Introduction to 1D Convolutional Neural Networks in Keras for Time Sequences

Nils Ackermann Follow
Sep 4, 2018 · 7 min read

Introduction

Many articles focus on two dimensional convolutional neural networks. They are particularly used for image recognition problems. 1D CNNs are covered to some extend, e.g. for natural language processing (NLP). Few articles provide an explanatory walkthrough on how to construct a 1D CNN though for other machine learning problems that you might be facing. This article tries to bridge this gap.

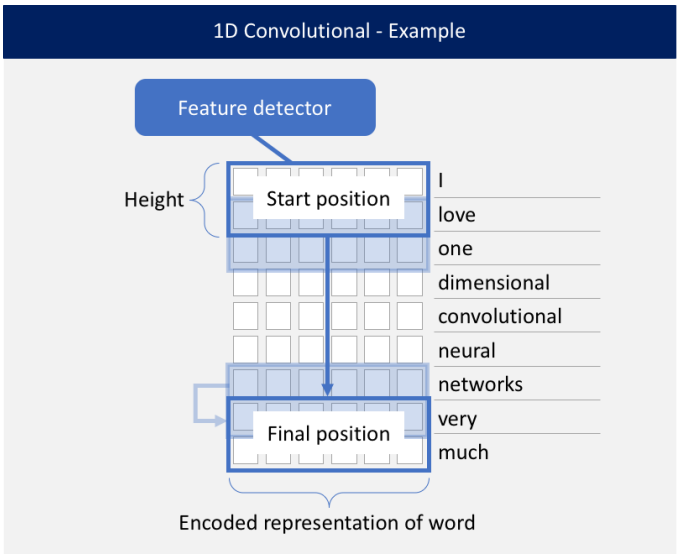
When to Apply a 1D CNN?

A CNN works well for identifying simple patterns within your data which will then be used to form more complex patterns within higher layers. A 1D CNN is very effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall data set and where the location of the feature within the segment is not of high relevance.

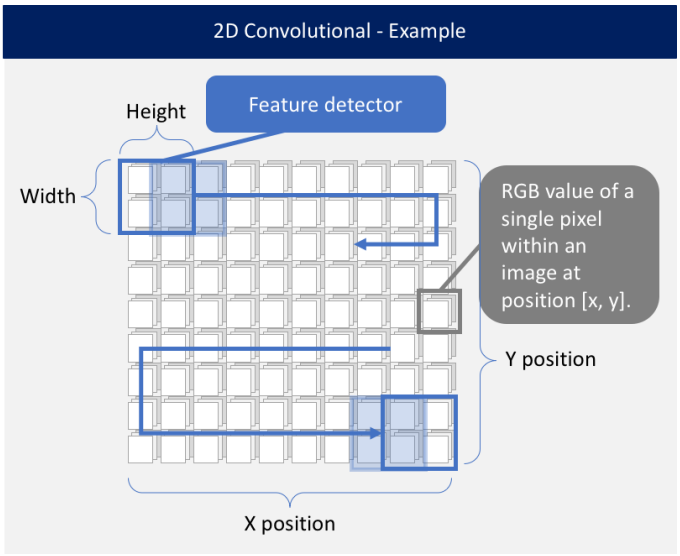
This applies well to the analysis of time sequences of sensor data (such as gyroscope or accelerometer data). It also applies to the analysis of any kind of signal data over a fixed-length period (such as audio signals). Another application is NLP (although here LSTM networks are more promising since the proximity of words might not always be a good indicator for a trainable pattern).

What is the Difference Between a 1D CNN and a 2D CNN?

CNNs share the same characteristics and follow the same approach, no matter if it is 1D, 2D or 3D. The key difference is the dimensionality of the input data and how the feature detector (or filter) slides across the data:



