Application of correlation learning algorithm

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Using and importing the required libraries to solve the question and implement it:

numpy:
To use its properties and functions in working with matrices

Matplotlib:
To show and draw the graph of the inputs and draw the obtained model

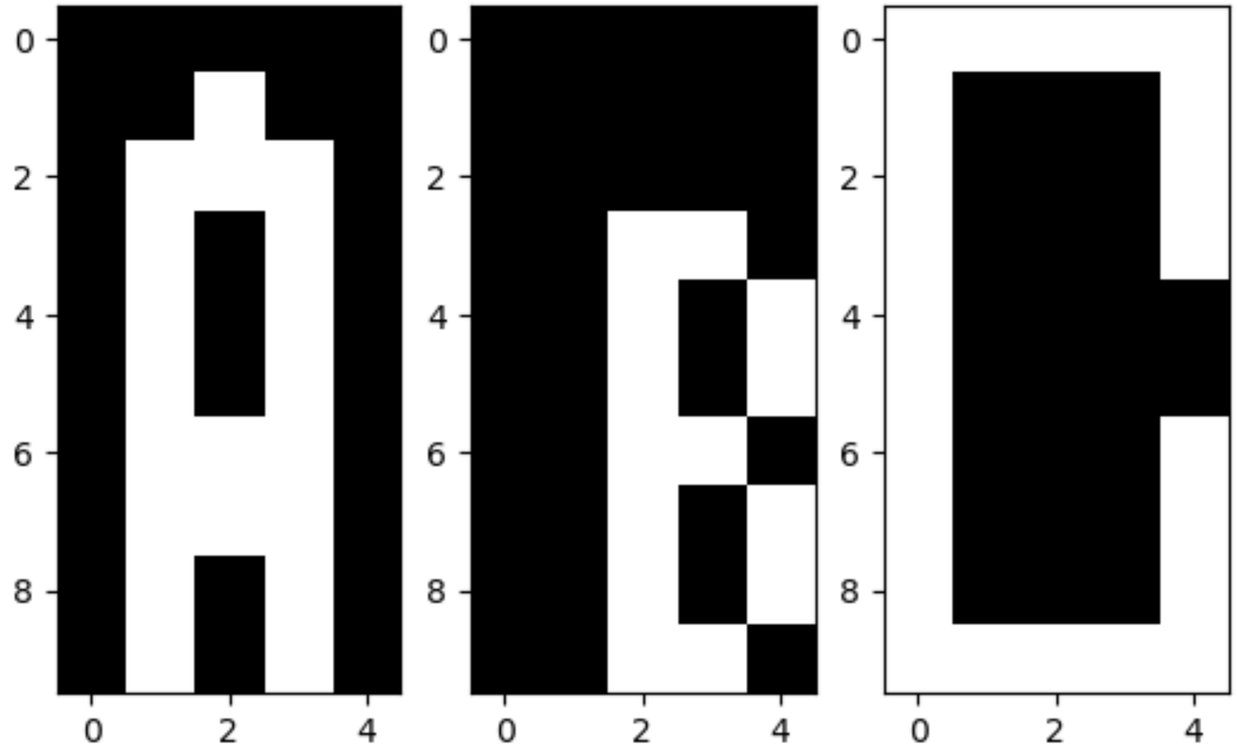
In [2]:
```
A = np.array([

   [0,0,0,0,0],
   [0,0,1,0,0],
   [0,1,1,1,0],
   [0,1,0,1,0],
   [0,1,0,1,0],
   [0,1,0,1,0],
   [0,1,1,1,0],
   [0,1,1,1,0],
   [0,1,0,1,0],
   [0,1,0,1,0]
   ])

B = np.array([

   [0,0,0,0,0],
   [0,0,0,0,0],
   [0,0,0,0,0],
   [0,0,1,1,0],
   [0,0,1,0,1],
   [0,0,1,0,1],
   [0,0,1,1,0],
   [0,0,1,0,1],
   [0,0,1,0,1],
   [0,0,1,1,0]
   ])

C=np.array([

   [1,1,1,1,1],
   [1,0,0,0,1],
   [1,0,0,0,1],
   [1,0,0,0,1],
   [1,0,0,0,0],
   [1,0,0,0,0],
   [1,0,0,0,1],
   [1,0,0,0,1],
   [1,0,0,0,1],
   [1,1,1,1,1]
   ])

C_noise=np.array([

   [0,0,1,0,1],
   [1,1,0,1,1],
   [1,0,0,1,1],
   [1,0,1,0,1],
   [0,0,0,0,0],
   [1,0,0,0,0],
   [0,0,0,0,1],
   [1,0,1,0,1],
   [1,1,0,1,0],
   [1,0,1,0,0]
   ])
```

In this section, we will define our inputs, which are alphabet letters A-B-C, as 10 x 5 arrays.

