Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 7: Convolutions beyond images

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

About this lecture

Although we focused on images in the previous lecture, many other types of data have a sequential / grid-like structure, albeit with different dimensionality: time-series, audio, videos, text, ...

In this lecture we will see how to extend CNNs to these other cases, and also some critical limitations, including their unsuitability in handling long-term or non-regular dependencies. This will provide motivation for the next class of models we will study, **transformers**.

Convolutions beyond images

1D and 3D convolutional layers

Temporal sequences

CNNs can be extended easily to other domains having grid-like structure of various dimensions.

For example, consider n steps of a time-series $\mathbf{x}_0, \dots, \mathbf{x}_{n-1} \in \mathbb{R}^c$, each step having c features (e.g., c different readings from different sensors).

We represent it using a matrix $\underset{(n,c)}{\mathbf{X}}$:

$$\mathbf{X} \sim (\overset{\text{Length of the sequence}}{t}, \overset{\text{Features}}{c})$$

We have a similar format for, e.g., text sequences, audio, DNA sequences...

1D convolutional layers

Given a patch size $s = 2k + 1$, define $\underset{(s,c)}{P_k(i)}$ as the rows in $\mathbf{X}$ at distance at most k from i.

A 1D convolutional layer $\underset{(s,c')}{\mathbf{H}} = \mathrm{Conv1D}(\mathbf{X})$, with c' an hyper-parameter that defines the output dimensionality, is defined row-wise as:

$$[\mathrm{Conv1D}(X)]_i = \phi(\mathbf{W} \cdot \mathrm{vect}(P_k(i)) + \mathbf{b}) \tag{1}$$

with trainable parameters $\underset{(c',sc)}{\mathbf{W}}$ and $\underset{(c')}{\mathbf{b}}$. Like in the 2D case, this layer is local (for a properly modified definition of locality) and equivariant to translations of the sequence.

Masked convolutions

For temporal domains, it is useful to define *causal* (masked) versions of the convolution operation.

For a temporal sequence $\underset{(n,c)}{\mathbf{X}}$, a **causal model** $\underset{(n,c')}{\mathbf{H}} = f(\mathbf{X})$ is such that $[\mathbf{H}]_i$ depends only on $[\mathbf{X}]_j$ for $j \leq i$.

$$\mathbf{h}_i = \phi\left(\left[\underbrace{\mathbf{W} \odot \mathbf{M}}_{\text{Masked weight matrix}}\right] \text{vect}(P_k(i)) + \mathbf{b}\right)$$

where $M_{ij} = 0$ if the weight corresponds to an element in the input such that $j > i$, 1 otherwise.

Causal convolutions

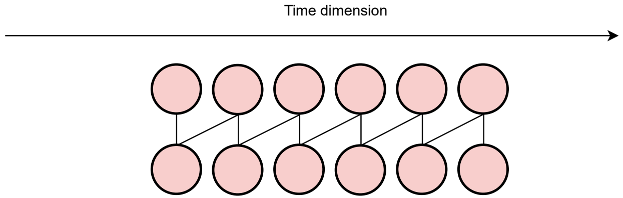


Figure 1: A 1D convolutional layer with causal convolutions. All links going right-to-left are removed. As a result, for each unit the output only depends on the previous time-instants.

Forecasting

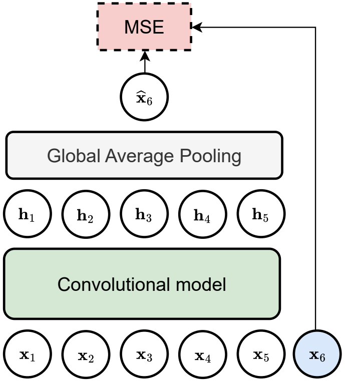
For time-series, a common task is **forecasting**, i.e., predicting the next step in the time-series. With a causal model, we have two options:

- Pool the output representation $\mathbf{H}$ over n, and apply a regressor head to predict $\mathbf{x}_n$ (also valid for non-causal models).
- Define a *shifted* target $\mathbf{Y} = [\mathbf{x}_1, \dots, \mathbf{x}_n]$, and train the model such that $\mathbf{H} \approx \mathbf{Y}$, i.e., at each time step the network predicts the next one:

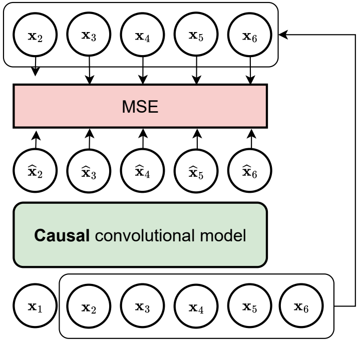
$$l(\widehat{\mathbf{Y}}, \mathbf{Y}) = \|\widehat{\mathbf{Y}} - \mathbf{Y}\|^2 = \sum_{i=1}^{n} \underbrace{\|\widehat{\mathbf{Y}}_i - \mathbf{Y}_i\|^2}_{\text{Loss when predicting } \mathbf{X}_{i+1}} \tag{2}$$

This is only possible with a causal model, otherwise information would ‘leak’ from the input.

Forecasting with non-causal vs. causal models


MSE
$\hat{\mathbf{x}}_6$
Global Average Pooling
$\mathbf{h}_1$ $\mathbf{h}_2$ $\mathbf{h}_3$ $\mathbf{h}_4$ $\mathbf{h}_5$
Convolutional model
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$ $\mathbf{x}_6$

(a) Single-step forecasting


$\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$ $\mathbf{x}_6$
MSE
$\hat{\mathbf{x}}_2$ $\hat{\mathbf{x}}_3$ $\hat{\mathbf{x}}_4$ $\hat{\mathbf{x}}_5$ $\hat{\mathbf{x}}_6$
Causal convolutional model
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$ $\mathbf{x}_6$

(b) Multi-step forecasting

Autoregressive models

Models of the second form are useful because they can be used to generate data in an **autoregressive** way. We prompt the model with the beginning of the sequence, and let it forecast the next step. Then, we concatenate it to the input and prompt again the model to generate another step, and so on.

During training, we can feed the network only with true samples (**teacher forcing**), or mix some of the network's own predictions. Extrapolating outside the training data windows may still be difficult.

An example

Suppose we have $n = 4$, and we have observed two values $\mathbf{x}_1$ and $\mathbf{x}_2$. We call the model a first time:

$$\begin{bmatrix} - \\ \widehat{\mathbf{x}}_3 \\ - \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right)$$

We add $\widehat{\mathbf{x}}_3$ to the sequence and continue calling the model autoregressively (we show in color the predicted values):

$$\begin{bmatrix} - \\ - \\ \widehat{\mathbf{x}}_4 \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \mathbf{0} \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_5 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_6 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \\ \widehat{\mathbf{x}}_5 \end{bmatrix}\right) \ldots$$

Autoregressive generation

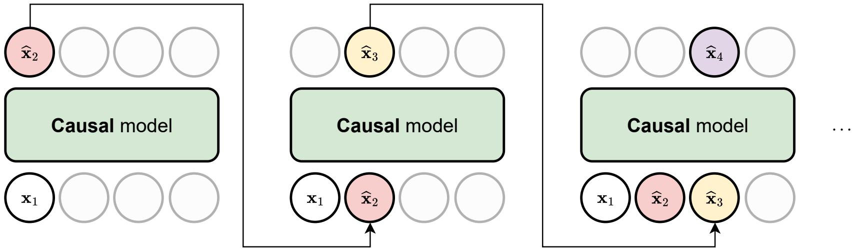


Figure 2: Example of autoregressive generation with a single step in input, and the model being called recursively multiple times. We show in gray padded values.

