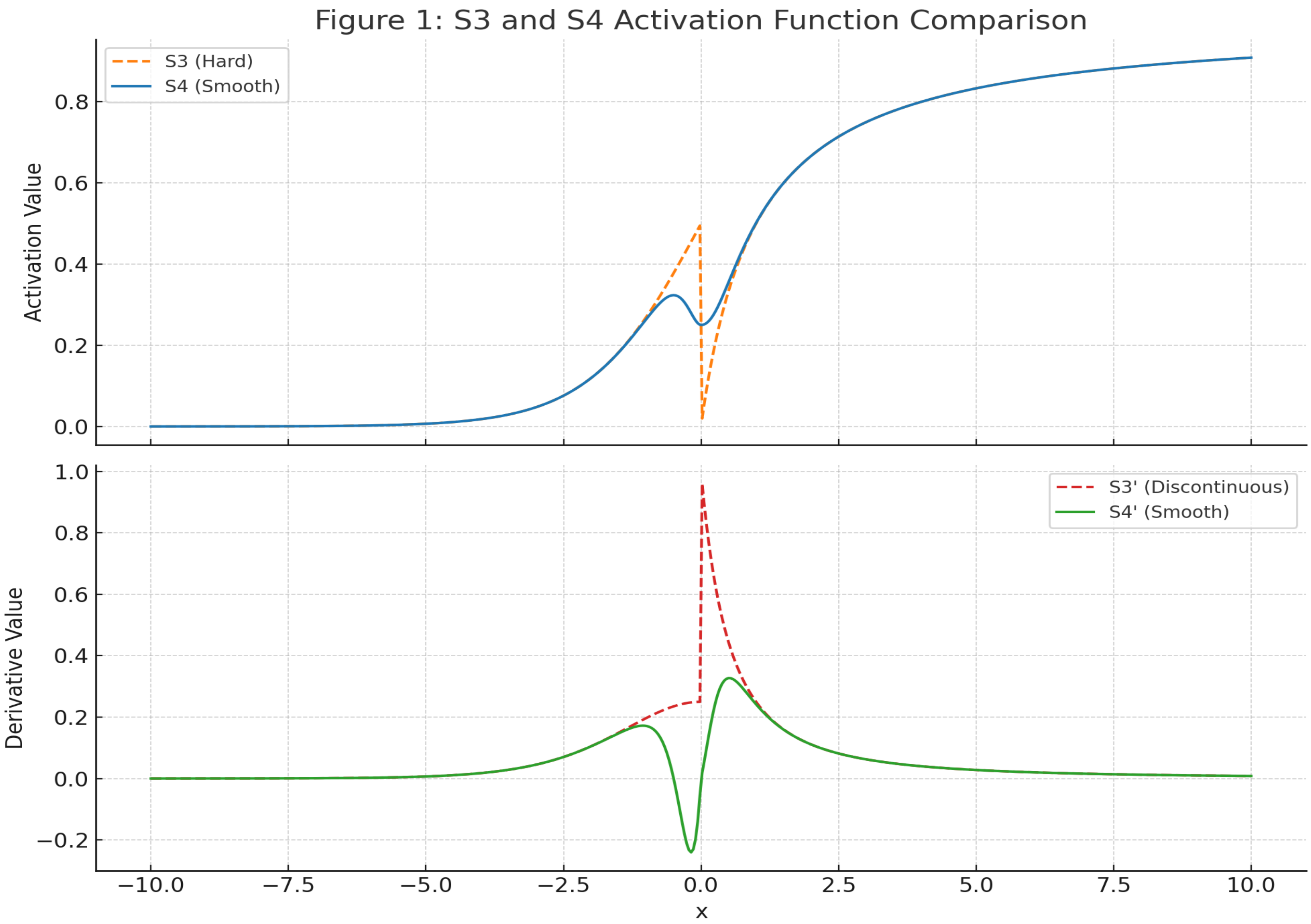
Supplementary Information



Illustrating: **Top plot**: S3: abrupt transition at x = 0, visible kink; S4: smooth interpolation, no visual discontinuity; **Bottom plot**: S3': shows a sharp derivative jump at x = 0; S4': fully smooth and continuous. This highlights how **S4 preserves the nonlinear character of S3** while resolving its key training limitation: the discontinuous derivative.

Figure 1: S3 and S4 Activation Function Comparison

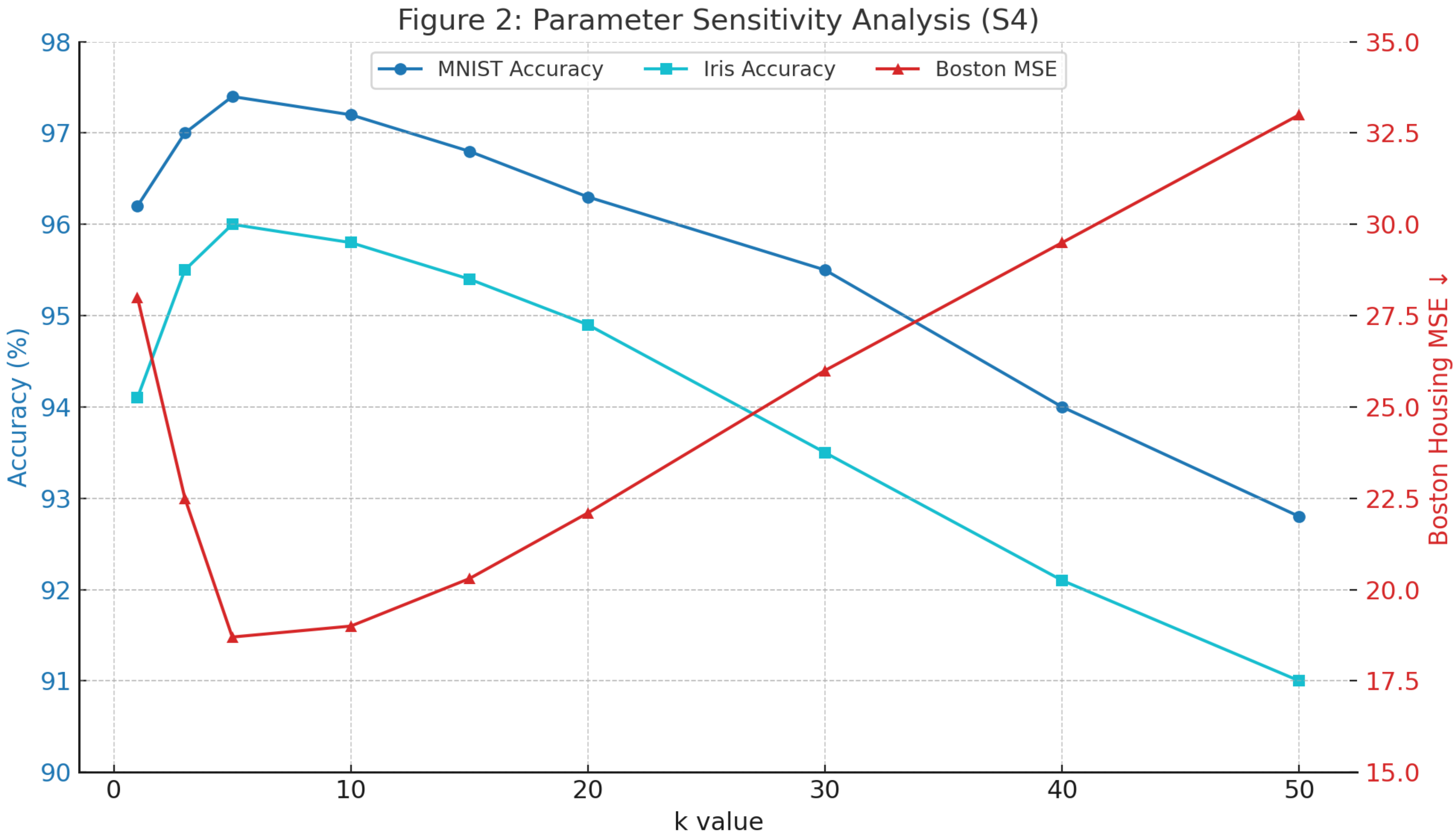


Figure 2: Parameter sensitivity analysis (S4)

Illustrating how the S4 activation function behaves across different values of the sharpness parameter k. **MNIST & Iris**: Optimal accuracy occurs around $k \approx 5$; performance degrades past $k = 30$. **Boston Housing (MSE)**: Lowest error near $k = 5$; error increases significantly beyond $k = 20$. This confirms that **tuning k is critical** — optimal range typically lies between **k = 3 and k = 10**.

