TP : Recurrent Neural Networks using Keras and Tensorflow

Mastère spécialisé

INSA Rouen

We will implement a Recurrent Neural Network (RNN) in Keras and apply it in a classification task and a sequence labeling task. For this second part, we will make use of the Connectionist Temporal Classification (CTC) for training the RNN in case of label sequences.

1 Introduction: First steps with Keras and Tensorflow

Tensorflow is an open source software library developed by Google for numerical computation using data flow graphs, where the nodes of the graph represent mathematical operations. Computations can be performed on one or more CPUs or GPUs. *Tensorflow* is well defined for applications in deep learning.

Keras is a higher-level library that uses *Tensorflow* (or another similar library as *Theano* or *CNTK*), and allows us to define neural network architectures in just a few lines of code.

Keras and Tensorflow are defined in a declaratory mode. One will first define a computation graph and then applied it on data.

2 Recurrent Neural Network for a classification task

In the virtual environment **/opt/venv/spyder/bin/activate**, open Spyder3 (for using Python3, Tensorflow and Keras).

2.1 Define the network architecture in Keras

We will experiment RNN in a classification task, i.e. where we have in input of our network a data represented as a time sequence and in output a unique label for the entire sequence. Here, experiments will be performed on the MNIST dataset which contains images of handwritten digits (Figure 1).

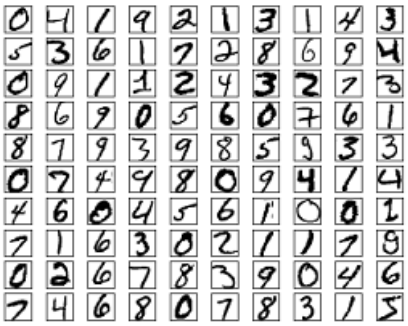

Figure 1: Images from the MNIST dataset.

A recurrent neural network is defined for dealing with sequential data by making use of the temporal context during the process. In addition to connections between neurons from different layers, there is connections between neurons of the same hidden layer.

One will define a function **classif_rnn_network** that build the computation graph related to a RNN for a classification task. Here, **all sequences have the same length**. The function

has two arguments: T, the length of the sequences, and D, the number of features. First, we will build a standard RNN using only one recurrent layer. Then we will improve the model with more depth and strongest types of layers.

1. In Keras, specific layers are defined as classes and can be imported from *keras.layers*. Import a *SimpleRNN* layer and call it in the function. Specify that there is 20 units.

2. In order to perform a classification task, we will use a fully connected layer as output layer. A fully connected layer is commonly called a *Dense* layer (and this is the name in Keras). To get probabilities in output, specify that the activation function is a *softmax*. The number of units is the number of classes. Define the *Dense* layer in your network architecture.

3. You have to specify that the *Dense* layer will be applied on the output of the *SimpleRNN* one. For that, add brackets containing the output of the *SimpleRNN* layer after initializing the *Dense* layer. It will be done as follow:

   ```
   out\_probs = Dense(...)(out_SimpleRNN)
   ```

4. Now one must specify the input of our network. This is done in Keras using an *Input* layer. Add it and indicate the shape of inputs (this is an argument of the *Input* layer). Note that the batch size (i.e. the number of data) is generally unknown when we build the computation graph. The dimensions is None and there is no need to specify it if one uses the argument *shape*.

   Be careful that the output of the *Input* layer is given in input of the *SimpleRNN* layer (for a correct network architecture).

5. We have defined the layers of our network and the succession of layers. To train the network and evaluate test data, there is a specific class in Keras, named *Model* which is well defined for such tasks. The *Model* class is imported from *keras.engine*. One provides as arguments the inputs and outputs of our network as follow:

   ```
   network = Model(inputs=my_inputs, outputs=out\_probs)
   ```

   where my_inputs is the output of the *Input* layer and *out_probs* is the output of the *Dense* layer.

6. Finally, we will configure the network using the method *compile*. One will specify here that we will use the *categorical_crossentropy* as loss, the *sgd* method (Stochastic Gradient Descent) for optimizing the parameters during training. We also specify that we want the *accuracy* as evaluation.

   ```
   network.compile(optimizer='sgd', loss='categorical_crossentropy',\
                                    metrics=['accuracy'])
   ```

To conclude the function **classif_rnn_network**, you have to return the network that have been compiled.

