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 5, solutions

Exercise 1: (3 points) In standard implementations, e.g., in TensorFlow/Keras, due to Python's way of storing multiarrays (the last index moves fastest), (activated) minibatches $\mathbf{A}$ of $n \in \mathbb{N}$ images with $c \in \mathbb{N}$ channels of height $h \in \mathbb{N}$ and width $w \in \mathbb{N}$ are stored in the 4D-tensor format

$$\mathbf{A} \in \mathbb{R}^{n \times c \times h \times w},$$

the so-called NCHW format. The filter bank $\mathbf{F}$ of $m \in \mathbb{N}$ filters (m feature maps) of width $f_w \in \mathbb{N}$ and height $f_h \in \mathbb{N}$ that sums over the aforementioned channels c is stored in the 4D-tensor format

$$\mathbf{F} \in \mathbb{R}^{m \times c \times f_h \times f_w}.$$

For every feature map there exists a bias, thus the bias is a vector $\mathbf{b} \in \mathbb{R}^m$.

Derive the backpropagation and kernel update of the feedforward step

$$\mathbf{Z}(i, j, :, :) = b(j)\mathbf{E} + \sum_{k=1}^{c} \mathbf{A}(i, k, :, :) * \mathbf{F}(j, k, :, :), \quad \mathbf{A}_{\text{new}} = a(\mathbf{Z})$$

for some activation function a by hand. Here $\mathbf{E}$ is a matrix comprising ones.
Hint: Consider backpropagation for one step with given $\mathbf{\Delta} \in \mathbb{R}^{n \times m \times z_h \times z_w}$ and $\mathbf{\Delta}^{\texttt{pre}} \in \mathbb{R}^{n \times c \times h \times w}$ to be evaluated. Make use of the multidimensional chain rule. Note, that for the feedforward step above one can show

$$\frac{\partial C}{\partial \mathbf{A}(i, k, :, :)} = \sum_{j=1}^{m} \frac{\partial C}{\partial \mathbf{Z}(i, j, :, :)} *_{\texttt{full}} \mathbf{F}(j, k, :: -1, :: -1)$$

for all $i \in \{1, \dots, n\}$, $k \in \{1 \dots, c\}$

Solution: We use the chain rule and partial derivatives. Due to the dimensions we use multi-indices. We define analogously to the backpropagation for dense layers the variables $\mathbf{\Delta}^{\texttt{pre}}$ of the previous layer, which has to be computed, and $\mathbf{\Delta}$ of the current layer, which we assume to be given,

$$\mathbf{\Delta}^{\texttt{pre}} \in \mathbb{R}^{n \times c \times h \times w}, \quad \mathbf{\Delta} \in \mathbb{R}^{n \times m \times z_h \times z_w},$$

where $z_h := h - f_h + 1$ and $z_w := w - f_w + 1$, by

$$(\mathbf{\Delta}^{\texttt{pre}})_{\mathbf{i}} := \frac{\partial C}{\partial (\mathbf{Z}^{\texttt{pre}})_{\mathbf{i}}}, \quad (\mathbf{\Delta})_{\mathbf{i}} := \frac{\partial C}{\partial (\mathbf{Z})_{\mathbf{i}}}, \quad \mathbf{i} \in \mathbb{N}^4.$$

Here, $\mathbf{Z}^{\texttt{pre}} \in \mathbb{R}^{n \times c \times h \times w}$ is the non-activated $\mathbf{A}$ of the previous layer, i.e., either for some activation function $a^{\texttt{pre}}$ we have $\mathbf{A} = a^{\texttt{pre}}(\mathbf{Z}^{\texttt{pre}})$, or for the input we have $\mathbf{Z}^{\texttt{pre}} = \mathbf{X}$, e.g., $a^{\texttt{pre}}$ is the identity.

