PROJECT

Project Report Submitted in partial fulfilment of the requirements

of the Course

CH-6870: Machine learning for process systems engineering

By

Abhay Sachan

CC24MTECH11004

భారతీయ సాంకేతిక విజ్ఞాన సంస్థ హైదరాబాద్
भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

M. Tech. in Climate Change

October 2024

Declaration

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honestly and integrity and have not misrepresented or fabricated or falsified any ideas, data, facts or sources in my submission. I understand that without the supervisor's permission, I should not submit this work to any documentations/conferences/publications. I understand that any violation of the above will be cause of disciplinary action by the institute and evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Abhay Sachan

CC24MTECH11004

Date: 12-10-2024

Place: IIT HYDERABAD

Index

Problem

Write a PYTHON code for the following problem.

1. Using the test function given to you, generate your own nonlinear dataset.

   a. Ensure that you have enough number of samples before training the ANN.
   b. Write the description of the data (as comments in the code and in the report).
   c. Save the data into an excel sheet.
   d. Read the data from the saved excel file and split the data into inputs and outputs. (The function values must be taken as the output and the rest columns as inputs)
   e. Normalize the data between 0 and 1 before training the ANN.
   f. Divide the data into 3 parts for training, validation, and testing (70%:15%:15%)

2. Create an artificial neural network in PYTHON with any number of nodes, any number of hidden layers and any activation function.

3. Train the neural network using Adam optimizer and MSE loss function.

4. Report the following:

   a. Mean squared error for training data R^2 values for training data.
   b. R^2 values for training data.

5. Predict the outputs for testing dataset using the trained network. Report R^2

values for the testing dataset.

6. Now train the same network using RMSProp optimizer and draw comparisons between Adam and RMSProp (report relevant metrics to justify your observations).

7. Perform an analysis by varying the number of hidden layers, number of hidden nodes, activation functions, number of epochs and sample size for training. (At least three variations in each of the parameters must be done). Plot a graph between the predicted outputs and the true outputs (for the training data and testing data) using the MATPLOTLIB package for each of the analysis performed.

Test function: f(x,y)= 3x^2 - 4xy + 2y^2 + 5

1. METHODOLOGY:

- The objective of this assignment is to develop an Artificial Neural Network (ANN) model that can learn and predict outcomes from a nonlinear dataset generated by the **Test function: f(x,y)= 3x^2 - 4xy + 2y^2 + 5**
- . The task involves:
    1. Generating a nonlinear dataset.
    2. Preprocessing the data (splitting, normalizing).
    3. Creating an ANN using different optimizers.
    4. Reporting performance metrics like Mean Squared Error (MSE) and R^2 values.
    5. Performing analysis by varying network parameters such as hidden layers, activation functions, and epochs.
- This report provides a detailed breakdown of the implementation and performance results, including comparative analysis of different configurations

2. Data Generation and Preprocessing:

- The dataset consists of inputs ‘x’ and ‘y’, uniformly sampled between -10 and 10, and an output ‘z’ derived using the equation:

  z=3x^2 - 4xy + 2y^2 + 5

- The code snippet to generate and save the dataset is as follows:

```
def generate_dataset(sample_size):
    np.random.seed(42)
    x = np.random.uniform(-10, 10, sample_size)
    y = np.random.uniform(-10, 10, sample_size)
    z = 3*x**2 - 4*x*y + 2*y**2 + 5
    data = np.column_stack((x, y, z))
    return data
```

- **Data Saving and Description:**

  The generated dataset was saved to an Excel file (nonlinear_dataset.xlsx), containing columns for X, Y, and Z. The dataset comprises 10,000 samples.

- **Data Preprocessing:**

  - **Reading Data:** The dataset was read from the Excel file using `pandas` and split into input features (`x`, `y`) and the output (`z`).

  - **Normalization:** Before training the ANN, both input features and outputs were normalized between 0 and 1 using `MinMaxScaler` from `scikit-learn` to ensure that the training process is more efficient and avoids any biases due to different feature scales.

- **Data Splitting**:

  - The normalized dataset was split into three subsets:

    - **Training Set**: 70% of the data
    - **Validation Set**: 15% of the data
    - **Testing Set**: 15% of the data

  The split was performed using train_test_split from scikit-learn.

  X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y_scaled, test_size=0.3, random_state=42)

  X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

3. Neural Network Model Creation:

A simple feedforward neural network is created using TensorFlow's Keras API. The model consists of:

- Input Layer: 2 nodes (for x and y).
- Hidden Layers: Two dense layers with ReLU activation and varying sizes (64 and 32 neurons respectively).
- I choose this configuration after running the analysis with varying parameters.
- Output Layer: A single output node predicting the value of z.

- **Model Architecture:**

- The model uses mean squared error (MSE) as the loss function, appropriate for regression tasks.
- Two optimizers are explored:
  - Adam Optimizer: An adaptive optimizer that works well for most deep learning tasks.
  - RMSProp Optimizer: Another adaptive optimizer with different decay behavior.

- **Training Process:**

The model was trained for 30 epochs using the training dataset, and validation was performed using the validation dataset.

model.compile(optimizer='adam', loss='mean_squared_error')
history_adam = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=30, verbose=0)

4. Performance Metrics for Adam Optimizer:

- **Mean Squared Error (MSE) and R^2 Values for Training:**

After training, the MSE and R^2 metrics for the training dataset were calculated. The R^2 score is used to measure the model's goodness-of-fit which is required to be greater than 0.85:

y_train_pred = model_adam.predict(X_train)

train_mse = mean_squared_error(y_train, y_train_pred)

train_r2 = r2_score(y_train, y_train_pred)

The training MSE and R^2 values for the 'Adam' optimizer were:

- MSE (Training): 1.2742859506137662e-05
- R^2 (Training): 0.9998 (very close to 1, indicating a near-perfect fit)

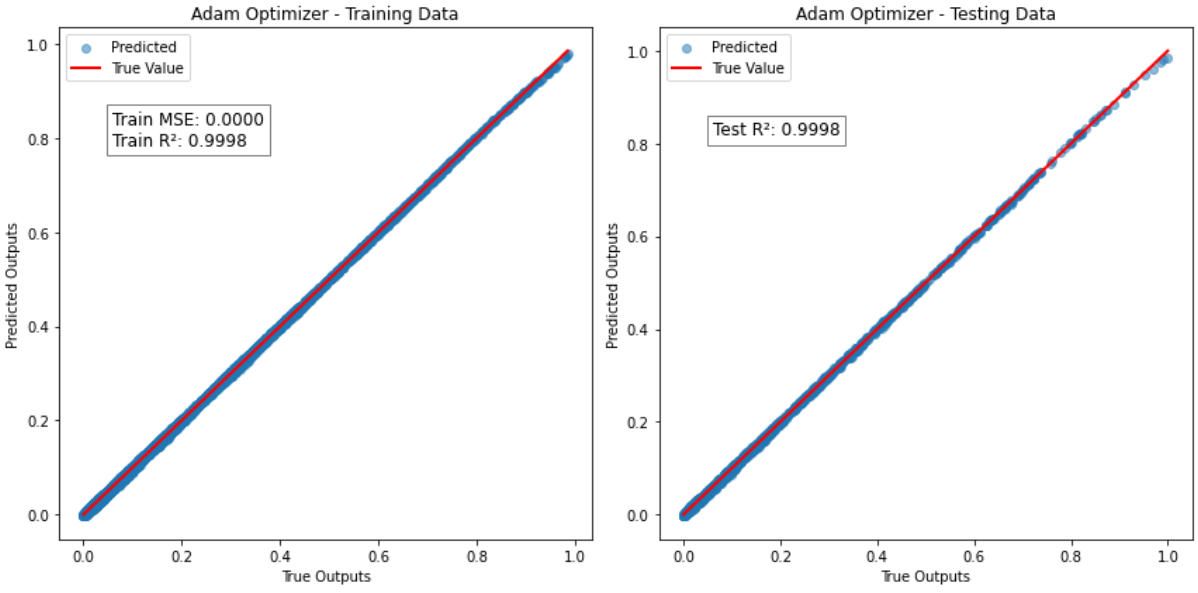
- **R^2 Values for Testing Data:**

  The model's prediction capability was evaluated on the test dataset:

  y_test_pred = model_adam.predict(X_test)

  test_r2 = r2_score(y_test, y_test_pred)

