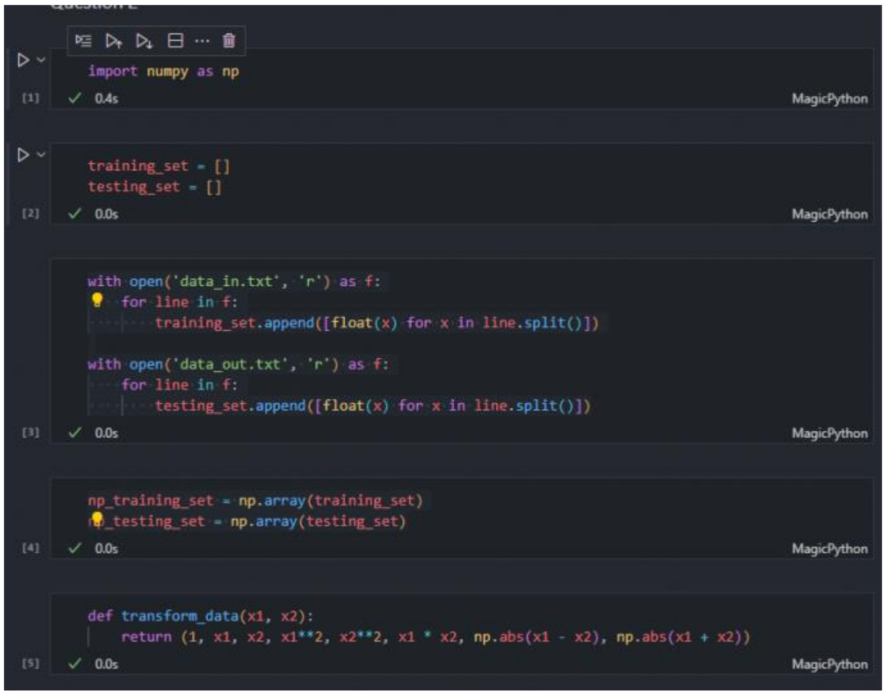
H.W.

Sunday, November 5, 2023 10:07 PM

All the code can be accessed here and executed in google colab or any Jupyter Notebook environment: https://github.com/DragonBoy25830/caltech-cs-156

1) In general, H' will have a lower complexity than H, so H' won't be able to explain more of f and the deterministic noise will increase, so $\boxed{\text{the answer is b.}}$

2)



