TOML lectures: Deep Learning

Jose M. Barcelo Ordinas, Jorge Garcia Vidal and Pau Ferrer Cid

May 26, 2024

Contents

1 Introduction

We recommend the following books on deep learning:

- "Deep Learning" by I. Goodfellow, Y. Bengio and A. Courville. You can find an online free version in `https://www.deeplearningbook.org/`.
- "Dive into Deep Learning" by A. Zhang, Z.C. Lipton, M. Li and A. J. Smola, `https://d2l.ai` and a free version PDF `https://d2l.ai/d2l-en.pdf`
- You can check many of the ideas discussed in these LNs using the configurable NNs in `http://playground.tensorflow.org/`.

This part of the course is divided in the following topics:

1. Feedforward neural network (FNN)
2. Convolutional neural network (CNN)
3. Autoencoders (AE)
4. Recurrent neural network (RNN) and long short-term memory (LSTM)
5. Numerical Computations

We are going to consider along this chapter that each measurement $\boldsymbol{x}$ has P features. i.e., $\boldsymbol{x}$=$(x_1, \ldots, x_P)$. Moreover, we will consider N measurements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$. In case of considering supervised methods, we will consider that the target values are $y_1, \ldots, y_N$.

2 Feedforward neural network (FNN)

A *Feedforward neural network* (FNN) or *multilayer perceptron* (MLP) is a funcion f_W that depends on parameters $W = [W^{(1)}, ..., W^{(L)}]$ (*weights*) that gives an estimation of a r.v. Y given some input values x which are a realization of a r.v. X:

$$\hat{\boldsymbol{x}} = f_W(\boldsymbol{x})$$

Feedforward neural networks are build by *composing* together a number of simpler vectorial functions $f^{(k)}(a^{k-1}; W^{(k)})$. For example:

$$\begin{array}{lll} \boldsymbol{x}^{(1)} & = f^{(1)}(\boldsymbol{x}^{(0)}; W^{(1)}) & = f(\boldsymbol{x}; W^{(1)}) \\ \boldsymbol{x}^{(2)} & = f^{(2)}(\boldsymbol{x}^{(1)}; W^{(2)}) & = f^{(2)}(f^{(1)}(\boldsymbol{x}; W^{(1)}); W^{(2)}) \\ \boldsymbol{y} & = f_W(\boldsymbol{x}) & = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}; W^{(1)}); W^{(2)}); W^{(3)}) \end{array}$$

The outer function in our composition chain is called the *output layer*. The inner function is the *input layer*, while the intermediate functions are called the *hidden layers*. The length of the chain (number of layers) is called the *depth* of the model (including output layer but excluding input layer). In fact, although the input layer could be counted in the depth, what it is important is the number of layers of adaptive weights, implying to remove the input layer in the count. Then, the NN would have a depth of 3 layers (2 hidden layers + 1 output layer).

If the number of compositions is large, we say that our network is a *deep* neural network. The dimension of the image of the hidden layers is called the *width* of the network. Since different layers may have different width some authors name the width of a neural network as the width of the widest layer of the neural network. A *shallow* network has few layers with very high width.

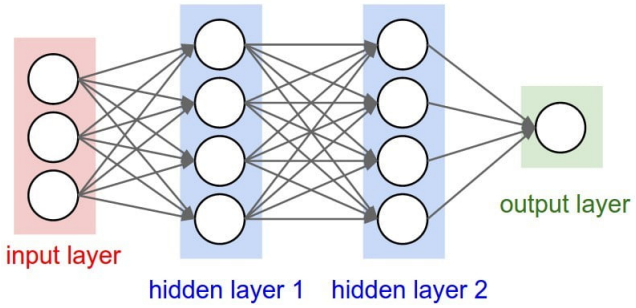


Figure 1: Example of NN with 2 hidden layers (width$^{(1)}$=4, width$^{(2)}$=4, width$^{(3)}$=1, depth=3).

The output layer is usually a classification or regression function, that acts on the output of the hidden layers. One can see the output of the hidden layers as a representation of the input data which is adequate for the classification or regression task of the output layer (i.e. it is a *kernel). This means that the main tasks of the input and hidden layers is to learn the most adequate representation for the data*, a task that is often done by hand or by using criteria like local smoothness when we use other ML methods. This *representation learning* is the key characteristic that differentiate deep neural networks from other ML methods.

2.1 Output and hidden layers

2.1.1 Perceptrons a.k.a neurons

The scalar function $f_i^{(l)}$ that evaluates each component of the vectorial function $f^{(l)}$ at layer l is called a *perceptron* or equivalently a *neuron*. For each perceptron we define a vector of *weights* ($\boldsymbol{W}$), a *bias* ($\boldsymbol{b}$), a *logit* and an *activation*, which

is the output of the perceptron. Let us assume a MLP with three layers (two hidden layers with 3 neurons each hidden layer). The input is a vector $\boldsymbol{x}$ of dimension P. We will focus in the example, see Figure 2, in the neuron $h_2^{(2)}$.

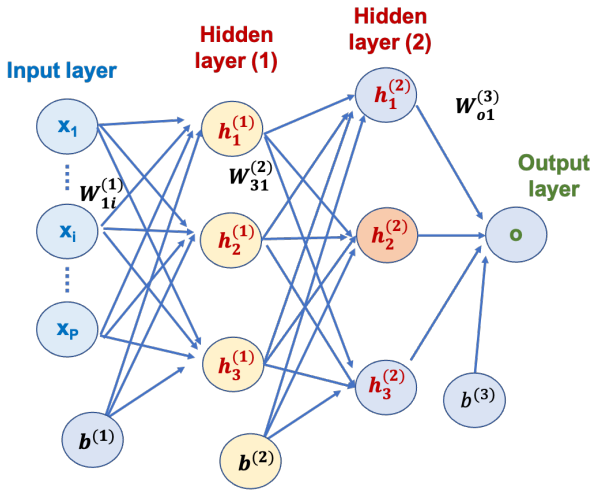


Figure 2: Example of perceptron (neuron).

We define a **logit** $z_i^{(l)}$ as a linear combination of weights multiplied by the inputs of a neuron and a bias. For a specific layer l:

$$z_j^{(l)} = \sum_{i=1}^{h_{l-1}} w_{ji}^{(l)} h_i^{(l-1)} + b_j^{(l)} \tag{2.1.1}$$

where h_{l-1} is the number of neurons in the previous layer and $h_j^{(l-1)}$ are the outputs (also called **activation units**) of the previous layer. We define w_{ji} as the weight that correspond from neuron i to neuron j. The term $b_j^{(l)}$ is the *bias* for the l-layer jth neuron, a fixed term added to the logit which does not depend on the activations of the previous layers. Finally, we define $\boldsymbol{h}^{(0)}=\boldsymbol{x}$. Looking at neuron $h_2^{(2)}$, Figure 2, its logit would be:

$$z_2^{(2)} = \sum_{i=1}^{3} w_{2i}^{(2)} h_i^{(1)} + b_2^{(2)}$$

The output of the neuron is call an **activation unit** and is the consequence of applying an **activation function** $f(\cdot)$ to the logit:

$$h_j^{(l)} = f(z_j^{(l)}) = f(\sum_{i=1}^{h_{l-1}} w_{ji}^{(l)} h_i^{(l-1)} + b_j^{(l)}) \tag{2.1.2}$$

In our example, the activation of neuron $h_2^{(2)}$ will be:

$$h_2^{(2)} = f(z_2^{(2)}) = f(\sum_{i=1}^{3} w_{2i}^{(2)} h_i^{(1)} + b_2^{(2)})$$

We can express this MLP network in vector form:

$$\begin{aligned} \boldsymbol{h}^{(1)} &= f(\boldsymbol{W}^{(1)}\boldsymbol{h}^{(0)} + \boldsymbol{b}^{(1)}) = f(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{h}^{(2)} &= f(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \\ o &= f(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}) \end{aligned} \tag{2.1.3}$$

2.1.2 Activation functions

The layers in a FFN are created by considering activation functions of different types.

1. **Activation functions for output layers:** for **regression problems** the output function for output layers is a *linear combination of inputs*, i.e. $a^{(L)} = z^{(L)}$, meaning that the activation function is the identity.

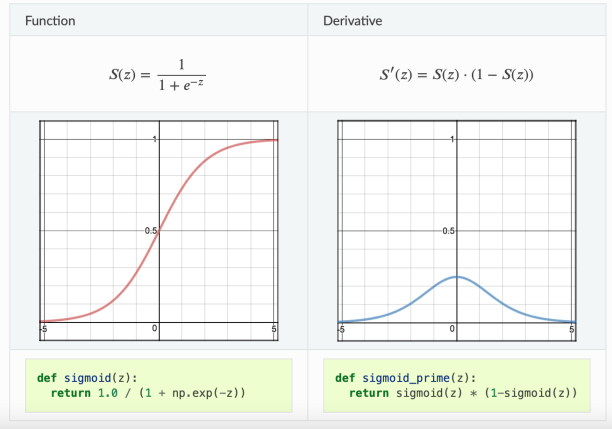


Figure 3: Activation functions: Sigmoid.

For **binary classification** problems a very commonly used activation function for output layers is the *sigmoid* function, Figure 3:

$$\sigma(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} \tag{2.1.4}$$

Note that we assume a single variable as input (z) and a single value for output, which can be interpreted as $p(y = 1|z)$. We can easily check

$\sigma'(z) = \sigma(z)(1 - \sigma(z))$. The sigmoid function squashes the result between [0,1], with 0 for negative arguments below a threshold, and 1 for positive arguments above a threshold. For arguments close to 0, it behaves as a linear function.

For **multiclass classification** we usually use for output a single-hot vector, and a common used activation function for output layers is the *softmax* function:

$$softmax(z_1, ..., z_n) = (\frac{e^{z_1}}{\sum_j e^{z_j}}, ..., \frac{e^{z_n}}{\sum_j e^{z_j}}) \tag{2.1.5}$$

Note that we assume a vector input of numbers and a vector output of probabilities $p(y_i = 1|z)$. So, the softmax takes as inputs n numbers and output a probability distribution over these n inputs. Then, the softmax acts as a smooth approximation to the *arg max* function: the function whose value is which index has the maximum. Very often we will use the log of the softmax function:

$$log\ softmax(z)_i = z_i - log \sum_j exp(z_j) \tag{2.1.6}$$

In general, mathematically, the softmax (sigmoid) function works well for classification, but computationally, we can suffer from numerical underflow and exponentiation overflow. For example, assume that our output is large value (z large), then e^z grows being larger than the data type supported by the computer, and we suffer *overflow*. On the other hand, when the output es very negative, we will suffer *underflow*. A solution to overcome underflow and overflow is to substract the max of the outputs, $\bar{z} = max_k\ z_k$:

$$\hat{y}_j = \frac{e^{z_j}}{\sum_j e^{z_j}} = \frac{e^{(z_j - \bar{z})} e^{\bar{z}}}{\sum_j e^{(z_j - \bar{z})} e^{\bar{z}}} = \frac{e^{(z_j - \bar{z})}}{\sum_j e^{(z_j - \bar{z})}} \tag{2.1.7}$$

prevents overflow since the numerator is always lower than 1 and the denominator is between [1,q]. However, it does not prevent underflow. For preventing overflow and underflow, we can take logarithms:

$$log\ \hat{y}_j = log\ \frac{e^{(z_j - \bar{z})}}{\sum_j e^{(z_j - \bar{z})}} = z_j - \bar{z} - log \sum_j e^{(z_j - \bar{z})} \tag{2.1.8}$$

2. **Activation functions for hidden layers** If the activation functions of all the hidden units in a network are taken to be linear, then for any such network we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation. Thus, there is few interest in neural networks with linear activation functions, and they typically are non-linear.

Sigmoids were initially used because of their good characteristics, such as being smooth and differentiable. However, sigmoids cause problems in cost function optimization, as their gradient disappears (**vanish gradient**) for large positive and negative arguments. Therefore, they are now only used for output activation units and not for hidden layers.

A classical activation function for hidden layers is the *hyperbolic tangent (tanh)*, Figure 4:

$$tanh(z) = 2(\sigma(2z) - 1/2) \tag{2.1.9}$$

The hyperbolic tangent tanh squashes the output between the interval [-1,1], and its derivative is symmetric with respect to the 0, with a maximum equal to 1 at the origin, and when the argument moves away from 0, the derivative goes to 0. Thus, tanh helps in centering the data since the mean for the hidden layer comes out to be 0 or very close to it. It also faces the problem of vanishing gradients, look at the gradient (Figure 4), similar to the sigmoid activation function. Plus the gradient of the tanh function is much steeper as compared to the sigmoid function. We can see in figures 3 and 4 that the sigmoid gradient reaches its maximum at a value of about 0.2 at z=0, and the tanh gradient reaches a maximum of 1 at z=0. That means that using the tanh activation function results in higher values of gradient during training and higher updates in the weights of the network. Moreover, since tanh is zero centered, in general behaves well with respect sigmoids, and converges faster than sigmoids.

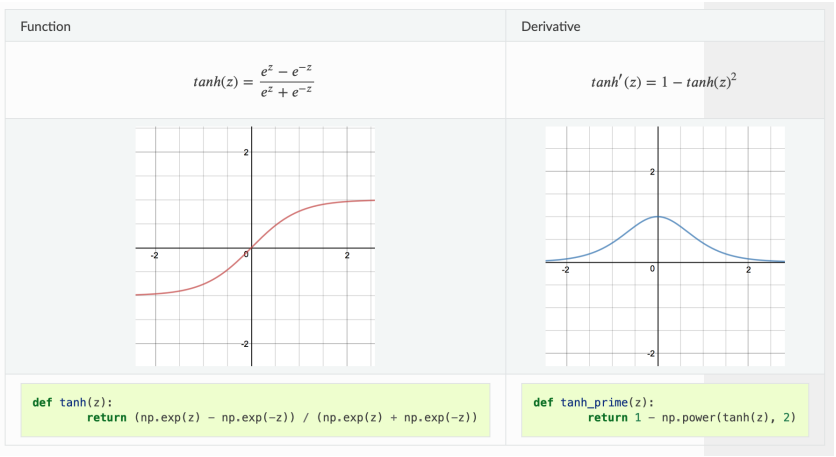


Figure 4: Activation functions: Tanh.

The most commonly used activation function in hidden layers is the *ReLU function*, Figure 5:

$$ReLU(z) = max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{2.1.10}$$

The goal of the ReLU is to discard negative values and retain positive values. The ReLU function has derivative 0, Figure 5, when the value is negative and derivative 1 when the value is positive. When the value is 0, by definition, the derivative is set to 0. The ReLU function behaves then well in the optimization of the cost function, avoiding vanishing gradients. ReLu has the advantage that it does not activate all neurons at the same time. This makes ReLu more efficient than sigmoid and tanh, as it has a non-saturating property. The drawback of ReLu is that for negative values, the gradient is always zero, which means that during backpropagation, the weights and biases of the neurons are not updated and are always zero (deactivated or dead). This is called the **dying ReLu problem**.

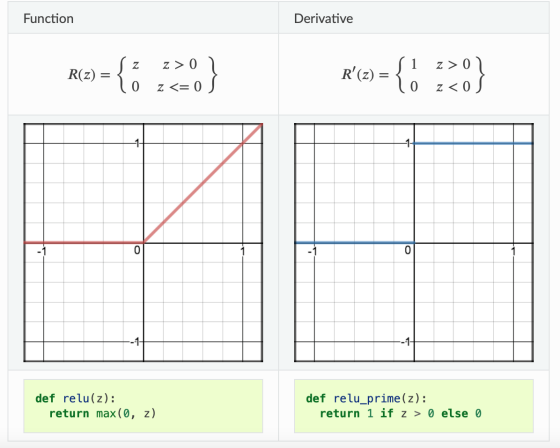


Figure 5: Activation functions: ReLU.

There are several variants of the ReLU function, like the *leaky ReLU*, Figure 6 that keeps some information when the value is negative by multiplying by a fixed (small) coefficient α. The goal is to solve the dying ReLU problem as it has a small positive slope in the negative area. It has all the advantages of ReLu, solves the problem of dying ReLu, but sometimes gives inconsistent results for negative input values.

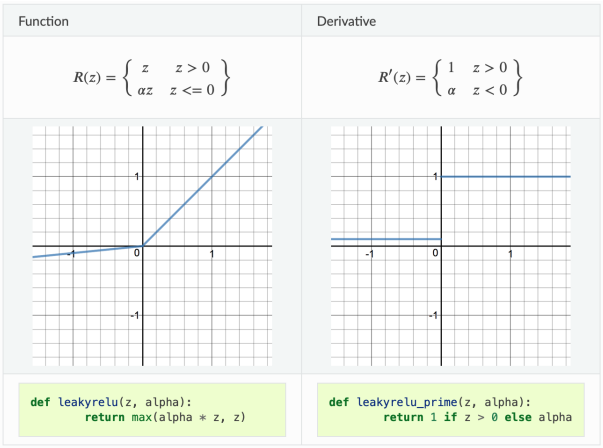


Figure 6: Activation functions: LeakyReLU.

$$LeakyReLU(z) = max(\alpha z, z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases} \tag{2.1.11}$$

Parametric ReLUs (pReLU) take the leaky ReLU further by making the coefficient α of leakage into a parameter that is learned along with the other neural network parameters.

The *Maxout* activation function is a generalization of ReLU. The input to a maxout function is a vector, which is divided into groups of k. The ouput is a vector. Each component of the output vector is the maximum value of a given group of k input coordinates, for instance:

$$\begin{aligned} maxout(z_1, z_2, z_3, z_4, z_5, z_6) &= (maxout_1, maxout_2) \\ maxout_1(z_1, z_2, z_3, z_4, z_5, z_6) &= max(0, z_1, z_2, z_3) \\ maxout_2(z_1, z_2, z_3, z_4, z_5, z_6) &= max(0, z_4, z_5, z_6) \end{aligned} \tag{2.1.12}$$

The main drawback of Maxout is that it is computationally expensive as it increases the number of parameters for each neuron.

Other alternatives are exponential linear units (ELUs) Function, the Swish function, the Gaussian error linear unit (GELU) function, the scaled exponential linear unit (SELU) function, etc. As a rule of thumb, you can begin with using the ReLU activation function and then move over to other activation functions if ReLU doesn't provide optimum results.

2.2 Neural networks

2.2.1 Linear Neural networks

Let us assume that we have a vector of p features and we take minibatches of size n among the N measurements.

