EENG582: Artificial Neural Networks

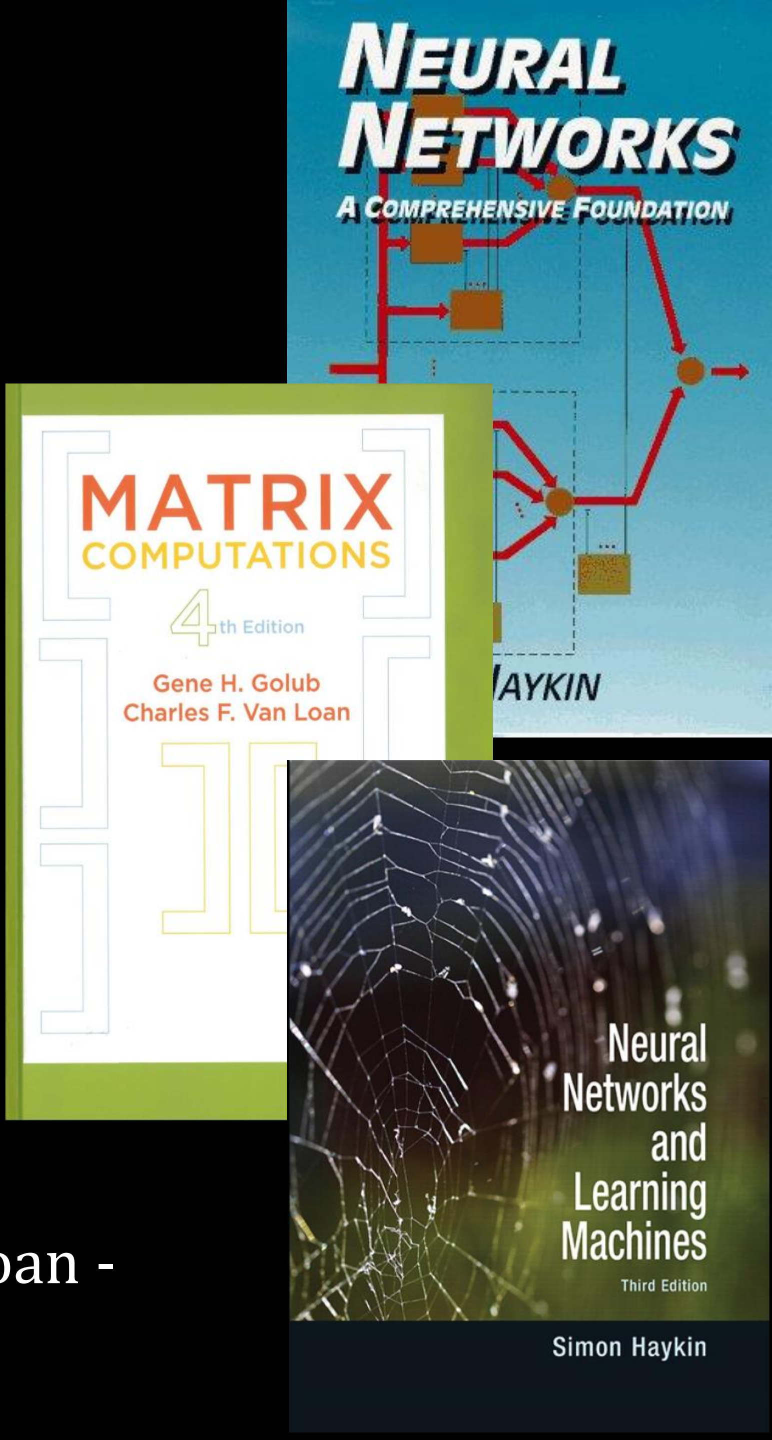
Reference 1: **Neural Networks A Comprehensive Foundation by S. Haykin**

Sections 3.3 – Unconstraint Optimization Techniques

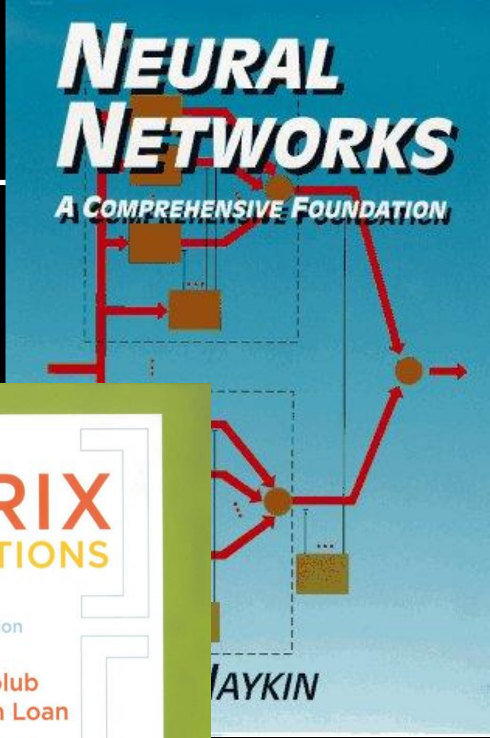
3. Single Layer Perceptrons + Optimization

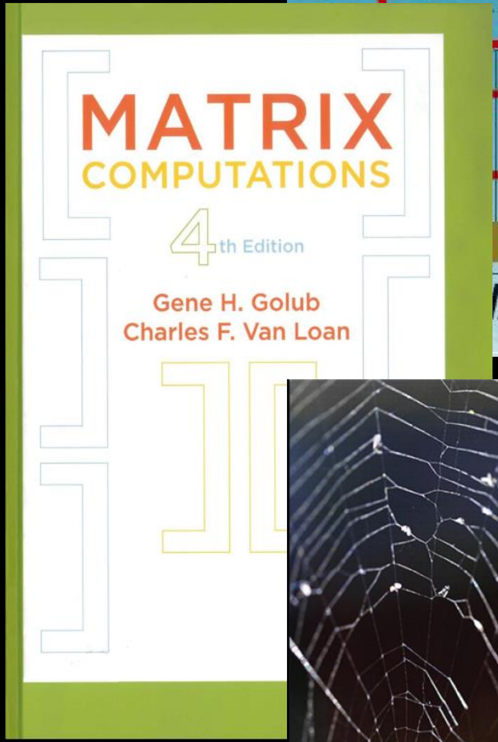
Prof. Dr. Hasan AMCA
Electrical and Electronic Engineering Department (ee.emu.edu.tr)
Eastern Mediterranean University (emu.edu.tr)

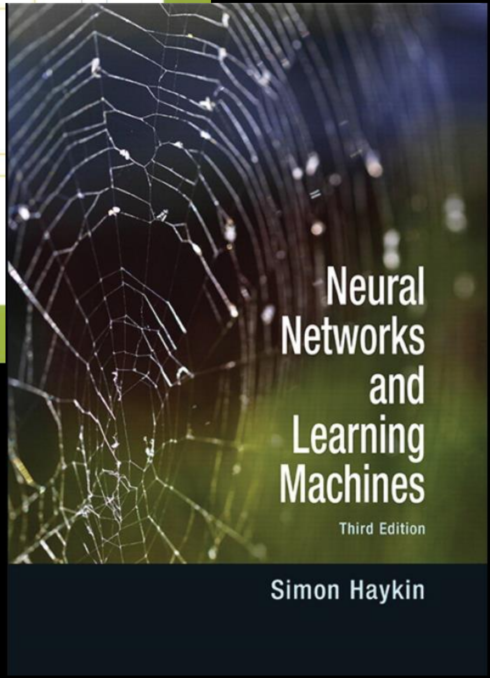
Reference 2: Matrix Computations, 4th Ed. By Golub and Van Loan - Chapter 11, Large Sparse Linear System Problems



3. Single Layer Perceptrons







3.3 Unconstrained Optimization Techniques

- Consider a continuously differentiable cost function $\mathcal{E}(\mathbf{w})$ that is a function of unknown weight vector $\mathbf{w}$ which maps elements of $\mathbf{w}$ into real numbers.
- To choose the optimal weight (parameter) vector $\mathbf{w}$ of an adaptive filtering algorithm which satisfy the condition

$$\mathcal{E}(\mathbf{w}^*) \leq \mathcal{E}(\mathbf{w}) \tag{3.5}$$

- we need to solve an *unconstrained optimization problem* which minimizes the cost function $\mathcal{E}(\mathbf{w})$ with respect to the weight vector $\mathbf{w}$.
- The necessary condition for optimality is

$$\nabla\mathcal{E}(\mathbf{w}^*) = \mathbf{0} \tag{3.7}$$

- Where ∇ is the gradient operator

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_m}\right]^T \tag{3.8}$$

3.3 Unconstrained Optimization Techniques

- and $\nabla\mathcal{E}(\mathbf{w})$ is the gradient vector of the cost function

$$\nabla\mathcal{E}(\mathbf{w}) = \left[\frac{\partial\mathcal{E}}{\partial w_1}, \frac{\partial\mathcal{E}}{\partial w_2}, \dots, \frac{\partial\mathcal{E}}{\partial w_m}\right]^T \tag{3.9}$$

- A class of *unconstrained optimization* algorithms that is particularly well suited for the design of adaptive filters is based on *local iterative descent*:

  Starting with an initial guess denoted by $\mathbf{w}(0)$, generate a sequence of weight vectors $\mathbf{w}(1)$, $\mathbf{w}(2)$, ... , such that the cost function $\mathcal{E}(\mathbf{w})$ is reduced at each iteration of the algorithm, as shown by,

$$\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n)) \tag{3.10}$$

  where $\mathbf{w}(n)$ is the old value of the weight vector and $\mathbf{w}(n + 1)$ is its updated value. The algorithm eventually converge onto the optimal solution $\mathbf{w}^*$.

- In the next section we describe three unconstrained optimization methods that rely on the idea of iterative descent.

Steepest Descent ∈ Gradient Descent

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

$$\text{coefficient}(\mathbf{w}) = 0.0$$

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

$$cost(\mathcal{E}) = f(\mathbf{w})$$

or

$$\mathcal{E} = \text{evaluate}(f(\mathbf{w}))$$

The derivative (of the function at a given point) of the cost is calculated. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

$$\delta = derivative(cost) = \partial\boldsymbol{\mathcal{E}}/\partial\mathbf{w}$$

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A learning rate parameter (α) must be specified that controls how much the coefficients can change on each update.