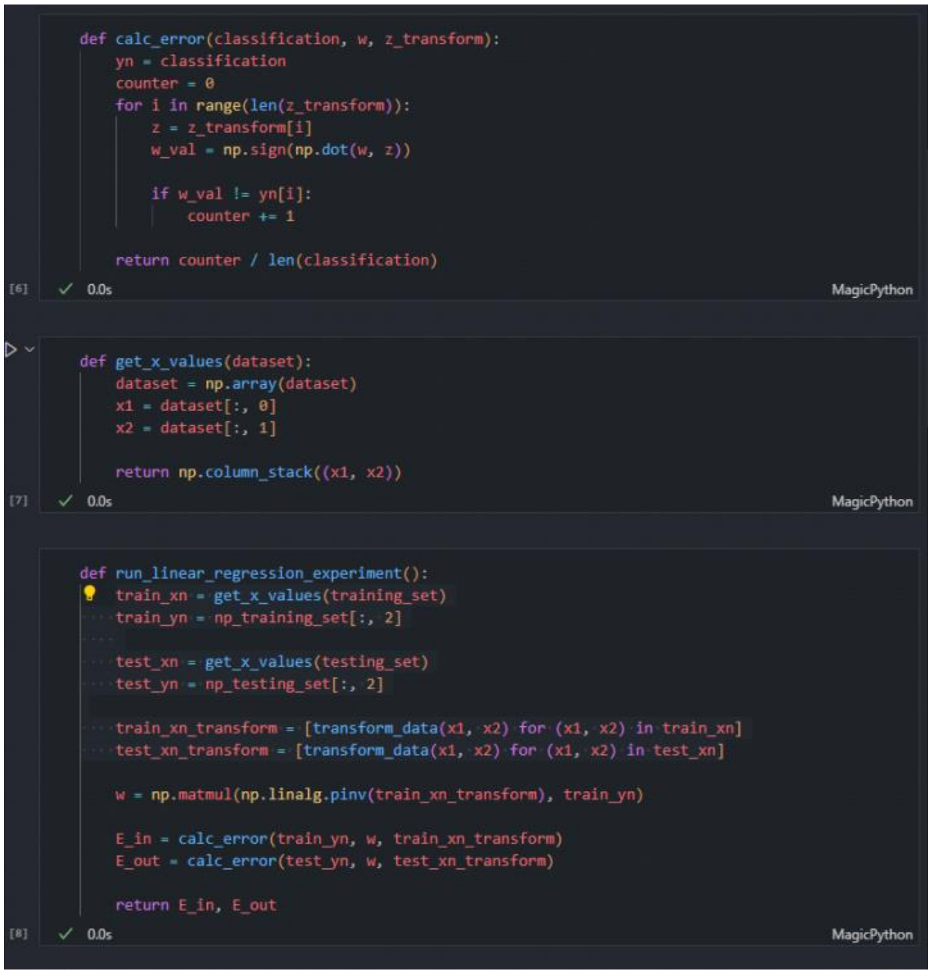
```python
import numpy as np
```

```python
training_set = []
testing_set = []
```

```python
with open('data_in.txt', 'r') as f:
    for line in f:
        training_set.append([float(x) for x in line.split()])

with open('data_out.txt', 'r') as f:
    for line in f:
        testing_set.append([float(x) for x in line.split()])
```

```python
np_training_set = np.array(training_set)
np_testing_set = np.array(testing_set)
```

```python
def transform_data(x1, x2):
    return (1, x1, x2, x1**2, x2**2, x1 * x2, np.abs(x1 - x2), np.abs(x1 + x2))
```



```
def calc_error(classification, w, z_transform):
    yn = classification
    counter = 0
    for i in range(len(z_transform)):
        z = z_transform[i]
        w_val = np.sign(np.dot(w, z))

        if w_val != yn[i]:
            counter += 1

    return counter / len(classification)
```

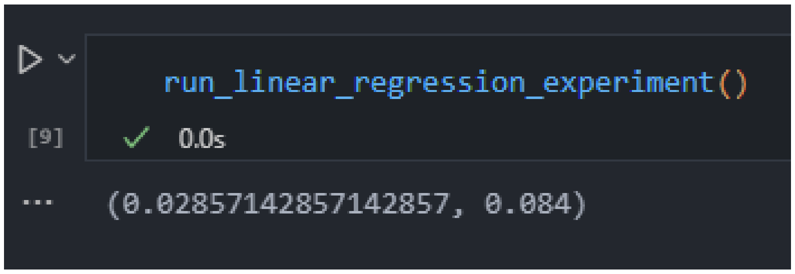
[6] ✓ 0.0s MagicPython

```
def get_x_values(dataset):
    dataset = np.array(dataset)
    x1 = dataset[:, 0]
    x2 = dataset[:, 1]

    return np.column_stack((x1, x2))
```

[7] ✓ 0.0s MagicPython

```
def run_linear_regression_experiment():
    train_xn = get_x_values(training_set)
    train_yn = np_training_set[:, 2]

    test_xn = get_x_values(testing_set)
    test_yn = np_testing_set[:, 2]

    train_xn_transform = [transform_data(x1, x2) for (x1, x2) in train_xn]
    test_xn_transform = [transform_data(x1, x2) for (x1, x2) in test_xn]

    w = np.matmul(np.linalg.pinv(train_xn_transform), train_yn)

    E_in = calc_error(train_yn, w, train_xn_transform)
    E_out = calc_error(test_yn, w, test_xn_transform)

    return E_in, E_out
```

