Sample solution for Exercise 2

1) Hand-Crafted Network

1. **logical OR**: We use the step function Θ as an activation function:

$$\Theta(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Then we simply have $f(z) = \Theta(Wz + b)$, with $1 \times D$ weight matrix W, whereby $W_{1,j} = 1$ for all $j = 1, ..., D$ and a scalar bias $b = -1$

2. **masked logical OR**: We define the $1 \times D$ weight matrix through $W_{1,j} = c_j$ where c is our binary mask, and a 1×1 bias $b = -1$, and use the step function:

$$g(z; c) = \Theta(Wz + b) \tag{2}$$

If the mask c is equal to the input z at any dimension j and $z_j = 1$ the threshold is reached and $g(z; c) = 1$ otherwise $g(z; c) = 0$. It's sketch is equivalent to the 'logical or' but with different weights.

3. **perfect match**: We define the $1 \times D$ weight matrix through $W_{1,j} = 2c_j - 1$, and a 1×1 bias $b = -\sum_{j=1}^{D} c_j$, and again use the step function as in (2). How this works can be seen by rewriting the function as follows (simplifies to W, b in (2)):

$$h(z; c) = \Theta\left(\sum_j (c_j - z_j)(1 - 2c_j)\right) \tag{3}$$

The $(c_j - z_j)$ term is ± 1 if $c_j \neq z_j$. The second term is ∓ 1, so the product of the terms contributes -1 to the sum whenever $c_j \neq z_j$, and 0 otherwise. So the sum can only be zero if every $c_j = z_j$. See figure 1.

- For this training set we adjust the dimension to $M = 3$. We use three linear decision boundaries for this task of mapping the X_i to the 8 corners of a three-dimensional hypercube. The decision boundary (i) can be expressed through

