URKUND HW2

Jens Ifver

September 2021

How many linearly separable 3-dimensional Boolean functions are there?

To answer this question we derive the number of separable functions for the functions mapping exactly k ($k = 0, 1, ..., 8$) patterns to 1 separately. We simplify the calculations by acknowledging that there is a symmetry between the functions for $k = 0$ and $k = 8$, $k = 1$ and $k = 7$, $k = 2$ and $k = 6$, $k = 3$ and $k = 5$, in the sense that they are each others negation. More specifically, their patterns are inversions of each others. So in this case, if a Boolean function is linearly separable, then so is the negation of that function. This means that the functions that have this symmetry have the same number of linearly separable functions. Hence we only have to derive the linearly separable functions for $k = 0, 1, 2, 3, 4$.

Before diving in to each case we also note that for two dimensions, XOR and XNOR are the linearly inseparable functions. This also holds for three dimensions, meaning that if either of the six sides of the "cube"(made up of all possible patterns) display any of these two problems, the function is linearly inseparable. In three dimensions we also have the case when patterns in opposite corners have target 1, which also implies inseparability for $k \leq 4$. Examples of these linearly inseparable functions are illustrated in Figure 1.

Taking the above notes into account we find that there is only one symmetry (here meaning cubes that can be mapped onto each other by rotation and/or reflection) for $k = 0, 1, 2, 3$ and two symmetries for $k = 4$ that are linearly separable. These symmetries are illustrated in Figure 2. Now we simply count the number of functions for each symmetry.

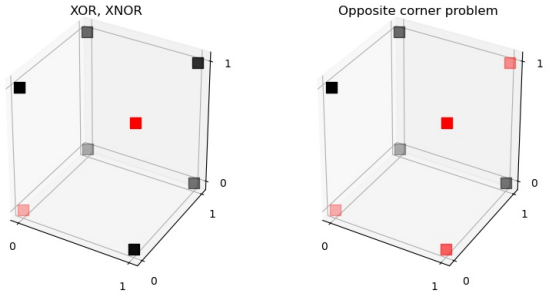

Figure 1: *Illustration of functions that are inseparable.*

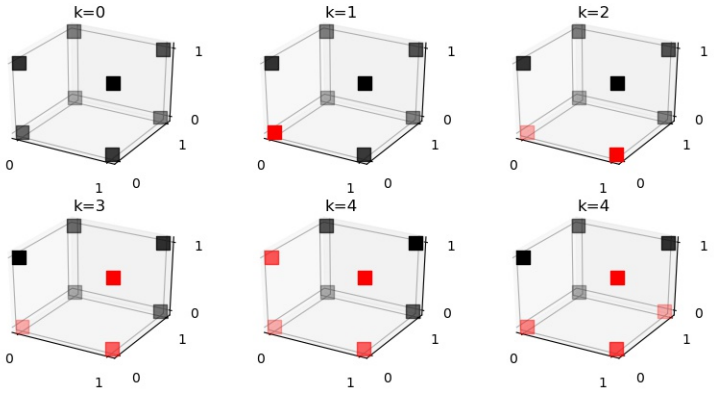

Figure 2: *Linearly separable symmetries for k = 0, 1, 2, 3, 4. Red squares represents target 1 and black squares represents target 0.*

In the table below we have the number of linearly separable functions for each of the symmetries represented in Figure 2. To obtain the total number (for $k = 0, 1, 2, 3, 4, 5, 6, 7, 8$) of linearly separable functions we calculate

