2.1 Introduction

- The problem of determining the smallest (or largest) value a function can take, referred to as its *global minimum* (or *global maximum*) is a centuries old pursuit that has numerous applications throughout the sciences and engineering.
- In this Chapter we begin our investigation of mathematical optimization by describing the *zero order optimization* techniques (also called *Hessian free optimization*)

- While not always the most powerful optimization tools at our disposal, these techniques are quite simple and often quite effective.
- Discussing zero order methods first also allows us to lay bear, in a simple setting, a range of crucial concepts we will see throughout the Chapters that follow in more complex settings.
- These concepts include the notions of *optimality*, *local optimization*, *descent directions*, *steplengths*, and more.

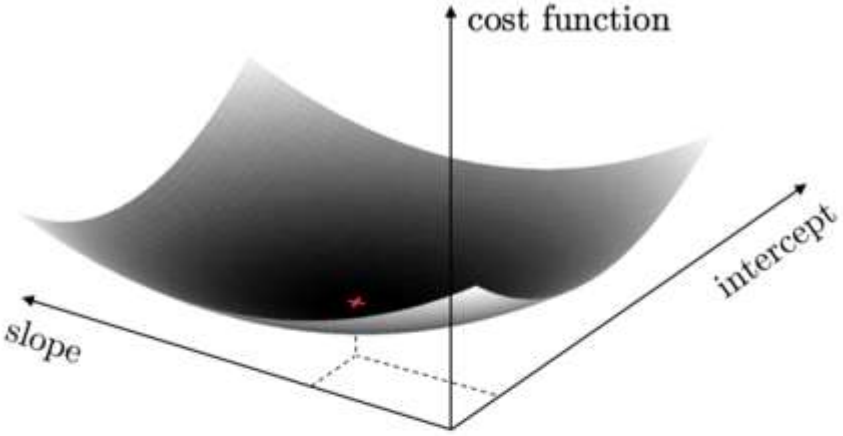
Visualizing minima and maxima

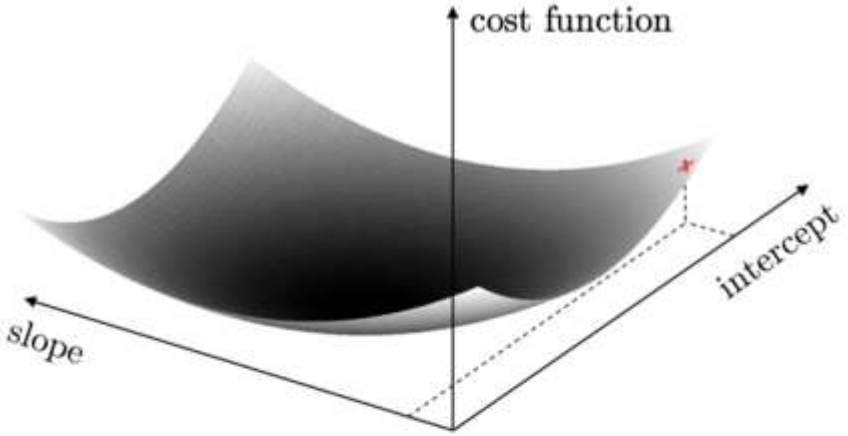
- When a function takes in only one or two inputs we can visually identify its minima or maxima by plotting it over a large swath of its input space.
- But what if a function takes in more than two inputs?
- We begin our discussion by first examining a number of low dimensional examples to gain an intuitive feel for how we might effectively identify these desired minima or maxima in general.

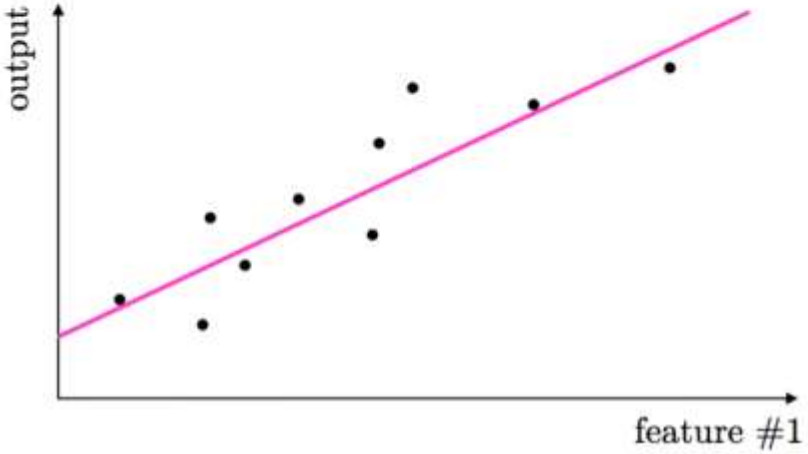
Example: Visual inspection of simple functions for minima and maxima

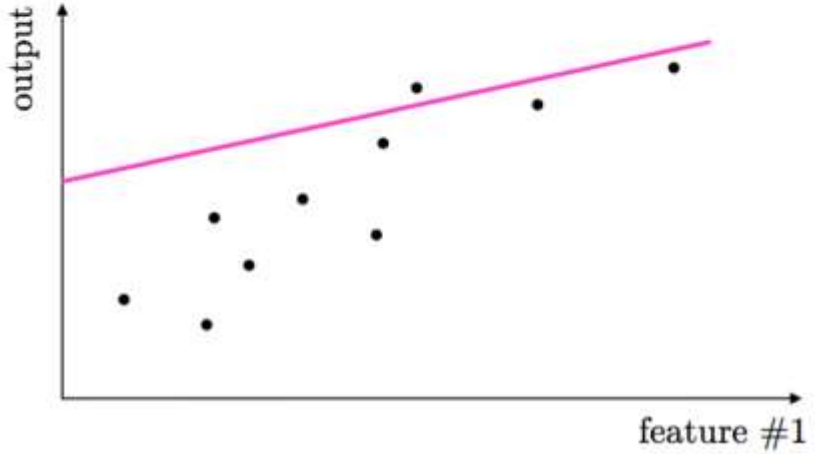
- Every machine learning problem has parameters that must be tuned properly to ensure optimal learning
- For example, there are two parameters that must be properly tuned in the case of a simple linear regression.
- That is, when fitting a line to a scatter of data: the slope and intercept of the linear model.
- These two parameters are tuned by forming what is called a *cost function* or *loss function*.

- This is a continuous function in both parameters that measures how well the linear model fits a dataset given a value for its slope and intercept.
- The proper tuning of these parameters via the cost function corresponds geometrically to finding the values for the parameters that make the cost function as small as possible
- Or, in other words, the parameters that minimize the cost function.
- The image below illustrates how choosing a set of parameters higher on the cost function results in a corresponding line fit that is poorer than the one corresponding to parameters at the lowest point on the cost surface.

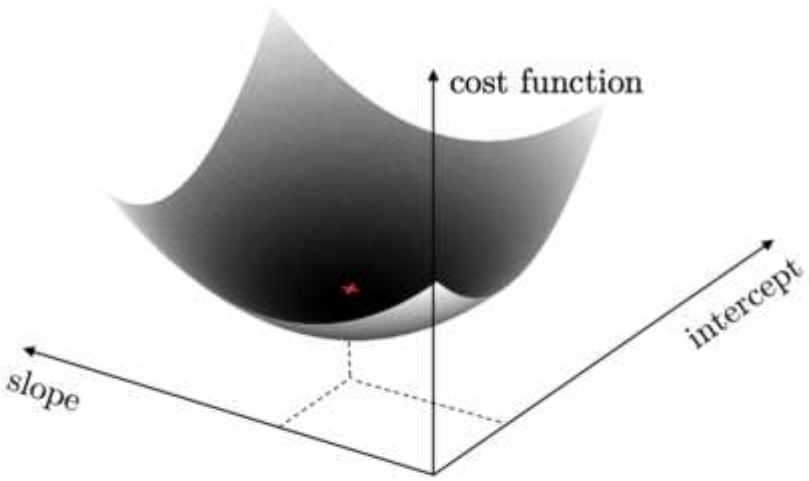

cost function
slope
intercept

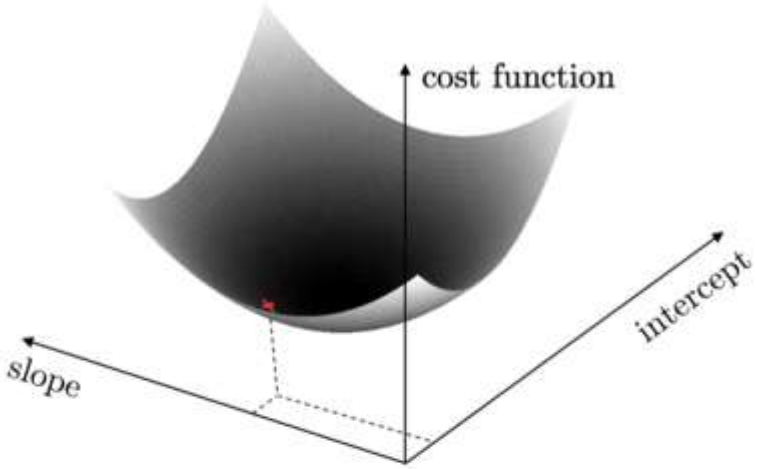

cost function
slope
intercept

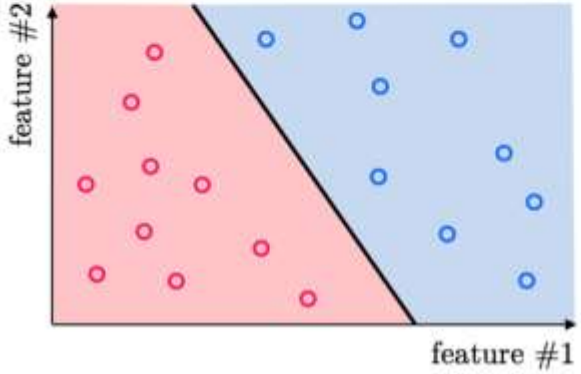

output
feature #1

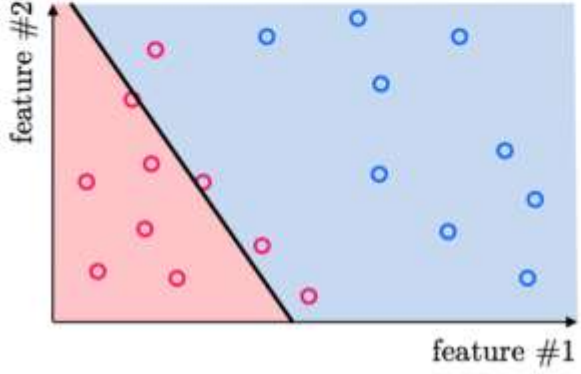

output
feature #1

- This same idea holds true for regression with higher dimensional input, as well as classification where we must properly tune parameters to *separate* classes of data.
- Again, the parameters minimizing an associated cost function provide the best classification result. This is illustrated for classification below.


cost function
slope
intercept


cost function
slope
intercept


feature #2
feature #1


feature #2
feature #1

- The tuning of these parameters require the *minimization of a cost function* can be formally written as follows.
- For a generic function $g(\mathbf{w})$ taking in a general ***N*** dimensional input $\mathbf{w}$ the problem of finding the particular point $\mathbf{v}$ where ***g*** attains its smallest value is written formally as

$$\underset{\mathbf{w}}{\text{minimize}} \quad g\left(\mathbf{w}\right)$$

- This formal problem can very rarely be solved 'by hand', instead we must rely on algorithmic techniques for finding function minima (or at the very least finding points close to them).
- In this part of the text we examine many algorithmic methods of *mathematical optimization*, which aim to do just this.

2.2 The Zero-Order Condition for Optimality

- In this Section we describe most basic mathematical definition of a function's minima - called the *zero-order condition*.

The zero-order definitions

- In many areas of science and engineering one is interested in finding the smallest points - or the global minima - of a particular function.
- For a function $g(\mathbf{w})$ taking in a general ***N*** dimensional input $\mathbf{w}$ this problem is formally phrased as

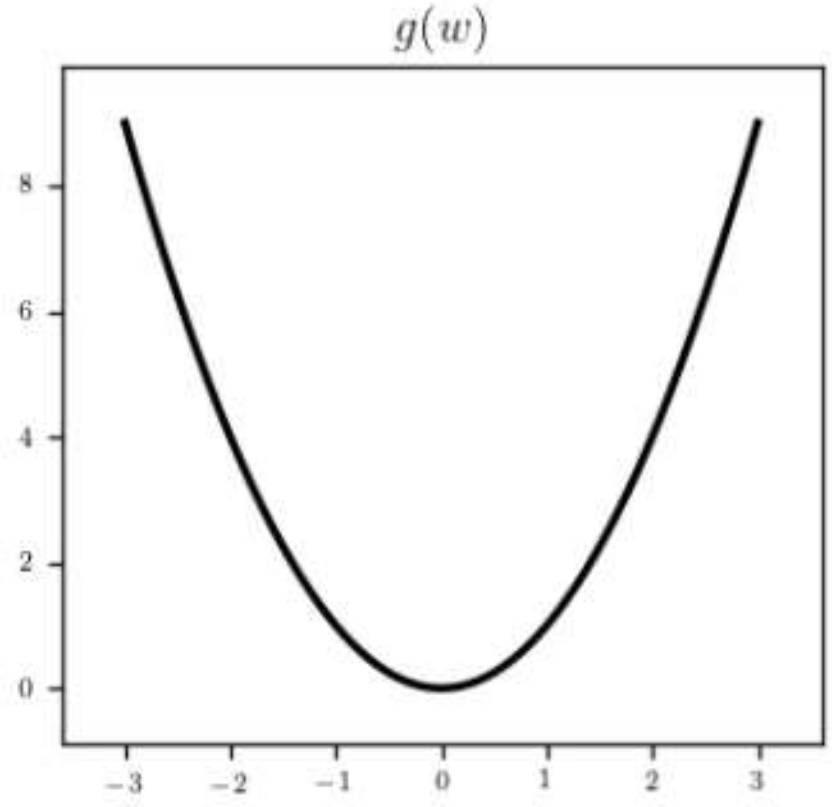
$$\underset{\mathbf{w}}{\text{minimize}} \quad g\left(\mathbf{w}\right)$$

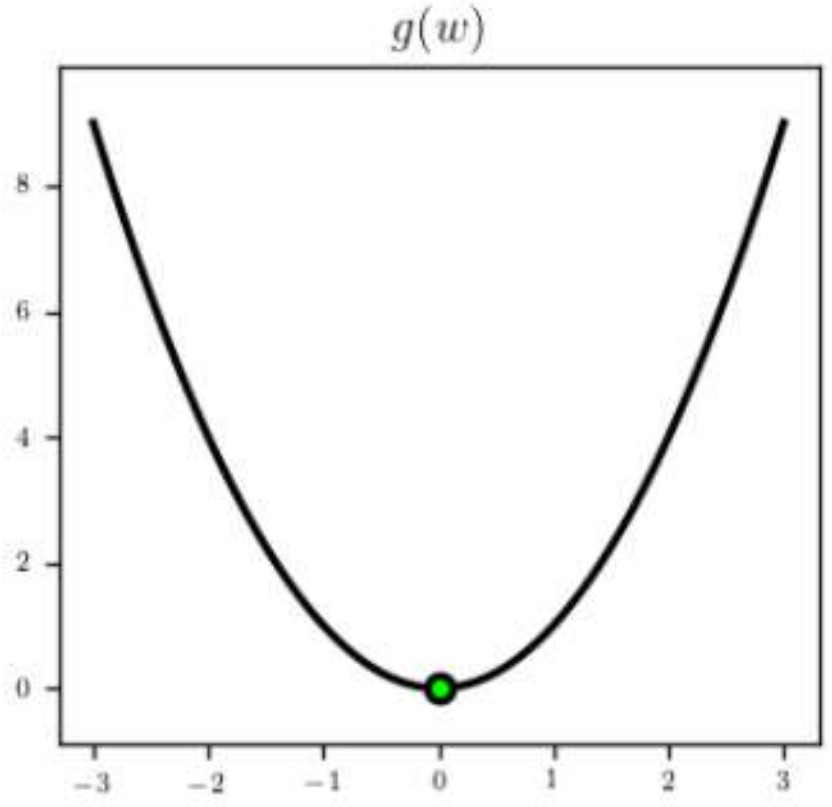
- This says formally: look over every possible input $\mathbf{w}$ and find the one that gives the smallest value of $g(\mathbf{w})$.

