CS189

1. (a) By myself. consulting piazza.

(b) I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

Yao Cao

HW9. Yao Cai 25933599

2. (a) suppose class i $P(Y=i|x) \geq P(Y=j|x)$ which can always be found.

$$f_{opt}(x) = \begin{cases} i & P(Y=i|x) \geq 1-\frac{\lambda_d}{\lambda_c} \\ c+1 & P(Y=i|x) \leq 1-\frac{\lambda_d}{\lambda_c} \end{cases}$$

with $P(Y=i|x) \geq 1-\frac{\lambda_d}{\lambda_c}$

$f_{opt}(x) = i$

$$\begin{aligned} R(f(x)|x) &= L(f(x),1)P(Y=1|x) + \cdots + L(f(x),c)P(Y=c|x) \\ &= L(i,1)P(Y=1|x) + \cdots + L(i,c)P(Y=c|x) \\ &= \lambda_c + \lambda_c + \cdots + 0 + \cdots + \lambda_c \\ &= \lambda_c(1-P(Y=i|x)) \\ &\leq \lambda_c\left(1-1+\frac{\lambda_d}{\lambda_c}\right) = \lambda_d \end{aligned}$$

with $P(Y=i|x) \leq 1-\frac{\lambda_d}{\lambda_c}$

$f_{opt}(x) = c+1$

$$\begin{aligned} R(f(x)|x) &= L(c+1,1)P(Y=1|x) + \cdots + L(c+1,c)P(Y=c|x) \\ &= \lambda_d \leq \lambda_c(1-P(Y=i|x)) \end{aligned}$$

$$\Rightarrow R(f_{opt}(x)|x) = \min(\lambda_d, \lambda_c(1-P(Y=i|x)))$$

Suppose we didn't pick i or $c+1$, but pick $j \neq i$ with $P(Y=j|x) \leq P(Y=i|x)$

$f(x) = j$

$$\begin{aligned} R(f(x)|x) &= L(j,1)P(Y=1|x) + \cdots + L(j,c)P(Y=c|x) \\ &= \lambda_c(1-P(Y=j|x)) \geq \lambda_c(1-P(Y=i|x)) \geq \min(\lambda_d, \lambda_c(1-P(Y=c|x))) \end{aligned}$$

So indeed $f_{opt}(x)$ is the best policy.

(b) if $\lambda_d = 0$.

then $f_{opt}(x) = c+1$

$$R(f(x)|x) = R(c+1|x) = \sum_{i=1}^{c} L(c+1, i)\, P(Y=i|x) = 0.$$

If being uncertain doesn't incur any loss, the best policy is to always being uncertain. (too conservative)

if $\lambda_d > \lambda_c$, then under no circumstances will one choose doubt since it always incurs the largest loss. The best policy in this case never chooses doubt. (too risky)

Intuitively, wrong choice is more wrong than no choice, so the loss for wrong choice should be larger than the loss for no choice. $(\lambda_c > \lambda_d)$

MLE :

3. (a) $X|L=1 \sim N(\mu_1, \Sigma)$

$$P(X|L=1) = \frac{1}{\sqrt{|\det(\Sigma)|}} \frac{1}{(\sqrt{2\pi})^d} e^{-\frac{1}{2}(X-\mu_1)^T \Sigma^{-1}(X-\mu_1)}$$

$X|L=2 \sim N(\mu_2, \Sigma)$

$$P(X|L=2) = \frac{1}{\sqrt{|\det(\Sigma)|}} \frac{1}{(\sqrt{2\pi})^d} e^{-\frac{1}{2}(X-\mu_2)^T \Sigma^{-1}(X-\mu_2)}$$

$$\frac{P(X|L=1)}{P(X|L=2)} = e^{-\frac{1}{2}(X^T\Sigma^{-1}X + \mu_1^T\Sigma^{-1}\mu_1 - 2\mu_1^T\Sigma^{-1}X) + \frac{1}{2}(X^T\Sigma^{-1}X + \mu_2^T\Sigma^{-1}\mu_2 - 2\mu_2^T\Sigma^{-1}X)}$$

$$= e^{-\frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_2^T\Sigma^{-1}\mu_2 - 2(\mu_1^T - \mu_2^T)\Sigma^{-1}X)}$$

