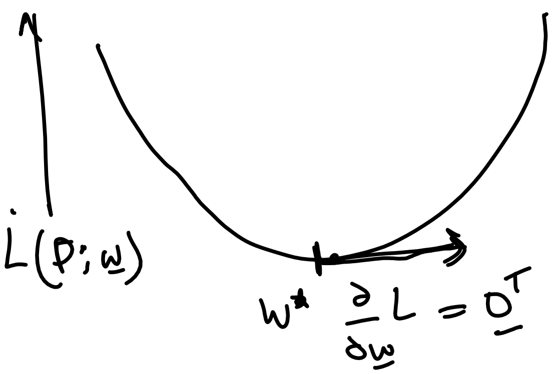
Training is about minimizing a loss function

① If the Loss function is quadratic

$$\underline{0}^T = \frac{\partial}{\partial \underline{w}} L(D; w) \rightarrow \text{Linear and hence solvable in closed form}$$

$\hookrightarrow \leq$ Degree 4

② Optimization of convex functions by Gradient Descent.
minimization



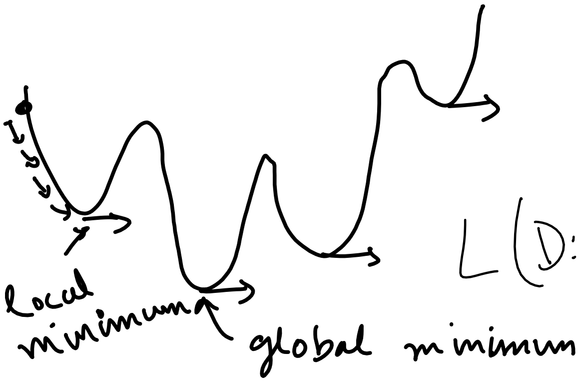
stepsize or learning rate

$$\underline{w}_1 = \underline{w}_0 - \alpha_0 \left[\frac{\partial}{\partial \underline{w}} L(D; \underline{w})\right]^T$$

$$\underline{w}_{t+1} = \underline{w}_t - \alpha_t \left(\frac{\partial}{\partial \underline{w}} L(D; \underline{w})\right)^T$$

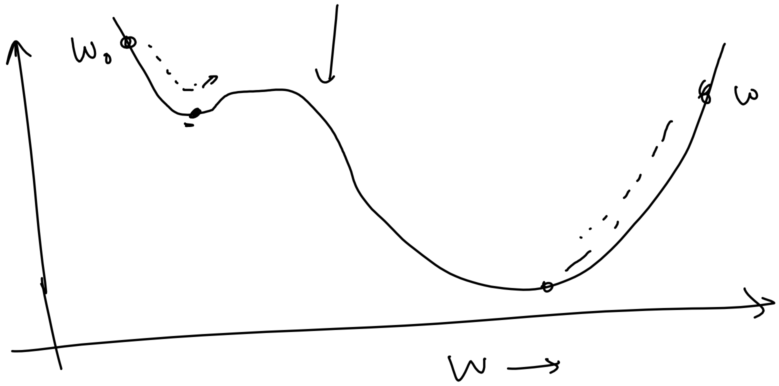
while $\|\nabla_{\underline{w}} L(D; w)\| \leq 10^{-4}$

$$\underbrace{\underline{w}_{t+1} = \underline{w}_t}_{\text{parameter vector}} - \underbrace{\alpha_t}_{\text{step size / learning rate}} \underbrace{\nabla_{\underline{w}} L(D; \underline{w})}_{\text{gradient}}$$



$L(D:w)$

non-convex function

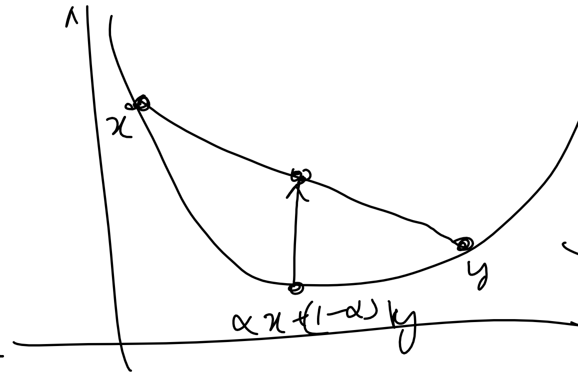


Plural : minima

Convex function

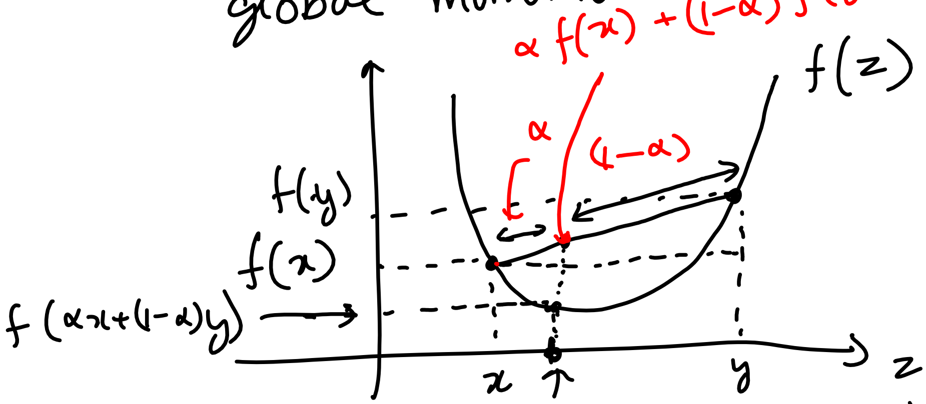
$f(\alpha x + (1-\alpha) y) \leq \alpha f(x) + (1-\alpha) f(y)$

$\alpha \in [0,1]$



① The function being optimized must be continuous
continuous

② Convex function : On a convex function gradient descent ~~provides~~ converges to global minimum



$0.25x + (1-0.25)y$

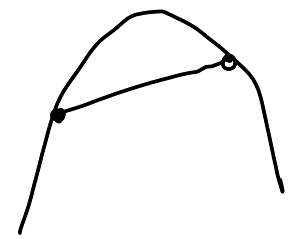
$\alpha x + (1-\alpha) y \quad \forall \alpha \in [0,1]$

$\alpha f(x) + (1-\alpha) f(y) \geq f(\alpha x + (1-\alpha) y)$

Second Derivative

$\frac{d^2 f(x)}{dx^2}$ describes the curvature of the function

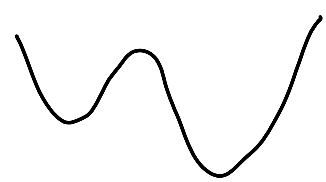
$\frac{d^2 f(x)}{dx^2} > 0 \quad \forall x \Leftrightarrow$ convex function



$\frac{d^2 f(x)}{dx^2} < 0 \quad \forall x \Leftrightarrow$  concave function

Disciplined Convex Programming

Convex



non convex
non concave

$$f_1(x) + f_2(x)$$

cvxpy → optimization

cvxpy → optimization with constraints

$$\underset{\underline{w}}{\text{minimize}} \; L(D; w)$$

$$\text{subject to} \quad \left.\begin{array}{l} A\underline{w} \geq b \\ \|\underline{w}\| = 1 \end{array}\right\} \text{constraint}$$

