Machine Learning - Michaelmas Term 2017
Lectures 8 & 9 : Optimisation

Lecturers: Christoph Haase & Varun Kanade

So far, we've seen that closed-form solutions can be obtained for some learning problems, such as minimising the least-squares objective, or the Ridge Regression objective. Others, such as minimising the Lasso objective or minimising the objective function that is the sum of the absolute values of the residuals have no such closed-form expressions. In fact, for the vast majority of the problems encountered in machine learning it is unlikely that simple closed-form solutions exist. In these instances, it is necessary to resort to general-purpose optimisation methods. The coverage of optimisation methods in this course will be brief and terse; the goal will be to make sure that we understand enough to implement machine learning algorithms. For further details please refer to books on convex optimisation (*e.g.,* Boyd and Vandenberghe (2004)) and non-convex optimisation (*e.g.,* Bertsekas (1999)).

Generally, we can take one of two approaches: We can frame the objective of our machine learning problem as a mathematical program and then use an existing solver for such programs as a blackbox. In this approach, the task is only to rephrase our goal in a framework suited to such blackbox solvers; while it never hurts to know how these blackboxes are implemented, we do not need to understand the actual implementation details. This approach is mostly effective when we can formulate the objective as a convex optimisation program. We'll only focus on problems that can be framed as linear programs, for which efficient algorithms and standard software implementations exist. General purpose convex optimisation programs may often end up being either too slow or an overkill for problems arising in machine learning.[1] For problems that cannot be framed as linear programs, we will usually use gradient-based optimisation methods. These are not *black-box*, in the sense that choosing the correct (hyper)-parameters such as the learning rate, can greatly affect the performance of the trained models and needs to be done carefully.

1 Convex Optimisation

All of the loss functions that we have seen so far in this course are convex. The property of convex functions that is most relevant to us is that every local optimum of a convex function is a global optimum. Finding optimal solutions, i.e., values for the parameters that minimise the loss function, reduces to solving a convex optimisation problem.

Let us make the terminology we use a bit more precise. A set $C \subseteq \mathbb{R}^D$ is *convex* if for any $\mathbf{x}, \mathbf{y} \in C$ and $\lambda \in [0, 1]$,

$$\lambda \cdot \mathbf{x} + (1 - \lambda) \cdot \mathbf{y} \in C$$

This means that whenever we take two points $\mathbf{x}, \mathbf{y} \in C$ and draw a line between $\mathbf{x}$ and $\mathbf{y}$ then all points lying on this line have to be in C. In convex optimisation, we try to find optimal values that lie inside a convex set. To familiarise ourselves with the definition of convex sets, let us take a look at a couple of examples:

- $\mathbb{R}^D$: The whole set $\mathbb{R}^D$ is a convex set, since for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ and $\lambda \in [0, 1]$, we have

$$\lambda \cdot \mathbf{x} + (1 - \lambda) \cdot \mathbf{y} \in \mathbb{R}^D$$

[1] There are many exceptions to this assertion. For instance it is possible to use generic quadratic programming solvers for support vector machines, which we will encounter in due course.

- Intersections of convex sets: Given convex sets $C_1, \ldots, C_n$ and $C := \bigcap_{1 \leq i \leq n} C_i$, for any $\mathbf{x}, \mathbf{y} \in C$ and $\lambda \in [0,1]$ we have $\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y} \in C$ since, by assumption, $\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y} \in C_i$ for all $1 \leq i \leq n$.

- Norm balls: For any L^p-norm $||\cdot||$, the *L^p-ball* of radius one

$$B = \{\mathbf{x} \in \mathbb{R}^D : ||\mathbf{x}|| \leq 1\}$$

  is convex, since for $\mathbf{x}, \mathbf{y} \in B$ and $\lambda \in [0,1]$ we have

$$\begin{aligned} ||\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}|| &\leq ||\lambda \cdot \mathbf{x}|| + ||(1-\lambda) \cdot \mathbf{y}|| \\ &= \lambda \cdot ||\mathbf{x}|| + (1-\lambda) \cdot ||\mathbf{y}|| \\ &\leq 1 \end{aligned}$$

- Polyhedra: Given an $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$, a *polyhedron* is the set

$$P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}\},$$

  where $\leq$ is interpreted component-wise. The set P is convex, since for $\mathbf{x}, \mathbf{y} \in P$ and $\lambda \in [0,1]$ we have

$$\begin{aligned} \mathbf{A} \cdot (\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}) &= \lambda \cdot \mathbf{A} \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{A} \cdot \mathbf{y} \\ &\leq \lambda \cdot \mathbf{b} + (1-\lambda) \cdot \mathbf{b} \\ &= \mathbf{b} \end{aligned}$$

- Positive semi-definite matrices: Recall that a matrix $\mathbf{A} \in \mathbb{R}^D$ is *positive semi-definite* if $\mathbf{A} = \mathbf{A}^\mathsf{T}$ and $\mathbf{x}^\mathsf{T} \cdot \mathbf{A} \cdot \mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{R}^D$. The set of all such matrices is typically denoted by $\mathbb{S}_+^D$ and called the *positive semidefinite cone*. In particular, $\mathbb{S}_+^D$ is convex, since for $\mathbf{A}, \mathbf{B} \in \mathbb{S}_+^D$ and $\lambda \in [0,1]$, we have

$$\mathbf{x}^\mathsf{T} \cdot (\lambda \cdot \mathbf{A} + (1-\lambda) \cdot \mathbf{B}) \cdot \mathbf{x} = \lambda \cdot \mathbf{x}^\mathsf{T} \cdot \mathbf{A} \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{x}^\mathsf{T} \cdot \mathbf{B} \cdot \mathbf{x} \geq 0$$

In addition, the functions that we will be optimising are convex. Formally, a function $f : \mathbb{R}^n \to \mathbb{R}$ defined on a convex domain is convex if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ where f is defined and $0 \leq \lambda \leq 1$,

$$f(\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}) \leq \lambda \cdot f(\mathbf{x}) + (1-\lambda) \cdot f(\mathbf{y})$$

Quadratic functions are probably the most prominent instance of convex functions, but there are many other that occur in the context of machine learning. Here, we take a look at a few:

- Affine functions: An *affine function* is of the form $f(\mathbf{x}) = \mathbf{b}^\mathsf{T} \cdot \mathbf{x} + c$, where $\mathbf{b}$ is a real vector of appropriate dimension, and $c \in \mathbb{R}$. Affine functions define hyperplanes in their ambient space. Let us quickly confirm that affine functions are indeed convex. Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ and $\lambda \in [0,1]$, we have

$$\begin{aligned} f(\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}) &= \mathbf{b}^\mathsf{T} \cdot (\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}) + c \\ &= \lambda \cdot \mathbf{b}^\mathsf{T} \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{b}^\mathsf{T} \cdot \mathbf{y} + c \\ &= \lambda \cdot (\mathbf{b}^\mathsf{T} \cdot \mathbf{x} + c) + (1-\lambda) \cdot (\mathbf{b}^\mathsf{T} \cdot \mathbf{y} + c) \\ &= \lambda \cdot f(\mathbf{x}) + (1-\lambda) \cdot f(\mathbf{y}) \end{aligned}$$

- Quadratic functions: A *quadratic function* is of the form $f(\mathbf{x}) = 1/2 \cdot \mathbf{x}^\mathsf{T} \cdot \mathbf{A} \cdot \mathbf{x} + \mathbf{b}^\mathsf{T} \cdot \mathbf{x} + c$, where $\mathbf{A}$ is a symmetric positive semidefinite matrix. To see that f is convex, note that $\nabla_\mathbf{x}^2 f(\mathbf{x}) = \mathbf{A}$, and since $\mathbf{A}$ is symmetric positive definite it has only non-negative eigenvalues, and hence $f(\mathbf{x})$ is convex. This can also be shown following the same line of reasoning as in the case of affine functions.

- Norms: Any function $f(\mathbf{x}) = ||\mathbf{x}||$ defined by an any L^p-norm $||\cdot||$ is convex. Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ and $\lambda \in [0,1]$,

$$\begin{aligned} f(\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}) &= ||\lambda \cdot \mathbf{x} + (1-\lambda) \cdot \mathbf{y}|| \\ &\leq ||\lambda \cdot \mathbf{x}|| + ||(1-\lambda) \cdot \mathbf{y}|| \\ &= \lambda \cdot ||\mathbf{x}|| + (1-\lambda) \cdot ||\mathbf{y}|| \\ &= \lambda \cdot f(\mathbf{x}) + (1-\lambda) \cdot f(\mathbf{y}) \end{aligned}$$

- Non-negative weighted sums of convex functions: Given convex functions $f_1(\mathbf{x}), \ldots, f_n(\mathbf{x})$ and weights $w_1, \ldots, w_n > 0$, it is an easy exercise to prove that $f(\mathbf{x})$ defined as follows is convex:

$$f(\mathbf{x}) := \sum_{i=1}^{n} f_i(\mathbf{x})$$

The catalogue above gives us a kind of toolbox in order to prove that a given function is convex: if you can decompose it into a non-negative weighted sum of functions listed above, you know that it is convex.

In convex optimisation, we are interested in finding points $\mathbf{x}$ that minimise the value $f(\mathbf{x})$, possibly subject to $\mathbf{x}$ constrained to lie in a convex set. Typically, instances of *convex optimisation problems* are presented as follows:

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) \\ &\text{subject to } g_i(\mathbf{x}) \leq 0 && i \in \{1, \ldots, m\} \\ &\qquad\quad h_j(\mathbf{x}) = 0 && j \in \{1, \ldots, n\} \end{aligned}$$

Here, $f(\mathbf{x}), g_1(\mathbf{x}), \ldots, g_m(\mathbf{x})$ are convex functions, and $h_1(\mathbf{x}), \ldots h_n(\mathbf{x})$ are affine functions. An *optimal value* of a convex optimisation problem is

$$v^* = \min\{f(\mathbf{x}) : g_i(\mathbf{x}) \leq 0, i \in \{1, \ldots, m\}, h_i(\mathbf{x}) = 0, j \in \{0, \ldots, n\}\}.$$

We allow v^* to take values $+\infty$ (in infeasible instances, i.e., where the above set is empty), or $-\infty$ (in unbounded instances, i.e., where the above set has no infimum). Whenever $f(\mathbf{x}^*) = v^*$ then $\mathbf{x}^*$ is a *globally optimal point*, which does not need to be unique. We call $\mathbf{x}$ *locally optimal* if it is feasible (i.e., fulfils all the constraints of the optimisation problem) and there is some $B > 0$ such that $f(\mathbf{x}) \leq f(\mathbf{y})$ for all feasible $\mathbf{y}$ such that $||\mathbf{x} - \mathbf{y}||_2 \leq B$. Note that if $\mathbf{x}$ is locally optimal then B can be chosen arbitrarily small. Of course, all those definitions can be defined in non-convex optimisation settings. What makes convex optimisation problems special is that all locally optimal points are globally optimal. This is not the case for non-convex optimisation problems.

Theorem 1. *In a convex optimisation problem, all locally optimal points are globally optimal.*

Proof. The proof is by contradiction. Suppose there is some locally optimal point $\mathbf{x}$ that is not globally optimal, i.e., there is some $\mathbf{y} \neq \mathbf{x}$ such that $f(\mathbf{y}) < f(\mathbf{x})$. Since $\mathbf{x}$ is locally optimal, we find some B such that $f(\mathbf{x}) \leq f(\mathbf{z})$ for all $\mathbf{z}$ such that $||\mathbf{x} - \mathbf{z}||_2 < B$. Now set $\mathbf{z} = \lambda \mathbf{y} + (1-\lambda) \cdot \mathbf{x}$, where

$$\lambda := \frac{B}{2 \cdot ||\mathbf{x} - \mathbf{y}||_2}.$$

Note that we may assume $\lambda \in (0, 1]$ since B can be chosen arbitrarily small. It is easily checked that $||\mathbf{x} - \mathbf{z}||_2 \leq B$, since

$$\begin{aligned} ||\mathbf{x} - \mathbf{z}||_2 &= ||\mathbf{x} - (\lambda \cdot y + (1-\lambda) \cdot \mathbf{x})||_2 \\ &= ||\lambda \cdot (\mathbf{x} - \mathbf{y})||_2 \\ &= B/2 \end{aligned}$$

But now convexity of f gives us the desired contradiction $f(\mathbf{z}) < f(\mathbf{x})$:

$$\begin{aligned} f(\mathbf{z}) &= f(\lambda \cdot \mathbf{y} + (1-\lambda) \cdot \mathbf{x}) \\ &\leq \lambda \cdot f(\mathbf{y}) + (1-\lambda) \cdot f(\mathbf{x}) \\ &< f(\mathbf{x}) \end{aligned}$$

□

Let us close this section by listing some popular classes of convex optimisation problems.

