Neural Networks for Data Science

Master's Degree in Data Science

Lecture 5: Convolutional models

Lecturer: S. Scardapane



SAPIENZA
UNIVERSITÀ DI ROMA

Convolutional neural networks

Images and MLPs

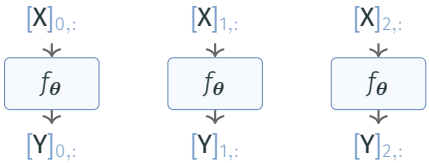
An **image** is a tensor $X \sim (H, W, C)$, where:

- H is the **height** of the image (e.g., 512 pixels);
- W is the **width** of the image (e.g., 1024 pixels);
- C is the number of **channels** (e.g., 3 channels for an RGB image, 1 channel for a greyscale image).

The first two axes have a precise *grid* ordering, while the channels do not have a precise ordering (i.e., we can switch RGB to GBR or BRG with no information loss).

We can share an MLP across the positions of a sequence $\mathbf{X} \sim (T, D)$ or an image $X \sim (H, W, C)$, applying it to each position independently:

$$[\mathbf{Y}]_{t,:} = f_{\boldsymbol{\theta}}([\mathbf{X}]_{t,:}),$$
$$[Y]_{i,j,:} = f_{\boldsymbol{\theta}}([X]_{i,j,:}), \quad Y \sim (H, W, C').$$



The number of parameters does not depend on the sequence length or on the size of the image. Transformers use these **positionwise MLPs** in every block.

A positionwise MLP mixes the features (channels) within a position, but no information moves between positions.

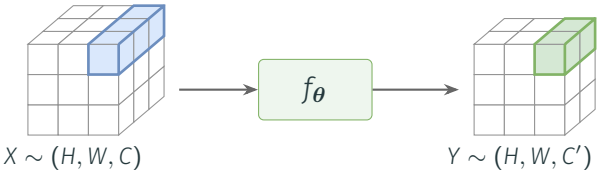
To exchange information, we need convolutions, attention, or a recurrent state.

A simple way to process the whole image is to **vectorize** it, apply a fully-connected layer, and reshape the output:

$$\underset{(HWC)}{\mathbf{x}} = \mathrm{vect}(X), \qquad Y = \mathrm{unvect}(\phi(\mathbf{W}\mathbf{x} + \mathbf{b}))\,.$$

We can choose the output shape to be (H, W, C'), so that the layer can be composed with other image layers.

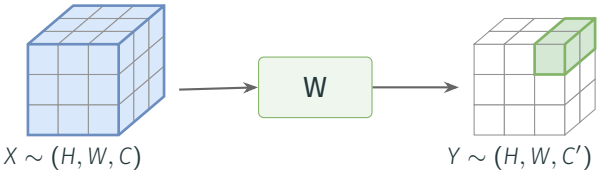
Now every output pixel can depend on the entire image. However, the weight matrix has $HWC \cdot HWC'$ parameters. Nothing in this layer forces it to use the spatial structure of the image.



Positionwise MLP

$[Y]_{i,j,:}$ sees only $[X]_{i,j,:}$

weights do not depend on H, W



Vectorized image

$[Y]_{i,j,:}$ sees all of X

a layer has $HWC \cdot HWC'$ weights

In between, each output pixel sees only a small **neighborhood** (locality), and we can share the weights across positions.

As a running example to visualize what follows, consider a 1D sequence (think of this as "*4 pixels with a single channel*"):

$$\mathbf{x} = [x_0, x_1, x_2, x_3]$$

In this case, we do not need any reshaping operations, and the previous layer (with $C' = 1$) can be written as:

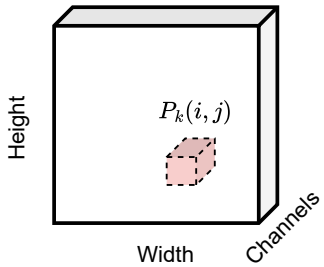
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} W_{00} & W_{01} & W_{02} & W_{03} \\ W_{10} & W_{11} & W_{12} & W_{13} \\ W_{20} & W_{21} & W_{22} & W_{23} \\ W_{30} & W_{31} & W_{32} & W_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Convolutional neural networks

Convolutional layers

With a **kernel size** $s = 2k + 1$, we take the $s \times s$ neighborhood centered at (i, j), including all C channels.
We write this local tensor as:

$$P_k(i, j) \sim (s, s, C).$$



A layer is **local** if $[Y]_{i,j,:}$ depends only on $P_k(i, j)$ for some **small** k (with $s \ll H, W$).

As a first step, we apply a **different** affine map to each neighborhood:

$$[Y]_{i,j,:} = \phi\left(\underset{(s^2C)}{\mathbf{W}_{ij}^\top \operatorname{vect}(P_k(i,j))} + \mathbf{b}_{ij}\right).$$

This **locally connected layer** has $HW(s^2CC' + C')$ parameters, which still grow with the image. On our running example with $s = 3$:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} W_{00} & W_{01} & 0 & 0 \\ W_{10} & W_{11} & W_{12} & 0 \\ 0 & W_{21} & W_{22} & W_{23} \\ 0 & 0 & W_{32} & W_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

We learn ten coefficients instead of sixteen. However, a pattern learned in one corner of the image must be learned again everywhere else.

