Formulaic Pythonic Pseudocode: Making Machine Learning Intuitive and Accessible for Developers and Students

Hadrian Lazic

April, 27, 2025

Abstract

Traditional mathematical notation in machine learning, especially in calculus-heavy fields like optimization, often creates unnecessary barriers for developers. Core techniques like the Adam optimizer are widely used but are typically buried in dense, symbolic math that obscures their logic and practical implementation.

This paper reimagines foundational ML concepts, including derivatives, gradients, and optimizers, through intuitive, Pythonic pseudocode and annotated examples. By replacing compressed, symbolic notation with formula-inspired code and visual breakdowns, it makes machine learning theory more transparent, approachable, and actionable.

The goal is to empower developers, especially those without advanced math backgrounds, to understand, implement, and innovate with confidence. In doing so, this work opens the field of machine learning to a broader, more creative programming community, lowering the barriers to entry and accelerating innovation.

1 Introduction

Mathematics, at its core, is just another language, a tool designed to abstract and communicate general ideas. In the world of machine learning (ML), these ideas are most often expressed through the language of calculus. Nearly all core ML algorithms are formalized as calculus-based equations, a field of mathematics developed in the 17th century to express dynamic and abstract concepts.

Yet in modern times, machine learning is implemented through code. And here lies a problem: while many developers are exceptionally skilled in programming and low-level systems, most have not been taught the calculus necessary to deeply understand the original mathematical formulations behind the algorithms they use. This disconnect prevents even advanced programmers from contributing meaningfully to the improvement or customization of algorithmic logic, simply because the underlying math is inaccessible. Even more concerning is the broader issue of educational inequity. A significant number of students, especially those from under-resourced schools, never receive formal instruction in calculus. Despite having strong coding skills and logical thinking, they remain excluded from deeper ML development due to this mathematical barrier. This artificially shrinks the talent pool and slows innovation, especially in a field that benefits from diverse perspectives and creative problem-solving.

While calculus remains a vital tool for developing machine learning theory, there is a need for a more inclusive bridge between math and code. This paper proposes a method for translating complex calculus-based ML formulas into **Formulaic Pythonic Pseudocode**, notation that retains the mathematical structure, yet is accessible, readable, and clear to those with a background in programming. By doing so, we eliminate the need to fluently read two separate "languages" mathematics and code, making foundational and advanced ML concepts more transparent, intuitive, and inclusive for a much wider community of developers. Even with improvements in Coding languages, the Formulaic Pythonic Pseudocode, would still hold up, because it not build around syntax, a changing thing, but coding conventions, such a for loop, which will always be a repeating loop for an iterations, like Sigma in Calculus.

2 Motivation and Background

As a student developer building frameworks in Rust, I found myself deeply challenged when working on one of my most ambitious projects: a Machine Learning framework. To implement all the algorithms that are in this very paper, I had to first decode the calculus behind them. The formulas were filled with cryptic symbols, many of which even the top math students at my school couldn't explain. For example, when I encountered the core equation for the Adam optimizer, it completely froze my peers. Nobody knew what the symbols meant or what steps they implied. But I didn't stop there. Through relentless research and trial and error, I translated the Adam optimizer into thirty-five lines of Rust, a modern language known for performance on par with traditional C-based programming environments, but with memory safety and a powerful package manager.

In the process, I realized something troubling: most machine learning papers weren't written to be understood by developers. The so-called "pseudocode" often wasn't code at all; it was just simplified calculus dressed up as programming logic. There were no loops, no data structures, just paragraphs of hyperparameters and unexplained terms. I want to bridge this gap: to help students, developers, and aspiring researchers understand ML not only mathematically, but practically. My goal is to offer a new, more accessible path into the field, starting with my own contributions.

3 Core Principles

This paper breaks complex Machine Learning ideas into approachable categories, aiming for clarity, transparency, and intuition.

- Clear definitions of key terms (e.g., weights, gradients)
- Graphs and diagrams to visualize data flow and processes
- Formulaic pseudocode to bridge mathematical concepts with code logic
- An emphasis on intuitive explanations before diving into formalism

Where appropriate, each major concept (e.g., GELU, Adam, Softmax) is typically introduced with:

1. A short, intuitive explanation.
2. Key constants and parameter breakdowns.
3. Supporting breakdowns and background details.
4. Formulaic pseudocode.
5. An in-depth process description.
6. Example use cases.
7. Common pitfalls or when not to use it.
8. Additional insights, if needed.
9. Incorporation of visual aids (graphs, diagrams) to enhance comprehension when beneficial. Slightly more academic, but still clear.

While not every topic requires every breakdown, this structure serves as a guiding framework to maximize accessibility and depth.

4 Target Audience

This work is written for:

- High school students learning Machine Learning.
- Coding bootcamp graduates.
- Developers without a strong calculus background.
- Anyone frustrated by unreadable ML papers.
- Current and future educators interested in rethinking traditional ML instruction models to better empower the next generation of developers and researchers.

It is **not** intended for those already fluent in math-heavy ML theory, rather, it fills the gap for coders who want to go deeper without waiting for university-level math.

5 Vision for the Future of ML (Machine Learning) Education

The traditional approach to teaching machine learning often relies heavily on abstract mathematical formalism, leaving many capable students and developers feeling excluded. This work proposes an alternative: by emphasizing intuitive, structured pseudocode and visual aids, we can make advanced machine learning concepts more accessible without sacrificing depth.

I envision a future where educators, curriculum designers, and self-learners embrace intuitive-first frameworks, empowering a broader, more diverse generation of machine learning innovators. By bridging the gap between coding logic and mathematical theory, we can create an environment where intuition and rigor grow together, not apart.

This paper aims to serve as both a tool for learners and a call to action for educators ready to rethink and reimagine the future of ML education.

6 Contribution Summary

In this paper, I present a new approach to machine learning accessibility by translating complex calculus-based ML concepts into code-like pseudocode, a format that mirrors real-world programming logic. Firstly, this includes inventing a new pseudo-code format called Formulaic Pythonic Pseudocode, to make ML formulas that were in calculus now accessible to developers who have not learned calculus, via code that makes use of conventions across coding languages. Existing researchers don't do this; they use mathematical pseudocode, which is not programmer-readable pseudocode. This is important, because it will allow even high schoolers, bootcamp grads, and junior developers to contribute to ML model development earlier. This paper also aims to set a new standard for ML education resources. The paper's Formulaic Pseudocode is designed to be a teaching tool and used as a reference by developers in real projects. My Rust ML framework successfully used the ideas from the paper as well.

The paper also created a structured framework of the core ML concepts, each with a simplified language description, visuals (like graphs), clear breakdowns of constants and parameters, and the Formulaic Pseudocode.

License & Attribution

Author GitHub: `https://github.com/had2020`

Contents

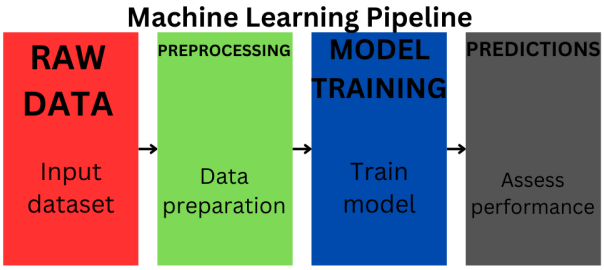

Figure 1: Neural networks follow the standard Machine Learning pipeline:starting with data collection and preprocessing, followed by model training, validation, and testing, in order to iteratively learn patterns and improve performance over time.

7 Fundations to Machine Learning

Machine Learning (ML) is the field focused on enabling machines to identify patterns within data. In the context of ML, these discovered patterns represent a form of intelligence. Machine learning networks are composed of interconnected parts (such as neurons and layers) and operate through a sequence of computational steps.

- **Gradient:** Tensor or Scalar value.
- **Scalar:** Having one parameter, or one value in your Tensor.
- **Vector:** One Dimensional Tensor.
- **Tensor:** Having more then one parameter, within a matrix.
- **Target:** The Gradient, you wish your network to have, in your output layer.
- **Output:** The Gradient, That is in the last layer of your network. Also known as the output layer.
- **Weights, and Bias:** The data within each layer split into a Tensor called Weights, and another called Bias. Together these store the connection strength between the values, think of the physical connections in a brain.
- **Underfitting and Overfitting:** If your model memorizes all the training data instead of learning patterns, it is overfitting.
- *There are other parts of this paper, which go more in depth with these concepts.*

7.1 What can a Tensor store

Tensors can be thought of as multidimensional arrays, and in the case of 2D tensors, they resemble matrices defined by an X and Y axis, essentially a flat grid of values. In most machine learning applications, 32-bit floating point (float32 or f32) or even half-precision formats are used, as 64-bit precision (float64 or f64) is typically unnecessary. Using lower precision reduces memory usage and computational overhead without significantly affecting performance.

7.2 Layer in a Machine Learning Context

A neural network typically consists of multiple layers. Each layer contains a weights tensor and a bias tensor. The weights control how input signals are scaled, while the bias tensor helps the network capture more complex relationships that cannot be learned through weighted inputs alone.

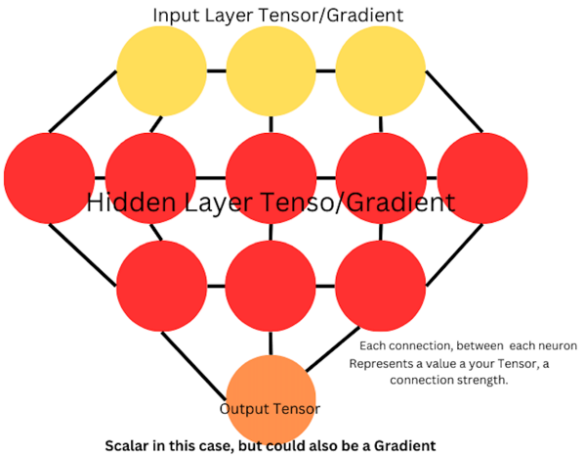

Figure 2: Basic structure of a fully connected neural network, showing Input, Hidden, and Output layers.

7.3 Neural Network

7.3.1 Understanding the Neural Network

Notice how each weight/neuron within the network is connected to nearby neurons in the adjacent top and bottom layers.

1. **Input:** Accepts raw data, which can be normalized to prevent gradient issues.
2. **Hidden:** Applies matrix multiplication to propagate signal strengths throughout the network.
3. **Output:** Produces the final prediction, which can be compared to the target to compute the network error.

Training Steps Overview: Neural networks must follow a series of steps to train correctly:

1. **Gather a Clean dataset:** Remove large outliers and ensure both input and target data are correctly paired.
2. **Define your model structure:** Choose the type of layers (Dense, Convolutional, Recurrent), determine the number of layers and units per layer, and select appropriate activation functions.
3. **Choose a loss function:** Examples include Mean Squared Error (MSE) or Mean Absolute Error (MAE).
4. **Choose an optimizer:** Use optimizers like Adam to adjust the learning rate and improve convergence based on gradients.
5. **Foward Pass:** Input is passed through the network. Each layer computes an activation, and a final prediction (output) is generated.
6. **Compute the Loss:** Compare the predicted output to the actual labels or target values using the chosen loss function.
7. **Backward Pass:** Compute the gradients of the loss with respect to all weights using the Chain Rule of calculus.
8. **Update weights:** Apply the optimizer to update the network's weights and biases based on the computed gradients.

There are dedicated sections within this paper that explore each of these processes in more detail. Forward Pass, Compute Loss, and Backward Pass are all parts of Backpropagation, the mechanism that allows neural networks to learn.

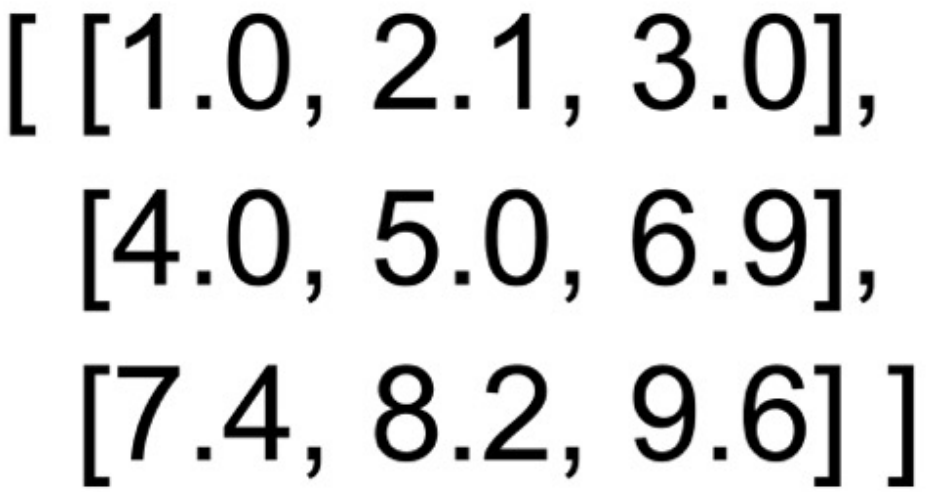

Figure 3: Example: Tensor/Matrix/Gradient, with shape: (3,3), Parameters: 9, Sum: 47.2

8 Tensors

In Machine Learning, tensors are often used to represent gradients, derivatives of functions, but tensors themselves can represent any multi-dimensional collection of data.

1. **What are parts, that make up Tensors.**
2. **Dimension:** In most cases in Machine Learning, we use the Second Dimension(2D). This paper will focus on the Second Dimension(2D), but concepts are still applicable.
3. **Shape:** In the Normal 2D Tensor/Matrix, we have an X and a Y. Also know as Rows and Cols, shorthand for Columns.
4. **Parameters:** The sum of X times Y, also known as the total number of connections.
5. **Weights:** The name of the main Tensor on each layer, the another is called bias.
6. **Bias:** The Tensor on each layer, which is added for complexity.
7. **Scalar vs Gradient:** Tensors could be both, a Tensor of with only one value, with a shape of (x:1, y:1), would be a Scalar. Gradients are bigger than the shape of a Scalar, and will require for loops.
8. **Logits:** Raw unnormalized data, in your Tensors.
9. **Element:** One specfic number inside the Tensor.

Some tensor operations require broadcasting, which automatically expands tensor shapes so that operations can be performed. This is especially important for operations like matrix multiplication, where tensors must have compatible shapes. Different types of tensor operations are used depending on the network design and computational needs.

(From scalar)-(vector)-(matrix 2D tensor)

8.1 Key Gradient Problems

Gradients get multipled many times throught the Machine Learning pipline, some values can become bigger than their datatype limits.

1. **Expoloding Gradients:** If the values of elements within a Tensor become so large, that they overflow the memory buffer specified by their datatype.
2. **Vanishing Gradients:** If the values of elements within a Tensor become so small, that they overflow the memory buffer specified by their datatype.

8.2 Tensor Rank NOT to be confused with Shape!

Rank and Shape, may sound quite similar, but they have some key differences in Machines Learning.

1. **Rank (Order):** Number of disensions (Scalers = 0, vectors = 1, matrices = 2).
2. **Shape:** The actual size of each dimension, for example (x3, y5).

8.3 Axis operations

Operations along axis 0 are performed row-wise, while operations along axis 1 are performed column-wise. For tensors with more than two dimensions, additional axes follow the same pattern, corresponding to deeper layers of structure.

