# ReLU and other Activation Functions

Emma Morrison, Jared Gode, and Jack Conner

June 9, 2025

## 1 Introduction

The rectified linear unit, or ReLU, has become the most widely accepted and used activation function in neural networks due to its simplicity and training performance. ReLU avoids some of the vanishing gradient issues that affect other popular activation functions like tanh and sigmoid. However, due to the structure of the ReLU function, namely, that all negative input values will output as 0, there has been an observed issue known as the dying ReLU problem. The dying ReLU problem is when a large number of neurons in the neural network output 0 for all inputs during training, leaving many dead neurons that don't provide any meaningful information to the network (1). As network depth increases, there will be an increase in the number of dead neurons. One reason for this issue is the common practice of symmetric initialization in the weight and bias matrices of the network. In fact, recent research has shown that for a symmetrically initialized network, as network depth approaches infinity, the network will fully die (3).

There are a few main strategies to try and address the dying ReLU problem. One is to modify the network architecture itself, while another is to add additional training steps in the network–like normalizing or a dropout technique. A third is to alter the weight and bias initialization mentioned previously. All of these have proven effective in certain cases to rectify the dying ReLU issue (3). The focus of this paper will be on a fourth method–slightly altering the activation function. Our goal is to investigate several alternative, ReLU-like activation functions on simulated data to determine the efficacy of this method in addressing the dying ReLU problem.

There are many different alternate activation functions. Current research is investigating quantum-inspired activations such as QReLU and m-QReLU, which leverage principles of superposition and entanglement to enhance non-linearity and model expressivity. These methods show improved classification accuracy in medical imaging (e.g., CT, MRI, and X-ray analysis) (2). However, the focus of our research is not on image classification, but simpler models with symmetric initialization–and the impact of variants of ReLU

on classification and predictive accuracy of these models.

Several ReLU variants have been introduced to mitigate the dying neuron problem. Leaky ReLU (LReLU) introduces a small slope for negative inputs. This slope is defined as alpha, which is usually given the weight 0.01, but in Pytorch, we use 0.2 or 0.3. While LReLU offers small improvements in gradient flow, it does not significantly enhance classification performance or convergence robustness (4).

Thus, Absolute Leaky ReLU (ALReLU) was developed to mitigate the negative output of LReLU. ALReLU takes the absolute value of the LReLU, stabilizing the negative portion and improving training convergence (4).

We also aimed to investigate the following variants: Exponential Linear Unit (ELU) and Scaled Exponential Linear Unit (SELU)–which both improve convergence speed and normalize outputs across layers by using a smooth exponent-esque curve to ensure differentiability–and the Parametric ReLU (PReLU), where alpha is adaptive and learned during training (2).

Vanishing gradients and dying ReLU remain central concerns in training deep networks. As gradients backpropagate through many layers, they can shrink to near-zero values. This hampers the training of early-layer weights and stalls convergence. Activation functions like ReLU mitigate this to some extent by maintaining non-zero gradients for positive inputs, but when outputs are consistently zero, learning halts — the core of the dying ReLU problem. Some of these activation functions–like PReLU–suffer from the vanishing gradient issue still. As such, we aim to see which alternative activation function is the most effective on several metrics of model performance to provide an alternative solution to the dying ReLU problem, targeting the activation function rather than weight initialization.

## 2 Definitions and Notation

Below are the definitions of each activation function we reference or investigate in the paper.

Rectified Linear Unit (ReLU):

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{1}$$

Leaky ReLU (LReLU):

$$f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0, \alpha > 0 \end{cases} \tag{2}$$

Absolute Leaky ReLU (ALReLU):

$$f(x) = \begin{cases} x & x > 0 \\ |\alpha x| & x \leq 0, \alpha > 0 \end{cases} \tag{3}$$

Exponential Linear Unit (ELU):

$$f(x) = \begin{cases} x & x > 0 \\ \alpha(\lceil^{x} - 1) & x \leq 0, \alpha > 0 \end{cases} \tag{4}$$

Scaled Exponential Linear Unit (SELU), for a $\lambda > 0$:

$$f(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(\lceil^{x} - 1) & x \leq 0, \alpha > 0 \end{cases} \tag{5}$$

Parametric ReLU (PReLU):

$$f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0, \alpha > 0, \text{learned during training} \end{cases} \tag{6}$$

Weighted F1 Statistic:

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7}$$

# 3 Methods

For our investigation, we designed a series of experiments to analyze how different activation functions perform in the presence of the dying ReLU problem. These functions would be tested against the baseline of a ReLU activation function, comparing each function's test accuracy, F1 score, (both for classification data), test MSE (for numeric data), and computational time to train.

