Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 7: Attention-based models

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Introduction

Applying CNNs to audio

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?*).

Just like an image is a 2D sequence of **pixels**, an audio is a 1D sequence of **samples**, obtained with a certain **sampling rate**, typically in one or two **channels**.

Figure 1: Simple example of an audio **waveform** (image source).

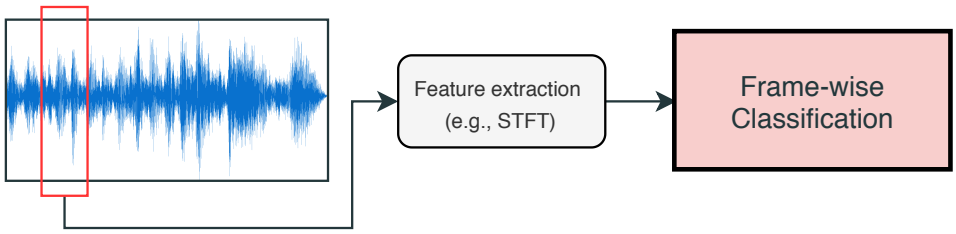

Figure 2: A standard workflow for audio classification: (i) extract a fixed-dimensional **frame** from the audio (e.g., 400 ms); (ii) extract a vector of features through frequency analysis; (iii) classify the frame. Frame-wise predictions can then be aggregated.

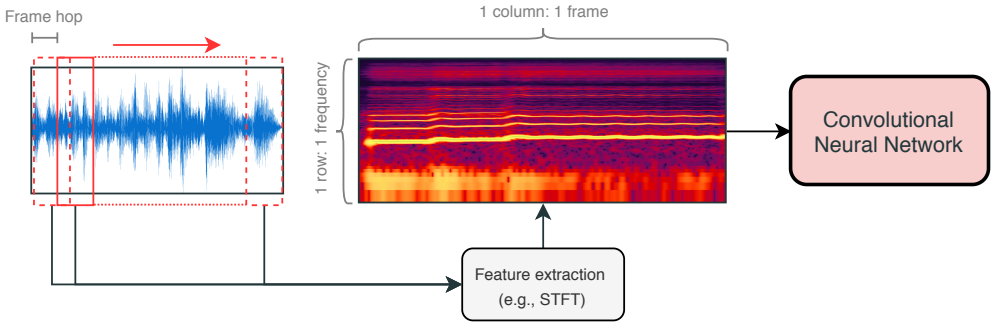

Figure 3: By sweeping the feature extractor for all frames, we obtain a time-frequency representation to be used by a standard CNN architecture.

To work with mini-batching, you still need to zero-pad all dimensions to a maximum length. For example, you can use `padded_batch()` with `tf.data`:

```
im1 = tf.random.normal(shape=(4, 3)) # Image 1 has shape (4, 3)
im2 = tf.random.normal(shape=(4, 2)) # Image 2 has shape (4, 2)

# We need to use from_generator instead of from_tensor_slices
data = tf.data.Dataset.from_generator([im1, im2].__iter__,
output_types=tf.float64)

for xb in data.padded_batch(2, padded_shapes=(4, 3)):
  print(xb) # Zero-padded to (4, 3)
```

We will see other solutions in future lab sessions.

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.

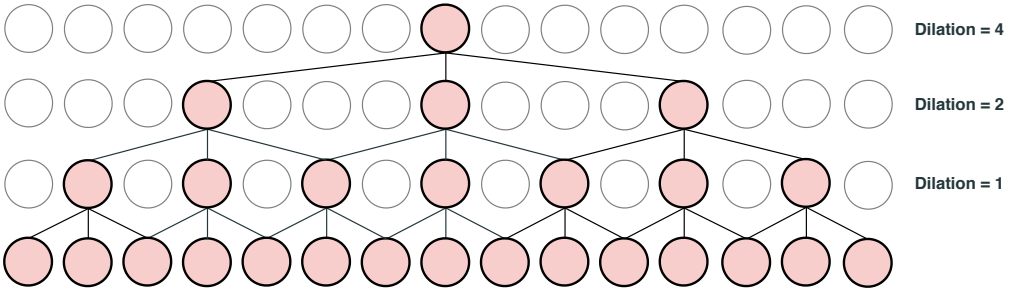

Figure 4: Example of dilated (atrous) convolution, with increasing dilation for each layer.