[8] ✓ 0.0s MagicPython



```
run_linear_regression_experiment()
```

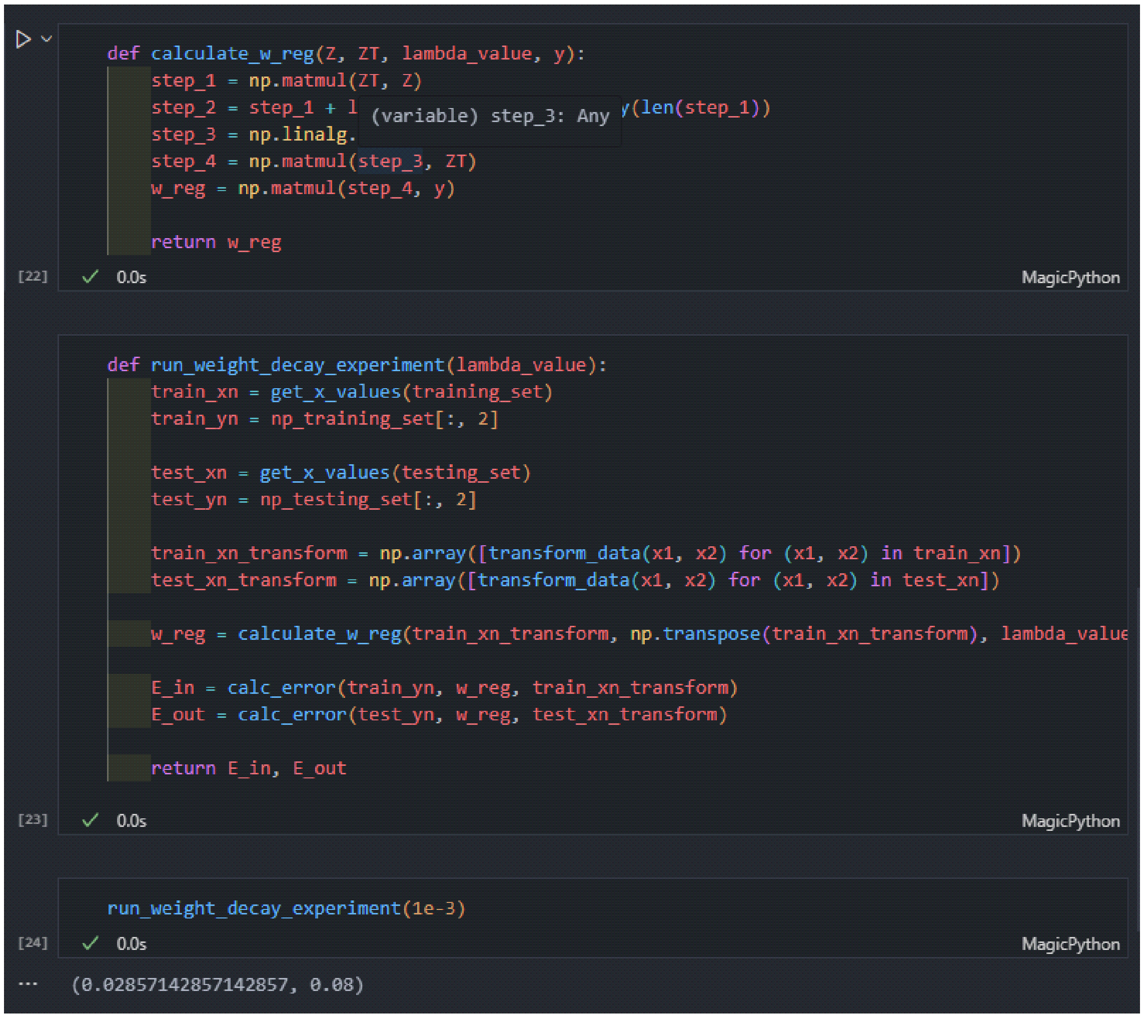
[9] ✓ 0.0s

```
(0.02857142857142857, 0.084)
```