Course on optimization
MEE 459/559

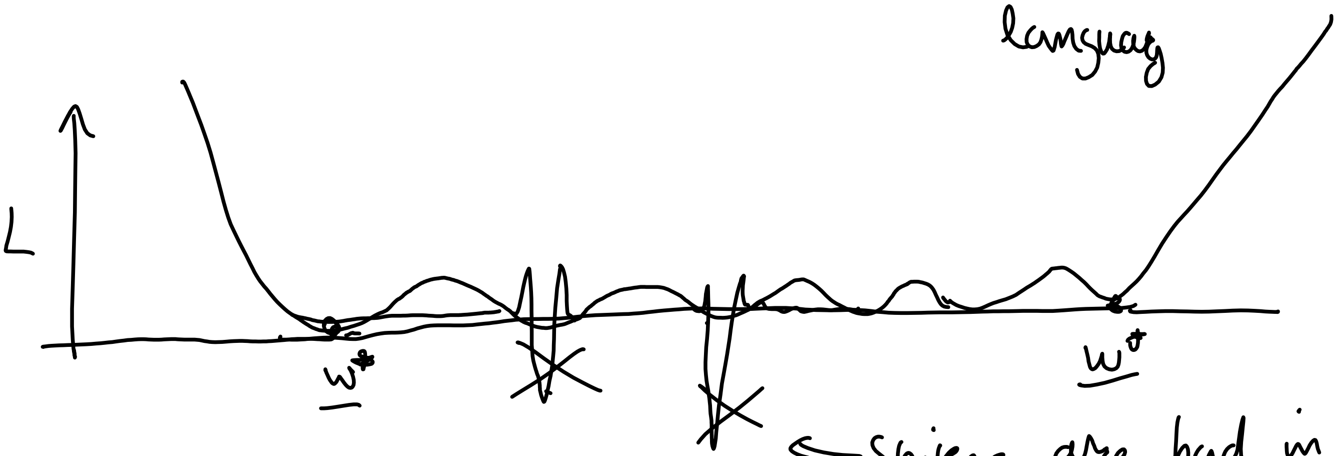
Problems in Neural Networks are non-convex

Answer: We dont know | NN people got lucky
we told you so

Brains (NNs) are interested in these problem

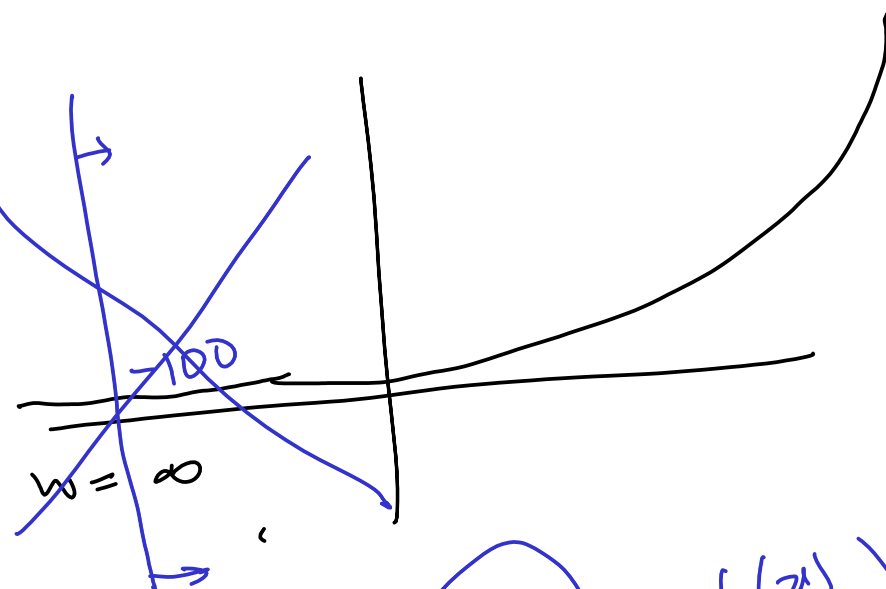
Hypothesis → The problems $\underline{\underline{we}}$ are interested in

vision
language



← Spikes are bad in case of
<u>Large data</u> problems
stochastic

Gradient Descent → Stochastic Gradient Descent



$$\exp(-w^T x) = 0$$

input independence

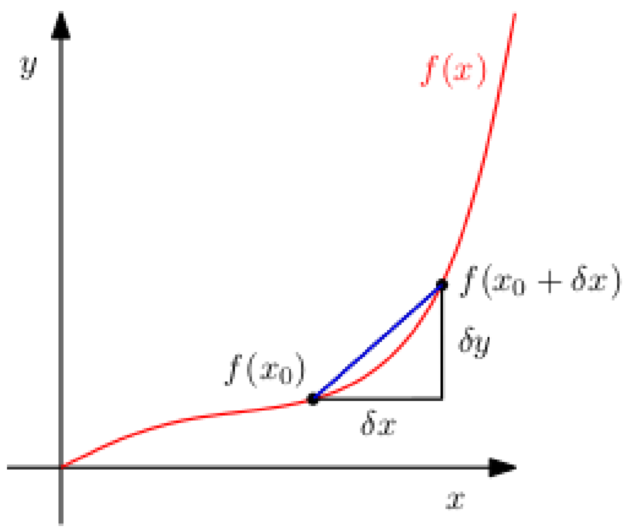
Continuous Optimization

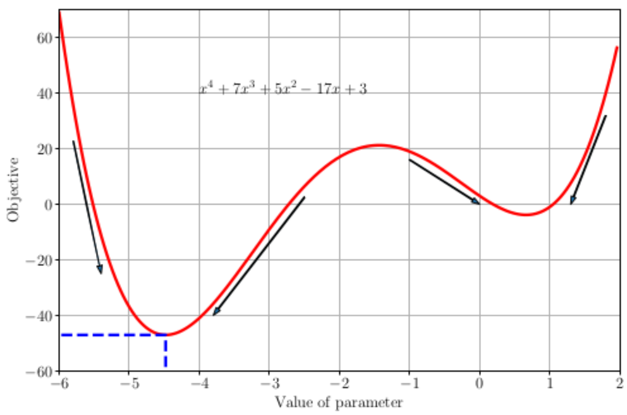
Vikas Dhiman

Thursday 25th September, 2025

Reading: Chapter 7: MML Book

0.1 Recall geometry of a derivative





1 Minimizing general functions

We cannot minimize general functions by solving

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{0}^\top \tag{1}$$

because the equation might not have a formula for it.

Instead we use iterative methods like gradient descent minimize general function $f(\mathbf{x})$.

1.0.1 Definition (Directional derivative)

Directional dervivative of a function $f(\mathbf{x}) : R^n \to R$ with respect to a given unit vector $\hat{\mathbf{u}} \in R^n | \|\hat{\mathbf{u}}\| = 1$ is defined as

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \hat{\mathbf{u}}) - f(\mathbf{x})}{\epsilon} \tag{2}$$

Ref Khan Academy
Ref Libretexts

Vector calculus chain rule (a theorem) Given a function composition $\mathbf{f}(\mathbf{x}) = \mathbf{g}(\mathbf{h}(\mathbf{x})) = (\mathbf{g} \circ \mathbf{h})(\mathbf{x})$ where $\mathbf{h} : R^n \to R^m$, $\mathbf{g} : R^m \to R^p$ and $\mathbf{h} : R^m \to R^p$

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \tag{3}$$

or denoting the derivatives as Jacobian matrices we have,

$$\mathcal{J}_{\mathbf{x}} \mathbf{f} = \mathcal{J}_{\mathbf{h}}[\mathbf{f}] \mathcal{J}_{\mathbf{x}}[\mathbf{h}] \tag{4}$$

