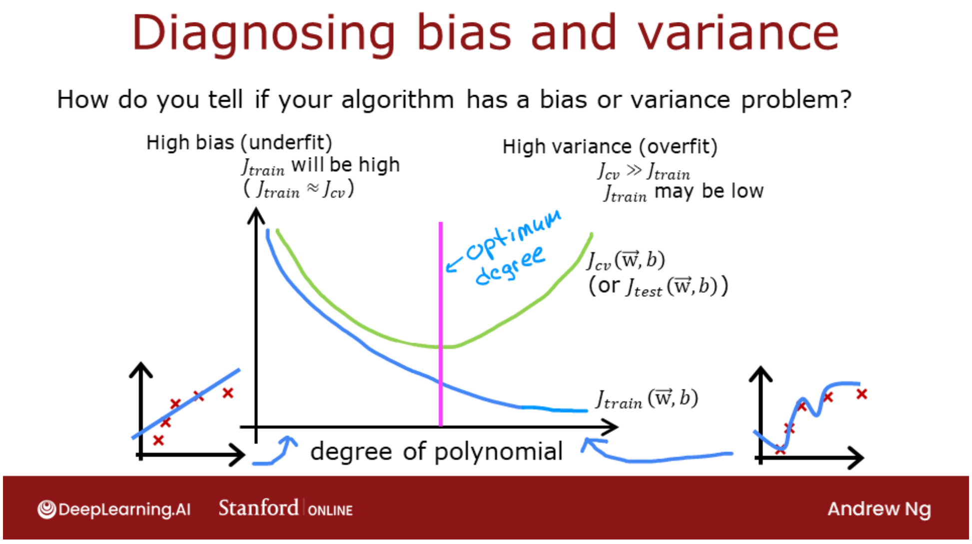
Practice Lab: Advice for Applying Machine Learning

In this lab, you will explore techniques to evaluate and improve your machine learning models.

Outline



1 - Packages

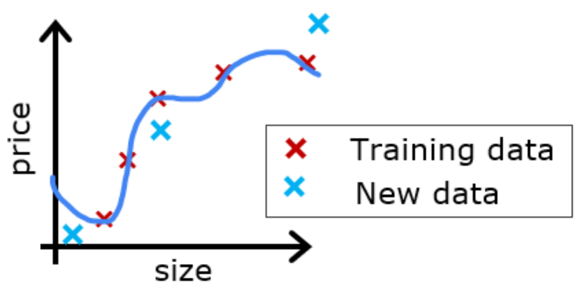
First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing Python.
- matplotlib is a popular library to plot graphs in Python.
- scikitlearn is a basic library for data mining
- tensorflow a popular platform for machine learning.

In [4]:
```
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import relu,linear
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.optimizers import Adam

import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)

from public_tests_a1 import *

tf.keras.backend.set_floatx('float64')
from assigment_utils import *

tf.autograph.set_verbosity(0)
```

2 - Evaluating a Learning Algorithm (Polynomial Regression)

Let's say you have created a machine learning model and you find it *fits* your training data very well. You're done? Not quite. The goal of creating the model was to be



$$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x^2 + \cdots + w_n x^n + b$$

able to predict
values for *new*
examples.

How can you test your model's performance on new data before deploying it?
The answer has two parts:

- Split your original data set into "Training" and "Test" sets.
  - Use the training data to fit the parameters of the model
  - Use the test data to evaluate the model on *new* data
- Develop an error function to evaluate your model.

2.1 Splitting your data set

Lectures advised reserving 20-40% of your data set for testing. Let's use an `sklearn` function train_test_split to perform the split. Double-check the shapes after running the following cell.

In [5]:
```
# Generate some data
X,y,x_ideal,y_ideal = gen_data(18, 2, 0.7)
print("X.shape", X.shape, "y.shape", y.shape)

#split the data using sklearn routine
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.33, ra
print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
print("X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
```

```
X.shape (18,) y.shape (18,)
X_train.shape (12,) y_train.shape (12,)
X_test.shape (6,) y_test.shape (6,)
```

2.1.1 Plot Train, Test sets

You can see below the data points that will be part of training (in red) are intermixed with those that the model is not trained on (test). This particular data set is a quadratic function with noise added. The "ideal" curve is shown for reference.