$$h^{(i)}(x) = \Theta(W^{(i)}x + b^{(i)}) \tag{4}$$

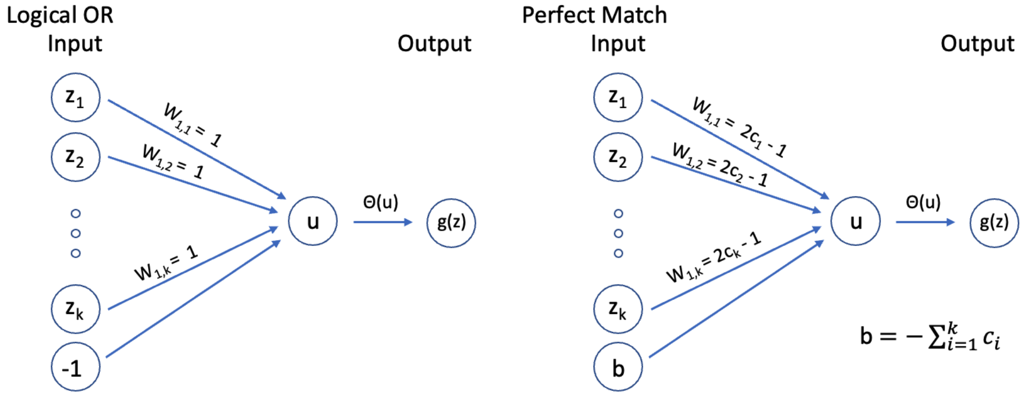


Figure 1: One-layer network sketches for task 1.1 and 1.3

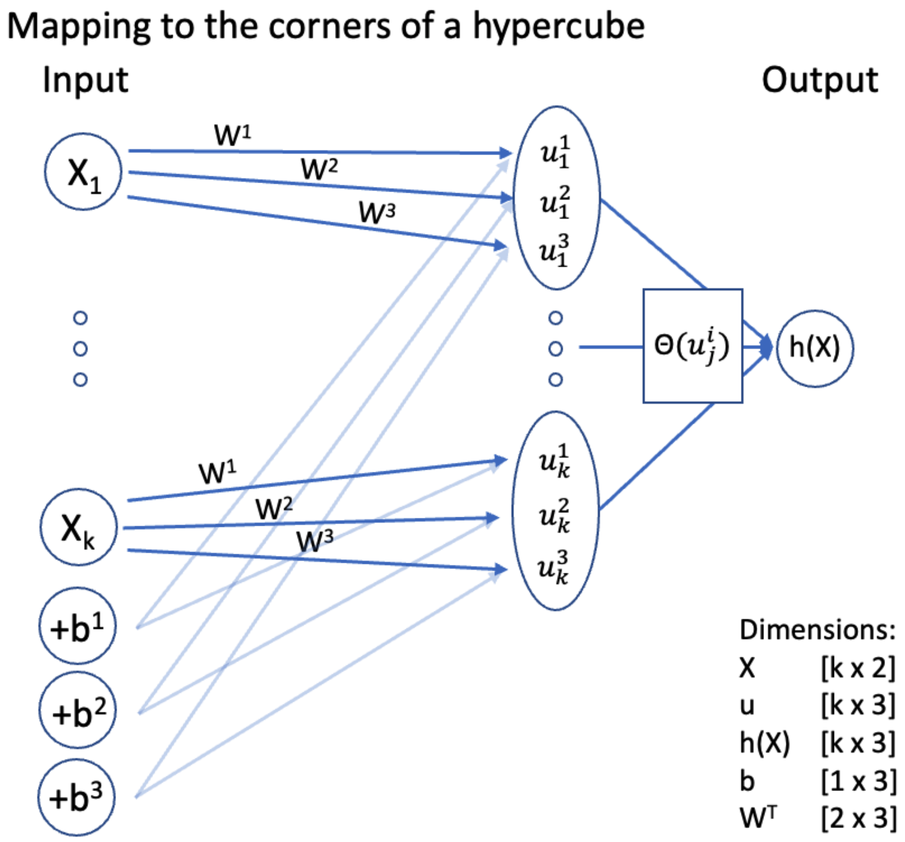


Figure 2: One-layer network sketch for mapping onto the corners of a hypercube.

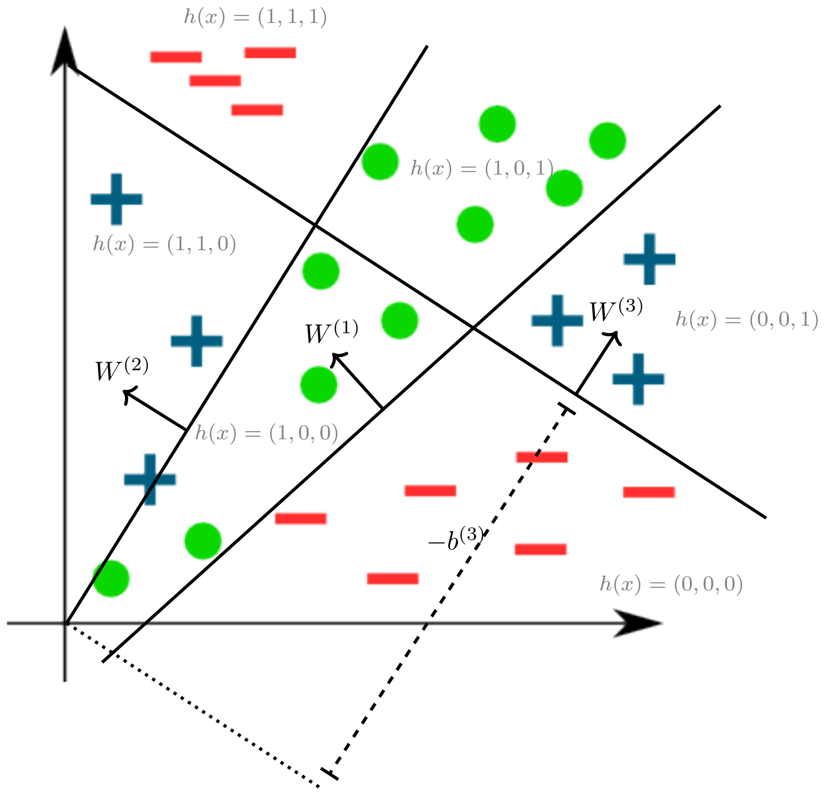
where the 1×2 weight matrix contains the normal vector of the boundary, and the bias term is the negative distance from the origin, in direction of the normal (see figures with decision boundaries).

For the three boundaries in total, we therefore have a 3×2 weight matrix $W = [W^{(1)}, W^{(2)}, W^{(2)}]$, and a 3-vector $b = (b^{(1)}, b^{(2)}, b^{(3)})$. The output of

$$h(x) = \Theta(Wx + b) \tag{5}$$

will therefore be $\in \{0, 1\}^3$ and thus on a corner of a three-dimensional hypercube, indicating on which side of each boundary the point lies (see figure). Note that the step function is applied pairwise here. See figure 2.

