The goal is to show that the use of backpropagation can capture properties of the input in the hidden layer that are not explicitly represented by the input.

The use of less hidden units than input units imposes a constraint on the problem and forces the neural network to rerepresent the input units.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn import preprocessing
```

This neural network's purpose is to learn the target function f(x) where f(x) is a vector which contains seven 0's and a 1. This has been represented below using a pandas dataframe where there are 8 rows which represent the 8 different vector combinations that can make up f(x).

In [2]:
```
index = ["1", "2", "3", "4", "5", "6", "7", "8"]
vectors = ["v_1", "v_2", "v_3", "v_4", "v_5", "v_6", "v_7", "v_8"]

data = np.array([[1, 0, 0, 0, 0, 0, 0 , 0],
                 [0, 1, 0, 0, 0, 0, 0 , 0],
                 [0, 0, 1, 0, 0, 0, 0 , 0],
                 [0, 0, 0, 1, 0, 0, 0 , 0],
                 [0, 0, 0, 0, 1, 0, 0 , 0],
                 [0, 0, 0, 0, 0, 1, 0 , 0],
                 [0, 0, 0, 0, 0, 0, 1 , 0],
                 [0, 0, 0, 0, 0, 0, 0 , 1]])
```

In [3]:
```
df = pd.DataFrame(data=data, index=vectors, columns=index)
df
```

Out[3]:

	1	2	3	4	5	6	7	8
v_1	1	0	0	0	0	0	0	0
v_2	0	1	0	0	0	0	0	0
v_3	0	0	1	0	0	0	0	0
v_4	0	0	0	1	0	0	0	0
v_5	0	0	0	0	1	0	0	0
v_6	0	0	0	0	0	1	0	0
v_7	0	0	0	0	0	0	1	0
v_8	0	0	0	0	0	0	0	1

The data is split into X and Y where X is the input that we want the neural network to use to predict the output values in Y.

Normally X is taken to be a subset of the total number of columns avaliable and Y is chosen to be the value that we wish to predict. For example, X could be the number of bedrooms and square footage in a house and Y could be the value of that house. We use the info in X to predict the value of Y and then we can compare this Y to the values in our training sample.

However as we want the input data and output data to be identical, the values of X and Y are equal.

In [4]:
```
# split into X and Y
Y = df
X = df

print(X.shape)
print(Y.shape)

# convert to numpy arrays
X = np.array(X)
```

```
(8, 8)
(8, 8)
```

The sizes of the input, hidden and output layers are defined to be 8,3 and 8 as the purpose of this exercise is to have a smaller hidden layer such that the 3 hidden layers will be forced to represent the 8 input layers.

In [5]:
```
input_unit = X.shape[0]
hidden_unit = 3
output_unit = Y.shape[0]
```

Inital values of W1 and W2 are created using a small random value. The 2 bias weights are initalised as zero vectors.

In [6]:
```
def parameters_initialization(input_unit, hidden_unit, output_unit)
    np.random.seed(2)
    W1 = np.random.randn(hidden_unit, input_unit)*0.1
    b1 = np.zeros((hidden_unit, 1))
    W2 = np.random.randn(output_unit, hidden_unit)*0.1
    b2 = np.zeros((output_unit, 1))
    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

The activation functions used in neural networks are Sigmoid, tanh, Softmax, ReLU, Leaky ReLU. A different activation fucntion can be chosen for each layer of the neural network.

T.Michell does not specify which activation function that was used to obtain the hidden values found in the diagram and thus two arbitrary activation functions are used. The tanh function is used for the hidden layer while the sigmoid function is used for the output layer.

In forward propagation, the input data is fed through the neural network in a forward direction and computes the predicted error.

In [7]:
```
def sigmoid(z):
    return 1/(1+np.exp(-z))
def forward_propagation(X, parameters):
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache = {"Z1": Z1,"A1": A1,"Z2": Z2,"A2": A2}

    return A2, cache
```

For back propagation, the partial derivatives of the error function E are computed. This allows the weights to be adjusted to increase the accuracy of the neural network. The gradient descent technique is used which is discussed below.

Note: In class, we focused on these partial derivatives using the sigmoid function for both the hidden layer and the output layer. However below, this changes for dW1 and db1 as we are now using the tanh function as the activation function for the hidden layer.

In [8]:
```
def backward_propagation(parameters, cache, X, Y):
    m = X.shape[1]

    W1 = parameters['W1']
    W2 = parameters['W2']
    A1 = cache['A1']
    A2 = cache['A2']

    dZ2 = A2-Y
    dW2 = (1/m) * np.dot(dZ2, A1.T)
    dZ2 = np.array(dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
    dZ1 = np.array(dZ1)
    dW1 = (1/m) * np.dot(dZ1, X.T)
    db1 = (1/m)*np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2,"db2": db2}

    return grads
```

The parameters are updated through each iteration using the gradient descent method. The batch method is implemeted below where each value is iterated through, this is chosen as the neural network and number of iterations is small. The learning rate that is chosen is 0.3 to keep with the values used in 'Machine Learning' by T.Mitchell.

