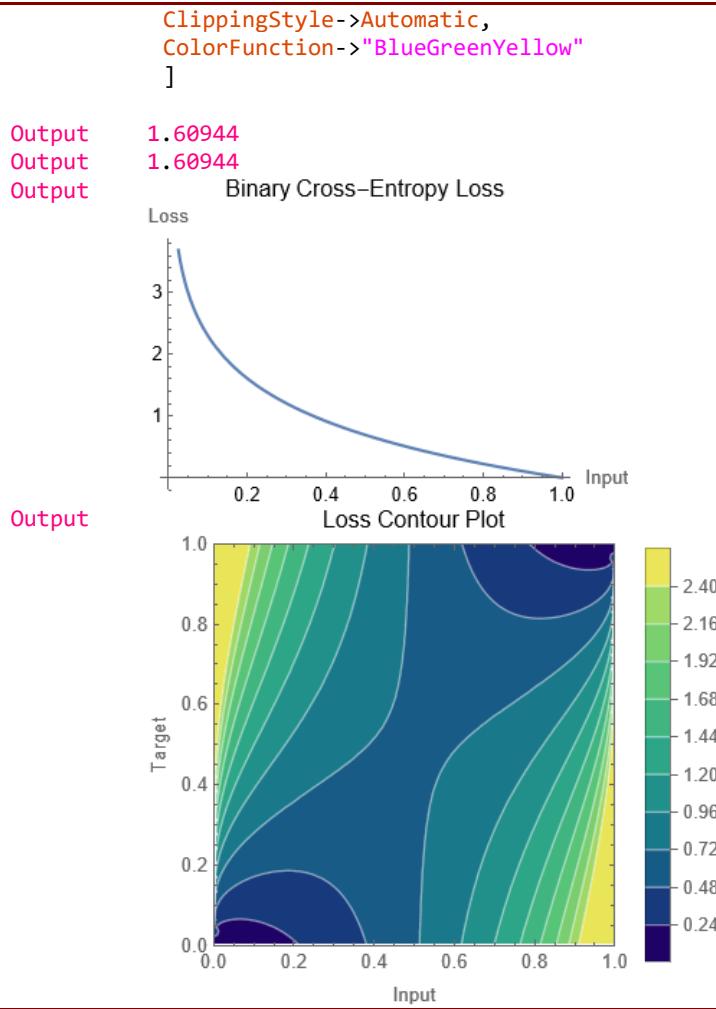
```
(* Define the Binary Cross-Entropy Loss function: *)
binaryCrossEntropyLoss[inputValue_,targetValue_]:=-
(targetValue*Log[inputValue]+(1-targetValue)*Log[1-inputValue]);

(* Evaluate the loss function on specific data: *)
lossValue=binaryCrossEntropyLoss[0.2,1]

(* Verify equivalence with built-in function: *)
builtInLoss=CrossEntropyLossLayer["Binary"][[<|"Input" -> 0.2,"Target" -> 1|>]

(* Plot the loss as the input approaches 1 when the target is fixed at 1: *)
Plot[
  binaryCrossEntropyLoss[input,1],
  {input,0,1},
  ImageSize->250,
  PlotLabel->"Binary Cross-Entropy Loss",
  AxesLabel->{"Input", "Loss"}
]

(* Create a contour plot to visualize loss variation with input and target values.
In general, the loss is minimized when the target approaches the input: *)
ContourPlot[
  binaryCrossEntropyLoss[inputValue,targetValue],
  {inputValue,0,1},
  {targetValue,0,1},
  Contours->10,
  PlotLegends->Automatic,
  PlotRangePadding->0,
  FrameLabel->{"Input", "Target"},
  ImageSize->250,
  PlotLabel->"Loss Contour Plot",
  ContourStyle->{White},
```

**Mathematica Code 4.58**

```

Input (* Create a CrossEntropyLossLayer where the input and target are single
probabilities: *)
loss=CrossEntropyLossLayer["Binary","Input"->"Real"]

(* Apply it to an input and a target:*)
loss[<|"Input"->0.2,"Target"->0.8|>

(* Thread the layer over a batch of inputs:*)
loss[<|"Input"->{0.2,0.1,0.1}, "Target"->{0.8,0.9,0.1}|>]

Output

```

```

CrossEntropyLossLayer[ ]
Output 1.33218
Output {1.33218, 2.08286, 0.325083}

```

Unit 4.6

NetTrain

While we may not have yet completed the exhaustive theoretical groundwork necessary for training NNs, such as exploring various optimization methods and regularization techniques, it is imperative to take our first steps in practical application. In the realm of machine learning, theory and practice are intertwined, each informing and enriching the other.

Building a NN using off-the-shelf software can be likened to a child constructing a toy from building blocks. In both cases, you have distinct elements (blocks in the toy analogy, components or modules in the NN) that serve specific purposes. These components might include layers (such as linear layers, activation layers, etc.), optimizers, loss functions, and more in the context of NNs. Like assembling blocks, the process involves compatibility, a structured assembly process, trial and error, iteration, and opportunities for learning and creativity. Just as a child learns spatial reasoning and problem-solving skills through building with blocks, developers learn about NN architectures and optimization techniques through experimentation and refinement, aiming to achieve optimal performance with their models.

Let us begin training our networks using `NetTrain` in Mathematica. This process will be executed step by step. Initially, in this section, we provide a basic introduction, based on the theoretical foundations established in previous chapters to ensure a solid understanding. Subsequently, we will delve into more intricate details in the forthcoming chapters.

In this section, we will explore the diverse functionalities of the `NetTrain` function, understanding its syntax, parameters, and various forms.

`NetTrain[net,{input1->output1,input2->output2,...}]`

trains the specified neural net by giving the inputi as input and minimizing the discrepancy between the outputi and the actual output of the net, using an automatically chosen loss function.

`NetTrain[net,<|port1->{data11,data12,...},port2->{...},...|>]`

trains the specified net by supplying training data at the specified ports.

`NetTrain[net,"dataset"]`

trains on a named dataset from the Wolfram Data Repository.

`NetTrain[net,f]`

calls the function f during training to produce batches of training data.

`NetTrain[net,data,"prop"]`

gives data associated with a specific property prop of the training session.

`NetTrain[net,data,All]`

gives a `NetTrainResultsObject[...]` that summarizes information about the training session.

The following options are supported:

<code>BatchSize</code>	<code>Automatic</code>	how many examples to process in a batch
<code>LearningRate</code>	<code>Automatic</code>	rate at which to adjust weights to minimize loss
<code>LearningRateMultipliers</code>	<code>Automatic</code>	set relative learning rates within the net
<code>LossFunction</code>	<code>Automatic</code>	the loss function for assessing outputs
<code>MaxTrainingRounds</code>	<code>Automatic</code>	how many times to traverse the training data
<code>Method</code>	<code>Automatic</code>	the training method to use
<code>PerformanceGoal</code>	<code>Automatic</code>	favor settings with specific advantages
<code>TargetDevice</code>	<code>"CPU"</code>	the target device on which to perform training
<code>TimeGoal</code>	<code>Automatic</code>	number of seconds to train for

<code>TrainingProgressMeasurements</code>	<code>Automatic</code>	measurements to monitor, track and plot during training
<code>TrainingProgressCheckpointing</code>	<code>None</code>	how to periodically save partially trained nets
<code>RandomSeeding</code>	<code>1234</code>	how to seed pseudorandom generators internally
<code>TrainingProgressFunction</code>	<code>None</code>	function to call periodically during training
<code>TrainingProgressReporting</code>	<code>Automatic</code>	how to report progress during training
<code>TrainingStoppingCriterion</code>	<code>None</code>	how to automatically stop training
<code>TrainingUpdateSchedule</code>	<code>Automatic</code>	when to update specific parts of the net
<code>ValidationSet</code>	<code>None</code>	the set of data on which to evaluate the model during training
<code>WorkingPrecision</code>	<code>Automatic</code>	precision of floating-point calculations

Remarks:

- Any input ports of the net whose shapes are not fixed will be inferred from the form of training data.
- Individual training data inputs can be scalars, vectors, numeric tensors.
- If the loss is not given explicitly using `LossFunction`, a loss function will be chosen automatically based on the final layer or layers in the net.
- With the default setting of `BatchSize->Automatic`, the batch size will be chosen automatically, based on the memory requirements of the network and the memory available on the target device. The maximum batch size that will be automatically chosen is 64.
- With the default setting of `MaxTrainingRounds->Automatic`, training will occur for approximately 20 seconds, but never for more than 10,000 rounds.
- With the setting of `MaxTrainingRounds->n`, training will occur for n rounds, where a round is defined to be a traversal of the entire training dataset.
- `NetTrain[net,data,All]` returns a `NetTrainResultsObject[...]` that contains values for all properties that do not require significant additional computation or memory.
- `NetTrainResultsObject[...][prop]` is used to look up property prop from the `NetTrainResultsObject`.
- Properties supported include:
`"ArraysLearningRateMultipliers"`, `"BatchesPerRound"`, `"BatchLossList"`,
`"BatchMeasurementsLists"`, `"BatchSize"`, `"BestValidationRound"`, `"CheckpointingFiles"`,
`"ExamplesProcessed"`, `"FinalLearningRate"`, `"FinalPlots"`, `"InitialLearningRate"`, `"LossPlot"`,
`"MeanBatchesPerSecond"`, `"MeanExamplesPerSecond"`, `"NetTrainInputForm"`,
`"OptimizationMethod"`, `"ReasonTrainingStopped"`, `"RoundLoss"`, `"RoundLossList"`,
`"RoundMeasurements"`, `"RoundMeasurementsLists"`, `"RoundPositions"`, `"TargetDevice"`,
`"TotalBatches"`, `"TotalRounds"`, `"TotalTrainingTime"`, `"TrainedNet"`, `"TrainingExamples"`,
`"TrainingNet"`, `"ValidationExamples"`, `"ValidationLoss"`, `"ValidationLossList"`,
`"ValidationMeasurements"`, `"ValidationMeasurementsLists"`, and `"ValidationPositions"`
- If a net already contains initialized or previously trained weights, these will not be reinitialized by `NetTrain` before training is performed.

Possible settings for Method include:

<code>"ADAM"</code>	stochastic gradient descent using an adaptive learning rate that is invariant to diagonal rescaling of the gradients
<code>"RMSProp"</code>	stochastic gradient descent using an adaptive learning rate derived from exponentially smoothed average of gradient magnitude
<code>"SGD"</code>	ordinary stochastic gradient descent with momentum
<code>"SignSGD"</code>	stochastic gradient descent for which the magnitude of the gradient is discarded

Suboptions for specific methods can be specified using `Method->{"method",opt1->val1,...}`. The following suboptions are supported for all methods:

<code>"LearningRateSchedule"</code>	<code>Automatic</code>	how to scale the learning rate as training progresses
<code>"L2Regularization"</code>	<code>None</code>	the global loss associated with the L2 norm of all learned arrays
<code>"GradientClipping"</code>	<code>None</code>	the magnitude above which gradients should be clipped
<code>"WeightClipping"</code>	<code>None</code>	the magnitude above which weights should be clipped

Mathematica Code 4.59

```

Input      (* The code demonstrates the basics of training a linear neural network for a simple
           regression task and visualizing the learned function: *)

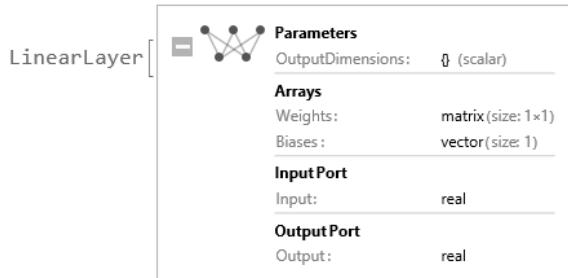
(* Train a single-layer linear net on input->output pairs: *)
(* Create a linear neural network with one layer: *)
trainednet=NetTrain[
  LinearLayer[],
  (* Input-output pairs for training: *)
  {1->2.9,2->6.1,3->9.0,4->12.1},
  (* Set the batch size for training: *)
  BatchSize->4,
  (* Maximum number of training iterations: *)
  MaxTrainingRounds->500,
  (* Learning rate for stochastic gradient descent: *)
  LearningRate->0.01,
  (* Specify mean squared loss as the training objective: *)
  LossFunction->MeanSquaredLossLayer[],
  (* Use stochastic gradient descent as the optimization method: *)
  Method->"SGD"
]

(* Use the trained network to predict the output for a single input value (2.1): *)
trainednet[2.1]

(* Use the trained network to predict outputs for multiple input values {1,2,3,4}: *)
trainednet[{1,2,3,4}]

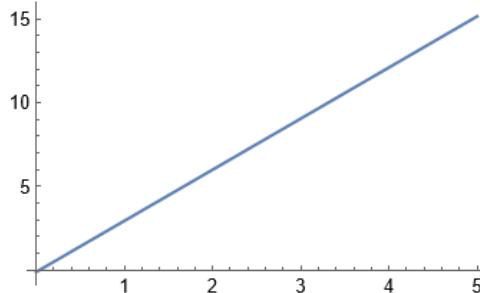
(* Visualize the learned linear function over the input range[0,5]: *)
Plot[
  trainednet[x],
  {x,0,5},
  ImageSize->250
]

```

Output

Output 6.305

Output {2.95, 6., 9.05, 12.1}



Mathematica Code 4.60

Input

```
(* This code creates and trains a neural network (NetChain) with a specific architecture {1,3,3,1}. It then uses input-output pairs for training, specified in a similar manner as the previous example: *)
(* Define and train a neural network with a chain of layers {1,3,3,1}: *)
trainedNetwork=NetTrain[
  (* The architecture of the neural network: *)
  NetChain[{1,3,3,1}],
  (* Training data as input->output pairs: *)
  {1->2.9,2->6.1,3->9.0,4->12.1},
  (* Set the batch size for training: *)
  BatchSize->4,
  (* Maximum number of training iterations: *)
  MaxTrainingRounds->500,
  (* Learning rate for stochastic gradient descent: *)
  LearningRate->0.001,
  (* Mean squared loss used as the training objective: *)
  LossFunction->MeanSquaredLossLayer[],
  (* Stochastic Gradient Descent is the optimization method: *)
  Method->"SGD"
]

(* Use the trained network to predict outputs for multiple input values {2,3,4,5}: *)
multiplePredictions=trainedNetwork[{2,3,4,5}]
```

Output

 <code>NetChain[</code>	Input real 1 LinearLayer vector (size: 1) 2 LinearLayer vector (size: 3) 3 LinearLayer vector (size: 3) 4 LinearLayer vector (size: 1) Output scalar
----------------------------	---

Output

```
{5.99999, 9.04999, 12.1, 15.15}
```

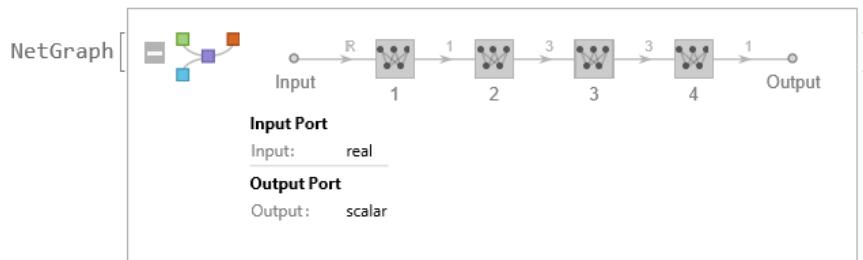
Mathematica Code 4.61

Input

```
(* Neural Network Training with Directed Graph: *)
(* This code creates and trains a neural network (NetGraph) with a directed graph of layers {1,3,3,1}. The connections between layers are specified by the graph {1->2->3->4}. Training data, as input-output pairs, is provided in a similar manner as the previous examples: *)
(* Define and train a neural network with a directed graph of layers {1,3,3,1}: *)
directedGraphNet=NetTrain[
  NetGraph[{1,3,3,1}, {1->2->3->4}],
  {1->2.9,2->6.1,3->9.0,4->12.1},
  BatchSize->4,
  MaxTrainingRounds->500,
  LearningRate->0.001,
  LossFunction->MeanSquaredLossLayer[],
  Method->"SGD"
]

(* Make several predictions at once: *)
multiplePredictions=directedGraphNet[{2,3,4,5}]
```

Output



Output {5.99999, 9.04999, 12.1, 15.15}

Mathematica Code 4.62

```
Input (* Neural Network Training and Data Representation: *)

(* Note that: If you don't explicitly specify hyperparameters such as learning
rate, batch size, etc., Mathematica will use default values for these parameters
during the training process. Defaults are pre-set values chosen by the developers
of the neural network functions in Mathematica. These defaults are often chosen to
work well across a broad range of problems, but they may not be optimal for every
specific case. If you have a specific problem or dataset, it's often a good practice
to experiment with different hyperparameter values to find the combination that
works best for your particular scenario: *)

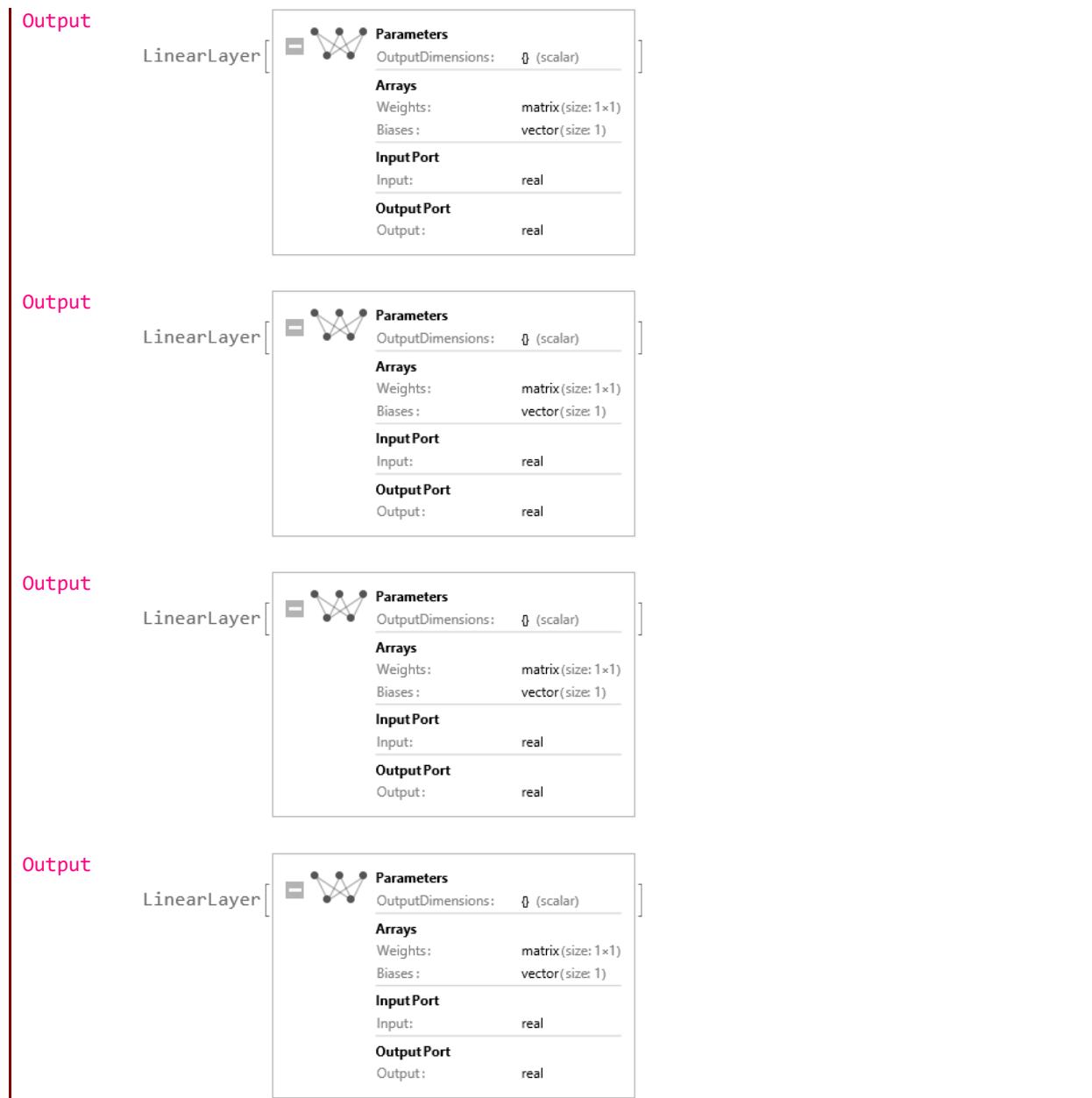
(* The following comments explain how the training data is represented in different
formats, including rules, lists, and associations: *)

(* Train a neural network using input-output pairs represented as rules: *)
NetTrain[
  LinearLayer[],
  {1->2.9, 2->6.1, 3->9.0, 4->12.1}
]

(* Alternative representation using lists for both inputs and outputs: *)
NetTrain[
  LinearLayer[],
  {1,2,3,4}->{2.9,6.1,9.,12.1}
]

(* Train using an association, explicitly naming input and output ports: *)
NetTrain[
  LinearLayer[],
  <|"Input" -> {1,2,3,4}, "Output" -> {2.9,6.1,9.,12.1}|>
]

(* Train with a list of associations, each containing input and output
information: *)
NetTrain[LinearLayer[], {
  <|"Input" -> 1, "Output" -> 2.9|>,
  <|"Input" -> 2, "Output" -> 6.1|>,
  <|"Input" -> 3, "Output" -> 9.|>,
  <|"Input" -> 4, "Output" -> 12.1|>
}]
```

**Mathematica Code 4.63**

```

Input (* Regression with Noisy Sine Wave Dataset: *)
(* This example creates a simple feedforward neural network, trains it on a noisy
sine wave dataset, and then tests it on a new input. The network architecture
consists of three hidden layers with 25, 5, and 1 neurons, respectively. Tanh
activation functions are applied after the linear layers to introduce non-linearity:
*)

(* Set a random seed for reproducibility: *)
SeedRandom[123];

(* Generate a noisy sine wave dataset: *)
inputs=Table[x,{x,0,4 Pi,0.01}];
target=Table[
  Sin[x]+RandomVariate[NormalDistribution[0,0.1]],

```

```

{x,0,4 Pi,0.01}
];

data=Transpose[{inputs,target}];

(* Define a neural network for regression: *)
regressionNetwork=NetChain[
{
  LinearLayer[25],ElementwiseLayer[Tanh],
  LinearLayer[5],ElementwiseLayer[Tanh],
  LinearLayer[1]
}

(* Train the neural network: *)
trainedNetwork=NetTrain[
  regressionNetwork,
  <|"Input"->inputs,"Output"->target|>,
  MaxTrainingRounds->1000
]

(* Visualize the trained network architecture: *)
Information[trainedNetwork,"SummaryGraphic"]

(* Visualize the regression results: *)
Show[
  ListPlot[
    data,
    PlotStyle->Blue,
    ImageSize->250,
    PlotLabel->"Noisy Sine Wave Dataset and
    Trained Network Regression"
  ],
  Plot[
    trainedNetwork[x],
    {x,0,4 Pi},
    PlotStyle->Purple,
    ImageSize->250
  ]
]

```

Output

```

NetChain [
  Input      array
  1 LinearLayer vector(size: 25)
  2 Tanh      vector(size: 25)
  3 LinearLayer vector(size: 5)
  4 Tanh      vector(size: 5)
  5 LinearLayer vector(size: 1)
  Output     vector(size: 1)
]

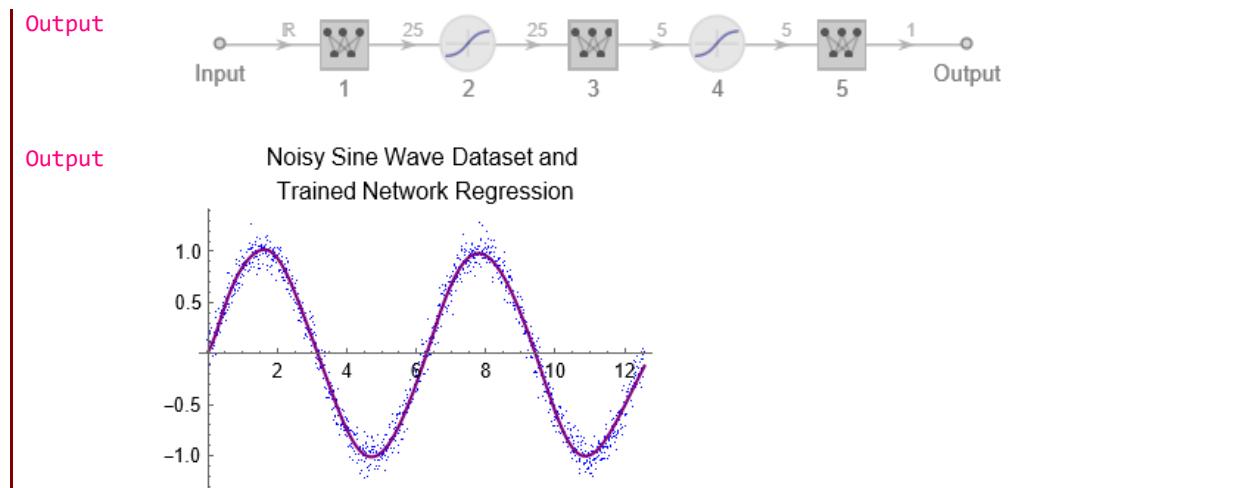
```

Output

```

NetChain [
  Input      real
  1 LinearLayer vector(size: 25)
  2 Tanh      vector(size: 25)
  3 LinearLayer vector(size: 5)
  4 Tanh      vector(size: 5)
  5 LinearLayer vector(size: 1)
  Output     scalar
]

```

**Mathematica Code 4.64**

```

Input      (* Regression with Neural Network: *)

(* The code aims to illustrate the training of a neural network for a regression
task on a 2D dataset, visualize the learned function, and evaluate the network's
prediction on a specific input: *)

(* Generate a dataset with input-output pairs for a 2D function: *)
trainingData=Flatten@Table[
  {x,y}=>Exp[-Norm[{x,y}]],
  {x,-3,3,.005},
  {y,-3,3,.005}
];

(* Define a neural network with three layers: *)
regressionNetwork=NetChain[{32,Tanh,1}];

(* Train the neural network using the generated dataset: *)
trainedNetwork=NetTrain[
  regressionNetwork,
  trainingData,
  BatchSize->1024,
  MaxTrainingRounds->10,
  LearningRate->0.01,
  LossFunction->MeanSquaredLossLayer[],
  Method->"SGD"
];

(* Evaluate the trained network on a specific input: *)
prediction=trainedNetwork[{1,0}]

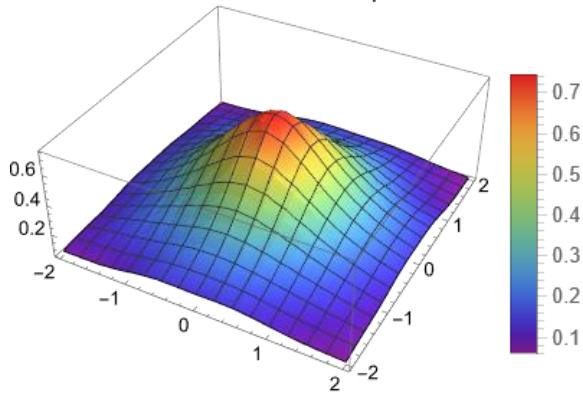
(* Plot the prediction of the network as a function of x and y: *)
Plot3D[
  trainedNetwork[{x,y}],
  {x,-2,2},
  {y,-2,2},
  ImageSize->250,
  ColorFunction->"Rainbow",
  PlotLegends->Automatic,
  NormalsFunction->None,
  PlotLabel->"3D Plot of Predicted Output"
]

```

Output 0.385327

Output

3D Plot of Predicted Output



Mathematica Code 4.65

Input (* Neural Network Training and Performance Visualization: *)

(* The code demonstrates how a neural network can be trained on noisy data to learn the underlying patterns, and how the trained network can be used to make predictions on new input data. The visualizations help in assessing the performance of the trained network and comparing it to the noisy dataset: *)

(* Set a random seed for reproducibility: *)
SeedRandom[123];

(* Generate a noisy two-dimensional dataset with random noise: *)
noisyData=Flatten[
Table[
{{x,y,Sin[x]+Cos[y]+RandomVariate[NormalDistribution[0,0.05]]}},
{x,-3,3,0.2},
{y,-3,3,0.2}
],
2
];

(* Generate a dataset with input-output pairs for a 2D function with added random noise: *)
trainingData=Flatten[
Table[
{{x,y}→Sin[x]+Cos[y]+RandomVariate[NormalDistribution[0,0.05]]},
{x,-3,3,0.2},
{y,-3,3,0.2}],
2
];

(* Define a neural network for regression: *)
regressionNetwork=NetChain[
{
LinearLayer[15],ElementwiseLayer[Ramp],
LinearLayer[15],ElementwiseLayer[Tanh],
LinearLayer[1]
}]

(* Train the neural network: *)
trainedNetwork=NetTrain[

```

regressionNetwork,
trainingData,
MaxTrainingRounds->50
]

(* Visualization: *)

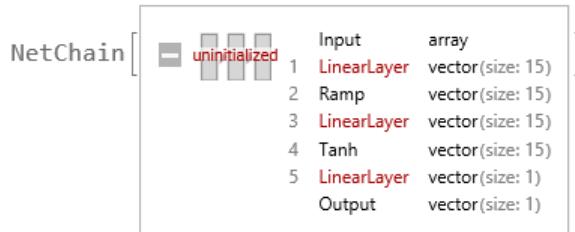
(* 3D Point Plot of the Noisy Data: *)
ListPointPlot3D[
noisyData,
ColorFunction->"Rainbow",
PlotLegends->Automatic,
ImageSize->250,
PlotLabel->"3D Point Plot of Noisy Data"
]

(* Contour Plot of the Noisy Data: *)
ListContourPlot[
noisyData,
ContourStyle->\{White\},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
LabelStyle->Directive[Black,10],
PlotLabel->"Contour Plot of Noisy Data",
ImageSize->250
]

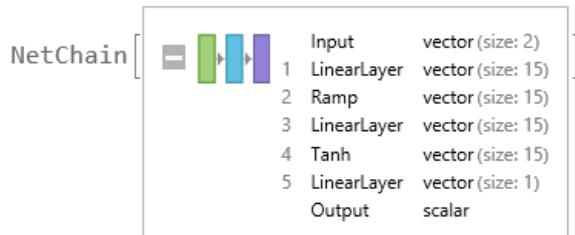
(* Contour Plot of the Trained Network Data: *)
ContourPlot[
trainedNetwork[{x,y}],
{x,-3,3},
{y,-3,3},
ContourStyle->\{White\},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
LabelStyle->Directive[Black,10],
ImageSize->250,
PlotLabel->"Contour Plot of Trained Network Data"
]

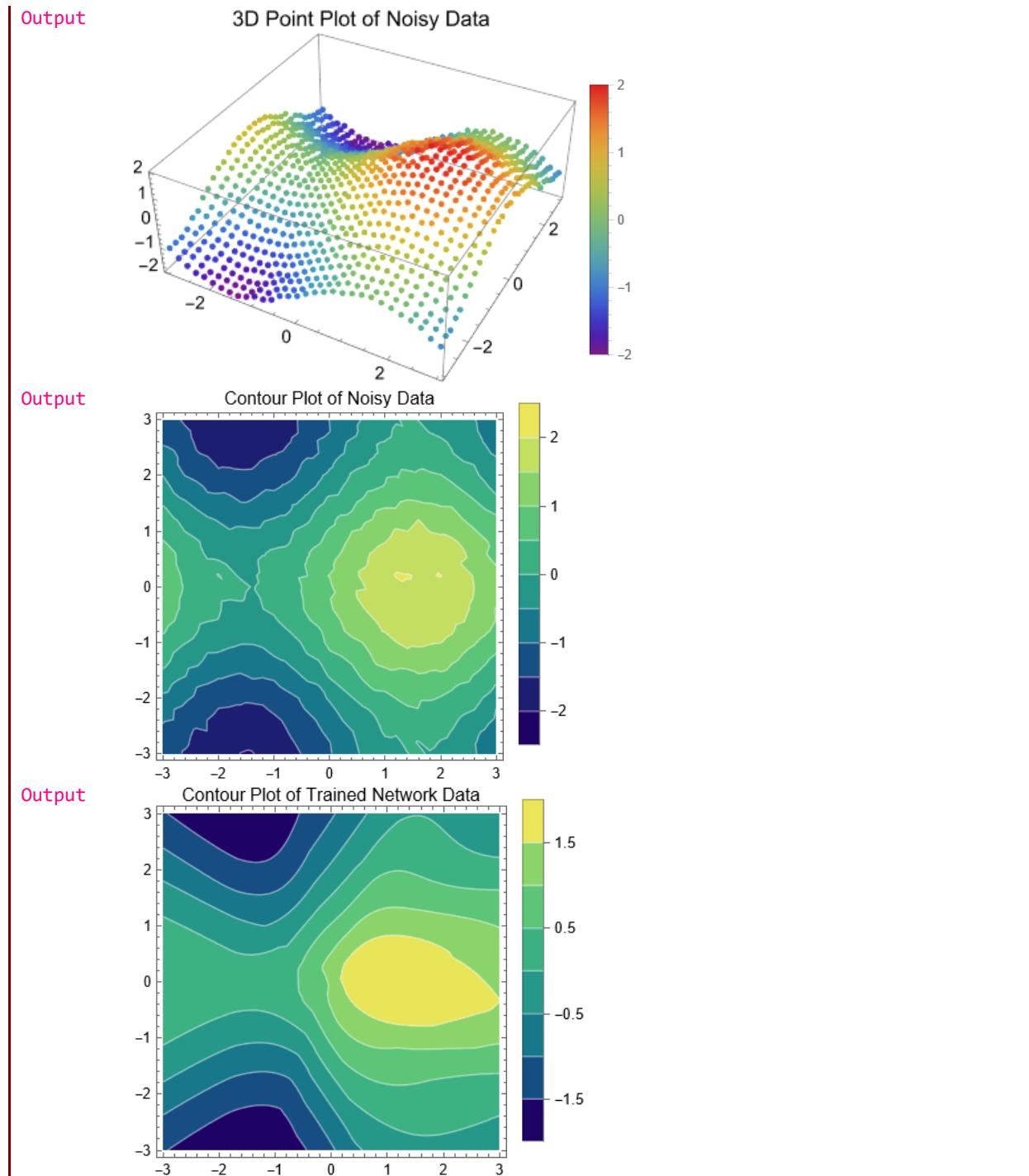
```

Output



Output



**Mathematica Code 4.66**

```
Input (* Neural Network Training and Analysis: *)
```

```
(* The code demonstrates how to use the NetTrainResultsObject to query and access various training-related information after training a neural network: *)
```

```
(* Generate a dataset for a 2D exponential decay function: *)
```

```
data=Flatten@Table[
 {x,y}->Exp[-Norm[{x,y}]],
```

```

{x,-3,3,.005},
{y,-3,3,.005}
];

(* Define a neural network for regression: *)
regressionNetwork=NetChain[{32,Tanh,1}];

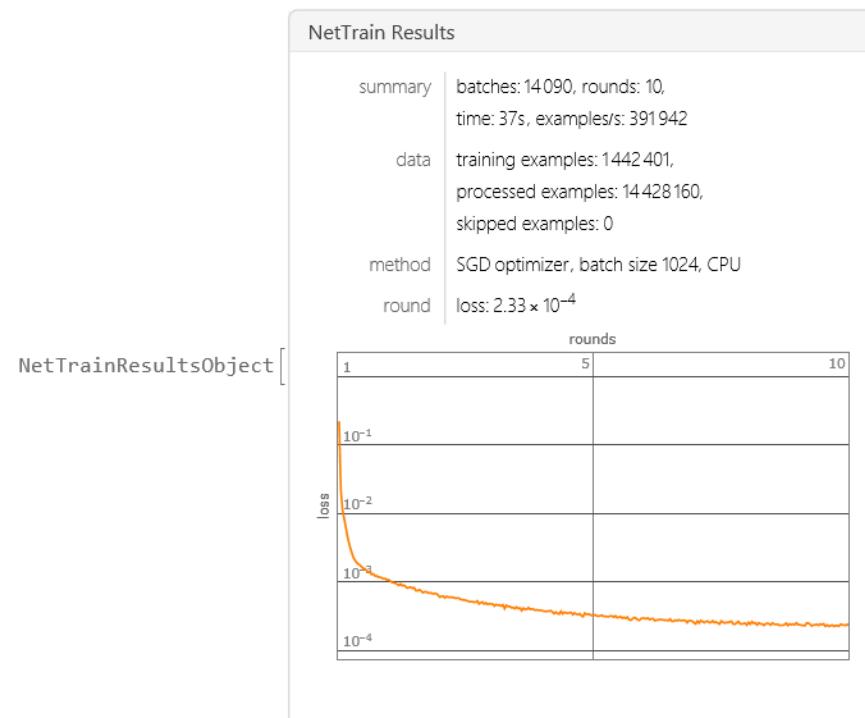
(* Train the neural network and obtain training results: *)
trainingResults=NetTrain[
  regressionNetwork,
  data,
  All,(* Train all available properties for the NetTrainResultsObject: *)
  BatchSize->1024,
  MaxTrainingRounds->10,
  LearningRate->0.01,
  LossFunction->MeanSquaredLossLayer[],
  Method->"SGD"
]

(* Query the results object for specific properties: *)
batchesPerRound=trainingResults["BatchesPerRound"]
batchSize=trainingResults["BatchSize"]
lossPlot=trainingResults["LossPlot"]
optimizationMethod=trainingResults["OptimizationMethod"]
initialLearningRate=trainingResults["InitialLearningRate"]
totalTrainingTime=trainingResults["TotalTrainingTime"]
trainedNet=trainingResults["TrainedNet"]
trainingNet=trainingResults["TrainingNet"]

(* Get a list of all available properties: *)
allProperties=trainingResults["Properties"]

```

Output



Output 1409

Output 1024

**Mathematica Code 4.67**

Input (* The code demonstrates the training and usage of a simple perceptron for binary classification and visualize its behavior in terms of probability across a range of inputs: *)

```
(* Define a perceptron for binary classification: *)
```

```

classificationPerceptron=NetChain[{LinearLayer[],LogisticSigmoid}];

(* Train the perceptron using labeled input-output pairs: *)
trainedPerceptron=NetTrain[
  classificationPerceptron,
  {1->False,2->False,3->True,4->True}
]

(* Predict whether a new input belongs to the positive class (True or False): *)
predictionSingleInput=trainedPerceptron[3.5]

(* Obtain the probability of the input being True by disabling the NetDecoder: *)
probabilitySingleInput=trainedPerceptron[3.5,None]

(* Make several predictions for a list of inputs at once: *)
predictionsMultipleInputs=trainedPerceptron[{-1,0,1,2,3,4,5,6}]

(* Visualize the probability as a function of the input: *)
Plot[
  trainedPerceptron[x,None],
  {x,0,5},
  ImageSize->250,
  PlotLabel->"Probability of Positive Class",
  AxesLabel->{"Input","Probability"},
  PlotStyle->Directive[Blue,Thick]
]

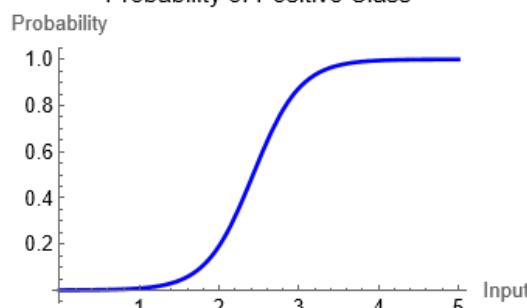
```

Output



Output
Output
Output
Output

True
0.974409
{False, False, False, False, True, True, True, True}
Probability of Positive Class

**Mathematica Code 4.68**

Input (* The code serves to illustrate the impact of nonlinearity in classification networks, emphasizing the importance of incorporating nonlinear activation functions to handle complex relationships in data, especially in scenarios where classes are not linearly separable. Create a synthetic training set consisting of points on a disk, separated into two classes by the circle r=0.5: *)

```

(* Generate a synthetic training set: *)
points=RandomPoint[Disk[],1000];
labels=Thread[Map[Norm,points]<0.5];

```

```

trainingData=points->labels;

(* Visualize the synthetic training set: *)
ListPlot[
{Pick[points,labels,True],Pick[points,labels,False]},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},PlotLegends->{"Class 1","Class 2"}
]

(* Create a network with two LinearLayer layers and a final transformation into a probability using an ElementwiseLayer: *)
classificationNet=NetChain[
{LinearLayer[20],LinearLayer[],ElementwiseLayer[LogisticSigmoid]},
"Output"->NetDecoder["Boolean"]
]

(* Train the network on the data: *)
trainedClassifier=NetTrain[classificationNet,trainingData];

(* Visualize the trained network's decision boundary. The net was not able to separate the two classes: *)
(* Explain the limitations of a network with two LinearLayer layers without an intervening nonlinearity: *)
ContourPlot[
trainedClassifier[{x,y},None],
{x,-1,1},
{y,-1,1},
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
LabelStyle->Directive[Black,10],
ImageSize->250,
PlotLabel->"Trained Classifier Decision Boundary"
]

(* Because LinearLayer is an affine layer, stacking the two layers without an intervening nonlinearity is equivalent to using a single layer. A single line in the plane cannot separate the two classes, which is the level set of a single LinearLayer: *)

(* Create a similar net with a Tanh nonlinearity between the two layers: *)
nonlinearClassifier=NetChain[
{
  LinearLayer[20],ElementwiseLayer[Tanh],
  LinearLayer[],ElementwiseLayer[LogisticSigmoid]
},
"Output"->NetDecoder["Boolean"]
]

(* The net can now separate the classes: *)
trainedNonlinearClassifier=NetTrain[nonlinearClassifier,trainingData];

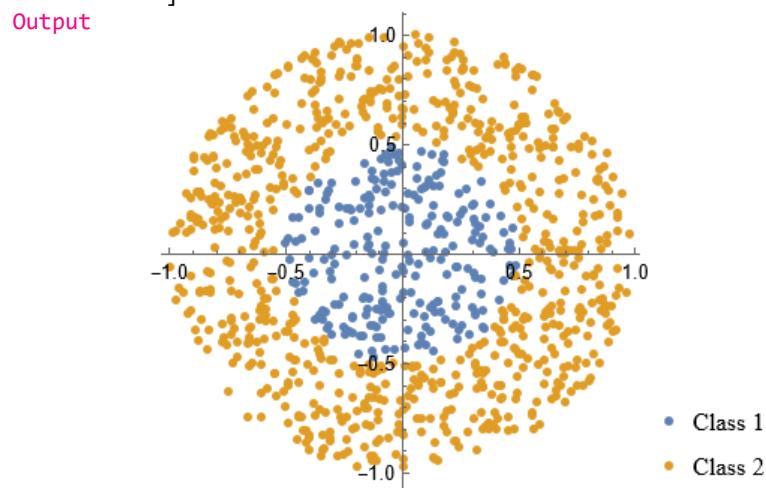
(* Visualize the decision boundary of the trained network with nonlinearity: *)
ContourPlot[
trainedNonlinearClassifier[{x,y},None],
{x,-1,1},

```

```

{y,-1,1},
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
PlotLegends->Automatic,
LabelStyle->Directive[Black,10],
ImageSize->250,
PlotLabel->"Trained Nonlinear Classifier Decision Boundary"
]

```

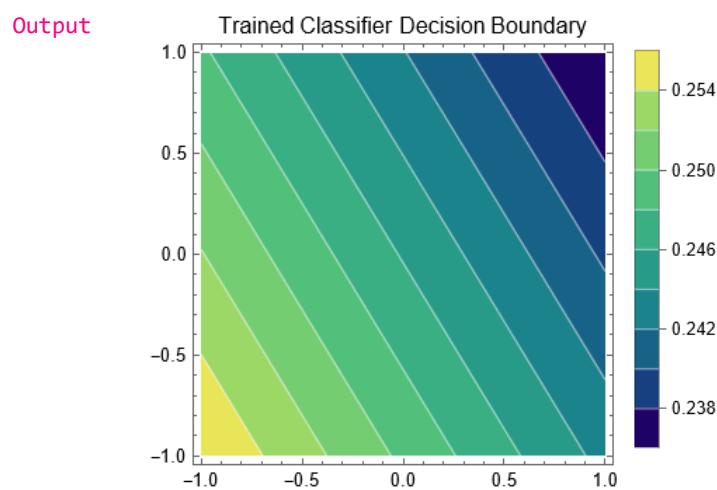


Output

```

NetChain [ [uninitialized] Input array
           1 LinearLayer vector(size: 20)
           2 LinearLayer real
           3 LogisticSigmoid real
           Output boolean ]

```



Output

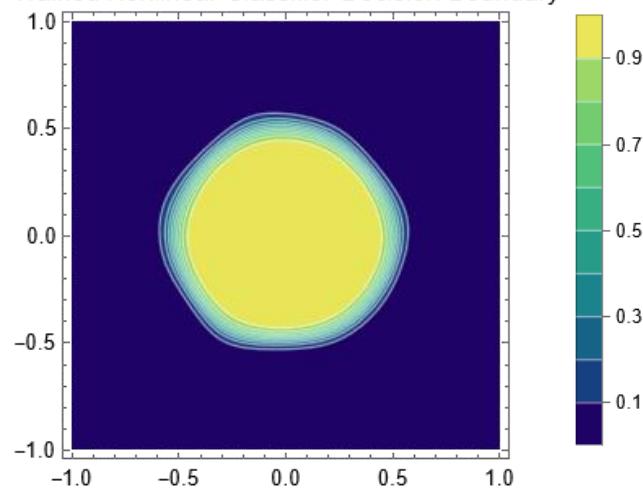
```

NetChain [ [uninitialized] Input array
           1 LinearLayer vector(size: 20)
           2 Tanh
           3 LinearLayer real
           4 LogisticSigmoid real
           Output boolean ]

```

Output

Trained Nonlinear Classifier Decision Boundary



CHAPTER 5

CHALLENGES IN NEURAL NETWORK OPTIMIZATION

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 4, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation.

Neural Networks (NNs) have revolutionized various fields, from image recognition to natural language processing, by enabling machines to learn complex patterns and make decisions with human-like accuracy. However, despite their remarkable capabilities, training NNs effectively remains a challenging task. This chapter delves into the myriad challenges encountered during the optimization process of NNs, exploring key concepts such as activation function (AF) saturation, vanishing and exploding gradients, weight initialization methods [47,48,49-53], non-zero centered AFs [54], feature scaling techniques, including standardization, normalization, and whitening [55-61], and normalization methods like Batch Normalization (BN) [62] and Layer Normalization (LN) [63].

- One of the fundamental challenges in NN optimization is AF saturation. AFs play a crucial role in introducing non-linearity to the model, enabling it to learn complex patterns. However, certain AFs, such as the Sigmoid or hyperbolic tangent (Tanh) functions, tend to saturate when the input values are too large or too small, leading to vanishing gradients and hampering the learning process.
- The vanishing and exploding gradients problem is another significant hurdle faced during NN training. In deep NNs, gradients can diminish exponentially or explode during backpropagation (BP), making it challenging to update the weights effectively. This phenomenon hinders the convergence of the model and affects its ability to generalize to unseen data.
- While non-zero centered AFs like ReLU and its variants indeed alleviate the vanishing gradient problem (VGP), they can introduce another issue known as "zig-zag" updates. These updates occur when the neuron's output oscillates between positive and negative values, causing the Gradient Descent (GD) updates to zig-zag back and forth. This oscillatory behavior can potentially slow down the learning process, as the network struggles to converge towards the optimal solution.
- Effective weight initialization is crucial for mitigating the issues of vanishing and exploding gradients. Common weight initialization techniques, such as random initialization, aim to set initial weights to small values to prevent saturation and maintain stable gradients during training. Kaiming or He initialization is a popular weight initialization method designed specifically for deep NNs. It initializes the weights of each layer based on the number of input units, effectively addressing the vanishing and exploding gradients problem and promoting faster convergence. Xavier or Glorot initialization is another widely used technique for weight initialization. It sets the initial weights using a uniform or normal distribution, scaled based on the number of input and output units, thus ensuring stable gradients and facilitating smoother training.
- Feature scaling techniques, including standardization, normalization, and whitening, are essential for preprocessing input data and enhancing the convergence of NNs. These methods ensure that input features are on similar scales, preventing certain features from dominating the learning process.
- BN and LN are techniques that normalize the activations of each layer, effectively stabilizing the learning process and accelerating convergence. These techniques mitigate the effects of internal covariate shift and facilitate smoother optimization of NNs.

In this chapter, building upon the foundational concepts established in earlier chapters, we delve deeper into the realm of NN construction using Mathematica, exploring advanced functionalities and techniques to enhance the efficiency and effectiveness of our models.

- Our first focus will be on monitoring training progress, a crucial aspect of NN development. We will discuss methodologies and tools to track key performance metrics such as loss, accuracy, and convergence rates in real-time. By closely monitoring training progress, developers can gain valuable insights into the behavior of their models, enabling informed decisions and efficient optimization strategies.
- We will introduce `NetPort` and `NetExtract`, which serve as crucial components for managing the flow of data within NN architectures. `NetPort` enables us to designate input and output ports within our networks, facilitating the seamless integration of various data sources and outputs. `NetExtract`, on the other hand, allows for the extraction of specific layers or parameters from networks, enabling the reusability and customization of complex architectures.
- Next, we will discuss `TrainingProgressMeasurements` and `TrainingProgressFunction`, which provide invaluable insights into the training process of NNs. These tools enable real-time monitoring and analysis of key performance metrics, empowering developers to fine-tune models for optimal results efficiently.
- Furthermore, we will explore the `NetTrainResultsObject`, a comprehensive representation of the training outcomes generated by the `NetTrain` function. This object encapsulates crucial information such as training/validation losses, accuracies, and other performance metrics, facilitating in-depth analysis and evaluation of model performance.
- Next, we address the challenge of saturation and vanishing gradients during training, which can hinder the convergence of NNs, particularly in deep architectures. We explore techniques to mitigate these issues, including advanced weight initialization methods. By effectively managing saturation and vanishing gradients, developers can foster more stable and reliable training dynamics, leading to improved model performance.
- In addition, we will delve into the initialization techniques, Xavier and Kaiming, using the `NetInitialize` function. These techniques play a pivotal role in setting the initial weights of NN layers, influencing the convergence and stability of the training process significantly.
- Furthermore, we will introduce the `BatchNormalizationLayer`, a fundamental tool for improving the training efficiency and generalization of NNs. BN mitigates the issues of internal covariate shift, promoting faster convergence and better performance across various tasks. The `BatchNormalizationLayer` can be easily incorporated into NN architectures using the `NetChain` or `NetGraph` constructs. It can be placed after a fully connected layer, or any other layer where normalization is desired.
- Finally, we delve into feature scaling techniques, specifically standardization, whitening, and Mahalanobis distances. Feature scaling plays a critical role in ensuring the stability and convergence of NNs by normalizing input data distributions. We examine the impact of different scaling methodologies on model performance and explore best practices for incorporating them into the training pipeline.

Throughout this chapter, we will provide practical examples and demonstrations to illustrate the application of these techniques. By mastering these advanced functionalities, readers will gain the necessary expertise to design, train, and optimize sophisticated NNs using Mathematica, empowering them to tackle complex problems across diverse domains effectively.

Unit 5.1

NetPort and NetExtract

NetPort["port"]

represents the specified input or output port for a complete net.

NetPort[{n, "port"}]

represents the specified port for layer number n in a NetGraph or similar construct.

NetPort[{"name", "port"}]

represents the specified port for the layer with the specified name.

NetPort[spec, port]

is treated as equivalent to NetPort[{spec, port}].

Remarks:

- **NetPort[{layer, "port"}]**: Refers to the named output port of a specific layer when used on the left-hand side of a rule. When used on the right-hand side of a rule, it refers to a named input port of a layer.
- **NetPort["port"]**: Refers to an input of the entire graph when used on the left-hand side of a rule. When used on the right-hand side of a rule, it refers to an output of the entire graph.
- **net[data, NetPort[oport]]** : Used to obtain the value of the net at a specific output port (oport) when it's applied to the specified data. Here, oport must refer to an output port, a subnet of the net, or a layer.
- **net[data, {NetPort[oport1], NetPort[oport2], ...}]**: Returns an association where the keys are the specified output ports (oport1, oport2, etc.), and the values are the values of the net at those output ports when the net is applied to the specified data.

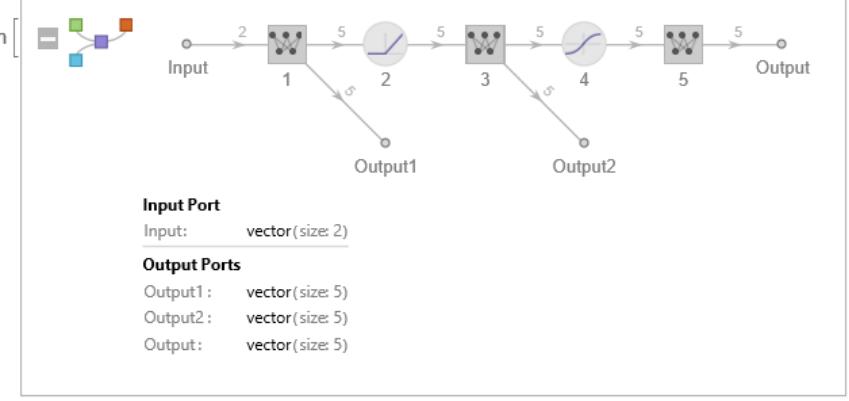
Mathematica Code 5.1

```
Input (* The code initializes and evaluates a neural network named net. This network, defined within a NetGraph, consists of multiple layers including LinearLayers and activation layers (ElementwiseLayer[Ramp] and ElementwiseLayer[Tanh]), forming a feedforward architecture with a sequential flow of information through the layers. Two distinct output ports ("Output1" and "Output2") are specified, enabling the retrieval of specific outputs from the network during evaluation. Additionally, the code demonstrates the initialization of the network parameters using NetInitialize, ensuring that the network is prepared for evaluation with the provided inputs. Finally, the network is evaluated with a given input, producing the requested outputs corresponding to the specified ports: *)
```

```
net=NetInitialize@NetGraph[
  {
    LinearLayer[5,"Input" -> 2], ElementwiseLayer[Ramp],
    LinearLayer[5], ElementwiseLayer[Tanh],
    LinearLayer[5]
  },
  {1->NetPort["Output1"], 3->NetPort["Output2"], 1->2->3->4->5}
]
```

```
net[{-1.2, 3.1},
{NetPort["Output1"], NetPort["Output2"], NetPort["Output"], NetPort[All, "Output"]}]
```

Output

```
NetGraph[]
  Input --> 1
  1 --> 2
  2 --> 3
  3 --> 4
  4 --> 5
  5 --> Output
  1 --> Output1
  1 --> Output2
```

Input Port
Input: vector(size: 2)

Output Ports

- Output1: vector(size: 5)
- Output2: vector(size: 5)
- Output: vector(size: 5)

Output

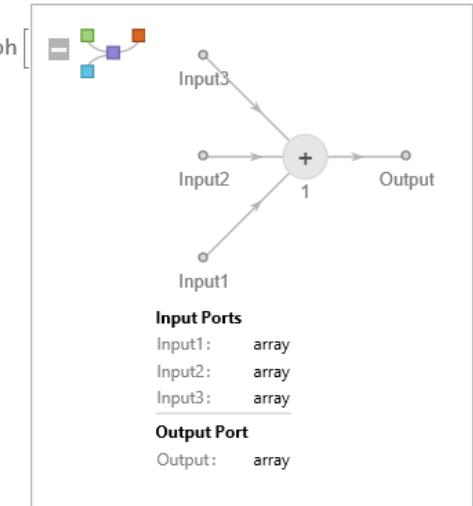
```
<| Output1 -> {0.391137, 6.67612, -7.2958, 1.75662, -5.31164},
  Output2 -> {-0.1352, -1.01155, -7.02377, 0.932496, 1.12231},
  Output -> {0.448372, -0.832601, 0.217107, -0.230463, -0.525316},
  NetPort[{1, Output}] -> {0.391137, 6.67612, -7.2958, 1.75662, -5.31164},
  NetPort[{2, Output}] -> {0.391137, 6.67612, 0., 1.75662, 0.},
  NetPort[{3, Output}] -> {-0.1352, -1.01155, -7.02377, 0.932496, 1.12231},
  NetPort[{4, Output}] -> {-0.134382, -0.766404, -0.999998, 0.731756, 0.80837},
  NetPort[{5, Output}] -> {0.448372, -0.832601, 0.217107, -0.230463, -0.525316}
  |>
```

Mathematica Code 5.2

Input (* The code initializes and evaluates a neural network named net. The network, defined within a NetGraph, consists of a single layer represented by TotalLayer[], which computes the total sum of its inputs. Three custom-named input ports, "Input1", "Input2", and "Input3", are defined, enabling the network to receive input data through specific channels. These inputs are then aggregated by the TotalLayer[]. After initialization, the network is evaluated with the input {4,5,5}, resulting in the computation of the total sum of the input data: *)

```
net=NetInitialize@NetGraph[{TotalLayer[]}, {NetPort["Input1"]->1, NetPort["Input2"]->1, NetPort["Input3"]->1}]
net[{4,5,5}]
```

Output

```
NetGraph[]
  Input3 --> 1
  Input2 --> 1
  Input1 --> 1
  1 --> Output
```

Input Ports

- Input1: array
- Input2: array
- Input3: array

Output Port

- Output: array

Output

```
14.
```

Mathematica Code 5.3

Input

```
(* The code demonstrates the creation and evaluation of a neural network chain (NetChain). The network consists of three element-wise layers: Tanh, Cos, and Sin, arranged sequentially to process input data through a series of element-wise operations. Through the use of NetPort, the code showcases different methods for retrieving outputs from specific layers or obtaining outputs from all layers within the network: *)

(* Create a NetChain: *)
net=NetChain[{ElementwiseLayer[Tanh],ElementwiseLayer[Cos],ElementwiseLayer[Sin]}]
]

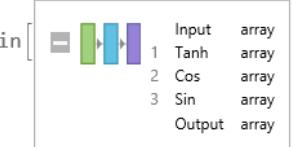
(* Obtain the output of the first layer of the net when applied to data: *)
net[{-1.2,3.1},NetPort[1,"Output"]]

(* This is equivalent to: *)
Tanh[{-1.2,3.1}]

(* Obtain the output of the first two layers of the net: *)
net[{-1.2,3.1},{NetPort[1,"Output"],NetPort[2,"Output"]}]

(* Obtain the outputs of all layers in the net: *)
net[{-1.2,3.1},NetPort[All,"Output"]]
```

Output

```
NetChain[]
```

	Input	array
1	Tanh	array
2	Cos	array
3	Sin	array
Output		array

Output

```
{-0.833655, 0.995949}
```

Output

```
{-0.833655, 0.995949}
```

Output

```
<|
NetPort[{1,Output}]->{-0.833655, 0.995949},
NetPort[{2,Output}]->{0.672174, 0.543706}
|>
```

Output

```
<|
NetPort[{1,Output}]->{-0.833655, 0.995949},
NetPort[{2,Output}]->{0.672174, 0.543706},
NetPort[{3,Output}]->{0.622689, 0.517311}
|>
```

NetExtract[layer, "param"]

extracts the value of a parameter for the specified net layer.

NetExtract[net, lspec]

extracts the layer identified by lspec from within the NetGraph or NetChain object net.

NetExtract[net, {lspec, "param"}]

extracts the value of the parameter param from the layer identified by lspec in net.

Remarks:

- The layer specification can be an integer indicating the nth layer or a string indicating a named layer.
- Parameter specifications can be the names of any of the arrays or options contained within a layer.

Mathematica Code 5.4

Input

```
(* The code constructs a neural network model using Wolfram Language's NetChain,
comprising layers with distinct names such as linear and activation layers. It
subsequently extracts various components from this network, including the first and
last layers, a specific named layer, the function utilized within a particular
layer, a selection of layers, and finally, all layers. Through these operations,
the code showcases the flexibility of Wolfram Language's neural network framework
in both constructing and inspecting neural network architectures, thereby
facilitating tasks such as layer manipulation, function extraction, and overall
network comprehension: *)

(* Define a neural network using NetChain with multiple layers: *)
chain=NetChain[
<|
  (* Define a linear layer with 4 output neurons: *)
  "linearLayer1"->LinearLayer[4],

  (* Define an activation layer with Ramp activation function: *)
  "activationLayer1"->Ramp,

  (* Define another linear layer with 7 output neurons: *)
  "linearLayer2"->LinearLayer[7],

  (* Define another activation layer with Tanh activation function: *)
  "activationLayer2"->Tanh,

  (* Define a linear layer with 1 output neuron: *)
  "linearLayer3"->LinearLayer[1]
|>
]

(* Extract the first layer of the neural network: *)
NetExtract[chain,1]

(* Extract the last layer of the neural network: *)
NetExtract[chain,-1]

(* Extract a specific named layer ("linearLayer1") from the neural network: *)
NetExtract[chain,"linearLayer1"]

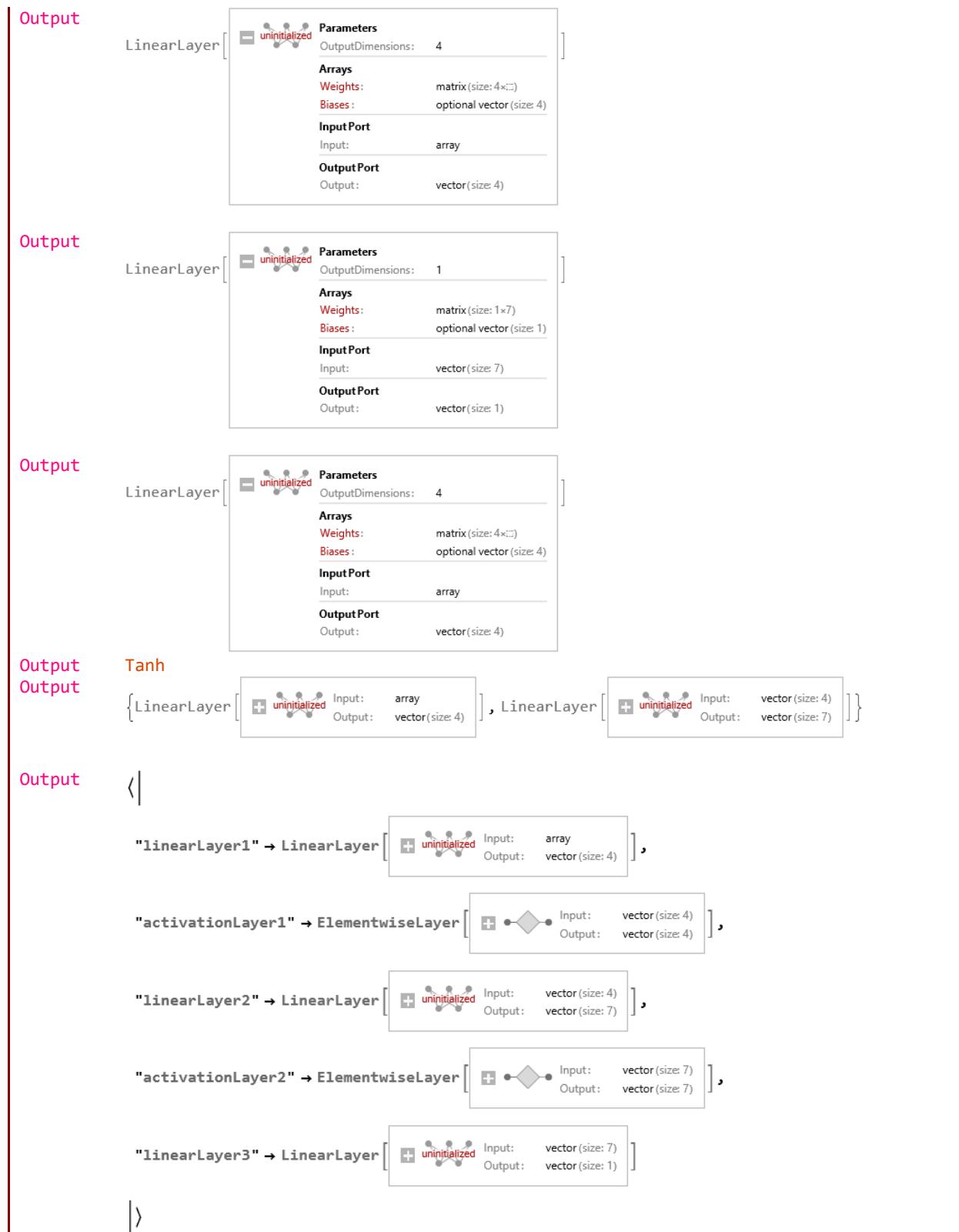
(* Extract the function utilized in a particular layer ("activationLayer2") of
the neural network: *)
NetExtract[chain,{4,"Function"}]

(* Extract multiple layers (the first and third layers) from the neural network: *)
NetExtract[chain,{{1},{3}}]

(* Extract all layers of the neural network: *)
NetExtract[chain,All]
```

Output

NetChain	[uninitialized	Input	array
		linearLayer1	LinearLayer	vector (size: 4)
		activationLayer1	Ramp	vector (size: 4)
		linearLayer2	LinearLayer	vector (size: 7)
		activationLayer2	Tanh	vector (size: 7)
		linearLayer3	LinearLayer	vector (size: 1)
			Output	vector (size: 1)

**Mathematica Code 5.5**

Input (* The code constructs a NetGraph object, which represents a neural network model with named layers. The network consists of two linear layers denoted as "linear1"

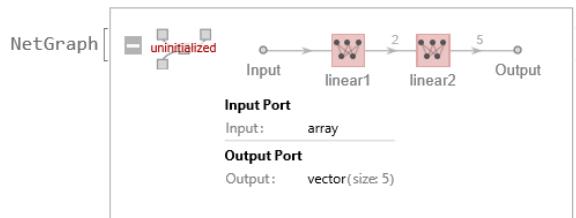
and "linear2", connected sequentially where the output of "linear1" serves as the input to "linear2". The subsequent operations involve extracting specific components from the network. Firstly, it extracts the layer named "linear1", which retrieves the first linear layer from the network. Secondly, it extracts all layers from the network, providing a list containing both "linear1" and "linear2". These operations exemplify the ability to access individual layers or all layers collectively within a NetGraph object, facilitating tasks such as layer inspection and manipulation:
*)

```
(* Create a NetGraph object with named layers: *)
graph=NetGraph[<|"linear1" -> LinearLayer[2], "linear2" -> LinearLayer[5]|>, {"linear1" -> "linear2"}]

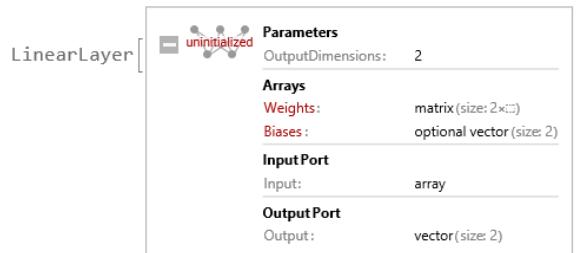
(* Extract a named layer: *)
NetExtract[graph, "linear1"]

(* Extract all the layers: *)
NetExtract[graph, All]
```

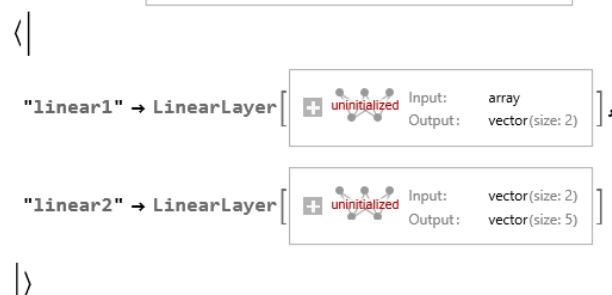
Output



Output



Output

**Mathematica Code 5.6**

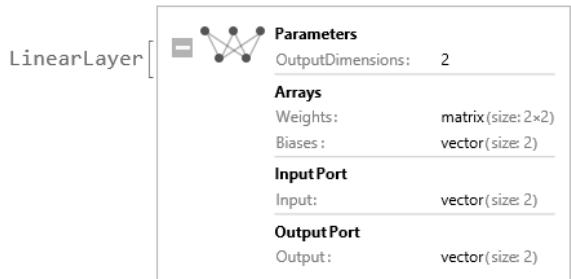
Input

(* The code initializes a linear layer with two input neurons and two output neurons. The layer is randomly initialized, and then the weight matrix is extracted from the layer. The weight matrix represents the parameters of the layer that map inputs to outputs. By using NetExtract with the argument "Weights", only the weight matrix of the layer is retrieved. Finally, the weight matrix is converted to a standard matrix format using Normal to display its values: *)

```
(* Create a randomly initialized layer: *)
layer=NetInitialize@LinearLayer[2,"Input" -> 2]
```

```
(* Extract the weight matrix from the layer: *)
NetExtract[layer,"Weights"]//Normal
```

Output



```
Output {{-0.359447, -0.0499231}, {-1.12706, 1.08654}}
```

Mathematica Code 5.7

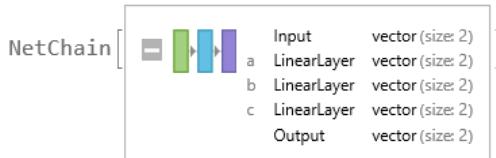
```
Input (* The code initializes a neural network chain (NetChain) with three layers, each
       having two neurons, and with a total input size of 2. The layers are named "a",
       "b", and "c". Following this initialization, the code extracts weights from the
       specified layers and all layers within the chain: *)
```

```
(* Create an initialized NetChain: *)
chain=NetInitialize@NetChain[<|"a"->2,"b"->2,"c"->2|>,"Input"->2]

(* Extract the weights from a specific layer: *)
NetExtract[chain,{2,"Weights"}]//Normal

(* Extract the weights from all layers: *)
NetExtract[chain,{All,"Weights"}]
```

Output



```
Output {{1.89365, -0.93114}, {-0.773974, 0.101079}}
```

Output

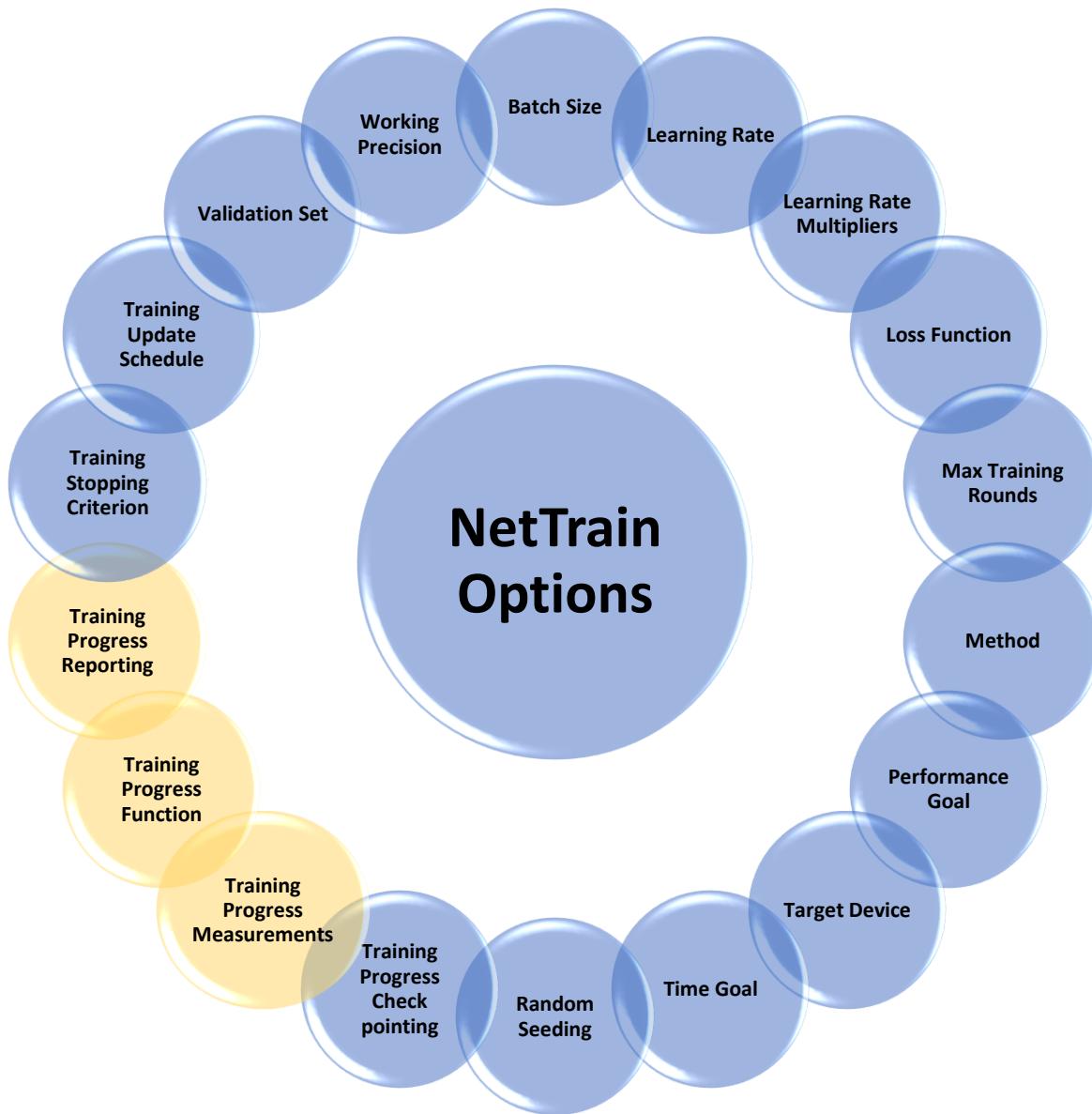
```
<|
  "a" → NumericArray [
    Type: Real32
    Dimensions: {2, 2}
  ],
  "b" → NumericArray [
    Type: Real32
    Dimensions: {2, 2}
  ],
  "c" → NumericArray [
    Type: Real32
    Dimensions: {2, 2}
  ]
|>
```

Unit 5.2

Monitor Training Progress

`NetTrain[net,{input1->output1,input2->output2,...}]`

trains the specified neural net by giving the inputi as input and minimizing the discrepancy between the outputi and the actual output of the net, using an automatically chosen loss function.



`TrainingProgressMeasurements` and `TrainingProgressFunction` are options used to monitor and customize the training process, while `NetTrainResultsObject` is the object containing information about the training results.

NetTrain[net,data,All]

gives a NetTrainResultsObject[...] that summarizes information about the training session.

NetTrainResultsObject[...]

represents an object generated by NetTrain that contains the trained net and other information about the training process.

NetTrainResultsObject[...][prop]

is used to look up property prop from the NetTrainResultsObject.

Properties supported include:

"ArraysLearningRateMultipliers"	an association of the learning rate multiplier used for each array in the network
"BatchesPerRound"	the number of batches contained in a single round
"BatchLossList"	a list of the mean losses for each batch update
"BatchMeasurementsLists"	list of training measurements associations for each batch update
"BatchSize"	the effective value of BatchSize
"BestValidationRound"	the training round corresponding to the trained net
"CheckpointingFiles"	list of checkpointing files generated during training
"ExamplesProcessed"	total number of examples processed during training
"FinalLearningRate"	the learning rate at the end of training
"FinalPlots"	association of plots for all losses and measurements
"InitialLearningRate"	the learning rate at the start of training
"LossPlot"	a plot of the evolution of the mean training loss
"MeanBatchesPerSecond"	the mean number of batches processed per second
"MeanExamplesPerSecond"	the mean number of input examples processed per second
"NetTrainInputForm"	an expression representing the originating call to NetTrain
"OptimizationMethod"	the optimization method used and all the related options
"ReasonTrainingStopped"	brief description of why training stopped
"RoundLoss"	the mean loss for the most recent round
"RoundLossList"	a list of the mean losses for each round
"RoundMeasurements"	association of training measurements for the most recent round
"RoundMeasurementsLists"	list of training measurements associations for each round
"RoundPositions"	the batch numbers corresponding to each round measurement
"TargetDevice"	the device used for training
"TotalBatches"	the total number of batches encountered during training
"TotalRounds"	the total number of rounds of training performed
"TotalTrainingTime"	the total time spent training, in seconds
"TrainedNet"	the optimal trained network found (default)
"TrainingExamples"	the number of examples in the training set
"TrainingNet"	the network as prepared for training
"ValidationExamples"	the number of examples in the validation set
"ValidationLoss"	the mean loss obtained on the ValidationSet for the most recent validation measurement
"ValidationLossList"	list of the mean losses on the ValidationSet for each validation measurement
"ValidationMeasurements"	association of training measurements on the ValidationSet after the most recent validation measurement
"ValidationMeasurementsLists"	list of training measurements associations on the ValidationSet for each validation measurement
"ValidationPositions"	the batch numbers corresponding to each validation measurement

`NetTrain[net,data,TrainingProgressFunction->f]`

TrainingProgressFunction

is an option for NetTrain that specifies a function to run periodically during training.

TrainingProgressFunction->f

specifies that f[assoc] is evaluated after every training round, where assoc is an association with the following keys:

<code>"AbsoluteBatch"</code>	total number of batches processed so far
<code>"Batch"</code>	current batch number within this round
<code>"BatchData"</code>	the most recent batch of data used to train the net
<code>"BatchesPerRound"</code>	the number of batches contained in a single round
<code>"BatchesPerSecond"</code>	the current training rate in batches per second
<code>"BatchIndices"</code>	list of indices in the original dataset for the most recent batch
<code>"BatchLoss"</code>	average loss of most recent batch
<code>"BatchLossList"</code>	list of batch losses for each batch update so far
<code>"BatchMeasurements"</code>	association of training measurements after the last batch update
<code>"BatchMeasurementsLists"</code>	list of training measurements associations for each batch update so far
<code>"BatchSize"</code>	the number of inputs contained in a batch
<code>"BestValidationRound"</code>	the training round corresponding to the current best net
<code>"CheckpointingFiles"</code>	list of checkpointing files generated so far
<code>"Event"</code>	the last event that occurred
<code>"ExampleLosses"</code>	losses taken by each example during training
<code>"ExamplesPerSecond"</code>	the training rate in input examples per second
<code>"ExamplesProcessed"</code>	total number of examples processed so far
<code>"Gradients"</code>	association between weight position within net and current gradient
<code>"GradientsRMS"</code>	root mean square of the weight gradients
<code>"GradientsVector"</code>	vector formed by flattening the current value of all weight gradients together
<code>"InitialLearningRate"</code>	the learning rate at the start of training
<code>"LearningRate"</code>	the current learning rate
<code>"MeanBatchesPerSecond"</code>	the mean number of batches processed per second
<code>"MeanExamplesPerSecond"</code>	the mean number of input examples processed per second
<code>"Net"</code>	current, partially trained network
<code>"OptimizationMethod"</code>	the name of the optimization method used
<code>"ProgressFraction"</code>	progress represented as a number between 0 and 1
<code>"Round"</code>	current round number
<code>"RoundLoss"</code>	average loss of most recent round
<code>"RoundLossList"</code>	list of round losses for each round so far
<code>"RoundMeasurements"</code>	association of training measurements for the training set after the last training round
<code>"RoundMeasurementsLists"</code>	list of training measurements associations for each round so far
<code>"TargetDevice"</code>	the device used for training
<code>"TimeElapsed"</code>	time elapsed since training began, in seconds
<code>"TimeRemaining"</code>	estimated time remaining, in seconds
<code>"TotalBatches"</code>	maximum number of training batches
<code>"TotalRounds"</code>	maximum number of training rounds
<code>"ValidationLoss"</code>	most recent validation loss
<code>"ValidationLossList"</code>	list of validation losses for each validation measurement so far
<code>"ValidationMeasurements"</code>	association of training measurements for the validation set
<code>"ValidationMeasurementsLists"</code>	list of training measurements associations for each validation measurement so far
<code>"Weights"</code>	association of current value of all weights
<code>"WeightsRMS"</code>	root mean square of the weights
<code>"WeightsLearningRateMultipliers"</code>	an association of the learning rate multiplier used for each weight
<code>"WeightsVector"</code>	vector formed by flattening the current value of all weights together

`NetTrain[net,data,"prop"]`

gives data associated with a specific property prop of the training session.

In `NetTrain[net,data,prop]`, the property prop can be any of the following:

<code>"TrainedNet"</code>	the optimal trained network found (default)
<code>"BatchesPerRound"</code>	the number of batches contained in a single round
<code>"BatchLossList"</code>	a list of the mean losses for each batch update
<code>"BatchMeasurementsLists"</code>	list of training measurements associations for each batch update
<code>"BatchPermutation"</code>	an array of the indices from the training data used to populate each batch
<code>"BatchSize"</code>	the effective value of BatchSize
<code>"BestValidationRound"</code>	the training round corresponding to the final trained net
<code>"CheckpointingFiles"</code>	list of checkpointing files generated during training
<code>"ExampleLosses"</code>	losses taken by each example during training
<code>"ExamplesProcessed"</code>	total number of examples processed during training
<code>"FinalLearningRate"</code>	the learning rate at the end of training
<code>"FinalNet"</code>	the most recent network generated in the training process, regardless of its performance on the validation set or other metrics
<code>"FinalPlots"</code>	association of plots for all losses and measurements
<code>"InitialLearningRate"</code>	the learning rate at the start of training
<code>"LossPlot"</code>	a plot of the evolution of the mean training loss
<code>"MeanBatchesPerSecond"</code>	the mean number of batches processed per second
<code>"MeanExamplesPerSecond"</code>	the mean number of input examples processed per second
<code>"NetTrainInputForm"</code>	an expression representing the originating call to NetTrain
<code>"OptimizationMethod"</code>	the name of the optimization method used
<code>"Properties"</code>	the full list of available properties
<code>"ReasonTrainingStopped"</code>	brief description of why training stopped
<code>"ResultsObject"</code>	a <code>NetTrainResultsObject[...]</code> containing a majority of the available properties in this table
<code>"RoundLoss"</code>	the mean loss for the most recent round
<code>"RoundLossList"</code>	a list of the mean losses for each round
<code>"RoundMeasurements"</code>	association of training measurements for the most recent round
<code>"RoundMeasurementsLists"</code>	list of training measurements associations for each round
<code>"RoundPositions"</code>	the batch numbers corresponding to each round measurement
<code>"TargetDevice"</code>	the device used for training
<code>"TotalBatches"</code>	the total number of batches encountered during training
<code>"TotalRounds"</code>	the total number of rounds of training performed
<code>"TotalTrainingTime"</code>	the total time spent training, in seconds
<code>"TrainingExamples"</code>	the number of examples in the training set
<code>"TrainingNet"</code>	the network as prepared for training
<code>"TrainingUpdateSchedule"</code>	the value of TrainingUpdateSchedule
<code>"ValidationExamples"</code>	the number of examples in the validation set
<code>"ValidationLoss"</code>	the mean loss obtained on the ValidationSet for the most recent validation measurement
<code>"ValidationLossList"</code>	list of the mean losses on the ValidationSet for each validation measurement
<code>"ValidationMeasurements"</code>	association of training measurements on the ValidationSet after the most recent validation measurement
<code>"ValidationMeasurementsLists"</code>	list of training measurements associations on the ValidationSet for each validation measurement
<code>"ValidationPositions"</code>	the batch numbers corresponding to each validation measurement
<code>"WeightsLearningRateMultipliers"</code>	an association of the learning rate multiplier used for each weight

Remarks:

- You can use any of the properties listed above as the third argument of `NetTrain` to retrieve specific information about the training process and results.

- Using these properties, you can analyze the performance of your neural network during training, inspect the trained model, and gather information that might be useful for further analysis or decision-making.
- An association of the form `<|"Property"->prop, "Form"->form, "Interval"->int|` can be used to specify a custom property whose value will be collected repeatedly during training.
- For a custom property, valid settings for prop can be any of the properties available in `TrainingProgressFunction`, or a user-defined function that is given the association of all the properties. Valid settings for form include `"List"`, `"TransposedList"` and `"Plot"`. Valid settings for `"Interval"` can be `"Batch"`, `"Round"` or a `Quantity[...]`. Supported units include `"Batches"`, `"Rounds"`, `"Percent"` and time units like `"Seconds"`, `"Minutes"` and `"Hours"`.

`NetTrain[net,data,TrainingProgressMeasurements->spec]`

TrainingProgressMeasurements

is an option for `NetTrain` that specifies measurements to make while training is in progress.

In `TrainingProgressMeasurements -> spec`, the following forms for spec are allowed:

<code>"measurement"</code>	a named, built-in measurement
<code>NetPort["output"]</code>	the value of an output port of the net
<code>NetPort["tdata"]</code>	the value of training data for the net
<code>NetPort[{lspec,"output"}]</code>	the value of an interior activation of the net
<code>NetPort[{lspec,"weight"}]</code>	the value of a weight array
<code>< "Measurement"->spec,... ></code>	a measurement with suboptions
<code>< "Measurement"->Function[...],... ></code>	a custom function to measure

For nets that contain a `CrossEntropyLossLayer`, the following built-in measurements are available:

<code>"Accuracy"</code>	fraction of correctly classified examples
<code>"Accuracy"->n</code>	fraction of examples with the correct result in the top n
<code>"AreaUnderROCCurve"</code>	area under the ROC curve for each class
<code>"CohenKappa"</code>	Cohen's kappa coefficient
<code>"ConfusionMatrix"</code>	counts c_{ij} of class i examples classified as class j
<code>"ConfusionMatrixPlot"</code>	plot of the confusion matrix
<code>"Entropy"</code>	entropy measured in nats
<code>"ErrorRate"</code>	fraction of incorrectly classified examples
<code>"ErrorRate"->n</code>	fraction of examples with the incorrect result in the top n
<code>"F1Score"</code>	F1 score for each class
<code>"FScore"->β</code>	$F\beta$ score for each class
<code>"FalseDiscoveryRate"</code>	false discovery rate for each class
<code>"FalseNegativeNumber"</code>	number of false negative examples
<code>"FalseNegativeRate"</code>	false negative rate for each class
<code>"FalseOmissionRate"</code>	false omission rate for each class
<code>"FalsePositiveNumber"</code>	number of false positive examples
<code>"FalsePositiveRate"</code>	false positive rate for each class
<code>"Informedness"</code>	informedness for each class
<code>"Markedness"</code>	markedness for each class
<code>"MatthewsCorrelationCoefficient"</code>	Matthews correlation coefficient for each class
<code>"NegativePredictiveValue"</code>	negative predictive value for each class
<code>"Perplexity"</code>	exponential of the entropy
<code>"Precision"</code>	precision for each class
<code>"Recall"</code>	recall rate for each class
<code>"ROCCurve"</code>	receiver operating characteristics (ROC) curve for each class
<code>"ROCCurvePlot"</code>	plot of the ROC curve
<code>"ScottPi"</code>	Scott's pi coefficient
<code>"Specificity"</code>	specificity for each class
<code>"TrueNegativeNumber"</code>	number of true negative examples
<code>"TruePositiveNumber"</code>	number of true positive examples

For nets that contain a `MeanSquaredLossLayer` or `MeanAbsoluteLossLayer`, the following built-in measurements are available:

<code>"FractionVarianceUnexplained"</code>	the fraction of output variance left unexplained by the net
<code>"IntersectionOverUnion"</code>	intersection over union for bounding boxes
<code>"MeanDeviation"</code>	mean absolute value of the residuals
<code>"MeanSquare"</code>	mean square of the residuals
<code>"RSquared"</code>	coefficient of determination
<code>"StandardDeviation"</code>	root mean square of the residuals

During the training process, `NetTrain` will compute these measurements at various intervals (e.g., after each training batch or after each training epoch) and provide them as part of the training progress output. This information can help you monitor the performance of your neural network as it learns from the data and make decisions about training parameters or model architecture.

Mathematica Code 5.8

```
Input (* The code demonstrates the training of a neural network using a noisy dataset and
       subsequent analysis of the training results. Initially, a synthetic dataset with
       added noise based on a Gaussian distribution is generated. Subsequently, a neural
       network architecture is defined with two hidden layers, each comprising 150 units
       and utilizing Tanh activation functions. The neural network is then trained using
       the noisy dataset, and the training results are stored in mlpTrainingResults. The
       code further inspects various properties of the trained network, including the loss
       plot, mean batches per second, optimization method, and the trained network itself:
*)

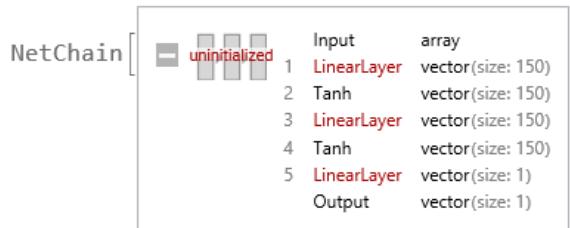
(* Generate a dataset with noise based on a Gaussian distribution: *)
noisyData=Table[x->Exp[-x^2]+RandomVariate[NormalDistribution[0,.15]],{x,-
3,3,.2}];

(* Define a neural network architecture with two hidden layers of 150 units each
   and Tanh activation functions: *)
mlp=NetChain[{150,Tanh,150,Tanh,1}]

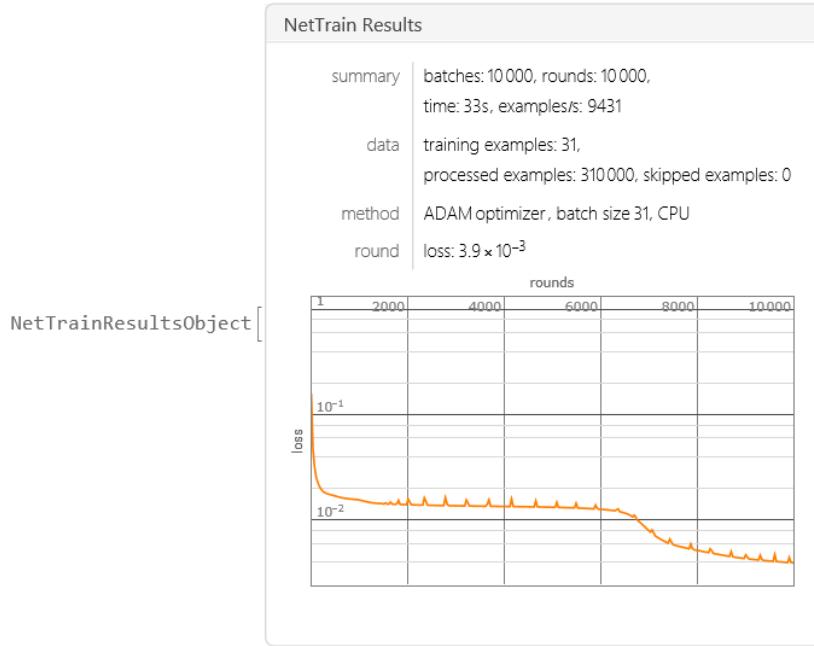
(* Train the neural network using the noisy dataset: *)
mlpTrainingResults=NetTrain[mlp,noisyData,All]

mlpTrainingResults["LossPlot"]
mlpTrainingResults["MeanBatchesPerSecond"]
mlpTrainingResults["OptimizationMethod"]
mlpTrainingResults["TrainedNet"]
mlpTrainingResults["TrainingNet"]
```

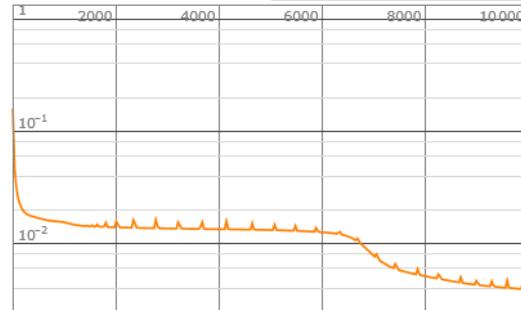
Output



Output



Output



Output

304.241

Output

```
{ADAM,Beta1->0.9,Beta2->0.999,Epsilon->1/100000,GradientClipping->None,L2Regularization->None,LearningRate->Automatic,LearningRateSchedule->None,WeightClipping->None}
```

Output

NetChain[

Input	real	
1	LinearLayer	vector (size: 150)
2	Tanh	vector (size: 150)
3	LinearLayer	vector (size: 150)
4	Tanh	vector (size: 150)
5	LinearLayer	vector (size: 1)
Output	scalar	

Output

NetGraph[

Input Ports	
Input:	real
Output:	scalar
Output Port	
Output:	real

Mathematica Code 5.9

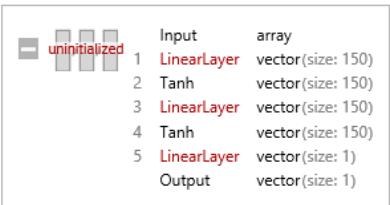
Input (* The code generates a synthetic dataset with added noise based on a Gaussian distribution. A neural network architecture with two hidden layers, each comprising 150 units with Tanh activation functions, is then defined. The code demonstrates three different methods for reporting training progress of the neural network using the noisy dataset: panel training progress reporting, progress indicator training progress reporting, and print training progress reporting. These methods offer various ways to monitor the training progress interactively, visually, or through printed updates: *)

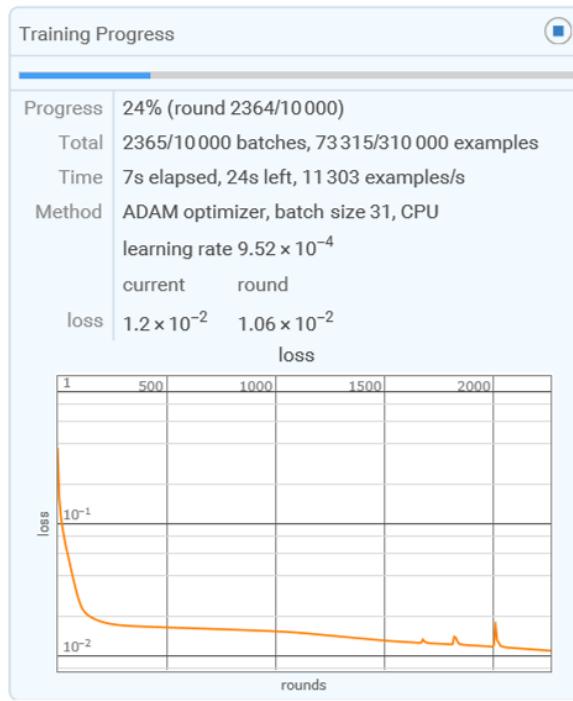
```
(* Generate a dataset with noise based on a Gaussian distribution: *)
noisyData=Table[x->Exp[-x^2]+RandomVariate[NormalDistribution[0,.15]],{x,-3,3,.2}];

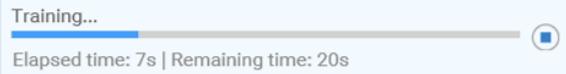
(* Define a neural network architecture with two hidden layers of 150 units each
and Tanh activation functions: *)
mlp=NetChain[{150,Tanh,150,Tanh,1}]

(* Train the neural network using the noisy dataset: *)
PanelTrainingProgressReporting=NetTrain[mlp,noisyData,TrainingProgressReporting-
>"Panel"];
ProgressIndicatorTrainingProgressReporting=NetTrain[mlp,noisyData,TrainingProgressR
eporting->"ProgressIndicator"];
PrintTrainingProgressReporting=NetTrain[mlp,noisyData,TrainingProgressReporting-
>"Print"];
```

Output

```
NetChain[]
```

Output

Output

```
Output Starting training.
Optimization Method: ADAM
Beta1: 9.00**^-1
Beta2: 9.99**^-1
Epsilon: 1.00**^-5
Gradient Clipping: -
L2 Regularization: -
Learning Rate: Automatic
Learning Rate Schedule: -
Weight Clipping: -
Device: CPU
Batch Size: 31
Batches Per Round: 1
Training Examples: 31
```

% current loss	round /10000	batch /1	examples processed	inputs /second	learning rate	time elapsed	time left
8	756	1	23436	12386	7.28**^-4	2s	24s
3.51**^-2	2.10**^-2						
15	1502	1	46562	11881	8.82**^-4	4s	23s
1.74**^-2	1.53**^-2						
23	2298	1	71238	14275	9.49**^-4	6s	20s
1.36**^-2	1.18**^-2						
31	3058	1	94798	10390	9.76**^-4	8s	18s
1.10**^-2	9.46**^-3						
38	3818	1	118358	12445	9.89**^-4	10s	16s
9.22**^-3	8.27**^-3						
46	4600	1	142600	11753	9.95**^-4	12s	14s
8.24**^-3	7.63**^-3						
53	5342	1	165602	12468	9.98**^-4	14s	12s
7.59**^-3	7.21**^-3						
61	6099	1	189069	11298	9.99**^-4	16s	10s
7.30**^-3	6.86**^-3						
69	6867	1	212877	13095	9.99**^-4	18s	8s
6.96**^-3	6.81**^-3						
76	7649	1	237119	13891	10.0**^-4	20s	6s
6.61**^-3	6.60**^-3						
84	8412	1	260772	13668	10.0**^-4	22s	4s
6.54**^-3	6.27**^-3						
92	9181	1	284611	11233	10.0**^-4	24s	2s
6.41**^-3	6.52**^-3						
100	9960	1	308760	12070	10.0**^-4	26s	0s
6.06**^-3	5.76**^-3						

Mathematica Code 5.10**Input**

(* The code demonstrates the process of training a neural network to solve a least-squares problem and monitoring its progress visually. Initially, synthetic data is generated based on a predefined function, and a neural network architecture is defined with specific layers and activation functions. Rather than using the default progress panel, the code dynamically updates and displays a plot illustrating the behavior of the network during training, overlaying the original data with the network's output in red. Training progress is monitored using this custom plot, providing real-time feedback on the network's performance. After a specified

```
duration of training, the trained network is plotted to visualize its performance
on the data: *)

(* Generate synthetic data based on a predefined function: *)
trainingData=Table[
  x->Cos[10 x]*Exp[-x^4]+x,
  {x,-3,3,.02}
];

(* Define function to visualize data: *)
visualizeData[points_,color_]:=ListLinePlot[
  points,
  PlotStyle->color,
  FrameTicks->False,
  PlotRangePadding->0.1,
  Frame->True,
  PlotRange->All,
  ImageSize->250
]

(* Plot original data: *)
originalPlot=visualizeData[trainingData[[All,2]],Blue]

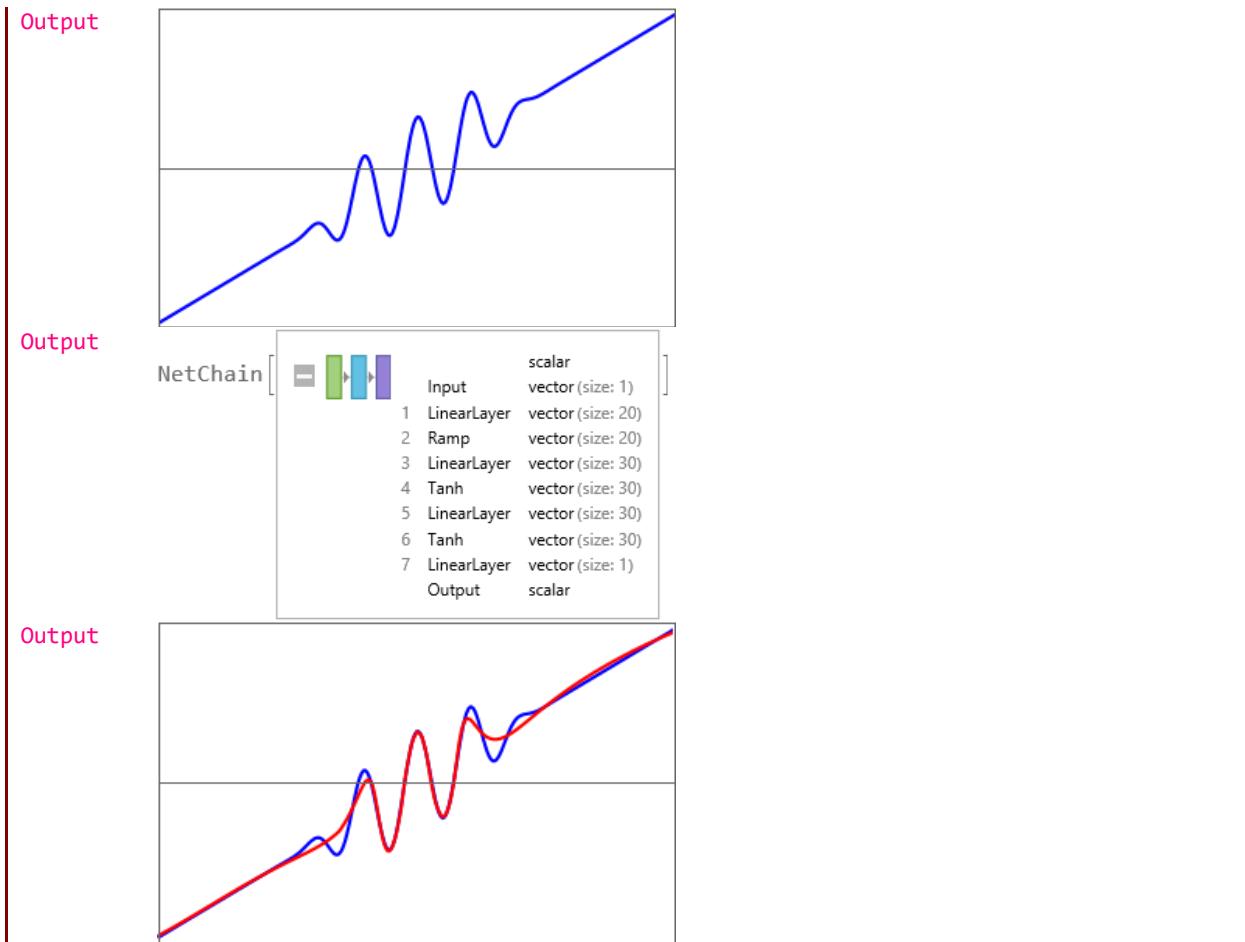
(* Define the neural network architecture: *)
net=NetChain[
  {20,Ramp,30,Tanh,30,Tanh,1},
  "Input"->"Scalar",
  "Output"->"Scalar"
];

(* Replace the default progress panel with a dynamically updated plot of the current
behavior of the net: *)

(* Function to dynamically plot the network's behavior during training: *)
plotTrainingProgress[net_,t_]:=Show[
  originalPlot,
  visualizeData[net@Range[-3,3,.02],Red],
  Epilog->Text[t,{10,0.9},{-1,0}],
  ImageSize->250
]

(* Train the network with progress monitoring using custom plot: *)
trainedNet=NetTrain[
  net,
  trainingData,
  (* Limit training time: *)
  MaxTrainingRounds->Quantity[10,"Seconds"],
  (* Custom progress reporting *)
  TrainingProgressReporting->{
    (* Update plot during training: *)
    plotTrainingProgress[#Net,#AbsoluteBatch]&,
    (* Update plot every 0.1 seconds: *)
    "Interval"->0.1
  }
]

(* Plot the final trained network after 10 seconds of training: *)
plotTrainingProgress[trainedNet,""]
```



Unit 5.3

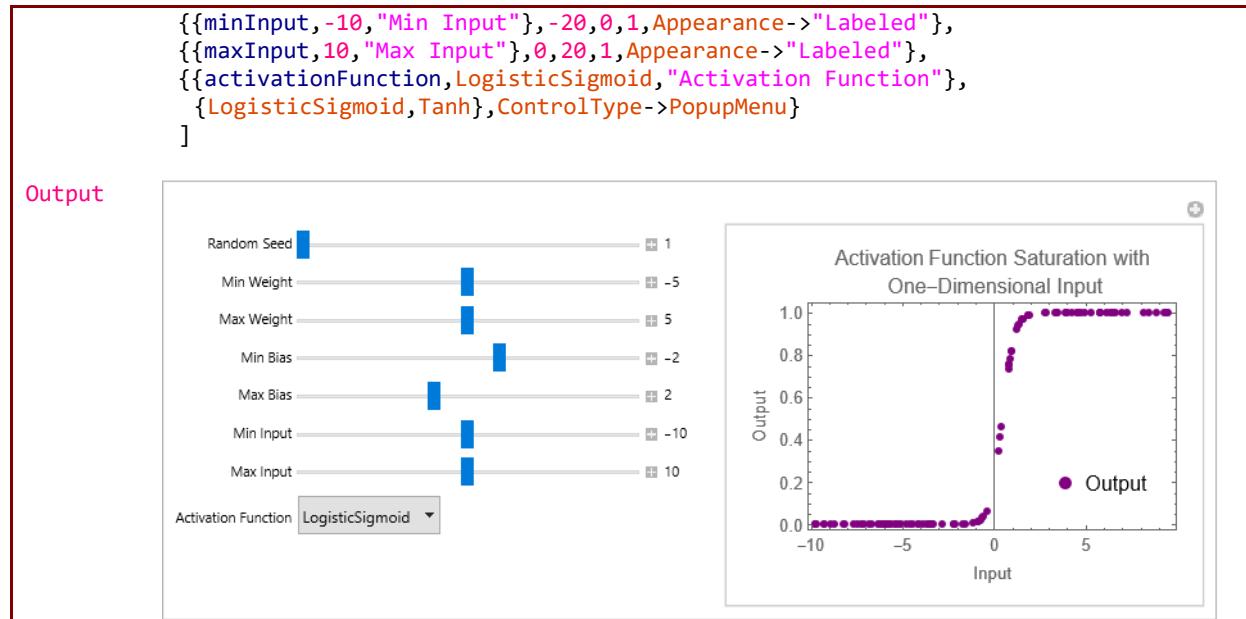
Monitor Saturation and Vanishing Gradients During Training

Mathematica Code 5.11

```

Input      (* The code generates a neural network configuration with randomized weights and
           biases within specified boundaries using Mathematica's NetChain. It then generates
           random input data within user-defined boundaries and applies the neural network to
           produce corresponding outputs, utilizing either a logistic sigmoid or hyperbolic
           tangent activation function chosen by the user. The results are visualized through
           a plot depicting the input data and neural network outputs, aiding in understanding
           the impact of activation function saturation on the data. Through interactive
           controls, users can adjust parameters such as random seed, weight and bias
           boundaries, input data boundaries, and activation function, facilitating
           exploration of different scenarios and their effects on the neural network's
           behavior: *)

Manipulate[
  (* Set random seed for reproducibility: *)
  SeedRandom[randomSeed];
  (* Generate Random Weight and Bias within specified boundaries: *)
  weight=RandomReal[{minWeight,maxWeight}];
  bias=RandomReal[{minBias,maxBias}];
  (* Define Neural Network using NetChain: *)
  neuralNet=NetChain[
    {
      LinearLayer[1,"Weights"→weight,"Biases"→bias],
      (* Add activation function to the neural network: *)
      ElementwiseLayer[activationFunction]
    },
    "Input"→1
  ];
  (* Generate Random Input Data within specified boundaries: *)
  inputData=RandomReal[{minInput,maxInput},{100,1}];
  (* Apply Neural Network to Inputs: *)
  outputData=neuralNet[inputData];
  (* Plot Input Data and Outputs: *)
  ListPlot[
    Transpose[
      {Flatten[inputData],Flatten[outputData]}],
    PlotStyle→{PointSize[0.02],Purple},
    PlotRange→All,
    Frame→True,
    FrameLabel→{"Input","Output"},
    PlotLegends→Placed[{"Output"},{0.8,0.2}],
    PlotLabel→"Activation Function Saturation with\nOne-Dimensional Input",
    ImageSize→250
  ],
  (* Controls: *)
  {{randomSeed,1,"Random Seed"},1,1000,1,Appearance→"Labeled"},
  {{minWeight,-5,"Min Weight"},-10,0,0.1,Appearance→"Labeled"},
  {{maxWeight,5,"Max Weight"},0,10,0.1,Appearance→"Labeled"},
  {{minBias,-2,"Min Bias"},-5,0,0.1,Appearance→"Labeled"},
  {{maxBias,2,"Max Bias"},0,5,0.1,Appearance→"Labeled"}]
```

**Mathematica Code 5.12**

Input (* The code generates synthetic input-output data and constructs a neural network with multiple layers, each employing logistic sigmoid activation functions. It then trains the network using the generated data, tracking activation output statistics during training. The code visualizes the mean activation values with error bars for each hidden layer over training epochs, providing insights into activation behavior and potential saturation. Additionally, histograms are created to visualize the distribution of activation values for each layer, aiding in understanding activation characteristics: *)

```
(* Generate synthetic input-output data: *)

(* Define a mixture distribution with three components: *)
distribution=MixtureDistribution[{1,2,3}, {NormalDistribution[1,1],
NormalDistribution[7,1], NormalDistribution[10,1]}];

(* Generate synthetic input data: *)
inputData=RandomVariate[distribution,{2000,200}];

(*Generate synthetic output data:*)
outputData=RandomReal[2,{2000,1}];

(* Plot the probability density function (PDF) of the distribution: *)
Plot[
 Evaluate[PDF[distribution,x]],
 {x,-5,15},
 PlotRange->All,
 Filling->Axis,
 PlotStyle->Purple,
 ImageSize->250,
 AxesLabel->{None,"PDF"}
]

(* Define the neural network architecture: *)
neuralNet=NetGraph[
 {
 (* Two neurons with logistic sigmoid activation for the first layer: *)
 }
```

```

2,LogisticSigmoid,
(* Two neurons with logistic sigmoid activation for the second layer: *)
2,LogisticSigmoid,
(* Two neurons with logistic sigmoid activation for the third layer: *)
2,LogisticSigmoid,
(* One neuron with logistic sigmoid activation for the output layer: *)
1,LogisticSigmoid
},
{1->2->3->4->5->6->7->8},
(* Define input size: *)
"Input"->200
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
neuralNet,
(*Specify random initialization method*)
Method->{"Random","Weights"->0.1,"Biases"->0.1},
(*Ensure reproducibility by setting random seed*)
RandomSeeding->Automatic
]

(*Train the network*)
trainingResults=NetTrain[
initializedNet,
(* Provide input-output pairs for training: *)
<|"Input"->inputData,"Output"->outputData|>,
All,
(* Limit the number of training rounds: *)
MaxTrainingRounds->100,
TrainingProgressMeasurements->{
    (* Track activation output statistics during training: *)
    <|"Measurement"->NetPort[{2,"Output"}],"Interval"->"Round"|>,
    <|"Measurement"->NetPort[{4,"Output"}],"Interval"->"Round"|>,
    <|"Measurement"->NetPort[{6,"Output"}],"Interval"->"Round"|>,
    <|"Measurement"->NetPort[{8,"Output"}],"Interval"->"Round"|>
}
];
(* Extract activation output data for each layer: *)
trainingData=trainingResults["RoundMeasurementsLists"];

activationOutputLayer2=Values[trainingData][[2]];
activationOutputLayer4=Values[trainingData][[3]];
activationOutputLayer6=Values[trainingData][[4]];
activationOutputLayer8=Values[trainingData][[5]];

(* Calculate mean and standard deviation for each layer's activation outputs: *)
meanLayer2=Mean/@activationOutputLayer2;
stdDevLayer2=StandardDeviation/@activationOutputLayer2;
meanWithStdDevLayer2=Around@@@Transpose[{meanLayer2,stdDevLayer2}];

meanLayer4=Mean/@activationOutputLayer4;
stdDevLayer4=StandardDeviation/@activationOutputLayer4;
meanWithStdDevLayer4=Around@@@Transpose[{meanLayer4,stdDevLayer4}];

meanLayer6=Mean/@activationOutputLayer6;
stdDevLayer6=StandardDeviation/@activationOutputLayer6;
meanWithStdDevLayer6=Around@@@Transpose[{meanLayer6,stdDevLayer6}];

```

```

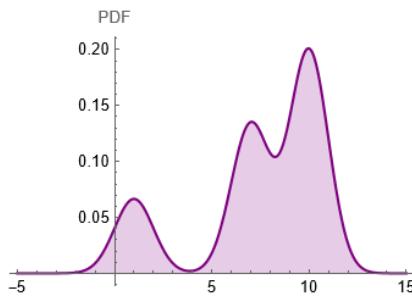
meanLayer8=Mean/@activationOutputLayer8;

(* Visualize mean activation values with error bars over training epochs: *)
ListLinePlot[
{meanWithStdDevLayer2,meanWithStdDevLayer4,meanWithStdDevLayer6,meanLayer8},
PlotStyle->{Opacity[0.6],Opacity[0.6],Opacity[0.6],Opacity[0.6]},
PlotRange->{0,1},
PlotLegends->{
  "Mean of Activations Layer 2",
  "Mean of Activations Layer 4",
  "Mean of Activations Layer 6",
  "Mean of Activations Layer 8"
},
IntervalMarkers->"Bars",
Frame->True,
FrameLabel->{"Epochs","Mean of Activation Values"},
PlotLabel->"Means with Standard Deviations (vertical bars)\n of the activation
values"
]

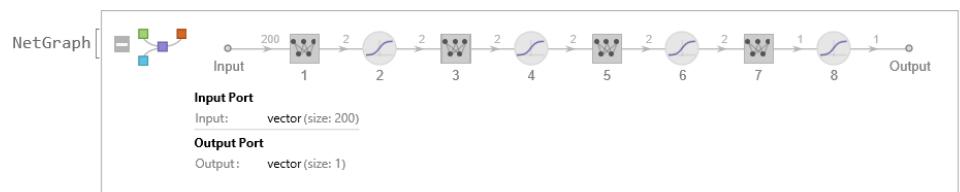
(* Create histograms to visualize the distribution of activation values for each
layer: *)
Table[
Histogram[
Flatten[layer[[3]]],
Automatic,
"Count",
PlotLabel->Style[Row[{ "Histogram of ",layer[[2]]}]],
AxesLabel->{layer[[1]],"Probability"},
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->300
],
{layer,{ "postActivLayer2", "Post-Activation Layer 2",activationOutputLayer2},
 {"postActivLayer4", "Post-Activation Layer 4",activationOutputLayer4},
 {"postActivLayer6", "Post-Activation Layer 6",activationOutputLayer6},
 {"postActivLayer8", "Post-Activation Layer 8",activationOutputLayer8}
 }
]
]

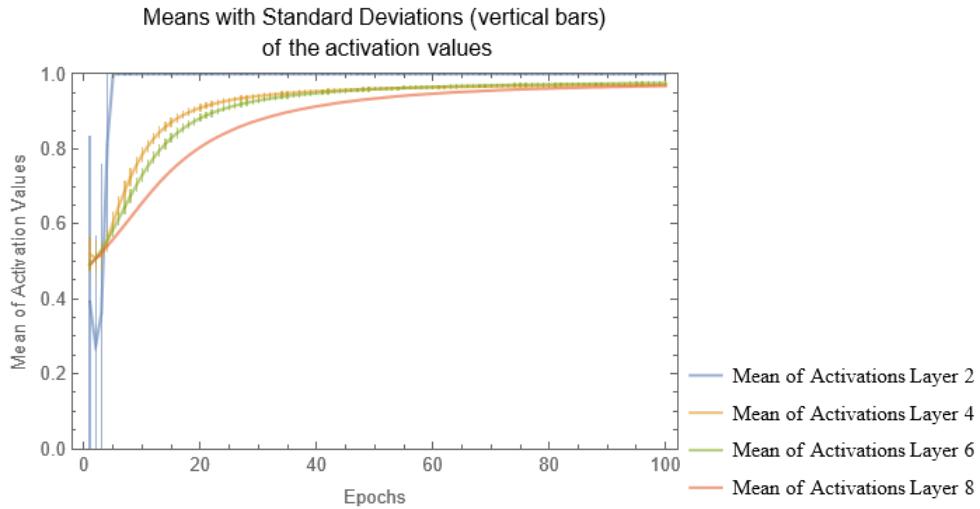
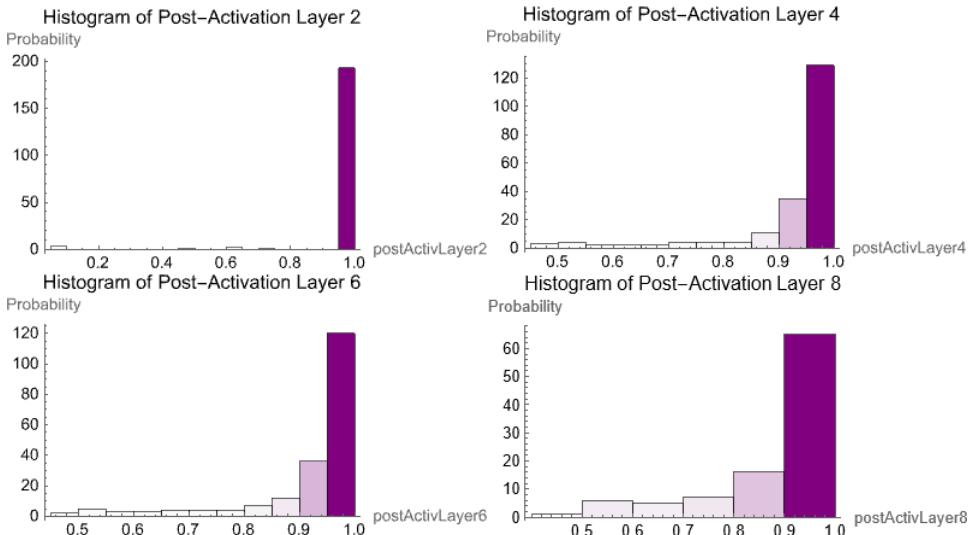
```

Output



Output



Output**Output****Mathematica Code 5.13****Input**

(* The code aims to train a neural network to learn a mapping from two-dimensional input points to their respective exponential distances from the origin. It first generates a dataset comprising pairs of input points and their corresponding exponential distances. Then, it defines a neural network architecture with multiple layers, consisting of logistic sigmoid activation functions. The network is initialized with random weights and biases for reproducibility. Subsequently, the NetTrain function is utilized to train the network on the dataset. Training progress is monitored using the "GradientsRMS" property, and an evolution plot is generated to visualize the progress of gradients over training batches. The training process is limited to 10 rounds with a batch size of 1024. The "GradientsRMS" property in NetTrain helps track the magnitudes of gradients for each layer during training, providing insights into the occurrence of vanishing gradients and their impact on training performance. Significantly smaller RMS values of gradients in earlier layers compared to deeper layers indicate the presence of the vanishing gradient problem. This information can guide adjustments to the network architecture or training parameters to mitigate the issue: *)

(* Generate dataset: pairs of input points and their corresponding exponential distances: *)

```

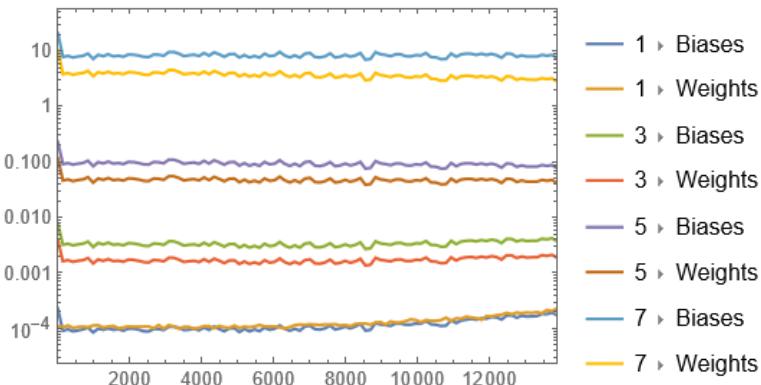
trainingData=Flatten@Table[{x,y}=>{Exp[-Norm[{x,y}]],{x,-3,3,.005},{y,-3,3,.005}];

(* Define neural network architecture: *)
neuralNet=NetChain[
{
  10,LogisticSigmoid,
  5,LogisticSigmoid,
  5,LogisticSigmoid,
  1
},
"Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
  neuralNet,
  (* Specify random initialization method: *)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Train the initialized network:*)
NetTrain[
  initializedNet,
  trainingData,
  (* Monitor training progress using GradientsRMS property: *)
  <|
    "Property"->"GradientsRMS",
    "Form"->"EvolutionPlot",
    "PlotOptions"->{ ScalingFunctions->"Log",ImageSize->300},
    "Interval"->"Batch"
  |>,
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Use a batch size of 1024: *)
  BatchSize->1024
]

```

Output**Mathematica Code 5.14**

Input (* We can update the previous version of the code by changing the parameter 'Interval' to 'Rounds' and removing the 'ScalingFunctions->"Log"' option from the 'PlotOptions': *)

```

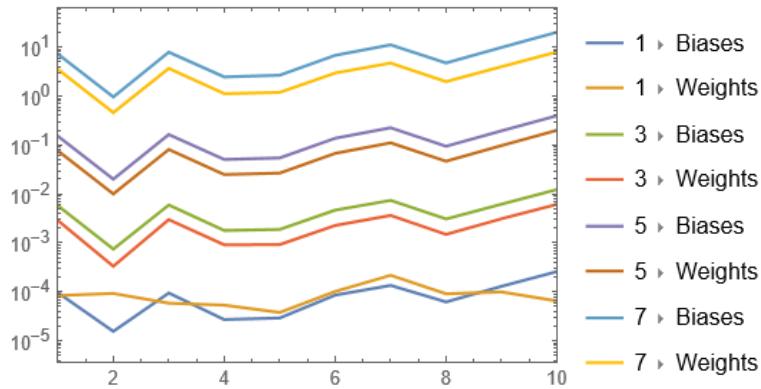
(* Generate dataset: pairs of input points and their corresponding exponential
distances: *)
trainingData=Flatten@Table[{x,y}=>{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-
3,3,.005}];

(* Define neural network architecture: *)
neuralNet=NetChain[
 {
  10,LogisticSigmoid,
  5,LogisticSigmoid,
  5,LogisticSigmoid,
  1
 },
 "Input"=>2
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
  neuralNet,
  (* Specify random initialization method:*)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Train the initialized network: *)
NetTrain[
  initializedNet,
  trainingData,
  (* Monitor training progress using GradientsRMS property: *)
  <|
    "Property"->"GradientsRMS",
    "Form"->"EvolutionPlot",
    "PlotOptions"->{ ImageSize->300},
    "Interval"->"Rounds"
  |>,
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Use a batch size of 1024: *)
  BatchSize->1024
]

```

Output**Mathematica Code 5.15**

Input (* The code aims to train a neural network to learn the relationship between two-dimensional input points and their corresponding exponential distances from the

origin. It achieves this by generating a dataset containing these pairs and defining a neural network architecture with multiple layers and logistic sigmoid activation functions. The network is then initialized with random weights and biases before being trained on the dataset. During training, the evolution of the root mean square (RMS) values of the weights for each layer is monitored to visualize the training progress and diagnose potential issues such as vanishing gradients. The training process is limited to 10 rounds with a batch size of 1024.

Monitoring the RMS values of the weights for each layer of the neural network provides insights into how the weights are being adjusted during training. If the RMS values of the weights do not change significantly over time (i.e., they remain relatively constant or change only minimally), it suggests that the gradients in those layers are small. This indicates that the network is experiencing difficulty in learning meaningful representations or updating the parameters effectively in those layers. Small changes in the RMS values of the weights imply that the gradients are not providing sufficiently strong signals to guide meaningful updates to the weights. This situation often indicates the presence of vanishing gradients: *)

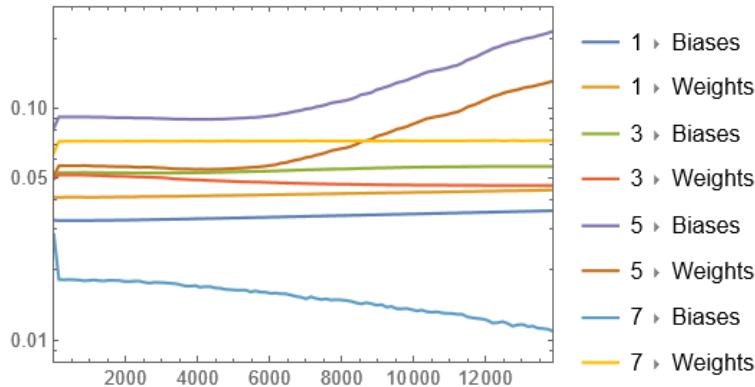
```
(* Generate dataset: pairs of input points and their corresponding exponential
distances: *)
trainingData=Flatten@Table[{{x,y}->{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-
3,3,.005}}];

(* Define neural network architecture: *)
neuralNet=NetChain[
  {
    10,LogisticSigmoid,
    5,LogisticSigmoid,
    5,LogisticSigmoid,
    1
  },
  "Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
  neuralNet,
  (* Specify random initialization method: *)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Train the initialized network: *)
NetTrain[
  initializedNet,
  trainingData,
  (* Monitor training progress using WeightsRMS property: *)
  <|
    "Property"->"WeightsRMS",
    "Form"->"EvolutionPlot",
    "PlotOptions"->{ ScalingFunctions->"Log",ImageSize->300},
    "Interval"->"Batch"
  |>,
  MaxTrainingRounds->10,
  BatchSize->1024
]
```

Output

**Mathematica Code 5.16**

```

Input (* The code aims to analyze the behavior of gradients during the training of a
       neural network designed to learn the mapping from two-dimensional input points to
       their corresponding exponential distances from the origin. It first generates a
       dataset comprising pairs of input points and their exponential distances, then
       defines a neural network architecture with multiple layers, including logistic
       sigmoid activation functions. The code trains the network on the dataset while
       monitoring the gradients of network parameters, particularly focusing on the first
       and last layers, to assess their distributions and changes over training epochs.
       Visualization and statistical analysis of gradient distributions provide insights
       into the training dynamics, including potential issues such as vanishing or
       exploding gradients: *)

(* Generate dataset:pairs of input points and their corresponding exponential
   distances: *)
trainingData=Flatten@Table[
  {x,y}~->{Exp[-Norm[{x,y}]]},
  {x,-3,3,.005},
  {y,-3,3,.005}
];

(* Define neural network architecture: *)
neuralNet=NetChain[
 {
  10,LogisticSigmoid,
  5,LogisticSigmoid,
  5,LogisticSigmoid,
  1
 },
 "Input"~->2
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
 neuralNet,
 (* Specify random initialization method: *)
 Method~->>{"Random","Weights"~->0.05,"Biases"~->0.05},
 (* Ensure reproducibility by setting random seed: *)
 RandomSeeding~->Automatic
];

(* Train the initialized network and monitor gradients: *)
gradients=NetTrain[
 initializedNet,
 trainingData,

```

```

<|
  (* Monitor gradients: *)
  "Property"->"Gradients",
  (* Monitor gradients over rounds: *)
  "Interval"->"Rounds"
|>,
MaxTrainingRounds->10,
BatchSize->1024
];

(* Extract and analyze gradients of the first and last layers: *)
(* Generate data for the first layer gradients for 10 epochs: *)
firstLayerGradientsData=Table[
  Flatten[
    Table[
      Values[gradients][[i]][[2]],
      {i,1,10}][[j]]
    ],
  {j,1,10}]//Normal;

(* Generate data for the last layer gradients for 10 epochs: *)
lastLayerGradientsData=Table[
  Flatten[
    Table[
      Values[gradients][[i]][[8]],
      {i,1,10}][[j]]
    ],
  {j,1,10}]//Normal;

(* Smooth kernel density estimation for first layer gradients: *)
smoothKernelFirstLayer=SmoothKernelDistribution[#]&/@firstLayerGradientsData;

(* Smooth kernel density estimation for last layer gradients: *)
smoothKernelLastLayer=SmoothKernelDistribution[#]&/@lastLayerGradientsData;

(* Visualize the resulting densities of the first layer gradients: *)
ListLinePlot3D[
  Table[
    {({#-1},x,PDF[smoothKernelFirstLayer[[#]],x}),
     {x,-0.001,0.001,0.0001}
    ]&@{1,2,3,4,5,6,7,8,9,10},
  Filling->Bottom,
  FillingStyle->Opacity[0.75],
  Joined->True,
  BoxRatios->{1,1,1},
  PlotRange->All,
  ImageSize->250,
  AxesLabel->{"Epoch","Gradient Value","Density"},
  PlotLabel->"Densities of the First Layer Gradients"
]

(* Visualize the resulting densities of the last layer gradients: *)
ListLinePlot3D[
  Table[
    {({#-1},x,PDF[smoothKernelLastLayer[[#]],x}),
     {x,-10,10,0.1}
    ]&@{1,2,3,4,5,6,7,8,9,10},
  Filling->Bottom,
  FillingStyle->Opacity[0.75],
  Joined->True,
]

```

```
BoxRatios->{1,1,1},
PlotRange->All,
ImageSize->250,
AxesLabel->{"Epoch", "Gradient Value", "Density"},
PlotLabel->"Densities of the Last Layer Gradients"
]

(* Plot the gradient distribution with respect to weights of the first layer for
10 epochs: *)
firstLayerWeightsGradientData=Table[
  Flatten[
    Table[
      Values[gradients][[i]][[[2]],
      {i,1,10}][[j]],
    ],
    {j,1,10}]//Normal;

DistributionChart[
  firstLayerWeightsGradientData,
  Joined->"Mean",
  ChartElementFunction->"PointDensity",
  ChartStyle->"Pastel",
  ImageSize->300,
  FrameLabel->{"Epochs of First Layer", "Gradient Values"},
  PlotLabel->"Gradient Distribution with Respect \n to Weights of the First Layer
for 10 Epochs"
]

(* Plot the gradient distribution with respect to weights of the last layer for 10
epochs: *)
firstLayerWeightsGradientData=Table[
  Flatten[
    Table[
      Values[gradients][[i]][[[8]],
      {i,1,10}][[j]],
    ],
    {j,1,10}]//Normal;

DistributionChart[
  firstLayerWeightsGradientData,
  Joined->"Mean",
  ChartElementFunction->"PointDensity",
  ChartStyle->"Pastel",
  ImageSize->300,
  FrameLabel->{"Epochs of Last Layer", "Gradient Values"},
  PlotLabel->"Gradient Distribution with Respect \n to Weights of the Last Layer
for 10 Epochs"
]

(* Plot the gradient distribution with respect to weights of the 4 layers in the
first epoch: *)
firstEpochLayerWeightsGradientData=Table[
  Flatten[
    Table[
      Values[gradients][[1]][[[i]],
      {i,2,8,2}
      ][[j]]
    ],
    {j,1,4}
  ]// Normal;
```

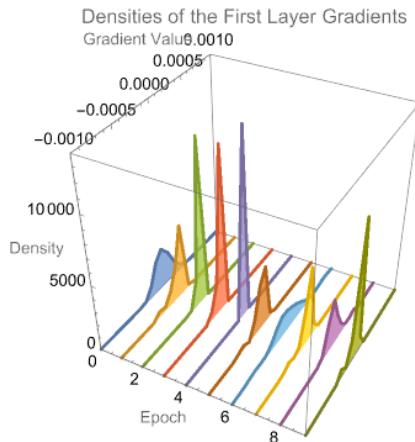
```

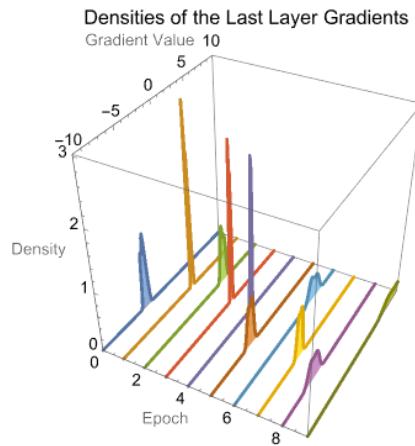
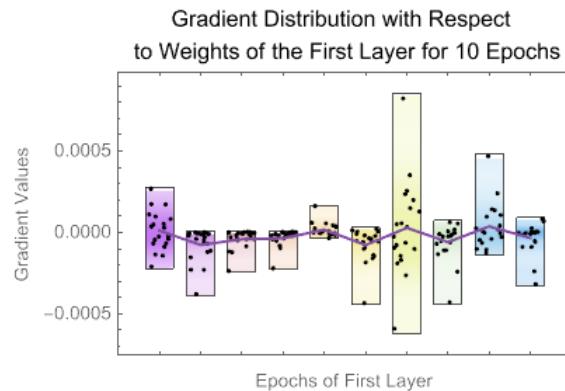
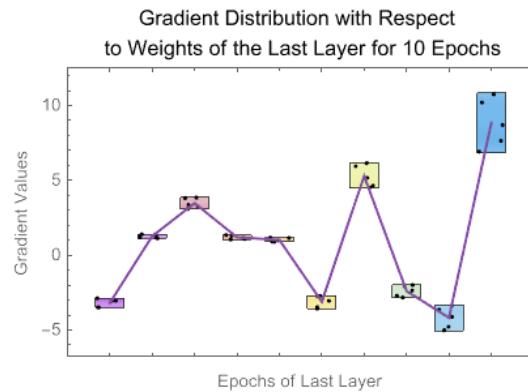
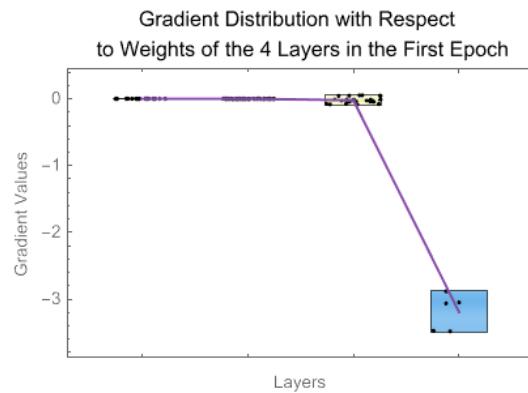
DistributionChart[
  firstEpochLayerWeightsGradientData,
  Joined->"Mean",
  ChartElementFunction->"PointDensity",
  ChartStyle->"Pastel",
  ImageSize->300,
  FrameLabel->{"Layers", "Gradient Values"},
  PlotLabel->"Gradient Distribution with Respect \n to Weights of the 4 Layers in
the First Epoch"
]

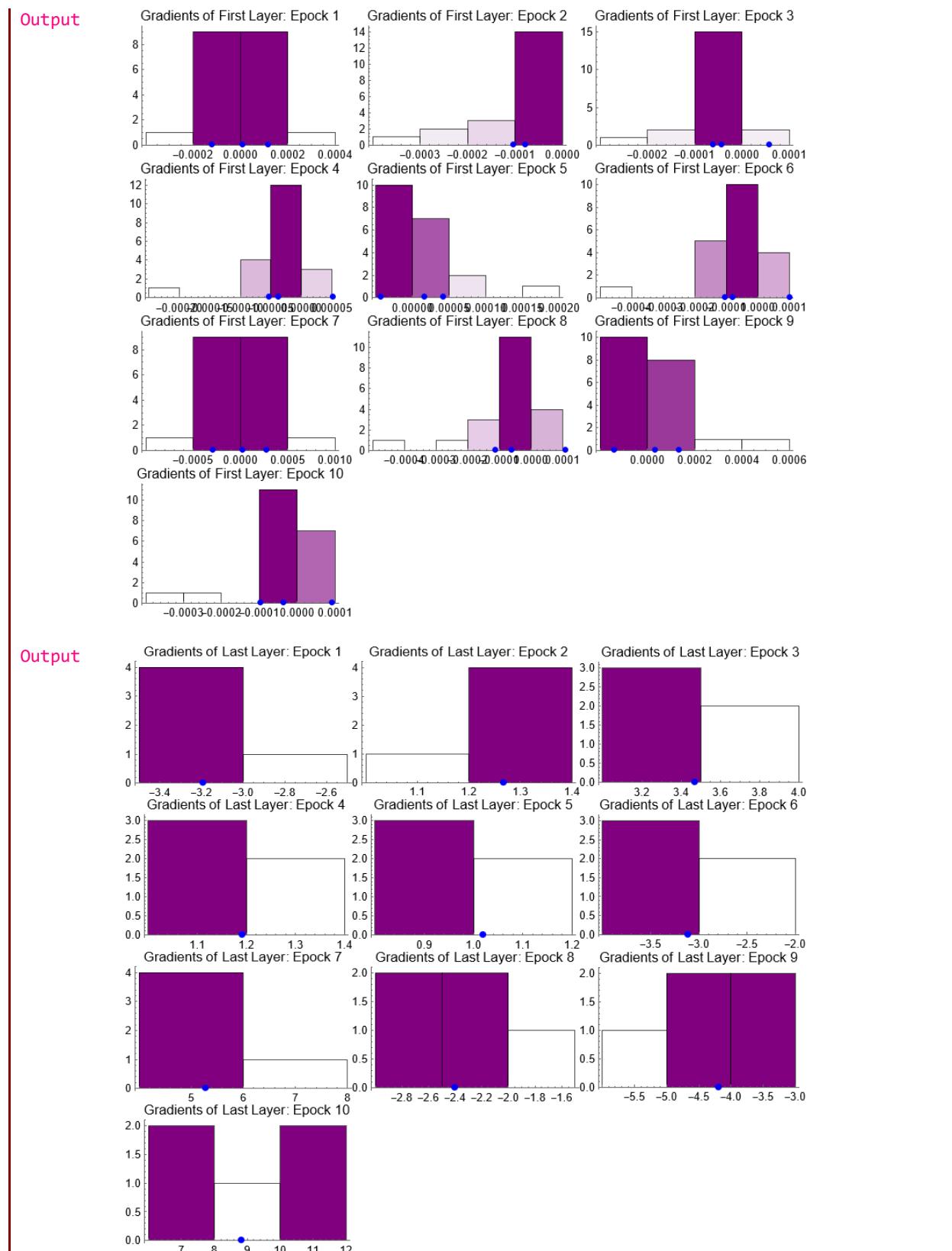
(* Histogram of the Gradient with Respect to Weights of the Fisrt and Last Layers
for 10 Epochs: *)
Table[
  Table[
    Table[
      Histogram[
        data=Flatten[
          Table[
            Values[gradients][[i]][[k[[2]]]],
            {i,1,10}
            ][[j]]
          ]// Normal,
        Automatic,
        Epilog->{
          Blue,
          PointSize[0.03],
          Point[{
            {StandardDeviation[data],0},
            {Mean[data],0},
            {-StandardDeviation[data],0}
            }]
          },
        ColorFunction->Function[{height},Opacity[height]],
        ChartStyle->Purple,
        ImageSize->200,
        PlotRange->Full,
        PlotLabel->Style[Row[{"Gradients of ",k[[1]],": Epoch ",j}]]
        ],
        {j,1,10}
        ],
      {k,{{"First Layer",2}, {"Last Layer",8}}}
    ]
  ]
]

```

Output



Output**Output****Output****Output**



Unit 5.4

NetInitialize (Xavier and Kaiming)

Mathematica Code 5.17

```

Input      (* This code defines a function xavierInitialize which takes the number of input
           neurons (nIn) and the number of output neurons (nOut) as arguments. It then computes
           the scaling factor according to the Xavier initialization formula and generates
           random weights using that scale. Finally, it returns the initialized weights as a
           matrix with dimensions (nOut,nIn). The code will generate both a plot showing the
           distribution of weights and a histogram displaying the frequency of different weight
           values. The RandomVariate function is used to generate weights from a uniform
           distribution with min=-scale and max=scale: *)

xavierInitialize[nIn_,nOut_]:=Module[
  {scale},
  scale=Sqrt[6/(nIn+nOut)];
  RandomVariate[UniformDistribution[{-scale,scale}],{nOut,nIn}]
]

(*Example usage*)
nIn=20; (*Number of input neurons*)
nOut=30; (*Number of output neurons*)

weights=xavierInitialize[nIn,nOut];

(*Plot of weight distribution*)
ListPlot[
  Flatten[weights],
  PlotRange->All,
  Filling->Axis,
  PlotStyle->{PointSize[Tiny],Purple},
  Frame->True,
  ImageSize->250,
  FrameLabel->{"Index","Weight"},
  PlotLabel->"Xavier Initialized Weights Distribution \n(Uniform Distribution)"
]

(* Histogram of weight distribution*)
Histogram[
  Flatten[weights],
  20,
  PlotRange->All,
  Frame->True,
  ColorFunction->Function[{height},Opacity[height]],
  ChartStyle->Purple,
  ImageSize->250,
  FrameLabel->{"Weight","Frequency"},
  PlotLabel->"Xavier Initialized Weights Histogram \n(Uniform Distribution)"
]

(* Plot the weight distributions with respect to weights of layer with nIn=20 and
nOut=30: *)
DistributionChart[
  weights,

```

```

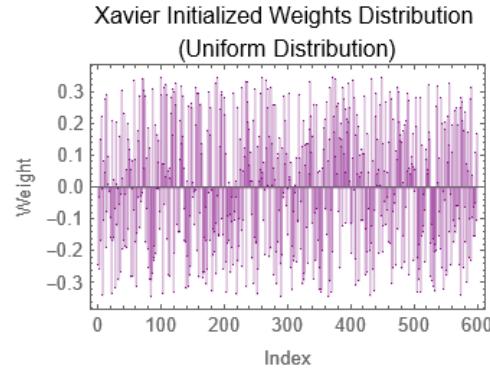
Joined->"Mean",
ChartElementFunction->"PointDensity",
ChartStyle->"Pastel",
ImageSize->300,
FrameLabel->{"Neurons", "Weights"},
PlotLabel->"Weight Distributions using Xavier initialized Weights \n of the 30
Neurons"
]

(* Smooth kernel density estimation for 30 neurons weights : *)
smoothKernelLastLayer=SmoothKernelDistribution[ #]&/@weights;

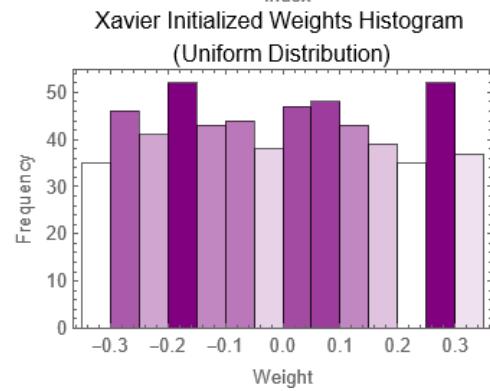
(* Visualize the resulting densities: *)
ListLinePlot3D[
Table[
{({#-1}),x,PDF[smoothKernelLastLayer[[#]],x]},
{x,-0.5,0.5,.01}
]&/@Range[1,30],
Filling->Bottom,
Joined->True,
BoxRatios->{1,1,1},
PlotRange->All,
ImageSize->300,
AxesLabel->{"Neurons", "Weights", "Density"},
PlotLabel->"Weight Distributions using Xavier initialized \n Weights of the 30
Neurons"
]

```

Output

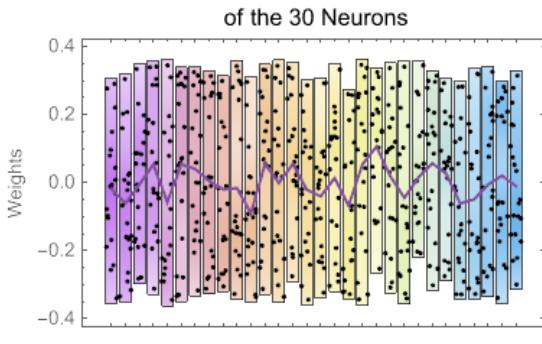


Output



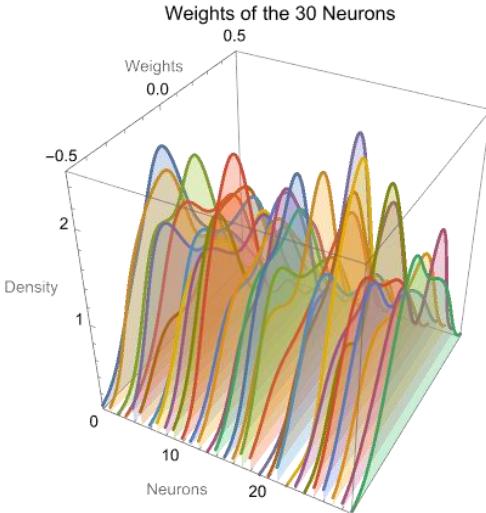
Output

Weight Distributions using Xavier initialized Weights



Output

Weight Distributions using Xavier initialized

**Mathematica Code 5.18**

```

Input      (* In this version, the RandomVariate function is used to generate weights from a
           normal distribution with mean 0 and standard deviation calculated according to the
           Xavier initialization formula. The rest of the code remains the same as before,
           generating both a plot and a histogram of the weight distribution: *)
xavierInitialize[nIn_,nOut_]:=Module[
  {scale},
  scale=Sqrt[1/(nIn)];
  RandomVariate[NormalDistribution[0,scale],{nOut,nIn}]
]

(*Example usage*)
nIn=20; (*Number of input neurons*)
nOut=30; (*Number of output neurons*)

weights=xavierInitialize[nIn,nOut];

(*Plot of weight distribution*)
ListPlot[
  Flatten[weights],
  PlotRange->All,
  Filling->Axis,
  PlotStyle->{PointSize[Tiny],Purple},
  Frame->True,
  ImageSize->250,
  FrameLabel->{"Index","Weight"},
```

```

PlotLabel->"Xavier Initialized Weights Distribution \n(Normal Distribution)"
]

(*Histogram of weight distribution*)
Histogram[
Flatten[weights],
20,
PlotRange->All,
Frame->True,
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250,
FrameLabel->{"Weight", "Frequency"},
PlotLabel->"Xavier Initialized Weights Histogram \n(Normal Distribution)"
]

(* Plot the weight distributions with respect to weights of layer with nIn=20 and
nOut=30: *)
DistributionChart[
weights,
Joined->"Mean",
ChartElementFunction->"PointDensity",
ChartStyle->"Pastel",
ImageSize->300,
FrameLabel->{"Neurons", "Weights"},
PlotLabel->"Weight Distributions using Xavier initialized Weights \n of the 30
Neurons"
]

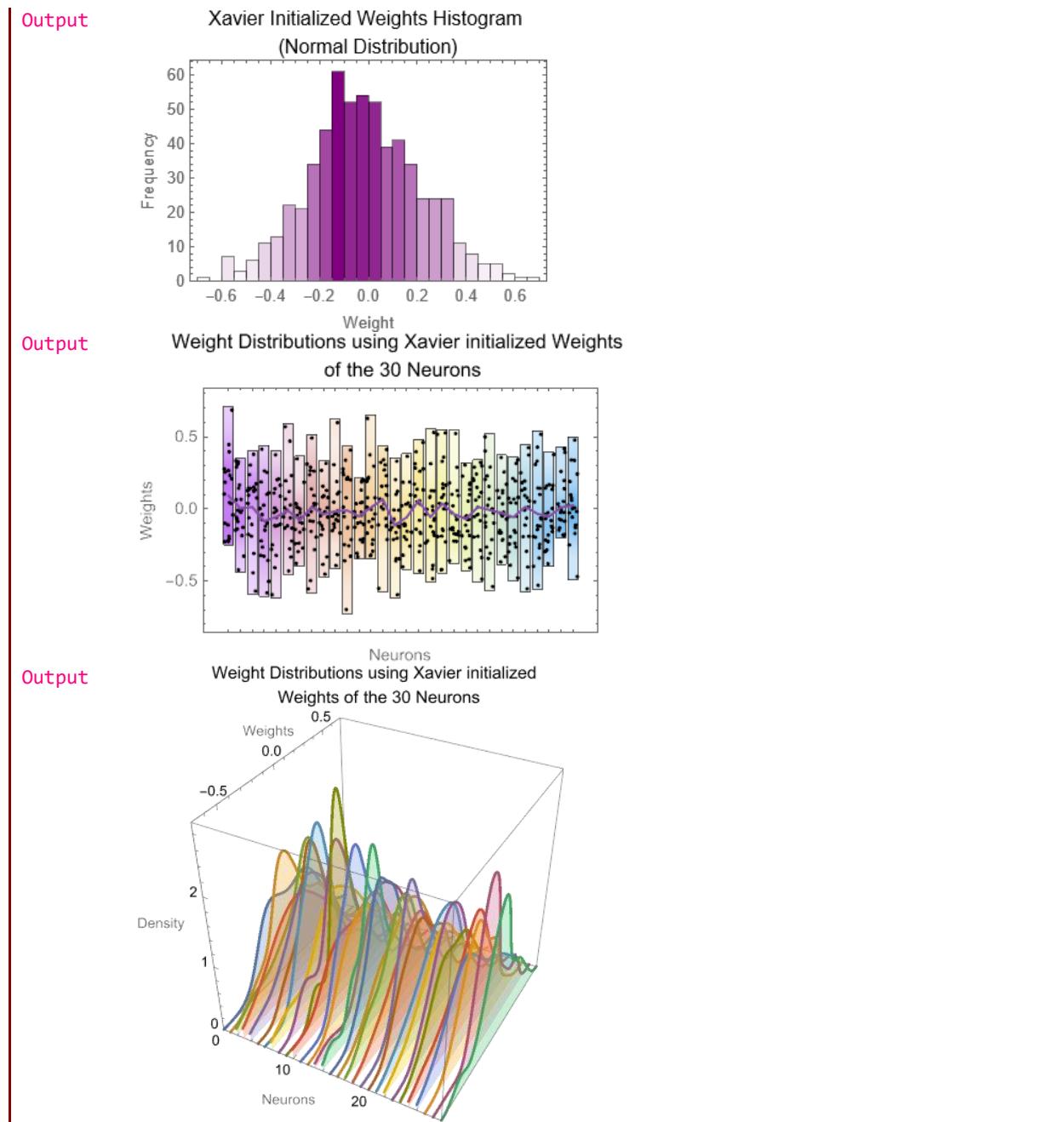
(* Smooth kernel density estimation for 30 neurons weights: *)
smoothKernelLastLayer=SmoothKernelDistribution[#]&/@weights;

(* Visualize the resulting densities: *)
ListLinePlot3D[
Table[
{(#-1),x,PDF[smoothKernelLastLayer[[#]],x]},
{x,-0.7,0.7,.01}
]&/@Range[1,30],
Filling->Bottom,
Joined->True,
BoxRatios->{1,1,1},
PlotRange->All,
ImageSize->300,
AxesLabel->{"Neurons", "Weights", "Density"},
PlotLabel->"Weight Distributions using Xavier initialized \n Weights of the 30
Neurons"
]

```

Output

Xavier Initialized Weights Distribution
(Normal Distribution)

**Mathematica Code 5.19****Input**

```
(* This code defines a function kaimingInitialize which takes the number of input
neurons (nIn), the number of output neurons (nOut), and optionally a distribution
(defaulting to a normal distribution with mean 0 and standard deviation 1) as
arguments. It then computes the scaling factor according to the Kaiming
initialization formula and generates random weights from the specified distribution,
scaled by this factor. Finally, it returns the initialized weights as a matrix with
dimensions (nOut,nIn): *)
```

```
kaimingInitialize[nIn_,nOut_,distribution_:NormalDistribution[0,1]]:=Module[
{scale},
scale=SquareRoot[2/nIn];
RandomVariate[distribution,{nOut,nIn}]*scale]
```

```

];
(*Example usage*)
nIn=20; (*Number of input neurons*)
nOut=30; (*Number of output neurons*)

weights=kaimingInitialize[nIn,nOut];

(*Plot of weight distribution*)
ListPlot[
Flatten[weights],
PlotRange->All,
Filling->Axis,
PlotStyle->{PointSize[Tiny],Purple},
ImageSize->250,
Frame->True,
FrameLabel->{"Index","Weight"},
PlotLabel->"Kaiming Initialized Weights Distribution"
]

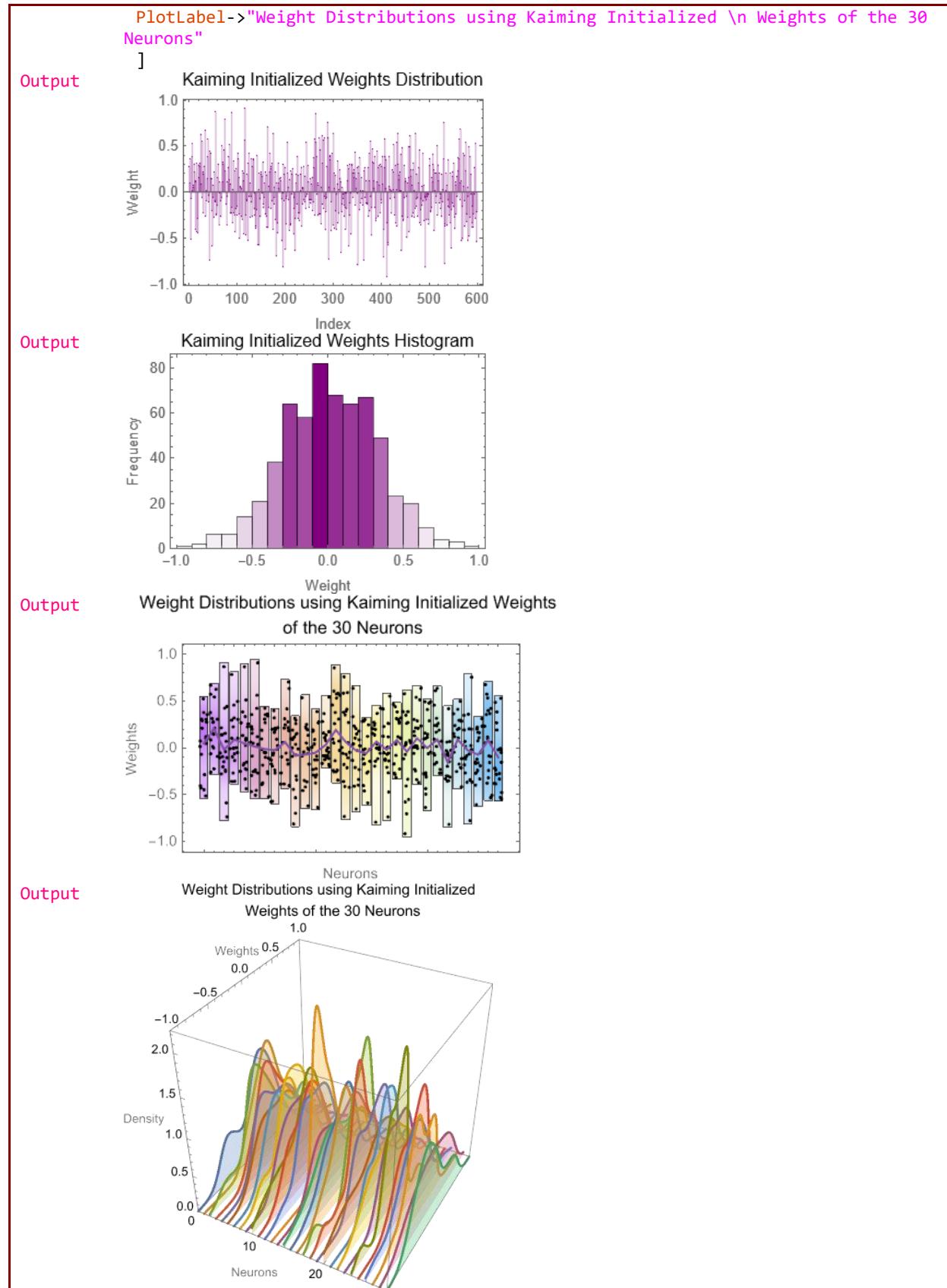
(*Histogram of weight distribution*)
Histogram[
Flatten[weights],
20,
PlotRange->All,
Frame->True,
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250,
FrameLabel->{"Weight","Frequency"},
PlotLabel->"Kaiming Initialized Weights Histogram"
]

(* Plot the weight distributions with respect to weights of layer with nIn=20 and
nOut=30: *)
DistributionChart[
weights,
Joined->"Mean",
ChartElementFunction->"PointDensity",
ChartStyle->"Pastel",
ImageSize->300,
FrameLabel->{"Neurons","Weights"},
PlotLabel->"Weight Distributions using Kaiming Initialized Weights \n of the 30
Neurons"
]

(* Smooth kernel density estimation for 30 neurons weights: *)
smoothKernelLastLayer=SmoothKernelDistribution[#]&/@weights;

(* Visualize the resulting densities: *)
ListLinePlot3D[
Table[
{(#-1),x,PDF[smoothKernelLastLayer[[#]],x]},
{x,-1,1,.01}
]&/@Range[1,30],
Filling->Bottom,
Joined->True,
BoxRatios->{1,1,1},
PlotRange->All,
ImageSize->300,
AxesLabel->{"Neurons","Weights","Density"},


```



Mathematica Code 5.20

```

Input (* The code aims to explore the effects of different weight initialization methods,
       specifically Xavier initialization with uniform and normal distributions, on the
       weights of a simple neural network. Firstly, a network structure comprising an
       input layer of 30 units followed by two fully connected layers with 100 units each
       and a ReLU activation function is defined. Then, the network weights are initialized
       using Xavier initialization with uniform and normal distributions, yielding two
       separate sets of weights. Histograms of the weights in the first layer are plotted
       for both cases, providing visual insights into their distribution characteristics:
       *)

(* Define a simple neural network structure: *)
simpleNetwork=NetChain[
  {
    LinearLayer[100,"Input"→30],
    Ramp,
    LinearLayer[100]
  }
];

(* Initialize the network using Xavier initialization with a uniform distribution: *)
initializedNetworkUniform=NetInitialize[
  simpleNetwork,
  Method→{"Xavier","Distribution"→"Uniform"}
];

(* Plot a histogram of the weights in the first layer: *)
weightsFirstLayerUniform=NetExtract[initializedNetworkUniform,{1,"Weights"}];

Histogram[
  Flatten[weightsFirstLayerUniform],
  20,
  PlotRange→All,
  ColorFunction→Function[{height},Opacity[height]],
  ChartStyle→Purple,
  ImageSize→250,
  Frame→True,
  FrameLabel→{"Weight","Frequency"},
  PlotLabel→"Xavier Initialized Weights Histogram \n (Uniform Distribution)"
]

(* Initialize the network using Xavier initialization with a normal distribution: *)
initializedNetworkNormal=NetInitialize[
  simpleNetwork,
  Method→{"Xavier","Distribution"→"Normal"}
];

weightsFirstLayerNormal=NetExtract[initializedNetworkNormal,{1,"Weights"}];

(*Plot a histogram of the weights in the first layer*)
Histogram[
  Flatten[weightsFirstLayerNormal],
  20,
  PlotRange→All,
  ColorFunction→Function[{height},Opacity[height]],
  ChartStyle→Purple,
  ImageSize→250,
  Frame→True,
]

```

```

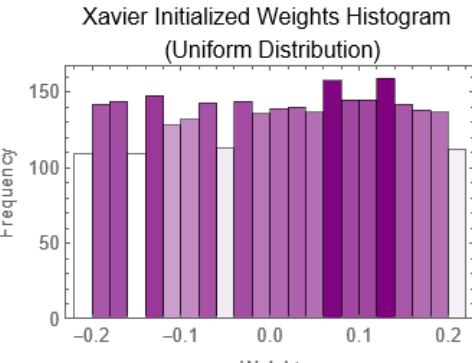
FrameLabel->{"Weight", "Frequency"},  

PlotLabel->"Xavier Initialized Weights Histogram \n (Normal Distribution)"  

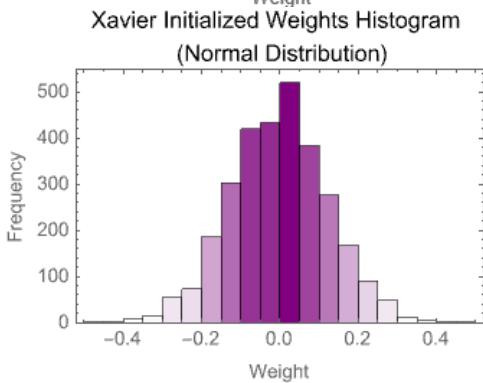
]

```

Output



Output

**Mathematica Code 5.21**

```

Input (* The code generates synthetic data and defines a neural network architecture with  
two linear layers and a Tanh activation function, initializing the weights using  
Xavier initialization with a uniform distribution. It then trains the network using  
the generated data and visualizes the distribution of weights in the first layer  
before and after training via histograms. Additionally, it utilizes kernel density  
estimation to analyze the first layer weights over 10 epochs, providing insights  
into the training process's stability and convergence: *)  
  

(* Generate synthetic data: *)  

data=Table[RandomReal[{-1,1},30]->UnitVector[2,RandomInteger[{1,2}]],{1000}];  
  

(* Define the network: *)  

net=NetChain[{LinearLayer[100,"Input"->30],Tanh,LinearLayer[2]}];  
  

(* Initialize the network using Xavier initialization with a uniform distribution:  
*)  

netInitialize=NetInitialize[net,Method->{"Xavier","Distribution"->"Uniform"}];  

weightsFirstLayerbefore=NetExtract[netInitialize,{1,"Weights"}];  
  

(* Train the network using NetTrain: *)  

trainedNet=NetTrain[netInitialize,data];  
  

(* Extract weights of the trained network: *)  

weightsFirstLayerafter=NetExtract[trainedNet,{1,"Weights"}];  
  

(* Plot a histogram of the weights in the first layer before training: *)  

Histogram[

```

```

Flatten[weightsFirstLayerbefore],
20,
PlotRange->All,
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250,
Frame->True,
FrameLabel->{"Weight","Frequency"},
PlotLabel->"Xavier Initialized Weights Histogram \n before training (Uniform
Distribution)"
]

(* Plot a histogram of the weights in the first layer after training: *)
Histogram[
Flatten[weightsFirstLayerafter],
20,
PlotRange->All,
ColorFunction->Function[{height},Opacity[height]],
ChartStyle->Purple,
ImageSize->250,
Frame->True,
FrameLabel->{"Weight","Frequency"},
PlotLabel->"Xavier Initialized Weights Histogram \n after training (Uniform
Distribution)"
]

(* Train the initialized network and monitor training progress using weights
property: *)
weights=NetTrain[
  netInitialize,
  data,
  (* Monitor training progress using Weights property: *)
  <|
    "Property"->"Weights",
    "Interval"->"Rounds"
  |>,
  MaxTrainingRounds->10
];

(* Generate data for the first layer weights for 10 epochs: *)
firstLayerWeightsData=Table[
  Flatten[
    Table[
      (* Extracting the weights of the first layer: *)
      Values[weights][[i]][[2]],
      {i,1,10}][[j]]
    ],
  {j,1,10}]//Normal;

(* Smooth kernel density estimation for first layer weights: *)
smoothKernelFirstLayer=SmoothKernelDistribution[#]&/@firstLayerWeightsData;

(* Visualize the resulting densities of the first layer weights: *)
ListLinePlot3D[
  Table[
    {(-1),x,PDF[smoothKernelFirstLayer[[#]],x]},
    {x,-1,1,0.1}
  ]&/@Range[1,10], (*Iterating through each epoch: *)
  Filling->Bottom,
  FillingStyle->Opacity[0.75],
]

```

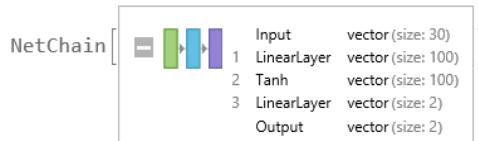
```

Joined -> True,
BoxRatios -> {1, 1, 1},
PlotRange -> All,
ImageSize -> 250,
AxesLabel -> {"Epoch", "Weights", "Density"},  

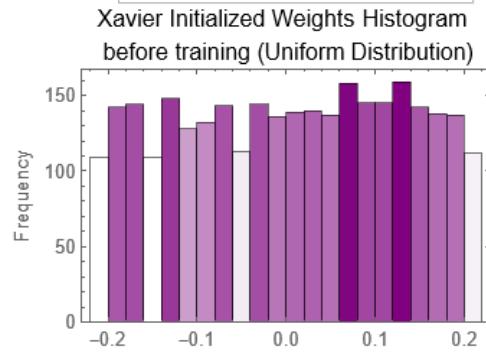
PlotLabel -> "Densities of the First Layer Weights \n for 10 Epochs"
]

```

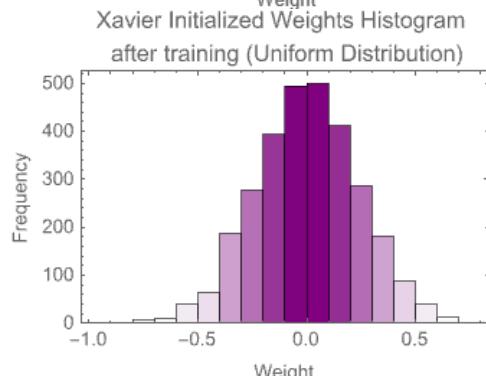
Output



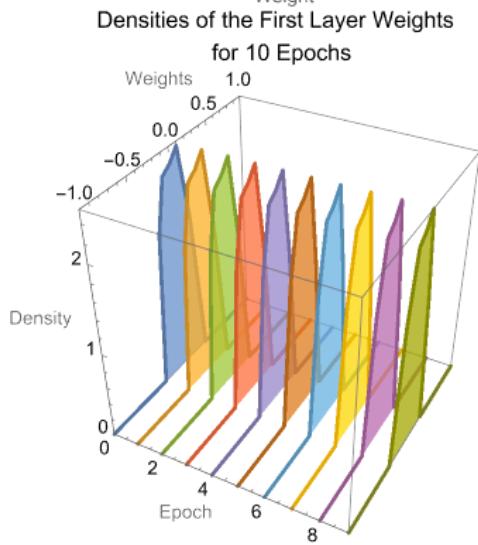
Output



Output



Output



Unit 5.5

Feature Scaling: Standardize, Whitenend and Mahalanobis Distances

Standardize[list]

shifts and rescales the elements of list to have zero mean and unit sample variance.

