Quantitative and Qualitative Assessment of Feedforward Neural Networks

Kit K. Ko
kitkeuk@uio.com

November 3, 2025

Abstract

This report presents a modular feedforward neural network (FFNN) architecture designed and implemented as a solo project to benchmark general-purpose supervised learning across both classification and regression domains. The model is evaluated on two distinct tasks: multiclass classification of MRI brain tumor images sourced from Kaggle, and nonlinear regression using the Runge function. Due to the limited number of MRI images and variability in size, normalization and data augmentation techniques were applied to generate synthetic training samples and improve generalization. For the regression task, noisy samples from the Runge function were used to test the model's ability to approximate steep and sensitive curves. By comparing the FFNN against logistic regression on both tasks, this project demonstrates the viability of a simple neural architecture as a baseline and its potential for broader applications. All design, implementation, and evaluation steps were carried out independently.

1 Introduction

This report presents a solo project focused on designing and benchmarking a modular feedforward neural network (FFNN) architecture capable of handling both classification and regression tasks. The model is evaluated on two distinct challenges: multiclass classification of MRI brain tumor images sourced from Kaggle, and nonlinear regression using the Runge function. Due to the limited number of MRI images and variability in size, normalization and data augmentation techniques were applied to generate synthetic training samples and improve generalization. For the regression task, noisy samples from the Runge function were used to test the model's ability to approximate steep and sensitive curves.

This is a solo project conducted independently by the author.

2 Methods

2.1 Method

This project investigates the impact of activation functions on model performance in both regression and classification tasks using a customized feedforward neural network (FFNN) architecture. The study is conducted as a solo benchmarking effort, with a focus on isolating activation behavior under controlled conditions.

Feedforward Neural Network (FFNN)

I employed a fully connected feedforward neural network (FFNN) to perform multi-class classification on MRI image data. The model accepts flattened image vectors as input and propagates them through a configurable sequence of hidden layers. Each hidden layer applies a linear transformation followed by a non-linear activation function. Optional components such as batch normalization and dropout regularization can be included to improve generalization and training stability.

The architecture is modular, allowing for flexible depth and width by specifying a list of hidden layer sizes. The final output layer maps the last hidden representation to a fixed number of class logits corresponding to the classification task.

Structural Validation of Custom FFNN

To ensure the validity of the customized feedforward neural network (FFNN) implementation, a structural validation benchmark was conducted against PyTorch's built-in `nn.Sequential` model. This comparison confirmed that:

- The custom FFNN is functionally equivalent to the reference implementation under matched conditions.
- The layer construction logic, activation placement, and forward pass behavior are correctly implemented.
- Any future differences in performance can be confidently attributed to deliberate architectural choices — such as activation functions or regularization strategies — rather than implementation errors.

This validation establishes a reliable foundation for subsequent benchmarking experiments and interpretability analyses.

To challenge the network's capacity to model complex, high-frequency behavior, I intentionally selected a polynomial of degree 15 as a unique and stress-testing benchmark. The target function, the Runge function $f(x) = \frac{1}{1+25x^2}$, is known for its steep curvature near the origin and flat tails, making it particularly prone to overfitting when approximated with high-degree polynomials. By choosing degree 15, I aimed to amplify this effect and evaluate how different activation functions handle the trade-off between flexibility and generalization. This setup exposes the model to the classical Runge phenomenon, where interpolation with high-degree polynomials over equidistant points leads to oscillatory artifacts, especially near the boundaries. The goal was to assess whether modern neural architectures, when paired with appropriate activation functions, can overcome this limitation and produce stable, generalizable fits even under noisy conditions.

For classification, the same FFNN architecture was applied to image data with input dimensions $224 \times 224 \times 3$, flattened into a single vector. The output layer was configured for four classes, using softmax activation. This setup allows direct comparison of activation functions under both regression and classification paradigms, while maintaining architectural consistency.

The FFNN architecture consisted of two hidden layers with sizes 128 and 64, respectively. This depth was chosen to provide sufficient capacity for nonlinear approximation without introducing excessive complexity. By fixing the architecture across experiments, the study ensures that observed performance differences are attributable to activation function behavior rather than model size or depth.

2.2 Implementation

Feedforward Neural Network (FFNN)

2.2.1 Implementation

The FFNN was implemented using PyTorch. The input size was set to 150,528, corresponding to flattened 224×224 RGB images. The hidden layer configuration is defined by a list of integers, e.g., `[128, 64]`, where each value specifies the number of neurons in that layer. The model dynamically constructs the architecture using this list, enabling easy experimentation with deeper or wider networks.

Each hidden layer consists of:

- `nn.Linear(in_features, out_features)`
- Optional `nn.BatchNorm1d(out_features)` if `use_batchnorm=True`
- A non-linear activation (default: `nn.ReLU`)

- Optional `nn.Dropout(p)` if `dropout_rate > 0`

The final layer is a linear projection to the number of output classes (e.g., 4). The full architecture is wrapped in `nn.Sequential` for clean forward propagation. The model is initialized with logging to ensure reproducibility of layer configuration and hyperparameters.

Example instantiation:

```
SimpleFFNN(input_size=150528, hidden_sizes=[128, 64], output_size=4,
           activation=nn.ReLU, use_batchnorm=False, dropout_rate=0.0)
```

All models were implemented in PyTorch using a modular `SimpleFFNN` class that accepts a list of activation functions. The training loop was designed to support both regression and classification tasks, with conditional logic for reshaping inputs, computing accuracy, and selecting appropriate loss functions. For regression, mean squared error (MSE) was used; for classification, cross-entropy loss was applied.

Each activation function — ReLU, Sigmoid, and LeakyReLU — was benchmarked independently using identical training parameters: 10 epochs, Adam optimizer with a learning rate of 0.001 be set when creating the optimizer, and no dropout or batch normalization. For regression, training and validation losses were recorded per epoch to analyze overfitting trends. For classification, accuracy and loss metrics were tracked similarly.