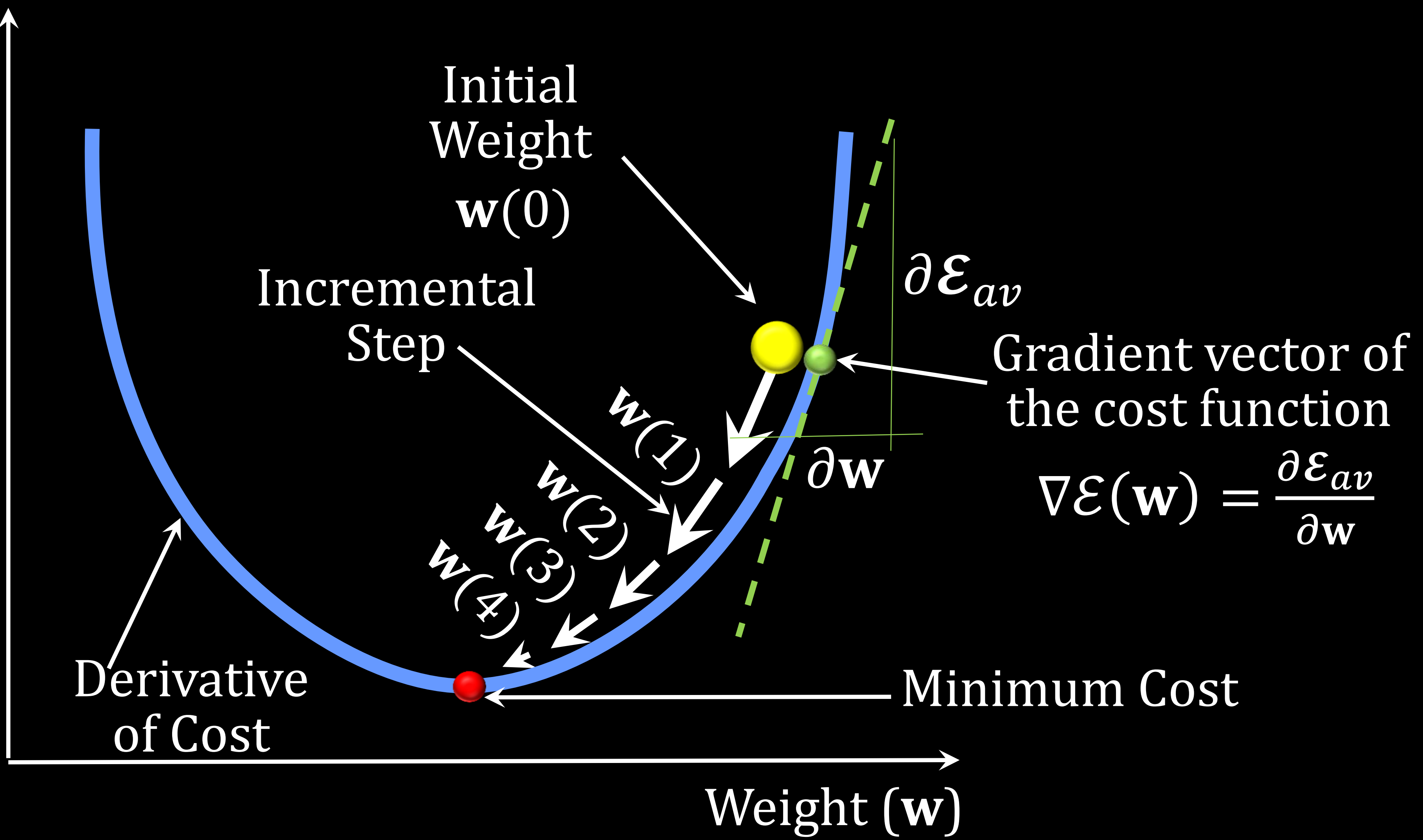
$$\mathbf{w} = \mathbf{w} - (\alpha \times \delta)$$

This process is repeated until cost of the coefficients ($\mathcal{E}$) is 0.0 or close enough to zero.

https://machinelearningmastery.com/gradient-descent-for-machine-learning/
https://realpython.com/gradient-descent-algorithm-python/

Steepest Descent

$$\underbrace{\mathcal{E}(\mathbf{w}(n+1))}_{\text{New (updated) value of } \mathbf{w}(n)} < \underbrace{\mathcal{E}(\mathbf{w}(n))}_{\text{Old value of } \mathbf{w}(n)}$$



Steepest Descent ∈ Gradient Descent

- **Gradient Descent** refers to any of a class of algorithms that calculate the gradient of the objective function, then move "downhill" in the indicated direction; the step length can be fixed, estimated (e.g., via line search), or ... (see this link for some examples).

- **Steepest Descent** is a special case of Gradient Descent where the step length is chosen to minimize the objective function value.

Algorithm 2: Gradient Descent		
	input	: $f:\mathbb{R}^n \to \mathbb{R}$ a differentiable function $\mathbf{x}^{(0)}$ an initial solution
	output	: $\mathbf{x}^*$ a local minimum of the cost function f.
1	**begin**	
2 3 4 5 6	$k \leftarrow 0$; **while**	STOP-CRIT and $(k < k_{\max})$ **do** $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\nabla f(\mathbf{x})$; With $\alpha^{(k)} = \arg\min_{\alpha\in\mathbb{R}_+} f\left(\mathbf{x}^{(k)} - \alpha\nabla f(\mathbf{x})\right)$; $k \leftarrow k+1$;
7	return	$\mathbf{x}^{(k)}$
8	**end**	

Steepest Descent

- In the method of steepest descent, the successive adjustments applied to the weight vector $\mathbf{w}$ are in the direction of steepest descent, that is, in a direction opposite to the gradient vector $\nabla\mathcal{E}(\mathbf{w})$.
- For convenience of presentation, we write.

$$\mathbf{g} = \nabla\mathcal{E}(\mathbf{w}) \tag{3.11}$$

- The steepest descent algorithm is described by

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\mathbf{g}(n) \tag{3.12}$$

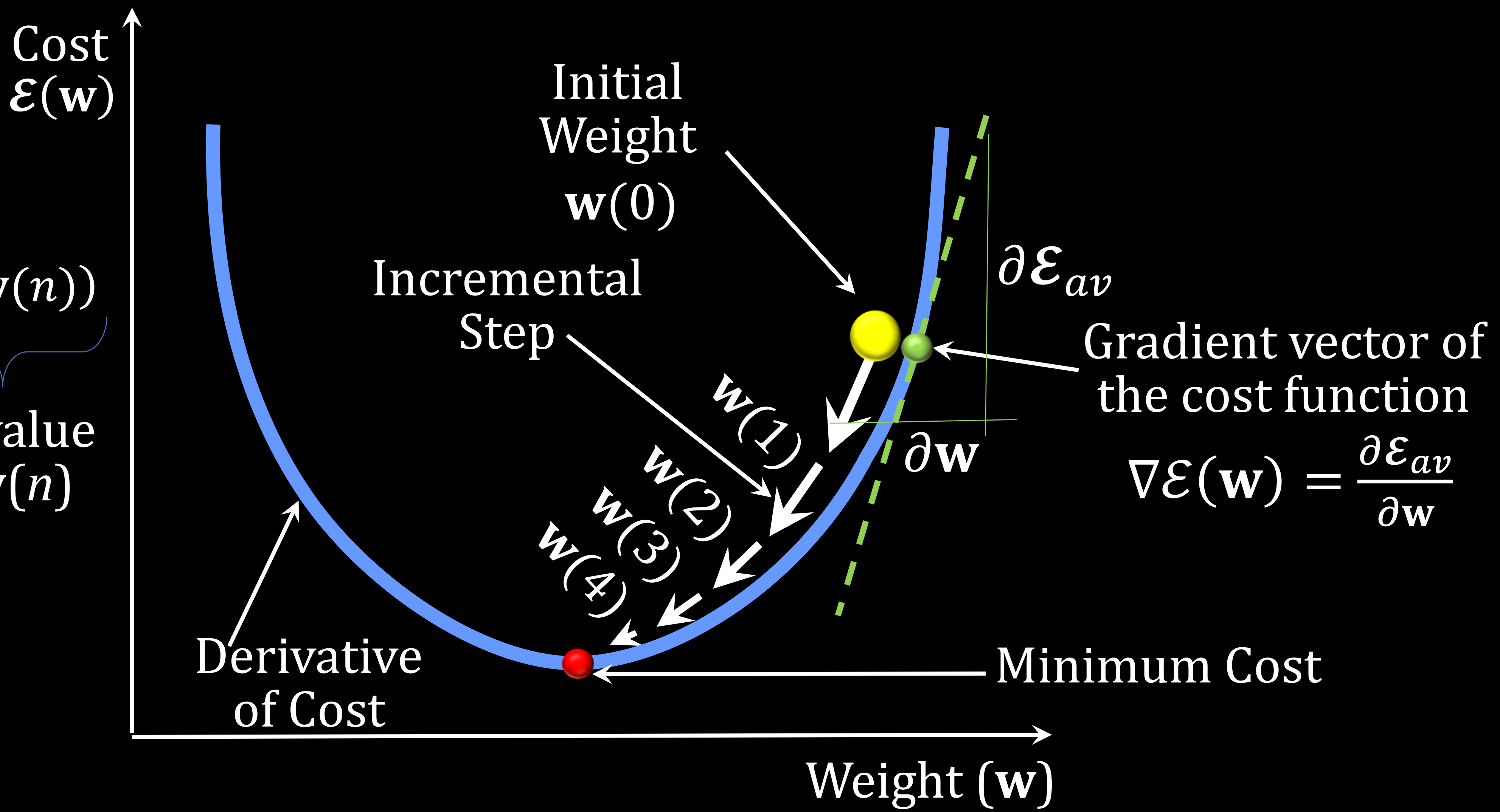
- where η is a positive constant called the *step-size* or *learning-rate parameter*, and $\mathbf{g}(n)$ is the gradient vector evaluated at the point $\mathbf{w}(n)$.
- In going from iteration *n* to *n*+1 the algorithm applies the correction

$$\Delta\mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) = -\eta\mathbf{g}(n) \tag{3.13}$$

- Which is the *error-correction rule*.