The code above outputs an Ein of 0.0286 and Eout of 0.084 which is closest in Euclidean distance to 0.03 & 0.08, so $\boxed{\text{the answer is a.}}$

3) The functions from before are used for this part as well. All I did was change the code for calculating

as well. All I did was change the code for calculating the weights.



```
def calculate_w_reg(Z, ZT, lambda_value, y):
    step_1 = np.matmul(ZT, Z)
    step_2 = step_1 + l (variable) step_3: Any y(len(step_1))
    step_3 = np.linalg.
    step_4 = np.matmul(step_3, ZT)
    w_reg = np.matmul(step_4, y)

    return w_reg
```
[22] ✓ 0.0s MagicPython

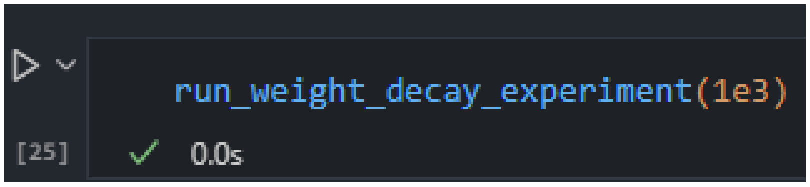
```
def run_weight_decay_experiment(lambda_value):
    train_xn = get_x_values(training_set)
    train_yn = np_training_set[:, 2]

    test_xn = get_x_values(testing_set)
    test_yn = np_testing_set[:, 2]

    train_xn_transform = np.array([transform_data(x1, x2) for (x1, x2) in train_xn])
    test_xn_transform = np.array([transform_data(x1, x2) for (x1, x2) in test_xn])

    w_reg = calculate_w_reg(train_xn_transform, np.transpose(train_xn_transform), lambda_value

    E_in = calc_error(train_yn, w_reg, train_xn_transform)
    E_out = calc_error(test_yn, w_reg, test_xn_transform)

    return E_in, E_out
```
[23] ✓ 0.0s MagicPython

```
run_weight_decay_experiment(1e-3)
```
[24] ✓ 0.0s MagicPython

```
(0.02857142857142857, 0.08)
```

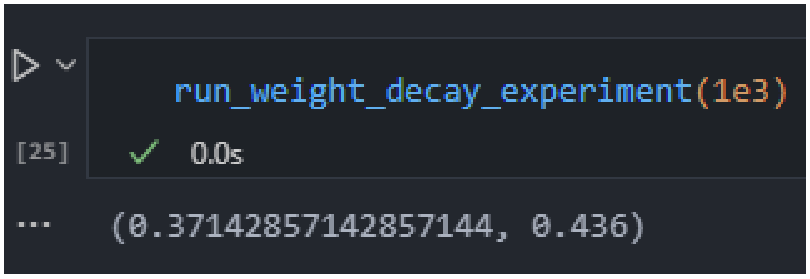
The code above outputs an Ein of 0.0286 and Eout of 0.08 which is closest in Euclidean distance to 0.03 & 0.08, so the answer is d.