To derive backpropagation, i.e., compute $\boldsymbol{\Delta}^{\texttt{pre}}$ from $\boldsymbol{\Delta}$, we use the chain rule twice, first,

$$(\boldsymbol{\Delta}^{\texttt{pre}})_{\mathbf{i}} = \frac{\partial C}{\partial \mathbf{Z}^{\texttt{pre}}_{\mathbf{i}}} = \sum_{\mathbf{j}} \frac{\partial \mathbf{A}_{\mathbf{j}}}{\partial \mathbf{Z}^{\texttt{pre}}_{\mathbf{i}}} \frac{\partial C}{\partial \mathbf{A}_{\mathbf{j}}} = (a^{\texttt{pre}})'(\mathbf{Z}^{\texttt{pre}}_{\mathbf{i}}) \cdot \frac{\partial C}{\partial \mathbf{A}_{\mathbf{i}}}$$

which gives the part $(a^{\texttt{pre}})'(\mathbf{Z}^{\texttt{pre}})$ that will be computed in the previous layer. The part that will be computed in the current layer is developed further using the chain rule for the second time,

$$\frac{\partial C}{\partial \mathbf{A}_{\mathbf{i}}} = \sum_{\mathbf{j}} \frac{\partial \mathbf{Z}_{\mathbf{j}}}{\partial \mathbf{A}_{\mathbf{i}}} \frac{\partial C}{\partial \mathbf{Z}_{\mathbf{j}}} = \sum_{\mathbf{j}} \frac{\partial \mathbf{Z}_{\mathbf{j}}}{\partial \mathbf{A}_{\mathbf{i}}} \boldsymbol{\Delta}_{\mathbf{j}}.$$

Now we consider the indices,

$$\frac{\partial C}{\partial \mathbf{A}(i_1, i_2, i_3, i_4)} = \sum_{j_1, j_2, j_3, j_4} \frac{\partial \mathbf{Z}(j_1, j_2, j_3, j_4)}{\partial \mathbf{A}(i_1, i_2, i_3, i_4)} \boldsymbol{\Delta}(j_1, j_2, j_3, j_4).$$

The derivative is zero when $j_1 \neq i_1$, since

$$\mathbf{Z}(j_1, j_2, :, :) = b(j_2)\mathbf{E} + \sum_{i_2=1}^{c} \mathbf{A}(j_1, i_2, :, :) * \mathbf{F}(j_2, i_2, :, :)$$

does not depend on $\mathbf{A}(i_1, :, :, :)$ for $j_1 \neq i_1$, thus we fix $j_1 = i_1$. The second index implements a sum over all channels, the derivative of a sum is the sum of derivatives, thus we split sums to obtain

$$\frac{\partial C}{\partial \mathbf{A}(i_1, i_2, i_3, i_4)} = \sum_{j_2} \left(\sum_{j_3, j_4} \frac{\partial \mathbf{Z}(i_1, j_2, j_3, j_4)}{\partial \mathbf{A}(i_1, i_2, i_3, i_4)} \boldsymbol{\Delta}(i_1, j_2, j_3, j_4) \right).$$

For fixed indices i_1, i_2, j_2 we have the derivative of the 2D convolution of the matrices $\mathbf{A}(i_1, i_2, :, :)$ with the 2D kernel $\mathbf{F}(j_2, i_2, :, :)$ applied to the matrix $\boldsymbol{\Delta}(i_1, j_2, :, :)$. This is given by the full convolution of the doubly flipped (rotated) kernel with $\boldsymbol{\Delta}(i_1, j_2, :, :)$, i.e., setting for simplicity $i = i_1$, $j = j_2$, $k = i_2$,

$$\frac{\partial C}{\partial \mathbf{A}(i, k, :, :)} = \sum_{j=1}^{m} \boldsymbol{\Delta}(i, j, :, :) *_{\texttt{full}} \mathbf{F}(j, k, :: -1, :: -1).$$

This quantity is handed to the previous layer.

To compute the bias and kernel update, we first compute the correct $\boldsymbol{\Delta}$ from the input of the next layer by componentwise multiplication with $a'(\mathbf{Z})$, compare with the layer-wise approach for dense layers and the above remarks.

The bias is easily obtained, as

$$\mathbf{Z}(i, j, :, :) = b(j)\mathbf{E} + \sum_{k=1}^{c} \mathbf{A}(i, k, :, :) * \mathbf{F}(j, k, :, :)$$

does only depend in the second position on $b(j)$, thus

$$\frac{\partial C}{\partial b(j)} = \sum_{i,k,\ell} \frac{\partial C}{\partial \mathbf{Z}(i, j, k, \ell)} \frac{\partial \mathbf{Z}(i, j, k, \ell)}{\partial b(j)} = \sum_{i,k,\ell} \boldsymbol{\Delta}(i, j, k, \ell).$$

The computation of all updates of biases $b(j)$ for $j = 1, \dots, m$ can be implemented fast and simultaneously as `np.sum(Delta, axis=(0, 2, 3))` in Python.