8.4 Dot Product

Calculated by multiplying corresponding (parallel position in each matrix) elements of two vectors and then summing those products.

```
Function DotProduct(VectorA, VectorB) -> ScalarResult:

// Step 1: Make sure both vectors have the same length.
if VectorA.len() != VectorB.len():
    return Error("Vectors must be the same length.")

// Step 2: Initialize result.
ScalarResult = 0

// Step 3: Multiply corresponding elements and sum them.
for i in VectorA.len():
    ScalarResult = ScalarResult + (VectorA[i] * VectorB[i])

return ScalarResult
```

Listing 1: Formulaic Pythonic Code for Dot Product

Use Cases

1. Combining all the features in a network.
2. Help computing angle between two vectors.
3. Used in cosine similarity for comparing.

interpreting The Results

1. If they agree completely the result is big.
2. If they disagree (opposite direction) the result is negative.
3. If they're perpendicular? the result is zero.

Understanding dot products is foundational for operations like fully connected layers and similarity-based retrieval systems in machine learning

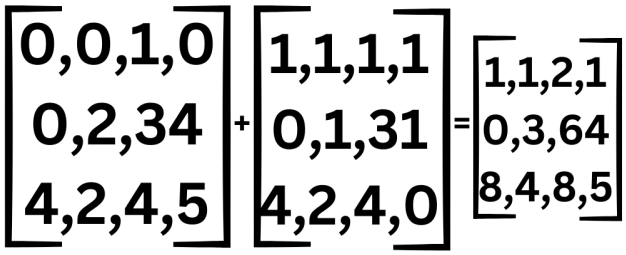

Figure 4: Tensor Addition

8.5 Tensor Addition

Tensors can have operations applied to them either by another Tensor. Addition is one of them. Tensor Addition can be done element-wise, with another Tensor of any size, unlike multiplication and divison operations.

```
Function Add(Tensor1, Tensor2) -> Tensor:

  ResultTensor = create empty tensor with same shape as Tensor1

  For row in Tensor1:
      For col in row:

          value1 = Tensor1[rowIndex][colIndex]
          value2 = Tensor2[rowIndex][colIndex]
          ResultTensor[rowIndex][colIndex] = value1 + value2

  Return ResultTensor
```

Listing 2: Formulaic Pythonic Code for Tensor Addition

You can use Tensor Addition to merge Graidents or Features during training, or add bias vectors in the output. You can also use it for Braodcasting, to create a tensor of the same shape, out of to differently shaped Tensors, in order to apply a matrix-wise operation, that is dependent on matching Tensor shapes.

8.6 Tensor Subtraction

Tensors can also be subtracted from each other. This operation is Element-wise. Keep in note it is the same as Tensor Addition, just the operation applied is subtraction.

```
Function Subtract(Tensor1, Tensor2) -> Tensor:

  ResultTensor = create empty tensor with same shape as Tensor1

  For row in Tensor1:
      For col in row:

          value1 = Tensor1[rowIndex][colIndex]
          value2 = Tensor2[rowIndex][colIndex]
          ResultTensor[rowIndex][colIndex] = value1 - value2

  Return ResultTensor
```

Listing 3: Formulaic Pythonic Code for Tensor Subtraction

8.7 Scalar Addition

One common way to perform Scalar Addition is through element-wise operations across an entire tensor. In this approach, the scalar value is added individually to each element of the tensor. This ensures a uniform transformation throughout the tensor. Alternatively, a scalar can be added using tensor broadcasting during tensor addition, but this often results in the scalar having minimal impact, especially at specific coordinates like (x1, y2), where its effect may be diluted by the dimensional context.

```
Function Add_Scalar(Tensor, Scalar) -> Tensor:
  For row in Tensor:
      For col in row:
        col += Scalar
```

Listing 4: Formulaic Pythonic Code for Scalar Addition

8.8 Scalar Subtraction

One common way to perform Scalar Subtraction is through element-wise operations across an entire tensor, with the result updating the element. In this approach, the scalar value is substracted individually to each element of the tensor. This ensures a uniform transformation throughout the tensor. Alternatively, a scalar can be subtracted using tensor broadcasting during tensor subtraction, but this often results in the scalar having minimal impact, especially at specific coordinates like (x1, y2), where its effect may be diluted by the dimensional context.

```
Function Sub_Scalar(Tensor, Scalar) -> Tensor:
  For row in Tensor:
      For col in row:
        col -= Scalar
```

Listing 5: Formulaic Pythonic Code for Scalar Subtraction

8.9 Scalar Multication

This is an Element-wise Multication operation preformed, to each index of the Tensor. Similary to Tensor Addition and Subtraction, only the operation is different.

```
Function Mut_Scalar(Tensor, Scalar) -> Tensor:
  For row in Tensor:
      For col in row:
        col = col * Scalar
```

Listing 6: Formulaic Pythonic Code for Scalar Multication

8.10 Scalar Division

Only the operation changed from Scalar Division to Scalar Multication.

```
Function Div_Scalar(Tensor, Scalar) -> Tensor:
  For row in Tensor:
      For col in row:
        col = col / Scalar
```

Listing 7: Formulaic Pythonic Code for Scalar Division

8.11 Use Cases of Tensor-Scalar Operations

Tensor to scalar operations in ML are essential for reducing multi-dimensional data into single, interpretable values. They are fundamental in loss functions (quantifying prediction errors), evaluation metrics (assessing model performance), regularization terms (penalizing model complexity), gradient aggregation (summarizing parameter updates), and for general summarization and monitoring of tensor states (like mean or norm).

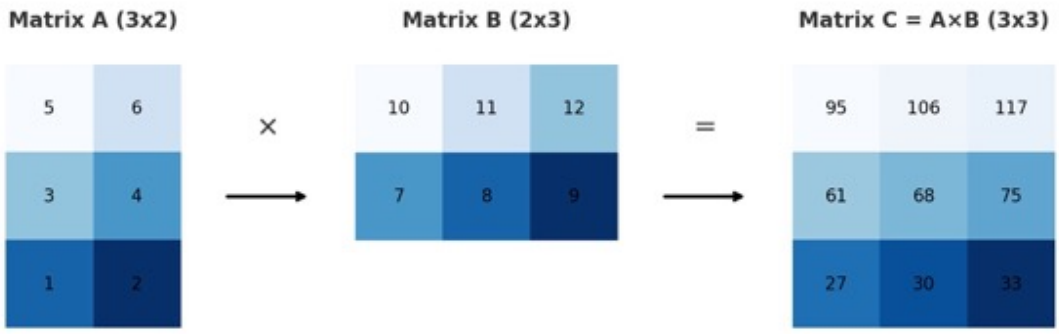

Figure 5: Example of Matrix Multiplication.

8.12 Matrix Multiplication

Matrix Multiplication, is one of the more complex operations on Tensors. Within the brain, neurons operate in parallel, continuously adjusting the strengths of their connections. This dynamic behavior can be mimicked in machine learning using matrix multiplication (MatMul), which allows us to simulate neural activity in a way that fits the linear processing style of digital computing hardware. Matrix multiplication is applied across entire matrices and requires compatible shapes, specifically, the number of columns in the first matrix must match the number of rows in the second. It also demands efficient access to large portions of the matrix data for optimal computation.

```
Function MatMul(TensorA, TensorB) -> TensorC:

// Get dimensions
rows_A = number of rows in TensorA
cols_A = number of columns in TensorA
cols_B = number of columns in TensorB

// Initialize TensorC with zeros of shape (rows_A, cols_B)
TensorC = empty matrix of size (rows_A x cols_B)

for row in rows_A:
  for col in cols_B:
    sum = 0
    for k in cols_A:
      sum += TensorA[row][k] * TensorB[k][col]
    TensorC[row][col] = sum

return TensorC
```

Listing 8: Formulaic Pythonic Code for Matrix Multiplication

Keep in mind that **k** represents the shared dimension index between the two tensors. It iterates through the elements being multiplied and summed, performing the core dot product step. This structure is essential for enabling linear parallelization in matrix multiplication.

8.12.1 Misconception: Matrix Division?

There is no Matrix Division, while Scalars could apply division to a whole Tensor, a Matrix applyig division, is not a defined general operation. As reverse transformations is too computational expansive.

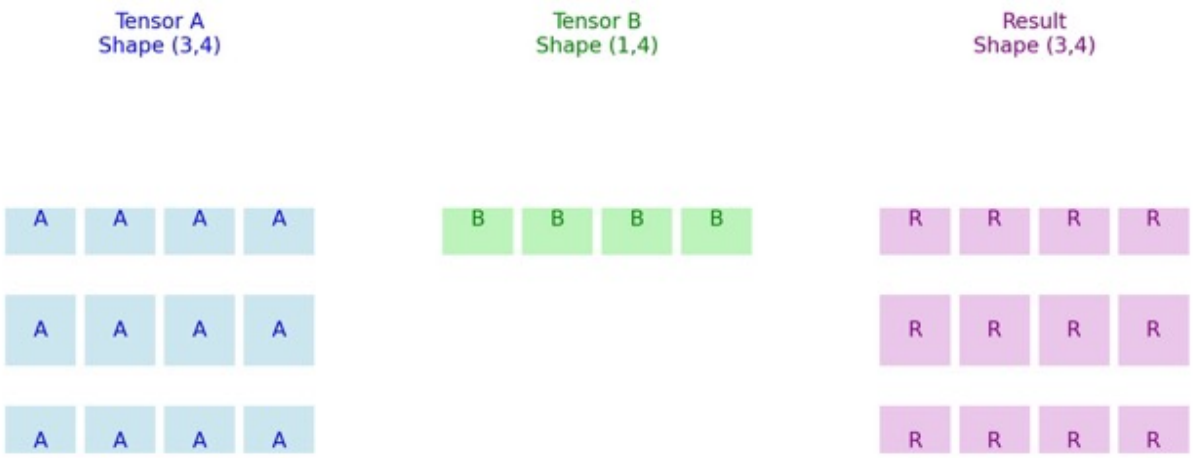

Figure 6: Example of correct broadcasting.

8.13 Broadcasting

When performing operations on tensors of differing shapes, refer to the Tensor Broadcasting Rules. These rules define how operations can be applied element-wise between tensors by automatically expanding their shapes to be compatible.

1. **BroadCasting Rules, for 2D Tensors.**
2. One Tensor is A, the other is B.
3. **A-Rows** equal **B-Rows**, or one of them is 1.
4. **A-Cols** equal **B-Cols**, or one of them is 1.
5. **If either dimension is 1**, it gets stretched to match the other.

So, in basic terms, if the dimension equal one, pad that one dimension. Otherwise if it's not one or equal, you have to try other methods: **Padding to Zero, Reshape, Unsqeeze, and Transpose.**

```
Function broadcastable(TensorA, TensorB) -> Tensor:

  if TensorA.rows == TensorB.rows or TensorA.rows == 1 or TensorB.rows == 1:
    if TensorA.cols == TensorB.cols or TensorA.cols == 1 or TensorB.cols == 1:
        //broadcast is valid
  else:
      //broadcast is invalid
```

Listing 9: Formulaic Pythonic Code for Tensor Subtraction

This check is important for **Matrix Multiplication**, because shapes need to be broadcastable in order for Matrix Multiplication to be applied.

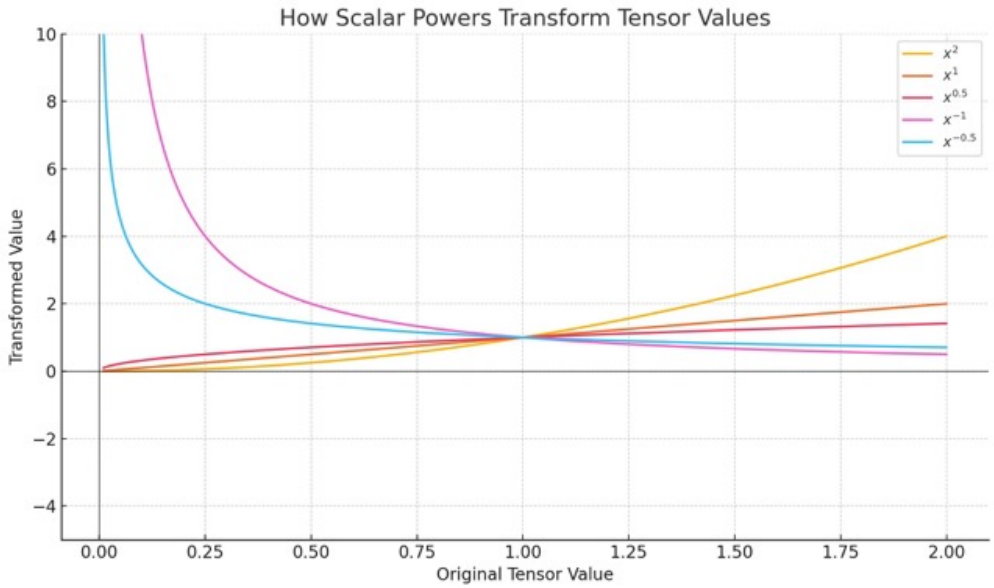

Figure 7: Powering is a Non-linear operation

8.14 Tensor to Raised/Powered, by a Scalar

This operation is performed element-wise. However, when raising a tensor to a scalar power, the operation remains element-wise, without the need for matrix multiplication.

```
Function Raise_by_Scalar(Tensor, Scalar) -> Tensor:
  for row in Tensor:
    for col in row:
      col = col ** Scalar
```

Listing 10: Formulaic Pythonic Code for Tensor Raised to a Tensor

8.14.1 Use Cases

1. **Activation Functions:** Some activation Functions power Maxtrixs by a Scalar.
2. **Regularization:** Keeping your model from generalizing the infomation/Overfitting.
3. **Normalization:** Some normalization Techniques involve squaring.
4. **Probability Distributions:** You might raise Tensors containing certain powers, for example when calculating likelihoods.

8.14.2 Important Notes

1. Only applies to element-wise contexts. It's not matrix multiplication, even if the tensor is 2D.
2. The scalar can be any real number, including negatives and fractions, as long as the base tensor values support the operation
3. Differentiability, they are safe to use in gradient based optimization.
4. Usage of non-integer scalar, zero or negative tensor values can cause trouble, depending on context.
5. Non-linear Transformation, since powers are Non-linear operations.

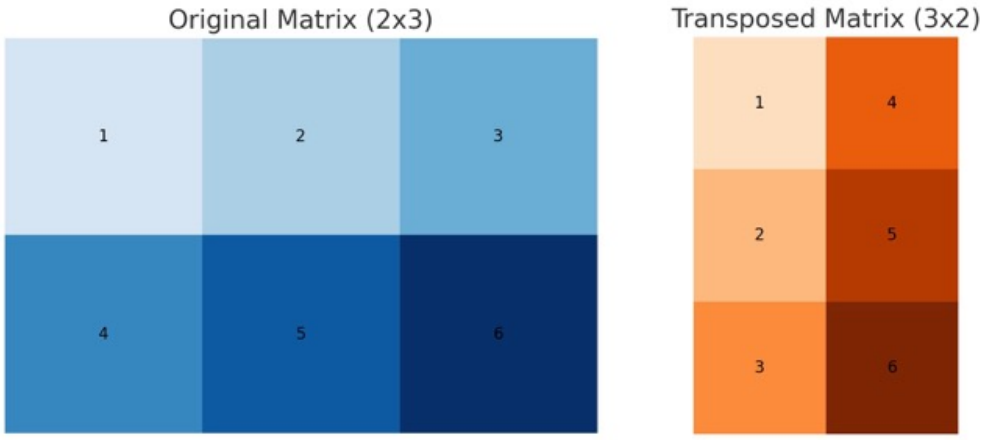

Figure 8: Diagram of a Tensor with Transposed applied, demonstrating the rotation.

