Exercise 1

Group 5

Task Organization:

Xinyue Cheng - 1.2 Polynomial curve fitting

Junao Li - 1.2 Polynomial curve fitting

Yuchen Li - 1.1 Reading

Maiqi Zhou - 1.1 Reading

1.1 Reading

This paper introduces an open-source benchmark suite for ultra-low-power tiny machine learning systems. The key contribution is that it is the first readily reproducible industry-standard benchmark suite in this field.

The MLPerf Tiny framework is a flexible and modular system that accommodates the needs of both hardware and software users. It includes extensive reference implementations that function as standards for the open-source community. Furthermore, this paper discusses about four benchmarks in the suite: Visual Wake Words, Image Classification, Keyword Spotting, Anomaly Detection, targeting various usecases.

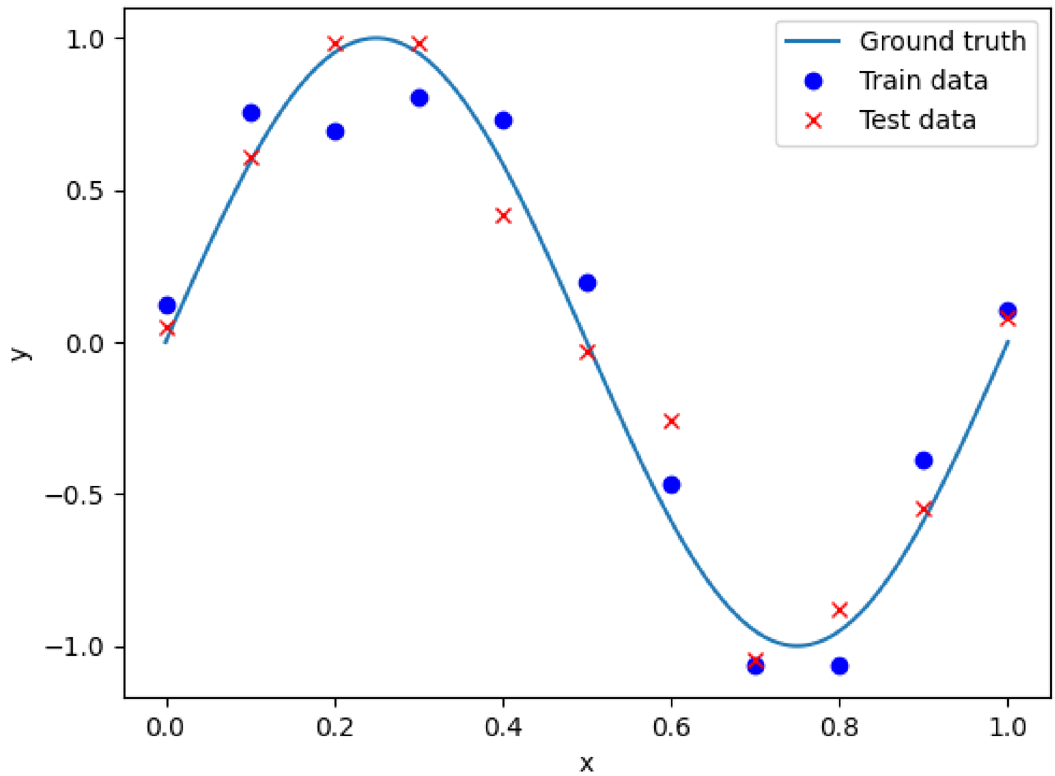
I fully accept this paper because a standard for performance measurement is essential to keep up with the rapid expansion in the IoT hardware and software industry. Additionally, by the time of this paper's publication, the benchmark presented has already been adopted by the TinyML research community as a standard test set, demonstrating the significant impact of this benchmark.

