Loss functions & training

Course of Machine Learning
Master Degree in Computer Science
University of Rome "Tor Vergata"
a.a. 2024-2025

Giorgio Gambosi

Loss function

In general, the loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ measures the **cost** of referring to y instead of the correct value t for any subsequent action, where y and t are elements of the target space.

In supervised learning, it provides a measure of the quality of the prediction returned by the prediction function h:

$$\mathcal{R}(\mathbf{x}, y) = L(h(\mathbf{x}), y)$$

It is a fundamental component of the empirical risk, which is the average value of the loss function applied to all predicted value-target value pairs in the training set $\mathcal{T}$:

$$\overline{\mathcal{R}}_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|} \sum_{(x,t) \in \mathcal{T}} L(h(x), t)$$

This gives a measure of the quality of the predictions made by h, at least with respect to the available data (the training set).

During the training phase, the empirical risk is minimized with respect to the prediction function h, specifically with regard to the set of parameters $\boldsymbol{\theta}$ that define the parametric function $h = h_\theta$.

This corresponds to minimizing the overall loss:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \sum_{i=1}^{n} L_i(\boldsymbol{\theta})$$

which is the sum of the loss functions $L_i = L(\boldsymbol{\theta}; \mathbf{x}_i, t_i)$ for each data point $(\mathbf{x}_i, t_i)$.

Loss function minimization approaches

How can we tackle the problem of minimizing the loss function? Ideally, our goal is to find a **global minimum** of the loss function, which would represent the best possible set of parameters for the model.

One common approach relies on calculus, specifically to setting all derivatives of the loss function with respect to the parameters to zero:

$$\nabla \mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \mathbf{0}$$

where ∇ is the gradient operator that, given a multivariate function $f(x_1, \ldots, x_m)$ returns the vector of its partial derivatives

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_m} \end{pmatrix}$$

observe that the gradient at any given point $\overline{x}_1, \ldots, \overline{x}_m$ in the domain space of f is a vector pointing in the direction of steepest ascent from that point.

Setting the gradient to $\mathbf{0}$ means solving the system of equations:

$$\frac{\partial}{\partial \theta_i} \mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = 0 \quad \forall i$$

where each partial derivative is set to zero, thereby identifying potential points where the loss could be minimal.

However, this method faces several challenges: as a first point, this system of equations often has multiple solutions, including local minima, maxima, and saddle points, making it difficult to identify a global minimum. Moreover, in many cases, solving these equations analytically is either extremely difficult or outright impossible.

Gradient descent

A local minimum of the empirical risk function $\overline{\mathcal{R}}_{\mathcal{T}}(\boldsymbol{\theta})$ can be computed numerically by means of iterative methods, such as gradient descent. The process typically begins by initializing the parameters at a starting point, $\boldsymbol{\theta}^{(0)} = (\theta_0^{(0)}, \theta_1^{(0)}, \ldots, \theta_d^{(0)})$, with an initial error value given by

$$\overline{\mathcal{R}}_{\mathcal{T}}(\boldsymbol{\theta}^{(0)}).$$

The iterative procedure then works by modifying the current parameter values $\boldsymbol{\theta}^{(i-1)}$ in the direction of the steepest descent of the empirical risk function $\overline{\mathcal{R}}_{\mathcal{T}}(\boldsymbol{\theta})$. Specifically, this means moving in the opposite direction of the gradient of the empirical risk evaluated at $\boldsymbol{\theta}^{(i-1)}$.

At each iteration i, the parameter $\theta_j^{(i-1)}$ is updated according to the following rule:

$$\theta_j^{(i)} := \theta_j^{(i-1)} - \eta \frac{\partial}{\partial \theta_j} \overline{\mathcal{R}}_{\mathcal{T}}(\boldsymbol{\theta})\Big|_{\boldsymbol{\theta}^{(i-1)}} = \theta_j^{(i-1)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \frac{\partial}{\partial \theta_j} L(h_{\boldsymbol{\theta}}(\mathbf{x}), t)\Big|_{\boldsymbol{\theta}^{(i-1)}},$$

where η represents the learning rate, which controls the step size during each update.

In matrix form, this update can be written as:

$$\boldsymbol{\theta}^{(i)} := \boldsymbol{\theta}^{(i-1)} - \eta \, \nabla \overline{\mathcal{R}}_{\mathcal{T}}(\boldsymbol{\theta})\Big|_{\boldsymbol{\theta}^{(i-1)}} = \boldsymbol{\theta}^{(i-1)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \nabla L(h_{\boldsymbol{\theta}}(\mathbf{x}), t)\Big|_{\boldsymbol{\theta}^{(i-1)}}.$$

While this method allows us to approximate a local minimum, the outcome depends heavily on the initial parameter values chosen. Some key issues to consider include:

- we are interested in finding a global minimum, but gradient descent can get stuck at local minima
- how do we handle saddle points, which are neither minima nor maxima?
- how quickly does the method converge, and what factors influence its speed?

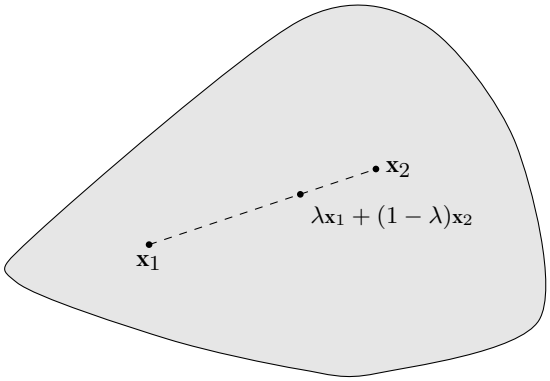
These challenges make the minimization process complex, but gradient descent remains one of the most widely used methods in practice for parameter optimization.

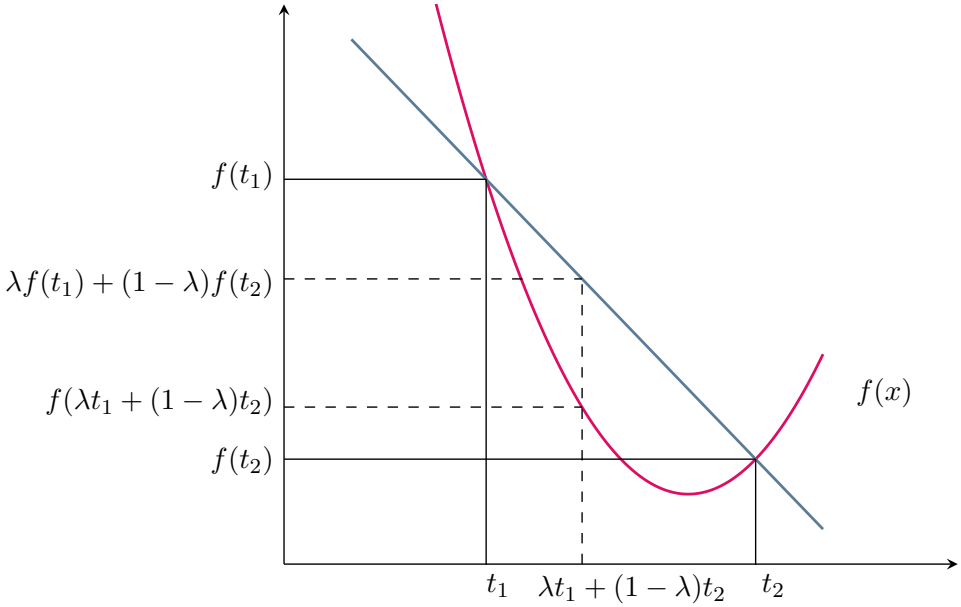
Convexity

A set of points $S \subset \mathbb{R}^d$ is convex iff for any $\mathbf{x}_1, \mathbf{x}_2 \in S$ and $\lambda \in (0, 1)$

$$\lambda \mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \in S$$

that is if all points on the line segment connecting $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to S themselves.





A function $f(\mathbf{x})$ is convex iff the set of points lying above the function is convex, that is, forall $\mathbf{x}_1, \mathbf{x}_2$ and $\lambda \in (0, 1)$,

$$f(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$$

$f(\mathbf{x})$ is strictly convex iff forall $\mathbf{x}_1, \mathbf{x}_2$ and $\lambda \in (0, 1)$,

$$f(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) < \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$$

Assuming $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ is convex is a relevant simplification: if $f(\mathbf{x})$ is a convex function, then any local minimum of f is also a global minimum. Moreover, if f is a **strictly convex** function, there exists only one local minimum for f (and it is global). That is, solving

$$\nabla\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \mathbf{0}$$

provides the global minimum.