9 Tensor Shaping and Boardcasting Operations

In Machine Learning, Tensors often need to be transformed to fit the architecture of a model. This section describes key shaping operations that enable compatibility across layers and models

9.1 Transpose

Transpose allows for Boardcasting, this allows Tensors of different shapes to become Matrix-wise compatible. Transpose solves the problem by rotating a Matrix. For example TensorA(x5, y1), would be rotated to match TenseorB(x1, y1).

```
Function Transpose(Tensor) -> Tensor:
  rows = Tensor.rows
  cols = Tensor.cols

  Transposed_Tensor = Empty_Tensor(cols, rows)

  for col in rows:
  for row in cols:
  Transposed_Tensor[row][col] = Tensor[row][col]

return Transposed_Tensor
```

Listing 11: Formulaic Pythonic Code for Flatten

This is especially useful when aligning shapes for operations such as matrix multiplication.

9.2 Padding or Padding Zeros

If Tensors are not the correct size, padding can be applied to adjust their dimensions. Padding typically works in two ways: either by adding zero values to fill missing rows or columns, or by trimming excess rows or columns to match the desired shape.

```
Function Padding(Tensor, Shape, Zero_value) -> Tensor:
  for row in Shape.x:
    for col in Shape.y:
      if Tensor.Shape.x < row:
        Tensor.append(empty_array)
      if Tensor.Shape.y < col:
        Tensor.append(Zero_value)
```

Listing 12: Formulaic Pythonic Code for Padding

9.3 Flatten

In Machine Learning, especially in neural networks, Flatten is used to convert multi-dimensional data (like 2D images or 3D tensors) into a 1D vector. This is essential because fully connected (dense) layers can only process 1D input. Flatten acts as a bridge between feature extraction layers (like Convolution or Pooling layers) and the classification or output layers, allowing the network to make predictions from complex data structures.

```
Function Flatten(Tensor) -> Tensor:
  flattened_tensor = empty_tensor // initialize
  flattened_tensor.append(empty_array)

  for row in Tensor:
    for col in row:
      flattened_tensor[0].append(col)
```

Listing 13: Formulaic Pythonic Code for Flatten

Flattening preserves element order row-wise, which is important in models.

9.4 Dimensional Products

This function computes the total number of elements implied by a shape array. It's commonly used to validate reshape operations.

```
Function Product(Shape_Array) -> Integer:
    result = 1
    for dim in Shape_Array:
        result *= dim
    return result
```

Listing 14: Formulaic Pythonic Code for Product

9.5 Reshape/Resize

In certain stages of a model's architecture, a Tensor may not have the desired shape for operations like Matrix Multiplication. In such cases, it is often necessary to reshape or resize the Tensor to match the required dimensions of another, ensuring compatibility for subsequent computations.

```
Function Reshape(Tensor, New_Shape) -> Tensor:
  // Flatten the original Tensor into a 1D array
  flat_data = Flatten(Tensor)

  // Check if the number of elements matches
  if Flat_data.Shape.x != Product(New_Shape):
    Throw Error("Shape mismatch: cannot reshape")

  // Build the reshaped Tensor
  Reshaped_Tensor = Empty_Tensor(New_Shape)
  index = 0

  for row in New_Shape.x:
      for col in New_Shape.y:
          Reshaped_Tensor[row][col] = Flat_Data[index]
          index += 1

  return Reshaped_Tensor
```

Listing 15: Formulaic Pythonic Code for Reshape/Resize

Reshape is only safe if the number of elements stays the same; otherwise, it risks runtime errors.

10 Simple Math for Machine Learning

At first glance, one might assume that mastering Machine Learning requires deep knowledge of calculus. In reality, only a few key mathematical concepts are needed to begin practicing effectively. This section introduces those essential ideas in a clear and approachable way.

10.1 Calculus

A language of math for abstract ideas, similar to code, but before modern computing. Most Machine Learning algorithms have formulas, that are described using Calculus but can be translated into Formulaic code. While Calculus is useful for developing such algorithmic ideas, it does not need to be understood to code such ideas.

10.2 Non-linearity

Non-linearity in bases, means that the output which is shown on the graph is not prefectly straight, it should look like the curve of a Polynomial. In math this means that our output should not match it's input, this creates Non-linearity.

10.3 Eulars Number

Eular's Number is the upper limit of a summation or a product. Euler's number is all about infinite limits, either a summation of reciprocals of factorials, or a compounding limit.

```
function calculate_e(terms):
  e = 0
  factorial = 1

  for n from 0 to terms - 1:
      if n > 0:
          factorial = factorial * n
      e = e + (1 / factorial)

  return e
```

Listing 16: Formulaic Pythonic Code for Eulars Number

Keep in mind, most coding languages already have Eular's number built in to std(Standard Library Namespace).

10.4 Mean of Tensor

In Machine Learning, one might wish to measure the tendency of numbers in a dataset, in this case a Tensors data. This can be achieved by adding up each element and then dividing by the number of elements.

```
function Tensor_Mean(Tensor) -> f32:
  data_sum = 0 //initialize
  data_points = Tensor.Shape.x * Tensor.Shape.y // getting the product
  for row in Tensor:
    for col in row:
      data_sum += col
  return data_sum / data_points
```

Listing 17: Formulaic Pythonic Code for Tensor Mean Number

10.5 Median of a Tensor

In some cases, it is useful to compute the central value of a tensor using the median operation. The median is a statistical measure that represents the middle value of a sorted dataset. To compute the median, one must first flatten and sort the tensor. If the total number of elements n is odd, the median is the value at position $(n+1)/2$. If n is even, the median is the average of the two central elements, located at positions

$n/2$ and $(n/2)+1$. This operation provides a robust measure of central tendency, especially in the presence of outliers.

```
function Tensor_Median(Tensor) -> f32:

  cols = Tensor.Shape.x
  rows = Tensor.Shape.y

  if rows.is_odd:
    row_index = (rows + 1) / 2

    if cols.is_odd:
      col_index = (cols + 1) / 2

      return Tensor[row_index][col_index]

    if cols.is_even:
      half_element = cols / 2
      one_over_half_element = (cols / 2) + 1

      return average(Tensor[row_index][half_element], Tensor[row_Index][
    one_over_half_element])

  if rows.is_even:
    half_row_index = rows / 2
    one_over_half_row_index = (rows / 2) + 1

    if cols.is_even:
      half_element = cols / 2
      one_over_half_element = (cols / 2) + 1

      return average(Tensor[half_row_index][half_element], Tensor[one_over_half_row_index][
    one_over_half_element])

    if cols.is_odd:
      col_index = (cols + 1) / 2

      return Tensor[row_index][col_index]
```

Listing 18: Formulaic Pythonic Code for Tensor Median Number

10.6 Absolute Tensor

In Machine Learning, it is often useful to apply the absolute value operation to every element within a tensor. Taking the absolute value means converting any negative numbers into their positive versions, while leaving positive numbers unchanged. This operation ensures that all elements of the tensor are non-negative.

```
function Tensor_Abosulute(Tensor) -> Tensor
  for row in Tensor:
    for col in row:
      col = col.abs()
```

Listing 19: Formulaic Pythonic Code for Tensor Absoulute

10.7 Tensor Standard Deviation (std)

To gain a better statistical understanding of a tensor, you can calculate its standard deviation (std). Standard deviation measures how much the values in the tensor vary around the mean. A higher standard deviation means the data is more spread out; a lower standard deviation means the data is more tightly clustered.

```
function std(Tensor) -> f32
  variance = 0 //initialize
  flattened_self = Tensor.flatten()
  n = flattened_self.Shape.x

```

```
  if n == 0:
    return 0
  else:
    mean = Tensor.sum() / n
    squared_differences = Empty_Tensor

    for x in flattened_self.shape.y:
      squared_differences.push((x - mean).powered(2))

    // Returning Variance
    if sample:
      variance = sum(squared_differences) / (n - 1)
    else:
      variance = sum(squared_differences) / n

    return variance
```

Listing 20: Formulaic Pythonic Code for Tensor std

10.8 Hyperparmeters

Parameters/Setting in a Model/Neural Network

Model hyperparameters:

1. Number of layers: (in a neural network)
2. Number of neurons per layer
3. Activation functions (e.g., ReLU, sigmoid)

Training hyperparameters:

1. Learning rate: how big each update step is during training
2. Batch size: how many samples are used per training step
3. Epochs: how many times the model sees the full dataset
4. Optimizer: (SGD, Adam, RMSprop)

Regularization hyperparameters:

1. Adding basic non-linearity
2. Dropout rate: percentage of neurons randomly ignored
3. L1/L2 regularization coefficients: penalize large weights

Tuning hyperparameters: optimized using

1. Grid search: try all combinations
2. Random search: try random combinations
3. Bayesian optimization, Hyperband, Optuna, etc.

10.9 Function Hyperparameters

Some mathematical functions require extra variables, known as hyperparameters, to control their behavior. While these often have recommended default values, they can be tuned to better fit the needs of a specific problem. For example, the hyperparameter beta1 in the Adam optimizer controls how much past gradients are averaged during training.

10.10 Derivatives

In short, a derivative measures how much something changes as something else changes.

For example, in symbols:

- If we have the function $f(x) = x^2$,
- its derivative is written as $f'(x) = 2x$.

This means that at $x = 3$, the function is changing at a rate of $2 \times 3 = 6$ units per 1 unit of x.

10.11 Epsilon

Epsilon is a very long and small number that one can use to act as a threshold on algorithms, to see where is a tolerable stopping point. For example Epsilon can also be a threshold, when a value gets smaller than epsilon, you can stop iterating. **Do not confuse with Eular's number as they are used quite differently.**

10.12 Geometric symbols and constants in Machine Learning

In machine learning, (pi) doesn't come up as often as other constants like e or special values in linear algebra, but it does sneak into several areas, especially when things get mathematical. It is used for statistical analysis in Machine Learning. Things like Probability and Statistics, and Backpropagation with Softmax + Cross Entropy, etc.

Theta while it may sound preplexing to have degrees in Machine Learning. Theta serves an important role getting the angle of the slope, with respect to it's gradient of change, with is important in Optimizers like Adam's Optimizer.

10.13 erf

Some Activation functions, Optimizers, and other algorithmic functions use a mathematical function which is called Error Function(erf). In short, the erf function is a way to calculate the likelihood that a value is less than or equal to a specific point on a normal distribution curve, which makes it very important to Machine Learning.

```
function erf(x) -> f32
// Constants
pi = std.pi // pi from your coding languages std.
sqrt_pi = sqrt(pi)

// Number of intervals for numerical integration (higher = more precision)
N = 1000
dx = x / N

// Initialize result of integration
result = 0

// Numerical integration using the trapezoidal rule
for i in N:
    t1 = i * dx
    t2 = (i + 1) * d

    result += (exp(-t1.raised(2)) + exp(-t2.raised(2))) * (t2 - t1) / 2

// Return the scaled value based on erf definition
return (2 / sqrt_pi) * result
```

Listing 21: Formulaic Pythonic Code for erf

Keep in mind some programming languages might already have a erf function for your element's datatype.

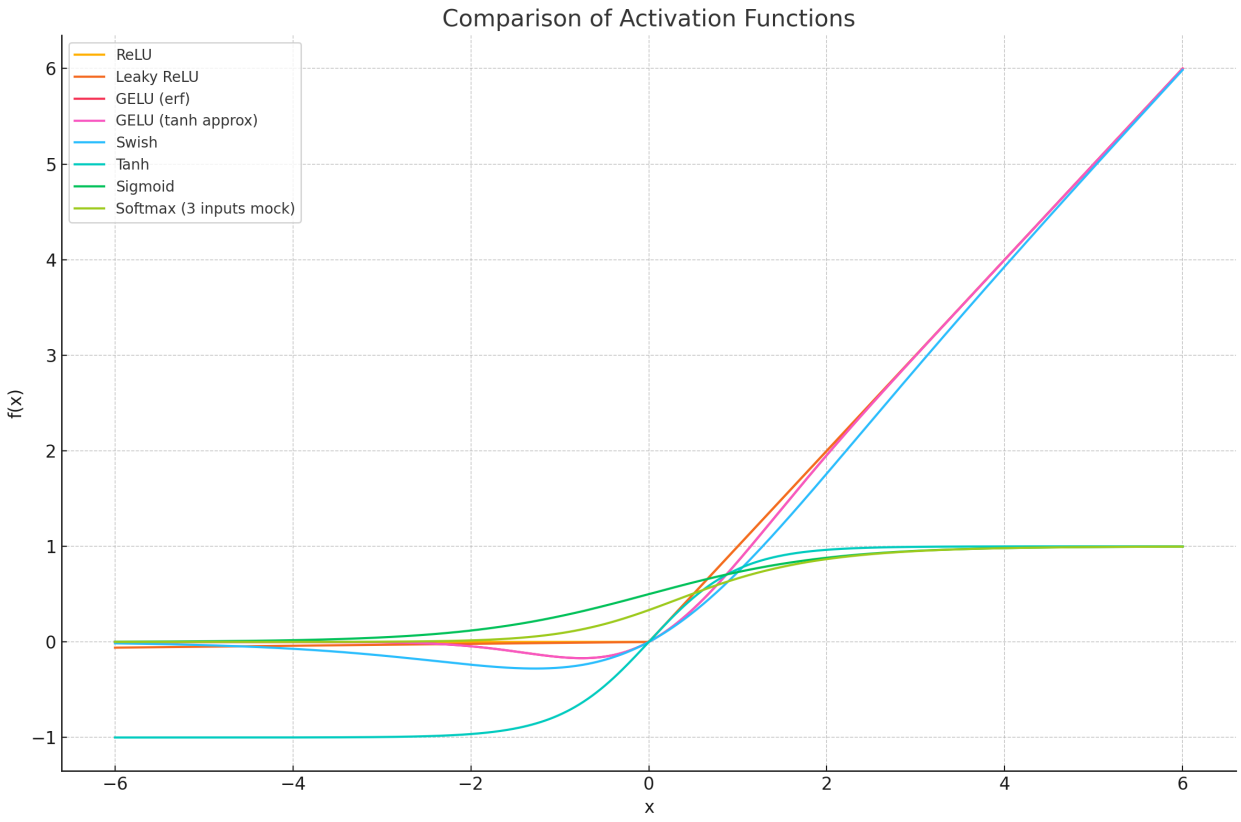

Figure 9: Comparison of common activation functions (ReLU, Leaky ReLU, GELU, Swish, Tanh, Sigmoid, Softmax)

11 Activation Functions

Meaning of Activation How much it take for the neuron to activate based on the level of signal.

Purpose of Activation Functions In the brain, neurons are connected to each other, and whether one fires to the next depends on the strength of its signal, plus a lot of complex chemistry and physics happening at the synapse (tiny gap between a connection of two neurons).

To capture the complexity of chemistry and physics that is too computational hard to run, and hard to measure fully, we have activation functions which mimic this purpose, and decide the strength of all the neurons.

They also help us connect patterns and relationships (through the strengths, the actual weights in your tensor) and through non-linearity, where your data doesn't just flow straight to the output. It gets shaped by something more complex: your activation function.

Section Ordering This order begins with foundational activation functions, gradually introducing smoother, more complex variants, and ends with GELU, which is commonly used in Transformer-based architectures.

