Deep Learning and Neural Networks

Demetrio Labate

February 22, 2024

Part 1
Feedforward Neural Networks

1.5 Advanced Implementation

MLP - Implementation

The design and implementation of MLPs for machine learning applications requires data preparation through

- **Batch normalization**

and optimization of the training process that may include

- **Normalized gradient descent**
- **Momentum method**
- **Regularization**
- **Stochastic and mini-batch gradient descent**
- **General steepest descent**
- **Early stopping**

Input normalization

In linear supervised learning, **normalization** of the the input feature of a dataset significantly aids in tuning the parameters of the model as it improves the properties of the cost function.

To carry out this operation to MLPs, we need to normalize the output of each network activation.
Moreover, since the activation outputs change during parameter tuning - e.g., whenever a gradient descent step is made - we must normalize every time we make a parameter update.

This leads to the incorporation of a normalization step onto the architecture of the MLP which is called every time weights are changed. This natural extension of input normalization is referred to as **batch normalization.**

Input normalization

In linear supervised learning, we employ a linear model

$$\Phi(x, W) = w_0 + x_1 w_1 + \cdots + x_D w_D$$

where $x = (x_, \ldots, x_D) \in \mathbb{R}^d$ is the input and $W = (w_0, \ldots, w_D)$ are the parameters or weights of the model.

When tuning these weights during training via the minimization of any cost function over a training set of P points $\{x^{(1)}, \ldots, x^{(P)}\}$, the input distribution along the n-th coordinate $\{x_n^{(p)}\}_{p=1}^{P}$ touching w_n may affects the cost function along the w_n direction making optimization via gradient descent more challenging.

Input normalization

To control the support of the cost function and make it easier to optimize, a solution is to normalize each input dimensions via the standard normalization, that is, by mean centering and re-scaling by its standard deviation.

Hence, for each input feature $x^{(p)}$, we make the change of variable

$$x_n^{(p)} \leftarrow \frac{x_n^{(p)} - \mu_n}{\sigma_n}$$

where σ_n and μ_n are the mean and standard deviation along the n-th coordinate of the training set, respectively.

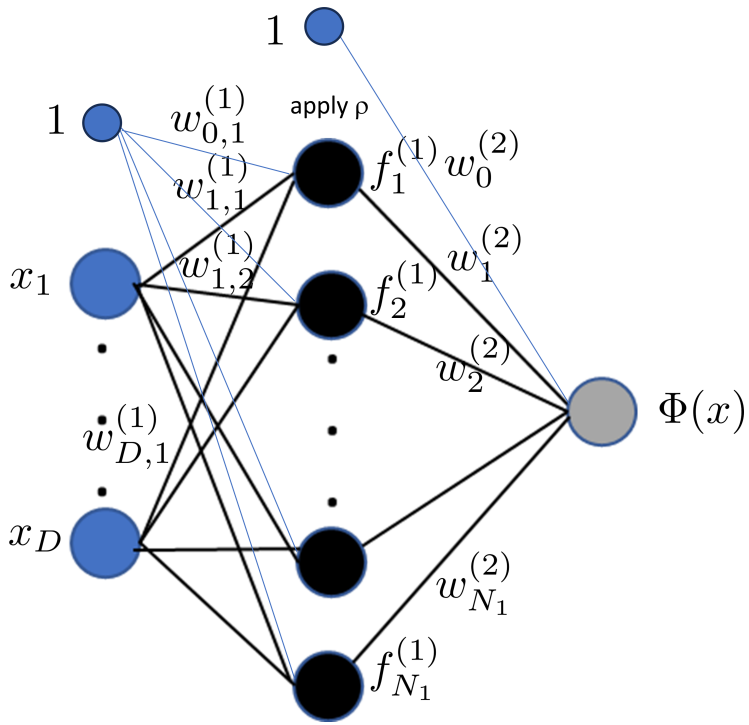
Note that once the training set $\{x^{(1)}, \ldots, x^{(P)}\}$ is normalized, they never change again and remain stable regardless of how we set the parameters of our model during training.

Batch normalization - single hidden layer

To understand the normalization step in the MLP, let us consider the simplest case of an MLP with **single hidden layer** where

$$\Phi(x, W) = w_0^{(2)} + f_1^{(1)}(x) w_1^{(2)} + \cdots + f_N^{(1)}(x) w_N^{(2)}$$

with $f_j^{(1)}(x) = \rho\left(w_{0,j}^{(L)} + \sum_{n=1}^{D} w_{n,j}^{(L)} x_n\right)$



Batch normalization - single hidden layer

In this situation where

$$\Phi(x, W) = w_0^{(2)} + f_1^{(1)}(x)w_1^{(2)} + \cdots + f_N^{(1)}(x)w_N^{(2)}$$

we can see that the data input does not affect the weights of the linear combination $w_1^{(2)}, \ldots, w_N^{(2)}$ (as was the case with the linear model), but does affect the internal weights $w_1^{(1)}, \ldots, w_N^{(1)}$.

Hence, by performing standard normalization on the input here we control the support of the cost function along the internal weights of a model employing single hidden layer elements.

We also observe that the weight $w_1^{(2)}, \ldots, w_N^{(2)}$ are affected by the output distribution of the hidden layer $f_1^{(1)}(x^{(p)}), \ldots, f_N^{(1)}(x^{(p)})$, where $\{x^{(p)} : p = 1, \ldots, P\}$ is the training set.

Batch normalization - single hidden layer

Thus, if the distribution touching the weight of a model affects the cost function then, in analogy to how we have normalized the input data, we need to normalize the output distribution of the hidden layer.

We apply standard normalization to each unit by mean centering and re-scaling as

$$f_j^{(1)}(x) \leftarrow \frac{f_j^{(1)}(x) - \mu_{f_j^{(1)}}}{\sigma_{f_j^{(1)}}}$$

where

$$\mu_{f_j^{(1)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(1)}(x^{(p)})$$

$$\sigma_{f_j^{(1)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}(f_j^{(1)}(x^{(p)}) - \mu_{f_j^{(1)}})^2}$$

Batch normalization

Unlike the distribution of the input data, the distribution of each of our single layer units changes every time the internal parameters of the system are changed.

That is, since each of the hidden layer units $f_j^{(1)}$ is a function of internal parameters $(w_{n,j}^{(1)})$, the distribution $\{f_j^{(1)}(x^{(p)})\}_{p=1}^{P}$ varies depending on the setting of these internal weights.

In the jargon of deep learning this is often referred to as **internal covariate shift** or just **covariate shift** of a network model.

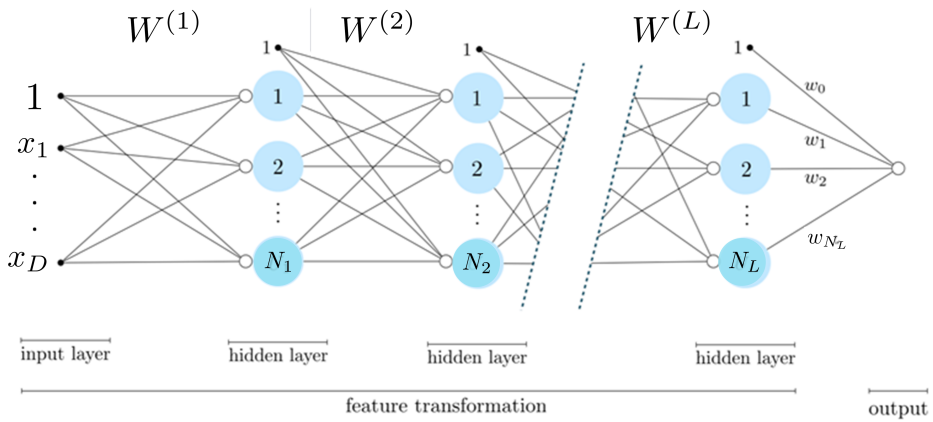
Since the weights of our model change during training - e.g., from one step of gradient descent to the next - in order to keep the unit distributions normalized we have to **normalize them at every step of parameter tuning** (e.g., via gradient descent).
To do this, a standard normalization step is usually built in directly into the MLP architecture itself.

Batch normalization - multiple hidden layers

We can now extend the normalization step in the MLP to the multilayer case where

$$\Phi(x, W) = w_0^{(L+1)} + f_1^{(L)}(x) w_1^{(L+1)} + \cdots + f_{N_L}^{(L)}(x) w_{N_L}^{(L+1)}$$

with $f_j^{(L)}(x) = \rho\left(w_{0,j}^{(1)} + \sum_{n=1}^{N_{L-1}} w_{n,j}^{(1)} f_n^{(L-1)}(x)\right)$



The output units $f_j^{(L)}(x)$ do not depends directly on the input x but on the output units of the preceding hidden layer $f_i^{(L-1)}(x)$.

Batch normalization - multiple hidden layers

In a general MLP, we standard normalize every unit in every layer. For any layer ℓ we apply standard normalization as

$$f_j^{(\ell)}(x) \leftarrow \frac{f_j^{(\ell)}(x) - \mu_{f_j^{(\ell)}}}{\sigma_{f_j^{(\ell)}}}$$

where

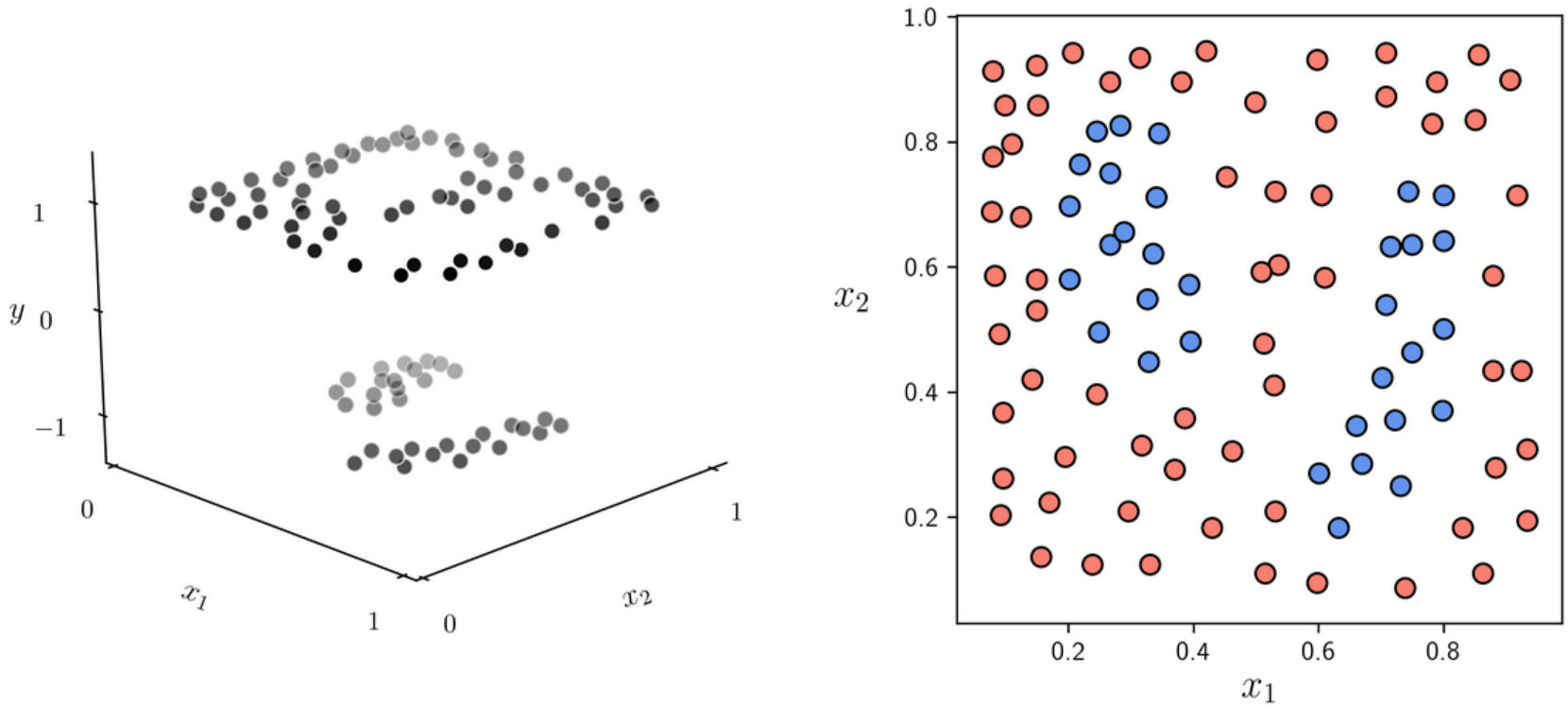
$$\mu_{f_j^{(\ell)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(\ell)}(x^{(p)})$$

$$\sigma_{f_j^{(\ell)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}(f_j^{(\ell)}(x^{(p)}) - \mu_{f_j^{(\ell)}})^2}$$

This procedure is referred as **batch normalization**.

Batch normalization - Example

To illustrate the impact of normalization, we consider the two-class classification dataset shown below.



For classification, we use a MLP with a single hidden layer, reLU activation function and 2 neurons in the hidden layer.

Batch normalization - Example

Thus, our model is simply

$$\Phi(x, W) = w_0^{(2)} + f_1^{(1)}(x) w_1^{(2)} + f_2^{(1)}(x) w_2^{(2)}$$

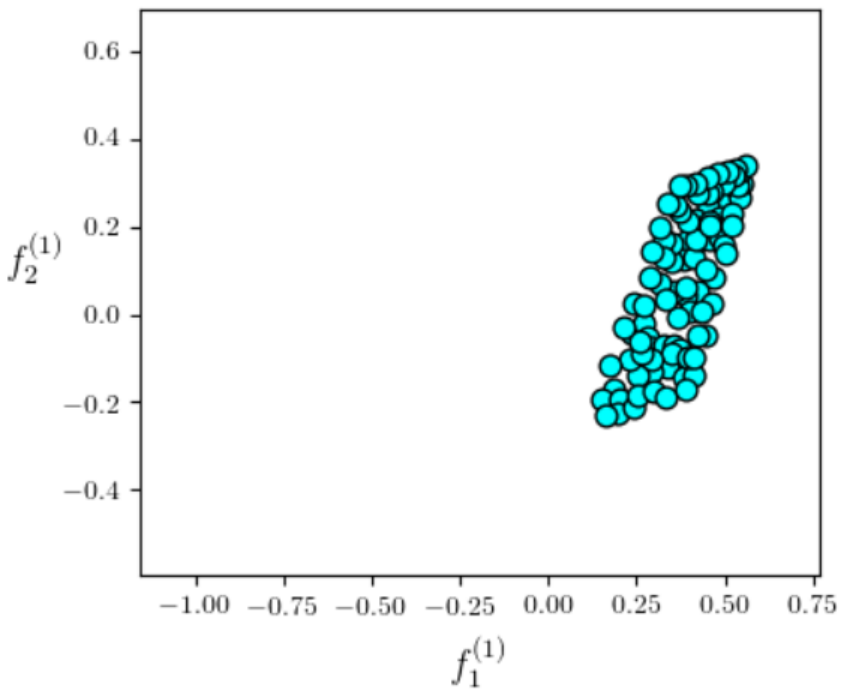
We run 5,000 steps of gradient descent to minimize loss (or cost) function under two different settings:

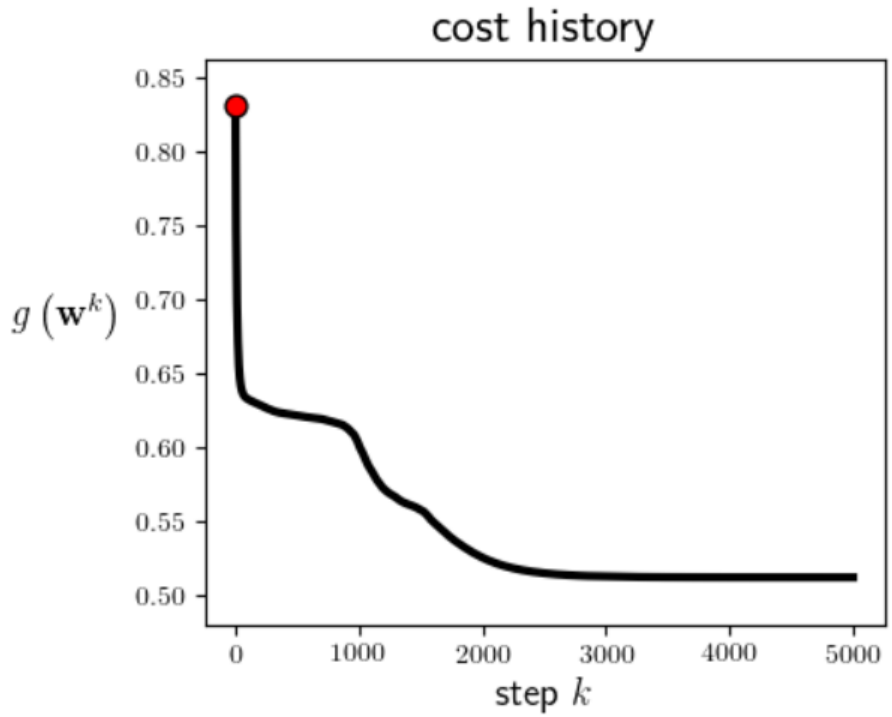
1. without batch normalization
2. with batch normalization

We will show the evolution of the loss function as a fnction of the number or iterations.
We will also show the evolution of the units $\{f_1^{(1)}(x_p), f_2^{(1)}(x_p)\}_{p=1}^{P}$

Batch normalization - Example

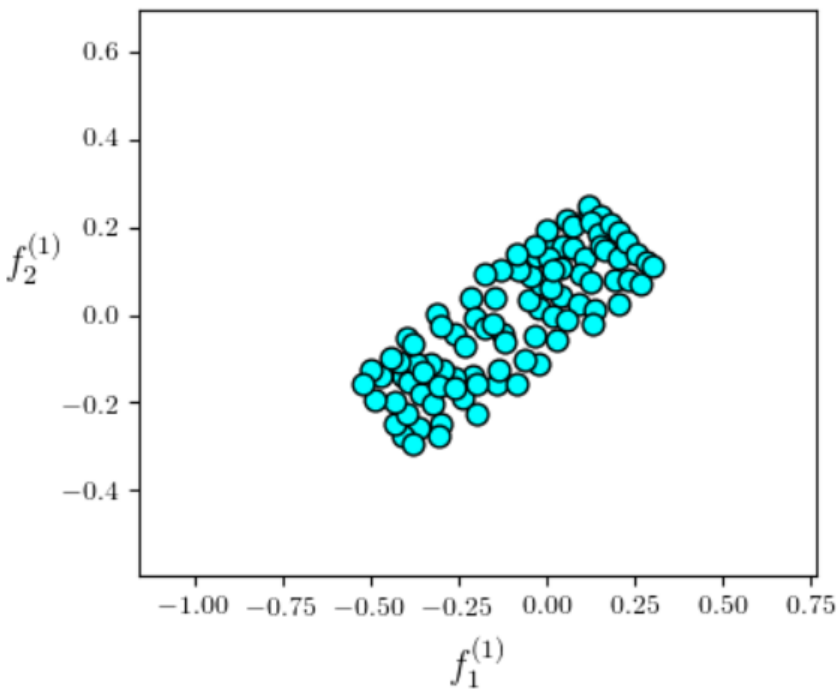
Case 1: no batch normalization

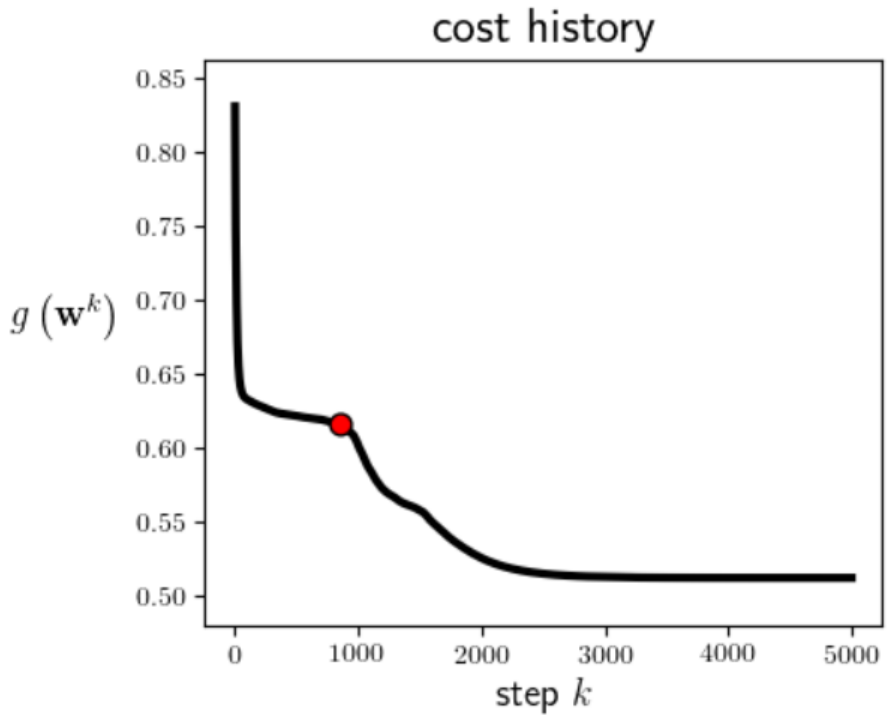




Batch normalization - Example

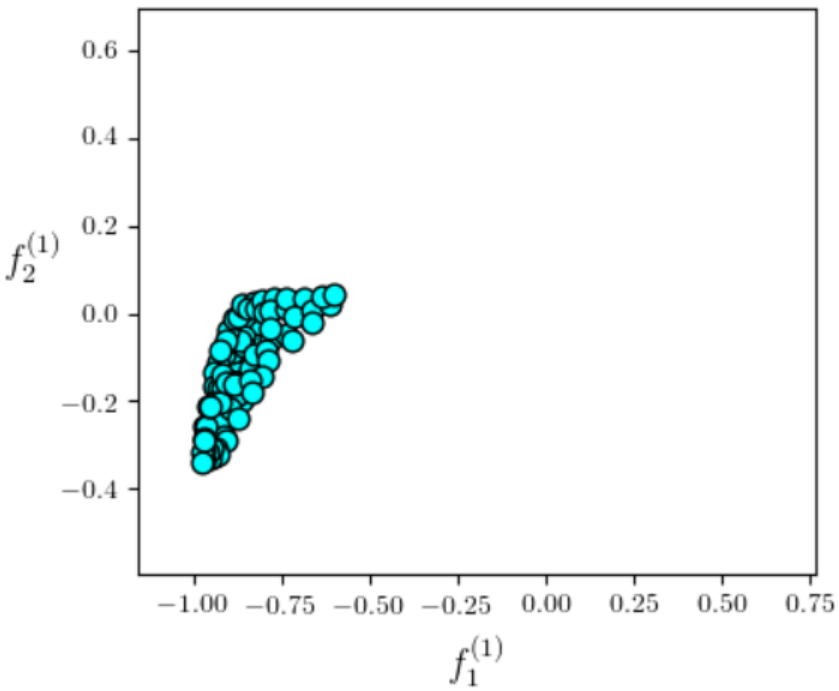
Case 1: no batch normalization

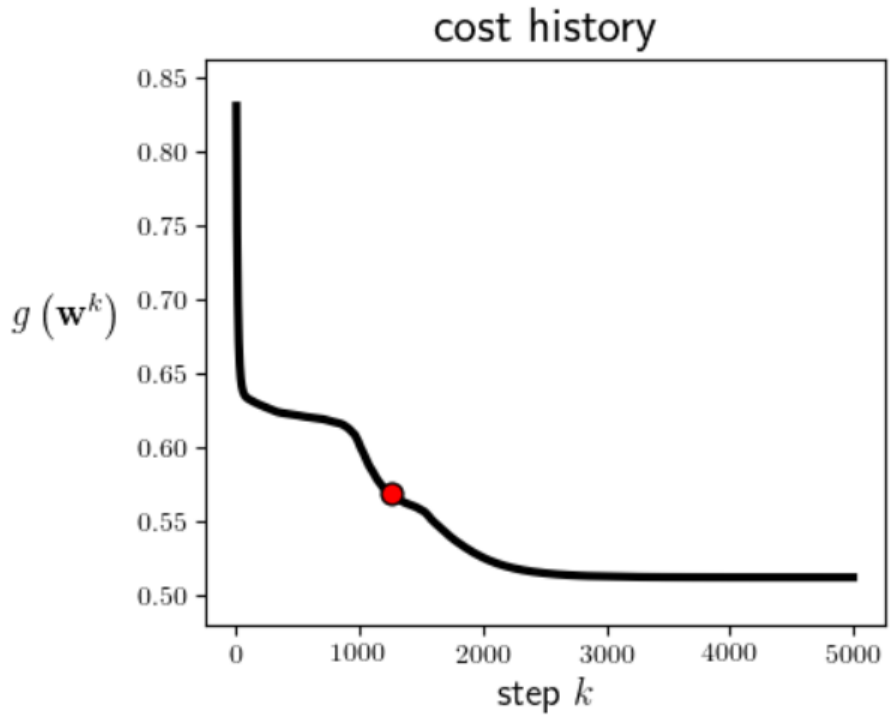




Batch normalization - Example

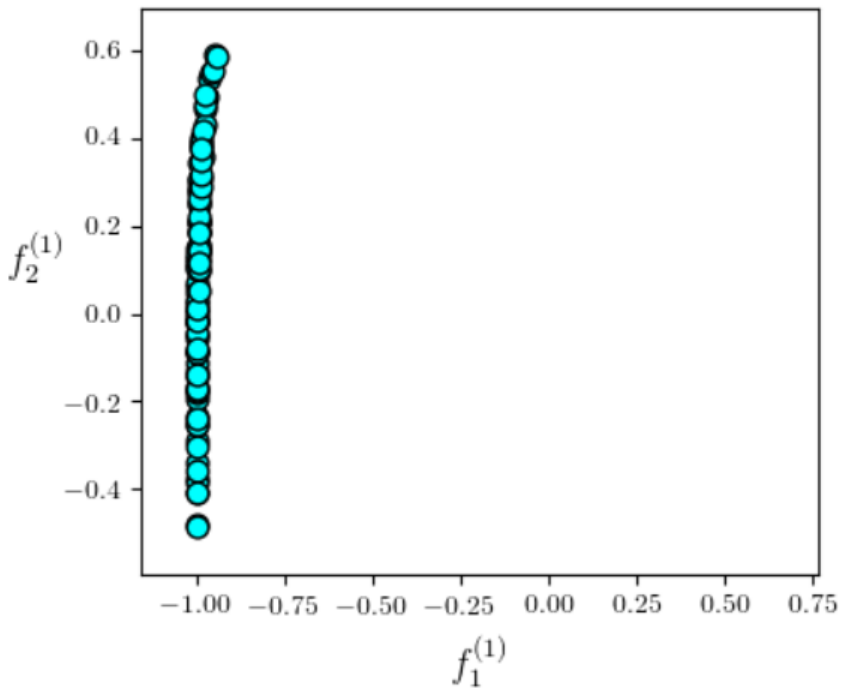
Case 1: no batch normalization

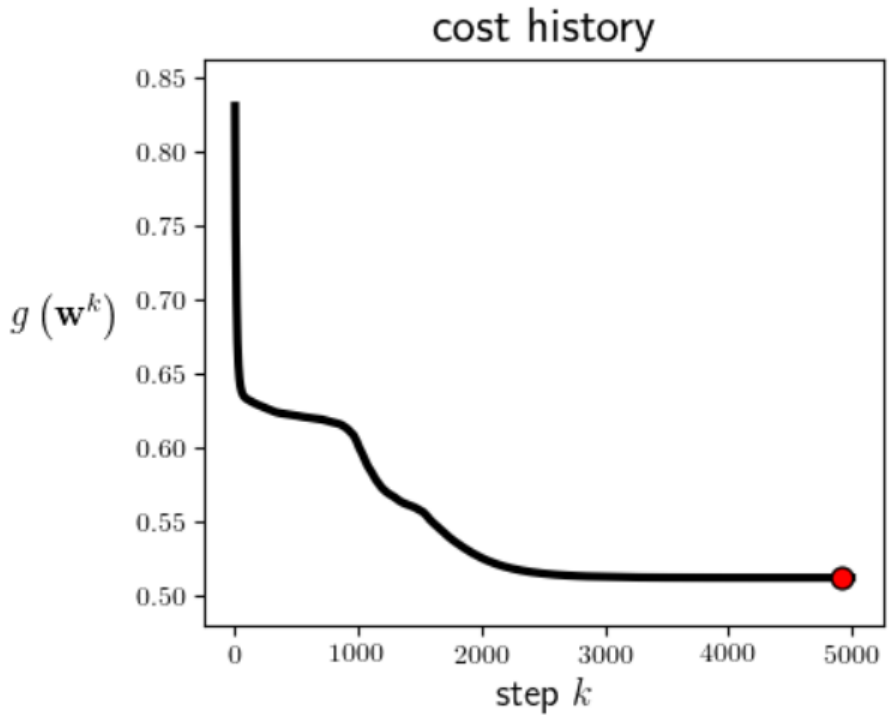




Batch normalization - Example

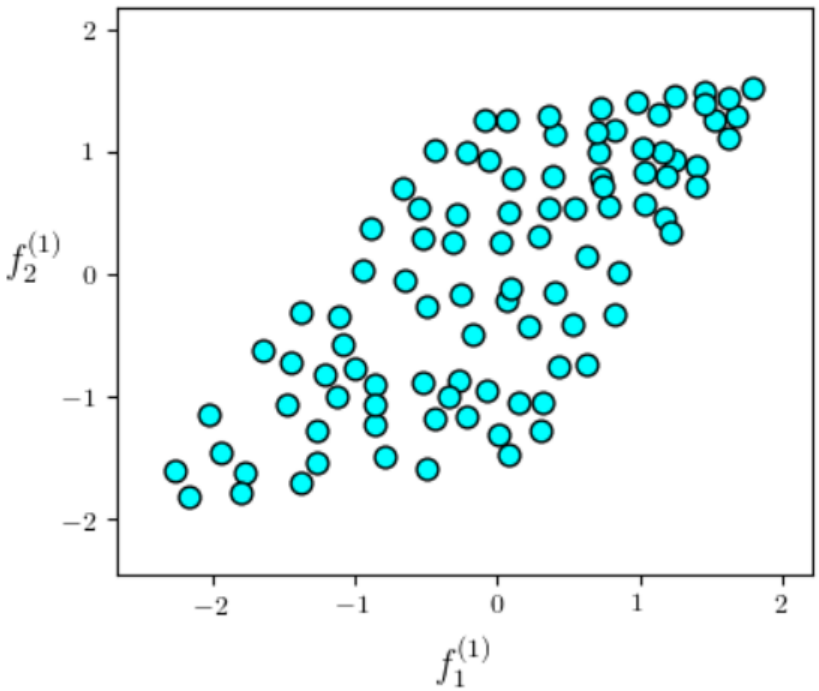
Case 1: no batch normalization

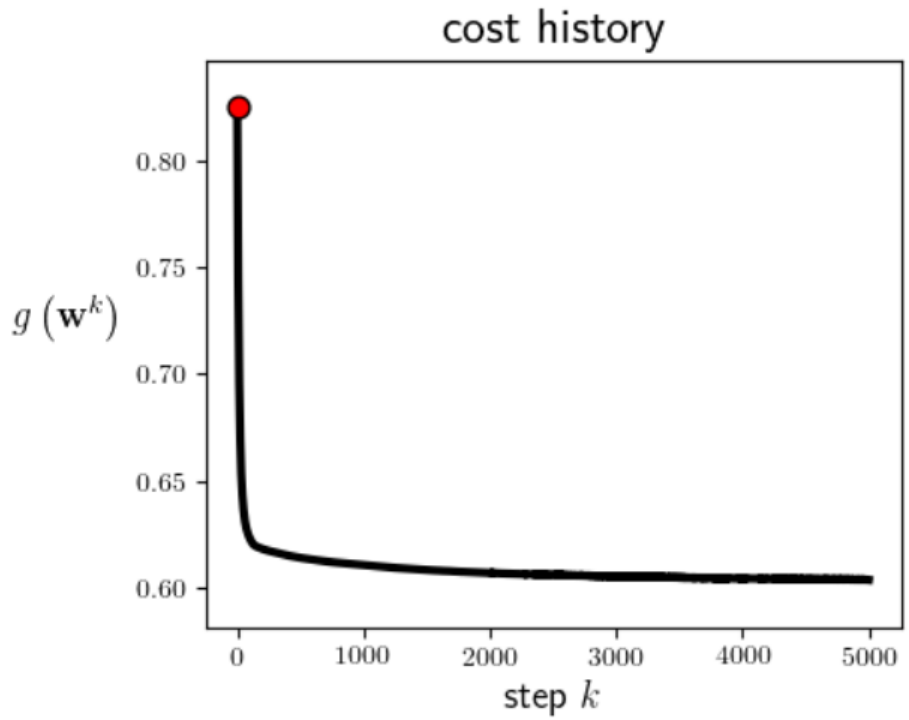




Batch normalization - Example

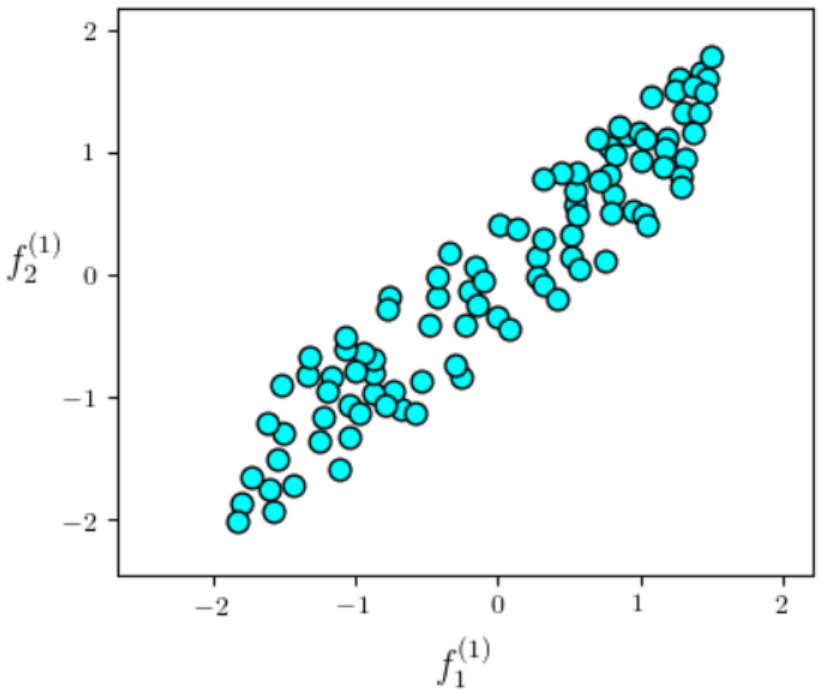
Case 1: batch normalization

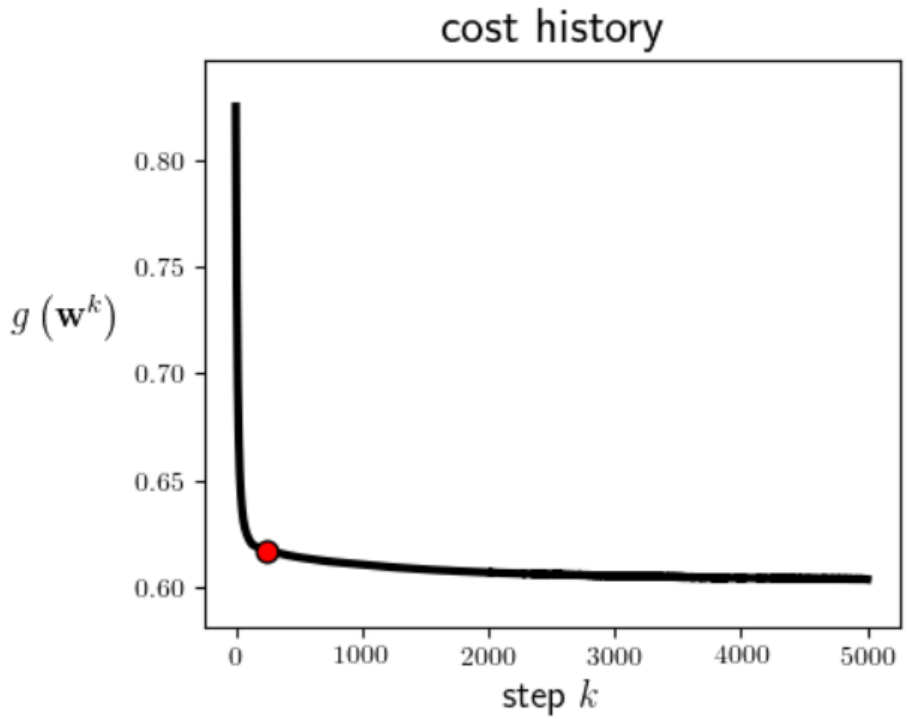




Batch normalization - Example

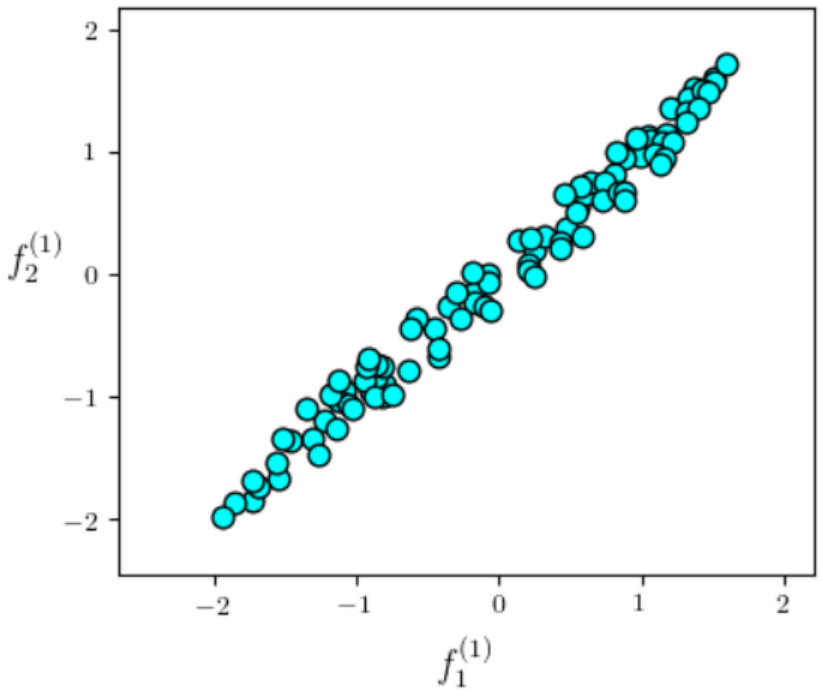
Case 1: batch normalization

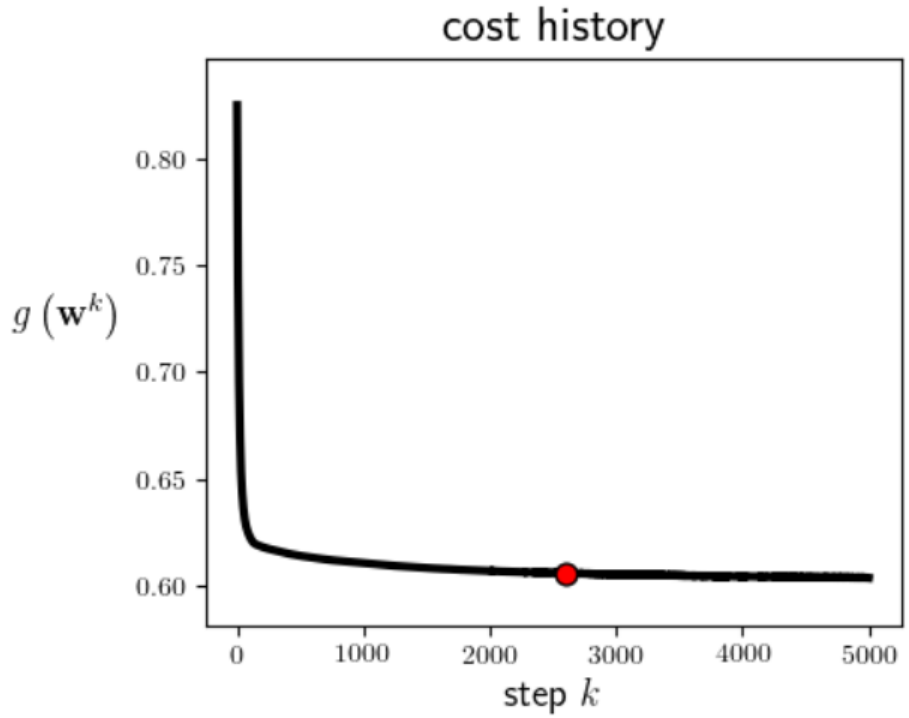




Batch normalization - Example

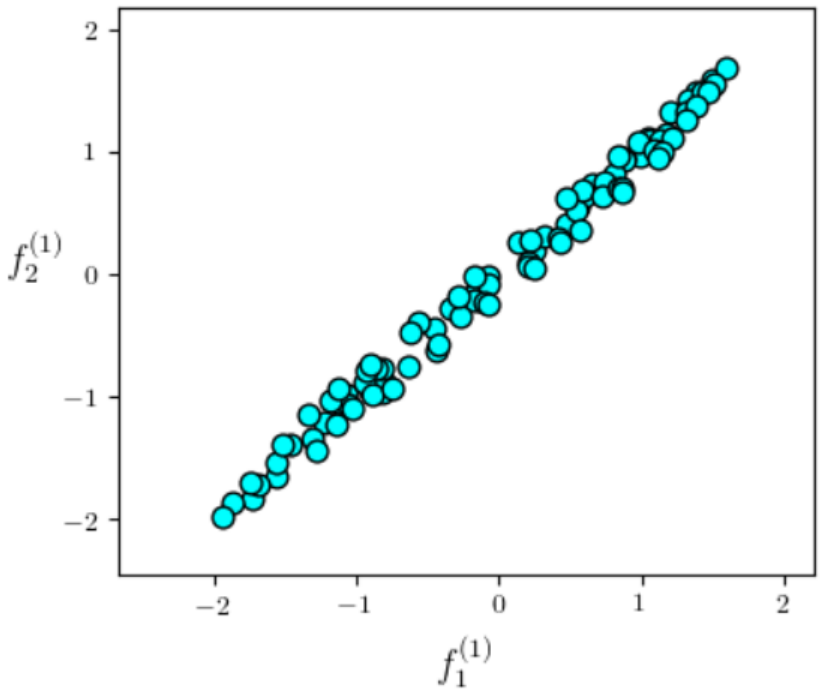
Case 1: batch normalization

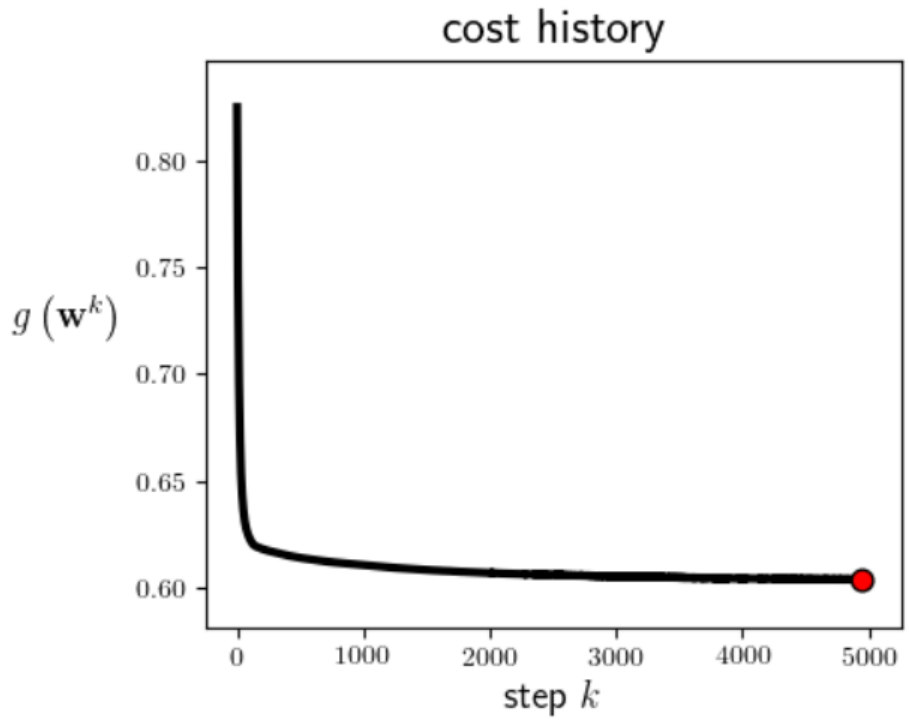




Batch normalization - Example

Case 1: batch normalization





Batch normalization - Example

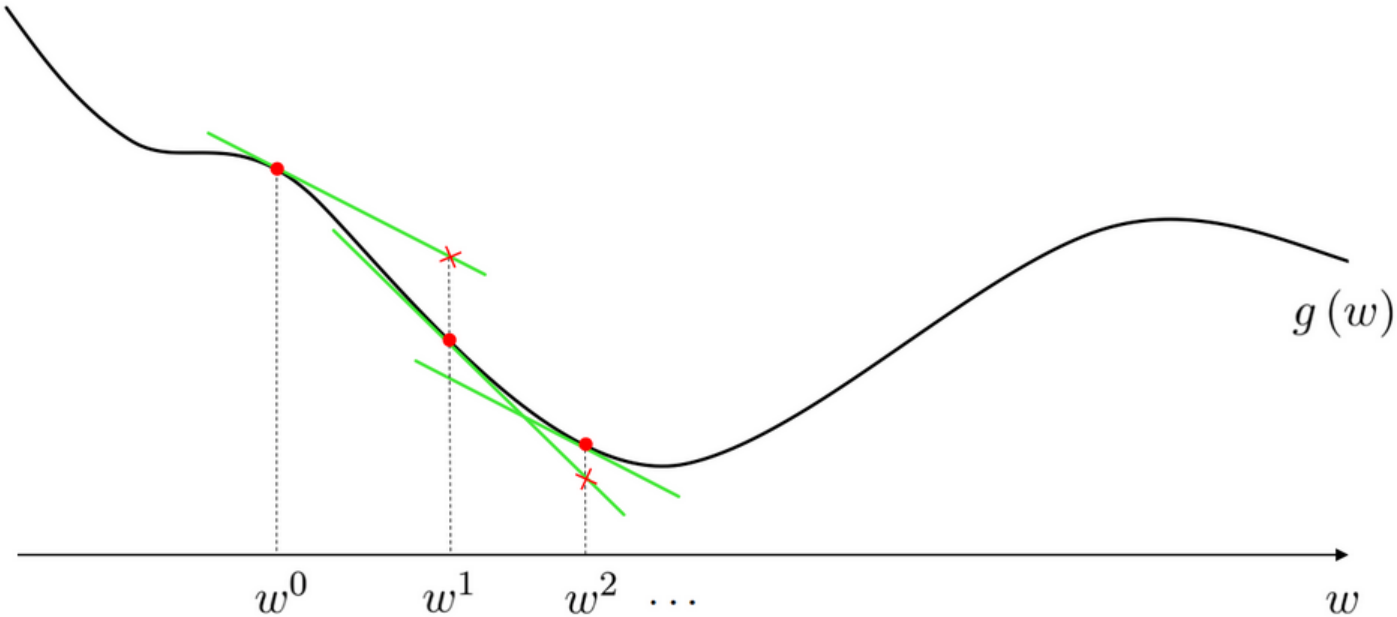
The example show that, in the absence of batch normalization, the distributions of the outputs $\{f_1^{(1)}(x_p), f_2^{(1)}(x_p)\}_{p=1}^P$ changes dramatically as the gradient descent algorithm progresses.

This sort of shifting distribution negatively effects the speed at which gradient descent can minimize the loss function.

When batch normalization is applied, the distribution of the outputs $\{f_1^{(1)}(x_p), f_2^{(1)}(x_p)\}_{p=1}^P$ is considerably more stable and the gradient descent converges more rapidly.

Gradient descent - Normalization

Recall that the **gradient descent** algorithm is a local optimization method where - at each step - we employ the negative gradient as our descent direction.



To find the local minima of a differentiable multivariate function g, starting from a point w^0, we recursively update the variable w as

$$w^{k+1} = w^k - \alpha \nabla(g(w^k))$$

the negative gradient at each step determines the direction to travel in next.

Gradient descent - Normalization

How much we end up traveling in the direction of negative gradient is determined by two factors:

1. the steplength parameter α
2. the magnitude of the gradient

This shows that we cannot fully control how much we travel at each step by tuning the step-length parameter α alone

Gradient descent - Normalization

To provide full control on how much we travel at each step, we can **normalize the gradient** by dividing off its magnitude to get the unit-length descent vector.
This operation defines the **normalized gradient descent step**

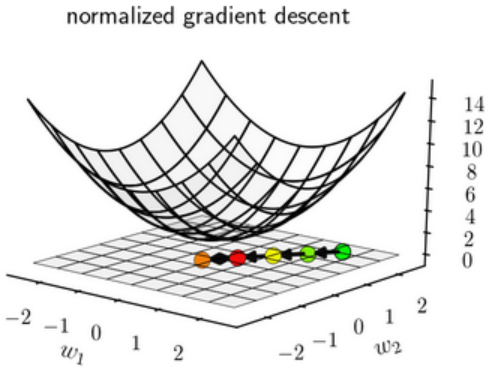
$$w^{k+1} = w^k - \alpha \frac{\nabla(g(w^k))}{\|\nabla(g(w^k))\|}$$

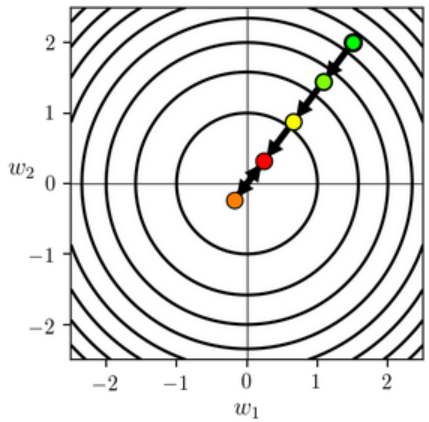
where, at each iteration, the step-length is now exactly equal to α, so the step-length can be easily tuned during training.

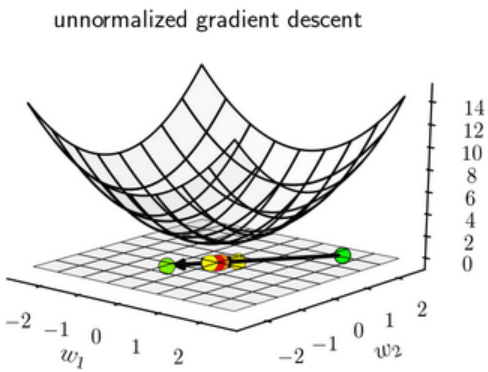
Note that dividing the negative gradient by its magnitude does not change its direction.

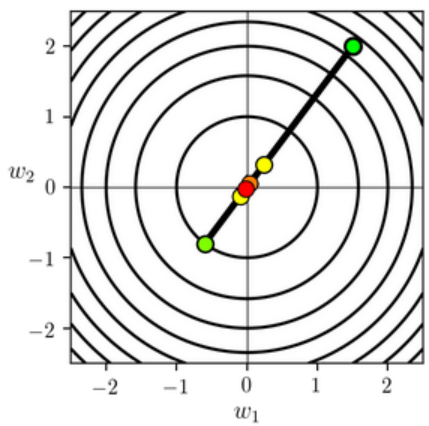
Gradient descent - Normalization

Example: We compare normalized and unnormalized gradient descent for the function $g(w) = w^2$, $w \in \mathbb{R}^2$.









In both cases, we use the same initial point and step-length parameter ($\alpha = 0.1$).

Gradient descent - Momentum

A common problem negatively affecting gradient descent in practical problem is when minima lie in a long narrow valley. In this case, convergence of the gradient descent step to the minima may become very slow.

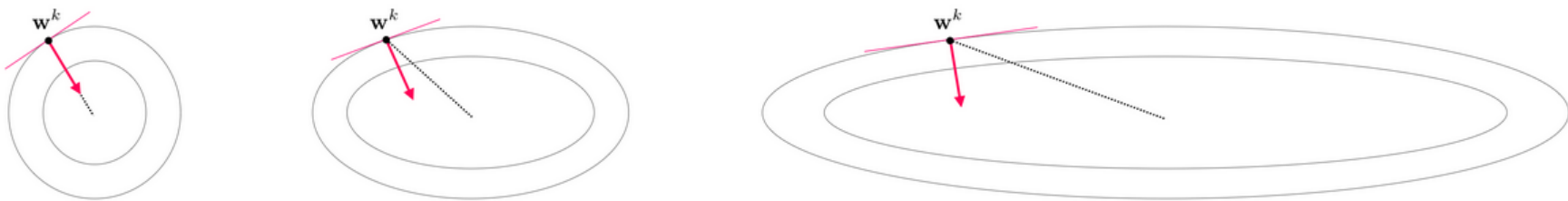


Figure shows that as the contours of a two-dimensional quadratic (think of a paraboloid) get more elongated, the gradient descent direction (red) diverges from the optimal direction (dashed black) connecting w^k to the desired minimum located in the center.

Gradient descent - Momentum

This problem can be addressed using a heuristic approach consisting of adding a **momentum term** to the standard gradient descent.

The momentum term is a weighted difference of the subsequent $k-1$-th and k-th gradient steps, that is

$$\beta(w^k - w^{k-1})$$

for some $\beta > 0$.

The momentum term has the effect of reducing the zig-zagging effect of regular gradient descent by tilting back the gradient steps (that move perpendicular to the contours of the cost function) towards the center where the minimum lies.

Gradient descent - Momentum

Figure shows that the addition of the momentum term (dashed black arrow) pushes the gradient toward the function's minimum, thus ameliorating the zig-zagging of the standard gradient descent (red arrow).

Gradient descent - Momentum

Example: We will show how momentum is used to speed up the minimization of a quadratic function using gradient descent.

For $w \in \mathbb{R}^2$, a general quadratic function has the form

$$g(w) = a + b^t w + w^t C w$$

where a is a scalar, b is a vector and C is a matrix.

We choose $a = 0$, $b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $C = \begin{pmatrix} 1 & 0 \\ 0 & 12 \end{pmatrix}$

We run gradient descent with momentum ($\beta = 0.3$) and without momentum ($\beta = 0$) with the step-length parameter set to $\alpha = 0.08$ in both cases.

Gradient descent - Momentum

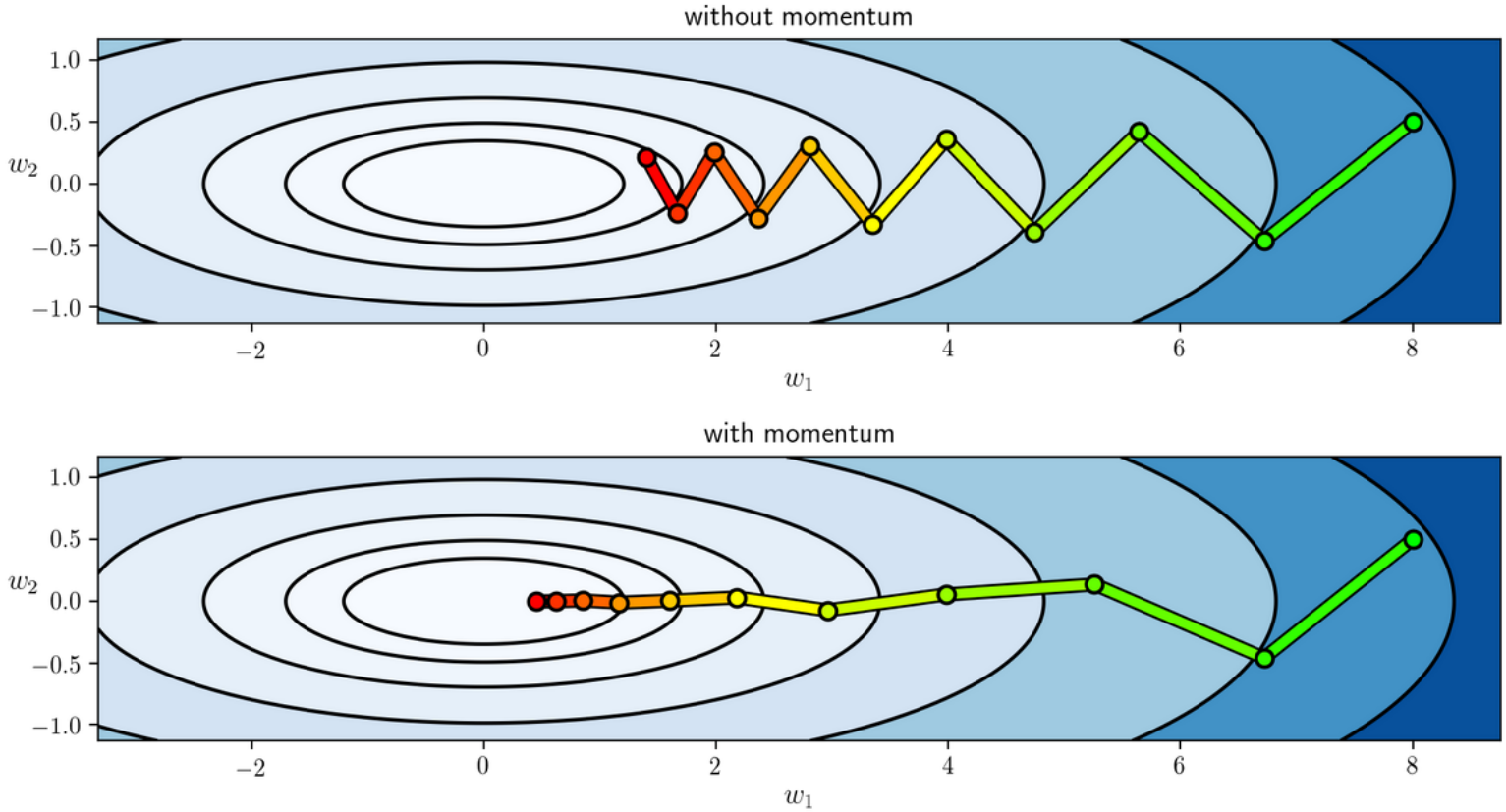


Figure shows that the addition of the momentum term (bottom panel) corrects the zig-zagging effect of standard gradient, achieving a better solution with the same number of iterations.

Gradient descent - Momentum

We repeat the gradient descent numerical experiment using again a quadratic function. This time we choose parameters $a = 0$, $b = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ and $C = \begin{pmatrix} 1 & 0 \\ 0 & 12 \end{pmatrix}$

We run gradient descent with momentum ($\beta = 0.7$) and without momentum ($\beta = 0$) with the step-length parameter set to $\alpha = 0.08$ in both cases.

Gradient descent - Momentum

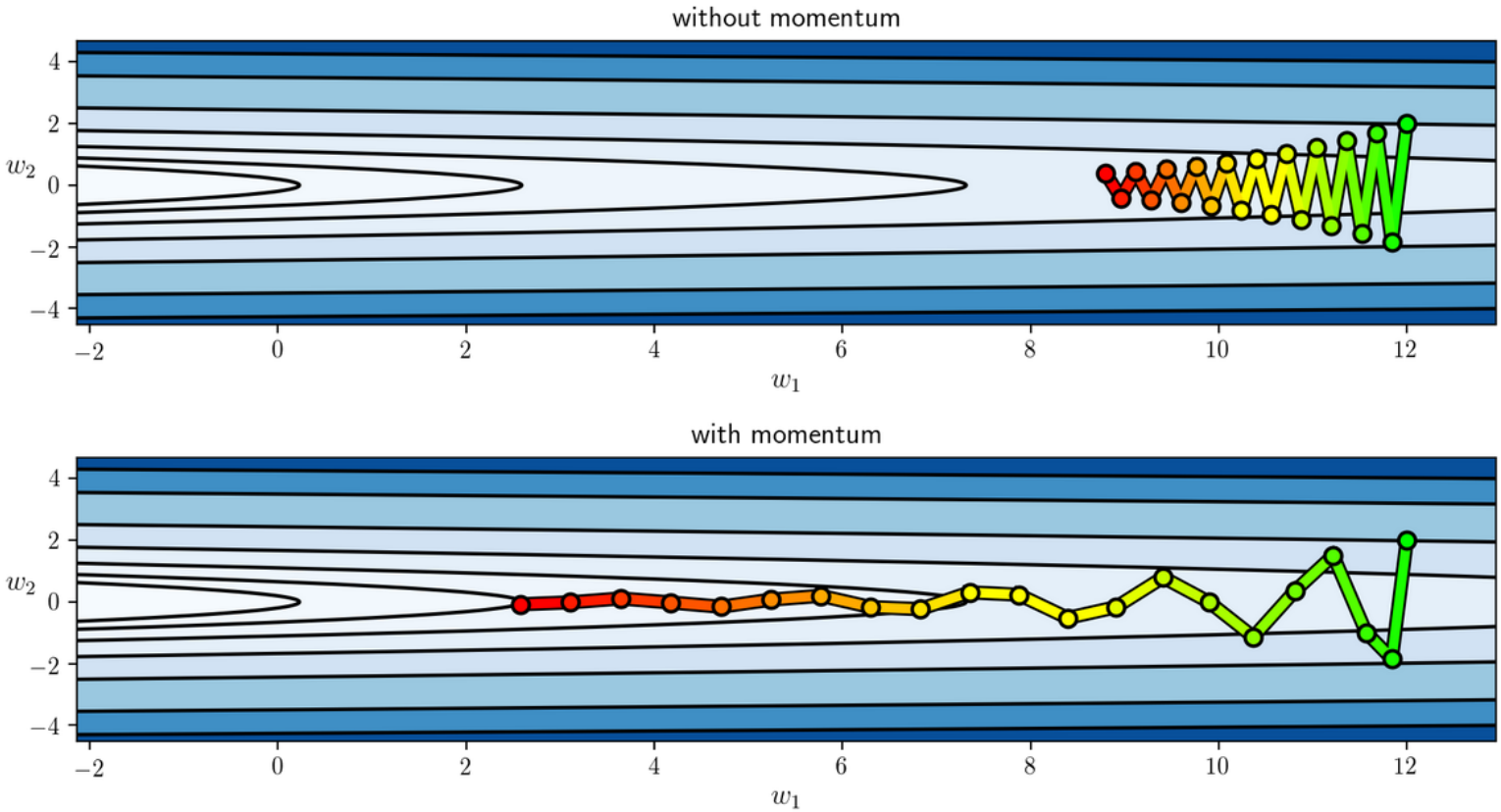


Figure again shows that the addition of the momentum term (bottom panel) corrects the zig-zagging effect of standard gradient, achieving a better solution with the same number of iterations.

Regularization - Combating non-convexity

Composition of nonlinear functions makes the loss function associated with feed-forward neural networks **non-convex**.

This means that one can no longer randomly initialize gradient descent (or any of its variants) and be sure the algorithm eventually finds the global minimum, as is the case with convex functions. In general, different random initializations may lead to different local minima.

This theoretical inconvenience does not preclude the use of non-convex loss functions, but motivates a variety of engineering fixes to the loss function, the optimizing algorithm, or both, which in practice allows us the effective use of non-convex loss in deep learning models.

Regularization - Combating non-convexity

A common regularizer to ameliorate non-convexity is the addition of a simple convex function to slightly convexify the non-convex loos function

This can improve numerical optimization by avoiding poor solutions near its saddle points or flat areas.

One of the most common regularizers used in practice is the squared ℓ^2 norm. In this case, the loss function $g(w)$ is replaced by

$$g(w) + \lambda \|w\|_2^2,$$

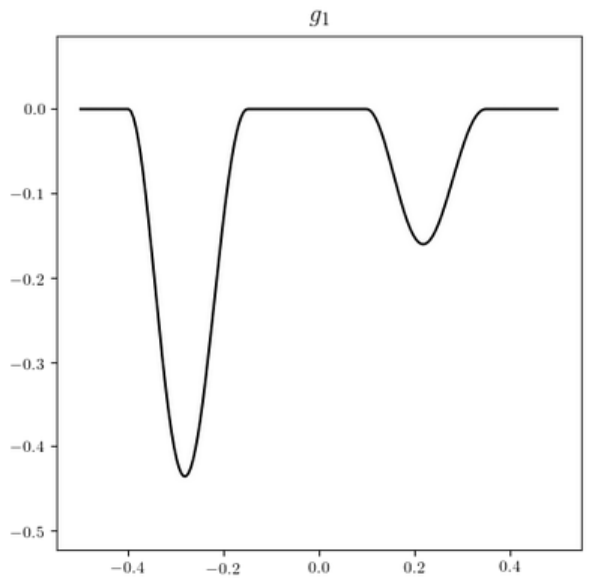
where $\lambda > 0$ is a parameter controlling the trade-off between the original loss function and the regularizer.

Regularization - Combating non-convexity

Example: Effect of regularization on a non-convex function.

We consider the non-convex loss function

$$g_1(w) = \max\left(0, \frac{1}{2}e^{-w}\sin(4\pi(w-0.1))\right)^2$$



Problem: if gradient descent is initialized at any point lying in the flat regions it will immediately halt.

Regularization - Combating non-convexity

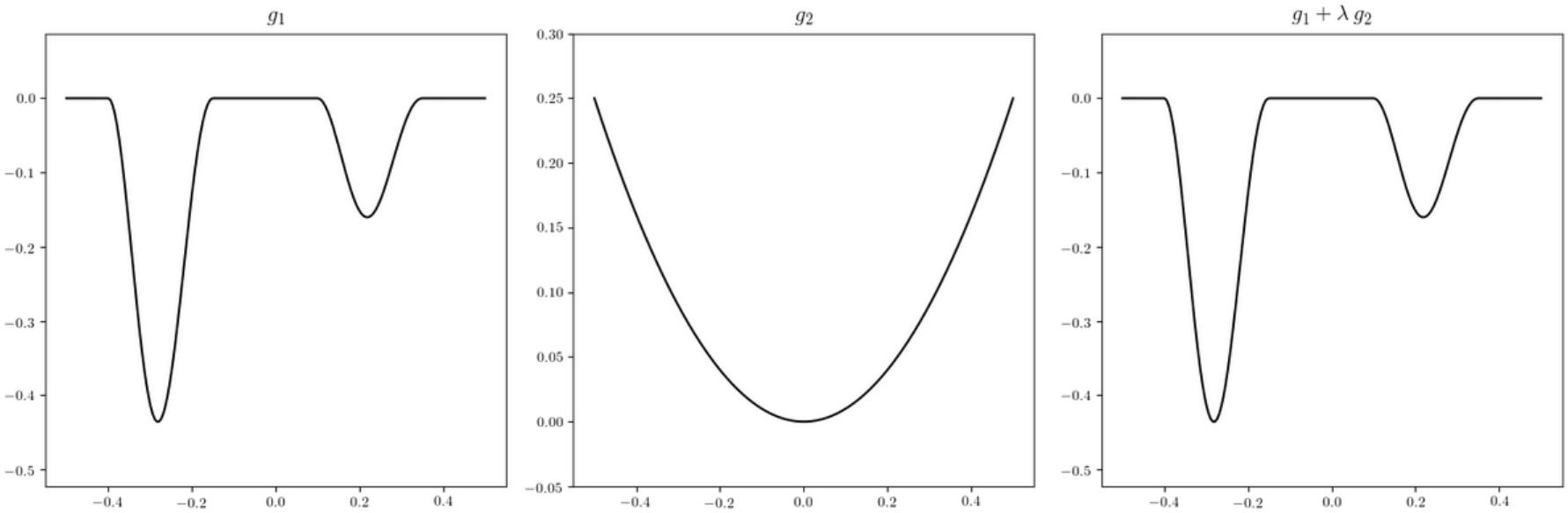
To improve the convergence of gradient descent, we add an ℓ^2-norm regularizer term $g_2(w) = \|w\|_2^2$ to the loss function

So we have the regularized loss function

$$g_1(w) + \lambda g_2(w) = \max\left(0, \frac{1}{2}e^{-w}\sin(4\pi(w-0.1))\right)^2 + \lambda\|w\|_2^2$$

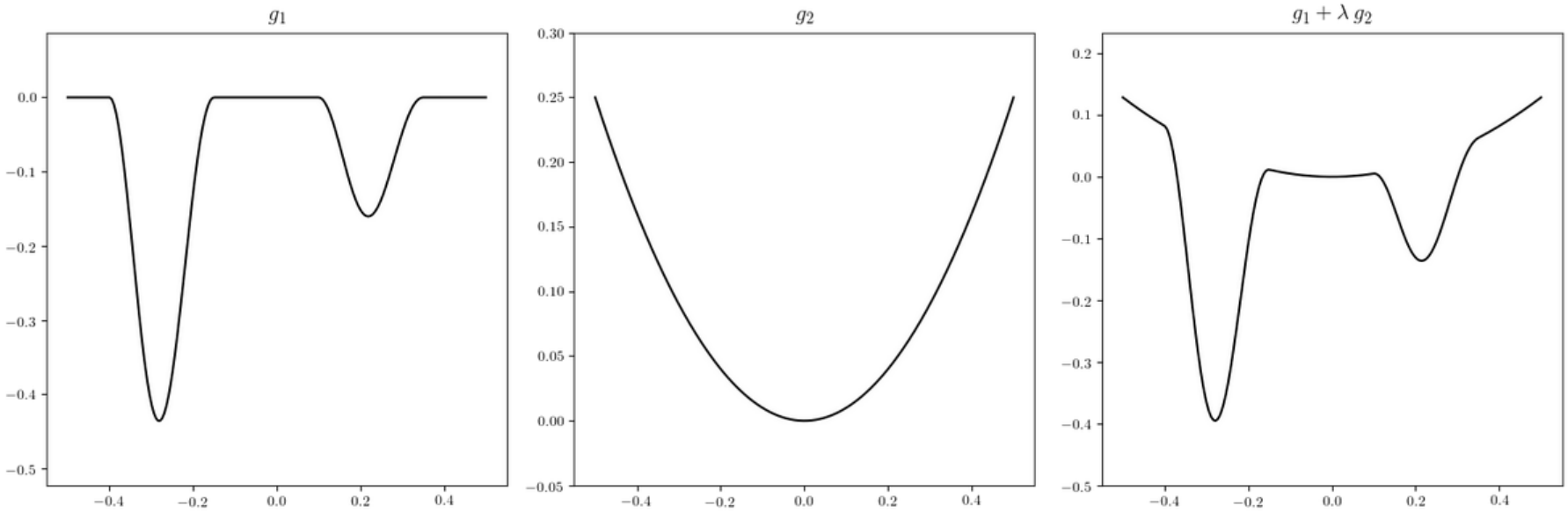
We will show how this changes the properties of the loss function when λ is varied in the interval $[0, 1]$

Regularization - Combating non-convexity



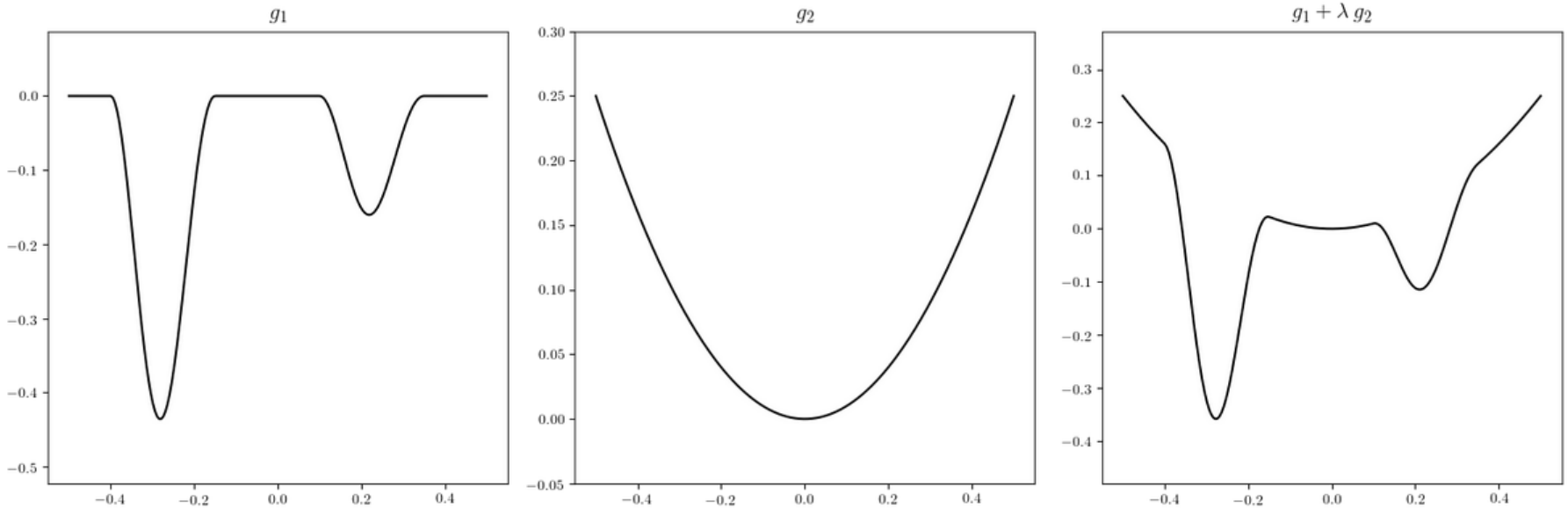
$$g_1(w) + \lambda g_2(w), \quad \lambda = 0$$

Regularization - Combating non-convexity



$$g_1(w) + \lambda g_2(w), \quad \lambda = 0.5$$

Regularization - Combating non-convexity



$$g_1(w) + \lambda g_2(w), \quad \lambda = 1$$

Stochastic and mini-batch gradient descent

Stochastic and **mini-batch gradient descent** provide an alternative to standard (or batch) gradient descent when applied to large datasets that require less storage space it in active memory and may converge faster.

The key observation is that (in most cases) the loss function can be written as a sum of individual costs over each data point of the training set $(x_p, y_p)_{p=1,\ldots,P}$

$$g(w) = \sum_{p=1}^{P} h(w, x_p, y_p)$$

For instance, in the case of squared error loss we have that

$$h(w, x_p, y_p) = (x_p^t w - y_p)^2$$

Stochastic gradient descent

The observation that the loss function can be decomposed over individual data points implies that we can write the gradient as a summation of the gradients of each summand term

$$\nabla g(w) = \nabla \left(\sum_{p=1}^{P} h(w, x_p, y_p) \right) = \sum_{p=1}^{P} \nabla h(w, x_p, y_p)$$

Hence, when minimizing a loss function via gradient descent, we can write

$$w^{k+1} = w^k - \alpha_{k+1} \nabla (g(w^k)) = w^k - \alpha_{k+1} \sum_{p=1}^{P} \nabla h(w^k, x_p, y_p)$$

Note that the steplength α_{k+1} may change at each step.

Stochastic gradient descent

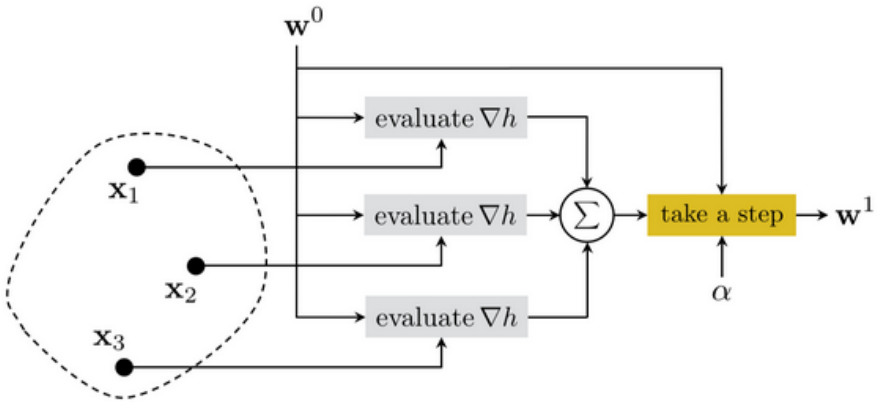
$$w^{k+1} = w^k - \alpha_{k+1} \nabla(g(w^k)) = w^k - \alpha_{k+1} \sum_{p=1}^{P} \nabla h(w^k, x_p, y_p)$$

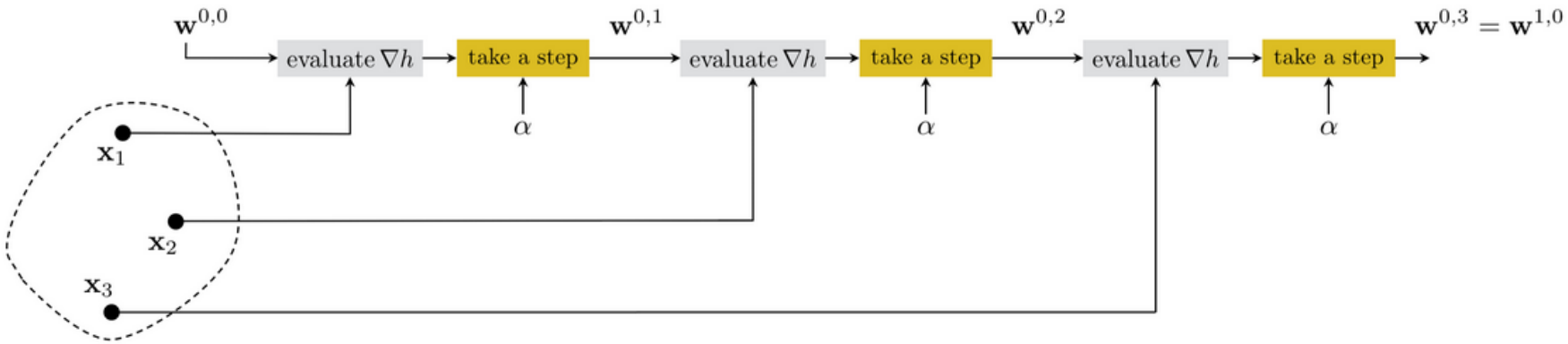
Remarks Formula shows we can (1) load a single data point at a time in memory, (2) compute the gradient of the cost with respect to that data point, (3) add the result to a container, (4) discard the data point to free up the memory, and (5) move to the next data point. After all data points are visited once, we can then take a gradient step using what is stored in the gradient container.

This scheme works for any large dataset. One just needs to wait for P evaluations of the gradient before taking a step.

Rather than waiting for all individual gradient terms to be computed, **stochastic gradient descent** takes a step as soon as a gradient term becomes available.

Stochastic gradient descent





Schematic comparison of first iteration of standard (top) and stochastic (bottom) gradient descent, using a dataset with $P = 3$ points.

Stochastic gradient descent

The k-th iteration of stochastic gradient descent, sometimes called an **epoch**, consists of P sequential point-wise gradient steps written as

$$w^{k,p} = w^{k,p-1} + \alpha_{k+1} \sum_{p=1}^{P} \nabla h(w^{k,p-1}, x_p, y_p), \quad p = 1, \ldots, P.$$

The double superscript $w^{k,p}$ reads "the p-th individual gradient step of the k-th stochastic gradient descent iteration.

Vocabulary note:

- In the standard (or batch) gradient descent we use **"step"** and **"iteration"** interchangeably, i.e., each iteration consists of one full gradient step in all P data points simultaneously as shown in Slide 45.
- Conversely, with the stochastic gradient descent, we refer to a single **"iteration"** or **"epoch"** as consisting of all P individual gradient steps, one in each data point, executed sequentially for $p = 1, \ldots, P$.

Mini-batch gradient descent

Implementation question: How do you choose the **steplength** α_k for stochastic gradient descent?

A diminishing steplength provably guarantees the convergence of the stochastic gradient method provided:

1. The steplength must diminish as the number of iterations increases: $\alpha_k \to 0$ as $k \to \infty$.
2. The sum of the steplengths is not finite: $\sum \alpha_k = \infty$.

Common choices of steplength with the stochastic gradient method include $\alpha_k = \frac{1}{k}$ or $\alpha_k = \frac{1}{\sqrt{k}}$

In some cases, one may successfully use other steplength rules such as fixed steplengths.

Mini-batch gradient descent

Mini-batch gradient descent is a modification of stochastic gradient descent.

The core idea starts with the observation that we can decompose the cost function defined over P individual data points as

$$g(w) = \sum_{p=1}^{P} h(w, x_p, y_p) = \sum_{j=1}^{J} \sum_{p \in P_j} h(w, x_p, y_p)$$

where $P_1, \ldots, P_J$ is a partition of the set $\{1, \ldots, P\}$ into J index sets.
These index sets divide the dataset into J disjoint subsets, each called a **mini-batch**; that is, every data point belongs to one and only one mini-batch.

Mini-batch gradient descent

Using this representation, the gradient is now decomposed over each mini-batch (as opposed to each data point):

$$\nabla g(w) = \nabla \left(\sum_{j=1}^{J} \sum_{p \in P_j} h(w, x_p, y_p) \right) = \sum_{j=1}^{J} \nabla \left(\sum_{p \in P_j} h(w, x_p, y_p) \right)$$

Mini-batch gradient descent is then the algorithm wherein we take gradient steps sequentially using each mini-batch.

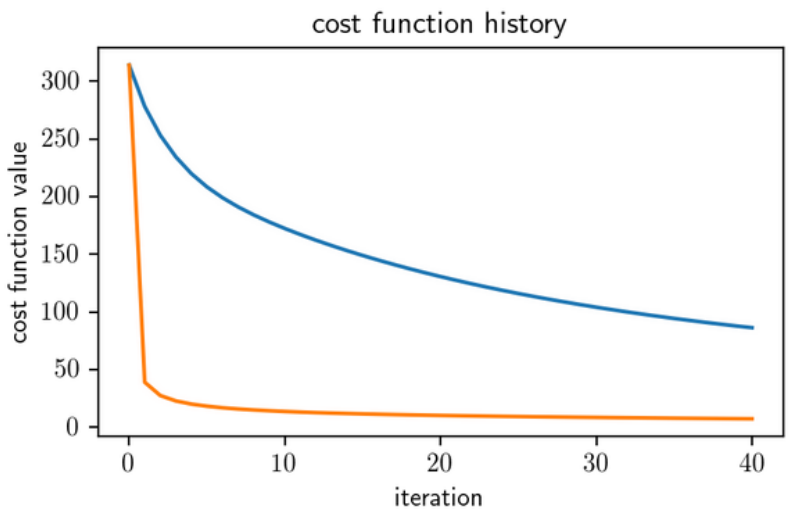
Typically one wants all mini-batches to have the same size - a parameter called the **batch size**.

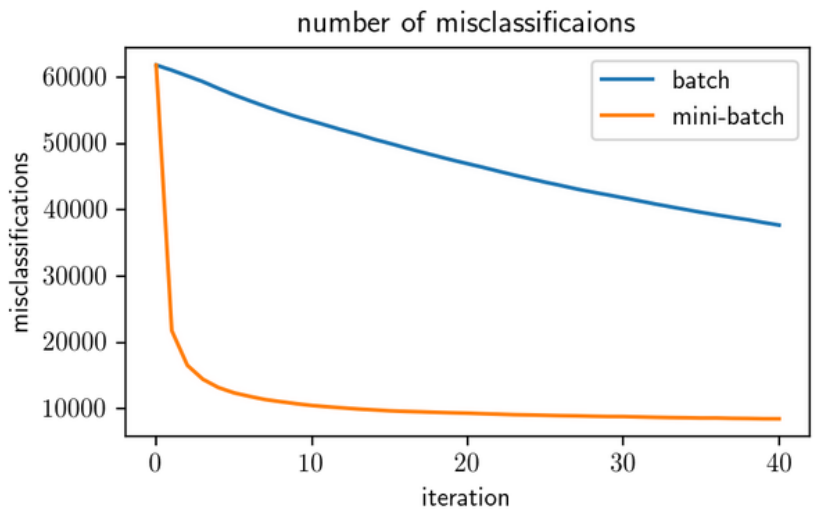
Note: a batch size of 1 turns mini-batch gradient descent into stochastic gradient descent, whereas a batch size of P turns it into the standard (or batch) gradient descent.

Mini-batch gradient descent

Example. Multiclass classification using a MLP on the MNIST dataset consisting of $P = 70,000$ images of hand-written digits $\{0, 1, \ldots, 9\}$.

We compare 40 iterations of standard stochastic and mini-batch (batch size =500) gradient descent using the multi-class softmax cost function; experiments were run with the same initialization and fixed steplength.





Mini-batch gradient descent

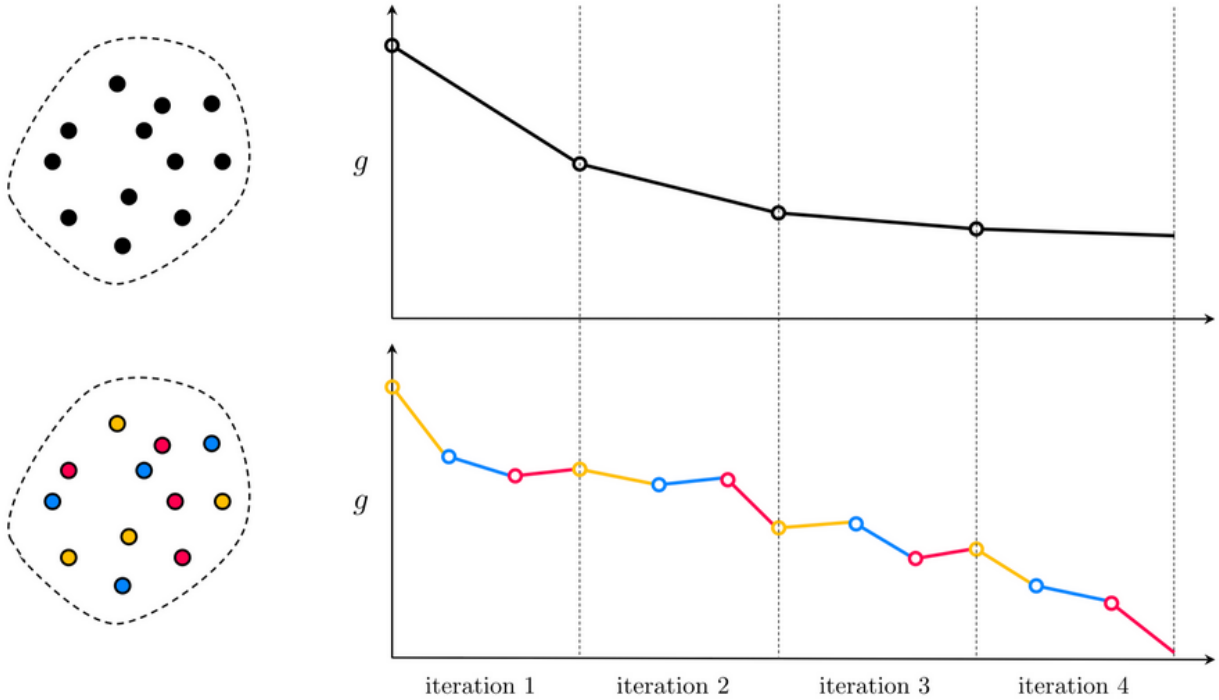
Discussion.

The result in the Example is indicative of a major computational advantage of stochastic/mini-batch gradient descent over the standard batch version for dealing with large datasets.

When initialized away from a point of convergence, the stochastic/mini-batch methods tend in practice to progress much faster towards a solution.

Because moderately accurate solutions (provided by a moderate amount of minimization of a cost function) tend to perform reasonably well in machine learning applications, and because with large datasets a random initialization will tend to lie far from a convergent point, in many cases even a few iterations of stochastic/mini-batch gradient descent can provide a relatively good solution.

Mini-batch gradient descent

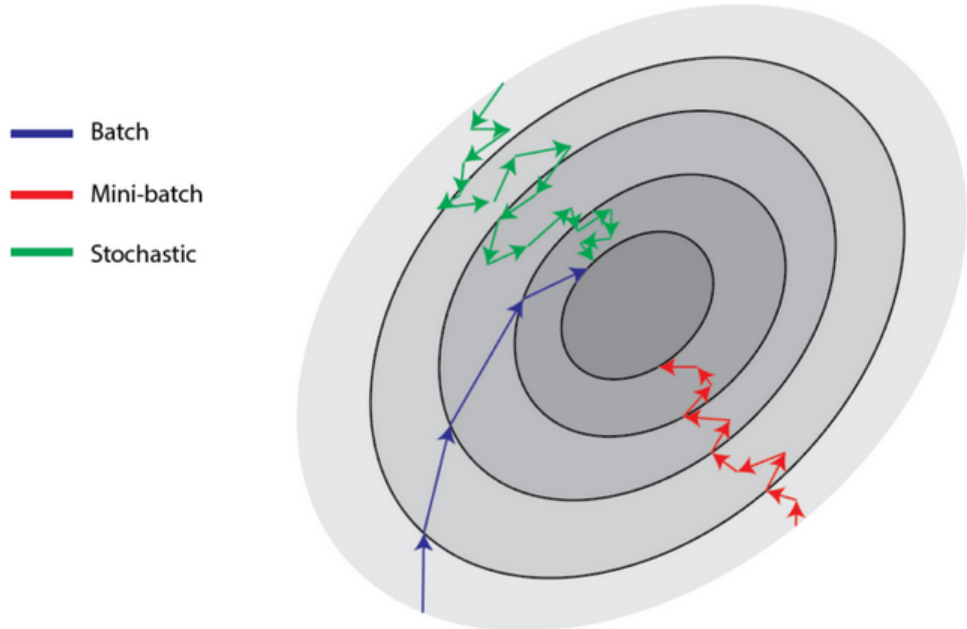


The figure shows that, taking a large number of 'imperfect' mini-batch steps, is typically more effective in reducing the cost function evaluation than taking a considerably smaller number of 'perfect' gradient steps via the standard (batch) stochastic descent method.

Mini-batch gradient descent

Implementation question: How do you choose **batch size** b for mini-batch gradient descent?

It must be $b \leq P$, where P the size of the training set.
Recall: for $b = P$, we have standard stochastic (batch) gradient descent and, for $b = 1$, we have stochastic gradient descent.



Note: case $b = P$, large P, may be impossible due to memory load.

Mini-batch gradient descent

While there is no formula for choosing the "optimal" b, it is generally observed that:

- Gradient with small batch size b oscillates much more compared to larger batch size and take longer to converge.
- For a non-convex loss landscape (as in the case of MLP), this oscillatory behavior helps come out of the local minima.
- Larger batches do fewer and coarser search steps for the optimal solution, and so by construction, will be less likely to converge on the optimal solution.
- Models trained with small batch sizes are often observed to generalize better.

Common practise is to choose the batch size as a power of two, in the range between 16 and 512.
$b = 16$ or 32 is a rule of thumb for the *initial choice*.

Steepest descent algorithm

Steepest descent algorithms are associated with different normalization of the gradient that may significantly impact the form descent direction.

The derivation of the gradient descent direction of a multivariate function $g(w)$ at a point v follows from the computation of the tangent hyperplane to g at this point:

$$h(w) = g(v) + \nabla g^t(v)(w - v)$$

It follows that the **descent direction** is given by the negative gradient as $-\nabla g(v)$ or, in the normalized form, as $-\frac{\nabla g(v)}{\|\nabla g(v)\|_2}$.

Steepest descent algorithm

The descent direction can be derived more formally as an optimization problem.

We want to find the unit-length direction d that provides the smallest evaluation on the hyperplane

$$g(v) + \nabla g^t(v)(w - v) = g(v) + \nabla g^t(v)d$$

This leads to the constrained minimization problem

$$\arg\min_d \nabla g^t(v)d \quad \text{subject to } \|d\|_2 = 1$$

whose solution is the **steepest descent direction.**

In this case, with an ℓ^2 constraint, the solution is $d = -\frac{\nabla g(v)}{\|\nabla g(v)\|_2}$ which is also the **gradient descent direction**.

Steepest descent algorithm

We can choose different norms to determine a steepest descent direction.

For example, replacing the ℓ^2 with the ℓ^1 or ℓ^∞ norms to have a similar looking problems

$$\arg\min_d \nabla g^t(v)d \quad \text{subject to } \|d\|_1 = 1$$

and

$$\arg\min_d \nabla g^t(v)d \quad \text{subject to } \|d\|_\infty = 1$$

The directions we find here will certainly be different than the ℓ^2 constrained version.
In the ℓ^1 constrained version, the solution is called **coordinate descent** solution. This leads to an algorithm that is computationally less involved than gradient descent.

Early stopping

Early Stopping consists simply in stopping the training of a neural network before a model starts to overfit.

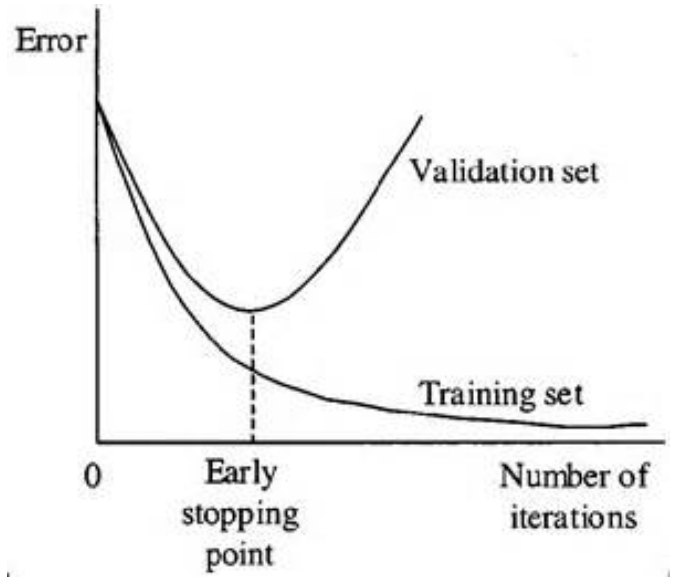
There are three main ways to apply early stopping.

1. **Training a model on a preset number of epochs.** This method is a simple. However, by running a set number of epochs, we run the risk of not reaching a satisfactory training point. With a higher learning rate, the model might possibly converge with fewer epochs, but it would require a lot of trial and error.

2. **Stop when the loss function update becomes small.** This approach is more sophisticated but assumes that the weight updates in gradient descent become significantly smaller as the model approaches minima. Usually, the training is stopped when the update becomes as small as 0.001, as stopping at this point minimizes loss and avoids unnecessary epochs. However, overfitting might still occur.

Early stopping

3. **Validation set strategy.** It requires to check how training and validation errors change with the number of epochs. Typically, the training error decreases until increasing epochs no longer impact the error. The validation error, however, initially decreases with increasing epochs, but after a certain point, it starts increasing. This is the point where a model should be early stopped.



Although the validation set strategy is the best in terms of preventing overfitting, it may take a large number of epochs.

1.6 Performance metrics

Performance metrics - Regression

Regression models have a continuous output.

So, a performance metric need to calculating some sort of distance between predicted value and ground truth.

Standard performance metrics include:

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- Root Mean Squared Error (RMSE)
- R-Squared

Performance metrics - Regression

Mean squared error (MSE) is the most popular metric used for regression problems.

It finds the average of the squared difference between the target value and the value predicted by the regression model

Given a set of ground truth values y_i, $i = 1, \ldots, N$ and the corresponding predicted values $\hat{y}_i$, $i = 1, \ldots, N$,

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Properties:

- It is differentiable, so it is easier to optimize.
- It penalizes small errors by squaring them, leading to an overestimation of how bad the model is.
- It is sensitive to outliers.

Performance metrics - Regression

Mean Absolute Error (MAE) replaced the ℓ^2 distance with the ℓ^1 distance.

It is the average of the difference (in absolute value) between the ground truth and the predicted values.

Given a set of ground truth values y_i, $i = 1, \ldots, N$ and the corresponding predicted values $\hat{y}_i$, $i = 1, \ldots, N$,

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

Properties:

- MAE is non-differentiable as opposed to MSE.
- It penalizes small errors less than MSE (no overestimation of how bad the model is).
- It is more robust towards outliers than MSE.

Performance metrics - Regression

Root Mean Squared Error (RMSE) is the square root of MSE.

It finds the square root of average of the squared difference between the target value and the value predicted by the regression model

Given a set of ground truth values y_i, $i = 1, \ldots, N$ and the corresponding predicted values $\hat{y}_i$, $i = 1, \ldots, N$,

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

Properties:

- It retains the differentiable property of MSE.
- It handles the penalization of smaller errors done by MSE by square rooting it.
- Since scale factors are essentially normalized, it is less prone than MSE to struggle in the case of outliers.

Performance metrics - Regression

R-Squared, the **Coefficient of Determination** is a measure of the quality of a linear regression model.

It explains how much (what %) of the total variation in the target y is explained by the variation in the regression line.

Given a set of ground truth values y_i, $i = 1, \ldots, N$ and the corresponding predicted values $\hat{y}_i$, $i = 1, \ldots, N$,

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

Properties:

- R^2 close to 1 means the regression was able to capture close to 100% of the variance in the target variable.
- R^2 close to 0 means the regression was able to capture close to 0% of the variance in the target variable.
- R^2 outside the range 0 to 1 occur when the model fits the data worse than the worst possible least-squares predictor.

Performance metrics - Regression

Remark.
Most authors advocate **against the use of the R^2 in nonlinear regression analysis** and recommend alternative measures.

For nonlinear regression models R^2 cannot be interpreted as the proportion of variance explained by the model.

In linear regression, R^2 compares the fit of the regression line with a horizontal line (forcing the slope to be 0.0). The horizontal line is the simplest case of a regression line, so this makes sense. With most models used in nonlinear regression, the horizontal line is not a simple case and can't be generated at all from the model with any set of parameters. So comparing the fits of the chosen model with the fit of a horizontal line does not make sense.

With nonlinear regression, R^2 can be negative.

Performance metrics - Classification

Classification models have discrete output, so a corresponding metric need to compare discrete classes in some form.

Standard classification performance metrics include:

- Accuracy
- Confusion Matrix (not a metric but a complemetary tool)
- Precision and Recall
- F1 score
- Area under Receiver Operating Characteristics (AUROC)
- Area under Precision-Recall curve

Performance metrics - Classification

Classification **Accuracy** is the simplest metric to use and implement.

It is defined as *the number of correct predictions divided by the total number of predictions*.

It is often multiplied by 100 and reported as a *percent*.

Remark. Accuracy can be a misleading metric for imbalanced data sets. Consider a sample with 95 negative and 5 positive values. Classifying all values as negative in this case gives 0.95 (or 95%) accuracy score. However, we missed all the positive values.

Performance metrics - Classification

Confusion Matrix is a tabular visualization of the ground-truth labels versus the model predictions.

This is not exactly a performance metric but rather a tool on which other metrics evaluate the results.

To illustrate the confusion matrix, we need to set some value for the null hypothesis as an assumption.

For example, let us consider the Breast Cancer data and let us assume our Null Hypothesis to be "the individual has no cancer".

Performance metrics - Classification

Each row of the confusion matrix counts the instances in a predicted class and each column represents the instances in an actual class.

		Predicted	
		Has Cancer	Doesn't Have Cancer
Ground Truth	Has Cancer	TP	FN
	Doesn't Have Cancer	FP	TN

Each cell in the confusion matrix represents one of the evaluation factors: True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN).

Performance metrics - Classification

- True Positive (TP) counts how many positive class samples the model predicted correctly.
- True Negative (TN) counts how many negative class samples the model predicted correctly.
- False Positive (FP), or *false alarm*, counts how many positive class samples the model predicted incorrectly. This factor represents Type-I error in statistical nomenclature.
- False Negative (FN), or *miss*, counts how many negative class samples the model predicted incorrectly. This factor represents Type-II error in statistical nomenclature.

Performance metrics - Classification

Precision (also **Positive Predictive Value (PPV)**) is the ratio of true positives over all predicted positives.

$$P = \frac{TP}{TP + FP}$$

$$= \frac{\text{cancer pat. correctly identified}}{\text{cancer pat. correctly identified} + \text{incorrectly labeled non-cancer pat. as cancer}}$$

The precision metric focuses on Type-I errors (FP), occurring when we reject a true null Hypothesis. So, in this case, Type-I error is incorrectly labeling non-cancer patients as cancerous.

A precision score 1 signifies that the model did not miss any true positives, and the predicted positives are all true positives. What it cannot measure is the existence of Type-II error, which is false negatives – cases when a cancer patient is identified as non-cancer.

A low precision score (below 0.5) means the classifier has a high number of false positives which can be an outcome of untuned model hyperparameters or imbalanced class.

Performance metrics - Classification

Recall (also **Sensitivity, True-Positive Rate (TPR)**) is the ratio of true positives over all positive samples.

$$
\begin{aligned}
R &= \frac{TP}{TP + FN} \\
&= \frac{\text{cancer pat. correctly identified}}{\text{cancer pat. correctly identified} + \text{incorrectly labeled cancer pat. as non-cancer}}
\end{aligned}
$$

The recall metric focuses on type-II errors (FN) occurring when we accept a false null hypothesis. So, in this case, type-II error is incorrectly labeling a cancer patient as non-cancer.

Recall equal 1 signifies that the model did not miss any true positives, and all the positives were correctly predicted. It cannot measure is the existence of type-I error, that is, false positives - cases when a non-cancer patient is identified as cancerous.

A low recall score (below 0.5) means the classifier has a high number of false negatives which can be an outcome of untuned model hyperparameters or imbalanced class.

Performance metrics - Classification

Precision and Recall, taken individually, do not provide a complete assessment of classification performance.

The **F1-score** metric uses a combination of precision and recall.

$$F1 = \frac{2\,TP}{2\,TP + FP + FN} = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

It is often used as a measure of overall classification performance.

High F1 score indicates a high precision as well as high recall. It gives good results also on imbalanced classification problems.

However, a low F1 score does not tell much as the problem could be low precision or low recall and it is unclear whether the model suffers from type-I or type-II error.

Performance metrics - Classification

Example.

Consider a set of 15 patients, 7 of which have cancer and the remaining 8 having no cancer.

Suppose we have a binary classifier that identifies 9 cancer patients, of which 6 are correct predictions. Hence 15-9=6 patients are predicted as non-cancer.

In this case, TP=6, FP=9-6=3, FN = 7-6=1, TN = 6-1=5

It follows that

$$P = \frac{6}{6+3} = \frac{2}{3}, \quad S = \frac{6}{6+1} = \frac{6}{7}, \quad F1 = \frac{12}{12+3+1} = \frac{3}{4}$$

Performance metrics - Classification

Example.

Consider the same set of 15 patients, 7 of which have cancer and the remaining 8 having no cancer.

Suppose we have a binary classifier that identifies 1 cancer patient correctly. Hence 15-1=14 patients are predicted as non-cancer.

In this case, TP=1, FP=0, FN = 7-1=6, TN = 8

It follows that

$$P = \frac{1}{1+0} = 1, \quad S = \frac{1}{1+6} = \frac{1}{7}, \quad F1 = \frac{2}{2+6} = \frac{1}{4}$$

In this example, high precision comes at the expense of low sensitivity.

Performance metrics - Classification

Example.

Consider the same set of 15 patients, 7 of which have cancer and the remaining 8 having no cancer.

Suppose we have a binary classifier that predicts all 15 patients as cancer patients. Hence 0 patients are predicted as non-cancer.

In this case, TP=7, FP=8, FN = 0, TN = 0

It follows that

$$P = \frac{7}{7+8} = \frac{7}{15}, \quad S = \frac{7}{7+0} = 1, \quad F1 = \frac{14}{14+8} = \frac{7}{11}$$

In this example, high sensitivity comes at the expense of low precision.

Performance metrics - Classification

The **Area under Receiver Operating Characteristics (AUC-ROC)** curve is the most complete way to assess classification performance.

It makes use of the **true positive rate** (TPR) (or recall or sensitivity) and the **false positive rate** (FPR) (or fallout)

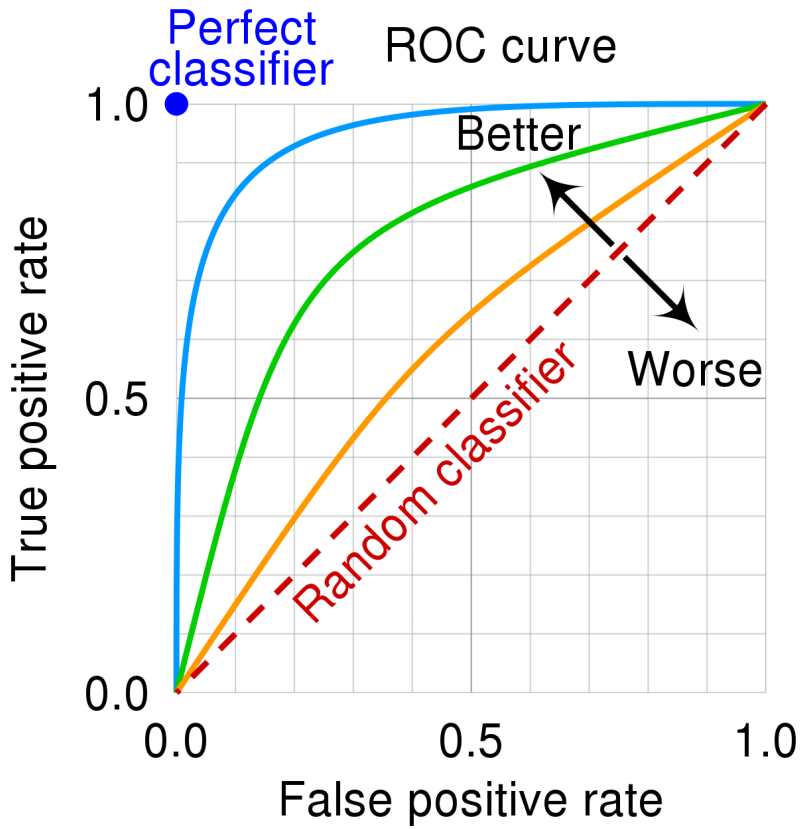
$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN} = 1 - TNR = 1 - \frac{TN}{TN + FP}$$

where TNR is the **true negative rate** or **specificity.**

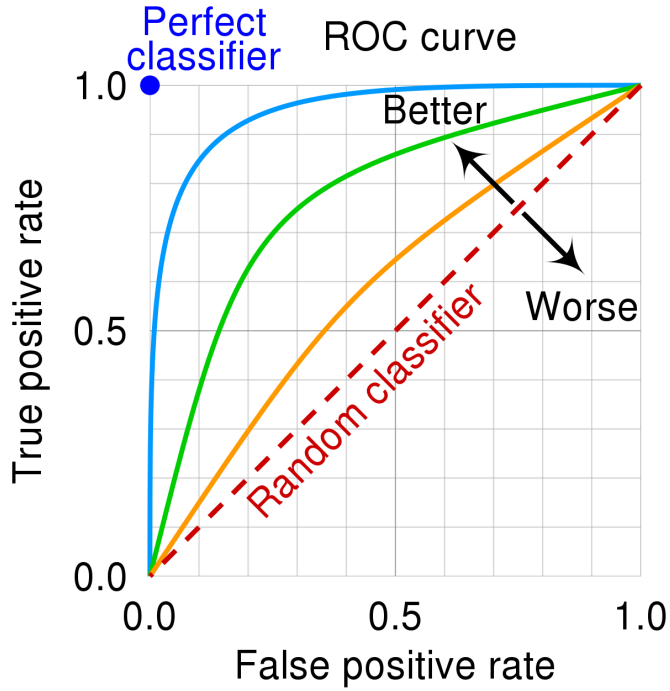
- TPR/recall measures the proportion of positive data points that are correctly considered as positive, with respect to all positive data points. The higher the TPR, the fewer positive data points we miss.
- FPR/fallout measures to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points. The higher the FPR, the more negative data points we misclassify.

Performance metrics - Classification

The ROC curve is obtained by plotting TPR vs FPR computed for many different thresholds values of the classifier.
The area under this curve, called the AUROC curve, quantifies the classification performance (higher area, better classifier).



Performance metrics - Classification



The best possible classifier would yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing 100% sensitivity (no false negatives) and 100% specificity (no false positives). The (0,1) point is also called a perfect classification.

A random guess (no-skill classifier) would give a point along a diagonal line - the so-called line of no-discrimination - from the bottom left to the top right corners, regardless of the positive and negative base rates.

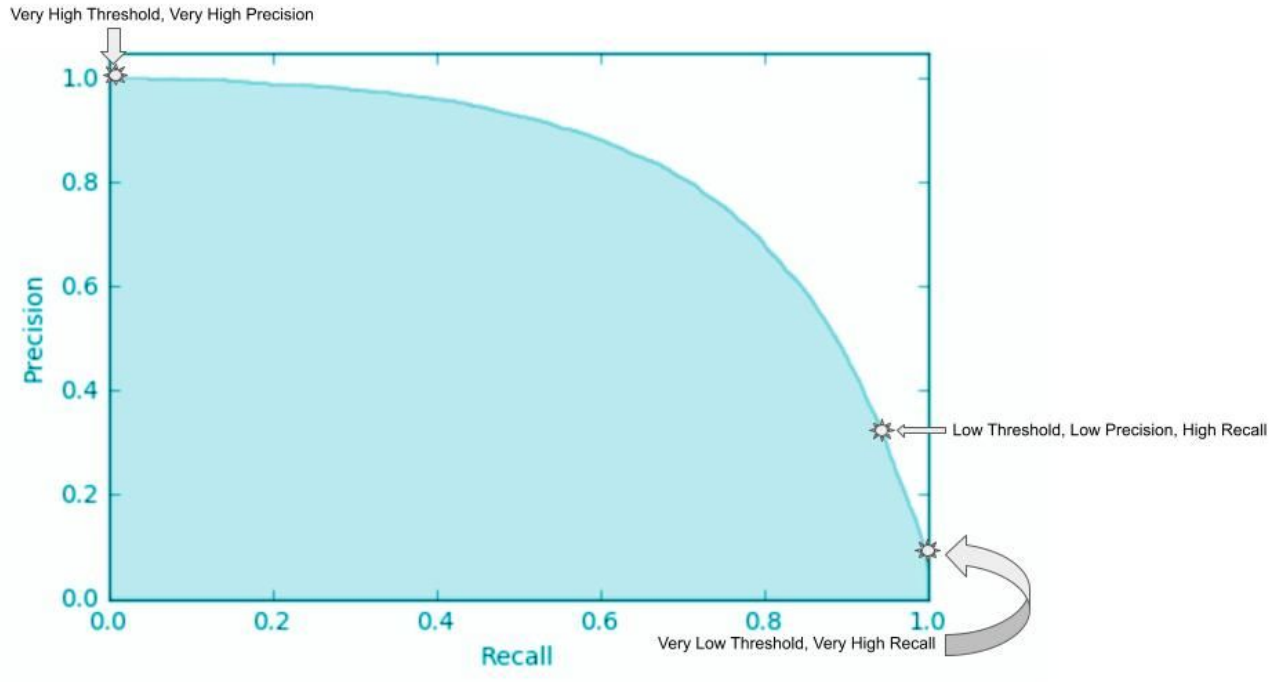
Performance metrics - Classification

When dealing with a highly skewed dataset (highly imbalanced data), the **Precision-Recall (PR) curve** gives a more informative picture of an algorithm's performance with respect to the ROC curve.

It is obtained by plotting P vs R computed for many different thresholds values of the classifier.

As in the AUROC case, a higher area corresponds to a better classifier since a high area under the Precision-Recall curve represents both high recall and high precision.

Performance metrics - Classification



A classifier with perfect skill is depicted as a point at a coordinate of (1,1) and a skillful classifier model is represented by a curve that bows towards a coordinate of (1,1).

A no-skill classifier will be a horizontal line on the plot with a precision that is proportional to the number of positive examples in the dataset. For a balanced dataset this will be 0.5.

Performance metrics - Classification

Main formulas.

total population $= P + N$
accuracy $= \frac{TP+TN}{P+N}$
sensitivity or recall or TPR $= \frac{TP}{P} = \frac{TP}{TP+FN} = 1 - FNR$
specificity or selectivity or TNR $= \frac{TN}{N} = \frac{TN}{TN+FP} = 1 - FPR$
precision or positive predictive value PPV $= \frac{TP}{TP+FP} = 1 - FDR$
negative predictive value NPV $= \frac{TN}{TN+FN} = 1 - FOR$
miss rate or false negative rate FNR $= \frac{FN}{P} = \frac{FN}{FN+TP} = 1 - TPR$
fall-out or false positive rate FPR $= \frac{FP}{N} = \frac{FP}{FP+TN} = 1 - PPV$
false discovery rate FDR $= \frac{FP}{FP+TP} = 1 - PPV$
false omission rate FOR $= \frac{FN}{FN+TN} = 1 - NPV$
F1 score $= \frac{2TP}{2TP+FP+FN}$

Performance metrics - Classification

		Predicted condition			
	Total population $= P + N$	**Predicted Positive (PP)**	**Predicted Negative (PN)**	Informedness, bookmaker informedness (BM) $= TPR + TNR - 1$	Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	**Positive (P)**	**True positive (TP),** hit	**False negative (FN),** type II error, miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate $= \frac{FN}{P} = 1 - TPR$
	Negative (N)	**False positive (FP),** type I error, false alarm, overestimation	**True negative (TN),** correct rejection	False positive rate (FPR), probability of false alarm, fall-out $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
	Prevalence $= \frac{P}{P + N}$	Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$
	Accuracy (ACC) $= \frac{TP + TN}{P + N}$	False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) $= \frac{TN}{PN} = 1 - FOR$	Markedness (MK), deltaP (Δp) $= PPV + NPV - 1$	Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$
	Balanced accuracy (BA) $= \frac{TPR + TNR}{2}$	F_1 score $= \frac{2\,PPV \times TPR}{PPV + TPR} = \frac{2\,TP}{2\,TP + FP + FN}$	Fowlkes–Mallows index (FM) $= \sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index $= \frac{TP}{TP + FN + FP}$

Sources: [1][2][3][4][5][6][7][8][9] view · talk · edit

Performance metrics - Confidence intervals

Much of machine learning involves **estimating** the performance of a predictive model.

For instance, in a binary classification problem, we wan to estimate the Accuracy of a predictor where

$$p := \text{Accuracy} = \frac{\text{correct predictions}}{\text{all predictions}} = \frac{TP + FN}{P + N}$$

Note that Accuracy is a **proportion** describing the ratio of correct (or incorrect) predictions made by the model.

Each prediction is a binary decision that could be correct or incorrect. Technically, this is a **Bernoulli trial** and the proportions in a Bernoulli trial have a specific distribution called a **binomial distribution**.

Performance metrics - Confidence intervals

When we use a test set to compute the Accuracy of a predictor, we are in fact estimating p. In other words, compute an **estimator of the accuracy** that we denote $\hat{p}$.

Question: How close is $\hat{p}$ to p?

Confidence intervals are a way of quantifying the uncertainty of an estimate.
They can be used to add a bounds or likelihood on a population parameter, such as a proportion, estimated from a sample of independent observations from the population - the test set in our case.

Confidence intervals are one of the methods studied in statistical inference.

Performance metrics - Confidence intervals

By the Central Limit Theorem, with large sample sizes n ($n \geq 30$) we can approximate the distribution of $\hat{p}$ with a Gaussian with a standard deviation

$$\sqrt{\frac{p(1-p)}{n}}$$

Hence, the $(1-\alpha)100$ percent confidence interval of p is

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

For the 90% CI: $\alpha = 0.1$, $z_{\alpha/2} = 1.645$
For the 95% CI: $\alpha = 0.05$, $z_{\alpha/2} = 1.960$

Performance metrics - Confidence intervals

Example. Consider a predictive model with a classification accuracy of 83% on a validation dataset with 50 samples, that is $n = 50$. We can calculate the 95% confidence interval of the classification accuracy p as

$$\hat{p} \pm 1.960\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} = 0.83 \pm 1.960\sqrt{\frac{0.83(0.17)}{50}} = 0.83 \pm 0.10$$

This means that the classification accuracy is

$$p = 0.83 \pm 0.10$$

or, alternatively,

$$p \in [0.73, 0.93]$$

with significance level $\alpha = 0.05$.

Performance metrics - Confidence intervals

Sometimes we do not know the distribution for a chosen performance measure.

I could also happen that the predicted variable is not normally distributed, and even when it is, the variance of the normal distribution might not be equal at all levels of the predictor variable.

In this case, we can use a **nonparametric method** for calculating confidence intervals, such as the **bootstrap confidence interval.**

The bootstrap is a simulated Monte Carlo method where samples are drawn from a fixed finite dataset with replacement and a parameter is estimated on each sample. This procedure leads to a robust estimate of the true population parameter via sampling.

Press link.