For this experiment, we generated several varying synthetic datasets to compare the impact of our metrics across several possible dimensions. We initialized 30 different datasets. These were of sample size 100 or 500, with input dimensions of 1, 2, or 5 (increased by one for classification datasets). All of the input features were sampled from a standard normal distribution of mean 0, variance 1. Random noise was simulated from a standard normal distribution of mean 0, variance 0.01. The target output curves for each numeric dataset had

one of four underlying curves–linear, quadratic, exponential, or sinusoidal–while each classification dataset had two outcome classes. These varying methods of initialization were aimed to help us investigate neural network performance under simple and complex inputs.

We then trained four neural networks on each dataset. Each of these neural networks had fixed width of 8, and varying depths of 1, 3, 10, and 30 layers. This was in order to investigate the impact of increasing depth on the dying ReLU problem. Each combination of dataset and activation function was trained ten times to account for variation due to random initialization and training dynamics. Each activation function that we used is defined above, and the code chunks are at the bottom of the paper. The definitions for most classes of the activation functions were adapted from PyTorch documentation, and the ALReLU function was coded by hand.

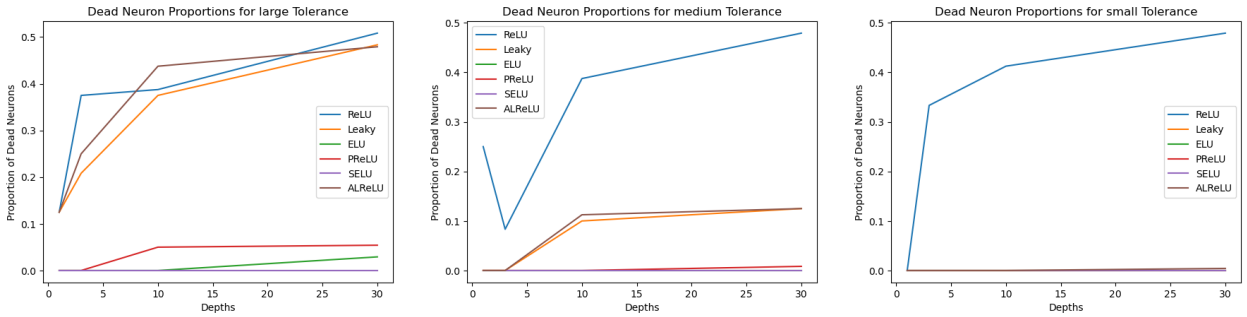
To verify the existence of the dying ReLU problem, we counted the amount of dead neurons under each network, with small to large tolerances. We used tolerances of $\epsilon = 0.01, 0.001, 0.00001$ to count the number of dead neurons on simple neural networks (one-dimensional input and output) for each activation function at each of the specified depths, where a neuron was counted as dead if its weight was less than $\epsilon$.

Finally, we analyzed the results from our networks based on a few key numeric and classification metrics–neural network test accuracy or MSE, time to train, and Weighted F1. The test accuracy/MSE gave us a reference to understand how well the neural network trained to predict the correct output on test data, to help us understand the model's qualitative performance. Time to train was recorded to understand the efficiency of each activation function and convergence time. Finally, the weighted F1 statistic–formula above (7)–provides another metric which is especially useful for classification outputs, as it balances precision and recall of the network.

# 4 Main Results

Below we see the results of our production of the dying ReLU problem (Figure 1).