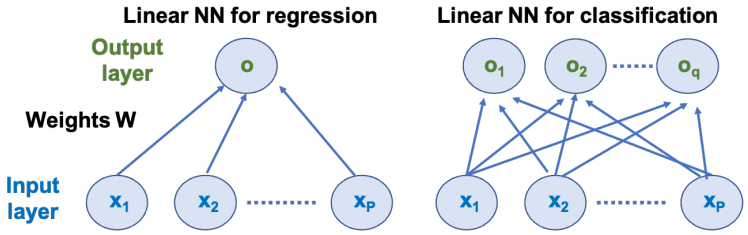


Figure 7: Linear neural networks for regression and classification.

A linear neural network model for regression, Figure 7, considers the following equations:

$$o = w_{11}x_1 + w_{12}x_1 + \cdots + w_{1p}x_p + b \tag{2.2.1}$$

Let us assume that we consider minibatches of size n, and we organize the inputs in matrix $\boldsymbol{X}\in\mathbb{R}^{n\times p}$ (where p is the number of features), and we have, in the regression problem, a single output, so, $\boldsymbol{w}\in\mathbb{R}^{p}$, and $b\in\mathbb{R}$. Now, we can express the linear neural network for classification in vector notation as:

$$\begin{aligned} \boldsymbol{o} &= \boldsymbol{X}\boldsymbol{w} + b \\ \hat{\boldsymbol{y}} &= \boldsymbol{o} \end{aligned} \tag{2.2.2}$$

with $\hat{\boldsymbol{y}}\in\mathbb{R}^{n}$.

For the linear neural network model for classification, Figure 7, the following equations are considered:

$$\begin{aligned} o_1 &= w_{11}x_1 + w_{12}x_1 + \cdots + w_{1p}x_p + b_1 \\ o_2 &= w_{21}x_1 + w_{22}x_1 + \cdots + w_{2p}x_p + b_2 \\ &\cdots \\ o_q &= w_{q1}x_1 + w_{q2}x_1 + \cdots + w_{qp}x_p + b_q \end{aligned} \tag{2.2.3}$$

Let us assume that we consider minibatches of size n, and we organize the inputs in matrix $\boldsymbol{X}\in\mathbb{R}^{n\times p}$ (where p is the number of features), and we have, in the classification problem, an output of q classes, so, $\boldsymbol{W}\in\mathbb{R}^{p\times q}$, and $\boldsymbol{b}\in\mathbb{R}^{1\times q}$. Now, we can express the linear neural network for classification in vector notation as:

$$\begin{aligned} \boldsymbol{O} &= \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b} \\ \hat{\boldsymbol{Y}} &= softmax(\boldsymbol{O}) \end{aligned} \tag{2.2.4}$$

with $\hat{\boldsymbol{Y}}\in\mathbb{R}^{n\times q}$.

2.2.2 Nonlinear neural networks

In order to apply nonlinear models to the classification and regression of data, we add hidden layers. This model is also called *multi-layer perceptron (MLP)*. For example, in Figure 8, a hidden layer with a width of h neurons have been added.

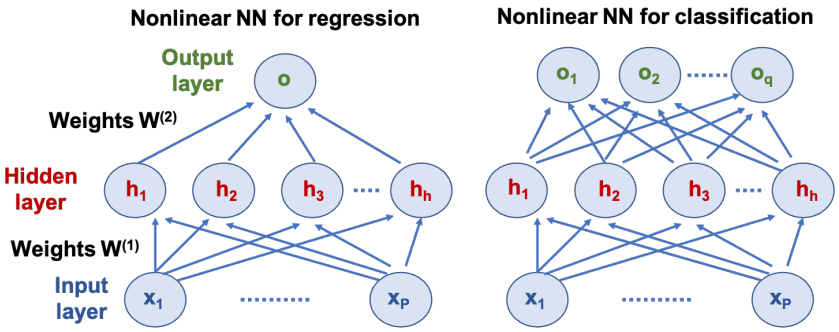


Figure 8: Nonlinear neural networks for regression and classification.

Let us assume that we consider minibatches of size n, and we organize the inputs in matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ (where p is the number of features), and we have, in the classification problem, an output of q classes. the weights and biases are defined as $\boldsymbol{W}^{(1)} \in \mathbb{R}^{p \times h}$, $\boldsymbol{b}^{(1)} \in \mathbb{R}^{1 \times h}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{h \times q}$ and $\boldsymbol{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Now, we can express the linear neural network for classification in vector notation as:

$$\begin{aligned} \boldsymbol{H} &= \phi(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{O} &= softmax(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\ \hat{\boldsymbol{Y}} &= \boldsymbol{O} \end{aligned} \tag{2.2.5}$$

We note that the activation units, $\phi(\cdot)$, in the hidden layer can not be linear units, since then, we would again obtain a linear model. To see this, assume a model with hidden layer linear activation units, it is to say, $\phi(\cdot)$ is the identity. Then:

$$\begin{aligned} \boldsymbol{O} &= softmax(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\ &= softmax((\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)})\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\ &= softmax(\boldsymbol{X}\boldsymbol{W}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\ &= softmax(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}) \end{aligned} \tag{2.2.6}$$

We can observe that eq.(2.2.6) is the same that the linear case, eq.(2.2.2) or eq.(2.2.4). As a conclusion, we have to use nonlinear activation $\phi(\cdot)$ units in the

hidden layer in order to build a nonlinear neural network:

$$\begin{aligned} \boldsymbol{H} &= \phi(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{O} &= softmax(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\ \hat{\boldsymbol{Y}} &= \boldsymbol{O} \end{aligned} \tag{2.2.7}$$

with $\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times q}$.

The regression case is similar but using a identity as output activation function: $\hat{\boldsymbol{y}}=\boldsymbol{o}$, $\boldsymbol{W}^{(1)} \in \mathbb{R}^{p \times h}$, $\boldsymbol{b}^{(1)} \in \mathbb{R}^{1 \times h}$, $\boldsymbol{w}^{(2)} \in \mathbb{R}^{h}$ and $b^{(2)} \in \mathbb{R}$:

$$\begin{aligned} \boldsymbol{H} &= \phi(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{o} &= \boldsymbol{H}\boldsymbol{w}^{(2)} + b^{(2)} \\ \hat{\boldsymbol{y}} &= \boldsymbol{o} \end{aligned} \tag{2.2.8}$$

2.3 Gradient Descent (SGD) - reminder

The most commonly used optimization method is the *gradient descent (GD) algorithm*, which requires the evaluation of the gradients $\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta})$ for the different elements of the set $\mathcal{T}$. For simplicity, let us assume a mean square error (MSE) without regularization as cost function with N samples:

$$L(\boldsymbol{\beta}, \boldsymbol{x}_n) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n; \boldsymbol{\beta}))^2 \tag{2.3.1}$$

where in $L(\boldsymbol{\beta}, \boldsymbol{x}_n)$ we explicitly express that the number of measurements has size n. Then, now let us take the gradient of the cost function:

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}, \boldsymbol{x}_n) = \frac{1}{N} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n; \boldsymbol{\beta})) \nabla_{\beta} f(\boldsymbol{x}_n; \boldsymbol{\beta}) \tag{2.3.2}$$

Now, the **batch gradient descent (BGD)** method updates the coefficients $\boldsymbol{\beta}^{(k)}$ until some criterion is satisfied (e.g., $||\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k)}, \boldsymbol{x}_n)||_2^2 \leq \epsilon$, for some ϵ>0):

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \gamma^{(k)} \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k)}, \boldsymbol{x}_n) \tag{2.3.3}$$

with $\gamma^{(k)}$ the step size that usually can be a fixed value or can be obtained using the *backtracking line search algorithm* (based on the Wolf condition). This algorithm, if trained with all samples N is called *batch gradient descent (BGD)*.

However, usually, as the size of the training set can be very large, instead of computing the sum for all the terms of the training set, *at each step we only evaluate the sum for a limited number of elements of* $\mathcal{T}$, which are randomly chosen (these chosen elements are called *minibatch*). This variation of the gradient descent method is called **stochastic gradient descent (SGD)** when the minibatch size B=1,

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \gamma^{(k)} \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k)}, \boldsymbol{x}_{n,n+1}) \tag{2.3.4}$$

Another variation is the **minibatch stochastic gradient descent (MBGD)** where the minibatch is of arbitrary size $B>1$.

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \gamma^{(k)} \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k)}, \boldsymbol{x}_{n,n+B}) \tag{2.3.5}$$

In general, SGD is good for on-line computation but the cost function fluctuates with high variance, while MBGD with mini-batches among d=50 to 256 reduces the variance of the stochastic gradient descent algorithm improving stability.

Note that in case of using MBGD, the cost function reduces to:

$$L(\boldsymbol{\beta}) = \frac{1}{2|B|} \sum_{n \in B} (y_n - f(\boldsymbol{x_n}; \boldsymbol{\beta}))^2 \tag{2.3.6}$$

where B is the set considered in the minibatch and $|B|$ its cardinality.

For example, in Figure 9, BGD, SGD and MBGD are compared for a deep learning application. BGD decays smoothly while SGD converges faster (see 100 iterations), but "noisy", however after 1000 iterations SGD has an accuracy of 92% and BGD of 96%, so in the long-run BGD outperforms SGD. MBGD is in the middle ground in terms of speed of convergence, but outperforms the others in accuracy (98% accuracy in 150 iterations). In addition, MBGD can be easily "parallelized".

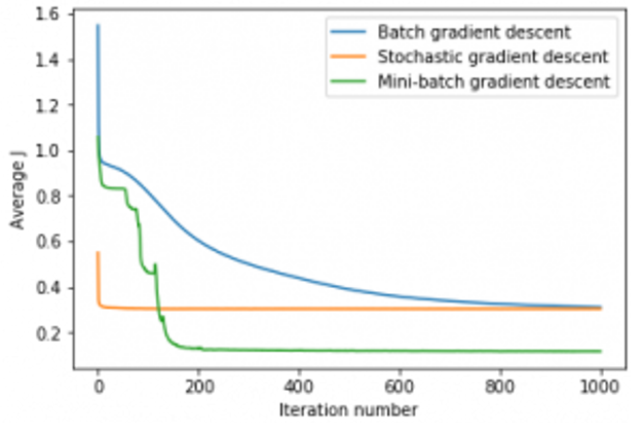


Figure 9: BGD, SGD and MBGD for a deep learning application.

Computing the stochastic gradient reduces the computational effort in a factor $\frac{N}{B}$, which can be very large (e.g., $N = 10^6$, $B = 256$), however the convergence path is more noisy in the case of SGD than in GD; see Figure 10 (also see Figure 9) where BGD, SGD and MBGD are compared.

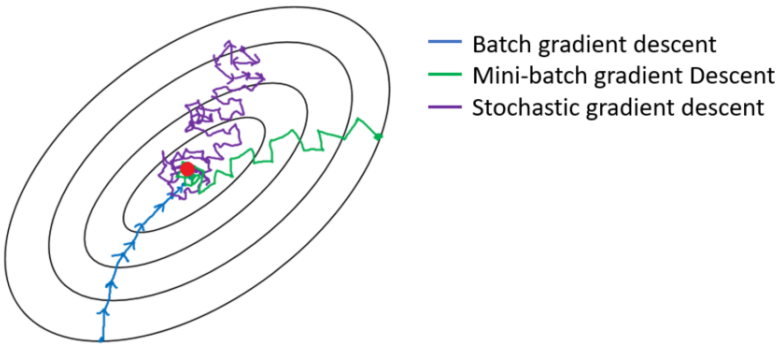


Figure 10: Noisy behavior of the SGD algorithm.

There are several options for computing the learning rate $\gamma^{(k)}$. The simplest choice of $\gamma^{(k)}$ at each step k is a constant learning rate γ_0. Simple dynamic choices of the learning rate $\gamma^{(k)}$ is to use an exponential decay $\gamma^{(k)}{=}\gamma_0 e^{-\lambda k}$, or a polynomial decay such as $\gamma^{(k)}{=}\gamma_0(\eta k+1)^{-\alpha}$ for some constants η, α.

The convergence of SGD is usually more sensitive to the learning rate than in GD. The usual convergence path starts pointing to the optimal solution but in the proximity area of the optimal solution it starts to take random steps. This is a problem in many optimization problems but may be not that important in ML optimization, as can be seen as a form of regularization in the solution, avoiding overfitting.

2.4 Backpropagation

When we apply the SGD algorithm, we must compute at each step the gradient of the loss function. As the number of parameters of the network can be very large, we must optimize the computation of this gradient. The so called *backpropagation algorithm* applies the chain rule of differentiation recursively to achieve an efficient computation of these gradients, both in terms of computations and memory. For simplifying the notation we do not include the biases, but as we mentioned before, biases are usually required in order to have an adequate approximation of our models.

Assume a L layer FNN network with width h for the first $L-1$ layers, and a width of 1 for the ouput layer. Let us see an example, Figure 8, in which we consider a single measurement (either regression or classification) with a cost

function that includes regularization:

$$\begin{aligned}
&\boldsymbol{h} = \phi(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) = \phi(\boldsymbol{W}^{(1)}\boldsymbol{x}) \\
&\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h} \\
&\hat{\boldsymbol{y}} = \boldsymbol{o} \\
&J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + s = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \frac{\lambda}{2}\sum_{l=1}^{L} ||\boldsymbol{W}^{(l)}||_F^2
\end{aligned} \tag{2.4.1}$$

where $||\cdot||_F$ is the Frobenius norm and for simplification we have removed the bias terms. For example, in the case of regression and a minibatch of size n, f is the identity, and the loss function is $L(\boldsymbol{o}, \boldsymbol{y})=\frac{1}{2n}||\boldsymbol{o}-\boldsymbol{y}||_2^2$, and the cost function results in:

$$J = \frac{1}{2n}||\boldsymbol{o} - \boldsymbol{y}||_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L} ||\boldsymbol{W}^{(l)}||_F^2 \tag{2.4.2}$$

Remember that the bias $\boldsymbol{b}^{(l)}$ components are not used in the regularization term (as we saw when we studied multiple linear regression with regularization). In any case, for simplicity in the following we will assume that we do not have bias.

In order to evaluate the gradient of a given entry $(\boldsymbol{x}, y)$ of the training set, we must do the following:

- *Forward step*: For the current values of weigths $\boldsymbol{W}^{(l)}$ and input $\boldsymbol{x}$, compute the values of the activations, logits, and output of the network, see eq.(2.4.1). The *forward step* (for a single measurement, n=1) consists of first computing:

  $$\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x} \tag{2.4.3}$$

  where $\boldsymbol{x}\in\mathbb{R}^p$, $\boldsymbol{W}^{(1)}\in\mathbb{R}^{h\times p}$, and $\boldsymbol{z}\in\mathbb{R}^h$. Then, we compute:

  $$\boldsymbol{h} = \phi(\boldsymbol{z}) \tag{2.4.4}$$

  where $\boldsymbol{h}\in\mathbb{R}^h$. Now, we compute the output as:

  $$\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h} \tag{2.4.5}$$

  where $\boldsymbol{h}\in\mathbb{R}^h$ and $\boldsymbol{W}^{(2)}\in\mathbb{R}^{q\times h}$, and $\boldsymbol{o}\in\mathbb{R}^q$.

- *Backward step*: Compute recursively the gradients of the cost function J with respect the weights $\boldsymbol{W}^{(l)}$ (and biases $\boldsymbol{b}^{(l)}$). The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters.

  For computing the gradients we *simply apply the chain rule starting with the outer layers*. Let us assume that Y=f(X), and Z=g(Y)=g(f(X)). The chain rule states that:

  $$\frac{\partial Z}{\partial X} = \frac{\partial Z}{\partial Y}\frac{\partial Y}{\partial X} \tag{2.4.6}$$

We now apply the chain rule to the cost function J:

$$\frac{\partial J}{\partial L} = 1; \;\; \frac{\partial J}{\partial s} = 1 \tag{2.4.7}$$

Now, we go in the backward direction, computing gradients:

$$\frac{\partial s}{\partial \boldsymbol{W}^{(1)}} = \lambda \boldsymbol{W}^{(1)}; \;\; \frac{\partial s}{\partial \boldsymbol{W}^{(2)}} = \lambda \boldsymbol{W}^{(2)} \tag{2.4.8}$$

and:

$$\frac{\partial J}{\partial \boldsymbol{o}} = \frac{\partial J}{\partial L}\frac{\partial L}{\partial \boldsymbol{o}} = \frac{\partial L}{\partial \boldsymbol{o}} \tag{2.4.9}$$

where $\partial J/\partial \boldsymbol{o} \in \mathbb{R}^q$. We observe here that if we are in a regression problem, $L(\boldsymbol{o}, \boldsymbol{y})=1/(2n)||\boldsymbol{o} - \boldsymbol{y}||_2^2$, and $\partial J/\partial \boldsymbol{o} = 1/n(\boldsymbol{o} - \boldsymbol{y})$. We now obtain the gradient $\partial J/\partial \boldsymbol{W}^{(2)} \in \mathbb{R}^{q \times h}$:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial J}{\partial L}\frac{\partial L}{\partial \boldsymbol{o}}\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W}^{(2)}} + \frac{\partial J}{\partial s}\frac{\partial s}{\partial \boldsymbol{W}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{o}}\boldsymbol{h}^\top + \lambda \boldsymbol{W}^{(2)} \tag{2.4.10}$$

Now, we have to obtain the gradient with respect $\boldsymbol{W}^{(1)}$. First, we need some intermediate results, like $\partial J/\partial \boldsymbol{h} \in \mathbb{R}^h$:

$$\frac{\partial J}{\partial \boldsymbol{h}} = \frac{\partial J}{\partial \boldsymbol{o}}\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} = (\boldsymbol{W}^{(2)})^\top \frac{\partial J}{\partial \boldsymbol{o}} \tag{2.4.11}$$

and $\partial J/\partial \boldsymbol{z} \in \mathbb{R}^h$:

$$\frac{\partial J}{\partial \boldsymbol{z}} = \frac{\partial J}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \frac{\partial J}{\partial \boldsymbol{h}} \odot \frac{\partial \phi(z)}{\partial \boldsymbol{z}} \tag{2.4.12}$$

where $\odot$ is the Hadamard product (element-wise product between matrices). Finally, we obtain $\partial J/\partial \boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times p}$:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial J}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}^{(1)}} + \frac{\partial J}{\partial s}\frac{\partial s}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial J}{\partial \boldsymbol{z}}\boldsymbol{x}^\top + \lambda \boldsymbol{W}^{(1)} \tag{2.4.13}$$

We can observe that the gradients can be computed at each layer recursively from the output towards the imput. Then, we can update $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ using the minibatch gradient descent algorithm (similarly we can obtain the gradient of the biases $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$).

3 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are specialized NNs for data with a grid topology such as images or time-series data. The reason for calling "convolutional" to these NN's is because the main mathematical operation that they perform is a convolution.