Convolutions beyond images

Applying CNNs to audio

Examples of audio classification

Many real-world problems require the classification of **audio samples**, e.g.:

1. Speech / non-speech identification (*is he/she speaking now?*);
2. Language identification (*is it Italian?*);
3. Genre / mood classification (*is it rock?*);
4. Determining the leading instrument;
5. Event recognition (*is someone shooting?*);
6. Scene recognition (*are they in a bus? at a restaurant?*).

Channels and sampling rate

An audio is a 1D sequence of **samples**, obtained with a certain **sampling rate**, typically in one or two **channels**.

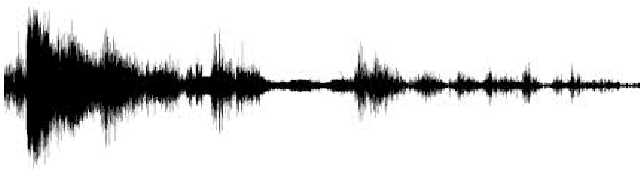


Figure 3: Simple example of an audio **waveform** (image source).

For a given audio, the number of samples can be very high: at a **resolution** of 44kHz, we have almost half a million samples for each 10 seconds.

Spectrogram classification

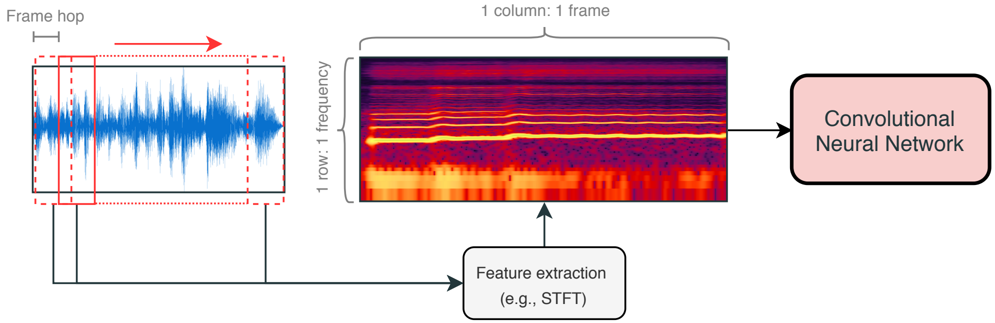


Figure 4: To simplify the problem we can sweep a window of fixed size (e.g., 40ms) over the signal, compute a frequency representation (e.g., with a short-term Fourier transform), and merge everything into a single **spectrogram**.

Sequence padding

Consider two audio files (or two time series, or two texts), described by their corresponding input matrices $\underset{(t_1,c)}{X_1}$ and $\underset{(t_2,c)}{X_2}$. The two inputs share the same number of channels c (e.g., the number of sensors), but they have different lengths, t_1 and t_2.

Convolutional models can handle variable-lenght inputs (**why?**), but in practice mini-batches cannot be built from matrices of different dimensions. This is handled by zero-padding the resulting mini-batch to the maximum dimension across the sequence length.

An example

Assuming for example, without lack of generality, $t_1 > t_2$, we can build a "padded" mini-batch as:

$$X = \text{stack}\left(\mathrm{X}_1, \begin{bmatrix} \mathrm{X}_2 \\ \mathbf{0} \end{bmatrix}\right)$$

where stack operates on a new leading dimension, and the resulting tensor X has shape $(2, t_1, c)$. We can generalize this to any mini-batch by considering the largest length with respect to all elements of the mini-batch.

Can we handle the full waveform?

By having sufficient computational power, one can also work on the raw audio waveform.

When using convolutions, a key idea is the use of **dilated convolutions** (a.k.a. **atrous** convolutions, from French *à trous*), where neighbours are selected with exponentially increasing steps.

In this way, the receptive field of an item increases *exponentially* with the number of layers.

