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 6, solutions

Exercise 1: (4 points) Derive an FFT for $n = 3^k$.
Hint: Similar to the case $n = 2 \cdot m$ in the lecture notes on page 112[1] consider $n = 3 \cdot m$ and decompose the DFT of size n into three DFT's of size m.

Solution: According to the lecture, a DFT for $n = 3 \cdot m$ can be decomposed into three DFTs of size m. Let

$$\omega_\ell := e^{-\frac{2\pi i}{\ell}}, \quad \ell \in \mathbb{N}_0, \quad \omega_\ell^\ell = 1.$$

The DFT is given for $n = 3m$ by

$$\widehat{x}_k = \sum_{j=0}^{3m-1} (\omega_{3m}^k)^j x_j, \quad k = 0, \dots, 3m-1.$$

We group the indices j in the DFT into the three distinct classes

$$j = 0 \pmod 3, \qquad j = 1 \pmod 3, \qquad j = 2 \pmod 3,$$

and the indices k into m distinct classes, $k = q \pmod m$. Then for $j = 3r + \ell$ with $j = \ell \pmod 3$ and $k = pm + q$ with $k = q \pmod m$,

$$(\omega_{3m}^k)^j = \omega_{3m}^{(pm+q)(3r+\ell)} = \omega_{3m}^{(3mpr+mp\ell+3qr+q\ell)} = \omega_3^{p\ell} \cdot \omega_m^{qr} \cdot \omega_n^{q\ell}.$$

The DFT can be expressed as

$$\begin{aligned}
\widehat{x}_k &= \sum_{j=0}^{3m-1} (\omega_{3m}^k)^j x_j = \sum_{\ell=0}^{2} \omega_3^{p\ell} \cdot \omega_n^{q\ell} \sum_{r=0}^{m-1} \omega_m^{qr} \cdot x_{3r+\ell} \\
&= \begin{cases}
\sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r} + \omega_3^0 \cdot \omega_n^q \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+1} + \omega_3^0 \cdot \omega_n^{2q} \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+2}, & p = 0, \\
\sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r} + \omega_3^1 \cdot \omega_n^q \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+1} + \omega_3^2 \cdot \omega_n^{2q} \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+2}, & p = 1, \\
\sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r} + \omega_3^2 \cdot \omega_n^q \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+1} + \omega_3^4 \cdot \omega_n^{2q} \sum_{r=0}^{m-1} (\omega_m^q)^r x_{3r+2}, & p = 2.
\end{cases}
\end{aligned}$$

With

$$\omega_3^0 = 1, \quad \omega_3^1 = \omega_3^4 = \omega_3, \quad \omega_3^2 = \overline{\omega_3}$$

[1]Version from 01.11.21

we see that

$$\mathbf{F}_{3m}[:,0,3,6,\ldots,1,4,7,\ldots,2,5,8,\ldots] = \begin{pmatrix} \mathbf{I}_m & \mathbf{\Omega}_m & \mathbf{\Omega}_m^2 \\ \mathbf{I}_m & \omega_3\mathbf{\Omega}_m & \overline{\omega_3}\mathbf{\Omega}_m^2 \\ \mathbf{I}_m & \overline{\omega_3}\mathbf{\Omega}_m & \omega_3\mathbf{\Omega}_m^2 \end{pmatrix} \begin{pmatrix} \mathbf{F}_m & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{F}_m & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{F}_m \end{pmatrix},$$

where

$$\mathbf{\Omega}_m := \mathsf{diag}(\omega_n^0, \omega_n^1, \ldots, \omega_n^{m-1}).$$