4)



```
run_weight_decay_experiment(1e3)
```
[25] ✓ 0.0s

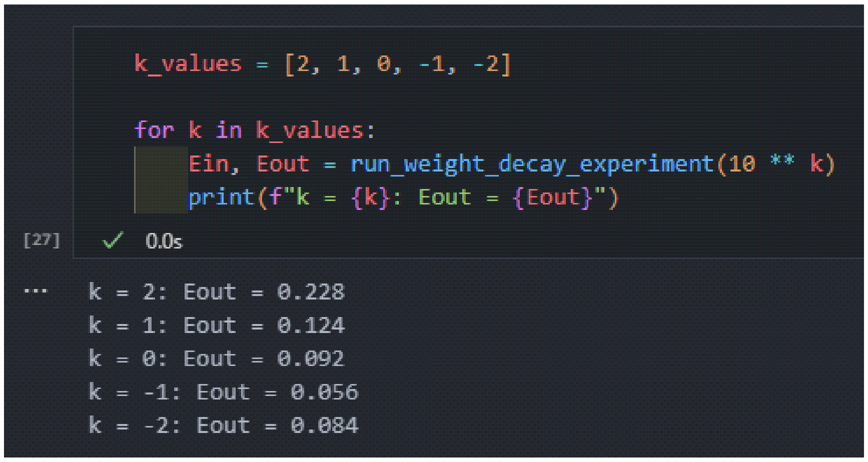
4)



```
run_weight_decay_experiment(1e3)
[25]  ✓ 0.0s
...   (0.37142857142857144, 0.436)
```

The code above outputs an Ein of 0.371 and Eout of 0.436 which is closest in Euclidean distance to 0.4 & 0.4, so $\boxed{\text{the answer is e.}}$

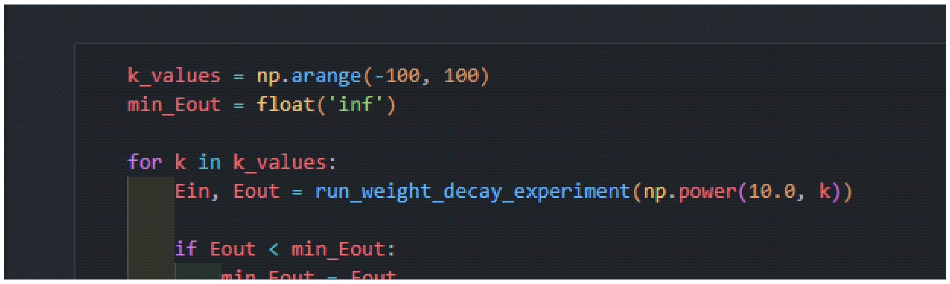
5)



```
k_values = [2, 1, 0, -1, -2]

for k in k_values:
    Ein, Eout = run_weight_decay_experiment(10 ** k)
    print(f"k = {k}: Eout = {Eout}")
[27]  ✓ 0.0s
...
k = 2: Eout = 0.228
k = 1: Eout = 0.124
k = 0: Eout = 0.092
k = -1: Eout = 0.056
k = -2: Eout = 0.084
```

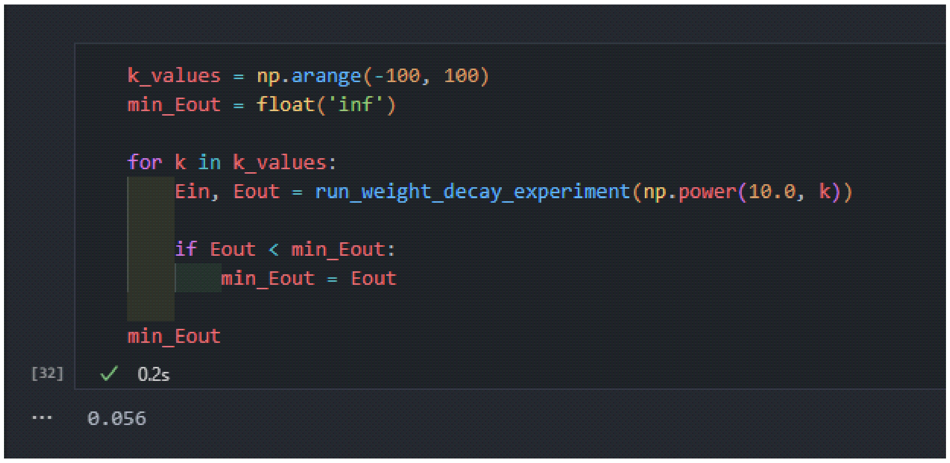
The above code shows that the smallest Eout = 0.056 occurs when K=-1, so $\boxed{\text{the answer is d.}}$

