*Chapter 4*

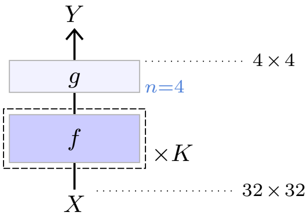
# *Model Components*

A deep model is nothing more than a complex tensorial computation that can ultimately be decomposed into standard mathematical operations from linear algebra and analysis. Over the years, the field has developed a large collection of high-level modules with a clear semantic, and complex models combining these modules, which have proven to be effective in specific application domains.

Empirical evidence and theoretical results show that greater performance is achieved with deeper architectures, that is, long compositions of mappings. As we saw in section § 3.4, training such a model is challenging due to the vanishing gradient, and multiple important technical contributions have mitigated this issue.

## 4.1 The notion of layer

We call layers standard complex compounded tensor operations that have been designed and empirically identified as being generic and efficient. They often incorporate trainable parameters and correspond to a convenient level of granularity for designing and describing large deep models. The term is inherited from simple multi-layer neural networks, even though modern models may take the form of a complex graph of such modules, incorporating multiple parallel pathways.



In the following pages, I try to stick to the convention for model depiction illustrated above:

- operators / layers are depicted as boxes,
- darker coloring indicates that they embed trainable parameters,
- non-default valued hyper-parameters are

added in blue on their right,

- a dashed outer frame with a multiplicative factor indicates that a group of layers is replicated in series, each with its own set of trainable parameters, if any, and

- in some cases, the dimension of their output is specified on the right when it differs from their input.

Additionally, layers that have a complex internal structure are depicted with a greater height.

## 4.2 Linear layers

The most important modules in terms of computation and number of parameters are the Linear layers. They benefit from decades of research and engineering in algorithmic and chip design for matrix operations.

Note that the term “linear” in deep learning generally refers improperly to an affine operation, which is the sum of a linear expression and a constant bias.

### Fully connected layers

The most basic linear layer is the fully connected layer, parameterized by a trainable weight matrix $W$ of size $D' \times D$ and bias vector $b$ of dimension $D'$. It implements an affine transformation generalized to arbitrary tensor shapes, where the supplementary dimensions are interpreted as vector indexes. Formally, given an input $X$ of dimension $D_1 \times \cdots \times D_K \times D$, it computes an output $Y$ of dimension $D_1 \times \cdots \times D_K \times D'$ with

$$\forall d_1,\ldots,d_K, \quad Y[d_1,\ldots,d_K] = WX[d_1,\ldots,d_K] + b.$$

While at first sight such an affine operation seems limited to geometric transformations such as rotations, symmetries, and translations, it can in fact do more than that. In particular, projections for dimension reduction or signal filtering, but also, from the perspective of the dot product being a measure of similarity, a matrix-vector product can be interpreted as computing matching scores between the queries, as encoded by the input vectors, and keys, as encoded by the matrix rows.

As we saw in § 3.3, the gradient descent starts with the parameters' random initialization. If this is done too naively, as seen in § 3.4, the network may suffer from exploding or vanishing activations and gradients [Glorot and Bengio, 2010]. Deep learning frameworks implement initialization methods that in particular scale the random parameters according to the dimension of the input to keep the variance of the activations constant and prevent pathological behaviors.

## *Convolutional layers*

A linear layer can take as input an arbitrarily-shaped tensor by reshaping it into a vector, as long as it has the correct number of coefficients. However, such a layer is poorly adapted to deal-

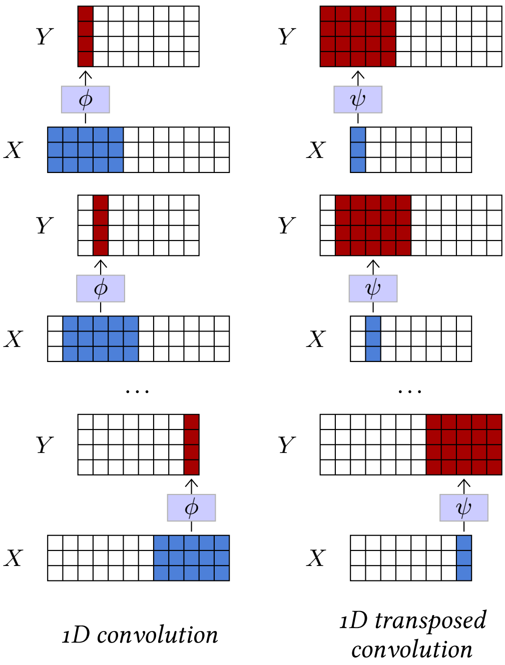


Figure 4.1: *A 1D convolution (left) takes as input a $D \times T$ tensor $X$, applies the same affine mapping $\phi(\cdot; w)$ to every sub-tensor of shape $D \times K$, and stores the resulting $D' \times 1$ tensors into $Y$. A 1D transposed convolution (right) takes as input a $D \times T$ tensor, applies the same affine mapping $\psi(\cdot; w)$ to every sub-tensor of shape $D \times 1$, and sums the shifted resulting $D' \times K$ tensors. Both can process inputs of different sizes.*