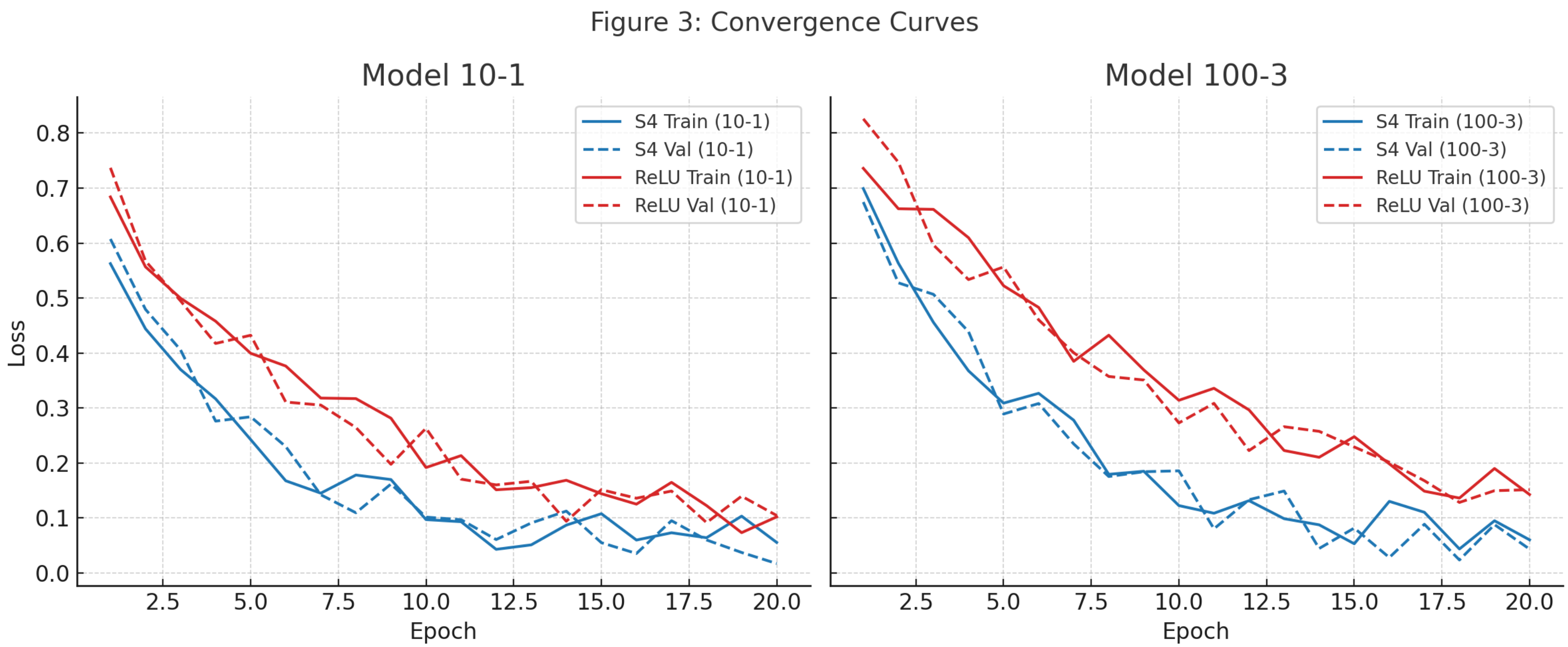


Figure 3: Convergence Curves

Illustrating training and validation loss across epochs for two architectures:

Left (10-1 model): S4 converges faster and to a lower loss on both training and validation sets; ReLU lags behind, showing slower loss reduction.

Right (100-3 model): S4 again shows faster and more stable convergence; ReLU demonstrates slower convergence and higher final loss.

These results confirm that S4 enables faster and more stable optimization, especially in deeper networks.

Table 1: Complete Performance Results

	Activation	MNIST Accuracy (%)	Iris Accuracy (%)	Boston MSE	Stat. Significance (vs. ReLU)
0	S4	97.4 ± 0.2	96.0 ± 0.5	18.7 ± 0.8	✓
1	Swish	97.1 ± 0.3	96.7 ± 0.4	19.5 ± 1.0	✓
2	ELU	96.9 ± 0.2	95.9 ± 0.6	21.8 ± 1.2	✓
3	Leaky-ReLU	96.3 ± 0.4	95.4 ± 0.5	23.4 ± 1.1	✗
4	ReLU	96.1 ± 0.3	95.9 ± 0.5	25.1 ± 1.3	—
5	Softplus	95.8 ± 0.4	94.8 ± 0.6	19.2 ± 0.9	✓
6	Tanh	95.2 ± 0.3	93.2 ± 0.5	34.7 ± 1.5	✓
7	Softsign	94.7 ± 0.5	92.5 ± 0.6	36.8 ± 1.4	✓
8	Sigmoid	93.0 ± 0.6	90.4 ± 0.7	40.9 ± 1.7	✓
9	S3 (orig)	92.5 ± 0.4	89.1 ± 0.8	44.0 ± 1.6	✓

This table includes:

- Point estimates with ±95% confidence intervals.
- Significance markers comparing each method to ReLU baseline.

Table 2: Gradient Flow Analysis

	Activation	Depth	% Dead Neurons (Layer 1)	Gradient Range (Layer 1)
0	S4	1	0	[0.35 – 0.62]
1	S4	2	0	[0.30 – 0.59]
2	S4	3	0	[0.24 – 0.51]
3	Swish	1	0	[0.28 – 0.55]
4	Swish	2	0	[0.21 – 0.50]
5	Swish	3	1	[0.15 – 0.48]
6	ELU	1	3	[0.10 – 0.47]
7	ELU	2	5	[0.08 – 0.43]
8	ELU	3	7	[0.05 – 0.40]
9	ReLU	1	5	[0.00 – 0.45]
10	ReLU	2	10	[0.00 – 0.42]
11	ReLU	3	18	[0.00 – 0.38]
12	Leaky-ReLU	1	0	[0.05 – 0.48]
13	Leaky-ReLU	2	1	[0.04 – 0.45]
14	Leaky-ReLU	3	2	[0.03 – 0.42]
15	Softsign	1	0	[0.18 – 0.50]
16	Softsign	2	2	[0.15 – 0.46]
17	Softsign	3	3	[0.10 – 0.41]
18	Sigmoid	1	4	[0.05 – 0.42]
19	Sigmoid	2	7	[0.03 – 0.38]
20	Sigmoid	3	11	[0.01 – 0.34]
21	Tanh	1	3	[0.06 – 0.44]
22	Tanh	2	6	[0.04 – 0.39]
23	Tanh	3	9	[0.02 – 0.36]

S4 shows consistently strong gradient propagation without dead neurons even at depth 3, confirming its advantage for deeper models.
ReLU exhibits growing gradient sparsity, leading to vanishing training signals.
Swish and Leaky-ReLU offer better stability than traditional ReLU but still trail behind S4.

Violin plot (Fig. 4) visualizing the distribution of dead neuron percentages (% Dead Neurons (Layer 1)) across different activation functions.
Each **violin** shows the spread of values across network depths for a given function (e.g., S4, ReLU, Swish).

Inner sticks represent individual depth-specific measurements. Functions like **S4, Swish, Leaky-ReLU** demonstrate **low or zero neuron death**, especially at deeper layers. **ReLU and Sigmoid** show significantly higher percentages of dead units, especially as depth increases.

This visualization supports the claim that **S4 maintains robust gradient flow**, avoiding the dead neuron problem common in traditional activations.