A recursive implementation is given in the following listing.

```python
import numpy as np
from scipy.linalg import dft

def fft3(x):
    n = len(x)
    if n == 1:
        y = x
    else:
        y = np.zeros(n, dtype='complex')
        m = n//3
        omega = np.exp(-2*np.pi*1j/n)
        d = omega**np.arange(m)
        om3 = np.exp(-2*np.pi*1j/3)
        oc3 = om3.conj()
        z_top = fft3(x[0:n:3])
        z_mid = d * fft3(x[1:n:3])
        z_bot = d * d * fft3(x[2:n:3])
        y[0*m:1*m] = z_top +     z_mid +     z_bot
        y[1*m:2*m] = z_top + om3*z_mid + oc3*z_bot
        y[2*m:3*m] = z_top + oc3*z_mid + om3*z_bot
    return y

k = 3
x = np.random.randint(0, 10, 3**k)
y_fft3 = fft3(x)
print("Our variant =\n", y_fft3)
y = np.fft.fft(x)
print("NumPy's variant =\n", y)
y_dft = dft(3**k) @ x
print("Multiplication by SciPy's DFT matrix =\n", y_dft)
print("Difference between Numpy's and our variant =",
      np.linalg.norm(y-y_fft3))
```

Exercise 2: (4 points)

a) Count operations in the following two algorithms to compute the full convolution $\mathbf{y} = \mathbf{x} * \mathbf{f}$ of $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{f} \in \mathbb{R}^k$:
   (i) directly compute the convolution,
   (ii) enlarge the dimension of $\mathbf{x}$ and $\mathbf{f}$ to the smallest $2^\ell \geqslant n+k-1$ by appending trailing zeros and use the radix-2 FFT/IFFT (w/o the permutation).

b) Write a Python script that plots for given $n \in \mathbb{N}$ the operation count of both approaches. For which k is FFT better?

Hint: You can use the following Python code for part (ii) of a).

```
import numpy as np
from scipy.linalg import dft

def fft2(x):
    n = len(x)
    if n == 1:
        y = x
    else:
        y = np.zeros(n, dtype='complex')
        m = n//2
        omega = np.exp(-2*np.pi*1j/n)
        d = omega**np.arange(m)
        z_top = fft2(x[0:n:2])
        z_bot = d * fft2(x[1:n:2])
        y[0:m] = z_top + z_bot
        y[m:n] = z_top - z_bot
    return y
```

Solution: We suppose that $n \geqslant k$, otherwise we switch the inputs.

a) Computing the convolution directly involves k multiplications and $k-1$ additions for every of the $n-k+1$ outputs that completely overlap. A partial overlap occurs $2(k-1)$ times, the number of multiplications and additions range between $k-1$ and 1 and $k-2$ and zero, respectively, i.e,

$$M(n,k) = k(n-k+1) + 2\sum_{j=1}^{k-1} j = nk - k(k-1) + k(k-1) = nk,$$

$$A(n,k) = (k-1)(n-k+1) + 2\sum_{j=1}^{k-2} j = nk - (n+k-1) = (n-1)(k-1).$$

where M denotes the number of multiplications and A the number of additions. Counting one addition and one multiplication as one operation, we end up with $\mathcal{O}(nk)$ operations.

b) We count the operations in the recursive implementation of the following listing:

```
import numpy as np
from scipy.linalg import dft

def fft2(x):
    n = len(x)
    if n == 1:
        y = x
    else:
        y = np.zeros(n, dtype='complex')
        m = n//2
        omega = np.exp(-2*np.pi*1j/n)
        d = omega**np.arange(m)
        z_top = fft2(x[0:n:2])
        z_bot = d * fft2(x[1:n:2])
        y[0:m] = z_top + z_bot
        y[m:n] = z_top - z_bot
    return y

k = 5
```

```
x = np.random.binomial(10, .5, 2**k)
y_fft2 = fft2(x)
print("Our␣variant␣=\n", y_fft2)
y = np.fft.fft(x)
print("NumPy's␣variant␣=\n", y)
y_dft = dft(2**k) @ x
print("Multiplication␣by␣SciPy's␣DFT␣matrix␣=\n", y_dft)
print("Difference␣between␣Numpy's␣and␣our␣variant␣=",
      np.linalg.norm(y-y_fft2))
```

We call `fft2` with a vector of length $m = 2^\ell$. We do not count the integer division nor the computation of ω. We count the computation of the vector of powers of ω as $m/2$ multiplications, together with the $m/2$ elementwise multiplications with these we have m multiplications per call to `fft2`. The computation of the upper and lower half of $\mathbf{y}$ consists of m additions. We call `fft2` twice with a vector of size $m/2$, fourth with a vector of size $m/4$, i.e.,

$$\begin{aligned} M(m) &= m + 2m/2 + 4m/4 + \cdots + 2^{\ell-1}m/2^{\ell-1} = \ell 2^\ell, \\ A(m) &= M(m). \end{aligned}$$

