

7. Workshop 2-4 : IMDB - NLP - Sequence Data with Conv1D

[Reference] :

François Chollet, **Deep Learning with Python**, Chapter 6, Section 4, Manning, 2018. [http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf#\(http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf\)](http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf#(http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf))

Implementing a 1D convnet

In Keras, you would use a 1D convnet via the `Conv1D` layer, which has a very similar interface to `Conv2D`. It takes as input 3D tensors with shape `(samples, time, features)` and also returns similarly-shaped 3D tensors. The convolution window is a 1D window on the temporal axis, axis 1 in the input tensor.

Let's build a simple 2-layer 1D convnet and apply it to the IMDB sentiment classification task that you are already familiar with.

As a reminder, this is the code for obtaining and preprocessing the data:

In [1]:

```
1 import keras
2 keras.__version__
```

```
/Users/macmini1/anaconda3/lib/python3.6/site-packages/h5py/
__init__.py:36: FutureWarning: Conversion of the second
argument of issubdtype from `float` to `np.floating` is de
precated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
```

```
from ._conv import register_converters as _register_conv
erters
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

In [2]:

```
1 from keras.datasets import imdb
2 from keras.preprocessing import sequence
3
4 max_features = 10000 # number of words to consider as features
5 max_len = 500 # cut texts after this number of words (among top ma
6
7 print('Loading data...')
8 (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max
9 print(len(x_train), 'train sequences')
10 print(len(x_test), 'test sequences')
11
12 print('Pad sequences (samples x time)')
13 x_train = sequence.pad_sequences(x_train, maxlen=max_len)
14 x_test = sequence.pad_sequences(x_test, maxlen=max_len)
15 print('x_train shape:', x_train.shape)
16 print('x_test shape:', x_test.shape)
```

```
Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
x_train shape: (25000, 500)
x_test shape: (25000, 500)
```

1D convnets are structured in the same way as their 2D counter-parts that you have used in Chapter 5: they consist of a stack of `Conv1D` and `MaxPooling1D` layers, eventually ending in either a global pooling layer or a `Flatten` layer, turning the 3D outputs into 2D outputs, allowing to add one or more `Dense` layers to the model, for classification or regression.

One difference, though, is the fact that we can afford to use larger convolution windows with 1D convnets. Indeed, with a 2D convolution layer, a 3x3 convolution window contains $3 \times 3 = 9$ feature vectors, but with a 1D convolution layer, a convolution window of size 3 would only contain 3 feature vectors. We can thus easily afford 1D convolution windows of size 7 or 9.

This is our example 1D convnet for the IMDB dataset:

In [3]:

```
1 from keras.models import Sequential
2 from keras import layers
3 from keras.optimizers import RMSprop
4
5 model = Sequential()
6 model.add(layers.Embedding(max_features, 128, input_length=max_len))
7 model.add(layers.Conv1D(32, 7, activation='relu'))
8 model.add(layers.MaxPooling1D(5))
9 model.add(layers.Conv1D(32, 7, activation='relu'))
10 model.add(layers.GlobalMaxPooling1D())
11 model.add(layers.Dense(1))
12
13 model.summary()
14
15 model.compile(optimizer=RMSprop(lr=1e-4),
16               loss='binary_crossentropy',
17               metrics=['acc'])
18 history = model.fit(x_train, y_train,
19                     epochs=10,
20                     batch_size=128,
21                     validation_split=0.2)
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 500, 128)	1280000
conv1d_1 (Conv1D)	(None, 494, 32)	28704
max_pooling1d_1 (MaxPooling1D)	(None, 98, 32)	0
conv1d_2 (Conv1D)	(None, 92, 32)	7200
global_max_pooling1d_1 (GlobalMaxPooling1D)	(None, 32)	0
dense_1 (Dense)	(None, 1)	33

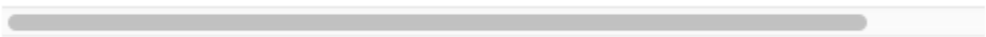
=====
Total params: 1,315,937
Trainable params: 1,315,937
Non-trainable params: 0

Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [=====] - 71s 4ms/ste

```

p - loss: 0.8337 - acc: 0.5091 - val_loss: 0.6875 - val_ac
c: 0.5622
Epoch 2/10
20000/20000 [=====] - 69s 3ms/ste
p - loss: 0.6700 - acc: 0.6408 - val_loss: 0.6641 - val_ac
c: 0.6604
Epoch 3/10
20000/20000 [=====] - 68s 3ms/ste
p - loss: 0.6233 - acc: 0.7548 - val_loss: 0.6075 - val_ac
c: 0.7444
Epoch 4/10
20000/20000 [=====] - 68s 3ms/ste
p - loss: 0.5251 - acc: 0.8093 - val_loss: 0.4845 - val_ac
c: 0.8058
Epoch 5/10
20000/20000 [=====] - 70s 3ms/ste
p - loss: 0.4084 - acc: 0.8486 - val_loss: 0.4289 - val_ac
c: 0.8300
Epoch 6/10
20000/20000 [=====] - 70s 4ms/ste
p - loss: 0.3468 - acc: 0.8651 - val_loss: 0.4216 - val_ac
c: 0.8354
Epoch 7/10
20000/20000 [=====] - 69s 3ms/ste
p - loss: 0.3076 - acc: 0.8610 - val_loss: 0.4356 - val_ac
c: 0.8218
Epoch 8/10
20000/20000 [=====] - 70s 4ms/ste
p - loss: 0.2745 - acc: 0.8545 - val_loss: 0.4236 - val_ac
c: 0.8134
Epoch 9/10
20000/20000 [=====] - 72s 4ms/ste
p - loss: 0.2523 - acc: 0.8363 - val_loss: 0.4328 - val_ac
c: 0.7912
Epoch 10/10
20000/20000 [=====] - 73s 4ms/ste
p - loss: 0.2272 - acc: 0.8111 - val_loss: 0.4927 - val_ac
c: 0.7560

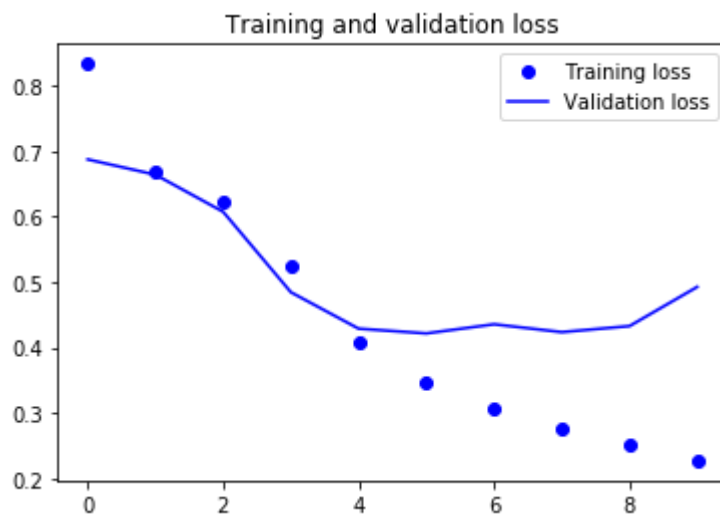
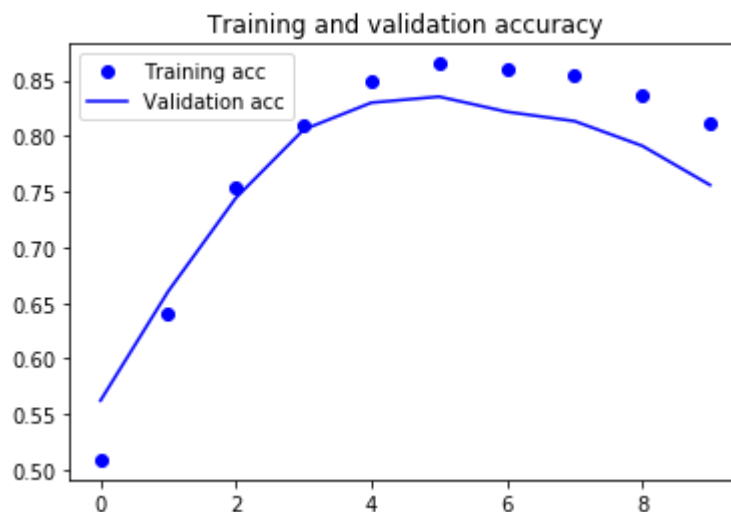
```



Here are our training and validation results: validation accuracy is somewhat lower than that of the LSTM we used two sections ago, but runtime is faster, both on CPU and GPU (albeit the exact speedup will vary greatly depending on your exact configuration). At that point, we could re-train this model for the right number of epochs (8), and run it on the test set. This is a convincing demonstration that a 1D convnet can offer a fast, cheap alternative to a recurrent network on a word-level sentiment classification task.

