131890-S Optimization methods **Winter 2023/2024**

Lecture 4: Local methods

Daniel Kaszyński *17 November 2023 r.*

4.1 Directional derivative and partial derivative

The directional derivative of a function measures how the function changes at a specific point in the direction of a given vector. In other words, it quantifies the rate of change of a function along a particular direction. It measures impact of the shift by vector h of function f.

Definition 1: Derivative of a 1D function

By a derivative of a 1D function $f : \mathbb{R} \to \mathbb{R}$ we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{4.1}$$

When dealing with a one-dimensional function, we implicitly assume that the vector h is simply equal to 1 (look Definition 1). This means that the shift in x domain is multiplied by 1 when moving in this space. In a more general case, especially when operating on multidimensional functions, we can move along different h vectors. They should be therefore included in the formula.

Definition 2: Directional derivative of a function

By a directional derivative of a multidimensional function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x+h \in \mathbb{D}$ at point x along vector h we call a function:

$$\frac{df}{dh}(x,h) = \lim_{\delta \to 0} \frac{f(x+\delta h) - f(x)}{\delta} \tag{4.2}$$

Definition 3: Directional derivative of a function (using gradient)

By a directional derivative of a multidimensional function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x+h \in \mathbb{D}$ at point x along vector h we call a function:

$$\frac{df}{dh}(x,h) = \nabla_f(x)h = |\nabla_f(x)||h| \cos(\alpha) \tag{4.3}$$

where $\nabla_f(x)$ is a gradient of a function f and α is an angle between vectors $\nabla_f(x)$ and h.

Partial derivative is a special case of directional derivative. A partial derivative describes the rate at which a multivariable function changes with respect to one of its variables while keeping the other variables constant. It is essentially the derivative of a function with respect to one of its independent variables, treating the other variables as constants. Partial derivative is also a sensitivity of a function.

Definition 4: Partial derivative

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Partial derivative of f at the point x with respect to variable x_i, $i = 1, 2, \ldots n$ we call the function:

$$\frac{\partial f}{\partial x_i}(x) = \frac{df}{de_i}(x)$$

where e_i is the i-th versor of space $\mathbb{R}^n$. Partial derivative of f with respect to x_i is then a directional derivative of f in direction of i-th versor, meaning that $h = e_i$.

4.2 Kernel

Kernel (also known as the null space or nullspace) represents the set of solutions or vectors that "vanish" or map to the zero vector under the given linear transformation or matrix operation. The concept of the kernel is fundamental in linear algebra and is used in various applications, including solving systems of linear equations and understanding properties of linear transformations.

Consider a linear map represented as a m × n matrix A with coefficients in a field K, that is operating on column vectors x with n components over K. The kernel of this linear map is the set of solutions to the equation $Ax = 0$, where 0 is understood as the zero vector.

$$\mathrm{N}(A) = \mathrm{Null}(A) = \ker(A) = \{\mathbf{x} \in K^n \mid A\mathbf{x} = \mathbf{0}\}. \tag{4.4}$$

Kernel of n-dimentional space is an identity matrix of size n. In simpler words, it is the $n \times n$ square matrix with ones on the main diagonal (from top left to bottom right) and zeros elsewhere.

$$\ker(A_{n\times n}) = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}$$

Each row of this matrix (also called versor of space) is named e_i, where $i = 1, 2, \ldots n$. They are useful while calculating partial derivative of a function (look Definition 4).

4.3 Optimality while working with multidimensional space

When working with multidimensional functions, we are often tasked with finding the extremum of those functions. If we are using local methods to do that, we should find direction or a vector that will point us towards said extremum. Local methods, as the name suggests, are considering only nearest neighbourhood of a point in space. Having that knowledge, the simplest way to find extremum is to follow direction of a quickest decrease (or increase) of a function. Vector that points in that direction is an optimal vector that we are searching for.

Optimality requires from us stating two conditions beforehand:

- Naming the scoring function

- Choosing if we want to minimize or maximize said function.

Scoring function in most cases is just a provided function f. We can also notice, that minimizing the function and maximizing the function are both very similar tasks. In fact, we can substitue maximizing the function with minimizing the inverse of this function.

$$\operatorname{argmax} f = \operatorname{argmin}(-f) \tag{4.5}$$

Example 1. Let $f(x, y) = x^2 + 2y^2$ and $x_0 = (2, 3)$. Plot for such a function would look as follows:

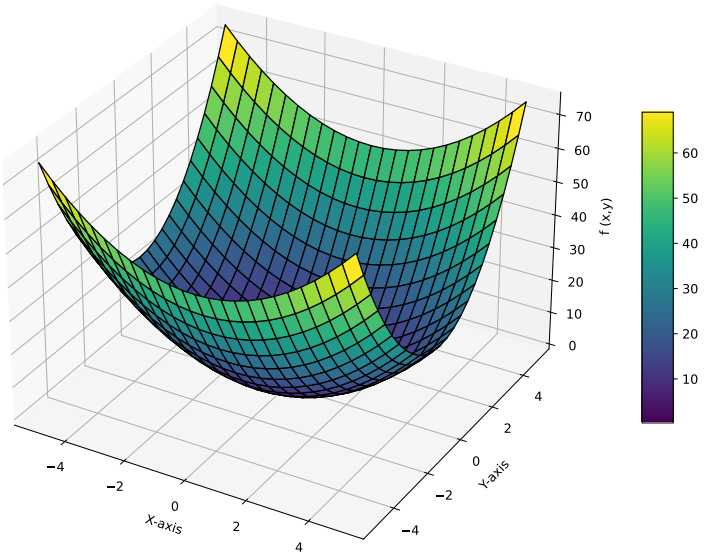

Figure 4.1: Function $f(x) = x^2 + 2y^2$

Function f has an extremum in point (0, 0). If we were to find that extremum from point x_0 we would need to find an optimal vector that represents quickest decrease in function f. This vector is represented by the antigradient of this function ($-\nabla_f(x)$).