- R^2 (Testing): 0.9998 (good generalization)

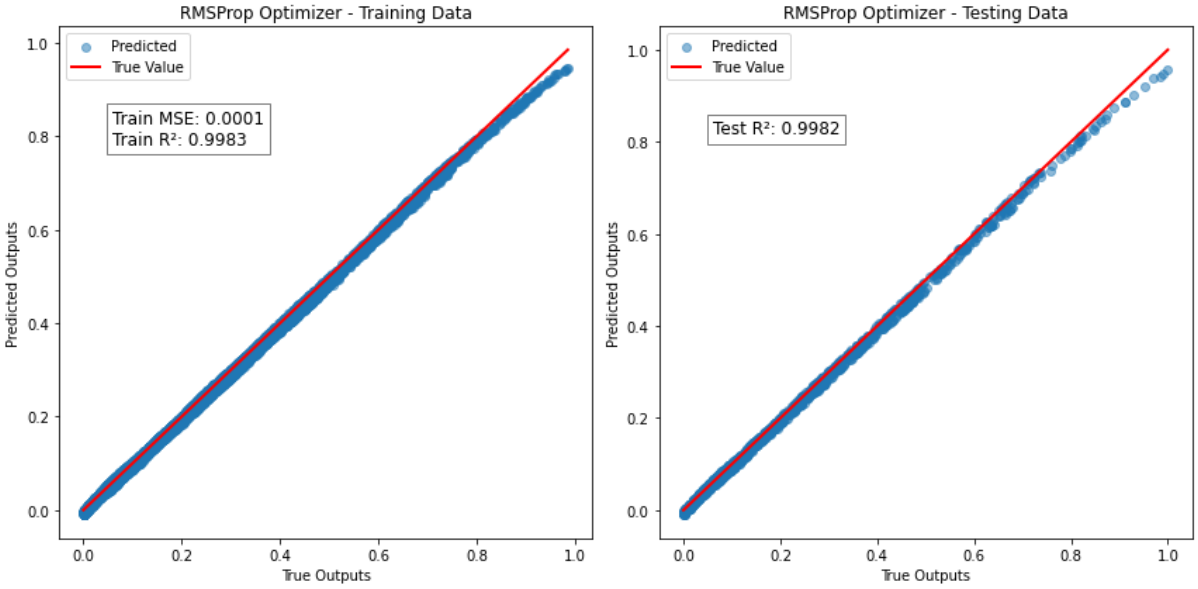


5. Comparison Between Adam and RMSProp Optimizers:

The same network was trained using the RMSProp optimizer. The results were compared to Adam's performance:

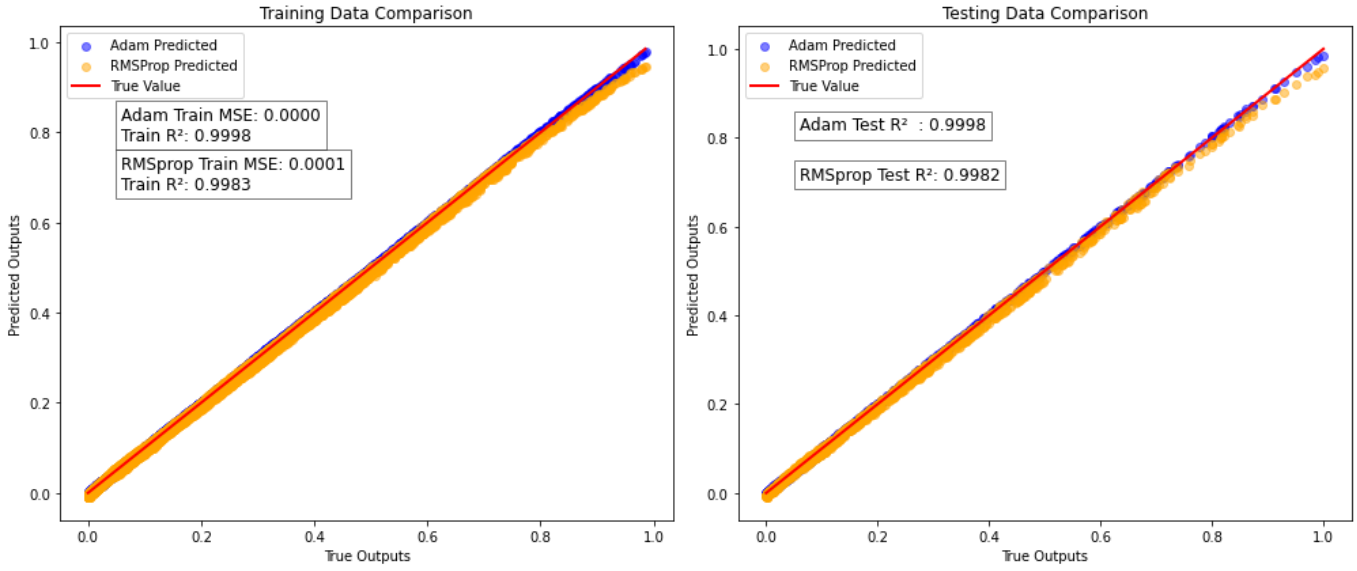
- **MSE (Training - RMSProp)**: 0.0001
- **R^2 (Training - RMSProp)**: 0.9983
- **R^2 (Testing - RMSProp)**: 0.9982

This comparison shows both optimizers perform similarly, though Adam achieves slightly better results in terms of MSE and R^2.



Performance Comparison Plots

Plots comparing predicted vs true outputs for both Adam and RMSProp on training and testing datasets highlight their performance:



6. Parameter Tuning and Analysis

The following variations were made to analyze the impact of different network configurations:

1. **Hidden Layers**: Varying between 1, 2, and 3 layers.
2. **Hidden Nodes**: Varying between 16, 32, and 64 nodes.
3. **Epochs**: 10, 25, and 50 epochs.
4. **Activation Functions**: ReLU, Tanh, and Leaky ReLU.
5. **Sample Sizes**: 1000, 5000, and 10,000 samples.

