In [13]:

```python
import numpy as np # numbers, matrices, numerical computing.
import math # mathematical functions.
import torch # Deep learning and neural networks.
from typing import List # provides annotations that hint the expected types of values in the code.
```

In [58]:

```python
# Step 1: We create random number generator object with a fixed seed.
rand = np.random.default_rng(seed=1)

# Step 2: Now we generate a random permutations of length n=10 using rand.
perm_1 = rand.permutation(10)
perm_2 = rand.permutation(10)
perm_3 = rand.permutation(10)
print('Output 1:', perm_1, perm_2, perm_3)
print('\n')

# Step 3: Reproduce the permutations using rand.
rand = np.random.default_rng(seed=1)
perm_4 = rand.permutation(10)
perm_5 = rand.permutation(10)
perm_6 = rand.permutation(10)
print('Output 2:', perm_4, perm_5, perm_6)
```

```
Output 1: [8 4 7 0 1 2 5 9 6 3] [0 1 8 6 5 7 9 2 3 4] [3 1 5 6 9 0 7 2 8 4]


Output 2: [8 4 7 0 1 2 5 9 6 3] [0 1 8 6 5 7 9 2 3 4] [3 1 5 6 9 0 7 2 8 4]
```

In [61]:

```python
# Step 1:
# Define a function which returns a random permutation and computes its order.
def order_of_perm(perm : List[int])-> int:

    # Step 2:
    # Define the length of permutation 'n'.
    n = len(perm)

    # Step 3:
    # Initialise 'seen' list with 'n' False entries.
    seen = [False] * n
    """
    - Initialise a list called seen, which asigns a value 'True' or 'False' to each element in the permutation.
    - This boolean value is based on the fact whether the element has has seen in a cycle.
    - Initially, we assign a 'False' value to each element in the permutation.
    """

    # Step 3:
    # Initialise a list to store the lengths of disjoint cycles of a permutation.
    disjoint_lengths = []
    """
    - The least common multiple of these lengths at the end gives the order of the permutation.
    """

    # Step 4:
    # Iterate over every element in our permutation one by one.
    for i in range(n):
        # Enter the loop if the element 'i' has not been seen.
        if seen[i] == False:
            j = i
            # Initialise a list that stores elements of this cycle and increments the cycle_length.
            cycle = []
            cycle_length = 0
            # Go through an entire disjoint one cycle
            while seen[j] == False:
                seen[j] = True
                j = perm[j]
                cycle.append(j)
                cycle_length = cycle_length + 1
                """
                - We iterate over unseen entries $j$ : $j=j$, $j= perm[j]$, $j=perm[perm[j]]$, ...
                - For each such entry, we  upadte our list cycle and also update the count of this 'cycle_length'.
                - This continues until we land on some entry that is seen previously.
                - This completes one complete disjoint cycle.
                """
            # Update the list of disjoint_lengths with the cycle_length just found
            disjoint_lengths.append(cycle_length)

    # Step 5:
    # Define order of permutation to be the lcm of lengths stored in disjoint_lengths.
    order = math.lcm(*[i for i in disjoint_lengths])

    # Step 6:
    # Return the permutation and its order.
    return order
```

Testing:

In [62]:

```python
# Let's see if how our function is performing for 10 random permutations of length 10.
rand = np.random.default_rng(seed=1)
"""
- We need a random generator object 'gen' to generate the permutation of a specific length 'n'.
"""
for _ in range(10):
    perm = rand.permutation(10)
    print(perm,order_of_perm(perm))
print('\n')

# We can see that if we restart our seed, our random permutations again start being generated in the same order.
rand = np.random.default_rng(seed=1)
for _ in range(5):
    perm = rand.permutation(10)
    print(perm,order_of_perm(perm))
```

```
[8 4 7 0 1 2 5 9 6 3] 8
[0 1 8 6 5 7 9 2 3 4] 8
[3 1 5 6 9 0 7 2 8 4] 6
[0 7 9 3 2 6 4 8 5 1] 8
[3 1 9 2 8 4 0 5 7 6] 20
[7 5 8 4 0 6 1 2 3 9] 6
[4 0 9 5 8 7 3 2 6 1] 10
[6 3 2 0 5 4 1 8 7 9] 4
[5 0 7 6 3 1 4 8 9 2] 12
[6 2 3 1 9 4 8 5 7 0] 21


[8 4 7 0 1 2 5 9 6 3] 8
[0 1 8 6 5 7 9 2 3 4] 8
[3 1 5 6 9 0 7 2 8 4] 6
[0 7 9 3 2 6 4 8 5 1] 8
[3 1 9 2 8 4 0 5 7 6] 20
```

Indentifying orders as classes to describe the classification problem:

We will now write a short snippet of code to generate 10 permutation with orders between 1 and 10. This means we have 10 classes depending on the order and we want to classify given set of permutations within these 10 equivalence classes.

