Technische Universität Hamburg
Institut für Mathematik
Lehrstuhl Numerische Mathematik

Jens-Peter M. Zemke
Jonas Grams

Mathematics of Neural Networks
winter semester 2021/2022
exercise sheet 1, solutions

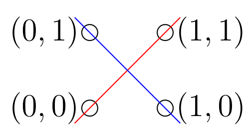
Exercise 1: ((1+1+1)+2+1 points)

a) Construct feedforward neural networks based on ReLU activations that implement
   (i) logical `AND` ($\wedge$) of two logical variables,
   (ii) logical `OR` ($\vee$) of two logical variables,
   (iii) logical `XOR` of two logical variables.
   Here, $0 \in \mathbb{R}$ encodes `False` and $1 \in \mathbb{R}$ encodes `True`, e.g., $0 \wedge 0 = 0$, $1 \vee 0 = 1$.

b) Can you find three feedforward neural networks based on ReLU activations that implement these three logical binary operators but with only *one* hidden layer such that the first weight matrix and first bias vector is the *same* for all three?

c) Prove that there does not exist a feedforward neural network based on ReLU activations that implements `XOR` with zero hidden layers.

Solution: We solve part b) and thus part a). Every binary logical function is uniquely defined by the values it takes on the four pairs $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. We take the view of a neural network as stacked SVM presented in the first lecture. The blue and the red line in the following picture can be used to distinguish between the four pairs.



We use a neural network with one hidden layer comprising $4 = 2^2$ neurons that identifies uniquely the pair in the preceeding picture by setting the weight matrix $\mathbf{W}_1 \in \mathbb{R}^{4\times 2}$ row-wise to the two normal vectors (with alternate signs, non-scaled) associated with the blue and red line and setting the bias $\mathbf{b}_1$ to the negative shift associated to these,

$$\mathbf{W}_1 := \begin{pmatrix} -1 & -1 \\ -1 & 1 \\ 1 & -1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 := -\mathbf{W}_1 \begin{pmatrix} .5 \\ .5 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}.$$

We feed in all possible four tuples $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ simultaneously as a matrix

$$\mathbf{X} := \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

We define $\mathbf{e}$ to be the vector comprising ones, here first in $\mathbb{R}^4$. In the hidden layer we obtain

$$\begin{aligned}\mathbf{Z}_1 := \mathbf{W}_1\mathbf{X} + \mathbf{b}_1\mathbf{e}^\mathsf{T} &= \begin{pmatrix} -1 & -1 \\ -1 & 1 \\ 1 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & -1 & -1 & -2 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix},\end{aligned}$$

thus after activation with ReLU we obtain the full-rank matrix

$$\mathbf{A}_2 := \mathsf{ReLU}(\mathbf{Z}_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{I}_4$$

instead of the rank-two matrix $\mathbf{Z}_1$. At this point it becomes obvious why we need non-linear activation functions. Now we only need to find a weight matrix $\mathbf{W}_2 \in \mathbb{R}^{4\times 3}$ and a bias $\mathbf{b}_2 \in \mathbb{R}^3$ such that

$$\mathbf{W}_2\mathbf{A}_2 + \mathbf{b}_2\mathbf{e}^\mathsf{T} = \mathbf{Y} := \begin{array}{c} \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} (\texttt{AND}) \\ (\texttt{OR}) \\ (\texttt{XOR}) \end{array}$$

holds true, since the activation with ReLU does not change any entry as all entries are already nonnegative. This is easily achieved using

$$\mathbf{W}_2 := \mathbf{Y} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \quad \mathbf{b}_2 := \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \mathbf{o}_3.$$

Thus the neural network defined by $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$ and $\mathbf{b}_2$ implements all three logical functions at once, the neural networks

$$\mathbf{W}_1, \quad \mathbf{b}_1, \quad \mathbf{e}_i^\mathsf{T}\mathbf{W}_2, \quad \mathbf{e}_i^\mathsf{T}\mathbf{b}_2, \quad i = 1, 2, 3$$

implement `AND` ($i = 1$), `OR` ($i = 2$), and `XOR` ($i = 3$). The first hidden layer can be used to implement *any* logical bivariate function; the first weights and biases are *universal*.

Part c) follows easily when considering the network as a SVM. As we can not separate the two points $(0, 0)$ and $(1, 1)$ from $(0, 1)$ and $(1, 0)$ by a single line, we need at least two of them.