In [6]:
```
fig, ax = plt.subplots(1,1,figsize=(4,4))
ax.plot(x_ideal, y_ideal, "--", color = "orangered", label="y_ideal", lw=1
ax.set_title("Training, Test",fontsize = 14)
ax.set_xlabel("x")
ax.set_ylabel("y")

ax.scatter(X_train, y_train, color = "red",           label="train")
ax.scatter(X_test, y_test,   color = dlc["dlblue"],   label="test")
ax.legend(loc='upper left')
plt.show()
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

2.2 Error calculation for model evaluation, linear

2.2 Error calculation for model evaluation, linear regression

When *evaluating* a linear regression model, you average the squared error difference of the predicted values and the target values.

$$J_\text{test}(\mathbf{w},b) = \frac{1}{2m_\text{test}}\sum_{i=0}^{m_\text{test}-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}_\text{test}) - y^{(i)}_\text{test})^2 \tag{1}$$

Exercise 1

Below, create a function to evaluate the error on a data set for a linear regression model.

In [9]:

```
# UNQ_C1
# GRADED CELL: eval_mse
def eval_mse(y, yhat):
    """ 
    Calculate the mean squared error on a data set.
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of each examp
    Returns:
      err: (scalar)             
    """
    m = len(y)
    err = 0.0
    for i in range(m):
    ### START CODE HERE ### 
        err+=(y[i]-yhat[i])**2
    ### END CODE HERE ### 
    err/= 2*m
    return(err)
```

In [10]:

```
y_hat = np.array([2.4, 4.2])
y_tmp = np.array([2.3, 4.1])
eval_mse(y_hat, y_tmp)

# BEGIN UNIT TEST
test_eval_mse(eval_mse)
# END UNIT TEST
```

```
All tests passed.
```

Click for hints

```
def eval_mse(y, yhat):
    """ 
    Calculate the mean squared error on a data set.
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of
each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of
each example
    Returns:
      err: (scalar)
```

```
      err: (scalar)
    """
    m = len(y)
    err = 0.0
    for i in range(m):
        err_i  = ( (yhat[i] - y[i])**2 )
        err   += err_i
    err = err / (2*m)
    return(err)
```

2.3 Compare performance on training and test data

Let's build a high degree polynomial model to minimize training error. This will use the linear_regression functions from `sklearn` . The code is in the imported utility file if you would like to see the details. The steps below are:

- create and fit the model. ('fit' is another name for training or running gradient descent).
- compute the error on the training data.
- compute the error on the test data.

In [11]:

```
# create a model in sklearn, train on training data
degree = 10
lmodel = lin_model(degree)
lmodel.fit(X_train, y_train)

# predict on training data, find training error
yhat = lmodel.predict(X_train)
err_train = lmodel.mse(y_train, yhat)

# predict on test data, find error
yhat = lmodel.predict(X_test)
err_test = lmodel.mse(y_test, yhat)
```

The computed error on the training set is substantially less than that of the test set.

In [12]:

```
print(f"training err {err_train:0.2f}, test err {err_test:0.2f}")
```

```
training err 58.01, test err 171215.01
```

The following plot shows why this is. The model fits the training data very well. To do so, it has created a complex function. The test data was not part of the training and the model does a poor job of predicting on this data.
This model would be described as 1) is overfitting, 2) has high variance 3) 'generalizes' poorly.

In [13]:

```
# plot predictions over data range
x = np.linspace(0,int(X.max()),100)  # predict values for plot
y_pred = lmodel.predict(x).reshape(-1,1)

plt_train_test(X_train, y_train, X_test, y_test, x, y_pred, x_ideal, y_ide
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

The test set error shows this model will not work well on new data. If you use the test error to guide improvements in the model, then the model will perform well on the test data... but the test data was meant to represent *new* data. You need yet another set of data to test new data performance.

