Deep Learning

Marc Lelarge, Jill-Jenn Vie, and Kevin Scaman

Class notes by Antoine Groudiev

Last modified 14th August 2024

Contents

Abstract

This document is Antoine Groudiev's class notes while following the class *Deep Learning* at the Computer Science Department of ENS Ulm. It is freely inspired by the lectures of Marc Lelarge, Jill-Jênn Vie, and Kevin Scaman.

1 Introduction and general overview

1.1 What is Deep Learning?

1.1.1 Neural networks

1.1.2 Timeline of Deep Learning

1.1.3 Recent applications and breakthroughs

1.1.4 Usual setup

1.1.5 Required skills

1.1.6 Building blocks of deep learning

1.1.7 Why deep learning now?

1.2 Machine Learning pipeline

1.2.1 Cats vs. dogs

1.2.2 Typical Machine Learning setup

1.2.3 Training objective

1.3 Multi-Layer Perceptron

1.3.1 Definition

1.3.2 PyTorch implementation

2 Automatic Differentiation

In the following, we will consider a "set" of data points

$$X \in \mathbb{R}^{N \times d}$$

made of N inputs of size d, and targets

$$Y \in \mathcal{Y}^n$$

where $\mathcal{Y}$ is an arbitrary set. It can be for instance $\mathcal{Y} = \mathbb{R}$ is the case of regression, a finite set such as $[\![1, C]\!]$ in the case of classification, or $\mathcal{Y} = \mathbb{R}^{d'}$ in a more general setup.

2.1 Introduction

As stated previously, neural networks is a very expressive class of functions. However, the associated optimization problem is in general non-convex, giving very few theoretical guarantees and no closed-form expression. In practice, this is not an issue, since such optimization problem can be solved using *gradient descent*.

2.1.1 Loss function

Gradient descent is done by minimizing the average of a differentiable loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. For instance, for regression, we might choose the squared error:

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

For classification, we might choose the logistic loss. Its expression for a two-classes model (that is $y \in \{0, 1\}$) is:

$$\mathcal{L}(\hat{y}, y) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

More generally, for a C-classes model (that is $y \in [\![1, C]\!]$), the cross-entropy loss is:

$$\mathcal{L}(\hat{y}, y) = \sum_{c=1}^{C} y_c \log \hat{y}_c$$

The average of the loss function is then given by:

$$J(f) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\left(f(X_n), Y_n\right)$$

which we will try to minimize.

2.1.2 Gradient descent

The idea behind gradient descent is therefore to be able to compute the gradient of $\mathcal{L}$ with respect to the paramters θ for each point of the dataset:

```
for epoch in range(EPOCHS):
    for x, y in zip(X, Y):
        compute ∇_θ L
        θ = θ − γ∇_θ L
```

Figure 2.1: Pseudo-code of gradient descent

The only remaining challenge is the computation of $\nabla_\theta \mathcal{L}$, preferably automatically; this is the problem which we will address in this chapter.

2.2 Optimization methods

2.2.1 Stochastic Gradient Descent

2.2.2 Batch gradient descent

2.2.3 Minibatch gradient descent

2.2.4 Newton's method

2.3 Computing gradients

2.3.1 By hand

The most straightforward approach to computing $\nabla_\theta \mathcal{L}$ would be to derive it on paper. Nevertheless, this is complicated, as it involves very long computations using matrix calculus.

Furthermore, this approach is not modular, as changing the loss function or adding a layer requires to re-derive the gradient from scratch. Such a method does not scale: if the computations

can be done in a reasonable amount of time for small models using linear and activation layers, complex models introduced in the next chapters have enormous gradient expressions, making the computation way too long and tidious to be done by hand.

2.3.2 Numerical differentiation

A first automatic approach to compute the gradient automatically would be to use *numerical differentiation*, a method to estimate the derivative of the function using finite differences. Recall that since the derivative of a real-valued function is the limit of its growth rate:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

one can approximate the derivative using the slope between two points close to x:

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}$$

for some small number h. However, this approach does not work well in practice because of round-off errors which can have a strong impact on the result and cause gradient descent to diverge.

2.3.3 Symbolic differentiation

To avoid round-off errors, another approach could be to use symbolic differentiation: the idea is to formally compute the expression of the derivative and then to evaluate it numerically. The issue with symbolic differentiation is its scalability: without optimization of the computation, it can produce exponentially large expressions that take a long time to symbolically compute and numerically evaluate. To maintain reasonable expression sizes, one needs to apply simplification operations between each step, resulting in a heavy computational cost.

Hopefully, we do not need all the expressivity that symbolic differentiation has: we are only interested in the numerical evaluation of the derivative; we do not need to keep the formal expression, only the numerical evaluation.

This is the idea of *automatic differentiation* with accumulation: we will keep the idea of symbolic differentiation by computing the derivative as an operation level, but reduce the size of intermediate computations by only keeping the numerical values.

2.4 Automatic differentiation

2.4.1 Computational Graphs

Automatic differentiation uses a data structure called *Computational Graphs* to represent the computation that is happening inside the model.

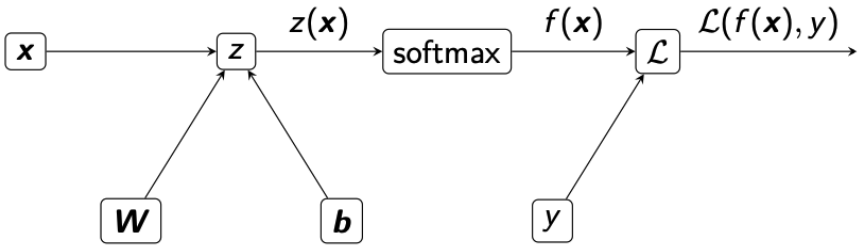

Figure 2.2: Computational graph for $\mathcal{L}(\text{softmax}(Wx + b), y)$

In Figure 2.2, reading from left to right, we can see that the weights W, the biases b and the input x are combined to create a function $z(x)$; to the result of this operation is applied softmax, creating $f(x)$. Finally, $f(x)$ is combined with y using the loss function $\mathcal{L}$, giving the final result of the computation, $\mathcal{L}(f(x), y)$.

Computational graphs can then be used to apply the main algorithm to compute the gradient, called *backpropagation*. We will illustrate its behavior by considering the function $f(x, y, z) = (x + y) \times z$, to which is associated the following computational graph:

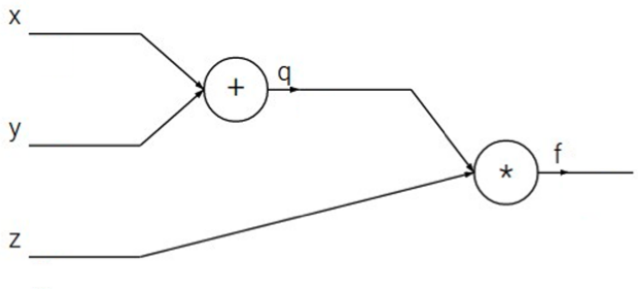

Figure 2.3: Computational graph for $f(x, y, z) = (x + y) \times z$

2.4.2 Forward pass

The first step of backpropagation is to compute the outputs during a *forward pass*. This is simply done by replacing each input by its numerical value, and applying the operations described by the nodes of the graph. Using $x = -2$, $y = 5$, $z = -4$ in the previous example yields:

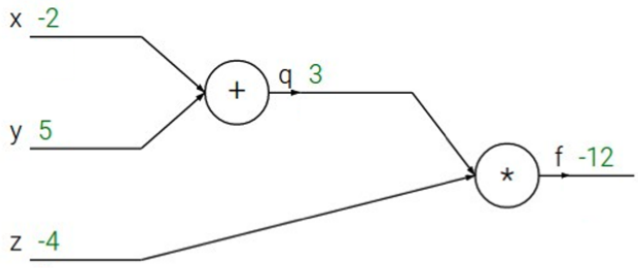

Figure 2.4: Forward pass

2.4.3 Backward pass

The backward pass allows us to compute the derivatives, in our case $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$. Instead of computing the value at each node from left to right (forward pass), we will compute the values of the derivatives from right to left, starting from the output (backward pass).

