Exercise 2

Deadline: 22.05.2023, 4:00 pm

This exercise focuses on basics of neural networks.

Regulations

Please create three files for your solution: `network.pdf` for your answers to the non-programming tasks (e.g. a scan of your hand-written solution), a jupyter notebook `network.ipynb` for the completed neural network code and the same file exported as a `.html` file (`network.html`). Zip all files into a single archive `ex02.zip` and upload this file to MaMPF before the given deadline.
Moreover, please set your **Anzeigename/display name** and **Name in Uebungsgruppen/name in tutorials** in MaMPF to your real name, which should be identical to your name in `muesli` and make sure you **join the submission** of your team via the invitation code before the submission deadline. Check out `https://mampf.blog/handing-in-homework-assignments` for instructions.

A Neural Playground

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.

1 Hand-Crafted Network (15 Points)

In this exercise we want to construct neural networks that classify an arbitrary training set with zero training error. As preparation, design single neurons (i.e. specify their weights, bias, and activation function) for the following tasks:

1. logical OR: map a binary input vector $z \in \{0,1\}^D$ to
$$z \to f(z) = \begin{cases} 1 & \exists j \text{ such that } z_j = 1 \\ 0 & \text{otherwise, i.e. } z = 0 \end{cases}$$
2. masked logical OR: for an arbitrary but fixed binary vector $c \in \{0,1\}^D$ map the input vector $z \in \{0,1\}^D$ to
$$z \to g(z;c) = \begin{cases} 1 & \exists j \text{ such that } c_j = 1 \text{ and } z_j = 1 \\ 0 & \text{otherwise, i.e. } z = 0 \text{ or } z_j = 1 \text{ occurs only at indices where } c_j = 0 \end{cases}$$
3. perfect match: for an arbitrary but fixed binary vector $c \in \{0,1\}^D$ map the input vector $z \in \{0,1\}^D$ to
$$z \to h(z;c) = \begin{cases} 1 & z = c \\ 0 & \text{otherwise} \end{cases}$$

Now use these building blocks to design a three-layer network for the dataset X, Y in Figure 1 (with 2-dimensional feature vectors X_i and class labels $Y_i \in$ {"red minus", "blue plus", "green circle"}). The first layer maps every X_i onto one **corner** of a hypercube $\{0,1\}^M$ such that each corner contains only points of the same class (you can map one class to multiple corners, but not multiple

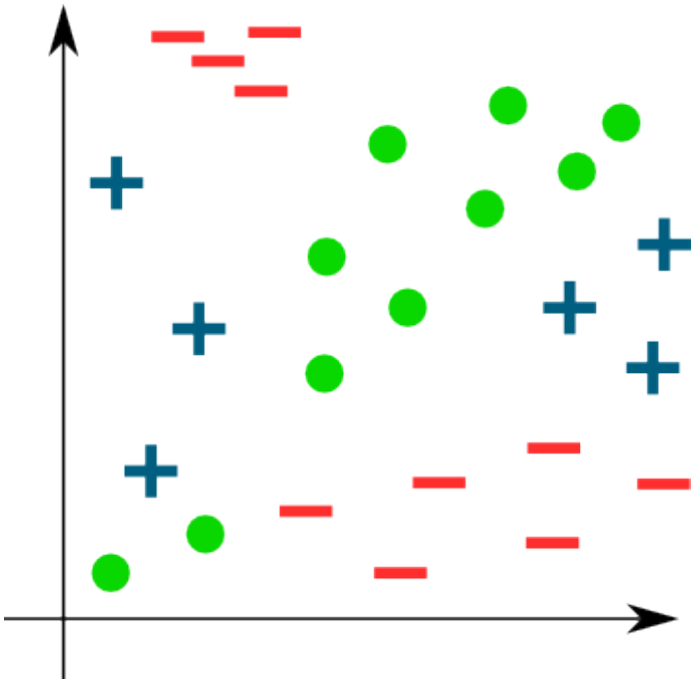

Figure 1: Example dataset with three classes (red minus, blue plus, green circle). Sketch the decision boundaries of your network here and indicate hypercube corners as described in the text.

classes onto one corner). The dimension M of the hypercube is a hyperparameter that you have to adjust for the given training set.

Copy figure 1 to your solution and draw the first layer's decision boundaries and their normal vectors. You do not need to specify precise equations for these boundaries. The normal vectors should point in the direction of positive pre-activation. Indicate for each decision region to which hypercube corner the corresponding feature points will be mapped.

Now specify the equations for the network's second and third (=output) layer to produce a one-hot encoding of the class labels with zero training error. A one-hot encoding for three classes is a binary vector that takes values $(1,0,0), (0,1,0), (0,0,1)$ for classes 1, 2, 3 respectively.

Sketch the resulting network and describe in words how it could be generalized to arbitrary many input dimensions and arbitrary (non degenerate) label distributions. Do you see potential problems with this zero training loss classifier?

2 Linear Activation Function (5 Points)

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

$$Z_0 = X \tag{1}$$
$$\tilde{Z}_l = Z_{l-1} \cdot B_l + b_l \quad \text{(multiply with weights and add bias vector)} \tag{2}$$
$$Z_l = \phi_l(\tilde{Z}_l) \quad \text{(apply the activation function element-wise to pre-activations)} \tag{3}$$

Where X and the Z_l and b_l are understood as row vectors.

Prove that if ϕ_l is the identity function then any network (with depth $L > 1$) is equivalent to a 1-layer neural network.

3 Programming a neural network (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` at `https://tinyurl.com/HD-AML-network-py` or via the exercise's external link on MAMPF). Complete the code (i.e. forward and backward passes through the network and performance validation) at the places marked with

```
... # your code here
```

and hand-in the completed file. Explain your implementation within comments. Compare the validation errors on the dataset generated by the `make_moons()` function for the following four 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])
```

Hint: In the `backward(..)` functions, you are supposed to implement back-propagation. The `upstream_gradient` at some layer l is the derivative received from layer $l+1$. (Recall that we are propagating gradients from back to front!) Mathematically, it is a row vector defined as

$$g_l^{(\text{upstream})} = \frac{\partial \mathcal{L}oss}{\partial Z_l}$$

where Z_l is the input of layer $l+1$, which is also the output of layer l. The function `backward(..)` in layer l should apply the chain rule to the `upstream_gradient` to compute the `downstream_gradient`, which is passed on to layer $l-1$.

$$g_l^{(\text{downstream})} = g_l^{(\text{upstream})} \cdot \frac{\partial Z_l}{\partial Z_{l-1}}$$

If layer l has trainable parameters, the `backward(..)` function must also compute the derivate with respect to them, e.g.

$$\frac{\partial \mathcal{L}oss}{\partial B_l} = \left(\frac{\partial Z_l}{\partial B_l}\right)^T \cdot g_l^{(\text{upstream})}$$

Parameter derivatives should be stored, so that the gradient step can be executed in the `update(..)` function later on.

Code stub '`network.py`' to be completed at '`... # your code here`':

```
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):
        # 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 (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)
```