**4a**

######code#######

def linear\_regression(X, Y, Xs\_test, Ys\_test):

"""

This function performs linear regression.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

Xs\_test: independent variables in test data.

Ys\_test: dependent variables in test data.

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

W = inv(X.T @ X) @ X.T @ Y

mses = []

for X\_test, Y\_test in zip(Xs\_test, Ys\_test):

Y\_pred = X\_test @ W

mse = np.mean(np.sqrt(np.sum((Y\_pred - Y\_test)\*\*2, axis=1)))

mses.append(mse)

return mses

def poly\_regression\_second(X, Y, Xs\_test, Ys\_test):

"""

This function performs second order polynomial regression.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

Xs\_test: independent variables in test data.

Ys\_test: dependent variables in test data.

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

X\_poly = generate\_polynomial\_features(X, 2)

Xs\_test\_poly = []

for X\_test in Xs\_test:

Xs\_test\_poly.append(generate\_polynomial\_features(X\_test, 2))

return linear\_regression(X\_poly, Y, Xs\_test\_poly, Ys\_test)

def poly\_regression\_cubic(X, Y, Xs\_test, Ys\_test):

"""

This function performs third order polynomial regression.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

Xs\_test: independent variables in test data.

Ys\_test: dependent variables in test data.

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

X\_poly = generate\_polynomial\_features(X, 3)

Xs\_test\_poly = []

for X\_test in Xs\_test:

Xs\_test\_poly.append(generate\_polynomial\_features(X\_test, 3))

return linear\_regression(X\_poly, Y, Xs\_test\_poly, Ys\_test)

def neural\_network(X, Y, Xs\_test, Ys\_test):

"""

This function performs neural network prediction.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

Xs\_test: independent variables in test data.

Ys\_test: dependent variables in test data.

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

# Build the model

model = Model(X.shape[1])

model.addLayer(DenseLayer(100, ReLUActivation()))

model.addLayer(DenseLayer(100, ReLUActivation()))

model.addLayer(DenseLayer(Y.shape[1],LinearActivation()))

model.initialize(QuadraticCost())

##train the model and plot learning curve

##standardize the features first!!!

scaler = StandardScaler()

X\_trans = scaler.fit\_transform(X)

hist = model.train(X\_trans, Y, 2000, GDOptimizer(eta=0.001))

# plt.plot(hist)

# plt.title('Learning curve')

# plt.show()

##evaluate the model

mses = []

for X\_test, Y\_test in zip(Xs\_test, Ys\_test):

X\_test\_trans = scaler.transform(X\_test)

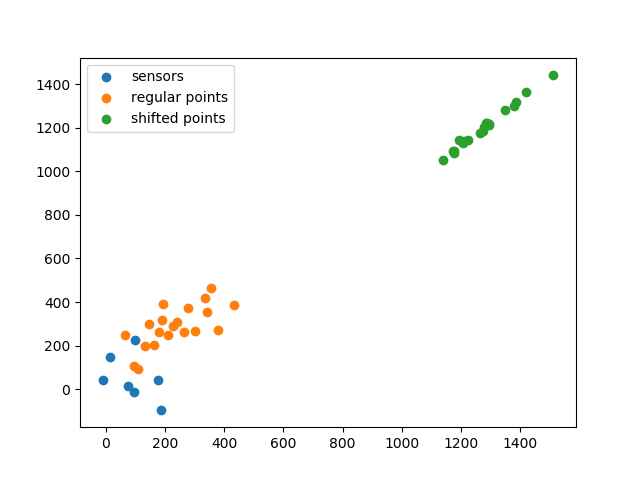
Y\_pred = model.predict(X\_test\_trans)

mse = np.mean(np.sqrt(np.sum((Y\_pred - Y\_test)\*\*2, axis=1)))

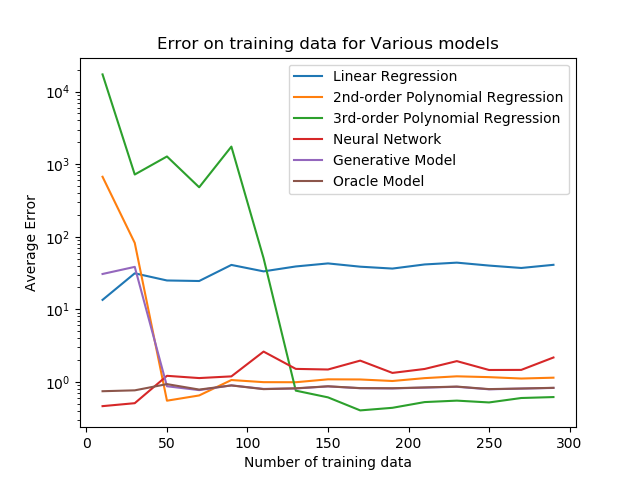
mses.append(mse)