1.2 Polynomial curve fitting

1.2.1 *generate data of the shape h(x) = sin(2πx)*

Use `torch.sin` to generate the ground truth data $h(x)$=sin$(2\pi x)h(x)$=sin$(2\pi$**x). The dataset consists of training points (blue dots) and test points (cross marks), both with a Gaussian noise level of 0.15. The curve represents the ground truth we aim to model

```
def ground_truth_function(x):

    *# Generate data of the form sin(2 \* Pi \* x)*

    result = torch.sin(2 * np.pi * x)

    return result
```



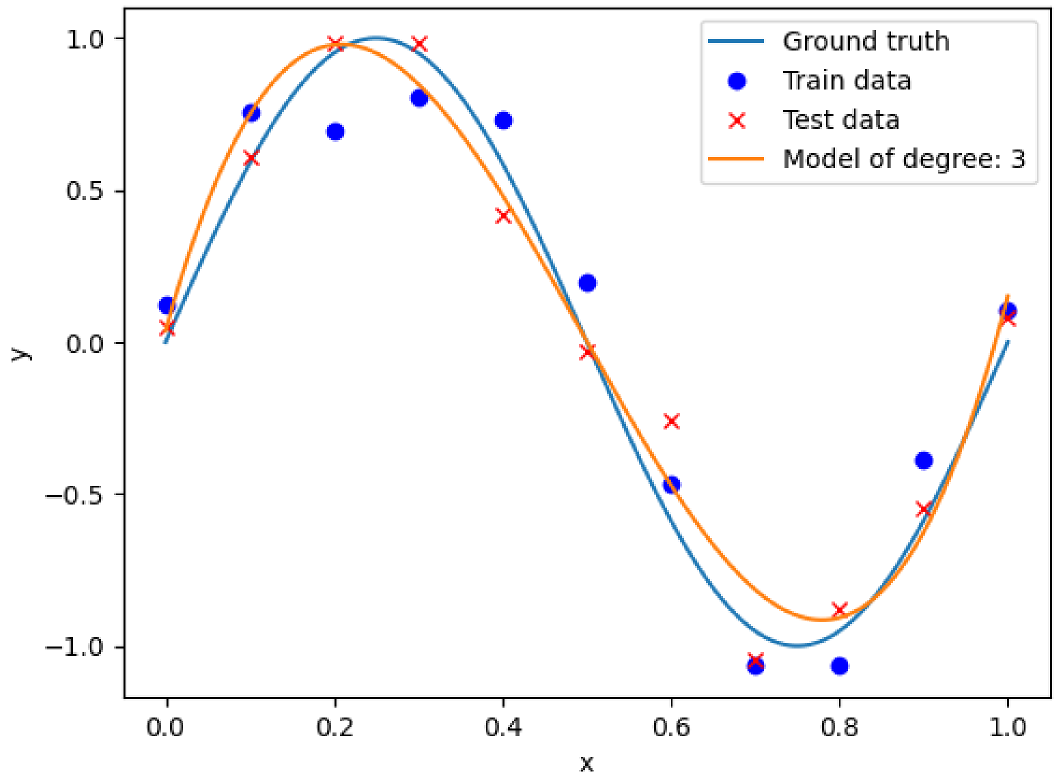
1.2.2 *Extend the error function to reflect the non-regularized error function from the lecture.*

Extend the error function to reflect the non-regularized form discussed in the lecture:

$E(w) = \frac{1}{2} \sum_{n=1}^{N} (h(x_n, w) - t_n)^2$

This error function calculates the mean squared error between the predicted outputs and the actual targets.