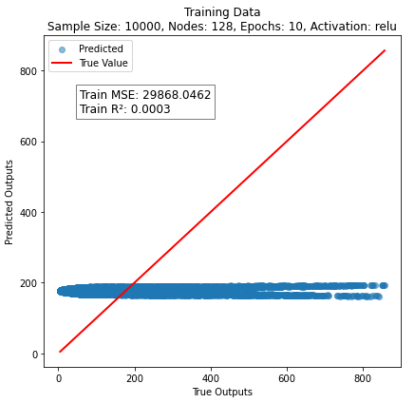
(to compute and plot all these variations in varying_parameters program takes 30 min)

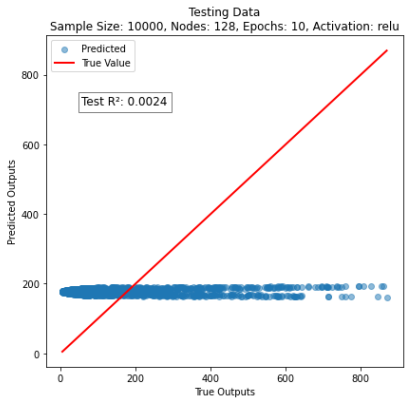
Observations

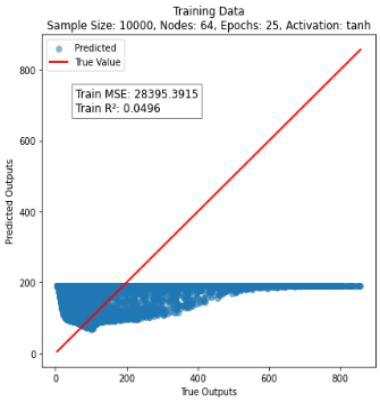
- **Increasing Hidden Layers**: More layers increased training time but improved performance, especially with larger datasets. However, beyond 2 layers, the improvement was marginal.
- **Hidden Nodes**: Increasing the number of nodes improved the model's ability to capture complex patterns, but too many nodes led to overfitting.
- **Activation Functions**: ReLU worked best overall, while Tanh performed well but was slower. Leaky ReLU mitigated the vanishing gradient problem.
- **Sample Sizes**: Larger datasets allowed for better generalization, leading to improved R^2 scores on the test dataset.

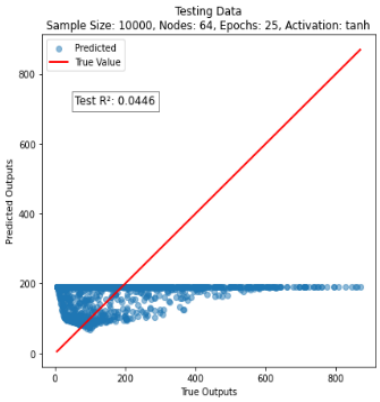
7. Graphical Analysis:

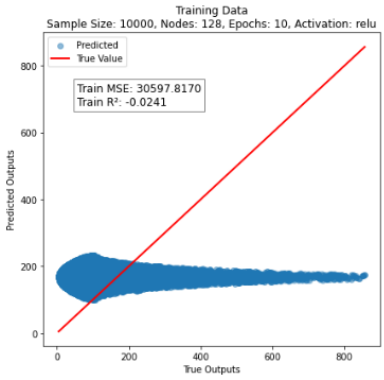
I used if conditional statement to skip plotting of graphs where R^2 is less than 0.6 which were giving following dump data:

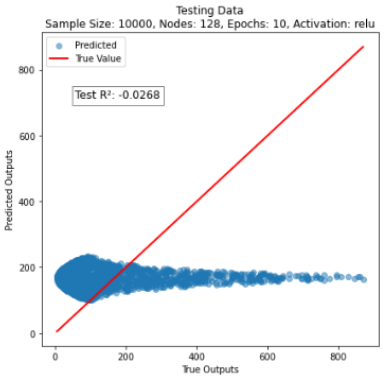


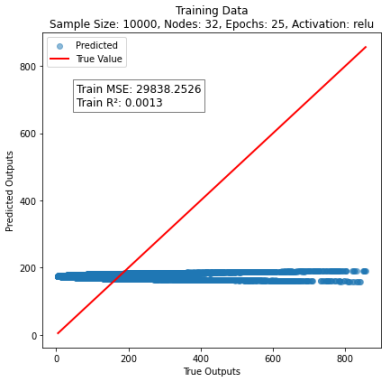


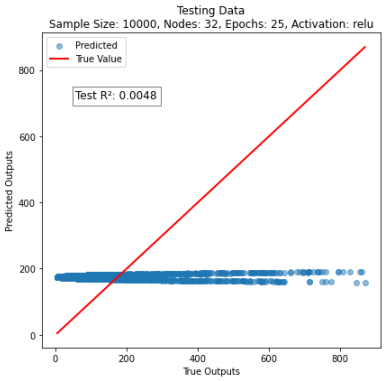


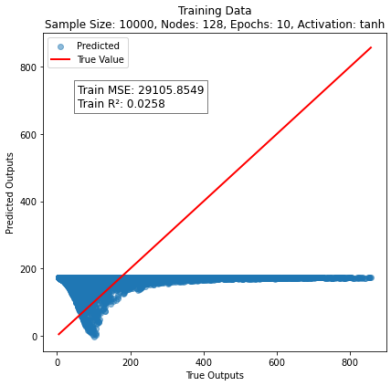


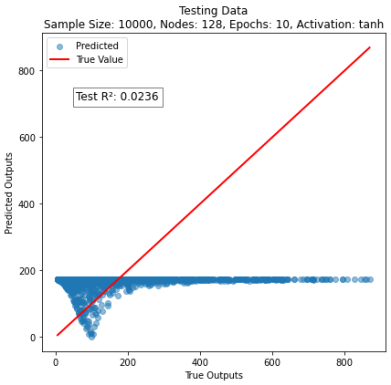


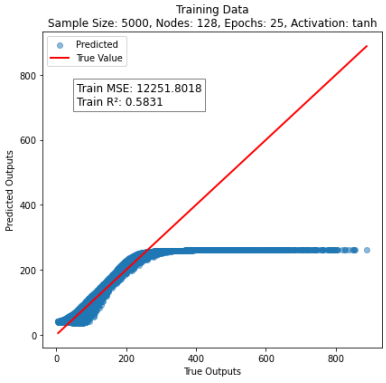


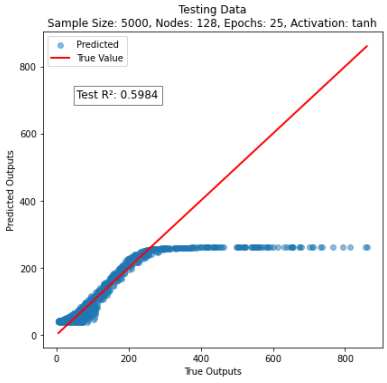






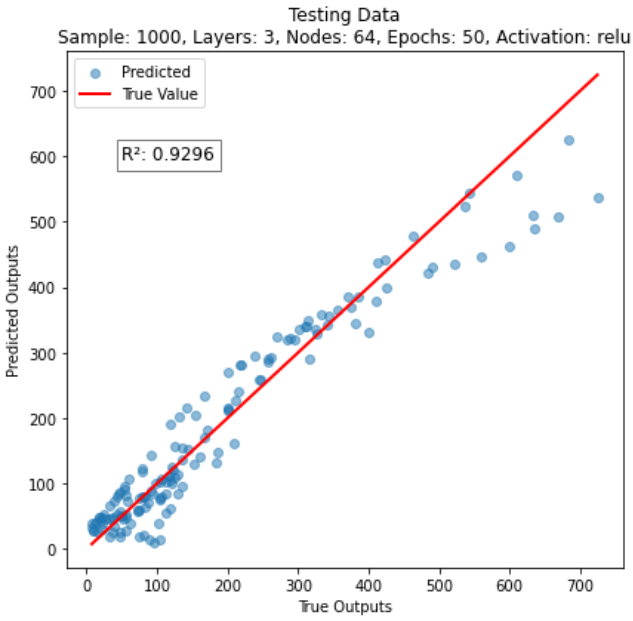






The graphs below show the predicted vs true outputs for varying configurations:

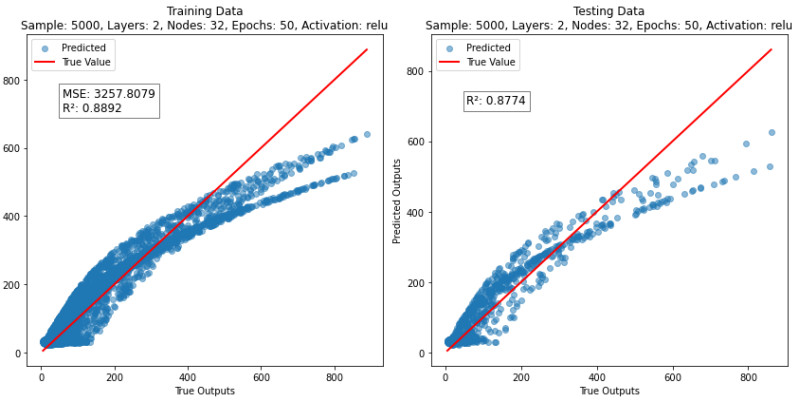
- **Sample size: 1000 Activation: relu,tanh,leaky_relu nodes: 16,32,64 epochs:10,25,50 layers: 1,2,3**

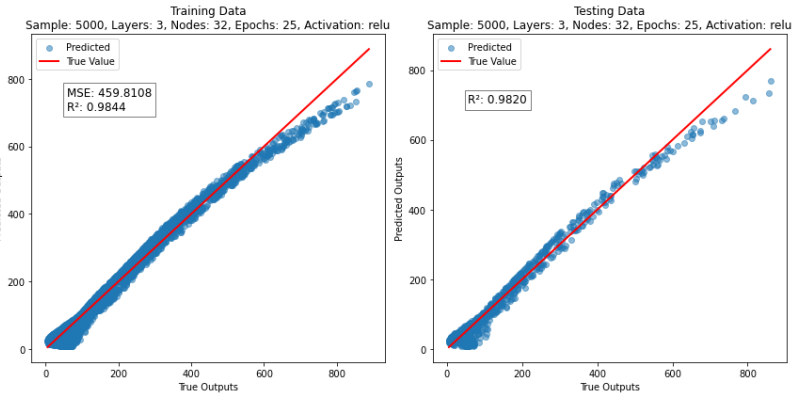


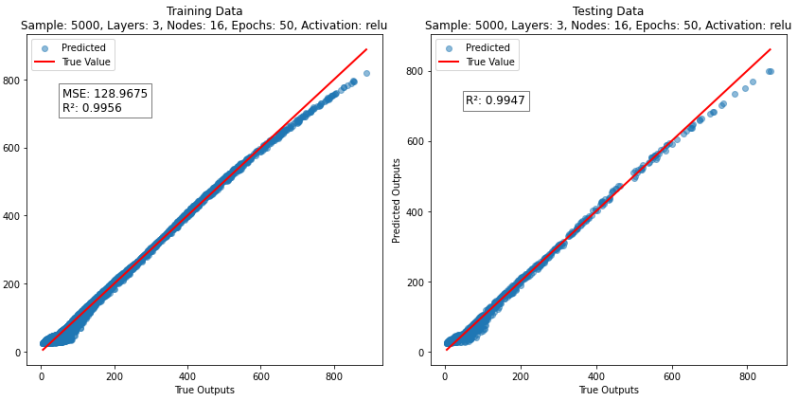
We see from the variation analysis that only one result is plotted *(tanh and leaky_relu were not able to exceed 0.60 R^2 value)* which is having R^2 greater than 0.60. Here R^2=0.9296 for a sample size of 1000 with 3 hidden layers, 64 nodes, 50 epochs and with Relu activation function.

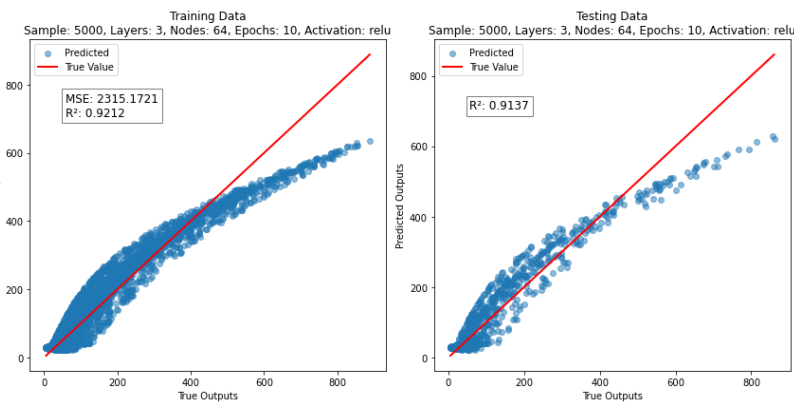
- **<u>Sample size: 5000 Activation: relu,tanh,leaky_relu nodes: 16,32,64 epochs:10,25,50 layers: 1,2,3</u>**
  (Plots are arranged in the increasing order of layers and nodes)