- Linear Programming (note that $\leq$ and $=$ symbols are meant to be read component-wise):

$$\begin{aligned} \text{minimise } & \mathbf{c}^\mathsf{T} \cdot \mathbf{x} + d \\ \text{subject to } & \mathbf{A} \cdot \mathbf{x} \leq \mathbf{e} \\ & \mathbf{B} \cdot \mathbf{x} = \mathbf{f} \end{aligned}$$

  Linear programs are probably the most popular class of convex optimisation problems. The objective function is linear, given by $\mathbf{c}^\mathsf{T}\mathbf{x}+d$, and there may be inequality and equality constraints. In general, it is unnecessary to allow both inequality and equality constraints, *e.g.,* equality constraints can be replaced by two opposite inequality constraints. However, depending on the exact solver used, details such as allowing explicit equality constraints, constraining variables to be non-negative, *etc.* can affect the running time significantly. A detailed study of the various solvers and the effect of these details on their performance is beyond what we can cover in this course. However, if efficiency is a major concern it is good to be aware of these issues so that they can be looked up before implementation.

  Although there are no closed-form solutions to a linear program, efficient algorithms to solve linear programs exist (both theoretically, a.k.a. polynomial-time and practically, *i.e.,* implementations that can handle thousands or tens of thousands of variables and constraints). If a linear program has an optimum then it is achieved at a vertex of the polytope defined by the linear constraints, cf. Figure 1. (In degenerate cases, it is possible that an entire face of the polytope achieves the minimum value).

  Below, we will see that the absolute loss objective can be encoded as a linear program.

- Quadratically Constrained Quadratic Programming:

$$\begin{aligned} \text{minimise } & \frac{1}{2}\mathbf{x}^\mathsf{T} \cdot \mathbf{B} \cdot \mathbf{x} + \mathbf{c}^\mathsf{T} \cdot \mathbf{x} + d \\ \text{subject to } & \frac{1}{2}\mathbf{x}^\mathsf{T} \cdot \mathbf{Q}_i \cdot \mathbf{x} + \mathbf{r}_i^\mathsf{T} \cdot \mathbf{x} + s_i \leq 0 \qquad\qquad i \in \{1, \ldots, m\} \\ & \mathbf{A} \cdot \mathbf{x} = \mathbf{b} \end{aligned}$$

  Quadratically constrained quadratic programming instances generalise linear programs by allowing for convex quadratic objectives and constraints. Any further details on available solvers and specialised techniques for dealing with them would go beyond the scope of this course.

- Semidefinite Programming:

$$\begin{aligned} \text{minimise } & \operatorname{tr}(\mathbf{C} \cdot \mathbf{X}) \\ \text{subject to } & \operatorname{tr}(\mathbf{A}_i \cdot \mathbf{X}) = b_i \qquad\qquad i \in \{1, \ldots, m\} \\ & \mathbf{X} \text{ positive semidefinite} \end{aligned}$$

  Here, $\operatorname{tr}(\mathbf{A})$ is the *trace* of the matrix $\mathbf{A}$, and C and the A_i are required to be symmetric. It is not immediately seen that semidefinite programs generalise quadratically constraints

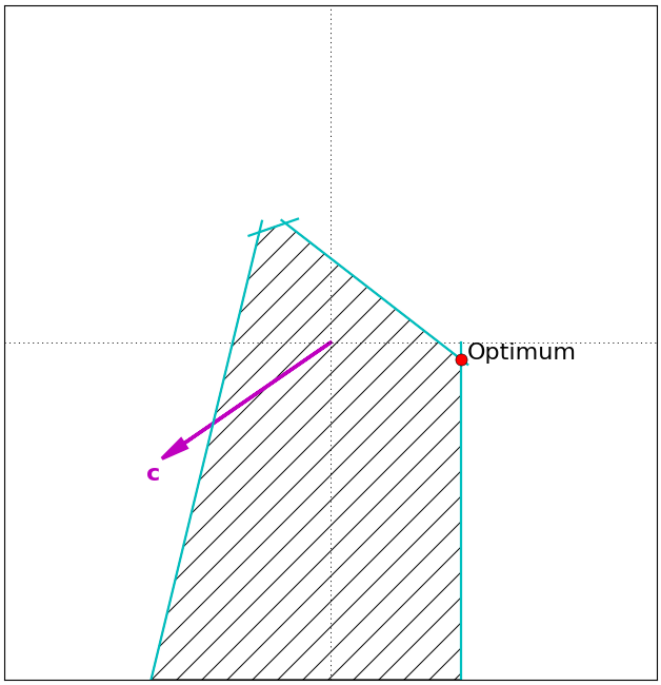


Figure 1: Linear Programming: The constraints define a polytope. If a solution exists and the minimum value achieved is finite, at least one solution lies at a vertex of the polytope.

quadratic programs and hence linear programs. The objective is to find a positive definite matrix $\mathbf{X}$ which has minimal trace when multiplied with C. It is good to have heard about semidefinite programs and how they look like, but we will not cover any further details on them in this course.

1.1 Minimising Absolute Loss Using Linear Programming

Let us now look at one of the objective functions that we encountered in the context of linear regression. In order to avoid the effect of outliers, we proposed minimising the following objective function to train a linear model:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} |\mathbf{x}_i^\mathsf{T}\mathbf{w} - y_i|. \tag{1}$$

As usual, our data is $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^N$, and we wish to train a linear model, $y = \mathbf{w}\cdot\mathbf{x}+\epsilon$. As we are minimising the sum of the absolute values of the residuals, we'll refer to this loss function as *absolute loss*.

