Deep learning

3.5. Gradient descent

François Fleuret
https://fleuret.org/dlc/

We saw that training consists of finding the model parameters minimizing an empirical risk or loss, for instance the mean-squared error (MSE)

$$\mathscr{L}(w, b) = \frac{1}{N} \sum_n (f(x_n; w, b) - y_n)^2 .$$

Other losses are more fitting for classification, certain regression problems, or density estimation. We will come back to this.

So far we minimized the loss either with an analytic solution for the MSE, or with *ad hoc* recipes for the empirical error rate (k-NN and perceptron).

Notes

We have seen in lecture 3.4. "Multi-Layer Perceptrons" that we can stack multi-dimensional perceptrons to a multi-layer perceptron, which has very good approximation properties.
But a core question remains: how do we train such a model? In each of the modules, there are weights and biases, and we need a way of finding the adequate values for those parameters.

There is generally no *ad hoc* method. The logistic regression for instance

$$P_w(Y = 1 \mid X = x) = \sigma(w \cdot x + b), \text{ with } \sigma(x) = \frac{1}{1 + e^{-x}}$$

leads to the loss

$$\mathscr{L}(w, b) = -\sum_n \log \sigma(y_n(w \cdot x_n + b))$$

which cannot be minimized analytically.

The general minimization method used in such a case is the **gradient descent**.

Given a functional

$$f : \mathbb{R}^D \to \mathbb{R}$$
$$x \mapsto f(x_1, \ldots, x_D),$$

its gradient is the mapping

$$\nabla f : \mathbb{R}^D \to \mathbb{R}^D$$
$$x \mapsto \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_D}(x) \right).$$

Notes

The gradient is defined for a function which goes to a 1D space, and associates to the input the vector composed of the partial derivatives. Its components quantify how much each input influences locally the value of f.

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of $\mathscr{L}$ around w_0

$$\tilde{\mathscr{L}}_{w_0}(w) = \mathscr{L}(w_0) + \nabla\mathscr{L}(w_0)^\top(w - w_0) + \frac{1}{2\eta}\|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on $\mathscr{L}$.

We have

$$\nabla\tilde{\mathscr{L}}_{w_0}(w) = \nabla\mathscr{L}(w_0) + \frac{1}{\eta}(w - w_0),$$

which leads to

$$\operatorname*{argmin}_w \tilde{\mathscr{L}}_{w_0}(w) = w_0 - \eta\nabla\mathscr{L}(w_0).$$

Notes

Here, $\mathscr{L}$ goes from the parameter space $\mathbb{R}^D$ to the loss space. So in the case of the MLP, D is the total number of parameters and $\mathscr{L}$ would be a quantity such as the MSE.
The minimum of $\tilde{\mathscr{L}}$ is achieved when "moving" from w_0 in a direction opposite to the gradient. The quadratic term makes it explicit how far the linear approximation of the loss can be trusted. Formulating the gradient descent with this linear approximation opens the way to more sophisticated extensions, such as natural gradient descent, which replaces the quadratic term by a problem specific one, like when the natural metric is not Euclidean (e.g. space of distributions).

The resulting iterative rule, which goes to the minimum of the approximation at the current location, takes the form:

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t),$$

which corresponds intuitively to "following the steepest descent".

This [most of the time] eventually ends up in a **local** minimum, and the choices of w_0 and η are important.

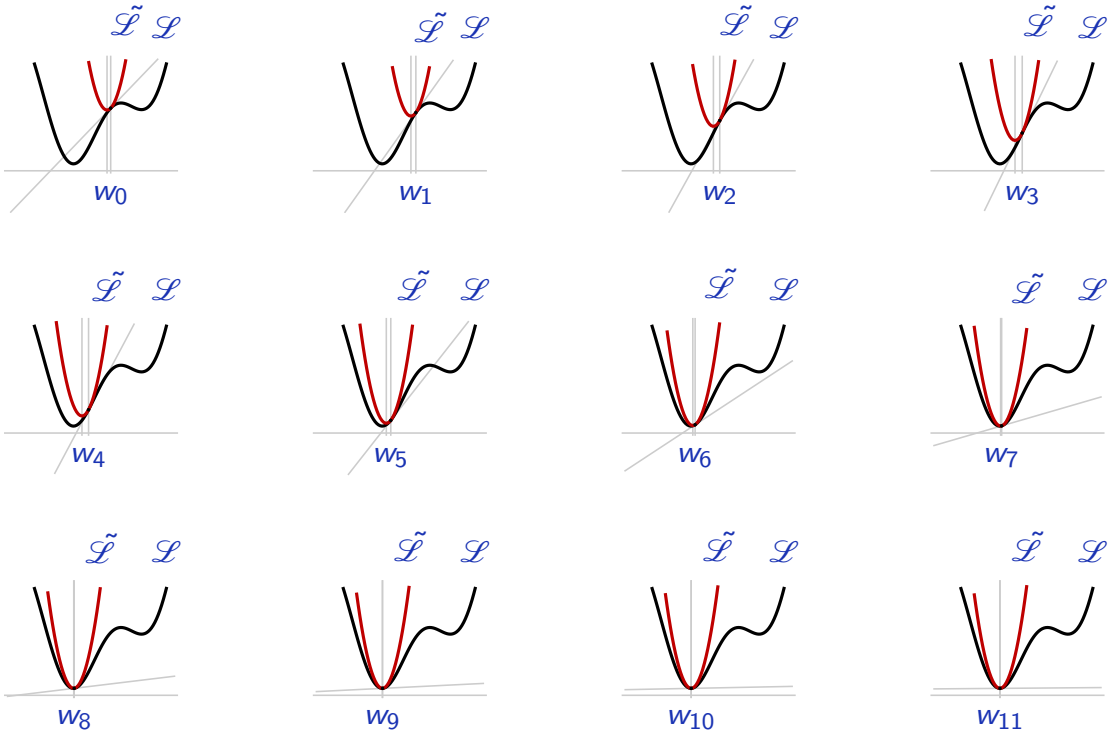
Notes

Here w_t is the value of the parameters of the model at iteration t of the training, and the new value of the parameter is w_{t+1}, obtained by subtracting $\eta \nabla \mathscr{L}(w_t)$ from w_t, moving in the "the steepest descent".
In practice, the gradient descent algorithm consists of:

- Initializing the parameters, often randomly: this is an important element in deep learning, which influences the training a lot, both regarding the convergence and the final performance.
- at every step, the gradient of the loss w.r.t. the current parameter is computed, and the parameters are "moved a little bit" in the opposite direction of the gradient, to make the loss go down.

The choice of w_0 (where we start from) and η (the **step size**, how much we move at each step, which can depend on t) is very important.

$\eta = 0.125$



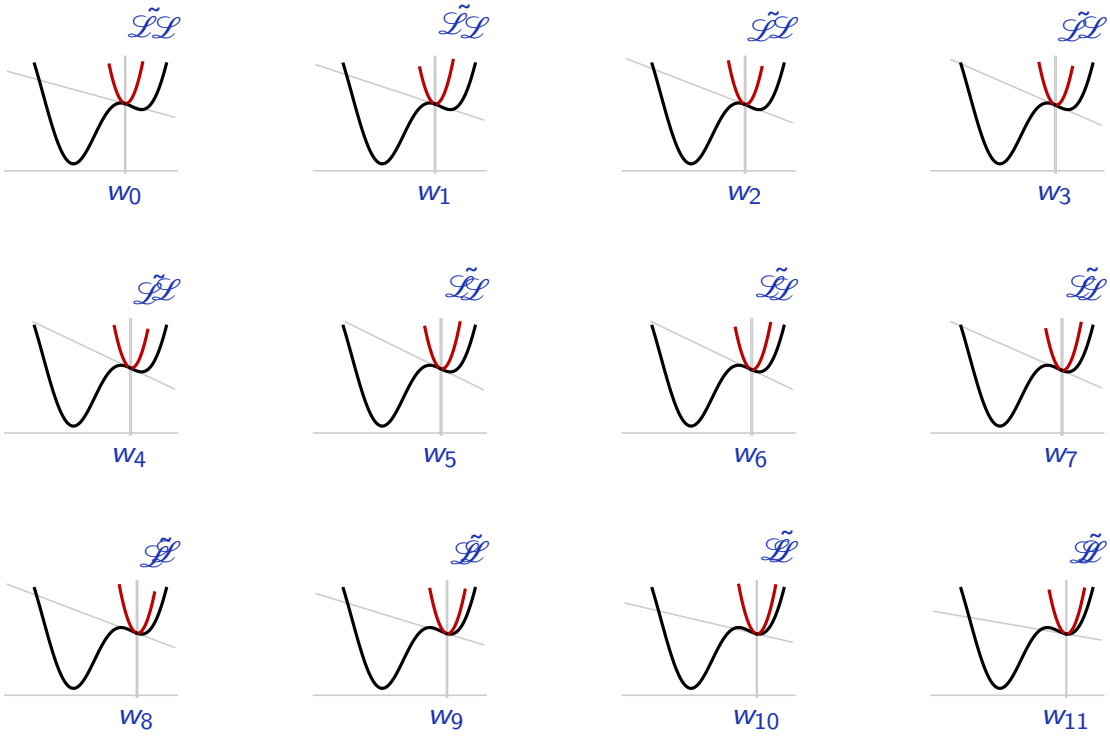
Notes

We illustrate the gradient descent algorithm with a parameter space of dimension 1:

- the black curve represents the loss $\mathscr{L}$, and the goal is to find the w which minimizes it;
- w_0 is the initial value of the parameter;
- the red curve is the quadratic approximation $\tilde{\mathscr{L}}$ of the loss $\mathscr{L}$;
- at each iteration, the new value of the parameter is the value that minimize $\tilde{\mathscr{L}}$, i.e. the vertex of the parabola.

On this example, w_0 and η lead to properly reaching the minimum of $\mathscr{L}$.

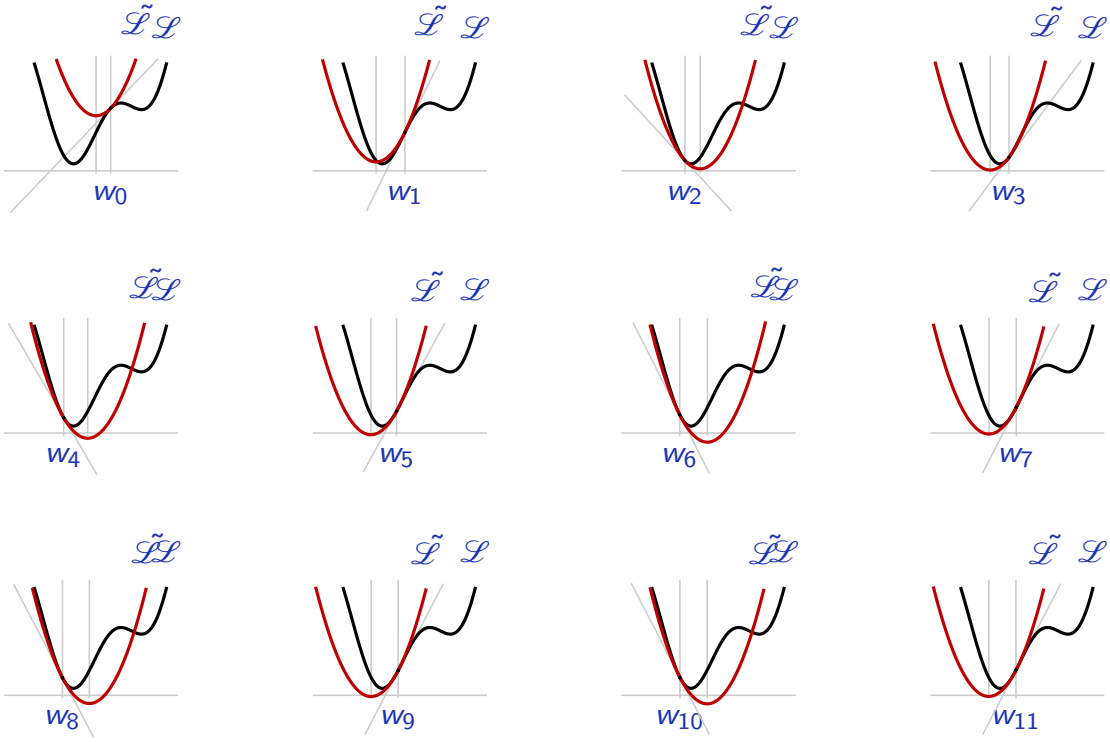
$\eta = 0.125$



Notes

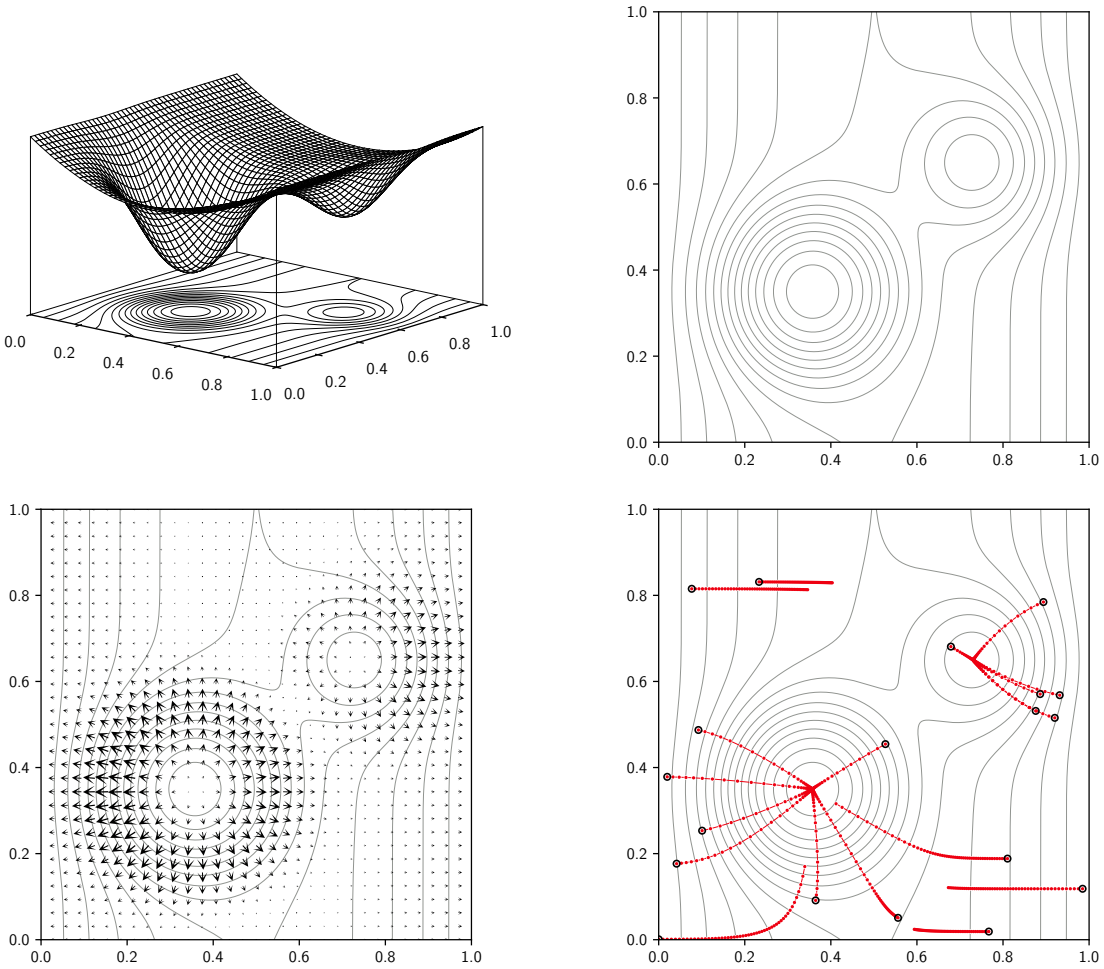
Here, we illustrate a first weakness of gradient descent.
With the same learning rate η bu by changing the starting point w_0 of the procedure, the algorithm now ends up in a local minimum.

$\eta = 0.5$



Notes

When η, is too large the algorithm ends up oscillating around a minimum.
The quadratic term should be such that it "fits" in the narrow valley where we want to converge, which translates in having an η small enough not to bounce around.



Notes

We can illustrate the gradient descent with a more complex example. The loss is now a function from $\mathbb{R}^2$ to $\mathbb{R}$. It has two local minima.
The top left picture is a 3D visualization of the loss $\mathscr{L}$. The x and y axes in the plan are the parameter space: $w \in \mathbb{R}^2$.
The top right picture is a top view of the loss with level lines.
The bottom left picture is a vector map of the gradient of the loss: at every point in the space, the gradient is the partial derivative of $\mathscr{L}$ with respect of x, then y, and is represented with a vector:

$$\left[\frac{\partial \mathscr{L}}{\partial x}, \frac{\partial \mathscr{L}}{\partial y}\right]$$

The length of each arrow is proportional to the norm of the gradient:

- in flat areas of the surface, gradients are small, and depicted arrows are short;
- in steep areas of the surface (sides of the paraboloids), the gradients are larger, and the arrows longer.

The bottom right image shows for different starting points the fifty successive values taken by the gradient descent algorithm:

- the starting points are depicted with a black circle,
- each red circle is the value taken by a w_t.

This example shows several weaknesses:

- when the gradient is small (top left, and bottom right), the algorithm moves slowly,
- in the top right part, the algorithm converges to the local minimum.

We saw that the minimum of the logistic regression loss

$$\mathscr{L}(w, b) = -\sum_n \log \sigma(y_n(w \cdot x_n + b))$$

does not have an analytic form.

We can derive

$$\frac{\partial \mathscr{L}}{\partial b} = -\sum_n \underbrace{y_n \, \sigma(-y_n(w \cdot x_n + b))}_{u_n},$$

$$\forall d, \ \frac{\partial \mathscr{L}}{\partial w_d} = -\sum_n \underbrace{x_{n,d} \, y_n \, \sigma(-y_n(w \cdot x_n + b))}_{v_{n,d}},$$

which can be implemented as

```
def gradient(x, y, w, b):
    u = y * ( - y * (x @ w + b)).sigmoid()
    v = x * u.view(-1, 1) # Broadcasting
    return - v.sum(0), - u.sum(0)
```

and the gradient descent as

```
w, b = torch.randn(x.size(1)), 0
eta = 1e-1

for k in range(nb_iterations):
    print(k, loss(x, y, w, b))
    dw, db = gradient(x, y, w, b)
    w -= eta * dw
    b -= eta * db
```

Notes

To compute the partial derivative of the loss w.r.t. the parameters, we have the following result:

$$\begin{aligned}(\log \sigma)'(x) &= \left(\log\left(\frac{1}{1+e^{-x}}\right)\right)' \\ &= -\frac{-e^{-x}}{1+e^{-x}} \\ &= \frac{1}{e^x+1} = \sigma(-x)\end{aligned}$$

With $\ell_n = \log \sigma(y_n(w \cdot x_n + b))$,

$$\frac{\partial \ell_n}{\partial b} = y_n \, \sigma(-y_n(w \cdot x_n + b))$$

and $\forall d$

$$\frac{\partial \ell_n}{\partial w_d} = x_{n,d} \, y_n \, \sigma(-y_n(w \cdot x_n + b)).$$

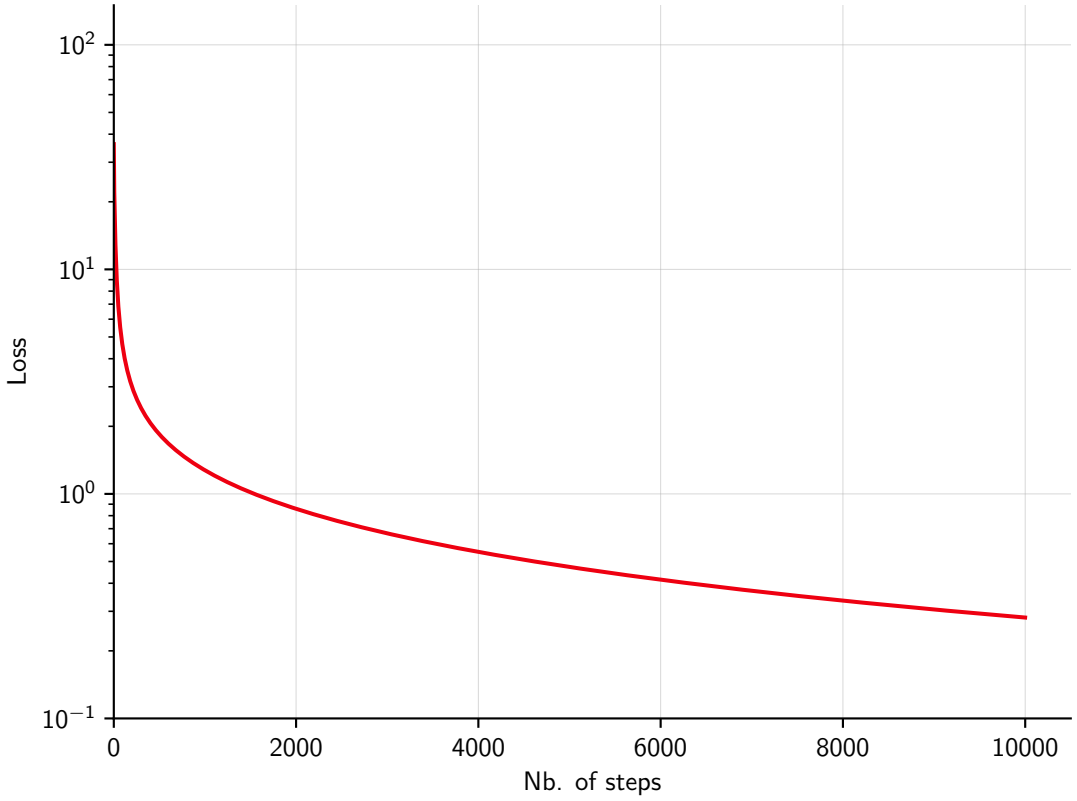
And finally, by using the linearity of the derivation operator,

$$\begin{aligned}\frac{\partial \mathscr{L}}{\partial b} &= -\sum_n \frac{\partial \ell_n}{\partial b} \\ &= -\sum_n y_n \, \sigma(-y_n(w \cdot x_n + b))\end{aligned}$$

and

$$\begin{aligned}\frac{\partial \mathscr{L}}{\partial w_d} &= -\sum_n \frac{\partial \ell_n}{\partial w_d} \\ &= -\sum_n x_{n,d} \, y_n \, \sigma(-y_n(w \cdot x_n + b))\end{aligned}$$

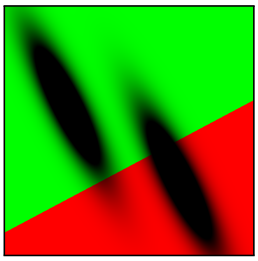
In the implementation, the weight and bias are initialized respectively with values drawn from a normal distribution, and with 0.
`v` is computed by reusing the computation of `u`: `v = x * u.view(-1, 1)`. `u` is re-shaped as a column vector which is then multiplied with each column of the sample tensor. `v.sum(0)` sums all the rows into one row and yields a tensor of size `(D,)`.



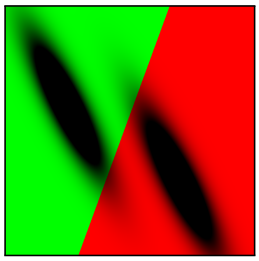
Notes

The plot shows the loss we aim at minimizing as a function of the number of iterations. The learning rate η was 0.1.

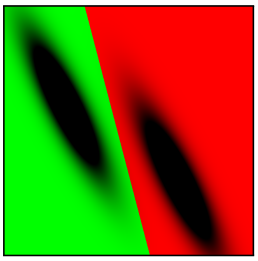
With 100 training points and $\eta = 10^{-1}$.



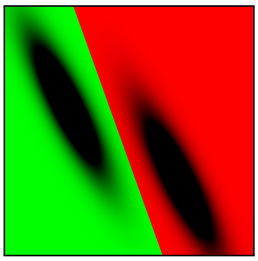
$n = 0$



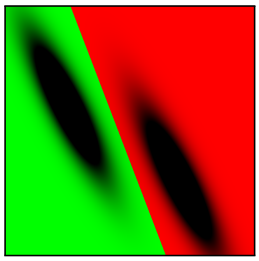
$n = 10$



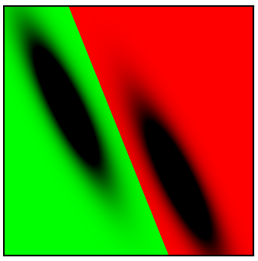
$n = 10^2$



$n = 10^3$



$n = 10^4$



LDA

Notes

The plot shows the decision boundary trained at different stages of the learning process:

- The two populations are depicted in black in each vignette and correspond to normal distributions with different means and the same covariance matrix. We ran the optimization with 100 points drawn from each class (not depicted).
- The red (resp. green) area is the set of points which are classified as being from class 0 (resp. 1) by the predictor.

At start, for $n = 0$, the weight vector is generated randomly and therefore performs poorly: the decision boundary does not separate the two populations.
As the training goes on, we see that the separation starts to better discriminate the two populations, which goes along the decrease of the loss.
As a comparison, the bottom-right image shows the optimal result obtained with LDA (see lecture 3.2. "Probabilistic view of a linear classifier").