2.2 Apply the RNN on MNIST

We will apply the recurrent neural network defined in the **classif_rnn_network** function. The code will be now written in the main function. One recalls that the main function is defined as follow:

```
if __name__ == '__main__':
```

1. Import the MNIST dataset from *keras.datasets*. In the main file, you can load the dataset as follow:

   ```
   (x_train, y_train), (x_test, y_test) = mnist.load_data()
   ```

   where x_train and x_test are sets of images (considered as the observation sequences) and y_train and y_test are lists containing the label related to each image (i.e. a digit).

2. Get the dimensions T and D and the number of training examples.

3. Call the function classif_rnn_network to get the network structure.

4. In the class Model in Keras, there is a method *fit* which is defined for training a model for a fixed number of epochs. An epoch is a training iteration where all the training examples have been seen one time in training. Call the *fit* method that have x_train, y_train as arguments.

5. You will use the *evaluate* method of the Model class to get an evaluation of the trained model on the testing set.

6. Note that if you run the code, it doesn't work as there is a problem of shape in output. This is due to the fact that the network that we have defined requires for each observation a list of 0,1 values as label, not the digit value. The function *to_categorical* from the *keras.utils* package converts a list of integer to a matrix of 0 and 1 and will solve the problem.

7. Print the output of the evaluation method: It returns the loss and the accuracy.

You can now applied your network on the mnist dataset. Note that the accuracy is very poor (around 45%).

2.3 Improve the network

There is many ways to improve your network such as:

1. Increasing the number of units,

2. Adding other layers (note that for stacking layers, layers that are below another recurrent layer must returns a sequence, and this can be specified using the *return_sequences* argument),

3. Replacing the SimpleRNN layer by GRU or LSTM layers,

4. Increasing the number of training epochs (this is an argument of the fit method, by default, 10 epochs are performed).

You can now get an accuracy which is over than 95%. Note than it can take a long time!

3 Recurrent Neural Networks for sequence labeling

The Connectionist Temporal Classification (CTC) is defined to perform a sequence labeling, i.e. where the label sequence in output can be shorter than the number of time frames of the observation sequence. This is a realistic task as it is not common to have one label per time frame. This approach has some similarities with the well-known Hidden Markov Models, such as the Forward-Backward process for instance. CTC relies on a cost function computed on the entire sequence that one will call *ctc_loss* to the next.

First, define a new function **rnn_seqlab_network** to define a recurrent neural network for sequence labeling. Make a copy of the RNN defined for a classification task, one will modify it to perform the CTC.

3.1 Change the Network structure for dealing with CTC

1. Make some changes in the layers for dealing with CTC. Now, each layer of the network must return a set of outputs per observation frame. You can make use of the *TimeDistributed* layer in Keras to get a set of probabilities at each time frame.

   There is no Model in Keras defined to perform a CTC approach. However, in the backend of Keras, there is a function for computing the CTC cost function using Tensorflow. The CTC loss function from Tensorflow has 4 input tensors: the true labeling, the predicted labeling, the input lengths and the label lengths:

- true labeling = the ground truth, which is a sequence of labels per observation sequence,
- predicted labeling = the output of the recurrent network (i.e. the output of the *TimeDistributed(Dense)*) This is a matrix of probabilities, where each term is the probability of having a label at a time frame,
- input lengths = the observation sequence length for each sequence, i.e. T times the number of data,
- label lengths = the length of true labeling for each observation sequence.

A CTCModel has been implemented at the LITIS lab. It extends the Model class in Keras to have a model that perform the CTC approach. You will make use of a part of the CTCModel implementation.

2. Replace the model use in the classification task by a CTCModel in the *rnn_seqlab_network* function (define a model and compile it).

3.2 Apply CTC for sequence labeling

One will experiment the network on a sequence of digits. This is images from the MNIST dataset that have been concatenated to form sequences of 5 digits.

1. *Pickle* is a module to define specific object for Python. This is a way to save and load easily Python structures. Import pickle and load the data as follow:

```
(x_train,y_train),(x_test,y_test) = pickle.load(open('./seqDigits.pkl', 'rb'))
```

2. In a similar way than for the classification task, in the main function, one will train a CTCModel using the *fit* method. As seen before, to perform the CTC approach, one needs to have 4 inputs for argument x:
   (a) The *true labeling* is the labels of the training set (y_train),
   (b) The *x_train_len* contains the length of each training observation sequence (each value of T, here all the observation sequences have the same length bu it can not be the case) Create this array (you can create a list and convert it in array using *np.asarray()*,
   (c) The *y_train_len* contains the length of each training label sequence (each value is of length 5, here all the label sequences have the same length bu it can not be the case) Create this array (you can create a list and convert it in array using *np.asarray()*,
   (d) The *predicted_labeling* is the output of the *TimeDistributed(Dense)* layer. This is compute by the system from the observation given in input of the model. So one has to provide the observations *x_train* in input of the fit method.