Introduction

Applying CNNs to text

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?*).

Generative applications are also extremely popular recently.

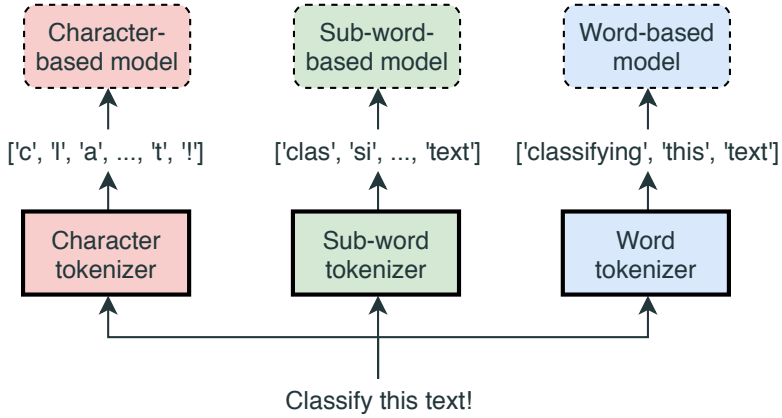

Figure 5: Tokenization can be performed at various levels in the sentence, and it is generally handled by an external library (e.g., Spacy).

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**).

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

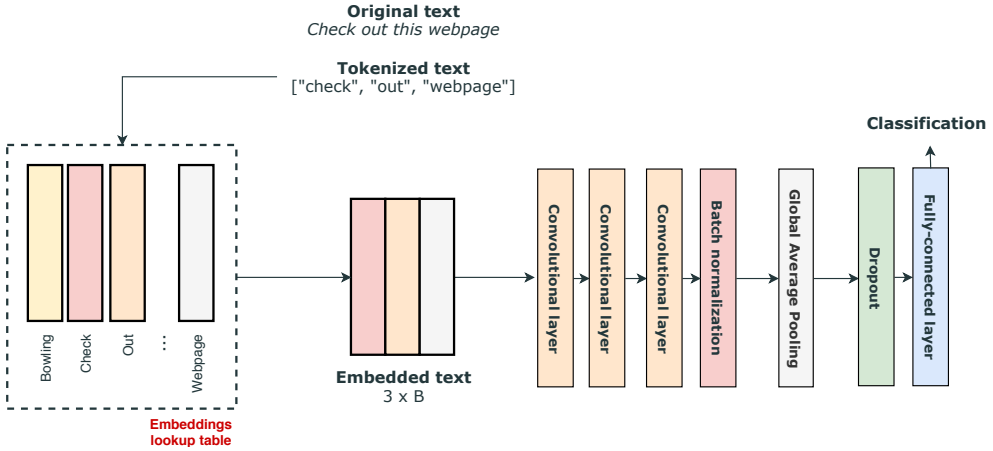

Figure 6: Using custom embeddings for text classification.

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())

