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 2, solutions

Exercise 1: (2 points) Check $\mathsf{SoftAbs}_k(x)$ for monotonicity, symmetry, and asymptotic behavior and calculate the derivative.

Solution: Symmetry is shown by

$$\mathsf{SoftAbs}_k(x) = \frac{kx^2}{1+|kx|} = \frac{k(-x)^2}{1+|k(-x)|} = \mathsf{SoftAbs}_k(-x).$$

We further see that for positive x

$$\mathsf{SoftAbs}_k(x) = \frac{kx^2}{1+kx} = \frac{1}{k(kx+1)} - \frac{1}{k} + x$$

and for negative x

$$\mathsf{SoftAbs}_k(x) = \frac{kx^2}{1-kx} = -\frac{1}{k(kx-1)} - \frac{1}{k} - x.$$

Thus

$$a_+(x) = -\frac{1}{k} + x$$

is an oblique asymptote to $\mathsf{SoftAbs}_k(x)$ when x tends to $+\infty$ and

$$a_-(x) = -\frac{1}{k} - x$$

is an oblique asymptote to $\mathsf{SoftAbs}_k(x)$ when x tends to $-\infty$.
We calculate the derivative:

$$\begin{aligned}\frac{d}{dx}\mathsf{SoftAbs}_k(x) &= \frac{d}{dx}\left(x \cdot \mathsf{SoftSign}_k(x)\right) = \frac{kx}{1+|kx|} + \frac{kx}{(1+|kx|)^2} = \frac{(1+|kx|)\cdot kx + kx}{(1+|kx|)^2} \\ &= \frac{(2+|kx|)}{(1+|kx|)^2} \cdot kx\end{aligned}$$

Hence $\mathsf{SoftAbs}_k(x)$ is strictly increasing for $x > 0$ and strictly decreasing for $x < 0$.

Exercise 2: (3 points) Prove that any continous function

$$f : [a, b] \to [0, \infty)$$

can be approximated arbitrarily well by a neural network $N : \mathbb{R} \to \mathbb{R}$ with a single hidden layer comprising n neurons for some $n \in \mathbb{N}$ based on ReLU activations.

Hint: Can you express the linear interpolant through the n equidistant points

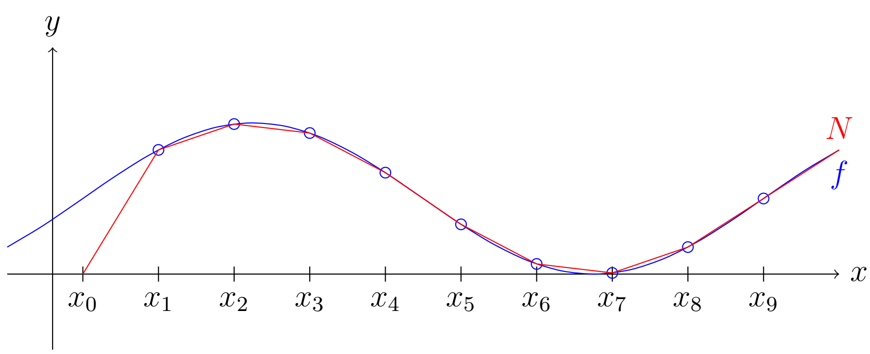
$$a = x_1, x_2, \ldots, x_n = b \quad \text{and function values} \quad f_i = f(x_i), \quad i = 1, \ldots, n$$

using linear combinations of ReLU functions $\max(0, x + b_i)$ with biases $b_1 = -(a - h), b_2 = -a, b_3 = -(a + h), b_4 = -(a + 2h), \ldots, b_n = -(b - h)$, where $h = x_2 - x_1$?

Solution: First let $n \in \mathbb{N}$, and $a = x_1, x_2, \ldots, x_n = b$ be equidistant points. These points devide the interval $[a, b]$ into $n - 1$ equal intervals $[x_i, x_{i+1}]$, $\quad i = 1, \ldots n - 1$, each of length $h = \frac{1}{n-1}$. To construct a neural network with one hidden layer comprising n neurons based on `ReLU` activations, we have to find the weights and biases for the network. Note that for $m, b \in \mathbb{R}$ we get

$$\mathsf{ReLU}(mx + b) = \max(0, mx + b) = m \max(0, x + b/m) = m\, \mathsf{ReLU}(x + b/m).$$

Thus we can set the weights of the hidden layer to 1, and only need to determine the weights for the output layer.
Now consider the following figure:



As described in the hint, we pick the biases of the hidden layer as

$$b_1 = -(a - h), b2 = -x_1 = -a, b_3 = -x_2 = -(a + h), \ldots, b_n = -x_{n-1} = -(a + (n - 2)h),$$

to possibly change the gradient after each intervall $[x_i, x_i + 1]$. Thus prior to $-b_1$ all ReLU functions $\max(0, x + b_i)$ are zero. Between $-b_1 = a - h$ and $-b_2 = a = x_1$ only one of these is non-zero, namely

$$\max(0, x + b_1) = x + b_1.$$

To achieve a possibly non-zero value $f(a) = f(x_1)$ at x_1 we choose a scalar multiple m_1 with

$$f(x_1) = m_1(x_1 + b_1) = hm_1 \quad \Rightarrow \quad m_1 = f(x_1)/h.$$

The function $m_1 \max(0, x + b_1)$ interpolates f at x_1. Next we consider the point $x_2 = a + h$. To achieve $N(x_2) = f(x_2)$, we need to adjust the gradient on the interval $[x_1, x_2]$, and thus find a second scalar multiple m_2 with

$$f(x_2) = m_1(x_2 + b_1) + m_2(x_2 + b2) = 2hm_1 + hm_2.$$

Generally for $i \in \{1, \dots, n\}$ we get

$$\begin{aligned} f(x_i) &= \sum_{j=1}^{i} m_j(x_i + b_j) \\ &= \sum_{j=1}^{i} m_j(a + (i-1)h - (a + (j-2)h)) \\ &= \sum_{j=1}^{i} m_j(i - j + 1)h. \end{aligned}$$

From these equations, we can derive the following uniquely solvable lower triangular system.

$$h \begin{pmatrix} 1 & & & \\ 2 & 1 & & \\ \vdots & \ddots & \ddots & \\ n & \cdots & 2 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}.$$

The solution of this system determines the weights for the output layer. The bias for the output layer can be set to 0. If we let $n \to \infty$ in the equidistant formulation, the ReLU-approximations will converge by continuity to the function f, since continuity at x is given by

$$\forall\, \epsilon > 0\ \exists\, \delta > 0 \,:\, |f(x) - f(y)| < \epsilon\ \forall\, |x - y| < \delta.$$

To show the convergence, let $\epsilon > 0$. We have to proof that we can find a neural network N with

$$|f(x) - N(x)| < \epsilon \quad \text{for all } x \in [a, b].$$

Due to the continuity of f, we find a $\delta > 0$ with $|f(x) - f(y)| < \epsilon/2$ for all $|x - y| < \delta$.
Let $n \in \mathbb{N}$ with $h = \frac{1}{n-1} < \delta$. Let N be a neural network as constructed above and let $i \in \{1, \dots, n-1\}$.
Since N is a linear polynomial with gradient $\frac{f(x_{i+1}) - f(x_i)}{h}$ on the interval $[x_i, x_{i+1}]$, we get

$$N(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{h}(x - x_i) \quad \text{for all } x \in [x_i, x_{i+1}].$$

This indicates

$$\begin{aligned} |f(x) - N(x)| &= |f(x) - f(x_i) - \frac{f(x_{i+1}) - f(x_i)}{h}(x - x_i)| \\ &\le |f(x) - f(x_i)| + |f(x_{i+1}) - f(x_i)|\frac{|x - x_i|}{h} \\ &\le |f(x) - f(x_i)| + |f(x_{i+1}) - f(x_i)| < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon \end{aligned}$$

for all $x \in [x_i, x_{i+1}]$. Since i was chosen arbitrary, we get

$$|f(x) - N(x)| < \epsilon \quad \text{for all } x \in [a, b].$$

Exercise 3: (3+3 points)

a) Add backpropagation to our feedforward neural network. You can use the given templates.
   - Add an attribute `data` to our `ReLU` class that stores data for the backpropagation from the last evaluation.
   - Add a method `backprop(self, delta)` to our `ReLU` class that performs the back-propagation $a'(\mathbf{z}) \circ \boldsymbol{\delta}$, where $\mathbf{z}$ can be recovered from `self.data`.
   - Add attributes `__a` and `__z`, both initialised with `None`, to our `DenseLayer` class which store the input and the affine linear combination $\mathbf{Wa} + \mathbf{b}$ from the last evaluation.
   - Add attributes `dW` and `db`, initialised with zeros, to our `DenseLayer` class that store the updates for the weights and biases.
   - Add the method `backprop(self, delta)` to our `DenseLayer` class that computes the $\boldsymbol{\delta}$ for this layer, the derivatives of the cost function with respect to the weights and the biases of this layer, and returns $\mathbf{W}^{\mathsf{T}}\boldsymbol{\delta}$ for the previous layer.
   - Implement the method `backprop(self, x, y)` in our `SequentialNet` class that computes and returns the derivatives of the cost function with respect to the weights and the biases for the training data `x` and `y` corresponding to one minibatch.

b) Implement `train(self, x, y, batch_size=16, epochs=10)` that uses repeated calls to `backprop()` to carry out `epochs` epochs of SGD with batch size `batch_size` for the training data `x` and `y` corresponding to the whole training set. For this
   - implement a class `SGD` which is initialised with a learning rate `eta` and has a method `update(self, data, ddata)` that performs a SGD update step on the data in the list `data` with update data in the list `ddata`,
   - add an attribute `optim` to your `DenseLayer` class, which is is given on initialization and an instance of SGD by default,
   - add a method `update(self)` to your `DenseLayer` class that updates the weights and biases for a layer using the `update` method from the optimizer given in `optim`.