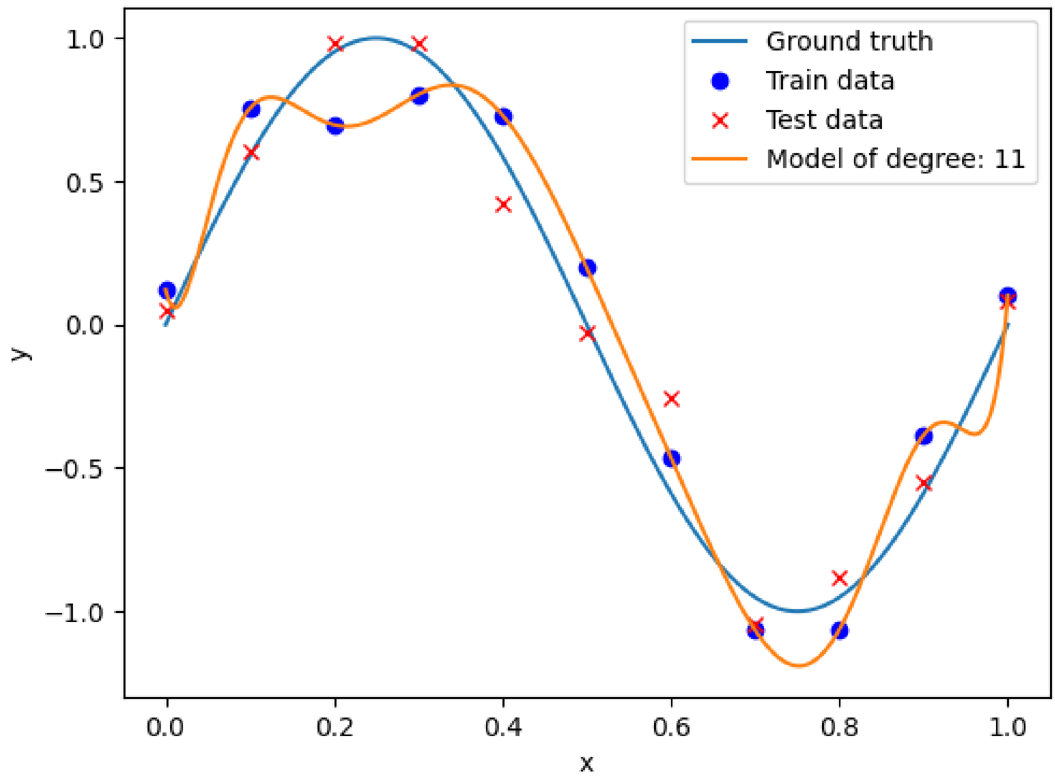
```
def error_function(*model*, *x_data*, *y_data*):

    y_pred = *model*(*x_data*)

    error = 0.5 * torch.mean((y_pred - *y_data*) ** 2)

    return error
```



1.2.3 *Create a new plot for an overfitted Polynomial of 11-th degree*

To demonstrate overfitting, fit an 11th-degree polynomial to the data. The resulting model fits the training data points perfectly but fails to generalize, as indicated by the poor fit over the test data. This model has minimized training error at the cost of losing generalization capability.

```
model_degree = 11
model = np.polynomial.Polynomial.fit(x_train, y_train, deg=model_degree)
train_err = error_function(model, x_train, y_train)
test_err = error_function(model, x_test, y_test)
```