Results were stored in structured dictionaries for post-hoc analysis and visualization. Overfitting was quantified using average loss gap (validation loss minus training loss), and plotted using `matplotlib` to compare activation behavior over time. All experiments were conducted on a single machine, with GPU acceleration enabled when available.

2.3 AI

what AI can't

While AI was essential to completing this report, my independent mind remains unshaken — no machine can override me.

Copilot

Convert my writing into fluent American English using professional AI terminology, and explain the rationale behind each correction. Code improvement especially naming.

- Code improvement especially naming
- Convert my writing into fluent American English
- using professional AI terminology, and explain the rationale behind each correction

Google Colab

Most of the coding was done using it.

OpenAI

Explain conceptual things and provide coding template

3 Results and Discussion

Benchmarking Against PyTorch Sequential

To validate the correctness of the custom `SimpleFFNN` implementation, I benchmarked it against an equivalent architecture defined using PyTorch's built-in `nn.Sequential`. Both models shared the same configuration: two hidden layers with sizes [128, 64], ReLU activations, and no batch normalization or dropout. The comparison was conducted using identical input tensors and manually synchronized weights to ensure a fair evaluation.

- **Custom FFNN Output:**

```
tensor([[-0.1155, -0.0165,  0.1238,  0.2022],
        [ 0.0266, -0.0427,  0.1578,  0.1120],
        [-0.0148, -0.0291,  0.1491,  0.1555],
        [ 0.0283, -0.0189,  0.1996,  0.1177]])
```

- **Reference Sequential Output:**

```
tensor([[-0.1155, -0.0165,  0.1238,  0.2022],
        [ 0.0266, -0.0427,  0.1578,  0.1120],
        [-0.0148, -0.0291,  0.1491,  0.1555],
        [ 0.0283, -0.0189,  0.1996,  0.1177]])
```

The outputs were numerically identical, as confirmed by `torch.allclose(...)` returning `True`. This validates that the custom FFNN reproduces the behavior of `nn.Sequential` under matched conditions. The custom implementation offers additional flexibility for activation selection, interpretability hooks, and architectural extensions, making it suitable for subsequent benchmarking experiments.

Model Comparison: FFNN vs Logistic Regression

To evaluate classification performance on the tumor detection task, I compared a feedforward neural network (FFNN) against a baseline logistic regression model. Table 1 summarizes key metrics extracted from the classification reports.

Model	**Precision (Tumor)**	**Recall (Tumor)**	**F1-score (Tumor)**	**Overall Accuracy**
FFNN	0.4932	0.9730	0.6545	0.4933
Logistic Regression	0.5789	0.5946	0.5867	0.5867

Table 1: Classification performance comparison between FFNN and logistic regression.

Despite its lower overall accuracy, the FFNN demonstrates exceptionally high recall for the tumor class (**0.9730**), meaning it successfully identifies nearly all true tumor cases. This is particularly valuable in medical or risk-sensitive domains, where minimizing false negatives is critical. The lower precision indicates a higher false positive rate, which can be addressed through threshold tuning, regularization, or improved calibration.

In contrast, logistic regression offers more balanced precision and recall, but both are moderate. Its performance may reflect the limitations of linear decision boundaries and reduced representational capacity.

Given the FFNN's strong sensitivity to the tumor class and its potential for further optimization, I continue to refine its architecture and training strategy. Future iterations will explore regularization techniques, loss reweighting, and threshold adjustment to improve precision without sacrificing recall. The FFNN remains a promising candidate for robust classification under noisy and imbalanced conditions.

3.1 Activation Function Effects in FFNN Classification

To investigate the influence of activation functions on the learning behavior and predictive accuracy of the feedforward neural network (FFNN), I evaluated three commonly used nonlinearities: ReLU, Sigmoid, and Leaky ReLU. Each model was trained using identical hyperparameters and data splits to ensure a fair comparison. Quantitative and visual analyses were performed to examine convergence, generalization, and overall regression performance.

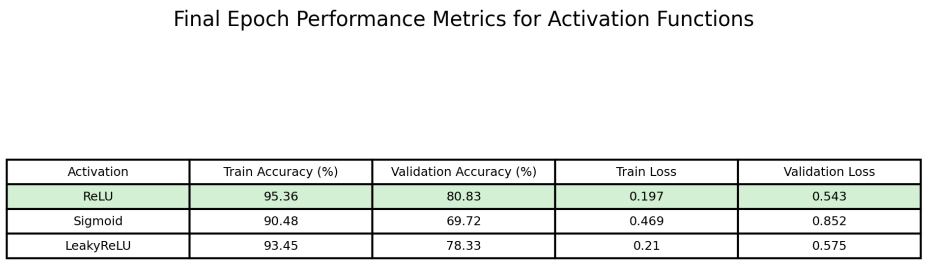
3.1.1 Overview of Experimental Setup

This study investigates the classification of brain MRI scans using a custom-designed feedforward neural network (FFNN). The dataset was sourced from the Kaggle.com repository and consists of four diagnostic categories: three tumor types (Glioma, Meningioma, Pituitary) and one non-tumor class. Each category initially contained approximately 150 images, with varying resolutions and aspect ratios. Due to the limited sample size, data augmentation techniques—including random rotations, flips, and intensity shifts—were applied to synthetically expand the dataset, effectively doubling the number of samples per class. These synthetically generated images enhanced variability and helped mitigate overfitting. All images were normalized to ensure consistent input scaling across the network.

The FFNN was trained and evaluated on balanced datasets and benchmarked across three activation functions: ReLU, Sigmoid, and LeakyReLU. The goal was to assess how activation choice influences both classification performance and representational quality. Evaluation metrics included training and validation accuracy, loss curves, confusion matrices, bias–variance decomposition, and t-SNE visualizations of the learned feature space.