- Drawn decision boundaries into the figure:



- For this task, we use a 3-layer network $f\Big(g\big(h(x)\big)\Big)$, with the one-layer networks from above.
  - $h(x)$ maps the coordinates x to a $z \in 0, 1^3$, e.g. the corners of a hypercube.
  - $g(z)$ maps all 8 possible $z \in \{0, 1\}^3$ to a $\{0, 1\}^8$ one-hot-encoded vector y, which indicates which z was the input, i.e. what partition of x space the point lies in. Each row of the constructed weight matrix and bias corresponds to weights and bias of a perfect match layer (1.3) with respective target c^j where $j \in [1, \dots, 8]$ for each corner in the hypercube. Thus, we need 8 instances of network 1.3. The weight matrix and bias then look like (in binary sorted order):

$$W = \begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & 1 \\ -1 & 1 & -1 \\ & \vdots & \\ 1 & 1 & 1 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 0 \\ -1 \\ -1 \\ \vdots \\ -3 \end{pmatrix} \tag{6}$$

  - $f(y)$ combines all individual regions belonging to one class through the 'masked logical `or`' operation, equivalent to network 1.2. We need one instance of network 1.2 for every class in the dataset. This operation maps the one-hot encoded corners of the hypercube to its respective class. Each row of the weight matrix then corresponds to one mask c^j of network 1.2. These masks need to be designed such that they assign each corner of the hypercube to one of the three classes. For the shown example (green circle: class 1, blue plus: class 2, red minus: class 3), the weight matrix then will be

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{7}$$

For the full sketch see figure 3.

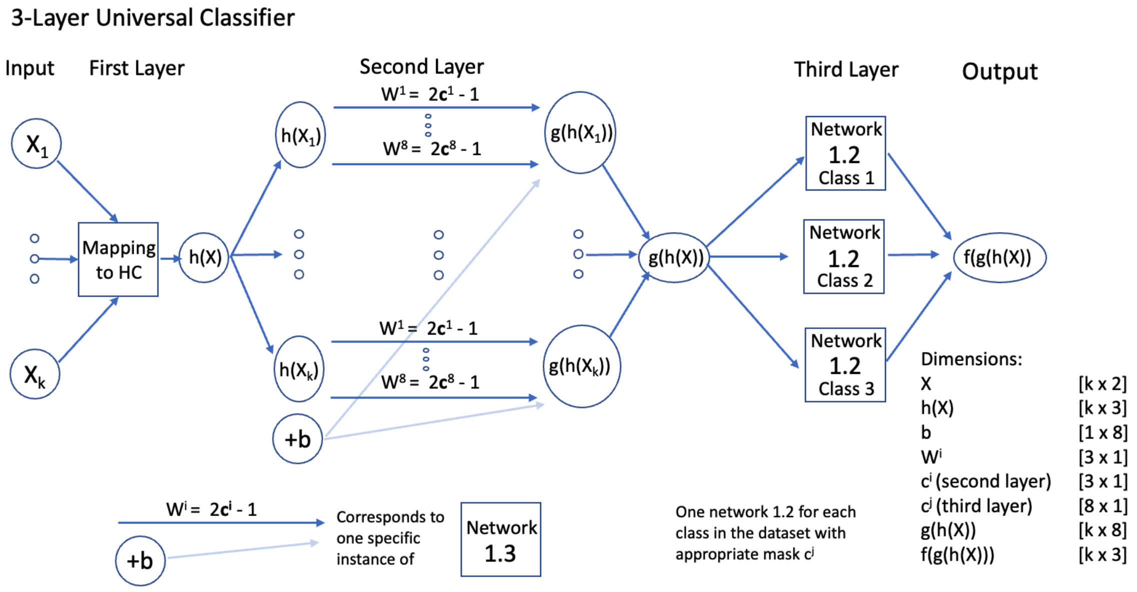


Figure 3: Sketch of 3-layer Universal Classifier

- We can describe a simple algorithm that generalizes this to arbitrary input dimensions: In d-dimensional space, with a set $\mathcal{X} = \{x^{(i)}\}$ of N datapoints and K classes:

  (1) Find a d-dimensional linear decision boundary so that the points on one side only come from a single class (guaranteed to be possible, because we can just split off a single point in the worst case)

  (2) If the points on the other side are also all from a single class, we are done. Otherwise, go back to step (1) for these points.