Steepest Descent

$$\underbrace{\mathcal{E}(\mathbf{w}(n+1))}_{\text{New (updated) value of } \mathbf{w}(n)} < \underbrace{\mathcal{E}(\mathbf{w}(n))}_{\text{Old value of } \mathbf{w}(n)}$$



Steepest Descent

- To show that the formulation of the steepest descent algorithm satisfies the condition of (3.10) for iterative descent, we use a first-order Taylor series expansion around $\mathbf{w}(n)$ to approximate $\mathcal{C}(\mathbf{w}(n+1))$ as

$$\mathcal{E}(\mathbf{w}(n+1)) \cong \mathcal{E}\big(\mathbf{w}(n)\big) + \mathbf{g}^T(n)\Delta\mathbf{w}(\mathrm{n}) = \mathcal{E}\big(\mathbf{w}(n)\big) + \eta\mathbf{g}^T(n)\mathbf{g}(\mathrm{n})$$

$$= \mathcal{E}\big(\mathbf{w}(n)\big) + \eta\,\|\mathbf{g}(\mathrm{n})\|$$

- for a positive learning-rate parameter η, the cost function is decreased as the algorithm progresses from one iteration to the next.
- Watch YouTube for Unconstraint Optimization-Steepest Descent Algorithm
  - Steepest Descent Algorithm https://www.youtube.com/watch?v=BBlDWNTimoA&t=596s
  - Applied Optimization - Steepest Descent with Matlab – YouTube

    https://www.youtube.com/watch?v=pt7PBjTLzzc

Steepest Descent

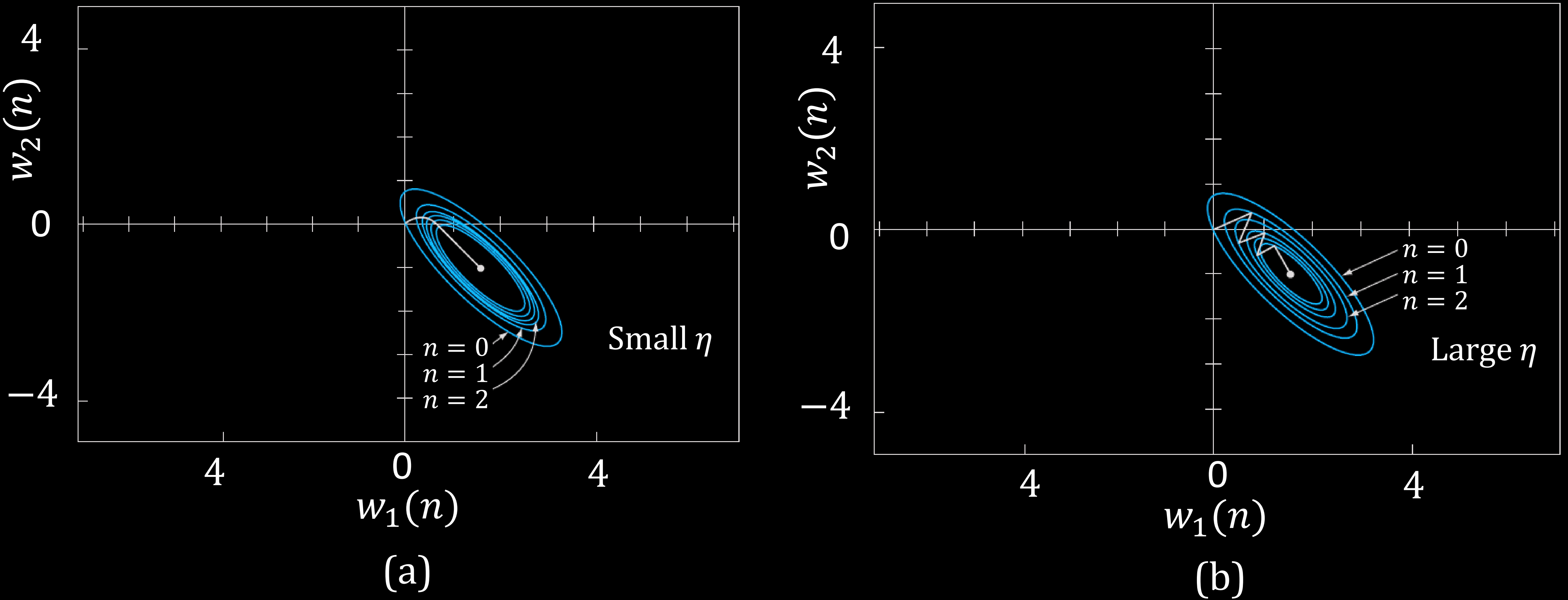
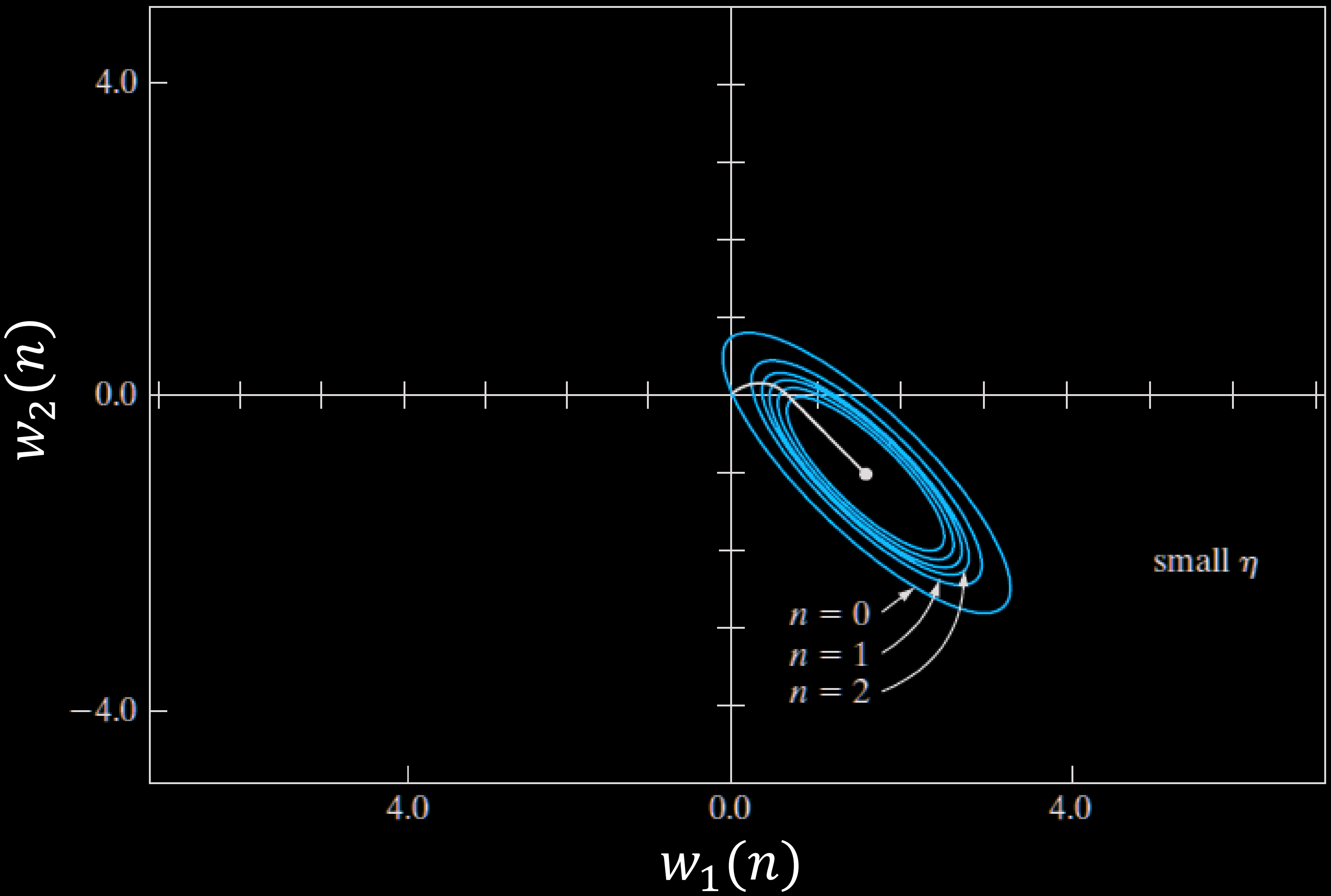
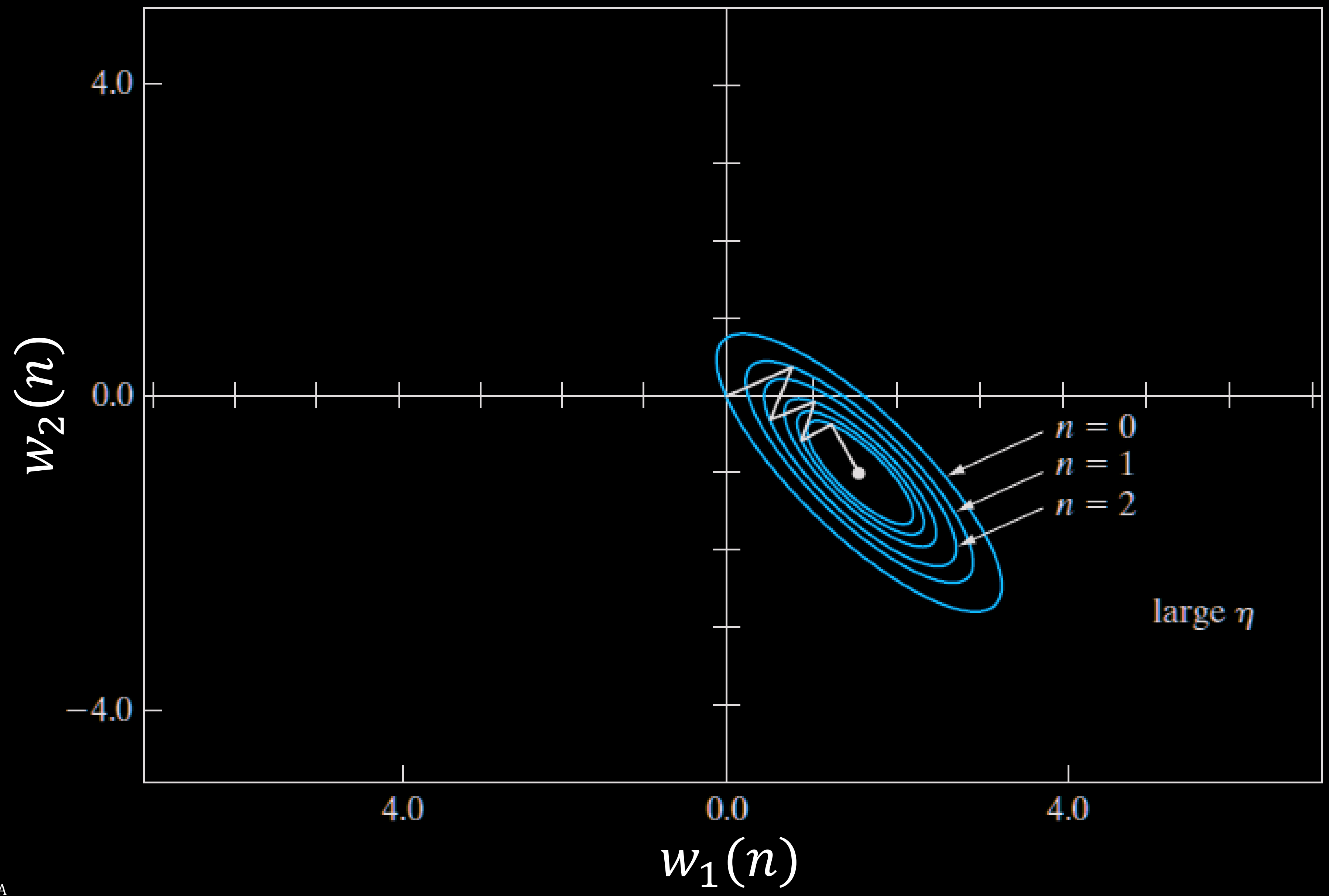


Fig 3.2 Trajectory of the method of steepest descent in a two-dimensional space for two different *values* of learning-rate parameter: (a) $\eta = 0.3$, (b) $\eta = 1.0$.
The coordinates w_1 and w_2 are elements of the weight vector $\mathbf{w}$.