- Lets first visually examine a few simple examples.
- In each case keep in mind how you would describe a minimum point - in plain English.
- After each picture we will then codify this intuitive understanding - forming our first formal definition of minimum points.

Example: Global minima of a quadratic

- Below we plot the simple quadratic $g(w) = w^2$ over a short region of its input space.
- Examining the left panel below, what can we say defines the smallest value(s) of the function here?


$g(w)$
8
6
4
2
0
−3
−2
−1
0
1
2
3


$g(w)$
8
6
4
2
0
−3
−2
−1
0
1
2
3

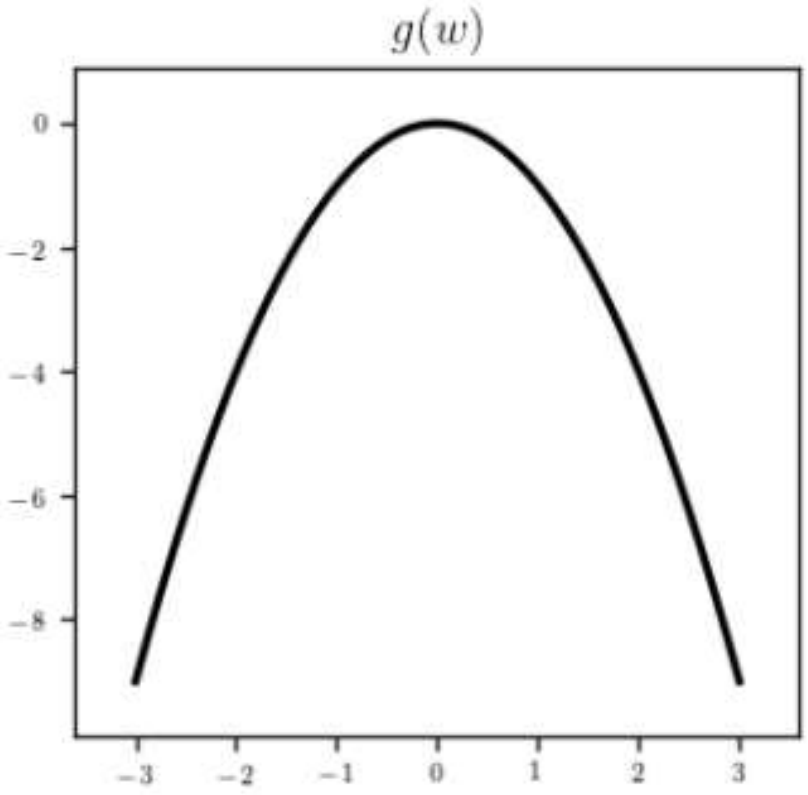
- Certainly the minimum is smaller than any other point on the function.
- Specifically the smallest value - the global minimum of this function - seemingly occurs close to $w^\star = 0$(marked as a green dot $(0, g(0))$ in the right panel).
- Formally a point $w^\star$ gives the smallest point on the function if

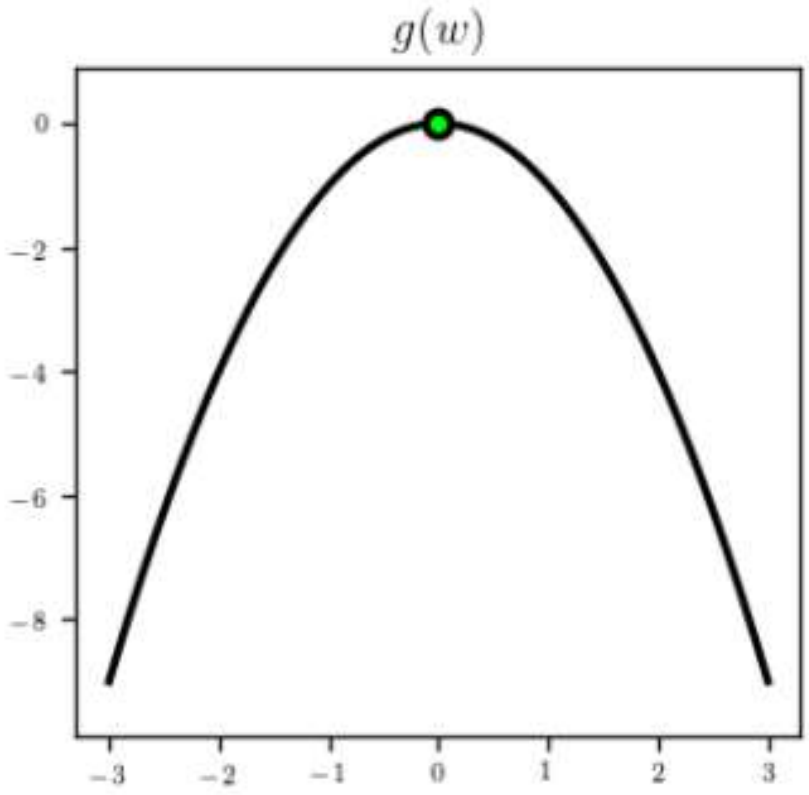
$$g(w^\star) \leq g(w) \text{ for all } w.$$

- This formal translation of what we know to be intuitively true is called the *zero-order definition of a global minimum point*.

Example: Global maxima of a quadratic

- Remember what happens if we multiply the quadratic function in the previous example by -1, as plotted below.
- The function flips upside down - now its global minima lie at $w^\star = \pm\infty$
- Now the point $w^\star = 0$ that used to be a *global minimum* is a *global maximum* - i.e., the largest point on the function (and is marked in green in the right panel below).


$g(w)$
0
−2
−4
−6
−8
−3
−2
−1
0
1
2
3


$g(w)$
0
−2
−4
−6
−8
−3
−2
−1
0
1
2
3

- How do we formally define a *global maximum*?
- Just like the global minimum in the previous example - it is the point that is larger than any other on the function i.e.,

$$g(w^\star) \geq g(w) \text{ for all } w.$$

- This direct translation of what we know to be intuitively true into mathematics is called the *zero-order definition of a global maximum point*.

These concepts of *minima* and *maxima* of a function are always related to each via multiplication by ***-1***.

That is, any point that is a minima of a function ***g*** is a maxima of the function ***-g***, and vice-versa.

- To express our pursuit of a global *maxima* of a function we write

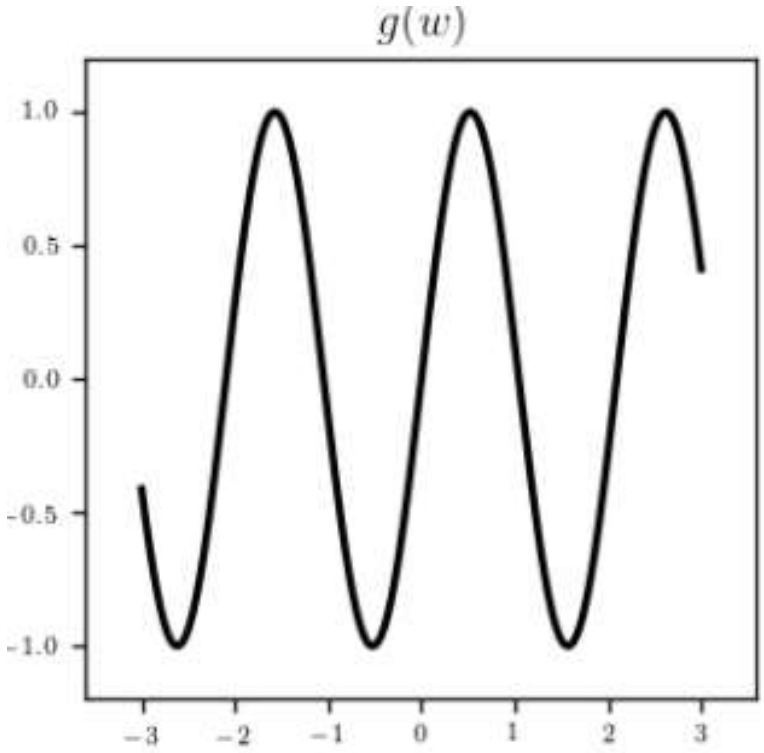
$$\underset{\mathbf{w}}{\text{maximize}} \quad g\left(\mathbf{w}\right).$$

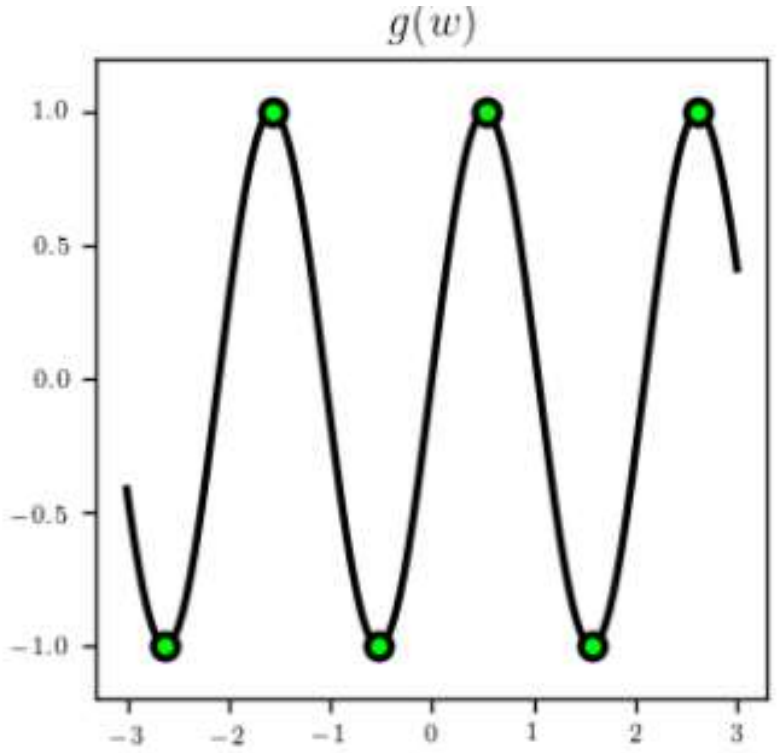
- But since minima and maxima are so related, we can always express this in terms of our minimize notation as

$$\underset{\mathbf{w}}{\text{maximize}} \quad g\left(\mathbf{w}\right) = -\underset{\mathbf{w}}{\text{minimize}} \quad g\left(\mathbf{w}\right).$$

Example: Global minima/maxima of a sinusoid

- Let us look at the sinusoid function $g(w) = \sin(2w)$ plotted below.
- Here we can clearly see that - over the range we have plotted the function - that there are two global minima and two global maxima (marked by green dots in the right panel).


$g(w)$
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3


$g(w)$
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3

- We can imagine as well of course that there are further minima and maxima (here one exists at every $4k+3$ multiple of $\frac{\pi}{2}$ for integer k's).
- This is just an example where - technically speaking - the function has an infinite number of global minima and maxima.