It is not immediately obvious how the absolute value in the objective function is to be dealt with using linear constraints or objective. In order to do so, we need to introduce extra variables, beyond the w_i. Consider a linear program with $D+N$ variables, $w_1, \ldots, w_D, \zeta_1, \ldots, \zeta_N$, defined as:

$$\begin{aligned}
&\text{minimise} && \sum_{i=1}^{N} \zeta_i \\
&\text{subject to} \\
&\quad \mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i \leq \zeta_i, && i = 1, \ldots, N && (2)\\
&\quad y_i - \mathbf{w}^\mathsf{T}\mathbf{x}_i \leq \zeta_i, && i = 1, \ldots, N && (3)
\end{aligned}$$

The claim is that the part of the solution of the above linear program, $(w_1, \ldots, w_D)$, is in fact $\mathbf{w}$ that minimises (1). Let's first argue that a solution to this program always exists, let $\mathbf{w} \in \mathbb{R}^D$ be any vector and let $\zeta_i = |\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i|$ for $i = 1, \ldots, N$. It is easy to see that $w_1, \ldots, w_D, \zeta_1, \ldots, \zeta_N$ is a feasible solution (not necessarily optimal) to the linear program. Let $\mathbf{w}^*$ and $\zeta_1^*, \ldots, \zeta_N^*$ be the optimal solution. We argue that it must be the case that $\zeta_i^* = |\mathbf{w}^* \cdot \mathbf{x}_i - y_i|$. Clearly $\zeta_i^* \geq |\mathbf{w}^* \cdot \mathbf{x}_i - y_i|$ because of the constraints (2) and (3). (In fact, this also shows that $\zeta_i \geq 0$.) But, since the linear program minimises the objective function $\sum_i \zeta_i$, it must be the case that $\zeta_i^* = |\mathbf{w}^* \cdot \mathbf{x}_i - y_i|$ for every i. Thus, $\mathbf{w}^*$ must also be the optimal solution to (1).

While some machine learning problems can indeed be posed as linear programs, many cannot. Let's consider the lasso objective for instance:

$$\mathcal{L}_{\text{lasso}}(\mathbf{w}) = \sum_{i=1}^{N} (\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)^2 + \lambda \sum_{i=1}^{D} |w_i|$$

As we've seen there are tricks to convert the absolute value part of the objective into linear inequality constraints. However, there is no way to re-phrase the quadratic part of the objective using linear constraints. Instead, we must resort to more general gradient-based optimisation methods.

2 Review of Multivariate Calculus

Let us briefly review a few concepts from multivariate calculus that we will require in optimisation methods. In order to keep the notation similar to that where we apply these notions, let's refer to our variables by $w_1, \ldots, w_D$, and consider the function $z = f(w_1, \ldots, w_D)$.

The gradient of f with respect to $\mathbf{w}$ is given by:

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \vdots \\ \frac{\partial f}{\partial w_D} \end{bmatrix}$$

We'll always denote the gradient using a column vector. As shown in Figure 2, the gradient is a vector that is orthogonal to the contour curves and points in the direction of the steepest increase. This suggests that in order to minimise a function, we should traverse in the direction opposite to the gradient.

The gradient includes all the first order partial derivatives of the function f. The Hessian is a matrix containing all the second order partial derivatives. Assuming all second order derivatives exist (which will be the case for functions we encounter), the Hessian is symmetric. The Hessian is given by:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \frac{\partial^2 f}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_D} \\ \frac{\partial^2 f}{\partial w_2 \partial w_1} & \frac{\partial^2 f}{\partial w_2^2} & \cdots & \frac{\partial^2 f}{\partial w_2 \partial w_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_D \partial w_1} & \frac{\partial^2 f}{\partial w_D \partial w_2} & \cdots & \frac{\partial^2 f}{\partial w_D^2} \end{bmatrix}$$

While the gradient indicates the direction in which the function increases the most, the Hessian captures the curvature of the function surface at any given point.

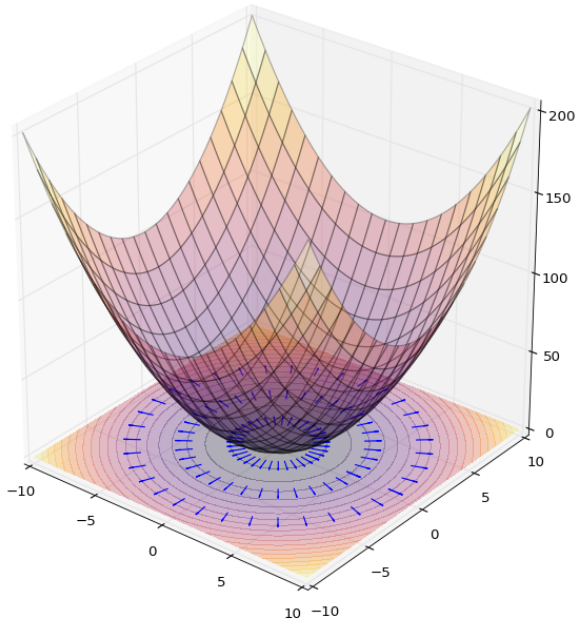
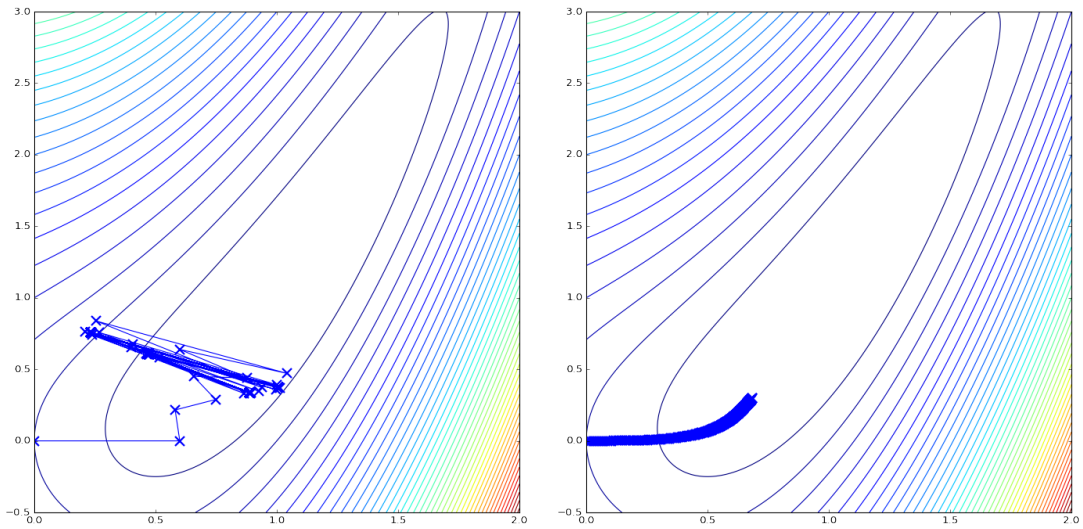
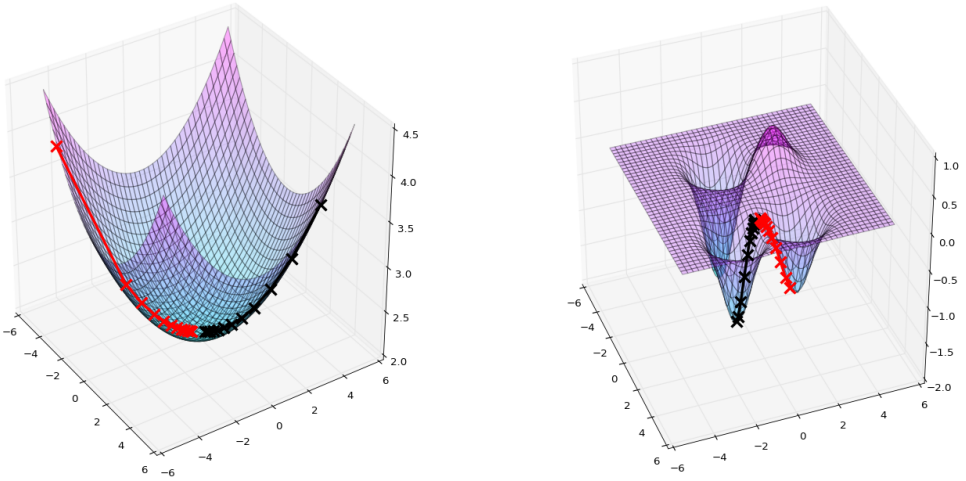