Activation	**Range**	**Non-linearity**	**Key Notes**
ReLU	$[0, \infty)$	Sharp	Very simple, fast, sparse activation
Leaky ReLU	$(-\infty, \infty)$	Sharp	Fixes dying ReLU by allowing small negative slope
Swish	$(-\infty, \infty)$	Smooth	Self-gated, better at small gradients
GELU (erf)	$(-\infty, \infty)$	Smooth	Probabilistic activation, used in transformers
tanh	$(-1, 1)$	Smooth	Zero-centered, good for recurrent networks
GELU (tanh)	$(-\infty, \infty)$	Smooth	Faster approximation of GELU

Table 1: Comparison of Activation Functions

Activation	**Additional Insights**
ReLU	Can suffer from dying neurons; most common in CNNs (Convolutional Neural Networks). No activation for negative input.
Leaky ReLU	Common negative slope is $\alpha = 0.01$. Good compromise between pure ReLU and more complex activations.
Swish	Invented by Google researchers. Improves performance on deep networks. Non-monotonic.
GELU (erf)	Behaves like ReLU for large positive values but smooth near zero, allowing better gradient flow.
tanh	Derivative saturates at edges, causing vanishing gradients if used improperly.
GELU (tanh)	Offers nearly identical performance to original GELU but much faster during inference.

Table 2: Further Notes on Activation Functions

Activation	**Usage Tip or Warning**
ReLU	Good default choice; however, monitor for dead neurons especially with high learning rates.
Leaky ReLU	Try when training gets stuck with ReLU. Reduces risk of neurons dying during training.
Swish	Slightly more expensive computationally, but improves accuracy in many tasks.
GELU (erf)	Best for transformer-based models like BERT and GPT-like architectures.
tanh	Preferred over sigmoid when zero-centered output is needed. Often used in RNNs.
GELU (tanh)	Recommended if hardware constraints make the original GELU too slow.

Table 3: Tips and Warnings for Activation Function Usage

11.1 ReLU

ReLU, in simple terms, means: The Function that flattens all the negative parts of your data to zero, so the output only keeps the positive stuff. That way, it adds some shape and transformation to your data instead of just copying it.

```
Function ReLU(Tensor) -> Tensor:
  for row in Tensor:
    for col in row:
      if col > 0:
        col = 0
      else:
        col // Element stays the same number.
```

Listing 22: Formulaic Pythonic Code for ReLU

11.1.1 What is ReLU in better depth

It is an element-wise operation, meaning it does not need to see your whole tensor at once, it goes row by col, going by each element in the matrix. For each element, if it is negative (less than 0), it is set to zero. Similar to max() in math.

11.1.2 Use Cases

1. Adding basic non-linearity
2. Efficient to compute, hardware side
3. Less Vanishing Gradients (your data becoming such a large negative number, the computers can't store it)
4. Less extra data, also known as (Sparsity)
5. Keep your network positive, by removing all negatives
6. Faster to train (faster convergence)
7. Basic relationships

11.1.3 Why not use ReLU?

1. If you use ReLU, your model will have trouble with large and more complex relations. So, it depends on the pattern you wish to capture.
2. When capturing (pattern seeking) negative relationships or values.e

11.1.4 Why is it faster to train?

You're trying to capture a simpler relationship, with a very efficient computation formula.

11.1.5 Alternative for ReLU, Leaky ReLU

1. Solves the Negative relationship to keep neurons out of the inactive state.
2. Able to calculate a tiny bit more relationships.

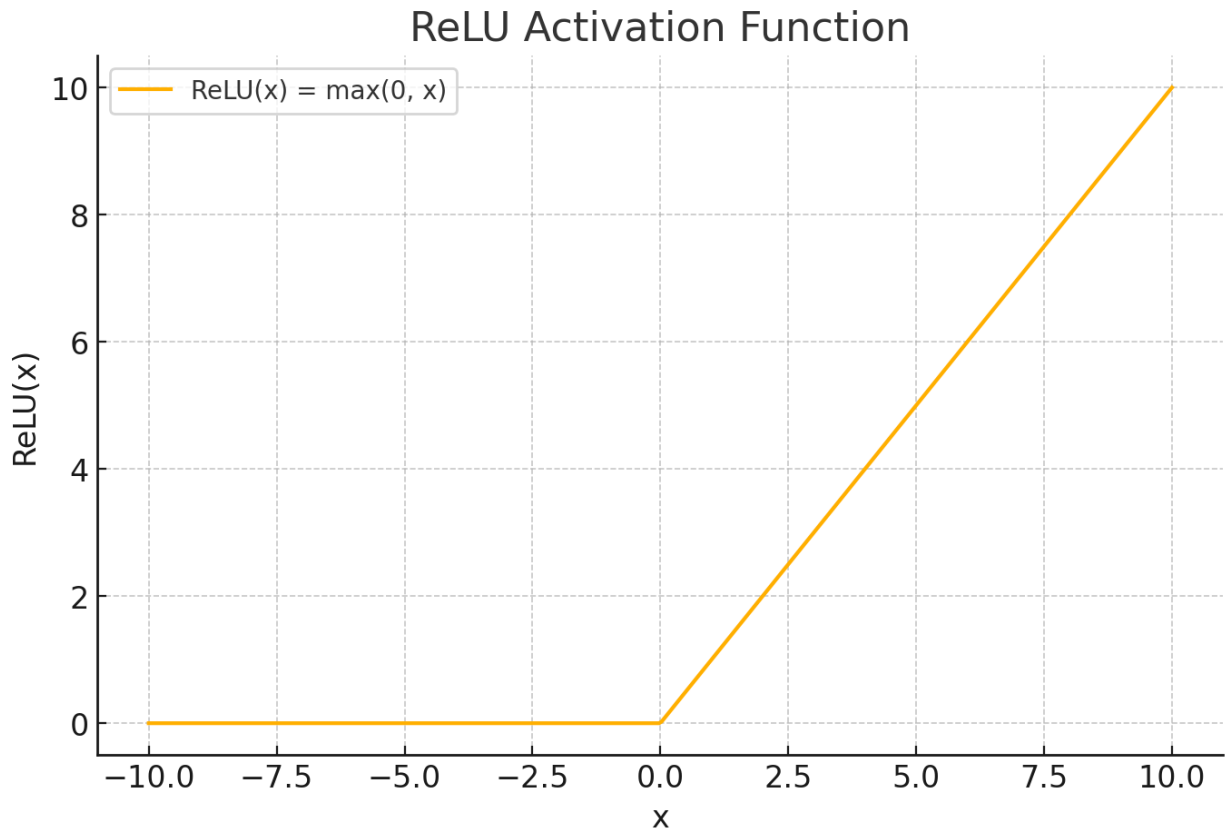

Figure 10: ReLU graphed mainly to show the curve.

11.1.6 ReLU graph

Notice how the graph remains flat and then abruptly shifts at a sharp angle. This discontinuity in slope is a hallmark of a non-smooth function. For example, ReLU exhibits this behavior with a sharp kink at zero. In contrast, smooth activation functions like GELU produce a more gradual polynomial-like curve with continuous derivatives. This smoothness allows the model to capture more nuanced patterns and learn more complex relationships in data, whereas non-smooth functions may limit expressiveness in certain cases.

1. **Slope, less than 0 region:** Remain at 0
2. **Slope, greater than 0 region:** Regular linear increase

11.2 Leaky ReLU (LReLU)

ReLU, in simple terms means: Making your data not look like it came from the inputs, by turning elements less than 0 into the negative slope * x.

```
Function LReLU(Tensor, negative_slope) -> Tensor:

  for row in Tensor:
    for col in row:

      if col > 0:
        col = col
      else:
        col = negative_slope * col
```

Listing 23: Formulaic Pythonic Code for LReLU

11.2.1 What is LReLU in better depth

It is an element wise operation, meaning it does not need to see your whole tensor at once, it goes row by column, going by each element in the matrix. For each element, if it is negative (less than 0), the same element is set to a negative slope of times, which is normally 0.01.

11.2.2 Use Cases

1. Adding basic non-linearity
2. Efficient to compute, hardware side
3. Less Vanishing Gradients (your data becoming such a large negative number, the computers can't store it), then ex, sigmoid, and tanh.
4. Faster to train (faster convergence)
5. Basic relationships
6. Alternative to ReLU in Hidden Layers
7. Getting "consistently zero neurons/data" from Relu
8. Keeping Neurons alive(when weights are 0 they are dead, because of no connection)

11.2.3 Why not use LReLU?

1. If you use LReLU, your model will have trouble, with large and more complex relations. So it depends on the pattern you wish to capture
2. When you don't wish to lose negative values

11.2.4 Why is it faster to train?

You're trying to capture a simpler relationship, with a very efficient to compute formula.

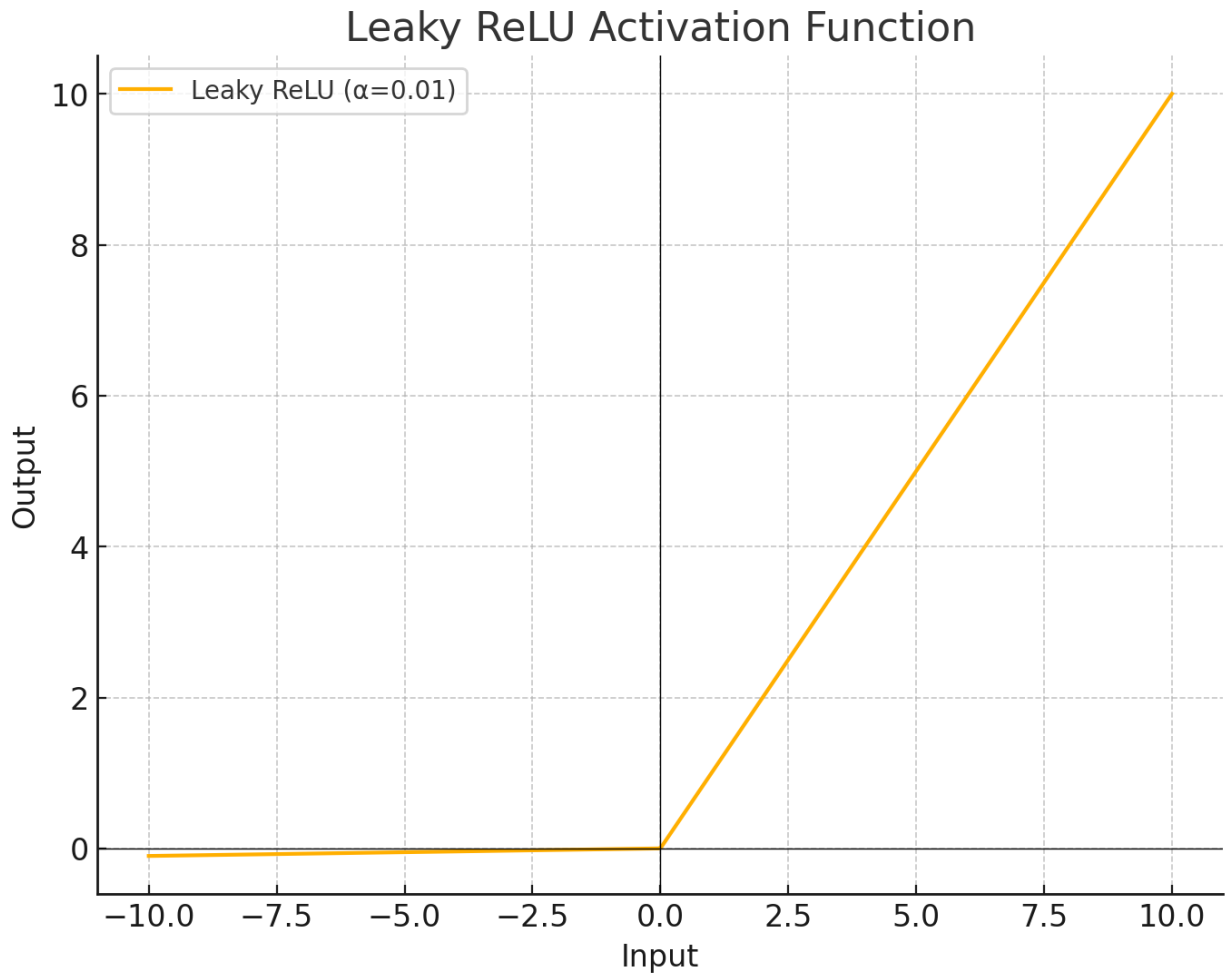

Figure 11: LReLU graphed mainly to show the curve, before 0.

11.2.5 LReLU graph

Notice how the graph no longer stays completely flat for negative inputs. Instead, as values drop below zero, the function "leaks," meaning the output is scaled by a small, fixed negative slope rather than zeroed out. This allows some gradient to flow during training, preventing the neuron from dying completely. After crossing zero, Leaky ReLU behaves like a standard linear function, just like regular ReLU.

1. **Slope, less than 0 region:** Slight downward tilt = "leak"
2. **Slope, greater than to 0 region:** region: Regular linear increase

11.3 Swish

When you need a smooth activation function that can induce non-linearity and learn complex relationships/patterns, it's like a smarter version of ReLU, because it does not cut off negative numbers, but gently pushes them down. Keep in mind, this function requires the use of Euler's number.

```
Function Swish(Tensor) -> Tensor:
  for row in Tensor:
    for col in row:
      col * (1 / (1 + (-x).exp()))
```

Listing 24: Formulaic Pythonic Code for Swish

You can also make use of sigmoid. As it is the same compution, in math.

```
Function Swish(Tensor) -> Tensor:
  for row in Tensor:
    for col in row:
      col * sigmoid(col)
```

Listing 25: Formulaic Pythonic Code for Swish

11.3.1 What is Swish in better depth

It is an element wise operation, meaning it does not need to see your whole tensor at once, it goes row by column, going by each element in the matrix. For each element, we times the index by itself times sigmoid, this is done to keep both positive and small negative value, unlike ReLU.

11.3.2 Use Cases

1. Adding non-linearity.
2. Basic relationships/patterns.
3. Computer vision.
4. Less likely to have exploding Gradients than ReLU.
5. Smooth, unlike ReLU.
6. Allows Non-zero for negatives.
7. Likely outperforms ReLU.
8. CNNs (very deep networks).
9. Transformers.
10. Better regression and classification than ReLU.
11. Replacement to Vanishing Gradient prone Sigmoid.
12. Performance over raw speed.

11.3.3 Why not use Swish?

1. Computational Cost.
2. ReLU is already quite good: Predictable, fast, easy debugging, and cheap to compute.
3. There are other ways to fix Gradient Flow, such as Batch Norm.
4. Swish is less standardized, so it's not universally supported in all frameworks.
5. Slower to train.