In [17]:

```python
# Generate random permutations of length 10 with order <= 10
rand = np.random.default_rng(seed=1)
count = 0
while count < 10:
    perm = rand.permutation(10)
    order = order_of_perm(perm)
    if order <= 10:
        print(f"Permutation: {perm}, Order: {order}")
        count += 1
```

```
Permutation: [8 4 7 0 1 2 5 9 6 3], Order: 8
Permutation: [0 1 8 6 5 7 9 2 3 4], Order: 8
Permutation: [3 1 5 6 9 0 7 2 8 4], Order: 6
Permutation: [0 7 9 3 2 6 4 8 5 1], Order: 8
Permutation: [7 5 8 4 0 6 1 2 3 9], Order: 6
Permutation: [4 0 9 5 8 7 3 2 6 1], Order: 10
Permutation: [6 3 2 0 5 4 1 8 7 9], Order: 4
Permutation: [1 5 4 9 6 8 7 3 2 0], Order: 10
Permutation: [7 8 2 4 6 0 3 5 9 1], Order: 3
Permutation: [3 6 1 4 9 2 8 5 0 7], Order: 10
```

Training a model to compute order of random permutations of fixed length.

In [59]:

```python
from torch.utils.data import DataLoader, Dataset #DataLoader efficiently load data in batches during training
#Dataset provides an abstract interface for custom datasets

import torch.nn as nn #building blocks and classes for creating neural networks.

from numpy.typing import NDArray # annotation for arrays (tensors) with a specific number of dimensions
```

Some background on one hot encoding:

The input text data can be categorical or numerical in nature. In this problem, we have numerical data as input which is a permutation of integers. We might think that our model would train on this classification problem of learning to determine order of permutation but due to numerical data in inputs, their is an inherent relationshp between the numbers due to natural ordering. This brings in numerical bias. For example, the model might assume that the number "4" is more important or significant than "1" or "2," which could lead to incorrect predictions.In order to remove this, we represent our inputs as one hot encoded vectors.

How to write a one hot encoded vector:

- First, we determine the unique categories (numbers) in the dataset, say 4.
- For each number, we create a binary vector of the same length as the number of unique categories, where all elements are 0 except for the position corresponding to the category. For example, "2" will be represented by $[0,1,0,0]$, "4" by $[0,0,0,1]$, and so on.

One-hot-encoding is a powerful technique to treat categorical or numerical data, but it can lead to increased dimensionality (by replacing numbers or catogaries by vectors), sparsity (a lot of zero values in the tensor), and overfitting (not responding well on unseen data, not learning but memorising answers).

Step 1: Generating a dataset to train the model:

An element of a dataset consists of an input permutation (perm) and it's expected output (class). We will replace numerical data in each permutation by a flat permutation matrix which is made up of binary vectors. Each such vector is defined to be non-zero only at the index corresponding to the number it replaces. For example, if we are looking at permutation $(0\ 1\ 2)$ of length 3, we will represent it by $(1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1)$

We define class of a permutation as (order - 1). This is because the index in Python starts with 0. We will be using one hot encoded vectors to represent these classes. For example, the order of $(0\ 1\ 2)$ os 1, so, it belongs to the class $[1,0,0]$.