As we map every vector that has more elements than $2^{\ell-1}$ to an extended version of size 2^ℓ, the function that counts the operations is piecewise linear. We call `fft2` twice, once for $\mathbf{x} \in \mathbb{R}^n$, once for $\mathbf{f} \in \mathbb{R}^k$. The componentwise multiplication adds 2^ℓ multiplications. We count the single call of `ifft2` as one call of `fft2`, thus we have in total $(3\ell+1)2^\ell$ multiplications and $3\ell 2^\ell$ additions, whch we count together as $(6\ell+1)2^\ell$ operations.

The two functions are thus given by

$$f_1(n,k) = nk + (n-1)(k-1), \quad f_2(n,k) = (6\ell+1)\ell 2^\ell, \quad \ell = \lceil \log(n+k-1) \rceil.$$

The following listing gives the plots for selected n and all $k = 1, \dots, n$.

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

for n in range(2,100):
    k = np.arange(1,n+1)

    direct = n*k + (n-1)*(k-1)
    ell = np.ceil(np.log2(n+k-1))
    fft = (6*ell+1)*(2**ell)

    print(direct.shape)

    plt.plot(k,direct,'r-',k,fft,'b.')
    plt.legend(['direct␣convolution','convolution␣via␣fft'])
    plt.title("n␣=␣{n}".format(n=n))
    if n == 80:
        plt.savefig('theory80.pdf')
    plt.show()
    plt.close()
```

For an example, see Figure 1.

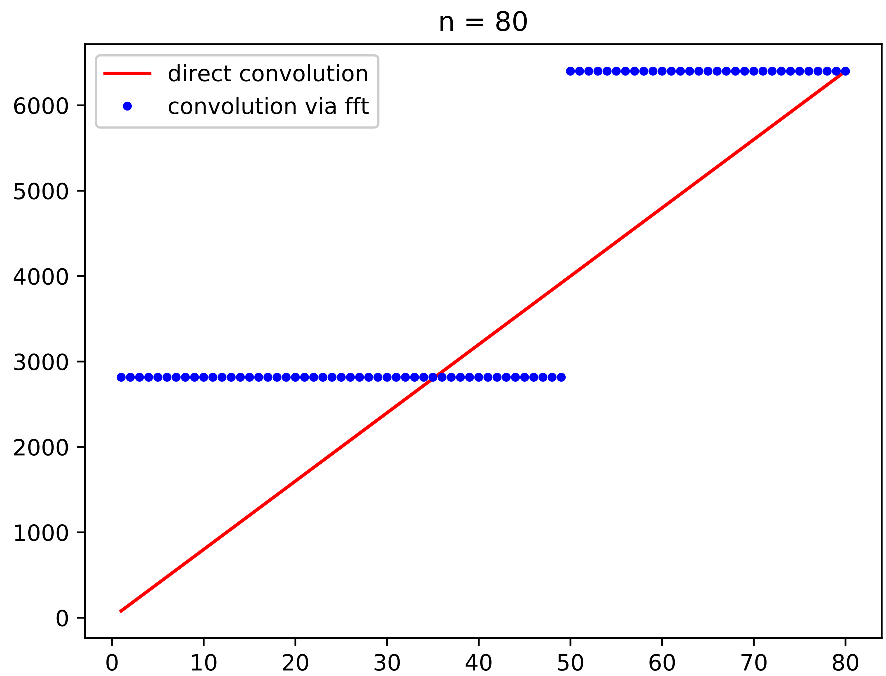

Figure 1: Theoretical comparison of direct and FFT-based convolution, $n = 80$.

Exercise 3: (4 points) In exercise 2 on the previous exercise sheet you implemented evaluation and backpropagation in a convolutional layer using the function `convolve2D()` of the SciPy package. Now we want to utilize the FFT approach and the `im2col` approach to calculate the convolution.

a) Add a method `evaluate_fft(self, a)` that implements the evaluation of a 2D convolutional layer with FFT and IFFT. Use the functions `rfft2` and `irfft2` from `scipy.fft`[2]

b) Add a method `evaluate_im2col(self, a)` that implements the evaluation of a 2D convolutionl layer with im2col. Use the functions `im2col` from the script `utils.py` to map the input to the corresponding im2col matrix. Use the attribute `cache` of `Conv2DLayer` to save the im2col matrix and the reshaped filterbank. Don't forget to reshape the result before applying the activation function.

c) Test your implementation with the code provided in `layers.py`

Solution: A possible solution for the evaluation with FFT and IFFT can be found in the following listing.

