3.1 Introduction

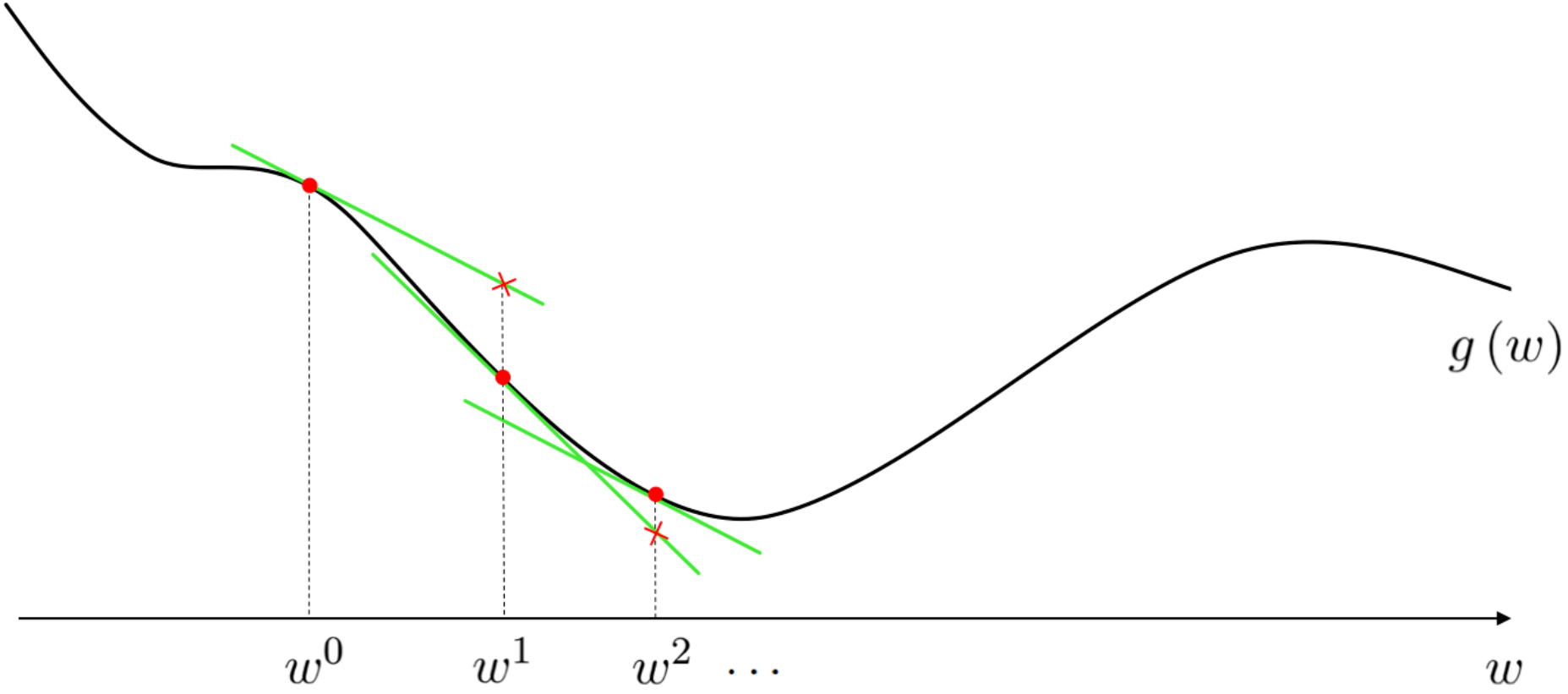
- In this Chapter we mirror the structure of our discussion in the previous one in describing *first order optimization methods*.
- We begin with a discussion of the *first order optimality condition* - which codifies how the first derivative(s) of a function characterize its minima.

- We then discuss some fundamental concepts related to the geometric nature of (tangent) hyperplanes and in particular the first order Taylor series.
- We then explore *first order algorithms* and detail extensively the *gradient descent* algorithm, a popular local optimization scheme that works by leveraging this first order geometry, as well as advanced versions of this algorithm.

Big picture view of the gradient descent algorithm

- The first derivative(s) of a function helps form the best *linear* approximation to the function locally (called the *first order Taylor series approximation*).
- It is extremely easy to compute the descent direction of a line or hyperplane regardless of its dimension.
- And the descent direction of the tangent hyperplane is also a descent direction for the function itself.

- *Gradient descent* is a local optimization algorithm where we simply steal this descent direction at each step (as illustrated below).

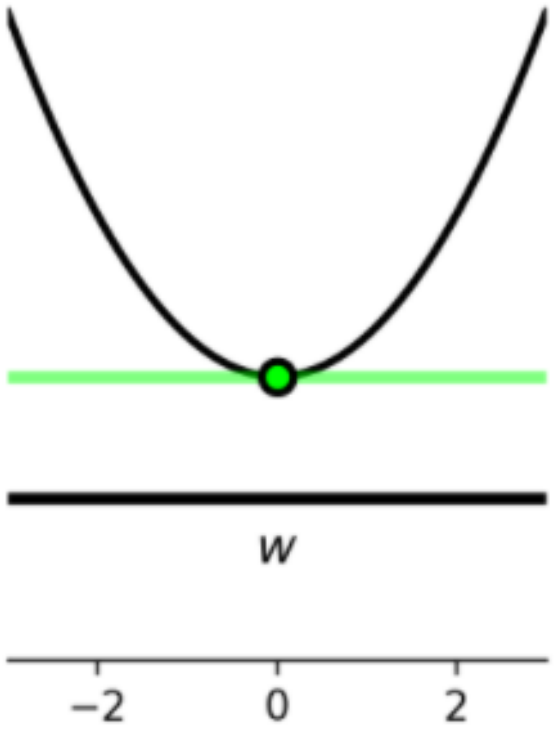

$g\,(w)$
w^0
w^1
w^2
$\cdots$
w

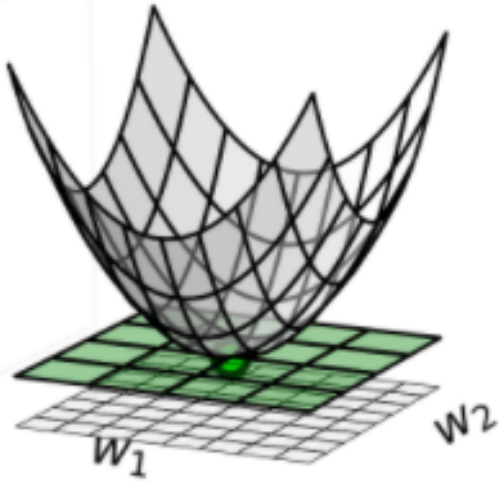
3.2 The first order optimality condition

- In this Section we discuss the foundational first order concept on which many practical optimization algorithms are built: the first order optimality condition.
- This is the first order analog of the zero order condition discussed in the previous Chapter.
- The first order condition codifies the consistent behavior of how any differentiable function's first derivative(s) behave at its minima and maxima.

The first-order condition

- Below we plot a quadratic functions in two and three dimensions, and mark the global minimum point on each with a green point.
- In each panel we also draw the first order Taylor series approximation - a tangent line/hyperplane - generated by the first derivative(s) at the function's minimum value.
- In terms of the behavior of the first order derivatives here we see - in both instances - that the tangent line/hyperplane is perfectly flat, indicating that the first derivative(s) is exactly zero at the function's minimum.


w
−2
0
2


w_1
w_2

- This sort of first order behavior is universal regardless of the function one examines and - moreover - it holds regardless of the dimension of a function's input.
- That is, first order derivatives are always zero at the minima of a function.
- This is because minimum values of a function are naturally located at 'valley floors' where a tangent line or hyperplane tangent to the function is perfectly flat, and thus has zero-valued slope(s).

- Codifying this in the language of mathematics, when ***N=1*** any point ***v*** where

$$\frac{\mathrm{d}}{\mathrm{d}w} g\left(v\right) = 0$$

is a potential minimum.

- Analogously with general ***N*** dimensional input, any ***N*** dimensional point $\mathbf{v}$ where *every* partial derivative of ***g*** is zero, that is

$$\frac{\partial}{\partial w_1} g(\mathbf{v}) = 0$$
$$\frac{\partial}{\partial w_2} g(\mathbf{v}) = 0$$
$$\vdots$$
$$\frac{\partial}{\partial w_N} g(\mathbf{v}) = 0$$

  is a potential minimum.

- This system of ***N*** equations is naturally referred to as the *first order system of equations*.
- We can write the first order system more compactly using gradient notation as

$$\nabla g\left(\mathbf{v}\right) = \mathbf{0}_{N\times 1}.$$

- *In principle* this is a very useful characterization of minimum points.
- It gives us a concrete alternative to seeking out a function's minimum points *directly* via some zero-order approach.
- The alternative - solve a function's first order system of equations.

- However two problems with the first order characterization of minima.
- First off, with few exceptions it is virtually impossible to solve a general function's first order systems of equations 'by hand'.
- That is, to solve such equations algebraically for 'closed form' solutions one can write out on paper.

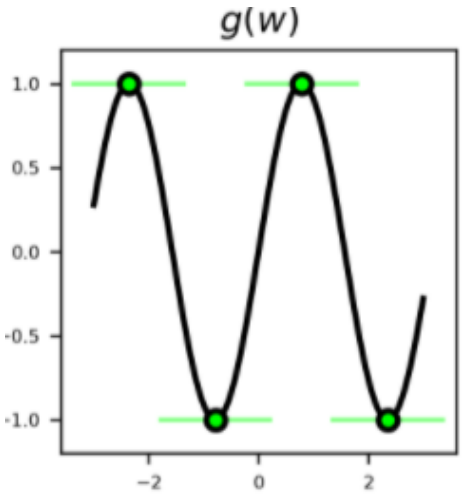
- The other problem: the *first order optimality condition* does not define only minima of a function, but other points as well.
- The first order condition also equally characterizes *maxima* and *saddle points* of a function - as we see in a few simple examples below.