We apply the same affine map and activation to every neighborhood:

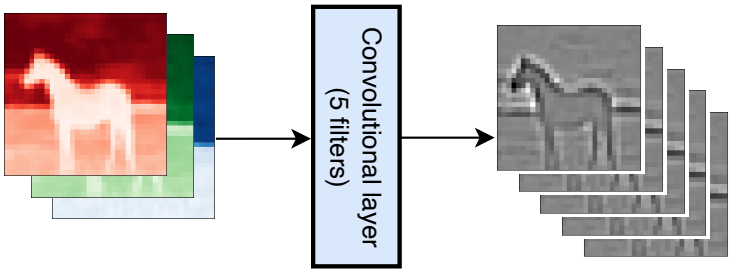
$$[Y]_{i,j,:} = \phi\left(\underset{(s^2C)}{\mathbf{W}^\top \operatorname{vect}(P_k(i,j))} + \mathbf{b}\right).$$

Unlike the locally connected layer, we share the weights $\mathbf{W} \sim (s^2C, C')$ and bias $\mathbf{b} \sim (C')$ **across all locations**.

This is a **convolutional layer**. It has $s^2CC' + C'$ parameters, regardless of the image height and width.

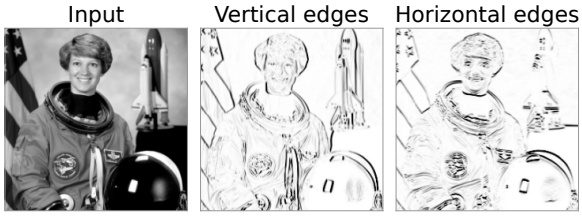
The convolution is the affine map inside the parentheses. We apply the usual pointwise activation ϕ to its output.

Each column of $\mathbf{W}$ defines a **filter**, producing one output channel. A spatial slice $[Y]_{:,:,a}$ is an **activation map**.



A filter responds where its local pattern appears. With one channel and $s = 3$, this filter detects vertical edges, and its transpose horizontal ones:

$$K_V = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$



A trained CNN learns similar edge and color detectors in its first layer. Later layers combine them into textures, parts and objects: a **feature hierarchy**.

Outputs in absolute value; image: NASA, via scikit-image. Zeiler and Fergus (2014), *Visualizing and Understanding Convolutional Networks*, ECCV.

For our four-element sequence, a size-3 kernel produces only two outputs without padding. We add one zero at each end to obtain four outputs:

$$\widetilde{\mathbf{x}} = \begin{bmatrix} 0 & x_0 & x_1 & x_2 & x_3 & 0 \end{bmatrix}^\top.$$

With **zero padding**, we treat values outside the input as zero.

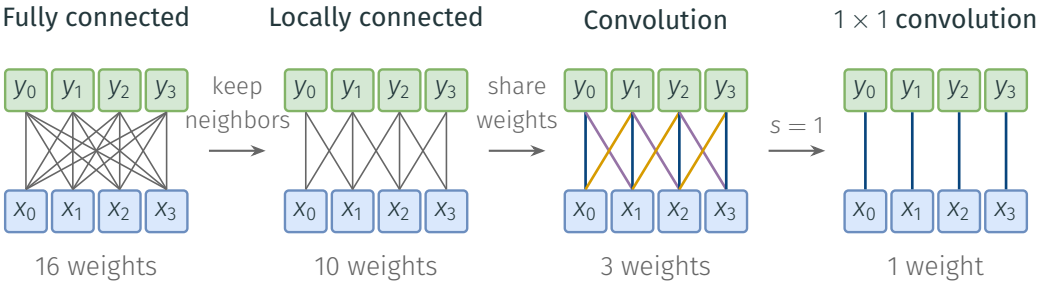
> For an odd kernel size s, we add $k = (s-1)/2$ zeros at each end to preserve the input length when evaluating every position.

In 2D, we pad each border with k rows or columns: one for a 3×3 kernel, two for a 5×5 kernel.

We apply a size-3 kernel to the padded sequence, without bias or activation:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} W_0 & W_1 & W_2 & 0 & 0 & 0 \\ 0 & W_0 & W_1 & W_2 & 0 & 0 \\ 0 & 0 & W_0 & W_1 & W_2 & 0 \\ 0 & 0 & 0 & W_0 & W_1 & W_2 \end{bmatrix} \begin{bmatrix} 0 \\ x_0 \\ x_1 \\ x_2 \\ x_3 \\ 0 \end{bmatrix}.$$

This is a **Toeplitz** matrix: the weights are constant along each diagonal. We learn three coefficients, compared with sixteen for the earlier dense layer.



