3.1 Optimization I

By the end of this notebook you should be able to:

1. Understand basic terminology of optimization
2. Recognise the different optimization problems that arise in machine learning
3. Describe a range of *loss functions* used in regression
4. Understand the following optimization algorithms used to minimize cost functions:
    - *Random Search*
    - *Coordinate Search*
    - *Gradient Descent*

1 Basic Terminology of Optimization

An optimization problem involves finding the vector $\hat{\mathbf{x}}$ that minimizes some function $f(\mathbf{x})$

$$\hat{\mathbf{x}} = \arg\min f(\mathbf{x})$$

1.1 The design vector $\mathbf{x}$

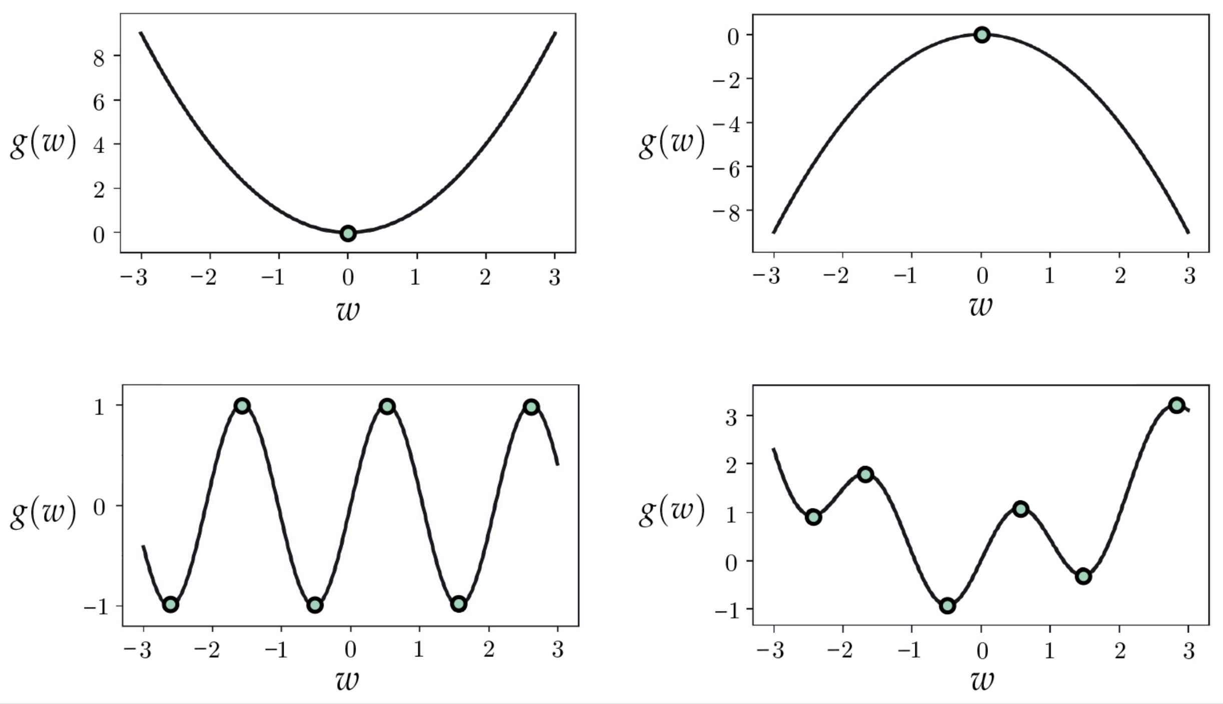
- The vector $\mathbf{x}$ is called the *design vector*
- Often, $\mathbf{x}$ will be a vector of real valued design variables:
    - In *bounded* optimization problems, the real-valued design variables have lower and upper bounds $l_i \leq x_i \leq u_i$.
    - In *unbounded* problems, the real-valued design variables have no bounds $-\infty \leq x_i \leq \infty$.
- However some components of $\mathbf{x}$ may be integer or categorical, depending on the problem