Solution: See the following listings for a possible implementation.

```python
import numpy as np

class ReLU():

    def __init__(self):

        self.data = None
        self.name = 'ReLU'

    def wfun(self, m, n):
        return np.random.randn(m,n)*np.sqrt(4/(m + n))

    def evaluate(self, x):

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

    def backprop(self, delta):

        return (self.data >= 0) * delta

class Abs():
```

```python

    def __init__(self):

        self.data  = None
        self.name  = 'Abs'

    def wfun(self, m, n):
        return np.random.randn(m,n)*np.sqrt(1/(m + n))

    def evaluate(self, x):

        self.data = x
        return np.abs(x)

    def backprop(self, delta):

        return np.sign(self.data)*delta
```

```python
import numpy as np

from activations import ReLU
from optimizers  import SGD

class DenseLayer:

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

        self.ni   = ni
        self.no   = no

        if afun is None:
            self.afun = ReLU()
        else:
            self.afun = afun

        if optim is None:
            self.optim = SGD()
        else:
            self.optim = optim

        self.W    = self.afun.wfun(no, ni)
        self.b    = np.zeros((no,1))
        self.__z  = None
        self.__a  = None
        self.dW   = np.zeros_like(self.W)
        self.db   = np.zeros_like(self.b)

    def evaluate(self, a):

        self.__a = a
        self.__z = self.W @ self.__a + self.b
```

```
        return self.afun.evaluate(self.__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

    def backprop(self, delta):

        delta   = self.afun.backprop(delta)
        self.dW = delta @ self.__a.T
        self.db = delta @ np.ones((delta.shape[1],1))

        return self.W.T @ delta

    def update(self):

        self.optim.update([self.W, self.b],
                          [self.dW, self.db])
```

```
import numpy as np

class SGD:

    def __init__(self, eta=.1):

        self.eta = eta

    def update(self, data, ddata):

        for p, dp in zip(data, ddata):

            p -= self.eta * dp
```

```
    def backprop(self, x, y):

        delta = (self.evaluate(x) - y)/y.shape[1]

        for layer in reversed(self.layers):

            delta = layer.backprop(delta)

    def train(self, x, y, batch_size=16, epochs=10):

        n_data    = y.shape[1]
        n_batches = int(np.ceil(n_data/batch_size))

        for e in range(epochs):

            p = np.random.permutation(n_data)
            for j in range(n_batches):
```

```python

            self.backprop(x[:,p[j*batch_size:(j+1)*batch_size]],
                          y[:,p[j*batch_size:(j+1)*batch_size]])

            for layer in self.layers:
                layer.update()
```

Exercise 4: (2+2 points) In multiclass classification the function $\mathsf{SoftMax} : \mathbb{R}^n \to (0,1)^n$,

$$\mathbf{p} := \mathsf{SoftMax}(\mathbf{x}) := \frac{e^{\mathbf{x}}}{\mathbf{e}^\top e^{\mathbf{x}}} = \begin{pmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{pmatrix} / \Big(\sum_{j=1}^n e^{x_j}\Big),$$

which returns a probability distribution, is used as last layer together with the Cross Entropy Loss function

$$H(\mathbf{y}, \mathbf{p}) := -\sum_{j=1}^n y_j \ln(p_j)$$

as cost function. In training, the vector $\mathbf{y}$ will be a unit vector, $\mathbf{p}$ will be the outcome of the $\mathsf{SoftMax}$-layer. This is known as categorical cross entropy loss. Compute the derivative of the

a) $\mathsf{SoftMax}$ function $\mathbf{p} = \mathsf{SoftMax}(\mathbf{x})$ with respect to $\mathbf{x}$,
b) Cross Entropy Loss function combined with the $\mathsf{SoftMax}$ layer, $H(\mathbf{y}, \mathsf{SoftMax}(\mathbf{x}))$, with respect to $\mathbf{x}$.

How do you initialize backpropagation in this case?

