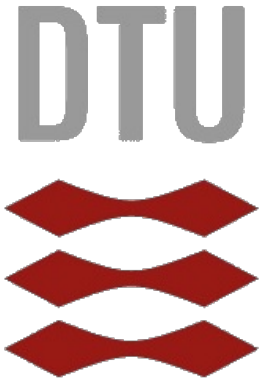


Danmarks Tekniske Universitet

Mathematical Modeling

02526

Report 2

Group 5

Members:	*Student number:*
Bhatnagar, Agrim	s225149
Crucirescu, Maxim	s225209
Gudmandsen, Mikkel Herskind	s214127
Pitman, Lucia Alice	s225114

Date: March 16, 2025

1 Introduction

This project presents the implementation, training, and analysis of a neural network model designed from scratch in Python to initially classify data for the sea/land classification problem in the Libyan desert. A neural network is a machine learning program, or model, that makes decisions in a manner similar to the human brain [?]. The neural network is built using fundamental operations such as rotation, bias addition, and activation functions. We develop and compare multiple design choices with varying depths, including a deep network with eight layers and a shallow network with four layers, to then assess their performance on the dataset. Furthermore, we investigate the impact of different activation functions and visualize decision boundaries to understand how network complexity influences classification accuracy. Model uncertainty is also analyzed by visualizing prediction variance through confusion matrix. The results demonstrate the advantages and limitations of deeper architectures and reveal insights into how activation functions contribute to model performance. The findings of this project provide a comprehensive understanding of neural network behavior and suggest potential improvements for more effective classification.

2 Methods and Materials

a. Data Processing

First thing we did with the data was to load the data. We did this by importing the pandas library to work with data in tabular format. Examining the first few rows and renaming the columns to "x", "y", and "label" for clarity making sure to recheck the data again after renaming. (as seen in the code)

Next we wanted to visualized the processed data. We applied a 90-degree counterclockwise rotation transformation to the data points meaning that each original (x,y) point becomes (y,-x) in the rotated coordinates.

We then where able to separate the data into two categories based on the "label" column. With the points labeled as 1 being 'seen' as 'Land' and the ones as 0 being 'seen' as 'Sea'. Next step was then to create a scatter plot to visualize the rotated data. The land points being yellow and the sea points being blue, with slight transparency for better visual clarity.

b. Constructing the Neural Networks

Two different network architectures were designed. One was a deep network, which consists of 8 layers while the shallow network consists of half of that, only having 4 layers.

Each layer of both networks consists of three key functions, which are as follows:

1. apply_rotate(input, angle): Rotates the input vector by a given angle and returns the results.
2. apply_bias(input, b): The function adds a bias term to the input vector and returns the results, allowing the neural network to shift its decision boundary away from the origin. This increases model flexibility by enabling neurons to produce non-zero outputs even with zero inputs, which is essential for properly learning the sea/land classification boundaries and is a key tunable parameter in your manual model implementation.
3. apply_activation(input, activation = 'abs'): The function applies a non-linear transformation to the input vector, with the default being the absolute value function. This non-linearity is crucial because it allows the neural network to learn complex patterns and relationships beyond what linear transformations alone can achieve, enabling the model to form more sophisticated decision boundaries for your sea/land classification task.

