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 4, solutions

Exercise 1: (2 points) Let $\mathbf{i} = (i_1, \dots, i_d)$ be a multi-index. Prove that multi-dimensional convolution

$$(\mathbf{X} * \mathbf{F})(n_1, \dots, n_d) = \sum_{\mathbf{i}} \mathbf{X}(n_1 - i_1, \dots, n_d - i_d) \cdot \mathbf{F}(i_1, \dots, i_d)$$

is commutative, i.e., $\mathbf{X} * \mathbf{F} = \mathbf{F} * \mathbf{X}$.

Solution: We define the multi-index $\mathbf{j} = (j_1, \dots, j_d)$ and set $\mathbf{j} = \mathbf{n} - \mathbf{i}$. Thus, $\mathbf{i} = \mathbf{n} - \mathbf{j}$ and

$$\begin{aligned}(\mathbf{X} * \mathbf{F})(n_1, \dots, n_d) &= \sum_{\mathbf{i}} \mathbf{X}(n_1 - i_1, \dots, n_d - i_d) \cdot \mathbf{F}(i_1, \dots, i_d)\\ &= \sum_{\mathbf{j}} \mathbf{X}(j_1, \dots, j_d) \cdot \mathbf{F}(n_1 - j_1, \dots, n_d - j_d) = (\mathbf{F} * \mathbf{X})(n_1, \dots, n_d).\end{aligned}$$

Exercise 2: (4 points) In Lecture 4 we showed that

$$\frac{\partial C}{\partial \mathbf{f}} = \widetilde{(\mathbf{a} * \widetilde{\boldsymbol{\delta}})} = \widetilde{\mathbf{a}} * \boldsymbol{\delta},$$

for the case of a 1D kernel $\mathbf{f}$ (see lecture notes, p. 93[1]), where the tilde represents a flipping of the entries. We now consider a 2D kernel $\mathbf{F} \in \mathbb{R}^{2\times 2}$ with input $\mathbf{A} \in \mathbb{R}^{3\times 3}$. The convolution $\mathbf{A} * \mathbf{F}$ is the given by

$$\begin{pmatrix} y_{1,1} & y_{1,2} \\ y_{2,1} & y_{2,2} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} * \begin{pmatrix} f_{1,1} & f_{1,2} \\ f_{2,1} & f_{2,2} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \star \begin{pmatrix} f_{2,2} & f_{2,1} \\ f_{1,2} & f_{1,1} \end{pmatrix},$$

or in vector notation by

$$\left(\begin{array}{c} y_{1,1} \\ y_{1,2} \\ \hline y_{2,1} \\ y_{2,2} \end{array}\right) = \left(\begin{array}{ccc|ccc|ccc} f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 & 0 & 0 & 0 \\ 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 \\ 0 & 0 & 0 & 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} \end{array}\right) \left(\begin{array}{c} a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \hline a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \hline a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{array}\right).$$

[1] Version from 26.10.2021

a) How does the $\mathbf{\Delta}$ from the backpropagation look like in this case?
b) Prove the equality

$$\widetilde{(\mathbf{A} * \widetilde{\mathbf{\Delta}})} = \widetilde{\mathbf{A}} * \mathbf{\Delta}.$$

Solution:

a) The input is a matrix $\mathbf{A} \in \mathbb{R}^{3\times 3}$, the kernel is $\mathbf{F} \in \mathbb{R}^{2\times 2}$, thus the output for a complete overlap of kernel and input is a matrix $\mathbf{Y} \in \mathbb{R}^{2\times 2}$. This implies that the convolution maps a layer from $\mathbb{R}^{3\times 3}$ to the next layer in $\mathbb{R}^{2\times 2}$, $\mathbf{Z} = \mathbf{A} * \mathbf{F} + \mathbf{B} \in \mathbb{R}^{2\times 2}$, so the backpropagation will start with a $\mathbf{\Delta} \in \mathbb{R}^{2\times 2}$ in the new layer. The update of the kernel is based on $\mathbf{A} \in \mathbb{R}^{3\times 3}$ and $\mathbf{\Delta} \in \mathbb{R}^{2\times 2}$ and gives a matrix in $\mathbb{R}^{2\times 2}$. The matrix $\mathbf{\Delta}$ and the flipped matrix $\widetilde{\mathbf{\Delta}}$ have the entries