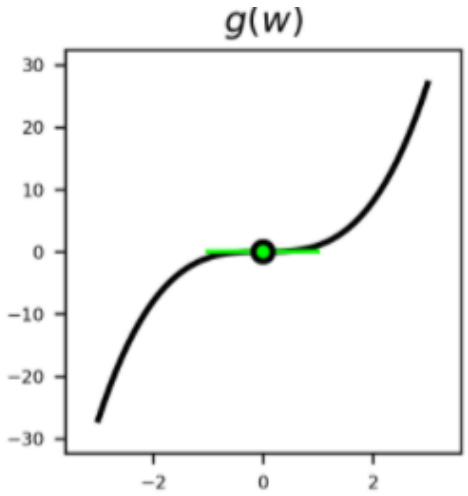
Example: Finding points of zero derivative for single-input functions graphically

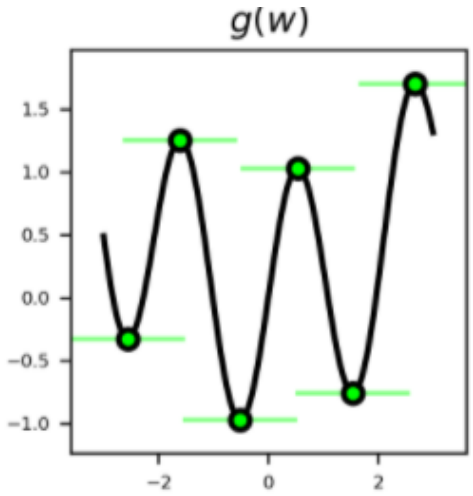
- Below we plot the three functions

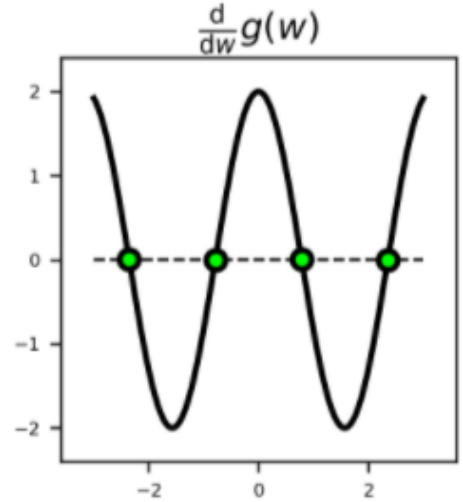
$$g(w) = \sin(2w)$$
$$g(w) = w^3$$
$$g(w) = \sin(3w) + 0.1w^2$$

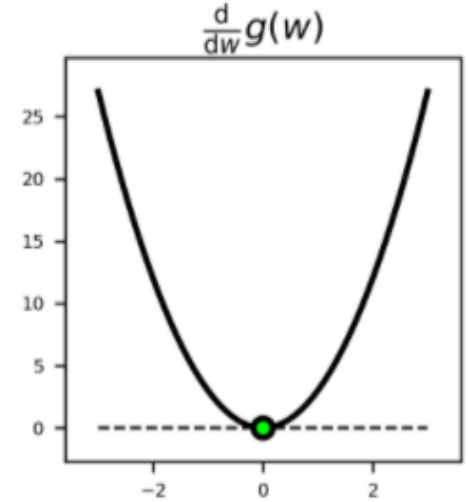
- For each we mark all the zero derivative points in green and draw the first order Taylor series approximations/tangent line.
- Below each function we plot its first derivative, highlighting the points where it takes on the value zero as well.

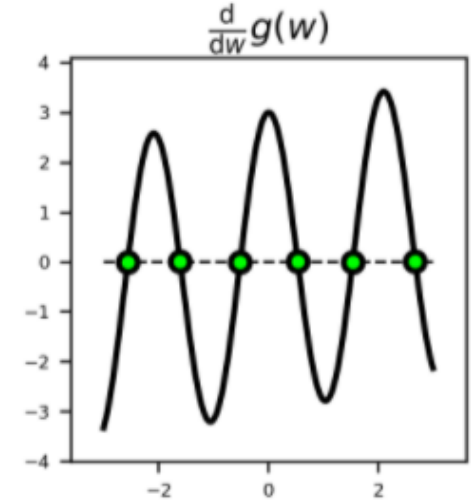

$g(w)$
1.0
0.5
0.0
-0.5
-1.0
−2
0
2


$g(w)$
30
20
10
0
−10
−20
−30
−2
0
2


$g(w)$
1.5
1.0
0.5
0.0
−0.5
−1.0
−2
0
2


$\frac{\mathrm{d}}{\mathrm{d}w}g(w)$
2
1
0
−1
−2
−2
0
2


$\frac{\mathrm{d}}{\mathrm{d}w}g(w)$
25
20
15
10
5
0
−2
0
2


$\frac{\mathrm{d}}{\mathrm{d}w}g(w)$
4
3
2
1
0
−1
−2
−3
−4
−2
0
2

- *local minima* or points that are the smallest with respect to their immediate neighbors, like the one around the input value ***w=2*** in the right panel
- *local and global maxima* or points that are the largest with respect to their immediate neighbors, like the one around the input value ***w=-2*** in the right panel
- *saddle points* like the one shown in the middle panel, that are neither maximal nor minimal with respect to their immediate neighbors

- The previous example illustrate the full swath of points having zero-valued derivative(s).
- This includes multi-input functions as well *regardless of dimension*.
- Taken together all such points are collectively referred to as *stationary points* or *critical points*.

Special cases where the first order system can be solved 'by hand'

- There are a handful of relatively simple but important examples where one can compute the solution to a first order system by hand.
- Or, at least, one can show algebraically that they reduce to a *linear system of equations* which can be easily solved numerically.

Example: Calculating stationary points of some single-input functions algebraically

- In this Example we use the first order condition for optimality to compute stationary points of the functions

$$
\begin{aligned}
g\left(w\right) &= w^3 \\
g\left(w\right) &= e^w \\
g\left(w\right) &= \sin\left(w\right) \\
g\left(w\right) &= a + bw + cw^2, \;\; c > 0
\end{aligned}
$$

- $g(w) = w^3$, plotted in the middle panel of the second figure above, the first order condition gives $g'(v) = 3v^2 = 0$ which we can visually identify as a saddle point at $v = 0$.
- $g(w) = e^w$, the first order condition gives $g'(v) = e^v = 0$ which is only satisfied as v goes to $-\infty$, giving a minimum.

- $g(w) = \sin(w)$ the first order condition gives stationary points wherever $g'(v) = \cos(v) = 0$ which occurs at odd integer multiples of $\frac{\pi}{2}$, i.e., maxima at

$$v = \frac{(4k+1)\pi}{2}$$

and minima at

$$v = \frac{(4k+3)\pi}{2}$$

where k is anyinteger.

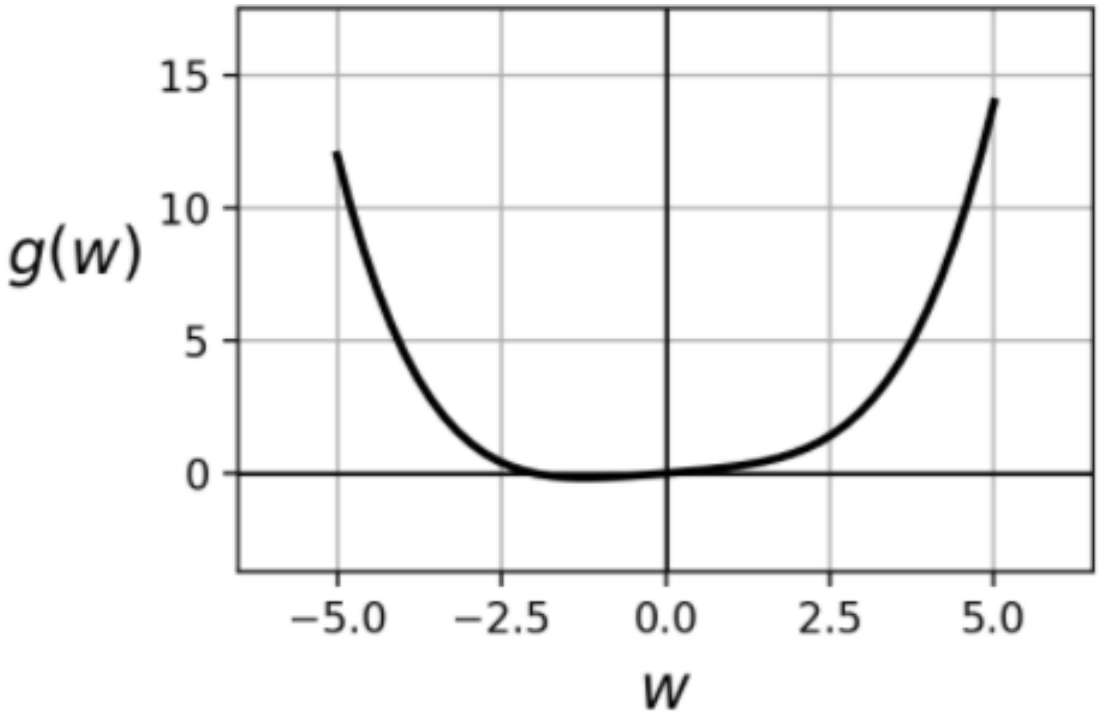
- $g(w) = w^2$ for which the first order condition gives $g'(v) = 2cv + b = 0$ with a minimum at $v = \frac{-b}{2c}$

Example: A simple looking function with difficult to compute (algebraically) global minimum

- Solving the first order equation for even a simple looking function can be quite challenging.
- Take, for example, the simple degree four polynomial

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$

- This is plotted over a short range of inputs containing its global minimum below.


15
10
5
0
g(w)
−5.0
−2.5
0.0
2.5
5.0
w

- The first order system here can be easily computed as

$$\frac{\mathrm{d}}{\mathrm{d}w} g(w) = \frac{1}{50}\left(4w^3 + 2w + 10\right) = 0$$

  This simplifies to

$$2w^3 + w + 5 = 0$$

- This has three possible solutions, but the one providing the minimum of the function $g(w)$ is

$$w = \frac{\sqrt[3]{\sqrt{2031}-45}}{6^{\frac{2}{3}}} - \frac{1}{\sqrt[3]{6\left(\sqrt{2031}-45\right)}}$$

which can be computed - after much toil - using centuries old tricks developed for just such problems.

Example: Stationary points of a general multi-input quadratic function

- Take the general multi-input quadratic function

$$g\left(\mathbf{w}\right) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$$

where $\mathbf{C}$ is an $N \times N$ symmetric matrix, $\mathbf{b}$ is an $N \times 1$ vector, and a is a scalar.

- Computing the first derivative (gradient) we have

$$\nabla g\left(\mathbf{w}\right) = 2\mathbf{C}\mathbf{w} + \mathbf{b}$$

- Setting this equal to zero gives a *symmetric and linear* system of equations of the following form whose solutions are stationary points of the original function

$$\mathbf{C}\mathbf{w} = -\tfrac{1}{2}\mathbf{b}$$

Coordinate descent and the first order optimality condition

- If we write out the first order system one equation at-a-time we have

$$\begin{aligned}\frac{\partial}{\partial w_1}g(\mathbf{v}) &= 0\\ \frac{\partial}{\partial w_2}g(\mathbf{v}) &= 0\\ &\vdots\\ \frac{\partial}{\partial w_N}g(\mathbf{v}) &= 0.\end{aligned}$$

- While this system cannot often be solved in closed form, a simple idea does lead to numerical approach to approximating it instances where *each individual equation* can be easily solved.

- The idea is this: instead of trying to solve the system of equations *at once* we solve each partial derivative equation *one at-a-time*.
- This is often called *coordinate descent*, since in solving each we move along the coordinate axes coordinate-wise (one at-a-time).

