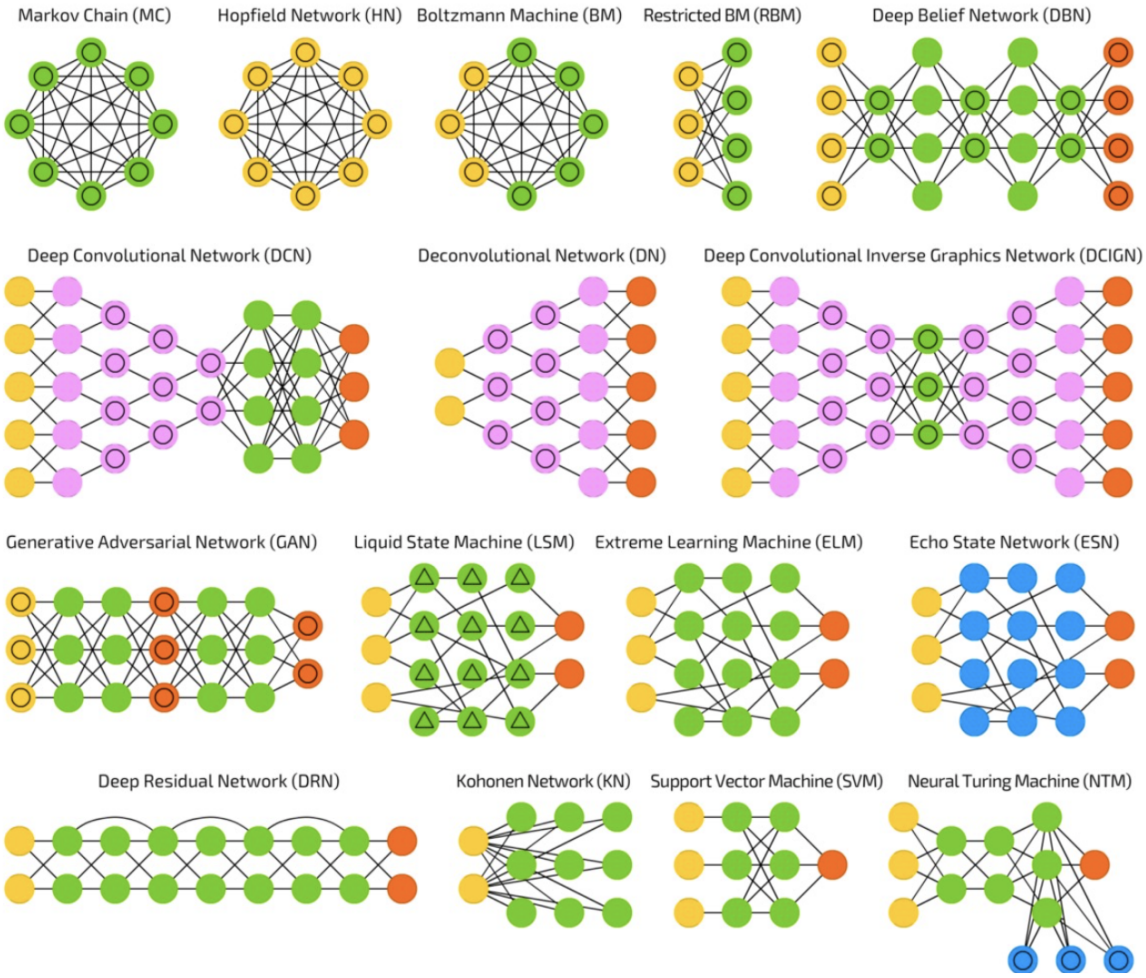
NOTES

Deep Learning

2223-1-FDS01Q012



Vittorio Haardt
853268
vittoriohaardt@gmail.com

April 6, 2023

Abstract

The aim of this course is to provide the theoretical foundations of mathematics and statistics for deep learning including linear algebra, optimization, regularization, and dimensionality reduction. The most important deep neural network architectures will be covered in this course. Thanks to a practical part of the course, the student will be able to handle the main tools for deep learning and then design and optimize a deep neural network.

Contents

Chapter 1

Introduction

1.1 Artificial Intelligence (AI)

Today, **artificial intelligence** (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research. AI has an impact in our life with many ethical implications.

It is important to look at some formal definitions. First definition was from John McCarthy, who coined the term in 1956 as "*the science and engineering of making intelligent machines*", and later (1997) he defines as "*Intelligence is the computational part of the ability to achieve goals in the world*".

One of the AI pioneers is Alan Turing. Quoting McCarthy: "The English mathematician Alan Turing may have been the first. He gave a lecture on it in 1947. He also may have been the first to decide that AI was best researched by programming computers rather than by building machines.". Alan Alan Turing's 1950 article Computing Machinery and Intelligence discussed conditions for considering a machine to be intelligent by introducing a Test called **the imitation game**, later known as **the Turing Test**.

He argued that if the machine could successfully pretend to be human to a knowledgeable observer then you certainly should consider it intelligent. The observer could interact with the machine and a human by teletype (to avoid requiring that the machine imitate the appearance or voice of the person), and the human would try to persuade the observer that it was human and the machine would try to fool the observer. The Turing test is a one-sided test. A machine that passes the test should certainly be considered intelligent, but a machine could still be considered intelligent without knowing enough about humans to imitate a human.

There are many AI branches, for example:

- Logic AI: programming an intelligent system with the use of the logic
- Pattern Recognition: AI programmed to compare what it sees with a pattern
- Common sense knowledge and reasoning: recently important for explainable AI
- Learning from experience: AI based on connectionism and neural nets
- Ontology: Formal Languages to describe structured knowledge
- Heuristics: approximation of formal solutions
- Genetic Programming: evolving programs following genetic processes
- many others

1.2 From AI to Deep Learning

1.2.1 Machine Learning

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers-problems that can be described by a list of formal, mathematical rules. Several artificial intelligence projects have sought to hard-code knowledge about the world in formal languages. A computer can reason about statements in these formal languages automatically using logical inference rules.

The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as *machine learning*. The simplest machine learning algorithms are linear regression or logistic regression.

- **Linear Regression**: describes a relationship between predictor (independent) and response (dependent) continuos variables, such as price, height, distance, etc.

$$y = \beta_o + \beta_1 x + \epsilon$$

- **Logistic Regression**: describes a relationship between predictor (independent) and response (dependent) categorical variables, such as yes/no, male/female etc.

$$p = \frac{1}{1 + e^{-(\beta_o + \beta_1 x)}}$$

We can start to talk about **learning a mapping function** f, where $x \to f(x) = y$, it is a function between predictor (independent) x and response (dependent) y variables. The performance of machine learning algorithms depends heavily on the representation of the data they are given, also called features.

One big challenge in deep learning is the **curse of dimensionality**, which refers to the fact that as the number of features or dimensions in a dataset grows, the amount of data required to generalize accurately also grows exponentially. This means that we need a much larger sample size than the number of dimensions to accurately represent the data. To address this challenge, various techniques are often used, including projections onto different spaces. The specific feature extraction techniques used will depend on the nature of the data being analyzed. For example, in the case of image data, *Gabor filtering* may be used to extract features, or histograms may be used to distinguish between different types of images (e.g. pasta vs. coffee). By using these techniques, we can reduce the dimensionality of high-dimensional datasets (e.g. 512x512 images) into a lower-dimensional vector (e.g. 40 dimensions) that can be used as input for a machine learning algorithm. It is important to note that while these techniques can help mitigate the curse of dimensionality, they may also result in some loss of information. Therefore, it is important to carefully consider the trade-offs between dimensionality reduction and information preservation when choosing feature extraction techniques.

However, for many tasks, it is difficult to know what features should be extracted. One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as **representation learning**. An example of a representation learning algorithm is the *autoencoder*, that is the combination of an encoder function that converts the input data into a different representation, and a decoder function that converts the new representation back into the original format (almost similar).

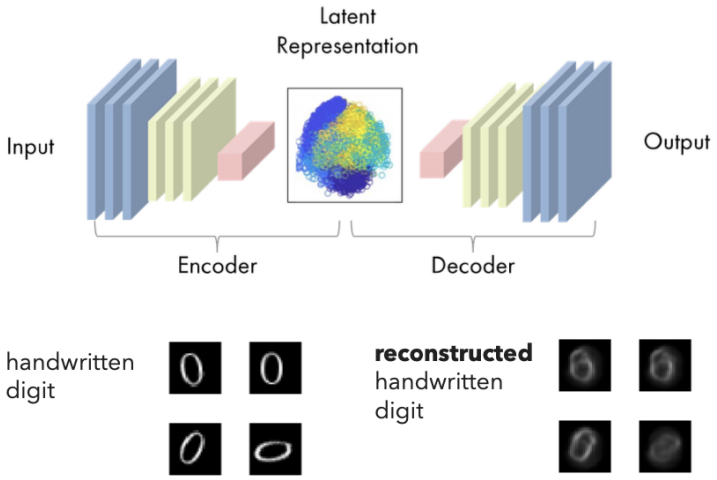


Figure 1.1: Autoencoder example.

Autoencoders are very generalizable and can be used on different data types, including images, time series, and text.

When designing features or algorithms for learning features, our goal is usually to separate the factors of variation that explain the observed data. Factors of variation can be thought of as concepts or abstractions that help us make sense of the rich variability in the data, like feature on an human face.

1.2.2 Deep Learning

Deep learning introduces representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. An example of a deep learning model is the feedforward deep network or *multilayer perceptron* (MLP).

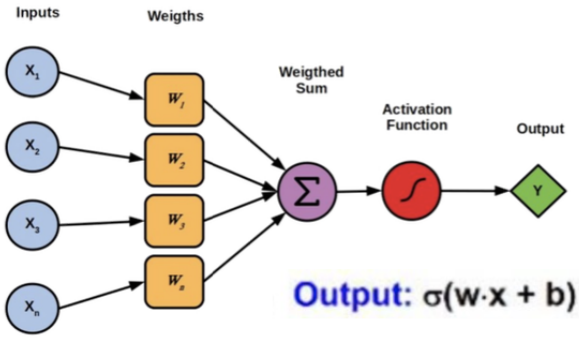


Figure 1.2: Artificial Neuron (perceptron).

A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions.

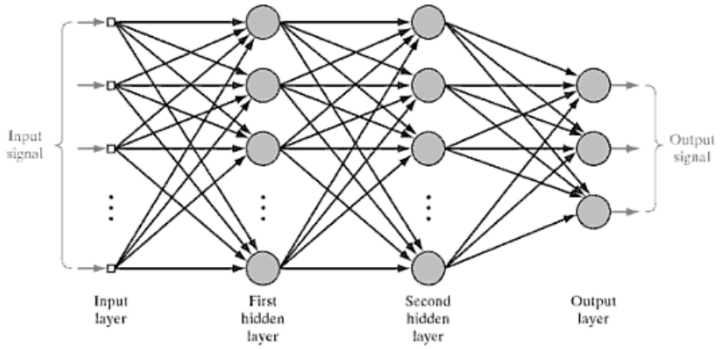


Figure 1.3: Multilayer Perceptron.

Deep learning can safely be regarded as the study of models that either involve a greater amount of composition of learned functions or learned concepts than traditional machine learning does. Deep Learning it is a type of machine learning, a technique that allows computer systems to improve with experience and data. Is the only viable approach to building AI systems that can operate in complicated, real-world environments.

Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a *nested hierarchy of concepts*, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. This concept is similar to the architecture used by mammalian brains for object recognition.

MLP A Multi-Layer Perceptron (MLP) is a classification technique based on separation between spaces of attributes. This models are very common nowadays for deep learning.

A **Multi-Layer Perceptron** consists of artificial neurons which communicate unidirectionally, from the input attributes X to the class attribute Y.

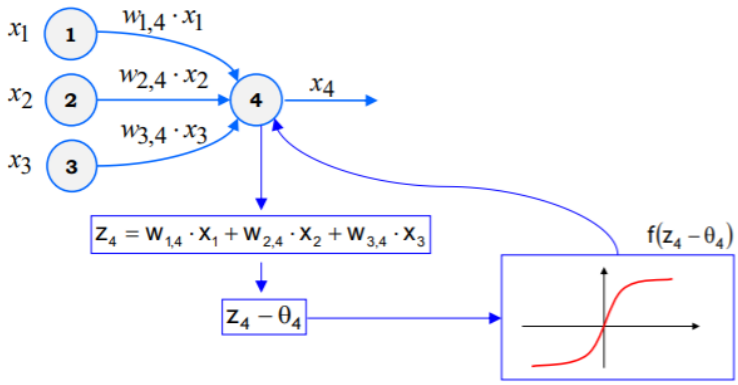


Figure 1.4: Example of the a MLP's structure.

In figure there are tree neuron of continuous parameters, to which is added a fourth neuron to compute a function. A weight w is associated with every neuron. A neuron compute a **linear combination** between the input attributes and its weight, the j-th neuron compute:

$$y_j = f(\sum_{i=1}^{n} w_{i,j} \cdot x_i - \theta_j)$$

For example, the fourth neuron in figure is:

$$z_4 = w_{1,4} \cdot x_1 + w_{2,4} \cdot x_2 + w_{3,4} \cdot x_3$$

To this computation is given a threshold (or bias) value θ applied to the linear combination. Subsequently, a transfer function is applied on the neuron minus the threshold, in our case $f(z_4 - \theta_4)$.

The **transfer function** is defined as the function applied on a neuron that gives the value brought to another communicating layer.

As the name suggests, a MLP use multiple perception. A perceptron is the most simple neural network with only one layer. It is composed by the inputs x with the wight w that combine in the output layer. As for the MLP the inputs are combined in a linear combination $H = w_0 + \sum_{j=1}^{p} w_j x_{ij}$ to which is applied a transfer function F. It is usable only if the problem is linear separable, because it produce a linear decision boundary. We introduce the MLP because it is an evolution of the perceptron with more complex decision boundaries.

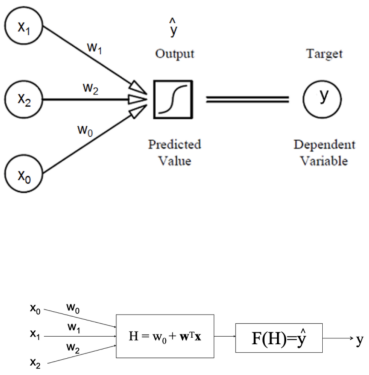


Figure 1.5: Structure of a perceptron.

The most used transfer function are the following:

- *Identity/linear*: multiple linear regression is a perceptron that essentially assumes that a linear activation function F is applied to the linear combination. The difference is that the network estimates the w weights with the descendent gradient.
- *Sigmoid/logistic*: the logistic regression is a simple perceptron with sigmoid F activation function. The difference is that the network estimates the weights w differently from maximum likelihood. It is useful if we want expected probability as output.
- *Sign*: the binary target is coded with $(+1, -1)$ if the dependent variable is binomial. F processes the combination and based on the sign obtains the output (the expected value). Consequently the sign will represent the decision boundary.
- *Kernel*: we may be transfer functions that activate the functions according to particular distances based on the Kernel. Which can be a Gaussian.

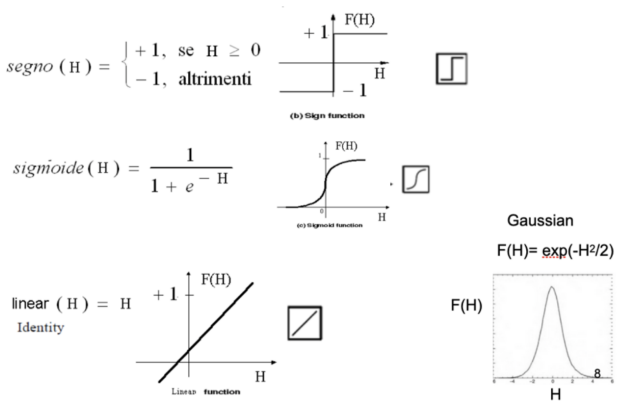


Figure 1.6: Perceptron's transfer functions.

Thinking of a MLP with 2 layers, the input attributes arrive with their weight in the first layer (hidden layer), they are treated as perceptrons so a transfer function F is applied on each one. Then they are used as input for the second layer (the output layer in this case) and linearly combined with new weights as $S = \sum_j w_j F(H_j)$. Then a new transfer function G is used on the linear combination S.

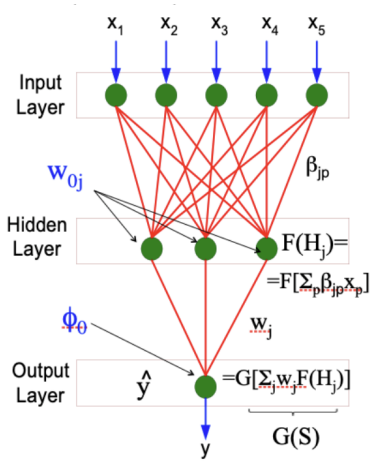


Figure 1.7: Perceptron's transfer functions.

In the traditional network there are typically two or tree hidden layers, but in the *deep learning* we can have a huge number of them. The transfer function typically used in the hidden layers are:

- *Hyperfolic tangent*: $F(H) = \frac{e^H - e^{-H}}{e^H + e^{-H}}$
- *Logistic*: $F(H) = \frac{1}{1+e^(-H)}$

Differently, it the used output layer the transfer functions G are:

- *Identity/linear*: $G(S) = S$. Used if the problem is a standard regression.
- *Logistic/sigmoid*: $G(S) = \frac{1}{1+e^(-S)}$. Used if the problem is a binary classification.
- *Softmax*: $G(S) = \frac{e^{S_i}}{\sum_j e^{S_j}}$. Used for multiclass problems.

Exploding gradient or vanishing gradient could cause errors at edges of the transfer function, so it is preferable to use more simple functions.

As anticipated there are tree type of neurons:

- **Input**: they are connected with the explanatory attributes and with *each* hidden node.
- **Hidden**: they receive as input the linear combination of the input neurons and the signal is propagated to the output ones.
- **Output**: they are associated with the class variable and receive the signals to be displayed.

Each input neuron is connected through directional links with all hidden neurons. The signal propagates form neurons of the input layer to neurons of the hidden layer, then the output neuron labels the records. This type of network is called **fully-connected**. Every connection has its weight and every node has its threshold θ_j and its transfer function.

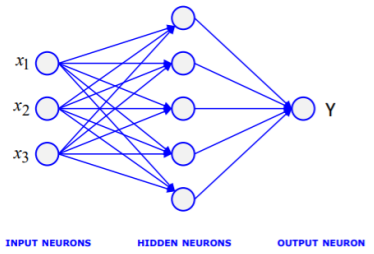


Figure 1.8: Simple MLP model.

MLPs can have different architectures, i.e., different number of hidden layers and different number of hidden neurons for each hidden layer. There are no constraints on communicating by skipping levels. However, it is not possible to communicate backwards, in fact these networks are called **feed-forward neural networks** (they can have up to hundreds of hidden layers.

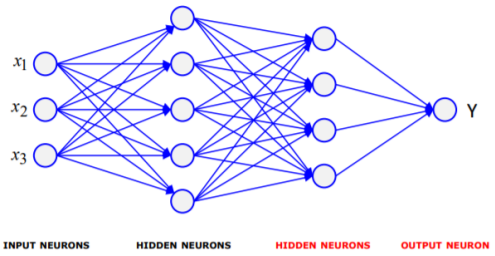


Figure 1.9: Example of a MLP with two hidden layer.

MLP learning is a difficult task, different optimization measures have been proposed in the specialized literature, many different learning algorithms have been designed and described, but **multiple local minima usually exist** and thus it is always extremely difficult to learn an MLP. The number of nodes, arc and hidden levels it is usually selected by trials and experience.

Now lets rapidly see how network parameters are estimated. They are not estimated with the gradient descendant but rather with **back propagation** algorithm. The mechanism of updating the estimates becomes complex because the output of the network depends on many factors that must be taken into account. This

is the loss function:

$$J(w) = \frac{1}{n}\sum_i (y_i - \hat{y_i})^2$$

Every iteration has two phases:

1. *Forward phase*: where, with random initial weights, the network is propagated from the input layers to the output ones, giving final outputs $\hat{y_i}$.
2. *Backward phase*: where the error between the observed y_i and predicted $\hat{y_i}$ is propagated/separated back to change the initial layers' weights in an adaptive way.

The parameters are updated in the gradient opposite direction, splitting the latter into the individual components that generated the outputs in the forward phase. The problem is to derive the gradient of the loss function as $J(w)$ is generated by a lot of steps in the hidden layers. We will split the gradient for each contribution to get the target.

A MLP tuning parameters are:

- *Early stop*: it can be used to prevent the MLP from overfitting. The last iteration can be selected using the residual sum of square on the training set.
- *Decay*: it adjusts the update rate of parameters (weights) from in sequential iterations.
- *The penalty parameter* λ: it tries to "reduce" the parameter estimates towards zero: large parameter estimates are penalized.

By using the back propagation algorithm to a perceptron, it can be viewed as a logistic regression that estimates parameters with a descendent gradient. It is important to observe if the algorithm converges (how many epochs are necessary).

MLPs are universal approximators, it means that with at least two layers and a sufficient number of neurons in the Hidden layer (at least three): can approximate the shape of the relationship between x and y with few information. The MLPs are considered *shallow networks* because they have few layers and many neurons, on the contrary of the deep networks.

1.3 Artificial Intelligence Evolution

The traditional machine learning approach typically involves several steps: raw input data is preprocessed and transformed into a set of relevant features, which are then used as input for a machine learning algorithm. The algorithm learns to identify patterns and relationships between the features and the output variable, and makes predictions on new, unseen data.

In contrast, deep learning models can often bypass the feature engineering step, as they are capable of learning hierarchical representations of the input data through the use of artificial neural networks. This means that raw input data can be fed directly into the network, and the network learns to extract the relevant features for the task at hand. However, it can still be challenging to find the right architecture and hyperparameters for the deep learning model, and in some cases, hand-designed features may still be necessary to achieve optimal performance.

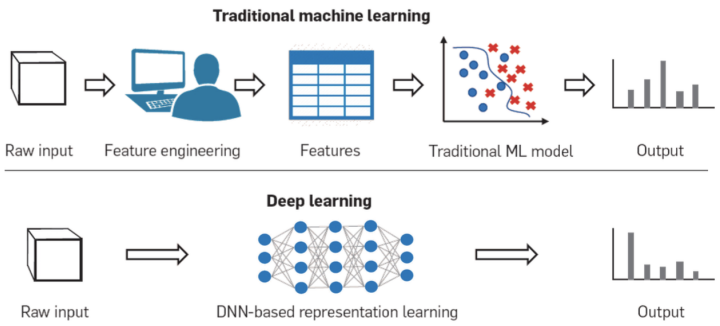


Figure 1.10: ML vs DL approach.

Hybrid pipelines, which combine both traditional machine learning and deep learning approaches, are becoming increasingly common. For example, in audio classification tasks, a spectrogram can be used as input to a deep learning model. The spectrogram is a visual representation of the audio waveform, which can be processed by a deep convolutional neural network to extract features and make predictions. This approach combines the advantages of both traditional feature engineering and deep learning methods to achieve high performance on audio classification tasks.

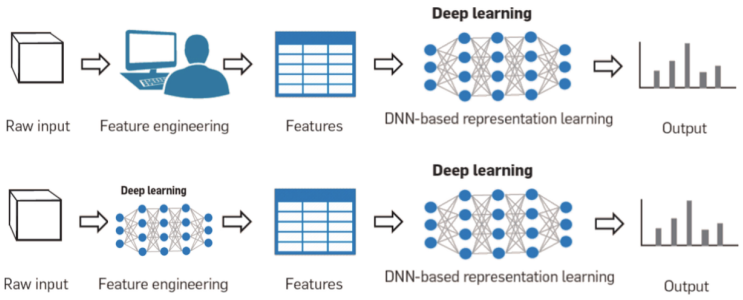


Figure 1.11: Advanced DL approach.

The original goal of the AI field was the construction of "thinking machines", that is, computer systems with human-like general intelligence. Due to the difficulty of this task, for the last few decades the majority of AI researchers have focused on what has been called "narrow AI", the production of AI systems displaying intelligence regarding specific, highly constrained tasks. In recent years, however, more and more researchers have recognized the necessity, and feasibility, of returning to the original goals of the field. Increasingly, there is a call for a transition back to confronting the more difficult issues of "human level intelligence" and more broadly "artificial general intelligence (AGI)."

Gato is what DeepMind describes as a "general-purpose" system, a system that can be taught to perform many different types of tasks. Researchers at DeepMind trained Gato to complete 604, to be exact, including captioning images, engaging in dialogue, stacking blocks with a real robot arm and playing Atari games.

1.4 Deep Learning

There are several definitions of **deep learning**:

- Definition 1: A class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification.
- Definition 2 : A sub-field within machine learning that is based on algorithms for learning multiple levels of representation in order to model complex relationships among data. Higher-level features and concepts are thus defined in terms of lower-level ones, and such a hierarchy of features is called a deep architecture. Most of these models are based on unsupervised learning of representations.
- Definition 3: A sub-field of machine learning that is based on learning several levels of representations, corresponding to a hierarchy of features or factors or concepts, where higher-level concepts are defined from lower- level ones, and the same lower-level concepts can help to define many higher-level concepts. Deep learning is part of a broader family of machine learning methods based on learning representations.
- Definition 4: Deep learning is a set of algorithms in machine learning that attempt to learn in multiple levels, corresponding to different levels of abstraction. It typically uses artificial neural networks. The levels in these learned statistical models correspond to distinct levels of concepts, where higher-level concepts are defined from lower-level ones, and the same lower- level concepts can help to define many higher-level concepts.
- Definition 5 : Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals: Artificial Intelligence. Deep Learning is about learning multiple levels of representation and abstraction that help to make sense of data such as images, sound, and text.
- Definition 6: Deep learning is a subset of machine learning that involves the use of artificial neural networks to enable computers to learn and make decisions based on data. These neural networks are composed of multiple layers of interconnected nodes or neurons that work together to process and analyze input data. The term "deep" in deep learning refers to the number of layers in these neural networks. Deep neural networks can have multiple layers, each of which performs a different type of computation, allowing them to learn complex patterns and relationships in data. This makes them particularly effective in tasks such as image recognition, natural language processing, and speech recognition.
- Definition 7: Deep learning is a type of machine learning that uses artificial neural networks to analyze and make decisions based on data. It involves building and training complex models with multiple layers of interconnected neurons, which can learn and recognize patterns in the data. This allows deep learning to be applied to tasks such as image and speech recognition, natural language processing, and many others.

A real issue with DL is computational cost, but it was mitigate in the last years by the increasing in computational resources. Computational cost is a measure of the amount of computer resources that a ML/DL algorithm uses in training and/or inference. It is useful to know how much time or computing power is required Computer resources are:

- Number of operations (Central Processing Unit - CPU, Graphical Processing Unit – GPU, Neural Processing Unit – NPU, others). It is measured as the number of floating point operations (FLOPs) or FLOPs per Second (FLOPS), and also as number of multiply-and-accumulate operations (MACs or MACCs).
- Memory usage (bytes) correlates with the number of parameters.

Chapter 2

Linear Algebra

2.1 Linear Algebra Bases

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms.

Linear algebra is important in deep learning because it provides a mathematical foundation for representing and manipulating high-dimensional data, which is ubiquitous in many machine learning applications. For example, the input data to a deep learning model is often represented as a high- dimensional tensor or matrix. Linear algebra provides powerful tools for manipulating and transforming these tensors, such as matrix multiplication, transpose, and inversion, which are used extensively in deep learning algorithms. Furthermore, many important operations in deep learning, such as gradient computation, optimization, and regularization, can be expressed in terms of linear algebra operations. Linear algebra also provides a way to formally analyze the performance and behavior of deep learning algorithms.

Scalars A **scalar** is a single number, like: integers, real numbers, rational numbers, etc. We denote it with italic font and lowercase: a, n, x. For real-valued scalar, we might say "Let $s \in R$ be the slope of a line", for natural number scalar, we might say "Let $n \in N$ be the number of units.

Vectors A **vector** is a 1D array of numbers, the numbers are arranged in order, and they can be: real, binary, integer, etc. Vectors are point in space with each elegant giving the coordinate along a different axis.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

If each element is in $\mathbb{R}$, and the vector has n elements, then the vector lies in the set formed lies in the set formed by taking the Cartesian product of $\mathbb{R}$ n times: $\mathbb{R}^n$.

2.1.1 Matrices

A **matrix** is a 2D array of numbers, so each element is identified by two indices instead of just one. The numbers are arranged in order and can be: real, binary, integer, etc.

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

If a real-valued matrix $\boldsymbol{A}$ has a height of m and a width of n, then we say that $A \in \mathbb{R}^{m \times n}$.

Tensors In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a **tensor**. We denote a tensor named "A" with this typeface: $\boldsymbol{A}$. More in general a tensor is an array of numbers, that may have:

- zero dimensions, and be a scalar
- one dimension, and be a vector
- two dimensions, and be a matrix
- or more dimensions.

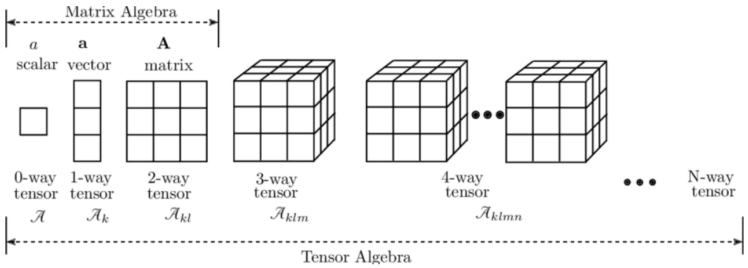


Figure 2.1: Summary of basis.

Matrix Transpose The **transpose** of a matrix is the mirror image of the matrix across a diagonal line, called the main diagonal, running down and to the right, starting from its upper left corner. The operation is defined such that: $(A^T)_{i,j} = A_{j,i}$.

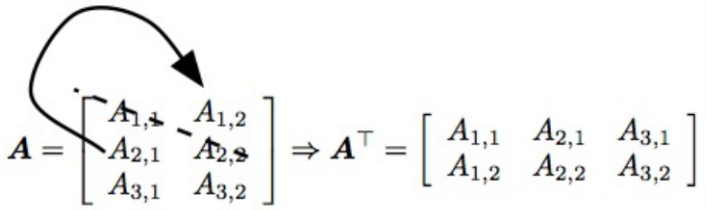


Figure 2.2: Matrix transposition.

The transpose of a matrix product has a simple form: $(AB)^T = B^T A^T$.

Matrix Product The matrix product of matrices $\boldsymbol{A}$ and B is a third matrix C. $\boldsymbol{A}$ must have the same number of columns as B has rows. If $\boldsymbol{A}$ is of shape $m \times n$ and B is of shape $n \times p$, then C is of shape $m \times p$.

$$\boldsymbol{C} = \boldsymbol{AB}$$

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

Matrix Sum We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements:

$$\boldsymbol{C} = \boldsymbol{AB} \text{ where } C_{i,j} = A_{i,j} + B_{i,j}$$

We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix:

$$\boldsymbol{D} = a \cdot \boldsymbol{B} + c \text{ where } D_{i,j} = a \cdot B_{i,j} + c$$

In the context of deep learning, we also use some less conventional notation. We allow the addition of matrix and a vector, yielding another matrix:

$$\boldsymbol{C} = \boldsymbol{A} + b \text{ where } C_{i,j} = A_{i,j} + b_j$$

In other words, the vector b is added to each row of the matrix. This implicit copying of b to many locations is called **broadcasting**. Note that the standard product of two matrices is not just a matrix containing the product of the individual elements.

Identity Matrix and Inversion Matrix An **identity matrix** is a matrix that does not change any vector when we multiply that vector by that matrix.

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n \boldsymbol{x} = \boldsymbol{x}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The structure of the identity matrix is simple: all of the entries along the main diagonal are 1, while all the other entries are zero. In the matrix inversion this matrix is such that: $\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n$.

Norms In machine learning, we usually measure the size of vectors using a function called ***norm***, functions that measure how "large" a vector is. Similar to a distance between zero and the point represented by the vector, it satisfies the following properties:

- $f(\boldsymbol{x}) = 0 \rightarrow \boldsymbol{x} = \boldsymbol{0}$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$
- $\forall \alpha \in \mathbb{R}, f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

The second one is the *triangle inequality*. There are several types of norms usable:

- L^p norm: $\| x \|_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$
- L^2 norm, euclidean norm: $\| x \|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$
- L^1 norm: $\| x \|_p = \sum_i |x_i|$
- Max norm, $p \rightarrow \infty$: $\| x \|_\infty = \max_i |x_i|$

Orthogonality Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are **orthogonal** to each other if $\boldsymbol{x}^T\boldsymbol{y} = 0$. If both vectors have nonzero norm, this means that they are at a 90 degree angle to each other. In $\mathbb{R}^n$ at most n vectors may be mutually orthogonal with nonzero norm. If the vectors are not only orthogonal but also have unit norm, we call them **orthonormal**. Pay careful attention to the definition of orthogonal matrices. Counterintuitively, their rows are not merely orthogonal but fully orthonormal. There is no special term for a matrix whose rows or columns are orthogonal but not orthonormal.

Frobenius Norm Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the **Frobenius norm**:

$$\| \boldsymbol{A} \|_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

The trace operator provides an alternative way of writing the Frobenius norm of a matrix:

$$\| \boldsymbol{A} \|_F = \sqrt{Tr(\boldsymbol{A}\boldsymbol{A}^T)}$$

Trace of a Matrix The **trace** operator gives the sum of all of the diagonal entries of a matrix:

$$Tr(\boldsymbol{A}) = \sum_i \boldsymbol{A}_{i,i}$$

The trace of a square matrix composed of many factors *invariant to moving* the last factor into the first position:

$$Tr(\boldsymbol{ABC}) = Tr(\boldsymbol{CAB}) = Tr(\boldsymbol{BCA})$$

his invariance holds even if the resulting product has a different shape. The trace operator is *invariant to the transpose* operator:

$$Tr(\boldsymbol{A}) = Tr(\boldsymbol{A}^T)$$

Another useful fact to keep in mind is that a scalar is its own trace:

$$a = Tr(a)$$

Vector Space A **vector space** consist of a set V (elements of V are called *vectors*), a field $\mathbb{F}$ (elements of $\mathbb{F}$ are called s*scalars*), and two operations

- An operation called *vector addition* that thakes two vectors $v, w \in V$, and produces a third vector, written $v + w \in W$.
- An operation called *scalar multiplication* that takes a scalar $c \in \mathbb{F}$ and a vector $v \in V$, and produces a new vector, written $cv \in V$.

Which satisfy the following conditions (called axioms).

1. *Associativity* of vector addition: $(u + v) + w = u + (v + w)$ far all $u, v, w \in V$.
2. *Existence of a zero vector*: There is a vector in V, written 0 and called the **zero vector**, which has the property that $u + 0 = U$ for all $u \in V$.
3. *Existence of negatives*: For every $u \in V$, there is vector V, written $-u$ and called the **negative of** u, which has the property that $u + (-u) = 0$.
4. *Associativity of multiplication*: $(ab)u = a(bu)$ for any $a, b \in \mathbb{F}$ and $u \in V$.
5. *Distributivity*: $(a + b)u = au = bu$ and $a(u + v) = au + av$ for all $a, b \in \mathbb{F}$ and $u, v \in V$.

6. *Unitary*: $1u = u$ for all $u \in V$.

Lets now see some definition/reminder.

- Vectors $v_1, .., v_n$ in V are **linearly independent** if whenever $k_1v_1 + k_2v_2 + ... + k_nv_n = 0$ for some $k_1, ..., k_n \in F$, then $k_1 = k_2 = ... = k_n = 0$.
- A **basis** for V is a linearly independent subset B of V with the property that every element of V can be written as a linear combination of elements of B (that is , a $k_1b_1 + k_2b_2 + ... + k_nb_n$,where the k_i's are in F and b_i's are in B).
- The **dimension** of V is the nymber of elements of a basis of B. (fact: every basis of vector space has the same number of elements, so this definition makes sense).

Determinant of a Matrix The **determinant** of Matrix is defined for square matrices and is a scalar obtained by summing and subtracting some matrix elements. It is usually represented with the following symbols: $det(A)$ or $|A|$.

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

Having $det(A) = 0$ means:

- The columns of the matrix are dependent vectors in $\mathbb{R}^n$
- The rows of the matrix are dependent vectors in $\mathbb{R}^n$ The matrix is not inevitable.

For a $n \times n$ matrix, the determinant is achieved using the Laplace expansion:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

We use the concept of co-factor and the concept of submatrix A_{ij} that is the matrix achieved removing the row i and the column j of the matrix A_{ij}. The co-factor of the element a_{ij} , $cof(a_{ij})$, is the following:

$$(-1)^{i+j} \cdot det(A_{ij})$$

The determinant of the matrix is given a:

row $det(A) = \sum_{j=1}^{n}[a_{ij} \cdot (-1)^{i+j} \cdot det(A_{ij})]$
column $det(A) = \sum_{i=1}^{n}[a_{ij} \cdot (-1)^{i+j} \cdot det(A_{ij})]$

Invertibility A matrix can be inverted if *the determinant is not equal to zero.*

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$A^{-1} = \frac{1}{det(A)} \begin{bmatrix} Coef(a_{11}) & Coef(a_{12}) & \cdots & Coef(a_{1n}) \\ Coef(a_{21}) & Coef(a_{22}) & \cdots & Coef(a_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ Coef(a_{n1}) & Coef(a_{n2}) & \cdots & Coef(a_{nn}) \end{bmatrix}^T$$

Matrix can't be inverted also if

- More rows than columns
- More columns than rows
- Redundant rows/columns ("linearly dependent", "low rank")

Let's consider the case of $\mathbb{R}^2$ with the field $\mathbb{R}$

BASIS 1 $(1, 0)$ and $(1, 1)$: Linear Independence between vectors does not imply that the vectors are orthogonal
BASIS 2 $(1, 0)$ and $(0, 1)$: Orthogonality implies linear independence

Canonical basis of a Vector space is the one whose vectors are the columns of the n×n identity matrix.

Systems of Linear Equations

System of linear equations is important in deep learning because many of the techniques used for training neural networks rely on the solution of such systems.

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

For example, the training of neural networks typically involves optimizing a cost function that measures the error between the output values produced by the network and the desired output values. This *cost function* can often be represented as a system of linear equations, which can be solved using techniques like *gradient descent*. Furthermore, many fundamental operations performed by neural networks, such as matrix multiplication and convolution, are based on the solution of systems of linear equations. Let us consider a system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where all the quantities are known apart from x. If we have a system of m linear equations and n variables, it can be rewritten as:

$$\begin{gathered} \boldsymbol{A}_{1,:}\boldsymbol{x} = b_1 \\ \boldsymbol{A}_{2,:}\boldsymbol{x} = b_2 \\ \cdots \\ \boldsymbol{A}_{m,:}\boldsymbol{x} = b_m \end{gathered}$$

Let us consider a system of m linear equations and n variables:

$$\begin{gathered} \boldsymbol{A}_{1,1}\boldsymbol{x}_1 + \boldsymbol{A}_{1,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{1,n}\boldsymbol{x}_n = b_1 \\ \boldsymbol{A}_{2,1}\boldsymbol{x}_1 + \boldsymbol{A}_{2,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{2,n}\boldsymbol{x}_n = b_2 \\ \cdots \\ \boldsymbol{A}_{m,1}\boldsymbol{x}_1 + \boldsymbol{A}_{m,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{m,n}\boldsymbol{x}_n = b_m \end{gathered}$$

A solution of the linear equation system is any vector of n scalars that satisfies all the equations. In case that all the coefficient b_i are zero, then the system has always a solution that is the one with all the x_j equal to zero.

A linear system of equation can have: no solution, many solutions, exactly one solution (this means multiplication by the matrix is an inevitable function). Solving a system using an inverse (Numerically unstable, buyt useful for abstract analysis).

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

$$\boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

This assumes that the matrix $\boldsymbol{A}$ is invertible (squared and with $det(A)$ not equal to zero). What if the $\boldsymbol{A}$ matrix is square but not invertible or rectangular? Considering again a linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

If A is invertible, then we have that:

$$\boldsymbol{A}\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{A}$$

If it is not true, then for instance $\boldsymbol{A}$ is a $n \times m$ matrix, then we need a candidate matrix $\boldsymbol{G}$ of order $m \times n$ such that

$$\boldsymbol{A}\boldsymbol{G}\boldsymbol{A} = \boldsymbol{A}$$

In this case, $\boldsymbol{x} = \boldsymbol{G}\boldsymbol{b}$ is a solution of the linear system. $\boldsymbol{G}$ is defined as the **generalized inverse**. There are different types of generalized inverse. Later we will talk about **pseudoinverse**.

By considering that:

$$\boldsymbol{A}\boldsymbol{G}\boldsymbol{A} = \boldsymbol{A}$$

And by considering to multiplying both sides of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ by $\boldsymbol{A}\boldsymbol{G}$, we have:

$$(\boldsymbol{A}\boldsymbol{G})\boldsymbol{A}\boldsymbol{x} = (\boldsymbol{A}\boldsymbol{G})\boldsymbol{b}$$

$$(\boldsymbol{A}\boldsymbol{G}\boldsymbol{A})\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}\boldsymbol{G}\boldsymbol{b}$$

Let us consider a system of linear equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

We can have four possibilities

1. $m > n$ the number of equations higher than the number of variables. The system of linear equations is overdetermined. It means that there may not be a unique solution to the system.
2. $m < n$ number of variables higher than the number of equations. The system of linear equations is undetermined. It means it has either no solution or infinitely many solutions.
3. $m = n$ number of variables equals to the number of equations. The system of linear equations is square. It means it has either exactly one solution, no solution or infinitely many solutions.
4. If all of the constant terms are zero, the system of linear equations is homogeneous.

Rank of a Matrix

The **rank** of the matrix $\boldsymbol{A}$ refers to the number of linearly independent rows or columns in the matrix. We refer to it as $rank(A)$.

- Column and row ranks are equal.
- A matrix is said to be of *rank zero* when all of its elements become zero.
- The rank of the matrix is the dimension of the vector space obtained by its columns.
- The rank of a matrix cannot exceed more than the number of its rows or columns.
- The rank of the null matrix is zero.

Let's consider $\boldsymbol{A}$ of size $m \times n$

- $m > n \rightarrow rank(A) \leq n$

- $m = n \rightarrow rank(A) \leq n$ or $rank(A_{\leq} m$
- $m < n \rightarrow rank(A) \leq m$

Rank is thus a measure of the *nondegenerateness* of the system of linear equations and linear transformation encoded by $\boldsymbol{A}$. The rank is useful to get the possible solutions of a linear system. Let considers the complete matrix, achieved by taking both the matrix $\boldsymbol{A}$ and the vector of coefficient $\boldsymbol{b}$.

$$(\boldsymbol{A}|\boldsymbol{b}) = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_1 \end{array}\right]$$

- $rank(A) < rankA|b) \rightarrow$ no solution, system impossible
- $rank(A) = rank(A|b) \rightarrow$ one or infinite solutions
- $rank(A) = rank(A|b) = n \rightarrow$ only one solution
- $rank(A) = rank(A|b) < n \rightarrow$ infinite solutions $(\infty^{n-rank(A)})$

The rank is useful to get the possible solutions of a linear system.

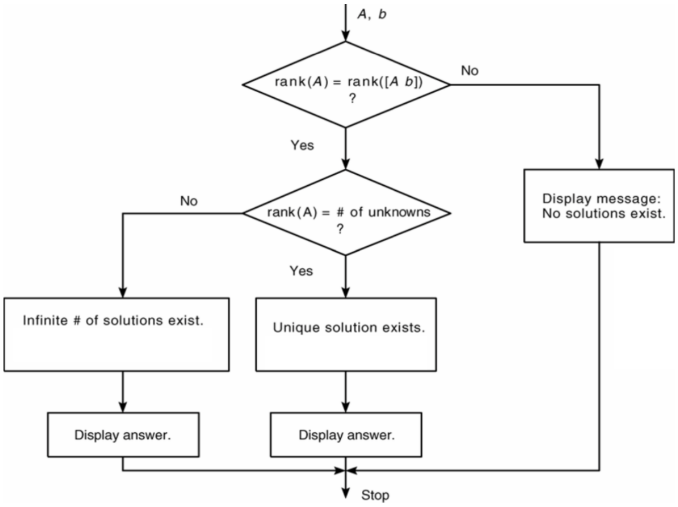


Figure 2.3: Rank schema.

Linear Dependence and Span Formally, a linear combination of some set of vectors $\{\boldsymbol{v}^1, ..., \boldsymbol{v}^{(n)}$ is given by multiplying each vector v(i) by a corresponding scalar coefficient and adding the results:

$$\sum_i ci\boldsymbol{v}^{(i)}$$

In the case of the matrix $\boldsymbol{A}$ and linear system

$$\begin{gathered} \boldsymbol{A}_{1,1}\boldsymbol{x}_1 + \boldsymbol{A}_{1,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{1,n}\boldsymbol{x}_n = b_1 \\ \boldsymbol{A}_{2,1}\boldsymbol{x}_1 + \boldsymbol{A}_{2,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{2,n}\boldsymbol{x}_n = b_2 \\ \cdots \\ \boldsymbol{A}_{m,1}\boldsymbol{x}_1 + \boldsymbol{A}_{m,2}\boldsymbol{x}_2 + ... + \boldsymbol{A}_{m,n}\boldsymbol{x}_n = b_m \end{gathered}$$

In the case of the matrix $\boldsymbol{A}$, the span of the set if vectors, known as the *column space* or the *range* of $\boldsymbol{A}$, is the set of all points obtainable by linear combination of the original vectors. This set of vectors forms a

basis for that subspace, determining whether $\boldsymbol{Ax} = \boldsymbol{b}$ has a solution thus amounts to testing whether $\boldsymbol{b}$ is in the span of the columns of $\boldsymbol{A}$. A set of vectors is **linearly independent** if no vector is the set is a linear combination of the other vectors. If we add a vector to a set that is a non linear combination for the other vectors in the set, the new vector does not add any points to the set's span. Going back to the system $\boldsymbol{Ax} = \boldsymbol{b}$, the matrix must contain at least on set of m linearly independent columns. This condition is both necessary and sufficient to have a solution for every value of b.

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Note that the requirement is for a set to have exactly m linear independent columns, not at least m. No set of m-dimensional vectors can have more than m mutually linearly independent columns, but a matrix with more than m columns may have more than one such set.

In order for the matrix to have an inverse, we additionally need to ensure that $\boldsymbol{Ax} = \boldsymbol{b}$ has at most one solution for each value of b. To do so, we need to ensure that the matrix has at most m columns. Otherwise there is more than one way of parametrizing each solution. Together, this means that the matrix must be *square*, that is, we require that $m = n$ and that all of the columns must be linearly independent. A square matrix with linearly dependent columns is known as **singular**. If A is *not square* or is *square but singular*, it can still be possible to solve the equation. However, we can not use the method of matrix inversion to find the solution.

Special Matrices and Vectors

- *Unit vector*: $\| \boldsymbol{x} \|_2 = 1$
- *Symmetric Matrix*: $\boldsymbol{A} = \boldsymbol{A}^T$
- *Orthogonal matrices are square*: $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I}$
- *Diagonal Matrix*: consists mostly of zeros and have non-zero entries only along the main diagonal. We write $diag(\boldsymbol{v})$ to denote a square diagonal matrix whose diagonal entries are given by the entries of the vector v. Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient. To compute $diag(v)x$, we only need to scale each element x_i by v_i . The inverse exists only if every diagonal entry is nonzero, $diag(\boldsymbol{v})^{-1} = diag([1/v_1, ..., 1/v_n]^T)$

2.1.2 Eigendecomposition

We can decompose *square* matrices in ways that show us information about their functional properties that is not obvious from the representation of the matrix as an array of elements. One of the most widely used kinds of matrix decomposition is called eigendecomposition, in which we decompose a square matrix into a set of eigenvectors and eigenvalues. Not every matrix can be decomposed into eigenvalues and eigenvectors. + Eigenvector and eigenvalue: An eigenvector of a square matrix A is a non-zero vector v such that multiplication by A alters only the scale of v:

$$\boldsymbol{Av} = \lambda \boldsymbol{v}$$

Here we are considering a special linear map (endomorphism) between a vector space V and itself A: $V \to V$. Eigendecomposition of a diagonalizable matrix:

$$\boldsymbol{A} = \boldsymbol{V} diag(\lambda)\, \boldsymbol{V}^{-}1$$

Every real symmetric matrix has a real, orthogonal eigendecomposition:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$$

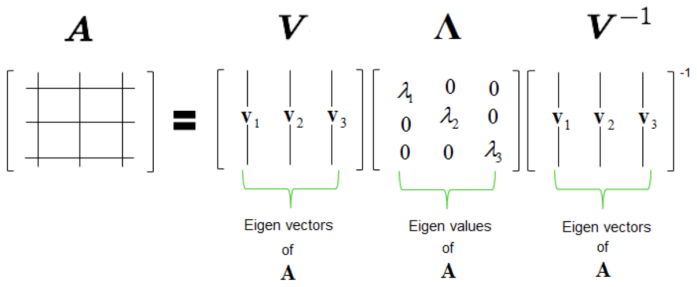


Figure 2.4: Eigendecomposition.

Here, we have a matrix with two orthonormal eigenvectors.

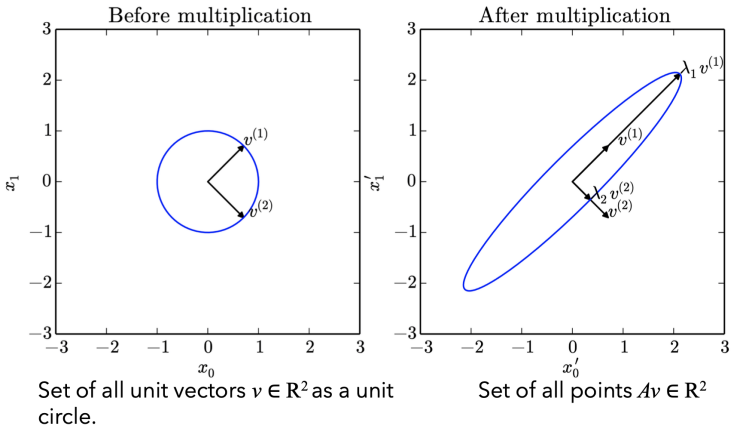


Figure 2.5: Effect of eigenvalues.

While any real symmetric matrix A is guaranteed to have an eigendecomposition, the eigendecomposition may not be unique. If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue, and we could equivalently choose a Q using those eigenvectors instead. By convention, we usually sort the entries of Λ in descending order. Under this convention, the eigendecomposition is unique only if all of the eigenvalues are unique. Eigenvectors of distinct eigenvalues are linearly independent. The maximum number of linearly independent eigenvectors of a matrix A $(n \times n)$ is at most equal to n with distinct eigenvalues. A matrix A is diagonalizable (i.e. similar to a diagonal matrix) if the eigenvectors form a basis.

The eigendecomposition of a matrix tells us many useful facts about the matrix. The matrix is singular if and only if any of the eigenvalues are zero. The eigendecomposition of a real symmetric matrix can also be used to optimize quadratic expressions of the form:

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x} \text{ subject to } \| \boldsymbol{x} \|_2 = 1$$

Whenever x is equal to an eigenvector of $\boldsymbol{A}$, f takes on the value of the corresponding eigenvalue. The *maximum* value of f within the constraint region is the *maximum eigenvalue* and its *minimum* value within the constraint region is the *minimum eigenvalue*. A matrix whose eigenvalues are all positive is called positive definite, while in case they are all positive or zero-valued is called positive semidefinite.

- Positive semidefinite matrix: $\forall \boldsymbol{x}, \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} \geq 0$
- Positive definite matrix: $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = 0$

A positive definite matrix is invertible and its inverse is still a positive definite matrix. The rank of a positive definite matrix is maximum (equal to the minimum dimension). A matrix that is

- **not positive definitive**
- **positive semidefinite**
- **indefinite**

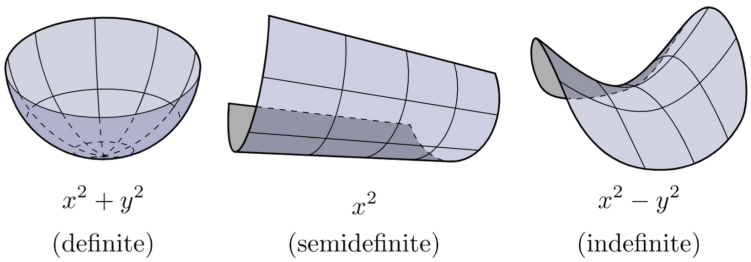


Figure 2.6: Geometric interpretation of positive definite, positive semidefinite or indefinite matrices.

2.2 Singular Value Decomposition

Similar to eigendecomposition but more general, matrix need not be product of three matrices: square. Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix (real numbers).

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$$

Then $\boldsymbol{U}$ is defined to be an $m \times m$ matrix, $\boldsymbol{D}$ to be $m \times n$ matrix, and $\boldsymbol{V}$ to be an $n \times n$ matrix. The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are both defined to be and *orthogonal matrices*. The matrix $\boldsymbol{D}$ is defined to be a *diagonal matrix*. Note that $\boldsymbol{D}$ is not necessarily square. The elements along the diagonal of $\boldsymbol{D}$ are known as the *singular values* of the matrix $\boldsymbol{A}$. The columns of $\boldsymbol{U}$ are known as the *left-singular vectors*. The columns of $\boldsymbol{V}$ are known as the *right-singular vectors*.

Given the SVD $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$. We can actually interpret the singular value decomposition of $\boldsymbol{A}$ in terms of the eigendecomposition of functions of $\boldsymbol{A}$.

- The left-singular vectors of $\boldsymbol{A}$ (columns of $\boldsymbol{U}$ $m \times m$) are the eigenvetors of $\boldsymbol{A}\boldsymbol{A}^T$
- The right-singular vectors of $\boldsymbol{A}$ (columns of $\boldsymbol{V}$ $n \times n$) are the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$
- The non-zero singular values of $\boldsymbol{A}$ (which is the diagonal of $\boldsymbol{D}$) are the square roots of the eigenvalues of $\boldsymbol{A}^T\boldsymbol{A}$
- The same is true for $\boldsymbol{A}\boldsymbol{A}^T$

Note that $\boldsymbol{A}\boldsymbol{A}^T$ is a $m \times m$ matrix while $\boldsymbol{A}^T\boldsymbol{A}$ is a $n \times n$ matrix.

Matrix inversion is not defined for matrices that are not square.

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

If $\boldsymbol{A}$ is taller than it is wide ($m > n$), then it is possible for this equation of have no solution. If $\boldsymbol{A}$ is wider than it is tall ($n > m$), then there could be multiple possible solutions. The **Moore-Penrose pesudoinverse** $\boldsymbol{A}^+$ allows us to make some headway in these cases.

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^T$$

So that is possible to define the following solution:

$$\boldsymbol{x} = \boldsymbol{A}^+\boldsymbol{y}$$

The SVD allows the computation of the pseudoinverse:

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^T$$

where $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ are the singular value decomposition of $\boldsymbol{A}$, and the pseudoinverse $\boldsymbol{D}^+$ of a diagonal matrix $\boldsymbol{D}$ is obtained by taking the reciprocal of its non-zero elements then taking the transpose of the resulting matrix.

$$\boldsymbol{x} = \boldsymbol{A}^+\boldsymbol{y}$$

If the equation has:

- **Exactly one solution**: using the pseudoinverse is the same as the inverse
- **Many solutions** ($n > m$): using the pseudo inverse gives us the solution with the smallest Euclidean norm of x among all passible solutions.
- **No solution** ($m > n$): using the pseudoinverse gives us the x with the smallest error $\| \boldsymbol{A}\boldsymbol{x}^-\boldsymbol{y} \|_2$

2.3 Principal Component Analysis

PCA is a very simple machine learning algorithm. We have m points in $\mathbb{R}^n$ $\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}$. We would like to apply *lossy compression* of these points.

$$\boldsymbol{x}^{(i)} \in \mathbb{R}^n \Rightarrow \boldsymbol{c}^{(i)} \in \mathbb{R}^l$$

If l is smaller than n, it will take less memory to store the code points than the original data. To solve the problem we need an:

- **Encoding function**: $f(\boldsymbol{x}) = \boldsymbol{c}$
- **Decoding function**: $\boldsymbol{x} \approx g(f(\boldsymbol{x}))$

One simple choice of decoding function is the following:

$$g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}, \text{ where } \boldsymbol{D} \in \mathbb{R}^{n \times l}$$

with the condition that D is orthogonal and unit norm and that the distance between the reconstructed points and the original is minimized, L2 norm or alternatively squared L2 norm.

$$\boldsymbol{c}^* = argmin_{\boldsymbol{c}} \| \boldsymbol{x} - g(\boldsymbol{c}) \|_2^2 = argmin_{\boldsymbol{c}} - 2\boldsymbol{x}^T\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^T\boldsymbol{c}$$

We can solve this optimization problem using vector calculus:

$$\nabla_c(-2\boldsymbol{x}^T\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^T\boldsymbol{c}) = \boldsymbol{0}$$
$$-2\boldsymbol{D}^T\boldsymbol{x} + 2\boldsymbol{c} = \boldsymbol{0}$$
$$\boldsymbol{c} = \boldsymbol{D}^T\boldsymbol{x} \rightarrow f(\boldsymbol{x}) = \boldsymbol{D}^T\boldsymbol{x}$$

Using a further matrix multiplication, we can also define the PCA reconstruction operation:

$$r(\boldsymbol{x}) = g(f(\boldsymbol{x})) = \boldsymbol{D}\boldsymbol{D}^T\boldsymbol{x}$$

Next, we need to choose the encoding matrix $\boldsymbol{D}$. To do so, we revisit the idea of minimizing the L^2 distance between inputs and reconstructions. Using the Frobenius norm

$$\boldsymbol{D}^* = arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j}(x_j^{(i)} - r(\boldsymbol{x}^{(i)})_j)^2} \text{ subject to } \boldsymbol{D}^T\boldsymbol{D} = \boldsymbol{I}_l$$

To derive the algorithm to find $\boldsymbol{D}^*$, we start by considering the case where $l = 1$

$$\boldsymbol{d}^* = arg\min_{\boldsymbol{d}} \sum_i \| \boldsymbol{x}^{(i)} - \boldsymbol{d}\boldsymbol{d}^T\boldsymbol{x}^{(i)} \|_2^2 \text{ subject to } \| \boldsymbol{d} \|_2 = 1$$
$$\boldsymbol{d}^* = arg\min_{\boldsymbol{d}} \sum_i \| \boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i)T}\boldsymbol{d}\boldsymbol{d} \|_2^2 \text{ subject to } \| \boldsymbol{d} \|_2 = 1$$

We can now rewrite the problem using a more compact notation as

$$\boldsymbol{d}^* = arg\min_{\boldsymbol{d}} \| \boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^T \|_F^2 \text{ subject to } \| \boldsymbol{d}^T\boldsymbol{d} \|_2 = 1$$
$$\boldsymbol{d}^* = arg\min_{\boldsymbol{d}} Tr((\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^T)^T(\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^T))$$

After several simplifications, we have

$$\boldsymbol{d}^* = arg\max_{\boldsymbol{d}} Tr((\boldsymbol{d}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{d}) \text{ subject to } \boldsymbol{d}^T\boldsymbol{d} = 1$$

This optimization problem may be solved using *eigendecomposition*. Specifically, the optimal d is given by the *eigenvector* of X^TX corresponding to the *largest eigenvalue*. In the general case, where $l > 1$, the matrix $\boldsymbol{D}$ is given by the l eigenvectors corresponding to the *largest eigenvalues*. This may be shown using proof by induction.

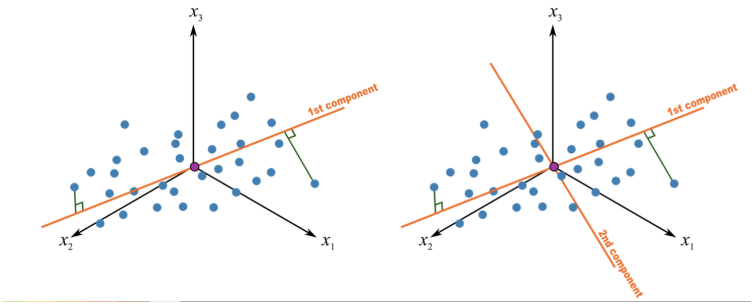


Figure 2.7: Solution of the optimization problem.

The first principal axis is the line through the origin of the data that, once the data is projected onto it, has the greatest variance along it and is said to :maximize the variance". Simply calculating the principal

components isn't dimension reduction at all. Dimension reduction comes into it regarding how many of the principal components we decide to keep in the remainder of our analysis. Principal Components Analysis (PCA) basically means to find and rank all the eigenvalues and eigenvectors of a covariance matrix. This is useful because high-dimensional data (with n features) may have nearly all their variation in a small number of dimensions l, i.e. in the subspace spanned by the eigenvectors of the covariance matrix that have the l largest eigenvalues. If we project the original data into this subspace, we can have a dimension reduction (from n to l) with hopefully little loss of information. We can transform the original data set so that the eigenvectors are the basis vectors and find the new coordinates of the data points with respect to this new basis.

Chapter 3

Machine Learning Basics

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe:

1. Combine a specification of a **dataset**
2. A **cost function**
3. An **optimization** procedure
4. A **model** (ANN)

3.1 Focus on data

Data are typically represented as a vector: $\boldsymbol{x} \in \mathbb{R}^d$, where each entry of the vectors x_j is another feature. Let's suppose we have n variables (features) $x_j = [x_{j,1}, ..., x_{j,d}]$. In case we have $\boldsymbol{X}$ made of m observations x_i:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_m \end{bmatrix} = \begin{bmatrix} x_{1,1}, ..., x_{1,d} \\ x_{2,1}, ..., x_{2,d} \\ \vdots \\ x_{m,1}, ..., x_{m,d} \end{bmatrix}$$

With output mapped from the input using a parametric function f

$$f(\Theta) : x \to y$$

"*A Computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*"

Formally (**supervised learning**) is composed by:

- Input data space $\boldsymbol{X}$
- Output (label) space $\boldsymbol{Y}$
- Unknown function $f(\Theta) : x \to y$
- We have a dataset $\boldsymbol{D} = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ with $x_i \in \boldsymbol{X}, y_i \in \boldsymbol{Y}$
- Finite $\boldsymbol{Y} \Rightarrow$ Classification
- Continuous $\boldsymbol{Y} \Rightarrow$ Regression
- **Targets** are in $\boldsymbol{Y}$:

  - Binary Classification: $\boldsymbol{Y} = \{-1, +1\}$
  - Univariate Regression: $\boldsymbol{Y} \equiv \mathbb{R}$
- A **Loss** (or **cost**) **function** $L : \boldsymbol{Y} \times \boldsymbol{Y} \to \mathbb{R}$
  - L maps decisions to **costs**. $L(\hat{y}, y)$ is the penalty for predicting $\hat{y}$ when the correct answer is y
  - Standard choice for classification: $0/1$ loss
  - Standard choice for regression: $L(\hat{y}, y) = (\hat{y} - y)^2 \to$ **Mean Square Error** (MSE)

To provide the goodness evaluation (it should be done on the entire dataset) we split the dataset $\boldsymbol{D}$ into (at least) 2 sets named **training data** and **Test data**, and additional split of the training in to training data and **Validation data** can be done, but it depends form the situation.

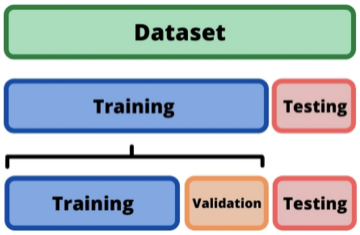
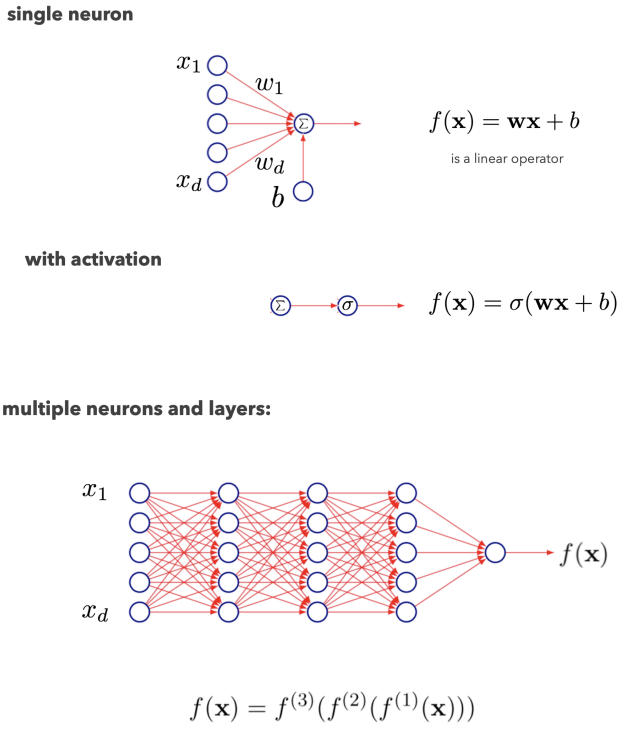


Figure 3.1: Dataset partition.

The model that we are going to use are deep network that are an evolution of the **single neuron** model.



There are several type of data that this type of model can handle as input, in particular lets see the input without labels, that can be: visual data, time series, textual data, generic multi variable data.

3.1.1 Visual Data

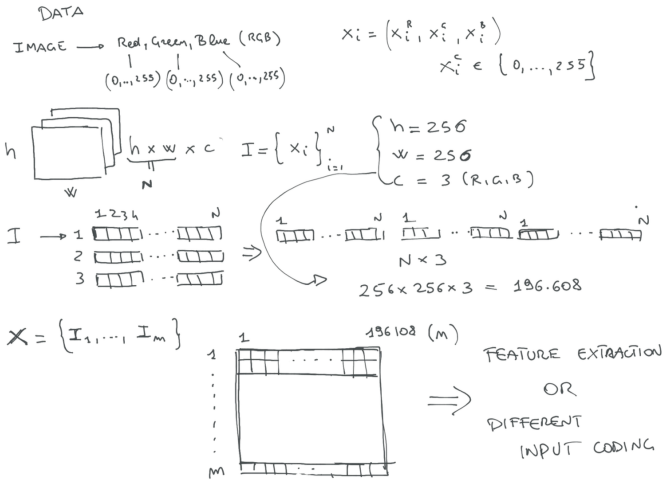
Visual data can be:

- Still images
  - RGB images: three channels images
  - N channels images
- Videos

Computational limitations can pose challenges in machine learning, particularly when working with large inputs such as high-resolution images. While downsampling the image resolution can be one way to address this issue, there are various other techniques and algorithms, such as data parallelism and model parallelism, that can be used to optimize computations on large inputs. In the context of downsampling, it is important to note that the process involves more than just reducing the dimensions of the image; it also involves resampling the image pixels to create a lower-resolution version of the original. For example, downsampling a Full HD image (1920×1080 pixels) might involve reducing the resolution to 960×540 or 480×270 pixels.

When working with RGB images, it is important to note that the dimensions of the image can vary depending on the original resolution. For example, an RGB image with a resolution of 256×256 pixels would have dimensions of $256 \times 256 \times 3$, not $256 \times 256 \times 3$ as stated in the original statement.

Converting an image matrix into a vector is a common technique used in machine learning, but it is important to note that there are various approaches for doing so. For example, the image matrix can be flattened into a 1-dimensional vector or reshaped into a higher-dimensional tensor. The resulting vector dimensions can also vary depending on the approach used.



One of the challenges in working with high-dimensional image data in machine learning is the issue of information loss. When an image is transformed into a vector format, as is often done in machine learning models, information about the relationships between neighboring pixels can be lost. This can result in a loss of *partial coherence* within the image. To address this issue, traditional approaches involved applying dimensionality reduction techniques, such as principal component analysis (PCA), to the image data. However, this approach had limitations in terms of preserving the information about the local relationships between pixels.

A newer approach to preserving local coherence in image data is through the use of local processing techniques, such as receptive fields. **Receptive fields** are a way of representing the local relationships between neighboring pixels within an image. They do this by defining a small region of the image, typically a square or rectangle, that is processed together by a group of neurons in the model. The use of receptive fields can help reduce the number of weights needed in the model, as compared to a fully connected network where every neuron is connected to every pixel in the image. However, fully connected networks may still be used in some cases where the relationships between pixels are not crucial for the task at hand.

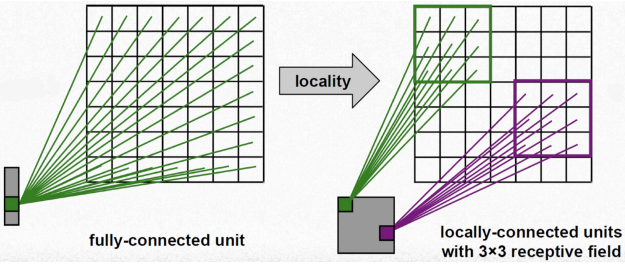


Figure 3.2: Receptive fields

A more advanced approach is to use locally connected units, which are similar to convolutional neural networks (CNNs) but without weight sharing. Locally connected units can be used to represent the local relationships between pixels in a more precise manner than receptive fields. This can result in better performance for certain types of image processing tasks, such as object detection and segmentation. While locally connected units are a powerful technique for preserving local coherence in image data, there are still trade-offs to consider. For example, the size and complexity of the model can be increased by using locally connected units, which can impact the computational demand of the model. Additionally, there may be limitations in terms of the types of relationships that can be captured using this approach.

The number of weights in a neural network plays a crucial role in determining the complexity of the problem being solved. When dealing with multi-channel images, such as RGB+D images that include a depth channel, local correlations still need to be considered. This can be achieved through the use of filters that take into account the local relationships between neighboring pixels across different channels.

In the case of video data, the temporal aspect adds another layer of complexity to the analysis. Videos are essentially a sequence of images captured at a certain frame rate, typically measured in frames per second (fps). To account for this temporal dimension, temporal filters can be used to capture patterns and relationships between frames over time. **Temporal filters** can be thought of as a way of processing video data in a similar way to how spatial filters process image data. Just as spatial filters are used to capture local relationships between pixels in an image, temporal filters capture relationships between frames of a video. This can be especially useful for tasks such as action recognition, where understanding the temporal dynamics of a video is crucial. However, using temporal filters also increases the number of weights in the model, which can further increase the computational complexity. As with image data, trade-offs need to be considered when deciding on the best approach to use when processing video data.

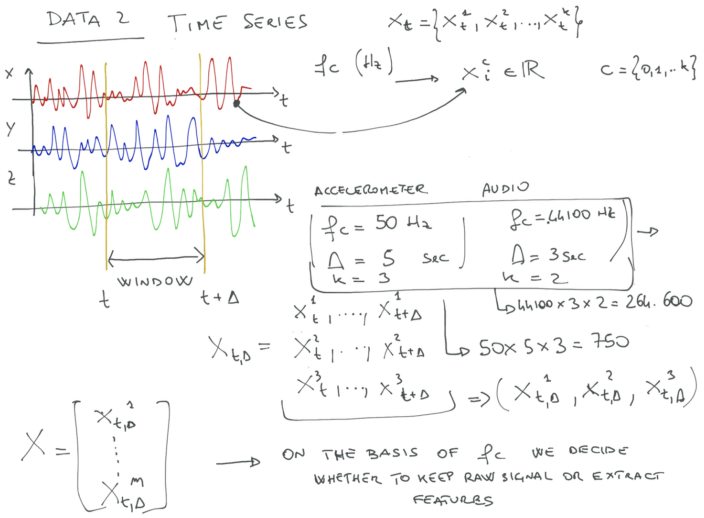
3.1.2 Time Series

Time series data can be:

- Audio signal

- Sensor-based signals
  - Inertial data (acchelerometer and gryscope)
  - ECG, EEG
  - etc...

Time series data, such as audio signals and sensor-based signals, pose a unique challenge for machine learning models. Unlike image data, where local correlations between neighboring pixels are crucial, time series data typically requires capturing temporal dependencies between adjacent data points. In other words, it's important to take into account the ordering of the data points over time.



One approach for processing time series data is to use fully connected layers, where each data point is connected to every neuron in the layer. However, this can quickly become computationally demanding, especially for long time series. On the basis of fully connected layers, it is necessary to decide whether to keep the raw signal or extract features that capture the most relevant aspects of the data.

For audio signal data, feature extraction can be used to capture important characteristics such as frequency and amplitude. Similarly, for sensor-based data such as inertial data from accelerometers and gyroscopes, feature extraction can be used to capture relevant features such as motion patterns and orientation. For ECG and EEG data, feature extraction can be used to capture important patterns in the signals such as heart rate variability and brain wave activity.

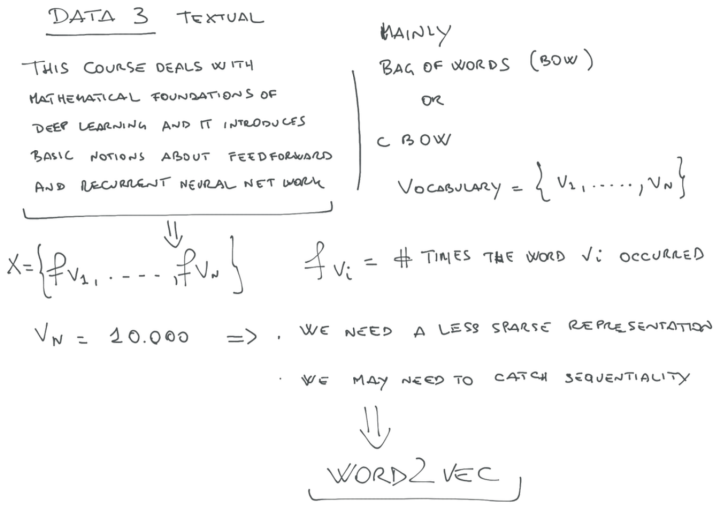
Another approach for processing time series data is to use recurrent neural networks (RNNs), which are designed specifically for sequential data. RNNs have a memory that allows them to maintain information about previous data points as they process new ones. This is particularly useful for tasks such as speech recognition or language modeling, where context and history play a critical role in understanding the meaning of a sequence.

3.1.3 Textual Data

Textual data can be:

- decument
- text-based speech

Textual data is another type of data that requires special consideration in machine learning. Unlike image or sensor data, textual data is comprised of sequences of discrete symbols, such as words or characters. This sequential nature means that the ordering of the symbols is important and must be taken into account.



One common approach for processing textual data is to use a **bag-of-words** (BOW) model, which represents each document as a vector of word frequencies. In this model, the order of the words is ignored and only the frequency of each word is retained. While this approach can be effective for some tasks, it ignores the sequential nature of the data and can lead to a loss of information.

Another approach is to use a **continuous bag-of-words** (CBOW) model, which learns to predict the probability of a target word given its context. This model still ignores the ordering of the words, but it can capture some of the semantic relationships between them.

More recently, word embedding techniques such as **Word2Vec** have become popular for processing textual data. These methods learn dense, low-dimensional representations of words that capture semantic relationships and can be used as input to machine learning models. These embeddings are learned in an unsupervised manner using the context of words in large corpora of text, and can be used to capture sequential information through techniques such as sliding windows.

When working with textual data, it's important to consider the sequential nature of the data and choose a model that can capture this information effectively. BOW models can be effective for some tasks, but may not capture the full meaning of the text. CBOW models can capture some of the semantic relationships between words, but still ignore the ordering of the words. Word embeddings such as Word2Vec can capture both semantic relationships and sequential information, but may require larger amounts of training data and computational resources. The specific approach used will depend on the task at hand and the resources available.

3.2 Cost Function

Most deep learning algorithms involve **optimization** of some sort. Optimization refers to the task of either *minimizing* or *maximizing* some function $f(\boldsymbol{x}, \theta)$ by altering θ. We usually phrase most optimization

problems in terms of minimizing $f(\boldsymbol{x}, \theta)$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\boldsymbol{x}, \theta)$. The function we want to minimize or maximize is called the **objective function** or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**. We might say

$$\theta^* = arg\min_{\theta} f(\boldsymbol{x}, \theta)$$

Let's think at linear functions

$$f(\boldsymbol{x}, \theta) = \theta_0 + \sum_{j=1}^{d} \theta_j x_{ij} = \theta^T \boldsymbol{x}$$

3.3 Optimization

3.3.1 Optimization Problem Types

Convex Optimization

Convex optimization plays a crucial role in the field of deep learning, where it is used extensively to train machine learning models. Deep learning models involve optimizing a large number of parameters, and convex optimization provides a principled and efficient framework for this task. Many of the commonly used optimization algorithms in deep learning, such as stochastic gradient descent and its variants, are based on convex optimization principles. Additionally, convex optimization techniques such as proximal operators and ADMM are also used in deep learning for tasks such as regularization and structured sparsity. In this chapter, we will explore the application of convex optimization in deep learning, including the optimization of neural networks and the use of convex optimization for training generative models such as GANs.

Convex set is $C \subseteq \mathbb{R}^n$ such that $x, y \in C \Rightarrow tx + (1-t)y \in C$ for all $0 \leq t \geq 1$.

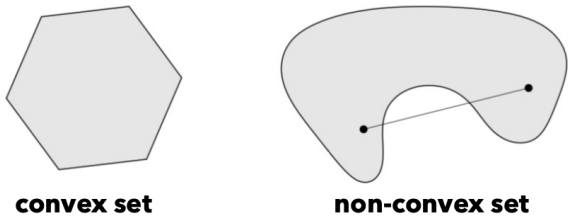


Figure 3.3: Optimization types.

Convex function is $f : \mathbb{R}^n \to \mathbb{R}$ such that $dom(f) \subseteq \mathbb{R}^n$ convex, and $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$ for all $0 \leq t \geq 1$ and all $x, y \in dom(f)$.

Figure 3.4: Convex function.

The **Optimization problem** is defined as it follows

$$\min_{x \in D} f(x)$$
$$\text{subject to } \begin{matrix} g_i(x) \leq 0, i = 1, ..., m \\ h_j(x) = 0, j = 1, ..., r \end{matrix}$$

Here $D = dom(f) \cap \bigcup_{i-1}^{m} dom(g_i) \cup \bigcap_{j=1}^{p} dom(h_j)$, common domain for all the functions. This is a convex optimization problem provided the functions f and $g_i, i = 1, ..., m$ are convex, and $h_j, J = 1, ..., p$ are affine:

$$h_j(x) = a_j^T x + b_j, j = 1, ..., p$$

For convex optimization problems, **local minima are global minima**. Formally, if x is feasible ($x \in D$ and satisfies all constrains) and minimizes f in a local neighborhood,

$$f(x) \leq f(y) \text{ for all feasible } y, \parallel x - y \parallel_2 \leq \rho$$

then

$$f(x) \leq f(y) \text{ for all feasible } y$$

This a very useful fact and will save us a lot of truble.

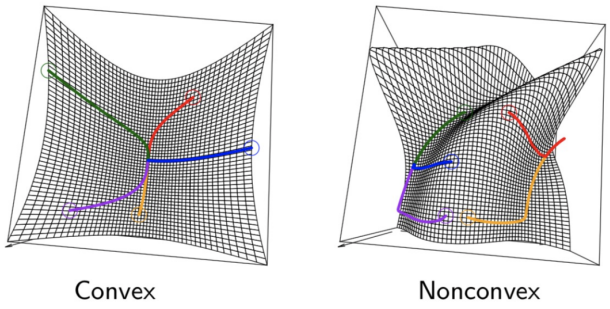


Figure 3.5: Local minima and global minima.

There are several examples of convex function, lets see some of them.

- Univariate functions:
  - Exponential function: e^{ax} is convex for any a over $\mathbb{R}$
  - Power function: x^a is convex for $a \geq 1$ or $a \leq 0$ over $\mathbb{R}_+$ (nonnegative reals)
  - Logarithmic function: $\log x$ is concave over $\mathbb{R}_+$
- Affine function: $a^T x + b$ is both convex and concave
- Quadratic function: $\frac{1}{2} x^t Q x + b^T x + c$ is convex provided that $Q \succeq 0$ (positive semidefinite)
- Least square loss: $\parallel y - Ax \parallel_2^2$ is always convex (since $A^T A$ is always positive semidefinite)
- Norm: $\parallel x \parallel$ is convex for any norm; e.g., l_p norms

Before proceeded we must define what is a **gradient**. Given a vector $\boldsymbol{x} = x_1, ..., x_d$ the gradient of a function is defined as ti follows:

$$\nabla f(\boldsymbol{x}) = \frac{\partial f(\boldsymbol{x})}{\partial x_1}, ..., \frac{\partial f(\boldsymbol{x})}{\partial x_d}$$

where $\frac{\partial f(\boldsymbol{x})}{\partial x_i}$ is the i-th partial derivative of $\boldsymbol{x}$.

Convex functions has three main properties that are the following ones:

- **First-order characterization**: if f is differentiable, then f is convex if and only if $dom(f)$ is convex, and
$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$
for all $x, y \in dom(f)$. Therfore for a differentiable convex function $\nabla f(x) \iff x$ minimizes f
- **Second-order characterization**: if f is twice differentiable, then f is convex if and only if $dom(f)$ is convex, and
$$\nabla^2 f(x) \succeq 0$$
for all $x \in dom(f)$
- **Jensen's inequality**: if f is convex, and X is a random variable supported on $dom(f)$, then
$$f(\mathbb{E}[X] \leq \mathbb{E}[f(X)]$$

Non-Convex Optimization

A non-convex optimization (NCO) is any problem where the objective or any of the constraints are non-convex.

Most problems in deep learning are difficult to express in terms of convex optimization. Convex optimization is used only as a subroutine of some deep learning algorithms. We know that the machine learning models that we introduce next work very well when trained with **gradient descent**. The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function.

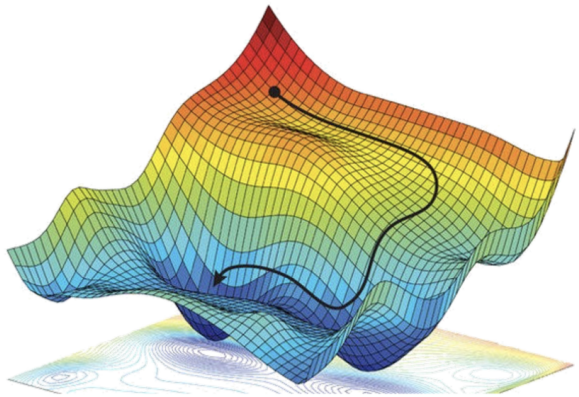


Figure 3.6: Non-convex optimization problem.

3.3.2 Gradient Descent

Before starting talking about the gradient descent is important to recall some crucial aspects.

- Supervised learning is defined in terms of pairs data sample and labels:
$$(x_i, y_i) i = 1, ..., m$$

- A parametric function maps the input data samples into output labels:

$$f(\boldsymbol{x}, \theta) : x \to y$$

- The optimal function is obtained through a training process subject to a cost function over $1, ..., n$ and tested on $n+1, ..., m$ samples:

$$f(x, \theta) = \hat{y}$$

- With cost (loss) to be minimized:

$$L(f(x, \theta), y) = L(\hat{y}, y) \to \theta^* = arg \min_{\theta} L(\hat{y}, y)$$

  Which is usually a non-convex optimization problem solved with convex optimization routines.

The intuition that lead to the discovery of gradient descent is that iterative algorithm that calculates the optimal x^* using step and first derivative until convergence using the following update rule:

$$x_{t+1} = x_t - \alpha \frac{d}{dx} f(x_t)$$

First derivative tells us how to change x in order to make a small improvement in $f(x)$. x_t is the position before the step, $x_t + 1$ is the new position, and α is the **learning rate** (positive scalar).

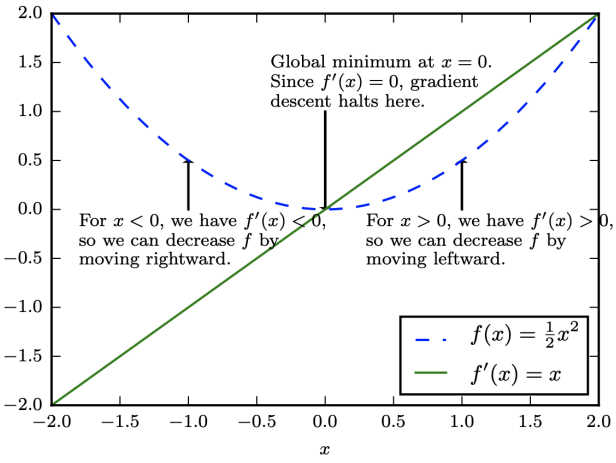


Figure 3.7: Update rule.

The gradient descent guaranteed to find **optimum** for convex functions and a **local optimum** for non-convex ones.

Basically gradient descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent, i.e., the negative gradient of the function at the current point. For a function $f(\theta)$ with parameters θ, the gradient descent algorithm can be described by the following update rule:

where θ_n is the parameter value at iteration n, α is the learning rate (a hyperparameter that determines the step size at each iteration), and $\nabla f(\theta_n)$ is the gradient of the function with respect to the parameters at iteration n.

$$\theta_{i+1} = \theta_i - \alpha \nabla J(\theta_i)$$

To find the optimum for a convex function, gradient descent moves iteratively towards the minimum of the function by following the negative gradient. For a convex function, this process will eventually converge to the global optimum, i.e., the minimum point of the function. However, for a non-convex function, gradient descent can only converge to a local optimum, as there may be multiple local minima with different depths. In this case, the final solution obtained by gradient descent will depend on the initial parameter values and the learning rate.

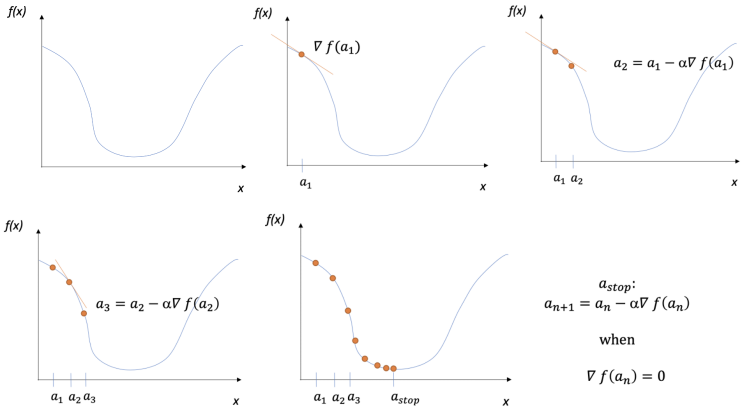


Figure 3.8: Iterations.

It is important to select the right learning rate for gradient descent algorithm, as it can greatly impact the convergence of the algorithm. A learning rate that is too high may cause the algorithm to overshoot the optimal solution and diverge, while a learning rate that is too low may cause the algorithm to converge very slowly.

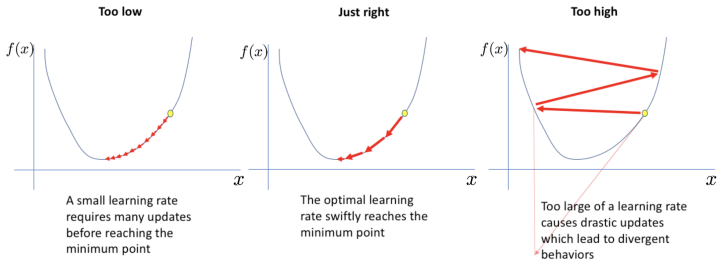


Figure 3.9: Example of different learning rates.

While using the derivative of the cost function, $\frac{d}{dx} f(x_k)$, to determine the convergence of the gradient descent algorithm may be appropriate in certain cases, it is not always reliable. The optimal solution of the cost function occurs where the derivative is equal to zero, i.e., $\frac{d}{dx} f(x_k) = 0$. However, there may be multiple local minima where the derivative is zero, and the algorithm may converge to a suboptimal solution if it gets stuck in one of these minima. For function of multiple variables we deal with gradient (partial derivatives):

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_t - \alpha \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_t)$$

with this stopping criterion:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_t) = 0$$

In terms of stopping criteria, using the derivative of the cost function may not be the ideal choice for determining the convergence of gradient descent. One common stopping criterion is to monitor the difference in the cost function between iterations and stop when the change is below a certain threshold, indicating that the algorithm has converged to a local minimum. Another approach is to set a maximum number of iterations, as the algorithm may not converge within a certain tolerance.

Optimization algorithms may fail to find a global minimum when there are **multiple local minima** or **plateaus present**. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly *low values of the cost function.*

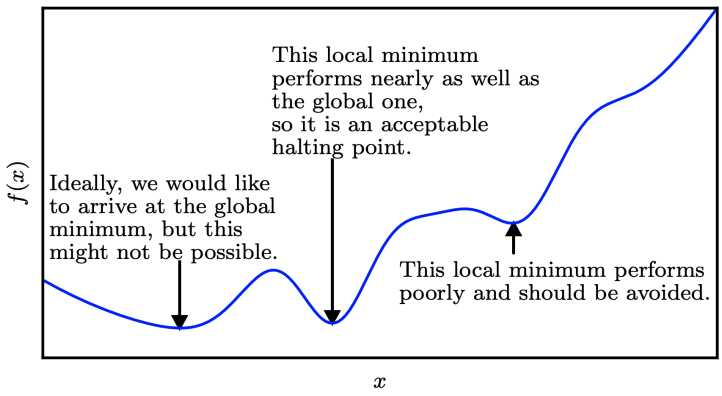


Figure 3.10: Multiple local minima on a function.

There are several variations of the gradient descent algorithm that have been developed to address its limitations and improve its performance.

Batch Gradient Descent Let's consider a function $J(\boldsymbol{x}, \theta)$. Using gradient descent means

$$\theta = \theta - \alpha \nabla_\theta J(\boldsymbol{x}, \theta)$$

As we need to calculate the gradients for the whole dataset to perform just one update, gradient descent can be very slow and is intractable for datasets that don't fit in memory. It is also named for this reason **batch grandient descent**. It is too sensitivity to the number of samples. Gradient descent algorithm can solve optimization problems in polynomial time $O(n^k)$ for some non-negative integer k, where n is the size of the input. In general GD becomes very slow when the amount of data grows.

Stochastic Gradient Descent If we have $m = 100,000$ observations and $n = 100$ variables (features). The sum of squared residuals consists of $100,000$ terms. We need to compute the derivative of this function with respect to each of the features, so we will be doing $100,000 \cdot 100 = 10,000,000$ computations per iteration. If we have 1000 iterations, in effect we have $10,000,000 \cdot 1000 = 10,000,000,000 = 1e^{10}$ computations to complete the algorithm. The solution is to *randomly picks* one data point from the whole data set at each iteration to reduce the computations enormously.

Let's consider a function $J(\boldsymbol{x}, \theta)$. Using gradient descent means

$$\theta = \theta - \alpha \nabla_\theta J(\boldsymbol{x}, \theta)$$

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$

$$\theta = \theta - \alpha \nabla_\theta J(x^{(i)}, \theta)$$

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. By *slowly decreasing the learning rate*, SGD shows the *same convergence behavior as gradient descent*, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Mini Batch SGD **Mini-batch gradient** descent finally takes the best of SGD and GD and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \alpha \nabla_\theta J(x^{(i:i+n)}, \theta)$$

- reduces the variance of the parameter updates, which can lead to more **stable convergence**
- can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient *very efficient*

Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

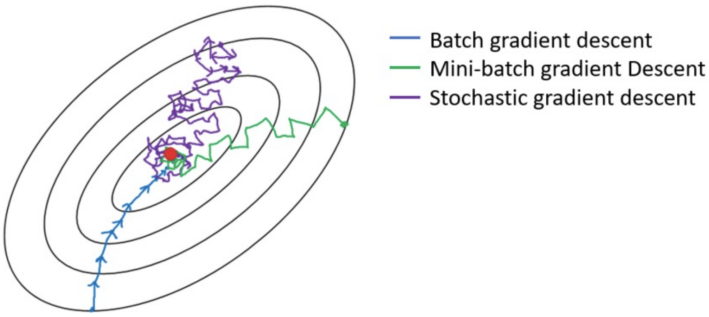


Figure 3.11: Batch GD vs SGD vs Mini barch SGD.

There are several variations of gradient descent that have been developed to address its limitations:

- SGD with Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam
- AdaMax
- Nadam
- AMSGrad

These algorithms use different approaches to update the parameters of the model during the training process, such as adapting the learning rate or using momentum to avoid getting stuck in local minima. Choosing the best optimization algorithm for a given problem is not always straightforward, and often requires trial and error. It is important to consider factors such as the size of the dataset, the complexity of the model, and

the desired level of accuracy when selecting an optimization algorithm. Additionally, some algorithms may perform better than others depending on the specific architecture of the neural network being trained.

3.4 Machine Learning Basics

Let's suppose that the training samples are n instances of pairs (x_i, y_i) drawn i.i.d. from a data distribution $P(x, y)$. Let's consider a parameterized function between the input and the output $f(x, \theta)$. The **risk** of the the classification/regression function f is defined as the **expected loss**:

$$R(\theta) = E[L(f(x, \theta), y)] = \int_{X \times Y} L(f(x, \theta), y)\, dP(x, y)$$

The goal of a learning algorithm is to find a set of parameters for which the *risk is minimum*:

$$\theta^* = arg \min_{\theta} R(\theta)$$

We usually refer to the **empirical risk loss** since the data distribution $P(x, y)$ is not known. The empirical risk loss is defined on the training set (by averaging the loss over the training set):

$$R_{emp}(\theta) = L(f(x, \theta), y) = \frac{1}{p} \sum_{i=1}^{p} L(f(x_i, \theta), y_y)$$

Note that by recalling the Low of Large Number $R_{emp}(\theta) \xrightarrow{p \to \infty} R(\theta)$ If the training set is a representative of the underlying (unknown) distribution P(x, y), the empirical loss is a proxy for the risk

$$\theta^* = arg \min_{\theta} R_{emp}(\theta)$$

We find the optimal θ^* using an *optimization algorithm.*

Example linear function Consider now a linear function $f(\boldsymbol{x}, \theta) = \theta_0 + \sum_{j=1}^{d} \theta_j x_{ij} = \theta^T \boldsymbol{x}$, N hypothesis and $d + 1$ dimensions. The empirical loss function is:

$$L(\theta, (X), \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} L(f(x_i, \theta), y_i)$$

Least squares minimizes empirical loss for squared loss L to find the optimal values

$$\theta^* = [\theta_0, \theta_1, ..., \theta_d]$$

Using least square a necessary condition to minimize L is

$$\nabla_\theta L(\theta, (X), \boldsymbol{y}) = 0$$

that is the gradient in all direction must be set to zero

$$\frac{\partial L(\theta)}{\partial \theta_0}, \frac{\partial L(\theta)}{\partial \theta_1}, ..., \frac{\partial L(\theta)}{\partial \theta_d}$$

Gives us $d + 1$ linear equations in $d + 1$ unknowns $\theta_0, \theta_1, ..., \theta_d$.

- Predictions: $\hat{\boldsymbol{y}} = \boldsymbol{X}\theta$

- Errors: $\boldsymbol{y} - \boldsymbol{X}\theta$

Using vector notation:

$$L(\theta, \boldsymbol{X}) = \frac{1}{N}(\boldsymbol{y} - \boldsymbol{X}\theta)^T(\boldsymbol{y} - \boldsymbol{X}\theta) = (\boldsymbol{y}^T - \theta^T\boldsymbol{X}^T)(\boldsymbol{y} - \boldsymbol{X}\theta)$$

We have:

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{1}{N}\frac{\partial}{\partial \theta}[\boldsymbol{y}^T\boldsymbol{y} - \theta^T\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{X}\theta + \theta^T\boldsymbol{X}^T\boldsymbol{X}\theta] = \frac{1}{N}\frac{\partial}{\partial \theta}[0 - \boldsymbol{X}^T\boldsymbol{y} - (\boldsymbol{y}^T\boldsymbol{X})^T + 2\boldsymbol{X}^T\boldsymbol{X}\theta] = -\frac{2}{N}(\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{X})\theta$$

We have

$$\frac{\partial L(\theta)}{\partial \theta} = -\frac{2}{N}(\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{X})\theta = 0$$
$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\theta \Longleftrightarrow \theta^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Where $\boldsymbol{X}^+ = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ is the **Moore-Penrose** pseudoinverse of $\boldsymbol{X}$. However, using the closed form is impractical, so optimization is required.

The central challenge in machine learning is that we must perform well on new, *previously unseen inputs*, not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error**, to be low as well. Our true goal is to *minimize the loss of the test points.*

$$f^* = arg\min_f \frac{1}{m-n}\sum_{1=n+1}^{m} L(X_i, Y_i, f(X_i))$$

The train and test data are generated by a probability distribution over datasets called the data generating process. We typically make a set of assumptions known collectively as the **i.i.d. assumptions**. These assumptions are that the examples in each dataset are independent from each other, and that the *train set and test set are identically distributed*, drawn from the same probability distribution as each other. One immediate connection we can observe between the training and test error is that the expected training error of a randomly selected model is equal to the expected test error of that model. The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.
2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: underfitting and overfitting.

- **Underfitting** occurs when the model is not able to obtain a sufficiently low error value on the training set.
- **Overfitting** occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its capacity that is its ability to fit a wide variety of functions.

Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set. One way to control the capacity of a learning algorithm is by choosing its hypothesis space, the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

3.4.1 Polynomial Regression

Consider 1D **polynomial** (not linear in x but still linear in θ)

$$f(x) = \theta_0, \theta_1 x, \theta_2 x^2 + ... + \theta_m x^m$$

Consider data drawn from a third order model. If model overfits, i.e. it is too sensitive to data, it will be unstable. We use split training/test or k-fold cross validation.

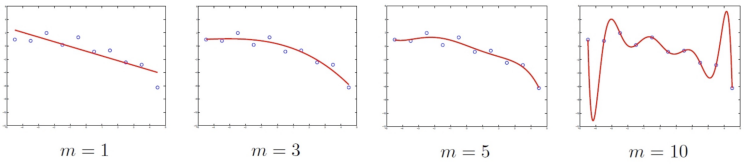


Figure 3.12: Different number of parameters.

By using cross validation we result in two lines, one for training error and one for generalization error. We wont to find the point of **optimal capacity**, which is between the underfitting zone and the overfitting zone. It is basically the point where the two error are at their lowest.

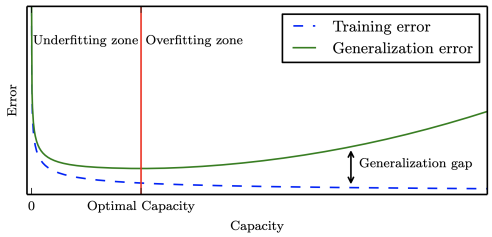


Figure 3.13: Graph for searching the optimal capacity.

Model *complexity is the number of independent parameters* to be fit ("degrees of freedom")

- Complex model ⇒ more sensitive to data ⇒ more likely to overfit
- Simple model ⇒ more rigid ⇒ more likely to underfit

We want to find the model with the right **bias-variance balance**.

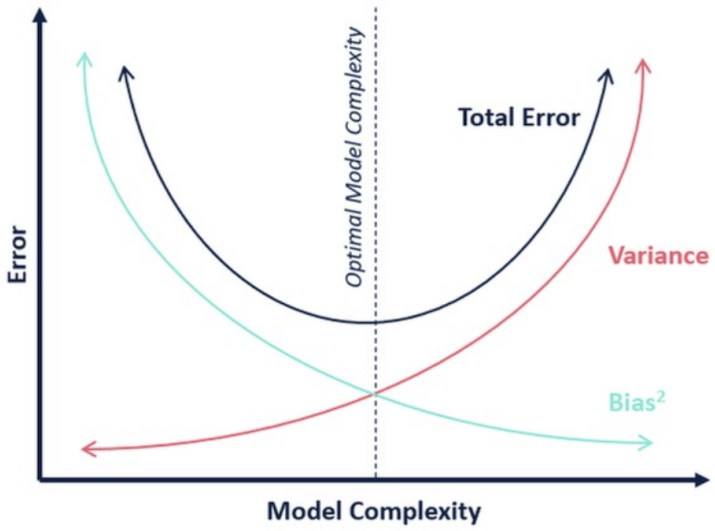


Figure 3.14: Optimal model complexity.

In order to solve this problem we have to:

- **Restrict model** complexity based on amount of data
- Find the **best model** complexity using model search
- **Increase the amount of data**, using augmentation, data generation or new real data (data-centric AI)
- Apply **regularization**

3.4.2 Regularization

The empirical loss function is such that $L(\theta, \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{N} \sum^{i=1\ N} L(f(x_i, \theta), y_i)$. We can modify the training criterion to include **weight decay**, so we end up with a **Regularized risk minimization**

$$\sum_{i=1}^{N} L(f(x_i, \theta), y_i) + \Omega(\theta)$$

For linear regression, the regularization term is for instance the norm of the vector of the weights and weighted by λ.

$$\theta^*_{ridge} = arg \min_{\theta} \sum_{i=1}^{N} L(f(x_i, \theta), y_i) + \lambda \sum_{j=1}^{m} (\theta_j)^2$$

where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights (**Ridge Regression**).

Let's now observe the difference between the results obtained with the empirical loss function and with the regularized empirical loss function.

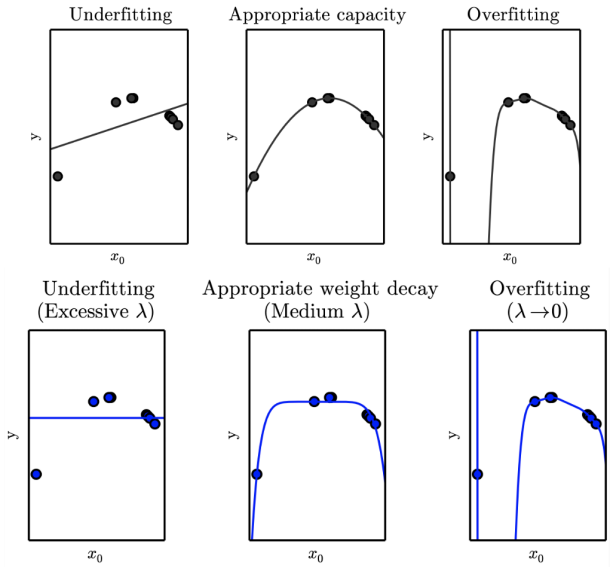


Figure 3.15: Empirical loss function in black vs regularized empirical loss function in blue.

In conclusion nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe:

- combine a specification of a dataset
- a cost function, (weight decay)
- an optimization procedure and (algorithm, learning rate)
- a model

Chapter 4

Feed Forward Networks

4.1 Artificial Neural Networks

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, i.e. models of how learning happens or could happen in the brain. As a result, one of the names that deep learning has gone by is **artificial neural networks** (ANNs). The modern term "deep learning" goes beyond the neuroscientific perspective on the current breed of machine learning models. It appeals to a more general principle of learning multiple levels of composition, which can be applied in machine learning frameworks that are not necessarily neurally inspired.

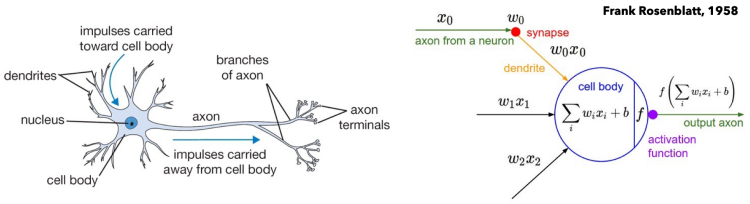


Figure 4.1: How neuron inspired ANN.

Let's now consider a multiple layer network. The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f^{(i)}(x)$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. Because the training data does not show the desired output for each of internal layers, these layers are called **hidden layers**.

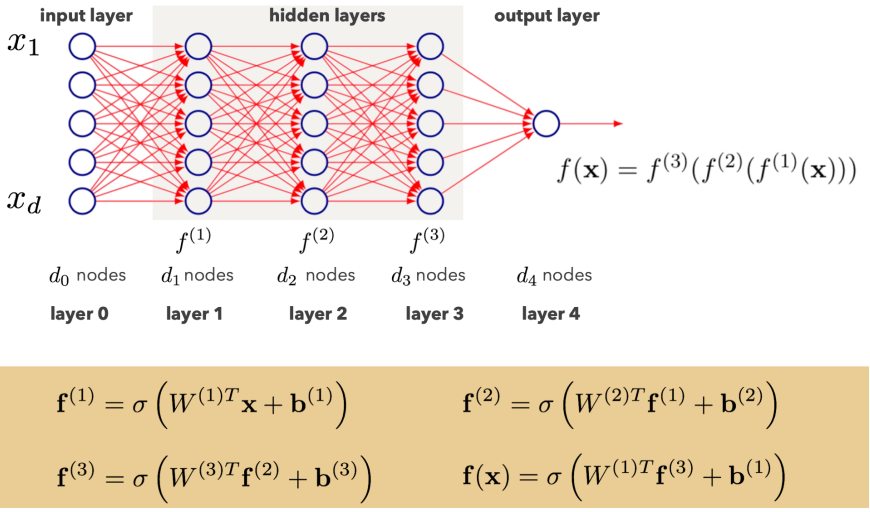


Figure 4.2: Definition of an ANN.

The mapping between input and output is usually non-linear. So to better draw decision boundaries we need to add **non-linearity** within our networks. The mapping between input and output data is often non-linear, which means that a straight line or a simple curve cannot accurately represent the relationship between the two. To better capture this non-linear mapping, neural networks use non-linear activation functions between layers. These **activation functions** introduce non-linearity into the network, allowing it to model more complex relationships between input and output.

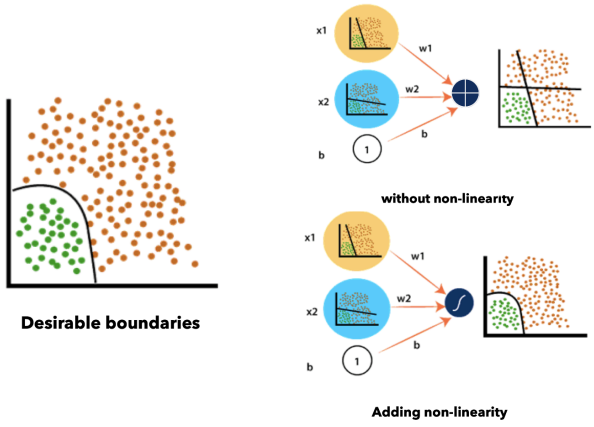


Figure 4.3: Difference made by non-linearity.

Feedforward networks have introduced the concept of a *hidden layer*, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values. The activation function allows the neural network to learn a *non-linear pattern* between inputs and target output variable. Activation function is at the end deciding what is to be fired to the next neuron.

4.1.1 Activation Functions

Activation functions help in keeping the value of the output from the node restricted to a *certain limit*. The input if not restricted to a certain limit can go very high in magnitude especially in case of very deep neural networks that have millions of parameters, thus leading to computational issues.

- **Vanishing gradient problem**: activation functions should not shift the gradient towards zero.
- **Computationally inexpensive**: Since activation functions are applied after every layer millions of times in deep networks, they should be at low computational cost.
- **Differentiable**: layers in the model need to be differentiable or at least differentiable in parts

A neural network will almost always have the same activation function in all hidden layers.

Backpropagation is an intelligent strategy used in neural networks to upgrade weights. Using one upgrade rule for each weight can cause many issues, such as the vanishing gradient problem. Additionally, since the mapping in neural networks is non-linear (i.e., $x = f(y)$ where f is a non-linear function), we have to introduce non-linear activation functions.

The sigmoid function, which saturates the input in the $[0, 1]$ interval, can cause too much stability in the training process. Furthermore, the computational demands of the sigmoid function can be problematic since each neuron in a multilayer perceptron (MLP) has its activation function. Hence, simpler activation functions are needed. Another important consideration is that the activation function must be differentiable (or at least partially differentiable), meaning that it is derivable and continuous.

The rectified linear unit (**ReLU**) function is a commonly used activation function in deep learning. It sets all negative values to zero, making it computationally efficient. An alternative to ReLU is the **leaky ReLU**, where negative values are replaced with small non-zero values. Trainable activation functions can also be composed of several non-trainable activation functions. Remember that trainable means differentiable.

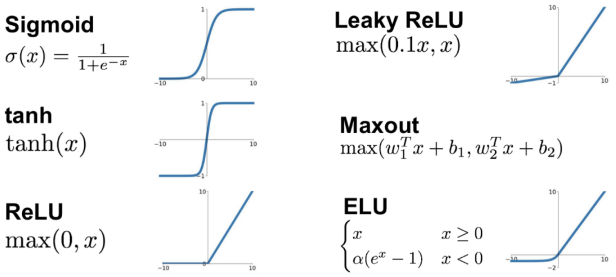


Figure 4.4: Example of sum activation functions.

In summary, there are two possibilities when it comes to activation functions: we either know the activation function or we don't. In the latter case, we have to try different possibilities, and the choice of activation function becomes one of the hyperparameters of the training process. Other hyperparameters include the learning rate, the number of neurons, and the number of layers. Usually, neural architecture search (**NAS**) is used to find suitable hyperparameters.

Usually when we have to choose the activation function the literature suggest the following approach:

- **Multi-layer Perceptron** (MLP): *ReLU* activation function.
- **Convolutional Neural Network** (CNN): *ReLU* activation function.
- **Recurrent Neural Network**: *Tanh* and/or **Sigmoid** activation function.

To sum up neural networks with trainable activation functions can be cheaper in terms of computational complexity since there are fewer parameters to control (as weights with fixed values), on the other hand, the performances in accuracy terms are potentially comparable to deeper architectures without trainable activation functions.

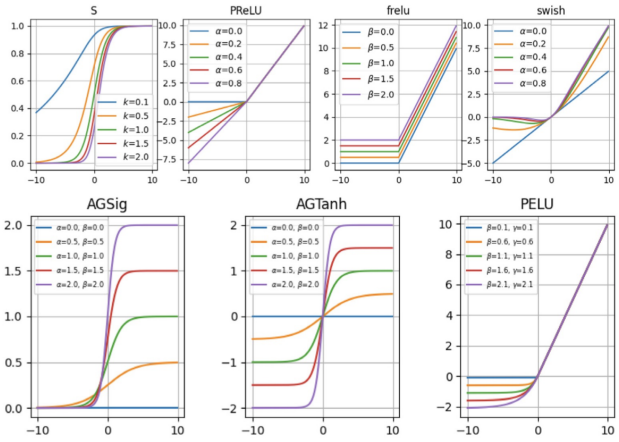


Figure 4.5: Examples of parameterized stander activation functions.

Let's now see a list of possible activation functions, with a brief despription:

1. *Identity*: $f(x) = x$
   This activation function simply returns the input as output, making it useful in certain scenarios where we don't want to change the input values.
2. *Sigmoid*: $f(x) = \frac{1}{1+e^{-x}}$
   The sigmoid function maps the input to a value between 0 and 1, making it useful for binary classification problems. However, it suffers from the vanishing gradient problem when the input is too large or too small.
3. *Tanh*: $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
   The hyperbolic tangent (tanh) function maps the input to a value between -1 and 1, making it useful for classification problems. It also suffers from the vanishing gradient problem.
4. *Arctan*: $f(x) = \tan^{-1}(x)$
   The arctan function maps the input to a value between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, making it useful in some regression problems.
5. *ReLU*: $f(x) = \max(0, x)$
   The rectified linear unit (ReLU) function sets all negative input values to zero and leaves positive values unchanged, making it computationally efficient. However, it suffers from the dying ReLU problem where some neurons can become inactive and never fire again.
6. *Leaky ReLU*: $f(x) = \max(\alpha x, x)$, where α is a small constant
   The leaky ReLU function is similar to ReLU, but instead of setting negative values to zero, it replaces them with a small non-zero value. This helps to prevent the dying ReLU problem.
7. *Randomized ReLU*: $f(x) = \max(0, x + \alpha\mathcal{N}(0, 1))$, where α is a small constant and $\mathcal{N}(0, 1)$ is a normal distribution
   The randomized ReLU function randomly adds noise to the input, making it more robust to noisy data.

8. *Parametric ReLU*: $f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$, where α is a learnable parameter
   The parametric ReLU function allows the negative slope to be learned during training, making it more flexible than leaky ReLU.
9. *Binary*: $f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$
   The binary function maps the input to either 0 or 1, making it useful in binary classification problems.
10. *Exponential Linear Unit* (ELU): $f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$, where α is a small constant
    The ELU function is similar to ReLU, but instead of setting negative values to zero, it smoothly transitions to a negative exponential curve. This helps to prevent the dying ReLU problem.
11. *Soft sign*: $f(x) = \frac{x}{1+|x|}$
    The soft sign function is a smooth approximation of the sign function that maps the input to a value between -1 and 1. It is useful in situations where the input can take on very large or very small values, as it prevents the output from becoming too large. The function is continuous and differentiable everywhere except for x = 0, where it has a vertical tangent.
12. *Inverse Square Root Unit* (ISRU): $f(x) = \frac{x}{\sqrt{1+\alpha x^2}}$, where α is a small constant
    The ISRU function normalizes the input by its square root, making it useful for reducing the impact of large values in the input.
13. *Inverse Square Root Linear Unit* (ISRLU): $f(x) = \begin{cases} \frac{x}{\sqrt{1+\alpha x^2}} & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$, where α is a small constant
    The ISRLU function is similar to ISRU, but it smoothly transitions to a linear function for positive inputs.
14. *Square Non-Linearity* (SQNL): $f(x) = \begin{cases} -1 & \text{if } x < -2 \\ x - \frac{x^2}{4} & \text{if } -2 \leq x < 0 \\ x + \frac{x^2}{4} & \text{if } 0 \leq x < 2 \\ 1 & \text{if } x \geq 2 \end{cases}$
    The SQNL function is a piecewise linear function that transitions smoothly from a negative constant to a quadratic function to a positive constant.
15. *Bipolar ReLU*: $f(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
    The bipolar ReLU function maps the input to either -1 or 1, making it useful in binary classification problems.
16. *Soft Plus*: $f(x) = \ln(1 + e^x)$
    The soft plus function is a smooth approximation of the ReLU function that is differentiable everywhere, making it useful for gradient-based optimization algorithms.

These are just a few examples of the many activation functions that have been proposed for use in neural networks. The choice of activation function depends on the specific problem and the architecture of the neural network being used.

4.2 Back-propagation

In this type of model is crucial the methodology for learning weights. Choosing the best set of weights. We evaluate the cost (loss) function.

$$\Theta^* = arg \min_{\Theta} L(y, \hat{y}; \Theta)$$

The first idea was to randomly perturb one weight for seeing if it improves the performance and in that case then save the change. Unfortunately it was a very inefficient method, because it need to do many passes over a sample set for just one weight change. Computing the gradient is complicated for a neural network, but can still be done efficiently and exactly using a different strategy. Another idea was to perturb all the weights in parallel, and then correlate the performance gain with weight changes. This idea was too hard to implement. Another idea of only perturb activations (since they are fewer) was then proposed, but it is still very inefficient.

Feedforward Propagation is the process of passing input data through an artificial neural network, layer by layer, to obtain an output. The input data is fed into the input layer of the neural network, and then it is processed in each subsequent hidden layer until it reaches the output layer. Each layer applies a set of weights to the input data and uses an activation function to produce an output, which is then passed on to the next layer. The output from the final layer of the network is the predicted output $\hat{y}$ or the estimated target variable.

During Training, the predicted output $\hat{y}$ from the network is compared to the actual output to compute a scalar cost function $L(\Theta)$. The cost function is a measure of how well the neural network is performing on a specific task, and it is usually defined as the difference between the predicted output and the actual output. The goal of training is to minimize this cost function by adjusting the parameters or weights of the neural network.

Backpropagation is a technique used to compute the gradient of the cost function with respect to the weights of the neural network. It allows information to flow backwards from the cost function to the individual layers of the neural network, enabling the network to adjust its weights to minimize the cost function. The backpropagation algorithm uses the chain rule of calculus to calculate the gradient of the cost function with respect to the weights in each layer of the network. This gradient is then used to update the weights of the network using an optimization algorithm, such as gradient descent. The process is repeated iteratively until the cost function is minimized, and the network has learned to make accurate predictions on the input data.

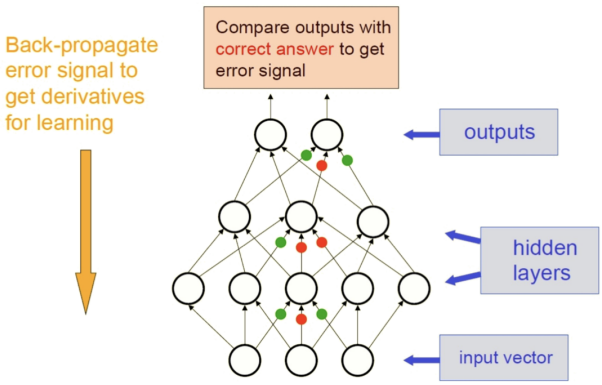


Figure 4.6: Backpropagation schema.

While computing an analytical expression for the gradient is a straightforward task, the numerical evaluation of such an expression can be computationally expensive, especially in complex models such as multi-layer neural networks. The backpropagation algorithm is a widely used method for computing the gradient efficiently in such models. The term "backpropagation" is often misunderstood as referring to the entire learning algorithm for multi-layer neural networks. In reality, backpropagation only refers to the method for computing the gradient. Another algorithm, such as stochastic gradient descent, is typically used to perform the actual learning using this gradient.

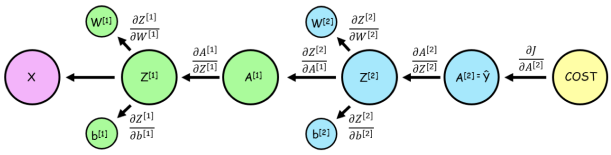
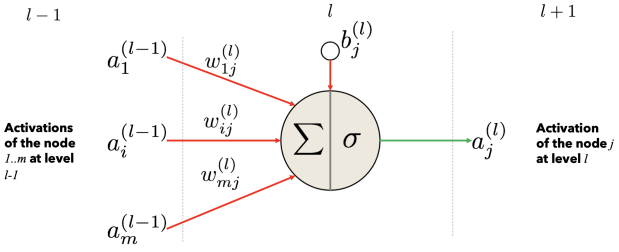


Figure 4.7: Information propagate back form the output to the input.

In the process of training a neural network, we often encounter **hidden units** whose functions may not be immediately apparent. However, by computing the rate at which the error changes as we modify a hidden unit's activity, we can begin to understand how these units are contributing to the overall performance of the network. To accomplish this, we use error derivatives with respect to hidden activities. Each hidden unit has the potential to affect many output units, and can have a unique impact on the overall error. The key is to combine these individual effects in a way that allows us to efficiently compute error derivatives for each hidden unit. Once we have these error derivatives for hidden activities, we can easily compute the error derivatives for the weights that connect these hidden units to the rest of the network. This allows us to iteratively adjust the weights until we achieve the desired level of accuracy.

Lets now see how it act on a single neuron, that simulate the level l of the network and neuron (node) j.

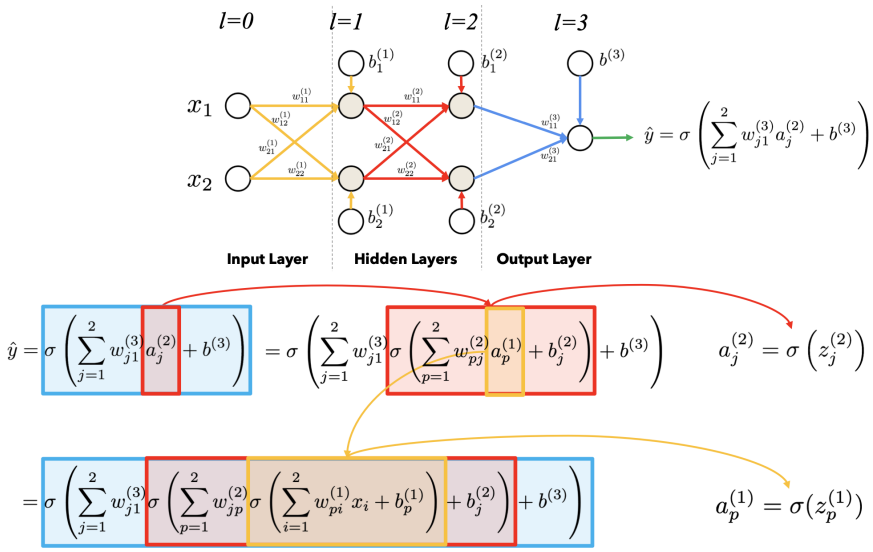


Where:

$$x_j^{(l)} = \sum_{i=1}^{m} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = \sigma(z_j^{(l)}) = \sigma(\sum_{i=1}^{m} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)})$$

Now let's the above example in a network with two input neuron and three layers.



Considering a network of L layers, the parameters of the network are:

$$\boldsymbol{W} = \{b_1^{(1)}, ..., b_{d_1}^{(1)}, ..., b_1^{(L)}, ...b_{d_L}^{(L)}, \boldsymbol{w}_{1:}^{(1)}, ..., \boldsymbol{w}_{d_1:}^{(1)}, ..., \boldsymbol{w}_{1:}^{(L)}, ..., \boldsymbol{w}_{d_L:}^{(L)}\} = \{\boldsymbol{b}^1, ..., \boldsymbol{b}^L, \boldsymbol{W}^{(1)}, ..., \boldsymbol{W}^{(L)}\}$$

4.2.1 Gradient Optimization

Using **RMSE** as loss (cost) function and mini batch **SGD** with m samples

$$L(y, \hat{y}) = \frac{1}{2^*N} \sum_{n=1}^{N} \parallel y_n - \hat{y}_n \parallel^2 \rightarrow L(y, \hat{y}) = \frac{1}{2m} \sum_{n=k+1}^{k+m} \parallel y_n - \hat{y}_n \parallel^2$$

with the following update rule

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(f(\boldsymbol{x}^{(k+1:k+m)}; \boldsymbol{w}), y^{(k+1:k+m))}$$

for each weight and bias at layer l an update rule should be calcualted

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial L}{\partial w_{ij}^{(l)}}$$
$$b_j^{(l)} = b_j^{(l)} - \alpha \frac{\partial L}{\partial b_j^{(l)}}$$

The chain rule is a fundamental principle used to compute the gradients of a composite function with respect to its inputs. Specifically, it allows us to efficiently compute the derivative of the error with respect to each weight in the neural network. The chain rule in calculus states that for a composite function $f(g(x))$, the derivative can be expressed as:

$$(f(g(x)))' = f'(g(x)) \cdot g'(x)$$

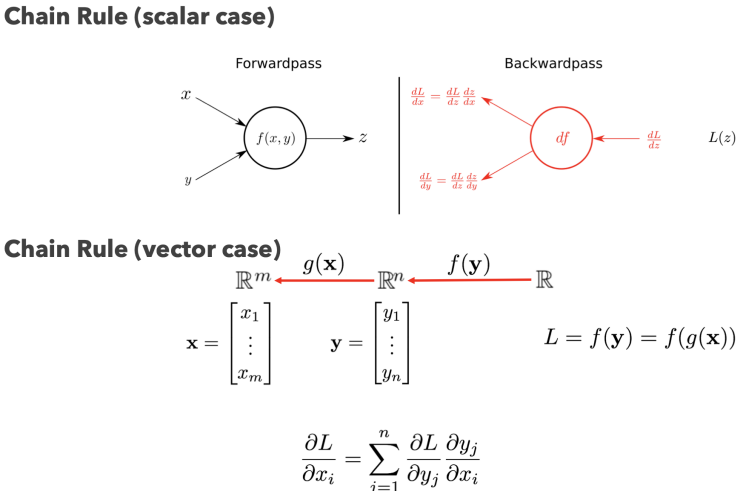


Figure 4.8: Chain rule in the vector and scalar cases.

We will now see how the gradient optimization work inside a network.

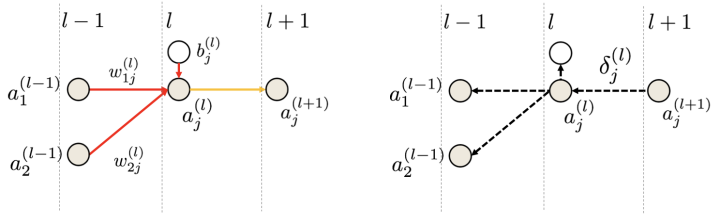


Figure 4.9: Example structure.

The chain rule for the weight i of the node $j(w_{ij})$ and the bias $j(b_j)$ at layer l:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} \qquad \frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial b_j^{(l)}}$$

And considering that $a_j^{(l)} = \sigma(z_j^{(l)})$ we have:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \qquad \frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial b_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}$$

- **Weight Update**:
  The **update rule** for the **weight** i of the node j at layer l:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial L}{\partial w_{ij}^{(l)}} \qquad a_j^{(l)} = \sigma(\sum_{i=1}^{2} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}) = \sigma(z_j^{(l)})$$

  By applying the **chain rule** and calculating the derivative:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}$$

So:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}$$

Where:

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \frac{\partial \sigma(z_j^{(l)})}{\partial z_j^{(l)}} = \sigma'(z_j^{(l)}) \qquad \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial [\sum_{i=1}^{2} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}]}{\partial w_{ij}^{(l)}} = a_i^{(l-1)}$$

And this result in:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) a_i^{(l-1)}$$

The gradient update w.r.t. the weight i of the node $j(w_{ij})$ at layer l is given by:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) a_i^{(l-1)} = \delta_j^{(l)} a_i^{(l-1)}$$

Where $\frac{\partial L}{\partial a_j^{(l)}}$ measures how fast the cost is changing as a function of the j-th output activation, $\sigma'(z_j^{(l)})$ measures how fast the activation σ is changing at $z_j^{(l)}$, and $\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)})$ is the **Local Gradient**.

- **Bias Update**:
  The **update rule** for the **bias** i of the node j at layer l:

$$b_j^{(l)} = b_j^{(l)} - \alpha \frac{\partial L}{\partial b_j^{(l)}} \qquad a_j^{(l)} = \sigma(\sum_{i=1}^{2} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}) = \sigma(z_j^{(l)})$$

  By applying the chain rule and calculating the derivative:

$$\frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) = \delta_j^{(l)}$$

  Where $\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)})$ is the **Local Gradient**.

We will now present an example of gradient optimization at output layer.

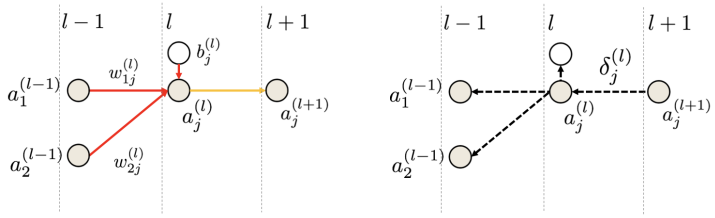


Figure 4.10: Same example structure as before.

Let's consider the output layer (single node $j = 1$) at $l = L = 3$ and the weight i:

$$\frac{\partial L}{\partial w_{i1}^{(3)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial w_{i1}^{(3)}} = \frac{\partial L}{\partial \hat{y}} \sigma'(z^{(3)}) a_i^{(2)} = \delta_{\hat{y}}^{(3)} a_i^{(2)}$$

Remember that:

$$\hat{y} = \sigma(\sum_{j=1}^{2} w_{j1}^{(3)} a_j^{(2)} + b^{(3)})$$

and

$$L(y, \hat{y}) = \frac{1}{2m} \sum_{n=k+1} k + m \parallel y - \hat{y} \parallel^2$$

So it result that:

$$\delta_{\hat{y}}^{(3)} = \frac{\partial L}{\partial \hat{y}} \sigma'(z^{(3)}) = \frac{1}{m} \sum_{n=k+1}^{k+m} (y_n - \hat{y}_n) \sigma'(z^{(3)})$$

4.2.2 Local Gradient

Let's begin by looking at the local gradient at output layer L.

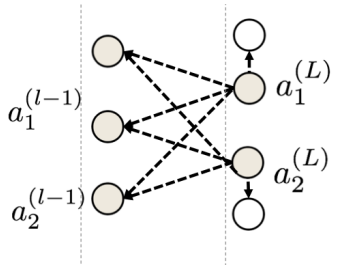


Figure 4.11: Local gradient schema.

In this case we have that:

$$\delta^{(L)} = \begin{bmatrix} \delta_1^{(L)} \\ \vdots \\ \delta_{dL}^{(L)} \end{bmatrix} \qquad \nabla_{(a)} L = \begin{bmatrix} \frac{\partial L}{\partial a_1^{(L)}} \\ \vdots \\ \frac{\partial L}{\partial a_{dL}^{(L)}} \end{bmatrix} \qquad \sigma'(z^{(L)}) = \begin{bmatrix} \sigma'(z_1^{(L)}) \\ \vdots \\ \sigma'(z_{dL}^{(L)}) \end{bmatrix}$$

Where:

$$\delta_1^{(L)} = \frac{\partial L}{\partial a_1^(L)} \sigma'(z_1^{(L)})$$

This means that:

$$\delta^{(L)} = \nabla_a L \odot \sigma'(z^{(L)})$$

Let's now see the local gradient on a single node.

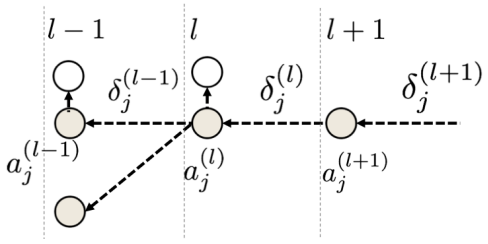


Figure 4.12: Local gradient on a single node schema.

The calculation of $\delta_j^{(l)}$ is such that:

$$\delta_j^{(l)} = \delta_j^{(l+1)} w_{ij}^{(l+1)} \sigma'(z_j^{(l)})$$

Proof. We want to demonstrate that:

$$\delta_j^{(l)} = \delta_j^{(l+1)} w_{ij}^{(l+1)} \sigma'(z_j^{(l)})$$

Let's consider a node j of the hidden layer at l and the weight i:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) a_i^{(l-1)} = \delta_j^{(l)} a_i^{(l-1)}$$

Where:

$$\frac{\partial L}{\partial a_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial a_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial a_j^{(l)}}$$
$$= \frac{\partial L}{\partial a_j^{(l+1)}} \sigma'(z_j^{(l+1)}) \frac{\partial z_j^{(l+1)}}{\partial a_j^{(l)}} = \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial a_j^{(l)}} = \delta_j^{(l+1)} w_{ij}^{(l+1)}$$

So we have that $\frac{\partial L}{\partial a_j^{(l)}} = \delta_j^{(l+1)} w_{ij}^{(l+1)}$. From the first equation we, by simplifying for $a_i^{(l-1)}$, that:

$$\frac{\partial L}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) = \delta_j^{(l)}$$

Now replacing it in the first one we have that:

$$\delta_j^{(l+1)} w_{ij}^{(l+1)} \sigma'(z_j^{(l)}) = \delta_j^{(l)}$$

Finally it is proved. □

So now looking at all the layers we know how the local gradients work.

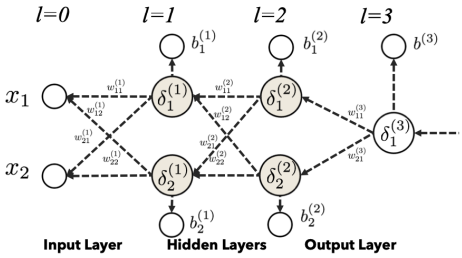


Figure 4.13: Local gradient on a network.

So we have that:

$$\delta^{(l)} = \begin{bmatrix} \delta_1^{(L)} \\ \vdots \\ \delta_{dl}^{(L)} \end{bmatrix} \qquad \sigma'(z^{(l)}) = \begin{bmatrix} \sigma'(z_1^{(l)}) \\ \vdots \\ \sigma'(z_{dl}^{(l)}) \end{bmatrix} \qquad \boldsymbol{w}^{(l+1)} = \begin{bmatrix} w_{1,1}, \cdots, w_{1,d_{l+1}} \\ \vdots \\ w_{d_l,1}, \cdots, w_{d_l,d_{l+1}} \end{bmatrix}$$

Where:

$$\delta_j^{(l)} = \delta_j^{(l+1)} w_{ij}^{(l+1)} \sigma'(z_j^{(l)})$$

This means that:

$$\delta^{(l)} = (\boldsymbol{w}^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

Let's now recall the **general equations** for gradient update that we have seen until this moment.

- **Local Gradient** at output level L: $\delta^{(L)} = \nabla_a L \odot \sigma'(z^{(L)})$
- **Local Gradient** at level l: $\delta^{(l)} = (\boldsymbol{w}^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$
- **Weight** i **update** at node j at level l: $\frac{\partial L}{\partial b_j^{(l)}} = a_i^{(l-1)} \delta_j^{(l)}$
- **Bias update** at node j at level l: $\frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)}$

4.2.3 Vanishing Gradient

In deep neural networks, the vanishing gradient problem is a common issue that arises during back-propagation, where the gradient of the cost function with respect to the weights becomes increasingly small as it is propagated through the layers. This is usually true that $\frac{\partial L}{\partial w_{ij}^{(l)}} \gg \frac{\partial L}{\partial w_{ij}^{(l-1)}}$, i.e., the lower you get in the network, the more the gradient vanishes. The cause of the vanishing gradient problem can be traced back to the chain rule of differentiation. Specifically, the gradient of the cost function with respect to the weights in layer $(l-1)$ is calculated as:

$$\frac{\partial L}{\partial w_i j^{(l-1)}} = \delta_j^{(l-1)} a_i^(l-2) = \delta_j^{(l)} w_{ij}^{(l)} \sigma'(z_{ij}^{(l-1)}) a_i^(l-2)$$

where $\delta_j^{(l)}$ is the error term in neuron j of layer l, $a_i^{(l-2)}$ is the activation of neuron i in layer $(l-2)$, $w_{ij}^{(l)}$ is the weight connecting neuron i in layer $(l-2)$ to neuron j in layer $(l-1)$, and $\sigma'(z_{ij}^{(l-1)})$ is the derivative of the activation function with respect to its input. The problem is that if any of these terms are small, the resulting gradient will be even smaller, making it difficult for the weights in earlier layers to be updated. This can lead to slow convergence or even stagnation in the training process. Therefore, various techniques have been developed to alleviate the vanishing gradient problem, such as weight initialization, using non-saturating activation functions, and batch normalization.

Back-propagation is a widely used algorithm for training deep neural networks. However, it becomes increasingly difficult to use for multiple hidden layers due to the vanishing gradient problem. The gradient becomes smaller and smaller as it propagates through the layers, which makes it difficult to update the weights in the earlier layers of the network. To overcome this problem, unsupervised pretraining with Restricted Boltzmann Machines (Hinton et al., 2006) was introduced. This method helped to train deep neural networks with multiple hidden layers by first pretraining the network in an unsupervised manner before fine-tuning it with labeled data. This technique was successful, but it required a large amount of computational resources. Later, it was found that deep neural networks can also be successfully trained by using many labeled data (e.g., now well possible for images), training "longer" (possible with GPUs), better weight initialization (new methods were developed such as Xavier), and regularization with "dropout." Sometimes, the use of rectified linear units $\max(0, x)$ or $\log(1 + ex)$ can also improve things.

4.2.4 Example

We will now present an example of how the back propagation methodology presented so far actually work in practice.

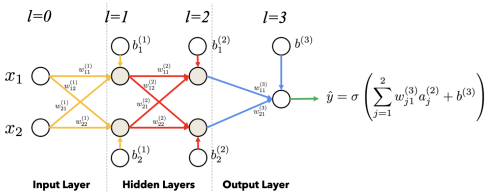


Figure 4.14: Three layer network.

Let's consider the case where $\sigma(x) = x$, that is $\sigma'(x) = 1$. Let's also consider a regression task with *MSE Loss*. Let's assume the use of *minibatch SGD* with $m = 3$ and a learning rate $a = 0.01$.

The equations to compute the prediction and all the activations are as follows:

$$\text{Layer 1} \begin{cases} a_1^{(1)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)} \\ a_2^{(1)} = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)} \end{cases}$$

$$\text{Layer 2} \begin{cases} a_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{21}^{(2)} a_2^{(1)} + b_1^{(2)} \\ a_2^{(2)} = w_{12}^{(2)} a_1^{(1)} + w_{22}^{(1)} a_2^{(1)} + b_2^{(2)} \end{cases}$$

$$\text{Layer 3} \begin{cases} \hat{y} = w_{11}^{(3)} a_1^{(2)} + w_{21}^{(3)} a_2^{(2)} + b_1^{(3)} \end{cases}$$

Let's set initial weights and biases to 0.5

$$w_{11}^{(1)} = w_{12}^{(1)} = w_{21}^{(1)} = w_{22}^{(1)} = w_{11}^{(2)} = w_{12}^{(2)} = w_{21}^{(2)} = w_{22}^{(2)} = w_{11}^{(3)} = w_{21}^{(3)} = 0.5$$

$$b_1^{(1)} = b_2^{(1)} = b_1^{(2)} = b_2^{(2)} = b_1^{(3)} = 0.5$$

The update rule for each weight i at node j and level l of the network after the processing of the m samples from the each minibatch is as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{1}{m} \sum_{k=1}^{m} \frac{\partial L(y_k, \hat{y}_k)}{\partial w_{ij}^{(l)}} \qquad b_j^{(l)} = b_j^{(l)} - \alpha \frac{1}{m} \sum_{k=1}^{m} \frac{\partial L(y_k, \hat{y}_k)}{\partial b_j^{(l)}}$$

$$\frac{\partial L(y_k, \hat{y}_k)}{\partial w_{ij}^{(l)}} = a_{i,k}^{(l-1)} \delta_{j,k}^{(l)} \qquad \frac{\partial L(y_k, \hat{y}_k)}{\partial b_j^{(l)}} = \delta_{j,k}^{(l)}$$

To calculate the gradient updates we need to calculate the following:

$$\delta_{1,k}^{(3)} = (y_k - \hat{y}_k)$$

$$\delta_{1,k}^{(2)} = w_{11}^{(3)} \delta_{1,k}^{(3)} \qquad \delta_{2,k}^{(2)} = w_{21}^{(3)} \delta_{1,k}^{(3)}$$

$$\delta_{1,k}^{(1)} = w_{11}^{(2)} \delta_{1,k}^{(2)} + w_{12}^{(2)} \delta_{2,k}^{(2)} \qquad \delta_{2,k}^{(1)} = w_{21}^{(2)} \delta_{1,k}^{(2)} + w_{22}^{(2)} \delta_{2,k}^{(2)}$$

Given x_1 and x_2 as input samples (with mini-batch $m = 3$), we have

k	x_1	x_2	$a_1^{(1)}$	$a_2^{(1)}$	$a_1^{(2)}$	$a_2^{(2)}$	$\hat{y}$	y	$\delta_1^{(3)}$	$\delta_1^{(2)}$	$\delta_2^{(2)}$	$\delta_1^{(1)}$	$\delta_2^{(1)}$	$a_1^{(2)}\delta_1^{(3)}$	$a_2^{(2)}\delta_1^{(3)}$	$a_1^{(1)}\delta_1^{(2)}$	$a_1^{(1)}\delta_2^{(2)}$	$a_2^{(1)}\delta_1^{(2)}$	$a_2^{(1)}\delta_2^{(2)}$	$x_1\delta_1^{(1)}$	$x_2\delta_1^{(1)}$	$x_1\delta_2^{(1)}$	$x_2\delta_2^{(1)}$
1	0.1	0.2	0.65	0.65	1.15	1.15	1.65	2	0.35	0.175	0.175	0.175	0.175	0.4025	0.4025	0.114	0.114	0.114	0.114	0.0175	0.0175	0.035	0.035
2	0.3	0.3	0.8	0.8	1.3	1.3	1.8	1.9	0.1	0.05	0.05	0.05	0.05	0.13	0.13	0.04	0.04	0.04	0.04	0.015	0.015	0.015	0.015
3	0.4	0.4	0.9	0.9	1.4	1.4	1.9	1.8	-0.1	-0.05	-0.05	-0.05	-0.05	-0.14	-0.14	-0.045	-0.045	-0.045	-0.045	-0.02	-0.02	-0.02	-0.02
μ						**AVERAGE**			0.117	0.058	0.058	0.058	0.058	0.3925	0.3925	0.037	0.037	0.037	0.037	0.0042	0.0042	0.01	0.01

Now we can proceed with the update.

$$
\begin{aligned}
w_{11}^{(1)} &= 0.5 - 0.01(0.0042) = 0.499... = w_{12}^{(1)} \\
w_{21}^{(1)} &= 0.5 - 0.01(0.01) = 0.499... = w_{22}^{(1)} \\
w_{11}^{(2)} &= 0.5 - 0.01(0.0037) = 0.499.. = w_{12}^{(2)} = w_{21}^{(2)} = w_{22}^{(2)} \\
w_{11}^{(3)} &= 0.5 - 0.01(0.3925) = 0.496... = w_{21}^{(3)} \\
b_1^{(1)} &= 0.5 - 0.01(0.0058) = 0.4994... = b_2^{(1)} \\
b_1^{(2)} &= 0.5 - 0.01(0.0058) = 0.4994... = b_2^{(2)} \\
b_1^{(3)} &= 0.5 - 0.01(1.227) = 0.4988... = b_2^{(3)}
\end{aligned}
$$

After this update we need to recompute the prediction for a new set of inputs and so new errors.

In the given example, the absence of an activation function reveals the occurrence of the vanishing gradient problem. When the gradient becomes too small during backpropagation, it can prevent the earlier layers from learning effectively, thus inhibiting the network's overall performance. Fortunately, several methods exist to circumvent this problem, such as utilizing linear units or modifying the learning rate in specific layers. For instance, we could choose to update only the weights in the last layers, effectively fixing the learning rate to zero in some layers. This approach could allow the earlier layers to preserve their initial weights, which could ultimately enhance the network's learning and convergence. Overall, understanding the vanishing gradient problem and adopting effective techniques to mitigate it is crucial for developing high-performing neural networks.