Exercise 2: (2+2 points) Read the Wikipedia entry

> https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Arnold_representation_theorem

and interpret the Kolmogorov–Arnold representation theorem (a) and the variant by Sprecher (b) each as a special type of feedforward neural network.

Solution: The Kolmogorov-Arnold representation theorem states that any multivariate continuous function $f : [0, 1]^n \subset \mathbb{R}^n \to \mathbb{R}$ can be represented exactly with $2n + 1$ continuous univariate functions Φ_q that depend on f and a matrix of $(2n+1)n$ universal univariate functions $\phi_{q,p}$, $q = 0, \dots, 2n$, $p = 1, \dots, n$ that do not depend on f in the form

$$f(\mathbf{x}) = f(x_1, \dots, x_n) = \sum_{q=0}^{2n} \Phi_q\Big(\sum_{p=1}^{n} \phi_{q,p}(x_p).\Big)$$

Sprecher modified the representation theorem above by proving the existance of a single continuous function Φ that replaces the $2n+1$ functions in the Kolmogorov-Arnold representation theorem if the inner functions $\phi_{q,p}$ are replaced by a single real univariate and monotone increasing function $\phi : [0,1] \to [0,1]$ with appropriate scales and a shifts. He proved that there exists $\eta \in \mathbb{Q}$ and $\lambda_1, \dots, \lambda_p \in \mathbb{R}$ such that

$$f(\mathbf{x}) = f(x_1, \dots, x_n) = \sum_{q=0}^{2n} \Phi\Big(\sum_{p=1}^{n} \lambda_p \phi(x_p + \eta q) + q.\Big)$$

Apart from the fact that Sprecher's original proof has faults that have been corrected later, see the paper by Jürgen Braun and Michael Griebel in „Constructive Approximation“,

https://doi.org/10.1007/s00365-009-9054-2,

both can be interpreted as a special form of feedforward neural network with one hidden layer comprising $2n+1$ neurons. In the original form we replace the affine linear mapping $\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$ and activation $\mathbf{a}_2 = a(\mathbf{z}_1)$ of the first stage by the nonlinear mapping $\mathbf{z} : \mathbb{R}^n \to \mathbb{R}^{2n+1}$ given componentwise by

$$z_q = \sum_{p=1}^{n} \phi_{q,p}(x_p)$$

followed by activation with the distinct activation functions Φ_q. Thus, the only multivariate function used is the sum. In the second stage this is followed by the trivial affine linear mapping defined by $\mathbf{W}_2 = \mathbf{e}^\mathsf{T}$ and $\mathbf{b}_2 = 0$ with $\mathbf{e} \in \mathbb{R}^{2n+1}$ as vector of all ones. The same form of network can be used in Sprecher's variant.

Another variant is given by expressing the representation theorem using two hidden layers, the first comprising $(2n+1)n$ neurons and the second comprising $2n+1$ neurons. We consider Sprecher's variant. In the first stage we map $\mathbb{R}^n$ to $\mathbb{R}^{(2n+1)n}$ by the affine linear map $\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$ given by the components

$$x_p \mapsto x_p + \eta q,$$

consider the case $n = 2$ and $(2n+1)n = 10$:

$$\mathbf{W}_1\mathbf{x} + \mathbf{b}_1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ \eta \\ 2\eta \\ 3\eta \\ 4\eta \\ 0 \\ \eta \\ 2\eta \\ 3\eta \\ 4\eta \end{pmatrix} = \begin{pmatrix} x_1 \\ x_1 + \eta \\ x_1 + 2\eta \\ x_1 + 3\eta \\ x_1 + 4\eta \\ x_2 \\ x_2 + \eta \\ x_2 + 2\eta \\ x_2 + 3\eta \\ x_2 + 4\eta \end{pmatrix} = \mathbf{z}_1$$

We then apply the activation function ϕ, $\mathbf{a}_2 = \phi(\mathbf{z}_1)$. In the second stage we use the affine linear mapping $\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_2 + \mathbf{b}_2$ defined by

$$\phi(x_p + \eta q) \mapsto \sum_{p=1}^{n} \lambda_p \phi(x_p + \eta q) + q =: z^{(2)}_{q+1},$$

e.g., for $n = 2$:

$$\mathbf{W}_2\mathbf{a}_2 + \mathbf{b}_2 = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & 0 & \lambda_2 & 0 & 0 & 0 & 0 \\ 0 & \lambda_1 & 0 & 0 & 0 & 0 & \lambda_2 & 0 & 0 & 0 \\ 0 & 0 & \lambda_1 & 0 & 0 & 0 & 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & 0 & \lambda_1 & 0 & 0 & 0 & 0 & \lambda_2 & 0 \\ 0 & 0 & 0 & 0 & \lambda_1 & 0 & 0 & 0 & 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \phi(x_1) \\ \phi(x_1+\eta) \\ \phi(x_1+2\eta) \\ \phi(x_1+3\eta) \\ \phi(x_1+4\eta) \\ \phi(x_2) \\ \phi(x_2+\eta) \\ \phi(x_2+2\eta) \\ \phi(x_2+3\eta) \\ \phi(x_2+4\eta) \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \mathbf{z}_2.$$