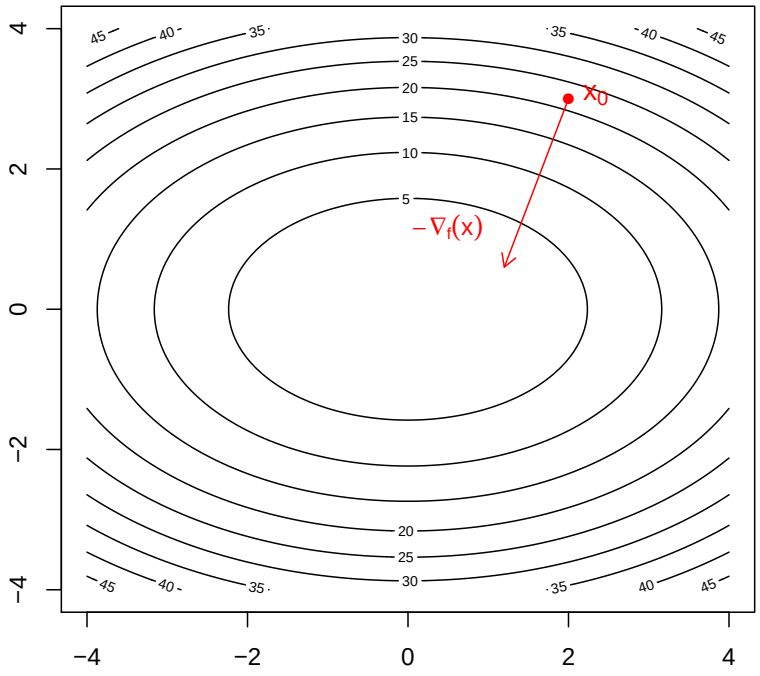

Figure 4.2: Vector starting at point x_0 that represents quickest decrease in function f

The directional derivative can help with finding the optimal vector. If we want to minimize a function (like in Example 1), lets consider mathematical formula for the directional derivative:

$$\underset{h}{\operatorname{argmin}} \frac{df}{dh}(x,h) = \underset{h}{\operatorname{argmin}} |\nabla_f(x)||h| \cos(\alpha) \tag{4.6}$$

Assuming that the length of the gradient $\nabla_f(x)$ is constant and the length of the vector h is constant, the main parameter that we can use is the angle between these vectors ($\cos(\alpha)$). The gradient shows us the direction of the fastest increase in the function value. Naturally, if we want to minimize the function, we should choose the opposite direction - antigradient. This is also confirmed by the formula above as $\cos(180^o)$ is equal to -1 while vector lengths are always positive values.

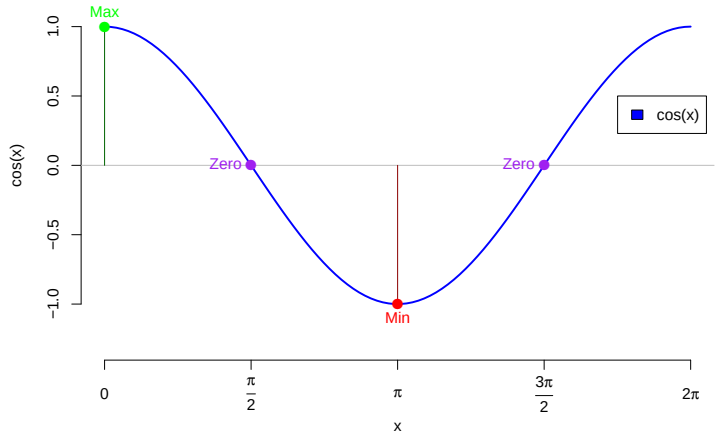

Figure 4.3: Cosinus function

4.4 Gradient descent

Gradient descent is an optimization algorithm commonly used to minimize nonlinear analytical functions. Those functions are often the cost or loss function in machine learning and other optimization problems. The goal of gradient descent is to iteratively move towards the minimum of a function by adjusting its parameters.

Definition 5: Gradient descent

Given that gradient descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - \alpha \nabla_f(x_k) \tag{4.7}$$

where $k \in \mathbb{N}$ is an iteration number, α is predefined learning rate and $\nabla_f(x_k)$ is the gradient of a function f in point x_k.

The general idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the each consecutive point, because this is the direction of steepest descent. On the other hand, stepping in the direction of the gradient will lead to a local maximum of that function.

Example 2. Knowing the formula for each step of gradient descent of function f, let $x_1 = x_0 - \alpha \nabla_f(x_0)$, where x_0 is a starting point, x_1 is the next point calculated in direction of biggest function decrease and parameter α is learning rate set to 0.1.

Following the same process, we can calculate the points even further down the line in the iterative manner:

- $x_2 = x_1 - \alpha \nabla_f(x_1)$
- $x_3 = x_2 - \alpha \nabla_f(x_2)$

- $x_4 = x_3 - \alpha \nabla_f(x_3)$ (...)

Let $f(x) = x^2$ and $x_0 = 1$. Values for the next points x_0, x_1, x_2... in gradient descent iterations are equal to:

- $x_1 = x_0 - \alpha \nabla_f(x_0) = 1 - 0.1 \cdot 2 = 0.8$
- $x_2 = x_1 - \alpha \nabla_f(x_1) = 0.8 - 0.1 \cdot 1.6 = 0.64$
- $x_3 = x_2 - \alpha \nabla_f(x_2) = 0.64 - 0.1 \cdot 1.28 = 0.512$
- $x_4 = x_3 - \alpha \nabla_f(x_3) = 0.512 - 0.1 \cdot 1.024 = 0.4096$ (...)