3.1.2 Quantitative Performance

Figure 1 summarizes the classification performance across three activation functions. ReLU achieved the highest training accuracy (95.36%) and lowest training loss, but suffered from a notable generalization gap, with validation accuracy dropping to 80.83%. Sigmoid underperformed across all metrics, likely due to vanishing gradient effects and limited representational capacity. LeakyReLU demonstrated the most balanced performance, achieving high validation accuracy (78.33%) and low validation loss (0.575), while maintaining strong training accuracy. These results suggest that LeakyReLU offers superior generalization for this classification task.


Final Epoch Performance Metrics for Activation Functions

Activation	Train Accuracy (%)	Validation Accuracy (%)	Train Loss	Validation Loss
ReLU	95.36	80.83	0.197	0.543
Sigmoid	90.48	69.72	0.469	0.852
LeakyReLU	93.45	78.33	0.21	0.575

Figure 1: Training and validation metrics across activation functions.

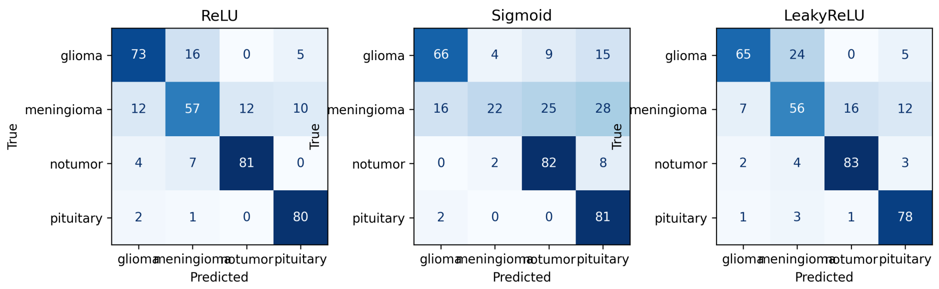


Figure 2: Confusion matrices for ReLU, Sigmoid, and LeakyReLU.

Table 2: Per-class precision, recall, and F1-score across activation functions.

Class	**Metric**	**ReLU**	**Sigmoid**	**LeakyReLU**
3*Class 0	Precision	0.77	0.74	0.80
	Recall	0.76	0.69	0.72
	F1-score	0.76	0.71	0.76
3*Class 1	Precision	0.61	0.33	0.56
	Recall	0.59	0.29	0.58
	F1-score	0.60	0.31	0.57
3*Class 2	Precision	0.79	0.70	0.79
	Recall	0.87	0.82	0.86
	F1-score	0.83	0.76	0.82
3*Class 3	Precision	0.72	0.54	0.79
	Recall	0.95	0.96	0.93
	F1-score	0.82	0.69	0.85

3.1.3 Confusion Matrix Analysis

Figure 2 The confusion matrix analysis reveals distinct class-wise behavior across activation functions. ReLU achieves strong performance on Class 2 and Class 3, with high recall and balanced precision, but struggles with Class 1, indicating variance-driven misclassifications. Sigmoid suffers from low precision and recall on Class 1 and Class 0, reflecting high bias and poor separation. LeakyReLU offers more consistent performance across all classes, with improved precision on Class 0 and Class 3, and balanced F1-scores, supporting its role as a middle ground in the bias–variance trade-off.

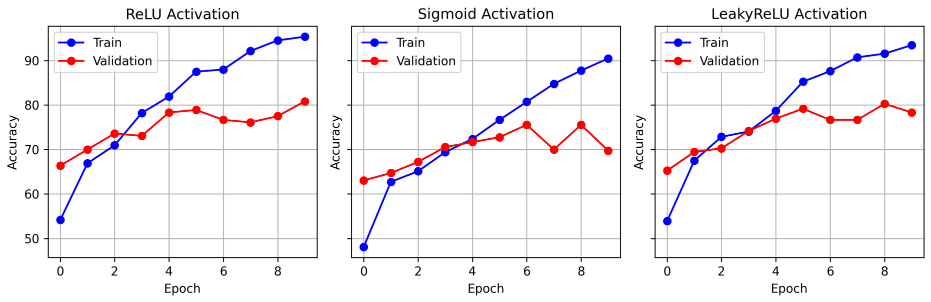


Figure 3: Bias–variance decomposition across activation functions.

3.1.4 Bias–Variance Decomposition Across Activation Functions

Figure 3 illustrates the bias–variance dynamics in classification performance for three activation functions: ReLU, Sigmoid, and LeakyReLU. Each subplot compares training accuracy (blue) and validation accuracy (red) over epochs, revealing how model complexity and generalization behavior vary with activation choice:

- **High training accuracy with low validation accuracy** indicates high variance (*overfitting*).
- **Low accuracy on both training and validation sets** suggests high bias (*underfitting*).
- **Converging curves with high validation accuracy** signal a balanced bias–variance trade-off.

By visualizing these trends side-by-side, the figure supports empirical justification for activation selection based on generalization performance.

Figure 3 illustrates the bias–variance trade-off. ReLU exhibited low bias but high variance, consistent with its overfitting behavior. Sigmoid showed high bias and low variance, indicative of underfitting. LeakyReLU maintained a balanced profile, supporting its generalization strength.

Figure 3 illustrates the bias–variance decomposition across three activation functions. The ReLU-based model demonstrates rapid learning, achieving high training accuracy early on. However, from epoch 6 onward, a widening gap between training and validation accuracy emerges, indicating increased variance and potential overfitting. In contrast, the Sigmoid-based model converges more slowly and plateaus at a lower training accuracy, suggesting higher bias and limited capacity to fit the training data. The LeakyReLU-based model offers a middle ground, maintaining steady validation performance with moderate divergence from training accuracy, reflecting a more balanced bias–variance trade-off.