  This will always terminate in $\leq N$ iterations, because the number of remaining points is reduced by at least 1 each iteration.

- In practice, this approach of a zero training loss classifier will of course lead to overfitting, and poor generalization. For the 2D case shown, we at least see some generalization, but e.g. for image classification it is sure to fail completely.

2) Linear Activation Function

A linear activation function takes the form $\phi_l(Z_l) = s * Z_l + c$ where s and c are scalars. The additive constant can for every layer be absorbed into the bias. Thus, applying the activation function can only mean scaling with some scalar s. We now construct a one-layer-network that simply computes

$$Z_L = AZ_0 \quad \text{with} \quad A \equiv s^L \prod_{l=1}^{L} B_l \tag{8}$$

We show that Z_L is the same as from the original network. This can be seen by recursively repeating the following operation:

$$\begin{aligned}
Z_1 &= sB_1Z_0 && \equiv A_1Z_0 && (9)\\
Z_2 &= sB_2Z_1 = sB_2A_1Z_0 && \equiv A_2Z_0 && (10)\\
Z_3 &= sB_3Z_2 = sB_3A_2Z_0 && \equiv A_3Z_0 && (11)\\
&\vdots && \vdots &&\\
Z_L &= sB_LA_{L-1}Z_0 && = AZ_0 && (12)
\end{aligned}$$

□

If we choose the weight matrix of the one-layer network as A it will be equivalent to any corresponding L-layer network with linear activation functions.