- To perform this coordinate descent we initialize at a point $\mathbf{w}^0$, updating its first coordinate by solving

$$\frac{\partial}{\partial w_1} g\left(\mathbf{w}^0\right) = 0$$

  for the optimal first weight $w_1^\star$.

- We do this again, and again, for each coordinate.

- Continuing this pattern to update the n^{th} weight we solve

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{n-1}\right) = 0$$

for $w_n^\star$, and update the n^{th} weight using this value forming the updated set of weights $\mathbf{w}^n$.

- After we sweep through all ***N*** weights a single time we can refine our solution by sweeping through the weights again.
- At the k^{th} such sweep we update the n^{th} weight by solving the single equation

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{k+n-1}\right) = 0$$

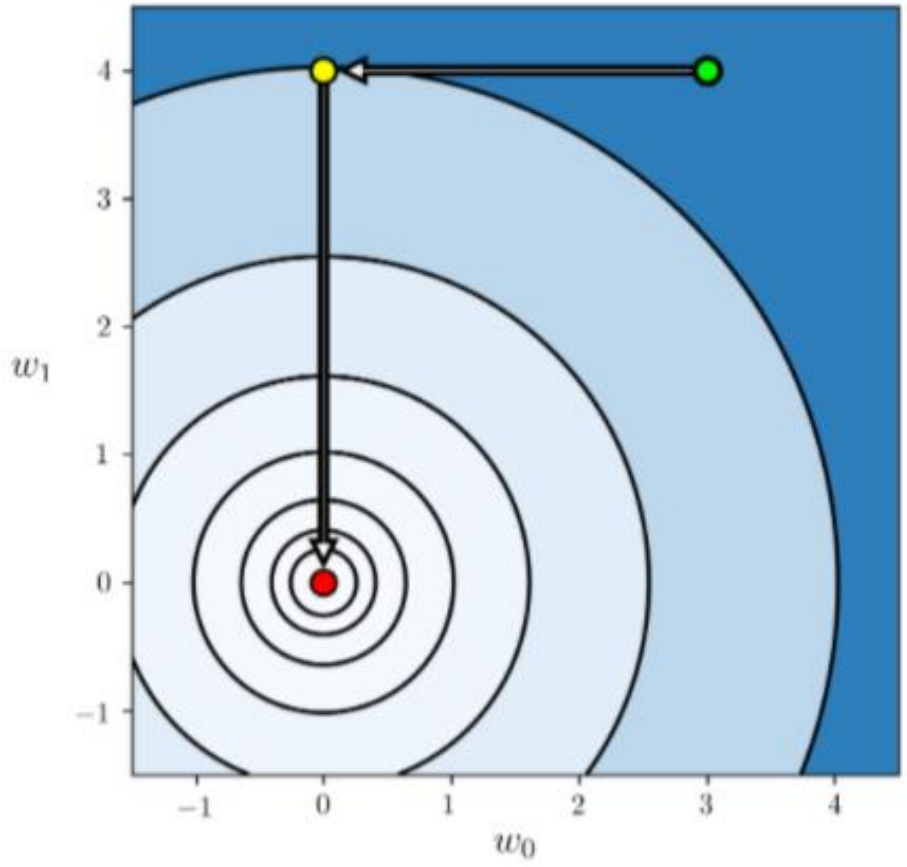
and update the n^{th} weight of $\mathbf{w}^{k+n-1}$, and so on.

Example: Minimizing convex quadratic functions via first order coordinate descent

- First we use this algorithm to minimize the simple quadratic

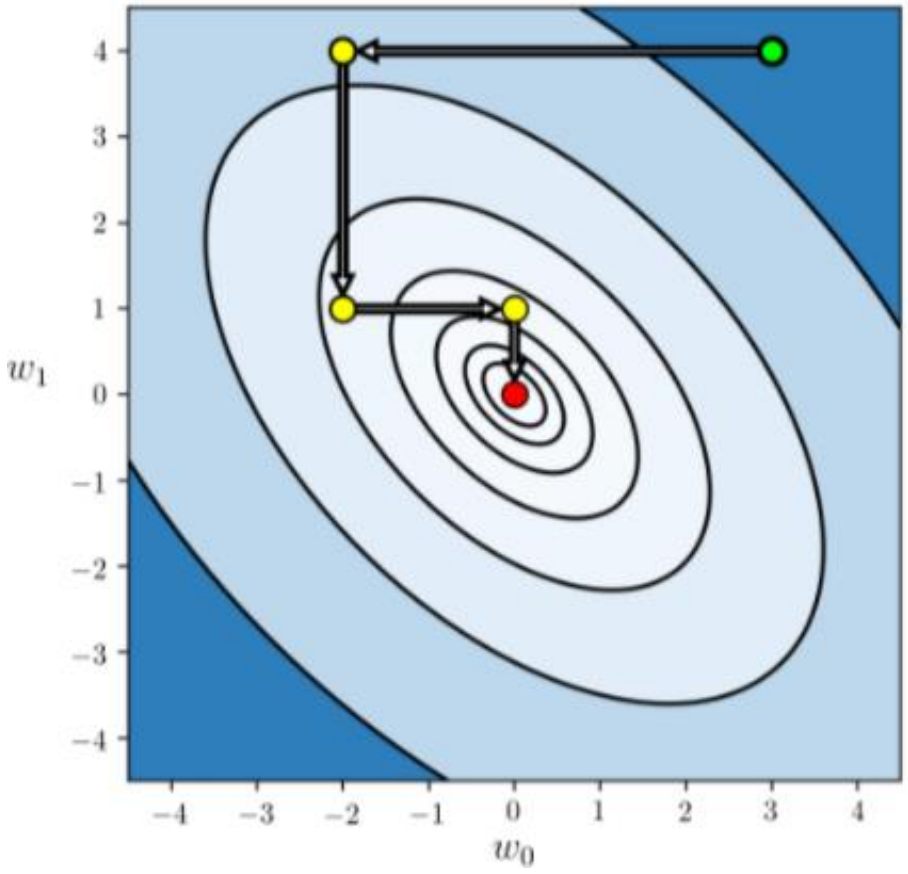
$$g(w_0, w_1) = w_0^2 + w_1^2 + 2$$

- We initialize at $\mathbf{w} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ and run ***1*** iteration of the algorithm - that is all it takes to perfectly minimize the function, as shown below.


4
3
2
1
0
−1
w_1
−1
0
1
2
3
4
w_0

- Below we show a run of ***2*** iterations of the method at the same initial point for the quadratic

$$g(w_0, w_1) = 2w_0^2 + 2w_1^2 + 2w_0 w_1 + 20$$


4
3
2
1
0
−1
−2
−3
−4
w_1
−4
−3
−2
−1
0
1
2
3
4
w_0

Example: Solving systems symmetric equations

- Note how in the previous example the first order system turned out to be *linear*.
- More specifically, we ended up using coordinate descent to solve simple instances of the symmetric linear system

$$\mathbf{C}\mathbf{w} = -\tfrac{1}{2}\mathbf{b}.$$

  which generates a convex quadratic (i.e., when $\mathbf{C}$ is positive semi-defininte).
- Indeed more generally, this coordinate descent method is one very popular way of solving such systems in general.

3.3. The Geometry of First Order Taylor Series

3.4 Computing Gradients Efficiently

- Think about how you perform basic arithemetic - say how you perform the multiplication of two numbers.
- If the two numbers are small - say ***8*** times ***7***, or ***5*** times ***2*** - you can likely do the multiplication in your head.
- Otherwise, for larger numbers, you use an *algorithm* you learned in school.

- The algorithm for multiplication is great - its *simple* and *repetitive*, built from a small list of basic rules, and works regardless of the two numbers you multiply together.
- But *performing* the algorithm yourself is *boring* and *time consuming*, and you can easily mess up too.

- For example, go ahead and compute $140,283,197,523 \times 224,179,234,112$ by hand, won't you?
- Instead you use a *calculator*, it *automates the process of using the multiplication algorithm*.

- An arithmetic calculator allows *you* to compute with much greater efficiency and accuracy, and empowers *you* to use the fruits of arithmetic computation for more important tasks.
- Computing derivatives is just like this.

- You likely learned a bunch of basic rules for computing derivatives in school, and can compute simple examples like $g(w) = w^3$ and $g(w) = \sin(w)$
- But what about this one?

$$g\left(w_1, w_2\right) = 2^{\sin\left(0.1w_1^2 + 0.5w_2^2\right)} \tanh\left(w_2^4 \tanh\left(w_1 + \sin\left(0.2w_2^2\right)\right)\right)$$

- You *could* compute the derivatives yourself, since the process is simple and repetitive, but its also *boring* and *time cosuming* and you could easily mess up.
- Your time is better spent doing more thought-intensive things, so why not use a calculator instead?

- A gradient calculator or *Automatic Differentiator* allows *you* to compute with much greater effeciency and accuracy, and empowers you to use the fruits of gradient computation for more important tasks.
- In Appendix B we describe how to use a powerful and easy to use `Python` Automatic Differentiator called `autograd`.

3.5 Gradient Descent

- In this Section we derive the *gradient descent algorithm*, building on our discussion of tangent hyperplanes in Section 3.4.

The gradient descent algorithm

- Remember, a general local optimization method looks like

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^k.$$

- Here $\mathbf{d}^k$ are *descent direction* vectors and α is called the *steplength* parameter.