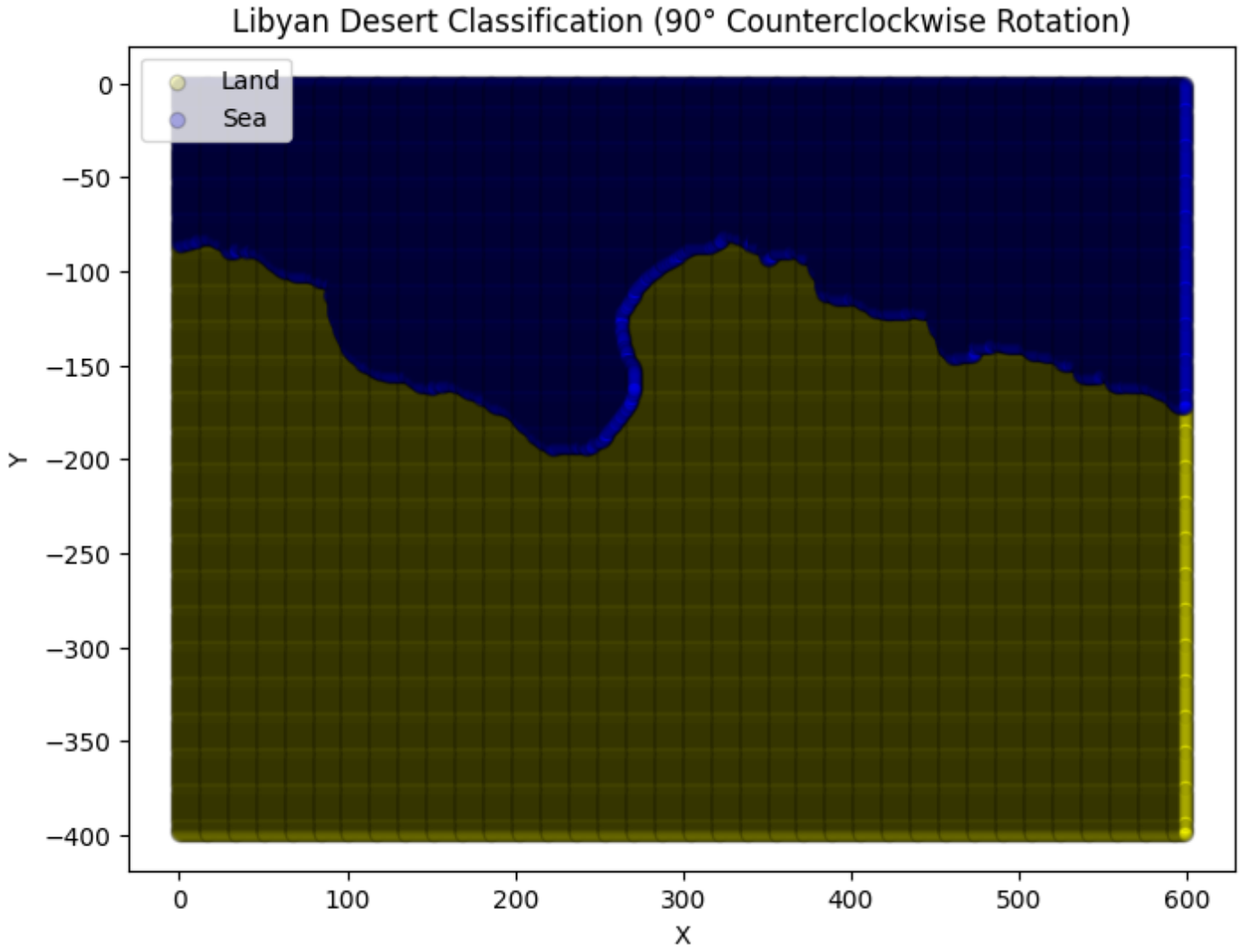
   Various activation functions are used in the project and will be individually discussed in more detail.

c. Visualization of Results

The data was visualized a couple of ways thought the project. Most notably plotted between various layers to see how the network is working. Equally as important however is the confusion matrix plotting.

For clarification a confusion matrix is a performance evaluation tool that displays the relationship between predicted and actual classifications in a tabular format. It works by organizing predictions into four categories: true positives (correctly identified positives), false positives (incorrectly identified positives), true negatives (correctly identified negatives), and false negatives (incorrectly identified negatives). Confusion matrices are especially useful because they provide a comprehensive view of model performance beyond simple accuracy, revealing specific error types and patterns.

This addition to the accuracy calculation performed in this stage of the process, the use of a confusion matrix is very beneficial. Confusion matrices also allow us to calculate important metrics like precision and recall, which helping identify whether a model struggles more with false positives or false negatives.



3 Part 1

The code for this section can be seen in the appendix. It is the application of the layers and the functions within them. The implementation functions in reference as outlined clearly in part b of the Methods and Materials section of the report.

A deep model was created with a 0.83 accuracy score which can be found in the Appendix 1. Using this code we receive the following Figure 1 where you can see the visualized layer transformations on some data points, caused by the activation functions which rotate, add bias, and apply an activation function.