As illustrated by Figure 1, the ReLU neural network consistently produced a larger proportion of dead neurons than any other activation function, especially for smaller tolerance levels. In particular, although LeakyReLU and ALReLU seem to have similar proportions of dead neurons at the large tolerance level ($\epsilon = 0.01$), their proportions quickly drop, and then go to zero as the tolerance drops. The other activation functions follow a similar pattern, with all except for SELU having some proportion of dead neurons at a large tolerance, but quickly dropping as the tolerance decreases. From these investigations, we also see that the proportion of dead neurons increases as depth increases for ReLU, indicating that the dying ReLU problem is particularly problematic in deeper networks. This is as expected based on previous literature

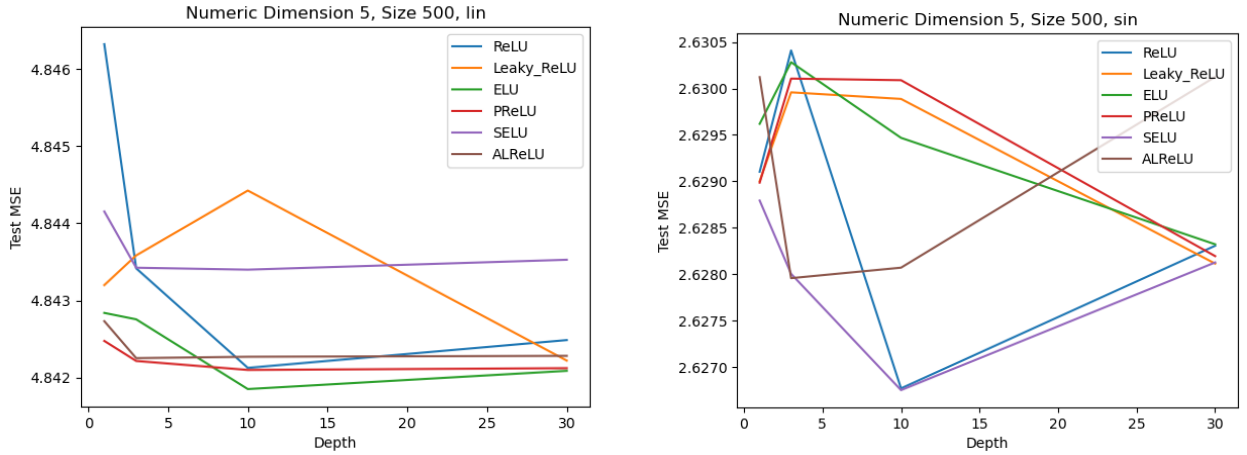



(a) Tolerance 0.01 (b) Tolerance 0.001 (c) Tolerance 0.00001

Figure 1: Dead Neuron Count over Activation Functions

discussed in the introduction.

We investigated each numeric model's performance first based on the test mean squared error. We aimed to minimize this amount to represent an effective neural network. Firstly the linear and sine models (Figure 2).



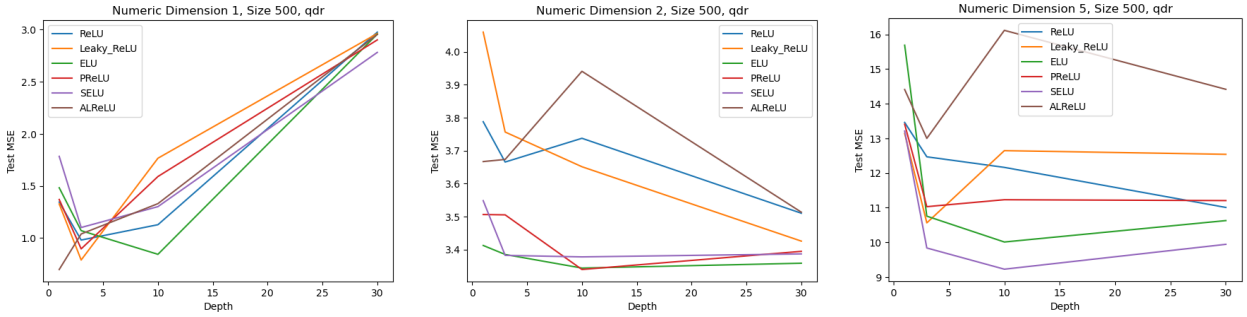

Linear, 500 Sample Size and Five Dimensions Sine, 500 Sample Size and Five Dimensions

Figure 2: Linear and Sinusoidal Model Performance with Largest Dataset

The figures above detail the performance of each different activation function. Shown above is the performance on the linear and sine models for a sample size of 500, and in a five-dimensional neural network. We measure the impact of depth on test mean squared error of the model performance.

One particularly import aspect of Figure 2 is the y-axis, showing the test MSEs. While we see some variation across each activation function, the relative scale of this variation is very small, with a range of less than 0.01 for each model. This indicates that activation function doesn't make much of a difference for the linear and sinusoidal underlying relationships, even for the most complex datasets of our set ($N = 500$, $d = 5$).