- Given what we saw in Section 3.4 we could naturally ask what a local method employing the negative gradient direction at each step might look like, and how it might behave.
- Setting the descent direction $\mathbf{d}^k = -\nabla g\left(\mathbf{w}^{k-1}\right)$ in the above formula, such a sequence of steps would then take the form

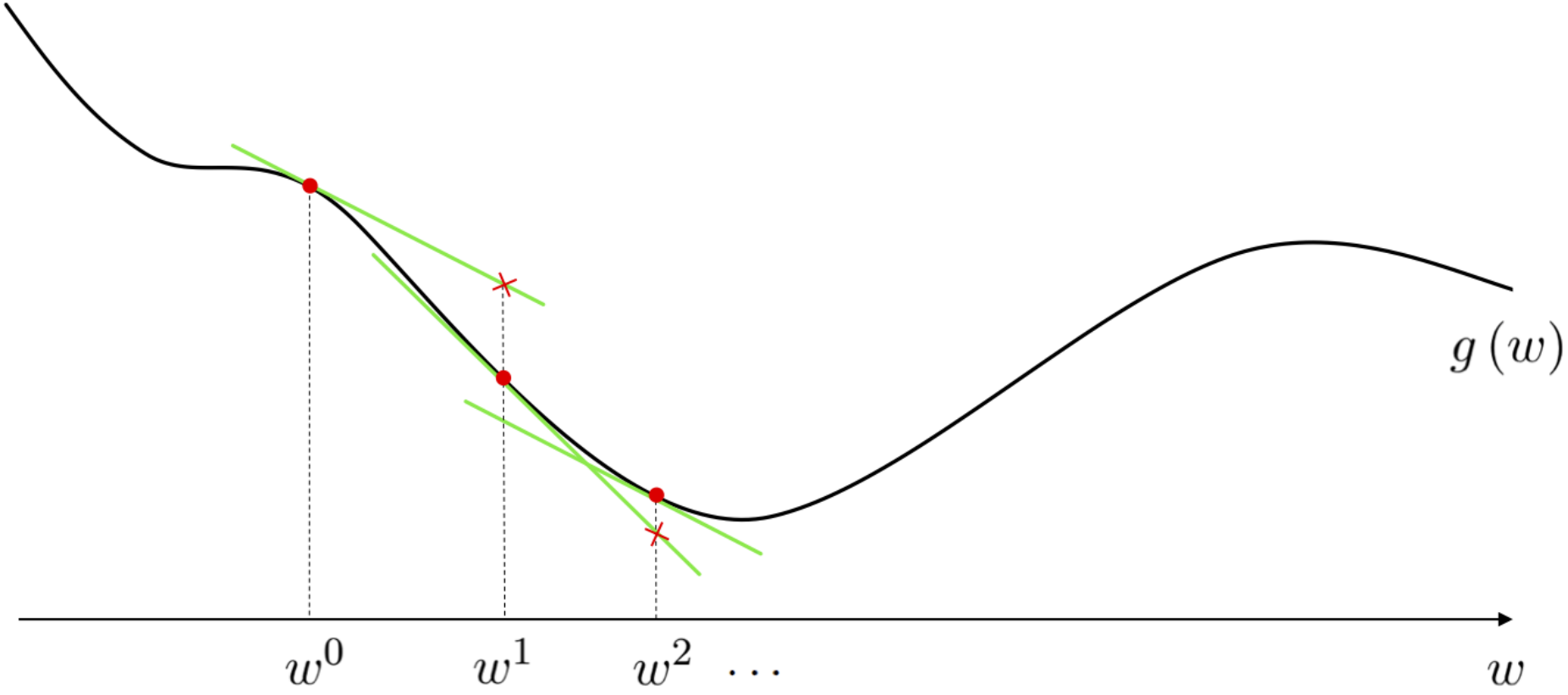
$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

- Because each and every direction is guaranteed to be one of descent, intuitively this seems like a potentially useful instance of local optimization.

- Indeed this is precisely the *gradient descent algorithm*.
- It is it called *gradient descent* - in employing the (negative) gradient as our descent direction - we are repeatedly *descending* in the *(negative) gradient direction* at each step.

- Appreciate the power of this descent direction - which is almost literally given to us - over the zero-order methods detailed in the previous Chapter.
- There we had to *search* to find a descent direction, here calculus provides us not only with a descent direction (without search), but an excellent one to boot.

- The path taken by gradient descent is illustrated figuratively below for a general single-input function.
- Beginning at the point w^0 , we make our first approximation is drawn below as a red dot, with the first order Taylor series approximation drawn in green.
- Moving in the negative gradient descent direction provided by this approximation we arrive at a point $w^1 = w^0 - \alpha \frac{\partial}{\partial w} g\left(w^0\right)$
- We then repeat this process at w^1 , moving in the negative gradient direction there, to $w^2 = w^1 - \alpha \frac{\partial}{\partial w} g\left(w^1\right)$, and so forth.


$g\,(w)$
w^0
w^1
w^2
$\cdots$
w

- Often gradient descent is far better than the zero order approaches discussed in the previous Chapter
- This is entirely due to the fact that the descent direction here - provided by calculus via the gradient - is universally easier to compute.

- Below we provide the generic pseudo-code and ``Python`` implementation of the gradient descent algorithm which will be used in a variety of examples that follow in this Section.

The gradient descent algorithm

1: **input:** function g, steplength α, maximum number of steps K, and initial point $\mathbf{w}^0$

2: `for` $k = 1...K$

3: $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$

4: **output:** history of weights $\left\{\mathbf{w}^k\right\}_{k=0}^{K}$ and corresponding function evaluations $\left\{g\left(\mathbf{w}^k\right)\right\}_{k=0}^{K}$

- How do we set the α parameter in general?
- There are many ways for choosing α for local optimization (as first discussed in Chapter 2) basic (and most commonly used).
- Indeed popular approaches are precisely those introduced in the (comparatively simpler) context of zero order methods in Section 2.3: that is fixed and diminishing steplegnth choices. We explore this idea further in a subsection below.

- When does gradient descent stop?
- Technically (when α is chosen well) the algorithm will *halt near stationary points of a function, typically minima or saddle points*.
- How do we know this? By the very form of the gradient descent step itself.

- Say the step

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

  does not move from the prior point $\mathbf{w}^{k-1}$ significantly.

- Then this can mean only one thing: *that the direction we are traveling in is vanishing* i.e., $-\nabla g\left(\mathbf{w}^{k}\right) \approx \mathbf{0}_{N\times 1}$
- This is - by definition - a *stationary point* of the function.

A generic ``Python`` implementation of the gradient descent algorithm

- Below we implement gradient descent as described above.
- It involves just a few requisite initializations, the computation of the gradient function via e.g., an Automatic Differentiator, and the very simple ``for`` loop.
- The output is a history of the weights and corresponding cost function values at each step of the gradient descent algorithm.

```
# import automatic differentiator to compute gradient module
from autograd import grad

# gradient descent function - inputs: g (input function), alpha (steplength parame
ter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute gradient module using autograd
    gradient = grad(g)

    # run the gradient descent loop
    weight_history = [w]           # container for weight history
    cost_history = [g(w)]          # container for corresponding cost function his
tory
    for k in range(max_its):
        # evaluate the gradient, store current weights and cost function value
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # record weight and cost
        weight_history.append(w)
        cost_history.append(g(w))
    return weight_history,cost_history
```

- Given the input to ***g*** is ***N*** dimensional a general random initialization - the kind that is often used - can be written as shown below.
- Here the function `random.randn` produces samples from a standard Normal distribution with mean zero and unit standard deviation. It is also common to scale such initializations by small constants like e.g., ***0.1***.

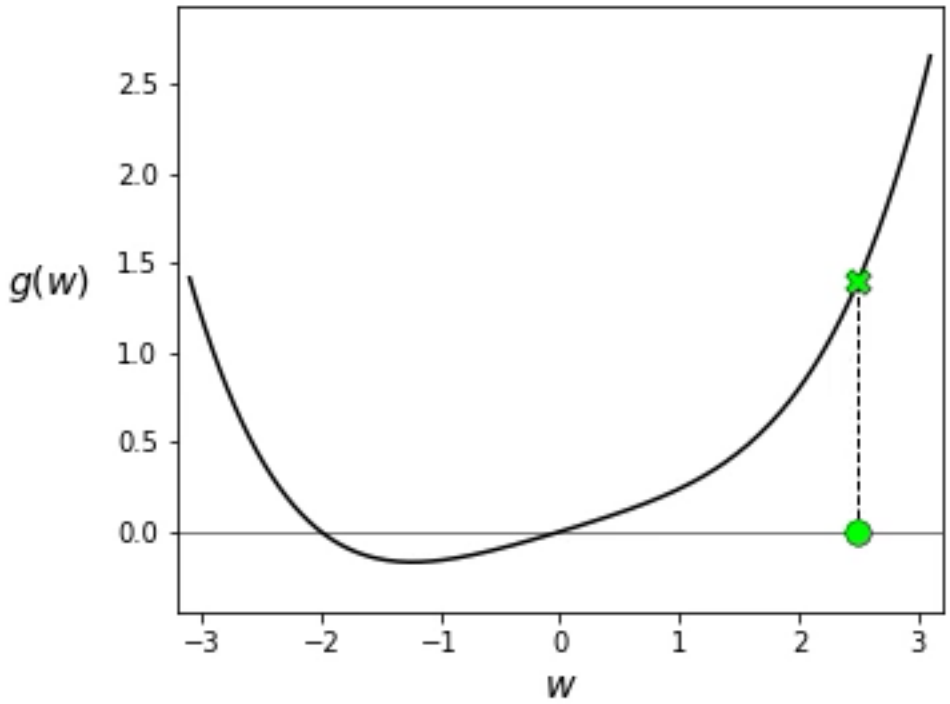
```
# a common initialization scheme - a random point
w = np.random.randn(N,1)
```

Example: A convex single input example

- Below we animate the use of gradient descent to minimize the polynomial function