Standardize[list,f1]

shifts the elements in list by f1[list] and rescales them to have unit sample variance.

Standardize[list,f1,f2]

shifts by f1[list] and scales by f2[list].

Remarks:

- **Standardize** shifts by a location and rescales by a scale estimated from the elements of list.
- **Standardize[list]** is effectively $(\text{list} - \text{Mean}[\text{list}])/\text{StandardDeviation}[\text{list}]$ for nonzero $\text{StandardDeviation}[\text{list}]$.
- **Standardize[list,f1]** is effectively $(\text{list} - \text{f1}[\text{list}])/\text{StandardDeviation}[\text{list}]$.
- **Standardize[list,f1,f2]** is effectively $(\text{list} - \text{f1}[\text{list}])/f2[\text{list}]$.

Mathematica Code 5.22

```
Input    (* Compute standard scores for data: *)
        sdata=Standardize[{6.5,3.8,6.6,5.7,6.0,6.4,5.3}]
        {Mean[sdata],Variance[sdata]}

Output   {0.75705,-1.99453,0.85896,-0.0582346,0.247497,0.655139,-0.465877}
Output   {0.,1.}
```

Mathematica Code 5.23

```
Input    (* The code defines a custom standardize function that standardizes numerical data
        by subtracting the mean and dividing by the standard deviation. It then utilizes
        the Simplify function to simplify expressions obtained by applying both the custom
        standardize function and the built-in Standardize function to a list of real
        numbers, {a,b,c}, aiming to validate the custom implementation against the built-
        in functionality and potentially gain insights into the simplification process: *)
(* Define a custom function 'standardize' which takes a list of numerical data as
input: *)
standardize[data_]:=Module[
{mean,stdDev},
(* Calculate the mean and standard deviation of the input data: *)
mean=Mean[data];
stdDev=StandardDeviation[data];
(* Standardize the data by subtracting the mean and dividing by the standard
deviation: *)
(data-mean)/stdDev
]

(* Attempt to simplify the expression obtained by applying the custom 'standardize'
function to the list {a,b,c}, assuming that a, b, and c are real numbers: *)
Simplify[standardize[{a,b,c}],{a,b,c}\[Element]Reals]
```

```

(* Attempt to simplify the expression obtained by applying the built-in
'Standardize' function to the list {a,b,c}, assuming that a, b, and c are real
numbers: *)
Simplify[Standardize[{a,b,c}],{a,b,c}\[Element]Reals]
Output {
(2 a - b - c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)]),
-((a - 2 b + c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)])),
-((a + b - 2 c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)]))
}

Output {
(2 a - b - c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)]),
-((a - 2 b + c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)])),
-((a + b - 2 c)/( Sqrt[3] Sqrt[a^2 + b^2 - b c + c^2 - a (b + c)]))
}

```

Mathematica Code 5.24

```

Input (* The code generates 200 random values from a normal distribution with a mean of
4 and a standard deviation of 1, computes descriptive statistics such as mean and
standard deviation for both the original and standardized data, and plots the
original data with its mean and  $\pm$ standard deviation bands as well as the
standardized data with similar visualizations. By standardizing the data, the code
allows for comparison between the original distribution and a standard normal
distribution, providing insights into the normalization process and its effects on
the data distribution: *)

sampledData=RandomReal[NormalDistribution[4,1],200];

(* Calculate the mean and standard deviation of the data: *)

(* Calculate standard deviation: *)
standardDeviation=StandardDeviation[sampledData];
(* Calculate mean: *)
meanValue=N[Mean[sampledData]];
(* Get length of data: *)
dataLength=Length[sampledData];

(* Standardize the generated data, shifting it to a standard normal distribution:
*)

(* Standardize data: *)
standardizedData=Standardize[sampledData];
(* Calculate standard deviation of standardized data: *)
standardDeviationStandardized=StandardDeviation[standardizedData];
(* Calculate mean of standardized data: *)
meanValueStandardized=N[Mean[standardizedData]];
(* Get length of standardized data: *)
dataLengthStandardized=Length[standardizedData];

(* Plotting the original data with mean and standard deviation bands: *)
ListPlot[
{
sampledData,
{{0,meanValue},{dataLength,meanValue}},
{{0,meanValue-standardDeviation},{dataLength,meanValue-standardDeviation}},
{{0,meanValue+standardDeviation},{dataLength,meanValue+standardDeviation}}
},
Joined->{False,True,True,True},
Filling->{1->meanValue,3->{4}},
PlotStyle->{Purple,Automatic,Automatic,Automatic},

```

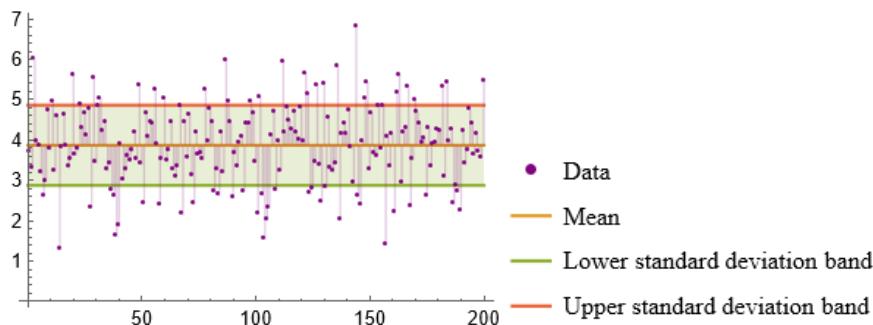
```

PlotLegends->{"Data", "Mean", "Lower standard deviation band", "Upper standard
deviation band"},
AxesLabel->Automatic,
ImageSize->250
]

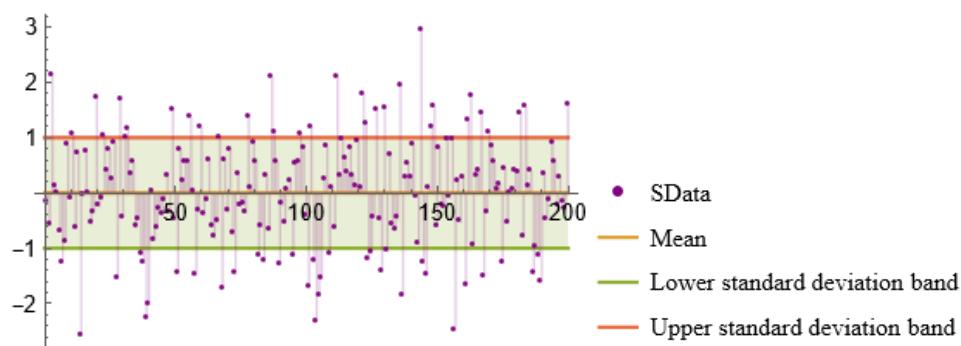
(* Plotting the standardized data with mean and standard deviation bands: *)
ListPlot[
{
  standardizedData,
  {
    {0,meanValueStandardized},
    {dataLengthStandardized,meanValueStandardized}
  },
  {
    {0,meanValueStandardized-standardDeviationStandardized},
    {dataLengthStandardized,meanValueStandardized-standardDeviationStandardized}
  },
  {
    {0,meanValueStandardized+standardDeviationStandardized},
    {dataLengthStandardized,meanValueStandardized+standardDeviationStandardized}
  }
},
Joined->{False,True,True,True},
Filling->{1->meanValueStandardized,3->{4}},
PlotStyle->{Purple,Automatic,Automatic,Automatic},
PlotLegends->{"SData", "Mean", "Lower standard deviation band", "Upper standard
deviation band"},
AxesLabel->Automatic,
ImageSize->250
]

```

Output



Output



Mathematica Code 5.25

```

Input (* This code will generate three plots: one for the original data, one for the
decorrelated data, and one for the whitened data. Additionally, it will print out
the covariance matrix of original correlated data, eigenvalues, eigenvectors,
covariance matrix of decorrelated data and covariance matrix of whitened data to
help understand the transformation steps. Decorrelation is achieved by transforming
the centered data with the eigenvectors. Whitening is done by scaling the
decorrelated data by the inverse square root of eigenvalues: *)

(* Step 1: Generate original data: *)
SeedRandom[123];
originalData=RandomVariate[MultinormalDistribution[{0,0},{{3,1.5},{1.5,2}}],1000]
;

(* Step 2: Center the data: *)
centeredData=(#-Mean[originalData])&/@originalData;

(* Step 3: Calculate covariance matrix: *)
covarianceMatrix=Covariance[centeredData];

(* Step 4: Compute eigenvalues and eigenvectors: *)
{eigenvalues,eigenvectors}=Eigensystem[covarianceMatrix];

(* Step 5: Decorrelate the data: *)
decorrelatedData=centeredData.Transpose[eigenvectors];

(* Step 6: Whiten the data: *)
whitenedData=decorrelatedData.DiagonalMatrix[1/Sqrt[eigenvalues]];

(* Calculate Covariance of decorrelated, and whitened data: *)
covDecorrelated=Covariance[decorrelatedData];
covWhitened=Covariance[whitenedData];

(* Plotting: *)
GraphicsRow[
{
ListPlot[
originalData,
PlotStyle->{PointSize[0.02],Opacity[0.4],Blue},
PlotLabel->"Original Correlated Data",
ImageSize->250
],
ListPlot[
decorrelatedData,
Epilog->{PointSize[0.02],Green,Point[covDecorrelated]},
PlotStyle->{PointSize[0.02],Opacity[0.4],Red},
PlotLabel->"Decorrelated Data",
ImageSize->250
],
ListPlot[
whitenedData,
Epilog->{PointSize[0.02],Green,Point[covWhitened]},
AspectRatio->1,
PlotStyle->{PointSize[0.02],Opacity[0.4],Purple},
PlotLabel->"Whitened Data",
ImageSize->250
]
}
]
]

```

```
(* Explanation: *)
Print["Eigenvalues: ", eigenvalues];
Print["Eigenvectors: ", eigenvectors];
Print["Covariance Matrix of Original Correlated Data: ", covarianceMatrix];
Print["Covariance Matrix of Decorrelated Data: ", covDecorrelated];
Print["Covariance Matrix of Whitened Data: ", covWhitened];
```

Output

Original Correlated Data Decorrelated Data Whitened Data

Output Eigenvalues: {3.94926, 0.953016}

Output Eigenvectors:

```
{
{-0.822513, -0.568747},
{0.568747, -0.822513}
}
```

Output Covariance Matrix of Original Correlated Data:

```
{
{2.98006, 1.40165},
{1.40165, 1.92222}
}
```

Output Covariance Matrix of Decorrelated Data:

```
{
{3.94926, 6.64399*10^-16},
{6.64399*10^-16, 0.953016}
}
```

Output Covariance Matrix of Whitened Data:

```
{
{1., 3.69713*10^-16},
{3.69713*10^-16, 1.}
}
```

Mathematica Code 5.26

Input (* The code generates a dataset from a bivariate normal distribution and conducts statistical analyses on it. Firstly, it whitens the data by removing the mean and scaling it using the inverse square root of the covariance matrix. Then, it calculates the Mahalanobis distances, which account for the covariance structure of the data, and Euclidean distances from each point to the mean. These distances are plotted for visualization alongside the original and whitened datasets, allowing for comparisons between Euclidean and Mahalanobis distances. The code provides insight into the effects of whitening on data distribution and the differences between Euclidean and Mahalanobis distance metrics in assessing data variability: *)

```
SeedRandom[23];
(* Generate data from a bivariate normal distribution: *)
sampleData=RandomVariate[BinormalDistribution[{0,0},{1,3},0.9],200];
```

```

(* Compute mean and covariance matrix of the data: *)
meanVector=Mean[sampleData];
covarianceMatrix=Covariance[sampleData];
inverseCovarianceMatrix=Inverse[covarianceMatrix];

(* Define the square root matrix function: *)
matrixSquareRoot[X_]:=Module[
{eigenSystem},
(* Compute the eigen decomposition of the input symmetric matrix X: *)
(* Extract eigenvectors and eigenvalues: *)
eigenSystem=Eigensystem[X];
(* Finally, compute the square root matrix: *)

Transpose[eigenSystem[[2]]].DiagonalMatrix[Sqrt[eigenSystem[[1]]]].eigenSystem[[2]
]
(* eigenSystem[[1]] contains eigenvalues, eigenSystem[[2]] contains eigenvectors: *)
]

(* Whiten the data: *)
whitenedData=Transpose[matrixSquareRoot[inverseCovarianceMatrix].Transpose[(sample
Data-ConstantArray[meanVector,Length[sampleData]])]];

(* Define the Mahalanobis distance function: *)
mahalanobisDistance[x_,meanVec_,covMatrix_]:=Sqrt[(x-
meanVec).Inverse[covMatrix].(x-meanVec)]

(* Calculate Mahalanobis distances: *)
mahalanobisDistances=mahalanobisDistance[#,meanVector,covarianceMatrix]&/@sampleDa
ta;

(* Calculate Euclidean distances: *)
euclideanDistances=EuclideanDistance[#,meanVector]&/@sampleData;

(* Calculate Euclidean distances of whitened Data: *)
euclideanDistancesOfWhitened=EuclideanDistance[#, {0,0}]&/@whitenedData;

(* Plot original and whitened data: *)
ListPlot[
{sampleData,whitenedData},
AspectRatio->1.7,
PlotRange->All,
PlotLegends->Placed[{"Original Data", "Whitened Data"}, {0.7,0.1}],
PlotStyle->{{PointSize[0.02],Opacity[0.4],Blue},{PointSize[0.02], ,Opacity[0.4]
Red}},
PlotLabel->"(a)",
ImageSize->200
]

(* Plot Mahalanobis distances of original data: *)
mahalanobisDistancesPlot=ListPlot[
mahalanobisDistances,
Filling->Axis,
PlotStyle->{Opacity[0.8],Purple},
PlotLabel->"(c)",
PlotRange->Full,
PlotLegends->Placed[{"Mahalanobis Distances of Original Data"}, {0.5,0.9}],
ImageSize->300
]

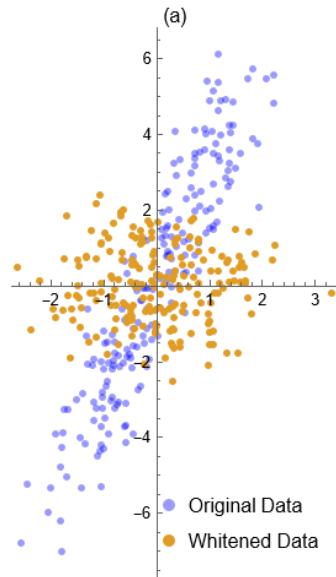
```

```
(* Plot Euclidean distances of original data: *)
euclideanDistancesPlot=ListPlot[
  euclideanDistances,
  Filling->Axis,
  PlotStyle->{Opacity[0.8],Red},
  PlotRange->All,
  PlotLegends->Placed[{"Euclidean Distances of Original Data"},{0.5,0.8}],
  ImageSize->300
];

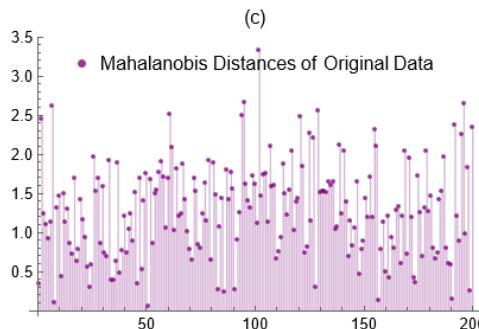
(* Plot Euclidean distances of whitened data: *)
euclideanDistancesOfWhitenedPlot=ListPlot[
  euclideanDistancesOfWhitened,
  Filling->Axis,
  PlotStyle->{Opacity[0.8],Blue},
  PlotRange->All,
  PlotLabel->"(d)",
  PlotLegends->Placed[{"Euclidean Distances of Whitened Data"},{0.5,0.9}],
  ImageSize->300
]

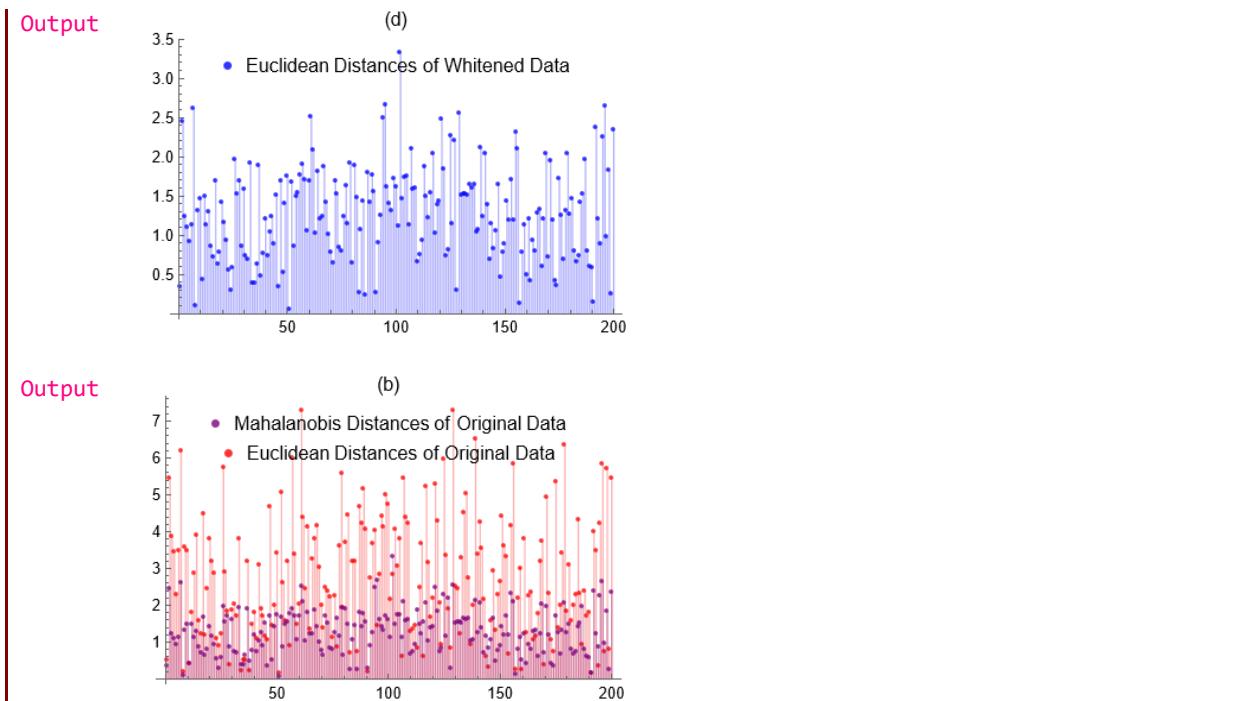
(* Combine plots of Euclidean and Mahalanobis distances: *)
Show[{euclideanDistancesPlot,mahalanobisDistancesPlot},PlotLabel->"(b)"]
```

Output



Output





Unit 5.6

BatchNormalizationLayer

MovingAverage[list,r]

gives the moving average of list, computed by averaging runs of r elements.

MovingAverage[list,{w1,w2,...,wr}]

gives the moving average of list, computed with weights wi.

ExponentialMovingAverage[list, α]

gives the exponential moving average of list with smoothing constant α .

MovingMedian[list,r]

gives the moving median of list, computed using spans of r elements.

MovingMap[f,data,w]

applies f to size w windows in the specified data.

MovingMap[f,data,wspec]

uses windows specified by wspec.

Remarks:

- **MovingAverage[list,r]** gives a list of the means of elements in list taken in blocks of length r.
- **MovingAverage** gives a list of length **Length[list]-r+1**.
- The smoothing constant α is typically a number between 0 and 1, but can be any expression.
- **ExponentialMovingAverage[x, α]** generates a list of results in which .
- The output from **ExponentialMovingAverage[list, α]** has the same length as list.
- **MovingMedian** gives a list of the medians of elements in list taken in blocks of length r.
- **MovingMedian** gives a list of length **Length[list]-r+1**.

Mathematica Code 5.27

```
Input (* Compute a pairwise simple moving average: *)
      MovingAverage[{a,b,c,d,e},2]

      (* Compute weighted moving averages: *)
      MovingAverage[{a,b,c,d,e},{1,2}]

Output {{(a+b)/2,(b+c)/2,(c+d)/2,(d+e)/2}
Output {1/3 (a+2 b),1/3 (b+2 c),1/3 (c+2 d),1/3 (d+2 e)}
```

Mathematica Code 5.28

```
Input (* Calculate the moving average of a list of integers: *)
      MovingAverage[{1,5,7,3,6,2},3]

      (* Calculate the moving average of a list of approximate numbers: *)
      MovingAverage[{1.2,5.2,3.4,4.5,2.3,4.5},3]

Output {13/3,5,16/3,11/3}
Output {3.26667,4.36667,3.4,3.76667}
```

Mathematica Code 5.29

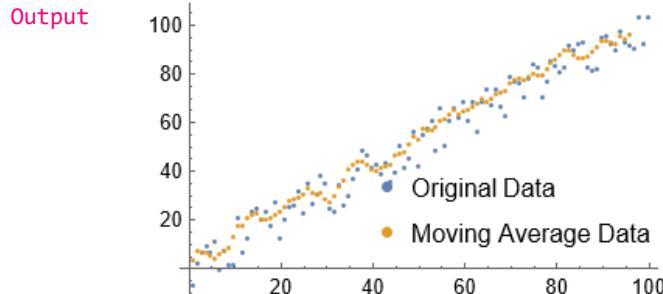
```
Input (*Smooth noisy data:*)
```

```
(* The goal of this code is to generate a dataset with noisy fluctuations, then
smooth out these fluctuations using a moving average filter, and finally visualize
both the original noisy data and the smoothed data: *)
```

```
(* Generate noisy data: *)
noisyData=Range[100]+RandomReal[{-10,10},100];

(* Calculate moving average to smooth the data: *)
smoothedData=MovingAverage[noisyData,5];
(* Using a window size of 5 for the moving average *)

(* Plotting the original noisy data and the smoothed data: *)
ListPlot[
{noisyData,smoothedData},
ImageSize->250,
PlotLegends->Placed[{"Original Data","Moving Average Data"},{0.7,0.3}]
]
```



Mathematica Code 5.30

Input

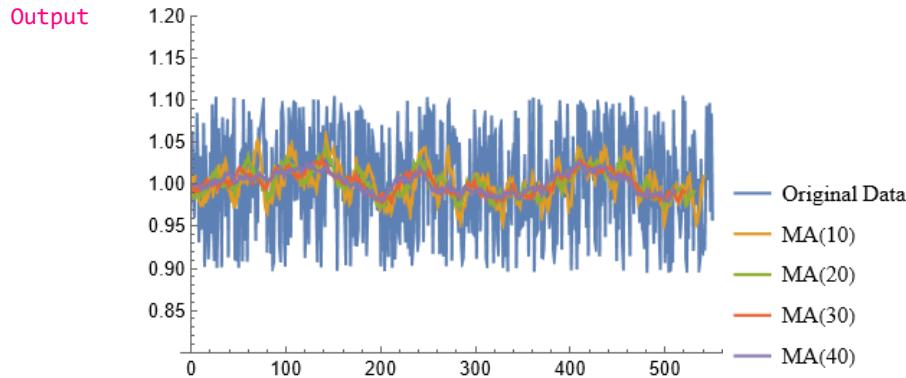
```
(* The code generates synthetic data with random noise added to a constant array,
mimicking real-world data fluctuations. It then calculates moving averages for
different window sizes (ranging from 10 to 40) to smooth out the data and highlight
underlying trends. Finally, it plots the original data alongside the moving
averages, offering a visual representation of the trend and the effectiveness of
different window sizes in capturing the underlying patterns: *)
```

```
(* Generate synthetic data with random noise: *)
initialData=ConstantArray[1.,550]+RandomReal[{-0.1,0.1},550];

(* Calculate moving averages: *)
movingAverages=Table[
  MovingAverage[initialData,WindowSizes],
  {WindowSizes,10,40,10}
];

(* Prepend original data to the list of moving averages for plotting: *)
plotData=Prepend[movingAverages,initialData[[1;;550]]];

(* Plotting: *)
ListLinePlot[
  plotData,
  PlotRange->{0.8,1.2},(*Set the range of the y-axis: *)
  PlotLegends->{"Original Data","MA(10)","MA(20)","MA(30)","MA(40)"},
  ImageSize->300
]
```

**Mathematica Code 5.31**

```
Input (* Exponential moving average in symbolic form: *)
ExponentialMovingAverage[{a,b,c}, $\alpha$ ]

(* Exponential moving average for numeric values: *)
ExponentialMovingAverage[Range[10],1/3]

Output {a,-a (-1+ $\alpha$ )+b  $\alpha$ ,c  $\alpha$ -(-1+ $\alpha$ ) (-a (-1+ $\alpha$ )+b  $\alpha$ )}
Output {1, 4/3, 17/9, 70/27, 275/81, 1036/243, 3773/729, 13378/2187, 46439/6561,
158488/19683}
```

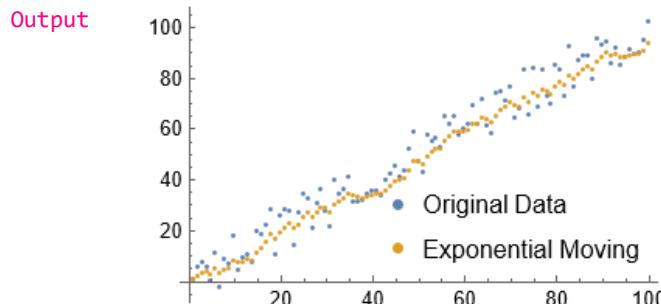
Mathematica Code 5.32

```
Input (* The goal of this code is to generate a dataset with noisy fluctuations, then
smooth out these fluctuations using an Exponential Moving Average filter, and
finally visualize both the original noisy data and the smoothed data: *)

(* Generate noisy data: *)
noisyData=Range[100]+RandomReal[{-10,10},100];

(* Calculate Exponential Moving Average to smooth the data: *)
smoothedData=ExponentialMovingAverage[noisyData,0.25];
(* Using a smoothing factor of 0.25 for the Exponential Moving Average *)

(* Plotting the original noisy data and the smoothed data: *)
ListPlot[
{noisyData,smoothedData},
ImageSize->250,
PlotLegends->Placed[{"Original Data","Exponential Moving"},{0.7,0.25}]
]
```

**Mathematica Code 5.33**

```
Input (* The code generates synthetic data with random noise added to a constant array,
mimicking real-world data fluctuations. It then calculates Exponential Moving
```

Average for different smoothing constants (0.1, 0.3 and 0.6) to smooth out the data and highlight underlying trends. Finally, it plots the original data alongside the Exponential Moving Average, offering a visual representation of the trend and the effectiveness of different smoothing constants in capturing the underlying patterns:
*)

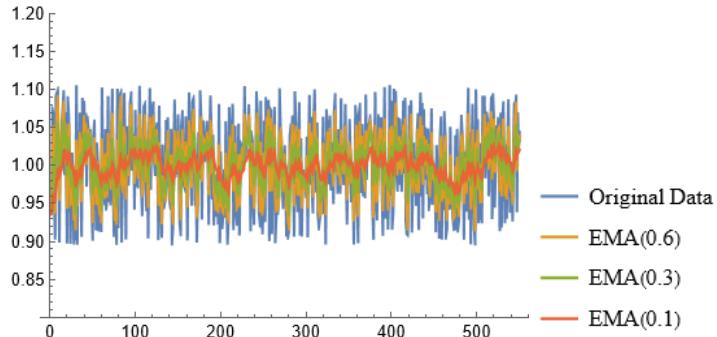
```
(* Generate synthetic data with random noise: *)
initialData=ConstantArray[1.,550]+RandomReal[{-0.1,0.1},550];

(* Calculate Exponential Moving Average: *)
exponentialMovingAverage=Table[
  ExponentialMovingAverage[initialData,smoothingconstant],
  {smoothingconstant,{0.6,0.3,0.1}}
];

(* Prepend original data to the list of Exponential Moving Average for plotting: *)
plotData=Prepend[exponentialMovingAverage,initialData[[1;;550]]];

(* Plotting: *)
ListLinePlot[
  plotData,
  PlotRange->{0.8,1.2},(*Set the range of the y-axis*)
  PlotLegends->{"Original Data","EMA(0.6)","EMA(0.3)","EMA(0.1)"},
  ImageSize->300
]
```

Output

**Mathematica Code 5.34**

Input	(* Calculate the Exponential Moving Average with Smoothing Constant 0: *) ExponentialMovingAverage[{a,b,c,d},0] (* Exponential moving average with a smoothing constant of 0 is a constant: *)
	(* Calculate the Exponential Moving Average with Smoothing Constant 1: *) ExponentialMovingAverage[{a,b,c,d},1] (* Exponential moving average with a smoothing constant of 1 is the original list: *)

Output	{a,a,a,a}
Output	{a,b,c,d}

Mathematica Code 5.35

Input	(* The code aims to offer an interactive platform for exploring moving averages and exponential moving averages applied to synthetic data. By setting a random seed for reproducibility, it generates a dataset with adjustable sample size and introduces sliders to dynamically modify the window size of the moving averages. Through ListLinePlot, the original data, moving average, and exponential moving average are
-------	--

visually represented with distinct colors, aiding in comprehension. The interactive interface allows users to observe how changes in the random seed, sample size, and window size influence the smoothing effect on the data, facilitating a deeper understanding of these statistical techniques within a visual context: *)

```
Manipulate[
(* Set seed for reproducibility: *)
SeedRandom[seed];
(* Generate synthetic data with random noise: *)
initialData=ConstantArray[1.,numSamples]+RandomReal[{-0.1,0.1},numSamples];

(* Calculate moving averages: *)
movingAverage=MovingAverage[initialData,windowSize];
exponentialMovingAverage=ExponentialMovingAverage[initialData,windowSize/100];

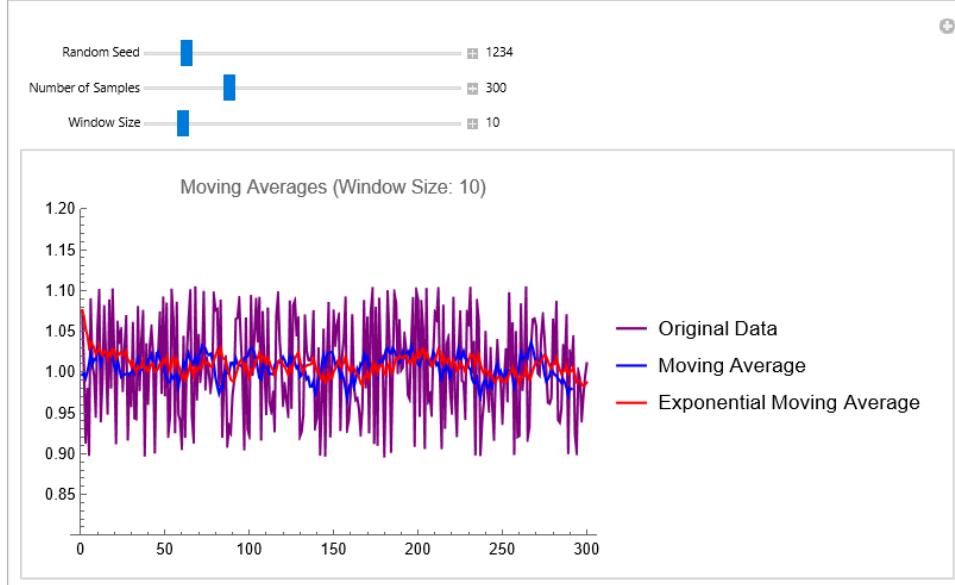
(* Prepend original data to the list of moving averages for plotting: *)
plotData={initialData,movingAverage,exponentialMovingAverage};
(* Plotting: *)
ListLinePlot[
plotData,
PlotRange->{0.8,1.2},(* Set the range of the y-axis: *)
PlotLegends->>{"Original Data","Moving Average","Exponential Moving Average"},

PlotStyle->{Purple,Blue,Red},
ImageSize->350,
PlotLabel->Row[{ "Moving Averages (Window Size: ",windowSize,")"}]
],
{{seed,1234,"Random Seed"},1,10000,1,Appearance->"Labeled"},

{{numSamples,300,"Number of Samples"},50,1000,50,Appearance->"Labeled"},

{{windowSize,10,"Window Size"},5,50,1,Appearance->"Labeled"}
]
```

Output

**Mathematica Code 5.36**

Input (* Moving median of a vector: *)
MovingMedian[{1,2,5,6,1,4,3},3]

Output {2,5,5,4,3}

Mathematica Code 5.37

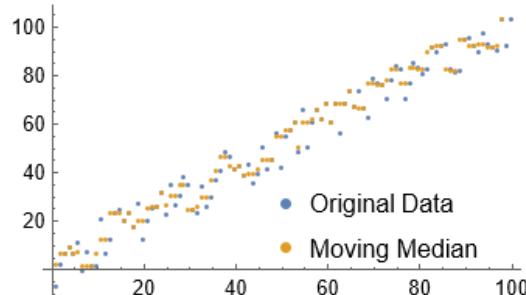
Input

```
(* The goal of this code is to generate a dataset with noisy fluctuations, then
smooth out these fluctuations using a Moving Median filter, and finally visualize
both the original noisy data and the smoothed data: *)

(* Generate noisy data: *)
noisyData=Range[100]+RandomReal[{-10,10},100];

(* Calculate Moving Median to smooth the data: *)
smoothedData=MovingMedian[noisyData,3];
(* Using blocks of length 3 for the Moving Median *)

(* Plotting the original noisy data and the smoothed data: *)
ListPlot[
{noisyData,smoothedData},
ImageSize->250,
PlotLegends->Placed[{"Original Data","Moving Median"},{0.7,0.25}]
]
```

Output**Mathematica Code 5.38**

Input

```
(* MovingMap of Mean over regular data is equivalent to MovingAverage: *)
data={Subscript[x,1],Subscript[x,2],Subscript[x,3],Subscript[x,4],Subscript[x,5],Sub
script[x,6]};
MovingMap[Mean,data,Quantity[3,"Events"]]
MovingAverage[data,3]
```

Output

```
{1/3 (x1+x2+x3),1/3 (x2+x3+x4),1/3 (x3+x4+x5),1/3 (x4+x5+x6)}
```

Output

```
{1/3 (x1+x2+x3),1/3 (x2+x3+x4),1/3 (x3+x4+x5),1/3 (x4+x5+x6)}
```

Mathematica Code 5.39

Input

```
(* MovingMap of Median over regular data is equivalent to MovingMedian: *)
data=RandomInteger[{-10,10},10]
MovingMap[Median,data,Quantity[3,"Events"]]
MovingMedian[data,3]
```

Output

```
{-9,5,10,-7,6,2,1,7,3,-10}
```

Output

```
{5,5,6,2,2,2,3,3}
```

Output

```
{5,5,6,2,2,2,3,3}
```

Mathematica Code 5.40

Input

```
(* MovingMap of Variance over regular data: *)
data=RandomInteger[{-10,10},40]
MovingMap[Variance,data,Quantity[10,"Events"]]
```

Output

```
{-8,-1,3,6,-1,3,4,1,-3,5,-6,4,9,-1,-4,1,-2,10,-1,-8,0,0,-3,-1,10,9,5,0,-8,-4,4,-8,-5,-10,-8,-3,-10,-7,6,-4}
```

```
Output {181/10, 1309/90, 662/45, 908/45, 115/6, 326/15, 64/3, 314/15, 907/30, 517/18,
1598/45, 1388/45, 148/5, 316/15, 316/15, 1382/45, 1702/45, 1123/30, 2689/90,
1712/45, 1448/45, 496/15, 208/5, 1958/45, 4861/90, 785/18, 2461/90, 938/45,
1589/90, 553/18, 553/18}
```

BatchNormalizationLayer[]

represents a trainable net layer that normalizes its input data by learning the data mean and variance.

The following optional parameters can be included:

"Epsilon"	0.001`	stability parameter
Interleaving	False	the position of the channel dimension
"Momentum"	0.9	momentum used during training

The following learnable arrays can be included:

"Biases"	Automatic	learnable bias array
"MovingMean"	Automatic	moving estimate of the mean
"MovingVariance"	Automatic	moving estimate of the variance
"Scaling"	Automatic	learnable scaling array

Remarks:

- With Automatic settings, the biases, scaling, moving mean and moving variance arrays are initialized automatically when `NetInitialize` or `NetTrain` is used.
- If biases, scaling, moving variance and moving mean have been set, `BatchNormalizationLayer[...][input]` explicitly computes the output from applying the layer.
- `BatchNormalizationLayer[...][{input1, input2, ...}]` explicitly computes outputs for each of the inputs.

Mathematica Code 5.41

```
Input (* Create a BatchNormalizationLayer: *)
BatchNormalizationLayer[]
```

Output

```
BatchNormalizationLayer[
```

Parameters	
uninitialized	Momentum: 0.9
	Epsilon: 0.001
	Interleaving: False
Arrays	
Scaling:	vector
Biases:	vector
MovingMean:	vector
MovingVariance:	vector
Input Port	
Input:	matrix(rank≥1)
Output Port	
Output:	matrix(rank≥1)

Mathematica Code 5.42

```
Input (* Create an initialized BatchNormalizationLayer that takes a vector and returns a
vector: *)
batchnorm=NetInitialize@BatchNormalizationLayer["Input" -> 3]

(* Extract the "Biases", "Scaling", "MovingMean" and "MovingVariance" parameters:*)

(* The default value for "Biases" chosen by NetInitialize is a zero vector: *)
NetExtract[batchnorm,"Biases"]//Normal
```

```
(* The default value for "Scaling" chosen by NetInitialize is a vector of 1s: *)
NetExtract[batchnorm, "Scaling"]//Normal
(* The default value for "MovingMean" chosen by NetInitialize is a zero vector:*)
NetExtract[batchnorm, "MovingMean"]//Normal
(* The default value for "MovingVariance" chosen by NetInitialize is a vector of
1s: *)
NetExtract[batchnorm, "MovingVariance"]//Normal
(* The default value for "Epsilon" chosen by NetInitialize is a vector of 0.001:*)
NetExtract[batchnorm, "Epsilon"]
(* The default value for "Momentum" chosen by NetInitialize is a vector of 0.9:*)
NetExtract[batchnorm, "Momentum"]

(* Apply the layer to an input vector: *)
batchnorm[{2,3,4}]
```

Output

BatchNormalizationLayer []

Parameters	
Momentum:	0.9
Epsilon:	0.001
Interleaving:	False
Arrays	
Scaling:	vector(size: 3)
Biases:	vector(size: 3)
MovingMean:	vector(size: 3)
MovingVariance:	vector(size: 3)
Input Port	
Input:	vector(size: 3)
Output Port	
Output:	vector(size: 3)

```
Output {0.,0.,0.}
Output {1.,1.,1.}
Output {0.,0.,0.}
Output {1.,1.,1.}
Output 0.001
Output 0.9
Output {1.999, 2.9985, 3.998}
```

Mathematica Code 5.43

Input

```
(* The code initializes a custom BatchNormalizationLayer with explicitly specified
parameters, including scaling factors, biases, epsilon value, momentum, and initial
values for moving mean and variance. By setting these parameters, the code aims to
customize the behavior of the BatchNormalizationLayer according to specific
requirements or experimental setups. Subsequently, the layer is applied to an input
vector to observe its normalization process and the resulting output, thereby
demonstrating how the specified parameters influence the layer's operation and its
effect on input data normalization within neural network architectures: *)
```

```
(* Create a BatchNormalizationLayer with the parameters explicitly specified: *)
batchnorm=NetInitialize@BatchNormalizationLayer[
  "Scaling" -> {1, 1, 1},
  "Biases" -> {1, -2, 1},
  "Epsilon" -> 0.1,
  "Momentum" -> 0.1,
  "MovingMean" -> {2, 2, 2},
  "MovingVariance" -> {2, 2, 2}
]
(* Apply the layer to an input vector: *)
batchnorm[{2,3,4}]
```

Output

BatchNormalizationLayer[

	Parameters
Momentum:	0.1
Epsilon:	0.1
Interleaving:	False
Arrays	
Scaling:	vector(size: 3)
Biases:	vector(size: 3)
MovingMean:	vector(size: 3)
MovingVariance:	vector(size: 3)
Input Port	
Input:	matrix(size: 3x...)
Output Port	
Output:	matrix(size: 3x...)

Output {1., -1.30993, 2.38013}

Mathematica Code 5.44

```
Input (* The code initializes a BatchNormalizationLayer with specific parameters:
 "Scaling" is set to {0,0,0}, indicating no scaling, and "Biases" is set to {1,2,20}.
 The purpose is to demonstrate that when the layer is applied to any input data, it
 returns the values specified for the "Biases" parameter, disregarding the input
 entirely. This behavior showcases how the biases directly affect the output of the
 layer, irrespective of the input values: *)
(* Create an initialized BatchNormalizationLayer with the "Scaling" parameter set
 to zero and the "Biases" parameter set to a custom value: *)
batchnorm=NetInitialize@BatchNormalizationLayer[
  "Scaling"->{0,0,0},
  "Biases"->{1,2,20}
]

(* Applying the layer to any input returns the value for the "Biases" parameter:*)
batchnorm[{1,2,3}]
batchnorm[{3,4,5}]
```

Output

BatchNormalizationLayer[

	Parameters
Momentum:	0.9
Epsilon:	0.001
Interleaving:	False
Arrays	
Scaling:	vector(size: 3)
Biases:	vector(size: 3)
MovingMean:	vector(size: 3)
MovingVariance:	vector(size: 3)
Input Port	
Input:	matrix(size: 3x...)
Output Port	
Output:	matrix(size: 3x...)

Output {1., 2., 20.}

Output {1., 2., 20.}

Mathematica Code 5.45

```
Input (* The code presents two methods for conducting batch normalization: a custom
 function, batchNormalizationLayer, and Mathematica's built-in
 BatchNormalizationLayer. The custom function is designed to manually compute batch
 normalization, allowing explicit control over parameters such as scaling, biases,
```

moving means, and variances. Conversely, the built-in function streamlines the process by initializing a batch normalization layer with predefined parameters, offering a more concise and efficient solution. By manually computing batch normalization, we gain a deeper understanding of its inner workings. However, it's important to note that in practice, deep learning frameworks like Mathematica provide efficient built-in implementations that handle the computation automatically, allowing for easier experimentation and faster training: *)

```

batchNormalizationLayer[input_,scalings_,biases_,movingMeans_,movingVariances_,
epsilon_]:=Module[
{sd=Sqrt[movingVariances+epsilon]},
(scalings*(input-movingMeans)/sd+biases)
]

(*Example usage*)
scalings=0.5;
biases=0.1;
movingMeans=0.2;
movingVariances=0.3;
epsilon=10^-5;
input=0.3;
normalizedOutput=batchNormalizationLayer[input,scalings,biases,movingMeans,
movingVariances,epsilon]

batchnorm=NetInitialize@BatchNormalizationLayer[
"Scaling" -> 0.5,
"Biases" -> 0.1,
"Epsilon" -> 10^-5,
"MovingMean" -> 0.2,
"MovingVariance" -> 0.3,
"Input" -> 1
];
batchnorm[0.3]

Output 0.191286
Output {0.191286}

```

Mathematica Code 5.46

Input (* The code aims to create an interactive environment for exploring the effects of batch normalization on a randomly generated dataset. It allows users to adjust parameters such as the number of samples, scaling, biases, epsilon, momentum, moving mean, moving variance, and random seed using sliders. The code generates a random dataset based on the specified parameters and applies a custom BatchNormalizationLayer to normalize the data. The original and normalized datasets are then plotted on a graph, with the original data represented in blue and the normalized data in red. This interactive visualization enables users to observe how changes in batch normalization parameters impact the distribution and normalization of the dataset, facilitating experimentation and understanding of batch normalization techniques: *)

```

Manipulate[
Module[
{data,normalizedData,batchnorm},
(* Set the random seed: *)
SeedRandom[seed];
(* Generate a random dataset: *)
data=RandomReal[{-10,10},{numberOfSamples,1}];
(* Initialize the BatchNormalizationLayer with custom parameters: *)
batchnorm=NetInitialize@BatchNormalizationLayer[
"Scaling" -> scaling,

```

```

"Biases" -> biases,
"Epsilon" -> epsilon,
"Momentum" -> momentum,
"MovingMean" -> movingMean,
"MovingVariance" -> movingVariance,
"Input" -> 1
];
(* Apply the BatchNormalizationLayer to the dataset: *)
normalizedData=Flatten[batchnorm/@data];

(* Ensure originalData is a flat list: *)
originalData=Flatten[data];
(* Plot the original and normalized datasets: *)
ListPlot[
{
(* Original data: *)
Transpose[{Range[numberOfSamples],originalData}],
(* Normalized data: *)
Transpose[{Range[numberOfSamples],normalizedData}]
},
PlotStyle->{Blue,Red},
PlotLegends->{"Original Data","Normalized Data"},
Frame->True,
FrameLabel->{"Sample","Value"},
PlotRange->Full,
Joined->True,
ImageSize->300
]
],
(* Manipulate sliders to control parameters: *)
{{numberOfSamples,200,"Number of Samples"},10,500,10,Appearance->"Labeled"},

{{scaling,{1}),"Scaling"},0,2,0.1,Appearance->"Labeled"},

{{biases,{1}),"Biases"},-2,2,0.1,Appearance->"Labeled"},

{{epsilon,0.1),"Epsilon"},0.01,1,0.01,Appearance->"Labeled"},

{{momentum,0.1),"Momentum"},0,1,0.1,Appearance->"Labeled"},

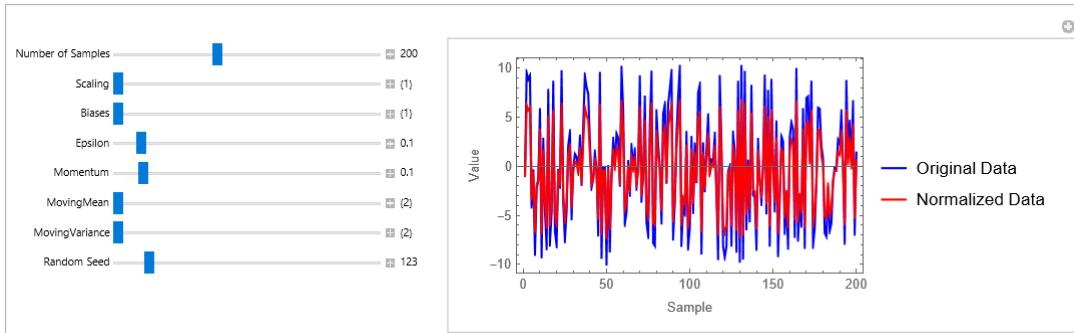
{{movingMean,{2}),"MovingMean"},0,5,0.1,Appearance->"Labeled"},

{{movingVariance,{2}),"MovingVariance"},0,5,0.1,Appearance->"Labeled"},

{{seed,123),"Random Seed"},0,1000,1,Appearance->"Labeled"}
]
]

```

Output



Mathematica Code 5.47

Input (* The code aims to explore the impact of overfitting and the effectiveness of Batch Normalization in improving generalization performance in neural networks (regularization). It begins by generating a noisy dataset based on a Gaussian curve, then constructs and trains a Multilayer Perceptron (MLP) without Batch Normalization layers. The code identifies overfitting, where the trained MLP captures both the underlying function and the noise in the data. Subsequently, it introduces Batch

```
Normalization layers into the MLP architecture, trains the modified network, and evaluates its performance. The Batch-Normalized MLP demonstrates better generalization, providing a smoother fit to the original Gaussian curve and mitigating the overfitting observed in the initial MLP: *)  
  
(* Generate a dataset with noise based on a Gaussian distribution: *)  
noisyData=Table[x->Exp[-x^2]+RandomVariate[NormalDistribution[0,.15]],{x,-3,3,.2}];  
  
(* Visualize the generated noisy dataset: *)  
noisyPlot=ListPlot[  
  List@@@noisyData,  
  PlotStyle->Purple,  
  PlotLabel->"Noisy Dataset",  
  ImageSize->250  
]  
  
(* Define a neural network architecture with two hidden layers of 150 units each and Tanh activation functions: *)  
mlp=NetChain[{150,Tanh,150,Tanh,1}]  
  
(* Train the neural network using the noisy dataset: *)  
mlpTrainingResults=NetTrain[mlp,noisyData,All]  
  
(* Evaluate the rounded loss of the trained neural network: *)  
mlpRoundLoss=mlpTrainingResults["RoundLoss"]  
  
(* The resulting neural network overfits the data, capturing both the underlying function and the noise. To visualize this, plot the function learned by the network alongside the original data: *)  
  
(* Extract the trained neural network from the training results: *)  
overfittedMLP=mlpTrainingResults["TrainedNet"];  
  
(* Plot the function learned by the trained neural network alongside the original noisy dataset: *)  
Show[  
  Plot[  
    overfittedMLP[x],  
    {x,-3,3},  
    PlotLabel->"Overfitted MLP",  
    ImageSize->250  
  ],  
  noisyPlot  
]  
  
(* Define a neural network architecture with Batch Normalization layers after each hidden layer: *)  
batchNormMLP=NetChain[  
{  
  150,BatchNormalizationLayer[],Tanh,  
  150,BatchNormalizationLayer[],Tanh,  
  1  
}]  
  
(* Train the Batch-Normalized neural network using the noisy dataset: *)  
batchNormTrainingResults=NetTrain[batchNormMLP,noisyData,All]
```

```
(* Extract the trained Batch-Normalized neural network from the training results:
*)
batchNormTrainedNet=batchNormTrainingResults["TrainedNet"];

(* Plot the function learned by the trained Batch-Normalized neural network
alongside the original noisy dataset: *)
Show[
Plot[
batchNormTrainedNet[x],
{x,-3,3},
PlotLabel->"Batch-Normalized MLP",
ImageSize->250
],
noisyPlot
]

```

Output

Output

Noisy Dataset

Output

NetChain

uninitialized	Input	array
1	LinearLayer	vector(size: 150)
2	Tanh	vector(size: 150)
3	LinearLayer	vector(size: 150)
4	Tanh	vector(size: 150)
5	LinearLayer	vector(size: 1)
Output	Output	vector(size: 1)

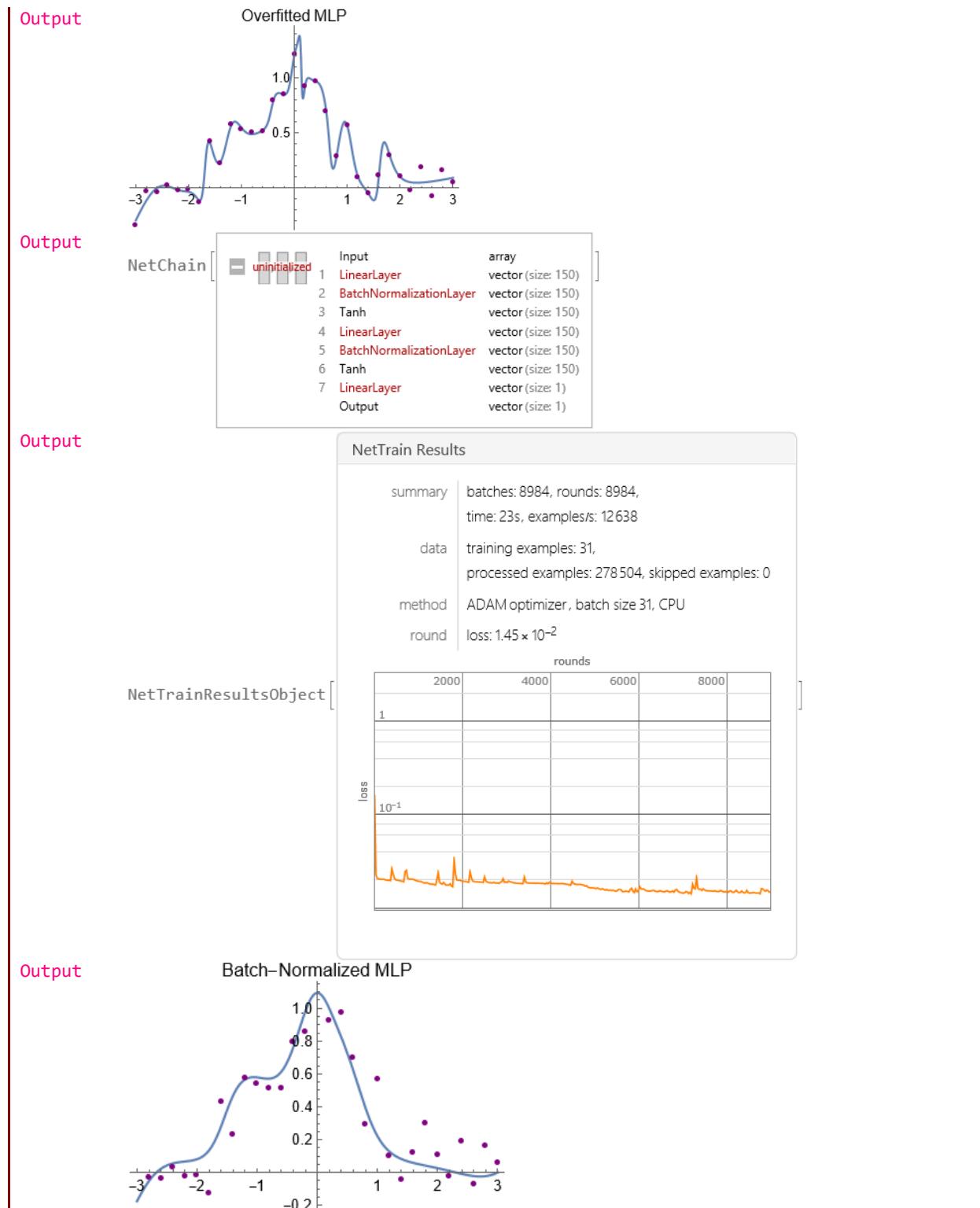
Output

NetTrain Results

summary	batches: 10 000, rounds: 10 000, time: 24s, examples/s: 13 033
data	training examples: 31, processed examples: 310 000, skipped examples: 0
method	ADAM optimizer, batch size 31, CPU
round	loss: 2.05×10^{-3}

Output

NetTrainResultsObject

**Mathematica Code 5.48****Input**

```
(* The code accomplishes several tasks in the context of neural network training and analysis. It begins by generating synthetic input-output data using a mixture distribution. Then, it defines a neural network architecture with BatchNormalization layers and LogisticSigmoid activations, initializes the network with random weights
```

and biases, and proceeds to train it using the generated data while monitoring activation output statistics. The code extracts and processes activation output data, calculating mean and standard deviation values for both BatchNormalization and LogisticSigmoid layers. Visualization steps include plotting mean activation values with error bars over training epochs, performing kernel density estimation and visualization, and creating histograms to depict the distribution of activation values for specific layers. BatchNormalization normalizes the inputs to each layer, helping to prevent activations from becoming too large or too small, which can lead to saturation. By normalizing the inputs, BatchNormalization helps to keep the activations within a more stable range, thereby alleviating the saturation problem associated with the logistic sigmoid function. The figures generated by the code illustrate various aspects of the neural network training process and activation behaviors. The first set of figures depicts the mean activation values with error bars over training epochs for both BatchNormalization and LogisticSigmoid layers, showcasing their convergence patterns and variability. Additionally, kernel density estimation plots visualize the distribution of activation values for specific layers, demonstrating the spread and density of activations throughout training. Histograms further elucidate the distribution of activation values for specific LogisticSigmoid layers, offering insights into the diversity and concentration of activations: *)

```
(* Generate synthetic input-output data: *)
distribution=MixtureDistribution[{1,2},{NormalDistribution[1,5],NormalDistribution[4,6]}];
inputData=RandomVariate[distribution,{2000,200}];
outputData=RandomReal[2,{2000,1}];

(* Define the neural network architecture: *)
neuralNet=NetGraph[
  {
    (* Two neurons with logistic sigmoid activation for the first layer: *)
    2,BatchNormalizationLayer[],LogisticSigmoid,
    (* Two neurons with logistic sigmoid activation for the second layer: *)
    2,BatchNormalizationLayer[],LogisticSigmoid,
    (* Two neurons with logistic sigmoid activation for the third layer: *)
    2,BatchNormalizationLayer[],LogisticSigmoid,
    (* One neuron with logistic sigmoid activation for the output layer: *)
    1,LogisticSigmoid
  },
  {1->2->3->4->5->6->7->8->9->10->11},
  (* Define input size: *)
  "Input"->200
];

(* Initialize the network with random weights and biases: *)
initializedNet=NetInitialize[
  neuralNet,
  (*Specify random initialization method*)
  Method->{"Random","Weights"->1,"Biases"->1},
  (*Ensure reproducibility by setting random seed*)
  RandomSeeding->Automatic
]

(* Train the network: *)
trainingResults=NetTrain[
  initializedNet,
  (* Provide input-output pairs for training: *)
  <|"Input"->inputData,"Output"->outputData|>,
  All,
  (* Limit the number of training rounds: *)
  MaxTrainingRounds->100,
  TrainingProgressMeasurements->{
```

```

(* Track activation output statistics during training: *)
<|"Measurement"->NetPort[{2,"Output"}],"Interval"->"Round"|>,
<|"Measurement"->NetPort[{5,"Output"}],"Interval"->"Round"|>,
<|"Measurement"->NetPort[{8,"Output"}],"Interval"->"Round"|>,
<|"Measurement"->NetPort[{3,"Output"}],"Interval"->"Round"|>,
<|"Measurement"->NetPort[{6,"Output"}],"Interval"->"Round"|>,
<|"Measurement"->NetPort[{9,"Output"}],"Interval"->"Round"|>
}
];

(* Extract activation output data for each layer: *)
trainingData=trainingResults["RoundMeasurementsLists"];
{
BNactivationOutputLayer2,
BNactivationOutputLayer5,
BNactivationOutputLayer8,
LogisticactivationOutputLayer3,
LogisticactivationOutputLayer6,
LogisticactivationOutputLayer9
}=Values[trainingData][[{2,3,4,5,6,7}]]];

(* Function to process activation output data and calculate mean and standard
deviation: *)
processLayer[output_]:=Module[
{mean,stdDev,meanWithStdDev},
mean=Mean/@output;
stdDev=StandardDeviation/@output;
meanWithStdDev=Around@@@Transpose[{mean,stdDev}];
meanWithStdDev
]
(* Calculate mean and standard deviation for each BN layer's activation outputs: *)
{
meanWithStdDevBNLayer2,
meanWithStdDevBNLayer5,
meanWithStdDevBNLayer8
}=processLayer/@{
BNactivationOutputLayer2,
BNactivationOutputLayer5,
BNactivationOutputLayer8};

(* Calculate mean and standard deviation for each LogisticSigmoid layer's activation
outputs: *)
{
meanWithStdDevLogisticLayer3,
meanWithStdDevLogisticLayer6,
meanWithStdDevLogisticLayer9
}=processLayer/@{
LogisticactivationOutputLayer3,
LogisticactivationOutputLayer6,
LogisticactivationOutputLayer9};

(* Visualize mean BN activation values with error bars over training epochs: *)
ListLinePlot[
{
meanWithStdDevBNLayer2,
meanWithStdDevBNLayer5,
meanWithStdDevBNLayer8
},
PlotStyle->{Opacity[0.6],Opacity[0.6],Opacity[0.6]},
PlotRange->Full,
PlotLegends->

```

```

    "Mean of BN layer 2",
    "Mean of BN layer 5",
    "Mean of BN layer 8"
  },
  IntervalMarkers->"Bars",
  Frame->True,
  FrameLabel->{"Epochs", "Mean of BN Activation Values"},
  PlotLabel->"Means with Standard Deviations (vertical bars)\n of the BN activation
values"
]

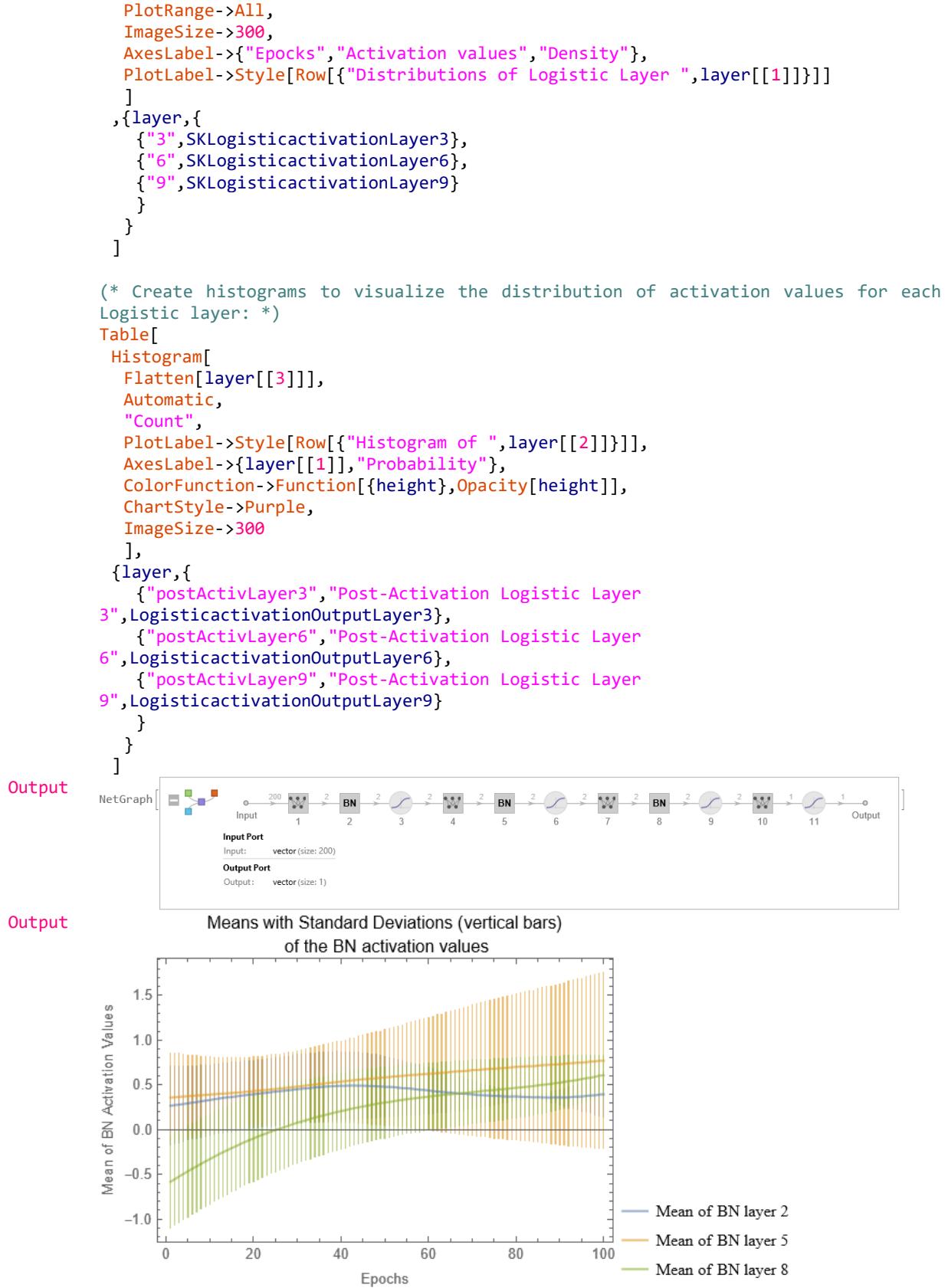
(* Visualize mean Logistic activation values with error bars over training epochs:
*)
ListLinePlot[
{
  meanWithStdDevLogisticLayer3,
  meanWithStdDevLogisticLayer6,
  meanWithStdDevLogisticLayer9
},
PlotStyle->{Opacity[0.6], Opacity[0.6], Opacity[0.6]},
PlotRange->Full,
PlotLegends->{
  "Mean of Logistic layer 3",
  "Mean of Logistic layer 6",
  "Mean of Logistic layer 9"
},
IntervalMarkers->"Bars",
Frame->True,
FrameLabel->{"Epochs", "Mean of Logistic Activation Values"},
PlotLabel->"Means with Standard Deviations (vertical bars)\n of the Logistic
activation values"
]

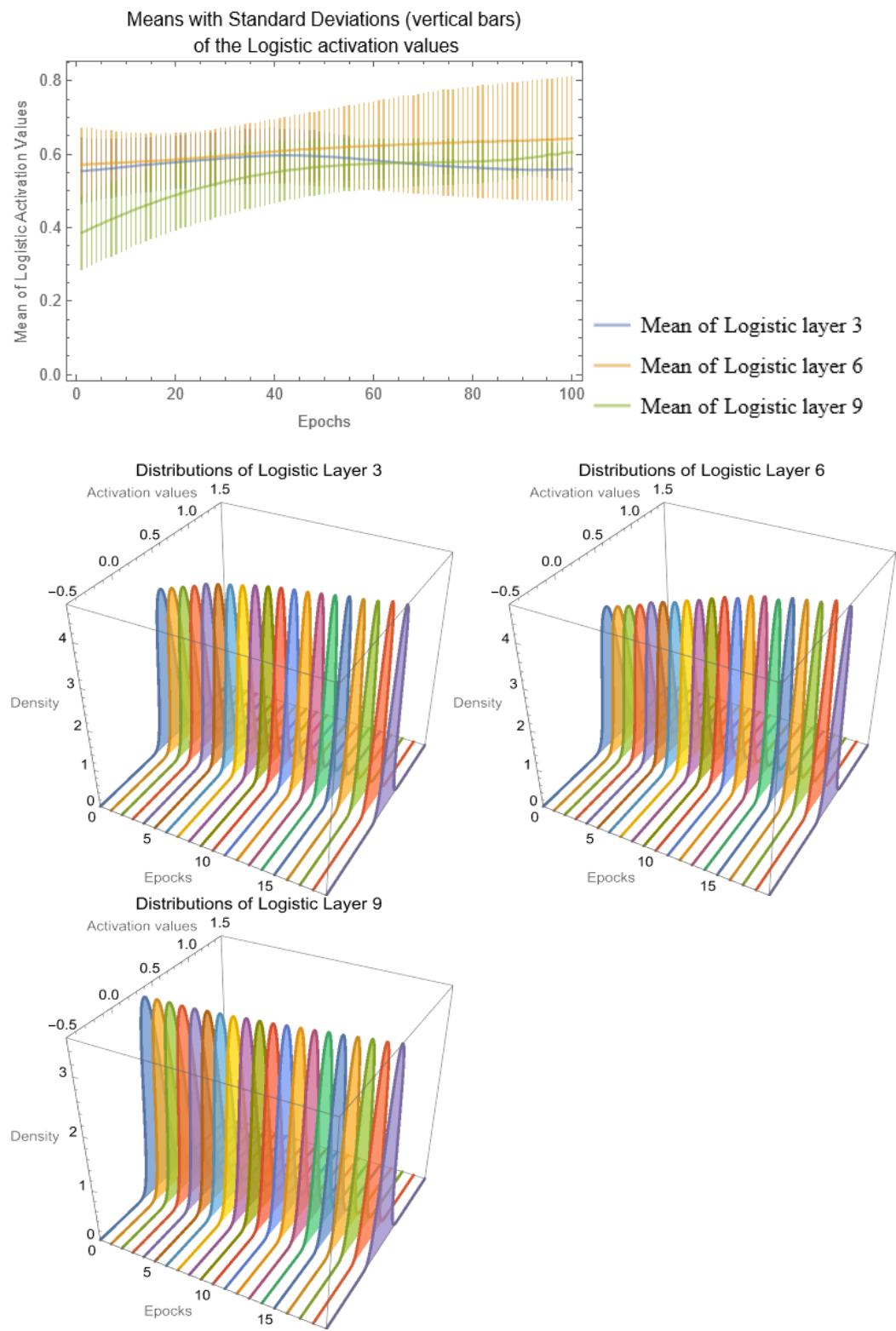
(* Smooth kernel density estimation: *)
createSmoothKernels[layer_]:=Module[
{sk},
  sk=SmoothKernelDistribution[#]&/@layer;
  Return[sk];
];

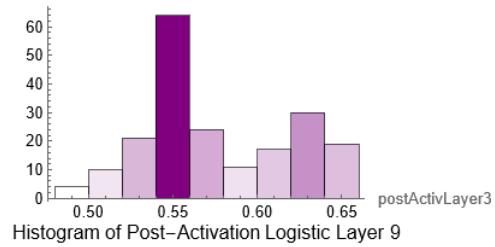
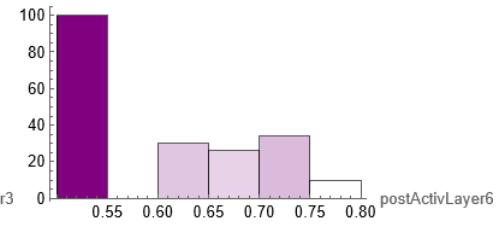
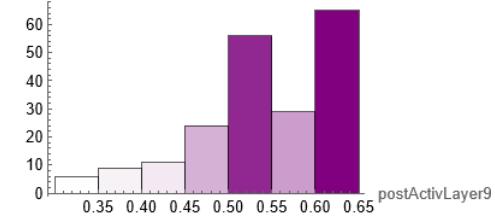
{
  SKBNactivationLayer2,SKBNactivationLayer5,
  SKBNactivationLayer8,SKLogisticactivationLayer3,
  SKLogisticactivationLayer6,SKLogisticactivationLayer9
}=createSmoothKernels/@{
  BNactivationOutputLayer2, BNactivationOutputLayer5,
  BNactivationOutputLayer8, LogisticactivationOutputLayer3,
  LogisticactivationOutputLayer6, LogisticactivationOutputLayer9};

(* Visualize the resulting densities over 20 epochs: *)
Table[
  ListLinePlot3D[
    Table[
      {(-1),x,PDF[layer[[2]][[#]],x]},
      {x,-0.5,1.5,.01}
    ]&/@Range[1,20],
    Filling->Bottom,
    FillingStyle->Opacity[0.75],
    Joined->True,
    BoxRatios->{1,1,1},
    PlotLabel->{"Epochs", "Density of Activation Values"},
    PlotRange->{{-1,20},{-0.5,1.5},{0,0.01}}
  ]
]

```



Output

OutputHistogram of Post-Activation Logistic Layer 3
ProbabilityHistogram of Post-Activation Logistic Layer 6
ProbabilityHistogram of Post-Activation Logistic Layer 9
Probability

CHAPTER 6

LEARNING RATE SCHEDULES AND GRADIENT DESCENT VARIANTS

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 5, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation.

In deep learning, the optimization process lies at the heart of training Neural Networks (NNs). Central to this optimization is the notion of the learning rate, a crucial hyperparameter that dictates the step size in the parameter space during optimization. Selecting an appropriate learning rate and its schedule significantly impacts the convergence speed and the final performance of the model. In this chapter, we delve into various learning rate schedules and adaptive algorithms that play pivotal roles in optimizing NNs. By understanding these techniques, practitioners can fine-tune their training processes and achieve better results in their machine-learning endeavors.

The chapter initiates an exploration of learning rate schedules [64-75], which entail predefined strategies for altering the learning rate throughout the training process. Learning rate schedules include step decay, inverse time decay, exponential decay, polynomial decay with warm restart, cyclical learning rate, stochastic gradient descent with warm restarts (cosine decay), exponential decay sine wave learning rate, Hessian-aware learning rate decay, etc., each with its own advantages and drawbacks. Understanding and appropriately implementing these schedules are crucial for achieving optimal convergence without encountering issues such as slow convergence or overshooting the minima.

Transitioning from learning rate schedules, we explore accelerated gradient descent [76-79], a variant of the traditional GD algorithm designed to speed up convergence. Two popular accelerated gradient descent algorithms are SGD with momentum and Nesterov accelerated gradient descent. In both cases, the momentum term helps the optimization algorithm to continue moving in the same direction or accelerate in the relevant direction, even if the gradient changes direction frequently or the surface of the loss function is highly irregular. Both methods help in smoothing out the updates, which is particularly useful when dealing with noisy or high-variance gradients common in SGD. The momentum term helps navigate through the irregularities of the loss surface more effectively, leading to a smoother and often faster path to the minimum. By leveraging past gradients, these methods can speed up convergence, reducing the time and computational resources needed for training. This leads to faster convergence and better overall performance in training NNs and other machine-learning models.

Following the discussion on accelerated gradient descent, we turn our attention to adaptive learning rate algorithms [80-93]. Adaptive learning rate algorithms adaptively adjust the learning rate during training based on past gradients, and other relevant metrics. These algorithms aim to strike a balance between the benefits of using large learning rates for fast convergence and the stability provided by smaller learning rates to prevent overshooting or oscillations. Popular examples include AdaGrad, RMSProp, AdaDelta, Adam, AdaMax, Nadam, and AMSGRAD, each with its own approach to adaptively scale the learning rates for individual model parameters.

Moreover, we explore second-order optimization methods [11,15,25,37]. Second-order optimization methods, such as Newton, Marquardt, and variants like conjugate gradient (Hestenes-Stiefel formula, Polak-Ribiere formula, Fletcher-Reeves formula), and quasi-Newton (rank one correction, DFP, and BFGS), etc., leverage information from the second derivatives of the loss function to guide the optimization process. By incorporating curvature information, these methods can converge faster and more accurately than first-order methods like GD. However, they often come

with higher computational costs and memory requirements due to the need to compute and store second-order derivatives or their approximations.

This chapter serves as a continuation of the foundational principles introduced in [Chapter 5](#). Here, we delve into the intricate landscape of learning rate schedules and adaptive algorithms, employing the computational power of Mathematica to elucidate their mechanisms and applications. Divided into two units, this chapter demystifies these fundamental concepts.

The first unit serves as a foundational exploration of learning rate schedules and adaptive optimization algorithms, leveraging the dynamic capabilities of Mathematica to provide intuitive insights. By building optimization algorithms from scratch, the reader gains a deeper understanding of the underlying principles. Through interactive manipulations, readers are guided step by step to comprehend the nuances of various algorithms. From elucidating traditional learning rate schedules such as step decay, inverse time decay, and exponential decay to unraveling sophisticated techniques like accelerated gradient descent and adaptive learning rate algorithms, this unit equips readers with a profound understanding of the mechanisms driving optimization processes. Moreover, Mathematica provides a comprehensive suite of built-in functions and tools for optimization, enabling users to implement, analyze, and visualize a diverse range of optimization problems.

In the second unit, the focus shifts towards practical applications, centering on the optimization of NNs using learning rate schedules and adaptive algorithms within the Mathematica environment. In this unit, we explore the powerful capabilities of Mathematica's `NetTrain` function, leveraging a range of optimization methods and suboptions to fine-tune the training process and enhance model performance. `NetTrain` serves as the cornerstone of NN training within Mathematica, providing a versatile and user-friendly interface for optimizing model parameters. At the heart of `NetTrain` lies a suite of optimization methods, including "`ADAM`", "`RMSProp`", "`SGD`", and "`SignSGD`", each offering unique strategies for navigating the high-dimensional parameter space of NNs. We will not only explore the selection and utilization of these optimization methods but also, we will go into the intricacies of their associated suboptions. These suboptions, including "`Momentum`", "`Beta1`", "`Beta2`", and "`Epsilon`", provide users with fine-grained control over the optimization process, allowing for nuanced adjustments to suit the specific requirements of their NN architectures and training objectives.

Furthermore, the "`LearningRateSchedule`" suboption allows you to specify a learning rate schedule, which can be a fixed learning rate, a schedule that decreases the learning rate over time (such as exponential decay or step decay), or any other custom schedule you define. Through comprehensive demonstrations and hands-on exercises, this unit empowers practitioners to harness the full potential of learning rate schedules and adaptive algorithms in enhancing the performance of NNs.

Together, these units shed light on the nuanced interconnections among learning rate schedules, adaptive algorithms, and NN optimization.

Unit 6.1

Understanding Learning Rate Schedules and Adaptive Algorithms with Mathematica

Mathematica Code 6.1

Step Decay

Input

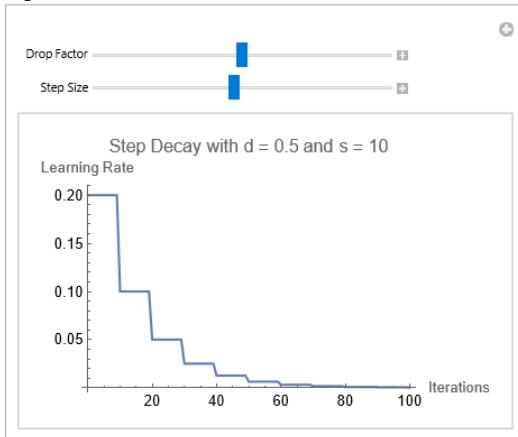
```
(* The code implements an interactive visualization tool using Mathematica's Manipulate function to explore the behavior of a step decay learning rate schedule. Users can adjust two parameters, the drop factor (d) and step size (s), via sliders to observe their impact on the learning rate decay. The code defines a step decay function that computes the learning rate based on the initial learning rate, drop factor, step size, and current iteration. Data points representing the learning rate at each iteration are generated and plotted dynamically: *)
Manipulate[
Module[
{data},
(* Define step decay function: *)

stepDecay[initialLearningRate_, dropFactor_, stepSize_, currentIteration_]:=initialLearningRate*dropFactor^Floor[currentIteration/stepSize];

(*Generate data for the selected dropFactor value (d) and step size (s)*)
data=Table[{i,stepDecay[initialLearningRate,d,s,i]},{i,0,100}];

(*Plot the learning rate with the selected values of d and s: *)
ListLinePlot[
data,
PlotRange->All,
PlotLabel->StringForm["Step Decay with d = `` and s = ``",d,s],
AxesLabel->{"Iterations","Learning Rate"},
ImageSize->300
]
],
{{d,0.5,"Drop Factor"},0.1,0.9,0.01},
{{s,10,"Step Size"},1,20,1},
Initialization:>{
(*Set parameters*)
initialLearningRate=0.2;
}
]
```

Output



Mathematica Code 6.2**Inverse Time Decay**

Input

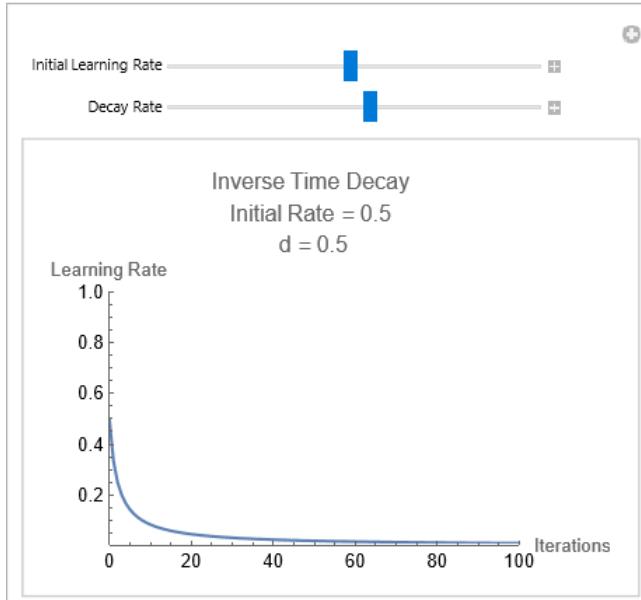
```
(* In this Manipulate interface, you can interactively choose different values for
the decay rate and observe the corresponding learning rate decay plot. Adjust the
decayRate slider to explore how the learning rate changes over iterations according
to the inverse time decay formula: *)

Manipulate[
Module[
{data},
(* Define inverse time decay function: *)

inverseTimeDecay[initialLearningRate_,decayRate_,currentIteration_]:=initialLearn
ingRate/(1+decayRate*currentIteration);

(* Generate data for the selected decay rate value (d): *)
data=Table[{i,inverseTimeDecay[initialLearningRate,d,i]},{i,0,100}];

(* Plot the learning rate with the selected values of d: *)
ListLinePlot[
data,
PlotRange->{{0,100},{0.002,1}},
PlotLabel->StringForm["Inverse Time Decay \nInitial Rate = `` \nd =
``",initialLearningRate,d],
AxesLabel->{"Iterations","Learning Rate"},
ImageSize->300
]
],
(* Manipulate parameters:*)
{{initialLearningRate,0.5,"Initial Learning Rate"},0.01,1,0.01},
{{d,0.5,"Decay Rate"},0.01,0.9,0.01}
]
```

Output**Mathematica Code 6.3****Exponential Decay**

Input

```
(* The code uses the standard exponential decay formula, where the learning rate
decreases exponentially with the increase in the step. Adjust the sliders for the
initial learning rate and decay rate to observe how the learning rate changes over
steps according to the standard exponential decay formula: *)


```

```

Manipulate[
Module[
{data},
(* Define standard exponential decay function: *)

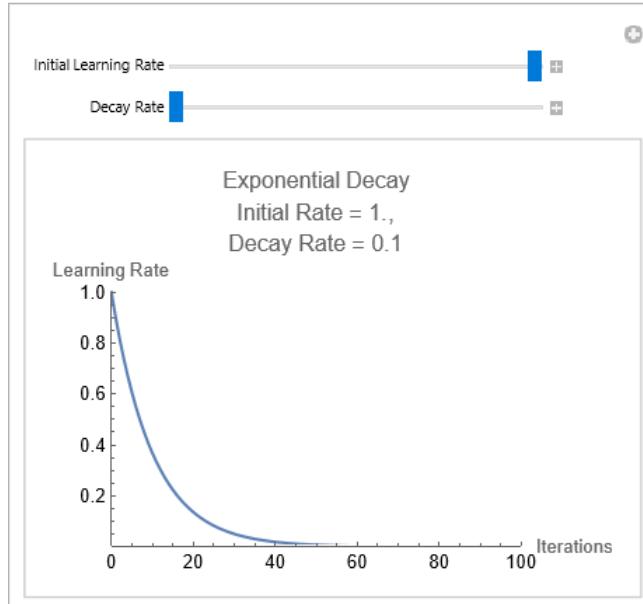
exponentialDecay[initialLearningRate_,decayRate_,currentIteration_]:=initialLearn-
ingRate*Exp[-decayRate*currentIteration];

(* Generate data for the learning rate with the selected parameters: *)
data=Table[{i,exponentialDecay[initialLearningRate,decayRate,i]},{i,0,100}];

(* Plot the learning rate with the selected parameters: *)
ListLinePlot[
data,
PlotRange->{{0,100},{0.002,1}},
PlotLabel->StringForm["Exponential Decay\nInitial Rate = ``,\nDecay Rate =
``",initialLearningRate,decayRate],
AxesLabel->{"Iterations","Learning Rate"},
ImageSize->300
]
],
(* Manipulate parameters: *)
{{initialLearningRate,0.5,"Initial Learning Rate"},0.01,1,0.01},
{{decayRate,0.5,"Decay Rate"},0.1,1,0.01}
]
]

```

Output

**Mathematica Code 6.4****Exponential Decay and Inverse Time Decay**

Input (* The code defines both exponential decay and inverse time decay functions and generates data for both methods. It then plots the learning rates over steps for comparison. Adjust the sliders for the initial learning rate, decay rate, and decay steps to observe how the learning rates evolve using both decay methods: *)

```

Manipulate[
Module[
{dataExp,dataInvTime},

(* Define exponential decay function: *)

```

```

exponentialDecay[initialLearningRate_, decayRate_, currentStep_]:=initialLearningRate
*Exp[-decayRate*currentStep];

(* Define inverse time decay function: *)

inverseTimeDecay[initialLearningRate_, decayRate_, currentIteration_]:=initialLearnin
gRate/(1+decayRate*currentIteration);

(* Generate data for exponential decay and inverse time decay: *)
dataExp=Table[{i,exponentialDecay[initialLearningRate,decayRate,i]},{i,0,100}];

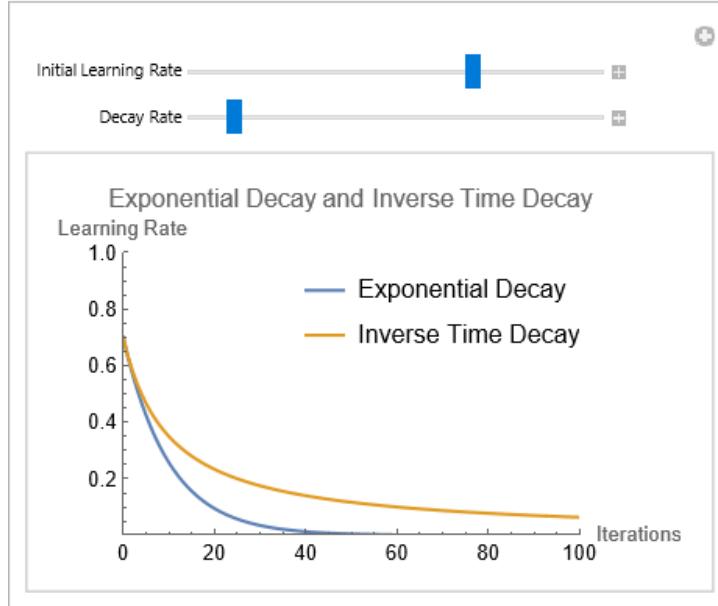
dataInvTime=Table[{i,inverseTimeDecay[initialLearningRate,decayRate,i]},{i,0,100}];

(* Plot the learning rates: *)
ListLinePlot[
{dataExp,dataInvTime},
PlotRange->{{0,100},{0.002,1}},
PlotLabel->"Exponential Decay and Inverse Time Decay",
PlotLegends->Placed[{"Exponential Decay","Inverse Time Decay"},{0.7,0.78}],
AxesLabel->{"Iterations","Learning Rate"},
ImageSize->300
],
];

(* Manipulate parameters: *)
{{initialLearningRate,0.7,"Initial Learning Rate"},0.01,1,0.01},
{{decayRate,0.1,"Decay Rate"},0.01,0.9,0.001}
]

```

Output

**Mathematica Code 6.5****Polynomial Decay**

Input

(* The code aims to create an interactive visualization tool for exploring polynomial decay in learning rate dynamics. Utilizing Mathematica's Manipulate function, users can dynamically adjust parameters such as the initial learning rate, power of the polynomial decay function, and total iterations via sliders. Within the module, a polynomial decay function is defined to compute the learning rate at each iteration based on the specified parameters. Data points representing

the learning rate over iterations are generated and plotted using `ListLinePlot`, with the plot displaying the decay curve: *)

```
Manipulate[
 Module[
 {data},
 (* Define polynomial decay function: *)

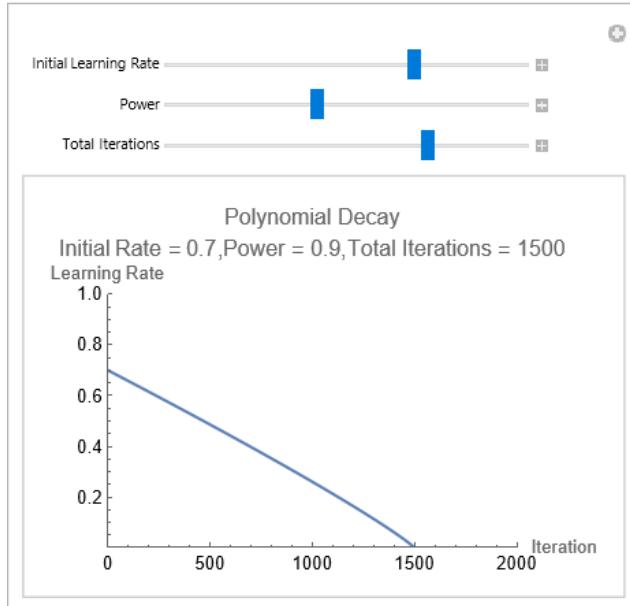
 polynomialDecay[initialLearningRate_, power_, totalIterations_, currentIteration_]:=
 initialLearningRate*(1-currentIteration/totalIterations)^power;

 (* Generate data for the polynomial decay: *)

 data=Table[{i,polynomialDecay[initialLearningRate,power,totalIterations,i]},{i,0,
 ,totalIterations}];

 (* Plot the polynomial decay: *)
 ListLinePlot[
 data,
 PlotRange->{{0,2000},{0.002,1}},
 PlotLabel->StringForm["Polynomial Decay\nInitial Rate = ``,Power = ``,Total
 Iterations = ``",initialLearningRate,power,totalIterations],
 AxesLabel->{"Iteration","Learning Rate"},
 ImageSize->300
 ],
 (* Manipulate parameters: *)
 {{initialLearningRate,0.7,"Initial Learning Rate"},0.01,1,0.01},
 {{power,0.9,"Power"},0.1,2,0.1},
 {{totalIterations,1500,"Total Iterations"},100,2000,100},
 Initialization:>{(*Set parameters*)}
 ]
```

Output

**Mathematica Code 6.6****Polynomial Decay with Warm Restarts**

Input (* The code aims to analyze and visualize polynomial decay with warm restarts, a technique often employed in training NN. It defines a function, `polynomialDecayWithRestart`, within a module to simulate learning rate with

periodic resets to prevent convergence to suboptimal solutions. Parameters such as the initial learning rate, powers representing different decay rates, total iterations, and warm restart fractions are defined to allow for customizable exploration of decay strategies. Data lists are generated for various decay powers using nested table commands, and these are plotted using `ListLinePlot`. The resulting plot displays the learning rate strategies, with each curve representing a different decay rate: *)

```

(* Define polynomial decay function with warm restarts: *)
polynomialDecayWithRestart[initialLearningRate_, power_, totalEpochs_, restartFractions_, currentIterations_]:=Module[
  {restartPoints, currentEpoch, currentLearningRate, currentIteration, totalEpoch},
  restartPoints=Round[restartFractions*totalEpochs];

  Piecewise[
  {
    {{currentLearningRate, totalEpoch}={initialLearningRate, restartPoints[[1]]},,
     restartPoints[[1]]>=currentIterations>=1},

    {{currentLearningRate, totalEpoch}={initialLearningRate*0.65, restartPoints[[2]]},,
     restartPoints[[2]]>=currentIterations>=restartPoints[[1]]+1},

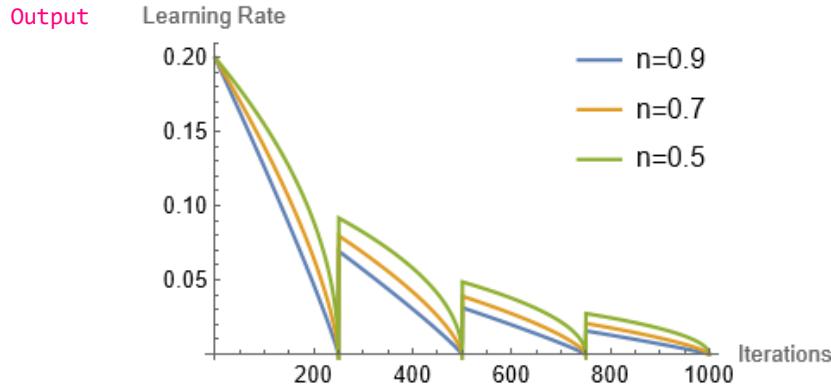
    {{currentLearningRate, totalEpoch}={initialLearningRate*0.65^2, restartPoints[[3]]},,
     ',
     restartPoints[[3]]>=currentIterations>=restartPoints[[2]]+1},
    {{currentLearningRate, totalEpoch}={initialLearningRate*0.65^3, totalEpochs},,
     totalEpochs>=currentIterations>=restartPoints[[3]]+1}
   }
  ];
  currentLearningRate*(1-currentIterations/totalEpoch)^power
];

(* Parameters: *)
initiallearningRate=0.2;
powers={0.9,0.7,0.5};
totalIterations=1000;
restartFractions={0.25,0.5,0.75};

(* Generate data for polynomial decay with warm restarts: *)
dataLists=Table[
  Table[
    {i,polynomialDecayWithRestart[initialLearningRate,power,totalIterations,restartFractions,i]},
    {i,0,totalIterations}
  ],
  {power,powers}
];

(* Plot the learning rate strategies: *)
ListLinePlot[
  dataLists,
  PlotRange->All,
  AxesLabel->{"Iterations", "Learning Rate"},
  PlotLegends->Placed[{"n=0.9", "n=0.7", "n=0.5"}, {0.85, 0.78}],
  ImageSize->300
]

```

**Mathematica Code 6.7****Cyclical Learning Rates**

Input

```
(* Cyclical Learning Rates (CLR) proves particularly valuable in overcoming challenges associated with saddle points in the loss landscape. The code presented below creates an interactive environment for exploring the impact of different parameters on the cyclical learning rate schedule. Key Parameters adjusted via Manipulate are base learning rate, the initial learning rate at the start of each cycle, max learning rate, the peak learning rate reached during a cycle, half cycle length, a parameter controlling how quickly the learning rate increases and decreases, and total iterations, the number of iterations to visualize. As users manipulate these parameters using sliders, the code dynamically recalculates and visualizes the corresponding cyclical learning rate schedule. The resulting plot provides insights into how changes in the learning rate parameters impact the training process, offering an intuitive understanding of CLR's adaptability and efficiency: *)
```

```
(* Define a function triangularLR to compute the learning rate based on the triangular learning rate schedule: *)
triangularLR[epoch_, stepsize_, baseLR_, maxLR_]:=Module[
{cycle,x,lr},
cycle=Floor[1+epoch/(2*stepsize)];
x=Abs[epoch/stepsize-2*cycle+1];

(* Compute the learning rate using the triangular function: *)
lr=baseLR+(maxLR-baseLR)*Max[0,(1-x)];

(* Return the computed learning rate: *)
lr
]

Manipulate[
Module[
{epochs,learningRates},

(* Generate data for plotting: *)
epochs=Range[1,totaliterations];

(* Calculate learning rates for each epoch using the triangularLR function: *)
learningRates=triangularLR[#,stepsize,baseLR,maxLR]&/@epochs;

(* Plotting with Manipulate: *)
ListLinePlot[
Transpose[{epochs,learningRates}],
PlotLabel->"Cyclical Learning Rate Schedule",
PlotRange->{{1,200},{0.005,0.4}}],
```

```

Joined->True,
Mesh->All,
MeshStyle->Directive[PointSize[0.01],Purple],
ImageSize->300
]
],
(* Manipulate parameters: *)
{{baseLR,0.1,"Base Learning Rate"},0.1,0.2,0.0001,Appearance->"Labeled"},  

{{maxLR,0.2,"Max Learning Rate"},0.2,0.3,0.001,Appearance->"Labeled"},  

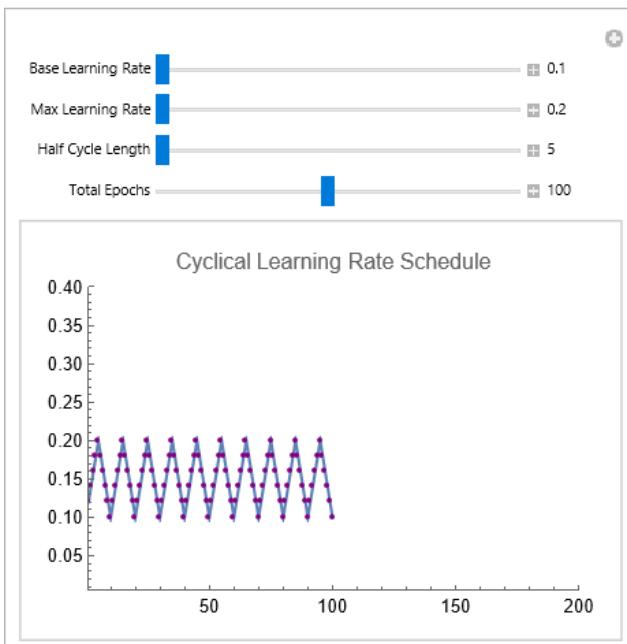
{{stepsize,5,"Half Cycle Length"},5,10,1,Appearance->"Labeled"},  

{{totaliterations,100,"Total Epochs"},10,200,20,Appearance->"Labeled"}  

]

```

Output



Mathematica Code 6.8

Cosine Annealing Learning Rate Scheduler

Input

```

(* The code implements a cosine annealing learning rate scheduler within a Manipulate environment, allowing users to interactively explore and visualize the learning rate schedule. It defines a function, CosineAnnealingLR, to compute learning rates based on parameters such as the minimum and maximum learning rates, total iterations in a cycle, and the current iteration number. The code generates learning rates over multiple cycles using the cosine annealing method and plots the learning rate schedule using ListLinePlot. Users can adjust hyperparameters such as the minimum and maximum learning rates, total iterations in a cycle, and the number of cycles through sliders to observe the corresponding changes in the learning rate schedule: *)

```

Manipulate

```

(*Function to calculate learning rate using Cosine Annealing*)
CosineAnnealingLR[minLR,maxLR,Ti,Tcur_]:=minLR+0.5 (maxLR-minLR)
(1+Cos[(Tcur/Ti) π]);

```

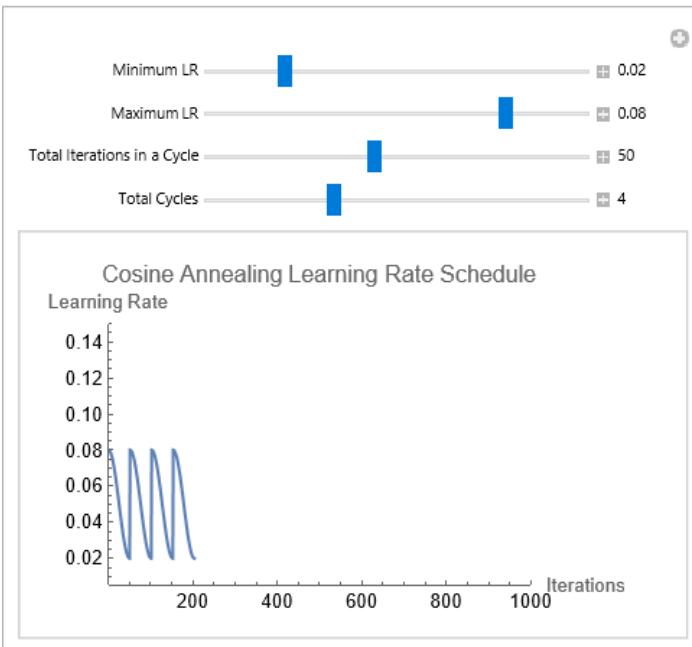
```

(*Generate learning rates over multiple cycles*)
cycles=Flatten[Table[Range[0,Ti],numCycles]];
learningRates=CosineAnnealingLR[minLR,maxLR,Ti,#]&/@cycles;
epochs=Range[1,Length[cycles]];
data=Transpose[{epochs,learningRates}];

```

```
(* Plot the learning rate schedule: *)
ListLinePlot[
  data,
  PlotRange->{{1,1000},{0.005,0.15}},
  AxesLabel->{"Iterations","Learning Rate"},
  PlotLabel->"Cosine Annealing Learning Rate Schedule",
  Joined->True,
  ImageSize->300
],
(* Manipulate parameters: *)
{{minLR,0.02,"Minimum LR"},0,0.1,0.0001,Appearance->"Labeled"},
{{maxLR,0.08,"Maximum LR"},0,0.1,0.0001,Appearance->"Labeled"},
{{Ti,50,"Total Iterations in a Cycle"},10,100,1,Appearance->"Labeled"},
{{numCycles,4,"Total Cycles"},1,10,1,Appearance->"Labeled"}
]
```

Output

**Mathematica Code 6.9****Stochastic Gradient Descent with Restarts**

Input (* The code constructs a learning rate schedule based on the stochastic gradient descent with restarts (SGDR) method utilizing cosine annealing. It initializes parameters such as the multiplier factor for cycle length, the initial cycle length, and the number of cycles. Subsequently, it calculates the cycle lengths and computes learning rates for each iteration within each cycle using the SGDRCosineLR function, which implements the cosine annealing formula. The resulting learning rate schedule is then visualized using ListLinePlot, displaying epochs against corresponding learning rates: *)

```
(* Define parameters for the learning rate schedule: *)
Tmult=2;
T[0]=10;
numofcycles=4;

(* Generate a list of cycle lengths: *)
Ti=Table[ T[i+1]=T[i]*Tmult;T[i+1],{i,0,numofcycles}];
TiValues=Flatten[PrependTo[Ti,{1,T[0]}]];
```

```
(* Define the SGDRCosineLR function to calculate learning rates: *)
ηmin=0.0;
ηmax=0.05;
SGDRCosineLR[ηmin_,ηmax_,Tcur_,Ti_]:=ηmin+1/2 (ηmax-ηmin) (1+Cos[Tcur/Ti π])

(* Compute learning rates for each iteration within each cycle: *)
lrValues=Flatten[Table[
  Table[
    SGDRCosineLR[ηmin,ηmax,t,TiValues[[j+1]]],
    {t,TiValues[[j]]-1,TiValues[[j+1]]-1}
  ],
  {j,1,Length[TiValues]-1}
]];

epochs=Range[0,Length[lrValues]-1];

(* Plot the learning rate schedule: *)
ListLinePlot[
  Transpose[{epochs,lrValues}],
  AxesLabel->{"Epoch","Learning Rate"},
  PlotLabel->"SGDR Learning Rate Schedule"
]
```

Output

Mathematica Code 6.10**Exponential Decay Sine Wave Learning Rate**

Input

```
(* The code implements a Manipulate interface in Mathematica, facilitating interactive exploration of a custom learning rate schedule. The schedule is defined by a function combining exponential decay and sine wave oscillation components. Users can adjust parameters such as the initial learning rate, decay rate, total iterations, oscillation parameter, and number of batches per epoch. These parameters influence the shape and behavior of the learning rate schedule, impacting the optimization process. The interface dynamically updates a plot showcasing the learning rate schedule, enabling users to visualize how changes in parameters affect the rate at which the algorithm adjusts its weights during training: *)
```

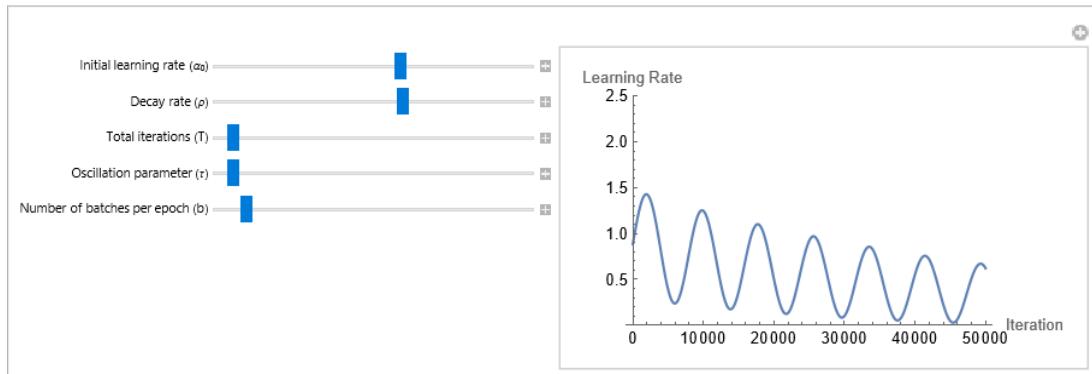
```
Manipulate[
Module[
{lrFunction,initialRate,decayRate,totalIterations,oscillationParam,batchesPerEpoch},
(*Define the custom learning rate function combining exponential decay and sine
wave
oscillation*)lrFunction[t_,initialRate_,decayRate_,totalIterations_,oscillationPara
m_,batchesPerEpoch_]:=initialRate Exp[-(decayRate t)/totalIterations]
```

```
(Sin[(oscillationParam t)/(2 π batchesPerEpoch)]+Exp[-(decayRate
t)/totalIterations]+0.5);

(*Set parameter values*)
initialRate=initialRate0;
decayRate=decayRate0;
totalIterations=totalIterations0;
oscillationParam=oscillationParam0;
batchesPerEpoch=batchesPerEpoch0;

(*Plot the learning rate schedule*)
Plot[
lrFunction[t,initialRate,decayRate,totalIterations,oscillationParam,batchesPerEpoch
],
{t,0,totalIterations},
AxesLabel->{"Iteration","Learning Rate"},
PlotRange->{Automatic,{0.005,2.5}},
ImageSize->300
],
{{initialRate0,0.59,"Initial learning rate ( $\alpha_0$ )"},0,1,0.01},
{{decayRate0,0.6,"Decay rate ( $\rho$ )"},0,1,0.01},
{{totalIterations0,50000,"Total iterations ( $T$ )"},100,1000000,1000},
{{oscillationParam0,0.05,"Oscillation parameter ( $\tau$ )"},0,1,0.01},
{{batchesPerEpoch0,10,"Number of batches per epoch ( $b$ )"},1,100,1},
ControlPlacement->Left
]
```

Output

**Mathematica Code 6.11****Hessian-Aware Learning Rate Schedule**

Input (* The code creates a Manipulate interface that allows users to visualize and interact with the Hessian-Aware Learning Rate schedule based on specific parameters. The code defines a piecewise learning rate function, allowing for different learning rates to be applied in various epochs of the training process. Users can adjust parameters such as the initial learning rate, decay factor, starting epoch for decay, ending epoch for decay, and total number of epochs. The interface dynamically updates the plotted learning rate decay curve based on the chosen parameter values, providing users with insights into how changes in these parameters affect the learning rate schedule during training: *)

```
Manipulate[
Module[
{lr,decayFunc,data},
(*Define the decay function*)
decayFunc[t_]:=decayFunc[t-1]*decayFactor^(1/(endEpoch-startEpoch));
```

```
(*Define the piecewise learning rate function*)
lr[t_,initialLR_,decayFactor_,startEpoch_,endEpoch_,totalEpochs_]:=Piecewise[
 {
 {initialLR,1<=t<startEpoch},
 {decayFunc[t],startEpoch<=t<=endEpoch},
 {decayFactor*initialLR,endEpoch<t<=totalEpochs}
 }
];
;

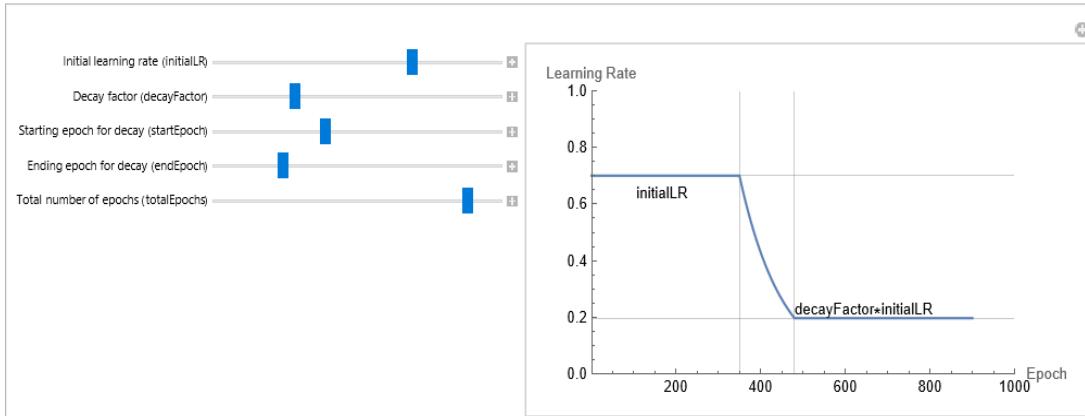
(*Set initial condition*)
decayFunc[startEpoch]=initialLR;

(*Generate data for plotting*)

data=Table[{t,lr[t,initialLR,decayFactor,startEpoch,endEpoch,totalEpochs]},{t,1,to
talEpochs}];

(*Plot the learning rate decay*)
ListLinePlot[
 data,
 PlotRange->{{1,1000},{0,1}},
 AxesLabel->{"Epoch","Learning Rate"},
 PlotRange->All,
 GridLines->{{startEpoch,endEpoch},{initialLR,decayFactor*initialLR}},
 Epilog->{
 Text["initialLR",{startEpoch,initialLR},{3,2}],
 Text["decayFactor*initialLR",{endEpoch,decayFactor*initialLR},{-1,-1}]
 }
 ],
(*Manipulate parameters*)
{{initialLR,0.7,"Initial learning rate (initialLR)"},0,1,0.01},
{{decayFactor,0.28,"Decay factor (decayFactor)"},0,1,0.01},
{{startEpoch,350,"Starting epoch for decay (startEpoch)"},1,totalEpochs-1,1},
{{endEpoch,480,"Ending epoch for decay (endEpoch)"},startEpoch+1,totalEpochs,1},
{{totalEpochs,900,"Total number of epochs (totalEpochs)"},10,1000,10},
ControlPlacement->Left
]
```

Output

**Mathematica Code 6.12****Gradient Descent with Momentum (1D)**

Input

(* The code showcases the implementation and visualization of gradient descent with momentum for optimizing the function $f(x)=x^2\sin(x)$. The code defines the function and its derivative, then employs a gradient descent algorithm with momentum to

iteratively update the value of x towards the minimum of the function. Through interactive controls, users can adjust parameters such as the initial point, number of steps, learning rate, and momentum parameter, allowing them to observe how different settings affect the optimization process, convergence speed, and trajectory: *)

```

(* Define the function to minimize: *)
targetFunction[x_]:=x^2*Sin[x]

(* Define the derivative of the function: *)
functionDerivative[x_]:=x^2 Cos[x]+2 x Sin[x]

(* Gradient Descent with Momentum: *)
gradientDescentWithMomentum[func_,funcDerivative_,initialPoint_,learningRate_,
momentumParameter_,iterations_]:=Module[
{x=initialPoint,v=0,path={initialPoint}},
For[
i=1,
i<=iterations,
i++,

(* Update velocity using momentum and current gradient: *)
v=momentumParameter*v-learningRate *funcDerivative[x];

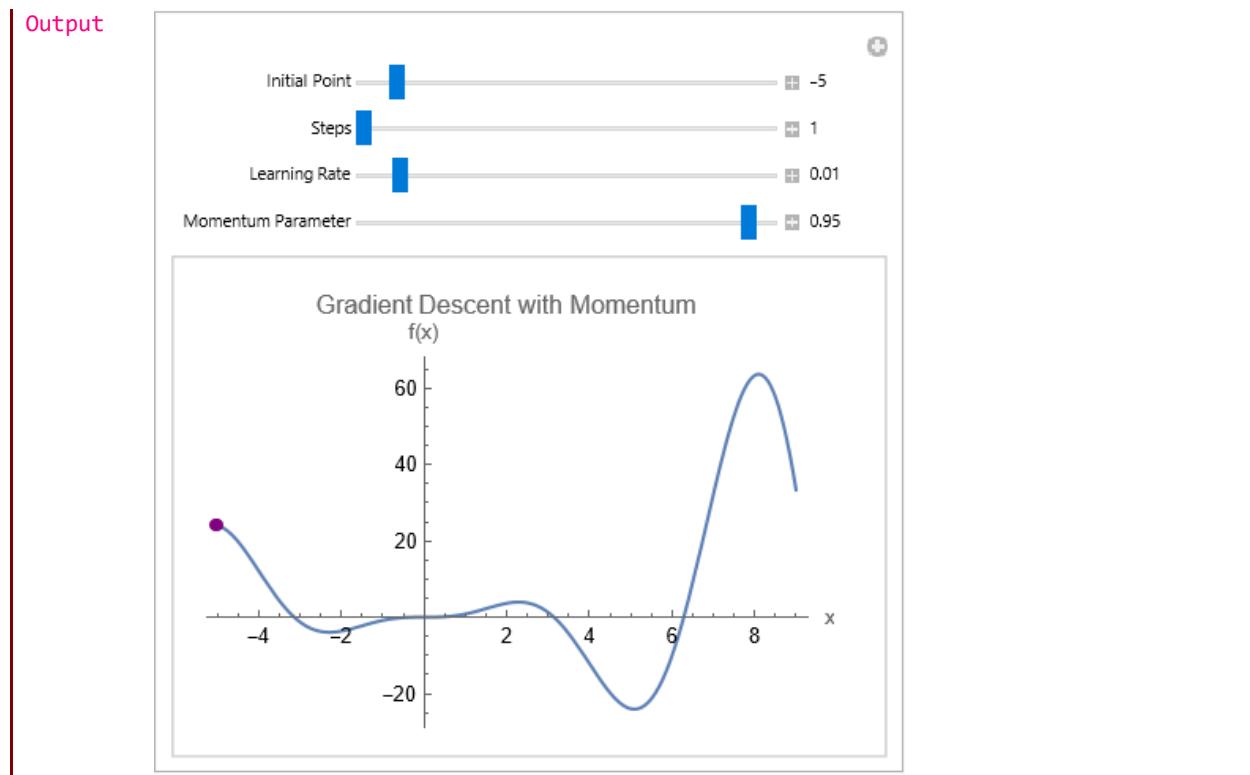
(*Update variable using velocity*)
x=x+v;

(* Append current variable to the optimization path: *)
AppendTo[path,x];
];
(* Return the optimization path: *)
Return[path];
]

(* Manipulate: *)
Manipulate[
(* Run gradient descent with momentum: *)
path=gradientDescentWithMomentum[targetFunction,functionDerivative,
initialPoint,learningRate,momentumParameter,steps];

(* Plot the path of the optimization: *)
Plot[
targetFunction[x],
{x,-5,9},
Epilog->{Purple,PointSize[0.02],Point[Thread[{path,targetFunction/@path}]]},
PlotLabel->"Gradient Descent with Momentum",
AxesLabel->{"x","f(x)"},
ImageSize->300
],


(* Manipulate Controls: *)
{{initialPoint,-5,"Initial Point"},-6,6,Appearance->"Labeled"},
{{steps,1,"Steps"},1,50,1,Appearance->"Labeled"},
{{learningRate,0.01,"Learning Rate"},0.001,0.1,Appearance->"Labeled"},
{{momentumParameter,0.95,"Momentum Parameter"},0.1,0.99,Appearance->"Labeled"}
]
]
```

**Mathematica Code 6.13****Gradient Descent with Momentum (2D)**

Input (* The code defines a target function $f(x,y)=x^2\sin(x)+y^2\sin(y)$ and its gradient, and implements gradient descent with momentum for optimization. It allows interactive manipulation of parameters such as the initial point, number of iterations, learning rate, and momentum parameter. The optimization process is visualized through a contour plot of the target function overlaid with the optimization path and step direction arrows. The primary goal is to demonstrate the convergence of the gradient descent algorithm with momentum towards the minimum point of the target function, providing an intuitive understanding of the optimization process: *)

```
(*Define the target function and its gradient*)
targetFunction[x_,y_]:=x^2*Sin[x]+y^2*Sin[y];

functionGradient[x_,y_]:=Grad[targetFunction[x,y],{x,y}];

(* Gradient Descent with Momentum: *)
gradientDescentWithMomentum[func_,grad_,initialPoint_,learningRate_,momentumParameter_,iterations_]:=Module[
  {x=initialPoint,v={0,0},path={initialPoint}},
  For[
    i=1,
    i<=iterations,
    i++,

    (* Update velocity using momentum and current gradient: *)
    v=momentumParameter*v-learningRate*grad@@x;

    (* Update variable using velocity: *)
    x=x+v;

    (* Append current variable to the optimization path: *)
    path=Append[path,x];
  ]
]
```

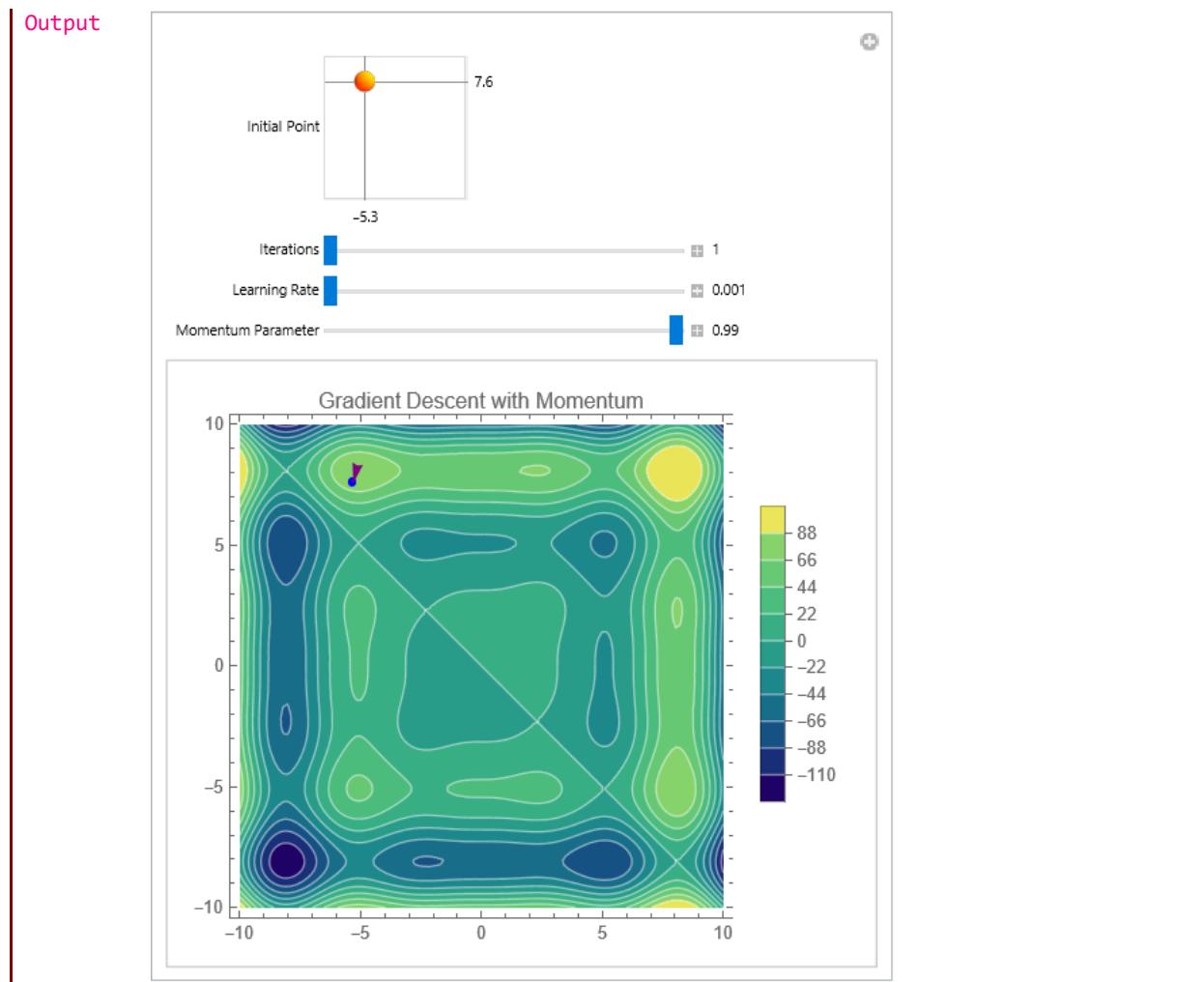
```
AppendTo[path,x];
];
(* Return the optimization path: *)
Return[path];
];

(* Run gradient descent with momentum and visualize: *)
Manipulate[
Module[
{path,minPoint},

(* Run gradient descent with momentum and find minimum point: *)
path=gradientDescentWithMomentum[targetFunction,functionGradient,initialPoint,learningRate,momentumParameter,iterations];

(* Calculate step direction arrows: *)
stepArrows=Arrow/@Partition[path,2,1];

(* Visualize: *)
Show[
ContourPlot[
targetFunction[x,y],
{x,-10,10},
{y,-10,10},
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->300,
PlotLegends->Automatic,
PlotLabel->"Gradient Descent with Momentum",
AspectRatio->Automatic
],
Graphics[
{
Blue,
PointSize[0.015],
Point[path],
Purple,
stepArrows
}
]
]
],
(* Manipulate Controls: *)
{{initialPoint,{-5.3,7.6},"Initial Point"},{-10,-10},{10,10},Appearance->"Labeled"},
{{iterations,1,"Iterations"},1,100,1,Appearance->"Labeled"},{{learningRate,0.001,"Learning Rate"},0.001,0.01,Appearance->"Labeled"},{{momentumParameter,0.99,"Momentum Parameter"},0.1,0.99,Appearance->"Labeled"}];
TrackedSymbols:>{initialPoint,iterations,learningRate,momentumParameter}
]
```

**Mathematica Code 6.14****Nesterov Accelerated Gradient Descent**

Input (* The code implements an interactive visualization of the Nesterov Accelerated Gradient Descent optimization algorithm. It defines an objective function $f(x_1, x_2) = \sin(x_1) + \cos(x_2)$ and its gradient, allowing users to adjust parameters such as learning rate, momentum, number of iterations, and initial guess through interactive controls. Within a module, the Nesterov Accelerated Gradient Descent algorithm is executed, updating parameters iteratively and storing the trajectory of parameter updates. The visualization consists of a contour plot of the objective function overlaid with the trajectory of parameter updates, enabling users to observe how the optimization algorithm progresses towards the function's minimum, with arrows indicating the direction of parameter updates at each iteration: *)

```
Manipulate[
Module[
{f,gradient,learningRate,momentum,numIterations,initialGuess,x,v,trajectory},

(* Define the objective function and its gradient: *)
f[x1_,x2_]:=Sin[x1]+Cos[x2];
gradient[x1_,x2_]={D[f[x1,x2],x1],D[f[x1,x2],x2]};

(* Parameters: *)
learningRate=lr;
momentum=m;
```

```
numIterations=n;
initialGuess={x0,y0};

(* Initialize variables: *)
x=initialGuess;
v={0,0};

(* Store trajectory for plotting: *)
trajectory={x};

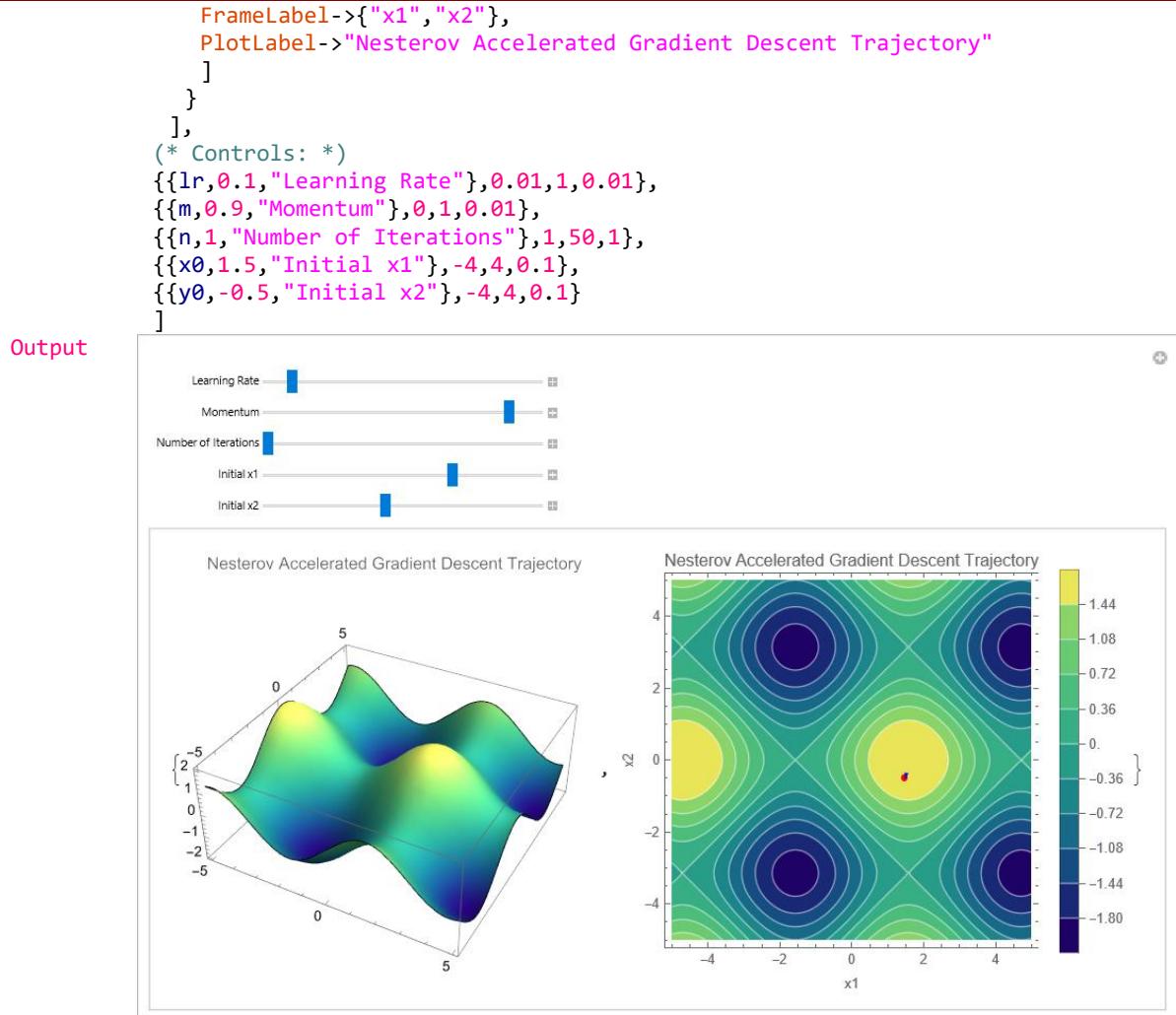
(* Nesterov Accelerated Gradient Descent: *)
Do[
  (* Compute gradient at lookahead parameters: *)
  lookaheadParams=x+momentum*v;
  grad=gradient@@lookaheadParams;

  (* Update momentum: *)
  v=momentum*v-learningRate*grad;

  (* Update parameters: *)
  x=x+v;

  (* Store parameter for plotting *)
  AppendTo[trajectory,x],
  {i,numIterations}
];

(*Generate contour plot with arrows*)
{
  Plot3D[
    f[x1,x2],
    {x1,-5,5},
    {x2,-5,5},
    PlotRange->All,
    Mesh->None,
    ImageSize->300,
    ColorFunction->"BlueGreenYellow",
    PlotLabel->"Nesterov Accelerated Gradient Descent Trajectory"],
  ContourPlot[
    f[x1,x2],
    {x1,-5,5},
    {x2,-5,5},
    PlotRange->All,
    Contours->10,
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    ImageSize->300,
    PlotLegends->Automatic,
    Epilog->{
      Red,
      PointSize[Medium],
      Point[trajectory],
      Blue,
      Arrowheads[0.02],
      Table[
        Arrow[{trajectory[[i]],trajectory[[i+1]]}],
        {i,Length[trajectory]-1}
      ]
    },
  ]
};
```

**Mathematica Code 6.15 AdaGrad Optimization**

Input

```

(* The code implements AdaGrad optimization to optimize the objective function  

sin(x)+cos(y). It defines functions to compute the gradient of the objective function,  

as well as the AdaGrad optimization procedure itself. Through a Manipulate interface,  

users can interactively set initial variable values, learning rate, epsilon, and the  

maximum number of iterations, while visualizing the trajectory of the optimization  

process on a contour plot of the objective function: *)

```

```

(* Define the objective function to optimize: *)
objectiveFunction[x_, y_] = Sin[x] + Cos[y];

(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_, y_] = {D[objectiveFunction[x, y], x], D[objectiveFunction[x, y], y]};

(* Define the AdaGrad optimization function: *)
AdaGradOptimization[gradient_, learningRate_, epsilon_, variables_, accumulatedGradients_] := Module[
{updatedVariables, updatedAccumulatedGradients},

```

(* Update accumulated gradients for each variable: *)

```

updatedAccumulatedGradients=MapThread[#1+#2^2&,{accumulatedGradients,gradient}];

(* Update variables using AdaGrad formula: *)
updatedVariables=MapThread[#1-(learningRate/Sqrt[#2+epsilon])
#3&,{variables,updatedAccumulatedGradients,gradient}];
{updatedVariables,updatedAccumulatedGradients}
];

(* Define AdaGrad optimization procedure: *)
OptimizeWithAdaGrad[initialVariables_,learningRate_,epsilon_,maxIterations_]:=Module[
{variables,accumulatedGradients,trajectory},
(* Initialize variables, accumulatedGradients and trajectory:*)
variables=initialVariables;
accumulatedGradients=ConstantArray[0,Length[initialVariables]];
trajectory={variables};

(* Perform AdaGrad optimization iteratively: *)
Do[
grad=gradientObjectiveFunction@@variables;

(* Update variables, and accumulatedGradients using AdaGrad: *)

{variables,accumulatedGradients}=AdaGradOptimization[grad,learningRate,epsilon,variables,accumulatedGradients];

(* Append current variables to trajectory: *)
AppendTo[trajectory,variables];

(* Iterate over the specified number of maxIterations: *)
{i,maxIterations}
];
trajectory
];

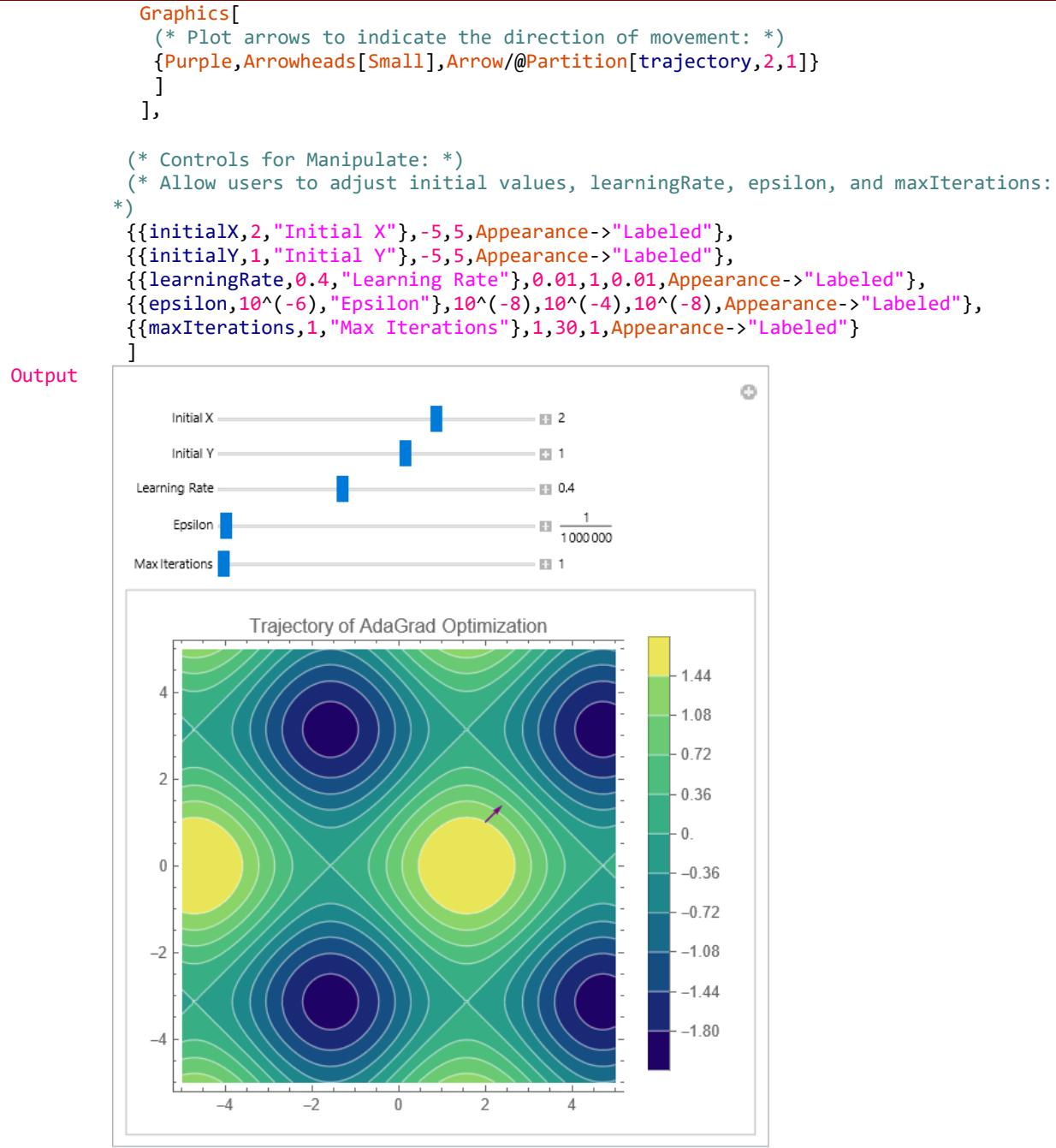
Manipulate[
(* Initial guess for variables: *)
initialVariables={initialX,initialY};

(* Perform AdaGrad optimization: *)

trajectory=OptimizeWithAdaGrad[initialVariables,learningRate,epsilon,maxIterations]
;

(* Plot the trajectory with arrows: *)
Show[
ContourPlot[
objectiveFunction[x,y],
{x,-5,5},
{y,-5,5},
PlotLegends->Automatic,
PlotLabel->"Trajectory of AdaGrad Optimization",
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->300
],

```

**Mathematica Code 6.16****RMSProp Optimization**

Input (* The code implements RMSProp optimization to minimize the objective function $\sin(x)+\cos(y)$. It defines the objective function and its gradient, implements the RMSProp optimization algorithm to iteratively update parameters based on gradients, and visualizes the optimization process using Manipulate. Users can adjust parameters such as initial values, learning rate, decay rate, epsilon, and maximum iterations to observe the trajectory of optimization on a contour plot of the objective function. This interactive visualization allows for an intuitive understanding of how different parameters affect the optimization process, facilitating experimentation and tuning for optimal performance: *)

(* Define the objective function to optimize: *)

```

objectiveFunction[x_,y_]:=Sin[x]+Cos[y];

(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_,y_]={D[objectiveFunction[x,y],x],D[objectiveFunction[x,y],y]};

(* Define the RMSProp optimization function: *)
RMSProp[grad_,lr_,decayRate_,epsilon_,vars_,varsSquaredGradients_]:=Module[
{updatedVars,updatedVarsSquaredGradients},

(* Update accumulated squared gradients for each variable: *)
updatedVarsSquaredGradients=decayRate*varsSquaredGradients+(1-
decayRate)*grad^2;

(* Update variables using AdaGrad formula: *)
updatedVars=vars-(lr/Sqrt[updatedVarsSquaredGradients+epsilon]) grad;
{updatedVars,updatedVarsSquaredGradients}
];

(* Define RMSProp optimization procedure: *)
OptimizeRMSProp[initialVars_,lr_,decayRate_,epsilon_,maxIterations_]:=Module[
{vars,varsSquaredGradients,trajectory},

(* Initialize variables, varsSquaredGradients and trajectory:*)
vars=initialVars;
varsSquaredGradients=ConstantArray[0,Length[initialVars]];
trajectory={vars};

(* Perform RMSProp optimization iteratively: *)
Do[
grad=gradientObjectiveFunction@@vars;

(* Update variables, and varsSquaredGradients using RMSProp: *)

{vars,varsSquaredGradients}=RMSProp[grad,lr,decayRate,epsilon,vars,varsSquaredGradi-
ents];

(* Append current variables to trajectory: *)
AppendTo[trajectory,vars];

(* Iterate over the specified number of maxIterations: *)
{i,maxIterations}
];
trajectory];

Manipulate[
(* Initial guess: *)
initialVars={initialX,initialY};

(* Perform RMSProp optimization: *)
trajectory=OptimizeRMSProp[initialVars,lr,decayRate,epsilon,maxIterations];

(* Plot the trajectory with arrows: *)
Show[
ContourPlot[
objectiveFunction[x,y],
{x,-5,5},
{y,-5,5},
PlotLegends->Automatic,
PlotLabel->"Trajectory of RMSProp Optimization",
];
]
];

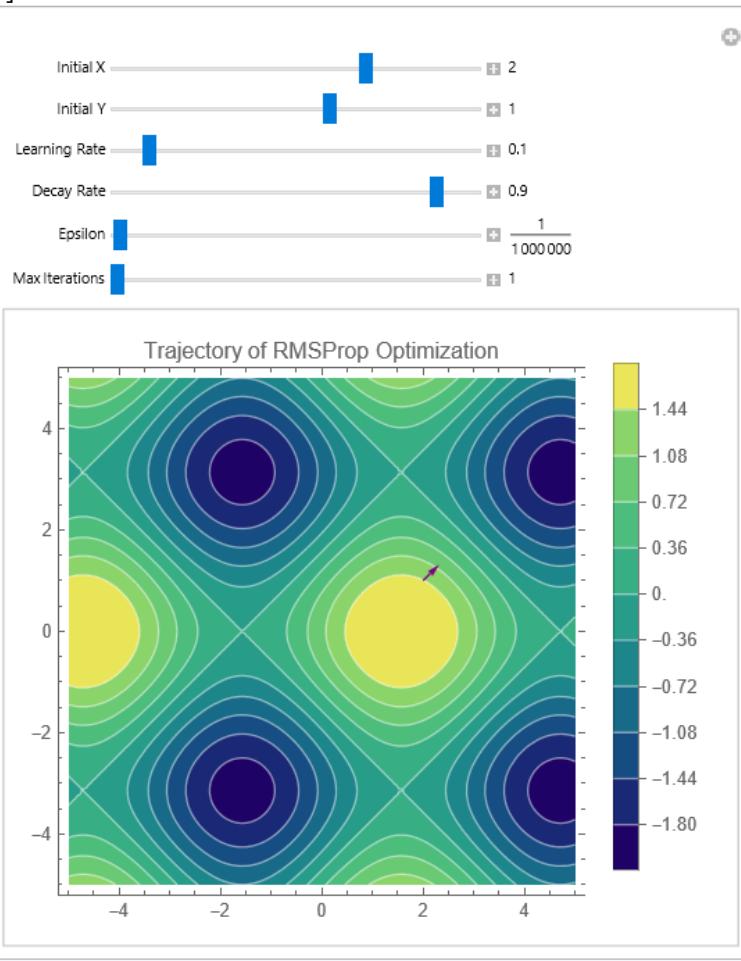
```

```

Contours->10,
ContourStyle->\{White\},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->300
],
Graphics[
(* Plot arrows to indicate the direction of movement: *)
{Purple,Arrowheads[Small],Arrow/@Partition[trajectory,2,1]}
]
],
(* Controls for Manipulate: *)
(* Allow users to adjust initial values, lr, decayRate, epsilon, and
maxIterations:*)
{{initialX,2,"Initial X"},-5,5,Appearance->"Labeled"},
{{initialY,1,"Initial Y"},-5,5,Appearance->"Labeled"},
{{lr,0.1,"Learning Rate"},0.01,1,0.01,Appearance->"Labeled"},
{{decayRate,0.9,"Decay Rate"},0.1,0.99,0.01,Appearance->"Labeled"},
{{epsilon,10^(-6),"Epsilon"},10^(-8),10^(-4),10^(-8),Appearance->"Labeled"},
{{maxIterations,1,"Max Iterations"},1,100,1,Appearance->"Labeled"}
]
]