3) Application

File `network_solution.py`:

```
import numpy as np
from sklearn import datasets


####################################


class ReLULayer(object):
    def forward(self, input):
        # remember the input for later backpropagation
        self.input = input
        # return the ReLU of the input
        # ===== Solution =====
        relu = input.copy()
        relu[relu<0] = 0
        # ===== End =====
        return relu

    def backward(self, upstream_gradient):
        # compute the derivative of ReLU from upstream_gradient and the stored
            input
        # ===== Solution =====
        downstream_gradient = upstream_gradient.copy()
        downstream_gradient[self.input<=0] = 0
        # ===== End =====
        return downstream_gradient

    def update(self, learning_rate):
        pass # ReLU is parameter-free


####################################

class OutputLayer(object):
    def __init__(self, n_classes):
        self.n_classes = n_classes

    def forward(self, input):
        # remember the input for later backpropagation
        self.input = input
        # return the softmax of the input
        # ===== Solution =====
        # substract maximum for numerical stability. As you can find out
        # quickly: It has no impact on the results of the softmax
        exp = np.exp(input - np.max(input))
        softmax = exp / np.sum(exp, axis=1, keepdims=True)
        # ===== End =====
        return softmax

    def backward(self, predicted_posteriors, true_labels):
        # return the loss derivative with respect to the stored inputs
        # (use cross-entropy loss and the chain rule for softmax,
        #  as derived in the lecture)
        # ===== Solution =====
        downstream_gradient = predicted_posteriors.copy()
        downstream_gradient[np.arange(len(true_labels)), true_labels] -= 1
        downstream_gradient /= len(true_labels)
        # ===== End =====
        return downstream_gradient

    def update(self, learning_rate):
        pass # softmax is parameter-free


####################################

class LinearLayer(object):
    def __init__(self, n_inputs, n_outputs):
        self.n_inputs  = n_inputs
        self.n_outputs = n_outputs
        # randomly initialize weights and intercepts
        # ===== Solution =====
        self.B = np.random.normal(0.0, 2.0/n_inputs, size=(n_inputs, n_outputs))
        self.b = np.random.normal(0.0, 2.0/n_inputs, size=(n_outputs,))
        # ===== End =====

    def forward(self, input):
        # remember the input for later backpropagation
        self.input = input
        # compute the scalar product of input and weights
        # (these are the preactivations for the subsequent non-linear layer)
        # ===== Solution =====
        preactivations = np.dot(input, self.B) + self.b
        # ===== End =====
        return preactivations

    def backward(self, upstream_gradient):
        # compute the derivative of the weights from
        # upstream_gradient and the stored input
        # ====== Solution =====
        self.grad_B = np.dot(self.input.T, upstream_gradient)
        self.grad_b = np.sum(upstream_gradient, axis=0)
        # compute the downstream gradient to be passed to the preceding layer
        downstream_gradient = np.dot(upstream_gradient, self.B.T)
        # ====== End =====
        return downstream_gradient

    def update(self, learning_rate):
        # update the weights by batch gradient descent
        self.B = self.B - learning_rate * self.grad_B
        self.b = self.b - learning_rate * self.grad_b

####################################

class MLP(object):
    def __init__(self, n_features, layer_sizes):
        # constuct a multi-layer perceptron
        # with ReLU activation in the hidden layers and softmax output
        # (i.e. it predicts the posterior probability of a classification
            problem)
        #
        # n_features: number of inputs
        # len(layer_size): number of layers
        # layer_size[k]: number of neurons in layer k
        # (specifically: layer_sizes[-1] is the number of classes)
        self.n_layers = len(layer_sizes)
        self.layers   = []

        # create interior layers
        n_in = n_features
        for n_out in layer_sizes[:-1]:
            self.layers.append(LinearLayer(n_in, n_out))
            self.layers.append(ReLULayer())
            n_in = n_out

        # create output layer
        n_out = layer_sizes[-1]
        self.layers.append(LinearLayer(n_in, n_out))
        self.layers.append(OutputLayer(n_out))

    def forward(self, X):
        # X is a mini-batch of instances
        batch_size = X.shape[0]
        # flatten the other dimensions of X (in case instances are images)
        X = X.reshape(batch_size, -1)

        # compute the forward pass
        # (implicitly stores internal activations for later backpropagation)
        result = X
        for layer in self.layers:
            result = layer.forward(result)
        return result

    def backward(self, predicted_posteriors, true_classes):
        # perform backpropagation w.r.t. the prediction for the latest mini-
            batch X
        # ===== Solution =====
        gradient = self.layers[-1].backward(predicted_posteriors, true_classes)
        for layer in self.layers[-2::-1]:
            gradient = layer.backward(gradient)
        # ===== End =====

    def update(self, X, Y, learning_rate):
        posteriors = self.forward(X)
        self.backward(posteriors, Y)
        for layer in self.layers:
            layer.update(learning_rate)

    def train(self, x, y, n_epochs, batch_size, learning_rate):
        N = len(x)
        n_batches = N // batch_size
        for i in range(n_epochs):
            # print("Epoch", i)
            # reorder data for every epoch
            # (i.e. sample mini-batches without replacement)
            permutation = np.random.permutation(N)

            for batch in range(n_batches):
                # create mini-batch
                start = batch * batch_size
                x_batch = x[permutation[start:start+batch_size]]
                y_batch = y[permutation[start:start+batch_size]]

                # perform one forward and backward pass and update network
                    parameters
                self.update(x_batch, y_batch, learning_rate)


##################################


if __name__=="__main__":

    # set training/test set size
    N = 2000

    # create training and test data
    X_train, Y_train = datasets.make_moons(N, noise=0.05)
    X_test,  Y_test  = datasets.make_moons(N, noise=0.05)
    n_features = 2
    n_classes  = 2

    # standardize features to be in [-1, 1]
    offset  = X_train.min(axis=0)
    scaling = X_train.max(axis=0) - offset
    X_train = ((X_train - offset) / scaling - 0.5) * 2.0
    X_test  = ((X_test  - offset) / scaling - 0.5) * 2.0

    # set hyperparameters (play with these!)
    layer_sizes = [2, 2, n_classes]
```

```
# layer_sizes = [3, 3, n_classes]
# layer_sizes = [5, 5, n_classes]
# layer_sizes = [30, 30, n_classes]
n_epochs = 500
batch_size = 200
learning_rate = 0.05

# create network
network = MLP(n_features, layer_sizes)

# train
network.train(X_train, Y_train, n_epochs, batch_size, learning_rate)

# test
predicted_posteriors = network.forward(X_test)
# determine class predictions from posteriors by winner-takes-all rule

# ===== Solution =====
predicted_classes = np.argmax(predicted_posteriors, axis=1)
# compute and output the error rate of predicted_classes
wrong_predictions = np.sum(predicted_classes!=Y_test)
error_rate = wrong_predictions / N
# ===== End =====

print("wrong predictions:", wrong_predictions, "/", N, "error rate:",
    error_rate)
```