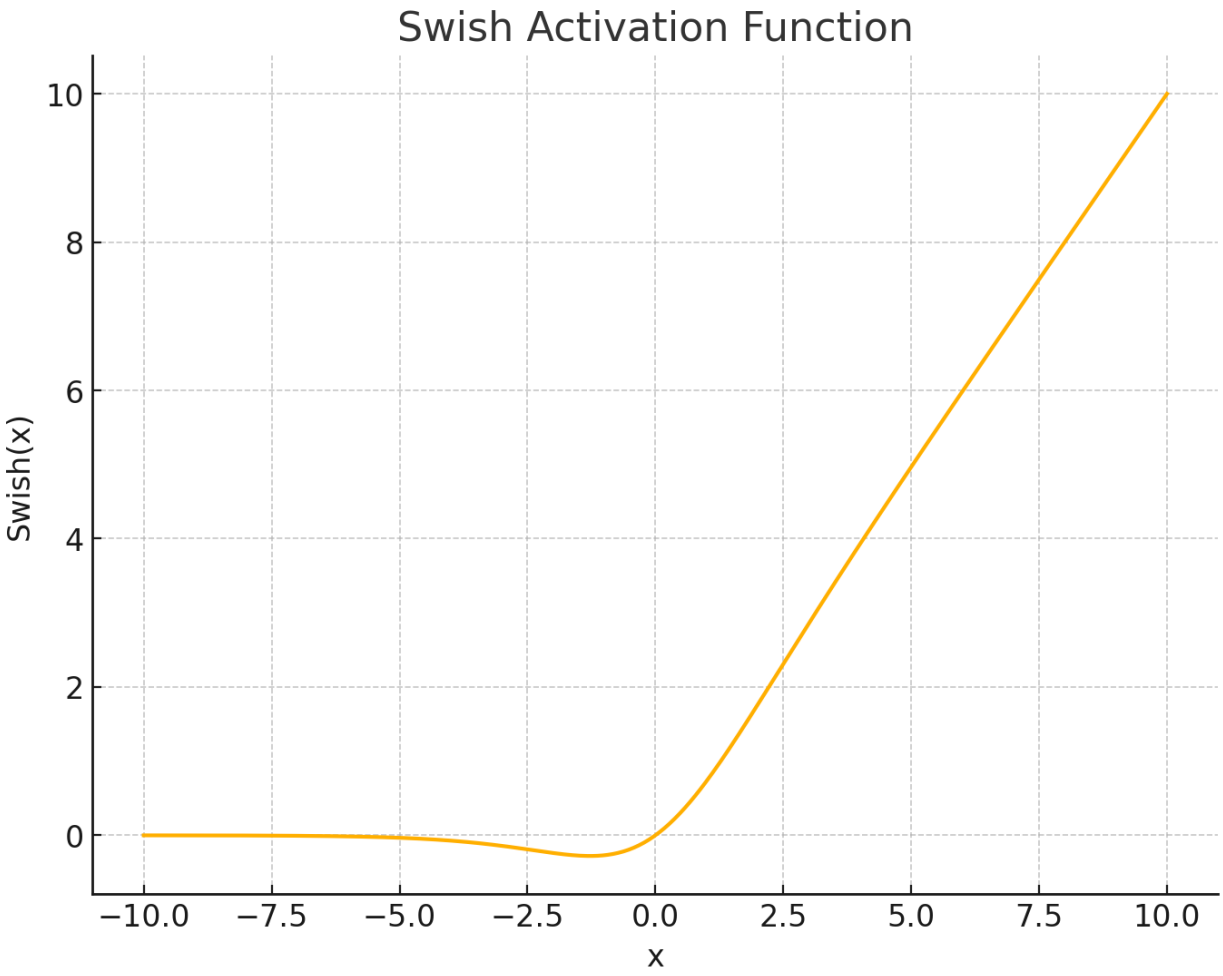

Figure 12: Swish graphed mainly to show the curve.

11.3.4 Swish Graph

The Swish activation function combines input x with the sigmoid of x, resulting in a smooth curve that keeps negative values (but shrinks them) and grows positively with positive inputs.

1. **Slope, less than 0 region:** When $x < 0$, the Swish function has a small positive slope. Unlike ReLU, which drops everything below zero to zero, Swish keeps negative values, just reduces them gently. This helps the network learn patterns in negative regions and improves gradient flow during backpropagation.

2. **Slope, greater than or equal to 0 region:** For $x \geq 0$, Swish behaves almost like a linear function. The output increases smoothly, and the slope approaches 1 as x becomes larger. This means large positive inputs pass through with minimal resistance, like ReLU, but with smoother transitions.

11.4 Sigmoid

Making your data not look like it came from the inputs, by applying math, to make numbers range from 0 and 1. This makes output classification easier and can show model confidence. All of this is done element-wise.

```
Function Sigmoid(Tensor) -> Tensor:
  for row in Tensor:
    for col in row:
        col = 1 / (1 + exp(-col))
```

Listing 26: Formulaic Pythonic Code for Sigmoid

11.4.1 What is Sigmoid in better depth

It is an element wise operation, meaning it does not need to see your whole tensor at once, it goes row by column, going by each element in the matrix. For each element, we apply the exponential function, with some other numbers, and simple operations that add more non-linearity, for smooth and more complex relationships.

11.4.2 Use Cases

1. Adding smooth non-linearity.
2. Smooth and differentiable, so we can use it in Gradient Descent.
3. Classification problems/relationships.
4. Ability to see Model confidence.
5. Computer vision type problems.
6. Making your output layer into a more understandable confidence or class answer. Also known as (Final layer of binary classification), due to its mapping from 0-1

11.4.3 Why not use Sigmoid?

1. Much more computationally slower than ReLU.
2. When capturing (pattern seeking) Negative relationships or values.
3. Stray away from usage in hidden layers (Layers between your input and output), as it will lead to Vanishing Gradients. I personally suffered with this myself.
4. Sigmoid is always positive, meaning it is harder for the network to converge efficiently.
5. For hidden layers, defer to using other activation functions, such as ReLU, GELU, or tanh.

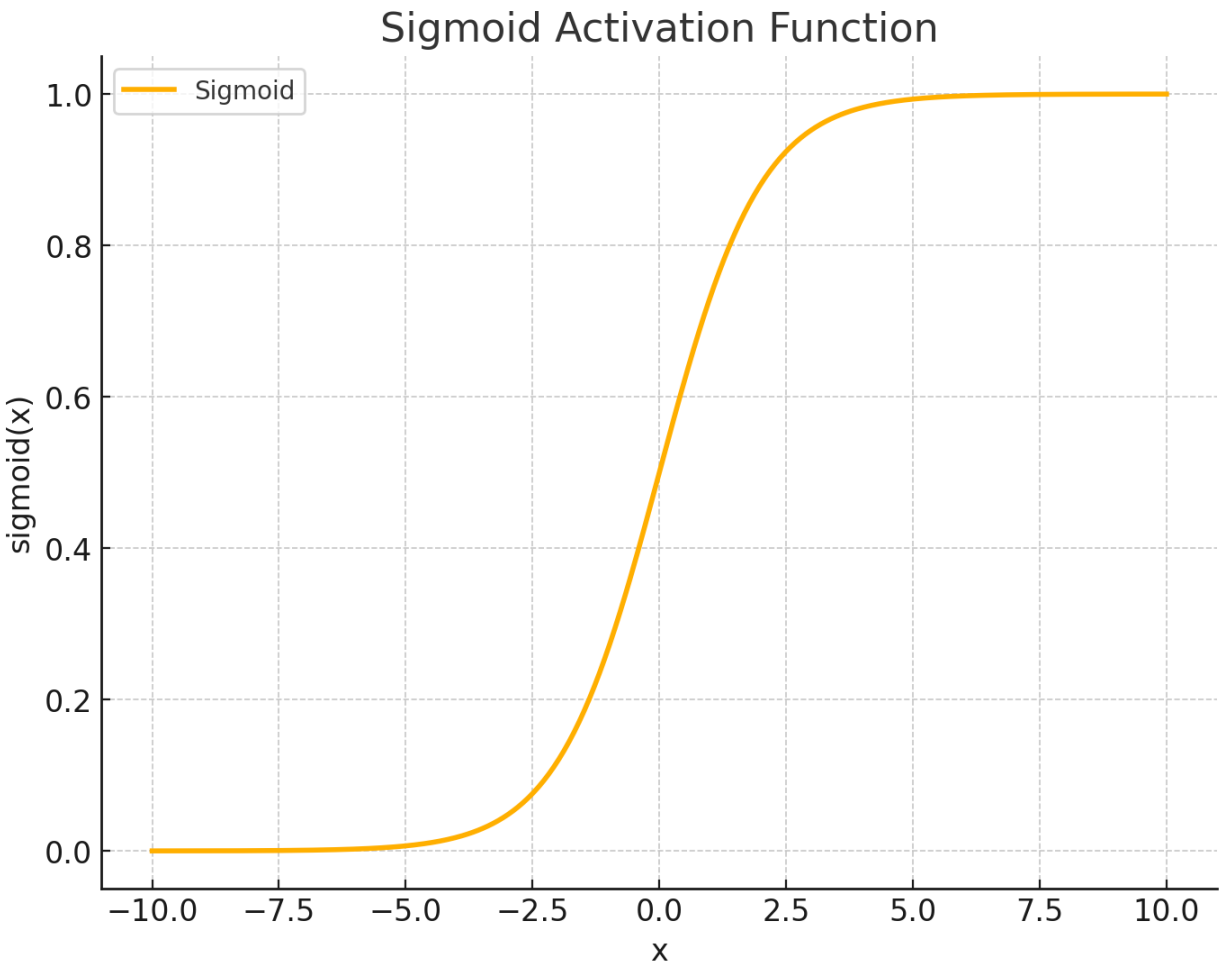

Figure 13: Sigmoid graphed, keep in mind that it may look like y is 0 or 1, but it is in reality really close but never so.

11.4.4 How can I use this binary classification on my output

Once your network reaches the output layer during backpropagation, you can apply the sigmoid function. This squashes the output to a value between 0 and 1. You can interpret this number either as a confidence score or a predicted class.

- For example, if your target is 0 and your model outputs 0.1, you can tell the model is fairly confident the input leans toward class 0.
- This is super useful in real tasks too. Imagine you're training a model to classify how bright a pixel is:

1. for example
2. 1: means white
3. 0: means black

If your input is a gray pixel, the model might output something like 0.9. After applying sigmoid, that output tells you the pixel is closer to white than black, but not entirely white either, more like "very light gray."

11.5 GELU(erf)

Instead of letting data pass through directly, GELU uses the erf function to gently curve the output. This smooth shape adds non linearity, meaning the output becomes less predictable and more expressive, helping the network better learn tricky patterns in your data.

```
Function GELU_erf(Tensor) -> Tensor:
  for row in Tensor:
    for col in row:
        col = 0.5 * col * (1 + erf(col / sqrt(2)))
```

Listing 27: Formulaic Pythonic Code for GELU(erf)

11.5.1 What is GELU(erf) in better depth

It is an element wise operation, meaning it does not need to see your whole tensor at once, it goes row by col, going by each element in the matrix.

For each element it applies a mathmatical function called erf, which is also just done in the Formulaic Code. If you are interested in eft, check inside erf.

11.5.2 Use Cases

1. Adding smooth non-linearity (balanced between ReLU's sparsity, and Sigmoid's smoothness).
2. Transformer-based models.
3. helps huge models train faster, but not small.
4. Making more stable and accurate Deep networks.
5. When you want stable learning.
6. Building big models.
7. Able to compute complex relationships.
8. Good performance to relationship ratio.

11.5.3 Why not use GELU(erf)?

1. Higher compute than ReLU or LeakyReLU.
2. Heavier on edge devices, or mobile inference.
3. Longer training times if not optimized.
4. May Overfit slightly more (capturing the whole data, and not simpler relationships).
5. It depends on the task.
6. Slower than GELU(tanh).

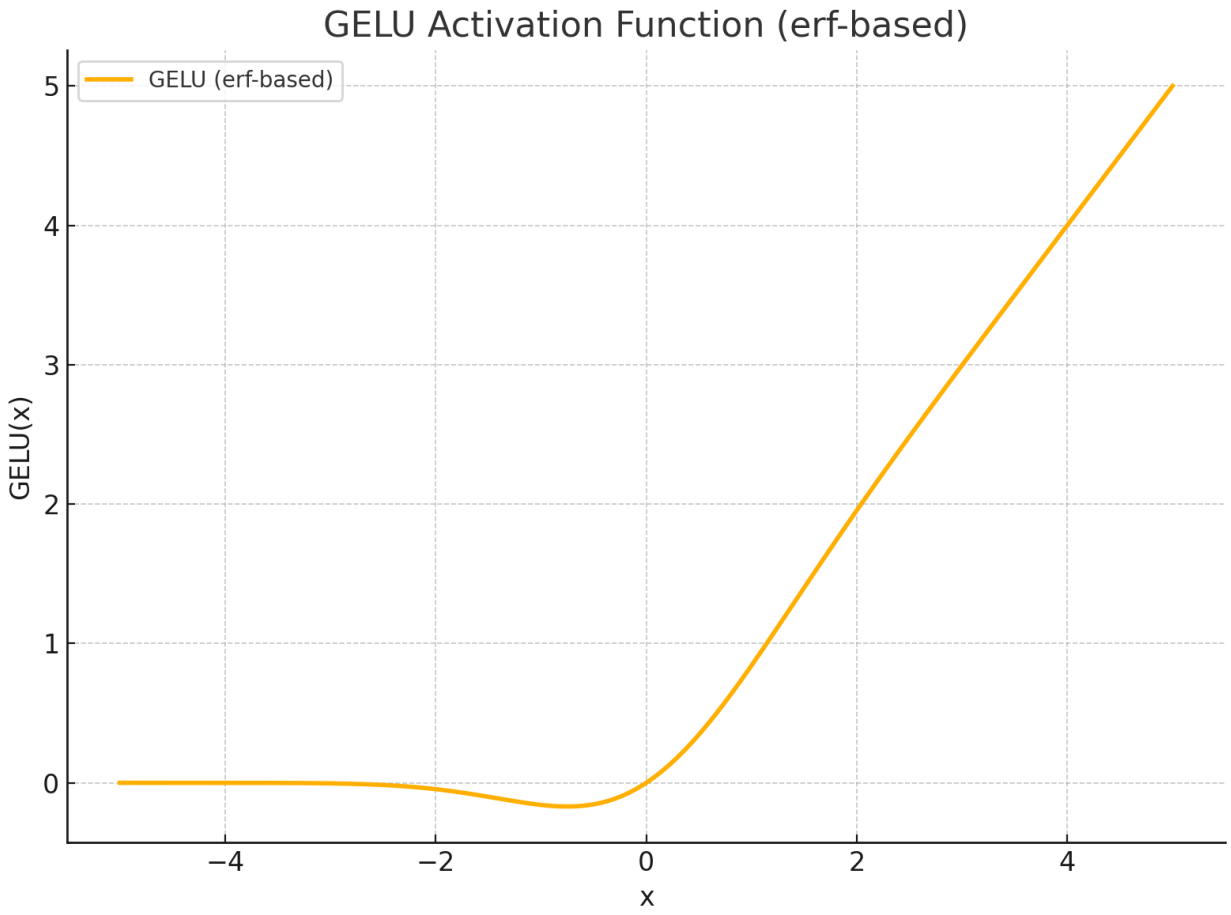

Figure 14: GELU(erf) graphed mainly to show the curve.

11.5.4 GELU(`erf`) Graph

The `erf` based version of GELU is the original formulation, using the Gaussian error function for smoother, more theoretically grounded activation. Here's how the slope behaves in different regions:

1. **Slope, less than 0 region:** For negative inputs, the output decays gradually toward zero in a soft, non-linear curve. Unlike hard threshold functions, this smooth suppression helps retain gradient flow even for weak negative signals, improving learning stability.

2. **Slope, greater than or equal to 0 region:** Positive inputs smoothly increase toward the identity line, with a natural curvature. This part of the function behaves almost linearly for large positive values, but with better gradient properties than ReLU due to its differentiability everywhere.