Upper bound fixed for order: max order = 10

In [56]:

```python
# Let's define a function that converts numerical data into binary array with respect to index.
def flat_permutation_matrix(perm:tuple) -> NDArray:
    n = len(perm)
    mat = np.zeros((n,n))
    for i, v in enumerate(perm):
        mat[i,v] = 1
    return mat.reshape((n**2))

rand = np.random.default_rng(seed=1)
perm1 = rand.permutation(3)
perm2 = rand.permutation(3)
perm3 = rand.permutation(3)
print(perm1,', Permutation matrix 1:',flat_permutation_matrix(perm1))
print('\n')
print(perm2,', Permutation matrix 2:',flat_permutation_matrix(perm2))
print('\n')
print(perm3,', Permutation matrix 3:',flat_permutation_matrix(perm3))
```

```
[0 1 2] , Permutation matrix 1: [1. 0. 0. 0. 1. 0. 0. 0. 1.]


[2 0 1] , Permutation matrix 2: [0. 0. 1. 1. 0. 0. 0. 1. 0.]


[1 0 2] , Permutation matrix 3: [0. 1. 0. 1. 0. 0. 0. 0. 1.]
```

In [52]:

```python
# Step 1:
# Define a function to generate a tensor dataset of random permutations and their respective class.
def generate_dataset(s : int, n: int, L: int):
    """
    - s: Fixed seed of numpys random generator default_rng.
    - n: Fixed length of each permutation
    - L: Total number of permutations.
    """

    # Step 2:
    # Generate and stack L permutations of length n in an array of shape (L x n) with order <= 10
    # Compute the (order - 1) of these permutations and stack them in an array of shape (L x 1).
    gen = np.random.default_rng(s)
    permutation_list = []
    order_list = []
    count = 0
    while count < L:
        perm = gen.permutation(n)
        order = order_of_perm(perm)
        if order <= 10:
            permutation_list.append(perm)
            order_list.append((order -1))
            count += 1
    permutations = np.stack([perms for perms in permutation_list], axis=0)
    gen = np.random.default_rng(s)
    classes = np.stack([orders for orders in order_list], axis=0)
    #print(permutations) # Check if it works.
    """
    - By default np.stack(arrays, axis=0) stacks the arrays along axis=0, that is, stacking along rows.
    - Restart with the same seed s. This is crucial or else the orders won't belong to the above permutations.
    """

    # Step 3: Replace the stack of permutations with the flat permutation matrix.
    permutation_mats = np.stack([flat_permutation_matrix(p) for p in permutations])

    # Step 4:
    # Covert our numpy stacks into torch tensors.
    permutations_tensor = torch.tensor(permutation_mats)
    classes_tensor = torch.tensor(classes)

    # Step 5:
    # Convert the classes_tensor into one hot encoded vector representation.
    classes_tensor_hot = torch.nn.functional.one_hot(classes_tensor, num_classes=10)

    # Step 5:
    # Return the permutations_tensor and its orders_tensor wrapped in a TensorDataset.
    return torch.utils.data.TensorDataset(permutations_tensor.float(), classes_tensor_hot.float())
    """
    - PyTorch’s TensorDataset is a Dataset wrapping tensors.
    - This will make it easier to access both the independent and dependent variables in the same line as we train.
    """


# Step 6:
# Let's generate datstset of 10 permutations of length 7 and their respective classes.
rand = np.random.default_rng(seed=1)
dataset = generate_dataset(1, 7, 10)
for i in range(10):
    print(dataset[i])
```

```
(tensor([0., 0., 0., 0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.]), tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.]))
(tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
        0., 0., 0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0.]), tensor([0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]))
(tensor([0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0.]), tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.]))
(tensor([1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0.]), tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]))
(tensor([0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
        0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 0.]), tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
(tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0.]), tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]))
(tensor([1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0.]), tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]))
(tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1.,
        0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]), tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.]))
(tensor([1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
        1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.]), tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]))
(tensor([0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1.]), tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]))
```

STEP 2: Forward pass.

We now define our model and it's forward pass. It involves creating the architecture of the neural network and specifying how data flows through the network to produce predictions.

- **Model Architecture:** The first step is to decide on the architecture of the neural network. This includes determining the depth of the network, the width of each layer, the type of layers (e.g., fully connected, convolutional, recurrent).
- **Model Class**: In Python, you create a custom model by defining a class that inherits from PyTorch's nn.Module. The class serves as a blueprint for the neural network.
- **Constructor (init method):** Within the model class, you define the constructor (**init** method), where you specify the width of layers and parameters of the neural network. You can also pass any hyperparameters required for the model.
- **Forward Pass (forward method):** The most critical part of the model class is the forward pass, which is defined in the forward method. This method outlines how data moves through the layers of the network to produce predictions. It takes the input data as an argument and returns the model's output. The input tensor undergoes transformations in this method.

