Homework 4: Bias/Variance, Overfitting, & Generalization

In [1]:
```
# MAKE SURE TO RUN THIS CELL
%matplotlib notebook
```

Note: For this homework, we will be doing a lot of polynomial regression. Please do not use existing polyfit tools such as those provided in scikit-learn or numpy unless otherwise noted. You need practice implementing it yourself by hand.

In [2]:
```
import base64
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

Introduction

Here I've provided 2 convenient plotting functions for you to use. `function_plot` will be used to plot the polynomials you're given/trying to estimate, while `scatter_plot` will be used to plot the generated data.

In [3]:
```
def funtion_plot(f, featurize_fn=None):
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    x1 = np.arange(-1, 1, 0.1)
    x2 = np.arange(-1, 1, 0.1)
    x1, x2 = np.meshgrid(x1, x2)
    grid_points = np.stack((x1.flatten(), x2.flatten()), axis=1)

    if featurize_fn is not None:
        grid_points = featurize_fn(grid_points)
    y = f(grid_points)
    y = y.reshape(x1.shape)

    surf = ax.plot_surface(x1, x2, y, cmap=cm.coolwarm,
                           linewidth=0, antialiased=False)

    fig.colorbar(surf, shrink=0.5, aspect=5)

    plt.show()
```

In [4]:
```
def scatter_plot(x, y, featurize_fn=None):
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    surf = ax.scatter(x[:,0], x[:,1], y, cmap=cm.coolwarm, c=y)

    fig.colorbar(surf, shrink=0.5, aspect=5)

    plt.show()
```

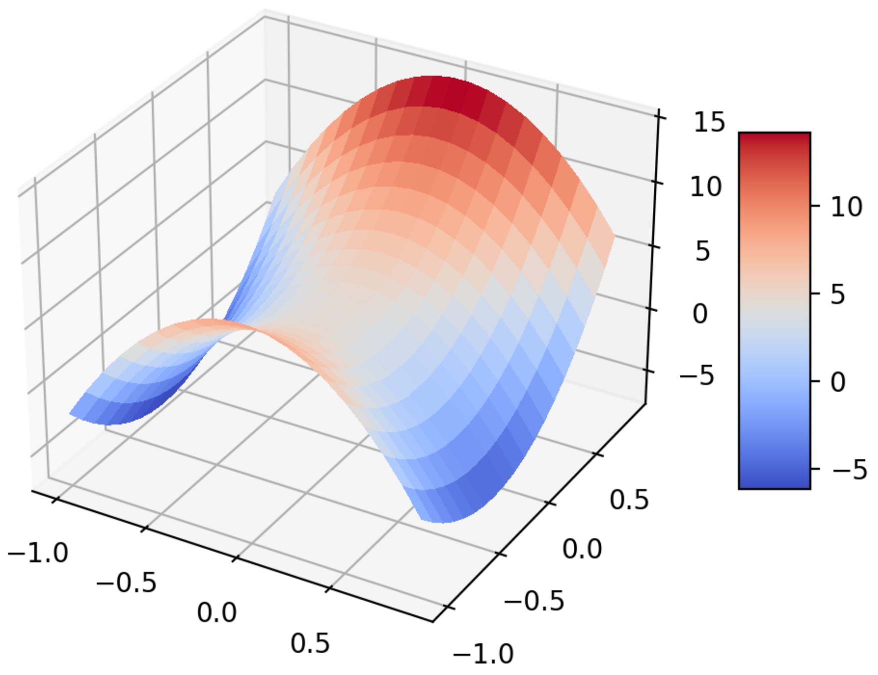
Here, I am defining a polynomial defined over 2 variables in `f_2d_polynomial` . This is the "true" function we will be trying to estimate. `f_2d_polynomial` takes in an Nx2 numpy array and returns a vector of length N, representing the value of the polynomial at each point. Note that I have slightly obfuscated the code so you can't just read off the polynomial coefficients. Obviously it is easy to reverse engineer, but these homeworks are provided in good faith for your learning experience.

I've also defined a `generate_data` function which samples random points in the interval $[-1, 1]^2$, evaluates them using the true polynomial, and then adds gaussian noise to the output to represent measurement error.

In [5]:
```
def f_2d_polynomial(x):
    x0, x1 = x[:,0], x[:,1]
    secret = b'NCp4MSArIHgwKngxIC0gMTIqeDAqKjIgKyA4KngxKioyICsgNQ=='
    y = eval(base64.b64decode(secret))
    return y