1.2 The objective function $f(\mathbf{x})$

- The function f is called the *objective function*
- Sometimes the problem is to maximize $f(\mathbf{x})$; but this is equivalent to minimizing $-f(\mathbf{x})$
- Sometimes, we will know the first derivatives of f, and possibly the second derivatives too
    - This is an important consideration when choosing an optimization algorithm to optimize f
- Evaluating f may be very computationally expensive (e.g. days to compute), or it may be cheap (< 1s)
    - This is also an important consideration when choosing an optimization algorithm to optimize f
- Sometimes we may have more than one objective, i.e. we may need to minimize $f_i(\mathbf{x}), i = 1, 2, \ldots N$.
    - This is called *multi-objective optimization*. It is not considered much in ML text-books, but is very applicable to ML and is a useful technique to know.

1.3 The solution $\hat{\mathbf{x}}$

- We generally want to find the **global** minimum of f:
    - This is the $\hat{\mathbf{x}}$ such that $f(\hat{\mathbf{x}}) \leq f(\mathbf{x})$ for all $\mathbf{x}$ in our search space
- There may be multiple global minima for f, or there may be just one
    - If there are multiple, then ideally we'd like our algorithm to return them all
- f may have also one or more **local** minima:
    - A point $\mathbf{x}^*$ is a local minimum if $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x}$ near $\mathbf{x}^*$
    - The meaning of *near* is relative: it just means within some neighbourhood around $\mathbf{x}^*$
- When we have multiple objectives, things become a bit different:
    - Generally the objectives 'compete' and we are interested in finding a set of optimal solutions called the *Pareto optimal set* (that represents the best possible trade-off in objectives)



2. Optimization problems in machine learning

There are a number of different optimization problems that arise in machine learning, e.g.:

1. The optimization of model *parameters*, e.g. the weights in linear regression
2. The optimization of model *hyperparameters*, e.g. the choice of λ in regularization, the choice of degree in polynomial regression
3. The optimal selection of a subset of features to train from

Optimization of model parameters

The model that we have focused on so far is the linear regression model:

- Linear model $f(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i$
- Or more generally: $f(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i \phi(x_i)$ where $\phi()$ is a function used to achieve non-linearity (e.g. polynomials, sin function used last week).

A linear model has parameters w_i that controls its behaviour.

During *training*, we search for the weight vector $\hat{\mathbf{w}}$ that minimizes a *cost function* $J(\mathbf{w})$.

- the weight vector $\mathbf{w}$ is our design vector. Often θ is usually used to represent the parameters of a general model, but we will continue with $\mathbf{w}$.
- the cost function J is our objective function to be minimized.

Empirical risk minimization

Typically J is defined as an average *loss* over the training set:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} L(f(\mathbf{x}_i; \mathbf{w}), y_i)$$

where:

- N is the number of examples in the training set

- L is known as a per-example *loss function*; it usually represents the residual

Training a model using a cost function of this form is known as *empirical risk minimization*.

3 Loss functions used in regression

Squared Error (L2 Loss)

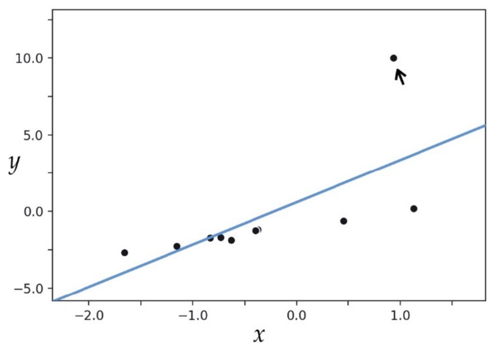
We have already seen the mean squared error cost function

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i;\mathbf{w}))^2$$

which uses squared error as the loss:

$$L(f(\mathbf{x}_i;\mathbf{w}), y_i) = (y_i - f(\mathbf{x}_i;\mathbf{w}))^2$$

Susceptibility of L2 Loss to outliers



One downside of using the L2 loss is that squaring the residual increases the importance of larger errors.

