1. $\underbrace{E_x[(\bar{g}(x) - f(x))^2]}_{\text{deterministic noise}}$

$H' \subset H \rightarrow$ how does deterministic noise behave

deterministic noise = discrepancy between target and best hypothesis from hypothesis set H

If we use a subset of H, H', then there are less possibilities for the best hypothesis, limiting it more. This means that it is less likely to be as close to $f(x)$, so the deterministic noise will generally increase.

$\Rightarrow$ [b]

2.

```python
import pandas as pd
import numpy as np

def transform_points(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2,
      x2**2,
      x1 * x2,
      np.abs(x1 - x2),
      np.abs(x1 + x2)
  ))
  return transformed

def linear_regression(data, Y):
  transpose = data.T
  inv = np.linalg.inv(np.dot(transpose, data))
  w = np.dot(np.dot(inv, transpose), Y)
  return w

def find_error(X, Y, w):
  Y_pred = np.sign(np.dot(X, w))
  return np.sum(Y_pred != Y) / len(Y)

train = np.loadtxt('in.dta.txt')
test = np.loadtxt('out.dta.txt')
```

```python
transformed_train = transform_points(train)
transformed_test = transform_points(test)
weights = linear_regression(transformed_train, train[:, 2])
in_error = find_error(transformed_train, train[:, 2], weights)
out_error = find_error(transformed_test, test[:, 2], weights)

print("in sample error: ", in_error)
print("out of sample error: ", out_error)

final = np.array([in_error, out_error])
option1 = np.array([0.03, 0.08])
option2 = np.array([0.03, 0.1])
option3 = np.array([0.04, 0.09])
option4 = np.array([0.04, 0.11])
option5 = np.array([0.05, 0.1])
print(np.linalg.norm(final - option1))
print(np.linalg.norm(final - option2))
print(np.linalg.norm(final - option3))
print(np.linalg.norm(final - option4))
print(np.linalg.norm(final - option5))
```

```
in sample error:  0.02857142857142857
out of sample error:  0.084
0.004247448213519576
0.016063648910709254
0.012907836569230304
0.028400919789646935
0.026742918192848512
```

Based on my code above, the in-sample and out-of-sample error are about 0.03 and 0.08, which is closest to option [a]

3. Using the following solution for linear regression with weight decay: (slide 11, lecture 12)

$$w_{reg} = (X^T X + \lambda I)^{-1} X^T y$$

```
def weight_decay(data, Y, k):
  size = len(data[0])
  transpose = data.T
  inv = np.linalg.inv(np.dot(transpose, data) + (10**k)*np.eye(size))
  w = np.dot(np.dot(inv, transpose), Y)
  return w


train = np.loadtxt('in.dta.txt')
test = np.loadtxt('out.dta.txt')

transformed_train = transform_points(train)
transformed_test = transform_points(test)
weights = weight_decay(transformed_train, train[:, 2], -3)
in_error = find_error(transformed_train, train[:, 2], weights)
out_error = find_error(transformed_test, test[:, 2], weights)

print("in sample error: ", in_error)
print("out of sample error: ", out_error)

final = np.array([in_error, out_error])
option1 = np.array([0.01, 0.02])
option2 = np.array([0.02, 0.04])
option3 = np.array([0.02, 0.06])
option4 = np.array([0.03, 0.08])
option5 = np.array([0.03, 0.1])
print(np.linalg.norm(final - option1))
print(np.linalg.norm(final - option2))
print(np.linalg.norm(final - option3))
print(np.linalg.norm(final - option4))
print(np.linalg.norm(final - option5))
```

```
in sample error:  0.02857142857142857
out of sample error:  0.08
0.06280842267708746
0.04090806018078958
0.021759351731039742
0.0014285714285714284
0.020050955496597432
```

Based on my code above, the in and out of sample errors for $n = -3$ are about 0.03 and 0.08, which is closest to option [d]