choose $L=1$ if $\frac{P(X|L=1)}{P(X|L=2)} > 1$

choose $L=2$ otherwise

$$-\frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_2^T\Sigma^{-1}\mu_2 - 2(\mu_1^T - \mu_2^T)\Sigma^{-1}X) > 0$$

$$\Rightarrow 2(\mu_1^T - \mu_2^T)\Sigma^{-1}X > \mu_1^T\Sigma^{-1}\mu_1 - \mu_2^T\Sigma^{-1}\mu_2$$

$$\boxed{\hat{L}_{MLE} = \begin{cases} 1 & \text{if } (\mu_1^T - \mu_2^T)\Sigma^{-1}(X - \frac{1}{2}(\mu_1+\mu_2)) > 0 \\ 2 & \text{otherwise} \end{cases}}$$

MAP : $P(L=1|X) = \frac{P(X|L=1)P(L=1)}{P(X)} = \frac{P(X|L=1)\pi_1}{P(X)}$

$P(L=2|X) = \frac{P(X|L=2)\pi_2}{P(X)}$

if $\pi_2 = \pi_1 = 0.5$, then these two rules are the same

$$\frac{P(L=1|X)}{P(L=2|X)} = \frac{P(X|L=1)\pi_1}{P(X|L=2)\pi_2} > 1$$

$$\boxed{\hat{L}_{MAP} = \begin{cases} 1 & \text{if } (\mu_1^T - \mu_2^T)\Sigma^{-1}(X - \frac{1}{2}(\mu_1+\mu_2)) > \log(\frac{\pi_2}{\pi_1}) \\ 2 & \text{otherwise} \end{cases}}$$

(b) $\Sigma_{XX} = E_X[(X - E_X(X))(X - E_X(X))^T]$
$= E_X(XX^T) - E_X(X)E_X(X)^T$

$E_X(XX^T) = E_Y E(XX^T | Y)$
$= \frac{1}{2}E(XX^T | Y = \binom{1}{0}) + \frac{1}{2}E(XX^T | Y = \binom{0}{1})$

$X - u_1 | Y = \binom{1}{0} \sim N(0, \Sigma)$
$\Rightarrow E_{X|\binom{1}{0}}((X - u_1)(X - u_1)^T) = \Sigma$
$\Rightarrow E(XX^T | Y = \binom{1}{0}) = \Sigma + u_1 u_1^T$

$E_X(XX^T) = \frac{1}{2}(\Sigma + u_1 u_1^T) + \frac{1}{2}(\Sigma + u_2 u_2^T)$
$= \Sigma + \frac{1}{2}(u_1 u_1^T + u_2 u_2^T)$

$E_X(X) = E_Y E(X|Y)$
$= \frac{1}{2}E(X|\binom{1}{0}) + \frac{1}{2}E(X|\binom{0}{1})$

$= \frac{1}{2}u_1 + \frac{1}{2}u_2$

$E_X(X)E_X(X)^T = \frac{1}{4}(u_1 + u_2)(u_1 + u_2)^T$
$= \frac{1}{4}(u_1 u_1^T + u_2 u_2^T + u_1 u_2^T + u_2 u_1^T)$

$\Sigma_{XX} = \Sigma + \frac{1}{2}(u_1 u_1^T + u_2 u_2^T) - \frac{1}{4}(u_1 u_1^T + u_2 u_2^T + u_1 u_2^T + u_2 u_1^T)$

$= \Sigma + \frac{1}{4}(u_1 u_1^T + u_2 u_2^T - u_1 u_2^T - u_2 u_1^T)$

$= \Sigma + \left(\frac{1}{2}(u_1 - u_2)\right)\left(\frac{1}{2}(u_1 - u_2)^T\right)$