Steepest Descent



Steepest Descent



Steepest Descent

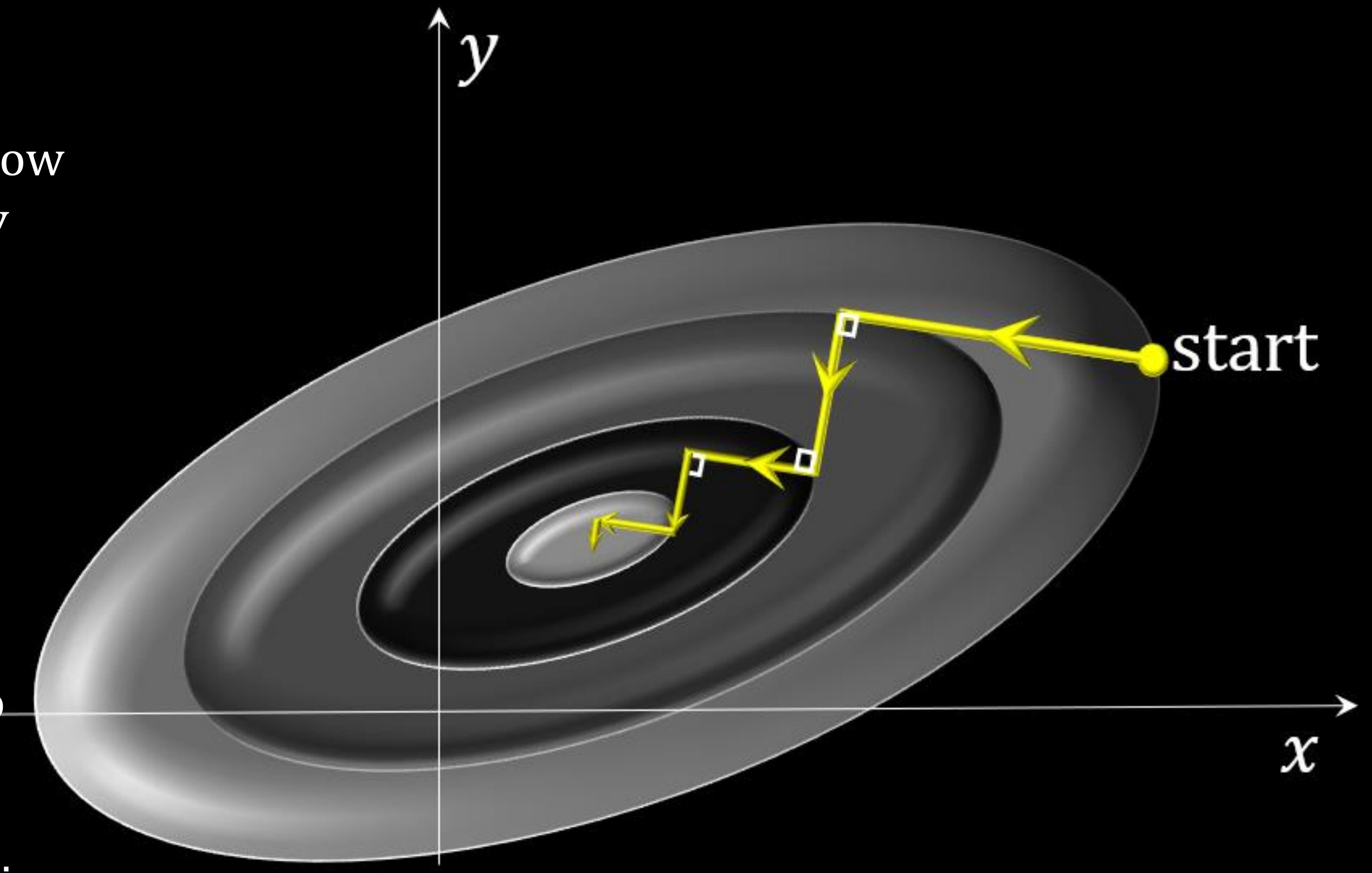
A simple Gradient Descent Algorithm is as follows:

1. Obtain a function to minimize $F(x)$
2. Initialize a value x from which to start the descent or optimization from
3. Specify a learning rate that will determine how much of a step to descend by or how quickly you converge to the minimum value
4. Obtain the derivative of that value x (the descent)
5. Proceed to descend by the derivative of that value multiplied by the learning rate
6. Update the value of x with the new value descended to
7. Check your stop condition to see whether to stop
8. If condition satisfied, stop. If not, proceed to step 4 with the new x value and keep repeating algorithm.

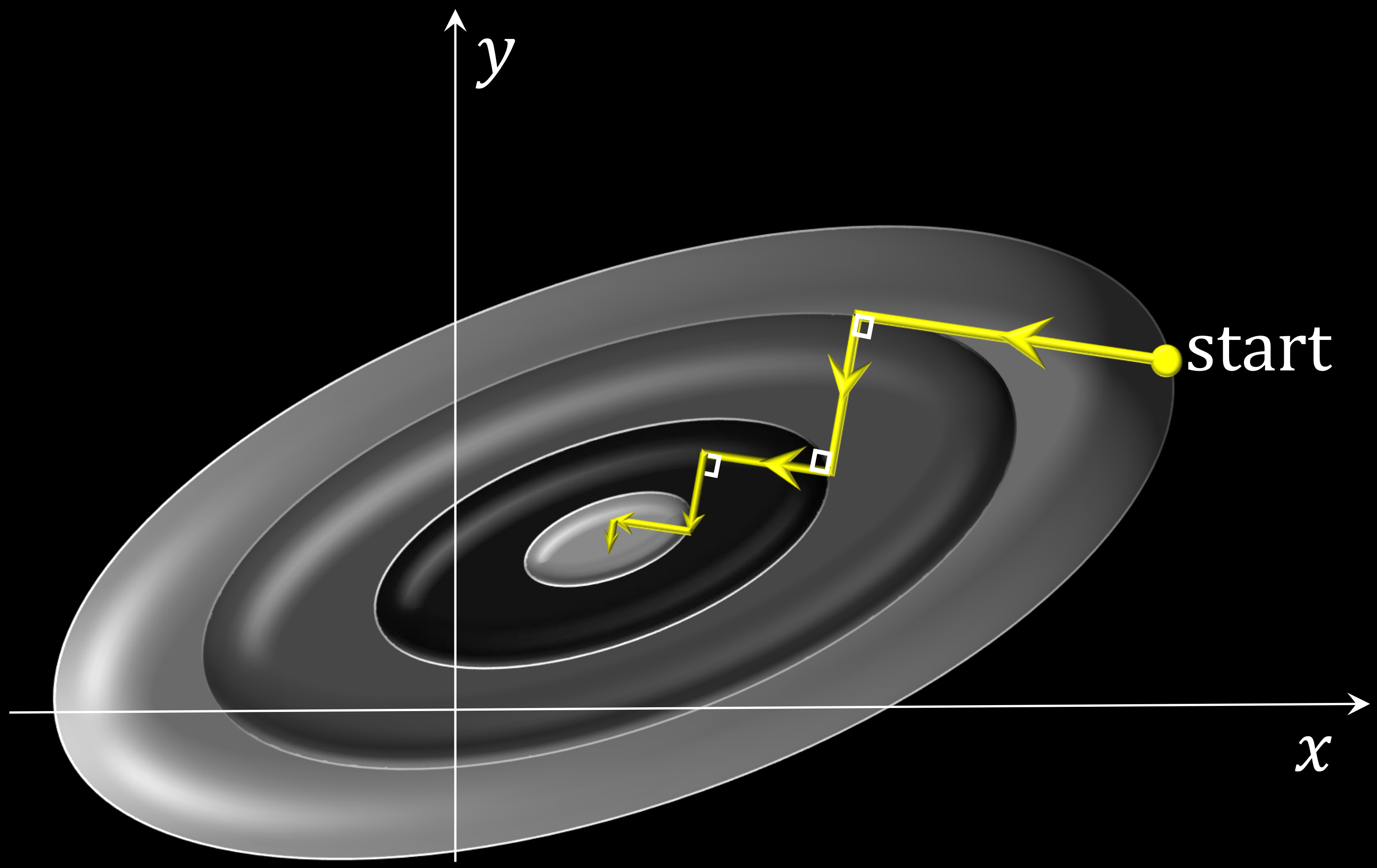
Steepest Descent

A simple gradient Descent Algorithm

1. Obtain a function to minimize $F(x)$
2. Initialize a value x from which to start the descent or optimization from
3. Specify a learning rate that will determine how much of a step to descend by or how quickly you converge to the minimum value
4. Obtain the derivative of that value x (the descent)
5. Proceed to descend by the derivative of that value multiplied by the learning rate
6. Update the value of x with the new value descended to
7. Check your stop condition to see whether to stop
8. If condition satisfied, stop. If not, proceed to step 4 with the new x value and keep repeating algorithm.