The update of the kernel bank is defined by

$$(d\mathbf{F})_{\mathbf{i}} := \frac{\partial C}{\partial \mathbf{F}_{\mathbf{i}}} = \sum_{\mathbf{j}} \frac{\partial \mathbf{Z}_{\mathbf{j}}}{\partial \mathbf{F}_{\mathbf{i}}} \frac{\partial C}{\partial \mathbf{Z}_{\mathbf{j}}} = \sum_{j_1, j_2, j_3, j_4} \frac{\partial \mathbf{Z}(j_1, j_2, j_3, j_4)}{\partial \mathbf{F}(i_1, i_2, i_3, i_4)} \boldsymbol{\Delta}(j_1, j_2, j_3, j_4).$$

As $\mathbf{Z}(:, j_2, :, :)$ does not depend on $\mathbf{F}(i_1, :, :, :)$ for $i_1 \neq j_2$ we set $i_1 = j_2$ and split the remaining sums,

$$d\mathbf{F}(j_2, i_2, i_3, i_4) = \sum_{j_1} \left(\sum_{j_3, j_4} \frac{\partial \mathbf{Z}(j_1, j_2, j_3, j_4)}{\partial \mathbf{F}(j_2, i_2, i_3, i_4)} \boldsymbol{\Delta}(j_1, j_2, j_3, j_4) \right).$$

For simplicity, we set $j = j_2$, $k = i_2$, and $i = j_1$,

$$d\mathbf{F}(j, k, i_3, i_4) = \sum_{i} \left(\sum_{j_3, j_4} \frac{\partial \mathbf{Z}(i, j, j_3, j_4)}{\partial \mathbf{F}(j, k, i_3, i_4)} \boldsymbol{\Delta}(i, j, j_3, j_4) \right).$$

For fixed i, j, k the term in parentheses corresponds to the (known) derivative of the convolution of $\mathbf{A}(i, k, :, :)$ with $\mathbf{F}(j, k, :, :)$ applied to $\boldsymbol{\Delta}(i, j, :, :)$, i.e.,

$$d\mathbf{F}(j, k, :: -1, :: -1) = \sum_{i} \mathbf{A}(i, k, :, :) * \boldsymbol{\Delta}(i, j, :: -1, :: -1).$$

This finishes the solution.

Exercise 2: (5 points) Add a class `Conv2DLayer()` that is initialized by

`Conv2DLayer(tensor, fshape, afun=ReLU(), optim=SGD())`,

to `layers.py` where

- `tensor` is a tuple (c, h, w) of channels, height, and width of the (activated) images on input, e.g., for MNIST $(1, 28, 28)$ and
- `fshape` is a tuple (m, f_h, f_w) describing the filter bank of m filters of size $f_h \times f_w$, e.g., for a typical filter bank $(10, 3, 3)$.

The class should store additionally

- `afun`: the activation function a, default value `ReLU()`,
- `optim`: the optimizer used on this layer, default value `SGD()`,
- `initializer`: the initializer for the filter bank with `RandnAverage()` as default value
- `f`: the filter bank initialized with the method `ffun` from the initializer and scaled by `self.afun.factor`,
- `b`: the bias, zero on initialization,
- `__a`: the input $\mathbf{a}$ from the last evaluation, initialized by `None`,
- `__z`: the result $\mathbf{z} = \mathbf{a} * \mathbf{f} + \mathbf{b}$ after applying the filter bank, initialized by `None`,
- `df`: the update of `f` from the backpropagation, zero on initialization,
- `db`: the update of `b` from the backpropagation, zero on initialization.

Add the following methods, using `convolve2d` of the SciPy package wherever necessary.

- `evaluate(self, a)`: Evaluate the feedforward step (see lecture notes, p.106[1], or exercise 1)

[1] Version from 01.11.2021

- `update(self)`: updates the filter bank and the bias using the method `update()` of the optimizer.

to the new class. A method `backprop` is already implemented in `layers.py`. Check your implementation using gradient checking which is provided to you in the file `layers.py`.

