CNNs for time series classification

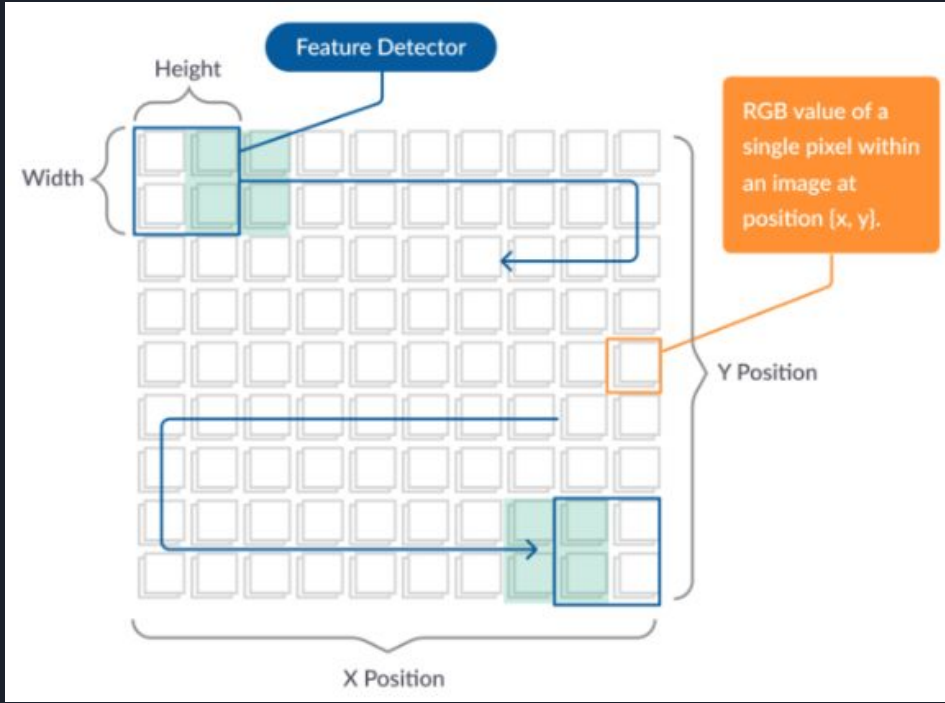
By Anthony Mendil

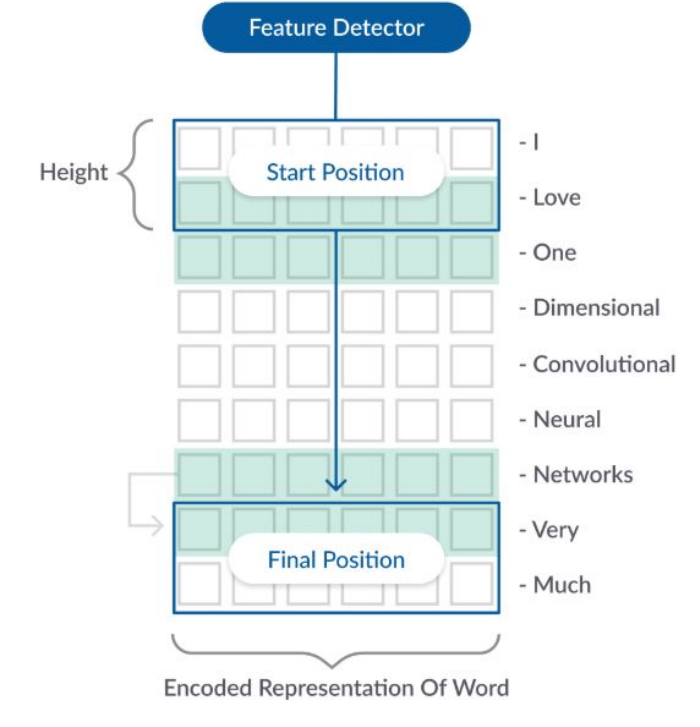
Introduction

- primarily CNNs were used for image classification (mostly 2D CNNs)
  - input is a grid
  - Dimension for example can be: 1920 (x-Pixel) * 1080 (y-Pixel) * 3 (rgb)
- but recently they were also used for time series classification if the time series has a fixed length (e.g. 20 steps) (otherwise LSTMs are more popular)
- since our task is a time series classification with fixed length the following questions are interesting
  - how to structure our data so that it can be used with a CNN
  - which type of CNN could be used

CNNs in general

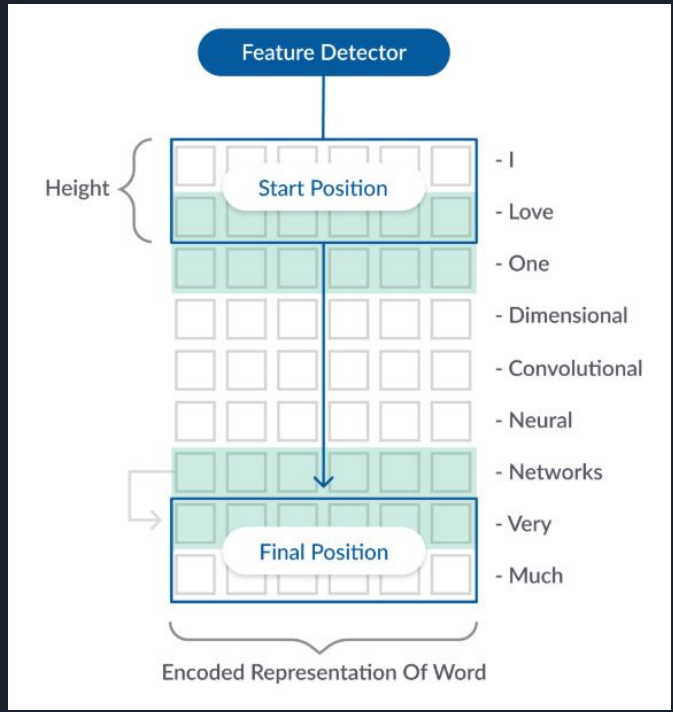
- general idea of CNNs: the complexity of the recognized patterns gets higher with each convolutional layer (lines → circles → heads → faces ...)
- there are different types of CNNs
- The classical one used for image classification: 2D CNN
- there are also 1D CNNs and 3D CNNs
- for the following discussion some information is important:
  - a difference between the types of CNNs: The dimension of the convolution (1D CNN has 1D convolution etc.)
  - the kernel size specifies how many timesteps to include in one step (size of convolution window)
  - (take a look at the following images)