def generate_data(noise_strength, num_points):
    x = np.random.rand(num_points, 2) * 2 - 1
    y = f_2d_polynomial(x)
    noise = np.random.randn(*y.shape)
    return x, y + noise * noise_strength
```

First, let's take a look at what our polynomial truly looks like.

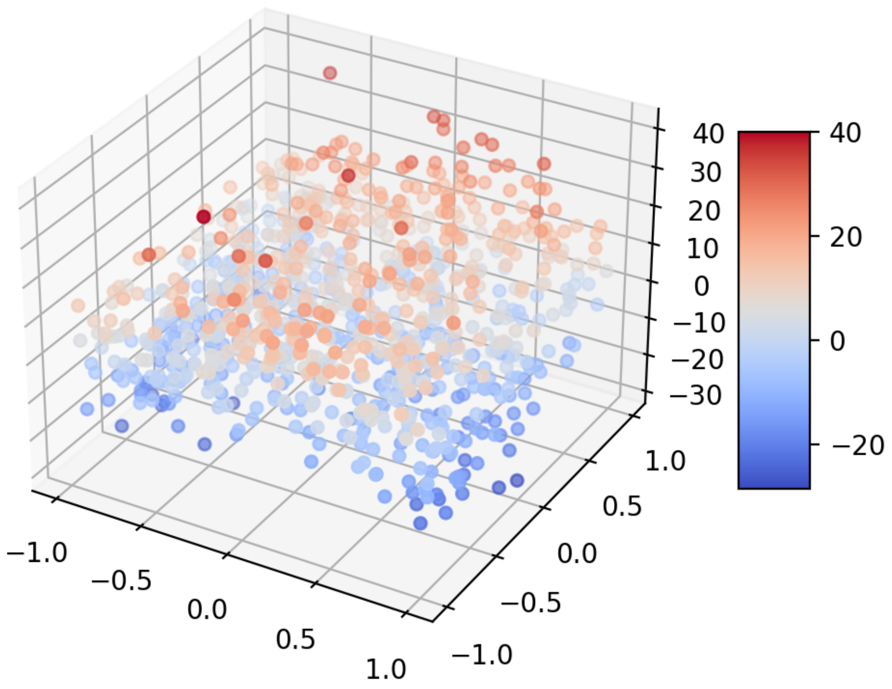
In [6]:
```
funtion_plot(f_2d_polynomial)
```



Very nice! Feel free to move and inspect the plot with your mouse to get a better understanding of the polynomial. Now, let's sample some noisy training data and plot it to see what it looks like.

```
In [7]:  X_train, y_train = generate_data(noise_strength=10, num_points=1000)
```

```
In [8]:  scatter_plot(X_train, y_train)
```



Very nice! The data looks very noisy, but we can still make out the overall trend of the underlying function. Let's now try to recover the original function using this noisy data.

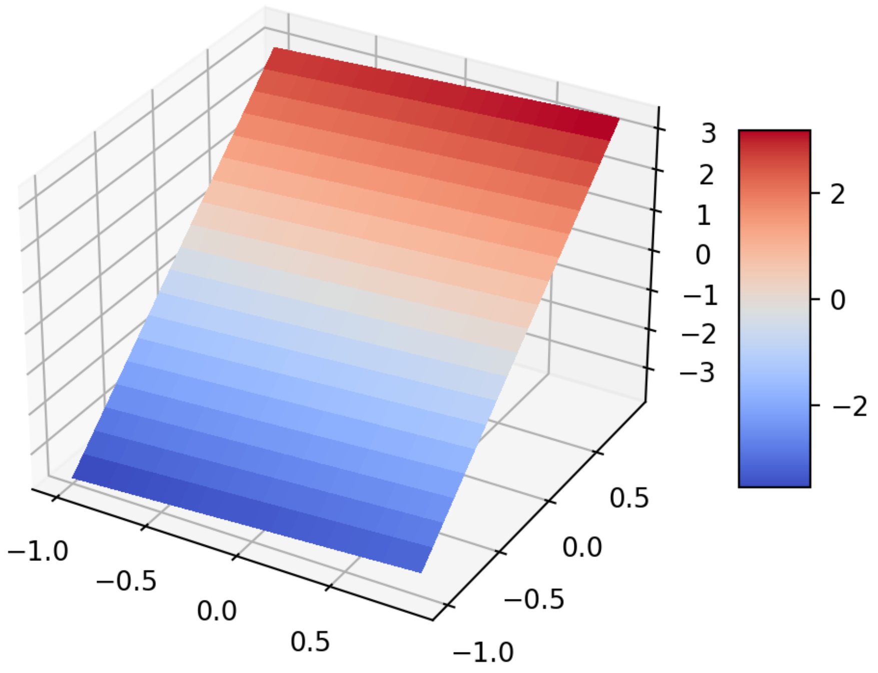
Part A: Polynomial Regression

When given new data, your instinct should always be to try linear regression first. In this homework, we will start with linear regression and then build it into polynomial regression (estimating with a polynomial of degree n) by adding polynomial features to our data before performaing linear regression on it.

First, perform ordinary linear regression on your 2-dimensional data points x. You are not allowed to call any external libraries or functions. Use only very basic numpy functions for your implementation. You may use `np.linalg.inv` if you wish.

In this next codeblock, use `X_train` and `y_train` to create a function `estimated_f` that takes in new data points in a Nx2 array and returns a vector of N labels. Your resulting plot should look like a 2-dimensional line (plane).

In [9]:
```
Xt = np.transpose(X_train)
XtX = np.dot(Xt, X_train)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)
estimated_f = lambda y: np.dot(y, solution)

