Homework 2

Hai Dinh (19960331-4494)

1. 3D Boolean Functions

1.1 How many 3D Boolean functions are there?

Each 3D Boolean function has 3 inputs, where each input can either be `true` or `false` (so only 2 possibilities). Hence, there are $2^3 = 8$ combinations of these 3 input variables.

Each of these combinations above can be mapped to the output of either `true` or `false` (again 2 possibilities). Hence, there are a total of $2^8 = 256$ such mappings possible, and this is the reason why there are 256 different 3D Boolean functions.

In general, there are $2^{(2^n)}$ nth-dimensional Boolean functions.

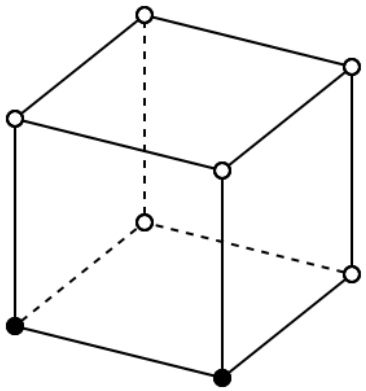
1.2. How many symmetries of 3D Boolean functions mapping 2 input patterns to 1?

For each cube representing a 3D Boolean function, a white corner will represent target $t^{(\mu)} = 0$ and a black corner will represent target $t^{(\mu)} = 1$.

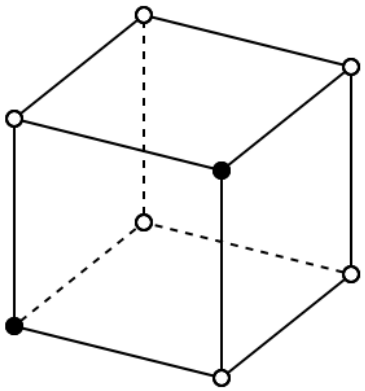
This means a 3D Boolean function that maps 2 of the 8 possible inputs patterns to output 1 must be represented by a cube where exactly 2 of corners are colored black.

In this case, we will have 3 symmetries in total:

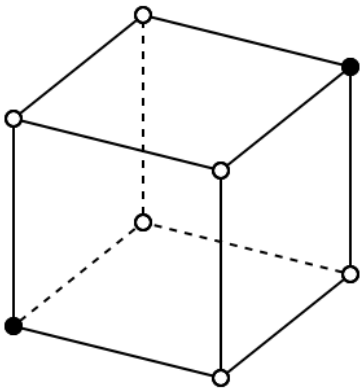
- **Symmetry 1**: This is when 2 corners on the same edge of the cube are black.
- **Symmetry 2**: This is when 2 corners on the diagonal of a surface of the cube are black.
- **Symmetry 3**: This is when 2 corners on the diagonal of the cube itself are black.



Symmetry 1



Symmetry 2



Symmetry 3

1.3. How many linearly separable 3D Boolean functions are there?

Graphically speaking, a 3D Boolean function is linearly separable if it can be represented by a cube where it is possible for a 2D plane to separate all white corners from black corners. This can only be achieved if a black corner is connected to at least another black corner (in case there are 2 or more black corners in the cube). So our strategy is to identify all symmetries that are linearly separable, and then count how many Boolean functions belong to each of the symmetries.

Let k be the number of black corners in the cube. For each $k \in \{1, \ldots, 8\}$, the idea now is to identify all symmetries that are linearly separable. As shown in the figure below, there is only 1 such symmetry for $k \in \{0, 1, 2, 3\}$, but there are 2 such symmetries for $k = 4$.

- For $k = 0$, the symmetry has 1 Boolean function, where all corners are white.
- For $k = 1$, the symmetry has 8 Boolean functions, because the black corner can be placed on any of the 8 corners of the cube.
- For $k = 2$, the symmetry has 12 Boolean functions, because the 2 black corners can be placed along any of the 12 edges of the cube.
- For $k = 3$, the symmetry has 24 Boolean functions. This is because the 3 black corners can actually be placed in 4 different ways (or orientations if you will) on a single surface of the cube. And since the cube has 6 surfaces, this amounts to $4 \times 6 = 24$ functions.
- For $k = 4$, the 1st symmetry has 6, and 2nd one has 8, giving the total of 14 Boolean functions.