Feature Detector
Height
Width
RGB value of a single pixel within an image at position (x, y).
Y Position
X Position


Feature Detector
Height
Start Position
- I
- Love
- One
- Dimensional
- Convolutional
- Neural
- Networks
- Very
Final Position
- Much
Encoded Representation Of Word

Approaches

- in the following I will present 2 approaches
- both use a 1D CNN
  - is the preferred CNN for time series classification
  - the convolution is only one dimensional
  - because we want to include all data from a timestep
- both approaches explained in the following could turn out effective so I could implement both and compare them

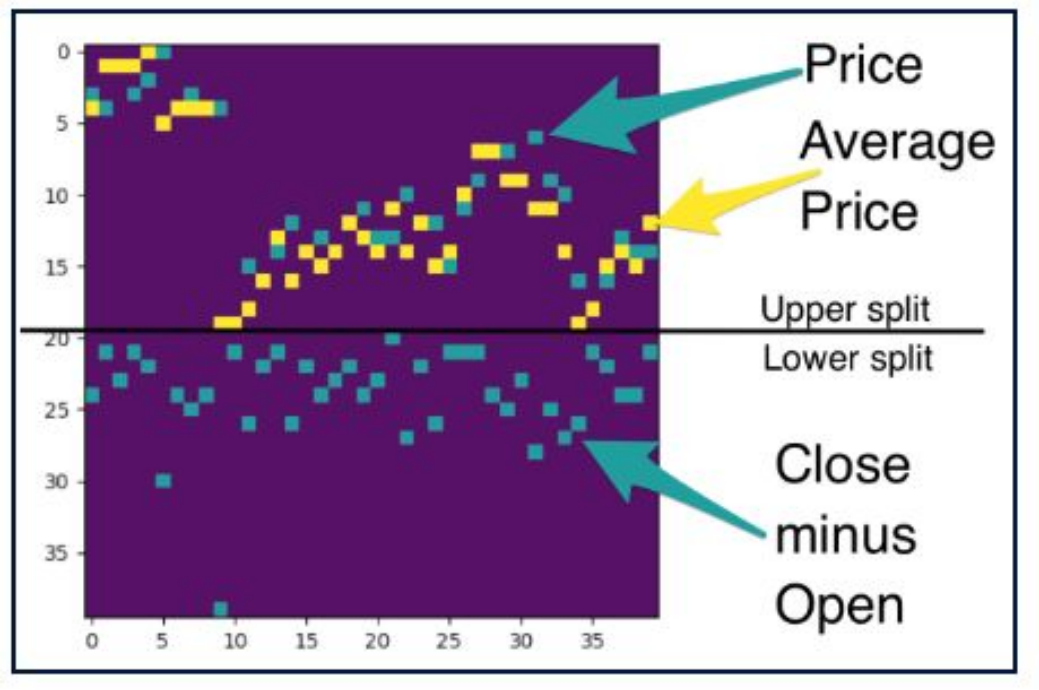


Approach 1

- more classical one
- transform the data into a 2D grid
- length is 20 (number of turns) (40 maybe if each card individually)
- width is 1 (penalties) + 1 (matching pairs) + 2 (coding for the two cards) = 4
- Result: 20*4 grid
- maybe normalize data etc.
- kernel size at least 2 → includes multiple turns in one step and therefore may find patterns of the combined turns (kernel size of 1 would just look at each turn individually)

Approach 2

- very different and interesting approach in my opinion
- creating an synthetic image out of the data
- not very common
- I found a person who used this approach for predicting the stock market
- he had graphs of the stock values etc. and created an image out of them and fed them into the CNN
- he used 2D CNN (with 2D convolution) and added a fake depth channel to the grid (don’t see the benefit yet, but could also be tested)
    - so either 1D CNN or 2D CNN
- such an image can be seen on the next slide


0
5
10
15
20
25
30
35
0
5
10
15
20
25
30
35
Price
Average
Price
Upper split
Lower split
Close
minus
Open

Approach 2

- in our case:
- we have 2 graphs
  - the penalties per round (maximum maybe 20 - haven't seen more than 15 on graphs)
  - the matching pairs per round (maximum is 14)
- instead of including the specific values we create an image of the graphs
  - dimensions for matching pairs: 20 (turns) *14 (max pairs)
  - dimensions for penalties: 20 (turns) * 20 (max penalties)
  - dimensions for the cards turned: 20 (turns) * 7 (or 14?) (different cards)
- then we stack graphs on top of eachother to create the synthetic image (like the picture in the slide before)

Conclusion

- implement both (or 3 if I also want to try 2D CNN in second approach) approaches and compare them with each other
- then in paper compare best approach with lstm
- any additional ideas or approaches?

Sources

- Approach 1

  → https://missinglink.ai/guides/keras/keras-conv1d-working-1d-convolutional-neural-networks-keras/

- Approach 2:

  → http://amunategui.github.io/unconventional-convolutional-networks/