Optimization

One common application of analysis relevant to computer science is optimization. These are frequently of use in problems where computer scientists are working with people from other disciplines rather than working on problems that arise out of computer science itself; this includes engineering, science and finance, these days the most obvious application is in data science.

In a typical optimization problem the goal is to minimize some function, E that depends on lots of parameters, so $E(\mathbf{x})$ where

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{1}$$

E is often called an *objective function* or *loss function* and the idea is to find the values of $\mathbf{x}$ that minimize E.

This problem occurs in a diversity of situations. In physics E might be an energy function related to the configuration of a mechanical system with $\mathbf{x}$ representing the positions of the components, or, for a particle physicist, a configuration of fields like the electromagnetic field or its generalizations with $\mathbf{x}$ representing field values. For someone working on machine learning it might measure the error made by a neural network, in this situation E is called an error function and $\mathbf{x}$ are the parameters describing the neural network; the parameters often labelled with ws that describe the connection strengths in the network. In statistics E might be the log-likelihood, a measure of how well a statistical model predicts the data. In neuroscience, the model might be a computational model of a neuron or synapse and $\mathbf{x}$ would correspond to the components of that model, for example the number of each of the different types of ion gates. In this example optimization the model to fit electro-physiological recording will predict the composition of the neuron or synapse.

In any case, the idea is to find the value of $\mathbf{x}$ that gives the lowest value of $E(\mathbf{x})$. The best way of doing this depends on the dimension of $\mathbf{x}$, on how much is known about E, on its properties and how easy it is to calculate. The hardest optimization problems are often ones where E has many local minima so that optimization algorithms end up finding these and missing the global minimum. This isn't a problem we are going to discuss here in detail, but we will note in passing that it is a feature of the loss function for machine learning which is, to some extent, solved by stochastic gradient descent.

Gradient descent

Obviously, if we know the functional form of E we can find its minimum using calculus. For example, say

$$E(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \tag{2}$$

we would calculate the two partial derivatives and then find the optimum point by setting them to zero, so

$$\begin{aligned} 0 &= \frac{\partial E}{\partial x_1} = 2(1 - x_1) - 400x_1(x_2 - x_1^2) \\ 0 &= \frac{\partial E}{\partial x_2} = 200(x_2 - x_1^2) \end{aligned} \tag{3}$$

which is solved by $x_1 = x_2 = 1$. However, this isn't the sort of problem we are thinking about here, we are envisaging situations where E cannot be treated analytically like this, so, for example, E itself must be calculated numerically for a given value of $\mathbf{x}$.

Another approach might be to divide the space of $\mathbf{x}$ values up into a grid and to check them all; this is known as grid-search and is sometimes the best method for problems where $\mathbf{x}$ has only a small number of dimensions and E is very quick to calculate. However, E might be slow to calculate, for example, in neuroscience examples, the model might take quite a while to run. Furthermore, the number of grid points may be very large, particularly if the variability of E means the grid has to be fine, or if the number of dimensions is not small. Often grid search is impractical or out of the question.

Another popular set of methods is related to gradient descent. Roughly speaking this involves working out which direction is down and heading that way. The gradient is a vector pointing in the direction E increases most, the simplest gradient method is to start at an initial value $\mathbf{x}_0$ and iterate

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla E \tag{4}$$

where ∇E is the gradient and η is a, usually small, step size. Now, if the functional form of E is known, the gradient can be worked out using calculus, and even in situations where E has to be calculated numerically it may be possible to find a similar numerical scheme to calculate the gradient. This is often the case for error functions in machine learning, for example. However, it is not always the case that the gradient can be easily estimated; in problems

coming from neuroscience, for example, the models are highly non-linear and there is no easy way to find the gradient.

Even if the gradient is known, gradient descent methods can be tricky. One problem is with the size of η: the scale of the E landscape can change; often if $\mathbf{x}$ is a long way away from the optimal value E is very smooth so it would be useful to take long steps, but near the minimum the features become much finer, so small steps are needed. If η is too small the method spends for ages taking tiny steps down the hill side, if it is too big it steps right over important features at the bottom. Another problem is with thin valleys, if the step size is too large then often successive steps oscillate backwards and forwards across the valley rather than following it down to the minimum. There are solutions to these problems, for example the method called conjugate gradients seeks to choose the successive steps to follow the gradient but to avoid the sort of repetitive backtracking that can happen at thin valleys.