Theorem (Directional derivative is gradient dot product with the direction) Express the trajectory in the direction $\mathbf{u}$ as a function of time t as

$$\mathbf{g}(t) = \mathbf{x} + t\hat{\mathbf{u}} \tag{5}$$

Note that the Jacobian of $\mathbf{g}(t)$ wrt t is simply $\mathbf{u}$,

$$\mathcal{J}_t \mathbf{g}(t) = \hat{\mathbf{u}} \tag{6}$$

Recall the definition of directional derivative,

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon\hat{\mathbf{u}}) - f(\mathbf{x})}{\epsilon}. \tag{7}$$

Compare it with the derivative of $f(\mathbf{g}(t))$ with respect to t at $t = 0$

$$\frac{\partial f(\mathbf{g}(t))}{\partial t} = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + (t+\epsilon)\hat{\mathbf{u}}) - f(\mathbf{x} + t\hat{\mathbf{u}})}{\epsilon}\Big|_{t=0}. \tag{8}$$

$$\frac{\partial f(\mathbf{g}(t))}{\partial t} = \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon\hat{\mathbf{u}}) - f(\mathbf{x})}{\epsilon} = D_{\mathbf{u}} f(\mathbf{x}). \tag{9}$$

We can compute $\frac{\partial f(\mathbf{g}(t))}{\partial t}$ by chain rule,

$$D_{\mathbf{u}} f(\mathbf{x}) = \mathcal{J}_t f(\mathbf{g}(t)) = \mathcal{J}_{\mathbf{x}} f(\mathbf{x}) \mathcal{J}_t \mathbf{g} = \nabla_{\mathbf{x}}^{\top} f(\mathbf{x}) \hat{\mathbf{u}} \tag{10}$$

1.0.2 Theorem : The direction of stepest ascent and descent

Let $\hat{\mathbf{u}}$ be of unit magnitude. The directional derivative represents how the function changes in the direction $\hat{\mathbf{u}}$.

$$D_{\hat{\mathbf{u}}} f(\mathbf{x}) = \nabla_{\mathbf{x}}^{\top} f(\mathbf{x}) \hat{\mathbf{u}} = \|\nabla_{\mathbf{x}} f(\mathbf{x})\| \cos(\theta), \tag{11}$$

where θ is the angle between $\nabla_{\mathbf{x}} f(\mathbf{x})$ and $\hat{\mathbf{u}}$. The change is maximum when $\theta = 0$ and $\cos(\theta) = 1$ and the change is minimum when $\theta = 180°$ and $\cos(\theta) = -1$.

In other words the function f increases the most (stepest ascent) when $\hat{\mathbf{u}} \propto \nabla_{\mathbf{x}} f(\mathbf{x})$ and decreases the most (steepest descent) when $\hat{\mathbf{u}} \propto -\nabla_{\mathbf{x}} f(\mathbf{x})$.

1.1 Gradient descent method

(Section 9.3 of Convex Optimization by Stephen Boyd and Lieven Vandenberghe)

1. Start from a random point $\mathbf{x}_0$, $\mathbf{x}_t \leftarrow \mathbf{x}_0$.
2. Move in the direction opposite to $\nabla_{\mathbf{x}} f(\mathbf{x})$. If we were at $\mathbf{x}_t$, then the next point is at $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \nabla_{\mathbf{x}} f(\mathbf{x})$, where $\alpha_t > 0$ is a positive scalar, called the step size or the learning rate.

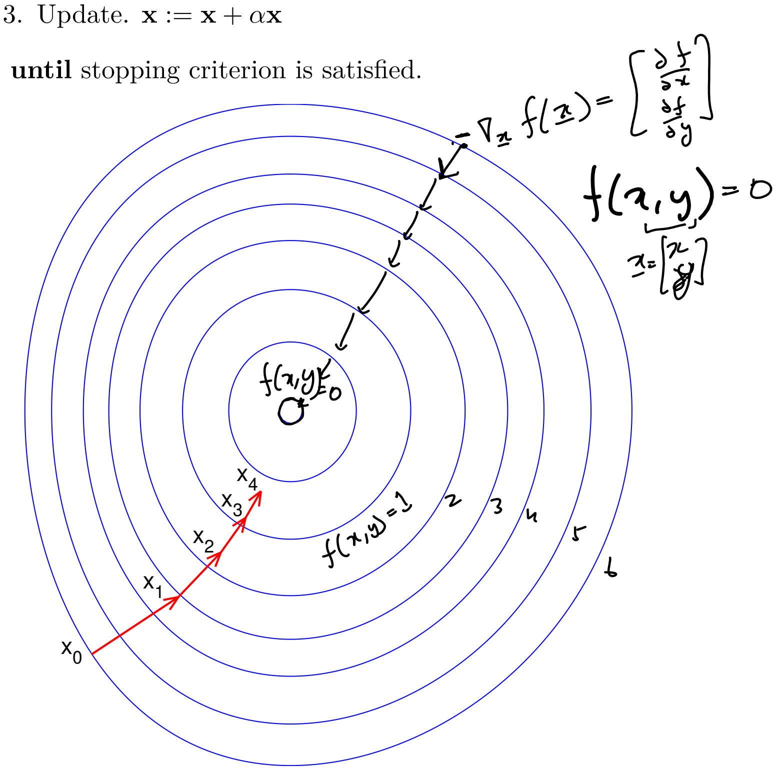
3. Stop when the gradient is almost zero $\|\nabla_{\mathbf{x}} f(\mathbf{x})\| < 10^{-4}$.

This corresponds to the following algorithm:
$\mathbf{x}_t = \mathbf{x}_0$

while ($\|\nabla_{\mathbf{x}} f(\mathbf{x})\| > 10^{-4}$) { $\mathbf{x}_t \leftarrow \mathbf{x}_t - \alpha_t \nabla_{\mathbf{x}} f(\mathbf{x})$ }
Algorithm 9.3 Gradient descent method.
given a starting point $\mathbf{x} \in \mathrm{dom} f$.
repeat

1. $\Delta x = -\nabla f(\mathbf{x})$.
2. Choose step size α
3. Update. $\mathbf{x} := \mathbf{x} + \alpha \mathbf{x}$

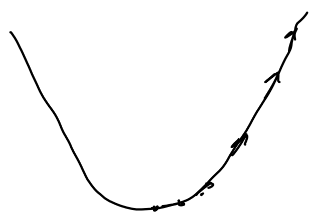
until stopping criterion is satisfied.

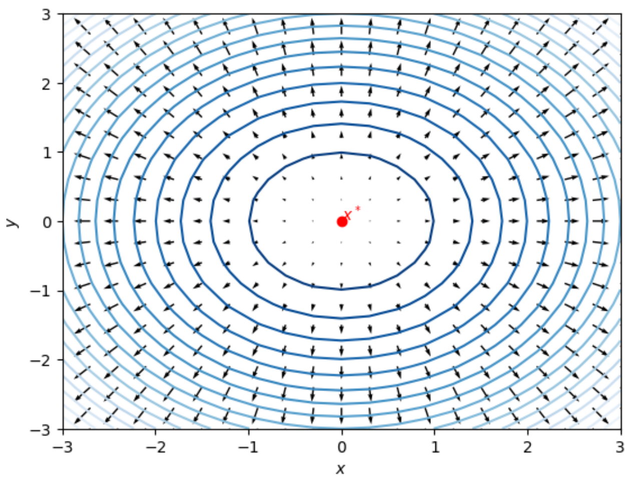


2 Gradient visualization

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation, rc
rc('animation', html='jshtml')
```