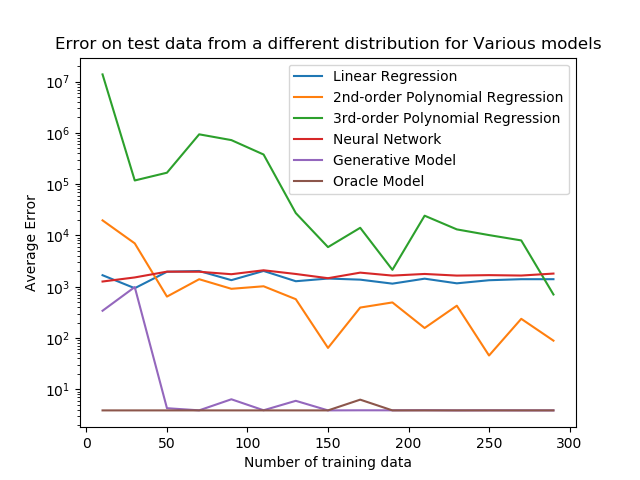
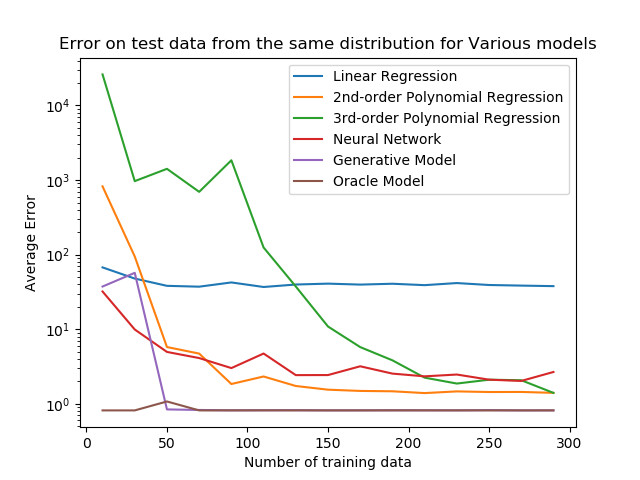
return mses

**4b**

****

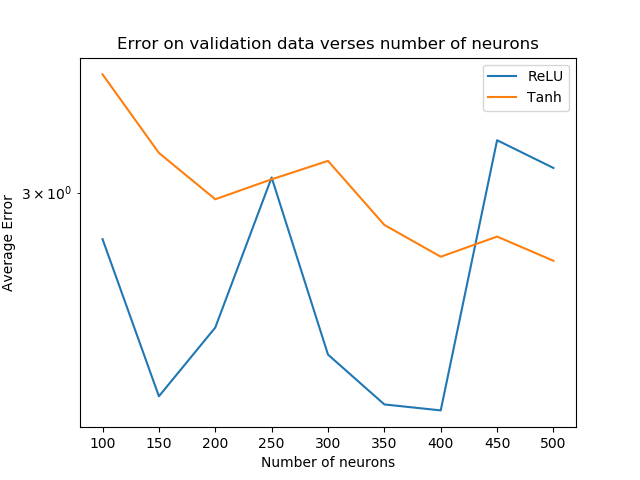
**4c**

****

****

* Oracle model works best for both test datasets since we know exact sensor location, and only need to infer object location by distance. Though for different test data, it doesn’t work as good as for similar test data. Runs slow.
* Generative model works good if we have training set larger than 50, since we can infer the sensor location reasonably good if we have 50 training object locations, and in that case, generative model works as good as oracle model. Runs slow.
* Linear regression has high bias and is underfitting.
* 2nd-order and 3rd-order polynomial regression works better with more training data, 3rd-order polynomial is overfitting since test error is larger than training error. Both cannot be generalized to different test data. Runs fast.
* The performance of neutral net is comparable to 2nd-order polynomial regression. It cannot be generalized to different test data. Efficiency depends on epochs and learning rate.

**4d**

****

* It seems ReLU works better than tanh as nonlinearity. For ReLU, the best choice seems to be 150, and for tanh, the best choice is 400.

########code#########

def neural\_network(X, Y, X\_test, Y\_test, num\_neurons, activation):

"""

This function performs neural network prediction.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

X\_test: independent variables in test data.

