Deep Learning

Marc Lelarge, Jill-Jenn Vie, and Kevin Scaman

Class notes by Antoine Groudiev

Last modified 17th 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.

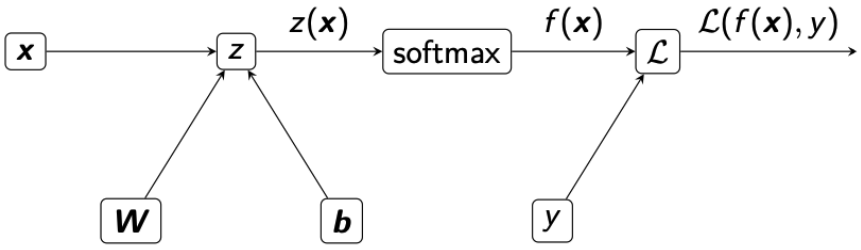


Figure 2.2: Computational graph for $\mathcal{L}(\text{softmax}(Wx + b), y)$

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:

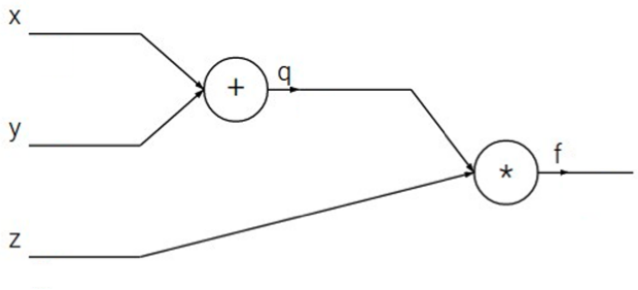


Figure 2.3: Computational graph for $f(x, y, z) = (x + y) \times z$

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:

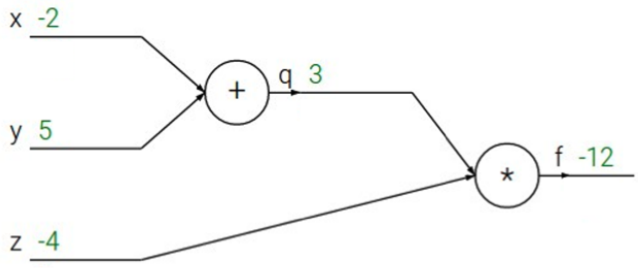


Figure 2.4: Forward pass

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.

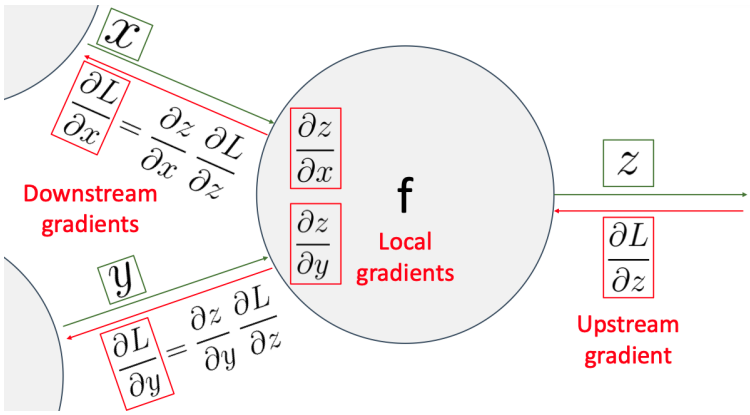


Figure 2.5: Local process of computing the downstream gradient

2.4.5 A complete example

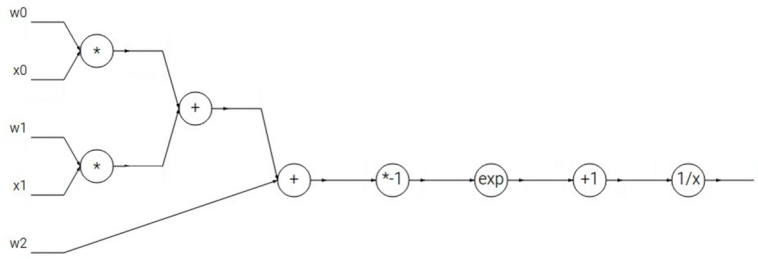


Figure 2.6: Computational graph for the function $f(x, w) = \frac{1}{1+e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$

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:

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

MSE has a probabilistic interpretation, fitting the following probabilistic model: we are trying to learn a certain function g_θ, parametered by $\theta \in \mathbb{R}^p$. To do so, we are given tuples of data of the form (X_i, Y_i), where Y_i is the label corresponding to X_i. We assume that the probabilistic relationship between X_i and Y_i is the following:

$$Y_i = g_\theta(X_i) + \varepsilon_i$$

where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2 I_d)$ are i.i.d. centered Gaussian random variables (that is of mean 0 and variance σ^2). We also assume that the X_i are i.i.d. and independent of θ.

Let's apply the Maximum Likelihood principle to this probabilistic model. The likelihood for the data to be drawn from a given θ is:

$$\begin{aligned}
\mathbb{P}_\theta((X_i, Y_i)_i) &= \prod_i \mathbb{P}(X_i) \cdot \mathbb{P}_\theta\left(\varepsilon_i = Y_i - g_\theta(X_i)\right) \\
&= \prod_i \mathbb{P}(X_i) \cdot \frac{1}{\sqrt{(2\pi)^d |\sigma^2 I_d|}} \cdot \exp\left(-\frac{1}{2}(Y_i - g_\theta(X_i) - 0)^\top (\sigma^2 I_d)^{-1} (Y_i - g_\theta(X_i) - 0)\right) \\
&= \prod_i \frac{\mathbb{P}(X_i)}{\sqrt{(2\pi\sigma^2)^d}} \cdot \exp\left(-\frac{1}{2\sigma^2}\|Y_i - g_\theta(X_i)\|_2^2\right) \\
&\propto \exp\left(-\frac{\sum_i \|Y_i - g_\theta(X_i)\|_2^2}{2\sigma^2}\right) \\
&= \exp\left(-\frac{\ell(Y_i, g_\theta(X_i))}{2\sigma^2}\right)
\end{aligned}$$

Therefore, maximizing the log-likelihood is equivalent to minimizing the MSE.

In `PyTorch`, means square error can be used with the line `criterion = nn.MSELoss()`. It supports several parameters, such as `reduction='sum'`, `reduction='mean'`, which are described in the documentation. Once set, `criterion` takes as input two tensors of shapes $[N, d]$, where N is the batch size and d is the dimension of the output vectors. Note that when $d = 1$, the output should be of size $[N, 1]$ and not $[N]$; in that case, `x.unsqueeze(-1)` can come in handy.

3.2.2 Cross Entropy

Definition (Cross Entropy)**.** The *cross entropy* is the loss function defined by:

$$\begin{aligned}
\ell : \mathbb{R}^C \times [\![1, C]\!] &\longrightarrow \mathbb{R} \\
x, y &\longmapsto -\log\left(\frac{\exp(x_y)}{\sum_i \exp(x_i)}\right)
\end{aligned}$$

Similarly to MSE, minimizing the cross entropy corresponds to maximizing the log-likelihood for a certain probabilistic model. Assume that there is a parameter $\theta \in \mathbb{R}^p$ such that, for all classes $k \in [\![1, C]\!]$,

$$\log \mathbb{P}(Y_i = k | X_i) \propto g_\theta(X_i)_k$$

where X_i are i.i.d. and independent of θ. Then, the likelihood for the data to be drawn for a given θ is:

$$\begin{aligned}
\mathbb{P}_\theta((X_i, Y_i)_i) &= \prod_i \mathbb{P}(X_i) \cdot \mathbb{P}_\theta(Y_i | X_i) \\
&\propto \prod_i \frac{\exp(g_\theta(X_i)_{Y_i})}{\sum_k \exp(g_\theta(X_i)_k)}
\end{aligned}$$

In `PyTorch`, cross entropy can be used with the line `criterion = nn.CrossEntropyLoss()`. It supports several parameters, such as `reduction='sum'`, `reduction='mean'`, which are described in the documentation. Once set, `criterion` takes as input two tensors: the *scores* (a tensor of shape $[N, C]$), and either a class index per sample, or class probabilities for each sample. Note that `nn.CrossEntropyLoss` is the composition of `nn.LogSoftmax` and `nn.NLLLoss`.

Finally, be aware that gradient can explode when going through a softmax, due to numerical errors. This is however taken care of by the `PyTorch` implementation of `nn.CrossEntropyLoss` and `nn.LogSoftmax`.

3.3 First-order optimization

3.3.1 Introduction

Given an objective function $\mathcal{L} : \mathbb{R}^d \longrightarrow \mathbb{R}$, the goal of optimization is to find a minimized $\theta^* \in \mathbb{R}^d$ of $\mathcal{L}$, that is:

$$\theta^* \in \arg\min_{\theta \in \mathbb{R}^d} \mathcal{L}(\theta)$$

A standard approach to doing so is to use an iterative algorithm relying on the gradient $\nabla\mathcal{L}(\theta_t)$ at each iteration $t \geqslant 0$.

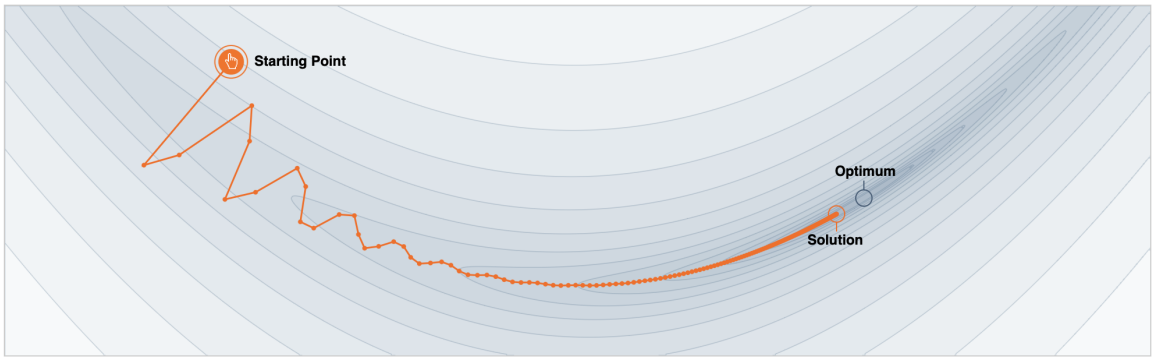


Figure 3.1: Simulation of gradient descent from `https://distill.pub/2017/momentum/`.

3.3.2 Gradient descent structure

Iterative optimization algorithms all follow the same structure:

Initialization We start with a random parameter $\theta_0 \in \mathbb{R}^p$. The choice of the initial value is very important in practice, as a good initialization will help our model to converge and reduce overfitting.

Iteration The current value of the parameter is updated using a rule of the form

$$\theta_{t+1} = \varphi_t(\theta_t, \nabla\mathcal{L}(\theta_t), s_t)$$

where s_t is a hidden variable that is also updated at each iteration. The update function, φ_t, might depend on the step t (for instance when using various step sizes). The most basic version of gradient descent is $\theta_{t+1} = \theta_t - \eta \cdot \nabla\mathcal{L}(\theta_t)$, where η is the learning rate (or step size).

Stopping time The algorithm stops after a certain number T of iterations. This is important in practice and T should be picked carefully to avoid overfitting.

3.3.3 Difficulties in neural network training

In practice, training deep neural network can become challenging because of numerous difficulties.

Non-convexity If $\mathcal{L}$ is convex, that is:

$$\forall \theta, \theta' \in \mathbb{R}^p, \quad \mathcal{L}\left(\frac{\theta + \theta'}{2}\right) \leqslant \frac{\mathcal{L}(\theta) + \mathcal{L}(\theta')}{2}$$

the optimization problem is fairly simple. Most theoretical results use this assumption in order to prove the convergence of the algorithm. In practice, this is often not the case, making the optimization problem very hard.

High dimensionality The number of parameters d can be huge: for instance, the VGG-16 network, a convolutional neural network trained on the ImageNet dataset, has $p = 138\,357\,544$ parameters.

Access to the gradient The gradient of $\mathcal{L}$ is often too expensive to compute on the entire dataset. In practice, $\nabla\mathcal{L}(\theta_t)$ is replaced by a stochastic or mini-batch approximation $\tilde{\nabla}_t$.