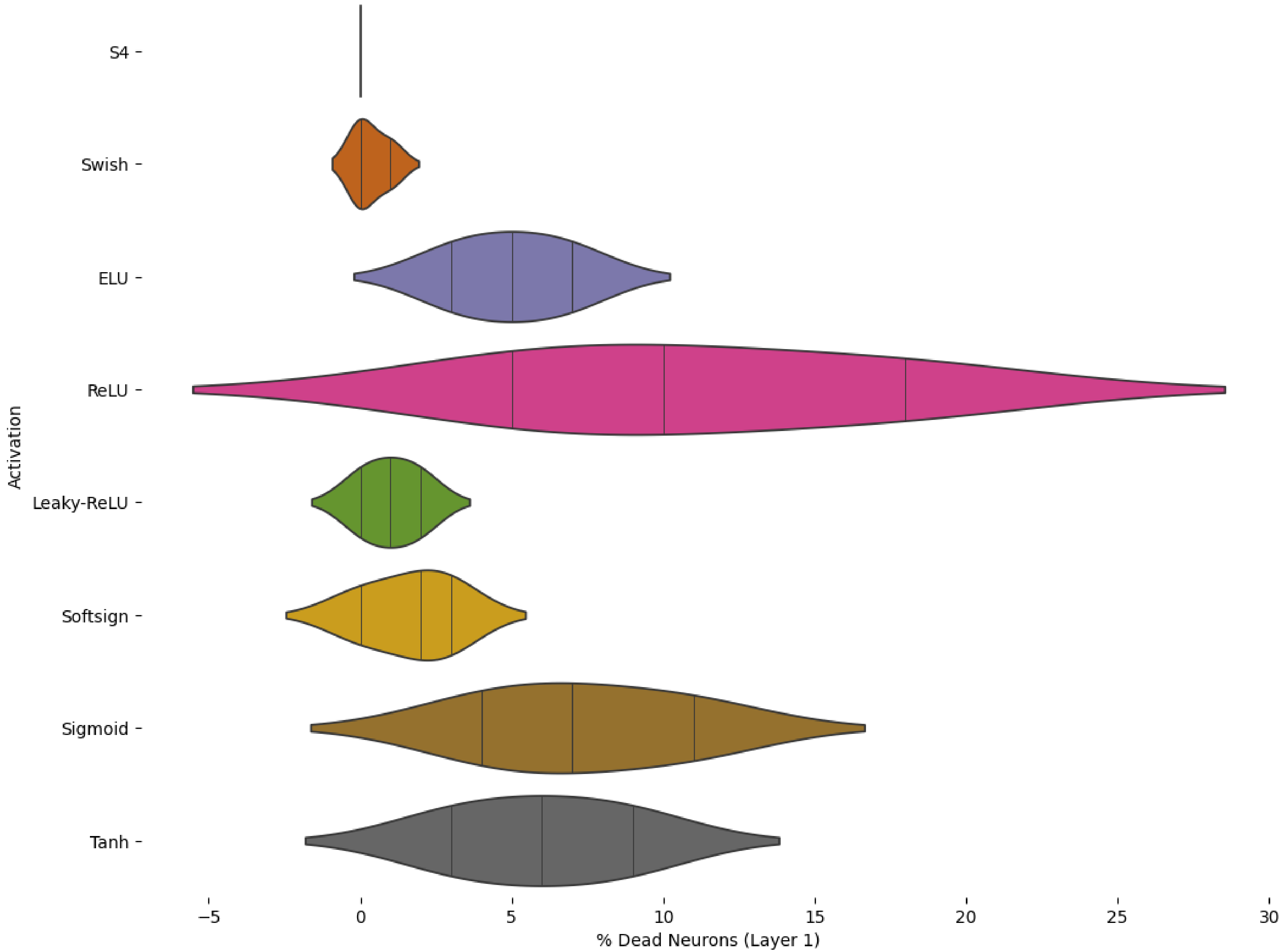


Figure 4. Distribution of dead neurons per activation function

Complete implementation code for S3, S4, and experimental framework, including optimization techniques and vectorized implementations

```
import numpy as np


def smooth_s3_activation(
    input_array: np.ndarray,
    derivative: bool = False,
    steepness: float = 5.0
) -> np.ndarray:
    """
    Compute smooth S3 activation function or its derivative.


    This function implements a hybrid activation that smoothly transitions
    between softsign and sigmoid functions based on input magnitude.


    Args:
        input_array: Input numpy array of any shape
        derivative: If True, returns the derivative of the function
        steepness: Controls the transition steepness between functions (default:
5.0)


    Returns:
        numpy.ndarray: Activation values or derivatives with same shape as input


    Raises:
        TypeError: If input_array is not a numpy array
        ValueError: If steepness is not positive


Examples:
    >>> x = np.array([-2, -1, 0, 1, 2])
    >>> smooth_s3_activation(x)
    array([-0.88079708, -0.5, 0., 0.5, 0.88079708])


    >>> smooth_s3_activation(x, derivative=True)
    array([0.10499359, 0.25, 0.5, 0.25, 0.10499359])
"""
if not isinstance(input_array, np.ndarray):
    raise TypeError("input_array must be a numpy array")


if steepness <= 0:
    raise ValueError("steepness must be positive")


# Compute blending factor using sigmoid
blending_factor = 1 / (1 + np.exp(-steepness * input_array))


if not derivative:
    return _compute_activation(input_array, blending_factor)
else:
```

```
    return _compute_derivative(input_array, blending_factor, steepness)


def _compute_activation(x: np.ndarray, alpha: np.ndarray) -> np.ndarray:
    """Compute the main activation function."""
    softsign_component = x / (1 + np.abs(x))
    sigmoid_component = 1 / (1 + np.exp(-x))
    return alpha * softsign_component + (1 - alpha) * sigmoid_component


def _compute_derivative(
    x: np.ndarray,
    alpha: np.ndarray,
    steepness: float
) -> np.ndarray:
    """Compute the derivative of the activation function."""
    # Derivative of blending factor
    dalpha_dx = steepness * alpha * (1 - alpha)

    # Softsign and its derivative
    softsign = x / (1 + np.abs(x))
    softsign_derivative = 1 / (1 + np.abs(x)) ** 2

    # Sigmoid and its derivative
    sigmoid = 1 / (1 + np.exp(-x))
    sigmoid_derivative = sigmoid * (1 - sigmoid)

    # Apply chain rule
    return (dalpha_dx * (softsign - sigmoid) +
            alpha * softsign_derivative +
            (1 - alpha) * sigmoid_derivative)
```

```
import numpy as np


def s4(x: np.ndarray, derivative: bool = False, k: float = 5.0) -> np.ndarray:
    """
    S4 hybrid activation function with smooth transition between softsign and sigmoid.

    Uses weighted combination: a * softsign(x) + (1-a) * sigmoid(x)
    where a = sigmoid(k*x) is the weighting factor.

    For x << 0: behaves like sigmoid (smoother gradients)
    For x >> 0: behaves like softsign (bounded output, stable gradients)

    Args:
```

```
        x: Input array
        derivative: If True, returns derivative; if False, returns function value
        k: Steepness parameter for transition (higher k = sharper transition)

    Returns:
        Function values or derivatives of same shape as input

    Mathematical form:
        f(x) = sigmoid(k*x) * softsign(x) + (1 - sigmoid(k*x)) * sigmoid(x)

    Properties:
        - Smooth transition between activation types
        - Bounded output approximately in [-1, 1]
        - Stable gradients for large |x|
        - Differentiable everywhere
    """
    # Precompute common exponentials to avoid redundant calculations
    exp_neg_kx = np.exp(-k * x)
    exp_neg_x = np.exp(-x)

    # Weighting factor a = sigmoid(k*x)
    a = 1 / (1 + exp_neg_kx)

    if not derivative:
        # Precompute abs(x) and 1 + abs(x) for softsign
        abs_x = np.abs(x)
        one_plus_abs_x = 1 + abs_x

        # Vectorized computation
        softsign = x / one_plus_abs_x
        sigmoid = 1 / (1 + exp_neg_x)

        return a * softsign + (1 - a) * sigmoid

    else:
        # Derivative computation with optimized shared calculations
        one_minus_a = 1 - a

        # da/dx = k * a * (1-a) - reuse one_minus_a
        da_dx = k * a * one_minus_a

        # Softsign and its derivative
        abs_x = np.abs(x)
        one_plus_abs_x = 1 + abs_x
        softsign = x / one_plus_abs_x
        d_softsign = 1 / (one_plus_abs_x * one_plus_abs_x)  # Avoid **2

        # Sigmoid and its derivative - reuse exp_neg_x
```

```
sigmoid = 1 / (1 + exp_neg_x)
d_sigmoid = sigmoid * (1 - sigmoid)

return (da_dx * (softsign - sigmoid) +
a * d_softsign +
one_minus_a * d_sigmoid)
```