- Starting from the output node, we have that $\frac{\partial f}{\partial f} = 1$.
- Going backward to the z input node, we have that $\frac{\partial f}{\partial z} = q$, since $f = qz$. We can then use the results of the forward pass to find the value of q, and deduce $\frac{\partial f}{\partial z} = 3$.
- Similarly, $\frac{\partial f}{\partial q} = z = -4$.
- Going further back into the graph, we aim at computing $\frac{\partial f}{\partial y}$. Using chain rule, we know that:

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

This equation can be interpreted in terms of "gradient stream": we want to compute the *downstream gradient* $\frac{\partial f}{\partial y}$, that is the gradient "after the node". This gradient can therefore be expressed using the chain rule as the product of the *local gradient* $\frac{\partial q}{\partial y}$ and the *upstream gradient* $\frac{\partial f}{\partial q}$, that is the gradient computed at the previous node.

$$\underbrace{\frac{\partial f}{\partial y}}_{\text{Downstream}} = \underbrace{\frac{\partial q}{\partial y}}_{\text{Local}} \underbrace{\frac{\partial f}{\partial q}}_{\text{Upstream}}$$

Like in previous cases, we can compute the local gradient $\frac{\partial q}{\partial y} = 1$ since $q = x + y$. The upstream gradient is also known at this point of the pass, due to the backward direction, hence $\frac{\partial f}{\partial y} = 1 \times -4 = -4$

- The same approach using the chain rule can be used for $\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x}\frac{\partial f}{\partial q} = 1 \times -4 = -4$.

2.4.4 Modularity

A benefit of backpropagation is its modularity: the gradient computation can be broke down into the computation of the downstream gradient knowing the upstream gradient and the local gradient of the node.

Consider for instance a function f taking x and y as inputs, and producing an output z. This function is a node somewhere in a possibly very complex computational graph, but we do not need the whole information about the rest of the computation: to compute the downstream gradient, we only need local information, that is upstream and local gradients.

We are given the upstream gradient of the loss that we want to compute with respect to our output, $\frac{\partial \mathcal{L}}{\partial z}$. Our goal is now simply to propagate the gradient computation by providing the downstream derivatives, that is $\frac{\partial \mathcal{L}}{\partial x}$ and $\frac{\partial \mathcal{L}}{\partial y}$. Since we know the expression of f, we are able to compute the local derivatives $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$; using chain rule, we can therefore provide the downstream gradient.

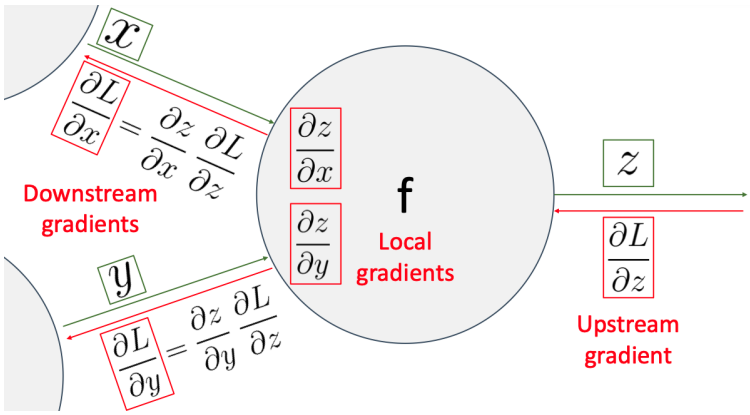

Figure 2.5: Local process of computing the downstream gradient

2.4.5 A complete example

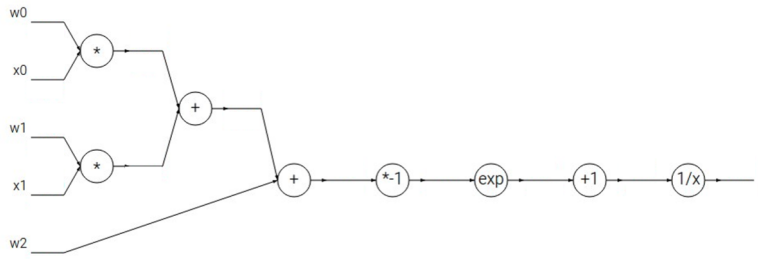

Figure 2.6: Computational graph for the function $f(x, w) = \frac{1}{1+e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$

We do not have to break down the gradient computation only into elementary operations such as additions or multiplications: we can define blocks, such as "Sigmoid", and hard-code their gradients to avoid using automatic differentiation on it.

2.5 Extension to multivariate calculus

So far, we only considered backpropagation in the case of scalars. The same principle can nevertheless be extended to multivariate calculus using vectors and matrices.

2.5.1 Reminder on vector derivatives

For a real-valued function $f : \mathbb{R} \longrightarrow \mathbb{R}$, the regular derivative is a scalar:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial f}{\partial x} \in \mathbb{R}$$

For a function taking a vector and returning a scalar, that is $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, its derivative is its *gradient*, that is:

$$\nabla f \in \mathbb{R}^n \quad \text{where} \quad (\nabla f)_i = \frac{\partial f}{\partial x_i}$$

The i-coordinate of the gradient is the partial derivative of f with respect to the i-th variable of the input vector.

Finally, for a differentiable function taking a vector and returning another vector, that is $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, its derivative is its *Jacobian*, that is:

$$J_f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^\top f_1 \\ \vdots \\ \nabla^\top f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathscr{M}_{m,n}(\mathbb{R})$$

2.5.2 Example: linear layer gradient

Consider a linear layer of the form $f(x) = Wx$ where W is an $m \times n$ matrix. The i-th coordinate of the output of is

$$f_i = W_i x = \sum_j W_{i,j} x_j$$

where W_i is the i-th row of W. Therefore, its Jacobian is:

$$(J_f)_{i,j} = \frac{\partial f_i}{\partial x_j} = W_{i,j}$$

hence $J_f = W$.

2.5.3 Generalized multivariate chain rule

Consider two differentiable functions $f : \mathbb{R}^m \longrightarrow \mathbb{R}^k$ and $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, and $a \in \mathbb{R}^n$. The chain rule is expressed as:

$$D_a(f \circ g) = D_{g(a)}f \circ D_a g \tag{2.5.1}$$

where $D_a f$ for instance is the derivative of f evaluated in a. Furthermore, the Jacobians verify:

$$J_{f \circ g}(a) = J_f(g(a))J_g(a) \tag{2.5.2}$$

3 Optimization and loss functions

3.1 Tensors in `PyTorch`

A tensor is a d-dimensional array in `PyTorch`. Tensors are used in deep learning to represent all kind of data, from images to weight matrices.

Tensor creation A tensor can be created from a list: `x = torch.Tensor([[1,0,2],[3,2,2]])`. By default, tensors are not deepcopied, but can be cloned: `x.clone()`. Each tensor has a data type, which can be specified at its creation: `x = torch.Tensor(..., dtype=torch.int64)`. In the case of data (for instance, training examples), the first dimension of a tensor is usually the samples: `x.shape[0]` is the batch size.