3.3.4 Gradient approximations

In the following, we will consider that the objective function $\mathcal{L}$ is a loss function between a prediction and a label for one specific example, that is:

$$\forall i \in [\![1, n]\!], \quad \mathcal{L}_i(\theta) = \ell(g_\theta(x_i), y_i)$$

Our objective function $\mathcal{L}$ will be the training error:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_i(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$$

In the following, $\eta > 0$ denotes a *learning rate* (or *step-size*). The following variants of gradient descent are often used:

Batch gradient descent It uses the true gradient:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{n}\sum_{i=1}^{n} \nabla\mathcal{L}_i(\theta)$$

This approach gives the best possible gradient, but is very expensive if the dataset is large (when n is big).

Stochastic gradient descent The gradient is approximated with one random sample

$$\theta_{t+1} = \theta_t - \eta \cdot \mathcal{L}_{i_t}(\theta)$$

This approach is very efficient but often instable, since one element might contain noise, making the convergence difficult.

Mini-batch gradient descent The gradient is approximated with multiple random samples:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{b}\sum_{i=1}^{b} \nabla\mathcal{L}_{i_{b,t}}(\theta)$$

where $b > 0$ is the *batch size*. This method is a good tradeoff between speed and convergence, with a parameter b that can be adjusted to obtain the best results.

Keep in mind that the final goal is to reduce the population risk, that is $\mathbb{E}[\ell(g_\theta(X), Y)]$. We need to pay attention to overfitting in addition to using the optimization algorithm to reduce the training error. In this class, we focus specifically on the performance of the optimization algorithm in minimizing the objective function, rather than the model's generalization error. In the next chapters, we will see techniques to avoid overfitting.

3.4 Convergence analysis

We aim at providing theoreticla guarantees over the convergence rates of our algorithms. Optimizing non-convex functions is hard in general, but we will make the following assumptions:

- The objective function is non-convex but differentiable and β-smooth, that is:

$$\forall \theta, \theta' \in \mathbb{R}^p, \quad \|\nabla \mathcal{L}(\theta) - \nabla \mathcal{L}(\theta')\|_2 \leqslant \ \beta \cdot \|\theta - \theta'\|_2$$

- We access unbiased noisy gradients $\tilde{\nabla}_t$, where:

$$\mathbb{E}[\tilde{\nabla}_t] = \nabla \mathcal{L}(\theta_t) \quad \text{and} \quad \mathbb{V}[\tilde{\nabla}_t] \leqslant \sigma^2$$

Property 3.1 (Worst-case convergence to global optimum)**.** For any first-order algorithm, there exists a smooth function $\mathcal{L}$ such that its approximation error is at least:

$$\mathcal{L}(\theta_t) - \mathcal{L}(\theta^*) = \Omega(t^{-\frac{1}{d}})$$

This is prohibitive for large dimensional spaces, since this makes the convergence extremly slow. Therefore, we often choose to restrict ourselves to guarantees for local optima.

Theorem (Convergence of non-convex SGD to a stationary point)**.** Let $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ be a smooth function and $\Delta = \mathcal{L}(\theta_0) - \mathcal{L}(\theta^*)$. Then, SGD with step-size:

$$\eta = \min\left(\frac{1}{\beta}, \sqrt{\frac{2\Delta}{T\beta\sigma^2}}\right)$$

achieves the error:

$$\mathbb{E}\left[\min_{t \leqslant T} \|\nabla \mathcal{L}(\theta_t)\|^2\right] \leqslant \frac{4\beta\Delta}{T} + \sqrt{\frac{8\beta\Delta\sigma^2}{T}}$$

Furthermore,

- without noise, $\eta = \frac{1}{\beta}$ is optimal, and require $O\left(\frac{\beta\Delta}{\varepsilon^2}\right)$ iterations for $\|\nabla \mathcal{L}(\theta_t)\| \leqslant \varepsilon$
- with noise, $\eta = O\left(T^{-\frac{1}{2}}\right)$ is optimal, and requires $O\left(\frac{\beta\Delta\sigma^2}{\varepsilon^4}\right)$ iterations for $\|\nabla \mathcal{L}(\theta_t)\| \leqslant \varepsilon$
- if η is fixed and $\sigma > 0$, there is a lower limit in $O\left(\sqrt{\eta\beta}\sigma\right)$ for $\|\nabla \mathcal{L}(\theta_t)\|$

These guarantees prove that GD and SGD converge to a stationary point. Nevertheless, we have no guarantee that this stationary point is actually a local minimum. A local minimum can be defined using second order derivatives: it is both a **stationary** point, that is $\nabla \mathcal{L}(\theta) = 0$, and a **convexity** point, where the Hessian $H_{\mathcal{L}}(x)$ is semi-definite positive. The problem is that stationary points can block our algorithm by cancelling the gradient: we are then stuck in saddle points.

An idea to converge towards a local minimum is to add a small noise, allowing the parameter to escape saddle points. When slightly pushed away from the equilibrium by the noise, the parameter will come back to it only is the equilibrium is stable, that is for local minima.

We make the additional assumption that the Hessian $H_{\mathcal{L}}$ is ρ-Lipschitz with respect to the spectral norm. Therefore, with probability at least $1-\delta$, the number of iterations to reach a gradient norm $\|\nabla\mathcal{L}(\theta_t)\| \leqslant \varepsilon$ and near-convexity $\lambda_1(H_{\mathcal{L}}(\theta_t)) \geqslant -\sqrt{\rho\varepsilon}$ is bounded by:

$$O\left(\frac{\beta\Delta}{\varepsilon^2}\cdot\log\left(\frac{p\beta\Delta}{\varepsilon\delta}\right)^4\right)$$

To conclude:

- The loss landscape of deep learning training is non-convex and potentially difficult to optimize
- SGD converges to a stationary point in $O(\varepsilon^{-4})$ iterations
- GD converges to a stationary point in $O(\varepsilon^{-2})$ iterations
- Adding small noise to GD make it converge to a local minimum in $O(\varepsilon^{-2}\log(\varepsilon^{-1})^4)$ iterations.
- Convergence to a global minimum is prohibitive in high-dimensional spaces. We need at least $\Omega(\varepsilon^{-d})$ iterations for smooth functions; we need additional assumptions to ensure fast rates, such as the Polyak-Losasiewicz condition.

3.5 Gradient descent variants

Mini-batch gradient descent is the algorithm of choice when training a neural network. The term SGD is usually employed even when mini-batches are used! But despite working very well on fundamental examples (such as MLPs), some important questions remain. For instance, choosing learning rates can be difficult: how can we adapt the learning rate during training? Do we apply the same learning rate to all parameter updates? How to espace saddle points when the gradient is close to zero in all dimensions? We will introduce modifications to SGD to address these issues.[1]

3.5.1 Momentum

The core idea of *momentum* is to accelerate SGD by dampening oscillations, that is averaging the last values of the latest gradients. We maintain a hidden state v_t which is updated at each step, and used to update the parameter:

$$\begin{aligned} v_{t+1} &= \gamma\cdot v_t + \eta\cdot\nabla\mathcal{L}(\theta_t)\\ \theta_{t+1} &= \theta_t - v_{t+1}\end{aligned}$$

where γ is a decay factor, with a typical value of $\gamma = 0.9$. Intuitively, v_t will store a sort of average of the previous gradients; note that:

$$\forall k\geqslant 0,\quad v_{t+1} = \gamma^k\cdot v_{t-k} + \eta\sum_{i=0}^{k}\gamma^i\nabla\mathcal{L}(\theta_{t-i})$$

If we choose $v_0 = 0$, then we have:

$$v_{t+1} = \eta\sum_{i=0}^{t}\gamma^{t-i}\nabla\mathcal{L}(\theta_i)$$

which corresponds to a weighted sum of the previous gradients, each with a factor decreasing as the number of iterations increases.

[1] For more information, you can read this nice survey by Sebastian Ruder of different gradient descent optimization algorithms.

3.5.2 Nesterov accelerated gradient

With momentum, we first compute the gradient and then make a step following our momentum and add the gradient. Nesterov proposed to first make the step following the momentum, and then adjusting by computing the gradient locally. Formally, this gives us:

$$v_{t+1} = \gamma \cdot v_t + \eta \cdot \nabla \mathcal{L}(\theta_t - \gamma \cdot v_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

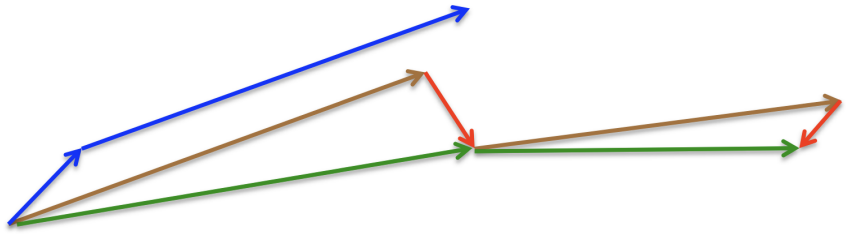


Figure 3.2: Nesterov method. Brown corresponds to the accumulated gradient jump $\gamma \cdot v_t$, red to the gradient correction $\eta \cdot \nabla \mathcal{L}(\theta_t - \gamma \cdot v_t)$, and blue to the standard momentum.

3.5.3 AdaGrad

We would like to adapt our updates to each individual parameter, i.e. have a different decreasing learning rate for each parameter. This is what AdaGrad (standing for Adaptative Gradient) does:

$$s_{t+1,i} = s_{t,i} + \nabla \mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i} + \varepsilon}} \cdot \nabla \mathcal{L}(\theta_t)_i$$

This provides a decreasing learning rate, allowing to go fast at the beginning and slower when approaching the minimum. This requires no manual tuning of the learning rate. Typical default values are $\eta = 0.01$ and $\varepsilon = 10^{-8}$.

3.5.4 RMSProp

The problem with AdaGrad is that the learning rate goes to zero, and never forgets about the past. An idea to avoid this is to use an exponential average instead:

$$s_{t+1,i} = \gamma \cdot s_{t,i} + (1 - \gamma) \cdot \nabla \mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i} + \varepsilon}} \cdot \nabla \mathcal{L}(\theta_t)_i$$

where γ is a decay factor (similarly to the idea of momentum), with default values $\gamma = 0.9$ and $\eta = 0.001$.

3.5.5 Adam

From the algorithm variations presented previously emerge two big ideas: using momentum to store the gradient, and having individual learning rates. Combining RMSProp and momentum gives us a widely used algorithm, Adam (for Adaptative Moment Estimation). It uses the

following equations:

$$
\begin{aligned}
m_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1) \cdot \nabla \mathcal{L}(\theta_t) \\
v_{t+1} &= \beta_2 \cdot v_t + (1-\beta_2) \cdot \nabla \mathcal{L}(\theta_t)^2 \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \varepsilon} \cdot \hat{m}_{t+1}
\end{aligned}
$$

While they seem complex, these are just the extension and combinaison of the algorithms detailled before. m_t and v_t are estimates for the first and second moments of the gradients, just like we used in momentum and RMSProp. Because $m_0 = v_0 = 0$, these estimates are biased towards 0, thus the factors $(1-\beta^{t+1})^{-1}$ are here to counteract these biases. Typical values are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

3.5.6 AMSGrad

Sometimes, Adam forgets too fast. To fix it, we replace the moving average by a max operator:

$$
\begin{aligned}
m_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1) \cdot \nabla \mathcal{L}(\theta_t) \\
v_{t+1} &= \beta_2 \cdot v_t + (1-\beta_2) \cdot \nabla \mathcal{L}(\theta_t)^2 \\
\hat{v}_{t+1} &= \max(\hat{v}_t, v_{t+1}) \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \varepsilon} \cdot m_{t+1}
\end{aligned}
$$

3.6 `PyTorch` optimizers

All these optimizers are built-in in `PyTorch`. They share the similar constructor:

```
torch.optim.*(params, lr=..., momentum=...)
```

Default values are different for all optimizers. In this construction, `params` should be an iterable (like a list) containing the parameters to optimize over. It can be obtained from any model with `module.parameters()`. The `setp` method updates the internal state of the optimizer acoording to the `grad` attributes of the `params`, and updates the latter according to the internal state.

A simple training loop is of the form:

```
criterion = nn.NLLLoss()
optimizer = torch.optim.SGD(model.parameters(),lr = 0.001)
model.train()
for epoch in range(num_epochs):
    for inputs, targets in dataloader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
```

Figure 3.3: A simple `PyTorch` training loop.

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)`.