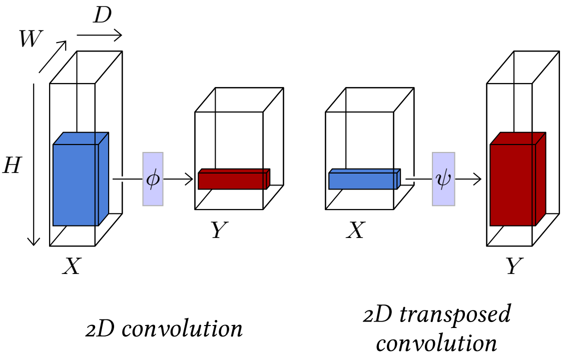


Figure 4.2: *A 2D convolution (left) takes as input a* $D \times H \times W$ *tensor* $X$*, applies the same affine mapping* $\phi(\cdot; w)$ *to every sub-tensor of shape* $D \times K \times L$*, and stores the resulting* $D' \times 1 \times 1$ *tensors into* $Y$*. A 2D transposed convolution (right) takes as input a* $D \times H \times W$ *tensor, applies the same affine mapping* $\psi(\cdot; w)$ *to every* $D \times 1 \times 1$ *sub-tensor, and sums the shifted resulting* $D' \times K \times L$ *tensors into* $Y$*.*

ing with large tensors, since the number of parameters and number of operations are proportional to the product of the input and output dimensions. For instance, to process an RGB image of size $256 \times 256$ as input and compute a result of the same size, it would require approximately $4 \times 10^{10}$ parameters and multiplications.

Besides these practical issues, most of the high-dimension signals are strongly structured. For

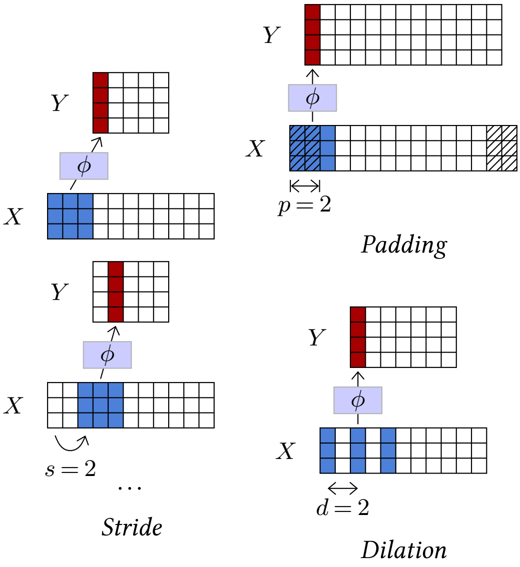


Figure 4.3: *Beside its kernel size and number of input / output channels, a convolution admits three hyper-parameters: the stride* $s$ *(left) modulates the step size when going through the input tensor, the padding* $p$ *(top right) specifies how many zero entries are added around the input tensor before processing it, and the dilation* $d$ *(bottom right) parameterizes the index count between coefficients of the filter.*

instance, images exhibit short-term correlations and statistical stationarity with respect to translation, scaling, and certain symmetries. This is not reflected in the inductive bias of a fully connected layer, which completely ignores the signal structure.

To leverage these regularities, the tool of choice is convolutional layers, which are also affine, but process time-series or 2D signals locally, with the same operator everywhere.

A 1D convolution is mainly defined by three hyper-parameters: its kernel size $K$, its number of input channels $D$, its number of output channels $D'$, and by the trainable parameters $w$ of an affine mapping $\phi(\cdot;w):\mathbb{R}^{D\times K}\to\mathbb{R}^{D'\times 1}$.

It can process any tensor $X$ of size $D\times T$ with $T\geq K$, and applies $\phi(\cdot;w)$ to every sub-tensor of size $D\times K$ of $X$, storing the results in a tensor $Y$ of size $D'\times(T-K+1)$, as pictured in Figure 4.1 (left).

A 2D convolution is similar but has a $K\times L$ kernel and takes as input a $D\times H\times W$ tensor (see Figure 4.2, left).

Both operators have for trainable parameters those of $\phi$ that can be envisioned as $D'$ filters of size $D \times K$ or $D \times K \times L$ respectively, and a bias vector of dimension $D'$.

Such a layer is equivariant to translation, meaning that if the input signal is translated, the output is similarly transformed. This property results in a desirable inductive bias when dealing with a signal whose distribution is invariant to translation.

They also admit three additional hyper-parameters, illustrated on Figure 4.3:

- The padding specifies how many zero coefficients should be added around the input tensor before processing it, particularly to maintain the tensor size when the kernel size is greater than one. Its default value is $0$.

- The stride specifies the step size used when going through the input, allowing one to reduce the output size geometrically by using large steps. Its default value is $1$.