This is again followed by an activation, here with Φ, $\mathbf{a}_3 = \Phi(\mathbf{z}_2)$. The final stage is given by $\mathbf{W}_3 = \mathbf{e}^\mathsf{T}$ for $\mathbf{e} \in \mathbb{R}^{2n+1}$ comprising all ones and $\mathbf{b} = 0$.

Exercise 3: (1+1+2 points)

a) Read the Wikipedia entry on the Basic Linear Algebra Subprograms (`BLAS`),

   https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

b) Find out which variant of the `BLAS` your installed variant of NumPy uses by invoking

```
import numpy as np
np.__config__.show()
```

c) Implement a Python script that computes for given $n, k \in \mathbb{N}$ the product of two matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times k}$ using NumPy and
   (i) nk scalar products (`BLAS LEVEL 1`),
   (ii) k matrix-vector products (`BLAS LEVEL 2`),
   (iii) one matrix-matrix product (`BLAS LEVEL 3`).

   How fast is each variant, say, e.g., for $n = 10.000$ and $k = 100$? You might have to reduce these numbers depending on your computer.

Solution: See the listing `numpy_blas.py`:

```
import numpy as np
import time

np.__config__.show()

n = 10000
k = 100

A = np.random.random( (n,n) )
B = np.random.random( (n,k) )
C = np.zeros( (n,k) )

```

```
t = time.time()
for j in range(k):
    for i in range(n):
        C[i,j] = A[[i],:]@B[:,[j]]

level1_time = time.time() - t
print("BLAS LEVEL 1:", level1_time, "Sekunden")

t = time.time()
for j in range(k):
    C[:,j] = A@B[:,j]

level2_time = time.time() - t
print("BLAS LEVEL 2:", level2_time, "Sekunden")

t = time.time()
C = A@B
level3_time = time.time() - t
print("BLAS LEVEL 3:", level3_time, "Sekunden")

print("speedup 1 vs. 2:", level1_time/level2_time)
print("speedup 2 vs. 3:", level2_time/level3_time)
print("speedup 1 vs. 3:", level1_time/level3_time)
```

On my machine the output gives a speedup of 10.77 when using `BLAS LEVEL 2` in place of `BLAS LEVEL 1` and a speedup of 7.6 when using `BLAS LEVEL 3` in place of `BLAS LEVEL 2`, resulting in a speedup of 81.93 when replacing `BLAS LEVEL 1` by `BLAS LEVEL 3`.

Exercise 4: ((2+2+2)+2+2 points)

a) Implement a feedforward neural network in Python.

  (i) Implement the `ReLU` activation function as class in the provided `activation.py` script which has a method `evaluate(self, x)` that performs the application of the `ReLU` activation function

  (ii) Implement a class `DenseLayer` in the given `layers.py` script which
  - is initialized with integers specifiying the number of inputs and outputs, and an activation function (which is `ReLU` by default)
  - has the attributes `W` and `b` for the weight matrix and the bias of this layer
  - has the method `evaluate(self, a)` that performs the evaluation on the
  - has the methods `set_weights` and `set_bias` for setting the weight matrix and the bias of a layer.

  (iii) Implement a class `SequentialNet` in the provided `networks.py` script which
  - has the attributes `layers` that stores all layers of the network and an integer `no` indicating the current number of outputs
  - is initialized by an integer indicating the number of inputs and an (optional) list of layers
  - has the method `evaluate(self,x)` that performs the feed forward with input `x`.

b) Write a Python script that tests the feed forward process for some given neural net. You may use your results from exercise 1.

c) (optional) Add a method `draw()` to the class `SequentialNet` that draws the neural network using circles for neurons and lines between them for the connecting weights.