# ...
```

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).

This is a very active research area, currently reaching state-of-the-art results in a variety of NLP tasks.

Multi-head attention layer

Moving beyond convolutional layers

Note how the assumption of **locality** we made for convolutional layers might not be optimal in all these cases. For text, audio, etc., long-term dependencies can be important.

Until a few years ago, **recurrent neural networks** (RNNs) were a viable alternative, but because of their structure, they are highly time-consuming to use. Recently, **transformer** models have become more common, especially when trained from huge datasets. In fact, most pre-trained word embeddings are built on this architecture.

The core of the transformer is a new layer called **multi-head attention** (MHA). It replaces the assumption of locality with a more general notion of **sparsity** of interactions.

The original Transformer (Vaswani et al., 2017), was an encoder-decoder model for NLP tasks. Today, similar models are gaining interest in audio, computer vision, biology, etc.

Multi-head attention layer

Nadaraya-Watson estimators

To understand the MHA layer and the idea of sparse, consider the famous k-NN algorithm. Given a dataset $(\mathbf{x}_i, y_i)$, the output of the k-NN (assuming a regression task) is computed as:

$$f(\mathbf{x}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x})} y_i \,,$$

where $\mathcal{N}_k(\mathbf{x})$ returns the k points in the dataset closest to $\mathbf{x}$ according to some distance function $d(\mathbf{x}, \mathbf{x}_i)$. The output of the k-NN is *sparse* because it only depends on k points out of the full dataset.

The operation of selecting the closest k points is not differentiable, so we cannot use it in a neural architecture. We consider a simple generalization over *all* points:

$$f(\mathbf{x}) = \sum_i d(\mathbf{x}, \mathbf{x}_i) y_i \,.$$

For a distance function that decreases fast, this will approximate the k-NN since most coefficients will be ≈ 0. For example, we can use the Gaussian distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \exp(-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2) \,.$$

We can constrain the weights to sum to 1 by normalizing them:

$$f(\mathbf{x}) = \sum_i \frac{d(\mathbf{x}, \mathbf{x}_i)}{\sum_j d(\mathbf{x}, \mathbf{x}_j)} y_i \,.$$

This is an example of a **Nadaraya-Watson kernel estimator**. Note that if we substitute the Gaussian distance function we obtain:

$$f(\mathbf{x}) = \sum_i \text{softmax}(-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2) y_i \,.$$

Using the softmax, we can rewrite the previous layer as:

$$f(\mathbf{x}) = \sum_i \text{softmax}\left(a(\mathbf{x}, \mathbf{x}_i)\right) y_i \,.$$

We will call $a(\cdot, \cdot)$ the **attention scoring function**, its outputs the **unnormalized attention scores**, and the output of the softmax the **attention scores**.

We still need to solve a few issues to have a proper neural layer: the layer must be *composable* and have some *trainable parameters*.

Multi-head attention layer

