Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Aprendizagem com dados não estruturados

Deep Learning - 2nd assignment

Authors:
Lucas Fischer
Joana Martins

Professor:
Rui Rodrigues

June 2019

1 Introduction

1.1 Transfer Learning

Transfer learning allows us to take advantage of previously built networks to make new neural networks based on previously acquired knowledge. For example, we can use part of a previous network that learned low level features and apply it to a new problem with similar low-level features. Another possible application is to use part of a network that has been trained in a set of examples in a network which will be used in another set of examples of the same kind. This is the goal of the first part of this assignment. We train a network in the MNIST dataset and then use part of it in a new network model to identify a set of handwritten digits. Since both datasets contain images of the same kind: handwritten digits, one expects there to be some common low-level features than can be learned from one dataset and reused in another network applied to the second dataset.

1.2 Time series (ECG)

The interpretation of 12-lead ECG is a very important topic since it can allow for real-time arrhythmia detection. An essential part of this interpretation is the detection of the so-called QRS complex (see a pictorial representation of these in 1).

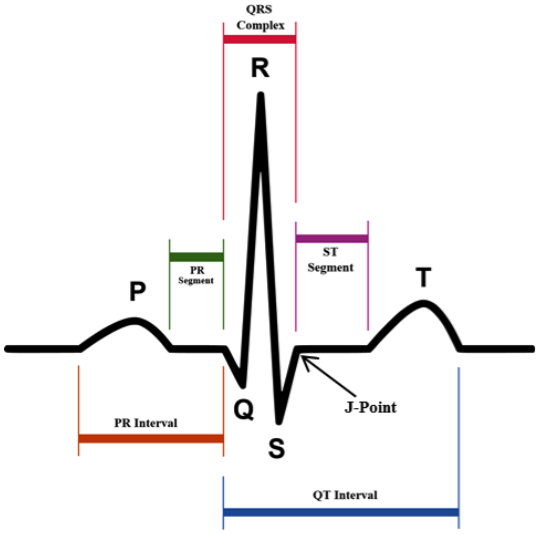

Figure 1. QRS complex (figure taken from [3]).

In this work, we implement a QRS detection algorithm using two types of neural networks: a feedforward network and a recurrent neural network. For training we use the INCART dataset [1] and for testing the mitbih Arrhythmia dataset [1].

The INCART dataset contains the signals from Holter records, with 12 channels and the INCART dataset contains ECG records with 2 channels only. In our analysis we use the first for training and validation and the latter for testing. Because of the different sampling frequencies in the two datasets, resampling of the data from the INCART dataset is done in order to match the sampling frequency of this to that of the mitbih Arrhythmia dataset.

It should be noted that, since these signals are from measures taken from real patients, some preprocessing may be required. As Pan & Tompkins refer [2], ECG recordings from patients can present noise from multiple sources such as motion of the electrodes, interference from nearby power lines, high-frequency T-waves and baseline wander.

In this work we make use of the Python waveform-database (WFDB) package [4] which contains useful functions to read the signals and annotations from the ECG datasets, functions needed for the resampling of the data and for comparing the predicted peak positions with the target ones, among others.

2 Part 1 - Transfer Learning

This work is divided in three parts. First, we build a neural network and train it on the MNIST dataset. Secondly, we build a new network reusing the weights obtained in the training of the first network and apply it to our new dataset of handwritten letters. Finally, we compare the accuracy of the network built from scratch and the network which reuses the weights of the first.

We start by building and training our neural network, which is a convolution network composed of convolution layers combined with fully connected layers for multi-class classification. The convolution layers are responsible for extracting different features of the digits and the fully connected layer combines the *knowledge* from them to classify the input data. In this type of networks, the input size is fixed.