  > - Inside the forward method, you call each layer in sequence, passing the data from one layer to the next. You can also introduce activation functions (e.g., ReLU, sigmoid, softmax) between layers to introduce non-linearity and enable the network to learn complex patterns.
  > - The last layer of the network is typically the output layer, which produces the final predictions based on the learned representations from the previous layers. The activation function used in the output layer depends on the type of task (e.g., binary classification, multi-class classification, regression).

In [49]:

```python
# Step 1:
# Define a neural network model called FCModel.
class FCModel(nn.Module):
    """
    This line defines a new Python class named FCModel, which will be a custom neural network model.
    It inherits from nn.Module, which is the base class for all PyTorch neural network models
    """

    # Step 2:
    # Define the constructor method for the FCModel class.
    def __init__(self, n: int, hidden_layers: List[int]):
        """
        n: An integer representing the input dimension (number of features).
        hidden_layers: A list of integers representing the width of each hidden layer.
        """
        super().__init__()
        self.n = n
        self.hidden_layers = hidden_layers
        """
        These lines store the values of n and hidden_layers as attributes of the FCModel object so that
        they can be accessed later within the class.
        """

        # Step 3:
        # Create a zipped list of fully connected (nn.Linear) layers based on the number of input nodes,
        # the number of hidden nodes in each hidden layer, and the number of output nodes.
        self.layers = nn.ModuleList([
            nn.Linear(in_dim, out_dim)
            for in_dim, out_dim in zip([n**2, *hidden_layers], [*hidden_layers, 10])
        ])
        """
        - The nn.ModuleList is a container for holding layers in PyTorch.
          It is used here to store the list of linear layers that were created in the list.
        - The zip() function is used to pair corresponding elements from two lists: [n**2, *hidden_layers] and
          [*hidden_layers, 10]. This creates pairs of (in_dim, out_dim) values.
        """

    # Step 4:
    # Define the forward method via which the model predicts the output.
    def forward(self, x: torch.tensor):

        # Step 5:
        # Apply linear transformation on the input tensor x defined by the layer's weights and biases.
        for i, layer in enumerate(self.layers):
            x = layer(x)

            # Step 6:
            # Introduce non-linearity using ReLU. (We don't want to ReLU the last layer)
            if i != len(self.layers) - 1:
                x = nn.functional.relu(x)
            """
            - This for loop iterates over each layer in the self.layers list.
            """

        # Step 7:
        # Ensure that the final output values are within the range of 0 to 1 by using Sigmoid activation function.
        return torch.sigmoid(x)


# Test if the model has all its sizes etc set up right.
rand = np.random.default_rng(seed=1)
model0 = FCModel(n=7, hidden_layers=[500, 100])
rand = np.random.default_rng(seed=1)
inputs, expected_output = dataset[:]
print(model0(inputs))
print(model0(inputs).shape) # Check the shape of the output tensor.
```

```
tensor([[0.4802, 0.4883, 0.4962, 0.4984, 0.4893, 0.5093, 0.4933, 0.4898, 0.5114,
         0.5130],
        [0.4837, 0.4938, 0.5060, 0.4859, 0.4983, 0.5210, 0.4900, 0.4884, 0.5161,
         0.5097],
        [0.4869, 0.4901, 0.4974, 0.4932, 0.4965, 0.5155, 0.4934, 0.4924, 0.5121,
         0.5100],
        [0.4787, 0.4992, 0.4965, 0.5087, 0.5056, 0.5178, 0.4907, 0.4986, 0.5172,
         0.5052],
        [0.4921, 0.4939, 0.5020, 0.4959, 0.5051, 0.5129, 0.4967, 0.4900, 0.5161,
         0.5083],
        [0.4853, 0.5011, 0.4950, 0.4803, 0.4931, 0.5157, 0.4845, 0.4905, 0.5158,
         0.5157],
        [0.4847, 0.4956, 0.5129, 0.4879, 0.4965, 0.5269, 0.5003, 0.4984, 0.5125,
         0.5124],
        [0.4867, 0.4881, 0.5029, 0.4910, 0.4994, 0.5192, 0.4985, 0.4816, 0.5218,
         0.5069],
        [0.4822, 0.5016, 0.5052, 0.5057, 0.5039, 0.5208, 0.4963, 0.4856, 0.5235,
         0.4984],
        [0.4691, 0.4913, 0.4959, 0.4926, 0.4889, 0.5136, 0.4960, 0.4875, 0.5198,
         0.5050]], grad_fn=<SigmoidBackward0>)
torch.Size([10, 10])
```

STEP 3: Count Correct Function to check accuracy.