Visualization of a dilated 1D convolution

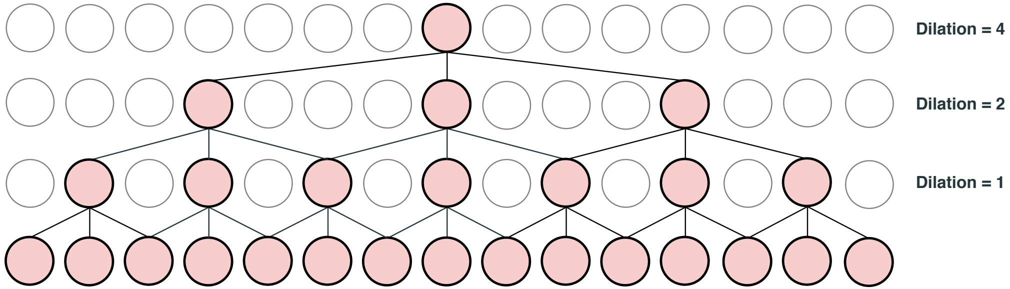


Figure 5: Example of dilated (atrous) convolution, with increasing dilation for each layer.

WaveNet

WaveNet (2016) was one of the first models to propose a 1D convolutional network dilated and causal convolutions for generating speech. In their design, dilation factors increased from 1 to an upper bound (e.g., 512), before restarting from 1.

Note that autoregressive generation of speech can be very slow, because the model must be called hundreds of thousands of times: although convolutive models have given way to transformers today, this problem is still true (e.g., the recent research on **speculative decoding** in LLMs).

Oord, A.V.D., et al., 2016. **WaveNet: A generative model for raw audio**. *arXiv preprint arXiv:1609.03499*.

Convolutions beyond images

Applying CNNs to text

Applying convolutive networks to text processing

Text data is another field with a vast range of possible applications, e.g.:

1. Recognition of a topic (*is it talking about soccer?*);
2. Hate speech recognition (*is it respecting our code of conduit?*);
3. Sentiment analysis (*is it a positive review?*);
4. Web page classification (*is it an e-commerce website?*).

Causal models applied to text are the backbone of LLM models such as ChatGPT.

Handling text data

Text must be preprocessed properly to be handled by a neural network. At the very least we need to perform two basic operations:

1. **Tokenize** the text to split it in into basic units (**tokens**) that will form a sequence.
2. **Embed** each token (one or more characters), i.e., convert it into numerical features usable by a neural network.

By the end of this procedure we should have a matrix $\underset{(n,d)}{\mathbf{X}}$ of n tokens, each of size d.

In our discussion, we are ignoring a large number of possible preprocessing operations, e.g., stemming, stop word removal, etc.

Convolutions beyond images

Text tokenization

Text tokenization

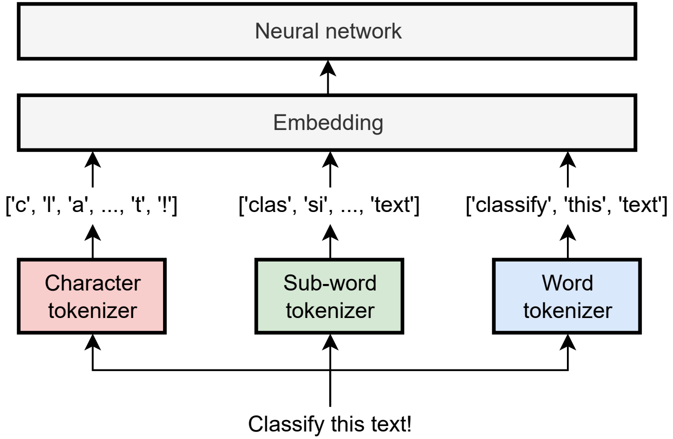


Figure 6: Tokenization can be performed at various levels in the sentence, and it is generally handled by an external library (e.g., Spacy).

The importance of tokenization

Tokenization is crucial because it determines what the network will 'see'. Consider that v15 of Unicode has almost 150k possible symbols, which can be combined in many forms (numbers, dates, function names, ...).

Subword tokenization is very common, especially when learning the subwords from large corpora of texts (e.g., **byte pair encoding**, BPE). See `https://huggingface.co/docs/transformers/tokenizer_summary` for a summary.