In this law of learning the ideal condition is that:
1- As much as possible, our entries should be orthogonal and have little correlation with each other.
2- Our inputs should be normal
We try to consider matrices with low correlation conditions as much as possible to get close to ideal conditions.
You will also see in the next slides that we have normalized the matrices.

```
figure, axis= plt.subplots(1,3)
axis[0].imshow(A, cmap="gray")
axis[1].imshow(B, cmap="gray")
axis[2].imshow(C, cmap="gray")
```

Out[2]: <matplotlib.image.AxesImage at 0x7fa77216ea00>



We draw our input matrices which are alphabet letters A-B-C.

In [ ]:
```
a = A.reshape((1,50))
b = B.reshape((1,50))
c = C.reshape((1,50))
C_noise = C_noise.reshape((1,50))


a = a / np.linalg.norm(a)
b = b / np.linalg.norm(b)
c = c / np.linalg.norm(c)
C_noise = C_noise / np.linalg.norm(C_noise)
```

In [3]:
```
def set_weights():
    W = np.zeros((50,50))
    return W
```

In [4]:
```
def calculate_output(inp, W):
    y = np.dot(inp,W.T)
    return y
```

In [5]:
```
def update_wieghts(W,inp,y):
    delta_W = np.dot(inp.T,y)
    W = W + delta_W
    return W
```

We prepare input matrices for processing and training:
1- We convert them into a 1 x 50 array.
2- As we mentioned in the previous slide, we normalize them.
We define the weights matrix with this function. This matrix is a 50 x 50 matrix because we have 50 neurons to produce 0 or 1 output. Also, each neuron has 50 input dimensions. Because all dimensions are connected to all neurons.
This function calculates the output of the neuron: by multiplying the input in the weights matrix
This function trains and updates the weights matrix:
This learning rule updates the weight matrix by multiplying the desired output by the inputs.

In [6]:
```
weights = set_weights()
weights = update_wieghts(weights,a,a)
weights = update_wieghts(weights,b,b)
weights = update_wieghts(weights,c,c)
```

In [7]:
```
w_max = weights.max()
w_min = weights.min()

weights = ((weights)-(w_min))/(w_max - w_min)
```

In [8]:
```
out_a = calculate_output(a,weights).reshape((10,5))
out_b = calculate_output(b,weights).reshape((10,5))
out_c = calculate_output(c,weights).reshape((10,5))
out_C_noise = calculate_output(C_noise,weights).reshape((10,5))
```

We define the weights matrix.
This function trains and updates the weights matrix:
So in this (self-associating) question, the desired output, and the inputs are the same.
We normalize the matrix of weights.
After the training phase, we test the outputs of this neural network with the same input data.
We also test a noise matrix of the letter C in it.

In [9]:
```
def square_error_loss(y_true, y_pred):
    return (y_true - y_pred) ** 2
```

In [10]:
```
def mean_square_error(y_true, y_pred):
    return np.sum(square_error_loss(y_true, y_pred)) / (y_true.shape[0]*y_true.shape[1])
```

In [11]:
```
print('MSE for A:'+str(mean_square_error(A,out_a/out_a.max())))
print('MSE for B:'+str(mean_square_error(B,out_b/out_b.max())))
print('MSE for C:'+str(mean_square_error(C,out_c/out_c.max())))
print('MSE for C with noise:'+str(mean_square_error(C,out_C_noise/out_C_noise.max())))
```

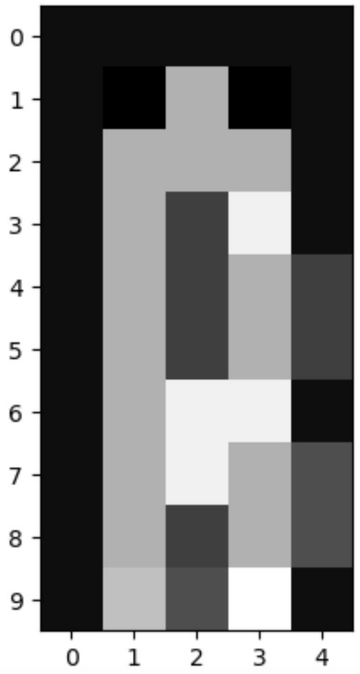
```
MSE for A:0.04193702615941536
MSE for B:0.02906574394463667
MSE for C:0.04336273780422999
MSE for C with noise:0.10825363674899967
```

These two functions are written to calculate the error resulting from the associated output of a letter of the alphabet compared to the matrix itself. The inputs are normalized and we also normalize the outputs and normalize them between 0 and 1.
The MSE function works like this: it calculates the average errors of each array: the average error of each array to the power of two

We can see the output of this self-associating neural network in the next slides.