Self-attention layer

The previous model cannot be layered, but we can obtain a composable variant by using a weighted average of *inputs* instead of the *outputs*:

$$h(\mathbf{x}) = \sum_i \text{softmax}(a(\mathbf{x}, \mathbf{x}_i))\mathbf{x}_i \,.$$

We also need to add some trainable parameters to learn via gradient descent:

$$\underbrace{\mathbf{q} = \mathbf{W}_q^\top \mathbf{x}}_{\text{query}}, \quad \underbrace{\mathbf{k}_i = \mathbf{W}_k^\top \mathbf{x}_i}_{\text{key}}, \quad \underbrace{\mathbf{v}_i = \mathbf{W}_v^\top \mathbf{x}_i}_{\text{value}}$$

$$h(\mathbf{x}) = \sum_i \text{softmax}(a(\mathbf{q}, \mathbf{k}_i))\mathbf{v}_i \,.$$

To summarize, the input of our layer is a *set* of n vectors $\{\mathbf{x}_i\}$, which we can stack into a matrix $\underset{(n,d)}{\mathbf{X}}$.

If we apply the layer in turn to each element i in the input, we obtain an operation called **self-attention**, that provides a new vectorial representation of each input:

$$\underset{(n,q)}{\mathbf{Q}} = \mathbf{X}\mathbf{W}_q, \quad \underset{(n,q)}{\mathbf{K}} = \mathbf{X}\mathbf{W}_k, \quad \underset{(n,v)}{\mathbf{V}} = \mathbf{X}\mathbf{W}_v$$

$$\mathbf{h}_i = \sum_j \text{softmax}(a(\mathbf{Q}_i, \mathbf{K}_j))\mathbf{V}_j \,.$$

We started our explanation by considering the k-NN, where each element of the set is an element of the dataset.

However, the layer is generic for any set of vectors. Below we consider the application where the input is a set of words (the words composing a sentence), each represented through a vectorial embedding.

When the layer is applied to a batch of elements, it computes the attention function independently for every element of the batch (i.e., each sentence can *attend* only to words in the same sentence).

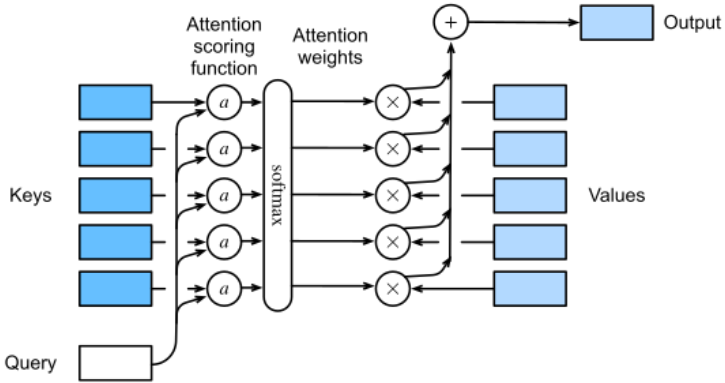

Figure 7: Visualization of the attention operation (book, Chapter 10.3).

It is common to compute the attention score using a dot-product formulation (because it is extremely efficient):

$$a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{q}}.$$

In this case, the entire self-attention layer has a particularly easy formulation:

$$\underset{(n,v)}{\mathbf{H}} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{q}}\right)\mathbf{V}.$$

A common generalization of self-attention is called **multi-head attention**. It works by computing $i = 1, \ldots, k$ separate sets of keys, querys, and values:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_{q,i}, \quad \mathbf{K}_i = \mathbf{X}\mathbf{W}_{k,i}, \quad \mathbf{V}_i = \mathbf{X}\mathbf{W}_{v,i}$$

$$\mathbf{H}_i = \text{softmax}\left(\frac{\mathbf{Q}_i\mathbf{K}_i^\top}{\sqrt{q}}\right)\mathbf{V}_i\,.$$

Then, we concatenate everything and perform a final linear projection:

$$\mathbf{H} = \begin{bmatrix}\mathbf{H}_1 & \cdots & \mathbf{H}_k\end{bmatrix}\mathbf{W}_o\,.$$

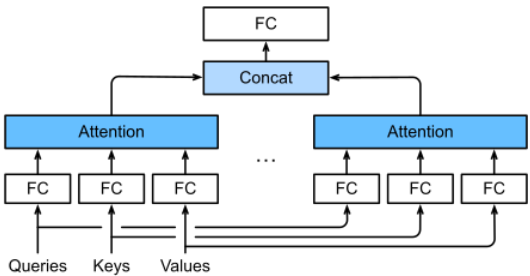

Figure 8: Visualization of the multi-head attention operation (book, Chapter 10.5).

Multi-head attention layer

The Transformer model