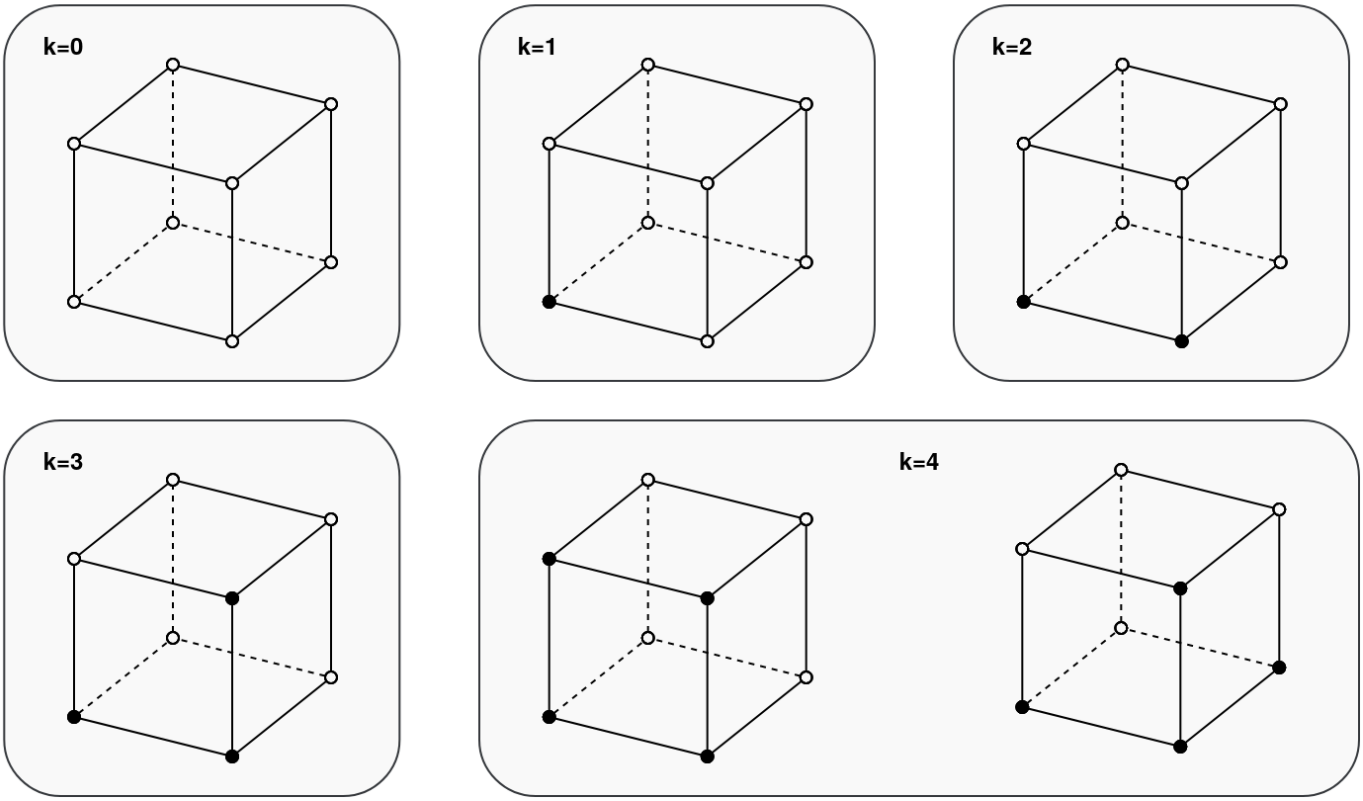
  The 1st symmetry has 6 functions, because the 4 black corners can be placed on any of the 6 surfaces of the cube.

  The 2nd symmetry has 8 functions, because the "middle" black corner (the one that is connected to the other 3 black corners) can be placed on any of the 8 corners of the cube.

