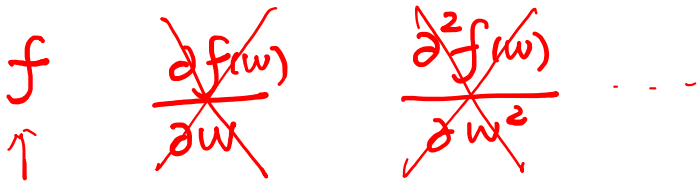
Zero-Order Optimization Techniques

Instructor: Hui Guan

Slides adapted from: https://github.com/jermwatt/machine_learning_refined

- 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 Module we begin our investigation of mathematical optimization by describing the *zero order optimization* techniques (also called *derivative-free optimization*)



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

Outline

- Minima and Maxima
- The Zero-Order Condition for Optimality
- Global optimization methods
- Local optimization methods
- Random search
- Coordinate search and descent

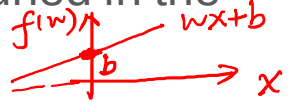
Visualizing minima and maxima

- 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.

- 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.

input variable

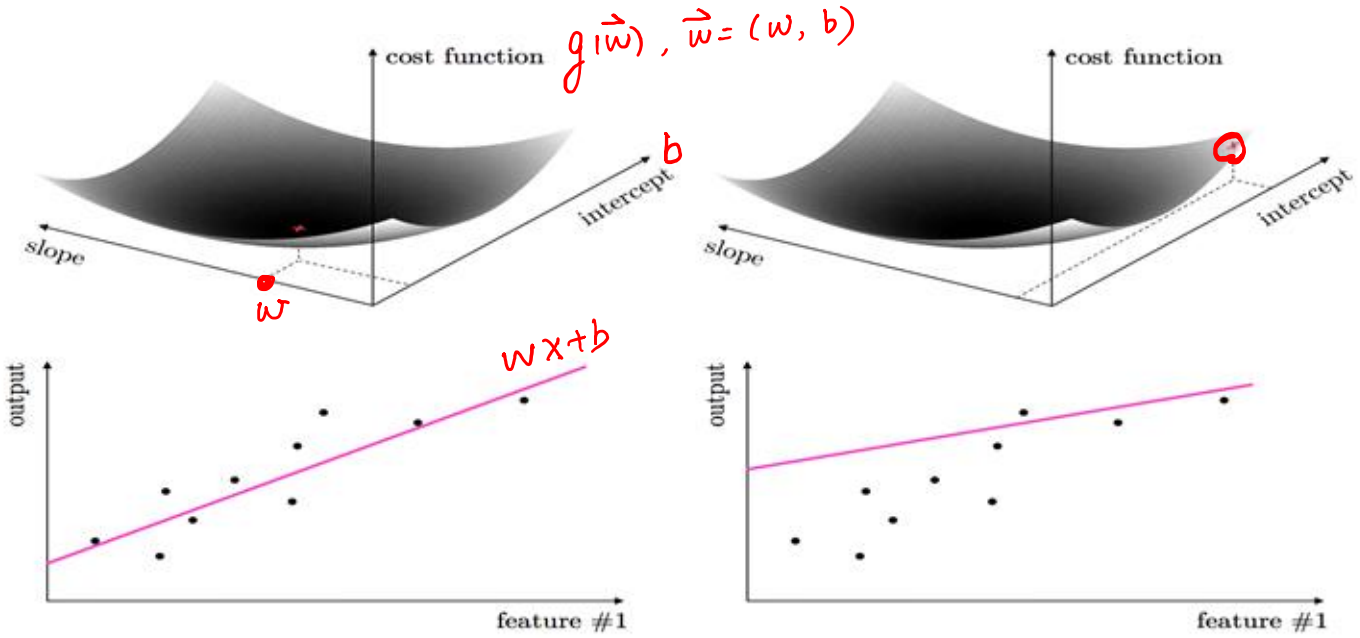
$f(w) = W \cdot X + b$



- 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*.

$$\min_{w} \; g(w) = g(f(w), y)$$

- 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, e.g., **the parameters that minimize the cost function**.

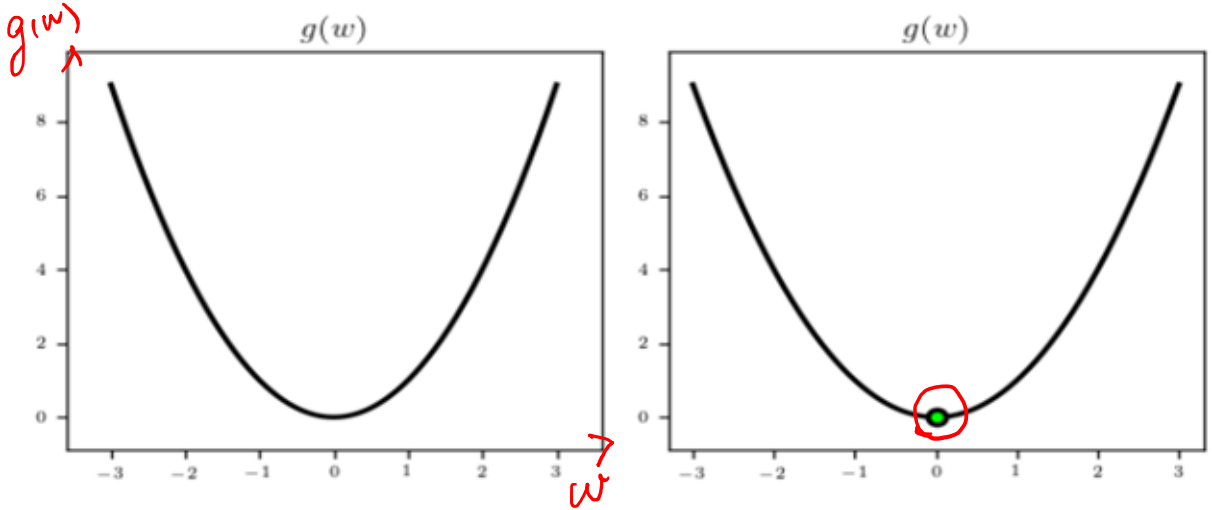


- The tuning of these parameters require the ***minimization of a cost function*** can be formally written as follows.
- For a generic function $g(\mathbf{w})$ taking in a general ***N*** dimensional input $\mathbf{w}$ the problem of finding the particular point $\mathbf{v}$ where ***g*** attains its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

$$\mathbf{w}^* = \arg\underset{\mathbf{w}}{\text{minimize}} \quad g\left(\mathbf{w}\right)$$

2.2 The Zero-Order Condition for Optimality

- 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?

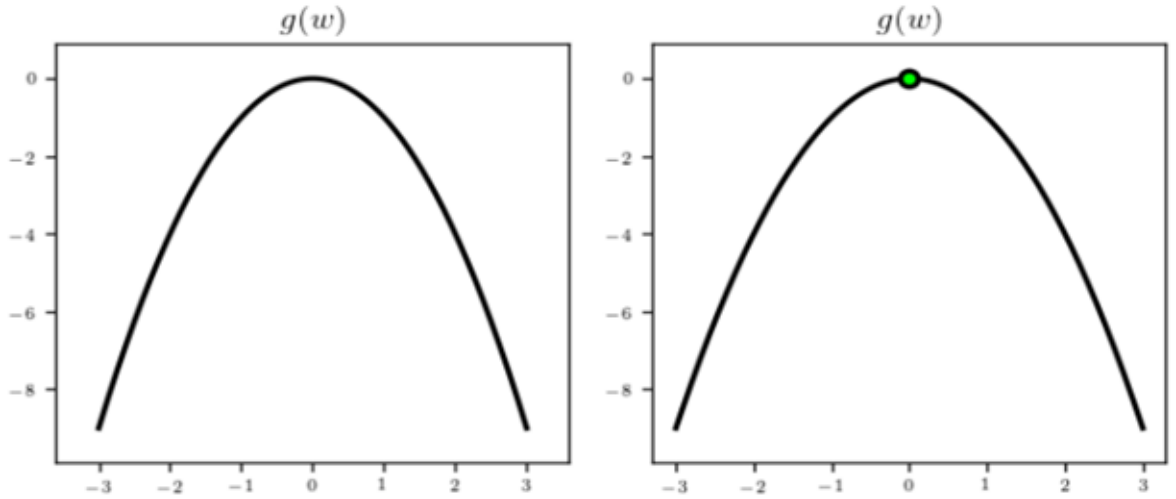


- Formally a point $w^\star$ gives the smallest point on the function if

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

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

- If we multiply the quadratic function in the previous example by -1, as plotted below.
- 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).