In [5]:

```
1 import matplotlib.pyplot as plt
2 %matplotlib inline
3
4 acc = history.history['acc']
5 val_acc = history.history['val_acc']
6 loss = history.history['loss']
7 val_loss = history.history['val_loss']
8
9 epochs = range(len(acc))
10
11 plt.plot(epochs, acc, 'bo', label='Training acc')
12 plt.plot(epochs, val_acc, 'b', label='Validation acc')
13 plt.title('Training and validation accuracy')
14 plt.legend()
15
16 plt.figure()
17
18 plt.plot(epochs, loss, 'bo', label='Training loss')
19 plt.plot(epochs, val_loss, 'b', label='Validation loss')
20 plt.title('Training and validation loss')
21 plt.legend()
22
23 plt.show()
```



Combining CNNs and RNNs to process long sequences

Because 1D convnets process input patches independently, they are not sensitive to the order of the timesteps (beyond a local scale, the size of the convolution windows), unlike RNNs. Of course, in order to be able to recognize longer-term patterns, one could stack many convolution layers and pooling layers, resulting in upper layers that would "see" long chunks of the original inputs -- but that's still a fairly weak way to induce order-sensitivity. One way to evidence this weakness is to try 1D convnets on the temperature forecasting problem from the previous section, where order-sensitivity was key to produce good predictions. Let's see:

Download `jena_climate_2009_2016.csv` dataset

- from Kaggle: jena_climate_2009_2016 <https://www.kaggle.com/stytch16/jena-climate-2009-2016/downloads/jena-climate-2009-2016.zip/1>
(<https://www.kaggle.com/stytch16/jena-climate-2009-2016/downloads/jena-climate-2009-2016.zip/1>)

In [6]:

```
1  # We reuse the following variables defined in the last section:
2  # float_data, train_gen, val_gen, val_steps
3
4  import os
5  import numpy as np
6
7  data_dir = './'
8  fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')
9
10 f = open(fname)
11 data = f.read()
12 f.close()
13
14 lines = data.split('\n')
15 header = lines[0].split(',')
16 lines = lines[1:]
17
18 float_data = np.zeros((len(lines), len(header) - 1))
19 for i, line in enumerate(lines):
20     values = [float(x) for x in line.split(',')[1:]]
21     float_data[i, :] = values
22
23 mean = float_data[:200000].mean(axis=0)
24 float_data -= mean
25 std = float_data[:200000].std(axis=0)
26 float_data /= std
27
28 def generator(data, lookback, delay, min_index, max_index,
29              shuffle=False, batch_size=128, step=6):
30     if max_index is None:
31         max_index = len(data) - delay - 1
32     i = min_index + lookback
33     while 1:
34         if shuffle:
35             rows = np.random.randint(
36                 min_index + lookback, max_index, size=batch_size)
37         else:
38             if i + batch_size >= max_index:
39                 i = min_index + lookback
40             rows = np.arange(i, min(i + batch_size, max_index))
41             i += len(rows)
42
43         samples = np.zeros((len(rows),
44                             lookback // step,
45                             data.shape[-1]))
46         targets = np.zeros((len(rows),))
47         for j, row in enumerate(rows):
48             indices = range(rows[j] - lookback, rows[j], step)
49             samples[j] = data[indices]
50             targets[j] = data[rows[j] + delay][1]
51         yield samples, targets
52
53 lookback = 1440
54 step = 6
55 delay = 144
56 batch_size = 128
57
```

```
58 train_gen = generator(float_data,
59                        lookback=lookback,
60                        delay=delay,
61                        min_index=0,
62                        max_index=200000,
63                        shuffle=True,
64                        step=step,
65                        batch_size=batch_size)
66 val_gen = generator(float_data,
67                    lookback=lookback,
68                    delay=delay,
69                    min_index=200001,
70                    max_index=300000,
71                    step=step,
72                    batch_size=batch_size)
73 test_gen = generator(float_data,
74                     lookback=lookback,
75                     delay=delay,
76                     min_index=300001,
77                     max_index=None,
78                     step=step,
79                     batch_size=batch_size)
80
81 # This is how many steps to draw from `val_gen`
82 # in order to see the whole validation set:
83 val_steps = (300000 - 200001 - lookback) // batch_size
84
85 # This is how many steps to draw from `test_gen`
86 # in order to see the whole test set:
87 test_steps = (len(float_data) - 300001 - lookback) // batch_size
```