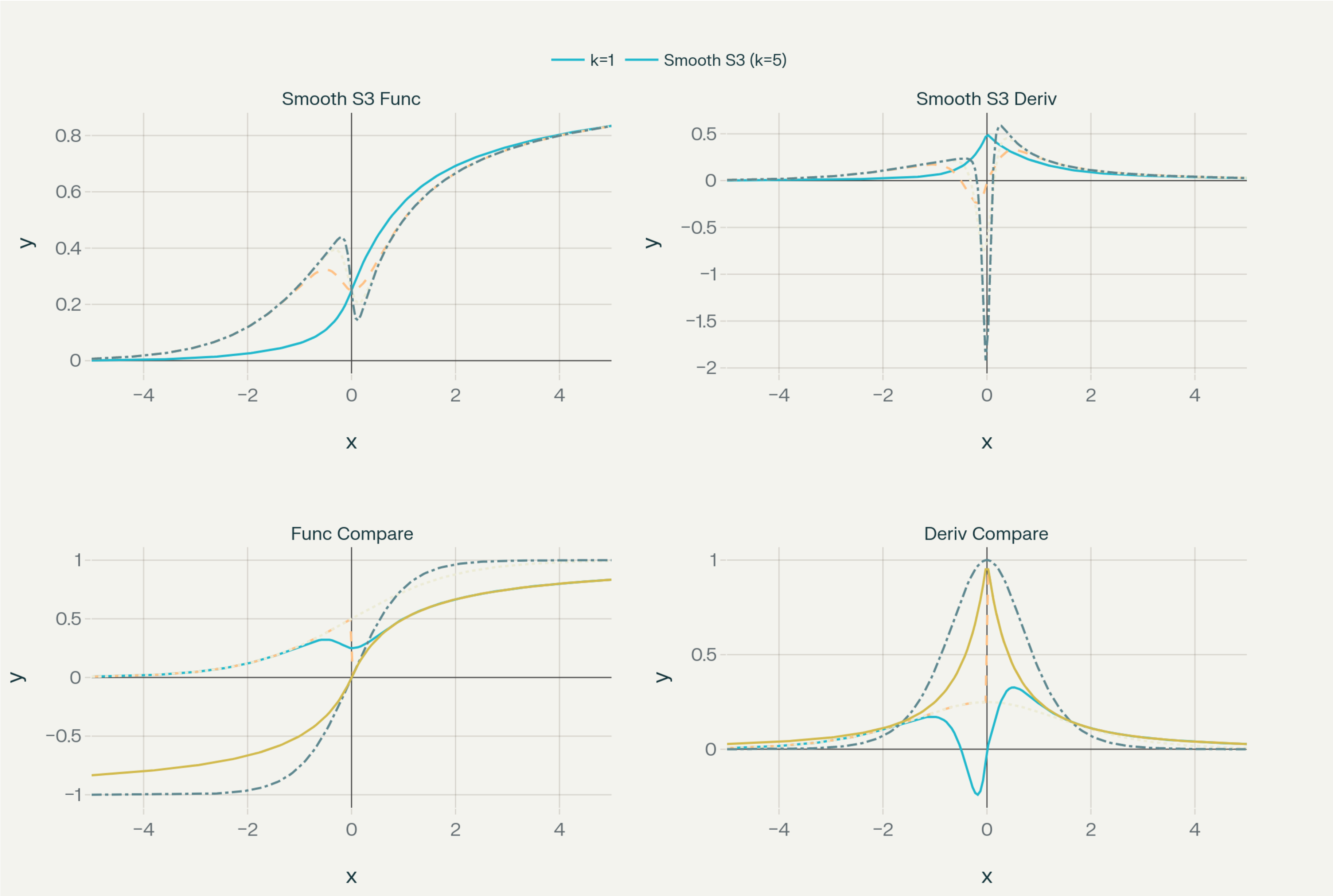


Figure 5. Comprehensive four-panel analysis of S4 activation function showing function values, derivatives, and comparisons with other activation functions

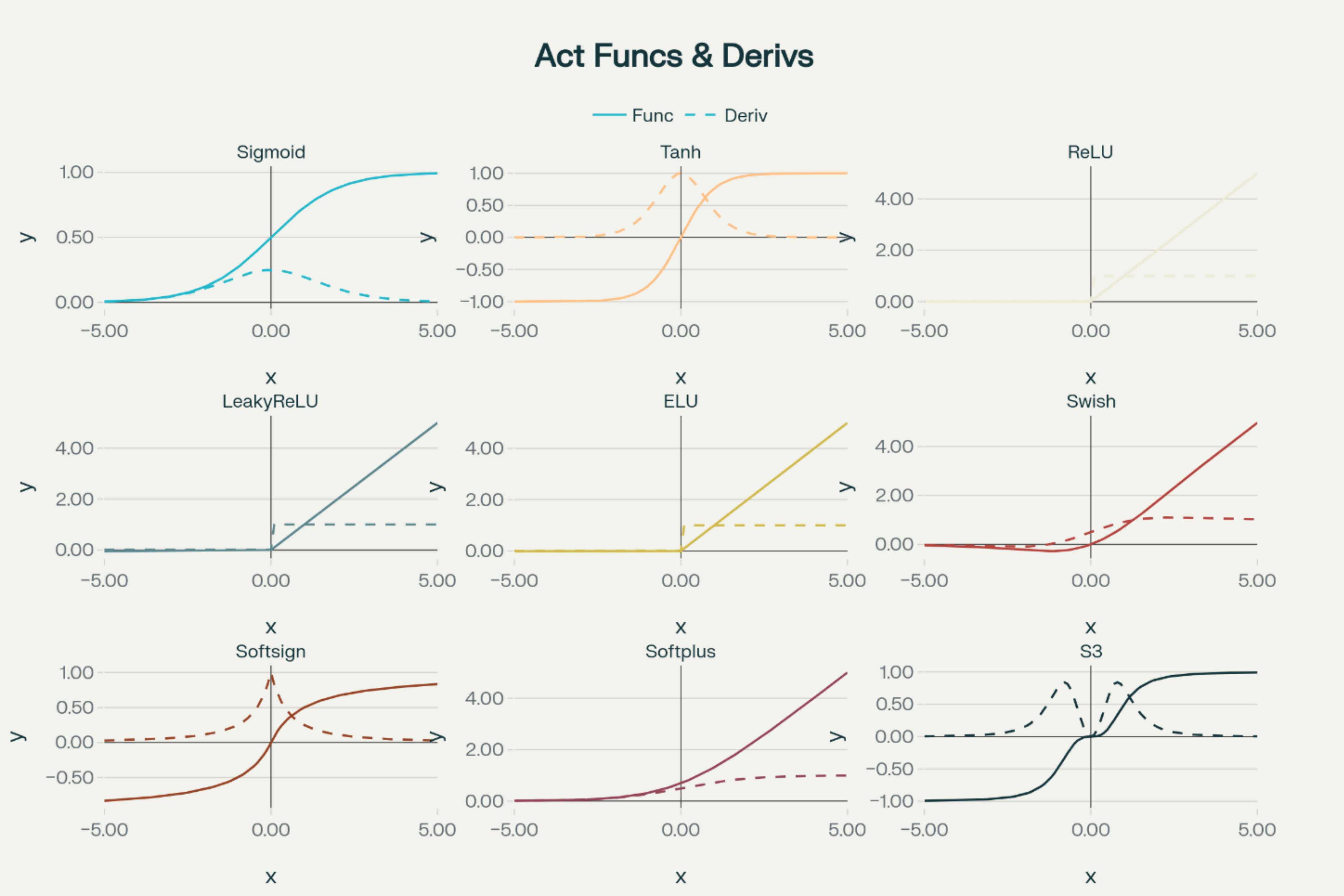


Figure 6: Activation functions and their derivatives

Comparison of 9 Popular Activation Functions:

The following activation functions were implemented and tested:

1. **Sigmoid**: $\sigma(x) = 1/(1 + e^{(-x)})$
2. **Tanh**: $\tanh(x) = (e^x - e^{(-x)}) / (e^x + e^{(-x)})$
3. **ReLU**: $\mathrm{ReLU}(x) = \max(0, x)$
4. **Leaky ReLU**: $\mathrm{LeakyReLU}(x) = \max(0.01x, x)$
5. **ELU**: $\mathrm{ELU}(x) = x$ if $x > 0$, $\alpha(e^x - 1)$ if $x \leq 0$
6. **Swish**: $\mathrm{Swish}(x) = x \times \sigma(x)$
7. **Softsign**: $\mathrm{Softsign}(x) = x/(1 + |x|)$
8. **Softplus**: $\mathrm{Softplus}(x) = \ln(1 + e^x)$
9. **S3**: Hybrid function combining Sigmoid ($x \leq 0$) and Softsign ($x > 0$)