$$\mathbf{\Delta} = \begin{pmatrix} \delta_{1,1} & \delta_{1,2} \\ \delta_{2,1} & \delta_{2,2} \end{pmatrix}, \qquad \widetilde{\mathbf{\Delta}} = \begin{pmatrix} \delta_{2,2} & \delta_{2,1} \\ \delta_{1,2} & \delta_{1,1} \end{pmatrix}.$$

b) We switch from matrices to vectors and use the commutativity of the convolution,

$$\left(\begin{array}{c} z_{1,1} \\ z_{1,2} \\ \hline z_{2,1} \\ z_{2,2} \end{array}\right) = \left(\begin{array}{ccc|ccc|ccc} f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 & 0 & 0 & 0 \\ 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} & 0 \\ 0 & 0 & 0 & 0 & f_{2,2} & f_{2,1} & 0 & f_{1,2} & f_{1,1} \end{array}\right) \left(\begin{array}{c} a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \hline a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \hline a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{array}\right) + \left(\begin{array}{c} b_{1,1} \\ b_{1,2} \\ \hline b_{2,1} \\ b_{2,2} \end{array}\right)$$

$$= \left(\begin{array}{cc|cc} a_{2,2} & a_{2,1} & a_{1,2} & a_{1,1} \\ a_{2,3} & a_{2,2} & a_{1,3} & a_{1,2} \\ \hline a_{3,2} & a_{3,1} & a_{2,2} & a_{2,1} \\ a_{3,3} & a_{3,2} & a_{2,3} & a_{2,2} \end{array}\right) \left(\begin{array}{c} f_{1,1} \\ f_{1,2} \\ \hline f_{2,1} \\ f_{2,2} \end{array}\right) + \left(\begin{array}{c} b_{1,1} \\ b_{1,2} \\ \hline b_{2,1} \\ b_{2,2} \end{array}\right) = \mathbf{T}\mathbf{f} + \mathbf{b}.$$

The derivative of $\mathbf{z}$ with respect to $\mathbf{f}$ is thus given by $\mathbf{T}^\mathsf{T}$ and

$$\left(\begin{array}{c} \partial C/\partial f_{1,1} \\ \partial C/\partial f_{1,2} \\ \hline \partial C/\partial f_{2,1} \\ \partial C/\partial f_{2,2} \end{array}\right) = \frac{\partial C}{\partial \mathbf{f}} = \frac{\partial \mathbf{z}}{\partial \mathbf{f}} \frac{\partial C}{\partial \mathbf{z}} = \mathbf{T}^\mathsf{T} \boldsymbol{\delta} = \left(\begin{array}{cc|cc} a_{2,2} & a_{2,3} & a_{3,2} & a_{3,3} \\ a_{2,1} & a_{2,2} & a_{3,1} & a_{3,2} \\ \hline a_{1,2} & a_{1,3} & a_{2,2} & a_{2,3} \\ a_{1,1} & a_{1,2} & a_{2,1} & a_{2,2} \end{array}\right) \left(\begin{array}{c} \delta_{1,1} \\ \delta_{1,2} \\ \hline \delta_{2,1} \\ \delta_{2,2} \end{array}\right).$$

We rewrite the latter convolution as matrix-vector product with the vector of the elements of the matrix $\mathbf{A}$,

$$\left(\begin{array}{cc|cc} a_{2,2} & a_{2,3} & a_{3,2} & a_{3,3} \\ a_{2,1} & a_{2,2} & a_{3,1} & a_{3,2} \\ \hline a_{1,2} & a_{1,3} & a_{2,2} & a_{2,3} \\ a_{1,1} & a_{1,2} & a_{2,1} & a_{2,2} \end{array}\right) \left(\begin{array}{c} \delta_{1,1} \\ \delta_{1,2} \\ \hline \delta_{2,1} \\ \delta_{2,2} \end{array}\right) = \left(\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} \\ 0 & 0 & 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 \\ \hline 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 & 0 & 0 \\ \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 & 0 & 0 & 0 \end{array}\right) \left(\begin{array}{c} a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \hline a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \hline a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{array}\right)$$

We note that flipping the vector of the elements of $\mathbf{A}$ results in the vector of the doubly flipped activated element $\widetilde{\mathbf{A}}$,

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \rightsquigarrow \begin{pmatrix} a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \hline a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \hline a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix} \Leftrightarrow \begin{pmatrix} a_{3,3} \\ a_{3,2} \\ a_{3,1} \\ \hline a_{2,3} \\ a_{2,2} \\ a_{2,1} \\ \hline a_{1,3} \\ a_{1,2} \\ a_{1,1} \end{pmatrix} \rightsquigarrow \begin{pmatrix} a_{3,3} & a_{3,2} & a_{3,1} \\ a_{2,3} & a_{2,2} & a_{2,1} \\ a_{1,3} & a_{1,2} & a_{1,1} \end{pmatrix} = \widetilde{\mathbf{A}}.$$