Y\_test: dependent variables in test data.

num\_neurons: number of neurons in each layer

activation: type of activation, ReLU or tanh

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

# Build the model

if activation == "ReLU":

activation\_func = ReLUActivation

if activation == "tanh":

activation\_func = TanhActivation

model = Model(X.shape[1])

model.addLayer(DenseLayer(num\_neurons, activation\_func()))

model.addLayer(DenseLayer(num\_neurons, activation\_func()))

model.addLayer(DenseLayer(Y.shape[1],LinearActivation()))

model.initialize(QuadraticCost())

##train the model and plot learning curve

##standardize the features first!!!

scaler = StandardScaler()

X\_trans = scaler.fit\_transform(X)

hist = model.train(X\_trans, Y, 2000, GDOptimizer(eta=0.001))

# plt.plot(hist)

# plt.title('Learning curve')

# plt.show()

##evaluate the model

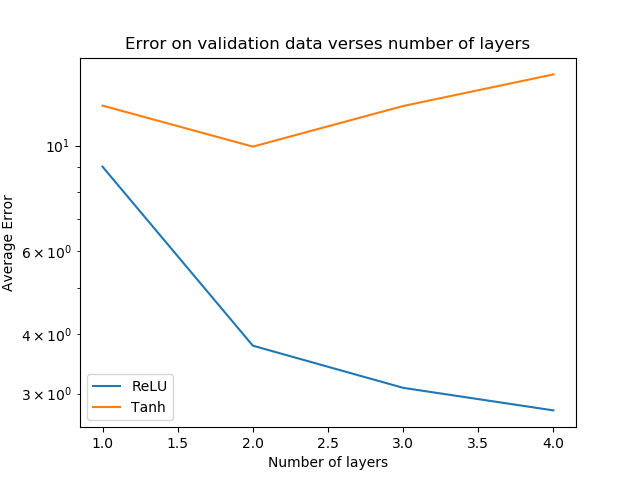
X\_test\_trans = scaler.transform(X\_test)

Y\_pred = model.predict(X\_test\_trans)

mse = np.mean(np.sqrt(np.sum((Y\_pred - Y\_test)\*\*2, axis=1)))

return mse

**4e**

****

* Again, ReLU works better than tanh. For ReLU, the deeper the better. For tanh, layers of 2 works the best.

########code#########

def get\_num\_neurons(N, k):

"""

given number of parameters N and number of hidden layer k, returns the number of neurons per layer

"""

if k == 1:

return int(N/10.0)

else:

return int((sqrt(4\*N\*(k-1)) - (k+9))/2/(k-1))

##small test of get\_num\_neurons

# N = 10000

# for k in range(1, 5):

# l = get\_num\_neurons(N, k)

# print(l)

# print((k-1)\*l\*l + (k+9)\*l +2)

def neural\_network(X, Y, X\_test, Y\_test, num\_layers, activation):

"""

This function performs neural network prediction.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

X\_test: independent variables in test data.

Y\_test: dependent variables in test data.

num\_layers: number of layers in neural network

activation: type of activation, ReLU or tanh

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

num\_neurons = get\_num\_neurons(10000, num\_layers)

# Build the model

if activation == "ReLU":

activation\_func = ReLUActivation

if activation == "tanh":

activation\_func = TanhActivation

model = Model(X.shape[1])

for i in range(num\_layers):

model.addLayer(DenseLayer(num\_neurons, activation\_func()))

model.addLayer(DenseLayer(Y.shape[1],LinearActivation()))

model.initialize(QuadraticCost())

##train the model and plot learning curve

##standardize the features first!!!

scaler = StandardScaler()

X\_trans = scaler.fit\_transform(X)

hist = model.train(X\_trans, Y, 2000, GDOptimizer(eta=0.0001))

# plt.plot(hist)

# plt.title('Learning curve')

# plt.show()

##evaluate the model

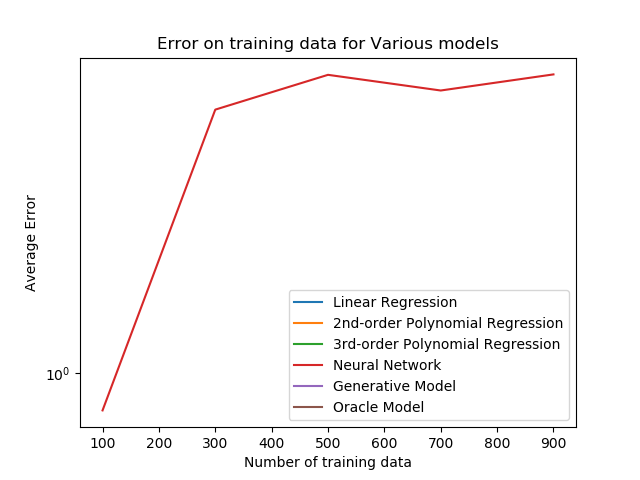
X\_test\_trans = scaler.transform(X\_test)