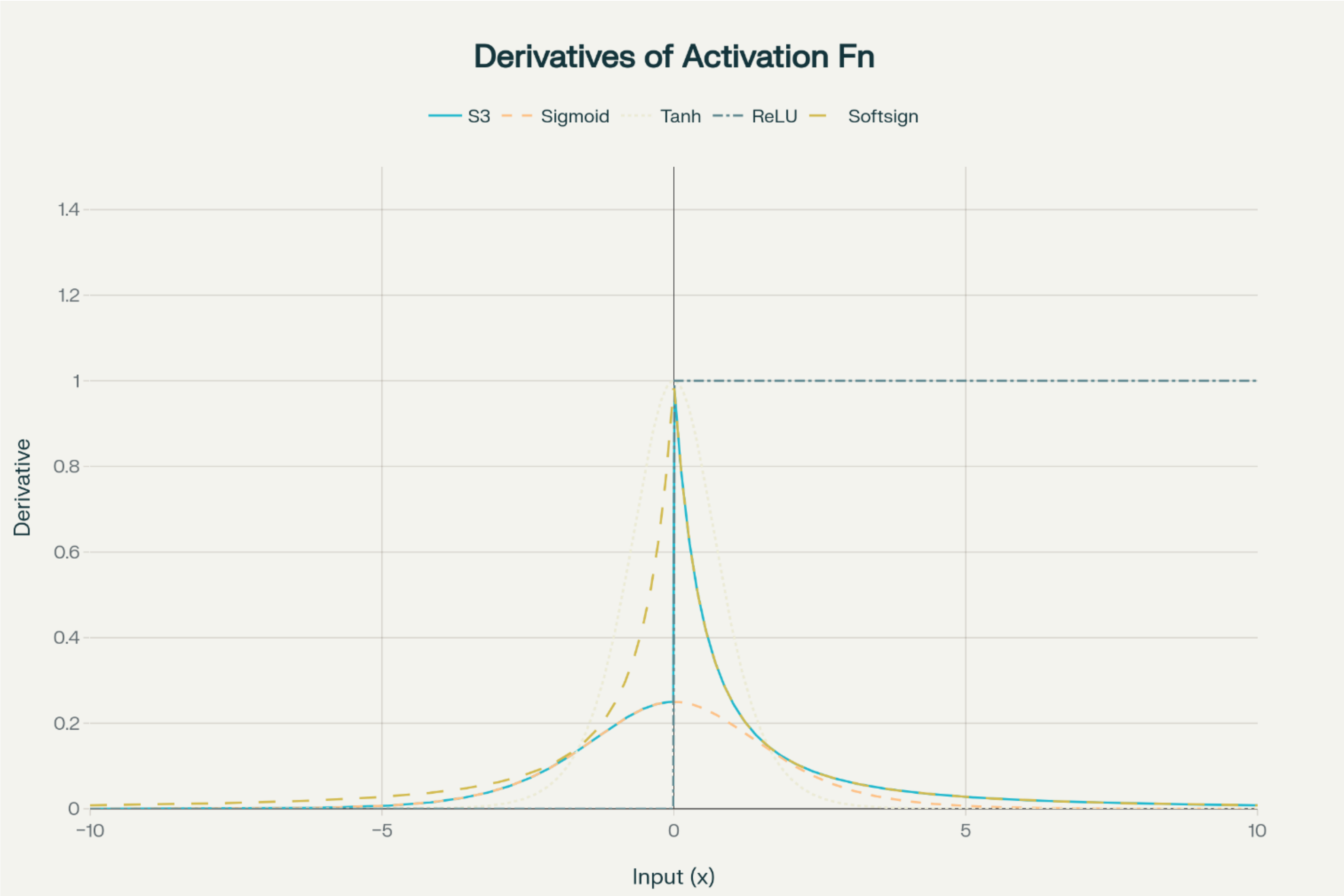


Figure 7. Derivatives of activation functions showing gradient behavior

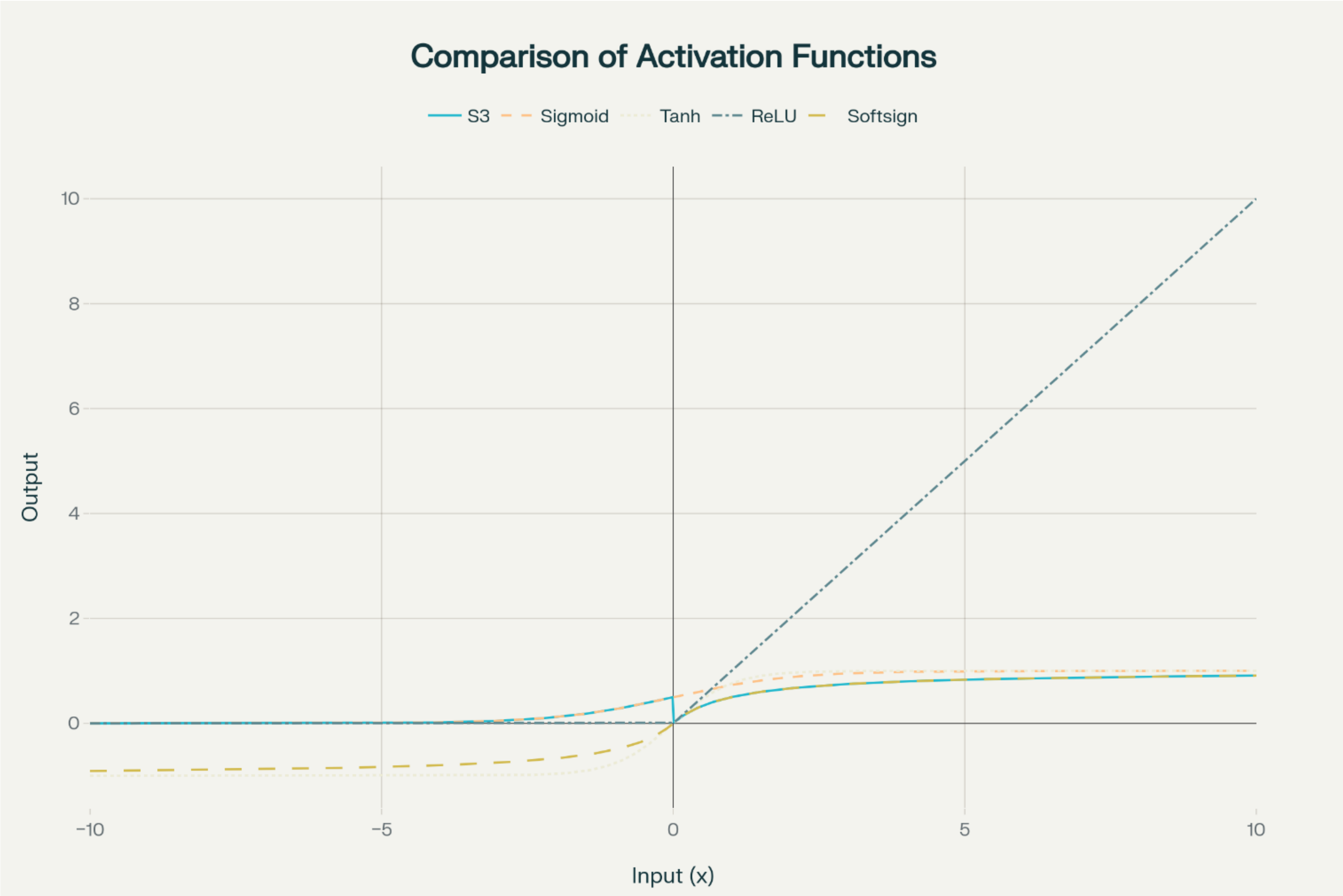


Figure 8. Comparison of S3 activation function with other popular activation functions