```
def plot_gradients(func, gradfunc):
    x, y = np.mgrid[ -3:3:21j,
                     -3:3:21j]
    bfx = np.array([x, y])
    f = func(x,y)
    [dfdx, dfdy] = gradfunc(x,y)
    fig, ax = plt.subplots()
    ctr = ax.contour(x, y, f, 20, cmap='Blues_r')
    ax.quiver(x, y, dfdx, dfdy)
    ax.plot([0], [0], 'ro')
    ax.text(0, 0, '$x^*$', color='r')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    plt.show()

def f(x, y): return  x**2 + y**2
def gradf(x, y): return 2*x, 2*y
plot_gradients(f, gradf)
```





2.1 Quadratic function example:

A quadratic function $f(\mathbf{x}) = x_1^2 + x_2^2$ has level sets as circles:

$$S(f, c) = \{\mathbf{x} : x_1^2 + x_2^2 = c\} \tag{12}$$

It has the parameteric form as

$$S(f, c) = \{\mathbf{g}(c, \theta) = \begin{bmatrix} \sqrt{c}\cos(\theta) \\ \sqrt{c}\sin(\theta) \end{bmatrix} : \theta \in [0, 2\pi)\} \tag{13}$$

The gradient is:

$$\nabla_{\mathbf{x}} f(\mathbf{x})^\top = 2\mathbf{x}^\top = 2\begin{bmatrix}\sqrt{c}\cos(\theta) & \sqrt{c}\sin(\theta)\end{bmatrix} \tag{14}$$

and
The derivative of curve with respect to θ the tangent to the curve:

$$\mathcal{J}_\theta \mathbf{g}(c, \theta) = \begin{bmatrix}\frac{\partial}{\partial\theta}\sqrt{c}\cos(\theta) \\ \frac{\partial}{\partial\theta}\sqrt{c}\sin(\theta)\end{bmatrix} = \begin{bmatrix}-\sqrt{c}\sin(\theta) \\ \sqrt{c}\cos(\theta)\end{bmatrix} \tag{15}$$

$$\nabla_{\mathbf{x}} f(\mathbf{x})^\top \mathcal{J}_\theta \mathbf{g}(c, \theta) = 2\begin{bmatrix}\sqrt{c}\cos(\theta) & \sqrt{c}\sin(\theta)\end{bmatrix}\begin{bmatrix}-\sqrt{c}\sin(\theta) \\ \sqrt{c}\cos(\theta)\end{bmatrix} = 0 \tag{16}$$

2.2 Taylor series approximation

2.2.1 A bit more about the step size/learning rate

Taylor series expansion originates from integration by parts

The fundamental theorem of calculus states that

$$f(x) = f(a) + \int_a^x f'(t)\, dt \tag{17}$$

Now we can integrate by parts and use the fundamental theorem of calculus again to see that

$$f(x) = f(a) + \Big(xf'(x) - af'(a)\Big) - \int_a^x tf''(t)\, dt \tag{18}$$

$$= f(a) + x\left(f'(a) + \int_a^x f''(t)\, dt\right) - af'(a) - \int_a^x tf''(t)\, dt \tag{19}$$

$$= f(a) + (x-a)f'(a) + \int_a^x (x-t)f''(t)\, dt, \tag{20}$$

Recall the Taylor series expansion of a function $f(x)$ around x_0

$$f(x) = f(x_0) + \frac{df(x)}{dx}(x-x_0) + \frac{1}{2!}\frac{d^2 f(x)}{dx^2}(x-x_0)^2 + \dots + \frac{1}{n!}\frac{d^n f(x)}{dx^n}(x-x_0)^n + \dots \tag{21}$$

Vectorized Taylor series expansion is

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla_{\mathbf{x}}^\top f(\mathbf{x})(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2!}(\mathbf{x} - \mathbf{x}_0)^\top \underbrace{\mathcal{H}f(\mathbf{x})}(\mathbf{x} - \mathbf{x}_0)^\top + \dots \infty \tag{22}$$

While optimizing around the point $\mathbf{x}_t$, we can use the Taylor series expansion to find a local quadratic approximation to find the next best minima:

What is the second derivative in vector form?

Hessian: defined for a scalar valued vector function

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$Hf(\underline{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & & \vdots \\ \vdots & & \ddots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}_n \qquad [Hf(\underline{x})]_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left(\frac{\partial f}{\partial x_j}\right)$$

$$= \frac{\partial}{\partial \underline{x}}\left(\begin{bmatrix} - \frac{\partial f}{\partial \underline{x}} - \\ \\ \end{bmatrix}_{1\times n}^{T} \right) = J_{\underline{x}}\left[\nabla_{\underline{x}} f(\underline{x})\right]$$

Second derivative ≥ 0 in vector domain?

A sq matrix A is positive definite iff $A \in \mathbb{R}^{n\times n}$

$H_x f(\underline{x}) \succ 0$: $\underline{x}^T A \underline{x} > 0 \quad \forall \underline{x} \in \mathbb{R}^n$

$H_x f(\underline{x}) \prec 0$ negative definite

Test for positive definiteness

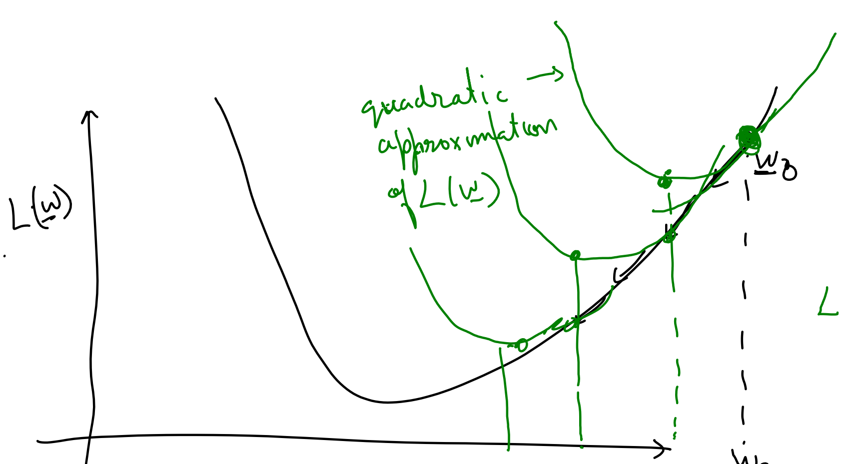
$A \succ 0 \iff$ all the eigen values > 0

$$\underbrace{A\underline{v} = \lambda \underline{v}}_{\text{solutions to the eqn}}$$

λs are called eigen values

$\underline{v}$s are called eigen vectors

Why are second derivatives important but not often used in optimization?



$$\underline{w}_{t+1} = \underline{w}_t - \alpha_t \nabla_{\underline{w}} L(\underline{w})$$

step size / learning rate

$$L(\underline{w}) = \overset{\text{const}}{L(\underline{w}_t)} + \overset{\text{linear}}{\nabla_w^T L(\underline{w}_t)(\underline{w}-\underline{w}_t)}$$

$$+ \underbrace{\frac{1}{2}(\underline{w}-\underline{w}_t)^T \mathcal{H} L(\underline{w}_t)(\underline{w}-\underline{w}_t)}_{\text{quadratic}} + \ldots \infty$$

$$\underline{w}_{t+1} = \underline{w}_t - \left[\mathcal{H} L(\underline{w}_t)\right]^{-1} \nabla_{\underline{w}} L(\underline{w}_t)$$

Newton's method

Why not in NN? $\longrightarrow$ $\underline{w}$ is big. $4\underline{b}$, $7\underline{b}$, $27\underline{b}$

$$\underline{w} \in \mathbb{R}^n \implies HL(\underline{w}) \in \mathbb{R}^{n \times n}$$

$$\begin{bmatrix} \ddots_n & 0 \\ 0 & \ddots \end{bmatrix}$$

$- \quad \underline{b}$

Gauss Newton

$$L(D, w) = \| \underline{l}_D(\underline{w}) \|_2^2$$

$$\underline{l}(\underline{w}) = \begin{bmatrix} y_i - \underline{w}^T x_i \\ \vdots \\ \underline{y_n} - \underline{w}^T \underline{x_n} \end{bmatrix}$$

$$H\, L(D; w) \approx \left[J_{\underline{w}} \underline{l}_D(\underline{w}) \right] \left[J_{\underline{w}} \underline{l}_D(\underline{w}) \right]^T$$

Why not in convex opt? $\longrightarrow$ because second derivatives are hard to find.

$$\hat{f}(\mathbf{x}_{t+1}) = f(\mathbf{x}_t) + \nabla_{\mathbf{x}}^{\top} f(\mathbf{x})(\mathbf{x}_{t+1} - \mathbf{x}_t) + \frac{1}{2!}(\mathbf{x}_{t+1} - \mathbf{x}_t)^{\top} \mathcal{H} f(\mathbf{x})(\mathbf{x}_{t+1} - \mathbf{x}_t)^{\top} \quad (23)$$

At the optimal point of the quadratic approximation the derivative is zero:

$$\frac{\partial \hat{f}(\mathbf{x}_{t+1})}{\partial \mathbf{x}_{t+1}} = \mathbf{0}^{\top} \quad (24)$$

$$\nabla_{\mathbf{x}}^{\top} f(\mathbf{x}) + (\mathbf{x}_{t+1} - \mathbf{x}_t)^{\top} \mathcal{H} f(\mathbf{x}) = \mathbf{0}^{\top} \quad (25)$$

Taking transpose and rearranging the terms we get:

$$\mathbf{x}_{t+1} - \mathbf{x}_t = -[\mathcal{H} f(\mathbf{x})]^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (26)$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t - [\mathcal{H} f(\mathbf{x})]^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (27)$$

This is the update rule for the Newton's method for optimization. Note that the step size here is inversely proportional the second derivative.

2.3 Taylor series approximation

Approximate the following function to a quadratic function near the point $\mathbf{x}_0 = [-2, 3]$

$f(x) = 0.06 \exp(2x_1 + x_2) + 0.05 \exp(x_1 - 2x_2) + \exp(-x_1)$

$$f(\mathbf{x}) = 0.06 \exp([2, 1]\mathbf{x}) + 0.05 \exp([1, -2]\mathbf{x}) + \exp([-1, 0]\mathbf{x}) \quad (28)$$

Let $\mathbf{b}_1^{\top} = [2, 1]$, $\mathbf{b}_2^{\top} = [1, -2]$, $\mathbf{b}_3^{\top} = [-1, 0]$.

$$\nabla_{\mathbf{x}}^{\top} f(\mathbf{x}) = 0.06\mathbf{b}_1^{\top} \exp(\mathbf{b}_1^{\top}\mathbf{x}) + 0.05\mathbf{b}_2^{\top} \exp(\mathbf{b}_2^{\top}\mathbf{x}) + \mathbf{b}_3^{\top} \exp(\mathbf{b}_3^{\top}\mathbf{x}) \quad (29)$$

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = 0.06\mathbf{b}_1 \exp(\mathbf{b}_1^{\top}\mathbf{x}) + 0.05\mathbf{b}_2 \exp(\mathbf{b}_2^{\top}\mathbf{x}) + \mathbf{b}_3 \exp(\mathbf{b}_3^{\top}\mathbf{x}) \quad (30)$$

$$H f(\mathbf{x}) = \nabla_{\mathbf{x}}^{\top}(\nabla_{\mathbf{x}} f(\mathbf{x})) = 0.06\mathbf{b}_1 \exp(\mathbf{b}_1^{\top}\mathbf{x})\mathbf{b}_1^{\top} + 0.05\mathbf{b}_2 \exp(\mathbf{b}_2^{\top}\mathbf{x})\mathbf{b}_2^{\top} + \mathbf{b}_3 \exp(\mathbf{b}_3^{\top}\mathbf{x})\mathbf{b}_3^{\top} \quad (31)$$

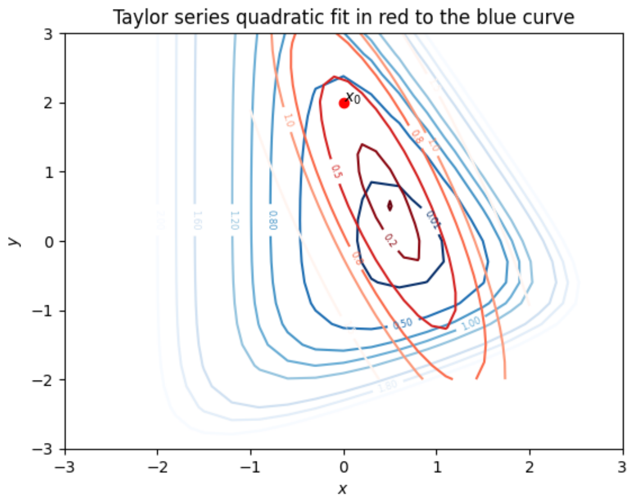
$$H f(\mathbf{x}) = 0.06 \exp(\mathbf{b}_1^{\top}\mathbf{x})\mathbf{b}_1\mathbf{b}_1^{\top} + 0.05 \exp(\mathbf{b}_2^{\top}\mathbf{x})\mathbf{b}_2\mathbf{b}_2^{\top} + \exp(\mathbf{b}_3^{\top}\mathbf{x})\mathbf{b}_3\mathbf{b}_3^{\top} \quad (32)$$