Neural networks

In the case of machine learning, the loss function is usually minimized using stochastic gradient descent. The loss function is typically a measure of how well the true outcome was predicted by the network. A common measure is cross-entropy loss: in a network performing a classification problem the output layer might give probabilities for each of the possible classes and the cross-entropy loss for input $\mathbf{u}$ is

$$L(\mathbf{u};\theta) = -\log p_* - \sum_{i\neq *} \log(1-p_i) \tag{5}$$

where p_* is the probability the network assigns to the correct class and p_i for $i \neq *$ is the probability it assigns to the other classes. Here we have used $\mathbf{u}$ for the input to the neural network, for example the pixel values in an image classification problem, this is potentially confusing since in the machine learning literature we often use $\mathbf{x}$ for inputs, but I didn't want to do that here since above that's the name we give for the parameters we are optimizing; however, to follow machine learning tradition those parameters are called θ here.

We aren't going to discuss why cross entropy is a good choice of loss function or how the logarithms appear, suffice to say the choice is well motivated by information theory. The important thing though is to understand

the loss function depends on parameters, θ that describe the network, these are thought of as connection strengths between nodes and effective describe matrix multiplications that form part of the transformation from input, $\mathbf{u}$, to output, the p_i's that estimate the probability of different classes.

Note, however, that the loss above is only calculated for one input-output pair; the goal, in a sense, would be to minimize the "global loss":

$$L(\theta) = \frac{1}{N} \sum_{\text{all possible inputs}} L(\mathbf{u}; \theta) \tag{6}$$

where N is some sort of mythical number counting all possible inputs. Of course, we don't have all possible input and, in any case, if we knew the pairings of all possible inputs and the corresponding class, we wouldn't need to do machine learning. In practice we have only the training set; we will leave aside considerations of over-training, though they are important, this suggests we should minimize

$$L(\theta, \text{data}) = \frac{1}{\text{n}} \sum_{\text{all data}} \text{L}(\mathbf{u}; \theta) \tag{7}$$

where n is the amount of data. It seems like a good strategy to calculate this on the data set, work out the gradients of $L(\theta, \text{data})$ with respect to the parameters we are lumping together under the name θ and then do gradient descent. In fact, it works better to break the data up into smaller, random, sets called batches and do gradient descent in turn on the loss for each batch:

$$L(\theta, \text{batch}) = \frac{1}{\text{r}} \sum_{\text{data in this batch}} \text{L}(\mathbf{u}; \theta) \tag{8}$$

where r is the batch size. This has two advantages, the first is purely practical, working with batches makes the autograd and matrix calculations easier for working out the gradient, but the second is theoretical, this introduces a bit of noisiness into the gradient calculation; the noise is the "stochastic" in stochastic gradient descent. This noise may make the optmization work better by helping the descent find its ways along narrow valleys and hopping it out of shallow local minima or plateaux.

We don't want to spend too much time about this, except to wrap up by noting that a lot of though has been given in machine learning to the other

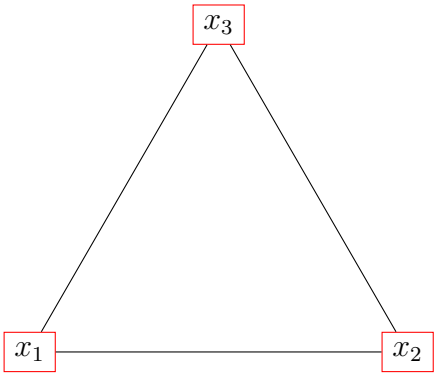

Figure 1: A simplex is a generalization of a triangle to higher numbers of dimensions, but here the 2-dimensional simplex is illustrated and this is just a triangle. A simplex is not regular in general but in many of these figures they are shown as regular to make things look clearer. In the description of the downhill simplex algorithm the vertices are ordered so $E(\mathbf{x}_1) \leq E(\mathbf{x}_2) \leq \ldots E(\mathbf{x}_{n+1})$.

issue we noted with gradient flow, the problem of setting the learning rate. In machine learning there are a collection of algorithms, such as AdaGrad, RMSProp and most famously Adam that set good learning rates for each parameter.