$$(1 + 8 + 12 + 24) * 2 + 6 + 8 = 104.$$

k	0	1	2	3	4(1)	4(2)
Linearly separable functions	1	8	12	24	6	8

Code HW3 part2

Jens Ifver

September 2021

```
    import pandas as pd
import numpy as np

### FUNCTIONS ###

# Normalization
def normalize(data, data_val):
    outdata = data.copy()
    outdata_val = data_val.copy()
    m1 = np.mean(data[:,0])
    m2 = np.mean(data[:,1])
    outdata[:,0] = (outdata[:,0] - m1) / np.std(data[:,0])
    outdata[:,1] = (outdata[:,1] - m2) / np.std(data[:,1])
    outdata_val[:,0] = (outdata_val[:,0] - m1) / np.std(data[:,0])
    outdata_val[:,1] = (outdata_val[:,1] - m2) / np.std(data[:,1])
    return outdata, outdata_val

# Initialization of weights and thresholds
def init_weights_W(M1):
    return np.random.normal(loc = 0, scale =np.sqrt(1/M1), size = M1)
def init_weights_w(M1):
    return np.random.normal(loc = 0, scale = np.sqrt(0.5), size = (M1,2))
def init_thresholds(M1):
    return np.zeros(M1), 0

# Local fields
def calc_local_field_b_mat(x_mat, w_mat, theta_vec):
    M1 = w_mat.shape[0]
    p = x_mat.shape[0]
    b_mat = np.zeros((M1, p))
    for mu in range(p):
        for jt in range(M1):
            b_mat[jt, mu] = (-theta_vec[jt] + w_mat[jt,0]*x_mat[mu,0] +
            w_mat[jt,1]*x_mat[mu,1])
    return b_mat
```

```
def calc_local_field_B(V_mat, W_mat, theta):
    M1, p = V_mat.shape
    B_vec = np.zeros(p)
    for mu in range(p):
        tmp = 0
        for jt in range(M1):
            tmp += W_mat[jt] * V_mat[jt,mu]
        B_vec[mu] = tmp - theta
    return B_vec

# Neuron calucations
def calc_neurons(b_mat):
    return np.tanh(b_mat)
def calc_output(B_vec):
    return np.tanh(B_vec)

# Back propagation step 1
def deriv_tanh(bi):
    return 1 - np.tanh(bi)**2
def calc_Delta(t_vec,O_vec, B_vec):
    p = t_vec.shape[0]
    Delta_vec = np.zeros(p)
    for mu in range(p):
        Delta_vec[mu] =  (t_vec[mu] - O_vec[mu]) * deriv_tanh(B_vec[mu])
    return Delta_vec
def calc_change_W_vec(eta, Delta_vec, V_mat):
    return eta * np.matmul(V_mat, Delta_vec)
def calc_change_Theta(Delta_vec, eta):
    return -eta * sum(Delta_vec)

# Back propagation step 2
def calc_delta_mat(Delta_vec, W_mat, b_mat):
    M1, p = b_mat.shape
    delta_mat = np.zeros((M1,p))
    for mu in range(p):
        for jt in range(M1):
            delta_mat[jt,mu] = Delta_vec[mu] * W_mat[jt] * deriv_tanh(b_mat[jt,mu])
    return delta_mat
def calc_change_w_mat(eta, delta_mat, x_mat):
    return eta * np.matmul(delta_mat, x_mat)
def calc_change_theta_vec(eta, delta_mat):
    return -eta * np.sum(delta_mat, axis = 1)

# Update weights
def update_weights(W_mat, w_mat, eta, change_W, change_w):
    return np.add(W_mat, change_W), np.add(w_mat, change_w)
```

```
def update_thresholds(theta_vec, Theta, change_theta_vec, change_Theta):
    return np.add(theta_vec, change_theta_vec), Theta + change_Theta

# Classification error
def calc_C(t_vec, O_vec):
    tmp = np.sum( np.absolute( np.subtract( np.sign(O_vec), t_vec ) ) ) /(2*t_vec.shape[0
    return tmp
def class_error_validation(x_val, t_val, w_mat, W_mat, theta_vec, Theta):
        b_mat = calc_local_field_b_mat(x_val, w_mat, theta_vec)
        V_mat = calc_neurons(b_mat)
        B_vec = calc_local_field_B(V_mat, W_mat, Theta)
        O_vec = calc_output(B_vec)
        return calc_C(t_val, O_vec), O_vec

# Running the network
def run_training(M1, x_mat, t_vec, x_val, t_val, epochs, eta, train_frac):
    # Initialize
    W_mat = init_weights_W(M1)
    w_mat = init_weights_w(M1)
    theta_vec, Theta = init_thresholds(M1)
    val_C = np.zeros(epochs)
    best_val_C = np.inf
    # Training loop
    for ep in range(epochs):
        for it in range(x_mat.shape[0]):
            indices = np.random.choice(range(x_mat.shape[0]), size = int (x_mat.shape[0]*
            x_it = np.copy(x_mat[indices,:])
            t_it = np.copy(t_vec[indices])
            b_mat = calc_local_field_b_mat(x_it, w_mat, theta_vec)
            V_mat = calc_neurons(b_mat)
            B_vec = calc_local_field_B(V_mat, W_mat, Theta)
            O_vec = calc_output(B_vec)
            Delta_vec = calc_Delta(t_it, O_vec, B_vec)
            change_W_vec = calc_change_W_vec(eta, Delta_vec, V_mat)
            change_Theta = calc_change_Theta(Delta_vec, eta)
            delta_mat = calc_delta_mat(Delta_vec, W_mat, b_mat)
            change_w_mat = calc_change_w_mat(eta, delta_mat, x_it)
            change_theta_vec = calc_change_theta_vec(eta,delta_mat)
            W_mat, w_mat = update_weights(W_mat, w_mat, eta, change_W_vec, change_w_mat)
            theta_vec, Theta = update_thresholds(theta_vec, Theta,
            change_theta_vec, change_Theta)

        # Classification error validation set
        val_C[ep], output = class_error_validation(x_val,t_val, w_mat, W_mat,
            theta_vec, Theta)
        if (val_C[ep] < best_val_C):
```

```
            best_val_C = val_C[ep]
            best_w = w_mat
            best_W = W_mat
            best_t = theta_vec
            best_T = Theta
    return val_C, best_w, best_W, best_t, best_T

### Calculations ###

# Load data
colnames = ['x1', 'x2', 't']
train = pd.read_csv('training_set.csv', names = colnames)
validation = pd.read_csv('validation_set.csv', names = colnames)
train_norm, validation_norm = normalize(train.to_numpy(), validation.to_numpy())
x_mat = train_norm[:,:2]
t_vec = train_norm[:,2]
x_val = validation_norm[:,:2]
t_val = validation_norm[:,2]

# Run training
M1 = 50
eta = 0.005
epochs = 50
train_frac = 1/500
val_C, best_w, best_W, best_t, best_T = run_training(M1, x_mat, t_vec, x_val, t_val, epo
print(val_C)

# Checking output
best_val, output = class_error_validation(x_val, t_val, best_w, best_W, best_t, best_T)
print(best_val)

# Plotting result
import matplotlib.pyplot as plt
plt.scatter(x_val[:,0], x_val[:,1],s = 10, c = np.sign(output))
plt.scatter(x_val[:,0], x_val[:,1],s = 10, c = t_val)

# Save as CSV
pd.DataFrame(best_w).to_csv('w1.csv', header = False, index = False)
pd.DataFrame(best_W).to_csv('w2.csv', header = False, index = False)
pd.DataFrame(best_t).to_csv('t1.csv', header = False, index = False)
best_T.tofile('t2.csv',sep=',')
```