6)



```
k_values = np.arange(-100, 100)
min_Eout = float('inf')

for k in k_values:
    Ein, Eout = run_weight_decay_experiment(np.power(10.0, k))

    if Eout < min_Eout:
        min_Eout = Eout
```

6)



```
k_values = np.arange(-100, 100)
min_Eout = float('inf')

for k in k_values:
    Ein, Eout = run_weight_decay_experiment(np.power(10.0, k))

    if Eout < min_Eout:
        min_Eout = Eout

min_Eout
```

```
[32]  ✓ 0.2s
···   0.056
```

The code above calculates E_{out} for integer values of K from -100 to 100. The minimum E_{out} is equal to 0.056 which is closest to 0.06, so $\boxed{\text{the answer is b}}$

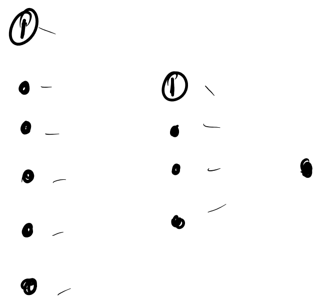
7) If $a<b$, then $\mathcal{H}_a \in \mathcal{H}_b$. Using this fact, we can write that based off the given constraint,

$\mathcal{H}(Q, C, Q_0) = \mathcal{H}_{Q_0-1}$ if $C=0$ since the constraint zeroes out all the terms of L_q after for $Q > Q_0$. As such, $\mathcal{H}(10,0,3) = \mathcal{H}_2$ & $\mathcal{H}(10,0,4) = \mathcal{H}_3$. Since $\mathcal{H}_2 \in \mathcal{H}_3$, we can say that $\mathcal{H}(10,0,3) \cap \mathcal{H}(10,0,4) = \mathcal{H}_2$. As such, $\boxed{\text{the answer is c.}}$

8) $w_{ij}^{(\ell)} x_i^{(\ell-1)}$ = calculate x_j

$w_{ij}^{(\ell)} \delta_j^{(\ell)}$ = calculate δ_j

$x_i^{(\ell-1)} \delta_j^{(\ell)}$ = update weights



We calculate the total # of operations by looking at how many times each type of operation is performed:

1) Calculate x_j

$$\sum_{i=0}^{d^{L-1}} w_{ij}^{(\ell)} x_i^{(\ell-1)} = \underbrace{\sum_{i=0}^{d^0} w_{ij}^{(1)} x_i^{(0)}}_{} + \underbrace{\sum_{i=0}^{d^1} w_{ij}^{(2)} x_i^{(1)}}_{}$$

6 weights for each of the 3 inputs in $d^{(1)}$, so 18 ops.

4 weights for the 1 input in $d^{(2)}$, so 4 ops.

2) Calculate δ_j - Backpropagation

$$\sum_{j=1}^{d^{(\ell)}} w_{ij}^{(\ell)} \delta_j^{(\ell)} = \underbrace{\sum_{j=1}^{d^{(1)}} w_{ij}^{(1)} \delta_j^{(1)}}_{}$$

δ's are only calculated for $d^{(1)}$, so there are 3

we only calculate δ's in the backward step for the four units/nodes in $d^{(1)}$ because $d^{(2)}$ is the base case so there are 3 δ's calculated so 3 ops