In audio and text, the ith row of $\mathbf{X}$ represents a single time-step or a single text token (e.g., a word). In a MHA layer, their ordering is lost, because the layer is *equivariant* to the ordering (similar to the GAT layer for graphs).

If we multiply $\mathbf{X}$ by a permutation matrix $\mathbf{P}$ (see the lecture on GNNs), then:

$$\mathrm{MHA}(\mathbf{PX}) = \mathbf{P} \cdot \mathrm{MHA}(\mathbf{X}) \,.$$

This is not a good property to have for sequences.

Before the first MHA layer, we concatenate to the input X a matrix of **positional embeddings** $\underset{(n,e)}{\mathsf{E}}$:

$$\mathsf{X}' = [\mathsf{X} \,\|\, \mathsf{E}] \,,$$

where each row $[\mathsf{E}]_i$ should *uniquely* encode the position of every element of the sequence.

Using this strategy, we 'break' the equivariance:

$$\mathrm{MHA}(\mathsf{PX} \,\|\, \mathsf{E}) \neq \mathsf{P} \cdot \mathrm{MHA}(\mathsf{X} \,\|\, \mathsf{E}) \,.$$

We can encode the position for a sequence of maximum length p with a one-hot vector of dimension p, e.g.:

$$\mathbf{E}_0 = [1, 0, 0, \ldots] \ , \quad \mathbf{E}_1 = [0, 1, 0, \ldots] \ , \quad \mathbf{E}_2 = [0, 0, 1, \ldots] \ , \quad \cdots \ .$$

Or with a single increasing scalar:

$$\mathbf{E}_0 = [0/p] \ , \quad \mathbf{E}_1 = [1/p] \ , \quad \mathbf{E}_2 = [2/p] \ , \quad \cdots \ .$$

Both strategies are not particularly good empirically.

We can *learn* the positional embeddings using the `tf.keras.layers.Embeddi` layer:

- To each position i we associate an embedding vector of fixed dimension.
- The embeddings are trained with the rest of the network.

Note that we need to fix the maximum length of the sentence. For longer sentences, we need to linearly interpolate the set of vectors up to a larger dimension (this is the strategy used in BERT and the Vision Transformer described below).

Consider a single sinusoidal function of frequency ω:

$$\mathbf{E}_i = [\sin(i\omega)] \ .$$

We can interpret this as a clock with frequency ω: for two points inside a single rotation, it will give us their relative distance. For other points, the distance will be precise modulo the frequency.

To uniquely identify any possible position, we can consider multiple sinusoids, each with a frequency $\omega_j, j = 1, \ldots, e$:

$$\mathbf{E}_i = \left[\sin(i\omega_0), \sin(i\omega_1), \ldots, \sin(i\omega_e)\right].$$

You can think of this as a clock with e different hands, each rotating at its own frequency. This is a nice representation because it can possibly generalize to any length, without the need to impose a maximum length *a priori*.

An empirically good choice for the frequencies (popularized by (Vaswani et al., 2017)) is:

$$\omega_j = \frac{1}{10000^{j/e}}\,.$$

For $j = 0$, this has frequency 2π. For $j = e$, this has frequency $10000 \cdot 2\pi$. In the middle, the frequency are increasing at a geometric progression.

To reduce the number of parameters, it is also common to *sum* the positional encodings instead of concatenating (in which case the dimension e is equal to d):

$$\mathsf{X}' = \mathsf{X} + \mathsf{E}\,.$$

A popular extension is to alternate sines and cosines of the same frequency:

$$[\mathbf{E}]_{i,2j} = \sin\left(\frac{i}{10000^{2j/e}}\right), \tag{1}$$

$$[\mathbf{E}]_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/e}}\right). \tag{2}$$

One important property of this encoding is that it is possible to translate an encoding via matrix multiplication:

$$[\mathbf{E}]_{i+p} = [\mathbf{E}]_i \mathbf{T}(p) \quad \text{for some } \mathbf{T}(p).$$

See https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ and references therein.

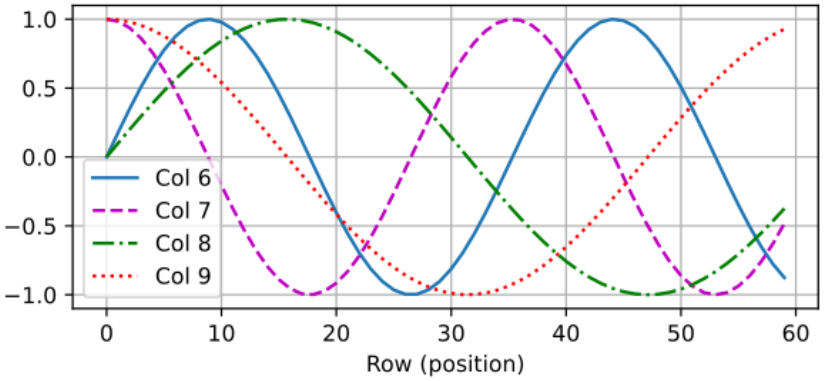

Figure 9: Visualization of the sinusoidal positional encodings (book, Chapter 10.6).

It is common to use the MHA layer inside a more complex block, called the **transformer** block, composed of a MHA layer, two layer normalization operations, two residual connections, and a so-called **position-wise** network:

1. Start with a MHA layer: $\mathbf{H} = \text{MHA}(\mathbf{X})$.
2. Add a residual connection and a layer normalization operation: $\mathbf{H} = \text{LayerNorm}(\mathbf{H} + \mathbf{X})$.
3. Apply a fully-connected model $g(\cdot)$ on each row: $\mathbf{F} = g(\mathbf{H})$.
4. Do again step 2: $\mathbf{H} = \text{LayerNorm}(\mathbf{F} + \mathbf{H})$.

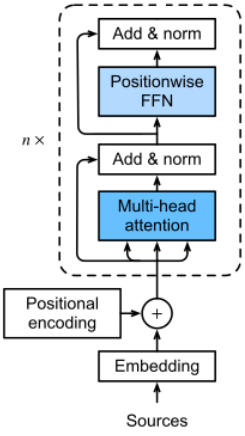

Figure 10: The final model is built with positional encodings and a stack of n transformer blocks (adapted from Chapter 10.7 of the book).

To perform classification or regression, we can apply a final global pooling on the n tokens and one or more fully-connected layers.

An alternative that is empirically found to work well is the **class token**, which is an additional *trainable* token $\mathbf{c}$ added to the input matrix:

$$\underset{(n+1,d)}{\mathbf{X}'} = \begin{bmatrix} \mathbf{X} \\ \mathbf{c}^\top \end{bmatrix}.$$

The transformer model is applied to the matrix $\mathbf{X}'$ as input ($\mathbf{H} = \text{Transformer}(\mathbf{X}')$), and classification is performed on its last row:

$$\mathbf{y} = \text{softmax}(\mathbf{W}^\top [\mathbf{H}]_{n+1}).$$

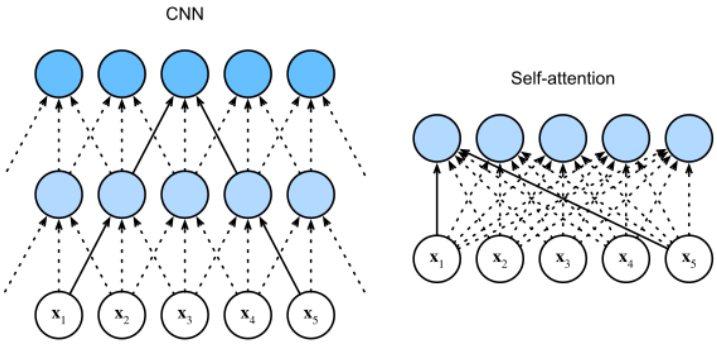

Figure 11: Adapted from Chapter 10.6 of the book.

Consider a 1D convolutional operation $\underset{(n,d)}{\mathbf{H}} = \text{Conv1D}(\underset{(n,d)}{\mathbf{X}})$ with a filter size of k. Computing the output requires $\mathcal{O}(nkd^2)$ operations.

Self-attention (with one head) requires $\mathcal{O}(nd^2)$ for computing keys, queries, and values, and $\mathcal{O}(n^2d)$ for computing the output. The n^2 term limits the applicability to long sequences, unless more advanced models are used (e.g., $n < 512$ in many text models).

However, a single layer of MHA has a receptive field of n, while the convolutive layer has a receptive field of k.

Practical transformer models

Text Transformers

The majority of pre-trained word embedding models we discussed above are, in fact, standard transformer models trained on the sequence of text tokens.

- **BERT-like** models are pre-trained by masking one word in a sentence, and reconstructing the full sentence in output.
- **GPT-like** models are (causal) variants pre-trained to generate the sequence auto-regressively.

