```python
# Artificial Neural Network  Homework 1, Nicole Adamah
# Calculating the number of linear separable boolean functions
import numpy as np
import itertools
from tqdm import tqdm
trials = 10**4
epochs = 20
eta = 0.05
N = 2 # change n to the desired dimension
counter = 0
variance = 1/N
used_bool = []
x_innt = []
# initializing the input arrays
input = list(itertools.product([-1, 1], repeat=N))
for x in input:
   x_innt.append(x)
x_innt = np.array(x_innt)
def sgn(x):
   return -1 if x < 0 else 1

def learning_rule(x, w, theta):
   b = (np.matmul(x,w)) - theta
   return b

for trial in tqdm(range(trials)):
   bool_f = np.random.choice([-1, 1], size=(2 ** N), p=[1/2,1/2])
   bool_f = bool_f.tolist()
   if bool_f not in used_bool:
       a = np.sqrt(((12) * variance)) / 2
       w = np.random.uniform(-a, a, size=N)
       theta = 0
       for epoch in range(0, epochs):
           total_error = 0
           for mu in range(0,2**N):
               used_bool.append(bool_f)
               pattern = x_innt[mu,:]
               b = learning_rule(pattern, w, theta)
               outputs = sgn(b)
               error = (bool_f[mu]) - (outputs)
               w += eta * error * pattern
               theta -= eta * error
               total_error += abs(error)

           if total_error ==  0:
               counter += 1
               break
print(counter)
```

Homework 1 Boolean functions
Nicole Adamah 21/9/2022

Method:
I approached the task by first defining all variables and making the inputs by looping binary numbers with regards to the dimension. After initializing the variables I made the big loop that iterates n for 10 000 trials. The main tasks inside the loop is to sample a random boolean function and using the update rule to train a perceptron to classify linear separable functions. I then store the number of functions that are linearly separable, in other words where the classification is 100% accurate and the total error for the whole vector is equal to zero. In this task we use the signal function which affects the method in such a way that the input and output vectors consist of +1 and -1. The learning rule with weight updates iterates for 20 epochs and the dimensions that were used was 2,3,4,5.

Results:
Table 1. Results for linearly separable boolean functions for n - dimensions

n	Linearly separable functions	Total amount of possible functions
2	16	2^2^2 = 16
3	104	2^2^3 = 256
4	253	2^2^4 = 65536
5	0	2^2^5 = 4294967296