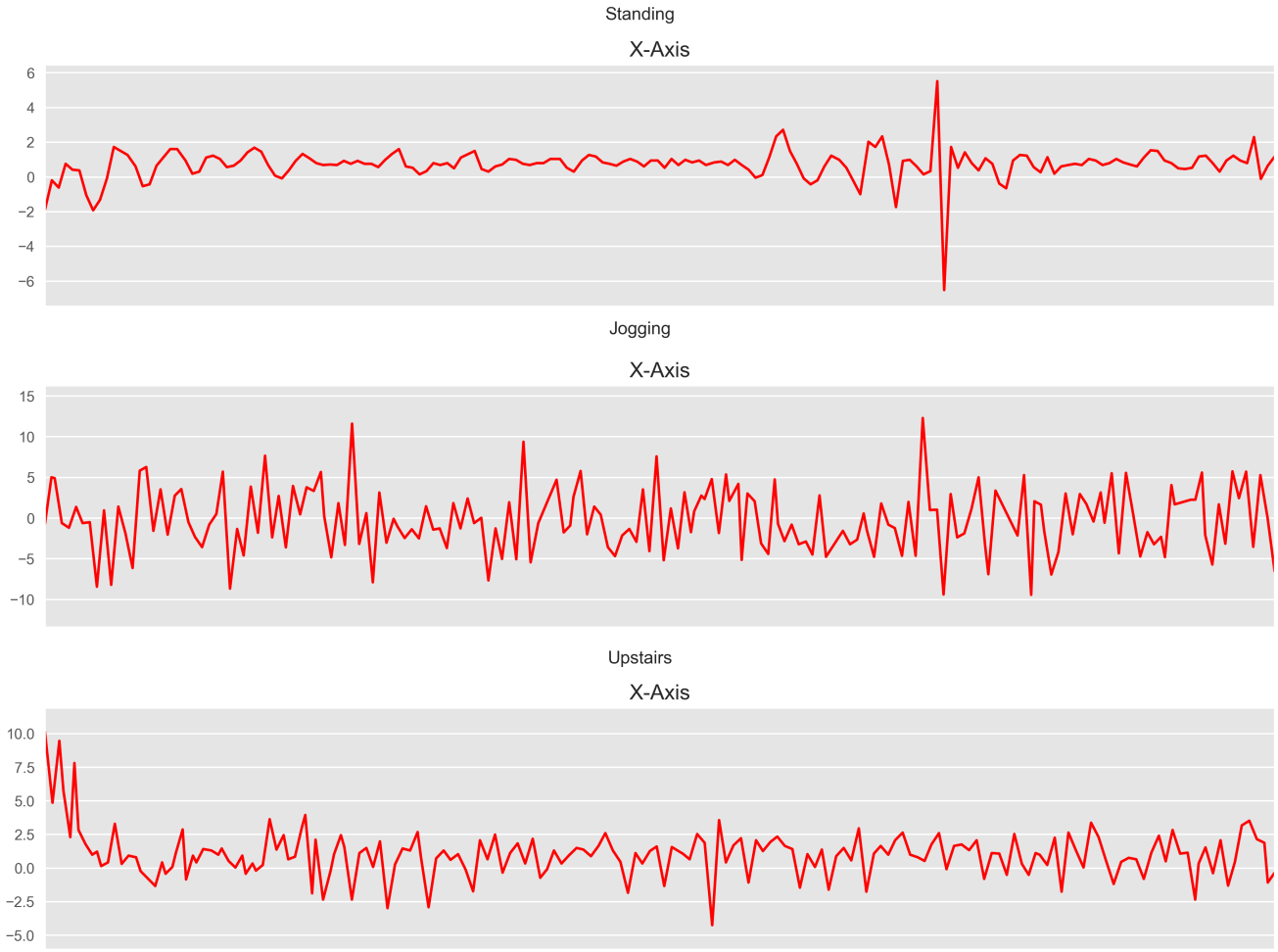
In this example for natural language processing, a sentence is made up of 9 words. Each word is a vector that represents a word as a low dimensional representation. The feature detector will always cover the whole word. The height determines how many words are considered when training the feature detector. In our example, the height is two. In this example the feature detector will iterate through the data 8 times.



In this example for computer vision, each pixel within the image is represented by its x- and y position as well as three values (RGB). The feature detector has a dimension of 2 x 2 in our example. The feature detector will now slide both horizontally and vertically across the image.