Solution:

a) To compute the Jacobi matrix

$$\frac{\partial \mathsf{SoftMax}(\mathbf{x})}{\partial \mathbf{x}}$$

of the $\mathsf{SoftMax}$ function we take a look at the elements of the Jacobian and use the quotient rule,

$$\frac{\partial \frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}}}{\partial x_i} = \begin{cases} \frac{e^{x_i}\left(\sum_{k=1}^n e^{x_k}\right) - e^{x_i} e^{x_i}}{\left(\sum_{k=1}^n e^{x_k}\right)^2} = \frac{e^{x_i}}{\sum_{k=1}^n e^{x_k}} \cdot \frac{\sum_{k=1}^n e^{x_k} - e^{x_i}}{\sum_{k=1}^n e^{x_k}} = p_i(1 - p_i), & j = i, \\ \frac{-e^{x_j} e^{x_i}}{\left(\sum_{k=1}^n e^{x_k}\right)^2} = \frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}} \cdot \frac{-e^{x_i}}{\sum_{k=1}^n e^{x_k}} = p_j(0 - p_i), & j \neq i. \end{cases}$$

Thus,

$$\frac{\partial \frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}}}{\partial x_i} = p_j(\delta_{j,i} - p_i).$$

b) The derivative of the Cross Entropy Loss function is given by

$$\begin{aligned} \frac{\partial H(\mathbf{y}, \mathbf{p})}{\partial x_i} &= \frac{\partial - \sum_{j=1}^n y_j \ln(p_j)}{\partial x_i} = -\sum_{j=1}^n y_j \frac{\partial \ln(p_j)}{\partial x_i} = -\sum_{j=1}^n y_j \frac{\partial \ln(p_j)}{\partial p_j} \frac{\partial p_j}{\partial x_i} \\ &= -\sum_{j=1}^n \frac{y_j}{p_j} p_j(\delta_{j,i} - p_i) = -y_i + p_i \sum_{j=1}^n y_j = p_i - y_i, \end{aligned}$$

since $\mathbf{y}$ represents a probability distribution and thus the sum of all elements gives one. In vector form, the gradient is given by

$$\frac{\partial H(\mathbf{y}, \mathbf{p})}{\partial \mathbf{x}} = \mathbf{p} - \mathbf{y}.$$

To initialize the backpropagation we simply have to set $\boldsymbol{\delta}_{m-1} = \mathbf{p} - \mathbf{y}$. This is in contrast to the mean square error $\|\mathbf{a}_m - \mathbf{y}\|_2^2/2$, where we additionally had to multiply $\mathbf{a}_m - \mathbf{y}$ elementwise with the derivative of the activation function to obtain $\boldsymbol{\delta}_{m-1}$.

Exercise 5: (1+1+2 points) Test your class `SequentialNet` or an implementation using TensorFlow/Keras with

a) the data from exercise 1b) of the first exercise sheet with one hidden layer comprising 3, 4, 10, or 100 neurons and ReLU activation functions,

b) the data given by

```
x = np.expand_dims(np.linspace(0, np.pi/2, 2000),0)
y = np.concatenate((np.sin(x), np.cos(x)))/2+.5
x_train, y_train = x[:,::2], y[:,::2]
x_test, y_test = x[:,1::2], y[:,1::2]
```

with one hidden layer of 100 neurons and ReLU activation functions,

c) MNIST with input layer of size 784, a hidden layer of size 100, and an output layer of size 10 with ReLU activation functions, either with the squared error cost function or a SoftMax layer followed by cross entropy loss. This data set is provided to you in the file `mnist.npz`. Further information can be found here:

http://yann.lecun.com/exdb/mnist/

Import the data set using `DATA = np.load('mnist.npz')` function followed by `x_train, y_train = DATA['x_train'], DATA['y_train']` and `x_test, y_test = DATA['x_test'], DATA['y_test']`.
Note that you need to reshape the data before you can train your neural net.

If you want to use TF/Keras you can find a description on

https://keras.io/getting-started/sequential-model-guide/

and a code skeleton in `TF_skeleton.py`.

Solution: The hand-made solution is based on our Python class `SequentialNet`, the solution based on TF/Keras is based on the description mentioned in the exercise. The main difference is that TF/Keras is based on the *transposed* input and output. Additionally the mean squared error is used as a loss function, so compared to our cost function it is divided by the number of predictions. Because we didn't further specify it weights and biases in the TF/Keras version they are initialised using a uniform distribution. In the neural net based on our class a standard normal distribution is used.

a) The implementation of the four neural nets of part a) is given in the following listing for our class `SequentialNet`, where we included the case of the absolute value as activation function,

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

from networks    import SequentialNet
from activations import Abs
from optimizers  import SGD
from layers      import DenseLayer
```

```

##################################################
# Aufgabe 6a)                                    #
##################################################