Figure 2: Surface plot, contour curves and gradients for the quadratic function $f(w_1, w_2) = w_1^2 + w_2^2$.



(a) (b)

Figure 3: Figures based on those appearing in (Murphy, 2012, Chap. 8). The function considered is $f(w_1, w_2) = \frac{1}{2}(w_1^2 - w_2)^2 + \frac{1}{2}(w_1 - 1)^2$. (a) With a large step-size the trajectory of the gradient descent algorithm can be quite erratic. (b) With a very small step-size gradient descent is very slow to converge.



(a) (b)

Figure 4: Convergence of gradient descent trajectories starting from different points for (a) a convex function, (b) a non-convex function.

3 The Gradient Descent Algorithm

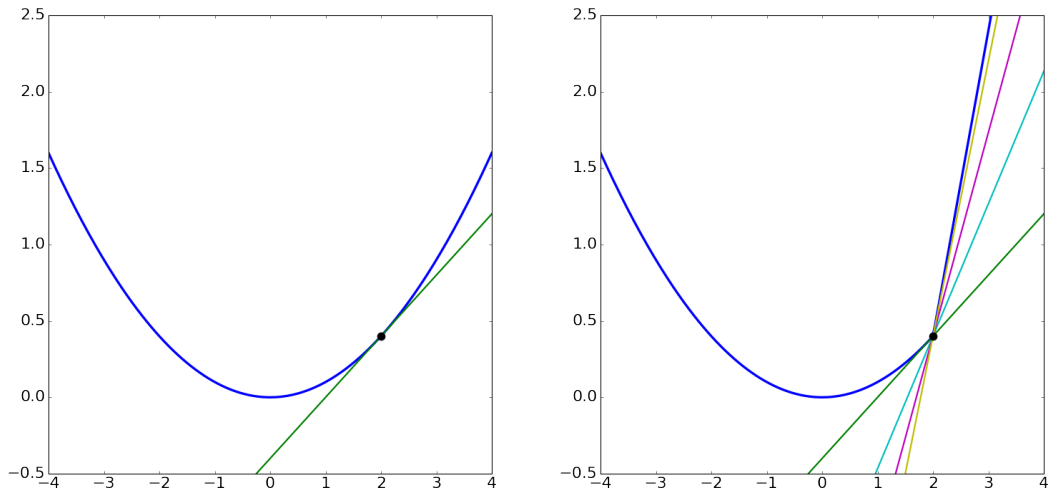
Gradient descent is one of the simplest, but very general and for this reason very powerful algorithm for optimisation. Let $\mathbf{w}_0$ be some starting point. At the heart of gradient descent is the following iterative step:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}_t = \mathbf{w}_t - \eta_t \nabla f(\mathbf{w}_t)$$

In order to not overburden ourselves with ∇s, whenever there is no fear of ambiguity, we'll refer to the gradient simply as $\mathbf{g}_t$. The term η_t in the update above is referred to as a step-size, or in the context of machine learning, the *learning rate*. Choosing the right step-size is important, a step-size that is too large may result in $\mathbf{w}_t$ diverging, whereas a step-size that is too small may result in very slow convergence (or no convergence at all!). This is illustrated in Fig. 3. There are several methods (both theoretically sound and practical heuristics) for choosing the step-size. One such option is called line-search, where the step is taken to the global minimum of the uni-variate function obtained by projecting in the direction of the gradient (refer to the optimisation literature for further details). In machine learning, it is common to let the step-size be a hyperparameter (along with the others) and choose it by cross-validation.

A function f is said to be convex, if for any $\mathbf{w}$, $\mathbf{w}'$, and $\alpha \in [0, 1]$, $f(\alpha\mathbf{w} + (1-\alpha)\mathbf{w}') \leq \alpha f(\mathbf{w}) + (1-\alpha) f(\mathbf{w}')$. Convex functions are important because they are particularly easy to optimise. Furthermore, there are (gradient-based) methods that are guaranteed to converge to a point that achieves the global minimum. This is a reason why an effort is made to choose 'loss' functions that while being useful are also convex.[2] For non-convex functions, gradient-based algorithms may converge to local minima or even saddle points. Figure 4 shows the trajectories of gradient descent on a convex and non-convex function. For convex functions, no matter where the starting point is chosen ultimately the trajectory will converge to a point

[2]In machine learning, it is important to choose a loss function that is useful, in the sense that the value of the loss function should be small if the output is close to what we would consider suitable, *e.g.,* predicting the future value of the pound to within a few pence. However, it is also important to be able to optimise the objective obtained by using this loss function, and hence choosing them to be convex is common practice. There has been somewhat of a break from this practice starting with deep neural networks; for training deep neural networks there are no reasonable convex loss functions and it is common to use methods that find local minima for non-convex functions.