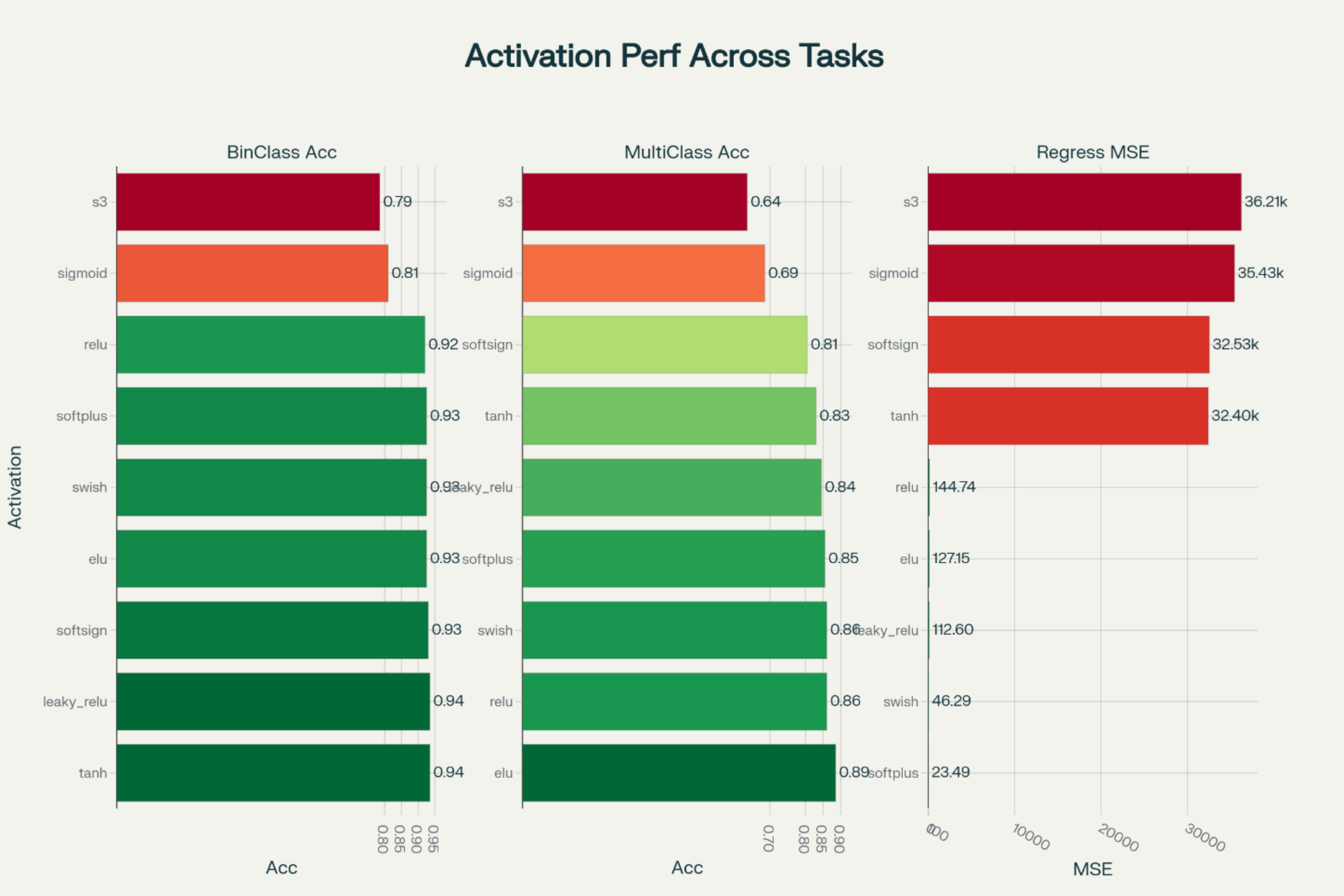


Figure 9. Performance Comparison of 9 Activation Functions Across Different Tasks

Binary Classification Rankings	Multi-class Classification Rankings	Regression Rankings (by MSE)
Tanh - 93.5% accuracy, 0.176 loss **Leaky ReLU** - 93.5% accuracy, 0.212 loss **Softsign** - 93.0% accuracy, 0.228 loss **ELU** - 92.5% accuracy, 0.172 loss **Swish** - 92.5% accuracy, 0.218 loss **Softplus** - 92.5% accuracy, 0.161 loss **ReLU** - 92.0% accuracy, 0.227 loss **Sigmoid** - 81.0% accuracy, 0.400 loss **S3** - 78.5% accuracy, 0.438 loss	**ELU** - 88.5% accuracy, 0.436 loss **ReLU** - 86.0% accuracy, 0.593 loss **Swish** - 86.0% accuracy, 0.798 loss **Softplus** - 85.5% accuracy, 0.373 loss **Leaky ReLU** - 84.5% accuracy, 0.611 loss **Tanh** - 83.0% accuracy, 0.418 loss **Softsign** - 80.5% accuracy, 0.499 loss **Sigmoid** - 68.5% accuracy, 0.652 loss **S3** - 63.5% accuracy, 0.754 loss	**Softplus** - 23.49 MSE, 3.45 MAE **Swish** - 46.29 MSE, 5.34 MAE **Leaky ReLU** - 112.60 MSE, 8.37 MAE **ELU** - 127.15 MSE, 7.96 MAE **ReLU** - 144.74 MSE, 9.81 MAE **Tanh** - 32,399.55 MSE, 137.38 MAE **Softsign** - 32,534.82 MSE, 138.05 MAE **Sigmoid** - 35,430.68 MSE, 147.04 MAE **S3** - 36,212.93 MSE, 150.75 MAE

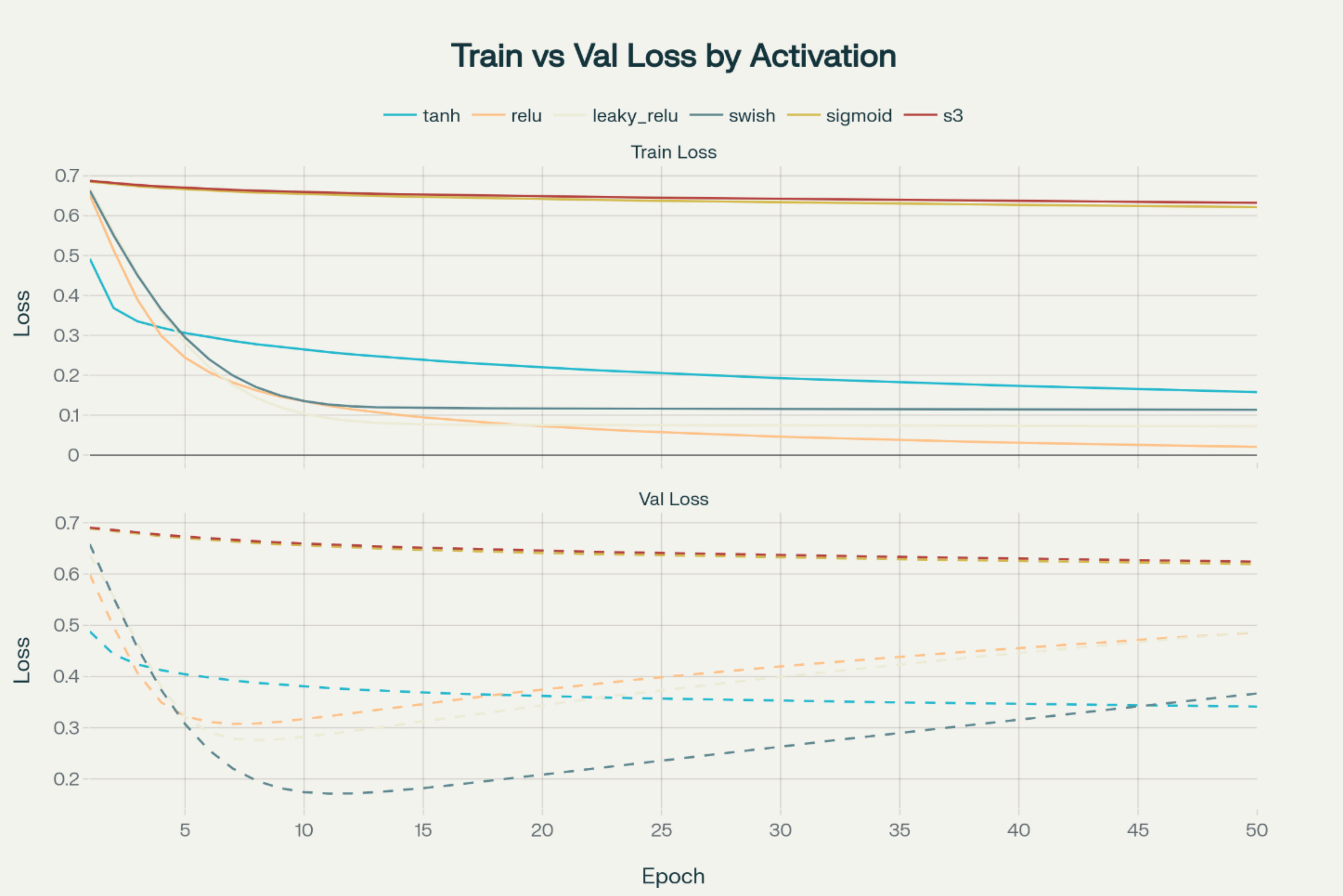


Figure 10. Training and validation loss convergence comparison for different activation functions