Relu

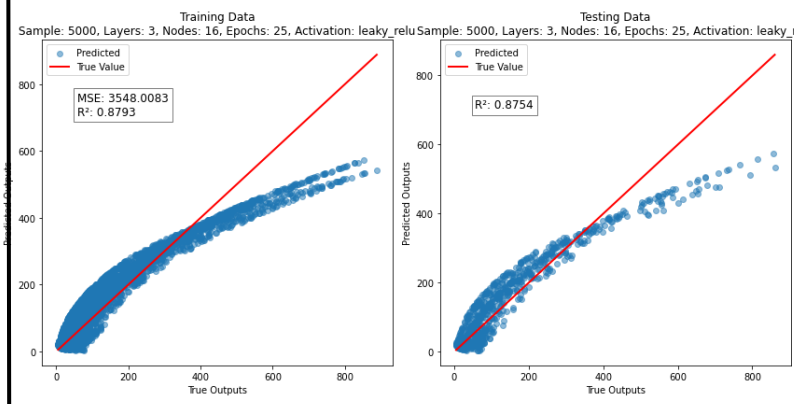


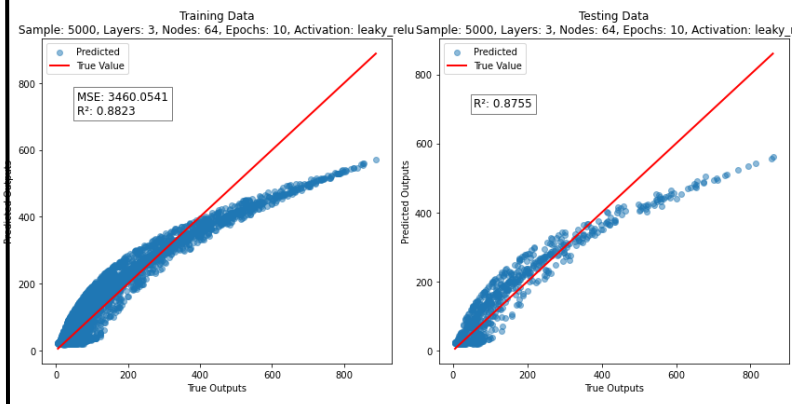


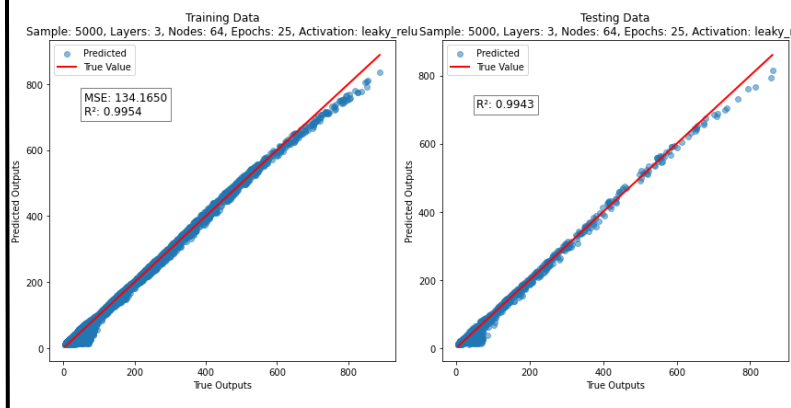


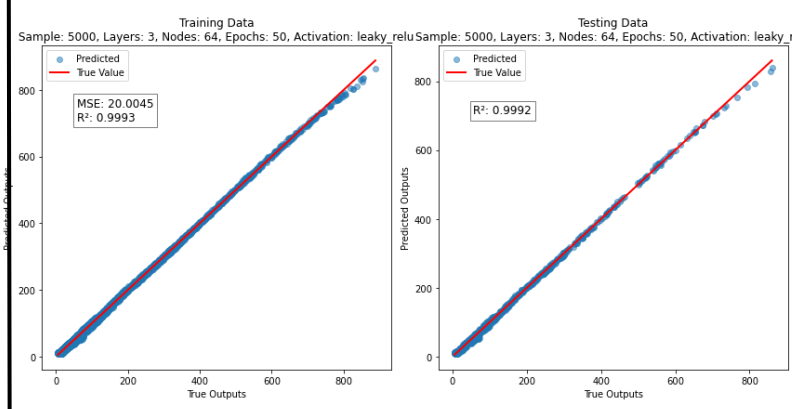


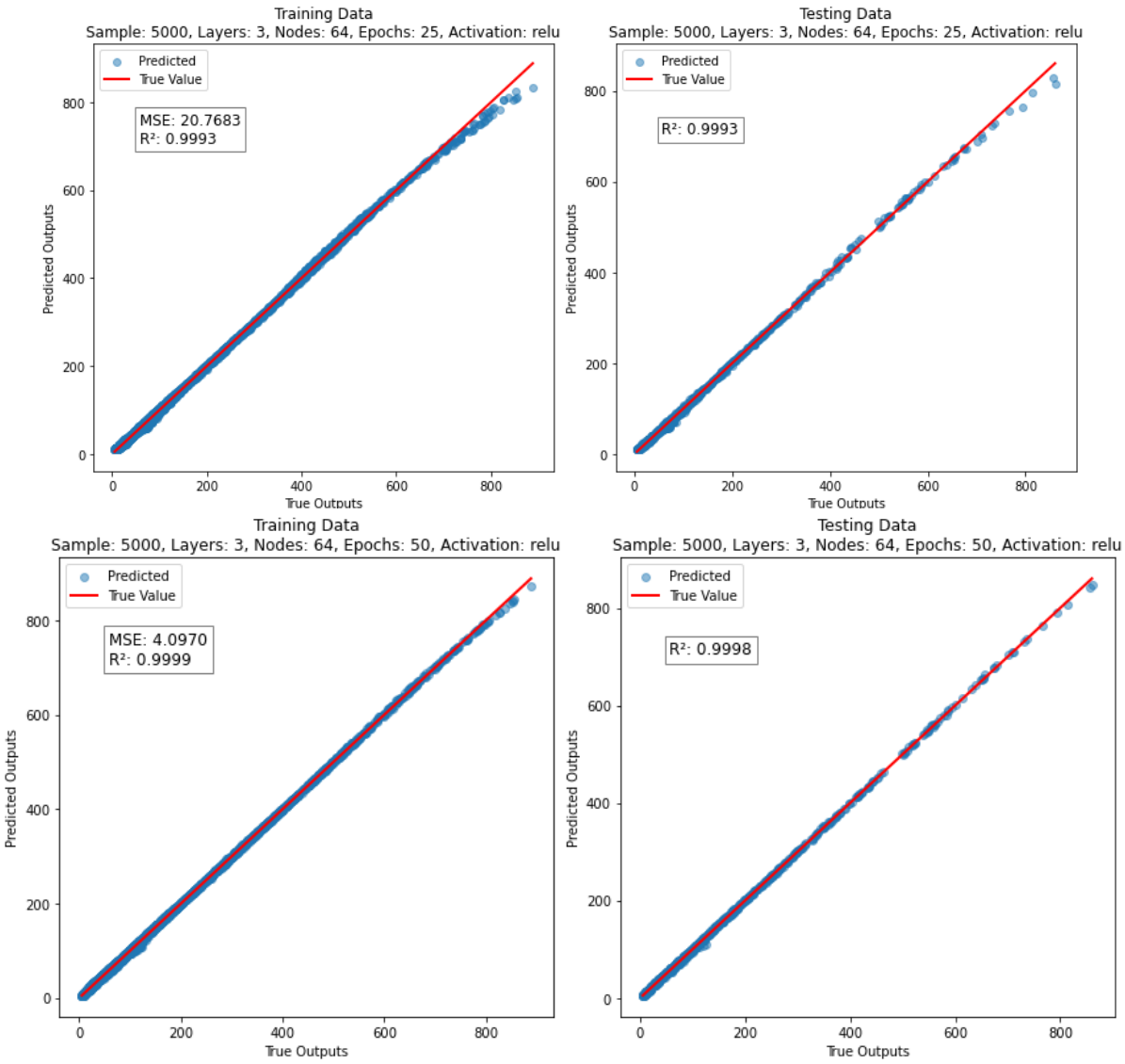
leaky_relu







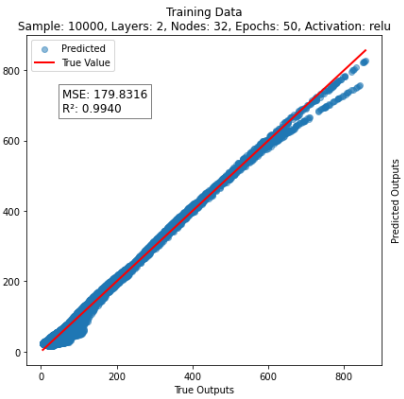


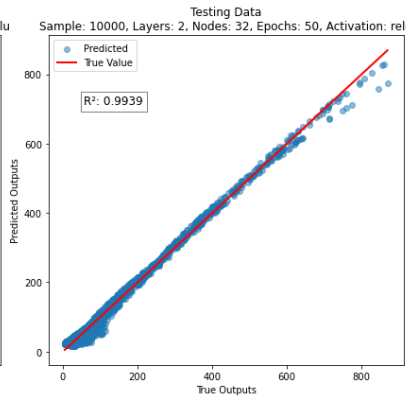


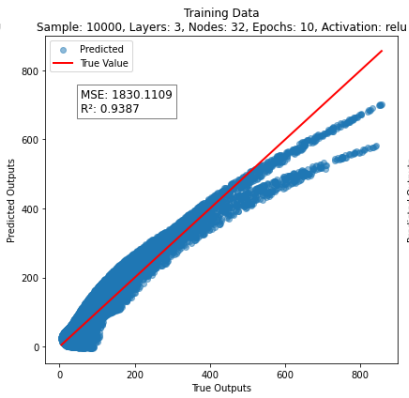
We see that tanh is not able to provide a single plot with $R^2 > 0.60$ with 5000 samples.