- 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*.

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

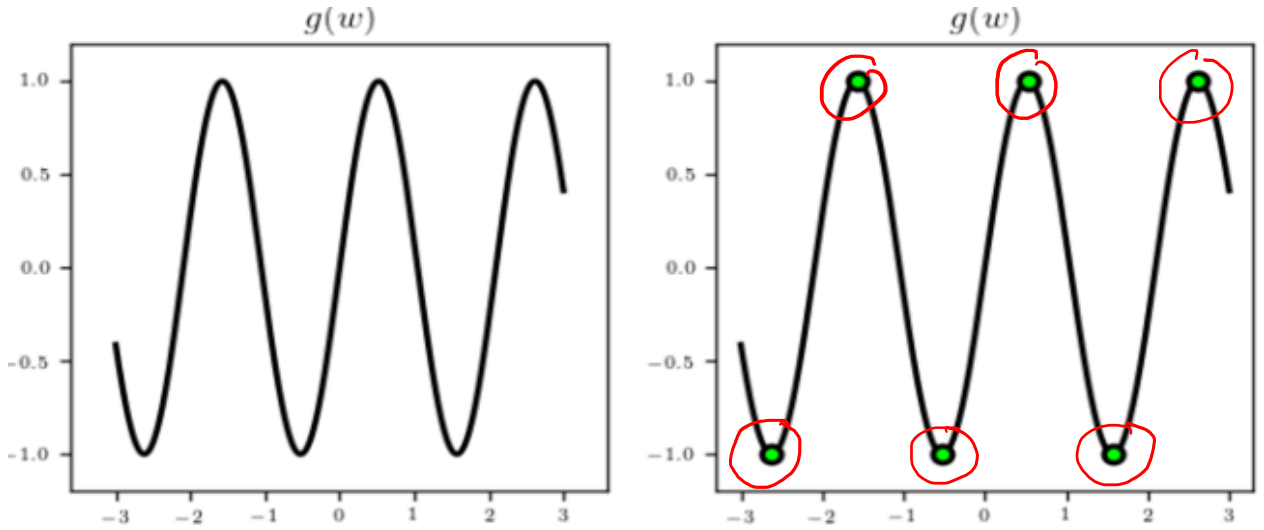
$$\underset{\mathbf{w}}{\text{maximize}} \quad g\left(\mathbf{w}\right).$$

- 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).



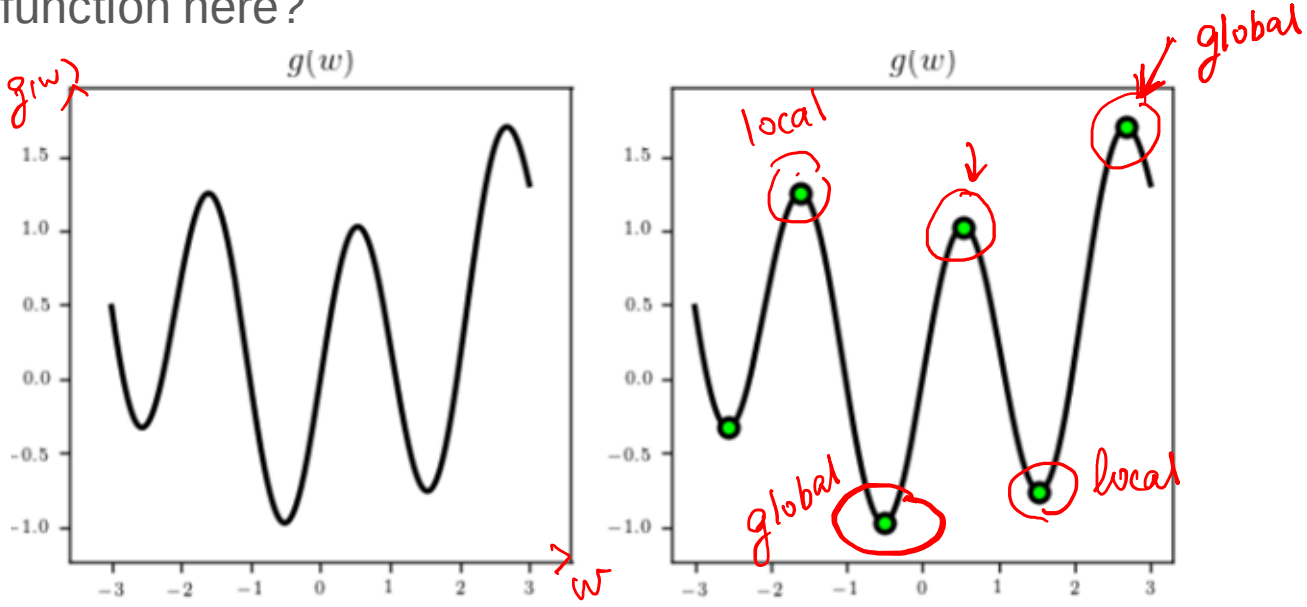
Example: Minima and maxima of the sum of a sinusoid and a quadratic

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

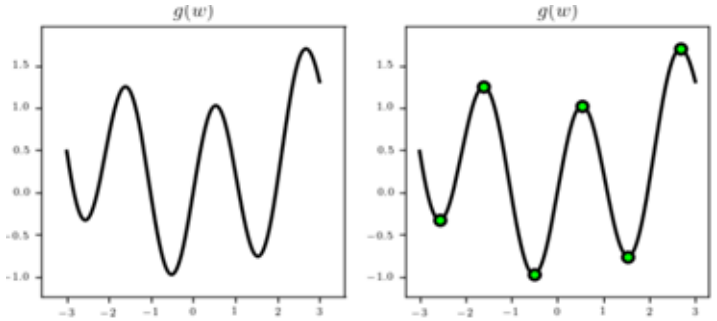
$$g(w) = \sin(3w) + 0.1w^2$$

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?



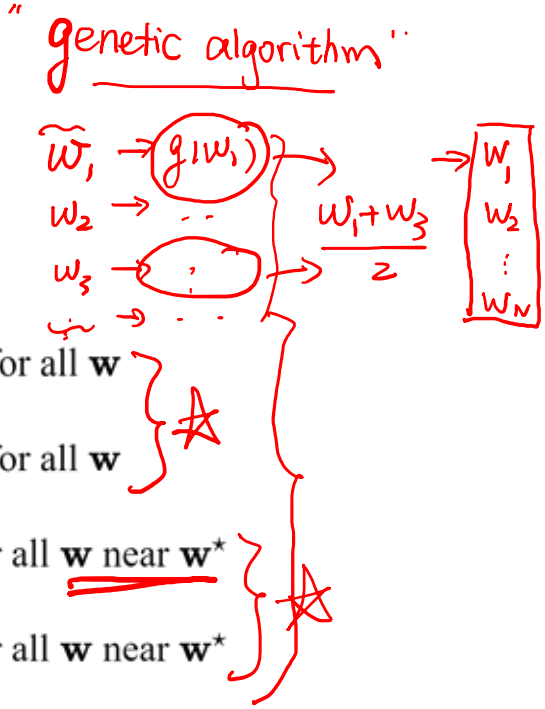
- Here we have a global minimum around $w^\star = -0.5$ and a global maximum around $w^\star = 2.7$
- We also have minima and maxima that are *locally optimal* - for example the point around $w^\star = 0.8$ is a local maximum.
- Likewise the point near $w^\star = 1.5$ is a *local minimum* - since it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

Why are these called *zero-order* conditions? Because each of their definitions involves only a function itself - and nothing else.

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**
  - **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.

Choosing input points

- 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.