11.6 tanh

tanh, short for "hyperbolic tangent," is a smooth, S shaped curve that transforms input values into a range between -1 and 1. Imagine you have a number line, and you're trying to "squish" all your outputs into a tight range while keeping their signs and relative magnitudes. tanh does exactly that negative values stay negative, positive ones stay positive, and the transition from -1 to 1 is smooth and non linear.

```
Function tanh(Tensor) -> Tensor:
  e = Eulars_number # constant
  for row in Tensor:
    for col in row:
        tanh_numerator = e.powered_to(col) - e.powered_to(-col)
        tanh_denominator = e.powered_to(col) + e.powered_to(-col)
        tanh = tanh_numerator / tanh_denominater
```

Listing 28: Formulaic Pythonic Code for tanh

11.6.1 What is tanh in better depth

It is an element-wise operation, meaning that it does not need to see your whole tensor at once; it goes row by column, going by each element in the matrix. For each element, we perform tanh in math.

11.6.2 Use Cases

1. Adding non-linearity.
2. Older neural nets, where outputs need to swing negative.
3. Zero-centered, unlike functions like Sigmoid, that are always positive.
4. RNNs (Recurrent Neural Networks), helps during time step in some optimizers.

11.6.3 Why not use GELU(erf)?

1. Vanishing gradients if inputs are saturated (very positive or very negative), you can try to avoid this via Normalizing the inputs.
2. Can have slower learning from Vanishing gradients, so ReLU and GELU often replace it.
3. Slower Convergence than ReLU based one's.
4. Don't use in Deep CNNs or Transformers, ReLU, GELU, and Swish usually work well there.
5. It depends on the task.
6. Where speed or newer functions matters.

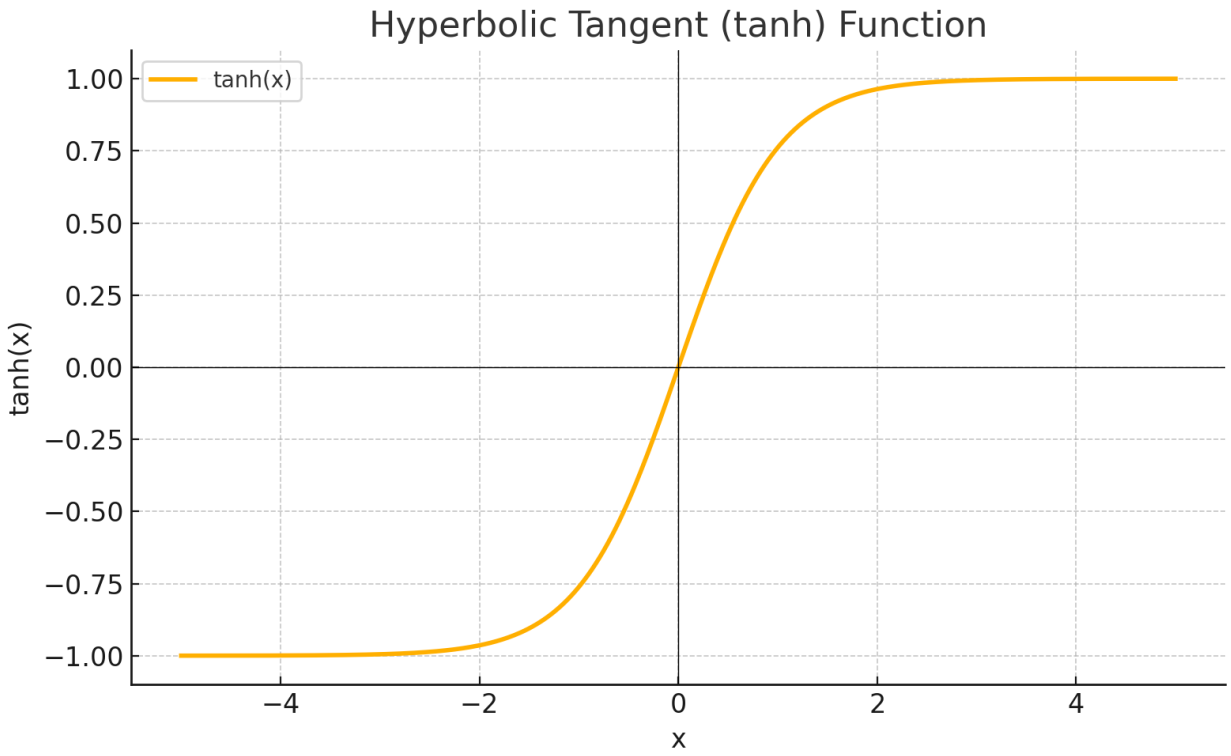

Figure 15: tanh graphed mainly to show the curve.

11.6.4 tanh graph

The graph of `tanh` is an S shaped curve (also known as a sigmoid curve), but unlike the standard sigmoid, it's centered at zero. This helps the network learn faster and avoid biased outputs.

1. **Slope in the region less than 0:** As input values become more negative, the output of `tanh` approaches -1. The slope is still positive in this region but gets smaller (flatter) the further left you go. This means very negative inputs don't have much effect, `tanh` "saturates" at -1.

2. **Slope in the region greater than or equal to 0:** For small positive values, the slope is steep, meaning the function is sensitive to changes in input. As inputs become more positive, the slope again flattens, and the output approaches +1. Like on the left side, this saturation helps clip extreme values while preserving the sign and non linearity.

11.7 GELU(tanh)

GELU(tanh) is a quicker version of GELU(erf) that uses the `tanh` function to create a smooth curve. This adds non-linearity, meaning the function doesn't just copy the input. Instead, it transforms it in a way that helps neural networks learn more complex and subtle patterns.

```
Function GELU_tanh(Tensor) -> Tensor:
  # Hyperparmeters
  e = eulars_number
  a1 = 0.044715 # GELU formula constant

  for row in Tensor:
    for col in row:
        tanh_numerator = e.powered_to(col) - e.powered_to(-col)
        tanh_denominator = e.powered_to(col) + e.powered_to(-col)
        tanh = tanh_numerator / tanh_denominater
        value = 0.5 * col * ((1 + tanh) * (FRAC_2_SQRT_PI * (col + (a1 * col).powered_to(3))
    ));
```

Listing 29: Formulaic Pythonic Code for GELU(tanh)

11.7.1 What is GELU(tanh) in better depth

It is an element wise operation, meaning it does not need to see your whole tensor at once, it goes row by col, going by each element in the matrix. For each element, we preform tanh in math.

11.7.2 Use Cases

1. Adding non-linearity.
2. Older neural nets, where outputs need to swing negative.
3. Zero-centered, unlike functions like Sigmoid, that are always positive.
4. RNNs (Recurrent Neural Networks), helps during time step in some optimizers.

11.7.3 Why not use GELU(tanh)?

1. Vanishing gradients if inputs are saturated (very positive or very negative), you can try to avoid this via Normalizing the inputs.
2. Can have slower learning from Vanishing gradients, so ReLU and GELU often replace it.
3. Slower Convergence than ReLU based one's.
4. Don't use in Deep CNNs or Transformers, ReLU, GELU, and Swish usually work well there.
5. Where speed or newer functions matters.

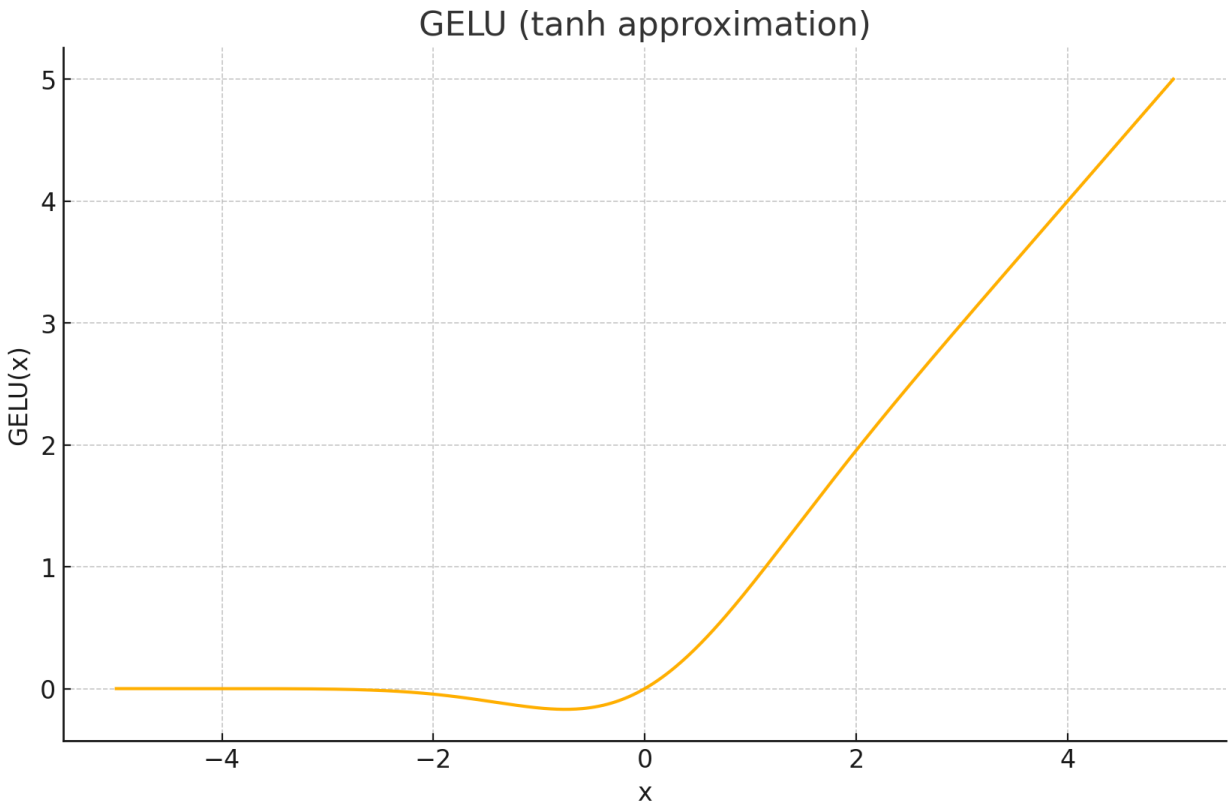

Figure 16: GELU(tanh) graphed mainly to show the curve.

11.7.4 GELU(`tanh`) Graph

The GELU activation function using the `tanh` approximation produces a smooth curve that behaves differently in the negative and positive input regions. Below is a breakdown of how the slope changes across those regions:

1. **Slope, less than 0 region:** In this region, the output gradually tapers toward zero. Unlike ReLU or Leaky ReLU, GELU softly suppresses negative inputs instead of cutting them off or linearly leaking them. This allows the network to retain some weak signals, which can help in learning subtle patterns.

2. **Slope, greater than or equal to 0 region:** For non negative inputs, the output rises in a nonlinear but smooth fashion, closely resembling the identity function for large positive values. This region preserves strong positive signals while maintaining differentiability, which benefits gradient based optimization.

11.8 Softmax

You've got a bunch of numbers, and you want to turn them into probabilities, numbers between 0 and 1, where everything adds up to 1. Softmax takes each number, makes it positive and exaggerated (using exponential math), then scales them all so they sum to 1. This is so You can get human reable chance, on the Output layer.

11.8.1 Step breakdown

All of these steps apply something to your Tensor, and hand that down to the next, like an assembly line.

11.8.2 Steps per row

1. Stabilize (Optionally).
2. Get exponentials.
3. Local sum for each row.
4. Normalize, for human readable Output chance classification.

11.8.3 Stabilize (Optionally)

In short terms, it is done to avoids overflow in exp by subtracting the largest value. It can be done by, Subtracting maximum/max() value of the entire vector/ or row, or last axis of the tensor, to the Tensor. Doing this can prevent large numbers in the input, from overflowing with exp, and going to infitiy, causing Gradient Explosion, especially in Float32, Main common element datatype.

11.8.4 Get exponentials

Makes all values positive and highlights the largest.

11.8.5 Local sum for each row

Prepares for the normalization in the next step.

11.8.6 Normalize

Converts into a probability distribution (output readable by humans/models).

```
Function Softmax(Tensor) -> Tensor:
  Result = []  # Final result to return

  for row in Tensor:
      # Step 0: Stabilize (optional) by subtracting max
      max_val = max(row)

      # Step 1: Compute exponentials of adjusted values
      exps = []
      for x in row:
          exp_val = exp(x - max_val)
          exps.append(exp_val)

      # Step 2: Sum the exponentials
      sum_exps = 0
      for value in exps:
          sum_exps += value

      # Step 3: Normalize to get probabilities
      softmax_row = []
      for value in exps:
          prob = value / sum_exps
          softmax_row.append(prob)
```

```
24
25      # Add the row result to the final output
26      Result.append(softmax_row)
27
28  return Result
```

Listing 30: Formulaic Pythonic Code for Softmax

11.8.7 What is Softmax() in better depth

It is a NOT element wise, but matrix wise (all elements in matrix factored together). The reason it is done matrix wise, is because of the probability distribution, showing all the possible outcomes.

11.8.8 Use Cases

1. Turns numbers into confidence scores.
2. When using Cross-Entropy Loss.
3. Probabilistic output = Human-readable decision.
4. Making your output layer into more a understandable confidence or class answer. Aka (Final layer of binary classification), due to it mapping from 0-1.

11.8.9 Why not use Softmax?

1. In binary classification, you usually use sigmoid.
2. When doing regression (predicting real values), softmax doesn't make sense.
3. Never for a general activation function on input or hidden layers.

11.8.10 How Can I Use This Classification on My Output?

Similar to the Sigmoid function, Softmax provides a measure of confidence for each prediction. However, unlike Sigmoid, which is best for binary classification, Softmax allows for multiple classes, assigning a probability to each one. This makes it especially useful when dealing with problems that require mutually exclusive outcomes. The result is a set of human, readable confidence scores, one for each class, which sum to 1.

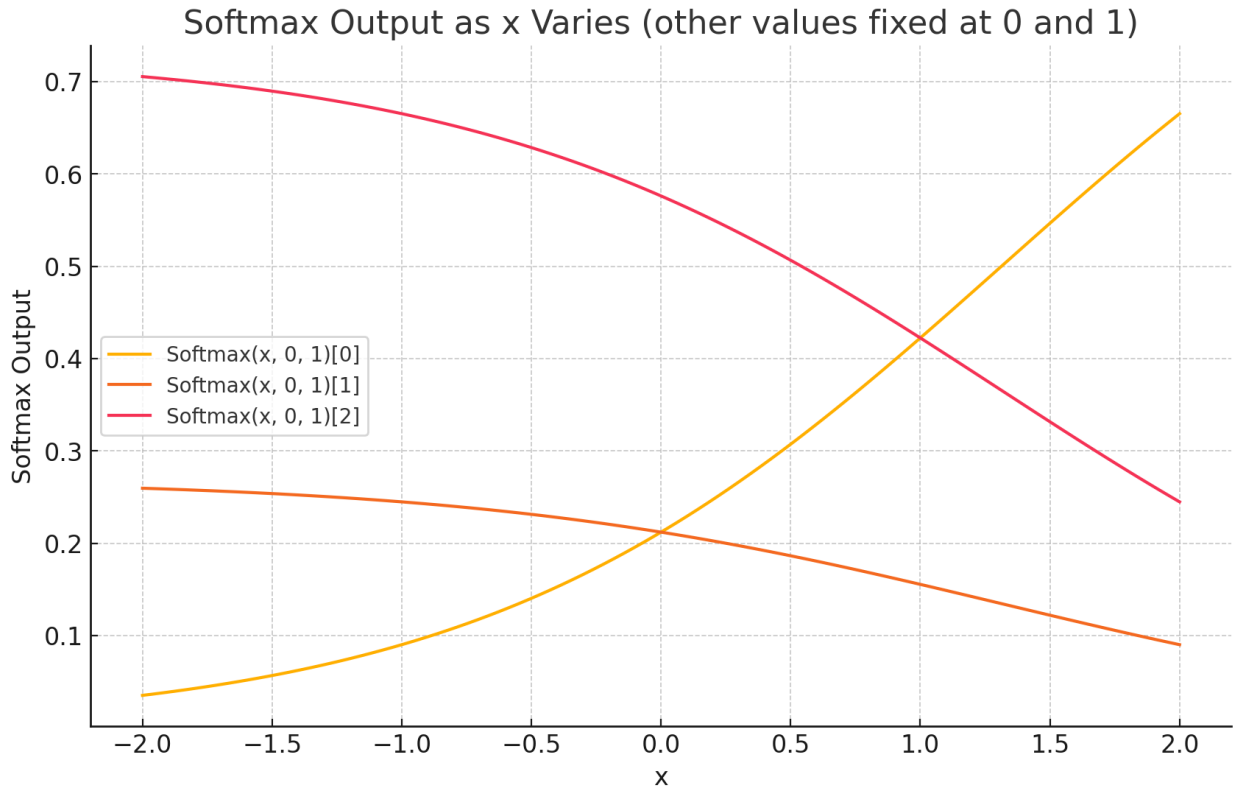

Figure 17: Softmax graphed mainly to show the binary classification with 3 classes.