funtion_plot(estimated_f)
```



As you can see, our plot doesn't look like the true function at all. In this case, are we overfitting or underfitting? (Discuss how expressive our model is and how that factors into your answer.)

Our model is underfitting here, since it is unable to capture most of the patterns in the data. Since our model can only model 2d planes, it is not expressive enough to be able to express a higher degree function such as the one we are trying to find.

Now instead of a line, let's estimate our function with a degree-2 polynomial. Note that this will make our model more expressive. How? Before, with a line, our model looked like this:

$$f_{hat}(x_1, x_2) = a_0 + a_1 * x_1 + a_2 * x2$$

If we want to use a 2d polynomial, our model will now look like this:

$$f_{hat}(x_1, x_2) = a_0 + a_1 * x_1 + a_2 * x2 + a_3 * x_1 * x_2 + a_4 * x_1^2 + a_5 * x_2^2$$

Note that we've now included all terms and cross-terms up to degree 2. But this function is no longer linear in x_1 and x_2! This poses a challenge for us, since all we know how to do is linear regression. However, supposed someone gave us the higher order terms. What would our function look like then?:

$$f_{hat}(x_1, x_2, x_3, x_4, x_5) = a_0 + a_1 * x_1 + a_2 * x2 + a_3 * x_3 + a_4 * x_4 + a_5 * x_5$$

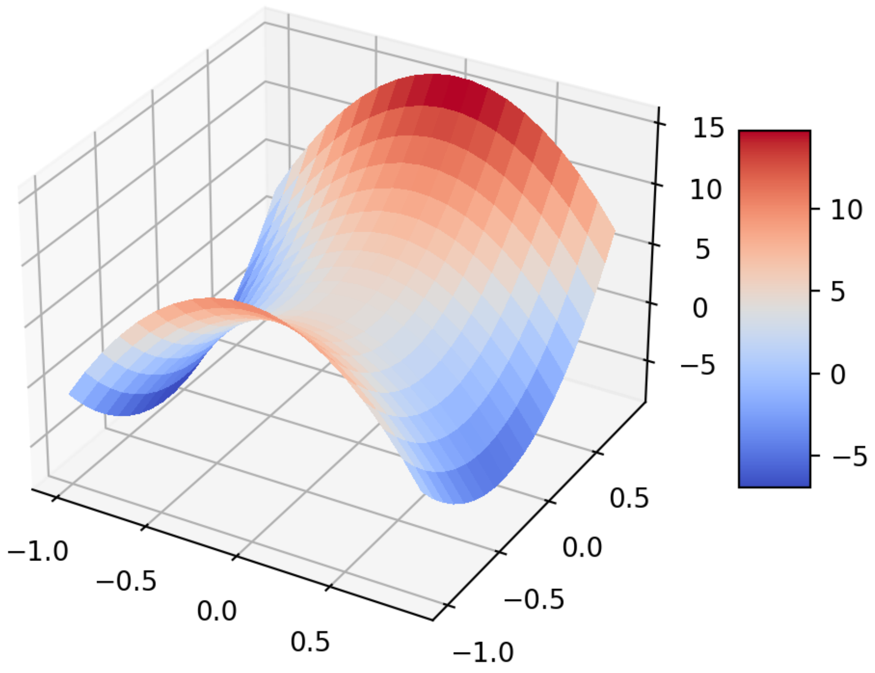
