

Exercise 2

Deadline: 21.05.2019, 2:00 pm

This exercise focuses on basics of neural networks.

Regulations

Please create two files for your solution: **network.pdf** for your answers to the tasks (e.g. a scan of your hand-written solution) and **network.py** for the completed neural network code. Zip both files along with the answers to task 1 into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_exercise02.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise02.zip`

and upload it to Moodle before the given deadline. We will give zero points if your zip-file does not conform to the naming convention.

1 Comment on your solution to exercise 1

Study the sample solution `ex01_sample_solution.html` on Moodle and use it to comment on your own solution to this exercise. Specifically, copy your original notebook `logistic-regression.ipynb` to `logistic-regression-commented.ipynb` and export it to `logistic-regression-commented.html` in the end. Insert comments as markdown cells starting with

```
<span style="color:green;font-weight:bold">Comment</span>
```

in order to clearly distinguish your comments from other cell types. The point of these comments is that you identify your errors and bugs yourselves, so that you learn from your mistakes. In addition, the tutor will have an easier time distinguishing between the initial mistake and consequential errors caused by the first one and will only deduct points for the former. If you fail to hand in comments, the tutor is not required to make this distinction and will deduct points for all errors alike.

A Neural Playground (0 Points)

Have a look at <http://playground.tensorflow.org/> and play around with it for a while to get some feeling for neural networks.

This is not an official exercise, so you don't need to hand in anything.

2 Classification Capacity (15 Points)

In this exercise we want to proof that neural networks can classify an arbitrary training set with zero training error. First, we will construct single layer networks that fulfill specific tasks. Second, we combine them to a multilayer network that can classify a given dataset "flawlessly". For each task draw a sketch of the network, specify which activation functions are used and how the weights must be chosen.

2.1 Simple Networks (10 Points)

First, design one-layer neural networks that perform the following tasks:

1. logical OR operation on a binary input vector $z \in \{0, 1\}^m$

$$\text{i.e. } z \rightarrow f(z) = \begin{cases} 0 & \forall i : z_i = 0 \\ 1 & \text{otherwise} \end{cases}$$

2. for an arbitrary but fixed binary vector $c \in \{0, 1\}^m$ map the input vector $z \in \{0, 1\}^m$ to

$$z \rightarrow f(z) = \begin{cases} 1 & z = c \\ 0 & \text{otherwise} \end{cases}$$

3. for the dataset X, Y displayed in Figure 1 (with feature vectors X_i and associated classes $Y_i \in \{\text{red minus, blue plus, green circle}\}$) map every X_i onto the corners of a hypercube $\{0, 1\}^m$ such that each corner contains only one class (you can map one class to multiple corners, but not multiple classes onto one corner). The dimension m of the hypercube is not a priori fixed and may be adjusted to fit the dataset. Draw the decision boundaries of your network in Figure 1 (you do not need to specify the precise equations of these boundaries) and indicate for each region to which hypercube corner it will be mapped. How can this be generalized to arbitrary many input dimensions and arbitrary (non degenerate) class distributions?

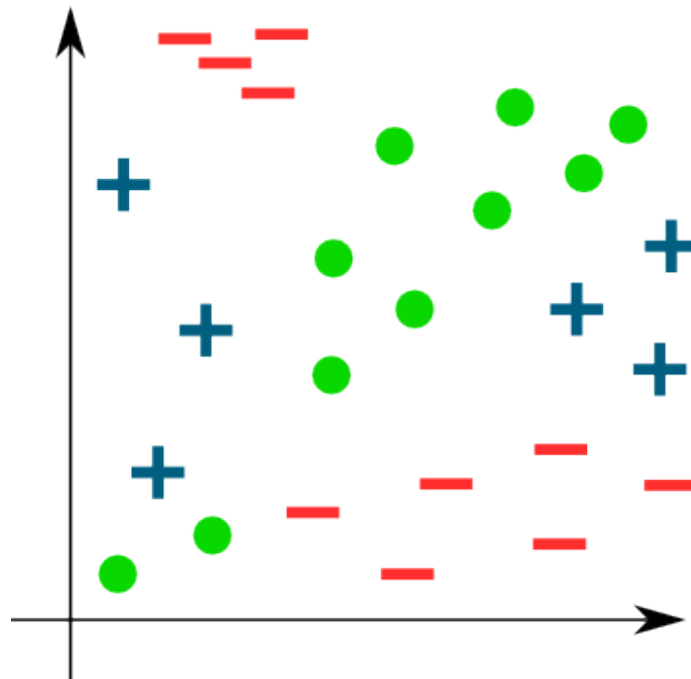


Abbildung 1: Example dataset with three classes (red minus, blue plus, green circle). Sketch the decision boundaries of your network here and draw the matching labeled hypercube.

2.2 3-layer Universal Classifier (5 Points)

Combine the networks from above into a universal classifier which classifies an arbitrary training set with zero training error. Draw a sketch of the network and explain in a few sentences how it works. Do you see any problems with this zero training loss classifier?

3 Linear Activation Function (5 Points)

For a feed forward network the output of each layer l is calculated iteratively by

$$Z_0 = \begin{bmatrix} 1 \\ \mathbf{X}^T \end{bmatrix} \quad (1)$$

$$\tilde{Z}_l = B_l Z_{l-1} \quad (2)$$

$$Z_l = \phi_l(\tilde{Z}_l) \quad (3)$$

Prove that if ϕ_l is a linear mapping then any network (with depth $L > 1$) is equivalent to a 1-layer neuronal network.

4 Application (20 Points)

In this exercise we want to implement a simple Multi-Layer Perceptron classifier using `numpy`. The python code below defines an MLP class with ReLU activations in the hidden layers and softmax output (you can download it as `network.py` from Moodle). Complete the code (i.e. forward and backward passes through the network and performance validation) at the places marked with

`... # your code here`

Explain your implementation within comments. Compare the validation errors for the networks

```
MLP(n_features, [2, 2, n_classes])
MLP(n_features, [3, 3, n_classes])
MLP(n_features, [5, 5, n_classes])
MLP(n_features, [30, 30, n_classes])
```

File `network.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
        relu = ... # your code here
        return relu

    def backward(self, upstream_gradient):
        # compute the derivative of ReLU from upstream_gradient and the stored input
        downstream_gradient = ... # your code here
        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
```

```

        softmax = ... # your code here
    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)
    downstream_gradient = ... # your code here
    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
        self.B = np.random.normal(...) # your code here
        self.b = np.random.normal(...) # your code here

    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)
        preactivations = ... # your code here
        return preactivations

    def backward(self, upstream_gradient):
        # compute the derivative of the weights from
        # upstream_gradient and the stored input
        self.grad_b = ... # your code here
        self.grad_B = ... # your code here
        # compute the downstream gradient to be passed to the preceding layer
        downstream_gradient = ... # your code here
        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):
        # construct 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 (linear + ReLU)
        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 last linear layer + 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
    ... # your code here

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 = [5, 5, n_classes]
    n_epochs = 5

```

```
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
predicted_classes = ... # your code here
# compute and output the error rate of predicted_classes
error_rate = ... # your code here
print("error rate:", error_rate)
```