HW3: Restricted Boltzman Machine

Jens Ifver

September 2021

The Restricted Boltzman Machine (RBM) we use in this task consist of three visible neurons V_j and M hidden neurons h_i where $M = 1, 2, 4, 8$. We use the CD-k algorithm to train the RBM to learn the distribution of the XOR-dataset where four patterns are given 1/4 probability and the rest zero. We use the CD-k algorithm to adjust the weights and thresholds so that the Boltzman distribution $P_B(x)$ approximate the input distribution $P_{data}(x)$. To estimate $P_B(x)$ we iteratively feed random 3-dimensional Boolean patterns and use the weights and thresholds obtained in the CD-k algorithm to apply neuron updates and then keep count of the number of times the state converge to every pattern. In theory $M = 2^N/2 - 1$ hidden neurons are enough for $P_B(x)$ to get very close to $P_{data}(x)$. To measure how similar the two distributions are we use the Kullback-Leibler divergence D_{KL}. In Figure 1 we see that D_{KL} decreases as the number of hidden neurons increase. This is due to the fact that the hidden neurons encode correlations between the visible neurons and each of the hidden neurons can describe one of the binary patterns. Therefor it makes sense that for $M = 1, 2$ the distributions have a higher divergence and for $M = 4, 8$ there is a lower divergence.