2.1 Network with weight reuse

We start by loading the MNIST dataset and reshaping the data:

```
((train_x, train_y), (test_x, test_y)) = keras.datasets.mnist.load_data()
train_x = train_x.reshape((train_x.shape[0], 28, 28, 1))
test_x = test_x.reshape((test_x.shape[0], 28, 28, 1))
```

Each example from the dataset has 28x28 pixels, each with a given value for the black and white (BW) channel, so the reshaped matrix for each example will be 28x28x1. To normalize the values of the inputs to 1 we divide its values by 255:

```
train_x = train_x.astype("float32") / 255.0
test_x = test_x.astype("float32") / 255.0
```

Since the goal of our network is to do multi-class classification, we need to do one-hot encoding such that the training and testing labels have as many elements as the number of digits, ten, so that the result of the classifier is a vector where all elements are zero except for the one corresponding to the digit obtained by the classifier:

```
train_y = keras.utils.to_categorical(train_y, 10)
test_y = keras.utils.to_categorical(test_y, 10)
```

For practical purposes, we create a placeholder for the input data that has the dimensions we will use in both networks:

```
inputs = Input(shape=(28, 28, 1), name="inputs")
```

Our input layer is a 2D convolution layer followed by a RELU activation and normalization:

```
layer = Conv2D(32, (3, 3), padding="same", input_shape=(28, 28, 1))(inputs)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
```

where we used the inputs placehold to fix the shape of the inputs which will be passed to it. This layer has 32 filters and we use `padding="same"` to padd the input data with values taken from the input matrix itself instead of zeros. The RELU activation function is used so that the result of these layers becomes nonlinear and the network can do more than just linear combinations of the input data in the learning process. The second argument of the convolution layer, `(3, 3)` sets the size of the kernel used in the convolution, in this case, 3x3.

This layer is followed by another convolution layer with exactly the same parameters:

```
layer = Conv2D(32, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
```

This portion of the network is followed by a MaxPooling layer which decreases the number of features by a factor of two in each dimension, so that we pass from a matrix of 28x28 elements to one with 14x14:

```
layer = MaxPooling2D(pool_size=(2, 2))(layer)
```

A dropout layer is then added for regularization purposes. Turning off some of the neurons (in this case 25%) at a time, can prevent some overfitting:

```
layer = Dropout(0.25)(layer)
```

Additional layers as described above follow, with twice the filters as before (64 instead of 32):

```
layer = Conv2D(64, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)

layer = Conv2D(64, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)

# After this pooling we have a 7*7 tensor
layer = MaxPooling2D(pool_size=(2, 2))(layer)
layer = Dropout(0.25)(layer)
```

This last MaxPooling layer decreased again by a factor of two our feature number so that we now have 7x7 features instead of the initial 28x28.

The fully connected part of the network, where the classification will be done, requires a 1D input so we need a Flatten layer that converts our tensor of features in a vector:

```
features = Flatten(name="features")(layer)
```

Finally, the layers of the fully connected part of our neural network are as follows:

```
layer = Dense(512)(features)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)
layer = Dense(10)(layer)
layer = Activation("softmax")(layer)
```

Importantly, the activation function at the end must be of the type softmax since this function results in a probability for each of our possible labels. This is a typical activation function in multi-class classification problems. If one then desires to obtain the predicted class for a particular example, this can be achieved by taking the class for which the probability is the highest.

The code described above details the architecture of our neural network. It is then necessary to specify the inputs and outputs of our model:

```
old_model = Model(inputs=inputs, outputs=layer)
```

compile it:

```
old_model.compile(optimizer=SGD(lr=INIT_LR, momentum=0.9, decay=INIT_LR / NUM_EPOCHS),
    loss="categorical_crossentropy", metrics=["accuracy"])
```

and train it:

```
history = old_model.fit(train_x, train_y, validation_data=(test_x, test_y), batch_size=BS,
    epochs=NUM_EPOCHS, callbacks=[tensorboard_callback])
```

Since we are interested in doing transfer learning by reusing the weights in a new network to be used with another dataset of handwritten digits, it is necessary to save them. This is done with the instruction:

```
old_model.save_weights('./files/mnist_model.h5')
```

To reuse these weights one needs to reload them and prevent the network from changing them, so that what was learned is not *lost*, i.e., the network should not retrain the previously trained part of the network:

```
old_model.load_weights("./files/mnist_model.h5")
for layer in old_model.layers:
    layer.trainable = False
```

The layers of our new model are specified in the following lines:

```
layer = Dense(512)(old_model.get_layer("features").output)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)
layer = Dense(256)(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dense(128)(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.2)(layer)
layer = Dense(26)(layer)
layer = Activation("softmax")(layer)
```

In the first layer of our new model (first line of the above block of code), the inputs used are the output of the previously saved model: `old_model.get_layer("features").output`. This connects the first layer of our model to the old model.