where $x_3 = x_1 * x_2$, $x_4 = x_1^2$, $x_5 = x_2^2$.

Now this function is linear in its inputs! And all we've done is use the existing features to create more features and add them to our dataset. This is exactly the idea behind polynomial regression - append higher-order polynomial terms as features to your dataset and then perform standard linear regression!

Now, perform 2d polynomial regression. You are not allowed to call any external libraries or functions. Use only very basic numpy functions for your implementation. You need to featurize your data appropriately by hand. You may use `np.linalg.inv` if you wish.

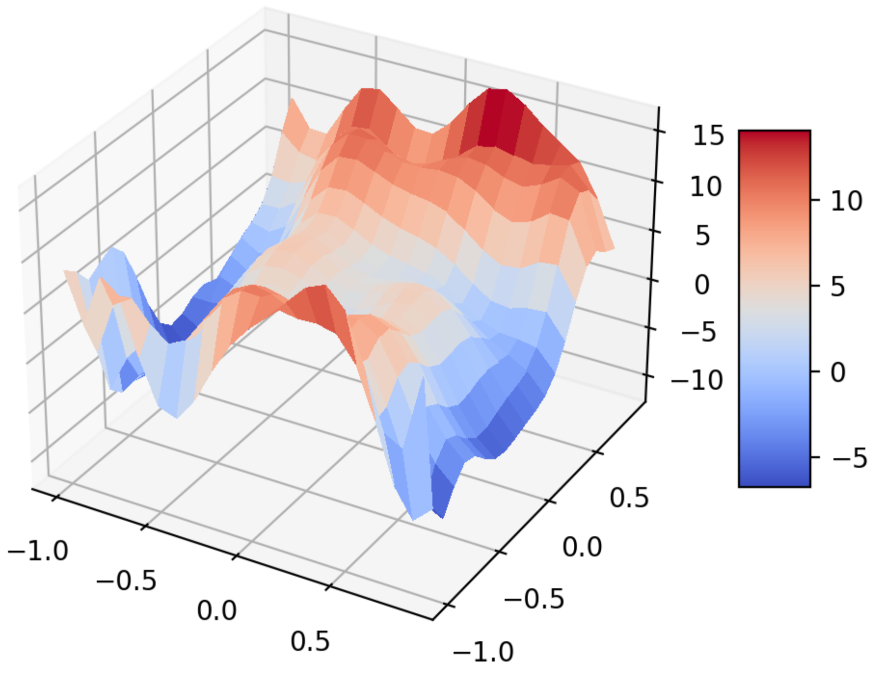
In this next codeblock, use `X_train` and `y_train` to create 2 functions: `featurize_fn` that takes in data points in a Nx2 array and returns a Nxd array containing your featurized data (don't forget to add your bias term), and `estimated_f` which takes in new data in a Nxd array and returns a vector of N labels. Your resulting plot should look very close to the ground truth function.

In [10]:
```
featurize_fn = lambda x: [[1, r[0], r[1], r[0]*r[1], r[0]**2, r[1]**2] for r in
X = featurize_fn(X_train)
Xt = np.transpose(X)
XtX = np.dot(Xt, X)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)
estimated_f = lambda x: np.dot(x, solution)