The result from the problem with linear separable boolean functions is presented in table 1. As one can see the result for two dimensions and three dimension were exact if compared to the true vales(https://en.wikipedia.org/wiki/Linear_separability). But the result for four dimensions and five dimensions were not correct according to the true vales(https://en.wikipedia.org/wiki/Linear_separability). This result is obtained because the total number of possible functions increases very much for five and four dimensions which makes it impossible for the computer program to find all the linearly separable functions with only 10 000 trials. The functions are computed randomly so the chance that the computed functions are the linearly separable functions is small, because of the 10 000 trials. The computer program counts all iterations even if the same functions were produced twice. The conclusion that can be drawn is that the amount of trials need to increase in order for the network to find linearly separable functions for n > 3 but still, it is not sure that the program finds all linearly separable functions.

```python
# Artificial Neural Network  Homework 1, Nicole Adamah
# Library for one-step error probability
import numpy as np

def pattern_Generator(patterns, N):
   return np.random.choice([-1,1], size=(patterns, N))

def sgn(x):
   return -1 if x < 0 else 1

class Hopfield:
   def __init__(self, patterns, N, zerodiagonal:bool):
       self.N = N
       self.zerodiagonal = zerodiagonal
       self.patterns = pattern_Generator(patterns, N)
       self.w = np.zeros((N,N))
       self.hebbs_rule()

   def hebbs_rule(self):

       for x in self.patterns:
           o = x.reshape(self.N,1)
           ot = np.transpose(o)
           self.w += np.matmul(o, ot)# Matrix product of two arrays
           if self.zerodiagonal == True:
               np.fill_diagonal(self.w, 0)
       return self.w / self.N

   def asynchronous(self, state, i):
       self.neuron_weight = self.w[i, :]
       b = np.matmul(self.neuron_weight, state)
       signum = sgn(b)
       if signum == 0:
           signum = 1
       return signum
```

```python
# Artificial Neural Network  Homework 1, Nicole Adamah
# Calculating the one-step error probability
from Hopfield import *
import numpy as np
from tqdm import tqdm
patterns = [12, 24, 48, 70, 100, 120]
trials = 10**5
p_errors = []
N = 120

for p in patterns:
   error = 0
   for i in tqdm(range(trials)):

       H = Hopfield(p,N, zerodiagonal=False) # Depending on task,
change the Zerodiagonal to true or false
       # Pick a neuron and pattern to feed from random integers of N
and p
       random_neuron = np.random.randint(N)
       random_pattern = np.random.randint(p)
       picked_pattern = H.patterns[random_pattern]

       # asynchronous update rule
       S1 = H.asynchronous(picked_pattern, random_neuron)
#Distorted pattern
       S0 = picked_pattern[random_neuron] # stored pattern

       if (S1 != S0): # If the distorted pattern doesn't converge to
the stored, error counts +1
           error+=1

   One_Step_error = error/trials
   p_errors.append(One_Step_error)
   print(f"The One-step error probability is {One_Step_error: .04f}
for p = {p} with trials = {trials}")
```

```python
# Artificial Neural Network  Homework 1, Nicole Adamah
# Library for Recognizing digits
import numpy as np
def sgn(x):
   return -1 if x < 0 else 1

class Hopfield:
   def __init__(self, patterns, N, zerodiagonal:bool):
       self.N = N
       self.zerodiagonal = zerodiagonal
       self.patterns = patterns
       self.w = np.zeros((N,N))
       self.hebbs_rule()

   def hebbs_rule(self):
       for x in self.patterns:
           self.w += np.matmul(x, x.T)  # Matrix product of two
arrays

       if self.zerodiagonal == True:
               np.fill_diagonal(self.w, 0)
       return self.w / self.N

   def asynchronous(self, state, i):
       update_state = np.copy(state)
       update_state[i,:] = sgn(np.matmul(self.w[i,:], state))

       return update_state

   def update_state(self, state):
       S0 = state
       S1 = state
       while True:
           for i in range(self.N):
               S1 = self.asynchronous(S1,i)
           if np.all(S0 == S1):
               return S1
           S0 = S1
```

```
# Artificial Neural Network  Homework 1, Nicole Adamah
# Recognizing digits with Hopfields network
import numpy as np
from HopfieldDigit import*
import matplotlib.pyplot as plt

x1=np.array([ [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],[ -1, -1,
-1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
              [ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1,
-1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
              [ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1,
-1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
              [ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1,
-1, 1, 1, 1, -1],[ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
              [ -1, 1, 1, 1, -1, -1, 1, 1, 1, -1],[ -1, -1, 1, 1, 1,
1, 1, 1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1] ])

x2=np.array([ [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1,
1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1,
1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1,
1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1,
1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1,
1, 1, 1, -1, -1, -1],[ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
              [ -1, -1, -1, 1, 1, 1, 1, -1, -1, -1] ])

x3=np.array([ [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1, 1,
1, 1, -1, -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
              [ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],[ -1, -1, -1,