(a) (b)

Figure 5

that is a global minimiser (assuming step-sizes are chosen suitably). On the other hand, if the function is non-convex, even when starting from points that are very close to each other, the trajectories may converge to different points; even worse, the value of the function at the points of convergence may vary wildly.

3.1 Subgradients

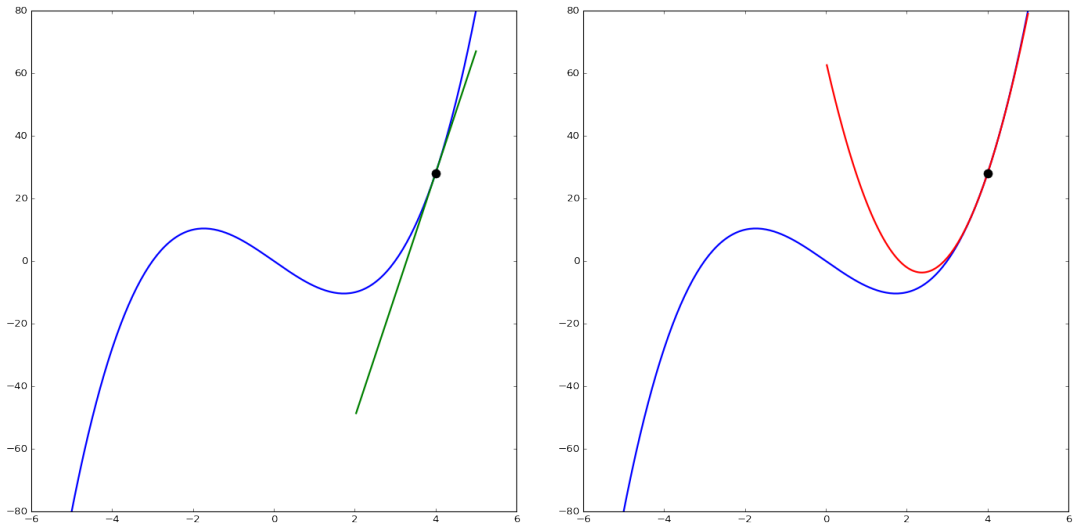
Towards the end of Section 1, we mentioned that linear programming could not be used to minimise the lasso objective:

$$\mathcal{L}_{\text{lasso}}(\mathbf{w}) = \sum_{i=1}^{N} (\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)^2 + \lambda \sum_{i=1}^{D} |w_i|$$

While gradient descent can be applied, there is still the problem that $\mathcal{L}_{\text{lasso}}$ is not differentiable at points where some $w_i = 0$. This turns out not to be a major problem, instead we can use what is called a sub-gradient (or sub-derivative). We'll focus our attention on functions that are convex, where it is relatively easy to define a sub-derivative. For the most part in machine learning, we'll only need to use sub-derivates for two types of functions $f(w) = |w|$ and $f(w) = \max(0, w)$; both these functions are convex. We define a vector $\mathbf{g}$ to be a sub-gradient of f at a point $\mathbf{w}_0$, if it satisfies:

$$f(\mathbf{w}) \geq f(\mathbf{w}_0) + \mathbf{g}^\mathsf{T}(\mathbf{w} - \mathbf{w}_0) \tag{4}$$

Figure 5 shows the derivative and sub-derivatives for a function in one dimension. In one dimension, the vector $\mathbf{g}$ is simply a scalar indicating the slope of the derivative. For f convex, if f is differentiable there is a unique (tangent) line that lies beneath the curve, if f is not differentiable, there may be several such lines and any such slope is a sub-derivative at the point. Similarly in higher dimensions, if f is convex and differentialble, the gradient $\nabla_{\mathbf{w}} f|_{\mathbf{w}_0}$ is the unique vector that satisfies the inequality (4). If f is convex, but not differentiable at $\mathbf{w}_0$, there may be several such vectors $\mathbf{g}$, all of them are subgradients. For the purposes of optimisation, it is fine to choose any of them and take a step in that direction. This is referred to as *sub-gradient descent*. Usually, the term subgradient refers to the entire set of vectors satisfying the inequality (4).



(a) (b)

Figure 6: Function approximations used in gradient-based approaches: (a) Linear approximation (used in vanilla gradient descent), (b) Quadratic approximation (used in Newton's method).

Example 1: If $f(\mathbf{w}) = |w_1| + |w_2| + |w_3| + |w_4|$ for $\mathbf{w} \in \mathbb{R}^4$, then the subgradient at the point $\mathbf{w} = [2, -3, 0, 1]^\mathsf{T}$ is the set $\{[1, -1, \gamma, 1]^\mathsf{T} \mid \gamma \in [-1, 1]\}$.

Example 2: If $f(x) = \max(0, x)$, the sub-derivative of f at $x = 0$ is the interval $[0, 1]$.

4 Newton's Method : Second Order Methods

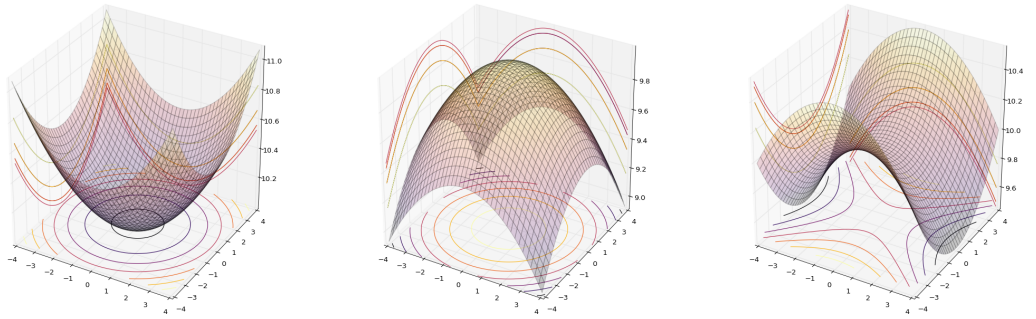
Gradient descent is a very general method for optimisation. However, it only uses the first order partial derivatives, and for this reason can be slower to converge in some instances. We can view the gradient descent approach as approximating the function locally by a linear function, and then minimising the linear function instead. However, unless this linear function is constant, *i.e.,* the gradient is zero, in which case the trajectory is at a stationary point anyway, the minimum of this linear approximation will be negative infinity. For this reason, a suitably small step-size is essential.