Also see `https://platform.openai.com/tokenizer` to visualize a tokenization in practice.

Convolutions beyond images

Token embeddings

One-hot encoding

The simplest vectorial embedding for text is a **one-hot encoding** according to a predefined **dictionary**:

- **Character-level**: each character is represented by a 1-of-C binary vector, where C is the number of allowable characters.
- **Sub-word/word-level**: similar, but each word/sub-word is represented with respect to a fixed vocabulary of sub-words / words.
- **Sentence-level**: each sentence can be represented by summing the one-hot encodings for the single tokens (**bag-of-words**).

Dense embeddings

One-hot vectors are very simplistic representations of the information contained in text. A more general solution is to *learn* a set of embeddings:

1. For every possible token c, initialize randomly a fixed-size vector $\mathbf{v}_c$.
2. During training, substitute each token in the sequence with the corresponding vector (look-up).
3. The set of vectors $\mathbf{v}_c$ can be optimized together with the parameters of the neural network by doing gradient descent.

Visualizing the embedding procedure

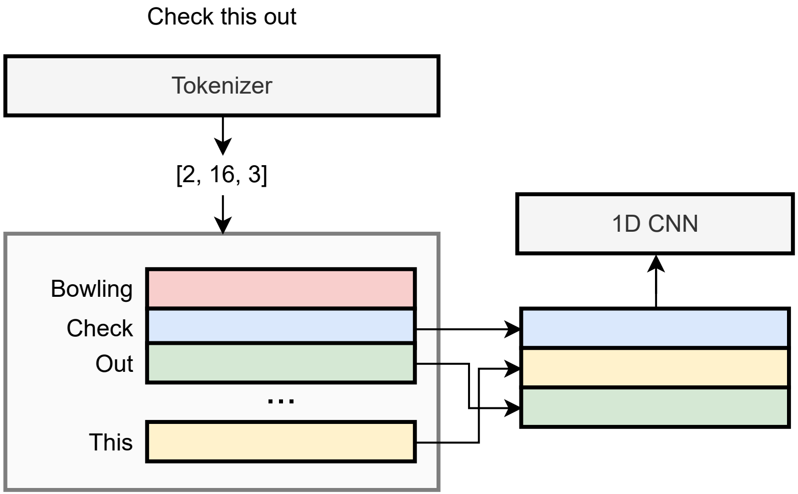


Figure 7: Using custom embeddings for text classification.

Custom embeddings in Keras

Custom embeddings are extremely simple to train within a TensorFlow model:

```
model = Sequential()

# Get embeddings for each token
# Input must be (batch_size, max_sentence_length)
# Every element of the input is an index [0, ..., dictionary_size-1]
model.add(Embedding(dictionary_size, B, input_length=max_sentence_lengt

# Optional: Get average embedding for the sentence
model.add(GlobalAveragePooling1D())

# ...
```

Dense vs. sparse embeddings

One-hot encodings are *sparse*, in the sense that most values are 0. In addition, they are not very informative: for example, the distance between any two encodings $\mathbf{e}_0$ and $\mathbf{e}_1$ is either 0 or $\sqrt{2}$.

Instead, trained embeddings can capture a rich semantic underlying the data, which is reflected in the possibility of doing algebraic manipulations on the embeddings themselves, e.g., see `https://projector.tensorflow.org/`. However, they can also capture biases of the data.[1]

[1] Bolukbasi, T. et al., 2016. **Man is to computer programmer as woman is to homemaker? Debiasing word embeddings**. *NeurIPS*.

Pre-trained text embeddings

Text embeddings can also be **pre-trained** using a variety of algorithms:

1. **Word2Vec** (Mikolov et al., 2013);
2. **Global Vectors for Word Representation** (GloVe) (Pennington et al., 2014);
3. **Embeddings from Language Models** (ELMo) (Peters et al., 2018);
4. **Generative Pre-Training** (GPT) (Radford et al., 2018);
5. **Bidirectional Encoder Representations from Transformers** (BERT) (Devlin et al., 2018).