funtion_plot(estimated_f, featurize_fn=featurize_fn)
```



Very nice! Now let's see what the plot looks like when we include higher order terms up to degree ten. From now on, you may use `PolynomialFeatures` (imported below) to help aid constructing polynomial features.

In [11]:
```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(10, include_bias = True)
featurize_fn = lambda x: poly.fit_transform(x)
X = featurize_fn(X_train)
Xt = np.transpose(X)
XtX = np.dot(Xt, X)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)
estimated_f = lambda x: np.dot(x, solution)

funtion_plot(estimated_f, featurize_fn=poly.fit_transform)
```



We can see clear signs of overfitting from this plot. Please describe whether you expect this function (degree 10) or the previous one (degree 2) to perform better on new test data sampled from the same distribution.

I expect the degree 2 function to perform better because it is a more general model for data of the type we're being given than the degree 10 model, which overfits on the training dataset

Part B: Bias/Variance of Polynomial Degree

To confirm you answer, let's generate some test data and compare errors of models with varying degrees from degree 0 to degree 15.

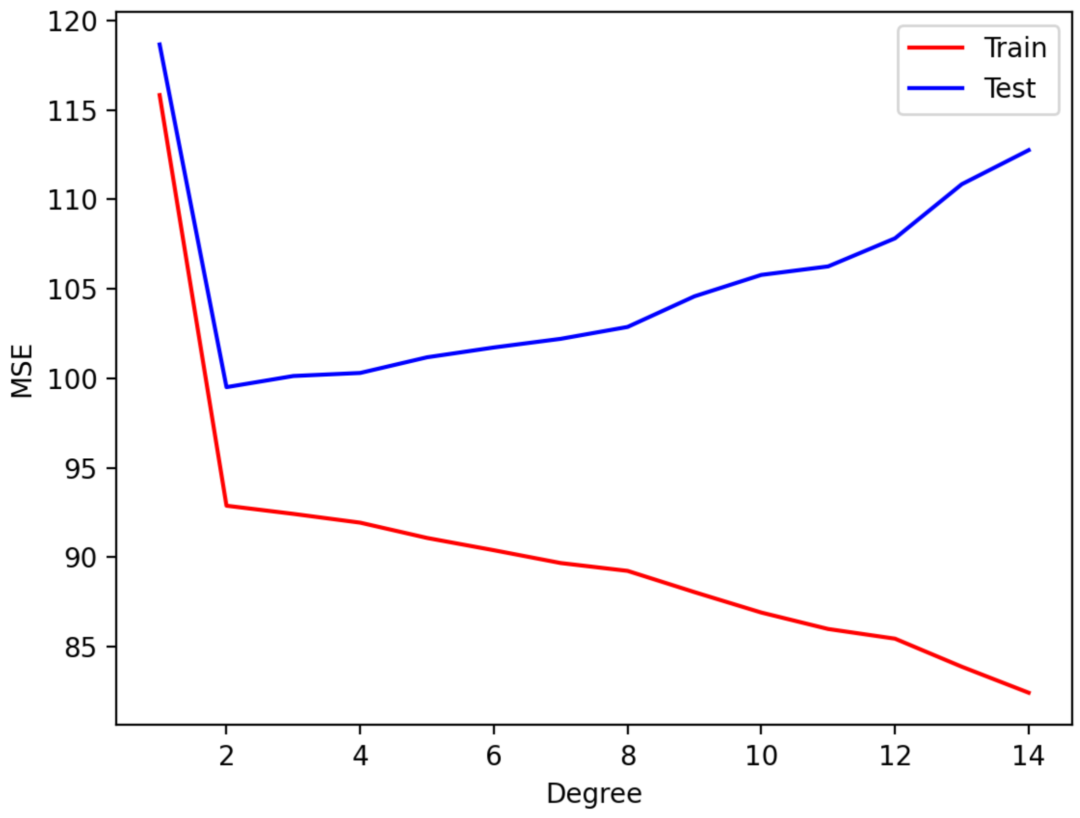
In [12]:
```
X_test, y_test = generate_data(noise_strength=10, num_points=10000)
```

For each degree from 0 to 15, compute the best fit polynomial based on the training data. You may use `PolynomialFeatures` for the feature computation, but the regression calculation must be done by hand. Then store the training error and test error respectively for each model.

In [13]:
```
train_err, test_err = [], []