```

Output

**Mathematica Code 6.17****AdaDelta Optimization**

Input (* The code aims to optimize the objective function $\sin(x)+\cos(y)$ using the AdaDelta optimization algorithm. It begins by defining the objective function and its gradient, followed by the implementation of the AdaDelta optimization method, which iteratively

updates parameters based on gradients and accumulations. Through the manipulation interface, users can adjust initial values, the decay rate (ρ), epsilon, and maximum iterations to observe the trajectory of optimization on a contour plot of the objective function. The interactive visualization aids in understanding the behavior of the optimization algorithm under different parameter settings, offering insights into its convergence process: *)

```

(* Define the objective function to optimize: *)
objectiveFunction[x_,y_]:=Sin[x]+Cos[y];
(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_,y_]={D[objectiveFunction[x,y],x],D[objectiveFunction[x,y],y]};

(* Define the AdaDelta function: *)
AdaDelta[grad_,rho_,epsilon_,vars_,accumulators_,deltas_]:=Module[
{updatedVars,updatedAccumulators,updatedDeltas,dx},
(* Update accumulated squared gradients for each variable: *)
updatedAccumulators=rho*accumulators+(1-rho)*grad^2;
(* Compute the change in variables using AdaDelta formula: *)
dx=(Sqrt[deltas+epsilon])/Sqrt[updatedAccumulators+epsilon])*grad;

(* Update accumulated squared changes in variables: *)
updatedDeltas=rho*deltas+(1-rho)*dx^2;

(* Update variables using the computed change: *)
updatedVars=vars-dx;
{updatedVars,updatedAccumulators,updatedDeltas}
];

(* Define AdaDelta optimization: *)
OptimizeAdaDelta[initialVars_,rho_,epsilon_,maxIterations_]:=Module[
{vars,accumulators,deltas,trajectory},
(* Initialize variables, accumulators, deltas, and trajectory: *)
vars=initialVars;
accumulators=ConstantArray[0,Length[initialVars]];
deltas=ConstantArray[0,Length[initialVars]];
trajectory={vars};

(* Perform AdaDelta optimization iteratively: *)
Do[
grad=gradientObjectiveFunction@@vars;

(* Update variables, accumulators, and deltas using AdaDelta: *)

{vars,accumulators,deltas}=AdaDelta[grad,rho,epsilon,vars,accumulators,deltas];

(* Append current variables to trajectory: *)
AppendTo[trajectory,vars];

(* Iterate over the specified number of maxIterations: *)
{i,maxIterations}
];
trajectory];

Manipulate[
(* Initial guess: *)
initialVars={initialX,initialY};
(* Perform AdaDelta optimization: *)

```

```

trajectory=OptimizeAdaDelta[initialVars,rho,epsilon,maxIterations];
(* Plot the trajectory with arrows: *)
Show[
ContourPlot[
objectiveFunction[x,y],
{x,-6,6},
{y,-6,6},
PlotLegends->Automatic,
PlotLabel->"Trajectory of AdaDelta Optimization",
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->300
],
Graphics[
(* Plot arrows to indicate the direction of movement: *)
{Purple,Arrowheads[Small],Arrow/@Partition[trajectory,2,1]}
]
],
(* Controls for Manipulate: *)
(* Allow users to adjust initial values, rho, epsilon, and maxIterations: *)
{{initialX,2,"Initial X"},-5,5,Appearance->"Labeled"},  

{{initialY,1,"Initial Y"},-5,5,Appearance->"Labeled"},  

{{rho,0.9,"Rho"},0.1,0.99,0.01,Appearance->"Labeled"},  

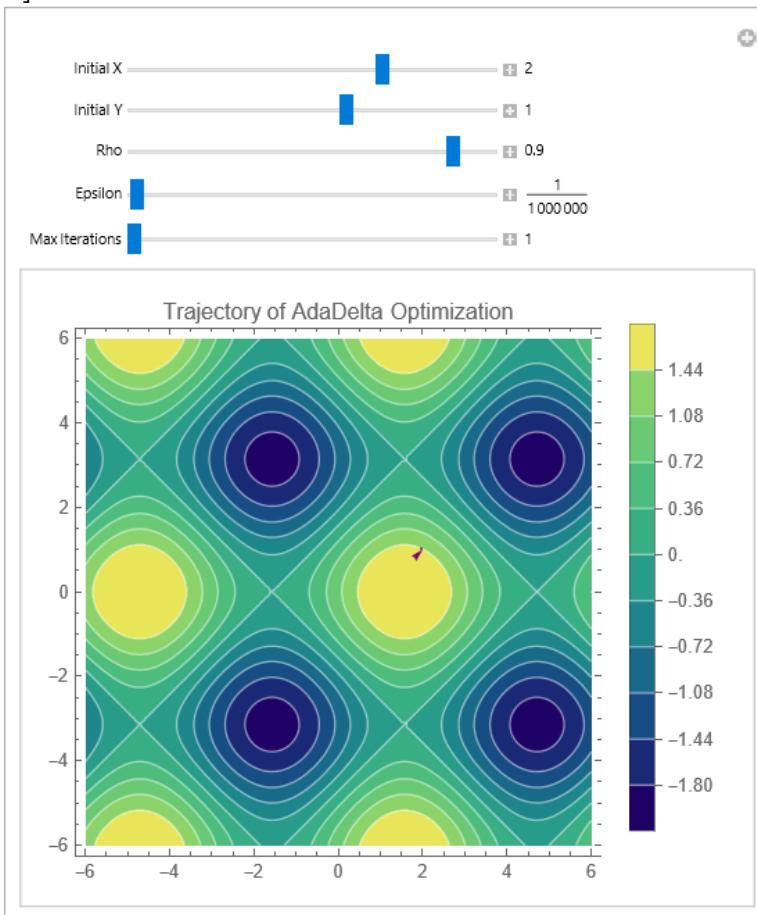
{{epsilon,10^(-6),"Epsilon"},10^(-8),10^(-4),10^(-8),Appearance->"Labeled"},  

{{maxIterations,1,"Max Iterations"},1,600,1,Appearance->"Labeled"}  

]
]

```

Output



Mathematica Code 6.18**Adam Optimization**

Input

```

(* The code aims to minimize the objective function sin(x)+cos(y) using the Adam optimization algorithm. It begins by defining the objective function and its gradient. The Adam algorithm is then implemented, updating parameters iteratively based on gradients and moving averages of past gradients and squared gradients. Through an interactive visualization using Manipulate, users can adjust parameters such as initial values, learning rate, beta1, beta2, epsilon, and maximum iterations to observe the trajectory of optimization on a contour plot of the objective function. This visualization facilitates an intuitive understanding of how different parameters influence the optimization process and aids in tuning for optimal performance: *)

(* Define the objective function to optimize: *)
objectiveFunction[x_,y_]:=Sin[x]+Cos[y];
(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_,y_]:={D[objectiveFunction[x,y],x],D[objectiveFunction[x,y],y]};
(* Define the Adam function: *)
Adam[grad_,lr_,beta1_,beta2_,epsilon_,vars_,m_,v_,t_]:=Module[
{mnew,vnew,mhat,vhat,varsnew},
(* Update moving averages of gradients and squared gradients: *)
mnew=beta1*m+(1-beta1)*grad;
vnew=beta2*v+(1-beta2)*grad^2;
(* Bias-corrected first and second moment estimates: *)
mhat=mnew/(1-beta1^t);
vhat=vnew/(1-beta2^t);
(* Update variables using Adam update rule: *)
varsnew=vars-(lr/Sqrt[vhat+epsilon]) mhat;
{varsnew,mnew,vnew}
];

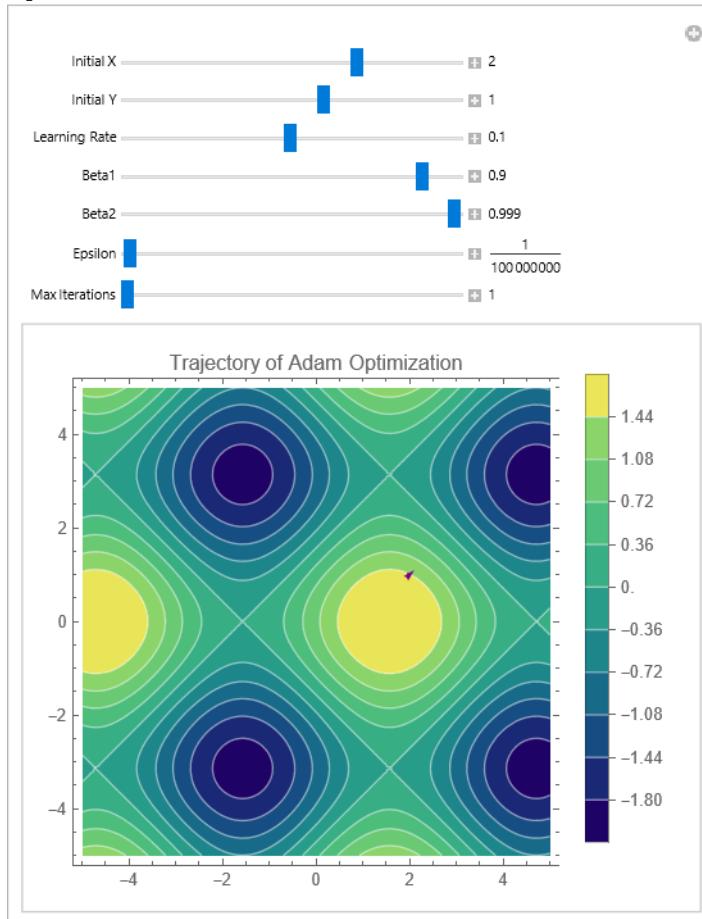
(* Define Adam optimization: *)
OptimizeAdam[initialVars_,lr_,beta1_,beta2_,epsilon_,maxIterations_]:=Module[
{vars,m,v,t,trajectory},
(* Initialize variables, m, v, t, and trajectory: *)
vars=initialVars;
m=ConstantArray[0,Length[initialVars]];
v=ConstantArray[0,Length[initialVars]];
t=0;
trajectory={vars};

(* Perform Adam optimization iteratively: *)
Do[
grad=gradientObjectiveFunction@@vars;
t++;
(* Update variables, m, and v using Adam: *)
{vars,m,v}=Adam[grad,lr,beta1,beta2,epsilon,vars,m,v,t];
(* Append current variables to trajectory: *)
AppendTo[trajectory,vars];
(* Iterate over the specified number of maxIterations: *)
{i,maxIterations}
];
trajectory];
Manipulate[
(* Initial guess: *)
initialVars={initialX,initialY};
(* Perform Adam optimization: *)
trajectory=OptimizeAdam[initialVars,lr,beta1,beta2,epsilon,maxIterations];
]

```

```
(* Plot the trajectory with arrows: *)
Show[
  ContourPlot[
    objectiveFunction[x,y],
    {x,-5,5},
    {y,-5,5},
    PlotLegends->Automatic,
    PlotLabel->"Trajectory of Adam Optimization",
    Contours->10,
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    ImageSize->300
  ],
  Graphics[
    (* Plot arrows to indicate the direction of movement: *)
    {Purple, Arrowheads[Small], Arrow/@Partition[trajectory, 2, 1]}
  ]
],
(* Controls for Manipulate: *)
{{initialX, 2, "Initial X"}, -5, 5, Appearance -> "Labeled"}, 
{{initialY, 1, "Initial Y"}, -5, 5, Appearance -> "Labeled"}, 
{{lr, 0.1, "Learning Rate"}, 0.001, 0.2, 0.001, Appearance -> "Labeled"}, 
{{beta1, 0.9, "Beta1"}, 0, 1, 0.01, Appearance -> "Labeled"}, 
{{beta2, 0.999, "Beta2"}, 0, 1, 0.001, Appearance -> "Labeled"}, 
{{epsilon, 10^(-8), "Epsilon"}, 10^(-10), 10^(-6), 10^(-10), Appearance -> "Labeled"}, 
{{maxIterations, 1, "Max Iterations"}, 1, 100, 1, Appearance -> "Labeled"}]
```

Output



Mathematica Code 6.19**AdaMax optimization**

Input

```

(* The code aims to optimize the objective function sin(x)+cos(y) using the AdaMax
optimization algorithm. It begins by defining the objective function and its gradient,
essential for the optimization process. The AdaMax algorithm, characterized by its
unique update rules for moving averages of gradients, is then implemented to
iteratively update parameters such as learning rate, beta1, beta2, and epsilon to
minimize the objective function. Through an interactive visualization using
Manipulate, users can adjust parameters and observe the trajectory of optimization
on a contour plot of the objective function, providing insights into the optimization
process and facilitating parameter tuning for optimal performance: *)
(* Define the objective function to optimize: *)
objectiveFunction[x_,y_]:=Sin[x]+Cos[y];

(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_,y_]:={D[objectiveFunction[x,y],x],D[objectiveFunction[x,y],y]};

(* Define the AdaMax function: *)
AdaMax[grad_,lr_,beta1_,beta2_,epsilon_,vars_,m_,u_,t_]:=Module[
{mnew,unew,varsnew},

(* Update moving averages of gradients and max of gradients: *)
mnew=beta1*m+(1-beta1)*grad;
unew=Max[beta2*u,Abs[grad]];

(* Update variables using AdaMax update rule: *)
varsnew=vars-(lr/(1-beta1^t))*(mnew/unew);
{varsnew,mnew,unew}];

(* Define AdaMax optimization: *)
OptimizeAdaMax[initialVars_,lr_,beta1_,beta2_,epsilon_,maxIterations_]:=Module[
{vars,m,u,t,trajectory},

(* Initialize variables, m, u, t, and trajectory: *)
vars=initialVars;
m=ConstantArray[0,Length[initialVars]];
u=ConstantArray[0,Length[initialVars]];
t=0;
trajectory={vars};
Do[
  grad=gradientObjectiveFunction@@vars;
  t++;
  (* Update variables, m, and u using AdaMax update rule: *)
  {vars,m,u}=AdaMax[grad,lr,beta1,beta2,epsilon,vars,m,u,t];
  (* Append current variables to trajectory: *)
  AppendTo[trajectory,vars];
  (* Iterate over the specified number of maxIterations: *)
  {i,maxIterations}
];
trajectory];

Manipulate[
(* Initial guess: *)
initialVars={initialX,initialY};

(* Perform AdaMax optimization: *)
trajectory=OptimizeAdaMax[initialVars,lr,beta1,beta2,epsilon,maxIterations];

(* Plot the trajectory with arrows: *)

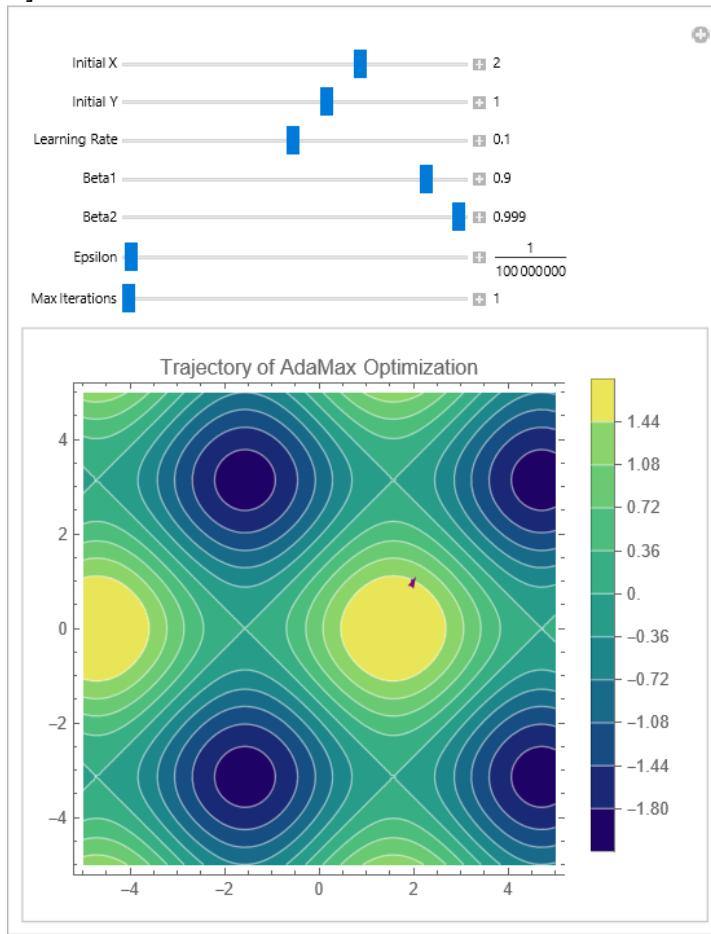
```

```

Show[
ContourPlot[
  objectiveFunction[x,y],
  {x,-5,5},
  {y,-5,5},
  PlotLegends->Automatic,
  PlotLabel->"Trajectory of AdaMax Optimization",
  Contours->10,
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  ImageSize->300
],
Graphics[
  (* Plot arrows to indicate the direction of movement: *)
  {Purple,Arrowheads[Small],Arrow/@Partition[trajectory,2,1]}]
],
(* Controls for Manipulate: *)
{{initialX,2,"Initial X"},-5,5,Appearance->"Labeled"},
{{initialY,1,"Initial Y"},-5,5,Appearance->"Labeled"},
{{lr,0.1,"Learning Rate"},0.001,0.2,0.001,Appearance->"Labeled"},
{{beta1,0.9,"Beta1"},0,1,0.01,Appearance->"Labeled"},
{{beta2,0.999,"Beta2"},0,1,0.001,Appearance->"Labeled"},
{{epsilon,10^(-8),"Epsilon"},10^(-10),10^(-6),10^(-10),Appearance->"Labeled"},
{{maxIterations,1,"Max Iterations"},1,100,1,Appearance->"Labeled"}
]
]

```

Output



Mathematica Code 6.20**AMSGrad Optimization**

Input

```

(* The code aims to optimize the objective function sin(x)+cos(y) using the AMSGard
optimization algorithm, which addresses limitations of other algorithms like Adam by
maintaining a maximum of past squared gradients. It begins by defining the objective
function and its gradient, essential components for the optimization process. The
AMSGard algorithm is then implemented to iteratively update parameters such as
learning rate, beta1, beta2, and epsilon, with the goal of minimizing the objective
function. Through an interactive visualization using Manipulate, users can explore
the trajectory of optimization on a contour plot of the objective function, gaining
insights into the optimization process and the impact of different parameter settings
on convergence behavior: *)

(* Define the objective function to optimize: *)
objectiveFunction[x_,y_]:=Sin[x]+Cos[y];

(* Define the gradient of the objective function: *)
gradientObjectiveFunction[x_,y_]={D[objectiveFunction[x,y],x],D[objectiveFunction[x,
,y],y]};

(* Define the AMSGard function: *)
AMSGard[grad_,lr_,beta1_,beta2_,epsilon_,vars_,m_,v_,vhat_]:=Module[
{mnew,vnew,vhatnew,varsnew},

(* Update moving averages of gradients, squared gradients, and vhat: *)
mnew=beta1*m+(1-beta1)*grad;
vnew=beta2*v+(1-beta2)*grad^2;
vhatnew=Max[vhat,vnew];
(* Update variables using AMSGard update rule: *)
varsnew=vars-(lr/Sqrt[vhatnew+epsilon]) mnew;
{varsnew,mnew,vnew,vhatnew}
];

(* Define AMSGard optimization: *)
OptimizeAMSGard[initialVars_,lr_,beta1_,beta2_,epsilon_,maxIterations_]:=Module[
{vars,m,v,vhat,trajectory},

(* Initialize variables, m, v, vhat, and trajectory: *)
vars=initialVars;
m=ConstantArray[0,Length[initialVars]];
v=ConstantArray[0,Length[initialVars]];
vhat=0;
trajectory={vars};

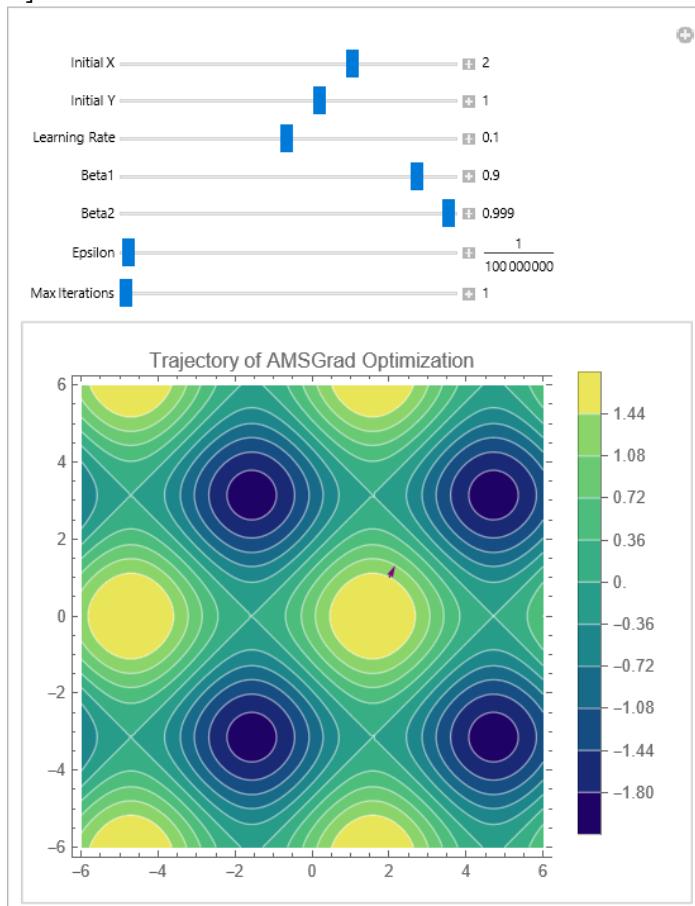
(* Perform AMSGard optimization iteratively: *)
Do[
grad=gradientObjectiveFunction@@vars;
(* Update variables, m, v and vhat using AMSGard update rule: *)
{vars,m,v,vhat}=AMSGard[grad,lr,beta1,beta2,epsilon,vars,m,v,vhat];
(* Append current variables to trajectory: *)
AppendTo[trajectory,vars];
(* Iterate over the specified number of maxIterations: *)
{i,maxIterations}
];
trajectory];

Manipulate[
(* Initial guess: *)
initialVars={initialX,initialY};
(* Perform AMSGard optimization: *)
trajectory=OptimizeAMSGard[initialVars,lr,beta1,beta2,epsilon,maxIterations];

```

```
(* Plot the trajectory with arrows: *)
Show[
  ContourPlot[
    objectiveFunction[x,y],
    {x,-6,6},
    {y,-6,6},
    PlotLegends->Automatic,
    PlotLabel->"Trajectory of AMSGrad Optimization",
    Contours->10,
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    ImageSize->300
  ],
  Graphics[
    (* Plot arrows to indicate the direction of movement: *)
    {Purple, Arrowheads[Small], Arrow/@Partition[trajectory, 2, 1]}
  ]
],
(* Controls for Manipulate: *)
{{initialX, 2, "Initial X"}, -5, 5, Appearance->"Labeled"}, 
{{initialY, 1, "Initial Y"}, -5, 5, Appearance->"Labeled"}, 
{{lr, 0.1, "Learning Rate"}, 0.001, 0.2, 0.001, Appearance->"Labeled"}, 
{{beta1, 0.9, "Beta1"}, 0, 1, 0.01, Appearance->"Labeled"}, 
{{beta2, 0.999, "Beta2"}, 0, 1, 0.001, Appearance->"Labeled"}, 
{{epsilon, 10^(-8), "Epsilon"}, 10^(-10), 10^(-6), 10^(-10), Appearance->"Labeled"}, 
{{maxIterations, 1, "Max Iterations"}, 1, 100, 1, Appearance->"Labeled"}]
```

Output



Mathematica Code 6.21**Newton Optimization Method**

Input

```

(* The code implements Newton's optimization method for a two-dimensional objective
function. It defines the objective function as a combination of trigonometric and
polynomial terms and computes its gradient and Hessian matrix. Newton's optimization
algorithm is then implemented, where the algorithm iteratively updates the current
solution based on the gradient and the curvature of the objective function. The code
includes a Manipulate interface allowing users to specify the initial guess,
tolerance level for convergence, and maximum number of iterations interactively. It
visualizes the optimization process with a contour plot of the objective function
and arrows indicating the trajectory of the optimization process: *)

(* Define the objective function: *)
objectiveFunction[x_,y_]:=Sin[x]+Cos[y]+x^4+y^4+x*y;

(* Define the gradient of the objective function: *)
objectiveGradient[x_,y_]:={D[objectiveFunction[x,y],x],D[objectiveFunction[x,y],y]};

(* Define the Hessian matrix of the objective function: *)
objectiveHessian[x_,y_]:=D[objectiveGradient[x,y],{{x,y}}];

(* Implement Newton's optimization method: *)
NewtonOptimization2D[{x0_,y0_},tolerance_,maxIterations_]:=Module[
{x=x0,y=y0,deltaX,deltaY,iteration=0,trajectory={}},

(* Store initial point in the trajectory: *)
AppendTo[trajectory,{x,y}];

(* Iteratively update the solution until convergence or maximum iterations: *)
While[
Norm[objectiveGradient[x,y]]>tolerance&&iteration<maxIterations,

(* Compute the Newton update direction: *)
{deltaX,deltaY}=-Inverse[objectiveHessian[x,y]].objectiveGradient[x,y];
(* Perform Newton's update rule: *)
x=x+deltaX;
y=y+deltaY;
(* Store the updated point in the trajectory: *)
AppendTo[trajectory,{x,y}];
(* Increment iteration count: *)
iteration++;
];
(* Return the trajectory and the final solution: *)
{trajectory,{x,y,objectiveFunction[x,y]}}
]

(* Define Manipulate with sliders: *)
Manipulate[
(* Perform Newton optimization: *)

{trajectory,{minimizerX,minimizerY,minimumValue}}=NewtonOptimization2D[{initialX,i
nitialY},tolerance,maxIterations];

(* Plot the optimization process with arrows: *)
plot=ContourPlot[
objectiveFunction[x,y],
{x,-10,10},
{y,-10,10},
PlotLegends->Automatic,

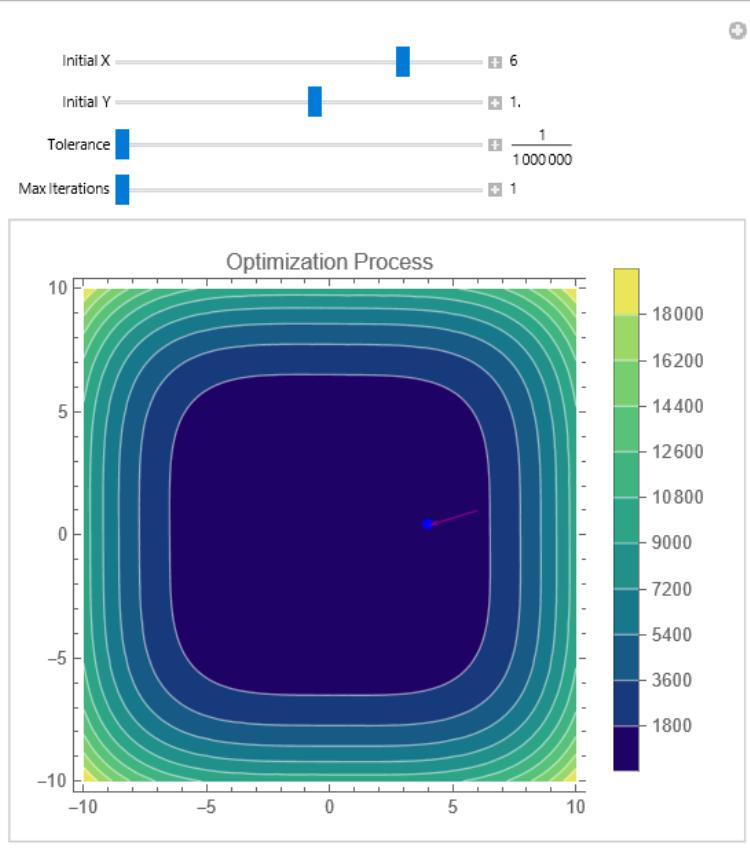
```

```

PlotLabel -> "Optimization Process",
Contours -> 10,
ContourStyle -> {White},
ClippingStyle -> Automatic,
ColorFunction -> "BlueGreenYellow",
ImageSize -> 300,
Epilog -> {
    Blue,
    PointSize[0.02],
    Point[{minimizerX, minimizerY}],
    Purple,
    Arrowheads[0.02],
    Arrow/@Partition[trajectory, 2, 1]
}
];
(* Display the plot: *)
plot,
(* Controls for Manipulate: *)
{{initialX, 6, "Initial X"}, -10, 10, 0.1, Appearance -> "Labeled"},
{{initialY, 1.0, "Initial Y"}, -10, 10, 0.1, Appearance -> "Labeled"},
{{tolerance, 10^(-6), "Tolerance"}, 10^(-8), 1, 10^(-8), Appearance -> "Labeled"},
{{maxIterations, 1, "Max Iterations"}, 1, 30, 1, Appearance -> "Labeled"}]
]

```

Output

**Mathematica Code 6.22****Marquardt Optimization Method**

Input (* The code defines an implementation of Marquardt's optimization method in Mathematica. It starts by defining the objective function $f(x,y)$, its gradient, and the Hessian matrix. The MarquardtOptimization2D function iteratively updates the solution until convergence or reaching the maximum number of iterations. Within each iteration, it computes the pseudo-inverse of the Hessian matrix, the search

direction, and updates the solution based on Marquardt's update rule, adjusting the damping factor accordingly. The Manipulate interface allows users to visualize the optimization process by adjusting the initial coordinates, damping factor, tolerance, and maximum iterations through sliders. Finally, it plots the optimization process and displays the trajectory with arrows indicating the direction of movement: *)

```

(* Define the objective function: *)
f[x_,y_]:=Sin[x]+Cos[y]+x^4+y^4+x*y;

(* Define the gradient of the objective function: *)
gradient[x_,y_]={D[f[x,y],x],D[f[x,y],y]};

(* Define the Hessian matrix of the objective function: *)
hessian[x_,y_]:=D[gradient[x,y],{{x,y}}];

(* Implement Marquardt's optimization method: *)
MarquardtOptimization2D[{x0_,y0_},λ0_,tol_,maxIterations_]:=Module[
{x=x0,y=y0,λ=λ0,deltaX,deltaY,iteration=0,trajectory={}}];

(* Store initial point in the trajectory: *)
AppendTo[trajectory,{x,y}];

(* Iteratively update the solution until convergence or the maximum number of
iterations: *)
While[
Norm[gradient[x,y]]>tol&&iteration<maxIterations,
(* Compute the pseudo-inverse of the Hessian matrix: *)
hessianInverse=PseudoInverse[hessian[x,y]+λ*IdentityMatrix[2]];
(* Compute the search direction: *)
{deltaX,deltaY}=-hessianInverse.gradient[x,y];
(* Perform Marquardt's update rule: *)
xNew=x+deltaX;
yNew=y+deltaY;

(* Check if the new solution improves the objective function: *)
If[
f[xNew,yNew]<f[x,y],
(* Accept the step: *)
x=xNew;
y=yNew;
λ=λ/2; (* Decrease the damping factor. *),
(* Reject the step*)
λ=2*λ; (* Increase the damping factor. *)];

(* Store the updated point in the trajectory: *)
AppendTo[trajectory,{x,y}];
(* Increment iteration count: *)
iteration++;
];
(* Return the trajectory and the final solution: *)
{trajectory,{x,y,f[x,y]}}
]

(* Define Manipulate with sliders: *)
Manipulate[
(* Perform Marquardt optimization: *)
{trajectory,{minimizerX,minimizerY,minimumValue}}=MarquardtOptimization2D[{initial
X,initialY},λ,tolerance,maxIterations];
(* Plot the optimization process with arrows: *)

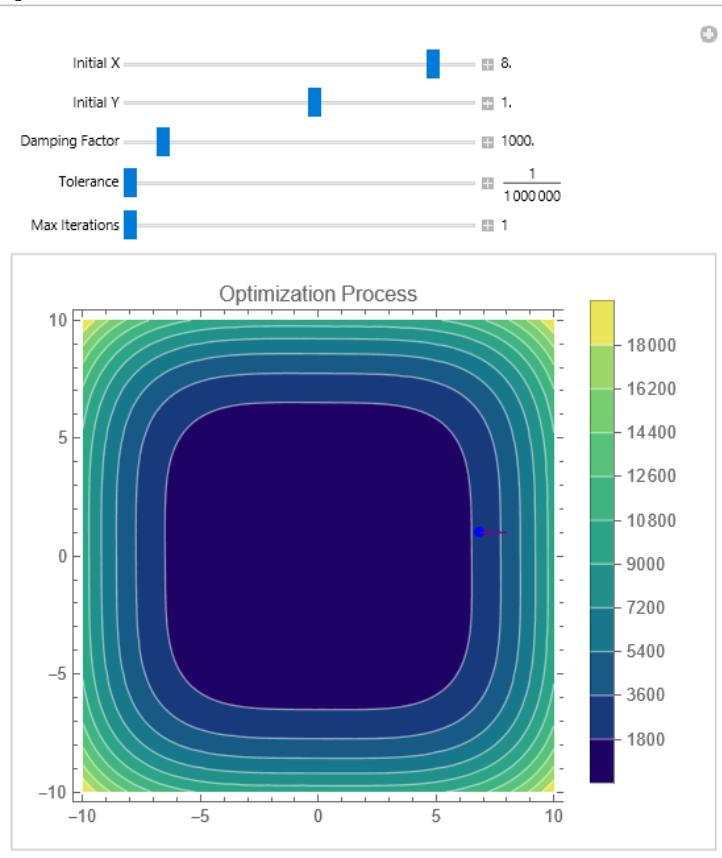
```

```

plot=ContourPlot[
  f[x,y],
  {x,-10,10},
  {y,-10,10},
  PlotLabel->Row[{"Optimization Process"}],
  PlotLegends->Automatic,
  Contours->10,
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  ImageSize->300,
  Epilog->{
    Blue,
    PointSize[0.02],
    Point[{minimizerX,minimizerY}],
    Purple,
    Arrowheads[0.02],
    Arrow/@Partition[trajectory,2,1]
  ];
(* Display the plot: *)
plot,
(* Controls for Manipulate: *)
{{initialX,8.0,"Initial X"},-10,10,0.1,Appearance->"Labeled"},
{{initialY,1.0,"Initial Y"},-10,10,0.1,Appearance->"Labeled"},
{{λ,1000.0,"Damping Factor"},0.1,10000,10,Appearance->"Labeled"},
{{tolerance,10^(-6),"Tolerance"},10^(-8),1,10^(-8),Appearance->"Labeled"},
{{maxIterations,1,"Max Iterations"},1,20,1,Appearance->"Labeled"}
]

```

Output



Mathematica Code 6.23**Conjugate Gradient**

Input

```

(* The code employs a Manipulate interface for interactive exploration of an
optimization process aiming to minimize the objective function  $(1-x)^2+100(-x^2-y)^2+1$ . Utilizing the Built-in method, "ConjugateGradient", the algorithm iterates
to find the minimum, with initial guesses for x and y specified by x0 and y0. The
optimization aims for a high degree of accuracy and precision, with AccuracyGoal
and PrecisionGoal both set to 20, and WorkingPrecision set to 40 for numerical
computations. Throughout the optimization, the steps taken are recorded for
visualization. The resulting optimization path is depicted alongside the contour
plot of the objective function, providing insights into the convergence behavior
and the impact of varying initial guesses: *)

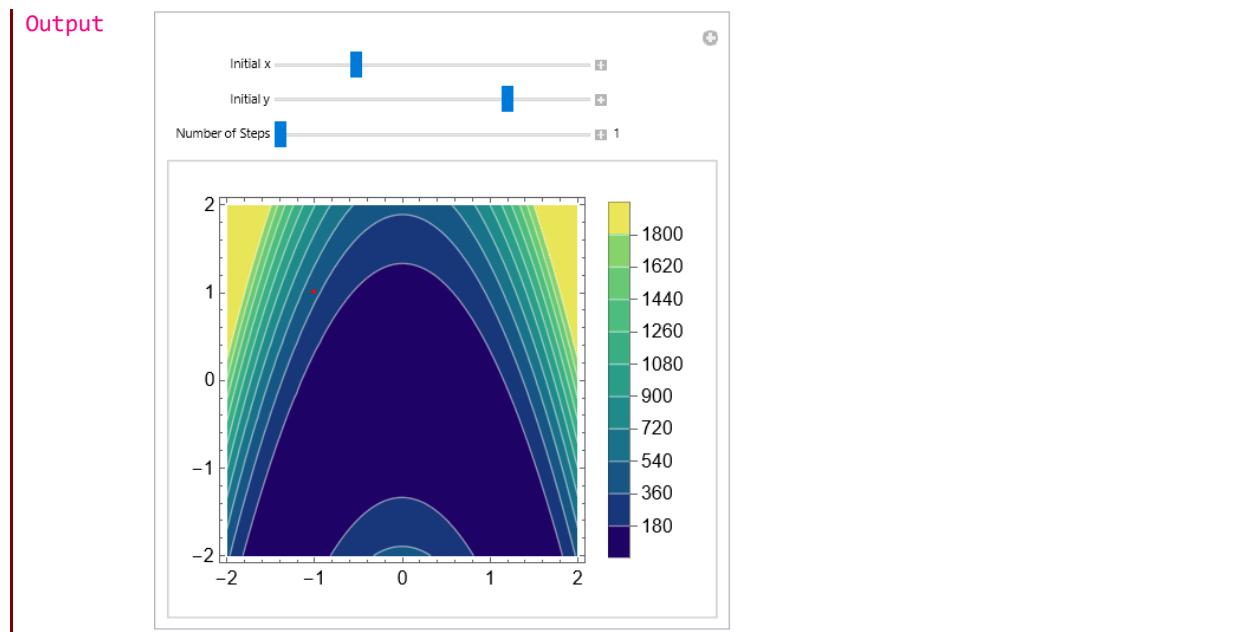
(* Define a Manipulate expression allowing interactive exploration of the
optimization process: *)
Manipulate[
minimumPath=Reap[
FindMinimum[
(* Function to minimize: *)
(1-x)^2+100 (-x^2-y)^2+1,
(* Initial guesses for x and y: *)
{{x,x0},{y,y0}},
(* Optimization method: *)
Method->"ConjugateGradient",
AccuracyGoal->20,
PrecisionGoal->20,
WorkingPrecision->40,
(* Record optimization steps: *)
StepMonitor:>Sow[{x,y}]]][[2,1]];

(* Join initial point with optimization path: *)
minimumPath=Join[{{x0,y0}},minimumPath];

(* Trim the path to display only the selected number of steps: *)
trimmedPath=Take[minimumPath,{1,numSteps}];

(* ContourPlot to visualize the function and optimization path: *)
ContourPlot[
(1-x)^2+100 (-x^2-y)^2+1,
{x,-2,2},
{y,-2,2},
Epilog->{Red,Line[trimmedPath],Point[trimmedPath]},
PlotLegends->Automatic,
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->250,
LabelStyle->Directive[Black,12]
],
{{x0,-1,"Initial x"},-2,2,0.1},
{{y0,1,"Initial y"},-2,2,0.1},
{{numSteps,1,"Number of Steps"},1,Length[minimumPath],1,Appearance->"Labeled"}
]

```

**Mathematica Code 6.24****Quasi-Newton**

```

Input (* The code utilizes the FindMinimum function to minimize the function  $(x^2+y-11)^2+(x+y^2-7)^2$  using the Built-in "QuasiNewton" optimization method. It sets the optimization goals with an AccuracyGoal and PrecisionGoal both set to 20, indicating the desired absolute and relative accuracies for the minimum value of the function. Additionally, the WorkingPrecision parameter is set to 20, specifying the precision used in computations during the optimization process. Through these goals, the algorithm aims to find a minimum of the function with high precision and accuracy, ensuring that the result meets stringent numerical criteria. The Manipulate expression allows for interactive exploration of the optimization process by varying the initial values of x and y: *)

(* Define a Manipulate expression allowing interactive exploration of the optimization process: *)
Manipulate[  

  minimumPath=Reap[  

    FindMinimum[  

      (* Objective function to minimize: *)  

      (x^2+y-11)^2+(x+y^2-7)^2,  

      (* Initial guesses for x and y: *)  

      {{x,initialX},{y,initialY}},  

      (* Optimization method: *)  

      Method->"QuasiNewton",  

      AccuracyGoal->20,  

      PrecisionGoal->20,  

      WorkingPrecision->20,  

      (* Record optimization steps: *)  

      StepMonitor:>Sow[{x,y}]  

    ]  

  ][[2,1]];  

  (* Join initial point with optimization path: *)  

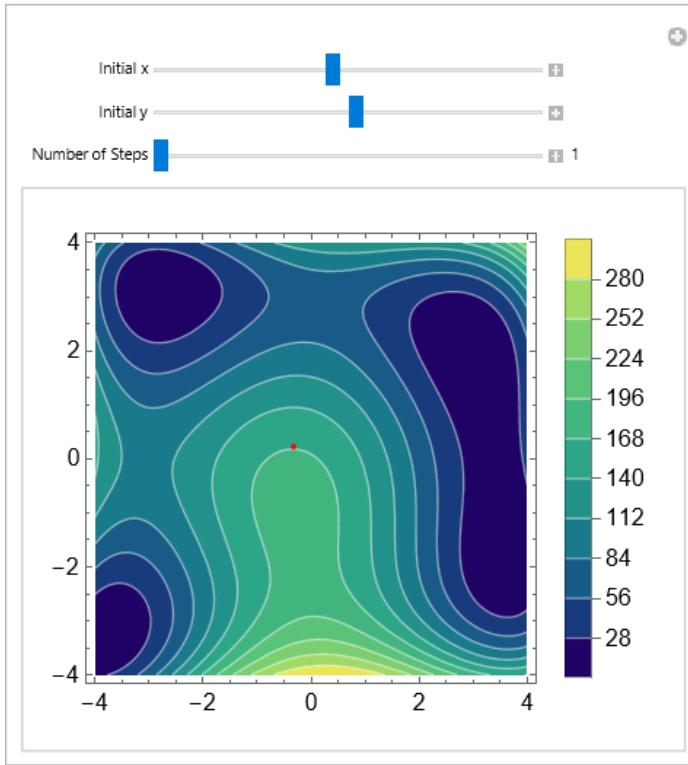
  minimumPath=Join[{{initialX,initialY}},minimumPath];
]

```

```
(* Trim the path to display only the selected number of steps: *)
trimmedPath=Take[minimumPath,{1,numSteps}];

(* ContourPlot to visualize the objective function and optimization path: *)
ContourPlot[
  (x^2+y-11)^2+(x+y^2-7)^2,
  {x,-4,4},
  {y,-4,4},
  PlotLegends->Automatic,
  Contours->10,
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  ImageSize->250,
  (* Overlay optimization path on plot: *)
  Epilog->{Red,Line[trimmedPath],Point[trimmedPath]},
  LabelStyle->Directive[Black,12]
],
{{initialX,-0.3,"Initial x"},-4,4,0.1},
{{initialY,0.2,"Initial y"},-4,4,0.1},
{{numSteps,1,"Number of Steps"},1,Length[minimumPath],1,Appearance->"Labeled"}
]
```

Output

**Mathematica Code 6.25****Conjugate Gradient, Newton, and Quasi-Newton methods**

Input

(* The code implements an interactive visualization tool for exploring the optimization process of an objective function in two variables using various optimization methods. Through a Manipulate interface, users can adjust parameters such as the initial values of x and y, the optimization method (including options like Conjugate Gradient, Newton's method, or Quasi-Newton method), and the number of optimization steps to display. The objective function's contour plot is generated to visualize its behavior, with the optimization path overlaid to illustrate the algorithm's progression towards the minimum. This tool facilitates an intuitive

understanding of optimization algorithms by offering real-time visual feedback and allowing users to experiment with different settings, aiding in the exploration of optimization strategies and their convergence properties: *)

```

Manipulate[
Module[
{minimumPath,trimmedPath},
(*Find the minimum of the objective function using the selected method*)
minimumPath=Reap[
FindMinimum[
(*Objective function to minimize*)
(x^4-16*x^2+5*x)/2+(y^4-16*y^2+5*y)/2,
(*Initial guesses for x and y*)
{{x,initialX},{y,initialY}},

(*Optimization method*)
Method->Evaluate[method/. {"ConjugateGradient"->"ConjugateGradient",
"Newton"->"Newton",
"QuasiNewton"->"QuasiNewton"
}],
AccuracyGoal->20,
PrecisionGoal->20,
WorkingPrecision->40,
(*Record optimization steps*)
StepMonitor:>Sow[{x,y}]
]
][[2,1]];

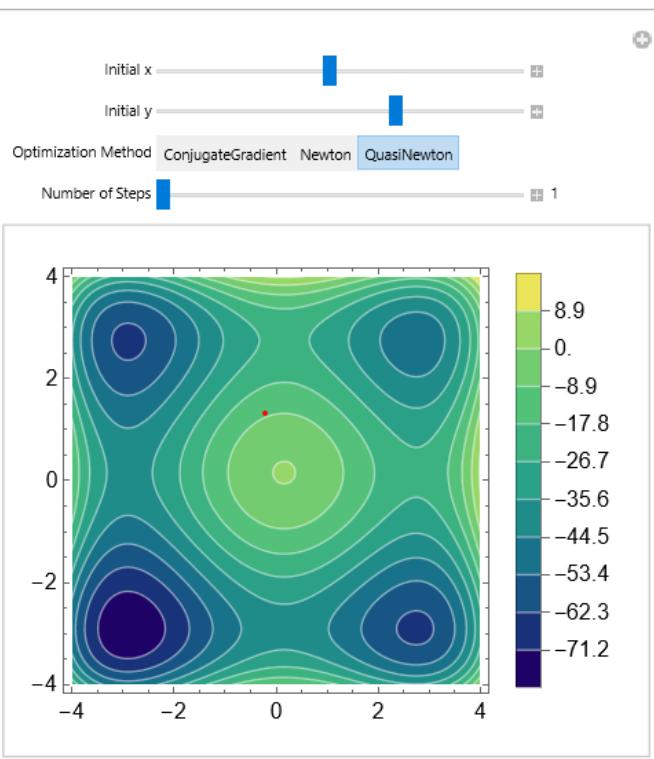
(* Join initial point with optimization path: *)
minimumPath=Join[{{initialX,initialY}},minimumPath];

(*Trim the path to display only the selected number of steps*)
trimmedPath=Take[minimumPath,{1,numSteps}];

(*ContourPlot to visualize the objective function and optimization path*)
ContourPlot[
(x^4-16*x^2+5*x)/2+(y^4-16*y^2+5*y)/2,
{x,-4,4},
{y,-4,4},
PlotLegends->Automatic,
Contours->10,
ContourStyle->{White},
ClippingStyle->Automatic,
ColorFunction->"BlueGreenYellow",
ImageSize->250,

(*Overlay optimization path on plot*)
Epilog->{Red ,Line[trimmedPath],Point[trimmedPath]},
LabelStyle->Directive[Black,12]
]
],
{{initialX,-0.2,"Initial x"},-4,4,0.1},
{{initialY,1.3,"Initial y"},-4,4,0.1},
{{method,"QuasiNewton","Optimization
Method"}, {"ConjugateGradient","Newton","QuasiNewton"}},
{{numSteps,1,"Number of Steps"},1,Length[minimumPath],1,Appearance->"Labeled"}
]
]

```

Output

Unit 6.2

Optimizing Neural Networks with Learning Rate Schedules and Adaptive Algorithms in Mathematica

In Mathematica, `NetTrain` is a part of the neural network framework, which allows you to train neural networks easily. You can specify additional options to customize the training process. The `Method` option in `NetTrain` is indeed crucial for customizing the optimization algorithm used during training. It allows you to choose the specific optimization algorithm that best suits your neural network and training data.

Possible settings for `Method` include:

<code>"ADAM"</code>	stochastic gradient descent using an adaptive learning rate that is invariant to diagonal rescaling of the gradients
<code>"RMSProp"</code>	stochastic gradient descent using an adaptive learning rate derived from exponentially smoothed average of gradient magnitude
<code>"SGD"</code>	ordinary stochastic gradient descent with momentum
<code>"SignSGD"</code>	stochastic gradient descent for which the magnitude of the gradient is discarded

For the method `"SGD"`, the following additional suboptions are supported:

<code>"Momentum"</code>	0.93	how much to preserve the previous step when updating the derivative
-------------------------	------	---

For the method `"ADAM"`, the following additional suboptions are supported:

<code>"Beta1"</code>	0.9	exponential decay rate for the first moment estimate
<code>"Beta2"</code>	0.999	exponential decay rate for the second moment estimate
<code>"Epsilon"</code>	0.00001`	stability parameter

For the method `"RMSProp"`, the following additional suboptions are supported:

<code>"Beta"</code>	0.95	exponential decay rate for the moving average of the gradient magnitude
<code>"Epsilon"</code>	0.000001	stability parameter
<code>"Momentum"</code>	0.9	momentum term

For the method `"SignSGD"`, the following additional suboption is supported:

<code>"Momentum"</code>	0.93	how much to preserve the previous step when updating the derivative
-------------------------	------	---

Suboptions for specific methods can be specified using `Method -> {"method", opt1 -> val1, ...}`. The following suboption is supported for all methods:

<code>"LearningRateSchedule"</code>	Automatic	how to scale the learning rate as training progresses
-------------------------------------	-----------	---

In Mathematica's `NetTrain` function, the `"LearningRateSchedule"` option allows for dynamic adjustment of the learning rate during training. The learning rate for each batch is calculated as

$$\text{initial} * f[\text{batch}, \text{total}],$$

where: `initial` is the initial learning rate specified using the `"LearningRate"` option.

`f` is a function that takes two arguments: batch and total.

`batch` is the current batch number.

`total` is the total number of batches that will be visited during training.

The value returned by `f` should be a number between 0 and 1, representing the scale by which to adjust the initial learning rate for the current batch. You can customize the `f` function to implement different learning rate schedules according to your specific requirements, such as step decay, linear decay, or custom schedules based on the training progress. Adjusting the learning rate dynamically can help improve the convergence and performance of the neural network during training.

Mathematica Code 6.26**Step Decay (NN)**

Input

```
(* The code aims to train a neural network to learn a mapping from two-dimensional input points to their corresponding exponential distances from the origin. It starts by generating a dataset comprising pairs of input points and their exponential distances, then defines a neural network architecture with logistic sigmoid activation functions and initializes it with random weights and biases. Two different learning rate decay schedules (step decay) are employed during training to monitor the network's learning progress, and the training is limited to 10 rounds with a batch size of 1024. By visualizing the training progress through the evolution of learning rates over batches, the code assesses the impact of different learning rate schedules on the network's ability to learn the desired mapping accurately: *)

(* Generate dataset: pairs of input points and their corresponding exponential distances: *)
trainingData=Flatten@Table[{x,y}->{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-3,3,.005}];

(* Define neural network architecture: *)
network=NetChain[
  (* Three hidden layers with logistic sigmoid activation: *)
  {
    10,LogisticSigmoid,
    5,LogisticSigmoid,
    5,LogisticSigmoid,
    1
  },
  (* Input layer with 2 neurons representing 2D input points: *)
  "Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Define a function for training: *)
trainNetwork[learningRateSchedule_]:=NetTrain[
  initializedNetwork,
  trainingData,
  (* Monitor learning rate during training: *)
  <|
    "Property"->"LearningRate",
    "Interval"->"Batches"
  |>,
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Set initial learning rate: *)
  LearningRate->0.3,
  (* Use a batch size of 1024: *)
  BatchSize->1024,
  (* Use stochastic gradient descent with specified learning rate schedule: *)
  Method->{"SGD","LearningRateSchedule"->learningRateSchedule}
]

(* Define learning rate schedules using step decay: *)
```

```
(* With "LearningRateSchedule"->f, the learning rate for a given batch will be
calculated as initial*f[batch, total], where batch is the current batch number,
total is the total number of batches that will be visited during training, and
initial is the initial learning rate specified using the LearningRate option. The
value returned by f should be a number between 0 and 1: *)
```

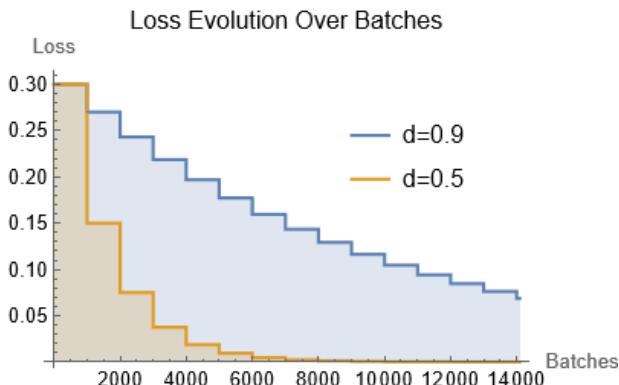
```
(* Step decay:  $\alpha_t = \alpha_0 * d^{\lfloor t/s \rfloor}$ ,  $\alpha_0=0.3$  is the initial learning rate,  $d=0.9$  or  $d=0.5$  is
the constant factor (drop factor or decay factor) by which the learning rate drops
each time,  $s=1000$  is the step size, indicating after how many iterations the learning
rate should be decayed: *)
```

```
decaySchedule1[b_,bmax_]:=0.9^Floor[b/1000];
decaySchedule2[b_,bmax_]:=0.5^Floor[b/1000];

(* Train the network using different learning rate schedules: *)
{result1,result2}=trainNetwork/@{decaySchedule1,decaySchedule2};
```

```
(* Visualize training progress: *)
ListPlot[
{result1,result2},
Filling->Axis,
Mesh->All,
Joined->True,
PlotLabel->"Loss Evolution Over Batches",
AxesLabel->{"Batches","Loss"},
PlotLegends->Placed[{"d=0.9","d=0.5"},{0.75,0.7}],
ImageSize->300
]
```

Output

**Mathematica Code 6.27****Exponential Decay Schedules (NN)**

Input (* Two different learning rate decay schedules (two exponential decay schedules) are employed during training to monitor the network's learning progress, and the training is limited to 5 rounds with a batch size of 5000. By visualizing the training progress through the evolution of learning rates over rounds, the code assesses the impact of different learning rate schedules on the network's ability to learn the desired mapping accurately: *)

```
(* Generate dataset: pairs of input points and their corresponding exponential
distances: *)
trainingData=Flatten@Table[{x,y}->{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-
3,3,.005}];
```

```
(* Define neural network architecture: *)
network=NetChain[
(* Three hidden layers with logistic sigmoid activation: *)
```

```

{
  10,LogisticSigmoid,
  5,LogisticSigmoid,
  5,LogisticSigmoid,
  1
},
(* Input layer with 2 neurons representing 2D input points: *)
"Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Define a function for training: *)
trainNetwork[learningRateSchedule_]:=NetTrain[
  initializedNetwork,
  trainingData,
  {
    (* Monitor Loss during training: *)
    "BatchLossList",
    (* Monitor learning rate during training: *)
    <|
      "Property"->"LearningRate",
      "Interval"->"Batches"
    |>
    },
    (* Limit training to 5 rounds: *)
    MaxTrainingRounds->5,
    (* Set initial learning rate: *)
    LearningRate->0.3,
    (* Use a batch size of 5000: *)
    BatchSize->5000,
    (* Use stochastic gradient descent with specified learning rate schedule: *)
    Method->{"SGD","LearningRateSchedule"->learningRateSchedule}
  ]
]

(* Define learning rate schedules using exponential decay: *)

(* With "LearningRateSchedule"->f, the learning rate for a given batch will be
calculated as initial*f[batch, total], where batch is the current batch number, total
is the total number of batches that will be visited during training, and initial
is the initial learning rate specified using the LearningRate option. The value returned
by f should be a number between 0 and 1: *)

(* Exponential Decay:  $\alpha_t = \alpha_0 e^{-d*t}$ , d is the decay rate parameter. Note that,
we used "Interval"->"Batches" not "Interval"->"Rounds": *)

(* Define a function for generating learning rate schedules: *)
generateSchedule[decayRate_]:=Function[
  {b,bmax},
  Exp[-b*decayRate]
]

(* Define learning rate schedules: *)

```

```

decaySchedule1=generateSchedule[0.001];
decaySchedule2=generateSchedule[0.01];

(* Train the network using different learning rate schedules: *)
{loss1,result1,loss2,result2}=Flatten[trainNetwork@{decaySchedule1,decaySchedule2
},1];

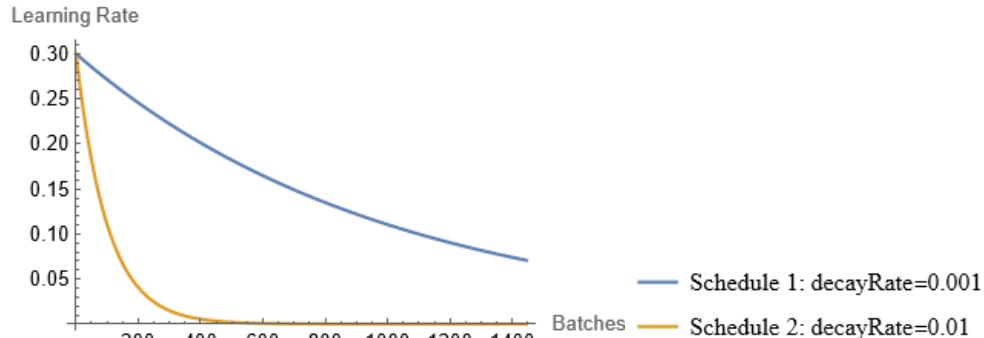
(* Visualize training progress: *)
ListPlot[
{result1,result2},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotLabel->"Learning Rate Evolution Over Batches",
PlotLegends-> {"Schedule 1: decayRate=0.001","Schedule 2: decayRate=0.01"},
ImageSize->300 ,
AxesLabel->{"Batches","Learning Rate"}
]

ListPlot[
{loss1,loss2},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotStyle->{Opacity[0.9],Opacity[0.5]},
PlotLabel->"Loss Evolution Over Batches",
PlotLegends-> {"Loss of Schedule 1","Loss of Schedule 2"},
ImageSize->300 ,
AxesLabel->{"Batches","Loss"}
]

```

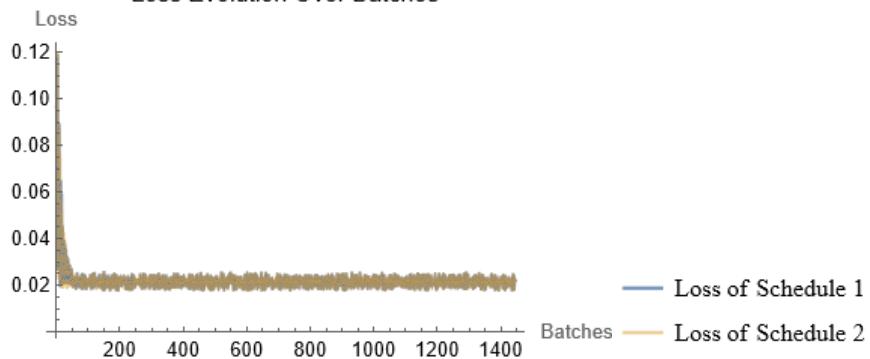
Output

Learning Rate Evolution Over Batches



Output

Loss Evolution Over Batches



Mathematica Code 6.28**Polynomial Learning Rate (NN)**

Input

```
(* Two different learning rate decay schedules (polynomial learning rate policy)
are employed during training to monitor the network's learning progress, and the
training is limited to 10 rounds with a batch size of 1024. By visualizing the
training progress through the evolution of learning rates over batches, the code
assesses the impact of different learning rate schedules on the network's ability
to learn the desired mapping accurately: *)

(* Generate dataset: pairs of input points and their corresponding exponential
distances: *)
trainingData=Flatten@Table[{x,y}->{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-
3,3,.005}];

(* Define neural network architecture: *)
network=NetChain[
  (* Three hidden layers with logistic sigmoid activation: *)
  {
    10,LogisticSigmoid,
    5,LogisticSigmoid,
    5,LogisticSigmoid,
    1
  },
  (* Input layer with 2 neurons representing 2D input points: *)
  "Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Random","Weights"->0.05,"Biases"->0.05},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Define a function for training: *)
trainNetwork[learningRateSchedule_]:=NetTrain[
  initializedNetwork,
  trainingData,
  (* Monitor learning rate during training: *)
  <|
    "Property"->"LearningRate",
    "Interval"->"Batches"
  |>,
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Set initial learning rate: *)
  LearningRate->0.3,
  (* Use a batch size of 1024: *)
  BatchSize->1024,
  (* Use stochastic gradient descent with specified learning rate schedule: *)
  Method->{"SGD","LearningRateSchedule"->learningRateSchedule}
]

(* Define learning rate schedules using polynomial learning rate policy: *)

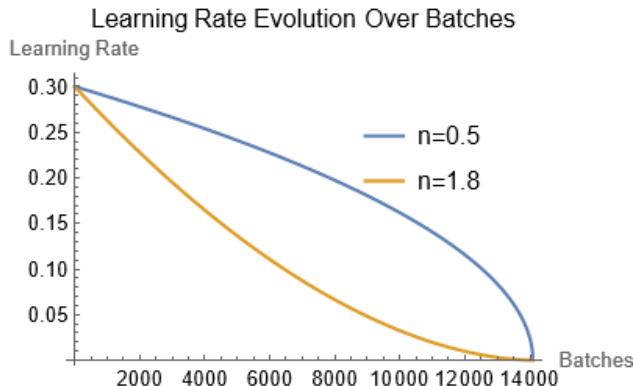
(* With "LearningRateSchedule"->f, the learning rate for a given batch will be
calculated as (initial*f[batch, total]), where batch is the current batch number,
total is the total number of batches that will be visited during training, and
```

initial is the initial learning rate specified using the LearningRate option. The value returned by f should be a number between 0 and 1: *)

```
(* Polynomial Learning Rate Policy function  $\alpha_t = \alpha_0 * (1-t/T_t)^n$ , where  $\alpha_0=0.3$  represents the initial learning rate, t signifies the number of iterations, and  $T_t$  denotes the total number of iterations. The power term, n=0.5 or n=1.8, within the formula, serves as a critical determinant, shaping the decay characteristics of the learning rate. *)
decaySchedule1[b_,bmax_]:= (1-b/bmax)^0.5;
decaySchedule2[b_,bmax_]:= (1-b/bmax)^1.8;

(* Train the network using different learning rate schedules: *)
{result1,result2}=trainNetwork/@{decaySchedule1,decaySchedule2};

(* Visualize training progress: *)
ListPlot[
{result1,result2},
Mesh->All,
Joined->True,
PlotLabel->"Learning Rate Evolution Over Batches",
PlotLegends->Placed[{"n=0.5","n=1.8"},{0.75,0.7}],
ImageSize->300 ,
AxesLabel->{"Batches","Learning Rate"}
]
```

Output**Mathematica Code 6.29****Triangular Learning Rate (NN)**

Input (* Two different learning rate decay schedules (two triangular learning rate schedules) are employed during training to monitor the network's learning progress, and the training is limited to 20 rounds with a batch size of 1024. By visualizing the training progress through the evolution of learning rates over rounds, the code assesses the impact of different learning rate schedules on the network's ability to learn the desired mapping accurately: *)

```
(* Generate dataset: pairs of input points and their corresponding exponential distances: *)
trainingData=Flatten@Table[{x,y}->{Exp[-Norm[{x,y}]]},{x,-3,3,.005},{y,-3,3,.005}];

(* Define neural network architecture: *)
network=NetChain[
(* Three hidden layers with logistic sigmoid activation: *)
{
  10,LogisticSigmoid,
  5,LogisticSigmoid,
```

```

5,LogisticSigmoid,
1
},
(* Input layer with 2 neurons representing 2D input points: *)
"Input"->2
];

(* Initialize the network with random weights and biases: *)
initializedNetwork=NetInitialize[
network,
(* Specify random initialization method: *)
Method->{"Random","Weights"->0.05,"Biases"->0.05},
(* Ensure reproducibility by setting random seed: *)
RandomSeeding->Automatic
];

(* Define a function for training: *)
trainNetwork[learningRateSchedule_]:=NetTrain[
initializedNetwork,
trainingData,
{
(* Monitor Loss during training: *)
"RoundLossList",
(* Monitor learning rate during training: *)
<|
"Property"->"LearningRate",
"Interval"->"Rounds"
|>
},
(* Limit training to 20 rounds: *)
MaxTrainingRounds->20,
(* Set initial learning rate: *)
LearningRate->0.3,
(* Use a batch size of 1024: *)
BatchSize->1024,
(* Use stochastic gradient descent with specified learning rate schedule: *)
Method->{"SGD","LearningRateSchedule"->learningRateSchedule}
]

(* Define learning rate schedules using triangular method: *)

(* With "LearningRateSchedule"->f, the learning rate for a given batch will be
calculated as initial*f[batch, total], where batch is the current batch number, total
is the total number of batches that will be visited during training, and initial
is the initial learning rate specified using the LearningRate option. The value returned
by f should be a number between 0 and 1: *)

(* Triangular learning rate schedule:  $\alpha_t = \alpha_{\min} + (\alpha_{\max} - \alpha_{\min}) * \max[0, (1-x)]$ , where
 $x = |t/s - 2c + 1|$ , and c can be calculated as,  $c = \lfloor 1 + t/2s \rfloor$ , where  $\alpha_{\min}$  is the specified
lower (i.e., base) learning rate, t is the number of iterations of training, and  $\alpha_t$ 
is the computed learning rate. s is half the period or cycle length and  $\alpha_{\max}$  is the
maximum learning rate boundary. Note that, we used "Interval"->"Rounds" not
"Interval"->"Batches": *)

(* Define a function for generating learning rate schedules: *)
generateSchedule[baseLR_,maxLR_,stepsize_]:=Module[
{cycle,x},
Function[
{b,bmax},
cycle=Floor[1+b/(2*stepsize)];
]
];

```

```

x=Abs[b/stepsize-2*cycle+1];
baseLR+(maxLR-baseLR)*Max[0,(1-x)]
]

(* Define learning rate schedules: *)
decaySchedule1=generateSchedule[0.001,0.01,2];
decaySchedule2=generateSchedule[0.1,0.5,5];

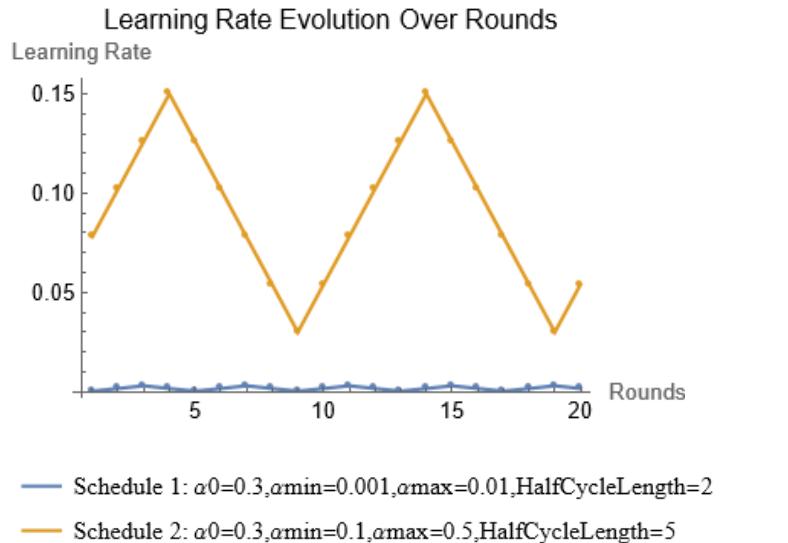
(* Train the network using different learning rate schedules: *)
{loss1,result1,loss2,result2}=Flatten[trainNetwork/@{decaySchedule1,decaySchedule2
},1];

(* Visualize training progress: *)
ListPlot[
{result1,result2},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotLabel->"Learning Rate Evolution Over Rounds",
PlotLegends-> {"Schedule 1:
 $\alpha_0=0.3, \alpha_{min}=0.001, \alpha_{max}=0.01, \text{HalfCycleLength}=2$ ", "Schedule 2:
 $\alpha_0=0.3, \alpha_{min}=0.1, \alpha_{max}=0.5, \text{HalfCycleLength}=5$ "},
ImageSize->300 ,
AxesLabel->{"Rounds", "Learning Rate"}
]

ListPlot[
{loss1,loss2},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotLabel->"Loss Evolution Over Rounds",
PlotLegends-> {"Loss of Schedule 1", "Loss of Schedule 2",
ImageSize->300 ,
AxesLabel->{"Rounds", "Loss"}
]

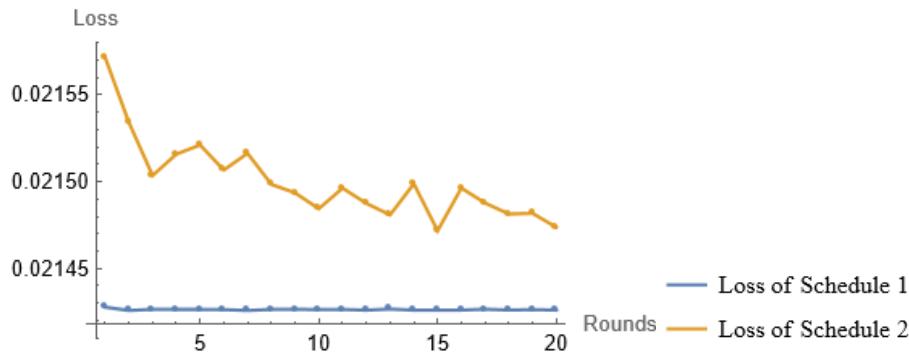
```

Output



Output

Loss Evolution Over Rounds

**Mathematica Code 6.30****Adam Optimizer (NN)**

```

Input (* The code aims to train a neural network using the Adam optimizer with various
       hyperparameter configurations, starting with generating example data representing
       2D input points and their corresponding output values. The neural network
       architecture, consisting of linear layers and ReLU activation functions, is defined
       and initialized with Kaiming initialization. Through the training process, the
       network learns to minimize the loss function on the provided dataset, with training
       progress monitored and recorded. By adjusting the beta1 and beta2 parameters of the
       Adam optimizer, the code systematically explores different optimization settings to
       observe their impact on training performance. Finally, the code visualizes the
       evolution of loss over training rounds for each hyperparameter configuration,
       providing insights into how different optimization settings influence the network's
       learning dynamics: *)

(* Generate some example data for training: *)
trainingData=Flatten@Table[{x,y}=>{Sin[x]+Sin[y]},{x,-3,3,.005},{y,-3,3,.005}];

(* Define the architecture of the neural network: *)
network=NetChain[
  {
    LinearLayer[10],ElementwiseLayer[Ramp],
    LinearLayer[10],ElementwiseLayer[Ramp],
    LinearLayer[1]
  },
  (* Input layer with 2 neurons representing 2D input points: *)
  "Input"->2
];

(* Initialize the network with Kaiming initialization: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Kaiming","Distribution"->"Normal"},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

(* Train the neural network using the Adam optimizer: *)
trainNetwork[method_,beta1_,beta2_]:=NetTrain[
  initializedNetwork,
  trainingData,
  (* Monitor Loss during training: *)
  "RoundLossList",
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Set initial learning rate: *)
  ...
];

```

```

LearningRate->0.1,
(* Use a batch size of 1024: *)
BatchSize->1024,
(* Use ADAM optimization method with specified Parameters
(beta1,beta2,epsilon): *)
Method->{method,"Beta1"->beta1,"Beta2"->beta2,"Epsilon"->0.00001}
];

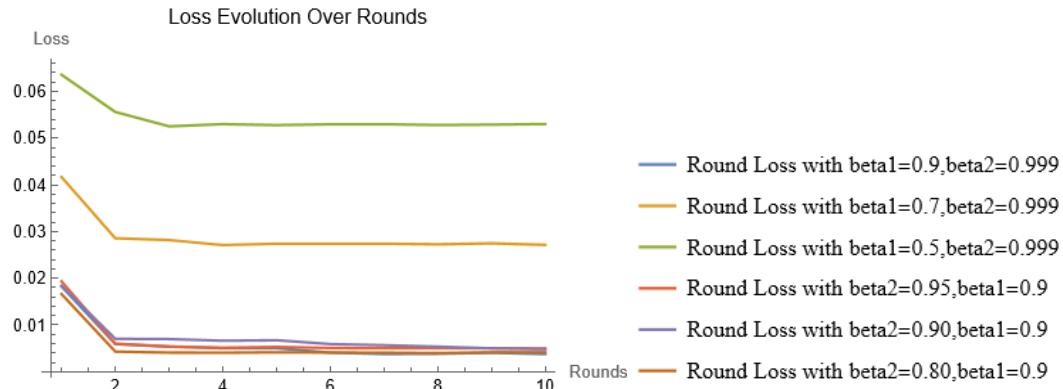
(* Train the network using different beta1 with beta2=0.999: *)
roundLoss1=trainNetwork[ "ADAM",0.9,0.999];
roundLoss2=trainNetwork[ "ADAM",0.7,0.999];
roundLoss3=trainNetwork[ "ADAM",0.5,0.999];

(* Train the network using different beta2 with beta1=0.9: *)
roundLoss4=trainNetwork[ "ADAM",0.9,0.95];
roundLoss5=trainNetwork[ "ADAM",0.9,0.90];
roundLoss6=trainNetwork[ "ADAM",0.9,0.80];

(* Plot the Loss evolution over rounds: *)
ListPlot[
{roundLoss1,roundLoss2,roundLoss3,roundLoss4,roundLoss5,roundLoss6},
PlotLegends->{
"Round Loss with beta1=0.9,beta2=0.999",
"Round Loss with beta1=0.7,beta2=0.999",
"Round Loss with beta1=0.5,beta2=0.999",
"Round Loss with beta2=0.95,beta1=0.9",
"Round Loss with beta2=0.90,beta1=0.9",
"Round Loss with beta2=0.80,beta1=0.9"
},
Joined->True,
PlotRange->Full,
PlotLabel->"Loss Evolution Over Rounds",
AxesLabel->{"Rounds","Loss"}
]

```

Output

**Mathematica Code 6.31****RMSProp optimizer (NN)**

Input

(* The code aims to train a neural network using the RMSProp optimizer with various hyperparameter configurations, starting with generating example data representing 2D input points and their corresponding output values. The neural network architecture, consisting of linear layers and ReLU activation functions, is defined and initialized with Kaiming initialization. Through the training process, the network learns to minimize the loss function on the provided dataset, with training progress monitored and recorded. By adjusting the beta and momentum parameters of the RMSProp optimizer, the code systematically explores different optimization

settings to observe their impact on training performance. Finally, the code visualizes the evolution of loss over training rounds for each hyperparameter configuration, providing insights into how different optimization settings influence the network's learning dynamics: *)

```

(* Generate some example data for training: *)
trainingData=Flatten@Table[{x,y}->{Sin[x]+Sin[y]},{x,-3,3,.005},{y,-3,3,.005}];

(* Define the architecture of the neural network: *)
network=NetChain[
  {
    LinearLayer[10],ElementwiseLayer[Ramp],
    LinearLayer[10],ElementwiseLayer[Ramp],
    LinearLayer[1]
  },
  (* Input layer with 2 neurons representing 2D input points: *)
  "Input"->2
];

(* Initialize the network with Kaiming initialization: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Kaiming","Distribution"->"Normal"},
  (* Ensure reproducibility by setting random seed: *)
  RandomSeeding->Automatic
];

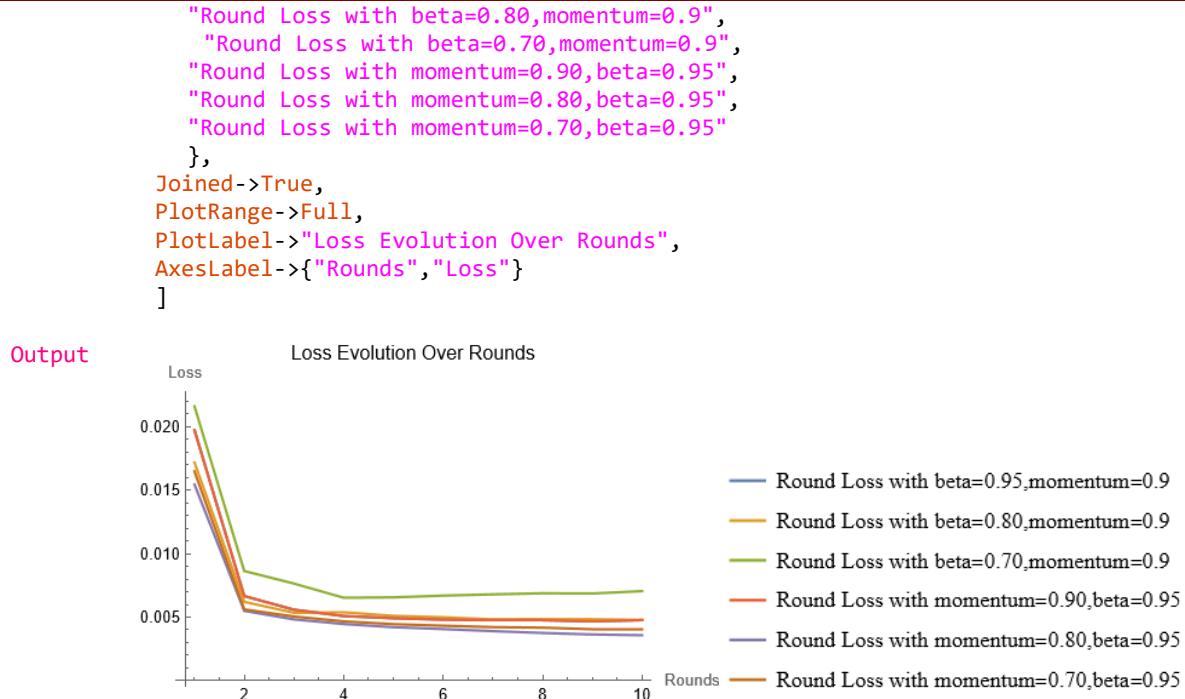
(* Train the neural network using the RMSProp optimizer: *)
trainNetwork[method_,beta_,momentum_]:=NetTrain[
  initializedNetwork,
  trainingData,
  (* Monitor Loss during training: *)
  "RoundLossList",
  (* Limit training to 10 rounds: *)
  MaxTrainingRounds->10,
  (* Set initial learning rate: *)
  LearningRate->0.01,
  (* Use a batch size of 1024: *)
  BatchSize->1024,
  (* Use RMSProp optimization method with specified Parameters (beta, momentum, epsilon): *)
  Method->{method,"Beta"->beta,"Momentum"->momentum,"Epsilon"->0.000001}
];

(* Train the network using different beta with momentum=0.9: *)
roundLoss1=trainNetwork["RMSProp",0.95,0.9];
roundLoss2=trainNetwork["RMSProp",0.80,0.9];
roundLoss3=trainNetwork["RMSProp",0.70,0.9];

(* Train the network using different momentum with beta=0.95: *)
roundLoss4=trainNetwork["RMSProp",0.95,0.90];
roundLoss5=trainNetwork["RMSProp",0.95,0.80];
roundLoss6=trainNetwork["RMSProp",0.95,0.70];

(* Plot the Loss evolution over rounds: *)
ListPlot[
  {roundLoss1,roundLoss2,roundLoss3,roundLoss4,roundLoss5,roundLoss6},
  PlotLegends->{
    "Round Loss with beta=0.95,momentum=0.9",
    "Round Loss with beta=0.80,momentum=0.9",
    "Round Loss with beta=0.70,momentum=0.9",
    "Round Loss with beta=0.95,momentum=0.80",
    "Round Loss with beta=0.95,momentum=0.70",
    "Round Loss with beta=0.95,momentum=0.90"
  }
];

```

**Mathematica Code 6.32** ADAM, RMSProp, SGD, with step decay and triangular schedule (NN)

Input

```

(* The code aims to train a neural network on generated example data by comparing
the performance of different optimization methods and learning rate schedules. It
first generates training data based on a predefined function and defines a neural
network architecture. The network is then initialized with appropriate weights using
Kaiming initialization. The code proceeds to train the network using various
optimization methods (ADAM, RMSProp, SGD) and learning rate schedules (step decay,
triangular schedule) while monitoring training progress by tracking loss function
evolution and learning rate changes. The goal is to visualize and analyze how
different optimization strategies impact the training process, facilitating the
identification of the most effective approach for the given dataset and network
architecture: *)

```

```

(* Generate some example data for training: *)
trainingData=Flatten@Table[{x,y}=>{Sin[x]+Sin[y]},{x,-3,3,.005},{y,-3,3,.005}];

(* Define the architecture of the neural network: *)
network=NetChain[
{
  LinearLayer[10],ElementwiseLayer[Ramp],
  LinearLayer[10],ElementwiseLayer[Ramp],
  LinearLayer[1]
},
(* Input layer with 2 neurons representing 2D input points: *)
"Input"->2
];

(* Initialize the network with Kaiming initialization: *)
initializedNetwork=NetInitialize[
  network,
  (* Specify random initialization method: *)
  Method->{"Kaiming","Distribution"->"Normal"}
];

```

```

(* Train the neural network using the different optimizer and
learningRateSchedule*)
trainNetwork[method_,learningRateSchedule_]:=NetTrain[
  initializedNetwork,
  trainingData,
  {
    (* Monitor Loss during training: *)
    "RoundLossList",
    (* Monitor learning rate during training: *)
    <|
      "Property"->"LearningRate",
      "Interval"->"Batches"
      |>
    },
    (* Limit training to 5 rounds: *)
    MaxTrainingRounds->5,
    (* Set initial learning rate: *)
    LearningRate->0.1,
    (* Use a batch size of 1024: *)
    BatchSize->1024,
    (* Use different optimization methods with specified learningRateSchedule: *)
    Method->{method,"LearningRateSchedule"->learningRateSchedule}
  ];
]

(* Define learning rate schedules: *)

(* With "LearningRateSchedule"->f, the learning rate for a given batch will be
calculated as initial*f[batch, total], where batch is the current batch number, total
is the total number of batches that will be visited during training, and initial
is the initial learning rate specified using the LearningRate option. The value returned
by f should be a number between 0 and 1: *)

(* Step decay:  $\alpha_t = \alpha_0 * d^{[t/s]}$ ,  $\alpha_0=0.3$  is the initial learning rate,  $d=0.9$  is the
constant factor (drop factor or decay factor) by which the learning rate drops each
time,  $s=1000$  is the step size, indicating after how many iterations the learning rate
should be decayed.*)

decaySchedule1[b_,bmax_]:=0.9^Floor[b/1000];

(* Triangular learning rate schedule:  $\alpha_t=\alpha_{\min}+(\alpha_{\max}-\alpha_{\min})*\max[0,(1-x)]$ , where
 $x=|t/s-2c+1|$ , and c can be calculated as,  $c=[1+t/2s]$ , where  $\alpha_{\min}$  is the specified
lower (i.e., base) learning rate, t is the number of iterations of training, and  $\alpha_t$ 
is the computed learning rate. s is half the period or cycle length and  $\alpha_{\max}$  is the
maximum learning rate boundary. Note that, we used "Interval"->"Batches" not
"Interval"->"Rounds": *)

generateSchedule[baseLR_,maxLR_,stepsize_]:=Module[
  {cycle,x},
  Function[
    {b,bmax},
    cycle=Floor[1+b/(2*stepsize)];
    x=Abs[b/stepsize-2*cycle+1];
    baseLR+(maxLR-baseLR)*Max[0,(1-x)]
  ]
]

decaySchedule2=generateSchedule[0.1,1,500];

(* Train the network using different learning rate schedules: *)

```

```

6}=Flatten[
  trainNetwork@@@{
    {"ADAM",decaySchedule1},
    {"RMSProp",decaySchedule1},
    {"SGD",decaySchedule1},
    {"ADAM",decaySchedule2},
    {"RMSProp",decaySchedule2},
    {"SGD",decaySchedule2}
  },1];

(* Visualize training progress: *)
ListPlot[
{result1,result2,result3,result4,result5,result6},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotLabel->"Learning Rate Evolution Over Batches",
PlotLegends-> {
  "ADAM with Schedule 1",
  "RMSProp with Schedule 1",
  "SGD with Schedule 1",
  "ADAM with Schedule 2",
  "RMSProp with Schedule 2",
  "SGD with Schedule 2"
},
ImageSize->300 ,
AxesLabel->{"Batches","Learning Rate"}
]

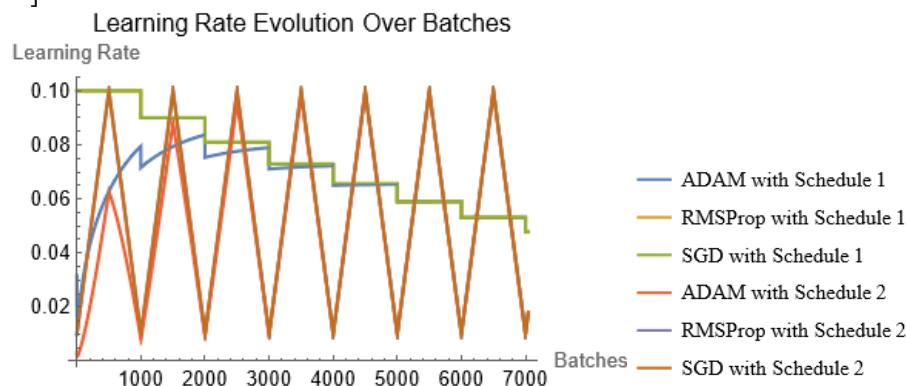
```

```

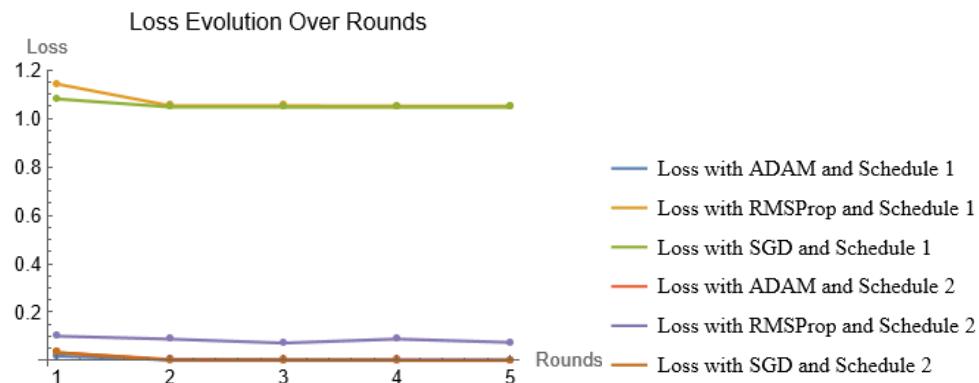
ListPlot[
{loss1,loss2,loss3,loss4,loss5,loss6},
PlotRange->Full,
Mesh->All,
Joined->True,
PlotLabel->"Loss Evolution Over Rounds",
PlotLegends-> {
  "Loss with ADAM and Schedule 1",
  "Loss with RMSProp and Schedule 1",
  "Loss with SGD and Schedule 1",
  "Loss with ADAM and Schedule 2",
  "Loss with RMSProp and Schedule 2",
  "Loss with SGD and Schedule 2"
},
ImageSize->300 ,
AxesLabel->{"Rounds","Loss"}
]

```

Output



Output



CHAPTER 7

STRATEGIES FOR GENERALIZATION AND HYPER-PARAMETER TUNING

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 6, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation.

In this chapter, we delve into some of the most critical concepts and methodologies that form the backbone of machine learning, guiding the development of models that are not only powerful but also robust and generalizable across various scenarios. We focus on the aspects of Neural Network (NN) training that determine how effectively a model can generalize from training data to unseen data. Generalization is the ultimate test of a NN's performance, assessing its ability to apply learned patterns to new datasets.

- We begin by addressing the fundamental challenges of overfitting and generalization. Overfitting occurs when the NN learns the details and noise in the training data to an extent that it negatively impacts the performance of NN on new data. Conversely, generalization refers to the model's ability to apply what it has learned to unseen data. We will discuss strategies to balance this, including the importance of a robust model architecture.
- Next, we explore the bias-variance trade-off, a pivotal concept that helps in diagnosing the performance of machine learning algorithms. Bias refers to errors due to overly simplistic assumptions in the learning algorithm. Variance refers to errors from sensitivity to small fluctuations in the training set. High bias can cause a model to miss the relevant relations between features and target outputs (underfitting), whereas high variance can cause modeling the random noise in the training data (overfitting).
- To assess a model's generalization, we will split the data into three sets: training, validation, and testing [31-38]. The training set is used to train the model, the validation set is used to tune the model's hyperparameters and prevent overfitting, and the test set is used to evaluate the model's performance as it simulates real-world, unseen data.
- As we measure the success of our models, performance measures come into play. Common metrics [94-99] include accuracy, precision, recall, the F1 score for classification tasks, and mean squared error or mean absolute error for regression tasks. We will explore how these metrics can guide hyperparameter tuning.
- The practice of tuning hyperparameters is essential for optimizing model performance [100-108]. Techniques such as grid search and random search are popular methods for exploring the hyperparameter space. Grid search evaluates the model across a grid of hyperparameter combinations, while random search randomly selects combinations, offering a balance between exploration and exploitation.
- Gaussian processes (GPs) [109-126] are a probabilistic model used in machine learning to predict the distribution of possible outcomes rather than just the best estimate. GPs are particularly useful for understanding model uncertainty, which can be leveraged for Bayesian optimization (BO) in hyperparameter tuning.
- Further enhancing our toolkit for hyperparameter optimization, we will introduce tuning hyperparameters with BO. BO is a strategy for the global optimization of objective functions that are noisy, expensive to evaluate, or have no closed form. It is particularly useful for tuning hyperparameters in scenarios where

evaluations are costly or time-consuming. BO uses past evaluation results to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function.

- Lastly, we will discuss Acquisition Functions (ACFs). ACFs in BO are used to select the next set of hyperparameters to evaluate. Common ACFs include Expected Improvement (EI), Probability of Improvement (PI), and Upper Confidence Bound (UCB). These functions help in deciding which hyperparameter settings are likely to yield improvements over the best current observations.

In this chapter, by leveraging Mathematica's advanced functionalities, we will explore various techniques to optimize our models and enhance their predictive capabilities. The chapter is structured into five distinct units, each designed to progressively deepen the reader's understanding and proficiency in constructing and optimizing NNs using Mathematica. From exploring the fundamental challenges of overfitting to mastering advanced techniques like BO, this chapter provides a step-by-step guide to refining predictive models and enhancing their performance in real-world scenarios.

- **UNIT 7.1: The Fundamentals of Overfitting: What It Is and Why It Happens**
This unit begins by addressing the fundamental concept of overfitting, a common pitfall in NN training. Through illustrative examples and Mathematica simulations, we will learn how to recognize overfitting.
- **UNIT 7.2: Performance Metrics**
Understanding how to evaluate a NN is crucial. Performance metrics are the lens through which the effectiveness of NNs is viewed. This unit focuses on the various performance metrics available in Mathematica. For regression, common metrics include mean squared error, mean absolute error, root mean squared error, and R-squared. For classification, common metrics include accuracy, precision, recall (sensitivity), F1 score, and confusion matrix. During the training process, you can use the **TrainingProgressMeasurements** option in Mathematica to monitor these metrics and track the progress of your model. This provides valuable feedback on how well the model is learning and allows you to make adjustments as needed to improve its performance.
- **UNIT 7.3: Gaussian Processes Implementation in Mathematica from Scratch**
GPs offer a robust probabilistic approach to modeling in machine learning. GPs indeed serve as a cornerstone in BO. This unit demonstrates how to implement GPs in Mathematica from scratch to provide insightful predictions and uncertainty estimations.
- **UNIT 7.4: Setting Up BO in Mathematica**
The journey through NN optimization would be incomplete without addressing BO. BO is a powerful technique for optimizing expensive, black-box functions. With Mathematica's **BayesianMinimization** and **BayesianMaximization** functions, we will explore how to apply this sophisticated approach to streamline the hyperparameter tuning process. Additionally, utilizing Mathematica's **GaussianProcess** method, we will demonstrate how GPs can be implemented, in a simple way, to make predictive models.
- **UNIT 7.5: Automated Hyperparameter Tuning with Mathematica**
Finally, the chapter concludes by discussing automated hyperparameter tuning techniques, harnessing Mathematica's capabilities to simplify and accelerate the search for optimal model parameters. Techniques like grid search and random search, implemented through Mathematica's versatile programming environment, will be showcased. Moreover, using Mathematica's built-in functions **BayesianMinimization** and **BayesianMaximization**, we will explore how to automate the search for the best model parameters, thereby simplifying the model-building process and improving performance.

By the end of this chapter, readers will be equipped with a deep understanding and practical skills to leverage Mathematica for building and optimizing NNs, ready to tackle complex real-world data challenges.

Unit 7.1

The Fundamentals of Overfitting: What It Is and Why It Happens

Mathematica Code 7.1

```

Input      (* The code aims to analyze the performance of polynomial regression models in
           approximating a sinusoidal function with added noise. It generates a set of training
           data points by sampling from the sinusoidal function with random noise.
           Subsequently, polynomials of different degrees (0,1,3, and 9) are fitted to the
           training data to capture the underlying relationship between the input variable
           (x) and the target variable (y). The code then visualizes the training data along
           with the original sinusoidal function curve to understand the data distribution.
           Additionally, separate plots are generated for each fitted polynomial curve to
           assess their performance in approximating the underlying function. Through this
           analysis, the code aims to evaluate the trade-off between model complexity and
           goodness of fit, providing insights into the effectiveness of polynomial regression
           models for the given dataset: *)

(* Generate random seed for reproducibility: *)
SeedRandom[134];

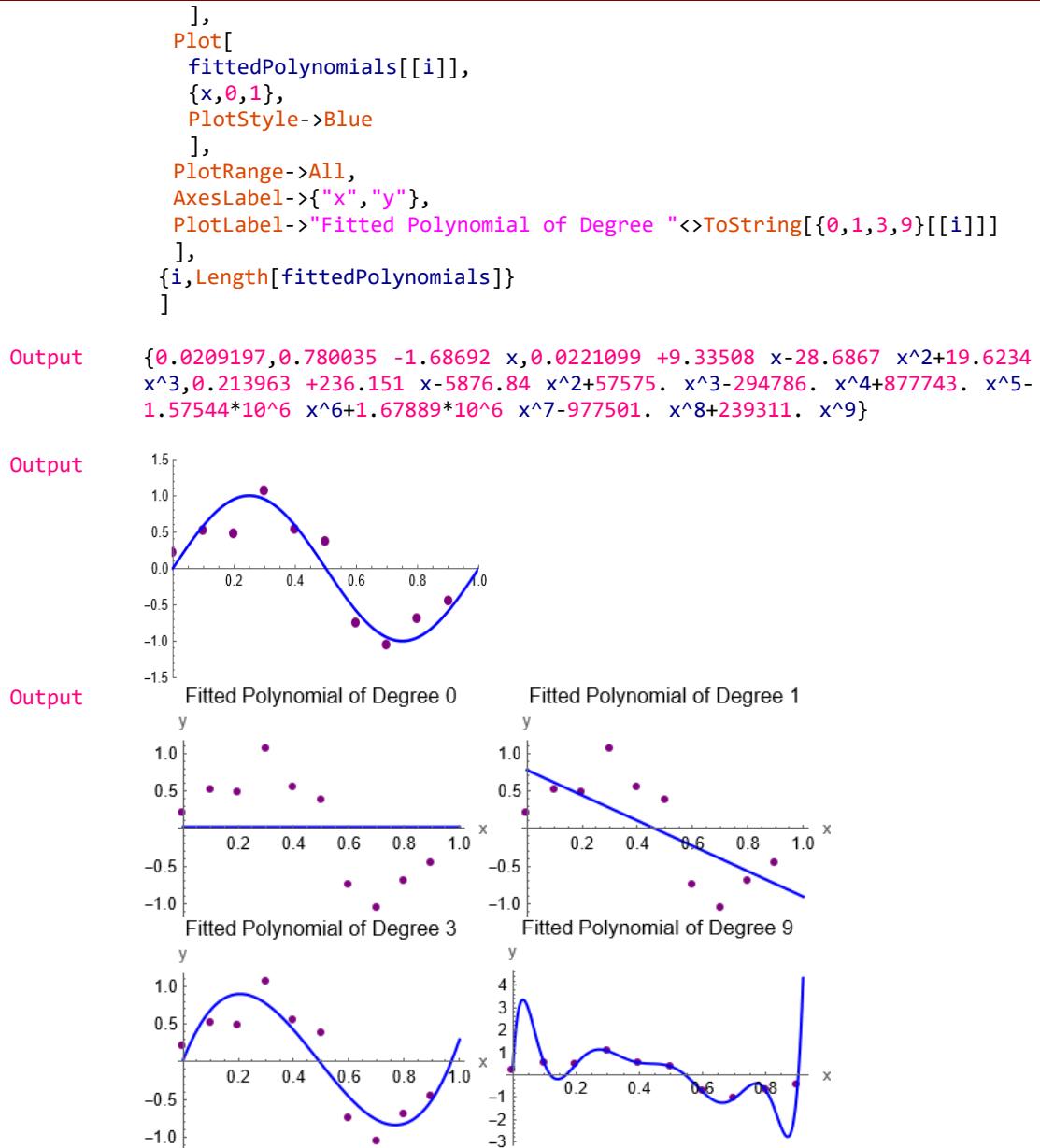
(* Generate training data: sinusoidal function with added random noise. *)
trainingData=Table[
  {x,Sin[2 π x]+RandomVariate[NormalDistribution[0,0.3]]},
  {x,0,0.9,0.1}
];

(* Fit polynomials of different degrees to the training data: *)
fittedPolynomials=Table[
  Fit[trainingData,Table[x^i,{i,0,m}],x],
  {m,{0,1,3,9}}
]

(* Plot the training data with the original sinusoidal function curve: *)
ListPlot[
  trainingData,
  PlotStyle->{PointSize[0.025],Purple},
  PlotRange->{{0,1},{-1.5,1.5}},
  FrameLabel->{"x","y"},
  AxesOrigin->{0,0},
  ImageSize->270,
  Epilog->{Blue,Thick,Line[Transpose[{Range[0,1,0.01],Sin[2 π
Range[0,1,0.01]]}]]}
]

(* Plot each fitted polynomial separately: *)
Table[
  Show[
    ListPlot[
      trainingData,
      PlotStyle->{PointSize[0.025],Purple},
      PlotRange->All,
      ImageSize->200
  ]
]

```

**Mathematica Code 7.2**

Input (* The code aims to assess the performance of polynomial regression models by evaluating the root mean square (RMS) errors for varying polynomial degrees using both training and test datasets. It begins by generating a synthetic dataset composed of sinusoidal data points with added random noise. Subsequently, it defines a function to compute the residual sum of squares (RSS) to quantify the discrepancy between the fitted polynomial curves and the actual data. After generating a finer-resolution test dataset, the code fits polynomial models of increasing degrees to the training data and calculates the RMS errors for both training and test datasets. The resulting RMS error values are plotted against the polynomial degrees, providing insights into the trade-off between model complexity and generalization performance, aiding in the selection of an optimal model complexity to avoid underfitting or overfitting: *)

```
SeedRandom[12];
```

```

(* Generate training dataset with sinusoidal function and noise: *)
trainingData=Table[
  {x,Sin[2 π x]+RandomVariate[NormalDistribution[0,0.3]]},
  {x,0,1,0.1}
];

(* Define a function to calculate Root Mean Square Error (RSS): *)
calculateRSS[data_,fit_]:=Sqrt[ (1/Length[data])*Total[(fit-data[[All,2]])^2]];

(* Generate test dataset with finer resolution: *)
testData=Table[
  {x,Sin[2 π x]+RandomVariate[NormalDistribution[0,0.2]]},
  {x,0,1,0.01}
];

(* Define polynomial orders to test: *)
orders=Range[0,10];

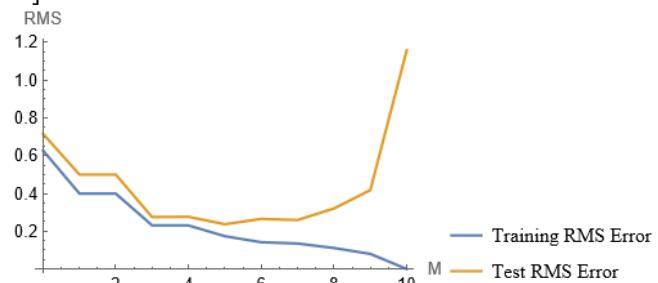
(* Calculate RMS for training data for each polynomial order: *)
trainingRMS=Table[
  (* Fit a polynomial of degree m to the training data: *)
  fit=Fit[trainingData,Table[x^i,{i,0,m}],x];
  (* Generate fitted values for each training data point: *)
  fitValues=Table[fit,{x,trainingData[[All,1]]}];
  (* Calculate RSS between fitted values and actual training data: *)
  calculateRSS[trainingData,fitValues],
  {m,orders}
];

(* Calculate RMS for test data for each polynomial order: *)
testRMS=Table[
  (* Fit a polynomial of degree m to the training data: *)
  fit=Fit[trainingData,Table[x^i,{i,0,m}],x];
  (* Generate fitted values for each test data point: *)
  fitValues=Table[fit,{x,testData[[All,1]]}];
  (* Calculate RSS between fitted values and actual test data: *)
  calculateRSS[testData,fitValues],
  {m,orders}
];

(* Plot the results: *)
ListlinePlot[
  (* Plot both training and test RMS errors: *)
  {Transpose[{orders,trainingRMS}],Transpose[{orders,testRMS}]},
  PlotLegends->{"Training RMS Error","Test RMS Error"},
  AxesLabel->{"M","RMS"},
  PlotRange->All,
  ImageSize->250
]

```

Output



Mathematica Code 7.3

```

Input      (* The goal of this code is to illustrate how the behavior of fitted polynomial
           curves changes as the size of the dataset varies. By generating datasets of
           different sizes and fitting polynomial curves of the same degree to each dataset,
           the code aims to demonstrate the relationship between dataset size and model
           complexity. Specifically, it seeks to show that as the dataset size increases, the
           overfitting problem becomes less severe, allowing for the fitting of more complex
           models without encountering significant overfitting. The code generates synthetic
           data points simulating a sinusoidal function with added random noise, creating two
           datasets with different step sizes. It then fits polynomial curves of degree 9 to
           each dataset and visualizes the data points alongside the corresponding fitted
           curves: *)

(* Seed the random number generator for reproducibility: *)
SeedRandom[134];

(* Function to generate data points: *)
generateData[stepSize_]:=Table[
  (* Generate data points with random noise: *)
  {x, Sin[2 π x]+RandomVariate[NormalDistribution[0,0.3]]},
  (* Iterate over x values with the specified step size: *)
  {x,0,0.9,stepSize}
]

(* Function to fit a polynomial to the data: *)
(* Use built-in function Fit to fit a polynomial of given degree: *)
fitPolynomial[data_,degree_]:=Fit[data,Table[x^i,{i,0,degree}],x]

(* Function to plot the data along with the fitted curve: *)
plotDataAndFit[data_,fits_,label_]:=Show[
  (* Plot the data points: *)
  ListPlot[
    data,
    PlotStyle->{PointSize[0.025],Opacity[0.5],Purple},
    PlotRange->All,
    ImageSize->250
  ],
  (* Plot the fitted curve: *)
  Plot[
    Evaluate[fits],
    {x,0,1}
  ],
  (* Set the plot range and labels: *)
  PlotRange->{{0,1},{-4,4}},
  AxesLabel->{"x","y"},
  (* Set the label for the plot: *)
  PlotLabel->label
];

(* Generate data points with a step size of 0.1: *)
data1=generateData[0.1];
(* Generate data points with a step size of 0.01: *)
data2=generateData[0.01];

(* Fit a polynomial of degree 9 to data1: *)
fits1=fitPolynomial[data1,9]
(* Fit a polynomial of degree 9 to data2: *)
fits2=fitPolynomial[data2,9]

(* Plot data1 along with its fitted curve and label: *)

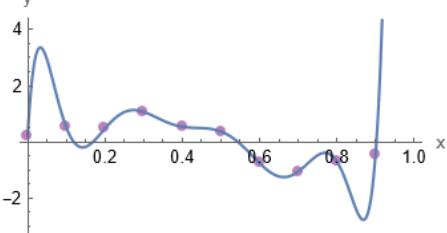
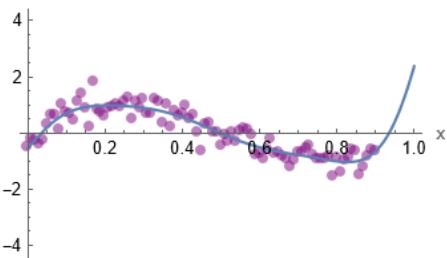
```

```

plotDataAndFit[data1,fits1,"Training Data Points 1 and its Fit: N=10, M=9"]
(* Plot data2 along with its fitted curve and label: *)
plotDataAndFit[data2,fits2,"Training Data Points 2 and its Fit: N=91, M=9"]

Output 0.213963 +236.151 x-5876.84 x^2+57575. x^3-294786. x^4+877743. x^5-1.57544*10^6
x^6+1.67889*10^6 x^7-977501. x^8+239311. x^9

Output -0.545944+14.5445 x+56.2802 x^2-1413.97 x^3+8655.16 x^4-27326.1 x^5+49215. x^6-
50767.8 x^7+27871.1 x^8-6301.36 x^9

Output Training Data Points 1 and its Fit: N=10, M=9

Output Training Data Points 2 and its Fit: N=91, M=9


```

Mathematica Code 7.4

```

Input (* The code aims to analyze the performance of polynomial regression models in
approximating a sinusoidal function with added noise. It begins by generating
multiple training datasets with varying levels of noise, using a sinusoidal function
as the underlying pattern. Then, it fits polynomials of different degrees to each
dataset, allowing for exploration of the relationship between model complexity and
goodness of fit. The fitted polynomials are visualized alongside the original
training data, aiding in the assessment of how well the models capture the
underlying sinusoidal function. Through this analysis, the code aims to provide
insights into the effectiveness of polynomial regression models in capturing
complex relationships while navigating the trade-off between model complexity and
accuracy: *)

(* Define a function to generate training data with specified noise level: *)
generateTrainingData[noise_]:=Table[
  {x, Sin[2 π x]+RandomVariate[NormalDistribution[0,noise]]},
  {x,0,0.9,0.1}
]

(* Define a function to fit polynomials of different degrees: *)
fitPolynomials[data_,degrees_]:=Table[
  Fit[data,Table[x^i,{i,0,m}],x],
  {m,degrees}
]

(* Define a function to plot training data and fitted polynomials: *)
plotFittedPolynomials[trainingData_,fittedPolynomials_,degrees_,dataNo_]:=Show[

```

```

ListPlot[
  trainingData,
  PlotStyle -> {PointSize[0.025], Purple},
  PlotRange -> All,
  ImageSize -> 250
],
Plot[
  Evaluate[fittedPolynomials],
  {x, 0, 1},
  PlotLegends -> Placed[{"M=1", "M=3", "M=9"}, {0.7, 0.8}]
],
PlotRange -> Full,
AxesLabel -> {"x", "y"},
PlotLabel -> "Fitted Polynomial of Degree " <> ToString[degrees] "\n for Dataset "
<> ToString[dataNo]
]

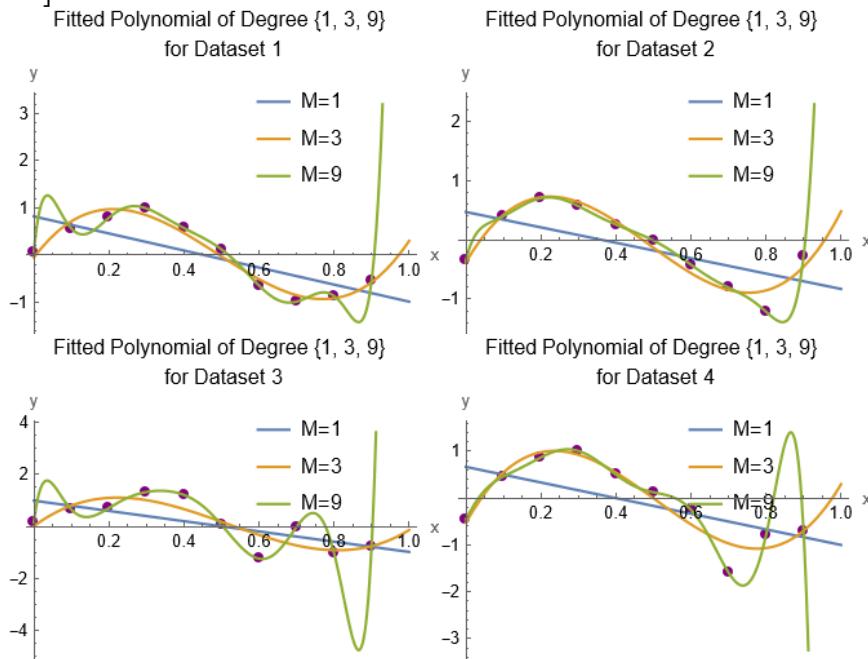
(* Generate random seed for reproducibility: *)
SeedRandom[134];

(* Generate training data with different noise levels: *)
trainingDatas = Table[generateTrainingData[i], {i, {0.1, 0.2, 0.3, 0.4}}];

(* Fit polynomials of different degrees to the training data and plot: *)
fittedPolynomials = Table[
  fitPolynomials[trainingData, {1, 3, 9}],
  {trainingData, trainingDatas}
];
plots = MapThread[
  plotFittedPolynomials,
  {
    trainingDatas,
    fittedPolynomials,
    {{1, 3, 9}, {1, 3, 9}, {1, 3, 9}, {1, 3, 9}},
    {1, 2, 3, 4}
  }
]

```

Output



Mathematica Code 7.5

```

Input (* The code generates synthetic noisy data representing a mathematical function
   and trains a multilayer perceptron (MLP) neural network on this data. The MLP
   architecture comprises two hidden layers with 150 units each, utilizing hyperbolic
   tangent activation functions. Following training, the network's performance is
   evaluated on a separate test dataset to assess its generalization capabilities.
   Despite achieving low loss on the training data, the trained model demonstrates
   poor performance on the test set, indicating potential overfitting. This disparity
   is visually demonstrated by comparing the model's predictions to both the training
   and test data, highlighting the need for careful evaluation of model generalization
   beyond training performance: *)

(* Generate synthetic noisy training data based on a Gaussian function and add
Gaussian noise: *)
trainingData=Table[
  (* Apply noise with standard deviation of 0.15: *)
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,.15]],
  {x,-3,3,.2}
];

(* Visualize the training data and the noise-free original function for comparison:
*)
(* Plot the training data with red points: *)
trainingPlot=ListPlot[
  List@@@trainingData,
  PlotStyle->Red,
  ImageSize->250,
  AxesLabel->{"x", "y"},
  PlotLabel->"Training Data"
];

(* Plot the original function without noise using a blue line: *)
originalCurve=Plot[
  Exp[-x^2],
  {x,-3,3},
  PlotStyle->Blue,
  ImageSize->250,
  AxesLabel->{"x", "y"},
  PlotLabel->"Original Curve with Noise"
];
(* Combine the plots to show both the noisy training data and the original
function: *)
Show[originalCurve,trainingPlot]

(* Define and train a multilayer perceptron neural network: *)
(* Create an MLP with two hidden layers, each with 150 units using Tanh activation:
*)
mlp=NetChain[{150,Tanh,150,Tanh,1}];

(* Train the MLP on the generated training data, limiting the training time to 30
seconds: *)
trainingResults=NetTrain[mlp,trainingData,All,TimeGoal->30]

(* Display the final loss after training to assess fit quality: *)
finalLoss=trainingResults[ "RoundLoss"]

(* Extract the trained network for further evaluation: *)
trainedNet=trainingResults[ "TrainedNet"]

(* Plot predictions from the trained model against the original training data: *)

```

```

predictedTrainingPlot=Plot[
  trainedNet[x],
  {x,-3,3},
  ImageSize->250,
  AxesLabel->{"x", "y"},
  PlotLabel->"Trained Model Prediction"
];
Show[predictedTrainingPlot,trainingPlot]

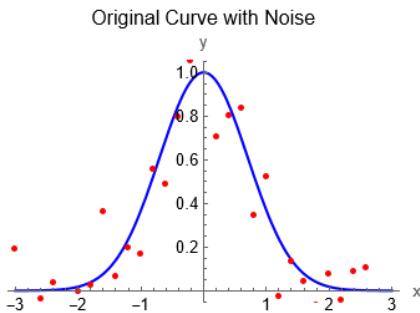
(* Generate a separate test dataset with higher noise to evaluate generalization:
*)
testData=Table[
  (* Apply higher noise with standard deviation of 0.35: *)
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,.35]],
  (* Same range and steps as training data: *)
  {x,-3,3,.2} ];
testX=Keys[testData];
testY=Values[testData];

(* Visualize how the trained model performs on new, unseen (test) data: *)
predictedTestPlot=Plot[
  trainedNet[x],
  {x,-3,3},
  ImageSize->250,
  AxesLabel->{"x", "y"},
  PlotLabel->"Trained Model Prediction on Test Data"
];

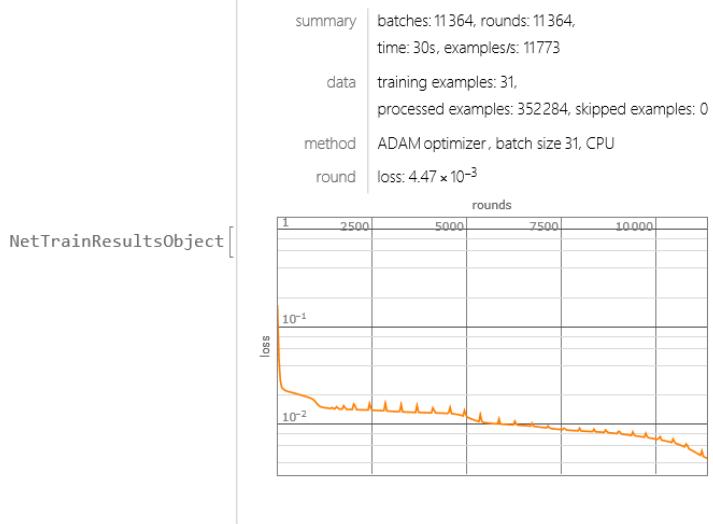
(* Plot the test data with orange points for comparison: *)
testPlot=ListPlot[
  List@@@testData,
  PlotStyle->Orange,
  ImageSize->250,
  AxesLabel->{"x", "y"},
  PlotLabel->"Test Data"
];

(* Show both the test data and the model's predictions on the same plot to evaluate
accuracy: *)
Show[predictedTestPlot,testPlot]

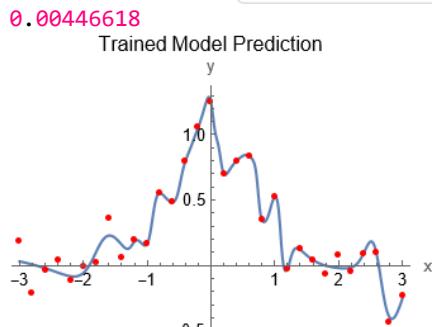
```

Output

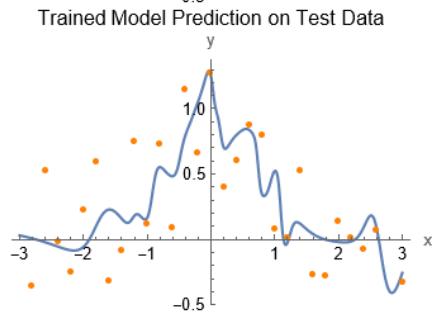
Output



Output
Output



Output



Unit 7.2

Performance Metrics

In Mathematica, `ValidationSet` is an option used in functions like `NetTrain`, `Predict`, and `Classify`. It's utilized for specifying a dataset that the neural network or machine learning model will not use for training but instead use for validation during the training process. This helps in assessing how well the model generalizes to unseen data and avoids overfitting.

`TrainingProgressMeasurements` is another option used in functions like `NetTrain` to specify what training progress metrics should be computed and returned during the training process. These metrics can include things like loss values, accuracy, etc. It allows you to monitor the progress of your training and make decisions accordingly, such as whether to stop training or adjust parameters.

`NetTrainResultsObject` is the output object returned by the `NetTrain` function in Mathematica after training a neural network. It contains various pieces of information about the training process and the trained network, such as final training and validation loss, training time, the trained neural network itself, etc. This object can be further used for evaluation, testing, or deployment of the trained model.

`ValidationMeasurements` is an option used in functions like `Predict`, `Classify`, and `NetTrain`. It allows you to specify what measurements you want to compute on the validation set during the evaluation of a trained model.

1. Validation Set

`ValidationSet`

is an option for `Predict`, `Classify`, `NetTrain`, and related functions that specifies the validation set to be used during the training phase.

Remarks:

- With `ValidationSet -> data`, model and hyperparameter selections are done by testing performance on `data`. `data` can be given in any format allowed for the training set.
- With `ValidationSet -> Automatic`, cross-validation methods on the original data supplied to `Predict`, `Classify`, etc. will be used instead.
- `ValidationSet -> data` is typically used when the data in the training set and the data that one wishes to predict or classify come from different sources.
- If a validation set is specified, `NetTrain` will return the net that produced the lowest validation loss during training with respect to this set.

The following settings for `ValidationSet` can be given:

<code>None</code>	use only the existing training set to estimate loss (default)
<code>data</code>	validation set in the same form as training data
<code>Scaled[frac]</code>	reserve a specified fraction of the training set for validation
<code>{spec, "Interval" -> int}</code>	specify the interval at which to calculate validation loss

2. Training Progress Measurements

`TrainingProgressMeasurements`

is an option for `NetTrain` that specifies measurements to make while training is in progress.

For nets that contain a `CrossEntropyLossLayer`, the following built-in measurements are available:

<code>"Accuracy"</code>	fraction of correctly classified examples
<code>"Accuracy" -> n</code>	fraction of examples with the correct result in the top n

"AreaUnderROCCurve"	area under the ROC curve for each class
"CohenKappa"	Cohen's kappa coefficient
"ConfusionMatrix"	counts c_{ij} of class i examples classified as class j
"ConfusionMatrixPlot"	plot of the confusion matrix
"Entropy"	entropy measured in nats
"ErrorRate"	fraction of incorrectly classified examples
"ErrorRate"->n	fraction of examples with the incorrect result in the top n
"F1Score"	F1 score for each class
"FScore"-> β	$F\beta$ score for each class
"FalseDiscoveryRate"	false discovery rate for each class
"FalseNegativeNumber"	number of false negative examples
"FalseNegativeRate"	false negative rate for each class
"FalseOmissionRate"	false omission rate for each class
"FalsePositiveNumber"	number of false positive examples
"FalsePositiveRate"	false positive rate for each class
"Informedness"	informedness for each class
"Markedness"	markedness for each class
"MatthewsCorrelationCoefficient"	Matthews correlation coefficient for each class
"NegativePredictiveValue"	negative predictive value for each class
"Perplexity"	exponential of the entropy
"Precision"	precision for each class
"Recall"	recall rate for each class
"ROCCurve"	receiver operating characteristics (ROC) curve for each class
"ROCCurvePlot"	plot of the ROC curve
"ScottPi"	Scott's pi coefficient
"Specificity"	specificity for each class
"TrueNegativeNumber"	number of true negative examples
"TruePositiveNumber"	number of true positive examples

For nets that contain a `MeanSquaredLossLayer` or `MeanAbsoluteLossLayer`, the following built-in measurements are available:

"FractionVarianceUnexplained"	the fraction of output variance left unexplained by the net
"IntersectionOverUnion"	intersection over union for bounding boxes
"MeanDeviation"	mean absolute value of the residuals
"MeanSquare"	mean square of the residuals
"RSquared"	coefficient of determination
"StandardDeviation"	root mean square of the residuals

Remarks:

- `NetTrain[net,data,All]` returns a `NetTrainResultsObject[...]` that contains values for all properties that do not require significant additional computation or memory.
- `NetTrainResultsObject[...][prop]` is used to look up property `prop` from the `NetTrainResultsObject`.
- Associations of the final value of all measurements can be obtained after training by specifying `"RoundMeasurements"` and `"ValidationMeasurements"` as properties in `NetTrain[net,data,properties]` or in a `NetTrainResultsObject`.

Remember, in `NetTrain[net,data,prop]`, the property `prop` can be any of the following:

"BestValidationRound"	the training round corresponding to the final trained net
"ValidationExamples"	the number of examples in the validation set
"ValidationLoss"	the mean loss obtained on the ValidationSet for the most recent validation measurement
"ValidationLossList"	list of the mean losses on the ValidationSet for each validation measurement

"ValidationMeasurements"	association of training measurements on the ValidationSet after the most recent validation measurement
"ValidationMeasurementsLists"	list of training measurements associations on the ValidationSet for each validation measurement
"ValidationPositions"	the batch numbers corresponding to each validation measurement

Remark:

NetTrain[net,data,{prop1,prop2,...}] returns a list of the results for the propi.

Mathematica Code 7.6

```

Input (* The code is designed to create synthetic data from a Gaussian distribution,
       construct a neural network with two hidden layers, and train this network using
       the generated data with added noise. The main objectives are to accurately model
       the underlying data patterns, optimize the network parameters for best performance,
       and ensure the model's generalization to new, unseen data through rigorous training
       and validation. The training process, limited to 50 rounds, includes monitoring
       metrics such as mean deviation, mean square error, R-squared, and standard deviation
       to assess accuracy and prevent overfitting, providing a comprehensive approach to
       training and evaluating the neural network's performance. All the results from the
       training session are stored in the NetTrainResultsObject. This object provides a
       convenient way to access various details about the training process, including
       performance metrics, the best model configuration, and other properties that can
       help in analyzing and understanding the behavior and effectiveness of the trained
       neural network. You can query this object for specific details like validation
       losses, training metrics, and the final trained model, which are crucial for further
       evaluation and refinement of your model: *)

(* Define the underlying data function as a Gaussian distribution: *)
dataFunction[x_]:=Exp[-x^2];

(* Set the noise level for generating synthetic data: *)
noiseLevel=0.15;

(* Generate training data: sample points from the defined function and add Gaussian
   noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Generate validation data in the same manner but with fewer points: *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

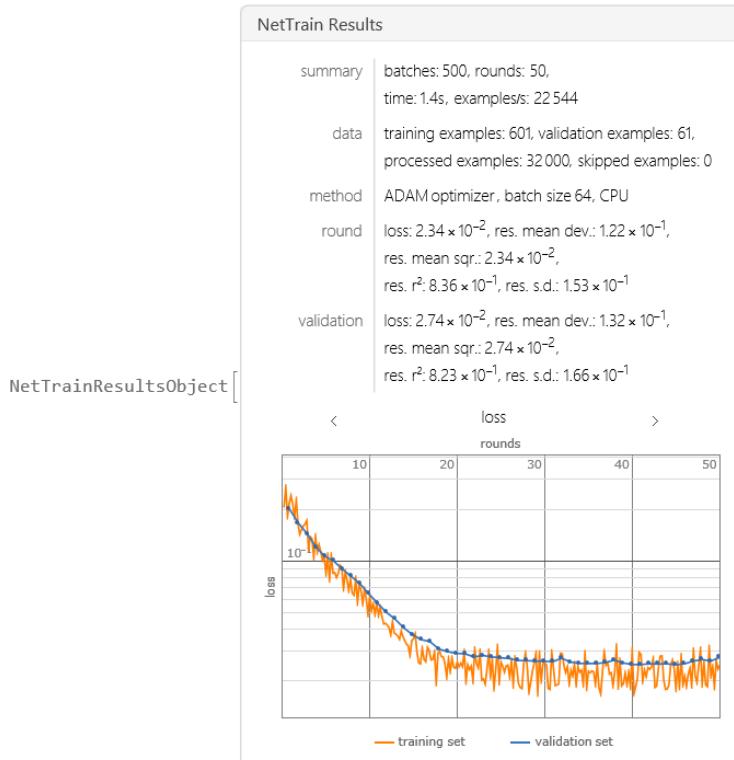
(* Define a neural network with two hidden layers and one output layer: *)
net=NetChain[
  {
    (* First hidden layer with 50 neurons and Tanh activation: *)
    LinearLayer[50],Tanh,
    (* Second hidden layer with 30 neurons and Tanh activation: *)
    LinearLayer[30],Tanh,
    (* Output layer with 1 neuron: *)
    LinearLayer[1]
  }
];

(* Obtain a NetTrainResultsObject for a training session: *)

```

```
netTrainResults=NetTrain[
  net,
  trainingData,
  All,
  ValidationSet->validationData,
  TrainingProgressMeasurements-
>{"MeanDeviation", "MeanSquare", "RSquared", "StandardDeviation"},  

  MaxTrainingRounds->50
]
```

Output**Mathematica Code 7.7**

```
Input (* Access various details about the training process from NetTrainResultsObject: *)
(* Define the underlying data function as a Gaussian distribution: *)
dataFunction[x_]:=Exp[-x^2];

(* Set the noise level for generating synthetic data: *)
noiseLevel=0.15;

(* Generate training data: sample points from the defined function and add
Gaussian noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Generate validation data in the same manner but with fewer points: *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];
```

```
(* Generate test data in the same manner: *)
testData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Define a neural network with two hidden layers and one output layer: *)
net=NetChain[
{
  (* First hidden layer with 50 neurons and Tanh activation: *)
  LinearLayer[50],Tanh,
  (* Second hidden layer with 30 neurons and Tanh activation: *)
  LinearLayer[30],Tanh,
  (* Output layer with 1 neuron: *)
  LinearLayer[1]
}
];

(* Train the network using the generated data: *)
netTrainResults=NetTrain[
  net,
  trainingData,
  All,
  ValidationSet->validationData,
  TrainingProgressMeasurements-
>{"MeanDeviation","MeanSquare","RSquared","StandardDeviation"},
  (* Limit the training to 50 rounds: *)
  MaxTrainingRounds->50
];

(* Extract and list all properties available from the training results: *)
netTrainResults["Properties"]

(* Retrieve the trained network: *)
trainedNet=netTrainResults["TrainedNet"]

(* Get the list of validation loss for each training round: *)
validationLossList=netTrainResults["ValidationLossList"];

(* Get validation measurement lists for all tracked metrics: *)
validationMeasurementsLists=Values[netTrainResults["ValidationMeasurementsLists"]]
];

(* Find the training round with the best (lowest) validation loss: *)
minPosition1=netTrainResults["BestValidationRound"]
minPosition2=Position[validationLossList,Min[validationLossList]]
minPosition3=Position[validationMeasurementsLists[[1]],Min[validationMeasurements
Lists[[1]]]]

(* Extract the minimum validation loss value from the results*)
minValidationLossValue1=Min[netTrainResults["ValidationLossList"]]
minValidationLossValue2=Min[netTrainResults["ValidationMeasurementsLists"][[1]]]

(* Get the most recent validation loss from the final training round: *)
mostRecentValidationLoss1=netTrainResults["ValidationLoss"]

(* Get the most recent validation loss and Validation Measurements from the final
training round: *)
mostRecentValidationLoss1=Values[netTrainResults["ValidationMeasurements"]]
```

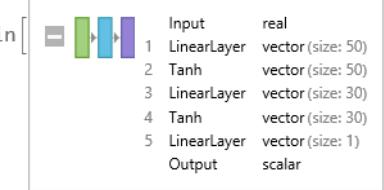
```
(* Retrieve final training plots if available: *)
netTrainResults["FinalPlots"]

(* Visualize the model's predictions along with the actual test data: *)
plotModel=Plot[
  trainedNet[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

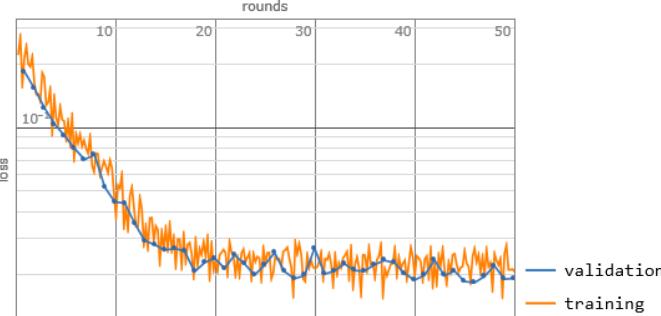
plottestData=ListPlot[
  List@@@ testData,
  PlotStyle->Orange,
  PlotLegends->{"Test Data"}
];

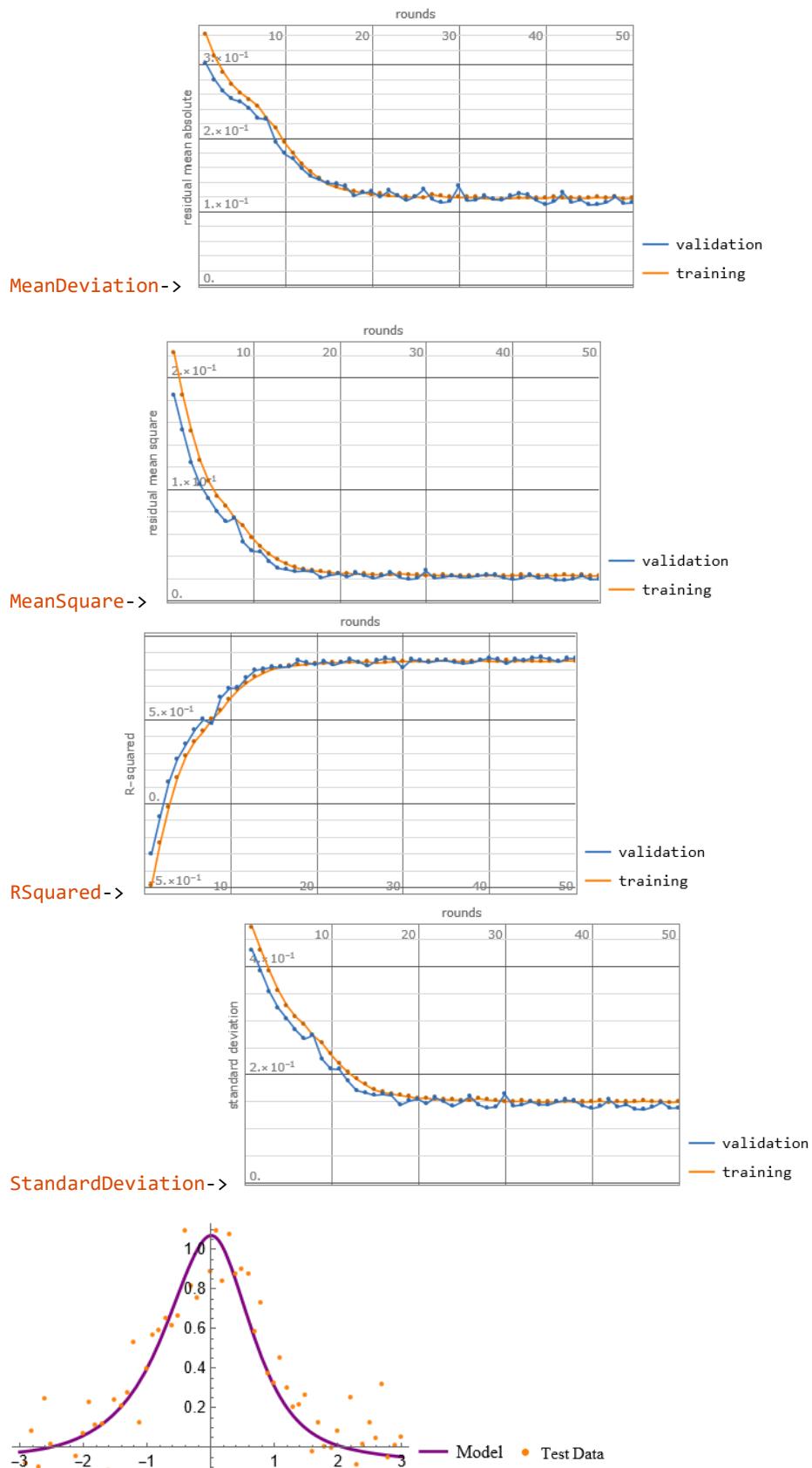
(* Display the model and test data together for comparison: *)
Show[plotModel,plottestData,ImageSize->250]

Output {ArraysLearningRateMultipliers,BatchesPerRound,BatchesPerSecond,BatchLossList,BatchMeasurements,BatchMeasurementsLists,BatchSize,BestValidationRound,CheckpointingFiles,ExamplesProcessed,FinalLearningRate,FinalPlots,InitialLearningRate,InternalVersionNumber,LossPlot,MeanBatchesPerSecond,MeanExamplesPerSecond,NetTrainInputForm,OptimizationMethod,ReasonTrainingStopped,RoundLoss,RoundLossList,RoundMeasurements,RoundMeasurementsLists,RoundPositions,SkippedTrainingData,TargetDevice,TotalBatches,TotalRounds,TotalTrainingTime,TrainedNet,TrainingExamples,TrainingNet,TrainingUpdateSchedule,ValidationExamples,ValidationLoss,ValidationLossList,ValidationMeasurements,ValidationMeasurementsLists,ValidationPositions}

Output NetChain[]
  Input      real
  1 LinearLayer vector (size: 50)
  2 Tanh      vector (size: 50)
  3 LinearLayer vector (size: 30)
  4 Tanh      vector (size: 30)
  5 LinearLayer vector (size: 1)
  Output     scalar

Output 46
Output {{46}}
Output {{46}}
Output 0.0183498
Output 0.0183498
Output 0.0192765
Output {0.0192765, 0.111954, 0.0192765, 0.86395, 0.13884}

Output 
```



Mathematica Code 7.8

Input	(* Similar to the previous code, this version accesses various details about the training process by utilizing 'ValidationMeasurements': *) (* This code explicitly specifies "ValidationMeasurements" as a property to retrieve in the NetTrain function. This shows a clear intent to focus on and directly access detailed validation metrics throughout the training process. This approach ensures that these specific metrics are highlighted and analyzed, which is particularly useful for a more detailed performance evaluation. Note that, the previous code does not explicitly specify a focus on validation measurements within the NetTrain function. It saves all training results in a NetTrainResultsObject, which can store a wide range of information but does not emphasize specific validation metrics unless manually accessed later: *) (* Define the underlying data function as a Gaussian distribution: *) dataFunction[x_]:=Exp[-x^2]; (* Set the noise level for generating synthetic data: *) noiseLevel=0.15; (* Generate training data: sample points from the defined function and add Gaussian noise: *) trainingData=Table[x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]], {x, -3, 3, 0.01}]; (* Generate validation data in the same manner but with fewer points: *) validationData=Table[x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]], {x, -3, 3, 0.1}]; (* Define a simple neural network structure: *) net=NetChain[{ LinearLayer[50],Tanh, LinearLayer[30],Tanh, LinearLayer[1] }; (* Associations of the final value of all measurements can be obtained after training by specifying "ValidationMeasurements" as properties in NetTrain[net, data, properties]: *) netTrainResults=NetTrain[net, trainingData, "ValidationMeasurements", ValidationSet->validationData, TrainingProgressMeasurements- >{"MeanDeviation", "MeanSquare", "RSquared", "StandardDeviation"}, MaxTrainingRounds->50]
Output	< Loss->0.0196389, MeanDeviation->0.105876, MeanSquare->0.0196389, RSquared->0.86975, StandardDeviation->0.140139 >

Mathematica Code 7.9

```

Input (* Access various details about the training process from NetTrainResultsObject: *)
(* The code aims to generate a synthetic dataset within a unit disk, label the data
based on proximity to the center, and visualize this distribution. It constructs a
simple neural network for binary classification, which includes linear and non-
linear layers to capture complex patterns. The network is trained using the
generated data, validated with a separate set, and evaluated using several
performance metrics. The training process is visualized through a contour plot
showing the decision boundaries and other training-related plots like the confusion
matrix. The goal is to demonstrate the complete workflow of data preparation, model
training, and performance evaluation in a neural network application: *)

(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less than
0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Generate 500 random points within a unit disk for validation: *)
validationPoints=RandomPoint[Disk[],500];

(* Assign labels using the same criteria as for the training set: *)
validationLabels=Thread[Map[Norm,validationPoints]<0.5];

(* Combine the points and labels into validation data: *)
validationData=validationPoints->validationLabels;

(* Visualize the synthetic training set with different colors for each class: *)
ListPlot[
{
  Pick[trainpoints,trainlabels,True],
  Pick[trainpoints,trainlabels,False]
},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},
PlotLegends->{"Class 1","Class 2"}
]

(* Define a neural network with two linear layers and activation functions: *)
net=NetChain[
{
  LinearLayer[20],ElementwiseLayer[Tanh],
  LinearLayer[],ElementwiseLayer[LogisticSigmoid],
  "Output"->NetDecoder["Boolean"]
}

(* Train the neural network on the training data with specified metrics and
validation: *)
netTrainResults=NetTrain[
  net,
  trainingData,
  All,
  ...
]

```

```

ValidationSet->validationData,
TrainingProgressMeasurements->{
    "FalseNegativeNumber", "FalsePositiveNumber", "TrueNegativeNumber",
    "TruePositiveNumber", "Accuracy", "F1Score",
    "MatthewsCorrelationCoefficient", "Precision", "Recall",
    "ScottPi", "ConfusionMatrixPlot"
}
]

(* Association of training measurements on the ValidationSet after the most recent
validation measurement: *)
netTrainResults["ValidationMeasurements"]

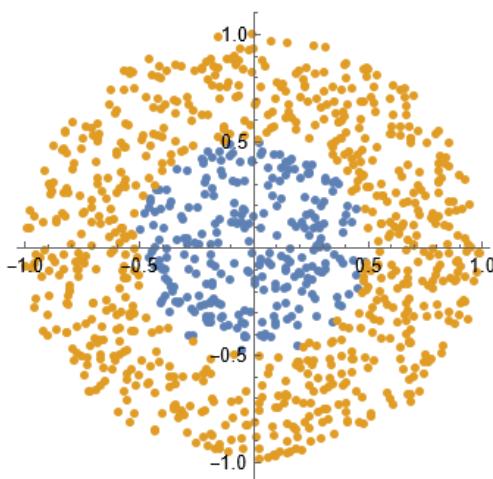
(* Extract the trained network for further use or analysis: *)
trainedNet=netTrainResults["TrainedNet"]

(* Visualize the decision boundary of the trained network: *)
ContourPlot[
    trainedNet[{x,y},None],
    {x,-1,1},
    {y,-1,1},
    ContourStyle->{White},
    ClippingStyle->Automatic,
    ColorFunction->"BlueGreenYellow",
    PlotLegends->Automatic,
    LabelStyle->Directive[Black,10],
    ImageSize->250,
    PlotLabel->"Trained Nonlinear Classifier Decision Boundary"]

(* Display additional training plots like loss and confusion matrix: *)
netTrainResults["FinalPlots"]

```

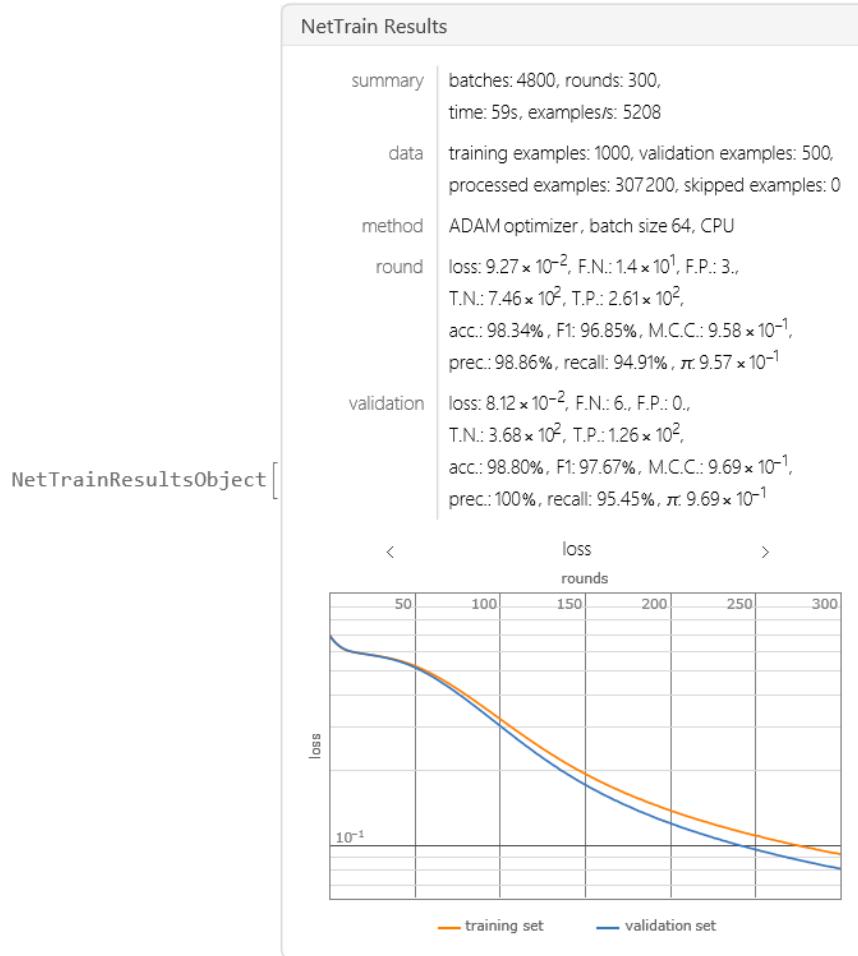
Output



Output

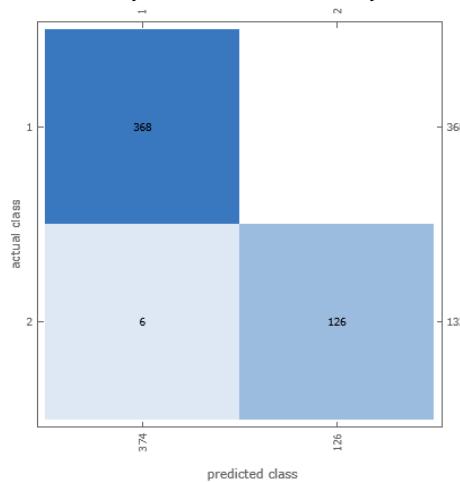
NetChain		
uninitialized	Input	array
1	LinearLayer	vector(size: 20)
2	Tanh	vector(size: 20)
3	LinearLayer	real
4	LogisticSigmoid	real
	Output	boolean

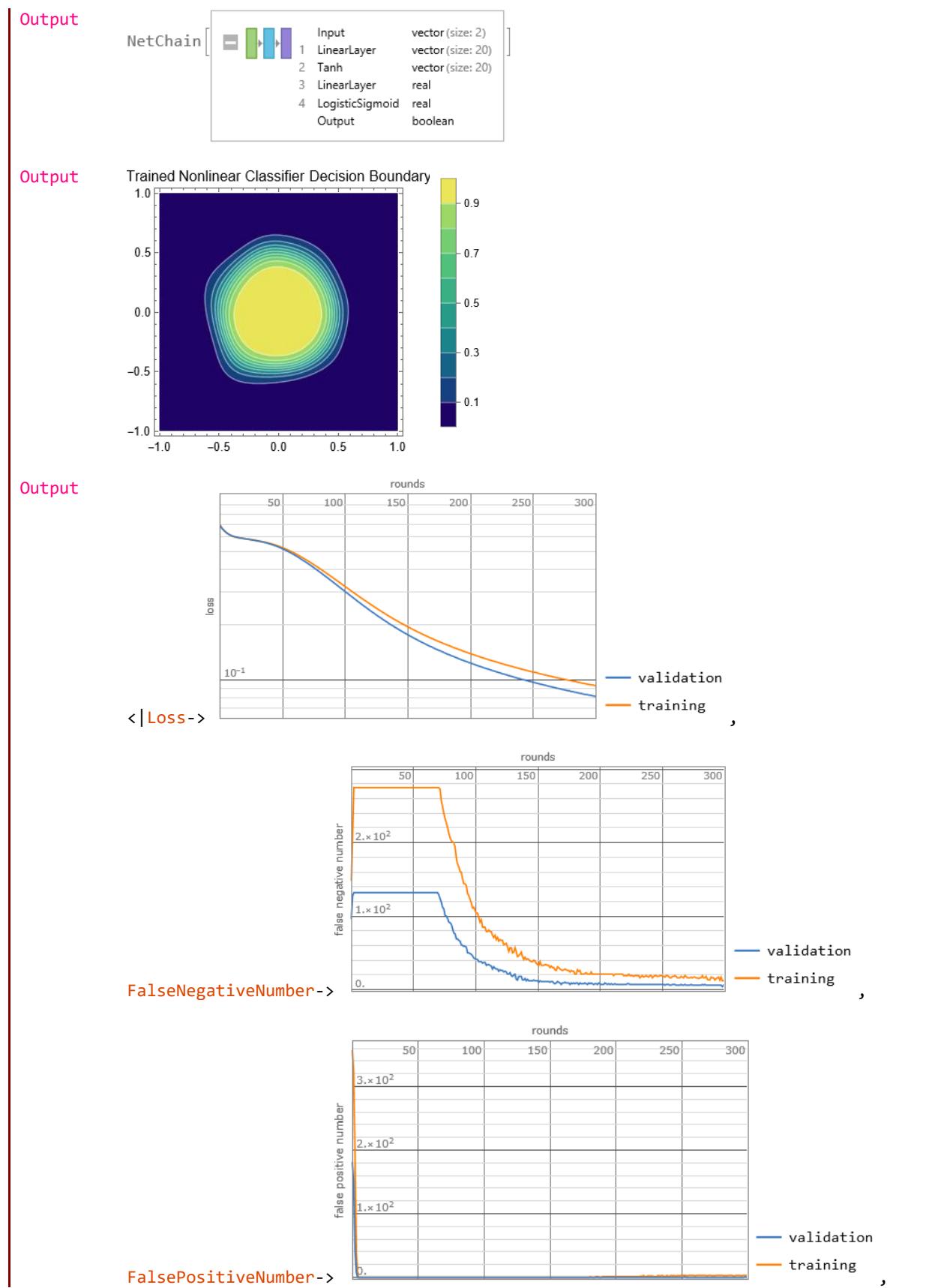
Output

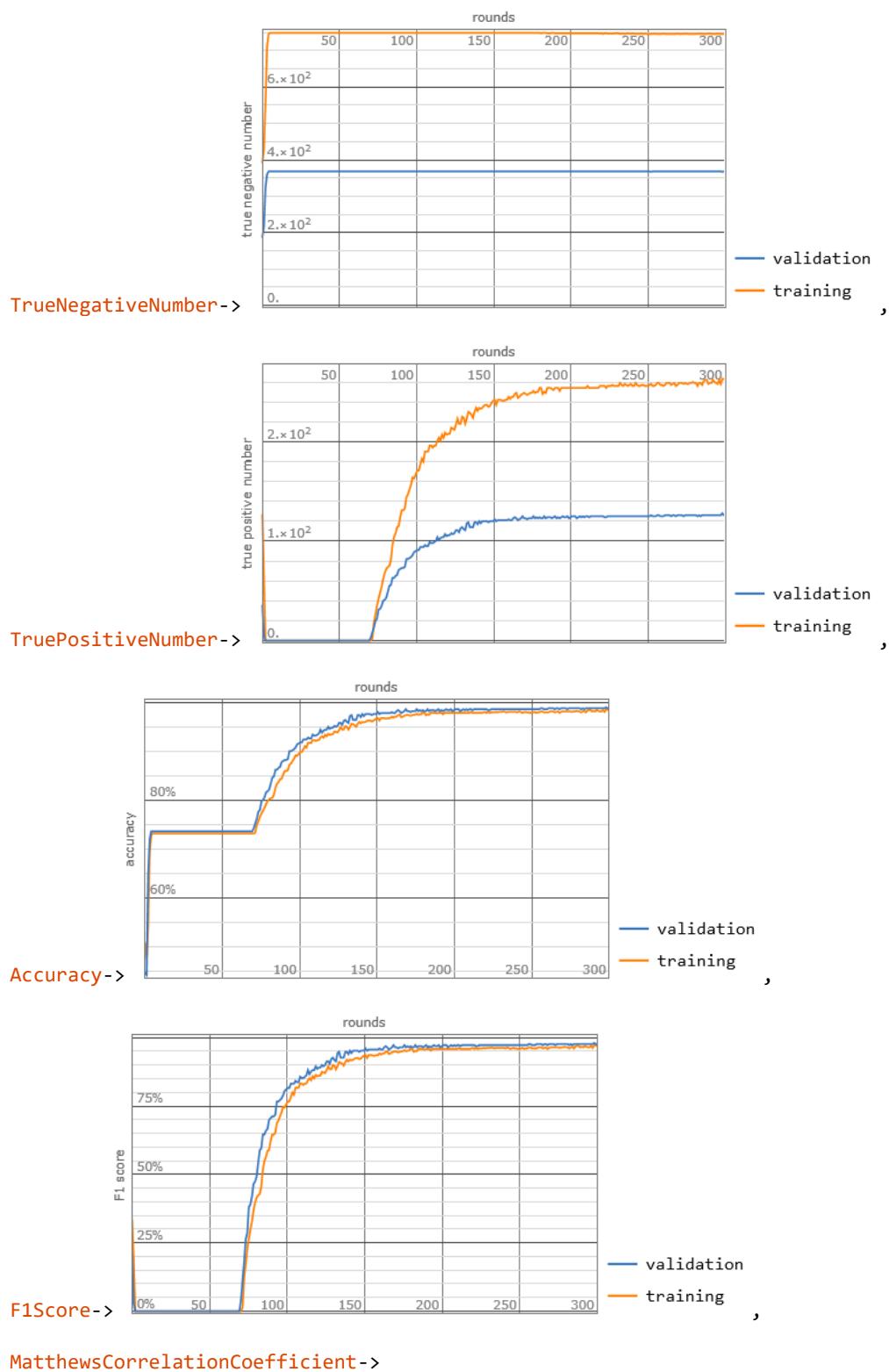


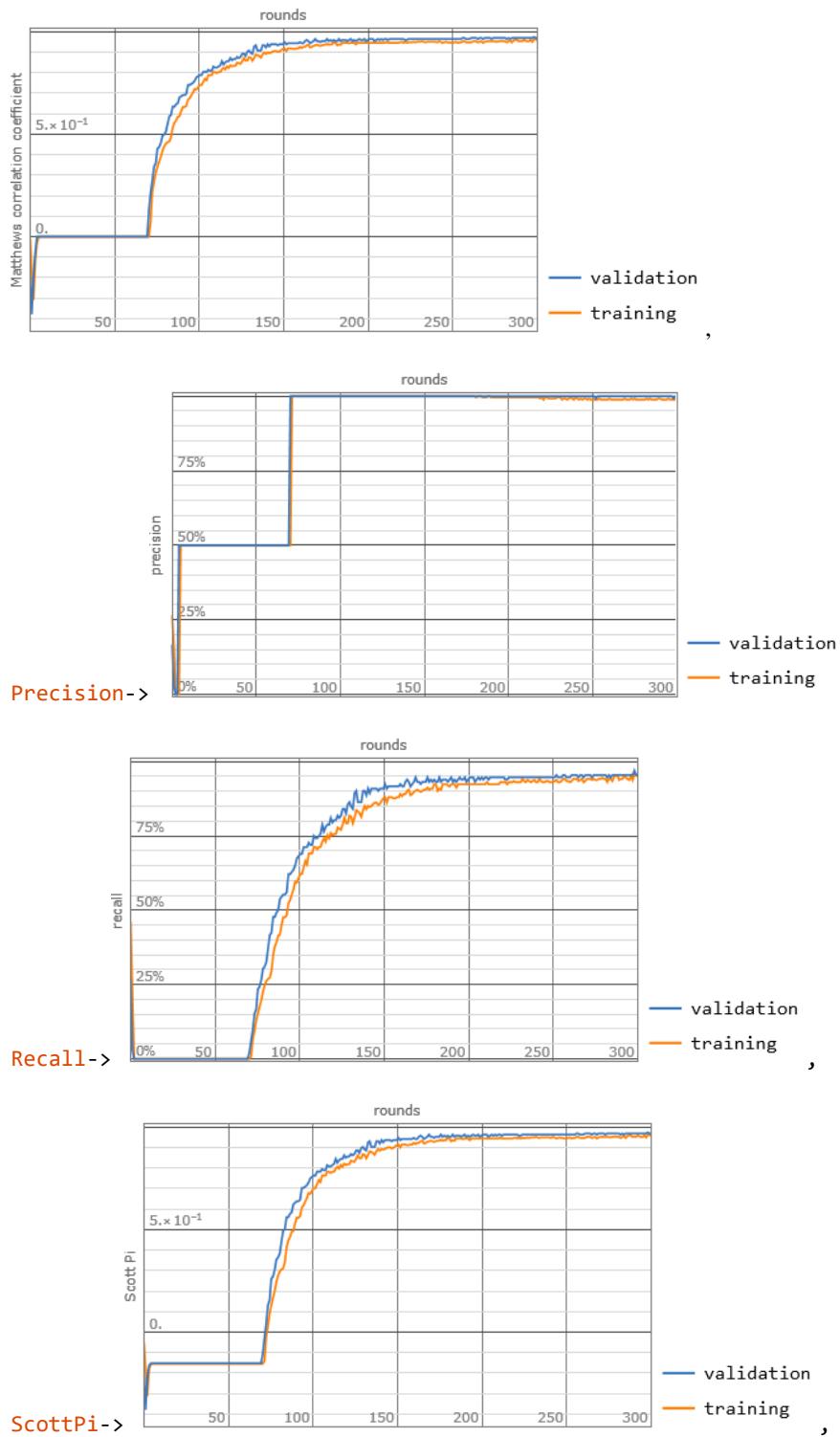
Output

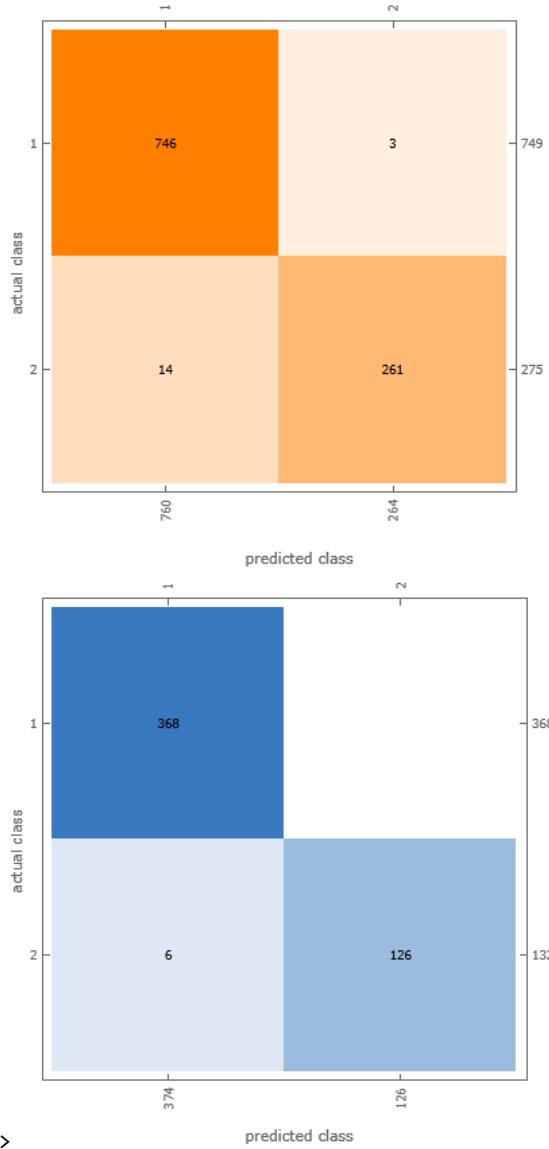
```
<| Loss->0.0811929, FalseNegativeNumber->6., FalsePositiveNumber->0., TrueNegativeNumber->368., TruePositiveNumber->126., Accuracy->0.988, F1Score->0.976744, MatthewsCorrelationCoefficient->0.96914, Precision->1., Recall->0.954545, ScottPi->0.968658, ConfusionMatrixPlot->{
```