```
# Define the function
def f(x):

    """
    1. For an input x of shape x.shape = (2,)
       f(x) must return a scalar
```

```
    2. For an input x of shape x.shape = (m, 2),
        f(x) must return an array of shape (m,)
        which contains f(x) is computed for m values of x

    3. For an input x of shape x.shape = (m, n, 2)
        f(x) must return an array of shape (m, n)
        which contains f(x) is computed for (m x n) values of x
    """
    return (0.06 * np.exp(x @ [2, 1])
            + 0.05* np.exp(x @ [1, -2])
            + np.exp(x @ [ -1, 0]))

# Compute its derivative, the gradient function
def grad_f(x):
    """
    1. For an input x of shape x.shape = (2,)
        grad_f(x) must return a n array of shape (2,)

    2. For an input x of shape x.shape = (m, 2),
        grad_f(x) must return an array of shape (m, 2)
        which contains grad_f(x) is computed for m values of x

    3. For an input x of shape x.shape = (m, n, 2)
        grad_f(x) must return an array of shape (m, n, 2)
        which contains grad_f(x) is computed for (m x n) values of x
    """
    coeff1 = np.array([2, 1])
    coeff2 = np.array([1, -2])
    coeff3 = np.array([ -1, 0])
    # Slicing using np.newaxis or None, increases the dimension by 1.
    # https://numpy.org/doc/stable/reference/constants.html#numpy.newaxis
    return (0.06 * np.exp(x @ coeff1)[..., None] * coeff1
            + 0.05 * np.exp(x @ coeff2)[..., None] * coeff2
            + np.exp(x @ coeff3)[..., None] * coeff3)

def numerical_jacobian(f, x, h=1e -10):
    n = x.shape[ -1]
    eye = np.eye(n)
    x_plus_dx = x + h * eye # n x n
    num_jac = (f(x_plus_dx) - f(x)) / h # limit definition of the formula # n x m
    if num_jac.ndim >= 2:
        num_jac = num_jac.swapaxes( -1, -2) # m x n
    return num_jac