```
    def evaluate_fft(self, a):

      self.__a = a

      n, c, h, w = a.shape
      m, fh, fw  = self.fshape
      dh, dw     = fh - 1, fw - 1
      zh, zw     = h - dh, w - dw

```

[2]More information can be found at https://docs.scipy.org/doc/scipy/reference/tutorial/fft.html

```
      # Apply FFT
      a_hat = rfft2(a, (h + dh, w + dw))
      f_hat = rfft2(self.f, (h + dh, w + dw))

      self.__z = np.zeros((n, m, zh, zw))
      for i in range(n):
        for j in range(m):

          """
          Compute the full convolution with applying
          the componentwise product of a_hat and f_hat.
          z_hat then has the shape (c, h+dh, w+dw) and has
          to be restricted and summed over all channels.
          """
          z_hat = irfft2(a_hat[i,:,:,:] * \
                         f_hat[j,:,:,:],
                         (h + dh, w + dw))
          self.__z[i,j,:,:] = self.b[j] + \
            z_hat[:, dh:-dh, dw:-dw].sum(axis=0)

      return self.afun.evaluate(self.__z)
```

The evaluation with im2col is shown in the following listing.

```
    def evaluate_im2col(self, a):

        self.__a = a

        n, c, h, w = a.shape
        m, fh, fw = self.fshape
        zh, zw = h - fh + 1, w - fw + 1

        # Create im2col matrix
        a_col = im2col(a, fh, fw)
        # reshape filterbank
        f_row = self.f[:,:,::-1,::-1].reshape(m, -1)
        self.cache = a_col, f_row

        # BLAS Level 3: GEMM
        z_mat = f_row @ a_col + self.b.reshape(-1, 1)

        # reshape
        self.__z = z_mat.reshape(m, n, zh, zw).transpose(1, 0, 2, 3)
        return self.afun.evaluate(self.__z)
```

Exercise 4: (4 points) Compare our implementation of a neural network with TensorFlow.

a) Develop a convolutional neural network for the Fashion MNIST dataset. Try to achieve at least 80 percent accuracy.
b) Implement the network with TensorFlow and our library.
c) Compare the time needed for **one** epoch of training in both cases.Use `im2col` for evaluation in our implementation. The time for the TensorFlow network is printed while the network is trained.

You may use the following skeleton provided in `skeleton.py`.

```
import numpy              as np
```

```python
import matplotlib.pyplot as plt
import tensorflow.keras  as tfk

from random import randrange
from time   import time      # For time measuring

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

DATA = np.load('fashion_mnist.npz')
x_train, y_train = DATA['x_train'].reshape(60000,28,28), DATA['y_train']
x_test,  y_test  = DATA['x_test'].reshape(10000,28,28), DATA['y_test']
x_train, x_test  = x_train / 255.0, x_test / 255.0

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

bs, ep, eta = 128, 10, .001
"""
Categories for Fashion MNIST. Category i is ct[i].
"""
ct = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
      'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot']

"""
TODO Set up the network with our library
"""
net = SequentialNet((1,28,28))

"""
Add the layers to your network. As first hidden layer
you can use for example:

net.add_conv2D((32,3,3),
               afun=ReLU(),
               optim=Adam(),
               initializer=HeUniform(),
               eval_method='im2col')
"""