Operations Most operations on tensors (`+`, `*`, ...) are performed coordinate-wise and need matching sizes. Mathematical functions from the `torch` library, such as `torch.exp` or `x**2`, are vectorized and therefore performed coordinate-wise. Notably, matrix multiplication can be perfomed using the `x @ y` syntax.

Shapes A tensor shape can be obtained by calling `x.shape`. Reshaping can be performed using `x.view(1,3,-1)`, where `-1` acts as a wildcard for `PyTorch` to fill in automatically the appropriate dimension. For instance, `x.view(-1)` flattens the tensor into a one-dimensional vector. The operation `x.unsqueeze(0)` adds a dimension of size 1 to the tensor.

Gradients Tensors can have an associated gradient, stored in `x.grad`. We can remove (and clone) this tensor from the computation of the gradient by using `y = x.detach()`.

Other operations See the `PyTorch` documentation for numerous operations defined on tensors. Most convenient ones include `torch.sum(x)`, `torch.mean(x)`. A tensor can also be converted to a `NumPy` array using `x.numpy()` or `x.detach().numpy()`.

Manipulating tensors and tensor sizes is complex and leads to many bugs in deep learning projects. Many errors can go unnoticed due to wrong tensor sizes and Python's dynamic typing. As a general advice, always verify your intermediate computations using for instance `print(x[:5])`, and your tensor shapes with `print(x.shape)`!

3.2 Loss functions

3.2.1 Mean Square Error

Definition (Mean Square Error)**.** The *mean square error* is the loss function defined by:

$$\begin{aligned} \ell : \mathbb{R}^d \times \mathbb{R}^d &\longrightarrow \mathbb{R} \\ x, y &\longmapsto \|x - y\|_2^2 \end{aligned}$$

3.2.2 Cross Entropy

3.3 First-order optimization

3.4 Convergence analysis

3.5 Gradient descent variants

4 Convolutional Neural Networks

4.1 Introduction

Convolutional Neural Networks (CNNs) is a class of models widely used in computer vision. While Fully Connected Neural Networks are very powerful machine learning models, they do not respect the 2D spatial structure of the input images. For instance, training a Multilayer Perceptron on a dataset of 32×32 images required the model to start with a `Flatten` layer, that reshaped matrix images of size `(32, 32)` to flattened vectors of size `(1024, 1)`. Similarly, different color channels were handled separately, reshaping tensor images of dimensions `(32, 32, 3)` to `(3072, 1)`.

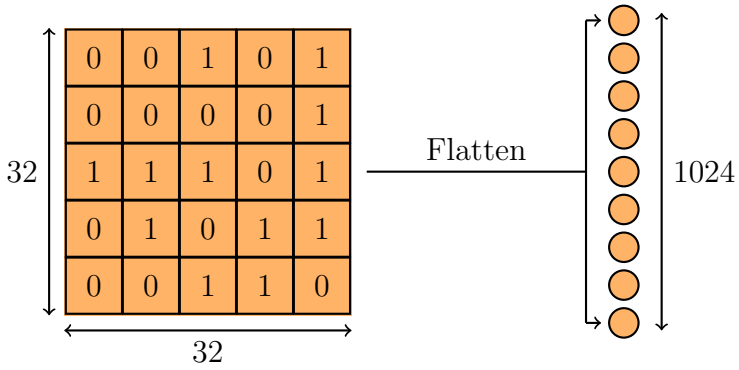

Figure 4.1: Flatten layer breaking the spatial strucutre of input data

CNNs introduce new operators taking advantage of the spatial structure of the input data, while remaining compatible with automatic differentiation. While MLPs build the basic blocks of Deep Neural Networks using Fully-Connected Layers and Activation Layers, this chapter will introduce three new types of layers: *Convolution Layers*, *Pooling Layers*, and *Normalization*.

4.2 Convolution Layers

Similarly to Fully-Connected Layers, *Convolution Layers* have learnable weights, but also have the particularity to respect the spatial information.

4.2.1 Input shape

A Fully-Connected layer (also known as Linear layer) receives some flattened vector and outputs another vector:

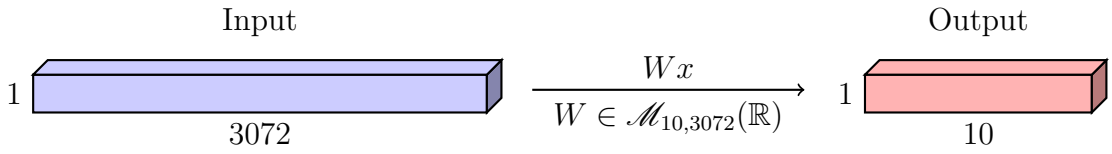

Figure 4.2: Fully-Connected Layer

Instead, a CNN takes as an input a 3D volume: for instance, an image can be represented as a tensor of shape $3 \times 32 \times 32$, the first dimension being the number of channels (red, green, blue), and the other two being the width and height of the image.

4.2.2 Kernels

The convolutional layer itself consists of small kernels (also called filters) used to *convolve* with the image, that is sliding over it spatially, and computing the dot products at each possible location.

Definition (Kernel)**.** A *kernel* (or *filter*) is a tensor of dimensions $D \times K \times K$, where D is the number of channels (or "depth") of the input, and K is a parameter called *kernel size*.

Definition (Convolution of two matrices)**.** Given two matrices $A = (a_{i,j})_{i,j}$ and $B = (b_{i,j})_{i,j}$ in $\mathscr{M}_{m,n}(\mathbb{R})$, the *convolution of A and B*, noted $A * B \in \mathbb{R}$, is the following:

$$A * B = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{(m-i+1),(n-j+1)} \cdot b_{i,j} \tag{4.2.1}$$

This corresponds to the dot product in the space $\mathscr{M}_{m,n}(\mathbb{R})$.

Definition (Kernel convolution)**.** An input of shape $C \times H \times W$ can be processed by a kernel of shape $C \times K \times K$ by computing at each possible spatial position the convolution between the kernel and the submatrix of the input.

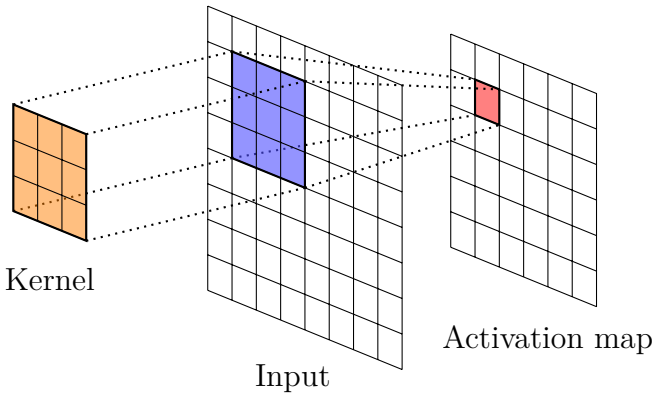

Figure 4.3: Kernel convolution

The output of this operation in an *activation map* of dimension $1 \times (H - K + 1) \times (W - K + 1)$ representing for each pixel the convolution between the kernel and the corresponding chunk of the image.