A simple but relevant case is when $f(\mathbf{x})$ is quadratic. This is the case for a number of simple ML models, but unfortunately not true for more complex models such as neural networks.

Let us remind that:

1. the sum of (strictly) convex functions is (strictly) convex
2. the product of a (strictly) convex function and a constant is (strictly) convex

Since

$$\overline{\mathcal{R}}_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} L(h(\mathbf{x}),t) \propto \sum_{(\mathbf{x},t)\in\mathcal{T}} L(\boldsymbol{\theta};\mathbf{x},t)$$

this implies that

- if $L(\boldsymbol{\theta};\mathbf{x},t)$ is (strictly) convex then the overall cost is also (strictly) convex
- if $L(\boldsymbol{\theta};\mathbf{x},t)$ is convex then any local minimum of the empirical risk is also a global one
- if $L(\boldsymbol{\theta};\mathbf{x},t)$ is strictly convex then there exists only one minimum of the empirical risk

Some common loss functions

Loss functions for regression

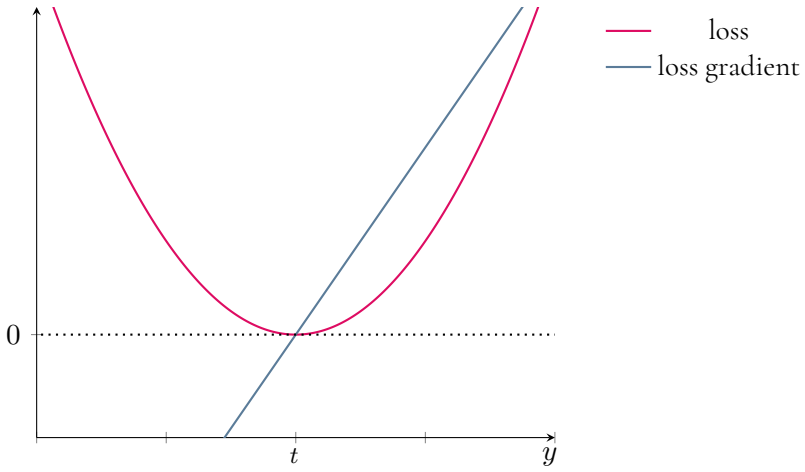
Let us first consider the case of regression.

- both y and $h(\mathbf{x})$ are real values
- loss is related to some type of point distance measure

Quadratic loss

The most common loss function for regression is the quadratic loss

$$L(y,t) = (y-t)^2$$



Quadratic loss

- Applying quadratic loss results in the empirical risk

$$\overline{\mathcal{R}}_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} (h(\mathbf{x})-t)^2$$

- in the common case of linear regression, the prediction is performed by means of a linear function $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$: this results into an overall loss to be minimized

$$\mathcal{L}(\mathbf{w}, b; \mathcal{T}) = \sum_{(\mathbf{x},t)\in\mathcal{T}} (\mathbf{w}^T\mathbf{x} + b - t)^2$$

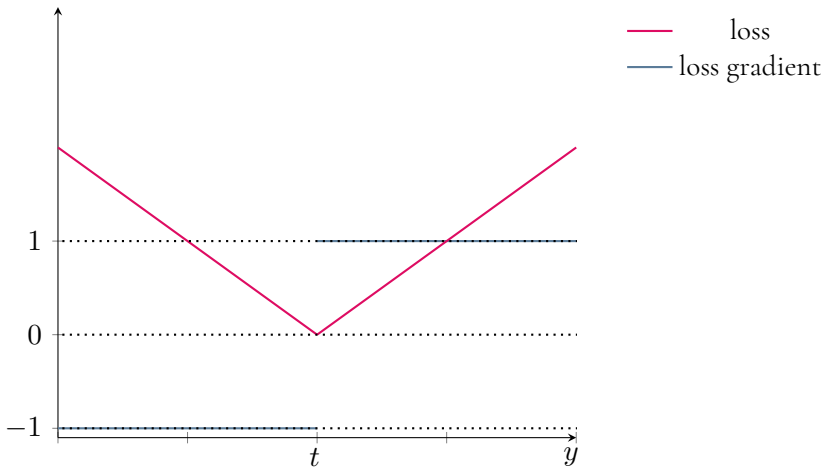
- since the quadratic function is strictly convex, the overall loss has only one local minimum (which is global)
- the gradient is linear

$$\frac{\partial}{\partial w_i}\mathcal{L}(\mathbf{w}, b; \mathcal{T}) = \sum_{(\mathbf{x},t)\in\mathcal{T}} (\mathbf{w}^T\mathbf{x} + b - t)w_i \qquad \frac{\partial}{\partial b}\mathcal{L}(\mathbf{w}, b; \mathcal{T}) = \sum_{(\mathbf{x},t)\in\mathcal{T}} (\mathbf{w}^T\mathbf{x} + b - t)$$

Absolute loss

Quadratic loss is easy to deal with mathematically, but not robust to outliers, i.e. pays too much attention to outliers.

A different loss function for regression is the **absolute loss** $L(t, y) = |t - y|$



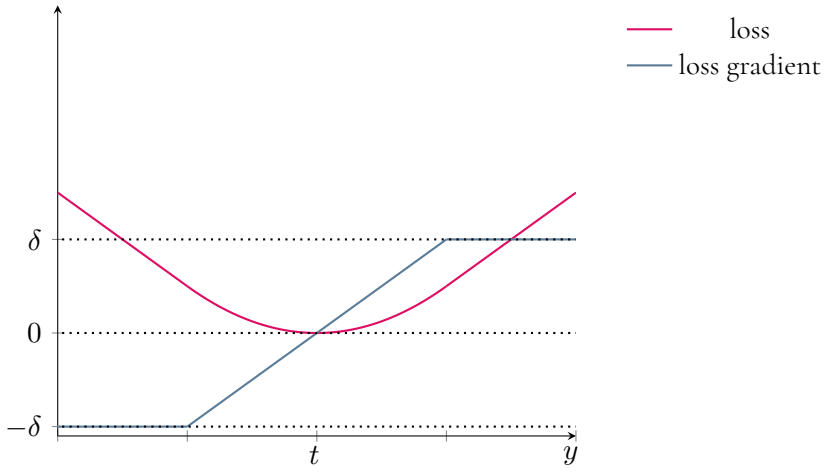
Absolute loss

The gradient is now piecewise constant.

Huber loss

Another different loss function for regression is **Huber loss** which is quadratic for small values and linear after a given threshold

$$L(t, y) = \begin{cases} \frac{1}{2}(t - y)^2 & |t - y| \leq \delta \\ \delta(|t - y|) - \frac{\delta}{2} & |t - y| > \delta \end{cases}$$



Huber loss

Loss functions for classification

Essentially, two approaches, depending on what we expect the prediction return:

- prediction returns a specific class (prediction function)
- prediction returns a probability distribution on the set of classes (prediction distribution)

This implies a different definition of error

- first case: coincidence of predicted and real classes
- second case: cumulative difference between predicted probability and $0/1$ for all classes

We consider the binary case, with two classes identified by target values -1 and 1.
Assume a real value is returned as a prediction

0/1 loss

The most "natural" loss function in classification is **0/1 loss**

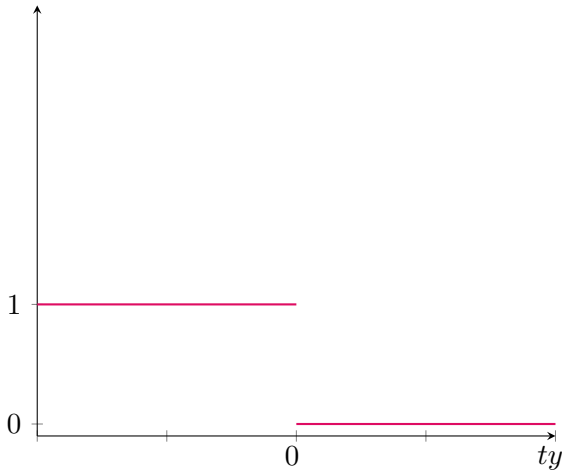
$$L(t, y) = \begin{cases} 1 & \text{sgn}(t) \neq y \\ 0 & \text{sgn}(t) = y \end{cases}$$

where $\text{sgn}(x)$ is 1 if $x > 0$ and -1 otherwise.
This can be written as:

$$\mathbb{1}[ty < 0]$$

Using 0/1 loss is problematic, since:

- it is not convex
- it is not smooth (first derivative undefined in some points or not continue)
- its gradient is 0 almost everywhere (undefined at 0): gradient descent cannot be applied



0/1 loss

- if we assume a linear prediction function

$$\overline{\mathcal{R}}_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \mathbb{1}[(\mathbf{w}^T\mathbf{x} + b)y < 0]$$

  the problem is finding the values $\mathbf{w}$, b which minimize the overall number of errors: this is NP-hard, hence a computationally intractable problem.

Convex surrogate loss functions are used instead of 0/1 loss. They:

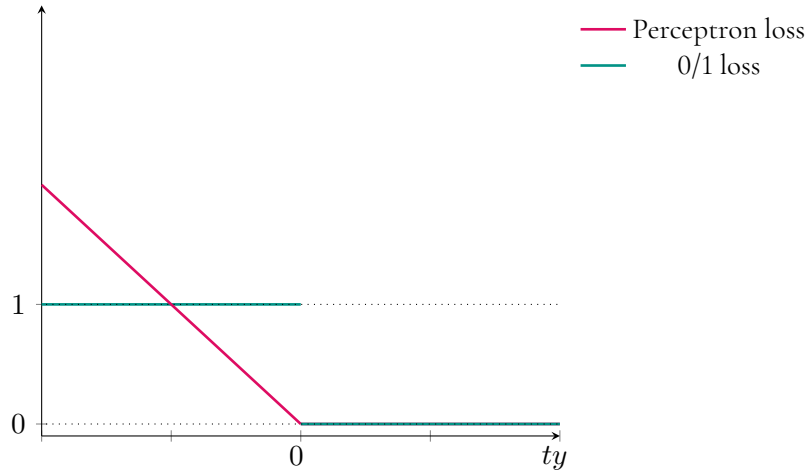
- approximate 0/1 loss **from above**: real 0/1 error always less than function loss
- are convex: unique local minimum = global minimum
- are smooth: may use derivatives to find minimum

The main difference between them is the relevance given to erroneous predictions

Perceptron loss

0/1 loss assigns the same cost 1 to each error.

- If we assume a prediction t is a real value: then, in the case of a misclassified element, the error can be measured as $-ty > 0$. That is, $L(t, y) = \max(0, -yt)$
- in the case of correctly classified element, the error is 0, while in the case of a wrong prediction, the error is equal to $|t|$
- Main difference: relevance given to erroneous predictions. The perceptron loss penalizes predictions which are largely wrong (for example a very negative value while correct class is 1)
- continuous, gradient continuous almost everywhere, convex (but not strictly convex), not surrogate

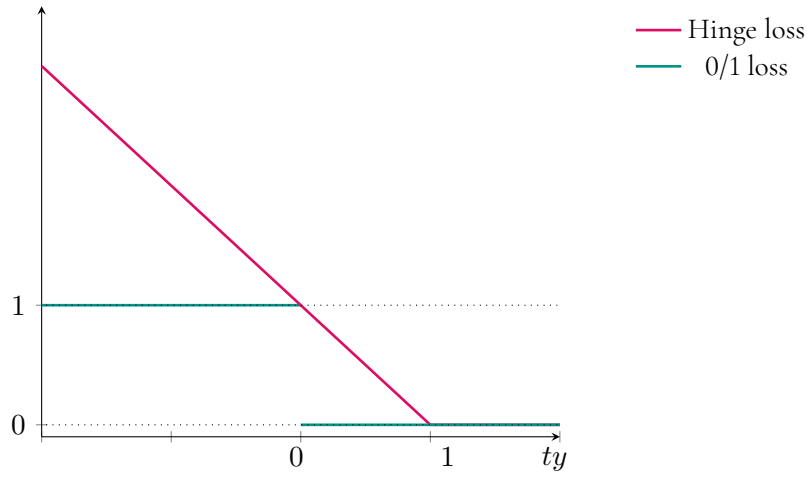


Hinge loss

- used in support vector machine training
- related to perceptron loss, but surrogate
- assume a prediction

$$L(t, y) = \max(0, 1 - yt)$$

- correct predictions can be penalized if "weak" (small value of t)
- continuous, gradient continuous almost everywhere, convex (but not strictly convex), surrogate



Hinge loss $L_H(y, t) = \max(0, 1 - yt)$ is not differentiable wrt to y at $ty = 1$. The same holds for perceptron loss at $ty = 0$.

For example,

$$\frac{\partial}{\partial y} L_H = \begin{cases} -t & ty < 1 \\ 0 & ty > 1 \\ \text{undefined} & ty = 1 \end{cases}$$

This is a problem if gradient descent should be applied. In this case a **subgradient** can be used.

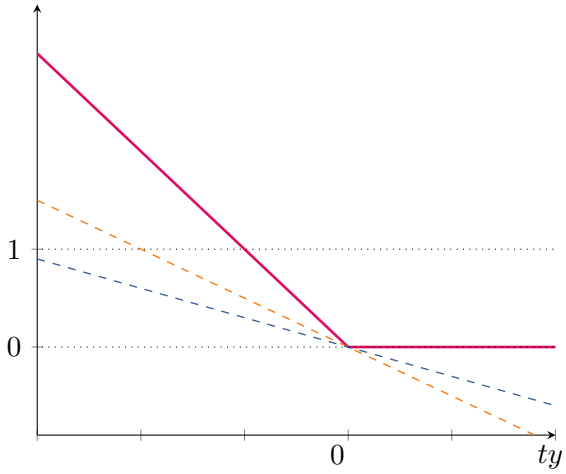
Given a convex function (such as hinge loss) f, at each differentiable point x the corresponding gradient $\nabla(x)$ provides a function which lower bounds f

$$f(x') \geq f(x) + \nabla f \mid_x (x - x')$$

If x is a singular point, where f is not differentiable and $\nabla f|_x$ does not exist, a **subgradient** $\overline{\nabla} f$ is **any** function which lower bounds f

$$f(x') \geq f(x) + \overline{\nabla} f|_x (x - x')$$

In the case of hinge loss, we may observe that any line whose slope in $[-t, 0]$ (if $t = 1$, in $[0, -t]$ if $t = -1$) is a subgradient



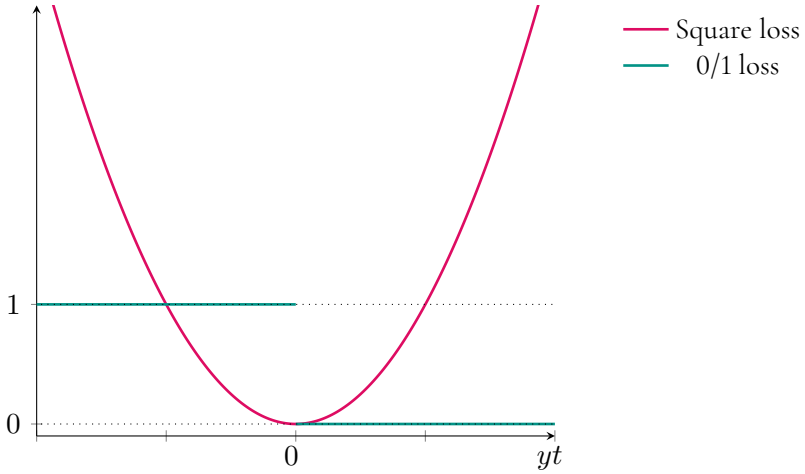
We may then choose the horizontal axis as the subgradient to use

Square loss in classification

- adapted to the classification case

$$L(t, y) = (1 - yt)^2$$

- continuous, gradient continuous, convex, not surrogate
- largely wrong predictions can be too penalized
- symmetric around 0: even largely correct predictions are penalized

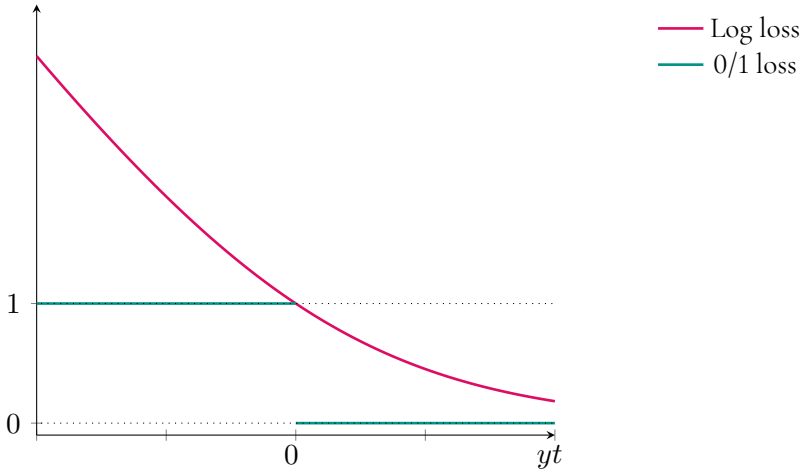


Log loss (cross entropy)

- used in logistic regression

$$L(t, y) = \frac{1}{\log 2} \log(1 + e^{-yt})$$

- a smoothed version of hinge loss
- continuous, gradient continuous, convex, surrogate
- largely wrong predictions can be too penalized



Log loss is related to the **cross entropy** measure widely applied in probabilistic classification
Given distributions p, q the cross entropy of q wrt p is defined as

$$- \mathop{\mathbb{E}}_{x \sim p} [\log q(x)] = - \int p(x) \log q(x) dx$$

The cross entropy is a measure of how much p and q are different: it is related to the Kullback-Leibler divergence

$$KL(p||q) = - \int p(x) \log \frac{q(x)}{p(x)} dx = - \int p(x) \log q(x) dx + \int p(x) \log p(x) dx = - \mathop{\mathbb{E}}_{x \sim p} [\log q(x)] - H(p)$$

where $H(p) = - \mathop{\mathbb{E}}_{x \sim p} [\log p(x)]$ is the **entropy** of p

- the entropy $H(p)$ denotes the expected number of bits per symbol x in a transmission channel where the distribution of symbols $p(x)$ is known
- the **cross entropy** $-\mathop{\mathbb{E}}_{x\sim p}[\log q(x)]$ denotes the total expected number of bits per symbol x in a transmission channel where the distribution of symbols $q(x)$ is used, instead of $p(x)$
- the KL divergence $KL(p||q)$ denotes the additional (with respect to the minimum) expected number of bits per symbol x in a transmission channel where the distribution of symbols $q(x)$ is used, instead of $p(x)$

Consider now a classifier which predicts the probability that an element is in class C_1 and let

- $p(\mathbf{x})$ be the probability that the element is in class C_1: in the training set this is either 0 or 1, that is equal to the target value t
- $y(\mathbf{x})$ be the predicted probability of the element being in class C_1

The cross entropy $\mathrm{CE}(\mathcal{T})$ between real and predicted probability distribution over the set of elements can be estimated as the average

$$\mathrm{CE}(\mathcal{T}) = -\frac{1}{|\mathcal{T}|}\sum_{(\mathbf{x},t)\in\mathcal{T}}\Big(t\log y(\mathbf{x}) + (1-t)\log(1-y(\mathbf{x}))\Big) = -\frac{1}{|\mathcal{T}|}\Big(\sum_{(\mathbf{x},t)\in C_1}\log y(\mathbf{x}) + \sum_{(\mathbf{x},t)\in C_0}\log(1-y(\mathbf{x}))\Big)$$

Assume now the classifier is a logistic regression, that is

$$y(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}+b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$$

then,

$$\mathrm{CE}(\mathcal{T}) = \frac{1}{|\mathcal{T}|}\Big(\sum_{(\mathbf{x},t)\in C_1}\log(1+e^{-(\mathbf{w}^T\mathbf{x}+b)}) + \sum_{(\mathbf{x},t)\in C_0}\log(1+e^{\mathbf{w}^T\mathbf{x}+b})\Big)$$

Assuming now that the target encodes classes as $t\in\{-1,1\}$ (that is class C_0 is denoted by $t=-1$ and class C_1 by $t=1$) we have

$$\mathrm{CE}(\mathcal{T}) = \frac{1}{|\mathcal{T}|}\sum_{(\mathbf{x},t)\in\mathcal{T}}\log(1+e^{-t(\mathbf{w}^T\mathbf{x}+b)})$$

that, apart from the constant $\log 2$ corresponds to the empirical risk if log loss is applied

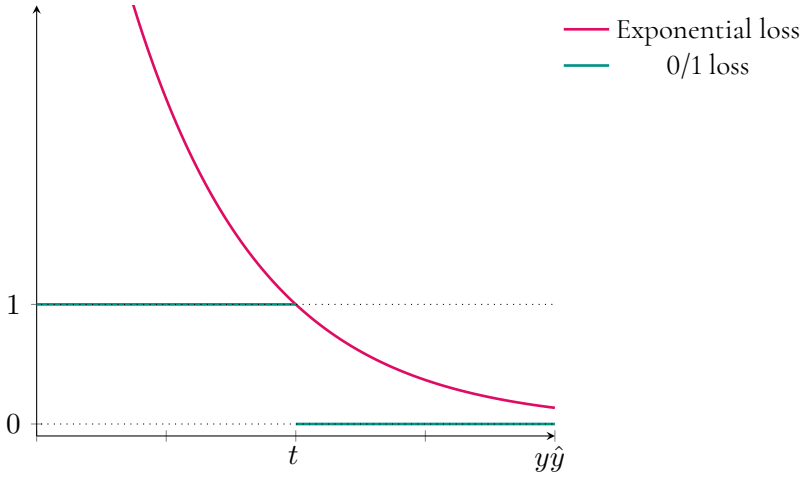
$$\overline{\mathcal{R}}_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|\log 2}\sum_{(x,t)\in\mathcal{T}}\log(1+e^{-t(\mathbf{w}^T\mathbf{x}+b)})$$

Exponential loss

- used in boosting (Adaboost)

$$L(t,y) = e^{-yt}$$

- penalizes wrong predictions more than log loss: penalty grows more quickly as errors become larger
- continuous, gradient continuous, convex, surrogate



Computing h^*

- In most cases, $\Theta = \mathbb{R}^d$ for some $d > 0$: in this case, the minimization of $\overline{\mathcal{R}}_{\mathcal{T}}(h_\theta)$ is unconstrained and a (at least local) minimum could be computed setting all partial derivatives to 0

$$\frac{\partial}{\partial \theta_i} \overline{\mathcal{R}}_{\mathcal{T}}(h_\theta) = 0$$

that is, setting to zero the gradient of the empirical risk with respect to the vector of parameters θ

$$\nabla \overline{\mathcal{R}}_{\mathcal{T}}(h_\theta) = \mathbf{0}$$

- The analytical solution of this set of equations is usually quite hard
- Numerical methods can be applied

Gradient descent

Gradient descent performs minimization of a function $J(\theta)$ through iterative updates of the current value of θ, starting from an initial value $\theta^{(0)}$, in the opposite direction to the one specified by the current value of the gradient $\nabla J|_{\theta^{(k)}}$

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla J|_{\theta^{(k)}}$$

that is, for each parameter θ_i

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \eta \frac{\partial J(\theta)}{\partial \theta_i}\bigg|_{\theta^{(k)}}$$

η is a tunable parameter, which controls the amount of update performed at each step

Batch gradient descent

If minimization of the Empirical Risk is performed, gradient descent takes the form

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \nabla L(h_\theta(\mathbf{x}), t)^{(k)}$$

that is,

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \frac{\partial}{\partial\theta_i} L(h_\theta(\mathbf{x}), t)\Big|_{\theta^{(k)}}$$

For example, in the case of linear regression

$$h(\mathbf{x}) = \sum_{j=1}^{d} \theta_j x_j + \theta_0$$

where the loss function is usually the squared distance

$$L(h(\mathbf{x}), t) = (h(\mathbf{x}) - t)^2 = \left(\sum_{j=1}^{d} \theta_j x_j + \theta_0 - t\right)^2$$

the gradient is

$$\frac{\partial}{\partial\theta_i} L(h_\theta(\mathbf{x}), t) = \left(\sum_{j=1}^{d} \theta_j x_j + \theta_0 - t\right) x_i \qquad i = 1, \ldots, d$$

$$\frac{\partial}{\partial\theta_0} L(h_\theta(\mathbf{x}), t) = \left(\sum_{j=1}^{d} \theta_j x_j + \theta_0 - t\right)$$

which results in the following updates

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t\right) x_i \qquad i = 1, \ldots, d$$

$$\theta_0^{(k+1)} = \theta_0^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(\mathbf{x},t)\in\mathcal{T}} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t\right)$$

This is called **batch gradient descent**: observe that, at each step, all items in the training set must be considered.

As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory. Batch gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

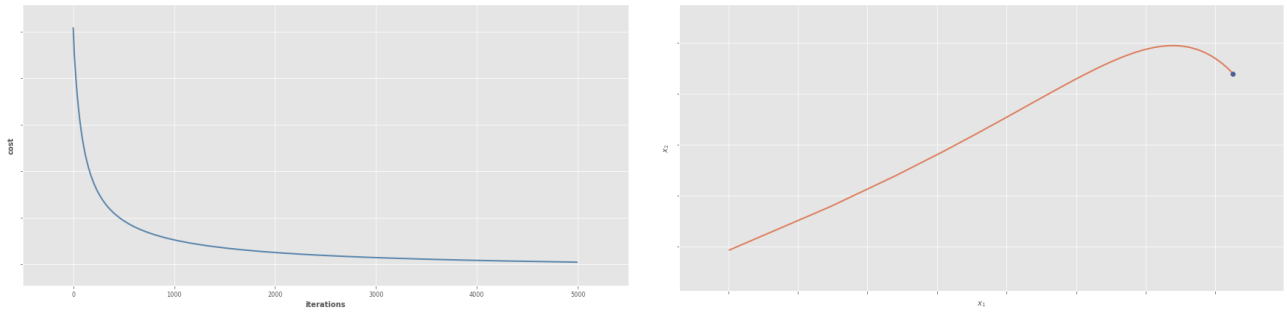
In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, the gradient vector `params_grad` of the loss function for the whole dataset w.r.t. the parameter vector `params` is computed. State-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters.

Next, parameters are updated in the direction of the gradients with the learning rate determining how big of an update we perform.

Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.



(a) Cost vs number of iterations

(b) Trajectory in feature space

Figure 1: Batch gradient descent behavior

Stochastic gradient descent

Batch gradient descent recomputes gradients for all items in the dataset before each parameter update, hence it requires long and expensive computations, especially for large datasets (as is often the case, especially in Deep Learning). SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

Stochastic gradient descent deals with this issue by performing the parameter update at each step, on the basis of the evaluation of the gradient at a single item $(\mathbf{x}_j, t_j)$ of the training set.

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla L(h_\theta(\mathbf{x}_j), t_j)^{(k)}$$

or

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \eta \frac{\partial}{\partial \theta_i} L(h_\theta(\mathbf{x}_j), t_j) \Big|_{\theta^{(k)}}$$

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Figure 2.

Batch gradient descent steadily converges to a local minimum, while SGD' trajectory is more erratical, with local cost increases. This on one side makes it possible to jump to new and potentially better local minima; on the other side, it makes convergence to the exact minimum more difficult when such minimum has been almost reached and small updates should be made. However, it has been shown that if the learning rate is slowly decreased, the same convergence behaviour of batch gradient descent is obtained, almost certainly converging to a local minimum.

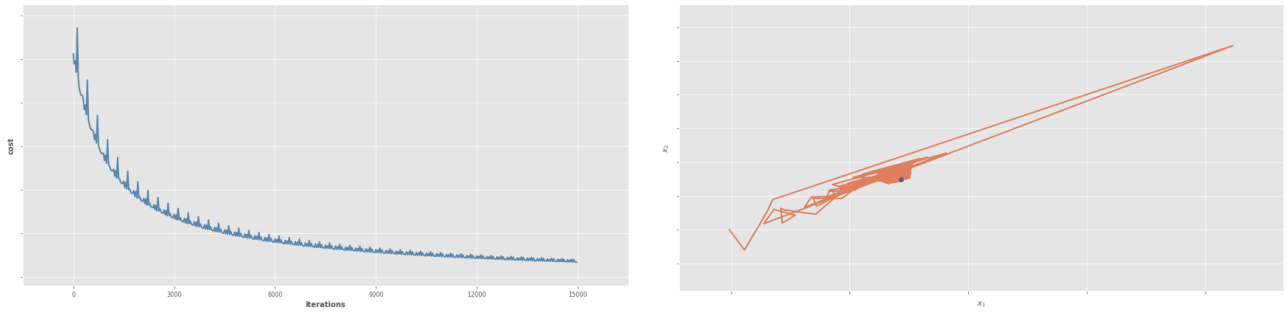
The code fragment below simply introduces a loop over the training examples and evaluates the gradient w.r.t. each example. It is usually suggested to shuffle the training data at every epoch, as done here.

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

In the case of linear regression this results into

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \eta \left(\sum_{r=1}^{d} \theta_r^{(k)} x_{jr} + \theta_0^{(k)} - t \right) x_{ji} \qquad i = 1, \ldots, d$$

$$\theta_0^{(k+1)} = \theta_0^{(k)} - \eta \left(\sum_{r=1}^{d} \theta_r^{(k)} x_{jr} + \theta_0^{(k)} - t \right)$$



(a) Cost vs number of iterations

(b) Trajectory in feature space

Figure 2: Stochastic gradient descent behavior

Mini-batch gradient descent

An intermediate case is the one when a subset B_r of size m of the items in the training is considered at each step for gradient evaluation

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\eta}{m} \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}), t)|_{\theta^{(k)}}$$

that is,

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{m} \sum_{(\mathbf{x},t)\in B_r} \frac{\partial}{\partial\theta_i} L(h_\theta(\mathbf{x}), t)|_{\theta^{(k)}}$$

This is called **mini-batch gradient descent**.
This approach

- reduces the variance of the parameter updates, which can lead to more stable convergence wrt SGD
- limits the amount of items considered for gradient evaluation before a parameter update is performed.

Observe that the size m of mini-batches is itself a tunable parameter. Common values 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.

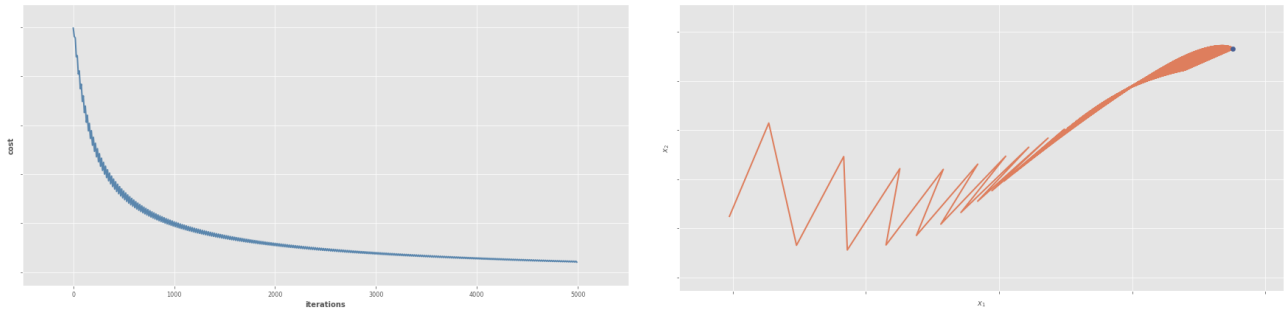
In code, instead of iterating over examples, we now iterate over mini-batches of size m:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=m):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

In the case of linear regression it is clearly

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{m} \sum_{(\mathbf{x},t)\in B_r} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t \right) x_i \qquad i = 1, \ldots, d$$

$$\theta_0^{(k+1)} = \theta_0^{(k)} - \frac{\eta}{m} \sum_{(\mathbf{x},t)\in B_r} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t \right)$$



(a) Cost vs number of iterations

(b) Trajectory in feature space

Figure 3: Mini-batch gradient descent behavior

Open issues

The approaches considered up to now differ by the number of items considered for gradient evaluation at each step, before a parameter update is performed. However, they not guarantee good convergence, due to a few challenges that need to be addressed:

- Choosing a proper value for η can be difficult. A too small learning rate may lead to very slow convergence, while a too large learning rate can affect convergence and cause the loss function to fluctuate around the minimum, or even to diverge.
- In order to deal with this issue, we could apply some mechanism to adjust the learning rate during training by reducing it either according to a pre-defined schedule or when the loss function decrease between epochs falls below a threshold. Both schedules and thresholds, however, should be defined in advance and are thus unable to adapt to the characteristics of a dataset.
- The same learning rate applies to updating all parameter.
- In many cases, such as for example in neural networks (and Deep learning) is highly non-convex, with many local minima and saddle points. The approaches considered above could find it hard to not get trapped in these scenarios, in particular in the case of saddle points, which are usually surrounded by a plateau, making it hard for simple gradient descent methods to escape, as the gradient is almost zero in all dimensions.

Momentum gradient descent

This approach is based on a physical interpretation of the optimization process, interpreted as the movement of a body of mass $m = 1$, under the effect of a weight force F, on the surface of the cost function $J(\theta)$. The weight force is assumed to be $F(\theta) = -\nabla U$, where $U(\theta) = \eta J(\theta)$ is the potential energy of the body at point θ (we assume the constant g of the weight force $F = -mgh$ is then equal to η). In this model, the negative $-\eta\nabla J$ of the gradient is then equal to the force (and acceleration, since $m = 1$) vector applied on the body at point θ.

In gradient descent, the movement of the body at a point θ is determined by the acceleration ∇J at that point, since $\theta^{(k+1)} = \theta^{(k)} - \eta\nabla J|_{\theta^{(k)}}$.

In momentum gradient descent, we refer to a model which is more consistent with the situation of a body moving on a surface under the effect of the weight force. In this model, the movement of the body at point θ is determined by its speed $v(\theta)$ at that point, that is, $\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)}$, where the difference of velocity derives from the acceleration at point θ, that is $v^{(k+1)} = v^{(k)} - \eta\nabla J|_{\theta^{(k)}}$.

This results in the following operations at each step

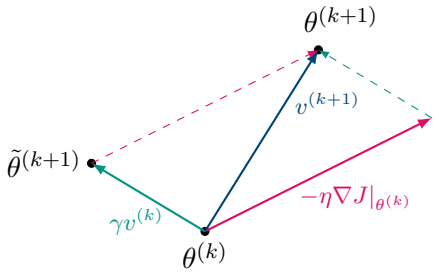


Figure 4: Momentum gradient descent

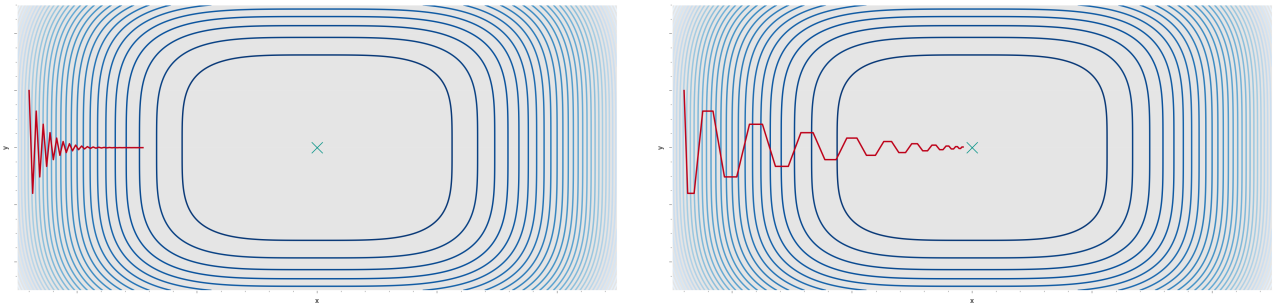
$$v^{(k+1)} = v^{(k)} - \eta \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)\,|_{\theta^{(k)}} = v^{(k-1)} - \eta \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)\,|_{\theta^{(k-1)}} - \eta \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)\,|_{\theta^{(k)}} = \cdots$$

$$= v^{(0)} - \eta \sum_{i=0}^{k} \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)\,|_{\theta^{(i)}}$$

$$\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)} = \theta^{(k)} v^{(0)} - \eta \sum_{i=0}^{k} \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)\,|_{\theta^{(i)}}$$

that corresponds to define the update in terms of the sum of past gradients (integral of past accelerations in physics. The momentum v_i increases for dimensions whose gradients are consistently directed in the same directions, while decreasing for dimensions whose gradients change directions at each step.

Referring to that physical model makes the algorithm tend at each step to keep, at least in part, the direction of the preceding step, since $v^{(k+1)} = v^{(k)} - \eta\nabla J|_{\theta^{(k)}}$, thus rewarding directions which are returned consistently in the sequence of steps. This can be clearly seen in Figure 5, where the momentum leads to a limitation to the size of oscillations in the direction orthogonal to the one towards the minimum. This does not happen in the case of simple gradient descent, where $v^{(k+1)} = v^{(k)} - \eta\nabla J|_{\theta^{(k)}}$.



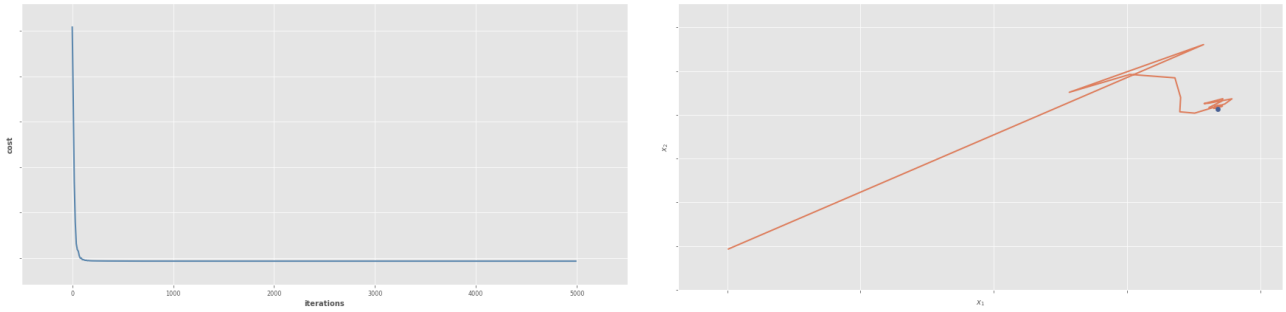
(a) GD without momentum

(b) Momentum GD

Figure 5: Momentum effect on trajectory

In momentum gradient descent it is usually introduced a second parameter γ, which affects the fraction of $v^{(k)}$ that is considered for the computation of $v^{(k+1)}$. In terms of physical model, this corresponds to introducing an attrition coefficient. Applying the approach to the case of mini-batches, we get:

$$
\begin{aligned}
v^{(k+1)} &= \gamma v^{(k)} - \eta \sum_{(\mathbf{x},t)\in B_r} \nabla L(h_\theta(\mathbf{x}),t)|_{\theta^{(k)}} \\
\theta^{(k+1)} &= \theta^{(k)} + v^{(k+1)}
\end{aligned}
$$



(a) Cost vs number of iterations

(b) Trajectory in feature space

Figure 6: Momentum gradient descent behavior

In the case of linear regression, this results into:

$$
\begin{aligned}
v_i^{(k+1)} &= \begin{cases} \gamma v_i^{(k)} - \eta \displaystyle\sum_{(\mathbf{x},t)\in B_r} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t \right) x_i & i = 1,\dots,d \\ \gamma v_i^{(k)} - \eta \displaystyle\sum_{(\mathbf{x},t)\in B_r} \left(\sum_{j=1}^{d} \theta_j^{(k)} x_j + \theta_0^{(k)} - t \right) & i = 0 \end{cases} \\
\theta_i^{(k+1)} &= \theta_i^{(k)} + v_i^{(k+1)}
\end{aligned}
$$

Nesterov accelerated gradient descent

In momentum gradient descent, adding $\gamma v^{(k)}$ to $\theta^{(k)}$ provides an approximation

$$
\tilde{\theta}^{(k+1)} \triangleq \theta^{(k)} + \gamma v^{(k)}
$$

of the real value $\theta^{(k+1)}$

$$
\begin{aligned}
\tilde{\theta}^{(k+1)} &= \theta^{(k)} + \gamma v^{(k)} \\
v^{(k+1)} &= \gamma v^{(k)} - \eta \nabla J|_{\theta^{(k)}} \\
\theta^{(k+1)} &= \theta^{(k)} + v^{(k+1)}
\end{aligned}
$$

Nesterov accelerated gradient follows the same approach of momentum GD, with the only difference that, at each step, the gradient is not evaluated at the current point $\theta^{(k)}$. Instead, gradient evaluation is done with an

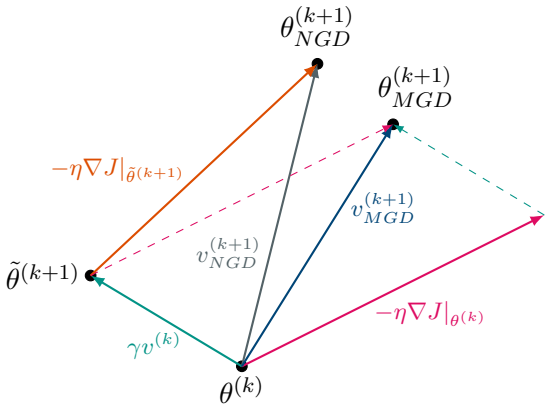
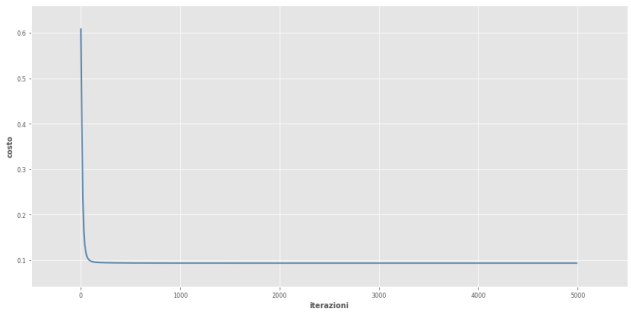


Figure 7: Nesterov vs momentum GD steps

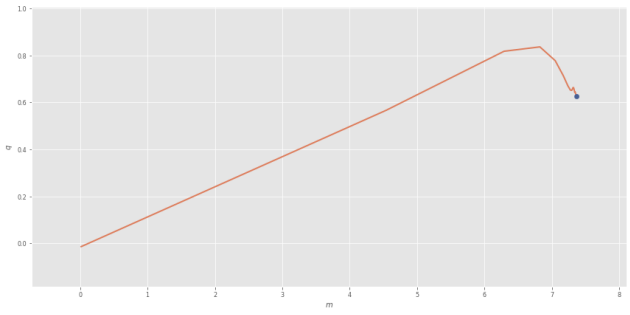
approximated look-ahead, at point $\tilde{\theta}^{(k+1)}$, which is expected to be nearer to point at the next step $\tilde{\theta}^{(k+1)}$. In such a way, changes of v (and of θ) are anticipated with respect to what happens in momentum gradient descent.

$$\tilde{\theta}^{(k+1)} = \theta^{(k)} + \gamma v^{(k)}$$
$$v^{(k+1)} = \gamma v^{(k)} - \eta\nabla J\,|_{\theta^{(k+1)}}$$
$$\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)}$$

- The same approach of momentum gradient descent is applied, with the gradient estimation performed not at the current point $\theta^{(k)}$, but approximately at the next point $\theta^{(k+1)}$
- The approximation derives by considering $\tilde{\theta}^{(k)} = \theta^{(k)} + \gamma v^{(k)}$ instead of $\theta^{(k+1)}$
- The updates of v and θ are considered in advance with respect to momentum GD



(a) Cost vs number of iterations



(b) Trajectory in feature space

Figure 8: Nesterov accelerated gradient descent behavior

Dynamically updating the learning rate

The learning rate η is a crucial parameter in gradient descent

- Too large: overshoots local minimum, loss increases

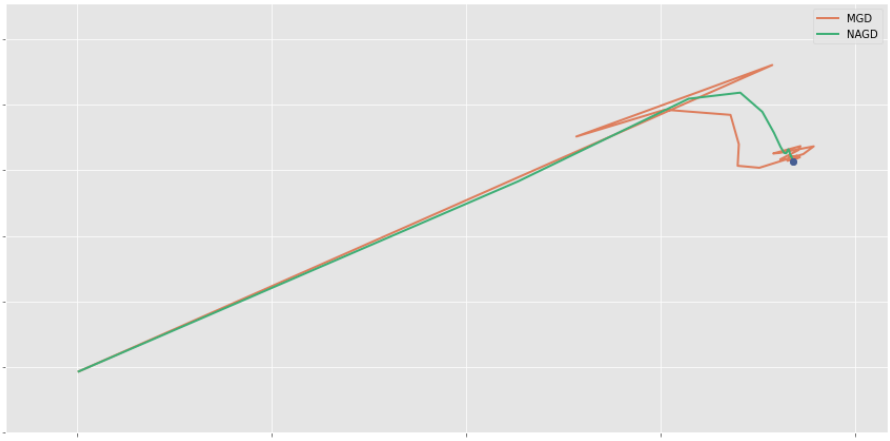


Figure 9: Comparison of MGD and NGD trajectories

- Too small: makes very slow progress, can get stuck

A good learning rate allows making steady progress toward local minimum. However, a learning rate whose value is the same along all process would result in the possibility of too short steps at the beginning (if η is small) or too long steps as the local minimum neighborhood is reached.

These contrasting requirements can be satisfied by gradually decreasing of the learning rate according to a learning rate schedule, that is updating η at each step, or epoch, by applying a predefined rule.

- Step decay drops the learning rate by a constant factor c every K steps (or epochs). That is, every K epochs decay $\eta = \frac{\eta}{c}$
- Exponential decay: at each iteration, $\eta^{(k)} = \eta^{(0)} e^{-\alpha k}$
- $\frac{1}{t}$ decay: $\eta^{(k)} = \frac{\eta^{(0)}}{1 + \alpha k}$

The main problem with learning rate schedules is that their hyperparameters must be defined in advance and they depend heavily on the type of model and problem. Another problem is that the same learning rate is applied to all parameter updates.

It seems preferrable, instead, to update each parameter θ_i independently from the other ones, with a learning rate η_i which is dynamically updated according to the history of values of the derivative of the cost function wrt θ_i.

This makes it possible, for parameters with large derivatives in the preceding steps to be associated to smaller learning rates, in such a way that the following updates are limited. On the contrary, parameters which were almost constant in the last steps will be assigned higher learning rates, to make updates more sensitive to small values of the derivative.

Adagrad

In gradient descent the update of parameter θ_j is the following

$$\theta_j^{(k+1)} = \theta_j^{(k)} - \eta \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}}$$

where the learning rate η is equal for all parameters.

We now rewrite this update in terms of the parameter update $\Delta\theta_{j,k}$, as a sequence of three steps:

$$
\begin{aligned}
g_{j,k} &= \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}} \\
\Delta_{j,k} &= -\eta g_{j,k} \\
\theta_j^{(k+1)} &= \theta_j^{(k)} + \Delta_{j,k}
\end{aligned}
$$

Adagrad modifies this behavior for what regards the computation of $\Delta_{j,k}$ by adapting the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

In Adagrad, each parameter update refers to a learning rate $\eta_j^{(k)}$, that is

$$
\Delta_{j,k} = -\eta_j^{(k)} g_{j,k}
$$

where $\eta_j^{(k)}$ is dependent on the parameter itself and a common predefined learning rate η

$$
\eta_j^{(k)} = \frac{\eta}{\sqrt{G_{j,k} + \varepsilon}}
$$

and

$$
G_{j,k} = \sum_{i=0}^{k} g_{j,i}^2
$$

is the sum of the squared derivatives of the loss function wrt to θ_i computed for all previous iterations. ε is a small smoothing constant, introduced to avoid null denominators.

This results into

$$
\Delta_{j,k} = -\frac{\eta}{\sqrt{G_{j,k} + \varepsilon}} g_{j,k}
$$

The update of θ_j at the $(k+1)$-th iteration is then defined as

$$
\begin{aligned}
g_{j,k} &= \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}} \\
G_{j,k} &= G_{j,k-1} + g_{j,k}^2 \\
\Delta_{j,k} &= -\frac{\eta}{\sqrt{G_{j,k} + \varepsilon}} g_{j,k} \\
\theta_j^{(k+1)} &= \theta_j^{(k)} + \Delta_{j,k}
\end{aligned}
$$

where $G_{j,k} = 0$ if $k < 0$.

As it can be seen, learning rates decrease at each step, with the ones associated to parameters which had large gradients in the past decreasing more. The learning rates of parameters which had large gradients in the past (hence were characterised by large variations in value) will be decreased faster, and the values of such parameters will be less modified. The opposite happens for parameters whose values remained almost constant in the past: their learning rates will be larger, "pushing" them more quickgly towards a stable value

However, in both cases, since the denominator of η_j however increases at each iterations, the learning rate monotonically decreases towards values small enough to forbid real updates of the solution.

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use default values $\eta \simeq 0.01$ and $\varepsilon \simeq 10^{-8}$, but tuning of such values can be performed to improve the method performances.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: since every added term is positive, the accumulated sum keeps growing during training. This in turn, as observed above, causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

RMSprop

RMSprop is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, RMSprop restricts the window of accumulated past gradients to some fixed size w.

In RMSprop, we replace the sum over past squared gradients $G_{j,k}$ with its decaying version $\tilde{G}_{j,k}$. That is, the sum of past squared derivatives is still considered, but with a decreasing relevance of long past ones. This is obtained through a **decay**, obtained by applying a coefficient $0 < \gamma < 1$

$$\begin{aligned}\tilde{G}_{j,k} &= \gamma \tilde{G}_{j,k-1} + (1-\gamma) g_{j,k}^2 \\ &= \gamma(\gamma \tilde{G}_{j,k-2} + (1-\gamma) g_{j,k-1}^2) + (1-\gamma) g_{j,k}^2 = \gamma^2 \tilde{G}_{j,k-2} + (1-\gamma)(\gamma g_{j,k-1}^2 + g_{j,k}^2) \\ &= \cdots \\ &= (1-\gamma) \sum_{i=0}^{k} \gamma^{k-i} g_{j,i}^2\end{aligned}$$

since we assume $\tilde{G}_{j,k} = 0$ if $k < 0$.

$\eta_j^{(k)}$ is now defined as

$$\eta_j^{(k)} = -\frac{\eta}{\sqrt{\tilde{G}_{j,k} + \varepsilon}}$$

It is recursively defined by referring to a decaying sum of all past squared derivatives. The sum $\tilde{G}_{j,k}$ at step k depends (as a fraction γ, similarly to Momentum GD) only on the previous sum and the current gradient.

This results into the following step, at the $k+1$-th iteration

$$\begin{aligned}g_{j,k} &= \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}} \\ \tilde{G}_{j,k} &= \gamma \tilde{G}_{j,k-1} + (1-\gamma) g_{j,k}^2 \\ \Delta_{j,k} &= -\frac{\eta}{\sqrt{\tilde{G}_{j,k} + \varepsilon}} g_{j,k} \\ \theta_j^{(k+1)} &= \theta_j^{(k)} + \Delta_{j,k}\end{aligned}$$

RMSprop is characterized by the two parameters $\eta, \gamma, \varepsilon$: common values for such parameters are $\gamma \simeq 0.9$, $\eta \simeq 0.1$ and $\varepsilon \simeq 10^{-8}$.

Adadelta

Adadelta is an extension of RMSprop in which no value η has to be arbitrarily defined: it is instead substituted by the decayed sum of previous squared updates, with the same decay γ applied for derivatives

$$\overline{G}_{j,k} = \gamma \overline{G}_{j,k-1} + (1-\gamma)\Delta^2_{j,k} = (1-\gamma)\sum_{i=0}^{k} \gamma^{k-i}\Delta^2_{j,i}$$

assuming, again, $\overline{G}_{j,k} = 0$ if $k < 0$.
The update rule is then defined as

$$\begin{aligned}
g_{j,k} &= \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}} \\
\tilde{G}_{j,k} &= \gamma \tilde{G}_{j,k-1} + (1-\gamma) g^2_{j,k} \\
\Delta_{j,k} &= -\frac{\sqrt{\overline{G}_{j,k-1} + \varepsilon}}{\sqrt{\tilde{G}_{j,k} + \varepsilon}} g_{j,k} \\
\overline{G}_{j,k} &= \gamma \overline{G}_{j,k-1} + (1-\gamma)\Delta^2_{j,k} \\
\theta^{(k+1)}_j &= \theta^{(k)}_j + \Delta_{j,k}
\end{aligned}$$

If we consider the Root Mean Square, smoothed by ϵ and decayed by γ, of a sequence $a_i, i = 0, \ldots, n$

$$RMS(a_0, \ldots, a_n) = \sqrt{\gamma^n a_0 + \gamma^{n-1} a_1 + \gamma^{n-2} a_2 + \cdots + a_n + \varepsilon}$$

then, an interpretation of the update rule is that the current gradient is weighted by the ratio of the RMS of the past $k-1$ updates and the RMS of the past $k-1$ derivatives (plus the current one), that we may assume is a measure of the expected effect of a unit of parameter increase on the update of the loss function.

Common default values of the method parameters are $\gamma \simeq 0.95$, and $\varepsilon \simeq 10^{-8}$. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Adam

Adam (Adaptive Moment Estimation) is another method that computes adaptive learning rates for each parameter. In addition to storing the exponentially decaying sum $\tilde{G}_{j,k}$ of past squared derivatives $g^2_{j,k}$ like Adadelta and RMSprop (to be used in the same way as in such methods), Adam also keeps an exponentially decaying sum $\tilde{H}_{j,k}$ of past (non squared) derivatives $g_{j,k}$, as a substitute to the derivative $g_{j,k}$ in the iteration step.

$$\begin{aligned}
\tilde{G}_{j,k} &= \gamma \tilde{G}_{j,k-1} + (1-\gamma) g^2_{j,k} \\
\tilde{H}_{j,k} &= \beta \tilde{H}_{j,k-1} + (1-\beta) g_{j,k}
\end{aligned}$$

Since it is assumed that $\tilde{H}_{j,k} = \tilde{G}_{j,k} = 0$ if $k < 0$ and γ, β values are usually both close to 1, the methods presents a tendency (bias) to return small values of $\tilde{H}_{j,k}$ and $\tilde{G}_{j,k}$, especially during the initial time steps.

This issue is managed by applying a bias correction:

$$\hat{G}_{j,k} = \frac{\tilde{G}_{j,k}}{1-\gamma^k}$$
$$\hat{H}_{j,k} = \frac{\tilde{H}_{j,k}}{1-\beta^k}$$

Parameters are updated just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\begin{aligned}
g_{j,k} &= \frac{\partial J(\theta)}{\partial \theta_j}\Big|_{\theta^{(k)}} \\
\tilde{G}_{j,k} &= \gamma \tilde{G}_{j,k-1} + (1-\gamma) g_{j,k}^2 \\
\tilde{H}_{j,k} &= \beta \tilde{H}_{j,k-1} + (1-\beta) g_{j,k} \\
\hat{G}_{j,k} &= \frac{\tilde{G}_{j,k}}{1-\gamma^k} \\
\hat{H}_{j,k} &= \frac{\tilde{H}_{j,k}}{1-\beta^k} \\
\Delta_{j,k} &= -\frac{\eta}{\sqrt{\hat{G}_{j,k}+\varepsilon}} \hat{H}_{j,k} \\
\theta_j^{(k+1)} &= \theta_j^{(k)} + \Delta_{j,k}
\end{aligned}$$

Common default values of the method parameters are $\eta \simeq 0.001$, $\gamma \simeq 0.9$, $\beta \simeq 0.999$, and $\varepsilon \simeq 10^{-8}$. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Second order methods

Maxima (or minima) of $J(\theta)$ can be found by searching points where the gradient (all partial derivatives) is zero.

Any iterative method to compute zeros of a function (such as Newton-Raphson) can then be applied on the gradient $\nabla J(\theta)$

The basic idea of Newton's method is to use both the first-order derivative (gradient) and second-order derivative (Hessian matrix) to approximate the objective function with a quadratic function, and then solve the minimum optimization of the quadratic function. This process is repeated until the updated variable converges.

The one-dimensional Newton's iteration formula is shown as

$$\theta_j^{(k+1)} = \theta_j^{(k)} - \frac{J'(\theta)}{J''(\theta)}\Big|_{\theta^{(k)}}$$

More general, the high-dimensional Newton's iteration formula is

$$\theta^{(k+1)} = \theta^{(k)} - \left(H(J)^{-1}\nabla J\right)\Big|_{\theta^{(k)}}$$

where $H(J)$ is the Hessian matrix of $J(\theta)$. More precisely, if the learning rate (step size factor) is introduced, the iteration formula is shown as

$$\begin{aligned}
\Delta^{(k)} &= -\left(H(J)^{-1}\nabla J\right)\Big|_{\theta^{(k)}} \\
\theta^{(k+1)} &= \theta^{(k)} + \eta_t \Delta^{(k)},
\end{aligned}$$

where $\Delta^{(k)}$ is the Newton's direction, η is the step size. This method can be called damping Newton's method.

Geometrically speaking, Newton's method operates by fitting the local surface of the current position with a quadratic surface, while gradient descent fits the current local surface with a plane.

Newton's method is an iterative algorithm that requires the computation of the inverse Hessian matrix of the objective function at each step, which makes the storage and computation very expensive.

To overcome the expensive storage and computation, approximate algorithms were considered such as **quasi-Newton methods**. The essential idea of all quasi-Newton methods is to use a positive definite matrix to approximate the inverse of the Hessian matrix, thus simplifying the complexity of the operation.