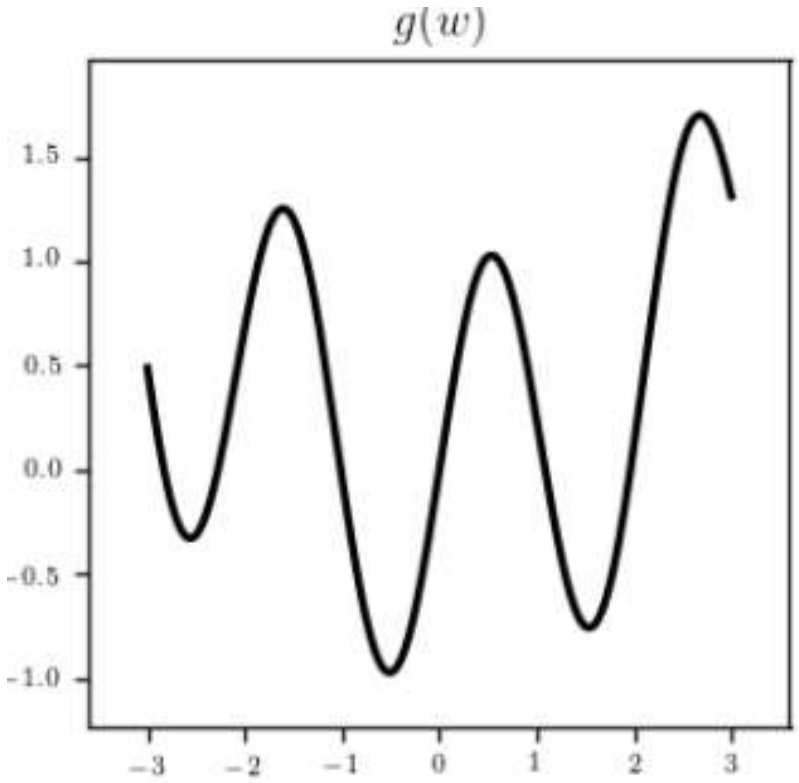
Example: Minima and maxima of the sum of a sinusoid and a quadratic

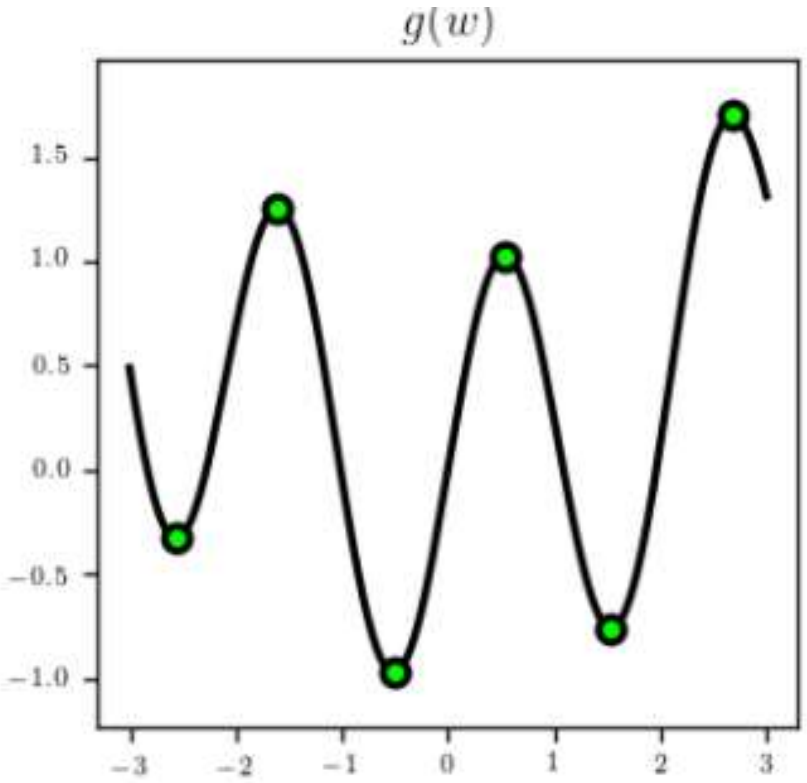
- Let’s look at a weighted sum of the previous two examples, the function

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

  over a short region of its input space.

- Examining the left panel below, what can we say defines the smallest value(s) of the function here?


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


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

- Here we have a global minimum around $w^\star = -0.5$ and a global maximum around $w^\star = 2.7$
- We also have minima and maxima that are *locally optimal* - for example the point around $w^\star = 0.8$ is a local maximum.
- Likewise the point near $w^\star = 1.5$ is a *local minimum* - since it is the smallest point on the function nearby.

- We can formally say that the point $w^\star$ is a local minimum of the function $g(w^\star)$ as

$$g(w^\star) \leq g(w) \text{ for all } w \text{ near } w^\star$$

- The statement $\text{for all } w \text{ near } w^\star$ is relative, and simply describes the fact that the point $w^\star$ is smaller than its neighboring points.

- This is the *zero-order definition of local minima*.
- The same formal definition can be made for local maximum points as well, switching the $\leq$ sign to $\geq$, just as in the case of global minima/maxima.

The zero-order condition for optimality

- From these examples we have seen how to formally define global minima/maxima as well as the local minima/maxima of functions we can visualize.
- These formal definitions directly generalize to a function of any dimension - in general taking in ***N*** inputs.
- Packaged together these zero-order conditions are often referred to as *the zero-order condition for optimality*.

The zero order condition for optimality: A point $\mathbf{w}^\star$ is

- - a global minimum of $g(\mathbf{w})$ if and only if $g(\mathbf{w}^\star) \leq g(\mathbf{w})$ for all $\mathbf{w}$
- - a global maximum of $g(\mathbf{w})$ if and only if $g(\mathbf{w}^\star) \geq g(\mathbf{w})$ for all $\mathbf{w}$
- - a local minimum of $g(\mathbf{w})$ if and only if $g(\mathbf{w}^\star) \leq g(\mathbf{w})$ for all $\mathbf{w}$ near $\mathbf{w}^\star$
- - a local maximum of $g(\mathbf{w})$ if and only if $g(\mathbf{w}^\star) \geq g(\mathbf{w})$ for all $\mathbf{w}$ near $\mathbf{w}^\star$

The zero-order jargon, in context

- Here we have seen zero-order definitions of various important points of a function.
- Why are these called *zero-order* conditions? Because each of their definitions involves only a function itself - and nothing else.

- This bit of jargon - 'zero-order'- is commonly used when discussing a mathematical function in the context of calculus where we can have *first-order*, *second-order*, etc., derivatives e.g., a zero-order derivative of a function is just the function itself.
- In the future we will see *higher order* definitions of optimal points e.g., *first-order* definitions that involve the first derivative of a function and *second-order* definitions involving a function's second derivative.

2.3 Global optimization methods

- In this Section we describe the first approach one might take to approximately minimize an arbitrary function: evaluate the function using a large number of input points and treat the input that provides the lowest function value as the approximate global minimum of the function.
- This idea mimics how we as humans might find the approximate minimum of a function 'by eye' - i.e., by drawing it and visually identifying its lowest point.

- While easy to implement and perfectly adequate for functions having low-dimensional input, this naturally zero-order framework fails miserably when the input dimension of a function grows to even moderate size.
- This happens essentially for the same reason our ability to visually identify an approximate minimum of a function fails once the input size is greater than 2 or 3: the number of inputs to examine simply becomes infeasible.

Choosing input points

- To determine the smallest point of a function taking in one or two inputs we - as humans - can simply draw the function and determine 'by eye' where the smallest point lies.
- We can mimic this visual approach by simply evaluating a function over a large number of its input points and designating the input that provides the smallest result as our approximate global minimum.
- This approach is called a *global optimization method*.
- How do we choose the inputs to try out with a generic function?

- We clearly cannot try them all - even for a single-input function - since there are (technically speaking) an infinite number of points to try for any continuous function.
- So - as one might guess - we can take two approaches to choosing our (finite) set of input points to test: we can sample them *uniformly* over an evenly spaced grid, or pick the same number of input points at random.
- We illustrate both choices in the example below.

Example: Evaluating a quadratic to determine its minimum

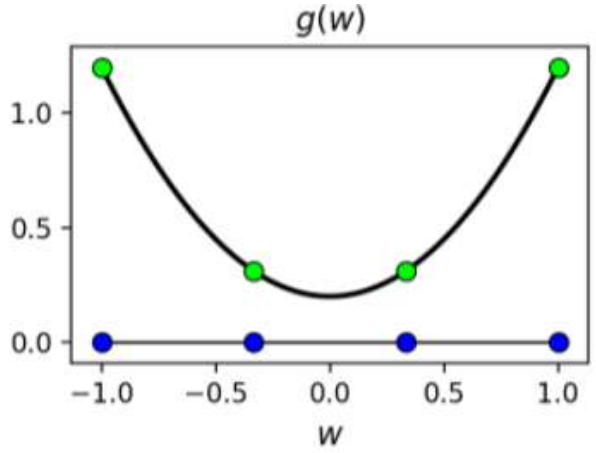
- Here we illustrate two sampling methods for finding the global minimum of simple 2-d and 3-d quadratic functions

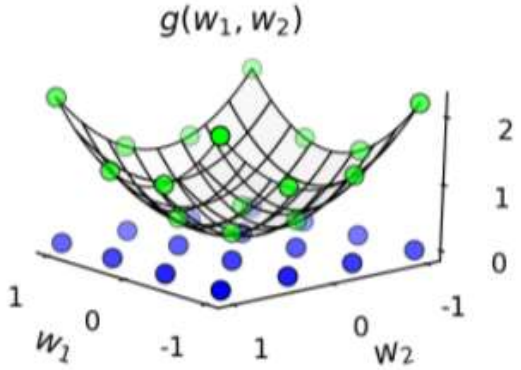
$$g(w) = w^2 + 0.2$$
$$g(w_1, w_2) = w_1^2 + w_2^2 + 0.2$$

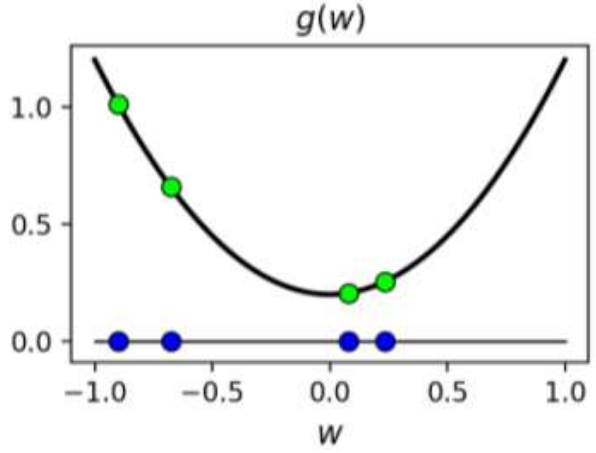
- The former has global minimum at $w = 0$, and the latter at

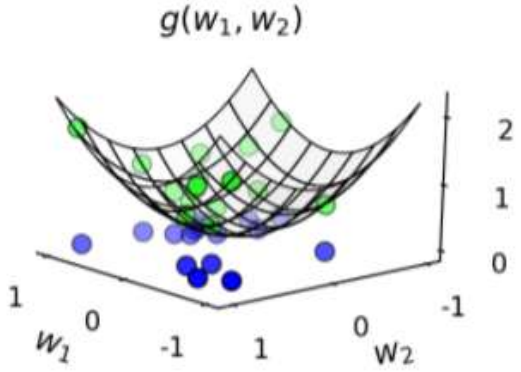
$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

- In the top two panels we illustrate how to *evenly sample* the input of each function
- In the bottom two panels we randomly sample (uniformly at random on the interval [-1,1] in each input axis)

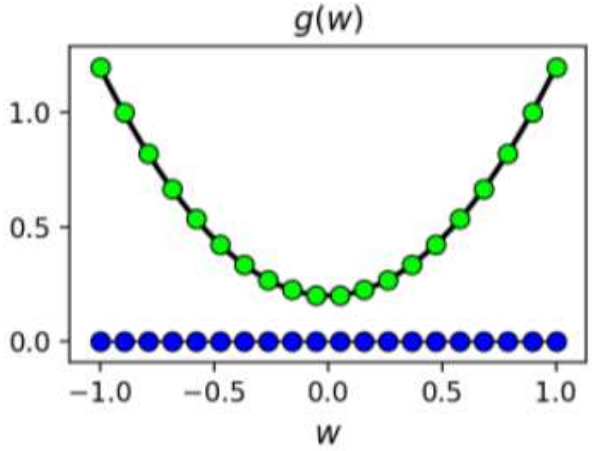

g(w)
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
w

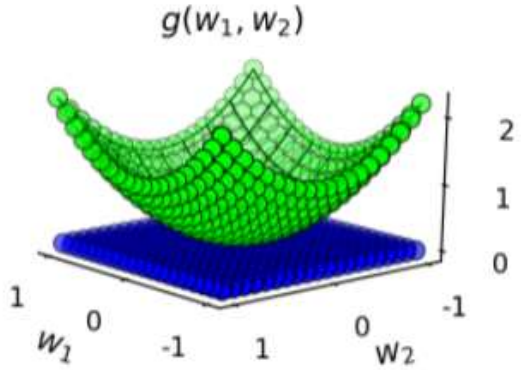

g(w1, w2)
2
1
0
1
0
-1
w1
-1
1
0
w2

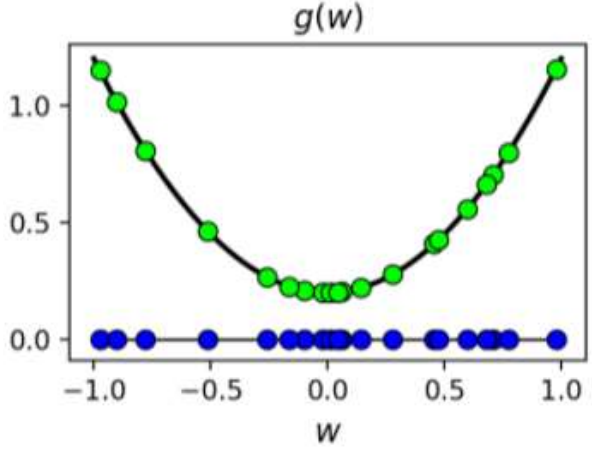

g(w)
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
w

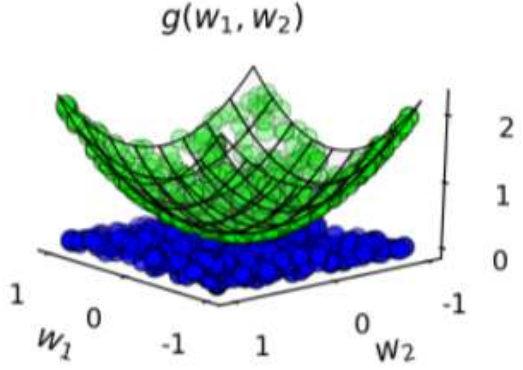

g(w1, w2)
2
1
0
1
0
-1
w1
-1
1
0
w2

- If we take enough samples we could certainly find an input very close to the true global minimum of either function.
- For example if we run this experiment again using 20 samples for the 2-d quadratic, and 400 samples for the 3-d quadratic (so that we are sampling with a 20×20 grid in the case of even sampling), using either approach we are able to find either the global minimum or a point very close to it.


g(w)
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
w


g(w1, w2)
2
1
0
1
0
-1
w1
-1
1
0
w2


g(w)
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
w


g(w1, w2)
2
1
0
1
0
-1
w1
-1
1
0
w2

- Notice with global optimization we really are employing the simple zero-order optimality condition, from a set of K chosen inputs $\left\{\mathbf{w}^k\right\}_{k=1}^K$ we are choosing the one input $\mathbf{w}^j$ lowest on the cost function

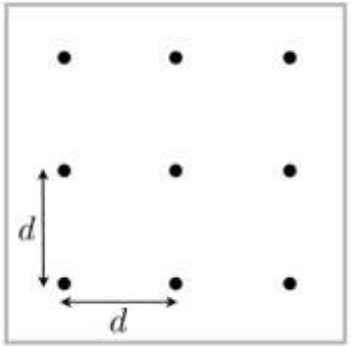
$$g\left(\mathbf{w}^j\right) \leq g\left(\mathbf{w}^k\right) \qquad k = 1, \ldots, K$$

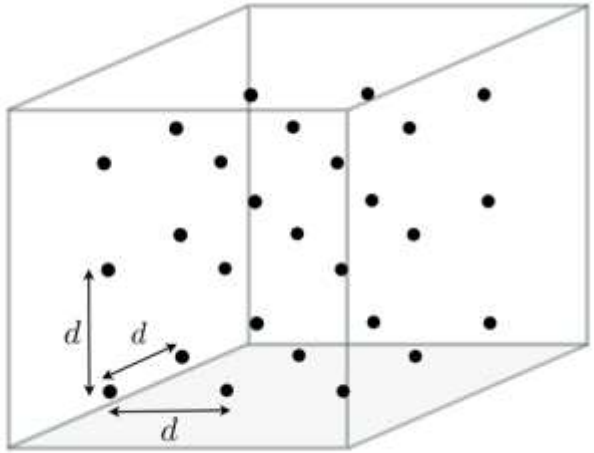
- This is indeed an approximation to the zero-order optimality condition discussed in the previous Section.