   For training the model, provide both terms in a list and in the right order:

```
[x_train, y_train, x_train_len, y_train_len]
```

3. To conclude, call the predict method from CTCModel to predict the test sequences
4. Run the process. You can observe that the loss decrease during training. You can print the predicted label sequences with the ground truth (i.e. real label).

4 Dealing with variable length sequences

Until now, we have fixed the dimensions given in input of our networks (e.g. T and D in input of a RNN). However, this is common to have variable length input sequences. However, in Keras and Tensorflow, data are represented with Tensors that are of fixed size. The way for dealing with variable length sequences and fixed-size tensors in Keras and Tensorflow is to pad sequences to the longest one of the batch and to not considered the padded value using a specific layer called Masking.

1. First, one must to replace dimension T in the shape of the *Input* layer with *None*. T is now no longer required.

2. Define a **Masking** layer after the *Input* layer. Specify the *mask_ value*. This allow to no considered, at computation time, the times frame containing only the specific value *mask_ value*.

3. In the main file, use the method *pad_ sequences* from *sequence* in *keras.preprocessing* to pad observations and labelings structures. The padded structure are then given in input of the fit, predict or evaluate methods.

4. Load the file *seqDigitsVar.pkl* containing variable length sequences. This is sequences of digits from the MNIST dataset, having 2 to 5 digits. Train the model and predict a part of test sequences.

5 Increase the robustness of the network

5.1 Bidirectional network and optimization methods

1. **Bidirectional layers:** add a **Bidirectional** layer in your RNN. It has a type of neuron (a layer in Keras) as argument.

2. **Optimization:** There is many optimization techniques to optimize the model parameters. We have used the standard *Stochastic Gradient Descent (SGD)* but there is more recent techniques such as *RMSProp* or *Adam* that can be applied.

5.2 Preventing overfitting

1. **Dropout:** Dropout is a regularization technique for reducing overfitting in neural networks. It consists in dropping out a percentage of units in a neural network at each training iteration. The dropout can generally be used as argument of a layer. Apply a dropout on each recurrent layer.

2. **Gaussian noise:** Applying a Gaussian noise on the input observations is a way to prevent overfitting. **GaussianNoise** is a specific layer defined for such a task.

6 For deepening...

6.1 Error analysis

Three metrics are commonly used to evaluate a text recognition:

- Character Error Rate (CER)
- Word Error Rate (WER)
- Sequence Error Rate (SER)

The CER is based on the *edit-distance* measure (also called the *Levenshtein distance*) that computes the minimal cost to get the ground truth character sequence from the predicted character sequence. This consists of a number of insertions, substitutions and deletions. The cost is then divided by the length of the ground truth to get the Character Error Rate.

In the same way, one can compute the WER at a word-level. Finally, the SER is the error on the entire sequence. If the edit-distance is equal to 0 (i.e. there is no insertion, substitution or deletion) on the entire sequence, the sequence is well predicted, otherwise there is at least one error and it increase the sequence error rate. If sequences are words, the SER is the WER.

There is many way to compute those metrics as the edit distance can be computed in Python using the *edit-distance* package or in Tensorflow using *tf.edit_ distance*. The *evaluate* method in CTCModel can be used to compute the label error rate and sequence error rate (here this relates to the CER and WER metrics respectively). Even so, you can implement the metric computations in Python using the *edit-distance* package.

6.2 Standardize

Standardize (i.e. *centrer-réduire* in french) is a common way for preprocessing input data. This allows to made the features uniform and to get input values closed to zero (the values follow a Gaussian of mean 0 and variance 1). Compute the mean and variance of values on the training examples and standardize both the train and test data.

6.3 Tensorboard

Tensorboard is a visualization tool provided with TensorFlow. It is defined in *keras.callbacks*. It consists in writing a log for TensorBoard, which allows you to visualize especially the computation graph and evaluation metrics. First, you have to provide the callback *Tensorboard* as an argument of a *fit* method. It can be applied to the *fit* method of the model_train in the CTCModel. Otherwise, there is a specific CTCTensorBoard that can be used. Then, launch Tensorboard from a Terminal using the following command:

```
tensorboard --logdir=/full_path_to_your_tensorboardLogs
```