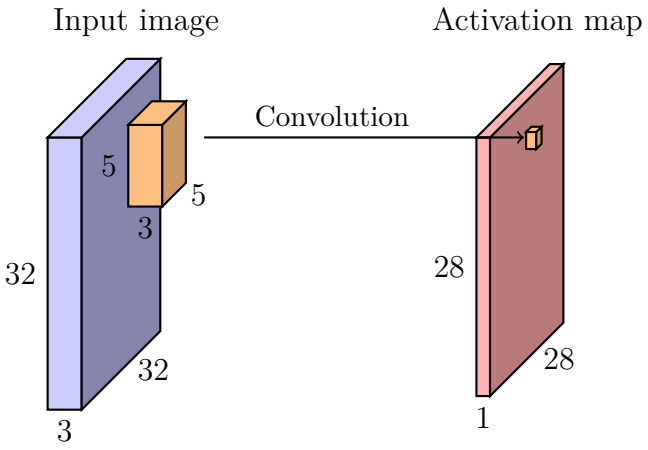

Figure 4.4: Input and output of the convolution operation

Intuitively, the result of the kernel convolution tells us for each pixel *how much the neighbourhood of the input pixel corresponds to the kernel*.

Example (Gaussian blur)**.** Let $G \in \mathscr{M}_3(\mathbb{R})$ be the following kernel:

$$G := \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Each coefficient of this matrix is an approximation of the Gaussian distribution. Applying this kernel to an image produces a smoothed version of the input.

Example (Sobel operator)**.** Let S_x and $S_y \in \mathscr{M}_3(\mathbb{R})$ be the following kernels:

$$S_x := \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad \text{and} \qquad S_y := S_x^\top = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The convolution between these operators and an image produces horizontal and vertical derivatives approximations of the image pixels.

Input image Sobel operator applied to the image

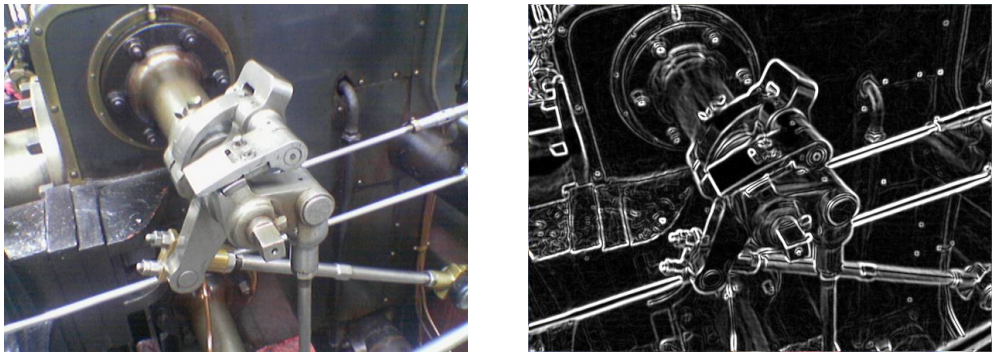

Figure 4.5: Effect of the Sobel operator on an image

These two examples show that kernels used in convolutional layers express meaningful transformations of the input, justifying their use in CNNs. For instance, one could hardcode different

kernels (gaussian blur, Sobel operator, vertical/horizontal lines extraction) to extract interesting features from an image, and plug these features into an MLP to obtain an improved classifier compared to a basic, flattening MLP. We will see that instead, CNNs have learnable kernel weights, allowing the model to choose the kernels that it considers bests.

4.2.3 Multiple kernels

In Figure 4.4, we used simply one kernel to compute one activation map. In practice, we repeat this process multiple times: we consider a set (or *bank*) of filters having different weights values, and for each kernel of the set, we compute its activation map.

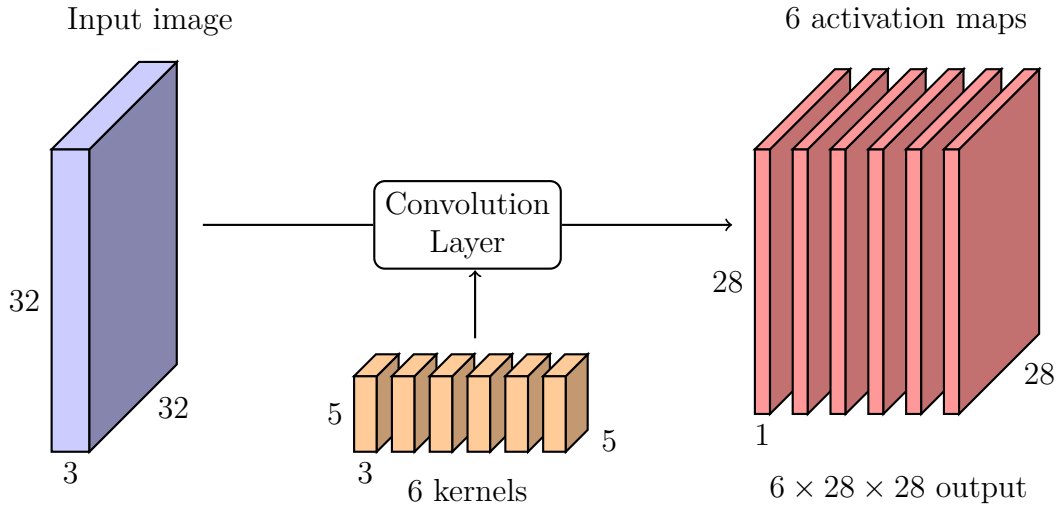

Figure 4.6: Convolutional Layer using 6 kernels

Using a bank containing C' filters, the output of the convolutional layer in an *activation map* of dimension $C' \times (H - K + 1) \times (W - K + 1)$ representing for each pixel the convolution between the given kernel and the corresponding chunk of the image.

Remark (Biases in Convolutional Layers)**.** *Similarly to fully-connected layers, we often add to the activation map of each kernel a bias of size* $1 \times (H - K + 1) \times (W - K + 1)$*. Those biases might me ommited in the rest of the chapter for the sake of simplicity.*

4.2.4 Stacking convolutions

Like previously introduced layers, convolutional layers can be stacked to form deep networks. The layer shapes need to match, in particular the output channels of a layer must match the input channels of the next layer, and the output height and width must match the next input height and width.

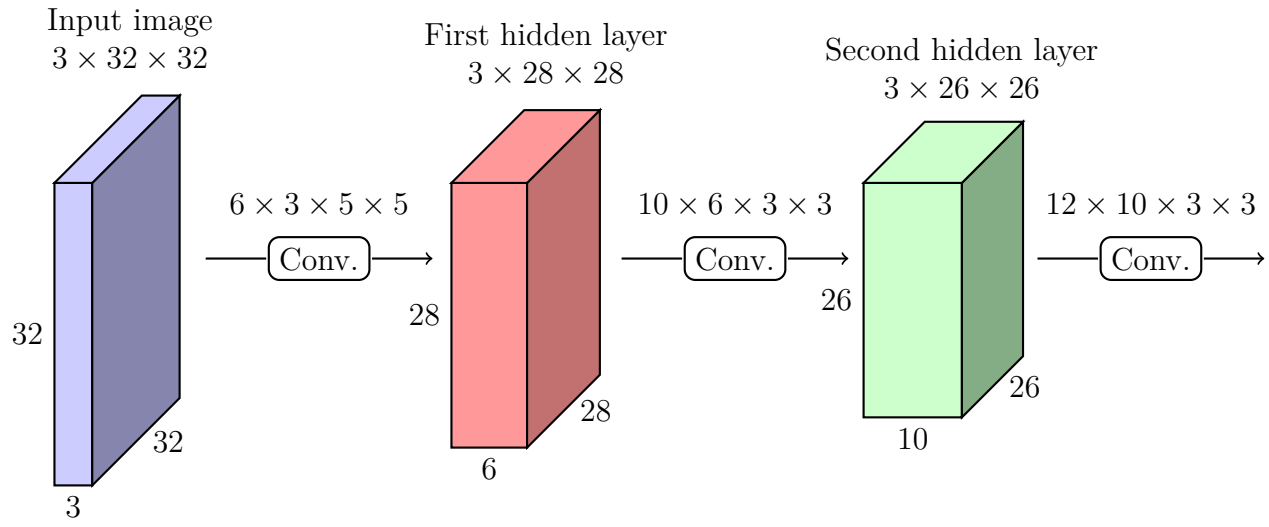

Figure 4.7: Stacking of 3 Convolutional Layers of correct shapes

However, stacking two convolution layers next to each other produces another convolutional layers, and do not add representation power. Therefore, we use the exact same solution as for linear classifiers: we introduce non-linear layers using activation functions in between convolutional layers.