The curse of dimensionality and the failure of global optimization

- While this sort of zero-order evaluation works fine for low-dimensional functions, regardless of how the input points are chosen, it fails quickly as we try to tackle functions of larger dimensional input.
- This makes them unusable in modern machine learning since the functions we often deal with have input dimensions ranging from the hundreds to the hundreds of millions.

- If sampled *uniformly* we need *exponentially* more points to sample an input space as its dimension increases, as illustrated below.
- Here we sample an input space of dimension ***1***, ***2***, and ***3***, respectively keeping each point a distance ***d*** apart (along the coordinate axis).

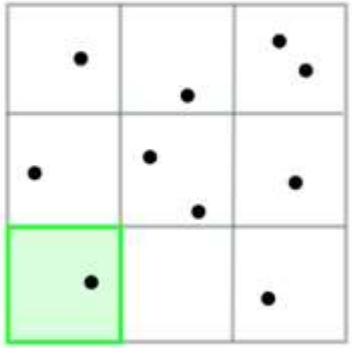




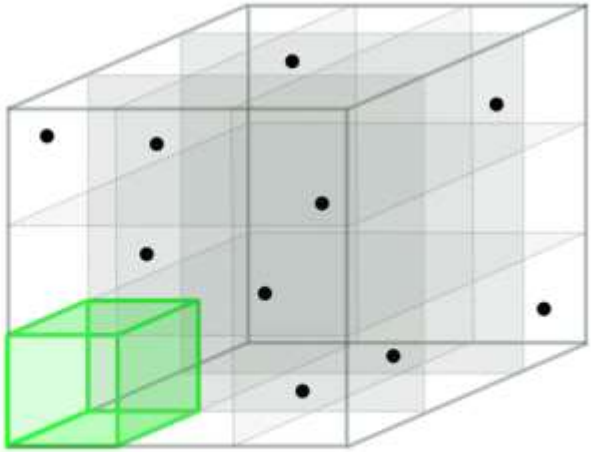
- The number of points required to achieve this increases from ***3*** to ***9*** to ***27*** as we increase our input dimension from ***1*** to ***2*** to ***3***.

- This issue is not ameliorated if we take samples randomly.
- Below we illustrate what happens if we take a fixed number of 10 points and spread them randomly over an input space.
- As the dimension of the input space *increases* the density of our sampling decreases exponentially.

3/10



1/10



0/10

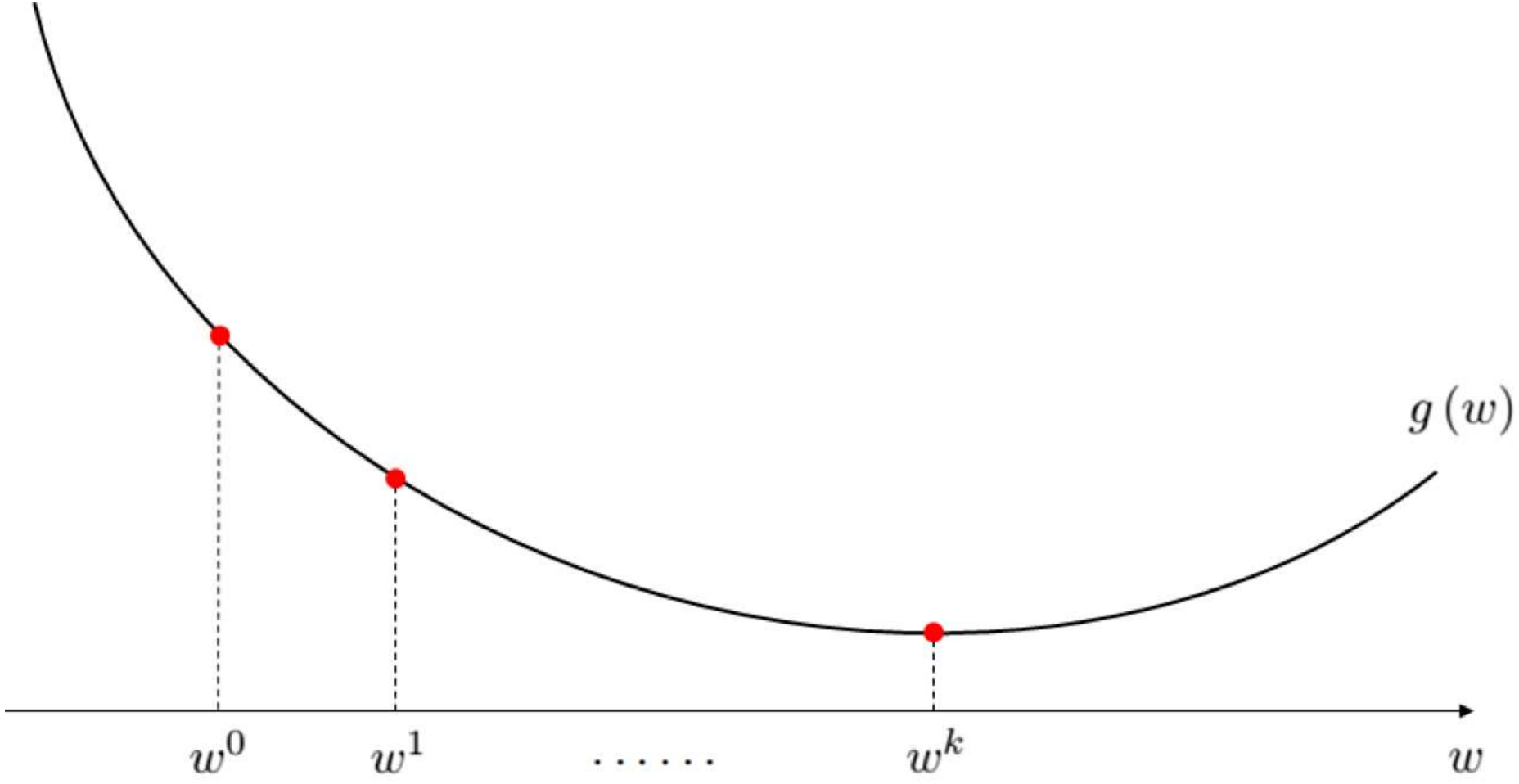
2.4 Local optimization methods

- *Local optimization methods* work by taking a single sample input and refine it sequentially, driving it towards an approximate minimum point.
- Local optimization methods are by far the most popular mathematical optimization schemes used in machine learning today.
- Indeed local optimization algorithms being the subject of the remaining of this as well as the next two Chapters.
- While there is substantial variation in the kinds of specific local optimization methods we will discuss going forward, they all share a common overarching framework which we introduce in this Section.

The big picture

- As we saw in the previous Section global optimization methods test a multitude of simultaneously sampled input points to determine an approximate minimum of a given function $g(\mathbf{w})$.
- *Local optimization methods*, on the other hand, work in the opposite manner *sequentially* refining a single sample input called an *initial point* until it reaches an approximate minimum.

- Starting with a sample input / initial point $\mathbf{w}^0$, local optimization methods refine this initialization sequentially *pulling the point* 'downhill' towards points that are lower and lower on the function.
- From $\mathbf{w}^0$ the point is 'pulled' downhill to a new point $\mathbf{w}^1$ lower on the function i.e., where $g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right)$. The point $\mathbf{w}^1$ then itself 'pulled' downwards to a new point $\mathbf{w}^2$, etc.


$g\,(w)$
w^0
w^1
. . . . . .
w^k
w

- This refining process yields a sequence of K points (starting with our initializer)

$$\mathbf{w}^0,\ \mathbf{w}^1,\ \ldots,\ \mathbf{w}^K$$

- Here each subsequent point is (again generally speaking) on a lower and lower portion of the function i.e.,

$$g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right) > \cdots > g\left(\mathbf{w}^K\right)$$

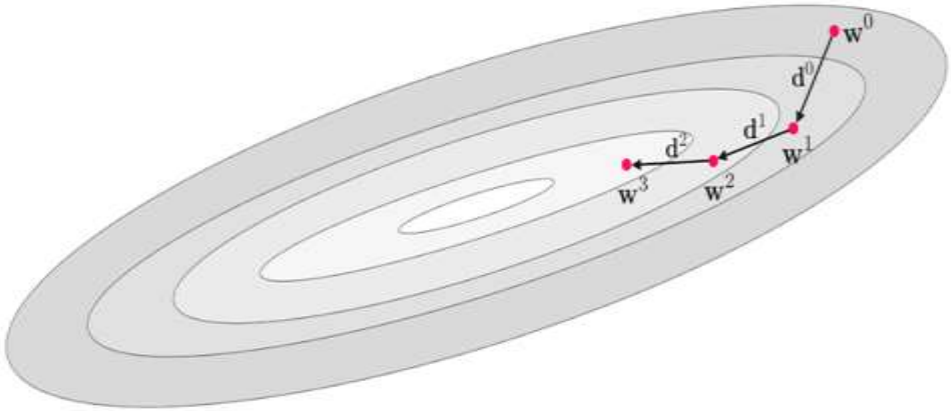
- Unlike the global approach, a wide array of specific local optimization methods scale gracefully with input dimension.
- This is because the sequential refinement process - which can be designed in a variety of ways - can be made extremely effective.

The general framework

- Every local method, regardless of form, works generally as follows.
- To take the first step from an initial point $\mathbf{w}^0$ to the very first update $\mathbf{w}^1$ (which should be lower on the function than the initializer) what is called a *descent direction* is at $\mathbf{w}^0$ is found.
- This is a direction vector $\mathbf{d}^0$ in the input space that stems from $\mathbf{w}^0$ - i.e., it begins at $\mathbf{w}^0$ and points away from the initial point - that if followed leads to an input point with lower function evaluation.

- When such a direction is found the first update $\mathbf{w}^1$ is then literally the sum

$$\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0$$


$\mathbf{w}^0$
$\mathbf{d}^0$
$\mathbf{d}^1$
$\mathbf{d}^2$
$\mathbf{w}^1$
$\mathbf{w}^2$
$\mathbf{w}^3$

- To refine the point $\mathbf{w}^1$ we look for a new descent direction $\mathbf{d}^1$ - one that moves 'downhill' stemming from the point $\mathbf{w}^1$.
- When we find such a direction the second update $\mathbf{w}^2$ is then formed as the sum

$$\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1.$$

- And we keep going like this, determining further descent directions producing - in the end - a sequence of input points (starting with our initializer) that goes as follows.

$$
\begin{aligned}
&\mathbf{w}^0 \\
&\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0 \\
&\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1 \\
&\vdots \quad \vdots \quad \vdots \\
&\mathbf{w}^K = \mathbf{w}^{K-1} + \mathbf{d}^{K-1}
\end{aligned}
$$

- Here $\mathbf{d}^{k-1}$ is the descent direction defined at the k^{th} step of the process:

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

- And we have $g(\mathbf{w}^0) > g(\mathbf{w}^1) > g(\mathbf{w}^2) > \cdots > g(\mathbf{w}^K)$

- How are these *descent directions* stemming - stemming from each subsequent update - actually found?

- There are a multitude of ways of determining proper descent directions - indeed in the remaining Sections of this Chapter we discuss *zero-order* (also called *Hessian free*) approaches for doing this.
- In the following Chapters we describe *first* and *second* order approaches (i.e., approaches that leverage the first and/or second derivative of a function to determine descent directions).
- In other words - it is precisely this - how the descent directions are determined - which distinguishes local optimization schemes from one another.

The steplength parameter

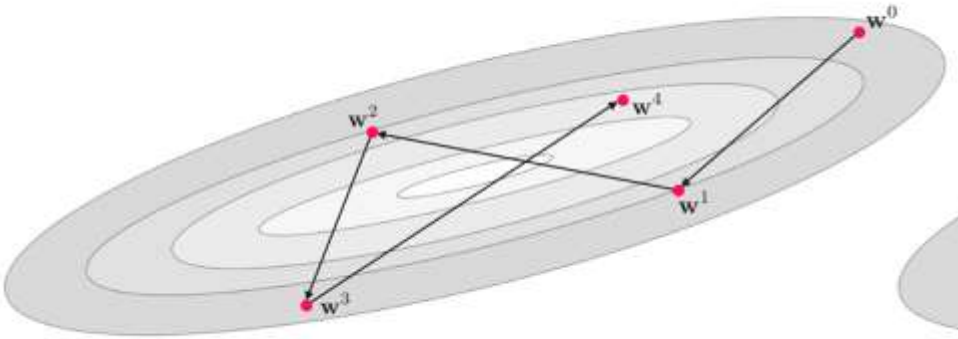
- We can easily calculate precisely how far we travel at the k^{th} step of a generic local optimization scheme.