- The dilation specifies the index count between the filter coefficients of the local affine operator. Its default value is $1$, and greater values correspond to inserting zeros between the coefficients, which increases the filter / kernel size while keeping the number of trainable parame-

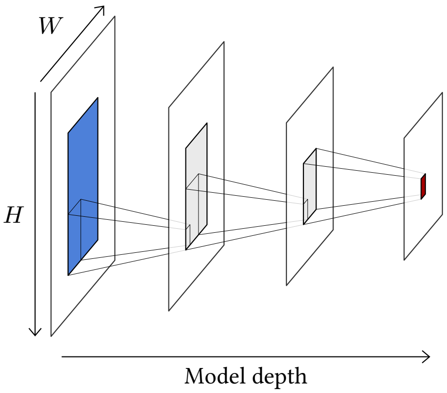


Figure 4.4: *Given an activation in a series of convolution layers, here in red, its* <u>*receptive field*</u> *is the area in the input signal, in blue, that modulates its value. Each intermediate convolutional layer increases the width and height of that area by roughly those of the kernel.*

ters unchanged.

Except for the number of channels, a convolution's output is usually smaller than its input. In the 1D case without padding nor dilation, if the input is of size $T$, the kernel of size $K$, and the stride is $S$, the output is of size $T' = (T - K)/S + 1$.

Given an activation computed by a convolutional layer, or the vector of values for all the channels at a certain location, the portion of the input signal that it depends on is called its receptive field (see Figure 4.4). One of the $H \times W$ sub-tensors corresponding to a single channel of a $D \times H \times W$ activation tensor is called an activation map.

Convolutions are used to recombine information, generally to reduce the spatial size of the representation, in exchange for a greater number of channels, which translates into a richer local representation. They can implement differential operators such as edge-detectors, or template matching mechanisms. A succession of such layers can also be envisioned as a compositional and hierarchical representation [Zeiler and Fergus, 2014], or as a diffusion process in which information can be transported by half the kernel size when passing through a layer.

A converse operation is the transposed convolution that also consists of a localized affine operator, defined by similar hyper and trainable parameters as the convolution, but which, for instance, in the 1D case, applies an affine mapping $\psi(\cdot;w) : \mathbb{R}^{D \times 1} \rightarrow \mathbb{R}^{D' \times K}$, to every $D \times 1$ sub-tensor of the input, and sums the shifted $D' \times K$ resulting tensors to compute its output. Such an operator increases the size of the signal and can be understood intuitively as a synthesis process (see Figure 4.1, right, and Figure 4.2, right).

A series of convolutional layers is the usual architecture for mapping a large-dimension signal, such as an image or a sound sample, to a low-dimension tensor. This can be used, for instance, to get class scores for classification or a compressed representation. Transposed convolution layers are used the opposite way to build a large-dimension signal from a compressed representation, either to assess that the compressed representation contains enough information to reconstruct the signal or for synthesis, as it is easier to learn a density model over a low-dimension representation. We will revisit this in § 5.2.

## *4.3 Activation functions*

If a network were combining only linear components, it would itself be a linear operator, so it is essential to have non-linear operations. These are implemented in particular with activation functions, which are layers that transform each component of the input tensor individually through a mapping, resulting in a tensor of the same shape.

There are many different activation functions, but the most used is the Rectified Linear Unit (ReLU) [Glorot et al., 2011], which sets negative values to zero and keeps positive values unchanged (see Figure 4.5, top right):

$$\mathrm{relu}(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{otherwise.} \end{cases}$$

Given that the core training strategy of deep-learning relies on the gradient, it may seem problematic to have a mapping that is not differentiable at zero and constant on half the real line. However, the main property gradient descent requires is that the gradient is informative on average. Parameter initialization and data normalization make half of the activations positive

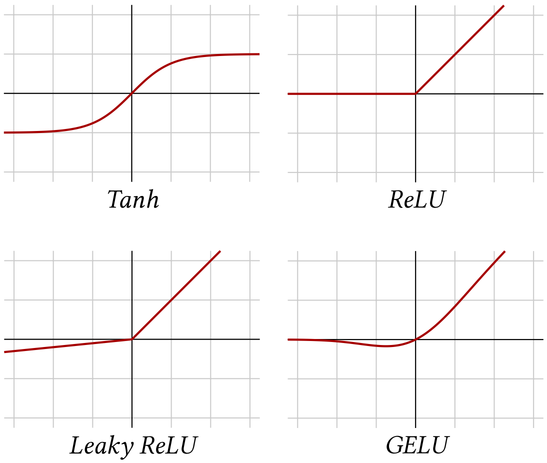


Figure 4.5: *Activation functions.*

when the training starts, ensuring that this is the case.

Before the generalization of ReLU, the standard activation function was the hyperbolic tangent (Tanh, see Figure 4.5, top left) which saturates exponentially fast on both the negative and positive sides, aggravating the vanishing gradient.