In [7]:

```
1 from keras.models import Sequential
2 from keras import layers
3 from keras.optimizers import RMSprop
4
5 model = Sequential()
6 model.add(layers.Conv1D(32, 5, activation='relu',
7                           input_shape=(None, float_data.shape[-1])))
8 model.add(layers.MaxPooling1D(3))
9 model.add(layers.Conv1D(32, 5, activation='relu'))
10 model.add(layers.MaxPooling1D(3))
11 model.add(layers.Conv1D(32, 5, activation='relu'))
12 model.add(layers.GlobalMaxPooling1D())
13 model.add(layers.Dense(1))
14
15 model.compile(optimizer=RMSprop(), loss='mae')
16 history = model.fit_generator(train_gen,
17                               steps_per_epoch=500,
18                               epochs=20,
19                               validation_data=val_gen,
20                               validation_steps=val_steps)
```

Epoch 1/20

500/500 [=====] - 43s 86ms/step -
loss: 0.4226 - val_loss: 0.4450

Epoch 2/20

500/500 [=====] - 42s 84ms/step -
loss: 0.3637 - val_loss: 0.4434

Epoch 3/20

500/500 [=====] - 43s 86ms/step -
loss: 0.3388 - val_loss: 0.4633

Epoch 4/20

500/500 [=====] - 43s 87ms/step -
loss: 0.3233 - val_loss: 0.4415

Epoch 5/20

500/500 [=====] - 43s 86ms/step -
loss: 0.3120 - val_loss: 0.4391

Epoch 6/20

500/500 [=====] - 44s 87ms/step -
loss: 0.3019 - val_loss: 0.4467

Epoch 7/20

500/500 [=====] - 44s 87ms/step -
loss: 0.2932 - val_loss: 0.5030

Epoch 8/20

500/500 [=====] - 43s 85ms/step -
loss: 0.2871 - val_loss: 0.4702

Epoch 9/20

500/500 [=====] - 43s 87ms/step -
loss: 0.2809 - val_loss: 0.4588

Epoch 10/20

500/500 [=====] - 44s 88ms/step -
loss: 0.2771 - val_loss: 0.4509

Epoch 11/20

500/500 [=====] - 43s 86ms/step -
loss: 0.2731 - val_loss: 0.4478

Epoch 12/20

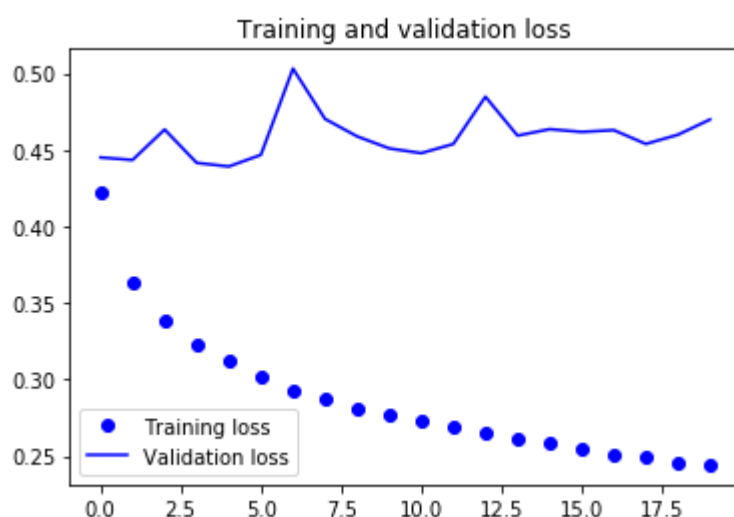
500/500 [=====] - 44s 87ms/step -
loss: 0.2691 - val_loss: 0.4538

```
Epoch 13/20
500/500 [=====] - 43s 87ms/step -
loss: 0.2648 - val_loss: 0.4847
Epoch 14/20
500/500 [=====] - 43s 87ms/step -
loss: 0.2614 - val_loss: 0.4593
Epoch 15/20
500/500 [=====] - 43s 86ms/step -
loss: 0.2585 - val_loss: 0.4635
Epoch 16/20
500/500 [=====] - 44s 88ms/step -
loss: 0.2549 - val_loss: 0.4617
Epoch 17/20
500/500 [=====] - 44s 87ms/step -
loss: 0.2513 - val_loss: 0.4629
Epoch 18/20
500/500 [=====] - 44s 87ms/step -
loss: 0.2491 - val_loss: 0.4538
Epoch 19/20
500/500 [=====] - 43s 86ms/step -
loss: 0.2460 - val_loss: 0.4598
Epoch 20/20
500/500 [=====] - 43s 87ms/step -
loss: 0.2443 - val_loss: 0.4698
```