Finally, in the initialization of the new model the use of the output of the old model as input is also set: `model = Model(inputs=old_model.get_layer("inputs").output, outputs=layer)`.

The new model is compiled in the line:

```
model.compile(optimizer=SGD(lr=1e-2, momentum=0.9), loss="categorical_crossentropy",
    metrics=["accuracy"])
```

where `SGD` is the steepest gradient descent optimizer, `lr` is the learning rate, and the loss is measured by the categorical cross entropy.

To train the network with the new data dataset, we need to load its features and labels, apply reshaping:

```
new_train_x = new_train_x.reshape((new_train_x.shape[0], 28, 28, 1))
new_test_x = new_test_x.reshape((new_test_x.shape[0], 28, 28, 1))
```

and do one-hot encoding as before:

```
new_train_y = keras.utils.to_categorical(new_train_y, 26)
new_test_y = keras.utils.to_categorical(new_test_y, 26)
```

The only difference is that, while the size of the images is the same (28x28 pixels), there are now 26 labels instead of 10.

The model was trained:

```
fitting = model.fit(new_train_x, new_train_y, batch_size=NEW_BS, epochs=NEW_NUM_EPOCHS,
    callbacks=[tensorboard_callback])
```

and the test error was measured with the data reserved for testing:

```
model.evaluate(x = new_test_x, y = new_test_y)
```

For the parameters used, a batchsize of 20 and 100 epochs, the following results were obtained:

Table 1. Loss and accuracy of the network with reused weigths

Test Loss	Test Accuracy
0.986	0.780

2.2 Network built from scratch

The neural network trained from scratch has the same layers as the previous network but is trained as a whole with the second dataset. For the same parameters (batchsize=20 and 100 epochs), the loss and accuracy obtained for the test data is:

Table 2. Loss and accuracy of the network built from scratch

Test Loss	Test Accuracy
1.116	0.755

2.3 Conclusions

When comparing the loss and accuracy obtained from the network built from scratch with the one whose convolution part was previously trained with the MNIST dataset (see tables 1 and 2), a slight improvement in the accuracy and decrease in the loss is observed for the network with transfer learning.

However, more important than this improvement is the fact that in the network that reused the weights obtained from the training with the MNIST dataset, they were achieved in much less time: 1 ms per training step instead of the 4 ms/step observed for the network from scratch.

The speedup in achieving a similar accuracy is a result of the fact that the model whose convolution layers were pre-trained with the MNIST dataset has already learned some of the low-level features (edges, shapes) of handwritten digits. When the weights resulting from this training are re-used, since both datasets correspond to the same kind of images (handwritten digits), the model has already learned some of the features and does not have to train during as much time as in the case it is starting from scratch.

3 Part 2 - Time series (ECG)

3.1 Pre-processing

Before training the feedforward neural network and the recurrent neural network some pre-processing of the ECG data files was necessary. Since this process can take a non-negligible amount of time, and should not have to be repeated every time we wish to train a network, we have made a separate jupyter notebook script to carry out this step and record new files with the processed training and test data (`QRS_preprocess.ipynb`). It involves changing the sampling frequency of the training dataset so that it matches that of the test data (they have sampling frequencies of 257 and 360 respectively) and removing the baseline of the signals.