ConfusionMatrixPlot ->

predicted class

| >

Mathematica Code 7.10

```

Input (* Similar to the previous code, this version accesses various details about the
       training process by utilizing 'ValidationMeasurements': *)

(* The introduction of "ValidationMeasurements" in NetTrain suggests a focus on
   in-depth analysis and monitoring of the model's performance specifically on the
   validation data throughout the training process. This can be critical for
   understanding how well the model generalizes to new, unseen data and for making
   adjustments to model parameters or training procedures based on validation
   feedback: *)

(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less
   than 0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

```

```

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Generate 500 random points within a unit disk for validation: *)
validationPoints=RandomPoint[Disk[],500];

(* Assign labels using the same criteria as for the training set: *)
validationLabels=Thread[Map[Norm,validationPoints]<0.5];

(* Combine the points and labels into validation data: *)
validationData=validationPoints->validationLabels;

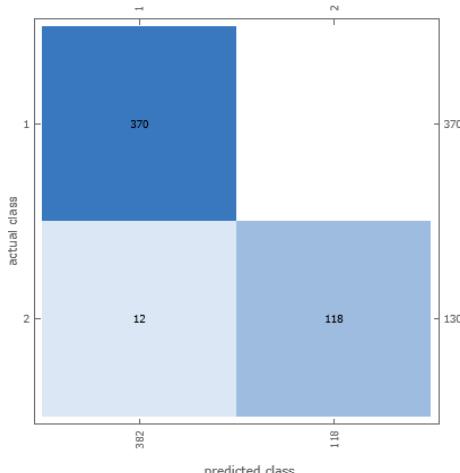
(* Define a neural network with two linear layers and activation functions: *)
net=NetChain[
{
  LinearLayer[20],ElementwiseLayer[Tanh],
  LinearLayer[],ElementwiseLayer[LogisticSigmoid]],
 "Output"->NetDecoder["Boolean"]
]

(* Train the neural network on the training data with specified metrics and
validation: *)
netTrainResults=NetTrain[
  net,
  trainingData,
  "ValidationMeasurements",
  ValidationSet->validationData,
  TrainingProgressMeasurements->{
    "FalseNegativeNumber", "FalsePositiveNumber", "TrueNegativeNumber",
    "TruePositiveNumber", "Accuracy", "F1Score",
    "MatthewsCorrelationCoefficient", "Precision", "Recall",
    "ScottPi", "ConfusionMatrixPlot"
  }
]

```

Output

<| Loss->0.110256, FalseNegativeNumber->12., FalsePositiveNumber->0., TrueNegativeNumber->370., TruePositiveNumber->118., Accuracy->0.976, F1Score->0.951613, MatthewsCorrelationCoefficient->0.937645, Precision->1., Recall->0.907692, ScottPi->0.935655, ConfusionMatrixPlot->



>

Unit 7.3

Gaussian Processes Implementation in Mathematica From Scratch

Mathematica Code 7.11

```

Input      (* The code is a demonstration of constructing and visualizing a multivariate
           normal distribution. It starts by defining a zero-mean vector and a specifically
           structured covariance matrix to reflect varying degrees of correlation between
           variables. The code visualizes this matrix to elucidate the underlying correlation
           structure and proceeds to generate random variates, which it then plots as discrete
           functions. This visualization serves to both educate and illustrate the impact of
           covariance on the behavior of a multivariate distribution: *)

(* Define a mean vector for the multivariate distribution, initializing all elements
   to zero: *)
zeroMeanVector={0,0,0,0,0};

(* Define a 5x5 covariance matrix to represent correlations between variables.
   Diagonal elements are 1, indicating a variance of 1 for each variable. Off-diagonal
   elements decrease with increasing distance, indicating varying levels of
   correlation: *)
correlationMatrix={
  {1,0.3,0.01,0,0},
  {0.3,1,0.3,0.01,0},
  {0.01,0.3,1,0.3,0.01},
  {0,0.01,0.3,1,0.3},
  {0,0,0.01,0.3,1}
};

(* Visualize the correlation matrix using MatrixPlot, highlighting the structure
   and correlation values: *)
MatrixPlot[
  correlationMatrix,
  ImageSize->250,
  PlotLabel->"Covariance Matrix",
  PlotLegends->Automatic,
  FrameTicks->Insert[
    Table[
      Transpose[{Range[5],Map[Style[#,FontFamily-
>"Times"]&,Table[Subscript[HoldForm[y],i],{i,1,5}]]}],
      2
    ],
    None,
    {{2},{2}}
  ]
]

(* Define the multivariate normal distribution using the zero mean vector and
   correlation matrix. This distribution will be used to generate random samples: *)
multivariateDistribution=MultinormalDistribution[zeroMeanVector,correlationMatrix
];

(* Set a random seed for reproducibility of the sampling process: *)

```

```

SeedRandom[1];

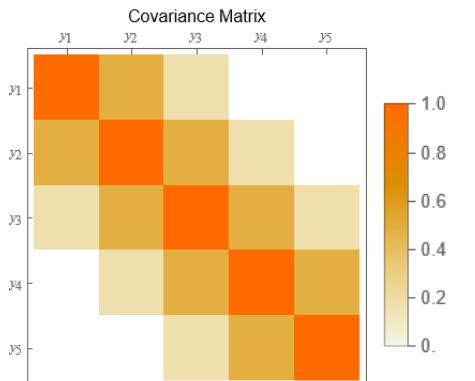
(* Generate 6 random variates from the multivariate normal distribution. Each
sample is a vector of 5 elements: *)
randomSamples=RandomVariate[multivariateDistribution,6];

(* Plot the generated random samples as discrete functions over integer points
{1,2,3,4,5}: *)
ListPlot[
Transpose[{Range[5],#}]&/@randomSamples,
Joined->True,
PlotMarkers->"OpenMarkers",
ImageSize->250,
PlotLabel->"Random Discrete Functions",
Frame->{True,True,False,False},
Axes->False,
PlotLegends->{"Sample 1","Sample 2","Sample 3","Sample 4","Sample 5","Sample
6"},

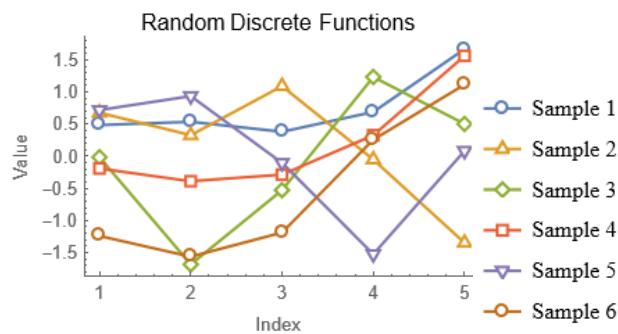
FrameLabel->{"Index","Value"}
]

```

Output



Output

**Mathematica Code 7.12**

Input

(* The enhanced Mathematica code with the Manipulate function is designed to provide an interactive educational tool that allows users to dynamically explore the effects of varying correlation coefficients within a covariance matrix on a multivariate normal distribution. By enabling real-time adjustments of the correlation values and the number of samples, the tool visualizes how these changes affect the structure of the covariance matrix and the characteristics of the generated random samples. This setup not only deepens understanding of complex statistical concepts like covariance and correlation but also serves as an invaluable resource for teaching and learning about the behavior and variability of multivariate distributions, making theoretical statistics concepts tangible and visually engaging: *)

```
Manipulate[
Module[
{zeroMeanVector, correlationMatrix, multivariateDistribution, randomSamples, matrixPlot, samplesPlot},
(* Define a zero mean vector for the multivariate distribution: *)
zeroMeanVector=ConstantArray[0,5];

(* Define a 5x5 covariance matrix based on adjustable correlation coefficients: *)
correlationMatrix=
{
{1,corr1,corr2,corr3,corr4},
{corr1,1,corr1,corr2,corr3},
{corr2,corr1,1,corr1,corr2},
{corr3,corr2,corr1,1,corr1},
{corr4,corr3,corr2,corr1,1}
};

(* Visualize the correlation matrix: *)
matrixPlot=MatrixPlot[
correlationMatrix,
ImageSize->250,
PlotLabel->"Covariance Matrix",
PlotLegends->Automatic
];

(* Define the multivariate normal distribution with the given mean and covariance
matrix: *)
multivariateDistribution=MultinormalDistribution[zeroMeanVector,correlationMatrix
];

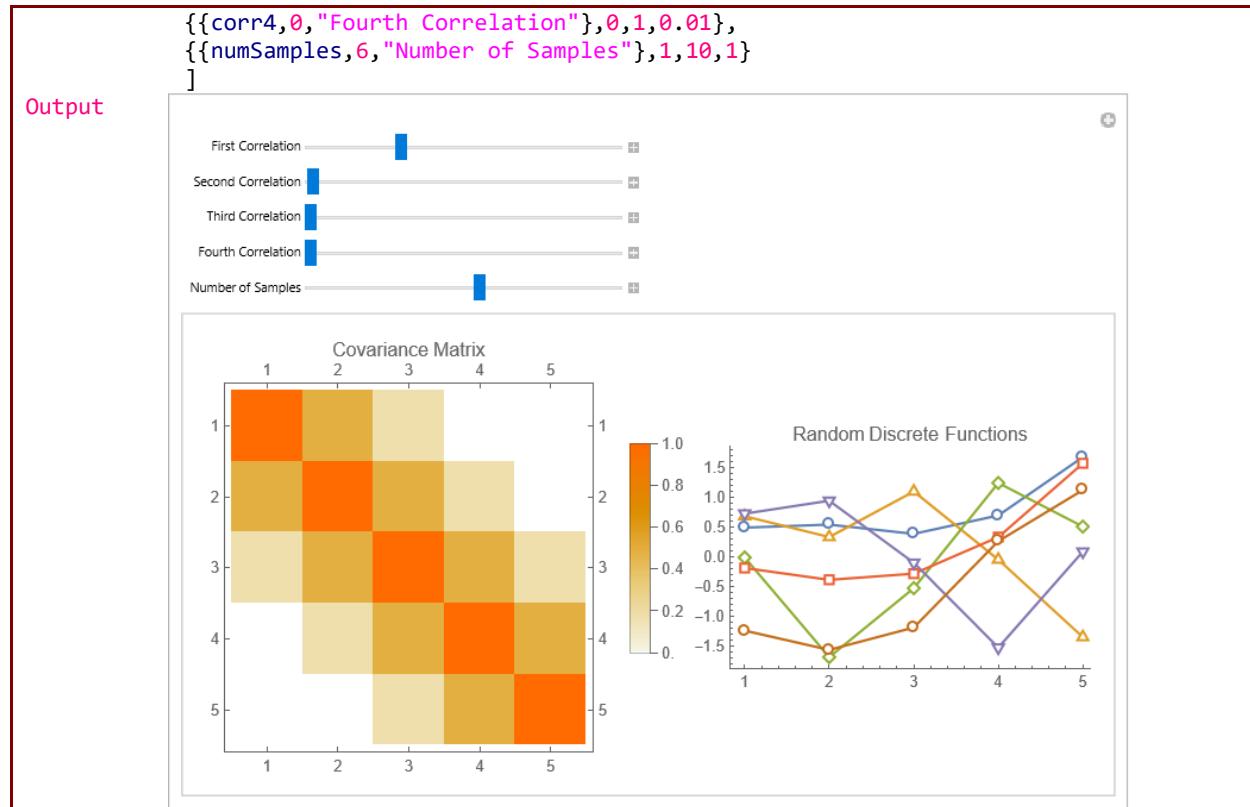
(* Set a random seed for reproducibility: *)
SeedRandom[1];

(* Generate random variates from the multivariate normal distribution: *)
randomSamples=RandomVariate[multivariateDistribution,numSamples];

(* Plot the generated random samples as discrete functions over integer points
{1,2,3,4,5}: *)
samplesPlot=ListPlot[
Transpose[{Range[5],#}]&@randomSamples,
Joined->True,
PlotMarkers->"OpenMarkers",
ImageSize->250,
PlotLabel->"Random Discrete Functions",
Frame->{True,True,False,False},
Axes->False
];

(* Display both plots side by side: *)
Row[{matrixPlot,samplesPlot}],

(* Controls for interactive manipulation: *)
{{corr1,0.3,"First Correlation"},0,1,0.05},
{{corr2,0.01,"Second Correlation"},0,1,0.01},
{{corr3,0,"Third Correlation"},0,1,0.01},
```

**Mathematica Code 7.13**

Input

```
(* The code demonstrates the setup and application of a Gaussian Process (GP) using
a squared exponential covariance function. It calculates and visualizes a
covariance matrix to illustrate the impact of point distances on their correlations,
essential for understanding GP behavior. The code then generates and visualizes
multiple samples from the GP, showcasing potential realizations over a defined
range, thus offering insights into the variability and smoothness of the process:
*)

(* Define a squared exponential covariance function to model the relationship
between two points based on their Euclidean distance: *)
SquaredExponentialCovariance[x1_,x2_]:=Exp[-(x1-x2)^2];

(* Generate an evenly spaced sequence of input points from 0 to 5. These points
are the domain over which the Gaussian Process is evaluated: *)
inputPoints=Range[0,5,.05];

(* Determine the total number of input points generated: *)
numberOfPoints=Length[inputPoints];

(* Compute the covariance matrix for the set of input points using the squared
exponential covariance function. This matrix quantifies the expected degree of
similarity between any two points based on their squared exponential relationship:
*)
covarianceMatrix=Outer[SquaredExponentialCovariance,inputPoints,inputPoints];

(* Add a small value to the diagonal elements (nugget effect) to ensure the matrix
is numerically stable and positive definite: *)
covarianceMatrix=covarianceMatrix+10^-6*IdentityMatrix[numberOfPoints];
```

```

(* Visualize the covariance matrix using MatrixPlot, providing insights into how
covariance changes with the distance between points: *)
MatrixPlot[
  covarianceMatrix,
  ImageSize->250,
  PlotLabel->"Covariance Matrix Visualization",
  PlotLegends->Automatic
]

(* Initialize the random seed: *)
SeedRandom[1];

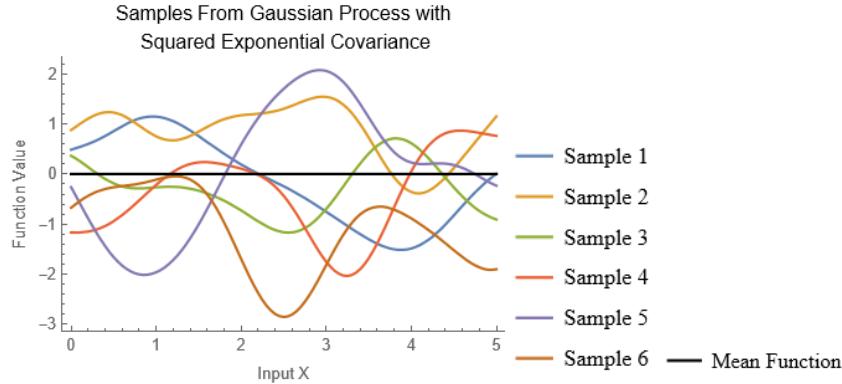
(* Define a mean vector for the multivariate normal distribution, initialized to
zero for all points: *)
meanVector=ConstantArray[0,numberofPoints];

(* Sample 6 random functions from the multivariate normal distribution defined by
the mean vector and the covariance matrix: *)
sampledFunctions=RandomVariate[
  MultinormalDistribution[meanVector,covarianceMatrix],
  6
];

(* Display the sampled functions from the Gaussian Process, illustrating the range
of variation among functions consistent with the specified covariance: *)
Show[
  ListLinePlot[
    Transpose[{inputPoints, #}]&/@sampledFunctions,
    Axes->None,
    Filling->None,
    Frame->{True,True,False,False},
    FrameLabel->{"Input X","Function Value"},
    PlotLabel->"Samples From Gaussian Process with\n Squared Exponential
Covariance",
    PlotLegends->Array["Sample "<>ToString[#]&,6],
    ImageSize->300
  ],
  ListLinePlot[
    Transpose[{inputPoints,meanVector}],
    PlotStyle->Black,
    PlotLegends->>{"Mean Function"}
  ]
]

```

Output

Output**Mathematica Code 7.14**

```

Input (* The code aims to provide an interactive and educational experience by allowing users to dynamically adjust the scale parameter and number of samples in Gaussian Process (GP). This interactivity helps illustrate how these parameters influence the covariance structure and the variability of function realizations within GP. The visualization of both covariance matrices and sampled functions facilitates a deeper understanding of key concepts like scale sensitivity and numerical stability: *)

(* Define a generic squared exponential covariance function. This function takes a scale parameter and the positions of two points ( $x_1, x_2$ ) and computes their covariance based on the squared exponential kernel, which models the expected similarity between points as a function of their distance and the specified scale: *)
SquaredExponentialCovariance[scale_, x1_, x2_] := Exp[-(x1 - x2)^2/scale];

(* Generate an evenly spaced sequence of input points from 0 to 5: *)
inputPoints = Range[0, 5, .05];

(* Determine the total number of input points generated: *)
numberOfPoints = Length[inputPoints];

(* Create a covariance matrix for a given scale parameter. This matrix is computed by applying the squared exponential covariance function to every pair of input points, which quantifies the expected degree of similarity based on their distance and the scale. A small constant (nugget effect) is added to the diagonal elements to ensure numerical stability by making the matrix positive definite: *)
CovarianceMatrix[scale_] := Outer[SquaredExponentialCovariance[scale, #1, #2] &, inputPoints, inputPoints] + 10^-6 IdentityMatrix[numberOfPoints];

(* Mean vector for the multivariate normal distribution: *)
meanVector = ConstantArray[0, numberOfPoints];

(* Function to visualize GP realizations along with covariance matrix visualization: *)
VisualizeGP[scale_, samples_] := Module[
  {covarianceMatrix, sampledFunctions, matrixPlot, functionPlot},
  (* Compute the covariance matrix for the given scale: *)
  covarianceMatrix = CovarianceMatrix[scale];
  (* Sample functions from the GP: *)
  sampledFunctions = RandomVariate[MultinormalDistribution[meanVector, covarianceMatrix], samples];
  matrixPlot = MatrixPlot[covarianceMatrix];
  functionPlot = Plot[#, {x, 0, 5}] & /@ sampledFunctions;
  Grid[{matrixPlot, functionPlot}, Alignment -> Center]
]


```

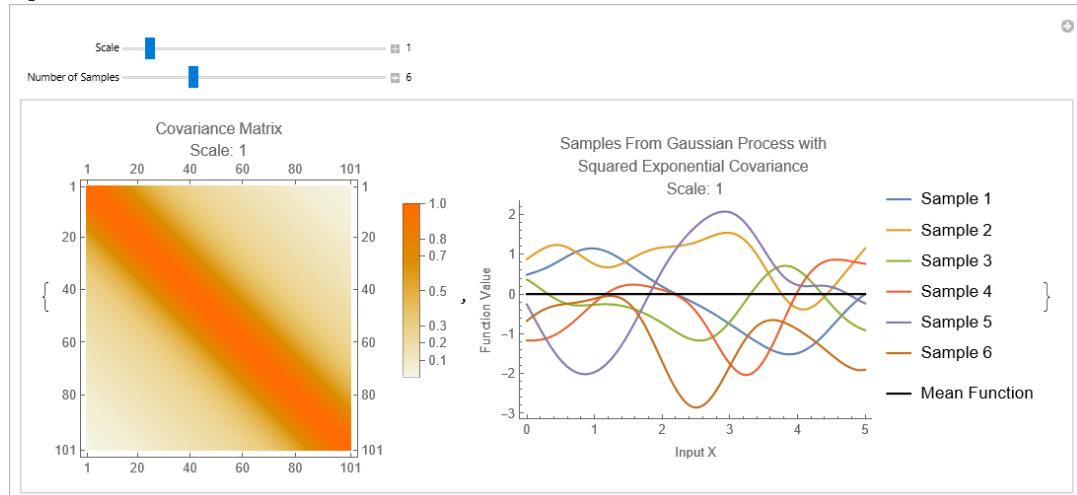
```

(* Plot the covariance matrix: *)
matrixPlot=MatrixPlot[
  covarianceMatrix,
  ImageSize->250,
  PlotLabel->"Covariance Matrix\nScale: "<>ToString[scale],
  PlotLegends->Automatic
];

(* Plot the sampled functions: *)
functionPlot=Show[
  ListLinePlot[
    Transpose[{inputPoints, #}]&/@sampledFunctions,
    Axes->None,
    Filling->None,
    Frame->{True,True,False,False},
    FrameLabel->{"Input X","Function Value"},
    PlotLabel->"Samples From Gaussian Process with \nSquared Exponential
Covariance \nScale: "<>ToString[scale],
    PlotLegends->Array["Sample "<>ToString[#]&,samples],
    ImageSize->300
  ],
  ListLinePlot[
    Transpose[{inputPoints,meanVector}],
    PlotStyle->Black,
    PlotLegends->{"Mean Function"}
  ]
];
(* Return both plots as a pair: *)
{matrixPlot,functionPlot}
];
(* Manipulate wrapper to adjust scale parameter and number of samples dynamically:
*)
Manipulate[
  (* Initialize the random seed: *)
  SeedRandom[1];
  VisualizeGP[scale,numSamples],
  {{scale,1,"Scale"},0.1,10,0.05,Appearance->"Labeled"},
  {{numSamples,6,"Number of Samples"},1,20,1,Appearance->"Labeled"},
  Initialization->{scale=1,numSamples=6}
]

```

Output



Mathematica Code 7.15

```

Input      (* Define the Matérn covariance function with nu=3/2: *)
MaternCovariance32[scale_,x1_,x2_]:=Module[
  {r=Abs[x1-x2]},
  (1+Sqrt[3]*r/scale)*Exp[-Sqrt[3]*r/scale]
];

(* Generate an evenly spaced sequence of input points from 0 to 5: *)
inputPoints=Range[0,5,.05];

(* Calculate the total number of input points: *)
numberOfPoints=Length[inputPoints];

(* Function to compute a covariance matrix with a nugget effect for numerical
stability: *)
CovarianceMatrix[scale_]:=Outer[
  MaternCovariance32[scale,#1,#2]&,amp;
  inputPoints,
  inputPoints
]+10^-6*IdentityMatrix[numberOfPoints];

(* Initialize the random seed for reproducibility: *)
SeedRandom[1];

(* Mean vector for the multivariate normal distribution, assumed zero for
simplicity: *)
meanVector=ConstantArray[0,numberOfPoints];

(* Function to visualize Gaussian Process realizations along with covariance matrix
visualization: *)
VisualizeGP[scale_,samples_]:=Module[
  {covarianceMatrix,sampledFunctions,matrixPlot,functionPlot},

  (* Compute the covariance matrix for the given scale: *)
  covarianceMatrix=CovarianceMatrix[scale];

  (* Sample functions from the GP: *)
  sampledFunctions=RandomVariate[
    MultinormalDistribution[meanVector,covarianceMatrix],
    samples
  ];

  (* Visualize the covariance matrix: *)
  matrixPlot=MatrixPlot[
    covarianceMatrix,
    ImageSize->250,
    PlotLabel->"Covariance Matrix\nScale: "<>ToString[scale],
    PlotLegends->Automatic
  ];

  (* Display sampled functions: *)
  functionPlot=Show[
    ListLinePlot[
      Transpose[{inputPoints, #}]&/@sampledFunctions,
      Axes->None,
      Filling->None,
      Frame->{True,True,False,False},
      FrameLabel->{"Input X","Function Value"},
      PlotLabel->"Samples From Gaussian Process with \nMatérn Covariance (v=3/2)
      \nScale: "<>ToString[scale],
      PlotStyle->{Black,Thickness[2]}
    ]
  ];
];

```

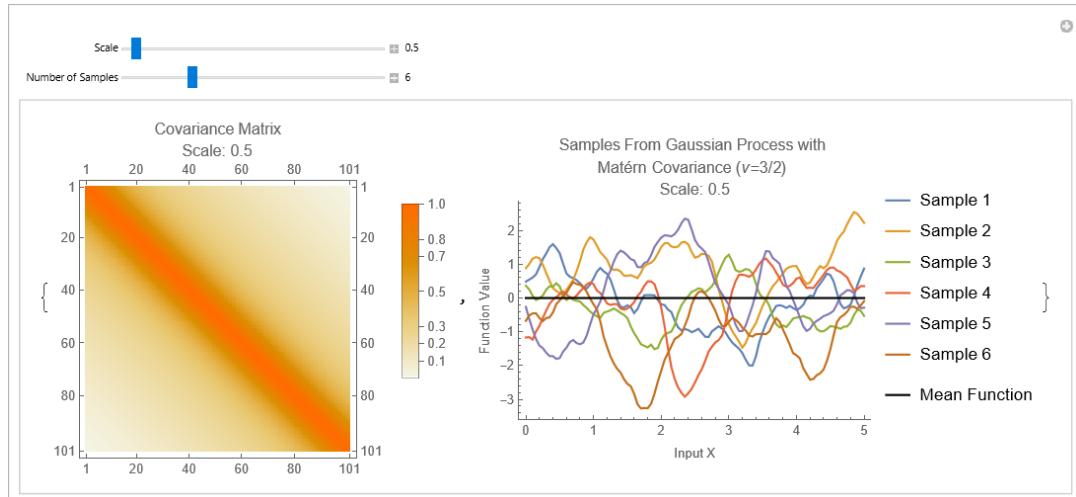
```

PlotLegends->Array["Sample " <> ToString[#[#]&,samples],
ImageSize->300
],
(* Overlay the mean function plot: *)
ListLinePlot[
Transpose[{inputPoints,meanVector}],
PlotStyle->Black,
PlotLegends->{"Mean Function"}
]
];
{matrixPlot,functionPlot}];

(* Manipulate wrapper to dynamically adjust the scale parameter and number of
samples: *)
Manipulate[
(* Initialize the random seed: *)
SeedRandom[1];
VisualizeGP[scale,numSamples],
(* Slider for scale adjustment: *)
{{scale,0.5,"Scale"},0.1,10,0.05,Appearance->"Labeled"},
(* Slider for adjusting the number of samples: *)
{{numSamples,6,"Number of Samples"},1,20,1,Appearance->"Labeled"},
Initialization->{scale=1,numSamples=6}
]
]

```

Output



Mathematica Code 7.16

Input

```

(* Posterior Distribution, Method 1: *)
(* The code executes Gaussian Process (GP) regression to model a sinusoidal
function, generating predictions at both training and extended test input
locations. It constructs covariance matrices using a squared exponential kernel,
computes predictive means and covariances, and evaluates uncertainty through 95%
confidence intervals. The code also simulates possible outcomes by sampling from
the predictive distribution. For visualization, it plots the predictive mean,
confidence intervals, actual function values, sampled paths, and training data.
This visual representation helps in assessing the accuracy, reliability, and
variability of the model's predictions, demonstrating the practical application
and interpretative power of GP regression in a statistical learning context: *)

(* Number of training observations: *)
numTraining=4;

```

```
(* Training input locations, uniformly spaced within one period of a sine wave: *)
trainingInputs=Transpose[{Subdivide[0,2 Pi,numTraining-1]}];

(* Training outputs based on the sine function: *)
trainingOutputs=Sin[trainingInputs];

(* Squared Euclidean distance matrix for training inputs: *)
trainingDistMatrix=Outer[EuclideanDistance,trainingInputs,trainingInputs,1]^2;

(* Small epsilon value for numerical stability in the covariance matrix: *)
stabilityEpsilon=10.^-6;

(* Covariance matrix for the training data using a squared exponential kernel: *)
covarianceMatrixTraining=Exp[-
  trainingDistMatrix]+DiagonalMatrix[ConstantArray[stabilityEpsilon,numTraining]];

(* Visualization of covariance matrix for the training data: *)
MatrixPlot[
  covarianceMatrixTraining,
  ImageSize->Small
]

(* Test input locations, extending slightly beyond the training input range: *)
testInputs=Transpose[{Subdivide[-0.5,2 Pi+0.5,70]}];

(* Squared Euclidean distance matrix for test inputs: *)
testDistMatrix=Outer[EuclideanDistance,testInputs,testInputs,1]^2;

(* Covariance matrix for test inputs: *)
covarianceMatrixTest=Exp[-
  testDistMatrix]+DiagonalMatrix[ConstantArray[stabilityEpsilon,Length[testInputs]]];

(* Visualization of covariance matrix for test inputs: *)
MatrixPlot[
  covarianceMatrixTest,
  ImageSize->Small
]

(* Squared distances between test inputs and training inputs: *)
crossDistMatrix=Outer[EuclideanDistance,testInputs,trainingInputs,1]^2;

(* Cross-covariance matrix between test inputs and training inputs: *)
crossCovarianceMatrix=Exp[-crossDistMatrix];

(* Visualization of cross-covariance matrix between test inputs and training
inputs: *)
MatrixPlot[
  crossCovarianceMatrix,
  ImageSize->Small,
  AspectRatio->1/2
]

MatrixPlot[
  Transpose[crossCovarianceMatrix],
  ImageSize->Small,
  AspectRatio->1/2
]

(* Inverse of the training data covariance matrix: *)
```

```

inverseCovarianceMatrix=Inverse[covarianceMatrixTraining];

(* Predictive mean vector for test inputs: *)
predictiveMean=crossCovarianceMatrix.inverseCovarianceMatrix.trainingOutputs;

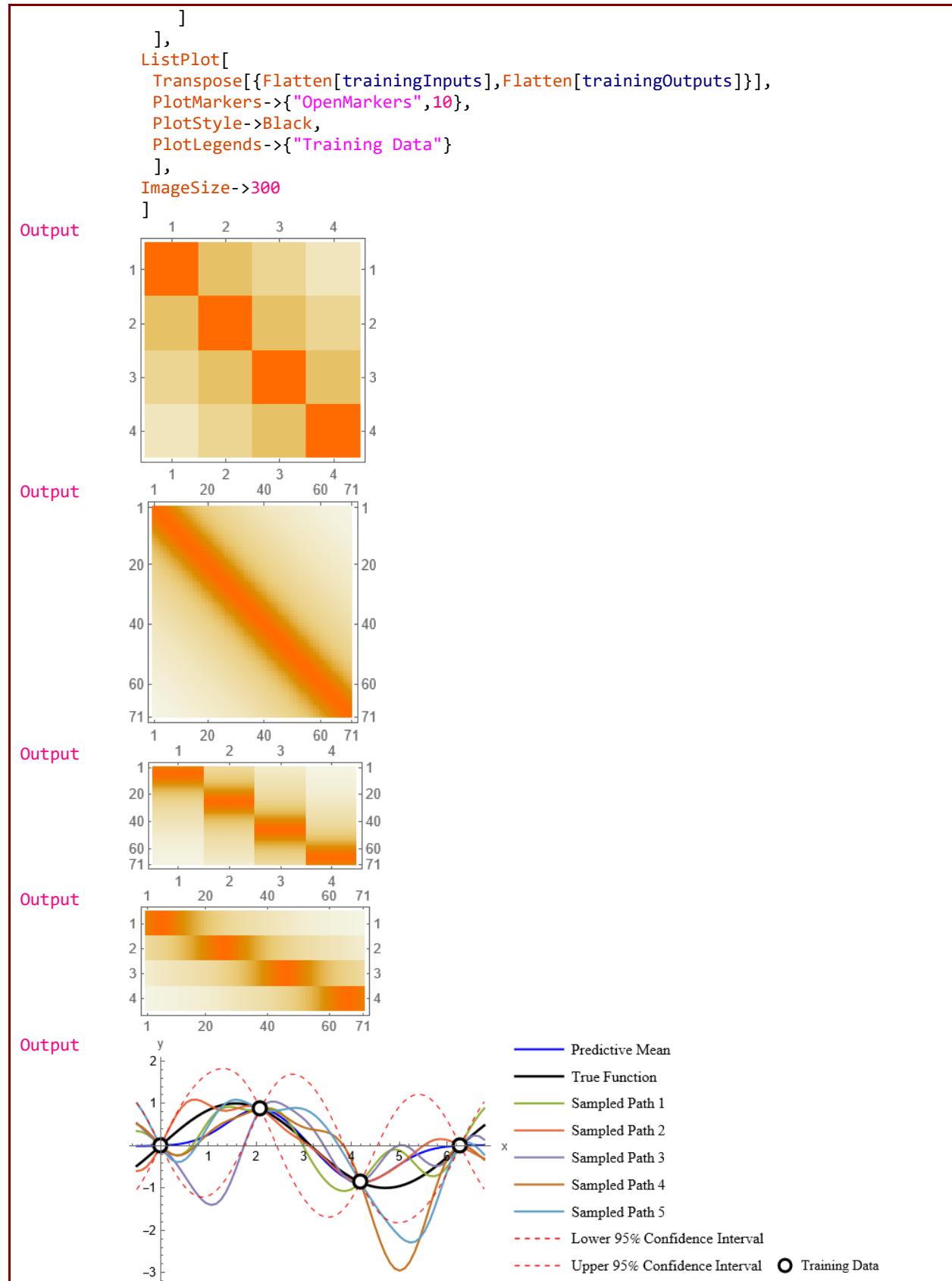
(* Predictive covariance matrix for test inputs: *)
predictiveCovariance=covarianceMatrixTest-
crossCovarianceMatrix.inverseCovarianceMatrix.Transpose[crossCovarianceMatrix];

(* Quantiles for constructing confidence intervals: *)
confidenceLower=predictiveMean+Quantile[NormalDistribution[0,1],0.05]
Sqrt[Diagonal[predictiveCovariance]];
confidenceUpper=predictiveMean+Quantile[NormalDistribution[0,1],0.95]
Sqrt[Diagonal[predictiveCovariance]];

(* Sampling from the predictive distribution: *)
sampledPredictions=RandomVariate[
  MultinormalDistribution[
    Flatten[predictiveMean],
    predictiveCovariance
  ],
  6
];

(* Visualization of the results: *)
Show[
  ListLinePlot[
    Join[
      {
        (* Predictive mean as a blue line: *)
        Transpose[{Flatten[testInputs],Flatten[predictiveMean]}],
        (* True function values as a thick black line: *)
        Transpose[{Flatten[testInputs],Flatten[Sin[testInputs]]}]
      },
      (* Sampled predictive paths: *)
      Table[
        Transpose[{Flatten[testInputs],sampledPredictions[[i]]}],
        {i,1,5}
      ],
      {
        (* Lower confidence interval as a dashed red line: *)
        Transpose[{Flatten[testInputs],Flatten[confidenceLower]}],
        (* Upper confidence interval as a dashed red line: *)
        Transpose[{Flatten[testInputs],Flatten[confidenceUpper]}]
      }
    ],
    PlotStyle->Join[
      {Blue,{Thick,Black}},
      (* Using Automatic for sampled paths: *)
      Table[Automatic,{5}],
      {
        {Thickness[0.003],Red,Dashed},
        {Thickness[0.003],Red,Dashed}
      }
    ],
    AxesLabel->{"x","y"},
    PlotRange->Full,
    PlotLegends->Join[
      {"Predictive Mean","True Function"},
      Table["Sampled Path "<>ToString[i],{i,1,5}],
      {"Lower 95% Confidence Interval","Upper 95% Confidence Interval"}
    ]
  ]
];

```



Mathematica Code 7.17

```

Input (* Posterior Distribution, Method 2: *)

(* The code serves as a comprehensive demonstration of a Gaussian Process (GP)
modeled with a squared exponential kernel, specifically focusing on the process's
behavior when conditioned on a single training data point. It starts by defining
the covariance function and generating a grid of evaluation points, followed by
the computation of the initial covariance matrix which is then adjusted based on
the known training data to reflect updated beliefs about correlations within the
process. This adjustment, illustrated through MatrixPlot, highlights the impact of
the training data on the GP's covariance structure. The code further computes the
predictive mean based on the conditioned covariance and samples multiple function
realizations from this distribution, showcasing the variability and potential
shapes of the GP consistent with the observed data. These steps are visualized in
a comprehensive plot that combines the sampled functions, predictive mean, and
training data, providing a clear visual representation of the Gaussian Process
methodology and its application in probabilistic modeling and Bayesian inference:
*)

(* Define the squared exponential kernel: *)
SquaredExponentialKernel[x1_,x2_]:=Exp[-(x1-x2)^2];

(* Create a sequence of evenly distributed evaluation points for the Gaussian
Process: *)
evaluationPoints=Range[0,5,.05];

(* Total number of evaluation points: *)
totalPoints=Length[evaluationPoints];

(* Compute the initial covariance matrix using the squared exponential kernel for
all pairs of evaluation points: *)
initialCovariance=Outer[SquaredExponentialKernel,evaluationPoints,evaluationPoint
s];

(* Apply a nugget effect to ensure the covariance matrix is numerically stable and
positive definite: *)
stabilizedCovariance=initialCovariance+10^-6*IdentityMatrix[totalPoints];

(* Define single training data point for GP conditioning: *)

(* Location of the training input: *)
trainingInput={3.};

(* Corresponding output value at the training input: *)
trainingOutput={1.};

(* Convert training input to index in evaluation points for accurate
conditioning: *)
trainingIndex=Nearest[evaluationPoints->"Index",trainingInput][[All,1]];

(* Adjust the covariance matrix to reflect conditioning on the training data. This
matrix represents the updated belief about point correlations after observing the
training data: *)
conditionedCovariance=stabilizedCovariance-
stabilizedCovariance[[All,trainingIndex]].Inverse[stabilizedCovariance[[trainin
gIndex,trainingIndex]]].stabilizedCovariance[[trainingIndex,All]];

conditionedCovariance=conditionedCovariance+10^-6*IdentityMatrix[totalPoints];

```

```
(* Visualize the conditioned covariance matrix. This plot provides insights into
how the knowledge of the training data influences the correlations across the
process: *)
MatrixPlot[
  conditionedCovariance,
  ImageSize->250,
  PlotLabel->"Conditioned Covariance Matrix",
  PlotLegends->Automatic
]

(* Compute the predictive mean of the Gaussian Process at each point, given the
training data: *)

predictiveMean=initialCovariance[[All,trainingIndex]].Inverse[initialCovariance[[
  trainingIndex,trainingIndex]]].trainingOutput;

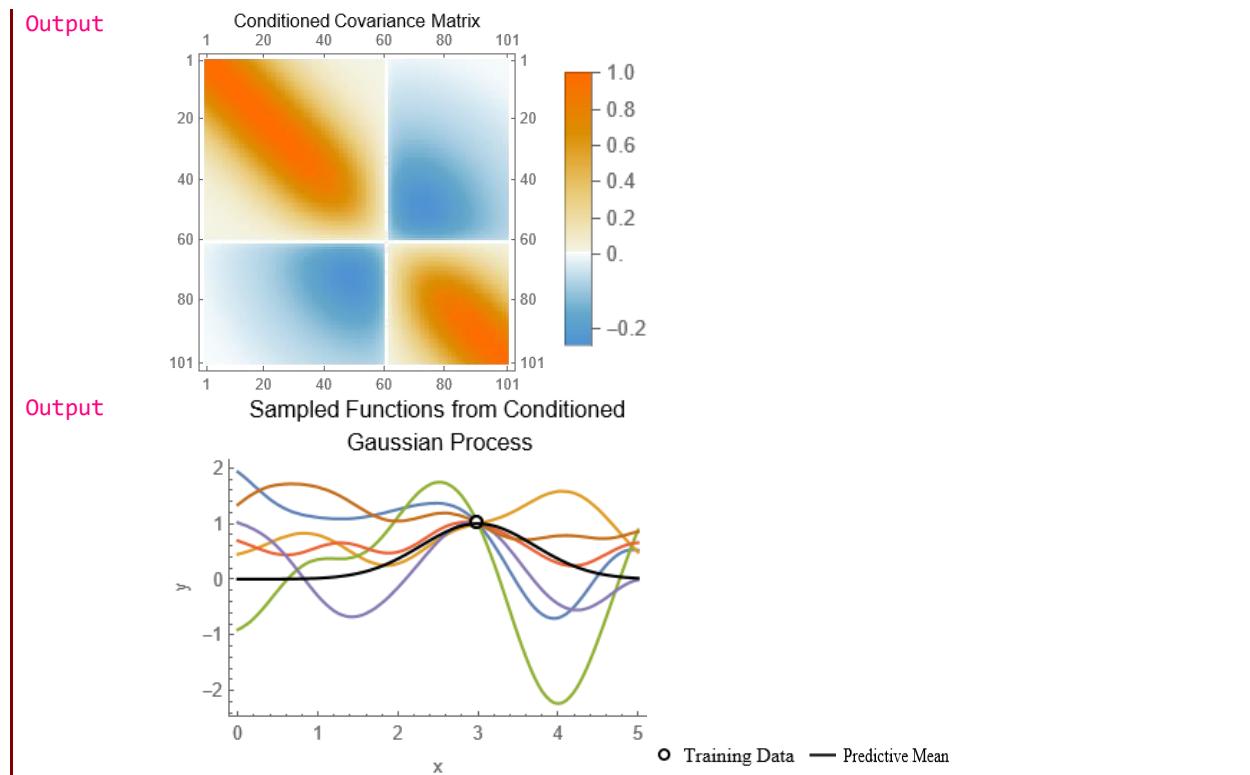
(* Initialize random seed for reproducibility in generating function samples: *)
SeedRandom[12];

(* Generate several function realizations from the conditioned Gaussian Process,
illustrating potential variations that align with both the prior assumptions and
observed data: *)

sampledFunctions=RandomVariate[
  MultinormalDistribution[predictiveMean,conditionedCovariance],
  6
];

(* Visualize the sampled functions, the mean prediction, and the training data in
a composite plot. This illustrates the variability of the Gaussian Process and its
adherence to the known data: *)

Show[
  ListPlot[
    Map[Transpose[{evaluationPoints, #}]&,sampledFunctions],
    Joined->True,
    Axes->None,
    Frame->{True,True,False,False},
    FrameLabel->{"x","y"},
    PlotLabel->"Sampled Functions from Conditioned\n Gaussian Process",
    ImageSize->250
  ],
  ListPlot[
    Transpose[{Part[evaluationPoints,trainingIndex],trainingOutput}],
    PlotMarkers->"OpenMarkers",
    PlotStyle->Black,
    PlotLegends->{"Training Data"},
    ImageSize->250
  ],
  ListLinePlot[
    Transpose[{evaluationPoints,predictiveMean}],
    PlotStyle->Directive[GrayLevel[0],Opacity[1]],
    PlotLegends->{"Predictive Mean"},
    ImageSize->250
  ]
]
```

**Mathematica Code 7.18**

Input

```
(* The code utilizes the Manipulate function to create an interactive visualization for exploring Gaussian Processes (GPs) conditioned on training data. It starts by defining a squared exponential kernel for covariance, generates a set of evaluation points, and computes an initial covariance matrix. This matrix is stabilized and then adjusted based on user-provided training data. The code visualizes the resulting conditioned covariance matrix, computes the predictive mean, and generates multiple potential function realizations. These components are visualized together, demonstrating the influence of training data on GP behavior and prediction variability. The interactive controls allow users to manipulate training data and the number of samples to see their effect on the Gaussian Process outcomes: *)
```

```
Manipulate[
(* Define the squared exponential kernel: *)
SquaredExponentialKernel[x1_,x2_]:=Exp[-(x1-x2)^2];

(* Create a sequence of evenly distributed evaluation points for the Gaussian Process: *)
evaluationPoints=Range[0,5,.05];

(* Total number of evaluation points: *)
totalPoints=Length[evaluationPoints];

(* Compute the initial covariance matrix using the squared exponential kernel for all pairs of evaluation points: *)

initialCovariance=Outer[SquaredExponentialKernel,evaluationPoints,evaluationPoint
s];

(* Apply a nugget effect to ensure the covariance matrix is numerically stable and positive definite: *)
stabilizedCovariance=initialCovariance+10^-6*IdentityMatrix[totalPoints];
```

```
(* Convert training input to index in evaluation points for accurate conditioning:<*)
trainingIndex=Nearest[evaluationPoints->"Index",{trainingInput}][[All,1]];

(* Adjust the covariance matrix to reflect conditioning on the training data. This
matrix represents the updated belief about point correlations after observing the
training data: *)
conditionedCovariance=stabilizedCovariance-
stabilizedCovariance[[All,trainingIndex]].Inverse[stabilizedCovariance[[trainingI
ndex,trainingIndex]]].stabilizedCovariance[[trainingIndex,All]];

conditionedCovariance=conditionedCovariance+10^-6*IdentityMatrix[totalPoints];

(* Visualize the conditioned covariance matrix. This plot provides insights into
how the knowledge of the training data influences the correlations across the
process: *)
matrixPlot=MatrixPlot[
  conditionedCovariance,
  ImageSize->250,
  PlotLabel->"Conditioned Covariance Matrix",
  PlotLegends->Automatic
];

(* Compute the predictive mean of the Gaussian Process at each point, given the
training data: *)

predictiveMean=initialCovariance[[All,trainingIndex]].Inverse[initialCovariance[[
  trainingIndex,trainingIndex]]].{trainingOutput};

(* Initialize random seed for reproducibility in generating function samples: *)
SeedRandom[12];

(* Generate several function realizations from the conditioned Gaussian Process,
illustrating potential variations that align with both the prior assumptions and
observed data: *)

sampledFunctions=RandomVariate[MultinormalDistribution[Flatten[predictiveMean],co
nditionedCovariance],numSamples];

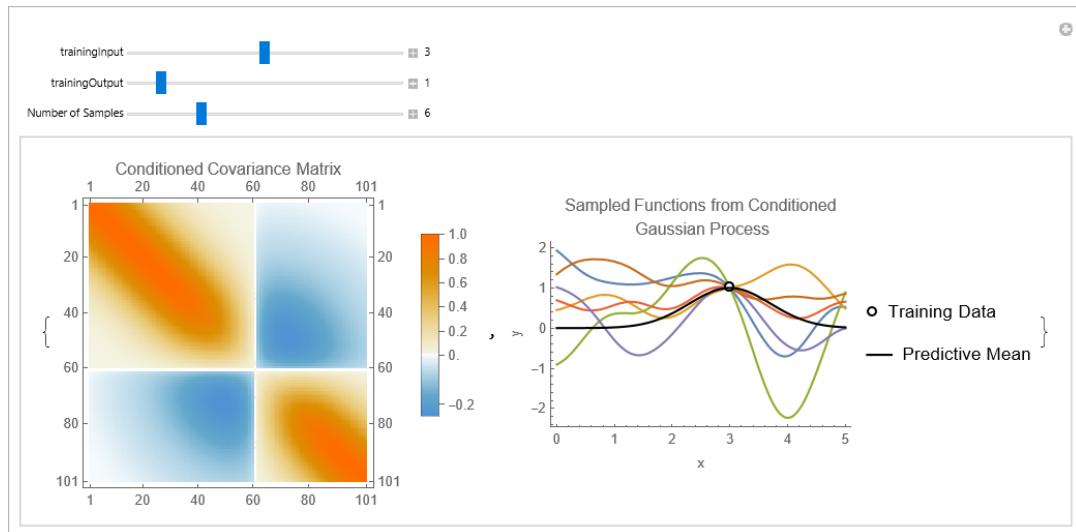
(* Visualize the sampled functions, the mean prediction, and the training data in
a composite plot. This illustrates the variability of the Gaussian Process and its
adherence to the known data: *)
functionsPlot=Show[
  ListPlot[
    Map[Transpose[{evaluationPoints,#}]&,sampledFunctions],
    Joined->True,
    Axes->None,
    Frame->\{True,True,False,False\},
    FrameLabel->\{"x","y"\},
    PlotLabel->"Sampled Functions from Conditioned\n Gaussian Process",
    ImageSize->250
  ],
  ListPlot[
    Transpose[\{Part[evaluationPoints,trainingIndex],{trainingOutput}\}],
    PlotMarkers->"OpenMarkers",
    PlotStyle->Black,
    PlotLegends->\{"Training Data"\},
    ImageSize->250
  ],
];
```

```

ListLinePlot[
Transpose[{evaluationPoints,predictiveMean}],
PlotStyle->Directive[GrayLevel[0],Opacity[1]],
PlotLegends->{"Predictive Mean"},
ImageSize->250
]
];
{matrixPlot,functionsPlot},

(* Controls for interactive manipulation: *)
{{trainingInput,3,"trainingInput"},1,5,0.1,Appearance->"Labeled"}, 
{{trainingOutput,1,"trainingOutput"},0.5,5,0.1,Appearance->"Labeled"}, 
{{numSamples,6,"Number of Samples"},1,20,1,Appearance->"Labeled"}, 
Initialization->{trainingInput=3,trainingOutput=1}
]

```

Output**Mathematica Code 7.19**

Input

```

(* Posterior Distribution: *)
(* The code is designed to demonstrate the principles of Gaussian Processes (GPs)
conditioned on training data, illustrating how known observations influence the
mean and covariance structure of the GP. The main goals are to visually represent
the effect of conditioning on the covariance matrix, generate and display multiple
function realizations from the conditioned GP, and serve as an educational tool
for understanding the dynamics of GPs: *)

(* Define the squared exponential covariance function between any two points x1
and x2: *)
SquaredExponential[x1_,x2_]:=Exp[-(x1-x2)^2];

(* Create an array of evenly spaced points to evaluate the Gaussian Process: *)
evaluationPoints=Range[0,5,.05];

(* Total number of evaluation points: *)
numPoints=Length[evaluationPoints];

(* Compute the covariance matrix for all pairs of evaluation points: *)
initialCovariance=Outer[SquaredExponential,evaluationPoints,evaluationPoints];

(* Stabilize the covariance matrix by adding a small constant to its diagonal
(nugget effect): *)

```

```
stabilizedCovariance=initialCovariance+10^-6*IdentityMatrix[numPoints];

(* Training data points and their observed outputs: *)
trainingInputs={2.,4.,1.,3.};
trainingOutputs={0.5,-1.,-1.4,1};

(* Convert training input locations to indices corresponding to the nearest points
in evaluationPoints: *)
trainingIndices=Flatten[Nearest[evaluationPoints->"Index",trainingInputs]];

(* Update the covariance matrix to reflect the presence of training data: *)
conditionedCovariance=stabilizedCovariance-Dot[
  Part[stabilizedCovariance,All,trainingIndices],
  Inverse[Part[stabilizedCovariance,trainingIndices,trainingIndices]],
  Part[stabilizedCovariance,trainingIndices,All]
];

conditionedCovariance=conditionedCovariance+IdentityMatrix[numPoints]/10^6;

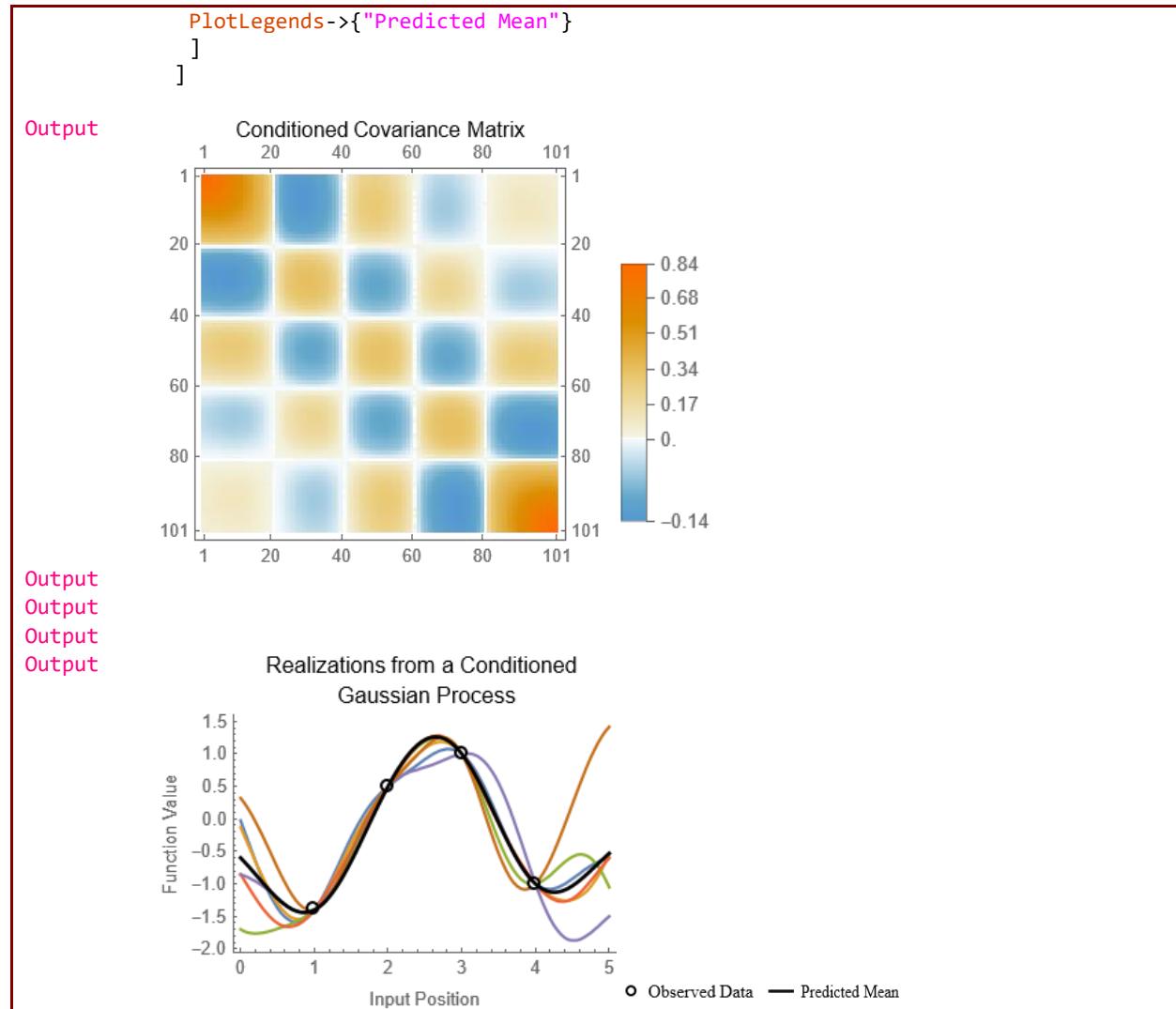
(* Plot the adjusted covariance matrix after conditioning: *)
MatrixPlot[
  conditionedCovariance,
  ImageSize->250,
  PlotLabel->"Conditioned Covariance Matrix",
  PlotLegends->Automatic
]

(* Compute the conditional mean vector based on the training data: *)
conditionalMean=stabilizedCovariance[[All,trainingIndices]].Inverse[stabilizedCovariance[[trainingIndices,trainingIndices]]].trainingOutputs;

(* Seed the random number generator for reproducibility: *)
SeedRandom[2];

(* Sample multiple realizations from the conditioned Gaussian Process: *)
sampledPaths=RandomVariate[MultinormalDistribution[conditionalMean,conditionedCovariance],6];

(* Display the sampled functions, with training data points and the predicted mean
function: *)
Show[
  ListPlot[
    Map[Transpose[{evaluationPoints,#}]&,sampledPaths],
    Axes->None,
    Filling->None,
    Frame->{True,True,False,False},
    FrameLabel->{"Input Position","Function Value"},
    PlotLabel->"Realizations from a Conditioned \n Gaussian Process",
    Joined->True,
    ImageSize->250
  ],
  ListPlot[
    Transpose[{evaluationPoints[[trainingIndices]],trainingOutputs}],
    PlotMarkers->"OpenMarkers",
    PlotStyle->Black,
    PlotLegends->>{"Observed Data"}
  ],
  ListLinePlot[
    Transpose[{evaluationPoints,conditionalMean}],
    PlotStyle->Directive[Black,Thick],
    PlotLegends->{"Predicted Mean Function"}]
]
```



Unit 7.4

Setting Up Bayesian Optimization in Mathematica

Bayesian optimization is a powerful strategy for the optimization of objective functions that are expensive to evaluate. It is particularly useful in scenarios where acquiring new data points is costly or experiments take a long time. This unit provides a practical guide to implementing Bayesian optimization in Mathematica using several of its built-in functions: `Predict`, `GaussianProcess`, `BayesianMinimization`, and `BayesianMaximization`.

- The `Predict` function in Mathematica is a high-level machine learning function used to create predictive models from data. It automatically selects the appropriate machine learning algorithm based on the nature of the data provided. This function can handle various types of data, including numeric, categorical, and textual data, making it versatile for a range of predictive tasks. Once trained, the model can be used to make predictions on new data. It supports different prediction methods like linear regression, decision trees, neural networks, and more, depending on the structure and complexity of the data.
- In Mathematica, `GaussianProcess` can indeed be used as a method within the `Predict` function. This approach allows you to utilize Gaussian process models for predictive tasks.
- `BayesianMinimization` is a function in Mathematica that applies Bayesian optimization techniques to find the minimum of a function. This method is useful for optimization problems where evaluations of the function are costly or time-consuming. Bayesian optimization uses a probabilistic model to predict the function's behavior and makes decisions based on these predictions to sample the most promising areas. It is particularly effective for optimizing hyperparameters in machine learning models or any scenario where the objective function does not have a simple analytical form and requires expensive evaluations.
- `BayesianMaximization` is similar to `BayesianMinimization` but focuses on finding the maximum value of a function. Like its minimization counterpart, it uses Bayesian optimization strategies to intelligently explore the function space, balancing the exploration of new areas with the exploitation of known good areas. This function is particularly useful when dealing with complex optimization landscapes that are difficult to navigate with traditional methods due to multiple local maxima or the expensive nature of function evaluations.

These functions represent a powerful suite of tools in Mathematica for handling various types of data analysis, predictive modeling, and optimization tasks, each equipped with sophisticated algorithms and easy-to-use interfaces to cater to both novice users and expert analysts.

1. Predict

`Predict[{in1->out1,in2->out2,...}]`

generates a `PredictorFunction[...]` based on the example input-output pairs given.

`Predict[{in1,in2,...}->{out1,out2,...}]`

generates the same result.

`Predict[training,input]`

attempts to predict the output associated with input from the training examples given.

`Predict["name",input]`

uses the built-in predictor function represented by "name".

`Predict[predictor,opts]`

takes an existing predictor function and modifies it with the new options given.

2. GaussianProcess

GaussianProcess

Method for `Predict`.

`Predict[data, Method -> "GaussianProcess"]`

Infers values by conditioning a Gaussian process on the training data.

The following options can be given:

<code>AssumeDeterministic</code>	<code>False</code>	whether or not the function should be assumed to be deterministic
<code>"CovarianceType"</code>	<code>Automatic</code>	the covariance type to use
<code>"EstimationMethod"</code>	<code>"MaximumPosterior"</code>	the method to infer the values
<code>"OptimizationMethod"</code>	<code>Automatic</code>	the optimization method to estimate parameters

Possible settings for `"CovarianceType"` include:

<code>"SquaredExponential"</code>	exponential kernel
<code>"HammingDistance"</code>	exponential kernel for nominal variables
<code>"Periodic"</code>	periodic kernel
<code>"RationalQuadratic"</code>	rational quadratic kernel
<code>"Linear"</code>	linear kernel
<code>"Matern5/2"</code>	Matérn kernel with exponent 5/2
<code>"Matern3/2"</code>	Matérn kernel with exponent 3/2
<code>"Composite"</code>	a composition of the previous kernels
<code>assoc</code>	specify a different kernel for each feature type

Possible settings for `"EstimationMethod"` include:

<code>"MaximumPosterior"</code>	maximize the posterior distribution
<code>"MaximumLikelihood"</code>	maximize the likelihood
<code>"MeanPosterior"</code>	mean of the posterior distribution

Possible settings for `"OptimizationMethod"` include:

<code>"SimulatedAnnealing"</code>	uses simulated annealing to find the minimum
<code>"FindMinimum"</code>	uses <code>FindMinimum</code> to find the minimum

3. BayesianMinimization/BayesianMaximization

BayesianMinimization [f, {conf1, conf2, ...}]

gives an object representing the result of Bayesian minimization of the function f over the configurations confi.

BayesianMinimization[f, reg]

minimizes over the region represented by the region specification reg.

BayesianMinimization[f, sampler]

minimizes over configurations obtained by applying the function sampler.

BayesianMinimization [f, {conf1, conf2, ...} -> nsampler]

applies the function nsampler to successively generate configurations starting from the confi.

BayesianMaximization[f, {conf1, conf2, ...}]

gives an object representing the result of Bayesian maximization over the function f over the configurations confi.

BayesianMaximization[f, reg]

maximizes over the region represented by the region specification reg.

BayesianMaximization[f, sampler]

maximizes over configurations obtained by applying the function sampler.

BayesianMaximization[f, {conf1, conf2, ...} -> nsampler]

applies the function nsampler to successively generate configurations starting from the confi.

Remarks:

- `BayesianMinimization[...]` returns a `BayesianMinimizationObject[...]` whose properties can be obtained using `BayesianMinimizationObject[...]["prop"]`.
- `BayesianMinimizationObject[...][{prop1, prop2, ...}]` gives the list of properties `{prop1, prop2, ...}`.
- `BayesianMaximization[...]` returns a `BayesianMaximizationObject[...]` whose properties can be obtained using `BayesianMaximizationObject[...]["prop"]`.
- `BayesianMaximizationObject[...][{prop1, prop2, ...}]` gives the list of properties `{prop1, prop2, ...}`.
- Configurations can be of any form accepted by `Predict` (single data element, list of data elements, association of data elements, etc.) and of any type accepted by `Predict` (numerical, textual, sounds, images, etc.).

Possible properties include:

<code>"EvaluationHistory"</code>	configurations and values explored during minimization/maximization
<code>"MinimumConfiguration"</code>	configuration found that minimizes/maximizes the result from f
<code>"MaximumConfiguration"</code>	
<code>"MinimumValue"</code>	estimated minimum/maximum value obtained from f
<code>"MaximumValue"</code>	
<code>"Method"</code>	method used for Bayesian maximization
<code>"NextConfiguration"</code>	configuration to sample next if maximization were continued
<code>"PredictorFunction"</code>	best prediction model found for the function f
<code>"Properties"</code>	list of all available properties

`BayesianMinimization`/`BayesianMaximization` takes the following options:

<code>AssumeDeterministic</code>	<code>False</code>	whether to assume that f is deterministic
<code>InitialEvaluationHistory</code>	<code>None</code>	initial set of configurations and values
<code>MaxIterations</code>	<code>100</code>	maximum number of iterations
<code>Method</code>	<code>Automatic</code>	method used to determine configurations to evaluate
<code>RandomSeeding</code>	<code>1234</code>	what seeding of pseudorandom generators should be done internally

Possible settings for Method include:

<code>Automatic</code>	automatically choose the method
<code>"MaxExpectedImprovement"</code>	maximize expected improvement over current best value
<code>"MaxImprovementProbability"</code>	maximize improvement probability over current best value

Mathematica Code 7.20

```
Input      (* The code showcases the versatility of the Predict function by demonstrating its usage with diverse types of training data. Each example employs a distinct format of training data, including single-input rules, multi-feature examples, input-output lists, and nominal value mappings, illustrating the function's adaptability to different data structures: *)
(* Specify the training set as a list of rules between an input example and the output value: *)
predictorWithSingleInput=Predict[{10->3.5,20->7.2,30->10.8,40->14.1,50->18.2}]

(* Applying predictorWithSingleInput to a new input 15: *)
predictorWithSingleInput[15]

(* Each example can contain a list of features: *)
predictorWithMultipleInputs=Predict[{{-0.3,0.8}->2.1,{0.1,-0.5}->3.2,{0.5,-0.2}->1.5,{0.2,0.3}->4.0,{0.5,-0.1}->4.7}]
```

	(* Applying predictorWithMultipleInputs to a new input {0.1,-0.3}: *) predictorWithMultipleInputs[{0.1,-0.3}]
	(* Specify the training set as a list of rule between a list of inputs and a list of outputs: *) predictorWithInputList=Predict[{10,20,30,40,50}->{3.5,7.2,10.8,14.1,18.2}]
	(* Applying predictorWithInputList to a new input 45: *) predictorWithInputList[45]
	(* Predict a variable from a nominal value: *) nominalPredictor=Predict[{"C"->6.3,"D"->7.8,"C"->8.1,"D"->9.5,"D"->10.2}]
	(* Applying nominalPredictor to a new input "D": *) nominalPredictor["D"]
Output	PredictorFunction[ Input type: Numerical Method: NearestNeighbors Number of training examples: 5]
Output	5.35
Output	PredictorFunction[ Input type: NumericalVector (2) Method: DecisionTree Number of training examples: 5]
Output	4.35
Output	PredictorFunction[ Input type: Numerical Method: NearestNeighbors Number of training examples: 5]
Output	16.15
Output	PredictorFunction[ Input type: Nominal Method: GradientBoostedTrees Number of training examples: 5]
Output	8.38009

Mathematica Code 7.21

```
Input (* The code demonstrates Gaussian Process (GP) regression, a method well-suited
for predictive modeling with limited data, particularly for quantifying uncertainty
in predictions. The code trains a GP model on a small dataset, retrieves detailed
model information, and visualizes the probability density functions (PDFs) to
analyze predictive distributions at specific and varied input values. It further
assesses model accuracy and reliability by comparing predicted means and confidence
intervals against actual data: *)
```

```
(* Train a predictor on labeled examples using a Gaussian Process method: *)
trainingData={-1.3->1.3,1.3->1.4,3.2->1.8,4.6->1.7};
predictor=Predict[trainingData,Method->"GaussianProcess"];
```

```
(* Retrieve and display general information about the predictor, like the model
type, training points, etc: *)
Information[predictor]
```

```
(* Get the conditional distribution of the predicted value for a specific feature:
*)
predictedDistribution=predictor[2,"Distribution"]
```

```
(* Plot the probability density function (PDF) of the distribution for a given
feature value: *)
```

```
Plot[
  PDF[predictedDistribution,y],
  {y,0,3},
  PlotRange->Full,
  ImageSize->250,
  FrameLabel->{"Value","Probability Density"},
  PlotLabel->"Probability Density Plot: \n Predicted Distribution at Feature Value 2"]

(* Calculate and plot the PDFs for a range of input values to visualize how predictive uncertainty varies: *)
pdfs=Table[
  PDF[predictor[i,"Distribution"],y],
  {i,-2,5,0.1}
];

Plot[
  pdfs,
  {y,1,2},
  PlotRange->Full,
  ImageSize->300,
  PlotStyle->{Thickness[0.001]},
  FrameLabel->{"Value","Probability Density"},
  PlotLabel->"Predicted Probability Density Functions\n for Various Input Values"]

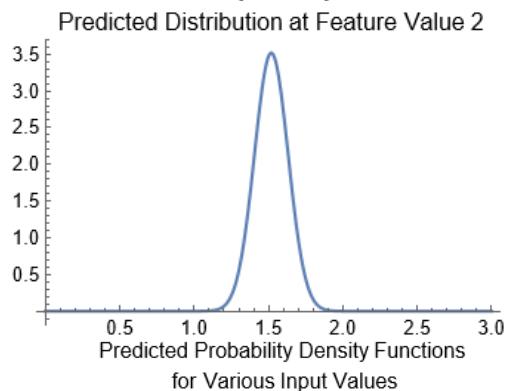
(* Visualize the predicted mean, upper, and lower bounds of confidence intervals across the input range and compare these predictions with the original training data: *)
Show[
  Plot[
    {
      (* Predicted mean: *)
      predictor[x],
      (* Upper bound of confidence interval: *)
      predictor[x]+StandardDeviation[predictor[x,"Distribution"]],
      (* Lower bound of confidence interval: *)
      predictor[x]-StandardDeviation[predictor[x,"Distribution"]]
    },
    {x,-2,6},
    PlotStyle->{Blue,{Opacity[0.3],Purple},{Opacity[0.3],Purple}},
    (* Fill between the upper and lower confidence intervals: *)
    Filling->{2->{3}},
    FillingStyle->{Opacity[0.001],Blue},
    Exclusions->False,
    ImageSize->300,
    PerformanceGoal->"Speed",
    PlotLegends->{"Prediction","Confidence Interval"},
    PlotLabel->"Gaussian Process Prediction with Confidence Intervals"
  ],
  ListPlot[
    (* Plot the actual training data points for comparison: *)
    List@@@trainingData,
    PlotStyle->{Red,PointSize[Large]},
    PlotLegends->{"Data"}
  ]
]
```

Output

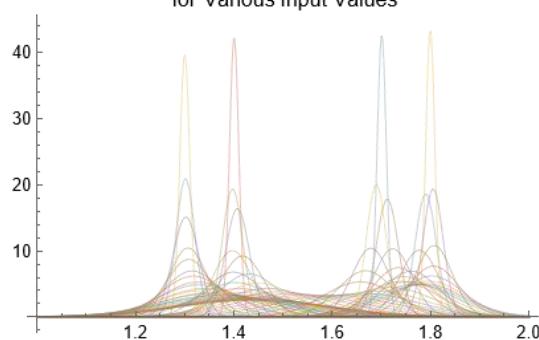
Predictor information	
Data type	Numerical
Standard deviation	0.217 ± 0.086
Method	GaussianProcess
Single evaluation time	4.58 ms/example
Batch evaluation speed	34.8 examples/ms
Loss	-0.190 ± 0.24
Model memory	134. kB
Training examples used	4 examples
Training time	652. ms

Output
Output

NormalDistribution[1.51826, 0.113466]
Probability Density Plot:

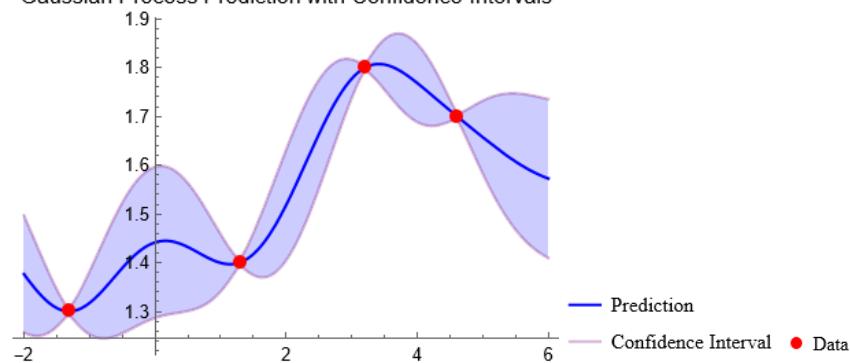


Output



Output

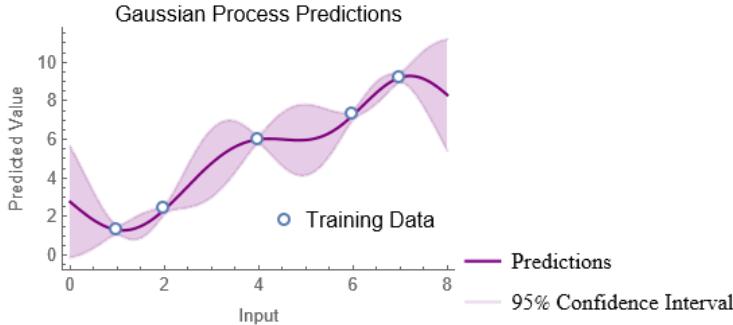
Gaussian Process Prediction with Confidence Intervals



Mathematica Code 7.22

Input	(* Above Code: Visualizes the predicted mean along with upper and lower confidence bounds based on standard deviation. This Code: Uses quantiles to determine and plot confidence intervals, explicitly calculating them based on a defined confidence level (e.g., 95%), providing a more customized view of uncertainty: *) (* Define the dataset: *) dataset={1->1.3,2->2.4,4->6,6->7.3,7->9.2}; (* Train a Gaussian Process predictive model on the dataset: *) predictiveModel=Predict[dataset, Method->"GaussianProcess"]; (* Make a prediction for the value at x=4.5: *) predictedValue=predictiveModel[4.5] (* Calculate the distribution of the prediction at x=4.5: *) predictionDistribution=predictiveModel[4.5,"Distribution"] (* Define the confidence level: *) confidenceLevel=0.95; (* Plot the predictions and confidence intervals: *) Show[Plot[{ predictiveModel[x], Quantile[predictiveModel[x,"Distribution"],(1+confidenceLevel)/2], Quantile[predictiveModel[x,"Distribution"],(1-confidenceLevel)/2] }, {x,0,8}, PlotStyle->{Purple,{Opacity[0.15],Purple},{Opacity[0.15],Purple}}, Filling->{2->{3}}, FillingStyle->{Opacity[0.001],Purple}, Exclusions->False, FrameLabel->{"Input","Predicted Value"}, PlotLabel->"Gaussian Process Predictions", PlotLegends->{"Predictions",ToString[Round[100 confidenceLevel]]<>"% Confidence Interval"}, Frame->{True,True,False}, FrameTicksStyle->Thin, FrameStyle->Thin, ImageSize->250, PerformanceGoal->"Speed"], (* Plot the original data points: *) ListPlot[Apply[List,dataset,{1}], PlotMarkers->"OpenMarkers", PlotLegends->Placed[{"Training Data"},{0.75,0.2}]]]
Output	6.00389
Output	NormalDistribution[6.00389,0.707892]

Output



Mathematica Code 7.23

Input

```
(* The code aims to generate a synthetic dataset, train multiple Gaussian Process
regression models using various covariance functions, and visualize their
predictions alongside confidence intervals. It creates a dataset by mapping values
from -10 to 10 to a noisy sine function, trains models with covariance functions
such as Squared Exponential, Periodic, Rational Quadratic, Matern 5/2, and Matern
3/2, and predicts over the same range. Each model's predictions and 95% confidence
intervals are plotted against the actual sine function and the original data points
for comparison, allowing an analysis of how different covariance functions affect
the model's predictive accuracy and uncertainty estimation: *)

(* Define the dataset: *)
dataset=Table[x->Sin[x]+RandomReal[0.3],{x,-10,10,1}];

(* Train a Gaussian Process predictive model on the dataset: *)
{predictorSE,predictorPeriodic,predictorRQ,predictorMatern52,predictorMatern32}=P
redict[
  dataset,
  Method->{"GaussianProcess","CovarianceType"->#}]&/@{
  "SquaredExponential",
  "Periodic",
  "RationalQuadratic",
  "Matern5/2",
  "Matern3/2"
};

(* Define the confidence level: *)
confidenceLevel=0.95;

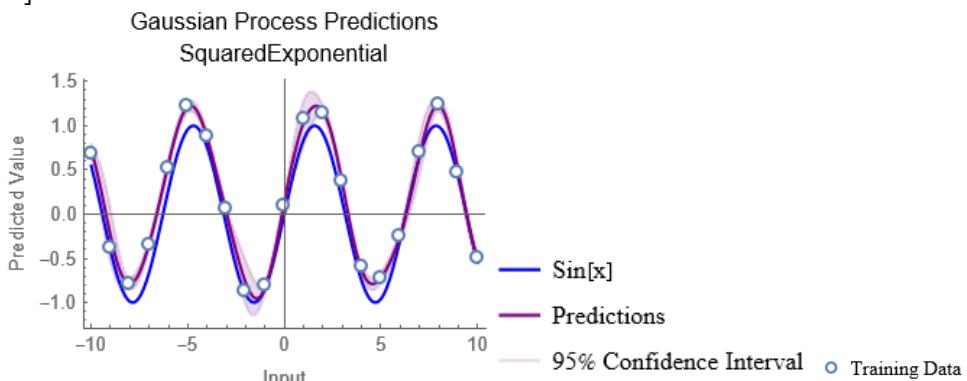
(* Plot the predictions and confidence intervals: *)
Table[Show[
  Plot[
    {Sin[x],
     predictiveModel[[1]][x],
     Quantile[predictiveModel[[1]][x,"Distribution"],(1+confidenceLevel)/2],
     Quantile[predictiveModel[[1]][x,"Distribution"],(1-confidenceLevel)/2]
    },
    {x,-10,10},
    PlotStyle->{Blue,Purple,{Opacity[0.15],Purple},{Opacity[0.15],Purple}},
    Filling->{3->{4}},
    FillingStyle->{Opacity[0.001],Purple},
    Exclusions->False,
    FrameLabel->{"Input","Predicted Value"},
    PlotLabel->"Gaussian Process Predictions\n"<>predictiveModel[[2]],
    PlotLegends->{"Sin[x]","Predictions",ToString[Round[100 confidenceLevel]]<>%
    Confidence Interval"},
    Frame->{True,True,False,False},
    PlotRange->{-10,10}
  ]
],{i,5}],{i,1,5}];
```

```

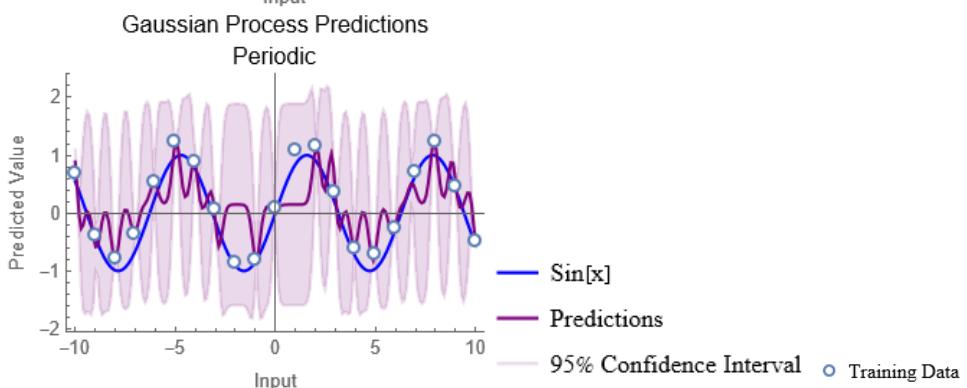
FrameTicksStyle->Thin,
FrameStyle->Thin,
ImageSize->250,
PerformanceGoal->"Speed"
],
(* Plot the original data points: *)
ListPlot[
  Apply[List,dataset,{1}],
  PlotMarkers->"OpenMarkers",
  PlotLegends->{"Training Data"}
]
],{predictiveModel,{{
  predictorSE,"SquaredExponential"},{
  predictorPeriodic,"Periodic"},{
  predictorRQ,"RationalQuadratic"},{
  predictorMatern52,"Matern5/2"},{
  predictorMatern32,"Matern3/2"}}
}
]
]

```

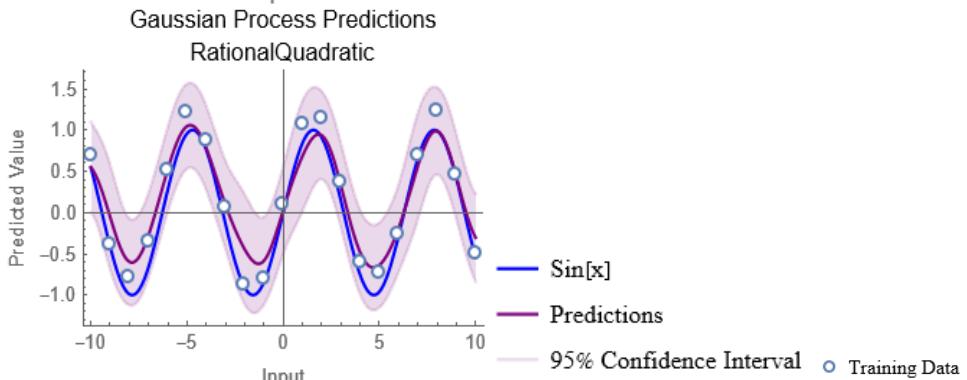
Output

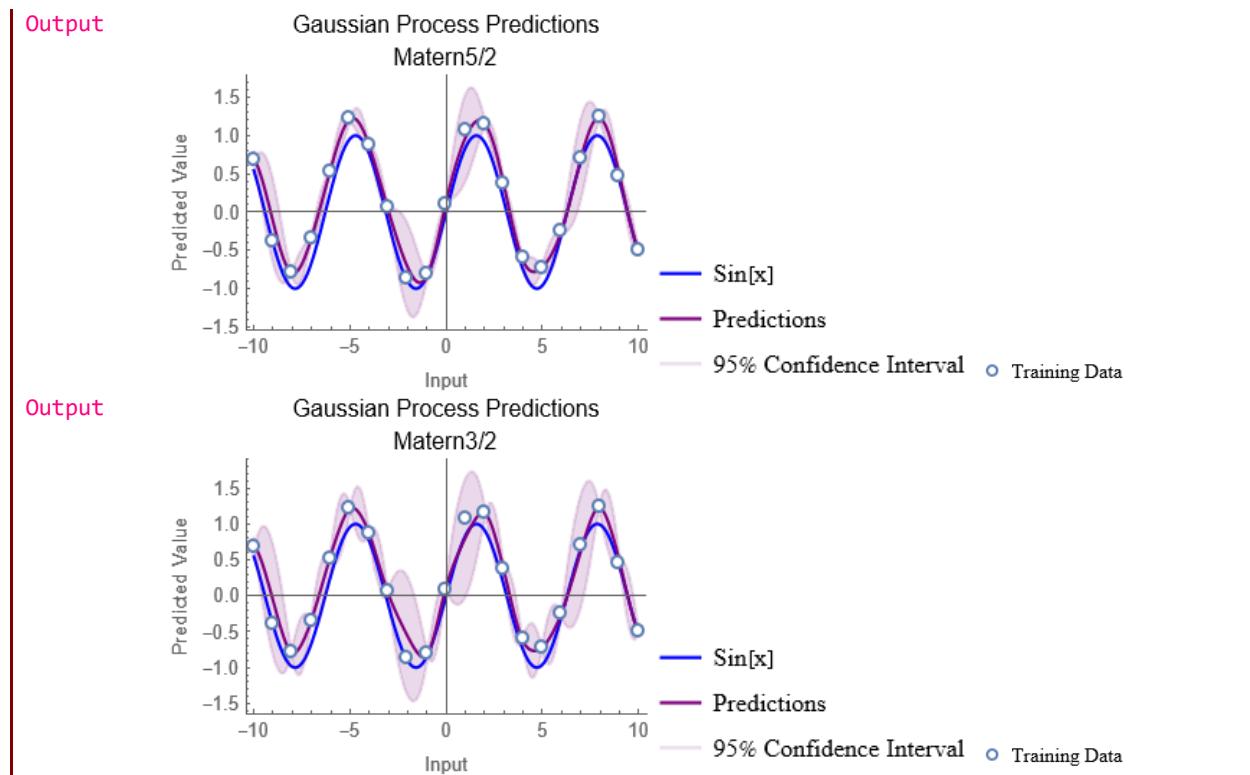


Output



Output



**Mathematica Code 7.24**

Input

```
(* The code demonstrates Bayesian optimization to find the maximum of a complex
function, sin(3x)·Exp[-0.3 x], across an interval [0,10]. This function,
characterized by its oscillatory and decaying nature, poses a significant challenge
due to multiple local maxima. The code efficiently identifies the optimal input
configuration and the corresponding maximum function value, showcasing Bayesian
optimization's ability to navigate complex landscapes. Additionally, it visualizes
the function and the optimization outcome, enhancing the understanding of the
process and its effectiveness: *)

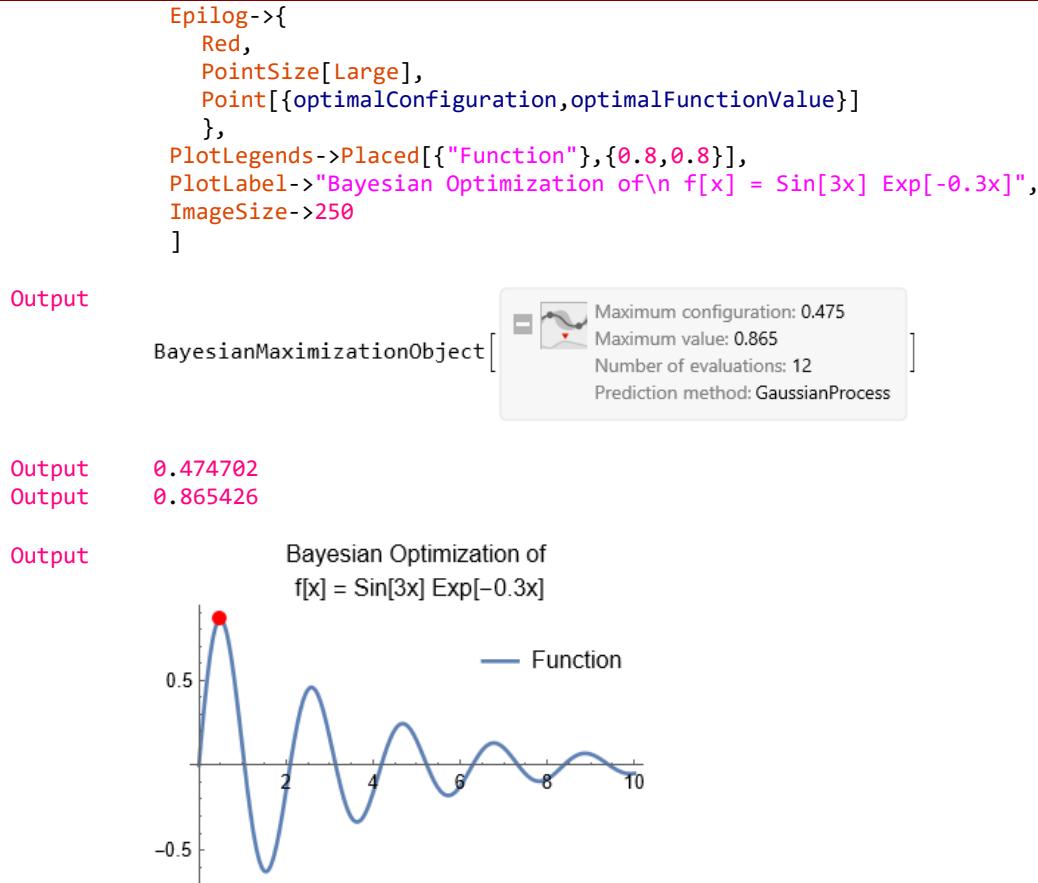
(* Define the function to be optimized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x]

(* Perform Bayesian Maximization on the function: *)
optimizationResult=BayesianMaximization[objectiveFunction,Interval[{0,10}]]

(* Extract the optimal configuration (the x-value) from the optimization result.
This value represents the input at which the function is estimated to reach its
maximum within the defined interval: *)
optimalConfiguration=optimizationResult["MaximumConfiguration"]

(* Retrieve the highest function value achieved during the optimization. This value
corresponds to the maximum output of the function at the optimal configuration,
indicating the peak performance observed by the Bayesian optimization process: *)
optimalFunctionValue=optimizationResult["MaximumValue"]

(* Plot the function and the maximum point: *)
Plot[
  Sin[3 x] Exp[-0.3 x],
  {x,0,10},
  PlotStyle->Thick,
  PlotRange->All,
```

**Mathematica Code 7.25**

Input (* The code utilizes Bayesian optimization to find the maximum value of the function $f(x)=\sin(3x)\cdot\text{Exp}[-0.3x]$ within a specified interval. By performing Bayesian optimization on the function, it identifies the input configuration (x-value) from a predefined set where the function is estimated to reach its maximum. Small differences in the initial configurations lead to variations in the estimated maximum value. The code then generates a plot that visually represents the function over a defined range of x-values and marks the point of maximum with a red dot: *)

```

(* Define the function to be optimized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x]

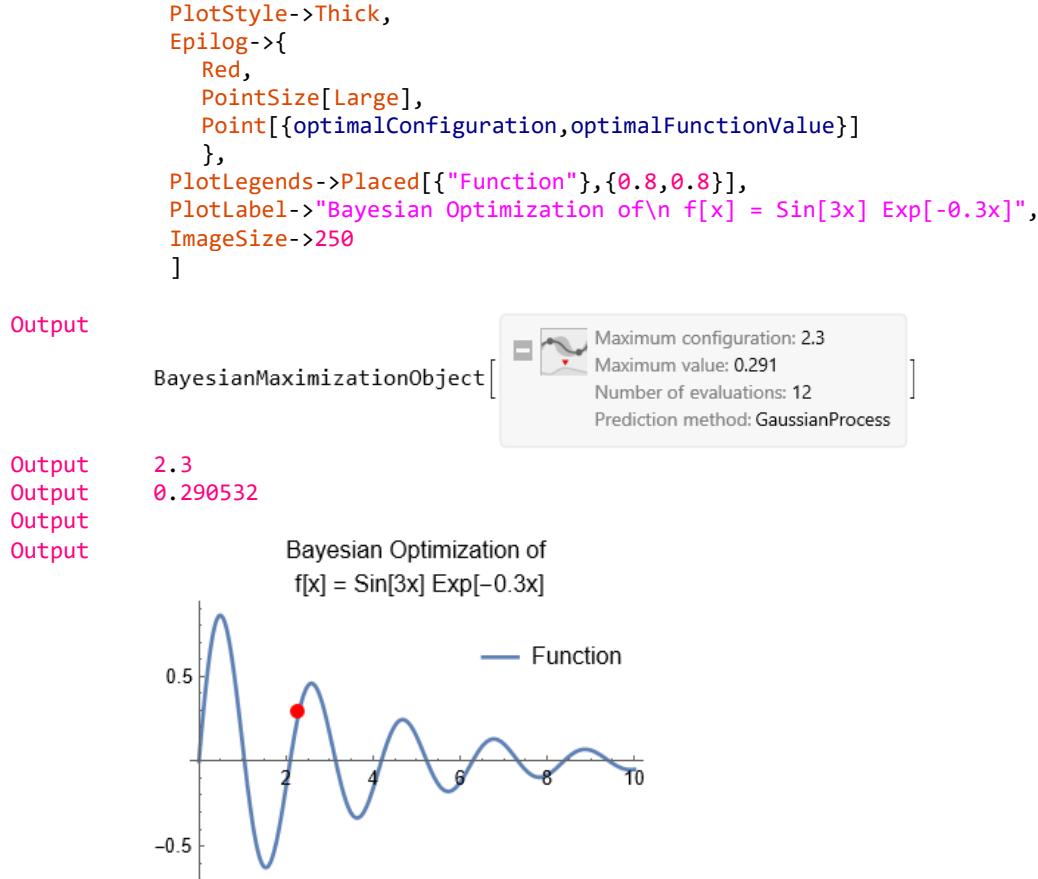
(* Perform Bayesian Maximization on the function over the set of configurations
(list of data elements): *)
optimizationResult=BayesianMaximization[objectiveFunction,{5.7,1.1,3.4,6.8,2.3,1.2}]

(* Get the maximum configuration over the set: *)
optimalConfiguration=optimizationResult["MaximumConfiguration"]

(* Retrieve the highest function value achieved during the optimization: *)
optimalFunctionValue=optimizationResult["MaximumValue"]

(* Plot the function and the maximum point: *)
Plot[
  objectiveFunction[x],
  {x,0,10},

```

**Mathematica Code 7.26**

```

Input (* Small differences in the initial configurations lead to variations in the
       estimated maximum value. In this case we use Table[RandomReal[{5,10}],10]: *)

(* Define the function to be optimized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x]

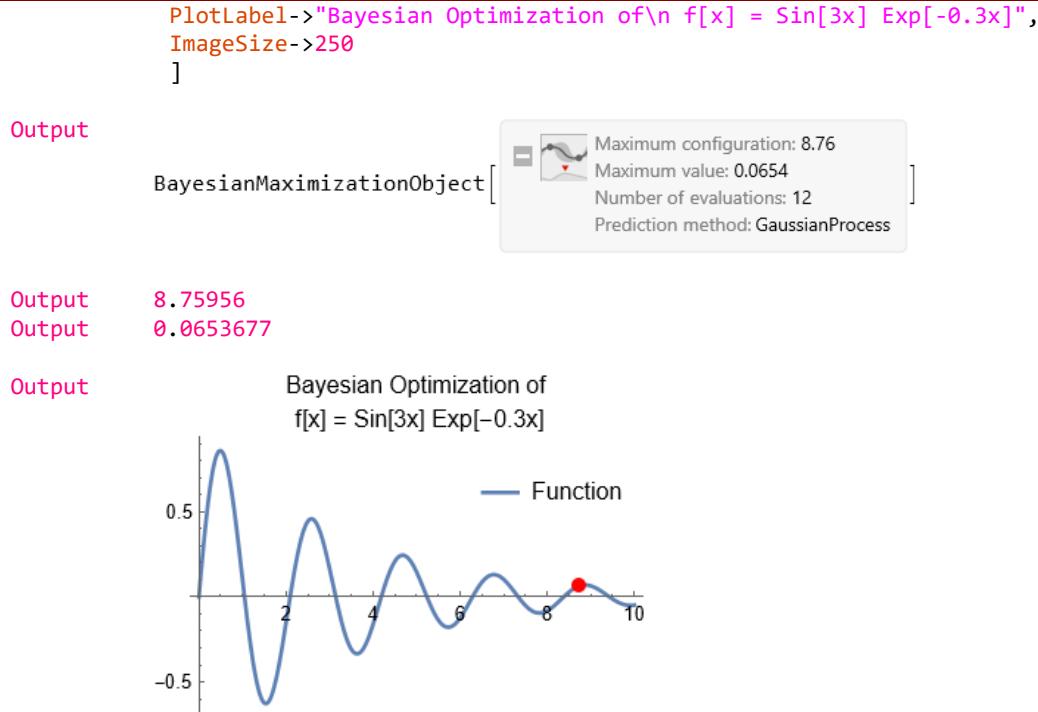
(* Perform Bayesian Maximization on the function: *)
optimizationResult=BayesianMaximization[objectiveFunction,Table[RandomReal[{5,10}],
{10}]] 

(* Get the maximum configuration over the set: *)
optimalConfiguration=optimizationResult["MaximumConfiguration"]

(* Retrieve the highest function value achieved during the optimization: *)
optimalFunctionValue=optimizationResult["MaximumValue"]

(* Plot the function and the maximum point: *)
Plot[
  objectiveFunction[x],
  {x,0,10},
  PlotStyle->Thick,
  Epilog->{
    Red,
    PointSize[Large],
    Point[{optimalConfiguration,optimalFunctionValue}]
  },
  PlotLegends->Placed[{"Function"},{0.8,0.8}],

```

**Mathematica Code 7.27**

```

Input (* The code aims to minimize the function f(x)=sin(3x)-Exp[\[Minus]0.3 x] using
       Bayesian optimization within the interval [0,20]. By defining the objective
       function and applying Bayesian minimization, the code seeks to identify the input
       configuration that results in the lowest function value. After the optimization
       process, it retrieves the configuration and the corresponding minimum function
       value. The code then generates a plot of the function over the interval [0,20] and
       marks the minimum point with a red dot, providing a visual representation of the
       optimization outcome: *)

(* Define the function to be optimized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x]

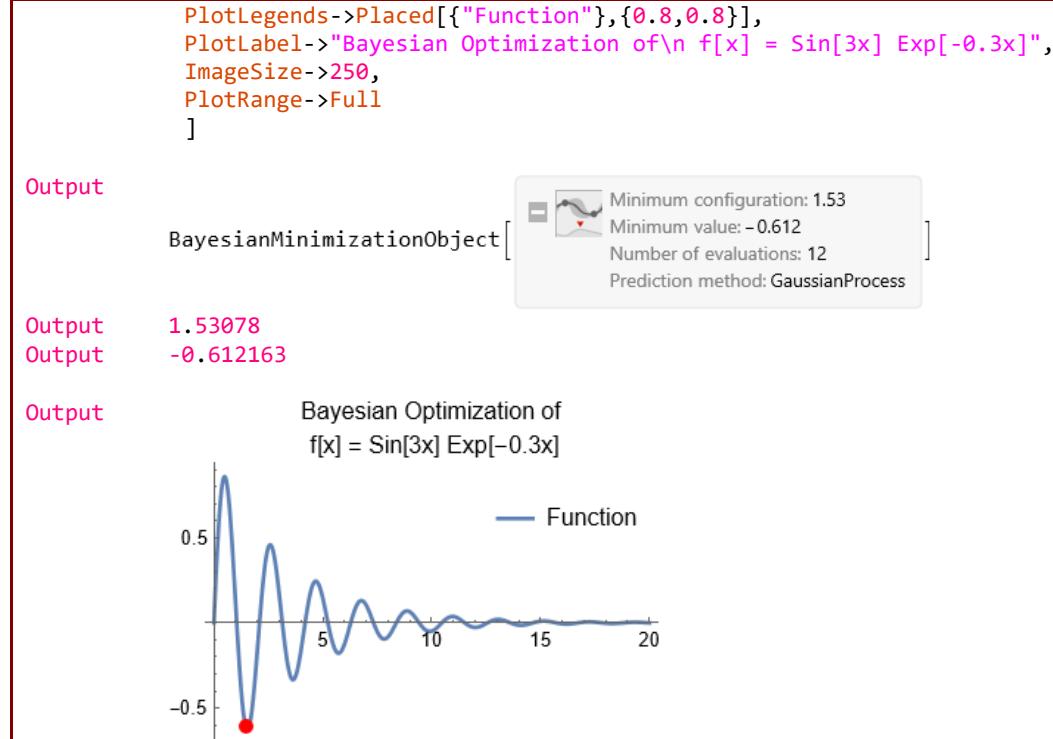
(* Perform Bayesian Minimization on the function: *)
optimizationResult=BayesianMinimization[objectiveFunction,Interval[{0,20}]]

(* Use the resulting BayesianMinimizationObject[...] to get the estimated minimum
   configuration:*)
optimalConfiguration=optimizationResult["MinimumConfiguration"]

(* Retrieve the smallest function value achieved during the optimization: *)
optimalFunctionValue=optimizationResult["MinimumValue"]

(* Plot the function and the minimum point: *)
Plot[
  objectiveFunction[x],
  {x,0,20},
  PlotStyle->Thick,
  Epilog->{
    Red,
    PointSize[Large],
    Point[{optimalConfiguration,optimalFunctionValue}]
  },
]

```

**Mathematica Code 7.28**

Input

```

(* The code explores and compares two different methods for Bayesian optimization
applied to the objective function f(x)=sin(3x)-Exp[\[Minus]0.3 x] over the
interval[0,10]. Firstly, it employs the "MaxExpectedImprovement" method, which aims
to maximize the expected improvement in function value over the search space, and
retrieves the maximum configuration and corresponding highest function value
achieved during optimization. Secondly, it applies the "MaxImprovementProbability"
method, focusing on maximizing the probability of improvement over the current best
value, and similarly retrieves the maximum configuration and function value. By
comparing the outcomes of these two methods, the code enables an assessment of
their effectiveness in identifying the global maximum of the objective function,
providing insights into the performance and behavior of different Bayesian
optimization strategies: *)

```

```

(* Define the objective function over an interval region: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x]

interval=Interval[{0,10}];

(* Specify the method for exploring configurations: *)
bayesianOptimization1=BayesianMaximization[
  objectiveFunction,
  interval,
  Method->"MaxExpectedImprovement"
]

(* Get the maximum configuration over the set: *)
optimalConfiguration1=bayesianOptimization1["MaximumConfiguration"]

(* Retrieve the highest function value achieved during the optimization: *)
optimalFunctionValue1=bayesianOptimization1["MaximumValue"]

```

```
(* Specify a different method: *)
bayesianOptimization2=BayesianMaximization[
  objectiveFunction,
  interval,
  Method->"MaxImprovementProbability"
]

(* Get the maximum configuration over the set: *)
optimalConfiguration2=bayesianOptimization2["MaximumConfiguration"]

(* Retrieve the highest function value achieved during the optimization: *)
optimalFunctionValue2=bayesianOptimization2["MaximumValue"]

Output
      BayesianMaximizationObject[ Maximum configuration: 0.475
                                    Maximum value: 0.865
                                    Number of evaluations: 12
                                    Prediction method: GaussianProcess]
```

Output 0.474702
 Output 0.865426

```
Output
      BayesianMaximizationObject[ Maximum configuration: 0.475
                                    Maximum value: 0.851
                                    Number of evaluations: 12
                                    Prediction method: GaussianProcess]
```

Output 0.474702
 Output 0.851354

Mathematica Code 7.29

```
Input    (* The code aims to utilize Bayesian optimization to find the maximum value of the
          objective function  $f(x)=\sin(3x)\cdot\text{Exp}[-0.3x]$  over the interval  $[0,5]$ . Through
          BayesianMaximization, various properties of the optimization process are explored,
          including evaluation history, exploration method, and the probabilistic model of
          the function. Additionally, the code retrieves information such as the next best
          configuration to explore, offering guidance for further optimization if needed.
          Finally, the code visualizes both the original objective function and the
          predictions of the probabilistic model over the interval  $[0,5]$ , providing insights
          into the behavior of the function and the effectiveness of the optimization process:*)
```

```
(* Define the objective function to be maximized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x];
```

```
(* Define the region over which to maximize the function: *)
optimizationRegion=Interval[{0.,5.}];
```

```
(* Perform Bayesian Maximization to maximize the objective function over the
specified region: *)
bayesianOptimization=BayesianMaximization[objectiveFunction,optimizationRegion]
```

```
(* Get the list of available properties to query: *)
availableProperties=bayesianOptimization["Properties"]
```

```
(* Get the history of evaluations during the optimization process: *)
evaluationHistory=bayesianOptimization["EvaluationHistory"]
```

```
(* Get information about the method used for exploration: *)
explorationMethod=bayesianOptimization["Method"]
```

```
(* Get the current probabilistic model of the function: *)
```

```

probabilisticModel=bayesianOptimization["PredictorFunction"]

(* Find the best configuration to explore if the maximization were continued: *)
nextConfiguration=bayesianOptimization["NextConfiguration"]

(* Find a list of properties simultaneously: *)
simultaneousProperties=bayesianOptimization[{"MaximumConfiguration","MaximumValue",
", "Method"}]

(* Visualize the function and the model's predictions: *)
Plot[
{objectiveFunction[x],probabilisticModel[x]},
{x,0,5},
Frame->{True,True,False,False},
FrameLabel->{"Configuration","Function Value"},
PlotLegends->{"Function","Model"},
ImageSize->250,
Epilog->{
  Red,
  PointSize[Large],
  Point[{simultaneousProperties[[1]],simultaneousProperties[[2]]}]
  }
]

```

Output

```
BayesianMaximizationObject[ Maximum configuration: 0.514
Maximum value: 0.857
Number of evaluations: 12
Prediction method: GaussianProcess]
```

Output

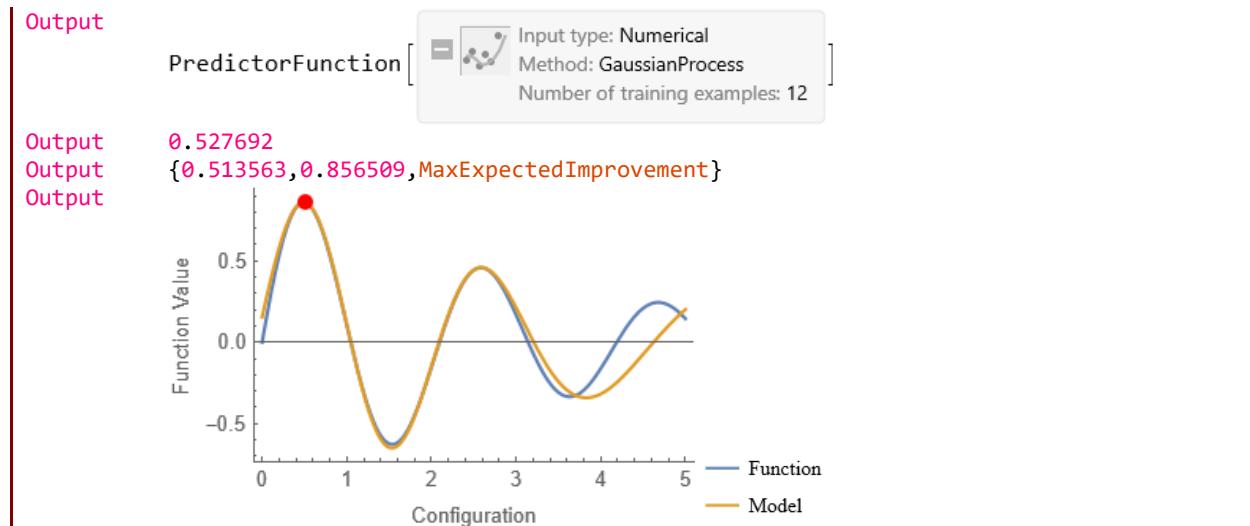
```
{MaximumConfiguration,MaximumValue,EvaluationHistory,PredictorFunction,ObjectiveFunction,Method,Properties,NextConfiguration}
```

Output

Configuration	Value
2.60982	0.45692
0.431117	0.845077
0.414024	0.835906
1.00612	0.0909042
0.613076	0.802204
4.84709	0.214788
3.70555	-0.326601
0.643079	0.772141
1.74939	-0.508755
0.513563	0.856824
0.542566	0.848411
2.37009	0.361455

Output

```
MaxExpectedImprovement
```

**Mathematica Code 7.30**

Input

```
(* The code conducts Bayesian minimization to find the minimum of the specified
objective function  $f(x)=\sin(3x)\cdot\text{Exp}[-0.3x]$  over the interval  $[0,5]$ . Through
BayesianMinimization, the code retrieves various properties of the optimization
process, including the estimated minimum configuration and value, as well as the
history of evaluations made during the minimization. Additionally, the code
visualizes the objective function and the probabilistic model on the same plot,
allowing for an assessment of how well the function is modeled, particularly near
the minimum: *)

(* Define the objective function to be minimized: *)
objectiveFunction[x_]:=Sin[3 x] Exp[-0.3 x];

(* Define the region over which to minimize the function: *)
minimizationRegion=Interval[{0,5}];

(* Perform Bayesian Minimization to minimize the objective function over the
specified region: *)
bayesianMinimization=BayesianMinimization[objectiveFunction,minimizationRegion];

(* Query available properties of the BayesianMinimizationObject: *)
availableProperties=bayesianMinimization["Properties"]

(* Retrieve the estimated minimum configuration: *)
minimumConfiguration=bayesianMinimization["MinimumConfiguration"]

(* Retrieve the estimated minimum value: *)
minimumValue=bayesianMinimization["MinimumValue"]

(* Retrieve the list of configurations and function values explored during the
minimization: *)
evaluationHistory=bayesianMinimization["EvaluationHistory"]//Normal

(* Retrieve information about a list of properties simultaneously: *)
simultaneousProperties=bayesianMinimization[{"NextConfiguration","Method"}]

(* Retrieve the probabilistic model of the function: *)
probabilisticModel=bayesianMinimization["PredictorFunction"];

(* Visualize the function and the model's predictions: *)
```

```

Show[
  (* Visualize how well the function is modeled, particularly near the minimum: *)
  Plot[
    {objectiveFunction[x],probabilisticModel[{x}]},
    {x,0,5},
    PlotLegends->{"Function","Model"},
    Frame->{True,True,False,False},
    FrameLabel->{"Configuration","Function Value"},
    Epilog->{
      Red,
      PointSize[Large],
      Point[{minimumConfiguration,minimumValue}]
    }
  ],
  (* Plot the evaluation history: *)
  ListPlot[
    evaluationHistory,
    PlotMarkers->"OpenMarkers",
    PlotLegends->{"Evaluation History"}
  ],
  PlotLabel->"Objective Function, Probabilistic Model\n and Evaluation History",
  ImageSize->250
]

Output {MinimumConfiguration,MinimumValue,EvaluationHistory,PredictorFunction,ObjectiveFunction,Method,Properties,NextConfiguration}

Output 1.56664
Output -0.61745

Output {<|Configuration->2.60982,Value->0.45692|>,<|Configuration->0.431117,Value->0.845077|>,<|Configuration->2.71878,Value->0.42229|>,<|Configuration->3.83488,Value->-0.276356|>,<|Configuration->4.70554,Value->0.243686|>,<|Configuration->3.71624,Value->-0.32412|>,<|Configuration->1.64323,Value->-0.596447|>,<|Configuration->1.40564,Value->-0.577055|>,<|Configuration->1.74939,Value->-0.508755|>,<|Configuration->1.56664,Value->-0.624958|>,<|Configuration->1.38428,Value->-0.559475|>,<|Configuration->4.84801,Value->0.214477|>}

Output {0.0582229,MaxExpectedImprovement}

          Objective Function, Probabilistic Model
          and Evaluation History


```

Unit 7.5

Automated Hyperparameter Tuning with Mathematica

In machine learning, hyperparameter optimization is crucial for enhancing model performance, and it can be approached using methods like grid search, random search, and Bayesian optimization. Grid search evaluates every possible combination of a predefined set of hyperparameters, ensuring a thorough exploration of the space but often at a high computational cost. Random search, on the other hand, samples hyperparameters randomly from specified distributions, providing a more efficient alternative that can sometimes yield better results in less time. Bayesian optimization utilizes a probabilistic model to intelligently select hyperparameters based on past evaluations, aiming to find optimal solutions faster by focusing on more promising areas of the hyperparameter space. Each method has its advantages, and the choice depends on the specific constraints and goals of your project, such as computational resources and the sensitivity of model performance to hyperparameters. For grid search, you can use nested loops to iterate through combinations of hyperparameters. Similarly, for a random search, you can generate random samples of hyperparameters using Mathematica's random number generation functions like `RandomChoice`. Bayesian optimization, facilitated by functions like `BayesianMinimization/BayesianMaximization` in Mathematica, employs probabilistic models to sequentially select hyperparameters, updating the model based on observed results to iteratively explore the parameter space efficiently.

Mathematica Code 7.31

```
Input (* The code is designed to generate synthetic data by adding noise to a mathematical function, train and evaluate neural network models using a grid search over different configurations of learning rates, neuron counts, and optimization methods. It aims to identify the optimal model parameters by evaluating performance metrics like Mean Squared Error (MSE) and R-Squared (RS). The code systematically tests various combinations of hyperparameters, trains the networks, and visually compares the best models' predictions against the actual test data to assess their effectiveness. This process exemplifies practical applications of machine learning techniques, including hyperparameter tuning, performance evaluation, and result visualization: *)  
  
(* Generate the training data using a Gaussian-modulated exponential function with added noise: *)  
  
dataFunction[x_]:=Exp[-x^2];  
noiseLevel=0.15;  
  
(* Training data with added noise: *)  
trainingData=Table[  
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],  
  {x,-3,3,0.01}  
];  
  
(* Validation data with added noise: *)  
validationData=Table[  
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],  
  {x,-3,3,0.1}  
];  
  
(* Test data similar to validation data for final model testing: *)  
testData=Table[  
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],  
  {x,-3,3,0.1}
```

```
];

(* Set up arrays for hyperparameter tuning: learning rates, neuron counts in layers
and optimization method: *)
learningRates={0.001,0.05,0.1};
neuronCounts={10,30,40};
optimizationmethod={"ADAM","SGD","RMSProp"};

(* Perform a grid search over the specified ranges of hyperparameters: *)
results=Table[
  Module[
    {net,trainedNet,validationMSE,validationRS},

    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        LinearLayer[50],Tanh,
        LinearLayer[neurons],Tanh,
        LinearLayer[1]
      }
    ];

    (* Train the network on training data and validate it: *)
    trainedNet=NetTrain[
      net,
      trainingData,
      All,
      ValidationSet->validationData,
      TrainingProgressMeasurements->{"MeanSquare","RSquared"},
      LossFunction->MeanSquaredLossLayer[],
      BatchSize->64,
      LearningRate->learningRate,
      Method->method,
      TimeGoal->5
    ];
  ];

  (* Calculate MSE on validation data for model evaluation: *)
  validationMSE=Values[trainedNet["ValidationMeasurements"]][[2]];
  validationRS=Values[trainedNet["ValidationMeasurements"]][[3]];

  {learningRate,neurons,method,validationMSE,validationRS,trainedNet}
  ],
  (* Grid of hyperparameters : *)
  {learningRate,learningRates},
  {neurons,neuronCounts},
  {method,optimizationmethod}
];

(* Flatten the results and sort by MSE and RS to find the best parameters: *)
sortedResultsByMSE=SortBy[Flatten[results,2],#[[4]]&];
sortedResultsByRS=SortBy[Flatten[results,2],#[[5]]&];

(* Extract the best parameter configurations based on MeanSquare (MSE): *)
bestParametersByMSE=First[sortedResultsByMSE]

(* Extract the best parameter configurations based on RSquared (RS): *)
bestParametersByRS=Last[sortedResultsByRS]

(* Retrieve the best model from the best parameters: *)
bestModelByMSE=bestParametersByMSE[[6]][["TrainedNet"]];
```

```
(* Retrieve the best model from the best parameters: *)
bestModelByRS=bestParametersByRS[[6]][["TrainedNet"]];

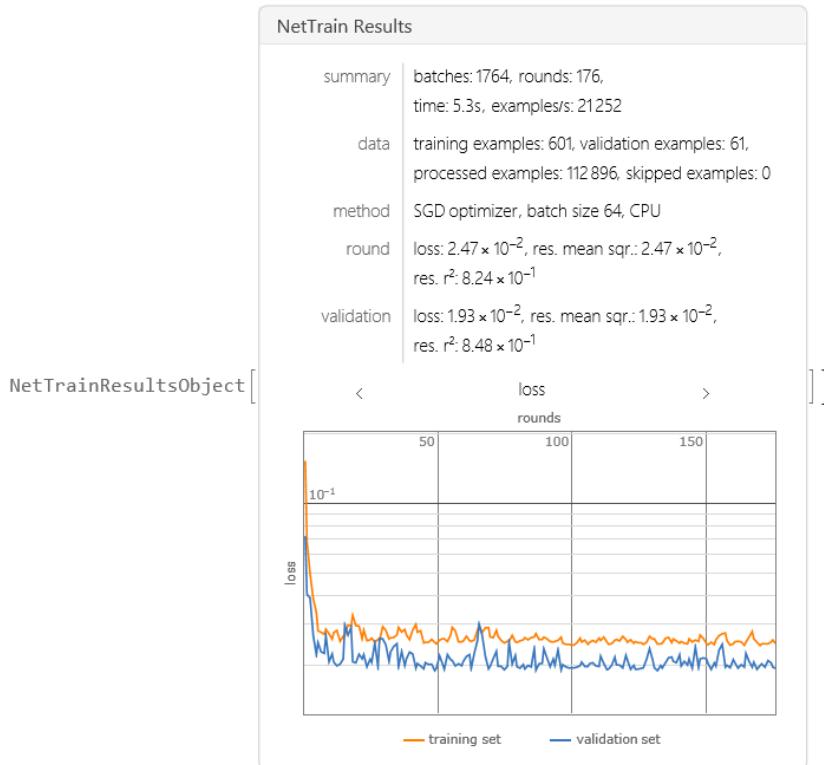
(* Visualize the best model's predictions along with the actual test data: *)
plotModelByMSE=Plot[
  bestModelByMSE[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

plotModelByRS=Plot[
  bestModelByRS[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

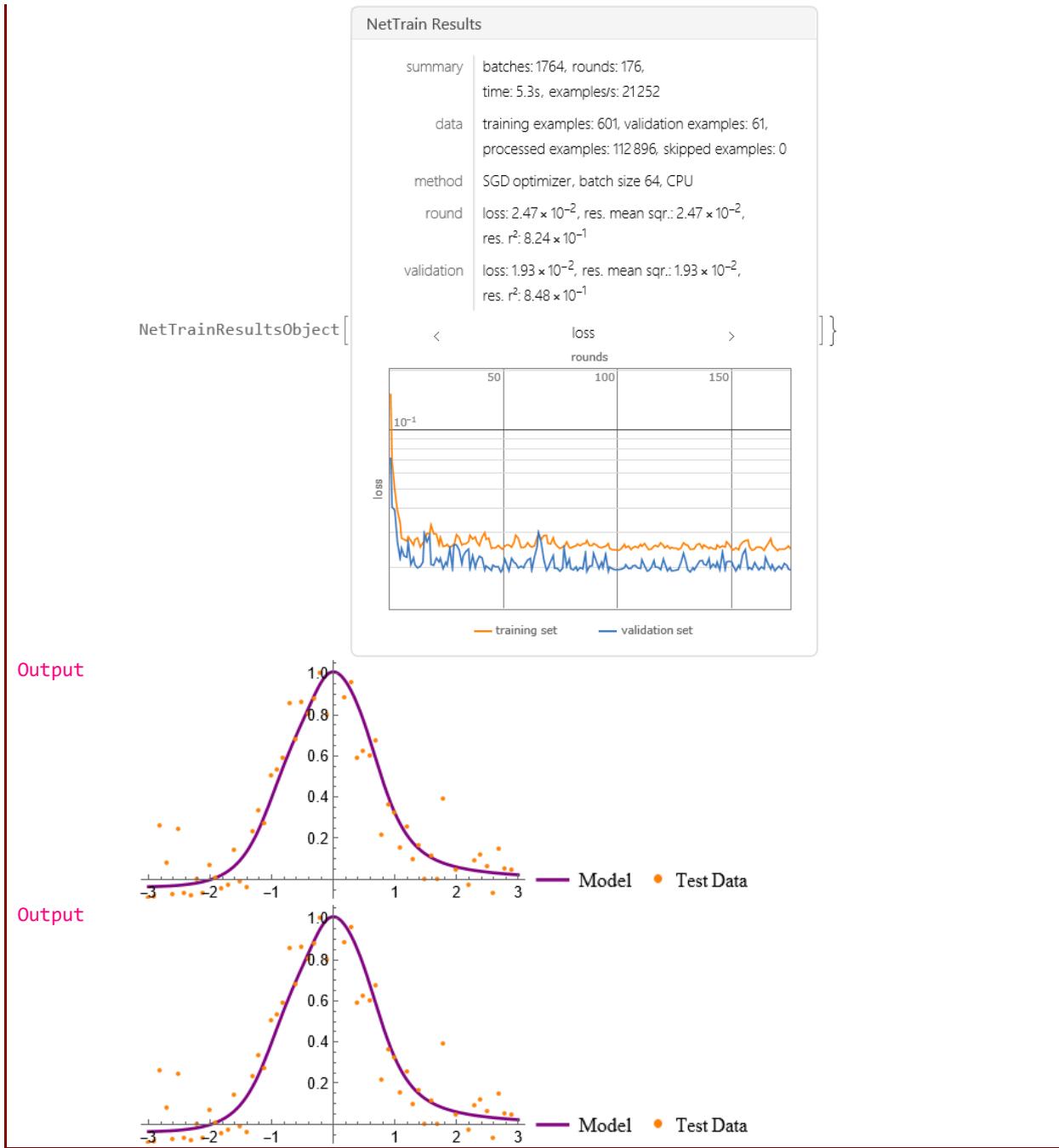
plotTestData=ListPlot[
  List@@@ testData,
  PlotStyle->Orange,
  PlotLegends->{"Test Data"}
];

(* Display the model and test data together for comparison: *)
Show[plotModelByMSE,plotTestData,ImageSize->250]
Show[plotModelByRS,plotTestData,ImageSize->250]
```

Output {0.05, 40, "SGD", 0.0193281, 0.84803,



Output {0.05, 40, "SGD", 0.0193281, 0.84803,

**Mathematica Code 7.32**

Input (* The goal of the code is to perform random search hyperparameter tuning for a neural network model. This involves generating synthetic datasets for training, validation, and testing, which simulate a Gaussian-modulated exponential function with noise. The code randomly explores different combinations of learning rates, neuron counts, and optimization methods to efficiently identify optimal configurations for the neural network. Each configuration is evaluated using metrics like Mean Squared Error (MSE) and R-Squared (RS). Ultimately, the code aims to find and visually compare the best neural network model against actual test data, assessing its ability to accurately predict and generalize from the given inputs. This approach allows for an efficient search across a potentially large parameter space, offering a practical alternative to the exhaustive grid search

```

method, especially useful in scenarios with limited computational resources or when
dealing with a high-dimensional hyperparameter space: *)

(* note that in this example, we will generate new data by using RandomVariate.
Therefore, we cannot accurately compare the results of random search and grid
search. Our objective is to elucidate the workings of each search method: *)

(* Generate the training data using a Gaussian-modulated exponential function with
added noise:*)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise:*)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Validation data with added noise:*)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Test data similar to validation data for final model testing:*)
testData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Set up arrays for hyperparameter tuning: learning rates, neuron counts in
layers and optimization method: *)
learningRates={0.001,0.05,0.1};
neuronCounts={10,30,40};
optimizationMethods={"ADAM","SGD","RMSProp"};

(* Number of random trials*)
numTrials=5;

(* Perform random search over the specified ranges of hyperparameters*)
results=Table[
  Module[
    {learningRate,neurons,method,net,trainedNet,validationMSE,validationRS},

    (* Randomly select hyperparameters *)
    learningRate=RandomChoice[learningRates];
    neurons=RandomChoice[neuronCounts];
    method=RandomChoice[optimizationMethods];

    (* Define a simple neural network structure:*)
    net=NetChain[
      {
        LinearLayer[50],Tanh,
        LinearLayer[neurons],Tanh,
        LinearLayer[1]
      }
    ];
    (* Train the network on training data and validate it: *)
    trainedNet=NetTrain[
      net,

```

```

    trainingData,
    All,
    ValidationSet->validationData,
    TrainingProgressMeasurements->{"MeanSquare", "RSquared"},
    LossFunction->MeanSquaredLossLayer[],
    BatchSize->64,
    LearningRate->learningRate,
    Method->method,
    TimeGoal->5
];

(* Calculate MSE on validation data for model evaluation: *)
validationMSE=Values[trainedNet["ValidationMeasurements"]][[2]];
validationRS=Values[trainedNet["ValidationMeasurements"]][[3]];

{learningRate,neurons,method,validationMSE,validationRS,trainedNet}
],
{numTrials}
];

(* Flatten the results and sort by MSE and RS to find the best parameters: *)
sortedResultsByMSE=SortBy[results,#[[4]]&];
sortedResultsByRS=SortBy[results,#[[5]]&];

(* Extract the best parameter configurations based on MeanSquare (MSE): *)
bestParametersByMSE=First[sortedResultsByMSE]

(* Extract the best parameter configurations based on RSquared (RS): *)
bestParametersByRS=Last[sortedResultsByRS]

(* Retrieve the best model from the best parameters: *)
bestModelByMSE=bestParametersByMSE[[6]]["TrainedNet"];
bestModelByRS=bestParametersByRS[[6]]["TrainedNet"];

(* Visualize the best model's predictions along with the actual test data: *)
plotModelByMSE=Plot[
  bestModelByMSE[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

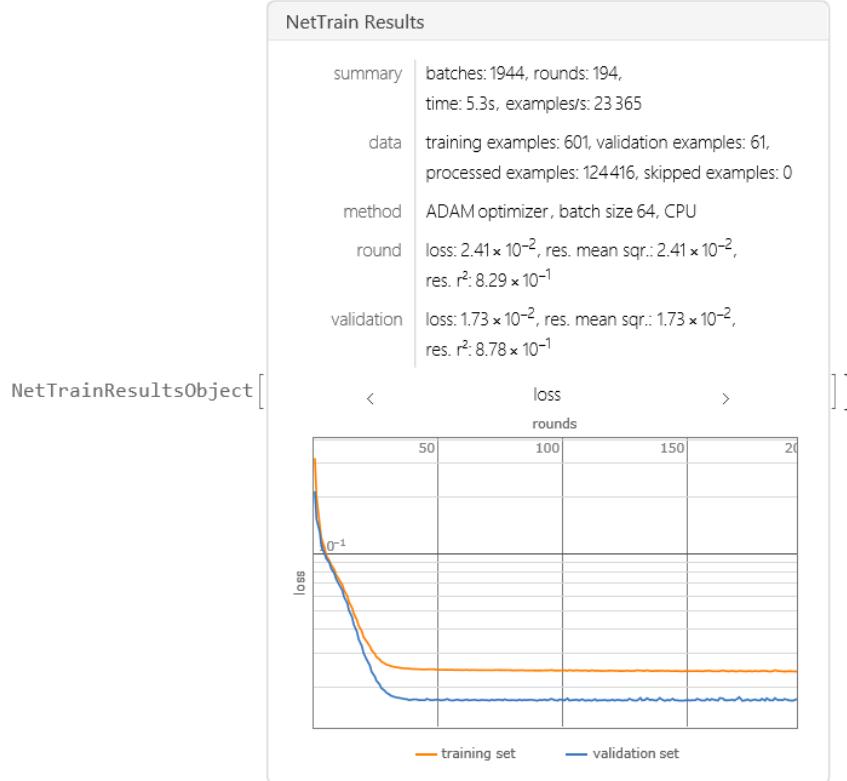
plotModelByRS=Plot[
  bestModelByRS[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

plotTestData=ListPlot[
  List@@@ testData,
  PlotStyle->Orange,
  PlotLegends->{"Test Data"}
];

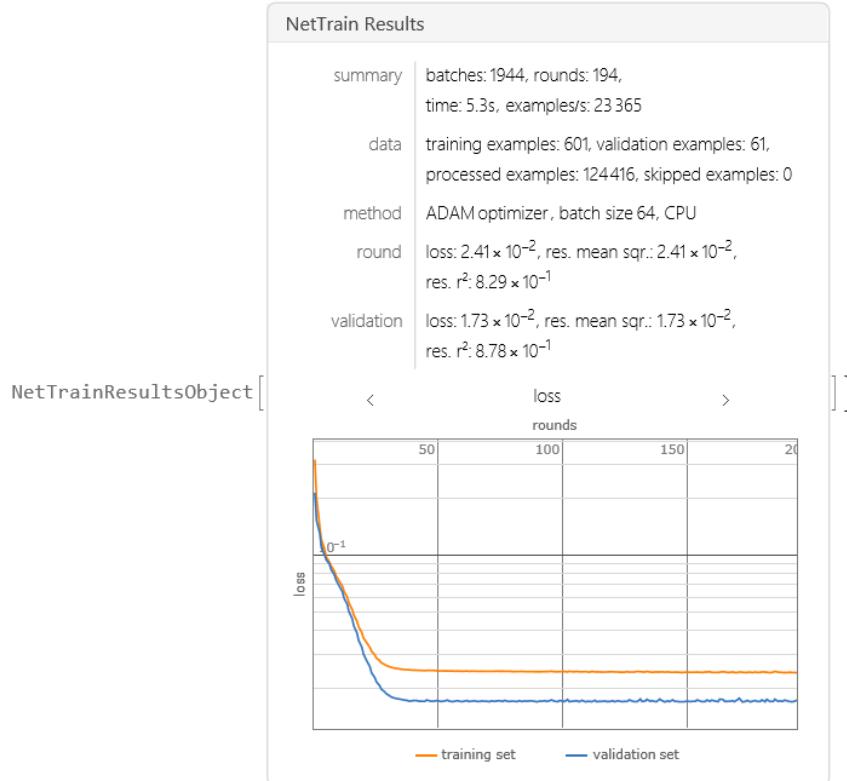
(* Display the model and test data together for comparison: *)
Show[plotModelByMSE,plotTestData,ImageSize->250]
Show[plotModelByRS,plotTestData,ImageSize->250]

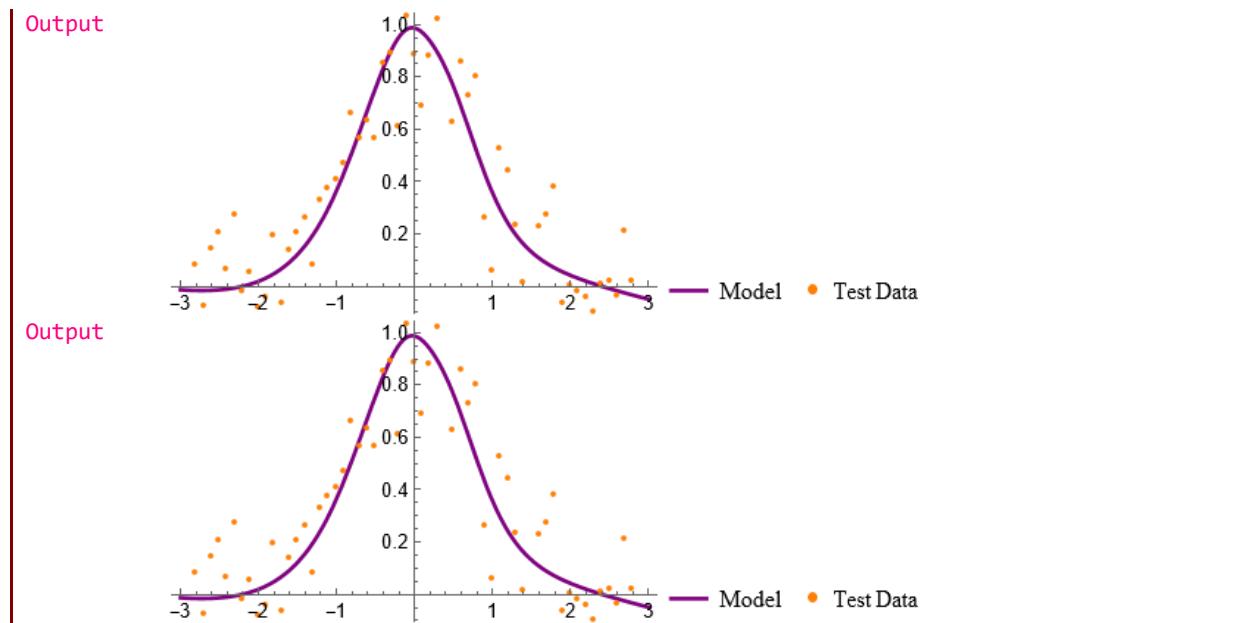
Output {0.001, 10, "ADAM", 0.0172739, 0.878063,

```



Output {0.001, 10, "ADAM", 0.0172739, 0.878063,



**Mathematica Code 7.33**

```

Input (* The code encapsulates a complete workflow for developing and optimizing a neural
       network to approximate a Gaussian-modulated exponential function with added noise,
       representing typical real-world data scenarios. It starts by generating synthetic
       training, validation, and test datasets, then defines a neural network with two
       Tanh-activated hidden layers. The core of the code involves optimizing the learning
       rate using Bayesian minimization to minimize the mean squared error on the
       validation set, thereby tuning the model for better generalization. Post
       optimization, the model is trained with the best-found learning rate and evaluated
       against the test data. Finally, the results are visualized to assess the model's
       predictive accuracy and fit, providing a comprehensive view of the model's
       performance and the effectiveness of the chosen hyperparameter: *)

(* Define a function based on a Gaussian-modulated exponential decay: *)
dataFunction[x_]:=Exp[-x^2];

(* Set the standard deviation of the Gaussian noise to be added to the data,
   simulating measurement errors or intrinsic variability: *)
noiseLevel=0.15;

(* Generate training data by evaluating the theoretical function at finely spaced
   intervals over a defined range, adding random Gaussian noise to each point: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Generate validation data similar to the training data but with less frequent
   sampling (step of 0.1): *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Generate test data using the same approach as validation data: *)
testData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],

```

```
{x,-3,3,0.1}
];

(* Define the objective function that will be minimized during the hyperparameter
optimization process. This function constructs a neural network, trains it with a
specific learning rate, and evaluates its performance using the mean squared error
on the validation set: *)
objectiveFunction[learningRate_]:=Module[
  {net,trainedNet,validationMSE},
  net=NetChain[
    {
      LinearLayer[50],Tanh,
      LinearLayer[30],Tanh,
      LinearLayer[1]
    }
  ];

  trainedNet=NetTrain[
    net,
    trainingData,
    All,
    ValidationSet->validationData,
    TrainingProgressMeasurements->{"MeanSquare"},
    LossFunction->MeanSquaredLossLayer[],
    BatchSize->64,
    (* Set the learning rate as specified by the optimization function: *)
    LearningRate->learningRate,
    Method->"ADAM",
    MaxTrainingRounds->5
  ];
  (* Extract the mean squared error from validation measurements: *)
  validationMSE=Values[trainedNet["ValidationMeasurements"]][[2]];
  validationMSE
];

(* Use Bayesian optimization to intelligently explore the space of possible learning
rates and find the one that minimizes validation MSE: *)
optimizationResult=BayesianMinimization[
  objectiveFunction,
  Interval[{0.001,0.1}]
];

(* Extract the optimal learning rate found by the optimization process and the
corresponding lowest MSE achieved: *)
bestParameters=optimizationResult["MinimumConfiguration"]
bestMSE=optimizationResult["MinimumValue"]

(* Retrieve the complete history of learning rates and MSEs evaluated during the
optimization, which can be useful for analysis: *)
evaluationHistory=optimizationResult["EvaluationHistory"]

(* Train the final model using the best learning rate determined through
optimization. This model will be evaluated against the test data: *)
finalModel=NetTrain[
  NetChain[
    {
      LinearLayer[50],Tanh,
      LinearLayer[30],Tanh,
      LinearLayer[1]
    }
  ]
```

```

    ],
    trainingData,
    All,
    LossFunction->MeanSquaredLossLayer[],
    BatchSize->64,
    LearningRate->bestParameters,
    Method->"ADAM",
    MaxTrainingRounds->5
];

(* Extract the final trained neural network for use in plotting predictions: *)
finalModeltoplot=finalModel["TrainedNet"];

(* Plot the predictions from the final trained model across the range of input data
to visualize how well it has learned to approximate the underlying function: *)
plotModel=Plot[
    finalModeltoplot[x],
    {x,-3,3},
    PlotStyle->{Purple,Thick},
    PlotLegends->{"Model"}
];

(* Create a scatter plot of the actual test data points to compare against the
model's predictions. This helps visually assess the accuracy and fit of the model.*)
plottestData=ListPlot[
    List@@@ testData,
    PlotStyle->Orange,
    PlotLegends->{"Test Data"}
];

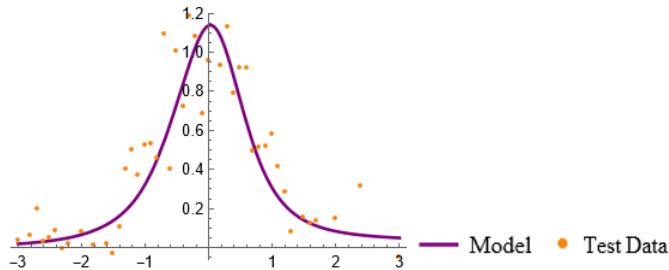
(* Combine the plot of the model's predictions with the scatter plot of the test
data to provide a comprehensive visualization of model performance: *)
Show[plotModel,plottestData,ImageSize->250]

```

Output
Output
Output

Configuration	Value
0.0526745	0.0392535
0.00953612	0.0244517
0.0114483	0.0240527
0.0983352	0.0439889
0.0357097	0.0261189
0.0819675	0.0248096
0.024149	0.0310045
0.0704681	0.0315944
0.0441047	0.0298763
0.00206204	0.0406232
0.0783361	0.0319067
0.0885859	0.0345115

Output

**Mathematica Code 7.34**

```

Input (* The enhanced code orchestrates a comprehensive process for developing a neural
       network tailored to model a Gaussian-modulated exponential function, incorporating
       noise to simulate real-world data. It features dynamic architecture configuration
       and employs Bayesian optimization to systematically explore and refine a range of
       hyperparameters - including learning rates, neuron counts, and optimizer types -
       to minimize validation mean squared error (MSE). The optimization is conducted in
       two stages to efficiently pinpoint optimal settings, which are then used to train
       the final model. This model is rigorously evaluated against test data and visually
       assessed through plots that compare its predictions with actual data, demonstrating
       its predictive accuracy and generalization capabilities: *)

(* Define the data generation function based on a Gaussian-modulated exponential
   function: *)
dataFunction[x_]:=Exp[-x^2];

(* Set the noise level for generating synthetic data: *)
noiseLevel=0.15;

(* Generate training data with fine granularity by adding Gaussian noise to simulate
   measurement errors: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Generate validation data: *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Generate test data: *)
testData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

(* Define the objective function for Bayesian optimization, which trains a neural
   network and computes the MSE on the validation set: *)
objectiveFunction[{learningRate_,neurons1_,neurons2_,method_}]:=Module[
  {net,trainedNet,validationMSE},
  net=NetChain[
    {
      LinearLayer[neurons1],Tanh,
      LinearLayer[neurons2],Tanh,
      LinearLayer[1]
    }
  ];
  trainedNet=net[validationData];
  validationMSE=Mean[(dataFunction[#[[1]]]-#[[2]])^2]&/@validationData;
  validationMSE
];

```

```
trainedNet=NetTrain[
    net,
    trainingData,
    All,
    ValidationSet->validationData,
    TrainingProgressMeasurements->{"MeanSquare"},
    LossFunction->MeanSquaredLossLayer[],
    BatchSize->64,
    LearningRate->learningRate,
    Method->method,
    MaxTrainingRounds->10
];
(* Extract the MSE from validation measurements: *)
validationMSE=Values[trainedNet["ValidationMeasurements"]][[2]];
validationMSE
];

(* Define ranges and options for hyperparameters: learning rates, number of neurons
in each layer, and optimizer methods: *)
learningRates={0.00001,0.0001,0.001,0.01,0.1};
numberOfNeuronsLayer1={30,40};
numberOfNeuronsLayer2={5,10,30};
optimizers={"ADAM","SGD","RMSProp"};

(* Randomly generate 50 configurations of hyperparameters from predefined choices
to initialize the Bayesian optimization: *)
conf=Table[
{
    (* Randomly select a learning rate: *)
    RandomChoice[learningRates],
    (* Randomly select the number of neurons for the first layer: *)
    RandomChoice[numberOfNeuronsLayer1],
    (* Randomly select the number of neurons for the second layer: *)
    RandomChoice[numberOfNeuronsLayer2],
    (* Randomly select an optimizer: *)
    RandomChoice[optimizers]
},
(* Generate 50 different configurations: *)
50]

(* Perform initial Bayesian Minimization over the hyperparameter space to find a
near-optimal configuration with a limited number of iterations: *)
optimizationResult1=BayesianMinimization[
    objectiveFunction,
    conf,
    Method->"MaxExpectedImprovement",
    MaxIterations->5
];

(* Store the initial evaluation history from the first round of optimization: *)
evaluationHistory1=optimizationResult1["EvaluationHistory"]

(* Continue the Bayesian optimization with the previously obtained evaluation
history to refine the search for optimal hyperparameters: *)
optimizationResult2=BayesianMinimization[
    objectiveFunction,
    conf,
    InitialEvaluationHistory->evaluationHistory1,
    Method->"MaxExpectedImprovement"
];
```

```

(* Store the final evaluation history and extract the best hyperparameters and
their corresponding MSE: *)
evaluationHistory2=optimizationResult2["EvaluationHistory"]
bestParameters=optimizationResult2["MinimumConfiguration"]
bestMSE=optimizationResult2["MinimumValue"]

(* Train the final model using the optimal parameters found from Bayesian
optimization for final evaluation: *)
finalModel=NetTrain[
  NetChain[
    {
      LinearLayer[bestParameters[[2]]],Tanh,
      LinearLayer[bestParameters[[3]]],Tanh,
      LinearLayer[1]
    },
    trainingData,
    All,
    LossFunction->MeanSquaredLossLayer[],
    BatchSize->64,
    LearningRate->bestParameters[[1]],
    Method->bestParameters[[4]],
    MaxTrainingRounds->10
  ];
]

(* Extract the final trained neural network for plotting its predictions: *)
finalModeltoplot=finalModel["TrainedNet"];

(* Visualization: *)

(* Plot the predictions from the final trained model across the range of input data
to visualize how well it has learned to approximate the underlying function: *)
plotModel=Plot[
  finalModeltoplot[x],
  {x,-3,3},
  PlotStyle->{Purple,Thick},
  PlotLegends->{"Model"}
];

(* Create a scatter plot of the actual test data points to compare against the
model's predictions. This helps visually assess the accuracy and fit of the model:
*)
plottestData=ListPlot[
  List@@@ testData,
  PlotStyle->Orange,
  PlotLegends->{"Test Data"}
];

(* Combine the plot of the model's predictions with the scatter plot of the test
data to provide a comprehensive visualization of model performance: *)
Show[plotModel,plottestData,ImageSize->250]

Output {{0.1,40,30,ADAM},{0.0001,30,30,SGD},{0.01,40,10,ADAM},{0.0001,30,5,ADAM},{0.001,30,5,SGD},{0.0001,40,10,SGD},{0.1,40,5,ADAM},{0.1,30,30,RMSProp},{0.0001,40,30,RMSProp},{0.1,40,10,RMSProp},{0.1,40,10,RMSProp},{0.0001,40,10,SGD},{0.001,40,30,RMSProp},{0.1,40,10,ADAM},{0.001,40,5,RMSProp},{0.0001,30,30,ADAM},{0.1,40,30,RMSProp},{0.001,30,30,ADAM},{0.1,40,5,ADAM},{0.0001,40,30,SGD},{0.1,30,5,SGD},{0.001,30,30,ADAM},{0.001,30,5,SGD},{0.0001,30,10,ADAM},{0.01,30,10,ADAM},{0.0001,30,10,SGD},{0.01,30,5,SGD},{0.0001,40,5,RMSProp}

```

```

op}, {0.001, 40, 30, ADAM}, {0.001, 30, 30, SGD}, {0.1, 40, 10, ADAM}, {0.00001, 30, 30, ADAM}, {0
.01, 40, 5, ADAM}, {0.00001, 30, 10, SGD}, {0.00001, 40, 5, SGD}, {0.001, 30, 10, RMSProp}, {0.00
01, 30, 10, SGD}, {0.01, 30, 5, RMSProp}, {0.0001, 30, 10, SGD}, {0.01, 30, 30, RMSProp}, {0.0000
1, 30, 30, SGD}, {0.0001, 40, 30, SGD}, {0.1, 30, 30, SGD}, {0.001, 40, 10, SGD}, {0.1, 40, 10, ADAM
}, {0.01, 40, 5, RMSProp}, {0.01, 40, 10, ADAM}, {0.0001, 40, 30, RMSProp}}

```

Output

Configuration				Value
0.1	40	5	ADAM	0.0293566
0.00001	30	30	SGD	0.251469
0.01	40	5	ADAM	0.0249366
0.01	40	5	ADAM	0.0249366
0.01	40	5	ADAM	0.0249366

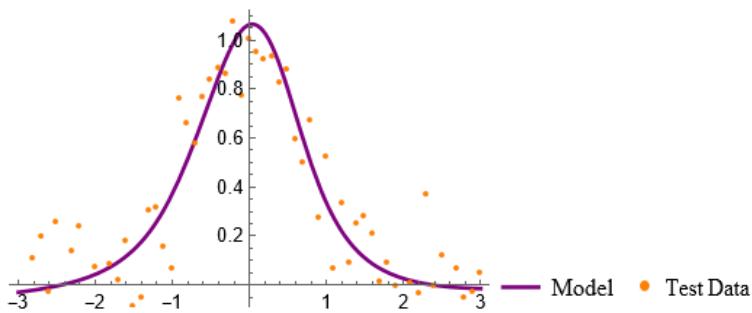
Output

Configuration				Value
0.1	40	5	ADAM	0.0293566
0.00001	30	30	SGD	0.251469
0.01	40	5	ADAM	0.0249366
0.01	40	5	ADAM	0.0249366
0.00001	40	5	SGD	0.688036
0.00001	40	5	SGD	0.688036
0.00001	40	5	SGD	0.688036
0.00001	30	5	ADAM	0.705337
0.00001	30	5	ADAM	0.705337
0.00001	30	5	ADAM	0.705337
0.00001	30	5	ADAM	0.705337
0.00001	30	5	ADAM	0.705337
0.00001	30	5	ADAM	0.705337
0.00001	40	5	SGD	0.688036

Output
Output

{0.01, 40, 5, ADAM}
0.0250422

Output



CHAPTER 8

REGULARIZATION TECHNIQUES

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 7, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation.

In machine learning, particularly in the field of deep learning, the complexity of models often leads to a significant challenge: overfitting. Overfitting occurs when a model learns the detail and noise in the training data to an extent that it negatively impacts the performance of the model on new data. The essence of regularization is to constrain our model in such a way that it avoids overfitting and thus performs better on unseen datasets. This chapter offers a comprehensive exploration of the strategies used to improve the generalization ability of neural networks (NNs). By introducing regularization techniques into the training process, we aim to develop models that are not only accurate on the training data but also robust and adaptable to new, unseen data. This chapter will cover three principal regularization techniques: penalty-based regularization, early stopping, and ensemble methods (dropout), each addressing overfitting in a unique way.

Penalty-Based Regularization:

Penalty-Based Regularization involves modifying the loss function by adding a penalty term to it. This term penalizes certain values of model parameters to discourage complexity:

- L_1 regularization (Lasso) promotes sparsity by adding a penalty equivalent to the absolute value of the coefficients, which can reduce the number of features in the model [127-138].
- L_2 regularization (Ridge) adds a penalty equivalent to the square of the magnitude of the coefficients, encouraging the model parameters to be small and distributing the error among them.
- Elastic net combines both L_1 and L_2 penalties and is useful when there are correlations among features.

Early Stopping:

Early Stopping is a different kind of regularization technique that does not modify the loss function but rather alters the training process itself. Early stopping involves monitoring the model's performance on a validation set and stopping the training as soon as the performance starts to degrade, despite improvements on the training set. This method not only helps in preventing overfitting but also saves computational resources by reducing unnecessary training iterations.

Ensemble Methods (Dropout):

Ensemble methods [139,140], with focus on dropout, is a powerful regularization technique particularly popular in training deep NNs. Unlike penalty-based methods, dropout works by randomly deactivating a subset of neurons in each training iteration. This randomness helps to break up situations where network layers co-adapt to correct mistakes from prior layers, thus making the model more capable of generalizing well.

By integrating these regularization techniques, machine learning practitioners can enhance model performance significantly. This chapter will equip you with the knowledge to understand these methods conceptually, ensuring that your NNs are not just powerful, but also robust and efficient.