"grid search"

"random sampling"

Example: Evaluating a quadratic to determine its minimum

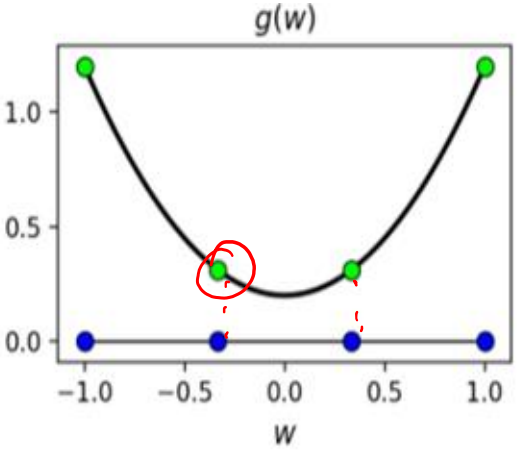
- Here we illustrate two sampling methods for finding the global minimum of simple 2-d and 3-d quadratic functions

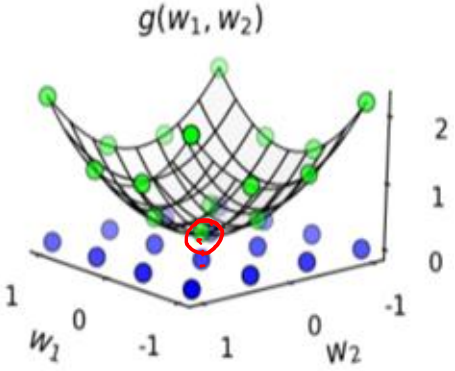
$$g(w) = w^2 + 0.2$$
$$g(w_1, w_2) = w_1^2 + w_2^2 + 0.2$$

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

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

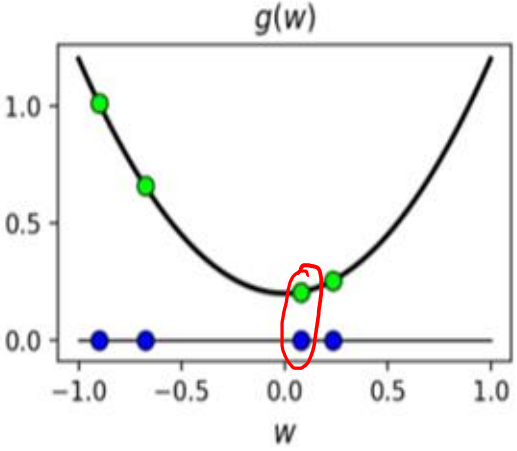
evenly sample

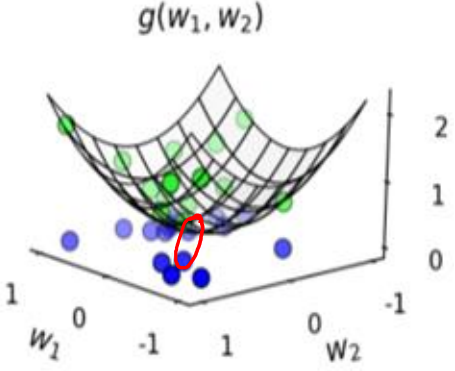




$g(w_1, \ldots \quad w_{100})$

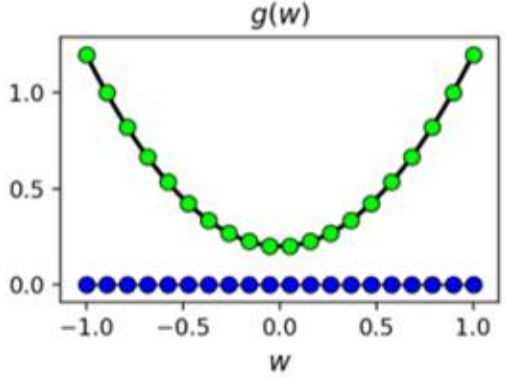
randomly sample

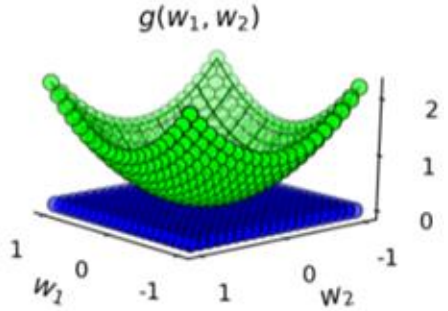




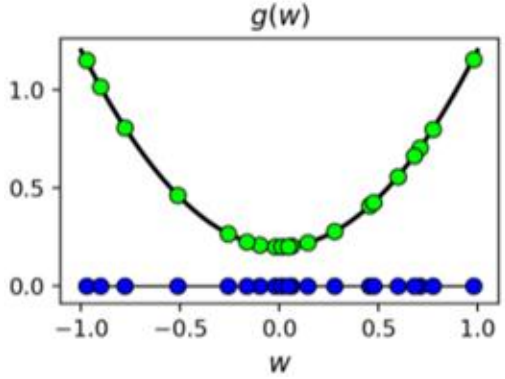
- If we take enough samples we could certainly find an input very close to the true global minimum of either function.

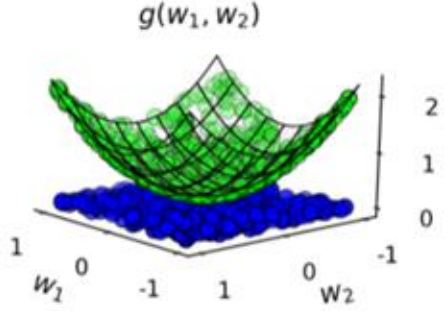
20 samples





400 samples





- 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 an approximation to the zero-order optimality condition.

Q: Pros vs. Cons of random search and uniform search?

Journal of Machine Learning Research 13 (2012) 281-305 Submitted 3/11; Revised 9/11; Published 2/12

Random Search for Hyper-Parameter Optimization

James Bergstra JAMES.BERGSTRA@UMONTREAL.CA
Yoshua Bengio YOSHUA.BENGIO@UMONTREAL.CA
Département d'Informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC, H3C 3J7, Canada

https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

Abstract

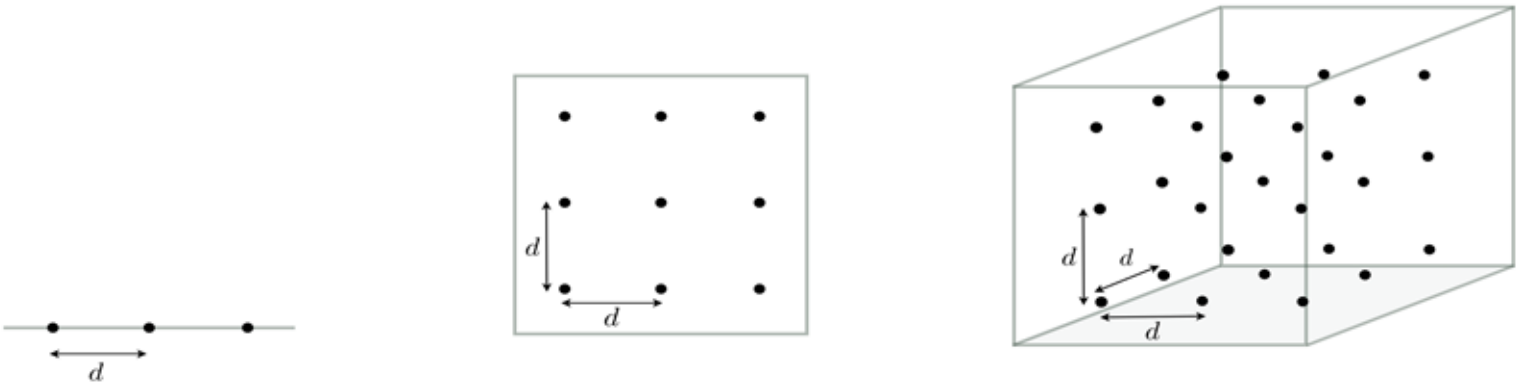
Grid search and manual search are the most widely used strategies for hyper-parameter optimization. This paper shows empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid. Empirical evidence comes from a comparison with a large previous study that used grid search and manual search to configure neural networks and deep belief networks. Compared with neural networks configured by a pure grid search, we find that random search over the same domain is able to find models that are as good or better within a small fraction of the computation time. Granting random search the same computational budget, random search finds better models by effectively searching a larger, less promising configuration space. Compared with deep belief networks configured by a thoughtful combination of manual search and grid search, purely random search over the same 32-dimensional configuration space found statistically equal performance on four of seven data sets, and superior performance on one of seven. A Gaussian process analysis of the function from hyper-parameters to validation set performance reveals that for most data sets only a few of the hyper-parameters really matter, but that different hyper-parameters are important on different data sets. This phenomenon makes grid search a poor choice for configuring algorithms for new data sets. Our analysis casts some light on why recent "High Throughput" methods achieve surprising success—they appear to search through a large number of hyper-parameters because most hyper-parameters do not matter much. We anticipate that growing interest in large hierarchical models will place an increasing burden on techniques for hyper-parameter optimization; this work shows that random search is a natural baseline against which to judge progress in the development of adaptive (sequential) hyper-parameter optimization algorithms.