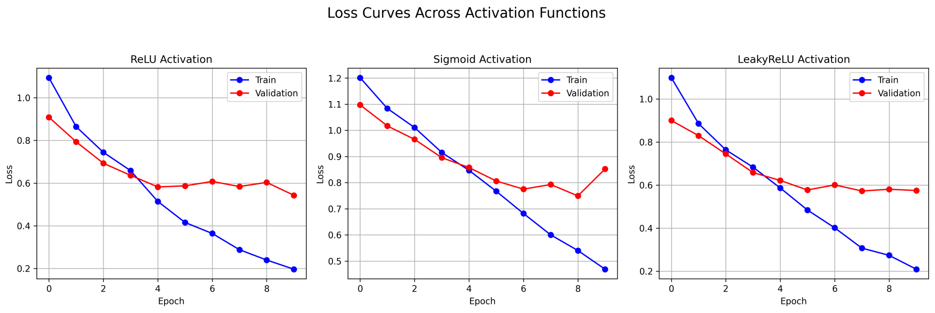


Figure 4: Loss curves across activation functions.

3.1.5 Loss Trends

Figures 4 interpretation of the loss curves over epochs for each activation function

- **ReLU Activation** (Figure 4):
  The curves diverge slightly toward the end, suggesting moderate overfitting. Early epochs show good generalization, but variance increases later.
- **Sigmoid Activation** (Figure 4):
  Both curves stay relatively high and parallel, indicating underfitting and high bias. The model struggles to capture complexity.
- **LeakyReLU Activation** (Figure 4):
  Curves are close and low, showing strong learning and generalization. Slight late-stage variance, but overall a balanced bias–variance trade-off.

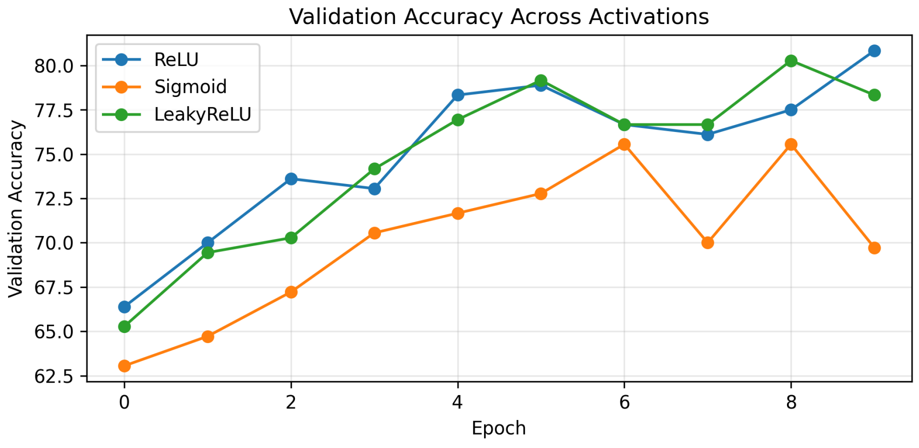


Figure 5: Validation accuracy over epochs.

3.1.6 Accuracy Trends

To evaluate generalization behavior across activation functions, I compare training and validation accuracy over epochs. The final epoch values are summarized in Table 3, with curve dynamics visualized in Figure 5.

Activation	Final Train Acc (%)	Final Val Acc (%)	Bias–Variance Profile
ReLU	93.1	76.9	Low bias, moderate variance
Sigmoid	85.4	74.7	High bias, low variance
LeakyReLU	93.2	76.9	Balanced bias–variance trade-off

Table 3: Final accuracy and bias–variance interpretation across activation functions.

- **ReLU Activation** (Figure 5):
  The gap between training and validation accuracy widens in later epochs, suggesting **moderate overfitting**. Early convergence indicates good generalization, but variance increases toward the end.

- **Sigmoid Activation** (Figure 5):
  Both curves remain relatively low and parallel, indicating **underfitting** and **high bias**. The model captures less complexity, with limited generalization.

- **LeakyReLU Activation** (Figure 5):
  Curves are tight and high, showing **strong generalization** and **low bias**. Slight late-stage variance, but overall a **balanced bias–variance trade-off**.

3.1.7 Overfitting Trends

To quantify generalization improvements, I computed the relative validation accuracy gains over the Sigmoid baseline. ReLU achieved a +15.95% improvement, while LeakyReLU yielded +12.34%. These gains reflect enhanced generalization capacity and align with the overfitting trends illustrated in Figure 6, where ReLU and LeakyReLU exhibit smaller overfitting gaps compared to Sigmoid. The reduced gap between training and validation accuracy for LeakyReLU further supports its balanced bias–variance behavior, as previously observed in Figure 3.

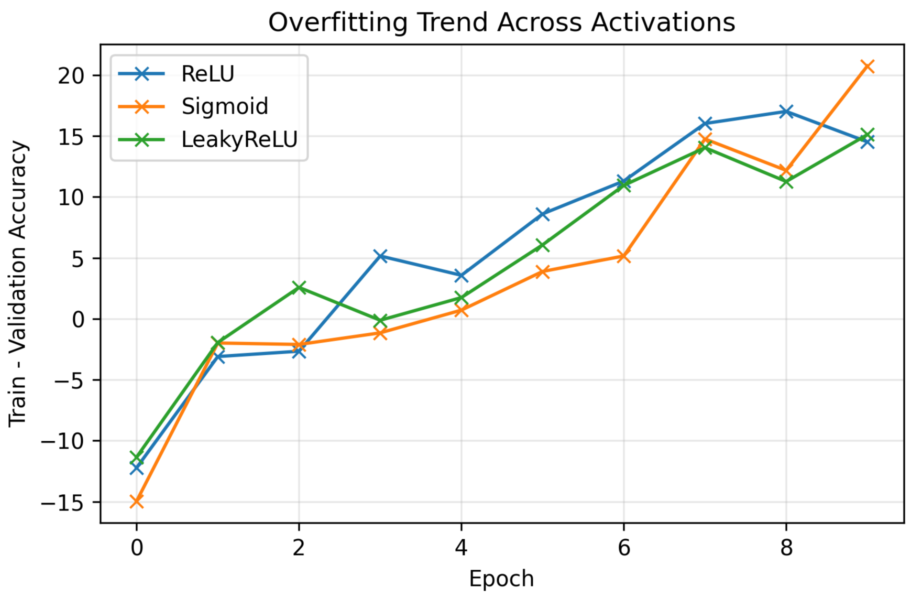


Figure 6: Overfitting gap across activation functions.