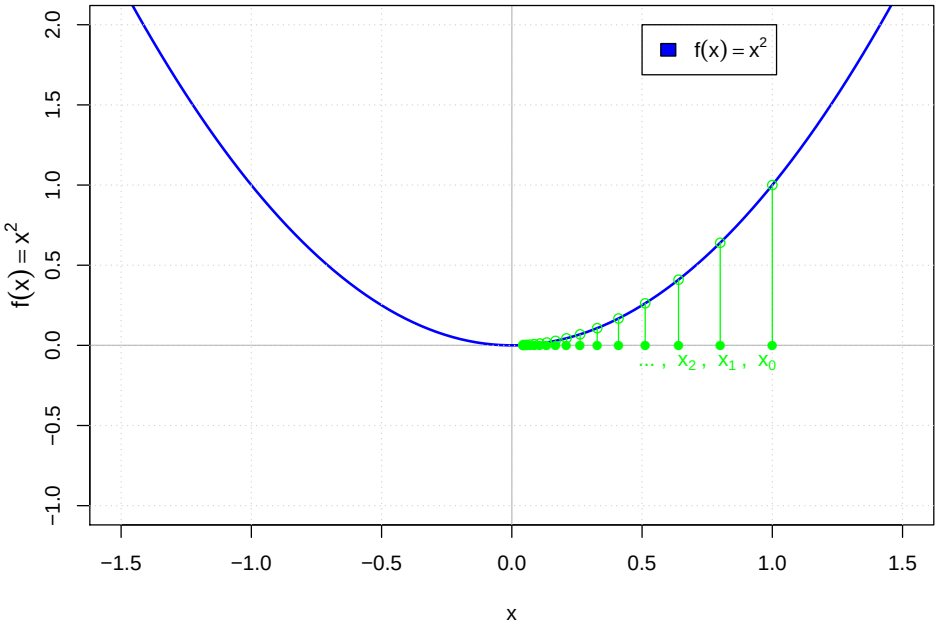

Figure 4.4: Converging of consecutive values of points x_0, x_1, x_2... to extremum of function $f(x) = x^2$ upon introducing learning parameter $\alpha = 0.1$

Using programming language **R** we can write a simple gradient descent implementation. First of all, we need a function to calculate the gradient. We could use $grad(f, x)$ from *numDeriv* library or we could implement it from scratch like so:

```
if(!require(docstring)) library(docstring) # A library for code documentation!

num_grad <- function(f, x, h = 10^-6){
  #' Central Numerical Gradient
  #'
  #' @description Function responsible for determining the numerical gradient
  #' by calculating the vector of central partial derivatives.
  #'
  #' @param f function. The function which gradient we determine.
  #' @param x numerical vector. The point at which the numerical gradient
```

```
  #' is determined.
  #' @param h scalar. Finite difference value.
  #'
  #' @usage num_grad(f, x)
  #'
  #' @return Vector of partial derivatives of function f.

  n <- length(x)
  g <- rep(NA, n) # memory preallocation for storing gradient
  e <- diag(n)

  # calculation of partial derivatives
  for(i in 1 : n) g[i] = (f(x+h*e[i,])-f(x-h*e[i,]))/(2*h)
  return(g)
}
```

Listing 1: Numerical gradient implementation

The above function has comments that allow us to view the function documentation. It will contain a short description, input parameters and return values. We can access the documentation using the following command:

```
# Provides access to documentation
?num_grad
```

Listing 2: Function documentation access

After running this command, the following documentation will be displayed:

Central Numerical Gradient

Description

Function resposible for determining the numerical gradient by calculating the vector of central partial derivatives.

Usage

```
num_grad(f, x)
```

Arguments

f	function. The function which gradient we determine
x	numerical vector. The point at which the numerical gradient is determined
h	scalar. Finite difference value

Value

Vector of partial derivatives of function f.

Figure 4.5: Numerical gradient function docstring documentation

Now we can test how this numerical gradient function works. To accomplish this, first we need to prepare input parameters. Afterwards, we can invoke said function, check out its results and optionally benchmark it.

```r
# Input parameters
my_fun <- function(x) 2*x[1]^2 + x[2]^2
c(3,4) -> x0  # Caution! Arrows not only to the left!

# Results
my_fun(x) # 2*3^2+4^2 = 17 # Try also: sin(x^2);
x0 <- c(-3, -2)
my_fun(x0) # 2*(-3)^2+(-2)^2 = 22
num_grad(my_fun, x0, 10^-6) # returns vector [-12, -4]

# Benchmark 1
if(!require(numDeriv)) library(numDeriv)   # Library for numerical calculations
grad(my_fun, x0)    # gradient
hessian(my_fun, x0) # hessian

# Benchmark 2
if(!require(Deriv)) install.packages(Deriv); # Library for symbolic calculations
my_fun <- function(x, y) 2*x^2 + y^2
df <- Deriv(my_fun)
cat('f =  ',  deparse(my_fun)[2], '\n')
cat('df = ', deparse(df)[2])
```