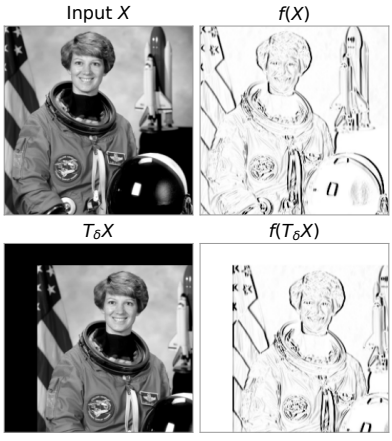
Each layer simplifies the previous one: locality removes connections, and weight sharing ties the remaining ones (same color, same weight). With $s = 1$, a convolution is a layer of the positionwise MLP we started from: a 1×1 **convolution**.

Shared weights give the same response to the same neighborhood (patch), wherever it appears in the image.

We write T_δ for a shift by δ pixels. A layer f is **translation equivariant** if shifting the input shifts its output by the same amount:

$$f(T_\delta X) = T_\delta f(X).$$

For convolution evaluated at every pixel, this holds away from image boundaries. Zero padding can change the response near an edge.

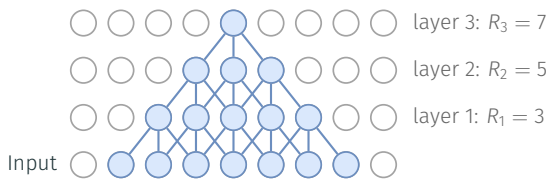


Each convolutional layer is local, but larger regions of the input image can be processed by stacking them.

The **receptive field** of $[Y]_{i,j,:}$ is the region of the input that can affect that output.

Away from the boundaries, L layers with kernel size s, evaluated at every pixel, give a receptive field of width:

$$R_L = 1 + L(s-1).$$



We can collect flattened, padded neighborhoods as columns of $\mathbf{U}(X)$:

$$\underset{(C',HW)}{\widetilde{\mathbf{Z}}} = \underset{(C',s^2C)}{\mathbf{W}^\top} \underset{(s^2C,HW)}{\mathbf{U}(X)} + \mathbf{b}.$$

```
import torch
X = torch.randn(1, 3, 64, 64) # (B, C, H, W)
conv = torch.nn.Conv2d(3, 100, 5, padding=2)
Z = conv(X) # (1, 100, 64, 64)
```

PyTorch is channels-first, (B, C, H, W), while our math is channels-last, (H, W, C). In practice, `nn.Conv2d` uses a specialized implementation for the hardware, so we do not construct $\mathbf{U}(X)$ explicitly.

PyTorch: Unfold and NVIDIA: convolution implementations.

Convolutional neural networks

Building a CNN

We can define a convolutional block by composing convolutional layers f_i, each including its activation function:

$$Y = (f_L \circ \cdots \circ f_2 \circ f_1)(X).$$

With the padding introduced earlier, these blocks can change the number of channels, but they keep the resolution (H, W). We often halve it between blocks.

After we halve the resolution, every later layer grows the receptive field **twice as fast**: $2(s-1)$ input pixels per layer instead of $s-1$. The following layers also process four times fewer pixels, so they are cheaper to run.

In a convolution with **stride**, we evaluate every r-th position along each spatial axis, where r is the stride parameter.

For example, for $r = 2$, we have:

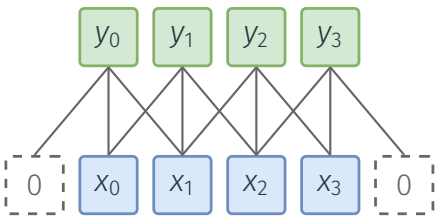
$$[Y]_{i,j,:} = \phi(\mathbf{W}^\top \operatorname{vect}(P_k(2i, 2j)) + \mathbf{b}) .$$

With the padding introduced earlier, $Y \sim (\lceil H/2 \rceil, \lceil W/2 \rceil, C')$, which becomes $(H/2, W/2, C')$ when H and W are even.

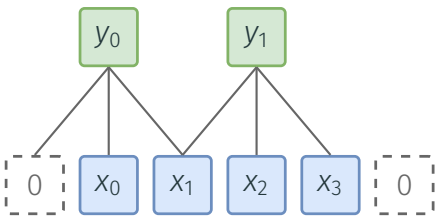
The `nn.Conv2d` layer we saw before has a `stride` parameter.

On our running example, a size-3 kernel with one zero of padding at each end:

Stride 1: four outputs



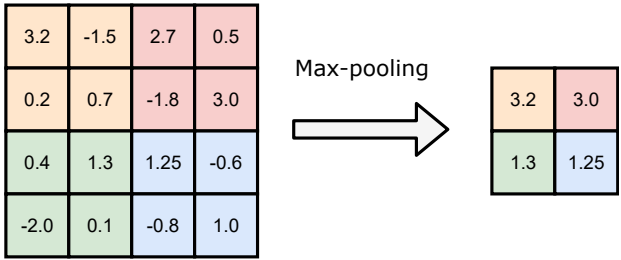
Stride 2: two outputs



With stride 2, we keep every other output of the stride-1 layer.

A **max-pooling** layer takes the maximum over small blocks of the input. With 2×2 blocks:

$$[Y]_{i,j,a} = \max_{u,v \in \{0,1\}} [X]_{2i+u,\, 2j+v,\, a}.$$



It acts on each channel separately and has no parameters. With the mean instead of the maximum, we obtain **average pooling**. Unlike convolutions, we usually take even block sizes (2×2, 4×4, ...).