$$\Sigma_{XY} = E((X-E(X))(Y-E(Y))^T)$$
$$= E(XY^T) - E(X)E(Y)^T$$

$$E(XY^T) = E_Y E(XY^T \mid Y)$$
$$= \frac{1}{2}E(XY^T \mid \begin{pmatrix}1\\0\end{pmatrix}) + \frac{1}{2}E(XY^T \mid \begin{pmatrix}0\\1\end{pmatrix})$$
$$= \frac{1}{2}E(X \mid \begin{pmatrix}1\\0\end{pmatrix})(1 \ \ 0) + \frac{1}{2}E(X \mid \begin{pmatrix}0\\1\end{pmatrix})(0 \ \ 1)$$
$$= \frac{1}{2}u_1(1 \ \ 0) + \frac{1}{2}u_2(0 \ \ 1)$$
$$= \frac{1}{2}(u_1 \ \ u_2)$$

$$\Sigma_{YY} = E[(Y-E(Y))(Y-E(Y))^T]$$
$$= E(YY^T) - E(Y)E(Y)^T$$
$$= \frac{1}{2}\begin{pmatrix}1 & 0\\0 & 0\end{pmatrix} + \frac{1}{2}\begin{pmatrix}0 & 0\\0 & 1\end{pmatrix} - \frac{1}{4}\begin{pmatrix}1\\1\end{pmatrix}(1 \ \ 1)$$
$$= \frac{1}{2}\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} - \frac{1}{4}\begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}$$
$$= \frac{1}{4}\begin{pmatrix}1 & -1\\-1 & 1\end{pmatrix}$$

$$\begin{pmatrix}1\\0\end{pmatrix}(1 \ \ 0) = \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}$$
$$\begin{pmatrix}0\\1\end{pmatrix}(0 \ \ 1) = \begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$$

$$E(Y) = \frac{1}{2}\begin{pmatrix}1\\0\end{pmatrix} + \frac{1}{2}\begin{pmatrix}0\\1\end{pmatrix}$$
$$= \frac{1}{2}\begin{pmatrix}1\\1\end{pmatrix}$$

4a

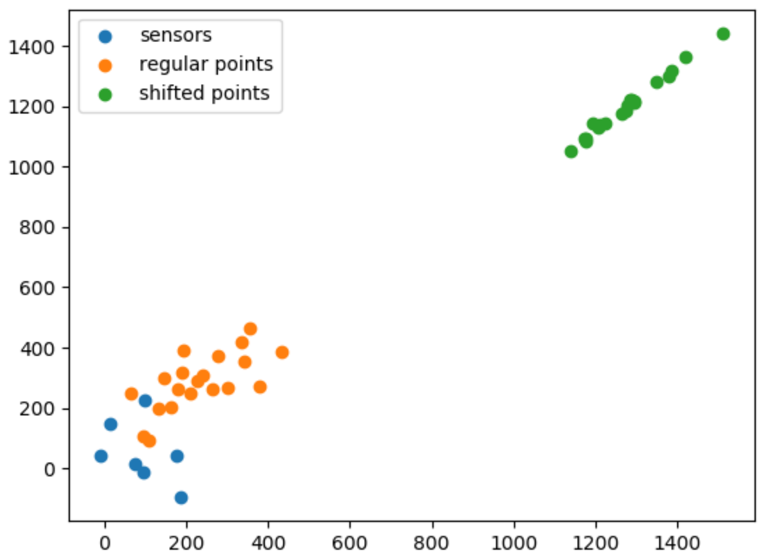
######code#######

```
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))
```

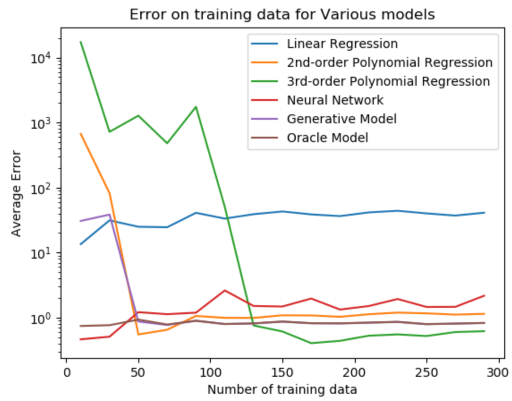
```
    return linear_regression(X_poly, Y, Xs_test_poly, Ys_test)


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())
```