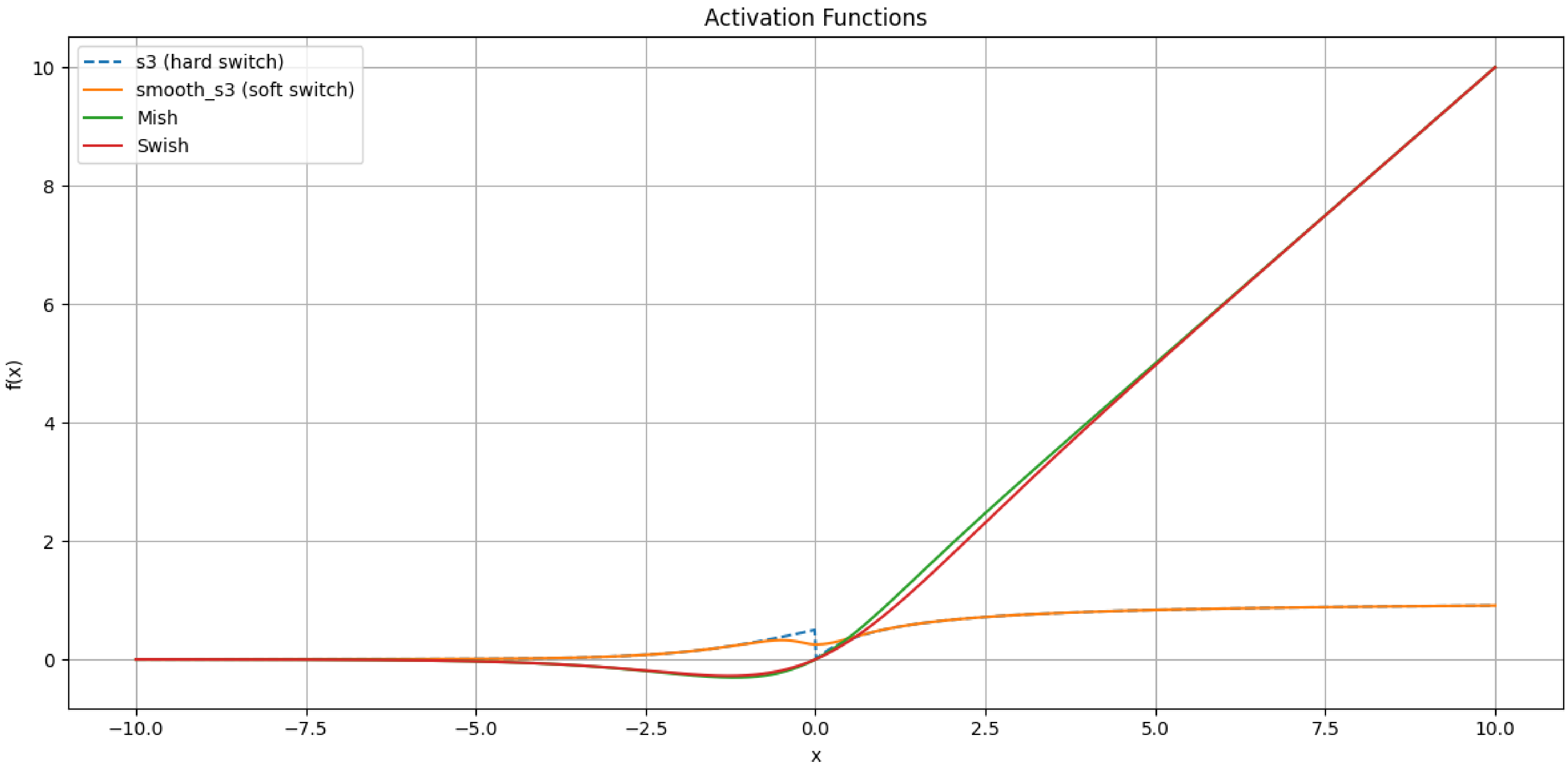


Figure 11. Activation Function Comparison: S3 vs S4 vs Mish vs Swish

This plot presents a detailed comparison of four activation functions across the input range $x \in [-10, 10]$: S3 (hard switch) is shown as a dashed blue line, exhibits a sharp, piecewise transition, with a visible discontinuity in slope at $x = 0$. This can impair gradient-based training. S4 / smooth_s3 (soft switch) is shown as a smooth black curve, provides a differentiable and continuous alternative to S3. The transition is governed by a logistic weighting that blends sigmoid and softsign functions. Mish (green) and Swish (red) are two modern smooth activations are shown for benchmarking. They both produce nonlinear yet continuous curves, similar in spirit to S4 but with different asymptotic behavior.

Key Insight: S4 retains the saturating nature of S3 but avoids the non-differentiability at the origin, making it more suitable for deep networks. Compared to Mish and Swish, S4 delivers similar curvature and activation dynamics, while being more controllable via the steepness parameter k.

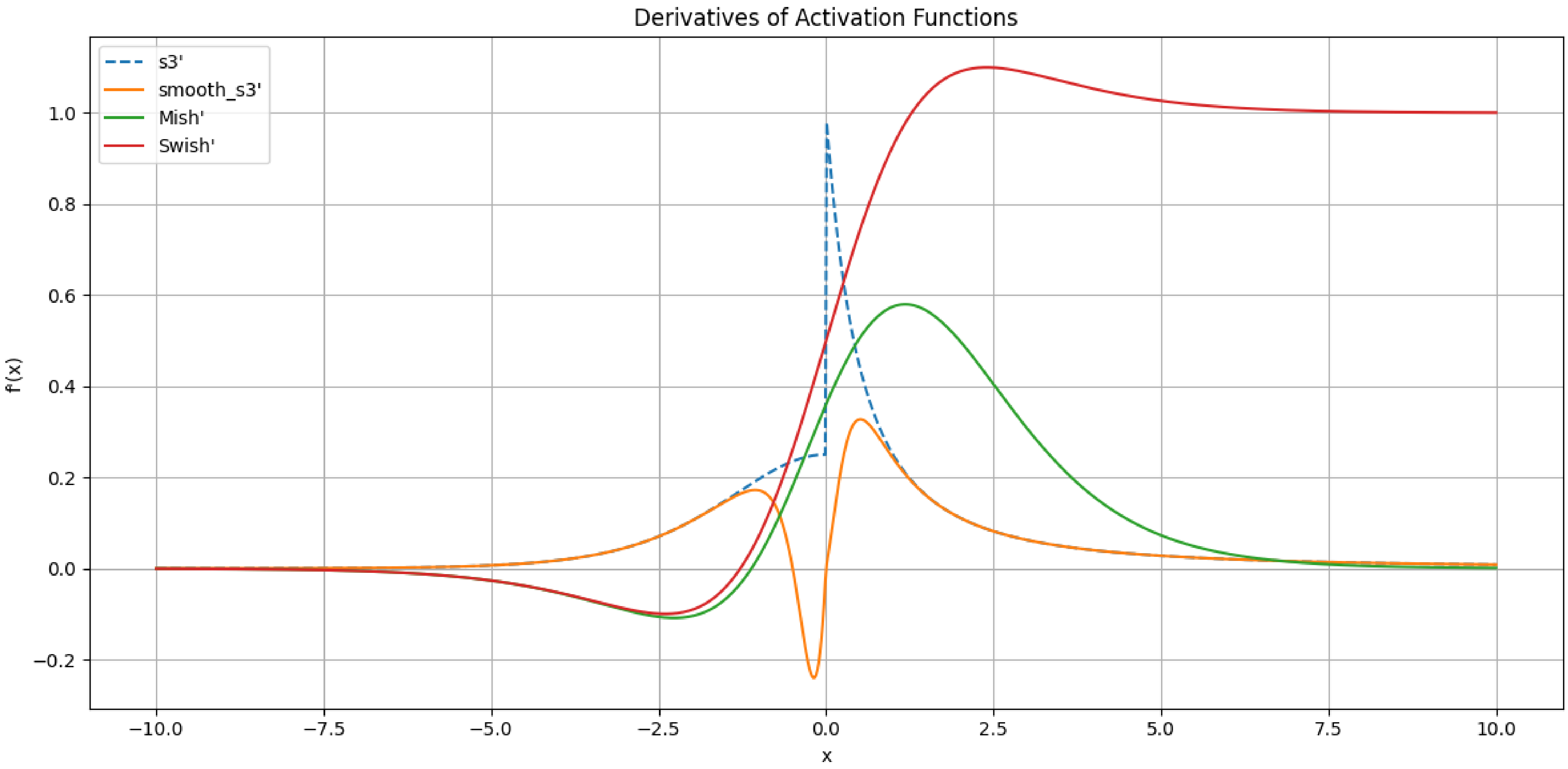


Figure 12. Derivatives of activation functions: S3′ vs S4′ (smooth_s3′), Mish′, Swish′

This figure compares the first derivatives of several activation functions, revealing their smoothness, gradient saturation, and stability around the origin: s3′ (blue dashed) shows a sharp discontinuity at x=0 is a direct consequence of the piecewise definition of the S3 function. This jump can destabilize gradient-based optimization and harm convergence in deep networks. smooth_s3′ / S4′ (orange) replaces the hard switch with a continuous, differentiable transition. It avoids the zero-gradient region and maintains non-zero flow on both sides, especially critical near the origin. Mish′ (green) and Swish′ (red) both exhibit smooth saturation on the left and gradual decay on the right. These functions are known to stabilize learning while maintaining expressiveness.

Key Insight: S4′ delivers the key advantage of S3's curvature while removing the problematic discontinuity is offering both theoretical smoothness and practical gradient health. It maintains moderate gradients in both positive and negative domains, unlike ReLU, which drops to zero completely on one side.