# Compare our grad_f with numerical gradient
```

```
def check_numerical_jacobian(f, jac_f,  nD=2, **kwargs):
    x = np.random.rand(nD)
    num_jac = numerical_jacobian(f, x, **kwargs)
    return np.allclose(num_jac, jac_f(x), atol=1e -06, rtol=1e -4) # m x n

## Throw error if grad_f is wrong
assert check_numerical_jacobian(f, grad_f)

## Gradient of gradient
def hessian_f(x):
    """
    1. For an input x of shape x.shape = (2,)
        hessian_f(x) must return a n array of shape (2, 2)

    2. For an input x of shape x.shape = (m, 2),
        hessian_f(x) must return an array of shape (m, 2, 2)
        which contains hessian_f(x) is computed for m values of x

    3. For an input x of shape x.shape = (m, n, 2)
        hessian_f(x) must return an array of shape (m, n, 2, 2)
        which contains hessian_f(x) is computed for (m x n) values of x
    """
    coeff1 = np.array([2, 1])
    coeff2 = np.array([1, -2])
    coeff3 = np.array([ -1, 0])
    return (0.06 * np.exp(x @ coeff1)[..., None, None] * np.outer(coeff1, coeff1)
            + 0.05 * np.exp(x @ coeff2)[..., None, None] * np.outer(coeff2, coeff2)
            +  np.exp(x @ coeff3)[..., None, None] * np.outer(coeff3, coeff3))

## Throw error if hessian_f is wrong
assert check_numerical_jacobian(grad_f, hessian_f)


def taylor_series_quad_approx(x0, func, grad_func, hessian_func):
    def quad_func(x):
        x_min_x0 = (x -x0)[..., None] # make column vectors
        x_min_x0_T = (x -x0)[..., None, :] # make row vectors
        grad_f_x0_T = grad_func(x0)[..., None, :] # make row vectors
        return (func(x0)
                + grad_f_x0_T @ x_min_x0
                + x_min_x0_T @ hessian_func(x0) @ x_min_x0).squeeze(axis=( -1, -2))

    return quad_func

def plot_contours(func, ax=None, cmap='Blues_r', levels=20,
                  xrange=slice( -3,3,21j),
```

```
                  yrange=slice( -3,3,21j)):
    x, y = np.mgrid[xrange,
                    yrange]
    bfx = np.concatenate([x[..., None],
                          y[..., None]], axis= -1)
    f = func(bfx)
    if ax is None:
        fig, ax = plt.subplots()
    ctr = ax.contour(x, y, np.log(f), levels, cmap=cmap)
    ax.clabel(ctr, ctr.levels, inline=True, fontsize=6)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    return ax

# Fit a quadratic curve around this curve:
ax = plot_contours(f,
                   levels=[0.01, 0.5, 0.8, 1.0, 1.2, 1.6, 1.8, 2.0])
x0 = np.array([0, 2])
ax.plot([x0[0]], [x0[1]], 'ro')
ax.text(x0[0], x0[1], '$x_0$')
quad_func = taylor_series_quad_approx(x0, f, grad_f, hessian_f)
# x, y = np.mgrid[ -3:3:21j,
#                   -3:3:21j]
# bfx = np.concatenate([x[..., None],
#                       y[..., None]], axis= -1)
# print(bfx.shape)
# print(f(bfx).shape)
# print(grad_f(bfx).shape)
# print(quad_func(bfx).shape)
plot_contours(quad_func, ax=ax, cmap='Reds_r',
              levels=[0.1, 0.2, 0.5, 0.8, 1.0, 1.5],
              xrange=slice( -1,2,21j),
              yrange=slice( -2,3,21j))
ax.set_title("Taylor series quadratic fit in red to the blue curve")
plt.show()
```



2.3.1 Example Taylor series 1

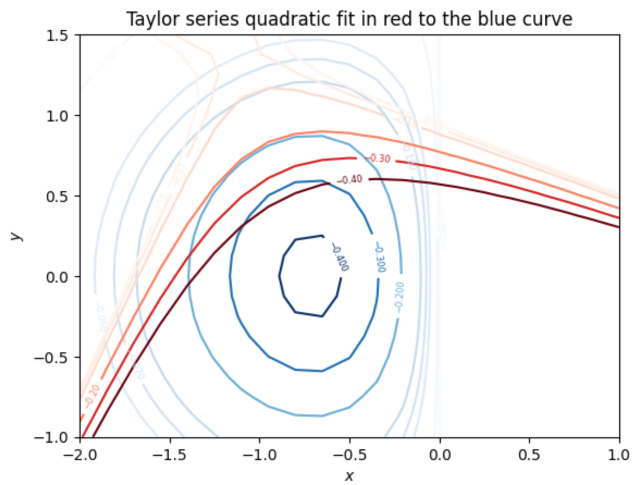
Following the example above, fit a quadratic function to the function using Taylor series expansion near the points $\mathbf{x}_0 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ then visualize the contour plots of the original function and the quadratic function (50 marks),

$$f(\mathbf{x}) = x_1 \exp(-(x_1^2 + x_2^2)) \tag{33}$$

```
def f(x):
    return (x @ [1, 0]) * np.exp( -(x * x).sum(axis= -1))

def grad_f(x):
    coeff = np.array([1, 0])
    return np.exp( -(x * x).sum(axis= -1, keepdims=True)) * coeff - 2 * f(x)[..., None] * x

## Throw error if grad_f is wrong
assert check_numerical_jacobian(f, grad_f)

# def grad_f1(x):
#     coeff = np.array([1, 0])
#     return np.exp( -(x * x).sum(axis= -1, keepdims=True)) * coeff

# def hessian_f1(x):
#     coeff = np.array([1, 0])
#     return - 2 * np.exp( -(x * x).sum(axis= -1))[..., None, None] * (coeff[:, None] @ x[..

# ## Throw error if grad_f is wrong
# assert check_numerical_jacobian(grad_f1, hessian_f1)
```

```
# def grad_f2(x):
#     return - 2 * f(x)[..., None] * x

# def hessian_f2(x):
#     ones = np.ones_like(x)
#     return ( - 2 * f(x)[..., None, None] * np.eye(x.shape[ -1])
#             - 2 * x[..., None] @ grad_f(x)[..., None, :])
# assert check_numerical_jacobian(grad_f2, hessian_f2)
def hessian_f(x):
    coeff = np.array([1, 0])
    ones = np.ones_like(x)
    return ( - 2 * np.exp( -(x * x).sum(axis= -1))[..., None, None] * (coeff[:, None] @ x[..
            - 2 * f(x)[..., None, None] * np.eye(x.shape[ -1])
            - 2 * x[..., None] @ grad_f(x)[..., None, :])

assert check_numerical_jacobian(grad_f, hessian_f)

x, y = np.mgrid[ -3:3:21j, -3:3:21j]
bfx = np.concatenate((x[..., None], y[..., None]), axis= -1)
f(bfx).shape, grad_f(bfx).shape, hessian_f(bfx).shape


def plot_contours(func, ax=None, cmap='Blues_r', levels=20,
                  xrange=slice( -3,3,21j),
                  yrange=slice( -3,3,21j)):
    x, y = np.mgrid[xrange,
                    yrange]
    bfx = np.concatenate([x[..., None],
                          y[..., None]], axis= -1)
    f = func(bfx)
    if ax is None:
        fig, ax = plt.subplots()
    ctr = ax.contour(x, y, f, levels, cmap=cmap)
    ax.clabel(ctr, ctr.levels, inline=True, fontsize=6)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    return ax