Here are our training and validation Mean Absolute Errors:

In [8]:

```
1 import matplotlib.pyplot as plt
2 %matplotlib inline
3
4 loss = history.history['loss']
5 val_loss = history.history['val_loss']
6
7 epochs = range(len(loss))
8
9 plt.figure()
10
11 plt.plot(epochs, loss, 'bo', label='Training loss')
12 plt.plot(epochs, val_loss, 'b', label='Validation loss')
13 plt.title('Training and validation loss')
14 plt.legend()
15
16 plt.show()
```



The validation MAE stays in the low 0.40s: we cannot even beat our common-sense baseline using the small convnet. Again, this is because our convnet looks for patterns anywhere in the input timeseries, and has no knowledge of the temporal position of a pattern it sees (e.g. towards the beginning, towards the end, etc.). Since more recent datapoints should be interpreted differently from older datapoints in the case of this specific forecasting problem, the convnet fails at producing meaningful results here. This limitation of convnets was not an issue on IMDB, because patterns of keywords that are associated with a positive or a negative sentiment will be informative independently of where they are found in the input sentences.

One strategy to combine the speed and lightness of convnets with the order-sensitivity of RNNs is to use a 1D convnet as a preprocessing step before a RNN. This is especially beneficial when dealing with sequences that are so long that they couldn't realistically be processed with RNNs, e.g. sequences with thousands of steps. The convnet will turn the long input sequence into much shorter (downsampled) sequences of higher-level features. This sequence of extracted features then becomes the input to the RNN part of the network.

This technique is not seen very often in research papers and practical applications, possibly because it is not very well known. It is very effective and ought to be more common. Let's try this out on the temperature forecasting dataset. Because this strategy allows us to manipulate much longer sequences, we could either look at data from further back (by increasing the

lookback parameter of the data generator), or look at high-resolution timeseries (by decreasing the step parameter of the generator). Here, we will chose (somewhat arbitrarily) to use a step twice smaller, resulting in twice longer timeseries, where the weather data is being sampled at a rate of one point per 30 minutes.

In [9]:

```
1 # This was previously set to 6 (one point per hour).
2 # Now 3 (one point per 30 min).
3 step = 3
4 lookback = 720 # Unchanged
5 delay = 144 # Unchanged
6
7 train_gen = generator(float_data,
8                       lookback=lookback,
9                       delay=delay,
10                      min_index=0,
11                      max_index=200000,
12                      shuffle=True,
13                      step=step)
14 val_gen = generator(float_data,
15                    lookback=lookback,
16                    delay=delay,
17                    min_index=200001,
18                    max_index=300000,
19                    step=step)
20 test_gen = generator(float_data,
21                     lookback=lookback,
22                     delay=delay,
23                     min_index=300001,
24                     max_index=None,
25                     step=step)
26 val_steps = (300000 - 200001 - lookback) // 128
27 test_steps = (len(float_data) - 300001 - lookback) // 128
```

This is our model, starting with two Conv1D layers and following-up with a GRU layer:

In [10]:


```
1 model = Sequential()
2 model.add(layers.Conv1D(32, 5, activation='relu',
3                           input_shape=(None, float_data.shape[-1])))
4 model.add(layers.MaxPooling1D(3))
5 model.add(layers.Conv1D(32, 5, activation='relu'))
6 model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
7 model.add(layers.Dense(1))
8
9 model.summary()
10
11 model.compile(optimizer=RMSprop(), loss='mae')
12 history = model.fit_generator(train_gen,
13                               steps_per_epoch=500,
14                               epochs=20,
15                               validation_data=val_gen,
16                               validation_steps=val_steps)
```

Layer (type)	Output Shape	Param #
conv1d_6 (Conv1D)	(None, None, 32)	227
max_pooling1d_4 (MaxPooling1D)	(None, None, 32)	0
conv1d_7 (Conv1D)	(None, None, 32)	515
gru_1 (GRU)	(None, 32)	624
dense_3 (Dense)	(None, 1)	33

Total params: 13,697
Trainable params: 13,697
Non-trainable params: 0

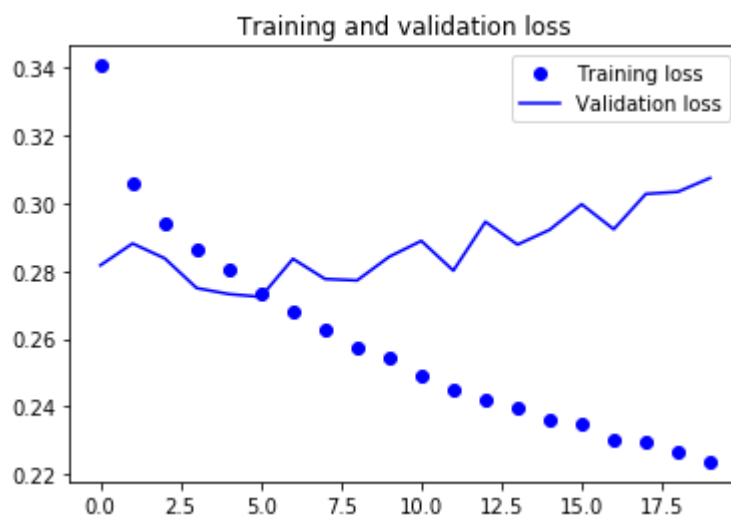
Epoch 1/20
500/500 [=====] - 86s 173ms/step
- loss: 0.3406 - val_loss: 0.2817
Epoch 2/20
500/500 [=====] - 85s 169ms/step
- loss: 0.3059 - val_loss: 0.2881
Epoch 3/20
500/500 [=====] - 84s 169ms/step
- loss: 0.2941 - val_loss: 0.2837
Epoch 4/20
500/500 [=====] - 85s 171ms/step

```
- loss: 0.2863 - val_loss: 0.2749
Epoch 5/20
500/500 [=====] - 85s 170ms/step
- loss: 0.2802 - val_loss: 0.2732
Epoch 6/20
500/500 [=====] - 85s 170ms/step
- loss: 0.2735 - val_loss: 0.2723
Epoch 7/20
500/500 [=====] - 85s 169ms/step
- loss: 0.2681 - val_loss: 0.2836
Epoch 8/20
500/500 [=====] - 85s 170ms/step
- loss: 0.2625 - val_loss: 0.2776
Epoch 9/20
500/500 [=====] - 84s 169ms/step
- loss: 0.2573 - val_loss: 0.2772
Epoch 10/20
500/500 [=====] - 88s 176ms/step
- loss: 0.2545 - val_loss: 0.2842
Epoch 11/20
500/500 [=====] - 87s 174ms/step
- loss: 0.2489 - val_loss: 0.2889
Epoch 12/20
500/500 [=====] - 87s 175ms/step
- loss: 0.2452 - val_loss: 0.2801
Epoch 13/20
500/500 [=====] - 91s 182ms/step
- loss: 0.2420 - val_loss: 0.2945
Epoch 14/20
500/500 [=====] - 89s 178ms/step
- loss: 0.2397 - val_loss: 0.2878
Epoch 15/20
500/500 [=====] - 91s 182ms/step
- loss: 0.2360 - val_loss: 0.2922
Epoch 16/20
500/500 [=====] - 92s 184ms/step
- loss: 0.2351 - val_loss: 0.2997
Epoch 17/20
500/500 [=====] - 91s 181ms/step
- loss: 0.2300 - val_loss: 0.2923
Epoch 18/20
500/500 [=====] - 89s 178ms/step
- loss: 0.2294 - val_loss: 0.3028
Epoch 19/20
500/500 [=====] - 88s 176ms/step
- loss: 0.2268 - val_loss: 0.3034
Epoch 20/20
500/500 [=====] - 84s 168ms/step
- loss: 0.2237 - val_loss: 0.3074
```



In [11]:

```
1 loss = history.history['loss']
2 val_loss = history.history['val_loss']
3
4 epochs = range(len(loss))
5
6 plt.figure()
7
8 plt.plot(epochs, loss, 'bo', label='Training loss')
9 plt.plot(epochs, val_loss, 'b', label='Validation loss')
10 plt.title('Training and validation loss')
11 plt.legend()
12
13 plt.show()
```



Judging from the validation loss, this setup is not quite as good as the regularized GRU alone, but it's significantly faster. It is looking at twice more data, which in this case doesn't appear to be hugely helpful, but may be important for other datasets.

Wrapping up

Here's what you should take away from this section:

- In the same way that 2D convnets perform well for processing visual patterns in 2D space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular NLP tasks.
- Typically 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of `Conv1D` layers and `MaxPooling1D` layers, eventually ending in a global pooling operation or flattening operation.
- Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before a RNN, shortening the sequence and extracting useful representations for the RNN to process.

One useful and important concept that we will not cover in these pages is that of 1D convolution with dilated kernels.