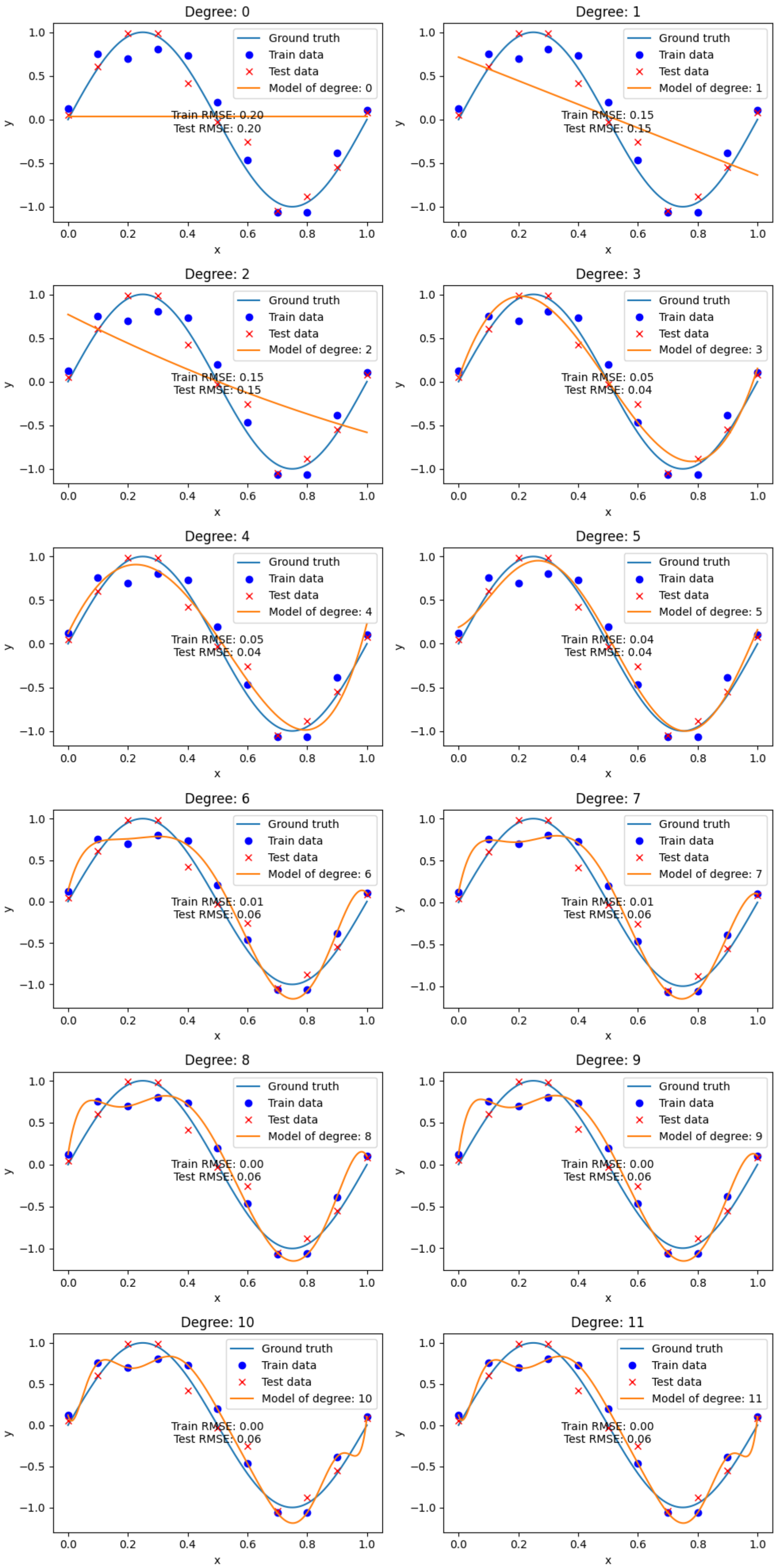
1.2.4 *Now vary the degree of the Polynomial from 0 to 11.*