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

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

bs, ep, eta = 1, 5000, .1

######################################
# Tests with ReLU activation funtion #
######################################

layers = [DenseLayer(2, 3, optim=SGD(eta)),
          DenseLayer(3, 3, optim=SGD(eta))]
netz = SequentialNet(layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y␣=\n', y)
print('ytilde␣=\n', ytilde)
print('error␣=', np.linalg.norm(y - ytilde))
print('n␣=␣3,␣ReLU')

layers = [DenseLayer(2, 4, optim=SGD(eta)),
          DenseLayer(4, 3, optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y␣=\n', y)
print('ytilde␣=\n', ytilde)
print('error␣=', np.linalg.norm(y - ytilde))
print('n␣=␣4,␣ReLU')

layers = [DenseLayer(2, 10, optim=SGD(eta)),
          DenseLayer(10, 3, optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y␣=\n', y)
print('ytilde␣=\n', ytilde)
print('error␣=', np.linalg.norm(y - ytilde))
print('n␣=␣10,␣ReLU')

layers = [DenseLayer(2, 100, optim=SGD(eta)),
          DenseLayer(100, 3, optim=SGD(eta))]
```

```
netz = SequentialNet(2, layers)

print('y =\n', y)
print('ytilde =\n', ytilde)
print('error =', np.linalg.norm(y - ytilde))
print('n = 100, ReLU')

######################################
# Tests with Abs activation funtion #
######################################

layers = [DenseLayer(2, 3, afun=Abs(), optim=SGD(eta)),
          DenseLayer(3, 3, afun=Abs(), optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)


ytilde = netz.evaluate(x)

print('y =\n', y)
print('ytilde =\n', ytilde)
print('error =', np.linalg.norm(y - ytilde))
print('n = 3, Abs')

layers = [DenseLayer(2, 4, afun=Abs(), optim=SGD(eta)),
          DenseLayer(4, 3, afun=Abs(), optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y =\n', y)
print('ytilde =\n', ytilde)
print('error =', np.linalg.norm(y - ytilde))
print('n = 4, Abs')

layers = [DenseLayer(2, 10, afun=Abs(), optim=SGD(eta)),
          DenseLayer(10, 3, afun=Abs(), optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y =\n', y)
print('ytilde =\n', ytilde)
print('error =', np.linalg.norm(y - ytilde))
print('n = 10, Abs')

layers = [DenseLayer(2, 100, afun=Abs(), optim=SGD(eta)),
          DenseLayer(100, 3, afun=Abs(), optim=SGD(eta))]
netz = SequentialNet(2, layers)
netz.train(x, y, bs, ep)

ytilde = netz.evaluate(x)

print('y =\n', y)
```

```
print('ytilde␣=\n', ytilde)
print('error␣=', np.linalg.norm(y - ytilde))
print('n␣=␣100,␣Abs')
```

The following listing gives the TF/Keras counterpart,

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from time import sleep

# training data for exercise 1b), sheet 1
x_train = np.array( [[0,0], [0,1], [1,0], [1,1]] )
y_train = np.array( [[0,0,0],  # xor, and, or
                     [1,0,1],
                     [1,0,1],
                     [0,1,1]])

# overall parameters
bs, ep, eta = 1, 1000, .1
sleep_time = 2

############################################################
# use minimal net for exercise 1b), sheet 1 as example
############################################################
# TF/Keras model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(3, input_shape=(2,),
                          activation='relu',
                          dtype='float64'),
    tf.keras.layers.Dense(3, activation='relu',
                          dtype='float64')
])

# TF/Keras training
sgd = tf.keras.optimizers.SGD(learning_rate=eta,
                              momentum=0.0, nesterov=False)
model.compile(optimizer=sgd,
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_train)
print("y_train␣=", y_train)
print("y_tilde␣=", y_tilde)
print("error␣=", np.linalg.norm(y_train-y_tilde))
print("n␣=␣3")
sleep(sleep_time)

############################################################
# use net of exercise 1b), sheet 1 as example
############################################################
# TF/Keras model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(4, input_shape=(2,),
                          activation='relu',
                          dtype='float64'),
```

```
    tf.keras.layers.Dense(3, activation='relu',
                          dtype='float64')
])

# TF/Keras training
sgd = tf.keras.optimizers.SGD(learning_rate=eta,
                              momentum=0.0, nesterov=False)
model.compile(optimizer=sgd,
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_train)
print("y_train =", y_train)
print("y_tilde =", y_tilde)
print("error =", np.linalg.norm(y_train-y_tilde))
print("n = 4")
sleep(sleep_time)

###########################################################
# use 10 neurons in hidden layer for exercise 1b), sheet 1
###########################################################
# TF/Keras model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=(2,),
                          activation='relu',
                          dtype='float64'),
    tf.keras.layers.Dense(3, activation='relu',
                          dtype='float64')
])

# TF/Keras training
sgd = tf.keras.optimizers.SGD(learning_rate=eta,
                              momentum=0.0, nesterov=False)
model.compile(optimizer=sgd,
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_train)
print("y_train =", y_train)
print("y_tilde =", y_tilde)
print("error =", np.linalg.norm(y_train-y_tilde))
print("n = 10")
sleep(sleep_time)

###########################################################
# use 100 neurons in hidden layer for exercise 1b), sheet 1
###########################################################
# TF/Keras model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(100, input_shape=(2,),
                          activation='relu',
                          dtype='float64'),
    tf.keras.layers.Dense(3, activation='relu',
                          dtype='float64')
```

```
])

# TF/Keras training
sgd = tf.keras.optimizers.SGD(learning_rate=eta,
                              momentum=0.0, nesterov=False)
model.compile(optimizer=sgd,
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_train)
print("y_train␣=", y_train)
print("y_tilde␣=", y_tilde)
print("error␣=", np.linalg.norm(y_train-y_tilde))
print("n␣=␣100")
```