Other popular activation functions follow the same idea of keeping positive values unchanged and squashing the negative values. Leaky ReLU [Maas et al., 2013] applies a small positive multiplying factor to the negative values (see Figure 4.5, bottom left):

$$\operatorname{leaky relu}(x) = \begin{cases} ax \text{ if } x < 0, \\ \quad x \text{ otherwise.} \end{cases}$$

And GELU [Hendrycks and Gimpel, 2016] is defined using the cumulative distribution function of the Gaussian distribution, that is:

$$\operatorname{gelu}(x) = xP(Z \leq x),$$

where $Z \sim \mathcal{N}(0,1)$. It roughly behaves like a smooth ReLU (see Figure 4.5, bottom right).

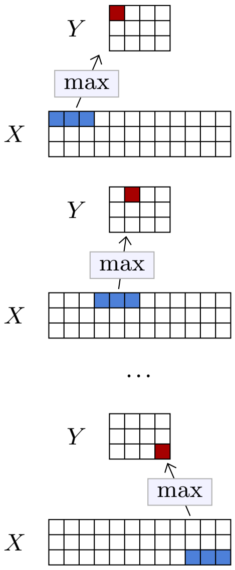
The choice of an activation function, in particular among the variants of ReLU, is generally driven by empirical performance.

## 4.4 *Pooling*

A classical strategy to reduce the signal size is to use a pooling operation that combines multiple activations into one that ideally summarizes the information. The most standard operation of this class is the max pooling layer, which, similarly to convolution, can operate in 1D and 2D and is defined by a kernel size.

In its standard form, this layer computes the maximum activation per channel, over non-overlapping sub-tensors of spatial size equal to the kernel size. These values are stored in a result tensor with the same number of channels as the input, and whose spatial size is divided by the kernel size. As with the convolution, this operator has three hyper-parameters: padding, stride, and dilation, with the stride being equal to the kernel size by default. A smaller stride results in a larger resulting tensor, following the same formula as for convolutions (see § 4.2).

The max operation can be intuitively interpreted as a logical disjunction, or, when it follows a series of convolutional layers that compute local scores for the presence of parts, as a way of encoding that at least one instance of a part is present. It loses precise location, making it



*1D max pooling*

Figure 4.6: *A 1D max pooling takes as input a* $D \times T$ *tensor* $X$*, computes the max over non-overlapping* $1 \times L$ *sub-tensors (in blue) and stores the resulting values (in red) in a* $D \times (T/L)$ *tensor* $Y$*.*

invariant to local deformations.

A standard alternative is the average pooling layer that computes the average instead of the maximum over the sub-tensors. This is a linear operation, whereas max pooling is not.

## 4.5 Dropout

Some layers have been designed to explicitly facilitate training or improve the learned representations.

One of the main contributions of that sort was dropout [Srivastava et al., 2014]. Such a layer has no trainable parameters, but one hyper-parameter, $p$, and takes as input a tensor of arbitrary shape.

It is usually switched off during testing, in which case its output is equal to its input. When it is active, it has a probability $p$ of setting to zero each activation of the input tensor independently, and it re-scales all the activations by a factor of $\frac{1}{1-p}$ to maintain the expected value unchanged (see Figure 4.7).

The motivation behind dropout is to favor meaningful individual activation and discourage group representation. Since the probability that a group of $k$ activations remains intact through a dropout layer is $(1-p)^k$, joint representations become unreliable, making the training procedure avoid them. It can also be seen as a noise injection that makes the training more robust.

When dealing with images and 2D tensors, the

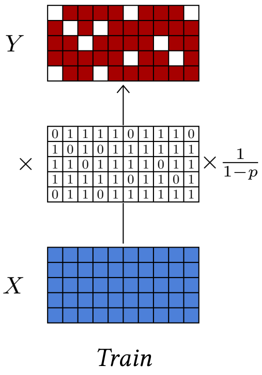


Figure 4.7: *Dropout can process a tensor of arbitrary shape. During training (left), it sets activations at random to zero with probability $p$ and applies a multiplying factor to keep the expected values unchanged. During test (right), it keeps all the activations unchanged.*

short-term correlation of the signals and the resulting redundancy negate the effect of dropout, since activations set to zero can be inferred from their neighbors. Hence, dropout for 2D tensors sets entire channels to zero instead of individual activations (see Figure 4.8).

Although dropout is generally used to improve training and is inactive during inference, it can be used in certain setups as a randomization strategy, for instance, to estimate empirically confidence scores [Gal and Ghahramani, 2015].