For the training data, the pre-processing step involves first cycling through the data files, reading the signals two channels of interest (II and V1):

```
signal, info = wfdb.rdsamp(file_path, channel_names = ["II", "V1"])
```

and resampling it so that the data point sampling frequency matches that of the test data:

```
signal, annotations = processing.resample_multichan(signal, annotations, 257, 360)
```

We also need to build the target data. For that, the peaks that correspond to QRS complexes, which are denoted by the letters N,L,R,B,A,a,J,S,V,r,F,e,j,n,E,/,f,Q,? are identified in the annotations list and a target vector is built which is zero everywhere except at the QRS complexes positions, where we add a parabola:

```
symbol_positions = annotations.sample
symbol_list = annotations.symbol

# Filtering out all the lines that do not have a QRS symbol
qrs_symbs = ['N','L','R','B','A','a','J','S','V','r','F','e','j', 'n', 'E', '/', 'f', 'Q',' ?']
qrs_symbol_positions = [symbol_positions[idx] for idx, symb in enumerate(symbol_list) if symb in
    qrs_symbs]
...
target_vec = parabola(qrs_symbol_positions, len(signal), 3)
```

We have also opted to remove QRS complex positions that have negative positions.

Finally, we use build a dictionary where we store our feature values and the generated target values and use the `pickle` library to record them in new processed files for future use in the neural networks:

```
output_dict = {
    "channelII": signal_II,
    "channelV1": signal_V1,
    "label": target_vec
}
pkl.dump(output_dict, open(output_file_name, "wb"), protocol=pkl.HIGHEST_PROTOCOL)
```

For the test data, a similar process was implemented with the small difference that in this case there is no need to identify the channels to read as, unlike the training data, this only has two channels and also no need to resample the data.

3.2 Feedforward network

The first network implemented was a simple Feedforward neural network. This network is not the most suited for the type of data used in this assignment but it was used for baselining purposes.

The network is created using `Keras` sequential API and consists of 4 layers, one input-layer with 64 neurons, two hidden-layers with 64 neurons each and an output-layer with as many neurons as the multiplication of the length of the sequence times the number of inputs used.

```
ffwdModel = tf.keras.Sequential()
ffwdModel.add(layers.Dense(64, activation='relu',input_shape=(2*seqL*ninputs,)))
ffwdModel.add(layers.Dense(64, activation='relu'))
ffwdModel.add(layers.Dense(64, activation='relu'))
ffwdModel.add(layers.Dense(seqL*ninputs))
```

Having established the architecture of our network we can move on to compile it in order to correctly initialize it in the tensorflow graph and fit it to our training data in order to obtain the trained network with the learned weights. After each epoch is passed we validate our network with the validation data in order to mitigate the effects of overfitting.

```
ffwdModel.compile(optimizer=tf.train.RMSPropOptimizer(0.001), loss='MSE',metrics=['mae'])
ffwdModel.fit(trainData,  epochs=15, steps_per_epoch=1000, validation_data=valData,
    validation_steps=100)
```

Once the network has been trained we can save its weights so that we don't have to repeat the process of training the network again.

```
ffwd_weights_file_path = weights_path + 'FFWD_weights.h5'
ffwdModel.save_weights(ffwd_weights_file_path)
```

The input to our neural network is a tensorflow Dataset consisting in three channels: two input channels and a target channel. In order to create this training dataset all training files are sent to the tensorflow Dataset, then the function `selectFrom1ecg` is used to obtain a random segment of the three channels from a random file. The two first input channels are concatenated together in order to obtain a one dimensional input array.

```
inpOutSegment = tf.random_crop(ecgBdata[random_file_idx],[numChan, segmentL])
if(feed_forward):
    channelII = inpOutSegment[0,:]
    channelV1 = inpOutSegment[1,:]
    target = inpOutSegment[2,:]
    inputs = tf.concat((channelII, channelV1), axis = -1)
    return inputs,target
```

It's important to note that the random segment retrieved from the different channels must correspond to the same span of time in all channels.

An analogous process is used to create the test set, and after obtaining the trained network we can obtain the predictions for the test set.

```
with tf.Session() as sess:
    sess.run(iterator.initializer)
    inp, targ = sess.run(next_element)
    targets.append(targ)
    print(f"Getting batch {i}")
output = model.predict(inp)
```

We retrieved the predictions of our model using 100 batches from the test set. The results for these predictions were obtained as described in subsection 3.4 and are displayed in table 3.