Downhill simplex

The problem of optimization is a rich and complicated one; here we will consider in detail one simple approach: the downhill simplex or Nelder-Mead method. For many problems it will not be the fastest way to find the optimum in terms of run time or the number of function evaluations, but it is often a method that works without having to learn too much about the properties of the objective function; in other words, it is often the quickest way to solve the problem if you include the researchers' time along with the time it takes to run. It also does not require gradient information.

A simplex is a generalization of a triangle, just as a triangle is defined in two-dimensional space and has three vertices and a tetrahedron is defined in three-dimensional space and has four vertices, a n-dimensional simplex has $(n + 1)$-vertices. Here we will be considering simplices in the space

of parameter values $\mathbf{x}$ so the vertices will be $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n+1}\}$. Simplices are not regular in general, the distances between the vertices need not be the same, but it is required that it isn't flat: if you remove one vertex the remaining n vertices should be linearly independent.

Now the idea of downhill simplex is that a simplex kind of ooches around looking for the minimum. It does this, roughly speaking, by taking the worst vertex, the one with the largest value of E, and replacing it with a better one. There are a number of different operations that achieve this, which is used depends on the situation.

For simplicity let's imagine the vertices are in order according to the objective function, so

$$E(\mathbf{x}_1) \leq E(\mathbf{x}_2) \leq \ldots \leq E(\mathbf{x}_{n+1}) \tag{9}$$

This order is done at the start of each iteration, though obvious in practice this doesn't involve renumbering, that is just done here for notational convenience. The idea generally is to replace $\mathbf{x}_{n+1}$. It is useful to define the centroid of the remaining points:

$$\mathbf{x}_o = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \tag{10}$$

Now let $\mathbf{x}_r$ be the point you get to by reflecting $\mathbf{x}_{n+1}$ in the n-dimensional simplex formed by the remaining points. It is

$$\mathbf{x}_r = \mathbf{x}_o + (\mathbf{x}_o - \mathbf{x}_{n+1}) \tag{11}$$

Often this will be a sort of 'reasonable point' so that if it replaced $\mathbf{x}_{n+1}$ it would not be the best nor the worst, that is

$$E(\mathbf{x}_1) \leq E(\mathbf{x}_r) < E(\mathbf{x}_n) \tag{12}$$

In this case use it is used to replace $\mathbf{x}_{n+1}$:

$$\mathbf{x}_{n+1} \leftarrow \mathbf{x}_r \tag{13}$$

This is called reflection and is shown in Fig.2.

There are other cases. The reflected point x_r might be better than all of the existing points, so

$$E(\mathbf{x}_r) < E(\mathbf{x}_1) \tag{14}$$

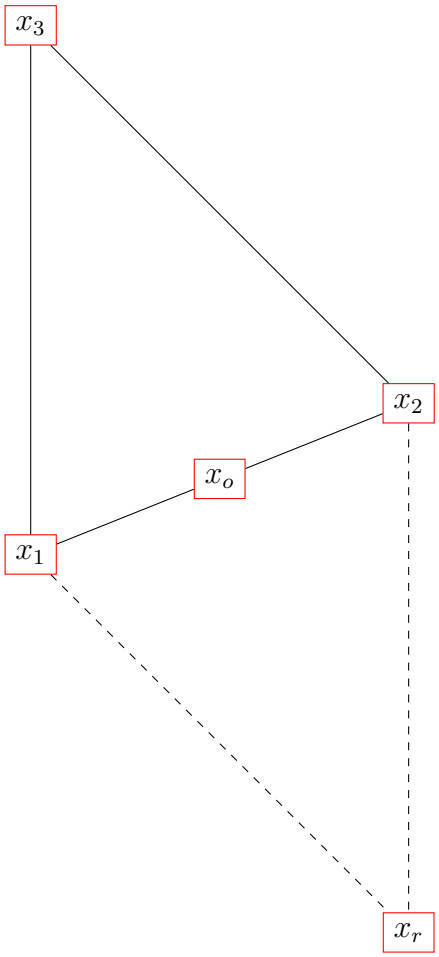

Figure 2: The reflected point is constructed by reflecting the point with the worst value of the objective function through the centroid of the other points.

in which case it is worth trying to go even further in the same direction by considering the so called expansion point $\mathbf{x}_e$:

$$\mathbf{x}_e = \mathbf{x}_o + 2(\mathbf{x}_o - \mathbf{x}_{n+1}) \tag{15}$$

If $E(\mathbf{x}_e) < E(\mathbf{x}_r)$ then

$$\mathbf{x}_{n+1} \leftarrow \mathbf{x}_e \tag{16}$$

otherwise

$$\mathbf{x}_{n+1} \leftarrow \mathbf{x}_r \tag{17}$$

as before. The expansion point is illustrated in Fig. 3. One advantage of expansion is that it makes the simplex bigger; since going from the reflection point to the expansion point gives an even lower value of the objective function it is likely that the relevant features of the landscape are larger than the simplex. This is one advantage of the simplex approach, the size of the simplex changes to match scale of detail in the landscape.