Then, the investigation of quadratic models revealed different performance results for different underlying relationships in the data (Figure 3).




(a) Dimension 1 (b) Dimension 2 (c) Dimension 5

Figure 3: Sample Size 500 Quadratic Model Performance

In the graphs above, you can see the sample size 500, and all three dimensions of input, and those effects on the performance of our neural network.

In the one-dimension dataset, we do not see much difference between the activation functions or over depth, as there is a slight increase that is relatively small compared to the higher dimensions to the right.

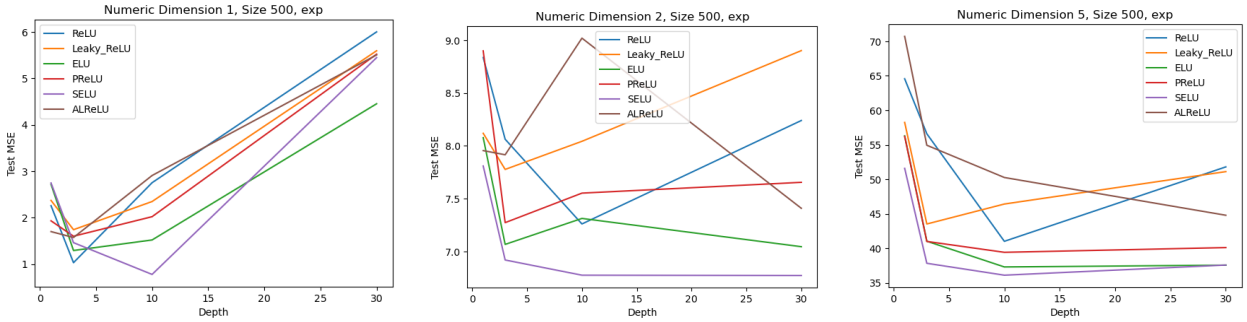
In the two-dimension dataset, ReLU, Leaky ReLU, and ALReLU have the highest MSEs, with ELU, SELU, and PReLU outperforming them consistently. However, counterintuitively, we find a decrease in the test MSE as depth increases. We can attribute this to chance, though, as, if we look at the scale of the y-axis again, we see that the range of the MSEs is fairly low overall.

This trend differs in the five-dimension dataset–not only is there a difference in patterns, but there is a substantial increase in MSE overall. The ending MSE for SELU, the best performing activation function, is around 10, whereas in the two-dimension dataset the ending MSE is around 3.25. ALReLU performs significantly worse than the other models, especially at higher depth, with an average test MSE of nearly 15, two points higher than the second-highest test MSE of 13 from LeakyReLU. Surprisingly, ReLU is the third-best performer off test MSE for the quadratic dataset, only beaten by ELU and SELU, which again have the lowest test MSE of the set of alternative activation functions.

Finally, we ran our exponential model (Figure 4).

In the one-dimension dataset, ReLU has comparatively large MSE compared to all other alterative activation functions as the depth approaches 30. All of the activation functions experience a dip in MSE as the model depth first increases, before all rising again consistently as the depth increases. This mirrors the results from the quadratic model as well, with ELU and SELU again outperforming the other models.

In the two-dimension dataset, the MSE is higher overall, as expected. Leaky ReLU, ALReLU, and PReLU end with the highest MSEs, with ELU and SELU outperforming all other models consistently, and




(a) Dimension 1 (b) Dimension 2 (c) Dimension 5

Figure 4: Sample Size 500 on Exponential Model Performance

remaining with the smallest MSE (around 7, others around 8).

This trend continues into the five-dimension dataset–but one large difference is once again the increase in MSE overall. The ending MSE for SELU, the best performing activation function, is around 37, whereas in the two-dimension dataset the ending MSE is around 7. ReLU continues to perform the worst, ending at around 50 MSE, but LeakyReLU is not far behind, with a near-50 test MSE as well. This dataset has the highest MSE over all the models, possibly indicating difficulty in the activation function capturing the underlying exponential relationship between the data.

Next, we investigated the training time for each model and network combination. The results for the classification models are below (Table 1).

From our table, we can see that over the classification datasets, Leaky ReLU, ELU, and SELU outperform others in time to train. In our numeric datasets, ELU and SELU consistently out perform in time to train.

The function that performed the worst, in terms of training time, was consistently PReLU, which trained the slowest across variations in depth, dimension, and number of input observations.