Steepest Descent

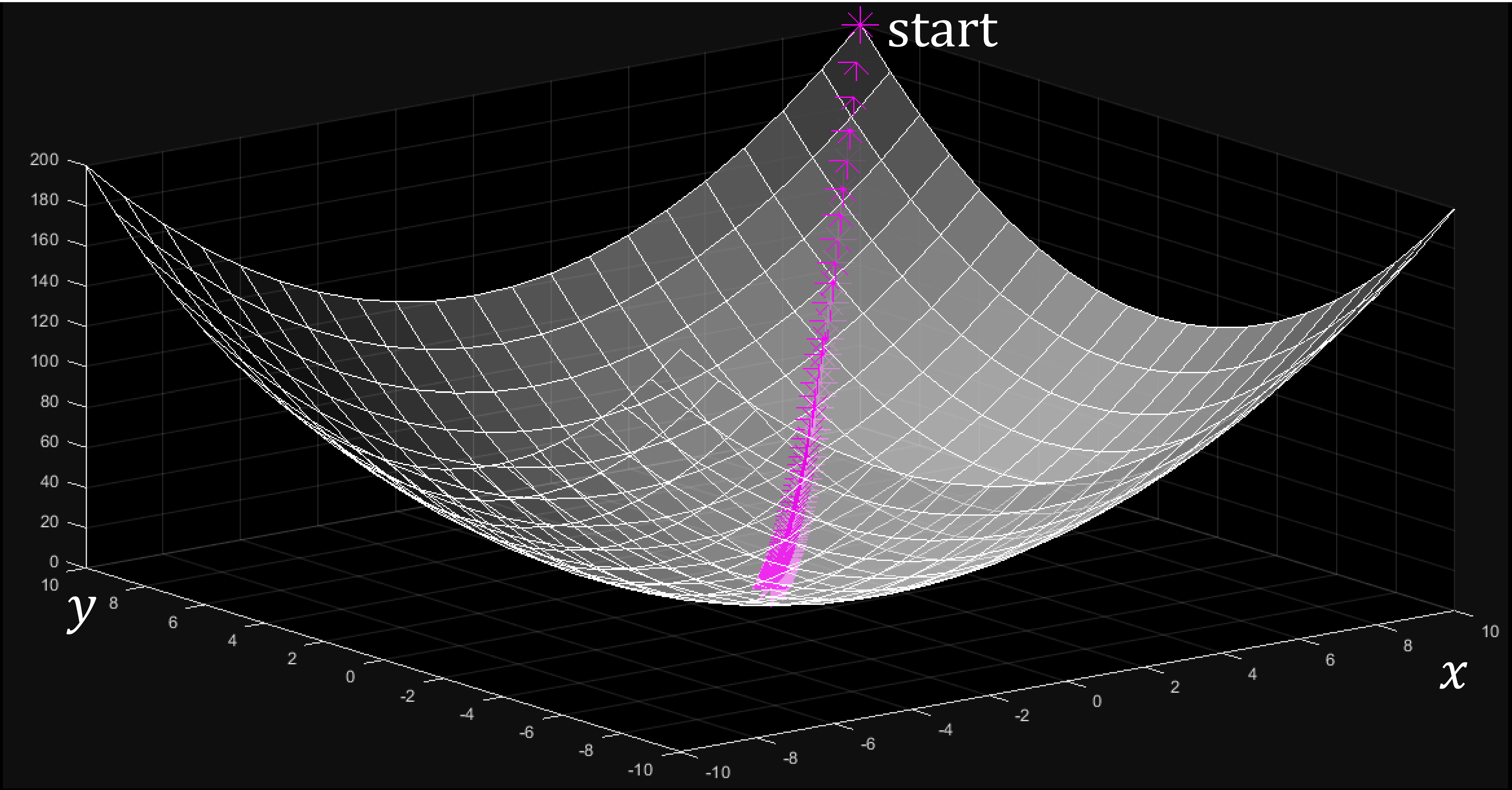


Steepest Descent

```
%Steepest Descent Contour Plot-MATLAB
%https://www.mathworks.com/matlabcentr
%al/fileexchange/35389-gradient-
%descent-visualization
clear all; close all; clc;
x=-10:1:10;
y=-10:1:10;
alpha=0.02;

[X Y] = meshgrid(x,y);
Z = X.^2 + Y.^2;

surf(X,Y,Z,'FaceAlpha','flat','AlphaDa
taMapping','scaled','AlphaData',gradie
nt(Z))
hold on;
colormap(gray);
meshgrid off
```

```
x0 = zeros(1000,2);
%x0(1,:) = randint(1,2,10);
x0(1,:) = [10 10];
x0(1,:)
plot3(x0(1,1),x0(1,2),(x0(1,1).^2 +
x0(1,2).^2),'m*','MarkerSize',20);

i=2;
while(1)
    pause
    % Gradient descent equation..
    x0(i,:) = x0(i-1,:) -
alpha.*2.*(x0(i-1,:));
    x0(i,:)
    plot3(x0(i,1),x0(i,2),(x0(i,1).^2
+ x0(i,2).^2),'m*','MarkerSize',20)
    i=i+1;
end
```

Steepest Descent



Newton's Method

- The basic idea of Newton's method is to minimize the quadratic approximation of the cost function $\mathcal{E}(\mathbf{w})$ around the current point $\mathbf{w}(n)$;
- Using Taylor series expansion of cost function around the point $\mathbf{w}(n)$, we have

$$\Delta\mathcal{E}\big(\mathbf{w}(n)\big) = \mathcal{E}\big(\mathbf{w}(n+1)\big) - \mathcal{E}\big(\mathbf{w}(n)\big) \cong \mathbf{g}^{T}(n)\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^{\mathrm{T}}(n)H(n)\Delta\mathbf{w}(n) \tag{3.14}$$

- Here, $\mathbf{g}(n)$ is the m-by-1 gradient vector of the cost function $\mathcal{E}(\mathbf{w})$ evaluated at the point $\mathbf{w}(n)$.

Algorithm for Newton's Method

Algorithm 3: Newton’s Method

input : $f: \mathbb{R}^n \rightarrow \mathbb{R}$ a twice differentiable function
$\mathbf{x}^{(0)}$ an initial solution

output : $\mathbf{x}^*$, a local minimum of the cost function f.

begin
$k \leftarrow 0$;
while STOP-CRIT and $(k < k_{\max})$ **do**
$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \delta^{(k)}$;
With $\delta^{(k)} = -\mathbf{H}_f\left(\mathbf{x}^{(k)}\right)^{-1} \nabla f\left(\mathbf{x}^{(k)}\right)$;
$k \leftarrow k + 1$;
return $\mathbf{x}^{(k)}$
end

Newton's Method

Algorithm 3: Newton's Method

input : $\mathcal{E}(\mathbf{w}(n)) : \mathbb{R}^n \to \mathbb{R}$ a twice differentiable function
$\mathbf{w}(0)$ an initial solution

output : $\mathbf{w}^*$, a local minimum of the cost function $\mathcal{E}$.

begin
  $n \leftarrow 0$;
  while STOP-CRIT and $(k < k_{\max})$ **do**
    $\mathbf{w}(n+1) \leftarrow \mathbf{w}(n) + \Delta\mathbf{w}(n)$;
    With $\Delta\mathbf{w}(n) = -\mathbf{H}^{-1}(\mathbf{w}(n))\Delta\mathcal{E}(\mathbf{w}(n))$;
    $n \leftarrow n + 1$;
  return $\mathbf{w}(n)$
end

Newton's Method

Correspondence Between $f(\mathbf{x})$ and $\mathcal{E}_{av}(\mathbf{w})$	
Quadratic function	Cost function
Parameter vector $\mathbf{x}(n)$	Synaptic weight vector $\mathbf{w}(n)$
Gradient vector $\partial f(\mathbf{x})/\partial \mathbf{x}$	Gradient vector $\mathbf{g} = \partial \mathcal{E}_{av}/\partial \mathbf{w}$
Matrix $\mathbf{A}$	Hessian matrix $\mathbf{H}$

Newton's Method

Algorithm 3: Newton's Method

input : $\mathcal{E}(\mathbf{w}(n))\colon \mathbb{R}^n \to \mathbb{R}$ a twice differentiable function
$\mathbf{w}(0)$ an initial solution

output : $\mathbf{w}^*$, a local minimum of the cost function $\mathcal{E}$.

begin
- $n \leftarrow 0$;
- **while** STOP-CRIT and $(k < k_{\max})$ **do**
  - $\mathbf{w}(n+1) \leftarrow \mathbf{w}(n) + \Delta\mathbf{w}(n)$;
  - With $\Delta\mathbf{w}(n) = -\mathbf{H}^{-1}(\mathbf{w}(n))\Delta\mathcal{E}(\mathbf{w}(n))$;
  - $n \leftarrow n + 1$;
- return $\mathbf{w}(n)$

end

Algorithm for Newton's Method

Algorithm 3: Newton's Method

input : $f\colon \mathbb{R}^n \to \mathbb{R}$ a twice differentiable function
$\mathbf{x}^{(0)}$ an initial solution

output : $\mathbf{x}^*$, a local minimum of the cost function f.

begin
- $k \leftarrow 0$;
- **while** STOP-CRIT and $(k < k_{\max})$ **do**
  - $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \delta^{(k)}$;
  - With $\delta^{(k)} = -\mathbf{H}_f\left(\mathbf{x}^{(k)}\right)^{-1}\nabla f\left(\mathbf{x}^{(k)}\right)$;
  - $k \leftarrow k + 1$;
- return $\mathbf{x}^{(k)}$

end

Newton's Method

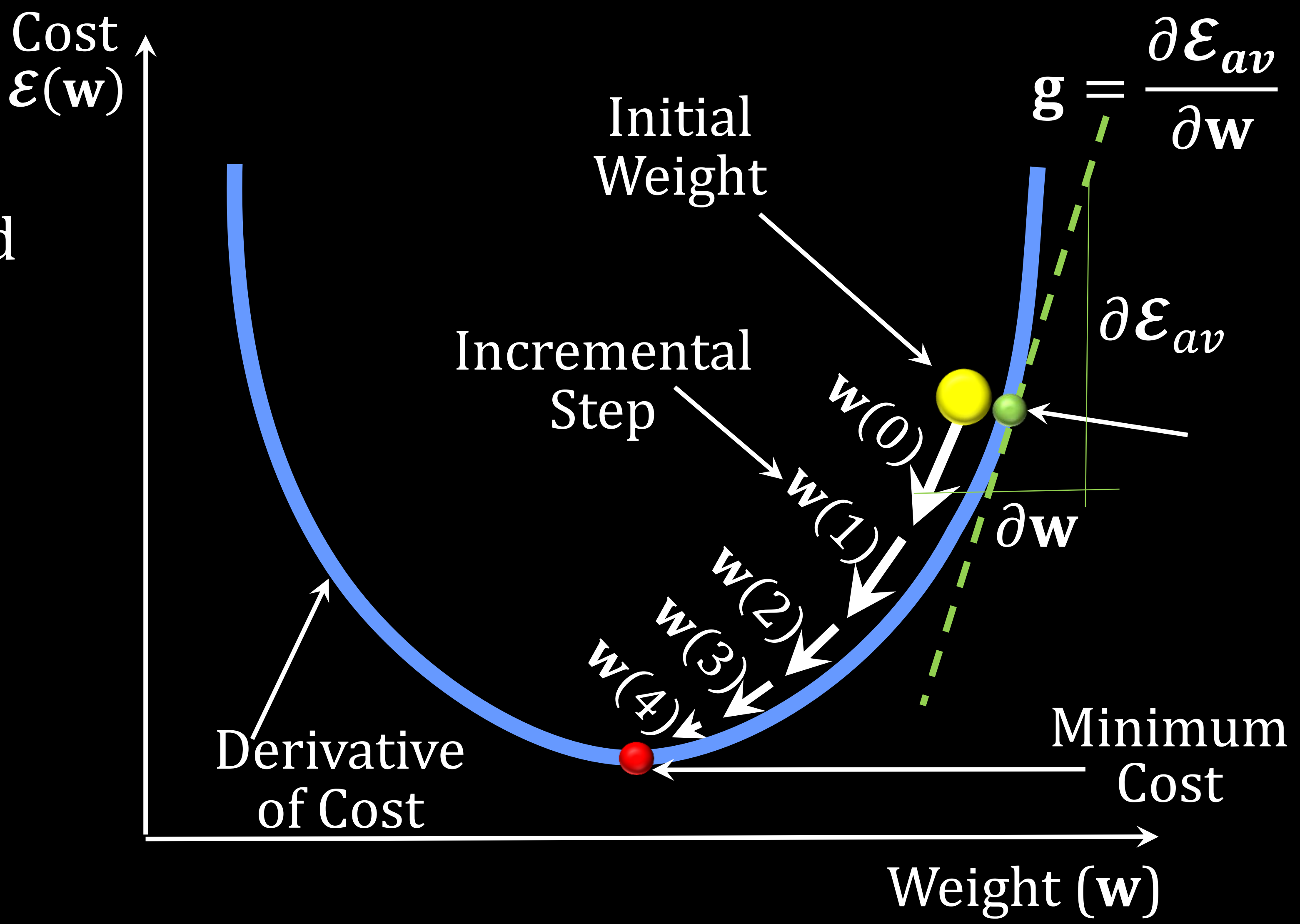
$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} + \Delta\mathbf{w}^{(k)}(n)$$

the change $\nabla\mathcal{E}(\mathbf{w})$ is minimized when $\mathbf{g}(n) + \mathbf{H}(n)\nabla\mathbf{w}(n) = 0$

That is:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \nabla\mathbf{w}(n)$$
$$= \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n)$$

$$\nabla\mathbf{w}(n) = -\mathbf{H}^{-1}(n)\mathbf{g}(n)$$



https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

Newton's Method

- The matrix $\mathbf{H}(n)$ is the m-by-m Hessian cost function of $\mathcal{E}(\mathbf{w})$ evaluated at the point $\mathbf{w}(n)$ defined by

$$\mathbf{H} = \nabla^2\mathcal{E}(\mathbf{w}) = \begin{bmatrix} \frac{\partial^2\mathcal{E}}{\partial w_1^2} & \frac{\partial^2\mathcal{E}}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^{2\mathcal{E}}\mathcal{C}}{\partial w_1 \partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2\mathcal{E}}{\partial w_2 \partial w_1} & \frac{\partial^2\mathcal{E}}{\partial w_2^2} & \cdots & \frac{\partial^2\mathcal{E}}{\partial w_2 \partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2\mathcal{E}}{\partial w_m \partial w_1} & \frac{\partial^2\mathcal{E}}{\partial w_m \partial w_2} & \cdots & \frac{\partial^2\mathcal{E}}{\partial w_m^2} \end{bmatrix} \tag{3.15}$$

- which is twice continuously differentiable with respect to the elements of $\mathbf{w}$.