Keywords: global optimization, model selection, neural networks, deep learning, response surface modeling

The curse of dimensionality and the failure of global optimization

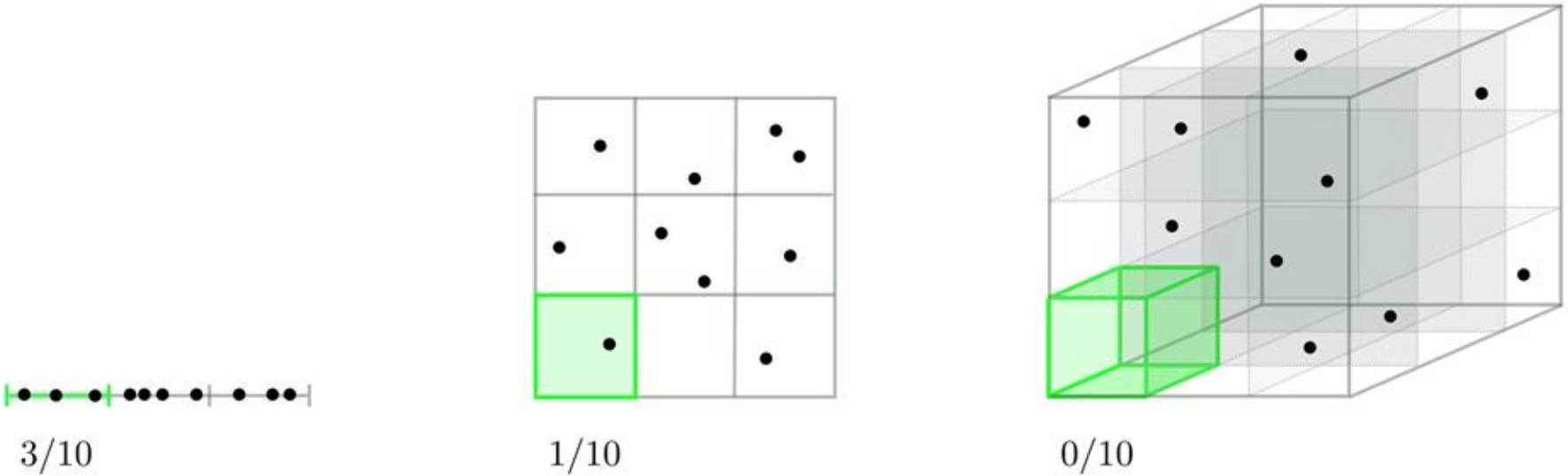
- The global optimization approach fails quickly as we try to tackle functions of larger dimensional input.
- This makes it unusable in modern machine learning, since the functions we often deal with have input dimensions ranging from the hundreds to the hundreds of millions.

- 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***.

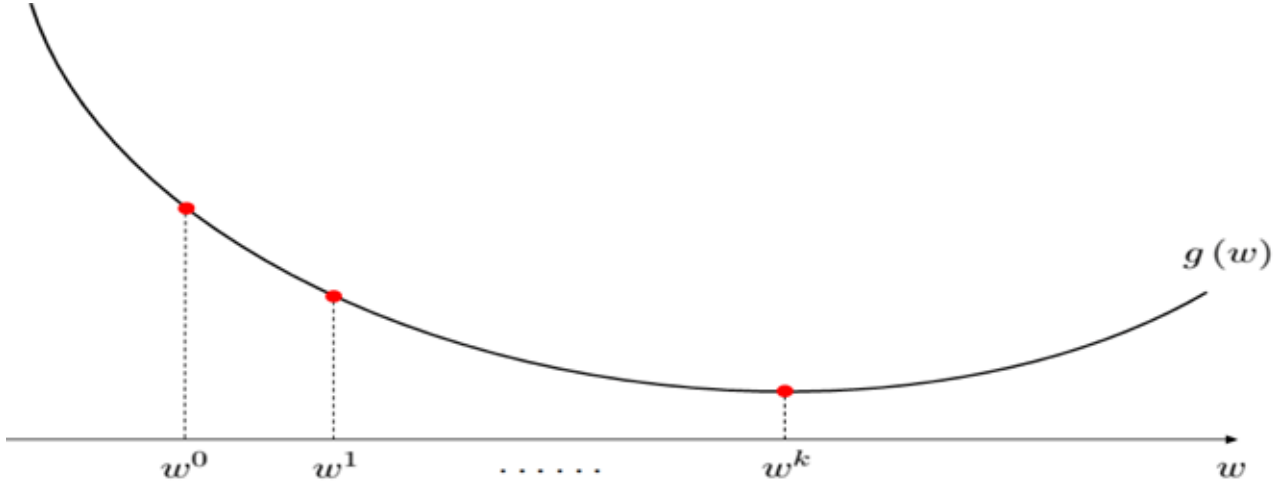
- This issue is not ameliorated if we take samples randomly.
- Below we illustrate what happens if we take a fixed number of 10 points and spread them randomly over an input space.
- **As the dimension of the input space *increases* the density of our sampling decreases exponentially.**



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**.
- 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 here.

- Starting with a sample input / initial point $\mathbf{w}^0$, local optimization methods refine this initialization sequentially *pulling the point* 'downhill' (decent) 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.



- 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.,

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

- The global approach vs. the local approach:
  - 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.

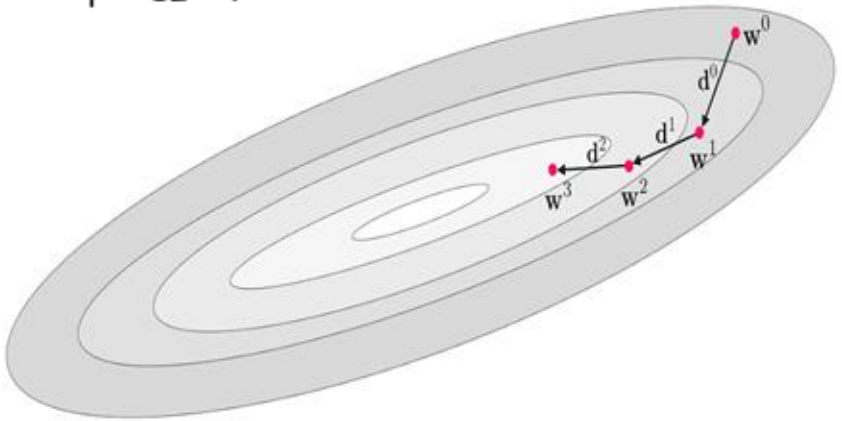
- The general framework of the local optimization methods:
- From an initial point $\mathbf{w}^0$, find a ***descent direction*** at $\mathbf{w}^0$ to derive $\mathbf{w}^1$

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

- From $\mathbf{w}^1$, identify a descent direction at. $\mathbf{w}^1$ to derive the next parameters

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

- Repeat the process until converge



Contour plot:
darker regions → higher on the function.

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

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

- *Zero-order approaches*
- *first* and *second* order approaches (i.e., approaches that leverage the first and/or second derivative of a function to determine descent directions).