3.1 Convolutions

A convolution is a mathematical operation between two functions f, w: $\mathbb{R}^n \to \mathbb{R}$ that is defined as:

$$s(\boldsymbol{x}) = (f * w)(\boldsymbol{x}) \stackrel{def}{=} \int_{-\infty}^{\infty} f(\boldsymbol{u})\ w(\boldsymbol{x} - \boldsymbol{u}) du \tag{3.1.1}$$

The discrete version is defined similarly by substituting integrals for summations:

$$s(i) = (f * w)(i) \stackrel{def}{=} \sum_{m=-\infty}^{\infty} f(m)\ w(i - m) \tag{3.1.2}$$

Convolutions are commutative, meaning that $s(i) = (f * w)(i) = (w * f)(i)$:

$$s(i) = \sum_{m=-\infty}^{\infty} f(m) w(i - m) = \sum_{m=-\infty}^{\infty} f(i - m)\ w(m) \tag{3.1.3}$$

The convolutions are extended to the case of having tensors (more dimensions). For example, a 2D tensor:

$$s(i, j) = (f * w)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n)\ w(i - m, j - n) \tag{3.1.4}$$

When dealing with convolutions, s($\boldsymbol{x}$)=(f∗w)($\boldsymbol{x}$), in deep learning terminology, s($\boldsymbol{x}$) is the *output* or *feature map*, the function f is the *input* and the function w is called the *kernel*. Convolutions measure the overlap between f and g when one function is "flipped" and shifted by $\boldsymbol{x}$. "Flip" means that, looking into the commutative version, as the index of the kernel increases, the index of the input decreases. In NN, a variation of the convolution called *cross-correlation* (although usually also called "convolution" in the NN literature) is used:

$$s(i, j) = (f * w)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(i, j)\ w(i + m, j + n) \tag{3.1.5}$$

3.2 The convolutional hidden unit

The convolutional layers of a CNN apply learned (trained) filters to the input images to create feature maps that summarise the presence of those features in the input. The stacking of convolutional layers allows the first convolutional layers to learn low-level features (e.g. lines) and the deeper layers of the model to learn higher-order or more abstract features, such as shapes or concrete objects.

Let us assume that we handle with images (2D), and for the moment we do not consider channels (e.g., RGB format). Let $X_{i,j}$ represents the pixel at position (*i-th*, *j-th*), and $H_{i,j}$ its hidden representation. We define a kernel matrix V

and a bias matrix U. Then the convolution (cross-correlation) consists of the operation:

$$H_{i,j} = U + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} V_{a,b}\, X_{i+a,j+b} \tag{3.2.1}$$

where (-Δ,Δ) is the *convolution window*, and usually is the shape of kernel V. The reason to keep V thresholded by Δ_1,Δ_2 is to follow the principle of *locality*, and thus we should not have to look very far away from location (i, j). For this reason, we define $V_{ab} = 0$ for values of a>Δ or b>Δ. In addition, U, the bias or offset, is constant, since the convolution has to be invariant to translations. Let us take an example in which we have a 3×4 (height×width) input matrix $\boldsymbol{X}$ and a 2×2 (height×width) kernel matrix $\boldsymbol{V}$, Figure 11:

$$\boldsymbol{X} = \begin{bmatrix} X_{0,0} & X_{0,1} & X_{0,2} & X_{0,3} \\ X_{1,0} & X_{1,1} & X_{1,2} & X_{1,3} \\ X_{2,0} & X_{2,1} & X_{2,2} & X_{2,3} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 3 & 5 \\ 2 & 4 & 6 & 8 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\boldsymbol{V} = \begin{bmatrix} V_{0,0} & V_{0,1} \\ V_{1,0} & V_{1,1} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 0 \end{bmatrix}$$

Computing the convolution outputs (not including the bias terms) results in:

$$\begin{aligned}
H_{0,0} &= X_{0,0}V_{0,0} + X_{0,1}V_{0,1} + X_{1,0}V_{1,0} + X_{1,1}V_{1,1} = 0{\times}1 + 1{\times}2 + 1{\times}2 + 4{\times}0 = 4 \\
H_{0,1} &= X_{0,1}V_{0,0} + X_{0,2}V_{0,1} + X_{1,1}V_{1,0} + X_{1,2}V_{1,1} = 1{\times}1 + 3{\times}2 + 4{\times}2 + 6{\times}0 = 15 \\
H_{0,2} &= X_{0,2}V_{0,0} + X_{0,3}V_{0,1} + X_{1,2}V_{1,0} + X_{1,3}V_{1,1} = 3{\times}1 + 5{\times}2 + 6{\times}2 + 8{\times}0 = 25 \\
H_{1,0} &= X_{1,0}V_{0,0} + X_{1,1}V_{0,1} + X_{2,0}V_{1,0} + X_{2,1}V_{1,1} = 1{\times}1 + 4{\times}2 + 1{\times}2 + 2{\times}0 = 11 \\
H_{1,1} &= X_{1,1}V_{0,0} + X_{1,2}V_{0,1} + X_{2,1}V_{1,0} + X_{2,2}V_{1,1} = 4{\times}1 + 6{\times}2 + 2{\times}2 + 3{\times}0 = 20 \\
H_{1,2} &= X_{1,2}V_{0,0} + X_{1,3}V_{0,1} + X_{2,2}V_{1,0} + X_{2,3}V_{1,1} = 6{\times}1 + 8{\times}2 + 3{\times}2 + 4{\times}0 = 28
\end{aligned} \tag{3.2.2}$$

and

$$H = \begin{bmatrix} H_{0,0} & H_{0,1} & H_{0,2} \\ H_{1,0} & H_{1,1} & H_{1,2} \end{bmatrix} = \begin{bmatrix} 4 & 15 & 25 \\ 11 & 20 & 28 \end{bmatrix}$$

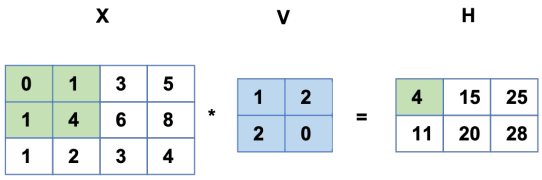


Figure 11: Example of a convolution operation with a 3×4 input and a 2×2 kernel.

we can observe that the shape of matrix H is 2×3. In general, if the shape of the input X is x_h×x_w (height×width) and the shape of the kernel is k_h×k_w, the shape of the hidden unit is:

$$h_h{\times}h_w = (x_h - k_h + 1){\times}(x_w - k_w + 1) \tag{3.2.3}$$

For this example, we have called the *feature map* the output of the convolution, it is to say, matrix H. On the other hand, we define the *receptive field* as all the elements (from all previous layers) that participate in an output during the forward phase. For example, for output H_{11}, the receptive field is $\{X_{1,1}, X_{1,2}, X_{2,1}, X_{2,2}\}$

3.2.1 Padding and stride

Assuming that the shape of the input X is $x_h \times x_w$ and the shape of the kernel is $k_h \times k_w$, the shape of the hidden unit is given by eq.(3.2.3). Padding and stride are techniques to better control the output of the convolution process.

Padding consists of adding zeros to the boundaries of the input X to increase the size in which the kernel can operate.

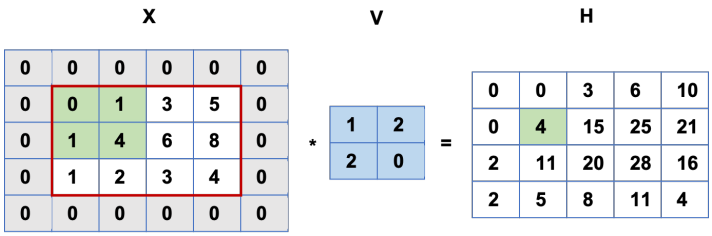


Figure 12: Example of the convolution operation with a 3×4 input, a 2×2 kernel, padding of 2×2.

Now, if the shape of the input X is $x_h \times x_w$, the shape of the kernel is $k_h \times k_w$, and we add p_h rows and p_w columns of padding, the shape of the hidden unit is:

$$h_h \times h_w = (x_h - k_h + p_h + 1) \times (x_w - k_w + p_w + 1) \tag{3.2.4}$$

We have slid the kernel one element at a time over the input. If we want to downsample, we can slide the window more than one element at a time. We define *stride* as the number of rows and columns traversed per slide. Now, if the shape of the input X is $x_h \times x_w$, the shape of the kernel is $k_h \times k_w$, we add p_h rows and p_w columns of padding, and the stride is of s_h positions in width and s_w positions in height, then, the shape of the hidden unit is:

$$h_h \times h_w = \lfloor \frac{(x_h - k_h + p_h + s_h)}{s_h} \rfloor \times \lfloor \frac{(x_w - k_w + p_w + s_w)}{s_w} \rfloor \tag{3.2.5}$$

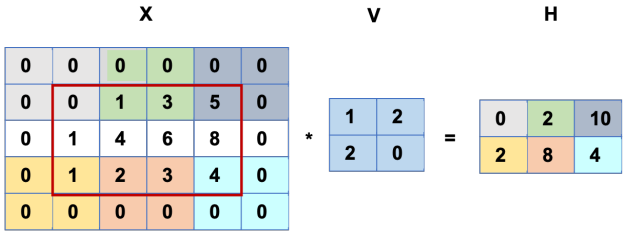


Figure 13: Example of the convolution operation with a 3×4 input, a 2×2 kernel, padding of 2×2, and a stride of 3×2.

Finally, there is also the possibility to apply an online activation function to each of the H outputs.

3.2.2 Channels

Images are expressed as the combination of colors, e.g., RGB (red, green and blue), meaning that each image is expressed as three matrices (or a 3D tensor) of values representing each color. In the same way, we can express the hidden layer as the superposition of several grids that perform convolutions.Each grid represents a channel, and the underlying idea is that channels allow you to search for different sets of features in the image. For example, eq.(3.2.1) can be expressed in case of having several channels as:

$$H_{i,j,d} = U_d + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} V_{a,b,c,d}\, X_{i+a,j+b,c} \tag{3.2.6}$$

where the output H is for channel d and depends on several input channels c.

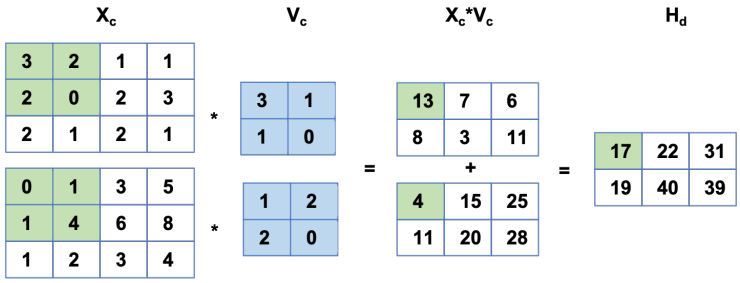


Figure 14: Example of the multi-input channel convolution operation with two 3×4 inputs and two 2×2 kernels.

We can extend the idea for multiple inputs and multiple outputs, so we use c_i

input channels to obtain c_o output channels, Figure 15. You have to note that the number of variables will be $c_i * c_o * k_h * k_w$ plus the biases.

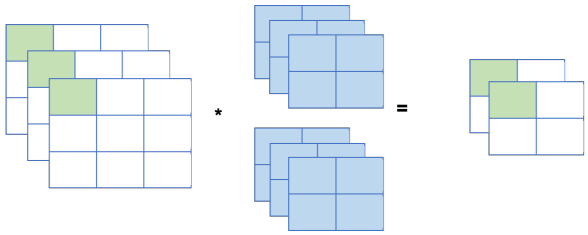


Figure 15: Example of the multi-input multi-output channel convolution operation.

3.2.3 Pooling

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input, which implies that small movements in the position of the feature in the input image will result in a different feature map. This can occur with cropping, rotating, shifting and other minor changes in the input image.To solve this problem, after performing the convolutional layer, it is necessary to perform down-sampling, which is the purpose of the pooling layer. The pooling layers are responsible for reducing the spatial dimensions of the input data, in terms of width and height, while retaining the most important information. The pooling layer is usually applied after a nonlinearity, e.g., ReLU, has been applied to the feature maps output by a convolutional layer, see Figure 17.

The pooling operation consists of a pooling window that slides over a region of the input as in convolutions. The difference is that pooling has no kernel, and performs simple operations such as averaging or obtaining the maximum over that region. For example, in figure 16, we have performed a 2×2 max pooling operation. The output at position H_{00} (in green) is the result of max(0,1,1,4)=4. Average pooling is the average of the values, in this case, it would produce (0+1+1+4)/4 =1.5. Sometimes, average pooling considers the sum of the values, multiplied by a weight plus a bias, and the result is passed through a sigmoid activation function, i.e., q=$\phi(W(0+1+1+4)+b)$, where the weight W and the bias b have to be trained. As in convolution, padding and stride operations are allowed.

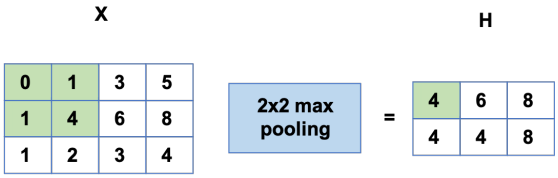


Figure 16: Example of a 2×2 max pooling operation.

Pooling helps to make the output *invariant* to small variations of the input, and it can be viewed as a strong prior that that the function the layer learns must be invariant to small translations. As a conclusion, the result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input.

3.2.4 Number of neurons and parameter sharing

Let us have a convolutional stage with a kernel of size $k_h * k_w$ and an output of size $h_h * h_w$. We then need $d * h_h * h_w$ neurons and $c * d * (k_h * k_w + 1) * (h_h * h_w)$ (including the bias or offset) connections between the inputs and outputs. The number of parameters or weights to be calculated in the training phase is $(c * d * (k_h * k_w + 1))$. The reason is that not all inputs are connected to all neurons, and only the receptive field of the neuron has to be connected. In addition, the network uses **shared parameters**, which means that the weights are shared among the neurons. In other words, the value of the weight applied to one input is tied to the value of a weight applied elsewhere.

In the case of a pooling layer, there are a single weight plus a bias for the kernel for each channel, thus only two parameters have to be trained. On the other hand, the number of connections are given by $c * d * (k_h * k_w + 1) * (h_h * h_w)$ (channels multiplied by neurons multiplied by kernel size plus the offset).

3.2.5 Integrating convolutions and pooling

A classical *convolution layer* in a convolution network, Figure 17, consist of several convolution in parallel followed by a non-linear activation function (e.g., a ReLU function). These acts as detectors of features. Finally, the result passes a pooling function to modify the output before going to the following convolution layer. Each convolution layer is in charge of detecting different sets of features.

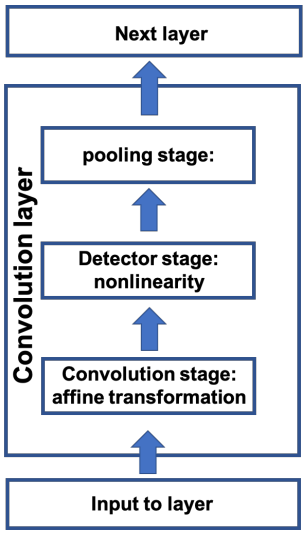


Figure 17: Convolution layer.

3.2.6 Example: LeNet-5 architecture

LeNet[1] was introduced by LeCun et al. in 1998 for handwriting recognition[2]. Although there are many new CNN architectures for image recognition and other applications, LeNet is a good exercise to learn how a CNN works. The network consists of 7 layers: 2 convolution layers (convolution + subsampling using an average pooling), two dense fully connected layers and a softmax classifier.

The convolutional layers extract low-level features such as edges, corners, and textures, while the pooling layers reduce the spatial dimensions of the feature maps. The fully connected layer then takes these abstracted features and combines them to make predictions about the digit shown in the image. For example, it might learn that a combination of curved lines, loops, and closed shapes indicates the presence of a particular digit. Then, the fully connected layer in a CNN is responsible for capturing global patterns and relationships in the input data by connecting every neuron from the previous layer to every neuron in the fully connected layer. It performs high-level reasoning and decision-making based on the learned features and contributes to regularization and model capacity control.

[1] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

[2] `https://www.kaggle.com/code/blurredmachine/lenet-architecture-a-complete-guide`

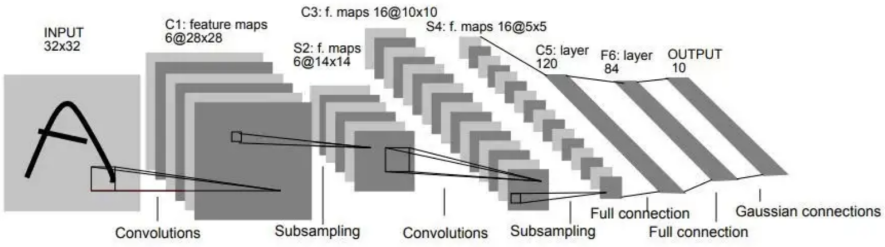


Figure 18: LeNet-5, original image published in LeCun et al. 1998 paper.

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

Figure 19: LeNet-5, table taken from kaggle lenet architecture (complete guide) footnote 2.

- The **input** (not included in the 7 layers) are standard MNIST images (32×32) greyscale images. The pixel values are normalized;
- **C1 layer-convolutional layer:** changes dimensions from 1 channel with (32×32) to 6 channels with (28×28). It consists of 6 (5×5) kernels, with 0 padding and 1 stride,
  - ❀ the **output feature map** is of size $\lfloor\frac{(32-5+0+1)}{1}\rfloor\times\lfloor\frac{(32-5+0+1)}{1}\rfloor = 28\times28$, eq.(3.2.5);
  - ❀ the **number of neurons** used is $6*28*28$=4.704;
  - ❀ the **number of learning parameters** is $6*(5*5+1)$=156 (note that we have added the bias to the filter weights);
  - ❀ the **number of connections** is $156*28*28$=122.304.
- **S2 layer-pooling layer (downsampling layer):** is a sub-sampling layer that uses 2×2 average pooling to produce 6 feature maps of 14×14. The input comes from the features maps of the C1 convolution layer, with a

stride of 2. The average pool result is obtained by adding the four filter values multiplied by a weight and a bias (W*(a+b+c+d)+bias). This sum is then passed through a sigmoidal activation function, that gives one of the 14 results of the pooled average filter.

- ❀ the **output feature map** is of size $(28-2+2)/2 \times (28-2+2)/2 = 14 \times 14$;
- ❀ the **number of neurons** used is $6 * 14 * 14 = 1.176$;
- ❀ the **number of learning parameters** is $6 * (1+1) = 12$ (the weight of the sums plus the bias);
- ❀ the **number of connections** is $6 * (2 * 2 + 1) * 14 * 14 = 5.880$.

