In this laboratory, you are going to implement recurrent neural networks (RNNs) and, more specifically, long short-term memory (LSTM) layers using Keras.

Task 1: Regression problem – prediction of stock prices

To start with, you will work on a task that is unrelated to medical imaging, but still a good example to start implementing a basic LSTM architecture.

You are provided with two files (*/DL_course_data/Lab5/train_data_stock.csv* and */DL_course_data/Lab5/val_data_stock.csv*), which contain information on the price of a certain stock across time. In this task, you will simply focus on the column "Open", which indicates the starting price of a stock on a certain trading day (indicated in the column "Date"). You can load and manipulate these data by using the library *pandas*:

```
import pandas as pd
# load csv files:
dataset_train = pd.read_csv('/DL_course_data/Lab5/train_data_stock.csv')
dataset_val = pd.read_csv('/DL_course_data/Lab5/val_data_stock.csv')
# reverse data so that they go from oldest to newest:
dataset_train = dataset_train.iloc[::-1]
dataset_val = dataset_val.iloc[::-1]
# concatenate training and test datasets:
dataset_total = pd.concat((dataset_train['Open'], dataset_val['Open']),
axis=0)
# select the values from the "Open" column as the variables to be predicted:
training_set = dataset_train.iloc[:, 1:2].values
val_set = dataset_val.iloc[:, 1:2].values
```

With this dataset, data normalization (between 0 and 1) is also recommended and it can be performed on the training dataset using *scikit-learn*:

```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range=(0, 1))
training_set_scaled = sc.fit_transform(training_set)
```

You want to predict the price of a stock at date D+1, knowing its value from date D-T to D, where T is the total number of time steps. In this task, you can set T=60. The final training and validation datasets are therefore obtained as:

```
# split training data into T time steps:
X_train = []
y_train = []
for i in range(T, len(training_set)):
    X_train.append(training_set_scaled[i-T:i, 0])
    y_train.append(training_set_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)
# normalize the validation set according to the normalization applied to the
training set:
inputs = dataset_total[len(dataset_total) - len(dataset_val) - 60:].values
inputs = inputs.reshape(-1, 1)
inputs = sc.transform(inputs)
# split validation data into T time steps:
X_val = []
```

```
for i in range(T, T + len(val_set)):
    X_val.append(inputs[i-T:i, 0])
X_val = np.array(X_val)
y_val = sc.transform(val_set)
# reshape to 3D array (format needed by LSTMs -> number of samples,
timesteps, input dimension)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_val = np.reshape(X_val, (X_val.shape[0], X_val.shape[1], 1))
```

Now you are finally ready to implement and compile your regression model. You can define a model having four LSTM layers (they can be imported from *keras.layers*), each one with 20 units (dimensionality of their output space) and each followed by a dropout layer. You can set the dropout rate to 0.2. As you have seen in the lecture, you can set *return_sequence=True* in order to use the whole output sequence in the subsequent layers, as well as *stateful = True* if you want to maintain a dependency between the states of different batches. When setting *stateful = True*, you are required to specify *batch_shape=(n_batches, input_size, input_dimension)* in your input tensor. In this task, you can set a batch size of 16.

You can compile the model by using an Adam optimizer with learning rate = 0.001, mean squared error as loss function and mean absolute error as evaluation metric. You can run 100 training epochs. How good is your model? Can you observe any difference in performance if you increase (e.g. double) the number of output units of the LSTM layers?

If you want to have a more precise estimation of the prediction error of your model on the validation data after training, you can apply the inverse of the normalization transformation:

```
predicted_stock_price = model.predict(X_val)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

Task 2: Classification problem – fiber bundle classification

You are now going to extend the LSTM network implementation of Task 1 to adapt it to a different problem, i.e. the classification of neural fiber tracts in the brain by receiving as input the coordinates of their points (reconstructed through tractography from diffusion MRI data). In particular, you are going to train a model that distinguishes whether an input fiber tract belongs to the left or the right corticospinal tract.

You are provided with the fiber tracts of four different subjects (IDs 599469, 599671, 601127 and 613538) from the Human Connectome Project dataset (folder: */DL_course_data/Lab5/HCP_lab/*). You should use three subjects as training set and the remaining one as validation set. The bundles of interest are saved in each of the subjects' folders as *CST_left.trk* and *CST_right.trk*. You can use the code below, which makes use of the library *nibabel*, to load the streamlines of interest and their corresponding class (0 = left, 1 = right). The function *load_streamlines* allows you to extract *n_tracts_per_bundle* tracts (selected randomly from the whole bundle) from each of the bundles of every subject. In this way, you are going to have a balanced training set. You can set *n_tracts_per_bundle* = 20. Every streamline is represented by a sequence of different 3D points (x, y and z coordinates) of variable length.