A standard CNN for classification has three parts, with $Z \sim (H', W', C')$ and $\mathbf{h} \sim (C')$:

Backbone:
convolutional and pooling layers.

$Z = (f_L \circ \cdots \circ f_1)(X)$

Pooling:
global pooling (or flattening).

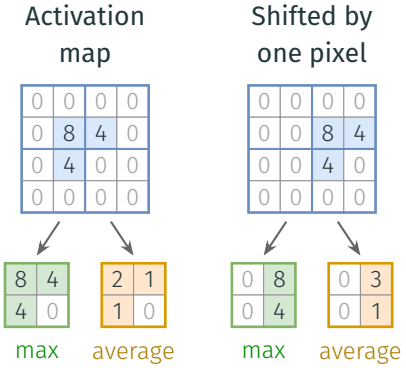
$\mathbf{h} = \frac{1}{H'W'} \sum_{i,j} [Z]_{i,j,:}$

Head:
e.g., an MLP or logistic regression.

$\mathbf{y} = \text{softmax}(g(\mathbf{h}))$

With global pooling, the output is roughly **invariant** to a translation. Image boundaries and downsampling can break exact invariance.

More recent CNNs add many variations to this basic architecture. How to choose the layers and their hyper-parameters is still an open *model selection* research issue.



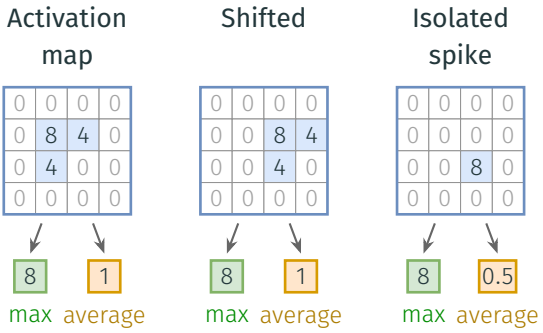
Inside the backbone, each channel is a detector that fires at the few pixels where its pattern appears.

Max pooling keeps the strongest response of each block (*is the pattern somewhere here?*), so a shift inside a block does not change it. The shifted map gives a shifted output: later layers can still combine *where* features are.

Average pooling dilutes each detection and blends nearby ones: the 8 and the 4 become a 3.

Boureau, Ponce and LeCun (2010), *A Theoretical Analysis of Feature Pooling in Visual Recognition*, ICML. Scherer, Müller and Behnke (2010), *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*, ICANN.

We pool globally only at the end, when the receptive field covers the image. The output has C' values for any H' and W'.



Global average pooling averages each channel over all pixels: shifting (or rearranging) the map does not change it.

It measures *how much* of the image shows the pattern. The global max gives 8 for an isolated spike too.

The gradient of the average reaches every position, not only the maximum.

Equivariance and global pooling build invariance into the architecture. We can also obtain it from the data.



With **data augmentation**, we train on random transformations $(\mathcal{A}(X), y)$ of each example that should not change its label.

The label must survive: a flip is fine for a cat, but turns a "b" into a "d".

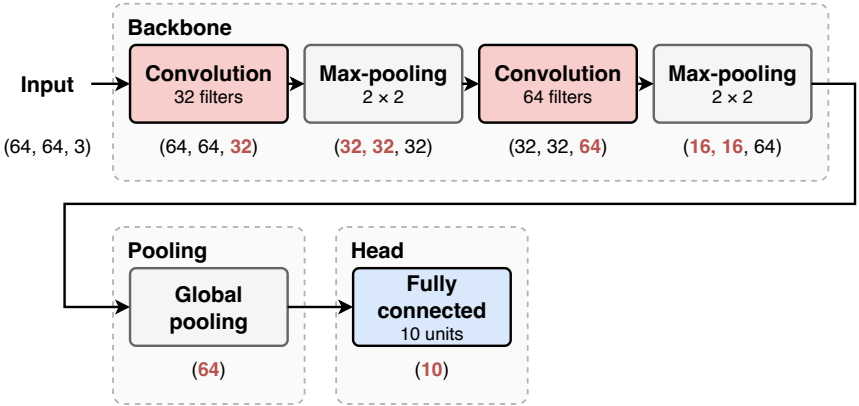


Figure 1: Down-sampling reduces the cost of subsequent layers. Global pooling makes the final classification dimensionality manageable.

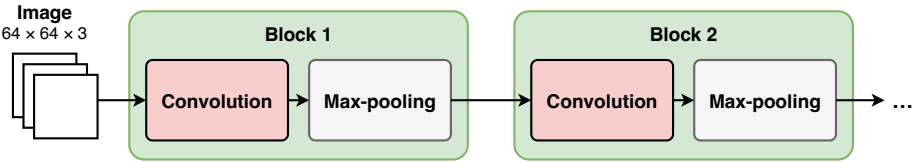


Figure 2: When CNNs become deep, it is simpler to reason in repeating *blocks* made of multiple components. This is easy using the layering abstraction.