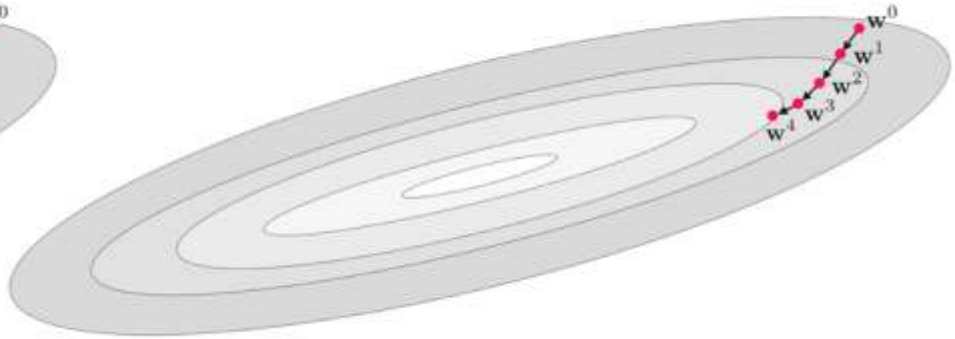
Measuring the distance traveled from the previous $(k-1)^{th}$ point we can see that we move a distance precisely equal to the length of the $(k-1)^{th}$ descent direction as

$$\left\|\mathbf{w}^{k}-\mathbf{w}^{k-1}\right\|_{2}=\left\|\ \left(\mathbf{w}^{k-1}+\mathbf{d}^{k-1}\right)-\mathbf{w}^{k-1}\right\|_{2}=\left\|\mathbf{d}^{k-1}\right\|_{2}.$$

- Depending on how our descent directions are generated we may or may not have control over their length (all we ask is that they point in the right direction, 'down hill').
- This can mean even if they point in the right direction - towards input points that are lower on the function - that their *length* could be problematic.

- If for example, they are too long then a local method can oscillate wildly at each update step never reaching an approximate minimum, or likewise if they are too short *in length* a local method we move so sluggishly slow that far too many steps would be required to reach an approximate minimum.
- Both of these potentialities are illustrated in the Figure below.


$\mathbf{w}^0$
$\mathbf{w}^1$
$\mathbf{w}^2$
$\mathbf{w}^3$
$\mathbf{w}^4$


$\mathbf{w}^0$
$\mathbf{w}^1$
$\mathbf{w}^2$
$\mathbf{w}^3$
$\mathbf{w}^4$

- Because of this potential problem many local optimization schemes come equipped with what is called a *steplength parameter* (also called a *learning rate* parameter).
- We control this value, and it helps us more directly control the length of each update step (hence the name *steplength parameter*).
- This is simply a parameter - typically denoted by the Greek letter α - that is added to the generic local optimization update scheme described above

- With a step-length parameter the generic k^{th} update step in equation (6) is written analogously as

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

The only difference between this form for the k^{th} step and the original is that now we scale the descent direction $\mathbf{d}^{k-1}$ by the parameter α - which we can set as we please.

Re-calculating the distance traveled by the k^{th} step with this added parameter we can see that

$$\left\|\mathbf{w}^{k}-\mathbf{w}^{k-1}\right\|_2 = \left\| \left(\mathbf{w}^{k-1}+\alpha\mathbf{d}^{k-1}\right)-\mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\mathbf{d}^{k-1}\right\|_2.$$

In other words the length of the k^{th} step is now proportional to the descent direction, and we can fine tune precisely how far we wish to travel in its direction by setting α as we please.

A common choice is to set α to some fixed small constant value for each of the K steps - but like local optimization methods themselves there are a number of ways of usefully setting the step-length parameter which we will discuss in the future.

2.5 Random search

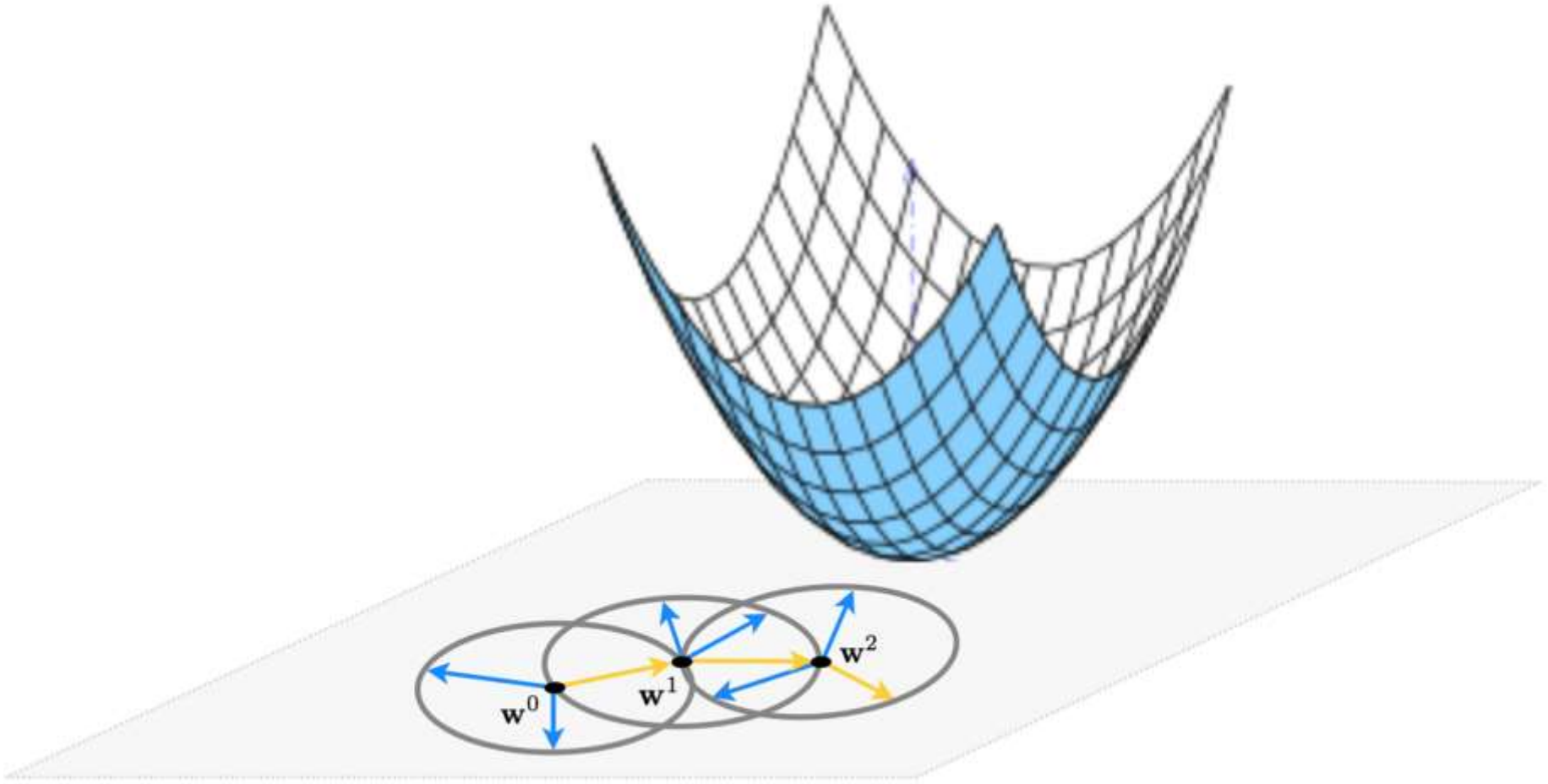
- In this Section we describe our first local optimization algorithms - *random local search*.
- With this instance of the general local optimization framework discussed in the previous Section we seek out a descent direction at each step by examining a number of random directions stemming from our current point.
- Like the global optimization scheme, scales poorly with the dimension of input and ultimately disqualifying random search for use with many modern machine learning problems.

- However this zero-order approach to local optimization is extremely useful as a simple example of the general framework introduced previously, allowing us to give simple yet concrete algorithmic example of universally present ideas like *descent directions*, various choices for the *steplength parameter*, and *issues of convergence*.

The random search algorithm

- The defining characteristic of the *random local search* (or just *random search*) - as is the case with every local optimization method - is how the descent direction $\mathbf{d}^{k-1}$ is chosen.
- With random search we do (perhaps) the laziest possible thing: we look locally around the current point in a fixed number of random directions for a point that has a lower evaluation, and if we find one we move to it.

- This idea illustrated figuratively in the picture below, where the function being minimized is the simple quadratic $g(w_1, w_2) = w_1^2 + w_2^2 + 2$, written more compactly as $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$.
- Here for visualization purposes we set the number of random directions sampled P = 3. At each step only one of the three candidates produces a *descent direction* - drawn as a yellow arrow - while the other two are *ascent directions* drawn in blue.


$\mathbf{w}^0$
$\mathbf{w}^1$
$\mathbf{w}^2$

More precisely, at the k^{th} step of random search we pick a number P of random directions to try out.

Generating the p^{th} random direction $\mathbf{d}^p$ stemming from the previous step $\mathbf{w}^{k-1}$ we have a candidate point to evaluate

$$\mathbf{w}_{\text{candidate}} = \mathbf{w}^{k-1} + \mathbf{d}^p$$

- After evaluating all P candidate points we pick the one that gives us the *smallest* evaluation i.e., the one with the index given by the smallest evaluation

$$s = \underset{p=1...P}{\text{argmin}} \; g\left(\mathbf{w}^{k-1} + \mathbf{d}^{p}\right)$$

- Finally, if best point found has a smaller evaluation than the current point i.e., if $g\left(\mathbf{w}^{k-1} + \mathbf{d}^{s}\right) < g\left(\mathbf{w}^{k-1}\right)$ then we move to the new point $\mathbf{w}^{k} = \mathbf{w}^{k-1} + \mathbf{d}^{s}$,otherwise we examine another batch of P random directions and try again.

- If we are to choose a set of directions at each step randomly, how shall we choose them?
- One idea could be to use choose some distribution - e.g., a Gaussian - and (at each step) use samples from this distribution as our candidate directions.
- The only issue with doing this is one of consistency: each of the candidate directions - if constructed in this way - would have different lengths.

- Since we have no apriori reason for doing this at each step, to keep our random candidate directions consistent we can normalize them to have the same length e.g., length one.
- Indeed this is how we illustrated the algorithm figuratively in the illustration above. If we use directions of unit-length in our algorithm - i.e., where $\|\mathbf{d}\|_2 = 1$ always - this means that at each step of the algorithm we move a distance of length one since

$$\|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \|\left(\mathbf{w}^{k-1} + \mathbf{d}\right) - \mathbf{w}^{k-1}\|_2 = \|\mathbf{d}\|_2 = 1.$$

- From here we can adjust each step to have whatever length we desire by introducing a *steplength parameter* α into each step to completely control how far we travel with each step (as discussed in the previous Section). This more general step looks like the following

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

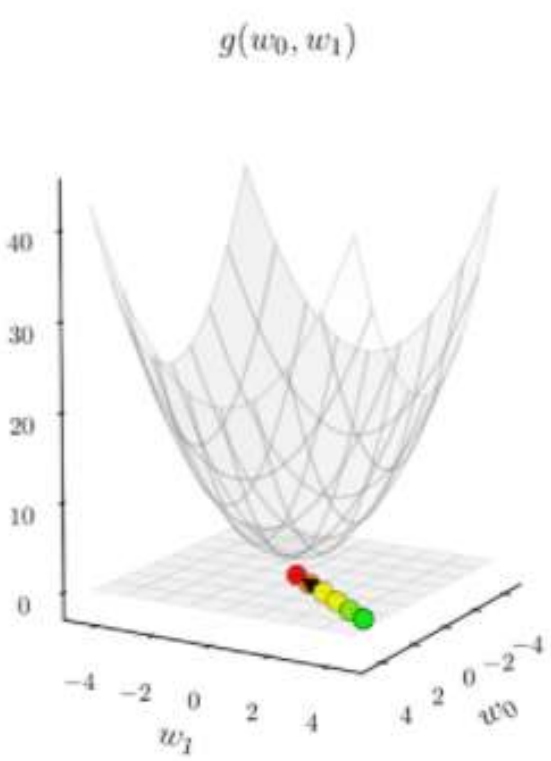
- The length of this step - using a unit-length directions - is now exactly equal to the steplength α, as

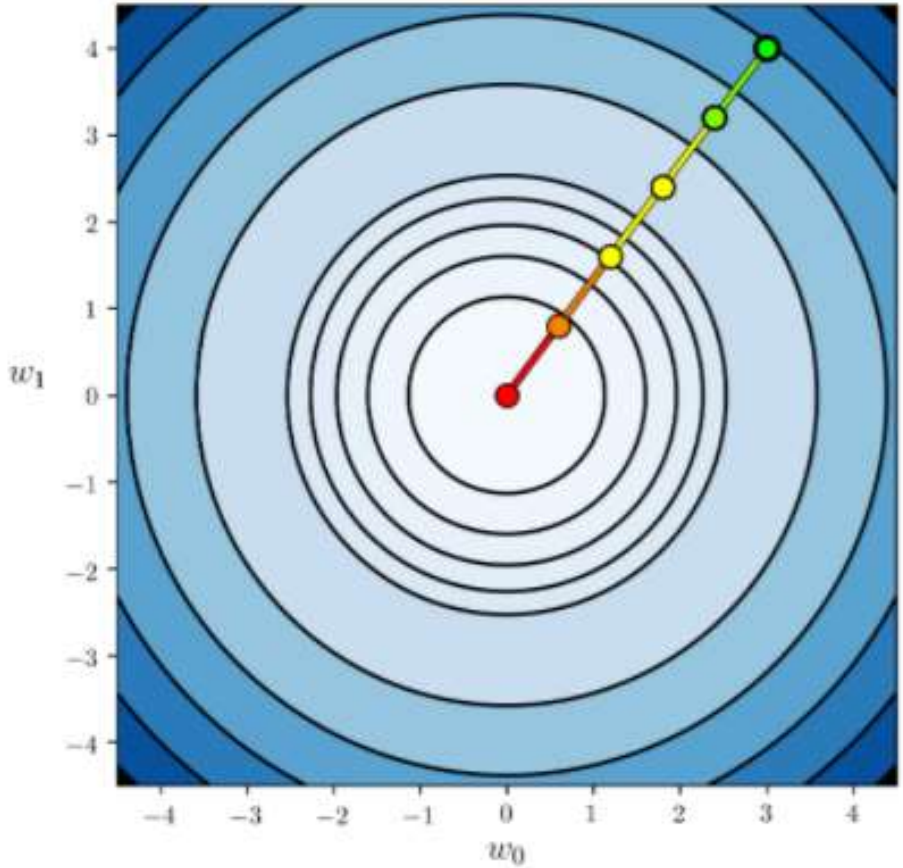
$$\|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \|\left(\mathbf{w}^{k-1} + \alpha\mathbf{d}\right) - \mathbf{w}^{k-1}\|_2 = \|\alpha\mathbf{d}\|_2 = \alpha\|\mathbf{d}\|_2 = \alpha$$

- Now at the k^{th} step we try out P unit-length random directions - but scaled by the steplength parameter so that the distance we travel is actually α - taking the one that provides the greatest decrease in function value.