Based on Figure 1 it can be seen that the sequential transformations demonstrate the neural network's evolving capability to separate the two classes through non-linear manipulations of the feature space. Throughout the three layers, the initial data undergoes rotation, bias addition, and absolute value activation, gradually enhancing the distinction between purple and yellow points. By Layer 3, the class separation appears more pronounced than in the original data, indicating the network's success in creating an effective non-linear decision boundary. The significant scale changes observed across layers (from the original Âą1.5 range to the final 339-343 range) illustrate how the network constructs its own internal data representation. The absolute value activation function plays a crucial role in this process by preserving spatial relationships while introducing the necessary non-linearity to capture complex patterns in the data, ultimately transforming the input space to facilitate more accurate classification.

4 Part 3

Implementing Softmax activation function

The Softmax activation function works by using probability distribution output. Softmax transforms inputs into a probability distribution, where all outputs sum to 1. This assessment makes it ideal for multi-class classification problems. Another benefit of the Softmax function is that the outputs can be directly interpreted as class probabilities, making the model's predictions more intuitive. Softmax is also differentiable, allowing for effective backpropagation during training.

One of the limitations of Softmax are that the exponential calculations can be quite computationally expensive, especially with many classes. There is also an element of numerical instability which has the possibility of causing overflow/underflow issues with very large or small input values. Another problem is overconfidence which often produces overly confident predictions, even when uncertainty is high.

Like sigmoid, softmax outputs are not zero-centered, which can slow down convergence during training. Its limitations also lie in multi-label classification, since outputs sum to 1, it's not appropriate for problems where multiple classes can be simultaneously active.

The softmax function that was implemented into the code as seen in the appendix 2 and then, as also seen in the appendix implemented into a deep model 3. This unfortunately only ever gave accuracy results of around 0.33. This was also true for when the softmax function was only used on the last layer of the model.

It is known that softmax suffers from vanishing gradient problem, this is the presumed cause for the problems faced when implementing this function. The vanishing gradient problem occurs because the gradients become extremely small as they're back propagated through multiple softmax layers, preventing effective learning. Its also possible that the problem came from using softmax at all. When softmax in every layer because when it is applied in each layer, it normalizes the outputs to sum to 1. This can severely restrict the information flow through the network, as the dynamic range of values becomes compressed with each layer. With 8 layers all using softmax, each subsequent layer receives inputs that have been repeatedly normalized, making it difficult for the network to maintain meaningful signal propagation.

These problems are highlighted in the visualized layer transformations, using the clear original data given in the project, on some data point as seen in Figure 2.

ReLU

The Rectified Linear Unit (ReLU) activation function,

$$f(z) = \max(0, z)$$

is widely used due to its simplicity and efficiency when training deep neural networks.

The idea behind ReLU is that it allows positive values (including 0) to remain unchanged, while transforming the negative values into 0. In addition, this aspect introduces a higher selectivity, because if a neuron is fed with mostly negative numbers, it will become inactive (will output 0). It has advantages, like reducing the computational power and might prevent overfitting of the model. On the other hand, it is prone to failing due to a phenomenon called: "dying ReLU", which means that neurons trained with negative values will have an output of zero and its gradient remains zero during back propagation, meaning that the neuron is not learning anymore, it is essentially dying.

Thankfully, there is the Leaky ReLU function which remediates this issue by giving each negative value a small slope instead of zero.

$$f(z) = \max(\alpha z, z) \quad \text{with } 0 < \alpha \ll 1$$

Tanh

Zero-centered outputs (-1 to 1) Still suffers from vanishing gradient Doesn't provide probability distribution

Effect on Problem Solving Choosing softmax affects how a problem is solved in several ways:

Problem Formulation: Forces the problem to be structured as selecting one class from many, rather than independent binary decisions. Loss Function: Typically paired with cross-entropy loss, which is particularly effective for classification tasks. Decision Boundaries: Creates decision boundaries that maximize the probability of the correct class. Interpretability: Makes the model's output directly interpretable as confidence levels in each class. Training Dynamics: May require techniques like temperature scaling or label smoothing to address overconfidence.