Note that we don't have to do the same analysis for $k \in \{5, 6, 7, 8\}$, because having k black corners is also the same as having $8 - k$ white corners, which will give the same results as when $k \in \{0, 1, 2, 3\}$. This also means we need to multiply the results by 2 for each $k \in \{0, 1, 2, 3\}$.

Therefore, the total number of 3D Boolean functions are that linearly separable is: $(1 + 8 + 12 + 24) \times 2 + 14 = 104$.

Linearly separable symmetries for different values of k:



2. Linear separability of 4D Boolean functions

2.1. Code

```
import time
import numpy as np

# Global parameters
eta = 0.02
max_number_of_updates = int(1e5)
max_number_of_repetitions = 10

# Loading data
X = np.genfromtxt("input_data_numeric.csv", delimiter=",")[:,1:]
Y = {
    "A": np.array([+1, +1, -1, +1, -1, +1, +1, +1, -1, -1, -1, -1, -1, -1, -1, -1]),
    "B": np.array([+1, -1, +1, +1, +1, -1, +1, +1, +1, +1, +1, +1, -1, +1, +1, +1]),
    "C": np.array([-1, -1, -1, +1, -1, -1, +1, -1, +1, -1, +1, +1, -1, +1, -1, +1]),
    "D": np.array([+1, -1, +1, -1, -1, +1, -1, -1, -1, -1, -1, -1, +1, -1, -1, +1]),
    "E": np.array([+1, +1, +1, +1, -1, -1, -1, -1, -1, +1, -1, -1, +1, -1, -1, -1]),
    "F": np.array([-1, -1, +1, +1, +1, -1, -1, -1, -1, +1, +1, -1, +1, +1, +1, -1]),
}

def check_linear_separability(function_name):
    start_time = time.time()
    y_true = Y[function_name]
    linearly_separable = False

    for i_repetition in range(1, max_number_of_repetitions + 1):
        W = np.random.uniform(low=-0.2, high=0.2, size=(4,))
        theta = np.random.uniform(low=-1.0, high=1.0, size=None)

        for i_update in range(1, max_number_of_updates + 1):
            # Forward propagate for all data points
            local_fields = (X @ W - theta) / 2
            outputs = np.tanh(local_fields)
            y_pred = np.sign(outputs)

            # Check convergence criteria using all data points
            if np.array_equal(y_pred, y_true):
                linearly_separable = True
                break

            # Backward propagate to update weights and threshold
            mu = np.random.choice(X.shape[0])
            error = (y_true[mu] - outputs[mu]) * (1 - np.tanh(local_fields[mu])**2)
            W += eta * error * X[mu]
            theta -= eta * error

        if linearly_separable:
            break

    print("Results for Boolean function " + function_name + ":")
    print("    Linearly separable:", "YES" if linearly_separable else "NO")
```

```
    print("    Final weights     :", W)
    print("    Final threshold   :", theta)
    print("    Total time taken  :", (time.time() - start_time), "seconds")
    print("    Total #repetitions:", i_repetition)
    print("    Total #updates in last repetition: ", i_update)

if __name__ == "__main__":
    [check_linear_separability(name) for name in Y.keys()]
```