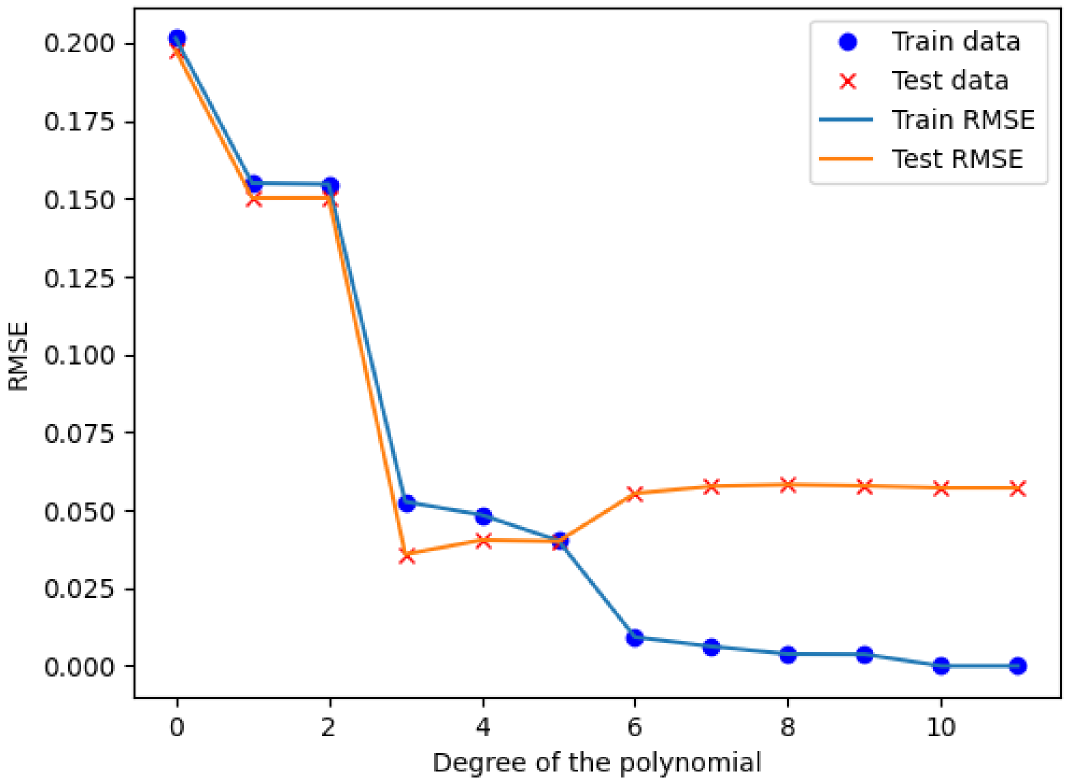
Investigate the impact of varying the polynomial degree from 0 to 11. Use the Root Mean Squared Error (RMSE) for assessment:

$$E_{RMS} = \sqrt{\frac{2E(w)}{N}}$$

```
# Define Root Mean Squared Error function
def rmse(model, x_data, y_data):
    y_pred = model(x_data)
    rmse = torch.sqrt(torch.mean((y_pred - y_data) ** 2))
    return rmse
```

This study reveals that higher-degree polynomials tend to overfit the training data, leading to larger errors on the test set compared to the training set.


Degree: 0
Ground truth
Train data
Test data
Model of degree: 0
Train RMSE: 0.20
Test RMSE: 0.20
Degree: 1
Model of degree: 1
Train RMSE: 0.15
Test RMSE: 0.15
Degree: 2
Model of degree: 2
Train RMSE: 0.15
Test RMSE: 0.15
Degree: 3
Model of degree: 3
Train RMSE: 0.05
Test RMSE: 0.04
Degree: 4
Model of degree: 4
Train RMSE: 0.05
Test RMSE: 0.04
Degree: 5
Model of degree: 5
Train RMSE: 0.04
Test RMSE: 0.04
Degree: 6
Model of degree: 6
Train RMSE: 0.01
Test RMSE: 0.06
Degree: 7
Model of degree: 7
Train RMSE: 0.01
Test RMSE: 0.06
Degree: 8
Model of degree: 8
Train RMSE: 0.00
Test RMSE: 0.06
Degree: 9
Model of degree: 9
Train RMSE: 0.00
Test RMSE: 0.06
Degree: 10
Model of degree: 10
Train RMSE: 0.00
Test RMSE: 0.06
Degree: 11
Model of degree: 11
Train RMSE: 0.00
Test RMSE: 0.06
x
y