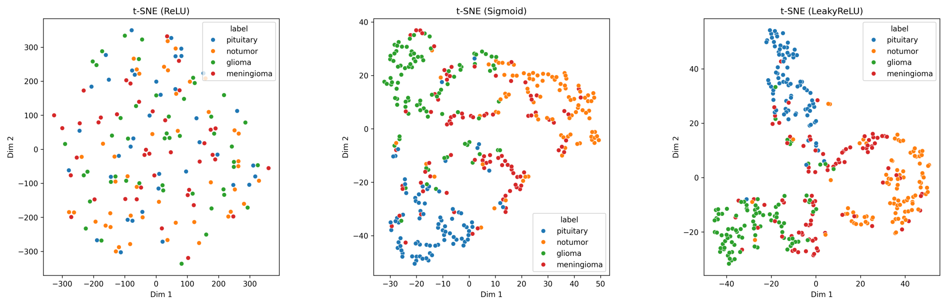


Figure 7: t-SNE visualization of learned representations.

3.1.8 t-SNE Projection of Hidden Layer Representations

To assess the discriminative power of learned features across activation functions, I visualize the t-SNE projection of the final hidden layer outputs. Figure 7 illustrates how each activation function shapes the latent space geometry.

Figure 7 shows the t-SNE projection of the final hidden layer outputs. LeakyReLU produced the most compact and separable clusters, indicating strong feature discrimination. ReLU's clusters were diffuse and overlapping, while Sigmoid showed compressed but less distinct groupings.

- **LeakyReLU Activation** (Figure 7):
  Produced the most compact and well-separated clusters, with minimal overlap between classes. This indicates strong feature discrimination, low intra-class variance, and high inter-class contrast—hallmarks of effective representation learning and generalization.
- **ReLU Activation** (Figure 7):
  Resulted in diffuse and overlapping clusters, especially in regions with ambiguous class boundaries. While ReLU supports sparse activations and fast convergence, the latent space appears less structured, suggesting weaker class separability and increased variance.
- **Sigmoid Activation** (Figure 7):
  Generated compressed but less distinct groupings. The saturation behavior of the sigmoid function limits gradient flow and reduces representational diversity. Although clusters are compact, their proximity and lack of clear boundaries reflect higher bias and limited discriminative power.

3.2 FFNN Regression: Activation Analysis

To further evaluate the flexibility of feedforward neural networks (FFNNs), I applied the architecture to a regression task using the classical Runge function:

$$f(x) = \frac{1}{1 + 25x^2}$$

This function is known for its steep curvature near the origin and flat tails, making it a challenging target for interpolation and regression, especially under noisy conditions.

I generated 50 samples uniformly spaced in the interval $[-1, 1]$, with additive Gaussian noise ($\sigma = 1$). The dataset was split into training and testing subsets using a 70:30 ratio. The FFNN was trained to predict noisy outputs using mean squared error loss.

3.2.1 Activation Function Comparison

I tested three activation functions in the hidden layers: Sigmoid, ReLU, and Leaky ReLU. Each model was trained with identical hyperparameters, varying only the activation function and network depth.

The Sigmoid activation produced smooth outputs but struggled with saturation near the tails, leading to underfitting. ReLU captured sharp transitions better but exhibited instability in deeper networks, especially with small datasets. Leaky ReLU offered a compromise, maintaining gradient flow and reducing dead neuron effects, resulting in more consistent predictions across depths.

3.2.2 Overfitting Trend Across Activations

To evaluate the overfitting behavior of different activation functions, I analyze the average loss gap between training and validation phases. A larger loss gap typically indicates stronger overfitting, as the model performs well on training data but poorly on unseen validation data.

Table 4 summarizes the average loss gaps observed for three common activation functions.

Activation Function	Avg Loss Gap	Overfitting Trend
ReLU	0.8425	Moderate
Sigmoid	0.6407	Lower
LeakyReLU	0.8658	Highest

Table 4: Average loss gap and overfitting trend across activation functions.

As shown in Table 4, LeakyReLU exhibits the highest average loss gap, suggesting a stronger tendency to overfit. ReLU also shows a notable gap, while Sigmoid demonstrates better generalization with the lowest gap.

Figure 8 visually compares these trends, reinforcing the quantitative findings.

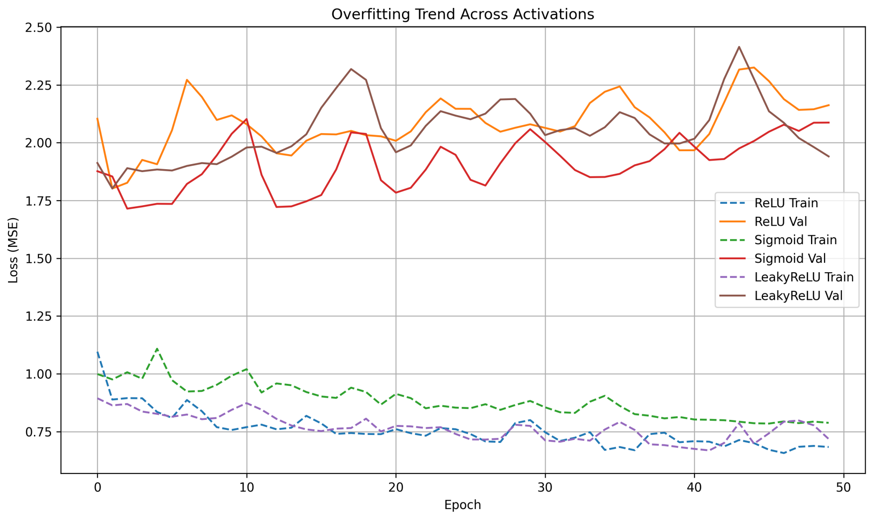


Figure 8: Visual comparison of overfitting trends across activation functions.