Example: Random search applied to minimize a simple quadratic

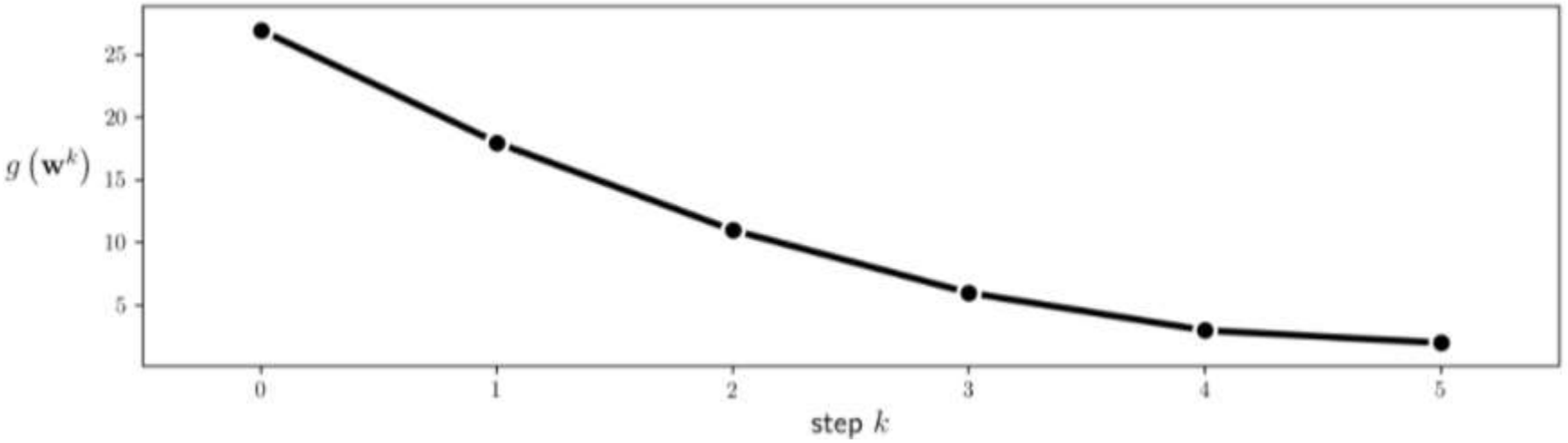
- Below we show the result of running random local search for 4 steps with $\alpha = 1$ for all steps, at each step searching for $P = 1000$ random directions to minimize the simple quadratic $g(w_0, w_1) = w_0^2 + w_1^2 + 2$
- A three-dimensional view is shown on the left, along with the set of steps produced by the algorithm colored from green - at the start of the run where we initialize at $\mathbf{w}^0 = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ - to red when the algorithm halts.
- In the right panel is the same picture, only viewed from directly above.


$g(w_0, w_1)$
40
30
20
10
0
−4
−2
0
2
4
w_1
4
2
0
−2
−4
w_0


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

- If N is greater than ***2*** we cannot make a plot like the ones above to tell how well a particular run of random search performed - or any local optimization method for that matter.

- A more general way to view the steps from the particular run of any local method regardless of input dimension is to plot the corresponding sequence of function evaluations.
- That is if the returned history of weights from a local method run is $\{\mathbf{w}^k\}_{k=0}^K$ we plot corresponding function evaluations $\{g\left(\mathbf{w}^k\right)\}_{k=0}^K$ as pairs $\left(k, g\left(\mathbf{w}^k\right)\right)$ as we demonstrate below.
- This allows us to tell - regardless of the input dimension ***N*** - how the algorithm performed, and is called a *cost function history plot*.

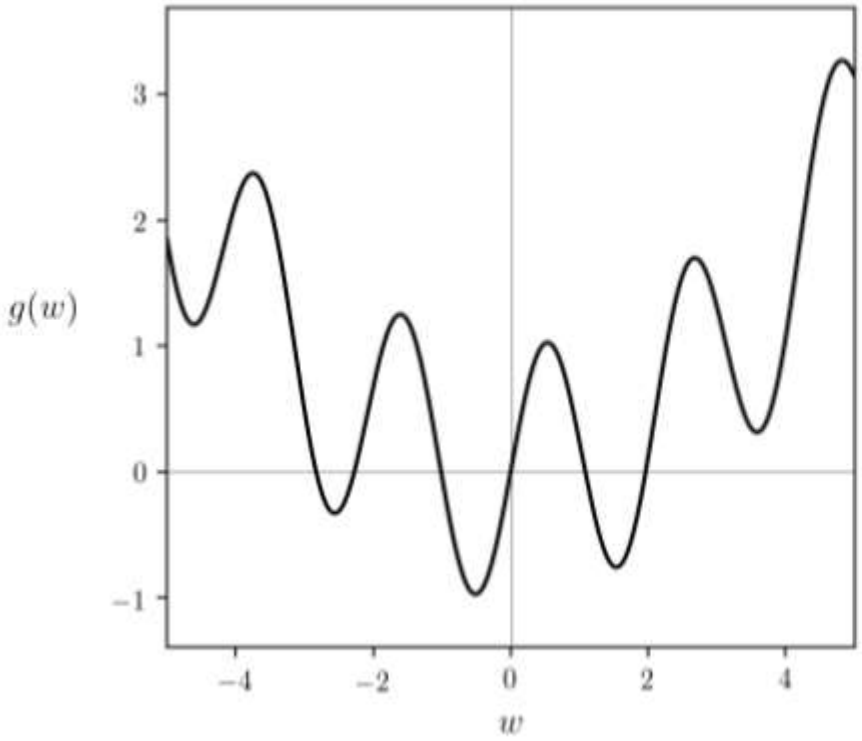

25
20
15
10
5
$g\left(\mathbf{w}^k\right)$
0
1
2
3
4
5
step k

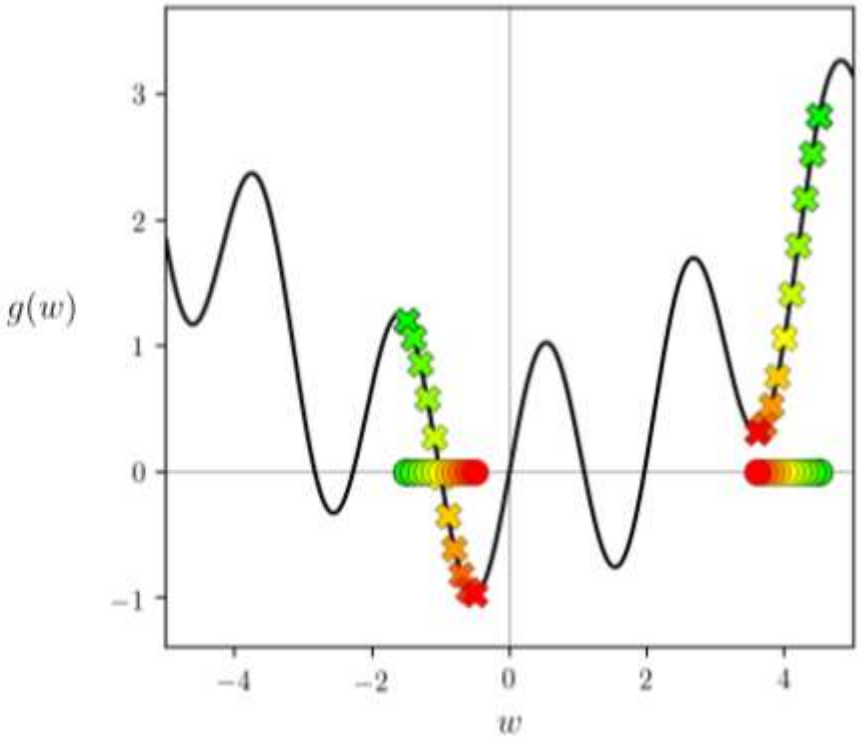
Example: Minimizing a function with many local minima using random search

- In this example we show what one may need to do in order to find the global minimum of a function using (normalized) random local search.
- For visualization purposes we use the single-input function

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

- We initialize two runs - at $w^0 = 4.5$ and $w^0 = -1.5$. For both runs we use a steplength of $\alpha = 0.1$ fixed for all 10 iterations.


$g(w)$
3
2
1
0
−1
−4
−2
0
2
4
w


$g(w)$
3
2
1
0
−1
−4
−2
0
2
4
w

- As can be seen by the result depending on where we initialize we may end up near a local or global minimum - here resulting from the first and second initialization respectively.

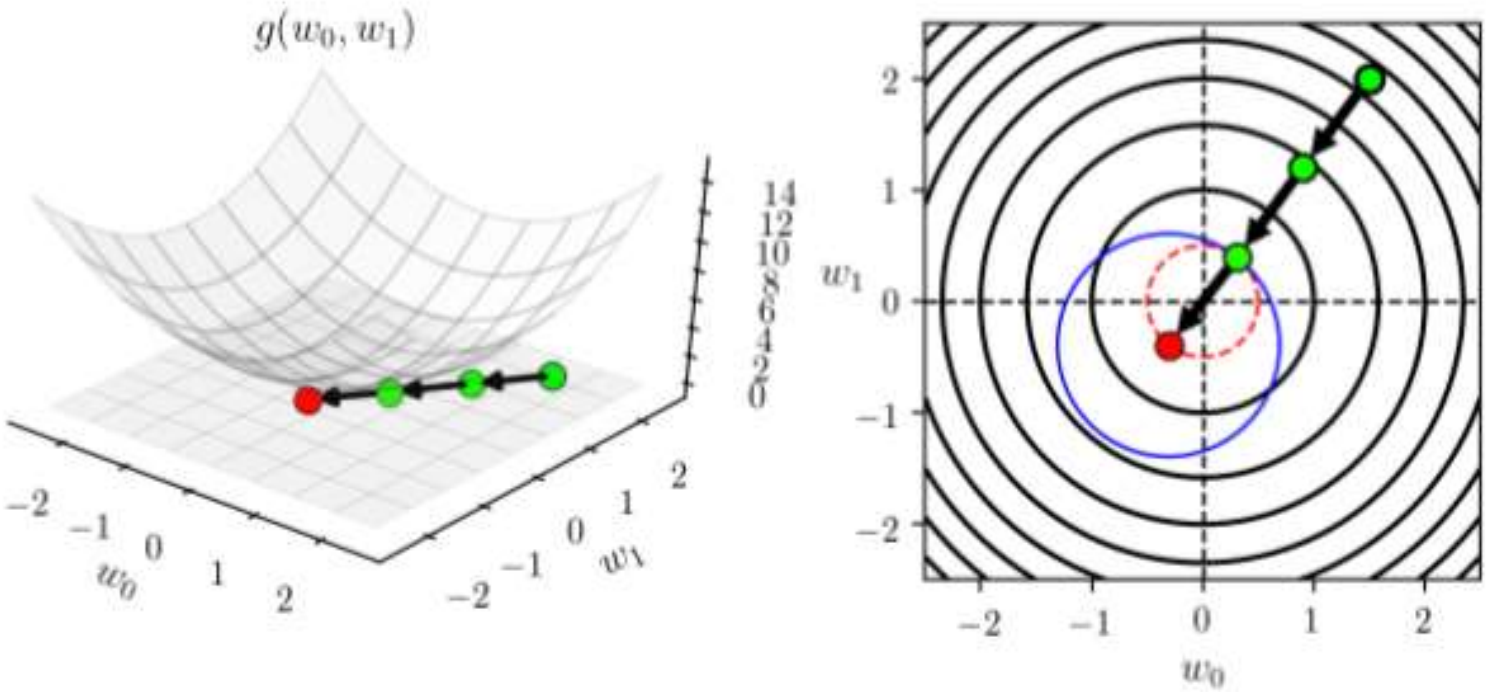
Example: Exploring fundamental steplength rules

- In the examples of the previous Subsection we steplength parameter α set fixed for all steps of each run.
- This choice - to take one value for α and use if to each and every step of the algorithm - is called a fixed *steplength rule*.
- One can also imagine changing the value of α from step-to-step in a single run of a local algorithm. A rule that adjusts the value of α from step-to-step is often referred to as an *adjustable* steplength rule, of which there are many in use.

- One of the most common adjustable steplength rule is the so-called *diminishing* steplength rule, which we explore here.

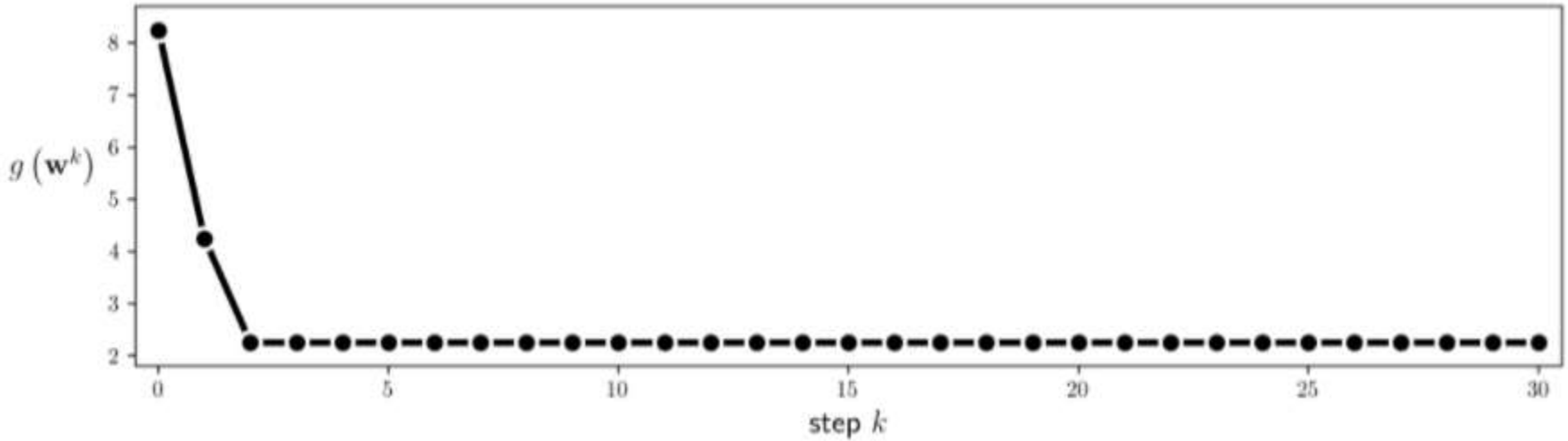
Example: Unit length steps fail to converge to global minimum

- Here we re-run the random local search algorithm using the same simple quadratic and algorithm settings as described in the first Example above.
- Now we initialize at the point $\mathbf{w}^0 = \begin{bmatrix} 1.5 \\ 2 \end{bmatrix}$ which prevents the algorithm from reaching the function's global minimum.