The steplength parameter

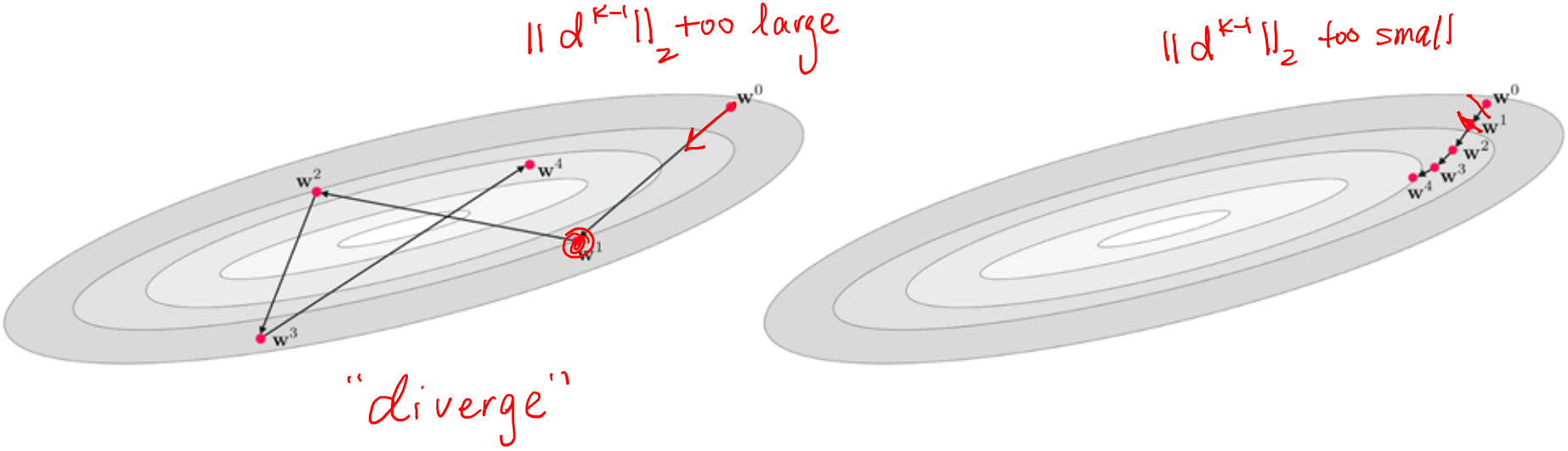
- We can easily calculate precisely how far we travel at the k^{th} step of a generic local optimization scheme.

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

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

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

- 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**.



- 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

$g(w)$

$\min \mathcal{L}(w) = \mathcal{L}_1(w) + \lambda \mathcal{L}_2(w)$

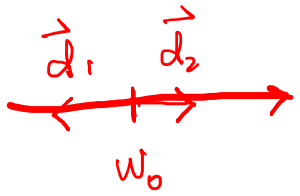
- With a step-length parameter the generic k^{th} update step is written analogously as

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

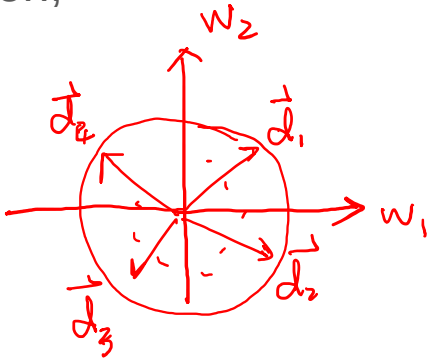
- The length of the k step is now proportional to the descent direction

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

2.5 Random search



- A local optimization algorithm - *random local search*.
  - 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.



sample P number of $\{\vec{d}\}$

$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ $\vec{w}^T = [w_1, w_2]$

- The function being minimized is the simple quadratic $g(w_1, w_2) = w_1^2 + w_2^2 + 2$, written more compactly as $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2.$ $\vec{w}^T \cdot \vec{w} = [w_1, w_2] \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = w_1^2 + w_2^2$
- 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.

① for t in 1 ... max-its

$\vec{w}^t = \vec{w}^{t-1} + \alpha \cdot \vec{d}^{t-1}$

② While $|g(w^t) - g(w^{t+1})| > \delta$

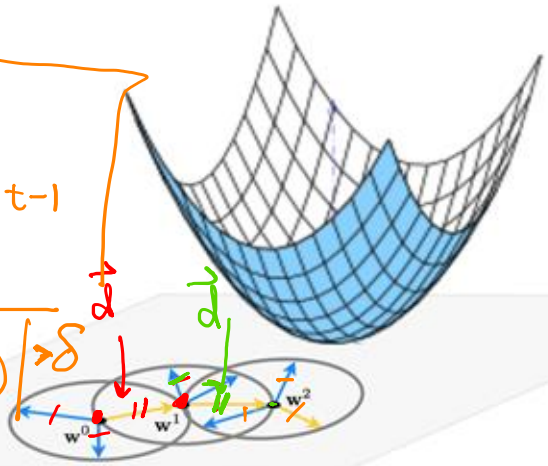
$\vec{w}^t$ . . .

$\|\vec{d}\|_2 = 1$

$\vec{w}' = \vec{w}^0 + \alpha \cdot \vec{d}$

$\vec{w}^2 = \vec{w}' + \alpha \cdot \vec{d}$

$g(w^t) \leq g(w^{t+1})$



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.

Q: How should we generate **d**?

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

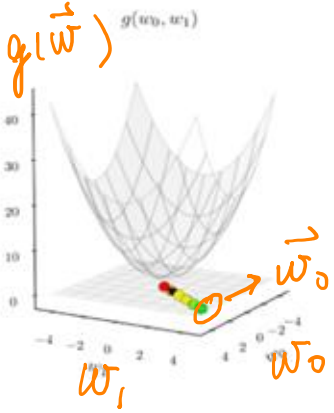
Q: How should we generate **d**?

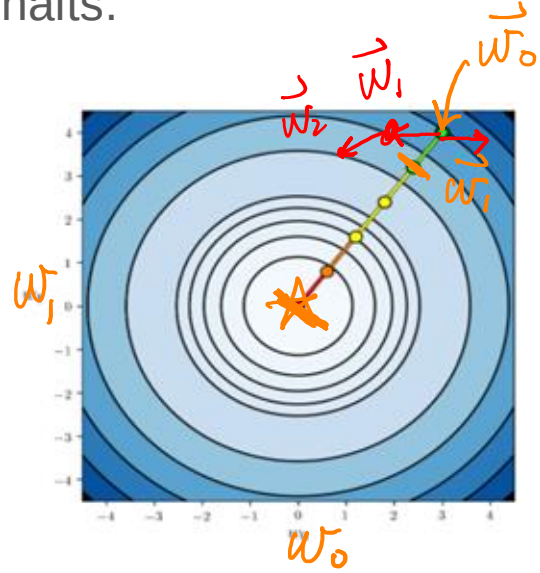
$$s = \underset{p=1\ldots P}{\operatorname{argmin}}\ g\left(\mathbf{w}^{k-1} + \mathbf{d}^p\right)$$

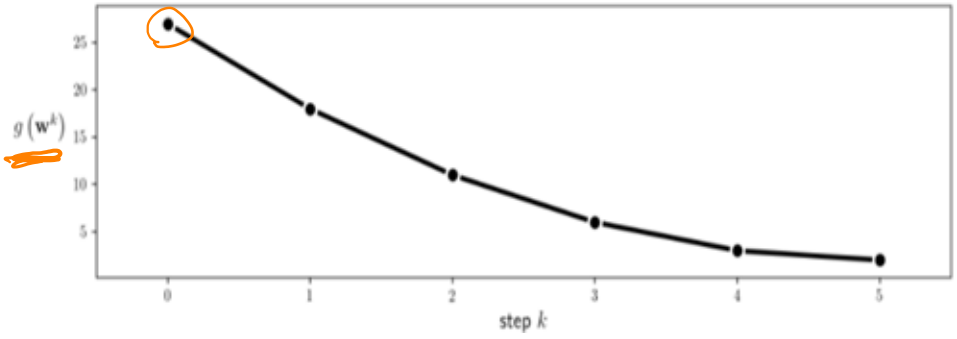
```
# construct set of random unit directions
directions = np.random.randn(num_samples,np.size(w))
norms = np.sqrt(np.sum(directions*directions,axis = 1))[:,np.newaxis]
directions = directions/norms
```