Newton's method uses the first and second order partial derivatives of the function. At point $\mathbf{w}_t$ in the trajectory, instead of approximating the function locally by a linear function, it is approximated by a quadratic function using the second degree Taylor approximation (see Fig. 6). The Newton step then directly takes us to the unique stationary point of this quadratic approximation. In higher dimensions, this involves computing and inverting the Hessian. Let $\mathbf{w}_t$ denote the point at the t^{th} iteration of Newton's method and let $\mathbf{g}_t$ and $\mathbf{H}_t$ denote the gradient and Hessian of f at $\mathbf{w}_t$ respectively. The local quadratic approximation to f around $\mathbf{w}_t$ is given by the multivariate form of Taylor's theorem,

$$f_{\text{quad}}(\mathbf{w}) = f(\mathbf{w}_t) + \mathbf{g}_t^\mathsf{T}(\mathbf{w} - \mathbf{w}_t) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_t)^\mathsf{T}\mathbf{H}_t(\mathbf{w} - \mathbf{w}_t)$$

A Newton step directly takes us to the stationary point of this quadratic approximation. We can compute the gradient of f_{quad} and set it to 0 to obtain $\mathbf{w}_{t+1}$. We have

$$\nabla_{\mathbf{w}} f_{\text{quad}} = \mathbf{g}_t + \mathbf{H}_t(\mathbf{w} - \mathbf{w}_t)$$



(a) Local Minimum (b) Local Maximum (c) Saddle Point

Figure 7: Different types of stationary points. The eigenvalues of the Hessian can be used to determine which kind of stationary point it is.

Setting $\nabla_{\mathbf{w}} f_{\text{quad}} = 0$, to get $\mathbf{w}_{t+1}$, we have

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{H}_t^{-1}\mathbf{g}_t$$

On the one hand, each Newton step is computationally quite expensive. It involves computing $D + \binom{D}{2}$ second order partial derivatives and then inverting the Hessian matrix. On the other hand, Newton's method typically converges in significantly fewer iterations compared to gradient descent. Thus, depending on the data-dimension and available computational resources we may wish to use Newton's method instead of gradient descent.

An important consideration to bear in mind is that Newton's method converges to stationary points and not necessarily minima. This can also be the case with gradient descent, however, a Newton step explicitly takes us to the stationary point of the quadratic approximation, which may not even be in a direction that decreases the function. When minimising convex functions, this is not a concern as there are no stationary points which are not global minima. However, it is believed that the lack of success of second-order methods when training deep neural networks is due to the abundance of saddle points in the landscape of the objective function used for training deep neural networks.[3] The Hessian at the stationary point reveals whether we are at a local minimum, local maximum or saddle point (see Fig. 7). If all the eigenvalues of the Hessian (the eigenvalues must be real if the Hessian is symmetric) are positive then we are at a local minimum, if they are all negative we are at a local maximum, if there are both positive and negative eigenvalues then we are at a saddle point. An eigenvalue of exactly 0 corresponds to the degenerate case, in which there is some direction in which the gradient is (locally) unchanging.

5 Optimisation Algorithms in Machine Learning

Let $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^N$ be the data at our disposal. Typically, in machine learning we optimise functions of the form:

$$\mathcal{L}(\mathbf{w}; \mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\mathbf{w}; \mathbf{x}_i, y_i) + \underbrace{\lambda\mathcal{R}(\mathbf{w})}_{\text{Regularisation Term}} \tag{5}$$

[3]This topic is actively researched at the moment. Thus, this should not be taken as a definitive mathematical statement, but an empirical observation. See the discussion in (Goodfellow et al., 2016, Chap 8) for more information.

The gradient of the objective function is,

$$\nabla_{\mathbf{w}}\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\nabla_{\mathbf{w}}\ell(\mathbf{w};\mathbf{x}_i,y_i) + \lambda\nabla_{\mathbf{w}}\mathcal{R}(\mathbf{w})$$

From the form of (5), we see that the objective function that we seek to minimise is composed of many terms, one corresponding to each datapoint and possibly a regularisation term. Similarly, the gradient is also composed of the sum of the individual gradients corresponding to each of these terms. For instance, for Ridge Regression, we have $\ell(\mathbf{w};\mathbf{x}_i,y_i) = (\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)^2$ and $\mathcal{R}(\mathbf{w}) = \mathbf{w}^\mathsf{T}\mathbf{w}$. Thus, the loss function and the gradient for Ridge Regression are expressed as:

$$\mathcal{L}_{\text{ridge}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)^2 + \lambda\mathbf{w}^\mathsf{T}\mathbf{w}$$

$$\nabla_{\mathbf{w}}\mathcal{L}_{\text{ridge}} = \frac{1}{N}\sum_{i=1}^{N}2(\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)\mathbf{x}_i + 2\lambda\mathbf{w}$$

5.1 Stochastic Gradient Descent (SGD)

One advantage of having an objective function that is composed of many individual terms that are added together is that we can try to speed-up the optimisation algorithm by using randomness. Let us consider the following: Pick $i \in \{1,\ldots,N\}$ uniformly at random, and compute the gradient $\mathbf{g}_i = \nabla_{\mathbf{w}}\ell(\mathbf{w};\mathbf{x}_i,y_i)$. What is $\mathbb{E}[\mathbf{g}_i]$? It turns out that this is given by,

$$\mathbb{E}[\mathbf{g}_i] = \frac{1}{N}\sum_{i=1}^{N}\nabla_{\mathbf{w}}\ell(\mathbf{w};\mathbf{x}_i,y_i)$$

So the expectation of $\mathbf{g}_i$ is exactly the same as the entire gradient (except for the regularisation term). Let's make this more concrete with the case of Ridge Regression, where $\ell(\mathbf{w};\mathbf{x}_i,y_i) = (\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)^2$, so $\nabla_{\mathbf{w}}\ell(\mathbf{w};\mathbf{x}_i,y_i) = (\mathbf{w}^\mathsf{T}\mathbf{x}_i - y_i)\mathbf{x}_i$.