- **C3 layer-convolutional layer:** applies 16 filters with 5×5 kernels, that changes dimensions from 14×14 to 10×10 resulting in a convolution layer with 16 feature maps. It uses a 1 stride and no padding. Remember that S2 produced 6 14×14 features maps as output. Now, each of the 16 filters at C3 takes some of these outputs as input (see Table 1). For example the first filter called 0 takes feature maps 0, 1 and 2 from S2 as inputs. In general, the first type (0-5) takes three consecutive feature maps from S2, the second type (6-11) takes four consecutive feature maps from S2, the third type (12-14) takes four discontinuous feature maps from S2, and the last typo (15) takes all features maps.

  - ❀ the **output feature map** is of size $(14-5+1)/1 \times (14-5+1)/1 = 10 \times 10$;
  - ❀ the **number of neurons** used is $16 * 10 * 10 = 1.600$;
  - ❀ the **number of learning parameters**: we have 3 types depending on how the input data (feature maps of S2 are combined, see Table 1. Then, we have [6* (5*5*3+1)] + [6* (5*5*4+1)] + [3* (5*5*4+1)] + [1* (5*5*6+1)] = 456 + 606 + 303 + 151 = 1516;
  - ❀ the **number of connections** is $1516 * 10 * 10 = 151.600$.

S2/C3	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Table 1: Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3.

- **S4 layer-pooling layer (downsampling layer):** again is a sub-sampling layer that uses 2×2 and a stride of 2 and a padding of 0. The 16 10×10

C3 convolutional filters are reduced to 16 5×5 after S4. average pooling to produce 6 feature maps of 14×14. The input comes from the features maps of the C3 convolution layer, with a stride of 2. The average pool result is obtained by adding the four filter values multiplied by a weight and a bias (W*(a+b+c+d)+bias). This sum is then passed through a sigmoidal activation function, that gives one of the 14 results of the pooled average filter.

- ❀ the **output feature map** is of size $(10-2+2)/2 \times (10-2+2)/2 = 5 \times 5$;
- ❀ the **number of neurons** used is $16 * 5 * 5 = 400$;
- ❀ the **number of learning parameters** is $16 * (1+1) = 32$ (the weight of the sums plus the bias);
- ❀ the **number of connections** is $(2 * 2 + 1) * 5 * 5 * 16 = 2.000$.

- **C5 is a fully connected convolutional layer** that has 120 5×5 filter, with 0 padding and 1 stride. Since the output of S4 has 400 neurons, then these 400 neurons are connected to these 120 neurons.
  - ❀ there are 120 **output feature maps** of size $(5-5+1)/1 \times (5-5+1)/1 = 1 \times 1$ each;
  - ❀ the **number of neurons** used is $1 * 1 * 120 = 120$;
  - ❀ the **number of learning parameters** is $(16 * 5 * 5 + 1) * 120 = 48.120$;
  - ❀ the **number of connections** is $48.120 * 1 * 1 = 48.120$.

- **C6 is a fully connected convolutional layer** that connects the 120 neurons of C5 to 84 neurons. The reason of having 84 neurons is that it corresponds to a 7×12 bitmap, where a -1 corresponds to *white* and 1 corresponds to *black*. This bitmap is thought for recognizing ASCII characters.
  - ❀ Input: 120 neurons;
  - ❀ Output: 84 neurons;
  - ❀ the **number of learning parameters** is $84 * (120 + 1) = 10164$.
  - ❀ the **number of connections** is $84 * (120 + 1) = 10164$.

- **Output layer** that connects the 84 neurons to 10 neurons. Each output is one of the 10 numbers 0,1,...,9 that appear in the images using a softmax activation output function.

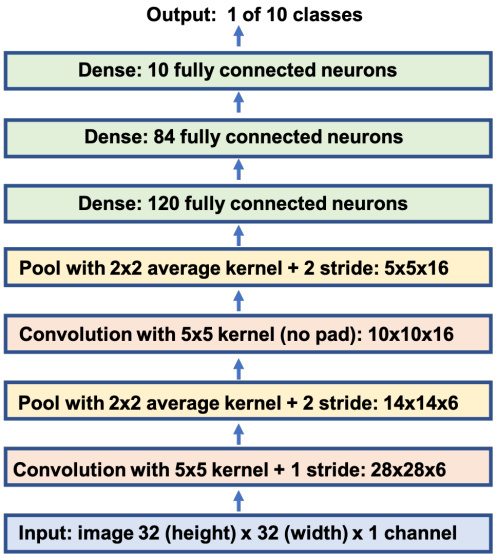


Figure 20: LeNet-5.

3.2.7 Other CNN: AlexNet architecture

AlexNet[3] (2012) uses an 8-layer CNN and takes images 227×227 color images, Figure 21. AlexNet is a deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. The neural network has 60 million parameters and 650.000 neurons, and it is one of the first reference CNN using GPU's to accelerate computation and also makes use of dropout as technique as regularization method.

[3]Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012).

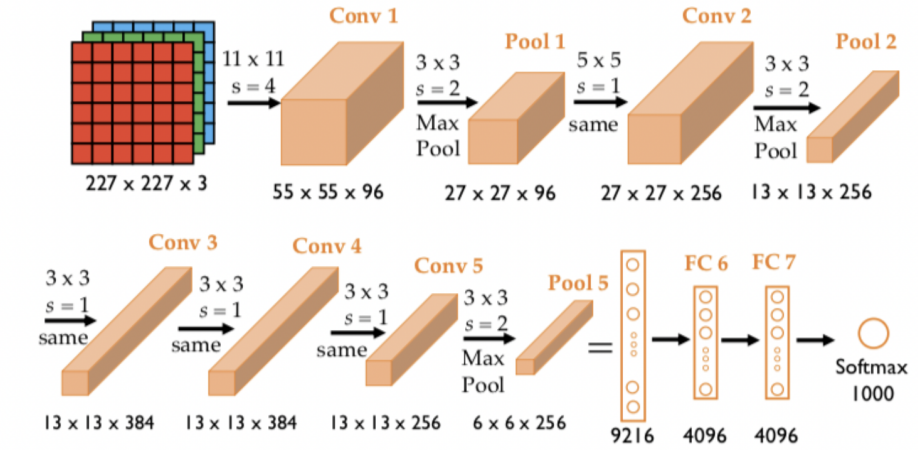


Figure 21: ALexNet, figure taken from Ahn H. Reynolds blog "Large-scale image recognition: AlexNet"(`https://anhreynolds.com/blogs/alexnet.html`).

- **1st convolutional layer:** 96 kernels of size 11×11×3, and a stride of 4, followed by a max pooling layer of size 3×3 and a stride of 2. Dimensions of the output are 96 55×55 feature maps after Conv 1 and 96 27×27 feature maps after Pool 1;
- **2nd convolutional layer:** 256 kernels of size 5×5, and a stride of 1, followed by a max pooling layer of size 3×3 and a stride of 2. Dimensions of the output are 256 27×27 feature maps after Conv 2 (p=2) and 256 13×13 feature maps after Pool 2;
- **3rd convolutional layer:** 384 kernels of size 3×3, and a stride of 1, no pooling. Dimensions of the output are 384 13×13 feature maps after Conv 3 (p=1) and 256 13×13 feature maps;
- **4th convolutional layer:** 384 kernels of size 3×3, and a stride of 1, no pooling. Dimensions of the output are 384 13×13 feature maps after Conv 4 (p=1) and 256 13×13 feature maps;
- **5th convolutional layer:** 256 kernels of size 3×3, and a stride of 1, followed by a max pooling layer of size 3×3 and a stride of 2. Dimensions of the output are 256 13×13 feature maps after Conv 5 (p=1) and 256 6×6 feature maps after Pool 5;
- **6th and 7th layers** are fully connected layers, each has 4096 neurons;
- **8th layer** is a 1000-way softmax.

	Activation shape	Activation size	# parameters
Input image	227 x 227 x 3	154587	0
Conv 1	55 x 55 x 96 (f=11 s = 4 p = 0)	290400	34944
Pool 1	27 x 27 x 96 (f=3 s = 2)	69984	0
Conv 2	27 x 27 x 256 (f=5 s = 1 p = 2)	186624	614,656
Pool 2	13 x 13 x 256 (f=3 s = 2)	43264	0
Conv 3	13 x 13 x 384 (f=3 s = 1 p = 1)	64896	885,120
Conv 4	13 x 13 x 384 (f=3 s = 1 p = 1)	64896	1,327,488
Conv 5	13 x 13 x 256 (f=3 s = 1 p = 1)	43264	884,992
Pool 5	6 x 6 x 256 (f=3 s = 2)	9216	0
FC 3	4096 x 1	4096	37,748,737
FC 4	4096 x 1	4096	16,777,217
Softmax	1000 x 1	1000	4096001

Figure 22: ALexNet activation shapes and sizes, and the number of parameters for each layer, figure taken from Ahn H. Reynolds blog "Large-scale image recognition: AlexNet"(`https://anhreynolds.com/blogs/alexnet.html`).

3.2.8 Other CNN

CNN architectures have evolved in the last years. For example, the **VGG (Visual Geometry Group) network**[4] (2014) use multiple convolutions, Figures 23 and 23[5] in between downsampling via max pooling in the form of a block (the so called VGG block). The objective is to see whether deeper or wider architectures perform better, e.g., two 3×3 convolutions touch the same number of pixels than a single 5×5 convolution:

$$h_h \times h_w = (x_h - k_{h1} + 1 - k_{h2} + 1) \times (x_w - k_{w1} + 1 - k_{w2} + 1) \tag{3.2.7}$$

$$= (x_h - k_h + 1) \times (x_w - k_w + 1) \tag{3.2.8}$$

[4]Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[5]Bacanin Dzakula, Nebojsa. "Convolutional neural network layers and architectures." Sinteza 2019-International Scientific Conference on Information Technology and Data Related Research. Singidunum University, 2019.

with k_h=$k_{h1} + k_{h2} - 1$ and k_w=$k_{w1} + k_{w2} - 1$.

Each VGG block uses a sequence of n 3×3 kernels (padding=1) for convolution followed by a 2×2 max pool layer with stride 2. Then, VGG blocks are concatenated, at the end dense fully connected neurons are used to obtain the output. Different VGG networks use different number of VGG blocks. The original paper introduces various VGG architectures, including VGG11, VGG13, VGG16, and VGG19, where the numbers represent the total layers in each architecture.

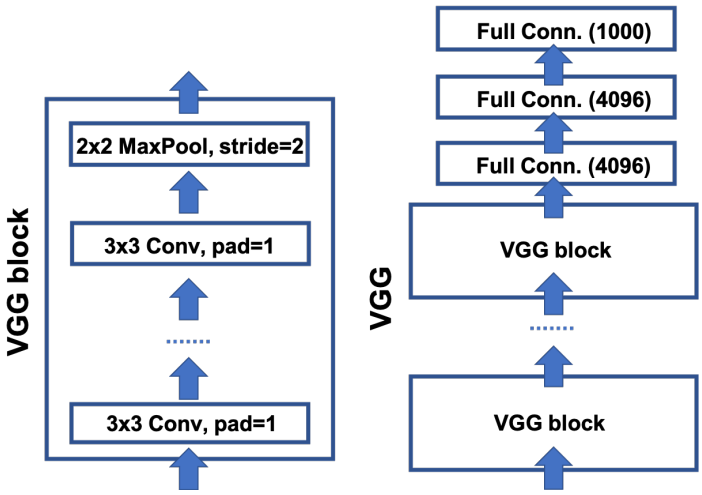


Figure 23: VGG architecture.

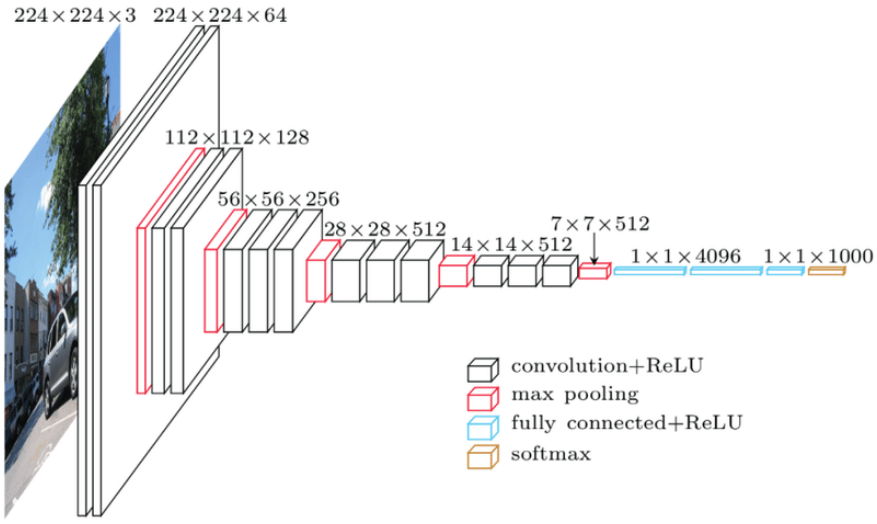


Figure 24: VGG architecture[5].

The drawbacks of LeNet, AlexNet or VGG are that at the end they use fully connected networks that require a lot of parameters (and memory), and it can not used fully connected networks at earlier stages because it would destroy the spatial structure (and they would require a lot of memory). For example, VGG-11 requires almost 400MB of RAM in single precision (FP32).

The **Network in Network (NiN)**[6] (2013) architecture solves these issues allowing 1×1 convolutions that add local nonlinearities across the channel activations, and act as a fully connected network without increasing the number of parameters since 1×1 convolutions use parameter sharing. Finally, at the last layer, the NiN architecture uses a global average pooling to integrate across all locations. Thus, a NiN block would be composed by a convolution layer (similar to the AlexNet) followed by two 1×1 convolutions, wherer 1×1 conv filters can be used to change the dimensionality in the filter space. For example if there are F filters, using F_1 1×1 conv filters (with $F_1<F$), we reduce dimensionality (less output filters) with the same h_h and h_w shape as the input. Again, if we want to increase dimensionality, we could use $F_1>F$.

[6]Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).

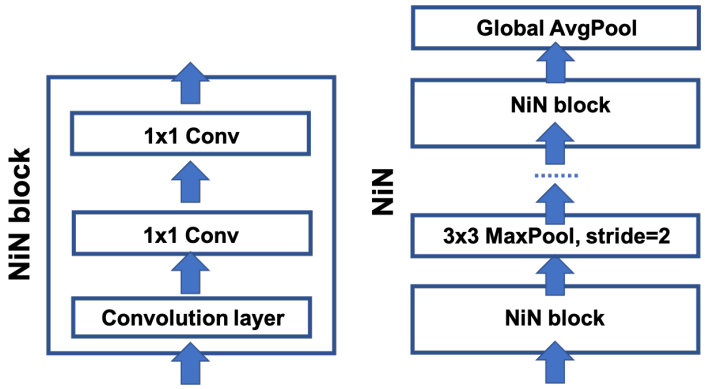


Figure 25: NiN architecture.

For example, a NiN network consisted of 1 NiN block with a 11×11 convolution (stride=4) + 3×3 MaxPool (stride=2), a second NiN block with a 5×5 convolution (pad=2) + 3×3 MaxPool (stride=2), a third NiN block with a 3×3 convolution (pad=1) + 3×3 MaxPool (stride=2), and a finally NiN block with a 3×3 convolution (pad=1) + the global average pooling layer.

The **GoogLeNet architecture**[7] (2015) is a multi-branch network that uses *inception blocks*, as main block architecture. Large convolutions require a large number of multiplications, while small convolutions are much faster and do not need much memory at the cost of not working so well. The idea of the inception block is that each unit from earlier layers corresponds to some region of the input image and these units are grouped into filter banks. In the lower layers (the ones close to the input) correlated units would concentrate in local regions. This means, we would end up with a lot of clusters concentrated in a single region and they can be covered by a layer of 1×1 convolutions in the next layer (see section 4 of the original paper[7] for further elaboration of these ideas). One big problem with the above modules is that even a modest number of 5×5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters. The solution is to apply dimension reductions and projections (1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions).

The inception block consists of four parallel branches, see Figure 26: three branches use convolutional layers os size 1×1, 1×1 (to reduce the number of input channels) plus a 3×3 and a 1×1 (to reduce the number of input channels) plus a 5×5, the fourth branch uses a 3×3 max pooling followed by a 1×1 convolutional layer. All these layers have the adequate padding if necessary. Finally, the four branches are concatenated along the channel dimension to produce the output. For memory efficiency during training, it seems beneficial to start us-

[7]Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

ing inception modules only at higher layers while keeping the lower layers in traditional convolutional fashion.

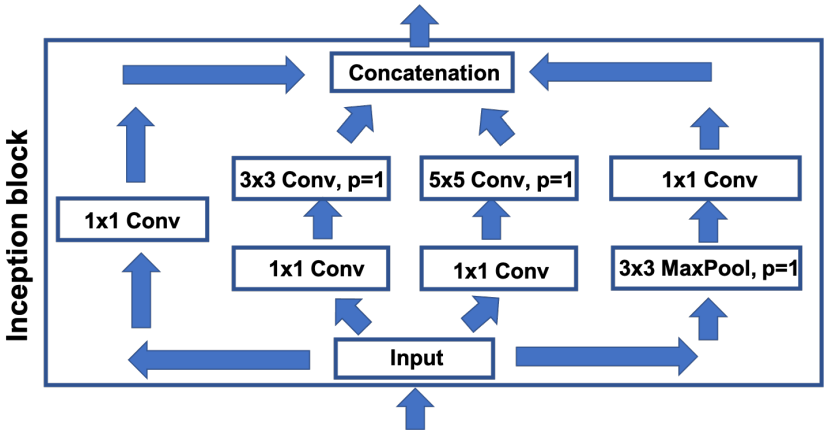


Figure 26: Inception architecture.

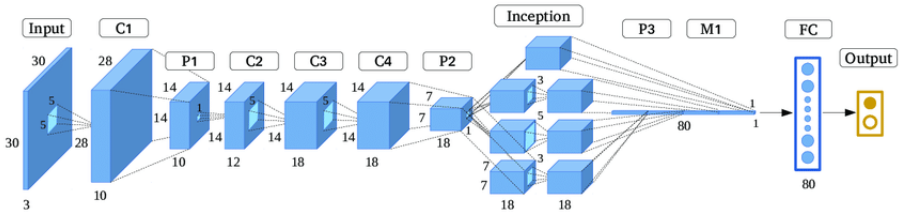


Figure 27: GoogleNet architecture.

A tuned hyperparameter is the number of output channels per layer. Now, GoogLeNet uses as *body* ((data processing)) 9 inception blocks, arranged in 3 groups with 3×3 max pooling to reduce dimensionality between the groups. Before the body, we can find the *steam* (data ingest) which consists of 3 convolutional layers each followed by a max pooling layer. The final output, *head* (prediction), is a global average pooling followed by a fully connected layer whose output is the number of labels.