These models are called **contextual** embeddings because the same word in different sentences can be encoded to different vectorial representations.

Qiu, X., et al., 2020. **Pre-trained models for natural language processing: A survey**. *Science China Technological Sciences*, pp. 1-26.

Because these models are trained from the raw text alone (no specific targets) they are called **self-supervised** models (we will cover this more in-depth later).

Their strengths is that scaling laws for transformers are empirically better than for other models (i.e., they benefit more from increasing the dataset by order of magnitude).

In natural language processing, this is also shown by the emergence of paradigms like **text-prompting** and **zero-shot learning**.

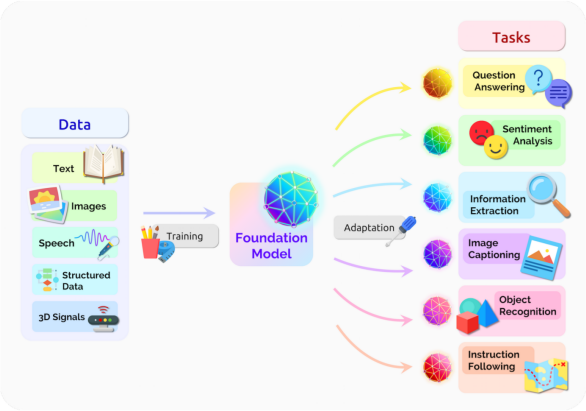

Fig. 2. A foundation model can centralize the information from all the data from various modalities. This one model can then be adapted to a wide range of downstream tasks.

Figure 12: An emerging name for these huge, pre-trained models is **foundation models**.

The original transformer model was a more general model defined for **sequence to sequence** (seq2seq) tasks, such as machine translation.

The model we discussed performs an **encoding** of the input sequence, which is then **decoded** by a second, masked transformer to generate the output sequence.

Vaswani, A., et al., 2017. **Attention is all you need**. In *Advances in neural information processing systems* (pp. 5998-6008).

Practical transformer models

Vision & Audio Transformers

One important realization of the last two years is that transformers can also benefit computer vision, especially when trained on huge datasets (e.g., ImageNet21k).

However, this requires to convert the original image (a 2D grid) into a 1D sequence (actually, a set together with the positional embeddings). Because this would scale quadratically in the number of pixels, a common solution is to work on **patches** of the original image.

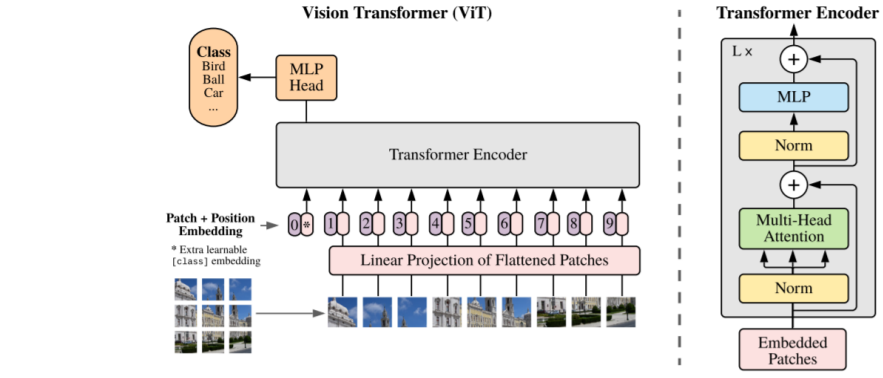

Figure 13: The Vision Transformer (ViT) is a standard transformer applied on top of image patches.

Dosovitskiy, A., et al., 2020. **An image is worth 16x16 words: Transformers for image recognition at scale**. *arXiv preprint arXiv:2010.11929*.

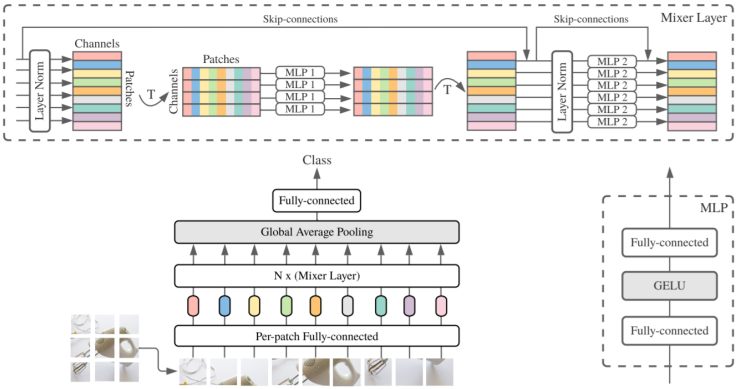

Figure 14: Mixer models are variants of the ViT, where the MHA is replaced by fully-connected layers.

Tolstikhin, I., et al., 2021. **MLP-Mixer: An all-MLP architecture for vision**. *arXiv preprint arXiv:2105.01601*.

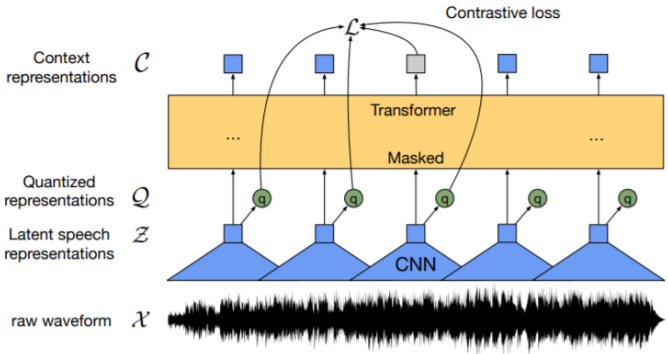

Figure 15: Architectures like **Wav2Vec 2.0** are pre-trained audio models exploiting transformers However, this is harder because of the nature of the audio signal.

Baevski, A., Zhou, H., Mohamed, A. and Auli, M., 2020. **wav2vec 2.0: A framework for self-supervised learning of speech representations**. *arXiv preprint arXiv:2006.11477*.

- (**Mandatory**) Chapter 10 of the book.
- (**Optional**) Chapter 14 for more details on pre-training transformer-based word embeddings.
- https://jalammar.github.io/illustrated-transformer/.