# Fit a quadratic curve around this curve:
ax = plot_contours(f,
                   levels=[ -4e -1, -3e -1, -2e -1, -1e -1, -7e -2, -5e -2, -1e -2, -1e -3,
                   xrange=slice( -2,1,21j),
                   yrange=slice( -1,1.5,21j))
x0 = np.array([ -1, 1])
quad_func = taylor_series_quad_approx(x0, f, grad_f, hessian_f)
```

```
plot_contours(quad_func, ax=ax, cmap='Reds_r',
              levels=[ -0.4, -.3, -.2, -.1, -0.07, -0.05],
              xrange=slice( -2,1,21j),
              yrange=slice( -1,1.5,21j))
ax.set_title("Taylor series quadratic fit in red to the blue curve")
plt.show()
```



3 Minimization by gradient descent

3.0.1 Example 1 : minimization by gradient descent

Find the minimizer of $f(\mathbf{x}) = 0.06\exp(2x_1+x_2)+0.05\exp(x_1-2x_2)+\exp(-x_1)$.
In vector form we can write it as:

$$f(\mathbf{x}) = 0.06\exp([2,1]\mathbf{x}) + 0.05\exp([1,-2]\mathbf{x}) + \exp([-1,0]\mathbf{x}) \tag{34}$$

$$\nabla_{\mathbf{x}}^{\top} f(\mathbf{x}) = 0.06\exp([2,1]\mathbf{x})[2,1]+0.05\exp([1,-2]\mathbf{x})[1,-2]+\exp([-1,0]\mathbf{x})[-1,0] \tag{35}$$

Algorithm 9.3 Gradient descent method.
given a starting point $\mathbf{x} \in \mathrm{dom} f$.
repeat

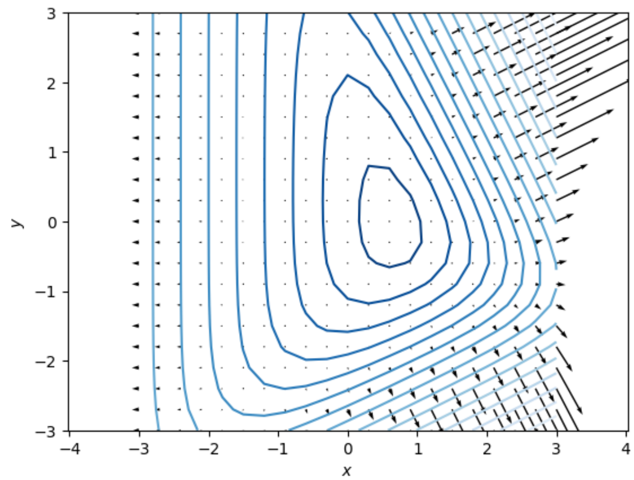
1. Choose step size α_t
2. Update. $\mathbf{x} := \mathbf{x} - \alpha_t \nabla f(\mathbf{x})$

until stopping criterion is satisfied.

```
# Define the function
def f(x):
    return (0.06 * np.exp(x @ [2, 1])
            + 0.05* np.exp(x @ [1, -2])
            + np.exp(x @ [ -1, 0]))

# Compute its derivative, the gradient function
def grad_f(x):
    coeff1 = np.array([2, 1])
    coeff2 = np.array([1, -2])
    coeff3 = np.array([ -1, 0])
    # Slicing using None, increases the dimension by 1.
    #
    return (0.06 * np.exp(x @ coeff1)[..., None] * coeff1
            + 0.05 * np.exp(x @ coeff2)[..., None] * coeff2
            + np.exp(x @ coeff3)[..., None] * coeff3)


assert check_numerical_jacobian(f, grad_f)
f(np.zeros((2,))), grad_f(np.zeros((2,)))
```

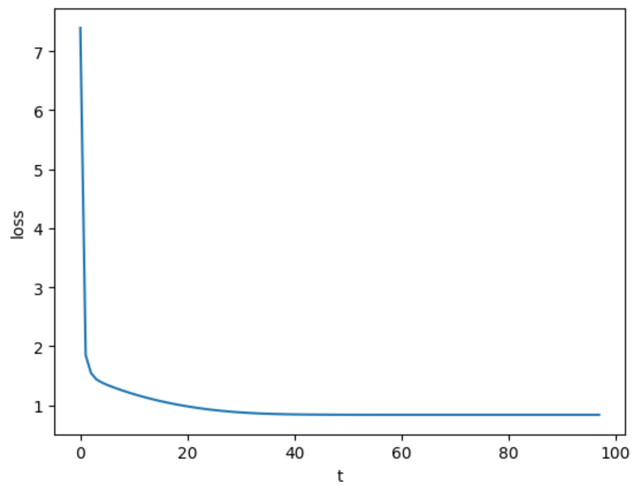
```
(np.float64(1.11), array([ -0.83, -0.04]))
```

```
def plot_gradients(func, gradfunc):
    x, y = np.mgrid[ -3:3:21j,
                    -3:3:21j]
    bfx = np.concatenate((x[..., None], y[..., None]), axis= -1)
    f = func(bfx)
    dfdx = gradfunc(bfx)
    fig, ax = plt.subplots()
    ctr = ax.contour(x, y, np.log(f), 20, cmap='Blues_r')
    ax.quiver(bfx[..., 0], bfx[..., 1], dfdx[..., 0], dfdx[..., 1])
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.axis('equal')
    plt.show()

plot_gradients(f, grad_f)
```



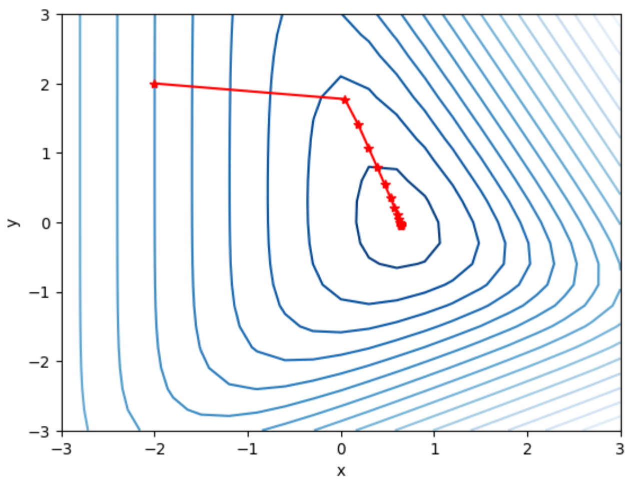

```
# Implement the gradient descent algorithm
def minimize(x0, f, grad_func, alpha_t=0.2, maxiter=100):
    t = 0
    xt = x0
    grad_f_t = grad_func(xt)
    list_of_xts, list_of_fs = [xt], [f(xt)] # for logging
    while np.linalg.norm(grad_f_t) > 1e -4: # < - - Check for convergence
        xt = xt - alpha_t * grad_f_t # < - - Main update step
        grad_f_t = grad_func(xt) # Compute the next gradient

        if t >= maxiter:  # Failsafe, if the algorithm does not converge
            break
        else:
            t += 1
        list_of_xts.append(xt) # for logging
        list_of_fs.append(f(xt)) # for logging

    return xt, list_of_xts, list_of_fs

x0 = np.array([ -2, 2])
#x0 = np.random.rand(2,2) * 4 - 2
OPTIMAL_X, list_of_xts, list_of_fs = minimize(x0,
                                              f, grad_f,
                                              alpha_t=0.2,
                                              maxiter=1000)

fig, ax = plt.subplots()
ax.plot(list_of_fs)
ax.set_xlabel('t')
ax.set_ylabel('loss')
```