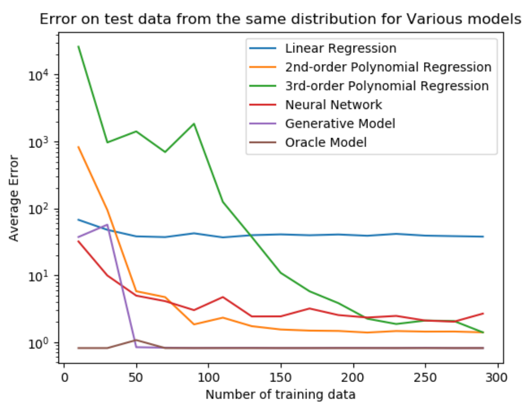
```
##train the model and plot learning curve
##standardize the features first!!!
scaler = StandardScaler()
X_trans = scaler.fit_transform(X)
hist = model.train(X_trans, Y, 2000, GDOptimizer(eta=0.001))
# plt.plot(hist)
# plt.title('Learning curve')
# plt.show()

##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
```

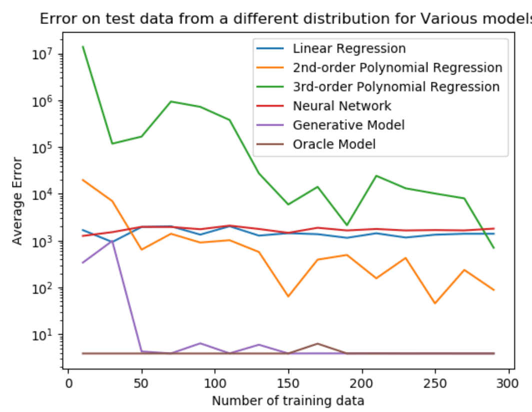
4b


sensors
regular points
shifted points
1400
1200
1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400

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.

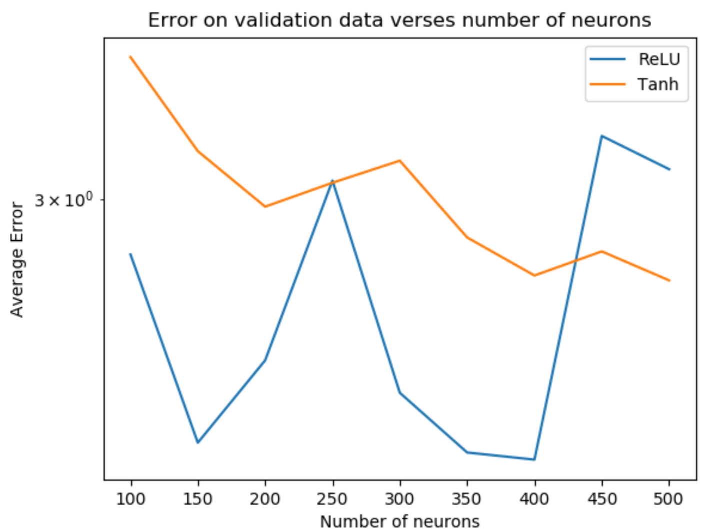
- Generative model works good if we have training set larger than 50, since we can infer the sensor location reasonably good if we have 50 training object locations, and in that case, generative model works as good as oracle model. Runs slow.