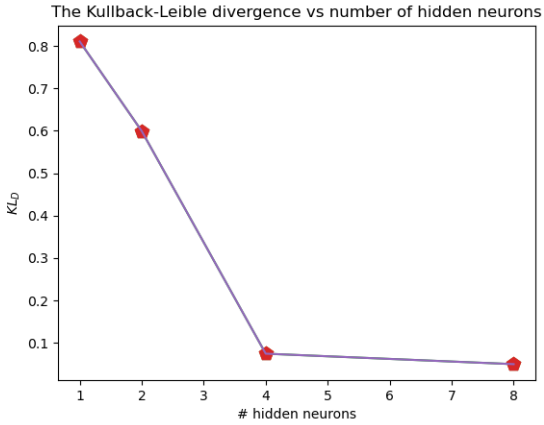

Figure 1: The figure shows the computed Kullback-Leibler divergence as a function of number of hidden neurons M in an RBM. Results were obtained using k=100, $\eta = 0.1$ and 200 trials for each M. The red dots represent the Kullback-Leibler divergence computed for $M = 1, 2, 4, 8$.

Code HW3 part 3

Jens Ifver

September 2021

```
    import numpy as np

### FUNCTIONS ###

def initialize_weights_thresholds(M):
    W_mat = np.random.normal(loc = 0, scale = 1, size = (M,3))
    theta_v = np.zeros(3)
    theta_h = np.zeros(M)
    return W_mat, theta_v, theta_h
def choose_pattern(X, n):
    p0 = np.random.randint(0,n)
    return np.copy(X[p0,:])
def update_hidden_n(W_mat, theta, state):
    bx = np.subtract(np.matmul(W_mat,state), theta)
    return np.copy(bx)
def update_visable_n(W_mat, theta, state):
    bx = np.subtract( np.matmul(np.transpose(state), W_mat), theta)
    return np.copy(bx)
def prob(bx):
    return 1/(1 + np.exp(-2*bx))
def mccullock_pitts(bx):
    px = prob(bx)
    rx = np.random.rand(1)
    if ( rx < px): return 1
    else: return -1
def change_w(b_vec_0, b_vec_k, visable_0, visable_k, eta):
    hard_brack = np.subtract( np.outer( np.tanh(b_vec_0), visable_0 ),
        np.outer( np.tanh(b_vec_k), visable_k ) )
    return eta * hard_brack
def change_theta_v(eta, visable_0, visable_k):
    return -eta * np.subtract( visable_0, visable_k )
def change_theta_h(eta, b_vec_0, b_vec_k):
    return -eta * np.subtract( np.tanh(b_vec_0), np.tanh(b_vec_k) )

# CD-k algorithm
```

```
def run_CD_k(epochs, patterns, M, k, eta, batch_size):
    W_mat, theta_v, theta_h = initialize_weights_thresholds(M)
    for ep in range(epochs):
        delta_w = np.zeros((M,3))
        delta_theta_v = np.zeros(3)
        delta_theta_h = np.zeros(M)
        for mu in range(batch_size):
            # Choose pattern
            choosen_pattern = choose_pattern(patterns, 4)
            visable_0 = np.copy(choosen_pattern)
            visable_n = np.copy(visable_0)

            # Update hidden neurons
            b_v_t = np.zeros(3)
            b_h_0 = np.zeros(M)
            b_h_t = np.zeros(M)
            b_h_0 = np.add( b_h_0,update_hidden_n(W_mat, theta_h, visable_0))
            hidden_n = np.zeros(M)
            for it in range(M):
                hidden_n[it] = mccullock_pitts(b_h_0[it])
            # Time loop
            for tx in range(k):
                # Update visable neurons
                b_v_t = update_visable_n(W_mat, theta_v, hidden_n)
                for jx in range(3):
                    visable_n[jx] = mccullock_pitts(b_v_t[jx])
                # Update hidden neurons
                b_h_t =  update_hidden_n(W_mat, theta_h,  visable_n)
                for ix in range(M):
                    hidden_n[ix] = mccullock_pitts(b_h_t[ix])

            # Compute weight and threshold increments
            delta_w = np.add(delta_w, change_w(b_h_0, b_h_t, visable_0, visable_n, eta))