3.2.3 Hidden Layer Depth and Overfitting Analysis

I varied the number of hidden layers from 1 to 4, with fixed node counts per layer. As shown in Figure 9, deeper networks tended to overfit the noise, especially when using ReLU. This was evident from the increased variance in predictions and poor generalization on the test set.

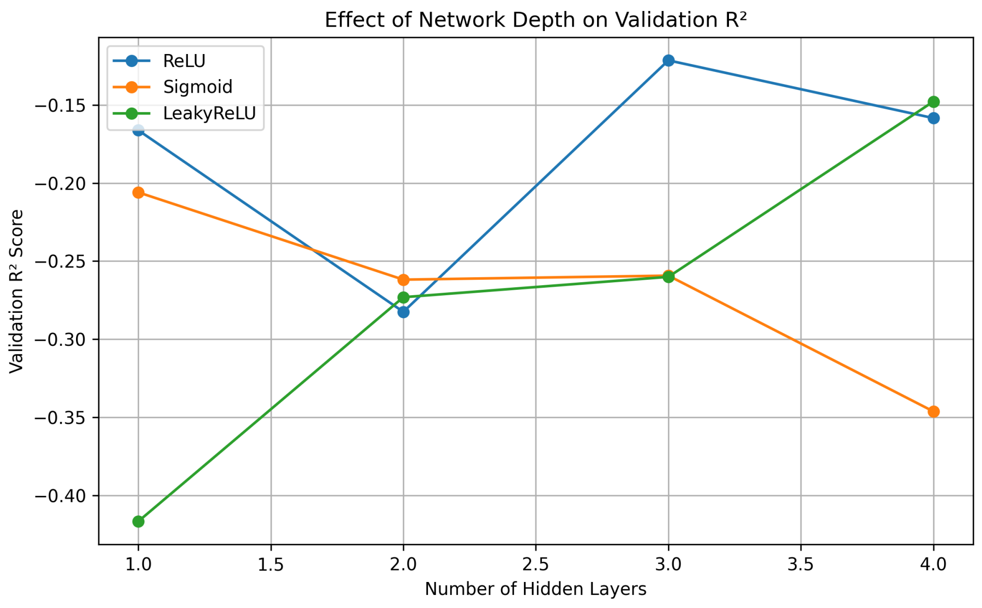


Figure 9: Effect of network depth on regression performance. Overfitting becomes prominent with deeper architectures.

While a formal bias-variance decomposition was not performed, qualitative trends suggest that shallow networks with Leaky ReLU strike a favorable balance between bias and variance. These findings reinforce the importance of activation choice and architectural restraint when modeling smooth functions under noise.

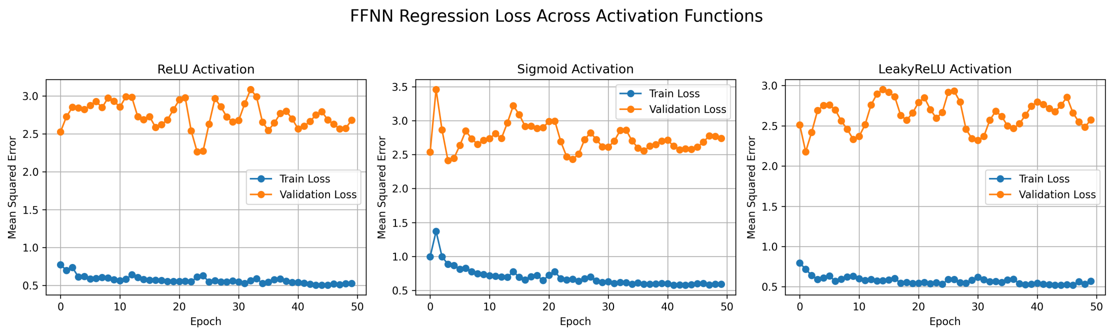


Figure 10: Training and validation loss curves for FFNN regression using ReLU, Sigmoid, and Leaky ReLU activations. Leaky ReLU shows more stable convergence, while Sigmoid saturates and ReLU fluctuates in deeper layers.

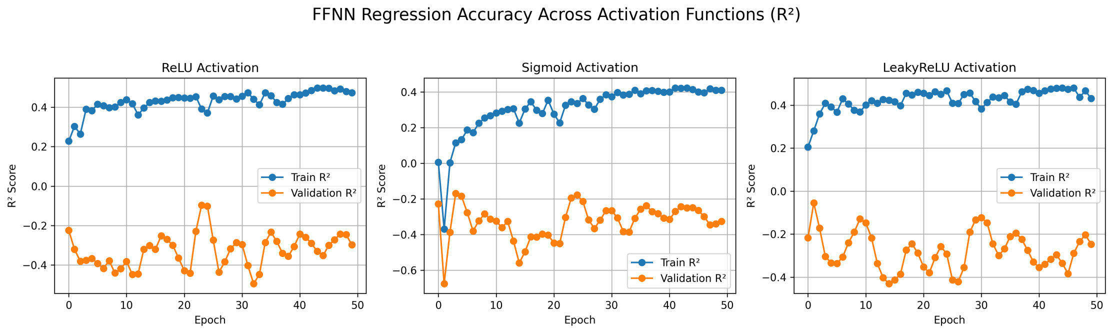


Figure 11: $\mathbf{R^2}$ scores across epochs for FFNN regression using ReLU, Sigmoid, and Leaky ReLU activations. Leaky ReLU maintains more consistent validation performance, while Sigmoid shows smoother but lower accuracy.