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right).$$

- Here $w_0 = 2.5$ and $\alpha = 1$

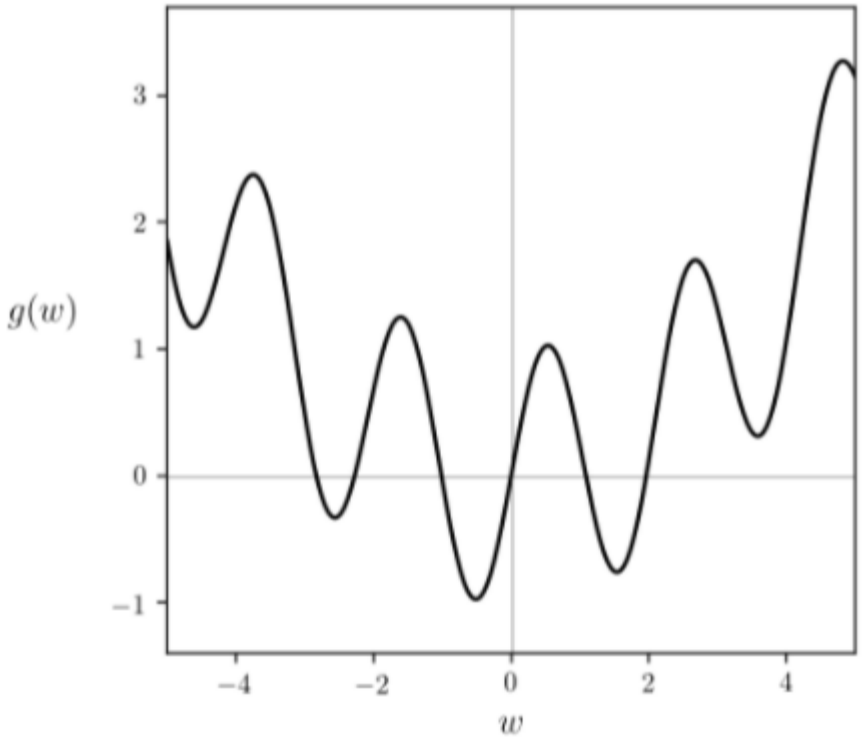

2.5
2.0
1.5
1.0
0.5
0.0
g(w)
−3
−2
−1
0
1
2
3
w

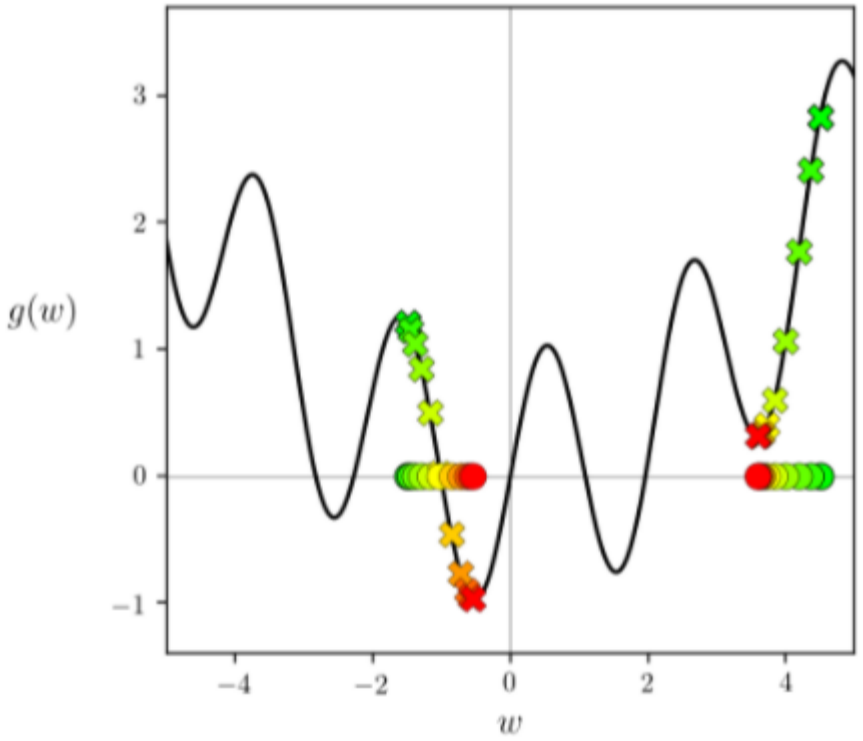
Example: A non-convex single input example

- Now we show the result of running gradient descent several times to minimize the function

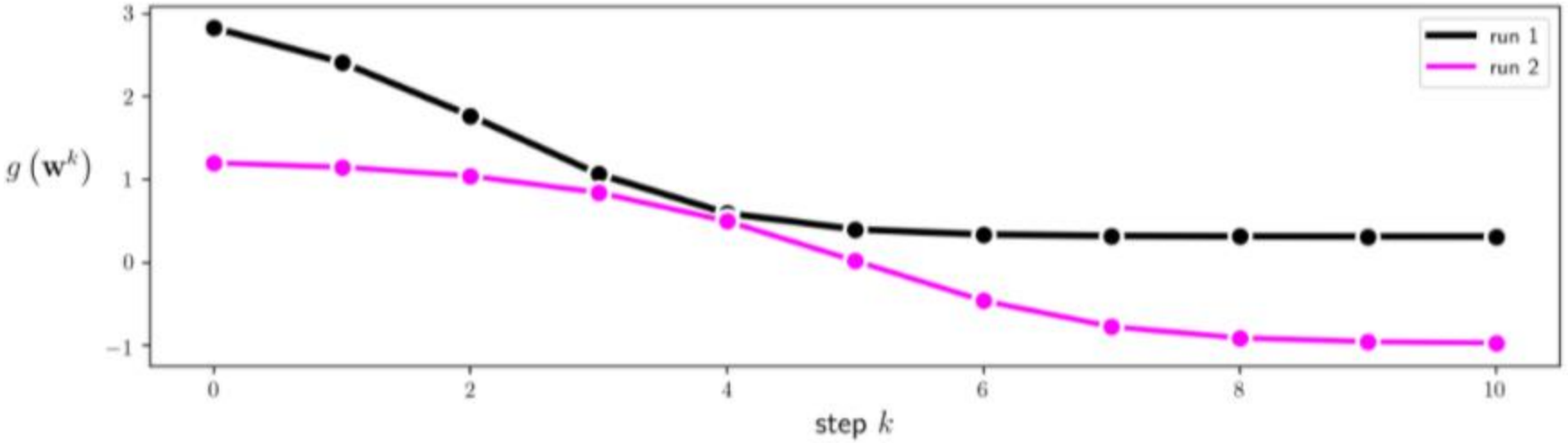
$$g(w) = \sin(3w) + 0.1w^2$$

- For general non-convex functions like this one, several runs (of any local optimization method) can be necessary to determine points near global minima.


$g(w)$
3
2
1
0
−1
−4
−2
0
2
4
w


$g(w)$
3
2
1
0
−1
−4
−2
0
2
4
w

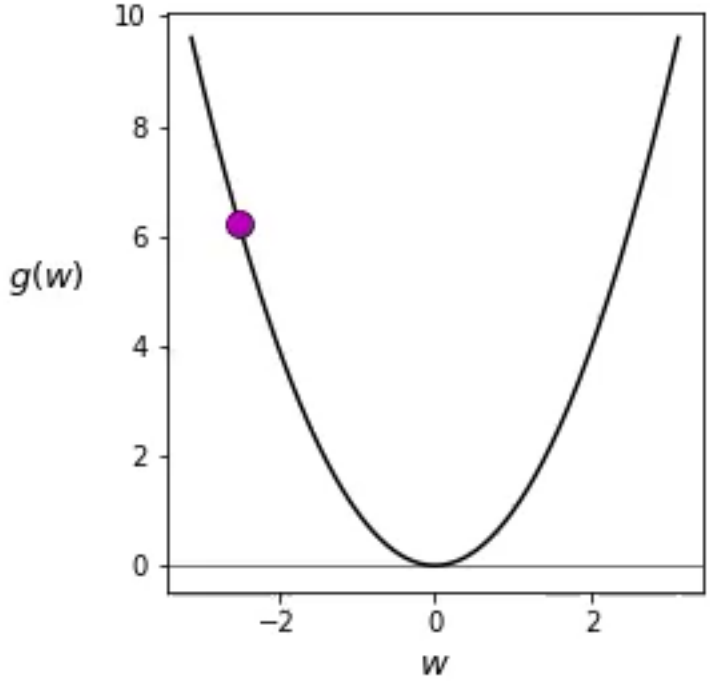
- Viewing the *cost function history plot* allows us to view the progress of gradient descent, regardless of the function's input dimension.

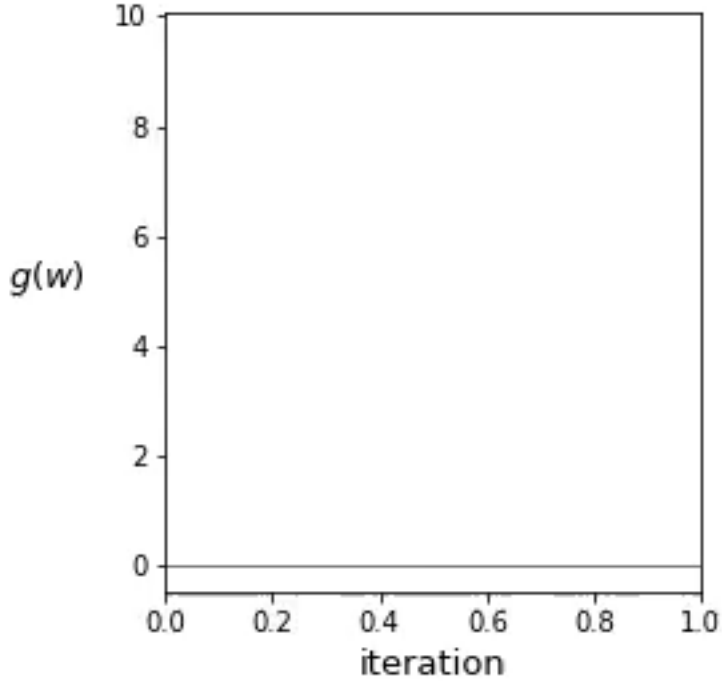

run 1
run 2
$g\left(\mathbf{w}^k\right)$
3
2
1
0
−1
0
2
4
6
8
10
step k

- As discussed in the prior Chapter, these plots are a valuable debugging tool, as well as a valuable tool for selecting proper values for the steplength α

Example: An example of fixed steplength selection for a single input convex function

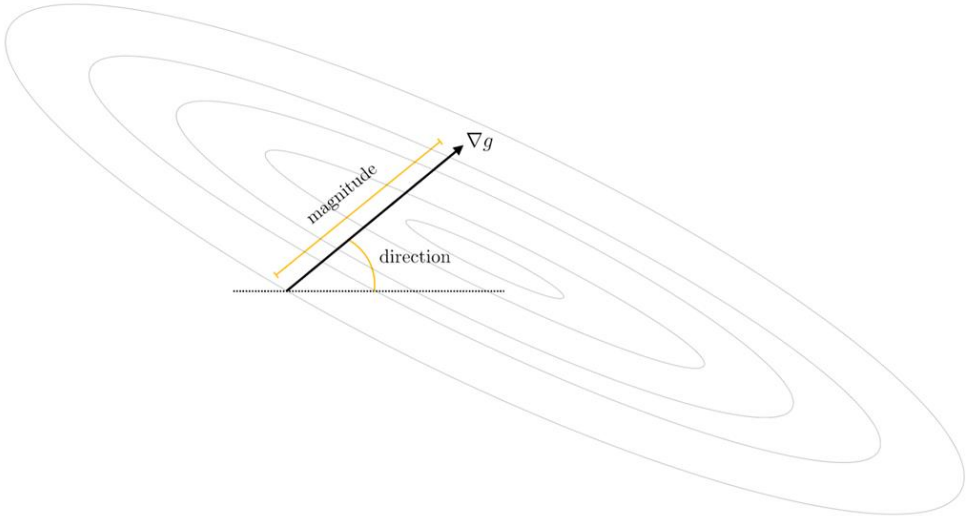
- At each step of gradient descent we *always* have a descent direction - this is defined explicitly by the negative gradient itself.
- However whether or not we descend in the function when taking this step depends completely on how far along it we travel, on our choice of the steplength parameter.
- We illustrate this general principle in the animation below, using ***5*** steps of gradient descent.


10
8
6
4
2
0
$g(w)$
−2
0
2
w


10
8
6
4
2
0
$g(w)$
0.0
0.2
0.4
0.6
0.8
1.0
iteration

3.6 Two issues with the negative gradient as a descent direction

- The negative gradient is not without its weaknesses as a descent direction, and in this Section we outline two significant problems with it that can arise in practice.
- Like any *vector* the negative gradient always consists fundamentally of a *direction* and a *magnitude*.



- Depending on the function being minimized either one of these attributes - or both - can present challenges when using the negative gradient as a descent direction.
- The *direction* of the negative gradient can *rapidly oscillate* or *zig-zag* during a run of gradient descent, often producing *zig-zagging* steps that take considerable time to reach a near mininum point.
- The *magnitude* of the negative gradient can *vanish rapidly* near stationary points, leading gradient descent to slowly crawl near minima and saddle points.