This becomes a particular problem when there are outliers in our training data

The search for optimal weights focuses on finding weights that minimize these large errors (at the expense of fitting to the normal examples). This is like overfitting to outliers in the training data.

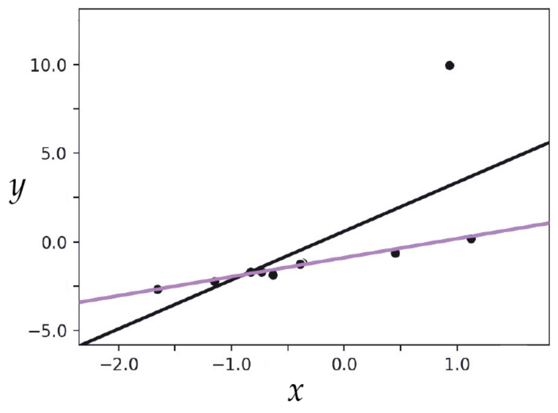
Absolute Error (L1 Loss)

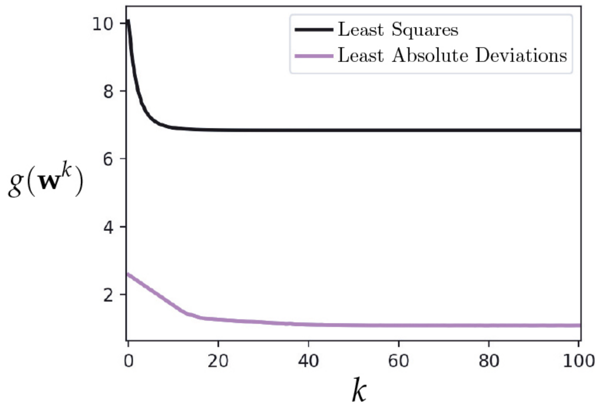
$$L(f(\mathbf{x}_i;\mathbf{w}), y_i) = |(y_i - f(\mathbf{x}_i;\mathbf{w}))|$$

- Gives rise to the Mean Absolute Deviations (MAD) cost function

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}|(y_i - f(\mathbf{x}_i;\mathbf{w}))|$$

- L1 loss is more robust to outliers in the training data as it does not place such emphasis on reducing individual large errors





The L1 loss function poses some difficulties for optimization algorithms:

- its gradient with respect to error (residual) does not really provide much information to optimization algorithms, as it is constant (it is equal to either -1 or 1)
- it is not differentiable at the point where the error is zero

Two loss functions which are robust to outliers, and overcome these limitations of L1 (to different degrees) are the Huber Loss and the Log-Cosh.

One of your lab exercises today is to use Huber Loss in scikit-learn.

Huber Loss

$$L(f(\mathbf{x}_i;\mathbf{w}), y_i) = \begin{cases} \frac{1}{2}(y_i - f(\mathbf{x}_i;\mathbf{w}))^2 & \text{for } |(y_i - f(\mathbf{x}_i;\mathbf{w}))| \leq \delta \\ \delta|(y_i - f(\mathbf{x}_i;\mathbf{w}))| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

where δ is a hyperparameter which can be tuned.

Log-Cosh Loss

$$L(f(\mathbf{x}_i;\mathbf{w}), y_i) = \log(\cosh(y_i - f(\mathbf{x}_i;\mathbf{w})))$$