"""
The Last layer should be a SoftMax Layer with 10 neurons
"""
net.add_dense(10, afun=SoftMax(),
              optim=Adam(),
              initializer=HeUniform())
"""
TODO Set up the network with TensorFlow
"""
input_shape = (28,28,1)
net_TF = tfk.Sequential()

net_TF.add(tfk.Input(shape=input_shape))
"""
```

```
Add the same layers as above to the network.
If you used a convolutional layer with 32 3x3 filters
as first layer, you can add it with
net_TF.add(tfk.layers.Conv2D(32, (3,3),
                             activation='relu',
                             kernel_initializer='he_uniform'))
"""

net_TF.add(tfk.Dense(10, activation='softmax',
                     kernel_initializer='he_uniform'))

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


start = time()
net.train(x, y_train, batch_size=bs, epochs=1)
t_train = time() - start

"""
TODO Train the TensorFlow network. With metrics=['accuracy'] you
     get the time needed for training one epoch.
"""

y_test = np.argmax(y_test, 1).T



y_tilde_TF = net_TF.predict(x_test.reshape(10000, 28, 28, 1))
guess_TF   = np.argmax(y_tilde_TF, 1).T
print('Accuracy␣with␣TensorFlow␣=', np.sum(guess_TF == y_test)/100)
```

Solution: A solution 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 time   import time      # For time measuring

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

DATA = np.load('fashion_mnist.npz')
x_train, y_train = DATA['x_train'].reshape(60000,28,28), DATA['y_train']
x_test,  y_test  = DATA['x_test'].reshape(10000,28,28), DATA['y_test']
x_train, x_test  = x_train / 255.0, x_test / 255.0

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

```

```
"""
Categories for Fashion MNIST. Category i is ct[i].
"""
ct = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
      'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle␣Boot']

"""
Set up the network with our library
"""
bs, ep, eta = 128, 10, .001


net = SequentialNet((1,28,28))
net.add_conv2D((32,3,3),
               afun=ReLU(),
               optim=Adam(eta),
               initializer=
               HeUniform(),
               eval_method='im2col')
net.add_pool2D((2,2))
net.add_conv2D((64,3,3),
               afun=ReLU(),
               optim=Adam(eta),
               initializer=HeUniform(),
               eval_method='im2col')
net.add_pool2D((2,2))
net.add_flatten()
net.add_dense(2048,
              afun=ReLU(),
              optim=Adam(eta),
              initializer=HeUniform())
net.add_dense(10,
              afun=SoftMax(),
              optim=Adam(eta),
              initializer=HeUniform())


"""
Set up the network with TensorFlow
"""
activation = 'relu'
input_shape=(28,28,1)

net_TF = tfk.Sequential()

net_TF.add(tfk.Input(shape=input_shape))
net_TF.add(tfk.layers.Conv2D(32, (3,3),
                             activation=activation,
                             kernel_initializer='he_uniform'))
net_TF.add(tfk.layers.MaxPool2D(pool_size=(2,2)))
net_TF.add(tfk.layers.Conv2D(64, (3,3),
                             activation=activation,
                             kernel_initializer='he_uniform'))
net_TF.add(tfk.layers.MaxPool2D(pool_size=(2,2)))
net_TF.add(tfk.layers.Flatten())
net_TF.add(tfk.layers.Dense(2048, activation=activation,
```

```
                                kernel_initializer='he_uniform'))
net_TF.add(tfk.layers.Dense(10, activation='softmax',
                                kernel_initializer='he_uniform'))

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

start = time()
"""
net.train(x, y_train, batch_size=bs, epochs=ep)
"""
t_train = time() - start

start = time()
net_TF.fit(x_TF, y_train, batch_size=bs, epochs=ep)
t_train_TF = time() - start

y_test = np.argmax(y_test, 1).T

y_tilde_TF = net_TF.predict(x_test.reshape(10000,28,28,1))
guess_TF   = np.argmax(y_tilde_TF, 1).T
print('Accuracy with TensorFlow =', np.sum(guess_TF == y_test)/100)
print('Time needed for training:', t_train_TF)

# Our implementation: 3630s (60,5 min)
# Tensorflow: 206 s (3,4 min)
```

In my case, the network needed approx 3600s (60 min) for training 10 epochs, while TensorFlow only needed approx 200s (3,3 min).If you are able to run CUDA on your system, TensorFlow can be significantly faster.