We observe that we need multiple starts to find a solution. The implementation with the absolute value tends to find a solution more easily. With the chosen parameters for the batch size, the number of epochs, and the learning rate, the net either converges to almost machine precision or stagnates, the reason is the dying of ReLU neurons. Here, regularization would help.

b) The implementation of the neural net of part b) is given in the following listing for our class `SequentialNet`, where again we included the case of the absolute value as activation function,

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

from networks    import SequentialNet
from activations import Abs
from layers      import DenseLayer
from optimizers  import SGD

################################################
# Aufgabe 6b)                                  #
################################################

#######################################
# ReLU as activations function        #
#######################################

x = np.expand_dims(np.linspace(0, np.pi/2, 2000), 0)
y = np.concatenate((np.sin(x), np.cos(x)))/2 + .5

x_train, y_train = x[:,::2], y[:,::2]
x_test, y_test = x[:,1::2], y[:,1::2]

bs, ep, eta = 1, 100, .1

layers=[DenseLayer(1, 100, optim=SGD(eta)),
        DenseLayer(100, 2, optim=SGD(eta))]
netz = SequentialNet(1, layers)
netz.train(x_train, y_train, bs, ep)

ytilde = netz.evaluate(x_test)

```

```
print('error =', np.linalg.norm(y_test - ytilde))

plt.title('approximate sine and cosine using ReLU')
plt.plot(x.T, y.T, '-')
plt.plot(x_test.T, ytilde.T, '-.')
plt.plot(x_test.T, np.log10(np.abs(y_test.T - ytilde.T)), '--')
plt.legend(['sin', 'cos',
            'appr. sin', 'appr. cos',
            'err. sin', 'err cos'])
plt.show()
##########################################
# Abs as activation function             #
##########################################

x = np.expand_dims(np.linspace(0, np.pi/2, 2000), 0)
y = np.concatenate((np.sin(x), np.cos(x)))/2 + .5

x_train, y_train = x[:,::2], y[:,::2]
x_test, y_test = x[:,1::2], y[:,1::2]

bs, ep, eta = 1, 100, .1

layers=[DenseLayer(1, 100, afun=Abs(), optim=SGD(eta)),
        DenseLayer(100, 2, afun=Abs(), optim=SGD(eta))]
netz = SequentialNet(1, layers)
netz.train(x_train, y_train, bs, ep)

ytilde = netz.evaluate(x_test)

print('error =', np.linalg.norm(y_test - ytilde))

plt.title('approximate sine and cosine using Abs')
plt.plot(x.T, y.T, '-')
plt.plot(x_test.T, ytilde.T, '-.')
plt.plot(x_test.T, np.log10(np.abs(y_test.T - ytilde.T)), '--')
plt.legend(['sin', 'cos',
            'appr. sin', 'appr. cos',
            'err. sin', 'err cos'])
plt.show()
```

The following listing gives the TF/Keras counterpart,

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

# training data
x = np.expand_dims(np.linspace(0, np.pi/2, 1000),0)
y = np.concatenate((np.sin(x), np.cos(x)))/2+.5
x_train = x.T
y_train = y.T

bs, ep, eta = 32, 100, .1
h_neurons = 100

#######################################################
```

```
# use FNN with one hidden layer comprising 100 neurons
#######################################################
# TF/Keras model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(h_neurons, input_shape=(1,),
                          activation='relu',
                          dtype='float64'),
    tf.keras.layers.Dense(2, activation='relu',
                          dtype='float64')
])

# TF/Keras training
sgd = tf.keras.optimizers.SGD(learning_rate=eta,
                              momentum=0.0, nesterov=False)
model.compile(optimizer=sgd,
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_train)
print("error =", np.linalg.norm(y_train-y_tilde))
plt.plot(x_train, y_train, '-')
plt.plot(x_train, y_tilde, '-.')
plt.plot(x_train, np.log10(np.abs(y_train-y_tilde)), '--')
plt.legend(['sin', 'cos',
            'appr. sin', 'appr. cos',
            'err. sin', 'err. cos'])
plt.show()
```