The choice of softmax is most appropriate when you need mutually exclusive class predictions with probabilistic interpretation, particularly in neural network classifiers.

Sigmoid

The sigmoid function (also known as the "logistic curve"),

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

is a commonly used activation function due to the way it maps any value on the entire real number line to a value in the range between 0 and 1. Very negative inputs are outputted to values close to 0 and very positive values end up close to 1. The output steadily increases through 0.5 "in the middle", or as inputs close to 0 rise from negative to positive. Essentially the activation is a measure of how positive the weighted sum is. Also note that the sigmoid function is equivalent to the hyperbolic tangent function, $\tanh(x)$, just rescaled and shifted vertically. The sigmoid function is especially useful compared to similar functions e.g.

$$\frac{x}{1 + |x|}, \quad \text{or} \quad \frac{x}{\sqrt{1 + x^2}}$$

These also map the real number line to the range between 0 and 1, but they both face the same issue. The sigmoid's associated loss function $-ln(\sigma(x))$ has a consistent gradient, which makes it much easier for the model to learn. The loss functions associated with the two other functions do not have this consistent gradient. Lastly, the sigmoid function is also related to probability. If two normal distributions have the same variance but different means, the probability of an observation belonging to the the normal distribution with the higher mean is described by the sigmoid function.

5 Conclusion

In summary, this project built neural networks from scratch to classify sea/land data. We compared shallow and deep networks, exploring different activation functions. Deep networks with Softmax struggled due to vanishing gradients, highlighting activation function importance. ReLU, Tanh, and Sigmoid had varied impacts. Confusion matrices improved performance evaluation. Visualizations showed network complexity's effect. The project provides insights for enhancing classification models in future work.

6 Appendix

Code 1: Deep Model with 0.83 Accuracy

```
deep_config = [
        {'angle': -np.pi/4, 'bias': np.array([-150, 150]), 'activation': 'abs'},
        {'angle': np.pi/3, 'bias': np.array([-100, 100]), 'activation': 'abs'},
        {'angle': -np.pi/6 , 'bias': np.array([-250, 250]), 'activation': 'abs'},
        {'angle': np.pi/8, 'bias': np.array([-200, 200]), 'activation': 'abs'},
        {'angle': -np.pi/3, 'bias': np.array([-150, 150]), 'activation': 'abs'},
        {'angle': np.pi/12, 'bias': np.array([-300, 300]), 'activation': 'abs'},
        {'angle': -np.pi/8, 'bias': np.array([-200, 200]), 'activation': 'abs'},
        {'angle': np.pi/6, 'bias': np.array([-100, 100]), 'activation': 'abs'}
    ]
```

Code 2: SoftMax implimenation code

```

def apply_activation(input, activation='abs'):
    """
    Applies the specified activation function to the input.

    Args:
        input: Input array or vector
        activation: String specifying the activation function to use

    Returns:
```