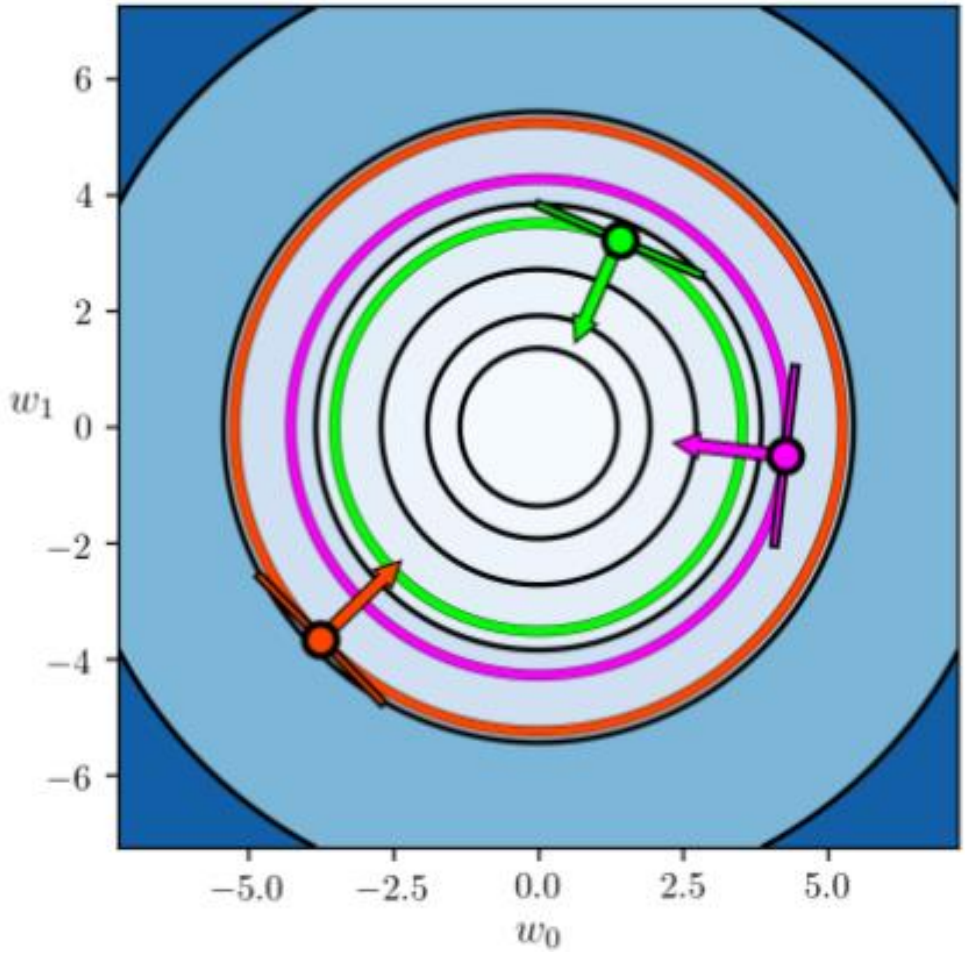
- These two problems present themselves in machine learning because many of the functions we aim to minimize have *long narrow valleys*, long flat areas where the contours of a function become increasingly parallel.

The (negative) gradient direction points perpendicular to the contours of any function

- A fundamental property of the (negative) gradient direction is that it always points perpendicular the contours of a function.
- This statement is universally true - and holds for *any* function and at *all* of its inputs.
- We illustrate this fact via several examples below.

Example: Gradient descent directions on the contour plot of a quadratic function

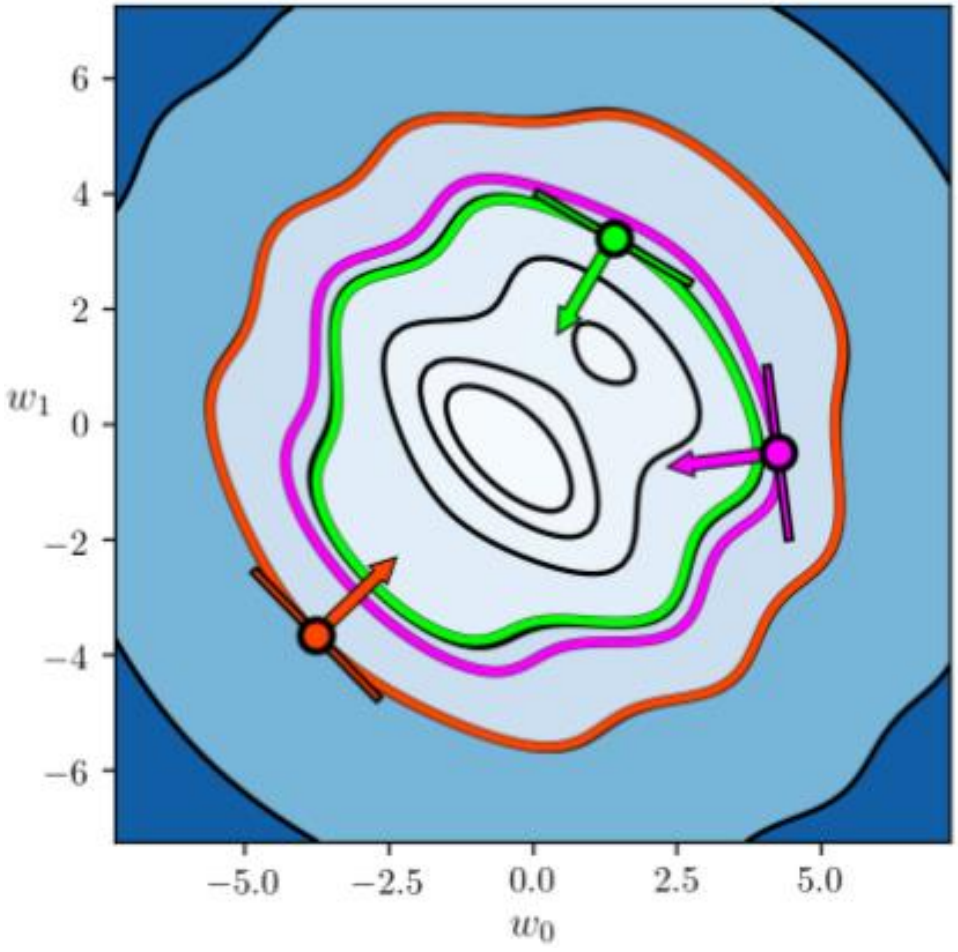
- Below we show $g\left(\mathbf{w}\right) = w_0^2 + w_1^2 + 2$, with gradient descent directions defined at three random points.
- The contour plot is colored *blue* - with darker regions indicating where the function takes on larger values, and lighter regions where it takes on lower values.
- Each of the points we choose are highlighted in a unique color, with the contour on which they sit on the function colored in the same manner.
- The *descent* direction defined by the gradient is perpendicular at each point is drawn as an arrow and the tangent line to the contour at each input is also drawn.


6
4
2
0
−2
−4
−6
w_1
−5.0
−2.5
0.0
2.5
5.0
w_0

Example: Gradient descent directions on the contour plot of a wavy function

- Here we show the contour plot and gradient descent directions in the same manner as the previous example for

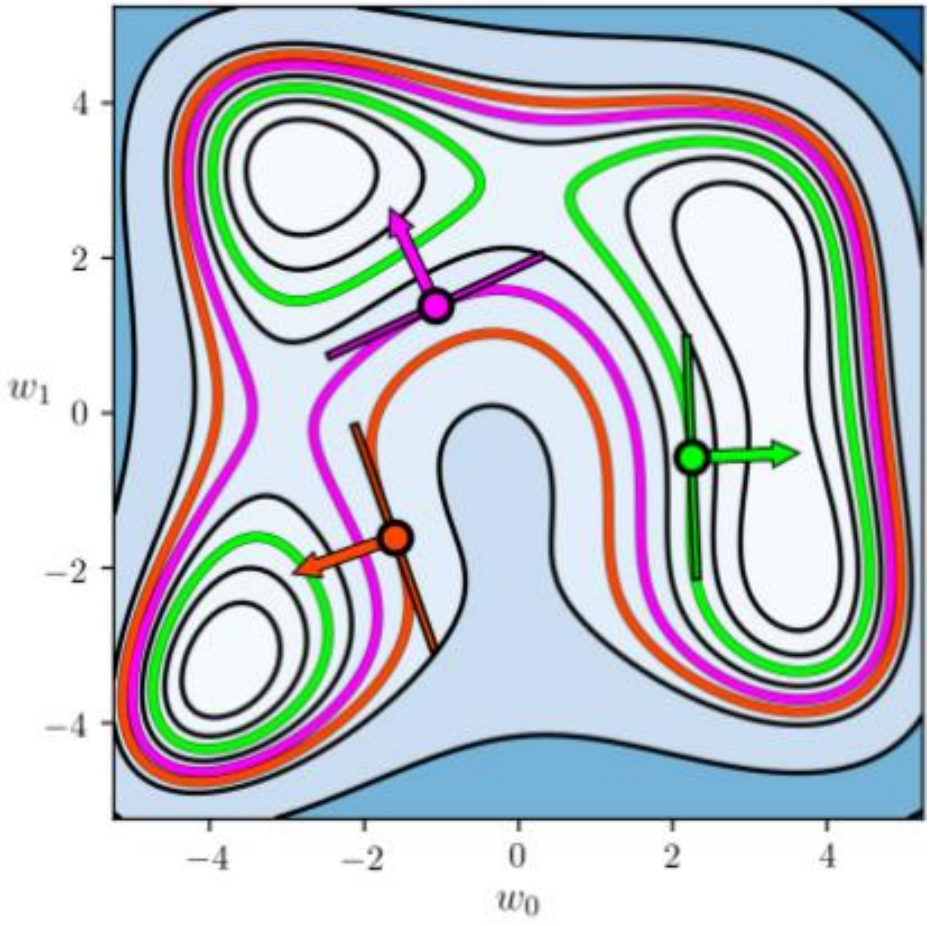
$$g\left(\mathbf{w}\right) = w_0^2 + w_1^2 + 2\sin\left(1.5\left(w_0 + w_1\right)\right)^2 + 2.$$


6
4
2
0
−2
−4
−6
w_1
−5.0
−2.5
0.0
2.5
5.0
w_0

Example: Gradient descent directions on the contour plot of a standard non-convex test function

- Finally we show the same sort of plot as in the previous example using the function

$$g\left(\mathbf{w}\right) = \left(w_0^2 + w_1 - 11\right)^2 + \left(w_0 + w_1^2 - 6\right)^2$$