- Differentiating (3.14) w.r.t. $\Delta\mathbf{w}$, the change $\nabla\mathcal{E}(\mathbf{w})$ is minimized when

$$\mathbf{g}(n) + \mathbf{H}(n)\nabla\mathbf{w}(n)=0$$

Newton's Method

- Solving this equation for $\nabla\mathbf{w}(n)$ yields

$$\nabla\mathbf{w}(n) = -\mathbf{H}^{-1}(n)\mathbf{g}(n)$$

- That is:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \nabla\mathbf{w}(n) = \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n) \tag{3.16}$$

- where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian cost function $\mathcal{E}(\mathbf{w})$.
- Newton's method converges quickly asymptotically and does not exhibit the zigzagging behavior that characterizes the method of steepest descent.
- Newton's Method for optimization: https://youtu.be/mzibEPOyTJU

Newton's Method

```
% Newton's Method (Without Pre-Conditioner)
% Written by Soumitra Sitole
% Date: Mar 4, 2017
%https://www.mathworks.com/matlabcentral/fi
%leexchange/62012-newtonian-method-
%optimizing-two-variable-functions
%Newtonian Method (Optimizing Two Variable Functions)
Clc; clear;
format long

% Function Definition (Enter your Function
here):
syms X Y;
f = X - Y + 2*X^2 + 2*X*Y + Y^2;

% Initial Guess (Choose Initial Guesses):
x(1) = 1;
y(1) = -5;
e = 10^(-8); % Convergence Criteria
i = 1; % Iteration Counter
```

```
% Gradient and Hessian Computation:
df_dx = diff(f, X);
df_dy = diff(f, Y);
J = [subs(df_dx,[X,Y], [x(1),y(1)])
subs(df_dy, [X,Y], [x(1),y(1)])]; %
Gradient
ddf_ddx = diff(df_dx,X);
ddf_ddy = diff(df_dy,Y);
ddf_dxdy = diff(df_dx,Y);
ddf_ddx_1 = subs(ddf_ddx, [X,Y],
[x(1),y(1)]);
ddf_ddy_1 = subs(ddf_ddy, [X,Y],
[x(1),y(1)]);
ddf_dxdy_1 = subs(ddf_dxdy, [X,Y],
[x(1),y(1)]);
H = [ddf_ddx_1, ddf_dxdy_1; ddf_dxdy_1,
ddf_ddy_1]; % Hessian
S = inv(H); % Search Direction
```

Newton's Method

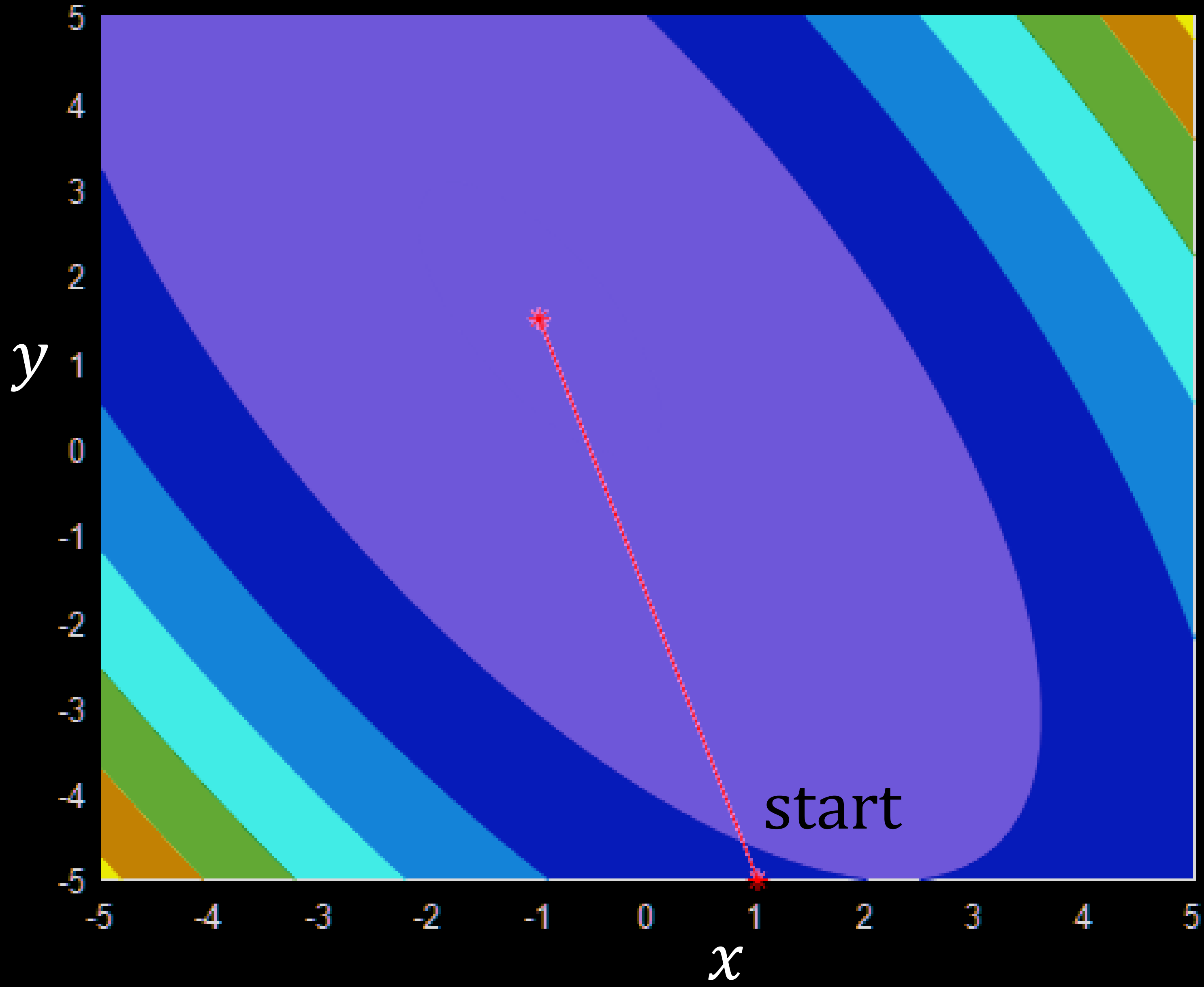
```
% Optimization Condition:
while norm(J) > e
    I = [x(i),y(i)]';
    x(i+1) = I(1)-S(1,:)*J';
    y(i+1) = I(2)-S(2,:)*J';
    i = i+1;
    J = [subs(df_dx,[X,Y], [x(i),y(i)])
subs(df_dy, [X,Y], [x(i),y(i)])]; % Updated
Jacobian
    ddf_ddx_1 = subs(ddf_ddx, [X,Y],
[x(i),y(i)]);
    ddf_ddy_1 = subs(ddf_ddy, [X,Y],
[x(i),y(i)]);
    ddf_dxdy_1 = subs(ddf_dxdy, [X,Y],
[x(i),y(i)]);
    H = [ddf_ddx_1, ddf_dxdy_1; ddf_dxdy_1,
ddf_ddy_1]; % Updated Hessian
    S = inv(H); % New Search Direction
end
```

```
% Result Table:`
Iter = 1:i;
X_coordinate = x';
Y_coordinate = y';
Iterations = Iter';
T =
table(Iterations,X_coordinate,Y_coordinate)
;

% Plots:
fcontour(f, 'Fill', 'On');
hold on;
plot(x,y,'*-r');
grid on;

% Output:
fprintf('Initial Objective Function Value:
```

Newton's Method

```
%d\n\n',subs(f,[X,Y], [x(1),y(1)]));
if (norm(J) < e)
    fprintf('Minimum succesfully
obtained...\n\n');
end
fprintf('Number of Iterations for
Convergence: %d\n\n', i);
fprintf('Point of Minima: [%d,%d]\n\n',
x(i), y(i));
fprintf('Objective Function Minimum Value
after Optimization: %f\n\n', subs(f,[X,Y],
[x(i),y(i)]));
disp(T)
```

Newton's Method



Gauss-Newton Method

- The Gauss-Newton method is applicable to a cost function that is expressed as the sum of error squares. Let

$$\mathcal{C}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n} e^2(i) \qquad (3.17)$$

- The error signal $e(i)$, which is a function of the weight vector $\mathbf{w}$ is fixed over the observation interval $1 \leq i \leq n$.
- Given an operating point $\mathbf{w}(n)$, we linearize dependence of $e(i)$ on $\mathbf{w}$ by

$$e'(i,\mathbf{w}) = e(i) + \left[\frac{\partial e(i)}{\partial \mathbf{w}}\right]^T_{\mathbf{w}=\mathbf{w}(n)'} \left(\mathbf{w} - \mathbf{w}(n)\right), \quad i = 1, 2, \dots, n \qquad (3.18)$$

- In matrix notation,

$$e'(n,\mathbf{w}) = e(n) + \mathbf{J}(n)\left(\mathbf{w} - \mathbf{w}(n)\right) \qquad (3.19)$$

Where the error vector

$$\mathbf{e}(n) = [e(1), e(2), \dots, e(n)]^T$$

Gauss-Newton Method

- The n-by-m Jacobian matrix $\mathbf{J}(n)$ of $e(n)$,

$$\mathbf{J}(n) = \begin{bmatrix} \frac{\partial e(1)}{\partial w_1} & \frac{\partial e(1)}{\partial w_2} & \cdots & \frac{\partial e(1)}{\partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e(2)}{\partial w_1} & \frac{\partial e(2)}{\partial w_2} & \cdots & \frac{\partial e(2)}{\partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e(n)}{\partial w_1} & \frac{\partial e(n)}{\partial w_2} & \cdots & \frac{\partial e(n)}{\partial w_m} \end{bmatrix} \qquad (3.20)$$