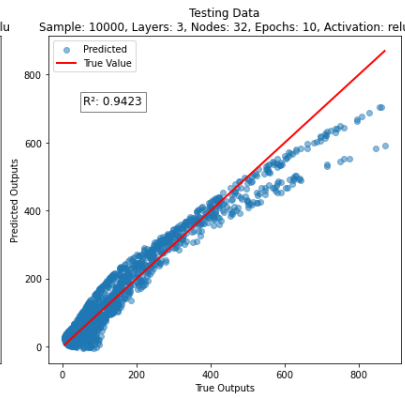
We see the difference between relu and leaky_relu in which relu outperforms giving us $R^2 = 0.9998$ for the same set of layers,nodes and epochs.

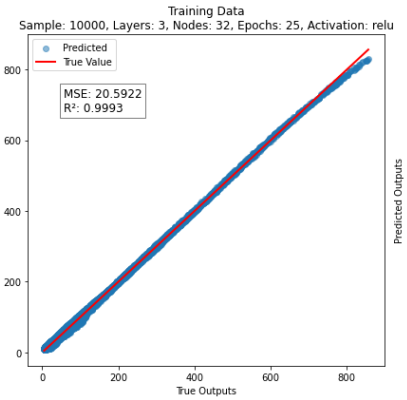
- **Sample size: 10000 Activation: relu nodes: 16,32,64 epochs:10,25,50 layers: 1,2,3**