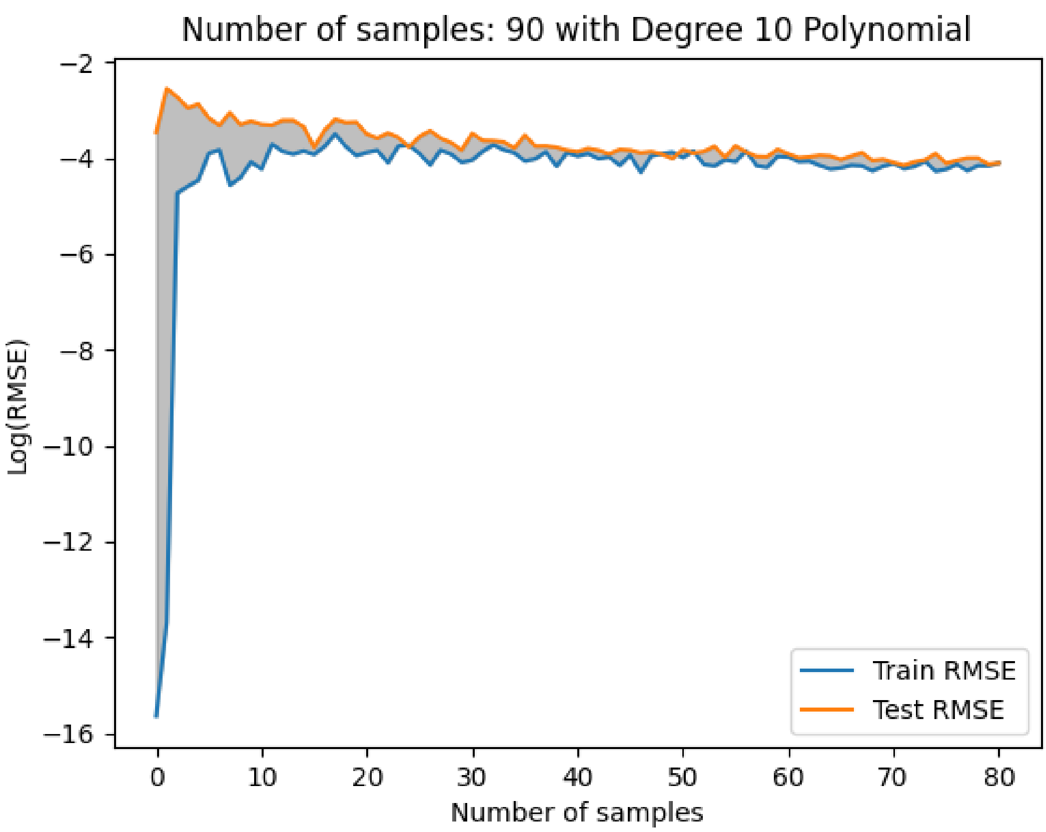
1.2.5 *Vary the size of the data, but keep the degree of the Polynomial constant 10*

Adjust the size of the dataset while keeping the polynomial degree constant at 10. The goal is to find the sample size where the RMSE between the training and test sets is small enough (less than 0.0001 difference).

```
# Starting with 10-th degree polynomial
"""
Change the sample size of the data until the RMSE difference between the train
and test data is less than 0.0001.
Parameters:
    x_train: torch.Tensor
    y_train: torch.Tensor
    x_test: torch.Tensor
    y_test: torch.Tensor
    model_degree: int
    noise_amplitude: float
    n_samples: int
    step_size: int

Returns:
    steps: int
    train_err: float
    test_err: float
    n_samples: int
```

```
"""
def vary_data_size(x_train, y_train, x_test, y_test, model_degree,
noise_amplitude, n_samples, step_size):
    steps = 0
    train_err = []
    test_err = []
    while True:
        steps += 1
        x_train = torch.linspace(0, 1, n_samples)
        y_train = ground_truth_function(x_train) + torch.normal(0.,
noise_amplitude, size=(n_samples,))
        x_test = torch.linspace(0, 1, n_samples)
        y_test = ground_truth_function(x_test) + torch.normal(0.,
noise_amplitude, size=(n_samples,))
        model = np.polynomial.Polynomial.fit(x_train, y_train, deg=model_degree)
        train_err.append(rmse(model, x_train, y_train))
        test_err.append(rmse(model, x_test, y_test))
        if abs(train_err[-1] - test_err[-1]) < 0.0001:
            break
        n_samples += step_size

    return steps, train_err, test_err, n_samples
```

It turns out that approximately 90 samples effectively mitigate overfitting for this task.



Conclusion

To mitigate overfitting, one can increase the sample size, choose an appropriate polynomial degree, or utilize a well-defined error function with regularization. These strategies can significantly enhance model generalization.