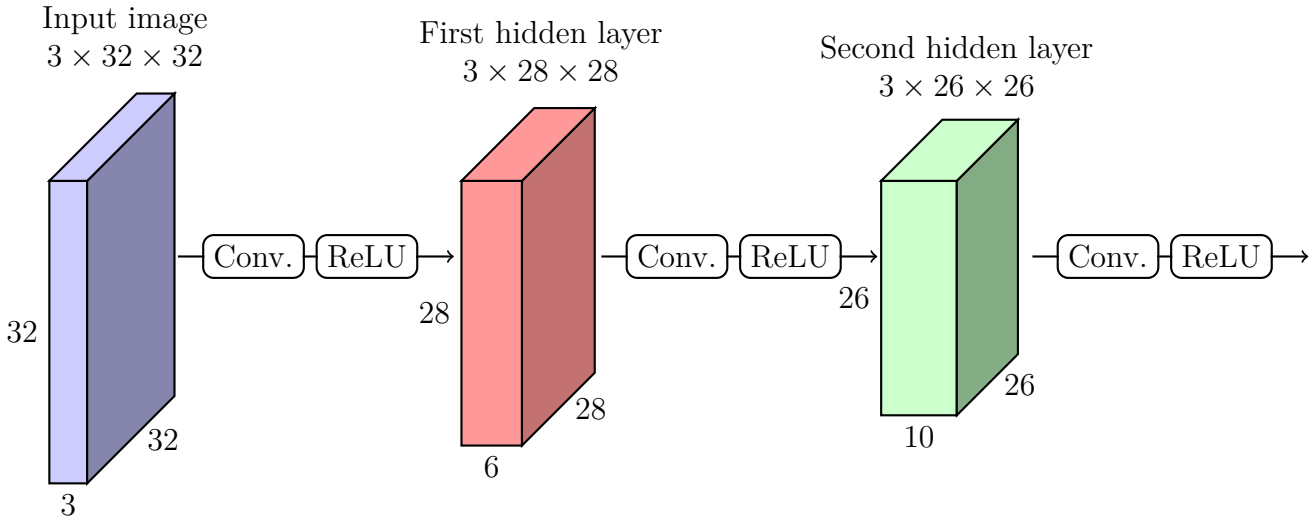

Figure 4.8: Adding ReLU layers in between Convolution Layers

4.2.5 Spatial dimensions and Padding

As stated previously, using an input of width W with a filter of kernel size K, the output width is $W - K + 1$. A problem with the approach is that features maps decrease in size with each layer. This creates an upper bound on the maximum number of layers that we can use for our model.

A solution to this is to introduce *padding* by adding zeros around the border of the input. When the kernel will slide around the edges of the input, a part of the coefficients that it will consider in its convolution will be zeros.

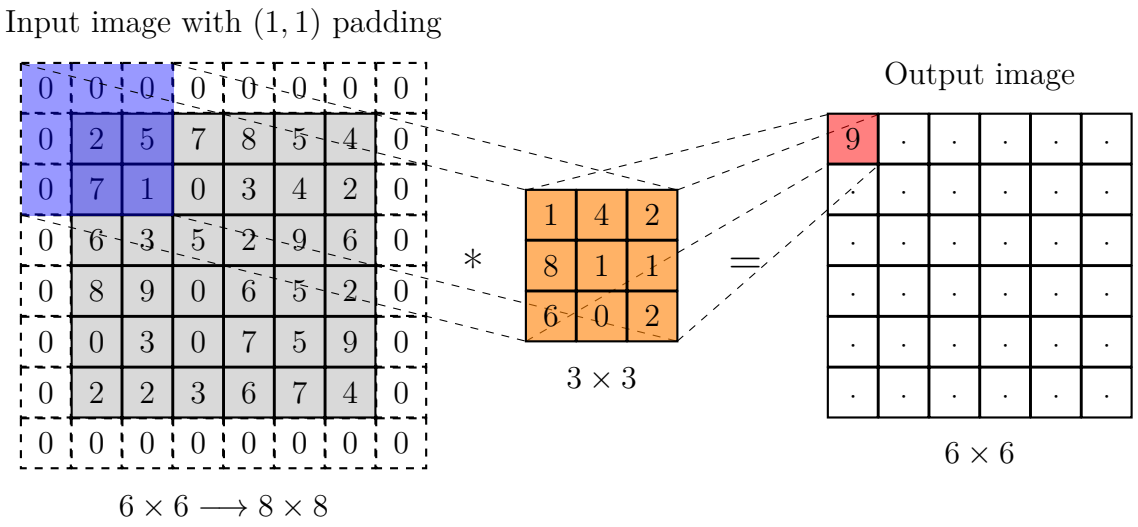

Figure 4.9: Adding padding around the input

Remark (Padding strategies)**.** *Even though we might imagine different padding strategies instead of always padding with zeros (for instance, nearest-neightbour padding, circular padding, random padding...), zero-padding seems to be both simple and effective in practice, and is the most commonly used strategy.*

Padding introduces an additional hyperparameter to the layer, P. Using padding, the width of the output of the layer becomes:

$$W' = W - K + 2P + 1 \tag{4.2.2}$$

A common way to set the value of P is to choose it such as the output have the same size as the input. This is achieved by taking $P = (K-1)/2$, called *same-padding*. This is only possible if K is odd, which is often the case in practice, since same-padding is often the expected behavior.

4.2.6 Receptive Fields

Definition (Receptive Field)**.** The *receptive field* of an output neuron is the set of neurons of the input of which the output neuron depends on.

By essence, Fully-connected layers have a trivial notion of receptive field: an output neuron is connected to each input neuron, its receptive field is therefore the entire input.

Convolution layers are build in such a way that each element in the output simply depends on a receptive field of size K (that is a square of area $K \times K$) in the input. As we stack convolutional layers after the others, each successive convolution adds $K-1$ to the receptive field size. After L layers, the receptive field size is $1 + L \times (K-1)$.

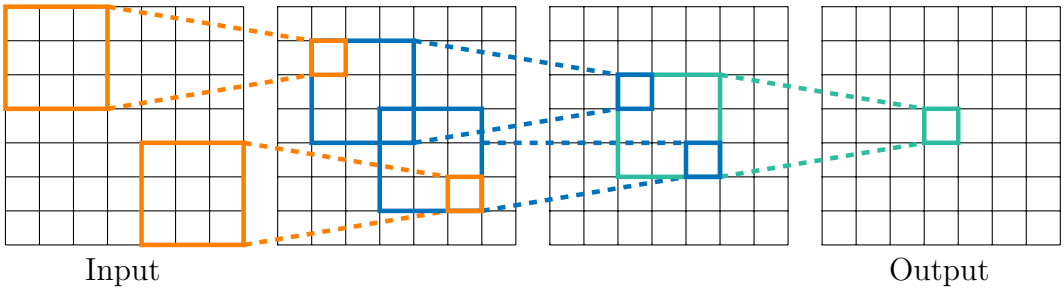

Figure 4.10: Receptive field of an output neuron

This linear growth shows that by stacking enough layers, each output neuron will eventually have the entire input image in its receptive field. Nevertheless, this can be a problem in practice as we might need many layers for each output to depend on the whole image.