We observe that sometimes the approximation fails, which occurs more frequently for ReLU than for the abolute value. Typical resulting pictures for the chosen batch size, the number of epochs, and the learning rate in case of a working run are given in Figure 1.

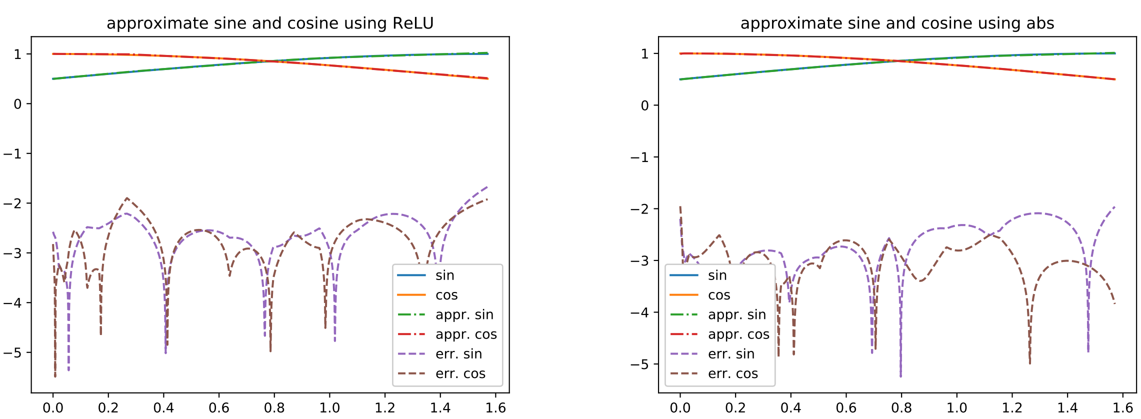


Figure 1: approximation of sine and cosine

c) The code for our class `SequentialNet` is given in the following listing,

```
import numpy             as np
import tensorflow        as tf
import matplotlib.pyplot as plt
from   random            import randrange
```

```python

from networks   import SequentialNet
from layers     import DenseLayer
from optimizers import SGD

DATA = np.load('mnist.npz')
x_train, y_train = DATA['x_train'], DATA['y_train'],
x_test, y_test   = DATA['x_test'], DATA['y_test']
x_train, x_test = x_train / 255.0, x_test / 255.0

x = x_train.reshape(60000, 784).T
I = np.eye(10)
y = I[:,y_train]

bs, ep, eta = 10, 10, .01

layers = [DenseLayer(784, 100, optim=SGD(eta)),
          DenseLayer(100, 10,  optim=SGD(eta))]
netz = SequentialNet(784, layers)
netz.train(x, y, bs, ep)

y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
guess   = np.argmax(y_tilde, 1).T
print('accuracy =', np.sum(guess == y_test)/100)

for i in range(4):

    k = randrange(y_test.size)
    plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=guess[0,k]))
    plt.imshow(x_test[k], cmap='gray')
    plt.show()
```

The TF/Keras counterpart is

```python
import numpy             as np
import tensorflow        as tf
import matplotlib.pyplot as plt
from   random            import randrange

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x = x_train.reshape(60000, 784)
I = np.eye(10)
y = I[y_train,:]

bs, ep, eta = 10, 10, .01

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(100, input_shape=(784,),
                          activation='relu',
                          dtype='float64'),
    tf.keras.layers.Dense(10, activation='relu',dtype='float64')
])

```

```python
sgd = tf.keras.optimizers.SGD(learning_rate=eta, momentum=0.0, nesterov=False)

model.compile(optimizer=sgd, loss='mean_squared_error',metrics=['accuracy'])
model.fit(x, y, epochs=ep, batch_size=bs)

y_tilde = model.predict(x_test.reshape(10000, 784))
guess   = np.argmax(y_tilde, 1)
print('accuracy =', np.sum(guess == y_test)/100)

for i in range(4):

    k = randrange(y_test.size)
    plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=guess[k]))
    plt.imshow(x_test[k], cmap='gray')
    plt.show()
```

The training of the net results typically in a value around 90% accuracy on the test set, best values are around 95–97%. This best value is more easily obtained for the absolute value activation function. The snippet presented in the lecture used additionally dropout after the hidden layer as regularization.