- The Jacobian $\mathbf{J}(n)$ is transpose of the m-by-n gradient matrix $\nabla\mathbf{e}(n)$, where

$$\nabla\mathbf{e}(n) = [\nabla e(1), \nabla e(2), \ldots, \nabla e(n)]$$

Gauss-Newton Method

- The updated weight vector $\mathbf{w}(n)$ is defined by

$$\mathbf{w}(n+1) = \arg\min_{\mathbf{w}} \left\{\frac{1}{2}\|e'(n,\mathbf{w})\|^2\right\} \qquad (3.21)$$

- The squared Euclidean norm of $e'(n,\mathbf{w})$ is then

$$\frac{1}{2}\|\mathbf{e}'(n,\mathbf{w})\|^2$$
$$= \frac{1}{2}\|\mathbf{e}(n)\|^2 + \mathbf{e}^T(n)\mathbf{J}(n)\big(\mathbf{w}-\mathbf{w}(n)\big) + \frac{1}{2}\big(\mathbf{w}-\mathbf{w}(n)\big)^T\mathbf{J}^T(n)\mathbf{J}(n)\big(\mathbf{w}-\mathbf{w}(n)\big)$$

- Differentiating this w.r.t. $\mathbf{w}$ and setting the result equal to zero, we obtain

$$\mathbf{J}^T(n)\mathbf{e}(n) + \mathbf{J}^T(n)\mathbf{J}(n)\big(\mathbf{w}-\mathbf{w}(n)\big) = 0$$

- Solving for $\mathbf{w}$

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \big(\mathbf{J}^T(n)\mathbf{J}(n)\big)^{-1}\mathbf{J}^T(n)\mathbf{e}(n) \qquad (3.22)$$

- (3.22) describes Gauss-Newton method (for only nonsingular $\mathbf{J}^T(n)\mathbf{J}(n)$)

Gauss-Newton Method

Algorithm 4: Gauss-Newton's Method

input : $\mathcal{E}\big(\mathbf{w}(n)\big): \mathbb{R}^n \to \mathbb{R}$ a function such that $\mathcal{E}\big(\mathbf{w}(n)\big) = \sum_{i=1}^{m} \mathcal{E}_i\big(\mathbf{w}(n)\big)$ where all $\mathcal{E}_i$ are differentiable functions from $\mathbb{R}^n$ to $\mathbb{R}$.
$\mathbf{w}(0)$ an initial solution

output : $\mathbf{w}^*$, a local minimum of the cost function $\mathcal{E}$.

begin

$n \leftarrow 0$;

while STOP-CRIT and $(n < n_{\max})$ **do**

$\mathbf{w}(n+1) \leftarrow \mathbf{w}(n) + \Delta\mathbf{w}(n)$;

With $\Delta\mathbf{w}(n) = arg \min_{\Delta\mathbf{w}} \left\| W\big(\mathbf{w}(n)\big) + J_{W(\mathbf{w}(n))} \Delta\mathbf{w}(n) \right\|^2$;

$n \leftarrow n + 1$;

return $\mathbf{w}(n)$

end

Adaptive Gradient (AdaGrad)

- For all the previously discussed algorithms the learning rate remains constant. So the key idea of AdaGrad is to have an adaptive learning rate for each of the weights.
- It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.
- For brevity, we use g_t to denote the gradient at time step t. $\mathbf{g}_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t, η is learning rate and $\nabla\theta_i$ is the partial derivative of loss function J(θ_i).

$$g_{t,i} = \nabla_\theta J(\theta_{i,t})$$

The SGD update for every parameter θ_i at time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta g_{t,i}$$

Adaptive Gradient (AdaGrad)

- In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where G_t is the sum of squares of the past gradients w.r.t to all parameters θ.

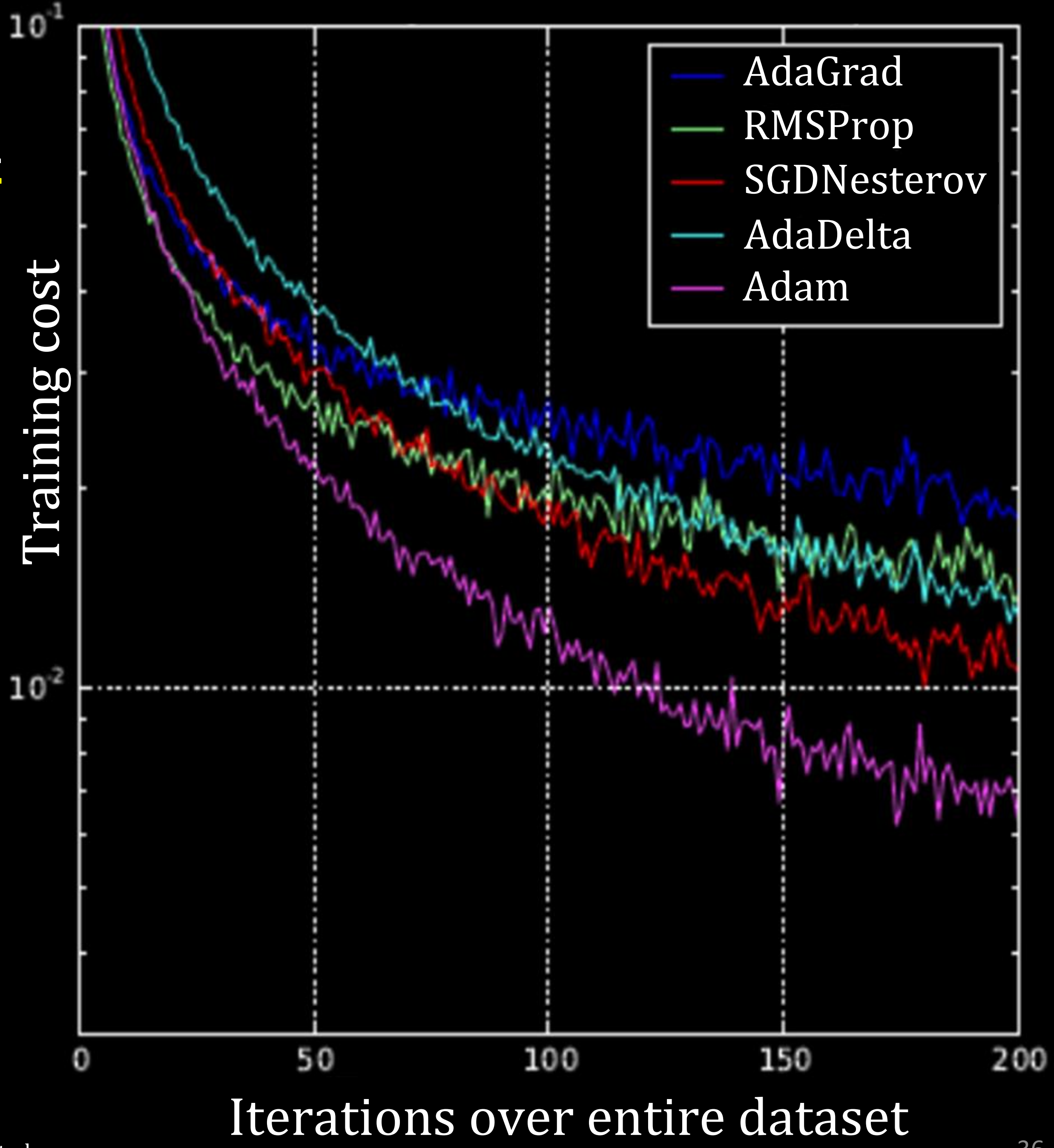
- The benefit of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01.
- Its main weakness is the accumulation of the squared gradients(G_t) in the denominator.
- Since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small and further resulting in a vanishing gradient problem.
- More on this subject can be found at the following reference

https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

Adaptive Gradient (AdaGrad)

Observing the animation at the beginning of this article, we consider the following points

- The SGD algorithm (red) is stuck at a saddle point. So SGD algorithm can only be used for shallow networks.
- All the other algorithms except SGD finally converges one after the other, AdaDelta being the fastest followed by momentum algorithms.
- AdaGrad and AdaDelta algorithm can be used for sparse data.
- Momentum and NAG work well for most cases but is slower.
- Animation for Adam is not available but from the plot above it is observed that it is the fastest algorithm to converge to minima.



Adaptive Moment Estimation (Adam)

- Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum.
- Adam computes adaptive learning rates for each parameter.
- In addition to storing an exponentially decaying average of past squared gradients m_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum.
- Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.
- Hyper-parameters $\beta_1, \beta_2 \in [0,1)$ control the exponential decay rates of these moving averages.

Optimization Algorithms in Neural Networks
https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

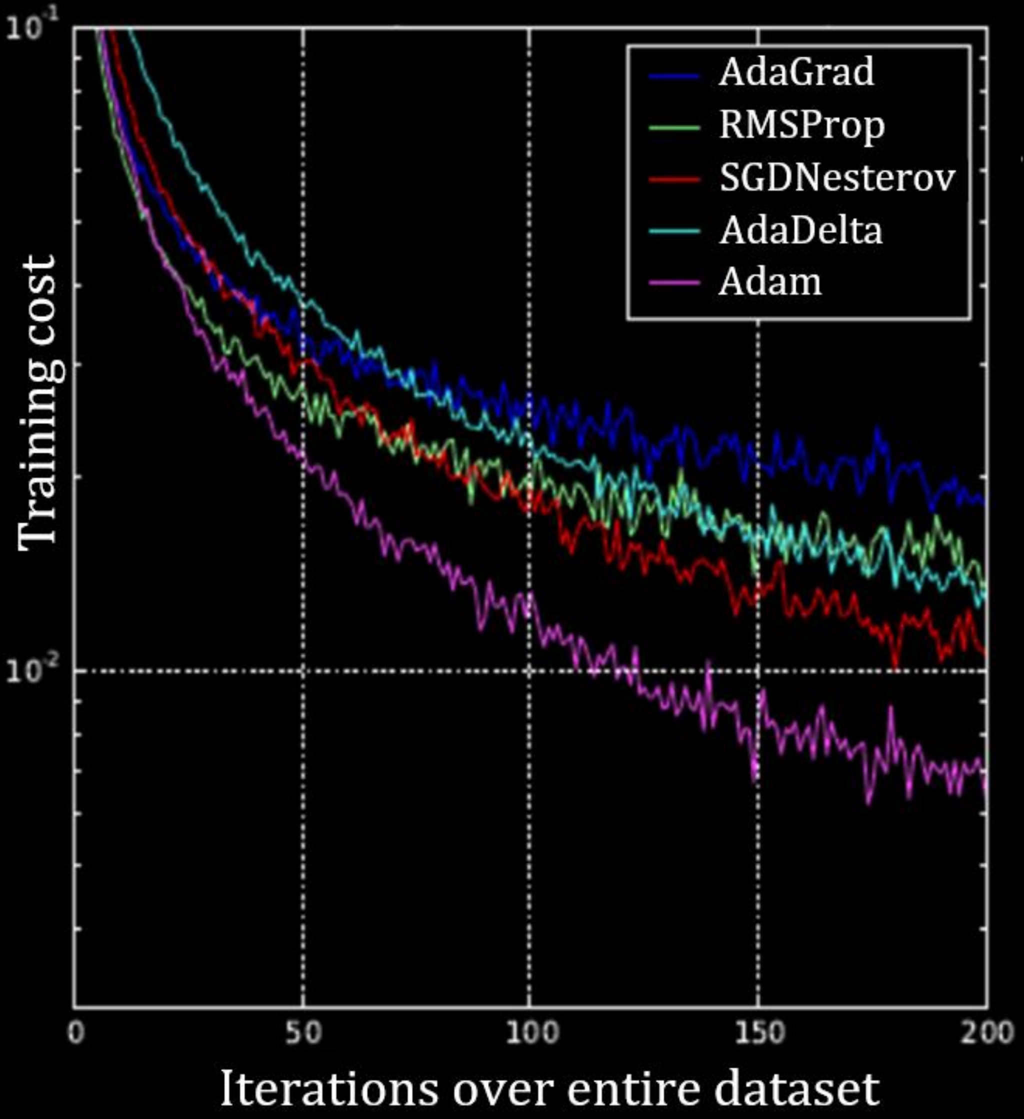
Gentle Introduction to the Adam Optimization Algorithm for Deep Learning
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Adaptive Moment Estimation (Adam)