Listing 3: Numerical gradient tests

Having a working implementation of the gradient function at hand, we can move on to creating a function implementing the gradient descent algorithm:

```r
gradient_descent <- function(f, x, a = 0.1, K = 100){
  #' Gradient Descent
  #'
  #' @description Function responsible for calculating gradient descent
  #' of function f over K steps.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x numerical vector. The starting point for the algorithm.
  #' @param a scalar. Optional parameter that determines learning rate (defaults to 0.1).
  #' @param K scalar. Optional parameter that determines maximum iteration limit (defaults
    to 100).
  #'
  #' @usage gradient_descent(f, x, a, K)
  #'
  #' @returns
  #' List of various outputs containing the following:
  #' * x_opt: found solution,
  #' * f_opt: value of target function in the found solution,
  #' * x_hist: history of explored solutions,
  #' * f_hist: history of target function values,
  #' * t_eval: time elapsed during algorithm calculations.

  start_time <- Sys.time()
  results <- list(x_opt = x,
                  f_opt = f(x),
                  x_hist = matrix(NA, nrow = K, ncol = length(x)),
                  f_hist = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)

  for(k in 2:  K){
    # description of the transition from point x_k to x_k+1
    x_new <- x - a * grad(f, x)

    # checking whether the new solution
```

```
    # is the best so far
    if(f(x_new) < results$f_opt){
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }

    results$x_hist[k,] <- x_new
    results$f_hist[k] <- f(x_new)

    x <- x_new
  }
  # time difference between the end and start of the algorithm
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 4: Gradient descent implementation

Example 3. Let $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

Function f at point x_{start} has value equal to $17\frac{1}{8}$. After $K = 100$ iterations of our implementation of gradient descent algorithm, this value is worked down to approximately 0.00711 which is closer to global minimum of the function f - zero. Values at each step of this algorithm can be seen on the plot below:

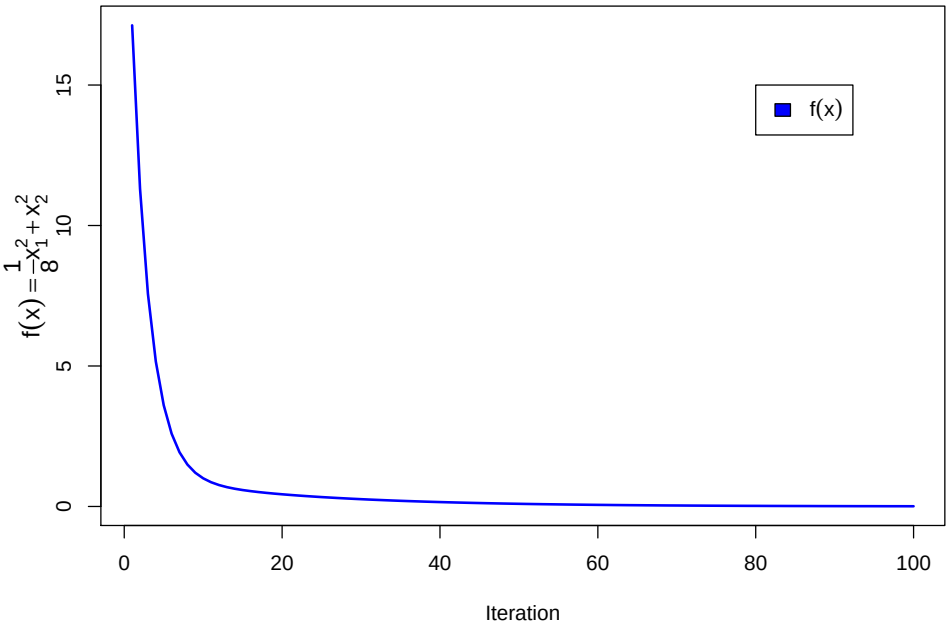

Figure 4.6: Values of function f gradually calculated over 100 iterations using gradient descent

We can plot not only values calculated by this algorithm, but also positions obtained in subsequent iterations in 2D space. The path covered by the gradient descent algorithm can be seen in the plot

below:

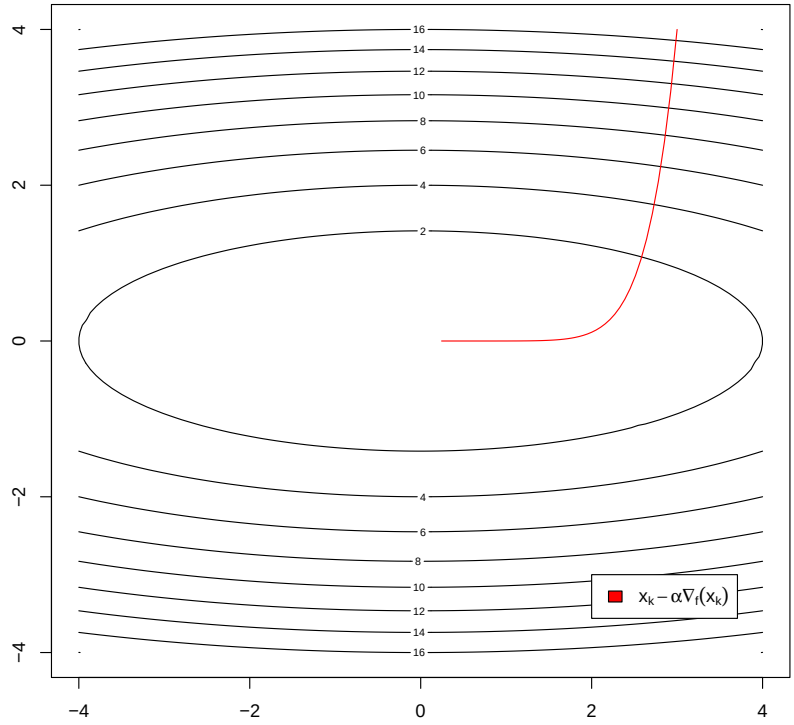

Figure 4.7: Path covered by the gradient descent algorithm on function f over 100 iterations in 2D space

4.5 Learning rate

The learning rate is a hyperparameter that plays a crucial role in controlling the step size at each iteration of the gradient descent algorithm. In the context of machine learning, learning rate is often represented by symbol α. The learning rate determines how much we should adjust the parameters with respect to the gradient of the cost function.

Why and how is it used? Lets consider following example:

Example 4. Knowing that the gradient indicates the greatest increase in function f, let $x_1 = x_0 - \nabla_f(x_0)$, where x_0 is a starting point and x_1 is the next point calculated in direction of biggest function decrease. For now assume that learning rate α is equal to 1.

Following the same process, we can calculate the points even further down the line in the iterative manner:

- $x_2 = x_1 - \nabla_f(x_1)$
- $x_3 = x_2 - \nabla_f(x_2)$

- $x_4 = x_3 - \nabla_f(x_3)$ (...)

Let $f(x) = x^2$ and $x_0 = 1$. Knowing that the gradient of a 1D function is a simple derivative of said function, we can calculate points $x_0, x_1, x_2...$ for $f(x)$:

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 2 = -1$
- $x_2 = x_1 - \nabla_f(x_1) = -1 - (-2) = 1$
- $x_3 = x_2 - \nabla_f(x_2) = 1 - 2 = -1$
- $x_4 = x_3 - \nabla_f(x_3) = -1 - (-2) = 1$ (...)