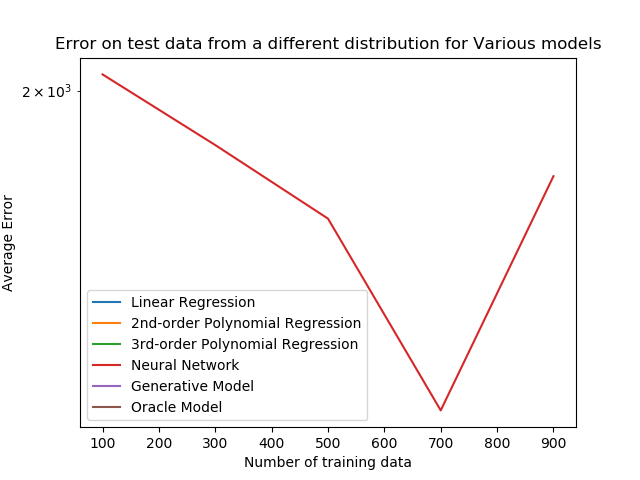
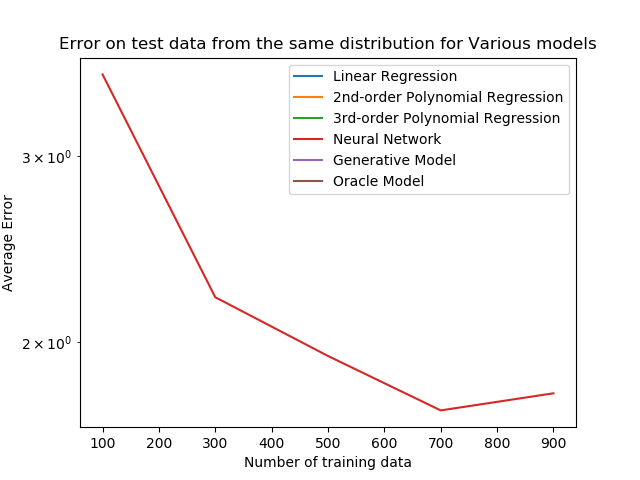
Y\_pred = model.predict(X\_test\_trans)

mse = np.mean(np.sqrt(np.sum((Y\_pred - Y\_test)\*\*2, axis=1)))

return mse

**4f**

****

****

* We cannot tune the hyper-parameters so that neural net works for dissimilar test data, even with more training data.
* Neural net learns the mapping from distances to locations. In training set, the objects are close to sensors, we have less local minimums. The mapping can be easily learned.
* However, when objects are far away from sensors, it creates a lot of local minimums, and the mapping between distance and location is not longer unique. Thus the afore-learned mapping cannot be generalized.

[Train, Validation, different\_Validation] for different number of training data:

[0.9456467906713405, 3.5849437201489303, 2047.8608505741149]

0th Experiment with 100 samples done...

[1.4873403810570629, 2.20476225997843, 1846.7085243459535]

0th Experiment with 300 samples done...

[1.5673367225277925, 1.9395501520699385, 1658.2617074790937]

0th Experiment with 500 samples done...

[1.5307813720713122, 1.7224134796161534, 1252.525280555132]

0th Experiment with 700 samples done...

[1.568350574413478, 1.7878546891328082, 1764.5032886336996]

0th Experiment with 900 samples done...

########code################

def neural\_network(X, Y, Xs\_test, Ys\_test, num\_layers, eta=0.0001, epochs=2000):

"""

This function performs neural network prediction.

Input:

X: independent variables in training data.

Y: dependent variables in training data.

X\_test: independent variables in test data.

Y\_test: dependent variables in test data.

num\_layers: number of layers in neural network

eta: learning rate

epochs: how many epochs to run

Output:

mse: Mean square error on test data.

"""

## YOUR CODE HERE

#################

num\_neurons = get\_num\_neurons(10000, num\_layers)

# Build the model

model = Model(X.shape[1])

for i in range(num\_layers):

model.addLayer(DenseLayer(num\_neurons, ReLUActivation()))

model.addLayer(DenseLayer(Y.shape[1],LinearActivation()))

model.initialize(QuadraticCost())

##train the model and plot learning curve

##standardize the features first!!!

scaler = StandardScaler()

X\_trans = scaler.fit\_transform(X)

hist = model.train(X\_trans, Y, epochs, GDOptimizer(eta=eta))

# plt.plot(hist)

# plt.title('Learning curve')

# plt.show()

##evaluate the model

mses = []

for X\_test, Y\_test in zip(Xs\_test, Ys\_test):

X\_test\_trans = scaler.transform(X\_test)

Y\_pred = model.predict(X\_test\_trans)

mse = np.mean(np.sqrt(np.sum((Y\_pred - Y\_test)\*\*2, axis=1)))

mses.append(mse)

return mses