11.8.11 Softmax Graph

Notice the three distinct curves in the graph, each one represents a different class example. The output for each class is denoted as Y, which reflects the predicted probability of that class. These probabilities always fall between 0 and 1. Conceptually, the Softmax function is similar to the Sigmoid function, but instead of evaluating a single outcome, it distributes probabilities across multiple possible classes.

12 Activation function derivatives

In backpropagation, derivatives of activation functions are applied according to the Calculus Chain Rule. This process adjusts the model's internal weights to minimize the loss. The activation function derivative tells us how sensitive the neuron's output is to changes in its input.

These derivatives assume x is the output of the corresponding activation functions applied to the weights, which is necessary for backpropagation.

12.1 ReLU derivative

When you calculate gradients, if the neuron output was negative or zero, the gradient gets killed (multiplied by 0). If it was positive, it passes through normally (multiplied by 1).

```
Function relu_derivative(x):
  if x > 0:
      return 1
  else:
      return 0
```

Listing 31: Formulaic Pythonic Code for ReLU derivative

12.2 Leaky ReLU derivative

The Leaky ReLU derivative controls how gradients flow during backpropagation. It returns 1 when the input is positive, and a small constant (like 0.01) when the input is negative. Unlike ReLU, which completely blocks gradients for negative inputs, Leaky ReLU allows a small slope even when inputs are negative, helping to prevent neurons from dying (getting stuck at zero) and keeping the network learning more smoothly.

```
Function leaky_relu_derivative(x, alpha):
  if x > 0:
      return 1
  else:
      return alpha
```

Listing 32: Formulaic Pythonic Code for Leaky ReLU derivative

12.3 Swish derivative

Swish derivative controls gradient flow for the Swish activation, which is smoother than ReLU, yet still not as hard to compute as GELU.

```
Function swish_derivative(x):
  s = sigmoid(x)
  return s + x * s * (1 - s)
```

Listing 33: Formulaic Pythonic Code for Swish derivative

Or as a **Sigmoid helper function**

```
Function swish_derivative(x):
  return 1 / (1 + exp(-x))
```

Listing 34: Formulaic Pythonic Code for Swish derivative as Sigmoid helper function

12.4 Sigmoid derivative

The Sigmoid derivative measures how fast the sigmoid curve changes. It's simply the sigmoid output times one minus itself. This keeps gradients small and smooth, but can cause vanishing gradients when it is near 0 or 1.

```
Function Sigmoid_Derivative_From_Output(sigmoid_output):
  return sigmoid_output * (1 - sigmoid_output)
```

Listing 35: Formulaic Pythonic Code for Sigmoid derivative

12.5 GELU(erf) derivative

The GELU (erf) derivative controls gradient flow for the smooth GELU activation, which softly blends negative and positive inputs.

```
Function GELU_Derivative_From_Output(x, gelu_output):
  SQRT_2PI = square_root(2 * pi) // Some coding langauges have a std constant for SQRT_2PI
  erf_input = x / square_root(2)

  derivative = 0.5 * (1 + erf(erf_input)) + (x * exp(-0.5 * x * x)) / (SQRT_2PI)

  return derivative
```

Listing 36: Formulaic Pythonic Code for GELU(erf) derivative

12.6 tanh derivative

The tanh derivative measures how the hyperbolic tangent activation changes. It's simply one minus the square of the tanh output, similarly to Sigmoid. This keeps gradients centered around zero, helping models converge faster, but gradients can still vanish for very large or very small inputs.

```
Function TanhDerivative(x)
  tanh_squared_x = x * x
  derivative = 1 - tanh_squared_x

  return derivative
```

Listing 37: Formulaic Pythonic Code for GELU(erf) derivative

12.7 GELU(tanh) derivative

GELU(tanh) derivative is the slope (rate of change) of the GELU(tanh) activation function at each x.

```
Function GELUDerivative(gelu_x)
  derivative = 0.5 + 0.5 * tanh(gelu_x) + (gelu_x * (1 - tanh(gelu_x)^2)) * 0.5

  return derivative
```

Listing 38: Formulaic Pythonic Code for GELU(tanh) derivative

12.8 Softmax derivative

While the Softmax function does have a derivative, it is **rarely if ever needed in practical machine learning**. Since Softmax is typically applied at the output layer for multi-class classification, and back-propagation focuses on computing gradients for internal layers, the explicit derivative of Softmax is usually bypassed. Instead, when combined with Cross-Entropy Loss, the gradient simplifies to a much easier expression without requiring the full derivative.

13 Loss Functions

In other Machine Learning algorithms like Optimizers, your network's loss during training will be required, in order to update your weights according to the loss, how will the network predicted correctly. In short, your loss is used in several algorithms, due to it's assessment of correct prediction.

Keep in mind, that loss is not error. Error is your actual/target minus predicted/output values, not a value measuring your predictions validity

13.1 Mean Abosulute Error Loss (MAE)

Measures the average squared difference between predicted and true values.

```
function mae_loss(actual: Tensor, predicted: Tensor) -> f32:
  // Subtract both Tensors, via Matrix Subtraction.
  diff = actual - predicted;

  // Apply Abosulute operation to each Element in the Tensor.
  abs_diff = diff.abs();

  // Get the mean value of the Tensor, covered earlier in the paper.
  abs_diff.mean()
```

Listing 39: Formulaic Pythonic Code for Mean Abosulute Error Loss (MAE)

13.2 Mean Squared Error Loss (MSE)

Instead of squaring the errors, it takes their absolute value.

```
function mse_loss(actual: Tensor, predicted: Tensor) -> f32:
  let sqared_error: Tensor = (actual - predicted).raised(2);

  // Get the mean value of the Tensor, covered earlier in the paper.
  sqared_error.mean()
```

Listing 40: Formulaic Pythonic Code for Mean Squared Error Loss (MSE)

13.3 Hinge Loss

For Support Vector Machines (SVMs). Encourages a margin between classes.

```
Function HingeLoss(predicted, actual):
  // actual should be -1 or +1
  loss = max(0, 1 - (predicted * actual))
  return loss
```

Listing 41: Formulaic Pythonic Code for Hinge Loss

13.4 Huber Loss

For Robust Regression, less sensitive to outliers than MSE. It's quadratic for small errors and linear for big errors.

```
Function HuberLoss(predicted, actual, delta):
  error = predicted - actual
  if abs(error) <= delta:
      loss = 0.5 * error * error
  else:
      loss = delta * (abs(error) - 0.5 * delta)
  return loss
```

Listing 42: Formulaic Pythonic Code for Huber Loss

13.5 Binary Cross Entropy (BCE)

Quantifies the dissimilarity between predicted probabilities and true binary labels in classification tasks. Measuring how well a model's probabilistic predictions align with the actual outcomes.

```
Function BinaryCrossEntropy(predicted, actual):
  // predicted: must be between 0 and 1 (probability).
  // actual: 0 or 1.
  // Uses epsilon the mathematical constant that is normally in std.

```

```
  predicted = clamp(predicted, epsilon, 1 - epsilon)

  loss = - (actual * log(predicted) + (1 - actual) * log(1 - predicted))
  return loss
```

Listing 43: Formulaic Pythonic Code for Binary Cross Entropy (BCE)

13.6 Categorical Cross Entropy (CCE)

For Multi-class Classification (more than two classes: cat, dog, bird, etc). Similar to BCE but extends it to multiple classes.

```
Function CategoricalCrossEntropy(predicted_probs, actual_label):
  // predicted_probs: a list of probabilities for each class (must sum to roughly 1).
  // actual_label: the index of the correct class (like 0, 1, 2...)
  // Uses epsilon the mathematical constant that is normally in std.

  predicted_probs = clamp_each(predicted_probs, epsilon, 1 - epsilon)

  loss = -log(predicted_probs[actual_label])
  return loss
```

Listing 44: Formulaic Pythonic Code for Categorical Cross Entropy (CCE)

Loss Function	**Key Behavior**
Mean Absolute Error (MAE)	Treats all errors equally by measuring the absolute difference between actual and predicted values.
Mean Squared Error (MSE)	Penalizes larger errors more severely by squaring them, making the model more sensitive to outliers.
Hinge Loss	Builds a margin of safety around classes, encouraging confident classification boundaries (commonly used in Support Vector Machines).
Huber Loss	Smoothly combines MAE and MSE behavior: acts like MSE for small errors and MAE for large errors, making it robust to outliers.
Binary Cross Entropy (BCE)	Measures the difference between two probability distributions in binary classification tasks (two classes: 0 or 1).
Categorical Cross Entropy (CCE)	Extends Cross Entropy to multi-class classification problems, comparing predicted probability distributions over multiple categories.

Table 4: Summary of Common Loss Functions and Their Behavior

14 Optimizers

In Machine Learning, networks rely on a learning rate to control how much their parameters are updated during training. Optimizers dynamically adjust the learning rate and update the network's weights based on how accurate the predictions are, allowing the model to learn more efficiently and converge toward better solutions.

Traditional optimizers such as Stochastic Gradient Descent (SGD) update parameters by stepping proportionally to the negative gradient. However, more sophisticated optimizers like Adam introduce adaptive learning rates and momentum, greatly improving convergence speed and stability."

14.1 Adam's Optimizer

The Adam (Adaptive Moment Estimation) optimizer is an advanced optimization algorithm that combines the benefits of two other methods: momentum and RMSProp. It maintains separate moving averages of both the gradients and their squares, allowing it to adaptively adjust the learning rate for each parameter. Adam

Aspect	Adam Optimizer	SGD Optimizer
Learning Rate Adjustment	Adaptive learning rate for each parameter	Fixed or manually decayed learning rate
Speed of Convergence	Faster convergence, especially early on	Slower convergence without momentum
Memory Usage	Higher (stores moment estimates)	Lower (only stores gradients)
Sensitivity to Learning Rate	Less sensitive to initial learning rate choice	Highly sensitive to learning rate choice
Use Case	Good for complex networks, sparse gradients	Good for simple or very large datasets
Mathematical Complexity	More complex (uses moving averages of gradients)	Simpler (straightforward gradient update)

Table 5: Comparison of Adam and SGD Optimizers

is widely used because it requires little memory, works well with sparse gradients, and typically converges faster than simpler methods.

```
// Recommanded learning rate 0.001
// passed in tensor which should be the backprogation deribvtive of the activiation(weights
    + bias)
// Self is the gradient passed into adam
// Just apply to tensor and bias in layer

function adam_optimizer(
  Tensor,
  m: Tensor, // initialize
  v: Tensor, // initialize
  learning_rate: f32,
  timestep: usize, // usized int
):
  beta1 = 0.9
  beta2 = 0.999
  epsilon = 1e-8

  for row in Tensor:
    for col in row:
      // gradient
      i: f32 = Tensor[row][col]

      // update bias first moment estimate
      m.data[row][col] = beta1 * m.data[row][col] + (1.0 - beta1) * i

      // update bias second moment estimate
      v[row][col] = beta2 * v[row][col] + (1.0 - beta2) * i.raised(2)

      // compute bias fixed estimates
      m_hat = m[row][col] / (1.0 - beta1.raised(timestep))
      v_hat = v[row][col] / (1.0 - beta2.raised(timestep))

      // update parameter
      Tensor[row][col] -= learning_rate * m_hat / (v_hat.sqrt() + epsilon)
```

Listing 45: Formulaic Pythonic Code for Adam's Optimizer

Quick Note: You don't need Adam's Optimizer per say, as SDG is already intergated into Backpropagation, but it increases training speed.

14.2 Stochastic gradient descent (SGD) Optimizer

Stochastic Gradient Descent (SGD) is one of the most fundamental and widely used optimization algorithms in Machine Learning. Instead of computing the gradient of the loss function using the entire dataset, SGD updates parameters using a single sample or a mini-batch, which introduces noise into the optimization

process. This noise can help escape local minima but often requires careful tuning of the learning rate and may converge slower compared to adaptive methods like Adam.

```
for param in parameters:
  param.value -= learning_rate * param.gradient
```

Listing 46: Formulaic Pythonic Code for SDG Optimizer

The Stochastic Gradient Descent (SGD) optimizer is essentially basic gradient descent and is already inherently integrated into the Backpropagation process.

15 Normalization

In Machine Learning, vanishing gradients and exploding gradients can be the downfall of functioning networks. To help prevent vanishing and exploding gradients, normalization can be used to keep data within a range that preserves important features while ensuring safety and maintaining uniformity.

Feature	Min-Max Normalization	Z-Score (Standardization)
Range after scaling	[0, 1] (or custom range)	Centered at 0 (mean=0, std=1)
Sensitive to outliers?	Yes, very sensitive	Less sensitive
Use case examples	Pixel data, bounded inputs	ML models needing normal distribution
Effect on distribution	Linear scaling	Centers and scales without changing shape
When to use?	Known min/max bounds	Data assumed normal

Table 6: Comparison of Min-Max Normalization and Z-Score Standardization

Quick Note: While you don't need Normalization, it really helps to reduce the vanishing and exploding gradients, depending on your input data.

15.1 Z-Score Normalization/Standardization

Centers data around 0 mean, 1 standard deviation.

```
function normalize(Tensor) -> Tensor:
  mean: f32 = Tensor.mean()
  std: f32 = Tensor.std(true) // Tensor Standard Deviation was covered before on the paper.
  return (Tensor.clone() - mean) / std
```

Listing 47: Formulaic Pythonic Code for Standardization

15.2 Min-Max Normalization

Squeezes every value into a nice [0, 1] range by re-scaling based on the minimum and maximum values.

```
function min_max_normalize(Tensor):
  min_value = find_minimum(Tensor)
  max_value = find_maximum(Tensor)

  for row in Tensor:
    for col in row:
      normalized_col = (col - min_value) / (max_value - min_value)
      col = normalized_col
```

Listing 48: Formulaic Pythonic Code for Min-Max Normalization

16 Tensor/Weights and Biases Initialization

To prevent Vanishing or Exploding Gradients, and to improve the convergence of networks, we can initialize weights and biases in a way that matches the expected activation behavior. Proper initialization helps maintain consistent gradient flow during training.