Exercise 6: (2+2 points) In this exercise we consider part of the proof of the backpropagation. Let

$$\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b}, \quad \mathbf{z}, \mathbf{b} \in \mathbb{R}^m, \quad \mathbf{a} \in \mathbb{R}^n, \quad \mathbf{W} \in \mathbb{R}^{m \times n}$$

denote the affine mapping in one of the layers. Let C be a cost function and let

$$\boldsymbol{\delta} = \frac{\partial C}{\partial \mathbf{z}} = \nabla_{\mathbf{z}} C \in \mathbb{R}^m$$

be given. Let the 3-tensor[1] $\mathbf{T} \in \mathbb{R}^{m \times m \times n}$ be defined by elements

$$t_{i,j}^{(\ell)} := \frac{\partial z_\ell}{\partial w_{i,j}}$$

for $\ell = 1, \dots, m$, $i = 1, \dots, m$, $j = 1, \dots, n$.

a) Compute for $\ell = 1, \dots, m$ the 2-tensor slices

$$\mathbf{T}^{(\ell)} := \begin{pmatrix} t_{1,1}^{(\ell)} & \cdots & t_{1,n}^{(\ell)} \\ \vdots & \ddots & \vdots \\ t_{m,1}^{(\ell)} & \cdots & t_{m,n}^{(\ell)} \end{pmatrix} \in \mathbb{R}^{m \times n}.$$

b) Show that the application of the 3-tensor $\mathbf{T}$ to the 1-tensor $\boldsymbol{\delta}$ gives the same 2-tensor as obtained using the elementwise chain rule,

$$\mathbf{T}\boldsymbol{\delta} := \sum_{\ell=1}^{m} \mathbf{T}^{(\ell)} \mathbf{e}_\ell^{\mathsf{T}} \boldsymbol{\delta} = \frac{\partial C}{\partial \mathbf{W}} \in \mathbb{R}^{m \times n}$$

[1] A 3-tensor is a natural generalization of a scalar (0-tensor), column vector (1-tensor), and matrix (2-tensor) to an additional dimension. Think of it as a cube of numbers, or as matrices stacked on top of each other towards the reader. Here, it makes sense to think of it as a row vector of matrices.

Solution:

a) We have

$$t_{i,j}^{(\ell)} = \frac{\partial z_\ell}{\partial w_{i,j}} = \frac{\partial \left(\sum_{k=1}^{n} w_{\ell,k} a_k + b_\ell\right)}{\partial w_{i,j}} = \begin{cases} a_j \text{ for } \ell = i, \\ 0 \text{ else.} \end{cases}$$

and thus

$$\begin{aligned} \mathbf{T}^{(\ell)} &= \begin{array}{c} \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ a_1 & \cdots & a_n \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \end{array} \begin{array}{l} \\ \\ \\ \leftarrow \ \ell\text{-th row} \\ \\ \\ \\ \end{array} \\ &= \mathbf{e}_\ell \mathbf{a}^\mathsf{T}. \end{aligned}$$

b) We need to calculate a tensor product. The matrix multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$

$$(\mathbf{A} \cdot \mathbf{B})_{i,j} = \sum_k a_{i,k} \cdot b_{k,j}$$

can be seen as being split in two parts. First we pair those entries of $\mathbf{A}$ and $\mathbf{B}$ where the second index of $\mathbf{A}$ and first index of $\mathbf{B}$ are identical. This is called *pairing*. Then we aggregate over the common index which is called *reducing*. One can also apply this logic to the matrix vector multiplication. Here we match the second index of $\mathbf{A}$ with the only index of $\mathbf{x}$:

$$(\mathbf{A} \cdot \mathbf{x})_i = \sum_k a_{i,k} \cdot x_k$$

Applying this in our case by pairing the index denoted by ℓ with the only index of δ and then reducing we get

$$(\mathbf{T}\boldsymbol{\delta})_{i,j} = \sum_{\ell=1}^{m} t_{i,j}^{(\ell)} \cdot \delta_\ell = a_j \cdot \delta_i \Rightarrow \ \mathbf{T}\boldsymbol{\delta} = \boldsymbol{\delta}\mathbf{a}^\mathsf{T}.$$

Using the hint in the exercise

$$\mathbf{T}\boldsymbol{\delta} = \sum_{\ell=1}^{m} \mathbf{T}^{(\ell)} \mathbf{e}_\ell^\mathsf{T} \boldsymbol{\delta} = \sum_{\ell=1}^{m} \mathbf{e}_\ell \mathbf{a}^\mathsf{T} \delta_\ell = \boldsymbol{\delta}\mathbf{a}^\mathsf{T}$$

we get the same result.