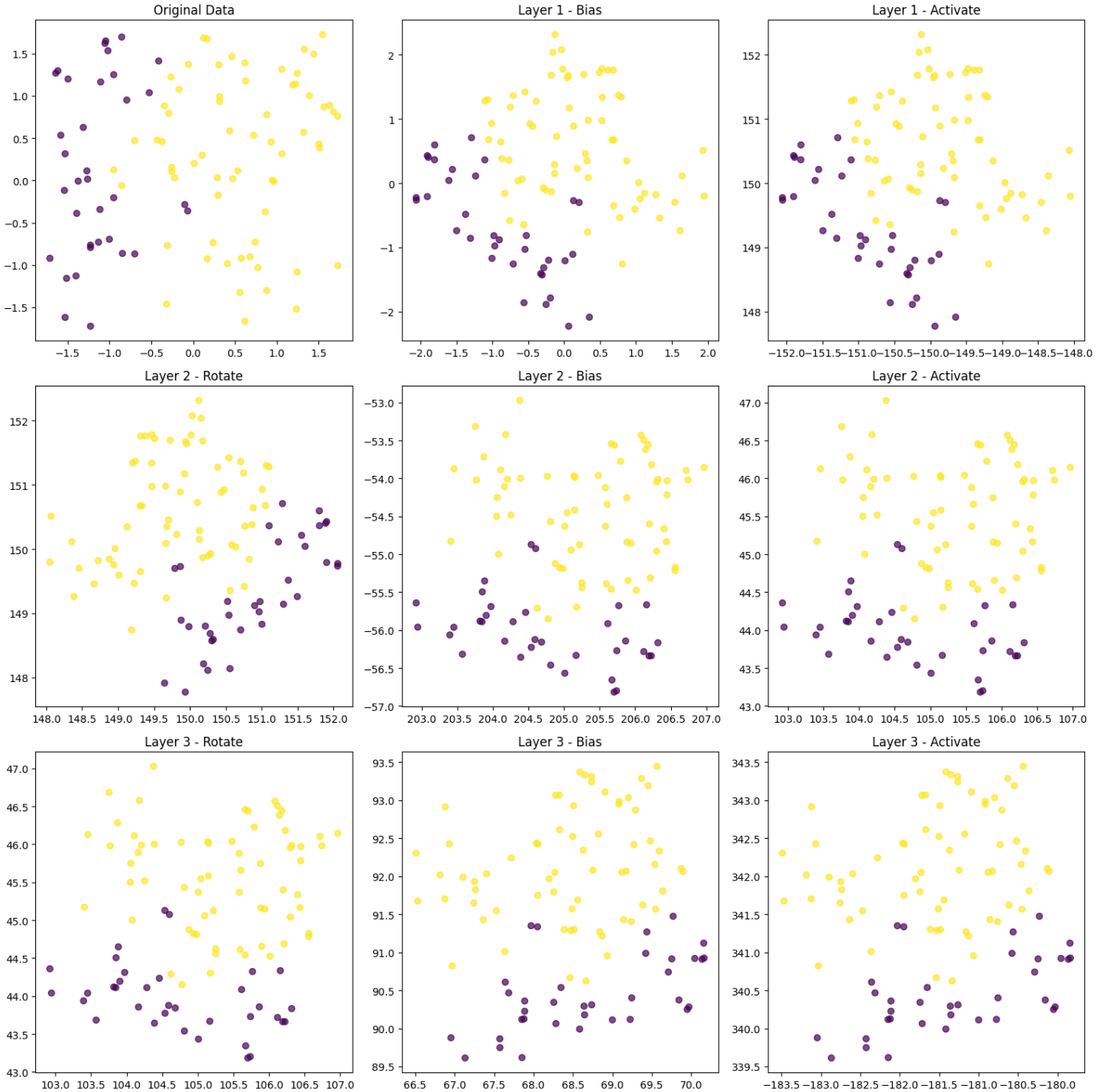


Figure 1: Visualized layer transformations of the deep model with an accuracy of 0.83

```
        Transformed input after applying the activation function
    """
    if activation == 'abs':
        return np.abs(input)
    elif activation == 'softmax':
        # Subtract max for numerical stability
        exp_x = np.exp(input - np.max(input, axis=-1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=-1, keepdims=True)
    # You can add other activation functions here
    else:
        raise ValueError(f"Activation function '{activation}' not recognized")
```

Code 3: SoftMax Deep Model code

```
deep_config = [
        {'angle': -np.pi/6, 'bias': np.array([-5, 5]), 'activation': 'softmax'},
        {'angle': np.pi/4, 'bias': np.array([-8, 8]), 'activation': 'softmax'},
        {'angle': -np.pi/3, 'bias': np.array([-7, 7]), 'activation': 'softmax'},
        {'angle': np.pi/5, 'bias': np.array([-10, 10]), 'activation': 'softmax'},
        {'angle': -np.pi/8, 'bias': np.array([-6, 6]), 'activation': 'softmax'},
        {'angle': np.pi/10, 'bias': np.array([-9, 9]), 'activation': 'softmax'},
        {'angle': -np.pi/12, 'bias': np.array([-7, 7]), 'activation': 'softmax'},
        {'angle': np.pi/8, 'bias': np.array([-5, 5]), 'activation': 'softmax'}
    ]
```