A solution to this problem is to downsample the image size inside the network. This can be done by adding another hyperparameter, *stride*.

4.2.7 Strided Convolution

Definition (Stride)**.** The hyperparameter *stride* defines the number of pixels between two applications of the kernel.

Stride effectively downsamples the size of the image. Applying a convolution between an image of width W and padding P with a kernel of size K and stride S produces the following output dimension:

$$W' = \frac{W - K + 2P}{S} + 1 \tag{4.2.3}$$

Note that choosing $S = 1$ in (4.2.3) gives the same result as (4.2.2). Depending on the implementation, the result can be rounded up or down in the case where it is not an integer. Usually, all the parameters are chosen such that S divides $W - K + 2P$.

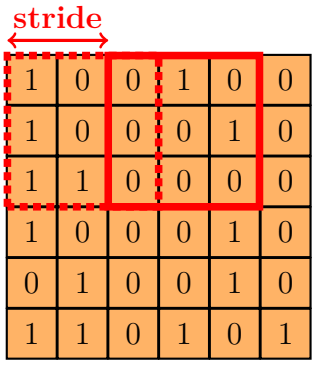

Figure 4.11: Effect of stride

4.3 Pooling Layers

4.3.1 Introduction

Computer Vision, one of the most frequent use for CNNs, often deals with images of high quality, making downsampling an important task to drastically reduce the number of layers and the quantity of VRAM used by the model. We saw a first approach to downsampling embedded in Convolutional Layers, that is strided convolution. Pooling Layers are layers dedicated to downsampling, without learnable parameters.

Pooling layers work similarly to convolutional layers, using a mechanism of kernels. Nevertheless, instead of applying a convolution between some kernel and the image, the layer will apply a pooling function to the area of the input. This will produce an activation map with dimensions depending on the hyperparameters of the layer – kernel size, padding and stride, the same as the parameters of a convolutional layer.

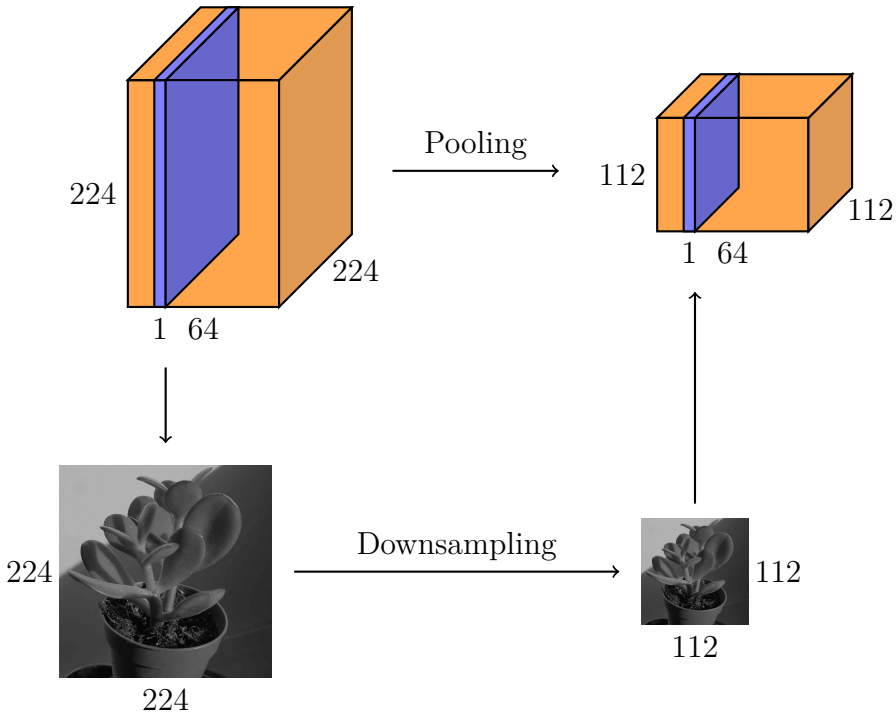

Figure 4.12: Behavior of a pooling layer

Another difference is that the operation is applied for each slice of the input volume. Choosing appropriate values for the hyperparameters (for instance $S \geqslant 2, \dots$) allow to downsample the input without the need for learnable parameters.

4.3.2 Max Pooling

Definition (Max Pooling function)**.** For $K \geqslant 1$, the *Max Pooling function* of kernel size K is:

$$\begin{aligned} \max_P : \mathscr{M}_K(\mathbb{R}) &\longrightarrow \mathbb{R} \\ (m_{i,j}) &\longmapsto \max_{i,j} m_{i,j} \end{aligned}$$

A *Max Pooling* layer applies the Max Pooling function to each location of size $K \times K$ in each slice of the input volume. We often choose the same kernel size as the slide (that is $K = S$) to avoid recovering twice the same pixel value. In this setting, it is equivalent to spliting each input channel into non-overlapping regions of size $K \times K$, from which are extracted the maximum value of the section and stored in the output channel.

Max Pooling has some advantages over convolutional layers with stride: it does not involve learnable parameters, reducing the computational cost, but also introduces translational invariance to small spatial shifts. Indeed, if the position of a specific maximum pixel is moved slightly, we might intuitively think that it will stay the maximum of its region, making the model robust to small translations.

4.3.3 Average Pooling

Definition (Average Pooling function)**.** For $K \geqslant 1$, the *Average Pooling function* of kernel size K is:

$$\begin{aligned} \operatorname{avg}_P : \mathscr{M}_K(\mathbb{R}) &\longrightarrow \mathbb{R} \\ (m_{i,j}) &\longmapsto \frac{1}{K^2} \sum_{i=1}^{K} \sum_{j=1}^{K} m_{i,j} \end{aligned}$$

It simply returns the average of the matrix coefficients.

Average Pooling works exactly the same as Max Pooling, but applying avg_P as a pooling function instead of $\max_P$.

4.4 A full CNN example: LeNet-5

Now that we have these different types of layers, we can stack them together to create a full CNN architecture. A classic model often fits the following architecture:

$$(\texttt{Conv, ReLU, Pooling})^{N_1} \rightarrow \texttt{Flatten} \rightarrow (\texttt{Linear, ReLU})^{N_2} \rightarrow \texttt{Linear}$$

As an example, we will take the 1998 model *LeNet-5*:

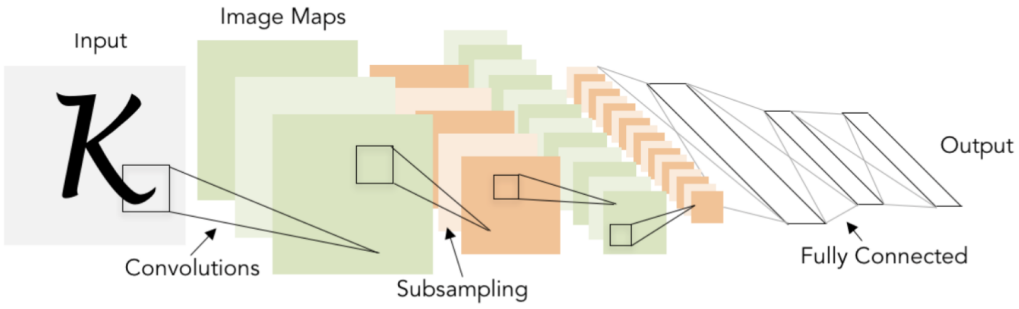

Figure 4.13: LeNet-5 architecture

The first blocks of Convolutional and Pooling layers progressively decrease the spatial size of the input, while increasing the number of channels: the total volume of the input is preserved.