Another possibility is that the reflection point would be the worst point if it replaced $\mathbf{x}_{n+1}$:

$$E(\mathbf{x}_r) > E(\mathbf{x}_n) \tag{18}$$

Replacing x_{n+1} with x_r in this situation would lead to wasteful repetitive moves. It also implies that the simplex is too big, reflecting the worst point causes it to skip over a region around the points $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x}_n\}$ where there are lower values of the objective function. To deal with this a new point is defined, the contraction point

$$\mathbf{x}_c = \mathbf{x}_o - (\mathbf{x}_o - \mathbf{x}_{n+1})/2 \tag{19}$$

If $E(\mathbf{x}_c) < E(\mathbf{x}_{n+1})$ then

$$\mathbf{x}_{n+1} \leftarrow \mathbf{x}_c \tag{20}$$

Finally, if this does not work, there is an emergency move called reduction which shrinks the whole simplex towards the best point. In this move all the points except $\mathbf{x}_1$ are replaced by the point half-way to $\mathbf{x}_1$ so

$$\mathbf{x}_i = (\mathbf{x}_i + \mathbf{x}_1)/2 \tag{21}$$

for $i = 2, 3, \ldots, (n+1)$. Reduction and contraction are both illustrated in Fig. 4 and a flow chart for the whole algorithm is given in Fig. 5.

Obviously after any of these moves the points are reordered, or in fact the best, worst and second worst points identified, and the algorithm repeats.