2.2. Results

```
Results for Boolean function A:
    Linearly separable: YES
    Final weights     : [ 0.38027807 -0.64544005 -0.29010912 -0.64814763]
    Final threshold   : 0.1591475899711617
    Total time taken  : 0.0032126903533935547 seconds
    Total #repetitions: 1
    Total #updates in last repetition:  105
Results for Boolean function B:
    Linearly separable: YES
    Final weights     : [-0.63748586 -0.15120174  0.64939551  0.31684523]
    Final threshold   : -1.1135515460963725
    Total time taken  : 0.0048370361328125 seconds
    Total #repetitions: 1
    Total #updates in last repetition:  170
Results for Boolean function C:
    Linearly separable: YES
    Final weights     : [ 0.42798437 -0.34069052  1.56669423 -0.403037  ]
    Final threshold   : 0.4375861576029526
    Total time taken  : 0.00819253921508789 seconds
    Total #repetitions: 1
    Total #updates in last repetition:  281
Results for Boolean function D:
    Linearly separable: NO
    Final weights     : [ 1.8602917   3.94948184 -2.24257376 -0.04425914]
    Final threshold   : 3.9712242478853836
    Total time taken  : 29.376785039901733 seconds
    Total #repetitions: 10
    Total #updates in last repetition:  100000
Results for Boolean function E:
    Linearly separable: NO
    Final weights     : [-5.21795027 -2.61466707 -5.17015406 -2.6411312 ]
    Final threshold   : 2.632913232394134
    Total time taken  : 29.003007888793945 seconds
    Total #repetitions: 10
    Total #updates in last repetition:  100000
Results for Boolean function F:
    Linearly separable: NO
    Final weights     : [-2.76792626  0.06457685  0.22788824  2.47077662]
    Final threshold   : -0.1663542164269784
    Total time taken  : 29.53967833518982 seconds
    Total #repetitions: 10
    Total #updates in last repetition:  100000
```

3. Two-layer perceptron

3.1. Method

Based on my experimentation, it's enough to have 8 neurons in the first hidden layer, and 16 neurons in the 2nd hidden layer. The learning rate can be set at 0.02. The weights are initialized with Gaussian distribution with mean 0, and 1 as the standard deviation. The thresholds are initialized with zeros (following the advice from the book). With this setup, the percentron is able to predict with less than 0.12 error rates on the validation approximately after 25 epochs.

The implementation closely follows "Stochastic Gradient Descent" algorithm from Chapter 6 in the course book, with some slight modifications here and there related to the notations and also the dimensionality of different matrix quantities. Note that all computations in the implementation have been vectorized using numpy for faster performance (this goes for both training and validation).

To explain the vectorizations done during the training of the perceptron (note that validation is closely similar), let's introduce some notations. For each layer l after the input layer, let:

- $N^{(l)}$ be the number of neurons/nodes in layer l.
- $W^{(l)}$ be the weight matrix of shape $N^{(l)} \times N^{(l-1)}$.
- $T^{(l)}$ be the threshold vector of shape $N^{(l)} \times 1$.
- $X^{(l)}$ be the input matrix of shape $N^{(l-1)} \times 1$.
- $B^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the local fields of the layer.
- $O^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the outputs of the layer.
- $E^{(l)}$ be the matrix of shape $N^{(l)} \times 1$, storing the errors of the layer.
- $\mathbf{t}$ be the target output (from the dataset) of shape 1×1.
- $\odot$ will be used to indicate Hadamard product (i.e., element-wise multiplication).

Step 1: For layer $l = 1, \ldots, L$, forward propagate by:

- $B^{(l)} := W^{(l)} X^{(l)} - T^{(l)}$
- $O^{(l)} := \tanh\left(B^{(l)}\right)$
- $X^{(l+1)} := O^{(l)}$

Step 2: Compute the error of the output layer (L) by:

- $E^{(L)} := \left(\mathbf{t} - O^{(L)}\right) \odot \left(1 - \tanh^2\left(B^{(L)}\right)\right)$

Step 3: For layer $l = L, \ldots, 2$, backward propagate the errors by:

- $E^{(l-1)} := \left(\left(W^{(l)}\right)^\intercal E^{(l)}\right) \odot \left(1 - \tanh^2\left(B^{(l-1)}\right)\right)$

Step 4: For layer $l = 1, \ldots, L$, update weights and thresholds using the computed errors:

- $W^{(l)} := W^{(l)} + \eta E^{(l)} \left(X^{(l)}\right)^\intercal$
- $T^{(l)} := T^{(l)} - \eta E^{(l)}$