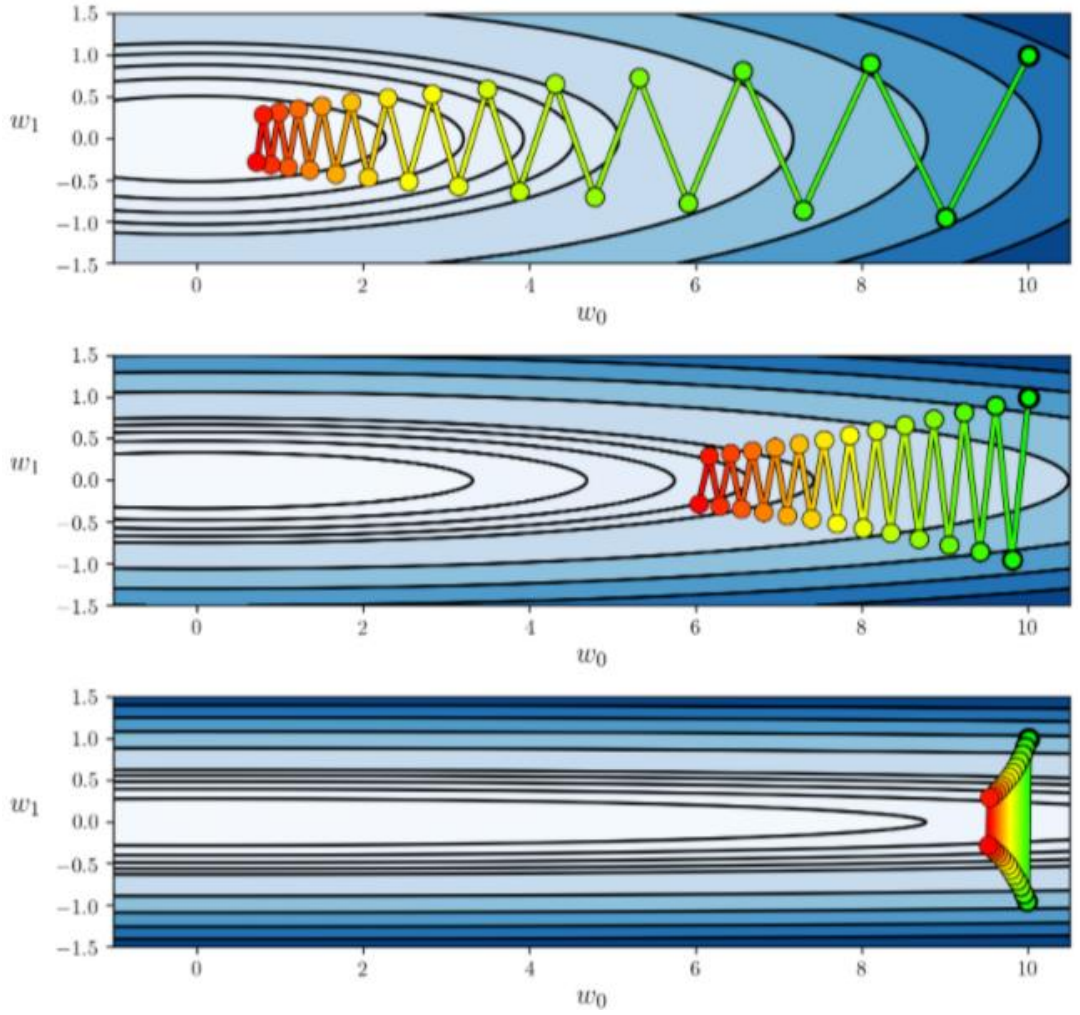

4
2
0
−2
−4
w_1
−4
−2
0
2
4
w_0

The 'zig-zagging' behavior of gradient descent

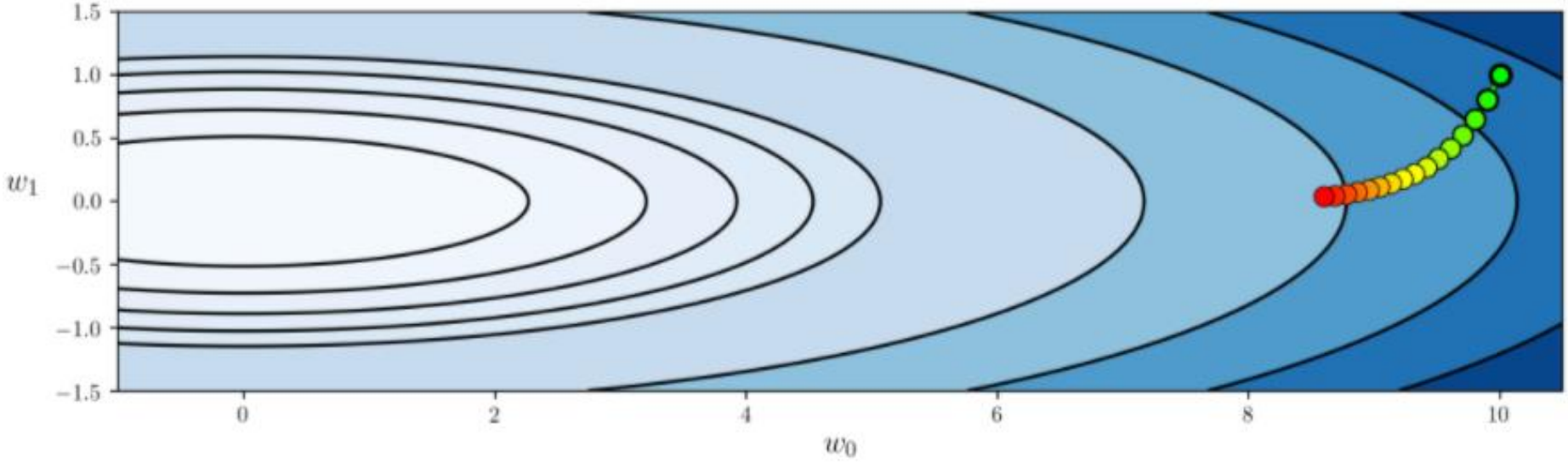
- Because the negative gradient *points perpendicular to the contours of a function* can make the negative gradient direction *oscillate rapidly* or *zig-zag* during a run of gradient descent.
- This in turn can cause zig-zagging behavior in the gradient descent steps themselves.
- *Too much* zig-zagging slows minimization progress and - when it occurs - many gradient descent steps are required to adequately minimize a function.

Example: Zig-zagging behavior of gradient descent on three simple quadratic functions

- We illustrate the zig-zag behavior of gradient descent with three dimensional quadratic $N = 2$ $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$
- .Not much progress is made with the third quadratic at all due to the large amount of zig-zagging.
- We can also see the cause of this zig-zagging: the negative gradient direction constantly points perpindicular to the contours of the function (this can be especially seen in the third case).


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
w_1
0
2
4
6
8
10
w_0

- It is the true that we can ameilorate this zig-zagging behavior by *reducing the steplength value*, as shown below.
- However this does not solve the underlying problem that zig-zagging produces - which is slow convergence.
- Typically in order to ameliorate or even eliminate zig-zagging this way requires a very small steplength, which leads back to the fundamental problem of slow convergence.


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
w_1
0
2
4
6
8
10
w_0

The slow-crawling behavior of gradient descent

- The *first order condition for optimality* discussed in [Section 3.2](https://jermwatt.github.io/machine_learning_refined/notes/3_First_order_methods/3_2_First.html), the (negative) gradient vanishes at stationary points.
- The vanishing behavior of the negative gradient magnitude near stationary points has a natural consequence for gradient descent steps - they progress very slowly, or 'crawl', near stationary points.
- This occurs because *unlike* the zero order methods discussed in the previous Chapter, *the distance traveled during each step of gradient descent is not completely determined by the steplength / learning rate value* α.

- This means that gradient descent steps make little progress towards minimization when near a stationary point
- Thus depending on the function many of them may be required to complete minimization.

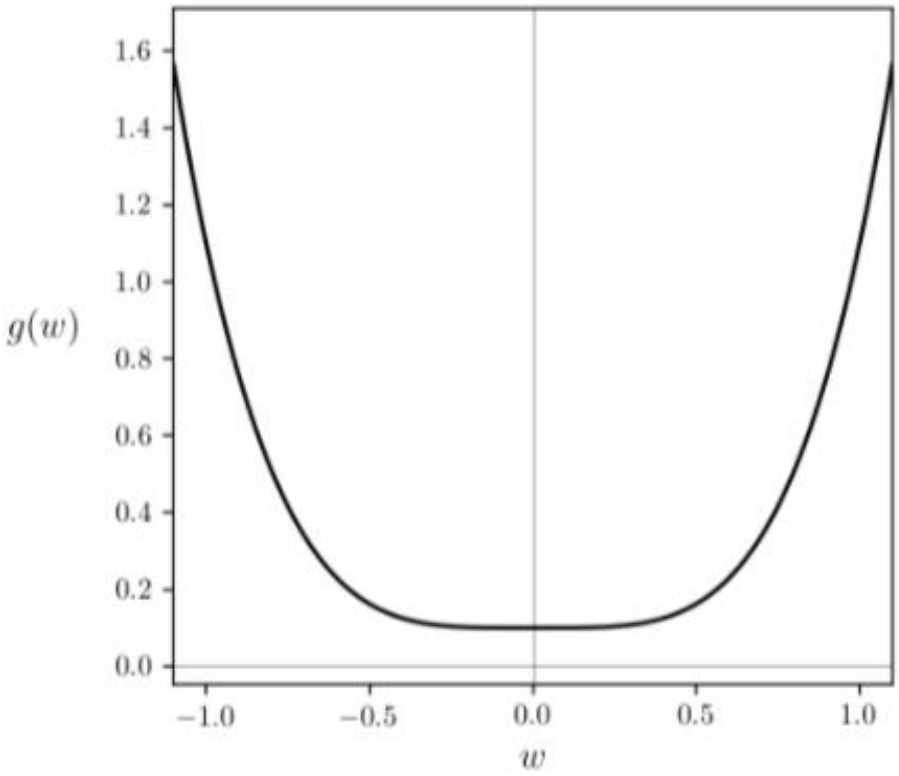
Example: Slow-crawling behavior of gradient descent near the minimum of a function

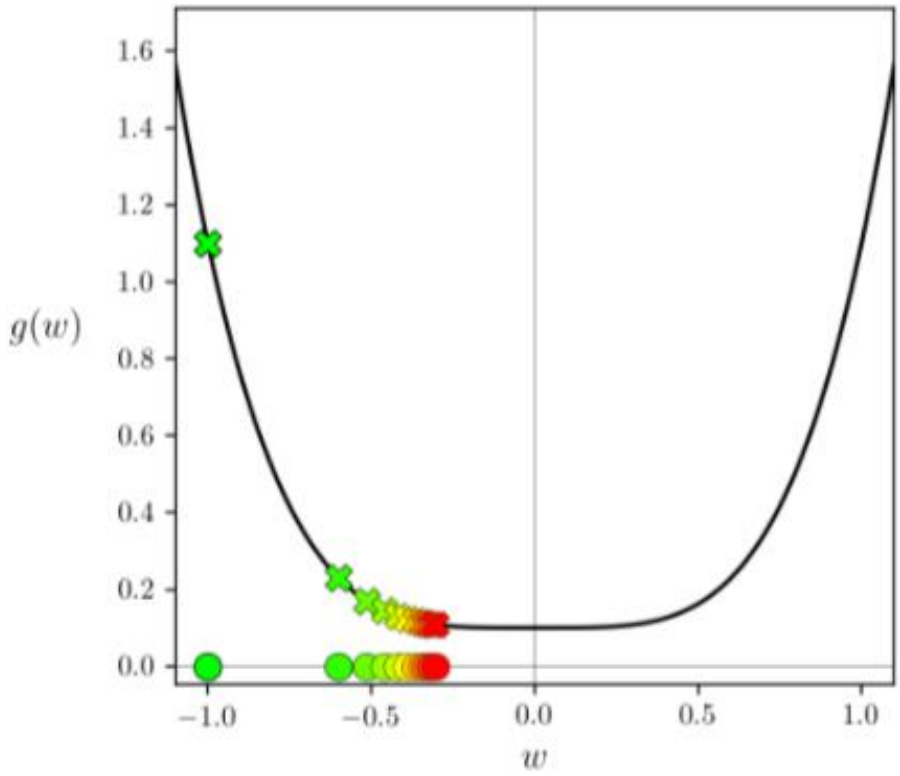
- Below we show another example run of gradient descent using a function

$$g(w) = w^4 + 0.1$$

  whose minimum is at the origin.