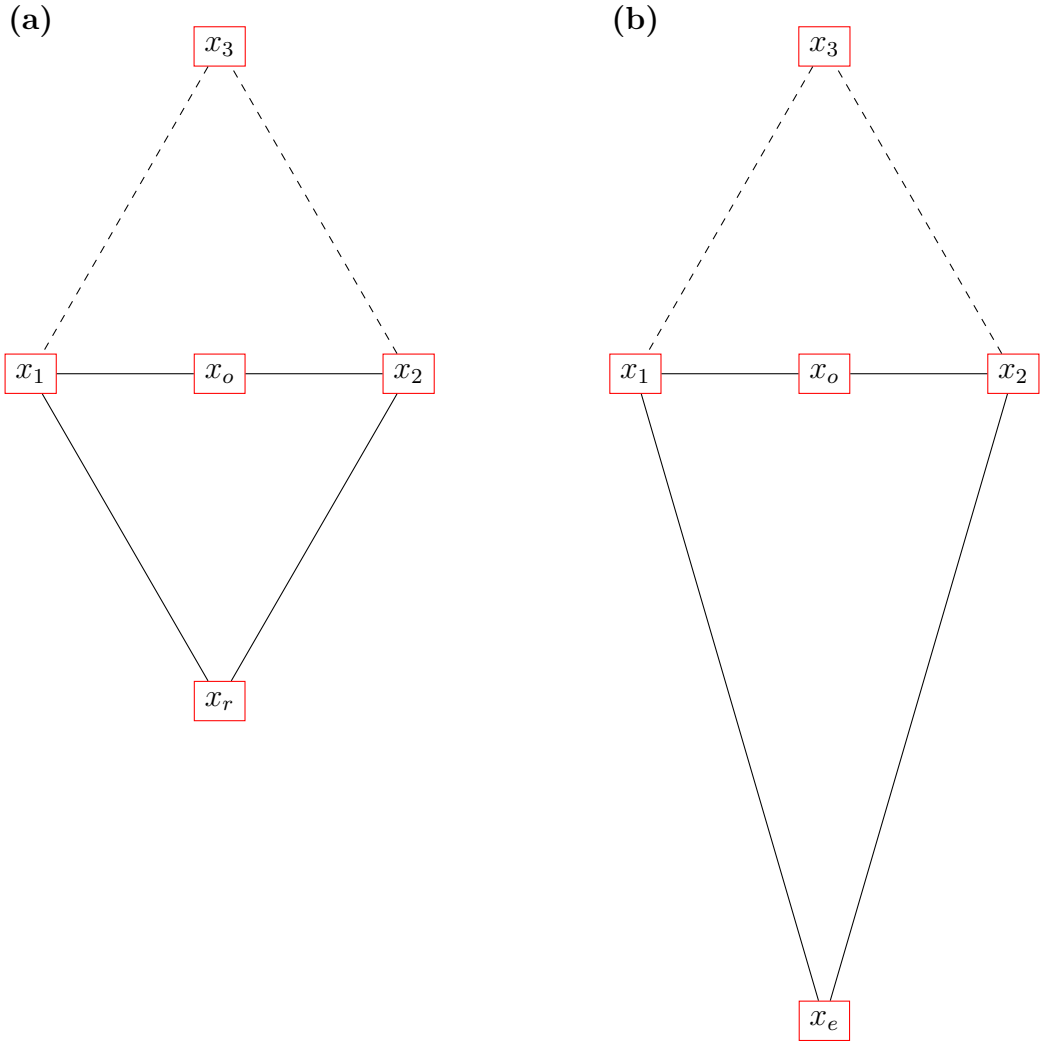

Figure 3: This illustrates reflection **(a)** and expansion **(b)**. In reflection the worst point is reflected through $\mathbf{x}_0$, the center of the remaining points to give a new point $\mathbf{x}_r$. Expansion goes from $\mathbf{x}_0$ to $\mathbf{x}_r$ and continues the same distance again in the same direction to give $\mathbf{x}_e$

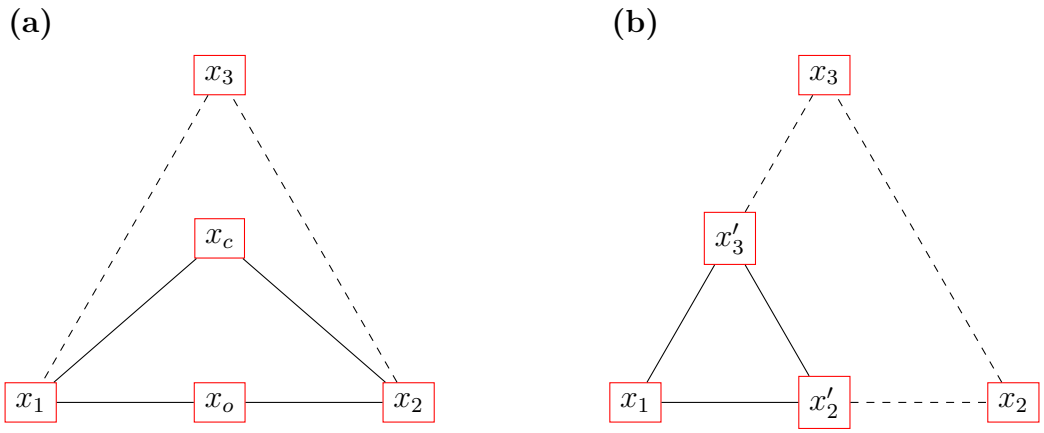

Figure 4: Contraction **(a)** and reduction **(b)**. In contraction a new point $\mathbf{x}_c$ is chosen which is half way between the worst point $\mathbf{x}_{n+1}$ and $\mathbf{x}_0$, the centre of the remaining points. In reduction the triangle is shrunk to half its size while keeping the best point.

However, practically, a stopping criterion needs to be devised. This usually says that the algorithm stops when the simplex vertices have values of the objective function that are extremely close to each other and are unchanging from iteration to iteration, where 'extremely' here stands for some specified tolerance. Often a maximum number of iterations is also specified in case the algorithm doesn't converge.

The downhill simplex is a simple, intuitively appealing and robust algorithm. It is easy to code. It is often the thing that you try to see if you can avoid doing something cleverer and hence trickier. However, there are better algorithms, often the best algorithm that like this that don't use gradient information is Powell's method which involve choosing a direction and optimizating the objective function in that direction, there are very good algorithms for minimizing functions of one variable. Another direction is then chosen and the one-dimensional optimization is repeated, and so on until the algorithm converges. The details of the algorithms account for how the directions are chosen and how the one-dimensional optimization is performed.