Initialization	Key Points
Xavier (Glorot)	• Balances variance between layers • Works best with tanh or sigmoid activations • Reduces vanishing/exploding gradients • Not ideal for ReLU-based networks
He (Kaiming)	• Designed for ReLU and Leaky ReLU • Helps avoid vanishing gradients • Promotes deeper network training • Can cause exploding gradients if misused
Zero Initialization	• All neurons learn the same features • Causes symmetry problem (no learning) • Only acceptable for biases • Never use for weights
Random Uniform	• Very simple to implement • No scaling for network depth • Can lead to unstable training • Useful for small, shallow networks

Table 7: Summary of Common Weight Initialization Methods with Key Points

16.1 Zero Initialization

Zero initialization is computationally efficient in the short term compared to any pseudo random method, but in the long run, it leads to much slower training times, due to too similar Gradients and causes vanishing gradients.

16.2 Pusdeo Random Initialization

Pusdeo Random Initialization is definitely less sparse then just zeros, but is still not super efficient, for training times.

```
function Random_Tensor(Shape, Min, Max) -> Tensor: // Assuming Tensors of 2nd Rank
  Tensor = // Empty_Tensor
  for row in Shape.x:
    Tensor.append(row)
    for col in Shape.y:
      Tensor[row].append(Pseudo_Random_Range_float32(Min, Max))
```

Listing 49: Formulaic Pythonic Code for Tensor Random Initialization

For Random Uniform Initialization, just set Min and Max to a uniform distribution like -0.5 and 0.5

16.3 Xavier and Glorot Initialization

Xavier (also called Glorot) Initialization sets initial weights by scaling them according to the number of input and output neurons, helping to maintain a stable variance of activations through layers and prevent vanishing or exploding gradients.

```
fan_in = // number of input Neurons/Neural units to the layer
fan_out = // number of output Neurons/Neural units from the layer

For weight in the layer:
  weight = Pseudo_Random_Range_float32( -sqrt(6) / sqrt(fan_in + fan_out), +sqrt(6) / sqrt(
    fan_in + fan_out) )
```

Listing 50: Formulaic Pythonic Code for Xavier and Glorot Initialization

16.4 He Initialization

Designed to improve the training of deep neural networks, particularly those using ReLU activations. It initializes weights by drawing random values from a normal distribution centered at zero, with a standard deviation. This scaling helps maintain stable signal flow through layers by compensating for ReLU's tendency to deactivate neurons, thus reducing the risk of vanishing gradients and promoting faster convergence during training.

```
For layer in network: // (except input layer)
  number_of_inputs = number of input connections to the neurons in this layer
  For weight in layer:
    weight = Random_Normal(mean=0.0, std_dev=sqrt(2 / number_of_inputs))
```

Listing 51: Formulaic Pythonic Code for He Initialization

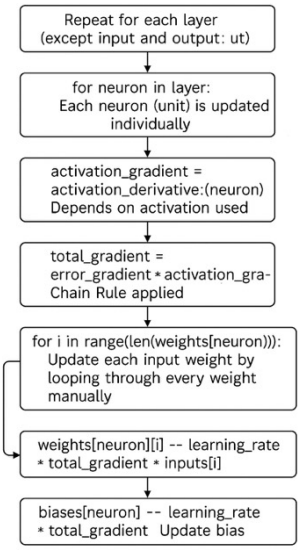

Figure 18: Backpropagation/Backward Formulaic code breakdown.

17 Backpropagation

Crucial for training models, it refines weights and biases to align outputs with targets.

```
// A neuron in this case is a separate weight in a Gradient/Tensor.
// Repeat for each layer (except input and output).
// This is the "backward" function: updates the weights and biases.

for neuron in layer: // Each neuron (unit) is updated individually
  error_gradient = 2.0 * (outputs[neuron] - targets[neuron]) // MSE derivative
  activation_gradient = activation_derivative(neuron) // Depends on activation used
  total_gradient = error_gradient * activation_gradient // Chain Rule applied

  for i in range(len(weights[neuron])):
  // Update each input weight by looping through every weight manually.

      weights[neuron][i] -= learning_rate * total_gradient * inputs[i]


  biases[neuron] -= learning_rate * total_gradient // Update bias
```

Listing 52: Formulaic Pythonic Code for Backpropagation

To **fully understand the formulaic code**, it's best to read through this entire section on backpropagation, along with the detailed breakdowns for each step.

Key subjects that one would need to program for Backpropagation.

- **Matrix multiplication:** The process of Matrix-wise multiplication, that requires Broadcastable Matrices.
- **Transposition:** Rotating a Matrix/Tensor to match the size of another for Matrix-wise operations.
- **Element-wise operations:** These operations act on individual corresponding elements of arrays. They perform a specific operation on each pair of elements at the same position in the arrays.
- **Activation functions:** Mathmatical function applyed to simulate a Weight/Neuron, deciding if the connecting signal, from the previous is strong enough to propagate.

- **Derivatives of those activations:** According to the chain rule of Calculus we need to calculate derivatives, so we can calculate how changes in the weights and biases in earlier layers affect the final output.

The Steps preformed in Backpropagation Keep in mind that a batch refers to a smaller subset within the larger dataset.

1. **Forward Pass:** Done in preparation for the Backprop, calculating the outputs layer by layer
2. **Loss Function:** Understanding the correctness of the network.
3. **Backward Pass:** Computing gradients layer by layer backwards to update the weights and biases.
4. **Gradient Descent:** *Optimization* algorithm that finds the lowest point (minimum) of a function by repeatedly moving in the direction of the steepest descent, based on the negative of the gradient.

- **Compute Output Layer Gradient:** The output layer gradient tells how much the final prediction (the last activation) was wrong compared to the true label
- **Compute Gradients for Hidden Layers:** Go layer by layer backwards, propagating the error backwards from the next layer and adjusting it based on how sensitive your current layer is.
- **Gradient Descent Step (Update Weights and Biases):** Updating the
- **Repeat for All Batches and Epochs:** Repeat the whole forward plus backward pass for each batch and every epoch.

This section will cover each and go over Forward Pass, Backward Pass, and Gradient Descent Step with Formulaic Pythonic Code.

17.1 Forward Pass

Forward Pass is the part where you compute the output of your model, given some input, by passing it through all the layers of the network.

1. **Input (Features):** Start with the input layer.
2. **Linear Transformation (Weights and Biases):** Then each layer does Matrix Multiplication and adds a bias
3. **Nonlinear Activation Function:** After the linear step, we apply an activation function, which allows for non-linearity and complex patterns.
4. **Stack Layers (Hidden Layers):** We repeat steps 2 and 3, for each hidden layer, until the final layer.
5. **Output Layer:** Usually you don't need an activation, but you might use something like Softmax or Sigmoid, for Binary classification.

In a way, you apply matrix multiplication and an activation function to each layer, passing the broadcastable output to the layer below, gradually working your way toward the output layer.

17.2 Loss function

You compare your prediction/output to the real true label/target, using a Loss function, for example: Mean Squared Error, or Mean Absolute Error. This is done once per a batch.

17.3 Backward Pass

The backward pass plays a critical role in the model's learning process. Propagating back from the output layer, it utilizes the loss to positively reinforce the network's weight adjustments for improved performance.

- **Compute the Gradients with (Chain Rule of Calculus):** set Gradients to how much each parameter (weight/bias) contributed to the error.

1. **Start at the Output layer**
2. **Move Backward: Hidden Layers**
3. **Once You Have the Deltas**
4. **Update Step**

17.4 Summary: Backpropagation

Backpropagation is the foundational algorithm that enables neural networks to learn from data. It systematically updates each weight and bias in the network by moving backward through the layers, calculating how much each parameter contributed to the final error. By applying the chain rule of calculus, the algorithm efficiently computes gradients layer by layer, allowing the model to adjust and minimize its loss function. The process consists of a Forward Pass to generate outputs, a Loss Function to measure errors, a Backward Pass to compute gradients, and a Gradient Descent step to update parameters. Repeating this cycle across batches and epochs enables the network to progressively improve its performance, ultimately learning complex patterns from the input data. Mastering backpropagation is crucial for understanding how modern deep learning models train and optimize.

18 Conclusion

In this paper, I set out to break down and demystify the core formulas of Machine Learning, removing unnecessary barriers that often block developers, especially those without access to calculus-intensive education. By making these fundamental concepts more approachable, we can open the door to a more diverse group of developers who might otherwise be excluded. For example, highly capable low-level engineers, who may not have taken advanced calculus classes, but excel in systems programming, Linux, WebAssembly, and even quantum computing, could bring their unique expertise to Machine Learning. This would help break what I call the double-genius barrier, a phenomenon where both deep mathematical and deep coding skills are assumed prerequisites. By lowering this barrier, we can significantly speed up innovation in the Machine Learning industry, and benefit humanity as a whole by welcoming a broader range of brilliant minds into the field.

18.1 Guidelines for Making Machine Learning More Accessible

First and foremost, educators and writers should not rely solely on advanced calculus formulas to explain Machine Learning concepts. Machine Learning is fundamentally rooted in machines and programming, not just pen, paper, and abstract mathematical thought detached from real-world application. At a minimum, educators should provide a pseudo-code or formulaic version alongside the traditional mathematical expressions. Most programmers either never studied advanced calculus in depth or have forgotten it over time. By bridging this gap, we can ensure that Machine Learning knowledge becomes more accessible to a wider and more diverse group of developers.

18.2 Machine Learning Without Walls: Empowering Global Innovation

Imagine a world where mathematicians focus entirely on advancing faster and more human-like activation functions, such as improved versions of Hodgkin–Huxley, one of the most biologically accurate activation models to date. Meanwhile, hardcore Linux and assembly programmers could directly embed neural networks

into the Linux kernel itself, optimizing tasks like real-time pattern recognition, such as determining the most efficient scroll speeds for devices. By removing the expectation that every developer must split their energy 50-50 between high-level mathematics and low-level coding, we could allow individuals to specialize more deeply in their natural strengths. Mathematicians alone cannot fully realize the future of Machine Learning, nor can programmers, but together, by combining cutting-edge mathematical discovery with low-level engineering excellence, we can accelerate innovation beyond anything achievable alone. This collaboration would not only boost individual efficiency but also create a stronger, faster, and more accessible Machine Learning ecosystem for everyone.

To mathematicians: you don't have to carry the future of Machine Learning alone. To programmers: your skills are just as vital. Together, we can break barriers that no single discipline could ever overcome.

18.3 A Complete Machine Learning Network Example

To show how these principles can transform Machine Learning in practice, I present a full network design built with accessibility and clarity at its core.

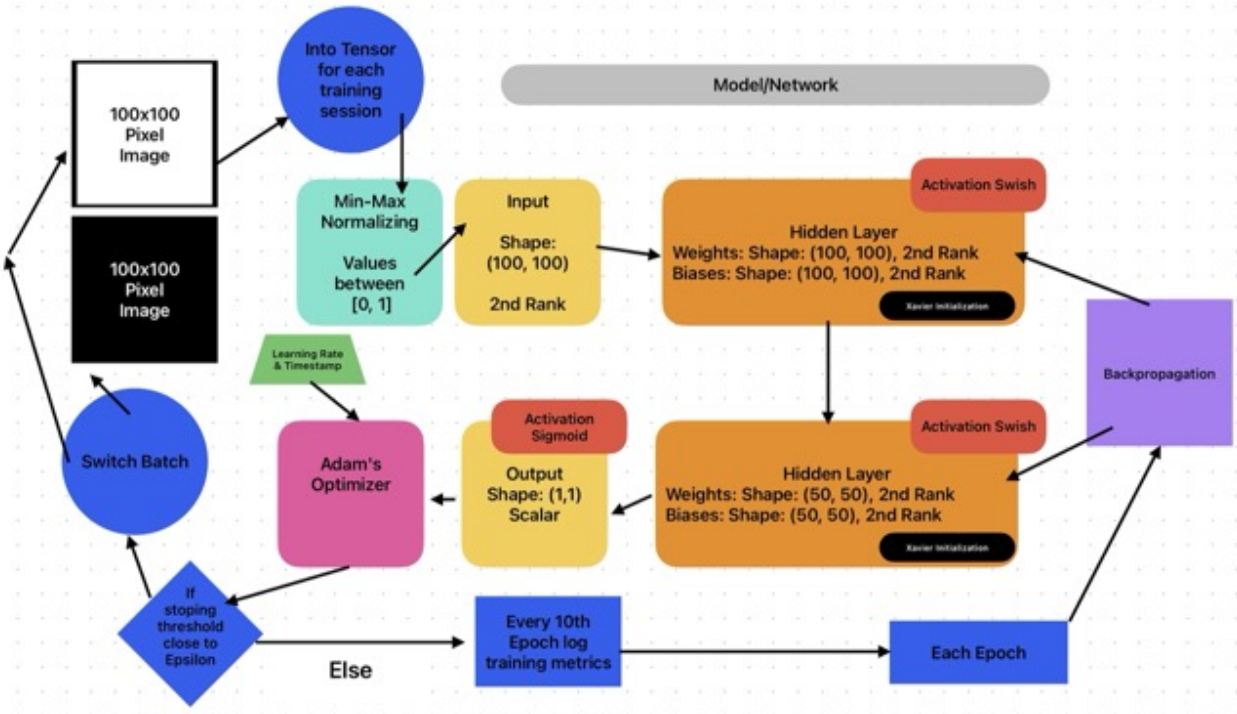

Figure 19: Complete machine learning network flow. This diagram shows the full cycle of an image-based network, including data preprocessing, tensor conversion, network layer flow with activations, training optimization using Adam's Optimizer, and backpropagation updates across epochs.

```
// 1. Preprocessing
For each image:
    Normalize pixel values to [0, 1]

// 2. Initialization
Initialize weights and biases for each layer with Xavier Initialization

// 3. Training Loop
For each epoch:
    For each image in batch:
```

```
        // Forward Pass
        Input = image tensor
        Hidden1 = Swish(Weights1 x Input + Biases1)
        Hidden2 = Swish(Weights2 x Hidden1 + Biases2)
        Output = Sigmoid(Weights3 x Hidden2 + Bias3)

        // Loss Calculation
        Loss = (Output - Target).raised2

        // Backward Pass (Backpropagation)
        Compute gradients of Loss w.r.t Weights and Biases
        Update Weights and Biases using Adam Optimizer

    // Logging
    If epoch mod 10 == 0:
        Log training metrics

    // Stopping Condition
    If Loss < Epsilon:
        Break training loop
```

Listing 53: Formulaic Pythonic Code for Example Complete machine learning network flow.

19 Final Reflection: The Power of Formulaic Thinking

To conclude, this example shows that the entire machine learning process can be captured cleanly through formulaic pseudocode, without hiding behind walls of pure math. From image preprocessing to network layer transitions, activations, optimization, and backpropagation, every step remains clear and accessible.

While mathematical notation like:

$$h_1 = \text{Swish}(W_1 x + b_1) \quad \text{and} \quad L = (\hat{y} - y)^2$$

is powerful, it often skips the operational thinking that real-world engineering demands.

Formulaic pseudocode, however, preserves the logic path step-by-step, making it easier to bridge understanding between mathematics and practical implementation.

This bridge is crucial.

By empowering both mathematicians and programmers to see through each other's worlds more clearly, we can push forward a future where neural networks are not only more powerful but also more understandable and accessible to all learners.

The journey from image to intelligence is not built on formulas alone, it is built on clarity, intuition, and the relentless pursuit of better communication.

"The journey from data to intelligence begins with clarity, not complexity."

License & Attribution

Author GitHub: `https://github.com/had2020`