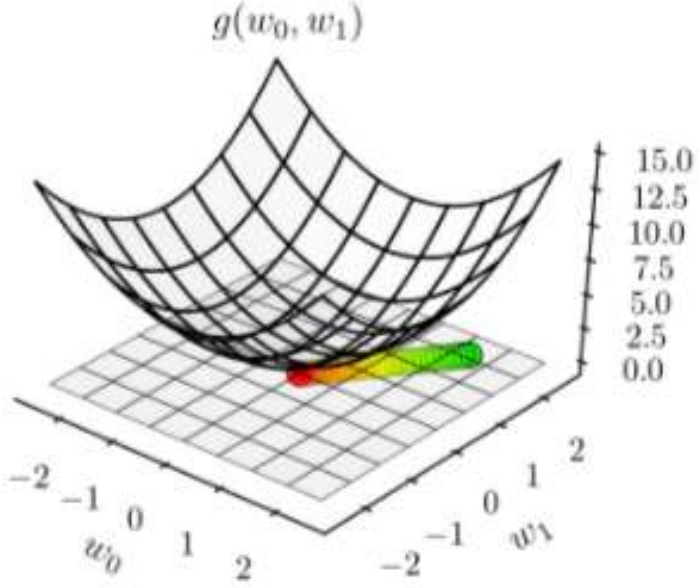

$g(w_0, w_1)$
w_0
w_1
w_1
w_0

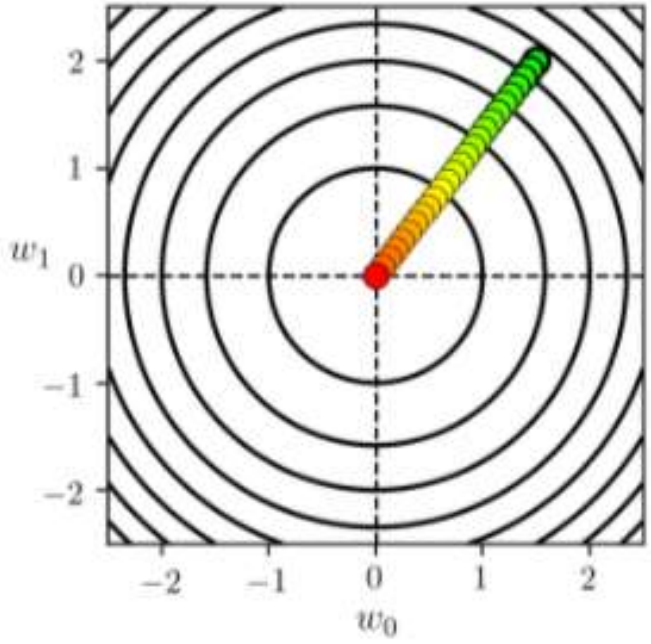
- The problem here is that each direction we take must have length one since we have set $\alpha = 1$ for all steps - thus the length of each step must be length one as well.
- If we could take *shorter* steps we could in fact descend onto lower contours of the quadratic, and get closer to the global minimum.

- We can visualize how the final 25 steps of this sort of run fail to descend by plotting the corresponding cost function history, which we do below.

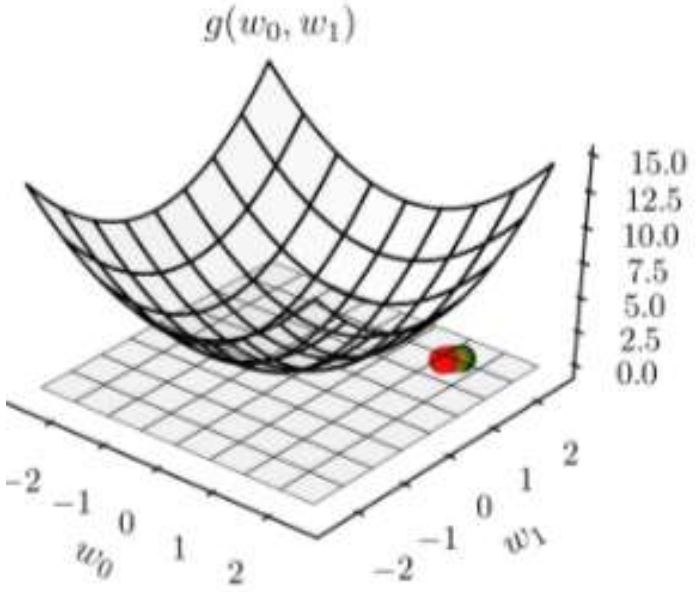

$g\left(\mathbf{w}^k\right)$
8
7
6
5
4
3
2
0
5
10
15
20
25
30
step k

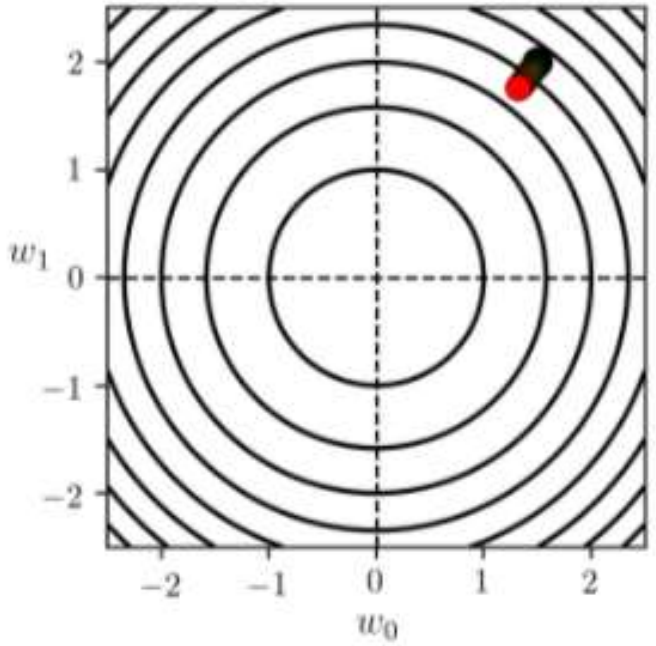
- Setting the steplength parameter α smaller we can look again make another run mirroring the one performed above, with much better results.
- Below we make the same run as above except now we set $\alpha = 0.1$ for all steps. Running the algorithm now we can see that it converges to a point much closer to the global minimum of the function at $\mathbf{w} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$


$g(w_0, w_1)$
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2
−1
0
1
2
w_0
w_1

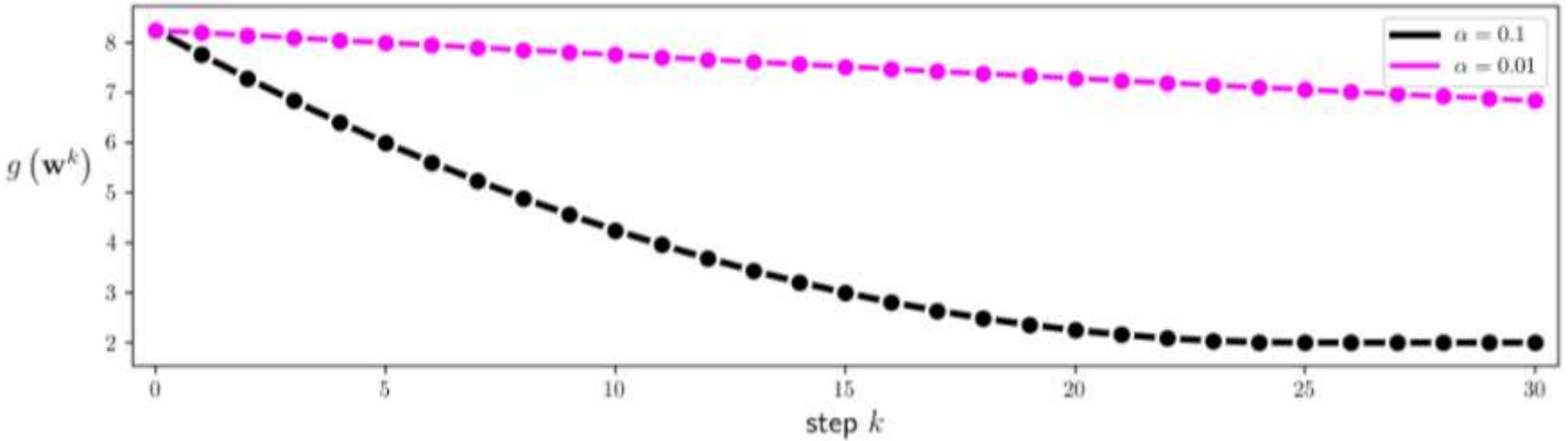

2
1
0
−1
−2
w_1
−2
−1
0
1
2
w_0

- Remember however that we need to be careful in choosing the steplength value with this simple quadratic, and by extension any general function.
- If - for example - we run the same experiment again but cut the step-length down to $\alpha = 0.01$ we do not reach a point anywhere near the global minimum, as we show by performing the same run but setting α to this value in the next Python cell.


$g(w_0, w_1)$
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2
−1
0
1
2
w_0
w_1


2
1
0
−1
−2
w_1
−2
−1
0
1
2
w_0

- Thus in general the combination of steplength and maximum number of iterations are best chosen together.
- The trade-off here is simple: a small stepsize combined with a large number of steps can guarantee convergence to towards a local minimum, but can be very computationally expensive.
- Conversely a large steplength and small number of maximum iterations can - as we saw in Example 6 - be cheaper but less effective at finding small evaluation points.


$g\left(\mathbf{w}^k\right)$
8
7
6
5
4
3
2
0
5
10
15
20
25
30
step k
$\alpha = 0.1$
$\alpha = 0.01$

- Another choice we have in choosing steplengths is to change its value at each step.
- For more advanced local search algorithms there are a host of ways of doing this.

- One simple approach: diminish the size of the steplength at each step.
- This is a safe choice of steplength because it ensures that the algorithm can get into any 'small nooks and crannies' where a function's minima may lie. This is often referred to as a *diminishing steplength rule*.

- A common way of producing a diminishing steplength is to set $\alpha = \frac{1}{k}$ at the k^{th} step of the process.

$$\|\mathbf{w}^k - \mathbf{w}^{k\ 1}\|_2 = \|\left(\mathbf{w}^{k\ 1} + \alpha\mathbf{d}\right) - \mathbf{w}^{k\ 1}\|_2 = \|\alpha\mathbf{d}\|_2 = \alpha\|\mathbf{d}\|_2 = \alpha = \frac{1}{k}.$$

- This gives us the benefit of shrinking the distance between subsequent steps as we progress, while at the same time we can see that

$$\sum_{k=1}^{K} \|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \sum_{k=1}^{K} \frac{1}{k}$$

- The beauty of this choice of stepsize is that clearly the stepsize decreases to zero as k increases i.e., $\alpha = \frac{1}{k} \longrightarrow 0$
- Simultaneously the total distance traveled by the algorithm goes to infinity as k increases i.e., $\sum_{k=1}^{K} \frac{1}{k} \longrightarrow \infty$
- In theory this means that an algorithm employing this sort of diminishing steplength rule can move around an infinite distance in search of a minimum all the while taking smaller and smaller steps.

2.6 Coordinate search and descent

- The coordinate search and descent algorithms are additional zero order local methods that get around the inherent scaling issues of random search by restricting the set of search directions to the coordinate axes of the input space.
- The concept is simple: random search was designed to minimize a function $g\left(w_1, w_2, \ldots, w_N\right)$ with respect to all of its parameters *simultaneously*.

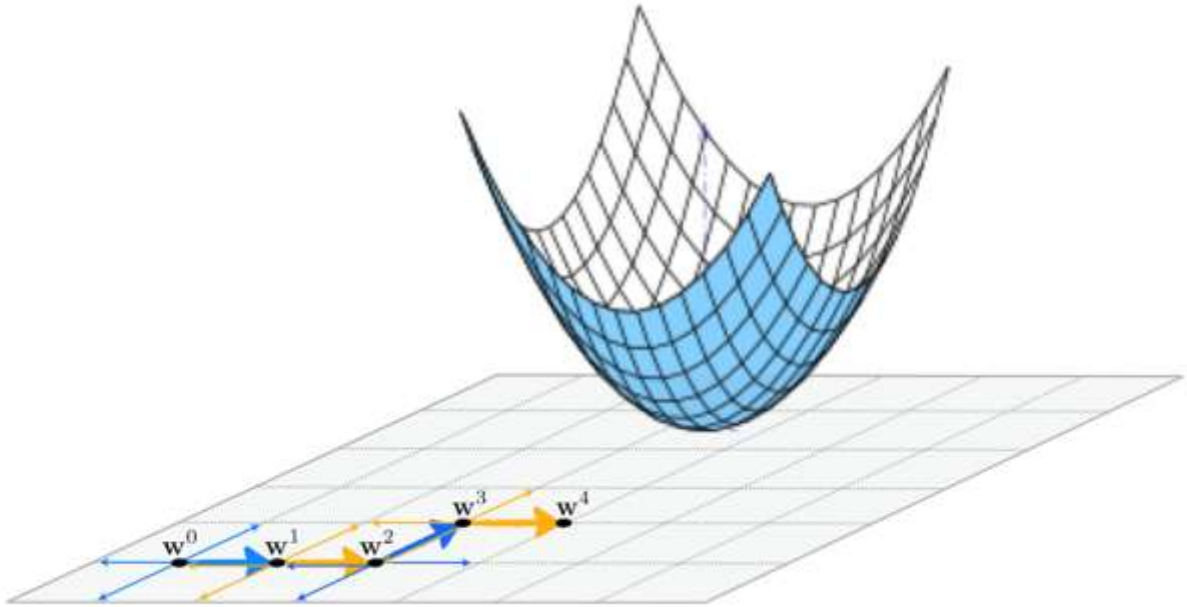
- With coordinate wise algorithms we attempt to minimize such a function with respect to one coordinate or weight at a time - or more generally one subset of coordinates or weights at a time - keeping all others fixed.
- While this limits the diversity of descent directions that can be potentially discovered, and thus more steps are often required to determine approximate minima, these algorithms are far more scalable than random search.

- Additionally - as was the case with the random search approach - these algorithms serve as predicates for an entire thread of higher order *coordinate descent methods* we will see in the next few Chapters.
- Through understanding coordinate search / descent in the comparatively simpler zero order framework we are currently in we lay the ground work for better understanding of this entire suite of powerful coordinate-based algorithms.