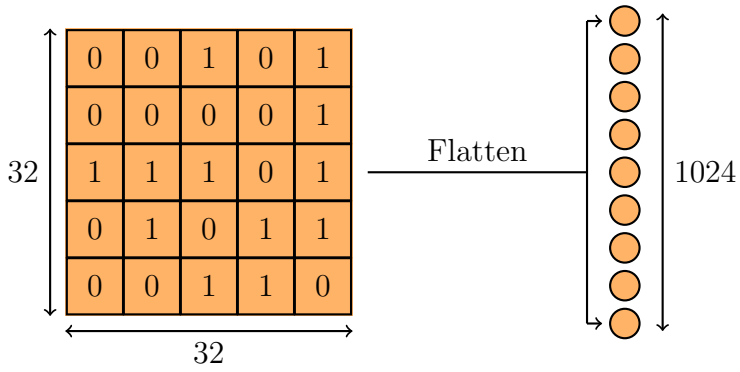


Figure 4.1: Flatten layer breaking the spatial strucutre of input data

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:

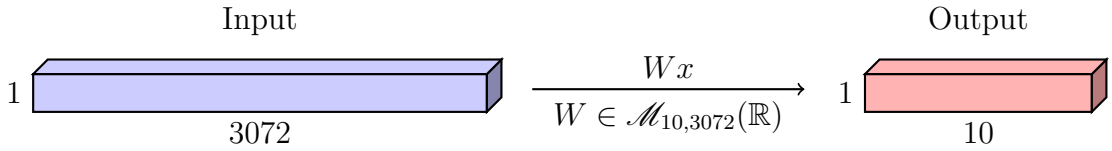


Figure 4.2: Fully-Connected Layer

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.

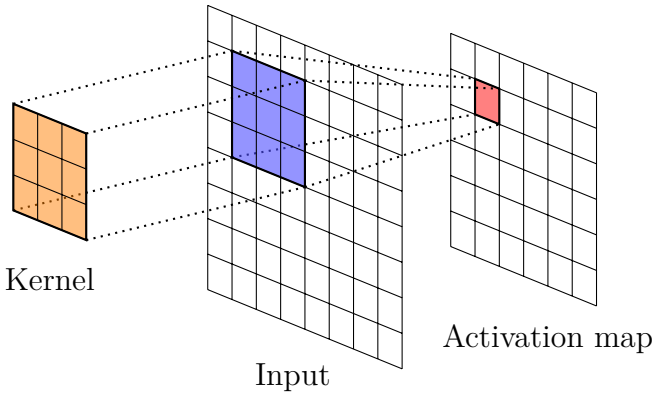


Figure 4.3: Kernel convolution

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.

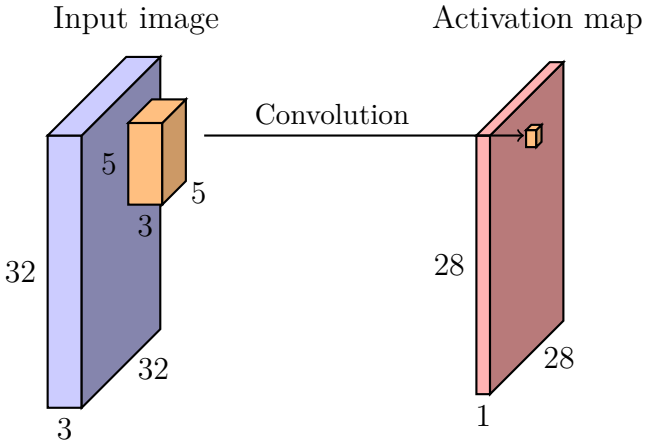


Figure 4.4: Input and output of the convolution operation

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

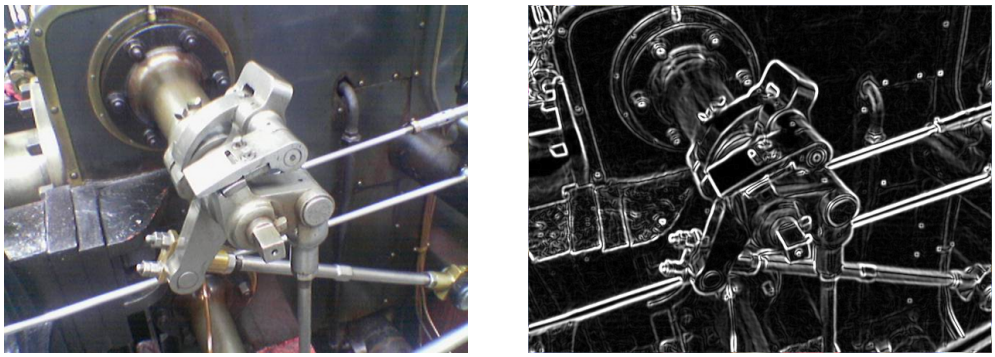


Figure 4.5: Effect of the Sobel operator on an image

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.

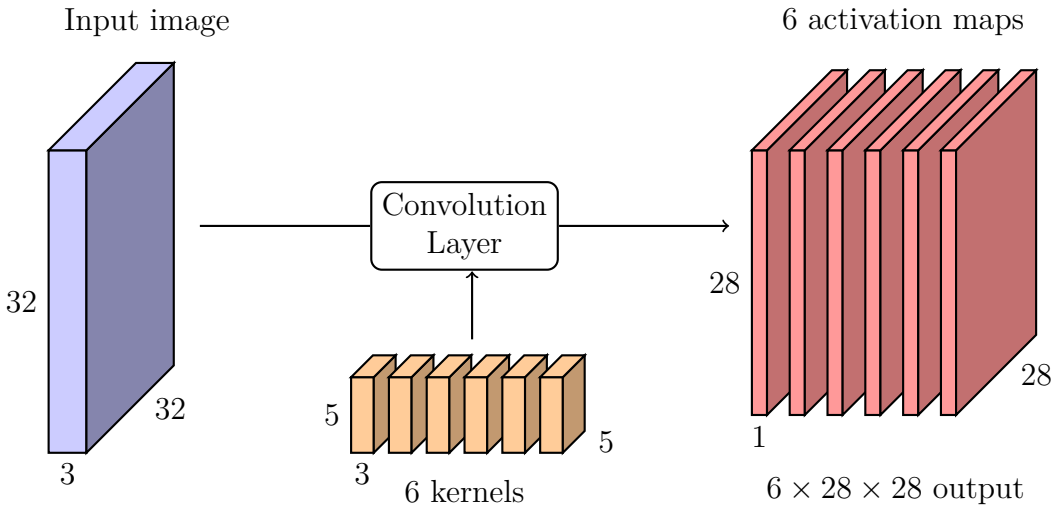


Figure 4.6: Convolutional Layer using 6 kernels

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.

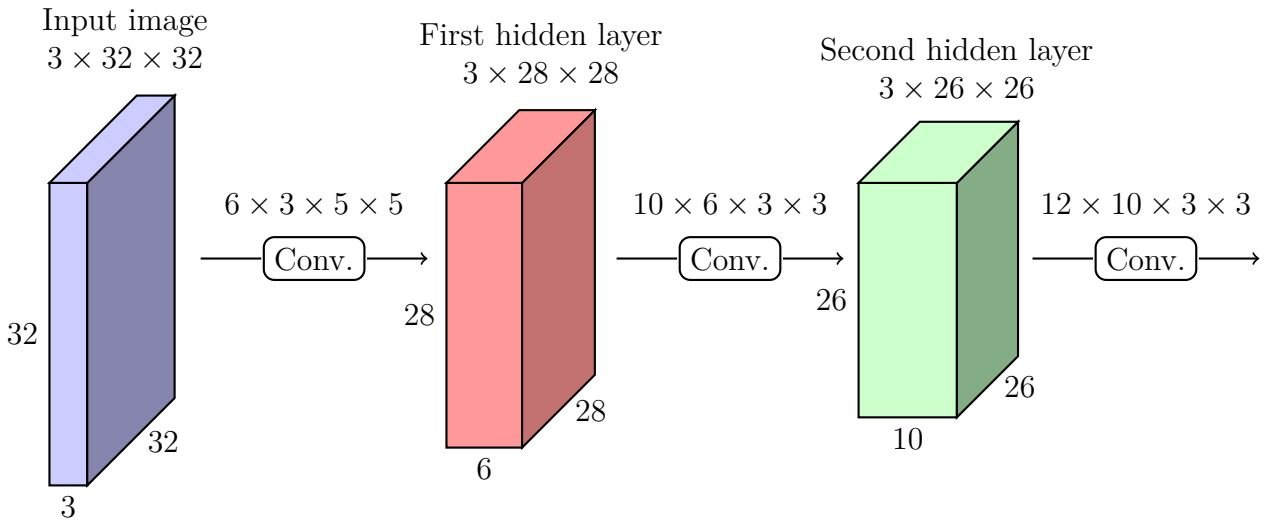


Figure 4.7: Stacking of 3 Convolutional Layers of correct shapes

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.

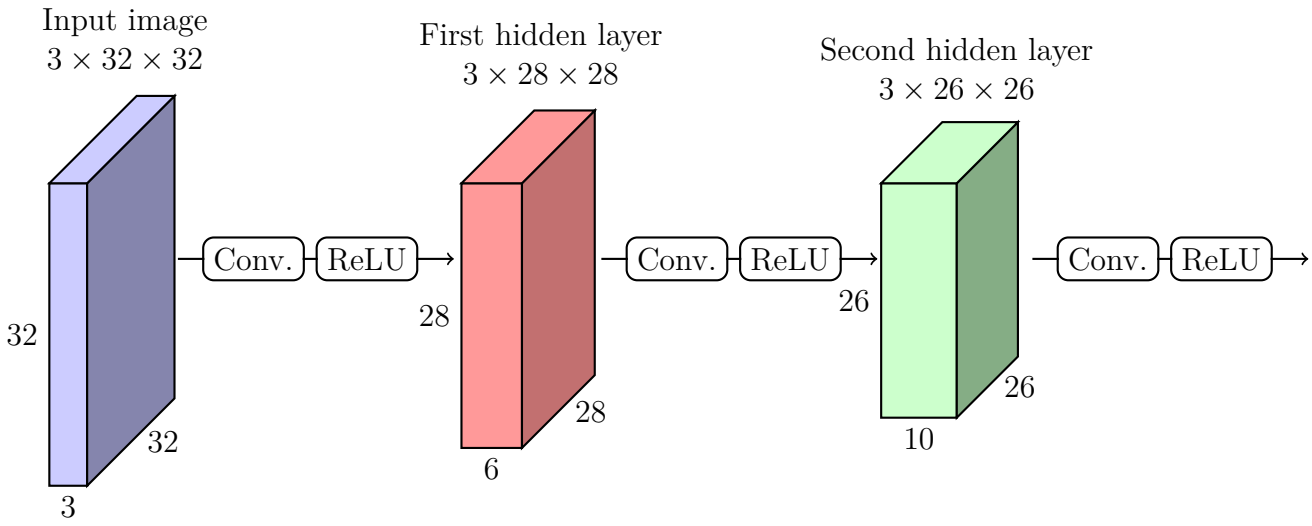


Figure 4.8: Adding ReLU layers in between Convolution Layers

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.

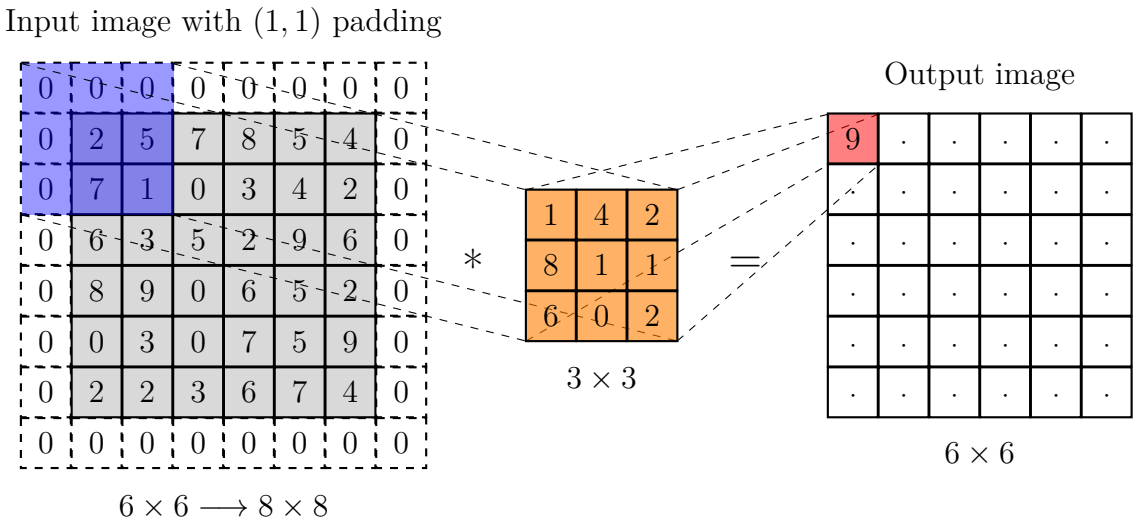


Figure 4.9: Adding padding around the input

Remark (Padding strategies)**.** *Even though we might imagine different padding strategies instead of always padding with zeros (for instance, nearest-neighbour 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 $1 + L \times (K-1)$.