In [47]:

```python
# Step 1:
# Define the count_correct function with two arguments: model and data.
def count_correct(model, data):

    # Step 2:
    # Unpack the given data conisisting of inputs (permutations) and expected outputs (order of permutations).
    perms, orders = data[:]
    """
    - Data is a tuple containing two tensors: perms and orders. 'data[:]' unpacks these two tensors.
    """

    # Step 3:
    # Call the models forward pass on permutation tensor to generate predicted outputs.
    with torch.no_grad():
        output = model(perms)
        classification = (model(perms) >= 0.5).int()
        """
        - 'with torch.no_grad()' ensures that the operations are performed without tracking gradients.
        """

    # Step 4:
    # Define a label 'total' that gives the total number of permutations.
    total = orders.shape[0]
    """
    - 'perms.shape[0]' accesses the size of the first dimension of the perms tensor.
    """

    # Step 5:
    # Initialize a counter variable 'correct' to keep track of the number of correct predictions
    correct = 0

    # Step 6:
    # For each input (i), check if predicted ouput (classification[i]) matches the expected output (order[i]).
    for i in range(orders.shape[0]):
        if torch.all(classification[i]==orders[i]):
            correct = correct +1
    """
        - torch.all checks if all elements in the tensors are equal, returning True if they are & False otherwise.
        - Here there is only one element in a single output.
    """

    # Step 7:
    # Ask the count function to return the number of correct predictions as well as the total predictions.
    return correct, total


# Step 8:
# Print the predicted outputs before training & compare them with the expected ouput in the dataset defined above.
perms, orders = dataset[:] #orders id expected output
outputs = model0(perms) #classification is predicted output
for i in range(10):
    print(outputs[i],orders[i])
print(count_correct(model0, dataset)) #0 out of 10 correct (as expected.)
```

```
tensor([0.4899, 0.5104, 0.4765, 0.5155, 0.4793, 0.5154, 0.4699, 0.4754, 0.5328,
        0.5014], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
tensor([0.4927, 0.5036, 0.4863, 0.5149, 0.4895, 0.5056, 0.4714, 0.4853, 0.5303,
        0.5107], grad_fn=<SelectBackward0>) tensor([0., 0., 1., 0., 0., 0., 0., 0., 0., 0.])
tensor([0.4981, 0.5074, 0.4744, 0.5236, 0.4704, 0.5101, 0.4731, 0.4836, 0.5141,
        0.5196], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
tensor([0.4981, 0.5041, 0.4808, 0.5140, 0.4862, 0.4965, 0.4688, 0.4843, 0.5322,
        0.5050], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
tensor([0.4912, 0.5040, 0.4951, 0.5189, 0.4878, 0.5022, 0.4848, 0.4864, 0.5257,
        0.5041], grad_fn=<SelectBackward0>) tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0.])
tensor([0.4875, 0.5141, 0.4842, 0.5224, 0.4681, 0.5031, 0.4709, 0.4883, 0.5281,
        0.5131], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
tensor([0.5021, 0.4998, 0.4750, 0.5225, 0.4772, 0.5098, 0.4630, 0.4791, 0.5163,
        0.5116], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
tensor([0.5018, 0.5017, 0.4801, 0.5220, 0.4794, 0.5086, 0.4836, 0.4769, 0.5142,
        0.5082], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
tensor([0.4940, 0.5019, 0.4838, 0.5242, 0.4787, 0.5040, 0.4684, 0.4860, 0.5094,
        0.5156], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
tensor([0.4953, 0.5094, 0.4743, 0.5259, 0.4655, 0.5218, 0.4671, 0.4733, 0.5365,
        0.5249], grad_fn=<SelectBackward0>) tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
(0, 10)
```

Step 4: Set the training hyper-parameters.