As we can see, a kind of deadlock has been achieved following such steps. Consecutive values just oscillate around the extremum, never converging to it.

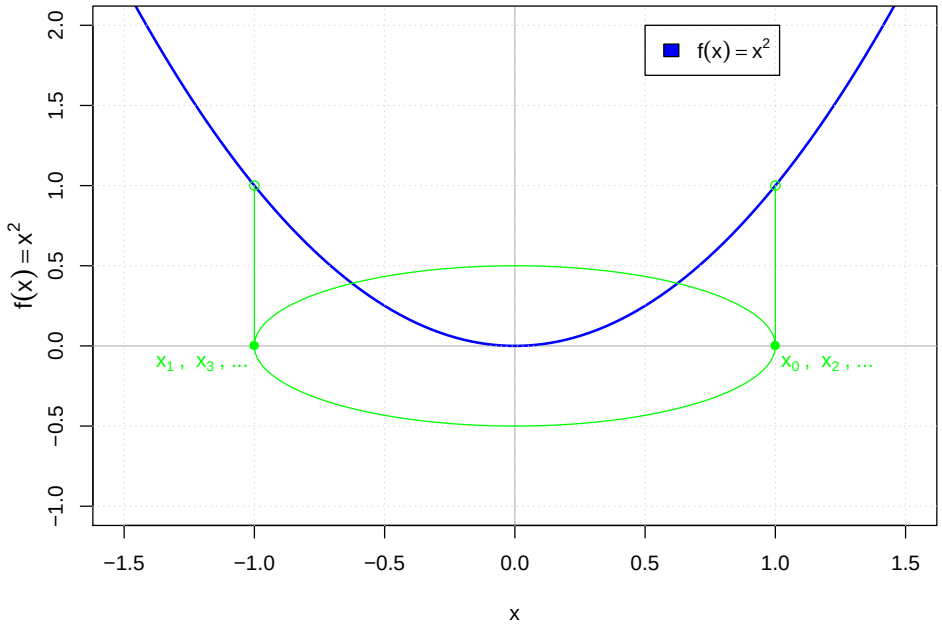

Figure 4.8: Simple deadlock created due to using whole gradient value while calculating points $x_0, x_1, x_2...$

The situation only gets worse if we assume that the function $f(x) = x^4$. When calculating points $x_0, x_1, x_2...$ we get increasingly larger values:

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 4 = -3$
- $x_2 = x_1 - \nabla_f(x_1) = -3 - (-108) = 105$
- $x_3 = x_2 - \nabla_f(x_2) = 105 - 4630500 = -4630395$
- $x_4 = x_3 - \nabla_f(x_3) \approx -4630395 - (-3.9711301e + 20) \approx -3.9711301e + 20$ (...)

One of the possible solutions for problem illustrated in Example 4 is to introduce some kind of learning parameter, learning space parameter or step size parameter (however we want to name it). This parameter would be responsible for controlling the size of the steps we are taking in each iteration. In other words, it would reduce the impact that gradient would have during each calculation. The mentioned parameter is exactly what learning rate is suppose to be. It is an integral part of the gradient descent algorithm and has big impact on how this algorithm performs.

The learning rate is a critical hyperparameter that needs to be carefully chosen. If the learning rate is too small, the algorithm may converge very slowly, requiring a large number of iterations to reach the minimum. On the other hand, if the learning rate is too large, the algorithm may overshoot the minimum and fail to converge or oscillate around the minimum.

Example 5. Let $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

The path covered by the gradient descent algorithm may vary depending on the chosen learnig rate. Let $\alpha_1 = 0.1, \alpha_2 = 0.4$ and $\alpha_3 = 0.8$. How gradient descent converges using this parameters can be seen on the plot below:

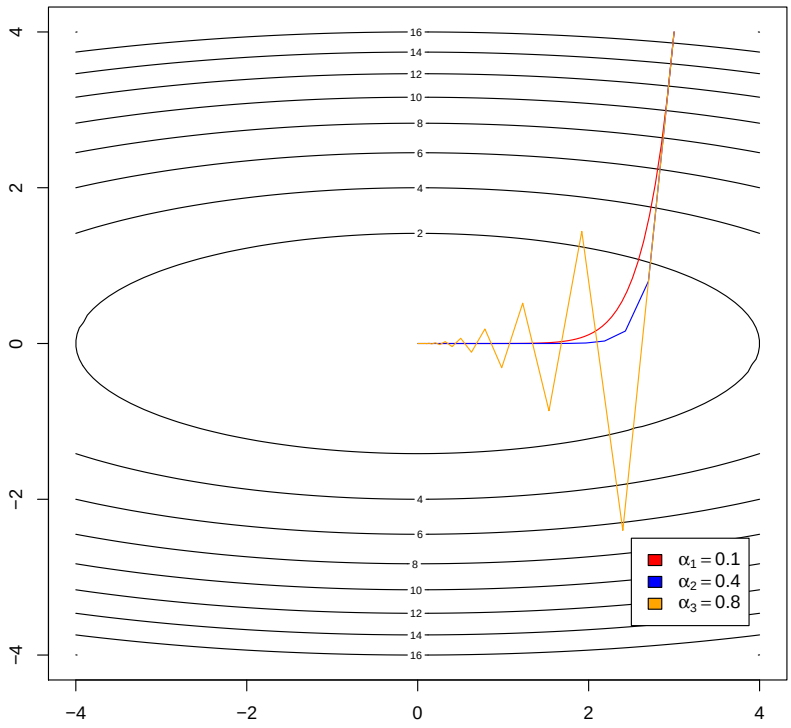

Figure 4.9: The impact of learning rate on the path traveled by the gradient descent algorithm on function f.

4.6 Gradient descent in neural networks

Gradient descent is often used when dealing with neural networks. It is a core optimization algorithm behind training of the neural network. The objective during training is to minimize a cost function, which measures the difference between the predicted outputs of the neural network and the actual target values.

Gradient descent is well-suited to use while optimizing multi-argument functions. Problems for which neural networks are used often deal with such functions. One of the common tasks for which neural networks are used is image classification.

When creating the neural network we need to specify its architecture, including the number of layers, the number of neurons in each layer, and the activation functions. The first layer of neurons, being the input layer, is directly related to the type of input data. In image classification, number of neurons in this layer is usually equal to number of pixels in analyzed image multiplied by 3 when dealing with colored input (for each of RGB channels).

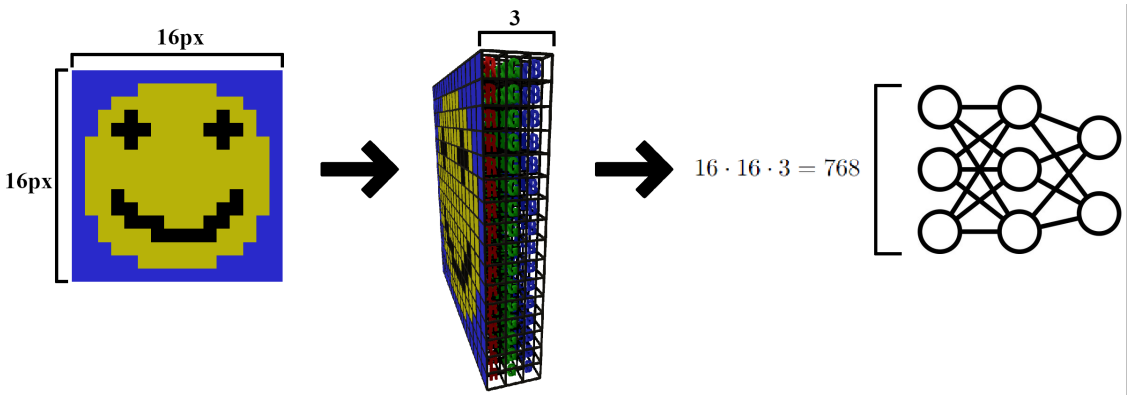

Figure 4.10: Number of pixels multiplied by 3 RGB channels as input for the neural network

In addition to the input layer, we also need to determine the number of neurons in the hidden layers and the output layer. The number and structure of hidden layers can vary depending on the chosen neural network approach. The number of neurons in the output layer however usually corresponds to the number of possible results - categories.