- We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.



https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

---

https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

Algorithm 1. Adam Optimization Algorithm

Require: Objective function $f(\theta)$, initial parameters θ_0, stepsize hyperparameter α, exponential decay rates β_1, β_2 for moment estimates, tolerance parameter $\lambda > 0$ for numerical stability, and decision rule for declaring convergence of θ_t in scheme.

procedure $\text{ADAM}(f, \theta_0 \;;\; \alpha,\, \beta_1,\, \beta_2)$
    $m_0,\, v_0,\, t \leftarrow [0,\, 0,\, 0]$ # Initialize moment estimates
    # and timestep to zero
    # Begin optimization procedure
    while θ_t has not converged **do**
        $t \leftarrow t + 1$ # Update timestep
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ # Compute gradient of objective
        $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ # Update first moment estimate
        $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$ # Update second moment estimate
        $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ # Create unbiased estimate $\widehat{m}_t$
        $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ # Create unbiased estimate $\widehat{v}_t$
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \lambda)$ # Update objective parameters
    return θ_t # Return final parameters

https://www.math.purdue.edu/~nwinovic/figures/adam.png

Adam Implementation in Python

```
for t in range(n_iter):
    # calculate gradient g(t)
    g = derivative(x[0], x[1])
    # build a solution one variable at a time
    for i in range(bounds.shape[0]):
        # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
        m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
        # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
        v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2
        # mhat(t) = m(t) / (1 - beta1(t))
        mhat = m[i] / (1.0 - beta1**(t+1))
        # vhat(t) = v(t) / (1 - beta2(t))
        vhat = v[i] / (1.0 - beta2**(t+1))
        # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + ep)
        x[i] = x[i] - alpha * mhat / (sqrt(vhat) + eps)
# evaluate candidate point
        score = objective(x[0], x[1])
        # keep track of solutions
        solutions.append(x.copy())
        # report progress
        print('>%d f(%s) = %.5f' % (t, x, score))
        k = k + 1
        cost[k] = score
    return solutions
```

Optimization Parameters

iterations n_iter = 60

steps size alpha = 0.02

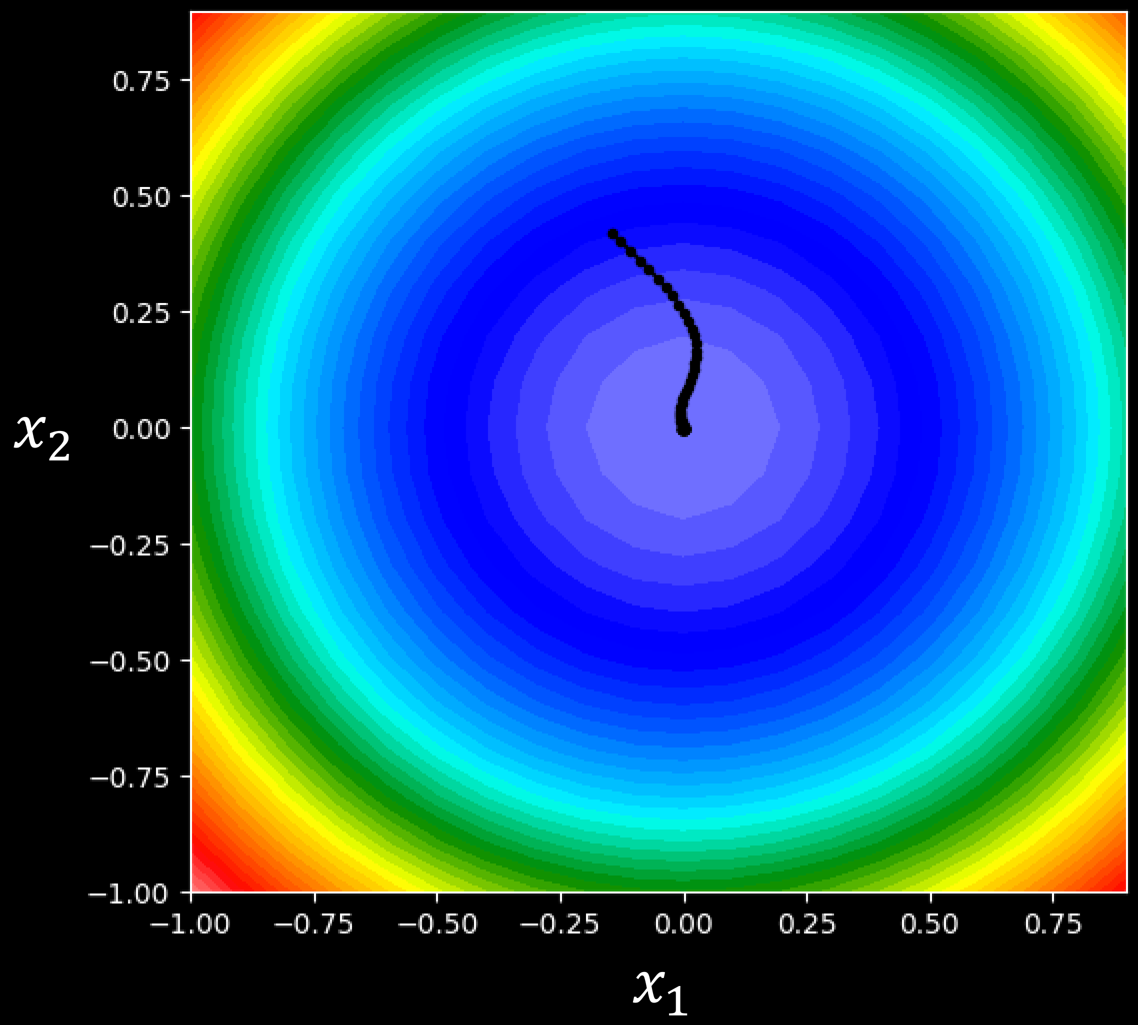
average gradient factor beta1 = 0.8

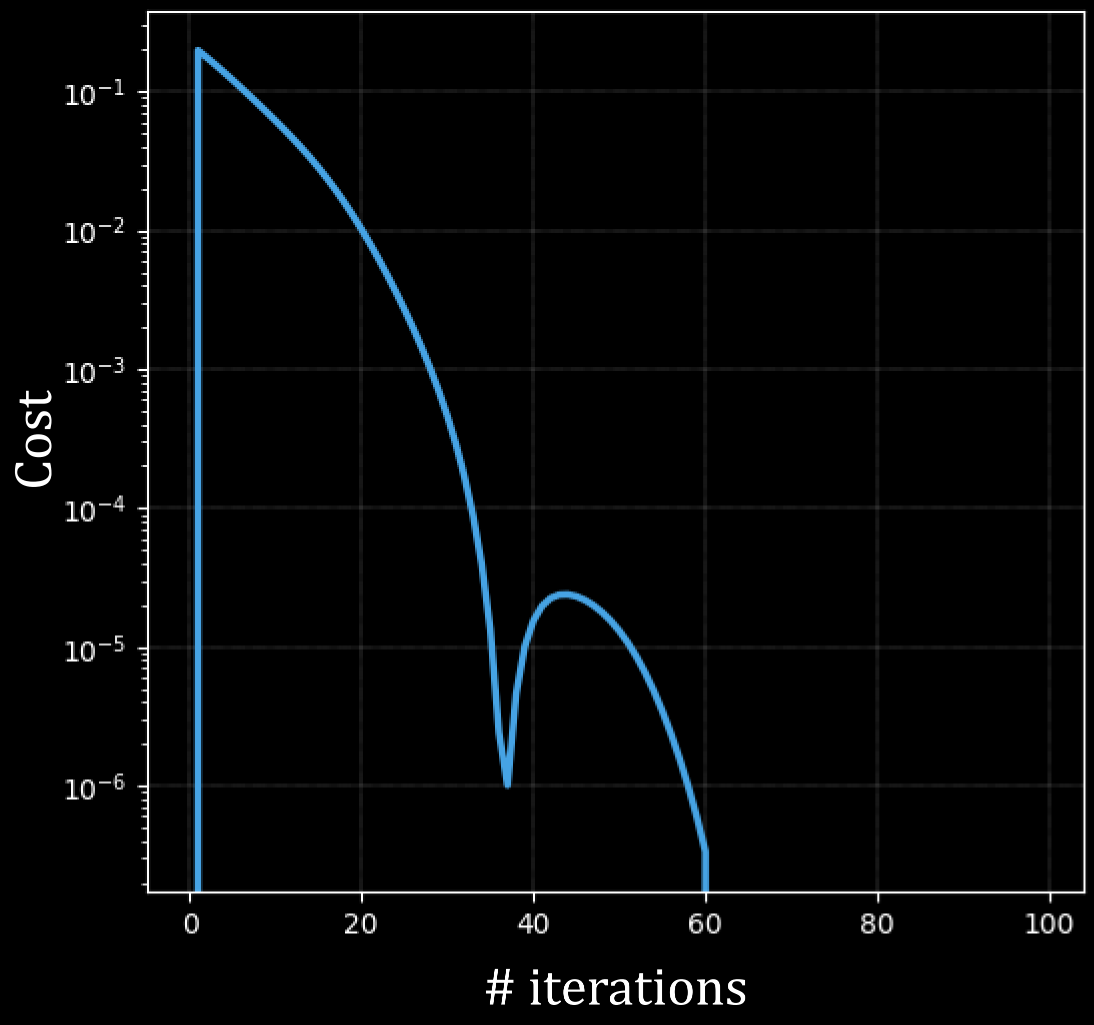
average squared gradient factor beta2 = 0.999

https://machinelearningmastery.com/adam-optimization-from-scratch/

Adaptive Moment Estimation (Adam)

Optimization Trajectory and Cost Function Plot





https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html