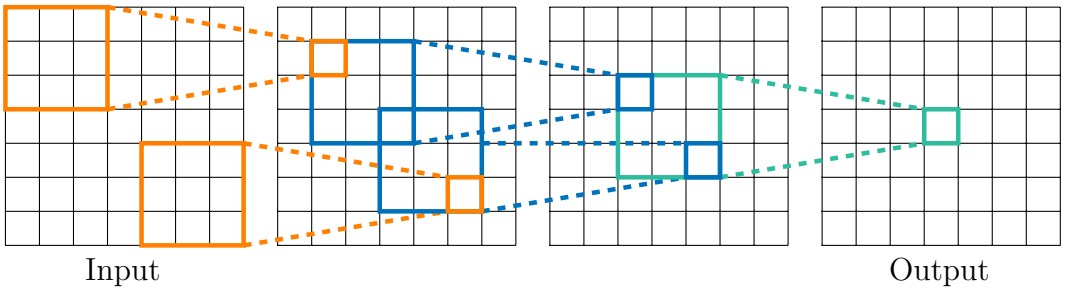


Figure 4.10: Receptive field of an output neuron

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

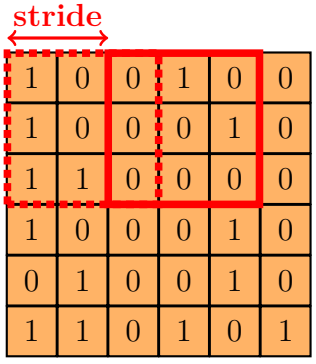


Figure 4.11: Effect of stride

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 $W - K + 2P$.

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.

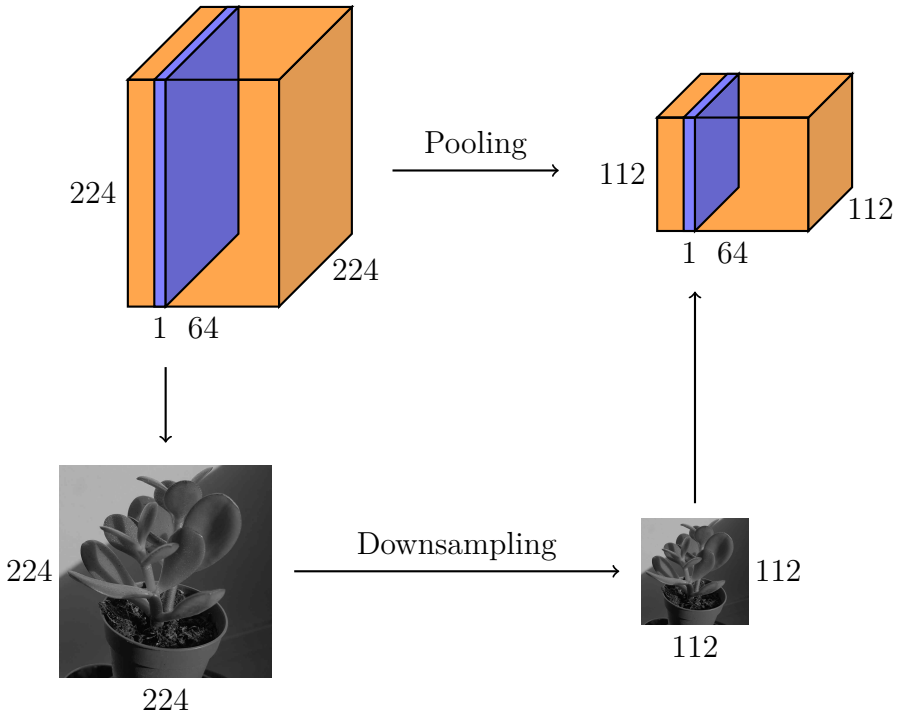


Figure 4.12: Behavior of a pooling layer

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:

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

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:

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

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

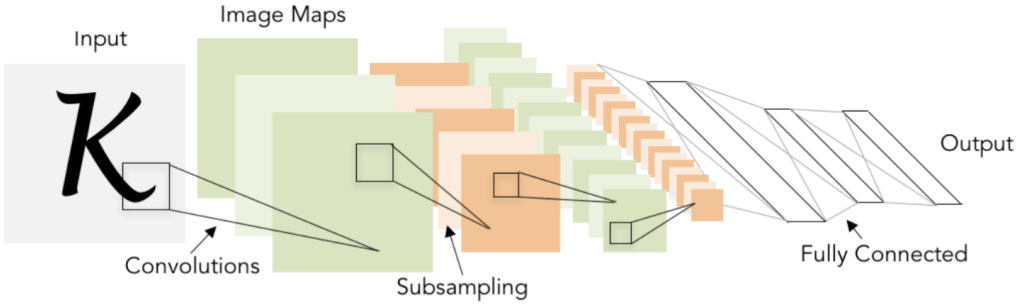


Figure 4.13: LeNet-5 architecture

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:

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

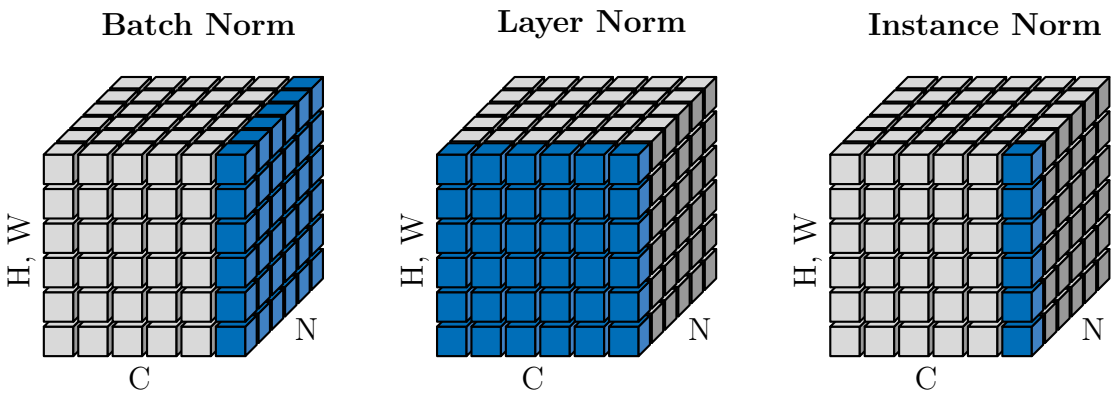


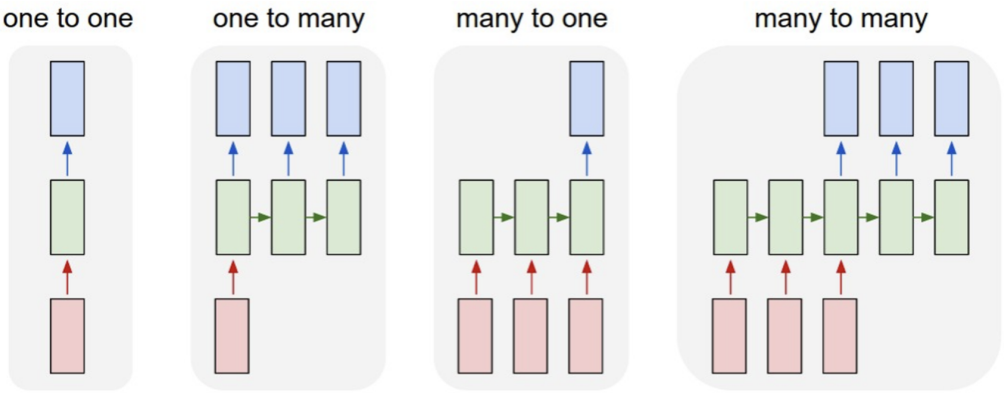
Figure 4.14: Visualization of the different types of normalization. The blue squares represent the parts normalized together.

5 Recurrent Neural Networks

5.1 Processing Sequences

So far, all the deep neural networks that we studied (MLPs, CNNs) had the same *feed-forward* global structure: the network receives an input of fixed size, which is fed into the successive layers of the network, which produces a fixed-size output. This is known as a *one-to-one* model, and is mostly used for image classification.

When considering other applications, we would like more flexibility on the input and output sizes. For image captioning, we would like a *one-to-many* model, where the input is a fixed size image, and the output a sentence of variable length. For text classification, such as sentiment analysis, we would like a *many-to-one* model, where the input is a sentence of variable length and the output a fixed-size vector of class probabilities. Finally, we might want both the input and output to be variable in length, for instance in the case of machine translation: this requires a *many-to-many* model.



Recurrent Neural Networks (RNNs) is a general paradigm which allows to handle all these different setups.

5.2 Simple Recurrent Neural Network

5.2.1 General form

In its simplest form, a Recurrent Neural Network posesses an internal hidden state which is updated each time that it reads an input. The handling of a sequential input (for instance, a sentence) follows this update loop: a fixed-size bit of the input is fed into the network, and is combined with the internal hidden state to produce an output and update the state. The next bit of the input is then fed to the network, and so on.

Formally, this can be written as a recurrence relationship:

$$h_t = f_W(h_{t-1}, x_t)$$

where x is a sequence of vectors, $(h_t)_t$ is the sequence of hidden states, and f_W is our network depending on some parameter W. To produce an output at each time step, we can simply transform the hidden state into an ouput using a feed-forward neural network:

$$y_t = g_{W'}(h_t)$$

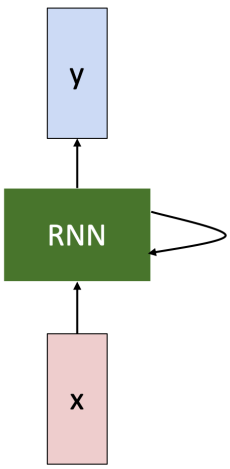


Figure 5.1: A simple RNN.

Note that the same functions f, g and the same parameters W, W' are used at each step, only the internal state changes.

5.2.2 A vanilla RNN

We can create a simple RNN built around a single hidden vector h_t:

$$h_t = \tanh\left(W_h h_{t-1} + W_x x_t\right)$$
$$y_t = W_y h_t$$

5.2.3 Computational graphs of RNNs

Alongside with this intuition of RNNs as networks with hidden cells, it is also useful to think of RNNs as computational graphs. The graph of an RNN can be unfolded over multiple time steps, expliciting the inputs and gradient flow.

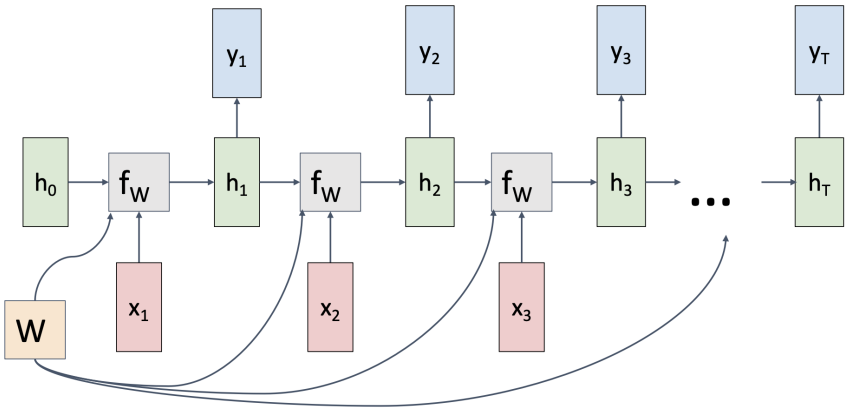


Figure 5.2: Unfolded computational graph of an RNN.

The initial hidden state h_0 is often initialized to 0 in most contexts. It is then fed to the f_W function with the first input x_1, which produces the new hidden state h_1. This hidden state can be used to compute the output y_1.

Note that the parameter W remains the same for all; when computing the gradient $\frac{\partial \mathcal{L}}{\partial W}$, we will sum all the graidents coming from each time step to compute the total gradient.

This unfolded view also allows to understand the relationship with the loss more explicitely.

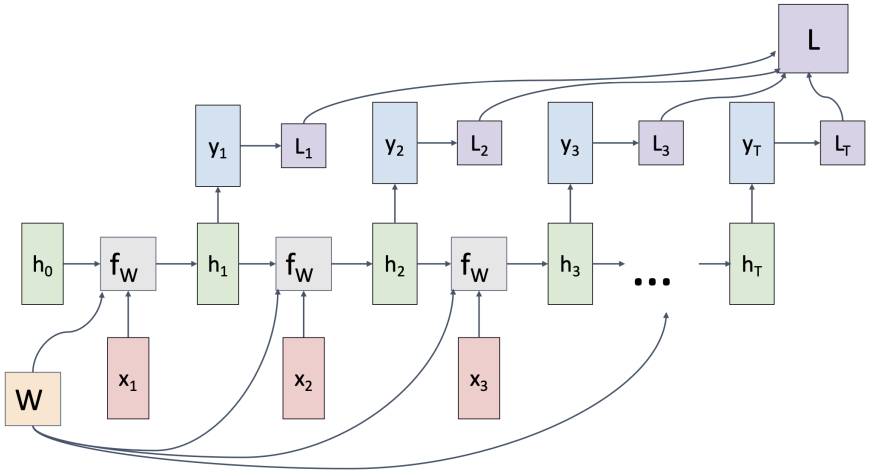


Figure 5.3: Loss of the unfolded graph.

Each output y_t produced can be fed into some loss, for instance by comparing it to some gound truth label. The individual losses $\mathcal{L}_t$ can then be summed into a total loss $\mathcal{L}$, from which the backpropagation will start. Therefore, if we want to compute the gradient $\frac{\partial \mathcal{L}}{\partial W}$, the gradient flow will start in $\mathcal{L}$, separate itself into the different temporal steps $\mathcal{L}_t$, and then combine itself back to W since the parameter is the same at each time step. This is called *backpropagation through time*, as the forward pass is done by generating outputs at increasing time steps, while the backward pass is done starting from the most recently generated outputs.

5.2.4 Many-to-one, one-to-many, many-to-many

The previous examples involve a mechanism in which an output y_t is produced for each input x_t. In the case of other types of models (many-to-one, one-to-many, mnay-to-many), we might produce no output for certain inputs, or outputs without inputs.

Many-to-one In the many-to-one situation, we usually decide to produce an output only for the final input. The decision is therefore made based only on the final hidden state of the layer, h_T; the intuition behind this is that h_T depends on all the previous hidden states h_t, and is therefore able to store the information used for the final decision.

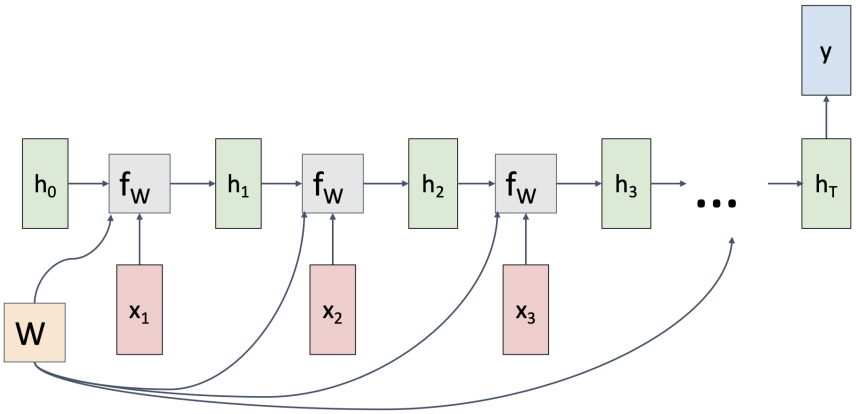


Figure 5.4: Many-to-one situation.

One-to-many In the one-to-many situation, we typically feed the function with the input only once. After that, the model will keep producing outputs depending solely on the previous hiddens state. This generation process usually stops when a certain output is produced, corresponding to some end-of-file token.

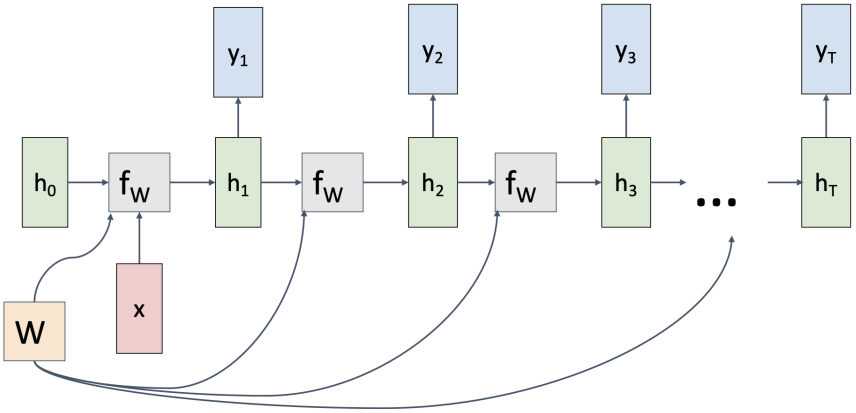


Figure 5.5: One-to-many situation.

Many-to-many While many-to-one and one-to-many situations are pretty straighforward, many-to-many requires more work, as it would be tricky to manage to make it work with only one recurrent neural network. For instance, in the case of machine translation, it is difficult to produce a translation before the whole input sentence is finished. It seems to be simpler to first read the original sentence, then generate a translated sentence. Furthermore, there is no guarantee that the best hidden states for input understanding are the same as the best hidden states for output generation. Therefore, the sequence to sequence situation is usually handled with two RNNs: an encoder and a decoder.

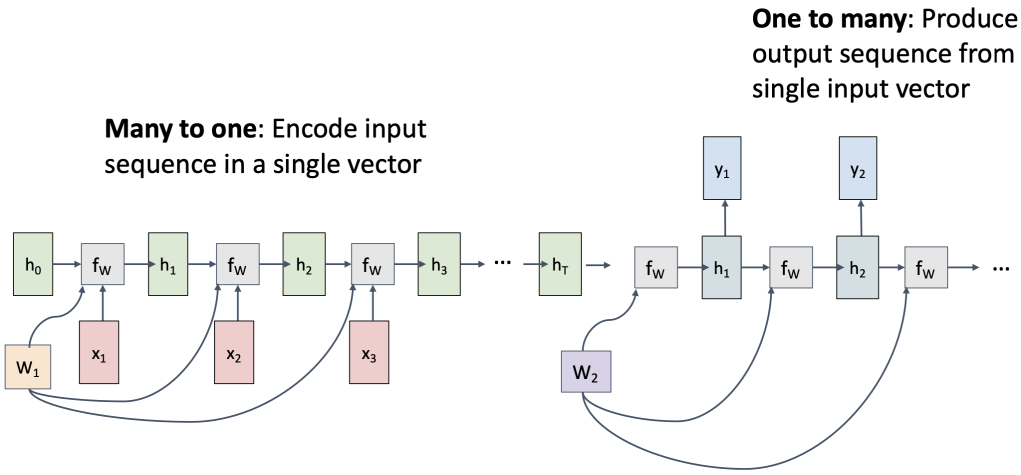


Figure 5.6: Encoder and decoder for the many-to-many situation.

The encoder will receive the input and summarize it all into the final hidden state of this first many-to-one RNN. This final vector is used by the decoder, a one-to-many RNN, which will use the accumulated context to generate outputs.

5.2.5 Applications

Recurrent Neural Networks open the door to many useful and fun applications.

For instance, RNNs can be trained on corpuses of text to generate new sentences following the same style. A nice `NumPy` implementation of such a vanilla RNN by Andrej Karpathy is available here[2]. When trained on large quantity of text such as Wikipedia, Shakespeare's work or the Linux kernel source code, it can produce new sentences following the style of the text corpus.

5.3 Gradient Flow in RNNs

Let's consider the previously introduced vanilla RNN. Its recurrence equation is of the form:

$$\begin{aligned} h_t &= \tanh\left(W_h h_{t-1} + W_x x_t\right) \\ &= \tanh\left(W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right) \end{aligned}$$

which corresponds to the following diagram:

[2] `https://gist.github.com/karpathy/d4dee566867f8291f086`

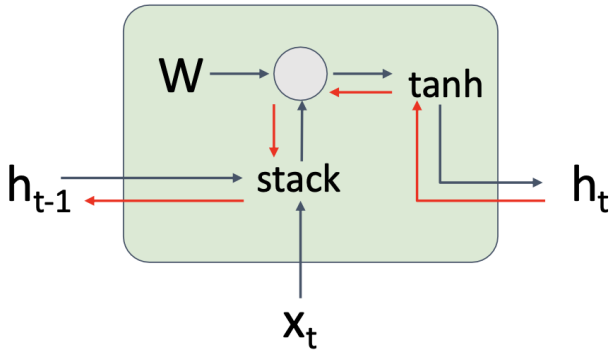


Figure 5.7: Simple computational graph of an RNN cell. Gradient flow is represented in red.

This architecture actually causes an issue in the way that the gradient flows throughout the graph. When unrolling the computational graph, we can see that in order to arrive up to h_0, the gradient must flow through every previous cell, and will therefore contain as many $W^\top$ factors as they are time steps.

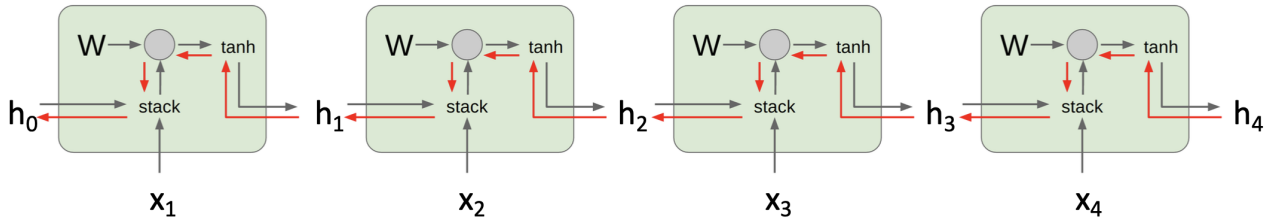


Figure 5.8: Unrolled computational graph of an RNN. Gradient flow is represented in red.

This can cause an issue in one of two possible ways. If the largest eigenvalue of the W matrix is greater than 1, the gradient will explode. If the largest eigenvalue if smaller than 1, the gradient will vanish.

A first solution to avoid the exploding gradient problem is *gradient clipping*. The idea is to scale the gradient if its norm is too big. Formally, we apply the transformation:

$$\nabla\mathcal{L} \longleftarrow \begin{cases} \nabla\mathcal{L} \cdot \frac{\theta}{\|\nabla\mathcal{L}\|} & \text{if } \|\nabla\mathcal{L}\| > \theta \\ \nabla\mathcal{L} & \text{otherwise} \end{cases}$$

where θ is some threshold.

To avoid vanishing gradient, the most widely used solution is to change the recurrent neural network architecture.

5.4 Long Short-Term Memory (LSTM)

5.4.1 Recurrent equation

The equation defining vanilla recurrent neural network is the following:

$$h_t = \tanh\left(W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right) \tag{5.4.1}$$

Instead, the *Long Short-Term Memory* (LSTM) architecture was introduced to solve the vanishing gradient problem. Its recurring equation goes as follow:

$$\begin{bmatrix} i \\ f \\ o \\ g \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{LSTM}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

Instead of keeping only one vector at each time step, two are used: the cell state c_t and the hidden state h_t. On top of this, four gates values, i, f, o and g are computed at each time step, which will be used to compute c_t and h_t. Each gate can somehow be intuitively interpreted:

- **Input gate** i specifies whether to write in the cell
- **Forget gate** f specifies whether to erase the cell from its previous content
- **Output gate** o specifies how much should the cell be revealed to the hidden state
- **Cell input gate** g specifies what to write in the cell

Note that the g gate is the only one to go through a tanh linearity. The i, f and o gates have coefficients in $[0, 1]$, which fits the opened/closed gate interpretation, while g has coefficients in $[-1, 1]$, which is the range of values of c_t.

The new cell value is a combinaison of its previous value c_{t-1} and the cell input gate g: the proportion of both values are controlled by the forget gate f and the input gate i. The new c_t value being computed, it is sent to the classical hidden state h_t; the output gate o controls what coefficients are being sent.

5.4.2 Gradient flow

While it might seem very complex, LSTM improves the gradient flow throughout the cell. An important behavior is that the gradient propagation from c_t to c_{t-1} only passes through an element-wise multiplication by f, and no matrix multiplication by $W^\top$.

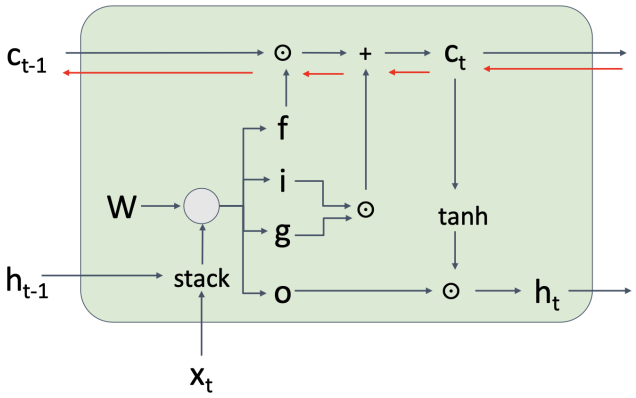


Figure 5.9: Gradient flow through an LSTM cell.

Therefore, chaining together multiple LSTM cells by unrolling the computational graph does not make the gradient vanish if f is working correctly: it creates an uninterrupted gradient flow from late cell states to early state cells.

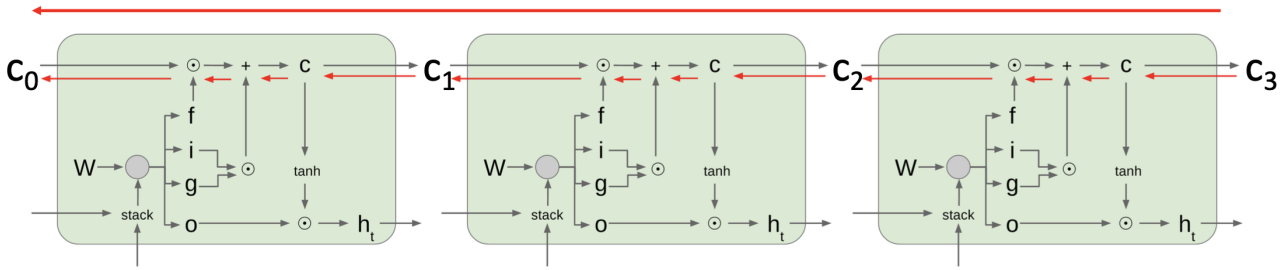


Figure 5.10: Uninterrupted gradient flow.

5.5 Multilayer Recurrent Neural Networks

So far, we only considered the use of a recurrent neural network with only one hidden cell. Empirically, neural networks with more layers often perform better, as we saw for MLPs and CNNs. We would like to apply this layering idea to RNNs.

The idea of multilayer RNNs is to feed then hidden states produced at each time steps into a second RNN, with different weight matrices.

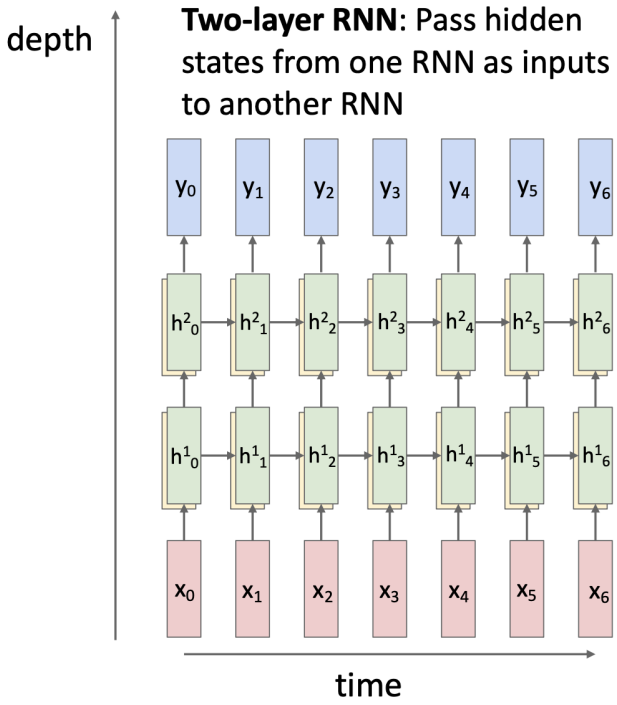


Figure 5.11: A two-layer RNN.

This process can be iterated to create deep recurrent neural networks, by adding two, three, ore more RNNs on top of each other. Note that while it is very common for CNNs to have very deep networks, RNNs are often used in practice with no more than three or four layers.

RNN: Summary

RNNs give us a lot of flexibility in architecture design by allowing one/many to one/many models. Vanilla RNNs are simple but don't work so well, because of exploding and vanishing gradients. LSTM is the most commonly used alternative to vanilla RNNs, used to solve the vanishing gradient problem.

6 Attention and Transformers

6.1 Sequence-to-sequence with RNNs and attention

6.1.1 Encoder-decoder RNNs and limitations

Originally, the transformer architecture was proposed for machine translation, and was later extended to other deep learning domains. To introduce the attention mechanism, let's consider a machine translation setup; we will start by building up on top of the RNN architecture introduced in the previous chapter.

We are given a sentence in English and want to translate it in French. To do so, we use two RNNs, an encoder which will handling the input tokens, and a decoder which will generate the output sentence.

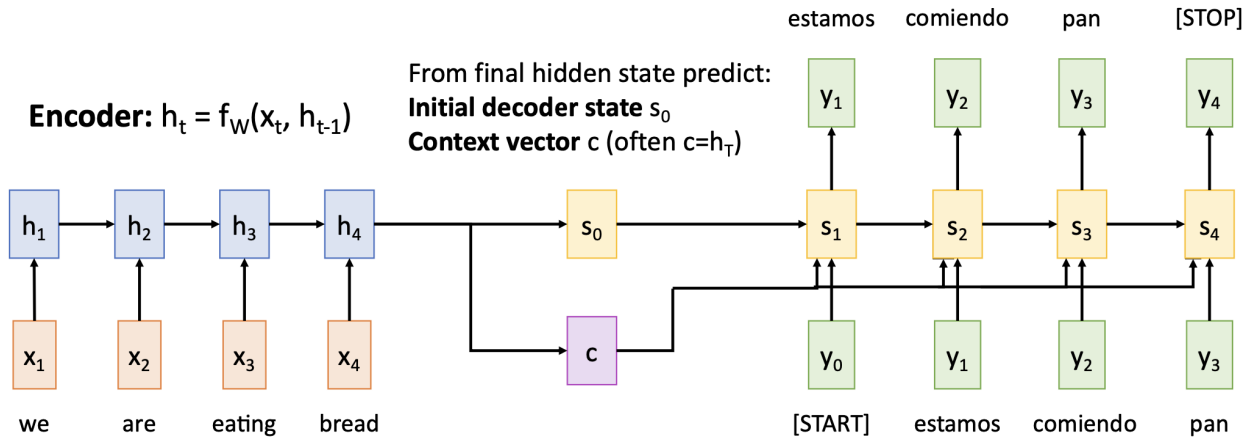


Figure 6.1: Sequence-to-sequence using an RNN.

After the processing of the original sentence, the encoder will summarize the entire context of that input sentence using two vectors: the initial decoder state s_0, and the context vector c. In practice, s_0 is often obtained by feeding h_T through a feed-forward neural network, and c is often set to h_T directly.

The decoder will receive a start token y_0 as its first input, as well as the context vector c; it will then generate the first output token y_1, which will be used as the second input. Note that the context c is fed to the decoder at each step of the generation, on top of the last generated token y_t. Formally, its recurring equation is of the form:

$$s_t = g_{W'}(y_{t-1}, s_{t-1}, c)$$

While this architecture is fairly reasonable, its bottleneck is that the entire context of the sentence must be summarized in the fixed-size context vector c. If the text to translate is too long (think of an entire book for instance), the model will not be able to fit all the context details in c. The idea to solve this issue is to compute a new context vector at each step of the decoder, and to allow the decoder to reconstruct the context vector by using different vectors focusing on different parts of the original sentence. This mechanism is called *attention*.

6.1.2 The attention mechanism

We will keep the general encoder-decoder structure, but instead change the way that the context is passed from the encoder to the decoder. We will use an MLP called f_{att}, which will compute

scalar alignment scores:

$$e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$$

Intuitively, the alignment score $e_{t,i} \in \mathbb{R}$ quantifies how much *attention* should be put in the hidden state of the encoder h_i, given the hidden state of the decoder s_{t-1}. These scalars will be used to construct a new context vector at each step of the decoder.

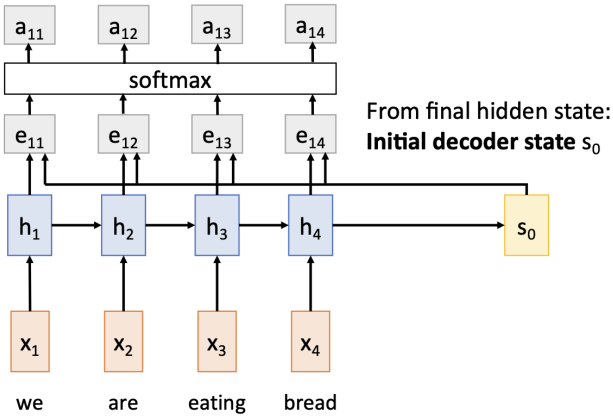


Figure 6.2: Applying softmax to alignment scores gives us attention scores.

The alignment scores are arbitrary real numbers. Therefore, we apply to them the softmax operations for each decoder state s_{t-1}, giving us *attention weights* $(a_{t,i})$ satisfying:

$$\sum_i a_{t,i} = 1$$

That being done, we can finally compute the context vector for each time step t of the decoder, using a sum of the hidden states (h_i) weighted by the attention scores $(a_{t,i})$:

$$c_t = \sum_i a_{t,i} h_i$$

This gives us the context vector c_t which will be used for the generation of y_t by the decoder. Note that this is all differentiable and allows us to backpropagate through the parameters of f_{att}; in particular, we do not need to supervise the attention weights.

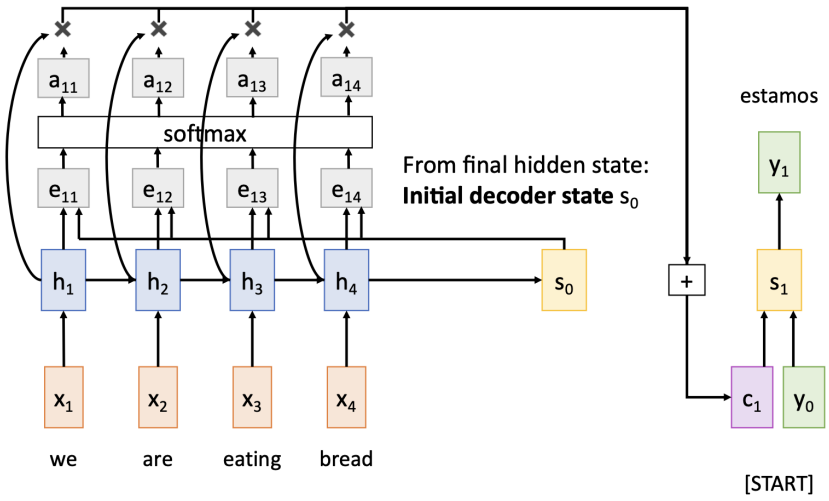


Figure 6.3: Computation of the context vector using attention scores.

This process that was applied for the first decoder step $t = 1$ can be iterated for every following step: we compute the alignment scores using the new hidden state s_1, then the attention scores, giving us the context vector c_t, which is fed into the decoder alongside y_{t-1}.

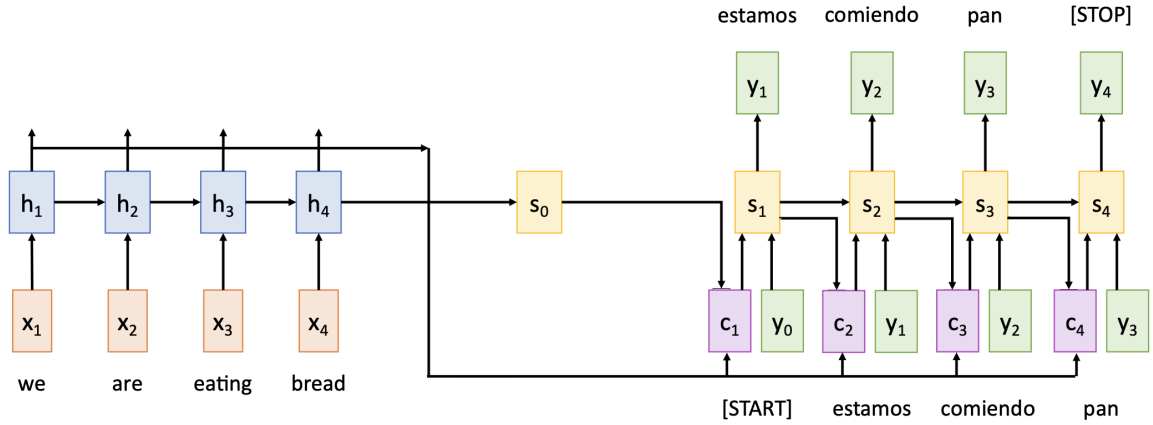


Figure 6.4: Unrolling the computational graph with attention.

This overcomes the bottleneck problem encountered for unique context vectors: at each step, the decoder is able to choose the relevant hidden states of the encoder, which selects only relevant information. When working with very long sequences, the model will be able to shift its attention around and focus on important parts of the inputs.

6.2 Visualizing and interpreting attention weights

The attention weights can be used to gain interpretability of the model: we can see for each output word which original words it was the most focused on.

Diagonal attention (the first four words and the last 5 words) means that words correspond in order between French and English. High coefficients outside the diagonal (for instance, "European"/"européenne") show words out of order between the two sentences. Finally, some lines and columns have more than one non-zero coefficient, such as the "was" or "say" columns: these show that the verb conjugation requires more than one word of context to be translated.

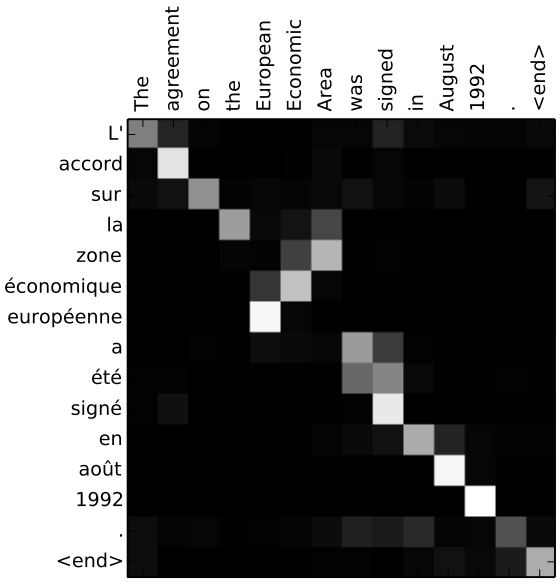


Figure 6.5: Visualization of attention weights for English-to-French translation.[3]

[3]Image taken from Bahdanau et al., "Neural machine translation by jointly learning to align and translate", ICLR 2015

6.3 Image captioning with RNNs and Attention

It is important to notice that the decoder does not use the fact that the hidden states $(h_i)_i$ form an ordered sequence, but only considers them as an unordered set $\{ h_i \mid i \in I \}$. This means that we can use the same architecture given any set of input hidden vectors $\{ h_i \mid i \in I \}$, especially for other types of data that do not form sequences.

Consider a deep convolutional neural network, without fully-connected layers at the end. The ouput of this CNN can be interpreted as a grid of feature vectors of the image. We see these feature vectors just like a sequence of hidden states of an encoder RNN: we can therefore apply the attention mechanism to them. We consider a simple model f_{acc} that we use to compute the alignment scores of each feature in the grid:

$$e_{t,i,j} = f_{\text{att}}(s_{t-1}, h_{i,j})$$

We then pass each grid of alignment scores into the softmax operator, giving us a normalized probability distribution, the grid of attention weights $(a_{t,i,j})$. Once again, the context of the image can be summarized with respect to the attention weights by computing a context vector c_t:

$$c_t = \sum_{i,j} a_{t,i,j} h_{i,j}$$

Feeding the context vector c_1 alongside the start token y_0 into the decoder RNN starting with hidden state s_0 can be used to generate a caption corresponding to the image.

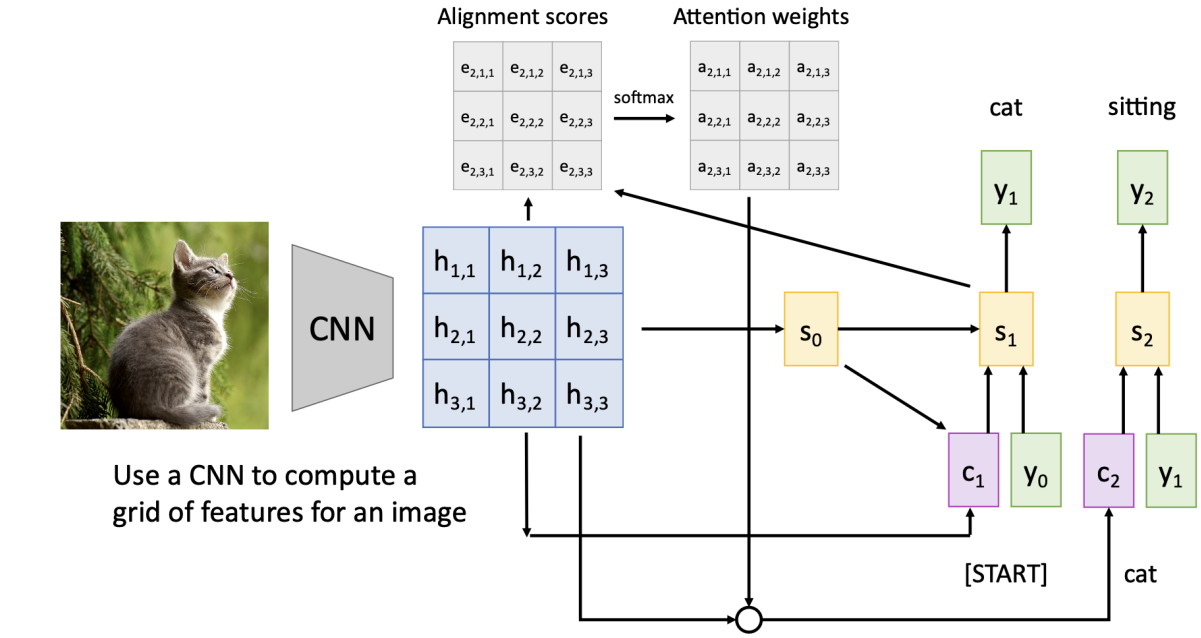


Figure 6.6: Applying attention to a grid of features for image captioning.

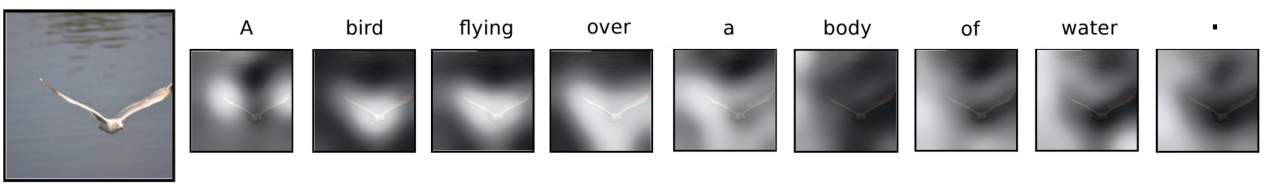


Figure 6.7: Visualizing attention over time during captioning.[4]

6.4 Attention Layer

We saw that attention could be used in a variety of situations, for input data such as hidden states of an RNN or features grid of a CNN. The attention mechanism can be generalized into an *Attention Layer*, which provides an abstraction to use attention in diverse contexts.

6.4.1 Simple generalization

Let's start by building an attention layer which exactly mimics the behavior described previously. In its simplest form, it uses the following inputs:

- A *query vector* q of size D_Q (the equivalent of the decoder states vectors s_t)
- A set of *inputs vectors* X of shape $N_X \times D_X$ (the equivalent of the encoder states vector h_i)
- A *similarity function* $f_{\text{acc}} : \mathbb{R}^{D_Q} \times \mathbb{R}^{D_X} \longrightarrow \mathbb{R}$ that can take q and a certain X_i as inputs and return their similarity (or alignment score)

Like we detailled previously, the computations done by the attention layer are:

- The similarities e of shape N_X, obtained by setting $e_i = f_{\text{acc}}(q, X_i)$
- The attention weights $a = \text{softmax}(e)$ of shape N_X
- The output vector $y = \sum_i a_i X_i$ of shape D_X

6.4.2 Changing the similarity function

While early papers on attention used a specialized function f_{acc} to compute similarity, it turns out to be much more simple and efficient to simply use a dot product for similarity, while keeping the same performances. Therefore, we change X to be of shape $N_X \times D_Q$, and we compute the similarities with:

$$e_i = q \cdot X_i$$

In practice, large similarities will cause softmax to saturate and give vanishing gradients: if one coefficient is much higher than the others, the softmax distribution will peak at this element, and have an almost-zero gradient everywhere. To avoid this, we normalize e by dividing it by $\sqrt{D_Q}$, giving us the *scaled dot product* for similarity:

$$e_i = \frac{q \cdot X_i}{\sqrt{D_Q}}$$

6.4.3 Multiple query vectors

In previous applications of the attention layer (sequence-to-sequence, image captioning), we always used a single query vector (the decoder states). We would like to generalize this to handle multiple query vectors at a time; therefore, we replace the query vector q by a set of query vectors Q of size $N_Q \times D_Q$. Instead of computing the similaritie score between the query vector and each input vector, we will compute the similarities for each query vector and for each input vector:

$$E_{k,i} = \frac{Q_k \cdot X_i}{\sqrt{D_Q}}$$

[4]Image taken from Xu et al., "Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

These operations can be written and implemented using a matrix multiplication:

$$E = \frac{Q \times X^\top}{\sqrt{D_Q}} \tag{6.4.1}$$

To ensure that each query vector is independent of the others, we take the softmax only over the D_Q dimension of E:

$$A_{k,i} = \text{softmax}_i(E_k) = \text{softmax}_i(E, \text{dim=1}) = \frac{e^{E_{k,i}}}{\sum_{i'} e^{E_{k,i'}}}$$

where A is therefore of shape $N_Q \times N_X$.

Finally, the output vector is replaced by a set of output vectors verifying:

$$Y_k = \sum_i A_{k,i} X_i$$

This output matrix of shape $N_Q \times D_X$ can be written directly as:

$$Y = AX \tag{6.4.2}$$

6.4.4 Key-value distinction

Note that in these computations, we are using the input vectors $X$for two different things. Firstly, we use them to compute the similarities with the query vectors Q (in Equation 6.4.1). Secondly, we use them to compute the output vectors Y (in Equation 6.4.2). In all generality, we can see this as two different things, and might want to use two separate sets of vectors to do so. Therefore, we often separate the input vectors into *key vectors* and *value vectors*.

The attention layer will still take as input a query matrix Q and an input matrix X, but will also use a learnable *key matrix* W_K of shape $D_X \times D_Q$ and a learnable *value matrix* W_V of shape $D_X \times D_V$. These learnable matrices will be used to translate the input vectors X into key vectors K and value vectors V:

$$\begin{cases} K = X \times W_K \\ V = X \times W_V \end{cases}$$

The similarities will then be computed using the key vectors:

$$E = Q \times K^\top$$

and the output vectors of shape $N_Q \times D_V$ will be computed using the value vectors:

$$Y = A \times V$$

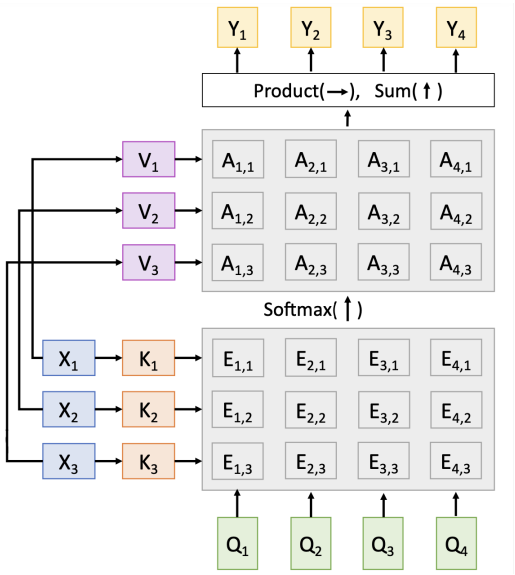


Figure 6.8: Schema of the attention layer making the distinction between keys and values.