We can reshape $\mathbf{W} \sim (s^2C, C')$ into a tensor $K \sim (s, s, C, C')$, with one $s \times s \times C$ filter $[K]_{:,:,:,a}$ for each output channel a. Writing every sum explicitly:

$$[Y]_{i,j,a} = \phi\Big(\sum_{u=-k}^{k} \sum_{v=-k}^{k} \sum_{c=0}^{C-1} [K]_{u+k,\, v+k,\, c,\, a}\, [X]_{i+u,\, j+v,\, c} + b_a \Big).$$

spatial mixing

channel mixing

Each output mixes all positions in the neighborhood and all input channels.

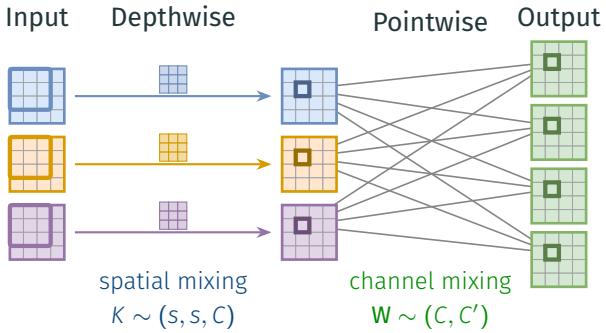
Each layer of the positionwise MLP we saw earlier is a 1×1 **pointwise convolution** (the same channel map at every pixel). Only the channel sum remains:

$$[Y]_{i,j,a} = \phi\Big(\sum_{c=0}^{C-1} [\mathsf{W}]_{c,a}\,[X]_{i,j,c} + b_a\Big), \qquad \mathsf{W} \sim (C, C').$$

With a **depthwise convolution**, each channel has instead its own spatial filter. For $K \sim (s, s, C)$ and $s = 2k + 1$, only the spatial sums remain:

$$[Y]_{i,j,a} = \phi\Big(\sum_{u=-k}^{k} \sum_{v=-k}^{k} [K]_{u+k,\,v+k,\,a}\,[X]_{i+u,\,j+v,\,a} + b_a\Big).$$

Output channel a depends only on input channel a, so the layer keeps $C' = C$.



A depthwise convolution followed by a pointwise convolution is called a **depthwise-separable convolution**: we separate spatial mixing from channel mixing, with $s^2C + CC'$ weights instead of s^2CC'. Processing groups of channels independently gives a *group convolution*.

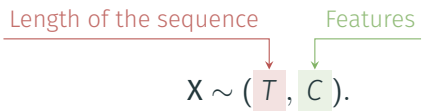
Convolutions beyond images

Sequences, text and generation

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

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

We represent it using a matrix:



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

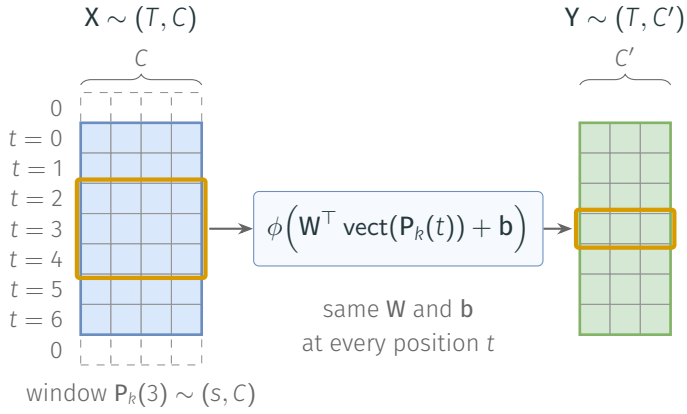
With kernel size $s = 2k+1$, we take the neighborhood $\mathbf{P}_k(t) \sim (s, C)$ around position t, including all C features.

The same definition now gives a **1D convolutional layer**:

$$[\mathbf{Y}]_{t,:} = \phi\Big(\mathbf{W}^\top \operatorname{vect}(\mathbf{P}_k(t)) + \mathbf{b}\Big), \qquad \mathbf{Y} \sim (T, C').$$

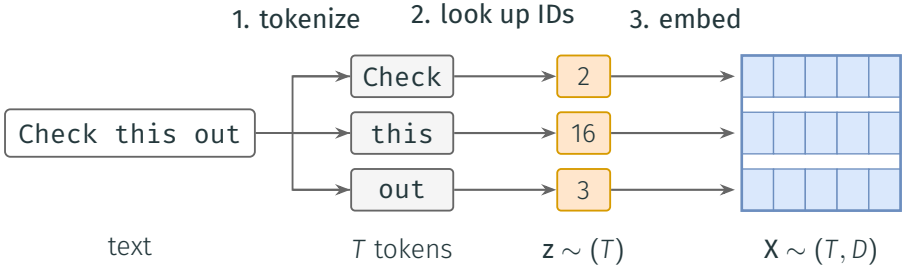
We share $\mathbf{W} \sim (sC, C')$ and $\mathbf{b} \sim (C')$ across positions. With stride 1 and k zeros at each end, the sequence length stays T.