-1, -1, 1, 1, 1, -1, -1],[ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
              [ -1, -1, -1, -1, -1, 1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1,
1, 1, 1, -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],
              [ 1, 1, 1, -1, -1, -1, -1, -1, -1, -1],[ 1, 1, 1, -1,
-1, -1, -1, -1, -1, -1],[ 1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
              [ 1, 1, 1, -1, -1, -1, -1, -1, -1, -1],[ 1, 1, 1, -1,
-1, -1, -1, -1, -1, -1],[ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1],
              [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1] ])

x4=np.array([ [ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],[ -1, -1, 1, 1, 1,
1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1,
-1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, 1, 1,
1, 1, 1, 1, -1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
```

```
              [ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1,
-1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, 1, 1, 1, -1],[ -1, -1, -1,
-1, -1, -1, 1, 1, 1, -1],[ -1, -1, 1, 1, 1, 1, 1, 1, 1, -1],
              [ -1, -1, 1, 1, 1, 1, 1, 1, -1, -1] ])

x5=np.array([ [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1,
-1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
              [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1,
-1, -1, -1, 1, 1, -1],[ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
              [ -1, 1, 1, -1, -1, -1, -1, 1, 1, -1],[ -1, 1, 1, 1, 1,
1, 1, 1, 1, -1],[ -1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1],[ -1, -1, -1,
-1, -1, -1, -1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1],[ -1, -1, -1,
-1, -1, -1, -1, 1, 1, -1],[ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
              [ -1, -1, -1, -1, -1, -1, -1, 1, 1, -1] ])

N = len(x1.flatten())
chosen_pattern = 3 #change number for corresponding input pattern to
feed

def feed_pattern(chosen_pattern):

   if chosen_pattern == 1:
       p = np.array([[1, 1, -1, -1, -1, -1, -1, -1, 1, 1], [-1, -1,
1, 1, 1, 1, 1, 1, 1, -1],  [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
      [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1,
-1, 1, 1, 1, -1],[-1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
      [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, 1, 1, 1, 1, 1,
1, -1, -1], [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
      [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1,
-1, 1, 1, 1, -1],[-1, -1, -1, -1, -1, -1, 1, 1, 1, -1],
      [-1, -1, -1, -1, -1, -1, 1, 1, 1, -1], [-1, -1, -1, -1, -1,
-1, 1, 1, 1, -1], [-1, -1, 1, 1, 1, 1, 1, 1, 1, -1],
      [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1]]).flatten()



   if chosen_pattern == 2:
       p = np.array([[1, 1, -1, -1, 1, -1, 1, 1, -1, -1], [1, 1, -1,
-1, 1, -1, 1, 1, -1, -1], [1, 1, -1, -1, 1, -1, 1, 1, -1, -1],
                             [1, 1, -1, -1, 1, -1, 1, 1, -1, -1], [1,
1, -1, -1, 1, -1, 1, 1, -1, -1], [1, 1, -1, -1, 1, -1, 1, 1, -1,
-1],
                             [1, 1, -1, -1, 1, -1, 1, 1, -1, -1], [1,
1, -1, 1, -1, 1, -1, 1, -1, -1], [1, 1, -1, 1, -1, 1, -1, 1, -1,
-1],
```

```
                              [1, -1, 1, -1, 1, -1, 1, 1, -1, -1], [1,
-1, 1, -1, 1, -1, 1, 1, -1, -1], [1, -1, 1, -1, 1, -1, 1, 1, -1,
-1],
                              [1, -1, 1, -1, 1, -1, 1, 1, -1, -1], [1,
-1, 1, -1, 1, -1, 1, 1, -1, -1], [1, -1, 1, -1, 1, -1, 1, 1, -1,
-1],
                              [1, -1, 1, -1, 1, -1, 1, 1, -1,
-1]]).flatten()
   if chosen_pattern == 3:
       p = np.array([[1, 1, 1, -1, -1, -1, -1, 1, 1, 1], [1, 1, 1,
-1, -1, -1, -1, 1, 1, 1], [1, 1, 1, -1, -1, -1, -1, 1, 1, 1],
                                     [1, 1, 1, -1, -1, -1, -1, 1, 1,
1], [1, 1, 1, -1, -1, -1, -1, 1, 1, 1], [1, 1, 1, -1, -1, -1, -1, 1,
1, 1],
                                     [1, 1, 1, -1, -1, -1, -1, 1, 1,
1], [1, 1, 1, -1, -1, -1, -1, 1, 1, 1], [1, 1, 1, -1, -1, 1, 1, -1,
-1, -1],
                                     [-1, -1, -1, 1, 1, 1, 1, -1, -1,
-1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1], [-1, -1, -1, 1, 1, 1, 1,
-1, -1, -1],
                                     [-1, -1, -1, 1, 1, 1, 1, -1, -1,
-1], [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1], [-1, -1, -1, 1, 1, 1, 1,
-1, -1, -1],
                                     [-1, -1, -1, 1, 1, 1, 1, -1, -1,
-1]]).flatten()

   return np.reshape(p, (-1, 1))

def pattern_reconizer(p, stored):
   for row, column in enumerate(stored):
       if np.all(p == column):
           return row + 1
       if np.all(p == -column):
           return -(row + 1)
   return len(stored) + 1


def typewriter_shape(p):
   p1 = np.reshape(p, (16, 10))
   return p1

def plot(initial_p, converged_p):
   plt.subplot(121)
   plt.title('initial pattern')
   plt.imshow(initial_p, cmap="gray")
   plt.subplot(122)
   plt.title('converged pattern')
   plt.imshow(converged_p, cmap="gray")
```

```
    plt.show()

if __name__ == "__main__":
    #store all patterns together
    stored = [np.reshape(x1, (-1, 1)), np.reshape(x2, (-1, 1)),
np.reshape(x3, (-1, 1)), np.reshape(x4, (-1, 1)),
              np.reshape(x5, (-1, 1))]
    # initialize the weight matrix
    H = Hopfield(stored, N, zerodiagonal = True)
    # Get the input pattern
    input = feed_pattern(chosen_pattern)
    # update with asynchronous update until converged
    steady = H.update_state(input)
    # change to typewriter scheme
    new_state = typewriter_shape(steady)
    initial_pattern = typewriter_shape(input)

    recognized_digit = pattern_reconizer(steady, stored)
    print(new_state)
    print(f"\nThe pattern converged to the state {recognized_digit}")
    plot(initial_pattern, new_state)
```