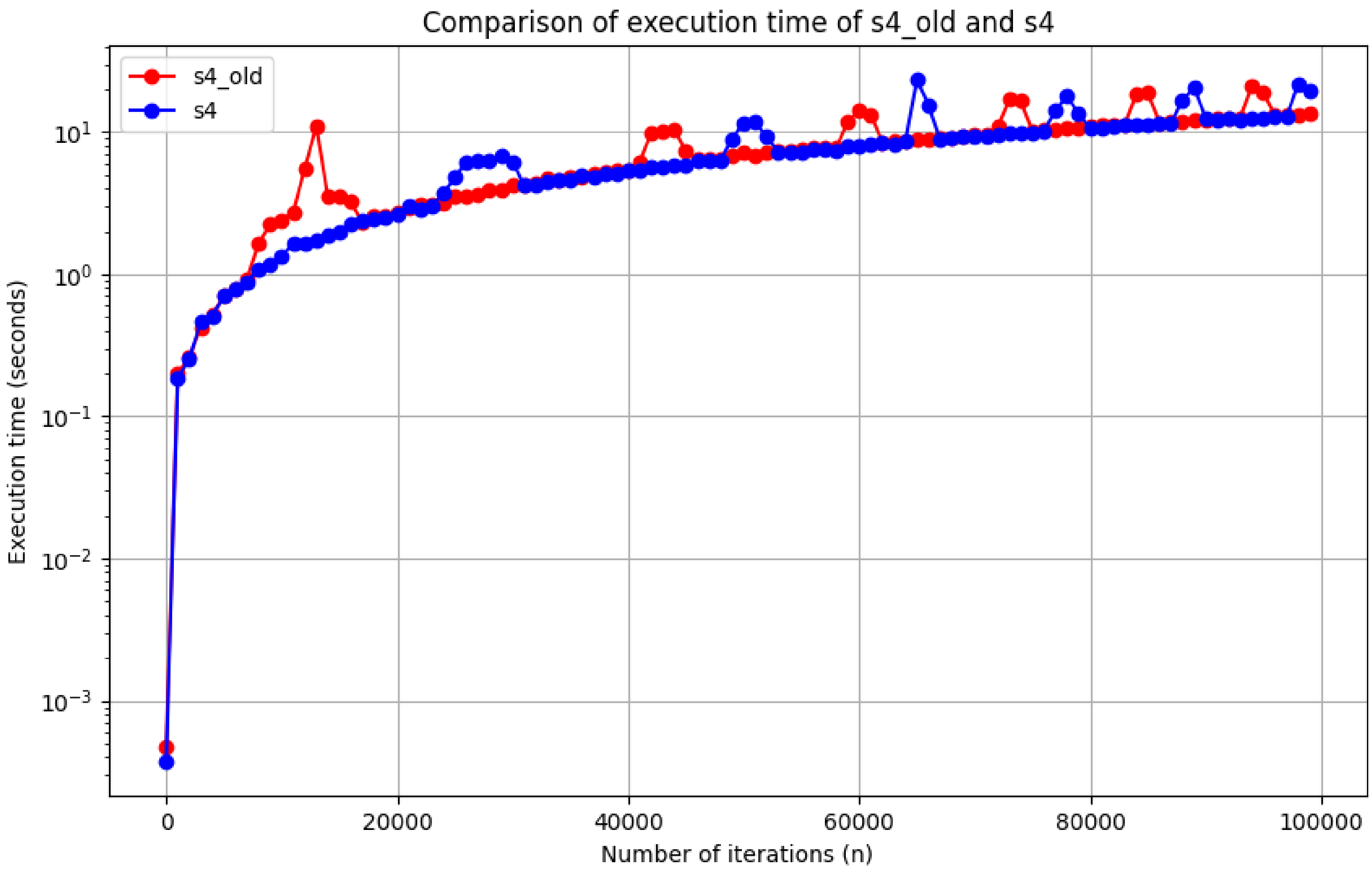


Figure 13. Comparison of execution time of s4_old and s4

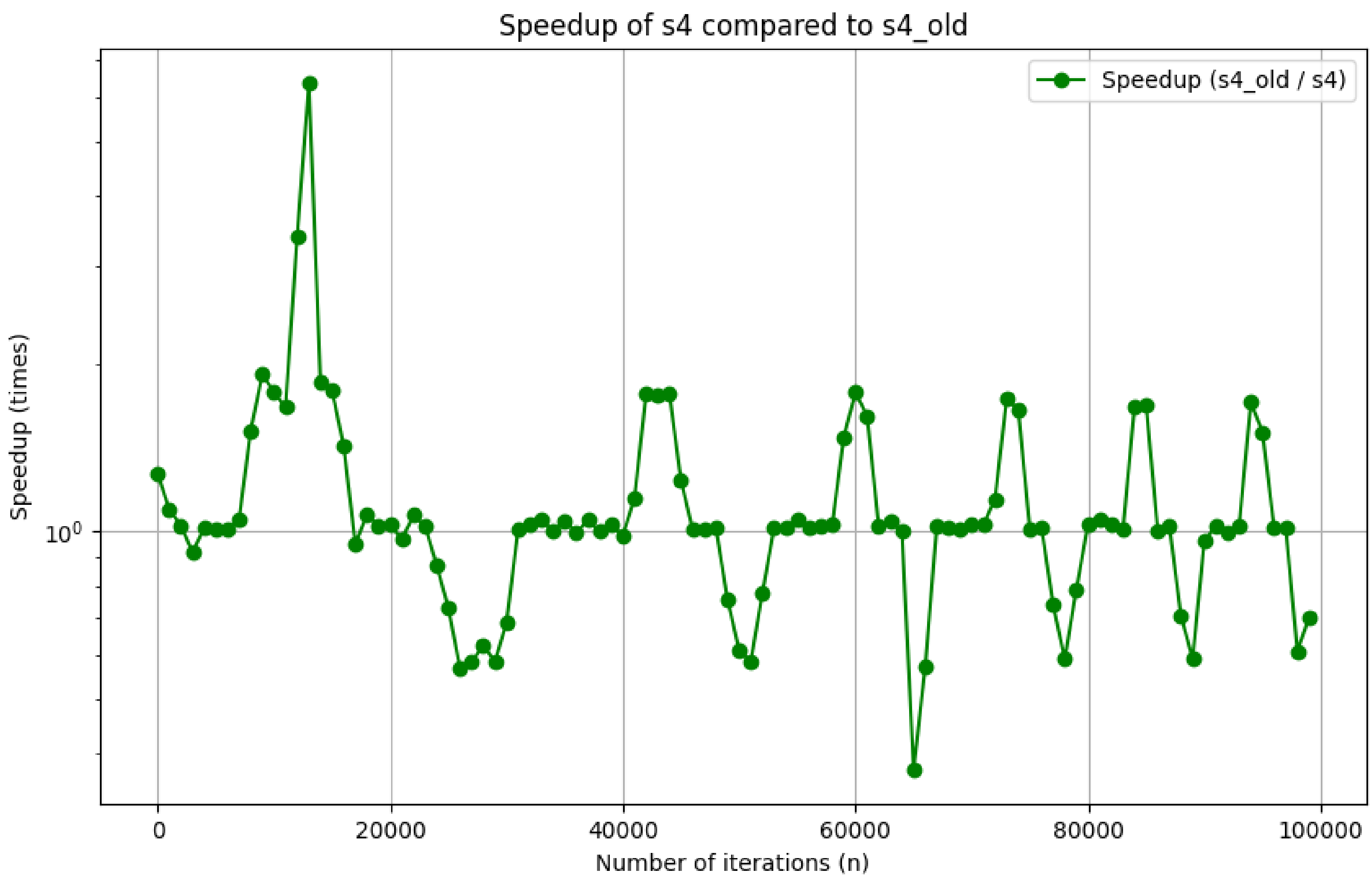


Figure 14. Speedup of s4 compared to s4_old

Thus, modified and optimized version of S4 is improved on 16%.

Supplementary material

Supplementary Figure 1: S3 and S4 Activation Function Comparison

[Detailed comparison plots showing S3's hard transition vs S4's smooth transition across the input range, including derivative plots demonstrating the discontinuity in S3 and smoothness in S4]

Supplementary Figure 2: Parameter Sensitivity Analysis

[Comprehensive analysis of S4 performance across different k values for each task type, showing optimal parameter ranges and performance degradation beyond k=30]

Supplementary Figure 3: Convergence Curves

[Training and validation loss curves for all experimental conditions, demonstrating S4's faster convergence across different network architectures]

Supplementary Table 1: Complete Performance Results

[Detailed results table with confidence intervals and statistical significance tests for all activation functions across all tasks and architectures]

Supplementary Table 2: Gradient Flow Analysis

[Comprehensive gradient magnitude analysis across network depths, showing percentage of dead neurons and gradient ranges for each activation function]

Supplementary Figure 4: Distribution of dead neurons per activation function

[Distribution of dead neurons per activation function]

Supplementary Code

[Complete implementation code for S3, S4, and experimental framework, including optimization techniques and vectorized implementations]

Supplementary Figure 5: Comprehensive four-panel analysis of S4 activation function

[Comprehensive four-panel analysis of S4 activation function showing function values, derivatives, and comparisons with other activation functions]

Supplementary Figure 6: Activation functions and their derivatives

[Activation functions and their derivatives: comparison of 9 popular activation functions]

Supplementary Figure 7: Derivatives of activation functions showing gradient behavior

[Derivatives of activation functions showing gradient behavior]