4 Optimization algorithms for determining optimal weights

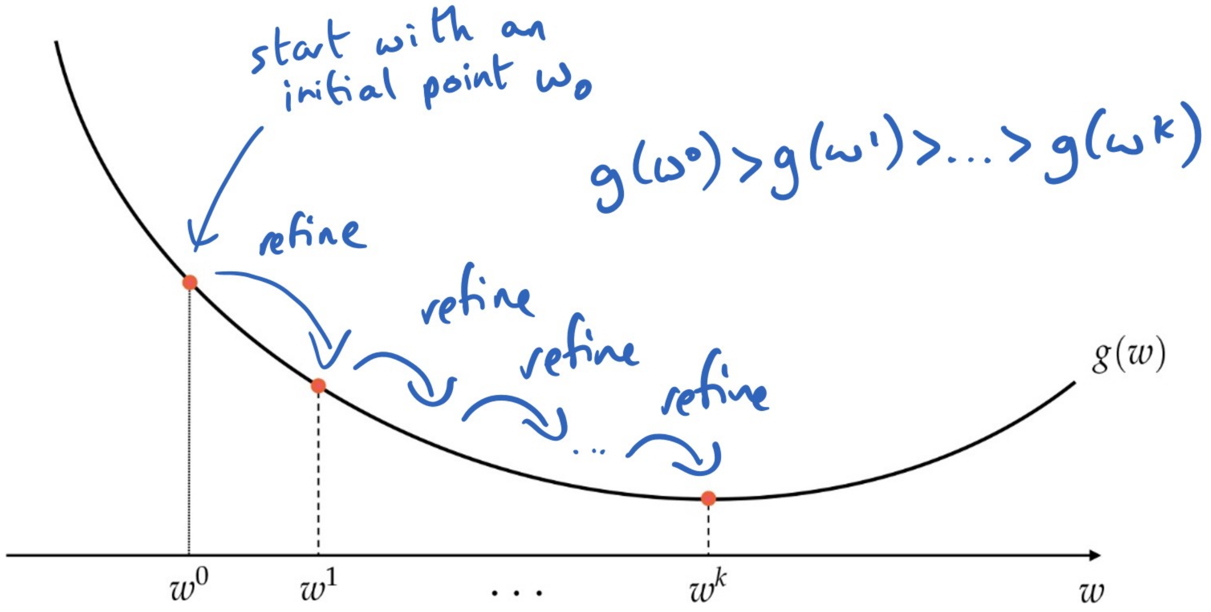
There are many, many optimization algorithms in use for determining optimal model weights (training)

- Some recent review papers are listed in the final lab exercise today for you to check out

We will look at some of the simplest because:

- They are commonly used
- Many of the more sophisticated algorithms build on top of them
- We are limited in time in what we can do!

General idea of local optimization methods

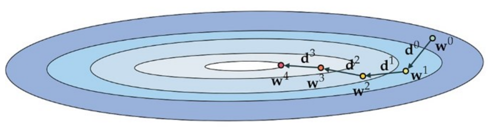


start with an initial point w_0
$g(w^0) > g(w^1) > \ldots > g(w^k)$
refine
refine
refine
refine
...
$g(w)$
w^0
w^1
$\cdots$
w^k
w

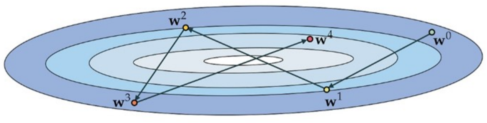
The general framework

- Start with an intial point $\mathbf{w}^0$
- From $\mathbf{w}^0$, determine a *descent direction* $\mathbf{d}^0$ in our design space, to take us to our next point $\mathbf{w}^1$ with lower objective function value, i.e. $g(\mathbf{w}^1) < g(\mathbf{w}^0)$
    - $\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0$
- From $\mathbf{w}^1$, determine a *descent direction* $\mathbf{d}^1$ in our design space, to take us to our next point $\mathbf{w}^2$ with lower objective function value, i.e. $g(\mathbf{w}^2) < g(\mathbf{w}^1)$
- Keep repeating this process: at the k-th iteration we are moving to point $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}$
- Descent directions can be determined in a variety of ways, with or without derivatives
    - This is what distinguishes different local optimization algorithms from one another

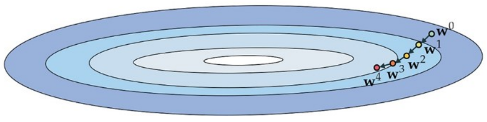
How far to travel in the descent direction?