In [9]:
```
def gradient_descent(parameters, grads, learning_rate):
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {"W1": W1, "b1": b1,"W2": W2,"b2": b2}

    return parameters
```

The neural network is ran for 5000 iterations by applying forward progation followed by backward propagation followed by gradient descent.

In [10]:
```
def neural_network_model(X, Y, hidden_unit, num_iterations):
    np.random.seed(32)

    parameters = parameters_initialization(input_unit, hidden_unit,

    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    for i in range(0, num_iterations):
        A2, cache = forward_propagation(X, parameters)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = gradient_descent(parameters, grads, 0.3)
    return parameters
parameters = neural_network_model(X, Y, 3, 5000)
```

Using the final values for the parameters W1,b1,W2,b2 and the input values X, a value for Y is predicted.

In [11]:
```
def prediction(parameters, X):
    A2, cache = forward_propagation(X, parameters)
    predictions = np.round(A2)

    return predictions
```

In [12]:
```
A2, cache = forward_propagation(X, parameters)
```

Looking at the values of the predicted Y, it is clear by comparision that this equals the real output Y.

In [13]:
```
predictions = prediction(parameters, X)
print('The predicted values for Y are \n' , predictions)
print('The true values for Y are \n' , Y)
```

```
The predicted values for Y are
 [[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
The true values for Y are
      1  2  3  4  5  6  7  8
v_1   1  0  0  0  0  0  0  0
v_2   0  1  0  0  0  0  0  0
v_3   0  0  1  0  0  0  0  0
v_4   0  0  0  1  0  0  0  0
v_5   0  0  0  0  1  0  0  0
v_6   0  0  0  0  0  1  0  0
v_7   0  0  0  0  0  0  1  0
v_8   0  0  0  0  0  0  0  1
```

In [14]:
```
w1 = parameters['W1']
b1 = parameters['b1']
w2 = parameters['W2']
b2 = parameters['b2']
```

The hidden values are shown below, as tanh was used as the activation function, the range of values are from -1 to 1. These are then normalised and rounded to the range of 0 to 1. Looking back, the sigmoid activation function would have been more suitable to this problem as it outputs out vaues from 0 to 1 thus no normalisation would have been required.

In [15]:
```
hidden_values = cache['A1'].T
```

In [16]:
```
hidden_values
```

Out[16]:
```
array([[ 0.99490787, -0.04302783, -0.92818738],
       [-0.90009014, -0.93967381,  0.98884459],
       [ 0.02941738,  0.99386195, -0.94039297],
       [ 0.98758792,  0.97646282,  0.94002666],
       [-0.06999129, -0.99436434, -0.91497241],
       [ 0.97236054, -0.97592586,  0.98942764],
       [-0.82942252,  0.97808555,  0.99074451],
       [-0.99540778,  0.04510161, -0.78967455]])
```

In [17]:
```
def NormalizeData(data):
    return (data - np.min(data)) / (np.max(data) - np.min(data))
```

In [18]:
```
normalised = NormalizeData(hidden_values)
normalised_hidden_values = preprocessing.minmax_scale(hidden_values
hidden_values_2 = [np.round(x,3) for x in normalised_hidden_values]
hidden_values_rounded = [np.round(x) for x in normalised_hidden_val
```

Below, the hidden values are seen first rounded to 3 decimal places and then rounded to 0 or 1.

In [24]:
```
print('The hidden values when normalised and rounded to 3 decimal p
hidden_values_2
```

```
The hidden values when normalised and rounded to 3 decimal places
are
```

Out[24]:
```
[array([1.   , 0.478, 0.006]),
 array([0.048, 0.028, 0.999]),
 array([0.515, 1.   , 0.   ]),
 array([0.996, 0.991, 0.974]),
 array([0.465, 0.   , 0.013]),
 array([0.989, 0.009, 0.999]),
 array([0.083, 0.992, 1.   ]),
 array([0.   , 0.523, 0.078])]
```

In [26]:
```
print('The hidden values when normalised and rounded to 0 decimal p
hidden_values_rounded
```

```
The hidden values when normalised and rounded to 0 decimal places
are
```

Out[26]:
```
[array([1., 0., 0.]),
 array([0., 0., 1.]),
 array([1., 1., 0.]),
 array([1., 1., 1.]),
 array([0., 0., 0.]),
 array([1., 0., 1.]),
 array([0., 1., 1.]),
 array([0., 1., 0.])]
```

It can be seen that each of the 8 vectors in the input are uniquely represented by a hidden layer value which consists of 3 binary values. Thus this shows how the use of backpropagation can capture properties of the input units.

(1,0,0,0,0,0,0,0) is represented by (1,0,0)
(0,1,0,0,0,0,0,0) is represented by (0,0,1)
(0,0,1,0,0,0,0,0) is represented by (1,1,0)
(0,0,0,1,0,0,0,0) is represented by (1,1,1)
(0,0,0,0,1,0,0,0) is represented by (0,0,0)
(0,0,0,0,0,1,0,0) is represented by (1,0,1)
(0,0,0,0,0,0,1,0) is represented by (0,1,1)
(0,0,0,0,0,0,0,1) is represented by (0,1,0)

In [ ]: