Artificial Neural Networks and Deep Learning

Niccoló Didoni

September 2022

Contents

Part I

Introduction

Chapter 1

Deep learning

1.1 Deep learning, machine learning and artificial intelligence

Artificial intelligence, machine learning and deep learning are terms sometimes used interchangeably, however, they refer to different concepts. Artificial intelligence is a vast subject that aims at building intelligent systems (whatever the definition of intelligence may be). Artificial intelligence is made of different sub-subjects that aim at building an intelligent system with their own tools (e.g., tree search, graph theory and exploration, logic, planning and so on). Machine learning is a sub-field of artificial intelligence that aims at building intelligent systems that are able to learn from experience. Machine learning comprises many different techniques that allow computers to learn from experience (or data). A more precise definition of machine learning is given by Mitchell and goes as follows.

> **Definition 1.1** (Machine learning)**.** *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance P at task t in T, as measured by P, improves with experience E.*

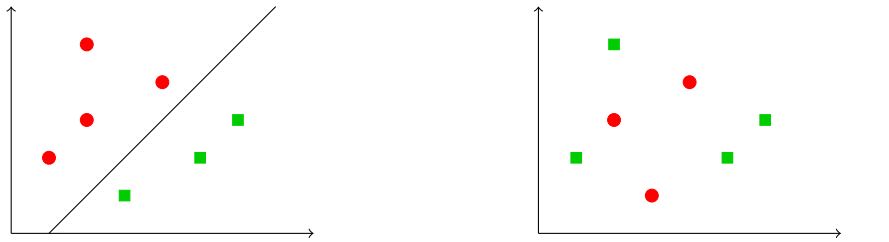
Neural networks are an example of a machine learning technique. The Definition 1.1 of machine learning given by Mitchell is very accurate but not that modern. In particular, in modern machine learning, we replace

- The concept of task with problems. In particular, in machine learning we can identify three kinds of problems:
  - **Supervised learning** problems. In supervised learning, a machine learns from a dataset that contains couples (x, t) where x is an input and t is the desired output related to the input x. The goal of the machine is to learn a function that maps the inputs x in the outputs t, even on data never seen before. If the output is discrete, we talk about **classification** problems, otherwise, if it's continuous, we talk about **regression** problems. An example of a supervised learning problem (classification) is telling photos of cats and dogs apart. The dataset contains couples where the first element is a photo of a cat or a dog while the second element contains the values 0 (representing a dog) or 1 (representing a cat). The machine uses the dataset to say if a new image depicts a cat or a dog.

  - **Unsupervised learning** problems. In unsupervised learning problems, a machine learns from a dataset containing only inputs x. Basically, the machine has to find regularities in the data set. For instance, a machine might want to group together users with the same interests given as input the videos watched on YouTube.
  - **Reinforcement learning** problems. In reinforcement learning problems, a machine has to choose the optimal decisions given the data to maximise a reward. Basically, the machine executes some actions and receives some reward afterwards. The goal is to find the actions that maximise the reward obtained.

- Experience with data.
- Performance with an error, or loss, function.

Having understood what's the difference between machine learning and artificial intelligence, we have to find out where deep learning fits in this picture. To understand what differentiates machine learning from deep learning we have to better analyse what's the goal of machine learning. Let's consider a classification problem. If we consider a data set $D = \{([x_{11}, x_{12}], t_1), \ldots, ([x_{1n}, x_{2n}], t_n)\}$ where each input point is done by two coordinates and each point belongs either to class C_1 ($t = 0$) or C_2 ($t = 1$). Given this scenario, we can represent the dataset in a bi-dimensional plane where the points belonging to class C_1 are shown as squares and points belonging to class C_2 as circles. In some cases, as in Figure 1.1a, the input points can be separated by a line, hence we say that they are linearly separable. On the other hand, as in Figure 1.1b, some datasets might contain scattered points that can't be separated by a linear boundary.



(a) A linearly separable dataset. (b) A non-linearly separable dataset.

Figure 1.1: An example of two datasets, one linearly separable, the other not linearly separable.

When approaching a machine learning problem, we have to find the right model to learn the correct classifier (i.e., the function that classifies the points in input). Learning the correct classifier sometimes is related to a correct representation of the data. In particular, every input point can be done by multiple features (i.e., coordinates on a Cartesian plane). If we think at our example, the features of an input point $\mathbf{x} = [x_1, x_2]$ are x_1 and x_2. When considering a machine learning problem can also combine features or use functions of the features to create new features (e.g., $x_3 = x_1^2$, $x_4 = x_2^4$). Finding a good model depends on the features we choose to use. Representing the data with the wrong set of features might lead to non-separable data. This process is usually done manually. Deep learning comes into play exactly in this context, in fact, it extends machine learning to the feature extraction part. Basically deep learning helps extract the right features so that we can later apply machine learning algorithms to them. This means that the term deep isn't related to the size of the dataset, but to the extent to which we apply machine learning. Basically,

- In machine learning, the algorithms are applied only to the learning phase.

- In deep learning, machine learning algorithms are applied to the feature extraction part and to the learning phase. In particular, deep learning pushes all its effort into learning which parameters are better to represent the data set and then uses a simple regression or classification problem for the actual learning part. The feature learning phase is usually done in a hierarchical way in which the machine initially learns general, high-level, features and then focuses on more complex and detailed ones.

For this reason, deep learning is also called **end-to-end learning**. To wrap things up, deep learning is about learning data representation from data, i.e., how to represent the data from the data itself. In other words, we want to find the right features from a given dataset D, so that D can be easily separated. Note that this process requires a lot of data. To have a complete picture, let us say that:

- **Artificial intelligence** is a discipline of computer science and mathematics that includes different techniques.
- **Machine learning** is a set of artificial intelligence techniques in which features have to be chosen by a human designer.
- **Deep learning** is a field of artificial intelligence that applies machine learning both to the learning and to the feature extraction phases.

Part II

Neural Networks

Chapter 2

Perceptron

2.1 Neurons

2.1.1 Perceptron model

Neural networks were born with artificial intelligence and machine learning and not with deep learning. In particular, at the beginning of AI and machine learning, researchers wanted computers to learn and they tried to emulate the functioning of a brain made of many neurons connected one to the other. To emulate the behaviour of a human brain we have to understand how actual neurons work. In a brain:

- The neurons can exchange electrical signals.
- The computation is distributed among all neurons.
- The architecture is fault-tolerant since, when a neuron dies, another can replace its job.
- Each neuron computes a non-linear function.
- Each neuron has incoming and outgoing connections with other neurons.
- When a neuron is reached by an electrical signal, the charge is collected inside the neuron and finally, when a certain level of charge is reached, the charge is released (sending an electrical signal to other neurons).

Given this description, researchers have developed the perceptron model. A perceptron, as shown in Figure 2.1:

- Receives a collection of inputs x_i, from x_1 to x_I.
- Computes the weighted sum, using weights $\mathbf{w} = [w_1, \ldots, w_I]$, and the function $\sum$, of the inputs $\mathbf{x} = [x_1, \ldots, x_I]$.

$$\sum_{1=0}^{I} w_i x_i$$

- Removes a bias b from the weighted sum of the inputs.

$$\sum_{i=1}^{I} w_i x_i - b$$

  Note that the bias can be seen as an input x_0 with weight $-w_0$. This means that the sum can be rewritten as

$$\sum_{i=0}^{I} w_i x_i$$

- Fires if the weighted sum, considering the bias b is greater than 0. In general, h, which is called **activator**, could be whatever non-linear function of the sum of the input and the weights, which computes if the neuron should fire or not. Usually, we consider a step function

$$h(\mathbf{x}, \mathbf{w}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{I} x_i w_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

  which has a high value (1 in this instance) if the sum is positive (i.e., the sum of the inputs is bigger than the bias) or a low value (0 in this instance) if the weighted sum is negative (i.e., the weighted sum of the inputs is smaller than the bias). Another good choice for high and low values is 1 and -1 (instead of 1 and 0). We will mostly stick to this second convention.

Basically, the neuron (i.e., the perceptron) fires (i.e., $h(x|w, b) = 1$) only if the weighted sum of the inputs (without bias) is greater or equal to a certain threshold, which is the bias. To see this in practice we can write that the neuron fires when

$$\sum_{i=1}^{I} w_i x_i - b > 0 \iff \sum_{i=1}^{I} w_i x_i > b \tag{2.1}$$

Note that the output of the summation neuron can also be written in vector form as

$$\mathbf{x}\mathbf{w}^T = \mathbf{w}\mathbf{x}^T \tag{2.2}$$

where

- $\mathbf{x} = [x_0, \ldots, x_I]$ is the row vector of the inputs.
- $\mathbf{w} = [w_0, \ldots, w_I]$ is the row vector of the weights.

Figure 2.1 shows the complete structure of a neuron, however, we can also use a more compact one in which the summation and step function nodes are condensed in a single node that computes the sum (including the bias) and fires whenever the sum is greater than 0. The new representation is shown in Figure 2.2.

Before going on, let us define once and for all a consistent notation for the cardinalities involved in a data set.

- The number of data points (i.e., of entries) of the data set is N. Basically a data set has N points $\mathbf{x}_1, \ldots, \mathbf{x}_N$.
- Each data point $\mathbf{x}_n$ has I features (or inputs). This means we can write $\mathbf{x}_n = [x_1, \ldots, x_I]$.
- The number of neurons in a network is J.

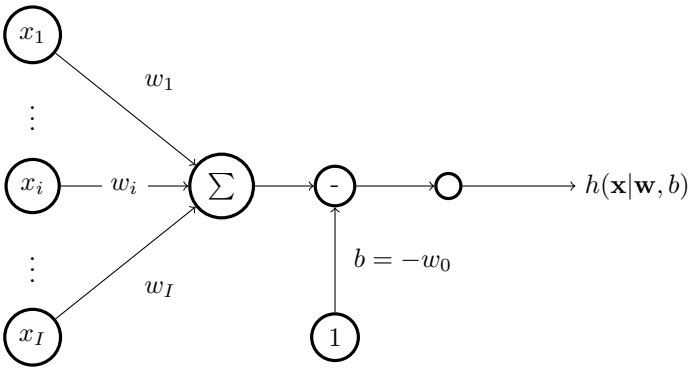


Figure 2.1: A neuron.

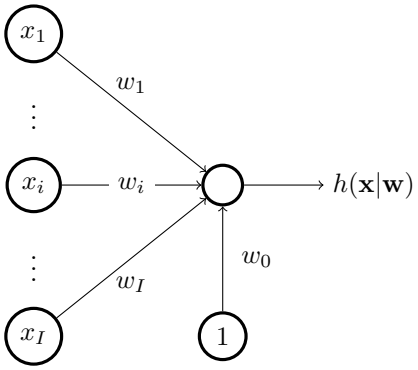


Figure 2.2: The compact representation of a neuron.

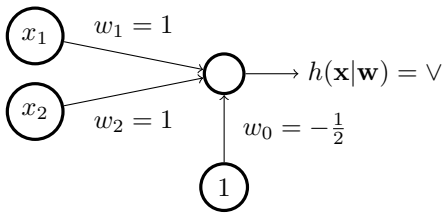
Logic operations with perceptrons

To experience the power of perceptrons we can show that a single neuron (i.e., a perceptron) can be used to build some logical operations such as the and and the or operations. Since or and and are binary operations, the neuron should have two inputs x_1 and x_2 and one output, which is the result of the operation. By trial and error, one can verify that if we assign $w_0 = -\frac{1}{2}$, $w_1 = w_2 = 1$ then $h(\mathbf{x}|\mathbf{w})$ is the result of the or operation. At the same way, if we consider $w_0 = -2$, $w_1 = \frac{3}{2}$ and $w_2 = 1$ then $h(\mathbf{x}|\mathbf{w})$ computes the and operation.

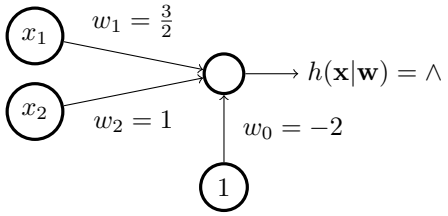
2.1.2 Hebbian learning

As for now, we have considered only the model of a neuron. This means that, if we have an input point $\mathbf{x} = [x_1, \ldots, x_I]$ we can compute if the neuron fires or not given such input.

What we want to understand now is how we should train such a model. Namely, we want to find out what are the values of the weights (which means also of the bias b, if we consider $b = -w_0$) for which the neuron can model the function we are interested in. To be more clear, say we have a data set generated by some function f (e.g., the or function) with the inputs and the related outputs computed by f. This means that the data set is made by couples $(\mathbf{x}_i, t_i)$ where $t_i = f(\mathbf{x}_i)$. Either we don't know f or we don't know how to represent it; in both cases, we want to use the data set to learn f. In the perceptron model, learning f means finding the right weights $\mathbf{w}$ for which



(a) A neuron computing the or operation.



(b) A neuron computing the and operation.

Figure 2.3: Neurons computing the and and or operations.

$h(\mathbf{x}|\mathbf{w}) \sim f(\mathbf{x})$ where $\sim$ indicates that h is as close as possible to f, i.e., that h has the minimum error with respect to f. The concept of error will be clearer later, however, let us imagine it has a quantity that tells us how good h is in modelling the behaviour of f. Also, note that h has to work not only on the behaviour of f on the data set but also on points it has never seen before.

The algorithm for training a perceptron is called Hebbian learning and it's roughly based (again) on the behaviour of a human brain. When we learn to do something (e.g., a mechanical action, like typing on the keyboard, or changing shift in a manual car) we are slow at doing what we learn. The more we train, the more we get faster and the actions we do are automatic. This can be seen as the bias reducing when a neuron fires. Namely, if a neuron receives a charge c and it's released in output (i.e., the charge is greater than the bias), then the next time the neuron will fire with a lower charge. The algorithm works as follows.

Algorithm 2.1 (Hebbian learning)**.** *Mathematically, we want to:*

1. *Initialise the weights $w_1, \dots, w_I$ with random values.*
2. *Take an input point $\mathbf{x}^{(k)}$ and feed the neuron with it.*
3. *Check the output of the neuron and*
   - *If the output $h(\mathbf{x}^{(k)}|\mathbf{w})$ corresponds to the expected output t_i, we do nothing.*
   - *If the output $h(\mathbf{x}^{(k)}|\mathbf{w})$ doesn't correspond to the expected output $t^{(k)}$, we update the weight with the following formula*

$$w_i^{(k+1)} = w_i^{(k)} + \Delta w_i^{(k)} \tag{2.3}$$
$$= w_i^{(k)} + \eta \cdot x_i^{(k)} \cdot t^{(k)} \tag{2.4}$$

4. *Go back to point 2 using another point $\mathbf{x}^{(k+1)}$.*

Let us now stop for a second to better understand the Hebbian learning algorithm. First, let's clear up the notation. The algorithm is iterative and at each iteration, we use a different data point. The iterations are numbered from 1 to K, hence we call:

- $\mathbf{x}^{(k)} = [x_1^{(k)}, \dots, x_I^{(k)}]$ an input point used at iteration k.
- $t^{(k)}$ the expected output of the input point $\mathbf{x}^{(k)}$ at iteration k.

- $w_i^{(k)}$ the value of the i-th weight when ending iteration k.

Let us now focus on how Δw_i^k is computed. This value represents the change that should be applied to each weight w_i and it's computed using:

- The **product** between the i-th input feature $x_i^{(k)}$ and the output $t^{(t)}$. Note that if the input has the same sign of the output, the value of Δ is positive hence the weight increases. On the other hand, if the signs are different, Δ is negative and the weight is decreased. This is the part that models human behaviour. If we receive a positive value from x_i and a positive value is released in output, then x_i is related to the output, hence we should increase the weight of w_i to say that x_i is important to compute the output. Also, note that increasing a weight is like decreasing the threshold.
- A **learning rate** η. The learning rate tells by how much the weights are increased or decreased. In the perceptron model, changing the value of η isn't very effective on the learning, hence one might want to use $\eta = 1$ for simplicity.

Finally, we have to define a couple more terms, which are important when talking about learning for neural networks in general. The Hebbian algorithm is iterative, which means that we try to improve the model using a point $\mathbf{x}_i$ at each iteration (instead of using all points at the same time). This type of procedure is called **online** learning. Another important thing to understand is that when we use all the points in a data set, we can reuse the points we already used to keep training the model. Every complete iteration of all points (i.e., every pass through the data) is called **epoch**. This means that if we have a data set of three points $D = \{(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), (\mathbf{x}_3, t_3)\}$, the algorithm uses $\mathbf{x}_1$, then $\mathbf{x}_2$, then $\mathbf{x}_3$ and an epoch ends. Now the algorithm can use $\mathbf{x}_1$ again and start a new epoch. This means that an epoch is very different from an iteration since an iteration considers one data point, while the epoch all the points in the data set.

One might ask why we want to use a point multiple times. To answer this question, say we initialise a network randomly (i.e., random weights) and we train it for the first data point $\mathbf{x}_1$. After training on $\mathbf{x}_1$ the output is correct when feeding the perceptron with $\mathbf{x}_1$ (this isn't necessarily true, but we'll do this assumption for this example). Now we use all the other points, which might change the weights, and the output, when using $\mathbf{x}_1$ might be wrong, hence we have to keep training the network.

Convergence

Previously we have presented the Hebbian learning algorithm, without giving any assurance on whether it always terminates and finds a good model for the data. In some cases we might get stuck and find out that we can't correctly model the data set. In general we can say that

Theorem 2.1 (Hebbian learning convergence)**.** *If the perceptron can learn the data in the data set D, then the Hebbian algorithm will converge independently from the starting point (i.e., from the random initialisation of the weights).*

This theorem tells us that if the perceptron is able to learn from the data, then the Hebbian algorithm will be able to find some weights $\mathbf{w}$ that can model the unknown function f.

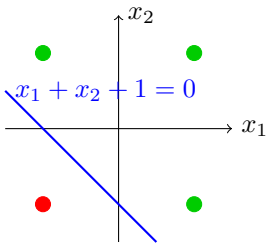
Theorem 2.1 says that the perceptron has to be able to learn from data, however we haven't said what does it mean yet. To understand this proposition, we have to look at the shape of a neuron's

output. Since we are considering a classification problem (with two inputs, so that we can draw the points in a bi-dimensional space), we can easily see that the equation used to split the points

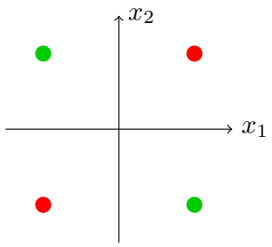
$$w_1 x_1 + x_2 w_2 = b \tag{2.5}$$
$$w_1 x_1 + x_2 w_2 - b = 0 \tag{2.6}$$

is a plane in the x_1, x_2 Cartesian plane. All points above that plane, i.e., for which $w_1 x_1 + x_2 w_2 > b$, are assigned to class C_1 while the points above the plane, i.e., for which $w_1 x_1 + x_2 w_2 < b$, are assigned to class C_2. In general (i.e., for an arbitrary I), the dividing plane is an hyperplane.

When it's possible to find a plane $w_1 x_1 + x_2 w_2 = b$, we say that the data is linearly separable and the perceptron can learn the data. In some cases, it's not possible to linearly separate the data, hence the perceptron can't learn. An example of linearly separable data is shown in Figure 2.4 while an example of non-separable (i.e., non-learnable) data is shown in Figure 2.4b. The perceptron isn't therefore a model for all problems and in fact it can't even be used for a simple task as computing the xor of two numbers. This problem is called the **xor problem**.



(a) The data set representing an or function.



(b) The data set representing a xor function.

Figure 2.4: A linearly separable and a linearly non-separable data set.

Since the perceptron can only divide data which is linearly separable, we say that the perceptron is a linear classifier.

Chapter 3

Neural networks

3.1 Multi-layer perceptron

Since a single neuron can't learn every data set (in particular it can't learn non separable data sets), we have to find more complex models. In machine learning one could use linear regression with basis functions or support vector machines exploiting the kernel trick. We will instead try to combine neurons to create complex shapes and architecture which might be able to learn non linearly separable data. Since we use many layers of perceptrons, we will call this new model **multi-layer perceptron**.

3.1.1 Activation function

Before introducing the general structure of a neural network, we have to revise the perceptron. In particular, we have to change the activation function used in a neuron. Previously, a neuron used a step function which is fine for a single perceptron, since Hebbian learning doesn't require to compute the derivative of the activation function h (remember that a step function isn't differentiable). Since we'll work with derivatives we have to switch to a differentiable activation function. Some examples of functions used as activation function are

- The **sigmoid function**
$$g(a) = \frac{1}{1 + e^{-a}} \tag{3.1}$$
The derivative of a sigmoid function is
$$g'(a) = g(a)\big(1 - g(a)\big) \tag{3.2}$$
- A **linear function**
$$g(a) = a \tag{3.3}$$
The derivative of a linear function is
$$g'(a) = 1 \tag{3.4}$$
- The **hyperbolic tangent**
$$g(a) = \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} \tag{3.5}$$

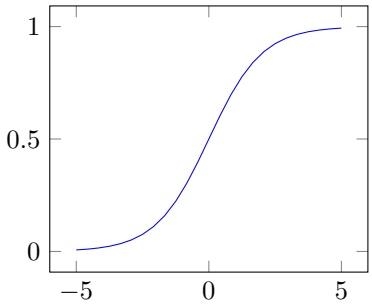


Figure 3.1: A sigmoid activation function.

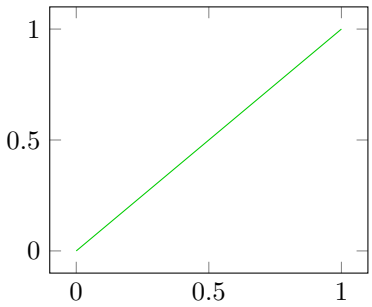


Figure 3.2: A linear activation function.

The derivative of an hyperbolic tangent function is

$$g'(a) = 1 - g(a)^2 \tag{3.6}$$

Note that, differently from the step function, the functions we have introduced return a value between 0 and 1 or between -1 and 1 (or in general between any two numbers). Deciding which activation function should be used depends on the specific problem, and on the output of the network. For instance, if we have to solve a regression problem with one output, that can take values from $-\infty$ to $+\infty$, we can use a linear function.

For the nodes in the hidden layers, we can either use a sigmoid or hyperbolic tangent activation

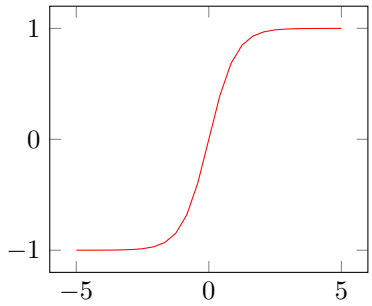


Figure 3.3: An hyperbolic tangent activation function.

function. We can even use a different type of function on different layers.

3.1.2 Neural network structure

A neural network is made of

- A layer of input nodes, called **input layer**. The input layer is made of I nodes containing the input variables (i.e., features) $x_1, \ldots, x_I$.
- One or more layers of neurons, called **hidden layers**. Each layer can have a different number of neurons. Selecting the right number of hidden neurons and layers is a difficult problem (which hasn't been solved yet). Usually, we have to do a model search using different topologies and number of neurons, using for instance cross-validation.
- A layer of output nodes called **output layer**. The output layer outputs the prediction of the classifier (or regressor). The output layer might have one or more nodes, depending on the problem. For instance in a multi-class classifier (with more than 2 classes), we have as many output nodes as the number of classes. With more than two classes, we usually use the one-hot representation in which class C_{i+1} is represented with a bit set to 1 in the i-th position. This means that if we have 3 classes, we need an output vector of three elements and class 1 is represented by the configuration $[1, 0, 0]$, class 2 by the configuration $[0, 1, 0]$ and class 3 by the configuration $[0, 0, 1]$. Also note that when the i-th node fires, it's output is the probability that the input belongs to class $i + 1$. Moving to regression, an example of problem in which one might want multiple outputs is predicting the coordinates of a point in a two-dimensional space.

Each node of a layer is connected to all nodes of the previous and next layer. Each connection is associated to a weight $w_{ij}^{(l)}$ in which

- l is the number of the layer to which to which we are connecting.
- j is the node of the layer l to which we are connecting.
- i is the index of the input.

The general structure of a neural network is shown in Figure 3.4. When using the network for getting a prediction (e.g., classifying an input point), the input features are putted in the input nodes and are propagated through the hidden layers towards the output layer. For this reason, the network we are studying is called **feed-forward neural network**.

If we observe the structure of a neural network we understand that the output of the whole network is the weighted sum of the outputs of a non linear function (possibly, if we consider for instance a sigmoid activation function), which is the weighted sum of the outputs of a non linear function, which is eventually the weighted sum of the inputs. This means that the output of a neural network is highly complex. In particular, it's that complex, and powerful, that the following theorem holds.

Theorem 3.1 (Universal approximation)**.** *A single hidden layer feed-forward neural network with S shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set.*

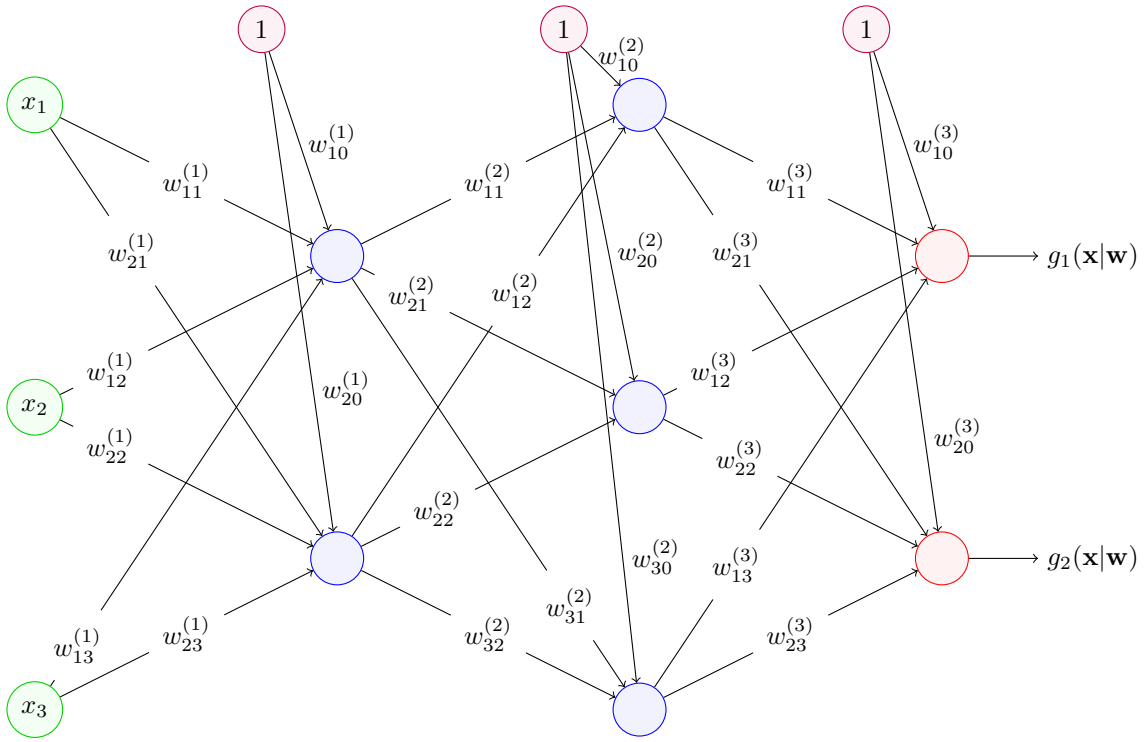


Figure 3.4: A neural network with two hidden layers.

3.1.3 Output nodes

Since the output layer is very important, we should describe it more in detail. First of all, the output layer can have any number of neurons. First, let us consider a binary classifier. In this case, we can either

- Use an hyperbolic tangent and encode class C_1 with the output 1 and class C_2 with the output -1.
- Use a sigmoid function and encode class C_1 with the output 0 and class C_2 with the output 1. In this case, the output is the probability of belonging to class C_2.

When we move to multi-class classification problems, we can encode the output with the **one-hot notation**. In this case, the output is encoded as a vector of K elements (K being the number of classes) and neuron k returns the probability that the input belongs to class C_k. The input is assigned to the class for which the probability is higher and the corresponding value in the output vector is set to 1 (whilst the others are set to 0). Consider for instance a three-class classification problem. Given an input $\mathbf{x}$, the output neurons of the network compute $[0.9, 0.1, 0.1]^T$. Since the first value is higher we obtain the output vector $[1, 0, 0]^T$ which tells us that $\mathbf{x}$ has been assigned to class C_1. Notice that the vector $[0.9, 0.1, 0.1]^T$ should be a probability vector, however the sum of probabilities doesn't sum up to 1. This could be a problem, hence we have to consider a normalised output for each neuron. The normalised output can be computed using the softmax function as

follows

$$y_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

where

- z_k is the output of the k-th output neuron.
- y_k is the k-th element of the output vector.

The softmax function doesn't only normalise the output probabilities, but it also increases the highest value and reduces the smallest (thanks to the exponential function). The same wouldn't have happened using a simple normalisation function

$$y_k = \frac{z_k}{\sum_{k=1}^{K} z_k}$$

which is only able to normalise the output.

In some cases an input point might belong to multiple classes. In such cases, we can use a logistic function on each neuron to check if the input belongs to a neuron.

3.2 Learning

3.2.1 Gradient descend

Now that we know how a neural network is made, we have to study how it's trained. Following Theorem 3.1 we can say that a neural network is a function y of x, w and b. Optimising the network and its parameters means therefore to find the set of parameters for which the output g of the network is close and similar to the desired output t

$$g(\mathbf{x}_n|\mathbf{w}) \sim t_n \tag{3.7}$$

This description is rather abstract since we haven't defined the concept of being close to the desired output. In machine learning, to find the best output function, we usually minimise the sum of squared errors J computed over the input points. This means that, if we have a data set

$$D = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$$

we compute the error, or cost,

$$J = \sum_{n=1}^{N} \big(t_n - g(\mathbf{x}_n|\mathbf{w})\big)^2$$

Note that this isn't the only error we can use (another example is cross entropy, which will be analysed later on) but we'll use it for now since it's quite straightforward to use. Now that we have a way to measure the error of a neural network (i.e., to measure how bad it is), we have to minimise such error. As always, to minimise a function (actually to find its stationary points), we have to compute its derivative and put it to 0.

$$\frac{\partial J(w)}{\partial w} = 0$$

Unfortunately, it's not possible to find a closed form solution to this equation for a feed-forward neural network, hence we have to use **gradient descend**. In particular, we can compute the value of a certain weight w at time l as

$$w^{(l+1)} = w^{(l)} - \eta \frac{\partial J(w)}{\partial w}\bigg|_{w=w^{(l)}}$$

Note that the minus sign is justified by the fact that the gradient returns the growing direction of the error function J, however we want to minimise its value, hence we have to go in the opposite direction. When performing gradient descend, differently from Hebbian learning, we have to choose appropriately the value of the learning rate η. In particular:

- If η is too big, we could bounce around the minimum.
- If η is too small, learning requires a lot of time.

This means that η should be bigger for wider valleys, however, while learning, we don't know the exact shape of the error function. This means that we have to be able to find a good trade-off between the learning speed and the preciseness of the algorithm.

Another important problem to consider when talking about gradient descend is the presence of local minima. In particular, the algorithm doesn't guarantee us to reach a global minima, in fact it could get stuck in a local minima (whose value is grater than the one of the global minima). To mitigate this problem, we can use some physics. In particular, we can add the concept of momentum, which adds some memory of the previous state of the system, to the weight update formula. The new formula becomes therefore

$$w^{(l+1)} = w^{(l)} - \eta \frac{\partial J(w)}{\partial w}\bigg|_{w=w^{(l)}} - \alpha \frac{\partial J(w)}{\partial w}\bigg|_{w=w^{(l-1)}}$$

The new update formula might solve the local minima problem, however:

- We have a new parameter α to set. Some algorithms (e.g. ADAM) are able to tune η and α automatically, however in general having one more parameter to set might be time and resource consuming.
- We could still get stuck in a local minimum. Repeating the learning phase many times (starting with a different initialisation every time we start a new learning process). The idea is to start from a random point, execute the learn algorithm and save the minimum value. Then execute the algorithm again starting from a new value and saving the minimum value found after learning. After repeating the learning process many times we can keep the minimum value obtained among all the minima found.

Note however that not finding the global minimum is not a big deal because of the overfitting problem. Basically, if we find the global minimum we could find a function that perfectly represents the data-set but that doesn't perform well on new, unseen data.

3.2.2 The error's derivative

For starters let us assume that the topology is fixed and in particular let us consider a simple but general topology with, as in Figure 3.5, with the following characteristics:

- It has only **one hidden layer**.

- The network is **dense**, or fully connected, meaning that every node of layer l is connected to every other node in layer $l+1$. Moreover, connections can't skip one layer (e.g., a node of layer l can't be directly connected to a node in layer $l+2$).
- The input layer has I nodes.
- The hidden layer has J neurons.
- The output layer has 1 neuron.
- The output of a node in layer l depends only on the nodes in the previous layers.

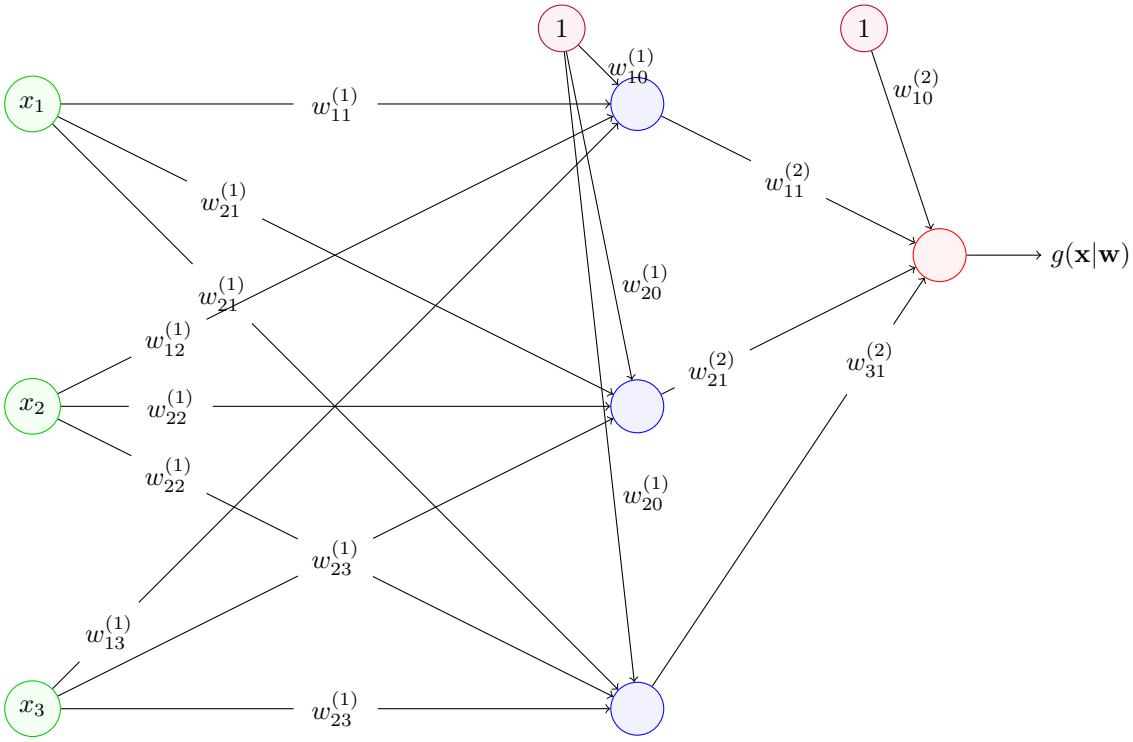


Figure 3.5: A neural network with one hidden layers.

Before going on it's useful to introduce some further notation. In particular, we want to represent the output of a layer in matrix form. The output $\mathbf{z}$ of layer l (where nodes take indices $j = 0, \ldots, J$) can be written as

$$\mathbf{z} = \mathbf{W} \cdot \mathbf{x} \tag{3.8}$$

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_J \end{bmatrix} = \begin{bmatrix} w_{01} & \ldots & w_{i1} & \ldots & w_{I1} \\ & \ddots & \vdots & \ddots & \\ w_{0j} & \ldots & w_{ij} & \ldots & w_{Ij} \\ & \ddots & \vdots & \ddots & \\ w_{0J} & \ldots & w_{iJ} & \ldots & w_{IJ} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_I \end{bmatrix} \tag{3.9}$$

Note that, in general, a network has many hidden layers, each of which can have a different number of neurons $J_1, J_2, \ldots, J_L$ (L being the number of hidden layers). In this case, we can compute the total number of neurons as

$$J_1 \cdot (I+1) + J_2 \cdot (J_1+1) + \cdots + J_L \cdot (J_{L-1}+1) \tag{3.10}$$

In this setting, the output $g_1(\mathbf{x}_n|\mathbf{w})$ of the network can be explicitly written as

$$g_1(\mathbf{x}_n|\mathbf{w}) = g_1\Big(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\Big(\sum_{i=1}^{I} w_{ji}^{(1)} \cdot x_{i,n}\Big)\Big)$$

The error can be written as

$$E(\mathbf{w}) = \sum_{n=1}^{N} \big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big)^2$$

This is where the fun begins, in fact we have to compute the derivative of the error, with respect to a specific weight $w_{a,b}$. Let's start by writing such derivative.

$$\frac{\partial E(\mathbf{w})}{\partial w_{a,b}^{(1)}} = \frac{\partial}{\partial w_{a,b}^{(1)}} \sum_{n=1}^{N} \big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big)^2 \tag{3.11}$$

Thanks to the linearity of the derivative, we can take the sum out of the error function and write

$$\frac{\partial E(\mathbf{w})}{\partial w_{a,b}^{(1)}} = \sum_{n=1}^{N} \frac{\partial}{\partial w_{a,b}^{(1)}} \big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big)^2 \tag{3.12}$$

Now we can compute the derivative of $\big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big)^2$ as

$$\frac{\partial}{\partial w_{a,b}^{(1)}} \big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big)^2 = 2\big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big) \frac{\partial}{\partial w_{a,b}^{(1)}} \Big[t_n - g_1(\mathbf{x}_n|\mathbf{w})\Big] \tag{3.13}$$

$$= 2\big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big) \Big[\frac{\partial}{\partial w_{a,b}^{(1)}} t_n - \frac{\partial}{\partial w_{a,b}^{(1)}} g_1(\mathbf{x}_n|\mathbf{w})\Big] \tag{3.14}$$

$$= 2\big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big) \cdot - \frac{\partial}{\partial w_{a,b}^{(1)}} g_1(\mathbf{x}_n|\mathbf{w}) \tag{3.15}$$

$$= -2\big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big) \cdot \frac{\partial}{\partial w_{a,b}^{(1)}} g_1(\mathbf{x}_n|\mathbf{w}) \tag{3.16}$$

$$\tag{3.17}$$

If we replace the result we have obtained in Equation 3.12 we obtain

$$\frac{\partial E(\mathbf{w})}{\partial w_{a,b}^{(1)}} = -2 \sum_{n=1}^{N} \big(t_n - g_1(\mathbf{x}_n|\mathbf{w})\big) \cdot \frac{\partial}{\partial w_{a,b}^{(1)}} g_1(\mathbf{x}_n|\mathbf{w}) \tag{3.18}$$

Now we have to focus on the derivative of function $g_1(\mathbf{x}_n|\mathbf{w})$. First things first, let's write the derivative.

$$\frac{\partial}{\partial w_{a,b}^{(1)}} g_1(\mathbf{x}_n|\mathbf{w}) = \frac{\partial}{\partial w_{a,b}^{(1)}} g_1\Big(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\Big(\sum_{i=1}^{I} w_{ji}^{(1)} \cdot x_{i,n}\Big)\Big) \tag{3.19}$$

This derivative is the derivative of a composite function. Remembering that the derivative of a composite function can be computed as

$$\frac{\partial f(g(x))}{\partial x} = f'(g(x)) \cdot g'(x)$$

we can write

$$\frac{\partial}{\partial w^{(1)}_{a,b}} g_1(\mathbf{x}_n|\mathbf{w}) = g'_1\Big(\sum_{j=0}^{J} w^{(2)}_{1j} \cdot h_j\Big(w^{(1)}_{ji} \cdot x_{i,n}\Big)\Big) \cdot \frac{\partial}{\partial w^{(1)}_{a,b}}\Big(\sum_{j=0}^{J} w^{(2)}_{1j} \cdot h_j\Big(\sum_{i=1}^{I} w^{(1)}_{ji} \cdot x_{i,n}\Big)\Big) \tag{3.20}$$

Exploiting the linearity of the derivative, we can also take the summation out of the derivative and write

$$\frac{\partial}{\partial w^{(1)}_{a,b}} g_1(\mathbf{x}_n|\mathbf{w}) = g'_1(\mathbf{x}_n|\mathbf{w}) \cdot \sum_{j=0}^{J}\Big[\frac{\partial}{\partial w^{(1)}_{a,b}}\Big(w^{(2)}_{1j} \cdot h_j\Big(\sum_{i=1}^{I} w^{(1)}_{ji} \cdot x_{i,n}\Big)\Big)\Big] \tag{3.21}$$

Now we can notice that the function

$$w^{(2)}_{1j} \cdot h_j\Big(\sum_{i=1}^{I} w^{(1)}_{ji} \cdot x_{i,n}\Big)$$

depends on $w^{(1)}_{a,b}$ only when $j = a$ because in all other cases, the weight $w_{a,b}$ can never appear. This means that the derivative is always 0, unless for $j = a$. This means that we obtain

$$\frac{\partial}{\partial w^{(1)}_{a,b}} g_1(\mathbf{x}_n|\mathbf{w}) = g'_1(\mathbf{x}_n|\mathbf{w}) \cdot \frac{\partial}{\partial w^{(1)}_{a,b}}\Big(w^{(2)}_{1a} \cdot h_j\Big(\sum_{i=1}^{I} w^{(1)}_{ai} \cdot x_{i,n}\Big)\Big) \tag{3.22}$$

Let's shift our attention to the derivative on the far right. Since we are deriving for a weight of the first level, and $w^{(2)}_{1a}$ is a weight of the second level, the latter is a constant (we are deriving for $w^{(1)}_{a,b}$). This means that we can write

$$\frac{\partial}{\partial w^{(1)}_{a,b}} g_1(\mathbf{x}_n|\mathbf{w}) = g'_1(\mathbf{x}_n|\mathbf{w}) \cdot w^{(2)}_{1,a} \cdot \frac{\partial}{\partial w^{(1)}_{a,b}} h_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) \tag{3.23}$$

We are almost at the end, we simply have to derive the last part of this equation. In particular, we want to compute

$$\frac{\partial}{\partial w^{(1)}_{a,b}} h_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) \tag{3.24}$$

Since this is a composite function we can apply the usual rule and obtain

$$\frac{\partial}{\partial w^{(1)}_{a,b}} h_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) = h'_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) \cdot \frac{\partial}{\partial w^{(1)}_{a,b}}\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) \tag{3.25}$$

At this point we can apply linearity and obtain

$$\frac{\partial}{\partial w^{(1)}_{a,b}} h_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) = h'_a\Big(\sum_{i=1}^{I} w^{(1)}_{a,i} \cdot x_{i,n}\Big) \cdot \sum_{i=1}^{I} \frac{\partial}{\partial w^{(1)}_{a,b}}\Big(w^{(1)}_{a,i} \cdot x_{i,n}\Big) \tag{3.26}$$

Now it's easy to notice that the only derivative which is not null is the one for which $i = b$, since in all other cases, the function is independent from $w_{a,b}$ since $i \neq b$. This means that the derivative of h_a can be written as

$$\frac{\partial}{\partial w_{a,b}^{(1)}} h_a\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) = h_a'\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) \cdot \frac{\partial}{\partial w_{a,b}^{(1)}} w_{a,b}^{(1)} \cdot x_{b,n} \tag{3.27}$$

Finally, we can compute the derivative of $w_{a,b}^{(1)} \cdot x_{b,n}$, which is $x_{b,n}$, and obtain

$$\frac{\partial}{\partial w_{a,b}^{(1)}} h_a\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) = h_a'\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) \cdot x_{b,n} \tag{3.28}$$

Putting the results obtained in Equations 3.18, 3.23 and 3.28 we obtain

$$\frac{\partial E(\mathbf{w})}{\partial w_{a,b}^{(1)}} = -2 \sum_{n=1}^{N} \left(t_n - g_1(\mathbf{x}_n|\mathbf{w})\right) \cdot g_1'(\mathbf{x}_n|\mathbf{w}) \cdot w_{1,a}^{(2)} \cdot h_a'\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) \cdot x_{b,n} \tag{3.29}$$

The derivative in Equation 3.29 can be used to update the value of weight $w_{a,b}^{(1)}$ using the gradient descend update formula for $w_{a,b}^{(1)}$. Let us now analyse some of the properties of the derivative we obtained:

- The derivative contains a sum over all points ($n = 1 \to N$) of the data set. This means that we need the whole data set to execute a single iteration of the gradient descend algorithm for a single weight. In other words, one iteration (or one step) of the gradient descend algorithm coincides with one epoch. The set of all samples in the data set can also be called batch, hence we say that weights are updated in batches.
- The derivative depends on g_1 and h_j we choose (and on their derivative). This means that we have to choose wisely the activation function used by each neuron.

3.2.3 Batches, stochastic gradient descent and mini-batches

The derivative used for the batch gradient descent algorithm can be written as

$$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

where

$$\frac{\partial E(\mathbf{x_n}, \mathbf{w})}{\partial w} = \left(t_n - g_1(\mathbf{x}_n|\mathbf{w})\right) \cdot g_1'(\mathbf{x}_n|\mathbf{w}) \cdot w_{1,a}^{(2)} \cdot h_a'\left(\sum_{i=1}^{I} w_{a,i}^{(1)} \cdot x_{i,n}\right) \cdot x_{b,n}$$

Since the derivative computed for a certain input $\mathbf{x}_n$ doesn't influence the derivative computed for a different input $\mathbf{x}_m$, we can compute each $\frac{\partial E(\mathbf{x_n}, \mathbf{w})}{\partial w}$ independently and then sum the results. All data can be loaded on a GPU to parallelise and accelerate the computation, however GPUs haven't enough memory for big volumes of data used to train a neural network. For this reason, we usually learn using one sample at a time. This algorithm is called **stochastic gradient descend** and uses

$$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

as update formula. In stochastic gradient descend, an iteration doesn't coincide with an epoch and in fact to end an epoch, the algorithm has to do multiple iterations, each time applying the update rule for a different input.

In between batch gradient descent and stochastic gradient descent, we find mini-batch gradient descent. In this case we use subsets of the data set to train the network during an iteration. The update formula used for mini-batch gradient descent is

$$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{1}{M} \sum_{n \in minibatch}^{M<N} \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

Note that in mini-batch gradient descent, an iteration doesn't correspond to an epoch. In particular, with a mini-batch that is $\frac{1}{S}$ the length of the data-set, an epoch corresponds to S iterations.

To sum things up:

- **Batch gradient descent** uses the whole data set (batch) during an iteration of the network. The update formula is

  $$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

  This technique has a low variance (i.e., no overfitting), however it's computationally heavy (because we need a lot of resources, given the number of points in the data set).

- **Stochastic gradient descent** uses just one data point during an iteration of the network. The update formula is

  $$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

  This technique is computationally easy to run, however it leads to overfitting models (i.e., with high variance).

- **Mini-batch gradient descent** is a technique in between batch and stochastic gradient descent. In this case we use a subset of the data set (i.e., a mini-batch) during each iteration. The update formula is

  $$\frac{\partial E(\mathbf{w})}{\partial w} = \frac{1}{M} \sum_{n \in minibatch}^{M<N} \frac{\partial E(\mathbf{x}_n, \mathbf{w})}{\partial w}$$

  This technique allows to find a good trade-off between variance and computational requirements.

3.2.4 The chain rule

We have seen a rather specific case for which we computed the derivative of the error function. Now we want to generalise what we've learned. First of all, let us underline that when we use the update formula for learning, we want to evaluate the derivative of the error for a given input point $\mathbf{x}_n$ and the weights $\mathbf{w}$, however the derivative has already been computed when designing the network. In particular, the derivative depends on the topology (connection between nodes and number of nodes for each layer) and on the activation function of each neuron.

The idea behind automating the process of learning, independently from the number of layers and neurons of the network is applying the chain rule used for derivatives.

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{3.30}$$

In particular, we want to expand the function $\frac{\partial E}{\partial w}$ deriving

- On the neurons, the neuron activation function over the previous node's weighted output (i.e., the product of the previous node's output and the weight of the incoming arc).

$$\frac{\partial h_j^{(l)}}{\partial w_{ji}^{(l)} o_i^{(l-1)}}$$

- On the arcs, the previous node's weighted output (i.e., the product of the previous neuron's output and the weight of the arc) over the weight of the arc.

$$\frac{\partial w_{ji}^{(l)} o_i^{(l-1)}}{\partial h_j^{(l-1)}}$$

An example is shown in Figure 3.6.

Since we can compute each component of the error's derivative, we can learn in two steps:

1. **Forward pass**. When we feed the network with a point $\mathbf{x}_i$, the input moves towards the output node and, for each node and arc, we can evaluate the respective derivative. Remember that we have already defined what is the mathematical expression of the derivative and we only have to evaluate its value for a certain input $\mathbf{x}_i$ and set of weights $\mathbf{w}$.

2. **Backward pass**. After reaching the output node, and having computed the output of the network, when fed with the input $\mathbf{x}_i$, we compare the output with the desired output t_i. Having computed each component of the derivative

$$\frac{\partial E}{\partial w_{ji}^{(l)}}$$

we can simply pass the network in the opposite direction (from the output neuron to the input nodes) and update the weights simply multiplying the components. In particular, for the weight on the output arc of an hidden layer we simply need the derivatives only from the output of the node to the output of the network.

The good thing about back propagation is that it can be applied to whatever architecture with whatever number of neurons.

3.3 Maximum likelihood estimation

Say the data points of a data set are generated by a Gaussian distribution with known variance σ^2. This means that the points x are independent identically distributed samples from a Gaussian.

$$x_1, x_2, \ldots, x_3 = \mathcal{N}(\mu, \sigma^2)$$

We know the variance of the distribution but not the expected value μ, hence we want to find the best Gaussian that fits the data. Namely, we want to find the Gaussian distribution which makes data more likely, i.e., for which observing the data in the data set is more likely.

Consider for instance the points in Figure 3.7 where the x axis represents the data points and the $y = p(x)$ axis the probability of a point of being x. The hypothesis centred in $x = 2$ is the

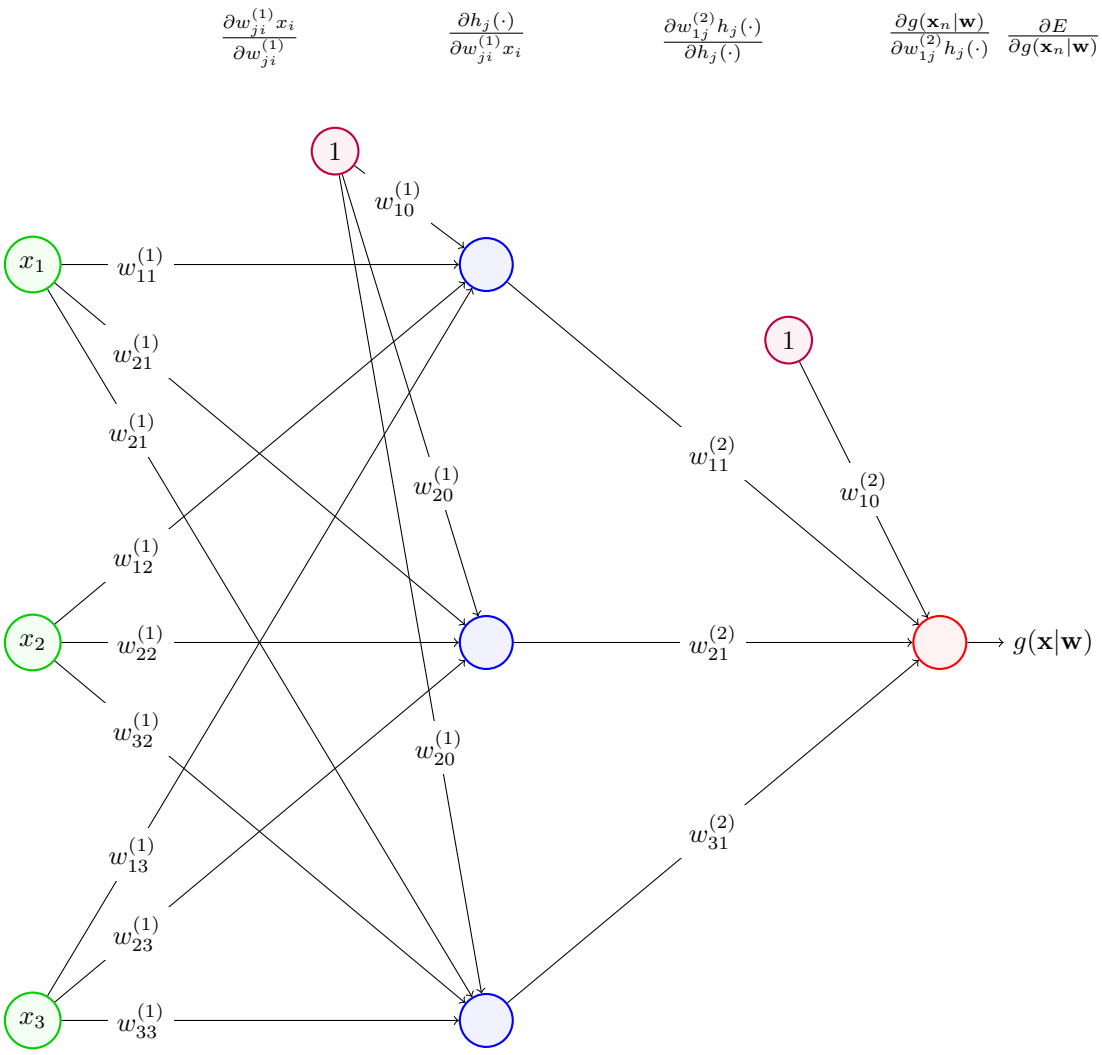


Figure 3.6: A neural network with one hidden layer.

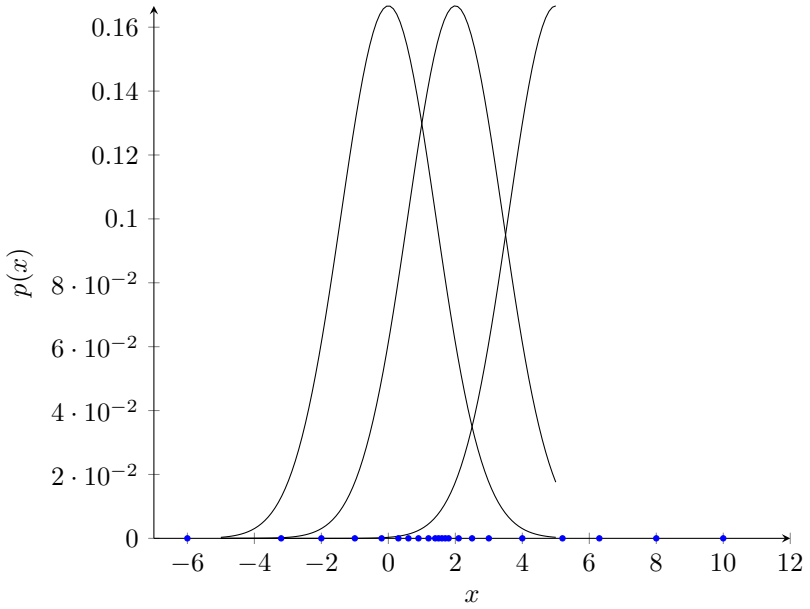


Figure 3.7: Points i.i.d Gaussian points and three tentative Gaussian distributions.

more likely to have generated the points, i.e., it's the one that makes most of the points likely to be observed.

Let us now model a maximum likelihood estimator. Let $\theta = (\theta_1, \dots, \theta_p)^T$ be a (column) vector of parameters. The likelihood of the points of the data set of being generated by the parameters θ can be written as

$$L = p(D|\theta)$$

We can also consider the logarithmic likelihood, since it will be easier to handle it later on.

$$l = \log p(D|\theta)$$

Thanks to the fact that the points in the data set are independent identically distributed, we can compute the likelihood $L(x_1, \dots, x_N|\theta)$ multiplying the single probabilities of each point in the data set x_n.

$$\begin{aligned} L(\mu) &= p(x_1, \dots, x_N|\mu, \sigma^2) \\ &= \prod_{n=1}^{N} p(x_n|\mu, \sigma^2) \\ &= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}} \end{aligned}$$

If we consider the logarithm of the likelihood, we obtain

$$\begin{aligned} l(\mu) &= \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}\right) \\ &= \sum_{n=1}^{N} log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}\right) \\ &= N\frac{1}{\sqrt{2\pi}\sigma} \sum_{n=1}^{N} log\left(e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}\right) \\ &= N\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 \end{aligned}$$

Since we want to maximise the likelihood, we have to compute the derivative of the likelihood and put it to 0. Namely, we want to compute

$$\frac{\partial l}{\partial \theta}$$

If we compute the derivative and put it to 0 we obtain

$$\mu^{MLE} = \frac{1}{N}\sum_{n=1}^{N} x_n$$

3.3.1 Maximum likelihood estimation for regression

If we were to apply maximum likelihood estimation to regression and neural networks, we can imagine that the neural network models exactly the desired function, and the desired output is computed as the sum of the network output and an error ε. Let us consider an error distributed as a Gaussian with 0 mean and variance σ^2. For this reason, we can say that the target outputs are distributed as a Gaussian centred in the output of the network $g(x_n|w)$ and with the variance σ^2 of the noise ε.

$$t_n \sim \mathcal{N}(g(x_n|w), \sigma^2)$$

When we introduced maximum likelihood we said that for $x \sim \mathcal{N}(\mu, \sigma^2)$ the probability distribution of x is

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If we replace x with t_n and μ with $g(x_n, w)$ we obtain that the probability distribution of t_n is

$$p(t_n|g(x_n, w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-g(t_n,w))^2}{2\sigma^2}}$$

3.3.2 Maximum likelihood estimation for classification

If we want to use maximum likelihood for classification, we have to use a Bernoulli distribution instead a Gaussian distribution, hence the desired output is distributed as

$$t_n \sim Be(g(x_n|w))$$

If we remember the probability distribution of a Bernoulli, we can write

$$p(t|g(x_n|w)) = g(x_n|w)^t \cdot (1 - g(x_n|w))^{1-t} \tag{3.31}$$

As for the general case, we can compute the likelihood of the data set as the product of the probability distribution of each data point, since the points are i.i.d. What we get is

$$L(w) = \prod_{n=1}^{N} p(t_n|g(x_n|w)) = \prod_{n=1}^{N} g(x_n|w)^t \cdot (1 - g(x_n|w))^{1-t_n}$$

If we consider the logarithmic likelihood we obtain

$$L(w) = \prod_{n=1}^{N} p(t_n|g(x_n|w)) = \sum_{n=1}^{N} t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

This loss is called binary **cross entropy**. In general, the cross entropy error is

$$E(w) = -\sum_{n=1}^{N} \sum_{c=1}^{C} t_n^c \log g(\mathbf{x}_n|w)$$

where c is the number of classes.

3.3.3 Errors

Until now we have described two different error functions:

- The **sum of squared errors**

$$E(w) = \sum_{n=1}^{N} \big(t_n - g(\mathbf{x}, w)\big)^2$$

- The binary **cross entropy**

$$E(w) = -\sum_{n=1}^{N} t_n \log \big(g(\mathbf{x}_n|w)\big) + (1 - t) \log \big(1 - g(\mathbf{x}_n|w)\big)$$

3.4 Training and overfitting

Overfitting is a very well known issue in machine learning and, if it can normally cause some problems, it is even worst in neural networks. Let's try to explain what overfitting is. Say we want to approximate some function f. Thanks to the universal approximation theorem (Theorem 3.1), we know that we can always build a neural network (even with a single layer in case of regression), that approximates f. However, even if we can theoretically learn such function,

- We might not find the correct weights w.
- We don't know how many neurons are required.
- We might learn a function which is good only at predicting the points we are using to train it.

The last problem we have shown is overfitting. Overfitting is tightly bounded to model complexity. More precisely, simple models might be too simple to capture and model the data set (this problem is called underfitting), hence they can generate an high error. To reduce such error we can consider more complex models with non linear features. We can increase the complexity of the model (especially with neural networks) so much that the model can classify every point in the training set with 0 error. This is a problem because when we have to classify a new point not in the data set, we might do an error which is bigger than the one of the linear (i.e., simple) model. Overfitting is given by the fact that the data set is generated by f plus some noise, and with overfitting we model also the noise and not f. An example of underfitting, overfitting and reasonable model is shown in Figure 3.8.

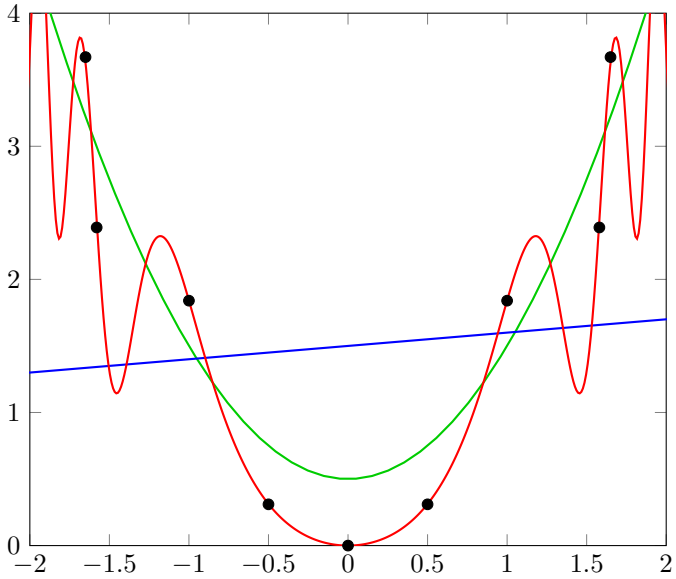


Figure 3.8: An example of overfitting, underfitting and a reasonable model for a data set generated by a quadratic function.

Overfitting usually happens in all models in which we do induction, namely when we try to build general rules from examples. To solve the problem of overfitting and still use induction, we have to constraint the induction process to the right models and hypothesis space. The goal now is to find out how to choose such models, which is a key aspect to have non-overfitting neural networks. The choice of the model sometimes goes under the name of **inductive hypothesis**, which is the the hypothesis we make so that the induction process can work.

3.4.1 Measuring generalisation

Since we need to understand if a model is generalising correctly or it's learning the data set by heart, we need to find a way to measure how good a model is at generalising. The methods we will analyse are valid in general for machine learning, hence they apply also to neural networks.

Training error

Something that should never be used for evaluating the generalisation power of a model is the training error, that is the error (usually the square error) computed on the training data itself. This is because the more we increase the complexity of the model, the more the error is reduced and we learn the data set by heart (which is exactly what we wanted to avoid).

Test error

Since we don't want to evaluate a model on the very dataset used to train it, we need a new set of data points which are used only for evaluating the model (but not for training it). Effectively we can divide data in

- A training-set, used for training.
- A test-set, used for evaluating the model.

Not only the two sets should be separated, but training and testing should also be performed by different people so that no bias is introduced in the testing procedure. In fact, the developers know how the model works, hence they could perform the tests in a way that the model might look better than it actually is. In some cases, we don't have two different datasets, thus we have to split the data set in training and test-set before working on the model. After having trained and fine-tuned the model with the training set, we can go back and test it with the test-set we initially left behind. The important thing is that the model should never be trained with the test-set. Usually, the training set is much bigger than the test set (80 to 20 or 90 to 10) because training increases the performance of the model, hence it's better to spend most of the resources on training. Note that in classification we have to preserve the class distribution when we sample from the dataset to build the test set. In particular, we have to apply stratified sampling. Say we have three classes C_1, C_2 and C_3 and we want to split the data 80 to 20 %. What stratified sampling does is to split every class in 80-20 % and then take the 80 % of each class and put it in the training set, while the remaining 20 % is put in the test set. Practically, if C_1 has 1000 samples, we put 800 in the training set and 200 in the test set. If C_2 has 100 samples, we put 80 in the training set and 20 in the test set and if C_3 has only 10 samples we put 8 of them in the training set and the remaining 2 in the test set. With this technique we are sure to have split the dataset in 80-20 but without changing the class distribution (i.e., the ratio of the classes in the training set is the same as the one in the test set).

In some cases the data set is so small that we can't remove some data and use it only for testing. In such cases, we can use techniques, called bootstrapping techniques, that remove some data with replacements. This means that part of the training points is used also for testing. In our setting however, we are required to have a lot of data, hence this situation will never happen.

Note that testing is also called **model evaluation** or **model assessment**.

Model selection

Between training and assessment we have to take decisions regrading the shape of the neural network as the number of layers and neuron to use or the activation function to assign to each neuron. We can't use neither the training set nor the test set to take such decisions, otherwise the model will be biased towards the training or the test set, respectively. This means that we need another data set, which is called **validation set**, which contains the data used to do model selection, i.e., to define the characteristics of the model (e.g., activation function, number of layers, number of neurons). With the introduction of the validation set, we can split the dataset in:

- **Test set**. The training set is obtained splitting the data set in training set and test set, as soon as we receive the dataset. The training set isn't touched until the very end and it's used only once.
- **Validation set**. The validation set is obtained, after creating the test set, splitting the training set in validation and training set. Usually the former is smaller than the former since we want to use most of the resources to train the network. The validation set is used to fine-tune the model, i.e., to do model selection, or, for neural networks, hyperparameter tuning.
- **Training set**. The training set is what remains after having created test and validation set.

Note that in this process we can cycle as many times as we want between training and validation until we have found a model that we really like. Only at the end of the process we can use the test set and evaluate our work. Note that if the model doesn't perform well with respect to the test set, we shall never go back to validation and training. Usually, we expect the test error to be a bit worst than the validation error since we have tuned the model using the validation set, however if we correctly executed the training-validation phase, we can be happy with the resulting model. One might ask what happens if the model performs poorly on the test test. In such cases, the only thing we can do is to throw everything away and start back the process (including the dataset partition).

Cross validation

Cross-validation is a nice technique that allows to do model selection using the training data. Depending on the quantity of data available, we can perform different types of cross-validation:

- **Hold out cross-validation**. Hold out cross-validation consists in taking some data from the training set and use it to estimate the error of the model trained with the remaining data points. Basically, this is the technique we have seen before. However, depending on how we have split training and test set, we might get very different results.
- **Leave-One-Out cross-validation**. In Leave-One-Out cross validation we take one sample out of the training set and we use the remaining points for training. After training we test the model on the sample we have extracted. The procedure is repeated with every sample of the training set and at the end we average the error over all the samples. This technique allows us to train and estimate the error on all the data. The main problem of LOO cross-validation is that it requires us to train a model as many times as the number of samples.
- **k-fold cross-validation**. This technique sit in between hold-out and LOO cross-validation, in fact instead of taking only one sample, we divide the training set in k folds (i.e., subsets) and we use one fold for estimating the error, and the others for training the model. The procedure is repeated for every fold (i.e., one fold at a time is used only for estimating the error) and the error is estimated averaging the errors of the single iterations. Note that k-fold we can also be used for testing.

Usually in neural networks we use, since we have a lot of data, we hold out both for model selection and testing.

Say that, after the model selection phase, we have chosen a particular model. When doing model selection, we have trained the same model with a lot of different training sets, hence the parameters of the model will be different one from the other. To decide which model to use (i.e., which set of parameters for a certain model to use) we can:

- Keep all the models and use a majority voting scheme (for classification) or average the output (for regression).
- Train the model again, this time using the whole training set, without leaving any point out for the validation set.

3.5 Techniques for preventing overfitting

Now that we know the procedure used to measure the performance of a machine learning problem, we can see practically how we can prevent overfitting in neural networks.

3.5.1 Early stopping

Early stopping is based on the fact that, for big neural networks, after training for a long time, we might eventually get to a null training error. This is however a bad sign since it means that we are learning the data and not the function we want to approximate. To see this in practice we can perform hold-out cross-validation during training to verify that, the more we train:

- The more the train error decreases.
- The more the error on the validation set increases, after some initial time in which it decreases.

This means that we can iterate the following process

1. Update the weights of the neural network using the training data.
2. Compute the error on the validation set.

until the validation error starts increasing. That's why this technique is called early stop, because we stop training before executing too many epochs. Early stopping tries to estimate the generalisation error online by using the validation set.

An implementation of early stopping usually waits for p epochs and, if during this time the validation error doesn't decrease, then the training stops. The parameter p is usually called **patience**.

Note that this tool allows us to build networks which are bigger than what we really need since it allows us to block overfitting, which usually happens in big networks (since they are more complex models).

Early stopping for model selection

Early stopping is a technique for preventing overfitting, however it can also be used to decide the number of neurons to use. The idea is to evaluate models with an increasing number of neurons and compute the error using the early stopping technique (i.e., we compute the error when early stopping decides to stop). The model selection phase ends as soon as we see that increasing the number of neurons also increases the error and we take the last model for which the error wasn't increasing yet. Basically, we repeat early stopping on models with increasing number of neurons, until the error starts increasing. With this procedure we can also test different activation functions. Basically, we can test different combinations and then we select the best one. This procedure is called **hyperparametes tuning** since we define the hyperparameters of the model, which will define how many parameters the network will have. Namely, hyperparameters are parameters that regulate the structure of the model.

Early stopping is a very useful technique and it should be used in most cases since it has very few drawbacks, the main being that we have to hold out some data.

3.5.2 Weight decay

Weight decay is a technique to reduce overfitting by reducing a model's freedom. Namely, weight decay reduces the domain of the weights (i.e., the values that the weights can take). This is a different approach with respect to early stopping because in the latter case we reduce the models we can use, whilst with weight decay we reduce the freedom of the models (but we can still use any model). In particular, we can reduce the freedom of a model by using a Bayesian approach. Let's start by saying what's the difference between a Bayesian and a frequentist approach.

- Following the frequentist approach, we maximise the data likelihood, i.e., the probability of observing the data D given certain weights w.

$$w_{optimal} = \arg\max_w P(D|w)$$

- Following the Bayesian approach, we build some prior distribution over the weights. This means that, even before training the data, we model the weights over what we know about the problem. Namely, we make some assumption on the distribution $p(w)$, called a-priori, of the parameters (i.e., on the value of the parameters with some variance). The a-priori distribution is then updated using the Bayes rules to obtain the distribution of the parameters given the data.

$$\hat{w}_{optimal} = \arg\max_w P(w|D) = \arg\max_w P(D|w) \cdot p(w)$$

  In this case, the samples are used to update our assumptions over the a-priori distribution of the parameters.

Using Bayesian statistics, we can fix some initial values for the parameters using the a-priori distribution and allow learning to change them only by a bit, effectively limiting the freedom of the model. The difficult part of this approach is finding the right prior. In particular, it has been observed that small weights can improve the generalisation capabilities of neural networks, hence we can use a prior distributed as a Gaussian with zero mean an fixed variance σ_w^2

$$P(w) \sim \mathcal{N}(0, \sigma_w^2)$$

Assigning this distribution we are saying that a parameter w should be 0, with some uncertainty based on the variance σ_w^2.

If we replace the Gaussian distribution in the formula for computing the parameters (remembering that the data is also distributed as a Gaussian) we obtain

$$\begin{aligned}\hat{w} &= \arg\max_w P(w|D) = \arg\max_w P(D|w) \cdot p(w) \\ &= \arg\max_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\left(t_n - g(x_n|w)\right)^2}{\sigma^2}} \cdot \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w^2} e^{-\frac{w_q^2}{\sigma_m^2}}\end{aligned}$$

As always we can apply the logarithm to obtain

$$\begin{aligned}\hat{w} &= \arg\max_w P(w|D) = \arg\max_w P(D|w) \cdot p(w) \\ &= \arg\max_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\left(t_n - g(x_n|w)\right)^2}{\sigma^2}} \cdot \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w^2} e^{-\frac{w_q^2}{\sigma_m^2}} \\ &= \arg\min_w \sum_{n=1}^{N} \frac{\left(t_n - g(x_n|w)\right)^2}{2\sigma^2} + \sum q = 1^Q \frac{(w_q)^2}{2\sigma_w^2}\end{aligned}$$

Now we can multiply everything by σ^2 and write $\gamma = \frac{\sigma^2}{\sigma_w^2}$ and obtain

$$\begin{aligned}
\hat{w} &= \arg\max_w P(w|D) = \arg\max_w P(D|w) \cdot p(w) \\
&= \arg\max_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\left(t_n - g(x_n|w)\right)^2}{\sigma^2}} \cdot \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w^2} e^{-\frac{w_q^2}{\sigma_m^2}} \\
&= \arg\min_w \sum_{n=1}^{N} \frac{\left(t_n - g(x_n|w)\right)^2}{2\sigma^2} + \sum q = 1^Q \frac{(w_q)^2}{2\sigma_w^2} \\
&= \arg\min_w \sum_{n=1}^{N} \left(t_n - g(x_n|w)\right)^2 + \gamma \sum q = 1^Q (w_q)^2
\end{aligned}$$

We can interpret the formula we have just written as the minimisation of a loss function made of two parts:

- The fitting part
$$\sum_{n=1}^{N} \left(t_n - g(x_n|w)\right)^2$$
which computes the error done by the predictor.
- A regularisation part
$$\sum q = 1^Q (w_q)^2$$
which increases when the parameters have higher values.

The parameter γ can be used to tune how much importance we want to give to the second part and in particular:

- If we put $\gamma = 0$ we get the old model which can suffer from overfitting.
- If we put $\gamma = \infty$ we get a model which doesn't care about the error with respect to the training points, hence an underfitting model.

Tuning gamma, we can build from very overfitting model to very underfitting ones. To find the best value of γ we can, as always use cross-validation (but not early stopping because with the right γ the model wont overfit). Usually, we expect the error to decrease as we increase γ until we reach a point in which it starts back increasing. This is where we should stop and pick γ. After finding the optimal γ^* we can train back the model using γ^* and without early stopping since we won't overfit. This is also a good place where to use k-fold cross validation.

This type of regression is called **ridge regression** and it's typically used when don't have a lot of data.

3.5.3 Dropout

Dropout is a stochastic technique for performing regularisation. The idea behind this technique is to train the weights without them relying on the presence of the other weights. In big neural networks sometimes happens that when a weight becomes really high, some other weights are reduced to compensate for that. This leads to overfitting and complexity in training. Moreover, the learned

function isn't very smooth. To solve this problem, we can randomly switch off some neurons, so that the network is forced to learn in such a way that weights can't rely on other weights.

In practice, we build a mask

$$m^{(l)} = [m_1^{(l)}, \dots, m_J^{(l)}]$$

for each layer where each $m_j^{(l)}$ is a binary value (1 if we want to consider the output of neuron j, 0 otherwise). At each iteration, we assign a probability $p_j^{(l)}$ of being active to each neuron of the network. The probability is used to compute the value of the mask as

$$m_j^l = Be(p_j^{(l)})$$

where Be is a Bernoulli distribution. Each element of the mask, whose value is either 1 or 0, is then multiplied by the input of the corresponding neuron. If the mask has value 1 the neuron is activated since its input is not 0, otherwise its input is blocked (being 0).

$$h^{(l)}\big(\mathbf{W}^{(l)} h^{(l-1)} \cdot m^{(l)}\big)$$

After updating the weights, the probabilities, hence the masks, are changed so that different neurons are deactivated.

It has been proved that dropout might improve generalisation, still it doesn't provide any assurance.

3.6 Training tricks for real word usage

3.6.1 Model evaluation

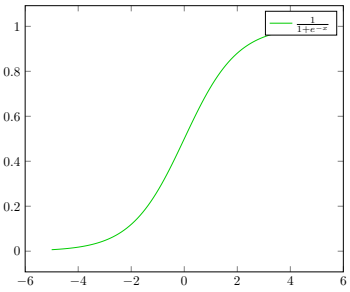
We've seen many ways to assess the quality of a model before testing it (which is the last step from which we can't come back). In particular, we've analysed k-fold cross validation and hold-out. Between these two, the latter is much more used than the former, which can instead be used, once the model architecture has been fixed, to do some hyperparameter tuning. This is because hold-out requires way less time for training than k-fold. Moreover, neural networks are based on the fact that we have a lot of data, hence having a fixed hold-out set for training should be enough to train the network and model the probability distribution of the input.

3.6.2 Activation functions

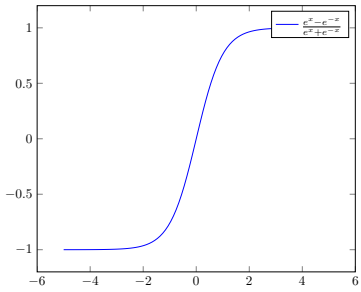
When choosing an activation function for neurons, we usually turn to the sigmoid function or the hyperbolic tangent. These functions are represented in Figure 3.9. If we plot their derivatives, obtaining what we see in Figure 3.10, we see that they go to 0 quite fast when increasing the value of the input. Since the activation function is evaluated over a weighted sum, with weights w, of the input, then the gradient goes to 0 if the weights are too big. This is a problem because it means that the gradient, which is computed over w, goes to 0 when the weights have high values. This means that a network with big weights will not learn since the gradient is 0 but it requires the gradient to move in the solutions space.

One might think that we only have to initialise the weights close to 0 to avoid this problem. This is in fact true, but, recalling that we have to compute

$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = -2 \sum_n^N (t_n - g_1(x_n, w)) \cdot g_1'(x_n, w) \cdot w_{1j}^{(2)} \cdot h_j'\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{in} \right) \cdot x_i$$

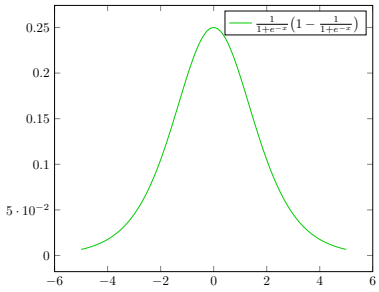


(a) The sigmoid activation function.

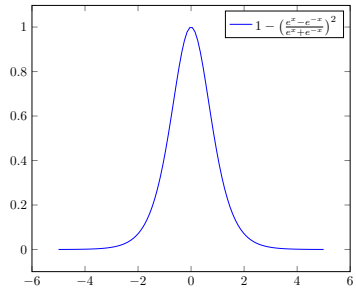


(b) The hyperbolic tangent activation function.

Figure 3.9: Sigmoid and hyperbolic tangent activation functions.



(a) The derivative of the sigmoid activation function.



(b) The derivative of the hyperbolic tangent activation function.

Figure 3.10: Sigmoid and hyperbolic tangent activation functions' derivatives.

the problem is that the weights have to be multiplied by the derivative of the activation function. If we consider the sigmoid activation function, the maximum value it can get is 0.25. If we multiply this value by weights which are smaller than 1 (since we want them close to 0), we converge to a gradient which is also close to 0. This phenomenon is called **vanishing gradient**. The same applies to hyperbolic tangent since at best we have derivative equal to 1, but this happens only when weights have value 0 (which doesn't make sense since weights have to be non-null to propagate a neuron's output).

Since the vanishing gradient problem comes from derivatives smaller than 1, we might be tempted to use activation functions with gradient always bigger than 1. This is also a problem, especially in deep networks, since we will get a very big gradient which will make the solution oscillate around the optimal solution.

The solution is to have a gradient always equal to 1. We know that the linear activation function has derivative always equal to 1, but we can't use it since we need some non-linearity (otherwise we don't have the result of Theorem 3.1). The solution is to use the Rectified Linear Unit, or *ReLU* function (Figure 3.11)

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This function doesn't only solve the problem of vanishing and exploding gradient, but it also has some very good properties, the most important being:

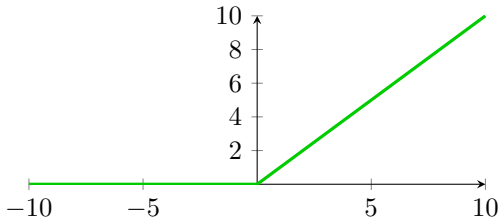


Figure 3.11: The ReLU activation function.

- It is way faster in convergence.
- It's faster and easier to compute.
- It has a sparse activation, i.e., only the neurons for which the activation value is greater than 0 will have non-zero output.

As always, there is no free lunch, hence the ReLU has some drawbacks, too. In particular:

- It's not differentiable in the origin. This problem is actually not very concerning since it's not differentiable in one point only and, in case we have to compute the derivative in the origin, we can choose the value we prefer (ideally, 1).
- The output is not zero-centred. Namely, the average of the activation functions' outputs of one layers is not 0 (all values are bigger than 0). This might introduce some inefficiencies but it still isn't a huge problem.
- It's unbounded, hence some weights might grow to a very large number hence making the model overfit.
- If a neuron has a negative value, its output will be 0, but also its gradient will be 0, hence the neuron will be switched off and it will never be on again. This problem is called **dying neuron** and usually shows up with high learning rates.

To solve the last problem, we can use a modified version of the $ReLU$ function, called $LeakyReLU$

$$LeakyReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \varepsilon x & \text{otherwise} \end{cases}$$

where ε is a small number (e.g., $\varepsilon = 0.01$). Note that the $LeakyReLU$ doesn't solve the vanishing gradient problem but it's a good trade-off between mitigating the vanishing gradient (since we still have derivative equal to 1 for $x \geq 0$) and the dying neuron problems (since we don't have 0 derivative for $x < 0$). Another possible solution is the ELU activation function

$$ELU(s) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

where α is a parameter that defines the slope of the leaky part and can be tuned.

Finally, another option is the **swish** function.

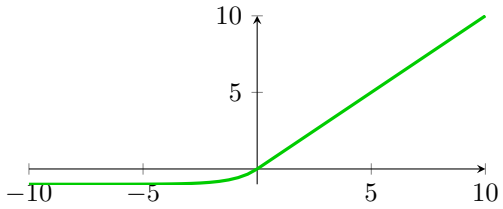


Figure 3.12: The ELU activation function.

3.6.3 Weight initialisation

The result of gradient descend is affected by the initial values of the weights. We've said that weights can be initialised at random but we can also decide how to initialise them. Some bad solutions are:

- Initialise all weights to **0**. This is a very bad idea since gradients will be 0, hence the model will not learn.
- Initialise all weights to **big values**. This is a very bad idea because the gradient explodes and the model takes a long time to converge.
- Initialise weights with small values, i.e.,

$$w \sim \mathcal{N}(0, \sigma^2 = 0.01)$$

  In general, this idea is fine, however it doesn't work with big neural networks.

If we really want to properly initialise weights, we can use:

- **Xavier** initialisation.
- **He** initialisation.

Xavier initialisation

The idea behind Xavier initialisation is that the input is propagated through the network and it gets multiplied by the weights. For this reason, if the weights are too big, the output is also big. Let us start (just like Xavier did) considering a linear activation function. This means that each output h_j of one layer is the weighted sum the inputs x_i, namely

$$h_j = w_{j1}x_1 + \cdots + w_{ji}x_i + \cdots + w_{jI}x_I$$

Let us now understand how much each element of this sum spreads from the initialisation, namely, let us compute the variance. What we get is

$$\text{var}(w_{ji}x_i) = E[x_i]^2\text{var}(w_{ji}) + E[w_{ji}]^2\text{var}(x_i) + \text{var}(x_i)\text{var}(w_{ji})$$

If we assume that the the weights and the input has been centred in 0 (which is safe to assume and it's also a common practice), then we get

$$\text{var}(w_{ji}x_i) = \text{var}(x_i)\text{var}(w_{ji})$$

This means that, assuming w_{ji} and x_i to be independent identically distributed, we can compute the variance of the input as

$$\text{var}(h_j) = \text{var}(w_{j1}x_1) + \cdots + \text{var}(w_{ji}x_i) + \cdots + \text{var}(w_{jI}x_I) = I \cdot \text{var}(x_i)\text{var}(w_{ji})$$

This means that:

- If the variance of the weights is greater than 1, the spread of the neurons increases.
- If the variance of the weights is smaller than 1, the spread of the neurons decreases.

Moreover, if we have an input with fixed variance 1, it will be amplified by a factor $I \cdot \text{var}(w_{ji})$. If we want to initialise the neurons with stable values, we can enforce the input amplification factor to be equal to 1, namely,

$$I \cdot \text{var}(w_{ji}) = 1$$

to obtain

$$w \sim \mathcal{N}\left(0, \frac{1}{I}\right) \sim \mathcal{N}\left(0, \frac{1}{n_{input}}\right)$$

Performing a similar reasoning, we can enforce

$$n_{out}\text{var}(w_{ji}) = 1$$

and, mixing this result with Xavier's we get

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

He initialisation

He applied the same reasoning used by Xavier but using *ReLU* functions instead of linear activations. As a result, he proposed

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

3.6.4 Batch normalisation

As we have seen, we can normalise input data so that it's centred around 0 with unit variance. The same can be done, using batch normalisation, with the inputs of activation functions. In particular, we can add a layer, called **batch normalisation layer**, between the fully connected layer and the activation functions to normalise the linear combination of the inputs. Note that normalisation is done before applying the activation functions, hence it prepares the input to pass to the activations. This normalisation has to be trained at training time since it's at training time that we want to optimise the fully connected layer such that it behaves properly. Formally, the batch normalisation algorithm is shown in Algorithm 1. In words, the algorithm:

1. Takes the input, i.e., a batch, $\mathcal{B}$ of a layer (i.e., the weighted sum of the previous layer's activations).
2. Computes the mean of the values x_i in the batch.
3. Computes the variance of the values x_i in the batch.

4. Normalises the values x_i to obtain $\hat{x}_i$. Note that this operation is linear hence it can be back-propagated.
5. Computes the outputs by scaling the normalised x_i (i.e., $\hat{x}_i$) by a value γ and by shifting them by β.

The parameters

$$\gamma^{(k)} = \sqrt{\text{var}(x^{(k)})} \tag{3.32}$$

and

$$\beta^{(k)} = E[x^{(k)}]$$

have to be trained to learn the best values to normalise the dataset (since for each batch, we are normalising only the batch itself). This parameters can then be used at runtime, to normalise any input the network has to classify. Note that building a batch normalisation without parameters is wrong since we have to re-scale vectors also during inference.

Algorithm 1 The batch normalisation algorithm.

procedure BATCH-NORMALISATION($\mathcal{B} = x_1, \ldots, x_m$, γ, β)
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}}$
$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$
end procedure

Batch normalisation has proven it has many advantages, the most important being:

- It improves gradient flow through the network.
- It allows using higher learning rates (which translates in faster learning).
- It reduces the strong dependence on weights initialisation.
- It acts as a form of regularisation slightly reducing the need for dropout.

3.6.5 Gradient descend with momentum

When using gradient descend, we can increase training speed using momentum. The idea is to also use the previous value of the gradient to compute the current gradient. Momentum can be applied after computing the gradient or before:

- In the first case, we use the gradient to move and then we use the previous gradient to add some momentum.
- In the second case, called Nesterov accelerated gradient, we first apply momentum to put ourselves in a better position and then we apply the gradient.

$$w^{k+\frac{1}{2}} = w^k - \alpha \frac{\partial E(w)}{\partial w}\bigg|_{w^{k-1}}$$

$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^{k+\frac{1}{2}}}$$

3.6.6 Adaptive learning rates

Stochastic gradient descend requires a learning rate to move in the parameters space. Some gradient descend algorithms allow to change dynamically the learning rate and the gradient (some adapt the direction of the gradient, some the gradient itself) during training. Some examples are:

- Resilient propagation.
- Adaptive gradient.
- RMSprop.
- AdaDelta.
- Adam.

Part III

Computer vision

Chapter 4

Introduction

4.1 The image recognition problem

Computer vision is a multidisciplinary research field that deals with how computer can be made able to gain an high level understanding from digital images or videos. One example of computer vision problem is object detection, i.e., the problem of, given an image (or a frame in a video), recognising the objects in an image and classify them. In practice, a semaphore camera can tell apart cars from lorries and bikes. Another practical example is pose estimation, that is estimating the position of a human body in space.

Initially, the image recognition problem has been approached from a classical computing perspective (i.e., using mathematics and statistics) but, nowadays we use machine learning and especially deep neural networks to solve it. Note that this doesn't mean that geometry based approaches (i.e., more classical ones) have to be completely discarded, in fact we can still use them for some specific problems or to improve machine learning models.

4.1.1 Images

Since we will have to work on images, it makes sense to analyse how they are represented and how we can manipulate them with a programming language. An image is a matrix or, in case of coloured images, a set of three matrices each of which dedicated to one colour (the colours being red, green and blue). We also say that a coloured image has three channels. In formulas, an image I is an element of $\mathbb{R}^{R\times C\times 3}$, i.e., its made of three bi-dimensional matrices R, G and B each of which is an element of $\mathbb{R}^{R\times C}$. Each pixel (i.e., a cell) of the bi-dimensional matrix is a number that goes from 0 to 255 and represents the colour's intensity in that pixel of the whole image. For instance, a pixel in $B \in \mathbb{R}^{R\times C}$ tells us the intensity of the blue in the same pixel of the whole image $I \in \mathbb{R}^{R\times C\times 3}$. This means that each pixel of an image can be represented as a triplet

$$(r, g, b)$$

where r, g and b are the intensities of the red, green and blue components. These intensities can be found also in the matrices R, G and B, respectively. Since each component has a range from 0 to 255, it can be represented in 1 byte, i.e., 8 bits. Having three components we can represent $2^{8\cdot 3} = 16,777,216$ (around 16 million) colours.

We can read an image in Python using the code snippet in Listing 4.1. Note that the image loaded in the program is usually bigger than the one saved on disk since some formats, like `jpeg`, compress the image.

```
from skimage.io import imread

# Read the image
I = imread('image_name.jpg')

# Extract the colour channels
R = I[:, :, 0]
G = I[:, :, 1]
B = I[:, :, 2]
```

Listing 4.1: Extracting colours from an image in Python.

Higher dimensional images

Images can be represented using more than three channels. Some examples are RMIs or geographical scans. In these cases, usually the resolution of the image is higher, which means that each pixel in a matrix can have an higher range of values (e.g., from 0 to 65535, using 2 bytes).

4.1.2 Videos

A video is a collection of images, called frames, hence it can be represented as a collection of tridimensional matrices, i.e., a matrix in four dimensions. A video V is therefore an element in $\mathbb{R}^{R \times C \times 3 \times T}$ where T is the number of frames in the image and represents the time elapsed.

4.2 Image transformation

Image transformation is an important task related to image classification problems and it's fundamental in deep learning methods used for image classification.

4.2.1 Local spatial transformation

Let's introduce the concept of local spatial transformation.

Definition 4.1 (Local spatial transformation)**.** *Given an image I with R rows and C columns, a local spatial transformation is an operation*

$$G(r, c) = T_U[I](r, c)$$

where

- *I is the **input image**, with a single colour channel, to be transformed.*
- *G is the **output** image.*
- *$T_U : \mathbb{R} \to \mathbb{R}$ is a **function transforming the image**.*
- *U is a **neighbourhood**, which identifies a region of the image which will concur in the output definition.*

In practice, taken a pixel in position (r, c) of the input image I, the output pixel in coordinates (r, c) is computed using a region U of the input image I. This means that, for each pixel in the input image, we have to define a neighbourhood U around it. The output of a pixel (r, c) is computed only using the values in its neighbourhood U. Figure 4.1 tries to depict this idea.

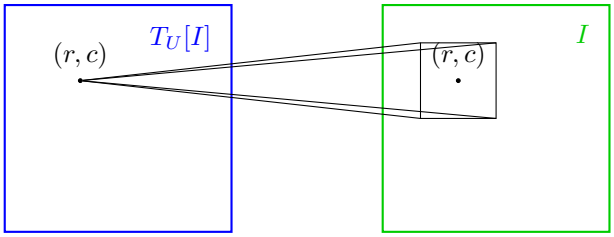


Figure 4.1: A local spatial transformation.

Typically, the input and output image should have the same size, however if we consider a region $U \in \mathbb{R}^{H \times W}$ with $H, W > 1$, it can't be used on border pixels.

Correlation function

The easiest way to define the transformation T_U is by using the average of the pixels in the neighbourhood U. More in general, we can consider a liner combination of the pixels in the neighbourhood. In particular, we can define the output pixel in position (r, c) obtained from transformation T_U as

$$T_U[I](r, c) = \sum_{(u,v) \in U} w(u, v) \cdot I(r + u, c + v) \tag{4.1}$$

where

- The neighbourhood U is described as a set of offsets. For instance, if the neighbourhood considers a pixel in each direction from (r, c), we get

$$U = \{(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1), (0, 0)\}$$

- $w(u, v)$ is a weight. Note that we have a weight for each element of the neighbourhood. The matrix of weights can be seen as an image itself and it's called **filter** or **kernel**. Note that the filter is independent from the input pixel and it's the same for each pixel.

This transformation is typically called **correlation**, however, in machine learning, we call it **convolution**. Convolution and correlation are the same operation, however the former multiplies the filter for $I(r - u, c - v)$, instead of $I(r + u, c + v)$. The correlation between a filter w (of size $(2L + 1) \times (2L + 1)$) and an image I can also be written as

$$(I \otimes w)(r, c) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} w(u, v) \cdot I(r + u, c + v) \tag{4.2}$$

The same operation can be performed in Python with the following piece of code.

```
from scipy import signal

C = signal.correlation(I, w)
```

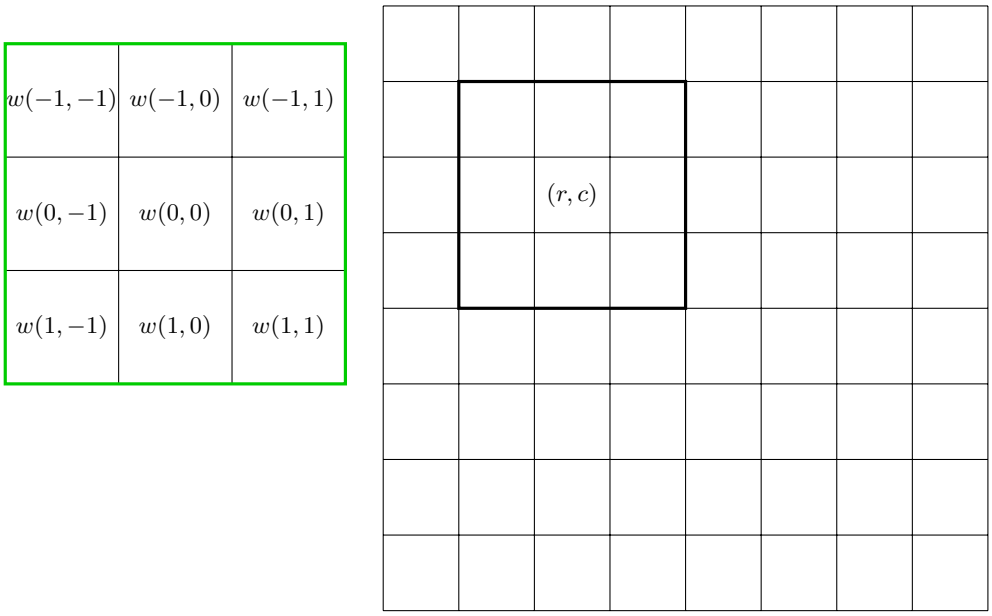


Figure 4.2: Weights used in a linear transformation.

Let us now understand what is the meaning of this transformation. To understand the meaning of correlation we can take a binary image (i.e., only the colours black and white can be displayed) and a binary filter. Since a pixel can be either 0 (if black) or 1 (if white), then convolution will return 0 whenever a black pixel is multiplied by a white pixel. If the filter is part of the image and we compute the correlation between image and filter we obtain a maximum in the region of the image where the filter is. This happens because, when computing correlation, the filter is moved over the input image and each pixel is multiplied by the pixel above it in the image. This means that when the filter perfectly overlaps with the area of the image from where it has been taken the convolution value is maximum since every white pixel is aligned with every other white pixel. Namely, correlation tells us where the filter (which in this example is a part of the image) is in the image, hence it's the most basic tool for doing template matching (i.e., finding an image, which is the filter, also called template, in a bigger image). This happens if we are trying to match a white pattern (e.g., a white text on a black background). If we swap the pixels' value (i.e., we match a black text on white background) we obtain a completely different result since the correlation isn't anymore the sum of pixels that match our template (i.e., the text in the filter). This problem is generated by the fact that pixels can take value 0, hence to solve it (in real images with more colours than simply black and white) we should always use normalisation when doing template matching.

4.3 Feed images to a neural networks

Say we want to build an image classifier. As for now, we know how to analyse and transform an image and how to build and train a neural network, but we don't know to to feed a neural network with an image. The easiest thing we can do is to flatten the pixels of an image (i.e., build a vector of pixel from a matrix of pixels) and feed the input layer of the neural network with the vector containing the pixels. This solution is very simple but impractical since, even for a small picture, it

would require a huge amount of input neurons.

To see why this approach is not suggested, let us consider a neural network with no hidden layer and let us fed such model with an image. This means that we can compute the output s_i, also called score, of the output neuron i as

$$s_i = \sum_{j=1}^{d} w_{i,j} x_j + b_i$$

where d is the number of input neurons. Note that in this case we aren't using the softmax function to normalise the scores, in fact, we can choose the class with the highest score.

$$\hat{y}_n = \arg\max s_i$$

If we consider images of 32×32 RGB pixels and 10 possible output class, we have that

- The input layer has $32 \cdot 32 \cdot 3 = 3072$ neurons.
- The output layer has 10 neurons.
- The network has $3072 \cdot 10 + 10 = 30730$ parameters.

If we add a single hidden layer with half the neurons of the input layer (i.e., 1536 neurons) we get a network with

- $3072 \cdot 1536 + 1536 = 4720128$ parameters between input and hidden layer.
- $1536 \cdot 10 + 10 = 15370$ parameters between the hidden layer and the output layer.

This means that the model has almost 5 millions parameters, which are too many.

Let us now assume that the network has been trained and let us consider a single input neuron. After training the network we got a $L \times d$ matrix $\mathbf{W}$ of weights (where L is the number on nodes in output layer and d is the number of nodes of the input layer). This means that each row of $\mathbf{W}$ is used to define the score of a different output node (i.e., row i is used to compute s_i). Moreover, if we consider a row of the matrix, it can be divided in three sections, each of which is associated to a colour channel. Each row of the matrix is a template and can be used to check how an image is correlated to a specific output class.

4.3.1 Linear classifier

Let us now consider a simple network, since we've seen that we can't add to many layers to a neural network for image classification. In particular, let us consider a linear classifier with a single layer and no activation function that adds non-linearities. The classifier $\mathcal{K}(\mathbf{x})$ can therefore be seen as a linear function

$$\mathcal{K}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where

- $\mathbf{W} \in \mathbb{R}^{L \times d}$ is the matrix of weights (L number of classes, d dimension of the input image),
- $\mathbf{b} \in \mathbb{R}^{L}$ is the vector of biases.

The function $\mathcal{K}$ computes the score of each class, but we want to obtain the predicted class $\mathbf{y}$. This can be achieved by selecting the biggest score. In practice we get

$$\hat{y} = \arg \max_{i=1,\dots,L} \mathbf{s}$$

where L is the number of classes (i.e., of elements in $\mathbf{s}$). The matrix of weights can be divided in areas and in particular, for each row of the matrix, the first part of the row accounts for pixels of a certain channel (e.g., red), the second part for another channel (e.g., green) and the last part for the last channel (e.g., blue).

Training

Let us now analyse how training is done on the linear classifier we have just analysed. Given a loss function $\mathcal{L}$ and a training set $\mathcal{D} = (\mathbf{x}_i, t_i)$, we want to compute the values of $\mathbf{W}$ and $\mathbf{b}$ that minimise the loss, namely

$$[\mathbf{W}, \mathbf{b}] = \arg \min_{\mathbf{W} \in \mathbb{R}^{L \times d}} \min_{\mathbf{b} \in \mathbb{R}^{L}} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \mathcal{L}(\mathbf{x}_i, y_i)$$

Geometric interpretation

Let us now give a geometric interpretation to the linear classifier and to its parameters $\mathbf{W}$. Each row of the weights matrix contains the weights for a class. In particular, row $\mathbf{W}[i :]$ contains the weights for class i. The score of class i is therefore computed as linear combination of $\mathbf{W}$'s i-th row and the input vector:

$$\mathbf{s}_i = \mathbf{W}[i, :] \cdot \mathbf{x} + \mathbf{b}_i$$

Note that this simple operation can be done in Python using the following piece of code:

```
import numpy as np

d = range(28 * 28 * 3)                          # dimension of the input image
L = range(3)                                    # number of classes

W = np.array([[0 for _ in d] for _ in L])       # bidimentional matrix of weights
x = np.array([0 for i in d])                    # input vector
b = np.array([0 for i in L])                    # bias vector

s = np.inner(W[i, :], x) + b[i]
```

Moreover, if we interpret each image as a point in $\mathbb{R}^d$, then the score of a class can be used to define a separating hyperplane that separates the points belonging to a class from the points that don't. If we consider a score with values in $[-1, 1]$, then 0 can be the threshold for assigning a point to a class and

$$s = \mathbf{W}[i, :] \cdot \mathbf{x} + \mathbf{b}_i = 0$$

is the hyperplane separating points belonging to class i from the others. More precisely, the points for which

$$\mathbf{W}[i, :] \cdot \mathbf{x} + \mathbf{b}_i \geq 0$$

belong to class i.

Template matching

Weights can also be interpreted as templates that match specific patterns of an image to classify it. In fact, we can represent each section of the matrix $\mathbf{W}$ (i.e., $\mathbf{W}[:, d/3]$) as an image to understand what features of the input image the model is recognising.

4.4 K-nearest neighbours

Images can be classified also using non-parametric methods. This means that we can build models that don't have to learn any parameter w, hence they don't have to be trained. An example is K-nearest neighbours.

4.4.1 Nearest neighbour

Before analysing the K-nearest neighbours model, let us consider a simpler version. In this model, called nearest neighbour we assign to a in image $\mathbf{x}$ the class of the closest point in the dataset. More formally, $\mathbf{x}$ is assigned to class y_{j^*} where

$$j^* = \arg \min_{i=1,\dots,N} d(\mathbf{x}_i, \mathbf{x}_j)$$

where d is a function that measures the distance between two points and N is the number of points in the dataset. Some examples are the L1 norm

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_1 = \sum_k |(x_{ik} - x_{jk})|$$

or the L2 norm

$$d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2 = \sqrt{\sum_k (x_{ik} - x_{jk})^2}$$

K-nearest neighbours

The K-nearest neighbours model extends the idea of nearest neighbour considering the K closest points and assigning a label through majority voting. Given an image $\mathbf{x}$, the model defines a neighbourhood as the K closest training images to $\mathbf{x}$ and assigns to $\mathbf{x}$ the label which has more represents in the neighbourhood. This means that, if the neighbourhood contains three images $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ belonging to classes C_1, C_2 and C_1, then the input $\mathbf{x}$ is assigned to class C_1 since the majority of points of the neighbourhood belong to class C_1. More formally, $\mathbf{x}$ is assigned to class y_{j^*} where

$$j^* = \text{mode}(\mathcal{U}_k(\mathbf{x}))$$

and $\mathcal{U}_k(\mathbf{x})$ contains the labels of $\mathbf{x}$'s neighbourhood.

Advantages and disadvantages

The K-nearest neighbours model has many advantages, some of which are:

- It's easy to implement.
- It takes no training time.

However, it also has some disadvantages:

- It's computationally demanding at test time and during inference since the algorithm has to visit every point to find the nearest to the point to classify.
- It requires large training sets to be stored in memory during inference (whilst for parametric models we only kept the parameters).
- It's rarely practical on images since high-dimensional objects are difficult to interpret. In particular, different images might have the same distance from another image, hence making hard understanding which class that image belongs to. This comes from the fact that measuring the distance between two images represented as pixel vectors isn't a good way to measure their similarity.

Chapter 5

Convolutional neural networks

5.1 Automatic feature extraction

As we might have understood, feeding a neural network directly with an image isn't always the right choice. A neural network works with features, hence we should be able to extract some features from images, instead of using the image itself. Usually, extracting features from images involves some experts. If we want to analyse ECGs' images we can ask a doctor to define the most important features needed to analyse an EGC result (e.g., the distance between spikes, or the frequency of spikes) and use them as input of a neural network. This means that, before feeding a neural network with an image, we have to extract the values for the features (e.g., get the frequency of the spikes and the distances between spikes) and use them as input values for the network which can now be used to classify the image. After having extracted the features, we can discard the image and work just with the features' values. The main advantages of this approach are:

- If the problem has very distinctive and significant features, we can use them to build a simpler model.
- Features are interpretable.
- We can adjust the features to improve performance.
- We need a limited amount of data.

However, this approach doesn't always work, especially when we have to classify natural images, because we can't manually find significant features. Usually this method is applied only in specific domains and problems in which it's easy to craft features by hand. Moreover, this approach might lead to overfitting models if the training data is used in the design phase.

To solve this issue we have to bring the feature extraction part in the neural network. This means that the neural network is now made of two parts:

- A **feature extraction** network which takes as input an image and outputs a vector of features.
- A **classifier** network that takes as input the features extracted by the previous part and classifies the input image using such features.

This approach is called **data-driven feature extraction** since the feature extraction part isn't done by a human expert but it's done directly on the data used for training the network. This new approach has many advantages, the most important ones being:

- We can optimise the features for classification.
- We can optimise the features while we train the network. This is a big difference with the previous approach in fact hand-crafted features were fixed and couldn't be changed while training the neural network.
- We don't need experts anymore, since features are extracted directly from the data.

It's not all good, though. The main disadvantages of data-driven feature extraction are:

- We lose the interpretablity of the features. Namely, we can't understand nor visualise what features of the image the network uses.
- We need a huge amount of data to train the feature extraction network (i.e., to learn what features should be used).

Note that the extraction and classification parts can use different models (i.e., different types of neural networks), and that's what we usually do in practice. Moreover, we can also use a model different from neural networks for classification (e.g., support vector machines, linear regressors). In this case however, we can't use gradient descend on the whole network. On the other hand, if we use a neural network both for feature extraction and classification, we can train both networks together and using the same training samples.

5.2 Convolution

A convolutional neural network is one of the most used models for feature extraction. This model is based on the **convolution** operation which takes an image and returns another image. In particular, as we have already mentioned, each output pixel is obtained as the weighted sum of a portion of input pixels. Convolution is basically the same as correlation (4.1), the only difference being how the filter's indices are used. In particular, the convolution transformation is defined as

$$T_U[I](r,c) = \sum_{(u,v)\in U} w(u,v)\cdot I(r-u,c-v) = \big(\mathbf{I}\odot\mathbf{w}\big)(r,c) \tag{5.1}$$

5.2.1 Padding

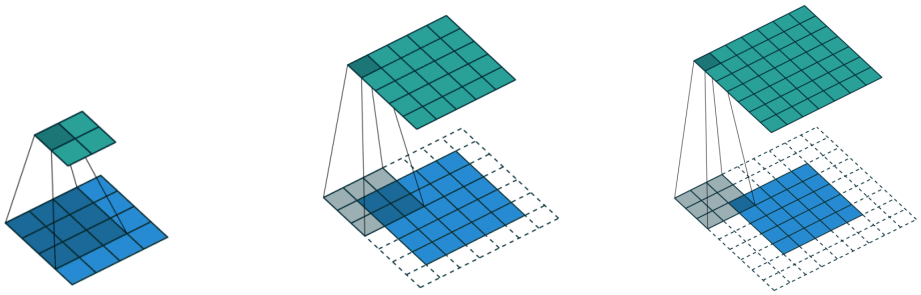
Convolution uses a filter to select the input pixels that are used to compute an output pixel. If the filter contains more than one pixel (as it usually does), we have to carefully define how the output for the pixels on the border of the image is defined. Let us consider for instance a 3×3 filter and a 5×5 image. If we consider the pixel in position $(0,4)$ we would need, to compute its output, the pixels

$$\{(-1,3),(0,3),(1,3),(-1,4),(0,4),(1,4),(-1,5),(0,5),(1,5)\}$$

Unfortunately, only 4 are actual pixels of the image, i.e., $(0,3)$, $(1,3)$, $(0,4)$, $(1,4)$ while the others are all out of bounds. We can solve this problems using padding, namely, adding pixels on the border of the image (usually, but not necessarily, initialised to 0). More precisely, there exist three type of padding:

- **No padding**, or valid padding. If we use no padding, we decide to reduce the size of the output image and compute only the output pixels for which all the required input pixels are available.

- **Half padding**, or same padding. If we use half padding, we add as many pixels as we require so that we can compute the output image. In this case the output image has the same size of the input image. Note that, if we consider a $N \times N$ filter (with N odd), we need to add $\frac{N-1}{2}$ rows or columns for each size. For instance, if we have a 3×3 filter, we add one row at the bottom of the image, one on top of the image, one column on the right and one on the left.
- **Full padding**. If we use full padding, we add padding to increase the size of the output image.



(a) No padding. (b) Same padding. (c) Full padding.

Figure 5.1: Different padding options.

5.2.2 Network layers

Neurons in a convolutional network are organised in three-dimensional structures. More precisely, each layer is made of a bi-dimensional image with multiple channels, namely, each layer is a three-dimensional matrix of pixels. Moreover, going towards the output of the convolutional neural networks, the layers decrease in size (i.e., number of rows and columns) but increase in number of channels. The last layer is a 1×1 image with many channels, which is the feature vector given in input to the classifier network. In some cases, the size of the last layer is not 1×1, hence we have to flatten it (i.e., build a vector from a matrix, row by row or column by column).

In other words, we can see a layers as volume which shrinks in size and increases in width (i.e., in number of channels). An example of convolutional neural network is shown in Figure 5.2.

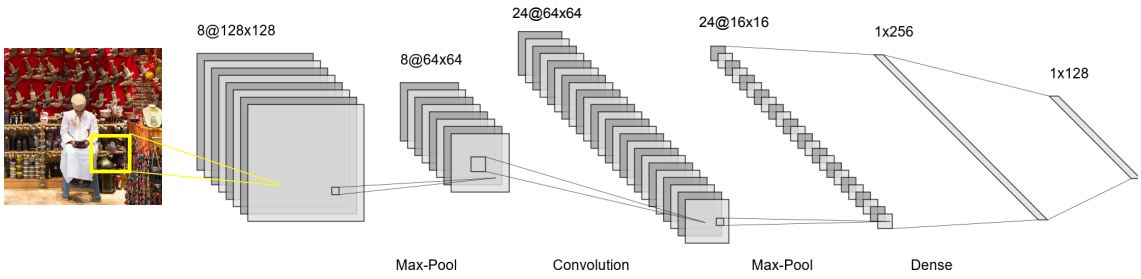


Figure 5.2: A convolutional neural network.

A convolutional neural network can contain different types of layers. In particular, the main layers are:

- **Convolutional** layers.
- **Activation** layers.
- **Pooling** layers.
- **Dense** layers.

Each type of layer has a different job and can be used to change the dimensions of an image and the values of its pixels. Note that, when describing layers we will refer to the input layer as the layer before the transformation and to the output layer as the layer after the transformation. Namely, they aren't necessarily input or output layers of the whole network. Also note that the term deep neural network refers to the fact that such network contains a huge number of layers.

Convolutional layer

A convolutional layer is obtained applying the convolution transformation to the previous layer. More precisely, each channel in the convolutional layer is obtained as the sum of the convolution of all input layer's channels. This means that all the channels in the previous layer are linearly combined (always using only the pixel in the neighbourhood and the filter's weights) to build a single output channel. Since the input image has multiple channels, the filter must have the same number of channels. In other words, a filter has the same number of channels of the input layer and a size $h \times w$, where h and w are hyperparameters of the network. Usually, filters have a small size. Note that, if we want to create multiple channels in the convolutional layer, we must have multiple filters and apply the convolution transformation. In other words, to build channel k we apply convolution using filter k. In formulas, the pixel (r, c) in channel z of the convolutional layer is computed as

$$a(r, c, z) = \sum_{(i,j,k) \in U} w^{(z)}(i, j, k) \cdot x(r + i, c + j, k) + b^{(z)} \tag{5.2}$$

where

- z is the index of the **output channel**.
- $x(r, c, k)$ is the **pixel** in position (r, c, k) of the **input** matrix (k being the channel).
- $w^z(i, j, k)$ is the **weight** in position (i, j, k) (k being the channel) in filter z.
- b^z is the **bias** of the **output** channel z.

Note that, **convolutional layers can increase the width of a layer but can't decrease its size** (unless we use no padding).

Since every channel of a convolutional layer is computed using a different filter, we can describe the layer as a set of filters. The output of a convolutional layer is also called **feature map**.

Activation layer

An activation layer is used to add some non-linearities in the network, using non-linear activation functions. Without activation layers, a convolutional neural network could be equivalent to a linear classifier.

More precisely, an activation layer is obtained performing an activation function on every pixel of the previous layer. This means that an activation layer has the same size of its input layer (i.e.,

the previous layer). In other words, **activations don't change the volume size**. The output of an activation layer is called **feature map** or **activations map**.

One of the most used activation functions is the Rectifier Linear Units (ReLU) function which maps all values smaller than 0 to 0 and the others to themselves.

$$T_{\text{ReLU}}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{5.3}$$

This activation function is very popular and works well, however it suffers from the dying neuron problem that consists of some neurons becoming insensitive to the input. A plot of the ReLU function is shown in Figure 5.3. Another viable choice is the leaky ReLU function which maps non negatives values in themselves and shrinks negative numbers.

$$T_{\text{Leaky ReLU}}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01 \cdot x & \text{if } x < 0 \end{cases} \tag{5.4}$$

Note that we can also use the sigmoid or hyperbolic activation functions, like we did in fully connected networks.

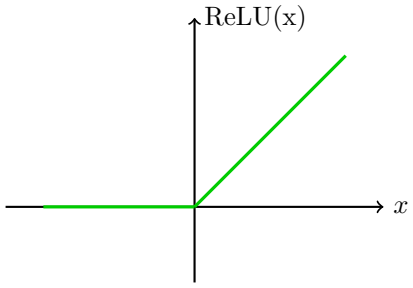


Figure 5.3: The ReLU activation function.

Pooling layer

A pooling layer is responsible for reducing the size of a layer. This means that **a pooling layer reduces the size of a layer, but doesn't increase its width**. This operation is called **down-sampling**. Differently from convolutional layers, pooling layers work separately on each channel. In particular, the pixels of input channel k are used to generate the pixels of output channel k. One of the most used pooling layer is the **max layer**. This layer

1. Takes a region of the input channel.
2. Selects the maximum value of that region.
3. Assigns the maximum value computed at step 2 to an output pixel.

Each channel of a pooling layer is computed with some sort of filter that takes a region of the input matrix and assigns the output to the respective region of the output matrix. For instance, in Figure 5.4, the filter has size 2×2 and the top-left region is used to compute the top-left pixel in the output channel. In this example we can notice another important characteristic of filters. Usually,

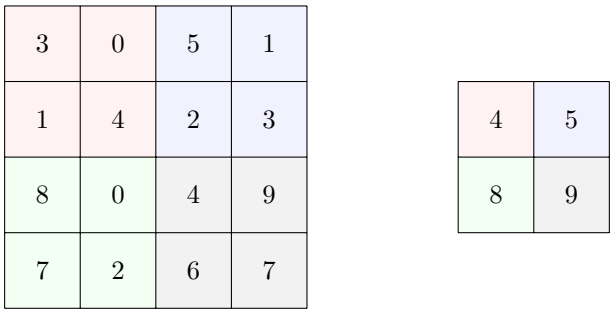


Figure 5.4: The computation of a max pooling layer's channel. The input layer is on the left, the output layer is on the right.

we have toughed of filters as sliding windows that could move by one row or column. In this case we notice that the filter moves two steps in any direction (we go from the region $[[3,0],[1,4]]$ directly to $[[5,1],[2,3]]$ without passing through $[[0,5],[4,2]]$). The step dimension is called **stride** and in max pooling it's usually equal to the size of the pooling region (in our example, in fact, the region is 2×2 and the stride is 2).

Dense layer

A dense layer behaves like in a multilayered perceptron (i.e., the neural network we studied) and works with 1×1 images with many channels, i.e., with vectors. This means that in a dense layer, each output neuron is connected to each input neuron.

We usually combine convolutional layers and pooling layers to reach a vector layer that can be used in a dense layer and eventually fed to the classifying network.

Comparison between CNNs and FFNNs

Say we want to feed a $32 \times 32 \times 1$ image directly to a feed-forward neural network (i.e., to a fully connected multilayered perceptron). The image must be flattered, hence we would obtain a input layer with $32 \cdot 32 \cdot 1 = 1024$ nodes. Also assume that the output layer has 10 nodes. With an input and an output layer only we would obtain a linear classifier, hence we have to add at least one inner layer. Say the hidden layer has 84 neurons. Such network has

- $1024 \cdot 84 + 84 = 86100$ parameters between the input and hidden layers.
- $84 \cdot 10 + 10 = 850$ parameters between the hidden and output layers.

In total the network has 86950 parameters. This is a big number, especially considering that the network used only one hidden layer. CNNs can have these many parameters but such numbers are usually reached with many convolutional layers (i.e. with more non-linearities). If we consider, for instance, an image with three channels, in a FFNN we approximately have 3 times the number of parameters of a CNN considering the same number of layers for both architectures.

5.2.3 Architecture of a convolution neural network

Usually, a CNN is made of a sequence convolutional layers, activation layers and pooling layers and a dense layer at the end to obtain the feature vector. For instance, a CNN could be made of

- A convolution layer to increase the width of the image.
- An activation layer to introduce non-linearities.
- A pooling layer to reduce the size of the image.
- A convolution layer.
- An activation layer.
- A pooling layer.
- A dense layer to obtain the feature vector.

5.3 Training

Now that we know what the architecture of a convolutional neural network is, we have to understand how these networks are trained.

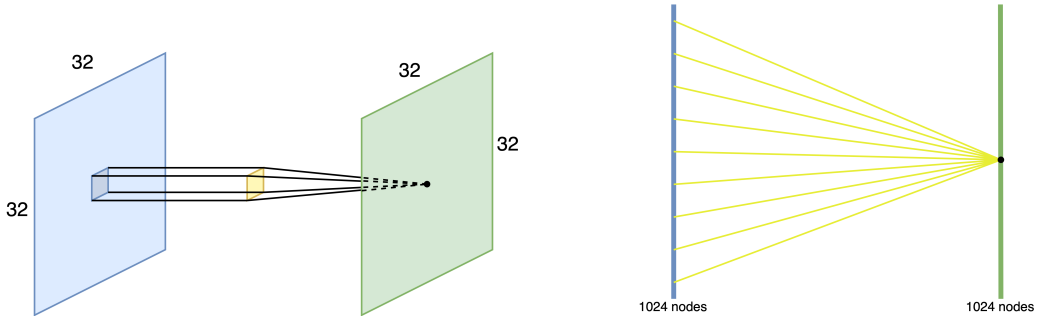
5.3.1 Sparse connectivity

A convolutional layer, being a linear combination of the inputs, can be seen as a fully connected layer of a feed-forward neural network. But a convolutional network is a composition of many convolutional layers, hence we can see a convolutional network as a multilayered perceptron (i.e., a feed-forward neural network). Let us now analyse how we can interpret a convolutional layer as a FFNN layer. In particular, let us consider a $32 \times 32 \times 1$ image and a $5 \times 5 \times 1$ filter. Since we have one filter only, the output is a $32 \times 32 \times 1$ image.

First of all, we have to flatten the pixels of the input and output images in two vectors, each containing 1024 nodes. Now we have to understand how the input neurons (which represent input pixels) are connected to the output neurons. A pixel $x(r, c)$ in the output is the linear combination of 25 input pixels, hence one neuron of the output layer is connected only to 25 input neurons using the weights defined by the filter. This connection is different from the one we've seen in FFNN in which an output neuron was connected to every input neuron. Given this difference, we call this connection **sparse connectivity**. This is the reason why we have fewer parameters in a CNN with respect to a FFNN. If we consider 2 filters instead of just one filter, hence obtaining 2 output layers, the output layer in the FFNN has double the number of nodes. In general, with n filters, the output layer has n times the number of nodes. Also note that an input pixel might influence more output pixels, hence in the FFNN, an input neuron might be connected to multiple output neurons.

Sparse and shared connectivity

The weight used in a FFNN with sparse connectivity are shared among multiple connections since they originate from the filter which uses the same weights for every output pixel. This means that not only we have fewer connections among input and output neurons, but the parameters are also fewer. This type of connectivity is called **sparse and shared connectivity** since the connections are sparse and the weights are shared.



(a) A convolutional layer taking as input a $32 \times 32 \times 1$ image and generating as output a $32 \times 32 \times 1$ image using one $3 \times 3 \times 1$ filter.

(b) Sparse connectivity between two layers of a FFNN. An output neuron is connected only to the input neurons used by the convolution in the CNN.

Figure 5.5: Representation of a convolutional layer in a CNN as a sparse layer in a feed-forward neural network.

5.3.2 Receptive field

Let us stop for a moment, before analysing the actual learning phase, to understand an important concept for convolutional neural networks.

> **Definition 5.1** (Receptive field)**.** *The receptive field of a convolutional neural network or, of one of its layer, is the number of pixels in input that are contributing to the value of an output pixel.*

Usually, the receptive field is computed over the whole network, namely considering the input and output of the convolutional neural network.

This value is very important because if we want a pixel, or a neuron in the equivalent FFNN representation, to analyse a certain feature of the input image, we have to let the neuron see that portion of the image. Basically, if an output neuron of the CNN has to capture a feature in the top right corner of the input image, we have to ensure that the pixels in the top right corner are in the receptive field of the output neuron. In other words, we can't pretend that a neuron captures a feature in a portion of the image if its value isn't influenced by the pixels in that portion of the input. The receptive field can be increased by:

- Increasing the number of layers.
- Increasing the size of the filters.
- Adding pooling layers since, being the output smaller, the neurons are influenced by more neurons with respect to the convolutional layer.

Layers' interpretation

As we move deeper down the network:

- Spatial resolution is reduced.
- The number of maps is increased.

- We search for more high-level patterns without caring about their location.

5.3.3 Back-propagation

Since a CNN can be seen as a FFNN, we can use gradient descend using back-propagation (i.e., the chain rule) to train a CNN. Training isn't however that straightforward in fact we have to deal with shared weights and different layers. Let us then analyse the derivative of every type of layer.

Convolutional layer

Convolution is a linear combination of the input values, hence the derivative is quite simple to compute.

Pooling layer

Since the most used pooling layer is the max-pooling layer, let us analyse its derivative. In a max-pooling layer, the derivative is equal to 1 or to 0, in particular the derivative is

- Equal to 1 for the input node that contributes to the output (i.e., at he location of the maximum of the neighbourhood).
- Equal to 0 otherwise.

If we consider Figure 5.4, the derivative of $x(0,0)$ with respect to $I(0,0)$, $I(0,1)$, $I(1,0)$ is 0 since the inputs aren't the maximum while the derivative of $x(0,0)$ with respect to $I(1,1)$ is 1 since $I(1,1)$ is the maximum of the square in the top left corner.

Activation layer

Let us analyse the derivative of the ReLU function, since it's one of the most used. The ReLU function is defined as

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

hence it's derivative is

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Note that the ReLU function is not differentiable in $x = 0$, but the derivative is put to 0.

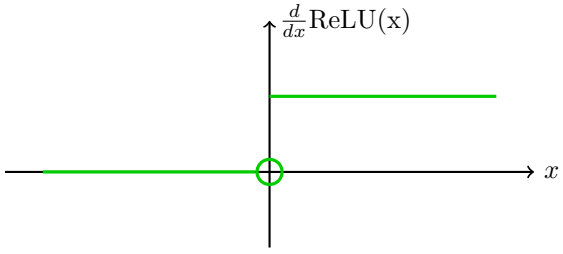


Figure 5.6: Derivative of the ReLU activation function.

5.4 Data scarcity

Training a convolutional neural network requires a huge amount of data, which we don't always have. In such cases we can use some techniques that allow to have a good network even with smaller datasets. Two important techniques to deal with data scarcity are

- **Data augmentation**.
- **Transfer learning**.

5.4.1 Data augmentation

Data augmentation allows to increase the volume of data by applying some transformation to the images in the dataset. The transformations can change the geometry of the images or other characteristics. Some examples of transformations are:

- **Shift**.
- **Rotation**.
- **Shear** (distortion along an axis).
- **Flipping**.
- **Scaling**.
- **Modify pixels' intensity**.
- **Modify the image's contrast**.
- **Modify the image's exposition**

These transformations can be executed at each training epochs. It's however important to remember that data augmentation should preserve its label (i.e., output class). For instance, if the size of an image is a key information to classify it, we can't scale the image because we would obtain a wrong output class.

Invariance

Data augmentation is good not only because it allows us to work with reasonable datasets, but also because it increases the invariance of the model. That is to say that it allows the model to learn how to recognise even transformed images. For instance, if one of the output classes is a plane and we do data augmentation using rotation, the classifier learns to recognise planes in all directions. In this case we say that we train the network to be rotation invariant.

Overfitting

Data augmentation is also able to reduce overfitting since data augmentation increases the size of the dataset we are training the network with, and overfitting reduces with more training data.

Test time augmentation

Even if a network is trained using data augmentation we can't achieve perfect invariance with respect to the considered transformations. In these cases we can perform data augmentation even during testing to improve prediction accuracy. Test time augmentation, or self-ensembling, works as follows:

1. Perform random augmentation on each test image I. Namely, from a test image I we obtain
$$A_l(I)$$
2. Classify the augmented images and save the predictions in a posterior vector $\mathbf{p}_l$. For each image, we obtain
$$\mathbf{p}_l = predict(A_l(I))$$
3. Average the predictions on each augmented image. If we call $\{\mathbf{p}_l\}_\mathbf{l}$ the set of prediction l, one for every image, we obtain
$$\mathbf{p} = avg(\{\mathbf{p}_l\}_\mathbf{l})$$
4. Take the average vector of posteriors for defining the final guess.

Test time augmentation is particularly useful for test images where the model is pretty unsure.

5.4.2 Confusion matrix

Confusion matrices are very useful tools to assess a model's performance. A confusion matrix is a $L \times L$ matrix with L number of classes. Each row of the matrix represent a true class (i.e., in the dataset) and each column represents a class predicted by the classifier. The element C_{ij} of the matrix corresponds to the percentage of elements belonging to class i that have been classified as elements of class j. This means that the ideal confusion matrix has all 1s on the diagonal and 0s elsewhere. The closest we are to this ideal situation, the better the model is. An example of confusion matrix is shown in Figure 5.7.

Binary classification

In binary classification we can see the output of a network as a single value, which is the probability p of belonging to class 1 (considering classes 0 and 1). Classification is then performed using a threshold Γ such that:

- If $p \geq \Gamma$, we assign the input to class 1.
- If $p < \Gamma$, we assign the input to class 0.

The value of Γ can be set to balance the trade-off between false-positives (False Positive Ratio) and true-positives (True Positive Ratio).

The performance of a binary classifier can also be monitored using the Receiver Operating Characteristic (ROC) curve, which doesn't depend on the threshold Γ.

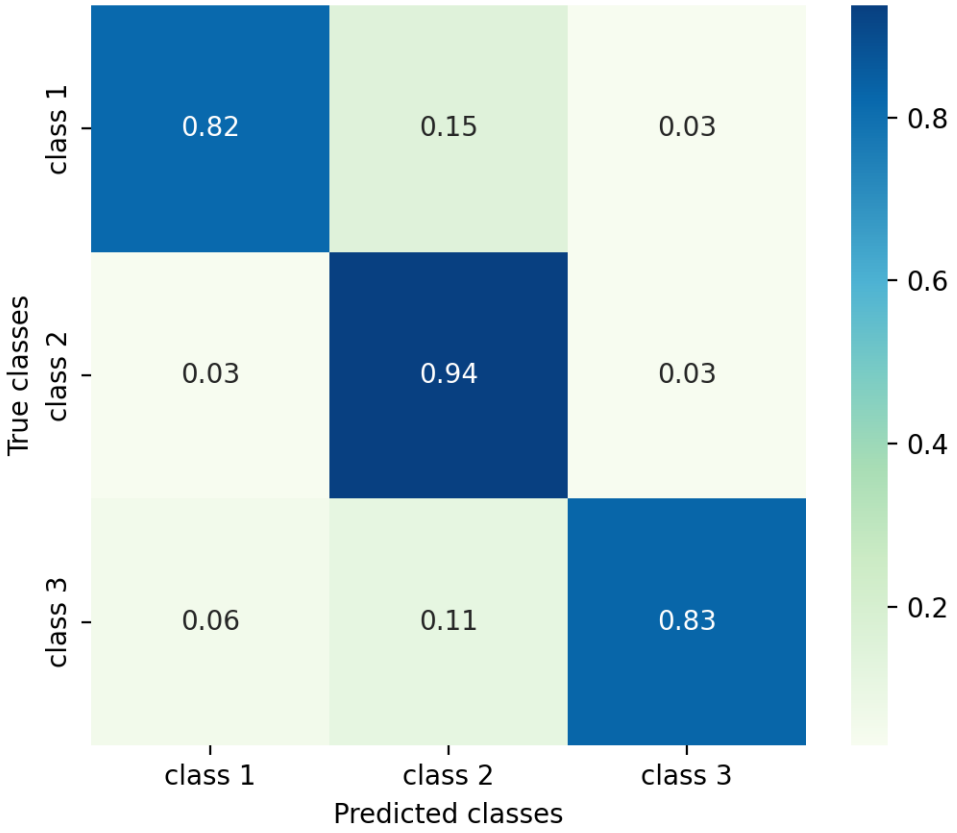


Figure 5.7: An example of confusion matrix.

5.4.3 Transfer learning

Say we are trying to build and train a CNN to solve a specific problem. If we haven't got enough data to adequately train the network, we can use an already trained CNN (stripped of its classification network) for feature extraction and then adapt such network to a classification network we build. Usually the pre-trained CNN has been trained for a different problem and on different data with respect to ours. This isn't however a big problem since a CNN for feature extraction is a general purpose block that simply has to extract some features from images. It's safe to assume that a sufficiently general CNN trained to recognise different images can be reused for another problem. The real problem-specific part is the classification network which should be designed ad-hoc for the problem we are trying to solve.

Training

Note that, even when using a pre-trained CNN, we still have to train the classifier. Depending on the volume of data we have and on whether the pre-trained CNN is close to our problem's domain we can use:

- **Transfer learning**. If the pre-trained CNN has been trained for a domain of application close to ours, we can freeze weights and parameters of the CNN and train the classifier only.
- **Fine tuning**. If the pre-trained CNN has been trained for a domain of application not so close to ours, we can train both the pre-trained CNN (used for feature extraction) and the classifier. Note that, the weights and parameters of the CNN aren't initialised at random but we use the original (pre-trained) weights of the CNN and learn starting from there. Note that, for the same optimiser, when performing fine tuning we can use lower learning rates than when training from scratches.

Adapting the architecture

A pre-trained CNN has been designed to take an image of a specific size and output a certain number of output features. This means that we have to adapt our architecture to the pre-trained CNN. More precisely,

- Using images of a different size isn't a big problem since convolution works independently from the input size. This means that we can put in input images with size different from the one of the CNN and still not break anything.
- The network breaks when we connect it to our classifier. This is where we have to design our classifier properly.

Another thing to keep in mind is that the pre-trained CNN has been designed with a specific receptive field, hence it might not work for our problem if we use images of different sizes.

5.5 Convolutional neural networks architectures

It makes sense to give a look at some famous CNNs, since we've seen that we can take the feature extractor of a CNN and use it on a different problem.

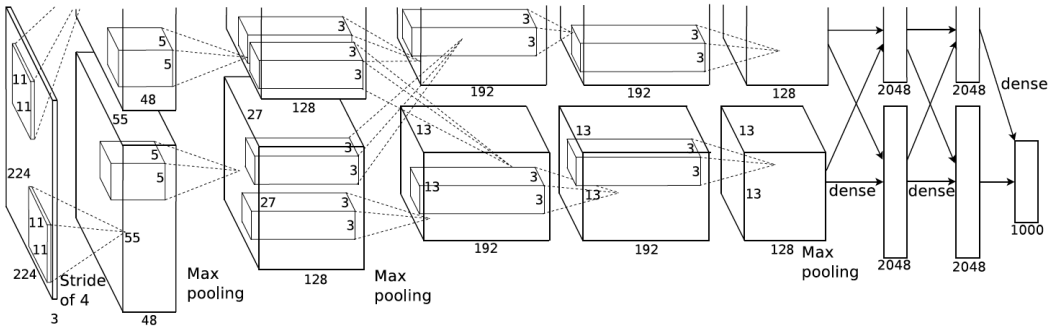


Figure 5.8: AlexNet architecture.

5.5.1 AlexNet

AlexNet is a convolutional neural network made of

- 5 convolutional layers using 11×11, 5×5 and 3×3 filters.
- 3 multilayered perceptron layers.

that takes as input $224 \times 224 \times 3$ images (i.e., RGB images). Note that the original paper proposing this architecture used images of size $227 \times 227 \times 3$. The network has around 60 million parameters, most of which are in the fully connected layers (3.7 millions in the convolutional network, 58.6 in the fully connected part). Note that this many parameters usually can't be stored on a single GPU, hence the network is divided in two and each part is loaded on a GPU.

To reduce overfitting, AlexNet uses:

- Dropout and weight decay.
- The ReLU activation function.
- Normalisation layers (which have been dropped in later versions of the architecture).
- Max pooling.

Moreover, the first layer uses stride 4 for its 96 11×11 filters, which introduces an implicit downsizing (more precisely, the images' size is divided by 4). A representation of AlexNet is shown in Figure 5.8. The paper proposing this architecture is available here.

5.5.2 VGG16

VGG16 is a well known general purpose CNN. Compared with AlexNet, VGG16 uses smaller filters which means that the network has to be deeper to achieve the same receptive field. This is because we can obtain the same receptive field with one big filter or multiple layers that use smaller filters. In particular, in this architecture, a 7×7 filter on a convolutional layer has been replaced by 3 convolutional layers using a 3×3 filter each. This leads to

- Less parameters, in fact we shift from $7 \times 7 = 49$ parameters to $3 \cdot 3 \cdot 3 = 27$ parameters (three 3×3 filters, having 9 parameters each, hence $9 \cdot 3 = 27$).

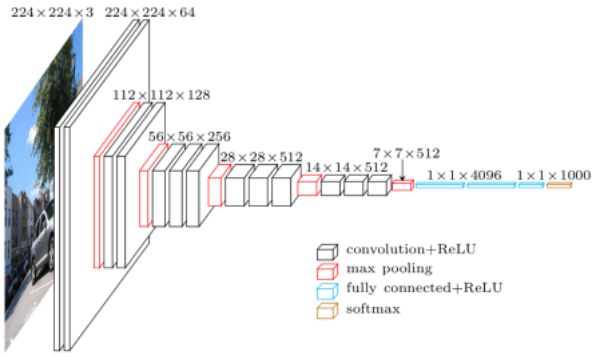


Figure 5.9: VGG16 architecture.

- More non-linearities since we have to use an activation function for every layer.
- The same 7×7 receptive field.

As a result the VGG16 has around double the number of parameters of AlexNet (around 138 million), most of which are part of the fully connected network. As for AlexNet, the input size is $224 \times 224 \times 3$.

A graphical representation of VGG16 is shown in Figure 5.9. The paper proposing this architecture is available here (`https://arxiv.org/abs/1409.1556`).

5.5.3 Network in network

The network in network architecture introduces two new type of layers,

- The **minconv layer**.
- The **global average pooling layer** (GAP layer).

Minconv layer

A minconv layer replaces convolution with a fully connected layer and the ReLU activation function. More precisely, the minconv layer uses a stack of fully connected layers followed by a ReLU activation (the ReLU is after each fully connected layer). Basically an output pixel p, instead of being computed as

$$p = ReLU\left(\sum_i w_i x_i + b\right)$$

is computed as

$$p = \text{ReLU}\left(\sum_i w_{i,1}\left(\text{ReLU}\left(\sum_i w_{i,2} x_i + b_2\right)\right) + b_1\right)$$

The main advantage of this architecture is that each layer has a more powerful functional approximation than a convolutional layer, which is just a linear combination followed by a ReLU activation. Minconv layers have been used in this architecture but aren't, in general, very popular.

Global averaging pooling layer

A global averaging pooling layer computes the average slice-wise of the input, returning a vector of pixels. Namely, an output pixel is computed as the average of all the pixels in an input channel. This means that a GAP layer massively reduces the dimensions of the input while keeping the same number of channels (i.e., input and output have the same number of channels). Moreover, since a pixel is computed using every pixel of the image, the receptive field of a pixel is the whole input image.

In the network in network architecture, GAP layers have been used instead of the fully connected layer at the end of the network. In practice, in the network in network architecture they have:

1. Removed the fully connected layer at the end of the network (i.e., after the CNN).
2. Introduced a GAP layer in replacement of the fully connected one.
3. Predicted the output using softmax after the GAP layer.

Note that the number of channels has to corresponds to the number of output classes if we want to put a GAP layer as last layer of the network. If this property doesn't hold, we can introduce an hidden layer to adjust the number of features.

In general, the main advantages of using a global averaging pooling layer at the end of a CNN are:

- We don't have parameters to optimise in the GAP layer since it's simply an average. This also means that the network is lighter and less prone to overfitting.
- Classification is performed using softmax.
- The model is more interpretable since we create a direct connection between channels and output classes.
- It increases the robustness to spatial transformation of the input images.
- The network can be used to classify images of different sizes since we simply have to compute the average of the pixels, independently of how many are there.

General architecture

The network in network stacks:

- Some layers, each made of a minconv, dropout and maxpooling layer.
- A GAP layer followed by softmax.

A representation of the network in network architecture is shown in Figure 5.10. The paper proposing this architecture is available here (`https://arxiv.org/abs/1312.4400`).

5.5.4 GoogleNet

GoogleNet is a convolutional neural network that exploits a novel layer called **inception module**. The main issue addressed by the GoogleNet is training a big and large neural network to improve visual recognition performance. The classical approach is to either increase the width or the depth of the network. These solutions lead however to a huge number of parameters and are prone to overfitting. Moreover, at times, some features can appear in different scales, hence we should be able to choose the correct kernel (i.e., filter) size. The inception module solves both problems. In practice, GoogleNet has only 5 million parameters and uses 22 layers of inception modules.

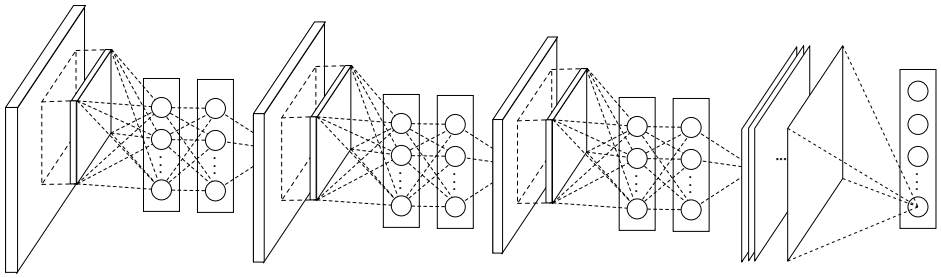


Figure 5.10: Network in network architecture.

Basic inception module

An inception module, differently from a normal convolution layer, has multiple branches that start from the input layer and, after some computation, are rejoined to build the output layer. Each branch applies convolution or pooling (mainly max pooling), in parallel, uses filters of different sizes and the output volumes of the convolution branches are concatenated to build the inception module's output. Note that, in order to concatenate the output of the parallel convolutions, we have to

- Use same padding (with zeros) for convolution to preserve the input size.
- Use fractional strides for pooling, i.e., instead of stride N use stride $\frac{N}{m}$ so that the output size is the same as the input size. More information is available here [1].

The basic inception module, among all filters, uses $1 \times 1 \times K$ filters (where K is the number of channels of the input). These filters are used to average the values of a single pixel in all channels. That is to say, an output pixel in position (r, c) is computed as the average of the pixels in position (r, c) of all the channels. In formulas, convolution becomes

$$a(r, c) = \sum_{k \in K} w(k) \cdot x(r, c, k) + b$$

where

- K is the number of channels of the input image.
- $w(k)$ is the weight of channel k.

The problem with this basic inception module is that, when the outputs are concatenated, the output of the module becomes very deep (also considering that we we might use a big number of filters for each branch). This means that stacking multiple inception modules becomes computationally expensive.

Inception module

Google decided to solve the problem of the basic inception module using those $1 \times 1 \times K$ filters we talked about early on. In particular, these filters can be used to reduce the depth of an image without executing many operations. Say, for instance, we have a $56 \times 56 \times 64$ volume. We can reduce its depth using 32 $1 \times 1 \times 64$ filters to obtain an output volume of size $56 \times 56 \times 32$ (since we used

[1] https://arxiv.org/abs/1412.6071

32 filters). This is very useful when the input of an inception module is very deep (as it happens when we stack multiple inception modules). More precisely, the $1 \times 1 \times K$ filters are added:

- Before higher dimension filters (i.e., before $3 \times 3 \times K'$ and $7 \times 7 \times K'$ filters).
- After max-pooling layers.

This allows the $3 \times 3 \times K'$ and $7 \times 7 \times K'$ filters to be less deep. With shallower filters we can improve the performance since we have to compute, for each pixel, $3^2 \cdot K'$ (or $7^2 \cdot K'$) operations instead of $3^2 \cdot K$ operations and, if $K' \ll K$, we can save a lot of time. Note that this operation is done with $1 \times 1 \times K$ filters since they require the smallest number of operations.

To sum things up: concatenating multiple convolution outputs generates very deep volumes which worsen the performance when computing convolution. To solve this problem we reduce the depth of every module's input layer before executing the convolution with large size filters.

General architecture

GoogleNet is made of 9 layers, each made of the inception block we have just described. More precisely we have:

- A traditional convolution with max-pooling layer.
- The 9 layers of inception modules.
- A global averaging layer.
- A linear classifier.
- Soft-max for classification.

Together with the soft-max classifier, the network contains two additional classifiers, placed in the middle of the network, that:

1. take the activation of a layer,
2. unroll it,
3. classify it.

This is done to define the loss used during training and to mitigate the dying neuron problem. Note that these classifiers are used only for training, hence they are thrown away after training and aren't used for prediction.

5.5.5 ResNet

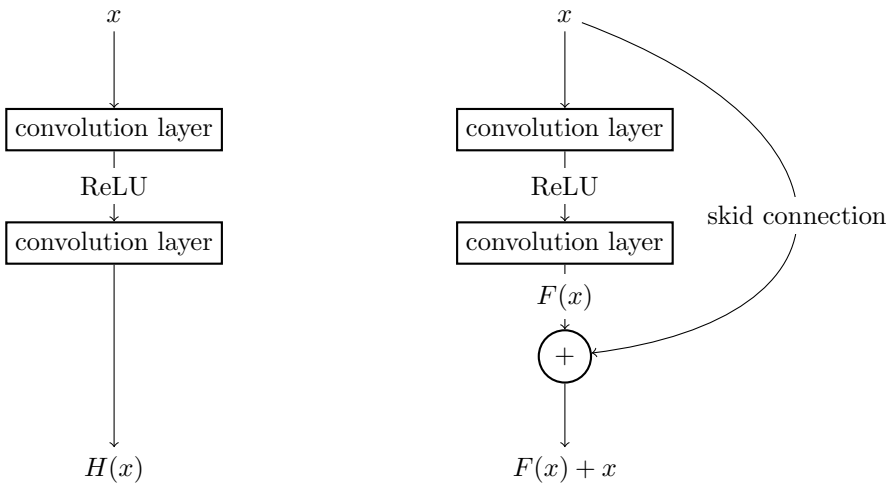
ResNet is a network that exploits a new layer. Before understanding how this new layer is done, let us analyse what is the idea behind this type of network and what problem its inventors were trying to solve. The problem that originated this network is that, the training error of deeper and bigger networks is typically higher than the testing error of networks with fewer layer. One might think that the reason is overfitting, however the same happens on training error (hence it can't be overfitting since an overfitting network has a small training error). The actual reason for this behaviour is that the deeper the network is, the harder training becomes. If this sounds strange, let us consider the following example. Say we have a network with 20 layers that we know is the best

network to solve a certain problem (i.e., it's the network with the minimum error). If we add more layers after those 20 layers, these should learn the identity function, since the previous 20 layers have minimum error. Namely, the added layer should learn not to modify the result of the previous layers. In other words, a deeper network should have a larger potential. This is not happening since the added 20 layers actually deteriorate the performance of the network leading to a larger error. This is because neural networks are not good at learning the identity function.

This intuition leads to the following block, called **residual block**. Say we have two convolution layers (with some activation function) that learn a function $H(x)$. Instead of using these blocks directly, we sum the input x of this sequence of layers to the output of the last layer. The connection between the input and the output is called **skid connection**. In this way the output is $F(x)+x$ and the function that the two layers have to learn is $F(x)$, which is called **residual function**, instead of $H(x)$. Note that the final outcome of the block is the same, hence

$$H(x) = F(x) + x$$

but the layers have to learn a simpler function $F(x)$ and we have to train a smaller network. A representation of a basic block is shown in Figure 5.11.



(a) A basic block without skid connection.

(b) A basic block in ResNet.

Figure 5.11: A basic block in ResNet.

We shall note two important things. First, the basic block shouldn't modify the spatial extent of the layers. In other words, the output of the second convolution $F(x)$ should have the same dimensions of the input x, otherwise we can't sum them. Moreover, learning using a basic block is improved, even if $F(x)$ and $H(x)$ require the same number of parameters to be trained.

General architecture

ResNet is a stack of 152 basic blocks. More precisely, the network alternates blocks that use

- Pooling by convolution with stride 2.
- Convolution with double the number of filters with respect to the previous layers.

The whole architecture is made of:

1. A convolutional layer.
2. 152 basic blocks.
3. A global averaging pooling layer with softmax for classification. This means that the network doesn't use a fully connected layer at the end for classification.

Wide ResNet

Wide resent is a variation of ResNet in which we use more filters during convolution. This allows to have wider residual blocks.

ResNeXt

ResNeXt mixes ResNet and GoogleNet using inception modules inside residual blocks.

5.5.6 DenseNet

In each block of DenseNet, each convolutional layer takes as input the output of the previous layers. Each layer is connected to every other layer in a feed-forward fashion. This:

- Alleviates the vanishing gradient problem.
- Promotes feature reuse since each feature is spread through the networks.

5.5.7 EfficientNet

EfficientNet proposes a new scaling method that uniformly scales all dimensions of depth, width and resolution using a simple yer highly effective compound coefficient.

5.6 Visualisation

We are now interested in analysing the filters of a convolutional neural network as images.

5.6.1 Maximally activating patches

After the first layer, it's impossible to understand what feature a filter is selecting, hence it's useless to check the filters' directly. What we can do is checking the activations. In particular, we can check what neurons are triggered when recognising a specific patter or class of images. In particular, we can define an algorithm that, for each neuron, finds the image that maximally activates it. This allows us to check, for each neuron, what characteristic of an image it's considering. The algorithm works as follows:

1. Choose a neuron N.
2. Take the training images.
3. Keep track of the activations for neuron N on all training images.

4. Select the image that maximally activated N (i.e., for which N had the maximum value). We can also show the receptive field of the considered neuron to better understand what part of the image has activated that neuron.

The result of the algorithm on the neurons of a deep layer is shown in Figure 5.12. Note that this is an a-posteriori analysis. This technique is very good for understanding deep layers since we can see what images are activating the neurons of the layer, even if we can't visualise the filters.

Figure 5.12: An example of maximally activating patches.

5.6.2 Image maximally activating a neuron

Neurons' activations can also be analysed using gradient ascend. In particular, when feeding the network with an image, we can compute a neuron's activation derivative with respect to the input and use it to maximise the input. Basically, we adjust the input image following the gradient (this time in the gradient direction, not like in gradient descend) to find the image that maximally activates the neuron. When applying this technique we obtain images that represent what pattern a certain neuron is recognising. Analysing the results when applying this technique we notice that:

- Shallow layers recognise fine and low-level patterns.
- Deep layers recognise complex and high-level patterns.

The process we have just described works as follows:

1. Train the network.
2. Fix the parameters.
3. Consider a neuron.
4. Maximise the value of the neuron over the set of training images.

This process can also be applied to the output neurons to understand what features of an image are used to recognise a specific class. In particular, we have to compute

$$\hat{I} = \arg\max_I S_c(I) + \lambda \|I\|_2^2$$

where λ is a normalisation parameter, c is an output class.

5.6.3 Meaning of filters

The first layers of a convolutional neural network match low level features like edges and simple patterns whilst deeper layers match more complex and high-level features. This makes sense since first we recognise basic shapes which are then combined by deeper layers to create more complex patters.

5.7 Data preprocessing

When we are given a dataset to train a network, it's better to do some operations before actually using it. In particular we might want to:

- Centre the data around the origin.
- Scale the data so that the standard deviation is 1.

Note that these are only options, hence one could choose to use one, the other, both or neither. Another operation we can do to preprocess the dataset is Principal Component Analysis (PCA). This technique allows to find out the k dimensions that have more variance and keep only such dimensions, remodelling the data around them. This allows to reduce the spatial extension of the data set, loosing as less information as possible (that's why we choose the directions with highest variance). PCA isn't however used that much in image recognition. Another technique we can adopt is whitening, which, as for PCA, isn't that popular in image recognition.

Data preprocessing improves the training performance of the model, but it's customary. This means that it's part of the model. For this reason, everything that concerns data preprocessing should be computed on the training dataset. Once we have defined a transformation θ using the dataset, we have to apply θ to validation and test set, too. This means that it's wrong to perform preprocessing and then divide the data in training, validation and test set. Say for instance, we want to centre the dataset around the mean. We have to:

1. Compute the mean on the train set.
2. Shift by the mean both train and test set.

5.7.1 Mean subtraction

Mean subtraction is a well established technique for centring a dataset of images around their mean. In particular, given (H, W, D) images, we can do one of the following:

- Subtract the mean image, which also is a (H, W, D) image, from each image of the dataset.
- Subtract the per-channel mean, that is we compute the mean along each channel and, for each channel, we subtract the mean from each pixel of that channel (i.e., we subtract the mean of channel i from every pixel of channel i). Note that, in this case, we only have to compute D numbers (i.e., the mean for each channel).

- Subtract the per-channel mean and divide by the per-channel standard deviation. The idea is similar to the one we saw in the previous point but, in this case, we also compute the standard deviation for each channel and we divide each pixel for the standard deviation of its channel. In this case the number of values to compute is 6 (3 means and 3 standard deviations).

5.7.2 Batch normalisation

Consider a batch of activations $\{x_i\}$ and for each activation x_i, subtract the mean of the batch and divide by the standard deviation. Basically, each activation x'_i (following the normalisation) is computed as

$$x'_i = \frac{x_i - E[x_i]}{\sqrt{\text{var}[x_i]}}$$

where $E[x_i]$ and $\text{var}[x_i]$ are computed for each batch $\{x_i\}$ and separately for each channel. These values, after training, are replaced by the running averages of values seen during training. A further parametric transformation can be

$$y_{i,j} = \gamma_j x'_i + \beta_j$$

where γ_j and β_j are learnable scale and shift parameters.

The main advantages of this technique are:

- Easier training.
- It allows to use higher learning rates, hence the algorithm converges faster.
- It improves gradient flow.
- The model is more robust to pure initialisation.
- We have no overhead during inference (hence at test time).
- It acts as regularisation during training.

Note that, the network behaves differently during training and testing. Moreover, during testing, batch normalisation becomes a linear operator, hence it can be fused with the previous fully-connected or convolutional layer. In practice, networks that use batch normalisation are significantly more robust to bad initialisation. Typically, batch normalisation is used in between fully connected layers of a deep convolutional network but sometimes also between convolutional layers.

Algorithm 2 Batch Normalising Transform, applied to activation x over a mini-batch.

Input: Values of x over a mini-batch $\mathcal{B} = \{x_1, \dots, x_m\}$
Input: Parameters γ, β to be learned.
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})$
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}}$
$y_i \leftarrow \gamma\hat{x}_i + \beta = \text{BN}_{\gamma,\beta}(x_i)$

5.8 Fully connected convolutional neural network

If we feed a neural network designed for (W, H, D) images with (W', H', D') images, it breaks in the last layer (i.e., in the fully connected network) since, in principle, convolution works with arbitrarily big images. On the other hand, the fully connected layer wants a fixed size input, hence with smaller or bigger images, we would obtain a smaller or bigger feature vector. The CNN can't therefore compute class scores, but it can extract features. Let us consider a fully connected network with an input and an output layer (which is a part of a bigger network). If this section has N input nodes and L output nodes, then the output of neuron i can be computed as

$$o_i = \sum_{j=1}^{N} w_{i,j} s_j + b_i$$

where s_j is the j-th input. Weights and inputs can be written as column vectors

$$\mathbf{w_i} = (w_{i,1}, \ldots, w_{i,N})^T$$

and

$$\mathbf{s} = (s_1, \ldots, s_N)^T$$

Thanks to this new notation, we can write the output o_i of neuron i as

$$o_i = \mathbf{w}_i^T \cdot \mathbf{s} + b_i$$

This expression resembles the one in Equation 5.1, used for convolution (at the origin), hence a fully connected layer can be seen as a convolution layer with $1 \times 1 \times N$ filters. In general a fully connected layer with A neurons in input and B neurons in output can be seen as a convolutional neural network with B filters of dimension $1 \times 1 \times A$. Each filter contains the weight of the fully connected layer for the corresponding output neuron. If the input image is bigger than expected, the input dense layer is a $M_1 \times M_2 \times N$ vector instead of a $1 \times 1 \times N$ vector (since convolution before has reduced the size, but not enough to reach 1×1). Since we apply convolution using L $1 \times 1 \times N$ filters, we get, as output an $M_1 \times M_2 \times L$ image.

As a result (by extension), a fully connected neural network can be seen as a convolutional network that uses $1 \times 1 \times C$ filters. Such network is called fully connected convolutional neural network or fully convolutional neural network.

To sum things up, we can fix a fully connected convolutional neural network using convolution layers with L filters of size $1 \times 1 \times N$ which allow to handle inputs of whatever size. In particular:

- If the input is a vector, the convolution layer works normally and generates a $1 \times 1 \times L$ vector.
- If the input is a $W \times H \times N$ image, then the filters allow to obtain an image with same size but less channels, namely a $W \times H \times L$ output.

When we reach the output layer of the whole network:

- A normal $1 \times 1 \times L$ vector provides the the probability that an input belongs to a certain class. In particular, the i-th value contains the probability that the input image belongs to class i.
- A $W \times D \times L$ image provides an heat map. Each pixel of the i-th heat map's channel contains the probability that the input image belongs to class i.

5.8.1 Heat maps

When feeding a fully connected convolutional neural network with an image bigger than the ones used for training, we obtain an output image having:

- Lower resolution than the input image since we have gone from N to L channels.
- Class probabilities for the receptive field of each pixel (assuming that softmax remains performed column-wise).

This is because, when using standard images (i.e., with the same size of training images), each output neuron represented a class. Now, each neuron still represents a class but it's not a single value but a matrix, hence an image. This image is very low resolution since it has only $M_1 \times M_2$ pixels, however it allows us to tell, where in the image the network recognised an output class. In other words, each channel of the output image represents an output class and a pixel $p(c, r)$ gets activated in channel k only if the neural network recognised that, in the original image, that area contained a subject belonging to class k. In practice, if class k recognise tyres and we feed an image of a car from the side, the pixels in the bottom left and right corners will have an high value in channel k since the tyres are in the bottom left and right corner of a car's image. As a result, we get, as output, the heat map that tells where we can find, in a larger dimension image, the image with standard dimension classified by the network.

An example of the last layer of a fully convolutional neural network is shown in Figure 5.13.

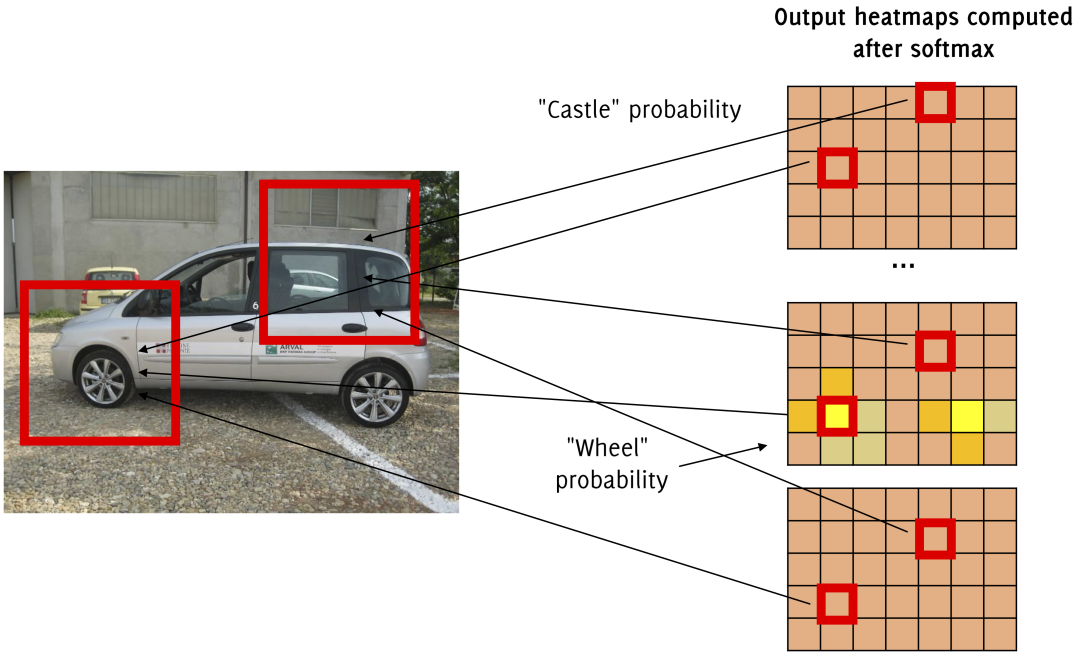


Figure 5.13: The heat-maps generated by the CNN when fed with higher dimension images. The images on the right are the $M_1 \times M_2$ images for each class k. Each image specifies, for each pixel, the probability that such pixel (that represents a region of the input image) belongs to class k.

Chapter 6

Segmentation

6.1 Semantic segmentation

Semantic segmentation aims at gather together pixels that belong to the same object. An example is, given an image, distinguish all the people from the background. Formally, we want to assign to each pixel a label, hence this problem is also called pixelwise classification. Even more formally:

> **Definition 6.1** (Semantic segmentation)**.** *Given an image $I \in \mathbb{R}^{R \times C \times 3}$ and a set of labels Λ we want to find a mapping*
>
> $$I \to S \in \Lambda^{R \times C}$$
>
> *where $S(x, y) \in \Lambda$ is the class associated to pixel (x, y), that assigns to each pixel $I(x, y)$ of I a class in Λ.*

From this definition it's clear that a pixel can't belong to two different classes. Also note that, there is no distinction between instances of the same class. That is to say, if in an image we have many people, we can't tell them apart. In other words, a pixel belongs to the class *people* and not to a specific person.

If we want to recognise also different instances of the same class (e.g., different people), we have to solve an instance segmentation problem.

6.2 Models

6.2.1 Fully convolutional neural network

The simplest approach one can take to solve the segmentation problem is to use fully convolutional neural networks. In fact, a fully convolutional neural network allows to find in an image the location of the classes we have in output. Adapting a FCNN for this problem is easy, in fact we simply have to build a FCNN that recognises the labels of the problem and then feed the network with a larger image that contains the objects classified by the network. The main problem with this simple approach is that the output image we obtain has a very low resolution (i.e., very few pixels). This means that we have to do some up-sampling to obtain an output of the same dimension of the input. To sum things up, the strategy we want to follow is:

1. Use a fully connected convolutional neural network to perform classification on a high resolution (big number of pixels) image.
2. Obtain low resolution heat-maps (one for each output class) that tells where the objects are in the input image.
3. Up-sample the heat-maps to obtain an heat map of the same resolution of the input image.

We have already explained the first two steps when discussing about fully convolutional neural networks. Let us now focus on the up-sampling part. We will analyse three ways for up-sampling an image:

- Direct up-sampling.
- Shift and stitch.
- Convolution only networks.

Direct up-sampling

Say we have a $\hat{R} \times \hat{C} \times L$ heat map that we want to up-sample. We can define an up-sampling (or down-sampling) factor f so that we obtain, as result of the up-sampling, a $f\hat{R} \times f\hat{C} \times L$ heat map. Up-sampling is done computing the maximum label posteriori on every heat map and expanding each pixel of the heat map with a factor f using the maximum posterior. If a pixel has is labelled with class C_1, and $f = 2$, we represent that pixel with a 2×2 block of pixels, each assigned to class C_1. The same example is shown in Figure 6.1.

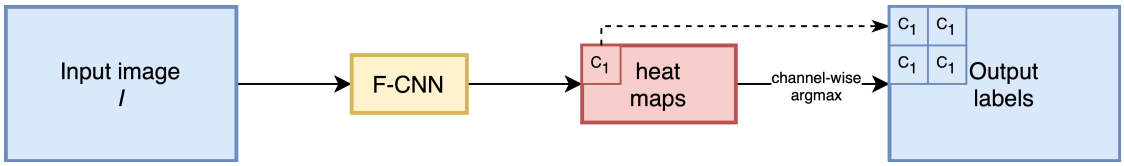


Figure 6.1: Direct up-sampling.

Shift and stitch

Shift and stitch is another method for up-sampling an heat map. In particular, it applies direct up-sampling using shifted images. If we have a down-sampling factor f, then we have to shift the input image in every direction and apply direct up-sampling with factor f to each of these shifted inputs. The results are finally combined to get an higher resolution image. More precisely, we have to

1. Compute heat maps for all f^2 possible shifts of the input (considering shifts in every direction, i.e., no shift, left, right, up and down).
2. Compute the class for each pixel in the heat map using argmax on the channels.
3. Combine the heat maps obtained. In particular, the pixels (r, c) of the different output heat maps are used to compute the $f \times f$ block of the output image.

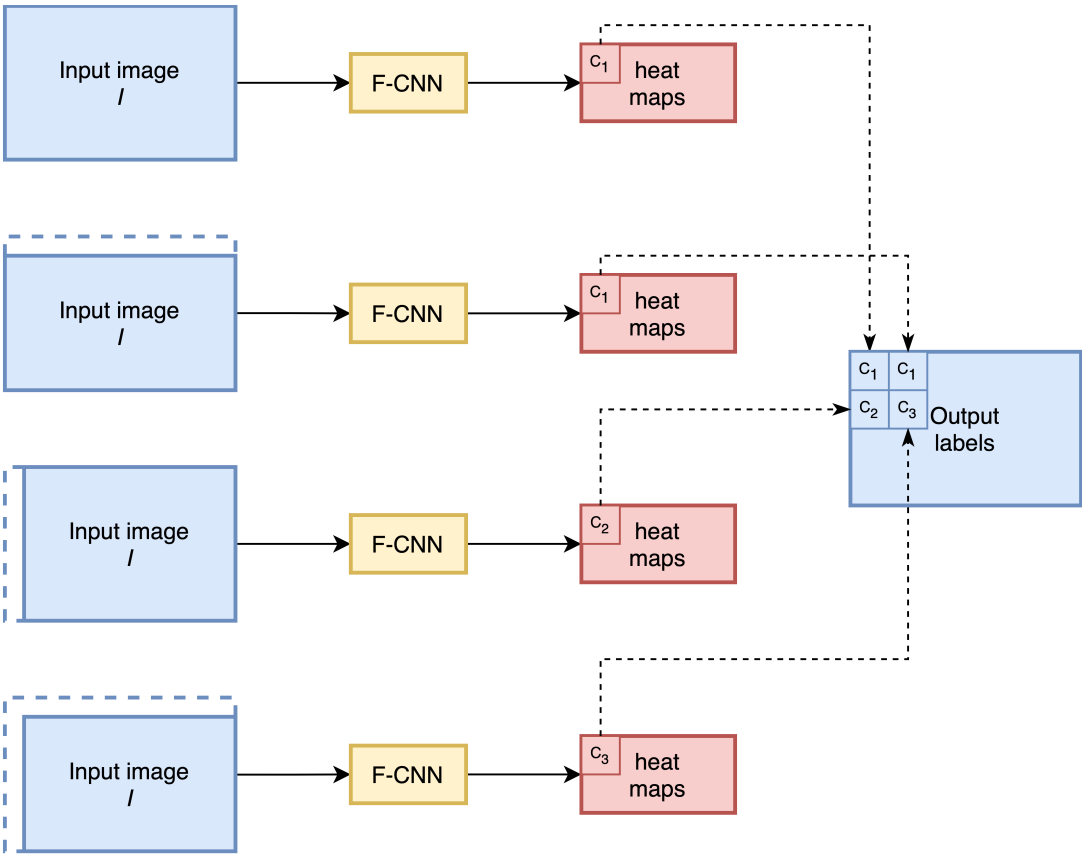


Figure 6.2: Shift and stitch up-sampling.

An example of shift and stitch up-sampling with $f = 2$ is shown in Figure 6.2.

The main advantages of this technique are:

- It exploits the whole depth of the network.
- It's possible to implement it efficiently through the à trous algorithm in wavelet. In particular, the network is trained with pooling, forcing a stride of 1. This means that no downsampling is performed. Moreover, the size of the following layer is increased after training.

However, it has some downsides, too. The main are:

- Up-sampling is rigid.
- It provides only a coarse estimate of the output image.

Only convolution networks

Another way for doing up-sampling is avoid doing any down-sampling. This can be achieved using no pooling layer. Basically, the network is made only of convolution layers (with activation functions). This solution works however it has many downsides:

- The output neurons have a narrow receptive field.
- It's not efficient since we have to compute convolution on images of the same size but with an increasing number of parameters.

6.2.2 Up-sampling block

Before analysing the block used to solve the up-sampling problem, let us analyse what are the challenges we have to face. When solving a segmentation problem, we want to:

- Go deep in the network to extract high level information on the input image.
- Stay local not to lose spacial resolution.

This is because:

- Global information finds out what's in the image.
- Local information finds out where is the subject.

In other words, semantic segmentation faces an inherent tension between semantics (what) and location (where).

Now that we have cleared this up, let us analyse what the basic architecture for solving a segmentation problem is and what novel block we need to implement it. A segmentation problem can be solved using a network made of two parts:

- A **down-sampling part**. This part reduces the spatial resolution of the image and classifies the output pixels like a fully convolutional network. This part is basically the classic CNN we have analysed so far.
- An **up-sampling part**. This part increases the spatial resolution of the first part's output. Namely, this network up-samples the predictions obtained through the first network.

Since we want to do two different operations, we require two different blocks:

- A block for down-sampling.
- A block for up-sampling.

We've already seen that down-sampling can be achieved with a pooling layer (typically max-pooling). On the other hand, we haven't seen any layer that can increase the spatial resolution (note that convolution can only increase the depth of an image or increase the resolution only with padding). This operation can be achieved in many different ways, some of which are:

- **Nearest neighbour**.
- **Bed of nails**.
- **Max unpooling**.
- **Transposed convolution**.

Nearest neighbour up-sampling

Nearest neighbour is the simplest way to perform up-sampling in the network architecture we have just discussed. If we want to increase the spatial resolution of a $R \times C \times 1$ (if we have more channels, we repeat this operation for each channel, but since the output of the first part has only one channel, we can focus on the single channel case) by a factor f, we can:

1. Create a $f \times f$ block for each pixel of the input, filled with the value of the corresponding pixel.
2. Create the output image with the $f \times f$ blocks, in the same position of their respective in the input image.

An example of nearest neighbour up-sampling is shown in Figure 6.3.

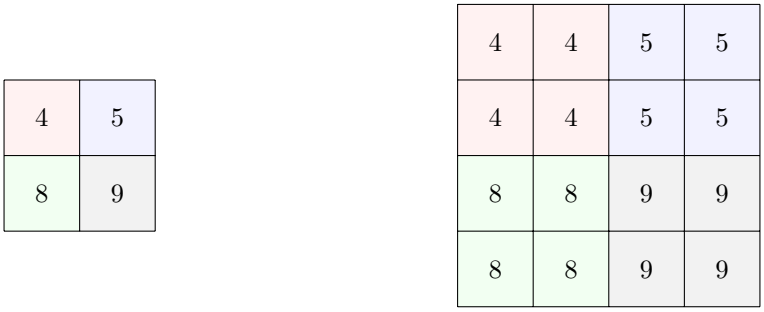


Figure 6.3: Nearest neighbour up-sampling.

Bed of nails up-sampling

Bed of nails up-sampling works similarly to nearest neighbour up-sampling. The main difference is that, instead of filling the $f \times f$ blocks with the value of the input pixel, we put in position $(0, 0)$ of each block the value of the pixel and we fill the other part with zeros. An example is shown in Figure 6.4.

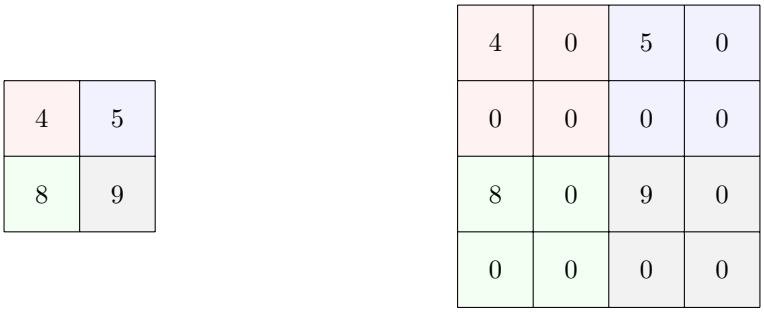


Figure 6.4: Bed of nails up-sampling.

Max unpooling

Max unpooling is one of the most popular ways to do up-sampling. Note that this technique is non-learnable, hence we don't have to train this part of the network. As for the techniques we've already analysed, a max unpooling block builds a $f \times f$ block for each input pixel. More precisely, a max unpooling layer keeps track, when performing max pooling, of the position of the max and puts the max back in its original position. The remaining part of the $f \times f$ block is filled with zeros. If we consider max pooling in Figure 5.4, the result of max unpooling, with respect to that pooling layer is shown in Figure 6.5.

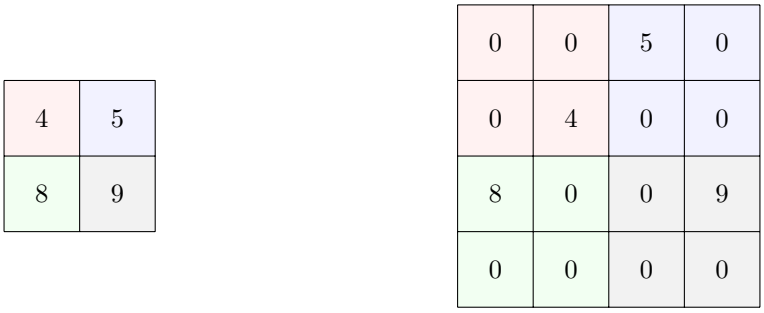


Figure 6.5: Max unpooling with respect to max pooling in Figure 5.4.

Transposed convolution

Differently from max unpooling, which doesn't need any learning, transposed convolution increases the size of the input using learnable filters. Transposed convolution roughly works as a reverse convolution in which the role of input and filter are swapped. Even if we use the filter as convolution input, we still have to learn the filter. In other words, it's like if the network was learning the input image of a convolution layer. As for normal convolution, transposed convolution can be applied with any stride and padding. More precisely, if we have an input image I and a filter f (that has higher resolution), the output O is computed as

$$O = \bigoplus_{(x,y) \in I} \text{pad}\big(I(x,y) \cdot F + b_{(x,y)}\big)$$

where

- $\bigoplus$ is the matrix sum.
- U is the neighbour of filter of size $f \times f$.
- $I(x,y) \cdot F + b_{(x,y)}$ is the matrix obtained multiplying each value of the filter F by $I(x,y)$ and adding $b(x,y)$ to each value.
- $\text{pad}(\cdot)$ is a function that takes a matrix and pads it with zeros in a way that its indices i are $0 \leq i < f$.

Note that some regions of the output image might overlap, hence the values of those regions are summed. As before we are considering only one channel since we are performing up-sampling after

classification and the first part of the network (the one for classification) returns a single channel image.

Note that, since filters can be learned, transposed convolution gives more degree of freedom. Also remember that this layer can be also called fractional strided convolution, backward strided convolution or deconvolution. An example of transposed convolution is shown in Figure 6.6.

As for normal convolution, transposed convolution can also be applied to images with multiple channels. The filter has the same number of channels as the input, in particular, if we have an input in $\mathbb{R}^{W \times H \times C}$, the filter will have dimension $\mathbb{R}^{f \times f \times C}$. By using D filters we can also modify the depth of the input from C to D. Long story short, a transposed convolution layer is able to do up or down-sampling (using D filters) and increase the size of the input (using convolution).

6.2.3 Skip connections

The architecture we've analysed until now is not very powerful. To increase its power we have to use skip connections. The idea behind skip connection is to use something of the down-sampling path in the up-sampling phase. In particular, if we have N down-sampling pooling layers, the first up-sampling layer is summed to the $N-1$-th down-sampling layer, the second with the $N-2$-th and so on. This is because the up-sampling factor is the same as the down-sampling factor, hence we can sum matrices with the same dimension.

Note that the up-sampling is performed using transposed convolution, hence the filters are learned. Moreover, the filters are initialised using bilinear interpolation.

An example of network that uses skip connections is the following. Consider a downsampling network made of convolutional layers alternated with pooling layers. After downsampling we create three different networks trained in sequence:

- The first network is trained.
- The second network is initialised with the weights of the previous network and deeper up-sampling blocks are summed, using skip connections, with some down-sampling blocks.
- The last network is initialised with the weights of the previous network using skip connections on shallower up-sampling blocks.

Finally, all fully connected layers are converted to 1×1 convolution layers.

6.2.4 U-net

U-net is an architecture used for segmentation. It starts from the architecture based on a down-sampling path followed by an up-sampling part and introduces some skip connections. The main characteristics of this architecture are:

- It uses a large number of feature channels in the up-sampling part.
- The network is symmetric.
- It uses excessive data augmentation by applying elastic deformations to the training images.

The network is made of a down-sampling part and of an up-sampling part. Each part is made of the same number of blocks so that the network is symmetric. The down-sampling network is made of blocks, each of which contains:

- A 3×3 convolution with ReLU activation, no padding and *valid* option.

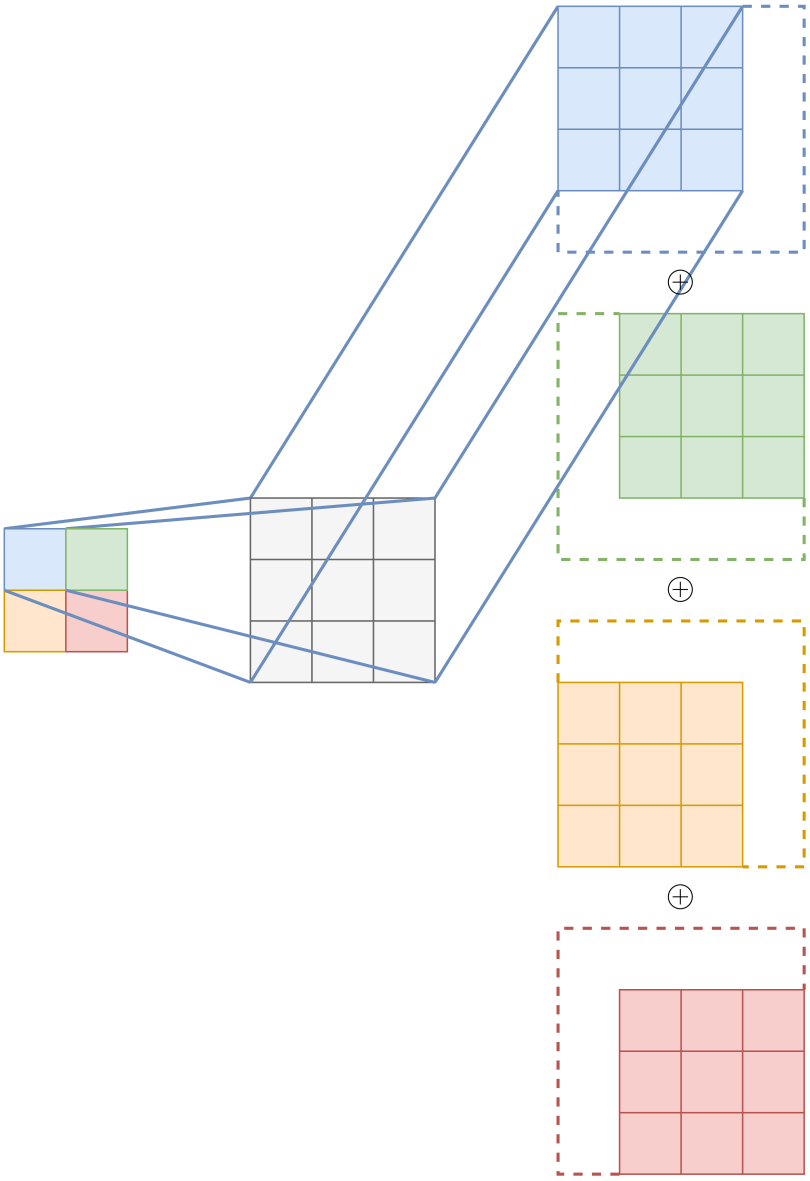


Figure 6.6: Transposed convolution.

- A 3×3 convolution with ReLU activation, no padding and *valid* option.
- A 2×2 max-pooling layer.

Each block doubles the number of feature maps. The blocks in the up-sampling part have the same components, but in reverse order. Namely, each block has

- A 2×2 up-sampling with transposed convolution that halves the number of feature maps but doubles the spatial resolution.
- A concatenation layer that concatenates the result of the up-sampling layer with the output of the max-pooling layer at the same level of the down-sampling network.
- A 3×3 convolution with ReLU activation, no padding and *valid* option.
- A 3×3 convolution with ReLU activation, no padding and *valid* option.

Note that this network doesn't have fully connected layers. L convolutions against $1 \times 1 \times N$ filters are use to predict the values out of the convolution feature maps. The output image is smaller than the input image by a constant border. The U-net architecture is shown in Figure 6.7.

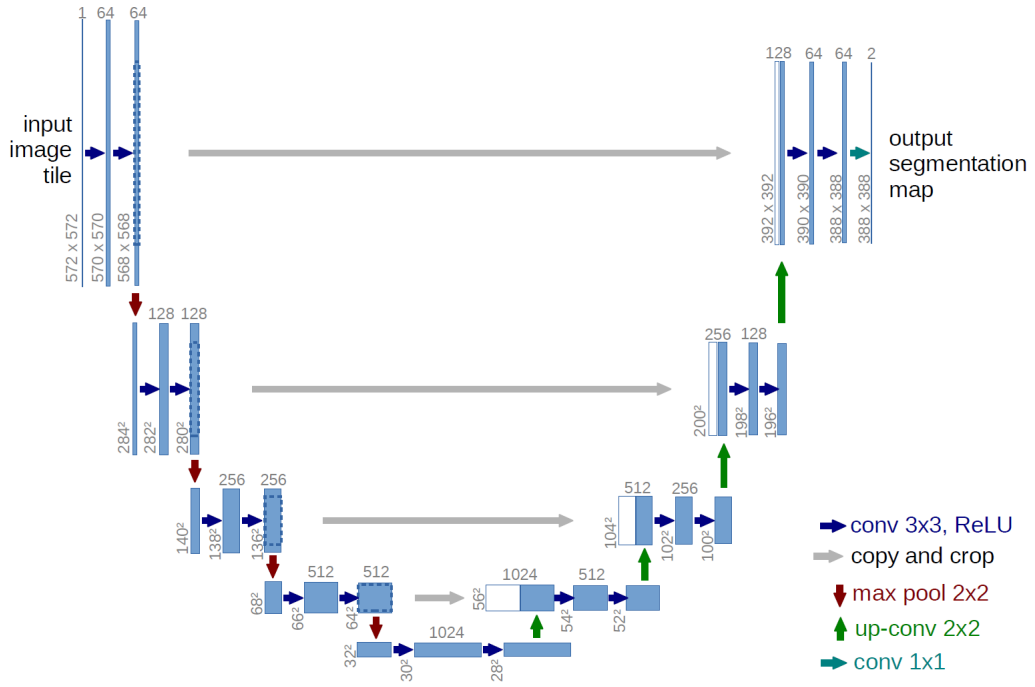


Figure 6.7: The U-net architecture.

Training

Training is performed using full-image training with a weighted loss function, therefore the parameters are computed as

$$\hat{\theta} = \min_{\theta} \sum_{x_j} w(x_j)\ell(x_j, \theta)$$

where the weights are computed as

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

where

- w_c is used to balance class proportions since we aren't using batch-based training.
- d_1 is the distance to the border of the closest cell.
- d_2 is the distance to the border of the second closest cell.

This means that a weight is large when the distance to the first two closest cells is small. This is because d_1 and d_2 are used as exponents with negative sign. Moreover, the distances are used to enhance the performance of the model at the borders of different objects.

6.3 Training

Let us discuss about training. If we want to train a network for segmentation, we need to annotate each pixel of each image in the data set. This means that the dataset is a set

$$\{(I, GT)\}$$

where

- I is the input image.
- GT is the image of labels, i.e., a matrix of pixels, each of which contains its class (i.e., its label for classification).

It's also worth noticing that the process of annotating each pixel of each image is very time consuming. COCO is an example of dataset that contains many labelled images. This dataset contains also annotations that are used for more complex problems.

6.3.1 Batch-based learning

One way to train a fully convolutional neural network is by using the patch-based approach. The idea is to divide the input image in smaller images (called patches) and label each patch with the class of the pixel at the centre of the patch. Patches can be chosen at random from the input images, but usually we impose that we have at least some objects to recognise and not only a background. In other words, patches that contain mainly the background are usually discarded. Note that, when using batch-based learning, we train the network as a fully convolutional classification network and then, only after training, we transform the weights in the fully connected layers to 1×1 convolutions so that we can work with images of arbitrary dimension.

The parameters of the network are trained to minimise the classification loss ℓ over a mini-batch $\mathbf{B}$, i.e.,

$$\hat{\theta} = \arg\min_{\theta} \sum_{x_j \in \mathbf{B}} \ell(x_j, \theta)$$

Note that the batches contain randomly chosen patches in which the image has been divided.

This approach is very inefficient since convolution on overlapping patches are repeated multiple times.

6.3.2 Full-image learning

Another technique for training a neural is full-image learning. In this case, instead of training the network on batches, we use directly the full size image and we train both the up-sampling and the down-sampling network. This means that, learning the parameters means computing

$$\hat{\theta} = \arg\min_{x_j \in I} \sum_{x_j \in R} \ell(x_j, \theta)$$

where x_j are all the pixels in a region R of the input image and the loss is evaluated over the corresponding labels in the annotation for semantic segmentation. Basically, each region provides already a mini-batch estimate for computing the gradient. Also note that full-image training is identical to batch-training where each batch consists of all the receptive fields of the units below the loss for an image.

In full-image training:

- The network is trained in an end-to-end manner to predict the segmented outputs.
- The loss is the sum of losses over different pixels. Derivatives can be computed through the whole network and the network can be trained through back-propagation.
- We don't need to pass through a classification network first.
- The network takes advantage of the fully connected convolutional neural network efficiency since it doesn't have to recompute the convolutional features in overlapping regions.

Full-image training is more efficient than batch-based training, however it has some limitations, too. In particular:

- Mini-batches in patch-wise training are assembled randomly but image regions in full-image training aren't. To make the estimated loss more stochastic we can compute the parameters as

  $$\hat{\theta} = \arg\min_{x_j \in I} \sum_{x_j \in R} M(x_j)\ell(x_j, \theta)$$

  where $M(x_j)$ is a binary random variable (i.e., 1 or 0).
- It's not possible to perform patch re-sampling to compensate for class imbalance (i.e., a class is more present in the data-set). To solve this issue we can weight the loss over different labels, hence the parameters should be computed as

  $$\hat{\theta} = \arg\min_{x_j \in I} \sum_{x_j \in R} w(x_j)\ell(x_j, \theta)$$

  where $w(x_j)$ is a weight that takes into account the true label of x_j.

Chapter 7

Localisation

7.1 The problem

Localisation tries to classify images and assign to the subject classified a bounding box, i.e., a square region of the image in which the object is fully contained. Say for instance we have images taken at a zoo, each of which contains only one animal, and we want to recognise the animal (animals are the classes) and the region of the image where it is. An instance is shown in Figure 7.1. Formally, the problem can be written as follows:

Definition 7.1 (Localisation)**.** *Given an input image $I \in \mathbb{R}^{R \times C \times 3}$ and a set of labels Λ, we want to find a mapping*

$$I \to (x, y, h, w, l)$$

where

- *l is the label, in Λ, corresponding to the class of the object in I.*
- *(x, y, h, w) are the coordinates (x and y) and the dimensions (height h and width w) of the bounding box enclosing the object.*

Note that this problem is different from segmentation since we don't want to classify each pixel, but we want to classify the whole image and then tell where the object we found is located in the image. Moreover, localisation classifies only one object differently from segmentation in which we could recognise different objects.

7.2 Basic convolutional neural network

The simplest way to solve the localisation problem is to use a classic convolutional network. The only difference with respect to a usual architecture is that the network should have two heads:

- One head that classifies the image, i.e., computes class l. This network uses dense layers and cross-entropy to compute loss.
- One head that computes the values x, y, h, w. These values are continuous, hence we can't use a classifier but we have to use a regressor with 4 outputs. This means that we don't have

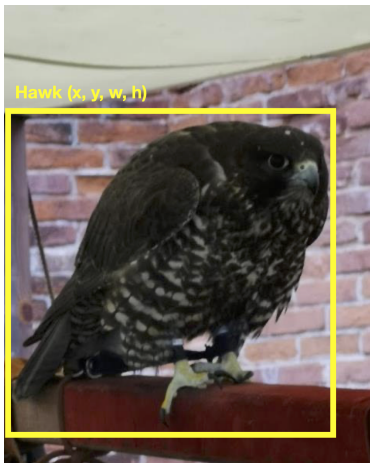


Figure 7.1: An image of an hawk classified and with its bounding box

to use cross-entropy as loss, too. An example of loss could be the l^2 norm of the difference between prediction vector $(\hat{x}, \hat{y}, \hat{h}, \hat{w})$ and the actual values (x, y, h, w), that is,

$$R(x) = \|(\hat{x}, \hat{y}, \hat{h}, \hat{w}) - (x, y, h, w)\|_2$$

The loss of the whole network has to be a single scalar since we compute the gradient of a scalar function with respect to the network parameters, however, we have two heads with different losses. We can compute the overall loss as convex combination of the two losses. In particular, if we call $S(x)$ the loss of the classifier and $R(x)$ the one of the regressor, the total loss for an input X can be computed as

$$L(X) = \alpha S(X) + (1 - \alpha)R(X)$$

where α isn't properly an hyperparameter of the network but a characteristic of the problem. This means that it might be tricky to tune it and we might need experimental guidance to set it (for instance using cross-validation). Also note that if $\alpha = 1$ we are basically solving only the classification problem whilst with $\alpha = 0$ we are solving only the regression problem.

Joint recognition

The same architecture can be used to recognise the joints of a human being. In particular, if we want to locate k joints, we have a $2k$ output vector (two coordinates for each joint). Moreover, we can have the network estimate joints that are not in the image because the network will predict the position of each of the k joints, even if some aren't in a specific image. In other words, the position of a joint is part of the output vector even if the joint isn't in an image.

7.3 Weakly supervised solutions

Localisation can be solved training a classification network without location-labelled data (basically, for each image, we only have its class). This result can be achieved revisiting the GAP layer since it has some localisation abilities. Moreover, a CNN trained on object categorisation is successfully able

to locate the discriminative regions for action classification as the objects humans are interacting with rather than human themselves.

7.3.1 Class Activation Mapping network

The basic network that uses a weakly-supervised solution and the GAP layer is called CAM. Such network is able to identify what regions of an image are used for discrimination. That is to say that a CAM network doesn't only recognise that an image contains a tree, but also what region of the image has been used to decide that it contains a tree. This means that the network can effectively understand where the tree is.

A CAM network is made of some convolution and activation layers completed by a GAP layer. The GAP layer returns a vector whose components are

$$F_k = \sum_{(x,y)} f_k(x, y)$$

where f_k is a feature map. Basically, each component F_k of the output vector is the average of a channel's pixels.

The output of the GAP layer is then connected to a dense layer, the output of which contains the probabilities of belonging to a class. More clearly, the output of the dense layer, which is the class probability P_c for each class c, is computed using the soft-max function, as

$$P_c = \frac{e^{S_c}}{\sum_{i \in C \setminus \{c\}} e^{S_i}}$$

where S_c is the score of class c, computed as the average of the outputs of the GAP layer, namely as

$$S_c = \sum_k w_{c,k} F_k$$

where $w_{c,k}$ is the weight associated to the connection between the input neuron F_k and the output neuron of class c. A summary of this part of the network is shown in Figure 7.2. Note that, when computing the score S_c of class c, the weight $w_{c,k}$ tells how important neuron F_k is for class c. Moreover, thanks to the softmax, the depth of the last convolutional layer can differ from the number of classes.

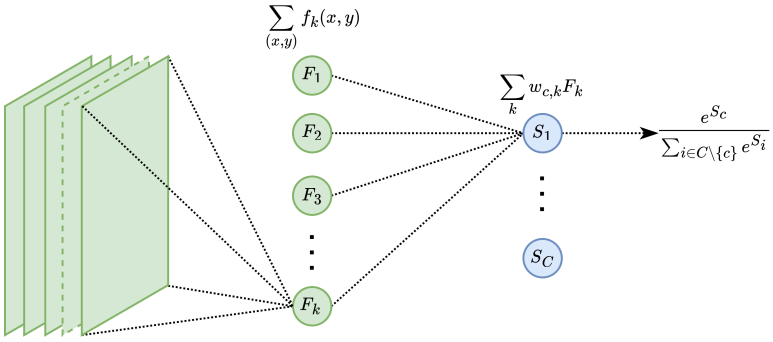


Figure 7.2: The final two layers of a CAM network.

From this description, we can write the score of class c as

$$\begin{aligned} S_c &= \sum_k w_{c,k} F_k \\ &= \sum_k w_{c,k} \sum_{(x,y)} f_k(x,y) \\ &= \sum_k \sum_{(x,y)} w_{c,k} f_k(x,y) \\ &= \sum_{(x,y)} \sum_k w_{c,k} f_k(x,y) \end{aligned}$$

Thanks to this new formulation we can express the score of one class in terms of its class activation mapping M_c, which is defined as

$$M_c(x,y) = \sum_k w_{c,k} f_k(x,y)$$

The class activation mapping of a pixel (x, y) for class c indicates the importance of the activations at (x, y) for predicting class c. Now that we know the concept of class activation mapping, let us understand what does the network do. Each channel of the last convolutional layer defines a map whose activations are stronger in the regions that allow to distinguish a certain feature. In other words, each channel of the last convolutional layer shows where in the image we can identify a certain feature. For each channel, this information is condensed in F_k, thanks to the GAP layer, hence each F_k contains the information of a certain map. Finally, the weights $w_{c,k}$ tell how important map k is for class c. To sum things up, the score is computed as the weighted sum (with weights $w_{c,k}$) of the maps defined by the last convolution's channels. Since each map defines where we can find a certain feature (which is used to recognise an object), the output contains the location of the most important features for assigning class c, hence the location of the object belonging to class c.

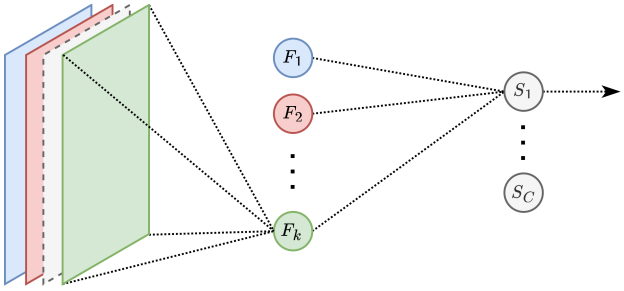


Figure 7.3: A CAM network with class activation mappings.

The network we described can be used with any pre-trained network (if we remove the classifier part) in fact we only need to add a global average pooling layer and a dense layer. Also note that the architecture is quite simple since we only require a few output neurons (as many as the number of classes) and one GAP layer but not hidden layer with many neurons. Note however that, if we use pre-trained models, the classification performance might drop since we are removing the part of the network (i.e., the classifier) with the most parameters.

Also note that the location accuracy, called CAM resolution, can improve by anticipating the GAP layer to larger convolutional feature maps. This however reduces the semantic information within these layers.

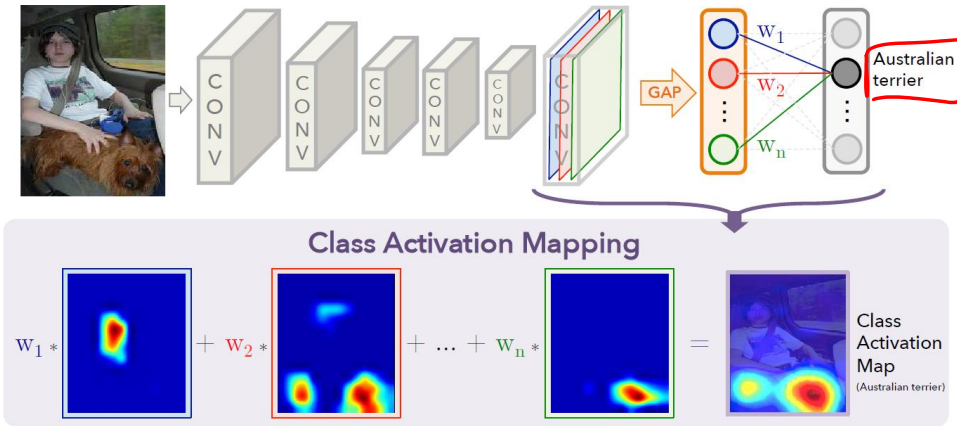


Figure 7.4: A CAM network with class activation mappings and heat maps.

Finally, one can also decide to use a Global Max Pooling layer which behaves similarly to a GAP layer, with the only difference that F_k is the maximum value of channel k. The main differences between these layers are:

- GAP encourages the identification of the whole object, as all the parts of the values of the activation map concur to the classification of the input.
- GMP promotes specific discriminating features.

7.3.2 Grad-CAM network

The Grad-CAM architecture derives from the CAM architecture. In particular, Grad-CAM doens't require us to modify the output, hence we don't need to plug in a single fully connected layer at the end. Grad-CAM computes low resolution heat-maps, using augmentation, and then combines them with a super-resolution algorithm to obtain an high resolution image. In other words, given an image, the model produces different heat maps referring to transformed version of that image (e.g., rotation, scaling) and then combines them to get an higher resolution heat map, which is more accurate. A visual representation of a grad-CAM network is shown in Figure 7.5. In the image, we compute $\mathbf{g}^c$ as

$$\mathbf{g}^c = ReLU\left(\sum_k w_k^c \mathbf{a}_k\right)$$

and the weights as

$$w_k^c = \frac{1}{nm}\sum_i \sum_j \frac{\partial y_c}{\partial \mathbf{a}_k(i,j)}$$

7.3.3 Saliency maps

Each saliency map is related to one class. If we compute the saliency map for the right class we get a correct heat-map, otherwise we get something completely wrong. When the network is wrong, saliency maps aren't very informative. Hence the idea was to provide, together with saliency maps that say where the object is, an intuit on of what the network can see. This can be estimated

taking the latent representation before the final classification and use it to reconstruct the input by transposedconvolution. What we get is a combination of what the network sees and where it sees it. This can be useful to check if it's looking in the right places for the right features. This architecture is shown in Figure 7.6[1].

[1] https://arxiv.org/pdf/2204.09920.pdf

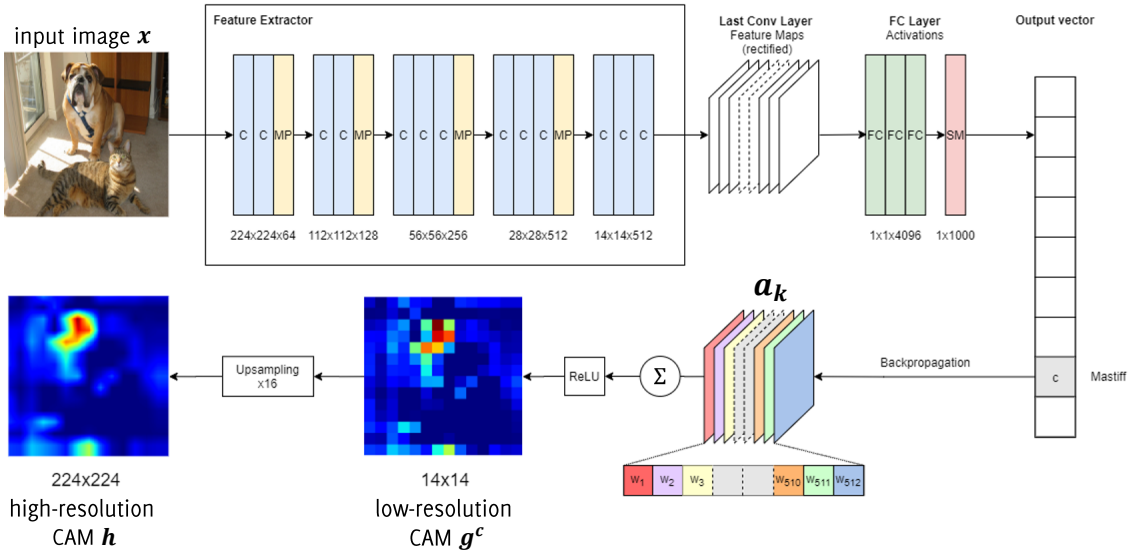


Figure 7.5: A grad-CAM network.

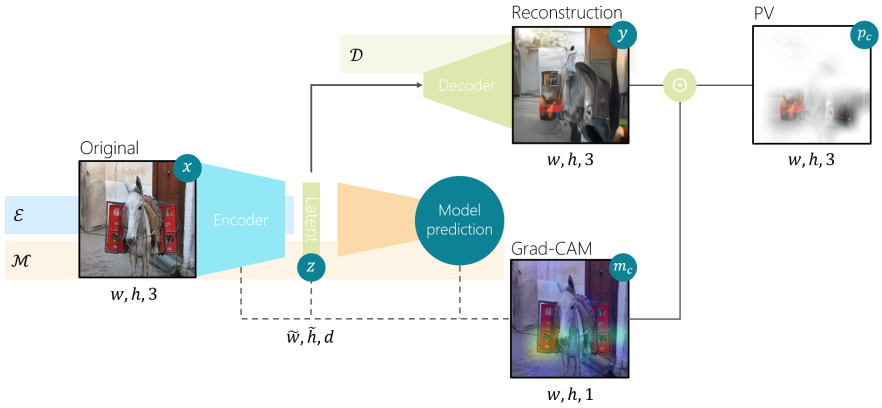


Figure 7.6: The architecture for reconstructing the input image using saliency maps.

Chapter 8

Object detection

8.1 The problem

Object detection is an expansion of localisation since we want to localise (i.e. classify and find the bounding box) multiple objects in an image. Formally, the problem can be described as follows:

Definition 8.1 (Object detection)**.** *Given an input image $I \in \mathbb{R}^{R \times C \times 3}$ and a set of labels Λ, we want to find a mapping*

$$I \to \{(x, y, h, w, l)_1, \dots, (x, y, h, w, l)_N\}$$

where $(x, y, h, w, l)_i$ is an object found by the network and

- *x, y, h and w are the coordinates and the size of the object's bounding box.*
- *$l \in \Lambda$ is the class (or label) of the object.*

Since we have multiple objects in an image, each point in the data-set contains:

- The image I.
- A list of bounding boxes and their labels.

8.2 Sliding windows

The first solution one can think of is a network that:

1. Takes a large image.
2. Crops the image, i.e., considers only a squared region of the image.
3. Predict the output of the cropped image using a CNN.

Note that we, if we use a pre-trained network, we might have to crop the image because the CNN might have a different input size then the one of the input image. This architecture has many problems, in fact:

- It's very inefficient since it doesn't re-use features common to different overlapping crops.
- It's hard to choose an appropriate crop size.
- It's difficult to detect objects that have different scales since we have only one crop size.

The only advantage is that:

- We can use a pre-trained network.

8.3 Region proposal

Region proposal algorithms (and networks) are able to extract bounding boxes from an image using gradient and colours. This means that we can:

1. Use a region proposal algorithm (or network) to extract region proposals for candidate objects. The algorithm returns a set of rounding boxes that might contain an object. Note that these algorithms were known before neural networks and have a very high recall but law precision.
2. The bounding boxes are classified by a classification network.

Note that the output classes should also contain a background label to classify regions where we have no object. This is the general framework. Let us now analyse some specific solutions.

8.3.1 R-CNN

An R-CNN network:

1. Selects interesting bounding boxes using an external region proposal algorithm.
2. Classifies the regions returned by the region proposal algorithm. In particular:
   (a) It resizes the region to a fixed size.
   (b) It uses a pre-trained network for feature extraction.
   (c) It classifies the region with a support vector machine.
   (d) It predicts the bounding box with a box regression network.

R-CNN works better than the solution with sliding windows, however, it has some downsides, too. In particular:

- It's slow at inference since it has to evaluate many candidate regions.
- It's bulky and slow at training.
- It uses an external region proposal algorithm which doesn't exploit neural networks.
- It has ad-hoc training objectives, hence we can't use an end-to-end training with backpropagation like we did for classical CNNs.

Note that the fact the network is slow at inference is very important since these networks are usually adopted for live video analysis (each frame is analyses as an image), hence they should be fast.

8.3.2 Fast R-CNN

The fast R-CNN architecture manages to obtain candidate regions using an external region proposal algorithm but mapping them directly and in parallel to feature vectors. The network works as follows:

1. A **deep convolutional network** extracts the features from the input image (i.e., it computes the image's feature map).
2. An **external region proposal algorithm** selects a region of interest (ROI) in the image. The ROI is an sub-image.
3. A **Region Of Interest projection layer** is used to project regions into feature maps of a fixed size (since candidate regions might have arbitrary sizes). This allows to re-use convolutional computation. in fact regions extracted from the external region proposal algorithm are directly mapped into feature vectors by the ROI projection layer.
4. A **Region Of Interest maxpooling layer** is used to extract a feature vector of size $H \times W$ from the feature maps. This is required since we need to have a vector of fixed size to feed the fully connected layers.
5. Two heads, made of fully connected layers, are used to predict the class and the bounding box of the input image.
6. The loss is computed as a convex combination of the losses of the two heads and learning is done with back-propagation on the whole network since we have a single loss.

This network is faster than a classical R-CNN, especially at inference, since common features are used on every region. Most of the time is spent on the ROI extraction part.

8.3.3 Faster R-CNN

The faster R-CNN architecture addresses the problem of finding candidate regions. In particular, this model replaces the external region proposal algorithm with a Region Proposing Network (RPN), which is a neural network for providing region proposals. One of the advantages of the RPN is that it uses the same feature maps used for classification. The general architecture is divided in two parts:

1. Feature extraction and region proposal. This part contains:
   (a) A **backbone network** (e.g., VGG16) to extract features.
   (b) A **Region Proposal Network** to estimate the regions of interest. The RPN usually produces around 300 regions of interest.
2. The classification part, which contains:
   (a) A **ROI pooling layer** to crop features.
   (b) A **fully connected layer** with softmax activation to predict the class of the input.
   (c) A **fully connected layer** with softmax activation to predict the bounding box offset. This network is used to increase the localisation capabilities of the model.

As we can see, the main difference with respect to fast R-CNN is the Region Proposal Network.

Region Proposal Network

The core of the faster R-CNN architecture is the Region Proposal Network. An RPN is based on anchor boxes, which are Regions Of Interests with different scales and ratios (but same size). The goal of the RPN is to associate to each spatial location, k anchor boxes with different scales and ratios. This means that the RPN outputs an $H \times W \times k$ matrix that represents the k candidates of size $H \times W$, each related to its estimate objectiveness score. In other words, the anchor boxes are fixed (i.e., they are like classes in a classification network) and the RPN has to evaluate which anchor box is better suited (or, better said, what is the probability that an anchor box contains the subject to classify) for a certain region of the image. The RPN network doesn't work on the image itself but on the feature vector extracted by the backbone. In general, a RPN is made of:

1. The **input layer**. This layer is the output of the previous feature-extraction network. The input layer is scanned using a sliding window of size $H \times W$.
2. An **intermediate convolution layer**, which is a standard convolution layer. This layer takes as input a region of size $H \times W \times D$ (obtained with sliding windows) and uses 256 filters of size 3×3 to reduce the depth of the input feature map. In particular, it maps each region to a lower dimensional vector of size $H \times W \times 256$.
3. An **output classification layer**. This layer has $2k$ neurons, two for each candidate anchor box. Each couple of neurons contains:
   - The probability p_i that the anchor box i contains an object (whatever object).
   - The probability $1 - p_i$ that the anchor box i doesn't contain an object.

   The output of this layer is obtained using convolutional layers of size 1×1.
4. An **output regression layer**. This layer is trained to estimate, for each anchor box i, the coordinates of the bounding box related to i. In other words, this layer adjusts the anchor box to better match the correct bounding box. Since this layer predicts, for every anchor, the coordinates of the bounding box, it outputs $4k$ values (i.e., coordinates). Finally, some region proposals are discarded using a non-maximum suppression based on the objectiveness score.

A RPN has one big advantage. If we want to predict different anchors, we simply have to define different labels, when training the RPN, associated to different anchors.

Training

Training involves 4 losses (two from the RPN, two from the classifier) which are:

- The RPN classifier loss that measures how correct is the probability that an anchor box contains an object. This loss is computed, using the intersection over the union, which measures, given two regions the ratio between the intersection of the two and their union.
- The RPN regression loss that measures how different the bounding box is from the actual bounding box containing an object. This loss is also computed using the intersection over the union.
- The final classification score, computed using categorical cross-entropy.
- The final bounding box coordinates and dimensions using categorical cross-entropy.

Training is therefore split into multiple phases:

1. Train the RPN keeping the backbone network frozen and training only the RPN layers. This means that we only trying to learn where bounding boxes are in the image.
2. Train the fast R-CNN using proposals from the RPN we've just trained.
3. Fine-tune the fast R-CNN and the backbone network.
4. Fine-tune the RPN with unfreezed backbone network.
5. Freeze the backbone network and the RPN and fine-tune only the last layers of the faster R-CNN.

Inference

Inference is done in two steps:

- The backbone network does feature extraction to obtain candidate regions.
- The classifiers labels the regions obtained at the previous step.

8.4 YOLO

The YOLO (You Only Look Once) network uses a different approach with respect to region-proposal architectures. The basic principle of working isn't that different with respect to R-CNN (and it's evolutions) but the former uses only one step, i.e., it doesn't use a region proposal algorithm. In particular, YOLO:

1. Divides the image in tiles of size 7×7.
2. Assigns B anchors to each tile.
3. Predicts
   - the offset of each anchor box (from the anchor assigned) and,
   - the score of each anchor box over the C categories (including the background).

   The output is therefore a $(7 \times 7 \times B \times (5 + C))$ matrix since we have B boxes, of size 7×7, each with the x, y, height and width offsets (hence, 4 values), a score for each category (hence, C scores) and the objectiveness score.

The resulting network is very large, however, since it requires a single step for prediction, it's also fast (definitely faster than region-proposal-based approaches) at inference. On the hand, YOLO is not as accurate as models that use a region-proposal algorithm (or network).

Chapter 9

Instance segmentation

9.1 The problem

Instance segmentation is a combination of semantic segmentation and object detection. The idea is to do object detection but, instead of finding the bounding box for each object, we want to classify the pixels belonging to a certain instance of a class. In other words, instance segmentation is a case of semantic segmentation in which we recognise different instances of a class. As an example, if we have an image with two dogs and one cat, instance segmentation assigns to each pixel label d_1 if it belongs to the first dog, label d_2 if it belongs to the second dog, label c if it belongs to the cat or label b if it belongs to the background class. Formally,

Definition 9.1 (Instance segmentation)**.** *Given an input image $I \in \mathbb{R}^{R \times C \times 3}$ and a set of labels Λ, we want to find a mapping*

$$I \to \{(x, y, h, w, l_i, S)_1, \ldots, (x, y, h, w, l_j, S)_N\}$$

where

- *l_i is a label from Λ corresponding to the i-th instance of an object with label $l \in \Lambda$.*
- *$(x, y, h, w)_i$ are the coordinates (x and y) and the dimensions (height h and width w) of the bounding box enclosing an instance i of the image. Basically, we want to find an object and then we want to draw a box that contains the whole object.*
- *S is the set of pixels, in each bounding box (x, y, h, w), corresponding to label l.*

9.2 Mask R-CNN

Since the problem combines two problems, we can try and combine the solutions to such problems. In particular, the mask R-CNN model combines:

- A faster R-CNN for object detection.
- A image segmentation network for doing semantic segmentation on the objects classified by the fast R-CNN network.

This result is obtained adding a branch to the faster R-CNN architecture which will have three tails (after the feature and ROI extraction part):

- An head to predict, for each class, the score of each region of interest.
- An head to predict, for each region of interest, the bounding box location and dimensions.
- An head to predict, for each region of interest, the mask used to filter only the pixels that belong to the object (i.e., to filter out background pixels).

Chapter 10

Metric learning

10.1 The problem

Let us consider the following problem to understand what metric learning is. Say we want to build a system that allows recognised people to enter a room, given an image of their face. This system should, given a list of photos of authorised people's faces,

1. Take a picture of the person that want to enter the room.
2. Check if the face in the picture is the same as one of the authorised people.

10.2 Simple classification network

The easiest way to solve this problem is to implement a simple classification network that has a class for each authorised person. This solution is rather simple to implement and exploits everything we've seen so far about CNNs for classification. Note that if we want to have a good network we require a good number of images for each authorised person, ideally in different light conditions.

This solution works however it doesn't scale well, in fact, if we want to add a new person to the list of authorised ones, we have to retrain the whole network. For this reason, this isn't the way to go.

10.3 Image comparison

Since we can't use a network in which each person represents a class, we have to switch to another strategy. A good idea is to try and measure the distance between two images to tell if one is similar to the other. Namely, given the face of a person, we want to check if it's close to one of the images in the list of authorised people. Note that, in this case we don't need a large number of images for each person but only one. The image of an authorised person is called **template**. Also note that this method scales really well since, if we want to add a new authorised person, we only have to add a template for such person.

Before going on, let us highlight an important concept. The solutions we are going to analyse use a CNN to extract the input image's features. The output of the feature-extraction CNN is sometimes called **latent representation** since it's a low dimensional, compressed representation

of the input image. The latent representation contains the features of the image and uses them to describe the input image.

Now that we have understood the general framework used, we have to define a way to compute the distance between two images.

10.3.1 Feature distance

Let us consider a network made only of a feature-extraction CNN that extracts the latent representation of the input image. This network can be used for template images T_i or for images I of people that have to be classified. This means that, if we feed the CNN with an image I and a template T_i, we get two vectors and we can compute their distance using ℓ^2 norm, i.e., we can compute

$$\|f(I) - f(T_i)\|_2$$

where $f(I)$ is the latent representation of the input I, i.e., the vector obtained feeding the feature-extraction network with image I.

Now that we know how to compute the distance between two images, we can assign an input I to the class that minimises the distance between I and a template, i.e.,

$$\hat{i} = \arg \min_{i=1,\dots,C} \|f(I) - f(T_i)\|_2$$

Note that, we don't need to compute $f(T_j)$ for every class j since these values can be pre-computed and stored.

10.3.2 Nearest-neighbour

The distance between latent representations can be used in a K-nearest-neighbours model, which is an unsupervised, non-parametric classification model. More precisely, given a point I to classify, if the majority of the K closest points to I are labelled with class C, then I is labelled with class C. This means that the complete model we will use is made of:

1. A feature-extraction CNN to extract the input's features.
2. A K-nearest-neighbour classifier.

10.3.3 Siamese network

The problem with computing the distances between latent representations is that we are computing the distance between latent representations extracted by a network that could be trained on a small number of images. Siamese networks try to solve this problem using a network that learns to compute the distance between images. In particular, a Siamese network is made of two identical feature-extraction CNNs, with same weights W, and fed with two different images I_i and I_j (one for each network).

Training

The idea is to feed one network with image I, to obtain $f_W(I)$ and the other with a template T_i, to obtain $f_W(T_i)$. Now that we have the values of latent representations, we can find the template $T_{\hat{i}}$ with minimum distance from I and assign I to class $\hat{i}$. Having understood how does the network

work, we have to analyse the learning process. We want the two networks to learn a function f_W (the same for both) such that

$$\|f_W(I) - f_W(T_i)\|_2 \leq \|f_W(I) - f_W(T_j)\|_2 \quad \forall i, j \in \{1, \dots, C\}, i \neq j$$

if image I belongs to class i. However, we have to deal with the fact that a Siamese network is made of two identical networks, fed with different images, but that have the same weights. This means that we can't use categorical cross-entropy as loss function. Instead, we have to use contrassive loss, defined as

$$\mathcal{L}_w(I_i, I_j, y_{i,j}) = \frac{1 - y_{i,j}}{2} \|f_w(I_i) - f_w(I_j)\|_2 + \frac{y_{i,j}}{2} \max\left(0, m - \|f_w(I_i) - f_w(I_j)\|_2\right)$$

where

- I_i and I_j are the input images of the Siamese network.
- $y_{i,j}$ is the label associated to images I_i, I_j. This value is
  - 0 if the images refer to the same person.
  - 1 otherwise.
- m is an hyperparameter indicating the margin we want (like in Hinge loss).

This means that we feed the network with couple of images and we tell it if the images represent the same person or not. In particular,

- If the images I_i and I_j belong to the same person we have $y_{i,j} = 0$ and only the first term is considered for computing the loss. The loss is therefore the distance between the two images since if the images refer to the same person, the further the images are, the worst.
- If the images I_i and I_j refer to different people we have $y_{i,j} = 1$ and only the second term is considered for computing the loss. The loss is therefore $m - \|f_w(I_i) - f_w(I_j)\|_2$ (or 0 if this value is negative) which is big when the distance is small (if the distance is 0, the loss is m, if the distance is m, the loss is 0). This makes sense because if the images refer to different people, we want the distance to be big, hence the loss has to be big when the distance is small.

The parameters of the network are therefore obtained as

$$W = \arg\min_w \sum_{i,j} \mathcal{L}_w(I_i, I_j, y_{i,j})$$

Note that this solution might look like the previous one but we have a radical difference. In the former case, the classifier was trained over generic images of faces whilst in this case we are comparing images directly, hence the latent representation is learned comparing images (and not learning the features of a face).

Inference

During inference, when we want to say if the an image depicts an authorised person, we can

1. Feed the input image I of the person that requests access to one of the networks (they are the same) and get the latent representation $f_W(I)$.

2. Identify the authorised person as the one with minimum distance from the image I, i.e.,

$$\hat{i} = \arg\min_u \frac{\sum_{T_{u,i}} \|f_W(I) - f_W(T_{u,i})\|_2}{|\{T_u\}|}$$

3. Give access if

$$\frac{\sum_{T_{j,i}} \|f_W(I) - f_W(T_{j,i})\|_2}{|\{T_u\}|} \leq \gamma$$

10.3.4 Triple Siamese loss

A triple Siamese network inherits the structure of a Siamese network but, instead of providing positive and negative example alternatively during training, we provide both. In particular, the network is made of three networks:

- One fed with the image I of the person to recognise.
- One fed with a positive example P, i.e. the template of the same class of I. Namely, we put the same person as the one shown in I.
- One fed with a negative example N, i.e. the template of one of the other classes. Namely, we put a different person than the one shown in I.

In this case, the loss becomes:

$$L_w(I, P, N) = \max\Big(0, m + \big(\|f_w(I) - f_w(N)\|_2 - \|f_w(I) - f_w(P)\|_2\big)\Big)$$

This loss, called triple loss, forces that a pair of samples from the same individual are smaller in distance then those with different ones. Moreover, m plays the role of the margin.

Chapter 11

Autoencoders

11.1 Multi-layer perceptron autoencoders

Autoencoders are unsupervised models used for data reconstruction. An autoencoder has

- An input $\underline{s}$ of size I.
- One (or mode) hidden layer.
- An output $\underline{o}$ of size I (same as the input).

The goal of the network is to extract and encode the features of the input $\underline{s}$ and then reconstruct, from the latent representation of the input, the output $\underline{o}$, such that

$$\|\underline{s} - \underline{o}\|_2$$

is as small as possible. Namely, we want to minimise the distance between the input $\underline{s}$ and the reconstructed output $\underline{o}$. As a result, the network can be divided in two parts, each of which computes a different function:

- The **encoder** tries to find a new representation $\mathcal{E}(I)$ of the input I. The encoder computes the encoding function $\mathcal{E}$. The output of the encoder is usually called **latent representation**.
- The **decoder** tries to reconstruct (and output) the input from the latent representation built by the encoder. The decoder computes the decoding function $\mathcal{D}$

The general architecture of an autoencoder is shown in Figure 11.1.

Ideally, we would like the network to learn the identity function (i.e., $\mathcal{D} \cdot \mathcal{E} = Id$) so that the output is identical to the input. Unfortunately, neural networks aren't good at learning the identity function. In particular:

- If the dimension of the hidden layer is bigger than the one of the input layer, the network can map some inputs to the hidden layer and propagate the activations to the output layer.
- If the dimension of the hidden layer is smaller than the one of the input layer, the network can't map directly the input to the hidden layer, hence it has to compress the information of the input layer to obtain a lower dimensional representation of the input. This representation is then expanded to recover the input. Note that we aren't guaranteed to precisely reconstruct the input since the output (i.e., the reconstructed input) is obtained from a compressed representation of the input.

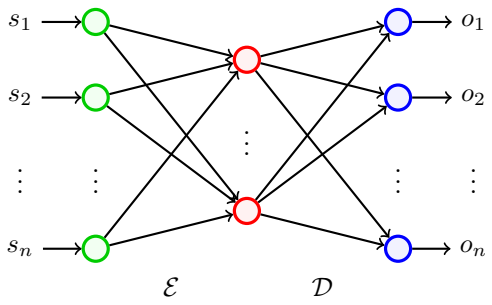


Figure 11.1: An autoencoder network

We are interested in the second option, since we want to compress and encode the information of the input in a low-dimensional representation. To achieve a network that can build a good representation of the input we have to minimise the distance between the input $\underline{s}$ and the output of the network, i.e.,

$$\ell = \|\underline{s} - \mathcal{D}(\mathcal{E}(\underline{s}))\|_2$$

Given this loss, we want to minimise its sum over the dataset's batches, namely we want to minimise

$$\sum_B \ell$$

As we can see, differently from supervised learning methods, we don't require annotations since we can use the input as comparison for the output. This means that this is an unsupervised learning model.

Note that, in some cases, we can add a regularisation term $\lambda\mathcal{R}(s)$ to the latent representation in order to satisfy some properties (e.g., sparsity).

11.1.1 Convolution autoencoders

Autoencoders can also be applied to image processing. In particular, we can adopt the same architecture with:

- An **encoder** that computes the latent representation of the input image. The encoder is implemented using one or more **convolutional layers** with ReLU activations.
- A **decoder** that reconstruct the input image from its latent representation. The decoder is implemented using one or more **transposedconvolutional layers** with ReLU activations.

11.1.2 Data augmentation

Autoencoders can be used to help training when we have a small dataset. Say we have a dataset $L = \{(s, y)_i\}$ with correct annotations and another big dataset $S = \{s_i\}$ with no or wrong annotations (which is like having no annotations at all). We would like to use S, being big, for our problem even if it has no annotations. To solve this problem we can

1. Take a classic architecture with a feature-extraction CNN and a classifier with dense layers.
2. Remove the classification network and add a decoder. Basically, we have built an autoencoder since the feature-extraction network is an encoder.

3. Train the whole network with the feature extractor and the decoder using both datasets. After training, the latent representation has meaningful features because the network has learned to reconstruct the input, for which it requires a latent representation with meaningful features. The key is that we have used a lot of images and we have avoided overfitting (which we would have obtained using only L). Note that we can use both datasets since we don't need annotations to train an autoencoder.

4. Remove the decoder and add a simple fully connected layer to classify the images in L.

5. Train the new network using the small dataset L.

Long story short, we have used the big dataset to learn the most difficult task (i.e., feature extraction) and then used a simple network to solve classification.

Chapter 12

Generative models

12.1 The problem

Generative models try to generate artificial images that look like real images. Basically, we provide real images to the model and we let it generate images that look like those provided. Let us now analyse this problem as a search in the space of all images. Images are points in a space of dimension $H \times W \times 3$ and we want to find the region of this space in which the images we are interest in lay. For instance, if we want to generate images of human faces, we have to find the region of space where images of human faces are. The space of interest is called **manifold**. Describing a manifold is very difficult, especially if we consider that images lay in a space with huge dimensions.

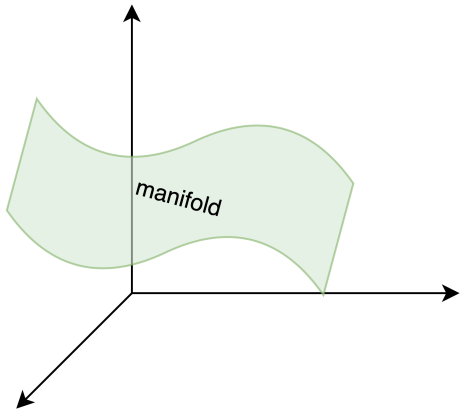


Figure 12.1: A manifold in the space of images.

12.1.1 Usages

Generative models can be used for:

- **Super-resolution**, namely for increasing the resolution of an image.

- **Data augmentation**, in fact we can generate images similar to the ones in the dataset hence increasing the number of images we can use for training.
- **Latent representation**.
- **Anomaly detection**.
- **Regularising the prior** in other problems.

Also note that some techniques used for solving this problem are giving birth to new training paradigms like adversarial training.

12.2 Autoencoders

The first solution one might think of is of using an autoencoder. In particular we can

1. Train an autoencoder on S.
2. Discard the encoder.
3. Pick random vectors $z \in \mathbb{R}^d$, with the same dimension d of the latent representation, from a random distribution ϕ_z.
4. Feed the network with z and let the decoder generate an image from z.

It's however difficult to learn the distribution ϕ_z of the latent representation to provide as input to the encoder.

12.3 Generative Adversarial Networks

As we have understood, it's impossible to learn the distribution of the latent representation and the same holds for the distribution of the manifold S. Namely we can't learn neither ϕ_z nor ϕ_S. This is where GANs come into play. A GAN fixes a certain input distribution ϕ_z, usually called **noise**, (e.g., a Gaussian distribution) in a small space (definitely smaller than the image space, e.g., $\mathbb{R}^{100}$) and learns to map that distribution to the image space. That is to say, a GAN

1. Draws some samples $z \in \phi_z$.
2. Feed z into the network, which transforms z to generate an image s which looks like sampled from the distribution ϕ_S of the manifold.

The transformation made by the GAN is learned thanks to a neural network. The network is trained in an unsupervised way, hence we don't need a labelled dataset.

12.3.1 General structure

Let us now analyse a basic structure for a GAN, which will then be refined. Let us consider a convolutional neural network with transposedconvolution, since we want to start from a low dimensional latent representation (in $\mathbb{R}^d$) and generate an image in $\mathbb{R}^{H \times W \times 3}$, with $d \ll H \times W \times 3$. Long story short, we need transposedconvolution since we want to increase the image size. Let us now feed such network with a random sample z drawn from the noise distribution ϕ_z. The network takes the input z and generates an image with higher resolution in the space $R^{H \times W \times 3}$.

12.3.2 Learning

Now that we have the architecture, we want to understand how to learn a good transformation function, that is to say a function able to map samples from the input distribution ϕ_z into samples in the manifold distribution ϕ_S. Note that in general a GAN maps samples in ϕ_z into samples in $\mathbb{R}^{H\times W\times 3}$, however we want only a subset $S \subset \mathbb{R}^{H\times W\times 3}$ that contains images of the type we want to generate. Learning requires a loss, which tells how realistic an image is. We can't however use a closed form solution, hence we have to find another way to assess whether an image is realistic or not. In particular, we can train another network whose purpose is only to say if an image generated by the image generator is realistic or not and is trained to distinguish if an image is real or not.

This means that a GAN is made of two networks:

- A generator G that:
  - Takes as input a random vector from the distribution ϕ_z.
  - Outputs a randomly generated image.
  - Uses a decoder as architecture.

  The generator is trained to fool the discriminator.
- A discriminator D that:
  - Takes as input an image generated by G or from the manifold S.
  - Outputs 1 if it thinks the image is real or 0 if it thinks the input image is fake.
  - Uses a CNN encoder (we want to reduce the size to 2 output neurons).

  The discriminator is trained to recognise if the image in input has been generated by G or is a real one.

Ideally G should win since we want it to fool D into thinking that an image it has generated is a real image.

Let us notice an important fact: the generator G has never seen true images but learns how to generate them thanks to the discriminator in fact

- G is rewarded when it can fool D.
- G is penalised when it can't fool D.

This is very different from autoencoders since they have to see images to learn how to build a good latent representation. Let us now formalise the two networks. The generator network G can be modelled as a function

$$\mathcal{G}(z, \theta_G) : \mathbb{R}^d \to \mathbb{R}^{N\times N}$$

in which

- $z \in \phi_z$ is a sample drawn from the random distribution ϕ_z.
- θ_G is the set of parameters of the generator.

The discriminator can be described as a function

$$\mathcal{D}(s, \theta_D) : \mathbb{R}^{N\times N} \to \{0, 1\}$$

where

- $s \in \mathbb{R}^{N \times N}$ is the input image.
- θ_G is the set of parameters of the discriminator.

Note that the two functions take only one input and the parameters are shown only to represent that we are talking about a neural network.

The discriminator wants:

- $\mathcal{D}(s, \theta_D) \approx 1$.
- $\mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D) \approx 0$, namely, $1 - \mathcal{D}(G(z, \theta_G), \theta_D) \approx 1$.

Namely, D wants to maximise

$$E_{s \sim \phi_S}\Big[\log\big(\mathcal{D}(s, \theta_D)\big)\Big] + E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]$$

This means that the discriminator computes

$$\hat{\theta}_D = \max_{\theta_D}\left(E_{s \sim \phi_S}\Big[\log\big(\mathcal{D}(s, \theta_D)\big)\Big] + E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]\right)$$

On the other side, the generator wants to minimise the value maximised by the discriminator, since G it's playing again D. Namely, the generator computes

$$\hat{\theta}_G = \min_{\theta_G}\left(\max_{\theta_D}\left(E_{s \sim \phi_S}\Big[\log\big(\mathcal{D}(s, \theta_D)\big)\Big] + E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]\right)\right)$$

As we can see, this is a min max optimisation problem which can't be solved directly. To solve the problem we have to do train the generator and the discriminator alternatively fixing the parameters of one network and computing using gradient ascend or descend for the other network. In particular, alternatively,

- We train the generator by freezing θ_D and computing

  $$\hat{\theta}_G = \min_{\theta_G}\left(E_{s \sim \phi_S}\Big[\log\big(\mathcal{D}(s, \theta_D)\big)\Big] + E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]\right)$$

  or, since the first term doesn't depend on θ_G,

  $$\hat{\theta}_G = \min_{\theta_G}\left(E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]\right)$$

- We train the discriminator by freezing θ_G and computing

  $$\hat{\theta}_D = \max_{\theta_D}\left(E_{s \sim \phi_S}\Big[\log\big(\mathcal{D}(s, \theta_D)\big)\Big] + E_{z \sim \phi_z}\Big[\log\big(1 - \mathcal{D}(\mathcal{G}(z, \theta_G), \theta_D)\big)\Big]\right)$$

Usually we do

1. k steps of stochastic gradient descend for the discriminator.
2. 1 step of stochastic gradient descend for the generator.

The learning algorithm is shown in Algorithm 3. Note that training can be executed with standard tools (backpropagation and dropout), however learning is rather unstable hence k has to be tuned properly. Moreover, the network requires a huge amount of data to properly learn how to generate realistic images. This is because the generator learns from errors and not directly from what it has to generate (i.e., we don't tell it what to generate but that a certain image doesn't look real).

Algorithm 3 The GAN network learning algorithm.

for $e = 1, ..., N_{epochs}$ **do**
  for $_{=}1, ..., k$ **do**
    $\mathbf{Z}_{batch} = \{\mathbf{z}_1, \dots, \mathbf{z}_m\}$
    $\mathbf{S}_{batch} = \{\mathbf{s}_1, \dots, \mathbf{s}_m\}$
    GRADIENT-DESCEND$(\nabla_{\theta_D} \big[\sum_e \log \mathcal{D}(\mathbf{s}_e, \theta_D) + \log \big(1 - \mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)\big])$
  end for
  $\mathbf{Z}_{batch} = \{\mathbf{z}_1, \dots, \mathbf{z}_m\}$
  GRADIENT-DESCEND$(\nabla_{\theta_G} \sum_e \big[\log \big(1 - \mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)\big])$
end for

12.3.3 Optimisations

GANs are generally hard to train. Luckily there exist tricks that can speed-up training time.

Gradient inversion

At the beginning, the value $\mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)$ is very small since the generator is not trained and produces images that are very different from real ones (hence the discriminator has not problem in telling apart real images from fake ones, and returns 0 when evaluating $\mathcal{G}(\mathbf{z}_e, \theta_G)$). Moreover, the gradient of the generator, where $\mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)$ is used, is

$$\sum \big[\log \big(1 - \mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)\big]$$

As shown in Figure 12.2, this function has a small gradient at the beginning (it's almost flat). This means that initially the model learns slowly and the speed increases as training goes on. To solve this issue we can use $-\log \big(\mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)$. In particular,

1. at the beginning we minimise $-\log \big(\mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)$ and,
2. as soon as training slows down, we switch back to maximising $\log \big(1 - \mathcal{D}(\mathcal{G}(\mathbf{z}_e, \theta_G), \theta_D)\big)$.

12.3.4 Mapping visualisation

Let us now show what happens when a GAN learns a mapping from the distribution ϕ_z to the distribution ϕ_S. Let us represent the points in the spaces $\mathbb{R}^{H \times W \times 3}$ of images and $\mathbb{R}^d$ as two lines. In this way we can represent a map from one point in $\mathbb{R}^d$ to one point in $\mathbb{R}^{H \times W \times 3}$ as a line. Let us also draw the distribution of images we are interested in, namely we want to represent where the manifold is in the space $\mathbb{R}^{H \times W \times 3}$. Figure 12.3 shows this setup. During training, as shown in Figure 12.4, the GAN learns to map the inputs $\mathbf{z}$ to the images in the manifold, hence it learns to map ϕ_z to ϕ_S. Hopefully, after training, the distribution learned by the network is the same as ϕ_S. This situation is shown in Figure 12.5.

12.3.5 Inference

The discriminator part of the GAN is discarded for inference and the generator is used to generate an image from some input noise, drawn from the same distribution ϕ_z used for training.

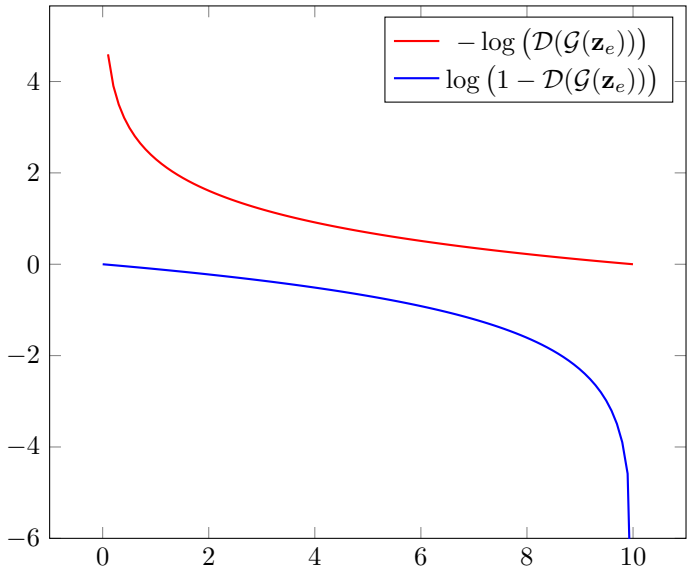


Figure 12.2: Functions optimised by the generator of a GAN.

12.3.6 Estimation of the generator's performance

Establishing the performance of a generator isn't easy since we should tell how realistic each image is. However, we can do an experiment to indicatively understand if the generator is able to map the input noise to the manifold. What we want to do is to take two points $\mathbf{z}_1$ and $\mathbf{z}_2$ in the distribution ϕ_z and map them to $\mathbb{R}^{H\times W\times 3}$ obtaining $\mathbf{s}_1 = \mathcal{G}(\mathbf{z}_1)$ and $\mathbf{s}_2 = \mathcal{G}(\mathbf{z}_2)$. Now we can

1. Draw a line between $\mathbf{z}_1$ and $\mathbf{z}_2$.
2. Take the points $\mathbf{z}_i$ on that line.
3. Map the points to $\mathbb{R}^{H\times W\times 3}$ to obtain $\mathbf{s}_i = \mathcal{G}(\mathbf{z}_i)$.
4. Draw a line between $\mathbf{s}_1$ and $\mathbf{s}_2$.
5. If the points $\mathbf{s}_i$ are realistic images, then $\mathbf{z}_i$ have been mapped to the line between $\mathbf{s}_1$ and $\mathbf{s}_2$ and it's highly probable that the GAN has learned to map ϕ_z to the manifold. Analysing the images on the line between $\mathbf{s}_1$ and $\mathbf{s}_2$ we can observe that they are the intermediate values between $\mathbf{s}_1$ and $\mathbf{s}_2$.

This process is shown in Figure 12.6.

12.3.7 Vector arithmetic

As for now, we have built a GAN able to build images close belonging (or close) to a manifold, however we haven't been able to generate specific images. For instance, we are able to generate images of faces but not the face of a woman or a face with some characteristics. Luckily, there exist ways to obtain such a result.

Say we want to obtain an a face of a woman with glasses. We can generate images $\mathcal{G}(\mathbf{z}_i)$ until we obtain:

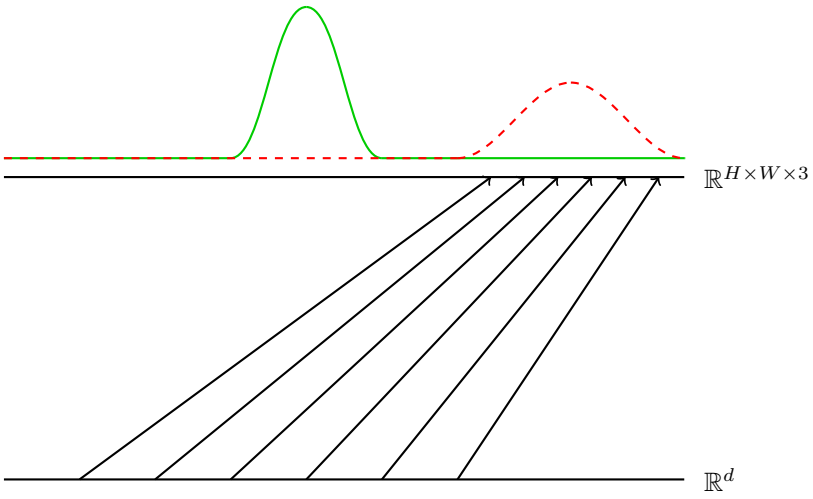


Figure 12.3: The distribution of ϕ_S (green) and $\mathcal{G}(\mathbf{z})$ (red, dashed) before training.

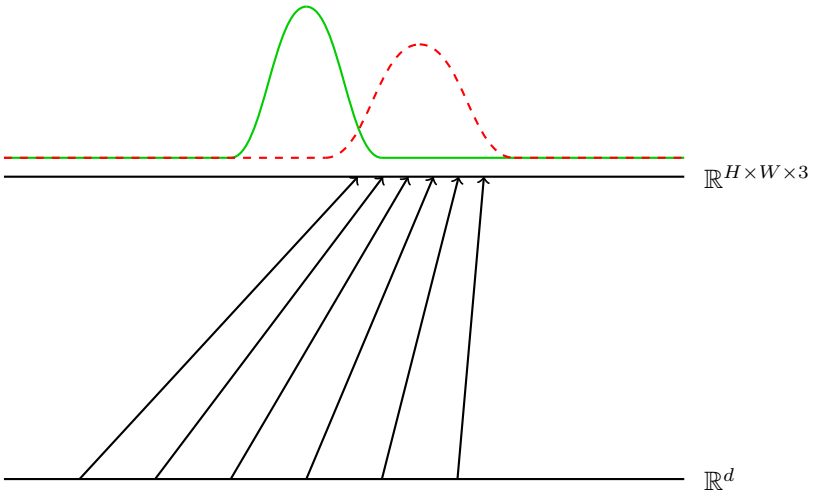


Figure 12.4: The distribution of ϕ_S (green) and $\mathcal{G}(\mathbf{z})$ (red, dashed) during training.

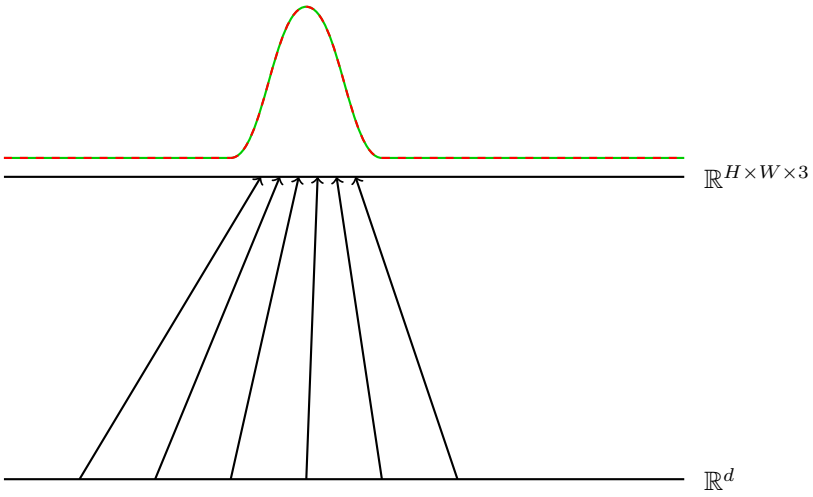


Figure 12.5: The distribution of ϕ_S (green) and $\mathcal{G}(\mathbf{z})$ (red, dashed) after training.

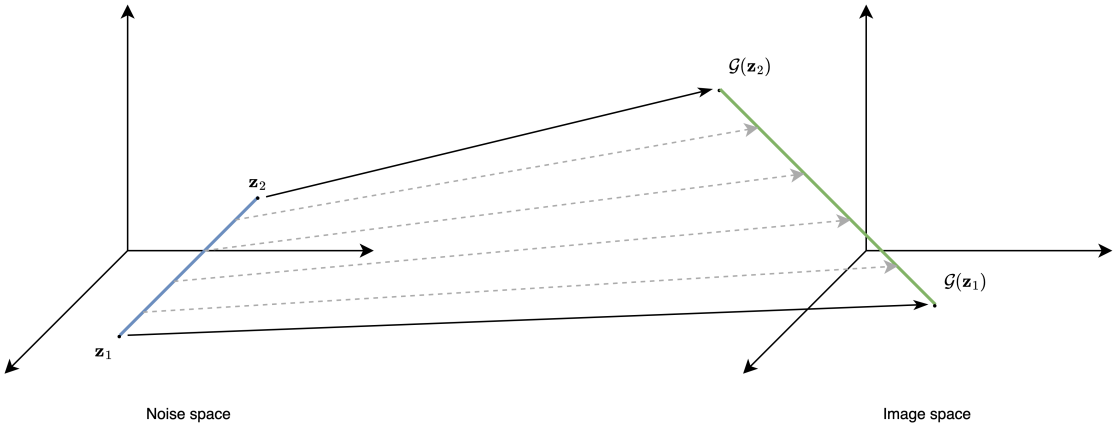


Figure 12.6: An example evaluation of a GAN's performance.

- Some images of a woman without glasses.
- Some images of a man without glasses.
- Some images of a man with glasses.

Now we can take the inputs $\mathbf{z}$ from which we have obtained the images above. In particular we call

- $\mathbf{z}_1 = \{\mathbf{z}_{1,1}, \mathbf{z}_{1,2}, \dots, \mathbf{z}_{1,a}\}$ the set of inputs that generate images of a man with glasses.
- $\mathbf{z}_2 = \{\mathbf{z}_{2,1}, \mathbf{z}_{2,2}, \dots, \mathbf{z}_{2,a}\}$ the set of inputs that generate images of a man without glasses.
- $\mathbf{z}_3 = \{\mathbf{z}_{3,1}, \mathbf{z}_{3,2}, \dots, \mathbf{z}_{3,a}\}$ the set of inputs that generate images of a woman without glasses.

Each vector $\mathbf{z}_{i,j}$ is a noise vector that maps to a specific image and the vectors $\mathbf{z}_i$ collect images that map to the same class (e.g., all noise vectors that maps to images of a woman with glasses). For each of these vectors $\mathbf{z}_i$, we can average the components to obtain

$$\bar{\mathbf{z}}_1 = \frac{1}{a}\sum_i \mathbf{z}_{1,i}$$

$$\bar{\mathbf{z}}_2 = \frac{1}{a}\sum_i \mathbf{z}_{2,i}$$

$$\bar{\mathbf{z}}_3 = \frac{1}{a}\sum_i \mathbf{z}_{3,i}$$

Now, if we want to obtain images of women with glasses, we can compute

$$\bar{\mathbf{z}} = \bar{\mathbf{z}}_1 - \bar{\mathbf{z}}_2 + \bar{\mathbf{z}}_3$$

In this way we have taken men with glasses, removed men without glasses (hence obtaining only glasses) and added women (hence obtaining women with glasses). If we now map $\bar{\mathbf{z}}$ to $\mathbb{R}^{H\times W\times 3}$ we obtain images of women with glasses. Basically, we can apply some vector arithmetic on the noise $\mathbf{z}$ to add and remove features of the output $\mathcal{G}$ and obtain, as a result, an image with such features.

12.3.8 GANs for anomaly detection

A generative adversarial network maps vectors in $\mathbb{R}^d$ from a certain distribution ϕ_z to a manifold $\mathbb{R}^{W\times H\times D}$ using a function $\mathcal{G}$. Assume now that we can invert $\mathcal{G}$ (which is actually not possible). This means that we can take take an image $I \in \mathbb{R}^{W\times H\times D}$ and map it to the distribution ϕ_z. If we obtain a high likelihood, then we can say that I is an image generated by $\mathcal{G}$, otherwise, if the likelihood is not big, we can say that the image isn't generated by $\mathcal{G}$, hence it might contain some anomaly. In other words, imagine that ϕ_z is a Gaussian distribution, hence we pick samples z centred in μ with a certain variance. If it's not likely that the vector $\mathcal{G}^{-1}(I)$ came from this distribution, then we can assume that I has some anomalies. In formulas, the lower the value

$$\phi_z\big(\mathcal{G}^{-1}(I)\big)$$

the highest the probability that I contains some anomalies.

The problem with this approach is that we can't invert the mapping function $\mathcal{G}$. The approach is however still valid, hence we should be able to find another way to map an image from $\mathbb{R}^{W\times H\times D}$ to $\mathbb{R}^d$. This can be done using an encoder network that takes an image in $\mathbb{R}^{W\times H\times D}$ and returns the corresponding vector in $\mathbb{R}^d$ that generated it. This type of network is called **bidirectional GAN**. The encoder $\mathcal{E}$:

- Brings an image back to the space of noise vectors.
- Can be used to reconstruct an input image as in autoencoders in fact if we take an input image s we can compute $\mathcal{G}(\mathcal{E}(s))$ (as an autoencoder) to obtain another image (hopefully as close as possible to s).

The architecture of a bidirectional GAN is shown in Figure 12.7.

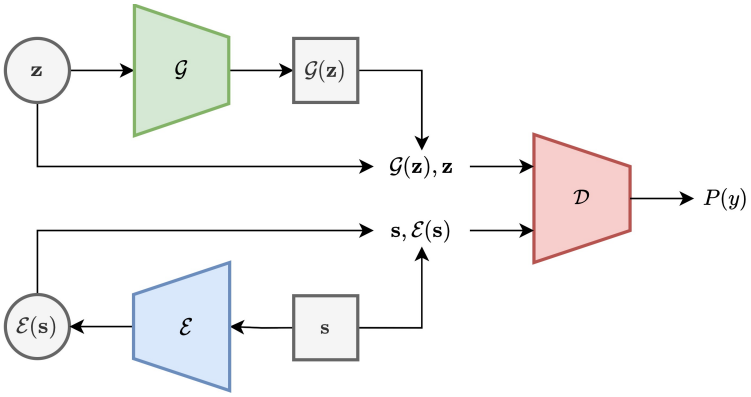


Figure 12.7: A bidirectional generative adversarial network.

Training

Both networks $\mathcal{G}$ and $\mathcal{E}$ can be trained using a discriminator that takes as input:

- An image $\mathbf{s}$.
- The encoded version of the image $\mathcal{E}(\mathbf{s})$.
- A noise vector $\mathbf{z}$.
- The image generated from the noise vector $\mathcal{G}(\mathbf{z})$.

and it:

- Discriminates between real images $\mathbf{s}$ and generated images $\mathcal{G}(\mathbf{z})$.
- Discriminates between couples $(\mathbf{s}, \mathcal{E}(\mathbf{s}))$ and $(\mathcal{G}(\mathbf{z}), \mathbf{z})$.

In particular, the discriminator maximises the function

$$V(\mathcal{D}, \mathcal{E}, \mathcal{G}) = E_{s\sim\phi_S}[\log \mathcal{D}(s, \mathcal{E}(s))] + E_{z\sim\phi_z}[1 - \log \mathcal{D}(\mathcal{G}(z), z)]$$

whilst encoder and generator minimise it. In general, the whole network computes

$$\min_{\mathcal{G},\mathcal{E}} \min_{\mathcal{D}} V(\mathcal{D}, \mathcal{E}, \mathcal{G}) = E_{s\sim\phi_S}[\log \mathcal{D}(s, \mathcal{E}(s))] + E_{z\sim\phi_z}[1 - \log \mathcal{D}(\mathcal{G}(z), z)]$$

Inference

At inference, the anomaly score combines:

- The image reconstruction error
$$\|\mathcal{G}(\mathcal{E}(\mathbf{s})) - \mathbf{s}\|_2$$
If we don't take the L2-norm, the difference $\mathcal{G}(\mathcal{E}(\mathbf{s})) - \mathbf{s}$ highlights the areas where the reconstructed image differs from the original one, hence the ares where the original image $\mathbf{s}$ presents some anomalies with respect to what the encoder expects.
- The discriminator loss
$$\big(\mathcal{D}(s, \mathcal{E}) - 1\big)^2$$

Another way of computing the anomaly score of an image is by using the following function

$$A(\mathbf{s}) = (1-\alpha)\|\mathcal{G}(\mathcal{E}(\mathbf{s})) - \mathbf{s}\|_2 + \alpha\|f(\mathcal{D}(\mathbf{s}, \mathcal{E}(\mathbf{s}))) - f(\mathcal{D}(\mathcal{G}(\mathcal{E}(\mathbf{s})), \mathcal{E}(\mathbf{s})))\|_2$$

where

- The first term is the **reconstruction loss**.
- The second term is the **distance among the latent representation** of $\mathcal{D}$.
- f is the function computed by a CNN that extracts the latent representation.

Limitations

The main limitations of bidirectional GANs are:

- They have little stability during training.
- There is no way of promoting better quality of reconstructed images.

12.4 Dall-E2

Generative Adversarial Networks are used to generate realistic images. Another interesting problem is generating images from their textual description. Dall-E2 is a program that solves this problem using a **diffusion model**, which is a generative model. The main difference with respect to what we've seen so far is that we have to extract features from text and not from images. This problem is solved by a model called **clip**. To sum thigs up, Dall-E2 architecture can be divided in:

- A clip model that extracts features from a text and feeds them to the diffusion model.
- The diffusion model that generates images from the features extracted by clip.

12.4.1 Clip

Clip is a huge model trained from image captions to generate consistent representations of both images and text. Clip is trained by contrastive loss from a training set of images (lower blue part in Figure 12.8) with their captions (upper green part in Figure 12.8). After training, only the text encoder (i.e., the upper part) is kept to obtain a latent representation from text. The output of clip, which is a latent representation of the text called **text embedding**, is then fed to a network that takes as input a random noise $z \in \phi_z$. The text latent representation is used to condition the noise and obtain a noise embedding conditioned on the input text. This representation is then fed to another network that generates the actual image. The architecture of the clip module is shown in Figure 12.8.

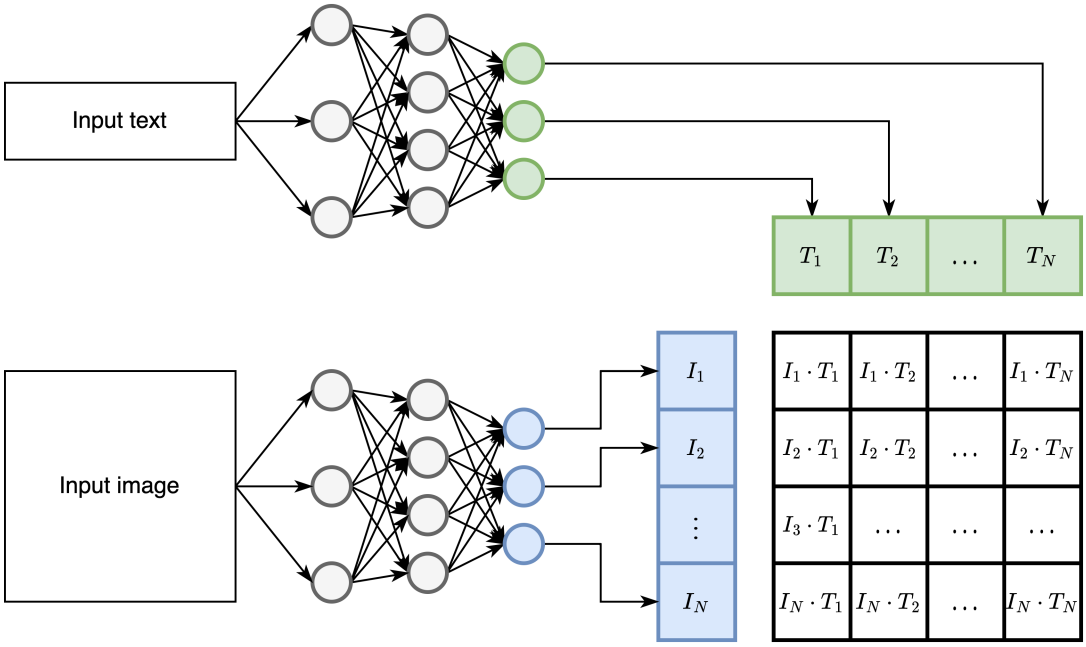


Figure 12.8: The clip module used by Dall-E 2 for extracting a text embedding from text.

Part IV

Recurrent Neural Networks

Chapter 13

Recurrent Neural Networks

13.1 Handling sequential data

Up until now we have considered neural networks that handle data statically. This means that the output of a neural network doesn't depend on the input's history, hence the network has no memory of the input. This is fine for some type of problems, however in some cases we might want to deal with sequential data, hence data in which we have a time dimension and the order in which data points are fed to the network is relevant. Note that sequential data isn't related to time only, but, to make things easy we will speak about time sequences. Everything we'll see is however valid for whatever type of sequence.

Sequential analysis is key, for instance, when analysing a phrase. Depending on the order in which words are issued, the phrase might have a completely different meaning, hence it's important to have a time dimension and a memory, when considering problems like this.

There exist different ways to handle sequences of data, some of which use classical neural networks or no neural network at all. Such models have, however, some issues hence we have to introduce models with memory. Some examples of models that can deal with sequences of data are:

- **Autoregressive models**.
- **Classical feed-forward neural networks**.
- **Recurrent neural networks (RNN)**, or more in general, models with memory.

Before diving deep into the third class of models, let us analyse, from a general perspective, the first two solutions, which don't involve memory and then the third. In the following overview and for RNNs, we will call

$$\mathbf{x}_t = \{x_1^{(t)}, x_2^{(t)}, \dots, x_i^{(t)}, \dots, x_I^{(t)}\}$$

one data point (i.e., one input vector) fed as input of the neural network at time t, which is a sequence of data (e.g., a phrase where each $x_i^{(t)}$) is a word.

13.1.1 Autoregressive models

Autoregressive models are memory-less linear models that don't require a neural network. The idea is to select a time window τ and use the outputs from previous instant $t-1$ to the τ-th previous input for predicting the output. In other words, the output y_t at time t is computed as a linear

combination of $y_{t-1}, \dots, y_{t-\tau}$. Note that, even if this model has some memory, it's limited to a fixed number of inputs.

13.1.2 Feed-forward neural networks

Classical feed forward neural networks, without memory, can be used for handing sequential data, too. The approach is similar to the one of autoregressive models. In particular, instead of feeding the neural network with the current input $\mathbf{x}_t$ we use τ previous inputs. In practice, if we want to consider τ previous inputs, we have to build an input layer with $\tau \cdot I$ neurons so that we can feed the network with $\mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-\tau}$. In other words, τ input vectors are concatenated and fed to a network with a bigger input layer. This process allows us to unroll time since we are putting inputs from different time instances in a single vector. An example of feed-forward neural network with multiple inputs is shown in Figure 13.1.

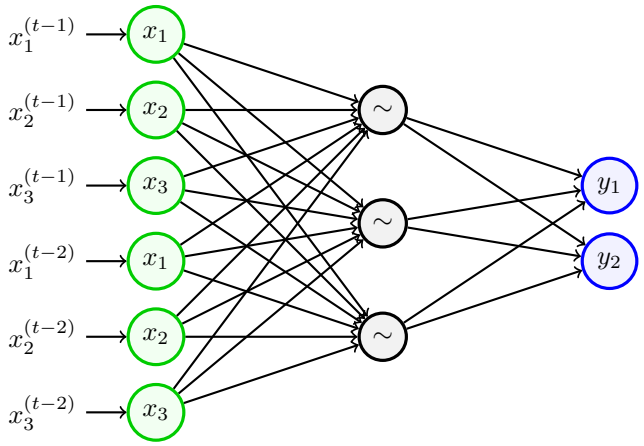


Figure 13.1: A feed-forward neural network fed with inputs of different time instances.

13.1.3 Models with memory

Models with memory use a different approach with respect to memory-less models. In particular, they take only one input per time instant. The network has an internal hidden representation of the state of the model, which is changed by the input at time t. More precisely, the hidden state at time t is a function of the input and the hidden state at time $t-1$. This means that we are adding an element of recursion to the network. The output of the system is a function of the hidden representation, hence it considers the history of inputs. Note that the internal state of the model can't be observed directly, but it's used by the system to compute the output for a given input.

13.2 Dynamical systems

The general structure of models with memory can be further detailed creating different models. Some of these are stochastic since they consider the noise possibly produced by the system. Some instances of stochastic models with memory are:

- **Kalman filtering** models. In Kalman filtering models:

  - The hidden state at time t is computed as function of the input at time t and the hidden state at time $t-1$.
  - The output at time t is computed as function of the hidden state.

  Moreover, state are continuous with Gaussian uncertainty and transformations are assumed to be linear.

- **Hidden Markov** models. In hidden Markov models:
  - The hidden state at time t is computed as stochastic function of the hidden state at time $t-1$. The state at time t can be computed using the Viterbi algorithm.
  - The output at time t is computed as function of the hidden state.

  Moreover, state are discrete and state transitions are stochastic.

13.3 Recurrent neural networks

Recurrent neural networks (RNNs), sometimes called Hellman networks, are deterministic models with memory that use recurrent connections to represent the memory of the past. Before explaining in detail what a recurrent connection is, let us state an important result.

Theorem 13.1 (Turing completeness of recurrent neural networks)**.** *With enough neurons and time, recurrent neural networks can compute anything that can be computed by a computer. That is to say, recurrent neural networks are Turing complete.*

13.3.1 Recurrent connections

Recurrent neural networks are based on recurrent connections. Let us consider a simple architecture, like the one shown in Figure 13.2, to analyse this type of connection.

The architecture shown in Figure 13.2 is like the one of a normal neural network, apart from some connections that go against the usual flow. In particular, in our example, we have two connections that go from a hidden neuron to an input neuron. These connections are called recursive connections and allow to store in c_i the output of a neuron so that it can be used in the next iteration. Note that the nodes connected with a recursive connection are like normal neurons, hence they are connected, as one would expect, to every other neuron of the next layer. The only difference is that they also exploit another connection, which is the recursive connection. In practice, when the network is fed with $\mathbf{x}_t$:

1. The previous state is in the neurons $c_i^{(t-1)}$. We can call $\mathbf{c}_{t-1}$ the vector of these neurons.
2. The previous state $\mathbf{c}_{t-1}$ is propagated to all the neurons of the next state.
3. The previous state is updated with the output $\mathbf{c}_t$ of the hidden neurons having a regressive connection. In particular, $c_i^{(t-1)}$ is updated with the output $c_i^{(t)}(\mathbf{x}_t, \mathbf{W}, \mathbf{c}_{t-1}, \mathbf{V})$ of the hidden neurons with recursive connections.

Note that a RNN can have only recursive neurons or a mix of recursive and normal neurons. The big advantage of this architecture with respect to memory-less ones is that we don't have a fixed

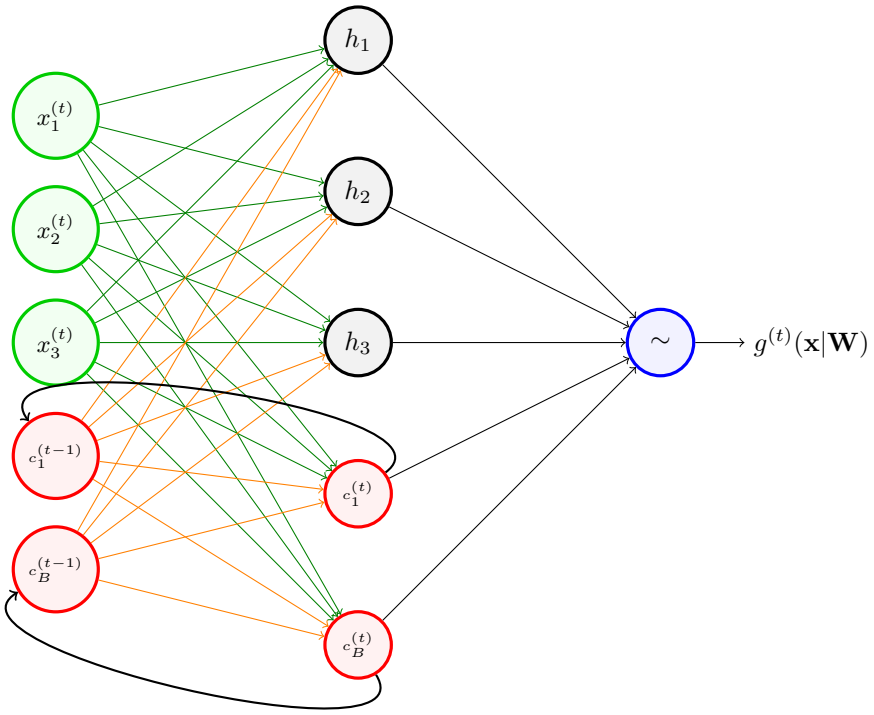


Figure 13.2: A recurrent neural network.

past time window (i.e., a fixed number of inputs that can be fed to the network) but we can store an history of arbitrary length (ideally of infinite length) represented with the recursive connections.

Now that we have a general idea of the network's architecture, let us explain more in detail how the outputs of the different nodes are computed. In this context, let us consider an architecture like the one in Figure 13.2 with

- I normal nodes in the input layer.
- B recursive nodes in the input layer.
- J normal nodes in the hidden layer.
- B recursive nodes in the hidden layer.
- 1 node in the output layer.

Hidden layer

First, let us consider the output of a simple neuron in the hidden layer. The output $h_j^{(t)}$ of neuron j at time t is computed as

$$h_j^{(t)}\left(\mathbf{x}_t, \mathbf{W}^{(1)}, \mathbf{c}_{t-1}, \mathbf{V}^{(1)}\right) = h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_i^{(t)} + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{(t-1)}\right)$$

where

- $\mathbf{W}^{(1)}$ is the matrix of weights from the input to layer 1 considering only neurons without recursive connections.
- $\mathbf{V}^{(1)}$ is the matrix of weights from the input to layer 1 considering only neurons with recursive connections.
- h_j is the activation function of neuron j.
- $\mathbf{x}_t$ is the input at time t.
- $\mathbf{c}_{t-1}$ is the previous state of the system.

Recursive hidden layer

Let us now consider neurons in the hidden layer that are connected, with a recursive connection, to a neuron of the previous layer. The output $c_b^{(t)}$ of recursive neuron b at time t is computed as

$$c_b^{(t)}\big(\mathbf{x}_t, \mathbf{W}_B^{(1)}, \mathbf{c}_{t-1}, \mathbf{V}_B^{(1)}\big) = c_b\bigg(\sum_{i=0}^{I} w_{bi}^{(1)} \cdot x_i^{(t)} + \sum_{d=0}^{B} v_{bd}^{(1)} \cdot c_d^{(t-1)}\bigg)$$

where

- $\mathbf{W}_B^{(1)}$ is the matrix of weights from the input to layer 1 considering only input neurons without recursive connections.
- $\mathbf{V}_B^{(1)}$ is the matrix of weights from the input to layer 1 considering only input neurons with recursive connections.
- c_b is the activation function of neuron b.
- $\mathbf{x}_t$ is the input at time t.
- $\mathbf{c}_{t-1}$ is the previous state of the system.

Output layer

Finally, let us consider the output neuron whose value can be computed as

$$g^{(t)}(\mathbf{x}|\mathbf{W}) = g\bigg(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^{(t)}\big(\mathbf{x}_t, \mathbf{W}^{(1)}, \mathbf{c}_{t-1}, \mathbf{V}^{(1)}\big) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^{(t)}\big(\mathbf{x}_t, \mathbf{W}_B^{(1)}, \mathbf{c}_{t-1}, \mathbf{V}_B^{(1)}\big)\bigg)$$

13.3.2 Learning with backpropagation through time

Having hopefully understood the structure of a RNN, we have to deal with training. It's in fact true that a RNN is a neural network and we can use backpropagation with gradient descend, however we have to take care of some important details. The main difference with respect to normal neural networks is that now we have a recursive connection that we have to develop during training. Being a recursion, we can always unroll, hence we can create an equivalent unrolled network in which the input and hidden layer (or in general, the layers containing recursive neurons) are repeated until we

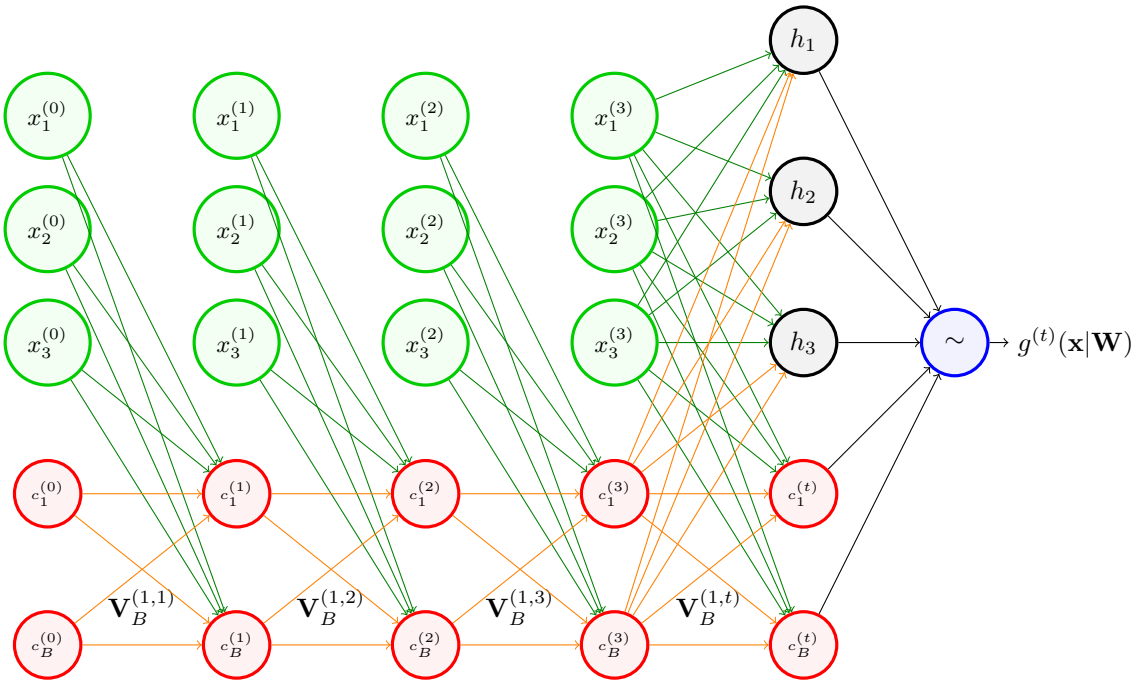


Figure 13.3: An unrolled recurrent neural network.

reach t_0. Note that only the last layer is connected to the output layer whilst the others are used only to compute the values of the recursive neurons. The unrolled network obtained from the one in Figure 13.2.

Considering a certain time instant t the weights $\mathbf{W}$ and $\mathbf{V}$ should be the same for each unrolled layer and should be the same of the actual layer. In other words, if we consider the layer with weights $\mathbf{W}_B^{(1)}$ and $\mathbf{V}_B^{(1)}$, then we should have

$$\mathbf{W}_B^{(1,\theta)} = \mathbf{W}_B^{(1)} \quad \forall \theta < t$$

and

$$\mathbf{V}_B^{(1,\theta)} = \mathbf{V}_B^{(1)} \quad \forall \theta < t$$

where $\mathbf{W}_B^{(1,\theta)}$ and $\mathbf{V}_B^{(1,\theta)}$ are the matrices of weights of the θ-th unroll. To obtain this result, we can initialise each replica $\mathbf{W}_B^{(1,\theta)}$ and $\mathbf{V}_B^{(1,\theta)}$ with the same value and then use the average of gradients at different time steps to update the gradient of each unrolled layer. In this way, all averages will be the same and each replica will contain the same value. In formulas, we write

$$\mathbf{W}_B = \mathbf{W}_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E}{\partial W_B^{(1,t-u)}}$$

and

$$\mathbf{V}_B = \mathbf{V}_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E^{(t)}}{\partial V_B^{(1,t-u)}}$$

where U is the number of steps (time instants) unrolled and E is the error. As explained before, the weights are updated summing up the gradient with respect to different weight replicas $\mathbf{W}_B^{(1,\theta)}$ and $\mathbf{V}_B^{(1,\theta)}$ and the divided by the number of replicas U. This process can end

- With a state initialised with value 0.
- Backpropagating also the initial state.

13.3.3 The vanishing gradient problem

Backpropagation in time works in theory, however if we consider a long unrolling sequence (i.e., a big value of U), we might encounter a problem, called **vanishing gradient problem**. To understand what's the issue with a big number of unrolls, let us consider the simplest network possible made, as shown in Figure 13.4, of three layers with one neuron per layer.

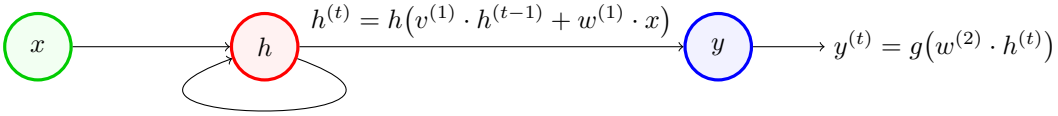


Figure 13.4: The simplest recursive neural network.

The gradient used for backpropagation over this network is computed as

$$\frac{\partial E}{\partial w} = \sum_{t=1}^{S} \frac{\partial E^{(t)}}{\partial w}$$

Let us now expand the derivative of the error with respect to the weights w.

$$\frac{\partial E^{(t)}}{\partial w} = \sum_{t=1}^{t} \frac{\partial E^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial w}$$

Finally, we can expand the term $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ related to recursion as

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} v^{(1)} h'\big(v^{(1)} \cdot h^{(i-1)} + w^{(1)} \cdot x\big)$$

Basically, we have written the derivative of each layer with respect to the previous one. If we consider the norm (i.e., the absolute value) of $\frac{\partial h_i}{\partial h_{i-1}}$, we can write

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \leq \|v^{(1)}\| \|h'(\cdot)\|$$

If we want to compute the norm of $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ we can simply multiply both sides by $t-(k+1)+1 = t-k$ and obtain

$$\left\| \frac{\partial h^{(t)}}{\partial h^{(k)}} \right\| \leq (\gamma_v \cdot \gamma_{h'})^{t-k}$$

where $\gamma_v = \|v^{(1)}\|$ and $\gamma_{h'} = \|h'(\cdot)\|$. If $(\gamma_v \cdot \gamma_{h'})$ is strictly smaller than 1, i.e., $\gamma_v \cdot \gamma_{h'} < 1$, then $(\gamma_v \cdot \gamma_{h'})^{t-k}$ converges to 0, then $\left\| \frac{\partial h^{(t)}}{\partial h^{(k)}} \right\|$ converges to 0 and the gradient $\frac{\partial E}{\partial w}$ is 0. But if the gradient is 0 we don't update the weights hence the network doesn't learn. That's why this problem is called of the vanishing gradient. Moreover, if $\gamma_v \cdot \gamma_{h'}$ is greater than 1, the gradient explodes (because of the exponentiation). This means that the gradient must be strictly equal to 1. Let's understand which functions can generate these problems. Two of the most used activation functions are the sigmoid and the hyperbolic tangent:

- The sigmoid activation function, defined as

  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  has derivative

  $$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$$

  As we can see, the maximum of this function is 0.25, hence we can't satisfy the condition

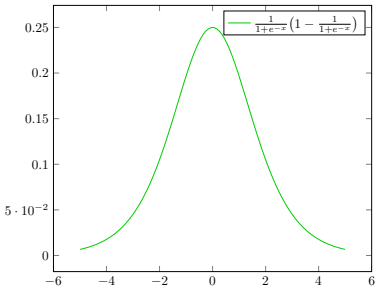
  

  Figure 13.5: The derivative of the sigmoid function.

  above. This means that the vanishing gradient problem presents itself if we use the sigmoid activation function with too many layers.

- The hyperbolic tangent activation function, defined as

  $$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  has derivative

  $$\tanh'(x) = 1 - \big(\tanh(x)\big)^2$$

  As we can see, the maximum of this function is 1, hence the gradient goes to 0 even in this case.

This means that neither the sigmoid nor the hyperbolic tangent hyperbolic functions work fine with many layers. Having an activation function with derivative 1 isn't however enough in fact, if $\gamma_v < 1$, the value $(\gamma_v \cdot \gamma_{h'})^{t-k}$ still converges to 0 and the vanishing gradient problem is still there. Moreover, if $\gamma_v > 1$, $(\gamma_v \cdot \gamma_{h'})^{t-k}$ increases exponentially, which is not good either. As a result, if we want to solve both problems, we have to use:

- activation functions whose derivative has only value 1 and,

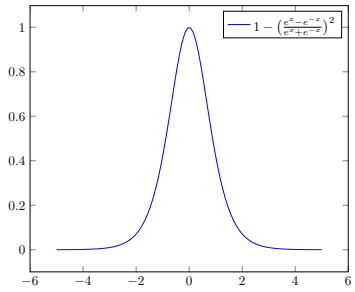


Figure 13.6: The derivative of the hyperbolic tangent function.

- weights $v^{(1)}$ whose value must be 1.

Only if these conditions are true together, the gradient doesn't vanish when the number of layers is high. An example of activation function that has derivative always equal to 1 is the ReLU activation function (the same used in CNNs) defined as

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

The derivative of this function is always 1 for values of x greater than 0. If we combine this with constant values for the weights $v^{(1)} = 1$, we get a model that doesn't suffer with the vanishing gradient problem. Note that, saying that the weights must be equal to 1 means that the recursive neurons only have to accumulate the input, hence this solution isn't that useful.

13.4 Long short-term memories

Long short-term memories solve the problems of basic RNNs. In particular, LSTMs allow to do more than simply accumulate values in recursive neurons. The basic block used in LSTMs is a constant error carousel (CEC). Such block represents a basic memory unit and inherits from the simple neural network (Figure 13.4) analysed when talking about the vanishing gradient. Since we are talking about memory blocks, we have to add some neurons to read from and write to memory. More precisely, the general structure of a CEC is the following:

- A recursive neuron with recurrent weight $v = 1$ accumulates the input.
- An input neuron used to write some value in memory.
- A write gate, which is a neuron fed with multiple values and having a non-linear activation function, is used to write data to memory (i.e., in the recursive neuron). Whenever the gate is on, the input is written in memory.
- An output neuron with a non-linear activation function outputs the value stored in memory.
- A read gate, which is a neuron fed with multiple values and having a non-linear activation function, is used to read data from the memory. Whenever the read gate is on, the value in memory is sent out in output.

13.4.1 LSTM chain

Now that we know how a block is made, let us consider a LSTM through time. Namely, let us unroll a LSTM through time such that the output of the CEC at time $t-1$ is fed as input of the CEC at time t. When considering this structure, we want to avoid that, during backpropagation, we pass through a sigmoid or hyperbolic tangent activation since we don't want to have the vanishing gradient problem. To achieve this result, let us consider the structure in Figure 13.7. Note that what we are going to say is analysed also at https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

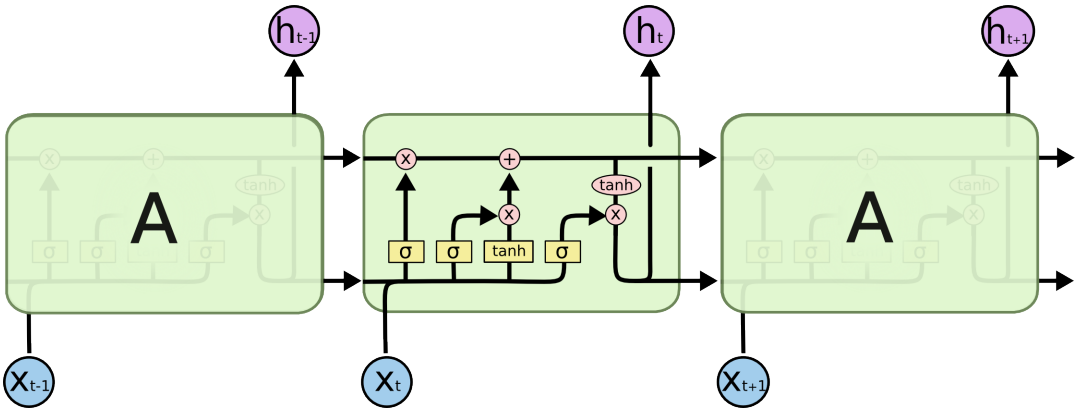


Figure 13.7: A LSTM chain.

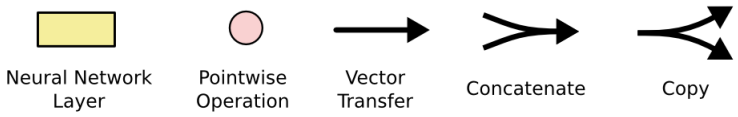


Figure 13.8: The notation used in Figure 13.7.

Let us now analyse a block of the chain, i.e., a CEC at one time instant, one piece at a time.

Input gate

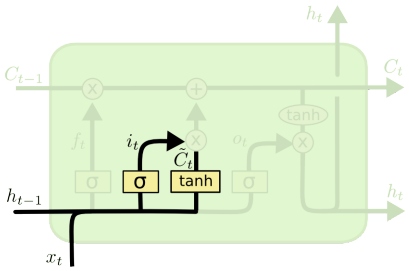
The first part we will analyse is the input gate, which is highlighted in Figure 13.9. This gate:

- Receives the input $\mathbf{x}_t$.
- Receives the past output $\mathbf{h}_{t-1}$.
- Computes the hyperbolic tangent of the inputs, whose result is (or can be) written in memory. In formulas
$$C_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_C)$$
where $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ is the vector obtained concatenating $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$.
- Has a gate that decides if the value C_t should be written to memory. The output of this gate is computed using a sigmoid function and:

- If the output of the sigmoid is 1, the value C_t is written in memory.
- If the output of the sigmoid is 0, the value C_t isn't written in memory.

In formulas, the output of this layer is

$$i_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_i)$$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Figure 13.9: A LSTM block with focus on the input gate.

Forget gate

Since the CEC is a memory block, we might want to reset the value stored in memory. The forget gate is used just for this purpose. This gate, highlighted in Figure 13.10, takes as input

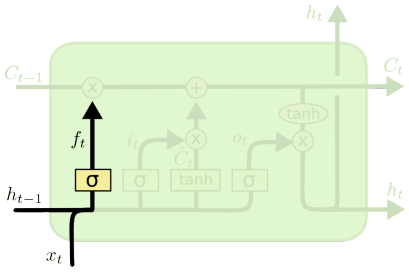
- The input $\mathbf{x}_t$.
- The past output $\mathbf{h}_{t-1}$.

and computes the function

$$f_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_f)$$

which is multiplied by the value of the memory. This means that,

- If $f_t = 0$, the value of the memory is reset.
- If $f_t = 1$, the value of the memory is kept.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Figure 13.10: A LSTM block with focus on the forget gate.

Memory gate

The memory gate, highlighted in Figure 13.11, is used to update the value stored in memory. In particular, this gate receives the values $\tilde{C}_t$ and i_t computed by the input gate and the value f_t computed by the forget gate and computes the new value in memory as

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Namely, the memory gate:

1. Resets the value in memory if $f_t = 0$.
2. Adds to the current value in memory (0 if it has just been reset) the value $\tilde{C}_t$ if $i_t = 1$.

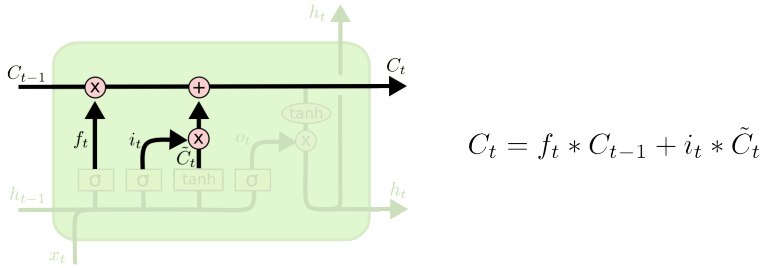


Figure 13.11: A LSTM block with focus on the memory gate.

Output gate

The output gate, as highlighted in Figure 13.12, takes:

- The value C_t in memory.
- The input $\mathbf{x}_t$.
- The past output $\mathbf{h}_{t-1}$.

and computes

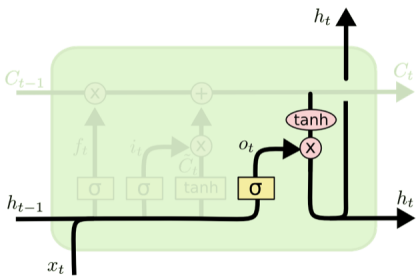
- The output $\tilde{\mathbf{h}}_t$ as
$$\mathbf{h}_t = \tanh(C_t)$$
- The control signal o_t as
$$o_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_o)$$

The output $\tilde{\mathbf{h}}_t$ is used only if the control signal is set to 1, hence the output of a LSTM block is

$$\mathbf{h}_t = o_t \cdot \tanh(C_t)$$

13.4.2 Learning

Given a LSTM block, the network has to learn the weights of the block (i.e., W_i, W_C, W_f, W_o). In particular, learning is done using backpropagation but, differently from RNNs, the derivative of the memory is always 1 since it's the value of the forget gate.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

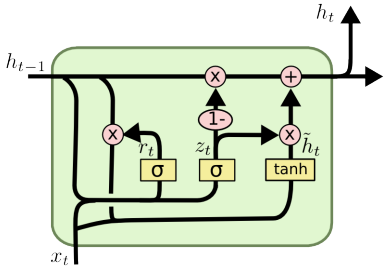
Figure 13.12: A LSTM block with focus on the output gate.

13.4.3 General block structure

Note that the structure of a LSTM block we have analysed can be generalised to an arbitrary number of neuron for each gate. For instance, memory can be made of multiple neurons, in which case we have multiple lanes and the function f_t is computed for each of this lanes. The same holds for every other memory line.

13.4.4 Gated Recurrent Unit

A Gated Recurrent Unit (GRU) is a simplified version of the LSTM block. The main difference is that a GRU doens't have a memory unit. An example is shown in Figure 13.13. Note that this architecture isn't much more efficient than a LSTM.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 13.13: A Gated Recurrent Unit.

13.4.5 Multiple layers

LSTM layers can be stacked one on top of the other (the output of one block is used as input of the next). In this case, the state of each block is used for the same block at the next time instant. The final output of this network, for each time instant, can be computed using a fully connected classifier stacked at the end of the LSTM blocks. An example is shown in Figure 13.14.

13.4.6 Bidirectional LSTM networks

As for now, we have considered sequences in which time flows in the direction we are used to. For instance if we consider a phrase as a sequence of words, we have analysed words form the first to the last (i.e., from the least recent to the most recent). It's however also possible to go the other way

around and analyse sequences in the opposite order. And this is not all, in fact one can also decide to use both approaches simultaneously, in particular we can:

- Analyse sequences from the beginning to the end and get a representation.
- Analyse sequences from the end to the beginning and get another representation.
- Concatenate the representations.

This can be done in a single network using two stacks of LSTM blocks concatenated at the end. The concatenated output is then passed through a fully connected classifier that predicts the output.

13.5 One dimensional convolution

Sequences can also be analysed using convolution. In particular, we want to use the same operation as convolution but applied to a vector (i.e., a sequence) instead of a matrix (i.e., an image). This means that the filters are one-dimensional vectors that slide through the input sequence and compute the weighted sum of the inputs. In formulas, the cell y_i in position i of the output $\mathbf{y}$ is

$$y_i = \sum_{\delta \in \{1,\dots,K\}} w_\delta x_{i+\delta}$$

where K is the dimension of the filter. Convolutions is a simple operation, hence one might thing networks with such layers are too easy to learn the patterns in the data set. This is rather false, in fact we can obtain really good results even with such simple networks.

13.6 Mixing sequential and static data

Sequential and static data can be combined. In particular, we can build models that receive as input static data and generate a sequence of data. More precisely, we can have four different combinations:

- **One-to-one** models. One-to-one models are those we've seen until now (e.g., image classification).
- **One-to-many** models. One-to-many models take one static input and generate a sequence of outputs. One example is a model that takes an image and returns a description of that image. Note that some text is a sequence of words, hence sequential data, since one word influences the ones around it.
- **Many-to-one** models. Many-to-one models take a sequence of inputs and generate a single output. An example is a model that takes a phrase and tells if that phrase expresses a positive or negative feeling (e.g., today I'm happy or today I'm sad).
- **Many-to-many** , or sequence-to-sequence, models. Many-to-many models take a sequence as input and generate another sequence as output. An example is text translation.

Let us now focus on two of these classes of models.

13.6.1 One to many models

One-to-many models take one static input and generate a sequence of outputs. Let us consider the example of image to text to understand some important things about this class of models. In this example, given a dictionary of words we can use, the model returns, for each element of the sequence, a vector of probabilities as long as the dictionary. The vector contains, for each word, the probability that such word is the right one to insert in the output text at that point of the sequence. Also note that each output of the sequence depends only on the input and the previous state and the state depends on the input. This means that at each step of the sequence we get a different word which is influenced by the previous words (i.e., the state of the model) and the input.

13.6.2 Sequence to sequence models

Many-to-many models take a sequence as input and generate another sequence as output. This class of models can take two different approaches. Let us consider the example of text translation to understand them. When we translate a text from one language to another we can:

- Translate every word one at a time without looking into the next one (or with a short view over the past and following words). In this case the translation is a bit clumsy since we can't build a translation using the phrase structure of the target language.
- Listen to the whole phrase to translate, represent and summarise the idea and the message the phrase wanted to express and then translate it. In this case we can elaborate the source sentence and translate it using the phrase structure of the target language since we want to express the concept we have in mind in another language.

This distinction can be done also in sequence-to-sequence models. In particular we can:

- Build a model that, for each input in the sequence, generates the respective output in output sequence. This is the case of word-to-word translation.
- Build a model that accumulates the input and builds a latent representation of the phrase and then generates the output sequence using only that latent representation and the previous output of the system. This is the case of concept-translation (i.e., when we build the concept in our mind and express it in the target language). In this case, the network behaves like an autoencoder since we are building a latent representation from the input and then building back the original text (but in another language). Finally, note that the model generates some output in the first phase, too, only we are discarding it.

The former model works but is more straightforward and less interesting to study. Let us then focus on the latter. The main things we have to address in such model are:

- How to encode every word in the dictionary.
- How to handle the fact that the output text might have a different length than the input.

For the first problem, let us use a map that associates an identifier to each word of the dictionary. This solution isn't the best one, but we will come back on how to represent text later on. For now, let us consider this solution. Let us now move to the second issue. To solve it, we have to find a way to tell the model that the input phrase is over so that it can add this information to its internal state. This information can then be used, thanks to training, to understand when the output phrase should stop. This means that we have to add to the dictionary some extra special words, which are used to say that a phrase is over. More precisely, we should add:

- <PAD>, that is used during training to generate text of the same size. This is because we use batches, which we want of the same size.
- <EOS>, that tells the decoder, during training where a sentence ends, and it allows the decoder to indicate the same thing in its outputs as well. In other words <EOS> is used to indicate where the output sequence should stop.
- <UNK>, it's used to replace words of the dictionary that don't show up very often in the dataset's texts. On real data, it can vastly improve the resource efficiency.
- <SOS/GO>, that tells the decoder where the input text ends. This is also the input to the first time step of the decoder to let the decoder know when to start generating output.

Now that we know the basic components that we need to build a sequence-to-sequence model, let us understand its basic overall structure and how it's used during training and inference.

Architecture

The simplest architecture we can think of is one recursive unit (we can think of LSTM or RNN) which

1. Takes one element of the input sequence.
2. Modifies it's internal state.
3. Generates an output using the input and the previous state.

Training

Before talking about training, let us explain what we can find in the dataset. A dataset is a set of texts, each of which can be written as:

$$T_x.< \texttt{GO} >.T_y.< \texttt{EOS} >$$

where

- T_x is the source text that has to be translated.
- <GO> is the special word that tells that the source text is over.
- T_y is the target text, i.e., the correct translation of the source text. This is basically the label of the input text.
- <EOS> is the special word that tells that the target text is over.
- . is the concatenation.

Training can be divided into two phases:

1. In the first phase, we want the network to build the internal representation of the source text. This means that the network is fed with the words in the input text and the outputs of the network are ignored.

2. In the second phase, which starts when the network receives as input a `<GO>`, the network is fed with the words of the target text T_y and the output is one word of the transition generated by the model. Each output can be compared with the next input of the sequence (if they are equal, the model has predicted that word correctly). The model stops when it generates the word `<EOS>`.

Given pairs $< S, T >$ where S is the source string and T is the target, and the output T' of the model we want to maximise the probability

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

Inference

During inference, the procedure is roughly the same, in fact we still have to phases:

1. In the first phase, the network builds the internal representation of the source text. This means that the network is fed with the words in the input text and the outputs of the network are ignored.

2. In the second phase, which starts when the network receives as input a `<GO>`, the network is fed with previous output of the sequence and generates the next output of the sequence until the word `<EOS>` is generated.

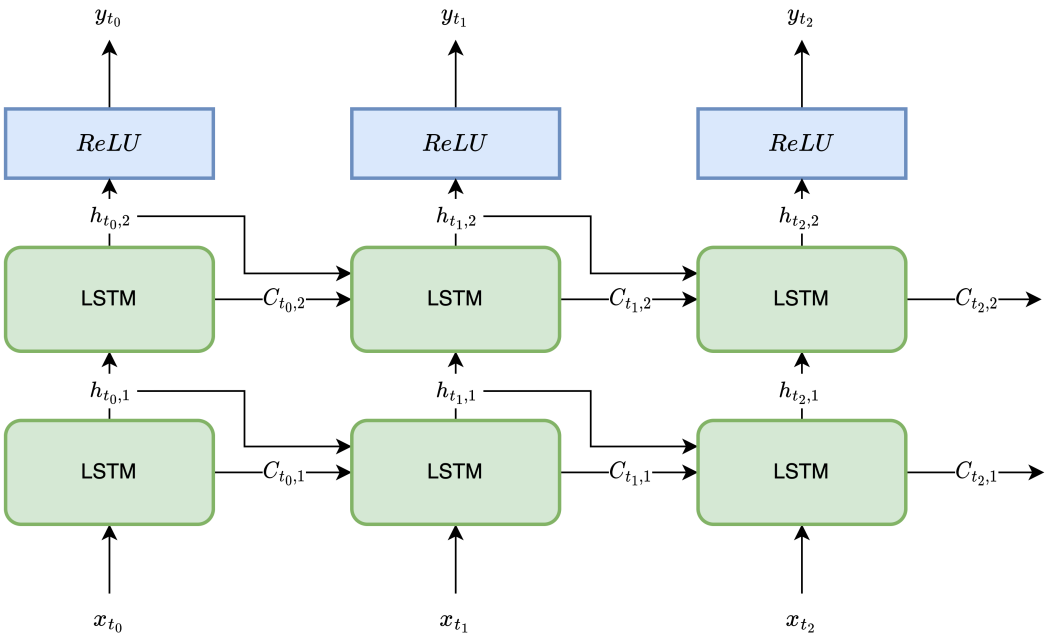
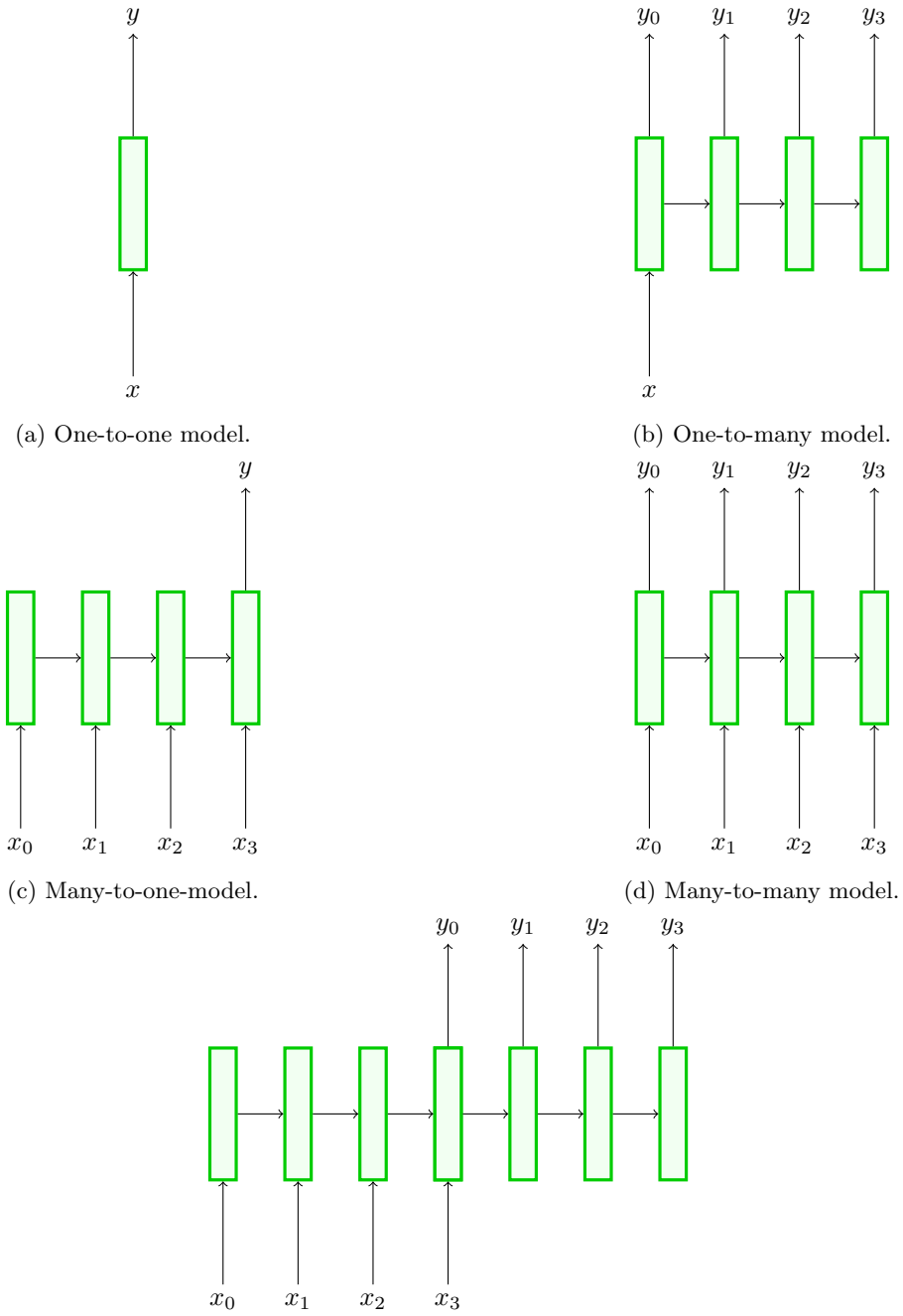


Figure 13.14: A network made of multiple LSTM layers.



(a) One-to-one model.

(b) One-to-many model.

(c) Many-to-one-model.

(d) Many-to-many model.

(e) Many-to-many model.

Figure 13.15: Categories of models that use sequences as input, output or both.

Chapter 14

Word embedding

14.1 Encoding phrases with numbers

Recurrent neural networks usually deal with phrases. It makes sense to understand how to encode words so that we can represents phrases efficiently. Note that we don't want to solve a problem with sequences. In fact we just want to find the correct representation of a set of words (independently from the fact that such words are then used in a sequence).

14.1.1 Bag of words

The first way we have to encode a text into a vector is to use the bag of words model. This model represents each phrase as a vector of the same length of the dictionary (i.e., of the list of all possible words we can find in a phrase). Each cell of the vector is associated with a word in the dictionary and the cell contains the number of times that word appears in the phrase. Equivalently, we can store the frequency with which the word appears. For instance if our dictionary is

```
{'the', 'a', 'cat', 'is, 'are', 'on', 'cat', 'table', kitchen'}
```

then the phrase `The cat is on the table`, is represented as

```
{'the':2, 'a':0, 'cat':1, 'is:1, 'are':0, 'on':1, 'cat':1, 'table':1, kitchen':0}
```

or more compactly as

```
[2, 0, 1, 1, 0, 1, 1, 1, 0]
```

In this case the dictionary is very small, but in real world scenarios, it contains more words. This means that a phrase is represented as a sparse vector since it contains way less words than the one in a dictionary. This means that the representation of a phrase is very sparse and high dimensional. Note that this model is suitable also for representing a single word. In this case the word is represeted in one-hot notation.

14.1.2 N-gram model

Another model used for encoding phrases is the N-gram model. The N-gram model is an extension of the bag of words model in which we don't consider single words but sequences of N words and

then we compute the probability of each of these of being in a phrase. This model computes the probability of a certain phrase s_k of length k as

$$P(s_k) = \prod_{i=1}^{k} P(w_i|w_1, \ldots, w_{i-1})$$

Basically, we are saying that the probability of a certain sentence is the product of the probability of choosing a word w_k given that we have chosen $w_{k-1}, \ldots, w_1$. If we assume that a word w_k is influenced only by a limited number of words in the past, then the model becomes

$$\hat{P}(s_k) = \prod_{i=1}^{k} P(w_i|w_{i-n+1}, \ldots, w_{i-1})$$

where n is the number of previous words that influence a word. This model works but it's practically unfeasible to use. Say for instance we have a 10-gram and a dictionary of 100000 words. This means that each sentence is a point in a 10-dimensional space with 100000 slots for each dimension. Since we have to compute the probability for each word, we have to compute 100000^{10} probabilities, which is a huge number. Moreover, the probability will be very low along all points and if we want to centre the probability on one region of space we have to add new phrases. Adding new sentences means however adding new words to the dictionary (since new sentences might contain words never seen before), hence increasing the sentence space.

14.2 Representing words

Another problem we have to face is how to represent single words. In general it could be enough to represent words with numbers. However, with a general encoding, we can't capture some characteristics that are important when working with text. Say for instance we use the one-hot encoding to represent words, which has been for a long time the preferred choice. This means that each word is a vector as long as the dictionary and it contains all 0s and a 1 in the position of the word. That is to say, the i-th word of the dictionary is represented as a vector with all 0s and a 1 in position i. With this representation, every word is orthogonal to the others and we can't say if two words have a similar meaning, hence they can be used exchangeably. In short, we can't assess the similarity between words. Word embedding tries to solve this problem by mapping a high-dimensional representation of a word, i.e., its one-hot encoding, to a lower dimensional input space such that similar words are represented as vectors that lie one close to the others. More formally:

> **Definition 14.1** (Word embedding)**.** *Word embedding is any technique mapping a word (or phrase) from it's original high-dimensional input space (the body of all words) to a lower-dimensional numerical vector space.*

This means that we want to map a word w in the vector space of the dictionary (formally $w \in \mathbb{Z}_2^{\|V\|}$) to a lower vector space with continuous values. This mapping should address:

- **Compression**. The destination space should be smaller than the starting one.
- **Densification**. The destination vectors should be more dense than the source ones.
- **Smoothing**. The vectors' values should be continuous instead of discrete.

- **Similarity**. Close words should be represented as vectors close one to the other. Related to similarity, we could require the model to represent the shift in meaning. This means that if we take two points W_1 and W_2 in the embedded word-space, the points on the line between W_1 and W_2 are the words which meaning is between the words W_1 and W_2.

14.2.1 NeuralNet language model

We need a neural network to overcome the problems of the N-gram model. The first attempt at using neural networks for modelling text and building a language model uses an architecture made of

1. An **input layer** that takes the previous $n-1$ words $w_{t-n+1}, \dots, w_{t-1}$ as one-hot vectors and maps them to another representation using a $|V| \times m$ table where $|V|$ is the number of words in the dictionary and m is the number of neurons we want to use for each word. If we call $C_{|V|,m}$ the function that maps the one-hot words in their representation, the output of this layer is the sequence
$$C_{|V|,m}(w_{t-n+1}), \dots, C_{|V|,m}(t-1)$$

2. A **linear projection layer** with $(n-1) \cdot m$ neurons. The input of the projection layer is
$$C_{|V|,m}(w_{t-n+1}), \dots, C_{|V|,m}(t-1)$$
obtained mapping the words in one-hot encoding with map C. The input vectors are then concatenated as to obtain a $1 \times |V|m$ vector and each cell of the vector is mapped to a neuron. The output of this layer is
$$\mathbf{z} = \mathbf{d} + \mathbf{H} \cdot \mathbf{x}$$
where $\mathbf{x}$ is the concatenation of the elements vectors $C_{|V|,m}(w_{t-n+1}), \dots, C_{|V|,m}(t-1)$ and $\mathbf{H}$ are the weights of the connections from projection to hidden layer.

3. An **hidden layer**. The hidden layer is a classical fully connected layer that uses the tanh activation function. The hidden layer takes as input the vector $\mathbf{z}$ and outputs
$$\begin{aligned}\mathbf{y} &= \mathbf{b} + \mathbf{U} \cdot \tanh(\mathbf{z}) \\ &= \mathbf{b} + \mathbf{U} \cdot \tanh(\mathbf{d} + \mathbf{H} \cdot \mathbf{x})\end{aligned}$$
where U is the matrix of weights from hidden to output layer.

4. An **output layer** with $|V|$ neurons. The output layer is a fully connected layer with softmax for classification. This layer takes as input the vector $\mathbf{y}$ and outputs, for each output node i, the probability of word w_i being equal to the true word w_t at step t.
$$\hat{P}(w_i = w_t | w_{t-n+1}, \dots, w_{t-1}) = \frac{e^{y_{w_i}}}{\sum_{i'}^{|V|} e^{y_{w_{i'}}}}$$

The architecture is shown in Figure 14.1. The goal of this network is to maximise the probability of the next word $\hat{P}$, namely, to find the word that has higher probability of being chosen. Equivalently, we can minimise
$$E = -\log \hat{P}(w_i = w_t | w_{t-n+1}, \dots, w_{t-1})$$

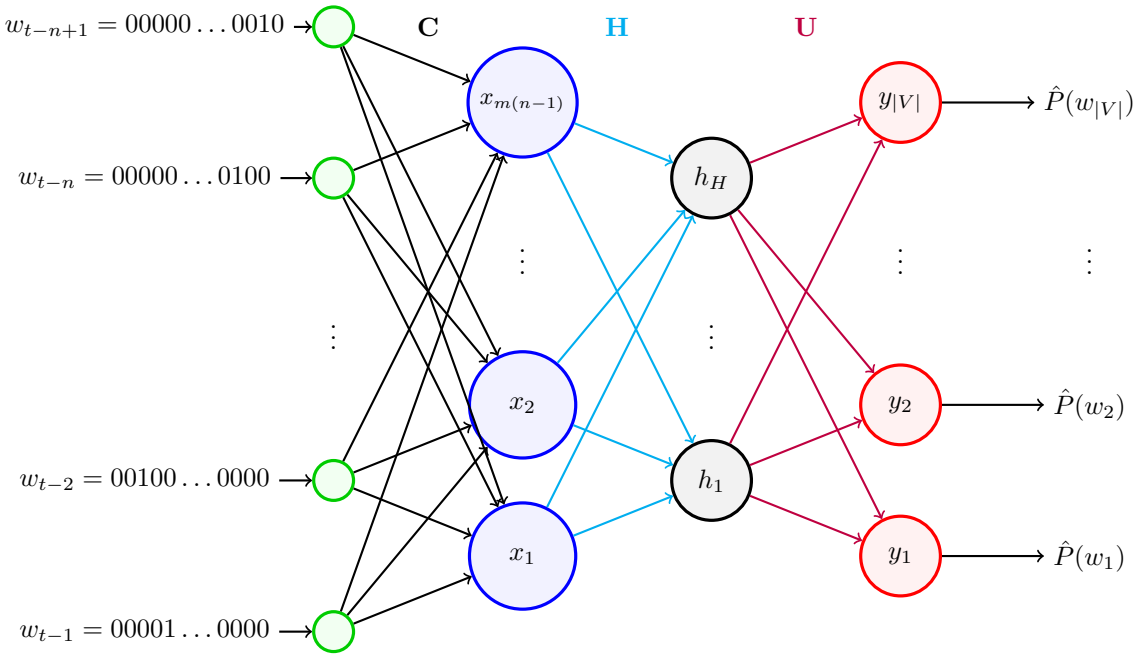


Figure 14.1: The neural net language model.

Encoding matrix

Let aside the weights **H** and **U**, we should focus our attention on the map C. In particular, the mapping between the input and the first hidden layer is obtained through a matrix **C** with dimension $|V| \times m$ where

- $|V|$ is the number of words in the dictionary.
- m is the number of neurons used to represent a word in the hidden layer.

The input layers' neuron values are obtained multiplying the input words in one hot encoding by the matrix **C**. In particular, the input can be represented as a $n \times |V|$ matrix **I**, i.e., a matrix with n rows, each of which is a word fed in input. If we multiply the input **I** by **C** we obtain a $n \times m$ matrix, which has, on each of the n rows, the m-dimensional representation of one input word. In particular, row r has the representation of the r-th input word. If we flatten the matrix $\mathbf{I} \cdot \mathbf{C}$ we obtain the neurons of the hidden layer. Put it in another way, if we call $\mathbf{I}_i$ a row of the matrix **I** and we multiply it by **C**, we obtain a $1 \times m$ vector, which is the m-dimensional representation of the i-th input word. If we concatenate the m-dimensional vectors of each input word, we obtain the first hidden layer.

14.2.2 Word2vec

The NeuralNet language model provides predictions for the next word and an implicit word mapping from the one-hot notation to another space of dimension m. The word2vec model focused on representing a word in a smaller space, instead of predicting the probability of the next word.

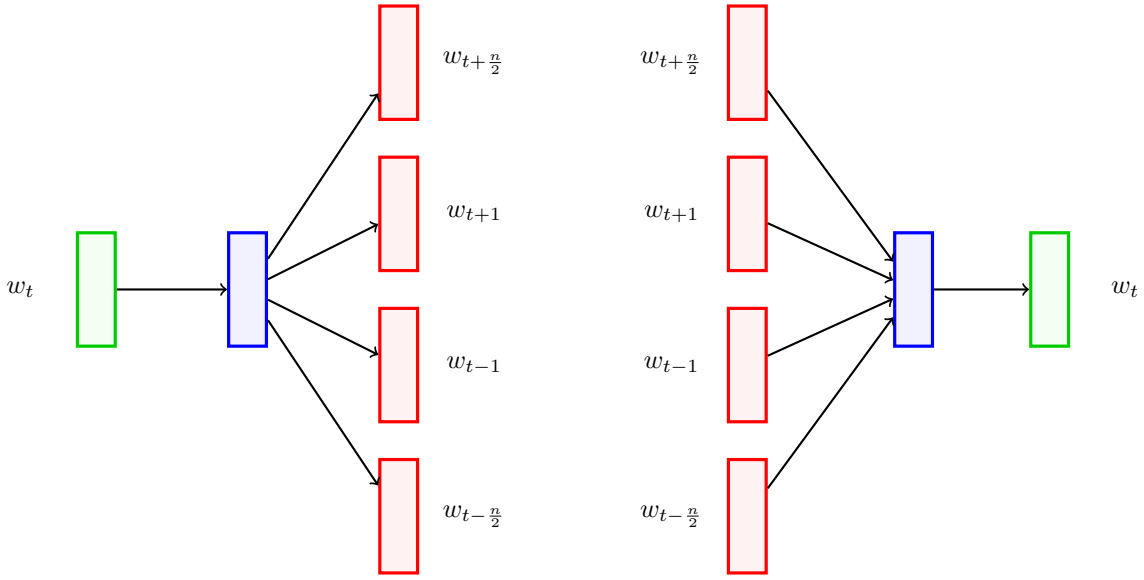
word2vec is a much simpler model with no hidden layer and represents each word using the words around it. This allows to solve all the problems of simple encoding, in fact:

- This representation is dense.
- This representation models similarities between words.
- We can explicitly map words that are not in the dictionary since they are encoded using the words around it.

From the word2vec model (i.e., the NeuralNet model with only the mapping part and shared projection layers), two models originated:

- **Skip-gram**. The skip-gram model takes a word and predicts the words around it.
- **Continuous Bag-Of-Words**. The CBOW model takes the words surrounding a word (i.e., before and after it in time) and uses them to predict the word itself.

These architectures are shown in Figure 14.2.



(a) The skip-gram architecture.

(b) The Continuous Bag-Of-Words architecture.

Figure 14.2: Architecture originated from the word2vec model.

Continuous Bag-Of-Words

The CBOW model takes as input the $\frac{n}{2}$ words before time t and the $\frac{n}{2}$ words after time t, namely

$$\mathbf{I} = (w_{t-\frac{n}{2},\dots,w_{t-1},w_{t+1},\dots,w_{t+\frac{n}{2}}})$$

and outputs the probability, for each word w_i in the dictionary, that word w_t is w_i, namely

$$P(w_i = w_t | w_{t-\frac{n}{2},\dots,w_{t-1},w_{t+1},\dots,w_{t+\frac{n}{2}}})$$

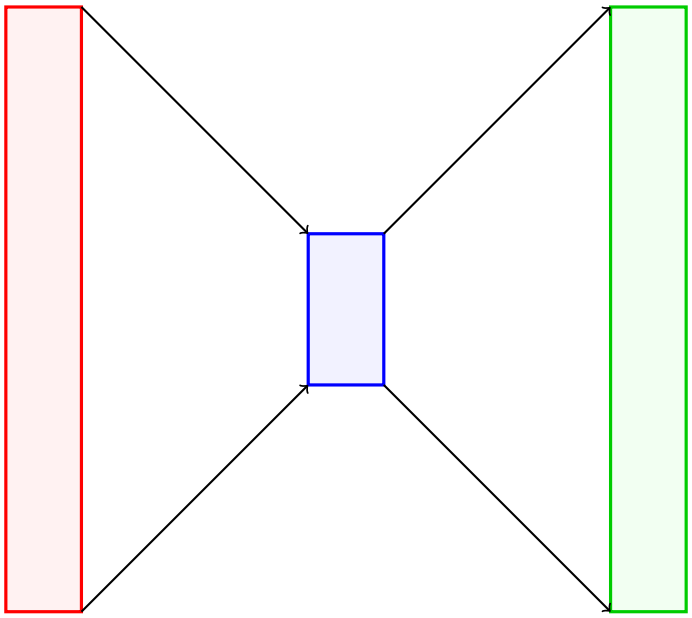


Figure 14.3: The complete CBOW architecture.

A hidden layer, called **projection layer**, is added between input and output layer. This layer has a number of neurons m much smaller than $|V|$ (words in dictionary), typically between 100 and 1000. The activations of these neurons are the m-dimensional representation of the word w_t in output. The objective of the model is to maximise the probability of the correct word, namely to minimise

$$E = -\log \hat{P}(w_t | w_{t-\frac{n}{2},\dots,w_{t-1},w_{t+1},\dots,w_{t+\frac{n}{2}}})$$

The goal is to find an encoding such that we average the encoding of the surrounding words, we get the likelihood of a word. Note that this model can be trained unsupervisedly since we know what is the word to be predicted.

The CBOW architecture uses the autoencoder structure since we map an high dimensional space (words encoded in one-hot) into a lower-dimensional one and then back to the original space (even if in this case the output space is the probability of each word). The main difference is that the matrix $\mathbf{C}$ used for encoding is the same as the matrix used for decoding. In particular, the decoding matrix is obtained as the transpose of $\mathbf{C}^T$. This means that the number of parameters of the network is only the dimension $|V| \times n$ of the matrix $\mathbf{C}$. This architecture is shown in Figure 14.3. Note that, even if this architecture is based on autoencoders, this is radically different. In this case we are encoding a word with the words around it, whilst with autoencoders we used the word itself. This is the reason why word2vec is much better than autoencoders.

Having removed the heavier part of the network, the word2vec model is very light. In particular its complexity is

$$n \cdot m + m \cdot \log |V|$$

Regularities

One of the main advantages of word embedding is modelling how close the meaning of words is. This means that two words with a similar meaning are represented with close vectors.

Experimental results show that names of countries are always at the same distance from names of their capital. This means that if we take the point representing the word *Italy* w_{Italy} and the vector representing the word *Roma*, w_{Roma}, their distance is the same as the one between $w_{England}$ and w_{London}. Moreover, the slope of the line passing between the points is roughly the same in both cases. The same regularities can be found in many other cases.

Moreover, it's possible to do vector operation to obtain the representation of words whose meaning can be roughly obtained from other words. For instance we can compute

$$w_{king} - w_{man} + w_{woman} \simeq w_{queen}$$

The true representation w_{queen} of the word *queen* isn't exactly the one obtained through vector operations, however it's very close.

14.2.3 GloVe

The GloVe model models the probability of two words of being in the same sentence. This model encodes the meaning of a word as vector offsets in an embedding space. In particular, the meaning is encoded by ratios of co-occurrence probabilities.

This model is trained by weighted least squares using the following loss function:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})\big(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\big)$$

This model is sometimes more efficient and easier to use than word2vec, but it depends on the particular problem.

14.3 Neural Turing Machines

Sequence to sequence models are very efficient in many situations, however there is one relevant case in which it performs badly. Say we want to translate a long phrase. In this case, we have to encode the input in a latent vector, decode the latent representation and learn how to structure the embedding space to encode sentences. If we have a small embedding space, we can't encode big sentences with many different words and a very big vocabulary. This problem is called encoding bottleneck. The memory of the system is therefore limited by the size of the embedding space. To solve this problem, we can build a model that includes a memory. This model is called Neural Turing Machine. Note that what we are going to say about NTMs is also shown here. All the credits for the images in this section are to them.

Since we are dealing with memory, the model has to learn what and where to write and read to and from memory. First, let us define what memory is. In NTM, a memory is an array of vectors M_i, as shown in Figure 14.4.

14.3.1 Reading and the attention mechanism

One operation neurons can do, and have to learn to do, is reading from memory. Reading from memory can't be done like in normal computers, moreover it has to be differentiable since we want

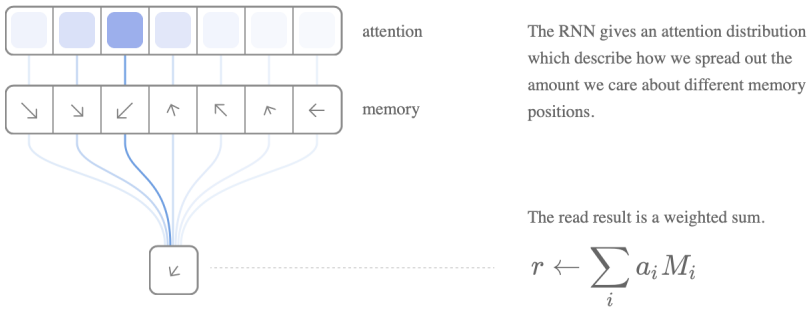


Figure 14.4: Reading from memory in a Neural Turing Machine.

to compute the gradient that has to be used for backpropagation. In particular, we want to make reading differentiable with respect to the location we read from, so that we can learn where to read. The same is true for writing, but let us focus on reading for the time being. Neural Turing Machines solve this problem by using the **attention mechanism**. The idea is to make the network output an attention distribution that describes how we spread the amount we care about different memory location. In other words, say we have a memory of 7 cells and we want to read from the third. The network will output a distribution with the highest probability in 3 and smaller probabilities the more we go far from 3, i.e., something like a normal distribution with mean 3. In practice, the value r read from memory is the weighted average of the values in memory, where the weight is the probability of a cell. Namely, we have

$$r = \sum_i a_i M_i$$

where

- a_i is the probability of cell i.
- M_i is the value of cell i.

The attention mechanism for writing is shown in Figure 14.5.

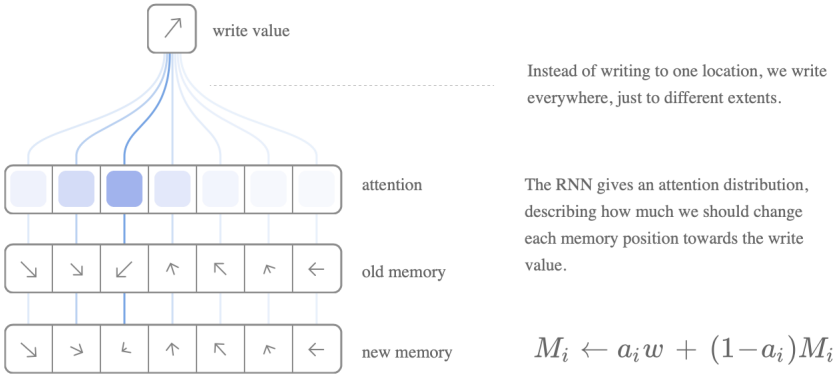


Figure 14.5: Writing to memory in a Neural Turing Machine.

14.3.2 Writing

The attention mechanism can be applied to writing, too. In particular, we can write in every cell at once, but giving more weight to one cell, which is the cell we want to write into. Namely, the values in the cells we don't care about are modified by a small value while the cell we want to write to is modified significantly. In practice, memory cell i is modified as follows:

$$M_i = a_i w + (1 - a_i) M_i$$

where

- M_i is the value of cell i.
- a_i is the probability (i.e., the weight) of writing in cell i.
- w is the value that has to written in memory.

The attention mechanism for reading is shown in Figure 14.6.

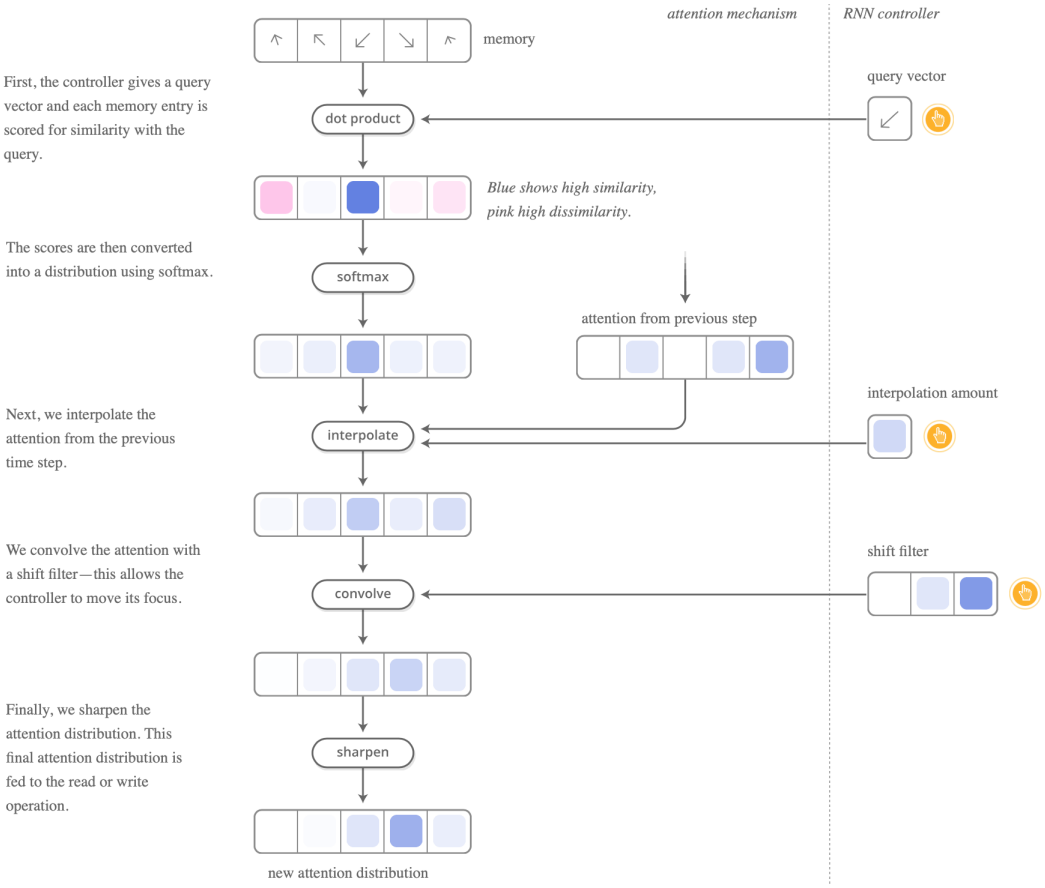


Figure 14.6: Network to obtain the attention distribution in a Neural Turing Machine.

14.3.3 Reading and writing positions

Having understood how writing and reading works, we have to understand how the network decides where to read or write. Basically, we have to analyse how the network generates the probabilities (or weights) a_i. NTMs use, to compute their attention, a combination of

- **Content-based attention**. Content-based attention allows NTMs to search the memory for places that match what they are looking for. This type of attention works like when we, as humans, look for memories related to some object or concept (even if they aren't in general related to it). For instance, if we associate home with dogs, family and a trampoline, our content-based attention, when searching for the concept of home, will look for the locations where dogs, family members and trampolines are located.
- **Location-based attention**. Location-based attention allows NTMs to search the memory for placed recently visited. This type of attention allows NTMs to execute loops.

14.3.4 Neural Network Machines capabilities

Thanks to their memory, NTMs can execute algorithms that weren't available to simpler neural networks. As an example, they can solve the problem we described when introducing NTMs, which is translating long sentences. NTMs can in fact learn to store a long sentence in memory and then loop over it. This is a great advantage with respect to sequence to sequence models, in fact in seq2seq models we could only store the input sentence and then translate its meaning but we couldn't move in the sentence. Moving in the sentence is particularly useful in some languages which put the verb at the end (like German), hence we have to read the whole sentence and then go back to the beginning to understand it.

This isn't all in fact since loops are available, NTMs can learn quite easily to sort sequences of numbers. Bubble sort requires to check couples of adjacent cells, which is quite easy to learn thanks to content- and location-based attention. NTMs can also mimic a lookup table.

Multiple models have been created extending NTMs to go beyond simple memory access. Some examples are:

- Neural GPUs able to add and multiply numbers (which NTMs couldn't do).
- NTMs trained using reinforcement learning.
- Neural Random Access Memories based on pointers.
- NTMs that use data structures like stacks and queues.

14.3.5 Attention mechanism in sequence to sequence models

The attention mechanism can be applied to sequence to sequence models too, and it can be particularly useful when translating sentences (for the reasons we have already discussed). Let us consider a data set

$$\Big\{((x_1,\dots,x_n),(x_1,\dots,x_m))_{i=1}^N$$

The role of the network is to model the generative probability

$$P(y_1,\dots,y_m|x)$$

In seq2seq models, attention is based on content-based attention. In particular:

1. The RNN generates a query describing what it wants to focus on.
2. Each item is dot-producted with the query to produce a score, describing how well it matches the query.
3. The scores are fed into a softmax to create the attention distribution.

More precisely, we have to:

1. Compare the current target hidden state $\mathbf{h}_t$ with the source states $\mathbf{h}_t$ to derive the attention scores:
$$\text{score}(\mathbf{h}_t \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^T \mathbf{W} \bar{\mathbf{h}}_s \\ \mathbf{v}_a^T \tanh(\mathbf{W}_1 \mathbf{h}_t + \mathbf{W}_2 \bar{\mathbf{h}}_s) \end{cases}$$
2. Apply the softmax function to the attention scores and compute the attention weights, one for each encoder token:
$$\alpha_{ts} = \frac{e^{\text{score}(\mathbf{h}_t \bar{\mathbf{h}}_s)}}{\sum_{s'=1}^{S} \text{score}(\mathbf{h}_t \bar{\mathbf{h}}_{s'})}$$
3. Compute the context vector as the weighted average of the source states.
$$\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s$$
4. Combine the context vector with the current target hidden state to get the final attention vector:
$$\mathbf{a}_t = f(\mathbf{c}_t, \mathbf{h}_t) = \tanh(\mathbf{W}_c[\mathbf{c}_t : \mathbf{h}_t])$$

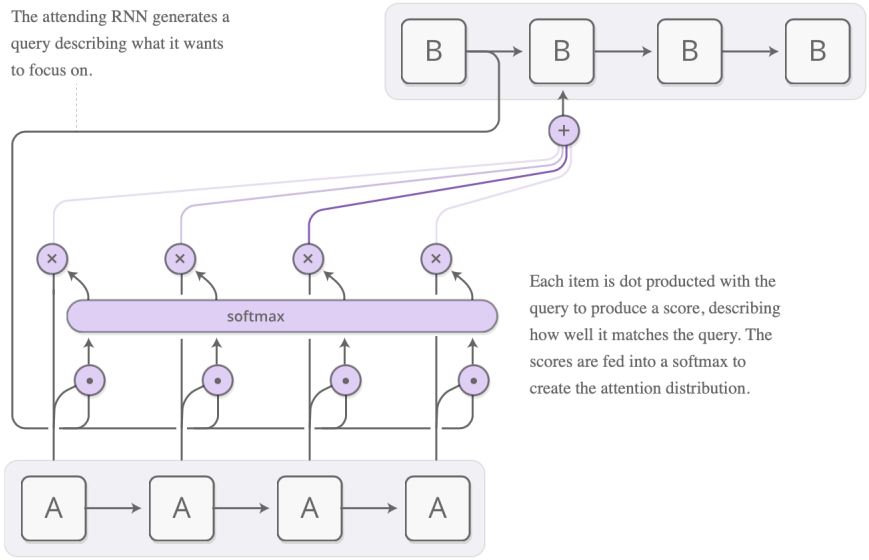


Figure 14.7: Attention mechanism applied to sequence to sequence models.

Attention visualisation

The attention distribution can be visualised. In particular, we can use an alignment matrix to visualise attention weights between source and target sentences. For each decoding step, i.e., for each generated token, the matrix describes which are the source tokens that are more present in the weighted sum that conditioned that decoding. This allows to understand where the model is paying the attention for generating an output token (i.e., an output word).

Attention in some fields

The attention mechanism can be used in many fields:

- In **translation**, the attention mechanism can be used to pass along information about each word the model sees and then for generating the output to focus on words.
- In **voice recognition**, the attention mechanism can be used to pass over an audio recording and focus only on the relevant parts for generating a transcript of the voice in the audio.
- In **image captioning**, the attention mechanism can be used to generate the description of the images (or better of the feature representations generated by a CNN) and to focus on the parts of the image used for generating that description. For instance if the description says that there is a dog, we can visualise where the dog is and that the model used the part of the image where the dog is to generate the word dog in the description.

Definitions

Theorems and principles