4.5 Normalization

Deep convolutional neural networks are described previously can be extremly effective when trained; nevertheless, they are also very difficult to train, and we need to properly prepare data for the descent to converge.

A recent solution to ease the training is to add *normalization layers* in the network.

4.5.1 Batch Normalization

The idea of Batch Normalization is to normalize the outputs of a layer, often by giving the output a zero mean and unit variance. This improves optimization by guaranteeing that a layer always receive similar data from the previous layer's output. Indeed, while training, the distribution of the output of a layer might change, which tends to increase the complexity of the learning process. In contrast, the output of a batch normalization layers always has the same mean and variance; therefore, the next layer can always "see" the same input data distribution, improving convergence.

Normalizing batches of activations can simply be implemented using the following formula:

$$\hat{x}^{(k)} := \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathbb{V}[x^{(k)}]}}$$

Hopefully, this is a differentiable function, and can therefore be seen as any other layer in the network, and implement its backward gradient.

4.5.2 Batch Normalization in practice

Consider a fully-connected setup, in which we have a batch x consisting of N inputs, each of size D. We will use the N batch samples to compute the empirical mean for each of the element of the D-shaped vector:

$$\mu_j := \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

which gives us a vector μ of size D. Similarly, the standard deviation of the batch can be computed element-wise:

$$\sigma_j^2 := \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

giving us a vector σ of size D as well. Finally, each element of the batch x is normalized, giving us $\hat{x}$:

$$\hat{x}_{i,j} := \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

The small constant ε is added at the denominator to avoid diving by zero when the standard deviation is null.

In practice, zero mean and unit variance can be too hard of a constraint on the network. We prefer to add learnable scale and shift parameters γ and β of size D, allowing the network to choose what mean and variance it wants for the next layer. The acutal output is therefore y, defined by:

$$y_{i,j} := \gamma_j \hat{x}_{i,j} + \beta_j$$

Note that learning $\gamma = \sigma$ and $\beta = \mu$ recovers the identity function, making the network able to bypass the normalization layer if it is not needed.

An important note is that batch normalization introduces dependence on the minibatches: previously, each image in the input batch was handled independently of the others, which is not the case anymore with batch normalization. We cannot do this at test time, since it would not guarantee the reproducibility of classification, and more importantly, it would be a security breach to make each classification depend on the other inputs. To avoid this, batch normalization layers behave differently during training and testing. During training, the layer normalizes the batches as described before, while maintaining the empirical mean and variance of the inputs, μ and σ. These empirical vectors learned during training are then treated as constants and used during testing to normalize the test inputs.

Hence, at test time, the normalization layer performs the following operation:

$$y = \gamma \cdot \frac{x - \mu}{\sigma} + \beta$$

Another interesting property of this method is that during testing, the batch normalization layer becomes a linear operator. Therefore, it can be fused with the previous fully-connected or convolutional layer. For instance, if the previous layer is a fully-connected, its weight matrix and bias vector can be modified using the four parameters of the normalization layer (learned mean and variance, and empirical mean and variance). Performing this operation guarantees that the batch normalization layer does not add any inference time.

4.5.3 Batch Normalization for Convolutional Networks

For fully-connected layers, the batch dimension is:

$$x : N \times D$$

Therefore, the batch normalization parameters are the following:

$$\begin{cases} \mu, \sigma : 1 \times D & \text{(empirical mean and standard deviation)} \\ \gamma, \beta : 1 \times D & \text{(learned mean and standard deviation)} \end{cases}$$

This needs to be adapted for convolutional layers, but remains very similar. The batch dimension is:

$$x : N \times C \times H \times W$$

Instead of averaging only over the batch elements, we will also average over both spacial dimensions. This means that we will take for each channel the average and standard deviation taking into account all the pixels of all the images in the batch. Therefore, the batch normalization parameters are the following:

$$\begin{cases} \mu, \sigma : 1 \times C \times 1 \times 1 & \text{(empirical mean and standard deviation)} \\ \gamma, \beta : 1 \times C \times 1 \times 1 & \text{(learned mean and standard deviation)} \end{cases}$$

which can also be seen as vectors of size C; it is nevertheless more convenient to see them in tensor form, $1 \times C \times 1 \times 1$.

4.5.4 Advantages and downsides of batch normalization

As stated previously, adding batch normalization in neural networks allows for models to train much faster, while adding no overhead at testing time when placed after fully-connected or convolutional layers.

Not only the models converge quicker at fixed learning rate, but it also stabilizes the model even with higher learning rates, allowing shorter training times without the risk of divergence.

While they are widely used in practice, normalization layers also come with downsides: the reason of the effectiveness of batch normalization is not well-understood theoretically, and it introduces complex code because of the distinction between training and testing. This is actually a very common source of bugs in projects: one have to remember to change the mode from training to testing.

Finally, batch normalization is not always appropriate: some very unbalanced data sets might not fit appropriately with batch normalization, and this can reduce the performance of the model. This depends highly on the application: for computer vision, batch normalization is most of the time suitable.

4.5.5 Layer and Instance normalizations

A variant to batch normalization, called *layer normalization*, has be proposed. It aims at unifying the behavior of the normalization layer at training and testing time. This guarantees, even at training time, the independence between elements of one batch. The idea is to normalize the input in the layer direction instead of the batch direction. For a fully-connected layer with batch dimension:

$$x : N \times D$$

the layer normalization parameters become:

$$\begin{cases} \mu, \sigma : N \times 1 & \text{(empirical mean and standard deviation)} \\ \gamma, \beta : \ 1 \times D & \text{(learned mean and standard deviation)} \end{cases}$$

This kind of normalization is currently used in recurrent neural networks (RNNs) and transformers.

The equivalent of layer normalization for convolutional layers is called *instance normalization*: similarly, instead of averaging over both the batch and spatial dimensions, we will only normalize over the spatial dimensions. For a convolution layer with batch dimension:

$$x : N \times C \times H \times W$$

the layer normalization parameters become:

$$\begin{cases} \mu, \sigma : N \times C \times 1 \times 1 & \text{(empirical mean and standard deviation)} \\ \gamma, \beta : \ 1 \times C \times 1 \times 1 & \text{(learned mean and standard deviation)} \end{cases}$$

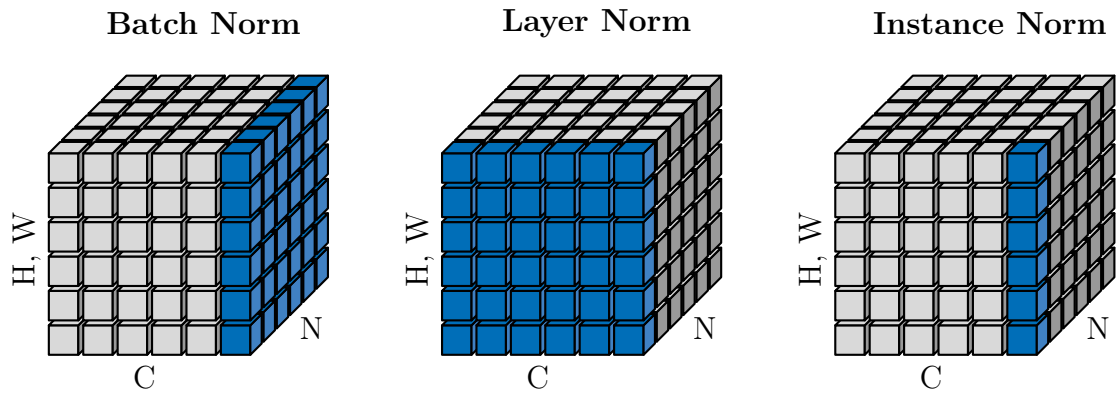

Figure 4.14: Visualization of the different types of normalization. The blue squares represent the parts normalized together.