This chapter delves into three essential regularization techniques implemented in Mathematica: `L2Regularization`, `TrainingStoppingCriterion`, and `DropoutLayer`. Each method addresses overfitting from a unique perspective, offering a robust toolkit for building and fine-tuning machine learning models.

In Mathematica, L_2 -regularization can be applied by specifying the `L2Regularization` suboption within the `Method` option of the `NetTrain` function. This integration allows users to seamlessly incorporate L_2 regularization into the training process, enhancing the model's ability to generalize.

An effective way to prevent overfitting is to stop the training process at the right moment before the model begins to overfit the training data. The `TrainingStoppingCriterion` option in `NetTrain` provides a systematic approach to determine when to halt training. This criterion can be based on various metrics, such as validation loss or accuracy, ensuring that the training process concludes when the model reaches optimal performance on validation data. Implementing a training stopping criterion helps in maintaining the model's generalization capability by avoiding excessive training.

In Mathematica, the `DropoutLayer` can be integrated into NN architectures, providing an easy-to-use mechanism for applying dropout regularization.

Unit 8.1

L_2 Regularization

In Mathematica, during the training of neural networks with the `NetTrain` function, you can apply ℓ_2 regularization by specifying `L2Regularization` as a suboption of the `Method` option. This regularization helps prevent overfitting by penalizing larger weights in the model's loss function.

The suboption "`L2Regularization`" can be given in the following forms:

<code>r</code>	use the value r for all weights in the net
<code>{lspec1->r1, lspec2->r2, ...}</code>	use the value ri for the specific part lspeci of the net

Mathematica Code 8.1

```

Input      (* The code aims to demonstrate the process of training neural networks on synthetic
           data, evaluating overfitting, and applying regularization to improve model
           performance. It generates synthetic data with Gaussian noise, visualizes this data,
           and constructs a neural network with two hidden layers. The network is trained on
           the data, and its predictions are compared to the original data to assess
           overfitting. A test dataset is generated to evaluate the network's performance
           using a mean squared loss function. The neural network is then retrained with L2
           regularization to prevent overfitting, and the performance of the regularized model
           is compared to the overfitted model to highlight the benefits of regularization:
           *)
(* Generate synthetic data: *)
syntheticData=Table[
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.2}
];

(* Plot the synthetic data in red color: *)
dataPlot=ListPlot[
  List@@@syntheticData,
  PlotStyle->Red,
  ImageSize->250,
  PlotLegends->{"Synthetic Data"},
  AxesLabel->{"x", "y"},
  PlotLabel->"Synthetic Data with Gaussian Noise"
];

(* Define a neural network with two hidden layers of 150 units each and Tanh
activation function: *)
neuralNetwork=NetChain[
  {
    150,Tanh,
    150,Tanh,
    1
  }
];

(* Train the neural network on the synthetic data for a maximum of 5000 rounds:
*)
trainingResults=NetTrain[
  neuralNetwork,
  syntheticData,
  All,
  MaxTrainingRounds->5000
]

```

```
(* Display the training loss over the training rounds: *)
Print["The average loss of most recent round is: ", trainingResults["RoundLoss"]];
];

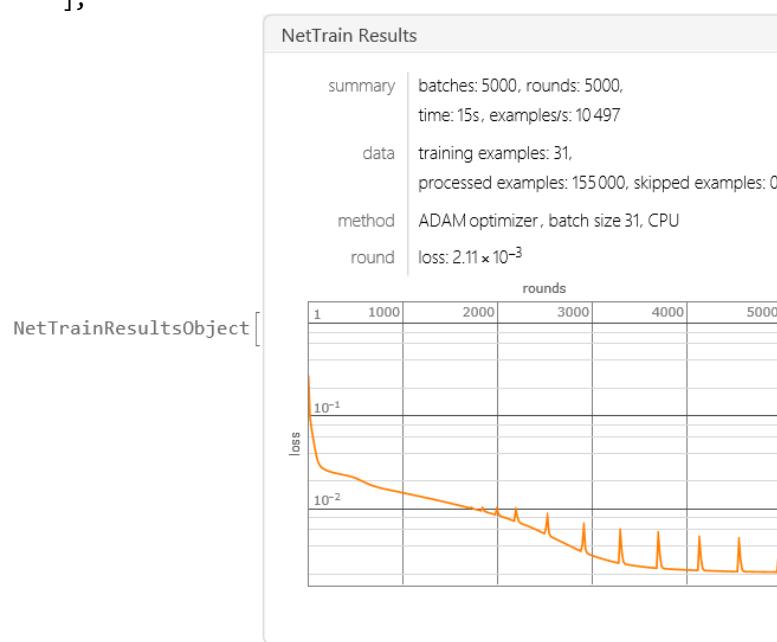
(* Retrieve the trained network that might be overfitting the data: *)
overfittedNetwork=trainingResults["TrainedNet"];

(* Plot the prediction of the overfitted network against the original data: *)
overfitPlot=Show[
  Plot[
    overfittedNetwork[x],
    {x,-3,3},
    PlotLegends->{"Overfitted Model"}
  ],
  dataPlot,
  PlotLabel->"Overfitted Model vs. Synthetic Data",
  ImageSize->250
]

(* Generate test data with the same distribution as the training data: *)
testData=Table[
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.2}
];
testXValues=Keys[testData];
testYValues=Values[testData];

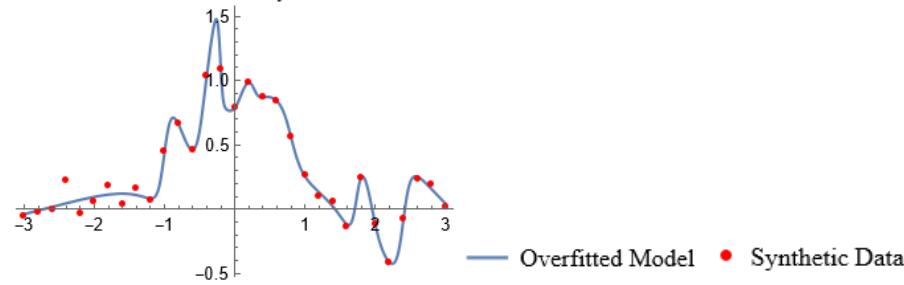
(* Define a function to calculate the mean squared loss on the test data: *)
calculateMeanTestLoss[network_]:=SquaredEuclideanDistance[network[testXValues],te
stYValues]/Length[testXValues];
(* Train the neural network with L2 regularization to prevent overfitting: *)
regularizedTrainingResults=NetTrain[
  neuralNetwork,
  syntheticData,
  All,
  Method->{"SGD","L2Regularization"->0.01},
  MaxTrainingRounds->5000
]
(* Retrieve the trained network with L2 regularization: *)
regularizedNetwork=regularizedTrainingResults["TrainedNet"];
(* Plot the prediction of the L2 regularized network against the original data: *)
regularizedPlot=Show[
  Plot[
    regularizedNetwork[x],
    {x,-3,3},
    PlotLegends->{"Regularized Model"}
  ],
  dataPlot,
  PlotLabel->"Regularized Model vs. Synthetic Data",
  ImageSize->250
]
(* Calculate and display the mean test loss for the L2 regularized network: *)
Print[
  "The mean test loss for the L2 regularized network is: ",
  calculateMeanTestLoss[regularizedNetwork]
];
(* Calculate and display the mean test loss for the overfitted network: *)
Print[
  "The mean test loss for the overfitted network is: ",
  calculateMeanTestLoss[overfittedNetwork]
```

];
Output

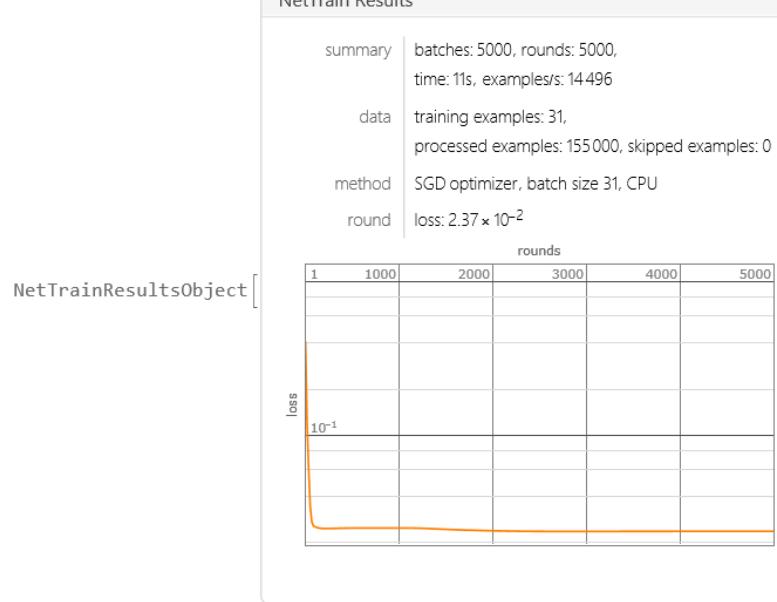


Output The average loss of most recent round is: 0.00211006

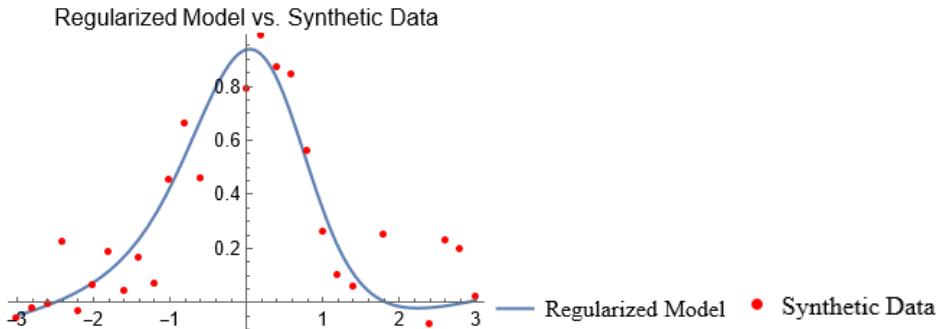
Output Overfitted Model vs. Synthetic Data



Output



Output



Output

```
The mean test loss for the L2 regularized network is: 0.0244639
```

Output

```
The mean test loss for the overfitted network is: 0.0343867
```

Mathematica Code 8.2

Input

```
(* The code demonstrates the impact of different L2 regularization rates on neural
network performance when fitting noisy synthetic data. It defines a function
g(x)=exp(\[Minus]x^2) and generates synthetic data by adding Gaussian noise to this
function. The code specifies three L2 regularization rates, constructs neural
networks with the same architecture for each rate, and trains these networks on
the synthetic data using stochastic gradient descent. The trained networks'
predictions are plotted alongside the original data to visually compare the effects
of the different regularization rates, highlighting how L2 regularization helps
mitigate overfitting: *)
```

```
(* Define the function g(x)=exp(-x^2): *)
g[x_]:=Exp[-x^2];

(* Generate synthetic data with added Gaussian noise: *)
syntheticData=Table[x->g[x]+RandomVariate[NormalDistribution[0,0.15]],{x,-3,3,0.1}];

(* Define different L2 regularization rates: *)
regularizationRates={0.1,0.01,0.000001};

(* Create a list of neural networks with the same architecture for each
regularization rate: *)
neuralNetworks=Table[
  NetChain[
    {
      (* First hidden layer with 100 units and Tanh activation: *)
      100,Tanh,
      (* Second hidden layer with 100 units and Tanh activation: *)
      100,Tanh,
      (*Output layer*)
      1
    }
  ],
 {rate,regularizationRates}
];

(* Train each neural network on the synthetic data with the corresponding L2
regularization rate: *)
trainedNetworks=Table[
  NetTrain[
    (*Neural network to be trained*)
    neuralNetworks[[i]],
    (*Training data*)
    syntheticData
  ],
 {i,Length[regularizationRates]}
];
```

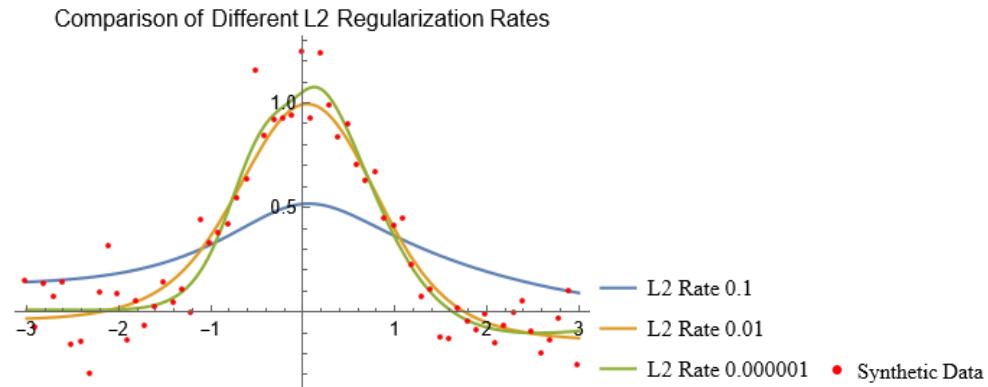
```

syntheticData,
(*Maximum number of training rounds*)
MaxTrainingRounds->5000,
(*Training method with L2 regularization*)
Method->{"SGD","L2Regularization"->regularizationRates[[i]]}
],
{i,1,Length[regularizationRates]}
];

(* Plot the predictions of the trained networks and the original data: *)
Show[
Plot[
(*Predictions of the trained networks*)
{trainedNetworks[[1]][x],trainedNetworks[[2]][x],trainedNetworks[[3]][x]},
{x,-3,3},
ImageSize->300,
PlotRange->All,
(*Legends for the different regularization rates*)
PlotLegends->{"L2 Rate 0.1","L2 Rate 0.01","L2 Rate 0.000001"},
PlotLabel->"Comparison of Different L2 Regularization Rates"
],
ListPlot[
(* Plot the original synthetic data: *)
List@@@syntheticData,
PlotStyle->{Red,PointSize[0.01]},
PlotLegends->{"Synthetic Data"}
]
]
]

```

Output



Unit 8.2

TrainingStoppingCriterion

TrainingStoppingCriterion

is an option for `NetTrain` that specifies a criterion for stopping training early in order to prevent overfitting.

Setting `TrainingStoppingCriterion -> "measurement"` specifies that training should be stopped if `measurement` stops improving. Possible values for measurement include:

<code>"Loss"</code>	network training loss
<code>key</code>	a <code>TrainingProgressMeasurements</code> key

Remarks:

- `TrainingStoppingCriterion -> Automatic` is equivalent to `TrainingStoppingCriterion -> "Loss"`.
- If the validation set is present, the stopping criterion is checked whenever the validation loss and metrics are calculated (once per round by default); otherwise, the stopping criterion is checked once per round.
- `TrainingStoppingCriterion` has a number of suboptions that can be specified using the `<|"Criterion"->"measurement", opt1->val1, opt2->val2, ...|>` syntax.
- Setting `TrainingStoppingCriterion -> <|"Criterion"->"measurement", "Patience"->n|>` specifies that training should be stopped if an improvement in measurement is not seen for n rounds in a row. The default value for n is 0.
- Setting `TrainingStoppingCriterion -> <|"Criterion"->"measurement", "InitialPatience"->n |>` specifies that the stopping criterion is only checked for the first time after n rounds. The default value for n is 0.
- Setting `TrainingStoppingCriterion -> <|"Criterion"->"measurement", "Improvement"->v|>` specifies the minimum change in measurement that is considered an improvement. Possible values for improvement are:

<code>"AbsoluteChange"</code>	stop training if measurement improves by less than v
<code>"RelativeChange"</code>	stop training if measurement improves by less than a factor v of the current best value

- If an improvement specification is not given, then `"AbsoluteChange"->0` is used.
- It is also possible to specify a function as the stopping criterion using `TrainingStoppingCriterion -> <|"Criterion"->func,...|>`, where `func` should return True if training should be stopped.

Mathematica Code 8.3

```
Input      (* The code trains a neural network on synthetic data generated from a Gaussian-like function with noise, using various training strategies and early stopping criteria to explore their effects on model performance and overfitting. The code first trains a network without stopping criteria to potentially overfit the data, then applies different forms of early stopping based on validation loss improvements (absolute, relative, and after a set number of rounds) to prevent overfitting and enhance generalization. Each model's performance is visualized against the training data, providing a clear comparison of how each stopping criterion influences the learning process and the model's ability to generalize beyond the training set: *)

(* Generating synthetic training and validation data: *)
trainingData=Table[
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.2}
];
validationData=Table[
```

```
x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],  
{x,-3,3,0.1}  
];  
  
(* Defining and training the neural network: *)  
initialNetwork=NetChain[  
  {  
    150,Tanh,  
    150,Tanh,  
    1  
  }  
];  
initialTrainingResults=NetTrain[  
  initialNetwork,  
  trainingData,  
  All,  
  MaxTrainingRounds->10000  
]  
  
(* Extracting the trained network for overfitting demonstration: *)  
overfittedNetwork=initialTrainingResults["TrainedNet"];  
  
(* Visualizing the performance of the overfitted network: *)  
Show[  
  Plot[  
    overfittedNetwork[x],  
    {x,-3,3},  
    ImageSize->200  
  ],  
  ListPlot[  
    List@@@trainingData,  
    PlotStyle->Red  
  ]  
]  
  
(* Re-training with a loss plot and validation data: *)  
basicTrainingResults=NetTrain[  
  initialNetwork,  
  trainingData,  
  "LossPlot",  
  ValidationSet->validationData,  
  MaxTrainingRounds->2000  
]  
  
(* Implementing early stopping based on validation loss improvement: *)  
earlyStoppingResults=NetTrain[  
  initialNetwork,  
  trainingData,  
  All,  
  ValidationSet->validationData,  
  MaxTrainingRounds->2000,  
  TrainingStoppingCriterion-><|"Criterion"->"Loss","Patience"->10|>  
];  
  
(* Extracting the early stopped network and visualizing its performance: *)  
earlyStoppedNetwork=earlyStoppingResults["TrainedNet"];  
earlyStoppingResults["LossPlot"]  
  
Show[  
  Plot[
```

```

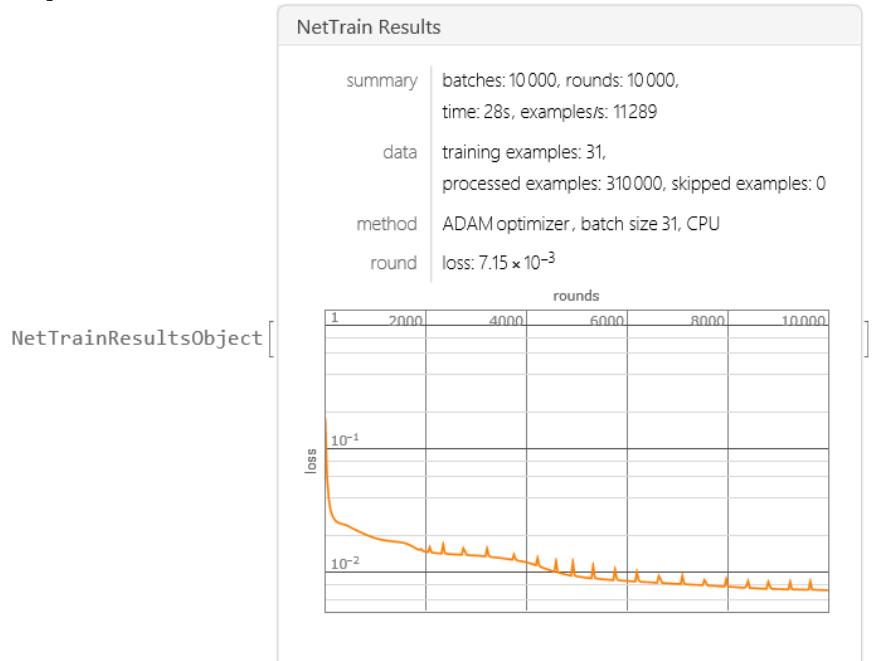
earlyStoppedNetwork[x],
{x,-3,3},
ImageSize->250
],
ListPlot[
List@@@trainingData,
PlotStyle->Purple
]
]

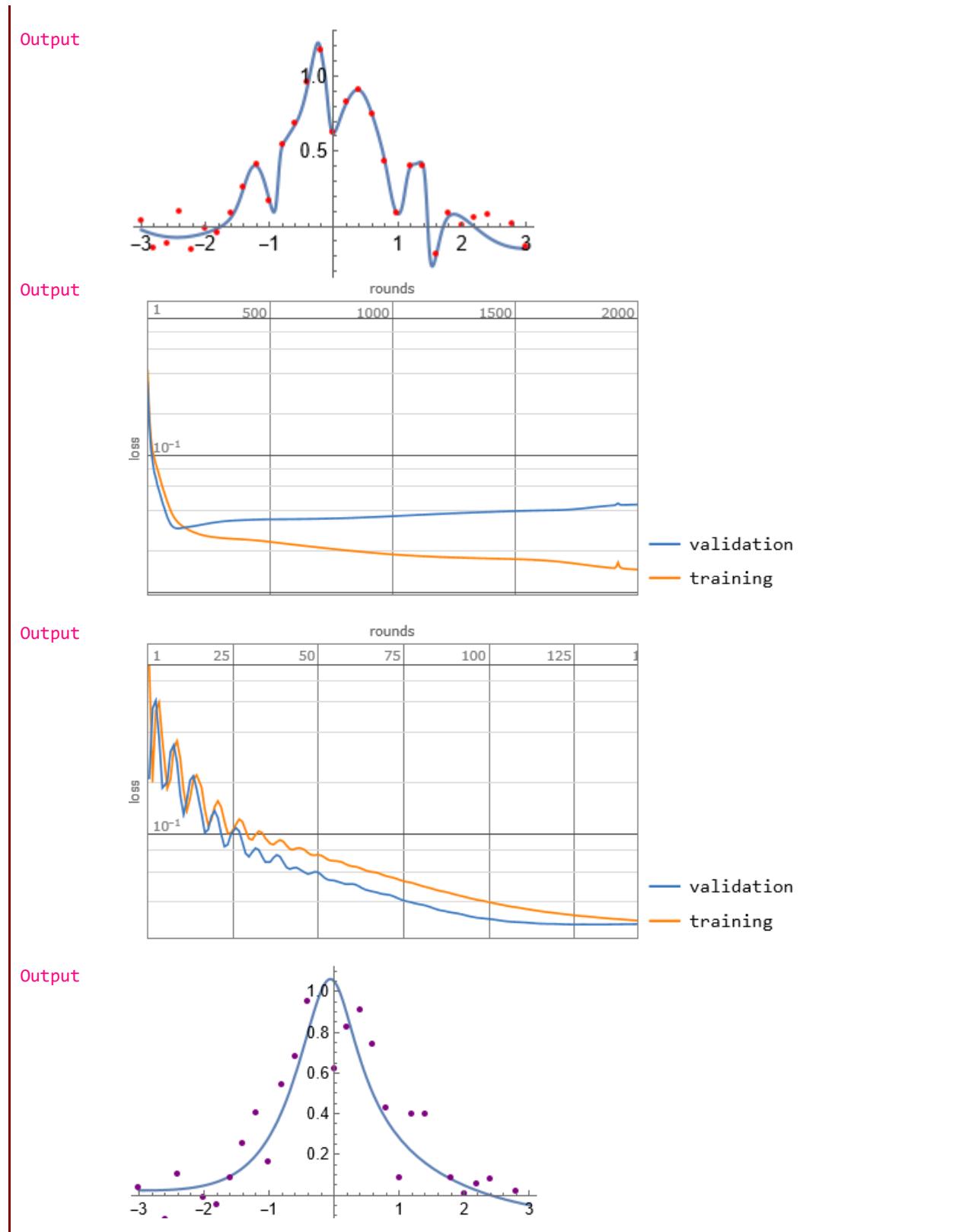
(* Additional training with a delayed stopping criterion: *)
delayedStoppingResults=NetTrain[
initialNetwork,
trainingData,
"LossPlot",
ValidationSet->validationData,
TrainingStoppingCriterion-><|"Criterion"->"Loss","InitialPatience"->300|>
]
(* Implementing fine-grained stopping criteria based on absolute and relative
changes: *)
fineGrainedStoppingResultsAbsolute=NetTrain[
initialNetwork,
trainingData,
"LossPlot",
ValidationSet->validationData,
TrainingStoppingCriterion-><|"Criterion"->"Loss","AbsoluteChange"->0.0001,"Patience"->10|>
]

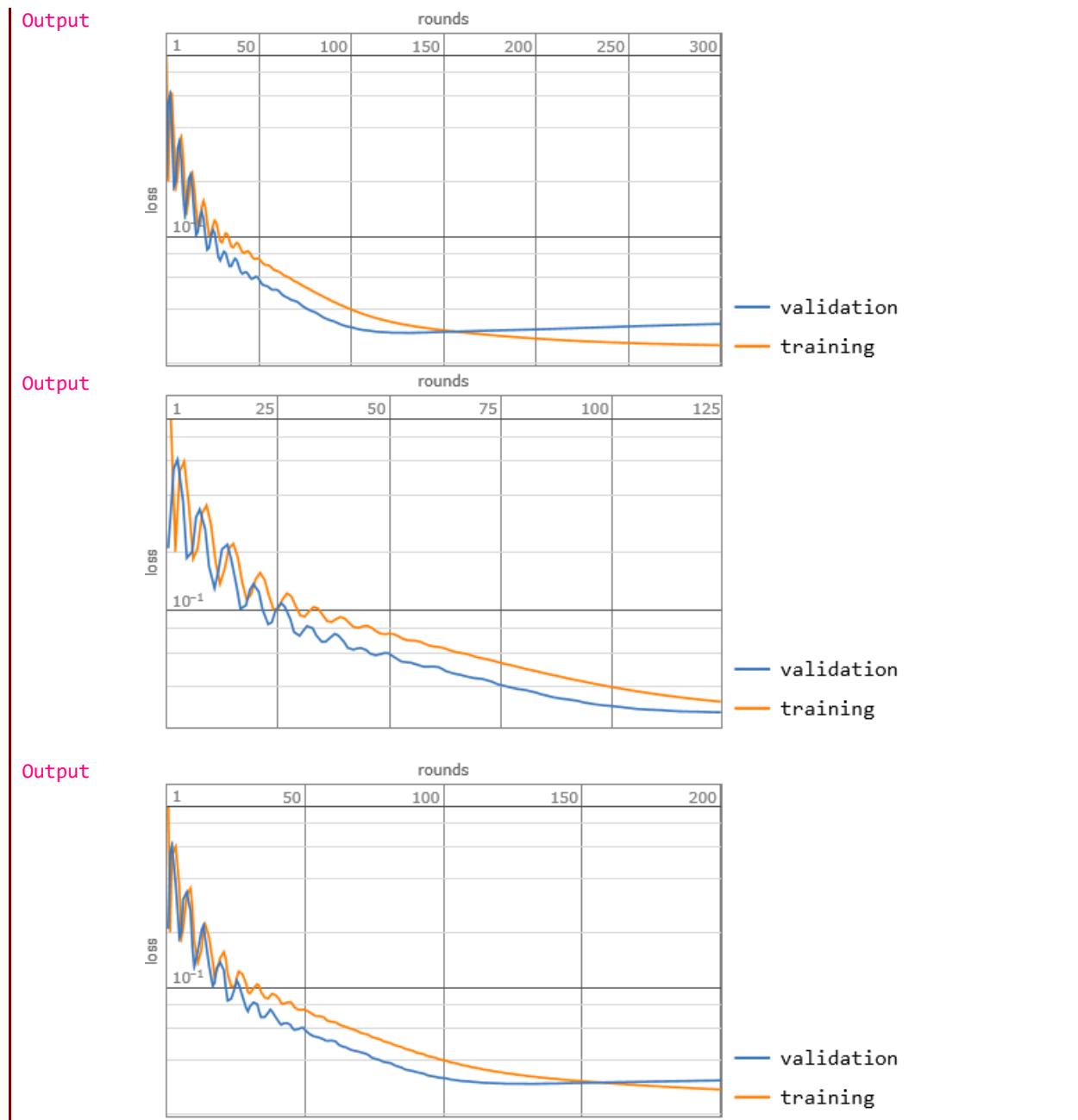
fineGrainedStoppingResultsRelative=NetTrain[
initialNetwork,
trainingData,
"LossPlot",
ValidationSet->validationData,
TrainingStoppingCriterion-><|"Criterion"->"Loss","RelativeChange"->0.01,"InitialPatience"->200|>
]
]

```

Output





**Mathematica Code 8.4**

Input (* The code is designed to create, train, and evaluate a neural network for binary classification. It begins by generating synthetic training and validation datasets consisting of points within a unit disk, labeled based on their proximity to the origin. The network, composed of two linear layers with Tanh and Logistic Sigmoid activations, is trained using metrics like recall and accuracy, and employs an early stopping criterion based on recall to prevent overfitting. Post-training, the code visualizes the decision boundary, displays training performance plots, and extracts the trained network for further analysis, effectively demonstrating an end-to-end machine learning pipeline within a binary classification context: *)

```
(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];
```

```
(* Assign labels based on whether the point's distance from the origin is less
than 0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Generate 500 random points within a unit disk for validation: *)
validationPoints=RandomPoint[Disk[],500];

(* Assign labels using the same criteria as for the training set: *)
validationLabels=Thread[Map[Norm,validationPoints]<0.5];

(* Combine the points and labels into validation data: *)
validationData=validationPoints->validationLabels;

(* Visualize the synthetic training set with different colors for each class: *)
ListPlot[
{
  Pick[trainpoints,trainlabels,True],
  Pick[trainpoints,trainlabels,False]
},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},
PlotLegends->{"Class 1","Class 2"}
]

(* Define a neural network with two linear layers and activation functions: *)
net=NetChain[
{
  LinearLayer[20],ElementwiseLayer[Tanh],
  LinearLayer[],ElementwiseLayer[LogisticSigmoid]},
 "Output"->NetDecoder["Boolean"]
];

(* Train the neural network on the training data with specified metrics and
validation: *)
netTrainResults=NetTrain[
  net,
  trainingData,
  All,
  ValidationSet->validationData,
  TrainingProgressMeasurements->{"Recall","Accuracy","ConfusionMatrixPlot"},
  (* Use TrainingStoppingCriterion to stop training when the validation recall
has stopped increasing for 20 iterations after Initial Patience 150 rounds: *)
  TrainingStoppingCriterion->{|"Criterion"->"Recall","InitialPatience"-
>150,"Patience"->20|}
];

(* Association of training measurements on the ValidationSet after the most
recent validation measurement: *)
netTrainResults["ValidationMeasurements"]

(* Extract the trained network for further use or analysis: *)
trainedNet=netTrainResults["TrainedNet"];

(* Visualize the decision boundary of the trained network: *)
```

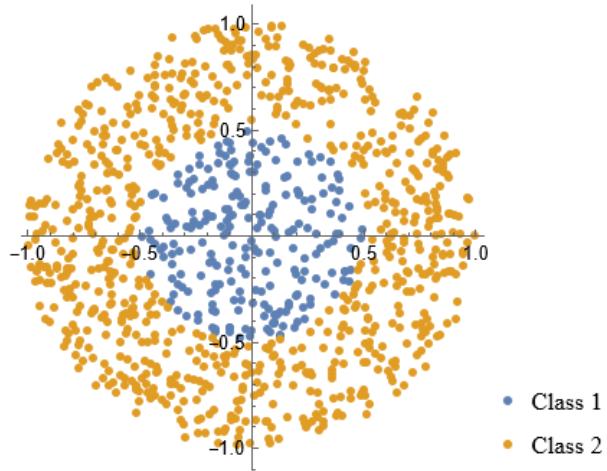
```

ContourPlot[
  trainedNet[{x,y},None],
  {x,-1,1},
  {y,-1,1},
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  PlotLegends->Automatic,
  LabelStyle->Directive[Black,10],
  ImageSize->250,
  PlotLabel->"Trained Nonlinear Classifier Decision Boundary"]

(* Display additional training plots like loss, Recall, Accuracy, and confusion
matrix: *)
lossPlot=netTrainResults["FinalPlots"][[1]]
recallPlot=netTrainResults["FinalPlots"][[2]]
accuracyPlot=netTrainResults["FinalPlots"][[3]]
confusionmatrixPlot=netTrainResults["FinalPlots"][[4]]

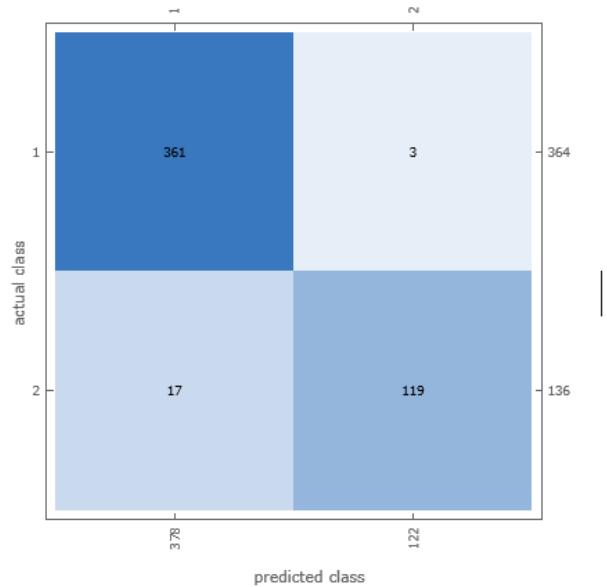
```

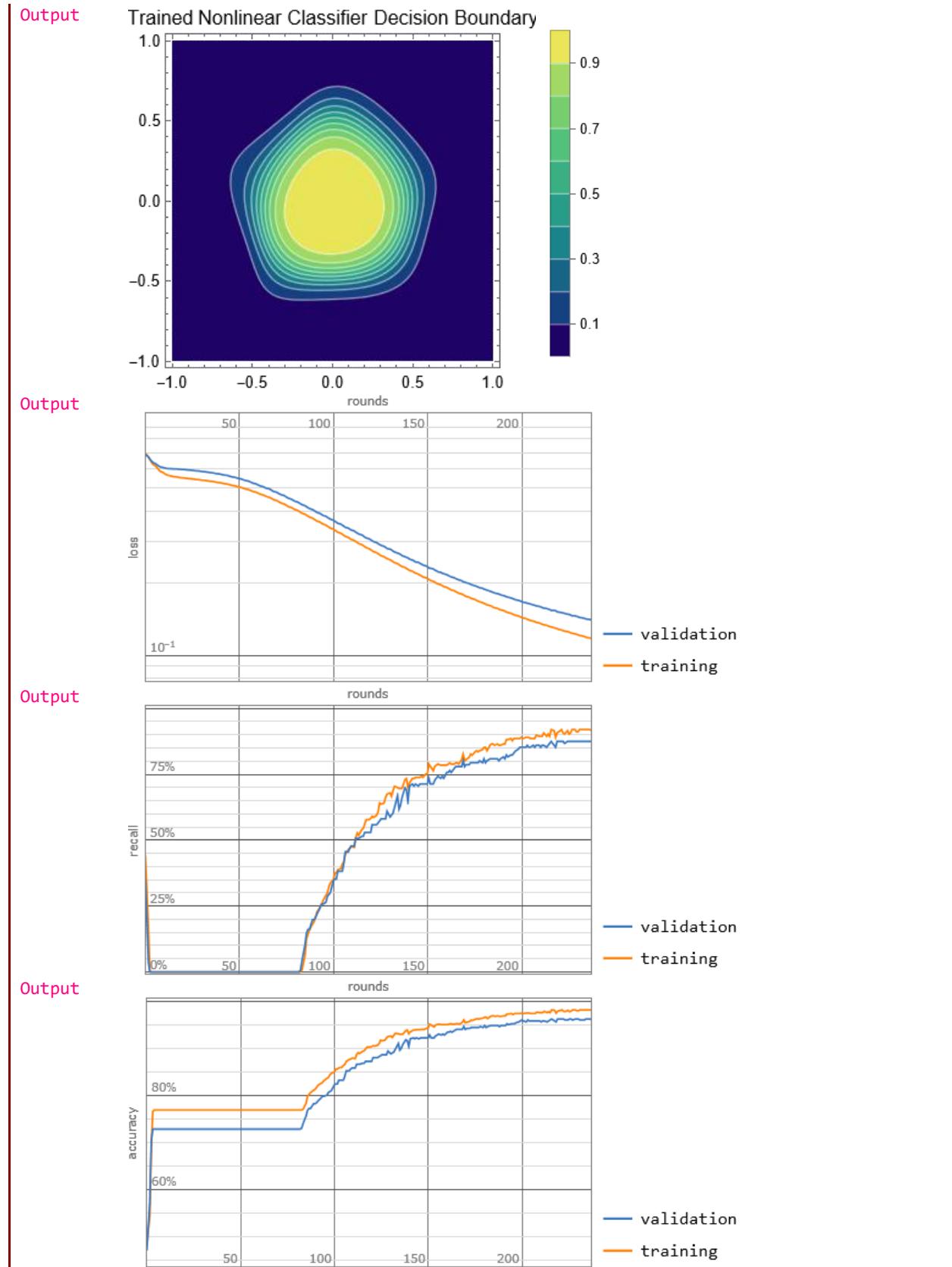
Output

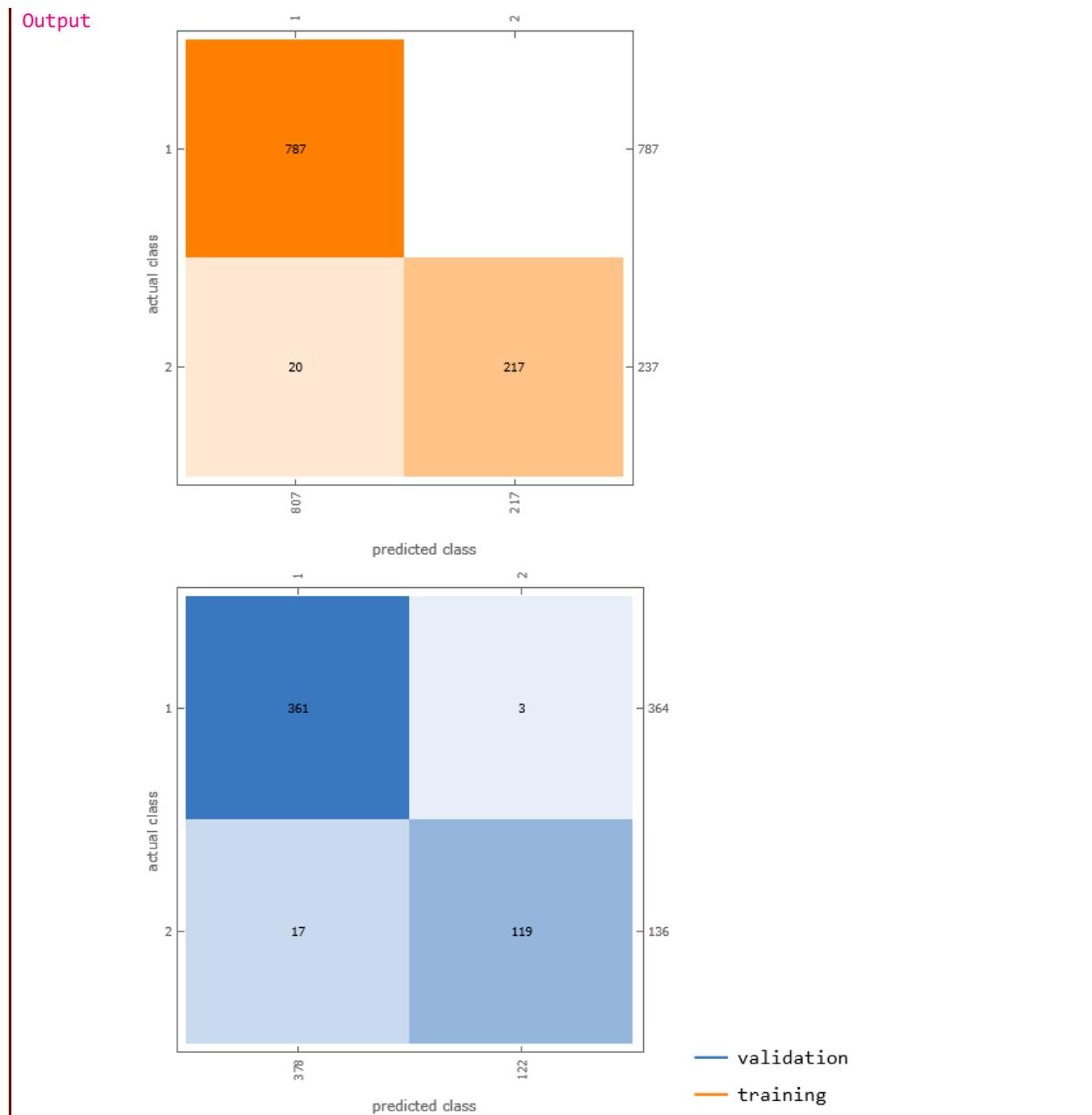


Output

<| Loss->0.140352,Recall->0.875,Accuracy->0.96,ConfusionMatrixPlot->





**Mathematica Code 8.5**

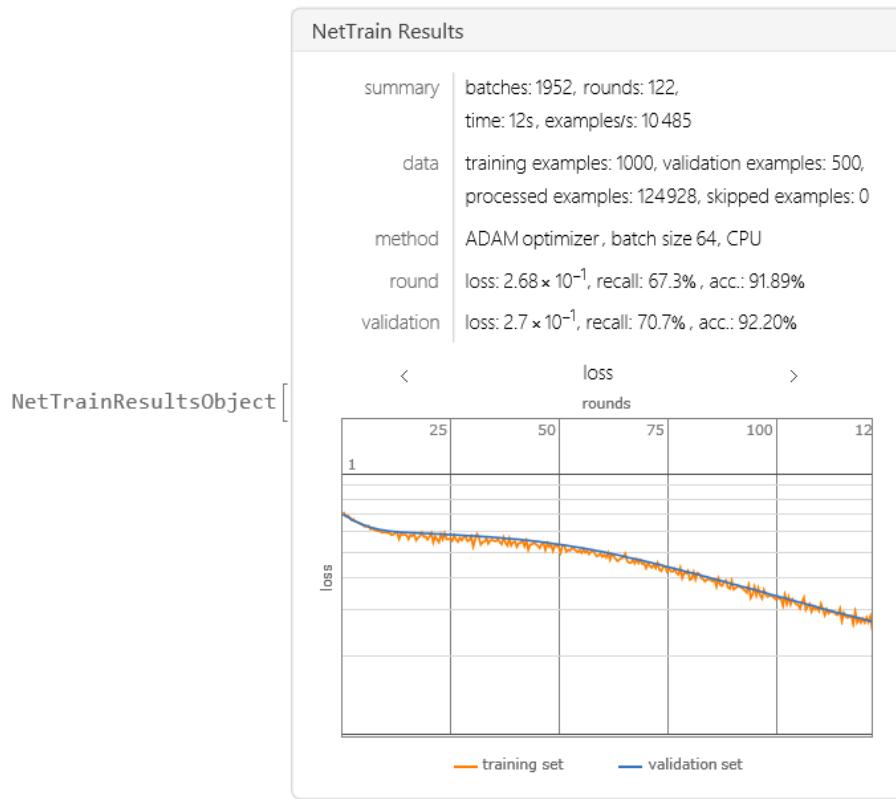
Input (* The previous code uses early stopping based on the recall metric, ceasing training if recall fails to improve within defined patience parameters. In contrast, the following code stops training based on validation loss, halting the process if the loss remains below a threshold of 0.3 for more than 10 consecutive validation checks. These differences highlight alternative approaches to preventing overfitting, with the first method prioritizing a specific performance metric (recall), and the second emphasizing overall error minimization: *)

```
(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];
```

```
(* Assign labels based on whether the point's distance from the origin is less
than 0.5: *)
```

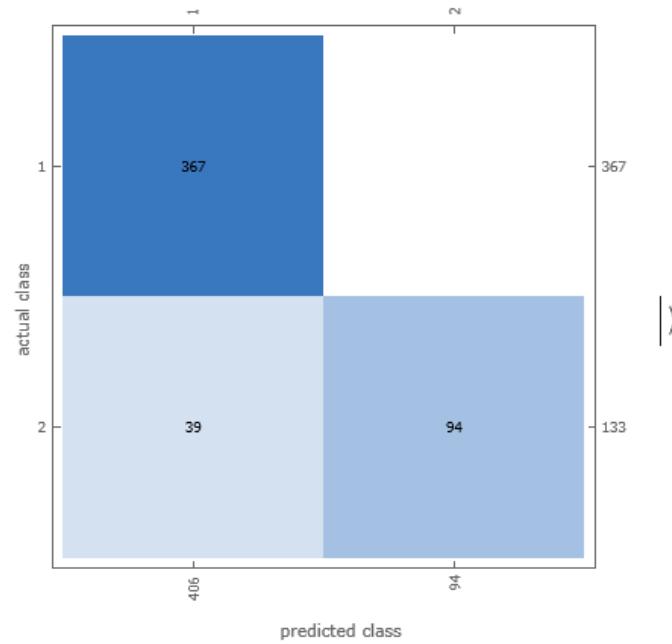
```
trainlabels=Thread[Map[Norm,trainpoints]<0.5];  
  
(* Combine the points and labels into training data: *)  
trainingData=trainpoints->trainlabels;  
  
(* Generate 500 random points within a unit disk for validation: *)  
validationPoints=RandomPoint[Disk[],500];  
  
(* Assign labels using the same criteria as for the training set: *)  
validationLabels=Thread[Map[Norm,validationPoints]<0.5];  
  
(* Combine the points and labels into validation data: *)  
validationData=validationPoints->validationLabels;  
  
(* Define a neural network with two linear layers and activation functions: *)  
net=NetChain[  
    {  
        LinearLayer[20],ElementwiseLayer[Tanh],  
        LinearLayer[],ElementwiseLayer[LogisticSigmoid]},  
    "Output"->NetDecoder[Boolean]  
];  
  
(* Train the neural network on the training data with specified metrics and  
validation: *)  
netTrainResults=NetTrain[  
    net,  
    trainingData,  
    All,  
    ValidationSet->validationData,  
    TrainingProgressMeasurements->{"Recall","Accuracy","ConfusionMatrixPlot"},  
    (* Stop training if the validation loss is less than 0.3 for more than 10 rounds  
in a row: *)  
    TrainingStoppingCriterion-><|"Criterion"->  
    >Function[#ValidationLoss<0.3],"Patience"->10|>  
]  
  
(* Association of training measurements on the ValidationSet after the most recent  
validation measurement: *)  
netTrainResults["ValidationMeasurements"]  
  
(* Extract the trained network for further use or analysis: *)  
trainedNet=netTrainResults["TrainedNet"];  
  
(* Visualize the decision boundary of the trained network: *)  
ContourPlot[  
    trainedNet[{x,y},None],  
    {x,-1,1},  
    {y,-1,1},  
    ContourStyle->{White},  
    ClippingStyle->Automatic,  
    ColorFunction->"BlueGreenYellow",  
    PlotLegends->Automatic,  
    LabelStyle->Directive[Black,10],  
    ImageSize->250,  
    PlotLabel->"Trained Nonlinear Classifier Decision Boundary"  
]
```

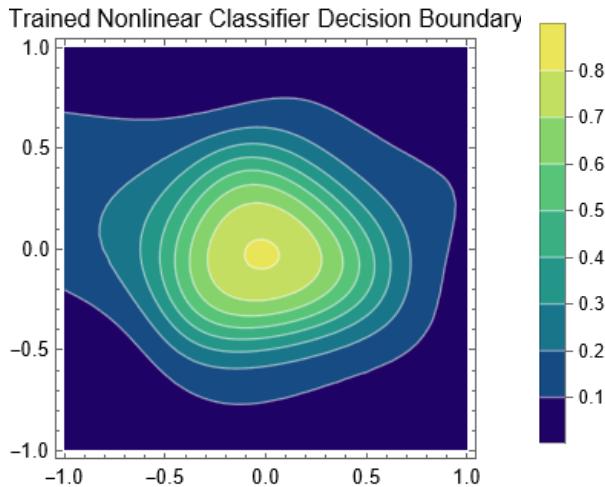
Output



Output

<| Loss->0.269947, Recall->0.706767, Accuracy->0.922, ConfusionMatrixPlot->



Output**Mathematica Code 8.6**

```

Input (* The direction of change that is considered an improvement depends on the
measurement. For "Loss", a decrease is an improvement. For other measurements, the
direction can be specified using the "Direction"→direction suboption of
TrainingProgressMeasurements. For built-in measurements, the direction is
automatically chosen as appropriate: *)

(* Stop training if the L1 norm of the first layer activations does not increase:
*)

(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less than
0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Generate 500 random points within a unit disk for validation: *)
validationPoints=RandomPoint[Disk[],500];

(* Assign labels using the same criteria as for the training set: *)
validationLabels=Thread[Map[Norm,validationPoints]<0.5];

(* Combine the points and labels into validation data: *)
validationData=validationPoints->validationLabels;

(* Define a neural network with two linear layers and activation functions: *)
net=NetChain[
  {
    LinearLayer[20],ElementwiseLayer[Tanh],
    LinearLayer[],ElementwiseLayer[LogisticSigmoid]],
  "Output"→NetDecoder["Boolean"]
];
(* Train the neural network on the training data with specified metrics and
validation: *)
netTrainResults=NetTrain[
  net,

```

```

    trainingData,
    All,
    ValidationSet->validationData,
    (* Stop training if the L1 norm of the first layer activations does not
    increase: *)
    TrainingProgressMeasurements-><|
      "Measurement"->NetPort[{"1","Output"}],
      "Aggregation"->"L1Norm",
      "Direction"->"Increasing"
    |>,
    TrainingStoppingCriterion-><|"Criterion"->"1/Output","Patience"->9|>
  ]

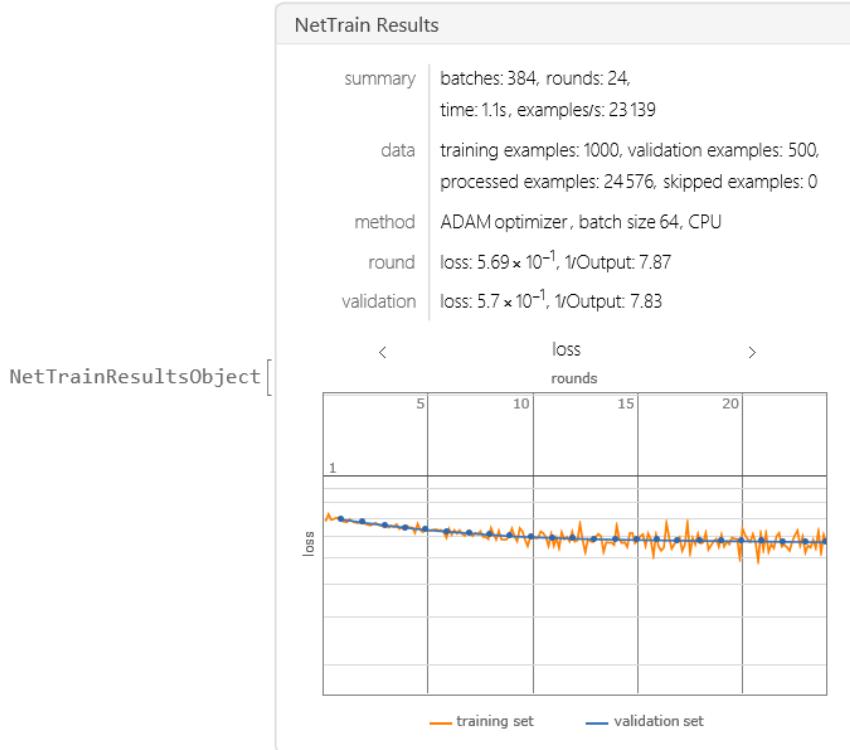
(* Association of training measurements on the ValidationSet after the most recent
validation measurement: *)
netTrainResults["ValidationMeasurements"]

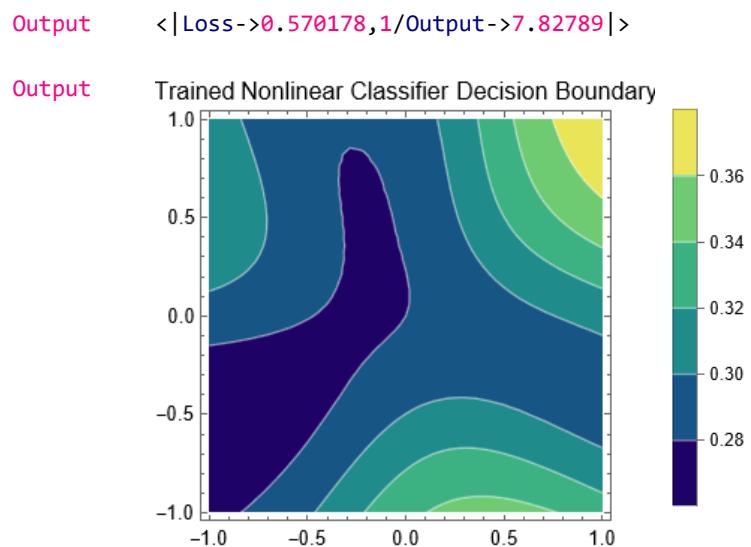
(* Extract the trained network for further use or analysis: *)
trainedNet=netTrainResults["TrainedNet"];

(* Visualize the decision boundary of the trained network: *)
ContourPlot[
  trainedNet[{x,y},None],
  {x,-1,1},
  {y,-1,1},
  ContourStyle->{White},
  ClippingStyle->Automatic,
  ColorFunction->"BlueGreenYellow",
  PlotLegends->Automatic,
  LabelStyle->Directive[Black,10],
  ImageSize->250,
  PlotLabel->"Trained Nonlinear Classifier Decision Boundary"
]

```

Output





Unit 8.3

DropoutLayer

DropoutLayer[]

represents a net layer that sets its input elements to zero with probability 0.5 during training.

DropoutLayer[p]

sets its input elements to zero with probability p during training.

Possible explicit settings for the **Method** option include:

"AlphaDropout"	keeps the mean and variance of the input constant; designed to be used together with the ElementwiseLayer["SELU"] activation
"Dropout"	sets the input elements to zero with probability p during training, multiplying the remainder by $1/(1-p)$

Remarks:

- **DropoutLayer** normally infers the dimensions of its input from its context in **NetChain** etc. To specify the dimensions explicitly as `{n1,n2,...}`, use `DropoutLayer["Input" -> {n1,n2,...}]`.
- `DropoutLayer[...][input]` explicitly computes the output from applying the layer.
- `DropoutLayer[...][{input1,input2,...}]` explicitly computes outputs for each of the `inputi`.
- `DropoutLayer` only randomly sets input elements to zero during training. During evaluation, `DropoutLayer` leaves the input unchanged, unless `NetEvaluationMode -> "Train"` is specified when applying the layer.

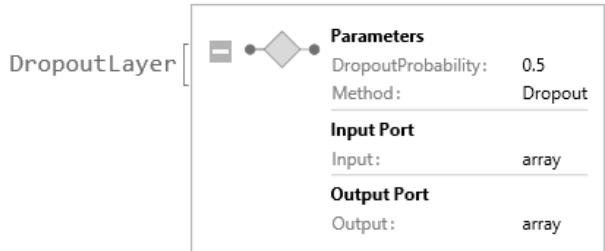
Mathematica Code 8.7

```
Input (*Define a DropoutLayer without specifying the dropout rate, which defaults to 0.5:
*)
drop=DropoutLayer[]

(* Apply the dropout layer to an input vector {1,2,3,4,5} in evaluation mode (which
remains unchanged): *)
drop[{1,2,3,4,5}]

(* Apply the dropout layer to the same input vector in training mode: *)
(* In training mode, the dropout layer randomly drops (sets to zero) some of the
input units: *)
drop[{1,2,3,4,5},NetEvaluationMode->"Train"]
```

Output



```
Output {1.,2.,3.,4.,5.}
Output {2.,0.,0.,8.,10.}
```

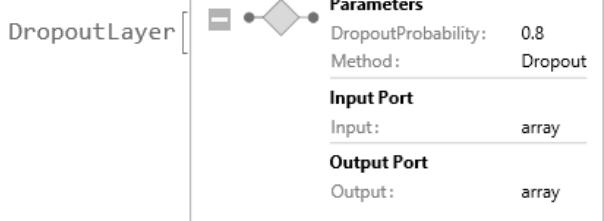
Mathematica Code 8.8

```
Input (* Define a DropoutLayer with a dropout rate of 0.8: *)
drop=DropoutLayer[0.8]
```

```
(* Apply the dropout layer to an input vector {1,2,3,4,5,6,7,8,9,10} in evaluation mode: *)
drop[{1,2,3,4,5,6,7,8,9,10}]
```

```
(* Apply the dropout layer to the same input vector in training mode: *)
drop[{1,2,3,4,5,6,7,8,9,10},NetEvaluationMode->"Train"]
```

Output



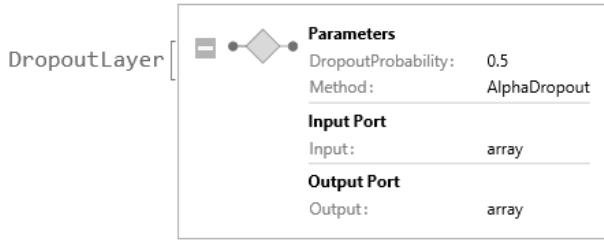
```
Output {1.,2.,3.,4.,5.,6.,7.,8.,9.,10.}
Output {5.,0.,0.,0.,0.,0.,0.,0.,0.,50.}
```

Mathematica Code 8.9

```
Input (* Create a DropoutLayer using "AlphaDropout" as the dropout method: *)
drop=DropoutLayer[Method->"AlphaDropout"]
```

```
(* Apply it to input data, specifying that training behavior be used: *)
drop[{1,2,3,4,5,6,7},NetEvaluationMode->"Train"]
```

Output



```
Output {1.6656,2.552,-0.779194,4.32481,5.21122,6.09762,-0.779194}
```

Mathematica Code 8.10

```
Input (* The code demonstrates the process of training neural networks on synthetic data, visualizing the effects of overfitting, and evaluating the effectiveness of dropout as a regularization technique. It generates synthetic data based on the function exp(-x^2) with added Gaussian noise and visualizes this data. A basic neural network is defined and trained on the synthetic data, and its overfitted predictions are plotted. New test data is generated for performance evaluation. Additionally, a dropout neural network is defined and trained to prevent overfitting. The predictions of both the overfitted and dropout networks are visualized, and their performance is compared by calculating and comparing the mean squared error on the test set, highlighting the benefits of dropout in improving generalization: *)
```

```
(* Generate synthetic data with added Gaussian noise: *)
syntheticData=Table[
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.1}
];
```

```
(* Plot the synthetic data: *)
dataPlot=ListPlot[
  List@@@syntheticData,
```

```
PlotStyle->Red,
PlotLabel->"Synthetic Data"
];

(* Define a neural network with two hidden layers of 150 units each and Tanh
activation functions: *)
basicNeuralNetwork=NetChain[
{
  150,Tanh,
  150,Tanh,
  1
}
];

(* Train the neural network on the synthetic data for a maximum of 10000 rounds: *)
trainingResultsBasic=NetTrain[
  basicNeuralNetwork,
  syntheticData,
  All,
  MaxTrainingRounds->10000
]

(* Retrieve the trained network that might be overfitting the data: *)
overfittedNetwork=trainingResultsBasic["TrainedNet"];

(* Plot the predictions of the overfitted network against the original data: *)
Show[
  Plot[
    overfittedNetwork[x],
    {x,-3,3},
    ImageSize->250,
    PlotLabel->"Overfitted Network Predictions"
  ],
  dataPlot
]

(* Generate test data with the same distribution as the training data: *)
testData=Table[
  x->Exp[-x^2]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.1}
];
testInputs=Keys[testData];
testOutputs=Values[testData];

(* Define a neural network with dropout to prevent overfitting: *)
dropoutNeuralNetwork=NetChain[
{
  150,DropoutLayer[0.8],Tanh,
  150,Tanh,
  1
}
];

(* Train the dropout neural network on the synthetic data for a maximum of 10000
rounds: *)
trainingResultsDropout=NetTrain[
  dropoutNeuralNetwork,
  syntheticData,
  All,
```

```

MaxTrainingRounds->10000
]

(* Retrieve the trained dropout network: *)
trainedDropoutNetwork=trainingResultsDropout["TrainedNet"];

(* Plot the predictions of the trained dropout network against the original data: *)
Show[
Plot[
trainedDropoutNetwork[x],
{x,-3,3},
PlotLabel->"Dropout Network Predictions",
ImageSize->250
],
dataPlot
]

(* Create a function to measure the mean squared error on the test set: *)
meanTestLoss[network_]:=SquaredEuclideanDistance[network[testInputs],testOutputs]/Length[testInputs];

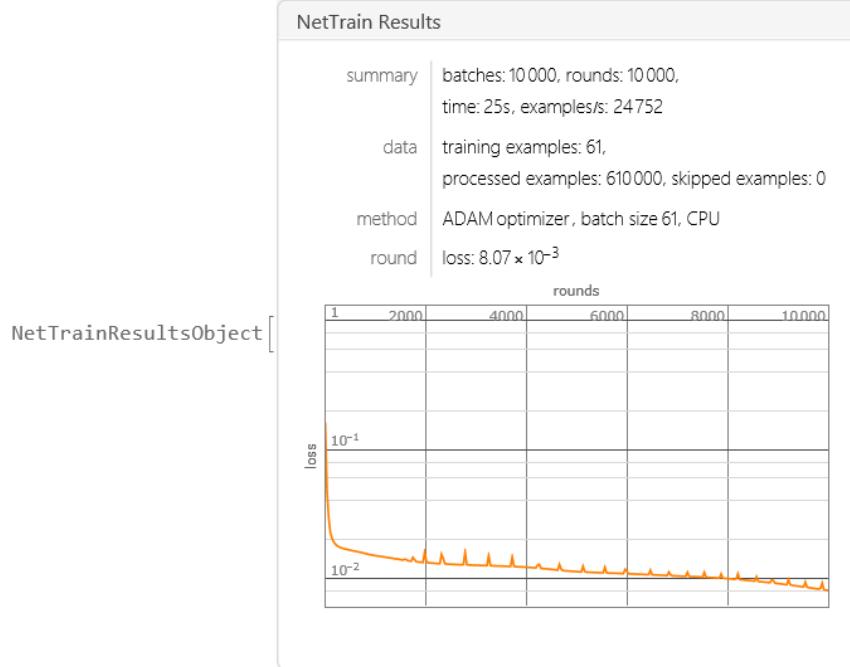
(* Calculate and display the mean test loss for the dropout network: *)
meanTestLossDropout=meanTestLoss[trainedDropoutNetwork];

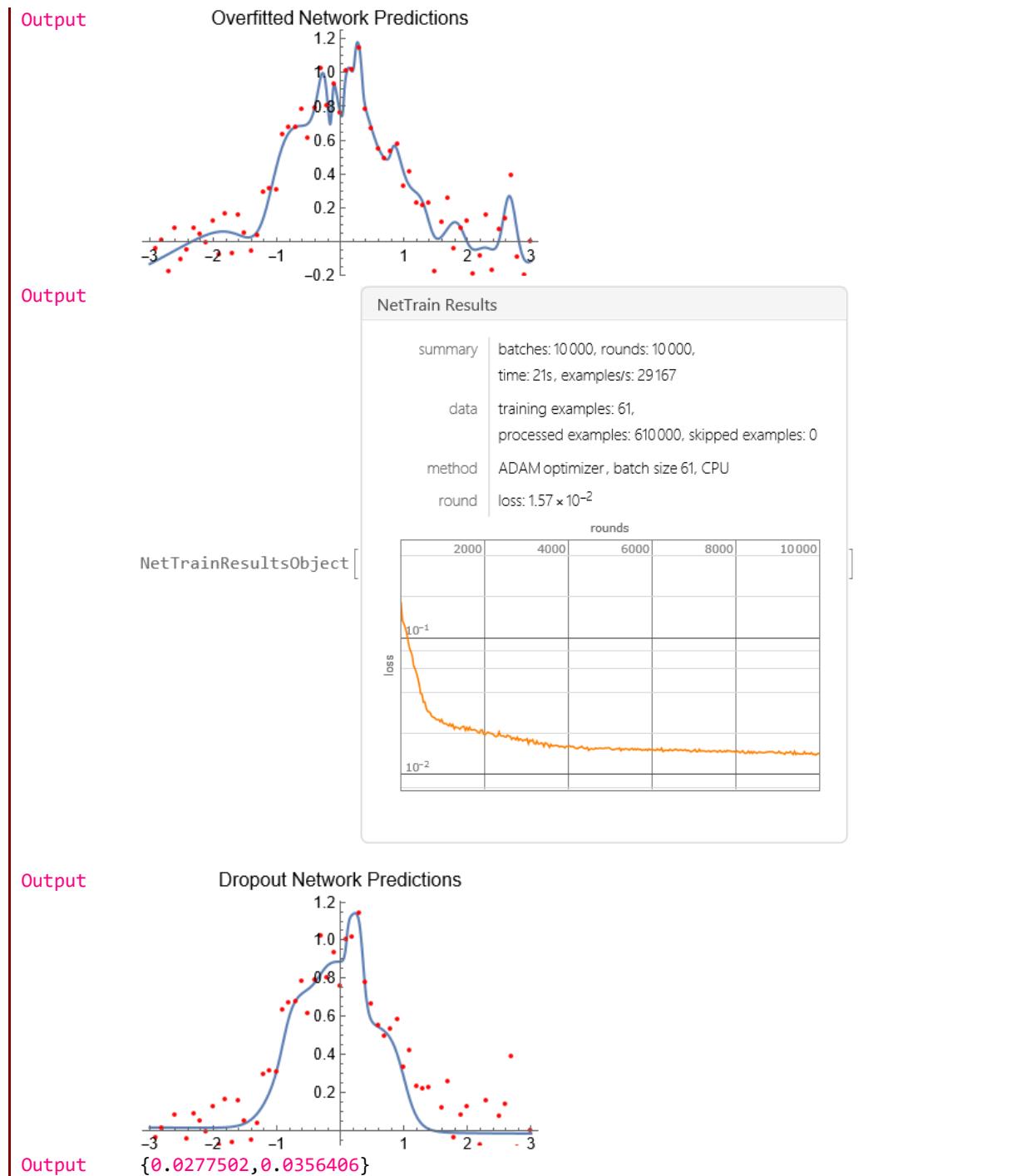
(* Calculate and display the mean test loss for the overfitted network: *)
meanTestLossOverfit=meanTestLoss[overfittedNetwork];

{meanTestLossDropout,meanTestLossOverfit}

```

Output



**Mathematica Code 8.11**

Input (* The code demonstrates the impact of different dropout rates on the performance of neural networks trained on synthetic data generated from the function $f(x)=\exp(-x^2)$ with added Gaussian noise. It defines this function, creates a noisy dataset, and constructs multiple neural networks with varying dropout rates (0.1, 0.5, 0.8). Each network is trained on the synthetic data for up to 5000 rounds using a specified learning rate. The code then plots the original data alongside the predictions of the trained networks, allowing for a visual comparison

```

of how different dropout rates influence the networks' ability to generalize and
fit the data, highlighting the effect of dropout as a regularization technique: *)

(* Define the target function g(x)=exp(-x^2): *)
targetFunction[x_]:=Exp[-x^2];

(* Generate synthetic data with added Gaussian noise: *)
syntheticData=Table[
  x->targetFunction[x]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.1}
];

(* Define different dropout rates: *)
dropoutRates={0.1,0.5,0.8};

(* Create a list of neural networks with the same architecture but different dropout
rates: *)
neuralNetworks=Table[
  NetChain[
    {
      (* First hidden layer with 100 units and Tanh activation: *)
      100,Tanh,
      (* Dropout layer with the specified rate: *)
      DropoutLayer[rate],
      (* Second hidden layer with 100 units and Tanh activation: *)
      100,Tanh,
      (*Output layer*)
      1
    }
  ],
  {rate,dropoutRates}
];

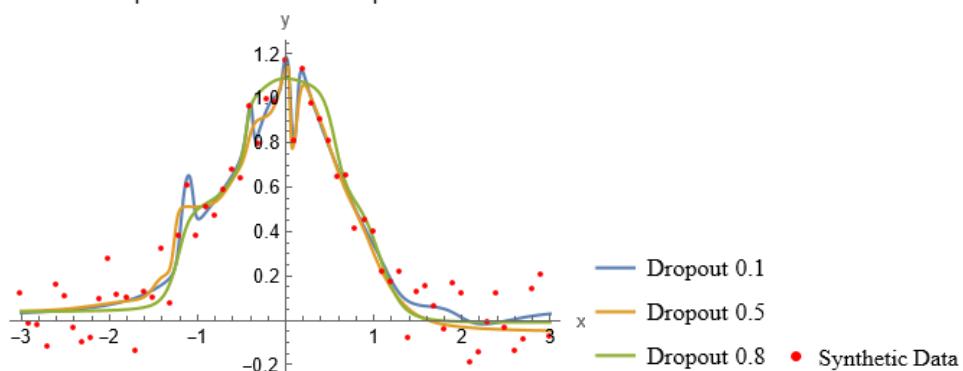
(* Train each neural network on the synthetic data with the corresponding dropout
rate: *)
trainedNetworks=Table[
  NetTrain[
    network,
    syntheticData,
    LearningRate->0.01,
    MaxTrainingRounds->5000
  ],
  {network,neuralNetworks}
];

(* Plot the original data and the models' predictions: *)
Show[
  Plot[
    {trainedNetworks[[1]][x],trainedNetworks[[2]][x],trainedNetworks[[3]][x]},
    {x,-3,3},
    ImageSize->300,
    PlotRange->All,
    PlotLegends->{"Dropout 0.1","Dropout 0.5","Dropout 0.8"},
    PlotLabel->"Comparison of Different Dropout Rates",
    AxesLabel->{"x","y"}
  ],
  ListPlot[
    List@@@syntheticData,
    PlotStyle->{Red,PointSize[0.01]},
    PlotLegends->{"Synthetic Data"}
  ]
];

```

Output

Comparison of Different Dropout Rates

**Mathematica Code 8.12**

Input (* The code demonstrates the effect of different dropout rates with L2 regularization on the performance of neural networks trained on synthetic data generated from the function $f(x)=\exp(-x^2)$ with added Gaussian noise. With L2 regularization, the weights of the neural network are constrained, which often leads to smoother and more stable predictions. In the figures, the predicted curves for networks trained with L2 regularization will appear smoother, better match the general trend of the data, showing less sensitivity to the noise, and less jagged compared to those without L2 regularization: *)

```
(* Define the target function g(x)=exp(-x^2): *)
targetFunction[x_]:=Exp[-x^2];

(* Generate synthetic data with added Gaussian noise: *)
syntheticData=Table[
  x->targetFunction[x]+RandomVariate[NormalDistribution[0,0.15]],
  {x,-3,3,0.1}
];

(* Define different dropout rates: *)
dropoutRates={0.1,0.5,0.8};

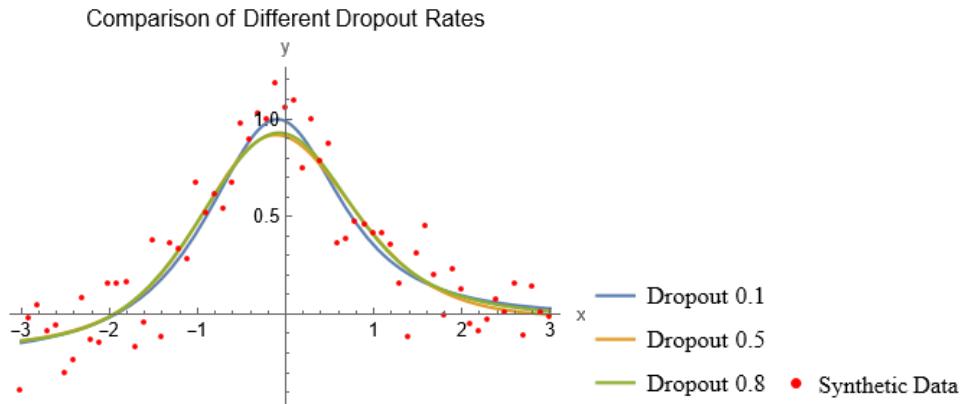
(* Create a list of neural networks with the same architecture but different dropout rates: *)
neuralNetworks=Table[
  NetChain[
    {(* First hidden layer with 100 units and Tanh activation: *)
     100,Tanh,
     (* Dropout layer with the specified rate: *)
     DropoutLayer[rate],
     (* Second hidden layer with 100 units and Tanh activation: *)
     100,Tanh,
     (* Output layer: *)
     1
    }
  ],
  {rate,dropoutRates}
];

(* Train each neural network on the synthetic data with the corresponding dropout rate: *)
```

```
trainedNetworks=Table[
  NetTrain[
    network,
    syntheticData,
    LearningRate->0.01,
    MaxTrainingRounds->5000,
    Method->{"SGD", "L2Regularization"->0.01}
  ],
  {network, neuralNetworks}
];

(* Plot the original data and the models' predictions: *)
Show[
  Plot[
    {trainedNetworks[[1]][x],trainedNetworks[[2]][x],trainedNetworks[[3]][x]},
    {x,-3,3},
    ImageSize->300,
    PlotRange->All,
    PlotLegends->{"Dropout 0.1","Dropout 0.5","Dropout 0.8"},
    PlotLabel->"Comparison of Different Dropout Rates",
    AxesLabel->{"x","y"}
  ],
  ListPlot[
    List@@@syntheticData,
    PlotStyle->{Red,PointSize[0.01]},
    PlotLegends->{"Synthetic Data"}
  ]
]
```

Output



CHAPTER 9

ADVANCED ACTIVATION FUNCTIONS

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 8, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation. This chapter also serves as a summary of the article titled *Deep Learning Activation Functions: Fixed-Shape, Parametric, Adaptive, Stochastic, Miscellaneous, Non-Standard, Ensemble*. For more details about activation functions, please refer to Ref [24]."

The activation functions (AFs) play a very crucial role in Neural Networks (NNs) by learning the abstract features through non-linear transformations. Some common properties of the AFs are as follows: a) it should add the non-linear curvature in the optimization landscape to improve the training convergence of the network; b) it should not increase the computational complexity of the model extensively; c) it should not hamper the gradient flow during training. Several AFs have been explored in recent years for deep learning to achieve the above-mentioned properties. For a more comprehensive background, we refer to [24] and the references cited therein.

In the present work, we categorized the AFs into six distinct groups: sigmoid-based, ReLU-based, ELU-based, miscellaneous, non-standard, and ensemble AFs.

Sigmoid-Based AFs: In order to introduce non-linearity into the NNs, the Logistic Sigmoid, and Tanh AFs have been used in the early days. The firing of biological neurons was the motivation for using the Logistic Sigmoid and Tanh AFs with artificial neurons. The Logistic Sigmoid and Tanh AFs majorly suffer from vanishing gradients. Several improvements have been proposed based on the Logistic Sigmoid AF. The most common Sigmoid-based/related functions are Tanh, HardSigmoid, and HardTanh, Penalized Tanh, Soft-Root-Sign, and Sigmoid-Weighted Linear.

ReLU-Based AFs: The saturated output and increased complexity are the key limitations of the above-mentioned Logistic Sigmoid-based AFs. The ReLU has become the state-of-the-art AF due to its simplicity and improved performance. Various variants of ReLU have been investigated by tackling its drawbacks, such as non-utilization of negative values, limited non-linearity, and unbounded output. The most common ReLU-based/related functions are: Leaky Rectified Linear Unit, Parametric ReLU, Randomized ReLU, Random Translation ReLU, Elastic ReLU, Elastic Parametric ReLU, Linearized Sigmoidal Activation, Rectified Linear Tanh, Shifted ReLU, Displaced ReLU and Multi-bin Trainable Linear Unit.

ELU-Based AFs: The major problem faced by the Logistic Sigmoid-based AFs is with its saturated output for large positive and negative input. Similarly, the major problem with ReLU-based AFs is the under-utilization of negative values leading to a vanishing gradient. In order to cope up with these limitations the ELU-based AFs have been used in the literature. The ELU-based AF utilizes the negative values with the help of the exponential function. Several AFs have been introduced in the literature as ELU variants. The most common ELU-based/related functions are Scaled ELU, Parametric ELU, Rectified Exponential Unit, Parametric Rectified Exponential Unit, and Elastic ELU.

Non-Standard AFs: Non-standard AFs include those that combine multiple standard functions or operate on different principles. The common non-standard AFs are Maxout and Softmax.

Miscellaneous AFs: Such as Swish-based/related AFs (Swish, E-Swish, HardSwish), SoftPlus-based/related AFs (SoftPlus, SoftPlus Linear Unit, Mish), Probabilistic AF (Gaussian Error Linear Unit, and Symmetrical Gaussian Error Linear Unit).

Combining AFs: Most of the Sigmoid, Tanh, ReLU, and ELU-based AFs are designed manually which might not be able to exploit the data complexity. Combining AFs are the recent trends. The most common AF are Mixed, Gated, and Hierarchical AFs, Adaptive Piecewise Linear Units, Mexican ReLU, Look-up Table Unit, and Bi-Modal Derivative Sigmoidal AFs. Unlike traditional AFs such as ReLU, Sigmoid, or ELU, which have fixed functional forms, adaptive AFs learn their parameters from the data, allowing the network to adapt more flexibly to different tasks.

In this chapter, we embark on an in-depth exploration of AFs and custom layers within NNs. The primary goal is to provide a comprehensive understanding of how these functions influence the behavior of NNs. We will achieve this through interactive simulations, detailed comparisons, and practical implementations of custom layers. This chapter is structured into three units, each focusing on a specific aspect of AFs and custom layers.

UNIT 9.1: Interactive Exploration of AF:

In this unit, we explore the behavior of AFs by creating interactive Manipulate visualizations. These visualizations allow us to dynamically change the parameters and observe how they affect the AFs. By adjusting parameters such as slope, threshold, and others, we gain deeper insights into how each AF transforms its input and impacts the overall behavior of a NN. This hands-on approach helps to solidify our understanding of the role of AFs in NNs. Through these interactive simulations, you will develop an intuitive grasp of the mathematical underpinnings and practical implications of various AFs. By experimenting with different parameter settings, you will see firsthand how AFs contribute to the non-linear transformations essential for NN learning.

UNIT 9.2: Custom Layers in NNs:

This unit delves into the construction and utilization of custom layers in NNs. Custom layers are fundamental components that allow for the extension and customization of NN architectures to better suit specific tasks. We will explore several key custom layers, including [FunctionLayer](#), [ParametricRampLayer](#), [SoftmaxLayer](#), [NetEncoder](#), and [NetDecoder](#). Key topics covered in this unit include an overview of custom layers and their role in NN design, a detailed exploration of [FunctionLayer](#) and how it allows for the definition of arbitrary functions within a network, understanding [ParametricRampLayer](#) and its use for implementing piecewise linear functions, an examination of [SoftmaxLayer](#) for transforming raw network outputs into probability distributions, and insights into [NetEncoder](#) and [NetDecoder](#) for preprocessing input data and postprocessing network outputs, respectively. By the end of this unit, you will have a solid understanding of how to create and integrate custom layers into NNs, enhancing their flexibility and capability to tackle a wide range of problems.

UNIT 9.3: Comparison of Some AFs:

This unit provides a detailed comparison of several commonly used AFs. We will analyze the characteristics and performance implications of ReLU, ELU, SELU, GELU, Swish, HardSwish, Mish, SoftPlus, HardTanh, HardSigmoid, Sigmoid, and Tanh. Each AF will be examined in terms of its mathematical formulation, gradient behavior, and suitability for different types of NN architectures. Through this comprehensive comparison, you will gain a clear understanding of the strengths and weaknesses of various AFs, enabling you to make informed decisions when designing and optimizing NNs.

By the end of this chapter, you will have a robust understanding of how AFs shape the learning dynamics of NNs and how custom layers can be leveraged to build more sophisticated and efficient models.

Unit 9.1

Interactive Exploration of Activation Functions

Mathematica Code 9.1

```

Input      (* The goal of the code is to define the logistic function (logisticFunction) and
           its derivative (logisticDerivative), and create an interactive visualization using
           the Manipulate function. This visualization allows users to dynamically adjust the
           input value via a slider, and observe the behavior of both the logistic function
           and its derivative over the range of x from -5 to 5. The plot includes legends to
           differentiate between the functions, and highlights the points on the curves
           corresponding to the current input value: *)

(* Logistic function definition: *)
logisticFunction[x_]:=1/(1+Exp[-x])

(* Derivative of the logistic function: *)
logisticDerivative[x_]:=logisticFunction[x] (1-logisticFunction[x])

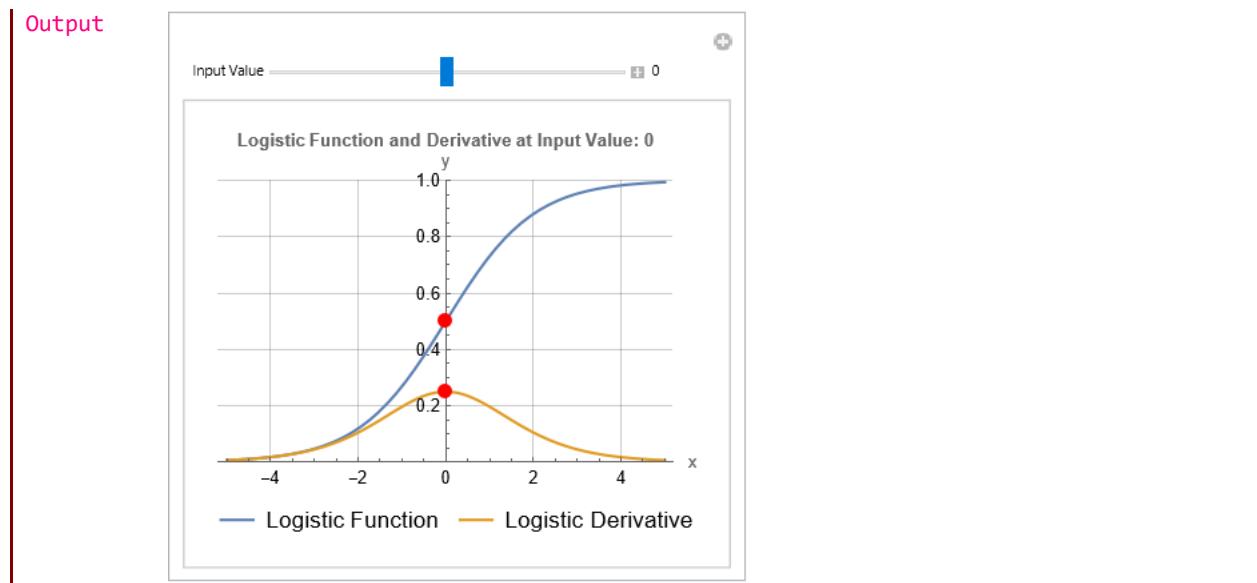
(* Interactive visualization using Manipulate: *)
Manipulate[
 Plot[
  (* Plot both the logistic function and its derivative:*)
  {logisticFunction[x],logisticDerivative[x]},
  {x,-5,5},
  PlotLegends->Placed[{"Logistic Function","Logistic Derivative"},Below],
  AxesLabel->{"x","y"},
  PlotRange->{All,{0,1}},

  (* Highlight the points on the curves for the current input value: *)
  Epilog->{
    Red,
    PointSize[0.03],

    Point[{{inputValue,logisticFunction[inputValue]},{inputValue,logisticDerivative[inputValue]}}]
  },
  PlotLabel->Style[Row[{"Logistic Function and Derivative at Input Value:",
  inputValue}],10,Bold],
  GridLines->Automatic,
  ImageSize->270
  ],

  (* Create a slider to adjust the input value from -5 to 5: *)
  {{inputValue,0,"Input Value"},-5,5,Appearance->"Labeled"}
]

```

**Mathematica Code 9.2**

```

Input (* The goal of the code is to define the Heaviside step function
(`heavisideStepFunction`) and the logistic sigmoid function
(`logisticSigmoidFunction`), and create an interactive visualization using the
`Manipulate` function. This visualization allows users to dynamically adjust the
steepness parameter (`k`) via a slider and observe the behavior of both functions
over the range of `x` from -5 to 5. The plot includes legends for differentiation,
and highlights specific points at (0,0) and (0,1). The interactive control aims to
provide an intuitive understanding of how the logistic sigmoid function changes
with different steepness parameters, facilitating a comparison with the Heaviside
step function: *)

(* Define the Heaviside step function with a threshold parameter: *)
heavisideStepFunction[x_,threshold_]:=If[x<threshold,0,1]

(* Define the logistic sigmoid function with a steepness parameter k: *)
logisticSigmoidFunction[x_,k_]:=1/(1+Exp[-k*x])

(* Interactive visualization using Manipulate: *)
Manipulate[
 Plot[
  (* Plot both the Heaviside step function and the logistic sigmoid function with
adjustable steepness: *)
  {heavisideStepFunction[x,0],logisticSigmoidFunction[x,k]},
  {x,-5,5},
  PlotStyle->Thick,
  AxesLabel->{"x","y"},
  PlotLegends->Placed[{"Heaviside Step Function","Logistic Sigmoid
(k="<>ToString[k]<>")"},Below],
  PlotRange->{0,1},
  PlotLabel->"Step Function vs. Logistic Sigmoid Function",
  ImageSize->250,
  GridLines->Automatic,
  (* Highlight the points at (0,0) and (0,1) on the plot: *)
  Epilog->
  {
    Red,
    PointSize[0.02],
    Point[{0,0}],
    Point[{0,1}]
  }
]

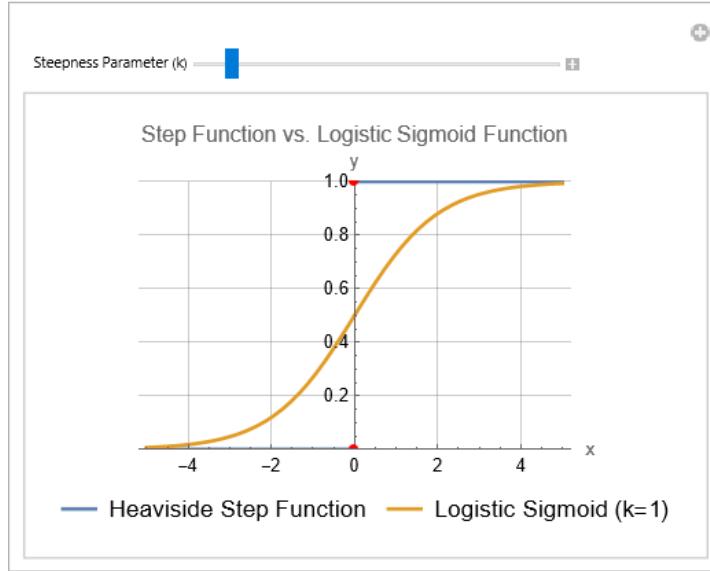
```

```

        Point[{0,1}]
    }
],
(* Create a slider to adjust the steepness parameter k from 0.1 to 10: *)
{{k,1,"Steepness Parameter (k)"},0.1,10,0.1}
]

```

Output

**Mathematica Code 9.3**

Input (* The goal of the code is to define the logistic sigmoid function (`logisticSigmoid`) and create an interactive visualization using the `Manipulate` function. This visualization allows users to dynamically adjust the threshold parameter using a slider and observe the behavior of both the standard Unit Step function and the logistic sigmoid function over the range of `x` from -5 to 5. Additionally, it compares the shifted Unit Step function (with an adjustable threshold) to the logistic sigmoid function. The interactive control provides real-time feedback on the threshold value, offering an intuitive understanding of how the step function's shift affects its comparison to the sigmoid function: *)

```

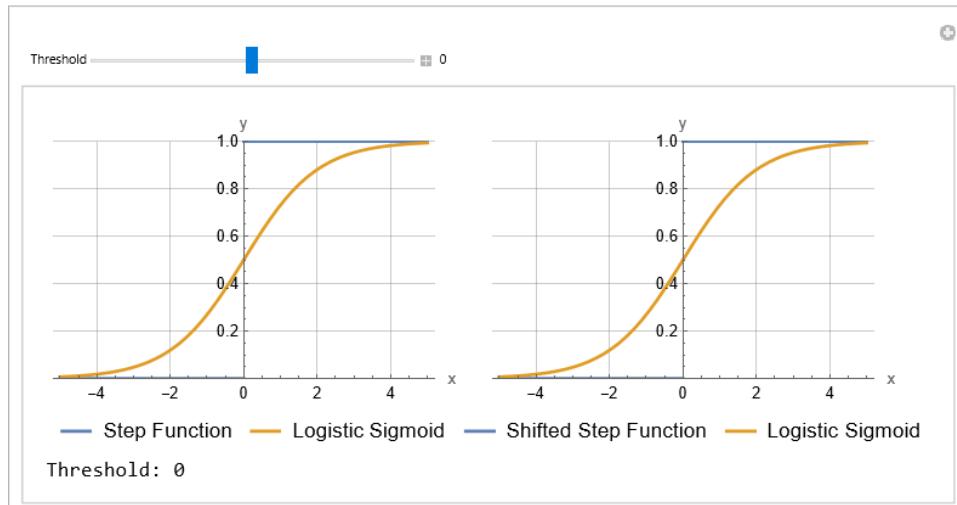
(* Define the logistic sigmoid function: *)
logisticSigmoid[x_]:=1/(1+Exp[-x])

(* Interactive visualization using Manipulate: *)
Manipulate[
Column[
{
(* Create a row to display two plots side by side: *)
Row[
{(* Plot the standard Unit Step function and the logistic sigmoid function: *)
Plot[
{UnitStep[x],logisticSigmoid[x]},
{x,-5,5},
PlotLegends->Placed[{"Step Function","Logistic Sigmoid"},Below],
AxesLabel->{"x","y"},
PlotRange->{All,{0,1}},
PlotStyle->Thick,
GridLines->Automatic,
ImageSize->250
],

```

```
(* Plot the shifted Unit Step function and the logistic sigmoid function:
*)
Plot[
{UnitStep[x-threshold], logisticSigmoid[x]},
{x, -5, 5},
PlotLegends -> Placed[{"Shifted Step Function", "Logistic Sigmoid"}, Below],
AxesLabel -> {"x", "y"},
PlotRange -> {All, {0, 1}},
PlotStyle -> Thick,
GridLines -> Automatic,
ImageSize -> 250
]
],
(* Display the current threshold value*)
Row[{"Threshold: ", threshold}]
],
(*Create a slider to adjust the threshold value from -5 to 5*)
{{threshold, 0, "Threshold"}, -5, 5, Appearance -> "Labeled"}
]
```

Output

**Mathematica Code 9.4**

Input (* The goal of the code is to define the logistic sigmoid and hyperbolic tangent (tanh) functions, along with their derivatives, and create visualizations to compare these functions. It calculates the absolute differences between the sigmoid and tanh functions, as well as their derivatives. The code then plots the sigmoid function, tanh function, and their absolute difference over the range of `x` from -5 to 5, and similarly plots the derivatives of these functions and their absolute difference. These visualizations include legends and axis labels to clearly present the comparisons, providing a comprehensive tool to understand the behavior and differences between the sigmoid and tanh functions and their respective derivatives: *)

```
(* Define sigmoid and tanh functions: *)
sigmoid[x_]:=1/(1+Exp[-x])
tanh[x_]:=Tanh[x]

(* Define the absolute difference between sigmoid and tanh: *)
absoluteDifference[x_]:=Abs[sigmoid[x]-tanh[x]]

(* Define the derivatives of sigmoid and tanh: *)
```

```

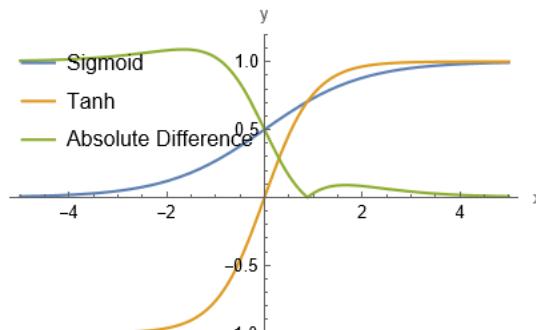
sigmoidDerivative[x_]:=sigmoid[x]*(1-sigmoid[x])
tanhDerivative[x_]:=1-tanh[x]^2

(* Define the absolute difference between the derivatives: *)
absoluteDifferenceDerivative[x_]:=Abs[sigmoidDerivative[x]-tanhDerivative[x]]

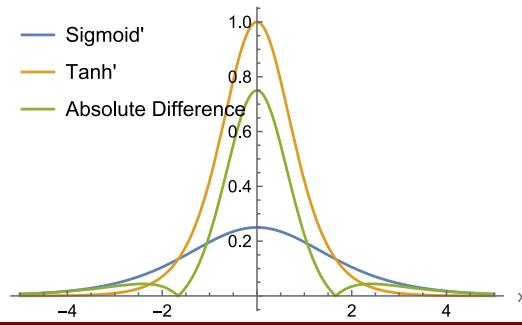
(* Plot the functions and absolute difference: *)
Plot[
 {sigmoid[x],tanh[x],absoluteDifference[x]},
 {x,-5,5},
 PlotLegends->Placed[{{"Sigmoid","Tanh","Absolute Difference"},{0.25,0.78}}],
 AxesLabel->{"x","y"},
 PlotRange->All,
 ImageSize->300
]
(* Plot the derivatives and absolute difference: *)
Plot[
 {sigmoidDerivative[x],tanhDerivative[x],absoluteDifferenceDerivative[x]},
 {x,-5,5},
 PlotLegends->Placed[{{"Sigmoid'","Tanh'","Absolute Difference"},{0.25,0.78}}],
 AxesLabel->{"x","y"},
 PlotRange->All,
 ImageSize->300
]

```

Output



Output

**Mathematica Code 9.5**

Input (* The code defines and explores the behavior of a penalized tanh activation function (penalizedTanh) that adjusts based on a hyperparameter alpha. Using the Manipulate function, it creates an interactive plot that allows users to dynamically adjust alpha with a slider and observe the function's behavior over the range of x from -3 to 3. This interactive visualization provides an intuitive way to understand how varying alpha impacts the penalized tanh activation function: *)

```

Manipulate[
 Module[

```

```

{penalizedTanh,plotRange},

(* Define the penalized tanh activation function: *)
penalizedTanh[x_]:=If[x>0,Tanh[x],alpha*Tanh[x]];
plotRange={-1,1};

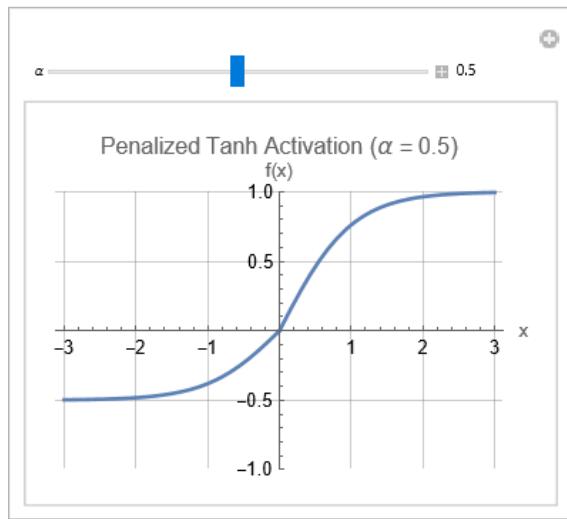
(* Plot the penalized tanh activation function: *)
Plot[
  penalizedTanh[x],
  {x,-3,3},
  PlotRange->plotRange,
  PlotStyle->Thick,
  AxesLabel->{"x","f(x)"},
  ImageSize->250,
  GridLines->Automatic,

  (* Add a label for the plot showing the current value of alpha: *)
  PlotLabel->Row[{"Penalized Tanh Activation (\u03b1 = ",alpha,")"}]
],
];

(* Create a slider to adjust the hyperparameter alpha from 0 to 1 with a step of
0.1: *)
{{alpha,0.5,"\u03b1"},0,1,0.1,Appearance->"Labeled"}
]

```

Output



Mathematica Code 9.6

Input (* The goal of the code is to define and explore the behavior of the Soft-Root-Sign (SRS) activation function and its derivative, parameterized by α and β . Using Mathematica's `Manipulate` function, it creates an interactive plot that allows users to adjust α and β with sliders, dynamically updating the visualization. The plot displays the SRS function and its derivative over the range of x from -10 to 10, with legends to distinguish between them, and a plot label showing the current parameter values. The plot range is set from -2 to 3 for clear visualization. This interactive tool helps users understand how the SRS function and its derivative change with different α and β values: *)

```

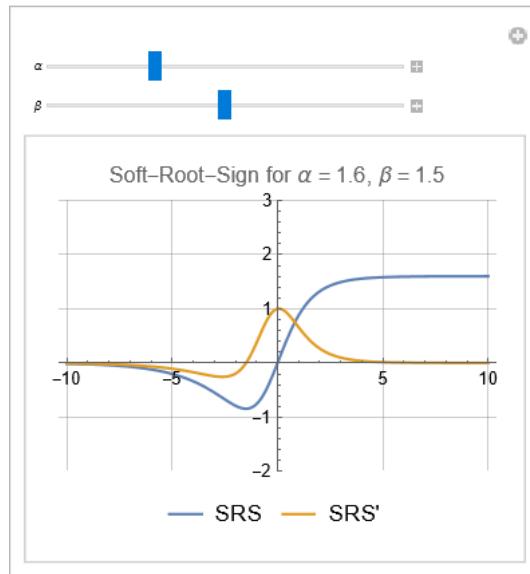
(* Define the Soft-Root-Sign activation function: *)
softRootSign[x_,\u03b1_,\u03b2_]:=x/((x/\u03b1)+Exp[-x/\u03b2])

```

```
(* Define the derivative of the Soft-Root-Sign activation function: *)
softRootSignDerivative[x_, α_, β_]:=Evaluate[D[softRootSign[t,α,β],t]/. t->x]

Manipulate[
 Plot[
 {softRootSign[x,α,β],softRootSignDerivative[x,α,β]},
 {x,-10,10},
 PlotRange->{-2,3},
 PlotLabel->StringForm["Soft-Root-Sign for α = `1`, β = `2`",α,β],
 PlotLegends->Placed[{"SRS","SRS'"},Below],
 ImageSize->250,
 GridLines->Automatic
 ],
 (* Create sliders to adjust the hyperparameter α and β: *)
 {{α,1.6,"α"},1,3,0.1},
 {{β,1.5,"β"},0.5,2.5,0.1}
 ]
```

Output

**Mathematica Code 9.7**

Input (* The code defines the ReLU (Rectified Linear Unit) activation function and create an interactive visualization using Mathematica's Manipulate function. This visualization allows users to dynamically adjust the slope of a line in the subdifferential interval at x=0 using a slider. The plot displays the ReLU function over the range of x from -1 to 1, with a thick blue line for the ReLU function, an adjustable orange line representing possible slopes, and a red point at (0, relu[0]). This interactive tool helps users understand the impact of different slopes in the subdifferential interval at x=0, enhancing comprehension of the ReLU function's behavior around the origin: *)

```
(* Define the ReLU activation function: *)
relu[x_]:=Max[0,x]

(* Create an interactive visualization using Manipulate: *)
Manipulate[
 Plot[
 relu[x],
 {x,-1,1},
 PlotStyle->Directive[Blue,Thick],
 Epilog-{
```

```

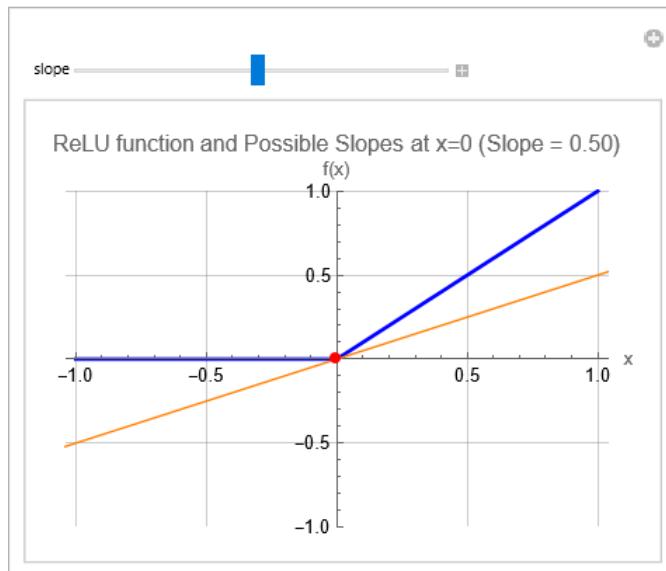
Orange,
Line[{{-1/slope,-1},{1/slope,1}}],
Red,
PointSize[0.02],
Point[{0,relu[0]}]
},
PlotRange->{-1,1},

(*Add a label for the plot with the current slope value*)
PlotLabel->Row[{ "ReLU function and Possible Slopes at x=0 (Slope =",
NumberForm[slope,{3,2}],")"}], 

AxesLabel->{"x","f(x)"}, 
ImageSize->310,
GridLines->Automatic
],
(* Create a slider to adjust the slope from 0.01 to 1 with a step of 0.1: *)
{{slope,0.5,"slope"},0.01,1,0.1}
]

```

Output



Mathematica Code 9.8

Input

```

(* The code defines two activation functions, ReLU and PReLU, and creates an
interactive visualization using Manipulate function to dynamically adjust the
parameter alpha for the PReLU function. The code generates a plot displaying both
functions over a specified range, allowing for real-time comparison as the alpha
parameter is adjusted using a slider: *)

(* Define the ReLU activation function: *)
ReLUFunction[x_]:=Max[0,x]

(* Define the PReLU activation function with parameter alpha: *)
PReLUFunction[x_,alpha_]:=Piecewise[{{x,x>0},{alpha*x,x<0}}]

(* Create a dynamic interface to manipulate the alpha parameter: *)
Manipulate[
 (*Plot both ReLU and PReLU functions on the same graph*)
 Plot[

```

```

{ReLUFunction[x],PReLUFunction[x,alpha]},  

{x,-3,3},  

PlotRange->{-3,3},  

PlotLegends->Placed[{"ReLU","PReLU"},Below],  

Epilog->Text[" $\alpha = " \> ToString[alpha],{2,1.5}"],  

PlotLabel->"ReLU and PReLU Activation Functions",  

ImageSize->250 ,  

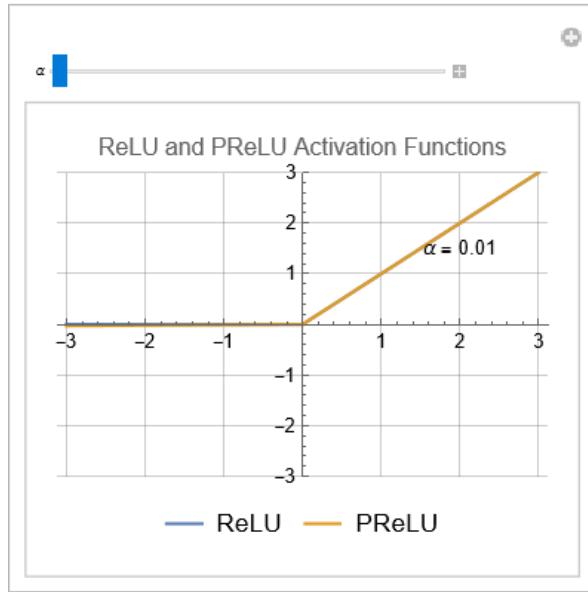
GridLines->Automatic  

],  

(* Slider to adjust the alpha parameter: *)  

{{alpha,0.01," $\alpha"},0,1,0.01}
]$$ 
```

Output

**Mathematica Code 9.9**

Input (* The code defines the ReLU and Parameterized Randomized ReLU (RReLU) activation functions, initializes random parameters for these functions, and creates an interactive visualization using Manipulate function. This visualization allows users to dynamically adjust the parameters α_1 and α_2 for the RReLU functions using sliders, and observe real-time changes in the plot. The plot displays the ReLU function and two RReLU functions, with shading between the curves to highlight differences. This interactive tool helps users understand the impact of the α parameters on the RReLU functions and compare them to the standard ReLU function: *)

```

(* Define the ReLU activation function: *)
ReLUFunction[x_]:=Max[0,x]

(* Define the Parameterized Randomized ReLU activation function with parameter
alpha: *)
ParameterizedRandomReLU[x_,alpha_]:=Max[alpha x,x]

(* Initialize specific alpha values: *)
initialAlpha1=RandomVariate[UniformDistribution[{0,1}]];
initialAlpha2=RandomVariate[UniformDistribution[{0,1}]];

(* Create a dynamic interface to manipulate the alpha parameters: *)
Manipulate[
(* Plot the ReLU and two RReLU functions on the same graph: *)

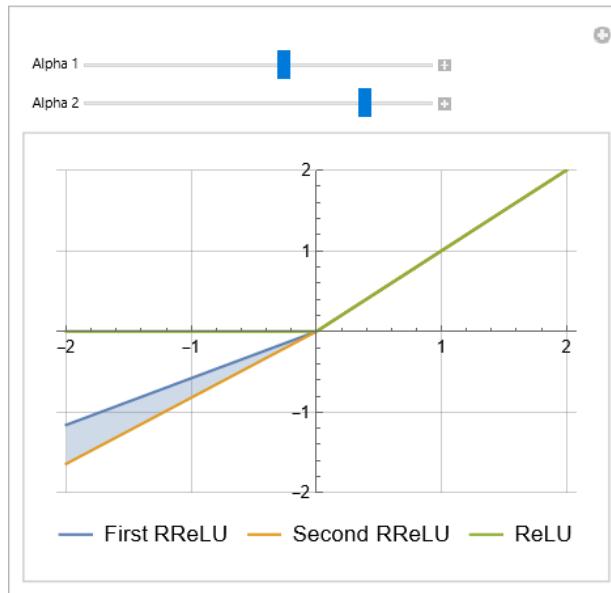
```

```

Plot[
{ParameterizedRandomReLU[x, alpha1], (* First RReLU function with alpha1: *)
 ParameterizedRandomReLU[x, alpha2], (* Second RReLU function with alpha2: *)
 ReLUFunction[x] (* Standard ReLU function: *)},
{x, -2, 2},
PlotRange->{-2, 2},
Filling->{1->{2}},
PlotLegends->Placed[{"First RReLU", "Second RReLU", "ReLU"}, Below],
FillingStyle->{LightGray, Opacity[0.3]},
GridLines->Automatic,
ImageSize->300
],
(* Slider to adjust the first alpha parameter: *)
{{alpha1, initialAlpha1, "Alpha 1"}, 0, 1, 0.01},
(* Slider to adjust the second alpha parameter: *)
{{alpha2, initialAlpha2, "Alpha 2"}, 0, 1, 0.01}
]

```

Output

**Mathematica Code 9.10**

Input (* The code defines the standard ReLU and a Random Transformed ReLU (RTReLU) function, incorporating an interactive interface using Manipulate function. It allows users to dynamically adjust the random seed and the standard deviation σ of a Gaussian distribution, which generates a random offset (a) for the RTReLU function. The plot updates in real-time, displaying both the RTReLU and ReLU functions over a specified range, with dashed lines indicating the offset values ($-a$ and a): *)

```

(* Define the Random Transformed ReLU activation function with a random offset (a): *)
RandomTransformedReLU[x_, a_] := If[x + a > 0, x + a, 0]

(* Define the standard ReLU activation function: *)
ReLUFunction[x_] := Max[0, x]

(* Create a dynamic interface to manipulate the random seed and standard deviation parameters: *)
Manipulate[

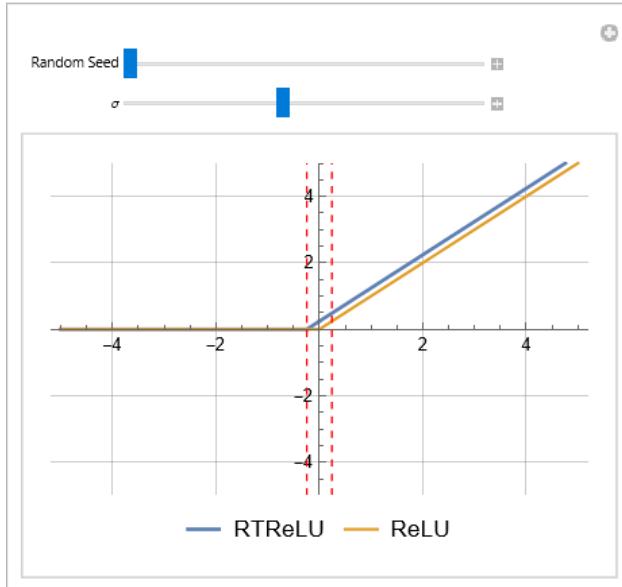
```

```
(* Set the random seed to ensure reproducibility: *)
SeedRandom[randomSeed];

(* Generate a new random offset based on the Gaussian distribution with mean 0
and standard deviation sigma: *)
a=RandomVariate[NormalDistribution[0,sigma]];

(* Plot the Random Transformed ReLU and standard ReLU functions on the same graph: *)
Plot[
{RandomTransformedReLU[x,a],ReLUFunction[x]},
{x,-5,5},
PlotRange->{Automatic,{-5,5}},
PlotLegends->Placed[{"RTReLU","ReLU"},Below],
PlotStyle->{Thick,Automatic},
GridLines->Automatic,
(* Add dashed vertical lines at positions -a and a*)
Epilog->{
Red,
Dashed,
Line[{{-a,-5},{-a,5}}],
Line[{{a,-5},{a,5}}]
},
ImageSize->300
],
(* Slider to adjust the random seed: *)
{{randomSeed,1,"Random Seed"},1,100,1},
(* Slider to adjust the standard deviation of the Gaussian distribution: *)
{{sigma,0.5,"σ"},0.1,1,0.1}
]
```

Output

**Mathematica Code 9.11**

Input (* The code defines the displaced ReLU, standard ReLU, and ELU (Exponential Linear Unit) activation functions, and creates an interactive interface using Manipulate function. This interface includes a slider that allows users to dynamically adjust the displacement parameter delta (δ) for the displaced ReLU function. The plot,

```
which updates in real-time, displays the displaced ReLU, standard ReLU, and ELU functions over a specified range, enabling users to visually compare them: *)
```

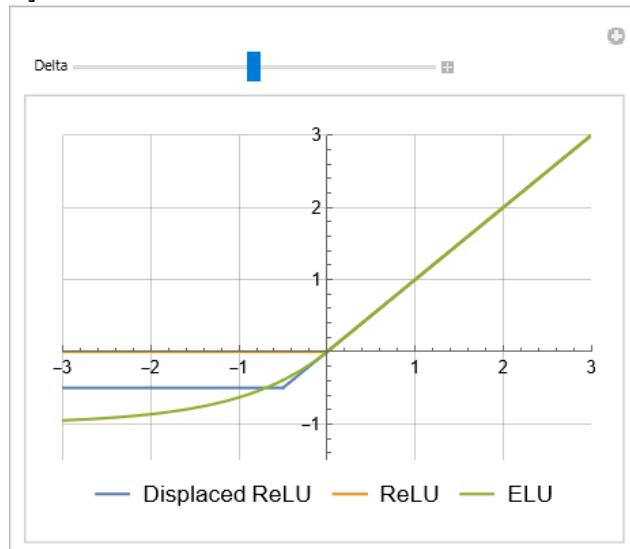
```
(* Define the activation functions: *)
(* Define the displaced ReLU activation function with a displacement parameter
delta: *)
displacedReLU[x_,delta_]:=If[x>=-delta,x,-delta]

(* Define the standard ReLU activation function: *)
ReLUFunction[x_]:=Max[0,x]

(* Define the ELU (Exponential Linear Unit) activation function with parameter
alpha: *)
ELUFunction[x_,alpha_]:=If[x>=0,x,alpha*(Exp[x]-1)]

(* Create Manipulate: *)
Manipulate[
 (* Plot the displaced ReLU, standard ReLU, and ELU functions on the same graph: *)
Plot[
 {displacedReLU[x,delta],ReLUFunction[x],ELUFunction[x,1]},
 {x,-3,3},
 PlotLegends->Placed[{"Displaced ReLU","ReLU","ELU"},Below],
 PlotRange->{{-3,3},{-1.5,3}},
 GridLines->Automatic,
 ImageSize->300
 ],
 (* Slider to adjust the delta parameter: *)
 {{delta,0.5,"Delta"},0,1,0.1}
]
```

Output

**Mathematica Code 9.12**

Input

```
(* The code defines the Elastic ReLU (EReLU) function along with its lower and
upper bounds, and creates an interactive visualization using Manipulate function.
This interface allows users to dynamically adjust the parameter alpha ( $\alpha$ ) and the
random seed, which controls the randomness in the EReLU function. By seeding the
random number generator and selecting a random parameter (R) from a uniform
distribution, the plot updates in real-time to display the EReLU function and its
bounds. The area between the bounds is filled to highlight the function's
variability: *)
```

```

(* Lower bound function for the Elastic ReLU (EReLU): *)
lowerBoundFunction[x_,alpha_]:=Piecewise[{{(1-alpha) x,x>0},{0,x<0}}]

(* Upper bound function for the Elastic ReLU (EReLU): *)
upperBoundFunction[x_,alpha_]:=Piecewise[{{(1+alpha) x,x>0},{0,x<0}}]

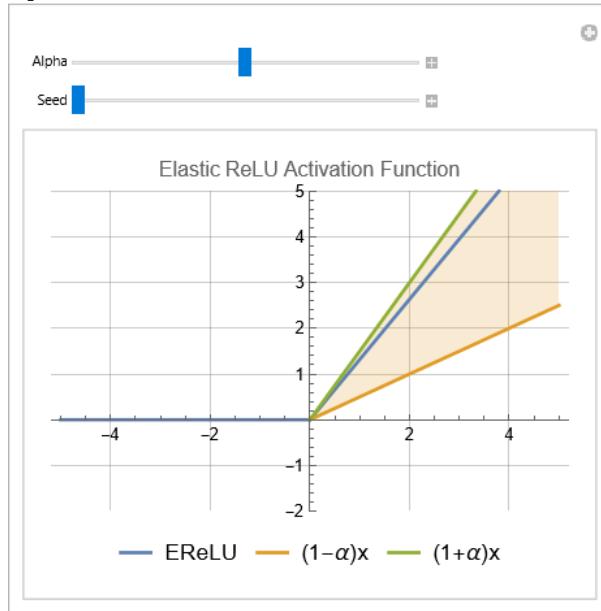
(* Define the Elastic ReLU (EReLU) function: *)
elasticReLUFunction[x_]:=Piecewise[{{randomParameter x,x>=0},{0,x<0}}]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Seed the random number generator for reproducibility: *)
  SeedRandom[randomSeed];

  (* Randomly select random parameter (R) from a uniform distribution within
   specified bounds: *)
  randomParameter=RandomVariate[UniformDistribution[{1-alpha,1+alpha}]];
  (* Plot EReLU along with lower and upper bounds: *)
  Plot[
    {elasticReLUFunction[x],lowerBoundFunction[x,alpha],upperBoundFunction[x,alpha]},
    {x,-5,5},
    PlotLegends->Placed[{"EReLU", "(1- $\alpha$ )x", "(1+ $\alpha$ )x"}, Below],
    (* Filling between the lower and upper bounds: *)
    Filling->{2->{3}},
    PlotRange->{-2,5},
    PlotLabel->"Elastic ReLU Activation Function",
    ImageSize->300,
    GridLines->Automatic,
    PlotStyle->Thick
  ],
  (* Slider for alpha parameter: *)
  {{alpha,0.5,"Alpha"},0,1,0.01},
  (* Slider for seed value: *)
  {{randomSeed,1,"Seed"},1,100,1}
]

```

Output



Mathematica Code 9.13

```

Input      (* The code defines the Elastic Parametric ReLU (EPReLU) function along with its
           lower and upper bounds, and creates an interactive visualization using
           Mathematica's `Manipulate` function. This interface allows users to dynamically
           adjust the parameter alpha ( $\alpha$ ), the negative slope parameter ( $a$ ), and the random
           seed, which controls the randomness in the EPReLU function. By seeding the random
           number generator and selecting a random parameter ( $R$ ) from a uniform distribution,
           the plot updates in real-time to display the EPReLU function and its bounds. The
           area between the bounds is filled to highlight the function's variability. This
           interactive tool helps users explore the impact of the parameters alpha ( $\alpha$ ) and
           ( $a$ ) and the effect of randomness on the EPReLU function: *)

(* Lower bound function for the Elastic Parametric ReLU (EPReLU): *)
lowerBoundEPReLU[x_,alpha_]:=Piecewise[{{(1-alpha) x,x>0},{0,x<0}}]

(* Upper bound function for the Elastic Parametric ReLU (EPReLU): *)
upperBoundEPReLU[x_,alpha_]:=Piecewise[{{(1+alpha) x,x>0},{0,x<0}}]

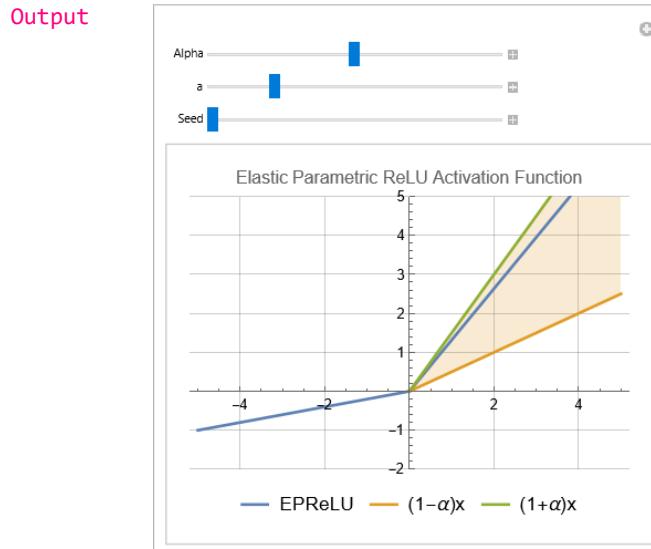
(* Define the Elastic Parametric ReLU (EPReLU) function: *)
elasticPReLUFunction[x_,a_]:=Piecewise[{{randomFactor x,x>0},{a*x,x<0}}]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Seed the random number generator for reproducibility: *)
  SeedRandom[randomSeed];

  (* Randomly select R from a uniform distribution within specified bounds: *)
  randomFactor=RandomVariate[UniformDistribution[{1-alpha,1+alpha}]]];

  (* Plot EPReLU along with lower and upper bounds: *)
  Plot[
    {elasticPReLUFunction[x,a],lowerBoundEPReLU[x,alpha],upperBoundEPReLU[x,alpha]},
    {x,-5,5},
    PlotLegends->Placed[{"EPReLU","(1- $\alpha$ )x","(1+ $\alpha$ )x"},Below],
    (* Filling between the lower and upper bounds: *)
    Filling->{2->{3}},
    PlotRange->{-2,5},
    PlotLabel->"Elastic Parametric ReLU Activation Function",
    ImageSize->300,
    GridLines->Automatic,
    PlotStyle->Thick
  ],
  (* Slider for alpha parameter: *)
  {{alpha,0.5,"Alpha"},0,1,0.01},
  (* Slider for a parameter: *)
  {{a,0.2,"a"},0,0.9,0.1},
  (* Slider for seed value: *)
  {{randomSeed,1,"Seed"},1,100,1}
]

```

**Mathematica Code 9.14**

```

Input (* The code defines the Linearized Sigmoidal and Adaptive Linearized Sigmoidal Activation (ALiSA) function along with its upper and lower bounds for both positive and negative input ranges. It creates an interactive interface using Mathematica's `Manipulate` function, allowing users to dynamically adjust the slope parameters alpha1 ( $\alpha_1$ ) and alpha2 ( $\alpha_2$ ) via sliders. The plot updates in real-time to display the ALiSA function and its bounds, with filled areas highlighting the function's variability: *)

(* Define the Adaptive Linearized Sigmoidal Activation function: *)
adaptiveLinearizedSigmoid[x_,alpha1_,alpha2_]:=Piecewise[
  {
    {alpha1 x-alpha1+1,1<=x< $\infty$ },
    {x,0<=x<=1},
    {alpha2 x,- $\infty$ <x<0}
  }
]
(* Upper bound function for the Adaptive Linearized Sigmoidal (ALiSA), 1<x< $\infty$ : *)
upperBoundPositive[x_,alpha1_]:=Piecewise[
  {{1.4 alpha1 x-1.4 alpha1+1,1<x< $\infty$ },{ "",x<1}}
]
(* Lower bound function for the Adaptive Linearized Sigmoidal (ALiSA),1<x< $\infty$ : *)
lowerBoundPositive[x_,alpha1_]:=Piecewise[
  {{0.7 alpha1 x-0.7 alpha1+1,1<x< $\infty$ },{ "",x<1}}
]
(* Upper bound function for the Adaptive Linearized Sigmoidal (ALiSA),- $\infty$ <x<0: *)
upperBoundNegative[x_,alpha2_]:=Piecewise[
  {{(1.5 alpha2) x,x<0},{ "",x>0}}
]
(* Lower bound function for the Adaptive Linearized Sigmoidal (ALiSA),- $\infty$ <x<0: *)
lowerBoundNegative[x_,alpha2_]:=Piecewise[
  {{(0.7 alpha2) x,x<0},{ "",x>0}}
]
(* Create Manipulate for interactive exploration: *)
Manipulate[
  Plot[
    {
      adaptiveLinearizedSigmoid[x,alpha1,alpha2],
      upperBoundPositive[x,alpha1],
      lowerBoundPositive[x,alpha1],
      upperBoundNegative[x,alpha2],
      lowerBoundNegative[x,alpha2]
    },
    {x,-4,4}
  ],
  {{alpha1,1.5,"Alpha"},0.5,2,1},
  {{alpha2,0.7,"a"},-1,1,1},
  {{Seed,1,"Seed"},1,10,1}
]

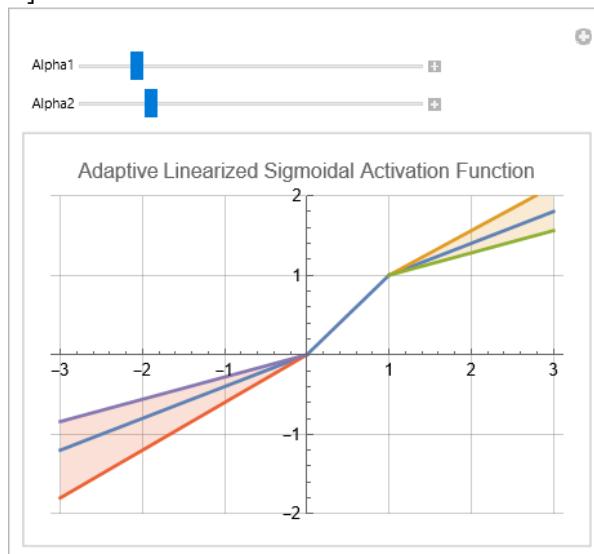
```

```

    upperBoundNegative[x,alpha2],
    lowerBoundNegative[x,alpha2]
  },
  {x,-3,3},
  Filling->{2->{3},4->{5}},
  PlotRange->{-2,2},
  PlotLabel->"Adaptive Linearized Sigmoidal Activation Function",
  ImageSize->300,
  GridLines->Automatic,
  PlotStyle->Thick
],
(* Slider for alpha1 parameter: *)
{{alpha1,0.4,"Alpha1"},0.1,2,0.1},
(* Slider for alpha2 parameter: *)
{{alpha2,0.4,"Alpha2"},0,2,0.1}
]

```

Output

**Mathematica Code 9.15**

Input (* The code defines the Rectified Linear Tanh (ReLTanh) activation function and its derivative, along with the standard Tanh function and its derivative. It creates an interactive interface using Mathematica's `Manipulate` function, allowing users to dynamically adjust the threshold parameters λ_{plus} and λ_{minus} via sliders. The plots, which update in real-time, display both the ReLTanh and Tanh functions and their derivatives over a specified range, with filled areas highlighting the differences: *)

```

(* Define the ReLTanh activation function: *)
reLTanh[x_,λplus_,λminus_]:=Piecewise[
{
  (* Right linear region: *)
  {Tanh'[λplus] (x-λplus)+Tanh[λplus],x>=λplus},
  (* Hyperbolic tangent region: *)
  {Tanh[x],λminus<x<λplus},
  (* Left linear region:*)
  {Tanh'[λminus] (x-λminus)+Tanh[λminus],x<=λminus}
}
]
(*Define the derivative of the ReLTanh activation function*)
reLTanhDerivative[x_,λplus_,λminus_]:=Piecewise[
{
  (* Right linear region derivative: *)

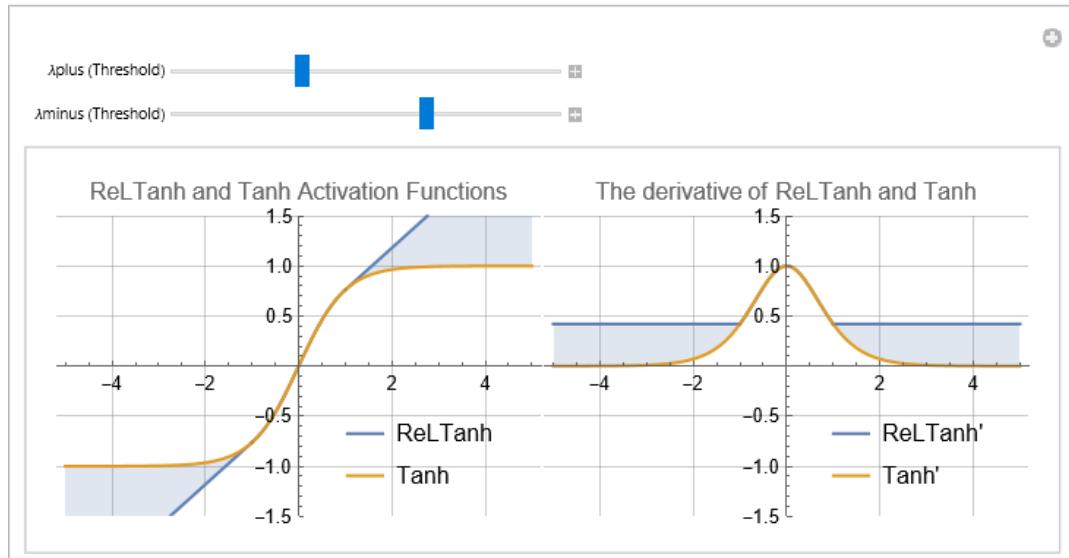
```

```

{Tanh'[\lambdaplus],x>=\lambdaplus},
(* Hyperbolic tangent region derivative: *)
{Sech[x]^2,\lambdaminus<x<\lambdaplus},
(* Left linear region derivative: *)
{Tanh'[\lambdaminus],x<=\lambdaminus}
}
]
(* Create Manipulate for interactive exploration: *)
Manipulate[
Row[
{
Plot[
{reLTanh[x,\lambdaplus,\lambdaminus],Tanh[x]},
{x,-5,5},
PlotRange->{-1.5,1.5},
Filling->{1->{2}},
PlotLabel->"ReLTanh and Tanh Activation Functions",
PlotLegends->Placed[{"ReLTanh","Tanh"},{0.75,0.2}],
GridLines->Automatic,
ImageSize->250
],
Plot[
{reLTanhDerivative[x,\lambdaplus,\lambdaminus],1-Tanh[x]^2},
{x,-5,5},
PlotRange->{-1.5,1.5},
Filling->{1->{2}},
PlotLabel->"The derivative of ReLTanh and Tanh",
PlotLegends->Placed[{"ReLTanh'","Tanh'"},{0.75,0.2}],
GridLines->Automatic,
ImageSize->250
]
}
],
(* Slider for \lambdaplus parameter: *)
{{\lambdaplus,1,"\lambdaplus (Threshold)"},0,3,0.1},
(* Slider for \lambdaminus parameter: *)
{{\lambdaminus,-1,"\lambdaminus (Threshold)"},-3,0,0.1}
]
]

```

Output



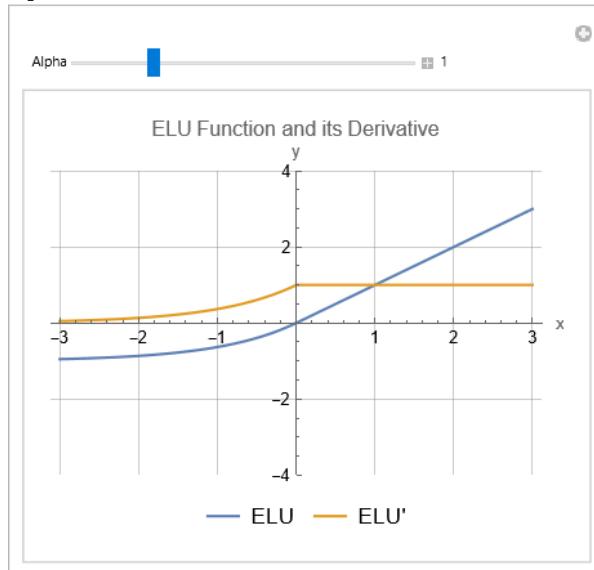
Mathematica Code 9.16

Input

```
(* The code defines the Exponential Linear Unit (ELU) activation function and its derivative, and creates an interactive interface using Mathematica's Manipulate function. This interface allows users to dynamically adjust the  $\alpha$  parameter via a slider. The plot, which updates in real-time, displays both the ELU function and its derivative over a specified range. This interactive tool helps users explore the impact of the  $\alpha$  parameter on the ELU function: *)
(* Define the ELU (Exponential Linear Unit) activation function: *)
eluFunction[x_,alpha_:1]:=If[x>=0,x,alpha*(Exp[x]-1)]

(* Define the derivative of the ELU activation function: *)
eluDerivative[x_,alpha_:1]:=If[x>=0,1,eluFunction[x,alpha]+alpha]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the ELU function and its derivative: *)
  Plot[
    {eluFunction[x,alpha],eluDerivative[x,alpha]},
    {x,-3,3},
    PlotRange->{-4,4},
    AxesLabel->{"x","y"},
    PlotLabel->"ELU Function and its Derivative",
    PlotLegends->Placed[{"ELU","ELU'"},Below],
    ImageSize->300 ,
    GridLines->Automatic
  ],
  (* Slider to adjust the alpha parameter: *)
  {{alpha,1,"Alpha"},0.1,4,Appearance->"Labeled"}
]
```

Output**Mathematica Code 9.17**

Input

```
(* The code defines the ReLU, ELU (Exponential Linear Unit), PReLU (Parametric ReLU), and sReLU (Shifted ReLU) activation functions along with their derivatives, and creates an interactive interface using Mathematica's Manipulate function. This interface allows users to dynamically adjust the alpha parameters for the ELU and PReLU functions via sliders. The plot, which updates in real-time, displays the four activation functions over a specified range, enabling users to compare them and observe the impact of the alpha parameters on the ELU and PReLU functions: *)
(* Define the ReLU activation function: *)
reluFunction[x_]:=Max[0,x]

(* Define the ELU activation function: *)
eluFunction[x_,alpha_:1]:=If[x>=0,x,alpha*(Exp[x]-1)]

(* Define the PReLU activation function: *)
preluFunction[x_,alpha_:1]:=If[x>=0,x,If[x<0,0,alpha*x]]

(* Define the sReLU activation function: *)
sreluFunction[x_,alpha_:1]:=If[x>=0,x,If[x<0,alpha*(x+1)]]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the four activation functions: *)
  Plot[
    {reluFunction[x],eluFunction[x,alpha],preluFunction[x,alpha],sreluFunction[x,alpha]},
    {x,-3,3},
    PlotRange->{-4,4},
    AxesLabel->{"x","y"},
    PlotLabel->"Four Activation Functions: ReLU, ELU, PReLU, and sReLU",
    PlotLegends->Placed[{"ReLU","ELU","PReLU","sReLU"},Below],
    ImageSize->300 ,
    GridLines->Automatic
  ],
  (* Sliders to adjust the alpha parameters: *)
  {{alpha,1,"Alpha"},0.1,4,Appearance->"Labeled"}]
```

```
(* ReLU function and its derivative: *)
relu[x_]:=Max[0,x]
reluDerivative[x_]:=If[x>=0,1,0]

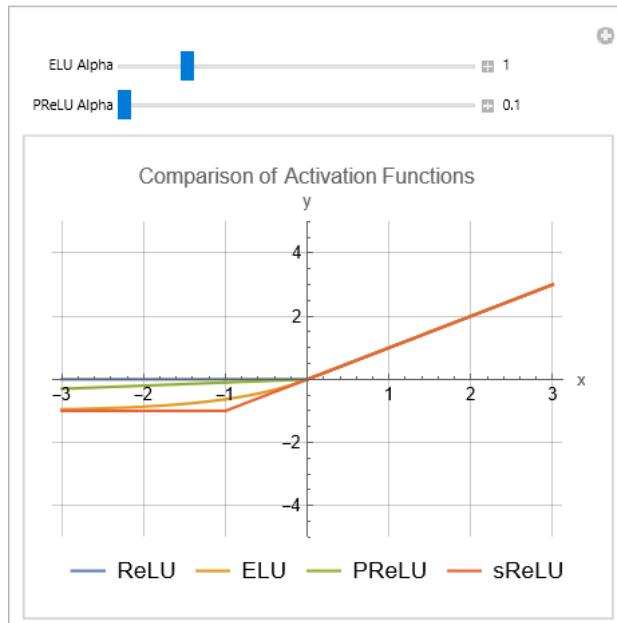
(* ELU (Exponential Linear Unit) function and its derivative: *)
elu[x_,alpha_:1]:=If[x>=0,x,Exp[x]-1]
eluDerivative[x_,alpha_:1]:=If[x>=0,1,elu[x,alpha]+alpha]

(* PReLU (Parametric ReLU) function and its derivative: *)
prelu[x_,alpha_:0.1]:=If[x>=0,x,alpha*x]
preluDerivative[x_,alpha_:0.1]:=If[x>=0,1,alpha]

(* Shifted ReLU function and its derivative: *)
srelu[x_]:=Max[-1,x]
sreluDerivative[x_]:=If[x>=-1,1,0]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the activation functions: *)
  Plot[
    {relu[x],elu[x,eluAlpha],prelu[x,preluAlpha],srelu[x]},
    {x,-3,3},
    PlotRange->{-5,5},
    AxesLabel->{"x","y"},
    PlotLabel->"Comparison of Activation Functions",
    PlotLegends->Placed[{"ReLU","ELU","PReLU","sReLU"},Below],
    ImageSize->300,
    GridLines->Automatic
  ],
  (* Slider to adjust the alpha parameter for the ELU function: *)
  {{eluAlpha,1,"ELU Alpha"},0.1,5,Appearance->"Labeled"}],
  (* Slider to adjust the alpha parameter for the PReLU function: *)
  {{preluAlpha,0.1,"PReLU Alpha"},0.1,5,Appearance->"Labeled"}]
]
```

Output



Mathematica Code 9.18

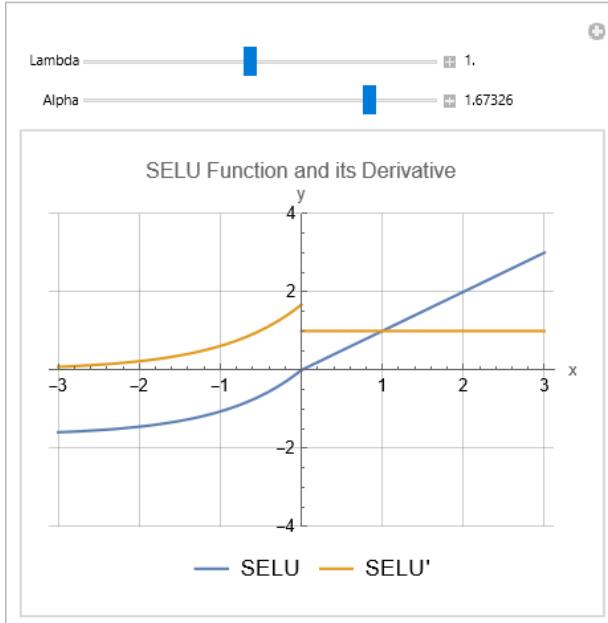
Input

```
(* The code defines the Scaled Exponential Linear Unit (SELU) activation function and its derivative, and creates an interactive interface using Mathematica's Manipulate function. This interface allows users to dynamically adjust the lambda ( $\lambda$ ) and alpha ( $\alpha$ ) parameters via sliders. The plot, which updates in real-time, displays both the SELU function and its derivative over a specified range: *)
```

```
(* Define the SELU (Scaled Exponential Linear Unit) activation function: *)
selu[x_, lambda_:1.0, alpha_:1.67326]:=lambda*If[x>0,x,alpha*(Exp[x]-1)]
```

```
(* Define the derivative of the SELU activation function: *)
seluDerivative[x_, lambda_:1.0, alpha_:1.67326]:=lambda*If[x>0,1,alpha*Exp[x]]
```

```
(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the SELU function and its derivative: *)
  Plot[
    {selu[x,lambda,alpha],seluDerivative[x,lambda,alpha]}, {x,-3,3},
    PlotRange->{-4,4}, AxesLabel->{"x", "y"}, PlotLabel->"SELU Function and its Derivative",
    PlotLegends->Placed[{"SELU", "SELU'"}, Below],
    ImageSize->300 , GridLines->Automatic
  ],
  (* Slider to adjust the lambda parameter: *)
  {{lambda,1.0,"Lambda"},0.1,2,Appearance->"Labeled"}, 
  (* Slider to adjust the alpha parameter: *)
  {{alpha,1.67326,"Alpha"},0.1,2,Appearance->"Labeled"}]
```

Output**Mathematica Code 9.19**

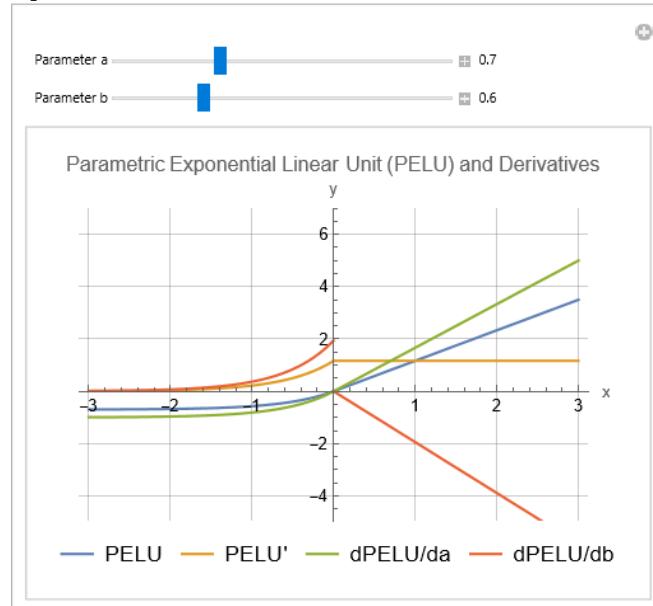
Input

```
(* The code defines the Parametric Exponential Linear Unit (PELU) activation function, its derivative with respect to x, and its partial derivatives with respect to parameters a and b. It creates an interactive interface using Mathematica's
```

Manipulate function, allowing users to dynamically adjust the values of a and b via sliders. The plot, which updates in real-time, displays the PELU function, its derivative, and its partial derivatives over a specified range: *)

```
(* Define the PELU activation function: *)
pelu[x_,a_:1,b_:1]:=If[x>=0,(a/b) x,a (Exp[x/b]-1)]
(* Define the derivative of the PELU activation function with respect to x: *)
peluDerivative[x_,a_:1,b_:1]:=If[x>=0,a/b,(a/b) Exp[x/b]]
(* Define the partial derivative of the PELU activation function with respect to
a: *)
peluPartialDerivativeA[x_,a_:1,b_:1]:=If[x>=0,1/b x,Exp[x/b]-1]
(* Define the partial derivative of the PELU activation function with respect to
b: *)
peluPartialDerivativeB[x_,a_:1,b_:1]:=If[x>=0,-a/b^2 x,a/b^2 Exp[x/b]]
(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the PELU function, its derivative, and its partial derivatives: *)
  Plot[
    {
      pelu[x,a,b],
      peluDerivative[x,a,b],
      peluPartialDerivativeA[x,a,b],
      peluPartialDerivativeB[x,a,b]
    },
    {x,-3,3},
    PlotRange->{-5,7},
    AxesLabel->{"x","y"},
    PlotLabel->"Parametric Exponential Linear Unit (PELU) and Derivatives",
    PlotLegends->Placed[{"PELU","PELU'","dPELU/da","dPELU/db"},Below],
    ImageSize->320 ,
    GridLines->Automatic
  ],
  (* Slider to adjust the parameter a: *)
  {{a,0.7,"Parameter a"},0.1,2,Appearance->"Labeled"},
  (* Slider to adjust the parameter b: *)
  {{b,0.6,"Parameter b"},0.1,2,Appearance->"Labeled"}
]
```

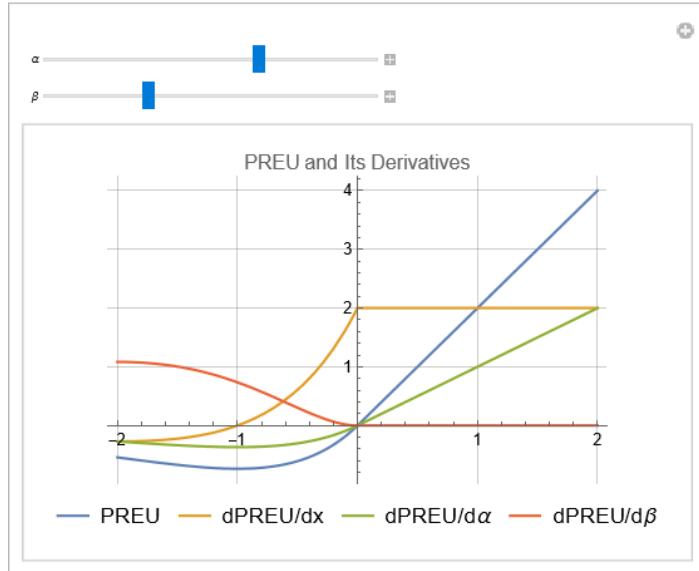
Output



Mathematica Code 9.20

Input

```
(* The code defines the Parametric Rectified Exponential Unit (PREU) activation function, its derivative with respect to x, and its partial derivatives with respect to parameters α and β. It creates an interactive interface using Mathematica's Manipulate function, allowing users to dynamically adjust the values of α and β via sliders. The plot, which updates in real-time, displays the PREU function, its derivative, and its partial derivatives over a specified range: *)
(* Define the PREU activation function: *)
PREU[x_,α_,β_]:=Piecewise[{{α*x,x>0},{α*x*Exp[β*x],x<=0}}]
(* Define the derivative of the PREU function with respect to x: *)
PREUDerivative[x_,α_,β_]:=D[PREU[t,α,β],t]/. t->x
(* Define the partial derivative of the PREU function with respect to α: *)
PREUAlphaDerivative[x_,α_,β_]:=D[PREU[x,s,β],s]/. s->α
(* Define the partial derivative of the PREU function with respect to β: *)
PREUBetaDerivative[x_,α_,β_]:=D[PREU[x,α,w],w]/. w->β
(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the PREU function and its derivatives: *)
  Plot[
    {
      PREU[x,α,β],
      PREUDerivative[x,α,β],
      PREUAlphaDerivative[x,α,β],
      PREUBetaDerivative[x,α,β]
    },
    {x,-2,2},
    PlotRange->All,
    PlotLegends->Placed[{"PREU", "dPREU/dx", "dPREU/dα", "dPREU/dβ"}, Below],
    PlotLabel->"PREU and Its Derivatives",
    ImageSize->300,
    GridLines->Automatic
  ],
  (* Slider to adjust the α parameter: *)
  {{α,2,"α"},0.1,3,0.1},
  (* Slider to adjust the β parameter: *)
  {{β,1,"β"},0.1,3,0.1}
]
```

Output

Mathematica Code 9.21

Input

```

(* The code defines the Elastic Exponential Linear Unit (EELU) activation function,
incorporating coefficient sampling during training where k=max[0,min[s,2]], 
s\[Tilde]N(1,\sigma), \sigma\[Tilde]U(0,\epsilon), and \epsilon\[Element](0,1]. It creates an interactive
interface using Mathematica's Manipulate function, allowing users to dynamically
adjust the values of \alpha, \beta, and the random seed via sliders. The plot, which updates
in real-time, displays the EELU function along with its lower and upper bounds over
a specified range: *)

(* Define the Elastic ELU (EELU) function: *)
EELU[x_,\alpha_,\beta_,k_]:=Piecewise[{{k*x,x>0},{\alpha*(Exp[\beta*x]-1),x<=0}}]

(* Define the coefficient sampling function: *)
sampleCoefficient[\sigma_]:=Module[
{s},
s=RandomVariate[NormalDistribution[1,\sigma]];
Max[0,Min[s,2]]
]

(* Create Manipulate for interactive exploration: *)
Manipulate[
Module[
{kValue},

(* Set the random seed for reproducibility: *)
SeedRandom[seed];

(* Sample \epsilon from a uniform distribution: *)
\epsilon=RandomReal[];

(* Sample \sigma from a uniform distribution bounded by \epsilon: *)
\sigma=RandomVariate[UniformDistribution[{0,\epsilon}]];

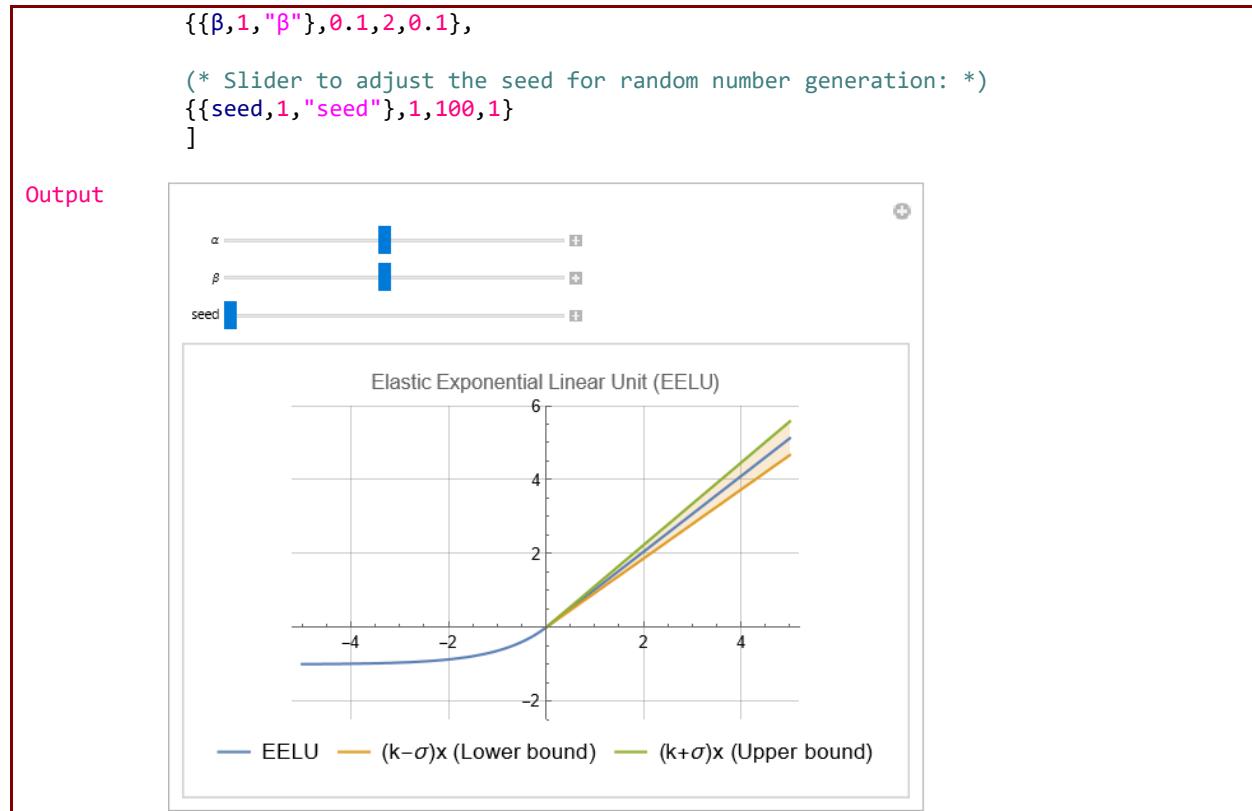
(* Compute the coefficient k based on the sampled \sigma and \epsilon: *)
kValue=sampleCoefficient[\sigma];

(* Define the lower bound function for the Elastic Parametric ReLU (EPReLU): *)
lowerBound[x_,\sigma_]:=Piecewise[{{{(kValue-\sigma) x,x>0},{"",x<0}}};

(* Define the upper bound function for the Elastic Parametric ReLU (EPReLU): *)
upperBound[x_,\sigma_]:=Piecewise[{{{(kValue+\sigma) x,x>0}, {"",x<0}}};

(* Plot the EELU function and its bounds: *)
Plot[
{EELU[x,\alpha,\beta,kValue],lowerBound[x,\sigma],upperBound[x,\sigma]},
{x,-5,5},
PlotRange->{-2.5,6},
(* Fill the area between the lower and upper bounds: *)
Filling->{2->{3}},
PlotLegends->Placed[{"EELU","(k-\sigma)x (Lower bound)","(k+\sigma)x (Upper
bound)"},Below],
PlotLabel->"Elastic Exponential Linear Unit (EELU)",
ImageSize->300 ,
GridLines->Automatic
],
],
(* Slider to adjust the \alpha parameter: *)
{\{\alpha,1,"\alpha"\},0.1,2,0.1},

(* Slider to adjust the \beta parameter: *)
]
]
]
```

**Mathematica Code 9.22**

Input

```

(* The code defines the Elastic Exponential Linear Unit (EELU) activation function,
its derivatives with respect to x, α, and β, and incorporates coefficient sampling
during training where k=max[0,min[s,2]], s~N(1,σ), σ~U(0,ε), and ε\[Element](0,1).
It creates an interactive interface using Mathematica's Manipulate function,
allowing users to dynamically adjust the values of α, β, and the random seed via
sliders. The plot, which updates in real-time, displays the EELU function, its
lower and upper bounds, and its derivatives over a specified range: *)

(* Define the Elastic ELU (EELU) function*)
EELU[x_, α_, β_, k_]:=Piecewise[{{k*x, x>0}, {α*(Exp[β*x]-1), x<=0}}]

(* Define the derivatives with respect to x, α, and β: *)
EELUDerivativeX[x_, α_, β_, k_]:=D[EELU[u, α, β, k], u]/. u->x
EELUDerivativeAlpha[x_, α_, β_, k_]:=D[EELU[x, α, β, k], α]/. α->α
EELUDerivativeBeta[x_, α_, β_, k_]:=D[EELU[x, α, β, k], β]/. β->β

(* Define the coefficient sampling function: *)
sampleCoefficient[σ_]:=Module[
{s},
s=RandomVariate[NormalDistribution[1,σ]];
Max[0,Min[s,2]]
]

(* Create Manipulate for interactive exploration: *)
Manipulate[
Module[
{kValue},
(* Set the random seed for reproducibility: *)