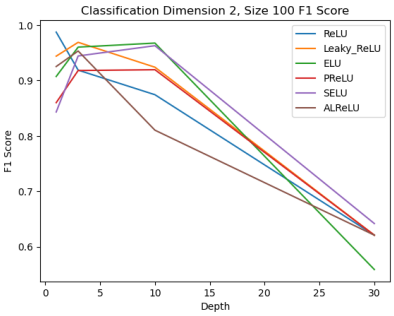
To the credit of ReLU, it was commonly the fastest in training time for the lower-depth models, showing why it is favored in practice. However, for deeper neural networks, Table 1 indicates that ReLU may not be the preferred activation function for deeper networks in terms of training time.

Our final metric was the weighted F1 statistic. Below are the F1 statistics for the small and large ($N = 100, 500$ respectively) classification datasets (Figures 5 and 6).

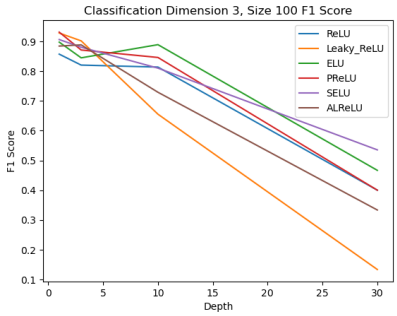
From the F1 analyses, we can see a clear indication that SELU outperforms all other alternative activation functions, especially as depth, dimension, and sample size increase. ALReLU, PReLU, and ReLU all seem to perform worse than the others, particularly for the higher sample size, but overall there doesn't seem to be much of a difference except in the results for SELU and ELU.

Table 1: Class Time Comparison Results

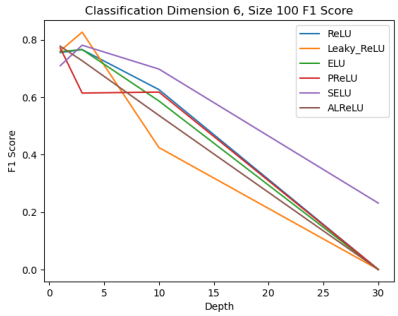
| Observations | Dimensions | Depth | ELU | Leaky ReLU | PReLU | ReLU | SELU | Best Activation |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | 1 | 0.00701 | 0.00835 | 0.01174 | 0.00720 | 0.00631 | selu |
| 100 | 1 | 3 | 0.02300 | 0.01142 | 0.01809 | 0.01958 | 0.02036 | leaky |
| 100 | 1 | 10 | 0.07889 | 0.03836 | 0.05135 | 0.02984 | 0.03729 | relu |
| 100 | 1 | 30 | 0.07596 | 0.08553 | 0.10791 | 0.08704 | 0.09179 | elu |
| 100 | 2 | 1 | 0.00643 | 0.00604 | 0.00711 | 0.00916 | 0.00708 | leaky |
| 100 | 2 | 3 | 0.01369 | 0.01301 | 0.01464 | 0.01000 | 0.01220 | relu |
| 100 | 2 | 10 | 0.04642 | 0.03749 | 0.04837 | 0.03315 | 0.02860 | selu |
| 100 | 2 | 30 | 0.08313 | 0.11876 | 0.13021 | 0.08642 | 0.13091 | elu |
| 100 | 5 | 1 | 0.00729 | 0.00701 | 0.00750 | 0.00711 | 0.00761 | leaky |
| 100 | 5 | 3 | 0.01226 | 0.01230 | 0.01448 | 0.01188 | 0.03231 | relu |
| 100 | 5 | 10 | 0.06530 | 0.02873 | 0.08635 | 0.03949 | 0.07589 | leaky |
| 100 | 5 | 30 | 0.08815 | 0.07436 | 0.14907 | 0.12574 | 0.09389 | leaky |
| 500 | 1 | 1 | 0.00675 | 0.01120 | 0.00984 | 0.00764 | 0.00805 | elu |
| 500 | 1 | 3 | 0.01154 | 0.01339 | 0.01739 | 0.02059 | 0.01429 | elu |
| 500 | 1 | 10 | 0.03733 | 0.06545 | 0.04057 | 0.03235 | 0.04552 | relu |
| 500 | 1 | 30 | 0.08672 | 0.09757 | 0.17272 | 0.09199 | 0.10382 | elu |
| 500 | 2 | 1 | 0.01069 | 0.00827 | 0.00885 | 0.00786 | 0.00771 | selu |
| 500 | 2 | 3 | 0.01417 | 0.01558 | 0.01557 | 0.01368 | 0.01628 | relu |
| 500 | 2 | 10 | 0.04528 | 0.02862 | 0.06832 | 0.04440 | 0.02957 | leaky |
| 500 | 2 | 30 | 0.08839 | 0.07809 | 0.13018 | 0.07997 | 0.13925 | leaky |
| 500 | 5 | 1 | 0.02044 | 0.01707 | 0.02414 | 0.01065 | 0.01613 | relu |
| 500 | 5 | 3 | 0.01261 | 0.01248 | 0.02769 | 0.01539 | 0.03649 | leaky |
| 500 | 5 | 10 | 0.03030 | 0.02991 | 0.05145 | 0.04160 | 0.02801 | selu |
| 500 | 5 | 30 | 0.12637 | 0.08091 | 0.11332 | 0.08599 | 0.08356 | leaky |