4.

```
train = np.loadtxt('in.dta.txt')
test = np.loadtxt('out.dta.txt')

transformed_train = transform_points(train)
transformed_test = transform_points(test)
weights = weight_decay(transformed_train, train[:, 2], 3)
in_error = find_error(transformed_train, train[:, 2], weights)
out_error = find_error(transformed_test, test[:, 2], weights)

print("in sample error: ", in_error)
print("out of sample error: ", out_error)

final = np.array([in_error, out_error])
option1 = np.array([0.2, 0.2])
option2 = np.array([0.2, 0.3])
option3 = np.array([0.3, 0.3])
option4 = np.array([0.3, 0.4])
option5 = np.array([0.4, 0.4])
print(np.linalg.norm(final - option1))
print(np.linalg.norm(final - option2))
print(np.linalg.norm(final - option3))
print(np.linalg.norm(final - option4))
print(np.linalg.norm(final - option5))
```

```
in sample error:  0.37142857142857144
out of sample error:  0.436
0.29169119819089645
0.21882357071860614
0.1536165382253048
0.07998775416478783
0.04596005364022374
```

Based on my code above, the in and out of sample errors for $k=3$ are about 0.37 and 0.44, which is closest to option [e]

5.

```
train = np.loadtxt('in.dta.txt')
test = np.loadtxt('out.dta.txt')

transformed_train = transform_points(train)
transformed_test = transform_points(test)

weights1 = weight_decay(transformed_train, train[:, 2], 2)
out1_error = find_error(transformed_test, test[:, 2], weights1)

weights2 = weight_decay(transformed_train, train[:, 2], 1)
out2_error = find_error(transformed_test, test[:, 2], weights2)

weights3 = weight_decay(transformed_train, train[:, 2], 0)
out3_error = find_error(transformed_test, test[:, 2], weights3)

weights4 = weight_decay(transformed_train, train[:, 2], -1)
out4_error = find_error(transformed_test, test[:, 2], weights4)

weights5 = weight_decay(transformed_train, train[:, 2], -2)
out5_error = find_error(transformed_test, test[:, 2], weights5)

print("k = 2: ", out1_error)
print("k = 1: ", out2_error)
print("k = 0: ", out3_error)
print("k = -1: ", out4_error)
print("k = -2: ", out5_error)
```

```
k = 2:  0.228
k = 1:  0.124
k = 0:  0.092
k = -1:  0.056
k = -2:  0.084
```

Based on my code above, the out of sample error is smallest for k=-1 (0.056) amongst the options => [d]

6.

```
train = np.loadtxt('in.dta.txt')
test = np.loadtxt('out.dta.txt')

transformed_train = transform_points(train)
transformed_test = transform_points(test)

min_error = 10**100
min_k = -1001

#test all integer values of k
#when k is larger than 20, (X^T)X + lambda*Y is not invertible
for i in range(-1000, 20):
  weights = weight_decay(transformed_train, train[:, 2], i)
  error = find_error(transformed_test, test[:, 2], weights)
  if error < min_error:
    min_error = error
    min_k = i

print("min error: ", min_error)
print("min k: ", min_k)
```

```
min error:  0.056
min k:  -1
```

Based on my code above, the minimum out-of-sample error for integer values of k is

0.056 => [b]

7. $\Phi: X \to Z$ (scalar x, vector z)

hypothesis set is linear combination of the polynomials: $(1, L_1(x), L_2(x), \ldots L_Q(x))$

$$H_Q = \left\{ h \mid h(x) = w^T z = \sum_{q=0}^{Q} w_q L_q(x) \right\}$$

$$H(Q, C, Q_0) = \{ h \mid h(x) = w^T z \in H_Q ; w_q = C \text{ for } q \geq Q_0 \}$$

↳ for polynomials of order $\geq Q_0$, coefficient is fixed @ C

[a] $H(10,0,3) \cup H(10,0,4) = H_4$ X → The left-hand side of this equality indicates the union of the set of hypotheses where the coefficient is 0 for degree at least 3, and the same set for degree at least 4. This just results in the set of hypotheses with degree up to 2 (because the polynomials with degrees 3 or 4 have coefficients of 0). This is not equivalent to H_4 because H_4 has all the hypotheses up to degree 4.

[b] $H(10,1,3) \cup H(10,1,4) = H_3$ X → The left-hand side of the equality represents the hypotheses set where the coefficients of polynomials with degree at least 3 is 1. Since $Q = 10$, this means

that all polynomials with degree 3 to 10 will have a coefficient of 1, which is not the same as H_3. H_3 only has polynomials up to degree 3.