Problem Statement

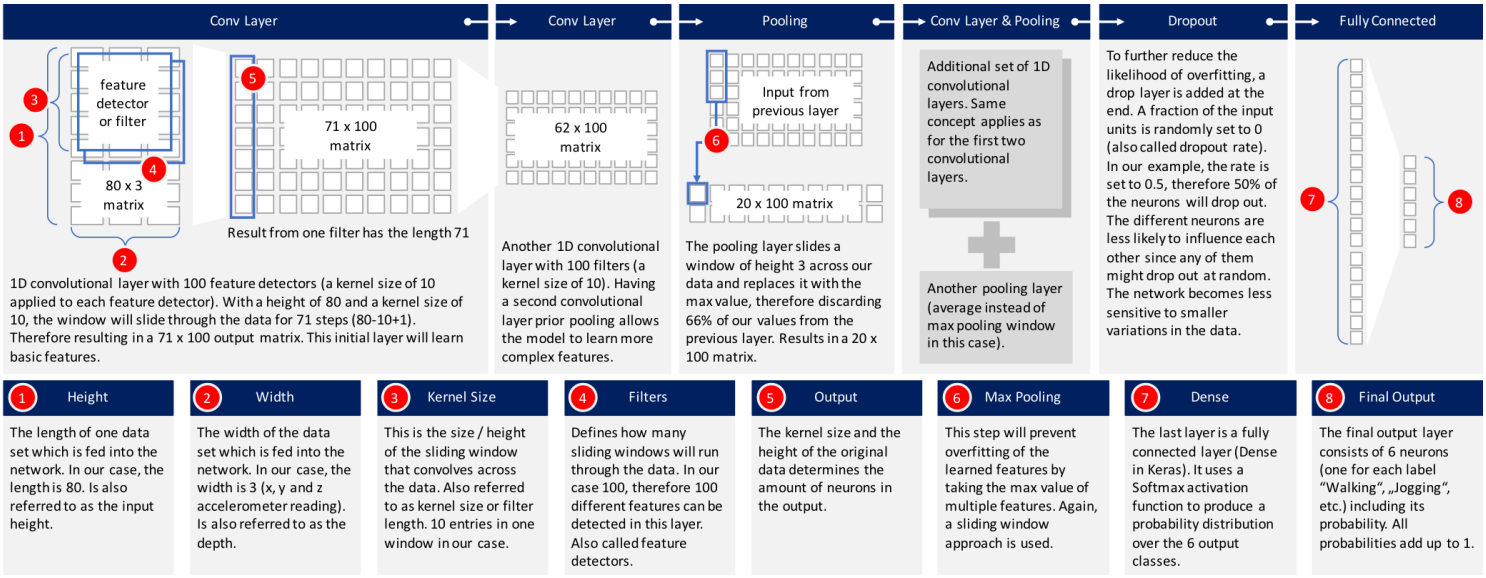
In this article we will focus on time-sliced accelerometer sensor data coming from a smartphone carried by its users on their waist. Based on the accelerometer data of the x, y and z axis, the 1D CNN should predict the type of activity a user is performing (such as “Walking”, “Jogging” or “Standing”). You can find more information in my two other articles here and here. Each time interval of the data will look similar to this for the various activities.



Example time series from the accelerometer data

How to Construct a 1D CNN in Python?

There are many standard CNN models available. I picked one of the models described on the Keras website and modified it slightly to fit the problem depicted above. The following picture provides a high level overview of the constructed model. Each layer will be explained further.



But let's first take a look at the Python code in order to construct this model:

```
model_m = Sequential()
model_m.add(Reshape((TIME_PERIODS, num_sensors), input_shape=(input_shape,)))
model_m.add(Conv1D(100, 10, activation='relu', input_shape=(TIME_PERIODS, num_sensors)))
model_m.add(Conv1D(100, 10, activation='relu'))
model_m.add(MaxPooling1D(3))
model_m.add(Conv1D(160, 10, activation='relu'))
model_m.add(Conv1D(160, 10, activation='relu'))
model_m.add(GlobalAveragePooling1D())
model_m.add(Dropout(0.5))
model_m.add(Dense(num_classes, activation='softmax'))
print(model_m.summary())
```

20180901_har_cnn_1.py hosted with ♡ by GitHub view raw

Running this code will result in the following deep neural network:

Layer (type)	Output Shape	Param #
reshape_45 (Reshape)	(None, 80, 3)	0
conv1d_145 (Conv1D)	(None, 71, 100)	3100
conv1d_146 (Conv1D)	(None, 62, 100)	100100
max_pooling1d_39 (MaxPooling	(None, 20, 100)	0

```
_________________________________________________________________
conv1d_147 (Conv1D)          (None, 11, 160)           160160
_________________________________________________________________
conv1d_148 (Conv1D)          (None, 2, 160)            256160
_________________________________________________________________
global_average_pooling1d_29  (None, 160)               0
_________________________________________________________________
dropout_29 (Dropout)         (None, 160)               0
_________________________________________________________________
dense_29 (Dense)             (None, 6)                 966
=================================================================
Total params: 520,486
Trainable params: 520,486
Non-trainable params: 0
_________________________________________________________________
None
```

Let's dive into each layer and see what is happening:

- **Input data:** The data has been preprocessed in such a way that each data record contains 80 time slices (data was recorded at 20 Hz sampling rate, therefore each time interval covers four seconds of accelerometer reading). Within each time interval, the three accelerometer values for the x axis, y axis and z axis are stored. This results in an 80 x 3 matrix. Since I typically use the neural network within iOS, the data must be passed into the neural network as a flat vector of length 240. The first layer in the network must reshape it to the original shape which was 80 x 3.

- **First 1D CNN layer:** The first layer defines a filter (or also called feature detector) of height 10 (also called kernel size). Only defining one filter would allow the neural network to learn one single feature in the first layer. This might not be sufficient, therefore we will define 100 filters. This allows us to train 100 different features on the first layer of the network. The output of the first neural network layer is a 71 x 100 neuron matrix. Each column of the output matrix holds the weights of one single filter. With the defined kernel size and considering the length of the input matrix, each filter will contain 71 weights.

- **Second 1D CNN layer:** The result from the first CNN will be fed into the second CNN layer. We will again define 100 different filters to be trained on this level. Following the same logic as the first layer, the output matrix will be of size 62 x 100.

- **Max pooling layer:** A pooling layer is often used after a CNN layer in order to reduce the complexity of the output and prevent overfitting of the data. In our example we chose a size of three. This means that the size of the output matrix of this layer is only a third of the input matrix.

- **Third and fourth 1D CNN layer:** Another sequence of 1D CNN layers follows in order to learn higher level features. The output matrix after those two layers is a 2 x 160 matrix.

- **Average pooling layer:** One more pooling layer to further avoid overfitting. This time not the maximum value is taken but instead the average value of two weights within the neural network. The output matrix has a size of 1 x 160 neurons. Per feature detector there is only one weight remaining in the neural network on this layer.

- **Dropout layer:** The dropout layer will randomly assign 0 weights to the neurons in the network. Since we chose a rate of 0.5, 50% of the neurons will receive a zero weight. With this operation, the network becomes less sensitive to react to smaller variations in the data. Therefore it should further increase our accuracy on unseen data. The output of this layer is still a 1 x 160 matrix of neurons.

- **Fully connected layer with Softmax activation:** The final layer will reduce the vector of height 160 to a vector of six since we have six classes that we want to predict (“Jogging”, “Sitting”, “Walking”, “Standing”, “Upstairs”, “Downstairs”). This reduction is done by another matrix multiplication. Softmax is used as the activation function. It forces all six outputs of the neural network to sum up to one. The output value will therefore represent the probability for each of the six classes.

Training and Testing the Neural Network

Here is the Python code to train the model with a batch size of 400 and a training and validation split of 80 to 20.

```
callbacks_list = [
    keras.callbacks.ModelCheckpoint(
        filepath='best_model.{epoch:02d}-{val_loss:.2f}.h5',
        monitor='val_loss', save_best_only=True),
    keras.callbacks.EarlyStopping(monitor='acc', patience=1)
]
```

```
]

model_m.compile(loss='categorical_crossentropy',
                optimizer='adam', metrics=['accuracy'])

BATCH_SIZE = 400
EPOCHS = 50

history = model_m.fit(x_train,
                      y_train,
                      batch_size=BATCH_SIZE,
                      epochs=EPOCHS,
                      callbacks=callbacks_list,
                      validation_split=0.2,
                      verbose=1)
```

20180901_har_cnn.py hosted with ♡ by GitHub view raw

The model reaches an accuracy of 97% for the training data.

```
...
Epoch 9/50
16694/16694 [==============================] - 16s 973us/step -
loss: 0.0975 - acc: 0.9683 - val_loss: 0.7468 - val_acc: 0.8031
Epoch 10/50
16694/16694 [==============================] - 17s 989us/step -
loss: 0.0917 - acc: 0.9715 - val_loss: 0.7215 - val_acc: 0.8064
Epoch 11/50
16694/16694 [==============================] - 17s 1ms/step - loss:
0.0877 - acc: 0.9716 - val_loss: 0.7233 - val_acc: 0.8040
Epoch 12/50
16694/16694 [==============================] - 17s 1ms/step - loss:
0.0659 - acc: 0.9802 - val_loss: 0.7064 - val_acc: 0.8347
Epoch 13/50
16694/16694 [==============================] - 17s 1ms/step - loss:
0.0626 - acc: 0.9799 - val_loss: 0.7219 - val_acc: 0.8107
```

Running it against the test data reveals an accuracy of 92%.