```

```

SeedRandom[seed];

(* Sample ε from a uniform distribution: *)
ε=RandomReal[];

(* Sample σ from a uniform distribution bounded by ε: *)
σ=RandomVariate[UniformDistribution[{0,ε}]];

(* Compute the coefficient k based on the sampled σ and ε: *)
kValue=sampleCoefficient[σ];

(* Define the lower bound function for the Elastic Parametric ReLU (EPReLU): *)
lowerBound[x_,σ_]:=Piecewise[{{(kValue-σ) x,x>0}, {"",x<0}}];

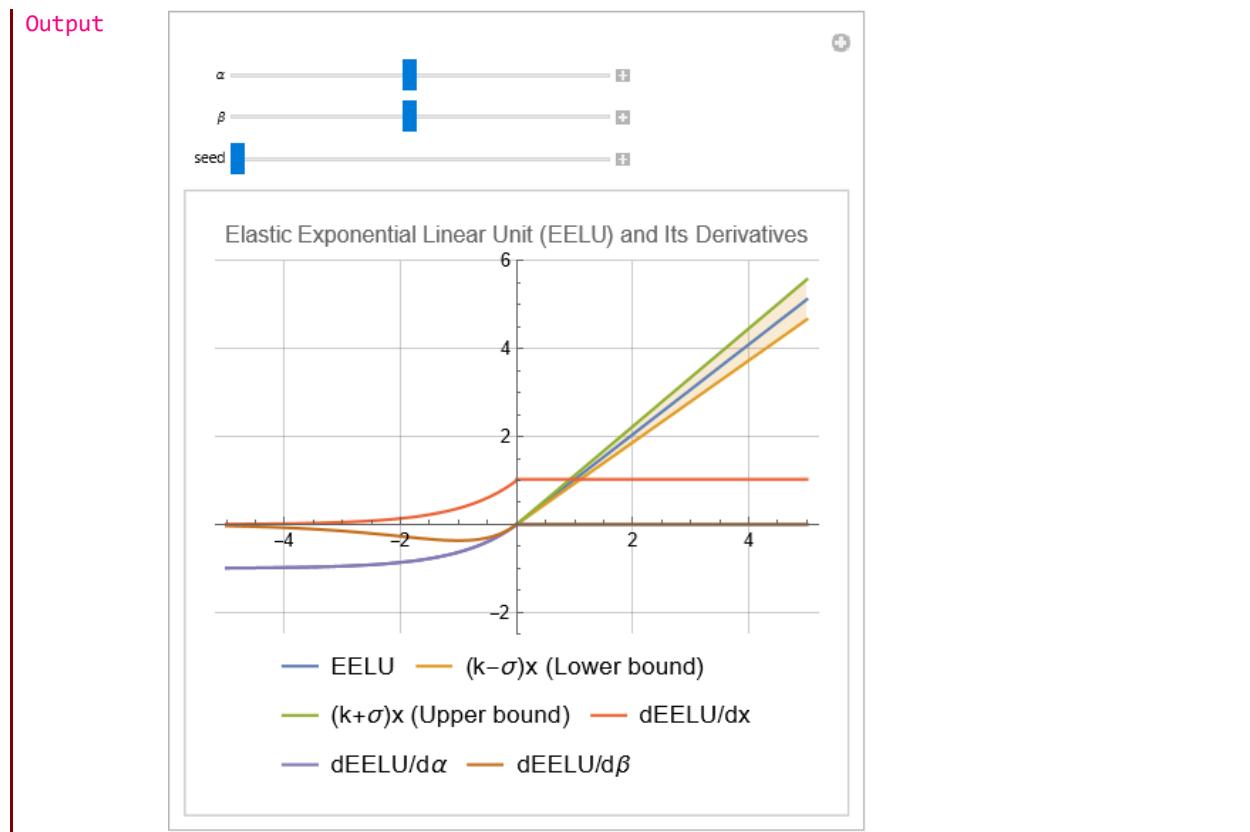
(* Define the upper bound function for the Elastic Parametric ReLU (EPReLU): *)
upperBound[x_,σ_]:=Piecewise[{{(kValue+σ) x,x>0}, {"",x<0}}];

(* Plot the EELU function, its bounds, and its derivatives: *)
Plot[
 {
 EELU[x,α,β,kValue],
 lowerBound[x,σ],
 upperBound[x,σ],
 EELUDerivativeX[x,α,β,kValue],
 EELUDerivativeAlpha[x,α,β,kValue],
 EELUDerivativeBeta[x,α,β,kValue]
 },
 {x,-5,5},
 PlotRange->{-2.5,6},
 (*Fill the area between the lower and upper bounds: *)
 Filling->{2->{3}},
 PlotLegends->Placed[
 {
 "EELU",
 "(k-σ)x (Lower bound)",
 "(k+σ)x (Upper bound)",
 "dEELU/dx",
 "dEELU/dα",
 "dEELU/dβ"
 },Below],
 PlotLabel->"Elastic Exponential Linear Unit (EELU) and Its Derivatives",
 ImageSize->320 ,
 GridLines->Automatic
 ]
 ],
 (* Slider to adjust the α parameter: *)
 {{α,1,"α"},0.1,2,0.1},

 (* Slider to adjust the β parameter: *)
 {{β,1,"β"},0.1,2,0.1},

 (* Slider to adjust the seed for random number generation: *)
 {{seed,1,"seed"},1,100,1}
]

```

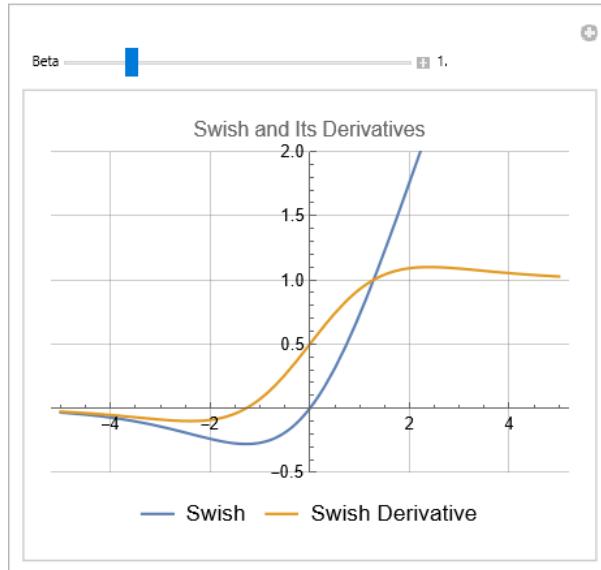
**Mathematica Code 9.23**

Input (* The code defines the Swish activation function and its derivative, and creates an interactive interface using Mathematica's Manipulate function. This interface allows users to dynamically adjust the β parameter via a slider. The plot, which updates in real-time, displays both the Swish function and its derivative over a specified range: *)

```
Manipulate[
Module[
{swishFun, swishDerivativeFun},
(* Define the Swish function: *)
swishFun[x_]:=x*LogisticSigmoid[\[Beta]*x];
(* Define the derivative of the Swish function: *)
swishDerivativeFun[x_]:=Module[
{s},
s=LogisticSigmoid[\[Beta]*x];
(*Compute the derivative*)
s+x*\[Beta]*s*(1-s)
];
(* Plot the Swish function and its derivative: *)
Plot[
{swishFun[x],swishDerivativeFun[x]},
{x,-5,5},
PlotLegends->Placed[{"Swish","Swish Derivative"},Below],
PlotRange->{-0.5,2},
PlotLabel->"Swish and Its Derivatives",
ImageSize->300,
GridLines->Automatic
],
],
```

```
(* Slider to adjust the β parameter: *)
{{β, 1.0, "Beta"}, 0.1, 5.0, 0.1, Appearance -> "Labeled"}
```

Output

**Mathematica Code 9.24**

Input (* The code defines the Exponential Swish (ESwish) activation function and its derivative, and creates an interactive interface using Mathematica's `Manipulate` function. This interface allows users to dynamically adjust the beta (β) parameter via a slider. The plot, which updates in real-time, displays both the ESwish function and its derivative over a specified range: *)

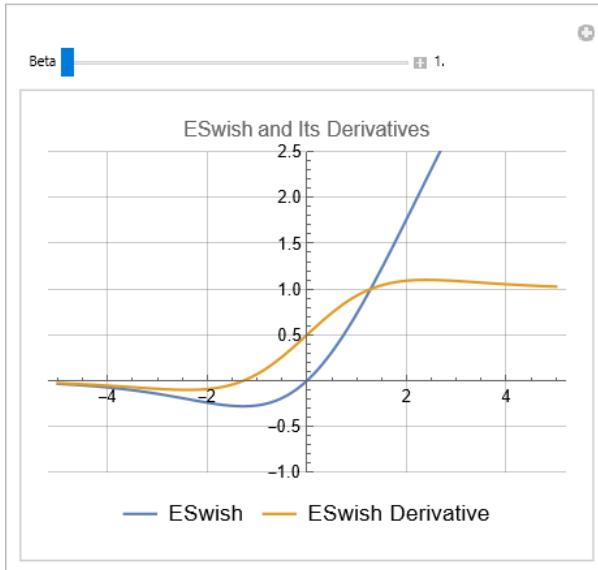
```
Manipulate[
 Module[
 {eswish, eswishDerivative},

 (* Define the ESwish function: *)
 eswish[x_, β_] := β*x*LogisticSigmoid[x];

 (* Define the derivative of the ESwish function: *)
 eswishDerivative[x_, β_] := eswish[x, β] + LogisticSigmoid[x]*(β - eswish[x, β]);

 (* Plot the ESwish function and its derivative: *)
 Plot[
 {eswish[x, β], eswishDerivative[x, β]},
 {x, -5, 5},
 PlotLegends -> Placed[{"ESwish", "ESwish Derivative"}, Below],
 PlotLabel -> "ESwish and Its Derivatives",
 PlotRange -> {-1, 2.5},
 ImageSize -> 300,
 GridLines -> Automatic
 ]
 ],
 (* Slider to adjust the β parameter: *)
 {{β, 1.0, "Beta"}, 1, 2, 0.1, Appearance -> "Labeled"}]
```

Output



Mathematica Code 9.25

```

Input (* The code defines the Exponential Linear Unit (ELU), Rectified Linear Unit (ReLU),
       Swish, and Exponential Swish (ESwish) activation functions, and creates an
       interactive interface using Mathematica's Manipulate function. This interface
       allows users to dynamically adjust the  $\alpha$  and  $\beta$  parameters via sliders. The plot,
       which updates in real-time, displays the ELU, ReLU, Swish, and ESwish functions
       over a specified range: *)

(* Define the ELU function: *)
ELU[x_, $\alpha$ _]:=Piecewise[{{x,x>0},{ $\alpha$ (Exp[x]-1),x<=0}}]

(* Define the ReLU function: *)
ReLU[x_]:=Max[0,x]

(* Define the Swish function: *)
swish[x_, $\beta$ _]:=x*LogisticSigmoid[ $\beta$ *x]

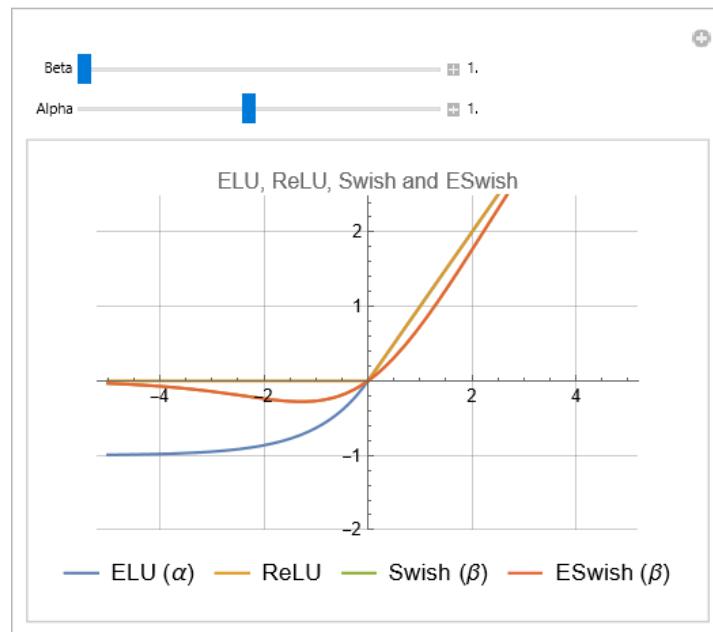
(* Define the ESwish function: *)
eswish[x_, $\beta$ _]:= $\beta$ *x*LogisticSigmoid[x]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the ELU, ReLU, Swish, and ESwish functions: *)
  Plot[
    {ELU[x, $\alpha$ ],ReLU[x],swish[x, $\beta$ ],eswish[x, $\beta$ ]},
    {x,-5,5},
    PlotLegends->Placed[{"ELU ( $\alpha$ )","ReLU","Swish ( $\beta$ )","ESwish ( $\beta$ )"},Below],
    PlotLabel->"ELU, ReLU, Swish and ESwish",
    PlotRange->{-2,2.5},
    ImageSize->300 ,
    GridLines->Automatic
  ],
  (* Slider to adjust the  $\beta$  parameter: *)
  {{ $\beta$ ,1.0,"Beta"},1,2,0.1,Appearance->"Labeled"},

  (* Slider to adjust the  $\alpha$  parameter: *)
  {{ $\alpha$ ,1.0,"Alpha"},0.1,2,0.1,Appearance->"Labeled"}
]

```

Output

**Mathematica Code 9.26**

```

Input      (* The code defines the Scaled Gaussian Error Linear Unit (SGELU) activation
           function with an adjustable  $\alpha$  parameter, along with the Gaussian Error Linear Unit
           (GELU), Rectified Linear Unit (ReLU), and Exponential Linear Unit (ELU) activation
           functions for comparison. It creates an interactive interface using Mathematica's
           Manipulate function, allowing users to dynamically adjust the  $\alpha$  parameter for the
           SGELU function via a slider. The plot, which updates in real-time, displays the
           SGELU, GELU, ReLU, and ELU functions over a specified range: *)

(* Define the SGELU function: *)
SGELU[x_,alpha_:1.702]:=alpha x Erf[x/Sqrt[2]]

(* Define the GELU function: *)
GELU[x_]:=x/2 (1+Erf[x/Sqrt[2]])

(* Define the ReLU function: *)
ReLU[x_]:=Max[0,x]

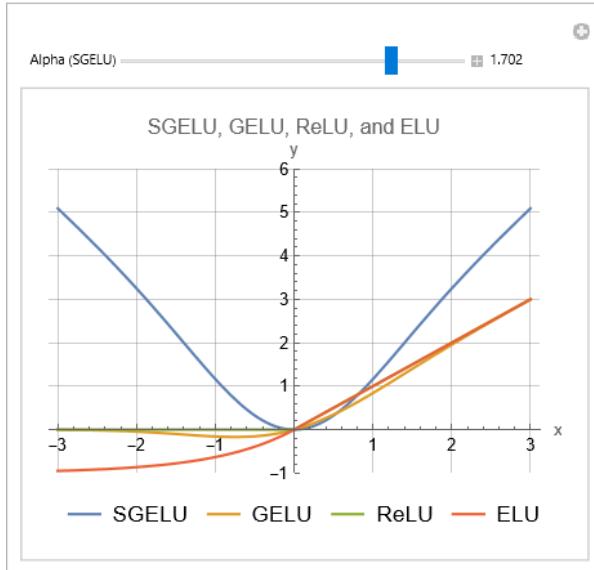
(* Define the ELU function: *)
ELU[x_,alpha_:1]:=Piecewise[{{alpha (Exp[x]-1),x<0},{x,x>=0}}]

(* Create Manipulate for interactive exploration: *)
Manipulate[
  (* Plot the SGELU, GELU, ReLU, and ELU functions: *)
  Plot[
    {SGELU[x,alpha],GELU[x],ReLU[x],ELU[x,1]},
    {x,-3,3},
    PlotLegends->Placed[{"SGELU","GELU","ReLU","ELU"},Below],
    PlotLabel->"SGELU, GELU, ReLU, and ELU",
    AxesLabel->{"x","y"},
    PlotRange->{-1,6},
    ImageSize->300 ,
    GridLines->Automatic
  ],
  (* Slider to adjust the alpha parameter for SGELU: *)
]

```

```
{\{alpha, 1.702, "Alpha (SGELU)"\}, 0.5, 2, Appearance -> "Labeled"},  
(* Track changes in the alpha parameter to update the plot in real-time: *)  
TrackedSymbols :> {\alpha}  
]
```

Output

**Mathematica Code 9.27**

Input (* The code defines the Maxout activation function, which takes the maximum of two linear functions $f_1(x) = w_1 \cdot x + b_1$ and $f_2(x) = w_2 \cdot x + b_2$ defined by parameters w_1 , b_1 , w_2 , and b_2 : $\text{Maxout}(x) = \max(f_1(x), f_2(x))$. This diagram is 2D and only shows how maxout behaves with a 1D input. It creates an interactive interface using Mathematica's Manipulate function, allowing users to dynamically adjust these parameters via sliders. The plot, which updates in real-time, displays the linear functions and the Maxout function over a specified range, and highlights specific points where the functions are evaluated. This interactive tool helps users explore the impact of the parameters on the Maxout function. By adjusting the sliders, users can see how changing the parameters affects the Maxout function and how it can be used to implement the ReLU, LReLU and PReLU functions, etc: *)

(* The Maxout function approximates the convex function formed by the maximum of $f_1(x)$ and $f_2(x)$. By adjusting the parameters w_1 , b_1 , w_2 , and b_2 , you can see how the Maxout unit approximates this convex function through different linear segments. The dashed lines represent the linear functions $f_1(x)$ and $f_2(x)$, respectively: *)

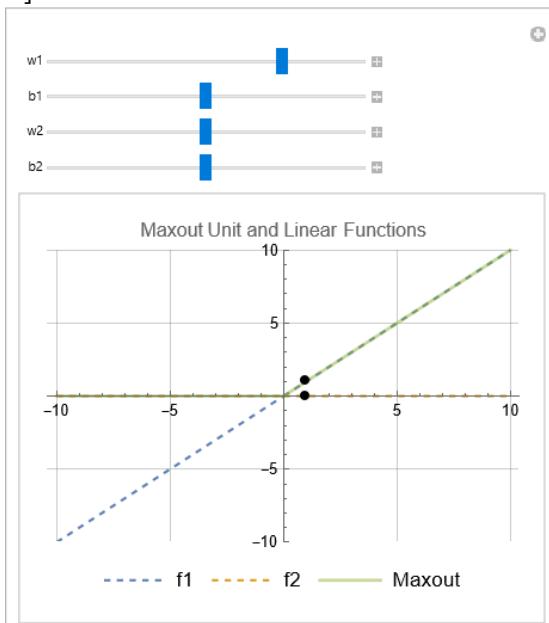
```
(* Define Maxout unit: *)  
maxout[x_, w1_, b1_, w2_, b2_] := Max[w1 x + b1, w2 x + b2]  
  
(* Define linear functions f1 and f2:*)  
f1[x_, w1_, b1_] := w1 x + b1  
f2[x_, w2_, b2_] := w2 x + b2  
  
(* Plot Maxout unit and linear functions: *)  
Manipulate[  
 Plot[  
 {f1[x, w1, b1], f2[x, w2, b2], maxout[x, w1, b1, w2, b2]},  
 {x, -10, 10},  
 PlotRange -> {-10, 10},
```

```

PlotStyle->{Dashed,Dashed,Directive[Opacity[0.5],Thick]},
PlotLegends->Placed[{"f1","f2","Maxout"},Below],
PlotLabel->"Maxout Unit and Linear Functions",
(* Highlight points on f1 and f2 at x=1*)
Epilog->{
  PointSize[0.02],
  Point[{{1,f1[1,w1,b1]},{1,f2[1,w2,b2]}}]
},
ImageSize->300,
GridLines->Automatic
],
(* Sliders to adjust the parameters of the linear functions: *)
{{w1,1,"w1"},-2,2},
{{b1,0,"b1"},-2,2},
{{w2,0,"w2"},-2,2},
{{b2,0,"b2"},-2,2}
]

```

Output

**Mathematica Code 9.28**

Input (* The code defines the Maxout activation function, which takes the maximum of three linear functions $f_1(x)=w_1 \cdot x + b_1$, $f_2(x)=w_2 \cdot x + b_2$, and $f_3(x)=w_3 \cdot x + b_3$ defined by parameters w_1 , b_1 , w_2 , b_2 , w_3 , and b_3 : $\text{Maxout}(x)=\max(f_1(x), f_2(x), f_3(x))$. This diagram is 2D and only shows how maxout behaves with a 1D input. The code creates an interactive interface using Mathematica's Manipulate function, allowing users to dynamically adjust these parameters via sliders. The plot, which updates in real-time, displays the three linear functions and the Maxout function over a specified range, and highlights specific points where the functions are evaluated. *)

(* In this code, you can adjust the parameters w_1 , b_1 , w_2 , b_2 , w_3 , and b_3 to see how the Maxout unit approximates the convex function formed by the maximum of $f_1(x)$, $f_2(x)$ and $f_3(x)$. The dashed lines represent the linear functions $f_1(x)$, $f_2(x)$ and $f_3(x)$, respectively: *)

```
(* Define Maxout unit: *)
maxout[x_,w1_,b1_,w2_,b2_,w3_,b3_]:=Max[w1 x+b1,w2 x+b2,w3 x+b3]
```

```
(* Define linear functions f1, f2 and f3: *)
```

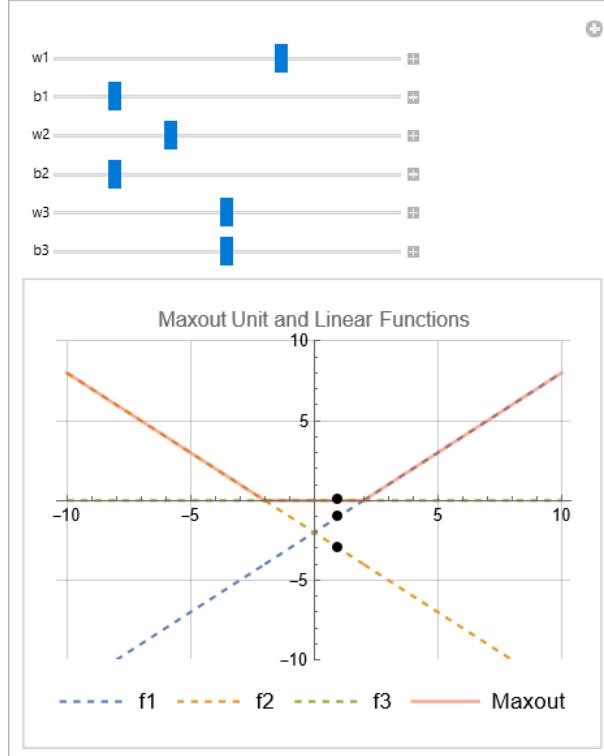
```

f1[x_,w1_,b1_]:=w1 x+b1
f2[x_,w2_,b2_]:=w2 x+b2
f3[x_,w3_,b3_]:=w3 x+b3

(* Plot Maxout unit and linear functions: *)
Manipulate[
  Plot[
    {f1[x,w1,b1],f2[x,w2,b2],f3[x,w3,b3],maxout[x,w1,b1,w2,b2,w3,b3]},
    {x,-10,10},
    PlotRange->{-10,10},
    PlotStyle->{Dashed,Dashed,Dashed,Directive[Opacity[0.5],Thick]},
    PlotLegends->Placed[{"f1","f2","f3","Maxout"},Below],
    PlotLabel->"Maxout Unit and Linear Functions",
    Epilog->{
      PointSize[0.02],
      Point[{{1,f1[1,w1,b1]},{1,f2[1,w2,b2]},{1,f3[1,w3,b3]}]}
    },
    ImageSize->300,
    GridLines->Automatic
  ],
  (* Sliders to adjust the parameters of the linear functions: *)
  {{w1,1,"w1"},-3,3},
  {{b1,-2,"b1"},-3,3},
  {{w2,-1,"w2"},-3,3},
  {{b2,-2,"b2"},-3,3},
  {{w3,0,"w3"},-3,3},
  {{b3,0,"b3"},-3,3}
]

```

Output

**Mathematica Code 9.29**

Input (* The code defines the softmax function, which converts a vector of inputs into a probability distribution. It creates an interactive interface using Mathematica's Manipulate function, allowing users to adjust the input values x, y, and z via

sliders. The interface displays the inputs and their corresponding softmax probabilities in a grid, visualizes the probabilities using a bar chart, and plots the softmax activation function over a specified range: *)

```

(* Define the softmax function: The softmax function converts a vector of values
into a vector of probabilities, where each probability is proportional to the
exponent of the corresponding input value: *)
softmax[inputs_]:=Module[
  {expInputs,expSum},
  (* Calculate the exponent of each input: *)
  expInputs=Exp[inputs];
  (* Sum of all exponentiated inputs: *)
  expSum=Total[expInputs];
  (* Normalize by dividing each exponentiated input by the sum: *)
  expInputs/expSum
]

(* Create an interactive Manipulate interface to adjust the input values and display
the softmax probabilities: *)
Manipulate[
  Module[
    {inputs,probabilities},
    (* Define the input vector: *)
    inputs={x,y,z};
    (* Calculate the softmax probabilities: *)
    probabilities=softmax[inputs];
    Column[
      {
        (*Display the inputs and their corresponding softmax probabilities in a grid:*)
        Grid[
          {
            {"Inputs","Softmax Probabilities"},
            {inputs,probabilities}
          }
        ],
        (* Visualize the softmax probabilities using a bar chart: *)
        BarChart[
          probabilities,
          ChartLabels->Range[3],
          PlotRange->{0,1},
          ChartStyle->"Rainbow",
          AxesLabel->{"Class","Probability"},
          PlotLabel->"Softmax Probabilities",
          ImageSize->300
        ],
        (* Plot the softmax activation function:*)
        Plot[
          Evaluate[softmax[{x,y,z}]],
          {x,-3,3},
          PlotRange->{{{-3,3},{0,1}}},
          PlotStyle->{Purple,Green,Red},
          PlotLabel->"Softmax Activation Function",
          ImageSize->300 ,
          GridLines->Automatic
        ]
      }
    ],
    (* Sliders to adjust the input values x, y, and z: *)
    {{x,1,"Input x"},-3,3,Appearance->"Labeled"},
    {{y,0,"Input y"},-3,3,Appearance->"Labeled"}
  ]
]

```

**Mathematica Code 9.30**

Input (* The code defines the Leaky Rectified Linear Unit (LReLU) and Exponential Linear Unit (ELU) activation functions, and creates a mixed activation function that combines these two functions using a weighted sum controlled by the parameter Rho (ρ). It also computes the derivative of the mixed activation function. Using Mathematica's Manipulate function, the code creates an interactive interface allowing users to adjust the parameters ρ , α , and β via sliders. The interface generates example input data, applies the mixed activation function and its derivative to this data, and plots the results: *)

```
(* Define the LReLU function with parameter alpha: *)
lreluFunction[x_, alpha_] := If[x > 0, x, alpha*x]

(* Define the Exponential Linear Unit (ELU) function with parameter beta:*)
eluFunction[x_, beta_] := If[x > 0, x, beta*(Exp[x] - 1)]

(* Define the mixed activation function: Combine LReLU and ELU using a weighted sum controlled by parameter Rho *)
mixedActivation[x_, Rho_, alpha_, beta_] := Rho*lreluFunction[x, alpha] + (1 - Rho)*eluFunction[x, beta]

(* Compute the derivative of mixedActivation with respect to x: *)
```

```

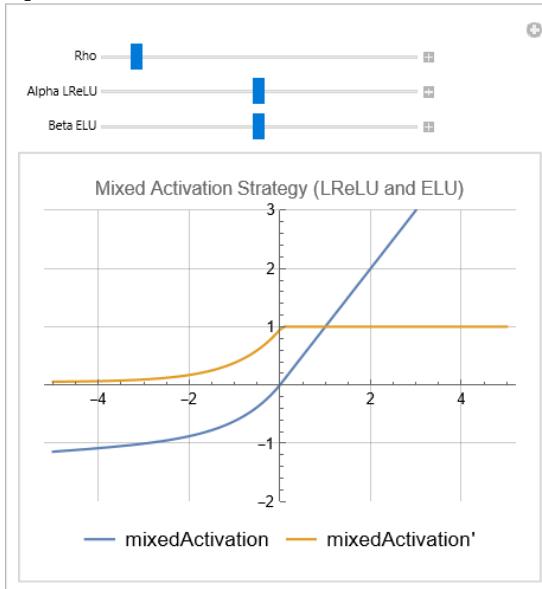
derivativeMixedActivation[x_, Rho_, alpha_, beta_] := D[mixedActivation[t, Rho, alpha, be
ta], t] /. t -> x

(* Generate some example data: *)
data = Range[-5, 5, 0.1];
n = Length[data];

(* Create an interactive Manipulate interface to adjust parameters and visualize
the results: *)
Manipulate[
Module[
{outputData, outputDataderivative},
(* Apply the mixed activation strategy to the data: *)
outputData = Table[
  mixedActivation[data[[i]], Rho, alpha, beta],
  {i, 1, n}
];
(* Compute the derivative of the mixed activation function for the input data: *)
]
outputDataderivative = Table[
  derivativeMixedActivation[data[[i]], Rho, alpha, beta],
  {i, 1, n}
];
(* Plot the mixed activation function and its derivative: *)
ListLinePlot[
{Transpose[{data, outputData}], Transpose[{data, outputDataderivative}]},
PlotRange -> {-2, 3},
PlotLegends -> Placed[{"mixedActivation", "mixedActivation'"}, Below],
PlotLabel -> "Mixed Activation Strategy (LReLU and ELU)",
ImageSize -> 300,
GridLines -> Automatic
]
],
(* Sliders to adjust the parameters Rho, alpha, and beta: *)
{{Rho, 0.1, "Rho"}, 0, 1, 0.01},
{{alpha, 0.5, "Alpha LReLU"}, 0, 1, 0.01},
{{beta, 1, "Beta ELU"}, 0, 2, 0.01}
]
]

```

Output



Mathematica Code 9.31

Input

```
(* The code defines the Leaky Rectified Linear Unit (LReLU) and Exponential Linear Unit (ELU) activation functions, and creates a gated activation function that combines these two functions using a gating mechanism controlled by the parameter omega (w). It also computes the derivative of the gated activation function. Using Mathematica's Manipulate function, the code creates an interactive interface allowing users to adjust the parameters w, α, and β via sliders. The interface generates example input data, applies the gated activation function and its derivative to this data, and plots the results: *)

(* Define the LReLU function with parameter alpha: *)
lreluFunction[x_,alpha_]:=If[x>0,x,alpha*x]

(* Define the ELU function with parameter beta: *)
eluFunction[x_,beta_]:=If[x>0,x,beta*(Exp[x]-1)]

(* Define the gated activation function: *)
gatedActivation[x_,omega_,alpha_,beta_]:=Module[
  {gatingMask,lreluResult,eluResult},
  (* Calculate the gating mask: *)
  gatingMask=LogisticSigmoid[omega*x];

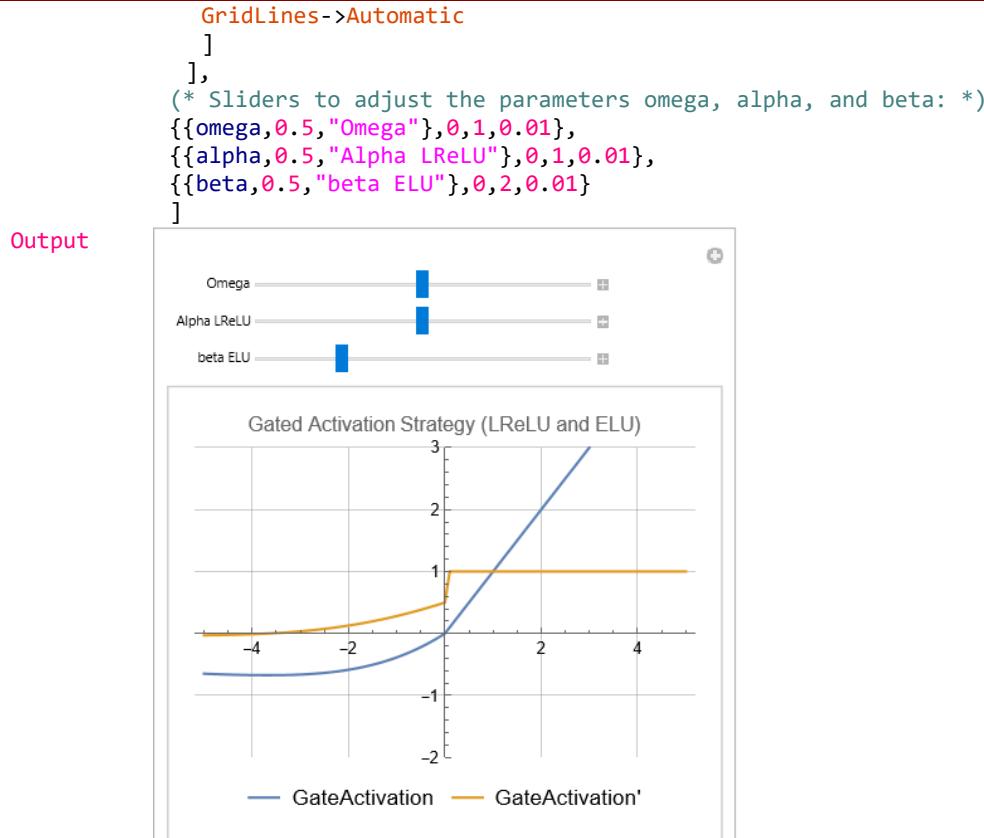
  lreluResult=lreluFunction[x,alpha];
  eluResult=eluFunction[x,beta];

  (* Combine the results using the gating mask: *)
  gatingMask*lreluResult+(1-gatingMask)*eluResult
];

(* Compute the derivative of GateActivation with respect to x: *)
derivativeGateActivation[x_,omega_,alpha_,beta_]:=D[gatedActivation[t,omega,alpha,beta],t]/. t->x

(* Generate some example data: *)
data=Range[-5,5,0.1];
n=Length[data];

(* Create an interactive Manipulate interface to adjust parameters and visualize the results: *)
Manipulate[
 Module[
  {outputData,outputGatedDerivative},
  (* Apply the gated activation strategy to the data: *)
  outputData=Table[
    gatedActivation[data[[i]],omega,alpha,beta],
    {i,1,n}
  ];
  (* Compute the derivative of the gated activation function for the input data: *)
  outputGatedDerivative=Table[
    derivativeGateActivation[data[[i]],omega,alpha,beta],
    {i,1,n}
  ];
  (* Plot the gated activation function and its derivative: *)
  ListLinePlot[
    {Transpose[{data,outputData}],Transpose[{data,outputGatedDerivative}]},
    PlotRange->{-2,3},
    PlotLabel->"Gated Activation Strategy (LReLU and ELU)",
    PlotLegends->Placed[{"GateActivation","GateActivation'"},Below],
    ImageSize->300,
  ]
]
```

**Mathematica Code 9.32**

Input (* The code defines the Adaptive Piecewise Linear (APL) activation function, which combines the Rectified Linear Unit (ReLU) with additional linear segments defined by parameters a and b. Using Mathematica's Manipulate function, the code creates an interactive interface allowing users to adjust these parameters via sliders. The interface generates a plot of the APL function over a specified range of input values, which updates in real-time as the parameters are adjusted: *)

```
(* Define the APL activation function: *)
APL[x_, a_, b_] := Module[
  {S, terms},
  (* Determine the number of segments: *)
  S = Length[a];
  (* Calculate each segment term: *)
  terms = Table[
    a[[s]]*Max[0, -x + b[[s]]],
    {s, S}
  ];
  (* Combine the terms with the ReLU component: *)
  Max[0, x] + Total[terms]
]

(* Create an interactive Manipulate interface to adjust parameters and visualize
the APL function: *)
Manipulate[
  Plot[
    APL[x, aValues, bValues],
    (* Plot the APL function: *)
    {x, -5, 5},
    GridLines -> Automatic
  ],
  (* Sliders to adjust the parameters omega, alpha, and beta: *)
  {{omega, 0.5, "Omega"}, 0, 1, 0.01},
  {{alpha, 0.5, "Alpha LReLU"}, 0, 1, 0.01},
  {{beta, 0.5, "beta ELU"}, 0, 2, 0.01}
]
```

```

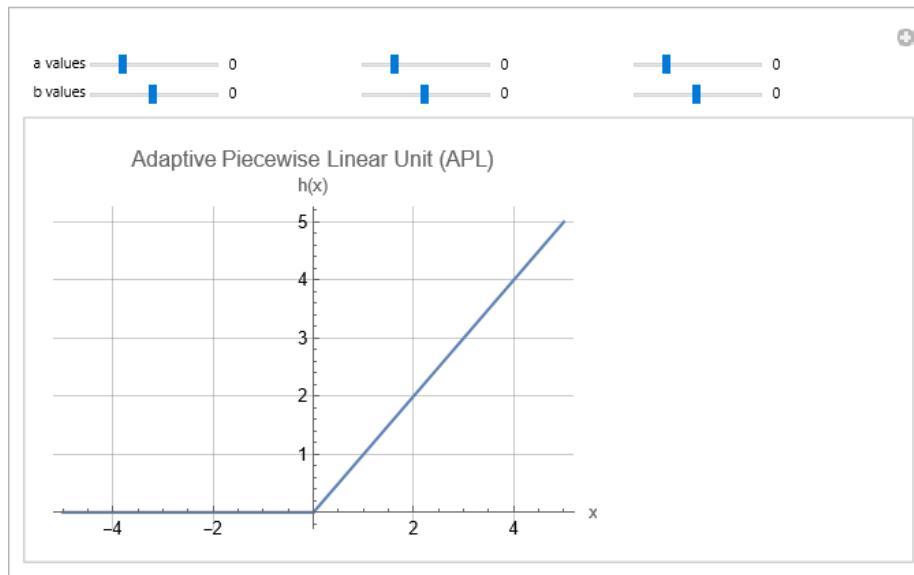
PlotRange->All,
AxesLabel->{"x","h(x)"},
PlotLabel->"Adaptive Piecewise Linear Unit (APL)",
ImageSize->300 ,
GridLines->Automatic
],
(* Sliders to adjust the parameters aValues and bValues: *)

(* aValues are the coefficients for the segments: *)
{{aValues,{0,0,0}, "a values"},{-1,-1,-1},{3,3,3},Appearance->"Labeled"},

(* bValues are the breakpoints for the segments: *)
{{bValues,{0,0,0}, "b values"},{-3,-3,-3},{3,3,3},Appearance->"Labeled"}
]

```

Output

**Mathematica Code 9.33**

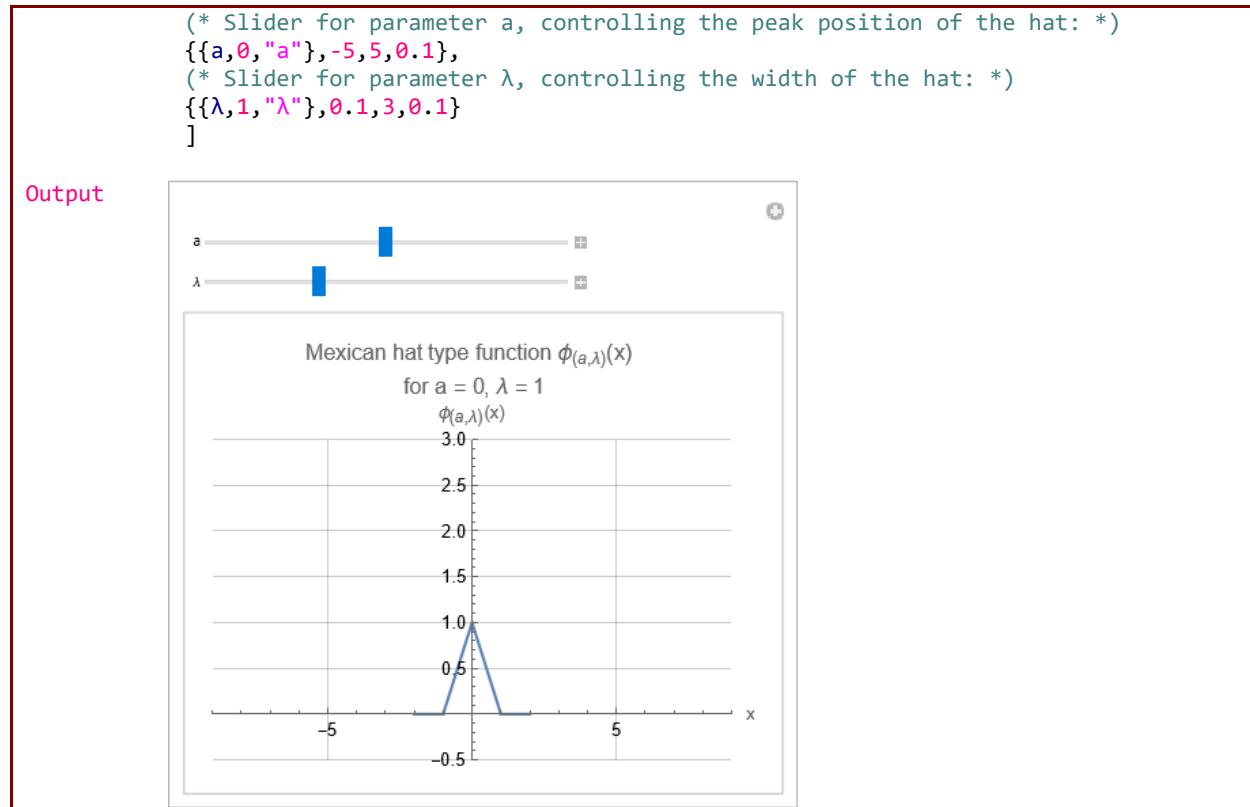
Input (* The code defines a Mexican hat type function, $\Phi(a,\lambda,x) = \max(\lambda - |x-a|, 0)$, and creates an interactive interface using Mathematica's Manipulate function. This interface allows users to adjust the parameters a and λ via sliders. The plot, which updates in real-time, displays the Mexican hat type function over a specified range of input values: *)

```

(* Define the Mexican hat type function: *)
 $\varphi[a_,\lambda_,x_]:=Max[\lambda-Abs[x-a],0]$ 

(* Create an interactive Manipulate interface to adjust parameters and visualize
the Mexican hat type function: *)
Manipulate[
(* Plot the Mexican hat type function: *)
Plot[
 $\varphi[a,\lambda,x]$ ,
{x,a-2 λ,a+2 λ},
PlotRange->{{-9,9},{-0.5,3}},
AxesLabel->{"x"," $\varphi(a,\lambda)(x)$ "},
PlotLabel->StringForm["Mexican hat type function  $\varphi(a,\lambda)(x)$  \n for a = `1`,  $\lambda$  = `2`",a,λ],
ImageSize->300,
GridLines->Automatic
],

```

**Mathematica Code 9.34**

Input (* The code defines a Mexican hat type function, a PReLU function, and a MeLU function that combines PReLU with a linear combination of Mexican hat type functions. This Manipulate allows you to interactively control the coefficients, centers, and widths of the MeLU Function, Mexican Hat Type functions, and PReLU updating the plot in real-time. Adjust the sliders to observe the changes in the MeLU function and its components. This code introduces a Random Seed slider (seed) that allows you to control the seed for the random number generation. Start with initial random values for the coefficients. These coefficients will be adjusted during the training process. The coefficients will be randomly generated (initialized) whenever the seed is changed (In practical applications, the coefficients in models like MeLU are typically learned from data using optimization techniques.): *)

```
(* Define Mexican Hat Type Function: *)
phi[a_, λ_][x_]:=Max[λ-Abs[x-a], 0]

(* Define MeLU Function: *)
MeLU[x_, a_, coefficients_, centers_, widths_]:=PReLU[x, a]+Sum[
  coefficients[[i]]* phi[centers[[i]], widths[[i]]][x],
  {i, Length[coefficients]}]
]

(* Define PReLU Function for illustration: *)
PReLU[x_, a_]:=Max[x, a x]

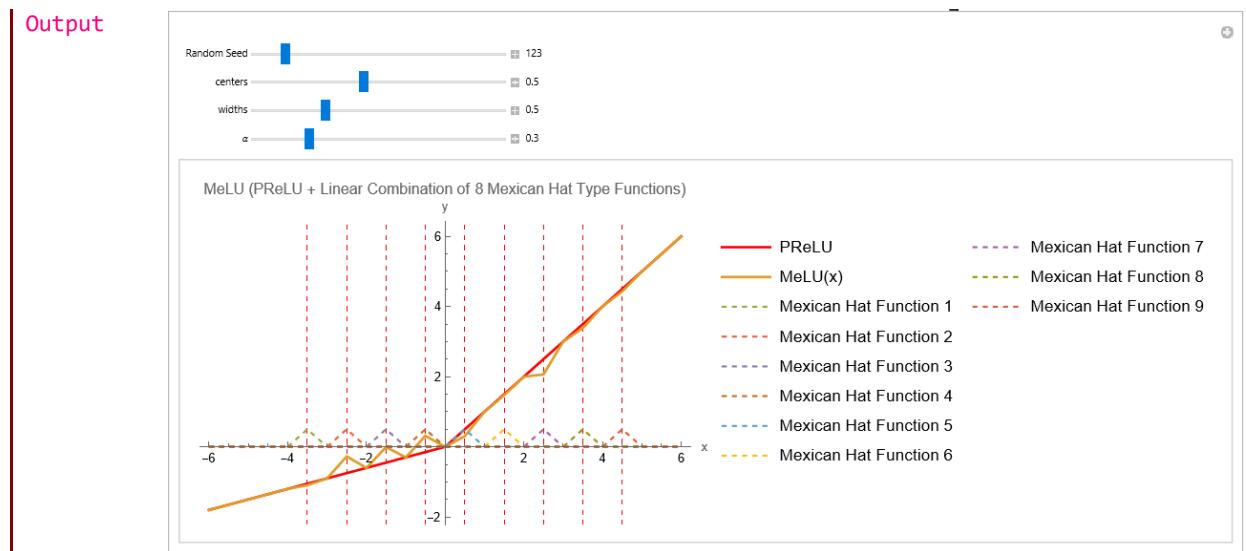
(* Create an interactive Manipulate interface to adjust parameters and visualize
the MeLU function and its components: *)
Manipulate[
  (* Set the random seed for reproducibility: *)
  SeedRandom[seed];
```

```
(* Generate random coefficients: *)
coefficients=RandomReal[{-1,1},9];

(* Define the widths for the Mexican Hat functions: *)
widths=Table[width,{i,9}];

(* Define the centers for the Mexican Hat functions: *)
centers=Table[i+shift,{i,-4,4,1}];

(* Plot the PReLU, MeLU and Mexican hat type functions: *)
Plot[
 {
  PReLU[x,a],
  MeLU[x,a,coefficients,centers,widths],
  Evaluate@Table[
   phi[[centers[[i]],widths[[i]]][x],
   {i,Length[centers]}]
   ]
 },
 {x,-6,6},
 PlotStyle->{
  Directive[Red,Thick],Thick,Dashed,Dashed,
  Dashed,Dashed,Dashed,Dashed,Dashed,Dashed
 },
 PlotLegends->{
  "PReLU",
  "MeLU(x)",
  "Mexican Hat Function 1",
  "Mexican Hat Function 2",
  "Mexican Hat Function 3",
  "Mexican Hat Function 4",
  "Mexican Hat Function 5",
  "Mexican Hat Function 6",
  "Mexican Hat Function 7",
  "Mexican Hat Function 8",
  "Mexican Hat Function 9"},
 AxesLabel->{"x","y"},
 PlotRange->All,
 PlotLabel->"MeLU (PReLU + Linear Combination of 8 Mexican Hat Type Functions)",
 GridLines->{centers,None},
 GridLineStyle->Directive[Red,Dashed],
 ImageSize->400
 ],
 (* Slider for random seed: *)
 {{seed,123,"Random Seed"},1,1000,1,Appearance->"Labeled"},
 (* Slider for shifting the centers: *)
 {{shift,0.5,"centers"},0.1,1,0.1,Appearance->"Labeled"},
 (* Slider for adjusting the widths: *)
 {{width,0.5,"widths"},0.1,1.5,0.1,Appearance->"Labeled"},
 (* Slider for  $\alpha$  in PReLU: *)
 {{a,0.3," $\alpha$ "},0.1,1,0.1,Appearance->"Labeled"},
 (* Track changes in these symbols: *)
 TrackedSymbols:>{seed,shift,width,a}
]
```

**Mathematica Code 9.35**

Input

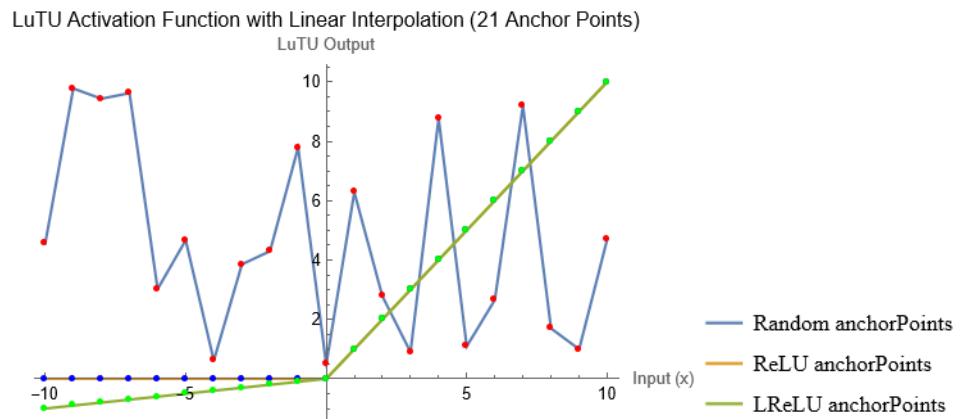
```
(* The LuTU activation function is defined using linear interpolation between
anchor points based on the specified definition ((1/s) (yi[[i]] (xi[[i+1]]-
x)+yi[[i+1]] (x-xi[[i]]))). The function is linearly interpolated between yi and
yi+1 for any input value x between xi and xi+1. The function is plotted for three
different sets of anchor points: random points, ReLU points, and LReLU points. The
plot shows how the LuTU function behaves based on these different anchor points.
The points are marked in red, blue, and green, respectively, on the plot: *)

(* Generate 21 anchor points: *)
(* Set random seed for reproducibility: *)
SeedRandom[123];
(* Random anchor points: *)
RandomAnchorPoints=Table[{i,RandomReal[{0,10}]],{i,-10,10}];
(* ReLU anchor points: *)
ReLUAnchorPoints=Table[{i,Max[0,i]},{i,-10,10}];
(* LReLU anchor points: *)
LReLUAnchorPoints=Table[{i,Max[0.1 i,i]},{i,-10,10}];

(* Define the LuTU activation function with linear interpolation: *)
LuTU[x_,anchorPoints_]:=Module[
{n,xi,yi,s,i},
(* Number of anchor points: *)
n=Length[anchorPoints];
(* x-coordinates of anchor points: *)
xi=anchorPoints[[All,1]];
(* y-coordinates of anchor points: *)
yi=anchorPoints[[All,2]];
(* Interval between anchor points: *)
s=xi[[2]]-xi[[1]];
(* Iterate through anchor points to find the interval containing x: *)
For[
i=1,i<n,i++,
If[
x>=xi[[i]]&&x<=xi[[i+1]],
(* Linearly interpolate between yi[i] and yi[i+1]: *)
Return[(1/s) (yi[[i]] (xi[[i+1]]-x)+yi[[i+1]] (x-xi[[i]]))]
]
];
];
```

```
(* If x is outside the range of anchor points, return 0 or some default value:
*)
Return[0];
]

(* Plot the LuTU activation function: *)
Plot[(* Plot LuTU for different anchor points: *)
{
  LuTU[x,RandomanchorPoints],
  LuTU[x,ReLUanchorPoints],
  LuTU[x,LReLUanchorPoints]
},
{x,-10,10},
PlotRange->All,
AxesLabel->{"Input (x)","LuTU Output"},
PlotLabel->"LuTU Activation Function with Linear Interpolation (21 Anchor Points)",
Epilog->
  Red,
  PointSize[0.01],
  Point[RandomanchorPoints],
  Blue,
  Point[ReLUanchorPoints],
  Green,
  Point[LReLUanchorPoints]
},
ImageSize->400,
PlotLegends->{"Random anchorPoints","ReLU anchorPoints","LReLU anchorPoints"}
]
```

Output**Mathematica Code 9.36**

Input (* The smoothing function is defined as $1/(2 \tau) (1 + \cos(\pi/\tau x))$ for $-\tau \leq x \leq \tau$, and 0 otherwise. The code also defines its derivative with respect to x. Using the Manipulate function, the parameter τ can be adjusted interactively via a slider. The plot, which updates in real-time, shows the smoothing function and its derivative, helping users explore the behavior of these functions with varying values of τ : *)

```
(* Define the smoothing function: *)
smoothingFunction[x_,\tau_]:=Piecewise[
 {
  {1/(2 \tau) (1+Cos[\pi/\tau x]),-\tau<=x<=\tau},
  {0,True}
}]
```

```

]

(* Define the derivative of the smoothing function: *)
derivativeFunction[x_,τ_]:=D[smoothingFunction[t,τ],t]/. t->x

(* Create an interactive Manipulate interface to adjust the parameter τ and
visualize the smoothing function and its derivative: *)
Manipulate[
(* Plot the smoothing function and its derivative: *)
Plot[
{smoothingFunction[x,τ],derivativeFunction[x,τ]},
{x,-2 τ,2 τ},
PlotRange->{{-2,2},{-7,7}},
AxesLabel->{"x","y"},
PlotLabel->StringForm["Smoothing Function and Its Derivative for τ = ``",τ],
PlotLegends->Placed[{"Smoothing Function","Derivative"},Below],
ImageSize->300,
GridLines->Automatic
],
(*Slider to adjust the parameter τ:*)
{{τ,1,"τ"},0.5,1.5,0.1,Appearance->"Labeled"}
]

```

Output

Mathematica Code 9.37

Input

```

(* The smoothed activation function is defined as the sum of individual terms, each
involving a learnable parameter yi and a cosine smoothing mask r(x-xi,t s). This
code visualizes the individual cosine smoothing functions, the scaled versions of
these functions, and the overall smoothed activation function. The anchor points
are marked on the plots to show their locations. By adjusting the parameters and
visualizing the results, one can explore the behavior of the smoothed activation
function and its components: *)

(* Define the Smoothing Function: This function applies a cosine smoothing within
the range -τ to τ. Outside this range, the function returns 0: *)
smoothingFunction[x_,τ_]:=If[-τ<=x<=τ,1/(2 τ) (1+Cos[(π/τ) x]),0]

(* Define the Smoothed Activation Function: This function computes the sum of
individual terms, each term is the product of a learnable parameter yi and the
smoothing mask r(x-xi,t s): *)

```

```

smoothedActivation[x_, anchorPoints_, t_, s_]:=Module[
{xi,yi},
(* Extract x-coordinates of anchor points: *)
xi=anchorPoints[[All,1]];
(* Extract y-coordinates of anchor points: *)
yi=anchorPoints[[All,2]];
Sum[
(* Compute each term of the sum: *)
anchorPoints[[i,2]]* smoothingFunction[x-anchorPoints[[i,1]],t,s],
{i,1,Length[xi]}
]
]

(* Example anchor points: Generate random y-values for anchor points spaced between-
10 and 10: *)
anchorPoints=Table[{i,RandomReal[{0,10}]}',{i,-10,10,1}];
(* Extract x-coordinates of anchor points: *)
xi=anchorPoints[[All,1]];

(* Plot individual cosine smoothing functions centered at each anchor point: *)
Plot[
Table[
smoothingFunction[x-anchorPoints[[i,1]],1],
{i,1,Length[xi]}
],
{x,-10,10},
PlotRange->All,
PlotLabel->"21 Cosine Smoothing Functions",
ImageSize->300
]

(* Plot individual scaled cosine smoothing functions: *)
Plot[
Table[
anchorPoints[[i,2]]* smoothingFunction[x-anchorPoints[[i,1]],1],
{i,1,Length[xi]}
],
{x,-10,10},
PlotRange->All,
PlotLabel->"21 Scaled Cosine Smoothing Functions",
Epilog->{Red,PointSize[0.01],Point[anchorPoints]},
ImageSize->300
]

(* Plot the smoothed activation function using the random anchor points: *)
Plot[
smoothedActivation[x,anchorPoints,1,1],
{x,-10,10},
PlotRange->All,
PlotLabel->"Smoothed Activation Function (21 Random Anchor Points)",
Epilog->{Red,PointSize[0.01],Point[anchorPoints]},
ImageSize->300
]

(* Plot the smoothed activation function along with individual cosine smoothing
functions and scaled cosine smoothing functions for comparison: *)
Plot[
{
(*Smoothed activation function*)
smoothedActivation[x,anchorPoints,1,1],
(*Individual cosine smoothing functions*)
}
]

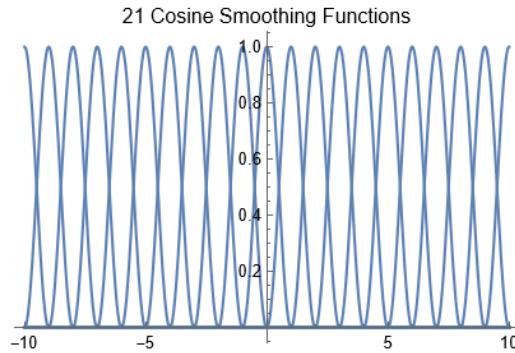
```

```

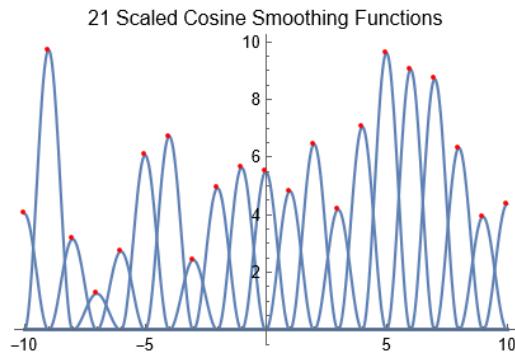
Table[
  smoothingFunction[x-anchorPoints[[i,1]],1],
  {i,1,Length[xi]}
],
(*Scaled cosine smoothing functions*)
Table[
  anchorPoints[[i,2]] *smoothingFunction[x-anchorPoints[[i,1]],1],
  {i,1,Length[xi]}
]
},
{x,-10,10},
PlotRange->All,
PlotLabel->"Activation Function with Cosine Smoothing",
Epilog->{Red,PointSize[0.01],Point[anchorPoints]},
ImageSize->300,
PlotLegends->{"Smoothed Activation (Random Anchor Points)","21 Cosine Smoothing Functions",
"21 Scaled Cosine Smoothing Functions"}
]

```

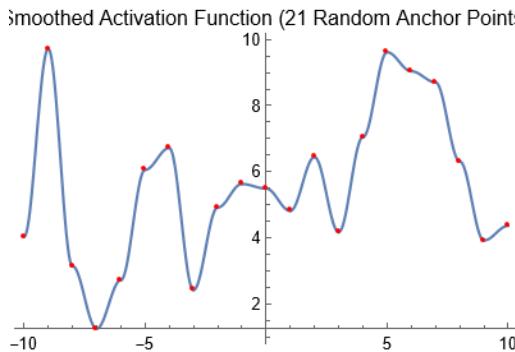
Output



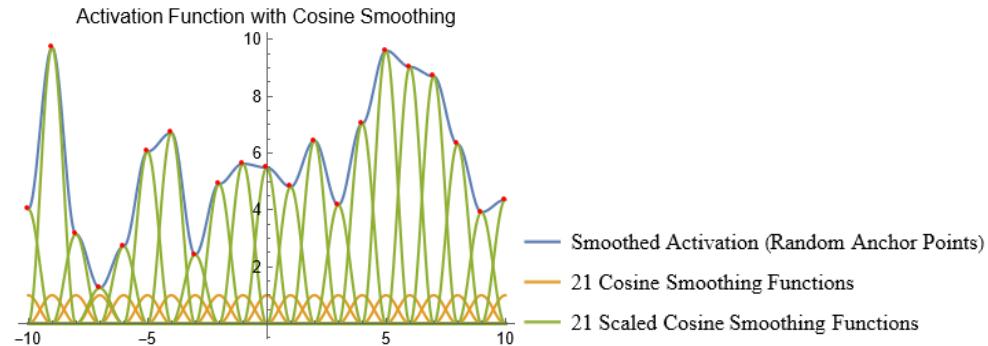
Output



Output



Output



Mathematica Code 9.38

Input

```
(* The code defines the Mixture of Gaussians Unit (MoGU) activation function as a sum of Gaussian functions, each parameterized by  $\lambda$ ,  $\mu$ , and  $\sigma$ , and computes its derivative. Using Mathematica's Manipulate function, the code creates an interactive interface that allows users to adjust these parameters via sliders. The plots, which update in real-time, display the MoGU activation function and its derivative over a specified range of input values: *)

(* Define the MoGU activation function: *)
moguActivation[x_, parameters_] := Total[
  parameters[[All, 1]]/Sqrt[2 \pi parameters[[All, 3]]^2] Exp[-(x -
  parameters[[All, 2]])^2/(2 parameters[[All, 3]]^2)]
];

(* Define the derivative of the MoGU activation function: *)
moguDerivative[x_, parameters_] := D[moguActivation[t, parameters], t] /. t -> x;

(* Visualization with Manipulate: *)
Manipulate[
 Column[
 {
 (* Plot the MoGU activation function: *)
 Plot[
  moguActivation[x, {{\lambda1, \mu1, \sigma1}, {\lambda2, \mu2, \sigma2}, {\lambda3, \mu3, \sigma3}}],
  {x, -5, 5},
  PlotRange -> All,
  AxesLabel -> {"x", "\sigma_{MoGU}(x)" },
  PlotLabel -> "MoGU Activation Function",
  PlotLegends -> {"MoGU Activation" },
  GridLines -> Automatic
 ],
 (* Plot the derivative of the MoGU activation function: *)
 Plot[
  moguDerivative[x, {{\lambda1, \mu1, \sigma1}, {\lambda2, \mu2, \sigma2}, {\lambda3, \mu3, \sigma3}}],
  {x, -5, 5},
  PlotRange -> All,
  AxesLabel -> {"x", "\sigma'_{MoGU}(x)" },
  PlotLabel -> "MoGU Activation Derivative",
  PlotLegends -> {"MoGU Derivative" },
  GridLines -> Automatic
 ]
 }
 ],
 (* Sliders to adjust the parameters  $\lambda$ ,  $\mu$ , and  $\sigma$  for three components of the MoGU function: *)
 {{\lambda1, 1, "\lambda1"}, 0.1, 2, 0.1, Appearance -> "Labeled"}, 
 {{\mu1, 0, "\mu1"}, -5, 5, 0.1, Appearance -> "Labeled"}, 
 {{\sigma1, 0.1, "\sigma1"}, 0.01, 0.5, 0.01, Appearance -> "Labeled"}]
```

```

{{\sigma_1, 1, "σ1"}, 0.1, 2, 0.1, Appearance -> "Labeled"},  

{{\lambda2, 0.5, "λ2"}, 0.1, 2, 0.1, Appearance -> "Labeled"},  

{{\mu2, 2, "μ2"}, -5, 5, 0.1, Appearance -> "Labeled"},  

{{\sigma2, 0.5, "σ2"}, 0.1, 2, 0.1, Appearance -> "Labeled"},  

{{\lambda3, 0.3, "λ3"}, 0.1, 2, 0.1, Appearance -> "Labeled"},  

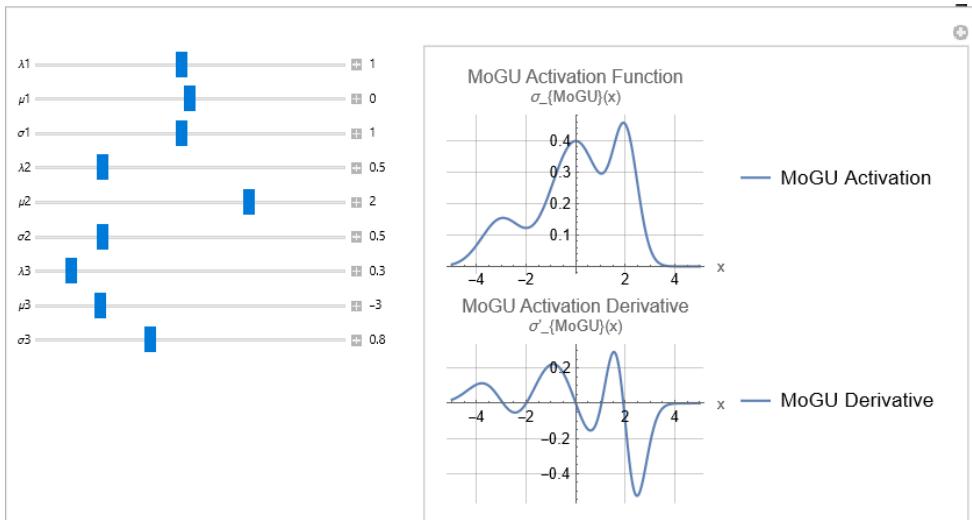
{{\mu3, -3, "μ3"}, -5, 5, 0.1, Appearance -> "Labeled"},  

{{\sigma3, 0.8, "σ3"}, 0.1, 2, 0.1, Appearance -> "Labeled"}  

]

```

Output



Mathematica Code 9.39

(* The code defines the BDAA1 function, which combines two logistic sigmoid functions and depends on the parameter a. It also computes the first derivatives of the BDAA1 function with respect to x and a. Using Mathematica's Manipulate function, the code creates an interactive interface that allows users to adjust the parameter a via a slider. The plot, which updates in real-time, displays the BDAA1 function and its derivatives over a specified range of input values: *)

```
(* Define the logistic sigmoid function: *)
σLogistic[x_]:=1/(1+Exp[-x])

(* Define the BDAA1 function: *)
σBDAA1[x_,a_]:=1/2 (σLogistic[x]+σLogistic[x+a])

(* Define the derivative of σBDAA1 with respect to x: *)
σBDAA1DerivativeX[x_,a_]:=D[σBDAA1[x,a],x]

(* Define the derivative of σBDAA1 with respect to x: *)
σBDAA1DerivativeA[x_,a_]:=D[σBDAA1[x,t],t]/. t->a

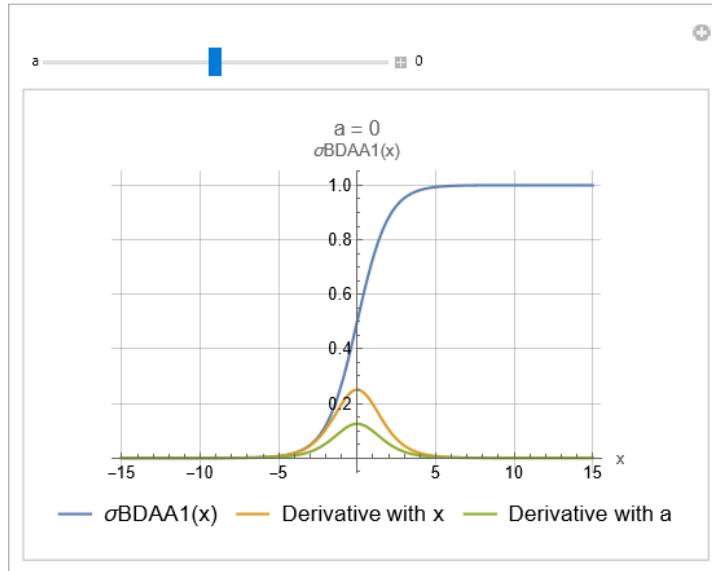
(* Manipulate to interactively change the value of 'a': *)
Manipulate[
 (* Plot the functions and their derivatives: *)
 Plot[
 Evaluate[{σBDAA1[x,a],σBDAA1DerivativeX[x,a],σBDAA1DerivativeA[x,a]}],
 {x,-15,15},
 PlotRange->All,
 PlotLegends->Placed[{"σBDAA1(x)", "Derivative with x", "Derivative with a"}, Below],
 PlotLabel->Row[{ "a = ",a}],
 AxesLabel->{{"x", "σBDAA1(x)"}}
```

```

ImageSize->300,
GridLines->Automatic
],
(* Slider to adjust the parameters a: *)
{{a,0,"a"},-10,10,0.1,Appearance->"Labeled"}
]

```

Output

**Mathematica Code 9.40**

Input (* The code defines the BDAA2 function, which combines two logistic sigmoid functions with an adjustment factor and depends on the parameter a. It also computes the first derivatives of the BDAA2 function with respect to x and a, as well as the first derivatives of the logistic sigmoid function with respect to x and a. Using Mathematica's Manipulate function, the code creates an interactive interface that allows users to adjust the parameter a via a slider. The plot displays the BDAA2 function and its derivatives over a specified range of input values: *)

```

(* Define the logistic sigmoid function: *)
σLogistic[x_]:=1/(1+Exp[-x])

(* Define the BDAA2 function: *)
σBDAA2[x_,a_]:=1/2 (σLogistic[x]+σLogistic[x+a]-1)

(* Define the derivative of the logistic sigmoid function with respect to 'x': *)
σLogisticDerivativeX[x_]:=σLogistic[x] (1-σLogistic[x])

(* Define the derivative of σBDAA2 with respect to 'x': *)
σBDAA2DerivativeX[x_,a_]:=1/2 (σLogisticDerivativeX[x]+σLogisticDerivativeX[x+a])

(* Define the derivative of the logistic sigmoid function with respect to 'a': *)
σLogisticDerivativeA[x_,a_]:=σLogistic[x+a] (1-σLogistic[x+a])

(* Define the derivative of σBDAA2 with respect to 'a': *)
σBDAA2DerivativeA[x_,a_]:=1/2 σLogisticDerivativeA[x,a]

(* Manipulate to interactively change the values of 'x' and 'a': *)
Manipulate[
(* Plot the functions and their derivatives: *)
Plot[

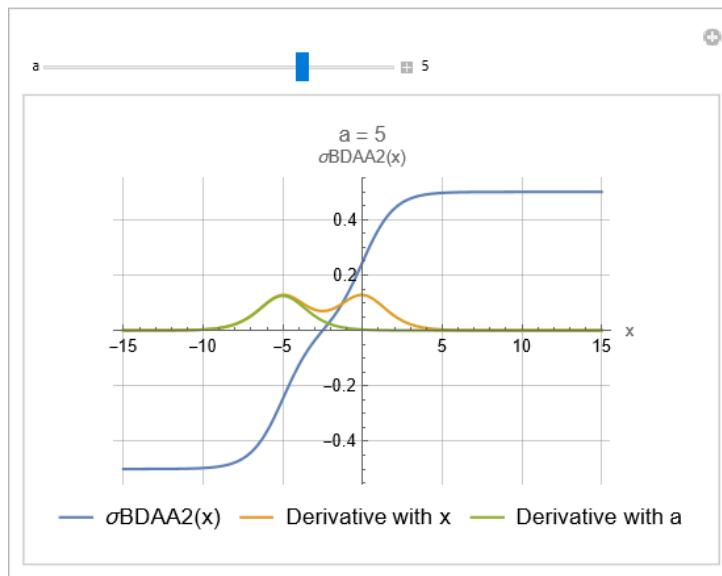
```

```

{σBDAA2[x,a],σBDAA2DerivativeX[x,a],σBDAA2DerivativeA[x,a]},
{x,-15,15},
PlotRange->All,
PlotLegends->Placed[{"σBDAA2(x)","Derivative with x","Derivative with
a"},Below],
PlotLabel->Row[{{"a = ",a}],
AxesLabel->{"x","σBDAA2(x)"},ImageSize->300,
GridLines->Automatic
],
(* Slider to adjust the parameters a: *)
{{a,5,"a"},-10,10,0.1,Appearance->"Labeled"}
]

```

Output

**Mathematica Code 9.41**

Input (* This code includes the function σBDAA3(x), its first derivative with respect to x, and its first derivative with respect to a in the same plot. The slider allows you to interactively change the value of a, and the plot updates in real-time.*)

```

(* Define the logistic sigmoid function: *)
σLogistic[x_]:=1/(1+Exp[-x])

(* Define the BDAA3 function: *)
σBDAA3[x_,a_]:=1/2 (σLogistic[x+a]+σLogistic[x-a])

(* Define the derivative of the logistic sigmoid function with respect to 'x': *)
σLogisticDerivativeX[x_]:=σLogistic[x] (1-σLogistic[x])

(* Define the derivative of σBDAA3 with respect to 'x': *)
σBDAA3DerivativeX[x_,a_]:=1/2 (σLogisticDerivativeX[x+a]+σLogisticDerivativeX[x-
a])

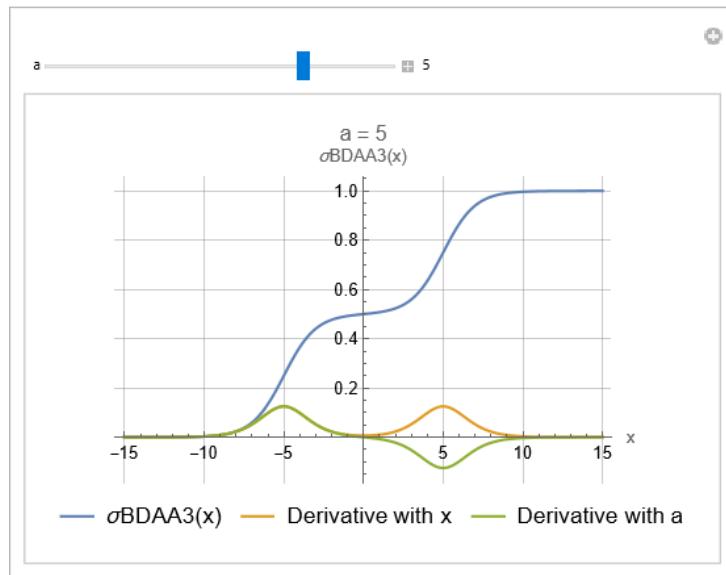
(* Define the derivative of the logistic sigmoid function with respect to 'a': *)
σLogisticDerivativeA[x_,a_]:=σLogistic[x+a] (1-σLogistic[x+a])-σLogistic[x-a] (1-
σLogistic[x-a])

(* Define the derivative of σBDAA3 with respect to 'a': *)
σBDAA3DerivativeA[x_,a_]:=1/2 σLogisticDerivativeA[x,a]

```

```
(* Manipulate to interactively change the values of 'x' and 'a': *)
Manipulate[
  (* Plot the functions and their derivatives: *)
  Plot[
    {σBDAA3[x,a], σBDAA3DerivativeX[x,a], σBDAA3DerivativeA[x,a]},
    {x,-15,15},
    PlotRange->All,
    PlotLegends->Placed[{"σBDAA3(x)", "Derivative with x", "Derivative with a"}, Below],
    PlotLabel->Row[{ "a = ", a}],
    AxesLabel->{ "x", "σBDAA3(x)" },
    ImageSize->300,
    GridLines->Automatic
  ],
  (* Slider to adjust the parameters a: *)
  {{a,5,"a"},-10,10,0.1,AxesLabel->"Labeled"}
]
```

Output



Mathematica Code 9.42

Input

```
(* This code includes the function σBDAA4(x), its first derivative with respect to x, and its first derivative with respect to a in the same plot. The slider allows you to interactively change the value of a, and the plot updates in real-time.*)

(* Define the logistic sigmoid function: *)
σLogistic[x_]:=1/(1+Exp[-x])

(* Define the BDAA3 function: *)
σBDAA3[x_,a_]:=1/2 (σLogistic[x+a]+σLogistic[x-a])

(* Define the BDAA4 function: *)
σBDAA4[x_,a_]:=σBDAA3[x,a]-1/2

(* Define the derivative of the logistic sigmoid function with respect to 'x': *)
σLogisticDerivativeX[x_]:=σLogistic[x] (1-σLogistic[x])

(* Define the derivative of σBDAA3 with respect to 'x'*)

```

```

σBDAA3DerivativeX[x_,a_]:=1/2 (σLogisticDerivativeX[x+a]+σLogisticDerivativeX[x-a])

(* Define the derivative of σBDAA4 with respect to 'x' using the chain rule: *)
σBDAA4DerivativeX[x_,a_]:= σBDAA3DerivativeX[x,a]

(* Define the derivative of the logistic sigmoid function with respect to 'a': *)
σLogisticDerivativeA[x_,a_]:=σLogistic[x+a] (1-σLogistic[x+a])-σLogistic[x-a] (1-σLogistic[x-a])

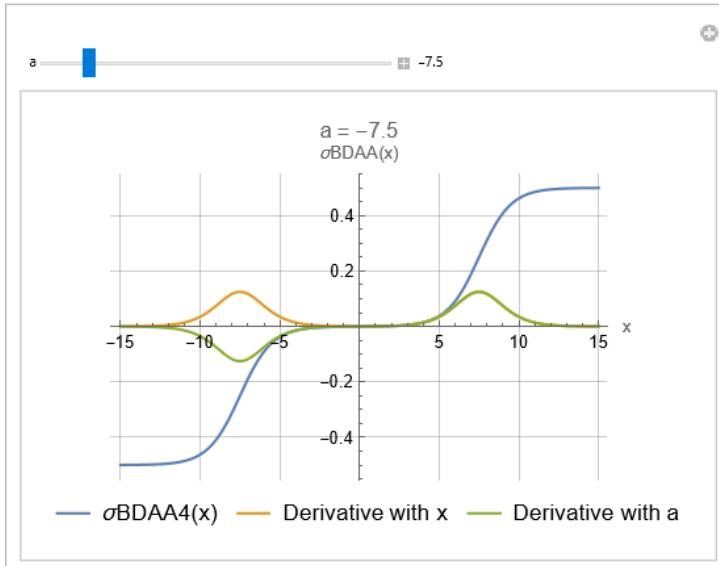
(* Define the derivative of σBDAA3 with respect to 'a': *)
σBDAA3DerivativeA[x_,a_]:=1/2 σLogisticDerivativeA[x,a]

(* Define the derivative of σBDAA4 with respect to 'a' using the chain rule: *)
σBDAA4DerivativeA[x_,a_]:= σBDAA3DerivativeA[x,a]

(* Manipulate to interactively change the values of 'x' and 'a': *)
Manipulate[
  (* Plot the functions and their derivatives: *)
  Plot[
    {σBDAA4[x,a],σBDAA4DerivativeX[x,a],σBDAA4DerivativeA[x,a]},
    {x,-15,15},
    PlotRange->All,
    PlotLegends->Placed[{"σBDAA4(x)", "Derivative with x", "Derivative with a"}, Below],
    PlotLabel->Row[{{"a = ",a}},
    AxesLabel->{"x", "σBDAA(x)"}, ImageSize->300,
    GridLines->Automatic
  ],
  (* Slider to adjust the parameters a: *)
  {{a,-7.5,"a"},-10,10,0.1,AxesLabel->"Labeled"}
]

```

Output



Unit 9.2

Custom Layers in Neural Networks

In the ever-evolving field of deep learning, the Wolfram Language provides a comprehensive suite of tools for constructing, training, and deploying neural networks. Mathematica, with its powerful symbolic and numerical capabilities, offers a rich set of built-in functions that simplify the creation of complex neural network architectures. This unit delves into some of these fundamental components, including `FunctionLayer`, `ParametricRampLayer`, `SoftmaxLayer`, `NetEncoder`, and `NetDecoder`.

FunctionLayer

`ElementwiseLayer` is designed for situations where you need to apply a function to each element of an input tensor independently. This layer is highly useful for implementing common activation functions (such as `ReLU`, `sigmoid`, and `tanh`) as well as custom transformations. By applying functions element-wise, you can tailor the behavior of your neural network to fit your specific needs, enhancing its performance and interpretability. On the other hand, `FunctionLayer` takes customization a step further by allowing you to define layers based on arbitrary functions that can operate on entire tensors. This is particularly useful when your desired transformation involves complex operations, multiple inputs, or non-standard tensor manipulations that go beyond element-wise operations.

FunctionLayer[f]

represents a net layer that applies function f to its input.

Remarks:

- `FunctionLayer` is used to define neural nets from usual Wolfram Language code.
- `FunctionLayer[f][x]` behaves in the same way as `f[x]`.
- The function f should involve only valid operations on arrays that produce an array or an association of arrays.
- Valid operations include arithmetic functions (`Plus`, `Times`, etc.), elementary functions (`Exp`, `Sqrt`, `Sin`, etc.), numerical functions (`Min`, `Round`, `Ramp`, etc.), array constructions (`Table`, `ConstantArray`, etc.), array operations (`Dot`, `Det`, `Tr`, etc.), descriptive statistics (`Mean`, `StandardDeviation`, etc.) and distance and similarities (`EuclideanDistance`, `HammingDistance`, etc.). It is also possible to use looping constructs (`Map`, `NestList`, `FoldList`, etc.) and list manipulation functions (`Part`, `Reverse`, etc.).
- Function f must take only one argument as input. The argument can be an array or an association of arrays.
- If the argument of f is a unique array, f can be defined by a pure function, a symbol or a composition of these. The resulting layer will have a unique input port called "`Input`".
- `FunctionLayer` can also be given a multiple-argument function f using the syntax `FunctionLayer[Apply[f]]`. In this case, ports are named automatically.
- `FunctionLayer[f,"port" -> shape]` can be used as in `NetGraph` to specify the shape, encoder or decoder of a given port.

Mathematica Code 9.43

```
Input      (* The code demonstrates the creation and usage of a `FunctionLayer` in Mathematica
           that calculates the standard deviation of a given set of numbers. It shows how to
           define this `FunctionLayer`, calculate the standard deviation both directly and
           through the `FunctionLayer`, and extract the underlying function and neural network
           structure from the `FunctionLayer`. Additionally, it includes a command to display
           the network graph of the `FunctionLayer`, providing a visual representation of the
           layer's architecture. This code is useful for understanding how to encapsulate
           mathematical functions within neural network layers and interact with these layers
           programmatically: *)
```

```
(* Define a FunctionLayer that calculates the standard deviation: *)
stdLayer=FunctionLayer[StandardDeviation]

(* Calculate the standard deviation of the given list of numbers directly: *)
standardDeviationResult=StandardDeviation[{1.3,2.1,2,3.56,2.4,4.31,6.35,7.,8.2}]

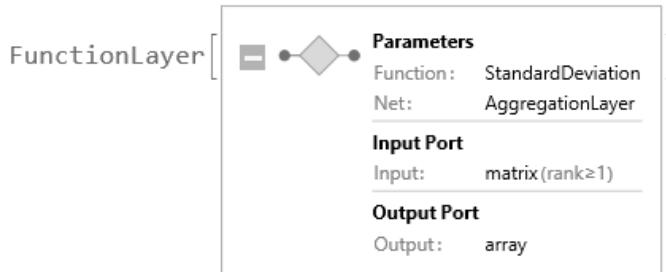
(* Calculate the standard deviation of the given list of numbers using the
FunctionLayer: *)
stdLayerResult=stdLayer[{1.3,2.1,2,3.56,2.4,4.31,6.35,7.,8.2}]

(* Extract the function from the FunctionLayer: *)
extractedFunction=NetExtract[stdLayer,"Function"]

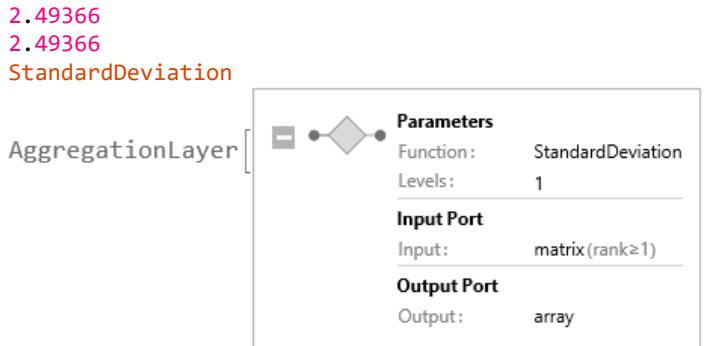
(* Extract the underlying neural network from the FunctionLayer: *)
extractedNet=NetExtract[stdLayer,"Net"]

(* Display the network graph of the FunctionLayer: *)
networkGraph=NetGraph[stdLayer]
```

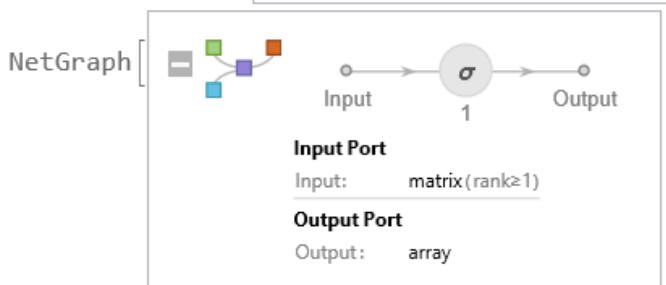
Output



Output
Output
Output
Output



Output

**Mathematica Code 9.44**

Input (* The code defines and utilizes a custom function that computes various statistical measures (Mean, Variance, Standard Deviation, Median, Min, and Max) for a given matrix of numerical values. It demonstrates the creation of a `FunctionLayer` from this custom function and its conversion into a `NetGraph` to facilitate its application within a neural network framework. The code generates a sample input

matrix, applies both the custom function and the `FunctionLayer` to this matrix, and computes the same statistical measures directly using Mathematica's built-in functions. Finally, it compares the results from these three methods to ensure consistency and accuracy in the computation of the statistical measures: *)

```

(* Define a function that calculates various statistical measures: *)
statisticalMeasuresFunction=Function[
  <|
    "Mean" -> Mean[#],
    "Variance" -> Variance[#],
    "StandardDeviation" -> StandardDeviation[#],
    "Median" -> Median[#],
    "Min" -> Min[#],
    "Max" -> Max[#]
  |>
];

```

(* Create a FunctionLayer using the defined function and convert it to a NetGraph: *)

```

statisticalMeasuresLayer=NetGraph@FunctionLayer[statisticalMeasuresFunction]
```

(* Generate a 7x3 matrix of numerical values, each element calculated as the square root of ($i+2*j$): *)

```

inputMatrix=N@Array[Sqrt[#1+2*#2]&,{7,3}]
```

(* Apply the function directly to the input matrix to compute the statistical measures: *)

```

statisticalMeasuresFunctionResult=statisticalMeasuresFunction[inputMatrix];
```

(* Apply the FunctionLayer (as a NetGraph) to the input matrix to compute the statistical measures: *)

```

statisticalMeasuresLayerResult=statisticalMeasuresLayer[inputMatrix];
```

(* Compute the statistical measures directly using built-in Mathematica functions: *)

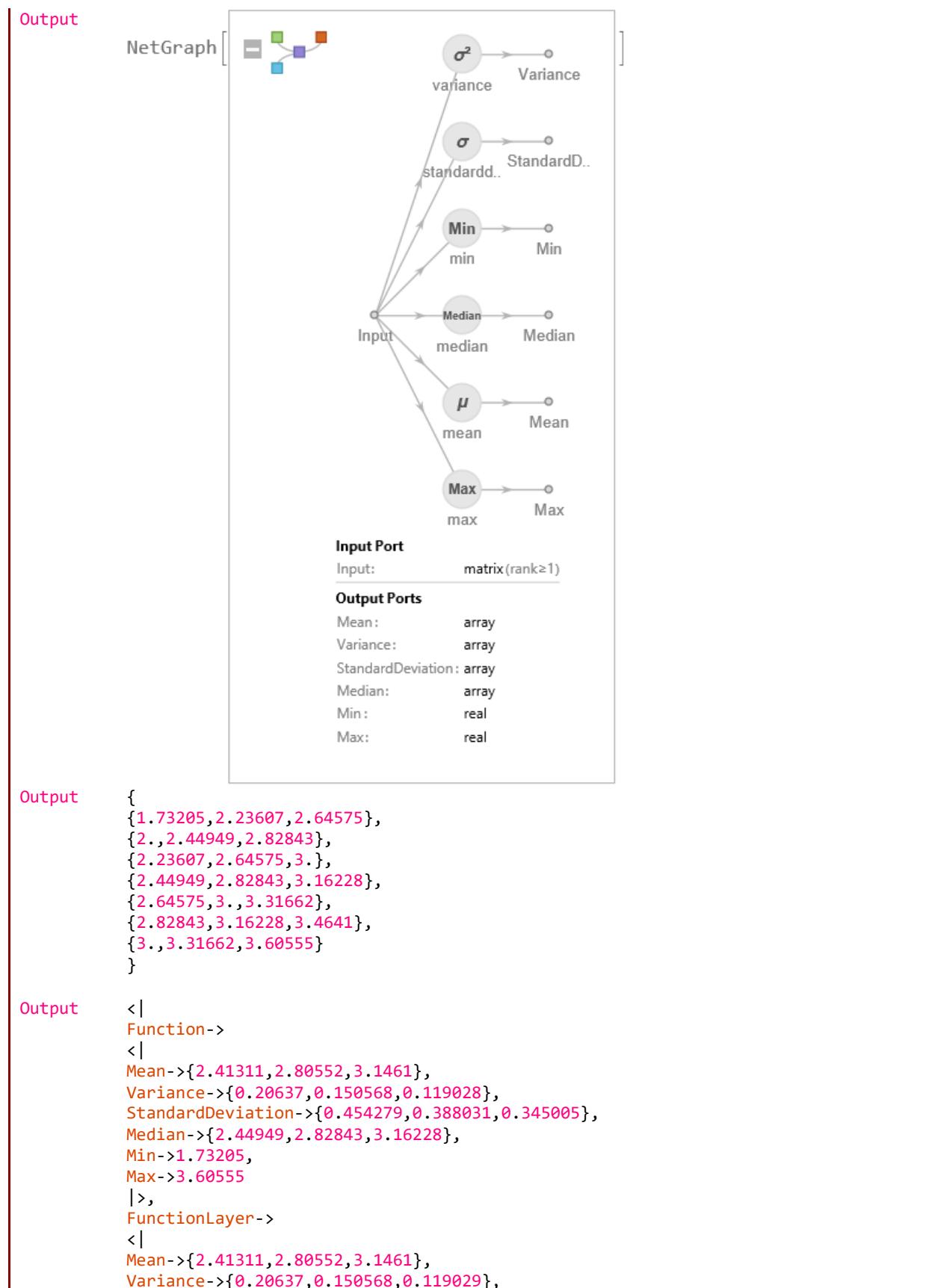
```

meanResult=Mean[inputMatrix];
varianceResult=Variance[inputMatrix];
standardDeviationResult=StandardDeviation[inputMatrix];
medianResult=Median[inputMatrix];
minResult=Min[inputMatrix];
maxResult=Max[inputMatrix];
```

(* Compare the results from the function, FunctionLayer, and direct methods: *)

```

comparisonResults=<|
  "Function" -> statisticalMeasuresFunctionResult,
  "FunctionLayer" -> statisticalMeasuresLayerResult,
  "Direct" -> <|
    "Mean" -> meanResult,
    "Variance" -> varianceResult,
    "StandardDeviation" -> standardDeviationResult,
    "Median" -> medianResult,
    "Min" -> minResult,
    "Max" -> maxResult
  |>
|>
```



```

StandardDeviation->{0.454279,0.388031,0.345005},
Median->{2.44949,2.82843,3.16228},
Min->1.73205,
Max->3.60555
|>
Direct->
<|
Mean->{2.41311,2.80552,3.1461},
Variance->{0.20637,0.150568,0.119028},
StandardDeviation->{0.454279,0.388031,0.345005},
Median->{2.44949,2.82843,3.16228},
Min->1.73205,
Max->3.60555
|>
|>

```

Mathematica Code 9.45

```

Input (* The code creates and utilizes custom `FunctionLayer` objects within `NetGraph` structures in Mathematica to perform normalization and summation operations on input data. Specifically, it demonstrates the creation of a `FunctionLayer` for standard normalization, another for exponential normalization, and one for computing the total sum. The code generates a sample input vector, applies these `FunctionLayer` objects to the input vector, and compares the results with those obtained using direct built-in Mathematica functions. It further extends the comparison to include the total sums of the normalized results, ensuring the consistency and accuracy of the computations performed by the `FunctionLayer` implementations: *)

(* Create a NetGraph using a FunctionLayer that normalizes its input: *)
normalizationLayer=NetGraph@FunctionLayer[Normalize]

(* Create a NetGraph using a FunctionLayer that normalizes the exponential of its input, dividing by the total sum: *)
exponentialNormalizationLayer=NetGraph@FunctionLayer[Normalize[Exp[#],Total]&

(* Create a NetGraph using a FunctionLayer that computes the total sum of its input: *)
totalLayer=NetGraph@FunctionLayer[Total]

(* Generate a sample input vector: *)
inputData={2.5,1.5,5.7,7.2,11.1}

(* Apply the normalization FunctionLayer to the input vector: *)
normalizationLayerResult=normalizationLayer[inputData]

(* Apply the exponential normalization FunctionLayer to the input vector: *)
exponentialNormalizationLayerResult=exponentialNormalizationLayer[inputData]

(* Apply the total FunctionLayer to the normalization results: *)
totalNormalizationLayerResult=totalLayer[normalizationLayerResult]
totalExponentialNormalizationLayerResult=totalLayer[exponentialNormalizationLayerResult]

(* Compute the normalization directly using built-in Mathematica functions: *)
directNormalizationResult=Normalize[inputData]

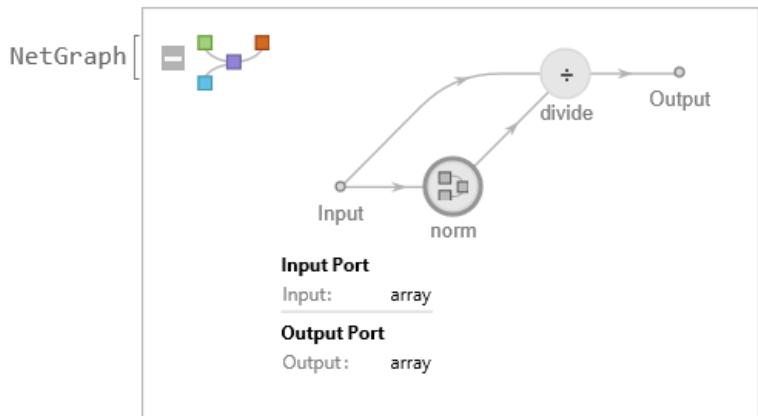
(* Compute the exponential normalization directly using built-in Mathematica functions: *)
directExponentialNormalizationResult=Normalize[Exp[inputData],Total]

```

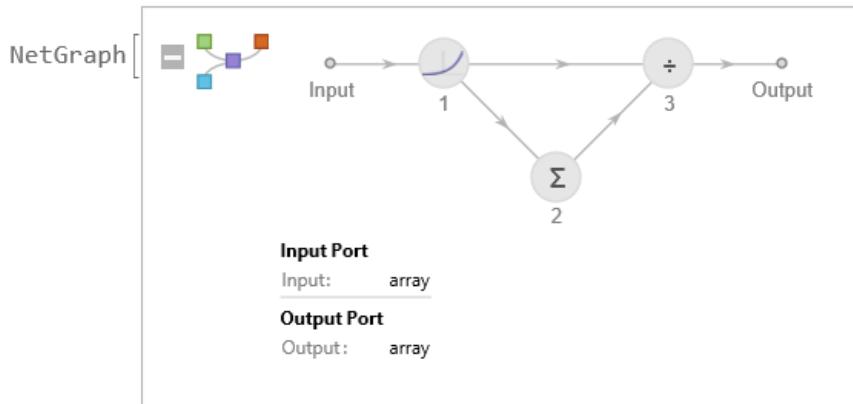
```
(* Compute the total sum directly using built-in Mathematica functions: *)
directTotalNormalizationResult=Total[directNormalizationResult]
directTotalExponentialNormalizationResult=Total[directExponentialNormalizationResult]

(* Compare the results from the FunctionLayer and direct methods: *)
comparisonResults=<|
  "NormalizationLayer"->normalizationLayerResult,
  "DirectNormalization"->directNormalizationResult,
  "TotalNormalizationLayer"->totalNormalizationLayerResult,
  "DirectTotalNormalization"->directTotalNormalizationResult,
  "ExponentialNormalizationLayer"->exponentialNormalizationLayerResult,
  "DirectExponentialNormalization"->directExponentialNormalizationResult,
  "TotalExponentialNormalizationLayer"->totalExponentialNormalizationLayerResult,
  "DirectTotalExponentialNormalization"->
  >directTotalExponentialNormalizationResult
|>
```

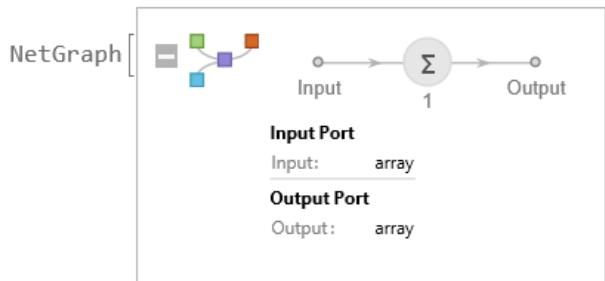
Output



Output



Output



```

Output {2.5,1.5,5.7,7.2,11.1}
Output {0.170088,0.102053,0.3878,0.489853,0.755189}
Output {0.000179614,0.000660761,0.00440637,0.019748,0.9756}
Output 1.90498
Output 1.
Output {0.170088,0.102053,0.3878,0.489853,0.755189}
Output {0.000179614,0.000660762,0.00440638,0.019748,0.9756}
Output 1.90498
Output 1.
<|
NormalizationLayer->{0.170088,0.102053,0.3878,0.489853,0.755189},
DirectNormalization->{0.170088,0.102053,0.3878,0.489853,0.755189},
TotalNormalizationLayer->1.90498,
DirectTotalNormalization->1.90498,
ExponentialNormalizationLayer-
>{0.000179614,0.000660761,0.00440637,0.019748,0.9756},
DirectExponentialNormalization-
>{0.000179614,0.000660762,0.00440638,0.019748,0.9756},
TotalExponentialNormalizationLayer->1.,
DirectTotalExponentialNormalization->1.
|>

```

Mathematica Code 9.46

```

Input (* Standardize[list] shifts and rescales the elements of list to have zero mean
       and unit sample variance. The code demonstrates the creation and application of a
       FunctionLayer in Mathematica for standardizing input data, and to compare its
       performance with the direct standardization method using built-in functions. The
       code creates a FunctionLayer that applies the Standardize function, and applies
       this layer to a sample input vector. It then computes the standardization directly
       using Mathematica's built-in Standardize function and compares the results from
       both methods to ensure consistency and accuracy: *)

(* Create a FunctionLayer that standardizes its input: *)
standardizationFunctionLayer=FunctionLayer[Standardize]

(* Create a NetGraph using the standardization FunctionLayer: *)
standardizationNetGraph=NetGraph@FunctionLayer[Standardize]

(* Generate a sample input vector: *)
inputData={1.5,2.3,3.7,4.2,5.1}

(* Apply the standardization FunctionLayer to the input vector: *)
standardizationLayerResult=standardizationFunctionLayer[inputData];

(* Compute the mean of the standardized input vector: *)
meanStandardizationLayerResult=Mean[standardizationLayerResult]

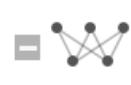
(* Compute the variance of the standardized input vector: *)
varianceStandardizationLayerResult=Variance[standardizationLayerResult]

(* Compute the standardization directly using the built-in Mathematica function: *)
directStandardizationResult=Standardize[inputData];

(* Compare the results from the FunctionLayer and the direct method: *)
comparisonResults=<|"StandardizationFunctionLayer"-
>standardizationLayerResult,"DirectStandardization"-
>directStandardizationResult|>

```

Output

FunctionLayer [ **Parameters**
Function: Standardize
Net: NormalizationLayer]

Input Port
Input: array

Output Port
Output: array

Output

NetGraph [ Input → N → Output]

Input Port
Input: array

Output Port
Output: array

```

Output {1.5,2.3,3.7,4.2,5.1}
Output 5.36442*10^-8
Output 1.
<|
StandardizationFunctionLayer->{-1.28108,-0.73008,0.234177,0.578554,1.19843},
DirectStandardization->{-1.28108,-0.73008,0.234177,0.578554,1.19843}
|>

```

Mathematica Code 9.47

Input (* The code creates an `AssociationMap` of `NetGraph` objects that encapsulate `FunctionLayer` operations for computing different distance metrics, specifically Euclidean and Manhattan distances. The code then demonstrates how to apply these `FunctionLayer` objects to a sample input vector consisting of two points in 2D space. Additionally, the code computes the same distances using Mathematica's built-in functions for verification: *)

```

(* Create an AssociationMap of NetGraphs using FunctionLayers for different
distance functions: *)
distanceFunctionLayers=AssociationMap[NetGraph@*FunctionLayer@*Apply,{EuclideanDistance,ManhattanDistance}]

(* Generate a numerical example input: a pair of points in 2D space: *)
point1={1.0,2.0};
point2={4.0,6.0};
exampleInput={point1,point2}

(* Compute distances using the FunctionLayers in the NetGraphs:*)
(* Apply EuclideanDistance FunctionLayer: *)
euclideanDistanceResult=distanceFunctionLayers[EuclideanDistance][exampleInput];

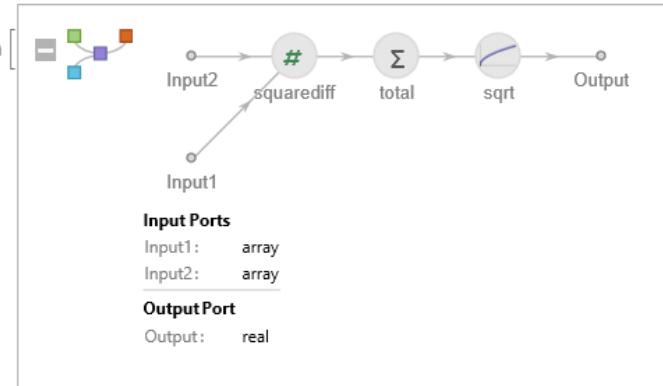
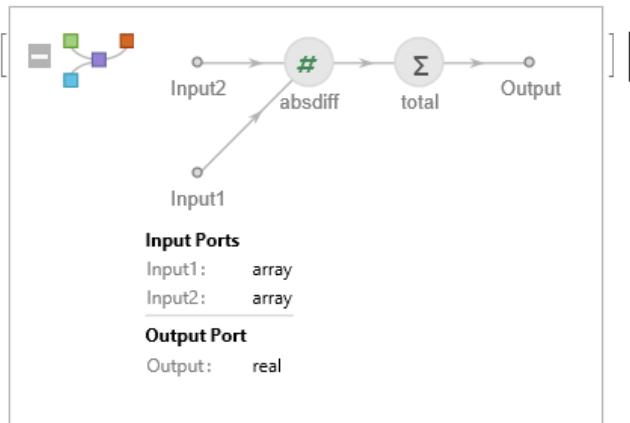
(* Apply ManhattanDistance FunctionLayer: *)
manhattanDistanceResult=distanceFunctionLayers[ManhattanDistance][exampleInput];

(* Compute distances directly using built-in Mathematica functions for
verification: *)
directEuclideanDistanceResult=EuclideanDistance@@exampleInput;
directManhattanDistanceResult=ManhattanDistance@@exampleInput;

```

```
(* Compare the results from the FunctionLayers and the direct methods: *)
comparisonResults=<|
  "EuclideanDistanceFunctionLayer"->euclideanDistanceResult,
  "DirectEuclideanDistance"->directEuclideanDistanceResult,
  "ManhattanDistanceFunctionLayer"->manhattanDistanceResult,
  "DirectManhattanDistance"->directManhattanDistanceResult|>
```

Output

`EuclideanDistance → NetGraph``, ManhattanDistance → NetGraph`Output
Output

```
  {{1.,2.},{4.,6.}}
<|
  EuclideanDistanceFunctionLayer->5.,
  DirectEuclideanDistance->5.,
  ManhattanDistanceFunctionLayer->7.,
  DirectManhattanDistance->7.
|>
```

ParametricRampLayer

ParametricRampLayer[]

represents a net layer that computes a leaky ReLU activation with a slope that can be learned.

ParametricRampLayer[levels]

specifies the levels on which each dimension has a specific slope.

The following optional parameter can be specified:

Slope "	Automatic	learnable tensor of slopes
LearningRateMultipliers	Automatic	learning rate multipliers for the slopes

Remarks:

- The slope is a coefficient of leakage applied to input negative values.
- By default, the slope is initialized to 0.1.
- `ParametricRampLayer["Slope" -> value, LearningRateMultipliers -> 0]` is a leaky ReLU with a fixed slope.
- `ParametricRampLayer` exposes the following ports for use in `NetGraph` etc.:

<code>"Input"</code>	an array of arbitrary rank
<code>"Output"</code>	an array with the same dimensions as the input

- When it cannot be inferred from other layers in a larger net, the option `"Input" -> {n1, n2, ...}` can be used to fix the input dimensions of `ParametricRampLayer`.

Mathematica Code 9.48

```
Input (* The code aims to initialize a ParametricRampLayer with 4 input dimensions,
       extract and display its initial slopes,(by default, the slope is initialized to
       0.1) and apply it to a set of specific inputs to observe the initial transformation.
       It then trains the layer using random input-output pairs of size 15x4, adjusting
       the slope parameters to minimize the prediction error. Finally, the code extracts
       and displays the slopes of the trained layer, illustrating the changes in parameters
       resulting from the training process: *)

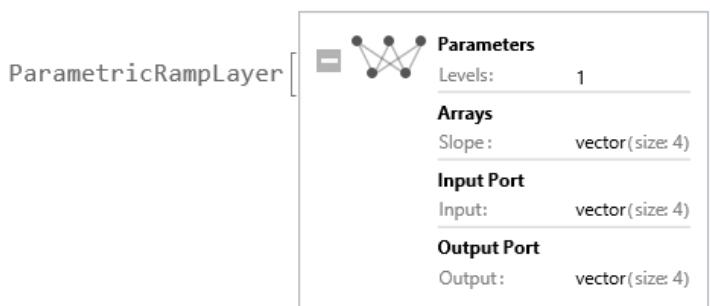
(* Initialize a ParametricRampLayer with 3 input dimensions: *)
parametricRampLayer=NetInitialize@ParametricRampLayer["Input" -> 4]

(* Extract and display the initial slopes of the ParametricRampLayer: *)
initialSlopes=Normal@NetExtract[parametricRampLayer,"Slope"]

(* Apply the initialized layer to some inputs: *)
outputs=parametricRampLayer[{{{-3,-2,-1,-4},{0,2,1,4}}}

(* Train the layer using random input-output pairs,each of size 15x3: *)
trainedLayer=NetTrain[parametricRampLayer,RandomReal[{-1,1},{15,4}]->RandomReal[{-1,1},{15,4}]]

(* Extract and display the slopes of the trained ParametricRampLayer: *)
trainedSlopes=Normal@NetExtract[trainedLayer,"Slope"]
```

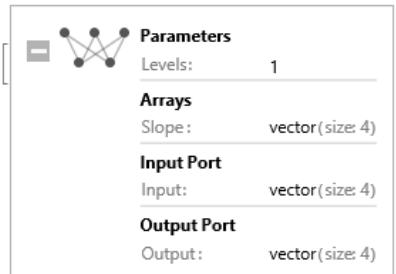
Output

Output `{0.1,0.1,0.1,0.1}`

Output `{
 {-0.3,-0.2,-0.1,-0.4},
 {0.,2.,1.,4.}
}`

Output

ParametricRampLayer

Output $\{-0.0547273, -0.0223373, 0.129081, -0.369611\}$ **Mathematica Code 9.49**

Input

```
(* The code initializes a ParametricRampLayer with a fixed slope of 0.1 and no learning rate multiplier for the slope, creating a leaky ReLU layer with 4 input dimensions. It then trains a neural network that includes this leaky ReLU layer, along with a linear layer, another linear layer, and a logistic sigmoid layer, to map integer inputs to boolean outputs. Finally, the code extracts and displays the slope of the leaky ReLU layer after training, confirming that the slope remains unchanged due to the fixed learning rate multiplier, ensuring the parameter stability throughout the training process: *)
```

```
(* Initialize a ParametricRampLayer with a fixed slope of 0.1 and no learning rate multiplier for the slope (creating a leaky ReLU with a fixed slope of 0.1) with 4 input dimensions: *)
```

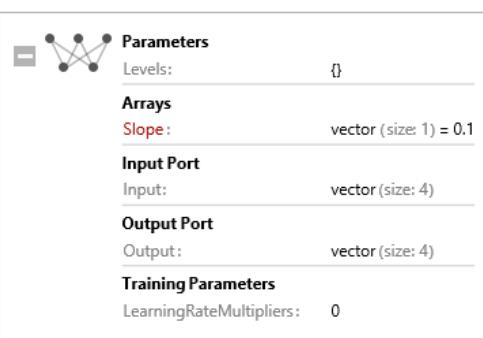
```
leakyReLULayer=ParametricRampLayer[
  {},
  "Slope" -> 0.1,
  LearningRateMultipliers -> 0,
  "Input" -> 4
]
```

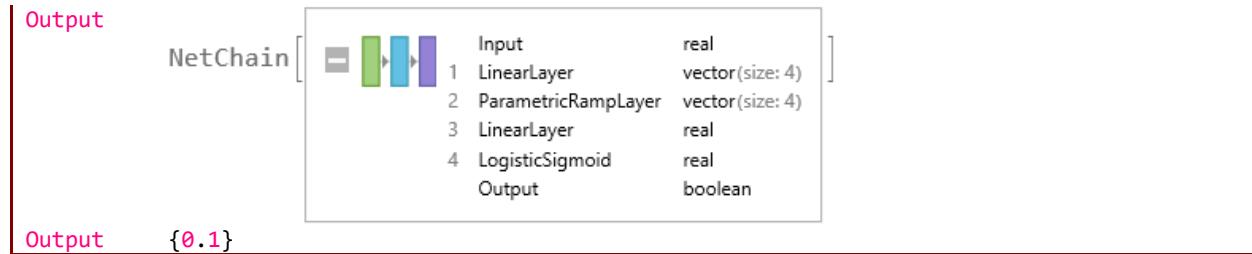
```
(* Train a neural network consisting of a linear layer, the initialized leaky ReLU layer, another linear layer, and a logistic sigmoid layer. The network is trained to map integer inputs to boolean outputs: *)
```

```
trainedNetwork=NetTrain[
  NetChain[
    {
      LinearLayer[4], leakyReLULayer,
      LinearLayer[{}], LogisticSigmoid
    },
    "Output" -> NetDecoder["Boolean"]
  ],
  {1 -> False, 2 -> True, 3 -> True, 4 -> False, 5 -> False, 6 -> True, 7 -> False}
]
(* Extract and display the slope of the leaky ReLU layer after training, confirming that it remains unchanged: *)
trainedSlope=Normal@NetExtract[trainedNetwork,{2,"Slope"}]
```

Output

ParametricRampLayer

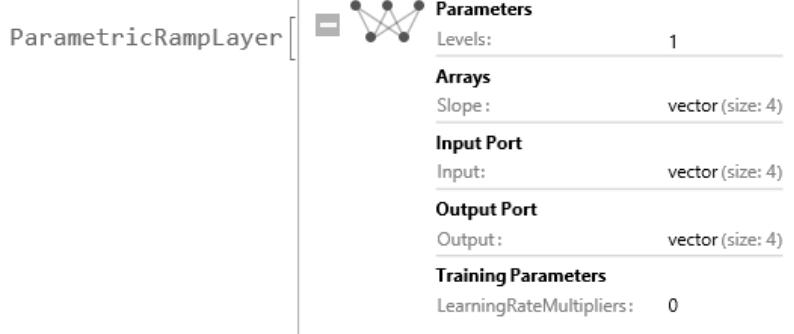


**Mathematica Code 9.50**

```
Input (* The code initializes a `ParametricRampLayer` with specified slopes of 0.1, 0.3, 0.4, and 0.5 for each of the 4 input dimensions, ensuring the slopes remain fixed during training by setting the learning rate multiplier to 0. It then applies the initialized layer to a set of input vectors `{{{-1,-1,-1,-1},{1,1,1,1}}}` to observe how the layer transforms these inputs based on the predefined slopes, demonstrating the layer's behavior with the given inputs: *)
```

```
(* Initialize a ParametricRampLayer with specified slopes for each input dimension and no learning rate multiplier for the slopes, with 4 input dimensions: *)
prelu=ParametricRampLayer["Slope"->{0.1,0.3,0.4,0.5},LearningRateMultipliers->0,"Input"->4]

(* Apply the initialized layer to a set of input vectors, observing how the layer transforms the inputs: *)
prelu[{{{-1,-1,-1,-1},{1,1,1,1}}}]
```

Output**Output**

```
{  
{-0.1,-0.3,-0.4,-0.5},  
{1.,1.,1.,1.}  
}
```

SoftmaxLayer

SoftmaxLayer[]

represents a softmax net layer.

SoftmaxLayer[n]

represents a softmax net layer that uses level n as the normalization dimension.

Remarks:

- **SoftmaxLayer[...][input]** explicitly computes the output for input.
- **SoftmaxLayer[...][{input1, input2, ...}]** explicitly computes outputs for each of the inputs.
- **SoftmaxLayer** is typically used inside **NetChain**, **NetGraph**, etc. to normalize the output of other layers in order to use them as class probabilities for classification tasks.

- **SoftmaxLayer** can operate on arrays that contain "Varying" dimensions.
- **SoftmaxLayer** exposes the following ports for use in **NetGraph** etc.:

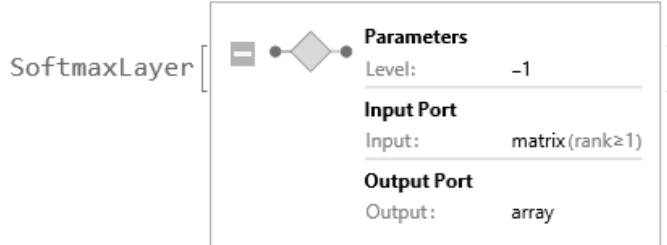
"Input"	a numerical array of dimensions $d_1 \times d_2 \times \dots \times d_n$
"Output"	a numerical array of dimensions $d_1 \times d_2 \times \dots \times d_n$

- When it cannot be inferred from other layers in a larger net, the option "**Input**"->**n** can be used to fix the input dimensions of **SoftmaxLayer**.
- **SoftmaxLayer[]** is equivalent to **SoftmaxLayer[-1]**.
- **SoftmaxLayer** effectively normalizes the exponential of the input array, producing vectors that sum to 1. For the default level of -1, the innermost dimension is used as the normalization dimension.
- When **SoftmaxLayer[-1]** is applied to a vector v, it produces the vector **Normalize[Exp[v], Total]**. When applied to an array of higher dimension, it is mapped onto level -1.
- When **SoftmaxLayer[n]** is applied to a k-dimensional input array $x_{(d_1) \dots d_{(k)}}$, it produces the array , where n is the summed-over index of x.

Mathematica Code 9.51

Input (* Create a SoftmaxLayer: *)
SoftmaxLayer[]

Output

**Mathematica Code 9.52**

Input (* The code defines a Softmax function that computes the Softmax values for a given input list by exponentiating each element, summing these exponentials, and normalizing each value by dividing it by the sum of the exponentials. It then tests this function using an example input vector `{{1.0,2.0,3.0}}` ,computes the resulting Softmax output, and verifies that the sum of the Softmax output is 1,confirming that the function correctly transforms the input values into a probability distribution: *)

```
(*Define the Softmax function*)
Softmax[x_]:=Module[
{expValues,sumExpValues},
(* Compute the element-wise exponential of the input list x: *)
expValues=Exp[x];
(* Compute the sum of the exponential values: *)
sumExpValues=Total[expValues];
(* Normalize the exponential values by dividing each by the sum: *)
expValues/sumExpValues
]

(* Example input vector: *)
inputVector={2.0,3.0,4.0,5.0};

(* Compute the Softmax output for the example input vector: *)
softmaxOutput=Softmax[inputVector]

(* Display the total of the Softmax output, which should be 1: *)
```

```
Total[softmaxOutput]

Output {0.0320586, 0.0871443, 0.236883, 0.643914}
Output 1.
```

Mathematica Code 9.53

Input (* The code initializes a SoftmaxLayer with 4 input dimensions to process vectors of this specific size. It then applies this layer to an example input vector {2.0,3.0,4.0,5.0}, computing the Softmax transformation to convert the input into a probability distribution. Finally, the code calculates and displays the total of the Softmax output, verifying that the sum is 1, which confirms the correct functionality of the Softmax layer by ensuring the output forms a valid probability distribution: *)

```
(* Initialize a SoftmaxLayer with 4 input dimensions: *)
softmaxLayer=SoftmaxLayer["Input"->{4}]

(* Apply the Softmax layer to an input vector*)
softmaxOutput=softmaxLayer[{2.0,3.0,4.0,5.0}]

(* Compute and display the total of the Softmax output, which should be 1: *)
outputSum=Total[softmaxOutput]
```

Output

```
SoftmaxLayer[ Parameters
Level: -1
Input Port
Input: vector (size:4)
Output Port
Output: vector (size:4)]
```

```
Output {0.0320586, 0.0871443, 0.236883, 0.643914}
Output 1.
```

Mathematica Code 9.54

Input (* The code initializes a SoftmaxLayer with input dimensions 4x2 to process matrices of this specific size. It then applies this layer to an example input matrix {{2,1}, {0,3}, {-2,4}, {5,2}}, computing the Softmax transformation for each row and converting the input rows into probability distributions, with the results displayed in matrix form. Finally, the code calculates and displays the total of each row of the Softmax output, verifying that each row sums to 1, which confirms the correct functionality of the Softmax layer by ensuring that the output forms valid probability distributions for each row: *)

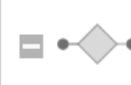
```
(* Initialize a SoftmaxLayer with input dimensions 4x2: *)
softmax=SoftmaxLayer["Input"->{4,2}]

(* Apply the Softmax layer to a 3x2 input matrix: *)
softmaxOutput=softmax[{{2,1},{0,3},{-2,4},{5,2}}];

(* Display the result in matrix form: *)
softmaxOutput//MatrixForm

(* Compute and display the total of each row of the Softmax output: *)
rowSums=Map[Total,softmaxOutput]
```

Output

SoftmaxLayer []

Parameters
Level: -1
Input Port
Input: matrix (size: 4×2)
Output Port
Output: matrix (size: 4×2)

Output ({
 {0.731059, 0.268941},
 {0.0474259, 0.952574},
 {0.00247262, 0.997527},
 {0.952574, 0.0474259}
 })

Output {1.,1.,1.,1.}

Mathematica Code 9.55

Input (* The code initializes a SoftmaxLayer with varying input dimensions, allowing it to process input vectors of different lengths. It then applies this layer to several input vectors of lengths 4, 3, and 2, respectively, computing the Softmax transformation for each vector to convert them into probability distributions. This demonstrates the flexibility and functionality of the SoftmaxLayer in handling inputs of varying sizes and producing correct probability distributions for each input vector: *)

```
(*Initialize a SoftmaxLayer with varying input dimensions*)
softmax=SoftmaxLayer["Input"→"Varying"]

(*Apply the Softmax layer to an input vector of length 4*)
softmax[{2,3,4,5}]

(*Apply the Softmax layer to an input vector of length 3*)
softmax[{2,3,4}]

(*Apply the Softmax layer to an input vector of length 2*)
softmax[{2,3}]
```

Output

SoftmaxLayer []

Parameters
Level: -1
Input Port
Input: vector of n scalars
Output Port
Output: vector of n reals

Output {0.0320586, 0.0871443, 0.236883, 0.643914}
Output {0.0900306, 0.244728, 0.665241}
Output {0.268941, 0.731059}

NetEncoder

```
NetEncoder["name"]
```

represents an encoder that takes a given form of input and encodes it as an array for use in a net.

NetEncoder[{"name", ...}]

represents an encoder with additional parameters specified.

Remarks:

- `NetEncoder[...][input]` gives the specified encoding for input.
- `NetEncoder[...][{input1, input2, ...}]` explicitly computes outputs for each of the inputs.
- Possible named encoders include:

<code>"Audio"</code>	encode audio as a sequence of waveform amplitudes
<code>"AudioMelSpectrogram"</code>	encode audio as a mel spectrogram
<code>"AudioMFCC"</code>	encode audio as a sequence of MFCC vectors
<code>"AudioSpectrogram"</code>	encode audio as a spectrogram
<code>"AudioSTFT"</code>	encode audio as a sequence of Fourier transforms
<code>"Boolean"</code>	encode True and False as 1 and 0
<code>"Characters"</code>	encode characters in a string as a sequence of integer codes or one-hot vectors
<code>"Class"</code>	encode a class label as an integer code or a one-hot vector
<code>"FeatureExtractor"</code>	encode any kind of input as in FeatureExtraction
<code>"Function"</code>	use a custom function to encode an input
<code>"Image"</code>	encode a 2D image as a rank-3 array
<code>"Image3D"</code>	encode a 3D image as a rank-4 array
<code>"SubwordTokens"</code>	encode tokens in a string as a sequence of integer codes
<code>"Tokens"</code>	encode tokens in a string as a sequence of integer codes
<code>"UTF8"</code>	encode strings as their UTF8 bytes
<code>"VideoFrames"</code>	encode a video as a sequence of rank-3 arrays

- A `NetEncoder` object can be attached to an input port of a net by specifying `"port" -> NetEncoder[...]` when constructing the net. Specifying `"port" -> "name"` will create an encoder using `NetEncoder["name"]` and attach it.

NetDecoder**NetDecoder["name"]**

represents a decoder that takes a net representation and decodes it into an expression of a given form.

NetDecoder[{"name", ...}]

represents a decoder with additional parameters specified.

Remarks:

- `NetDecoder[...][array]` gives the specified decoded form for array.
- `NetDecoder[...][{array1, array2, ...}]` explicitly computes outputs for each of the arrays.
- `NetDecoder[...][..., prop]` can be used to calculate a specific property for the input data.
- `NetDecoder[...][..., "Properties"]` gives the possible properties.
- Possible named decoders include:

<code>"Boolean"</code>	decode 1 and 0 as True and False
<code>"Characters"</code>	decode probability vectors as a string of characters
<code>"Class"</code>	decode probability arrays as class labels
<code>"CTCBeamSearch"</code>	decode sequences of probability vectors trained with a CTCLossLayer
<code>"Function"</code>	decode using a custom function
<code>"Image"</code>	decode a rank-3 array as a 2D image
<code>"Image3D"</code>	decode a rank-4 array as a 3D image
<code>"SubwordTokens"</code>	decode probability vectors as a string of subword tokens
<code>"Tokens"</code>	decode probability vectors as a string of tokens

- A `NetDecoder` object can be attached to an output port of a net by specifying `"port" -> NetDecoder[...]` when constructing the net. Specifying `"port" -> "type"` will create a decoder of the given type and attach it.

Mathematica Code 9.56

Input

```
(* The code aims to create a `NetEncoder` object to convert categorical gender labels ("male" and "female") into numerical values for use in machine learning models. It demonstrates the encoder's functionality by encoding both individual labels and a list of labels, transforming "male" to 1 and "female" to 2. This preprocessing step is essential for converting categorical data into a numerical format suitable for machine learning algorithms: *)

(* Create a class encoder for gender classification with classes "male" and "female": *)
genderEncoder=NetEncoder[{"Class", {"male", "female"}]

(* Encode a single class label: *)
encodedFemale=genderEncoder["male"]
encodedFemale=genderEncoder["female"]

(* Encode a list of class labels: *)
encodedGenderList=genderEncoder[{"male", "female", "female", "male"}]
```

Output

NetEncoder	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td>Type:</td> <td>Class</td> </tr> <tr> <td>Labels:</td> <td>{"male", "female"}</td> </tr> <tr> <td>Output Form:</td> <td>Index</td> </tr> <tr> <td>Dimensions:</td> <td>{ } (scalar)</td> </tr> <tr> <td>Multilabel:</td> <td>False</td> </tr> <tr> <td>Output:</td> <td>index (range: 1..2)</td> </tr> </table>	Type:	Class	Labels:	{"male", "female"}	Output Form:	Index	Dimensions:	{ } (scalar)	Multilabel:	False	Output:	index (range: 1..2)
Type:	Class												
Labels:	{"male", "female"}												
Output Form:	Index												
Dimensions:	{ } (scalar)												
Multilabel:	False												
Output:	index (range: 1..2)												

```
Output 1
Output 2
Output {1, 2, 2, 1}
```

Mathematica Code 9.57

Input

```
(* The code demonstrates the use of a NetEncoder object to encode categorical class labels into numerical values for machine learning applications. It shows the encoder's functionality by converting an individual class label ("dog") to its corresponding numerical value (2) and encoding a list of class labels ({"cat", "dog", "rabbit"}) into numerical values ({1,2,3}). Additionally, it illustrates the encoder's ability to handle a mixed list of class labels, converting ({"dog", "cat", "rabbit", "cat"}) into ({{2,1,3,3,1}}). This encoding process is crucial for preparing categorical data for models that require numerical input: *)

(* Create a class encoder for predefined classes: *)
classEncoder=NetEncoder[{"Class", {"cat", "dog", "rabbit"}]

(* Encode a single class label "dog": *)
encodedClass1=classEncoder["dog"]

(* Encode a list of class labels {"cat", "dog", "rabbit"}: *)
encodedClassList=classEncoder[{"cat", "dog", "rabbit"}]

(* Encode a mixed list of class labels: *)
encodedClassMixedList=classEncoder[{"dog", "cat", "rabbit", "cat", "cat"}]
```

Output

NetEncoder

Type:	Class
Labels:	{"cat", "dog", "rabbit"}
Output Form:	Index
Dimensions:	{ } (scalar)
Multilabel:	False
Output:	index (range: 1..3)

```
Output 2
Output {1,2,3}
Output {2,1,3,3,1}
```

Mathematica Code 9.58

Input

```
(* The code aims to demonstrate the use of a NetEncoder object to encode Boolean values into numerical values (1 for True and 0 for False). It showcases the encoder's functionality by converting both individual Boolean values (True to 1 and False to 0) and a list of Boolean values ({True, False, True}) into their corresponding numerical formats ({1,0,1}). This encoding process is crucial for preparing Boolean data for machine learning models that require numerical input: *)
(* Create a Boolean encoder: *)
booleanEncoder=NetEncoder["Boolean"]

(* Encode a single Boolean value True: *)
encodedBoolean1=booleanEncoder[True]

(* Encode a single Boolean value False: *)
encodedBoolean2=booleanEncoder[False]

(* Encode a list of Boolean values: *)
encodedBooleanList=booleanEncoder[{True,False,True}]
```

Output

NetEncoder

Type:	Boolean
Dimensions:	{ } (scalar)
Output:	boolean

```
Output 1
Output 0
Output {1,0,1}
```

Mathematica Code 9.59

Input

```
(* The code aims to create a `NetDecoder` object to convert numerical outputs representing class probabilities into categorical class labels ("a", "b", and "c"). It demonstrates the decoder's functionality by decoding both a single set of probabilities and multiple sets of probabilities, transforming them into their respective class labels. This preprocessing step is crucial for interpreting the probabilistic predictions of machine learning models, enabling the conversion of numerical outputs into meaningful categorical labels for decision-making: *)
(* Create a decoder for classifying numerical outputs into classes "a", "b", and "c": *)
classDecoder=NetDecoder[{"Class", {"a", "b", "c"}}]
```

```
(* Decode a single set of class probabilities: *)
decodedSingleClass=classDecoder[{0.5,0.1,0.4}]
```

```
(* Decode a list of sets of class probabilities: *)
```

```

decodedClassList=classDecoder[
  {
    {0.9,0.0,0.1},
    {0.9,0.1,0.0},
    {0.1,0.9,0.0},
    {0.0,0.9,0.1},
    {0.0,0.1,0.9},
    {0.1,0.0,0.9}
  }
]

```

Output

Type:	Class
Labels:	{"a", "b", "c"}
Input Depth:	1
Multilabel:	False
Dimensions:	3
Input:	Class

NetDecoder

```

Output      a
Output      {a,a,b,b,c,c}

```

Mathematica Code 9.60

Input

```

(* The code aims to demonstrate the use of a `NetDecoder` object to decode numerical
outputs representing class probabilities into predefined categorical class labels
("cat", "dog", "rabbit"). It shows the decoder's functionality by converting both
a single set of class probabilities (`{0.1,0.7,0.2}`) and multiple sets of class
probabilities (`{{0.1,0.7,0.2},{0.3,0.4,0.3}}`) into their respective class
labels, with "dog" being selected in both cases. This is crucial for interpreting
the probabilistic outputs of machine learning models, enabling the conversion of
numerical predictions into meaningful class labels for further analysis and
decision-making: *)

```

```

(* Create a class decoder for predefined classes: *)
classDecoder=NetDecoder[{"Class", {"cat", "dog", "rabbit"}}]

```

```

(* Decode a single set of class probabilities: *)
decodedClass1=classDecoder[{0.1,0.7,0.2}]

```

```

(* Decode a list of sets of class probabilities: *)
decodedClassList=classDecoder[{{0.1,0.7,0.2}, {0.3,0.4,0.3}}]

```

Output

Type:	Class
Labels:	{"cat", "dog", "rabbit"}
Input Depth:	1
Multilabel:	False
Dimensions:	3
Input:	Class

NetDecoder

```

Output      dog
Output      {dog,dog}

```

Mathematica Code 9.61

Input

```

(* The code aims to demonstrate the use of a `NetDecoder` object to decode numerical
outputs into Boolean values (`True` or `False`). It showcases the decoder's
functionality by converting both individual and lists of probability values
(e.g., 0.8, 0.3, `{0.95, 0.05, 0.6, 0.4}`) as well as explicit binary values (1 and
0, `{1, 0, 1, 0}`) into their corresponding Boolean values. This is essential for
interpreting the outputs of models that predict binary outcomes, enabling the

```

```

conversion of numerical and binary data into meaningful Boolean values for further
analysis and decision-making: *)

(*Create a Boolean decoder*)
booleanDecoder=NetDecoder["Boolean"]

(*Decode a single Boolean value from a probability (close to 1 represents True,
close to 0 represents False): *)
decodedBoolean1=booleanDecoder[0.8]

decodedBoolean2=booleanDecoder[0.3]

(* Decode a list of probabilities: *)
decodedBooleanList=booleanDecoder[{0.95,0.05,0.6,0.4}]

(*Decode a single Boolean value explicitly (1 represents True,0 represents
False)*)
decodedBooleanExplicit1=booleanDecoder[1]

decodedBooleanExplicit2=booleanDecoder[0]

(*Decode a list of explicit Boolean values (1s and 0s) *)
decodedBooleanExplicitList=booleanDecoder[{1,0,1,0}]

```

Output

NetDecoder [Type: Boolean
Input Depth: 0
Input: Boolean]

Output	True
Output	False
Output	{True, False, True, False}
Output	True
Output	False
Output	{True, False, True, False}

Mathematica Code 9.62

```

Input (* The code demonstrates the integration of a NetDecoder for predefined classes
("a","b","c") with a SoftmaxLayer in a neural network, enabling automatic decoding
of the network's output into class labels. It showcases the network's application
to both single and batch inputs, converting the highest probabilities to class
labels. Additionally, the code illustrates how to calculate and retrieve the
probabilities for each class for a single input, and how to compute the entropy
for a batch of inputs, providing a measure of the network's prediction uncertainty.
This setup is crucial for transforming numerical model outputs into interpretable
categorical results: *)

(* Create a class decoder for predefined classes and attach it to a neural
network layer to automatically decode the output: *)
classDecoder=NetDecoder[{ "Class", {"a", "b", "c"} }];
neuralNet=SoftmaxLayer["Output" -> classDecoder]

(* Apply the neural network to a single input: *)
singleInputResult=neuralNet[{1,2,3}]

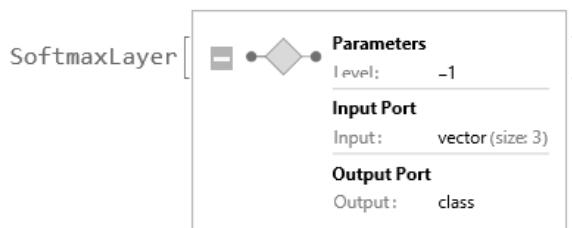
(* Apply the neural network to a batch of inputs: *)
batchInputResult=neuralNet[{{3,2,1},{1,2,3}}]

(* Calculate the probabilities of the decoder for a single input: *)
singleInputProbabilities=neuralNet[{3,2,1}, "Probabilities"]

```

```
(* Calculate the entropy for a batch of inputs: *)
batchInputEntropy=neuralNet[{{1,1,1},{1,3,9}),"Entropy"]
```

Output



Output

c

Output {a,c}

Output <| a->0.665241,b->0.244728,c->0.0900306 |>

Output {1.09861,0.0203172}

Unit 9.3

Comparison of Some Activation Functions

Activation functions play a crucial role in neural networks by introducing non-linearity, which enables the networks to learn complex patterns and representations. This unit provides a detailed comparison of several commonly used activation functions (ReLU, ELU, SELU, GELU, Swish, HardSwish, Mish, SoftPlus, HardTanh, HardSigmoid, Sigmoid, Tanh).