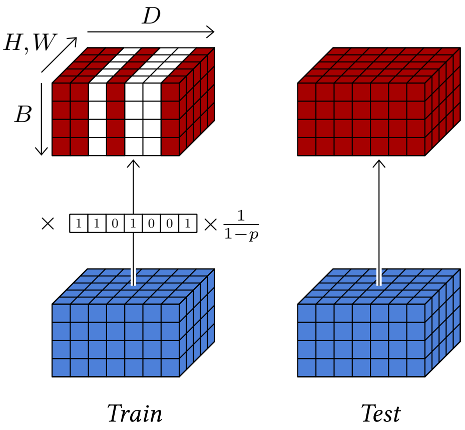


Figure 4.8: *2D signals such as images generally exhibit strong short-term correlation and individual activations can be inferred from their neighbors. This redundancy nullifies the effect of the standard unstructured dropout, so the usual dropout layer for 2D tensors drops entire channels instead of individual values.*

## 4.6 Normalizing layers

An important class of operators to facilitate the training of deep architectures are the normalizing layers, which force the empirical mean and variance of groups of activations.

The main layer in that family is batch normalization [Ioffe and Szegedy, 2015], which is the only standard layer to process batches instead of individual samples. It is parameterized by a hyper-parameter $D$ and two series of trainable scalar parameters $\beta_1,\ldots,\beta_D$ and $\gamma_1,\ldots,\gamma_D$.

Given a batch of $B$ samples $x_1,\ldots,x_B$ of dimension $D$, it first computes for each of the $D$ components an empirical mean $\hat{m}_d$ and variance $\hat{v}_d$ across the batch:

$$\hat{m}_d = \frac{1}{B}\sum_{b=1}^{B} x_{b,d}$$

$$\hat{v}_d = \frac{1}{B}\sum_{b=1}^{B} (x_{b,d} - \hat{m}_d)^2,$$

from which it computes for every component $x_{b,d}$ a normalized value $z_{b,d}$, with empirical mean $0$ and variance $1$, and from it the final result value $y_{b,d}$ with mean $\beta_d$ and standard de-

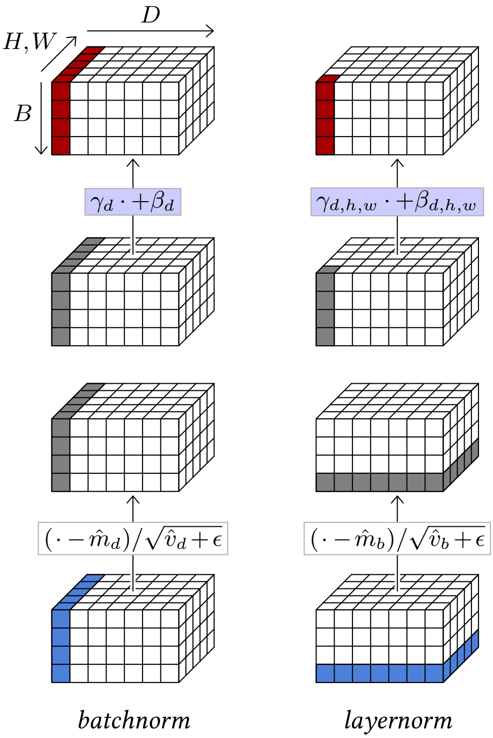

$D$
$H,W$
$B$
$\gamma_d \cdot + \beta_d$
$\gamma_{d,h,w} \cdot + \beta_{d,h,w}$
$(\cdot - \hat{m}_d)/\sqrt{\hat{v}_d + \epsilon}$
$(\cdot - \hat{m}_b)/\sqrt{\hat{v}_b + \epsilon}$
*batchnorm*
*layernorm*

Figure 4.9: *Batch normalization (left) normalizes in mean and variance each group of activations for a given* $d$*, and scales/shifts that same group of activation with learned parameters for each* $d$*. Layer normalization (right) normalizes each group of activations for a certain* $b$*, and scales/shifts each group of activations for a given* $d, h, w$ *with learned parameters indexed by the same.*

viation $\gamma_d$:

$$\forall b, \quad z_{b,d} = \frac{x_{b,d} - \hat{m}_d}{\sqrt{\hat{v}_d + \epsilon}}$$

$$y_{b,d} = \gamma_d z_{b,d} + \beta_d.$$

Because this normalization is defined across a batch, it is done only during training. During testing, the layer transforms individual samples according to the $\hat{m}_d$s and $\hat{v}_d$s estimated with a moving average over the full training set, which boils down to a fixed affine transformation per component.

The motivation behind batch normalization was to avoid that a change in scaling in an early layer of the network during training impacts all the layers that follow, which then have to adapt their trainable parameters accordingly. Although the actual mode of action may be more complicated than this initial motivation, this layer considerably facilitates the training of deep models.

In the case of 2D tensors, to follow the principle of convolutional layers of processing all locations similarly, the normalization is done per-channel across all 2D positions, and $\beta$ and $\gamma$ remain vectors of dimension $D$ so that the scaling/shift does not depend on the 2D position. Hence, if the tensor to be processed is of shape $B \times D \times H \times W$, the layer computes $(\hat{m}_d, \hat{v}_d)$, for $d = 1, \ldots, D$ from the corresponding $B \times H \times W$ slice, normalizes it accordingly, and finally scales and shifts its components with the trainable parameters $\beta_d$ and $\gamma_d$.