Each neuron in the network has connections with neighboring layers. This connections as well as neurons have special parameters called weights and biases. Weights represent the strength of connections between neurons in different layers of a neural network. Each connection between neurons is associated with a weight, and these weights are adjusted during the training process to enable the network to make accurate predictions. Biases are additional parameters in each neuron that allow the network to shift the activation function. They provide the model with flexibility to account for situations where the input to the neuron is insufficient to activate it. Biases are adjusted during training, along with weights, to improve the overall performance of the network. In a neural network layer, the output of each neuron is computed by applying a weighted sum of the inputs, followed by an activation function. Mathematically, the output O_j of neuron j in layer is given by:

$$O_j = \sigma\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right) \tag{4.8}$$

where w_{ij} is the weight of the connection between neuron i in the previous layer and neuron j in the current

layer, x_i is the input from neuron i, b_j is the bias of neuron j, σ is the activation function, and n is the number of neurons in the previous layer.

Weights and biases are chosen at random at first, but need to be adjusted during training of neural network. As one would expect, the results of randomly initialized networks are not good in most cases. To estimate how well a neural network performs, a so-called *costfunction* is created. It is a mathematical measure that quantifies the difference between the predicted output of the network and the actual target values. Common cost function used for regression problems, where the goal is to predict a continuous value, is called Mean Squared Error (MSE). The mean squared error is the average of the squared differences between the predicted and actual values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.9}$$

where n is the number of data points, y_i is the actual target, and $\hat{y}_i$ is the predicted output.

We can think of the cost function as a function that accepts all network weights and biases as input, and outputs one number that is an assessment of the network's performance. This multi-parameter function is then minimized using gradient descent. The vector created using gradient descent contains a number of changes that should occur in each of the weights and biases to approach the minimum of the cost function - so that the results obtained by the network are closer to the expected results.

$$\nabla C(w_0, w_1, ..., w_n) \tag{4.10}$$

where ∇C is gradient used in gradient descent, n is combined number of all the weights and biases and w_n is value of n-th weight or bias.

4.7 Steepest descent

Steepest descent is another optimization algorithm commonly used to minimize nonlinear analytical functions. It is very similar to the gradient descent algorithm in its structure. The goal of steepest descent is also to iteratively move towards the minimum of a function by adjusting its parameters. However, in contrary to gradient descent, we do not set the learning rate parameter of the algorithm upfront. We only provide the maximum learning rate value (maximum step size in direction of antigradient), and then the algorithm determines the optimal value of this parameter.

Definition 6: Steepest descent

Given that steepest descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - \alpha_{k-best}\nabla_f(x_k) \tag{4.11}$$

where $k \in \mathbb{N}$ is an iteration number, $\nabla_f(x_k)$ is the gradient of a function f in point x_k and α_{k-best} is the best learning rate found over g steps of linear search.

Using this algorithm, we are not only taking steps in the optimal direction, but also automatically select the best size of those steps. We can calculate the best step size or learning rate in several ways. One way is

to take advantage of the gradient descent formula and use it to calculate the its derivative over the learning rate α. Finding the minimum of said derivative would give us the optimal α parameter.