There are many other advanced CNN's not covered in this course, such as Resnet (Residual Networks) and ResNeXt, DenseNet (2017), that preserve and reuse features from earlier layers, or AnyNet (2020) that help us to understand how to design CNN's.

4 AutoEncoders (AE)

An **autoencoder** is a neural network that is trained to attempt to copy its input to its output, and it can be also defined as a NN that learns data in an unsupervised manner. The main application of an autoencoder is to learn a lower-dimensional representation (encoding) for a higher-dimensional data, typically for dimensionality reduction. An autoencoder has 3 parts, Figure 28:

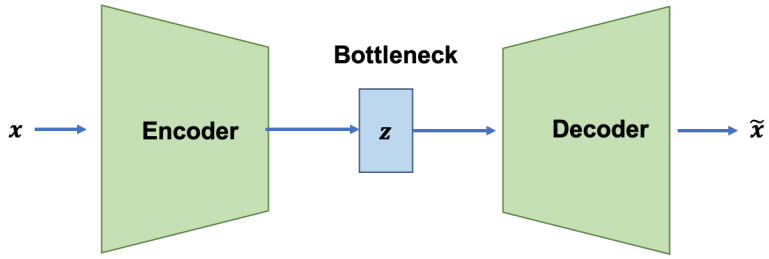


Figure 28: Autoencoder (AE) scheme.

- The **encoder** is a module that compresses the input data $\boldsymbol{x}$ (a vector) into an encoded representation that has less dimensions that the input data;
- The **bottleneck** is a module that contains the compressed knowledge representations $\boldsymbol{z}$ (a vector of lower dimension). The smaller the bottleneck, the lower the risk of overfitting, but the more information can slip through. The larger the dimension, the more the output resembles the input $\boldsymbol{x}$;
- The **decoder** is a module that decompresses the knowledge representations and reconstructs the data back from its encoded form. the goal of this module is to reconstruct the vector x from the latent space. Then, we can compare the reconstructed vector $\tilde{\boldsymbol{x}}$ with the correct vector $\boldsymbol{x}$.

We define the *latent space* as the space represented by vector z (in the bottleneck). Mathematically, we denote $\mathcal{X}$=$\mathbb{R}^m$ (input space) and $\mathcal{Z}$=$\mathbb{R}^n$ (latent space). Thus, f_ϕ:$\mathcal{X} \to \mathcal{Z}$ codes the input to the latent space with parameterization ϕ, and g_θ:$\mathcal{Z} \to \mathcal{X}$ codes (encoder) the latent space to the output space with parameterization θ. We define $\boldsymbol{z}$=$f_\phi(\boldsymbol{x})$ the latent variable or code, and $\tilde{\boldsymbol{x}}$=$g_\theta(\boldsymbol{z})$ decodes the latent variable to the estimated input. If the encoder is MLP (multi-layer perceptron), then:

$$f_\phi(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{4.0.1}$$

where σ is an activation function, e.g. a ReLu and the set of parameters ϕ to learn are $\boldsymbol{W}$ and $\boldsymbol{b}$. There are 4 characteristics that are important in an AE:

- the **code size** of the bottleneck decides how much the input is compressed and acts as a regularization term;

- the **number of layers** defines the depth to which the encoder and decoder go. Higher depths increase complexity, while lower depths favour the computational process;
- the **number nodes per layers** in the encoder/decoder defines the weights we use per layer. In general, the number of nodes decreases as we go towards the bottleneck;
- the **loss function** is highly dependent on the type of input and output we want the autoencoder to adapt to.

The first application of autoencoders are dimensionality reduction, PCA, information retrieval tasks (find entries in a database similar to the query) or anomaly detection among others. For example, in dimensionality reduction, the goal is for a given set of possible encoders and decoders, look for the pair that keeps the maximum of information when encoding and, so, has the minimum of reconstruction error when decoding. The idea of PCA is to build new independent features (the coded space) that are linear combinations of the input features and so that the projections of the data on the subspace defined by these new features are as close as possible to the initial data (in term of euclidean distance). In other words, PCA is looking for the best linear subspace of the initial space (described by an orthogonal basis of new features) such that the error of approximating the data by their projections on this subspace is as small as possible.

There are several types of autoencoders: i) undercomplete autoencoders, ii) regularized autoencoders (sparse, contractive and denoising autoencoders), and iii) variational autoencoders (for generative modelling).

4.1 Undercomplete autoencoders

An autoencoder whose code dimension is less than the input dimension is called *undercomplete autoencoder*. Learning an incomplete representation forces the autoencoder to capture the most relevant features of the training data.

Let us assume in Figure 28 that $\boldsymbol{z} = f(\boldsymbol{x})$ and $\tilde{\boldsymbol{x}} = g(\boldsymbol{z}) = g(f(\boldsymbol{x}))$. Then, the autoencoder minimizes a loss function $J = L(\boldsymbol{x}, \tilde{\boldsymbol{x}}) = L(\boldsymbol{x}, g(f(\boldsymbol{x})))$, meaning that the loss function penalizes $\tilde{\boldsymbol{x}}$ from being dissimilar from $\boldsymbol{x}$.

Autoencoders are considered an unsupervised learning technique since they don't need explicit labels to train on. But to be more precise they are self-supervised because they generate their own labels from the training data.

When the **decoder is linear** and the **loss function is the mean square error**, an undercomplete autoencoder learns to span the same subspace as PCA. This means that we form a lower-dimensional hyperplane to represent data in a higher-dimensional form without losing information. In this case, an autoencoder trained to perform the copy task has learned the main subspace of

the training data as a side effect. Taking the eigenfaces problem in which we performed the reconstruction as $\tilde{\mathbf{x}}_c = \mathbf{\Psi} + \mathbf{U}_k \mathbf{U}_k^\top \hat{\mathbf{x}}_c$, we can think that the autoencoder is performing the $\boldsymbol{U}_k^\top$ operation, and the decoder the reconstruction using $\boldsymbol{U}_k$.

Autoencoders with **nonlinear encoder** functions f and **nonlinear decoder** functions g can learn a more powerful nonlinear generalization of PCA. This form of nonlinear dimensionality reduction where the autoencoder learns a *nonlinear manifold* is also termed as **manifold learning**. A manifold is a topological space that locally resembles Euclidean space near each point, or we can think of a nonlinear manifold as referring to a manifold that is not an affine space. Nonlinear manifolds can be represented by their embedding, given by a vector in a low-dimensional "ambient" space. Among the several possibilities of learning the embedding, most consists of non-parametric methods based on the nearest-neighbor graph. This graph represents each training example with the tangent plane in the low-dimensional subspace that spans the directions of the variations associated with the diference vectors between the example data and its neighbors. This method works well if the manifold is smooth. If not, there can be variations not seen if there is not too many training data.

However, the autoencoder fails to learn useful information if they are given too much capacity (complexity). In other words, as the loss function has no explicit regularisation term, the only method to ensure that the model is not memorising the input data is by regulating the size of the bottleneck and the number of hidden layers within this part of the network. Remember that the capacity is the ability to fit a wide range of functions, thus increassing the capacity means that we can overfit the model. We can make the autoencoder more powerful by increasing the number of layers, the number of nodes per layer and, above all, the size of the code. Thus, increasing these hyperparameters will let the autoencoder to learn more complex codings. But we should be careful to not make it too powerful, because the autoencoder will simply learn to copy its inputs to the output.

4.2 Regularized autoencoders

Even in the case in which the code size is equal to the input or greater than the input (overcomplete case), the autoencoder can learn to copy the input to the output without learning anything useful. Then, the regularized autoencoder also has a regularization term to prevent the network from learning the identity function and mapping input into the output.

Regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output, and they can be non-linear and overcomplete. The properties that can learn include:

- smallness of the derivative of the representation (contractive autoencoders,

CAE),

- sparsity of the representation (sparse autoencoders, SAE), and
- robustness to noise or to missing inputs (denoising autoencoders, DAE).

The end result is to reduce the learned representation's sensitivity towards the training input.

4.2.1 Contractive autoencoders (CAE)

Contractive autoencoders[8] are a class of regularized autoencoders, in which a regularization term is added to the loss function. The main idea of contractive autoencoders is to make autoencoders robust to small perturbations (or disturbances) around the training points. This regularizer needs to conform to the expected squared Frobenius norm (sum of squared elements) of the Jacobian matrix for the encoder activation sequence, with respect to the input:

$$\min_{\theta,\phi} L(\boldsymbol{x}, \tilde{\boldsymbol{x}}, \phi, \theta) + \lambda L_{contractive}(\phi, \theta) \tag{4.2.1}$$

where $\lambda > 0$ measures how much contractiveness we want to enforce, θ and ϕ are the parameters of the encoder and decoder respectively. Then:

$$L_{contractive}(\phi, \theta) = E_{\boldsymbol{x}} \|\boldsymbol{J_f}(\boldsymbol{x})\|_F^2 = E_{\boldsymbol{x}} \|\nabla_{\boldsymbol{x}} f_\phi(\boldsymbol{x})\|_F^2 \tag{4.2.2}$$

is the expected Frobenius norm of the Jacobian matrix of f (autoencoder). So, small variations of $\boldsymbol{x}$ produce small variations in the code, which make that the extracted features resist infinitesimal input perturbations.

Because this penalty is applied only at training data, it forces the autoencoder to learn features that capture information about the training distribution. In other words, penalizing $\|\boldsymbol{J_f}(\boldsymbol{x})\|_F^2$ encourages the mapping to the feature space to be contractive in the neighborhood of the training data. As explained in paper[8], section 5.2, the robustness of the features can be seen as a contraction of the input space when projected in the feature space, in particular in the neighborhood of the examples from the data generating distributions. The Frobenius norm of the Jacobian at some point x measures the contraction of the mapping locally at that point. The directions that resist to this contracting pressure (strong invariance to input changes) are the directions present in the training set. As a conclusion, the penalty helps to carve a representation that better captures the local directions of variation dictated by the data (e.g. rotations or translations of objects), corresponding to a lower-dimensional non-linear manifold, while being more invariant to the vast majority of directions orthogonal (rare or small variations) to the manifold.

[8] Rifai, Salah, et al. "Contractive auto-encoders: Explicit invariance during feature extraction." Proceedings of the 28th international conference on international conference on machine learning. 2011.

Then, the goal of the CAE is to learn the manifold structure of the data. Directions of vector $\boldsymbol{x}$ with large $\boldsymbol{J_f}(\boldsymbol{x})$ rapidly change f, which represent the tangent planes of the manifold. Many singular values of $\boldsymbol{J_f}$ drop below 1 (so they are contractive and represent local variations in the data), and some remains close to 1 which represent the tangent directions that the autoencoder learns and that represent real variations in the data. Contractive autoencoders are better at feature extraction than denoising autoencoders.

4.2.2 Sparse autoencoders (SAE)

Sparse autoencoders are very similar to the undercomplete autoencoder but they are different in the way in which they regularize the input. While undercomplete autoencoders are regulated and fine-tuned by regulating the size of the bottleneck, the sparse autoencoder is regulated by changing the number of nodes at each hidden layer. The idea behind SAE is that we can achieve an information bottleneck (same information with fewer neurons) without reducing the number of neurons in the hidden layers. The number of neurons in the hidden layer can be greater than the number in the input layer. The loss function has a term that calculates the number of neurons that have been activated and provides a penalty that is directly proportional to that. This penalty, called the sparsity function, prevents the neural network from activating more neurons and serves as a regularizer. In other words, sparse autoencoders may include more hidden units than inputs, but only a small number of the hidden units are allowed to be active at the same time.

There are two ways of forcing sparsity:

- **K-sparse autoencoders**[9] set all but the highest k activations of the latent code to zero. Thus, $f_\theta(x_1, \ldots, x_n) = (b_1 x_1, \ldots, b_n x_n)$, where b_j=1 if $|x_j|$ is in the top k and b_j=0 otherwise. It is to say, in the feedforward phase, after computing the hidden code $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, rather than reconstructing the input from all of the hidden units, we identify the k largest (or the αk largest, with $\alpha \geq 1$) hidden units and set the others to zero. This selection step acts as a regularizer that prevents the use of an overly large number of hidden units when reconstructing the input.

- The other option is to add a **sparsity regularization loss**,

$$\min_{\theta,\phi} L(\boldsymbol{x}, \tilde{\boldsymbol{x}}, \phi, \theta) + \lambda L_{sparsity}(\phi, \theta) \tag{4.2.3}$$

  where $\lambda > 0$ measures how much sparsity we want to enforce. Let

$$\rho_k(x) = \frac{1}{n_k} \sum_{i=1}^{n_k} a_{k,i}(x) \tag{4.2.4}$$

[9] Makhzani, Alireza, and Brendan Frey. "K-sparse autoencoders." arXiv preprint arXiv:1312.5663 (2013).

be the actual sparsity of activation in each layer k, where $a_{k,i}(x)$ is the activation in the i-th neuron (there is n_k neurons at layer k) of the k-th layer upon input x. Then, the sparsity loss upon input x is given by a function s: $s(\rho_k(x), \tilde{\rho}_k)$, where $\tilde{\rho}_k$ is a desired sparisty. For example, function s can be the L-1 loss, i.e., $s(\rho, \tilde{\rho}) = |\rho - \tilde{\rho}|$, the L-2 loss i.e., $s(\rho, \tilde{\rho}) = |\rho - \tilde{\rho}|^2$, or the Kullback-Leibler (KL) divergence, i.e., $s(\rho, \tilde{\rho}) = \rho\ log[\rho/\tilde{\rho}] + (1-\rho)\ log[(1-\rho)/(1-\tilde{\rho})]$. Finally, the regularization term is expressed as the expection of the weighted function s:

$$L_{sparsity}(\phi, \theta) = \mathbb{E}_x \left[\sum_{k \in 1:K} w_k s(\hat{\rho}_k, \rho_k(x)) \right] \tag{4.2.5}$$

where w_k are the weigths at layer k. This form of regularization allows the network to have nodes in hidden layers dedicated to find specific features during training. Then, the regularisation problem is treated separately from the dimensionality of the latent space in the bottleneck.

4.2.3 Denoising autoencoders (DAE)

In **denoising autoencoders**, we try to minimize the reconstruction error term. In other words, the autoencoders seen try to reconstruct the input as the output. The goal is not only to eliminate noise, but to find a good representation of the data. The denoising autoencoder tries to reconstruct the output from corrupted or noisy input data. This noise is added randomly (e.g., additive white (isotropic) Gaussian noise, AWGN), only in the training phase, to the input data. Then, the noisy data is fed to the encoder-decoder architecture, and the output is compared with the ground truth data (undo the corruption rather than simply copying their input). The idea is that a corrupted input helps decrease the risk of overfitting and prevents the DAE from becoming an identity function. They can be useful in learning how to impute or fill in missing information during the reconstruction process.

If $\epsilon \sim N(0, \sigma^2)$, then let be $\hat{\boldsymbol{x}} = \boldsymbol{x} + \epsilon$ the corrupted version. The autoencoder, then, minimizes the loss function L($\boldsymbol{x}$, $g(f(\hat{\boldsymbol{x}}))$). The autoencoder performs the denoising by mapping the corrupted input data into a lower-dimensional manifold (like in undercomplete autoencoders), where filtering of noise becomes much easier than in the input space.

Contractive and denoising autoencoders are related. Contractive autoencoders make the feature extraction resist infinitesimal perturbations, while in the denoising autoencoder the reconstruction function resist small but finite-sized perturbations of the input.

Applications of DAE are image and audio denoising (cleaning and enhancing images by removing noise or denoise audio signals), processing sensor data, removing noise, and extracting relevant information from sensor readings, and

DAE are effective in unsupervised feature learning, capturing relevant features in the data without explicit labels.

4.3 Variational Autoencoders (VAE)

One of the problems in traditional AE is that the latent space is not continous. That means that there are "empty" space in the latent space, and where we try to reconstruct something from this "empty" space, the result is noise (or garbage). The solution are the variational autoencoder, in which distributions of the latent space are build, producing a continuos latent space.

A **variational autoencoder** is a generative model with a prior and noise distribution respectively. In other words, a VAE can be defined as being an autoencoder whose training is regularised to avoid overfitting and ensure that the latent space has good properties that enable generative process. Usually such models are trained using the Expectation-Maximization (EM) algorithm.

The difference with AE is that AE learns a compressed representation of the input data, and VAE learns a set of distribution parameters which describes the data, e.g., the mean (z_μ) and variance (z_σ) of a gaussian probability function. By sampling from these parameters, VAE can generate data instances that closely resembles the original data. The input data is generated with distribution p($\boldsymbol{X}$) which can be represented with latent variable $\boldsymbol{Z}$, generated with p($\boldsymbol{Z}$). Then, the joint distribution:

$$p(\boldsymbol{X}, \boldsymbol{Z}) = p_\theta(\boldsymbol{X}|\boldsymbol{Z})p_\theta(\boldsymbol{Z}) \tag{4.3.1}$$

where $p_\theta(\boldsymbol{X}|\boldsymbol{Z})$ is the likelihood and $p_\theta(\boldsymbol{Z})$ is the prior distribution in a Bayesian framework. Then, the process has two steps: i) sample $\boldsymbol{z}^i$ from the prior distribution $p_\theta(\boldsymbol{Z})$, and ii) sample $\boldsymbol{x}^i$ from the likelihood $p_\theta(\boldsymbol{X}|\boldsymbol{Z})$. The goal is to find:

- an efficient approximate ML/MAP for parameters $\boldsymbol{\theta}$;
- the posterior $p_\theta(\boldsymbol{Z}|\boldsymbol{X}) = p_\theta(\boldsymbol{X}|\boldsymbol{Z})p_\theta(\boldsymbol{Z})/p_\theta(\boldsymbol{X})$;
- an efficent approximate of variable $\boldsymbol{x}$.

The main challenge in this problem is the **intractability** of the true posterior $p_\theta(\boldsymbol{Z}|\boldsymbol{X})$, since for obtaining this posterior, we need the marginal (or **evidence**) $p_\theta(\boldsymbol{X})$ (integrating the likelihood multiplied by prior distribution). For the purpose of solving the above problems, let us introduce a recognition model $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ that is an approximation to the intractable true posterior $p_\theta(\boldsymbol{Z}|\boldsymbol{X})$. We will refer to the recognition model $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ (typically a NN) as a **probabilistic encoder**, since given a datapoint $\boldsymbol{x}$ it produces a distribution (e.g. a Gaussian of mean μ_z and variance σ_z^2) over the possible values of the code $\boldsymbol{z}$ from which the datapoint $\boldsymbol{x}$ could have been generated. In the same way, $p_\theta(\boldsymbol{X}|\boldsymbol{Z})$ is

a **probabilistic decoder**, since given a code $\boldsymbol{z}$ it produces a distribution over the possible corresponding values of $\boldsymbol{x}$.