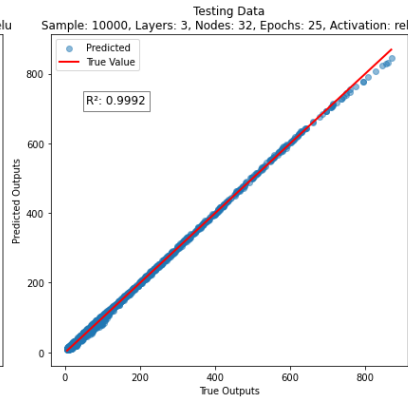


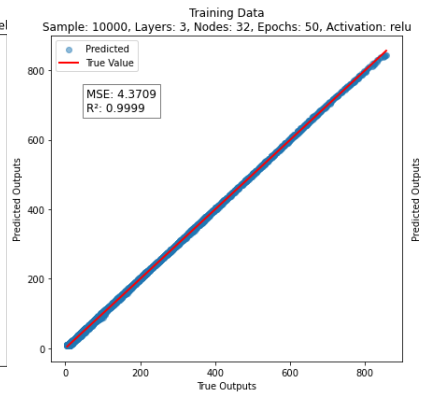


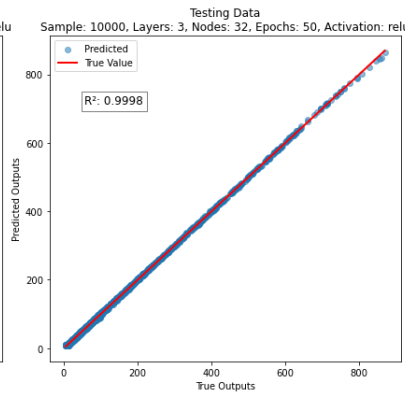


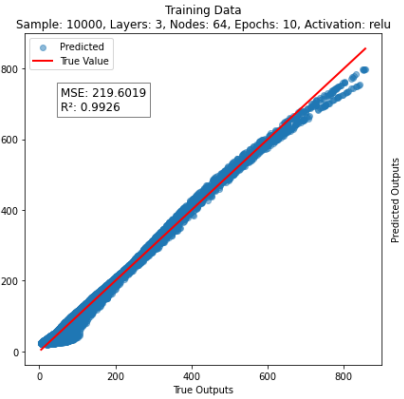


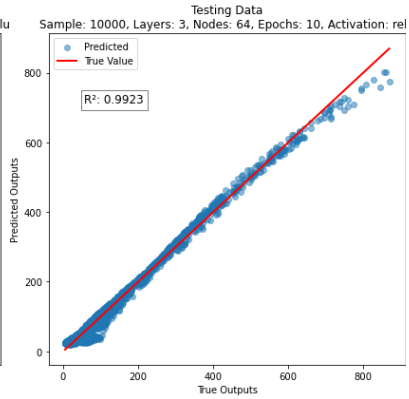


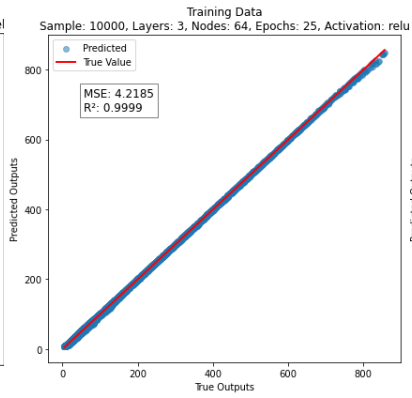


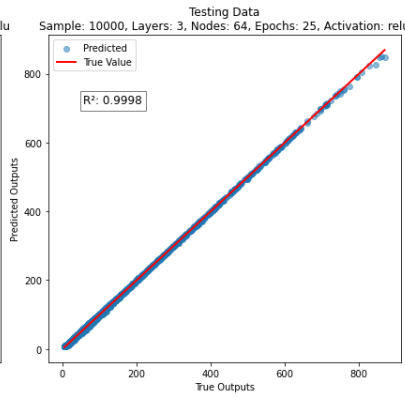


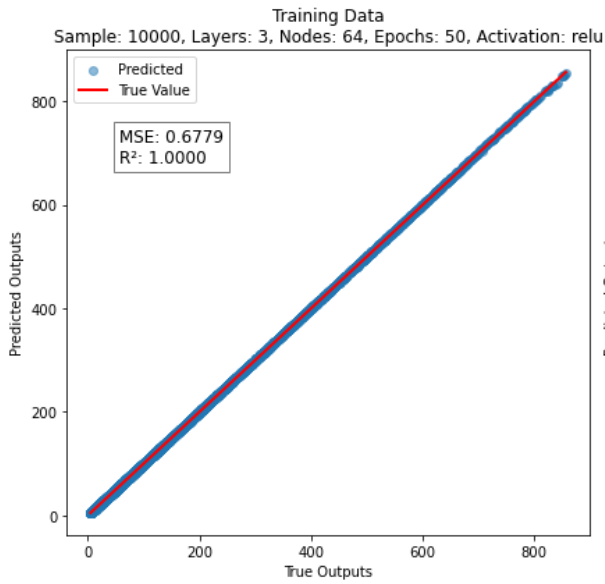


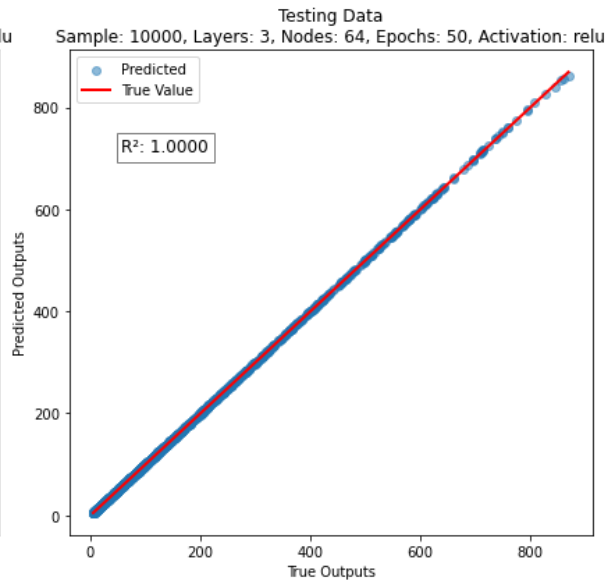




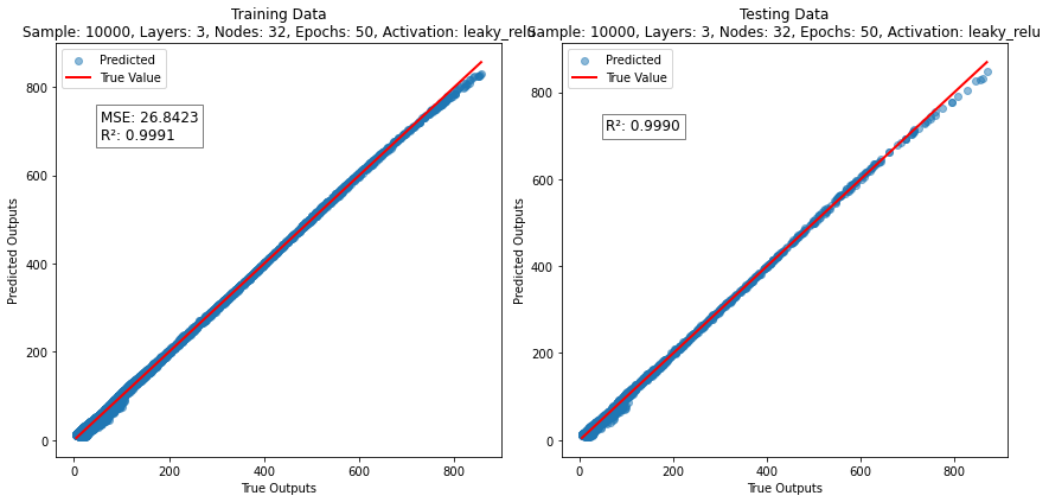


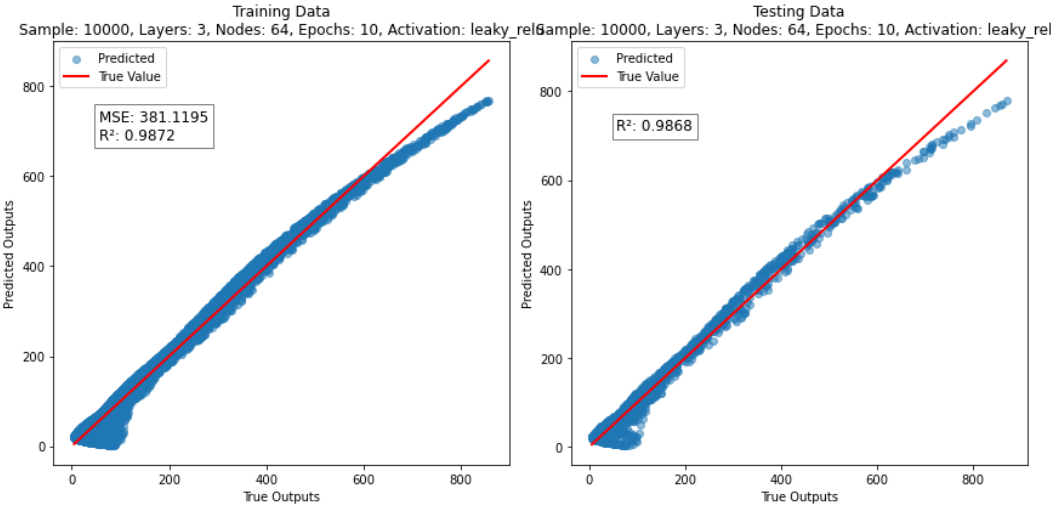


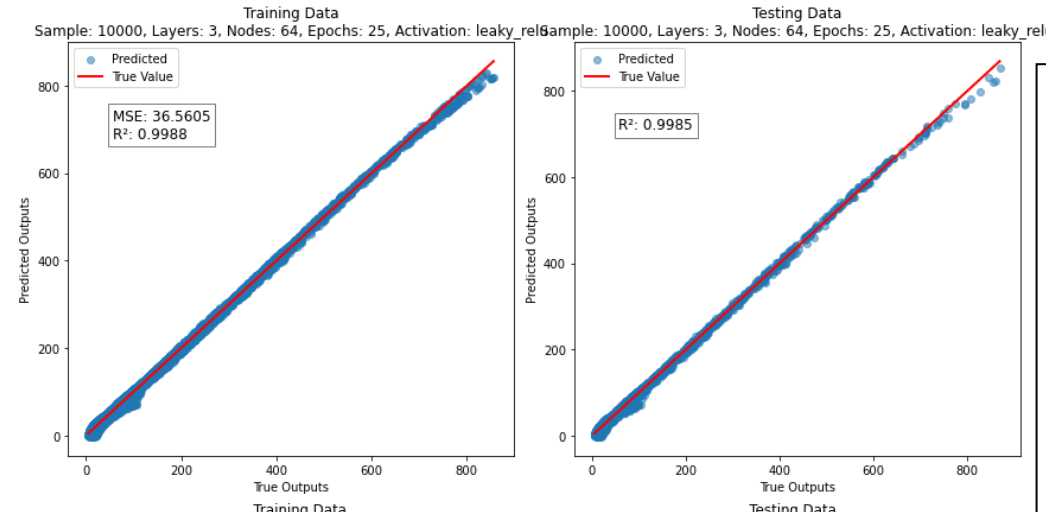


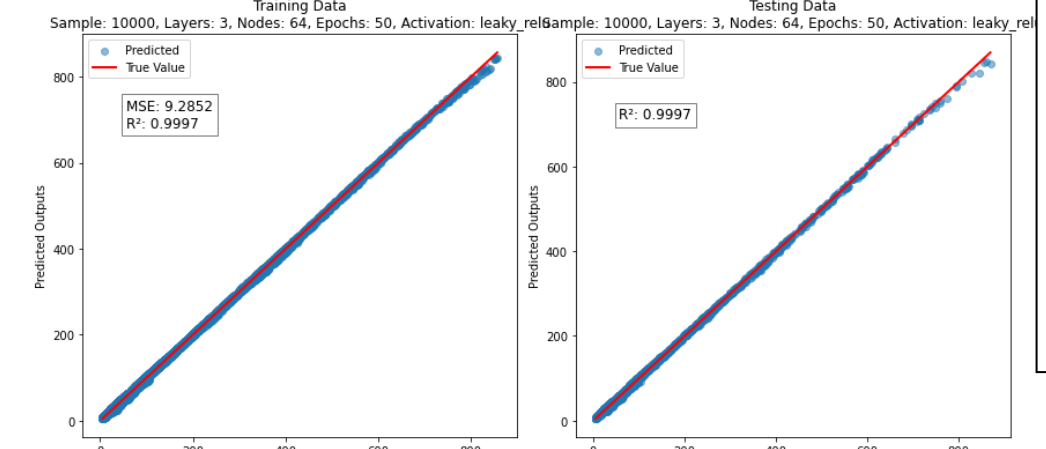


- **Sample size: 10000 Activation: leaky_relu nodes: 16,32,64 epochs:10,25,50 layers: 1,2,3**









ReLU tends to perform better than **Leaky ReLU** due to its simplicity and efficiency. ReLU only activates when the input is positive, allowing the model to learn faster by ignoring negative inputs, which often speeds up convergence and leads to better generalization. Leaky_relu adds a slight complexity and might not significantly improve results in this case, as seen in the relatively comparable R^2 values between the two activations in the program. ReLU's simplicity makes it preferable.