[c] $H(10,0,3) \cap H(10,0,4) = H_2$ ✓ → The left hand side represents the hypothesis set up to degree 2 because the polynomials with degrees at least 3 are given a coefficient of 0. Thus, this is the same as H_2.

[d] $H(10,1,3) \cap H(10,1,4) = H_1$ x → Similar to option [b], the left hand side is the interseetion of the hyputheses sets where polynomials of degrees at least 3 and 4 are given a coefficient of 1. This means that the degree of the hypotheses will be much more than 1.

=> [c]

8. $L=2$ (layers) $d^0=5$ $d^1=3$ $d^2=1$

↳input ↳hidden ↳output

$w_{ij}^l x_i^{l-1}$ → forward

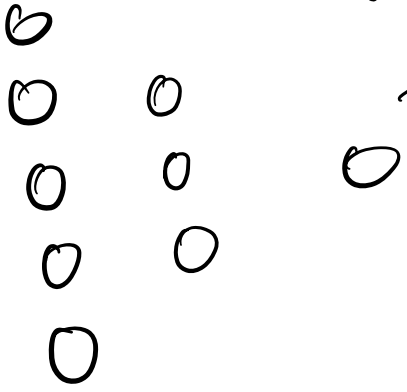
$w_{ij}^l \delta_j^l$

$x_i^{l-1} \delta_j^l$ → backward

only these count as operations

↳updating weights

*total # of operations in 1 backprop iteration w/SGD on 1 point



→these are not including bias nodes

total number of weights:

$6 \cdot 3 + 4 = 22$ weights

↳ outgoing from hidden layer

↳ incoming to hidden layer

↑ outgoing from input layer

for forward pass: $w_{ij}^l x_i^{l-1}$ for each weight ⇒ 22 operations

backward pass: $w_{ij}^l \delta_j^l$ → for all weights except ones from input layer ⇒ 3 operations

updating weights: $x_i^{l-1} \delta_j^l$ → for each weight ⇒ 22 operations

total: $22 + 3 + 22 = 47$ operations

47 is closest to 45 ⇒ [d]

9. 10 input, 36 hidden, 1 output
all layers are fully connected

weights are sums of products of # of nodes every two layers

→ we want to minimize the number of nodes in each layer. Since each hidden layer must have a $x_0^{(1)}$ as well as an active unit, each layer must have 2 nodes (18 hidden layers). There are no input weights for the $x_0^{(1)}$ unit in each layer, but there are output weights. So, the total number of weights is:

$$10 + 18 \cdot (2 \cdot 1) = 46$$

- 10: weights out of input
- 18: 18 hidden layers
- 2: each hidden layer has 2 weights in between bc. 2 outgoing and 1 incoming
- 1: $x_0^{(1)}$

$\Rightarrow$ [a]

10. maximum # of weights for this neural network: we want a large number of units per layer

If all 36 hidden units are in one layer in between the input and output, this will give $10 \cdot 35 + 36 = 386$ weights

- $10 \cdot 35$: 10 input, 35 active units in next layer
- $+36$: outgoing from hidden layer to output unit

However, this is not necessarily the maximum, so I wrote a backtracking algorithm to calculate the optimal distribution of the 36 hidden nodes:

Problem 10:

```
def find_total(cur_total, num_hidden_nodes, prev_layer, curMax, nodes, best):
    if num_hidden_nodes == 0:
      if cur_total + prev_layer > curMax:
            curMax = cur_total + prev_layer
            best.clear()
            best.extend(nodes)
      return curMax

    for i in range(2, num_hidden_nodes + 1):
        num_hidden_nodes -= i
        new_total = cur_total + (prev_layer) * (i - 1)
        nodes.append(i)
        curMax = find_total(new_total, num_hidden_nodes, i, curMax, nodes, best)
        num_hidden_nodes += i
        nodes.pop()

    return curMax
nodes = [10]
best = []
max_total = find_total(0, 36, 10, 0, nodes, best)
print(max_total)
print(best)
```

```
510
[10, 22, 14]
```

With two hidden layers with 22 and 14 units, respectively, the number of weights is:

$10 \cdot 21 + 22 \cdot 13 + 14 = 510 \Rightarrow$ [C e]