$$x_{k+1} = \underset{\alpha}{\operatorname{argmin}} \frac{d}{d\alpha} \left(x_k - \alpha \nabla_f(x_k)\right) \tag{4.12}$$

Unfortunately, there is a problem with this solution. If we were to use derivative in our equation, we would end up with symbolical algorithm. Implementing such an algorithm would require us to be able to quickly calculate the derivative of any function. This is not always an easy task. It is easier for computers to perform calculations using numerical algorithms. This leads us to the next solution - approximating the optimal learning rate value.

To approximate step size we can linearly search through different points along the direction pointed by the antigradient and check what results we get with them. Of course, we don't want to search through the infinite number of points we can find on this line. For this purpose, we should set the maximum search range (i.e. the upper α limit - α_{max}) as well as the starting point of the search x_k. However, there are also an infinite number of points on the section. Therefore, we also set the g parameter, which determines how many points we should check on this section.

The steepest descent algorithm resembles the gradient descent algorithm with one key difference. In each step of the algorithm, we determine g points in the direction of the antigradient and arrange them at an equal distance from each other on a section of length controlled by α_{max}. It is equivalent to discreetly going over g steps from $\alpha = 0$ to $\alpha = \alpha_{max}$ and choosing the best outcome.

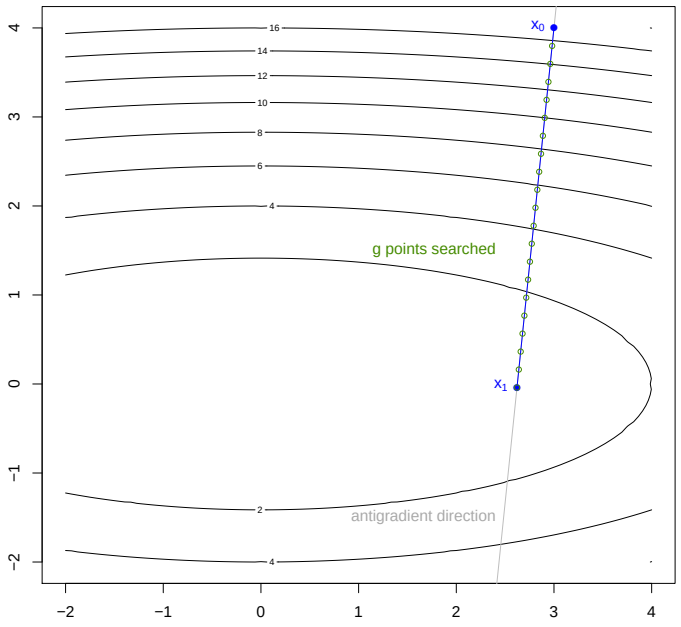

Figure 4.11: Points considered during one iteration of steepest descent algorithm of function f

This is somewhat of a heuristic approach. This algorithm will not provide us with optimal value of the step

size but an approximation of one. With a sufficiently large parameter g, we will obtain a value close to the optimal at a relatively low computational cost. The main advantage of this algorithm compared to the gradient descent algorithm is a better adjusted step size in each iteration. Unfortunately, this also has disadvantages - including higher computational costs of a single iteration, which, however, is often offset in the long run by the better quality of the obtained steps.

Example 6. Let $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

Additionally, let the parameters of steepest descent algorithm be $\alpha_{max} = 5$ and the parameter $g = 1000$. How steepest descent converges using this parameters can be seen on the plot below (alongside with path created by gradient descent):

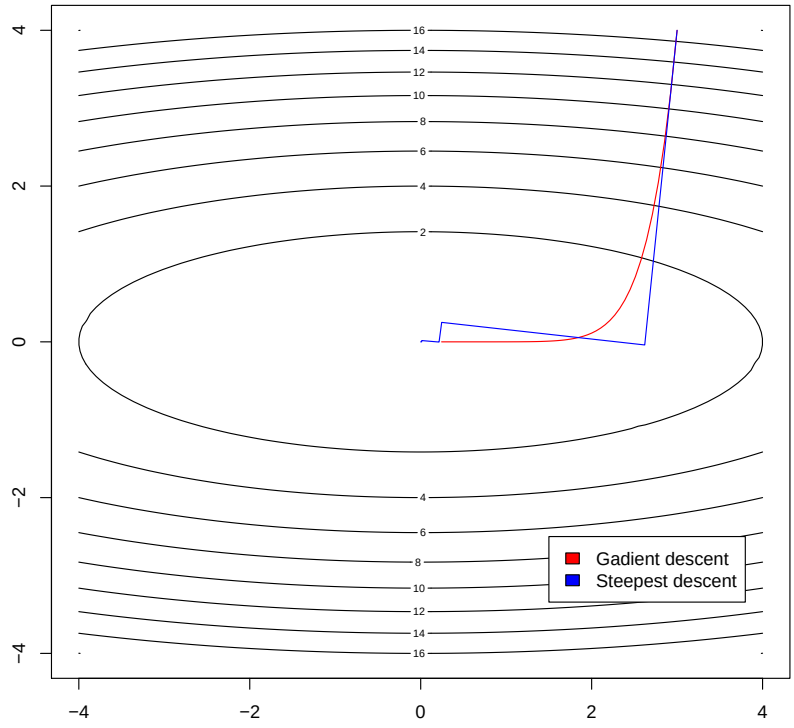

Figure 4.12: Comparison of the path obtained by the steepest descent and gradient descent algorithms on function f in 2D space

We can observe an interesting pattern in the path created by the steepest descent algorithm. The consecutive steps of the algorithm iterations create sections perpendicular to each other. The same pattern repeats itself and gets closer and closer to the extremum of the function. This behavior is not random or accidental. This is due to the fact that the gradient of the function at any point does not have to indicate the global extremum, but the direction of the fastest decline of the function value. The step size is adjusted linearly until the function stops decreasing and reaches stationarity. It would then reach the tangent space, where the function stops decreasing (and as it moves further, it starts increasing). From the newly determined point, since it lies on tangent space, the direction

of the function's fastest decrease is at an angle of 90^o from the direction of the last step.

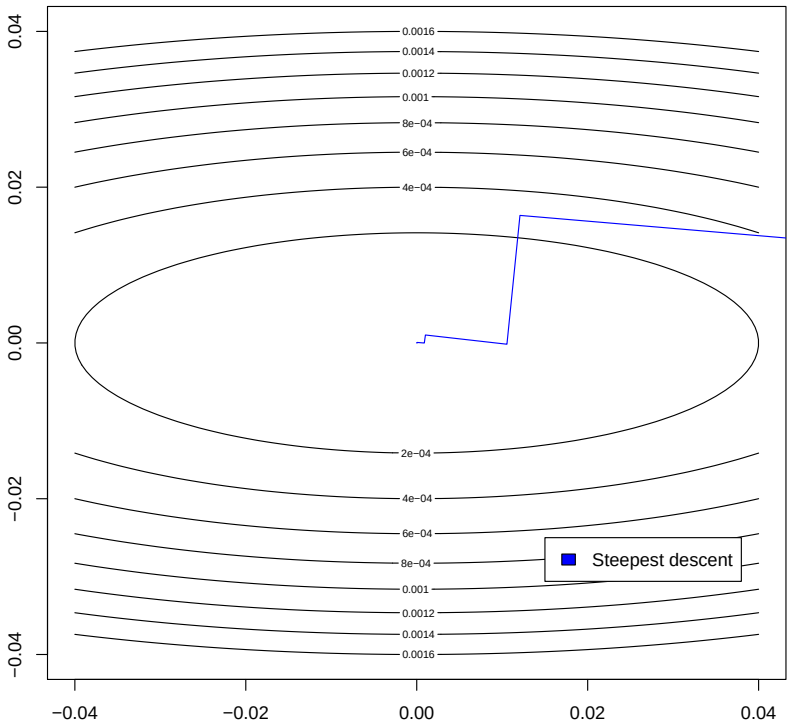

Figure 4.13: Closeup of the path obtained by the steepest descent algorithm on function f in 2D space

Using programming language **R** we can write a steepest descent implementation. First of all, we need a function to linearly search the best solution over g points:

```
line_search <- function(f, x0, x1, g = 100) {
  #' Line search
  #'
  #' @description Helping function responsible for finding the best point judging by
    function f
  #' over g linearly distributed points.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x0 numerical vector. The starting point for the algorithm.
  #' @param x1 numerical vector. The end point for the algorithm (or max range of step).
  #' @param g scalar. Optional parameter that is responsible for the number of iterations of
     searching for the best step size
  #' in each iteration of the algorithm itself (defaults to 100).
  #'
  #' @usage line_search(f, x0, x1, g)
  #'
  #' @returns
  #' x_best: best found point judging by function f

    # setting x0 as a starting point
```

```
    x_best <- x0
    # looping over g points in direction of point x1
    for(i in 1 : g) {
        t <- i / g
        x_t <- t*x1+(1-t)*x0
        if(f(x_t) < f(x_best)) {
            x_best <- x_t
        } else {
            break
        }
    }
    return(x_best)
}
```

Listing 5: Line search implementation

Heaving the line search function ready, we can proceed to implement the main part of the algorithm. It can be done as a following function *steepest descent* that is based on previous gradient descent implementation:

```
steepest_descent <- function(f, x, a = 5, g = 100, K = 100){
  #' Steepest Descent
  #'
  #' @description Function responsible for calculating steepest descent
  #' of function f in g points each iteration over K steps.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x numerical vector. The starting point for the algorithm.
  #' @param a scalar. Optional parameter that determines max learning rate (defaults to 5).
  #' @param g scalar. Optional parameter that is responsible for the number of iterations of searching for the best step size
  #' in each iteration of the algorithm itself (defaults to 100).
  #' @param K scalar. Optional parameter that determines maximum iteration limit (defaults to 100).
  #'
  #' @usage steepest_descent(f, x, a, g, K)
  #'
  #' @returns
  #' List of various outputs containing the following:
  #' * x_opt: found solution,
  #' * f_opt: value of target function in the found solution,
  #' * x_hist: history of explored solutions,
  #' * f_hist: history of target function values,
  #' * t_eval: time elapsed during algorithm calculations.

  start_time <- Sys.time()
  results <- list(x_opt = x,
                  f_opt = f(x),
                  x_hist = matrix(NA, nrow = K, ncol = length(x)),
                  f_hist = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)

  for(k in 2:  K){
    # description of the transition from point x_k to x_k+1
    x_new <- line_search(f, x, x - a * grad(f, x), g)

    # checking whether the new solution
    # is the best so far
    if(f(x_new) < results$f_opt){
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }
```

```
44
45      results$x_hist[k,] <- x_new
46      results$f_hist[k] <- f(x_new)
47
48      x <- x_new
49    }
50    # time difference between the end and start of the algorithm
51    results$t_eval <- Sys.time() - start_time
52    return(results)
53  }
```

Listing 6: Steepest descent implementation