As well as deciding which *direction* to head in at each iteration, we need to decide *how far* in that direction to go, This is given by the *norm* of the direction vector (square root of the sum of the square of the vector components).



Here, the direction vectors are too large: we oscillate around the minimum, without actually hitting it.



Here, the direction vectors are too small: it will take many steps to reach the minimum.

The steplength parameter

- Because of this potential problem, many local optimization algorithms have something called a *steplength parameter*
    - This is also referred to as *learning rate* in many algorithms
- The steplength parameter, denoted α, allows us to control the length of each update step as follows:

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

- Using the steplength parameter, we can fine tune precisely how far we want to travel in the descent direction at each iteration

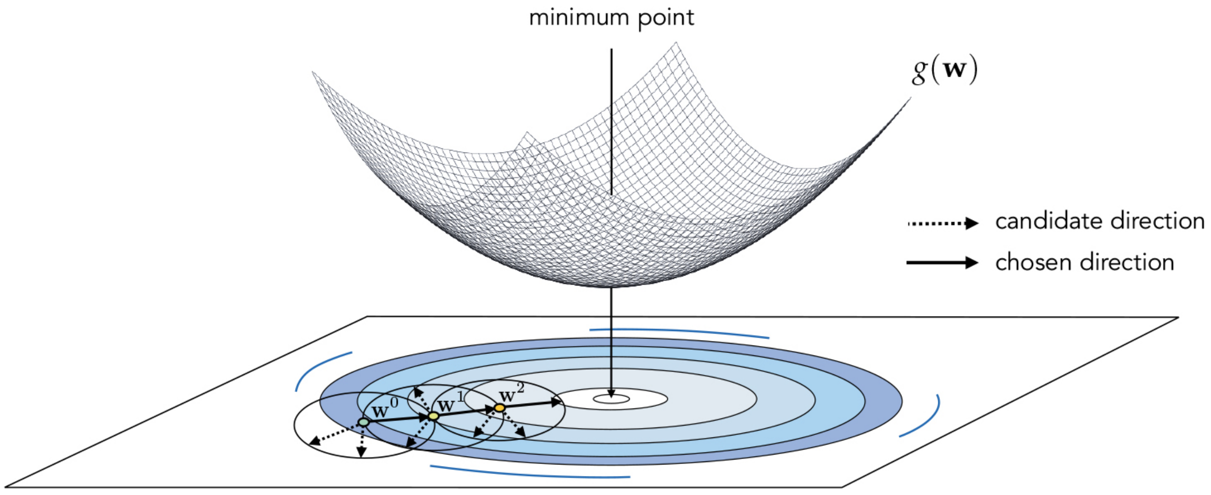
Random Search

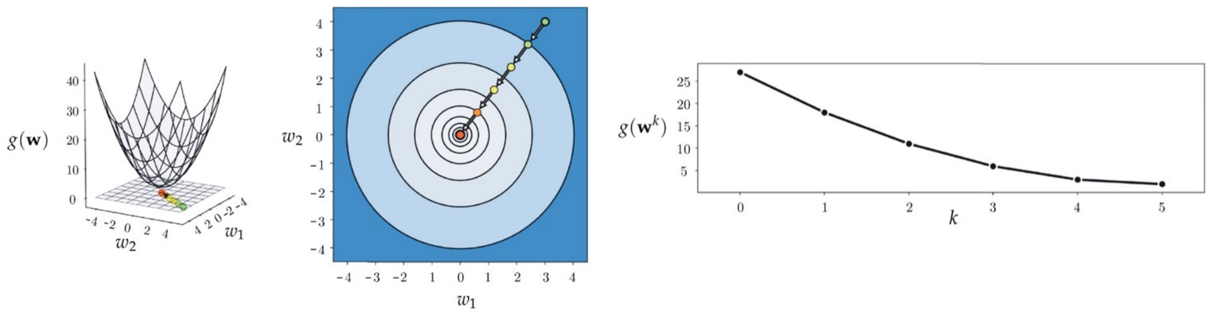
Random search chooses the descent direction from $\mathbf{w}^{k-1}$ by:

- Generating a number P of random directions $\{\mathbf{d}\}_{p=1}^{P}$ to try out from current point $\mathbf{w}^{k-1}$
- This gives rise to candidate points $\mathbf{w}^{k-1} + \mathbf{d}^p$ for $p = 1, 2, \ldots, P$
- Evaluate all of these P candidate points, and find the best (i.e. the one smallest objective function value)
- Suppose the best is $\mathbf{w}^{k-1} + \mathbf{d}^s$
    - If $g(\mathbf{w}^{k-1} + \mathbf{d}^s) < g(\mathbf{w}^{k-1})$ then we move to $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^s$
    - Otherwise we halt the method, or try another batch of P random directions.

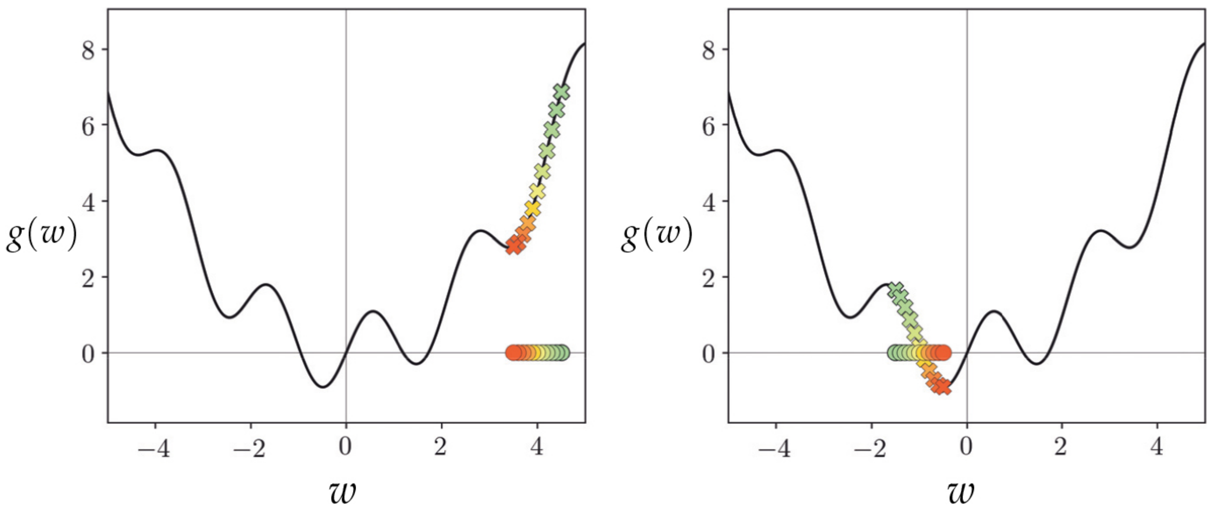
Controlling the length of each step

- When generating the P random directions $\{\mathbf{d}\}_{p=1}^{P}$, these are usually normalized (to have length 1)
- Then, the candidate points above become $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\mathbf{d}^p$ for $p = 1, 2, \ldots, P$