Now, we are interested in *using a neural network* for the probabilistic encoder $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ (the approximation to the posterior of the generative model $p_\theta(\boldsymbol{X}, \boldsymbol{Z})$) and where the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are optimized jointly with the AEVB algorithm.

Let the prior over the latent variables be the centered isotropic multivariate Gaussian :

$$p_\phi(\boldsymbol{Z}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{0}, \boldsymbol{I}) \tag{4.3.2}$$

and, we will assume that the true posterior (intractable) posterior represented by $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ takes the form of a multivariate isotropic Gaussian distribution, and its log is:

$$log\ p_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)}) = log\ \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}_{\boldsymbol{z}}^{(i)}, \boldsymbol{\sigma}_{\boldsymbol{z}}^{\boldsymbol{2}(i)} \boldsymbol{I}) \tag{4.3.3}$$

where $\boldsymbol{\mu}_{\boldsymbol{z}}^{(i)}$ and $\boldsymbol{\sigma}_{\boldsymbol{z}}^{\boldsymbol{2}(i)}$ are the outputs of the encoder MLP, see Figure 29.

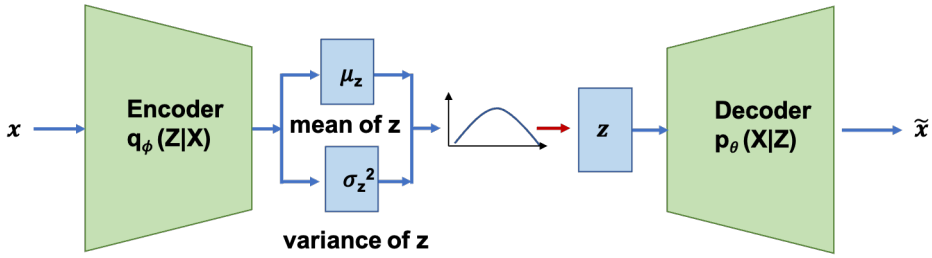


Figure 29: Variational autoencoder (VAE) scheme without using the reparametrization trick.

The idea is to sample $\boldsymbol{z}^{(i)}$ from the normal distribution given by eq.(4.3.3). The challenge in this figure is that we have a "random" node $\boldsymbol{z}$ in the bottleneck. We can observe how in Figure 29 the "red" arrow represents the sampling process. The problem is that we cannot backpropagate through a "random" node of the computation graph. The solution is to use what is called the *reparametrization trick* that consists of introducing a random (auxiliary) variable ϵ, of the same size than $\boldsymbol{z}$, multivariate Normal distributed $\mathcal{N}(\boldsymbol{\epsilon}; \boldsymbol{0}, \boldsymbol{I})$ from which we sample, see Figure 30, "red" arrow.

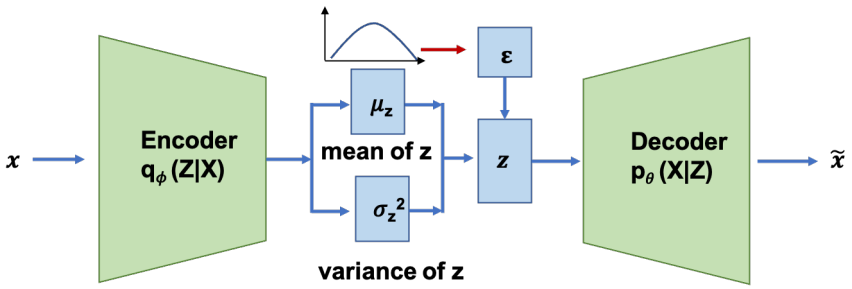


Figure 30: Variational autoencoder (VAE) scheme using the reparametrization trick.

Now, the computation graph is complete, meaning that the parameters go from decoder back to encoder, to be updated during backpropagation. Now, if we have to sample a $\boldsymbol{z}^{(i)}$ example, we will use $\boldsymbol{z}^{(il)} \sim g_\phi(\boldsymbol{x}^{(i)}, \boldsymbol{\epsilon}^{(l)}) = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(l)}$, where $\odot$ means element-wise product and $g(\cdot)$ is a differentiable function. For example if $\boldsymbol{\epsilon}$ is Gaussian distributed, then, $g(\cdot)$ can be Laplace, Elliptical, Student's t, Logistic, Uniform, Triangular and Gaussian distributions.

Finally, we have said that the encoder and the decoder are probabilistic MLP. For example, if we want a Gaussian MLP decoder, we can use the following:

$$\begin{aligned} log\ p(\boldsymbol{x}|\boldsymbol{z}) &= log\ \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2 \boldsymbol{I}) \\ \boldsymbol{\mu} &= \boldsymbol{W_1 h} + \boldsymbol{b_1} \\ log\ \boldsymbol{\sigma}^2 &= \boldsymbol{W_2 h} + \boldsymbol{b_2} \\ \boldsymbol{h} &= tanh(\boldsymbol{W_3 z} + \boldsymbol{b_3}) \end{aligned} \tag{4.3.4}$$

The reason of using $log\ \sigma^2$, is that the neural network, through the activation function, can produce negative values. In this way, if the $log\ \sigma^2$ is negative, we can recover the standard deviation σ using $e^{log\ \sigma^2/2} = e^{2log\ \sigma/2} = e^{log\ \sigma} = \sigma$, that always will be positive even if $log\ \sigma^2$ is negative. Finally, note that when this network is used as an encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, then $\boldsymbol{z}$ and $\boldsymbol{x}$ are swapped, and the weights and biases are variational parameters ϕ.

A final step is to make sure that the probabilistic encoder approximates the real distribution. To do this, we will have to add a term based on the Kullback-Liebler divergence to the loss function, so the cost function will be:

$$J = L(\boldsymbol{x}, \tilde{\boldsymbol{x}}) + D_{KL}[q_\phi(\boldsymbol{Z}|\boldsymbol{X}) \| p_\theta(\boldsymbol{X}|\boldsymbol{Z})] \tag{4.3.5}$$

The **Kullback-Liebler divergence ($D_{KL}[p\|q]$)** is defined as a measure of how one probability distribution p is different from a second, reference probability distribution q, thus $D_{KL}[p\|q]$ is telling us what is the dissimilarity of usign

distribution q instead of distribution p. We can express $D_{KL}[p\|q]$ as it is the expectation of the logarithmic difference between the probabilities p and q, where the expectation is taken using the probabilities p.

$$D_{\mathrm{KL}}(p \parallel q) = \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{p(x)}{q(x)} \right) = - \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{q(x)}{p(x)} \right) \tag{4.3.6}$$

the lower the KL divergence value, the better we have matched the true distribution with our approximation. In the context of machine learning, $D_{KL}[p\|q]$ is often called the information gain achieved if p would be used instead of q which is currently used.

We first obtain what is called the **autoencoder variational bound (AEVB)** algorithm. Remember that the marginal $\mathrm{p}(\boldsymbol{x}) = \mathrm{p}(x^{(1)}, \ldots, x^{(N)}) = \prod_i^N p(x^{(i)})$ and its $\log \mathrm{p}(\boldsymbol{x}) = \sum_i^N log\ p(x^{(i)})$, where we assume N data examples. This expression can be stated as a function of the **evidence lower bound (ELBO)** and the **Kullback-Liebler divergence ($D_{KL}[q\|p]$)**:

$$log\ p(\boldsymbol{x}) = D_{KL}[q_\phi(\boldsymbol{Z}|\boldsymbol{X}) \| p_\theta(\boldsymbol{X}|\boldsymbol{Z})] + ELBO \tag{4.3.7}$$

where $D_{KL}[q\|p]$ describes the agreement between distribution q and a reference distribution p. The $D_{KL}[q\|p]$ is zero if p and q are the same distribution and positive if they are not identical. It can be proof that the maximum of $\log \mathrm{p}(\boldsymbol{x})$ using the ELBO (AEVB algorithm) can be obtained by finding the parameters ϕ and θ that minimizes the ELBO (derivatives with respect ϕ and θ) equal to zero). The ELBO is given by:

$$ELBO = E_{q_\phi(\boldsymbol{Z}|\boldsymbol{X})} log\ p_\theta(\boldsymbol{X}|\boldsymbol{Z}) - D_{KL}[q_\phi(\boldsymbol{Z}|\boldsymbol{X}) \| p_\theta(\boldsymbol{Z})] \tag{4.3.8}$$

The first term is the expectation of the log likelihood of the decoder network given the encoder network's output. The second term is the KL divergence between the posterior distribution $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ (the encoder network) and the prior distribution $p_\theta(\boldsymbol{Z})$. Mininizing this expression brings the posterior $q_\phi(\boldsymbol{Z}|\boldsymbol{X})$ close to the prior $p_\theta(\boldsymbol{Z})$. Obtaining the gradient of the ELBO is difficult and ussually the Stochastic Gradient Variational Bayes (SGVB) estimator is used as a practical estimator of the lower bound and its derivatives w.r.t. the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$.

5 Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are a family of neural networks for processing sequential data. Let us assume data ($\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$, $\ldots$, $\boldsymbol{x}^{(T)}$), where each $\boldsymbol{x}^{(t)}$ is a real value vector. The reference or target data is expressed as ($\boldsymbol{y}^{(1)}$, $\boldsymbol{y}^{(2)}$, $\ldots$, $\boldsymbol{y}^{(T)}$). Thus, a training set is a sequence of T examples. Sequences can be finite or countably infinite length, and sequences can also be non-temporal. However, we are going to focus in temporal for simplicity, where the supscript (t) indicates the time step.

5.1 The recurrent neural networks (RNN) model

Let us take a simple example for simplicity, with one input node, one output node, and a single hidden layer with one node, see Figure 31.top).

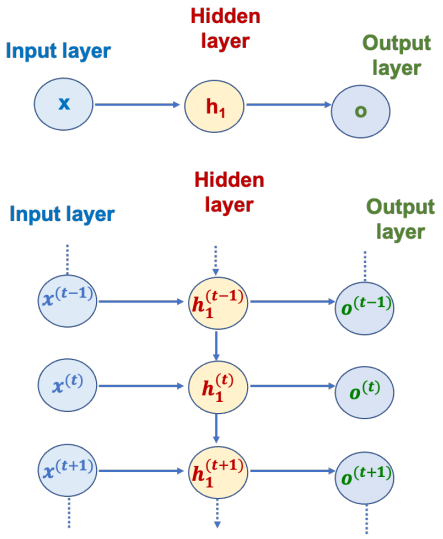


Figure 31: a) Top: a FNN with 1 one input node, 1 hidden layer node and 1 output node; b) below: same model using a recurrent neural networks (RNN) architecture.

Edges that connect adjacent time steps, called recurrent edges, may form cycles, including cycles of length one that are self-connections from a node to itself across time. Taking the example of Figure 31.below), the node in the hidden layer $\boldsymbol{h}^{(t)}$ in the RNN model takes at time (t) as input the current data point $\boldsymbol{x}^{(t)}$ and the output of the previous hidden layer node $\boldsymbol{h}^{(t-1)}$. Figure 32 shows the same example with two hidden layer nodes.

We can observe in Figure 32.left) the folded diagram of the RNN, in which the recurrence is expressed as a function f over the hidden layer. The unfolded graph provides an explicit description of which computations to perform, Figure 32.right), where we can observe the weights between the hidden layer at step $(t-1)$ and step (t). We can represent the unfolded recurrence at time t as:

$$\begin{aligned} \boldsymbol{h}^{(t)} &= g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \\ &= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}, \boldsymbol{W}) \end{aligned} \tag{5.1.1}$$

where the function $g^{(t)}(\cdot)$ takes the whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state. The unfolded recurrent structure allow us to factorize $g^{(t)}$ into repeated application of function f.

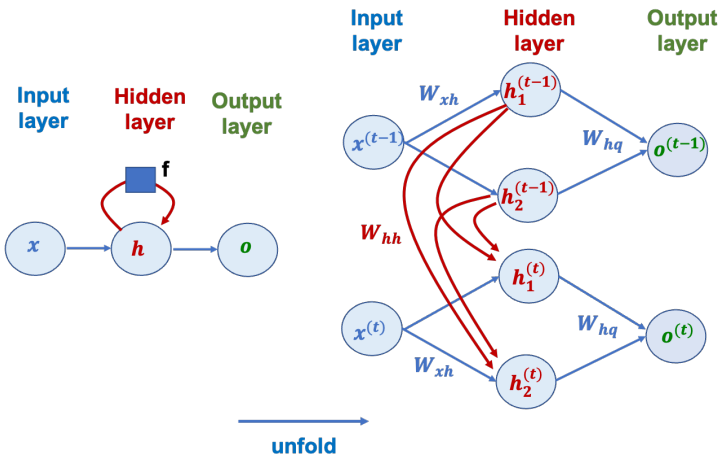


Figure 32: Example of a recurrent neural networks (RNN) using 1 one input node, 1 hidden layer node and 1 output node; a) left: circuit diagram, b) right: the same network seen as an unfolded computational graph.

We can then observe as the input $\boldsymbol{x}^{(t-1)}$ at time $(t-1)$ can influence the output $\boldsymbol{o}^{(t)}$ at time (t) and later by the recurrent connections. We can now express the hidden layer output of the current time step (t) as:

$$\begin{aligned} \boldsymbol{H}^{(t)} &= f(\boldsymbol{X}^{(t)}\boldsymbol{W}_{xh} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}_{hh} + \boldsymbol{b}_h) \\ \boldsymbol{O}^{(t)} &= \boldsymbol{H}^{(t)}\boldsymbol{W}_{hq} + \boldsymbol{b}_q \\ \hat{\boldsymbol{Y}}^{(t)} &= softmax(\boldsymbol{O}^{(t)}) \end{aligned} \tag{5.1.2}$$

Parameters of the RNN include the weights $\boldsymbol{W}_{xh} \in \mathbb{R}^{p \times h}$, $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ and bias $\boldsymbol{b}^h \in \mathbb{R}^h$, while the output has as parameters the weights $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$ and bias $\boldsymbol{b}^q$. Note that even at different time steps, RNNs always use these model parameters, meaning that the parameterization cost of an RNN does not grow as the number of time steps increases.

5.2 Backpropagation through time (BPTT)

Backpropagation through time (BPTT) is the version of the backpropagation for RNN. The idea is to apply the backpropagation to the unrolled computational graph.

5.3 Vanishing and exploding gradient problem

The main challenge in RNN training is that gradients vanish and explode when errors are backpropagated over many time steps. Consider the example of Figure 31.below) with one recurrent hidden layer node. Let us assume the input introduced to the network at time τ and the error at time t and let us assume inputs of zero in the intermediate time steps, then, the recurrent edge at the hidden node always has the same weight. Therefore, the contribution of the input at time τ to the output at time t will either explode or approach zero, exponentially fast, as $t - \tau$ grows large. Hence the derivative of the error with respect to the input will either explode or vanish.

This effect can be understood in terms of the long-term dependencies that appear when the computational graph becomes extremely deep, as in RNNs, where we build very deep computational graphs by repeatedly applying the same operation at each time step of a long time sequence.

Suppose a path that consists of multiplying repeatedly the same matrix $\boldsymbol{W}$ (as the RNN does). After t steps, this is equivalent to multiplying by $\boldsymbol{W}^t$. The reason is that we can think that the recurrent relation, assuming a very simple RNN with no inputs $\boldsymbol{x}$ and no activation units, performs as:

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^\top \boldsymbol{h}^{(t-1)} = (\boldsymbol{W}^t)^\top \boldsymbol{h}^{(0)} \tag{5.3.1}$$

If we perform an eigen-decomposition of matrix $\boldsymbol{W}^t$, we will obtain:

$$\boldsymbol{W}^t = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)^t = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^\top \tag{5.3.2}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix with the eigenvalues, and then $\boldsymbol{\Lambda}^t$ is a diagonal matrix with the eigenvalues to the power of t, and:

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q}\, \boldsymbol{h}^{(0)} \tag{5.3.3}$$

Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. This is called the **vanishing and exploding gradient problem**, meaning that the gradients of the weights also scale with the value of $\boldsymbol{\Lambda}^t$, and is a problem particular to RNN. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

In general, feedforward neural networks (FNN) does not use the same weighted matrix $\boldsymbol{W}$ even if they are deep, so they can avoid the vanishing and exploding gradient problem. However, RNN use the same matrix $\boldsymbol{W}$ for the recurrence (temporal sequence), and then they suffer the vanishing and exploding gradient problem.

There have been several solutions to the vanishing and exploding gradient problem such as *echo state networks*, *liquid state machines*, *leaky units* or *removing connections*. The most succesful mechanisms is the introduction of *gates* in which is called the long short-term memory (LSTM) mechanism.

5.4 Adding skip connections

One way to obtain coarse time scales is to add direct (skip) connections from variables in the distant past to variables in the present. Assuming that gradients in a classical RNN gradients vanish or explode exponentially w.r.t the number of time steps, e.g., τ, adding a skip connection with a delay of d will make the gradient decrease as a function of τ/d, although they can explode as a function of τ.

5.5 Leaky units

Leaky units are hidden units with self-connections that behave as moving averages. A moving average is a calculation to analyze data points by creating a series of averages of different subsets of the full data set. A moving average is commonly used with time series data to smooth out short-term fluctuations and highlight longer-term trends or cycles. In our case, the idea is to perform the operation: $a^{(t)} \leftarrow \alpha\, a^{(t-1)} + (1-\alpha)\, b^{(t)}$, where a is the average of some value b of a system. Whenever α is closed to one the moving average remembers information about the past for a long time, and when is close to zero, information about the past is rapidly discarded.

5.6 Removing connections

Another way to handle long-term dependencies is to remove recurring connections of length 1 and replace them with longer recurring connections, forcing them to work on a longer time scale. The difference with skip connections lies in the fact that some 1-hop connections are *removed* and replaced by longer connections, while skip connections consist of *adding* long connections.