Searching through the coordinate axes

- The *coordinate search* algorithm takes the theme of descent direction search and - instead of searching randomly - restricts the set of directions to the coordinate axes of the input space alone.
- While this significantly limits the kinds of descent directions we can recover it far more scalable than seeking out a good descent direction at random, and opens the search-approach to determining descent directions to usage with higher dimensional input functions.

- As illustrated in the Figure below for an ***N*** = 2 dimensional example, with coordinate search we seek out the best descent direction among only the coordinate axes of the input space.
- This means in general that for a function of input dimension ***N*** we only look over ***2N*** directions - the positive and negative versions of each coordinate input.
- As with the random local search algorithm we will use unit-length directions, meaning that at every step the set of directions we search over always consists of just positive and negative versions of the *standard basis*.


$\mathbf{w}^0$
$\mathbf{w}^1$
$\mathbf{w}^2$
$\mathbf{w}^3$
$\mathbf{w}^4$

- In an N dimensional input space the n^{th} standard basis vector - denoted $\mathbf{e}_n$ - is just a vector of all zeros whose n^{th} entry is set to $***1***$.
- Searching only over the positive and negative set of these directions means at the k^{th} step of this local method we search only over the set of ***2N*** candidate directions $\{\pm \mathbf{e}_n\}_{n=1}^{N}$ for the best descent direction (if one exists).

- This means - in particular - each pair of candidate points using a single standard basis direction looks like

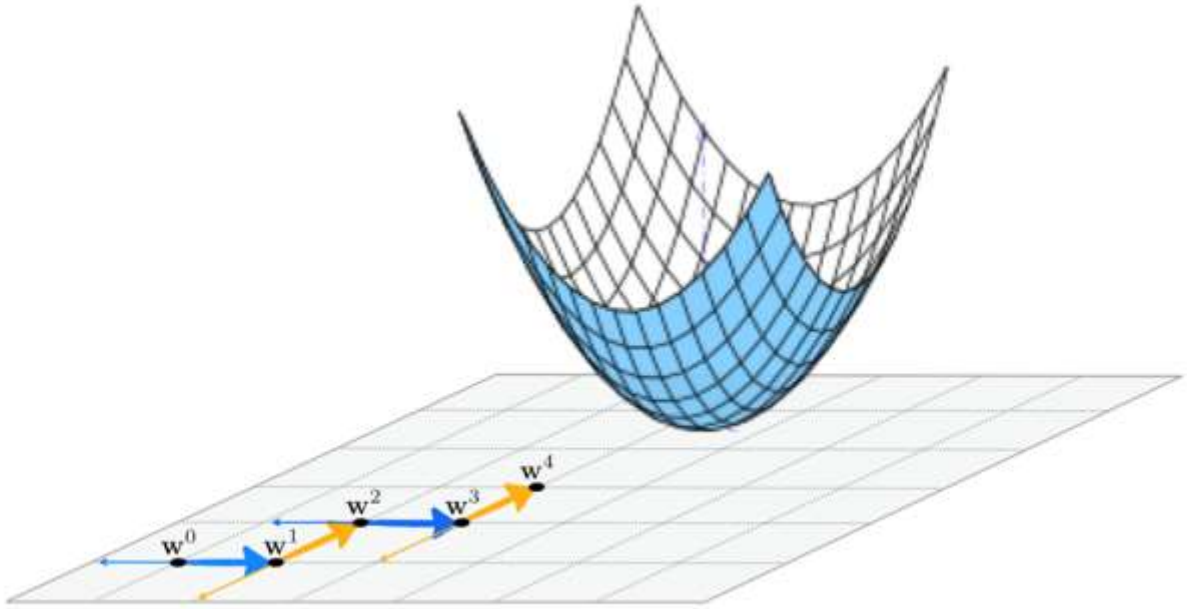
$$\mathbf{w}_{\text{candidate}} = \mathbf{w}^{k-1} \pm \alpha \mathbf{e}_n \, .$$

- It is this restricted set of directions we are searching over that distinguishes the coordinate search approach from the random search approach described in the previous Section, where the set of directions at each step was made up of random directions.
- While the diversity of the coordinate axes may limit the effectiveness of the possible descent directions it can encounter and thus require more steps to determine an approximate minimum, the restricted search makes coordinate search far more scalable than the random search method since at each step only $***2N***$ directions must be tested.

Zero-order coordinate descent

- A slight twist on the coordinate search produces a much more effective algorithm at precisely the same computational cost.
- Instead of collecting each coordinate direction (along with its negative), and then choosing a single best direction from this entire set, we can simply examine one coordinate direction (and its negative) at a time and step in this direction if it produces descent.

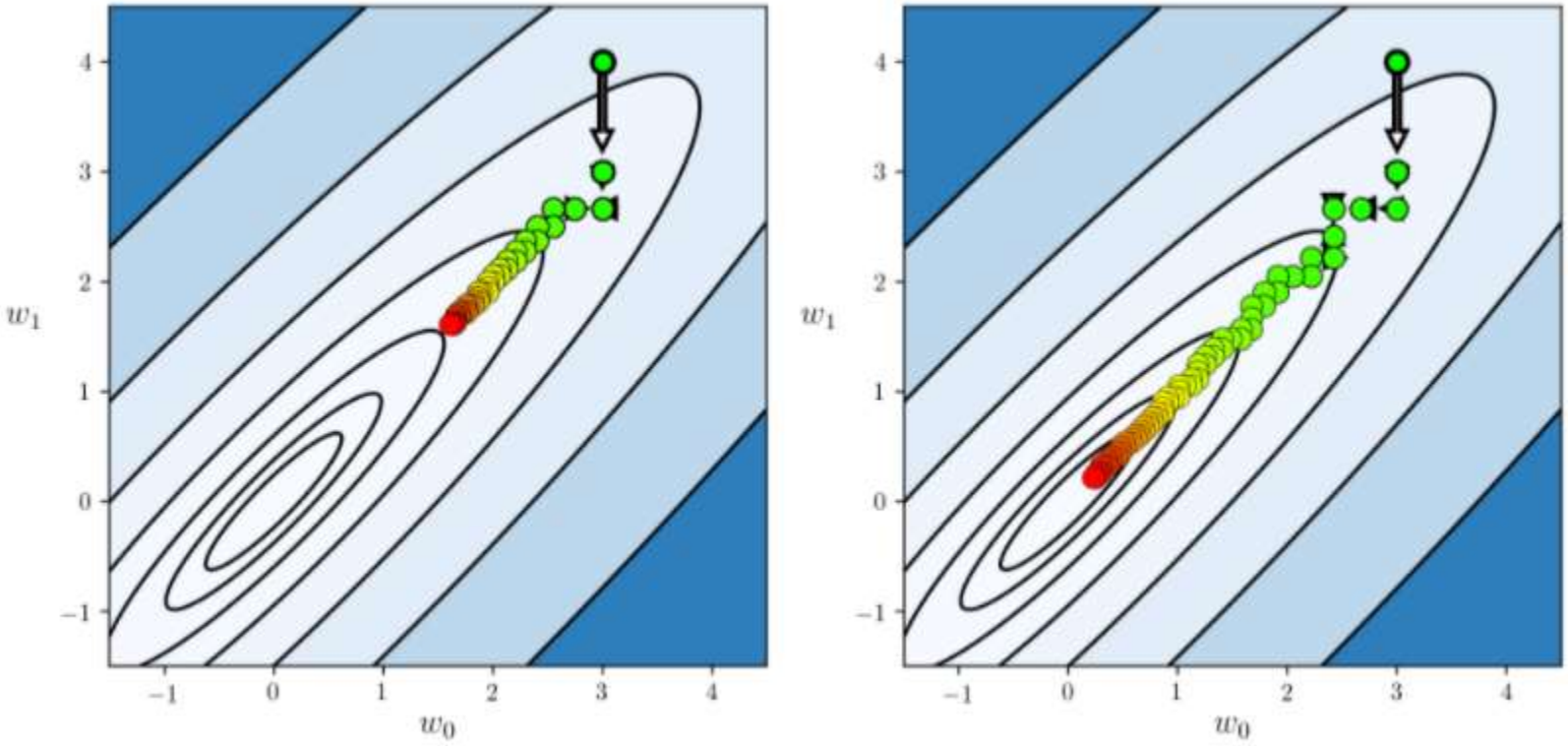
- Whereas with coordinate search we evaluate the cost function ***2N*** times (once per coordinate direction and its negative) to produce a single step, this alternative takes the same number of function evaluations but potentially moves ***N*** steps in doing so.
- In other words this means that for precisely the same cost as coordinate search we can (potentially) descent much faster with coordinate descent.


$\mathbf{w}^0$
$\mathbf{w}^1$
$\mathbf{w}^2$
$\mathbf{w}^3$
$\mathbf{w}^4$

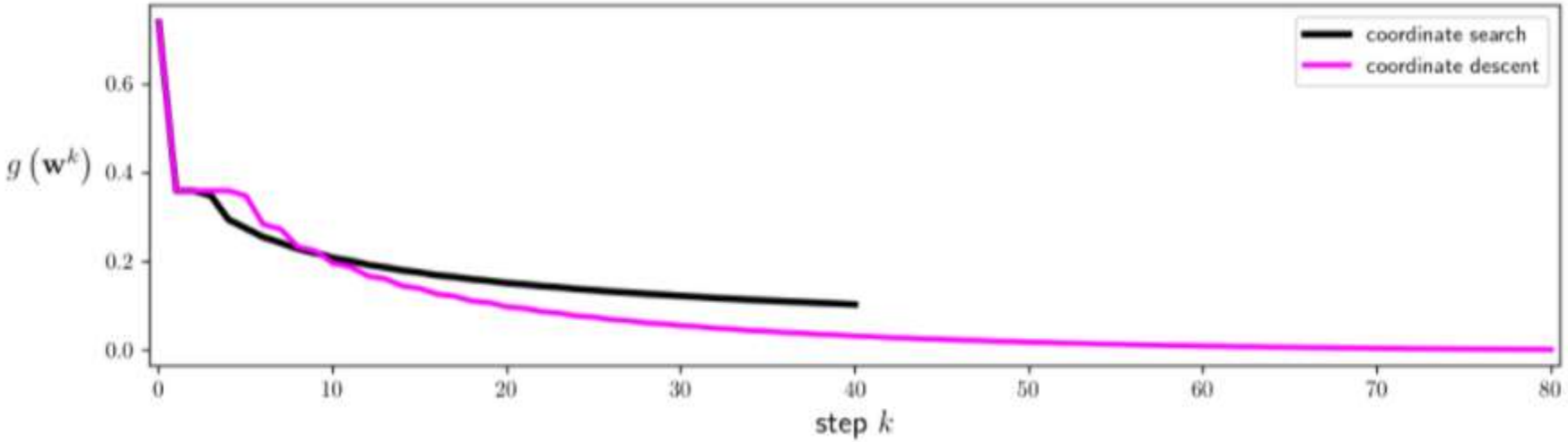
- This twist on the coordinate search approach is called a *coordinate descent*, since each step evaluates a single coordinate direction and decides whether or not to move in this direction alone.
- This particular algorithm - while itself being the most effective zero order method we have seen thus far by far - is a predicate for many useful coordinate descent approaches we will see in future Chapters as well.

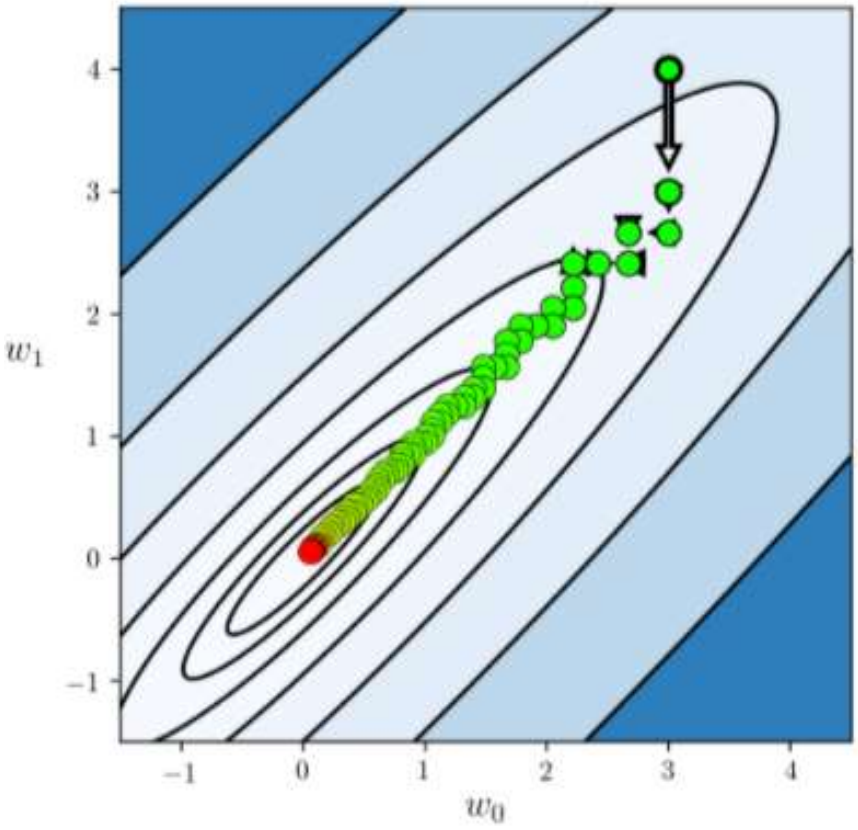
Example: Coordinate search versus coordinate descent

- In this example we compare the efficacy of coordinate search and the coordinate descent algorithm described above using the same function from the previous example.
- Here we compare ***20*** steps of coordinate search (left panel) and coordinate descent (right panel), using a diminishing step length for both runs.
- Because coordinate descent takes two steps for every single step taken by coordinate search we get significantly closer to the function minimum.


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

- We can view the precise difference more easily by comparing the two function evaluation histories via the cost function history plot, which we do below.
- Here we can see in this instance while the first several steps of coordinate search were more effective than their descent counterparts (since they search over the entire list of coordinate directions instead of one at a time), the coordinate descent method quickly overtakes search finding a lower point on the cost function.


coordinate search
coordinate descent
$g\left(\mathbf{w}^k\right)$
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
70
80
step k


4
3
2
1
0
−1
w_1
−1
0
1
2
3
4
w_0