(a) Dimension 2

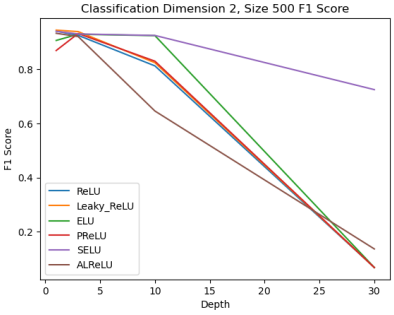



(b) Dimension 3

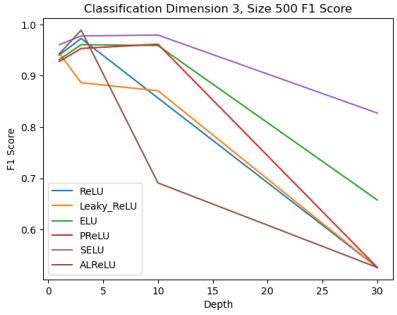



(c) Dimension 6

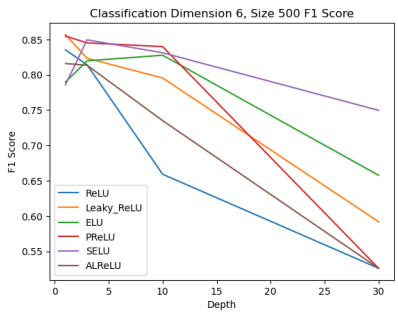
Figure 5: Sample Size 100 on F1 Score




(a) Dimension 2




(b) Dimension 3




(c) Dimension 6

Figure 6: Sample Size 500 on F1 Score

# 5 Conclusions

Our experiments assessed a neural network's performance and ability to mitigate the dying ReLU issue across numeric prediction, classification accuracy, and training efficiency, providing a comprehensive understanding of how various activation functions behave under changing network and input structures.

Numerically, we observed that SELU consistently yielded the minimum test MSE across most model types and dimensions—particularly in deeper neural networks. In the linear and sinusoidal models, MSE generally stayed in a similar range across depths, suggesting that the activation functions tested don't provide much difference in capturing these underlying relationships. For the more complex exponential and quadratic models, SELU, ELU, and ALReLU outperformed ReLU consistently, with SELU improving greatly over increased depth, sample size, and dimensionality. Notably, even when overall MSE increased by 10-20 in high-dimensional settings, SELU maintained a lower score.

Furthermore, we measured time to train the neural network. Leaky ReLU, ELU, and SELU demonstrated superior speed, particularly in classification datasets. For larger sample sizes and higher depths, ELU and SELU were consistently faster, making them appealing for real-world applications that require greater efficiency, or very large input datasets.

In general terms, the variation that had the clearest impact on training time was network depth, as greater network depth was consistently associated with longer training times. However, for certain activation functions, greater dimensionality or a greater number of observations sometimes shortened training times, such as with SELU, which achieved faster training times with 500 observations than with 100. This leads us to believe that network depth has the greatest impact on activation efficiency. This connects to the theorem in which an infinitely deep neural network will ultimately completely die, more and more neurons become dead, even in SELU, as our depth increases.