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

```python
class Conv2DLayer:

    def __init__(self, tensor, fshape, afun=None, optim=None, initializer=None):

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

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

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

        self.tensor = tensor
        self.fshape = fshape

        m, fh, fw = fshape

        c, h, w   = tensor
        self.f = self.initializer.ffun(m, c, fh, fw)
        self.f *= self.afun.factor
        self.b = np.zeros(m)

        self.__a = None
        self.__z = None

        self.df = np.zeros_like(self.f)
        self.db = np.zeros_like(self.b)

    def evaluate(self, a):

        n, c, h, w = a.shape
        m, fh, fw  = self.fshape

        self.__a = a

        self.__z = np.zeros((n, m, h - fh + 1, w - fw + 1))
        for j in range(m):
            for i in range(n):
                self.__z[i,j,:,:] += self.b[j]
                for k in range(c):
                    self.__z[i,j,:,:] += convolve2d(self.__a[i,k,:,:],\
                                                    self.f[j,k,:,:],
```

```python
                                                                    mode='valid')

        return self.afun.evaluate(self.__z)

    def backprop(self, delta):

        n, c, h, w = self.__a.shape
        m, fh, fw  = self.fshape
        # Compute a'(z)*delta
        delta = self.afun.backprop(delta)

        # Compute bias change
        self.db = np.sum(delta, axis=(0,2,3))

        # Compute df = delta * f^~. df has shape (m, c, fh, fw)
        for k in range(c):
            for j in range(m):
                for i in range(n):
                    self.df[j,k,:,:] += convolve2d(self.__a[i,k,::-1,::-1],
                                                   delta[i,j,:,:],
                                                   mode = 'valid')

        # Update delta
        # The new delta has the same shape as __a: (n,c,h,w)
        delta2 = np.zeros_like(self.__a)
        for k in range(c):
            for i in range(n):
                for j in range(m):

                    delta2[i, k,:,:] += convolve2d(delta[i,j,:,:],
                                                   self.f[j,k,::-1,::-1])

        return delta2

    def update(self):

        self.optim.update([self.f, self.b],
                          [self.df, self.db])
```

Exercise 3: (3 points) Add the classes `Pool2DLayer()` and `FlattenLayer()` to `layers.py`.

a) `Pool2DLayer()` should be initialized by `Pool2DLayer(area)`, where `area` is a tuple (h, w) describing the window where to compute the maximum, e.g., $(2, 2)$, and store the following additional variables
   - `shape`: a tuple n, c, h, w of number of images, channels, height, and width of the (activated) images on input, initialized by `None`,
   - `mask`: a tensor of boolean values storing the information which indices have “won” the max pooling.

   Add the methods
   - `evaluate(self, a)`, store the dimensions of the input `a` in `shape`, reshape your input as necessary, save the information which indices have “won” in `mask` and return the result of the max pooling,
   - `backprop(self, delta)`, use the information in `mask` and suitable reshaping to return the correct `delta`.

Also add the method `add_pool2D(self, area, strict=False)` to our `SequentialNet` class.

b) `FlattenLayer()` should be initialized by `FlattenLayer()`. This layer should map input tensors of size (n, c, h, w) to a matrix of size $(n, c * h * w)$ to be consistent with the storage layout in `DenseLayer()`. It should only store
   - `shape`: a tuple n, c, h, w of number of images, channels, height, and width of the (activated) images on input, initialized by `None`.

   Add the methods
   - `evaluate(self, a)`, store the dimensions of the input **a** in `shape` and return a matrix consistent with the storage layout in `DenseLayer()`,
   - `backprop(self, delta)`, returns a suitable reshaped delta.

   Also add the method `add_flatten(self)` to our `SequentialNet` class.

c) Add the method
   - `update(self)`.

   to both classes. What does it do?

d) What happens in your backpropagation for the class `Pool2DLayer()`, when the evaluation before has identified more than one maximum in one area? Add the boolean parameter `strict` to your class `Pool2DLayer()`, that is either `True` or `False` with default value `False`. If it is true the backpropagation distributes the corresponding delta evenly over every index where the evaluation of the pooling layer had identified a maximum before by dividing the input delta by the number of maximums. If it is false, it does the same but without dividing it by the number of maximums.

Check your implementation by using the code already provided in `layers.py`.