So, given a $B \times D$ tensor, batch normalization normalizes it across $b$ and scales/shifts it according to $d$, which can be implemented as a component-wise product by $\gamma$ and a sum with $\beta$. Given a $B \times D \times H \times W$ tensor, it normalizes across $b, h, w$ and scales/shifts according to $d$ (see Figure 4.9, left).

This can be generalized depending on these dimensions. For instance, layer normalization [Ba et al., 2016] computes moments and normalizes across all components of individual samples, and scales and shifts components individually (see Figure 4.9, right). So, given a $B \times D$ tensor, it normalizes across $d$ and scales/shifts also according to the same. Given a $B \times D \times H \times W$ tensor, it normalizes it across $d, h, w$ and scales/shifts according to the same.

Contrary to batch normalization, since it processes samples individually, layer normalization behaves the same during training and testing.

## 4.7 *Skip connections*

Another technique that mitigates the vanishing gradient and allows the training of deep architectures are skip connections [Long et al., 2014; Ronneberger et al., 2015]. They are not layers per se, but an architectural design in which outputs of some layers are transported as-is to other layers further in the model, bypassing processing in between. This unmodified signal can be concatenated or added to the input of the layer the connection branches into (see Figure 4.10). A particular type of skip connections are the residual connections which combine the signal with a sum, and usually skip only a few layers (see Figure 4.10, right).

The most desirable property of this design is to ensure that, even in the case of gradient-killing processing at a certain stage, the gradient will still propagate through the skip connections. Residual connections, in particular, allow for the building of deep models with up to several hundred layers, and key models, such as the residual networks [He et al., 2015] in computer vision (see § 5.2), and the Transformers [Vaswani et al., 2017] in natural language processing (see § 5.3), are entirely composed of blocks of layers with residual connections.

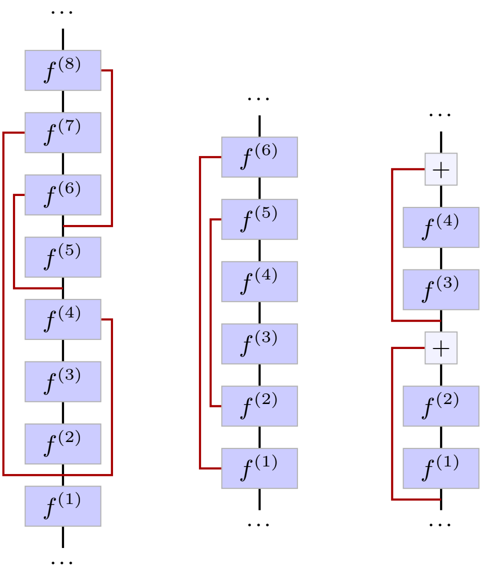


Figure 4.10: *Skip connections, highlighted in red on this figure, transport the signal unchanged across multiple layers. Some architectures (center) that downscale and re-upscale the representation size to operate at multiple scales, have skip connections to feed outputs from the early parts of the network to later layers operating at the same scales [Long et al., 2014; Ronneberger et al., 2015]. The residual connections (right) are a special type of skip connections that sum the original signal to the transformed one, and usually bypass at most a handful of layers [He et al., 2015].*

Their role can also be to facilitate multi-scale reasoning in models that reduce the signal size before re-expanding it, by connecting layers with compatible sizes, for instance for semantic segmentation (see § 6.4). In the case of residual connections, they may also facilitate learning by simplifying the task to finding a differential improvement instead of a full update.

## 4.8 Attention layers

In many applications, there is a need for an operation able to combine local information at locations far apart in a tensor. For instance, this could be distant details for coherent and realistic image synthesis, or words at different positions in a paragraph to make a grammatical or semantic decision in Natural Language Processing.

Fully connected layers cannot process large-dimension signals, nor signals of variable size, and convolutional layers are not able to propagate information quickly. Strategies that aggregate the results of convolutions, for instance, by averaging them over large spatial areas, suffer from mixing multiple signals into a limited number of dimensions.

Attention layers specifically address this problem by computing an attention score for each component of the resulting tensor to each component of the input tensor, without locality constraints, and averaging the features across the full tensor accordingly [Vaswani et al., 2017].

Even though they are substantially more complicated than other layers, they have become a standard element in many recent models. They are, in particular, the key building block of Trans-

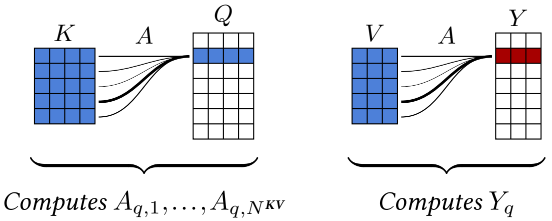


Figure 4.11: *The attention operator can be interpreted as matching every query $Q_q$ with all the keys $K_1,\dots,K_{N^{\text{KV}}}$ to get normalized attention scores $A_{q,1},\dots,A_{q,N^{\text{KV}}}$ (left, and Equation 4.1), and then averaging the values $V_1,\dots,V_{N^{\text{KV}}}$ with these scores to compute the resulting $Y_q$ (right, and Equation 4.2).*

formers, the dominant architecture for Large Language Models. See § 5.3 and § 7.1.