Table 3. Table describing the results obtained for running the feed forward neural network in 100 batches of the test set

Precision	False Positive Rate	Recall
0.8676 (190/219)	0.1324 (29/219)	0.0355 (190/5346)

3.3 Recurrent neural network

The second network implemented for this assignment was a recurrent neural network. The main idea behind these types of networks is to reuse the output of time instance t_{x-1} as input in time instance t_x. These types of networks are more suitable to sequence-like data as is the case of the data of the assignment.

The sequential API of `Keras` was also used to create the architecture of our network.

```
rnnModel = tf.keras.Sequential()
rnnModel.add(layers.CuDNNLSTM(units=numLstmUnits, return_sequences=True, input_shape =
    (seql_rnn, 2 * ninputs)))
rnnModel.add(layers.CuDNNLSTM(units=numLstmUnits, return_sequences=True))
rnnModel.add(layers.CuDNNLSTM(units=numLstmUnits, return_sequences=True))
rnnModel.add(layers.CuDNNLSTM(units=numLstmUnits, return_sequences=True))
rnnModel.add(layers.TimeDistributed(layers.Dense(ninputs)))
rnnModel.add(layers.Reshape((seql_rnn * ninputs, )))
```

The recurrent network consists of four LSTM layers that characterize it as a recurrent neural network. The layer `CuDNNLSTM` was used to take advantage of the GPU provided by Google's Colab.

The process of training and obtaining the predictions for this network is identical to the process done for the feed forward network. The inputs however have a different shape in this network. Since this network is considering the data to be a sequence we want the information of the first channel at time t_x to be followed by the information of the second channel at the same time t_x and so a concatenation of both channels was not possible. To overcome this situation a manipulation of the dataset was needed, where the input data matrix was first transposed so that the columns correspond to the channels and then this matrix was reshaped as to have as many rows as the desired sequence length and as many columns as needed to fulfil this requirement. Having the dataset with the correct shape we could train our network and proceed to obtain the predictions for the test set.

```
transposed = tf.transpose(inpOutSegment)
inputs = transposed[:, :-1]
target = transposed[:, -1]
inputs = tf.reshape(inputs, (seqL, -1))
# We need to re-transpose the target to turn int back into a one-row vector
target = tf.transpose(target)
return inputs, target
```

The results relative to the predictions for 100 batches from the test set were obtained using the method described in 3.4 and are listed in table 4.

Table 4. Table describing the results obtained for running the recurrent neural network in 100 batches of the test set

Precision	False Positive Rate	Recall
0.9985 (7731/7743)	0.0015 (12/7743)	0.6478 (7731/11934)

When comparing these results with the results present in table 3 it's possible to observe that the recurrent neural network is able to have a much higher recall then the feed forward network. This is due to the fact that a recurrent neural network is more suitable for this type of data.

3.4 Post-processing of predictions from networks

To obtain predictions for each of our networks for the test datasets we have built the following function:

```
def get_preds_targets(tf_dataset,model,nbatches = 100):
  predictions = []
  targets = []

  try:
    iterator = tf_dataset.make_initializable_iterator()
    next_element = iterator.get_next()

    for i in range(nbatches):
      with tf.Session() as sess:
        sess.run(iterator.initializer)
        inp, targ = sess.run(next_element)
        targets.append(targ)
```

```
            print(f"Getting batch {i}")
        output = model.predict(inp)
        predictions.append(output)

    except:
      print('something wrong!')
      pass

    return predictions, targets
```

In this way, we can send the test dataset and the network model for which we want to obtain the predictions (`testData_rnn` and `rnnModel` for the RNN and `testData` and `ffwdModel` for the FFWD network) and the number of batches to use.

We need this function to be able to recover the targets associated with each input batch so that we can then use the peak locations from the processed data and compare them with those of the corresponding predictions.