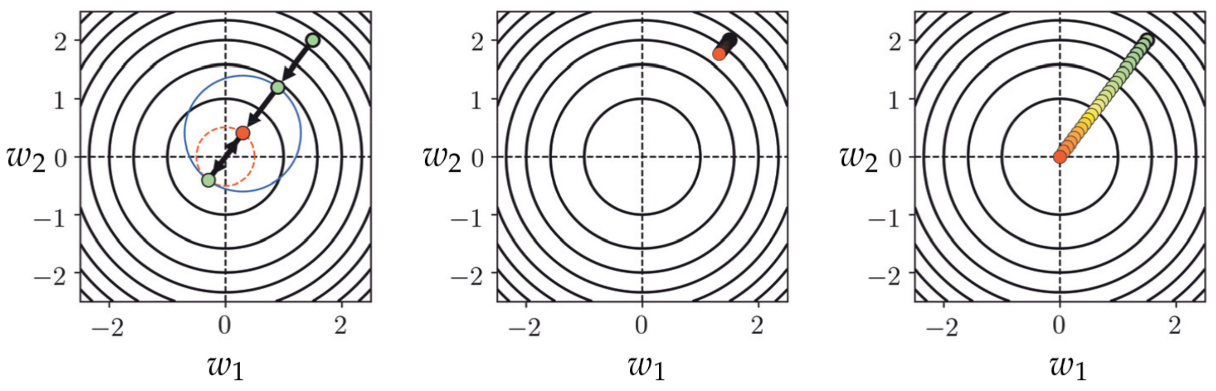


- Visualisation of random search with 5 iterations and $\alpha = 1$ for all steps
- Initial point is in green; final point is in red
- The right hand plot shows a cost function history plot - useful for higher dimensional problems when we can't plot the surface or contour plots



We can attempt *global* optimization using random search (or any other local optimization algorithm), by carying out multiple runs of the algorithm using different initializations.

Failure to converge



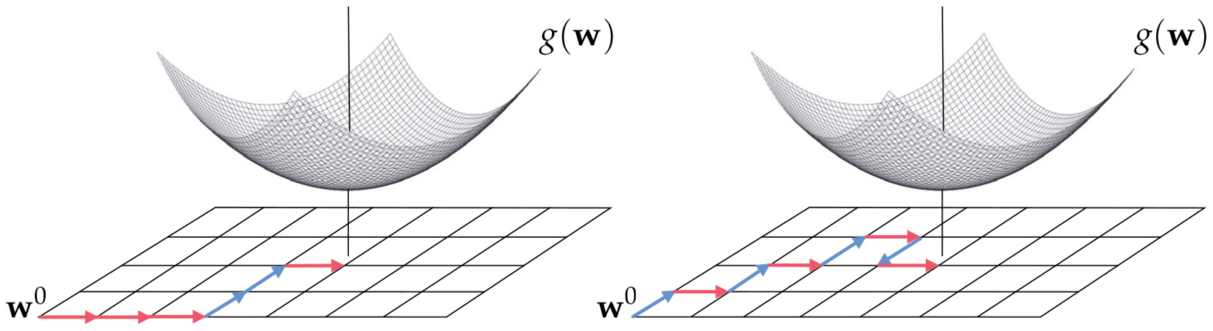
- Left hand figure: using a different starting point to before we arrive at the final sub-optimal point shown in red, for which every direction is ascent, so the algorithm halts.
- Middle figure: If we cut the steplength down to $\alpha = 0.01$, then it takes a long time to move anywhere
- Right hand figure: We have a steplength of $\alpha = 0.1$ for all steps; this converges to a point close to the optimum.

Diminishing steplength rules

- An alternative to using a fixed steplength throughout is to use a *diminishing steplength rule*
- Such a rule shrinks the steplength at each step of local optimization
- For example, a simple rule to shrink the stepength at iteration k would be to set $\alpha = \frac{1}{k}$

Part of your coursework will be to implement a diminishing steplength rule in a random search algorithm

Coordinate search and coordinate descent



- Left-hand figure: **Coordinate search** is like random search, except the descent directions are constrained to be parallel to the coordinate axes of the design space. Therefore, if design space has dimension N, then there are only $2N$ directions to consider at each iteration.
- Right-hand figure: **Coordinate descent** is similar, except at each iteration we simply examine one coordinate direction (and its negative) and step in this direction if it produces descent.