The proposal made during lecture is to separate data into three groups. The distribution of training, cross-validation and test sets shown in the below table is a typical distribution, but can be varied depending on the amount of data available.

data	% of total	Description
training	60	Data used to tune model parameters w and b in training or fitting
cross-validation	20	Data used to tune other model parameters like degree of polynomial, regularization or the architecture of a neural network.
test	20	Data used to test the model after tuning to gauge performance on new data

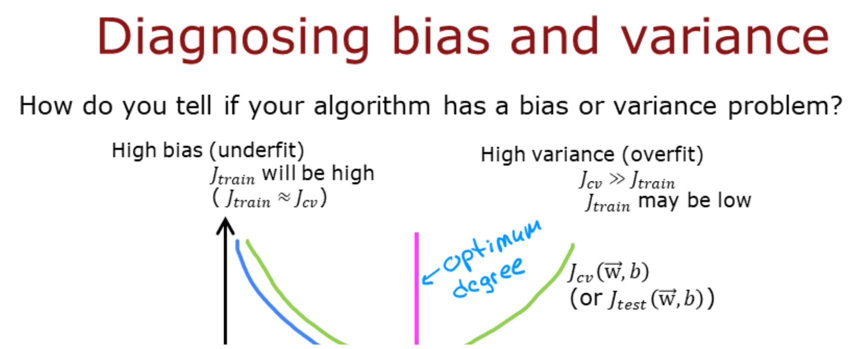
Let's generate three data sets below. We'll once again use `train_test_split` from `sklearn` but will call it twice to get three splits:

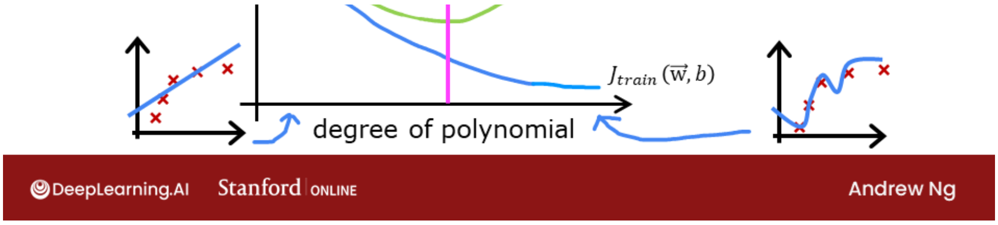
In [14]:

```python
# Generate  data
X,y, x_ideal,y_ideal = gen_data(40, 5, 0.7)
print("X.shape", X.shape, "y.shape", y.shape)

#split the data using sklearn routine
X_train, X_, y_train, y_ = train_test_split(X,y,test_size=0.40, random_sta
X_cv, X_test, y_cv, y_test = train_test_split(X_,y_,test_size=0.50, randor
print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
print("X_cv.shape", X_cv.shape, "y_cv.shape", y_cv.shape)
print("X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
```

```
X.shape (40,) y.shape (40,)
X_train.shape (24,) y_train.shape (24,)
X_cv.shape (8,) y_cv.shape (8,)
X_test.shape (8,) y_test.shape (8,)
```

3 - Bias and Variance





Above, it was clear the degree of the polynomial model was too high. How can you choose a good value? It turns out, as shown in the diagram, the training and cross-validation performance can provide guidance. By trying a range of degree values, the training and cross-validation performance can be evaluated. As the degree becomes too large, the cross-validation performance will start to degrade relative to the training performance. Let's try this on our example.

3.1 Plot Train, Cross-Validation, Test

You can see below the datapoints that will be part of training (in red) are intermixed with those that the model is not trained on (test and cv).

In [15]:

```
fig, ax = plt.subplots(1,1,figsize=(4,4))
ax.plot(x_ideal, y_ideal, "--", color = "orangered", label="y_ideal", lw=
ax.set_title("Training, CV, Test",fontsize = 14)
ax.set_xlabel("x")
ax.set_ylabel("y")

ax.scatter(X_train, y_train, color = "red",           label="train")
ax.scatter(X_cv, y_cv,       color = dlc["dlorange"], label="cv")
ax.scatter(X_test, y_test,   color = dlc["dlblue"],   label="test")
ax.legend(loc='upper left')
plt.show()
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

3.2 Finding the optimal degree

In previous labs, you found that you could create a model capable of fitting complex curves by utilizing a polynomial (See Course1, Week2 Feature Engineering and Polynomial Regression Lab). Further, you demonstrated that by increasing the *degree* of the polynomial, you could *create* overfitting. (See Course 1, Week3, Over-Fitting Lab). Let's use that knowledge here to test our ability to tell the difference between over-fitting and under-fitting.

Let's train the model repeatedly, increasing the degree of the polynomial each iteration. Here, we're going to use the scikit-learn linear regression model for speed and simplicity.

In [16]:

```
max_degree = 9
err_train = np.zeros(max_degree)
err_cv = np.zeros(max_degree)
x = np.linspace(0,int(X.max()),100)
y_pred = np.zeros((100,max_degree))  #columns are lines to plot
```

```
y_pred = np.zeros((100,max_degree))  #columns are lines to plot

for degree in range(max_degree):
    lmodel = lin_model(degree+1)
    lmodel.fit(X_train, y_train)
    yhat = lmodel.predict(X_train)
    err_train[degree] = lmodel.mse(y_train, yhat)
    yhat = lmodel.predict(X_cv)
    err_cv[degree] = lmodel.mse(y_cv, yhat)
    y_pred[:,degree] = lmodel.predict(x)

optimal_degree = np.argmin(err_cv)+1
```

Let's plot the result:

In [17]:
```
plt.close("all")
plt_optimal_degree(X_train, y_train, X_cv, y_cv, x, y_pred, x_ideal, y_ide
                   err_train, err_cv, optimal_degree, max_degree)
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

The plot above demonstrates that separating data into two groups, data the model is trained on and data the model has not been trained on, can be used to determine if the model is underfitting or overfitting. In our example, we created a variety of models varying from underfitting to overfitting by increasing the degree of the polynomial used.

- On the left plot, the solid lines represent the predictions from these models. A polynomial model with degree 1 produces a straight line that intersects very few data points, while the maximum degree hews very closely to every data point.
- on the right:
  - the error on the trained data (blue) decreases as the model complexity increases as expected
  - the error of the cross-validation data decreases initially as the model starts to conform to the data, but then increases as the model starts to over-fit on the training data (fails to *generalize*).

It's worth noting that the curves in these examples as not as smooth as one might draw for a lecture. It's clear the specific data points assigned to each group can change your results significantly. The general trend is what is important.

3.3 Tuning Regularization.

In previous labs, you have utilized *regularization* to reduce overfitting. Similar to degree, one can use the same methodology to tune the regularization parameter lambda (λ).

Let's demonstrate this by starting with a high degree polynomial and varying the regularization parameter.

In [18]:

```
lambda_range = np.array([0.0, 1e-6, 1e-5, 1e-4,1e-3,1e-2, 1e-1,1,10,100])
num_steps = len(lambda_range)
degree = 10
err_train = np.zeros(num_steps)
err_cv = np.zeros(num_steps)
x = np.linspace(0,int(X.max()),100)
y_pred = np.zeros((100,num_steps))  #columns are lines to plot

for i in range(num_steps):
    lambda_= lambda_range[i]
    lmodel = lin_model(degree, regularization=True, lambda_=lambda_)
    lmodel.fit(X_train, y_train)
    yhat = lmodel.predict(X_train)
    err_train[i] = lmodel.mse(y_train, yhat)
    yhat = lmodel.predict(X_cv)
    err_cv[i] = lmodel.mse(y_cv, yhat)
    y_pred[:,i] = lmodel.predict(x)

optimal_reg_idx = np.argmin(err_cv)
```

In [19]:

```
plt.close("all")
plt_tune_regularization(X_train, y_train, X_cv, y_cv, x, y_pred, err_trai
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

Above, the plots show that as regularization increases, the model moves from a high variance (overfitting) model to a high bias (underfitting) model. The vertical line in the right plot shows the optimal value of lambda. In this example, the polynomial degree was set to 10.

3.4 Getting more data: Increasing Training Set Size (m)

When a model is overfitting (high variance), collecting additional data can improve performance. Let's try that here.

In [20]:

```
X_train, y_train, X_cv, y_cv, x, y_pred, err_train, err_cv, m_range,degree
plt_tune_m(X_train, y_train, X_cv, y_cv, x, y_pred, err_train, err_cv, m_r
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

The above plots show that when a model has high variance and is overfitting, adding more examples improves performance. Note the curves on the left plot. The final curve with the highest value of m is a smooth curve that is in the center of the data. On the right, as the number of examples increases, the performance of the training set and cross-validation set converge to similar values. Note that the curves are not as smooth as one might see in a lecture. That is to be expected. The trend remains clear: more data improves generalization.