### *Attention operator*

Given

- a tensor $Q$ of queries of size $N^{\text{Q}} \times D^{\text{QK}}$,
- a tensor $K$ of keys of size $N^{\text{KV}} \times D^{\text{QK}}$, and
- a tensor $V$ of values of size $N^{\text{KV}} \times D^{\text{V}}$,

the attention operator computes a tensor

$$Y = \operatorname{att}(Q,K,V)$$

of dimension $N^{\text{Q}} \times D^{\text{V}}$. To do so, it first computes for every query index $q$ and every key index $k$ an attention score $A_{q,k}$ as the softargmax of the dot products between the query $Q_q$ and the keys:

$$A_{q,k} = \frac{\exp\left(\frac{1}{\sqrt{D^{\mathrm{QK}}}} Q_q \cdot K_k\right)}{\sum_l \exp\left(\frac{1}{\sqrt{D^{\mathrm{QK}}}} Q_q \cdot K_l\right)}, \tag{4.1}$$

where the scaling factor $\frac{1}{\sqrt{D^{\mathrm{QK}}}}$ keeps the range of values roughly unchanged even for large $D^{\mathrm{QK}}$.

Then a retrieved value is computed for each query by averaging the values according to the attention scores (see Figure 4.11):

$$Y_q = \sum_k A_{q,k} V_k. \tag{4.2}$$

So if a query $Q_n$ matches one key $K_m$ far more than all the others, the corresponding attention score $A_{n,m}$ will be close to one, and the retrieved value $Y_n$ will be the value $V_m$ associated to that key. But, if it matches several keys equally, then $Y_n$ will be the average of the associated values.

This can be implemented as

$$\mathrm{att}(Q,K,V) = \underbrace{\mathrm{softargmax}\left(\frac{QK^\top}{\sqrt{D^{\mathrm{QK}}}}\right)}_{A} V.$$

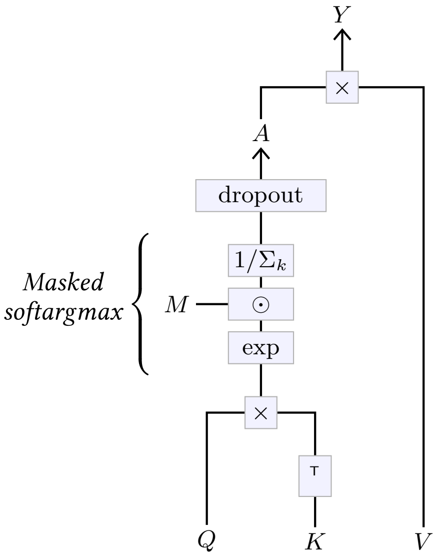


Figure 4.12: *The attention operator $Y = \mathrm{att}(Q,K,V)$ computes first an attention matrix $A$ as the per-query softargmax of $QK^\top$, which may be masked by a constant matrix $M$ before the normalization. This attention matrix goes through a dropout layer before being multiplied by $V$ to get the resulting $Y$. This operator can be made* causal *by taking $M$ full of 1s below the diagonal and zeros above.*

This operator is usually extended in two ways, as depicted in Figure 4.12. First, the attention matrix can be masked by multiplying it before the softargmax normalization by a Boolean matrix $M$. This allows, for instance, to make the operator causal by taking $M$ full of 1s below the diagonal and zero above, preventing $Y_q$ from depending on keys and values of indices $k$ greater than $q$. Second, the attention matrix is processed by a dropout layer (see § 4.5) before being multiplied by $V$, providing the usual benefits during training.

Since a dot product is computed for every query/key pair, the computational cost of the attention operator is quadratic with the sequence length. This happens to be problematic, as some of the applications of these methods require to process sequences of tens of thousands, or more tokens. Multiple attempts have been made at reducing this cost, for instance by combining a dense attention to a local window with a long-range sparse attention [Beltagy et al., 2020], or linearizing the operator to benefit from the associativity of the matrix product and compute the key-value product before multiplying with the queries [Katharopoulos et al., 2020].