for degree in range(1, 15):
    poly = PolynomialFeatures(degree, include_bias = True)
    featurize_fn = lambda x: poly.fit_transform(x)
    X = featurize_fn(X_train)
    Xt = np.transpose(X)
    XtX = np.dot(Xt, X)
    XtXinv = np.linalg.inv(XtX)
    solution = np.dot(np.dot(XtXinv, Xt), y_train)
    estimated_f = lambda x: np.dot(x, solution)
    predictions = estimated_f(featurize_fn(X_test))
    test_loss = np.sum((y_test - predictions)**2)/len(y_test)
    train_loss = np.sum((y_train - estimated_f(X))**2)/len(y_train)
    train_err.append(train_loss)
    test_err.append(test_loss)
```

Make a plot of your train and test losses. X axis should be polynomial degree and y axis should be MSE. Be sure to include a labeled legend and label each axis.

In [14]:
```
degree = [i for i in range(1, 15)]
fig = plt.figure()
plt.plot(degree, train_err, color='red', label='Train')
plt.plot(degree, test_err, color='blue', label='Test')
plt.xlabel("Degree")
plt.ylabel("MSE")
plt.legend()
plt.show()
```



This plot provides us a very clear picture of the bias/variance tradeoff in action. Note that while the train error only decreases with increasing polynomial degree, the test error starts to go back up beyond a certain point. Please explain why this phenomenon happens. Include in your explanation how bias and how variance affect (or don't affect) both train error and test error. Finally, give your recommendation for what degree you believe the polynomial to truly be.

This phenomenon happens due to overfitting, since the model has higher variance on data it hasn't seen. Variance can be seen as sort of a metric for how much the model is overfitting. As test error gets higher, the most likely indication is that variance is also getting higher in tandem.

Bias, however, will keep getting lower and lower in conjunction with the training data, since the model is getting closer and closer to the true dataset on the training dataset. I believe the polynomial is of degree 2 since test error is at a minimum at degree 2.

Part C: Bias/Variance of Extra Training Data

Now, let's see if we can ameliorate the effects of overfitting by adding additional training data. For this section, we will fit a polynomial of degree 10. Fill in the code block below. We will train various models of degree 10, each with a different training dataset size from 1000 to 30000 in steps of 1000. To reduce variance in the model fitting, we will repeat each fit 20 times and take the average of their errors to plot.

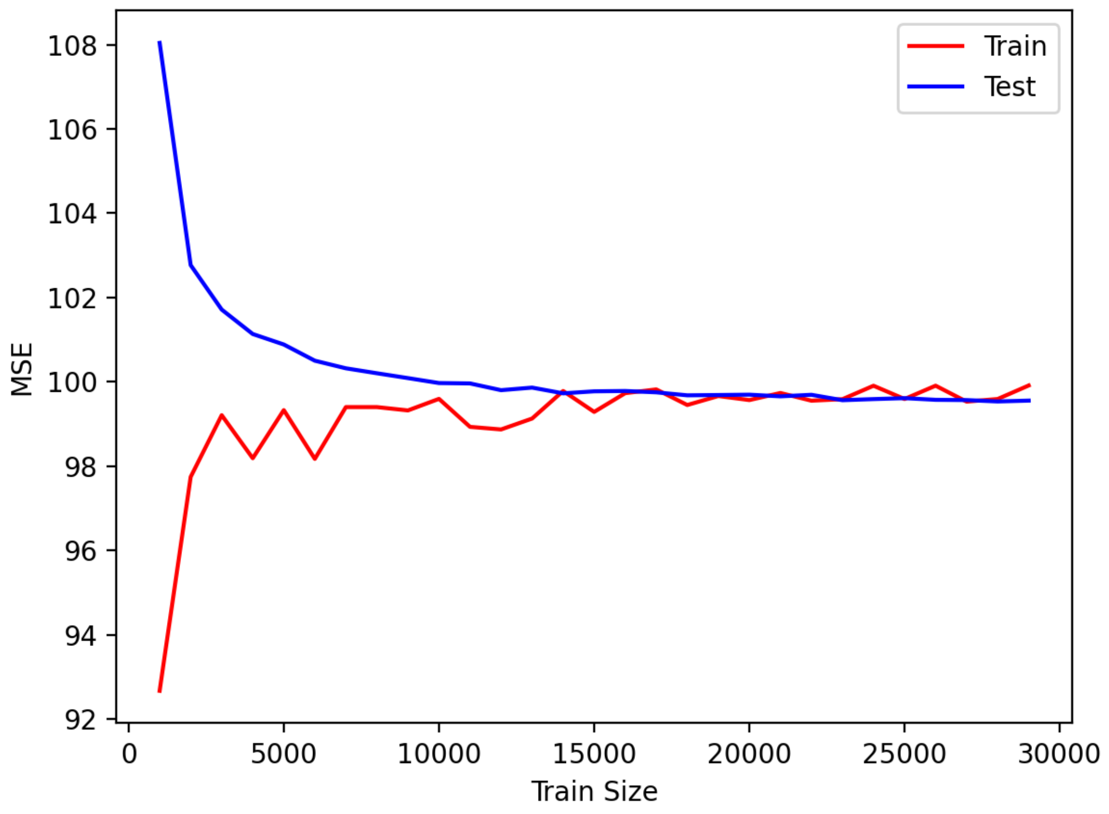
In [15]:

```
train_err, test_err = [], []

for train_size in range(1000, 30000, 1000):
    train_err_inner, test_err_inner = [], []

    for i in range(1, 20):
        X_train, y_train = generate_data(noise_strength=10, num_points=train_siz

        poly = PolynomialFeatures(10, include_bias = True)
        featurize_fn = lambda x: poly.fit_transform(x)