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







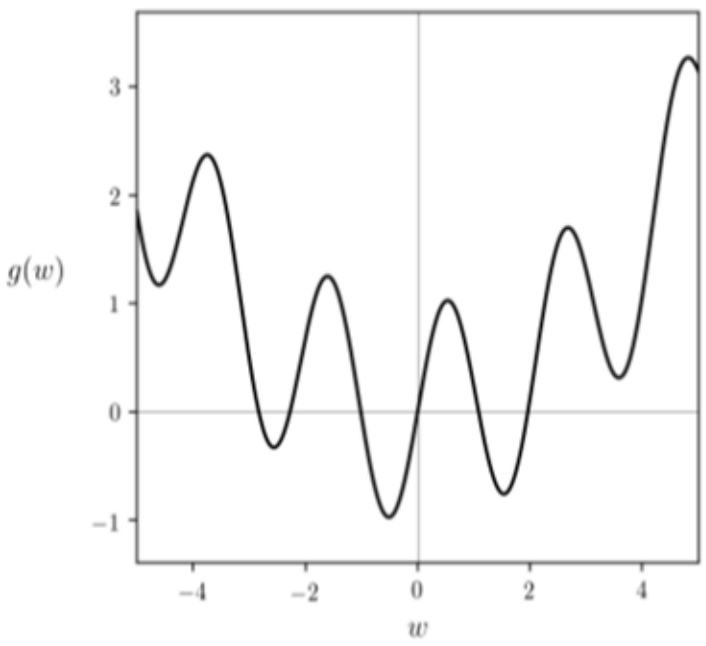
"cost function history plot"

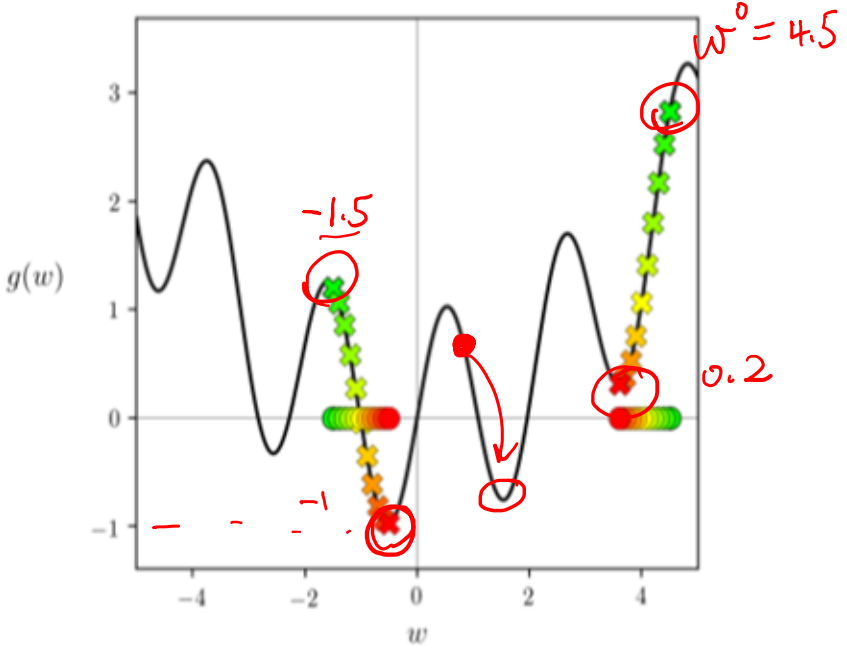
"loss curve"

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

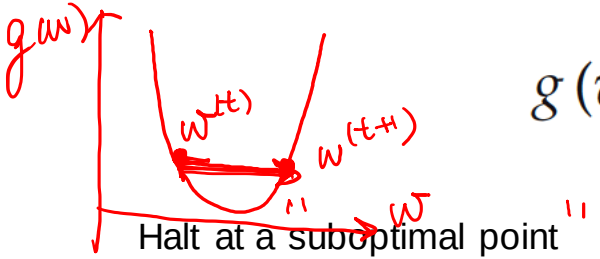
- Find the global minimum of the function. $g(w) = \sin(3w) + 0.1w^2$ using (normalized) random local search.

- 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.



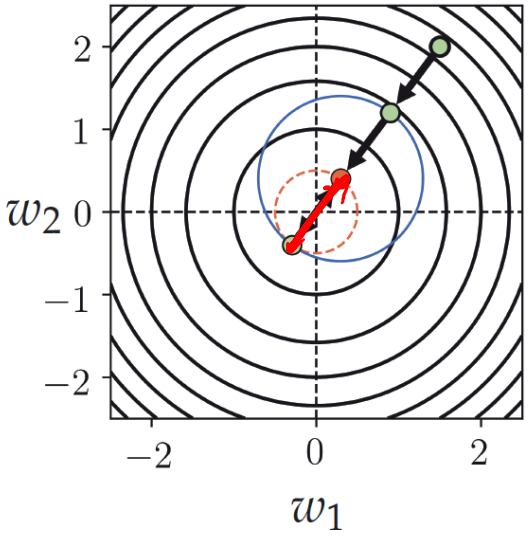


Diminishing Steplength

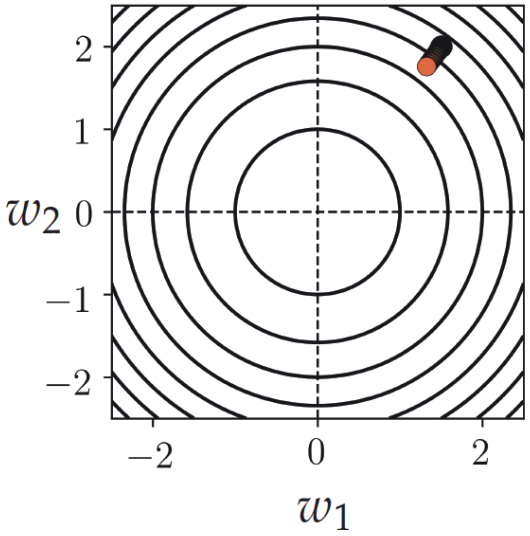


$$g(w_1, w_2) = w_1^2 + w_2^2 + 2$$

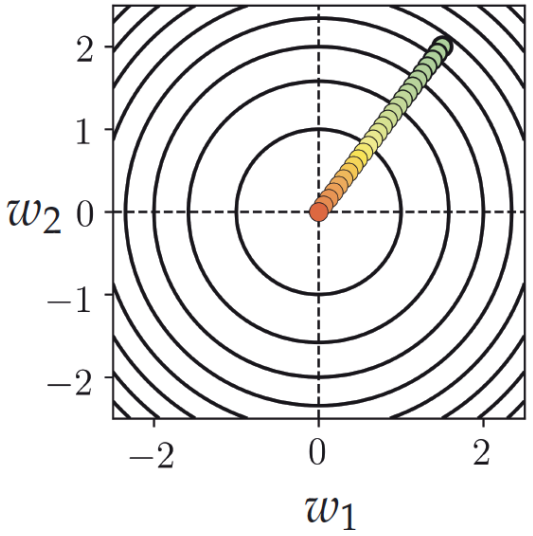
Halt at a suboptimal point



Slow convergence



Proper choice!



Determining **a proper steplength** is crucial to optimal performance with random search – and by extension many local optimization algorithms

- 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

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

- 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.

Q1: Do you know other diminishing steplength rule?

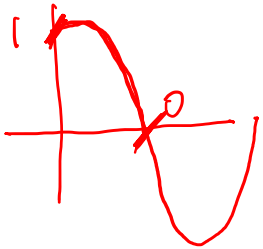
① $\alpha = \begin{cases} 1, & k \in [0, 100] \\ 0.1, & k \in [100, 200] \\ \vdots \end{cases}$ ResNet 2015

② $\alpha = \alpha_0 \cdot e^{-\beta k}$, $0 < e^{-\beta k} < 1$

Module: tf.keras.optimizers.schedules

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules

View aliases



Classes

`class CosineDecay`: A LearningRateSchedule that uses a cosine decay with optional warmup.

`class CosineDecayRestarts`: A LearningRateSchedule that uses a cosine decay schedule with restarts.

`class ExponentialDecay`: A LearningRateSchedule that uses an exponential decay schedule.

`class InverseTimeDecay`: A LearningRateSchedule that uses an inverse time decay schedule.

`class LearningRateSchedule`: The learning rate schedule base class.