In [54]:

```python
# Train and test data: by setting the random seed here, we get the different data each time.
train_data = generate_dataset(1, 7, 20000) #seed = 1
test_data = generate_dataset(2, 7, 2000)# seed = 2

# Model shape. Note that PyTorch will randomise the initial model parameters each time here.
Model = FCModel(7, [500, 100])       # can experiment here!!

# Loss function.
loss_fn = nn.BCELoss() # binary cross entropy

# Optimisation method.
optimiser = torch.optim.SGD(Model.parameters(), lr=0.2, momentum = 0.5) # can mess with learning rate if you want

# Epochs to use for training. The whole training dataset will be seen once during each epoch.
epochs = 100

# Minibatch size: the error over a minibatch will be accumulated before a gradient step.
# There are multiple minibatches within an epoch.
minibatch_size = 50
```

Step 5: Train the model.

In [60]:

```python
train_loss = np.zeros(epochs)
test_loss = np.zeros(epochs)
"""
- These are NumPy arrays initialized with zeros, each of length epochs.
  They will be used to store the training and testing losses for each epoch during the training process.
"""

# Loop through epochs:
for epoch in range(epochs):
    """
    - The outer loop runs for epochs number of times, where epochs is the number of times the
      entire dataset is passed through the model during training.
    """

    # Training Loop:
    for perms, expected in torch.utils.data.DataLoader(train_data, batch_size=minibatch_size, shuffle=True):
        """
        - The inner loop iterates over batches of training data (perms and expected) using the DataLoader
          from PyTorch. The DataLoader allows iterating through the dataset in batches to speed up training.
        """
        optimiser.zero_grad() # Clear's accumulated gradients after each loop.
        output = Model(perms) # Applies the forward pass.
        loss = loss_fn(output, expected) # Computes the loss.
        loss.backward() # Computes the gradients using backpropagation.
        optimiser.step() # Updates the model's parameters.

    # Evaluation Loop:
    with torch.no_grad():
        """
        - After each epoch, the model is evaluated on both the training and testing datasets
          (with torch.no_grad() to avoid accumulating gradients during evaluation).
        """
        train_inputs, train_expected_output = train_data[:]
        train_output = Model(train_inputs)
        train_loss[epoch] = float(loss_fn(train_output, train_expected_output)) #Compute and store Train Loss.

        test_inputs, test_expected_output = test_data[:]
        test_output = Model(test_inputs)
        test_loss[epoch] = float(loss_fn(test_output, test_expected_output)) #Compute and store Test Loss.

    # Print the results:
    if epoch % 2 == 0:
        """
        - This condition checks if the current epoch number (epoch) is divisible by 2.
          The purpose of this check is to control when to print the evaluation results.
        """

        correct, total = count_correct(Model, test_data)
        accuracy = correct / total
        print(f"Epoch {epoch:4}: Train loss {train_loss[epoch]:.2f}, Test loss {test_loss[epoch]:.2f}, {correct:5} out of {total:5} correct ({accuracy:.2%})")
        """
        The printed message includes:
        - The epoch number (epoch) formatted with four characters ({epoch:4}) to align the output.
        - The training loss for the current epoch (train_loss[epoch]) formatted with two decimal places
          ({train_loss[epoch]:.2f}).
        - The testing loss for the current epoch (test_loss[epoch]) formatted with two decimal places
          ({test_loss[epoch]:.2f}).
        - The number of correct predictions (correct) and the total number of samples (total) in the
          testing data, both formatted with five characters ({correct:5} and {total:5}) to align the output.
        - The accuracy on the testing data (accuracy) formatted as a percentage with two decimal places
          ({accuracy:.2%}).
        """
```

```
Epoch    0: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch    2: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch    4: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch    6: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch    8: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   10: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   12: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   14: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   16: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   18: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   20: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   22: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   24: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   26: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   28: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   30: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   32: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   34: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   36: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   38: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   40: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   42: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   44: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   46: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   48: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   50: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   52: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   54: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   56: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   58: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   60: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   62: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   64: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   66: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   68: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   70: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   72: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   74: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   76: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   78: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   80: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   82: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   84: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   86: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   88: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   90: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   92: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   94: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   96: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
Epoch   98: Train loss 0.00, Test loss 0.00,  1999 out of  2000 correct (99.95%)
```