6.5 Self-Attention Layer

6.5.1 Principle

We introduced a general architecture for the attention layer, which allows us to "search" for certain queries Q into some inputs X. A special case of this, often used in practice, is the *Self-Attention layer*: we only consider one input matrix X, which will be used both for inputs and queries.

Similarly to the key and value matrices, we will predict the query matrix Q using the input matrix X. We therefore add another learnable matrix W_Q of shape $D_X \times D_Q$, and we retrieve Q by computing:

$$Q = X \times W_Q$$

Everything else works in the exact same way as the attention layer with key-value distinction.

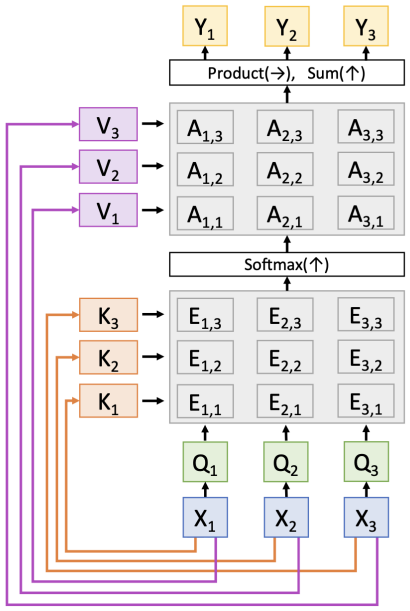


Figure 6.9: Self-attention layer.

6.5.2 Positional encoding

Note that permuting the input vectors will propagate the permutation up to the outputs; the self-attention layer is *permutation equivariant*, which can be interpreted as the fact that it works on a set of vectors.

This behavior might actually be unwanted, as the position of the vectors might bring more information. In order to make the self-attention processing aware of the positions, we often concatenate the input with the *positional encoding* of the vectors.

To do so, we often use a learned lookup table P, and append at the end of the i-th input vector the coefficient $P(i)$. This can also be done with a fixed – not-learned – function.

6.5.3 Masked Self-Attention Layer

For certain applications, such as language modelling where we want to predict the next word, we do not want the vectors to "look ahead" in the sentence, but only consider the previous vectors. In an RNN, this happens already by design, because of the way that the outputs are generated.

This behavior can be manually enforced in self-attention layers. To do so, we *mask* the layer, by setting some similarity scores $E_{k,i}$ to $-\infty$. This will set the corresponding attention scores $A_{k,i}$ to 0 because of the softmax formula; this way, the "future vectors" will not be taken into account when computing the weighted sum.

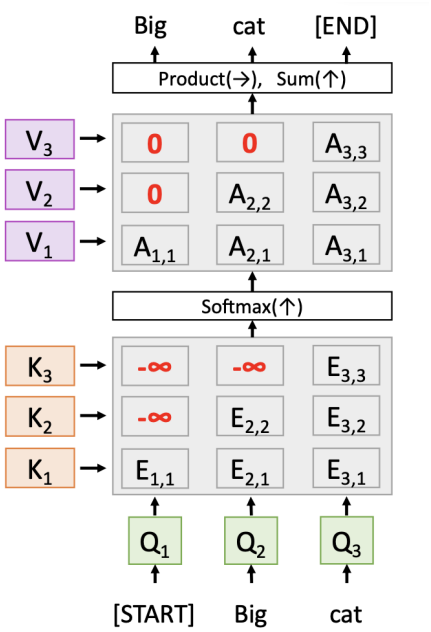


Figure 6.10: Masked self-attention layer.

6.5.4 Multi-Head Self-Attention Layer

Another variant of the attention layer is the *multi-head self-attention layer*. The idea is to use H "attention heads" in parallel, running H attention layers independently. Each input vector X_k is split into H chunks of equal sizes; all H sets of chunks are then fed into separate self-attention layers, producing H sets of output chunks which are then concatenated to produce the output vectors Y_k. This architecture is quite common in practice.

Such layers are used with two hyperparameters: D_Q, the query dimension, and H, the number of heads.

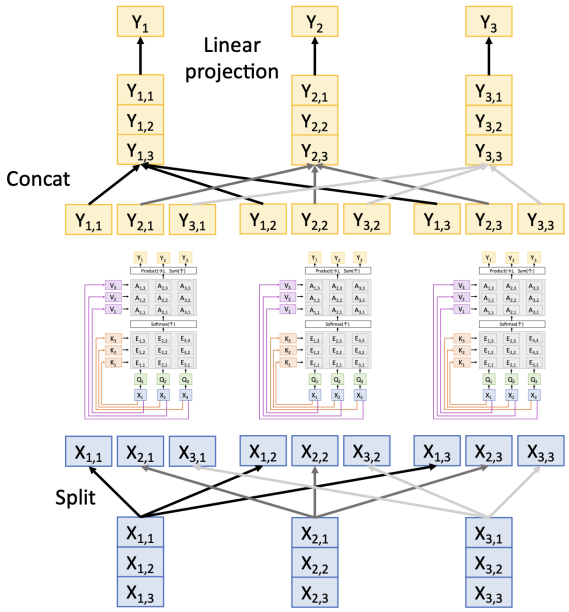


Figure 6.11: Multi-head self-attention layer.

6.5.5 Example: CNN with Self-Attention

A self-attention layer can be added at the end of a CNN to reproduce a similar behavior to the attention-based RNN for image captioning.

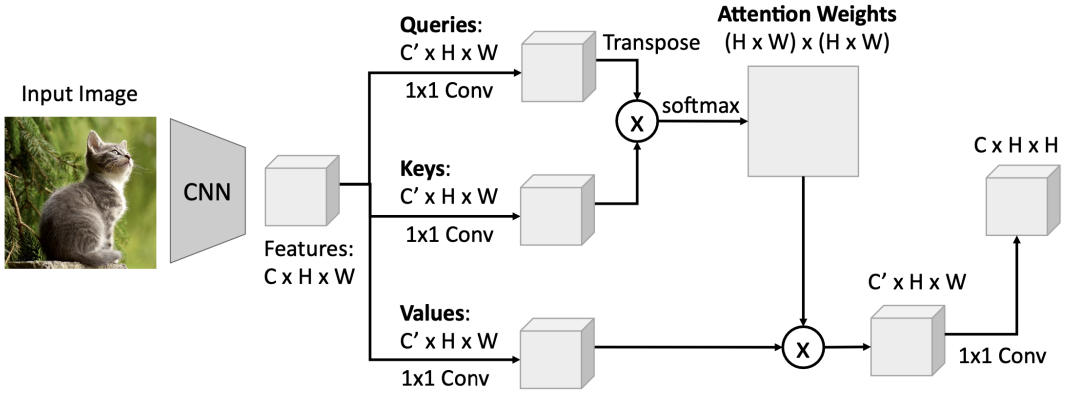


Figure 6.12: Self-attention layer used after the output of a CNN.

6.6 Transformers

6.6.1 Processing sequences

We saw three ways to process sequences, each with their upsides and downsides.

Recurrent Neural Networks are very good at processing long sequences, since the last hidden state h_T depends directly on all the previous inputs and hidden states. Their biggest drawback is that they are not parallelizable: we need to compute hidden states sequentially, which is unadapted to GPUs.

Convolutions, and especially 1D convolutions, provide a highly parallelizable alternative. Unfortunately, they do not really suit language processing and long sequences, since the context window of each output is restricted to the kernel size.

Finally, self-attention provide a great alternative, that is both adapted to long sequences, since each output vector depend on each input vector, and highly parallelizable. It remains very memory intensive, but can still run on large GPUs. What type of layers should we therefore use to build neural networks to process sequences?

6.6.2 The Transformer block

A very famous paper[5] showed that we can solely rely on attention layers to build sequence-processing networks. To do so, we build a new block type, called the *transformer* block, using only self-attention to compare input vectors.

The input vectors are first fed into a *multi-head self-attention layer*. To improve the gradient flow, a residual connection from the original vectors is added after the self-attention. The vectors are then independently normalized using *layer normalization*. Each vector is then fed into its own *MLP* (multi-layer perceptron); once again, a residual connection is added to the output. A final *layer normalization* step is added before the end of the block.

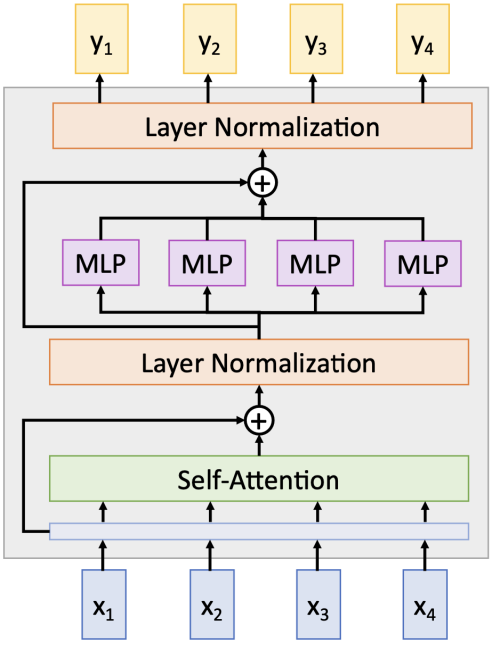


Figure 6.13: The transformer block.

Note that the self-attention layer is the only source of interaction between vectors: layer normalization and MLPs are performed independently. This architecture is highly scalable and highly parallelizable, because of the underlying self-attention properties.

6.6.3 Transformer model

A *Transformer model* is simply a sequence of transformer blocks. In the original paper, 6 transformer blocks were used for the encoder and 6 blocks for the decoder; each block had a query dimension $D_Q = 512$ and 6 attention heads.

Transformer models also are particularly adapted for finetuning: in natural language processing, transformers are often pre-trained using a lot of text from internet, and fed into a giant transformer model that can be later finetuned to fit specialized NLP tasks.

[5] Vaswani et al., "Attention is all you need", NeurIPS 2017

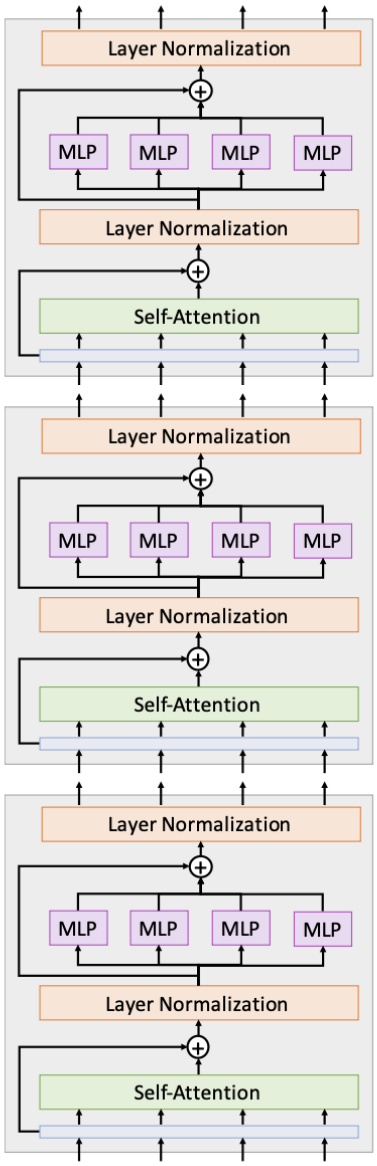


Figure 6.14: A transformer model using three transformer blocks.

Attention and Transformers: Summary

Adding attention to RNN models allow them to look at different part of the input sequence at different time steps; this avoids having to fit the entire context into one single fixed-size context vector.

Attention can be generalized into a self-attention layer, a powerful network primitive used to compare vectors with each others.

Transformers are new neural network models made of transformer blocks, simple wrappers around self-attentions layers.

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 $t \in [\![0, T]\!]$[6] the following process is iterated:

- The agent chooses an action A_t
- The environment computes the associated reward $R_t = \mathcal{R}(S_t, A_t)$

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

- 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:

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

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[7]:

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

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".

[7] 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.

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