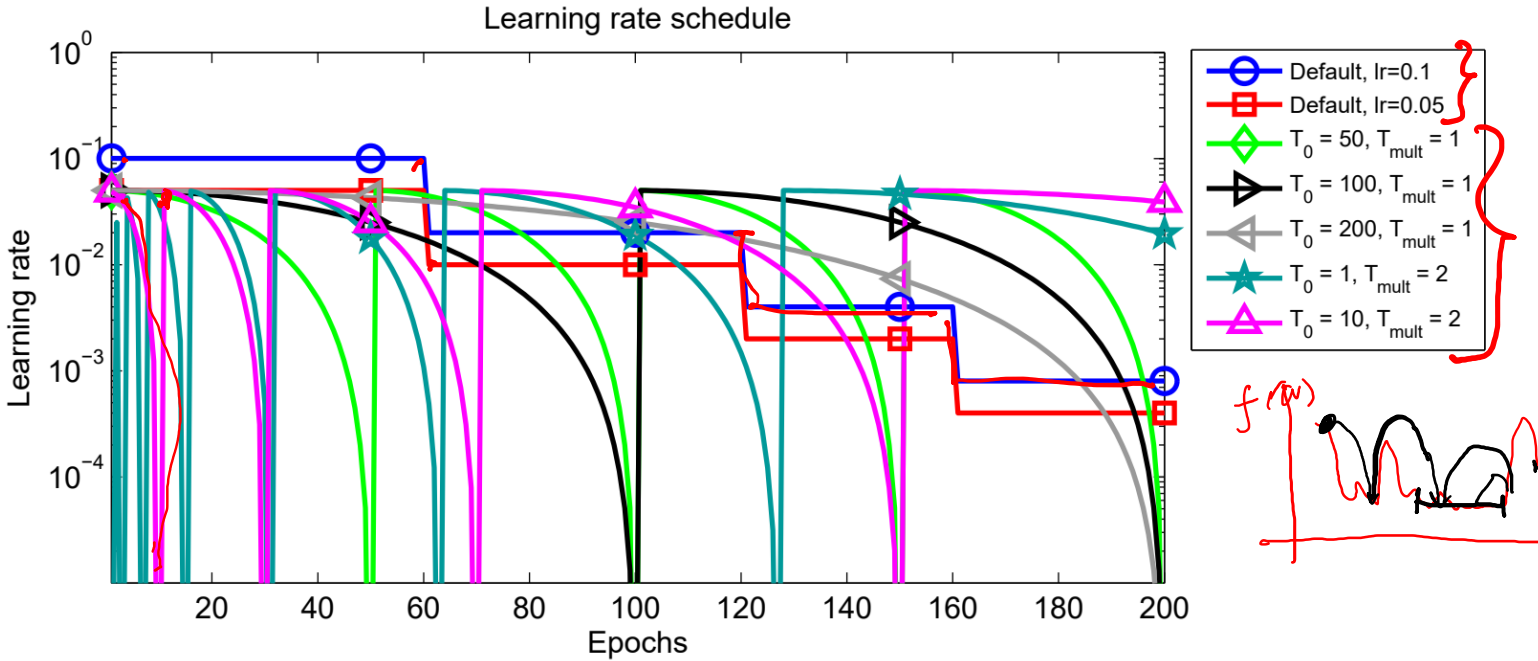
`class PiecewiseConstantDecay`: A LearningRateSchedule that uses a piecewise constant decay schedule.

`class PolynomialDecay`: A LearningRateSchedule that uses a polynomial decay schedule.

TensorFlow > API > TensorFlow v2.13.0 > Python Was this helpful?

Module: tf.keras.optimizers.schedules

Loshchilov & Hutter, ICLR2016, SGDR: Stochastic Gradient Descent with Warm Restarts



Module: tf.keras.optimizers.schedules

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules

$\alpha = 1/k$

```
def decayed_learning_rate(step):
  return initial_learning_rate / (1 + decay_rate * step / decay_step)
```

0.8 K

α_0

`class CosineDecay`: A LearningRateSchedule that uses a cosine decay with optional warmup.

`class CosineDecayRestarts`: A LearningRateSchedule that uses a cosine decay schedule with restarts.

`class ExponentialDecay`: A LearningRateSchedule that uses an exponential decay schedule.

`class InverseTimeDecay`: A LearningRateSchedule that uses an inverse time decay schedule.

`class LearningRateSchedule`: The learning rate schedule base class.

`class PiecewiseConstantDecay`: A LearningRateSchedule that uses a piecewise constant decay schedule.

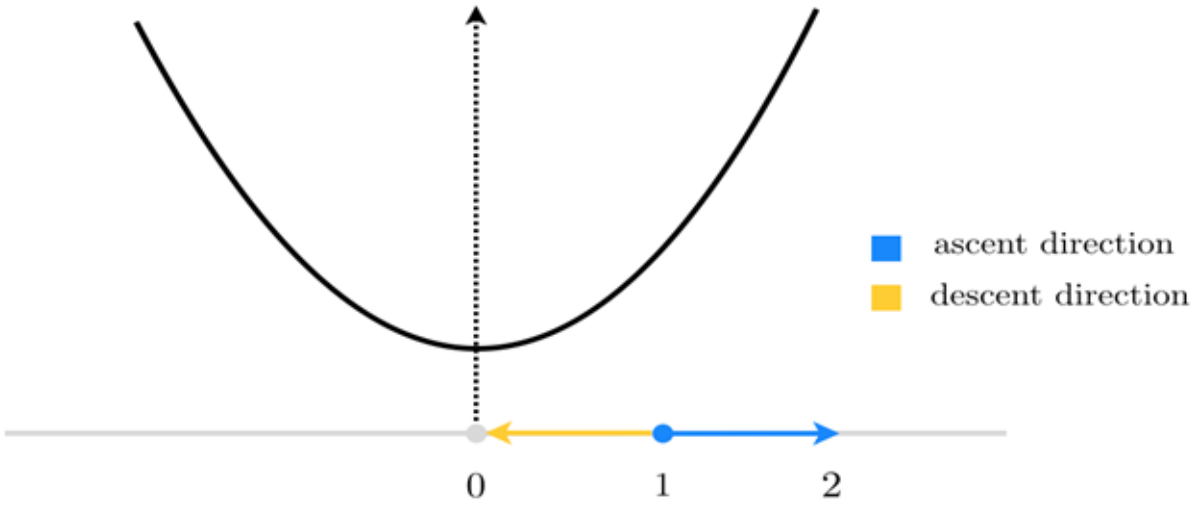
`class PolynomialDecay`: A LearningRateSchedule that uses a polynomial decay schedule.

The curse of dimensionality and random search

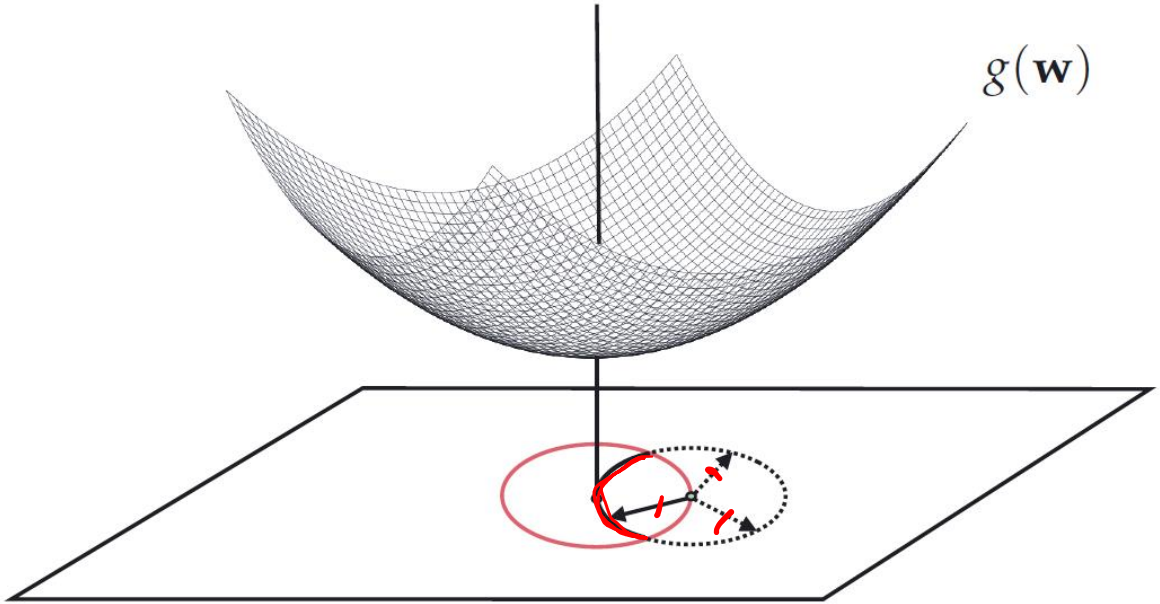
- As with the global optimization approach, the curse of dimensionality also poses a major obstacle to random search as the dimension of a function's input increases.
- We illustrate this using a sequence of simple quadratic functions (where we will gradually increase the input dimension ***N***)

$$g\left(\mathbf{w}\right) = \mathbf{w}^{T}\mathbf{w} + 2$$

- When N = 1, if we decide to choose our direction randomly we will have a $\frac{1}{2} = 50\%$ descent probability. Not too bad!



