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

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

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)
- `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`.

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?

[1]Version from 01.11.2021

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`.

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.

```python
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()
```