We obtain new input and target batches by calling the method `get_next()` from the object `iterator`. The inputs are used as an argument for the `predict()` method of the model, which returns the predicted QRS peaks signal. To record the predictions for all the batches, we append them and the targets to the lists `predictions` and `targets` respectively.

The predictions we obtain with our networks are quite noisy, and for this reason we need to post-process them before extracting the predicted peak positions.

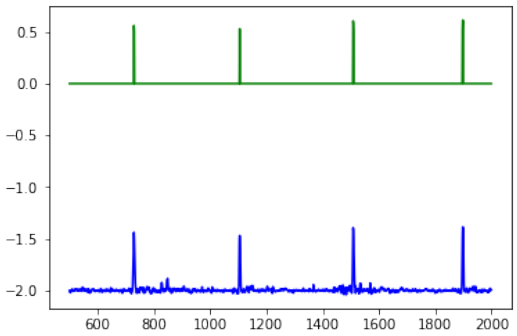

Figure 2. Example of predictions plot before post-processing and peaks obtained from predictions after applying threshold.

Since we need to apply this post-processing of the predictions to both networks, we have implemented a function that takes care of this process. In the following we describe the steps involved in this post-processing occuring inside the function `get_peaks`, that receives as arguments the predictions vector and the corresponding target vectors which were produced in the pre-processing phase, based on the annotations provided:

```
# First use a threshold to eliminate some smaller peaks
thresh = 0.5
peaks = np.where(predictions1D>thresh, predictions1D, 0)
```

The first step is to generate a new vector with the same length as the predictions vector `predictions1D` (blue line in Figure 2) but where we keep only the values above a given threshold (`thresh = 0.5`). This will remove all the noise and keep the predicted signal values for the peaks only (green line in Figure 2).

To actually find the locations of the peaks we then use the `find_local_peaks` method of the `wfdb.processing` library. However, this method also yields peak positions for intervals where the signal is constant. Therefore, many false peaks will be found in regions that are constant and equal to zero. To overcome this, it is necessary to remove from the peak locations list (`peak_locs`) all the peak locations for which the predictions value is above the threshold. The result of this is stored in the list `peak_locs_above_th`.

```
# Find local peaks in filtered predictions
peak_locs = find_local_peaks(peaks,8)

# Filter peak locations with condition that value of signal there must be
peak_locs_above_th = [p for p in peak_locs if peaks[p] > thresh]
```

To compare peak positions we need to provide both the predicted peak positions and the target peak positions. However, in the processed files we only kept the generated target channel, and not the peak locations. For this reason we apply the same post-processing to the target vector to recover the target peak positions.

```
peaks_targets1D = np.where(targets1D>thresh, targets1D, 0)
# Find local peaks in filtered predictions
targets_peak_locs = find_local_peaks(peaks_targets1D,8)
targets_peak_locs_above_th = [p for p in targets_peak_locs if peaks_targets1D[p] > thresh]

targets_peak_locs_above_th = np.array(targets_peak_locs_above_th)
peak_locs_above_th = np.array(peak_locs_above_th)
```

Having the peak locations for both the target and the predictions, we can now use the `compare_annotations` method to evaluate the precision and recall for both the networks. This is provided by the method `print_summary()`.

```
comparitor = compare_annotations(targets_peak_locs_above_th, peak_locs_above_th, 5, signal=None)
comparitor.print_summary()
```

This was done for both networks to obtain the results discussed above for each one.

References

[1] Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. Physiobank, physiotoolkit, and physionet. *Circulation*, 101(23):e215–e220, 2000. doi:10.1161/01.CIR.101.23.e215.

[2] J. Pan and W. J. Tompkins. A real-time qrs detection algorithm. *IEEE Transactions on Biomedical Engineering*, BME-32(3):230–236, March 1985. doi:10.1109/TBME.1985.325532.

[3] ACLS Medical Training. The basics of ecg, 2019. (Lecture slides). URL: https://www.aclsmedicaltraining.com/basics-of-ecg/.

[4] Chen Xie and Julien Dubiel. wfdb-python, 2019. URL: https://github.com/MIT-LCP/wfdb-python.