Mathematica Code 9.63

```

Input      (* The code aims to generate synthetic training data using a Gaussian-modulated
           exponential function with added noise and systematically evaluate the performance
           of various activation functions in a neural network. It achieves this by defining
           a neural network architecture with three hidden layers, each followed by one of
           the specified activation functions, and training the network on the generated data.
           During training, the GradientsRMS property is monitored to ensure proper training
           dynamics, and results are recorded for each activation function. This approach
           provides insights into how different activation functions impact the training
           process and performance of the neural network: *)
(* Generate the training data using a Gaussian-modulated exponential function with
   added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;
(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];
(* Set up array for activation functions: *)
activationFunctions={
  "ReLU","ELU","SELU",
  "GELU","Swish","HardSwish",
  "Mish","SoftPlus","HardTanh",
  "HardSigmoid","Sigmoid",Tanh
};

(* Loop through each activation function, train a neural network, and record the
   results: *)
results=Table[
  Module[
    {net,trainedNet},
    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        (* First hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Second hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Third hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Output layer with linear activation function: *)
        LinearLayer[1]
      }
    ];
    (* Train the network on training data: *)
    trainedNet=NetTrain[

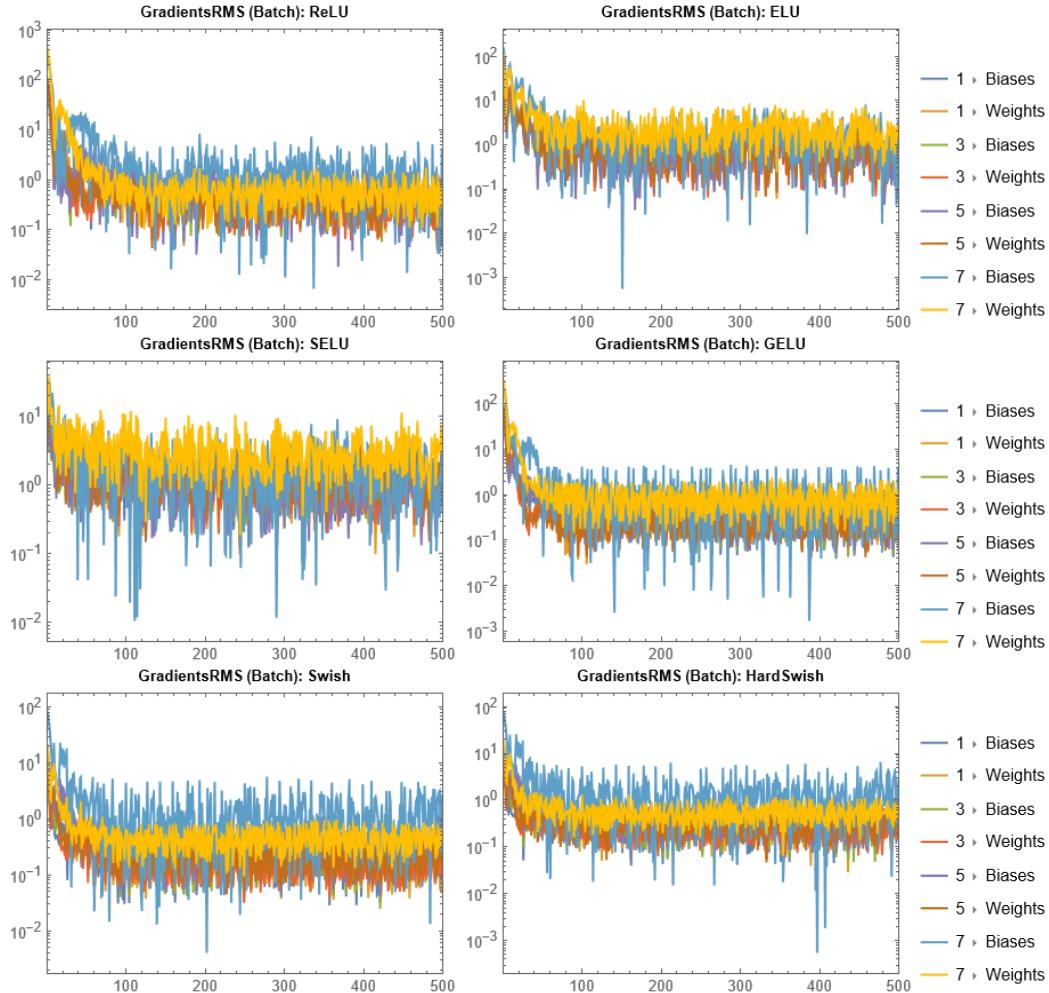
```

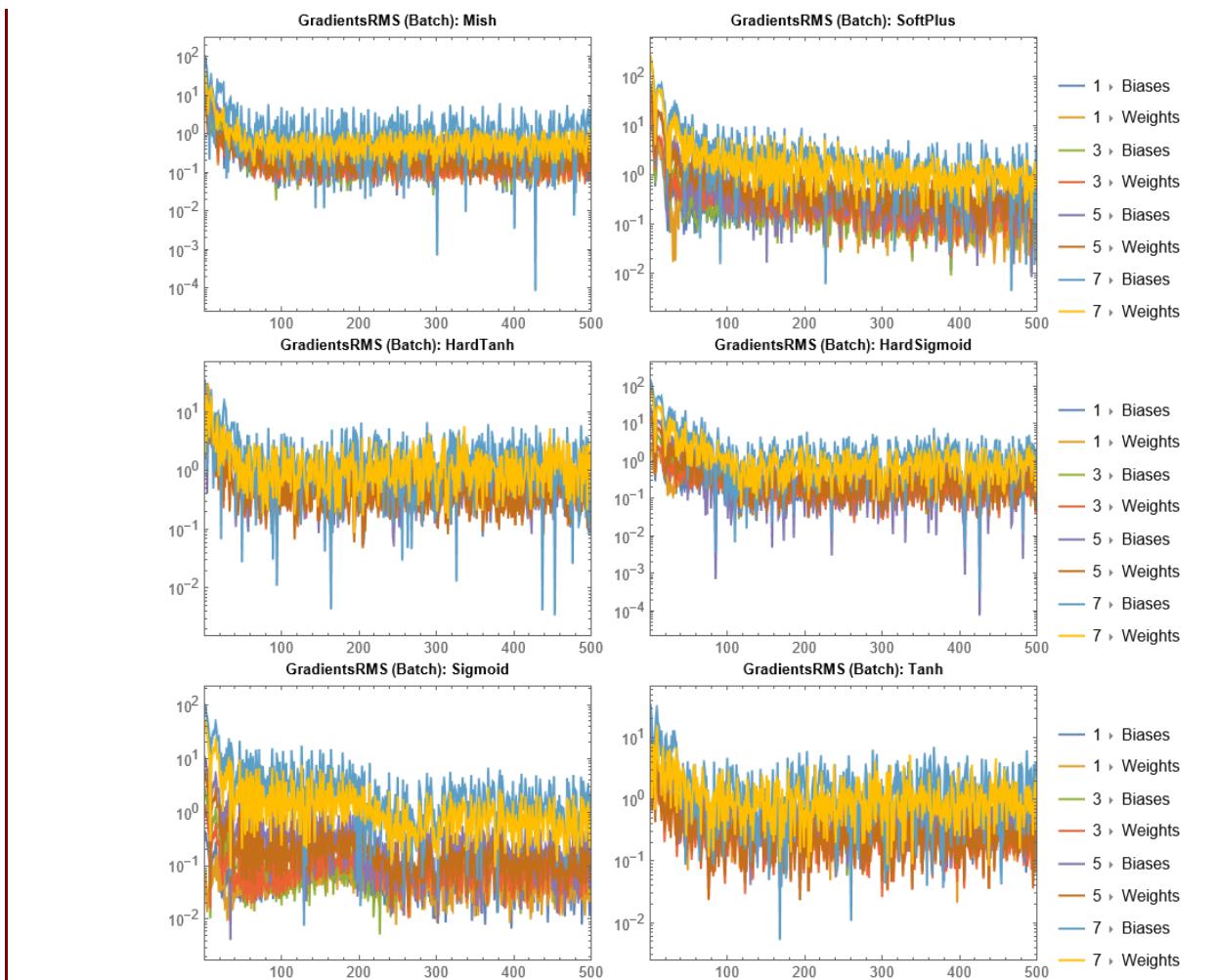
```

net,
trainingData,
(* Monitor training progress using GradientsRMS property: *)
<|
  "Property"->"GradientsRMS",
  "Form"->"EvolutionPlot",
  "PlotOptions"->{PlotLabel->Style[Row[{ "GradientsRMS (Batch):",
",functions}],10,Bold],ImageSize->300},
  "Interval"->"Batch"
|>,
(* Specify mean squared loss as the training objective: *)
LossFunction->MeanSquaredLossLayer[],
(* Set the batch size for training: *)
BatchSize->64,
(* Learning rate for "ADAM": *)
LearningRate->0.01,
(* Use "ADAM" as the optimization method: *)
Method->"ADAM",
(* Maximum number of training iterations: *)
MaxTrainingRounds->50
],
],
(* Iterate over the grid of activation functions: *)
{functions,activationFunctions}
]

```

Output



**Mathematica Code 9.64**

Input (* The above and the following codes are largely similar, both aiming to generate synthetic training data, train a neural network using various activation functions, and monitor the training process. However, they differ in how they monitor the training progress: the above code monitors the GradientsRMS property at the "Batch" interval, providing a more granular view of the gradient changes, while the following code monitors it at the "Rounds" interval, offering a broader view: *)

```
(* Generate the training data using a Gaussian-modulated exponential function with
   added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Set up array for activation functions: *)
activationFunctions={

  "ReLU", "ELU", "SELU",
  "GELU", "Swish", "HardSwish",
  "Mish", "SoftPlus", "HardTanh",
```

```

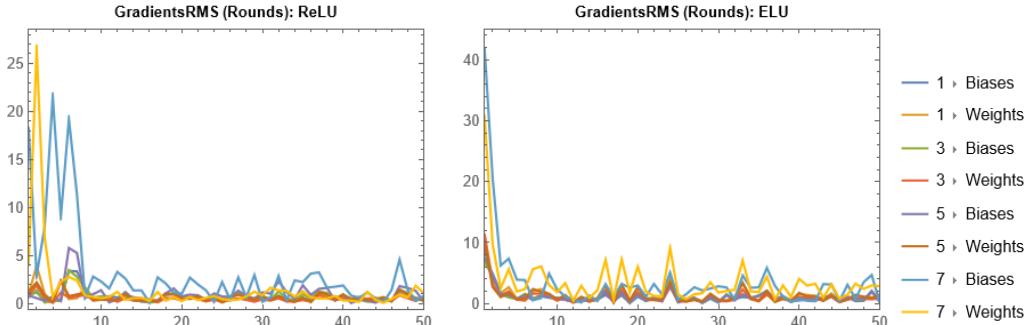
    "HardSigmoid", "Sigmoid", Tanh
  } ;
(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
Module[
{net,trainedNet}]

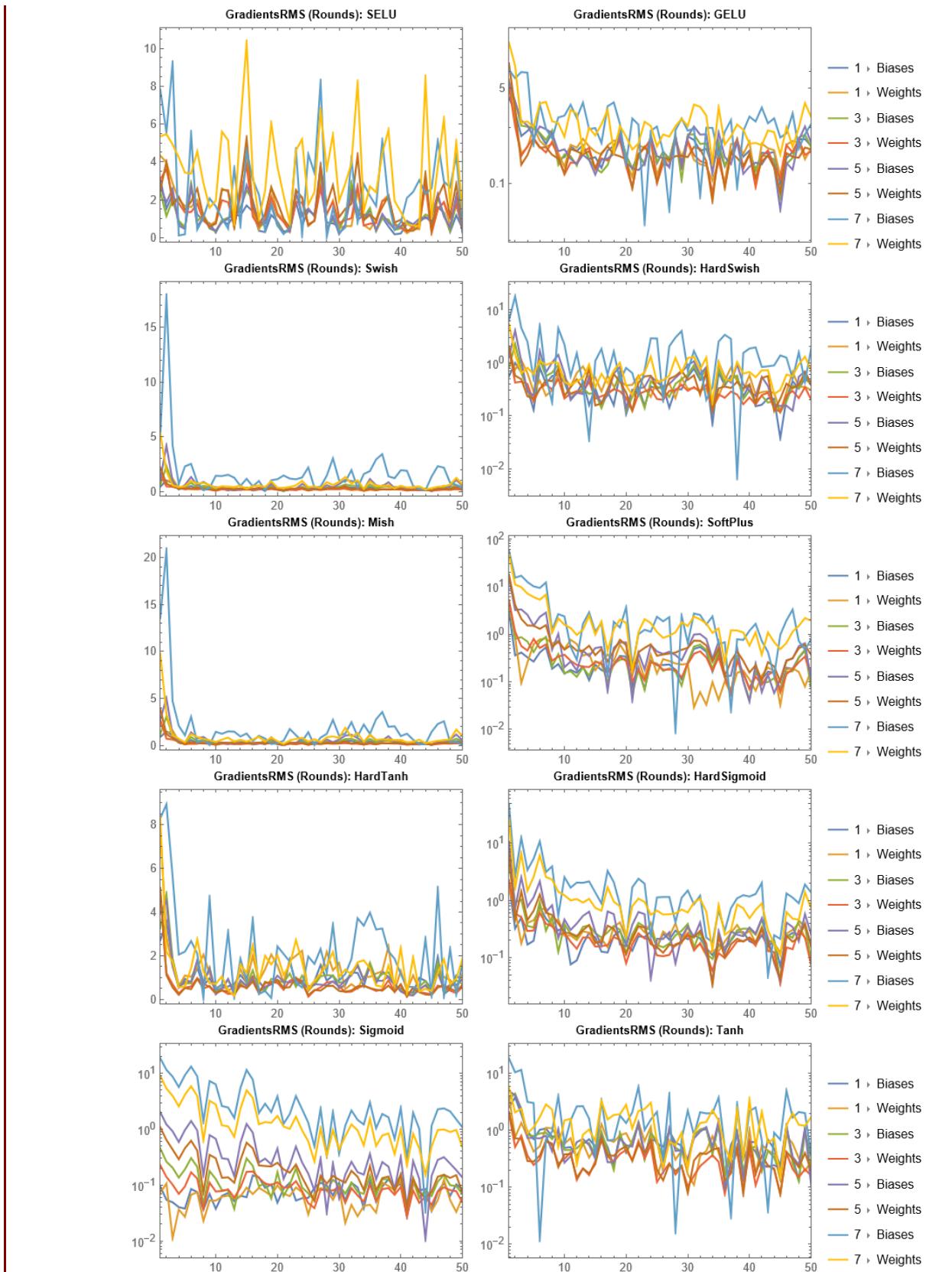
(* Define a simple neural network structure: *)
net=NetChain[
{
  (* First hidden layer with specified activation function: *)
  LinearLayer[10], ElementwiseLayer[functions],
  (* Second hidden layer with specified activation function: *)
  LinearLayer[10], ElementwiseLayer[functions],
  (* Third hidden layer with specified activation function: *)
  LinearLayer[10], ElementwiseLayer[functions],
  (* Output layer with linear activation function: *)
  LinearLayer[1]
};
];

(* Train the network on training data: *)
trainedNet=NetTrain[
  net,
  trainingData,
  (* Monitor training progress using GradientsRMS (Rounds) property: *)
<|
  "Property"->"GradientsRMS",
  "Form"->"EvolutionPlot",
  "PlotOptions"->{ PlotLabel->Style[Row[{ "GradientsRMS (Rounds):",
",functions}],10,Bold],ImageSize->300},
  "Interval"->"Rounds"
|>,
(* Specify mean squared loss as the training objective: *)
LossFunction->MeanSquaredLossLayer[],
(* Set the batch size for training: *)
BatchSize->64,
(* Learning rate for "ADAM": *)
LearningRate->0.01,
(* Use "ADAM" as the optimization method: *)
Method->"ADAM",
(* Maximum number of training iterations: *)
MaxTrainingRounds->50
]
],
(* Iterate over the grid of activation functions: *)
{functions,activationFunctions}
]

```

Output





Mathematica Code 9.65

```

Input      (* In this case, training progress is monitored using the WeightsRMS property at
          the "Rounds" interval. By recording and analyzing the results for each activation
          function, the code provides insights into their impact on the training dynamics
          and performance of the neural network: *)

(* Generate the training data using a Gaussian-modulated exponential function with
   added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Set up arrays for hyperparameter tuning: Activation Functions  *)
activationFunctions={
  "ReLU","ELU","SELU",
  "GELU","Swish","HardSwish",
  "Mish","SoftPlus","HardTanh",
  "HardSigmoid","Sigmoid",Tanh
} ;

(* Loop through each activation function, train a neural network, and record the
   results: *)
results=Table[
  Module[
    {net,trainedNet},

    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        (* First hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Second hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Third hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Output layer with linear activation function: *)
        LinearLayer[1]
      }
    ];

    (* Train the network on training data: *)
    trainedNet=NetTrain[
      net,
      trainingData,
      (* Monitor training progress using WeightsRMS (Rounds) property: *)
      <|
        "Property"->"WeightsRMS",
        "Form"->"EvolutionPlot",
        "PlotOptions"->{ PlotLabel->Style[Row[{ "WeightsRMS (Rounds):",
          ",functions}],10,Bold],
          ImageSize->300},
        "Interval"->"Rounds"
      |>,
      (* Specify mean squared loss as the training objective: *)
      LossFunction->MeanSquaredLossLayer[],

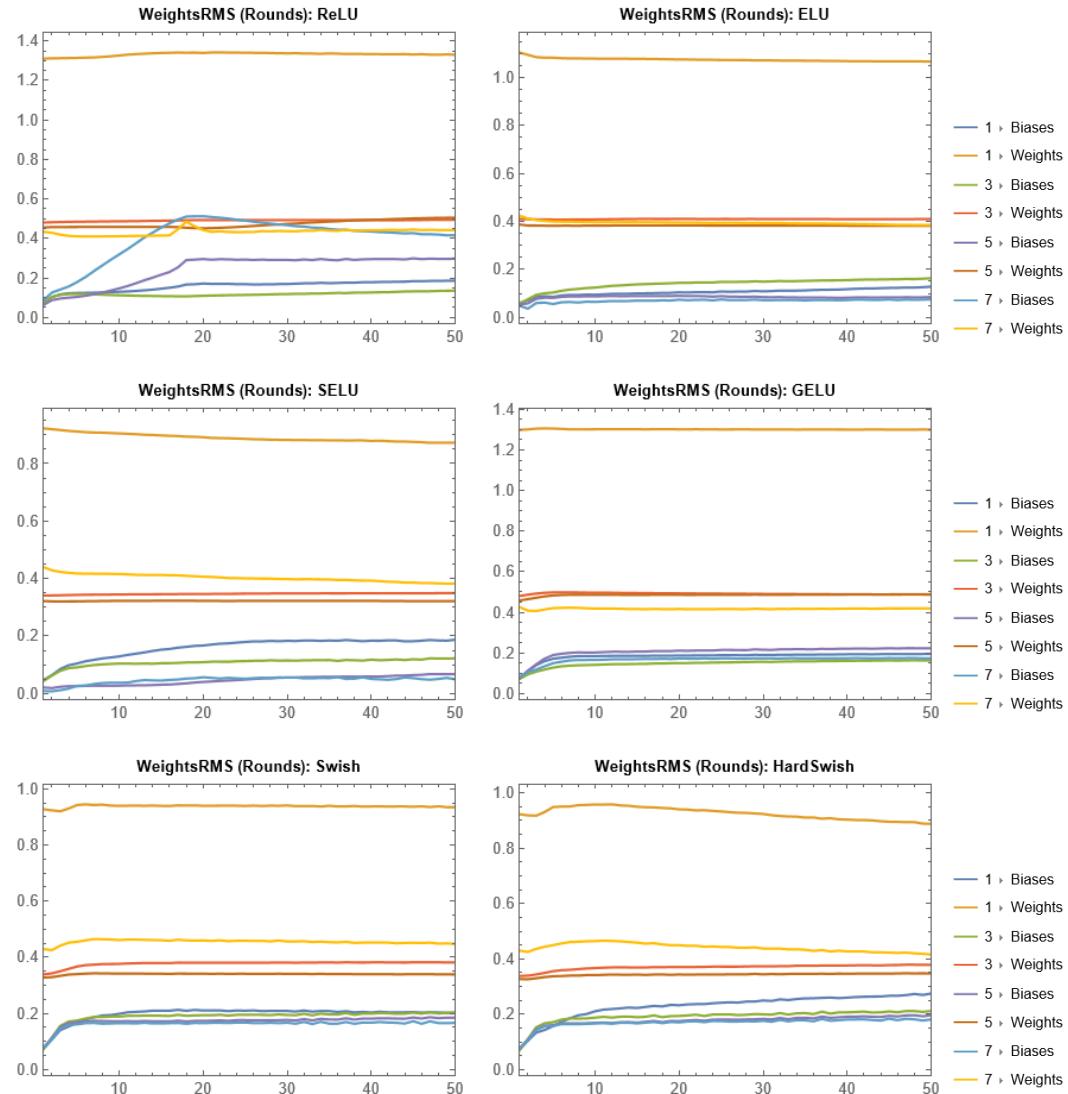

```

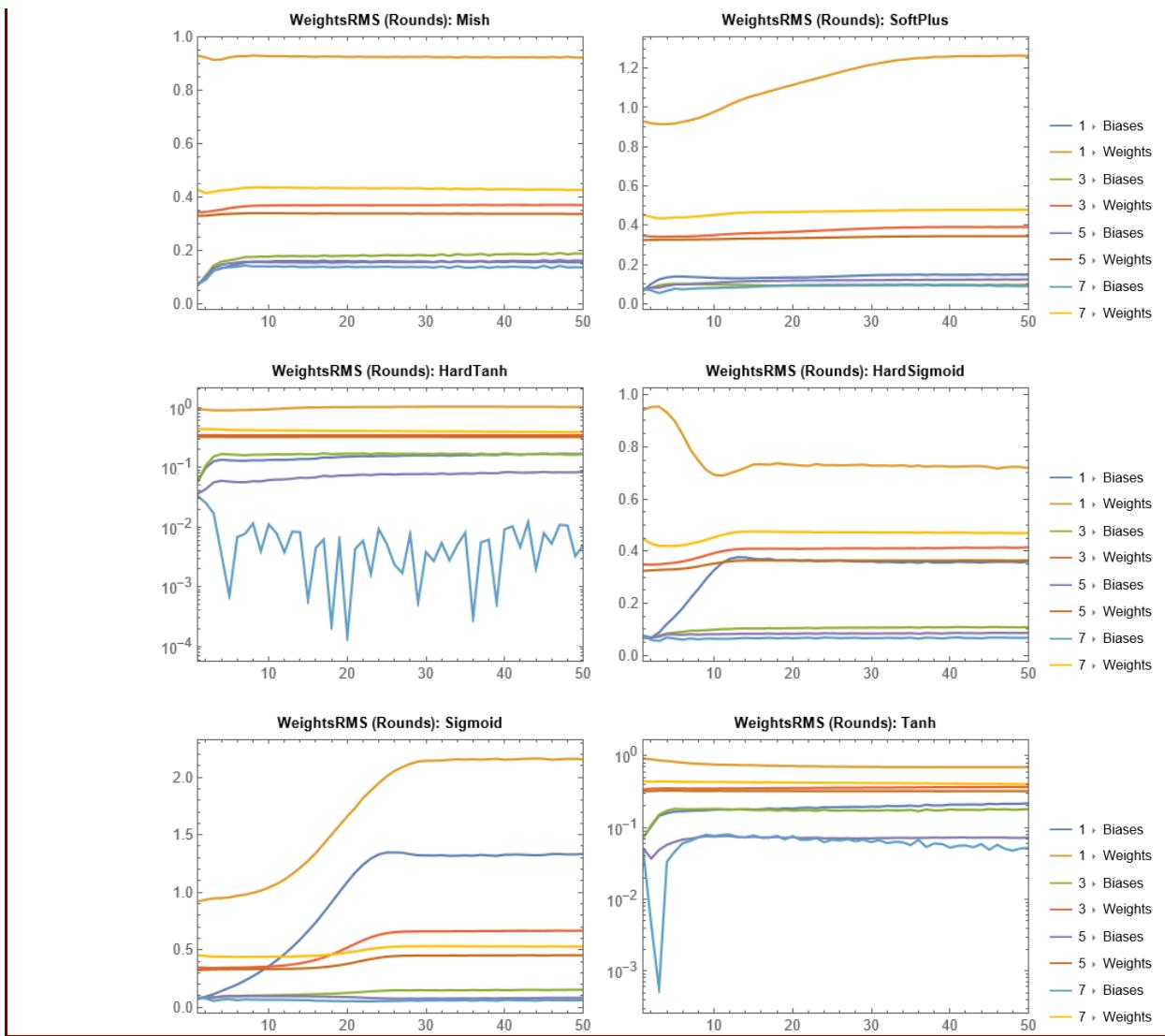
```

(* Set the batch size for training: *)
BatchSize->64,
(* Learning rate for "ADAM": *)
LearningRate->0.01,
(* Use "ADAM" as the optimization method: *)
Method->"ADAM",
(* Maximum number of training iterations: *)
MaxTrainingRounds->50
],
]

(* Iterate over the grid of activation functions: *)
{functions,activationFunctions}
]

```

Output

**Mathematica Code 9.66**

Input (* In this case, training progress is monitored using the RoundLoss property at the "Rounds" interval. By recording and analyzing the results for each activation function, the code provides insights into their impact on the training dynamics and performance of the neural network, aiding in the selection of suitable activation functions for future designs: *)

```
(* Generate the training data using a Gaussian-modulated exponential function
with added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

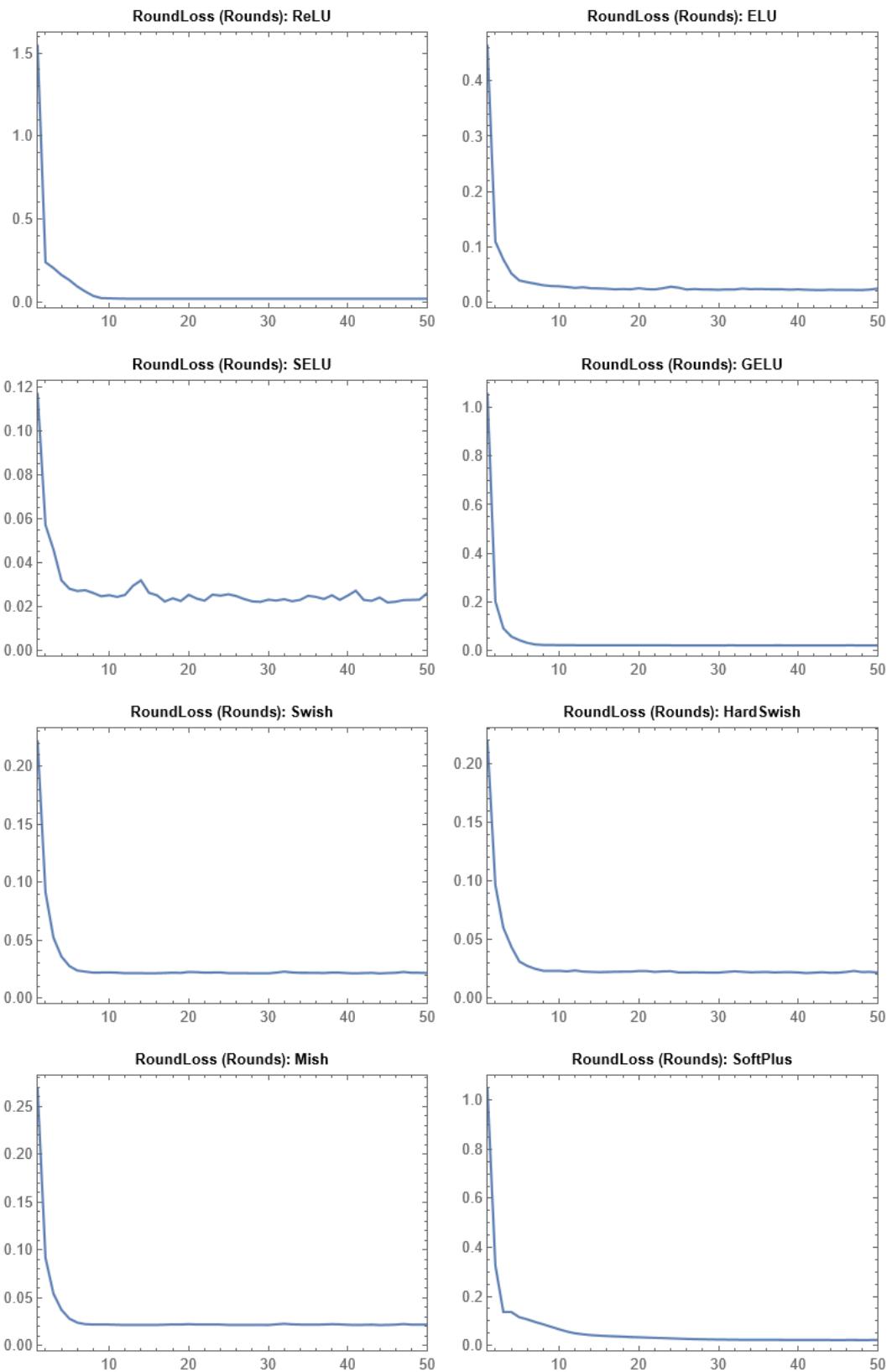
(* Set up arrays for hyperparameter tuning: Activation Functions *)
activationFunctions={
```

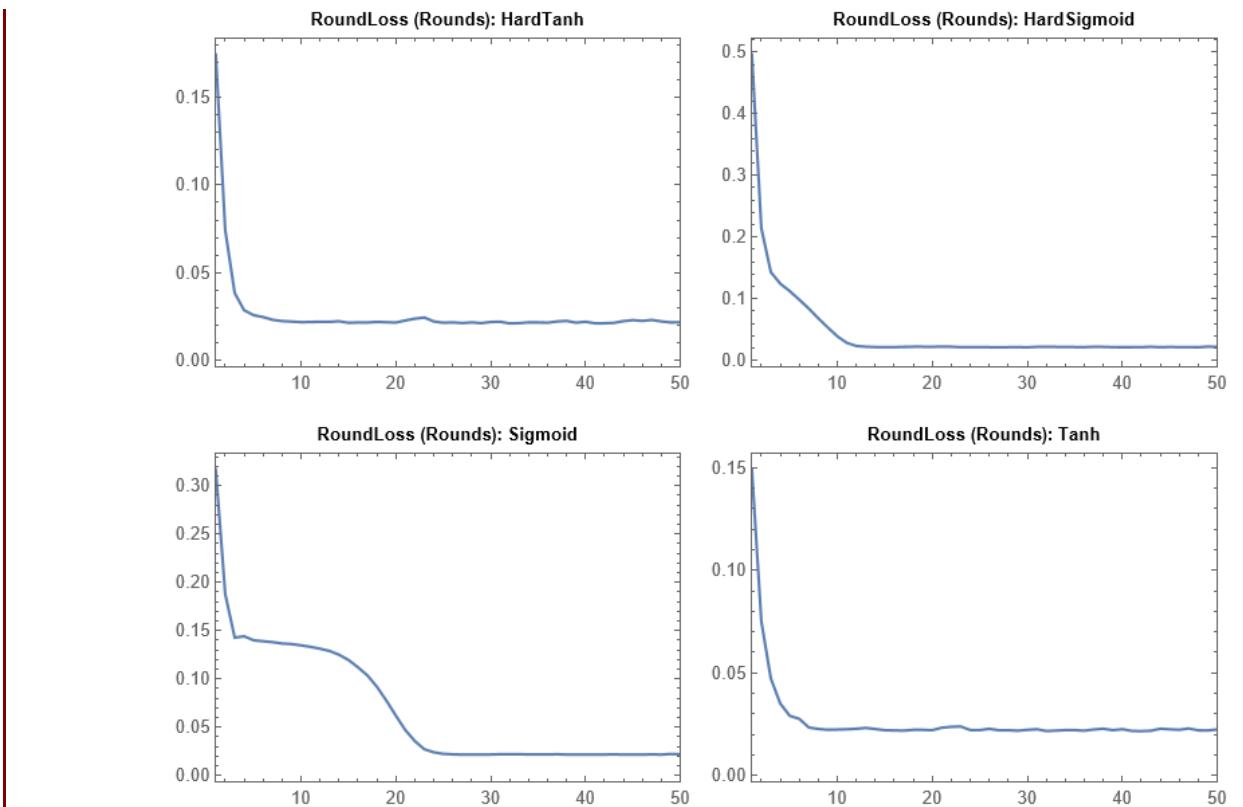
```
"ReLU","ELU","SELU",
"GELU","Swish","HardSwish",
"Mish","SoftPlus","HardTanh",
"HardSigmoid","Sigmoid",Tanh} ;

(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
  Module[
    {net,trainedNet},

    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        (* First hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Second hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Third hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Output layer with linear activation function: *)
        LinearLayer[1]
      }
    ];

    (* Train the network on training data: *)
    trainedNet=NetTrain[
      net,
      trainingData,
      (* Monitor training progress using RoundLoss property: *)
      <|
        "Property"->"RoundLoss",
        "Form"->"EvolutionPlot",
        "PlotOptions"->{PlotLabel->Style[Row[{"RoundLoss (Rounds):",
          functions}],10,Bold],
          ImageSize->300},
        "Interval"->"Rounds"
      |>,
      (* Specify mean squared loss as the training objective: *)
      LossFunction->MeanSquaredLossLayer[],
      (* Set the batch size for training: *)
      BatchSize->64,
      (* Learning rate for "ADAM": *)
      LearningRate->0.01,
      (* Use "ADAM" as the optimization method: *)
      Method->"ADAM",
      (* Maximum number of training iterations: *)
      MaxTrainingRounds->50
    ],
    ],
    (* Iterate over the grid of activation functions: *)
    {functions,activationFunctions}
  ]
]
```

Output

**Mathematica Code 9.67**

```

Input (* In this case, training progress is monitored using the ValidationLoss property
at the "Rounds" interval. By recording and analyzing the results for each activation
function, the code provides insights into their impact on the training dynamics
and performance of the neural network, aiding in the selection of suitable
activation functions for future designs: *)

(* Generate the training data using a Gaussian-modulated exponential function
with added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Validation data with added noise: *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];

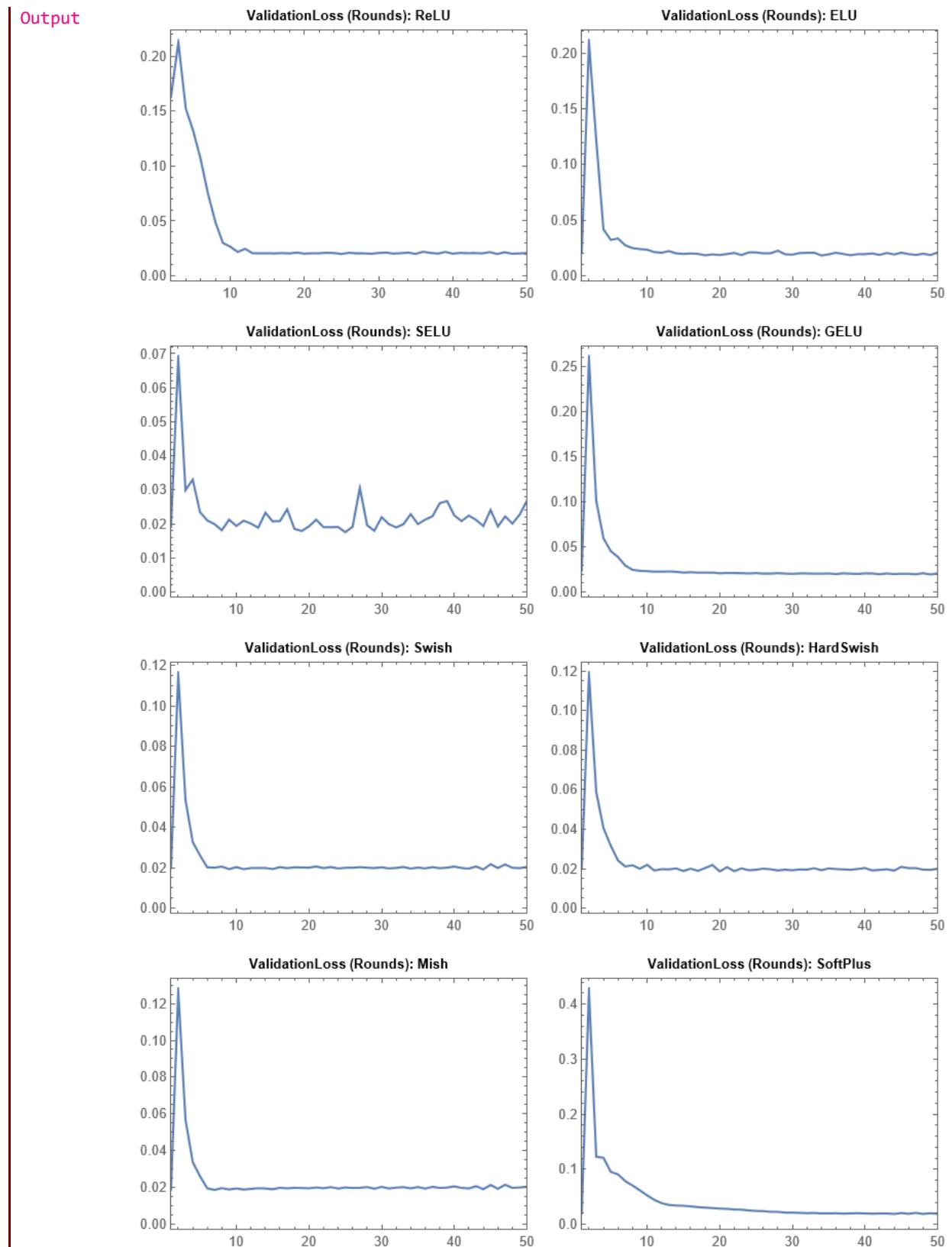
(* Set up arrays for hyperparameter tuning: Activation Functions: *)
activationFunctions={
  "ReLU", "ELU", "SELU",
  "GELU", "Swish", "HardSwish",
  "Mish", "SoftPlus", "HardTanh",
  "HardSigmoid", "Sigmoid", Tanh
} ;

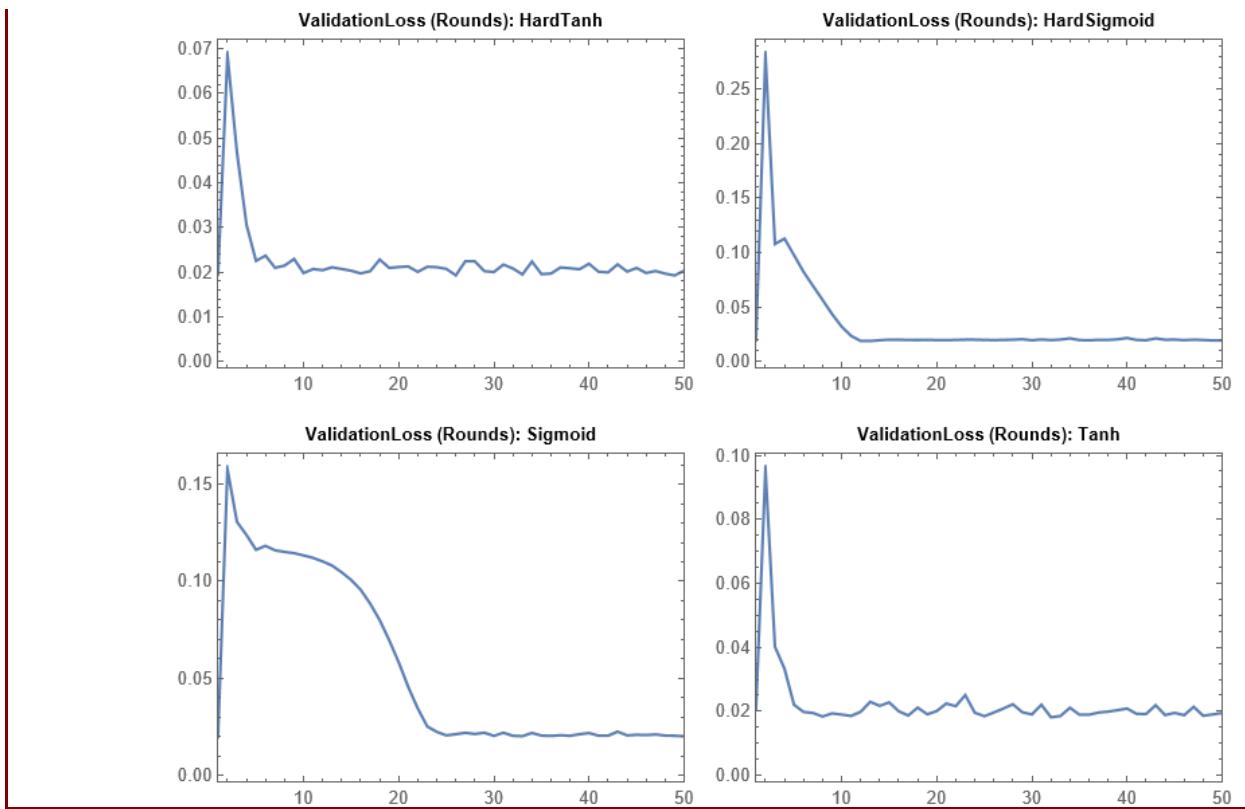
```

```
(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
  Module[
    {net,trainedNet},

    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        (* First hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Second hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Third hidden layer with specified activation function: *)
        LinearLayer[10],ElementwiseLayer[functions],
        (* Output layer with linear activation function: *)
        LinearLayer[1]
      }
    ];
    (* Train the network on training data and validate it: *)
    trainedNet=NetTrain[
      net,
      trainingData,
      (* Monitor training progress using ValidationLoss (Rounds) property: *)
      <|
        "Property"->"ValidationLoss",
        "Form"->"EvolutionPlot",
        "PlotOptions"->{PlotLabel->Style[Row[{"ValidationLoss (Rounds):",
          functions}],10,Bold],
          ImageSize->300},
        "Interval"->"Rounds"
      |>,
      ValidationSet->validationData,
      (* Specify mean squared loss as the training objective: *)
      LossFunction->MeanSquaredLossLayer[],

      (* Set the batch size for training: *)
      BatchSize->64,
      (* Learning rate for "ADAM": *)
      LearningRate->0.01,
      (* Use "ADAM" as the optimization method: *)
      Method->"ADAM",
      (* Maximum number of training iterations: *)
      MaxTrainingRounds->50
    ],
    (* Iterate over the grid of activation functions: *)
    {functions,activationFunctions}
  ]
]
```



**Mathematica Code 9.68**

Input (* The code aims to generate synthetic training data using a Gaussian-modulated exponential function with added noise, and systematically evaluate various activation functions in a neural network by training and validating the network with each function. It defines a neural network architecture with three hidden layers and an output layer, each hidden layer using one of the specified activation functions. The network is trained on the generated data using the ADAM optimization method with mean squared loss as the objective, and its performance is validated using a separate validation data set. The training progress is monitored using mean square error (MSE) measurements, and results, including validation MSE and loss plots, are recorded for each activation function. This process provides insights into the impact of different activation functions on the training dynamics and performance of neural networks, aiding in the selection of appropriate activation functions for future designs: *)

```
(* Generate the training data using a Gaussian-modulated exponential function
with added noise: *)
dataFunction[x_]:=Exp[-x^2];
noiseLevel=0.15;

(* Training data with added noise: *)
trainingData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.01}
];

(* Validation data with added noise: *)
validationData=Table[
  x->dataFunction[x]+RandomVariate[NormalDistribution[0,noiseLevel]],
  {x,-3,3,0.1}
];
```

```

(* Set up array for activation functions *)
activationFunctions={

    "ReLU", "ELU", "SELU",
    "GELU", "Swish", "HardSwish",
    "Mish", "SoftPlus", "HardTanh",
    "HardSigmoid", "Sigmoid", Tanh} ;

(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
    Module[
        {net,trainedNet,validationMSE},

        (* Define a simple neural network structure: *)
        net=NetChain[
            {
                (* First hidden layer with specified activation function: *)
                LinearLayer[10], ElementwiseLayer[functions],
                (* Second hidden layer with specified activation function: *)
                LinearLayer[10], ElementwiseLayer[functions],
                (* Third hidden layer with specified activation function: *)
                LinearLayer[10], ElementwiseLayer[functions],
                (* Output layer with linear activation function: *)
                LinearLayer[1]
            }
        ];
        (* Train the network on training data and validate it: *)
        trainedNet=NetTrain[
            net,
            trainingData,
            All,
            ValidationSet->validationData,
            TrainingProgressMeasurements->{"MeanSquare"},

            (* Specify mean squared loss as the training objective: *)
            LossFunction->MeanSquaredLossLayer[],

            (* Set the batch size for training: *)
            BatchSize->64,
            (* Learning rate for "ADAM": *)
            LearningRate->0.01,
            (* Use "ADAM" as the optimization method: *)
            Method->"ADAM",
            (* Maximum number of training iterations: *)
            MaxTrainingRounds->50
        ];
        (* Calculate MSE on validation data for model evaluation: *)
        (* Extract validation MSE from training results: *)
        validationMSE=Values[trainedNet["ValidationMeasurements"]][[[2]]];
        (* Get the loss plot from training results: *)
        lossplot=trainedNet["LossPlot"];
        (* Return validation MSE,trained network, and loss plot: *)
        {validationMSE,trainedNet,lossplot}
    ],
    (* Iterate over the grid of activation functions: *)
    {functions,activationFunctions}
];
(* Create a table of activation functions and their corresponding validation MSE: *)
];
Table[

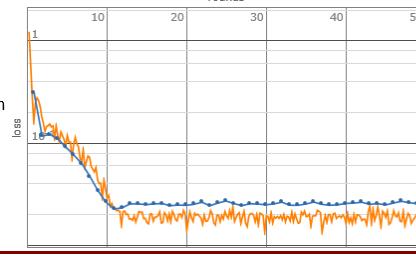
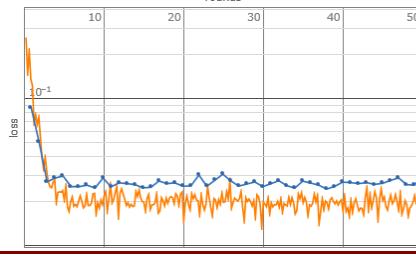
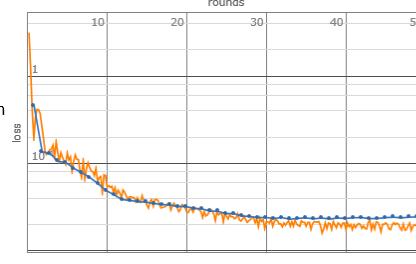
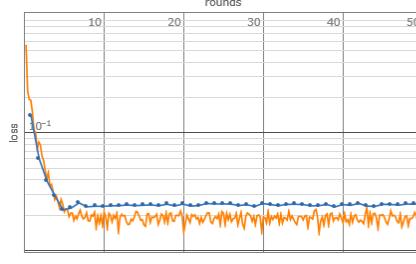
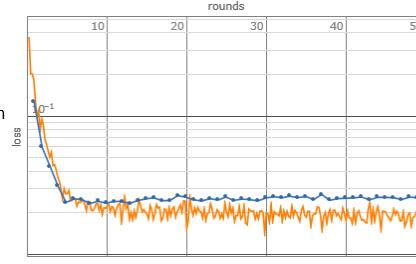
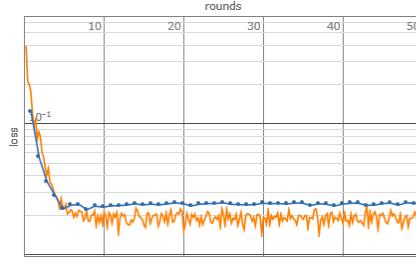
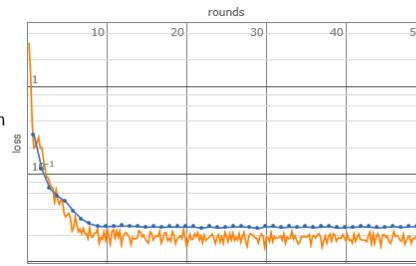
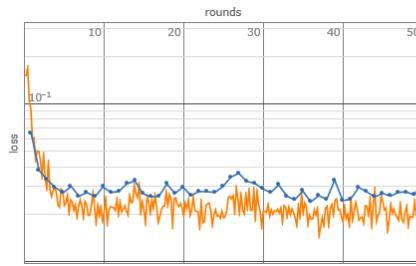
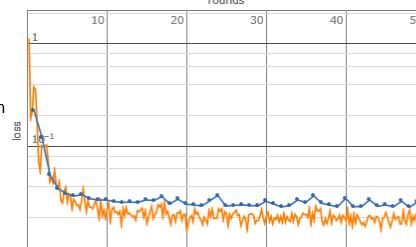
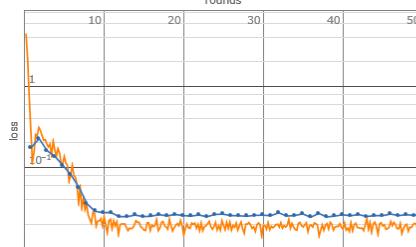
```

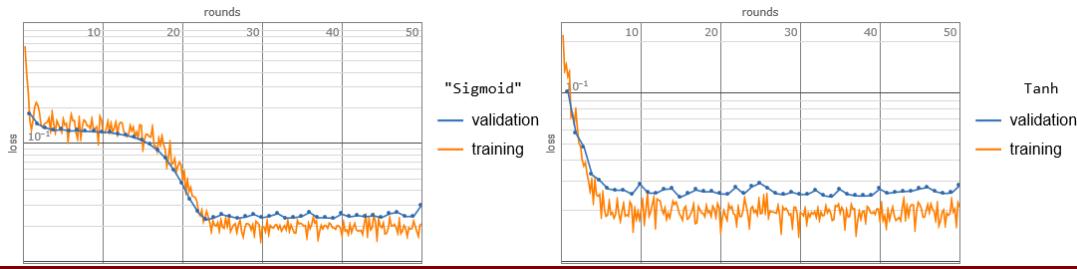
```

{activationFunctions[[i]],results[[i]][[1]]},
{i,1,12}]
(* Create overlays of loss plots and activation functions: *)
Table[
  Overlay[{results[[i]][[3]],activationFunctions[[i]]},Alignment->{0.9,0.4}],
  {i,1,12}
]
Output {{ReLU,0.0275686},{ELU,0.0247876},{SELU,0.0326586},{GELU,0.0247986},{Swish,0.0255731},{HardSwish,0.0261176},{Mish,0.0253922},{SoftPlus,0.0254517},{HardTanh,0.0303038},{HardSigmoid,0.0291263},{Sigmoid,0.030037},{Tanh,0.0281623}}

```

Output



**Mathematica Code 9.69**

```

Input (* The code aims to generate synthetic training data points within a unit disk,
label them based on their distance from the origin, and visualize the training set
to inspect the data distribution. It then evaluates the performance of various
activation functions in a neural network by training the network using each
function. The neural network, defined with two hidden layers and an output layer
using a logistic sigmoid activation, is trained on the synthetic data using the
ADAM optimization method, with training progress monitored through the GradientsRMS
property. The results, including the evolution of GradientsRMS, are recorded for
each activation function to compare their effectiveness, providing insights into
their impact on the training dynamics and classification performance, thus aiding
in the selection of suitable activation functions for future neural network designs:
*)

(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less
than 0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Visualize the synthetic training set with different colors for each class: *)
ListPlot[
{
  Pick[trainpoints,trainlabels,True],
  Pick[trainpoints,trainlabels,False]
},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},
PlotLegends->{"Class 1","Class 2"}
]

(* Set up array for activation functions *)
activationFunctions={

  "ReLU","ELU","SELU",
  "GELU","Swish","HardSwish",
  "Mish","SoftPlus","HardTanh",
  "HardSigmoid","Sigmoid",Tanh
} ;

(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
  Module[
    {net,trainedNet},

```

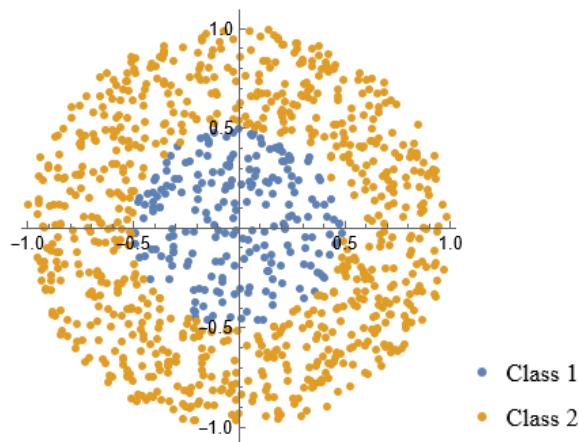
```

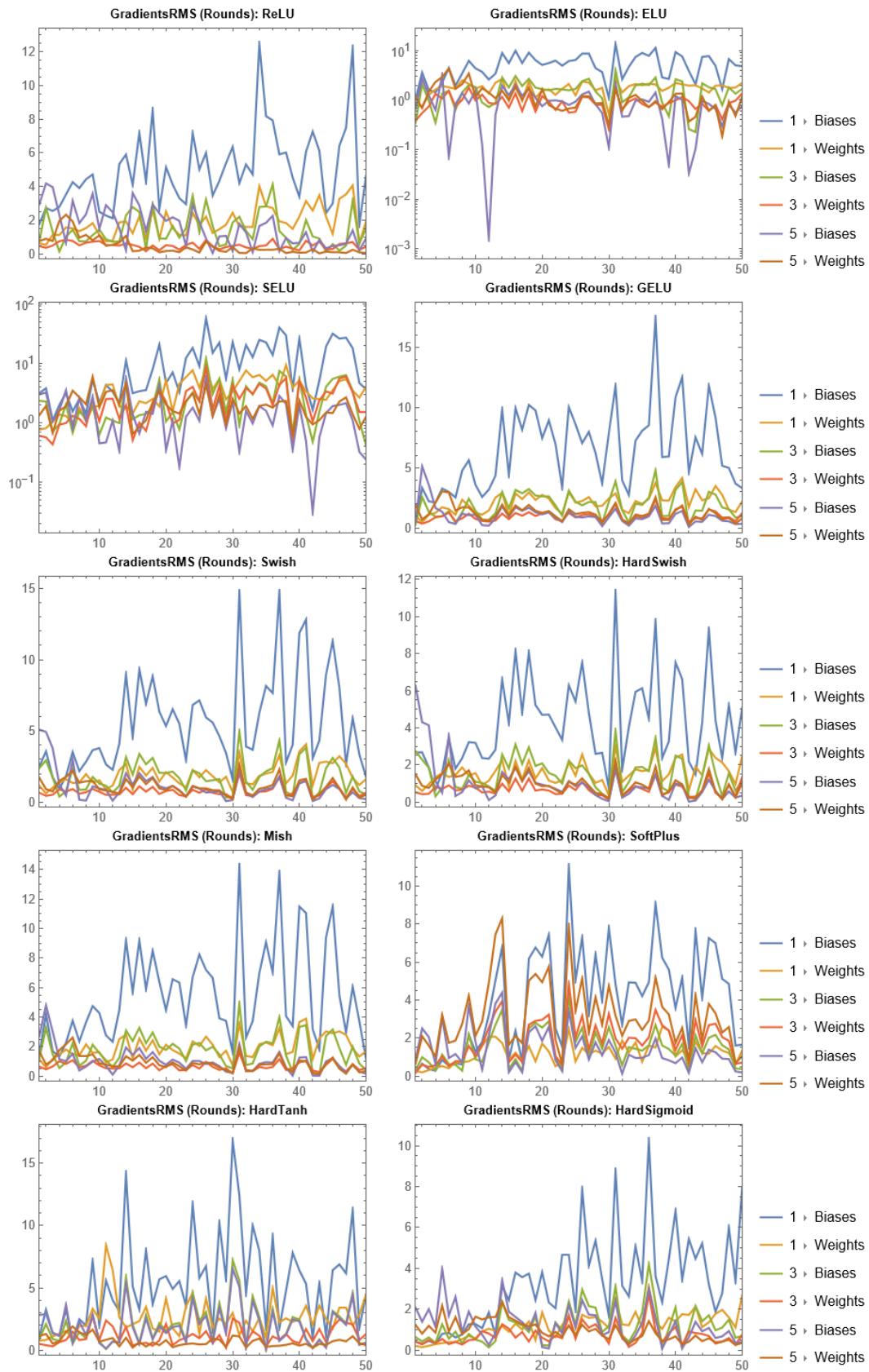
(* Define a simple neural network structure: *)
net=NetChain[
  {
    (* First hidden layer with specified activation function: *)
    LinearLayer[5],ElementwiseLayer[functions],
    (* Second hidden layer with specified activation function: *)
    LinearLayer[5],ElementwiseLayer[functions],
    (* Output layer with logistic sigmoid activation function: *)
    LinearLayer[],ElementwiseLayer[LogisticSigmoid]
  },
  "Output"->NetDecoder["Boolean"]
];

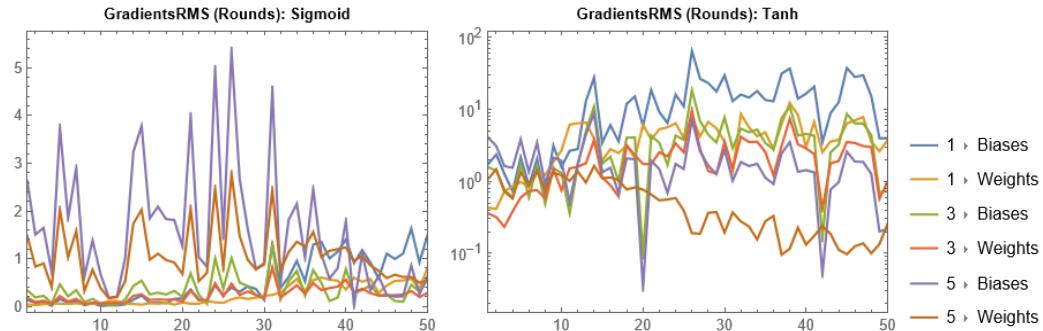
(* Train the network on training data: *)
trainedNet=NetTrain[
  net,
  trainingData,
  (* Monitor training progress using GradientsRMS property: *)
  <|
    "Property"->"GradientsRMS",
    "Form"->"EvolutionPlot",
    "PlotOptions"->{ PlotLabel->Style[Row[{ "GradientsRMS (Rounds):",
      "functions}], 10, Bold], ImageSize->300},
    "Interval"->"Rounds"
  |>,
  (* Set the batch size for training: *)
  BatchSize->64,
  (* Learning rate for "ADAM": *)
  LearningRate->0.01,
  (* Use "ADAM" as the optimization method: *)
  Method->"ADAM",
  (* Maximum number of training iterations: *)
  MaxTrainingRounds->50
]
],
(* Iterate over the grid of activation functions: *)
{functions,activationFunctions}
]

```

Output



Output

**Mathematica Code 9.70**

Input

```
(* The above and the following codes are largely similar, both aiming to generate synthetic training data, train a neural network using various activation functions, and monitor the training process. In this case, training progress is monitored using the WeightsRMS property at the "Rounds" interval: *)

(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less than 0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Visualize the synthetic training set with different colors for each class: *)
ListPlot[
{
  Pick[trainpoints,trainlabels,True],
  Pick[trainpoints,trainlabels,False]
},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},
PlotLegends->{"Class 1","Class 2"}
]

(* Set up array for activation functions: *)
activationFunctions={
  "ReLU","ELU","SELU",
  "GELU","Swish","HardSwish",
  "Mish","SoftPlus","HardTanh",
  "HardSigmoid","Sigmoid",Tanh
} ;

(* Loop through each activation function, train a neural network, and record the results: *)
results=Table[
  Module[
    {net,trainedNet},
    (* Define a simple neural network structure: *)
    net=NetChain[
      {
        (* First hidden layer with specified activation function: *)

```

```

LinearLayer[5],ElementwiseLayer[functions],
(* Second hidden layer with specified activation function: *)
LinearLayer[5],ElementwiseLayer[functions],
(* Output layer with logistic sigmoid activation function: *)
LinearLayer[],ElementwiseLayer[LogisticSigmoid]
},
"Output"->NetDecoder["Boolean"]
];

(* Train the network on training data: *)
trainedNet=NetTrain[
  net,
  trainingData,

  (* Monitor training progress using WeightsRMS property: *)
  <|
    "Property"->"WeightsRMS",
    "Form"->"EvolutionPlot",
    "PlotOptions"->{ PlotLabel->Style[Row[{ "WeightsRMS (Rounds):",
      "functions}], 10, Bold],
      ImageSize->300},
    "Interval"->"Rounds"
  |>,

  (* Set the batch size for training: *)
  BatchSize->64,

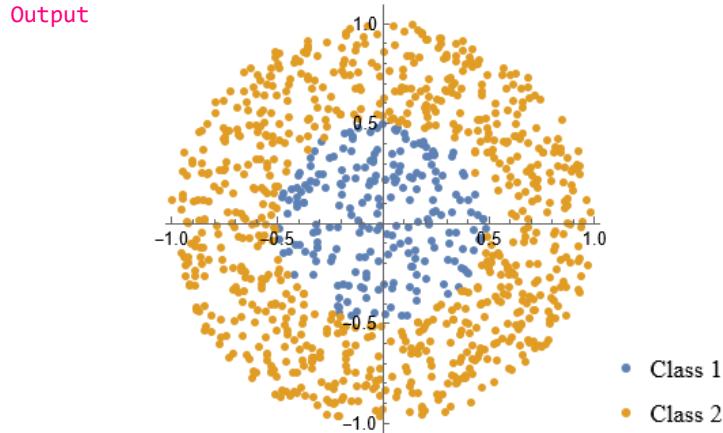
  (* Learning rate for "ADAM": *)
  LearningRate->0.01,

  (* Use "ADAM" as the optimization method: *)
  Method->"ADAM",

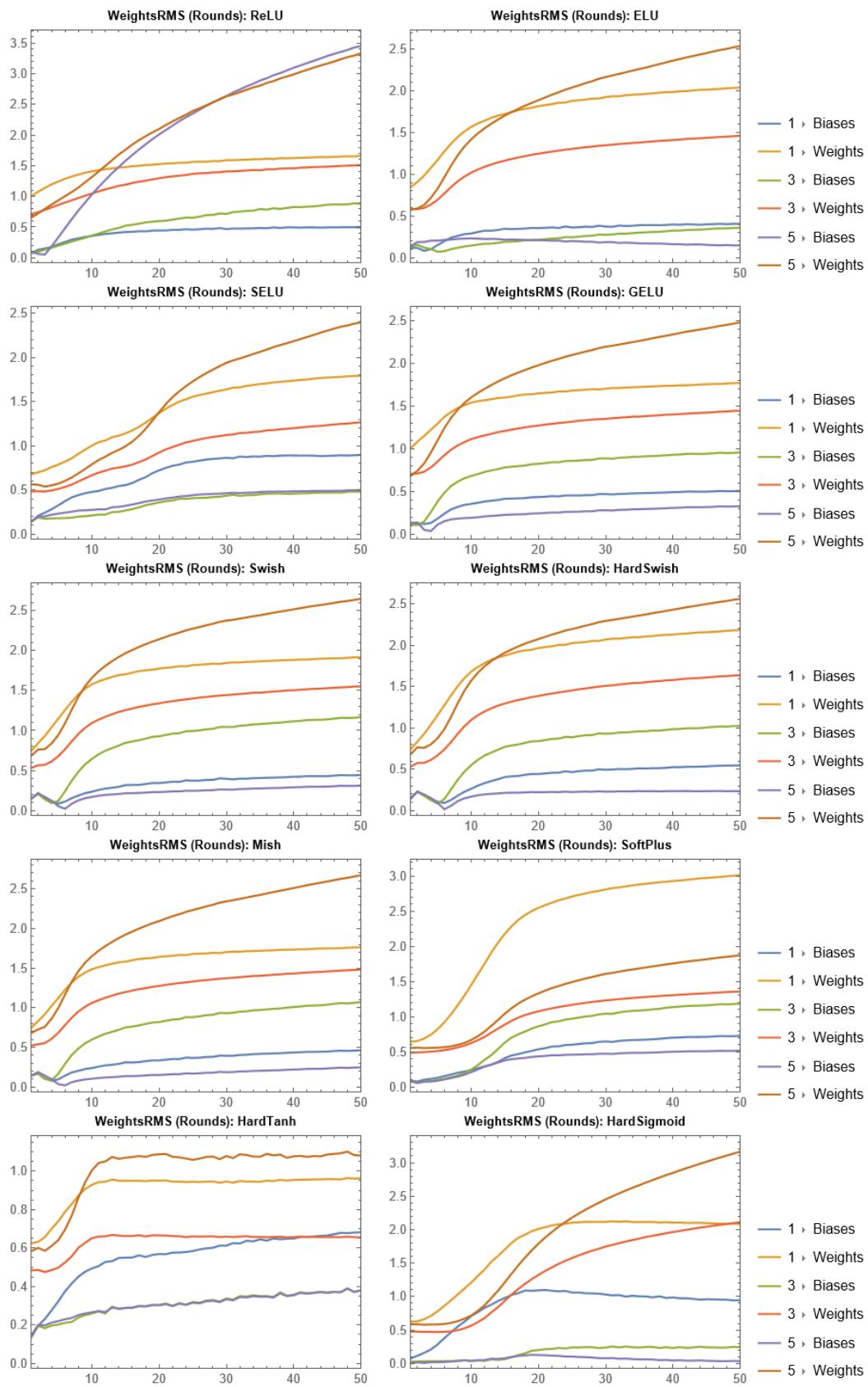
  (* Maximum number of training iterations: *)
  MaxTrainingRounds->50
]
],
];

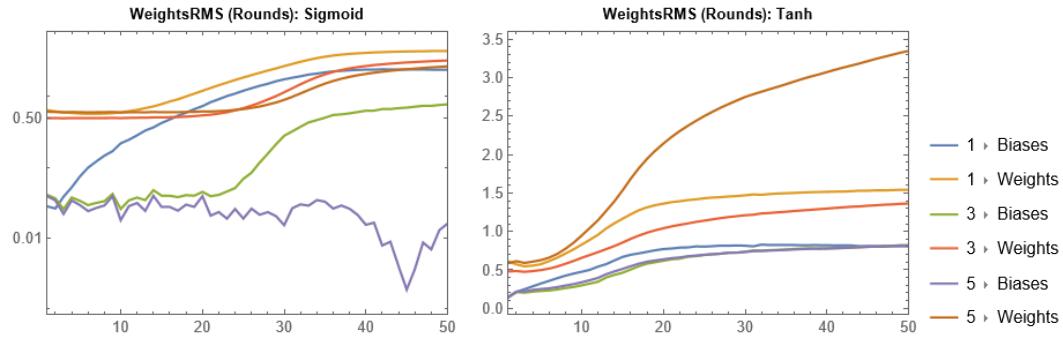
(* Iterate over the grid of activation functions: *)
{functions,activationFunctions}
]

```



Output



**Mathematica Code 9.71**

Input (* The code aims to generate synthetic training and validation data points within a unit disk, assigning binary labels based on their distance from the origin. It visualizes the training set to inspect data distribution and evaluates various activation functions in a neural network by systematically training and validating the network using each function. The neural network, with two hidden layers and an output layer using logistic sigmoid activation, is trained using the ADAM optimization method, with progress monitored through accuracy and confusion matrix plots. Results, including validation accuracy, confusion matrices, and loss plots, are recorded and compared for each activation function. The decision boundaries of the trained networks are visualized to understand the influence of each activation function, providing insights into their impact on the training dynamics and classification performance, aiding in selecting suitable activation functions for future designs: *)

```
(* Generate 1000 random points within a unit disk for training: *)
trainpoints=RandomPoint[Disk[],1000];

(* Assign labels based on whether the point's distance from the origin is less
than 0.5: *)
trainlabels=Thread[Map[Norm,trainpoints]<0.5];

(* Combine the points and labels into training data: *)
trainingData=trainpoints->trainlabels;

(* Generate 500 random points within a unit disk for validation: *)
validationPoints=RandomPoint[Disk[],500];

(* Assign labels using the same criteria as for the training set: *)
validationLabels=Thread[Map[Norm,validationPoints]<0.5];

(* Combine the points and labels into validation data: *)
validationData=validationPoints->validationLabels;

(* Visualize the synthetic training set with different colors for each class: *)
ListPlot[
{
  Pick[trainpoints,trainlabels,True],
  Pick[trainpoints,trainlabels,False]
},
AspectRatio->1,
ImageSize->250,
PlotStyle->PointSize[Medium],
FrameLabel->{"X","Y","Synthetic Training Set"},
PlotLegends->{"Class 1","Class 2"}
]
```

```

(* Set up array for activation functions: *)
activationFunctions={
    "ReLU","ELU","SELU",
    "GELU","Swish","HardSwish",
    "Mish","SoftPlus","HardTanh",
    "HardSigmoid","Sigmoid",Tanh
} ;

(* Loop through each activation function, train a neural network, and record the
results: *)
results=Table[
    Module[
        {net,trainedNet},

        (* Define a simple neural network structure: *)
        net=NetChain[
            {
                (* First hidden layer with specified activation function: *)
                LinearLayer[5],ElementwiseLayer[functions],
                (* Second hidden layer with specified activation function: *)
                LinearLayer[5],ElementwiseLayer[functions],
                (* Output layer with logistic sigmoid activation function: *)
                LinearLayer[],ElementwiseLayer[LogisticSigmoid]
            },
            "Output"→NetDecoder["Boolean"]
        ];

        (* Train the network on training data and validate it: *)
        trainedNet=NetTrain[
            net,
            trainingData,
            All,
            (* Use validation data for monitoring performance: *)
            ValidationSet→validationData,
            (* Monitor accuracy and confusion matrix during training: *)
            TrainingProgressMeasurements→{"Accuracy","ConfusionMatrixPlot"},
            (* Set the batch size for training: *)
            BatchSize→64,
            (* Learning rate for "ADAM": *)
            LearningRate→0.01,
            (* Use "ADAM" as the optimization method: *)
            Method→"ADAM",
            (* Maximum number of training iterations: *)
            MaxTrainingRounds→50
        ];
        (* Extract validation measurements from training results: *)
        validationMeasurements=trainedNet["ValidationMeasurements"];
        (* Get the loss plot from training results: *)
        lossplot=trainedNet["LossPlot"];
        (* Get the trained network: *)
        trainednet=trainedNet["TrainedNet"];
        (* Return validation measurements, loss plot, and trained network: *)
        {validationMeasurements,lossplot,trainednet}
    ],
    (* Iterate over the grid of activation functions: *)
    {functions,activationFunctions}
];

(* Display LossPlot for each activation function: *)
Table[
    Overlay[{results[[i]][[2]],activationFunctions[[i]]},Alignment→{0.9,0.4}],

```

```

{i,1,12}  

]  
  

(* Display Loss for each activation function: *)  

Table[  

  {activationFunctions[[i]],results[[i]][[1]][[1]]},  

{i,1,12}  

]  
  

(* Display validation accuracy for each activation function: *)  

Table[  

  {activationFunctions[[i]],results[[i]][[1]][[2]]},  

{i,1,12}  

]  
  

(* Display ConfusionMatrixPlot for each activation function: *)  

Table[  

  Overlay[{results[[i]][[1]][[3]],activationFunctions[[i]]},Alignment->{0.7,0.8}],  

{i,1,12}  

]  
  

(* Visualize the decision boundary of the trained network: *)  

Table[  

  ContourPlot[  

    results[[i]][[3]][{x,y},None],  

    {x,-1,1},  

    {y,-1,1},  

    ContourStyle->{White},  

    ClippingStyle->Automatic,  

    ColorFunction->"BlueGreenYellow",  

    PlotLegends->Automatic,  

    LabelStyle->Directive[Black,10],  

    ImageSize->250,  

    PlotLabel->Style[Row[{"Trained Nonlinear Classifier Decision \n Boundary With:",  

      activationFunctions[[i]]}],10,Bold]  

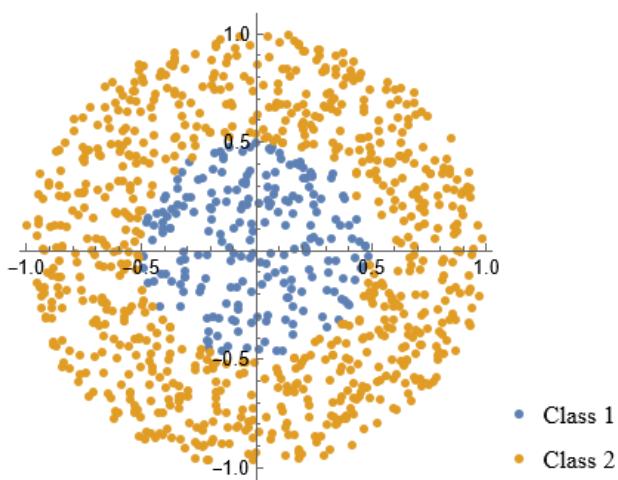
    ],  

{i,1,12}  

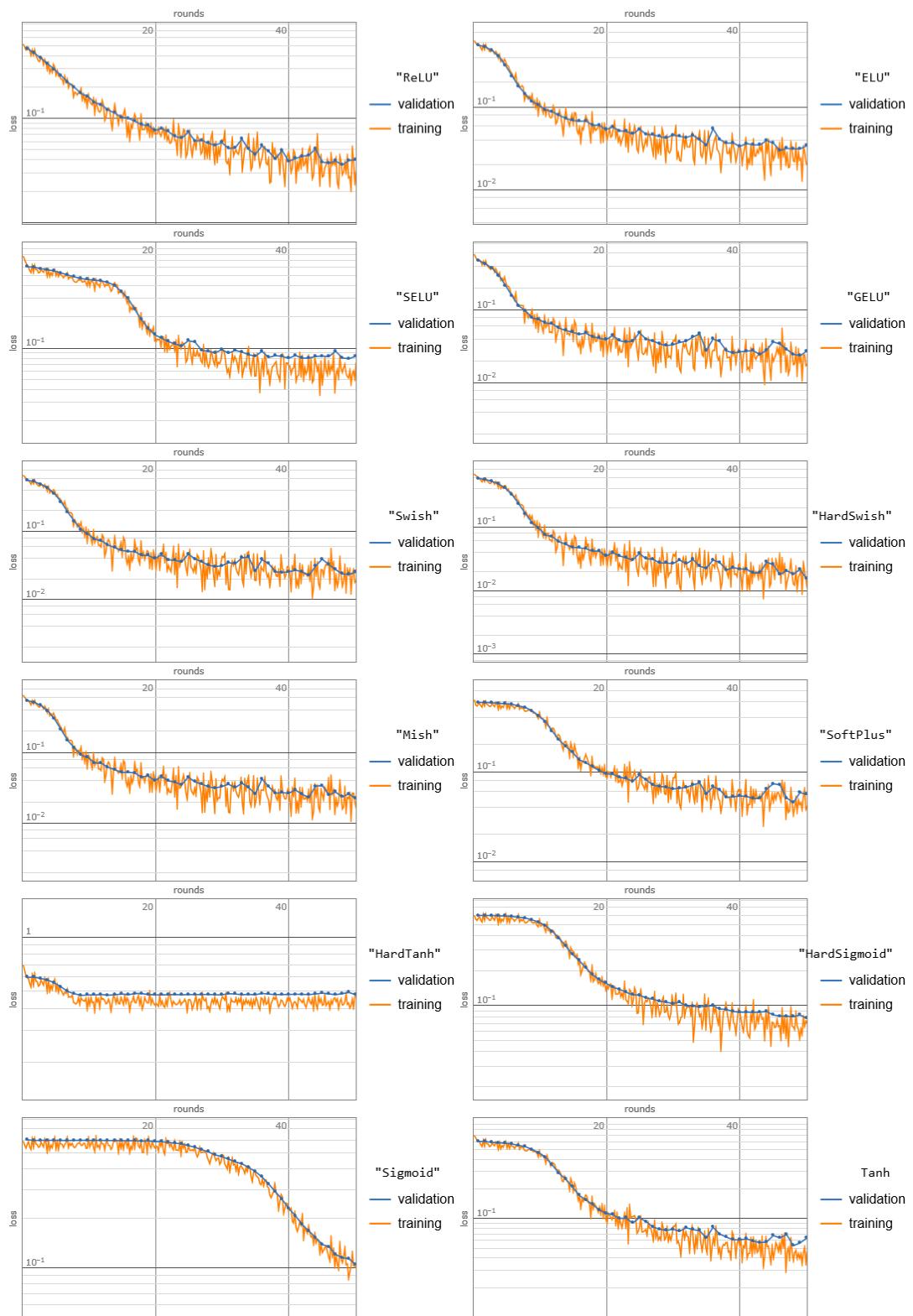
]

```

Output

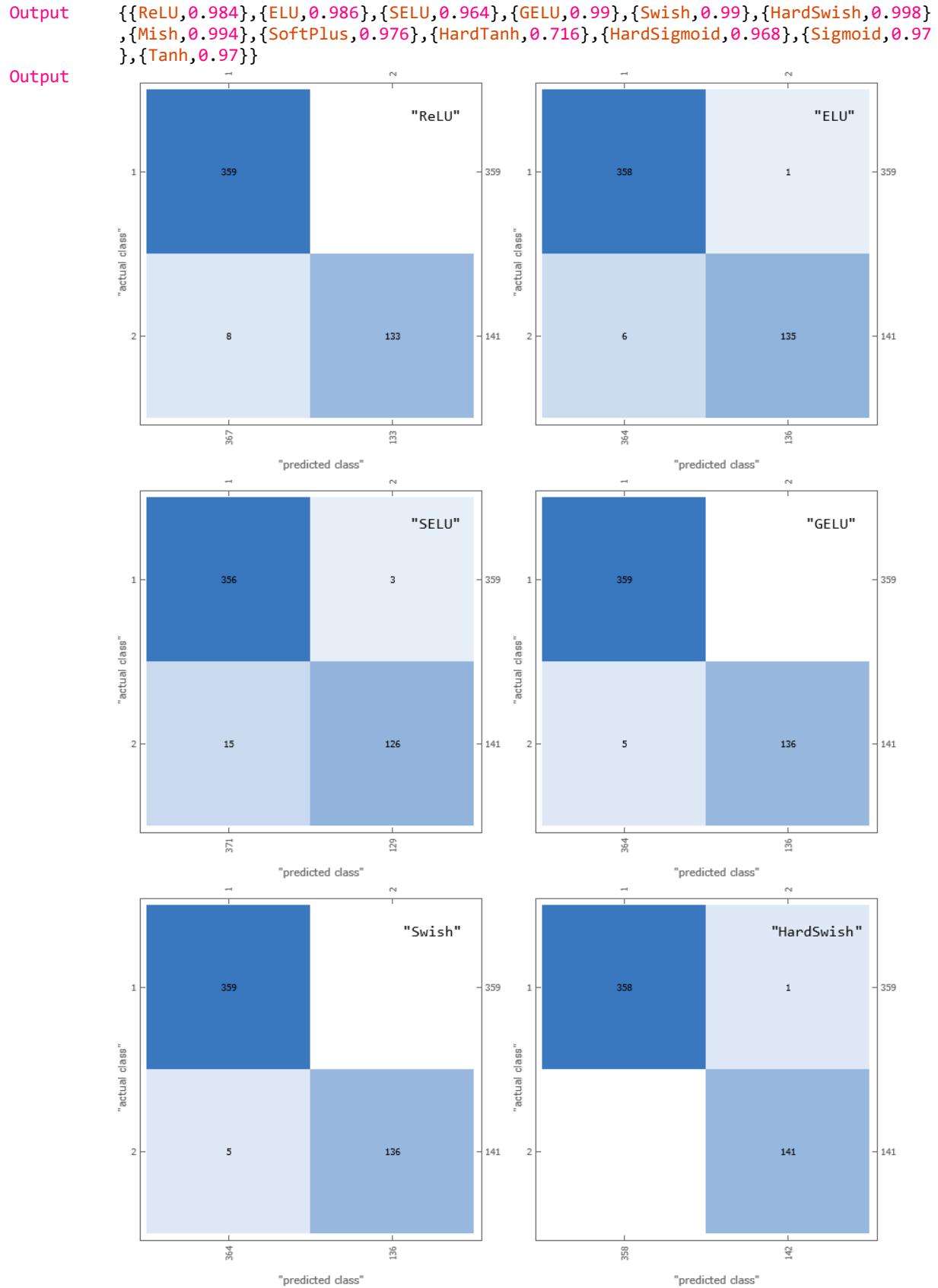


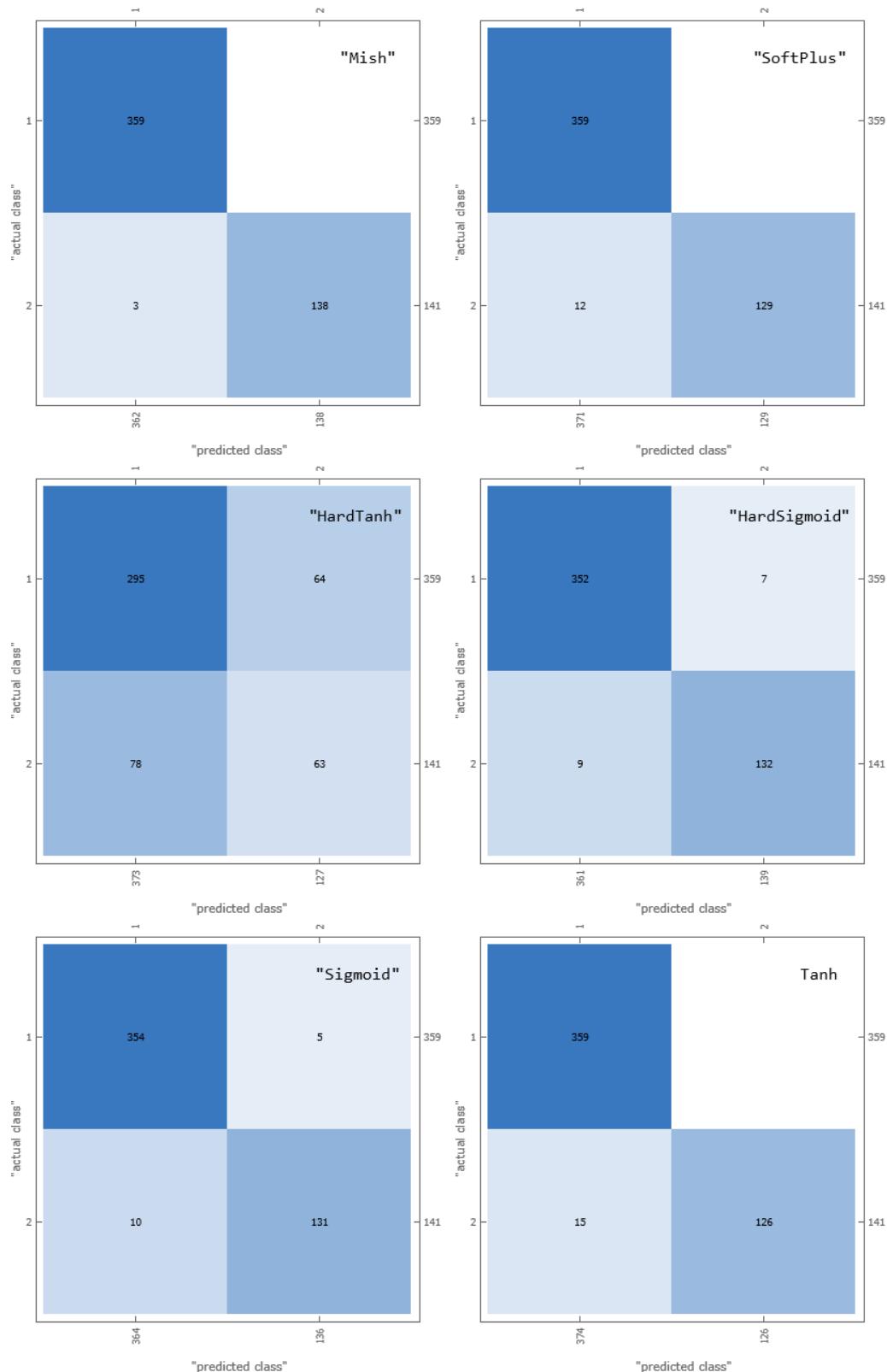
Output

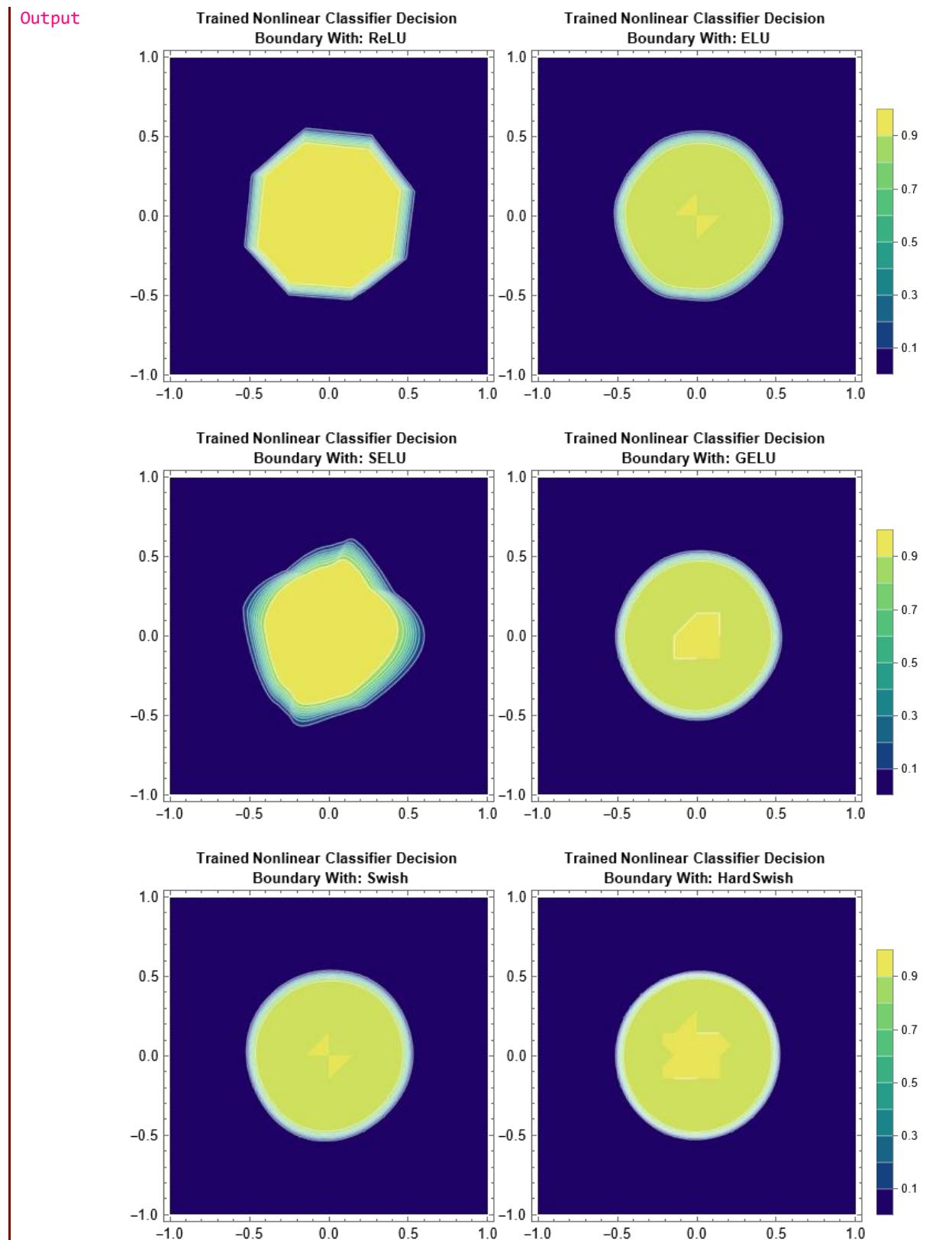


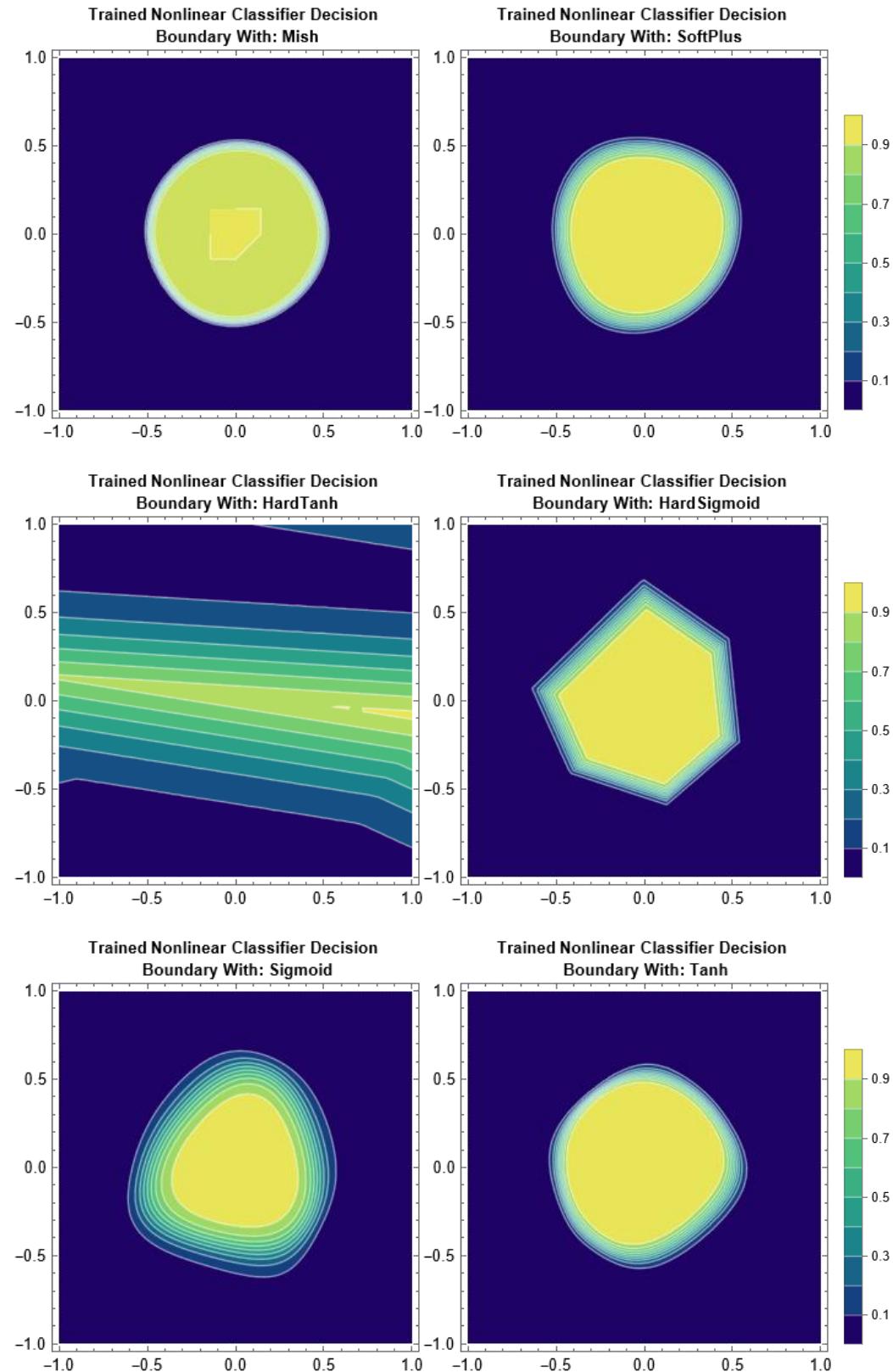
Output

```
[[{"activation": "ReLU", "loss": 0.0400866}, {"activation": "ELU", "loss": 0.0340551}, {"activation": "SELU", "loss": 0.0836259}, {"activation": "GELU", "loss": 0.0274739}, {"activation": "Swish", "loss": 0.0249386}, {"activation": "HardSwish", "loss": 0.0152379}, {"activation": "Mish", "loss": 0.0221938}, {"activation": "SoftPlus", "loss": 0.0558646}, {"activation": "HardTanh", "loss": 0.47632}, {"activation": "HardSigmoid", "loss": 0.0776821}, {"activation": "Sigmoid", "loss": 0.104511}, {"activation": "Tanh", "loss": 0.0634383}]]
```









CHAPTER 10

COMPLEX VALUED NEURAL NETWORKS

Remark:

This chapter provides a Mathematica implementation of the concepts and ideas presented in Chapter 9, [1], of the book titled *Artificial Neural Network and Deep Learning: Fundamentals and Theory*. We strongly recommend that you begin with the theoretical chapter to build a solid foundation before exploring the corresponding practical implementation. This chapter also serves as a summary of the article titled [Comprehensive Survey of Complex-Valued Neural Networks: Insights into Backpropagation and Activation Functions](#). For more details about complex valued activation functions, please refer to Ref [39].

In this chapter, we delve into the world of complex-valued activation functions (CVAFs) and their visualizations using Mathematica. For a more comprehensive background, we refer to [39] and the references cited therein. The ability to visualize these functions is crucial for gaining deeper insights into their behavior and understanding how they transform complex inputs. By leveraging Mathematica's powerful visualization tools, we can explore the intricacies of CVAFs in both two-dimensional and three-dimensional representations.

Three-dimensional plots are particularly useful for visualizing the magnitude and phase of complex functions. Mathematica's `ComplexPlot3D` function is designed to generate a three-dimensional representation of the magnitude of the function, `Abs[f]`, while using color to indicate the argument, `Arg[f]`, over a specified complex rectangle. This type of plot provides a comprehensive view, allowing us to observe how the magnitude and phase of a CVAF vary across the complex plane. We will demonstrate how to use `ComplexPlot3D` effectively to visualize various CVAFs, providing detailed examples and techniques to enhance your understanding. These visualizations are invaluable for analyzing the behavior of CVAFs, revealing patterns and characteristics that are not immediately apparent from the mathematical formulation alone.

While three-dimensional plots offer a detailed perspective, two-dimensional visualizations are often more accessible and easier to interpret for certain applications. The `ComplexPlot` function in Mathematica generates a two-dimensional representation of `Arg[f]` over the complex rectangle, effectively mapping the phase of the function onto a color gradient. This straightforward visualization technique is powerful for quickly assessing the function's behavior on the complex plane. Throughout this chapter, we will explore how to utilize `ComplexPlot` to visualize CVAFs, offering practical examples and tips for creating clear and informative plots. These two-dimensional visualizations complement the three-dimensional ones, providing a different yet equally valuable perspective on the behavior of CVAFs.

Interactivity is a key feature in Mathematica that allows for dynamic exploration of CVAFs. Using the `Manipulate` function, we can create interactive visualizations where the parameters of the AFs can be adjusted in real-time. This dynamic approach enables us to observe how changes in parameters affect the AFs, offering a hands-on way to explore their behavior. Interactive simulations and visualizations facilitate a deeper understanding of the mathematical underpinnings of CVAFs.

To provide a comprehensive resource, this chapter includes an extensive catalog of CVAFs, detailing their definitions, properties, and visual representations. The catalog covers a wide range of AFs, including but not limited to: Split-Step Function, Split-Sigmoid, Split-Parametric Sigmoid, Split-Tanh, Split-Sigmoid Tanh, Split-Hard Tanh, Split-CReLU, Split-QAM, Amplitude-Phase-Type Function, Amplitude-Phase Sigmoidal Function, Complex Cardioid, modReLU, Fully Complex Tanh, Fully Complex Logistic-Sigmoidal, Fully Complex Elementary Transcendental Function (ETF), zReLU, z3ReLU, zPReLU, z3PReLU.

Unit 10.1

Complex Numbers and Functions

Mathematica Code 10.1

```

Input      (* The code defines complex variables and functions, separates their real and
           imaginary parts, and visualizes these components using contour plots. Specifically,
           it creates a complex function f by squaring a complex variable z, then defines u
           and v as the real and imaginary parts of f respectively. It plots the contours of
           u and v for specified values in the uv-plane and also plots hyperbolas defined by
           x^2-y^2=c1 and 2xy=c2 in the xy-plane, providing a graphical representation of
           these mathematical relationships: *)

(* Define z as a complex number with real part x and imaginary part y: *)
z[x_,y_]:=x+I y
(* Define f as the square of the complex number z: *)
f[x_,y_]:=z[x,y]^2

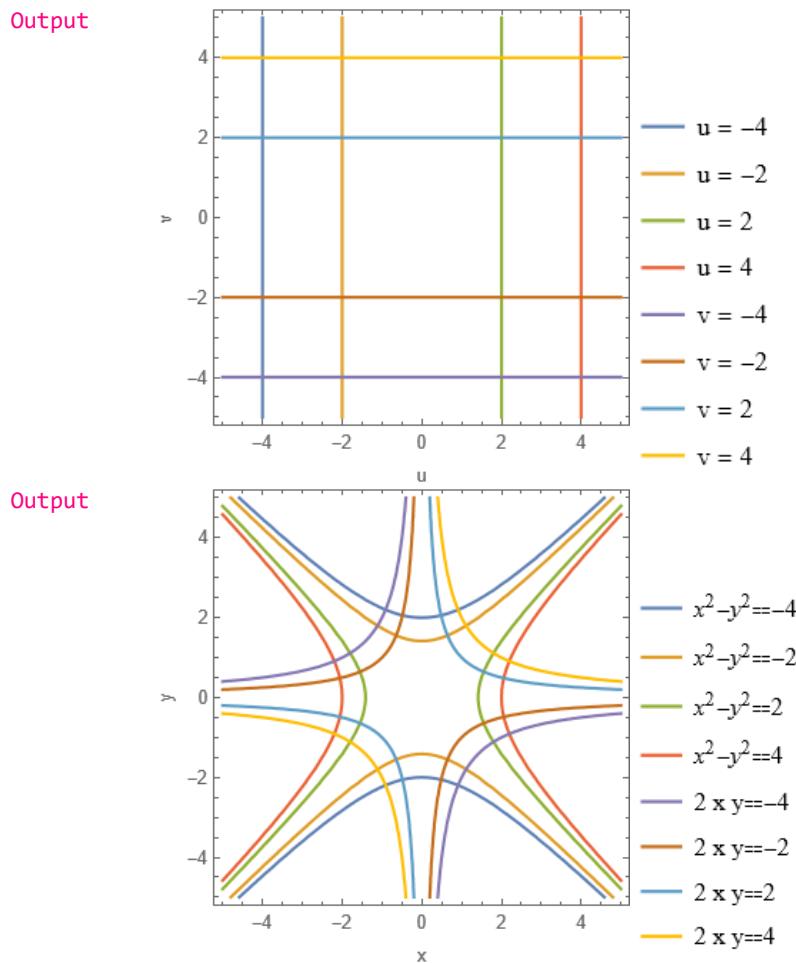
(* Define u as the real part of the complex function f: *)
u[x_,y_]:=Re[f[x,y]]
(* Define v as the imaginary part of the complex function f: *)
v[x_,y_]:=Im[f[x,y]]

(* Choose the range of c1 values for plotting hyperbolas of the form x^2-y^2=c1: *)
c1Values={-4,-2,2,4};
(* Choose the range of c2 values for plotting hyperbolas of the form 2xy=c2: *)
c2Values={-4,-2,2,4};

(* Contour plot of the functions u and v in the uv-plane: *)
ContourPlot[
  (* Equations for contour lines: *)
  {u==-4,u==-2,u==2,u==4,v==-4,v==2,v==4},
  {u,-5,5},
  {v,-5,5},
  FrameLabel->{"u","v"},
  PlotLegends->{"u = -4","u = -2","u = 2","u = 4","v = -4","v = -2","v = 2","v = 4"},
  PlotRange->All,
  ImageSize->250
]

(* Contour plot of the hyperbolas x^2-y^2=c1 and 2xy=c2 in the xy-plane: *)
ContourPlot[
  (* Equations for hyperbolas: *)
  {x^2-y^2== -4,x^2-y^2== -2,x^2-y^2== 2,x^2-y^2== 4,2 x y== -4,2 x y== -2,2 x y== 2,2 x y== 4},
  {x,-5,5},
  {y,-5,5},
  FrameLabel->{"x","y"},
  PlotLegends->{"x^2-y^2== -4","x^2-y^2== -2","x^2-y^2== 2","x^2-y^2== 4","2 x y== -4","2 x y== -2","2 x y== 2","2 x y== 4"},
  PlotRange->All,
  ImageSize->250
]

```

**Mathematica Code 10.2**

```

Input (* Define the complex mapping function: *)
f[z_]:=z^2

(* Define a function to extract real and imaginary parts: *)
getRealImag[expr_]:={Re[expr],Im[expr]}

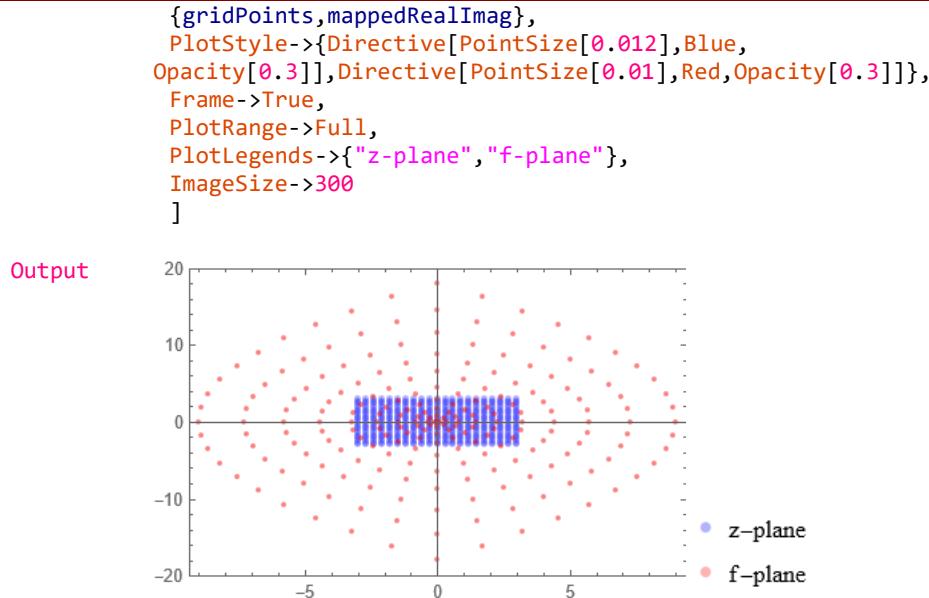
(* Create a grid of points in the z-plane: *)
gridPoints=Flatten[
  Table[
    {x,y},
    {x,-3,3,0.3},
    {y,-3,3,0.3}],
  1];
]

(* Apply the complex mapping function to each point in the grid: *)
mappedPoints=f[MapApply[Complex,gridPoints]];

(* Extract real and imaginary parts of the mapped points: *)
mappedRealImag=Map[getRealImag,mappedPoints];

(* Plot the z-plane points and the corresponding f-plane points: *)
ListPlot[

```

**Mathematica Code 10.3**

Input (* In this code, the Manipulate function allows you to interactively change the exponent in the complex mapping function $f(z)=z^{\text{exp}}$. The slider labeled "Exponent" allows you to dynamically adjust the value of exp, and the plot updates accordingly: *)

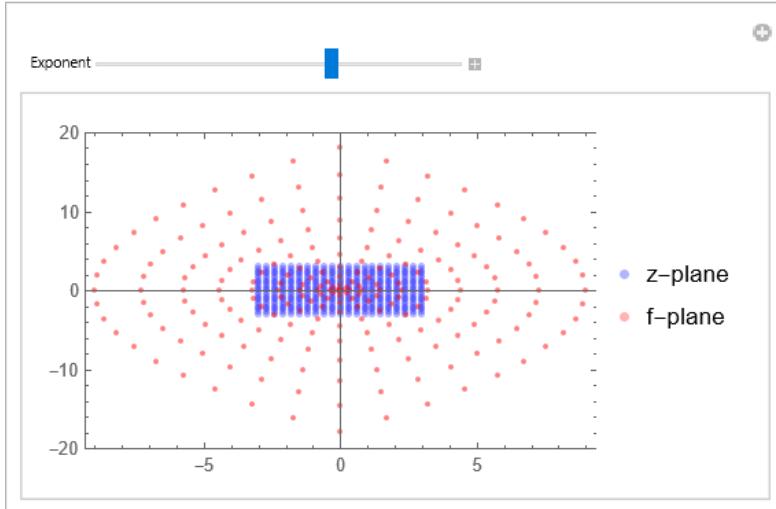
```
Manipulate[
(* Define the complex mapping function: *)
f[z_]:=z^exp;
(* Define a function to extract real and imaginary parts: *)
getRealImag[expr_]:={Re[expr],Im[expr]};
(* Create a grid of points in the z-plane: *)
gridPoints=Flatten[
Table[
{x,y},
{x,-3,3,0.3},
{y,-3,3,0.3}
],
1
];
(* Apply the complex mapping function to each point in the grid: *)
mappedPoints=f[MapApply[Complex,gridPoints]];

(* Extract real and imaginary parts of the mapped points: *)
mappedRealImag=Map[getRealImag,mappedPoints];

(* Plot the z-plane points and the corresponding f-plane points: *)
ListPlot[
{gridPoints,mappedRealImag},
PlotStyle-
>{Directive[PointSize[0.012],Blue,Opacity[0.3]],Directive[PointSize[0.01],Red,Opacity[0.3]]},
Frame->True,
PlotRange->Full,
PlotLegends->{"z-plane","f-plane"},
ImageSize->300
],
```

```
(* Manipulate controls: *)
{{exp, 2, "Exponent"}, 0.1, 3, 0.1}
]
```

Output

**Mathematica Code 10.4**

Input

```
(* The code visualizes and analyzes complex functions by transforming a grid of
points in the complex plane using the functions f(z)=z^2 and f(z)= 1/z, and plotting
the results. It generates the original grid and its transformations, converts
complex numbers to pairs of real coordinates for plotting, and creates 3D plots of
the magnitude and argument of the functions. Finally, it displays these
visualizations in a grid layout, allowing for easy comparison and exploration of
how the complex functions affect the plane: *)

(* Define the complex functions: *)

(* Function f1: squares the complex number z: *)
f1[z_]:=z^2
(* Function f2: computes the reciprocal of z, avoids division by zero: *)
f2[z_]:=If[z==0,Infinity,1/z]

(* Generate a grid of points from -2 to 2 on both real and imaginary axes with a
step of 0.4, excluding the origin*)
gridPoints=DeleteCases[
  Flatten[
    Table[
      x+I y,
      {x,-2,2,0.4},
      {y,-2,2,0.4}
    ]
  ],
  0+0. I
];

(* Transform the grid under the complex functions: *)

(* Apply f1 to each point in the grid: *)
transformedGrid1=gridPoints/. z_Complex:>f1[z];
(* Apply f2 to each point in the grid: *)
transformedGrid2=gridPoints/. z_Complex:>f2[z];
```

```

(* Convert complex points to pairs of real numbers for plotting: *)
toRealPairs[complexList_]:=Map[{Re[#],Im[#]}&，《complexList》

(* Create plots for the original and transformed grids: *)
plotGrid[points_,title_]:=ListPlot[
  (* Plot the transformed points: *)
  toRealPairs[points],
  AxesOrigin→{0,0},
  AspectRatio→1,
  PlotRange→{{-5,5},{-5,5}},
  GridLines→Automatic,
  Epilog→{Blue,PointSize[Medium],Point[toRealPairs[points]]},
  AxesLabel→{"Re","Im"},
  PlotLabel→title
];

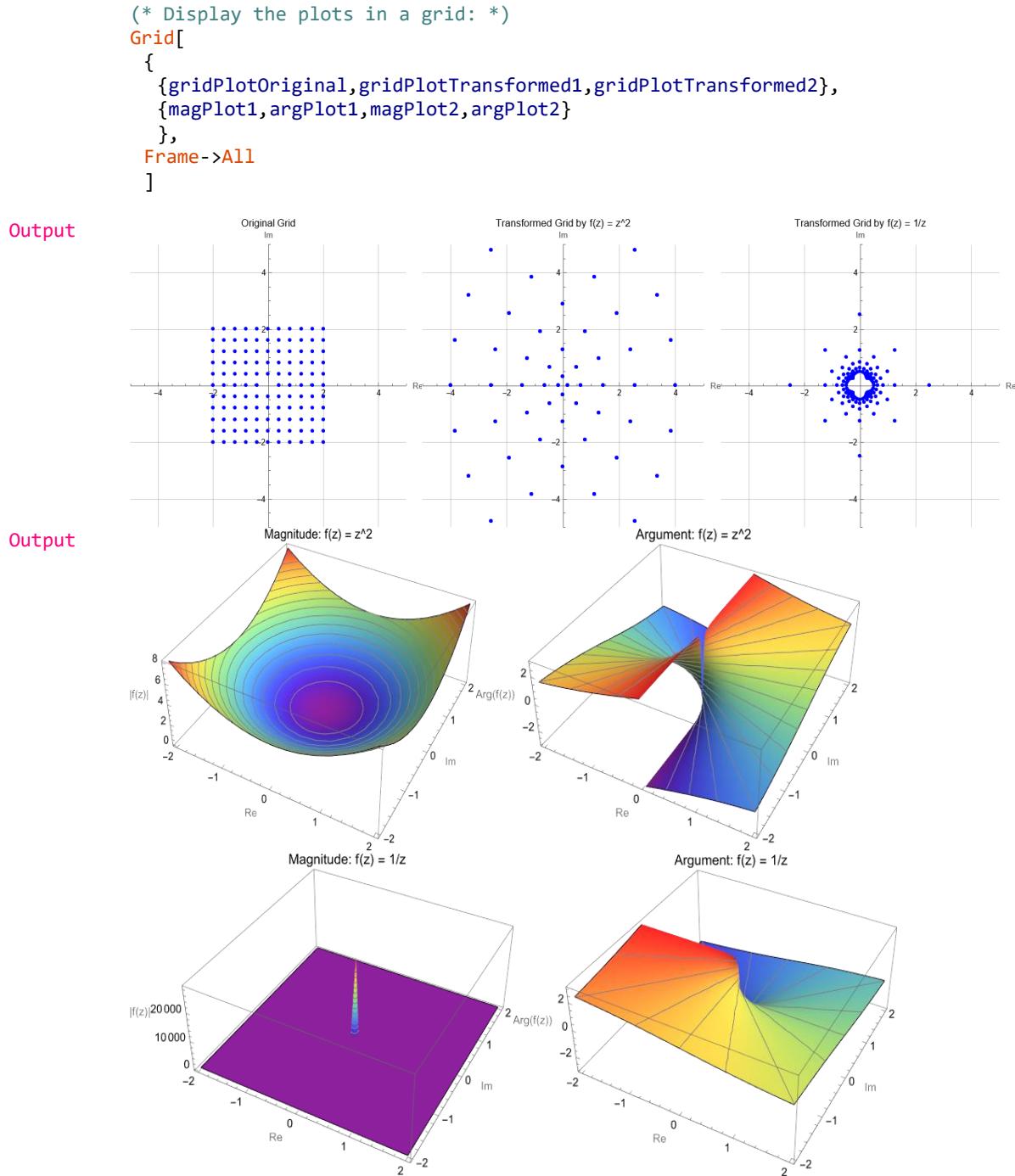
(* Plot the magnitude and argument of the complex functions: *)
plotMagnitudeAndArgument[f_,range_,title_]:=Module[
  {magnitude,argument},
  (* Plot the magnitude of the function: *)
  magnitude=Plot3D[
    Abs[f[x+I y]],
    {x,-range,range},
    {y,-range,range},
    PlotRange→All,
    ColorFunction→"Rainbow",
    MeshFunctions→{#3&},
    MeshStyle→Gray,
    PlotLabel→"Magnitude: "<>title,
    AxesLabel→{"Re","Im","|f(z)|"}
  ];
  (* Plot the argument of the function: *)
  argument=Plot3D[
    Arg[f[x+I y]],
    {x,-range,range},
    {y,-range,range},
    PlotRange→{-Pi,Pi},
    ColorFunction→"Rainbow",
    MeshFunctions→{#3&},
    MeshStyle→Gray,
    PlotLabel→"Argument: "<>title,
    AxesLabel→{"Re","Im","Arg(f(z))"}
  ];
  {magnitude,argument}
];

(* Generate the plots: *)

(* Plot the original grid: *)
gridPlotOriginal=plotGrid[gridPoints,"Original Grid"];
(* Plot the grid transformed by f1: *)
gridPlotTransformed1=plotGrid[transformedGrid1,"Transformed Grid by f(z) = z^2"];
(*Plot the grid transformed by f2*)
gridPlotTransformed2=plotGrid[transformedGrid2,"Transformed Grid by f(z) = 1/z"];

(* Plot the magnitude and argument of f1: *)
{magPlot1,argPlot1}=plotMagnitudeAndArgument[f1,2,"f(z) = z^2"];
(* Plot the magnitude and argument of f2: *)
{magPlot2,argPlot2}=plotMagnitudeAndArgument[f2,2,"f(z) = 1/z"];

```

**Mathematica Code 10.5**

Input (* The code aims to visualize and interactively explore the impact of parameters on the transformations of a complex plane grid by two parameterized complex functions ($f(z)=(a z)^2+b$ and $f(z)=c/z+d$). It generates a grid of complex points, excluding the origin, converts these points to real coordinate pairs for plotting, and uses `Manipulate` to allow dynamic adjustment of the parameters a , b , c , and d : *)

```
f1[z_,a_,b_]:= (a z)^2+b
```

```

f2[z_,c_,d_]:=If[z==0,Infinity,c/z+d] (*Avoid division by zero*)

(* Create a grid of points in the complex plane, excluding the origin: *)
gridPoints=DeleteCases[Flatten[Table[x+I y,{x,-2,2,0.4},{y,-2,2,0.4}]],0+0. I];

(* Convert complex points to pairs of real numbers for plotting: *)
toRealPairs[complexList_]:={Re[#],Im[#]}&/@complexList

(* Create plots for the original and transformed grids: *)
plotGrid[points_,title_]:=ListPlot[
  toRealPairs[points],
  AxesOrigin->{0,0},
  AspectRatio->1,
  PlotRange->{{{-5,5}},{{-5,5}}},
  GridLines->Automatic,
  Epilog->{Blue,PointSize[Medium],Point[toRealPairs[points]]},
  AxesLabel->{"Re","Im"},
  ImageSize->250,
  PlotLabel->title
];

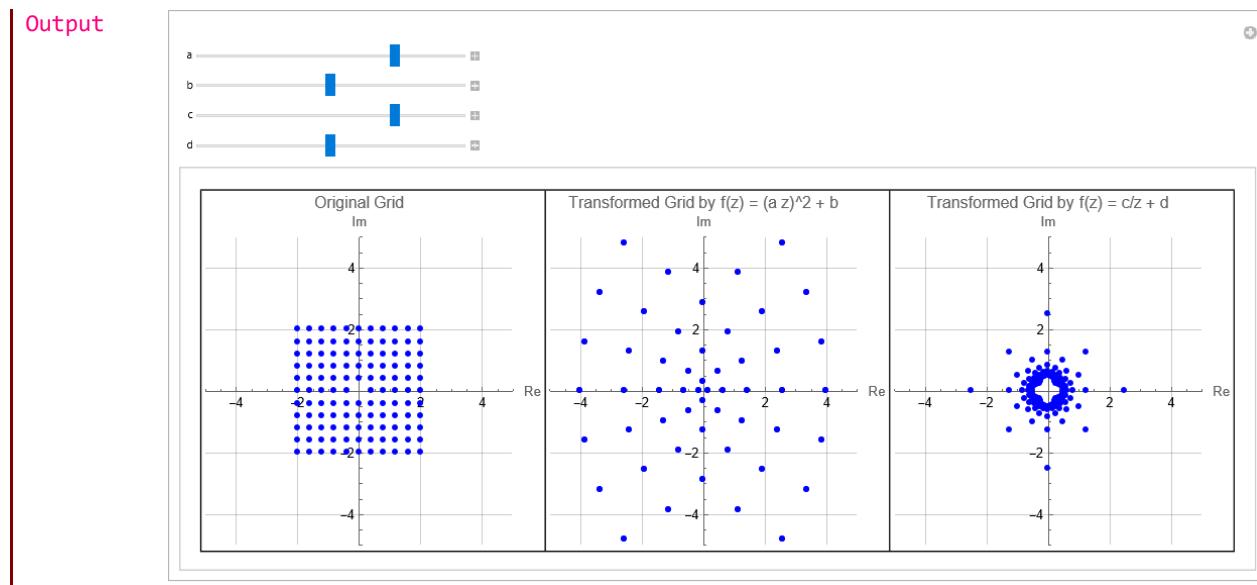
(* Use Manipulate for interactive visualization: *)
Manipulate[
  Module[
    {transformedGrid1,transformedGrid2,gridPlotOriginal,gridPlotTransformed1,gridPlotTransformed2},

    (* Transform the grid under the complex functions with parameters: *)
    transformedGrid1=gridPoints/. z_Complex:>f1[z,a,b];
    transformedGrid2=gridPoints/. z_Complex:>f2[z,c,d];

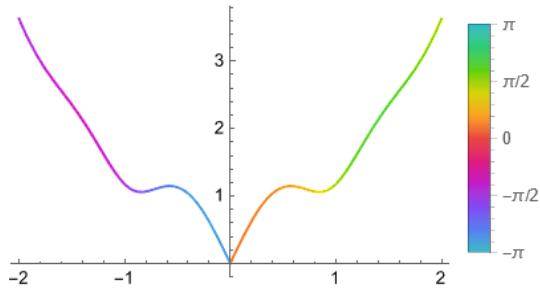
    (* Generate the plots: *)
    gridPlotOriginal=plotGrid[gridPoints,"Original Grid"];
    gridPlotTransformed1=plotGrid[transformedGrid1,"Transformed Grid by f(z) = (a z)^2 + b"];
    gridPlotTransformed2=plotGrid[transformedGrid2,"Transformed Grid by f(z) = c/z + d"];

    (* Display the plots: *)
    Grid[
      {
        {gridPlotOriginal,gridPlotTransformed1,gridPlotTransformed2}
      },
      Frame->All
    ],
    (* Slider for parameter a: *)
    {{a,1,"a"},-2,2,0.1},
    (* Slider for parameter b: *)
    {{b,0,"b"},-2,2,0.1},
    (* Slider for parameter c: *)
    {{c,1,"c"},-2,2,0.1},
    (* Slider for parameter d: *)
    {{d,0,"d"},-2,2,0.1}
  ]
]

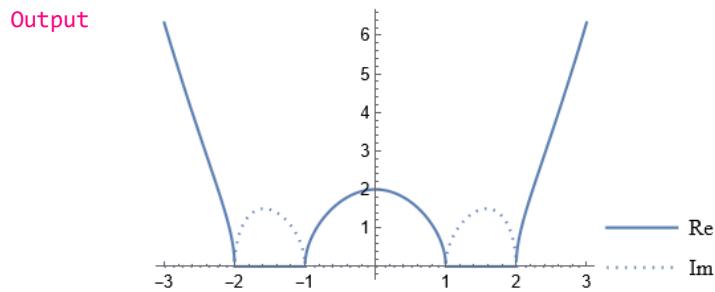
```

**Mathematica Code 10.6**

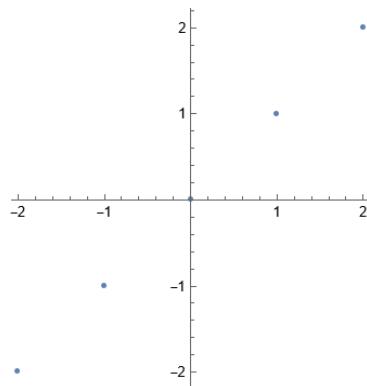
```
Input (* Plot the modulus of a complex function of a real variable colored by its
       argument: *)
AbsArgPlot[
  Sin[I x]+Sin[Pi x],
  {x,-2,2},
  PlotRange->Full,
  PlotLegends->Automatic,
  ImageSize->250
]
```

Output**Mathematica Code 10.7**

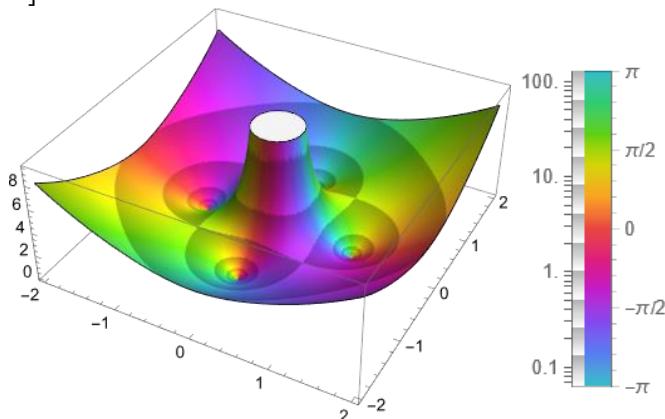
```
Input (* Plot the real and imaginary parts of a complex-valued function of a real
       variable: *)
ReImPlot[
  Sqrt[(x^2-1) (x^2-4)],
  {x,-3,3},
  PlotRange->Full,
  PlotLegends->Automatic,
  ImageSize->250
]
```

**Mathematica Code 10.8**

```
Input (*Plot a set of complex numbers:*)
ComplexListPlot[
{ -2-2 I, -1-I, 0, 1+I, 2+2 I },
PlotRange->Full,
ImageSize->250
]
```

Output**Mathematica Code 10.9**

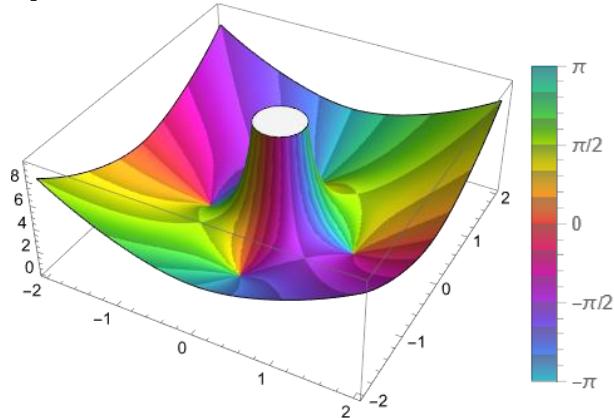
```
Input (* Use the "CyclicLogAbs" shading function to cyclically shade colors to give the
appearance of contours of constant Abs[f]: *)
ComplexPlot3D[
(z^4-1)/z^2,
{z, -2-2 I, 2+2 I},
ColorFunction->"CyclicLogAbs",
PlotLegends->Automatic,
ImageSize->300
]
```

Output

Mathematica Code 10.10

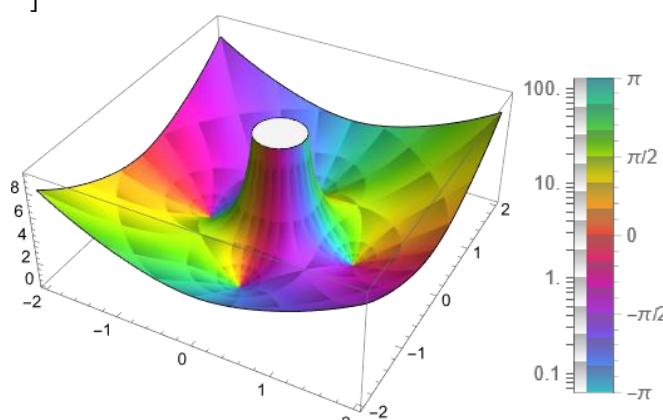
Input (* Use the "CyclicArg" shading function to cyclically shade colors to give the appearance of contours of constant Arg[f]: *)

```
ComplexPlot3D[
  (z^4-1)/z^2,
  {z,-2-2 I,2+2 I},
  ColorFunction->"CyclicArg",
  PlotLegends->Automatic,
  ImageSize->300
]
```

Output**Mathematica Code 10.11**

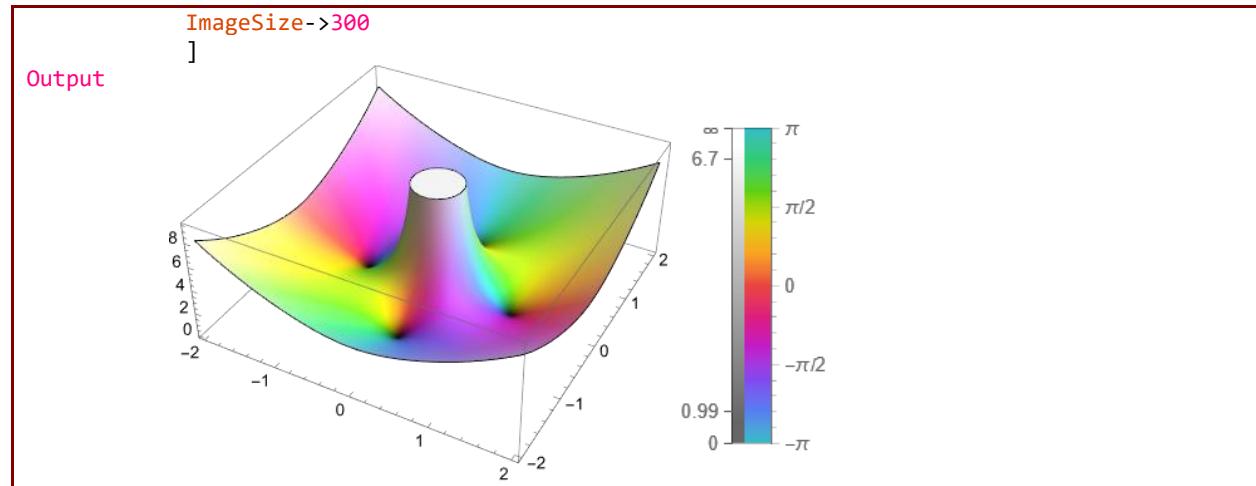
Input (* Use the "CyclicLogAbsArg" shading function to cyclically shade colors to give the appearance of contours of constant Abs[f] and constant Arg[f]: *)

```
ComplexPlot3D[
  (z^4-1)/z^2,
  {z,-2-2 I,2+2 I},
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  ImageSize->300
]
```

Output**Mathematica Code 10.12**

Input (* Use "QuantileAbs" to darken small values of Abs[f] and lighten large values of Abs[f]: *)

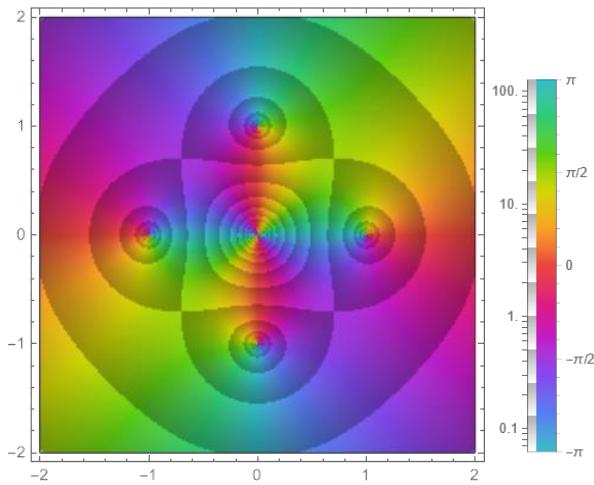
```
ComplexPlot3D[
  (z^4-1)/z^2,
  {z,-2-2 I,2+2 I},
  ColorFunction->"QuantileAbs",
  PlotLegends->Automatic,
```

**Mathematica Code 10.13**

Input (* Use "CyclicLogAbs" to cyclically shade colors to give the appearance of contours of constant Abs[f]: *)

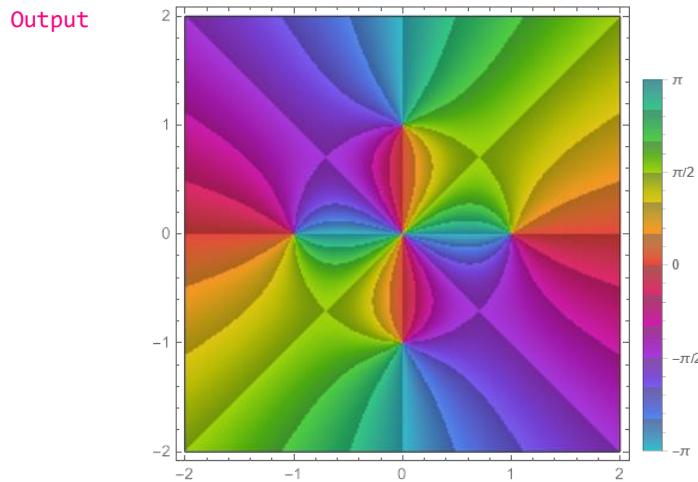
```
ComplexPlot[
  (z^4-1)/z^2,
  {z,-2-2 I,2+2 I},
  ColorFunction->{Automatic,"CyclicLogAbs"},
  PlotLegends->Automatic,
  ImageSize->300
]
```

Output

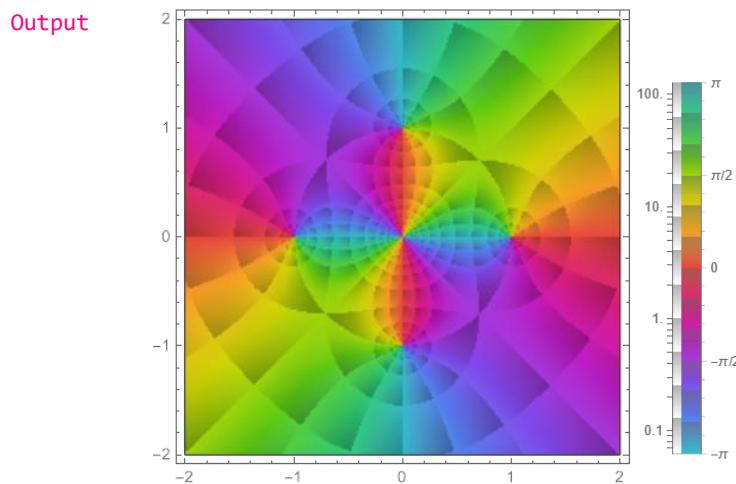
**Mathematica Code 10.14**

Input (* Use "CyclicArg" to cyclically shade colors to give the appearance of contours of constant Arg[f]: *)

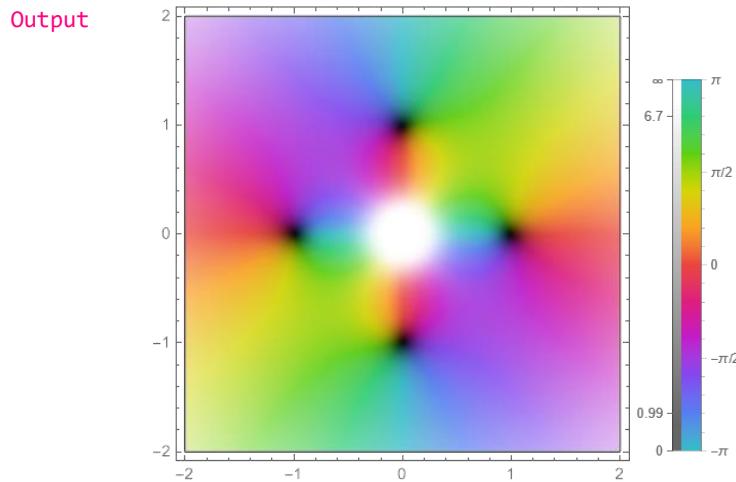
```
ComplexPlot[
  (z^4-1)/z^2,
  {z,-2-2 I,2+2 I},
  ColorFunction->"CyclicArg",
  PlotLegends->Automatic,
  ImageSize->300
]
```

**Mathematica Code 10.15**

```
Input (* Use "CyclicLogAbsArg" to cyclically shade colors to give the appearance of
       contours of constant Abs[f] and constant Arg[f]: *)
ComplexPlot[
  (z^4-1)/z^2,
  {z, -2-2 I, 2+2 I},
  ColorFunction -> "CyclicLogAbsArg",
  PlotLegends -> Automatic,
  ImageSize -> 300
]
```

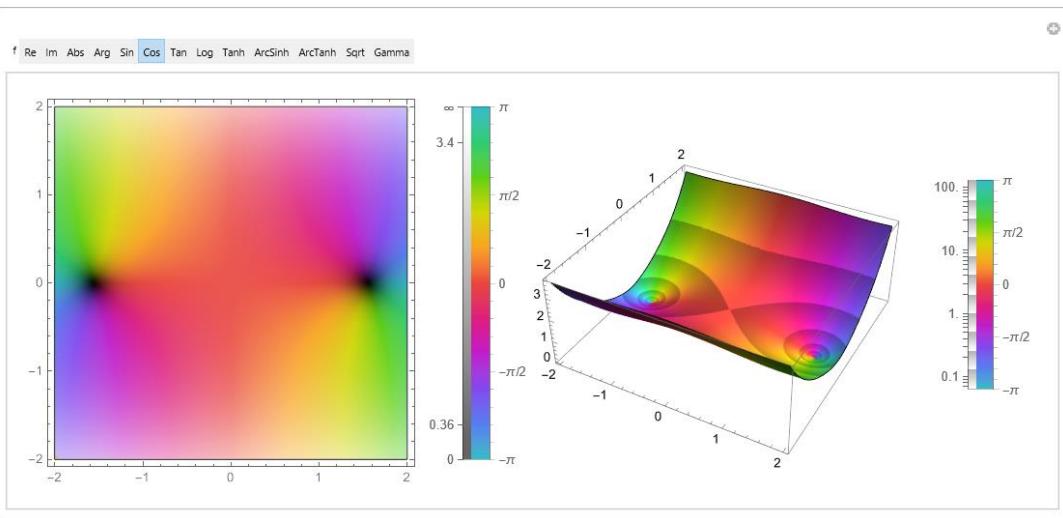
**Mathematica Code 10.16**

```
Input (* Use "QuantileAbs" to darken small values of Abs[f] and lighten large values of
       Abs[f]: *)
ComplexPlot[
  (z^4-1)/z^2,
  {z, -2-2 I, 2+2 I},
  ColorFunction -> "QuantileAbs",
  PlotLegends -> Automatic,
  ImageSize -> 300
]
```

**Mathematica Code 10.17**

Input (* The code allows the user to choose a complex-valued function f from a predefined set of options. The options include real and complex parts (Re, Im), magnitude (Abs), argument (Arg), and various transcendental functions (Sin, Cos, Tan, Log, Tanh, ArcSinh, ArcTanh, Sqrt, Gamma). Two types of complex plots are displayed side by side within a Row. The first plot is a 2D complex plot (ComplexPlot) using the chosen function $f(z)$. It uses the color function "QuantileAbs" and includes legends. The second plot is a 3D complex plot (ComplexPlot3D) using the same function and a different color function ("CyclicLogAbs"). It also includes legends and has a specified plot range. The Manipulate function provides an interactive interface, allowing the user to dynamically manipulate the displayed plots by selecting different functions for f : *)

```
Manipulate[
 Row[
 {
 ComplexPlot[
 f[z],
 {z, -2-2 I, 2+2 I},
 ColorFunction->"QuantileAbs",
 PlotLegends->Automatic,
 ImageSize->300
 ],
 ComplexPlot3D[
 f[z],
 {z, -2-2 I, 2+2 I},
 ColorFunction->"CyclicLogAbs",
 PlotRange->Full,
 PlotLegends->Automatic,
 ImageSize->300
 ]
 }
 ],
 {{f, Tanh}, {Re, Im, Abs, Arg, Sin, Cos, Tan, Log, Tanh, ArcSinh, ArcTanh, Sqrt, Gamma}},
 ControlType->SetterBar
 ]
```

Output

Unit 10.2

Complex Valued Activation Functions

Mathematica Code 10.18

```

Input      (* The code defines and visualizes the Split-Step activation function for a complex
           variable z=x+iy by applying the step function separately to its real and imaginary
           parts. It introduces a modular function `createSplitStepPlot` to generate and
           display 3D surface plots and slice contour plots for the real part, imaginary part,
           magnitude, and phase of the Split-Step function. The generated plots are displayed
           together, allowing for a comprehensive examination of the Split-Step function's
           behavior across different components in the complex plane: *)

(* Define a function to create the plots for different components of the Split-
Step Function: *)
createSplitStepPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Define a complex variable: *)
  znumber=x+I*y;
  (* Define the Split-Step Function: *)
  oSSF[u_]:=If[u>=0,1,0];
  oSplitStep:=oSSF[Re[znumber]]+I oSSF[Im[znumber]];

  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-6,6},
    {y,-6,6},
    ClippingStyle->None,
    AxesLabel->{"Re(z)","Im(z)"},
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
    Lighting->"Neutral"
  ];

  (* Generate the slice contour plot of the specified component: *)
  slice=SliceContourPlot3D[
    component,
    z==0,(* Define the slicing plane at z=0 *)
    {x,-6,6},
    {y,-6,6},
    {z,-1,1},
    Contours->15,
    Axes->False,
    PlotPoints->50,
    PlotRangePadding->0,
    ColorFunction->"Rainbow"
  ];

  (* Combine the 3D plot and the slice contour plot: *)
  Show[
    plot1,
    slice,
    PlotRange->All,
    PlotLabel->label,
  ]
]

```

```

BoxRatios->{1,1,1},
FaceGrids->{Back,Left},
ImageSize->200
]
]

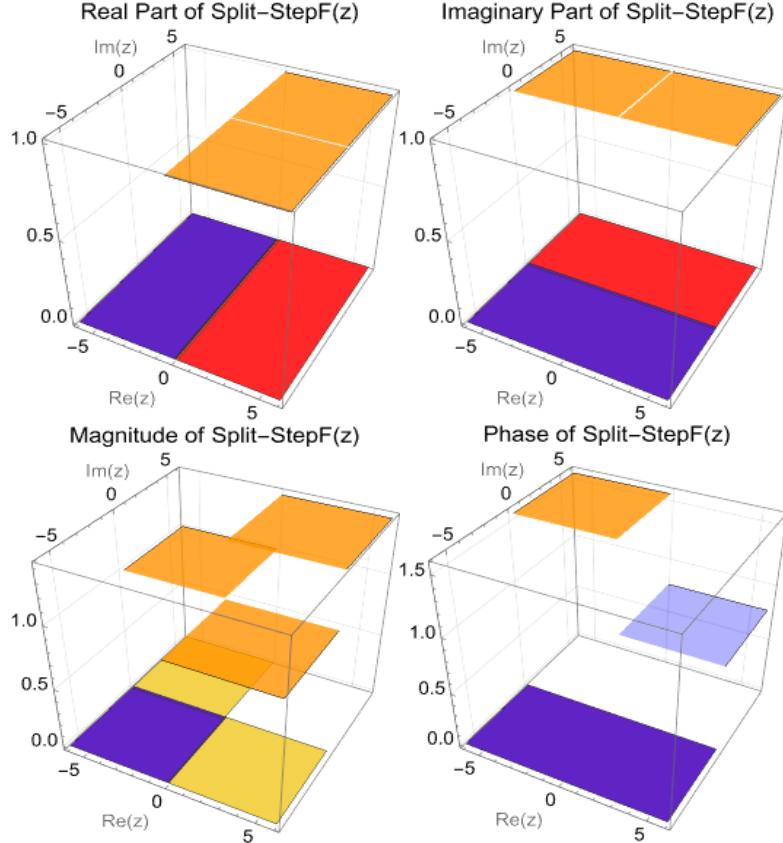
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
the Split-Step Function: *)
realSplitStepPlot=createSplitStepPlot[
  Re[\sigmaSplitStep],
  "Real Part of Split-StepF(z)"
];
imaginarySplitStepPlot=createSplitStepPlot[
  Im[\sigmaSplitStep],
  "Imaginary Part of Split-StepF(z)"
];

magnitudeSplitStepPlot=createSplitStepPlot[
  Abs[\sigmaSplitStep],
  "Magnitude of Split-StepF(z)"
];

phaseSplitStepPlot=createSplitStepPlot[
  Arg[\sigmaSplitStep],
  "Phase of Split-StepF(z)"
];
(*Display all four plots together in a list*)
{realSplitStepPlot,imaginarySplitStepPlot,magnitudeSplitStepPlot,phaseSplitStepPlot}

```

Output



Mathematica Code 10.19

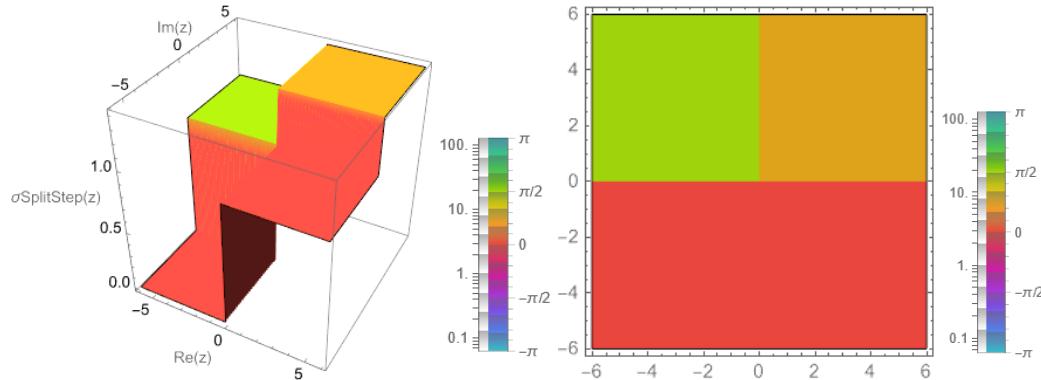
```

Input      (* Define the Split-Step Function: *)
oSSF[u_]:=If[u>=0,1,0];
oSplitStep:=oSSF[Re[z]]+I oSSF[Im[z]];

(* Generate a 3D plot of the Split-Step function over a specified complex range:
*)
ComplexPlot3D[
oSplitStep,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)","Im(z)","oSplitStep(z)"},
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->270
]

(* Generate a 2D complex plot of the sSplit-Step function over a specified complex
range: *)
ComplexPlot[
oSplitStep,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]

```

Output**Mathematica Code 10.20**

```

Input      (* The code defines and visualizes the split-sigmoidal activation function for a
complex variable z=x+iy by applying the sigmoid function separately to its real
and imaginary parts. It introduces a modular function `createSplitSigmoidalPlot`
to generate and display 3D surface plots and slice contour plots for the real part,
imaginary part, magnitude, and phase of the split-sigmoidal function. The generated
plots are displayed together, allowing for a comprehensive examination of the
split-sigmoidal function's behavior across different components in the complex
plane: *)

(* Define a function to create the plots for different components of the Split-
Sigmoidal function: *)
createSplitSigmoidalPlot[component_,label_]:=Module[
{plot1,slice},
(* Define a complex variable: *)

```

```

znumber=x+I*y;
(* Define the split-sigmoidal activation function: *)
sigmoidalR[u_]:=1/(1+Exp[-u]);
splitsigmoidal:=sigmoidalR[Re[znumber]]+I sigmoidalR[Im[znumber]];

(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-6,6},
  {y,-6,6},
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-Sigmoidal: *)
realSplitSigmoidalPlot=createSplitSigmoidalPlot[
  Re[splitsigmoidal],
  "Real Part of split-sigmoidalF(z)"
];

imaginarySplitSigmoidalPlot=createSplitSigmoidalPlot[
  Im[splitsigmoidal],
  "Imaginary Part of split-sigmoidalF(z)"
];

magnitudeSplitSigmoidalPlot=createSplitSigmoidalPlot[
  Abs[splitsigmoidal],
  "Magnitude of split-sigmoidalF(z)"
];

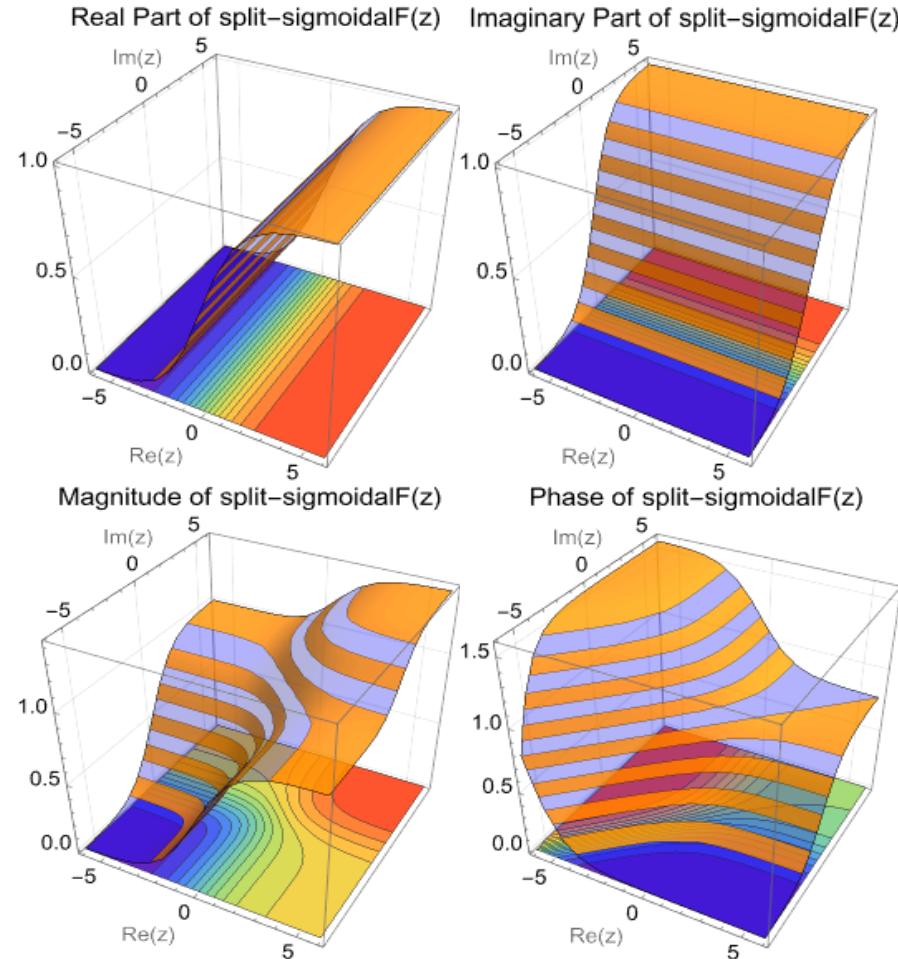
```

```

];
phaseSplitSigmoidalPlot=createSplitSigmoidalPlot[
  Arg[splitsigmoidal],
  "Phase of split-sigmoidalF(z)"
];

(* Display all four plots together in a list: *)
{realSplitSigmoidalPlot,imaginarySplitSigmoidalPlot,magnitudeSplitSigmoidalPlot,p
haseSplitSigmoidalPlot}

```

Output**Mathematica Code 10.21**

```

Input (* Define the split-sigmoidal activation function: *)
sigmoidalR[u_]:=1/(1+Exp[-u]);
splitsigmoidal:=sigmoidalR[Re[z]]+I sigmoidalR[Im[z]];

(* Generate a 3D plot of the split-sigmoidal function over a specified complex
range: *)
ComplexPlot3D[
  splitsigmoidal,
  {z,-6-6 I,6+6 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","splitSigmoidal(z)"}
]

```

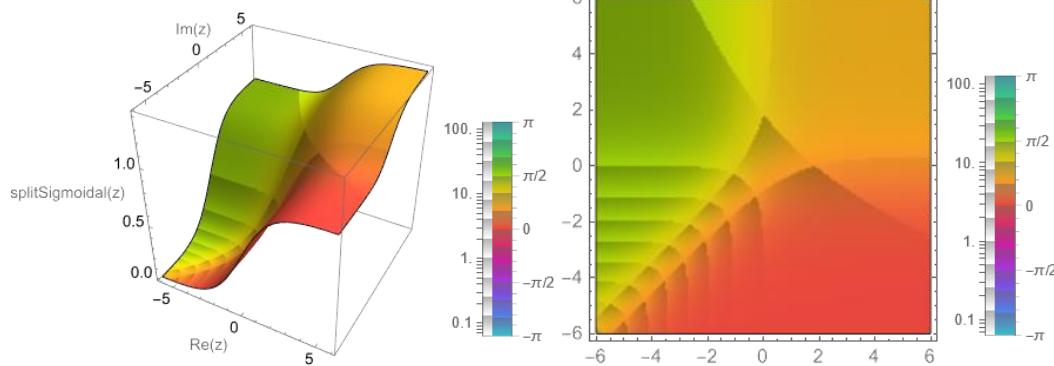
```

ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->270
]

(* Generate a 2D complex plot of the split-sigmoidal function over a specified
complex range: *)
ComplexPlot[
splitsigmoidal,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]

```

Output

**Mathematica Code 10.22**

Input (* The code defines a function `createSplitPSigmoidalPlot` to visualize various components (real part, imaginary part, magnitude, and phase) of the Split-Parametric Sigmoidal CVAF (Complex-Valued Activation Function), which applies a parametric sigmoidal function separately to the real and imaginary parts of the complex variable $z=x+iy$. By modularly generating 3D surface plots and slice contour plots for each component over the range [-6,6] for both real and imaginary parts, the function combines these plots for comprehensive visualization. This approach efficiently produces and displays plots for the real part, imaginary part, magnitude, and phase of the Split-Parametric Sigmoidal CVAF, allowing for a detailed comparison and understanding of its behavior in the complex plane: *)

```

(* Set parameters: *)
c1=0.7;
c2=3;

(* Define a function to create the plots for different components of the Split-
Parametric Sigmoidal CVAF: *)
createSplitPSigmoidalPlot[component_,label_]:=Module[
{plot1,slice},
(* Define a complex variable: *)
znumber=x+I*y;
(* Define the Split-Parametric Sigmoidal CVAF: *)
oSPSigmoidalR[znumber_,c1_,c2_]:=(2 c1)/(1+Exp[-c2 znumber])-c1;
oSPSigmoidal:=oSPSigmoidalR[x,c1,c2]+I oSPSigmoidalR[y,c1,c2];

```

```

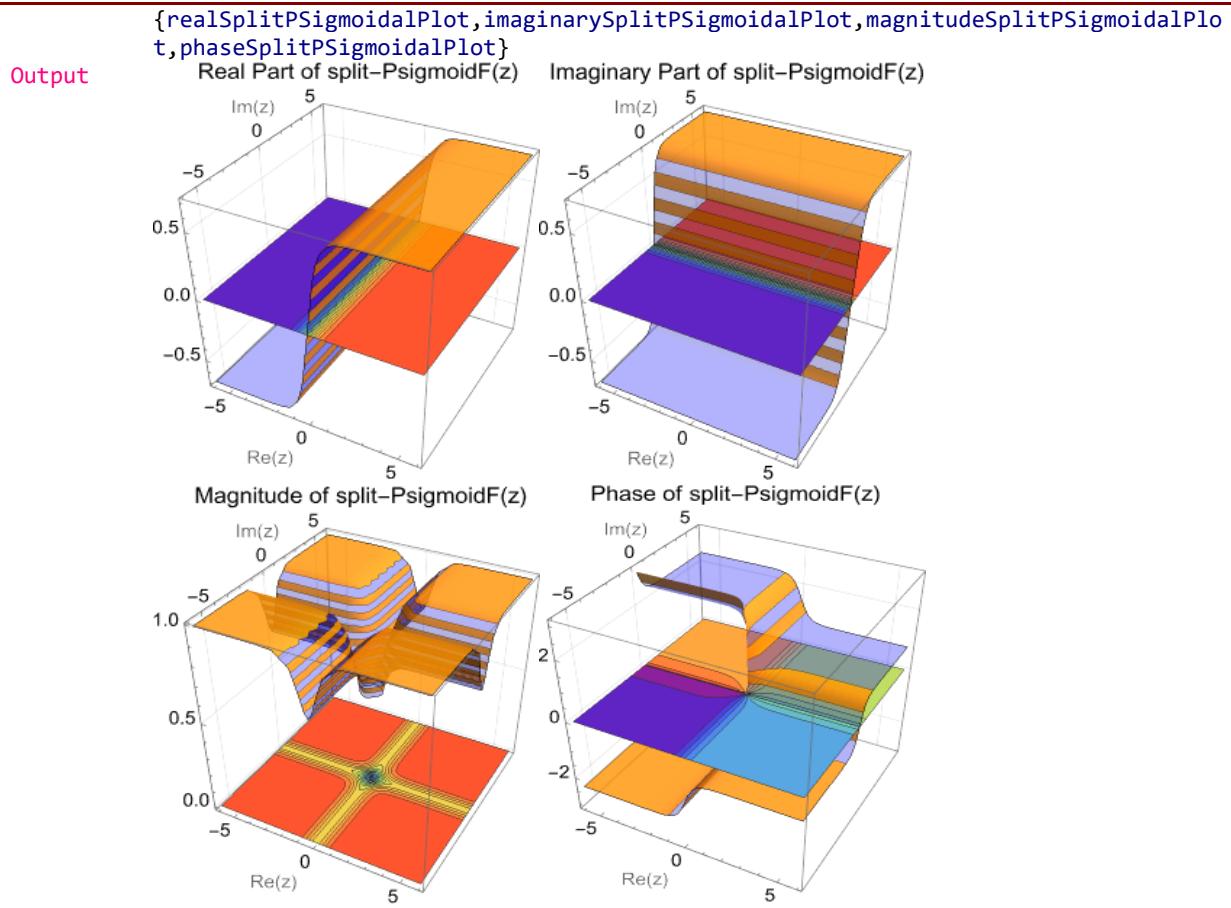
(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-6,6},
  {y,-6,6},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-Parametric Sigmoidal CVAF: *)
realSplitPSigmoidalPlot=createSplitPSigmoidalPlot[
  Re[\sigmaSPSigmoidal],
  "Real Part of split-Psigmof(z)"
];
imaginarySplitPSigmoidalPlot=createSplitPSigmoidalPlot[
  Im[\sigmaSPSigmoidal],
  "Imaginary Part of split-Psigmof(z)"
];
magnitudeSplitPSigmoidalPlot=createSplitPSigmoidalPlot[
  Abs[\sigmaSPSigmoidal],
  "Magnitude of split-Psigmof(z)"
];
phaseSplitPSigmoidalPlot=createSplitPSigmoidalPlot[
  Arg[\sigmaSPSigmoidal],
  "Phase of split-Psigmof(z)"
];
(* Display all four plots together in a list: *)

```

**Mathematica Code 10.23**

Input (* This Manipulate allows you to dynamically adjust the values of c1 and c2 using sliders, providing an interactive way to observe the changes in the 3D plot and ContourPlot of the Magnitude of the Split-Parametric Sigmoidal CVAF: *)

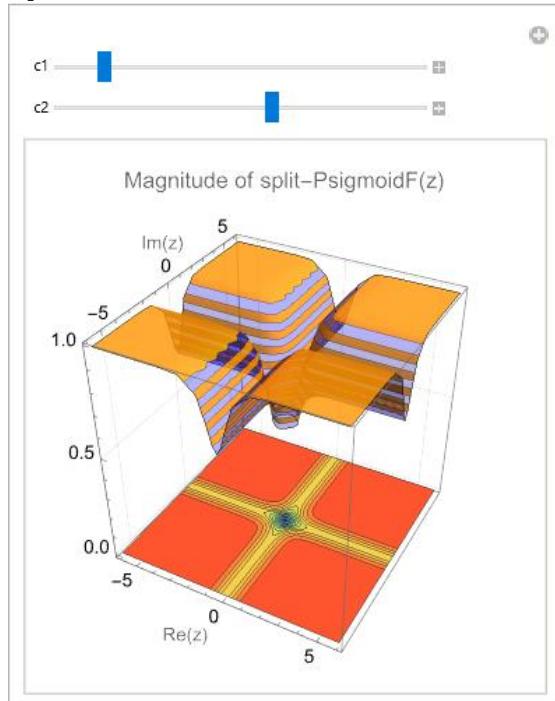
```
Manipulate[
Module[
{plot1,slice,x,y},
(* Define a complex variable: *)
znumber=x+I*y;
(* Define the Split-Parametric Sigmoidal CVAF and its derivative: *)
oSPSigmoidalR[znumber_,c1_,c2_]:=(2 c1)/(1+Exp[-c2 znumber])-c1;
oSPSigmoidal:=oSPSigmoidalR[x,c1,c2]+I oSPSigmoidalR[y,c1,c2];
(* Generate the 3D plot of the magnitude of the Split-Parametric Sigmoidal CVAF: *)
]
plot1=Plot3D[
Abs[oSPSigmoidal],
{x,-6,6},
{y,-6,6},
(* Plot options: *)
ClippingStyle->None,
AxesLabel->{"Re(z)", "Im(z)" },
MeshFunctions->{#3&},
Mesh->15,
MeshStyle->Opacity[.5],
MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
Lighting->"Neutral"
]
```

```

];
(* Generate the slice contour plot of the magnitude of the Split-Parametric
Sigmoidal CVAF: *)
slice=SliceContourPlot3D[
  Abs[oSPSigmoidal],
  z==0,
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];
(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  PlotRange->All,
  PlotLabel->"Magnitude of split-PsigmoidF(z)",
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->250
]
],
(* Interactive sliders for adjusting parameters c1 and c2: *)
{{c1,0.7,"c1"},0.1,5,0.1},
{{c2,3,"c2"},0.1,5,0.1}
]

```

Output

**Mathematica Code 10.24**

Input	(* Set parameters: *) c1=0.7;
-------	----------------------------------

```

c2=3;

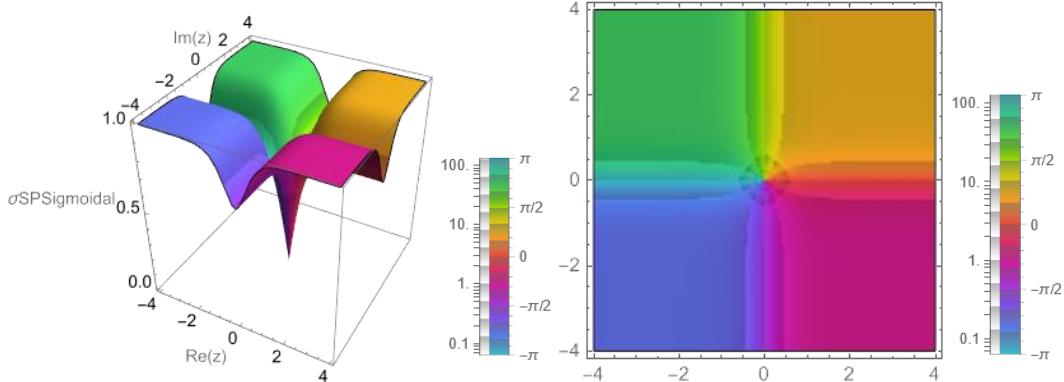
(* Define the Split-Parametric Sigmoidal CVAF: *)
oSPSigmoidalR[z_,c1_,c2_]:= (2 c1)/(1+Exp[-c2 z])-c1;
oSPSigmoidal:=oSPSigmoidalR[Re[z],c1,c2]+I oSPSigmoidalR[Im[z],c1,c2];

(* Generate a 3D plot of the Split-Parametric Sigmoidal function over a specified
complex range: *)
ComplexPlot3D[
oSPSigmoidal,
{z,-4-4 I,4+4 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)","Im(z)","oSPSigmoidal"},
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->250
]

(* Generate a 2D complex plot of the Split-Parametric Sigmoidal function over a
specified complex range: *)
ComplexPlot[
oSPSigmoidal,
{z,-4-4 I,4+4 I},
(* Plot options: *)
PlotRange->All,
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]

```

Output

**Mathematica Code 10.25**

Input (* The code defines a function `createSplitTanhPlot` to visualize various components (real part, imaginary part, magnitude, and phase) of the Split-Tanh function, which applies the hyperbolic tangent (tanh) separately to the real and imaginary parts of the complex variable $z=x+iy$. By modularly generating 3D surface plots and slice contour plots for each component over the range [-3,3] for both real and imaginary parts, the function combines these plots for comprehensive visualization. This approach efficiently produces and displays plots for the real part, imaginary part, magnitude, and phase of the Split-Tanh function, allowing for a detailed comparison and understanding of its behavior in the complex plane: *)

```

(* Define a function to create the plots for different components of the Split-
Tanh function: *)
createSplitTanhPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Define a complex variable: *)
  znumber=x+I*y;
  (* Define the Split-Tanh function: *)
  splitTanh:=Tanh[Re[znumber]]+I*Tanh[Im[znumber]];

  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-3,3},
    {y,-3,3},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)","Im(z)"},
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
    Lighting->"Neutral"
  ];

  (* Generate the slice contour plot of the specified component: *)
  slice=SliceContourPlot3D[
    component,
    z==0,(* Define the slicing plane at z=0 *)
    {x,-3,3},
    {y,-3,3},
    {z,-1,1},
    (* Plot options: *)
    Contours->15,
    Axes->False,
    PlotPoints->50,
    PlotRangePadding->0,
    ColorFunction->"Rainbow"
  ];

  (* Combine the 3D plot and the slice contour plot: *)
  Show[
    plot1,
    slice,
    (* Plot options: *)
    PlotRange->All,
    PlotLabel->label,
    BoxRatios->{1,1,1},
    FaceGrids->{Back,Left},
    ImageSize->200
  ]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-Tanh: *)
realSplitTanhPlot=createSplitTanhPlot[
  Re[splitTanh],
  "Real Part of split-Tanh(z)"
];

imaginarySplitTanhPlot=createSplitTanhPlot[

```

```

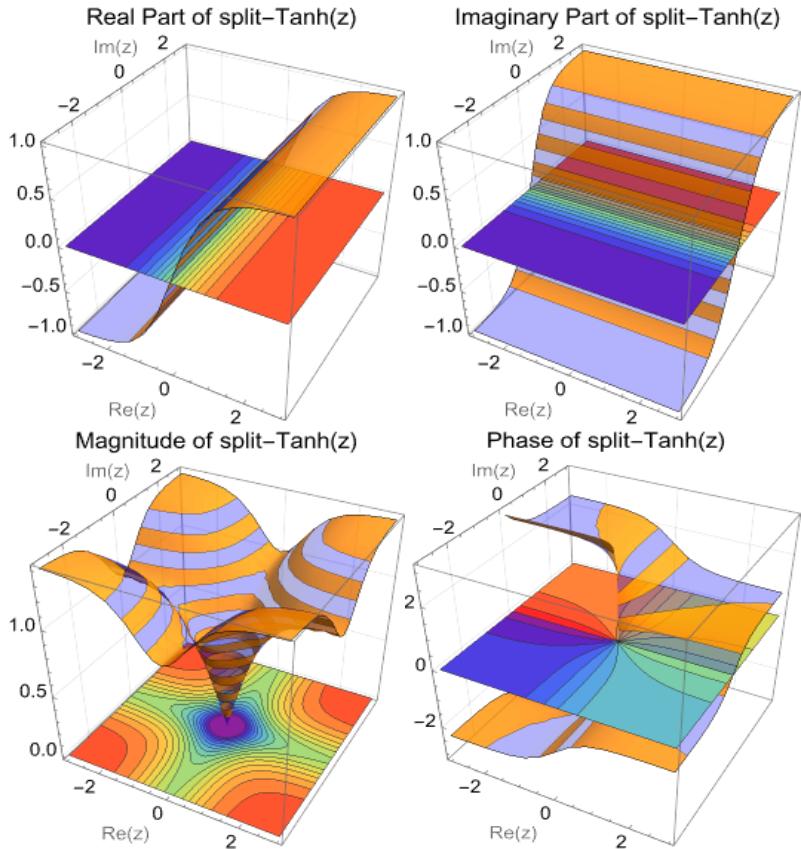
Im[splitTanh],
"Imaginary Part of split-Tanh(z)"
];

magnitudeSplitTanhPlot=createSplitTanhPlot[
Abs[splitTanh],
"Magnitude of split-Tanh(z)"
];

phaseSplitTanhPlot=createSplitTanhPlot[
Arg[splitTanh],
"Phase of split-Tanh(z)"
];

(* Display all four plots together in a list: *)
{realSplitTanhPlot,imaginarySplitTanhPlot,magnitudeSplitTanhPlot,phaseSplitTanhPlot}

```

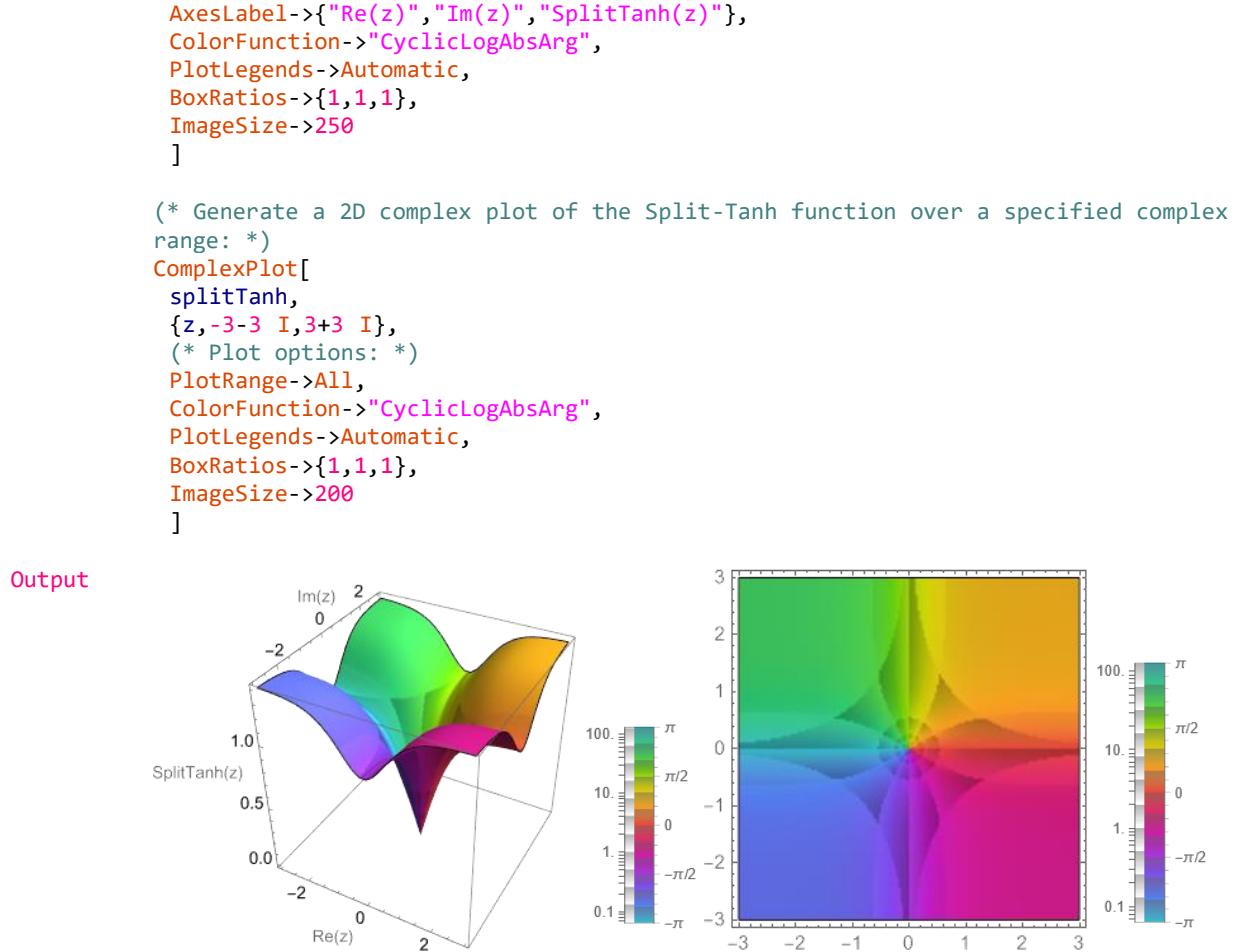
Output**Mathematica Code 10.26**

```

Input (* Define the Split-Tanh function: *)
splitTanh:=Tanh[Re[z]]+I*Tanh[Im[z]];

(* Generate a 3D plot of the Split-Tanh function over a specified complex range:
*)
ComplexPlot3D[
splitTanh,
{z,-3-3 I,3+3 I},
(* Plot options: *)
PlotRange->All,

```

**Mathematica Code 10.27**

Input (* The code defines a function `createSplitSTanhPlot` to visualize various components (real part, imaginary part, magnitude, and phase) of the Split-Sigmoidal Tanh function, which applies a modified tanh transformation separately to the real and imaginary parts of the complex variable $z=x+iy$. By modularly generating 3D surface plots and slice contour plots for each component over the range [-6,6] for both real and imaginary parts, the function combines these plots for comprehensive visualization. This approach efficiently produces and displays plots for the real part, imaginary part, magnitude, and phase of the Split-Sigmoidal Tanh function, allowing for a detailed comparison and understanding of its behavior in the complex plane: *)

```

(* Define a function to create the plots for different components of the Split-  

Sigmoidal Tanh function: *)  

createSplitSTanhPlot[component_,label_]:=Module[  

  {plot1,slice},  

  (* Define a complex variable: *)  

  znumber=x+Iy;  

  (* Define the Split-Sigmoidal Tanh function: *)  

  rePart=Tanh[Re[znumber]]/(1-(Re[znumber]-3) Exp[-Re[znumber]]);  

  imPart=Tanh[Im[znumber]]/(1-(Im[znumber]-3) Exp[-Im[znumber]]);  

  splitSTanh:=rePart+I imPart;  
  

  (* Generate the 3D plot of the specified component: *)  

  plot1=Plot3D[