When N = 2, decent probability < 0.5



- More specifically, in ***N*** we can write

$$\text{descent probability} < \frac{1}{2} \cdot \left(\frac{\sqrt{3}}{2}\right)^{N-1}$$

- So, for instance, when $N = 30$ the descent probability falls below 1%.
- Thus the probability of choosing a descent direction falls ***rapidly*** at each step of random search as the input dimension increases, making the method quite ineffective.

2.6 Coordinate search and descent

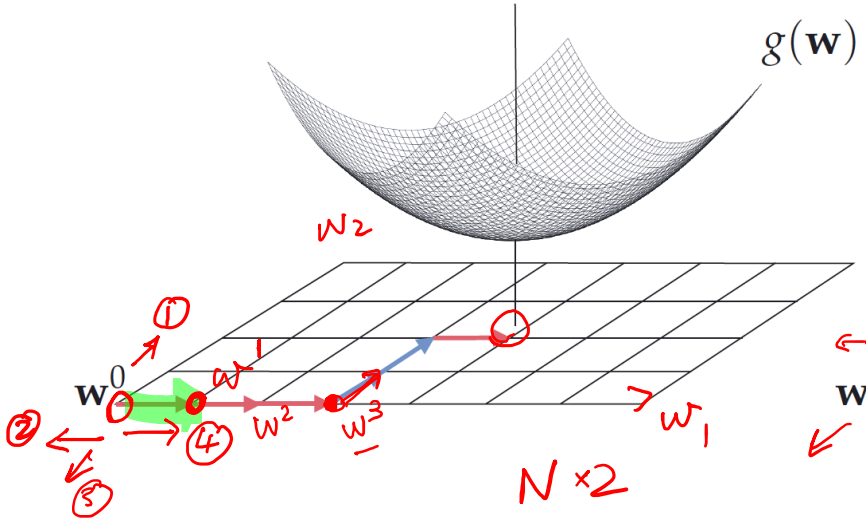
- The coordinate search and descent algorithms
  - zero order local methods that get around the inherent scaling issues of random search
  - **restrict the set of search directions to the coordinate axes of the input space.**
- The concept is simple: random search was designed to minimize a function $g(w_1, w_2, \ldots, w_N)$ 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.

"local"

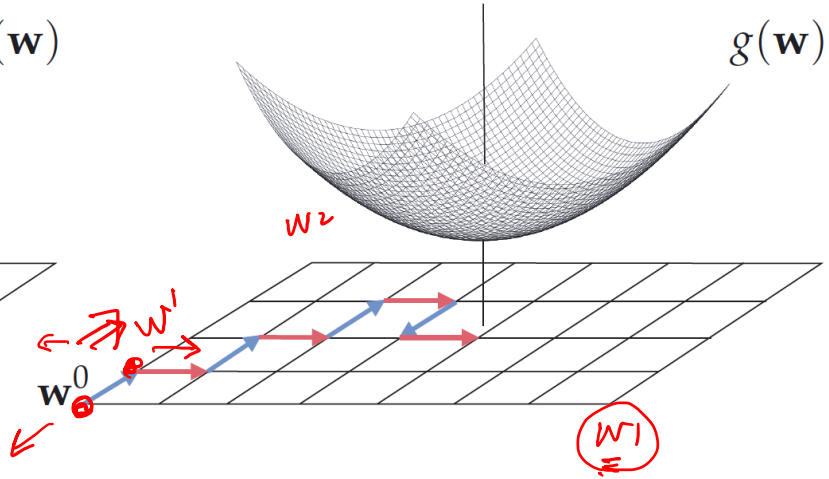
Coordinate search



At each step, try 2N directions and pick the one resulting in the largest decrease in cost

"super-local"

Coordinate descent



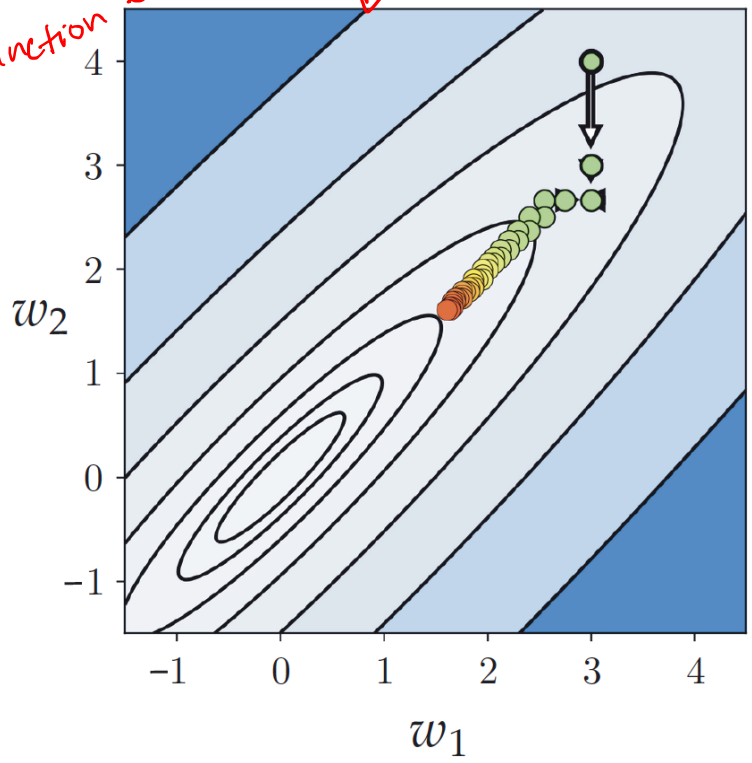
Take a step immediately after examining a coordinate.

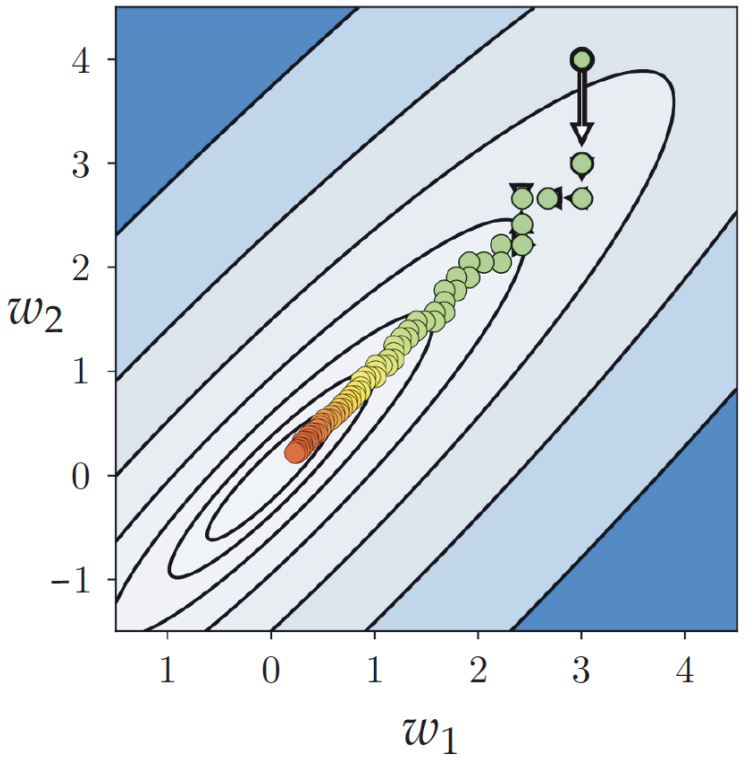
Coordinate search vs. coordinate descent with same number of evaluations.

fixed # function evaluation

coordinate descent makes significantly greater progress

Coordinate descent





Summary

- Mathematical optimization: a function's minima or maxima
- Zero-order condition for optimality
- Global optimization methods
- Local optimization methods
- Random search
- Coordinate search/decent

- Core concepts: decent directions, steplength/learning rates, diminishing steplength schemes, cost function history plots, curse of dimensionality, coordinate search/descent.