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

```python
class Pool2DLayer:

    def __init__(self, area, strict=False):

        self.area   = area
        self.shape  = None
        self.mask   = None
        self.strict = strict

    def evaluate(self, a):

        n, c, h, w = self.shape = a.shape

        ph, pw = self.area
        nh, nw = h // ph, w // pw
        oh, ow = h % ph, w % pw

        # Reduce a, if hight and width are odd
        a_reduced  = a[:,:,:h-oh,:w-ow]

        # Reshape reduced a to find maxima
        a_reshaped = a_reduced.reshape(n, c, nh, ph, nw, pw)

        # Find maxima and create mask
        z = a_reshaped.max(axis=(3,5))
        z_newaxis = z[:,:,:,np.newaxis,:,np.newaxis]
        self.mask = (a_reshaped == z_newaxis)
```

```python

        return z

    def backprop(self, delta):

        n, c, h, w = self.shape
        ph, pw     = self.area

        # New delta should be of shape (n, c, h, w).
        # It can be recovered from the reshaped form
        da_reshaped = np.zeros((n, c, h//ph, ph, w//pw, pw))

        # Broadcast delta to shape of da_reshaped
        delta_newaxis      = delta[:,:,:,np.newaxis,:,np.newaxis]
        delta_broadcast, _ = np.broadcast_arrays(delta_newaxis, da_reshaped)

        # Set points, where maximums where found to 1
        da_reshaped[self.mask]  = delta_broadcast[self.mask]

        # Strict mode for correction of the subgradients in case of multiple maxima
        if self.strict:
            da_reshaped /= np.sum(self.mask, axis=(3,5), keepdims=True)

        # Build da from da_reshaped
        da = np.zeros(self.shape)
        dah, daw = h - h%ph, w - w%pw
        da[:,:,:dah,:daw] = da_reshaped.reshape(n, c, dah, daw)

        return da

    def update(self):

        """
         No update is done for Pooling layers. To keep the update method from
         SequentialNet, it is included here, but nothing is done
        """
        pass

class FlattenLayer:

    def __init__(self):

        self.shape = None

    def evaluate(self, a):

        self.shape = a.shape
        # The -1 allows numpy to figure out the second dimension automaticly
        return a.reshape(self.shape[0],-1)

    def backprop(self, delta):

        # Reshape delta from shape (n, c*h*w) to shape (n, c, h, w)
        return delta.reshape(self.shape)
```

```

    def update(self):

        pass
```

Exercise 4: (3 points) Develop a CNN for the MNIST dataset, using all layers implemented in the exercises above. You can test your network for example using the following skeleton and either our implementation or TensorFlow.

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

from   random            import randrange
# If you use TF:
# import tensorflow.keras as tfk

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

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

"""
TODO Implement the network you have developed for exercise 4
     Note that x_train and x_test are of shape (60000,28,28)
     and (10000,28,28). You need to add an additional axis.
     This can be done with np.newaxis, e.g
     x_test = x_test[:,np.newaxis,:,:]
"""
netz=None

y_tilde = netz.evaluate(x_test)
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()
```

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

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

from   random            import randrange

from networks    import SequentialNet
from layers      import *
```

```
from optimizers  import *
from activations import *

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

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

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


net = SequentialNet((1, 28,28))
net.add_conv2D((32,3,3), afun=ReLU(), optim=Adam(eta))
net.add_pool2D((2,2))
net.add_flatten()
net.add_dense(100, afun=ReLU(), optim=Adam(eta))
net.add_dense(10, afun=SoftMax(), optim=Adam(eta))

net.train(x, y, bs, ep)

y_tilde = net.evaluate(x_test[:,np.newaxis,:,:])
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()
```

A solution using TensorFlow can be found in the following listing.

```
import numpy             as np
import matplotlib.pyplot as plt
import tensorflow.keras  as tfk

from   random            import randrange

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

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

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

x = x_train[:,:,:,np.newaxis]
```

```
I = np.eye(10)
y = I[y_train,:]

layers = [tfk.layers.Conv2D(32, (3, 3),
                            input_shape=(28,28,1),
                            activation='relu'),
          tfk.layers.MaxPool2D((2,2)),
          tfk.layers.Flatten(),
          tfk.layers.Dense(100,
                           activation='relu'),
          tfk.layers.Dense(10,
                           activation='softmax')]

net = tfk.Sequential(layers)

opt = tfk.optimizers.Adam(learning_rate=eta)
net.compile(optimizer=opt,
            loss='categorical_crossentropy',
            metrics=['accuracy'])

net.fit(x, y, batch_size=bs, epochs=ep)

y_tilde = net.predict(x_test[:,:,:,np.newaxis])
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()
```