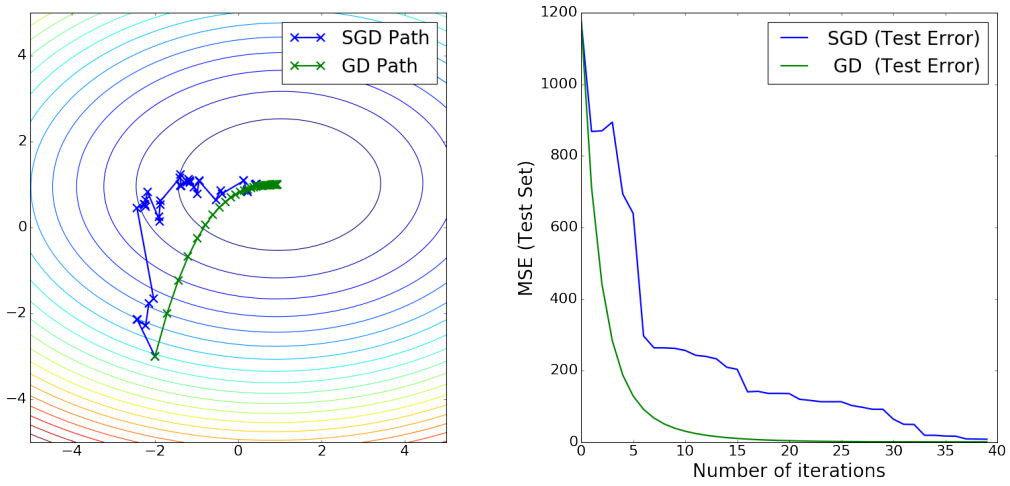
The update rule for the stochastic gradient descent algorithm is the same as that for gradient descent, but instead of computing the (average) gradient using the entire dataset, in SGD a single (random) datapoint and the gradient of the loss function for that datapoint is used. If the objective has a regularisation term, then the gradient of the regulariser also has to be added to the overall gradient.[4]

Figure 8(a) shows the trajectory of gradient descent and stochastic gradient descent for a simple linear regression problem. Figure 8(b) shows the test errors for the resulting model as a function of the number of iterations of gradient descent and stochastic gradient descent. Note that while stochastic gradient descent takes about four times as many iterations as gradient descent, each iteration of gradient descent requires $O(ND)$ time to compute the gradients, whereas in stochastic gradient descent it is only $O(D)$!

Online Learning and Minibatch

Another important advantage of SGD is that it can be implemented in an online fashion. If we don't have all of the data at once, or it needs to be read from disk (which could be slow), we can update the model as we receive the data, rather than wait to collect all the data first. While

[4]Note that the relative scale of the data to regularisation is important. If the original objective had the average loss over the dataset, then it is probably fine to just pick one point, calculate the gradient of the loss function for that point and add to it the gradient of the regularizer. However, if the sum of the loss is used instead of the average, the gradient of the regularizer should be scaled down appropriately when using SGD. In general, it may be necessary to choose the learning rate using some validation techniques.



(a) (b)

Figure 8: (a) Trajectories for gradient descent (GD) and (b) Stochastic gradient descent (SGD) for a simple linear regression problem. (b) Test errors for the resulting models as a function of the number of iterations.

there are theoretical results proving the convergence of stochastic gradient descent, in practice it is found that by *mini-batching* the performance of SGD-like algorithms can be improved significantly. The idea of *mini-batching* is simple, instead of just picking one datapoint as in SGD, we pick b points $B = \{(\mathbf{x}_{i_1}, y_{i_1}), \ldots, (\mathbf{x}_{i_b}, y_{i_b})\}$ and then use the average gradient over the minibatch to take the gradient-descent step. Again, if the original objective had a regularisation term, then the gradient of the regularisation term needs to be added to this. Mini-batching has the advantage of reducing the variance in the gradients and hence it is a bit more stable than (true) SGD.

6 Constrained Convex Optimisation

At times, we may encounter objective functions that need to be optimised over constrained sets. For example, an equivalent formulation of the Ridge Regression objective is:

$$\begin{aligned} &\text{minimise} && (\mathbf{X}\mathbf{w} - \mathbf{y})^\mathsf{T}(\mathbf{X}\mathbf{w} - \mathbf{y}) \\ &\text{subject to :} && \mathbf{w}^\mathsf{T}\mathbf{w} \leq R \end{aligned}$$

The Lasso objective can also be expressed in a similar form, replacing the constraint $\mathbf{w}^\mathsf{T}\mathbf{w} \leq R$ by $\sum_{i=1}^{D} |w_i| \leq R$. We may apply the gradient descent algorithm to such problems, however, even if we start from a point inside the constrained set, the gradient step may take us outside it. The simple solution is to then add a projection step after every gradient step: when the gradient step takes us outside the constrained set, we project to the nearest point that is inside it. For Lasso and Ridge Regression this projection operator has a particularly simple form, however in other instances the projection step can be quite expensive (it can itself be framed as a convex optimisation problem). Discussion of these projection operators is beyond the scope of this course.

7 Discussion

This has been a whirlwind tour of optimisation techniques. We've covered the basics of first-order and second-order methods. However, when actually using these methods in practice, it is often the case that there are "tricks of the trade" that can make a significant difference in the computational resources required to perform the optimisation as well as the quality of the obtained solution, especially when the function being optimised is not convex. To quote Boyd and Vandenberghe (2004), convex optimisation has become a technology, whereas non-convex optimisation is still bit of an art. Thus, these tricks are even more important when it comes to optimising non-convex objective functions, such as those encountered when training deep neural networks. Murphy (2012, Chaps. 8, 13) and Goodfellow et al. (2016, Chap. 8) discuss various extensions to the methods discussed in the lecture in greater detail and are well worth a read.

References

Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Kevin P. Murphy. *Machine Learning : A Probabilistic Perspective*. MIT Press, 2012.