$d^{(2)}$ is the base case

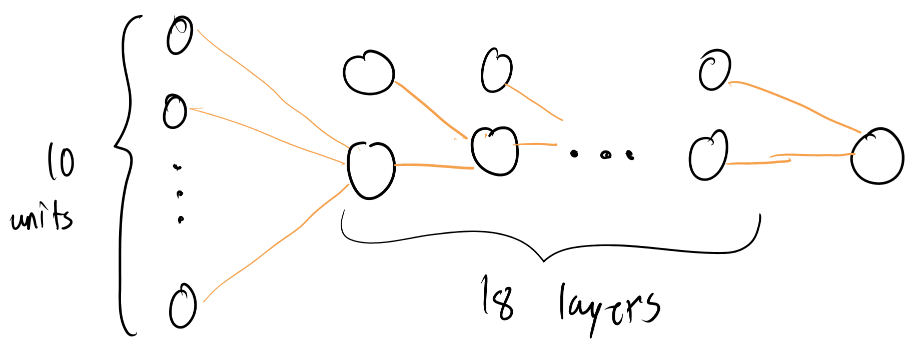
3) Update Weights

every W_{ij} is updated and there are 22 total weights, so there are 22 total ops.

total # of operations = $22+3+22 = 47 \approx 45$

$\boxed{\text{The answer is d.}}$

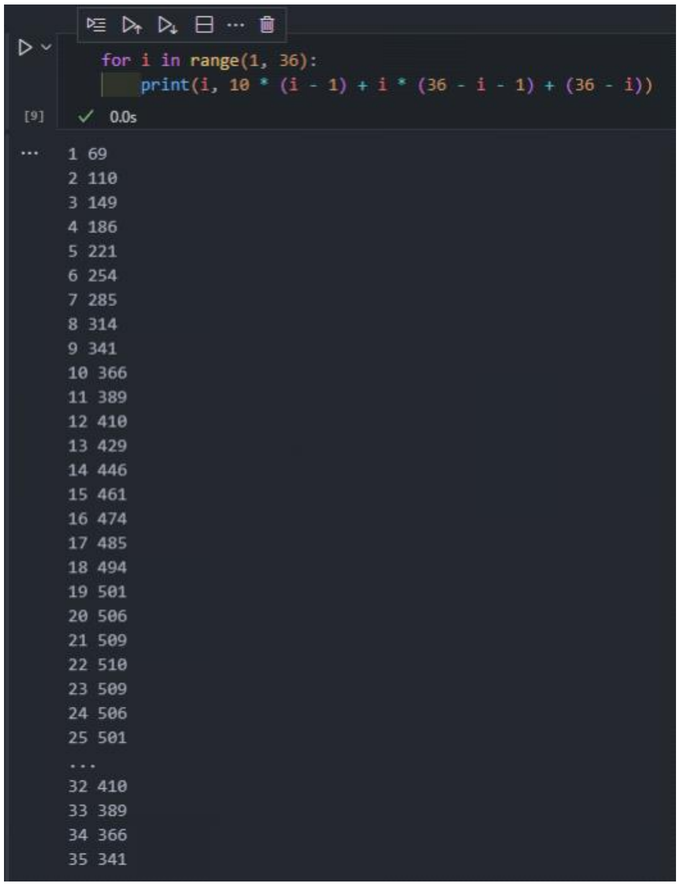
9) The simplest way to reduce the number of weights is to only have two unit in each hidden layer.



We have 10 weights from the input layer to the first hidden layer, 34 weights within the hidden layers and two weights from the last hidden layer to the output layer, so the minimum # of weights is 10+34+2 = 46.

$\boxed{\text{The answer is a.}}$

10)



```
for i in range(1, 36):
    print(i, 10 * (i - 1) + i * (36 - i - 1) + (36 - i))
```

[9] ✓ 0.0s

```
1 69
2 110
3 149
4 186
5 221
6 254
7 285
8 314
9 341
10 366
11 389
12 410
13 429
14 446
15 461
16 474
17 485
18 494
19 501
20 506
21 509
22 510
23 509
24 506
25 501
...
32 410
33 389
34 366
35 341
```

After some trial & error, it seems that two hidden layers maximizes the number of weights in the network. After writing some python code, we see that 22 units in the first layer and 14 in the second layer results in a maximum # of weights of 510. $\boxed{\text{The answer is e.}}$

$\downarrow$

$10(21)+22(13)+14=510$