        X = featurize_fn(X_train)
        Xt = np.transpose(X)
        XtX = np.dot(Xt, X)
        XtXinv = np.linalg.inv(XtX)
        solution = np.dot(np.dot(XtXinv, Xt), y_train)

        estimated_f = lambda x: np.dot(x, solution)
        predictions = estimated_f(featurize_fn(X_test))
        test_loss = np.sum((y_test - predictions)**2)/len(y_test)
        train_loss = np.sum((y_train - estimated_f(X))**2)/len(y_train)
        train_err_inner.append(train_loss)
        test_err_inner.append(test_loss)

    train_err.append(np.mean(train_err_inner))
    test_err.append(np.mean(test_err_inner))
```

Make an error plot similar to the one you made before. Make sure to label the axes properly.

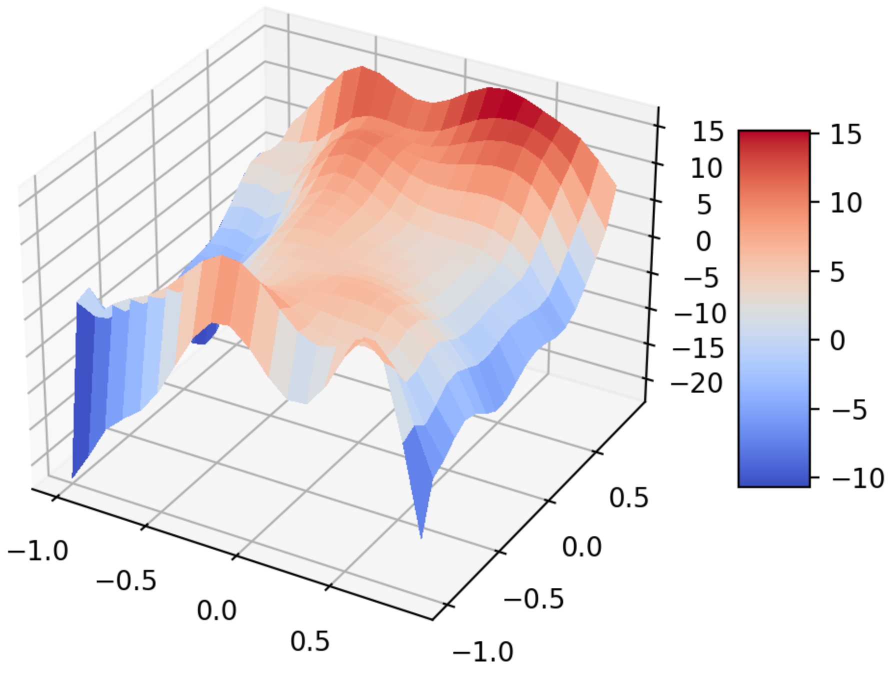
In [19]:

```
train_size = [i for i in range(1000, 30000, 1000)]
fig = plt.figure()
plt.plot(train_size, train_err, color='red', label='Train')
plt.plot(train_size, test_err, color='blue', label='Test')
plt.xlabel("Train Size")
plt.ylabel("MSE")
plt.legend()
plt.show()
```



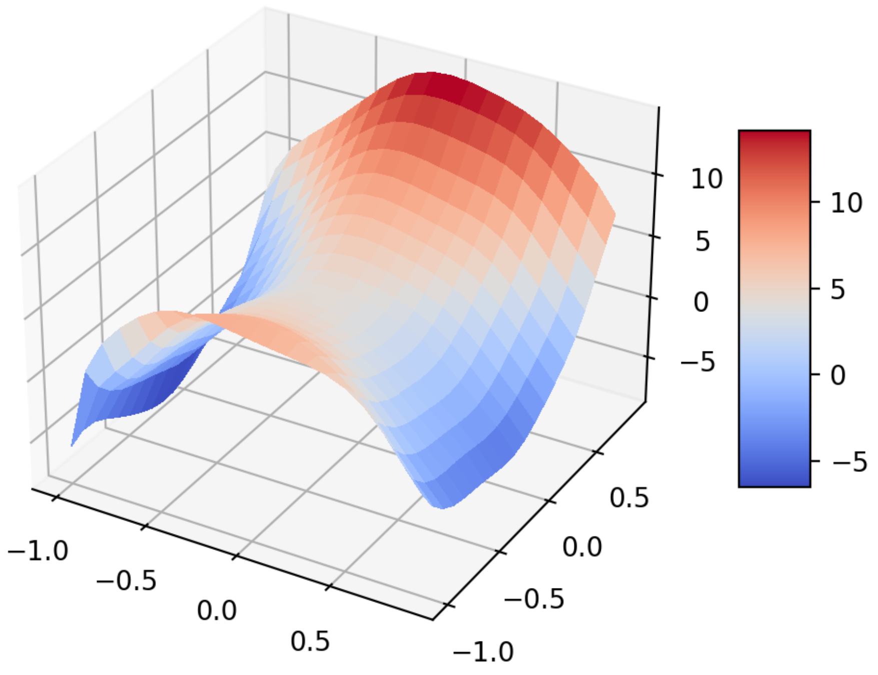
As we can see, as the size of our training set increases, the measured overfitting effect decreases quite significantly (although with diminishing returns). Now, let's see this effect visually. In the codeblock below, fit a polynomial of degree 10 trained on 1k datapoints and visualize the function.

In [20]:
```python
X_train, y_train = generate_data(noise_strength=10, num_points=1000)

poly = PolynomialFeatures(10, include_bias = True)
featurize_fn = lambda x: poly.fit_transform(x)