Like in the 2D case, the layer is local and translation equivariant away from the boundaries.



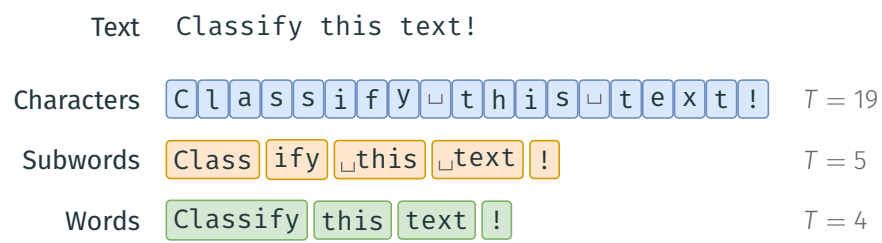
The input is a matrix: the window covers s consecutive rows and all C columns, and gives one output row with C' features. We slide the window along the time axis only.

Text must be preprocessed to be handled by a neural network. We turn it into a matrix in three steps:



By the end of this procedure we have a matrix $\mathbf{X} \sim (T, D)$ of T tokens, each represented by D features.

We first choose how to split (**tokenize**) the text, e.g., by character, by byte, by **subword** or by word:



Larger **tokens** give shorter sequences, but we need many more of them to cover all possible texts. Byte tokenization works like character tokenization on the 256 byte values of UTF-8 text.

Subwords from the GPT-4 tokenizer (`cl100k_base`), via tiktoken.

The set of tokens we can handle is the **vocabulary** or **dictionary**. Its size V depends on the type of token:

Characters or bytes
A small fixed set, e.g., the 256 byte values.

Words
The V most frequent words of a training corpus, plus a special *unknown* token.

Subwords
Reusable pieces learned from a corpus, e.g., with **byte pair encoding** (BPE).

Larger tokens need a larger vocabulary: a few hundred characters, but tens of thousands of words or subwords.

Word-level vocabularies can miss unseen words. Character tokenization produces longer sequences. With **subword tokenization**, we reuse pieces across words, for example (illustrative):

$$\begin{aligned} \texttt{playing} &\longmapsto (\texttt{play}, \texttt{ing}) \\ \texttt{played} &\longmapsto (\texttt{play}, \texttt{ed}) \end{aligned}$$

Each piece has an ID and a row in $\mathbf{E} \sim (V, D)$. Both words share the embedding for `play`.

Unseen words can split into known pieces, and frequent words can stay whole. Learned pieces need not match linguistic word parts.

We can learn the vocabulary with **byte pair encoding** (BPE):

1. Start the vocabulary with the single characters.
2. Merge the most frequent adjacent pair into a new token, **added to the vocabulary**.
3. Repeat until the vocabulary reaches the chosen size V.

Toy corpus: hug $\times 5$, hugs $\times 3$, pug $\times 2$. We count pairs within words:

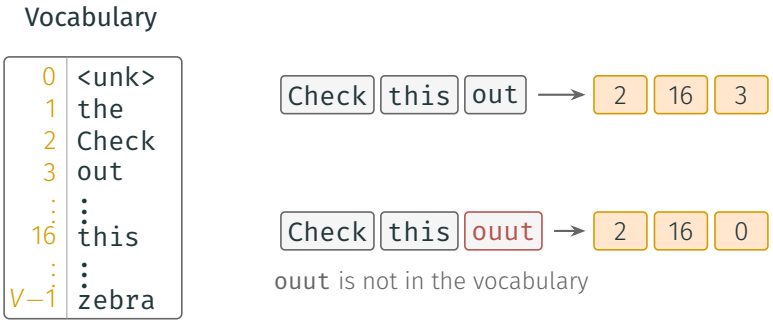
$$\begin{aligned} (\mathsf{u}, \mathsf{g}) &\longmapsto \mathsf{ug} && \text{10 occurrences} \\ (\mathsf{h}, \mathsf{ug}) &\longmapsto \mathsf{hug} && \text{8 occurrences} \end{aligned}$$

The vocabulary grows from $\{\mathsf{g}, \mathsf{h}, \mathsf{p}, \mathsf{s}, \mathsf{u}\}$ to $\{\ldots, \mathsf{ug}\}$ and then $\{\ldots, \mathsf{ug}, \mathsf{hug}\}$.

On new text, we apply the fixed merge rules in their learned order. **Byte-level BPE** starts from 256 byte values, so it can represent any UTF-8 text.

Hugging Face: Byte-Pair Encoding tokenization.

Each token in the text is converted into a discrete ID $z_t \in \{0, \ldots, V-1\}$, its position in the vocabulary, giving us an integer sequence $\mathbf{z} \sim (T)$:



A word outside the vocabulary, such as a typo, becomes the unknown token.