- This example shows how steps can be quite large far from a stationary point, but then get very small and crawls as we get closer and closer to the minimum of this function.


1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
$g(w)$
−1.0
−0.5
0.0
0.5
1.0
w

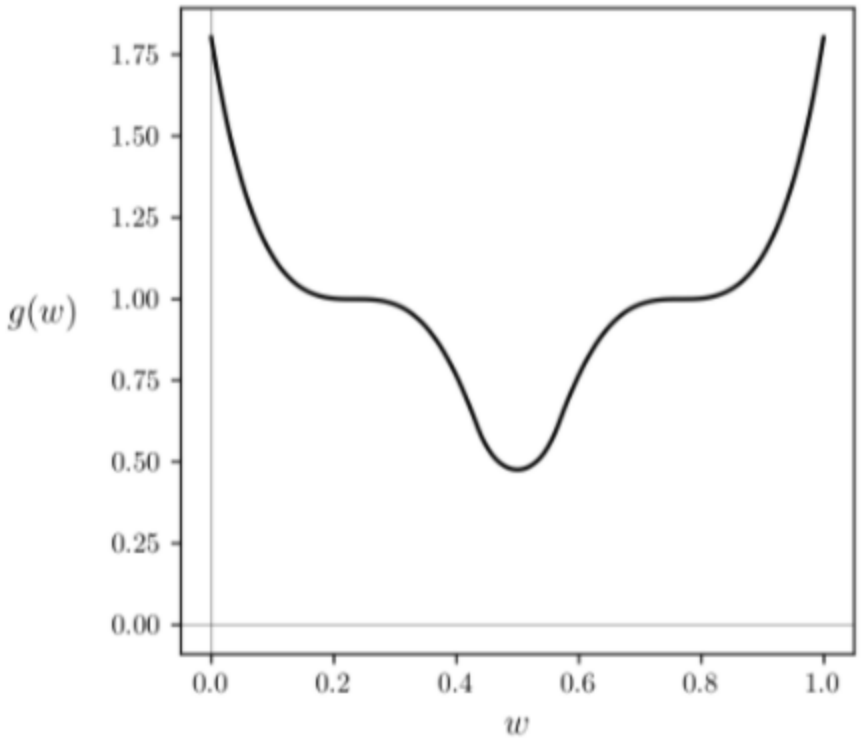

1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
$g(w)$
−1.0
−0.5
0.0
0.5
1.0
w

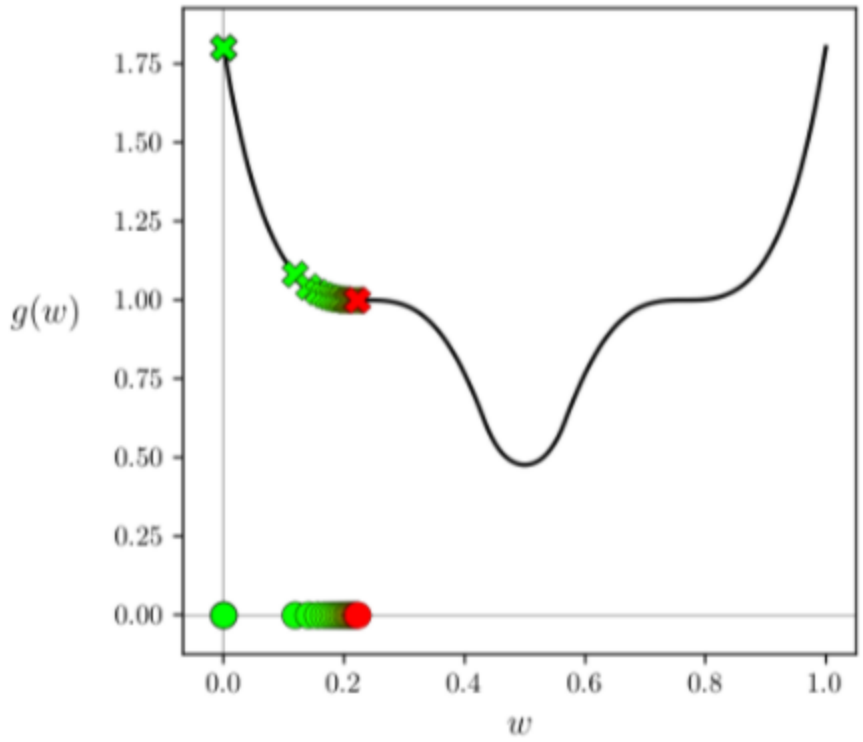
Example: Slow-crawling behavior of gradient descent near saddle points

- Now we illustrate the crawling issue of gradient descent near saddle points using the non-convex function

$$g(w) = \text{maximum}(0, (3w - 2.3)^3 + 1)^2 + \text{maximum}(0, (-3w + 0.7)^3 + 1)^2$$

- This function has a minimum at $w = \frac{1}{2}$ and saddle points at $w = \frac{7}{30}$ and $w = \frac{23}{30}$
- The fact that gradient descent crawls as it approaches this saddle point since the magnitude of the gradient vanishes here.


$g(w)$
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0
w


$g(w)$
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0
w

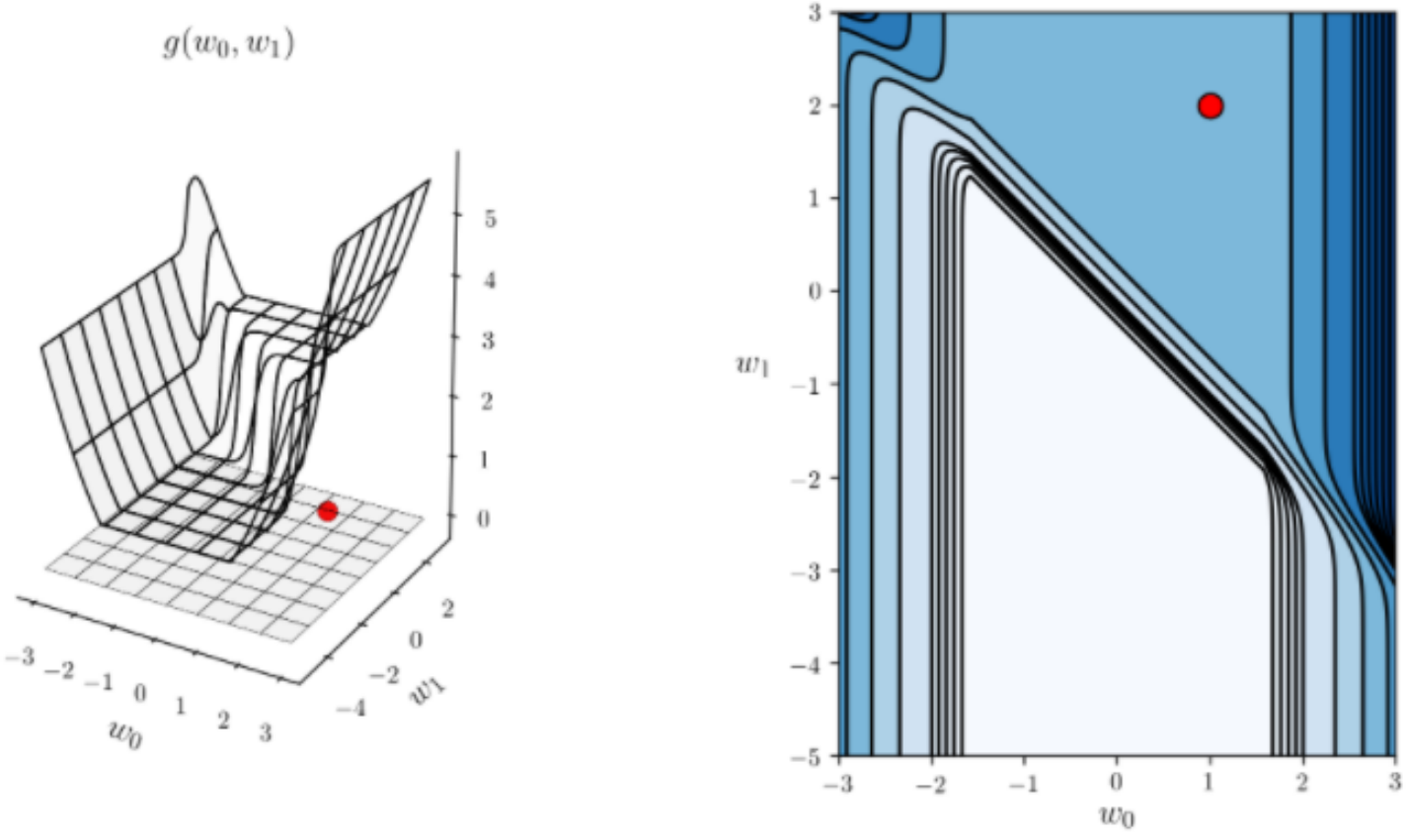
Example: Slow-crawling behavior of gradient descent in large flat regions of a function

- As another example, we attempt to minimize the function

$$g(w_0, w_1) = \tanh(4w_0 + 4w_1) + \max(1, 0.4w_0^2) + 1$$

  via gradient descent starting at the point $\mathbf{w}^0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

- The magnitude of the gradient being almost zero here, we cannot make much progress employing ***1000*** steps of gradient descent.


$g(w_0, w_1)$
5
4
3
2
1
0
−3 −2 −1 0 1 2 3
w_0
2 0 −2 −4
w_1
3
2
1
0
−1
−2
−3
−4
−5
w_1
−3 −2 −1 0 1 2 3
w_0