Gradient Descent

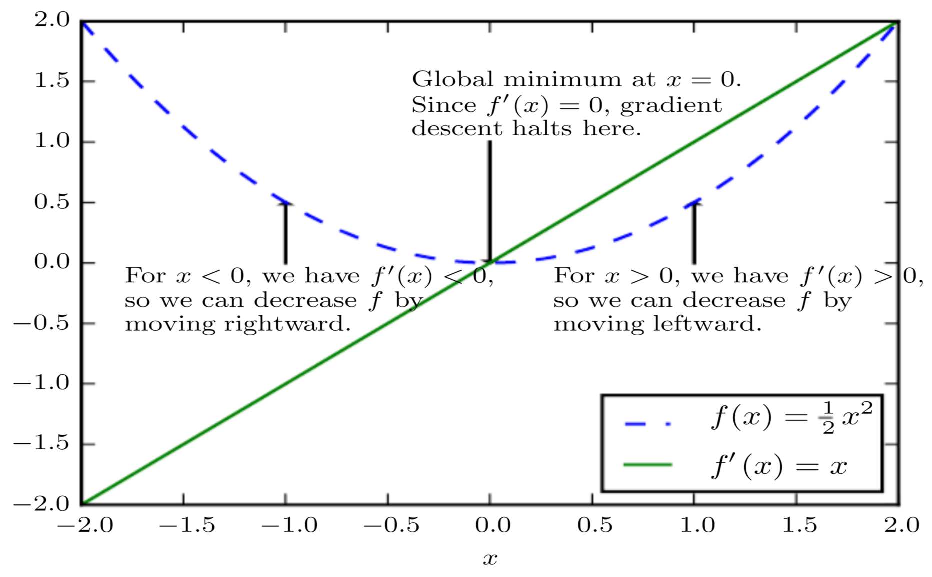
Consider a 1d function $f(x)$, with derivative $f'(x)$. Then $f'(x)$ gives the slope of $f(x)$ at x.

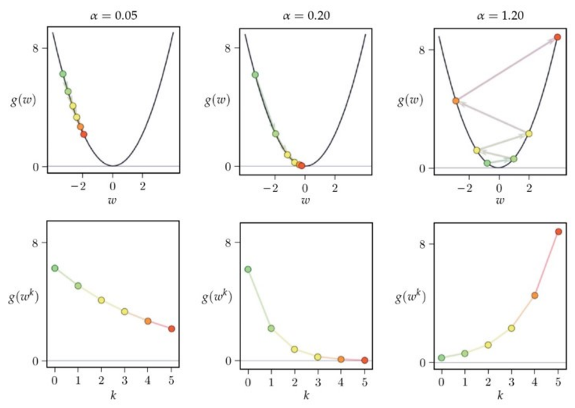
Consider an initial point x^*, and calculate $f(x^*)$ and $f'(x^*)$:

- If $f'(x^*)$ is positive, then:
  - if we increase x slightly (from x^*), then $f(x)$ will increase (from $f(x^*)$)
  - if we decrease x slightly (from x^*), then $f(x)$ will decrease (from $f(x^*)$)
- If $f'(x^*)$ is negative, then:
  - if we increase x slightly (from x^*), then $f(x)$ will decrease (from $f(x^*)$)
  - if we decrease x slightly (from x^*), then $f(x)$ will increase (from $f(x^*)$)

So the first derivative is useful, because it tells us how to change x in order to improve (decrease) $f(x)$

In multi-dimensional problems, this is generalized to the negative gradient $-\nabla g(\mathbf{w})$





Steplength is important in gradient descent. In each of these figures, steplength is fixed.

Left: steplength is too small, and takes a long time to converge to the minimum

Middle: steplength is suitable, and convergence towards the minimum is quick

Right: steplength is too large, and the objective actually gets worse

- Steplength (also known as learning rate) is another example of a hyperparameter
- That is, it is something that:
  - is not optimized during the training of our machine learning model (i.e. by optimizing the cost function)
  - but it is something that we should try to optimize ourselves

We will cover hyperparameter optimizaton next.