Solution: See the following listings for a possible implementation:

```python
import numpy as np

# ReLU family activation functions

class ReLU():

    def __init__(self):

        self.name = 'ReLU'

    def evaluate(self, x):

        self.data = x
        return x.clip(min = 0)
```

```python
import numpy as np

from activations import ReLU

class DenseLayer:

    def __init__(self,
                 ni, # Number of inputs
                 no, # Number of outputs
                 afun = None # Activationfunction for the layer
    ):

        self.ni   = ni
        self.no   = no

        # Set default activation function to ReLU
        if afun is None:
            self.afun = ReLU()
        else:
            self.afun = afun

        self.W    = np.zeros((no, ni))
        self.b    = np.zeros((no,1))

    def evaluate(self, a):

        z = self.W @ a + self.b
        return self.afun.evaluate(z)

    def set_weights(self, W):

        assert(W.shape == (self.no, self.ni))
        self.W = W

    def set_bias(self, b):

        assert(b.size == self.no)
```

```
        self.b = b
```

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

from activations import ReLU
from layers      import DenseLayer

class SequentialNet:

    def __init__(self, n, layers=None):

        if layers is None:
            self.layers = []
            self.no = n
        else:
            self.layers = layers
            self.no = layers[-1].no



    def evaluate(self, x):

        for layer in self.layers:

            x = layer.evaluate(x)

        return x

    def draw(self):
        num_layers = len(self.layers)
        max_neurons_per_layer = np.amax(self.layers.no)
        neurons_layers = [layer.no for layer in self.layers]
        dist = 2*max(1,max_neurons_per_layer/num_layers)
        y_shift = self.layers/2-.5
        rad = .3

        fig = plt.figure(frameon=False)
        ax = fig.add_axes([0, 0, 1, 1])
        ax.axis('off')

        for i in range(num_layers):
            if i == 0:
                for j in range(self.layers[0].ni):
                    circle = plt.Circle((i*dist, j-y_shift[i]),
                                         radius=rad, fill=False)
            else:
                for j in range(neurons_layers[i-1]):
                    circle = plt.Circle((i*dist, j-y_shift[i]),
                                         radius=rad, fill=False)
                ax.add_patch(circle)

        for i in range(num_layers-1):
            if i == 0:
                for j in range(self.layers[0].ni):
                    for k in range(self.layers[0].no):
```

```
                        angle = np.atan(float(j-k+y_shift[i+1]-y_shift[i]) / dist)
                        x_adjust = rad * np.cos(angle)
                        y_adjust = rad * np.sin(angle)
                        line = plt.Line2D((i*dist+x_adjust,
                                           (i+1)*dist-x_adjust),
                                          (j-y_shift[i]-y_adjust,
                                           k-y_shift[i+1]+y_adjust),
                                           lw=2 / np.sqrt(self.layers[i]
                                                    + self.layers[i+1]),
                                           color='b')
                        ax.add_line(line)
            else:
                for j in range(neurons_layers[i]):
                    for k in range(self.layers[i+1]):
                        angle = np.atan(float(j-k+y_shift[i+1]-y_shift[i]) / dist)
                        x_adjust = rad * np.cos(angle)
                        y_adjust = rad * np.sin(angle)
                        line = plt.Line2D((i*dist+x_adjust,
                                           (i+1)*dist-x_adjust),
                                          (j-y_shift[i]-y_adjust,
                                           k-y_shift[i+1]+y_adjust),
                                           lw=2 / np.sqrt(self.layers[i]
                                                    + self.layers[i+1]),
                                           color='b')
                        ax.add_line(line)

        ax.axis('scaled')
```

As an example of a calling sequence we used the neural net constructed in exercise 1b), where we explicitely set the weights and biases in the script `testffn.py` of the following listing. This is possible since these are public attributes. We could also have used private attributes but in that case we would have needed public methods `set_weight()` and `set_bias()`.

```
import numpy as np

from networks import SequentialNet
from layers   import DenseLayer

layers = [DenseLayer(2, 4),
          DenseLayer(4, 3)]
netz = SequentialNet(2, layers)

netz.layers[0].set_weights(np.array([[-1, -1],
                                     [-1,  1],
                                     [ 1, -1],
                                     [ 1,  1]]))

netz.layers[0].set_bias(np.array([[ 1],
                                  [ 0],
                                  [ 0],
                                  [-1]]))

netz.layers[1].set_weights(np.array([[0, 0, 0, 1],
                                     [0, 1, 1, 1],
                                     [0, 1, 1, 0]]))
```

```

X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])

Y = netz.evaluate(X)

print("input:\n", X)
print("output:\n", Y)
```

The net correctly computes the three binary logical functions. The plot of the net in this example can be seen in the following picture.