```
dataPath = '/DL_course_data/Lab5/HCP/'
train_subjects_list = # your choice of 3 training subjects
```

```
val_subjects_list = # your choice of 1 validation subjects
bundles_list = ['CST_left', 'CST_right']
X_train, y_train = load_streamlines(dataPath, train_subjects_list,
bundles_list, n_tracts_per_bundle)
X_val, y_val = load_streamlines(dataPath, val_subjects_list, bundles_list,
n_tracts_per_bundle)

import nibabel as nib
def load_streamlines(dataPath, subject_ids, bundles, n_tracts_per_bundle):
    X = []
    y = []
    for i in range(len(subject_ids)):
        for c in range((len(bundles))):
            filename = dataPath + subject_ids[i] + '/' + bundles[c] + '.trk'
            tfile = nib.streamlines.load(filename)
            streamlines = tfile.streamlines

            n_tracts_total = len(streamlines)
            ix_tracts = np.random.choice(range(n_tracts_total),
n_tracts_per_bundle, replace=False)
            streamlines_data = streamlines.data
            streamlines_offsets = streamlines._offsets

            for j in range(n_tracts_per_bundle):
                ix_j = ix_tracts[j]
                offset_start = streamlines_offsets[ix_j]
                if ix_j < (n_tracts_total - 1):
                    offset_end = streamlines_offsets[ix_j + 1]
                    streamline_j = streamlines_data[offset_start:offset_end]
                else:
                    streamline_j = streamlines_data[offset_start:]
                X.append(np.asarray(streamline_j))
                y.append(c)

    return X, y
```

Once the data are loaded, you can proceed with the training (by using the same network architecture of Task 1). However, you will need to address a few more issues first:

- The tract data have all different lengths, so it is not possible to provide the model with an input tensor with a fixed input size and to train batches of several tracts together. One way to solve this issue consists of setting the input shape to **(*None*, 3)** where *None* refers to the varying tract length, while 3 is the dimension of each tract point (x, y, z). In this way, the LSTM model will accept batches of different length, as long as all data inside a batch have the exact same length. In order to be sure that all tracts in a batch have same length, you can set ***batch_size*=1**. Then, you can build a custom batch generator:

```
from tensorflow.keras.utils import Sequence
class MyBatchGenerator(Sequence):

    def __init__(self, X, y, batch_size=1, shuffle=True):
        self.X = X
        self.y = y
```

```
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.on_epoch_end()

    def __len__(self):
        'Get number of batches per epoch'
        return int(np.floor(len(self.y)/self.batch_size))

    def __getitem__(self, index):
        return self.__data_generation(index)

    def on_epoch_end(self):
        'Shuffle indexes after each epoch'
        self.indexes = np.arange(len(self.y))
        if self.shuffle == True:
            np.random.shuffle(self.indexes)

    def __data_generation(self, index):
        Xb = np.empty((self.batch_size, *self.X[index].shape))
        yb = np.empty((self.batch_size, 1))
        for s in range(0, self.batch_size):
            Xb[s] = self.X[index]
            yb[s] = self.y[index]
        return Xb, yb
```

This custom batch generator is going to be used in this way during the model fitting:

```
model.fit_generator(MyBatchGenerator(X_train, y_train, batch_size=1),
epochs=n_epochs, validation_data=MyBatchGenerator(X_val, y_val,
batch_size=1), validation_steps=len(X_val))
```

- The input data are bidirectional, i.e. a tract can be described in the same meaningful way by reading it both from start to end and from end to start. Therefore, you will need to extend your LSTM model by making your first LSTM layer bidirectional. This can be done by simply using the Keras Bidirectional layer wrapper, which will take as an argument the LSTM layer of interest.
- This is a classification problem, so you should now choose a different loss function and metric compared to Task 1!

For this task, you can reduce the number of units of the LSTM layers to 10, as well as the number of epochs to 50. How good is your model? What happens if you further reduce the number of units?

Bonus task 1: the method of using always a batch size of 1 works, but it is also rather inefficient. Another way of approaching the problem of input data with different lengths consists of using padding and masking. The shorter sequences are padded with a special value (which you do not expect to find in a normal tract sequence) in order to reach a specific length L. In this way, all tracts will have the same length L. Then, when implementing your model, you will need to add a Masking layer before the LSTM layer. The Masking layer will search for the "special value" that you set as padding value and disregard that part of the input data. As a bonus task, you can try to implement this alternative solution.

Bonus task 2: in each subject folder, you will find the *.trk* files. Each file corresponds to a specific fiber bundle. You can select a couple of other bundles that you would like to automatically classify by using your model, and check whether you can optimize your model implementations so that it can work well with other classification tasks too. In this webpage you can see which fiber bundles the filenames correspond to: https://github.com/MIC-DKFZ/TractSeg. Please note that not all bundles are present in the folders due to memory limitations, so you should make sure that all subject folders contain the bundles that you would like to classify.

Task 3: Another U-Net extension

As a final task, you will need to go back to your U-Net implementation (from Lab 4), with the following parameters:

- Image size = 240 × 240
- Number of filters at the first layer = 16
- Optimizer = Adam
- Learning rate = 1e-5
- Drop-out rate = 0.2
- Batch size = 8
- Evaluation metrics = Dice coefficient, precision, recall
- Batch normalization = True

A recent paper by Azad et al. (*Bi-Directional ConvLSTM U-Net with Densely Connected Convolutions*) proposed an extension of the U-Net by adding bi-directional convolutional LSTMs, which combine the feature maps from the encoding path and the previous decoding up-convolutional feature maps in a non-linear way. The idea behind this is that feature maps from the encoding layer have higher resolution, while feature maps extracted from the up-convolutional layer contain more semantic information. Therefore, rather than simply concatenating them, the use of ConvLSTMs allows to implement a non-linear function between the two in order to preserve more information, which can also result in a more precise segmentation output. ConvLSTMs differ from standard LSTMs because they introduce convolution operations into the input-to-state and the state-to-state transitions.

In this paper other changes are proposed too, but, for this lab task, you can simply try to add these LSTM layers to your U-Net architecture. Once you have done this, you can see whether you can observe some changes in the performance in the brain tumor classification task that you got from Lab 4, or whether this extension does not help in this specific task.

You can keep the same network parameters used for Lab 4. The contracting path and bottleneck portions of the architecture should be the same as well, while the new changes should be introduced in the expanding path. After each transpose convolution, you will need to reshape the output feature maps that you would normally concatenate, in order to use them as inputs to your ConvLSTM2D layer. Only after this reshaping, they can be concatenated and fed into the ConvLSTM2D. You can use the following example code (from the first transpose convolution

step at the beginning of the expanding path) as a guide also for the next layers. Please note that *n_filters* refers to the number of filters at the base layer, which is set to 16.

```
# up-sampling:
up6 = Conv2DTranspose(n_filters * 8, (3, 3), strides=(2, 2),
padding='same')(conv5)
# reshaping:
x1 = Reshape(target_shape=(1, np.int32(img_size / 8), np.int32(img_size / 8),
n_filters * 8))(conv4)
x2 = Reshape(target_shape=(1, np.int32(img_size / 8), np.int32(img_size / 8),
n_filters * 8))(up6)
# concatenation:
up6 = concatenate([x1, x2], axis=1)
# LSTM:
up6 = ConvLSTM2D(n_filters*4, (3, 3), padding='same', return_sequences=False,
go_backwards=True)(up6)
# the function conv2d_block implements the usual convolutional block with 2
convolutional layer:
conv6 = conv2d_block(u6, n_filters=n_filters * 8, kernel_size=3, batchnorm=
True)
```

The flag *go_backwards=True* allows to generate an output by using data dependencies in both directions (i.e. forward and backward pass together, instead of using only the dependencies of the forward direction).

Once you are ready, you can train the network also using data augmentation and analyze your segmentation accuracy.

Good luck!