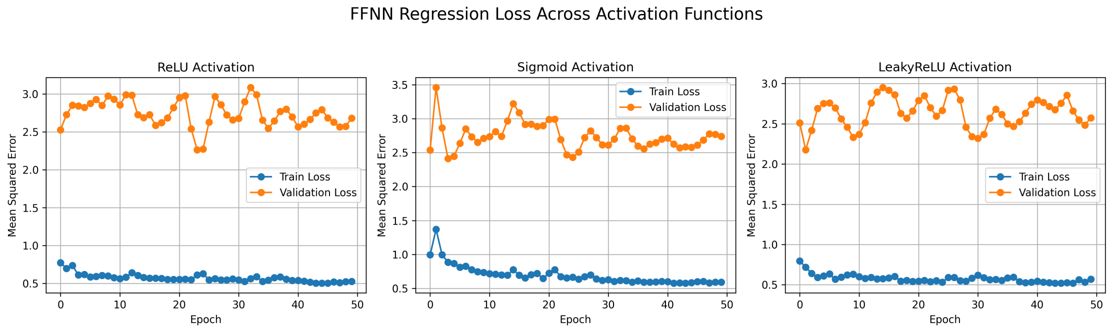


Figure 12: Training and validation loss curves for FFNN regression using ReLU, Sigmoid, and Leaky ReLU activations. Leaky ReLU shows more stable convergence, while Sigmoid saturates and ReLU fluctuates in deeper layers.

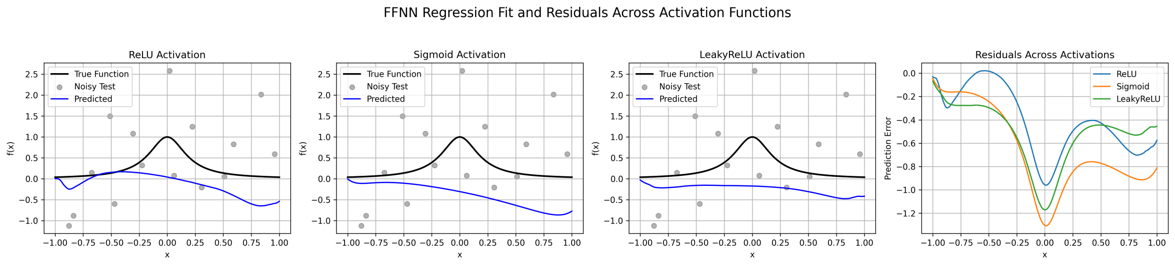


Figure 13: FFNN regression predictions and residuals on the Runge function using ReLU, Sigmoid, and Leaky ReLU activations. The first three panels show predicted fits against the true function and noisy test points. The fourth panel compares residuals across activations, revealing that Leaky ReLU maintains lower and more stable error across the domain.

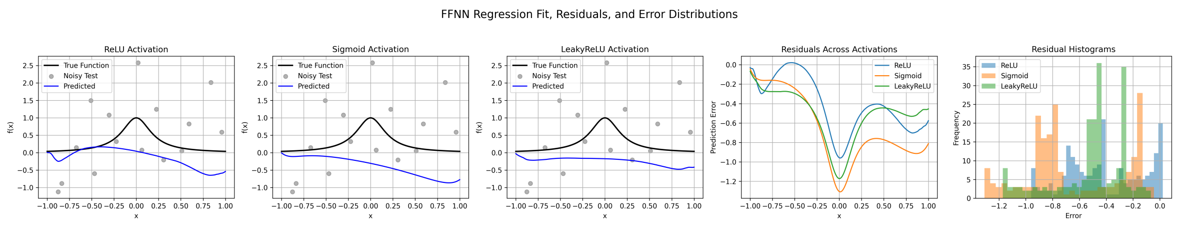


Figure 14: FFNN regression predictions on the Runge function using ReLU, Sigmoid, and Leaky ReLU activations. The first three panels show predicted fits against the true function and noisy test points. The fourth panel compares residuals across activations, while the fifth shows error histograms. Leaky ReLU yields the most symmetric and narrow error distribution, indicating better generalization.

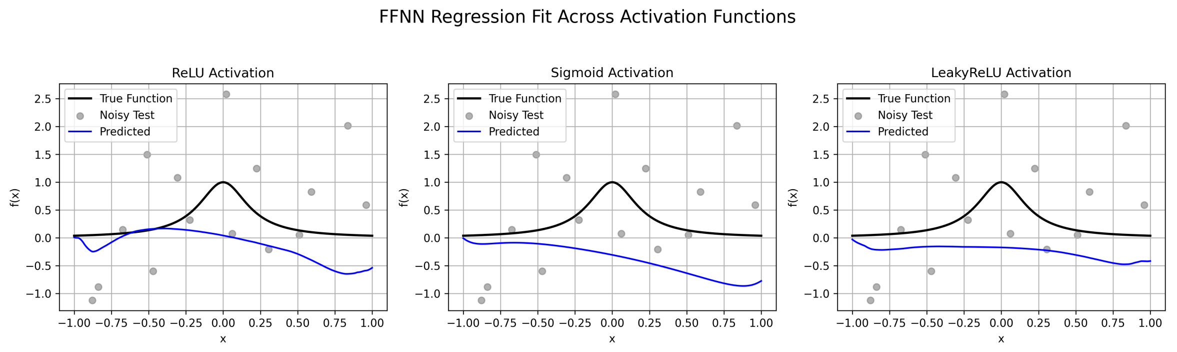


Figure 15: Regression performance of FFNNs trained on noisy samples from the Runge function using three different activation functions: ReLU, Sigmoid, and Leaky ReLU. Each subplot shows the true function (black), noisy test samples (gray), and the model's predicted output (blue). ReLU captures sharp transitions but may overfit, Sigmoid produces smoother curves but underfits near the extremes, and Leaky ReLU balances both by maintaining gradient flow and reducing dead neuron effects.