X = featurize_fn(X_train)
Xt = np.transpose(X)
XtX = np.dot(Xt, X)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)

estimated_f = lambda x: np.dot(x, solution)

funtion_plot(estimated_f, featurize_fn=poly.fit_transform)
```



Now, train a polynomial of degree 10 using 30k datapoints and visualize it.

In [21]:
```
X_train, y_train = generate_data(noise_strength=10, num_points=30000)

poly = PolynomialFeatures(10, include_bias = True)
featurize_fn = lambda x: poly.fit_transform(x)

X = featurize_fn(X_train)
Xt = np.transpose(X)
XtX = np.dot(Xt, X)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)

estimated_f = lambda x: np.dot(x, solution)

funtion_plot(estimated_f, featurize_fn=poly.fit_transform)
```



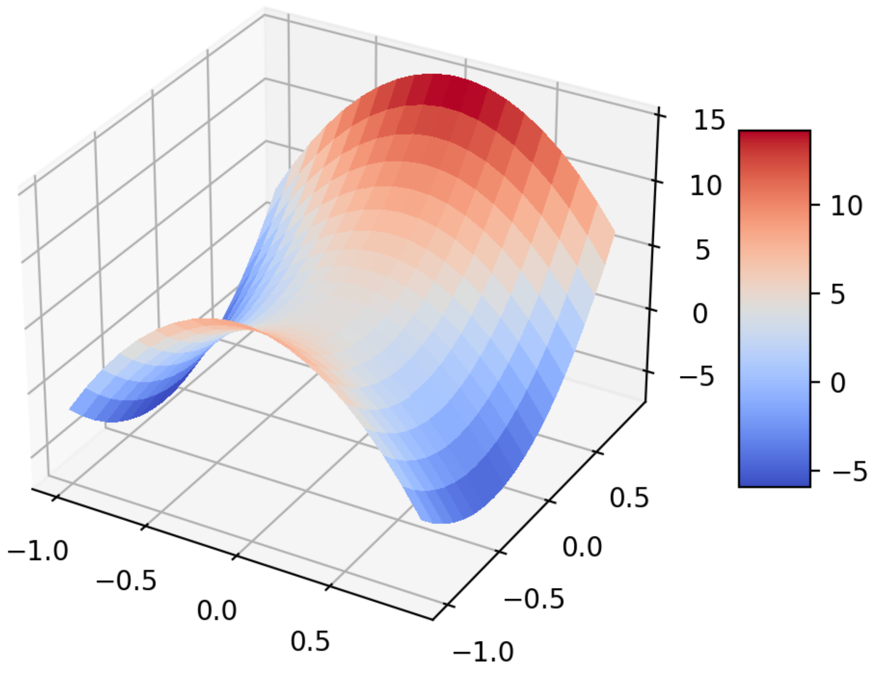
Explain why you think adding training data decreases the effect of overfitting.

The linear regression model is able to get a more accurate picture of the total possible dataset that it can operate on, and thus is able to predict with better accuracy what the correct function should actually be. In short, it looks at more data that could be closer to the test dataset.

Part D: Final answer

Based on your observations so far, write down your estimate for the true polynomial function. You may train new models, playing with the degree of the polynomial and the number of training points to come up with your answer.

In [22]:
```
featurize_fn = lambda x: [[1, r[0], r[1], r[0]*r[1], r[0]**2, r[1]**2] for r in
X = featurize_fn(X_train)
Xt = np.transpose(X)
XtX = np.dot(Xt, X)
XtXinv = np.linalg.inv(XtX)
solution = np.dot(np.dot(XtXinv, Xt), y_train)
estimated_f = lambda x: np.dot(x, solution)

funtion_plot(estimated_f, featurize_fn=featurize_fn)
```



In [23]:
```
print(solution)
```

```
[  4.88684819  -0.06848543   4.06088395   1.10099552 -11.7043694
   8.09551148]
```

I think the degree 2 polynomial did the best approximation of the function overall. For my final guess of the true model, I am going to round the coefficients in the solution for degree 2 above and translate them to my polynomial.

$$f(x_1, x_2) = 5 + 4x_1 + x_0x_1 - 12x_0^2 + 8x_1^2$$