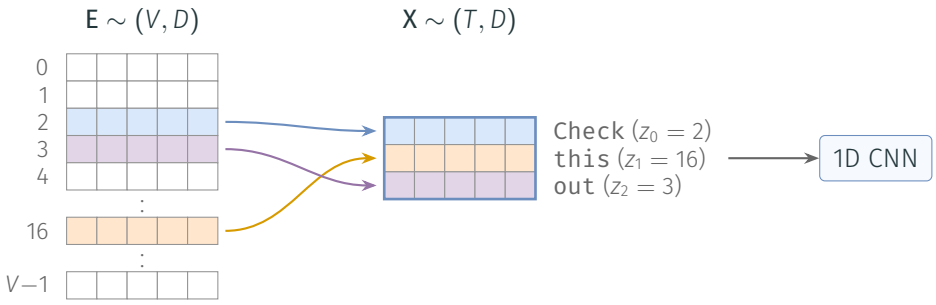
For embedding we learn an **embedding matrix** $\mathsf{E} \sim (V, D)$, with one row for each of the V tokens in the vocabulary. Embedding element z_t of the sequence is just a row lookup:

$$[\mathsf{X}]_{t,:} = [\mathsf{E}]_{z_t,:}$$

We initialize the embeddings randomly and train them with the network, or take them from some pre-trained model or algorithm. For example, GloVe and Word2Vec were two classic algorithms for this purpose.

Pre-trained embeddings also inherit the biases of their training text: for example, word analogies can reproduce gender stereotypes about professions.

Bolukbasi et al. (2016), *Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*, NeurIPS.



For $\mathbf{z} = (2, 16, 3)$, each ID selects one row of $\mathbf{E}$. With custom embeddings, we train $\mathbf{E}$ together with the rest of the network, e.g., a 1D CNN for text classification.

Two texts may have different lengths. Their embedded matrices $\mathbf{X}_1 \sim (T_1, D)$ and $\mathbf{X}_2 \sim (T_2, D)$ share the same feature dimension D.

For $T_1 > T_2$, we can build a padded mini-batch:

$$X = \mathsf{stack}\left(\mathbf{X}_1, \begin{bmatrix} \mathbf{X}_2 \\ \mathbf{0} \end{bmatrix}\right), \qquad X \sim (2, T_1, D).$$

We pad to the maximum sequence length in the mini-batch. The added zero rows are placeholders.

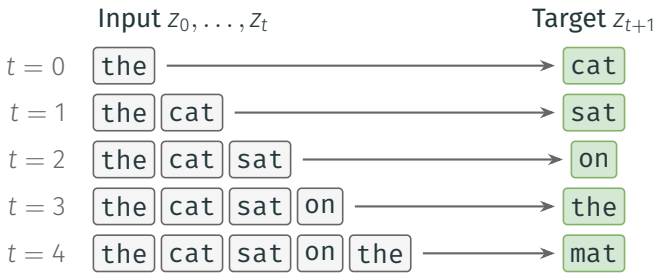
We keep a validity mask of real positions, so padded positions do not contribute to pooling or to the loss.

From the embedded input $\mathbf{X} \sim (T, D)$, the convolutional network produces features $\mathbf{Y} \sim (T, C')$. We can use them in two ways:

- For a **sequence-level task**, such as sequence classification, we pool over the valid positions and apply a classifier. We obtain one prediction for the whole text, such as its topic or sentiment.
- For a **token-level task**, we keep the positions and apply the same classifier g at each one, obtaining one prediction per token, such as whether each word is a noun or a verb.

A token-level task of special interest is **next-token prediction**: at each position, a vocabulary classifier $\mathbf{p}_t = \mathsf{softmax}(g([\mathbf{Y}]_{t,:})) \sim (V)$ predicts the following token z_{t+1}.

The targets come from the text itself, without human labels: the task is **self-supervised**, and a text of $T + 1$ tokens gives T examples:



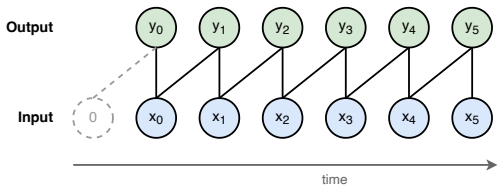
To get all T examples from one forward pass, $\mathbf{p}_t$ must not see z_{t+1}:

> For a sequence $\mathbf{X} \sim (T, C)$, a **causal model** $\mathbf{Y} = f(\mathbf{X})$ is such that $[\mathbf{Y}]_{t,:}$ depends only on $[\mathbf{X}]_{t',:}$ for $t' \leq t$.

We pad with $s-1$ zeros on the left only. The window $\overleftarrow{\mathbf{P}}(t) \sim (s, C)$ covers positions $t-s+1, \ldots, t$:

$$[\mathbf{Y}]_{t,:} = \phi\left(\mathbf{W}^\top \operatorname{vect}(\overleftarrow{\mathbf{P}}(t)) + \mathbf{b}\right).$$

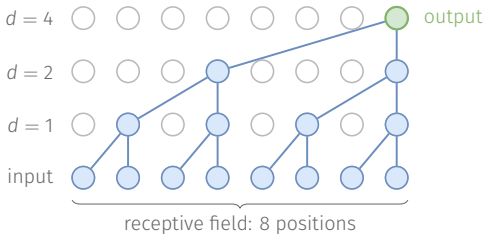
Equivalently, we zero the weights of a centered kernel that read the future.



Kernel size $s = 2$, with one zero on the left.