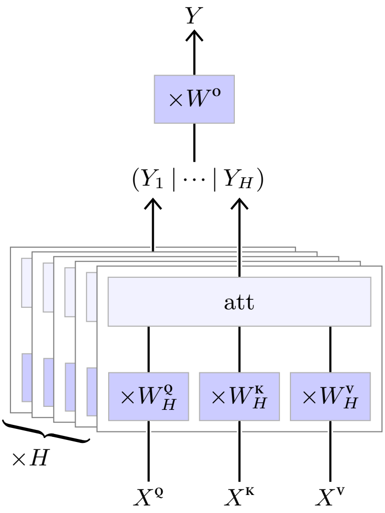

$Y$
$\times W^{\mathrm{O}}$
$(Y_1 \mid \cdots \mid Y_H)$
att
$\times W^{\mathrm{Q}}_H$
$\times W^{\mathrm{K}}_H$
$\times W^{\mathrm{V}}_H$
$\times H$
$X^{\mathrm{Q}}$
$X^{\mathrm{K}}$
$X^{\mathrm{V}}$

Figure 4.13: *The Multi-head Attention layer applies for each of its $h = 1, \ldots, H$ heads a parametrized linear transformation to individual elements of the input sequences $X^{\mathrm{Q}}, X^{\mathrm{K}}, X^{\mathrm{V}}$ to get sequences $Q, K, V$ that are processed by the attention operator to compute $Y_h$. These $H$ sequences are concatenated along features, and individual elements are passed through one last linear operator to get the final result sequence $Y$.*

## *Multi-head Attention Layer*

This parameterless attention operator is the key element in the Multi-Head Attention layer depicted in Figure 4.13. The structure of this layer is defined by several hyper-parameters: a number $H$ of heads, and the shapes of three series of $H$ trainable weight matrices

- $W^{\text{Q}}$ of size $H \times D \times D^{\text{QK}}$,
- $W^{\text{K}}$ of size $H \times D \times D^{\text{QK}}$, and
- $W^{\text{V}}$ of size $H \times D \times D^{\text{V}}$,

to compute respectively the queries, the keys, and the values from the input, and a final weight matrix $W^{\text{O}}$ of size $HD^{\text{V}} \times D$ to aggregate the per-head results.

It takes as input three sequences

- $X^{\text{Q}}$ of size $N^{\text{Q}} \times D$,
- $X^{\text{K}}$ of size $N^{\text{KV}} \times D$, and
- $X^{\text{V}}$ of size $N^{\text{KV}} \times D$,

from which it computes, for $h = 1, \dots, H$,

$$Y_h = \operatorname{att}\left(X^{\text{Q}} W_h^{\text{Q}}, X^{\text{K}} W_h^{\text{K}}, X^{\text{V}} W_h^{\text{V}}\right).$$

These sequences $Y_1, \dots, Y_H$ are concatenated along the feature dimension and each individual element of the resulting sequence is multiplied by $W^{\mathrm{O}}$ to get the final result:

$$Y = (Y_1 \mid \cdots \mid Y_H) W^{\mathrm{O}}.$$

As we will see in § 5.3 and in Figure 5.6, this layer is used to build two model sub-structures: self-attention blocks, in which the three input sequences $X^{\mathrm{Q}}$, $X^{\mathrm{K}}$, and $X^{\mathrm{V}}$ are the same, and cross-attention blocks, where $X^{\mathrm{K}}$ and $X^{\mathrm{V}}$ are the same.

It is noteworthy that the attention operator, and consequently the multi-head attention layer when there is no masking, is invariant to a permutation of the keys and values, and equivariant to a permutation of the queries, as it would permute the resulting tensor similarly.

## 4.9 Token embedding

In many situations, we need to convert discrete tokens into vectors. This can be done with an embedding layer, which consists of a lookup table that directly maps integers to vectors.

Such a layer is defined by two hyper-parameters: the number $N$ of possible token values, and the dimension $D$ of the output vectors, and one trainable $N \times D$ weight matrix $M$.

Given as input an integer tensor $X$ of dimension $D_1 \times \cdots \times D_K$ and values in $\{0,\ldots,N-1\}$ such a layer returns a real-valued tensor $Y$ of dimension $D_1 \times \cdots \times D_K \times D$ with

$$\forall d_1,\ldots,d_K,$$
$$Y[d_1,\ldots,d_K] = M[X[d_1,\ldots,d_K]].$$

## 4.10 Positional encoding

While the processing of a fully connected layer is specific to both the positions of the features in the input tensor and to the positions of the resulting activations in the output tensor, convolutional layers and Multi-Head Attention layers are oblivious to the absolute position in the tensor. This is key to their strong invariance and inductive bias, which is beneficial for dealing with a stationary signal.

However, this can be an issue in certain situations where proper processing has to access the absolute positioning. This is the case, for instance, for image synthesis, where the statistics of a scene are not totally stationary, or in natural language processing, where the relative positions of words strongly modulate the meaning of a sentence.

The standard way of coping with this problem is to add or concatenate to the feature representation, at every position, a positional encoding, which is a feature vector that depends on the position in the tensor. This positional encoding can be learned as other layer parameters, or defined analytically.

For instance, in the original Transformer model, for a series of vectors of dimension $D$, Vaswani et al. [2017] add an encoding of the sequence index as a series of sines and cosines at various frequencies:

$$\text{pos-enc}[t,d] = \begin{cases} \sin\left(\frac{t}{T^{d/D}}\right) & \text{if } d \in 2\mathbb{N} \\ \cos\left(\frac{t}{T^{(d-1)/D}}\right) & \text{otherwise,} \end{cases}$$

with $T = 10^4$.