- 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
```

4. (e)

input layer $\rightarrow$ 1st hidden $\rightarrow$ 2nd hidden $\overset{..}{\rightarrow}$ kth hidden $\rightarrow$ output layer

$$W^{l\times 7} \quad \underbrace{W^{l\times l} \quad \cdots \quad W^{l\times l}}_{(k-1)} \quad W^{2\times l}$$

$$b^{l\times 1} \quad b^{l\times 1} \quad b^{l\times 1} \quad b^{2\times 1}$$

$$\begin{aligned}
N \equiv \text{total number of parameters} &= 8l + 2(l+1) + (k-1)l(l+1) \\
&= 10l + 2 + (k-1)l^2 + (k-1)l \\
&= (k-1)l^2 + (k+9)l + 2
\end{aligned}$$

$$\Rightarrow (k-1)l^2 + (k+9)l + 2 - N = 0$$

$$l^2 + \frac{k+9}{k-1} l + \frac{2-N}{k-1} = 0$$

$$\left(l + \frac{k+9}{2(k-1)}\right)^2 = \frac{N}{k-1} + \left(\frac{k+9}{2(k-1)}\right)^2$$

$$= \frac{k^2 + 4Nk - 4N}{4(k-1)^2}$$

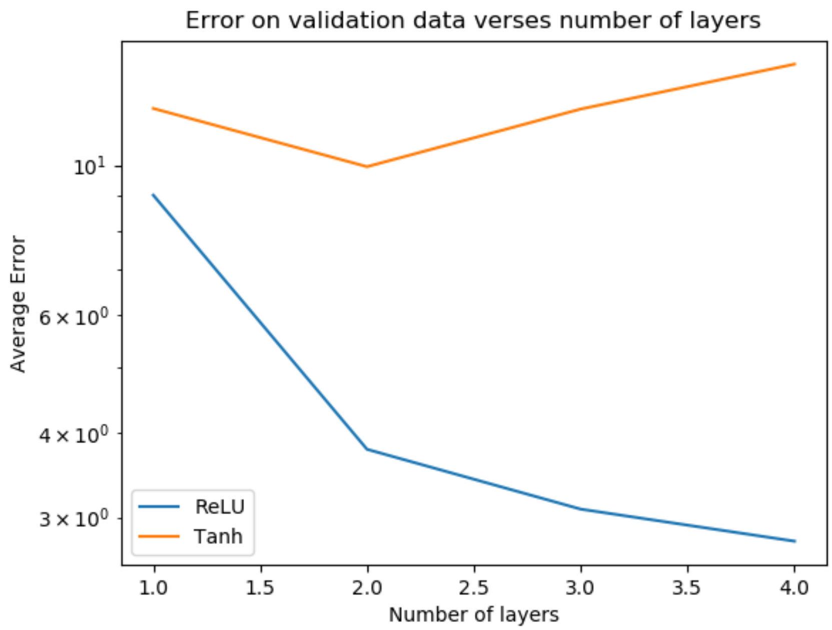
$$l = \sqrt{\frac{k^2+4Nk-4N}{4(k-1)^2}} - \frac{k+9}{2(k-1)}$$

$$\approx \frac{\sqrt{4N(k-1)} - (k+9)}{2(k-1)} \qquad (k \neq 1)$$

when $k=1$, $N = (k+9)l$

$$l = \frac{N}{k+9} = \frac{N}{10}$$

4e

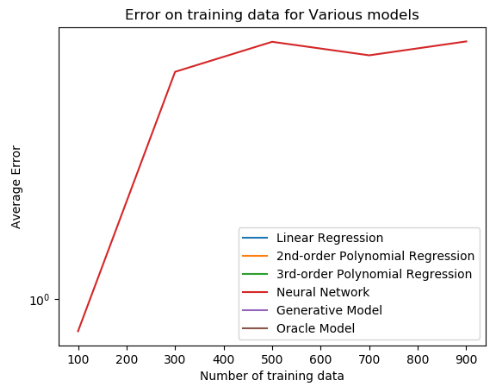


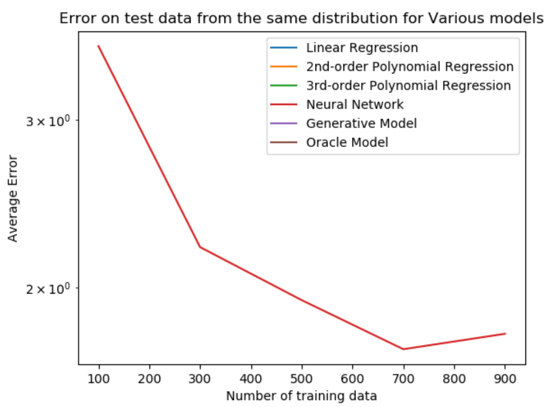
- Again, ReLU works better than tanh. For ReLU, the deeper the better. For tanh, layers of 2 works the best.

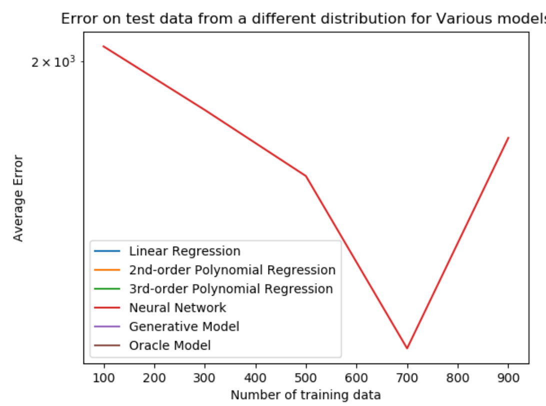
```
#######code#########
def get_num_neurons(N, k):
    """
    given number of parameters N and number of hidden layer k, returns the number of neurons per layer
    """
    if k == 1:
        return int(N/10.0)
    else:
        return int((sqrt(4*N*(k-1)) - (k+9))/2/(k-1))