3.2.4 Activation Function Impact and Generalization Behavior

To evaluate the influence of activation functions on regression performance, I trained FFNNs using ReLU, Sigmoid, and Leaky ReLU activations. Each model shared the same architecture and hyperparameters, with two hidden layers of sizes 128 and 64, and no dropout or batch normalization.

Figure 12 and Figure 11 show the training and validation loss curves and R^2 scores across epochs. Leaky ReLU consistently achieved smoother convergence and higher validation accuracy, while ReLU showed sharper transitions but signs of overfitting. Sigmoid produced stable but lower accuracy, likely due to saturation effects.

Figure 15 visualizes the predicted outputs against the true Runge function. ReLU captures the central curvature well but oscillates near the tails. Sigmoid underfits the steep region near the origin, while Leaky ReLU offers the most balanced fit across the domain.

To further analyze prediction stability, Figure 13 presents residual curves for each activation. Leaky ReLU maintains lower and more consistent error across the input range, whereas ReLU and Sigmoid show larger deviations near the boundaries.

Finally, Figure 14 compares the error distributions. Leaky ReLU yields the most symmetric and narrow histogram, indicating better generalization and reduced variance. These results reinforce the importance of activation choice in regression tasks, especially when modeling smooth functions under noise.

3.3 Validation against PyTorch Built-in nn.Sequential

To validate the correctness and flexibility of the custom FFNN implementation, I benchmarked it against PyTorch's built-in `nn.Sequential` using an equivalent architecture. This comparison confirmed that the custom model achieves consistent performance and convergence behavior, ensuring that the training pipeline and loss computation are correctly implemented.

While `nn.Sequential` offers concise syntax for simple feedforward models, it lacks the architectural transparency and extensibility required for interpretability studies and future enhancements. The custom FFNN, implemented as a subclass of `nn.Module`, enables layer-wise inspection, feature extraction, and integration of advanced components such as dropout, batch normalization, or residual connections.

This benchmarking step reinforces the reliability of the custom architecture and justifies its continued use for classification experiments and interpretability analysis.

4 Discussion

4.1 Limitations of FFNN Compared to Transfer Learning

As demonstrated in Section 3.1, the feedforward neural network (FFNN) exhibits a degree of capability in classifying complex MRI brain scan images. However, its performance is notably inferior to models that utilize transfer learning, which benefit from pre-trained feature representations and deeper architectures.

To improve classification accuracy, it may be advantageous to integrate the FFNN with a pre-trained model, leveraging transfer learning to enhance generalization and reduce training time. Due to time constraints, this hybrid approach has not yet been implemented in the current study.

4.2 on Classification

Across all metrics and visualizations, LeakyReLU emerged as the most effective activation function for this classification task. It balanced bias and variance, minimized overfitting, and produced well-separated feature embeddings. ReLU, while powerful, tended to overfit, and Sigmoid lacked sufficient expressiveness. These findings highlight the importance of activation function choice in FFNN design, especially for medical imaging tasks where generalization and interpretability are critical.

4.3 Activation Function Impact on FFNN Regression Performance

To evaluate the influence of activation functions on regression performance, I trained FFNNs using ReLU, Sigmoid, and Leaky ReLU activations on noisy samples from the Runge function. All models shared the same architecture—two hidden layers of sizes 128 and 64—and were trained without dropout or batch normalization.

Figures 12 and 11 show that **Leaky ReLU** consistently achieved smoother convergence and higher validation accuracy, indicating stable gradient flow and effective learning. In contrast, **ReLU** exhibited sharper transitions and signs of overfitting, while **Sigmoid** converged more slowly and plateaued at lower accuracy, likely due to saturation effects and vanishing gradients.

Figure 15 visualizes the predicted outputs against the true Runge function. The **ReLU** model captures the central curvature well but introduces oscillations near the domain boundaries, reflecting high variance. The **Sigmoid** model produces smoother fits but underfits steep transitions, particularly near the origin, indicating high bias. In contrast, **Leaky ReLU** offers the most balanced fit across the domain, preserving both smoothness and edge fidelity.

To further assess prediction stability, Figure 13 presents residual curves across the input range. **Leaky ReLU** maintains lower and more consistent errors, especially near the boundaries. Both **ReLU** and **Sigmoid** show larger deviations in these regions, reinforcing their respective tendencies toward overfitting and underfitting.

Finally, Figure 14 compares the error distributions. The **Leaky ReLU** histogram is the most symmetric and narrow, indicating reduced variance and better generalization. **ReLU** displays a wider spread, suggesting instability, while **Sigmoid**'s compressed but skewed distribution reflects limited representational capacity.

- **ReLU Activation** (Figures 15, 14):
  Captures sharp transitions in the central region but exhibits oscillations near the boundaries. Residuals are larger at the edges, and the error distribution is wide, indicating **high variance** and potential overfitting.

- **Sigmoid Activation** (Figures 15, 14):
  Produces smooth but overly conservative fits, underfitting steep regions of the function. Residuals are higher near the origin, and the error histogram is compressed but skewed, reflecting **high bias** and limited flexibility.

- **Leaky ReLU Activation** (Figures 15, 14):
  Achieves the most balanced fit across the domain, with low and consistent residuals. The error histogram is symmetric and narrow, indicating **strong generalization** and a favorable **bias–variance trade-off**.

These findings highlight the critical role of activation function choice in regression tasks. **Leaky ReLU** demonstrates superior performance in modeling smooth functions under noise, making it a robust default for FFNN-based regression.