3.2. Code

```
import argparse
import numpy as np
import matplotlib.pyplot as plt

# Load training data
train_set = np.genfromtxt("training_set.csv", delimiter=",").T
n_train = train_set.shape[1]

# Load validation data
val_set = np.genfromtxt("validation_set.csv", delimiter=",").T
```

```
n_val = val_set.shape[1]
X_val = val_set[:-1, :]
Y_val = val_set[-1:, :]
input_dim = X_val.shape[0]

# Pre-defined constants
accepted_error_rate = 0.12
eta = 0.02
n_epochs = 1000
n_layers = 3
n_dims = [input_dim, 8, 16, 1] # number of dimensions in all layers (including input layer)

# Initialize weights and thresholds for all layers
W = [np.random.normal(0, 1, size=(n_dims[l], n_dims[l-1])) for l in range(1, n_layers+1)]
T = [np.zeros(shape=(n_dims[l], 1)) for l in range (1, n_layers+1)]

def tanh_derivative(values):
    return (1 - np.tanh(values)**2)

def run_perceptron(inputs, targets, is_training=False):
    X = [inputs] # store inputs for all layers
    B = []       # store local fields for all layers
    O = []       # store outputs for all layers
    E = []       # store errors for all layers

    # Forward propagate
    for l in range(n_layers):
        B.append(W[l] @ X[-1] - T[l])
        O.append(np.tanh(B[-1]))

        if l != n_layers - 1:
            X.append(O[-1])

    # Backward propagate only during training (assuming batch size of 1)
    if is_training and inputs.shape[1] == 1:
        E.insert(0, ((targets - O[-1]) * tanh_derivative(B[-1])))

        for l in range(-1, -n_layers, -1):
            E.insert(0, (W[l].T @ E[0] * tanh_derivative(B[l-1])))

        for l in range(n_layers):
            W[l] += eta * E[l] @ X[l].T
            T[l] -= eta * E[l]

    return O[-1]

def main(args):
    np.random.seed(args.seed)

    for epoch in range(1, n_epochs + 1):
        np.random.shuffle(train_set.T)
        X_train = train_set[:-1, :]
        Y_train = train_set[-1:, :]
```

```
        # Training
        for mu in range(n_train):
            run_perceptron(X_train[:,mu,None], Y_train[:,mu,None], is_training=True)

        # Validating
        Y_pred = np.sign(run_perceptron(X_val, Y_val, is_training=False))
        error_rate = (Y_pred != Y_val).sum() / n_val

        # Print summary and check stopping criteria
        print("[Epoch " + str(epoch) + "]", "Error rate:", error_rate)
        if error_rate <= accepted_error_rate: break

    for l in range(n_layers):
        np.savetxt("{}/w{}.csv".format(args.outdir, l+1), W[l], delimiter=",")
        np.savetxt("{}/t{}.csv".format(args.outdir, l+1), T[l], delimiter=",")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Experiment with 2-layer perceptron")
    parser.add_argument("--outdir", "-o", type=str, default=".", help="Out directory")
    parser.add_argument("--seed", "-s", type=int, default=None, help="Seed for random generator")
    main(parser.parse_args())
```

3.3. Results

```
[Epoch 1] Error rate: 0.1442
[Epoch 2] Error rate: 0.1364
[Epoch 3] Error rate: 0.1378
[Epoch 4] Error rate: 0.135
[Epoch 5] Error rate: 0.1338
[Epoch 6] Error rate: 0.1294
[Epoch 7] Error rate: 0.1302
[Epoch 8] Error rate: 0.129
[Epoch 9] Error rate: 0.1292
[Epoch 10] Error rate: 0.1398
[Epoch 11] Error rate: 0.1262
[Epoch 12] Error rate: 0.1284
[Epoch 13] Error rate: 0.1322
[Epoch 14] Error rate: 0.1338
[Epoch 15] Error rate: 0.1264
[Epoch 16] Error rate: 0.1326
[Epoch 17] Error rate: 0.1296
[Epoch 18] Error rate: 0.1288
[Epoch 19] Error rate: 0.128
[Epoch 20] Error rate: 0.1258
[Epoch 21] Error rate: 0.1318
[Epoch 22] Error rate: 0.1222
[Epoch 23] Error rate: 0.1222
[Epoch 24] Error rate: 0.1206
[Epoch 25] Error rate: 0.1188
```