```
plt.show()
```

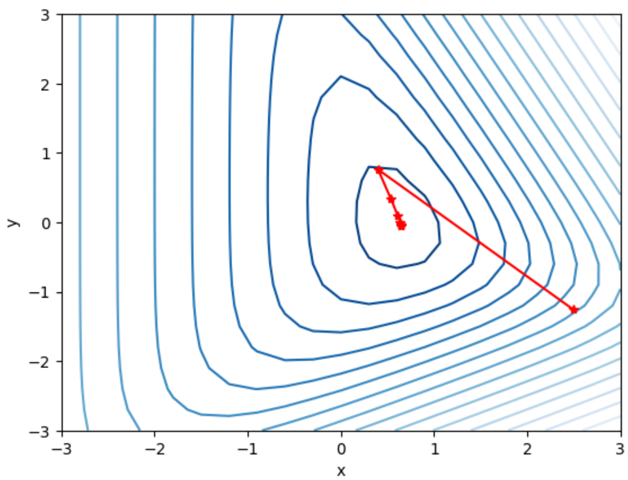



```
from matplotlib import animation, rc
rc('animation', html='jshtml')

class Anim:
    def __init__(self, fig, ax, func):
        self.fig = fig
        self.ax = ax
        x, y = np.mgrid[ -3:3:21j,
                        -3:3:21j]
        bfx = np.concatenate((x[..., None], y[..., None]), axis= -1)
        f = func(bfx)
        self.ctr = self.ax.contour(x, y, np.log(f), 20, cmap='Blues_r')
        self.ax.set_xlabel('x')
        self.ax.set_ylabel('y')
        #self.ax.clabel(self.ctr, self.ctr.levels, inline=True, fontsize=6)
        self.list_of_xs = []
        self.list_of_ys = []
        self.line2, = self.ax.plot([], [], 'r* -')


    def anim_init(self):
        return (self.line2,)

    def update(self, xt):
        self.list_of_xs.append(xt[0])
        self.list_of_ys.append(xt[1])
        self.line2.set_data(self.list_of_xs, self.list_of_ys)
        return self.line2,
```

```
fig, ax = plt.subplots()
a = Anim(fig, ax, f)
animation.FuncAnimation(fig, a.update, frames=list_of_xts[::5],
                        init_func=a.anim_init, blit=True, repeat=False)
```




```
# The learning rate can be sensitive to the starting points.
# Observe the behavior for different starting points
# For different starting points you might need different learning rate scheme

x0 = np.random.rand(2) * 6 - 3
OPTIMAL_X, list_of_xts, list_of_fs = minimize(x0,
                                              f, grad_f,
                                              alpha_t=0.2,
                                              maxiter=200)
fig, ax = plt.subplots()
a = Anim(fig, ax, f)
animation.FuncAnimation(fig, a.update, frames=list_of_xts[::10],
                        init_func=a.anim_init, blit=True, repeat=False)
```



3.0.2 Homework (ContinuousOptimization): Problem 1

(20 marks)

Following the example above, *implement your own* gradient descent algorithm that minimizes the following function (50 marks),

$$f(\mathbf{x}) = x_1 \exp(-(x_1^2 + x_2^2)) \tag{36}$$

Test your algorithm with the starting points of $\mathbf{x}_0 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ and $\mathbf{x}_0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ and learning rate of $\alpha_t = 0.25$.

```
x0 = np.array([ -1, 1])


def f(x):
    # YOUR CODE HERE
    raise NotImplementedError()

def grad_f(x):
    # YOUR CODE HERE
    raise NotImplementedError()


## Throw error if grad_f is wrong
assert check_numerical_jacobian(f, grad_f)

# Implement the gradient descent algorithm
def minimize(x0, f, grad_func, alpha_t=0.2, maxiter=100):
    t = 0
```

```
    xt = x0
    grad_f_t = grad_func(xt)
    list_of_xts, list_of_fs = [xt], [f(xt)] # for logging
    while np.linalg.norm(grad_f_t) > 1e -4: # < - - Check for convergence
        # 1. Update xt using gradient descent update
        # 2. Compute grad_f_t with new xt
        # YOUR CODE HERE
        raise NotImplementedError()
        if t >= maxiter:  # Failsafe, if the algorithm does not converge
            break
        else:
            t += 1
        list_of_xts.append(xt) # for logging
        list_of_fs.append(f(xt)) # for logging

    return xt, list_of_xts, list_of_fs


OPTIMAL_X, list_of_xts, list_of_fs = minimize(x0,
                                              f, grad_f,
                                              alpha_t=0.25,
                                              maxiter=200)

class Anim:
    def __init__(self, fig, ax, func):
        self.fig = fig
        self.ax = ax
        x, y = np.mgrid[ -2:1:21j,
                         -1:1:21j]
        bfx = np.concatenate((x[..., None], y[..., None]), axis= -1)
        f = func(bfx)
        self.ctr = self.ax.contour(x, y, f,
                                   levels=[ -4e -1, -3e -1, -2e -1, -1e -1, -7e -2, -5e -2,
                                   cmap='Blues_r')
        self.ax.set_xlabel('x')
        self.ax.set_ylabel('y')
        #self.ax.clabel(self.ctr, self.ctr.levels, inline=True, fontsize=6)
        self.list_of_xs = []
        self.list_of_ys = []
        self.line2, = self.ax.plot([], [], 'r* -')


    def anim_init(self):
        return (self.line2,)

    def update(self, xt):
```

```
        self.list_of_xs.append(xt[0])
        self.list_of_ys.append(xt[1])
        self.line2.set_data(self.list_of_xs, self.list_of_ys)
        return self.line2,


fig, ax = plt.subplots()
a = Anim(fig, ax, f)
anim1 = animation.FuncAnimation(fig, a.update, frames=list_of_xts[::10],
                                init_func=a.anim_init, blit=True, repeat=False)
anim1
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
NotImplementedError                       Traceback (most recent call last)
Cell In[13], line 14
     10     raise NotImplementedError()
     13 ## Throw error if grad_f is wrong
 - - -> 14 assert check_numerical_jacobian(f, grad_f)
     16 # Implement the gradient descent algorithm
     17 def minimize(x0, f, grad_func, alpha_t=0.2, maxiter=100):

Cell In[4], line 55, in check_numerical_jacobian(f, jac_f, nD, **kwargs)
     53 def check_numerical_jacobian(f, jac_f,  nD=2, **kwargs):
     54     x = np.random.rand(nD)
 - - -> 55     num_jac = numerical_jacobian(f, x, **kwargs)
     56     return np.allclose(num_jac, jac_f(x), atol=1e -06, rtol=1e -4)

Cell In[4], line 47, in numerical_jacobian(f, x, h)
     45 eye = np.eye(n)
     46 x_plus_dx = x + h * eye # n x n
 - - -> 47 num_jac = (f(x_plus_dx) - f(x)) / h # limit definition of the formula # n x m
     48 if num_jac.ndim >= 2:
     49     num_jac = num_jac.swapaxes( -1, -2) # m x n

Cell In[13], line 6, in f(x)
      4 def f(x):
      5     # YOUR CODE HERE
 - - - -> 6     raise NotImplementedError()

NotImplementedError:
```

```
assert np.allclose(OPTIMAL_X, [ -7.07e -01,  1.06e -04], atol=1e -3, rtol=1e -2)
```

```
x0 = np.array([ -1, -1])
# Repeat minmimization with a different starting point x0
# YOUR CODE HERE
raise NotImplementedError()
```

```
assert np.allclose(OPTIMAL_X, [ -7.07e -01,  1.06e -04], atol=1e -3, rtol=1e -2)
```