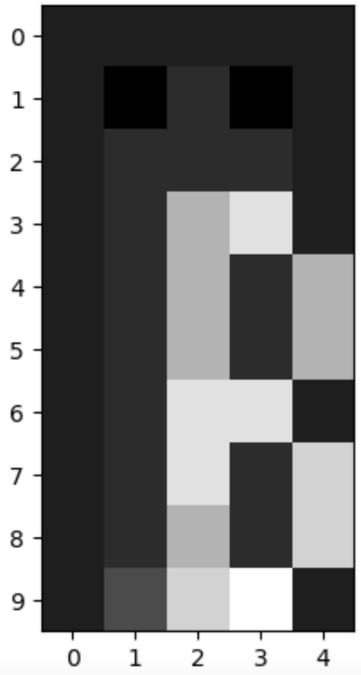
In [12]:
```
plt.imshow(out_a,
           interpolation='nearest',
           cmap = 'gray'
          )
plt.yticks(range(10))
plt.xticks(range(5))
```

Out[12]:
```
([<matplotlib.axis.XTick at 0x7fa77262dd90>,
  <matplotlib.axis.XTick at 0x7fa77262dd60>,
  <matplotlib.axis.XTick at 0x7fa77262dc40>,
  <matplotlib.axis.XTick at 0x7fa772670640>,
  <matplotlib.axis.XTick at 0x7fa772670d90>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



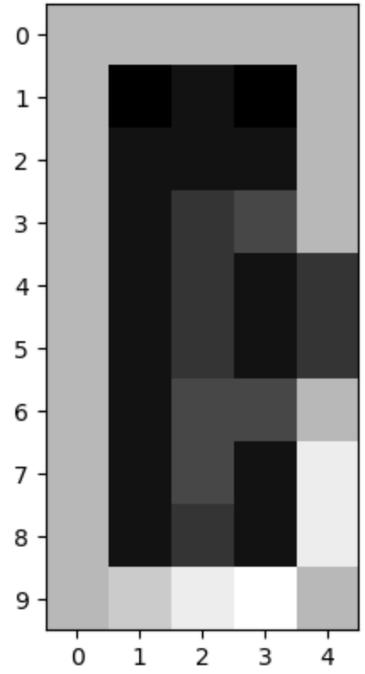
In [13]:
```
plt.imshow(out_b,
           interpolation='nearest',
           cmap = 'gray',
          )
plt.yticks(range(10))
plt.xticks(range(5))
```

Out[13]:
```
([<matplotlib.axis.XTick at 0x7fa77269fa00>,
  <matplotlib.axis.XTick at 0x7fa77269f6d0>,
  <matplotlib.axis.XTick at 0x7fa77269f9a0>,
  <matplotlib.axis.XTick at 0x7fa7727af6d0>,
  <matplotlib.axis.XTick at 0x7fa7727afe20>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



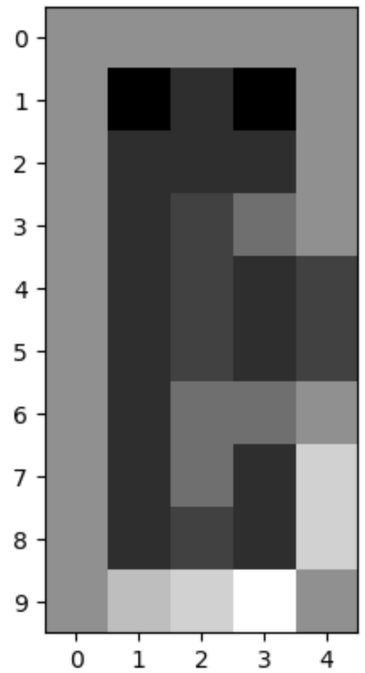
In [14]:
```
plt.imshow(out_c,
           interpolation='nearest',
           cmap = 'gray',
          )
plt.yticks(range(10))
plt.xticks(range(5))
```

Out[14]:
```
([<matplotlib.axis.XTick at 0x7fa7727cdbe0>,
  <matplotlib.axis.XTick at 0x7fa7727cdbb0>,
  <matplotlib.axis.XTick at 0x7fa7727a2fa0>,
  <matplotlib.axis.XTick at 0x7fa7730bf940>,
  <matplotlib.axis.XTick at 0x7fa7730c20a0>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



In [15]:
```
plt.imshow(out_C_noise,
           interpolation='nearest',
           cmap = 'gray',
          )
plt.yticks(range(10))
plt.xticks(range(5))
```

Out[15]:
```
([<matplotlib.axis.XTick at 0x7fa7731654f0>,
  <matplotlib.axis.XTick at 0x7fa773165970>,
  <matplotlib.axis.XTick at 0x7fa7730b5490>,
  <matplotlib.axis.XTick at 0x7fa7731ac580>,
  <matplotlib.axis.XTick at 0x7fa7731accd0>],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



You can see the output of this self-associating neural network for noisy data with 30% noise for the letter C.