##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
```

5. (a) $\Omega = \binom{n}{f_0^{(n)}, f_1^{(n)}, \dots f_m^{(n)}}$

$$= \frac{n!}{f_0^{(n)}! \cdots f_m^{(n)}!}$$

$$\approx \frac{\left(\frac{n}{e}\right)^n}{\prod_{j=0}^{m} \left(\frac{f_j^{(n)}}{e}\right)^{f_j^{(n)}}}$$

$$\ln\Omega = n(\ln n - 1) - \sum_{j=0}^{m} f_j^{(n)}(\ln f_j^{(n)} - 1)$$

$$= n(\ln n - 1) - \sum_{j=0}^{m} n\frac{f_j^{(n)}}{n}\left(\ln\left(\frac{f_j^{(n)}}{n}\right) + \ln n - 1\right)$$

$$= n(\ln n - 1) - \sum_{j=0}^{m} \frac{f_j^{(n)}}{n} \cdot n(\ln n - 1) - \sum_{j=0}^{m} n\frac{f_j^{(n)}}{n}\ln\left(\frac{f_j^{(n)}}{n}\right)$$

$$= -\sum_{j=0}^{m} n\frac{f_j^{(n)}}{n}\ln\left(\frac{f_j^{(n)}}{n}\right)$$

$$= \sum_{j=0}^{m} n\frac{f_j^{(n)}}{n}\ln\left(\frac{n}{f_j^{(n)}}\right) = nH\left(\frac{f^{(n)}}{n}\right)$$

$$\Omega = e^{nH(f^{(n)}/n)}$$

(b) $\lim_{n\to\infty} \frac{1}{n} \log P(F^{(n)} = f^{(n)})$

$$= \lim_{n\to\infty} \frac{1}{n} \log \left(e^{nH(f^{(n)}/n)} \prod_{j=0}^{m} P_j^{f_j^{(n)}} \right)$$

$$= \frac{1}{n}\left(nH(f) + \sum_{j=0}^{m} n f_j \ln P_j \right)$$

$$= H(f) + \sum_{j=0}^{m} f_j \ln P_j$$

$$= \sum_{j=0}^{m} f_j \ln \frac{1}{f_j} + \sum_{j=0}^{m} f_j \ln P_j$$

$$= \sum_{j=0}^{m} f_j \ln \frac{P_j}{f_j} = -\sum_{j=0}^{m} f_j \ln \frac{f_j}{P_j} = -KL(f, P)$$

(c) $KL(P, q_\theta) = \sum_{x\in X} \sum_{y\in Y} P(x,y) \ln \frac{P(x,y)}{q_\theta(x,y)}$

$$= \sum_{x\in X} \sum_{y\in Y} P(x,y) \ln \frac{P(x,y)}{q_\theta(y|x)\, q(x)}$$

$$= \sum_{x\in X} \sum_{y\in Y} P(x,y) \ln \frac{P(x,y)}{q(x)} - \sum_{x\in X} \sum_{y\in Y} P(x,y) \ln q_\theta(y|x)$$

$$= C - \sum_{x\in X} \sum_{y\in Y} P(x,y) \ln q_\theta(y|x)$$

(d) $\min_{\theta} KL(p, q_\theta) = \min_{\theta} \left(c - \sum_{x \in X} \sum_{y \in Y} p(x, y) \log q_\theta(y|x) \right)$

$= \min_{\theta} - \sum_{x \in X} \sum_{y \in Y} \frac{1}{n} \mathbb{1}(x = x_i, y = y_i) \log q_\theta(y|x)$

$= \min_{\theta} - \sum_{i} \frac{1}{n} \log q_\theta(y_i | x_i)$