            delta_theta_v = np.add(delta_theta_v , change_theta_v(eta, visable_0,visable
            delta_theta_h = np.add(delta_theta_h , change_theta_h(eta, b_h_0, b_h_t))
        # Adjust weights and thresholds
        W_mat = np.add(delta_w, W_mat)
        theta_v = np.add(theta_v, delta_theta_v)
        theta_h = np.add(theta_h, delta_theta_h)
    return W_mat, theta_v, theta_h

# Feed patterns
def feed_patterns(Xi, N_o, N_i, W_mat, theta_h, theta_v):
    M, v = W_mat.shape
    P_B = np.zeros(8)
    for ix in range(N_o):
```

```
        mu = choose_pattern(Xi, 8)
        visable_n = mu
        # Update hidden neurons
        b_h_0 = np.zeros(M)
        b_h_t = np.zeros(M)
        b_v_t = np.zeros(v)
        b_h_0 = update_hidden_n(W_mat, theta_h, visable_n)
        hidden_n = np.zeros(M)
        for hx in range(M):
            hidden_n[hx] = mccullock_pitts(b_h_0[hx])
        # Time loop
        for tx in range(N_i):
            # Update visable neurons
            b_v_t = update_visable_n(W_mat, theta_v, hidden_n)
            for jx in range(3):
                visable_n[jx] = mccullock_pitts(b_v_t[jx])
            # Update hidden neurons
            b_h_t = update_hidden_n(W_mat, theta_h, visable_n)
            for kx in range(M):
                hidden_n[kx] = mccullock_pitts(b_h_t[kx])
            for lx in range(8):
                if (np.sum (visable_n ==  Xi[lx,:]) == 3):
                    P_B[lx] += 1/(N_o * N_i)
    return P_B

# KL divergence
def KL(P_B):
    sm = 0
    P_data = np.array([0.25, 0.25, 0.25, 0.25])
    for it in range(4):
            sm += P_data[it] * np.log(P_data[it]/P_B[it])
    return sm
def KL_M(P_B_mat):
    KL_vec = np.zeros(4)
    for it in range(4):
        KL_vec[it] = KL(P_B_mat[it,:])
    return KL_vec

# Main function
def run_for_all_M(X_all):
    batch_size = np.array([1, 2, 30, 30])
    time = np.array([25, 50, 75, 100])
    Ms = np.array([1, 2, 4, 8])
    epochs = 200
    eta = 0.1
    k = 100
```

```
    N_outer = 1000
    N_inner = 1000
    P_B_mat = np.zeros((4,8))
    for m in range(4):
        W_mat, theta_v, theta_h = run_CD_k(epochs, X_all[:4,:], Ms[m], k, eta,batch_size
        P_B_mat[m,:] = feed_patterns(X_all, N_outer, N_inner, W_mat, theta_h, theta_v)
        print(time[m])
    KL_vec = KL_M(P_B_mat)
    return KL_vec

### CALCULATIONS ###

# Declare patterns
x1 = np.array([-1, -1, -1])
x2 = np.array([1, -1, 1])
x3 = np.array([-1, 1, 1])
x4 = np.array([1, 1, -1])
X = np.array([x1,x2,x3,x4])
x5 = np.array([1, 1, 1])
x6 = np.array([-1, 1, -1])
x7 = np.array([1, -1, -1])
x8 = np.array([-1, -1, 1])
X_all = np.array([x1, x2, x3, x4, x5, x6, x7, x8])

# Run for all M
KL_out1 = run_for_all_M(X_all)

# Plot result
import matplotlib.pyplot as plt
M = [1, 2, 4, 8]
plt.plot(M, KL_out1, 'p', markersize = 10)
plt.plot(M, KL_out1)
plt.xlabel('# hidden neurons')
plt.ylabel('$KL_D$')
plt.title('The Kullback-Leible divergence vs number of hidden neurons')
```