5 Recurrent Neural Networks

6 Attention and Transformers

7 Robustness and Regularity

8 Deep Reinforcement Learning

8.1 What is Reinforcement Learning?

Machine learning problems can be distinguished in three major paradigms. Given a set of inputs and corresponding outputs, *supervised learning* tries to learn the function mapping inputs to outputs. *Unsupervised learning* studies the structure of data, without being given labels. The third paradigm is called *reinforcement learning*: such problems aim at optimizing the actions of an agent in an environment to maximize its reward.

The actions of the agent impact the data collected and the state of the environment. Reinforcement learning is often applied to video games (Atari, Starcraft, ...), to maneuver robots, or to learn how to control complex systems such as cooling systems in datacenters.

Formally, we control an agent that can observe at each time step t the *state* of the environment, S_t. It can use this state to choose an action A_t, to which the environment will reply with a

reward R_t, and a new state S_{t+1}.

$$S_t \longrightarrow A_t \longrightarrow R_t \Longrightarrow S_{t+1} \longrightarrow A_{t+1} \longrightarrow \dots$$

This is called one *episode* of learning.

8.2 Markov Decision Process

8.2.1 Formalisation

The *Markov Decision Process* (MDP) is a mathematical formulation of the Reinforcement Learning process. It makes the assumption of the *Markov property*: the current state completely caracterizes the state of the world.

Definition (Markov Decision Process)**.** A Markov Decision Process is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$ where:

- $\mathcal{S}$ is the set of all states, the *state space*
- $\mathcal{A}$ is the set of possible actions, the *action space*
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a function mapping a (state, action) pair to an immediate reward. Note that the reward might be random; therefore, we have:

$$\mathcal{R}(a, s) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

- $\mathbb{P}$ is a probability distribution; for $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$, $\mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$ is the probability to transition from state s to state s' after choosing the action a.
- $\gamma \in [0, 1]$ is a *discount factor*, quantifying how much we value rewards coming soon conversely to rewards coming later. $\gamma = 1$ values equally all future rewards, while $\gamma = 0$ means that we only care about the next reward.

Initially (at time step $t = 0$), an initial state S_0 is sampled. Then for any $t \in [\![0, T]\!]$[1] the following process is iterated:

- The agent chooses an action A_t
- The environment computes the associated reward $R_t = \mathcal{R}(S_t, A_t)$
- The environment samples the next step $S_{t+1} \sim \mathbb{P}(\cdot | S_t, A_t)$
- The agent receives the reward R_t and the next step S_{t+1}
- ...

Definition (Discounted return)**.** The *discounted return* (also known as *cumulative discounted reward*) is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-a} R_k$$

The objective is therefore to maximize the expected return $\mathbb{E}[G_t]$, where the expectation is taken over the sampled states $(S_k)_{k>t}$ and rewards $(R_k)_{k>t}$.

Definition (Policy)**.** A *policy* $\pi : \mathcal{S} \to \mathcal{A}$ is a function that specifies which action to take in each state. The agent can choose a stochastic strategy, encoded by a probability distribution[2]:

$$\pi(a) = \mathbb{P}(A_t = a | S_t = s)$$

[1] We might have $T = +\infty$

[2] More precisely, we consider here only *stationary policies*, that is policies that only depends on the current state (and not the previous states and rewards, for instance). This is because there always exists an optimal policy that only depends on the current state; including information about previous states, actions or rewards does not provide better policies.

In this case, we maximize the expected return:

$$\max_{\pi} \mathbb{E}_{\pi}[G_t]$$

where $\mathbb{E}_{\pi}$ denotes the expectation under the use of policy π by the agent.

8.2.2 Example: Gridworld

Let's analyze a simple example of Reinforcement Learning problem. Gridworld is a task in which the states are the positions in the grid, and the actions are the movements in all 4 directions. The goal is two reach one of the terminal states, in the least number of actions.

Formally, we can define a Markov decision process by setting:

$$\mathcal{S} = \{\, (i,j) \mid 1 \leqslant i,j \leqslant 5 \,\} \qquad \mathcal{A} = \{\leftarrow, \rightarrow, \uparrow, \downarrow\} \qquad \mathcal{R} = s \longmapsto -1$$

8.2.3 Value function and Q-function

Note that the discounted return G_t is a random variable, since the rewards $(R_k)_k$ depend on the sampled states $(S_k)_k$. Therefore, we introduce the following two deterministic functions.

Definition (Value function)**.** Given a policy π, the *value function* v_π is defined by:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right]$$

where the expectation is taken over the sampled states $(S_k)_{k>t}$ and in which the successive actions A_k are picked using the policy: $A_k = \pi(S_k)$.

Definition (Q-function)**.** Given a policy π, the *Q-function* (also known as *action-value function*) q_π is defined by:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right]$$

where the expectation is taken over the sampled states $(s_k)_{k>t}$. The difference with the value function is that we assume that the action a is taken, making it in a sense "one episode after".

8.2.4 Optimal policy

The optimization problem associated with the Markov Decision Process is to select the best policy, that is the policy which maximizes the expected discounted return:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G_t] \tag{8.2.1}$$

where the expectation is taken over the sampled states $(S_k)_{k>t}$ and in which the successive actions A_k are picked using the policy: $A_k = \pi(S_k)$.

Definition (Optimal action-value function)**.** The optimal action-value function q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$q^*(s,a) = \max_{\pi} \mathbb{E} \left[G_t | S_t = s, A_t = a \right]$$

Theorem (Bellman's principle of optimality – 1952)**.** An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Property 8.1 (Bellman's equation)**.** Intuitively, if the optimal state-action values for the next time-step $q^*(s', a)$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma \cdot q^*(s, a)$. Formally, this gives us:

$$q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \cdot \max_{a'} q^*(s', a') | s, a \right] \tag{8.2.2}$$

Therefore, the optimal policy takes in each state the action maximizing $q^*(s, a)$.

8.2.5 Value iteration algorithm

We can derive an iterative update from Bellman's equation:

$$q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \cdot \max_{a'} q_i(s', a') | s, a \right]$$

At each step, we refine our approximation of q^* by following Bellman's equation. Under mathematical conditions, we will then have:

$$\lim_{i \to +\infty} q_i = q^*$$

Unfortunately, this idea is not scalable: it requires the computation of $q(s, a)$ for every (state, action) pair, even though the state space can be huge. For instance, if we try to apply this reinforcement learning approach to a video game, we need to compute the result for any combination of pixels on the screen.

Therefore, we use in practice an approximator of q instead of computing the exact value of q: this is called Q-learning.

9 Autoencoders

10 Generative Adversarial Networks

11 Normalizing Flows