A big problem is what happens when there is more than one minimum: the local minima that act as a trap for the simplex on its way to the global minimum. One approach to this is to set the simplex off from lots of differ-

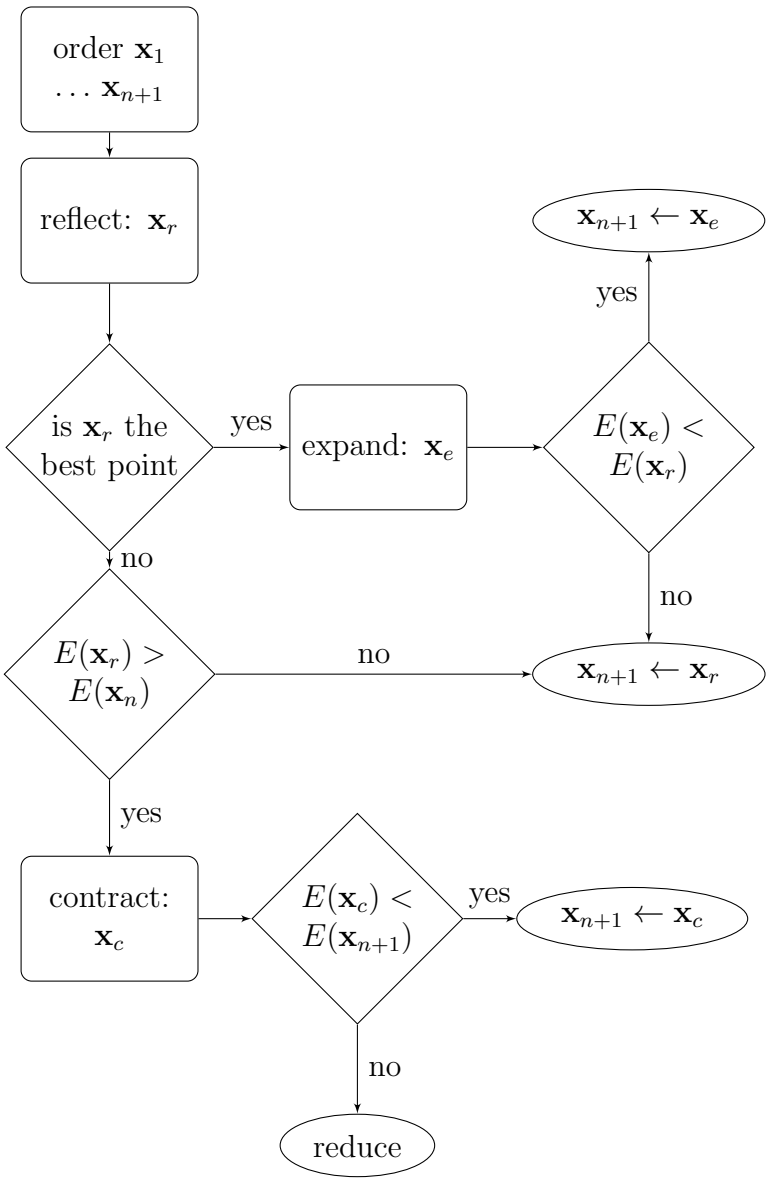

Figure 5: A flow chart for the downhill simplex algorithm. At the start, at the top, the points are put in order so that $\mathbf{x}_{n+1}$ is the worst point and $\mathbf{x}_1$ is the best. Next the reflected point is calculated to give $\mathbf{x}_r$ and $f(\mathbf{x}_r)$ is calculated. By comparing to the $f(\mathbf{x_i})$ it can be decided if $\mathbf{x}_r$ is the best point, that is $f(\mathbf{x}_r) < f(\mathbf{x}_1)$ and the flow chart has two branches depending on the answer. This carries out until an oval is reached, one or more points are changed and the process repeats.
`coms10013.github.io`

ent initial conditions in the hope that one of them will reach the absolute minimum. Another method is to use something like simulated annealing or the genetic algorithm, these are designed to work well when the answer is hiding behind lots of local minima, but tends to be slow and finickity.

Summary

In optimization we find the parameter values, or maybe a, minimum for an objective function. Of this is done using gradient descent:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla E \tag{22}$$

Stochastic gradient descent, for example, forms the basis for deep learning neural networks. For lots of complicated problems were the gradient is not available, downhill simplex works.