From a classification standpoint, weighted F1 scores further emphasized the observation that SELU outperforms other activation functions. This pattern was especially clear in complex models, where F1 performance improved with increasing depth, sample size, and dimensionality. ReLU and ALReLU consistently have lower F1 scores, especially in high-dimensional settings. These findings support the hypothesis that SELU's self-normalizing properties contribute not only to lower error but also to more consistent classification performance.

What we have effectively shown is the superiority of ReLU variants in mediating neural network performance when the dying ReLU problem is present. Furthermore, we have shown the superiority of SELU in metrics like minimizing mean squared error and F1 score. This conclusion emphasizes the importance of self-normalizing activations in deep architectures, particularly when depth exceeds 10 layers or when data is

high-dimensional, as demonstrated in our F1 score graphs. Although Leaky ReLU and PReLU were originally developed to prevent the dying ReLU problem, they underperform relative to SELU and ELU in both accuracy and training time. ALReLU produced some of the highest MSEs and slowest convergence rates, showing it's inefficiency in deep neural networks.

In terms of practical recommendations, we suggest that ReLU be used for shallow neural networks or simple datasets. For large, complex datasets and deep networks, we recommend using SELU or ELU, as these performed best across a variety of metrics and datasets, providing evidence of their ability to handle such network architecture.

In the future, it would be interesting to investigate how further changes in the input and neural network structure could affect the efficacy of the alternative functions. For instance, dropout, batch normalization, or layer-wise adaptive learning might further improve performance in combination with some of the alternative activation functions.

Furthermore, an investigation into SELU's strong performance in this experiment could provide further understanding of why this activation function consistently outperforms the others across the many metrics measured in our experiment. Additionally, specifically investigating SELU under other network constraints like added noise could help us understand how the self-normalization impacts accuracy and classification.

Finally, testing this hypothesis on real-world data, and not just synthetic data, would further emphasize the importance of our findings and highlight the applicability of alternative activation functions as another way to mitigate the dying ReLU problem.

# References


(1) Kenneth Leung. "The Dying ReLU Problem, Clearly Explained." Mar 30, 2021, Towards Data Science.

(2) Luca Parisi, Daniel Neagu, Renfei Ma, Felician Campean. "Quantum ReLU activation for Convolutional Neural Networks to improve diagnosis of Parkinson's disease and COVID-19." Expert Systems with Applications, Volume 187, 2022, https://doi.org/10.1016/j.eswa.2021.115892.

(3) Lu, Lu and Yeonjong Shin, and Yanhui Su, and George Em Karniadakis. "Dying ReLU and Initialization: Theory and Numerical Examples." 2020, http://dx.doi.org/10.4208/cicp.OA-2020-0165.

(4) Stamatis Mastromichalakis. "ALReLU: A different approach on Leaky ReLU activation function to improve Neural Networks Performance." 2021, https://arxiv.org/abs/2012.07564.


# Code

Listing 1: Classification Functions

```
class class_NN_ReLU(nn.Module):
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_ReLU, self).__init__()
        layers = []
        #Input layer
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(nn.ReLU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x) #Provides raw logits for use in cross-entropy loss

class class_NN_LeakyReLU(nn.Module):
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_LeakyReLU, self).__init__()
        layers = []
        #Input layer
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(nn.LeakyReLU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.LeakyReLU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class class_NN_ELU(nn.Module):
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_ELU, self).__init__()
```

```
        layers = []
        #Input layer
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(nn.ELU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.ELU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class class_NN_PReLU(nn.Module): # documentation recommends not using weight decay with this
    ↪ training
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_PReLU, self).__init__()
        layers = []
        #Input layer
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(nn.PReLU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.PReLU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class class_NN_SELU(nn.Module):
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_SELU, self).__init__()
        layers = []
        #Input layer
```

```
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(nn.SELU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.SELU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

class ALReLU(nn.Module):
    def __init__(self, alpha=0.01):
        super().__init__()
        self.alpha = alpha

    def forward(self, x):
        return torch.max(torch.abs(self.alpha * x), x)

class class_NN_ALReLU(nn.Module):
    def __init__(self, n_dim_x = 2, n_class_y = 1, depth = 1, width = 2):
        super(class_NN_ALReLU, self).__init__()
        layers = []
        #Input layer
        layers.append(nn.Linear(n_dim_x, width))
        layers.append(ALReLU())
        #Hidden layers
        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(ALReLU())
        layers.append(nn.Linear(width, n_class_y))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```