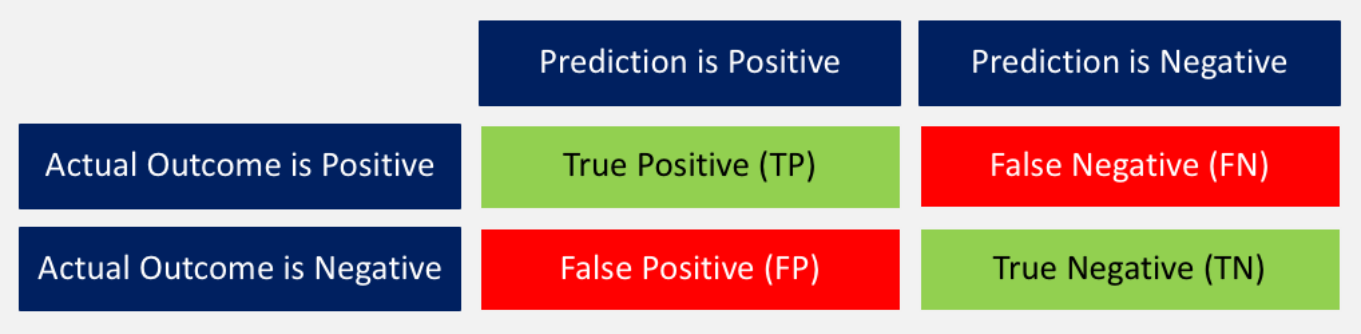
```
Accuracy on test data: 0.92

Loss on test data: 0.39
```

This is a good number considering that we used one of the standard 1D CNN models. Our model also scores well on precision, recall, and the f1-score.

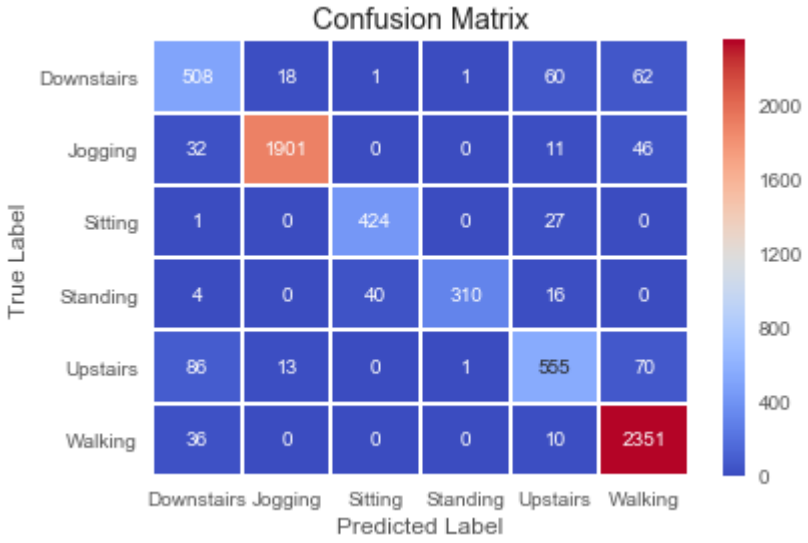
	precision	recall	f1-score	support
0	0.76	0.78	0.77	650
1	0.98	0.96	0.97	1990
2	0.91	0.94	0.92	452
3	0.99	0.84	0.91	370
4	0.82	0.77	0.79	725
5	0.93	0.98	0.95	2397
avg / total	0.92	0.92	0.92	6584

Here is a brief recap of what those scores mean:



- **Accuracy:** The ratio between correctly predicted outcomes and the sum of all predictions. ((TP + TN) / (TP + TN + FP + FN))
- **Precision:** When the model predicted positive, was it right? All true positives divided by all positive predictions. (TP / (TP + FP))
- **Recall:** How many positives did the model identify out of all possible positives? True positives divided by all actual positives. (TP / (TP + FN))
- **F1-score:** This is the weighted average of precision and recall. (2 x recall x precision / (recall + precision))

The associated confusion matrix against the test data looks as following.



Summary

In this article you have seen an example on how to use a 1D CNN to train a network for predicting the user behaviour based on a given set of accelerometer data from smartphones. The full Python code is available on github.

Links and References

- Keras documentation for 1D convolutional neural networks
- Keras examples for 1D convolutional neural networks
- A good article with an introduction to 1D CNNs for natural language processing problems

Disclaimer

The postings on this site are my own and do not necessarily represent the postings, strategies or opinions of my employer.