```

```

component,
{x,-6,6},
{y,-6,6},
(* Plot options: *)
ClippingStyle->None,
AxesLabel->{"Re(z)", "Im(z)" },
MeshFunctions->{#3&},
Mesh->15,
MeshStyle->Opacity[.5],
MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

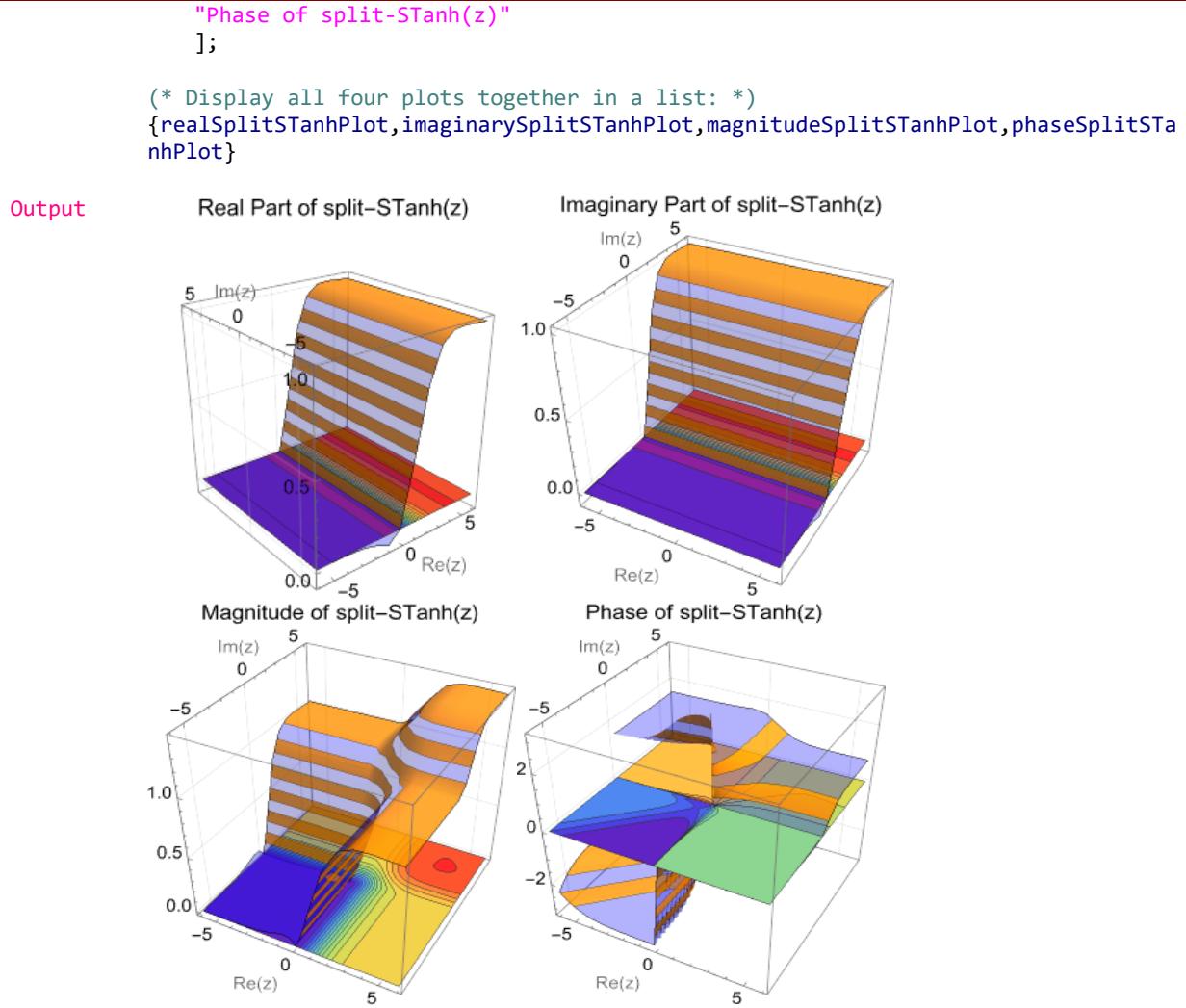
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-Sigmoidal Tanh: *)
realSplitSTanhPlot=createSplitSTanhPlot[
  Re[splitSTanh],
  "Real Part of split-STanh(z)"
];

imaginarySplitSTanhPlot=createSplitSTanhPlot[
  Im[splitSTanh],
  "Imaginary Part of split-STanh(z)"
];

magnitudeSplitSTanhPlot=createSplitSTanhPlot[
  Abs[splitSTanh],
  "Magnitude of split-STanh(z)"
];

phaseSplitSTanhPlot=createSplitSTanhPlot[
  Arg[splitSTanh],
  "Phase of split-STanh(z)"
];

```

**Mathematica Code 10.28**

```

Input (* Define the Split-Sigmoidal Tanh function: *)
rePart=Tanh[Re[z]]/(1-(Re[z]-3) Exp[-Re[z]]);
imPart=Tanh[Im[z]]/(1-(Im[z]-3) Exp[-Im[z]]);
splitSTanh:=rePart+I imPart;

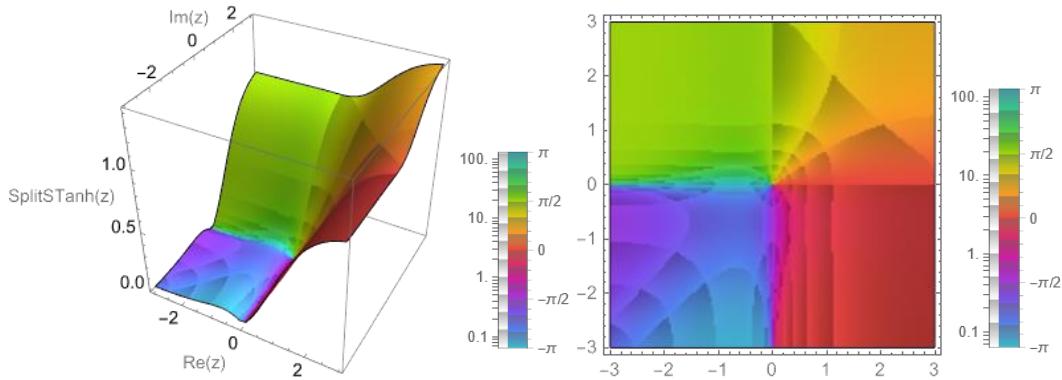
(* Generate a 3D plot of the Split-Sigmoidal Tanh function over a specified complex range: *)
ComplexPlot3D[
  splitSTanh,
  {z,-3-3 I,3+3 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","SplitSTanh(z)"},
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->250
]

(* Generate a 2D complex plot of the Split-Sigmoidal Tanh function over a specified complex range: *)

```

```
ComplexPlot[
  splitSTanh,
  {z, -3-3 I, 3+3 I},
  (* Plot options: *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->200
]
```

Output

**Mathematica Code 10.29**

Input (* The code defines a function `createSplitCReLUPlot` to visualize various components (real part, imaginary part, magnitude, and phase) of the Split-CReLU function, which applies a Rectified Linear Unit (ReLU) separately to the real and imaginary parts of the complex variable $z=x+iy$. By modularly generating 3D surface plots and slice contour plots for each component over the range [-6,6] for both real and imaginary parts, the function combines these plots for comprehensive visualization. This approach efficiently produces and displays plots for the real part, imaginary part, magnitude, and phase of the Split-CReLU function, allowing for a detailed comparison and understanding of its behavior in the complex plane: *)

```
(* Define a function to create the plots for different components of the Split-
CReLU function: *)
createSplitCReLUPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Define a complex variable: *)
  znumber=x+I*y;
  (* Define the Split-CReLU function: *)
  rePart=Max[Re[znumber],0];
  imPart=Max[Im[znumber],0];
  splitReLU:=rePart+I imPart;
  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-6,6},
    {y,-6,6},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)", "Im(z)" },
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
    Lighting->"Neutral"
  ]
  slice=ContourPlot[
    component,
    {x,-6,6},
    {y,-6,6},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)", "Im(z)" },
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
    Lighting->"Neutral"
  ]
  (* Create a grid of 3D plots for each component *)
  grid=Grid[{{plot1,slice}},Frame->True]
  (* Add labels to the grid *)
  grid=Grid[{{grid,Text[component]}},Frame->True]
  (* Return the final labeled grid *)
  grid
]
```

```
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

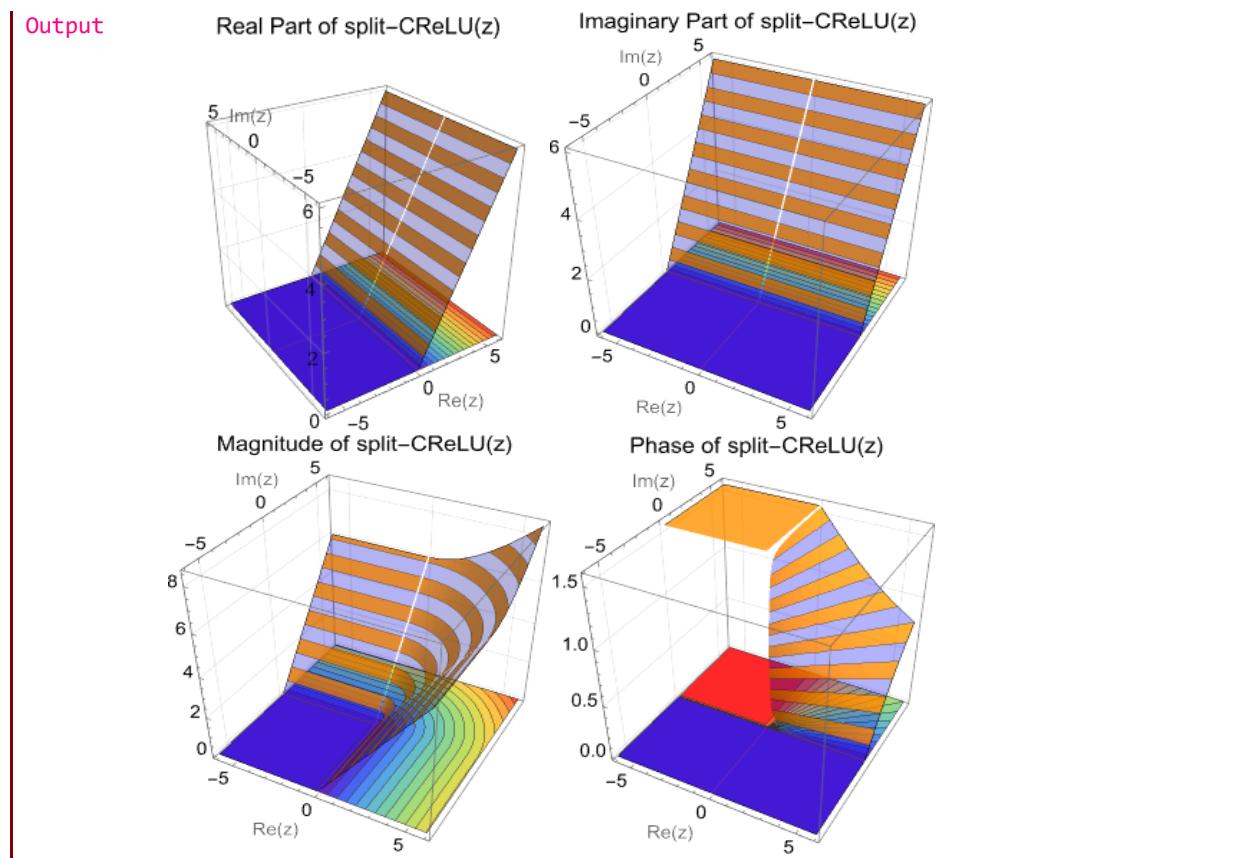
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-CReLU: *)
realSplitCReLUPlot=createSplitCReLUPlot[
  Re[splitReLU],
  "Real Part of split-CReLU(z)"
];

imaginarySplitCReLUPlot=createSplitCReLUPlot[
  Im[splitReLU],
  "Imaginary Part of split-CReLU(z)"
];

magnitudeSplitCReLUPlot=createSplitCReLUPlot[
  Abs[splitReLU],
  "Magnitude of split-CReLU(z)"
];

phaseSplitCReLUPlot=createSplitCReLUPlot[
  Arg[splitReLU],
  "Phase of split-CReLU(z)"
];

(* Display all four plots together in a list: *)
{realSplitCReLUPlot,imaginarySplitCReLUPlot,magnitudeSplitCReLUPlot,phaseSplitCReLUPlot}
```

**Mathematica Code 10.30**

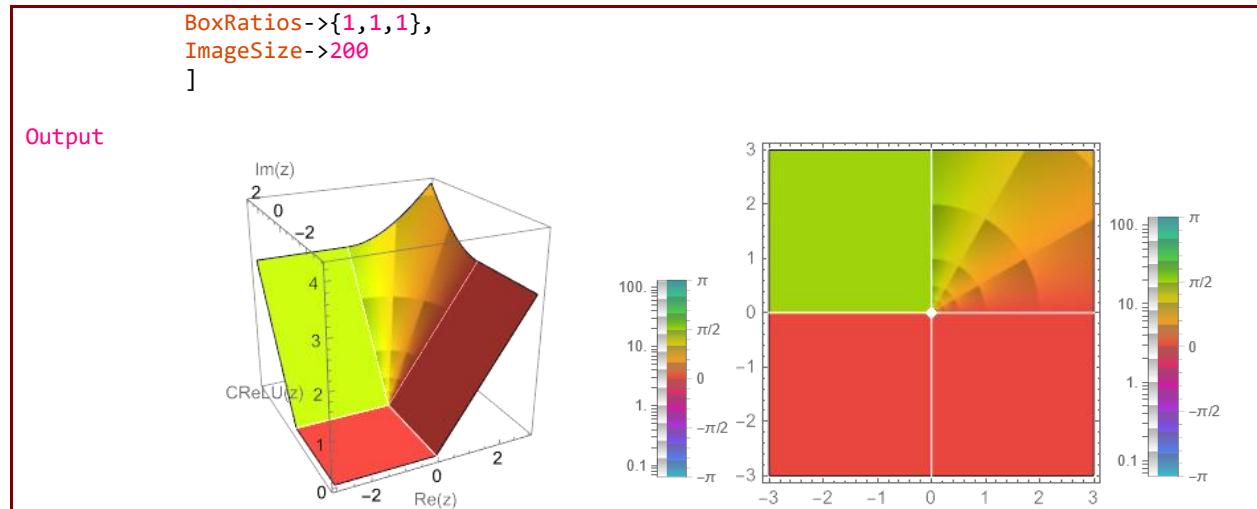
```

Input      (* Define the real and imaginary parts of the Split-CReLU function: *)
rePart=Max[Re[z],0];
imPart=Max[Im[z],0];
splitReLU:=rePart+I imPart;

(* Generate a 3D plot of the Split-CReLU function over a specified complex range: *)
ComplexPlot3D[
  splitReLU,
  {z,-3-3 I,3+3 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","CReLU(z)"},
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->250
]

(* Generate a 2D complex plot of the Split-CReLU function over a specified complex range: *)
ComplexPlot[
  splitReLU,
  {z,-3-3 I,3+3 I},
  (* Plot options: *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
]

```

**Mathematica Code 10.31**

Input

```
(* The code defines a modular function `createSplitQAMPlot` to visualize different components (real part, imaginary part, magnitude, and phase) of the Split-QAM function, which involves a sinusoidal transformation of a complex variable  $z=x+iy$  with a slope parameter (alpha). By generating 3D surface plots and slice contour plots for each component over the range [-6,6] for both real and imaginary parts, the function combines these plots for comprehensive visualization. This approach efficiently produces and displays the plots, allowing for a detailed comparison and understanding of the Split-QAM function's behavior in the complex plane. The final output includes all four generated plots presented together for easy comparison: *)
```

```
(* Define a function to create the plots for different components of the Split-QAM function: *)
createSplitQAMPlot[component_,label_]:=Module[
{plot1,slice},
```

```
(* Define a complex variable: *)
znumber=x+I*y;
```

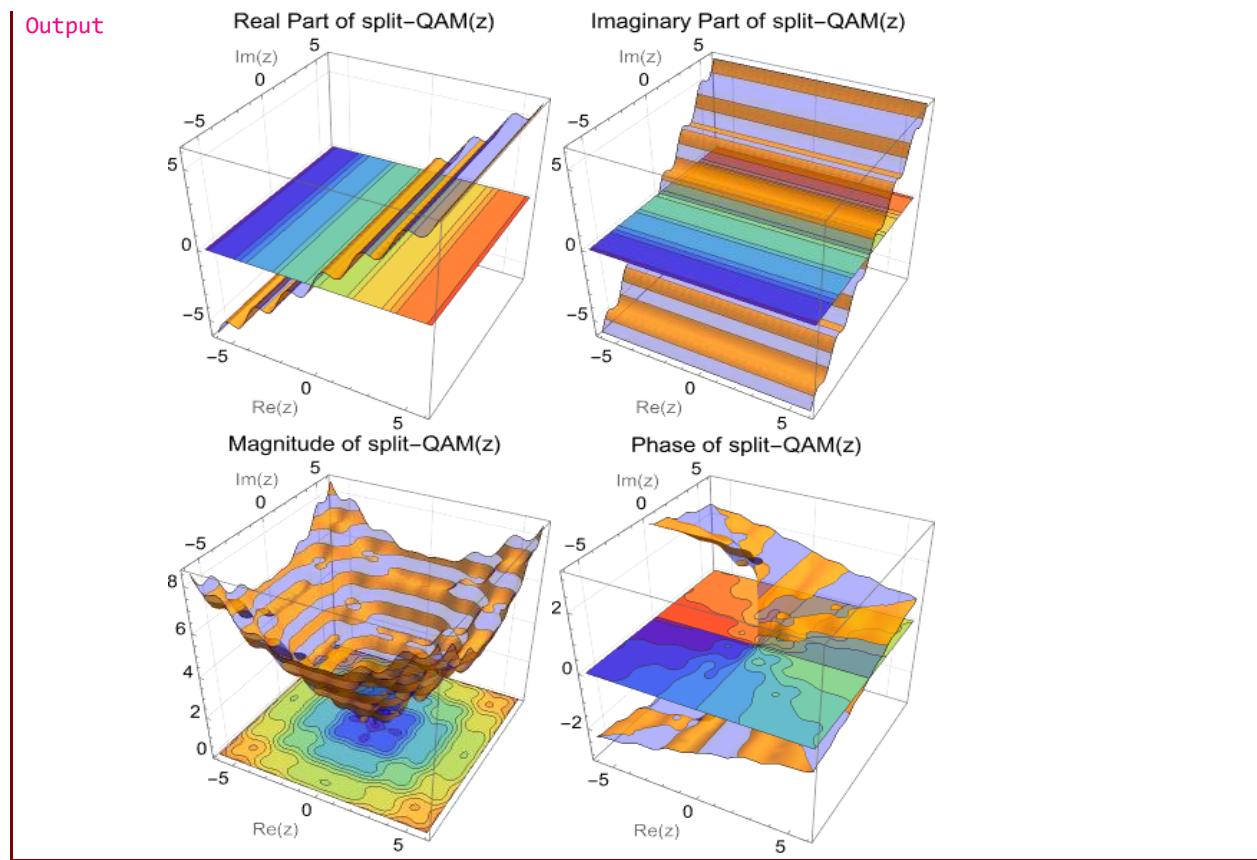
```
(* Value of slope parameter alpha: *)
alpha=0.5;
```

```
(* Define the Split-QAM function: *)
rePart=Re[znumber]+alpha Sin[Pi Re[znumber]];
imPart=Im[znumber]+alpha Sin[Pi Im[znumber]];
SplitQAM:=rePart+I imPart;
```

```
(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-6,6},
  {y,-6,6},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
 ]
```

```
];
(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(*Define the slicing plane at z=0*)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];
(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-QAM: *)
realSplitQAMPlot=createSplitQAMPlot[
  Re[SplitQAM],
  "Real Part of split-QAM(z)"
];
imaginarySplitQAMPlot=createSplitQAMPlot[
  Im[SplitQAM],
  "Imaginary Part of split-QAM(z)"
];
magnitudeSplitQAMPlot=createSplitQAMPlot[
  Abs[SplitQAM],
  "Magnitude of split-QAM(z)"
];
phaseSplitQAMPlot=createSplitQAMPlot[
  Arg[SplitQAM],
  "Phase of split-QAM(z)"
];
(* Display all four plots together in a list: *)
{realSplitQAMPlot,imaginarySplitQAMPlot,magnitudeSplitQAMPlot,phaseSplitQAMPlot}
```

**Mathematica Code 10.32**

Input (* This Manipulate allows you to dynamically adjust the value of α using sliders, providing an interactive way to observe the changes in the 3D plot and ContourPlot of the real Part of split-QAM(z) CVAF. *)

```

Manipulate[
Module[
{plot1,slice,x,y},
(* Define a complex variable: *)
znumber=x+I*y;

(* Define the split-QAM function: *)
rePart=Re[znumber]+ $\alpha$  Sin[Pi Re[znumber]];
imPart=Im[znumber]+ $\alpha$  Sin[Pi Im[znumber]];
SplitQAM:=rePart+I imPart;

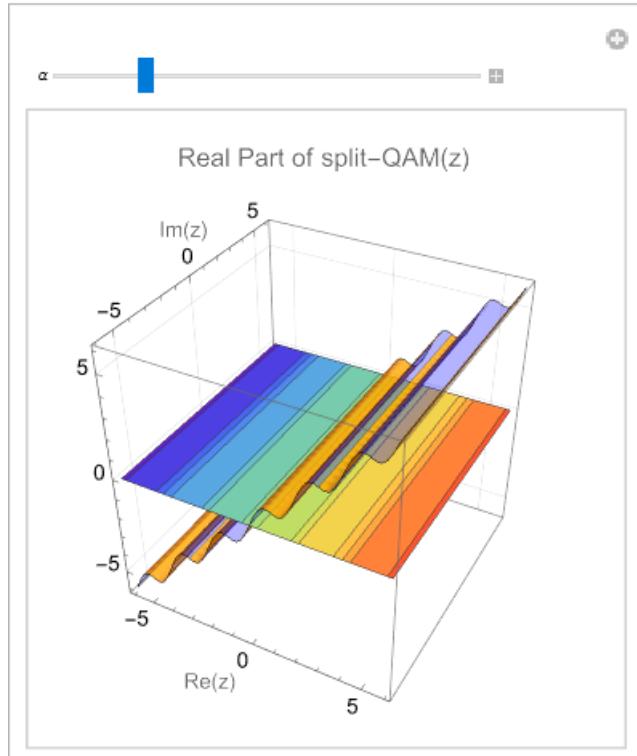
(*Generate the 3D plot of the real part of Split-QAM*)
plot1=Plot3D[
  Re[SplitQAM],
  {x,-6,6},
  {y,-6,6},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
]

```

```
];
(* Generate the slice contour plot of the real part of Split-QAM: *)
slice=SliceContourPlot3D[
  Re[SplitQAM],
  z==0,
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->"Real Part of split-QAM(z)",
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->250
]
],
(* Interactive slider for adjusting the parameter  $\alpha$ : *)
{{\alpha,0.5," $\alpha$ "},0.1,2,0.1}
]
```

Output



Mathematica Code 10.33

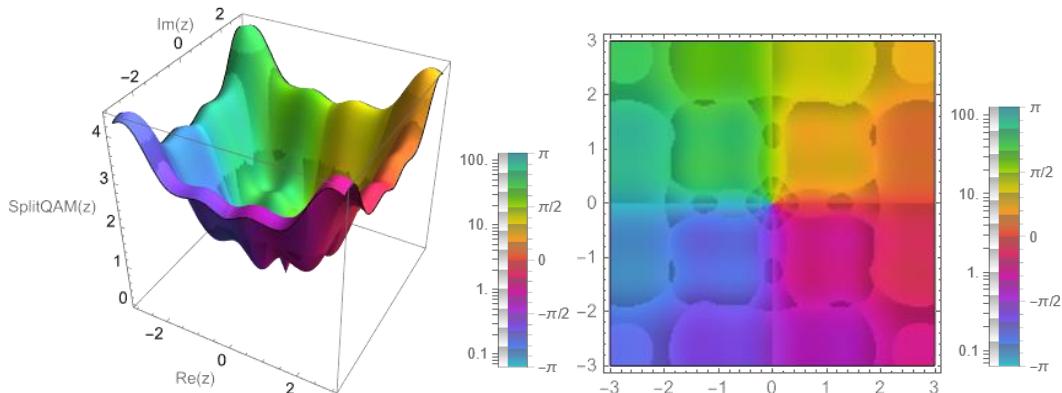
```

Input      (* Define the split-QAM function: *)
rePart=Re[z]+a Sin[Pi Re[z]];
imPart=Im[z]+a Sin[Pi Im[z]];
SplitQAM:=rePart+I imPart;

(* Generate a 3D plot of the split-QAM function over a specified complex range: *)
ComplexPlot3D[
 SplitQAM,
 {z,-3-3 I,3+3 I},
 (* Plot options: *)
 PlotRange->All,
 AxesLabel->{"Re(z)","Im(z)","SplitQAM(z)"},
 ColorFunction->"CyclicLogAbsArg",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->270
]

(* Generate a 2D complex plot of the split-QAM function over a specified complex
range: *)
ComplexPlot[
 SplitQAM,
 {z,-3-3 I,3+3 I},
 (* Plot options: *)
 PlotRange->All,
 ColorFunction->"CyclicLogAbsArg",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->200
]

```

Output**Mathematica Code 10.34**

```

Input      (* The code defines a function `createSplitAVPlot` to visualize various components
            (real part, imaginary part, magnitude, and phase) of the Split-absolute value
            (Split-AV)=Split-Hard Tanh function, which applies a custom transformation
            separately to the real and imaginary parts of the complex variable z=x+iy. By
            modularly generating 3D surface plots and slice contour plots for each component
            over the range [-6,6] for both real and imaginary parts, the function combines
            these plots for comprehensive visualization. This approach efficiently produces
            and displays plots for the real part, imaginary part, magnitude, and phase of the
            Split-AV Tanh function, allowing for a detailed comparison and understanding of
            its behavior in the complex plane: *)

```

```

(* Define a function to create the plots for different components of the Split-
absolute value (Split-AV)=Split-Hard Tanh function: *)
createSplitAVPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Define a complex variable: *)
  znumber=x+I*y;
  (* Define the Split-AV Tanh function: *)
  rePart=1/2 (Abs[Re[znumber]+1]-Abs[Re[znumber]-1]);
  imPart=1/2 (Abs[Im[znumber]+1]-Abs[Im[znumber]-1]);
  SplitAV:=rePart+I imPart;

  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-6,6},
    {y,-6,6},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)","Im(z)"},
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
    Lighting->"Neutral"
  ];

  (* Generate the slice contour plot of the specified component: *)
  slice=SliceContourPlot3D[
    component,
    z==0,(*Define the slicing plane at z=0*)
    {x,-6,6},
    {y,-6,6},
    {z,-1,1},
    (* Plot options: *)
    Contours->15,
    Axes->False,
    PlotPoints->50,
    PlotRangePadding->0,
    ColorFunction->"Rainbow"
  ];

  (* Combine the 3D plot and the slice contour plot: *)
  Show[
    plot1,
    slice,
    (* Plot options: *)
    PlotRange->All,
    PlotLabel->label,
    BoxRatios->{1,1,1},
    FaceGrids->{Back,Left},
    ImageSize->200
  ]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
Split-AV Tanh: *)
realSplitAVPlot=createSplitAVPlot[
  Re[SplitAV],
  "Real Part of Split-HardTanh(z)"
];

```

```

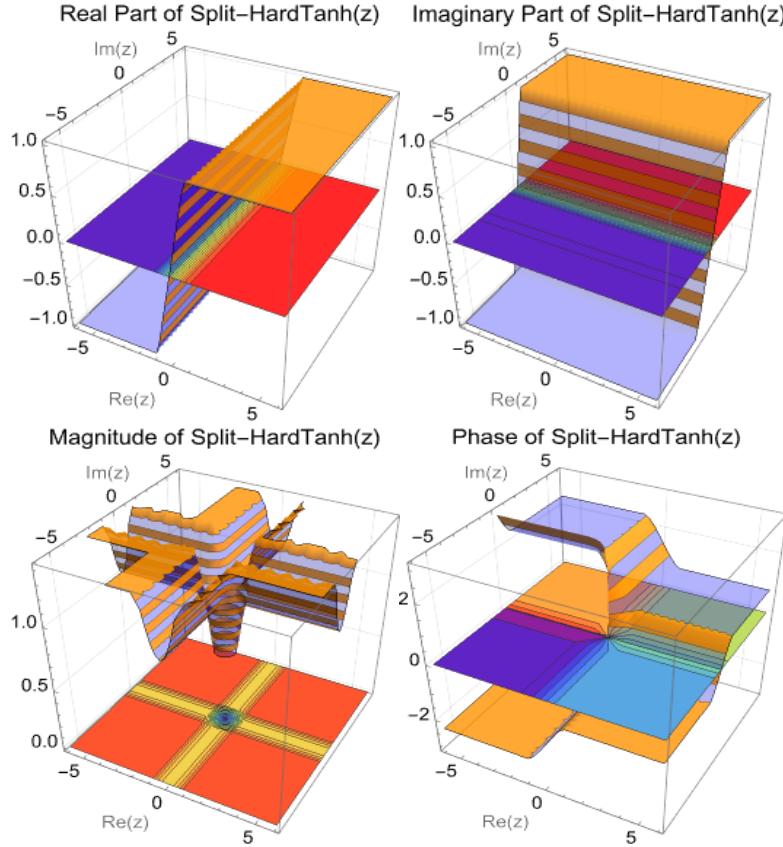
imagineySplitAVPlot=createSplitAVPlot[
  Im[SplitAV],
  "Imaginary Part of Split-HardTanh(z)"
];

magnitudeSplitAVPlot=createSplitAVPlot[
  Abs[SplitAV],
  "Magnitude of Split-HardTanh(z)"
];

phaseSplitAVPlot=createSplitAVPlot[
  Arg[SplitAV],
  "Phase of Split-HardTanh(z)"
];

(* Display all four plots together in a list: *)
{realSplitAVPlot,imagineySplitAVPlot,magnitudeSplitAVPlot,phaseSplitAVPlot}

```

Output**Mathematica Code 10.35**

```

Input      (* Define the Split-AV = Split- Hard Tanh function: *)

rePart=1/2 (Abs[Re[z]+1]-Abs[Re[z]-1]);
imPart=1/2 (Abs[Im[z]+1]-Abs[Im[z]-1]);
SplitAV:=rePart+I imPart;

(* Generate a 3D plot of the Split-Hard Tanh function over a specified complex
range: *)
ComplexPlot3D[

```

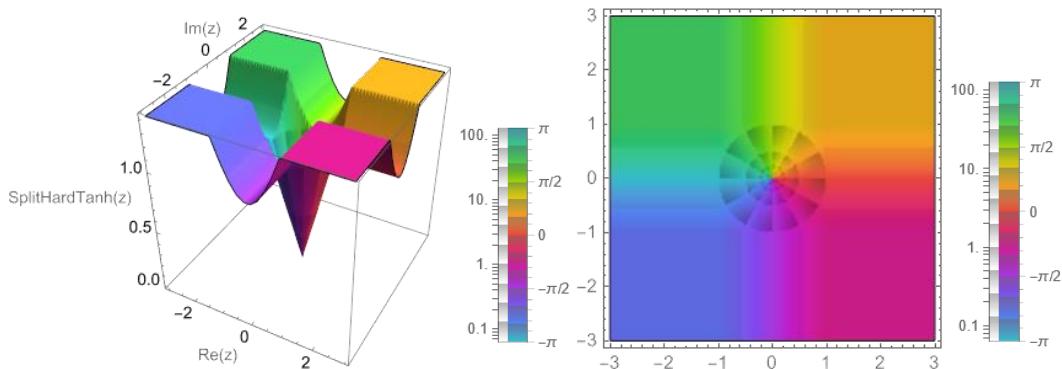
```

SplitAV,
{z,-3-3 I,3+3 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)","Im(z)","SplitHardTanh(z)"},
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->270
]

(* Generate a 2D complex plot of the Split-Hard Tanh function over a specified
complex range: *)
ComplexPlot[
SplitAV,
{z,-3-3 I,3+3 I},
(* Plot options: *)
PlotRange->All,
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]

```

Output

**Mathematica Code 10.36**

Input (* The code defines a function `createAFTAFPlot` that generates 3D surface plots and slice contour plots for different components (real part, imaginary part, magnitude, and phase) of the Amplitude-Phase-Type Activation Function (AFTAF), given by $AFTAF(z) = \tanh(|z|) \exp(i \arg(z))$. The function takes the component to be plotted and a label, then generates and combines the 3D and slice plots for the specified component. By calling `createAFTAFPlot` for each component, the code efficiently produces and displays plots for the real part, imaginary part, magnitude, and phase of the AFTAF function, allowing comprehensive visualization and comparison of these aspects over a specified complex range: *)

```

(*Define a function to create the plots for different components of the AFTAF
function*)
createAFTAFPlot[component_,label_]:=Module[
{plot1,slice},
(* Define a complex variable: *)
znumber=x+I*y;

(* Define the amplitude-phase-type activation function: *)
AFTAF:=Tanh[Abs[znumber]] Exp[I Arg[znumber]];

(* Generate the 3D plot of the specified component: *)

```

```
plot1=Plot3D[
  component,
  {x,-3,3},
  {y,-3,3},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(*Define the slicing plane at z=0*)
  {x,-3,3},
  {y,-3,3},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
AFTAF: *)
realAFTAFPlot=createAFTAFPlot[
  Re[AFTAF],
  "Real Part of APTF(z)"
];

imaginaryAFTAFPlot=createAFTAFPlot[
  Im[AFTAF],
  "Imaginary Part of APTF(z)"
];

magnitudeAFTAFPlot=createAFTAFPlot[
  Abs[AFTAF],
  "Magnitude of APTF(z)"
];

phaseAFTAFPlot=createAFTAFPlot[
```

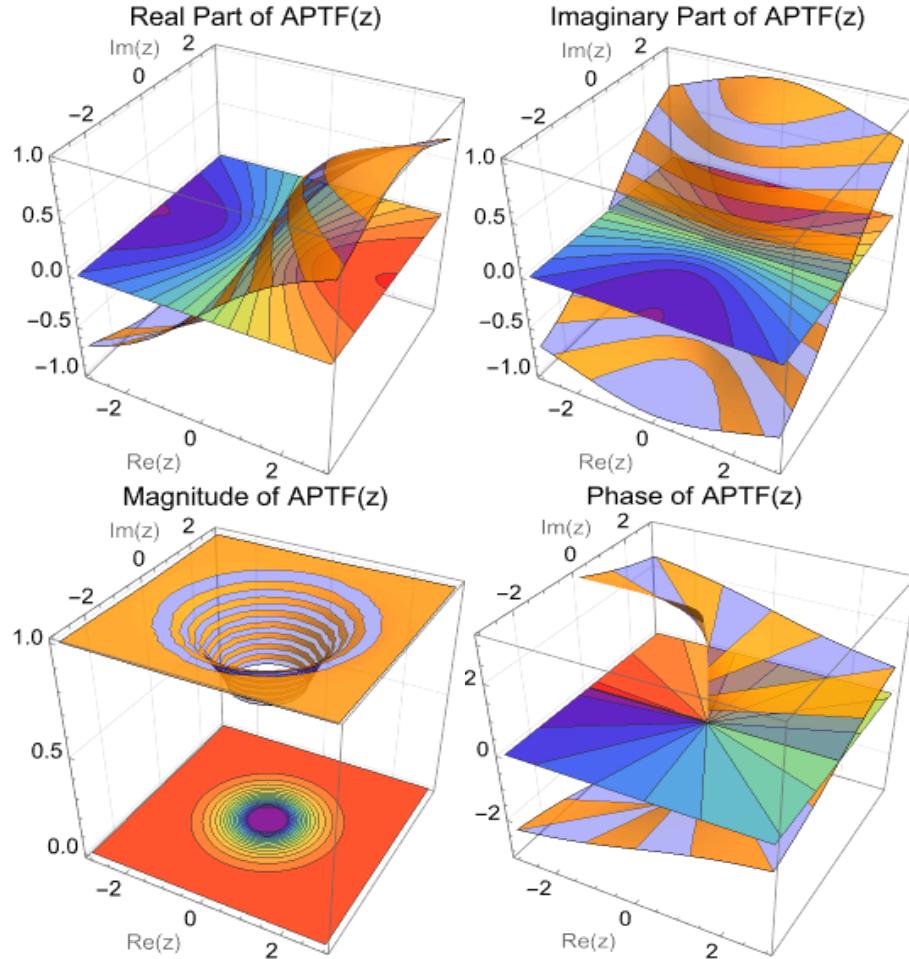
```

Arg[AFTAF],
"Phase of APTF(z)"
];

(* Display all four plots together in a list: *)
{realAFTAFPlot,imaginaryAFTAFPlot,magnitudeAFTAFPlot,phaseAFTAFPlot}

```

Output

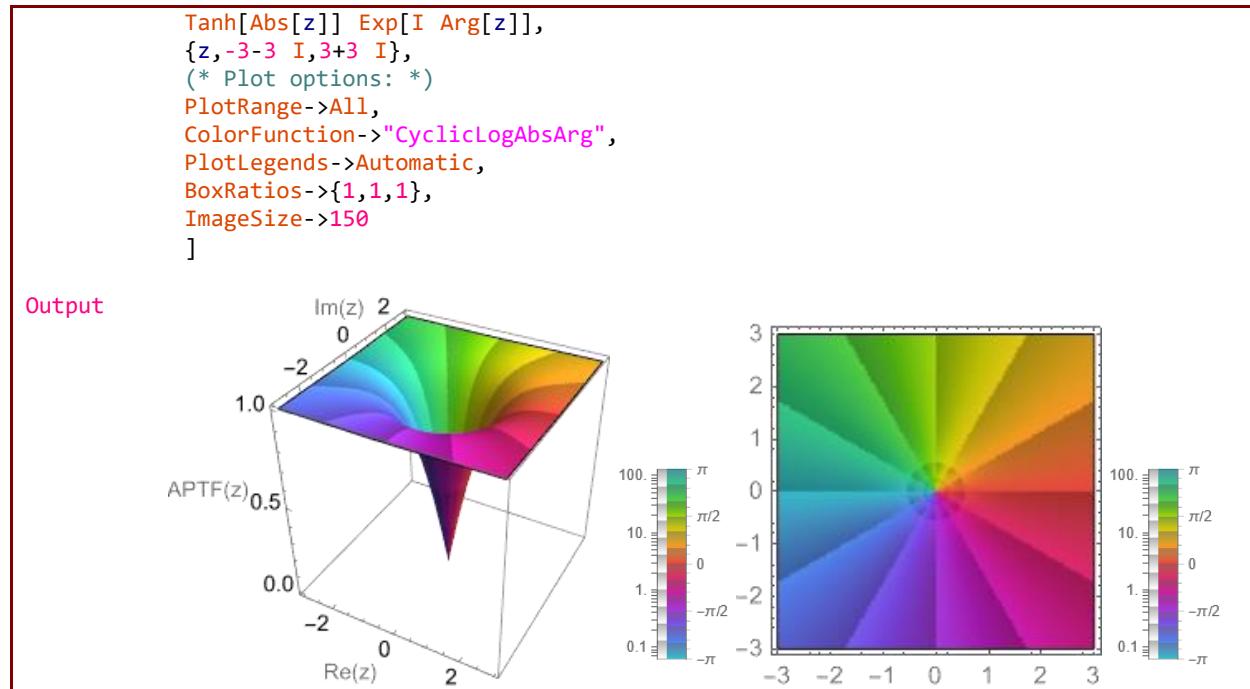
**Mathematica Code 10.37**

```

Input (* Define the amplitude-phase-type activation function and generate a 3D plot over
a specified complex range: *)
ComplexPlot3D[
Tanh[Abs[z]] Exp[I Arg[z]],
{z,-3-3 I,3+3 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)","Im(z)","APTF(z)"},
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]

(* Define the amplitude-phase-type activation function and generate a 2D complex
plot over a specified complex range: *)
ComplexPlot[

```

**Mathematica Code 10.38**

```

Input (* The code defines a function createAPSFPPlot that generates 3D surface plots and
       slice contour plots for different components (real part, imaginary part, magnitude,
       and phase) of the Amplitude-Phase Sigmoidal Function (APSF) defined as
       PASF=(b/(a*b+Abs[z]))*z with parameters a and b. The function takes the component
       to be plotted and a label, then generates and combines the 3D and slice plots for
       the specified component. By calling createAPSFPPlot for each component, the code
       efficiently produces and displays plots for the real part, imaginary part,
       magnitude, and phase of the APSF function, allowing comprehensive visualization
       and comparison of these aspects over a specified: *)

(* Define a function to create the plots for different components of the APSF
function: *)

createAPSFPPlot[component_,label_]:=Module[
{plot1,slice},
znumber=x+I*y;
a=1;
b=1;

(* Define the Amplitude-Phase Sigmoidal Function: *)
PASF:=(b/(a*b+Abs[znumber]))*znumber;

(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
component,
{x,-6,6},
{y,-6,6},
ClippingStyle->None,
AxesLabel->{"Re(z)","Im(z)"},
MeshFunctions->{#3&},
Mesh->15,
MeshStyle->Opacity[.5],
MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}}},

```

```
Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0: *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
APSF: *)

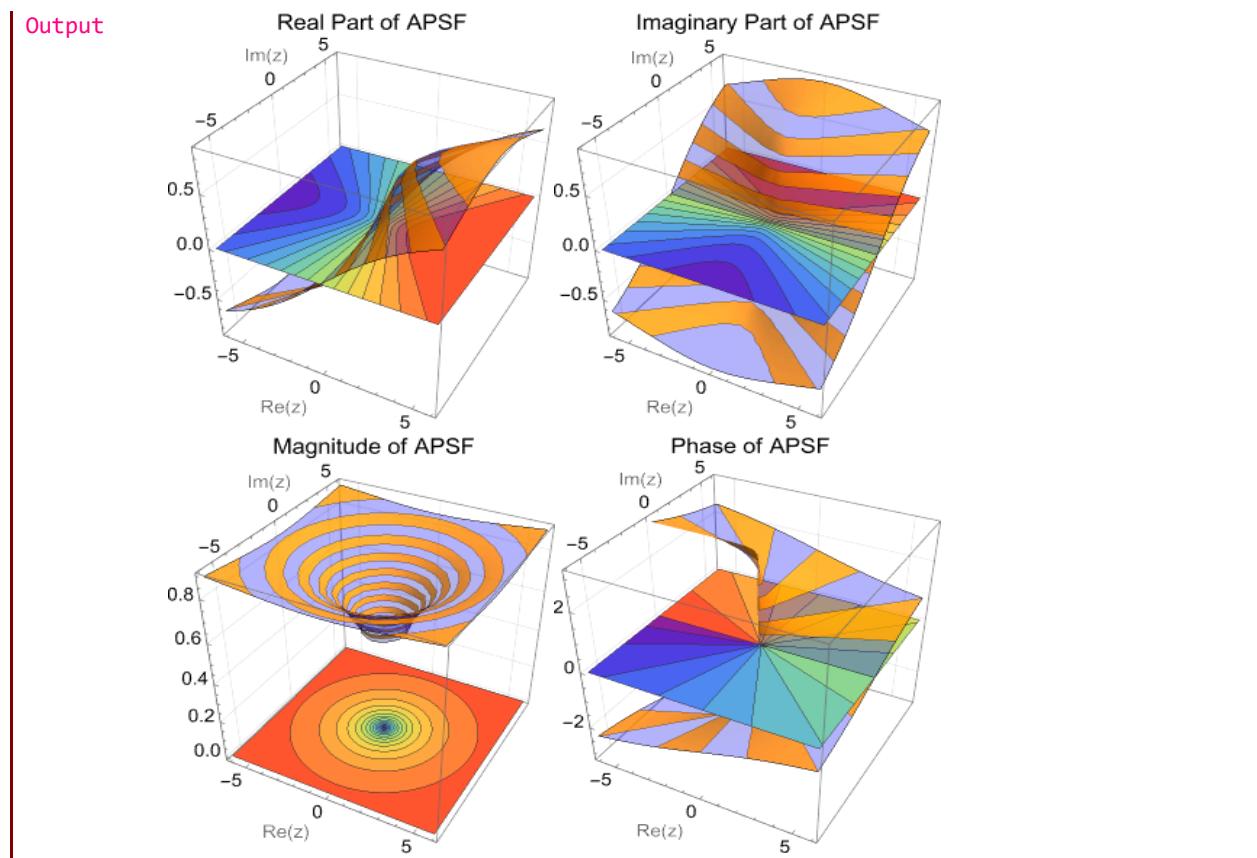
realAPSFPlot=createAPSFPlot[
  Re[PASF],
  "Real Part of APSF"
];

imaginaryAPSFPlot=createAPSFPlot[
  Im[PASF],
  "Imaginary Part of APSF"
];

magnitudeAPSFPlot=createAPSFPlot[
  Abs[PASF],
  "Magnitude of APSF"
];

phaseAPSFPlot=createAPSFPlot[
  Arg[PASF],
  "Phase of APSF"
];

(* Display all four plots together in a list: *)
{realAPSFPlot,imaginaryAPSFPlot,magnitudeAPSFPlot,phaseAPSFPlot}
```

**Mathematica Code 10.39**

```

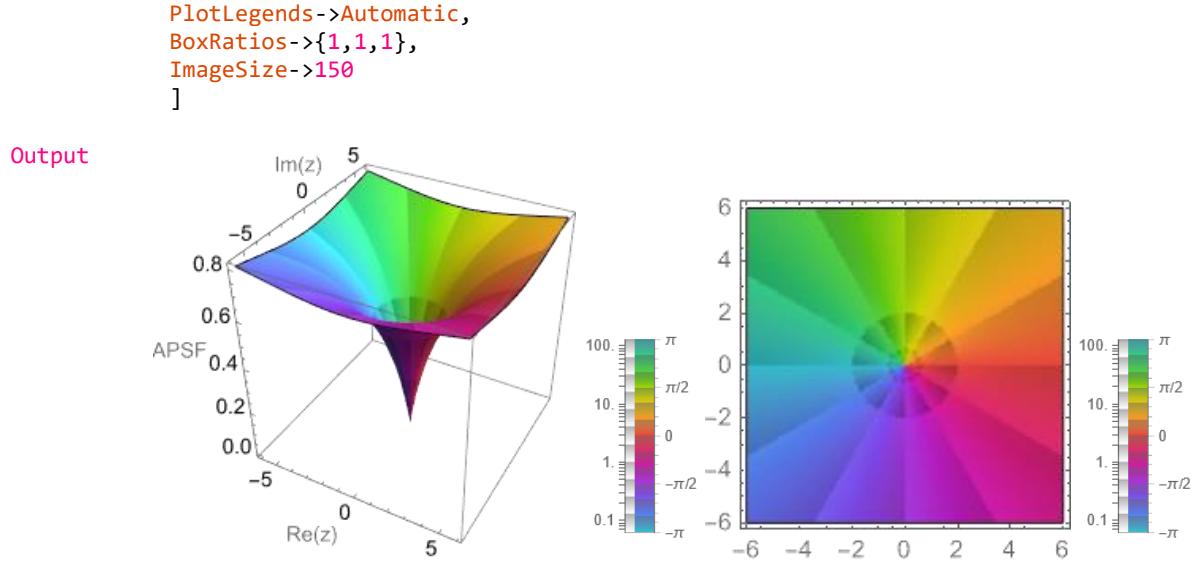
Input      (* Define the parameters a and b: *)
a=2;
b=1;

(* Define the Amplitude-Phase Sigmoidal Function (APSF): *)
PASF:=(b/(a*b+Abs[z]))*z

(* Generate a 3D plot of the APSF function over a specified complex range: *)
ComplexPlot3D[
 PASF,
 {z,-6-6 I,6+6 I},
 (* Plot options: *)
 PlotRange->All,
 AxesLabel->{"Re(z)","Im(z)","APSF"},
 ColorFunction->"CyclicLogAbsArg",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->200
]

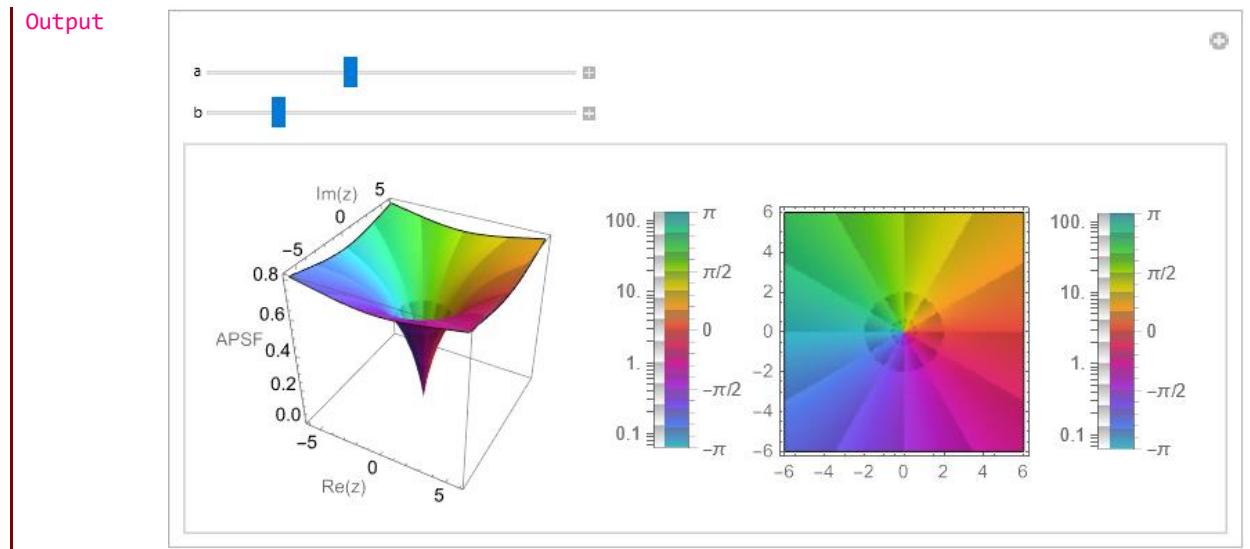
(* Generate a 2D complex plot of the APSF function over a specified complex
range: *)
ComplexPlot[
 PASF,
 {z,-6-6 I,6+6 I},
 (* Plot options: *)
 PlotRange->All,
 ColorFunction->"CyclicLogAbsArg",

```

**Mathematica Code 10.40**

Input (* The code provides an interactive interface for exploring the Amplitude-Phase Sigmoidal Function in both 3D and 2D plots. Users can dynamically adjust the parameters a and b to observe the impact on the function's behavior in real-time. The side-by-side display of the 3D and 2D plots allows for a comprehensive understanding of the function's characteristics: *)

```
Manipulate[
  PASF := (b/(a*b + Abs[z]))*z;
  Row[
    {
      ComplexPlot3D[
        PASF,
        {z, -6 - 6 I, 6 + 6 I},
        PlotRange -> All,
        AxesLabel -> {"Re(z)", "Im(z)", "APSF"},
        ColorFunction -> "CyclicLogAbsArg",
        PlotLegends -> Automatic,
        BoxRatios -> {1, 1, 1},
        ImageSize -> 200
      ],
      ComplexPlot[
        PASF,
        {z, -6 - 6 I, 6 + 6 I},
        PlotRange -> All,
        ColorFunction -> "CyclicLogAbsArg",
        PlotLegends -> Automatic,
        BoxRatios -> {1, 1, 1},
        ImageSize -> 150
      ]
    }
  ],
  {{a, 2, "a"}, 0.1, 5, 0.1},
  {{b, 1, "b"}, 0.1, 5, 0.1}
]
```

**Mathematica Code 10.41**

Input (* The code aims to visualize various components of the complex cardioid activation function, defined as $c\text{Cardioid}(z)= \frac{1}{2} (1+ \cos(\arg(z))) z$, in the complex plane where $z=x+i y$. It defines this function and uses a reusable function `createCardioidPlot` to generate 3D surface plots and slice contour plots for the real part, imaginary part, magnitude, and phase of the function. Each plot displays the specified component over a specified range, combining both 3D and contour visualizations to provide a comprehensive view. The final output is a collection of four plots, each highlighting a different component of the complex cardioid function: *)

```
(* Define a complex variable: *)
znumber=x+I*y;

(* Define the complex cardioid activation function: *)
cCardioid:=1/2 (1+Cos[Arg[znumber]])*znumber

(* Define a function to create the plots for different components of the complex
cardioid function: *)
createCardioidPlot[component_,label_]:=Module[
{plot1,slice},
(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-6,6},
  {y,-6,6},
  ClippingStyle->None,
  AxesLabel->{"Re(z)","Im(z)"},
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-6,6},
```

```

{y,-6,6},
{z,-1,1},
Contours->15,
Axes->False,
PlotPoints->50,
PlotRangePadding->0,
ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
plot1,
slice,
PlotRange->All,
PlotLabel->label,
BoxRatios->{1,1,1},
FaceGrids->{Back,Left},
ImageSize->200
]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
the complex cardioid function: *)
realCardioidPlot=createCardioidPlot[
Re[cCardioid],
"Real Part of Complex Cardioid"
];

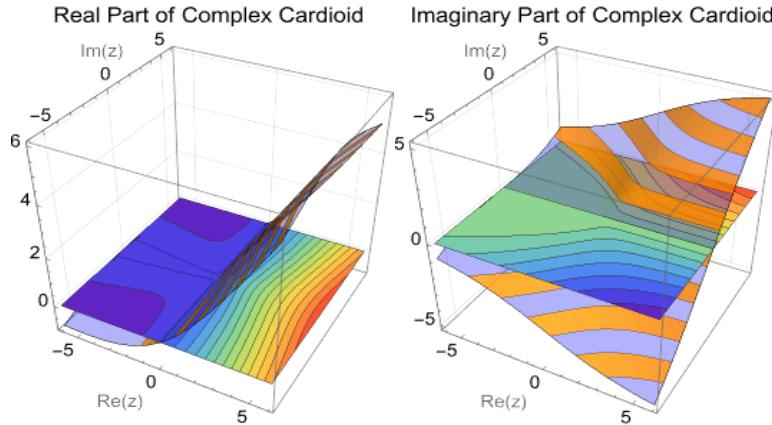
imaginaryCardioidPlot=createCardioidPlot[
Im[cCardioid],
"Imaginary Part of Complex Cardioid"
];

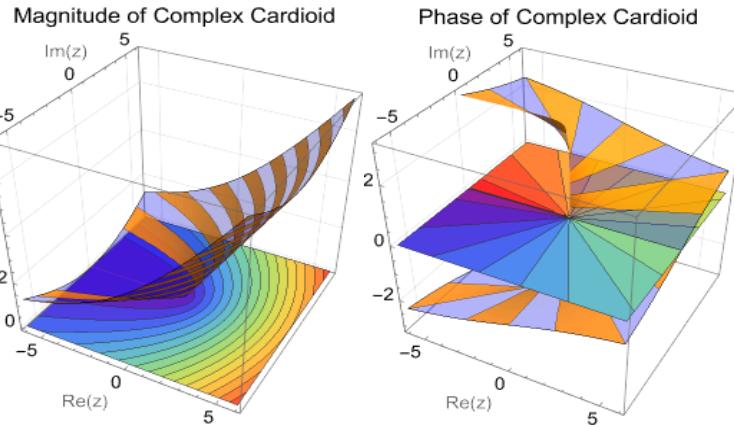
magnitudeCardioidPlot=createCardioidPlot[
Abs[cCardioid],
"Magnitude of Complex Cardioid"
];

phaseCardioidPlot=createCardioidPlot[
Arg[cCardioid],
"Phase of Complex Cardioid"
];
(* Display all four plots together in a list: *)
{realCardioidPlot,imaginaryCardioidPlot,magnitudeCardioidPlot,phaseCardioidPlot}

```

Output



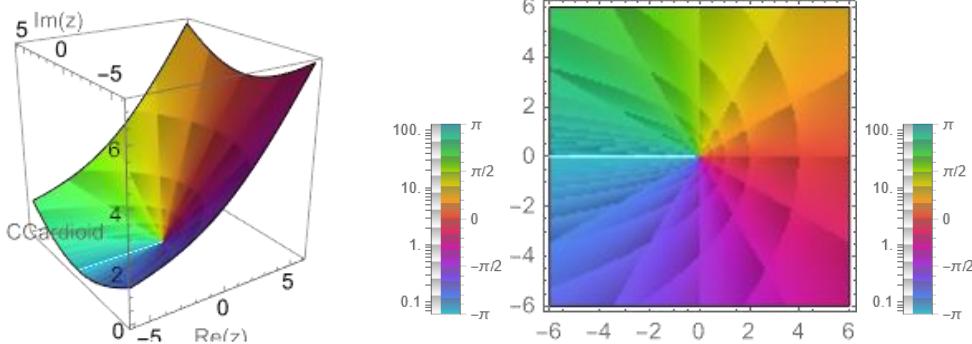
**Mathematica Code 10.42**

```

Input      (* Define the complex cardioid activation function: *)
CCardioid:=1/2 (1+Cos[Arg[z]])*z

(* Generate a 3D plot of the complex cardioid activation function over a specified
complex range: *)
ComplexPlot3D[
  CCardioid,
  {z, -6-6 I, 6+6 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)", "Im(z)", "CCardioid"},
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->200
]
(* Generate a 2D complex plot of the complex cardioid activation function over a
specified complex range: *)
ComplexPlot[
  CCardioid,
  {z, -6-6 I, 6+6 I},
  (* Plot options: *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->150
]

```

Output

Mathematica Code 10.43

```

Input      (* The code aims to visualize various components of the complex modReLU function,
           defined as a piecewise function operating on the magnitude and phase of the complex
           variable z=x+iy. It defines this function and uses a reusable function
           `createModReLUPlot` to generate 3D surface plots and slice contour plots for the
           real part, imaginary part, magnitude, and phase of the function. Each plot displays
           the specified component over a specified range, combining both 3D and contour
           visualizations to provide a comprehensive view. The final output is a collection
           of four plots, each highlighting a different component of the complex modReLU
           function: *)

(* Define a complex variable: *)
znumber=x+I*y;
b=-0.7;

(* Define the complex modReLU function: *)
modReLU:=Piecewise[
  {
    {(1+b/Abs[znumber]) znumber, Abs[znumber]+b>=0},
    {0,Abs[znumber]+b<0}
  }
]

(* Define a function to create the plots for different components of the modReLU
function: *)
createModReLUPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-2,2},
    {y,-2,2},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)","Im(z)"},
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity[.5],
    MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
    Lighting->"Neutral"
  ];
]

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-2,2},
  {y,-2,2},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,

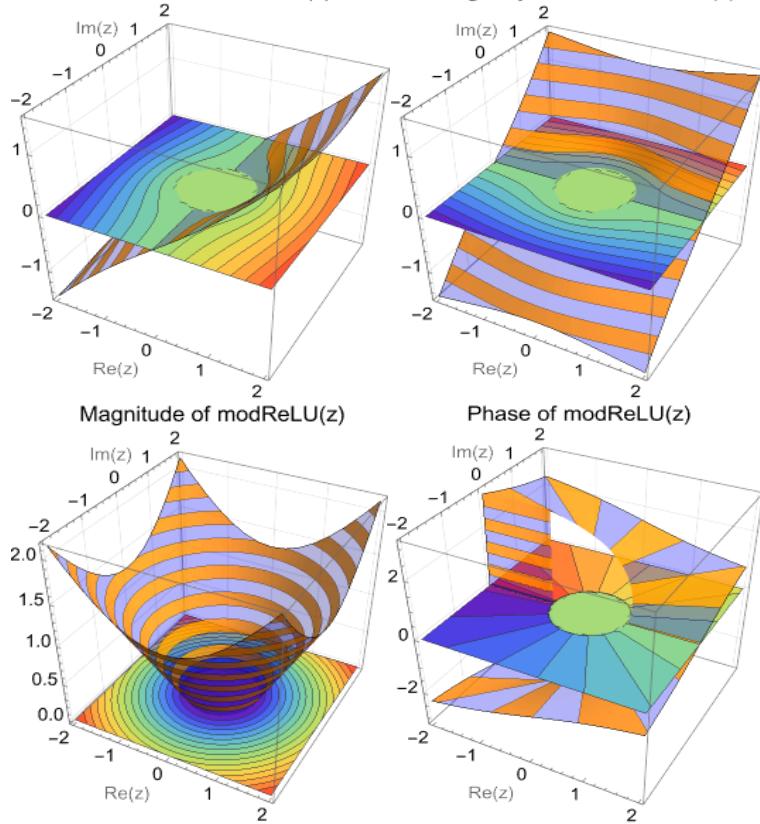
```

```

slice,
(* Plot options: *)
PlotRange->All,
PlotLabel->label,
BoxRatios->{1,1,1},
FaceGrids->{Back,Left},
ImageSize->200
]
]
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
modReLU(z): *)
realModReLUPlot=createModReLUPlot[
  Re[modReLU],
  "Real Part of modReLU(z)"
];
imaginaryModReLUPlot=createModReLUPlot[
  Im[modReLU],
  "Imaginary Part of modReLU(z)"
];
magnitudeModReLUPlot=createModReLUPlot[
  Abs[modReLU],
  "Magnitude of modReLU(z)"
];
phaseModReLUPlot=createModReLUPlot[
  Arg[modReLU],
  "Phase of modReLU(z)"
];
(* Display all four plots together in a list: *)
{realModReLUPlot,imaginaryModReLUPlot,magnitudeModReLUPlot,phaseModReLUPlot}

```

Output



Mathematica Code 10.44

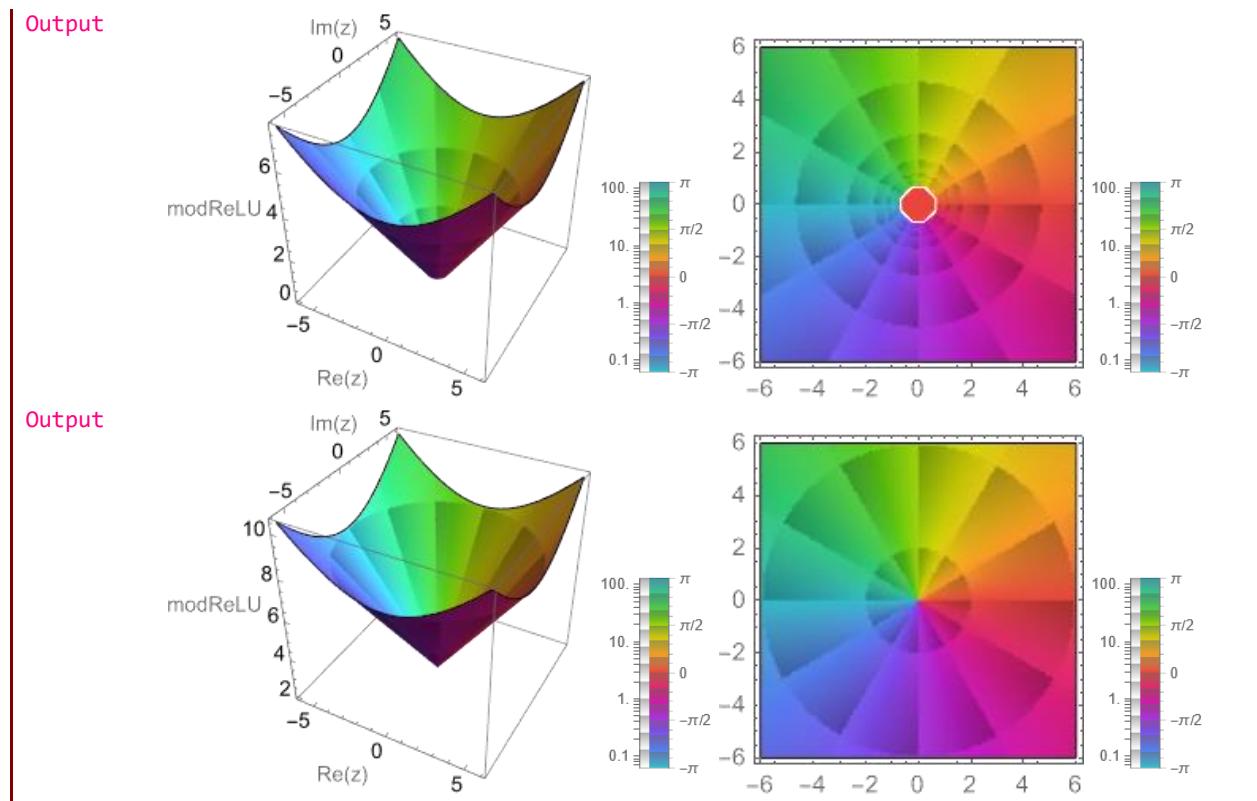
```

Input      (* The code defines a function `createComplexPlots` that generates both 3D and 2D
           complex plots of the modReLU function, which is parameterized by b. This function
           modularly creates the modReLU function and produces visualizations using
           `ComplexPlot3D` and `ComplexPlot` over a specified complex range. By calling
           `createComplexPlots` with different values of b (e.g., b=-0.7 and b=2), the code
           efficiently generates and displays the plots, allowing for a comparison of the
           function's behavior under different parameters: *)

(* Define the function to create complex plots: *)
createComplexPlots[b_]:=Module[
  {modReLU},
  (*Define the complex modReLU function*)
  modReLU:=Piecewise[
    {
      {(1+b/Abs[z]) z,Abs[z]+b>=0},
      {0,Abs[z]+b<0}
    }
  ];
  (* Generate a 3D plot of the complex modReLU function over a specified complex
range: *)
  plot3D=ComplexPlot3D[
    modReLU,
    {z,-6-6 I,6+6 I},
    (* Plot options: *)
    PlotRange->All,
    AxesLabel->{"Re(z)","Im(z)","modReLU"},
    ColorFunction->"CyclicLogAbsArg",
    PlotLegends->Automatic,
    BoxRatios->{1,1,1},
    ImageSize->200
  ];
  (* Generate a 2D complex plot of the complex modReLU function over a specified
complex range: *)
  plot2D=ComplexPlot[
    modReLU,
    {z,-6-6 I,6+6 I},
    (* Plot options: *)
    PlotRange->All,
    ColorFunction->"CyclicLogAbsArg",
    PlotLegends->Automatic,
    BoxRatios->{1,1,1},
    ImageSize->150
  ];
  {plot3D,plot2D}
]

(* Generate plots for b=-0.7: *)
plotsNeg07=createComplexPlots[-0.7];
(* Generate plots for b=2: *)
plots2=createComplexPlots[2];
(* Display the plots: *)
{plotsNeg07,plots2}

```

**Mathematica Code 10.45**

Input (* The code defines a function `createComplexPlots` that generates both 3D and 2D complex plots of the modReLU function, which is parameterized by b. It uses `Manipulate` to create an interactive interface allowing users to dynamically adjust the parameter b and observe the changes in the function's behavior in real-time. The function `createComplexPlots` modularly defines the modReLU function, generates visualizations using `ComplexPlot3D` and `ComplexPlot`, and returns both plots: *)

```
(* Define a function to create complex plots: *)
createComplexPlots[b_]:=Module[
{modReLU,plot3D,plot2D},

(* Define the complex modReLU function: *)
modReLU:=Piecewise[
{
{(1+b/Abs[z]) z,Abs[z]+b>=0},
{0,Abs[z]+b<0}
}];

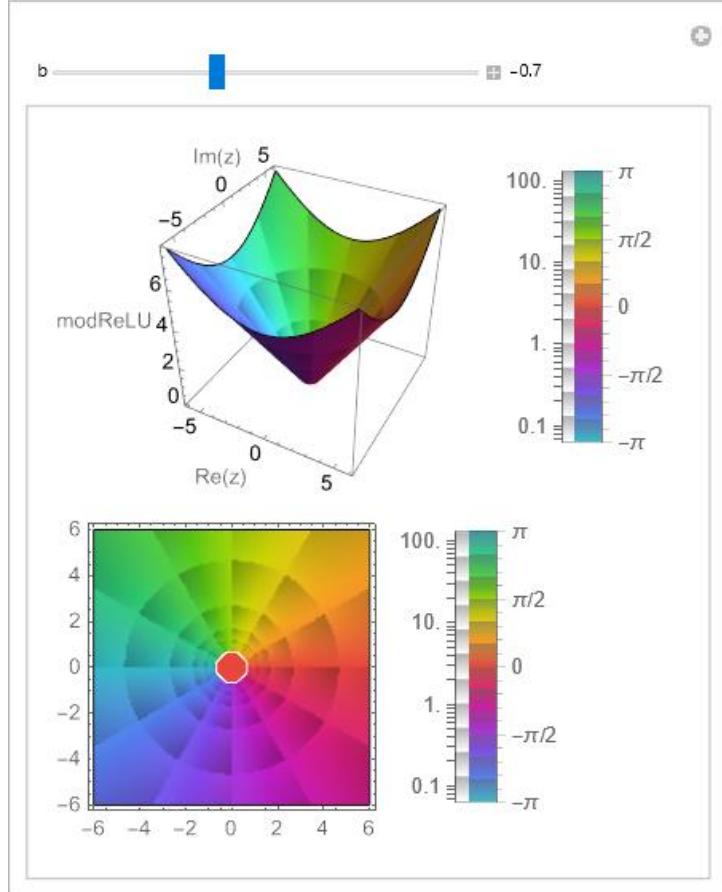
(* Generate a 3D plot of the complex modReLU function over a specified complex range: *)
plot3D=ComplexPlot3D[
  modReLU,
  {z,-6-6 I,6+6 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","modReLU"},
  ColorFunction->"CyclicLogAbsArg",
```

```

    PlotLegends->Automatic,
    BoxRatios->{1,1,1},
    ImageSize->200
];
(* Generate a 2D complex plot of the complex modReLU function over a specified
complex range: *)
plot2D=ComplexPlot[
  modReLU,
  {z,-6-6 I,6+6 I},
  (* Plot options: *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbsArg",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->150
];
{plot3D,plot2D}
]
(* Create an interactive Manipulate environment to adjust the parameter b: *)
Manipulate[
 Module[
 {plots},
 plots=createComplexPlots[b];
 Column[plots]],
 {{b,-0.7},-3,3,0.1,Appearance->"Labeled"},
 ControlPlacement->Top
]

```

Output



Mathematica Code 10.46

```

Input      (* The code aims to visualize various components of the complex hyperbolic tangent
          function tanh(z), where z=x+iy, in the complex plane. It defines this function and
          uses a reusable function `createTanhPlot` to generate 3D surface plots and slice
          contour plots for the real part, imaginary part, magnitude, and phase of the
          function. Each plot displays the specified component over a specified range,
          combining both 3D and contour visualizations to provide a comprehensive view. The
          final output is a collection of four plots, each highlighting a different component
          of the complex hyperbolic tangent function: *)

(* Define a complex variable: *)
z=x+I*y;

(* Define the complex tanh function: *)
tanhz=Tanh[z];

(* Define a function to create the plots for different components of the tanh
function: *)
createTanhPlot[component_,label_]:=Module[
{plot1,slice},
(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-3,3},
  {y,-3,3},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)","Im(z)"},
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity [.5],
  MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,
  z==0,(* Define the slicing plane at z=0 *)
  {x,-3,3},
  {y,-3,3},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]

```

```

        ]
]

(* Generate the plots for the real part, imaginary part, magnitude, and phase of
tanh(z): *)
realTanhPlot=createTanhPlot[
  Re[tanhz],
  "Real Part of Tanh(z)"
];

imagineTanhPlot=createTanhPlot[
  Im[tanhz],
  "Imaginary Part of Tanh(z)"
];

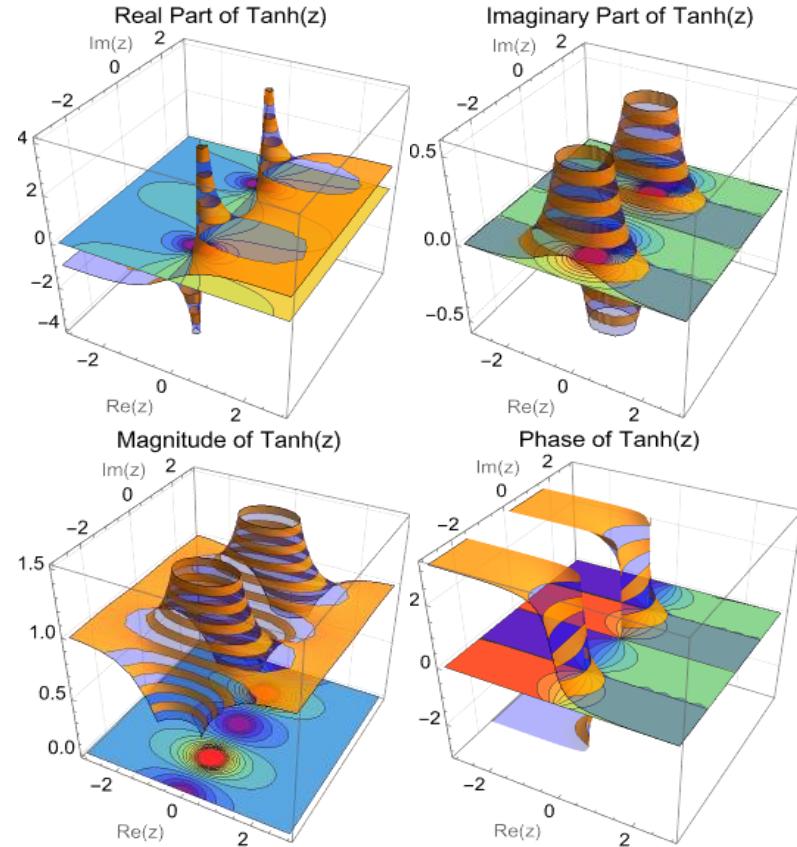
magnitudeTanhPlot=createTanhPlot[
  Abs[tanhz],
  "Magnitude of Tanh(z)"
];

phaseTanhPlot=createTanhPlot[
  Arg[tanhz],
  "Phase of Tanh(z)"
];

(* Display all four plots together in a list: *)
{realTanhPlot,imagineTanhPlot,magnitudeTanhPlot,phaseTanhPlot}

```

Output



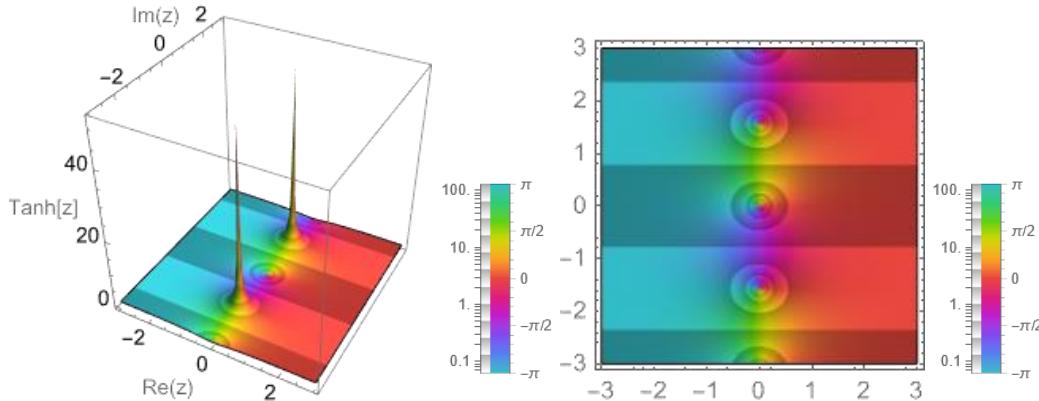
Mathematica Code 10.47

```

Input      (* Generate a 3D plot of the complex hyperbolic tangent function Tanh(z) over a
           specified complex range: *)
ComplexPlot3D[
  Tanh[z],
  {z,-3-3 I,3+3 I},
  (* Plot options: *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","Tanh[z]"},
  ColorFunction->"CyclicLogAbs",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->200
]

(* Generate a 2D complex plot of the complex hyperbolic tangent function Tanh(z)
over a specified complex range: *)
ComplexPlot[
  Tanh[z],
  {z,-3-3 I,3+3 I},
  (* Plot options: *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbs",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->150
]

```

Output**Mathematica Code 10.48**

```

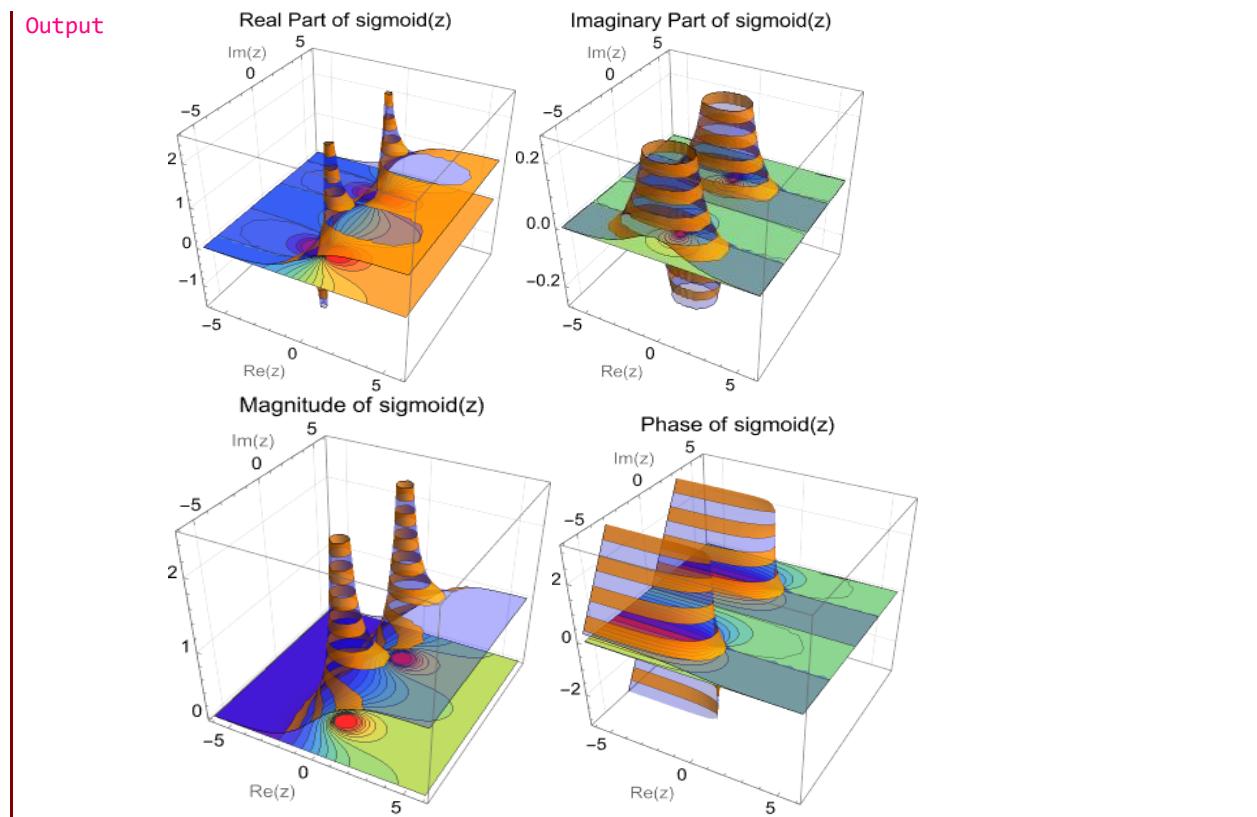
Input      (* The code aims to visualize various components of the complex sigmoid function
           sigma(z)= 1/(1+Exp[-z]) in the complex plane, where z=x+i y. It defines this
           function and uses a reusable function `createSigmoidPlot` to generate 3D surface
           plots and slice contour plots for the real part, imaginary part, magnitude, and
           phase of the function. Each plot displays the specified component over a specified
           range, combining both 3D and contour visualizations to provide a comprehensive
           view. The final output is a collection of four plots, each highlighting a different
           component of sigma(z): *)
(* Define a complex variable: *)
znumber=x+I*y;
(* Define the complex sigmoid function: *)
sigmoid:=1/(1+Exp[-znumber])
(* Define a function to create the plots for different components of the sigmoid
function: *)
createSigmoidPlot[component_,label_]:=Module[
  {plot1,slice},

```

```

(* Generate the 3D plot of the specified component: *)
plot1=Plot3D[
  component,
  {x,-6,6},
  {y,-6,6},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)", "Im(z)" },
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
];
(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  component,z==0,(* Define the slicing plane *)
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];
(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot1,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
sigmoid(z): *)
realSigmoidPlot=createSigmoidPlot[
  Re[sigmoid],
  "Real Part of sigmoid(z)"
];
imaginarySigmoidPlot=createSigmoidPlot[
  Im[sigmoid],
  "Imaginary Part of sigmoid(z)"
];
magnitudeSigmoidPlot=createSigmoidPlot[
  Abs[sigmoid],
  "Magnitude of sigmoid(z)"
];
phaseSigmoidPlot=createSigmoidPlot[
  Arg[sigmoid],
  "Phase of sigmoid(z)"
];
(* Display all four plots together in a list: *)
{realSigmoidPlot,imaginarySigmoidPlot,magnitudeSigmoidPlot,phaseSigmoidPlot}

```

**Mathematica Code 10.49**

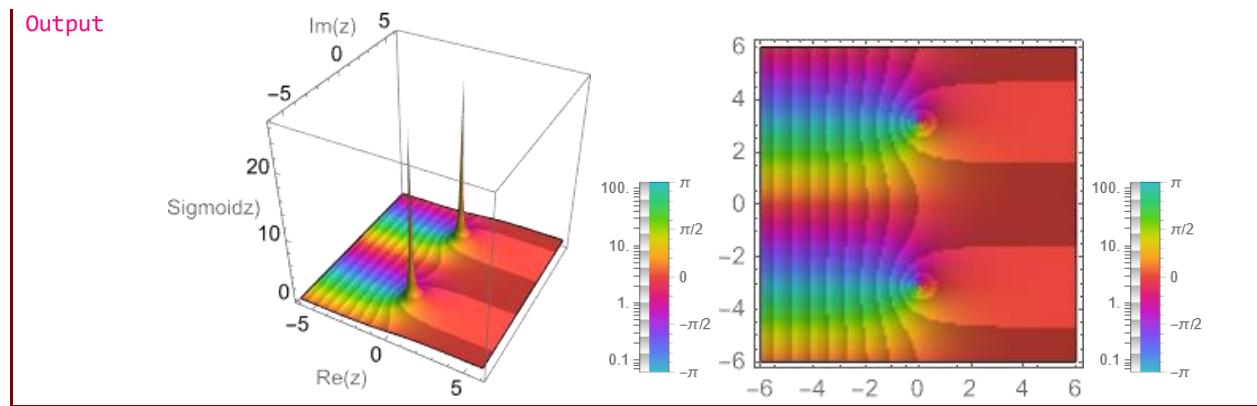
```

Input      (* Define the complex sigmoid function: *)
sigmoid[z_]:=1/(1+Exp[-z]);

(* Generate a 3D plot of the complex sigmoid function over a specified complex
range*)
ComplexPlot3D[
 sigmoid[z],
 {z,-6-6 I,6+6 I},
 (* Plot options: *)
 PlotRange->All,
 AxesLabel->{"Re(z)","Im(z)","Sigmoidz")},
 ColorFunction->"CyclicLogAbs",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->200
]

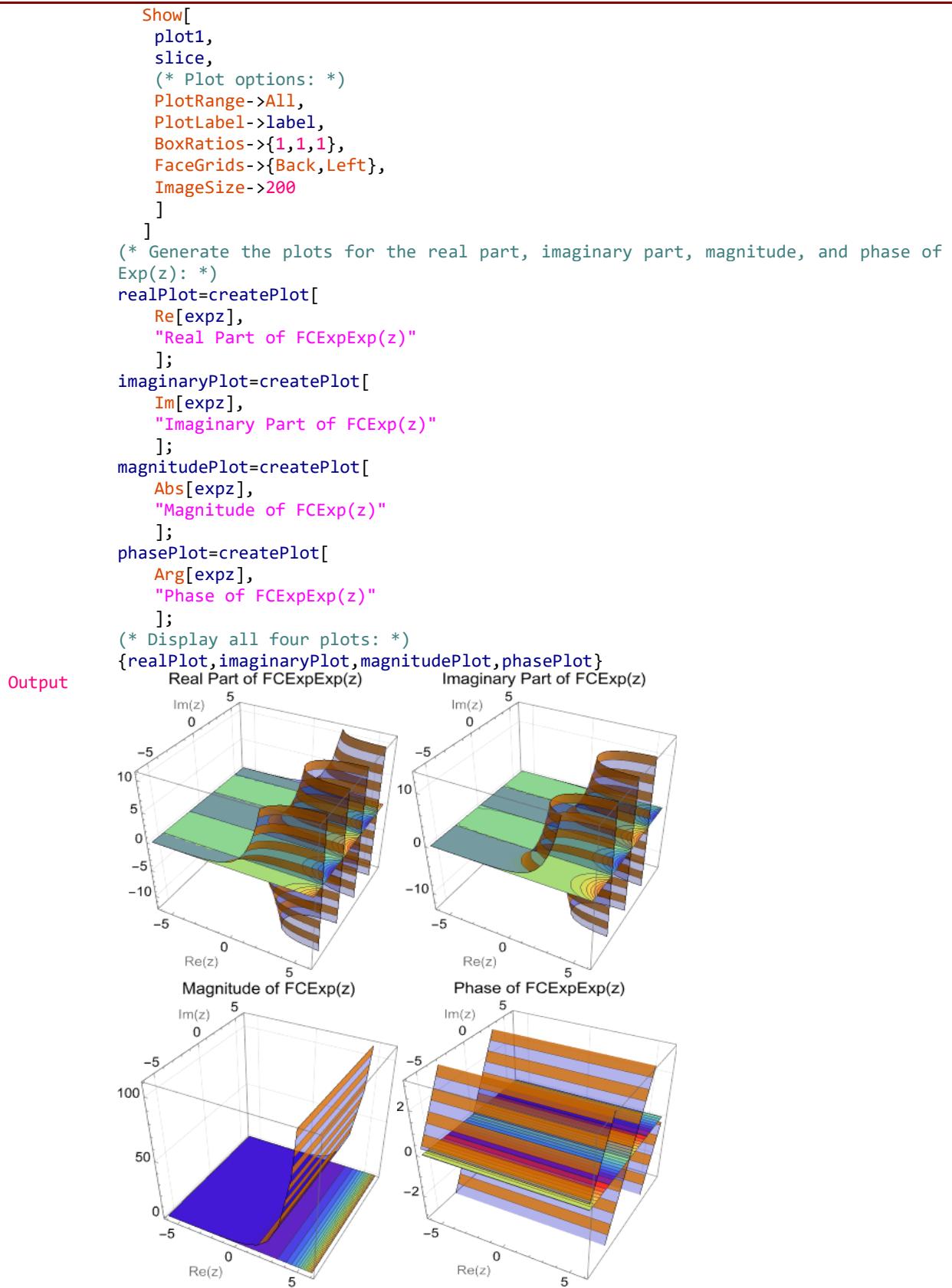
(* Generate a 2D complex plot of the complex sigmoid function over a specified
complex range: *)
ComplexPlot[
 sigmoid[z],
 {z,-6-6 I,6+6 I},
 (* Plot options *)
 PlotRange->All,
 ColorFunction->"CyclicLogAbs",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->150
]

```

**Mathematica Code 10.50**

Input (* The code aims to visualize different aspects of the complex exponential function $\exp(z)$ in the complex plane, where $z=x+i y$. It defines the function and then uses a reusable function `createPlot` to generate 3D surface plots and slice contour plots for the real part, imaginary part, magnitude, and phase of $\exp(z)$. Each plot displays the specified component over a specified range, combining both 3D and contour visualizations to provide a comprehensive view. The final output is a collection of four plots, each highlighting a different component of $\exp(z)$: *)

```
(* Define a complex variable: *)
z=x+I*y;
(* Define the complex exponential function: *)
expz=Exp[z];
(* Define a function to create the plots: *)
createPlot[component_,label_]:=Module[
  {plot1,slice},
  (* Generate the 3D plot of the specified component: *)
  plot1=Plot3D[
    component,
    {x,-6,6},
    {y,-6,6},
    (* Plot options: *)
    ClippingStyle->None,
    AxesLabel->{"Re(z)","Im(z)"},
    MeshFunctions->{#3&},
    Mesh->15,
    MeshStyle->Opacity [.5],
    MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
    Lighting->"Neutral"
  ];
  (* Generate the slice contour plot of the specified component: *)
  slice=SliceContourPlot3D[
    component,
    z==0,(* Define the slicing plane*)
    {x,-6,6},
    {y,-6,6},
    {z,-1,1},
    (* Plot options: *)
    Contours->15,
    Axes->False,
    PlotPoints->50,
    PlotRangePadding->0,
    ColorFunction->"Rainbow"
  ];
  (* Combine the 3D plot and the slice contour plot: *)
```



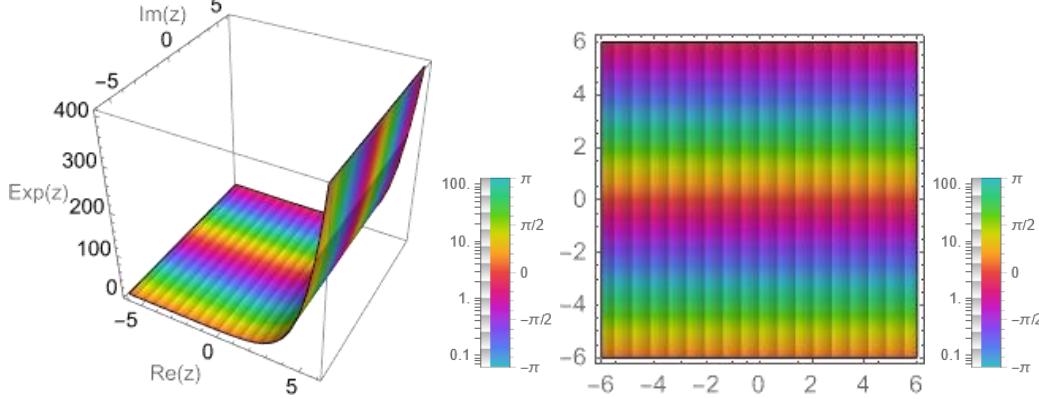
Mathematica Code 10.51

```

Input      (* Generate a 3D plot of the complex exponential function Exp(z) over a specified
           complex range*)
ComplexPlot3D[
  (* The function to be plotted: complex exponential function: *)
  Exp[z],
  {z,-6-6 I,6+6 I},
  (* Plot options *)
  PlotRange->All,
  AxesLabel->{"Re(z)","Im(z)","Exp(z)"},
  ColorFunction->"CyclicLogAbs",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->200
]

(* Generate a 2D complex plot of the complex exponential function Exp(z) over a
specified complex range: *)
ComplexPlot[
  (* The function to be plotted: complex exponential function: *)
  Exp[z],
  {z,-6-6 I,6+6 I},
  (* Plot options *)
  PlotRange->All,
  ColorFunction->"CyclicLogAbs",
  PlotLegends->Automatic,
  BoxRatios->{1,1,1},
  ImageSize->150
]

```

Output**Mathematica Code 10.52**

```

Input      (* The code defines and visualizes the `zReLU` activation function for a complex
           variable z=x+iy, which returns z if both the real and imaginary parts are positive,
           and 0 otherwise. It introduces a reusable function `createPlot` to generate and
           display 3D surface plots and slice contour plots for the real part, imaginary part,
           magnitude, and phase of the `zReLU` function. The generated plots are displayed
           together, allowing for a comprehensive examination of the `zReLU` function's
           behavior across different components in the complex plane: *)

(* Define a complex variable: *)
znumber=x+I*y;

(* Define the zReLU function: *)
zReLU:=Piecewise[


```

```

{
  {znumber,Re[znumber]>0&&Im[znumber]>0},
  {0,True}
}
]

(* Function to create and display plots: *)
createPlot[part_,label_]:=Module[
{plot,slice},

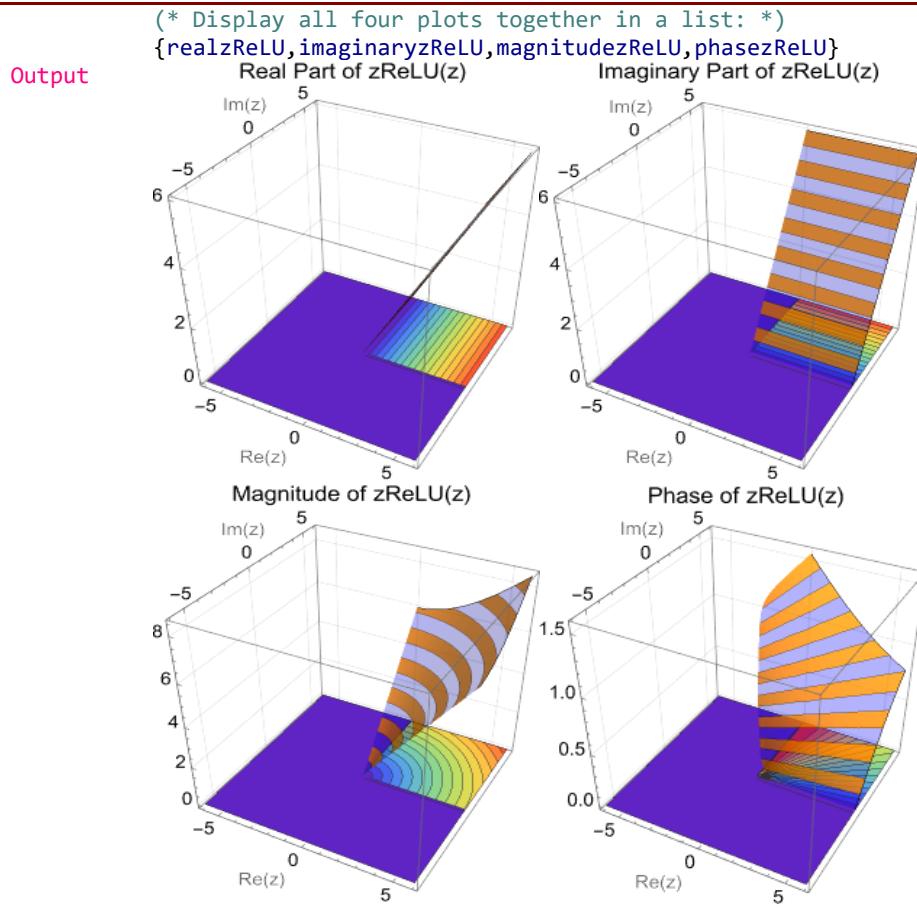
(* Generate the 3D plot of the specified component: *)
plot=Plot3D[
  part[zReLU],
  {x,-6,6},
  {y,-6,6},
  (* Plot options: *)
  ClippingStyle->None,
  AxesLabel->{"Re(z)","Im(z)"},
  MeshFunctions->{#3&},
  Mesh->15,
  MeshStyle->Opacity[.5],
  MeshShading->{{Opacity[.3],Blue},{Opacity[.8],Orange}},
  Lighting->"Neutral"
];

(* Generate the slice contour plot of the specified component: *)
slice=SliceContourPlot3D[
  part[zReLU],
  z==0,
  {x,-6,6},
  {y,-6,6},
  {z,-1,1},
  (* Plot options: *)
  Contours->15,
  Axes->False,
  PlotPoints->50,
  PlotRangePadding->0,
  ColorFunction->"Rainbow"
];

(* Combine the 3D plot and the slice contour plot: *)
Show[
  plot,
  slice,
  (* Plot options: *)
  PlotRange->All,
  PlotLabel->label,
  BoxRatios->{1,1,1},
  FaceGrids->{Back,Left},
  ImageSize->200
]
]

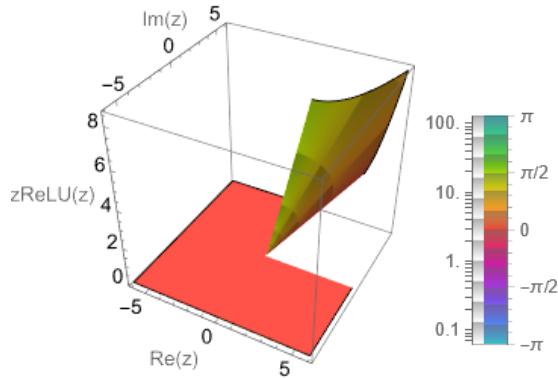
(* Generate the plots for the real part, imaginary part, magnitude, and phase of
zReLU: *)
realzReLU=createPlot[Re,"Real Part of zReLU(z)"];
imaginaryzReLU=createPlot[Im,"Imaginary Part of zReLU(z)"];
magnitudezReLU=createPlot[Abs,"Magnitude of zReLU(z)"];
phasezReLU=createPlot[Arg,"Phase of zReLU(z)"];

```

**Mathematica Code 10.53**

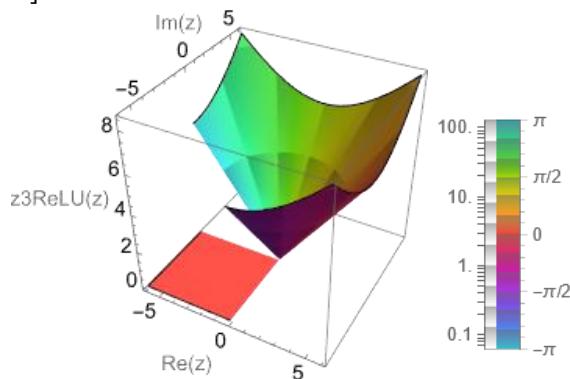
```
Input      (* Define the zReLU function: *)
zReLU:=Piecewise[
 {
 {z,Re[z]>0&&Im[z]>0},
 {0,True}
 ]
(* Generate a 3D plot of the zReLU function over a specified complex range: *)
ComplexPlot3D[
zReLU,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)", "Im(z)", "zReLU(z)" },
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
]
```

Output

**Mathematica Code 10.54**

```
Input (* Define the z3ReLU function: *)
z3ReLU:=Piecewise[
 {
 {z,Re[z]>0||Im[z]>0},
 {0,True}
 ]
(* Generate a 3D plot of the z3ReLU function over a specified complex range: *)
ComplexPlot3D[
 z3ReLU,
 {z,-6-6 I,6+6 I},
 (* Plot options: *)
 PlotRange->All,
 AxesLabel->{"Re(z)", "Im(z)", "z3ReLU(z)" },
 ColorFunction->"CyclicLogAbsArg",
 PlotLegends->Automatic,
 BoxRatios->{1,1,1},
 ImageSize->200
 ]
```

Output

**Mathematica Code 10.55**

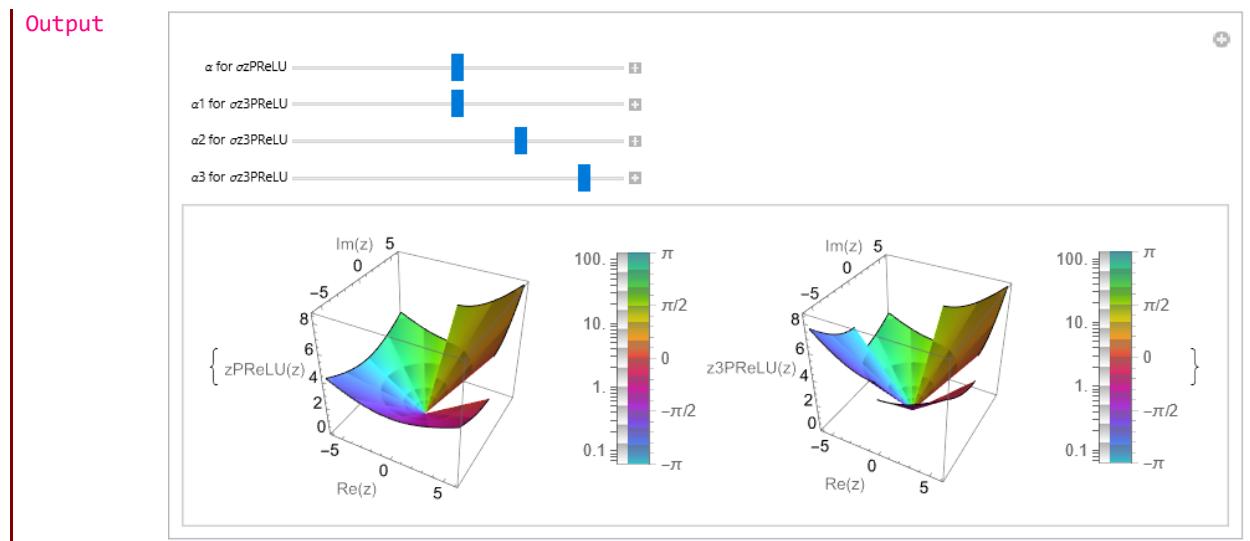
```
Input (* The code creates an interactive environment using the `Manipulate` function,
allowing users to dynamically adjust the parameters α, α1, α2, and α3 for two
complex-valued activation functions defined using piecewise conditions. The first
function, αzPReLU, changes its value based on whether both the real and imaginary
parts of the complex variable z are positive, while the second function, αz3PReLU,
considers different cases for combinations of positive and negative real and
imaginary parts. The code generates 3D plots for both functions, displaying their
behavior over a specified range of complex values. By manipulating the sliders,
users can observe how the surfaces of these functions change, enhancing their
```

```

understanding of the impact of parameter variations on complex activation
functions: *)
```

```

Manipulate[
{
(* Define the activation functions using Piecewise: *)
(* Define the σzPReLU activation function: *)
σzPReLU:=Piecewise[
{
{z,Re[z]>0&&Im[z]>0},
{α z,True}
];
(* Define the σz3PReLU activation function: *)
σz3PReLU:=Piecewise[
{
{z,Re[z]>0&&Im[z]>0},
{α1 z,Re[z]<=0&&Im[z]>0},
{α2 z,Re[z]>0&&Im[z]<=0},
{α3 z,True}
};
(* Generate a 3D plot for the σzPReLU activation function: *)
plot1=ComplexPlot3D[
σzPReLU,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)", "Im(z)", "zPReLU(z)" },
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
];
(* Generate a 3D plot for the σz3PReLU activation function: *)
plot2=ComplexPlot3D[
σz3PReLU,
{z,-6-6 I,6+6 I},
(* Plot options: *)
PlotRange->All,
AxesLabel->{"Re(z)", "Im(z)", "z3PReLU(z)" },
ColorFunction->"CyclicLogAbsArg",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
];
(* Display the plots side by side: *)
Row[{plot1,plot2}]
},
(* Manipulate parameters with sliders: *)
{{α,0.5,"α for σzPReLU"},0,1,0.1},
{{α1,0.5,"α1 for σz3PReLU"},0,1,0.1},
{{α2,0.7,"α2 for σz3PReLU"},0,1,0.1},
{{α3,0.9,"α3 for σz3PReLU"},0,1,0.1}
]
]
```

**Mathematica Code 10.56**

Input (* The code defines a function `generatePlots` to create combined 3D and slice contour plots of the magnitudes of various complex functions. The function takes a mathematical function and its label as inputs, generating 3D plots with specific visual styles and labels for the real and imaginary parts of the complex variable. It also creates slice contour plots at $z=0$ to show the function's behavior in that plane. The combined plots are displayed using the `Show` function. The code then defines a list of functions (`Tan`, `Tanh`, `ArcTan`, `ArcTanh`, `Sin`, `Sinh`, `ArcSin`, `ArcCos`, `ArcSinh`) with their labels, and iterates over this list to generate and display the corresponding plots for each function: *)

```
(* Define a function to generate the plots: *)
generatePlots[func_,label_]:=Module[
{z,plot1,slice},
z=x+I*y;

(* Generate the 3D plot of the magnitude of the function: *)
plot1=Plot3D[
Abs[func[z]],
{x,-3,3},
{y,-3,3},
(* Plot options: *)
AxesLabel->{"Re(z)","Im(z)"},
ClippingStyle->None,
MeshFunctions->{#3&},
Mesh->15,
MeshStyle->Opacity [.5],
MeshShading->{{Opacity [.3],Blue},{Opacity [.8],Orange}},
Lighting->"Neutral"
];

(* Generate the slice contour plot of the magnitude of the function: *)
slice=SliceContourPlot3D[
Abs[func[z]],
zz==0,
{x,-3,3},
{y,-3,3},
{zz,-1,1},
(* Plot options: *)
Contours->15,
```

```

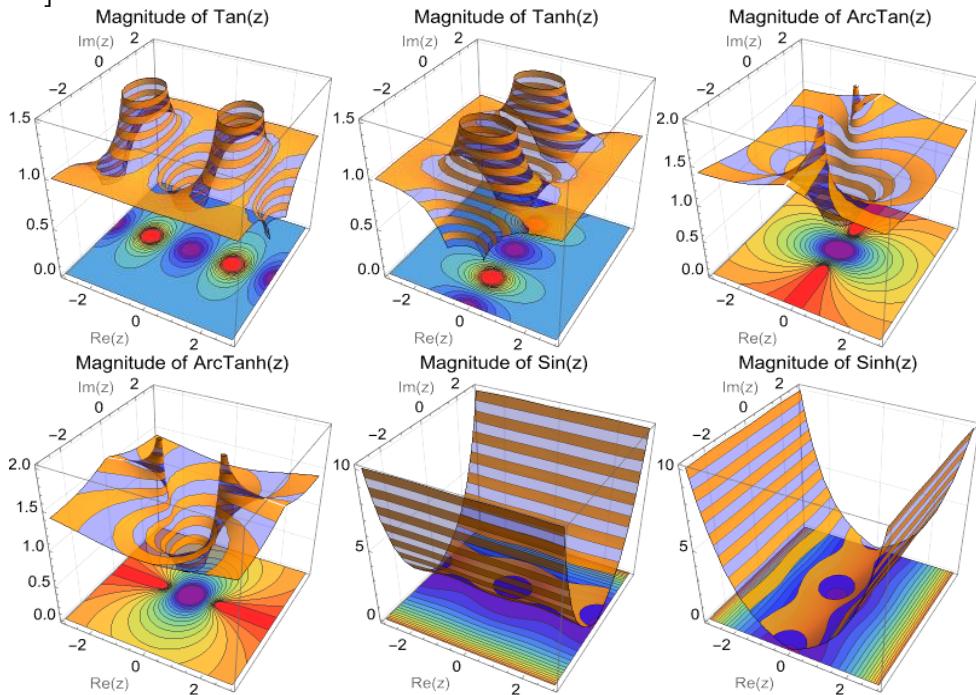
Axes->False,
PlotPoints->50,
PlotRangePadding->0,
ColorFunction->"Rainbow"
];

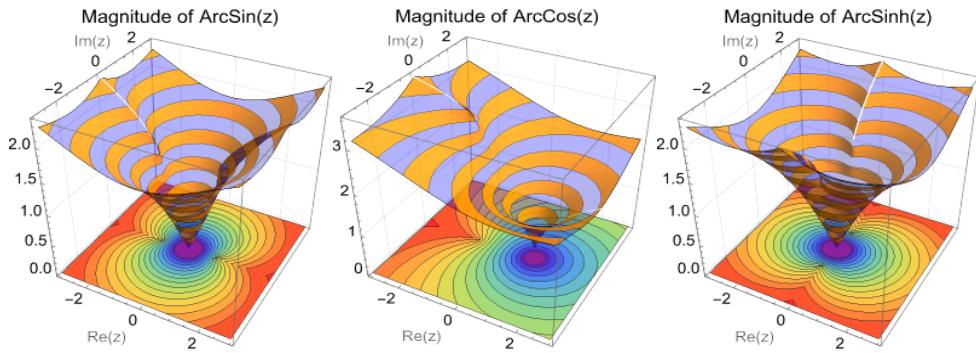
(* Combine the 3D plot and the slice contour plot: *)
Show[
plot1,
slice,
(* Plot options: *)
PlotRange->All,
PlotLabel->"Magnitude of "<>label,
BoxRatios->{1,1,1},
FaceGrids->{Back,Left},
ImageSize->200
]
]
(* Define the list of functions and their labels: *)
functions={
{Tan,"Tan(z)"}, {Tanh,"Tanh(z)"}, {ArcTan,"ArcTan(z)"}, {ArcTanh,"ArcTanh(z)"}, {Sin,"Sin(z)"}, {Sinh,"Sinh(z)"}, {ArcSin,"ArcSin(z)"}, {ArcCos,"ArcCos(z)"}, {ArcSinh,"ArcSinh(z)"}
};

(* Generate the plots using the defined function: *)
plots=Table[
generatePlots[func[[1]],func[[2]]],
{func,functions}
]

```

Output



**Mathematica Code 10.57**

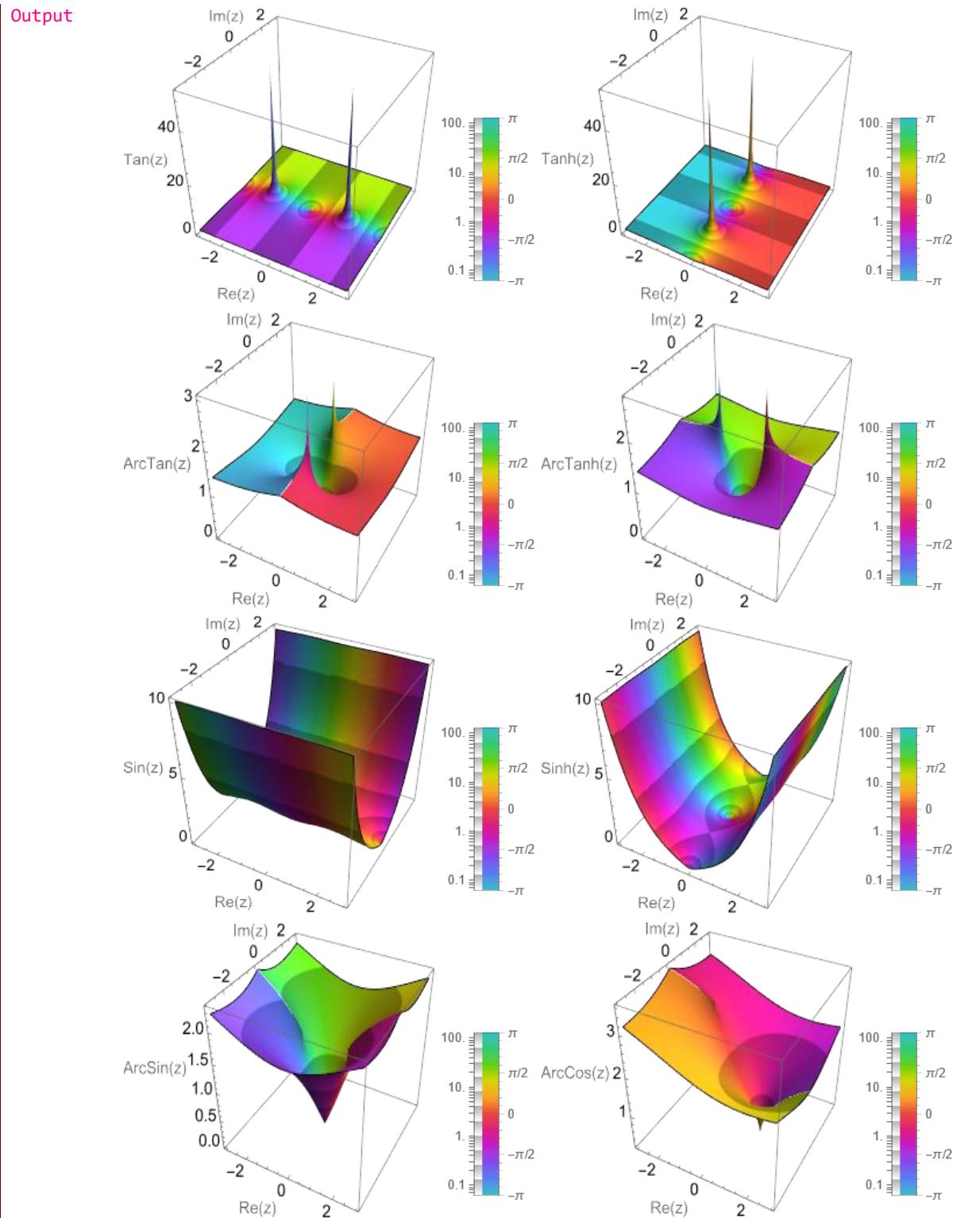
```

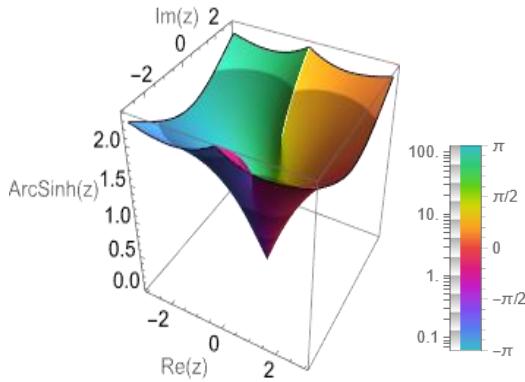
Input (* The code defines a list of complex functions along with their labels and uses
       Mathematica's `ComplexPlot3D` to generate 3D visualizations for each function over
       a specified range in the complex plane. By iterating over the list of functions
       with a `Table` construct, it produces a series of 3D plots that illustrate the real
       and imaginary components of the functions, facilitating better understanding and
       analysis of their behaviors: *)

(* Define the list of functions and their labels: *)
functions={
  {Tan[z], "Tan(z)"}, 
  {Tanh[z], "Tanh(z)"}, 
  {ArcTan[z], "ArcTan(z)"}, 
  {ArcTanh[z], "ArcTanh(z)"}, 
  {Sin[z], "Sin(z)"}, 
  {Sinh[z], "Sinh(z)"}, 
  {ArcSin[z], "ArcSin(z)"}, 
  {ArcCos[z], "ArcCos(z)"}, 
  {ArcSinh[z], "ArcSinh(z)"}
};

(* Generate ComplexPlot3D for each function: *)
plots=Table[
  ComplexPlot3D[
    func[[1]],
    {z,-3-3 I,3+3 I},
    (* Plot options: *)
    PlotRange->All,
    AxesLabel->{"Re(z)", "Im(z"),func[[2]]},
    ColorFunction->"CyclicLogAbs",
    PlotLegends->Automatic,
    BoxRatios->{1,1,1},
    ImageSize->200
  ],
  {func,functions}
]

```



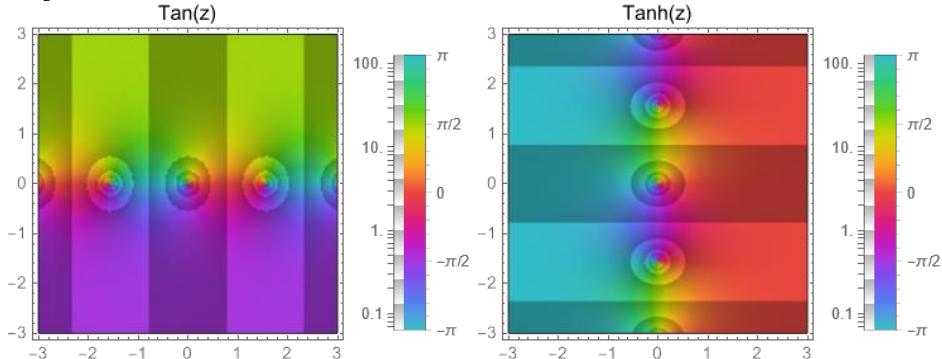
**Mathematica Code 10.58**

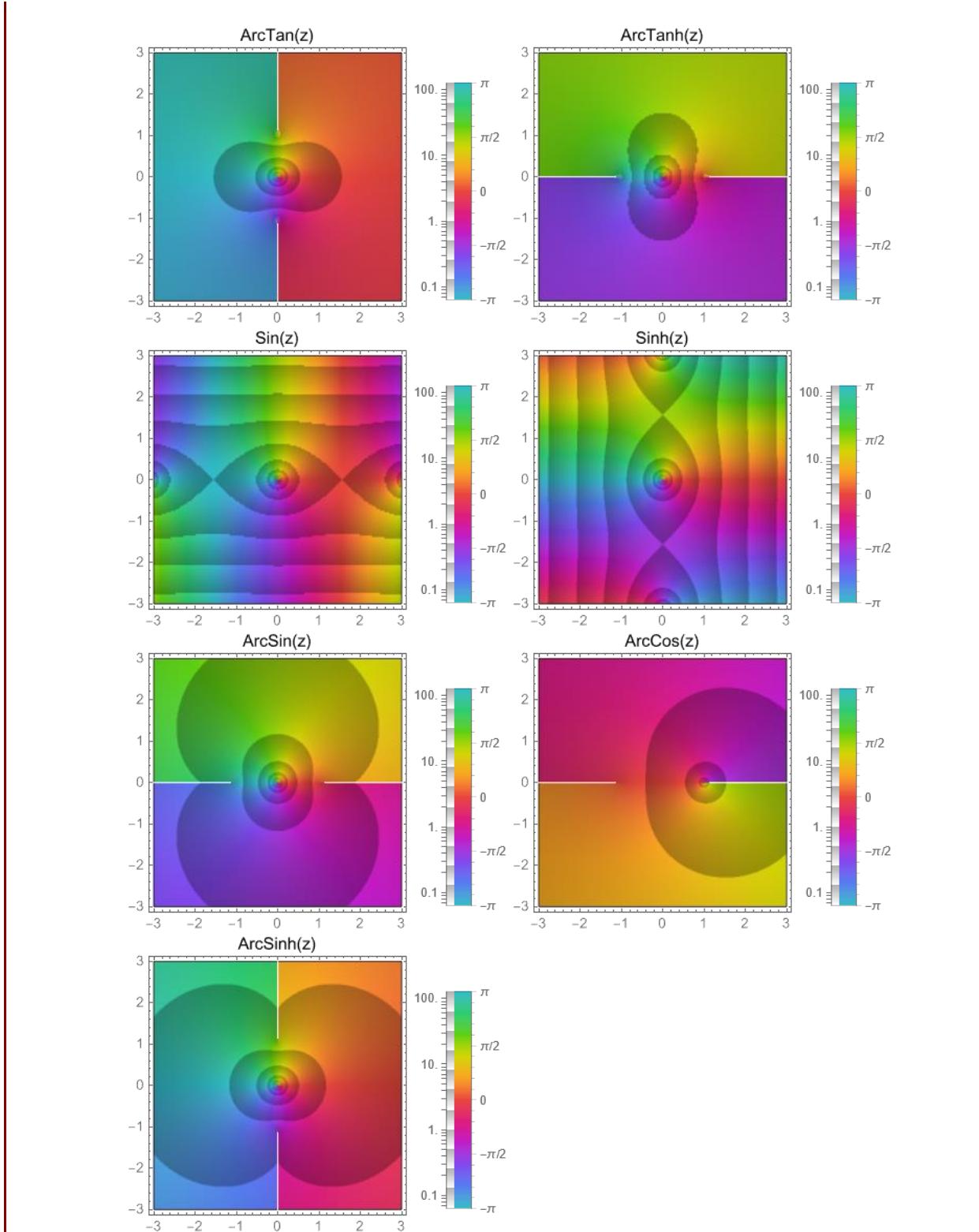
Input

```
(* The code defines a list of complex functions along with their labels and uses
Mathematica's `ComplexPlot` to generate 2D visualizations for each function over a
specified range in the complex plane. By iterating over the list of functions with
a `Table` construct, it produces a series of 2D Complex plots, facilitating better
understanding and analysis of their behaviors: *)
(* Define the list of functions and their labels: *)
functions={

{Tan[z],"Tan(z)"}, 
{Tanh[z],"Tanh(z)"}, 
{ArcTan[z],"ArcTan(z)"}, 
{ArcTanh[z],"ArcTanh(z)"}, 
{Sin[z],"Sin(z)"}, 
{Sinh[z],"Sinh(z)"}, 
{ArcSin[z],"ArcSin(z)"}, 
{ArcCos[z],"ArcCos(z)"}, 
{ArcSinh[z],"ArcSinh(z)"}
};

(* Generate a 2D complex plot for each function: *)
plots=Table[
ComplexPlot[
func[[1]],
{z,-3-3 I,3+3 I},
(* Plot options: *)
PlotRange->All,
PlotLabel->func[[2]],
ColorFunction->"CyclicLogAbs",
PlotLegends->Automatic,
BoxRatios->{1,1,1},
ImageSize->200
],
{func,functions}
]
```

Output



References

- [1] M. M. Hammad, "Statistics for Machine Learning with Mathematica Applications", *in arXiv*, 2023. doi: 10.48550/arXiv.2310.00004. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.00004>
- [2] M. M. Hammad, "Artificial Neural Network and Deep Learning: Fundamentals and Theory", 2024. doi: 10.13140/RG.2.2.18262.66889. [Online]. Available: <https://doi.org/10.13140/RG.2.2.18262.66889>
- [3] W. Mendenhall, B. M. Beaver, and R. J. Beaver, "Introduction to Probability and Statistics", 15th ed., New Delhi: Cengage Learning, 2019.
- [4] S. C. Gupta and V. K. Kapoor, "Fundamentals of Mathematical Statistics", Sultan Chand & Sons, 2000.
- [5] M. R. Spiegel, "Schaum's Outline of Theory and Problems of Statistics", 2nd ed., McGraw-Hill, 1989.
- [6] R. Hogg, J. McKean, and A. Craig, "Introduction to Mathematical Statistics", 7th ed., Pearson, 2012.
- [7] D. C. Montgomery and G. C. Runger, "Applied Statistics and Probability for Engineers", 7th ed., Wiley, 2020.
- [8] S. M. Ross, "Introduction to Probability and Statistics for Engineers and Scientists", 6th ed., Academic Press, 2020.
- [9] K. J. Hastings, "Introduction to Probability with Mathematica", 2nd ed., Chapman and Hall/CRC, 2009.
- [10] H. Ruskeepaa, "Mathematica Navigator: Mathematics, Statistics and Graphics", 3rd ed., Academic Press, 2009.
- [11] M. M. Hammad and M. M. Yahia, "Mathematics for Machine Learning and Data Science: Optimization with Mathematica Applications", *in arXiv*, 2023. doi: 10.48550/arXiv.2302.05964. [Online]. Available: <https://doi.org/10.48550/arXiv.2302.05964>
- [12] C. P. Simon and L. Blume, "Mathematics for Economists", New York: W. W. Norton & Co., 1994.
- [13] J. Lu, "Numerical Matrix Decomposition", *in arXiv*, 2021. doi: 10.48550/arXiv.2107.02579. [Online]. Available: <https://doi.org/10.48550/arXiv.2107.02579>
- [14] T. Parr and J. Howard, "The Matrix Calculus You Need for Deep Learning", *in arXiv*, 2018. doi: 10.48550/arXiv.1802.01528. [Online]. Available: <https://doi.org/10.48550/arXiv.1802.01528>
- [15] C. C. Aggarwal, "Linear Algebra and Optimization for Machine Learning: A Textbook", Springer, 2020.
- [16] E. K. P. Chong and S. H. Zak, "An Introduction to Optimization", 4th ed., Hoboken, New Jersey: Wiley, 2013.
- [17] M. A. Bhatti, "Practical Optimization Methods: With Mathematica® Applications", New York: Springer Verlag, 2013.
- [18] A. Gilat and V. Subramaniam, "Numerical Methods for Engineers and Scientists: An Introduction with Applications Using MATLAB", 3rd ed., Wiley, 2013.
- [19] S. T. Tan, "Multivariable Calculus", Cengage Learning, 2009.
- [20] D. G. Luenberger and Y. Ye, "Linear and Nonlinear Programming", New York: Springer, 2008.
- [21] S. Nayak, "Fundamentals of Optimization Techniques with Algorithms", Academic Press, 2020.
- [22] K. Deb, "Optimization for Engineering Design: Algorithms and Examples", New Delhi: Prentice-Hall of India, 2004.

References

- [23] G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell, "Engineering Optimization: Methods and Applications", New York: Wiley, 1983.
- [24] M. M. Hammad, "Deep Learning Activation Functions: Fixed-Shape, Parametric, Adaptive, Stochastic, Miscellaneous, Non-Standard, Ensemble", *in: arXiv*, 2024. doi: 10.48550/arXiv.2407.11090. [Online]. Available: <https://doi.org/10.48550/arXiv.2407.11090>
- [25] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning", MIT Press, 2016.
- [26] C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006.
- [27] T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction", 2nd ed., Springer, 2008. doi: 10.1007/978-0-387-84858-7. [Online]. Available: <https://doi.org/10.1007/978-0-387-84858-7>
- [28] J. Patterson and A. Gibson, "Deep Learning: A Practitioner's Approach", O'Reilly Media, 2017.
- [29] U. Michelucci, "Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks", Apress, 2018.
- [30] J. Krohn, G. Beyleveld, and A. Bassens, "Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence", Addison-Wesley Professional, 2019.
- [31] A. W. Trask, "Grokking Deep Learning", Manning Publications, 2019.
- [32] A. Géron, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems", O'Reilly Media, 2019.
- [33] K. Chaudhury, "Math and Architectures of Deep Learning", Manning Publications, 2021.
- [34] E. Raff, "Inside Deep Learning: Math, Algorithms, Models", Manning Publications, 2022.
- [35] N. Buduma, N. Buduma, and J. Papa, "Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms", O'Reilly Media, 2022.
- [36] C. M. Bishop and H. Bishop, "Deep Learning Foundations and Concepts", Springer, 2023.
- [37] C. C. Aggarwal, "Neural Networks and Deep Learning: A Textbook", Springer, 2023. doi: 10.1007/978-3-031-29642-0. [Online]. Available: <https://doi.org/10.1007/978-3-031-29642-0>
- [38] S. J. D. Prince, Understanding Deep Learning, Cambridge, MA: MIT Press, 2023.
- [39] M. M. Hammad, "Comprehensive Survey of Complex-Valued Neural Networks: Insights into Backpropagation and Activation Functions", *in: arXiv*, 2024. doi: 10.48550/arXiv.2407.19258. [Online]. Available: <https://doi.org/10.48550/arXiv.2407.19258>
- [40] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *in Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1, 2018. [Online]. Available: <http://jmlr.org/papers/v18/17-468.html>
- [41] C. Elliott, "The simple essence of automatic differentiation", *in Proceedings of the ACM on Programming Languages*, vol. 2, no. 70, pp. 1, 2018. doi: 10.1145/3236765. [Online]. Available: <https://doi.org/10.1145/3236765>
- [42] C. C. Margossian, "A review of automatic differentiation and its efficient implementation", *in WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 4, 2018. doi: 10.1002/widm.1305. [Online]. Available: <https://doi.org/10.1002/widm.1305>

References

- [43] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch", in *NIPS 2017 Workshop Autodiff*, Tech. Rep., 2017. [Online]. Available: <https://openreview.net/forum?id=BJJsrmfCZ>
- [44] G. Cybenko, "Approximation by superpositions of a sigmoidal function", in *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303, 1989. doi: 10.1007/BF02551274. [Online]. Available: <https://doi.org/10.1007/BF02551274>
- [45] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators", in *Neural Networks*, vol. 2, no. 5, pp. 359, 1989. doi: 10.1016/0893-6080(89)90020-8. [Online]. Available: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8)
- [46] A. Pinkus, "Approximation theory of the MLP model in neural networks", in *Acta Numerica*, vol. 8, pp. 143, 1999. doi: 10.1017/S0962492900002919. [Online]. Available: <https://doi.org/10.1017/S0962492900002919>
- [47] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (PMLR), vol. 9, pp. 249, 2010. [Online]. Available: <https://proceedings.mlr.press/v9/glorot10a.html>
- [48] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification", in *Proceedings of the IEEE International Conference on Computer Vision* (ICCV), pp. 1026, 2015. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html
- [49] S. K. Kumar, "On weight initialization in deep neural networks", in *arXiv*, 2017. doi: 10.48550/arXiv.1704.08863. [Online]. Available: <https://doi.org/10.48550/arXiv.1704.08863>
- [50] W. Cao, X. Wang, Z. Ming, and J. Gao, "A review on neural networks with random weights", in *Neurocomputing*, vol. 275, pp. 278, 2018. doi: 10.1016/j.neucom.2017.08.040. [Online]. Available: <https://doi.org/10.1016/j.neucom.2017.08.040>
- [51] W. Boulila, M. Driss, M. Al-Sarem, F. Saeed, and M. Krichen, "Weight initialization techniques for deep learning algorithms in remote sensing: Recent trends and future perspectives", in F. Saeed, T. Al-Hadhrami, E. Mohammed, and M. Al-Sarem, Eds., *Advances on Smart and Soft Computing*: vol. 1399 of *Advances in Intelligent Systems and Computing* (AISC), Springer, Singapore, pp. 477, 2022. doi: 10.1007/978-981-16-5559-3_39. [Online]. Available: https://doi.org/10.1007/978-981-16-5559-3_39
- [52] M. V. Narkhede, P. P. Bartakke, and M. S. Sutaone, "A review on weight initialization strategies for neural networks", in *Artificial Intelligence Review*, vol. 55, pp. 291, 2022. doi: 10.1007/s10462-021-10033-z. [Online]. Available: <https://doi.org/10.1007/s10462-021-10033-z>
- [53] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, "Efficient backprop", in G. B. Orr and K. R. Müller, Eds., *Neural Networks: Tricks of the Trade*: vol. 1524 of *Lecture Notes in Computer Science* (LNCS), Springer, Berlin, Heidelberg, 1998. doi: 10.1007/3-540-49430-8_2. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_2
- [54] N. N. Schraudolph, "Centering neural network gradient factor", in G. B. Orr and K. R. Müller, Eds., *Neural Networks: Tricks of the Trade*: vol. 1524 of *Lecture Notes in Computer Science* (LNCS), Springer, Berlin, Heidelberg, pp. 207, 1998. doi: 10.1007/3-540-49430-8_11. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_11
- [55] Reprint of: Mahalanobis, P.C. (1936) "On the Generalised Distance in Statistics", in *Sankhya A*, vol. 80, pp. 1, 2018. doi: 10.1007/s13171-019-00164-5. [Online]. Available: <https://doi.org/10.1007/s13171-019-00164-5>
- [56] X. D. Zhang, "A Matrix Algebra Approach to Artificial Intelligence", Springer, Singapore, 2020. doi: 10.1007/978-981-15-2770-8. [Online]. Available: <https://doi.org/10.1007/978-981-15-2770-8>

References

- [57] M. G. Kim, "Multivariate outliers and decompositions of Mahalanobis distance", in *Communications in Statistics - Theory and Methods*, vol. 29, no. 7, pp. 1511, 2000. doi: 10.1080/03610920008832559. [Online]. Available: <https://doi.org/10.1080/03610920008832559>
- [58] B. S. S. Mohan and C. C. Sekhar, "Class-specific Mahalanobis distance metric learning for biological image classification", in A. Campilho and M. Kamel, Eds., *Image Analysis and Recognition: 9th International Conference, ICIAR-2012*: vol. 7325 of Lecture Notes in Computer Science (LNCS), Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-31298-4_29. [Online]. Available: https://doi.org/10.1007/978-3-642-31298-4_29
- [59] R. De Maesschalck, D. Jouan-Rimbaud, and D. L. Massart, "The Mahalanobis distance", in *Chemometrics and Intelligent Laboratory Systems*, vol. 50, no. 1, pp. 1, 2000. doi: 10.1016/S0169-7439(99)00047-7. [Online]. Available: [https://doi.org/10.1016/S0169-7439\(99\)00047-7](https://doi.org/10.1016/S0169-7439(99)00047-7)
- [60] G. Martos, A. Muñoz, and J. González, "On the generalization of the Mahalanobis distance", in J. Ruiz-Shulcloper and G. S. di Baja, Eds., *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, Lecture Notes in Computer Science (LNCS)*, vol. 8258, pp. 125, Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-41822-8_16. [Online]. Available: https://doi.org/10.1007/978-3-642-41822-8_16
- [61] L. Novák and M. Vořechovský, "Generalization of coloring linear transformation", in *Transactions of the VSB - Technical University of Ostrava, Civil Engineering Series*, vol. 18, no. 2, pp. 31, 2018. doi: 10.31490/tces-2018-0013. [Online]. Available: <https://doi.org/10.31490/tces-2018-0013>
- [62] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *Proceedings of the 32nd International Conference on Machine Learning (PMLR)*, vol. 37, pp. 448, 2015. [Online]. Available: <https://proceedings.mlr.press/v37/ioffe15.html>
- [63] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization", in *arXiv*. doi: 10.48550/arXiv.1607.06450. [Online]. Available: <https://doi.org/10.48550/arXiv.1607.06450>
- [64] J. Konar, P. Khandelwal, and R. Tripathi, "Comparison of various learning rate scheduling techniques on convolutional neural network", in *IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, pp. 1, 2020. doi: 10.1109/SCEECS48394.2020.94. [Online]. Available: <https://doi.org/10.1109/SCEECS48394.2020.94>
- [65] S. Biswas and S. Dey, "Relative learning rate adaptation on loss feedback", in *TechRxiv*, 2023. doi: 10.36227/techrxiv.21980120.v1. [Online]. Available: <https://www.techrxiv.org/doi/full/10.36227/techrxiv.21980120.v1>
- [66] TensorFlow, "InverseTimeDecay," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/InverseTimeDecay
- [67] TensorFlow, "ExponentialDecay," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/ExponentialDecay
- [68] S. Li, L. Guo, and J. Liu, "Towards East Asian facial expression recognition in the real world: A new database and deep recognition baseline", in *Sensors*, vol. 22, no. 21, pp. 8089, 2022. doi: 10.3390/s22218089. [Online]. Available: <https://doi.org/10.3390/s22218089>
- [69] P. Mishra and K. Sarawadekar, "Polynomial learning rate policy with warm restart for deep neural network", in *IEEE Region 10 Conference (TENCON)*, pp. 2087, 2019. doi: 10.1109/TENCON.2019.8929465. [Online]. Available: <https://doi.org/10.1109/TENCON.2019.8929465>
- [70] W. Liu, A. Rabinovich, and A. C. Berg, "Parsenet: Looking wider to see better", in *arXiv*, 2015. doi: 10.48550/arXiv.1506.04579. [Online]. Available: <https://doi.org/10.48550/arXiv.1506.04579>

References

- [71] Y. Dauphin, H. de Vries, and Y. Bengio, "Equilibrated adaptive learning rates for non-convex optimization", in *Advances in Neural Information Processing Systems*, vol. 28, pp. 1504, 2015. [Online]. Available: <https://proceedings.neurips.cc/paper/2015/hash/430c3626b879b4005d41b8a46172e0c0-Abstract.html>
- [72] L. N. Smith, "Cyclical learning rates for training neural networks", in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464, 2017. doi: 10.1109/WACV.2017.58. [Online]. Available: <https://doi.org/10.1109/WACV.2017.58>
- [73] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts", in *arXiv*, 2016. doi: 10.48550/arXiv.1608.03983. [Online]. Available: <https://doi.org/10.48550/arXiv.1608.03983>
- [74] A. Gotmare, N. S. Keskar, C. Xiong, and R. Socher, "A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation", in *arXiv*, 2018. doi: 10.48550/arXiv.1810.13243. [Online]. Available: <https://doi.org/10.48550/arXiv.1810.13243>
- [75] W. An, H. Wang, Y. Zhang, and Q. Dai, "Exponential decay sine wave learning rate for fast deep neural network training", in *IEEE Visual Communications and Image Processing (VCIP)*, pp. 1, 2017. doi: 10.1109/VCIP.2017.8305126. [Online]. Available: <https://doi.org/10.1109/VCIP.2017.8305126>
- [76] B. T. Polya, "Some methods of speeding up the convergence of iteration methods", in *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1, 1964. doi: 10.1016/0041-5553(64)90137-5. [Online]. Available: [https://doi.org/10.1016/0041-5553\(64\)90137-5](https://doi.org/10.1016/0041-5553(64)90137-5)
- [77] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning", in *Proceedings of the 30th International Conference on Machine Learning (PMLR)*, vol. 28, no. 3, pp. 1139, 2013. [Online]. Available: <https://proceedings.mlr.press/v28/sutskever13.html>
- [78] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ", in *J. Dokl. Akad. Nauk SSSR*, vol. 269, pp. 543, 1983. [Online]. Available: <http://mi.mathnet.ru/dan46009>
- [79] Y. Nesterov, "Introductory Lectures on Convex Optimization: A Basic Course", Springer, 2004.
- [80] A. Mustapha, L. Mohamed, and K. Ali, "Comparative study of optimization techniques in deep learning: Application in the ophthalmology field", in *Journal of Physics: Conference Series*, vol. 1743, no. 1, 2021. doi: 10.1088/1742-6596/1743/1/012002. [Online]. Available: <https://doi.org/10.1088/1742-6596/1743/1/012002>
- [81] D. Soydaner, "A comparison of optimization algorithms for deep learning", in *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 34, no. 13, pp. 2052013, 2020. doi: 10.1142/S0218001420520138. [Online]. Available: <https://doi.org/10.1142/S0218001420520138>
- [82] M. J. Kochenderfer and T. A. Wheeler, "Algorithms for Optimization", MIT Press, 2019.
- [83] S. Santra, J. W. Hsieh, and C. F. Lin, "Gradient descent effects on differential neural architecture search: A survey", in *IEEE Access*, vol. 9, pp. 89602, 2021. doi: 10.1109/ACCESS.2021.3090918. [Online]. Available: <https://doi.org/10.1109/ACCESS.2021.3090918>
- [84] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation", in *Neural Networks*, vol. 1, no. 4, pp. 295, 1988. doi: 10.1016/0893-6080(88)90003-2. [Online]. Available: [https://doi.org/10.1016/0893-6080\(88\)90003-2](https://doi.org/10.1016/0893-6080(88)90003-2)
- [85] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", in *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121, 2011. [Online]. Available: <http://jmlr.org/papers/v12/duchi11a.html>
- [86] G. Hinton, "Neural networks for machine learning", Coursera, video lectures, 2012.

References

- [87] M. D. Zeiler, "Adadelta: An adaptive learning rate method", *in arXiv*, 2012. doi: 10.48550/arXiv.1212.5701. [Online]. Available: <https://doi.org/10.48550/arXiv.1212.5701>
- [88] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *in arXiv*, 2014. doi: 10.48550/arXiv.1412.6980. [Online]. Available: <https://doi.org/10.48550/arXiv.1412.6980>
- [89] T. Dozat, "Incorporating Nesterov momentum into Adam", *in International Conference on Learning Representations (ICLR)*, Workshop Track, pp. 1, 2016. [Online]. Available: <https://openreview.net/forum?id=OM0jvwB8jIp57ZJjtNEZ>
- [90] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of Adam and beyond", *in arXiv*, 2019. doi: 10.48550/arXiv.1904.09237. [Online]. Available: <https://doi.org/10.48550/arXiv.1904.09237>
- [91] A. Antoniou and W. S. Lu, "Practical Optimization: Algorithms and Engineering Applications", New York: Springer, 2021.
- [92] S. M. Goldfeld, R. E. Quandt, and H. F. Trotter, "Maximization by quadratic hill-climbing", *in Econometrica*, vol. 34, no. 3, pp. 541, 1966. doi: 10.2307/1909768. [Online]. Available: <https://doi.org/10.2307/1909768>
- [93] M. R. Hestenes and E. L. Stiefel, "Methods of conjugate gradients for solving linear systems", *in J. Res. Natl. Bur. Stand.*, vol. 49, pp. 409, 1952. [Online]. Available: <https://cir.nii.ac.jp/crid/1573105975360024576>
- [94] D. Chicco and G. Jurman, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation", *in BMC Genomics*, vol. 21, no. 6, 2020. doi: 10.1186/s12864-019-6413-7. [Online]. Available: <https://doi.org/10.1186/s12864-019-6413-7>
- [95] D. Chicco, N. Toetsch, and G. Jurman, "The Matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation", *in BioData Mining*, vol. 14, no. 13, 2021. doi: 10.1186/s13040-021-00244-z. [Online]. Available: <https://doi.org/10.1186/s13040-021-00244-z>
- [96] T. Fawcett, "An introduction to ROC analysis", *in Pattern Recognition Letters*, vol. 27, no. 8, pp. 861, 2006. doi: 10.1016/j.patrec.2005.10.010. [Online]. Available: <https://doi.org/10.1016/j.patrec.2005.10.010>
- [97] D. M. W. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation", *in arXiv*, 2020. doi: 10.48550/arXiv.2010.16061. [Online]. Available: <https://doi.org/10.48550/arXiv.2010.16061>
- [98] F. Provost and T. Fawcett, "Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking", O'Reilly Media, Inc., 2013.
- [99] A. Tharwat, "Classification assessment methods", *in Applied Computing and Informatics*, vol. 17, no. 1, pp. 168, 2021. doi: 10.1016/j.aci.2018.08.003. [Online]. Available: <https://doi.org/10.1016/j.aci.2018.08.003>
- [100] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", *in Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281, 2012. [Online]. Available: <http://jmlr.org/papers/v13/bergstra12a.html>
- [101] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization", *in Advances in Neural Information Processing Systems (NIPS)*, vol. 24, pp. 2546, 2011. [Online]. Available: <https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cf12577bc2619bc635690-Abstract.html>
- [102] L. Zahedi and F. G. Mohammad, "Search algorithms for automated hyper-parameter tuning", *in arXiv*, 2021. doi: 10.48550/arXiv.2104.14677. [Online]. Available: <https://doi.org/10.48550/arXiv.2104.14677>
- [103] H.-C. Kim and M.-J. Kang, "Comparison of hyper-parameter optimization methods for deep neural networks", *in Journal of IKEEE*, vol. 24, no. 4, pp. 969, 2020. doi: 10.7471/ikeee.2020.24.4.969. [Online]. Available: <https://doi.org/10.7471/ikeee.2020.24.4.969>

References

- [104] D. A. Anggoro and S. S. Mukti, "Performance comparison of grid search and random search methods for hyperparameter tuning in extreme gradient boosting algorithm to predict chronic kidney failure", in *International Journal of Intelligent Engineering and Systems*, vol. 14, pp. 198, 2021. [Online]. Available: <https://inass.org/wp-content/uploads/2021/10/2021123119.pdf>
- [105] R. Hossain and D. Timmer, "Machine learning model optimization with hyper parameter tuning approach", in *Global Journal of Computer Science and Technology*, vol. 21 (D2), pp. 7, 2021. [Online]. Available: <https://computerresearch.org/index.php/computer/article/view/2059>
- [106] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization", in *IEEE Congress on Evolutionary Computation (CEC)*, pp. 1551, 2021. doi: 10.1109/CEC45853.2021.9504761. [Online]. Available: <https://doi.org/10.1109/CEC45853.2021.9504761>
- [107] A. Nugroho and H. Suhartanto, "Hyper-parameter tuning based on random search for densenet optimization", in *7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pp. 96, 2020. doi: 10.1109/ICITACEE50144.2020.9239164. [Online]. Available: <https://doi.org/10.1109/ICITACEE50144.2020.9239164>
- [108] B. H. Shekar and G. Dagnew, "Grid search-based hyperparameter tuning and classification of microarray cancer data", in *Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, pp. 1, 2019. doi: 10.1109/ICACCP.2019.8882943. [Online]. Available: <https://doi.org/10.1109/ICACCP.2019.8882943>
- [109] E. Bernard, "Introduction to Machine Learning", Wolfram Media, Inc., 2021.
- [110] C. K. Williams and C. E. Rasmussen, "Gaussian Processes for Machine Learning", MIT Press, 2006.
- [111] R. B. Gramacy, "Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences", 1st ed., New York: Chapman and Hall/CRC, 2020.
- [112] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning", in *arXiv*, 2010. doi: 10.48550/arXiv.1012.2599. [Online]. Available: <https://doi.org/10.48550/arXiv.1012.2599>
- [113] L. Nguyen, "Tutorial on Bayesian optimization", in *preprints*, 2023. doi: 10.20944/preprints202303.0292.v1. [Online]. Available: <https://doi.org/10.20944/preprints202303.0292.v1>
- [114] P. I. Frazier, "A tutorial on Bayesian optimization", in *arXiv*, 2018. doi: 10.48550/arXiv.1807.02811. [Online]. Available: <https://doi.org/10.48550/arXiv.1807.02811>
- [115] A. A. Pawar and U. Warbhe, "Optimizing Bayesian acquisition functions in Gaussian processes", in *arXiv*, 2021. doi: 10.48550/arXiv.2111.04930. [Online]. Available: <https://doi.org/10.48550/arXiv.2111.04930>
- [116] G. D. Ath, R. M. Everson, A. A. M. Rahat, and J. E. Fieldsend, "Greed is good: Exploration and exploitation trade-offs in Bayesian optimisation", in *ACM Transactions on Evolutionary Learning and Optimization*, vol. 1, no. 1, pp. 1, 2021. doi: 10.1145/3425501. [Online]. Available: <https://doi.org/10.1145/3425501>
- [117] D. D. Cox and S. John, "A statistical method for global optimization", in *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 2, pp. 1241, 1992. doi: 10.1109/ICSMC.1992.271617. [Online]. Available: <https://doi.org/10.1109/ICSMC.1992.271617>
- [118] C. Bian, X. Wang, C. Liu, X. Xie, and L. Haitao, "Impact of exploration-exploitation trade-off on UCB-based Bayesian optimization", in *IOP Conference Series: Materials Science and Engineering*, vol. 1081, pp. 012023, 2021. doi: 10.1088/1757-899X/1081/1/012023. [Online]. Available: <https://doi.org/10.1088/1757-899X/1081/1/012023>

References

- [119] U. Noe and D. Husmeier, "On a new improvement-based acquisition function for Bayesian optimization", *in arXiv*, 2018. doi: 10.48550/arXiv.1808.06918. [Online]. Available: <https://doi.org/10.48550/arXiv.1808.06918>
- [120] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design", *in arXiv*, 2010. doi: 10.48550/arXiv.0912.3995. [Online]. Available: <https://doi.org/10.48550/arXiv.0912.3995>
- [121] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise", *in Journal of Basic Engineering*, vol. 86, no. 1, pp. 97, 1964. doi: 10.1115/1.3653121. [Online]. Available: <https://doi.org/10.1115/1.3653121>
- [122] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast Bayesian optimization of machine learning hyperparameters on large datasets", *in Proceedings of the 20th International Conference on Artificial Intelligence and Statistics* (PMLR), vol. 54, pp. 528, 2017. [Online]. Available: <https://proceedings.mlr.press/v54/klein17a.html>
- [123] S. Ament, S. Daulton, D. Eriksson, M. Balandat, and E. Bakshy, "Unexpected improvements to expected improvement for Bayesian optimization", *in Advances in Neural Information Processing Systems* (NeurIPS), vol. 36, 2023. [Online]. Available: <https://proceedings.neurips.cc/paper/2023/hash/419f72cbd568ad62183f8132a3605a2a-Abstract-Conference.html>
- [124] A. M. Vincent and P. Jidesh, "An improved hyperparameter optimization framework for AutoML systems using evolutionary algorithms", *in Scientific Reports*, vol. 13, no. 4737, 2023. doi: 10.1038/s41598-023-32027-3. [Online]. Available: <https://doi.org/10.1038/s41598-023-32027-3>
- [125] R. Astudillo and P. I. Frazier, "Bayesian optimization of composite functions", *in Proceedings of the 36th International Conference on Machine Learning* (PMLR), vol. 97, pp. 354, 2019. [Online]. Available: <https://proceedings.mlr.press/v97/astudillo19a.html>
- [126] J. Kong, T. Pourmohamad, and H. K. H. Lee, "Understanding an acquisition function family for Bayesian optimization", *in arXiv*, 2023. doi: 10.48550/arXiv.2310.10614. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.10614>
- [127] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Structured sparsity through convex optimization", *in Statistical Science*, vol. 27, no. 4, pp. 450, 2012. doi: 10.1214/12-STS394. [Online]. Available: <https://doi.org/10.1214/12-STS394>
- [128] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Optimization with sparsity-inducing penalties", *in Foundations and Trends in Machine Learning*, vol. 4, no. 1, pp. 1, 2012. doi: 10.1561/2200000015. [Online]. Available: <http://dx.doi.org/10.1561/2200000015>
- [129] J. Mairal, F. Bach, and J. Ponce, "Sparse modeling for image and vision processing", *in Foundations and Trends® in Computer Graphics and Vision*, vol. 8, no. 2-3, pp. 85, 2014. doi: 10.1561/0600000058. [Online]. Available: <http://dx.doi.org/10.1561/0600000058>
- [130] A. W. Yu, H. Su, and L. Fei-Fei, "Efficient Euclidean projections onto the intersection of norm balls", *in arXiv*, 2012. doi: 10.48550/arXiv.1206.4638. [Online]. Available: <https://doi.org/10.48550/arXiv.1206.4638>
- [131] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra, "Efficient projections onto the L1-ball for learning in high dimensions", *in the 25th international conference on Machine learning* (ICML), pp. 272, 2008. doi: 10.1145/1390156.1390191. [Online]. Available: <https://doi.org/10.1145/1390156.1390191>
- [132] K. Usman, H. Gunawan, and A. B. Suksmono, "Compressive sensing reconstruction algorithm using L1-norm minimization via L2-norm minimization", *in International Journal on Electrical Engineering and Informatics*, vol. 10, pp. 37, 2018. doi: 10.15676/ijeei.2018.10.1.3. [Online]. Available: <http://dx.doi.org/10.15676/ijeei.2018.10.1.3>
- [133] J. Songsiri, "Projection onto an L1-norm ball with application to identification of sparse autoregressive models", *in Asean Symposium on Automatic Control* (ASAC), 2011.

References

- [134] S. Rosset and J. Zhu, "Sparse, flexible and efficient modeling using L1 regularization", in *Studies in Fuzziness and Soft Computing*, vol. 207, Springer, Berlin, Heidelberg, pp. 375, 2006.
- [135] Y. Kim, M. J. Kim, and H. Kim, "Scaled norm-based Euclidean projection for sparse speaker adaptation", in *EURASIP Journal on Advances in Signal Processing*, no. 102, 2015. doi: 10.1186/s13634-015-0290-2. [Online]. Available: <https://doi.org/10.1186/s13634-015-0290-2>
- [136] M. Unser, "Representer theorems for sparsity-promoting L1 regularization", in *IEEE Transactions on Information Theory*, vol. 62, no. 9, pp. 5167, 2016. doi: 10.1109/TIT.2016.2590421. [Online]. Available: <https://doi.org/10.1109/TIT.2016.2590421>
- [137] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Convex optimization with sparsity-inducing norms", in S. Sra, S. Nowozin, and S. J. Wright, Eds., *Optimization for Machine Learning*, Neural Information Processing Series, The MIT Press, 2011.
- [138] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net", in *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 67, no. 2, pp. 301, 2005. doi: 10.1111/j.1467-9868.2005.00503.x. [Online]. Available: <https://doi.org/10.1111/j.1467-9868.2005.00503.x>
- [139] L. Breiman, "Bagging predictors", in *Machine Learning*, vol. 24, pp. 123, 1996. doi: 10.1007/BF00058655. [Online]. Available: <https://doi.org/10.1007/BF00058655>
- [140] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", in *the journal of machine learning research*, vol. 15, pp. 1929, 2014.