7 References

article hyperref

IBM. (n.d.). *Neural Networks*. Retrieved March 15, 2025, from https://www.ibm.com/think/topics/neural-networks

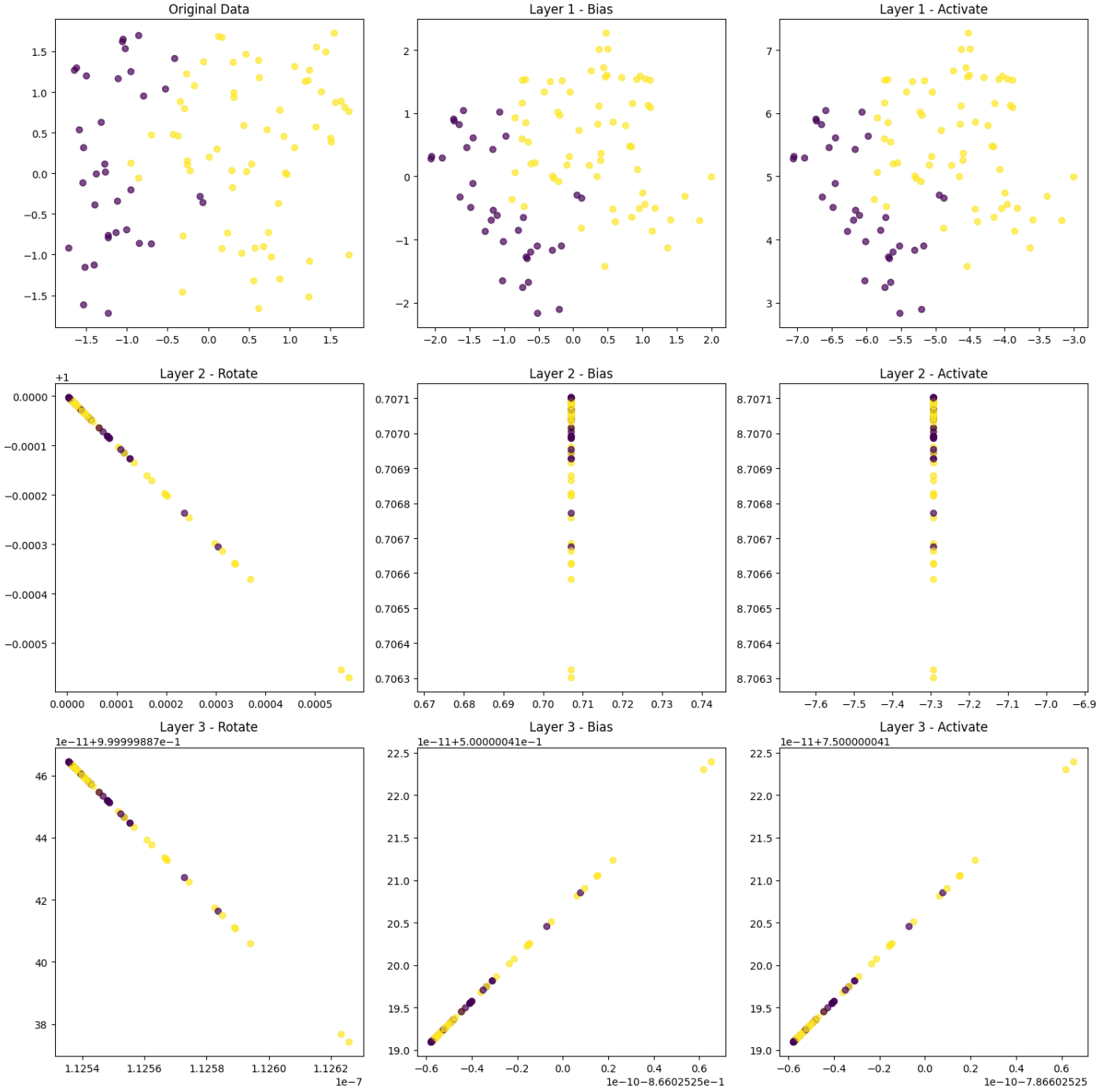


Figure 2: Visualized layer transformations of the softmax activation function