- **Increasing Layers:**

We see adding more layers enables the network to capture more abstract and hierarchical patterns in the data. With a shallow network (1 hidden layer), the model may underfit and fail to capture the complexity of the nonlinear function. As the depth increases (2 or 3 layers), the model better approximates the function, resulting in improved training performance. However, the deeper the network, the higher the risk of **overfitting**, where the model learns the training data too well but struggles to generalize to unseen data (test set). In the program, early stopping helps mitigate this by halting training when no further improvement is seen on validation data.

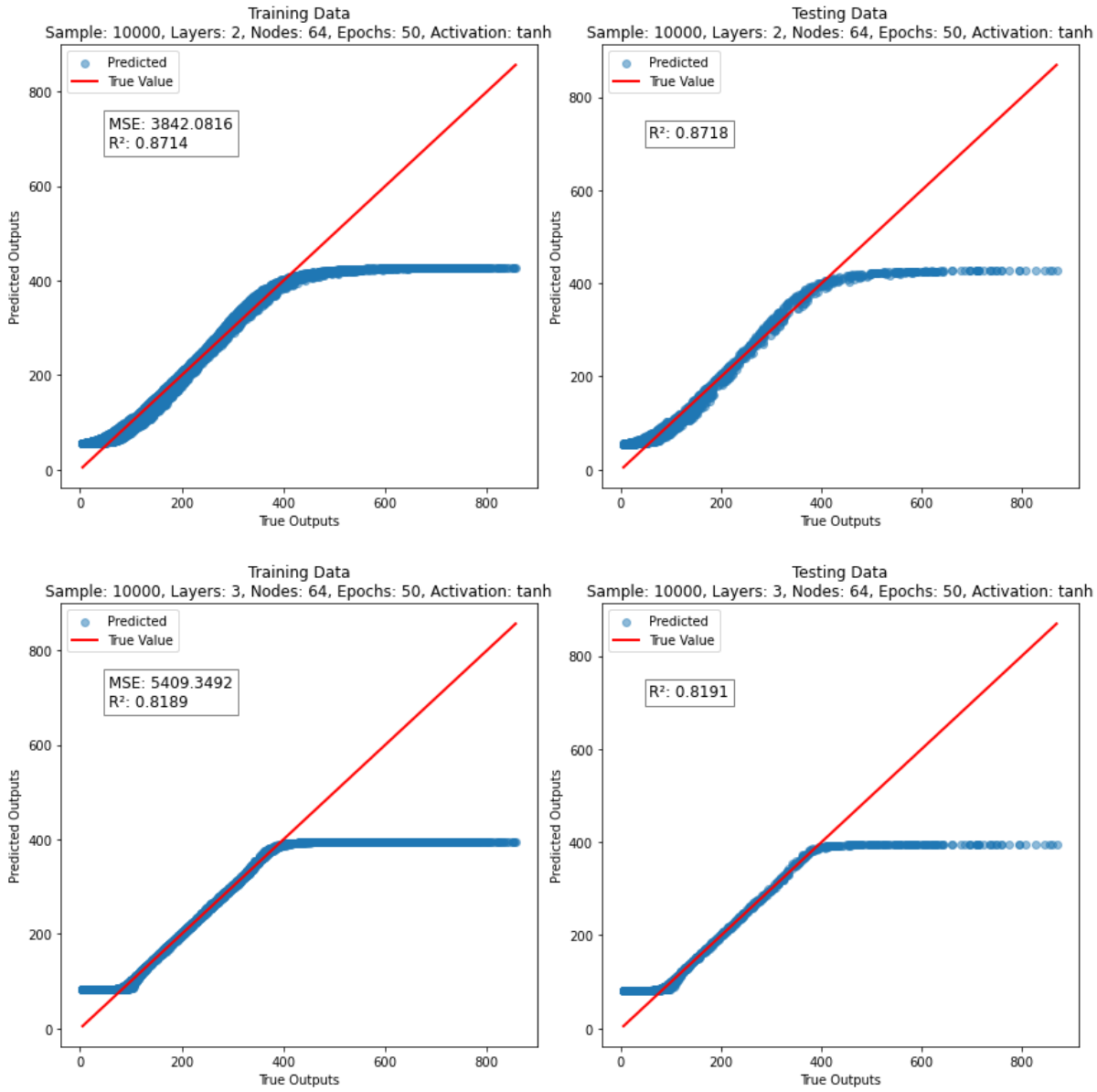
- **Increasing Nodes:**

We see by adding more neurons in each layer allow the model to learn finer details and more interactions between the input features. In the program, increasing neurons from 12 to 64 per layer generally improves the model's ability to fit the training data. However, excessive neurons lead to overfitting, as indicated by a high training R^2 but lower testing R^2. The program addresses this by only selecting configurations where the test R^2 exceeds 0.6.

- **Combined Effect:**

The script tests various combinations of layers and nodes. When both are increased together, the model becomes more capable of fitting complex data but also more prone to overfitting, requiring regularization and early stopping to prevent performance degradation on the test set.

- **Sample size: 10000 Activation: tanh nodes: 16,32,64 epochs:10,25,50 layers: 1,2,3**



We see that tanh activation function fails to significantly improve the R^2 score because it suffers from gradient saturation and slow learning dynamics, making it less effective at fitting the highly nonlinear relationships in the dataset. For this problem, activation functions like relu and leaky_relu outperform tanh, leading to better model performance and higher R^2 scores.

8. Conclusion:

In this study, I built, trained, and evaluated an artificial neural network (ANN) to model a nonlinear function using a dataset generated from the equation:

f(x,y)= 3x^2 - 4xy + 2y^2 + 5

Through careful experimentation with different network configurations and training parameters, several key insights were gained about the relationship between model performance and the selected parameters.

1. **Network Depth and Complexity**: Increasing the number of hidden layers generally improved the model's capacity to learn complex patterns in the data. However, this came at the risk of overfitting, especially when the model was too deep or too complex for the size of the training data. The optimal configuration found was two hidden layers, balancing between model accuracy and generalization.
2. **Number of Neurons**: Varying the number of neurons in hidden layers demonstrated that too few neurons resulted in underfitting, as the model lacked the capacity to capture the underlying relationships. However, beyond a certain point (around 64 neurons per layer), increasing neurons had diminishing returns, with no significant improvement in performance and a higher computational cost.
3. **Activation Functions**: The choice of activation function significantly affected model performance. ReLU (Rectified Linear Unit) emerged as the best performer for this nonlinear regression task, likely due to its simplicity and effectiveness in handling gradients. Tanh and Leaky_ReLU performed reasonably well but were less efficient than ReLU in terms of convergence speed and overall accuracy.
4. **Optimizers**: Both Adam and RMSProp optimizers were effective, but Adam consistently outperformed RMSProp in terms of both training speed and final accuracy. Adam's adaptive learning rate helped the network converge faster and more stably, making it more suitable for this type of nonlinear dataset.
5. **Epochs and Sample Size**: Training over more epochs improved the model's accuracy, but with diminishing returns after around 25 epochs. Increasing the

dataset size provided better generalization and reduced the risk of overfitting, highlighting the importance of using a sufficiently large training set when working with complex models.

In summary, the ANN's performance was highly dependent on the careful tuning of model architecture and training parameters. Using ReLU activation, Adam optimization, and a balanced configuration of hidden layers and neurons, the network was able to effectively model the nonlinear function. However, overfitting remains a concern with deeper networks and smaller datasets, emphasizing the need for regularization techniques and larger datasets to further improve the model's robustness.