5.7 Long short-term memory (LSTM)

LSTM model primarily in order to overcome the problem of vanishing gradients. LSTM are RNN that substitute hidden layer nodes by memory cells. The idea behind memory cells is as follows; RNNs have *long-term memory* through slowly changing weights during training. On the other hand, they have *short-term memory* through activation functions. The LSTM provides an intermediate type of storage through memory cells. A **memory cell** contains a set of units called gates and operations on the inputs and outputs of these units. The elements of a memory cell are, Figure 33:

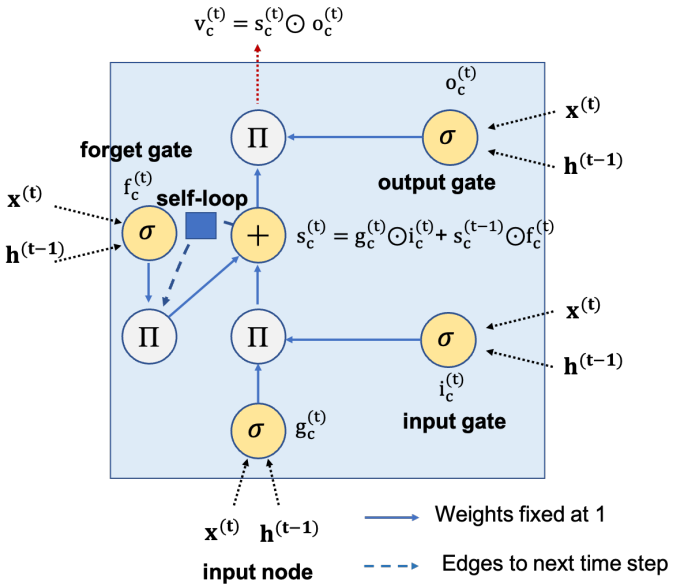


Figure 33: LSTM unit with input node ($g_c^{(t)}$), input gate ($i_c^{(t)}$), output gate ($o_c^{(t)}$), forget gate ($f_c^{(t)}$) and internal state ($s_c^{(t)}$).

A **gate** is a sigmoidal unit that, like the input node, takes activation from the current data point $\boldsymbol{x}^{(t)}$ as well as from the hidden layer at the previous time step $\boldsymbol{h}^{(t-1)}$. A gate is so-called because its value is used to multiply the value of another node. It is a gate in the sense that if its value is zero, then flow from the other node is cut off. If the value of the gate is one, all flow is passed through. Then, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity.

- **Input node (g_c):** is a node that takes activation in the standard way from the input layer $\boldsymbol{x}^{(t)}$ at the current time step (t) and (along recurrent edges) from the hidden layer at the previous time step $\boldsymbol{h}^{(t-1)}$. Typically, the summed weighted input is run through a tanh activation function, although in the original LSTM paper, the activation function is a sigmoid;
- **Input gate (i_c):** the value of the input gate i_c multiplies the value of the input node;
- **Internal state (s_c):** the memory cell has a linear activation. It has a self-looped connection with fixed unit weight that multiplies the forget gate. This edge is often called the *constant error carousel*. In vector notation, the update for the internal state is $s_c^{(t)} = g_c^{(t)} \odot s_c^{(t)} + s_c^{(t-1)} \odot f_c^{(t)}$ where $\odot$ is pointwise multiplication;

- **Forget gate (f_c):** they provide a method by which the network can learn to flush the contents of the internal state. A leaky unit was able to remember or accumulate information during a long period and then reset (forget) this information. The idea of the forget gate is that instead of reseting the long-term information by hand (leaky unit), let the NN to do ths reset using the forget gate;
- **Output gate (o_c):** the value v_c ultimately produced by a memory cell is the value of the internal state s_c multiplied by the value of the output gate o_c.

The propagation equations are as follows:

$$\begin{bmatrix} i_c \\ f_c \\ o_c \\ g_c \end{bmatrix} = \begin{bmatrix} \sigma(\boldsymbol{b_i} + \boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)}) \\ \sigma(\boldsymbol{b_f} + \boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)}) \\ \sigma(\boldsymbol{b_o} + \boldsymbol{W}_{xo}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)}) \\ \tanh(\boldsymbol{b_g} + \boldsymbol{W}_{xg}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hg}\boldsymbol{h}^{(t-1)}) \end{bmatrix} \tag{5.7.1}$$

Recall that the sigmoid σ obtains a gated value between 0 and 1. The output to the next layer and to the recurrent hidden node is given by:

$$h^{(t)} = tanh(s_c^{(t)}) \odot o_c^{(t)} \tag{5.7.2}$$

where the connection $\boldsymbol{h}^{(t)}$ can also be shut off using the output gate $o_c^{(t)}$.

5.8 Bi-LSTM

Bidirectional-LSTM are putting two RNN[10] (LSTM based) in a single architecture, figure 34. The idea is that for some applications it is enough to look back, but others can benefit from having some context about the future. Some RNN have tried the use of future information as context for current prediction have been attempted in the basic architecture of RNNs by delaying the output by a certain number of time frames, which means selecting a pre-defined amount of dealy. Bi-directional RNN's considers all available input sequence in both the past and future for estimation of the output vector. The input flows in both directions capturing the patterns and dependencies within the input data and updating their hidden states in each direction. Denoting the Bi-LSTM inputs as $[\ldots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \ldots]$, the hidden states of the forward and backward step, $h_{\mathbf{t}}^{\mathbf{F}}$ and $h_{\mathbf{t}}^{\mathbf{B}}$ are given as:

$$h_t^F = LSTM^F(x_t, h_{t-1}^F) \tag{5.8.1}$$

$$h_t^B = LSTM^B(x_t, h_{t+1}^B) \tag{5.8.2}$$

[10] Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE transactions on Signal Processing, 45(11), 2673-2681.

The output is y_t is given by:

$$y_t = \sigma(W^F h_t^F + W^B h_t^B + b_t) \tag{5.8.3}$$

with σ an activation function.

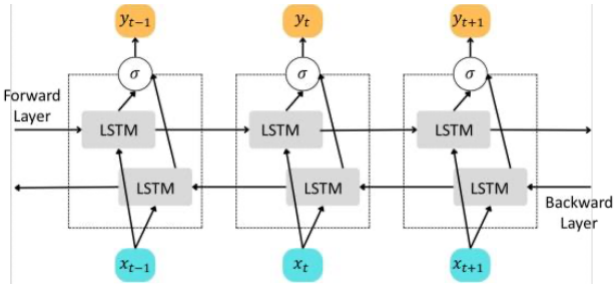


Figure 34: Bi-LSTM network.

6 Transformers and attention mechanisms

Attention mechanisms are designed to capture the contextual information from the input sequence. The attention mechanism proposed by Bahdanau[11] was applied to neural machine translation that attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation. However, attention mechanisms are now applied to many applications. Bahdanau attention was proposed to address the performance bottleneck of conventional encoder-decoder architectures, and overcomes the problem of using RNN encoder-decoder frameworks for neural machine translation, which encoded a variable-length source sentence into a fixed-length vector. Transformers (2017)[12] is an architecture that uses blocks of attention mechanisms, among other mechanisms.

6.1 Bahdanau attention mechanism

In these problems, we assume a RNN AE architecture[13], in which the input encoder data is $x_1, \ldots, x_T$, the output of the encoder is a vector c in the latent space, and the output of the decoder is $y_1, \ldots, y_T$, figure 35. We can observe

[11] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

[12] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[13] Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." arXiv preprint arXiv:1406.1078 (2014).

that:

$$h_t = f(x_t, h_{t-1}) \tag{6.1.1}$$

where f can be an LSTM, and

$$c = q(h_1, \ldots, h_T) \tag{6.1.2}$$

where q is a non-linear function. The decoder obtains an output in which:

$$y_t = g(y_{t-1}, s_t, c) \tag{6.1.3}$$

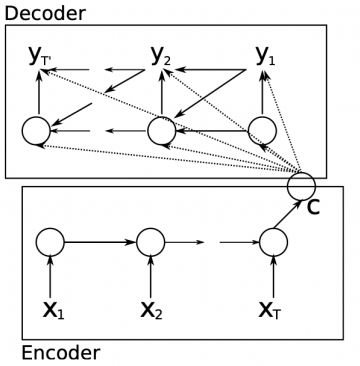


Figure 35: RNN-AE network, picture taken from[13].

with g a non linear function, s_t the output of the hidden state of the decoder, and c the output of the latent space. In this model, the decoder defines a probability over the translation y by decomposing the joint probability into the ordered conditionals:

$$p(\boldsymbol{y}) = \prod_{t=1}^{T} p\{y_t \mid y_{t-1}, \ldots, y_1, c) = \prod_{t=1}^{T} g(y_{t-1}, s_t, c) \tag{6.1.4}$$

The idea behind Bahdanau's attention mechanism[11] is that the translator must take into account not only the past, but also the future, and that there will be some words that bring more information to the translation, so the model must pay more attention to them than to others. Thus, if we define a Bi-RNN (e.g. a Bi-LSTM), figure 36 and defining:

$$p\{y_t \mid y_{t-1}, \ldots, y_1, \boldsymbol{x}) = g(y_{t-1}, s_t, c_t) \tag{6.1.5}$$

with s_i the hidden state of the RNN at time t, defined by

$$s_t = f(s_{t-1}, y_{t-1}, c_t) \tag{6.1.6}$$

where now, c_i is defined differently:

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj} h_j \tag{6.1.7}$$

where the context vector c_t depends on a sequence of annotations (h_1 , ... , h_{Tx}) to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the t-th word of the input sequence. The weights α_{tj} are computed as:

$$\alpha_{tj} = \frac{exp(e_{tj})}{\sum_{k=1}^{T_x} exp(e_{tk})} \tag{6.1.8}$$

$$e_{tj} = a(s_{t-1}, h_j) \tag{6.1.9}$$

where e_{tj} is an alignment model which scores how well the inputs around position j and the output at position i match. The probability α_{tj}, or its associated energy e_{tj}, reflects the importance of the annotation h_j with respect to the previous hidden state s_{t-1} in deciding the next state s_t and generating y_t. Intuitively, this implements a mechanism of attention in the decoder. The decoder decides parts of the source sentence to pay attention to.

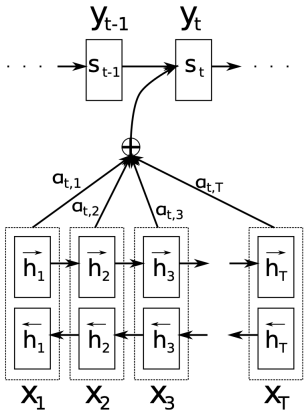


Figure 36: Bahdanau attention mechanisms, picture taken from [11]. The input is a source text (words) and the model tries to generate the t-th target word y_t.

6.2 Transformers

Transformers follow an architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, and was proposed for transduction models[12], see figure 37. The input and output are related to each

other through the context vector (the autoencoder transforms the input into a related output). An example of a transformer is translating a sentence in one language into the same sentence in another language. Another example of a transformer is image captioning, where the image is transformed into a caption that explains the content of the image.

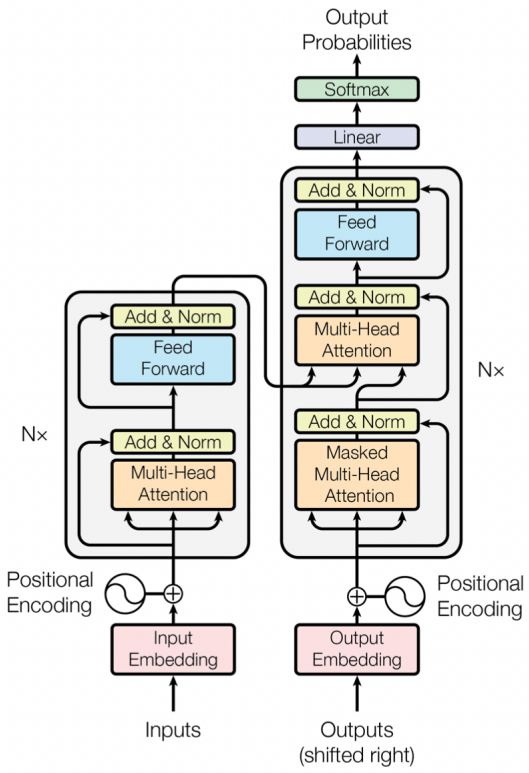


Figure 37: Transformer architecture, picture taken from [12].

The original transformer architecture is composed by 6 layers in the encoder, with two sublayers at each layer. The first sublayer is a multihead attention mechanism, and the second a fully connected NN.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

We can consider two kind of attention functions, figure 38: a scaled dot-product attention function, figure 38.left), and a multi-head attention function, figure

38.right).

In the scaled dot-product attention function, the outputs are aggregates of these interactions and attention scores using a Query–Key–Value (QKV) model. $\boldsymbol{Q}$ refers to the query vectors matrix, q_i being a single query vector associated with a single input word. $\boldsymbol{V}$ refers to the values vectors matrix, v_i being a single value vector associated with a single input word. $\boldsymbol{K}$ refers to the keys vectors matrix, k_i being a single key vector associated with a single input word. The output of the scaled dot-product attention function is:

$$Att(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{6.2.1}$$

where queries and keys have dimensions $\boldsymbol{Q}$,$\boldsymbol{K}\in\mathbb{R}^{d_k}$, and values with dimensions $\boldsymbol{V}\in\mathbb{R}^{d_v}$.

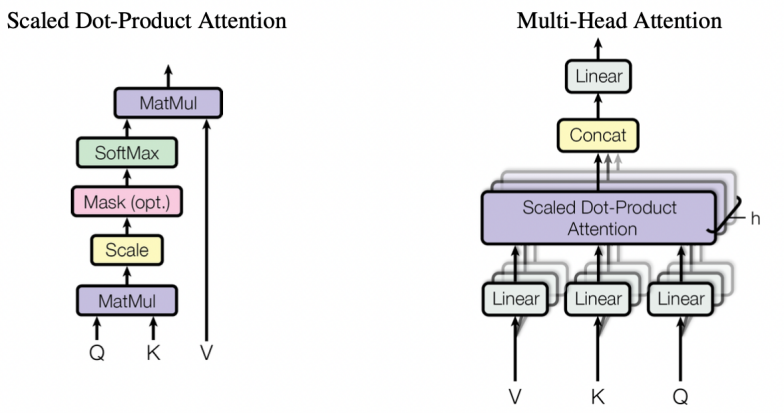


Figure 38: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel, picture taken from [12].

We can observe in equation (6.2.1), that the output of the attention are the values multiplied by weights, where the weights are a softmax function of dot products of keys and queries scaled by the square root of the dimension d_k. these matrices are the result of a matrix product between the input embeddings and 3 matrices of trained weights: $\boldsymbol{W}_q$, $\boldsymbol{W}_k$ and $\boldsymbol{W}_v$. These matrices are randomly initialised, and learned during training. The dot product between $\mathbf{Q}\mathbf{K}^\top$ is used to compute a sort of similarity score between the query and key vectors, as it is done when we search in databases. Given a query, you want to retrieve the closest sentence in meaning among all possible answers, and this is done by computing the similarity between sentences (question vs possible answers). In other words, closer query and key vectors will have higher dot products that will be normalised and applying the softmax between 0 and 1. Finally, multiplying

the softmax results (weights) to the value vectors will push down close to zero all value vectors for words that had a low dot product score between query and key vectors.

In the multi-head attention instead of performing a single attention function with n-dimensional keys, values and queries, the keys, values and queries are l times linearly projected with different dimensions. Each different projection is performed in parallel, yielding a d_v dimensional output. The idea is that by applying the attention mechanism multiple times in parallel (these parallel applications are the "heads"), the model can capture different types of relationships in the data simultaneously, e.g., relations between one-to-many words or many-to-many words of a sentence.

$$MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Concat(head_1, \dots, head_l)\mathbf{W^O} \tag{6.2.2}$$

where $head_i = Att(\mathbf{QW_i^Q}, \mathbf{KW_i^K}, \mathbf{VW_i^V})$, and the projections are parameter matrices $\mathbf{W_i^Q}\in\mathbb{R}^{\mathrm{n}\times\mathrm{d_q}}$, $\mathbf{W_i^K}\in\mathbb{R}^{\mathrm{n}\times\mathrm{d_k}}$ and $\mathbf{W_i^V}\in\mathbb{R}^{\mathrm{n}\times\mathrm{d_v}}$.

7 Numerical computations

7.1 Cost functions and stochastic gradient descent

To estimate the parameters $\boldsymbol{\beta}$, we define a *training set* $\mathbf{T}=\{y_n, \boldsymbol{x_n}\}_{n=1,\dots,N}$, and a *loss function* $L(\boldsymbol{\beta}) = l(y, \boldsymbol{x}, \boldsymbol{\beta})$ for every element of the training set.

7.1.1 Empirical risk

Recall that in supervised machine learning, we assume joint random variables (X, Y) of unknown joint distribution. Our goal is to find a function $f_{\boldsymbol{\beta}}(x) = \hat{y}$ that, for a pair (x, y) which is a realization of the random variables (X, Y), gives an estimate $\hat{y}$ of the value y given that we know x, i.e. $\hat{y} = f_{\boldsymbol{\beta}}(x)$. This estimate must be such that that for a loss function $L(\boldsymbol{\beta})$, we minimize the *risk* ($R_{f_{\boldsymbol{\beta}}}$) defined as:

$$R_{f_{\boldsymbol{\beta}}} = \mathbb{E}_{(X,Y)}[L(\boldsymbol{\beta})] \tag{7.1.1}$$

As we do not have the joint distribution of (X, Y), we minimize the *empirical risk* $\hat{R}_{f_{\boldsymbol{\beta}}}$ instead:

$$\hat{R}_{f_{\boldsymbol{\beta}}} = \hat{\mathbb{E}}_{(X,Y)}[L(\boldsymbol{\beta})] = \frac{1}{N}\sum_{n=1}^{N} l(y_n, f_{\boldsymbol{\beta}}(x_n); \boldsymbol{\beta}) \tag{7.1.2}$$

This means that optimization in machine learning has a different goal as optimization has in other fields, as we are not so much concerned with finding an absolute minimum for the empirical risk as with finding a value of the parameter

$\boldsymbol{\beta}$ that is good enough as to give a low risk. For instance, we know that overfitting appears when $\boldsymbol{\beta}$ gives a very low empirical risk but a relatively large risk. Stochastic gradient descent (SGD) is based on the idea that *we would rather have a less costly but less accurate solution to our empirical risk minimisation problem than a more costly and more accurate solution.*

7.1.2 SGD as an unbiased estimator of the true gradient

In ML problems, the (empirical) loss function function is used as cost function $J(\boldsymbol{\beta})$ with the goal of averaging the individual losses for all the elements of the training set. Then the cost function (without regularization) is expressed as:

$$J(\boldsymbol{\beta}) = \frac{1}{N}\sum_{n=1}^{N} l(y_n, f_{\boldsymbol{\beta}}(x_n); \boldsymbol{\beta}) \tag{7.1.3}$$

meaning that for the gradient we have:

$$\nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}) = \frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\beta}} l(y_k, f_{\boldsymbol{\beta}}(x_k)) \tag{7.1.4}$$

If we now define a uniform random variable I taking values in $\{1, .., N\}$, we have that $\nabla_{\boldsymbol{\beta}} l(y_I, f_{\boldsymbol{\beta}}(x_I))$ is a random variable that has expected value:

$$\mathbb{E}_I(\nabla_{\boldsymbol{\beta}} l(y_I, f_{\beta}(x_I))) = \nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}) \tag{7.1.5}$$

7.1.3 Cost functions for regression

In **regression problems**, a very common cost function is the regularized *mean square error* (MSE):

$$L(\boldsymbol{\beta}) = \frac{1}{2N}\sum_{n=1}^{N} ||y_n - f(\boldsymbol{x_n}; \boldsymbol{\beta})||_2^2 + \frac{1}{2N}\lambda\Omega(\boldsymbol{\beta}) \tag{7.1.6}$$

where $\Omega(\boldsymbol{\beta})$ is usually a norm, for instance $\Omega(\boldsymbol{\beta})$=$||\boldsymbol{\beta}||_2^2$ (l-2 norm) if $\boldsymbol{\beta}$ is a vector, and $\Omega(\boldsymbol{\beta})$=$||\boldsymbol{\beta}||_F^2$ (Frobenius norm) if $\boldsymbol{\beta}$ is a matrix, and where we usually exclude the *bias* parameters, i.e. constant terms that are added to the linear combinations of inputs at each neuron. Note that instead of averaging over N, we average over $2N$ because by using quadratic loss functions, when producing the gradient of the cost function, the factor 2 disappears.

7.1.4 Cost functions for classification

In **classification problems** a very common cost function is the so called *cross entropy function*:

$$L(\boldsymbol{\beta}) = -\frac{1}{N}\sum_{n=1}^{N} \mathbf{1}_{\{y_n=1\}} log(f(\boldsymbol{x_n};\boldsymbol{\beta})) + \mathbf{1}_{\{y_n=0\}} log(1 - f(\boldsymbol{x_n};\boldsymbol{\beta})) + \frac{1}{2N}\lambda\Omega(\boldsymbol{\beta}) \tag{7.1.7}$$

where we have also added a regularization term.

7.1.5 Optimization methods

The most commonly used optimization method is the *gradient descent algorithm*, which requires the evaluation of the gradients $\nabla_\beta L(\beta)$ for the different elements of the set $\mathbf{T}$. See section 2.3 for a review of SGD.

7.2 GPU computations

GPUs were originally designed to render graphics, while CPUs are designed to control the logic flow of any general-purpose program. GPUs are characterised by many more processing units executing in parallel the same operation on different data (SIMD parallelism) and higher aggregate memory bandwidth, while CPUs feature more sophisticated instruction processing and higher clock speed. At each step of the stochastic gradient descent algorithm, we must evaluate the gradients with respect to the parameters defining the FNN. As we have seen this can be seen as a consecutive matrix-by-matrix and matrix-by-vector multiplication. In addition, the gradients of a minibatch can be calculated in parallel. GPUs are particularly well suited for this type of task, which means that they speed up the learning phase for large networks (e.g. by a factor of 50), see Figure 39.

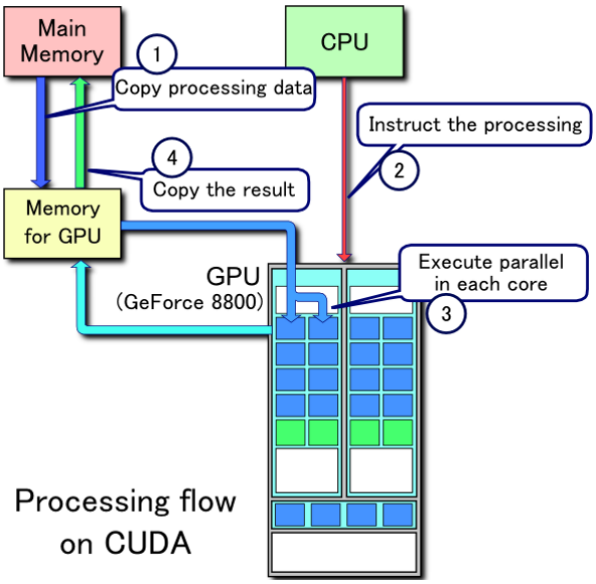


Figure 39: CUDA flow.

7.3 Ill-conditioned problems and non-convexity

In general, $L(\boldsymbol{\beta})$ in neural networks is *non-convex*, meaning that many of the properties of convex problems (e.g. local minima is a global minima) does not hold in these problems. Some of the more common problems that we encounter in optimizing these functions are:

- *Ill-conditioning*: the landscape of the loss function can have very deep and narrow valleys, meaning that the gradient will make the convergence path to bounce between the walls of the valley with very small progress towards the optimal solution. This is the result of an ill-conditioned Hessian matrix (i.e. the Hessian matrix can be positive definite but with a large ratio between the largest and smallest eigenvalues);
- *Local minima*: as the cost function is no convex, even if the algorithm converges to a local minima, this is probably not the global optimization solution. Due to the symmetry of the cost functions with respect to the parameters of a given layer, if we have a local minimum this in fact means that we have a large number of local minima. According to [Goodfellow]: "Whether networks of practical interest have many local minima of high cost and whether optimization algorithms encounter them remain open questions. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case.";

- *Saddle points*: according to [Goodfellow]: "Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher-dimensional spaces, local minima are rare, and saddle points are more common [...] the expected ratio of the number of saddle points to local minima grows exponentially with $|\boldsymbol{\beta}|$. To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In a single dimension, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads. This happens for many classes of random functions. Does it happen for neural networks?";

- *Cliffs and exploiting gradients*: neural networks with many layers often have extremely steep regions resembling cliffs. These are the result of multiplying several large weights together. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off the cliff structure altogether. One solution is to limit the maximum value of the gradient to a maximum value (*gradient clipping*);

- *Vanishing gradients*, this is probably the most important difficulty in training many NN's (e.g. recurrent NNs). It arises when the gradient with respect to some parameters becomes extremely small, because they are computed as the product of many terms that can be smaller than 1. This issue will be discussed later.

7.4 Reducing the noise of the convergence path

A method for reducing the noisy convergence paths in SGD is to low-pass filter the sequence of obtained gradients. If we reduce the learning rate γ to slow, convergence may not be achieved, , figure 40.a), and if we reduce the learning rate γ to fast, figure 40.b), convergence stalls. Here, we review some of the methods to minimize the cost function $\mathrm{L}(\boldsymbol{\beta})$ using GD algorithms.

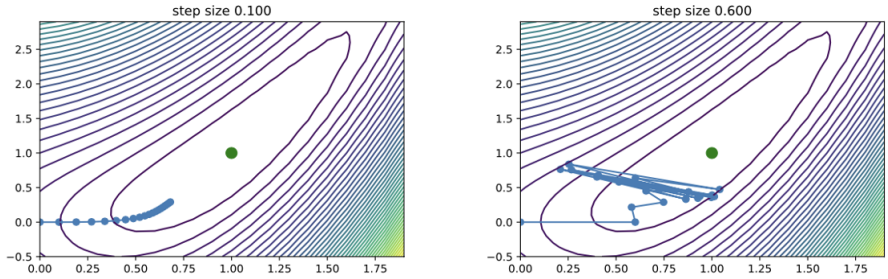


Figure 40: GD in a convex function where the minimum is at (1.0, 1.0); using step size (learning rate) of a) γ=0.1, and b) γ=0.6.

7.5 Momentum

The goal of **momentum** is to avoid ravines (areas where the surface curves much more steeply in one dimension than in another). To avoid oscillations around the slopes of ravines, momentum accelerate SGD in the relevant direction and dampens oscillations by adding a fraction α=0.9 (called the momentum coefficient). We define $\boldsymbol{v}$ as the velocity (called by someones momentum, $\boldsymbol{m}$), and $\boldsymbol{g}^{(k)}$=$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k)})$ the gradient. We then update by following the steps below:

$$\begin{aligned}\boldsymbol{v}^{(k)} &= \alpha \boldsymbol{v}^{(k-1)} + \boldsymbol{g}^{(k-1)} = \alpha \boldsymbol{v}^{(k-1)} + \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k-1)}) \\ \boldsymbol{\beta}^{(k)} &= \boldsymbol{\beta}^{(k-1)} - \gamma^{(k-1)} \boldsymbol{v}^{(k)} = \boldsymbol{\beta}^{(k-1)} - \gamma^{(k-1)} (\alpha \boldsymbol{v}^{(k-1)} + \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k-1)}))\end{aligned} \tag{7.5.1}$$

If α=0, we recover the original SGD method. We update the parameters with the average of gradients instead of the most recent gradient. The momentum is like an exponentially weighted moving average of the past gradients:

$$\boldsymbol{v}^{(k)} = \alpha \boldsymbol{v}^{(k-1)} + \boldsymbol{g}^{(k-1)} = \alpha^2 \boldsymbol{v}^{(k-2)} + \alpha \boldsymbol{g}^{(k-2)} + \boldsymbol{g}^{(k-1)} = \sum_{j=0}^{k-1} \alpha^j \boldsymbol{g}^{(k-j-1)} \tag{7.5.2}$$

In the limit, for constant gradient $\boldsymbol{g}$ and infinite k, $\boldsymbol{v}^{(\infty)}$=1/(1-$\alpha$)$\boldsymbol{g}$, meaning that we take a step of size γ/(1-α) instead of a step of size γ. Then, momentum plus SGD simulates the effect of a large mini-batch.

7.6 Nesterov Momentum

Nesterov accelerated gradient (NAG) or **Nesterov momentum** aims to improve momentum. The main problem with momentum is that we first compute the gradient and then make a big jump (using the momentum term in the gradient direction). The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. NAG makes a big

jump in the direction of the previous accumulated gradient and then measures the gradient where it ends up and makes a correction:

$$\begin{aligned}\boldsymbol{v}^{(k)} &= \alpha\boldsymbol{v}^{(k-1)} - \gamma^{(k-1)}\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k-1)} + \alpha\boldsymbol{v}^{(k-1)}) \\ \boldsymbol{\beta}^{(k)} &= \boldsymbol{\beta}^{(k-1)} + \boldsymbol{v}^{(k)} = \boldsymbol{\beta}^{(k-1)} + \alpha\boldsymbol{v}^{(k-1)} - \gamma^{(k-1)}\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}^{(k-1)} + \alpha\boldsymbol{v}^{(k-1)})\end{aligned} \tag{7.6.1}$$

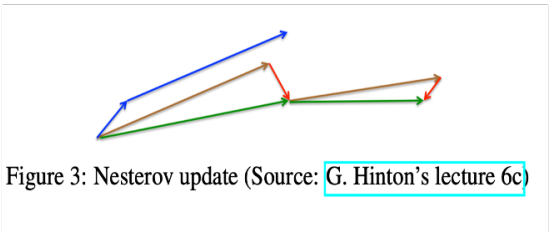


Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

Figure 41: Nesterov momentum: momentum (in blue) that obtains the gradient and makes a big jump; NAG (in brown) that makes a big jump, then a gradient correction ending in the green line improving the responsiveness.

7.7 Others

There are a whole set of variation methods that apply other ideas such as applying the Newton's method (and its variation called the quasi-Newton's method, BGFS). Other methods are SVRG and SAGA that aim to reduce the variance in SGD to have a convergence similar to BGDM. One of the most used is Adam that uses exponentially decaying averages of estimates of the first and second moments of the past gradients, and so on with methods such as Adagrad, Adadelta, AdaMax, RMSProp, etc, see [S. Ruder] tutorial.

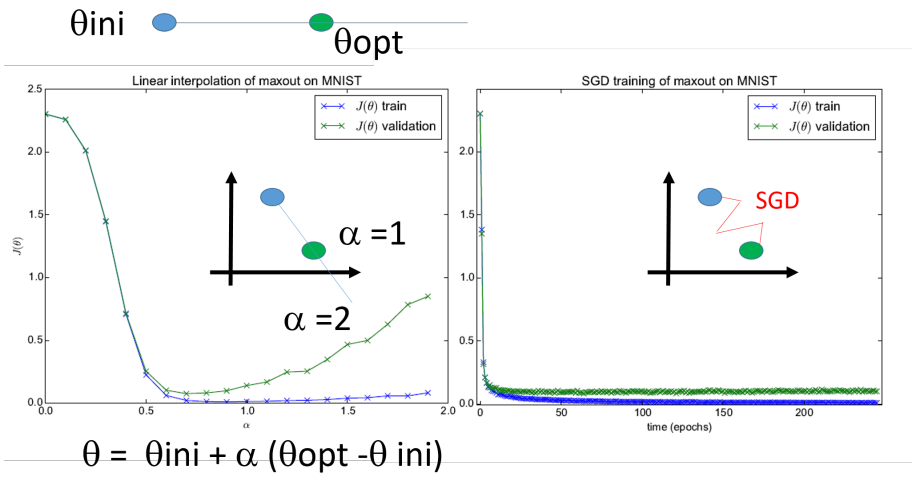
7.8 Initialization

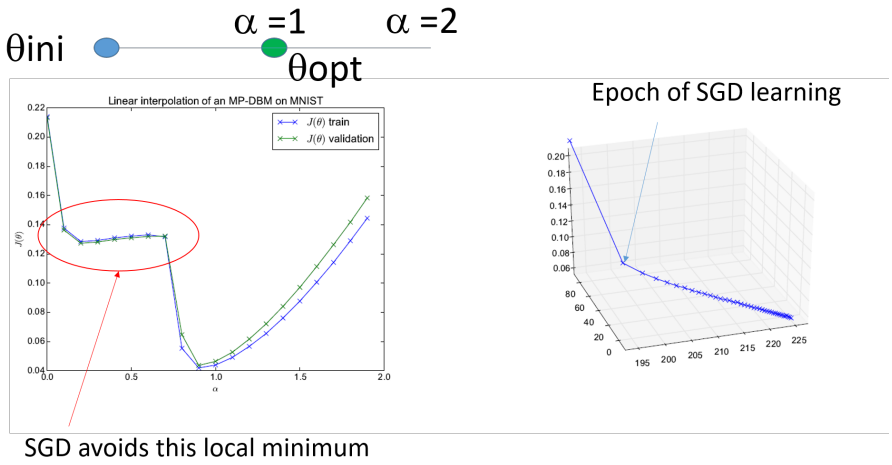
Another aspect that is equally important is initialization. Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly (e.g., taken from a Gaussian distribution or uniform distribution).

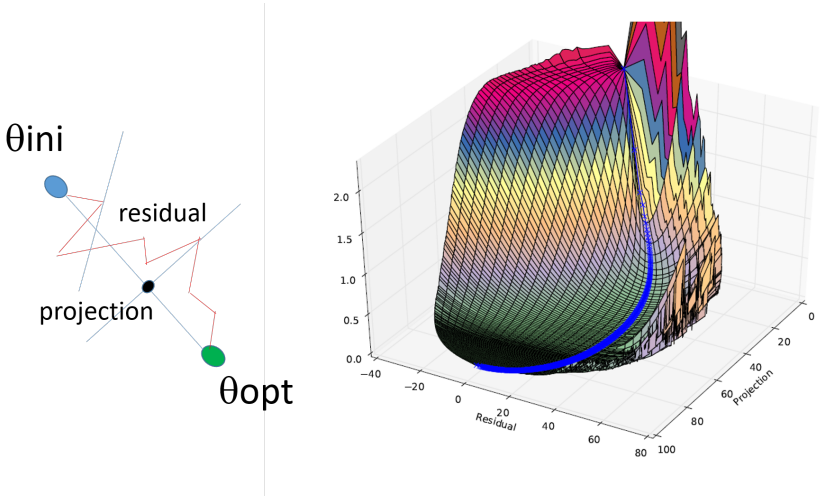
8 Visualization of the loss functions

For a given value x and y, understanding the shape of the cost function is extremelly challenging as it is a function of a large number of *weights*. Some attempts of understanding these landscapes give some intuition on the form of these functions.

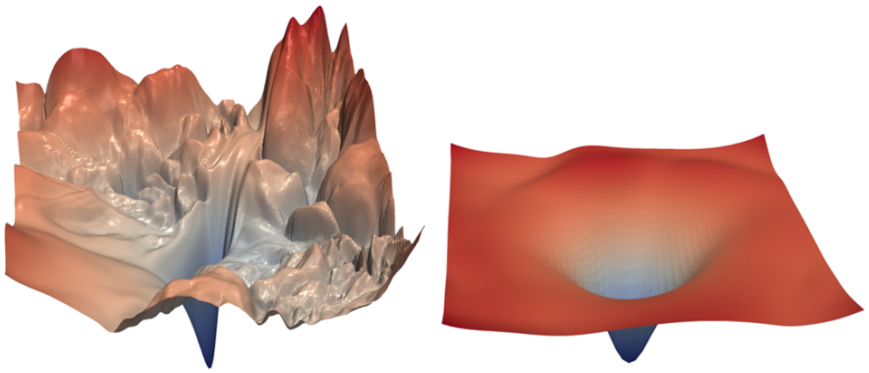
The following three figures show the values of the cost function in the line between an initialization point and the final optimal point found by the algorithm.







The following figures plot the values of the cost functions Using random directions plane and scaling for two different architectures (to be discussed in more detail in a following LNs). We observe the different landscapes that result from these two different architectures, which have immediate consequences in terms of convergence of the SGD. Note that minima in the figures are most probably saddle points as we only show two dimensions.



(a) without skip connections (b) with skip connections

9 Regularization

9.1 Norm penalty in the cost function

9.2 Dataset augmentation and robustness to noise

Used for instance in object recognition: take a picture and make transformations (rotations, translations, etc) to increase the size of the training set.

Another possibility is to add noise.

9.3 Early stopping

Keep a copy of the parameters for which the cost *for the validation set* reached a minimum. This means that for each iteration we run the model for a small validation set to assess this cost.

9.4 Parameter sharing

Set some parameters to be equal, meaning that we reduce the number of weights. For instance, used in Convolutional Neural Networks (CNNs).

9.5 Model averaging (Bagging)

Use different NN models and do the average of the result. Works well but requires a lot of computation is we use many models. Usually used in practice by averaging just a few number of models (e.g. 10).

Bagging involves constructing different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset.

We can obtain different NN models by use different initialization values of the weights, by using different minibatches in the SGD, etc.

9.6 Dropout

Use a single NN models and dataset but trying to achieve a similar effect as using many models. At each gradient computation use as parameter the value $b_{ij}\beta_{ij}$, where $b_{ij} = 1$ with a probability p, and $b_{ij} = 0$ with probability $1 - p$. The value of p may be different for different layers. For instance, for input layers we can use $p = 0.8$, while for hidden layers we can use $p = 0.5$.

Once the optimization is done, *multiply the parameters by* p, i.e. use in the testing $p\beta_{ij}$. This is called *weight scaling inference rule*.

There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.