A causal layer extends the receptive field by $s - 1$ positions, so long contexts need many layers. In a **dilated convolution** with dilation d, the window skips $d - 1$ positions between its taps:

$$\overleftarrow{\mathrm{P}}_d(t) \text{ covers positions } t - (s-1)d,\ \ldots,\ t - d,\ t.$$



With $d = 1, 2, 4, \ldots$ and $s = 2$, L layers see 2^L positions: the receptive field grows **exponentially** with depth.

WaveNet used this design to generate raw audio, one sample at a time.

van den Oord et al. (2016), *WaveNet: A Generative Model for Raw Audio*, arXiv.

We train on a **shifted target**: the input is $z_0, \ldots, z_{T-1}$ and the target is the same text moved by one position, $z_1, \ldots, z_T$. Each prediction $\mathbf{p}_t$ is compared with the next ID, averaging the cross-entropy over the T input positions:

$$\mathcal{L} = -\frac{1}{T} \sum_{t=0}^{T-1} \log [\mathbf{p}_t]_{z_{t+1}}.$$

During training we feed the true tokens (**teacher forcing**), so a causal CNN computes all predictions in parallel without seeing future targets.

Consider the probability $p(\mathbf{z})$ of a whole sequence. By the product rule of probability, we can always write it as

$$p(\mathbf{z}) = \prod_{t=0}^{T-1} p(z_{t+1} \mid z_0, \ldots, z_t),$$

where z_0 is a fixed start token, so it needs no factor. A causal model estimates every factor with the same network, $p(z_{t+1} \mid z_0, \ldots, z_t) \approx [\mathbf{p}_t]_{z_{t+1}}$.

Our loss is then $\mathcal{L} = -\frac{1}{T} \log p(\mathbf{z})$: training maximizes the likelihood of the training texts. A model of $p(\mathbf{z})$ over text is a **language model**. Since we can sample new sequences from it, it is also a **generative model**.

Section 8.4 of the book.

The same next-token predictor can generate text in an **autoregressive** way.

We prompt it with the beginning of the sequence and sample the next token from the predicted distribution. We concatenate its ID to the input and call the model again, continuing one token at a time. For example:

$$\texttt{Check} \longrightarrow \texttt{Check this} \longrightarrow \texttt{Check this out} \quad \ldots$$

The input now includes the model's own sampled tokens. As in the linear models lecture, we can sample with a **temperature** τ: a small τ gives predictable text, a large τ more varied text.

Training can compute many next-token predictions in parallel. Generation must wait for each sampled token before predicting the following one.

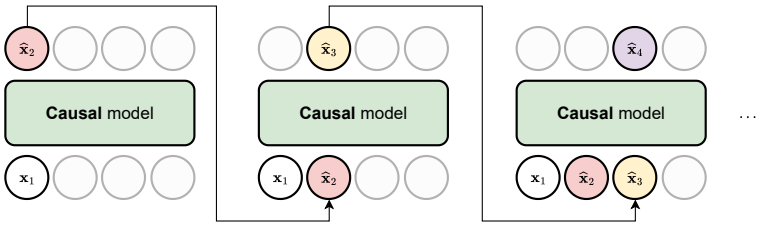
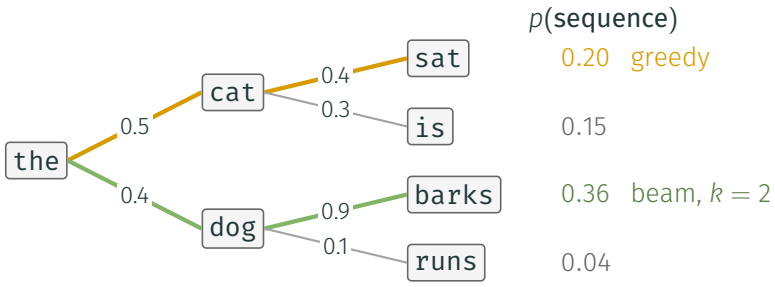


Figure 3: The hat marks a generated value: $\hat{x}_{t+1}$ is the model's prediction for position $t+1$, which is fed back as the next input. For text, we feed back sampled token IDs. Gray circles mark unused positions.

The rule we use to pick each token from $\mathbf{p}_t$ is called **decoding**:

- **Greedy decoding** always takes the most probable token. Since $p(\mathbf{z})$ is a product of many factors, a likely token now can lead to an unlikely sequence later.
- **Sampling** draws the token from $\mathbf{p}_t$, with a temperature τ ($\tau \to 0$ recovers greedy). Top-k and top-p sampling first discard the unlikely tail.
- **Beam search** keeps the k partial sequences with the highest $\log p$, extends each of them, and keeps the best k again.

Holtzman et al. (2020), *The Curious Case of Neural Text Degeneration*, ICLR.



Keeping **dog** lets beam search find a more likely sequence.

This lecture follows **Chapters 7 and 8** of the book.

- In the next lecture, we build deeper models: normalization layers, which extend the idea of normalizing the inputs, and residual connections.
- Then we replace fixed neighborhoods with **attention**, where the interactions between tokens depend on the input.