The two ways of writing the derivative of the cost function C with respect to the elements of $\mathbf{f}$ are given by flipping the rows of both sides or to flipping the rows of the vector of $\mathbf{A}$'s elements and the columns of the block Toeplitz matrix of the $\boldsymbol{\delta}$ elements. The flipping of the rows on both sides results in

$$\widetilde{\frac{\partial C}{\partial \mathbf{f}}} = \begin{pmatrix} \partial C/\partial f_{2,2} \\ \partial C/\partial f_{2,1} \\ \hline \partial C/\partial f_{1,2} \\ \partial C/\partial f_{1,1} \end{pmatrix} = \left(\begin{array}{ccc|ccc|ccc} \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 & 0 & 0 & 0 \\ 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} & 0 \\ 0 & 0 & 0 & 0 & \delta_{1,1} & \delta_{1,2} & 0 & \delta_{2,1} & \delta_{2,2} \end{array}\right) \begin{pmatrix} a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ \hline a_{2,1} \\ a_{2,2} \\ a_{2,3} \\ \hline a_{3,1} \\ a_{3,2} \\ a_{3,3} \end{pmatrix},$$

which corresponds to

$$\widetilde{\frac{\partial C}{\partial \mathbf{F}}} = \mathbf{A} * \widetilde{\boldsymbol{\Delta}} = \mathbf{A} \star \boldsymbol{\Delta}.$$

The flipping of the elements of $\mathbf{A}$ and the columns of the block Toeplitz matrix results in

$$\frac{\partial C}{\partial \mathbf{f}} = \begin{pmatrix} \partial C/\partial f_{1,1} \\ \partial C/\partial f_{1,2} \\ \hline \partial C/\partial f_{2,1} \\ \partial C/\partial f_{2,2} \end{pmatrix} = \left(\begin{array}{ccc|ccc|ccc} \delta_{2,2} & \delta_{2,1} & 0 & \delta_{1,2} & \delta_{1,1} & 0 & 0 & 0 & 0 \\ 0 & \delta_{2,2} & \delta_{2,1} & 0 & \delta_{1,2} & \delta_{1,1} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & \delta_{2,2} & \delta_{2,1} & 0 & \delta_{1,2} & \delta_{1,1} & 0 \\ 0 & 0 & 0 & 0 & \delta_{2,2} & \delta_{2,1} & 0 & \delta_{1,2} & \delta_{1,1} \end{array}\right) \begin{pmatrix} a_{3,3} \\ a_{3,2} \\ a_{3,1} \\ \hline a_{2,3} \\ a_{2,2} \\ a_{2,1} \\ \hline a_{1,3} \\ a_{1,2} \\ a_{1,1} \end{pmatrix},$$

which corresponds to

$$\frac{\partial C}{\partial \mathbf{F}} = \widetilde{\mathbf{A}} * \boldsymbol{\Delta} = \widetilde{\mathbf{A}} \star \widetilde{\boldsymbol{\Delta}}.$$

Thus we get the desired equality

$$\widetilde{(\mathbf{A} * \widetilde{\boldsymbol{\Delta}})} = \frac{\partial C}{\partial \mathbf{F}} = \widetilde{\mathbf{A}} * \boldsymbol{\Delta}.$$

Exercise 3: (4 points) In preparation for convolutional networks change the current implementation of `DenseLayer` to treat *derivatives* in place of *gradients*:

- Initialize with transposed weights and biases given by `np.zeros(no)`.
- Evaluate using $\mathbf{z} = \mathbf{a}\mathbf{W} + \mathbf{b}$, $a(\mathbf{z})$.
- Backpropagate using
$$\frac{\partial C}{\partial \mathbf{W}} = \mathbf{a}^\mathsf{T}\boldsymbol{\delta}, \quad \frac{\partial C}{\partial \mathbf{b}} = \boldsymbol{\delta},$$
now by abuse of notation denoting the *derivatives*, and output $\boldsymbol{\delta}\mathbf{W}^\mathsf{T}$.
- Which other parts do you have to change accordingly?