> Note that adding more examples when the model has high bias (underfitting) does not improve performance.

4 - Evaluating a Learning Algorithm (Neural Network)

Above, you tuned aspects of a polynomial regression model. Here, you will work with a neural network model. Let's start by creating a classification data set.

4.1 Data Set

Run the cell below to generate a data set and split it into training, cross-validation (CV) and test sets. In this example, we're increasing the percentage of cross-validation data points for emphasis.

In [21]:
```
# Generate and split data set
X, y, centers, classes, std = gen_blobs()

# split the data. Large CV population for demonstration
X_train, X_, y_train, y_ = train_test_split(X,y,test_size=0.50, random_st
X_cv, X_test, y_cv, y_test = train_test_split(X_,y_,test_size=0.20, randor
print("X_train.shape:", X_train.shape, "X_cv.shape:", X_cv.shape, "X_test
```

```
X_train.shape: (400, 2) X_cv.shape: (320, 2) X_test.shape: (80, 2)
```

In [22]:
```
plt_train_eq_dist(X_train, y_train,classes, X_cv, y_cv, centers, std)
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', '
home'), ('Back', 'Back to previous …
```

Above, you can see the data on the left. There are six clusters identified by color. Both training points (dots) and cross-validataion points (triangles) are shown. The interesting points are those that fall in ambiguous locations where either cluster might consider them members. What would you expect a neural network model to do? What would be an example of overfitting? underfitting?
On the right is an example of an 'ideal' model, or a model one might create knowing the source of the data. The lines represent 'equal distance' boundaries where the distance between center points is equal. It's worth noting that this model would "misclassify" roughly 8% of the total data set.

4.2 Evaluating categorical model by calculating classification error

The evaluation function for categorical models used here is simply the fraction of incorrect predictions:

$$J_{cv} = \frac{1}{m}\sum_{i=0}^{m-1}\begin{cases}1, & \text{if } \hat{y}^{(i)} \neq y^{(i)}\\ 0, & \text{otherwise}\end{cases}$$

Exercise 2

Below, complete the routine to calculate classification error. Note, in this lab, target values are the index of the category and are not one-hot encoded.

In [34]:

```
# UNQ_C2
# GRADED CELL: eval_cat_err
def eval_cat_err(y, yhat):
    """
    Calculate the categorization error
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of each examp
    Returns:|
      cerr: (scalar)
    """
    m = len(y)
    incorrect = 0
    for i in range(m):
    ### START CODE HERE ###
        if(yhat[i]!=y[i]):
            incorrect+=1
    ### END CODE HERE ###
    cerr=incorrect/m
    return(cerr)
```

In [35]:

```
y_hat = np.array([1, 2, 0])
y_tmp = np.array([1, 2, 3])
print(f"categorization error {np.squeeze(eval_cat_err(y_hat, y_tmp)):0.3f}
y_hat = np.array([[1], [2], [0], [3]])
y_tmp = np.array([[1], [2], [1], [3]])
print(f"categorization error {np.squeeze(eval_cat_err(y_hat, y_tmp)):0.3f}

# BEGIN UNIT TEST
test_eval_cat_err(eval_cat_err)
# END UNIT TEST
# BEGIN UNIT TEST
test_eval_cat_err(eval_cat_err)
# END UNIT TEST
```

```
categorization error 0.333, expected:0.333
categorization error 0.250, expected:0.250
 All tests passed.
 All tests passed.
```

Click for hints

```
def eval_cat_err(y, yhat):
    """
    Calculate the categorization error
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of
each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of
each example
    Returns:|
      cerr: (scalar)
    """
    m = len(y)
    incorrect = 0
```