Solution: A possible solution can be found in the following listing:

```
class DenseLayer:

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

        if initializer is None:
          self.initializer = RandnAvarage()
        else:
          self.initializer = initializer

        self.W    = self.afun.factor * self.initializer.wfun(ni, no)
        self.b    = np.zeros(no)
        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.__a @ self.W + self.b
        return self.afun.evaluate(self.__z)

```

```
    def set_weights(self, W):

        assert(W.shape == (self.ni, self.no))
        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 = self.__a.T @ delta
        self.db = np.sum(delta, 0)

        if not self.kernel_regularizer is None:
            self.kernel_regularizer.update(self.W, self.dW)

        return delta @ self.W.T

    def update(self):

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

Note that the calls of `backprop` in the `train` method have to be changed, too.

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

    n_data    = y.shape[0]
    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):

        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: (4 points)

a) Add a class `DropoutLayer` to `layers.py`:

   (i) Initialize with the following attributes:
   - `p`: the probability $\mathtt{p} \in [0.5, 0.8]$ with which neurons are kept, default: `p = .5` .
   - `mask`: a scaled version of a vector with 0's and 1's, specifiying the dropped neurons, set to `None` on initialization.
   - `trainable`: a Boolean value given on initialization (`True`: the network is trained, neurons are dropped, and the output is scaled; `False`: the network is not trained and the weights remain unchanged).

(ii) Add the following methods:
- `evaluate(self, a)`: If the network is trained, i.e., `trainable` is `True`, neurons are dropped and the input gets scaled. If the network is not trained, i.e., `trainable` is `False`, the input is returned.
  Hint: Implement the dropping of neurons by setting `self.mask` using
  `numpy.random.binomial(1, self.p, size=shape)`.
  The parameter `shape` is a tuple $(1, m)$ with m being the number of given input vectors. This ensures the same mask for the whole minibatch. The scaling factor is chosen as $1/(1-p)$.
- `backprop(self, delta)`: backpropagation by dropping neurons as given in `self.mask`.
- `update`: perform the update after backpropagation. It shouldn't do anything since there are no parameters to update.

b) Test your code by expanding exercise 5c) from sheet 2. Use the logistic function as activation function. Compare the result for different probabilities `p` $\in [0.5, 0.8]$ to a network without dropout.

Note that a higher learning rate is suggested for networks with dropout. Try different sizes for minibatches and different learning rates as well. Note that a higher learning rate is recommended for layers with dropout, use, e.g., `10*eta` as learning rate for layers w/ dropout, if `eta` is the learning rate for a layer w/o dropout.

Solution: A possible implementation can be found in the following listing:

```
class DropoutLayer:

    def __init__(self, p=.5, trainable=True):

        self.p = p
        self.trainable = trainable
        self.mask = None

    def evaluate(self, a):

      if self.trainable:

        shape = (1,) + a.shape[1:] # same mask for minibatch
        scale = 1/(1-self.p)
        self.mask = scale * \
            np.random.binomial(1, self.p, size=shape)
        drop_a = a * self.mask
        return drop_a
      else:
        return a

    def backprop(self, delta):

      delta *= self.mask
      return delta

    def update(self):
        pass
```

.

If a layer with dropout is evaluated for testing, the learned weights lead to a higher output since the weights were learned for thinned layers. Thus the output is typically scaled by p. Another approach, which we used for our implementation, is to scale the input while the network is trained.

A possible solution for part b) can be found in the following listing.

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

from networks    import SequentialNet
from layers      import *
from optimizers  import *
from activations import *

DATA = np.load('mnist.npz')
#DATA = np.load('fashion_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

categories = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle␣boot']

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

bs, ep, eta = 1000, 10, .001

print('Ohne␣Dropout')
layers = [DenseLayer(784, 100, afun=Logistic(), optim=SGD(eta)),
          DenseLayer(100, 10 , afun=Logistic(), optim=SGD(eta))]
netz = SequentialNet(784,layers)
netz.train(x, y, bs, ep)

y_tilde = netz.evaluate(x_test.reshape(10000,784))
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[k]))
    plt.imshow(x_test[k], cmap='gray')
    plt.show()

prop = np.linspace(.5, .8, 7, True)
for p in prop:
    print('Mit␣Dropout,␣p␣=', p)
    layers = [DenseLayer(784, 100, afun=Logistic(), optim=SGD(eta*10)),
              DropoutLayer(p),
              DenseLayer(100, 10, afun=Logistic(), optim=SGD(eta*10))]
    netz = SequentialNet(784, layers)
    netz.train(x, y, bs, ep)
```

```python

    y_tilde = netz.evaluate(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()
```